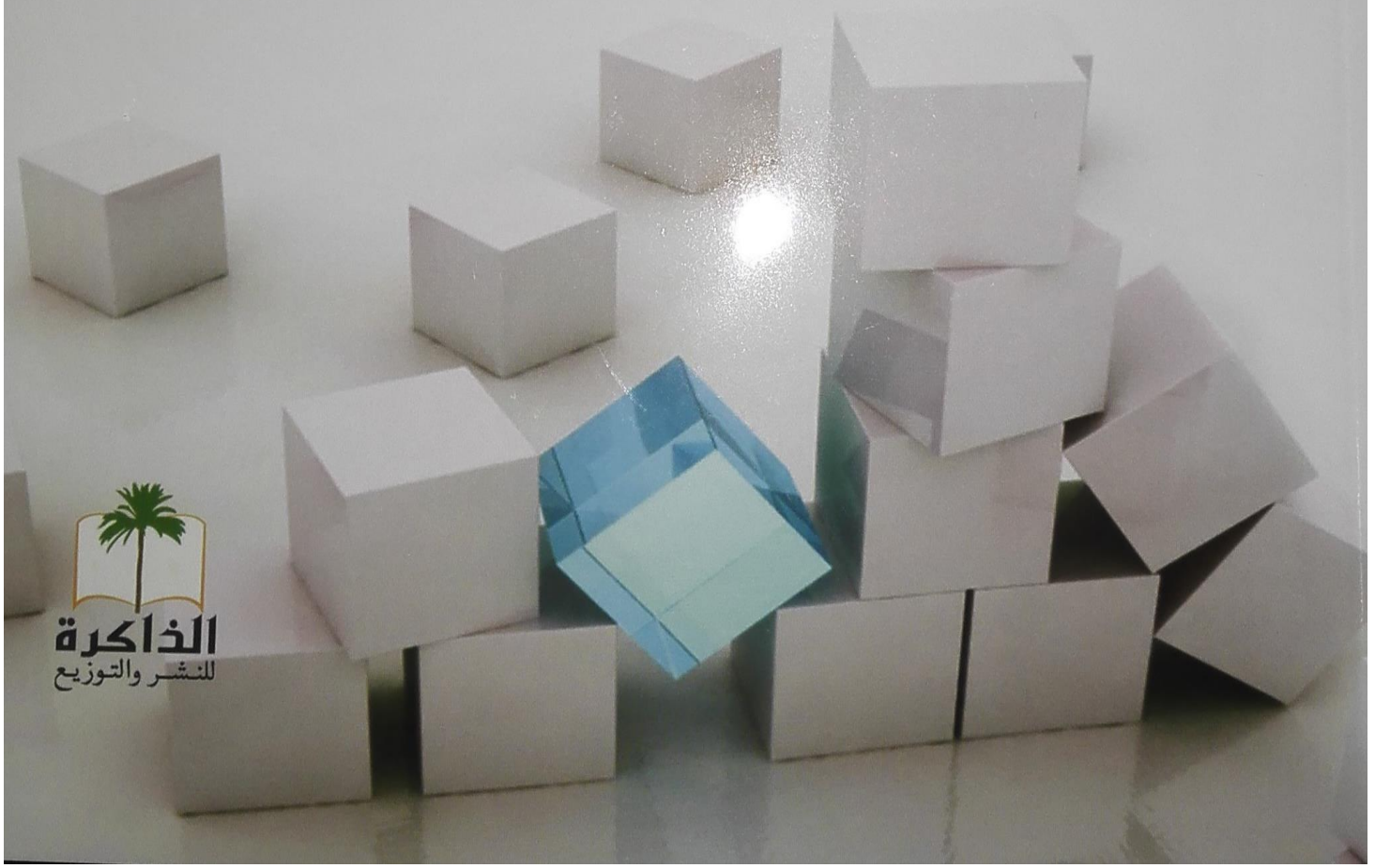
هياكل البيانات

باستخدام C++

الأستاذ الدكتور
نضال خضير العبادي



الذاكرة
للنشر والتوزيع

هياكل البيانات باستخدام C++

تأليف

الأستاذ الدكتور نضال خضير العبادي

هياكل البيانات باستخدام C++

هياكل البيانات باستخدام C++

العبادي، نضال

هياكل البيانات باستخدام C++

رقم الإيداع في دار الكتب والوثائق - بغداد: (2018/609)

رقم التصنيف: 0053.13

الواصفات: : / لغات البرمجة / / الحاسبات الكترونية / / لغة++C

تم إعداد بيانات الفهرسة والتصنيف الأولية من قبل دار الذاكرة للنشر والتوزيع

الطبعة الأولى

2018

بغداد - الصرافية – مجاور الجسر الحديدي

نقال: 07800740728 / 07700488780

بريد إلكتروني: info@althakera.com / www.althakera.com

الى روح والدي ...
الى روح والدتي ...
ما انا عليه هو بفضلكم
بعد الله
أسأل الله ان يغفر لكما

هياكل البيانات باستخدام C++

جدول المحتويات

المقدمة

الحمد لله حمدًا يليق بجلال وجهه وعظيم سلطانه، وأشهد أن لا إله إلا الله وحده لا شريك له، وأشهد أن محمدًا عبد الله ورسوله، صلوات الله وسلامه عليه وعلى آله وأصحابه الطيبين الطاهرين، ومَن سار على نهجه إلى يوم الدين. الحمد لله الذي جعل لنا من العلم نورا نهدي به.

عند الحديث عن اي فرع من فروع الحاسبات الرقمية فلا بد ان نستذكر ان الغاية من اختراع الحاسبات الرقمية الحديثة هو لغرض تسهيل وتسريع العمليات الحسابية المعقدة والتي تستهلك وقت. في غالبية التطبيقات فان قدرة الحاسبات على الخزن والوصول الى كمية المعلومات الكبيرة، فضلا عن سرعة اجراء العمليات الحسابية تلعب دور رئيس وتعتبر الميزة الاولى لها.

دائماً ما نجد أن الأشياء تمتاز بمجموعة صفات، وهذه الصفات تعتبر الخصائص المميزة والمحددة للأشياء، وتنتظم هذه الصفات طبيعياً بشكل بنائي منظم.

نحن نرغب لتجميع العناصر ذات العلاقة معا، ونرغب ايضا لتنظيم هذه البيانات على شكل تكتلات بطريقة مناسبة للبرمجة وكفوءة في التنفيذ.

تاريخيا فان هياكل البيانات تعتبر الدعامة الرئيسية لأقسام علوم الحاسبات. اننا ندرس هياكل البيانات لنتعلم كيفية كتابة برامج حاسوب أكثر كفاءة. على مدى 20 سنة الاخيرة ازداد التركيز على هذا الموضوع وأصبح واسعا بشكل ملحوظ.

واذ نحن نضع بين ايديكم هذا الكتاب الذي نرجو ان يكون في المستوى المطلوب، ونأمل اننا على الاقل لم نقصر ولم نهمل تبيان جواهر عناصر الكتاب.

لقد كانت رحلة جاهده للارتقاء بدرجات العقل ومعراج الافكار، فما هذا الا جهد مقل ولا ندعي فيه الكمال ولكن عذرنا انا بذلنا فيه قصارى جهدنا، فان أصبنا فذاك مرادنا وان أخطأنا فلنا شرف المحاولة والتعلم.

اننا نعتقد ان هياكل البيانات من المواضيع المعقدة نسبيا لذلك تحتاج الى جهد اضافي لتبسيطها وجعلها أسهل للفهم.

الكتاب تم كتابته بطريقة مبسطة لمساعدة طلبة علوم الحاسبات في الدراسة الاولية، وكذلك من الممكن ان يكون مساعد لمحترفي البرمجيات ممن يرغب ان يفهم هياكل البيانات بخطوات مبسطة. تم مراعاة عدم الاسهاب غير المبرر مما يجعل الكتاب يمنح القارئ فهما واسعا حول مفاهيم هياكل البيانات. أتمنى أن أكون موفقا في سردي لعناصر الكتاب سردا لا ملل فيه ولا تقصير موضحا الآثار الإيجابية والسلبية لهذا الموضوع الشائك الممتع.

ان الراغبين بتعلم هياكل البيانات يجب ان تكون لهم المعرفة بلغة البرمجة C او C++ والتي تستخدم كأساس برمجي لكل فقرات وفصول الكتاب.

ان المبرر وراء كتابة هذا الكتاب هو افتقار المكتبة العربية الى كتب عربية احترافية في مجال علوم الحاسبات. لذلك فان الهدف الاولي لكتابة هذا الكتاب هو تغطية التفاصيل والمواضيع الخاصة بموضوع هياكل البيانات بطريقة تساعد على فهم واستنباط المفاهيم الاساسية لهياكل البيانات فضلا عن انها تراعي جميع المستويات. كذلك فان الكتاب يوفر امثلة واشكال توضيحية تساعد على متابعة الموضوع خطوة خطوة.. ناهيك عن تعزيزها بمجموعة من البرامج التي كتبت بلغة البرمجة C++.

يختلف هذا الكتاب عن كتب علوم الحاسبات بشكل عام والتي تكون مملوءة بالنظريات والعلاقات الرياضية، وامثلة ربما تكون من الصعب حلها، حيث تم التركيز في هذا الكتاب على الشرح السلس للتقنيات التي تطبق على مشاكل العالم الحقيقي. تجنبنا البراهين المعقدة والرياضيات الثقيلة. واضيفت الكثير من الاشكال لتوضيح المواضيع المعقدة.

يتكون الكتاب من عدد من الفصول اشير لها هنا باختصار.

الفصل الاول: تم التركيز فيه على بعض المصطلحات العامة وتعريف هياكل البيانات وعلاقتها مع الخوارزميات. وتم توضيح الخوارزميات بشكل مختصر لبيان اهميتها في البرمجة.

الفصل الثاني: ركز على المصفوفات الاحادية والثنائية على اعتبار انها البداية لمواضيع هياكل البيانات، الفصل يحتوي على عدد من الامثلة للتوضيح.

الفصل الثالث: تم التطرق الى خوارزميات الترتيب وقد حاولنا في هذا الفصل تحليل ودراسة عدد من خوارزميات الترتيب بشكل مبسط مع امثلة توضيحية ورسوم توضيحية لغرض المتابعة.

الفصل الرابع: يتناول المكدس بالتفصيل والذي هو من مواضيع هياكل البيانات كثيرة الاستخدام. تم توضيح استخدام المكدس مع التعابير الجبرية وعزز الفصل بكثير من الأمثلة التوضيحية.

الفصل الخامس: نستمر في الفصل الخامس مع الطابور (الخطي والدائري) وكل ما يتعلق بهذا الموضوع من عمليات اضافة وحذف. تم توضيح تطبيقات ومميزات الطابور وعزز الفصل بعدد من الامثلة والبرامج.

الفصل السادس: تناولنا فيه القوائم الموصولة وانواعها الاربع، ونظرا لان هذا الموضوع فيه شيء من التعقيد لذلك حاولنا الاسهاب به قليلا. خوارزميات البحث والحشر والحذف كلها تم مناقشتها مع امثلة ورسوم توضيحية.

الفصل السابع: ناقش موضوع الاستدعاء الذاتي نظرا لاستخداماته الواسعة، حيث ركزنا على اعطاء القارئ فرصة لفهم المفاهيم الاساسية للاستدعاء الذاتي وتم توضيح كيف من الممكن استخدامه في حل المشاكل البرمجية. الخطوط العريضة لكتابة دوال استدعاء ذاتي تم توضيحها باستخدام الامثلة.

الفصل الثامن: تطرقنا فيه الى الاشجار بشكل عام واشجار البحث الثنائي كطريقة لتنظيم البيانات، مع بيان عدد من العمليات الاساسية التي تجرى على اشجار البحث الثنائي. الفصل ايضا تضمن بعض طرق البحث لإعطاء القارئ الفرصة للمقارنة.

الفصل التاسع: تم التطرق الى موضوع التكديس والعمليات التي تجرى عليه نظرا لأهمية الموضوع، تم توضيح العمليات باستخدام امثلة توضيحية معززة بالرسوم.

الفصل العاشر: الفصل العشر هو ختام الكتاب مع موضوع المخططات وكل ما يتعلق بها، تم توضيح المفاهيم والمصطلحات الاساسية المستخدمة مع المخططات وكذلك كيفية تنفيذها وعمليات البحث والحشر والحذف..

في ختام هذه المقدمة فاني لا ازيد على ما قاله عماد الاصفهاني "رأيت انه لا يكتب انسان كتابا في يومه إلا قال في غده لو غير هذا لكان أحسن ولو زيد كذا لكان يستحسن ولو قدم هذا لكان أفضل ولو ترك هذا لكان أجمل وهذا من أعظم العبر وهو دليل على استيلاء النقص على جملة البشر".

وأخيراَ بعد أن تقدمنا باليسير في هذا المجال الواسع آملين أن ينال القبول ويلقى الاستحسان.. وصل اللهم وسلم على سيدنا وحبيبنا محمد وعلى آل بيته اجمعين..

نضال العبادي

النجف الأشرف/ العراق 2018

comp_dep_educ@yahoo.com

المؤلف باختصار:

الاستاذ الدكتور نضال خضير العبادي.. حاصل على شهادة البكالوريوس في الهندسة، بكالوريوس في علوم الحاسبات، ماجستير ودكتوراه في علوم الحاسبات. عمل في مجال الصناعة وعمل في مجال التدريس في عدد من المؤسسات التعليمية. مقوم علمي لعدد من المجلات والمؤتمرات العلمية العربية والعالمية. أشرف على عدد من طلبة الدكتوراه والماجستير.

مؤلف لعدد من الكتب العلمية، وله العديد من البحوث العلمية في مجال اختصاصه. مجال الاختصاص الدقيق هو معالجة الصور والبيانات. حاليا استاذ في جامعة الكوفة – العراق.

هياكل البيانات باستخدام C++

الفصل الاول
هياكل البيانات والخوارزميات
DATA STRUCTURES AND ALGORITHMS

1.1 المقدمة

تستخدم كلمة هيكل وهيكلية في مجالات عديدة لتوصيف أغراض كبيرة تم بنائها من وحدات بنائية صغيرة بأسلوب تكراري، وهياكل البيانات هي كتلة منطقية نجمت عن تكرارية عناصر البيانات وفق ترتيب محدد وعلاقات، وتعتبر هياكل البيانات مرحلة وسيطة بين الملفات على وسائط التخزين وبين برامج التطبيقات.

فمثلاً أن أسماء أفراد العائلة تنتظم تحت بناء يشبه الشجرة "tree", حيث يعبر الجد عن الجذر فيها والأبناء عن الفروع وكذلك الأحفاد وهكذا.

وكمثال آخر ينتظم الناس في طابور لشراء حاجة ما، فيكون الواقف أولاً يمثل رأس الطابور والأخير يمثل ذيل الطابور.

أن دليل التلفون يضم في كل صفحة من صفحاته عمودين أحدهما يدل على أسماء أشخاص معينين، مؤسسات أو هيئات محددة، والعمود الأخر يشتمل على أرقام الهواتف التي يملكها الأفراد أو المؤسسات أو الهيئات الموجودة في العمود الأول.

ويكتمل هذا التشكيل المنظم بترتيب العمود الأول أبجدياً وذلك لتسهيل عملية البحث عن فرد أو مؤسسة أو هيئة معينة.

كل هذه الأمثلة تبين لنا حقيقة واحدة، هي أن المعلومات تمتاز بالترتيب، هذا الترتيب يعرف بالهيكل أو هياكل البيانات "data structure".

مصطلح هياكل البيانات يستخدم لوصف طريقة خزن البيانات، ومصطلح الخوارزمية يستخدم لوصف طريقة معالجة البيانات. هياكل البيانات والخوارزميات متعلقة واحدة مع الاخرى. ان اختيار هيكل بياني معين سيؤثر على نوع الخوارزمية التي ربما تستخدمها، واختيار الخوارزمية تؤثر على هياكل البيانات المستخدمة.

الخوارزمية هي عدد محدد من الايعازات، كل ايعاز له معنى واضح ومن الممكن انجازه بكمية جهد له حدود، وبطول وقت له حدود ايضا. بغض النظر عن قيم المدخلات التي تستخدم، فان الخوارزمية تنتهي بعد تنفيذ عدد محدد من الايعازات.

1.2 المعلومات والبيانات

تعرف البيانات على أنها مجموعة من الحقائق والأفكار التي لم يتم معالجتها، وتعرف بأنها المادة الخام للمعلومات، وحتى تصبح البيانات معلومات لابد أن تعالج هذه البيانات بطريقة معينة لتعطي بيانات ذات فائدة معينة تسمى بالمعلومات، تمتاز المعلومات عن البيانات في أنها مرتبة، مصنفه، وذات فائدة معينة.

1.3 هياكل بيانات

هياكل البيانات هي تمثيل لعلاقات منطقية موجودة بين عناصر البيانات المفردة. بكلام اخر، فان هياكل البيانات تحدد طريقة تنظيم كامل عناصر البيانات والتي تأخذ بالاعتبار ليس البيانات التي تخزن فقط لكن ايضا علاقتها مع بعض. مصطلح هياكل البيانات يستخدم لوصف طريقة خزن البيانات.

اذن هياكل البيانات هي طريقة خاصة لخزن وتنظيم البيانات في الحاسوب، بحيث من الممكن ان تستخدم بشكل كفوء.

لتطوير برنامج لخوارزمية معينة يجب علينا اختيار هيكل البيانات المناسب لهذه الخوارزمية. لذلك، فان البرنامج عبارة عن:

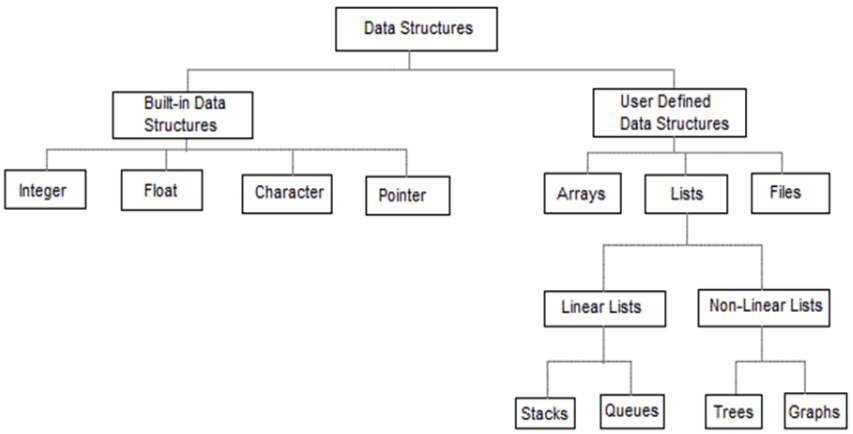
خوارزمية + هيكل بيانات = برنامج

هياكل البيانات يقال عنها خطية (linear) إذا كانت عناصرها عبارة عن متوالية او قائمة خطية. هياكل البيانات الخطية مثل المصفوفات، المكدس، الطابور، القوائم الموصولة تنظم بياناتها على شكل ترتيب خطي. ويقال عن هياكل البيانات انها غير خطية إذا كانت عناصرها عبارة عن تصنيف هرمي حيث تظهر عناصر بياناتها بمستويات مختلفة.

الاشجار والمخططات تستخدم بشكل كبير هياكل البيانات غير الخطية. هياكل الاشجار والمخططات تمثل علاقة هرمية بين عناصر البيانات المفردة.

1.4 انواع هياكل البيانات

انواع البيانات هي طريقة لتصنيف مختلف انواع البيانات مثل الاعداد الصحيحة، السلاسل الحرفية...الخ. والتي تحدد القيم التي من الممكن استخدامها مع نوع البيانات المقابل، نوع العمليات التي من الممكن انجازها على البيانات المقابلة.
البيانات على نوعين نوع بيانات مبني داخليا ونوع بيانات مشتق يبنى بواسطة المستخدم.



شكل1.1: انواع هياكل البيانات

1.4.1 هياكل البيانات المبنية داخليا Built-in Data Structures

هي هياكل البيانات الاساسية (وتسمى احيانا هياكل البيانات البدائية) والتي تعمل بشكل مباشر على ايعازات الماكنة. لها تمثيلات مختلفة في الحواسيب المختلفة. الارقام الصحيحة، ارقام النقطة العائمة، ثوابت الرموز، ثوابت السلاسل الحرفية، والمؤشرات جميعها ضمن هذا المفهوم.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

1.4.2 هياكل البيانات المشتقة Derived Data Type

هياكل البيانات المشتقة هي أكثر هياكل بيانات تعقيدا وتشتق من هياكل البيانات البدائية. هي تؤكد او تركز على تجميع نفس عناصر البيانات او عناصر بيانات مختلفة والتي لها علاقات فيما بينها (كل عناصر البيانات). المصفوفات، القوائم، والملفات تأتي ضمن هذا المفهوم. هياكل البيانات المشتقة هي البيانات التي تنفذ بشكل مستقل حيث من الممكن تنفيذها بطريقة او اخرى. انواع البيانات هذه عادة تبنى بدمج انواع البيانات البدائية او المبنية داخليا مع العمليات التي تشترك معها. مثل

- List
- Array
- Stack
- Queue

1.4.3 هياكل البيانات الثابتة والمتغيرة (الديناميكية)

من الممكن ان تصنف هياكل البيانات على نوعين وهما هياكل البيانات الثابتة وهياكل البيانات المتغيرة.

❖ **هياكل البيانات الثابتة**

مثل المتجهات والجداول والسجلات (المصفوفات)، حيث عند الاعلان عنها يجب تحديد حجم هذه البيانات فلا تقبل الاضافة فوق حجمها اثناء تنفيذ البرنامج.. وتمتاز

1. الوصول السريع للعناصر.
2. كلفة باهضه نسبيا بعمليات الحشر والازالة.
3. لها حجم ثابت.

❖ **هياكل البيانات المتغيرة**

هي هياكل البيانات التي تنمو وتتقلص اثناء التنفيذ مثل (القوائم الموصولة، المكدس، الطابور، الاشجار الثنائية)

1. سرعة في عمليات الحشر والازالة للعناصر.
2. بطيء في الوصول الى العناصر.
3. مرونة في تغيير الحجم.

وهياكل البيانات المتغيرة نوعان:

- ✓ هياكل بيانات خطية متغيرة: مثل القوائم، الطوابير، الملفات، المكدس.
- ✓ هياكل بيانات متشعبة (عشوائية): مثل الاشجار، والمخططات.

1.5 صفات هياكل البيانات

الصفات العامة التي تتميز بها هياكل البيانات هي:

- تحتوي على عناصر من الممكن ان تولد عنصر، مجموعة عناصر او هيكل بياني اخر.

- لها مجموعة من العمليات التي من الممكن ان تنجز على عناصر البيانات. مثل البحث، الحشر...الخ.
- تصف القواعد لكيفية علاقة عناصر البيانات مع بعضها.
- وقت التشغيل او التنفيذ لعمليات هياكل البيانات يجب ان يكون بأقل وقت ممكن.
- الذاكرة المستخدمة لعملية هياكل البيانات يجب ان تكون بأقل ما يمكن.

1.6 لماذا هياكل البيانات

نظرا لازدياد تعقيد التطبيقات ووفرة البيانات، فان هناك ثلاث مشاكل تطبيقية عامة تواجهنا هذه الايام.

1) **بحث البيانات**- نفرض مطلوب جرد مليون عنصر ضمن مخزن. لذا فان التطبيق يبحث عن كل عنصر. للبحث عن عنصر ضمن مليون عنصر مثلا، فان كل عنصر يستغرق وقتا لإيجاده ومعالجته، وبالتالي فان كل عنصر سيجعل عملية البحث أكثر بطأ. لو فرضنا ان هذه البيانات نمت (أي كبر حجمها وازداد عدد عناصرها) فان البحث بالتأكيد يصبح ابطأ.
2) **سرعة المعالج**- سرعة المعالج بالرغم من انها تبدو سريعة جدا، لكن هذه السرعة ستكون أكثر تحديدا إذا البيانات نمت الى بليون قيد مثلا.
3) **الطلبات المتعددة**- إذا كان هناك مليون مستخدم يبحث عن بيانات بشكل متزامن على خادم الويب، حتى وان كان الخادم سريع جدا فانه ربما سيفشل عندما يبحث البيانات.

هذه بعض المشاكل التي تستوجب ايجاد حل لها، هياكل البيانات تأتي للإنقاذ. البيانات من الممكن ان تنظم بهيكل بيانات بطريقة بحيث انها تساعد على ان لا تكون هناك حاجة لبحث كل البيانات المطلوبة، من الممكن ان تبحث غالبا بشكل

لحظي عند استخدام هيكل البيانات المناسب (بغض النظر عن سرعة المعالج وعدد طلبات البحث).

واحدة من الطرق التي ننظر بها الى هياكل البيانات هو ان نركز على نقاط القوة والضعف لكل نوع منها. في هذا الفصل سنوفر نظرة عامة على هذه النقاط والتي نوضحها في الجدول 1.1.

جدول 1.1: الصفات العامة لهياكل البيانات

هيكل البيانات	الفوائد	المساوئ
المصفوفات	سرعة حشر العناصر, سرعة كبيرة في الوصول الى عناصر المصفوفة	بطء عملية البحث, بطء عملية الحذف, حجمها ثابت
المصفوفات المرتبة	اسرع في عمليات البحث عن المصفوفات غير المرتبة	بطء عملية الحشر والحذف, حجمها ثابت
المكدس	توفر طريقة من يصل اخيرا, يخرج اولا	وصول بطيء الى العناصر الاخرى
الطابور	توفر طريقة من يصل اولا يخرج اولا	بطء الوصول الى العناصر الاخرى
القوائم الموصولة	سرعة الحشر, سرعة الحذف	بطء البحث
الاشجار الثنائية	سرعة البحث, الحشر, الحذف (اذا بقت الشجرة موزونة)	خوارزمية الحذف معقدة
جدول التجزئة	وصول سريع جدا اذا عرف المفتاح, سرعة الحشر.	بطء الحذف, الوصول يكون بطيء ان لم يعرف المفتاح, غير كفوءة باستخدام الذاكرة.
التكديس	سرعة في الحشر, الحذف	وصول بطيء الى باقي العناصر. الوصول الى اكبر عنصر
المخططات	ينمذج حالات العالم الحقيقي	بعض الخوارزميات بطيئة ومعقدة.

الجدول 1.1 يمثل خواص هياكل البيانات التي سنناقشها في هذا الكتاب، لذلك لا تندهش ان كانت غامضة بعض الشيء فسناتي عليها بالتفصيل في الفصول اللاحقة.

1.7 ماهي الخوارزمية

الخوارزمية هي مجموعة منتهية من الايعازات او المنطق، تكتب لغرض انجاز مهمه معرفة مسبقا. الخوارزمية هي ليست شفرة كاملة او برنامج، هي فقط المنطق الاساس (الحل) لمشكلة، والتي من الممكن توضيحها اما على شكل وصف ذا مستوى عال (باستخدام اللغات الحية مثلا) او باستخدام المخططات (flowchart).

الخوارزمية يقال عنها انها كفوءة وسريعة إذا كان وقت تنفيذها قليل وإذا لم تستهلك مساحة كبيرة من الذاكرة. ان كفاءة اداء أي خوارزمية تقاس بموجب ما يلي:

1. تعقيد الوقت Time Complexity
2. تعقيد الفضاء Space Complexity

1.7.1 تعقيد الفضاء

هي تمثل كمية فضاء الذاكرة المطلوب من الخوارزمية خلال دورة حياتها (خلال تنفيذها). ان تعقيد الفضاء يجب ان يؤخذ بنظر الاعتبار بشكل جدي في الانظمة التي تسمح بتعدد المستخدمين وفي حالة ان يكون حجم الذاكرة المتوفر محدود. الفضاء المطلوب من اي خوارزمية يساوي مجموع فضاء الذاكرة لاثنان من المكونات:

- جزء ثابت وهو الفضاء المطلوب لخزن بيانات معينة ومتغيرات، والتي لا تعتمد على حجم المشكلة. مثال المتغيرات البسيطة والثوابت المستخدمة، حجم البرنامج ...الخ.
- جزء متغير وهو فضاء مطلوب من المتغيرات التي لها حجم يعتمد على حجم المشكلة. مثال تخصيص الذاكرة الديناميكي، فضاء المكدس ...الخ.

تعقيد الفضاء S(P) لأي خوارزمية P يكون

S(P) = C + SP(I)

حيث ان C هي الجزء الثابت و SP(I) هي الجزء المتغير للخوارزمية التي تعتمد على الخاصية الانية (I). فيما يلي مثال نحاول من خلاله توضيح هذا المفهوم

Algorithm: SUM (A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

هنا لدينا ثلاث متغيرات (A, B, and C) وثابت واحد، لذلك فان **S(P) = 1+3.** الان الفضاء يعتمد على نوع البيانات للمتغيرات التي تعطى ونوع الثابت والتي ستضرب بها وفقا لحجمها في الذاكرة.

1.7.2 تعقيد الوقت

تعقيد الوقت لأي خوارزمية هي طريقة لتمثيل كمية الوقت المطلوبة من قبل البرنامج لإكمال عمله (اكمال التنفيذ). متطلبات الوقت من الممكن ان تعرف كدالة رقمية **T(n)** حيث ان **T(n)** من الممكن قياسها كعدد خطوات في البرنامج، توفر كل خطوة كمية وقت مستهلك.

على سبيل المثال اضافة اثنان من الارقام الصحيحة كل واحد لها n-bit ستحتاج الى n من الخطوات. بذلك فان الوقت الكلي للحساب هو **T(n) = c*n** حيث ان **c** تمثل الوقت اللازم لجمع اثنان من البتات. هنا نلاحظ ان **T(n)** تنمو (تكبر) بشكل خطي إذا زاد حجم المدخل.

تعقيد الوقت للخوارزميات عادة يوضح بشكل عام باستخدام حرف ((O الكبيرة.

تعقيد الوقت غالبا يقدر بحساب عدد الدوال الاولية التي تنجز بالبرنامج. ونظرا لان كفاءة الخوارزمية ربما تتغير مع اختلاف انواع البيانات المدخلة، عليه فأننا سنستخدم لكل خوارزمية (اسوأ تعقيد وقت **worst-case Time complexity**) وذلك لان هناك اقصى وقت يحسب لأي حجم مدخلات.

عادة متطلبات الوقت لخوارزمية يقع تحت نوعين:

- افضل وقت Best Case : اقل وقت مطلوب لتنفيذ البرنامج.
- اسوأ وقت Worst Case: أكبر وقت مطلوب لتنفيذ البرنامج.

1.7.3 حساب وقت التعقيد Calculating Time Complexity

الان دعنا نتدرج الى اهم موضوع يتعلق بتعقيد الوقت، وهو كيف من الممكن ان نحسب وقت التعقيد. هو يبدو احيانا مربك جدا، ولكن سنحاول توضيحها بطريقة مبسطة.

الان المقياس الاكثر عمومية لحساب وقت التعقيد هو (O) الكبيرة. هذه تزيل جميع المعاملات الثابتة، لذا فان وقت العمل للخوارزمية من الممكن ان يقدر نسبة الى (N)، حيث ان N تمثل عدد العبارات وتصل الى ما لانهاية. بشكل عام من الممكن ان نفكر بها مثل هذه:

```
Statement ;
```

اعلاه لدينا عبارة واحدة. تعقيد الوقت لها سيكون ثابت. وقت التشغيل للعبارة سوف لا يتغير نسبة الى N.

```
for (i=0 ; i < N ; i++)
{
```

```
  statement;
}
```

تعقيد الوقت للخوارزمية اعلاه سيكون خطي. وقت التشغيل او التنفيذ لحلقة التكرار تتناسب بشكل مباشر مع قيمة N (حيث ان N تمثل عدد مرات تنفيذ العبارة statement). عندما نضاعف N فان وقت التنفيذ سيضاعف.

```
for (i=0; i < N; i++)
{
  for (j=0; j < N; j++)
  {
    statement;
  }
}
```

هنا، تعقيد الوقت للخوارزمية اعلاه سيكون تربيعي. وقت التنفيذ لحلقتي التكرار تتناسب مع مربع N. فعندما تضاعف N, فان وقت التنفيذ يزداد بمقدار N * N.

```
while (low <= high)
{
  mid = (low + high) / 2;
  if (target < list[mid])
    high = mid - 1;
  else if (target > list[mid])
    low = mid + 1;
  else break;
}
```

الخوارزمية اعلاه هي لتجزئة مجموعة من الارقام الى النصف. الان، هذه الخوارزمية سيكون لها لوغاريثم وقت تعقيد. بالتاكيد فان وقت التنفيذ للخوارزمية يتناسب مع عدد مرات N التي من الممكن التقسيم على (2) (حيث ان N هنا تمثل الاعلى – الادنى (high-low)). هذا بسبب ان الخوارزمية تقسم مساحة العمل الى النصف في كل دورة.

```
void quicksort (int list [ ], int left, int right)
{
  int pivot = partition (list, left, right);
  quicksort (list, left, pivot - 1);
  quicksort (list, pivot + 1, right);
}
```

لنأخذ المثال السابق ونناقشه، الخوارزمية اعلاه هي جزء منطقي صغير من خوارزمية ترتيب الاعداد (سندرسها لاحقا). الان في هذه الخوارزمية سننصف القائمة كل مرة (يعني وقت التعقيد هو لوغاريثم)، ولكن نعيد التكرار N من المرات (حيث ان N يمثل حجم القائمة) (بمعنى اعادة التعقيد اللوغاريثمي بعدد N من المرات). لذا فان وقت التعقيد سيكون **N*log (N).**

وقت التنفيذ لعدد N من التكرارات والتي هي لوغاريثمية، وهذا يعني ان الخوارزمية هي لوغاريثمية وخطية بنفس الوقت.

ملاحظة: بشكل عام، عمل شيء مع كل عنصر في مصفوفة احادية هو خطي، عمل شيء مع كل عنصر في مصفوفة ثنائية هو تربيع، تقسيم مساحة العمل للنصف هو لوغاريثمي. في الجدول 1.2 بعض الحالات القياسية لتعقيد الوقت.

جدول 1.2: تعقيد الوقت لبعض الخوارزميات

الخوارزمية	تعقيد الوقت
ثابت constant	O(1)
لوغاريثمي logarithmic	O(log n)
خطي linear	O(n)
N log n	O(n log n)
مربع quadratic	$O(n^2)$
مكعب cubic	$O(n^3)$
متعدد الحدود polynomial	$n^{O(1)}$
الاسي exponential	$2^{O(n)}$

هياكل البيانات باستخدام C++

الفصل الثاني
المصفوفات ARRAYS

2.1 المقدمة

القوائم هي واحدة من أكثر هياكل البيانات التي يتكرر استخدامها. بالرغم من ان كل البرامج تشترك بنفس التعريف للقوائم وهي (متوالية من عناصر البيانات المتجانسة)، لكن نوع البيانات التي تخزن في القوائم تختلف من برنامج لأخر. بعضها يستخدم قوائم من الاعداد الصحيحة، اخرى تستخدم قوائم الرموز، الارقام الكسرية... وهكذا. في هذا الفصل سيتم التركيز على اول انواع هياكل البيانات وهي المصفوفات بنوعيها المصفوفات ذات البعد الاحادي والمصفوفات المتعددة الأبعاد مع امثلة توضيحية والعمليات التي من الممكن ان تجرى عليها.

2.2 المصفوفات

المصفوفات هي هيكل بيانات تخزن مجموعة من المتغيرات لها نفس النوع، فهي تجميع لكيانات البيانات المتشابهة والتي تخزن في مواقع ذاكرة متجاورة تحت اسم محدد. بكلام اخر، فان المصفوفة هي مجموعة من الكيانات (تسمى العناصر)، جميع هذه العناصر من نوع واحد ويتم خزنها في الذاكرة في مواقع متجاورة، وتعرف المصفوفة من خلال الاسم الذي يسند لها ويتم اختيار اسم المصفوفة وفقا لقواعد اختيار اسماء المتغيرات، ويستخدم هذا الأسم للإشارة الى المصفوفة وليس الى عناصر المصفوفة اذ ان عناصر المصفوفة يتم الاشارة الى كل واحد منها باستخدام أسم المصفوفة متبوعا برقم الدليل (الفهرس) الذي يشير الى موقع العنصر في المصفوفة.

تستخدم المصفوفة كبديل عن الاعلان عن عدد من المتغيرات المتشابهة النوع، فمثلا برنامج يحتاج الى عشرة متغيرات من نوع الأعداد الصحيحة، فبدلا من الأعلان عن عشرة متغيرات وبأسماء مختلفة يمكن ان تعلن عن هذه العناصر كمصفوفة من نوع الأعداد الصحيحة، حجمها عشرة عناصر ولها اسم واحد، وان

كل عنصر من الممكن ان يعامل كمتغير مفرد ليس له علاقة بباقي عناصر المصفوفة الأخرى ويتم الاشارة له من خلال أسم المصفوفة وموقع العنصر في المصفوفة.

2.3 المصفوفات الاحادية

المصفوفة الاحادية هي هيكل مركب لنوع بيانات يتكون من تجميع لعناصر متجانسة، مرتبة، ذات حجم ثابت، منتهية بحيث يكون الوصول لها ممكن. وعندما نقول منتهية هذا يعني ان العنصر الاخير محدد. حجم ثابت يعني حجم المصفوفة يجب ان يكون معروف مسبقا، وهي لا تعني بان كل خلية في المصفوفة يجب ان تحتوي على قيم ذات معنى. مرتبة تعني ان هناك عنصر اول وعنصر ثاني وهكذا. (الموقع النسبي للعناصر مرتبة، وليس بالضرورة ان تكون القيم في المصفوفة مرتبة). ولان جميع عناصر المصفوفة يجب ان تكون جميعها من نفس النوع، فهي ماديا متجانسة (اي كلها لها نفس نوع البيانات).

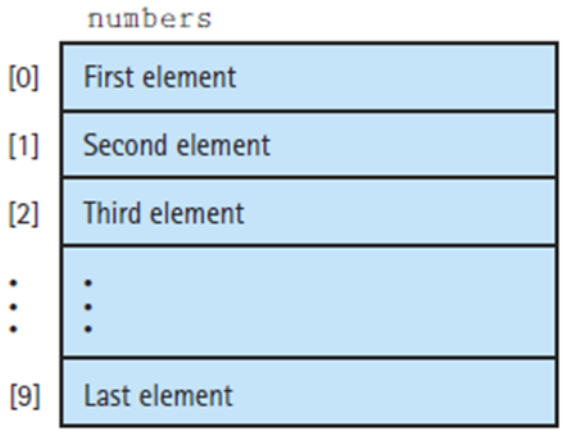
في C++ يتم الاعلان عن المصفوفة بتحديد نوعها اولا ثم اسم المصفوفة واخيرا حجمها كما في ادناه.

```
int numbers [10] ;
```

لاحظ هنا ان حجم المصفوفة يوضع بين قوسين مربعين. هذا الاعلان يعلن عن ترتيب خطي لمواقع المصفوفة (مواقع متجاورة), ومن الممكن ان نصور المصفوفة بعد ان يتم الاعلان عنها كما في الشكل 2.1 (لاحظ ان الاعلان عن المصفوفة يعني تحديد مواقع في الذاكرة خالية من القيم).

كل عنصر في المصفوفة من الممكن الوصول اليه مباشرة وذلك من خلال الموقع النسبي له في المصفوفة.

C++



شكل 2.1: مصفوفة احادية افتراضية

المصفوفة الاحادية هي عبارة عن سلسلة من العناصر المتشابهة النوع والتي تخزن في الذاكرة في مواقع متجاورة والتي من الممكن الاشارة لكل واحد من هذه العناصر بشكل مفرد من خلال اضافة الرقم الدليلي (index) الى الأسم التعريفي الوحيد لها، ومثلها مثل المتغيرات الاعتيادية فان المصفوفة يجب ان يتم الأعلان عنها قبل اول استخدام لها.

اسم المصفوفة هو اي اسم يتم اختياره من قبل المبرمج على ان يتبع القواعد المعروفة بتسمية المتغيرات، اما عدد العناصر التي تحتويها المصفوفة يجب ان يكون دائما محددا بين قوسين مربعين، وعند الاعلان عن المصفوفة فان المترجم سيحجز عددا من المواقع المتجاورة في الذاكرة طول كل موقع (عدد البايتات المحددة له) يساوي الحجم المحدد لذلك النوع، وطبعا نفترض ان هذه المواقع خالية من اي قيمة او ربما هي تحتوي على قيمة قديمة ليس لها علاقة بهذا البرنامج ويجب تغييرها.

ملاحظة: دائما الرقم الموجود بين القوسين المربعين والذي يمثل عدد العناصر يجب ان يكون من الاعداد الصحيحة الموجبة او حروف فقط. التعبيرات التالية غير مقبولة.

Static int value [0.02];

Float number [-90];

Char **s [$];**

ملاحظة: قبل استخدام أي مصفوفة احادية او متعددة الابعاد في البرنامج، يجب توفير المعلومات التالية الى المترجم او المفسر

1- نوع المصفوفة مثلا (int, float, char…).

2- اسم المصفوفة (ويتم اختياره من المبرمج ويفضل ان يدل على عمل المصفوفة).

3- عدد الأبعاد (subscript) في المصفوفة (هل المصفوفة احادية او متعددة الابعاد).

4- العدد الكلي لمواقع الذاكرة المخصصة او بتحديد أكثر، عدد العناصر لكل بعد من أبعاد المصفوفة.

2.4 انشاء المصفوفة Array Initialization

عند الاعلان عن المصفوفة فأنها ستنشأ (تخلق) كمصفوفة خالية من القيم، حيث ان عناصرها لا تحتوي على قيم مالم يتم خزن قيم فيها **اي** اسناد قيم ابتدائية لهذه العناصر وتسمى هذه العملية ابتداء المصفوفة، لذلك يجب عدم أجراء اي عملية على عناصر المصفوفة إذا لم يتم اسناد قيم لها، كما هو الحال مع المتغيرات الاعتيادية.

المصفوفات ايضا من الممكن ان تعرف كما في المتغيرات الاعتيادية على انها محلية او عامة (Global and Local)، وفي كلتا الحالتين سواء كانت مصفوفة عامة او محلية فان اسناد القيم لها من الممكن ان يتم بأكثر من طريقة، واحدة من هذه الطرق اسناد قيم ابتدائية لكل عنصر من عناصرها وذلك من خلال

وضع قيم بين قوسين متوسطين تفصل بين قيمة واخرى فارزة (بنفس طريقة كتابة المجاميع) ومساواتها الى المصفوفة (تسند لها قيم اثناء كتابة البرنامج) كما يأتي:

```
ABC [ 5 ] = { 5, -77, 34, 60, 2 } ;
```

ويجب ان تنتبه الى ان عدد القيم بين القوسين المتوسطين يجب ان لا تزيد على عدد عناصر المصفوفة التي تم الاعلان عنها في اعلان المصفوفة (فاذا أعلن عن مصفوفة من خمسة عناصر ووضع بين القوسين المتوسطين ستة قيم، في هذه الحالة سيصدر المترجم رسالة خطأ. القيم ستسند الى عناصر المصفوفة بالتتالي من اليسار الى اليمين (اي ان القيمة في اقصى اليسار (5) ستسند الى العنصر في الموقع (0)، والقيمة التي على يمينها (77-) ستسند الى العنصر في الموقع (1).. وهكذا).

ملاحظة: عندما يتم ابتداء القيم سوف تسند لعناصر المصفوفة، C++ يسمح بإمكانية ترك الاقواس المربعة فارغة []. في هذه الحالة، فان المترجم سيفرض حجم للمصفوفة يطابق عدد القيم الموجودة بين الاقواس المتوسطة. مثال

```
ABC [ ] = {2, 5, 8} ;
```

هنا سيحدد المترجم عدد العناصر (حجم المصفوفة) بثلاث.

ملاحظة: في ادناه بعض الامثلة المقبولة لابتداء المصفوفة

```
int  value [7] ={10, 11, 12, 13, 14, 15, 16};
float  coordinate [5] ={0, 0.45, -0.5, -4.0, 5.0};
char sex [2] ={`M`,` F `};
char name [5] ={`s`,`i`,`n`,`a`,`n`};
```

من الممكن انشاء المصفوفة وذلك بأسناد قيم لعناصر المصفوفة من لوحة المفاتيح عند تنفيذ البرنامج، وهذه هي الطريقة الافضل لأنها تعني ان البرنامج سيكون عام لأي قيم يمكن ادخالها ولا يحدد بالقيم التي تم كتابتها اثناء كتابة البرنامج كما في

الطريقة السابقة. هذه الطريقة وهي الاكثر شيوعا ستعتمد على حلقة التكرار لإسناد قيم لعناصر المصفوفة كما في المثال ادناه.

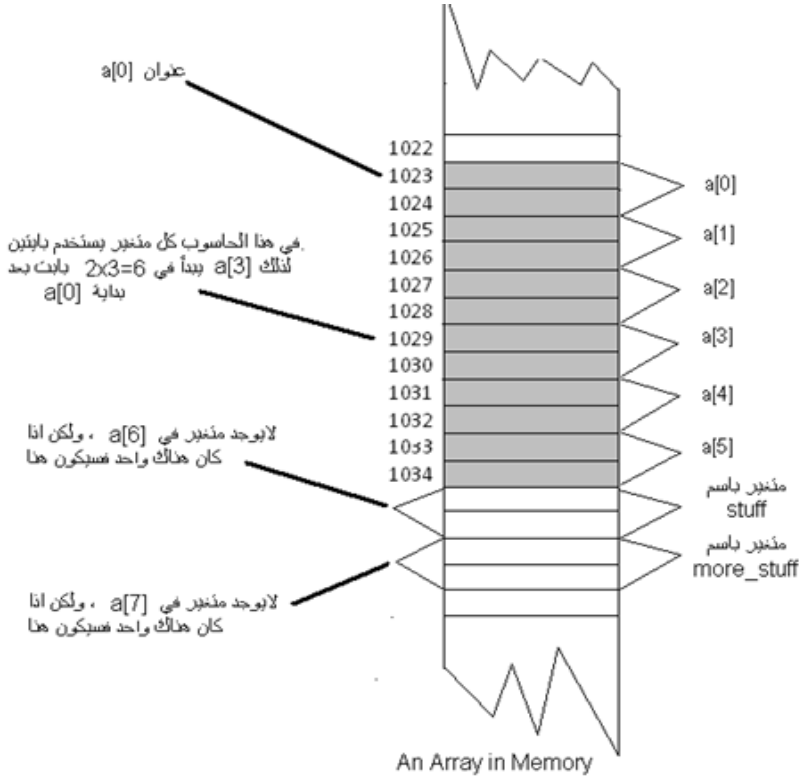
```
#include<iostream>
 using namespace std ;
void main (void) {
int A [7] ;
int   i ;
for ( i= 0 ; i <= 6 ; i++ )
  cin >> A [i] ;
}
```

لاحظ اننا استخدمنا حلقة تكرار بعدد عناصر المصفوفة وذلك لكي يتم المرور على جميع مواقع المصفوفة ويسند لها قيم. اما طباعة عناصر المصفوفة فتتم بنفس الطريقة التي استخدمنا فيها حلقة التكرار لأسناد قيم لعناصر المصفوفة مع تغيير ايعاز الادخال بإيعاز الاخراج.. على ان تنتبه الى انه لا يمكنك طباعة اي عنصر من عناصر المصفوفة إذا لم تسبقه بعملية اسناد قيم لعناصر المصفوفة.

ملاحظة: المصفوفة الاحادية عادة ودائما تبدأ بالموقع صفر.

مثال: برنامج يوضح طريقة اسناد وطباعة عناصر مصفوفة.

```
# include<iostream>
using namespace std;
void main (void) {
int A [7] ={11, 12, 13, 14, 15, 16, 17};
int   i ;
cout<< " contents of the array: \n " ;
for (i=0 ; i <= 6; i++)
  cout << a[i] << '\t' ;      }
```



شكل 2.2: يوضح كيفية خزن عناصر المصفوفة في ذاكرة الحاسوب.

ان من أكثر الاخطاء التي تحدث هي عندما تحاول خزن عنصر في مواقع ماوراء حجم المصفوفة، فمثلا لو عرفت مصفوفة كما يأتي:

int A [6] ;

عندما تستخدم هذه المصفوفة فان دليل المصفوفة يتراوح بين (0-5)، فاذا حددت الدليل بغير ذلك فان خطأ سيحدث. في معظم الحواسيب لا يوجد تحذير عند

استخدام دليل خارج حجم المصفوفة، كمثال افرض أنك حددت قيمة الى الدليل الموضح ادناه:

A [7] = 255 ;

هنا الحاسوب سيعامل هذا الامر على انه صحيح ويضع القيمة 225 في العنوان المناسب في الذاكرة، ولكن عند حساب موقع او عنوان هذ القيمة فأنها ستكون في العنوان الذي يحوي المتغير (more_stuff) كما في الشكل 2.2, ولذلك فان هذه القيمة الخاصة بالمتغير (more_stuff) سوف تتغير بشكل غير مقصود.

ملاحظة: ان حساب موقع اي عنصر من عناصر المصفوفة في الذاكرة يتبع العلاقة التالية:

الموقع في الذاكرة = العنوان الابتدائي + رقم الدليل * عدد بايتات نوع المصفوفة

العنوان الابتدائي هو عنوان اول عنصر بالمصفوفة (في الشكل 2.2 هو 1023 للمصفوفة A).

رقم الدليل هو موقع العنصر في المصفوفة مثلا العنصر في الموقع 5.

عدد بايتات النوع هو عدد البايتات التي تحدد لكل نوع في الذاكرة فمثلا يحدد بايتين للأعداد الصحيحة (من الممكن اربعة) وبايت للحروف وهكذا.

فمثلا لو أردنا استخراج موقع الذاكرة للإيعاز A [7] سيكون

العنوان = 1023 + 7 * 2 = 1037

2.5 الوصول الى عناصر المصفوفة

عند كتابة برنامج يحتوي على مصفوفة فان بإمكانك الوصول الى اي عنصر من عناصر المصفوفة بشكل مفرد وفي اي مكان من البرنامج وتتعامل مع هذا العنصر كما تتعامل مع اي متغير عادي من حيث القراءة والتحديث. ان الصيغة المستخدمة للإشارة الى اي عنصر يتم من خلال كتابة اسم المصفوفة متبوع برقم

موقع هذا العنصر في المصفوفة، ويكون الرقم محدد بين قوسين مربعين، كما في ادناه:

```
Name [ index ] ;
```

هذه الصيغة تمثل قيمة العنصر، وهي تكافئ اسم المتغير الاعتيادي وعلى هذه الصيغة بالإمكان اجراء كل العمليات التي بالإمكان اجراءها على المتغير الاعتيادي من ذلك النوع. فمثلا إذا كنت ترغب بأسناد القيمة 45 الى العنصر الثاني في المصفوفة (ABC)، فسيتم ذلك كما يأتي:

```
ABC [ 2 ] = 45 ;
```

كما يمكنك ان تمرر هذه القيمة الى متغير اخر اعتيادي مثلا (x)، وكما يأتي:

```
x = ABC [ 2 ] ;
```

عليه فان المتغير (x) ستكون قيمته مساوية الى 45.

ملاحظة: تستخدم الاقواس المربعة مع المصفوفات لأمرين:

الأول: يستخدم للأعلان عن حجم المصفوفة عندما يحتوي عددا صحيحا موجبا وقت الاعلان عن المصفوفة.

الثاني: يستخدم في تحديد موقع العنصر في المصفوفة.

ملاحظة: إذا لم يتم مساواة عدد عناصر المصفوفة خارجيا عند ابتداء المصفوفة كأن يكون عدد القيم المسندة والمحددة بين القوسين المتوسطين هو اقل من العدد الذي يحدد حجم المصفوفة، ففي هذه الحالة، فان هذه القيم ستسند الى العناصر المقابلة لها اما باقي العناصر فستسند لها القيم الافتراضية وهي صفر. مثال

```
Myarray  [ 5 ] = {3, 65, 21} ;
```

فستكون قيم العناصر كما يأتي:

```
Myarray [ 0 ] =  3 ;
Myarray [ 1 ] = 65 ;
Myarray [ 2 ] = 21 ;
```

```
Myarray [ 3 ] =  0 ;
Myarray [ 4 ] =  0 ;
```

ملاحظة: في C++ لا يسمح بالعمليات البسيطة التي تتضمن كامل المصفوفة. حيث ان اسم المصفوفة يعامل كمتغير منفصل للعمليات مثل عملية المساواة (الاسناد)، عمليات المقارنة ...وهكذا فمثلاً لو كانت (A, B) مصفوفتان من نفس النوع وذات الحجم فأن عمليات الاسناد والمقارنة يجب ان تجري فقط لعنصر مع عنصر اخر.

```
int A[4]={2, 3, 4, 5}
int B[4]={1, 3, 5, 7}
```

فالعمليات التالية مقبولة:

```
* if  (A [ 2 ] > B [ 2 ] )
   cout<<" array are different \n ";

* while ( A [ 1 ] == B [ 3 ] )
  cout<<" AAAAAAAAAA \n ";
```

العمليات التالية غير مقبولة:

```
* if (A == B)
    cout<<" array elements are equal \n ";
* while (A > B)
    cout<<" array processing \n ";
```

2.6 المصفوفات المتعددة الابعاد

المصفوفات المتعددة الابعاد من الممكن ان تعرف على انها مصفوفة المصفوفات، فمن الممكن ان تكون لك مصفوفة تحتوي على أكثر من بعد واحد، كل بعد من الممكن ان يمثل في المصفوفة كرقم دليلي، انت تعلم ان المصفوفة ذات البعد الواحد كان لها رقم دليلي واحد بعد اسم المصفوفة، لكن المصفوفة ذات

البعدين سيكون لها رقما دليل بعد اسم المصفوفة، والمصفوفة ذات الثلاثة ابعاد لها ثلاثة ارقام دليل بعد اسم المصفوفة وهكذا. المصفوفات من الممكن ان يكون لها اي عدد من الابعاد، ولكننا سنكتفي في هذا القسم بشرح المصفوفة ذات البعدين فضلا عن المصفوفة ذات البعد الواحد لانهما الاكثر استخداما، وجميع المصفوفات ذات الابعاد الاكثر تطابق المصفوفة ذات البعدين بالعمل. من الامثلة الجيدة للمصفوفات الثنائية هي رقعة الشطرنج، حيث تتكون من ثمانية صفوف وثماني اعمدة (كل صف يمثل مصفوفة احادية وكل عمود يمثل مصفوفة احادية ايضا)، المصفوفات الثنائية تتكون من صفوف واعمدة ترقم الصفوف ابتداءا من الرقم (0) وترقم الاعمدة ايضا ابتداءا من الرقم (0). وكل خلية في المصفوفة الثنائية تمثل موقع في الذاكرة وبالتالي ستحمل قيمة، وكما في المصفوفات الاحادية فان لكل مصفوفة ثنائية اسم وحيد تعرف به وهو اي اسم يتم اختياره من المبرمج على ان يتبع قواعد تسمية المتغيرات، وبالتأكيد فان لكل مصفوفة ثنائية نوع وهو يمثل نوع البيانات المخزنة في المصفوفة وبالإمكان استخدام اي نوع من الانواع المقبولة في لغة C++.

المصفوفات الثنائية لها استخدامات كثيرة وهي تساعد على تسهيل التعامل مع بعض المسائل المعقدة.. فمثلا لدينا عدد من المعامل (ثلاثة معامل.. معمل1، معمل2، معمل3) التي تنتج مواد كهربائية متشابهة مثل (تلفزيون، ثلاجة، غسالة، مجمدة، مكيف) فمن الممكن تمثيلها بمصفوفة ثنائية والتعامل معها على هذا المبدأ كما يلي:

	مكيف	مجمدة	غسالة	ثلاجة	تلفزيون
معمل1	20	12	56	34	23
معمل2	21	34	44	43	22
معمل3	12	15	23	31	42

شكل 2.3: مثال توضيحي لتمثيل المصفوفات الثنائية

الان لو سألنا كم غسالة انتجت في المعمل 1.. بالتأكيد سيكون الجواب 56، وإذا كان السؤال كم مكيف أنتج في المعمل 3 فسيكون الجواب 12 وهكذا (عليك الان ان تستنتج الطريقة التي تتعامل بها مع عناصر المصفوفة). حجم المصفوفة هو (3×5) (اسماء الاعمدة والصفوف في الشكل 2.3 هي للتوضيح).

2.6.1 الاعلان عن المصفوفة الثنائية

يتم الاعلان عن المصفوفة الثنائية بنفس الطريقة التي يتم فيها الاعلان عن المصفوفة الاحادية وذلك بكتابة نوع المصفوفة متبوعا باسم المصفوفة ثم عدد العناصر في المصفوفة وهنا يكون عدد العناصر موزعا على قوسين مربعين (لأنها ثنائية)، القوس المربع الاول يحمل عدد الصفوف في المصفوفة الثنائية والقوس المربع الثاني يمثل عدد الاعمدة في المصفوفة، وكما يأتي:

```
int TestArray [3][5] ;
```

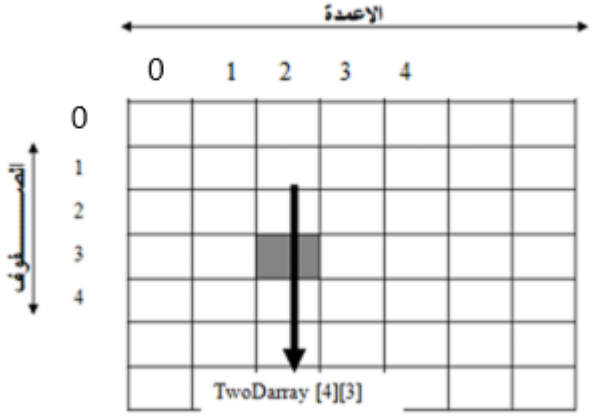
الاعلان اعلاه يمثل اعلان عن مصفوفة ثنائية (عدد الاقواس المربعة اثنان وهذا يعني انها ثنائية) من نوع الاعداد الصحيحة (اي ان جميع عناصرها من نوع الاعداد الصحيحة)، تحت اسم (TestArray) وهي تحتوي على ثلاثة صفوف وخمسة اعمدة (اي ان عدد عناصرها الكلي يساوي حاصل ضرب عدد الصفوف في عدد الاعمدة وسيكون مساوي 15 = 3x5).

ملاحظة: لا يجوز اطلاقا تخصيص القوس المربع الاول للأعمدة والثاني للصفوف، لان المترجم دائما ينظر الى القيمة التي في القوس المربع الاول على انها عدد الصفوف ونفس الشيء للقوس المربع الثاني حيث يعتبر القيمة التي فيه على انها عدد الاعمدة.

2.6.2 الوصول لعناصر المصفوفة الثنائية

الية الوصول الى اي عنصر في المصفوفة الثنائية يكون من خلال كتابة اسم المصفوفة ثم القوسين المربعين، حيث القوس الاول سيشير الى رقم الصف الذي

يتواجد به العنصر المطلوب، اما القوس المربع الثاني فيشير الى رقم العمود الذي يوجد فيه العنصر المطلوب (انظر الشكل 2.4)، اذ ان العنصر المطلوب هو العنصر المضلل وهو موجود بالصف الثالث والعمود الثاني، لذلك فان الوصول لأي عنصر من عناصر المصفوفات الثنائية يكون بدلالة رقم الصف ورقم العمود (ودائما القوس المربع الاول يستخدم لرقم الصف والقوس المربع الثاني لرقم العمود).



شكل 2.4: تمثيل للمصفوفة الثنائية

عند الوصول لأي عنصر من عناصر المصفوفة الثنائية فيمكنك التعامل معه واجراء كافة العمليات التي تتناسب مع نوعه كأي متغير اعتيادي، مثال

لغرض أسناد القيمة (56) لعنصر من عناصر مصفوفة ثنائية (نفرض ان العنصر هو في الموقع 5×3) فيتم ذلك كما يأتي:

```
TestArray [3][5] = 56 ;
```

لاحظ عند العمل على عناصر المصفوفة لا تحتاج لتحديد النوع لأنه تم تحديده عند الاعلان عن المصفوفة.

الان لو اردت طباعة قيمة هذا العنصر على الشاشة فسيكون كما يأتي:

```
cout << TestArray[3][5] ;
```

ويمكن مساواته لأي متغير اعتيادي مثل

```
x = TestArray[3][5] ;
```

طبعا ستكون قيمة المتغير (x) تساوي (56).

2.6.3 ابتداء المصفوفة الثنائية

يقصد بالابتداء هو اسناد قيم ابتدائية للمصفوفة ويكون بعدة طرق:

❖ يمكن ان تبتدأ المصفوفة الثنائية بنفس الطريقة التي تم فيها بدأ المصفوفة الاحادية وذلك من خلال كتابة اسم المصفوفة مع الاقواس التي تمثل الابعاد ومساواتها الى مجموعة من القيم (تتكون من مجموعة من القيم تفصل بين قيمة واخرى فارزة وتحدد القيم بين قوسين متوسطين مع ملاحظة ان عدد القيم يجب ان لا يزيد عن عدد عناصر المصفوفة) وكما يأتي:

```
int  theArray[5][3] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
```

في هذه الحالة فان اول ثلاث قيم يتم اسنادها الى المواقع الثلاث في الصف (1) وثاني ثلاث عناصر تسند الى المواقع الثلاث في الصف الثاني وهكذا.

❖ ويمكن ان تكون مجاميع ثلاثية ضمن المجموعة الرئيسة وتسندها للمصفوفة وكما يأتي:

```
int theArray [5][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15} } ;
```

المترجم سيهمل الاقواس الداخلية، هي توضع للمساعدة في فهم توزيع القيم بشكل سهل.

❖ بالإمكان اسناد قيم الى عناصر المصفوفة باستخدام لوحة المفاتيح اثناء تنفيذ البرنامج وذلك باستخدام حلقتي تكرار متداخلتين، الحلقة الخارجية تعمل كعداد

للصفوف (تضع مؤشر على الصفوف) بينما الحلقة الداخلية تعمل كعداد للأعمدة (تضع مؤشر على الاعمدة)، (بكلام اخر فان حلقتي التكرار ستعملان على وضع قيم للصفوف بالتتابع اي يتم وضع قيم لعناصر الصف (1) ابتداءا من العمود (1) الى العمود الاخير ثم ينتقل الى الصف الثاني وهكذا.

مثال: برنامج لقراءة مصفوفة ثنائية بإدخال قيم عناصر من لوحة المفاتيح.

```
#include <iostream>
using namespace std;

int main()
 {
 int SomeArray[5][4] ;
 for (int i = 0; i<5 ; i++)
 for (int j =0; j<4 ; j++)

 cin >> SomeArray [i][j] ;

 return 0;
        }
```

2.6.4 طباعة المصفوفة

يستخدم نفس البرنامج السابق لغرض طباعة عناصر المصفوفة على ان يتم ابدال امر الادخال بامر الاخراج وكما يأتي:

- مثال برنامج لقراءة وطباعة عناصر مصفوفة ثنائية.

```
#include <iostream>
using namespace std;
```

```
int main()
 {
 int SomeArray[5][4] ;
 for (int i = 0; i<5; i++)
 for (int j = 0; j<4; j++)

     cin >> SomeArray[i][j] ;

 for (int i = 0; i<5; i++)
 {
    for (int j = 0; j<4; j++)
         cout << SomeArray[i][j]<< "\t" ;
    cout << endl ;
 }
 return 0;
 }
```

لاحظ في المثال السابق لايمكن استخدام اوامر الاخراج مالم يتم اسناد قيم لعناصر المصفوفة باحدى طرق اسناد القيم المبينة اعلاه.

مثال: برنامج لقراءة مصفوفة اعداد صحيحة احادية حجمها (100 عنصر)، وايجاد العدد الاكبر في المصفوفة.

```
#include<iostream>
 using namespace std;

 void  main (void)
 {   int B[100], i, n, Larg ;
```

```
    cout<<"enter the elements "<<endl;
    for(i = 0 ; i <= 99; ++i)
      cin>> B[i] ;

    Larg = B[0];
       for (i=0 ; i <=99 ; ++i)
          if (Larg < b[i])
             Larg = b[i] ;
    cout<<" largest value in the array = "<< Larg ;
  }
```

مثال: برنامج لطباعة عناصر القطر الرئيس في المصفوفة ((5×5) D).

```
#include < iosream>
#define row  5
#define col  5
using namespace std;

main( ){
int D[row][col] ;
for ( int i=0 ; i< 5 ; i++ )
for ( int j =0 ; j< 5 ; j++ )
cin>>D [i][j] ;
for ( i =0 ; i < 5 ; i++ )
cout << D[i][i] << endl ;
return 0;
```

}

مثـال: برنـامج لقـراءة المصـفوفة ((4×5) AD)، ثـم بـدل عناصـر الصـف الثاني مع عناصر الصف الثالث.

```
#include <iostream>
using namespace std;

const   row = 4 ، col = 5 ;
void readarray ( AD[][] )
{
  for ( int i=0 ; i< row ; i++ )
  for ( int j =0 ; j<col ; j++ )
cin>>D[i][j] ;
}
void writearray ( AD[][] )
{
for ( int i=0 ; i< row ; i++ )  {
for ( int j =0 ; j< col ; j++ )
cout << D[i][j] << '\t' ;
cout << endl ;
}   }
main () {
int AD [row][col];
readarray ( AD );
writearray ( AD );
for ( in i =0 ; i < col ; i++ )
```

```
 {     int temp = AD[2][i] ;
       AD [2][i] = AD [3][i] ;
       AD [3][i] = temp ;
 }
writearray ( AD ) ;
return 0;
}
```

هياكل البيانات باستخدام C++

الفصل الثالث
الترتيب SORTING

3.1 المقدمة

الترتيب يشير الى تنظيم البيانات وفقا لصيغة او نظام معين. خوارزمية الترتيب تحدد الطريقة لتنظيم البيانات بترتيب معين. غالبية عمليات الترتيب المعروفة هي تعتمد على ترتيب الاعداد او المعاجم (الحروف).

ان اهمية عملية الترتيب تأتي من حقيقة ان بحث البيانات يكون بمستوى عال من الامثلية إذا كانت البيانات مرتبة بطريقة معينة. كذلك فان الترتيب يستخدم لتمثيل البيانات بصيغ أكثر قابلية للقراءة. بعض الامثلة للترتيب في حياتنا هي:

دليل الهاتف.. حيث ان دليل الهاتف يحفظ ارقام الهواتف للأشخاص مرتبة حسب اسمائهم. وبهذا فمن الممكن ان نبحث عن الاسماء والارقام بسهولة.

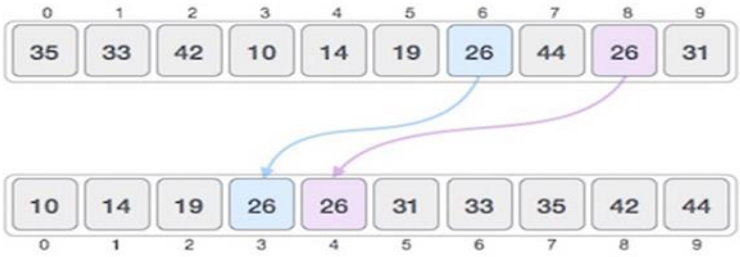
القاموس.. والقاموس هو لحفظ الكلمات مع معانيها، مرتبة بترتيب ابجدي ولذلك فان البحث عن اي كلمة يكون سهل.

خوارزميات الترتيب ربما تتطلب بعض مساحات الخزن الاضافية للمقارنة والخزن المؤقت لبعض العناصر القليلة من البيانات. هذه الخوارزميات التي لا تتطلب مساحة خزن اضافية، وعملية الترتيب تحدث في المكان او على سبيل المثال في المصفوفة نفسها، هذه تسمى ترتيب في المكان (in-place sorting). مثال على ذلك ترتيب الفقاعات Bubble sorting.

لكن في خوارزميات ترتيب اخرى، فان البرنامج يحتاج الى مساحة ربما تزيد او مساوية الى العناصر التي يراد ترتيبها. الترتيب الذي يحتاج الى مساحة مساوية او أكثر يسمى ترتيب ليس في المكان not-in-place sorting. مثال على ذلك طريقة الترتيب بالدمج Merge-sort

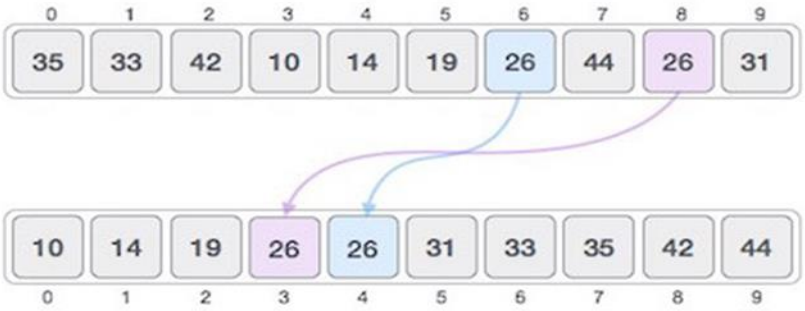
3.2 الترتيب المستقر وغير المستقر Stable and not Stable Sorting

ان خوارزمية الترتيب التي لم تغير ترتيب العناصر المتشابهة حسب ظهورها بعد اجراء ترتيب المحتويات، تسمى خوارزمية الترتيب المستقر كما في الشكل كما في الشكل 3.1.



شكل 3.1: توضيح لترتيب بيانات مستقر.

اما إذا كانت خوارزمية الترتيب، بعد ترتيب المحتويات، تغير ترتيب العناصر المتشابهة حسب ظهورها، فهذه تسمى ترتيب غير مستقر كما في الشكل 3.2.



شكل 3.2: توضيح ترتيب بيانات غير مستقر.

3.3 خوارزميات الترتيب الملائمة وغير الملائمة Adaptive and non-Adaptive Sorting Algorithms

خوارزمية الترتيب يقال عنها مناسبة او ملائمة إذا استفادت من ميزة العناصر المرتبة اصلا في القائمة المراد ترتيبها. عليه، عند الترتيب إذا كانت القائمة الاصلية لها عناصر مرتبة مسبقا، فان الخوارزمية المناسبة سوف تأخذ بنظر الاعتبار هذه العناصر ولا تعيد ترتيبها ثانية.

بينما الخوارزمية غير المناسبة هي الخوارزمية التي لا تأخذ بنظر الاعتبار العناصر التي هي مرتبة مسبقا. فهي تحاول اجبار كل عنصر لإعادة ترتيبه للتأكد من عملية الترتيب.

3.4 مصطلحات مهمة Important Terms

بعض المصطلحات التي صيغت بشكل عام عند مناقشة تقنيات الترتيب:

- **ترتيب تصاعدي Increasing order**

متوالية من القيم يقال عنها مرتبة تصاعديا إذا كان العنصر اللاحق أكبر من العنصر السابق. مثال 1,2,3,4,5 هي مرتبة تصاعديا، حيث كل عنصر هو أكبر من سابقه.

- **ترتيب تنازلي Decreasing order**

متوالية من القيم يقال عنها مرتبة ترتيب تنازلي إذا كان العنصر اللاحق أصغر من العنصر الحالي. كمثال 9, 8, 7, 6, 4, 1 هذه بترتيب تنازلي، حيث كل عنصر في المتوالية أصغر من العنصر الذي يسبقه.

3.5 ترتيب الفقاعة Bubble sort

خوارزمية الفقاعة هي خوارزمية ترتيب بسيطة. خوارزمية الترتيب هذه تعتمد المقارنة بين كل زوج من العناصر المتجاورة ويتم التبديل بين القيمتين حسب

طبيعة الترتيب ان كانتا غير مرتبة (بمعنى إذا كان المطلوب ترتيب تصاعدي فان العنصر الاصغر يجب ان يكون في البداية). هذه الخوارزمية غير مناسبة لمجموعة بيانات كبيرة.

لتوضيح هذه الخوارزمية سنلجأ لمثال بحيث نقترح مجموعة من القيم غير مرتبة ويتم خزنها بمصفوفة (عادة يفضل لترتيب ارقام ان تخزن هذه الارقام بمصفوفة لأنها تسهل العمل) ونحتاج الى ترتيبها ترتيب تصاعدي

قائمة الارقام هي (20, 3, 17, 19, 25, 35, 9, 42, 16, 27)

فسيكون عمل خوارزمية الفقاعة كما يلي:

عملية الترتيب ستتم بعدة مراحل، عدد هذه المراحل غير محدد ويختلف حسب طبيعة القيم المراد ترتيبها، الخوارزمية كما سبق وقلنا ستقارن كل عددين متجاورين وتبدل بينهما إذا كان التبديل ضروري، وتستمر بهذه العملية لحين اكمال جميع القيم في المصفوفة عندها تكون المرحلة الاولى قد انتهت وتبدأ مرحلة جديدة. تنتهي خوارزمية الترتيب عندما نصل الى مرحلة لا يوجد فيها تبديل بين قيمتين.

الان لنناقش المرحلة الاولى لهذه القيم ونستعين بالشكل 3.3, بداية سيتم مقارنة العددين (3 ,20) ولما كانت 3 أصغر من 20 لذلك تتم عملية التبديل لتكون 3 بالموقع الاول و 20 بالموقع الثاني. بعدها تتم مقارنة العددين (17 ,20) وحيث ان 17 أصغر من 20 تتم عملية التبديل.

ملاحظة: عملية المقارنة تبدأ بالعنصر الذي في الموقع (0) من المصفوفة ليقارن مع العنصر الذي في الموقع (1). ثم ينتقل المؤشر الى العنصر في الموقع (1) ليقارن مع العنصر في الموقع (2) .. وهكذا.

الان تقارن (19 ,20) وحيث ان 19 أصغر من 20 فكل واحدة تبدل الى موقع الاخرى.

(25 ,20) هنا 20 اصغر من 25 لذلك لا يوجد تبديل. وكذلك (35 ,25) لا يوجد تبديل.

العددين المتجاورين (35, 9) يتم بينهم التبديل لان 9 أصغر من 35, (35, 42) لا يوجد بينهم تبديل.

(42, 16) تجرى بينهم التبديل لان 16 اصغر من 42.

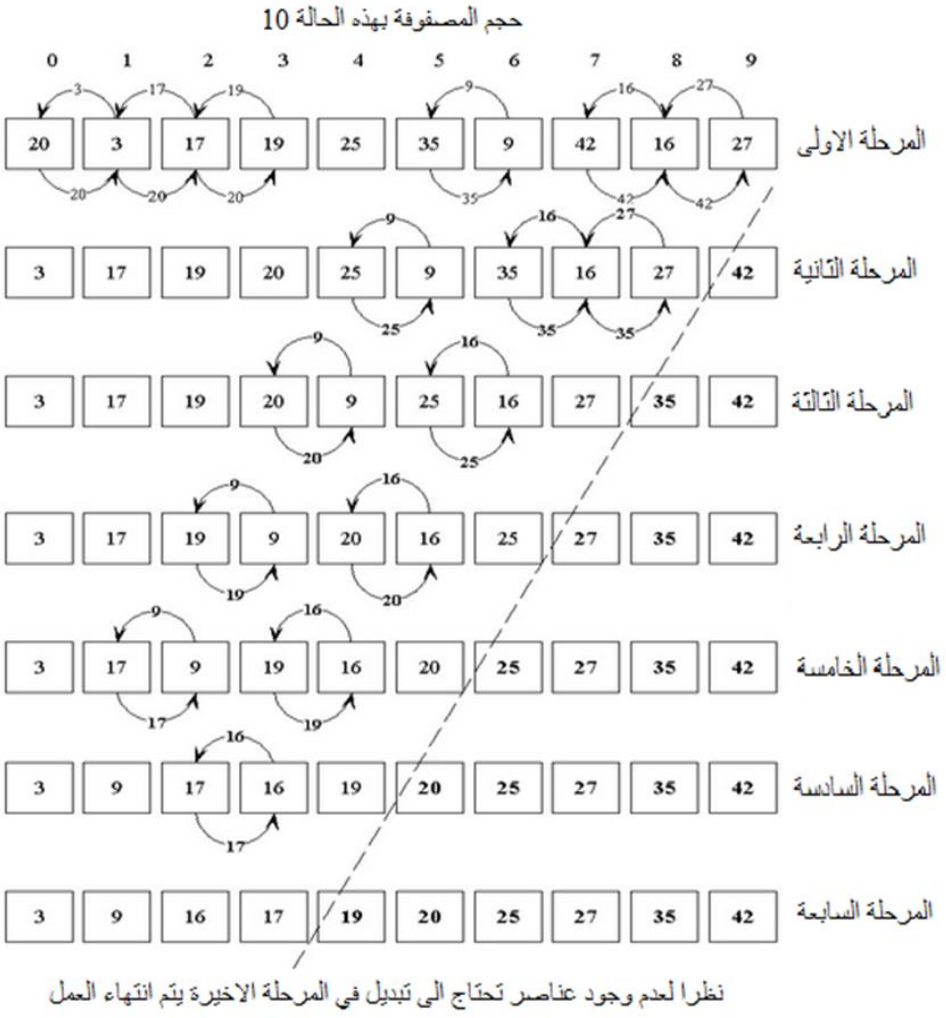
واخر التبديلات في المرحلة الاولى (42, 27) يتم بينهم التبديل لان 27 أصغر من 42.

ونظرا للوصول الى العنصر الاخير في المصفوفة فان هذه المرحلة (المرحلة الاولى) قد انتهت وسنبدأ مرحلة ثانية تتم فيها المقارنة من اول عنصر ونتبع ذات الخطوات اعلاه. تستمر هذه العمليات لحين ان نصل مرحلة لا يوجد فيها تبديل بذلك تكون جميع العناصر قد تم ترتيبها. الشكل 3.3 يوضح الخطوات الكاملة لهذه الخوارزمية.

❖ دالة الترتيب باستخدام خوارزمية الفقاعة.

```
void bubbleSort (int AB[ ] )
{
  for (int i = 0 ; i <= (array_size) ; i++)
  {
    for (int j = 0 ; j < (array_size) ; j++)
    {
      if (AB[ j ] > AB[ j+1])
      {
        int temp = AB[ j ];
        AB[ j ] = AB[ j+1 ];
```

```
        AB[ j+1 ] = temp;
} } } }
```



شكل 3.3: الخطوات الكاملة لترتيب الاعداد وفقا لخوارزمية الفقاعة.

3.6 ترتيب الحشر Insertion Sort

هذه الخوارزمية هي من نوع المقارنة في المكان اي لا تحتاج مساحة خزن اضافية كبيرة. هنا جزء من القائمة يتم المحافظة عليه دائما مرتب. مثال، الجزء الادنى من المصفوفة يحافظ عليه مرتب. العنصر (A) والذي نرغب ان نحشره في القائمة الجزئية المرتبة، يحتاج ان تجد المكان الملائم له ويتم حشره. من هذا جاءت تسمية الخوارزمية بخوارزمية ترتيب الحشر.

يتم بحث المصفوفة بشكل تسلسلي والعناصر غير المرتبة يتم نقلها وحشرها في قائمة جزئية مرتبة (في نفس المصفوفة). هذه الخوارزمية ليست مناسبة لمجموعة البيانات الكبيرة حيث ان معدل درجة التعقيد لها في اسوأ حالة هي ($O(n^2)$), حيث ان (n) تمثل عدد العناصر في القائمة.

الخطوة الاولى في عملية الترتيب بالحشر تعتمد ببساطة على مقارنة اول قيمتين في المصفوفة حيث ستكون القيم الصغيرة في البداية، لذلك إذا كانت القيمة الاولى أكبر من القيمة الثانية في المصفوفة عندها تتم عملية التبديل (ترتيب تصاعدي).

هذه الخطوة هي نواة تكوين القائمة الجزئية المرتبة وهي عادة تكون في بداية المصفوفة، وتكون القائمة الجزئية المرتبة الان تحتوي على عنصر واحد مرتب.

الخطوة الثانية هي مقارنة القيمتين اللاتي بعد القائمة الجزئية المرتبة، إذا كانت القيميتين غير مرتبة فتتم عملية التبديل، مع ملاحظة ان هذه القيمة التي تم ابدالها ستضاف الى القائمة الجزئية المرتبة، ولكي نحافظ على القائمة الجزئية مرتبة، عليه فان القيمة الجديدة تقارن مع جميع عناصر القائمة الجزئية لحين ان تكون في الموقع المناسب لها.

ستلاحظ ان القائمة الجزئية تكبر حجما بمرور الوقت وازدياد عدد المقارنات.

تستمر هذه العملية لحين ترتيب كامل عناصر القائمة.

3.6.1 خطوات خوارزمية الحشر

ان الخطوات اللازمة لعمل هذه الخوارزمية تتلخص بما يلي:

1. إذا كان العنصر هو اول عنصر، فهو سيكون مرتب ويعتبر العنصر الاول في القائمة الجزئية (برمجيا يتم اعادة 1).
2. استدعاء العنصر التالي في المصفوفة لغرض اضافته الى القائمة الجزئية (المرور على كل عناصر المصفوفة بالتوالي).
3. يتم مقارنة هذا العنصر مع كامل عناصر القائمة الجزئية لتحديد موقعه في القائمة الجزئية.
4. يتم تزحيف كل العناصر في القائمة الجزئية المرتبة والتي هي أكبر من القيمة المراد ترتيبها الى اليمين مرتبة واحدة (هنا الترتيب تصاعدي).
5. حشر القيمة التي حصلنا عليها في الخطوة B في الموقع المناسب لها.
6. طالما لم نصل الى نهاية المصفوفة، يتم اعادة الخطوات اعلاه من الخطوة 2.

3.6.2 كيف تعمل هذه الخوارزمية

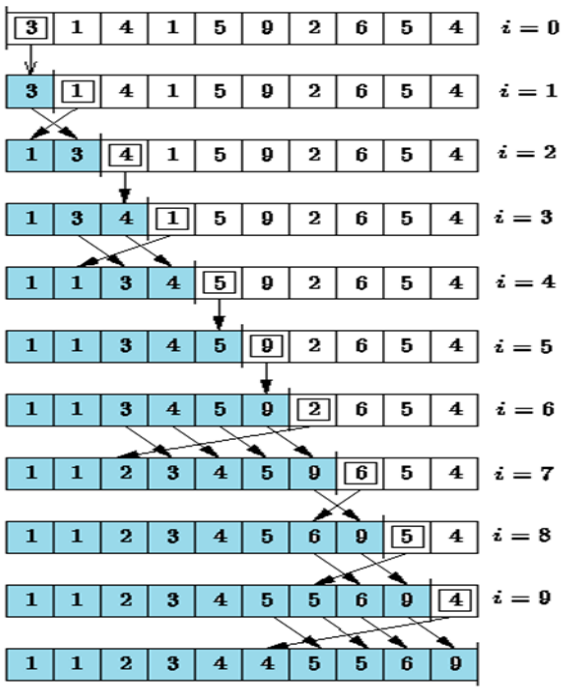
لكي نفهم طريقة الحشر بشكل جيد سنستعين بالشكل 3.4, وهو بداية يمثل مصفوفة من القيم الموزعة بشكل عشوائي، والمطلوب ان يتم ترتيب هذه القيم ترتيب تصاعدي او تزايدي. هنا في اي موقع (i) من المصفوفة (لاحظ ان المتغير (i), الذي يظهر بجانب المصفوفة في الشكل (3.4) يمثل موقع العنصر الذي سيتم مقارنته واضافته الى القائمة الجزئية المرتبة. كذلك فان القائمة الجزئية المرتبة هي تبدأ بالنمو من بداية المصفوفة كما في الشكل 3.4). مثلا في الموقع صفر هناك القيمة 3 وهي اول عنصر في القائمة الجزئية. عندما ننتقل الى الموقع 1 (i=1) حيث سيكون دور العنصر في الموقع 1 للاضافة الى القائمة الجزئية، هنا الموقع 1 في المصفوفة هو ذا قيمة تساوي 1 وتتم مقارنتها مع عناصر القائمة الجزئية لإيجاد المكان المناسب لها في القائمة الجزئية، وحيث ان القائمة الجزئية تحتوي على

القيمة 3 فقط، ونظرا لان القيمة 1 أصغر من 3 لذا تتم عملية تزحيف القيمة 3 الى اليمين بمقدار موقع واحد (المقصود تزحيفه الى الموقع المجاور في المصفوفة). وتصبح القائمة الجزئية المرتبة تحتوي على القيمتين (1, 3).

الان جاء دور (i=2) وفي هذا الموقع القيمة 4, ونظرا الى ان القيمة 4 هي ضمن الترتيب لذلك لا تنفذ عملية التزحيف وتضاف هذه القيمة الى نهاية القائمة الجزئية المرتبة.

عندما تكون (i=3) فان القيمة في هذا الموقع هي 1 لذلك ستقارن مع القيم التي في ذيل المصفوفة الجزئية المرتبة ابتداءا وكل قيمة اكبر منها تزحف لليمين موقع واحد, ستقارن مع القيمة 4 ونظرا لان 1 اصغر من 4 لذا تتم عملية تزحيف القيمة 4 الى الموقع الايمن المجاور لموقعها الحالي وهو الموقع (3) (لاحظ هنا ان موقعها الحالي (2) اصبح خاليا), وكما سبق واشرنا فان هذه القيمة تقارن مع جميع قيم المصفوفة الجزئية لحين ان يتم وضعها او حشرها في موقعها المناسب, لذا ستقارن مع القيمة 3 التي هي في الموقع 1, وايضا تجرى عملية التزحيف حيث تزحف القيمة 3 الى الموقع الخالي على يمينها (2) ويصبح موقعها (1) هو الفارغ او الخالي, وعند مقارنة القيمة 1 مع القيمة 1 فستكون القيمتان متساويتان لذلك توضع القيمة 1 في الموقع الخالي (1).

هذه العملية تستمر لحين الوصول الى نهاية القائمة وبالتالي سنحص على قائمة من القيم مرتبة ترتيب تصاعدي.



شكل 3.4: شكل توضيحي لتنفيذ خوارزمية الحشر.

❖ الدالة البرمجية لخوارزمية الحشر هي

```
void insertionSort(int AB[ ] )
{
  for (int i=1; i ‹ array_size ; i++)
  {
    int index = AB[ i ] ;     int j = i ;
    while (j > 0 && AB[ j-1 ] > index)
    {
        AB[ j ] = AB[ j-1 ];
```

```
        j--;
    }
    AB[ j ] = index;
} }
```

3.7 ترتيب الاختيار Selection Sort

ترتيب الاختيار هي خوارزمية بسيطة. هذه الخوارزمية تعتمد على المقارنة في المكان، حيث ان قائمة القيم (المصفوفة) يتم تقسيمها الى جزئين، الجزء الايسر (النهاية اليسرى من المصفوفة) وهو جزء القيم المرتبة، والجزء الايمن (النهاية اليمنى من المصفوفة) وهو جزء القيم غير المرتبة. ابتداء فان جزء القيم المرتبة سيكون فارغ، بينما جزء القيم غير المرتبة يمثل جميع عناصر القائمة.

يتم بحث القائمة غير المرتبة لإيجاد أصغر قيمة (في حالة الترتيب التصاعدي) ويتم ابدالها مع العنصر في اقصى اليسار من القائمة غير المرتبة، وبهذا سيكون هذا العنصر هو واحد من عناصر القائمة المرتبة. هذه العملية تستمر بنقل أحد عناصر القائمة غير المرتبة الى القائمة المرتبة (بمعنى ان القائمة المرتبة سيزداد حجمها مقابل تقلص حجم القائمة غير المرتبة).

واقعا ان العملية ببساطة تتم باختيار اول عنصر من القائمة غير المرتبة (العنصر في اقصى يسار القائمة غير المرتبة) ومقارنته بجميع عناصر القائمة غير المرتبة، إذا وجدنا عنصر أصغر منه يتم ابداله ويبقى معنا العنصر الاصغر، فاذا وصلنا الى نهاية القائمة غير المرتبة تتم اضافة هذا العنصر (الذي كان العنصر الاول في القائمة غير المرتبة والذي قيمته أصغر من قيم باقي العناصر في القائمة غير المرتبة) الى نهاية القائمة المرتبة (اي في النهاية اليمنى للقائمة المرتبة).

3.7.1 خطوات خوارزمية ترتيب الاختيار

A. تحجز القيمة التي في اول موقع من مواقع المصفوفة (الموقع صفر في بداية العمل) في متغير لنفرض اسمه (Min)، هذا يعني ان المتغير (Min) سيحفظ القيمة التي في الموقع صفر من المصفوفة على فرض ان هذه القيمة هي أصغر قيمة في المصفوفة او القائمة المرتبة.

B. يقارن هذا العنصر مع جميع العناصر الاخرى في المصفوفة. كلما وجدت قيمة أصغر من (Min) تتم عملية التبديل مع القيمة التي في (Min)(نؤكد على فرض الترتيب التصاعدي). ليكون في نهاية بحث المصفوفة العنصر الاول في القائمة المرتبة

C. لاحقا يتم اختيار العنصر اللاحق ليحجز في المتغير (Min) (يعتبر العنصر الاول في القائمة غير المرتبة).

D. يتم بحث جميع العناصر الباقية في المصفوفة (غير المرتبة) وتقارن مع (Min).

E. كلما وجدت قيمة أصغر من (Min) تتم عملية التبديل مع القيمة التي في (Min).

F. عند الوصول الى نهاية القائمة غير المرتبة يتم اضافة العنصر (Min) الى ذيل القائمة المرتبة (اي اقصى يمين القائمة المرتبة).

G. إذا لم يتم الوصول الى العنصر قبل الاخير في المصفوفة (القائمة غير المرتبة) ومعالجتة (باعتباره الاصغر ويقارن مع العنصر الاخير)، تعاد هذه العملية للخطوات السابقة من الخطوة C, لحين ان يتم ترتيب كامل المصفوفة.

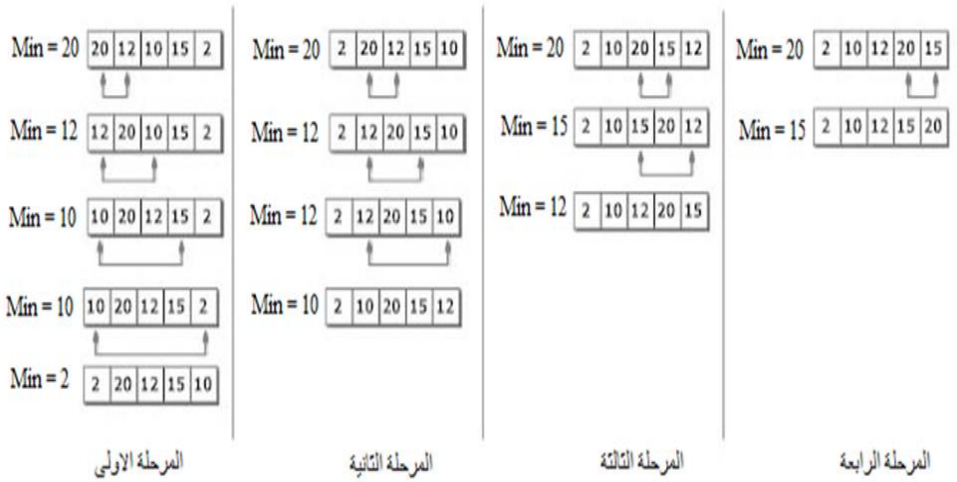
3.7.2 كيف تعمل الخوارزمية

في هذه الخوارزمية فان عملية الترتيب تتم بعدد من المراحل تصل الى (n – 1) حيث ان (n) تمثل عدد العناصر في المصفوفة.

سنوضح عمل هذه الخوارزمية باستخدام الشكل 3.5, في المرحلة الاولى سيتم اختيار العنصر الاول ويفرض على انه الاصغر ولذلك يوضع بمتغير لنقل اسمه (Min)، هذا العنصر الذي قيمته 20 ستتم مقارنته مع جميع العناصر الاخرى في المصفوفة، واول عنصر هو العنصر الذي قيمته 12, ولما كانت 12 أصغر من 20 لذلك ستتم عملية التبديل ويوضع كل عنصر بموقع الاخر (بمعنى ان التبديل يتم بين موقع المتغير (Min) (وهو الموقع صفر) وموقع العنصر الذي قيمته أصغر وهو الموقع الاول. بهذا، ان المتغير Min سيحمل القيمة 12 بدل من 20 بعد اجراء التبديل). تستمر عملية المقارنة وسيكون العنصر الاخر الذي يقارن مع Minهو 10, ولما كانت 10 أصغر من 12 لذا تتم عملية التبديل بحيث كل عنصر يحتل موقع العنصر الاخر، والمتغير Min ستكون قيمته هي 10.

تستمر عملية المقارنة وهذه المرة تكون المقارنة مع القيمة 15 وحيث ان 15 أكبر من 10 تبقى كل قيمة على وضعها ولا تجرى عملية تبديل. واخيرا جاء دور القيمة الاخيرة والتي قيمتها 2 وتقارن مع Min, ونظرا لان 2 أصغر من 10 تتم عملية التبديل ولتكن كل قيمة بموقع الاخرى كما في الشكل 3.5, ويصبح المتغير Min يحمل القيمة 2.

بهذا انتهت المرحلة الاولى، وحان دور المرحلة الثانية والتي في هذه الحالة سيتم وضع القيمة في الموقع اللاحق وهو موقع 1 في المصفوفة في المتغير Min, وهي القيمة 20. وكما في المرحلة الاولى، تعاد نفس الخطوات لعمليات المقارنة والتبديل، وهكذا لكل المراحل اللاحقة وكما واضح في الشكل 3.5.



شكل 3.5: مخطط توضيحي لتنفيذ خوارزمية الترتيب بالاختيار.

❖ الدالة البرمجية لخوارزمية الاختيار هي

```
void selectionSort (int AB[ ] ) {
  for (int i = 0 ; i ‹ array_size – 1 ; i++)
  {
    int min = i;
    for (int j = i+1 ; j ‹ array_size ; j++)
        if (AB [ j ] < AB [min])
            min = j ;
     int temp = AB[ i ];
    AB [ i ] = AB [min];
    AB [min] = temp;
} }
```

3.8 الترتيب بالدمج Merge Sort

ترتيب الدمج هي تقنية تعتمد على تقنية (divide and conquer)، وهي واحدة من الخوارزميات المعتبرة.

مع اسوأ تعقيد وقت فان ((complexity being O (n log n)).

ترتيب الدمج يقسم المصفوفة الى نصفين متساويين وبعدها تتم عملية دمجهم بطريقة مرتبة.

الخوارزمية ببساطة تقسم المصفوفة بشكل متكرر الى نصفين، ثم تقسم الناتج الى نصفين وهكذا لحين الحصول على عناصر صغيرة لا يمكن تقسيمها أكثر. عملية التقسيم هذه تحافظ على ترتيب هذه العناصر في المصفوفة الاصلية.

بعد عملية التقسيم تتم عملية الدمج بنفس طريقة التقسيم اي ان كل عنصرين متجاورين ناتجة من التقسيم تدمج ثانية لكن وفق شرط وهو ان تكون هذه العناصر مرتبة، بمعنى ان تتم مقارنة قيم هذين العنصرين وترتب ترتيب تصاعدي او تنازلي (اجراء التبديل بينهم إذا استوجب) ومن ثم تدمج.

في عملية الدمج الثانية تدمج كل قائمتين متجاورتين مع بعضهما (هنا كل قائمة لها عنصرين)، مع الاخذ بنظر الاعتبار بترتيب قيمهم. وهكذا تستمر العملية لحين اعادة الدمج لكامل القائمة.

3.8.1 خطوات خوارزمية الترتيب بالدمج

A. إذا كانت القائمة تحتوي على عنصر واحد، فان القائمة مرتبة ويتم اعادتها كقائمة مرتبة.
B. تقسيم القائمة بشكل متكرر الى نصفين متساويين لحين الوصول الى حالة لا يمكن معها اجراء تقسيم اضافي.
C. دمج القوائم الصغيرة بقوائم جديدة بحيث تكون عناصرها مرتبة.

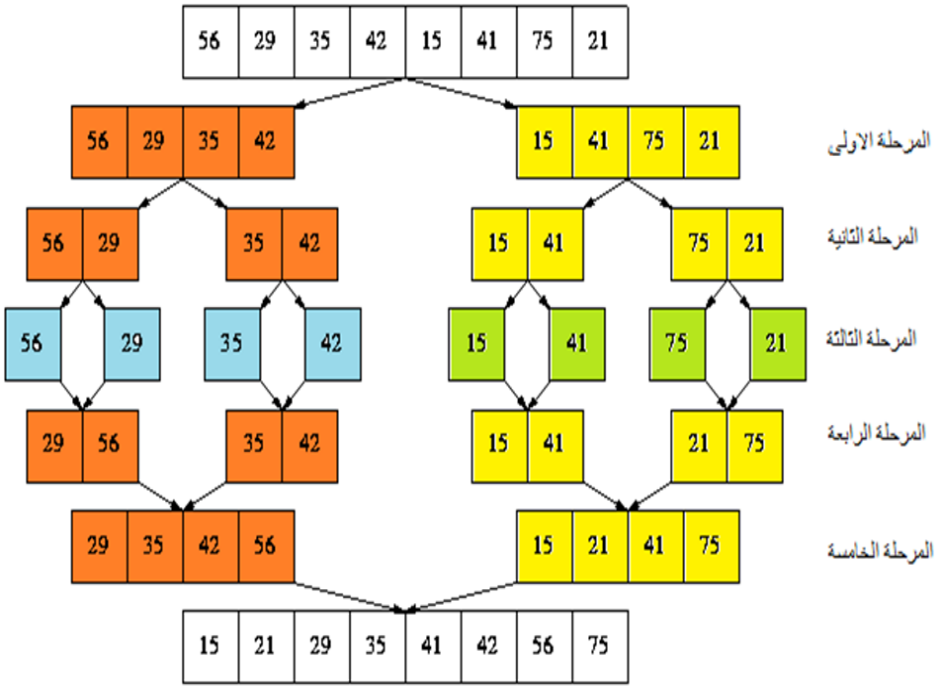
3.8.2 كيف تعمل الخوارزمية

في الشكل 3.6 مصفوفة من الاعداد غير المرتبة. لكي ننفذ عليها طريقة الدمج نبدأ بتقسيم المصفوفة والتي تتكون من 8 عناصر الى نصفين، بحيث يكون كل نصف يحتوي على 4 عناصر كما واضح في المرحلة الثانية من الشكل 3.6. وفي المرحلة الثالثة ايضا يتم التقسيم لنحصل على أربع مجاميع كل منها يحتوي على عنصرين فقط. وفي المرحلة الرابعة نقسم جميع المجاميع وسنحصل على ثمان مجاميع كل منها يحتوي عنصر واحد، ونظرا الى ان هذه المجاميع غير قابلة للانقسام اكثر لذلك يتوقف التقسيم.

الان حان دور الدمج وبنفس طريقة التقسيم وبشكل عكسي تتم عملية دمج كل عنصرين متجاورين مع ضرورة ان تكون العناصر في مرحلة الدمج مرتبة (هنا ترتيب تصاعدي).

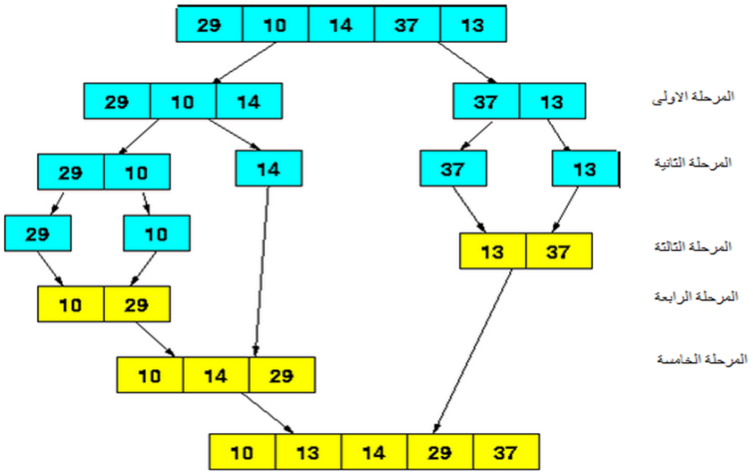
لاحظ سيتم دمج (56, 29) ونظرا لان هذين العنصرين غير مرتبة لذلك تتم عملية التبديل ثم الدمج وسنحصل على (29, 56)، نفس الشيء للعناصر التي بعدها وهم (35, 42) يتم دمجهم دون الحاجة للتبديل نظرا لأنها مرتبة. العنصرين الاخرين هم (15, 41) تدمج دون تبديل لأنها مرتبة. العناصر (75, 21) يتم ترتيبهم ثم دمجهم لتكون هذه المجموعة (21, 75). بهذا تكون المرحلة الرابعة قد انجزت.

في المرحلة الخامسة يتم دمج المجاميع الناتجة والتي عددها أربع مجاميع لنكون منهم مجموعتين، في هذه المرحلة ايضا من المفروض ان يتم التبديل بين العناصر الاربع إذا استوجب ذلك. واخيرا في المرحلة السادسة يتم دمج المجموعتين لنحصل على قائمة مرتبة.



شكل 3.6: مخطط توضيحي لطريقة تنفيذ خوارزمية الدمج.

ملاحظة: عندما يكون عدد عناصر المصفوفة ليس زوجي وبالتالي فان عملية التقسيم لا تتم بتقسيم المصفوفة الى قسمين متساويين كما في الشكل 3.6, لذلك في هذه الحالة سيتم التقسيم الى قسمين غير متساوين وتنجز كل الاعمال بشكل اعتيادي كما سبق ووضحنا، الشكل 3.7 يوضح هذه الحالة.



شكل 3.7: مخطط توضيحي لتنفيذ خوارزمية الدمج عندما يكون عدد عناصر المصفوفة فردي.

❖ الدالة البرمجية لخوارزمية الترتيب بالدمج

```
// arr[] دمج اثنان من المصفوفات الثانوية للمصفوفة
// arr[l..m] المصفوفة الثانوية الاولى هي
// arr[m+1..r] المصفوفة الثانوية الثانية هي
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m ;
    int n2 =  r - m;

    /* خلق مصفوفات مؤقتة */
    int L[n1], R[n2];
```

```
/* L[ ] and R[ ] نسخ البيانات الى المصفوفات المؤقتة*/
for (i = 0; i < n1; i++)
   L[i] = arr[l + i];
for (j = 0; j < n2; j++)
   R[j] = arr[m + 1+ j];

/* arr[l..r] دمج المصفوفات المؤقتة في المصفوفة*/
i = 0;     // ابتداء الدليل (الفهرس) الى المصفوفة الثانوية الاولى
j = 0;     // ابتداء الدليل (الفهرس) الى المصفوفة الثانوية الثانية
k = l;     // ابتداء الدليل (الفهرس) الى مصفوفة الدمج الثانوية
while (i < n1 && j < n2)
{
   if (L[i] <= R[j])
   {
      arr[k] = L[i];
      i++;
   }
   else
   {
      arr[k] = R[j];
      j++;
   }
```

```
        k++;
    }

    /* استنساخ العناصر المتبقية للمصفوفة L[], اذا كان هناك عناصر */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    استنساخ العناصر المتبقية للمصفوفة R[], اذا كان هناك عناصر*/
      are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

    */ arr هو الترتيب المطلوب للمصفوفة الثانوية r هو الفهرس الايمن, l هو الفهرس الايسر /*
void mergeSort(int arr[], int l, int r)
```

```
{
   if (l < r)
   {
      // Same as (l+r)/2, but avoids overflow for
      // large l and h
      int m = l+(r-l)/2;

      // ترتيب النصف الاول والثاني //
      mergeSort(arr, l, m);
      mergeSort(arr, m+1, r);
       merge(arr, l, m, r);
   }
}
```

3.9 الترتيب بطريقة شل Shell Sort

ترتيب شيل تم تسميتها باسم مكتشفها. فبدلا من مقارنة العناصر المتجاورة مثل خوارزمية الفقاعة, فان ترتيب شيل يكرر عملية المقارنة للعناصر التي يفصل بينها مسافة معينة (لنفرض d تمثل المسافة). ان قيمة d الابتدائية تبدا بما يساوي نصف حجم المصفوفة المدخلة وفي كل دورة يتم تنصيف هذه المسافة. العناصر تتم مقارنتها وتبدل عند الحاجة. العلاقة التالية يتم استخدامها

$$d = (N + 1) / 2$$

ان قيمة d دائما هي عدد صحيح, وان N تمثل عدد عناصر المصفوفة.

هذه الخوارزمية كفوءة لمجاميع البيانات متوسطة الحجم وهي في اسوأ الحالات تكون درجة التعقيد لها ((complexity is O(n)) حيث ان n تمثل عدد العناصر.

3.9.1 خطوات خوارزمية شل

A. احسب قيمة (d)، وفقا للعلاقة d = (N-1) / 2.
B. قسم القائمة الى مجاميع ثانوية صغيرة بمديات تساوي d.
C. رتب هذه المجاميع الثانوية باستخدام خوارزمية الحشر.
D. كرر هذه الاعمال لحين ان يتم ترتيب القائمة.

3.9.2 كيفية عمل الخوارزمية

عملية الترتيب بطريقة شل تعتمد اساسا على حساب قيمة (d) والتي يتم حسابها في كل مرحلة، ان هذه المسافة سيتم الاعتماد عليها لتحديد العناصر التي يتم مقارنتها مع بعض. لتبسيط الموضوع دعنا نرى الشكل 3.8 والذي يمثل مصفوفة من 6 عناصر موزعة بشكل عشوائي. بداية يتم حساب قيمة المسافة (d). في المرحلة الاولى تحسب المسافة وفقا للعلاقة ادناه، اما في المراحل اللاحقة فيتم تنصيف قيمة المسافة في كل مرحلة (كل مرحلة لها قيمة مسافة تساوي نصف قيمة مسافة المرحلة السابقة).

$$d = (N + 1) / 2$$

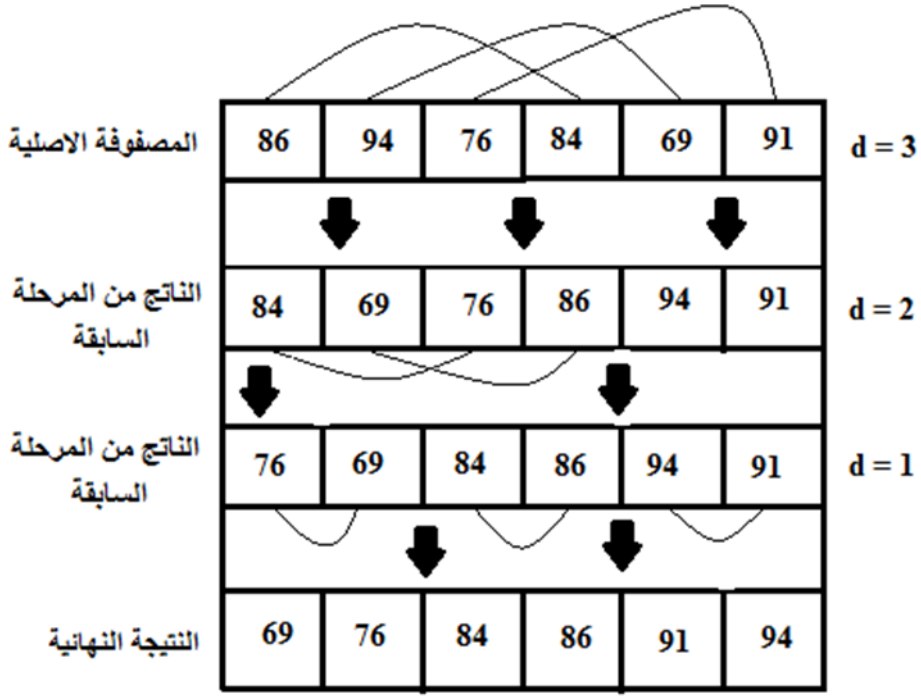
$$d = (6 + 1) / 2 = 3$$

هذا يعني ان العنصر الاول سيتم مقارنته مع العنصر الرابع، والعنصر الثاني يقارن مع العنصر الخامس، والعنصر الثالث يقارن مع العنصر السادس.

عند مقارنة العنصر الاول مع العنصر الرابع (84 ,86) نلاحظ ان 84 أصغر من 86 لذلك يتم التبديل بحيث كل عنصر يبدل الى موقع العنصر الاخر (ترتيب تصاعدي).

مقارنة العنصر الثاني مع العنصر الخامس (69 ,94) ايضا تتم عملية تبديل وذلك لان 69 أصغر من 94.

الحالة الاخرى مقارنة العنصر الثالث مع العنصر السادس (91 ,76) هنا كل عنصر يبقى بمكانه دون تبديل لان العنصرين مرتبين.



شكل 3.8: مخطط يوضح كيفية تنفيذ خوارزمية شل.

❖ الدالة البرمجية لخوارزمية الترتيب بطريقة شل

```
for (increment = size/2;increment > 0; increment /= 2)
{
   for(i = increment; i<size; i++)
   {
      temp = array[i];
      for(j = i; j >= increment ;j-=increment)
```

```
    {
      //perform the insertion sort for this section
      if (temp < array[j-increment])
      {
        array[j] = array[j-increment];
      }
      else
      {
        break;
      }
    }
    array[j] = temp;
  }
}
```

3.10 الترتيب السريع Quick Sort

هذه الخوارزمية ربما تكون الاكثر شيوعا واستخداما، حيث انها عادة الاسرع عمليا.

خوارزمية الترتيب السريع هي خوارزمية كفوءة جدا وتعتمد على تقسيم مصفوفة البيانات الى مصفوفات صغيرة. المصفوفة الكبيرة تقسم الى مصفوفتين واحدة تحمل قيم صغيرة واخرى تحمل قيم كبيرة، وعندما نقول صغيرة وكبيرة هي نسبة الى قيمة مقترحة، فالقيم التي أكبر من هذه القيمة المقترحة توضع بمصفوفة والقيم التي اقل من هذه القيمة المقترحة توضع بمصفوفة اخرى.

فاذا كانت لديك مصفوفة كبيرة، فيجب اخذ قيمة عشوائية من المصفوفة والتي بموجبها سيتم تجزئة المصفوفة. عندما يتم اختيار هذه القيمة فعليك ان تقارن جميع القيم الباقية في المصفوفة مع هذه القيمة. القيم التي هي أكبر من القيمة المختارة من الممكن وضعها في جزء المصفوفة الايمن، بينما القيم التي أصغر من الممكن وضعها في جزء المصفوفة الايسر.

هذه الخوارزمية كفوءة جدا لمجموعة البيانات الكبيرة، علما بان اسوأ حالة لها لتعقيد الوقت هو ((Time complexity is O (n log n) حيث ان n هو عدد العناصر في المصفوفة.

3.10.1 خطوات خوارزمية الترتيب السريع

A. اختيار القيمة ذات الفهرس الاعلى في المصفوفة (اخر قيمة بالمصفوفة) لتعتبر قيمة المحور (pivot) (احيانا يتم اختيار القيمة التي في وسط المصفوفة لتعتبر قيمة المحور.
B. اختيار متغيرين للإشارة الى الجانب الايمن والايسر من القائمة باستثناء المحور.
C. المؤشر الايسر يؤشر الى فهرس index المصفوفة الادنى.
D. المؤشر الايمن يؤشر الى فهرس المصفوفة الاعلى.
E. يتم تحريك المؤشر الايسر الى اليمين بواقع موقع واحد في المصفوفة كل مرة طالما القيمة التي يؤشر عليها المؤشر الايسر هي أصغر من قيمة المحور. ويقف المؤشر عند وجود قيمة اعلى من قيمة المحور.
F. المؤشر الايمن يتحرك موقع واحد في كل مرة باتجاه اليسار طالما قيم المواقع التي يمر عليها أكبر من قيمة المحور ويتوقف المؤشر على القيمة التي هي اقل من قيمة المحور.
G. إذا كلا المؤشرين في الخطوات (E, F) توقفا على موقعين مختلفين (اي ان المؤشر الايسر أصغر من المؤشر الايمن)، عند ذلك يتم تبديل القيم في هذين الموقعين.
H. إذا كانت قيمة المؤشر الايسر (رقم موقعه في المصفوفة) أكبر او تساوي من قيمة المؤشر الايمن، فان نقطة الالتقاء ستكون هي قيمة المحور.

استخدام خوارزمية المحور بشكل متكرر بطريقة الاستدعاء الذاتي سيوصلنا الى اجزاء صغيرة محتملة. كل جزء يعالج بطريقة الترتيب السريع.

ان خوارزمية الاستدعاء الذاتي للترتيب السريع هي:

a) اجعل قيمة الفهرس في اقصى اليمين (اي القيمة العليا) على انه المحور. من الممكن اختيار المحور على انه معدل القيمتين العليا والدنيا
   a. Pivot = (High + Low) / 2
b) قسم المصفوفة اعتمادا على قيمة المحور.
c) رتب الجزء الايسر بطريقة الاستدعاء الذاتي لخوارزمية الترتيب السريع.
d) رتب الجزء الايمن بطريقة الاستدعاء الذاتي لخوارزمية الترتيب السريع.

3.10.2 كيفية عمل خوارزمية الترتيب السريع

لكي يتم تنفيذ خوارزمية الترتيب السريع سيتم الاستعانة بالشكل 3.9 لمتابعة حركة البيانات.

بداية لابد من تحديد قيمة المحور (pivot) وهناك أكثر من طريقة لتحديد هذه القيمة، في هذا المثال سيتم اختيار ثلاث قيم ونختار القيمة الوسطى من بين هذه القيم الثلاث، القيم الثلاث اللائي سيتم اختيارهن هن القيمة الاولى بالمصفوفة، والقيمة الاخيرة بالمصفوفة، فضلا عن القيمة الوسطى في المصفوفة.

في الشكل 3.9 تم اختيار مجموعة من القيم موزعة عشوائيا، وتم خزنها في مصفوفة ومن هذه القيم سيتم اختيار القيم (24 ,81 ,57) ونظرا لان القيمة 57 هي القيمة الوسطى بين القيمتين الاخريين، لذلك سيتم اختيارها على انها قيمة المحور.

توضع قيمة المحور بمتغير خارجي، وتحرك القيمة (24) الى موقع المحور الذي تم اختياره (اي تنقل القيمة 24 الى الموقع صفر في المصفوفة وذلك لأنها أصغر من قيمة المحور)، ويبقى الموقع الاخير خالي لحين انتهاء مرحلة تقسيم المصفوفة.

ولغرض ترتيب هذه القيم ترتيب تصاعدي فان أحد اهداف الخوارزمية هو تقسيم المصفوفة الى قسمين، القسم الاول الذي يمثل المواقع ابتداءا من الموقع (0) وصعودا سيحتوي على القيم التي هي أصغر من قيمة المحور. والقسم الثاني الذي

سيكون بالاتجاه المعاكس اي يبدأ من اخر موقع في المصفوفة ويتجه باتجاه اليسار اي نزولا باتجاه بداية المصفوفة سيحتوي على القيم التي هي أكبر من قيمة المحور. لتنفيذ ذلك سنضع مؤشرين أحدهما في بداية القائمة والاخر على اخر عنصر في القائمة. كما في الشكل 3.9.

المؤشر الايمن سيتحرك الى اليسار كما سبق وأشرنا ليمر على عناصر المصفوفة واحد بعد الاخر، فاذا كان العنصر الذي يمر عليه أكبر من المحور يتركه ويستمر بالحركة، وإذا كان أصغر من قيمة المحور يقف عليه، بالمقابل المؤشر الايسر يتحرك باتجاه اليمين ليتوقف عند اول عنصر يصادفه أكبر من قيمة المحور.

في مثالنا هذا، يبدأ المؤشرين بالحركة فيتحرك المؤشر الايمن ويقف عند اول قيمة تصادفه وهي القيمة (49) والتي هي أصغر من قيمة المحور. بالمقابل فان المؤشر الايسر ايضا يتحرك ويقف عند القيمة (70) وذلك لأنها أكبر من قيمة المحور. عليه سيتم ابدال هاتين القيميتين واحدة بمكان الاخرى.

بعد اجراء التبديل تستمر حركة المؤشرات ويتحرك المؤشر الايمن ليقف عند القيمة (16) لأنها أصغر من قيمة المحور، بينما المؤشر الايسر يقف عند القيمة (97) لأنها أكبر من قيمة المحور. تجرى عملية التبادل بينهم. وتستمر حركة المؤشرات فيقف المؤشر الايمن عند القيمة (55) لأنها أصغر من قيمة المحور، والمؤشر الايسر سيقف عند القيمة (63) كونها أكبر من قيمة المحور وايضا تتم عملية التبديل. تستمر هذه العملية كما واضح في الشكل 3.9 لحين ان يصل المؤشر الايمن الى القيمة (9) والمؤشر الايسر يصل الى القيمة (76)، وحيث ان قيمة المؤشر الايمن (اي رقم الموقع الذي يقف عليه في المصفوفة) أصغر من قيمة المؤشر الايسر لذلك تتوقف الخوارزمية.

وبهذا فأننا قسمنا المصفوفة الى قسمين تكون قيم القسم الاول الذي في بداية المصفوفة هي أصغر من قيم عناصر القسم الثاني الذي هو في نهاية المصفوفة كما واضح من الشكل 3.9.

خطوة واحدة اضافية وهي ان القيمة الكبيرة التي يؤشر عليها المؤشر الايسر ستوضع في الموقع الاخير الفارغ، ليصبح موقعها خالي حيث ستوضع به قيمة المحور والذي سيكون في موقعه الصحيح.

بعد هذه المرحلة سيتم استدعاء خوارزمية الترتيب السريع لترتيب قيم النصف الاول من المصفوفة والتي قيمها اقل من (57) (طبعا هذا الاستدعاء هو استدعاء ذاتي ويتم تكراره لحين اكمال ترتيب كل العناصر).

ايضا سيتم اختيار ثلاث قيم لتحديد قيمة المحور (pivot) لهذا الجزء من المصفوفة، هنا القيم الثلاث هي (9 ,38 ,24) وبذلك فستكون قيمة المحور تساوي 24.

ونظرا لان القيم 9 أصغر من قيمة المحور فسيتم نقلها الى الموقع صفر ويبقى موقعها فارغ لحين انتهاء عملية ترتيب هذا الجزء من المصفوفة.

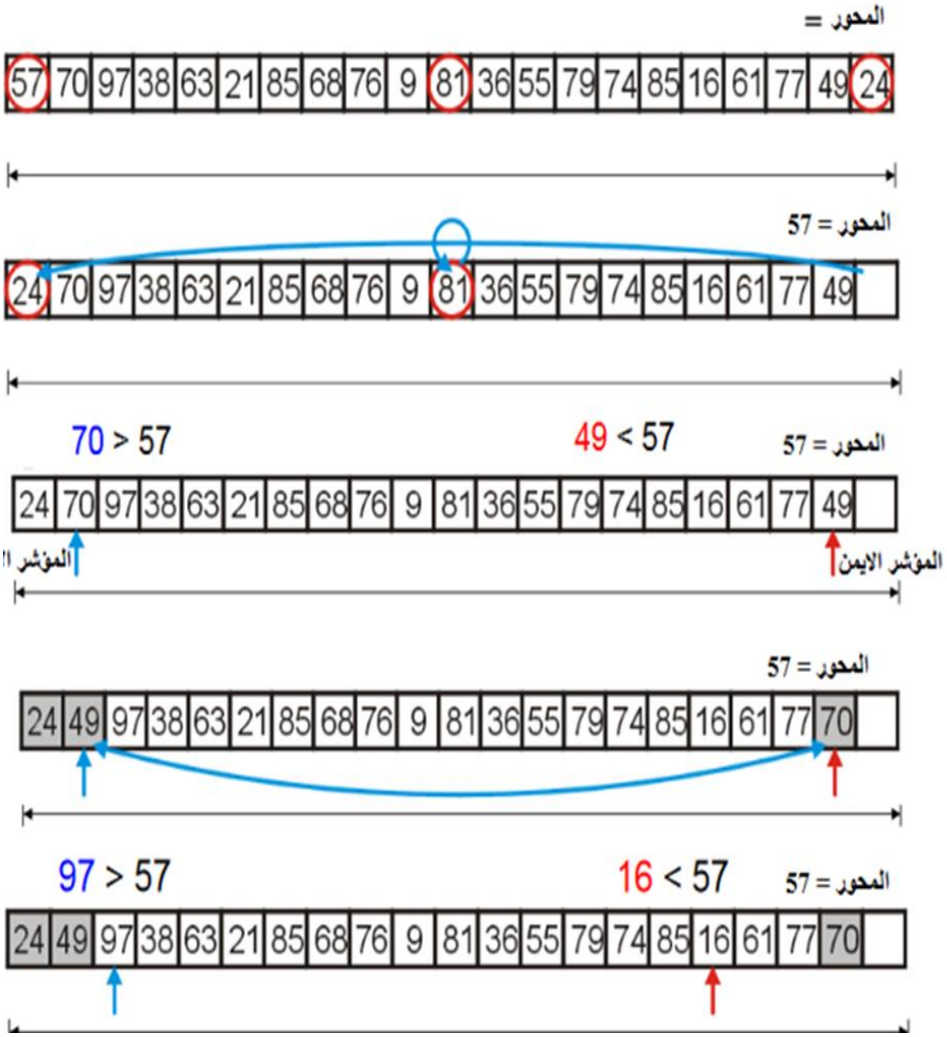
ايضا يستخدم مؤشرين ايمن وأيسر فيتحرك المؤشر الايمن ليقف عند القيمة (21), بينما يقف المؤشر الايسر بعد ان يتحرك على القيمة (49), يتم ابدال القيم (49 ,21) واحد بدل الاخر. يستمر المؤشران بالحركة ليقف المؤشر الايمن على القيمة (16)، ويقف المؤشر الايسر على القيمة (38)، وتجرى عملية التبديل.

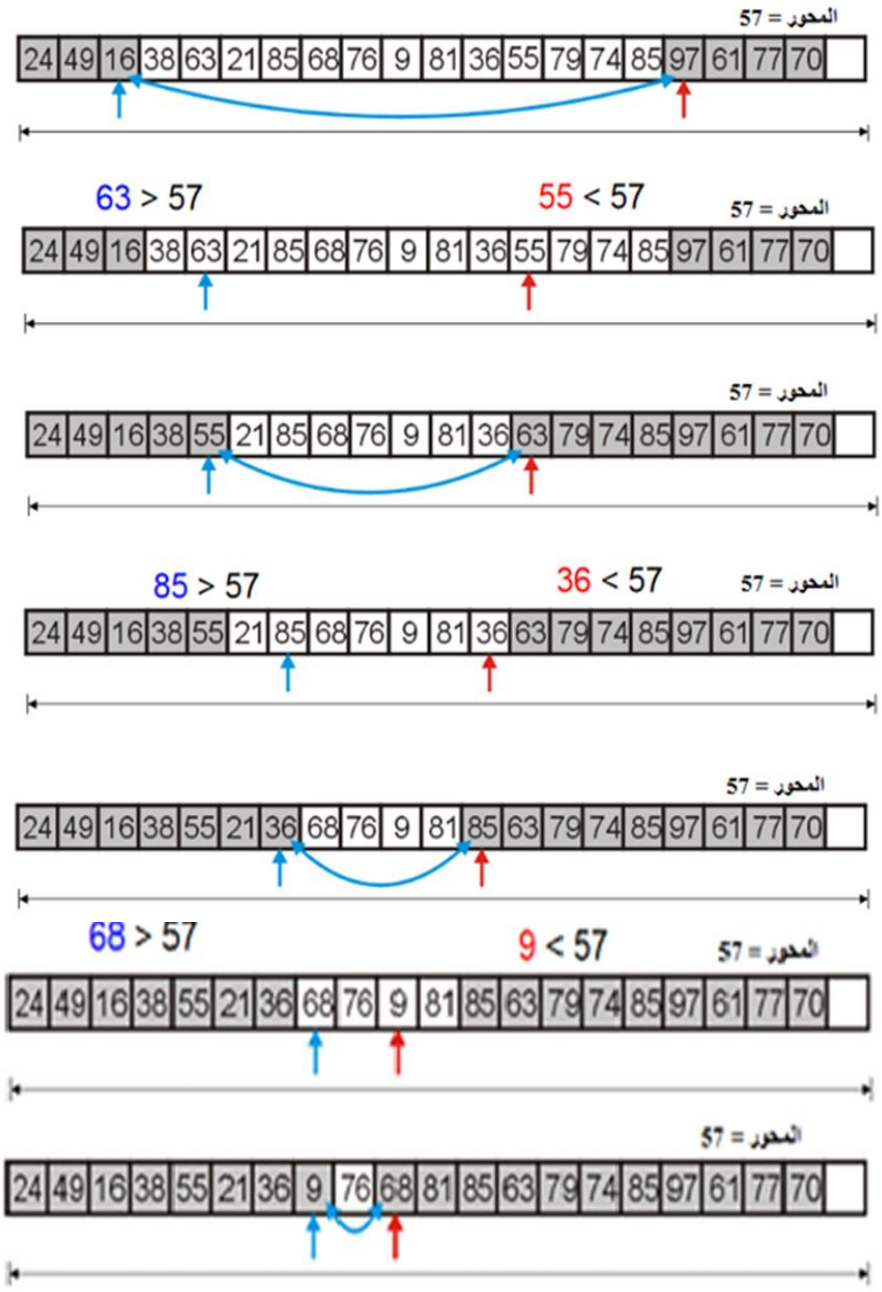
لاحظ هنا ان المؤشرين أصبحا بموقعين متعاكسين بمعنى ان قيمة المؤشر الايسر أكبر من قيمة المؤشر الايمن، لذلك نتوقف.

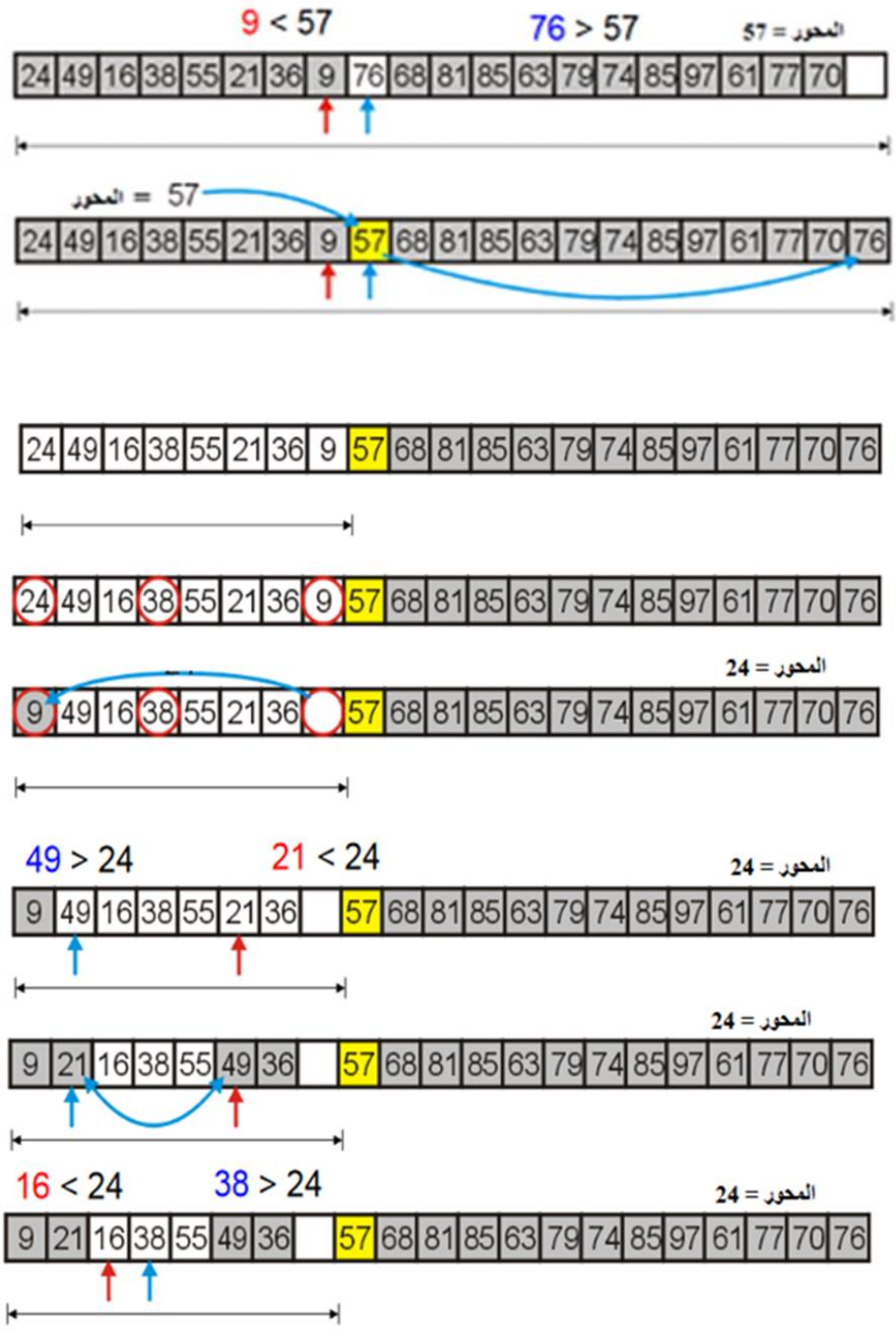
في هذه الحالة ونظرا لان القيمة التي يؤشر عليها المؤشر الايسر كبيرة فسيتم تحريكها ونقلها الى الموقع الخالي في نهاية هذا القسم من المصفوفة، ويتم وضع قيمة المحور في مكانها الذي أصبح خاليا.

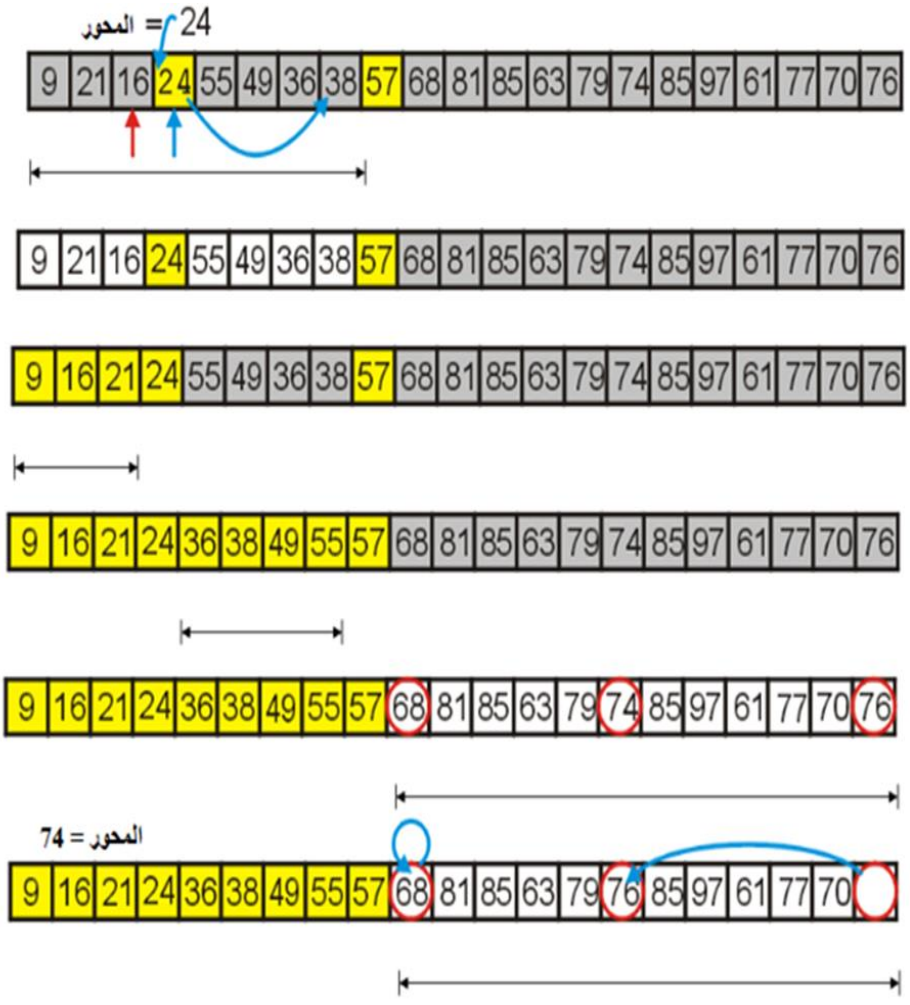
الان سيتم استدعاء دالة الترتيب السريع على النصفين المتولدين اخيرا من نصف المصفوفة التي قيمها أصغر من 57, وتعاد نفس الخطوات لأجل ترتيبها.

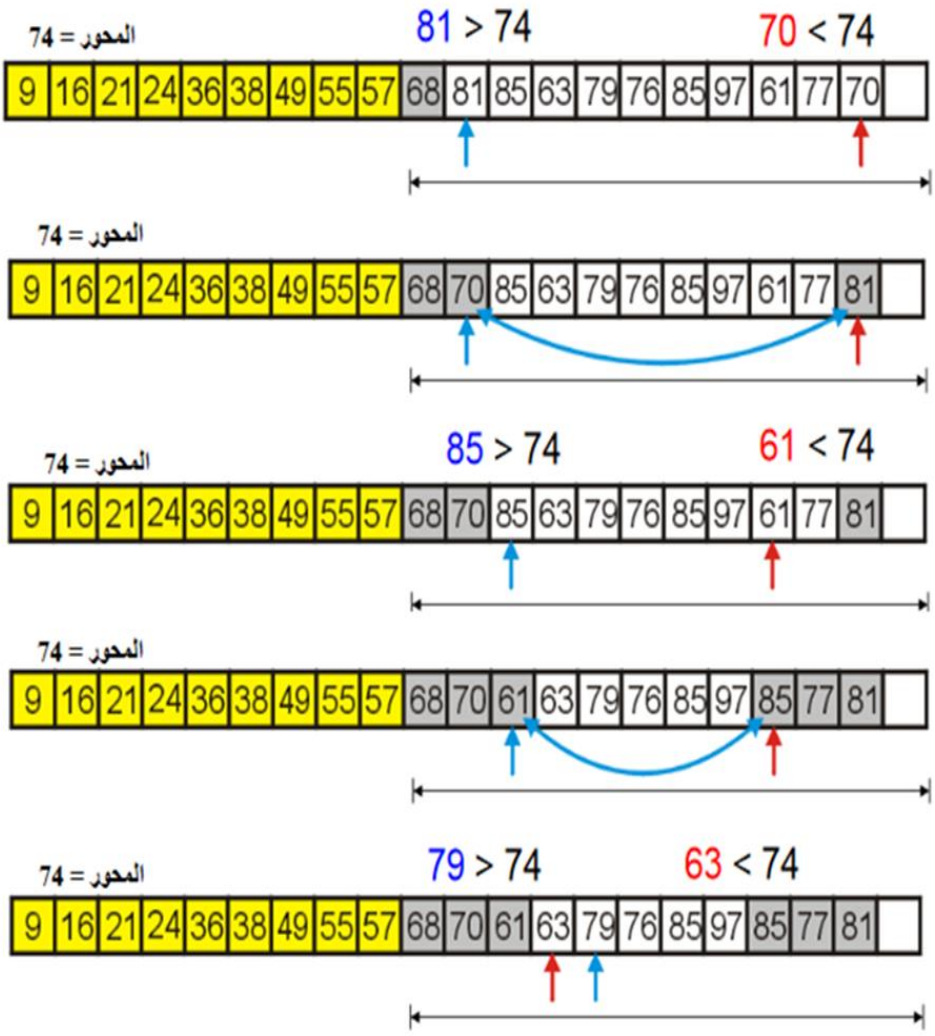
نفس الشيء يتم بالنسبة لعناصر القسم الثاني من المصفوفة والتي قيمها أكبر من 57 لغرض ترتيب عناصره. متابعة الشكل 3.9 يساعد على فهم أفضل للخوارزمية.

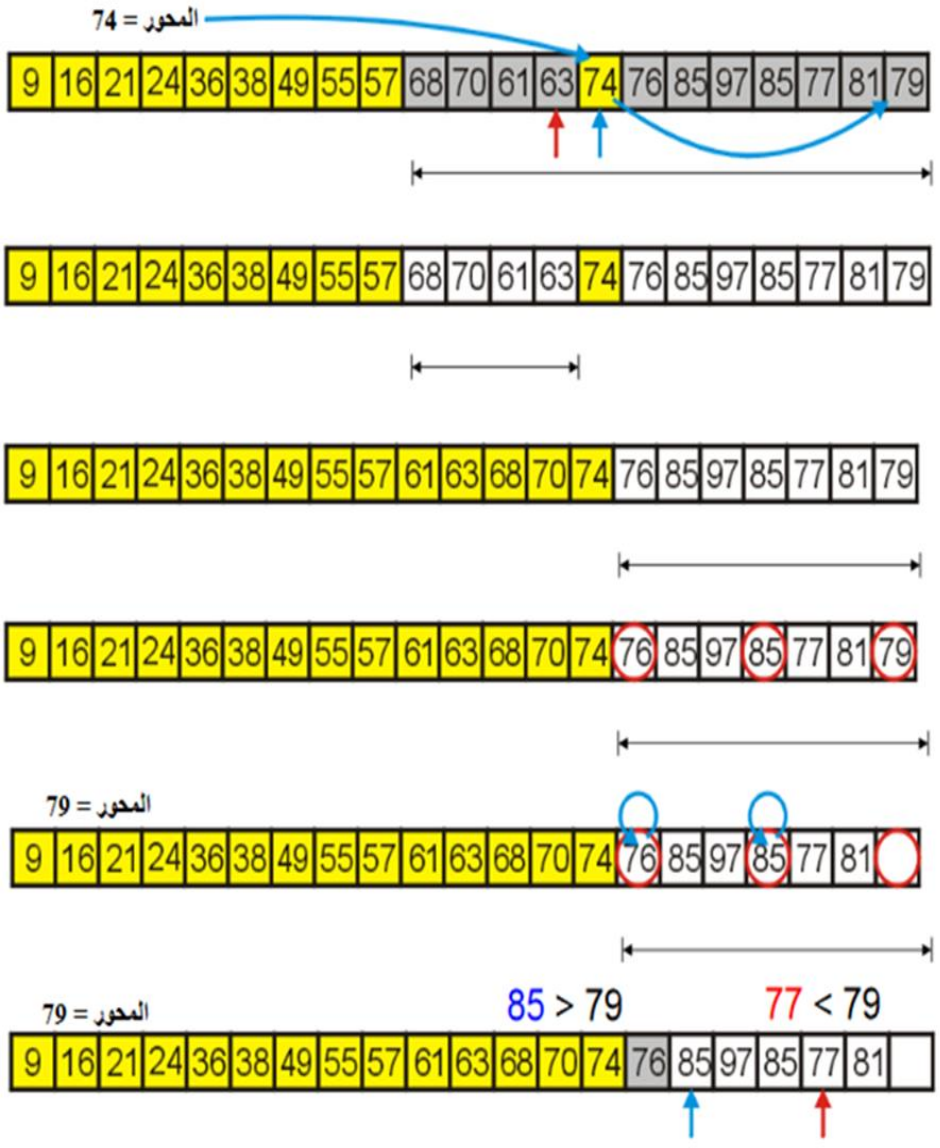


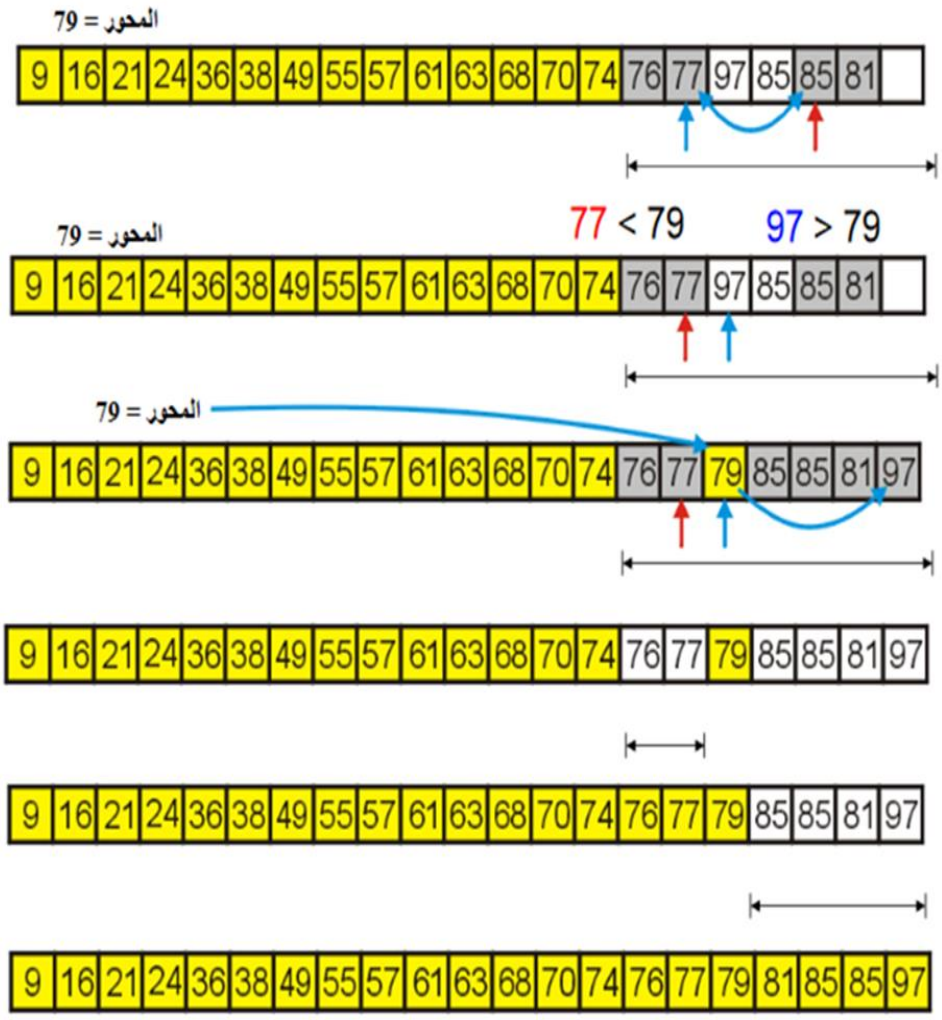

المحور = 57
24 49 16 38 63 21 85 68 76 9 81 36 55 79 74 85 97 61 77 70
63 > 57
55 < 57
المحور = 57
24 49 16 38 63 21 85 68 76 9 81 36 55 79 74 85 97 61 77 70
المحور = 57
24 49 16 38 55 21 85 68 76 9 81 36 63 79 74 85 97 61 77 70
85 > 57
36 < 57
المحور = 57
24 49 16 38 55 21 85 68 76 9 81 36 63 79 74 85 97 61 77 70
المحور = 57
24 49 16 38 55 21 36 68 76 9 81 85 63 79 74 85 97 61 77 70
68 > 57
9 < 57
المحور = 57
24 49 16 38 55 21 36 68 76 9 81 85 63 79 74 85 97 61 77 70
المحور = 57
24 49 16 38 55 21 36 9 76 68 81 85 63 79 74 85 97 61 77 70


9 < 57
76 > 57
المحور = 57
24 49 16 38 55 21 36 9 76 68 81 85 63 79 74 85 97 61 77 70
المحور = 57
24 49 16 38 55 21 36 9 57 68 81 85 63 79 74 85 97 61 77 70 76
24 49 16 38 55 21 36 9 57 68 81 85 63 79 74 85 97 61 77 70 76
24 49 16 38 55 21 36 9 57 68 81 85 63 79 74 85 97 61 77 70 76
المحور = 24
9 49 16 38 55 21 36 57 68 81 85 63 79 74 85 97 61 77 70 76
49 > 24
21 < 24
المحور = 24
9 49 16 38 55 21 36 57 68 81 85 63 79 74 85 97 61 77 70 76
المحور = 24
9 21 16 38 55 49 36 57 68 81 85 63 79 74 85 97 61 77 70 76
16 < 24
38 > 24
المحور = 24
9 21 16 38 55 49 36 57 68 81 85 63 79 74 85 97 61 77 70 76


المحور = 24
9 21 16 24 55 49 36 38 57 68 81 85 63 79 74 85 97 61 77 70 76
9 21 16 24 55 49 36 38 57 68 81 85 63 79 74 85 97 61 77 70 76
9 16 21 24 55 49 36 38 57 68 81 85 63 79 74 85 97 61 77 70 76
9 16 21 24 36 38 49 55 57 68 81 85 63 79 74 85 97 61 77 70 76
9 16 21 24 36 38 49 55 57 68 81 85 63 79 74 85 97 61 77 70 76
المحور = 74
9 16 21 24 36 38 49 55 57 68 81 85 63 79 76 85 97 61 77 70


المحور = 74
81 > 74
70 < 74
9 16 21 24 36 38 49 55 57 68 81 85 63 79 76 85 97 61 77 70
المحور = 74
9 16 21 24 36 38 49 55 57 68 70 85 63 79 76 85 97 61 77 81
المحور = 74
85 > 74
61 < 74
9 16 21 24 36 38 49 55 57 68 70 85 63 79 76 85 97 61 77 81
المحور = 74
9 16 21 24 36 38 49 55 57 68 70 61 63 79 76 85 97 85 77 81
المحور = 74
79 > 74
63 < 74
9 16 21 24 36 38 49 55 57 68 70 61 63 79 76 85 97 85 77 81


المحور = 74
9 16 21 24 36 38 49 55 57 68 70 61 63 74 76 85 97 85 77 81 79
9 16 21 24 36 38 49 55 57 68 70 61 63 74 76 85 97 85 77 81 79
9 16 21 24 36 38 49 55 57 61 63 68 70 74 76 85 97 85 77 81 79
9 16 21 24 36 38 49 55 57 61 63 68 70 74 76 85 97 85 77 81 79
المحور = 79
9 16 21 24 36 38 49 55 57 61 63 68 70 74 76 85 97 85 77 81
المحور = 79
85 > 79
77 < 79
9 16 21 24 36 38 49 55 57 61 63 68 70 74 76 85 97 85 77 81



شكل 3.9: رسم توضيحي لخطوات ترتيب قيم بطريقة الترتيب السريع.

❖ الدالة البرمجية لخوارزمية الترتيب السريع

هذه الدالة تستخدم معدل القيمة العليا والسفلى ليكون المحور

```
void quickSort(int arr[ ], int left, int right)
 {
  int i = left,  j = right ;
  int tmp;
  int pivot = arr[(left + right) / 2];

  /* التقسيم */
  while (i <= j) {
       while (arr[i] < pivot)
            i++;
       while (arr[j] > pivot)
            j--;
       if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
    }
}
/* الاستدعاء الذاتي */
if (left < j)
    quickSort(arr, left, j);
if (i < right)
    quickSort(arr, i, right);
}
```

3.11 البحث Searching

استعادة المعلومات هي واحدة من اهم تطبيقات الحاسبات. عادة تتضمن اعطاء جزء من المعلومات تسمى المفتاح, ويتم السؤال عن ايجاد القيد الذي يحتوي معلومات مشتركة اخرى. هذه تنجز بالذهاب اولا الى القائمة لمعرفة إذا كان المفتاح موجود او لا، هذه العملية تسمى بحث searching. انظمة الحاسوب تستخدم لخزن كميات كبيرة من البيانات، وتبرز دائما الحاجة لاستعادة قيود مفردة وفقا لبعض صفات البحث. ان عملية البحث عن عنصر معين في هياكل البيانات من الممكن ان يكون بسيط او معقد جدا.

البحث من الممكن ان يكون على شكل هياكل بيانات داخلية او هياكل بيانات خارجية. استعادة المعلومات حسب الصيغة المطلوبة هي الفعالية المركزية لكل تطبيقات الحاسبات.

هناك طرق بحث مختلفة وهي بشكل عام من الممكن ان نجملها بنوعين، وهما البحث الخطي والبحث غير الخطي.

3.11.1 البحث الخطي Linear Search

البحث الخطي هو خوارزمية بحث اساسية وبسيطة. البحث الخطي يبحث عن عنصر او قيمة في مصفوفة لحين ان يتم ايجاد العنصر المطلوب او لم يتم ايجاده، وطريقة البحث تكون ببحث تعاقبي او متتالي. هي تقارن المفتاح مع جميع العناصر الاخرى التي في القائمة، فاذا تم مطابقة المفتاح الذي نبحث عنه مع أحد عناصر القائمة فيتم اعادة موقع هذا المفتاح او العنصر وبخلاف ذلك يتم اعادة (كلمة خطا او من الممكن -1 او أي ترتيب اخر يشير الى عدم وجود العنصر في القائمة). البحث الخطي يطبق على القوائم المرتبة والقوائم غير المرتبة عندما يكون هناك عدد قليل من العناصر في القائمة.

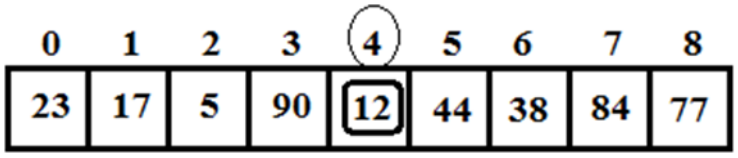
تحدد كفاءة الخوارزمية عن طريق حساب عدد المقارنات التي تقوم بها الخوارزمية.

ملاحظة: خوارزمية البحث المثالية هي التي تصل الى البيانات المرغوبة بأقل عدد من المقارنات الممكنة.

ملاحظة: عملية البحث تنتهي بواحدة من نتيجتين، واحدة للبحث الناجح والاخرى للبحث غير الناجح.

مثال : في الشكل 3.10 فان البحث عن القيمة (45) تعيد (غير موجود).. وهذه عملية بحث عير ناجح.

اما البحث عن القيمة (12) ستعيد الرقم (4) وهو يمثل موقع القيمة التي تم ايجادها.. وهذه عملية بحث ناجحة.



شكل 3.10: مصفوفة قيم يتم البحث فيها عن قيمة مفتاح.

تحليل كفاءة خوارزمية البحث تحدد بعدد مرات المقارنة كما سبق وأسلفنا.

ففي حالة البحث الناجح، فان أفضل حالة هي عندما نجد العنصر المطابق للمفتاح في الموقع الاول وبالتالي فان عدد مرات المقارنة لأفضل حالة في البحث الناجح هي مرة واحدة فقط ((O (1)).

اما اسوأ حالة في البحث الناجح عندما يكون العنصر الذي نبحث عنه اخر عنصر في القائمة وبالتالي فان عدد المقارنات ستكون ((O(N)).

اما في حالة البحث غير الناجح فان أفضل حالة واسوأ حالة متساوية وهي ((O(N)).

❖ دالة برمجية لتنفيذ عملية البحث الخطي:

```
function findIndex(values, target)
 {
  for(int i = 0; i < values.length; ++i)
    {
     if (values[i] == target)
        return i;
    }
  return -1;
 }
```

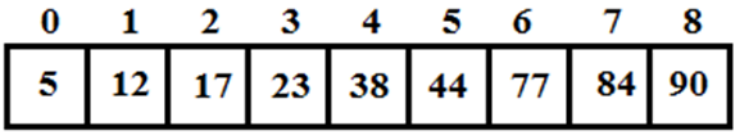
من الممكن استدعاء الدالة اعلاه مع تمرير المصفوفة والعنصر المطلوب البحث عنه كما يلي:

```
findIndex ([2, 5, 0, 9, 4, 3, 10, 7, 22, 12]);
```

3.11.2 البحث الثنائي Binary Search

هي طريقة اخرى من طرق البحث والتي تعمل على القوائم المرتبة (قيمها مرتبة ترتيب تصاعدي او تنازلي). البحث الثنائي ينفذ على المصفوفات او القوائم المرتبة, في البداية يتم مقارنة المفتاح الذي نبحث عنه مع القيمة التي في وسط المصفوفة. فاذا كانت متطابقة (متساوية) فان هذه القيمة يتم اعادتها. إذا القيمة اقل من القيمة التي في وسط المصفوفة، فان القيمة التي نبحث عنها ستكون في النصف الاسفل او النصف الاول من المصفوفة، اما إذا كانت أكبر من القيمة التي في الوسط فهذا يعني انها تقع في النصف العلوي او النصف الثاني من المصفوفة (في حال ان المصفوفة مرتبة تصاعديا). هذه العملية تكرر وفي كل مرة ننصف جزء المصفوفة الذي حدد في المرة السابقة، ونحدد في اي جزء يقع المفتاح لنعمل على ذلك الجزء مع اهمال باقي الاجزاء.

مثال: في الشكل 3.11 مصفوفة مرتبة مطلوب البحث عن القيمة (44).



شكل 3.11: مصفوفة مرتبة.

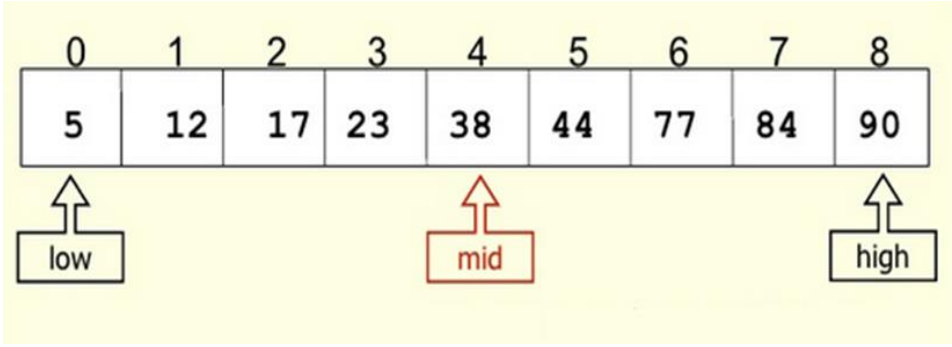
❖ خطوات البحث الناجح..

الخطوة الاولى: في هذه المرحلة من المفروض ان نجد الموقع الوسط وبعدها تتم عملية مقارنة القيمة التي نبحث عنها مع هذه القيمة في المنتصف.

موقع الوسط = (اعلى موقع + أدني موقع) / 2

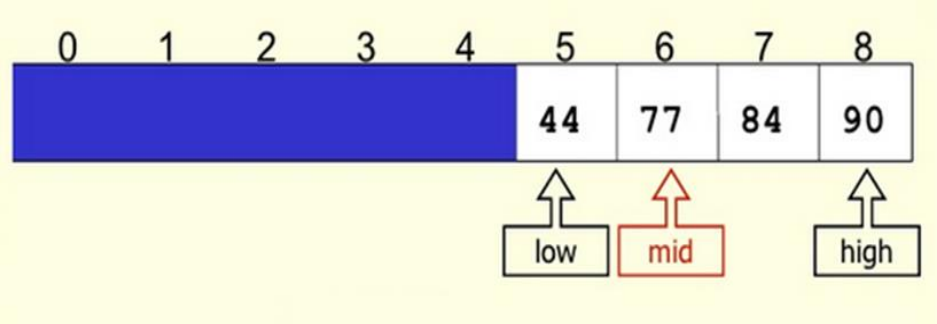
موقع الوسط = (8 + 0) / 2 = 4

لاحظ هنا ان القيمة في الموقع الوسط كما في الشكل 3.12 هي (38), ونظرا لان القيمة التي نبحث عنها (44) أكبر من قيمة الوسط (38), لذلك سنتوجه للبحث في النصف الاعلى ونهمل النصف الادنى. ان الموقع الاسفل في نصف المصفوفة الذي تم اختباره سيكون (5) (لان المنتصف هو 4 لذلك سيكون الموقع الاسفل الى يمين المنتصف اي المنتصف + 1)، والموقع الاعلى سيكون (8).



شكل 3.12: تحديد المنتصف فضلا عن القيمة السفلى والعليا للمصفوفة.

الخطوة الثانية: ايضا نحاول ايجاد المنتصف للنصف الاعلى من المصفوفة، ونقارن.
موقع الوسط = 2 / (5 + 8) = 6 (دائما الكسور تقرب الى اقرب عدد صحيح باستخدام عملية تدوير فمثلا 6.5 تقرب الى 6 وهكذا).
الان، القيمة في موقع الوسط هي 77 كما في الشكل 3.13, ونظرا لان 44 اصغر من 77 لذلك فانا الخطوة اللاحقة ستكون البحث في النصف الادنى من المصفوفة الجديدة، وسيكون الموقع الاعلى هو (5) (لاحظ هنا ان الموقع الاعلى سيكون الوسط -1)، بينما الموقع الادنى سيكون (5).

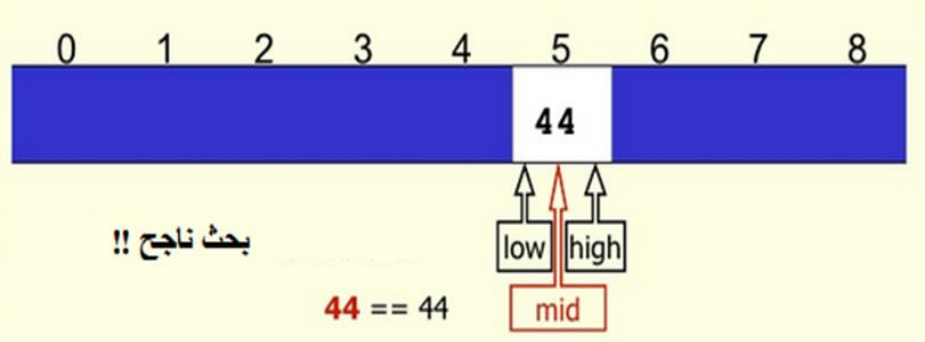


شكل 3.13: تحديد المنتصف للنصف العلوي من المصفوفة.

ملاحظة: عندما نختار النصف الادنى من المصفوفة فان الموقع الادنى للمصفوفة هو ذات الموقع الادنى لهذا النصف من المصفوفة، بينما الموقع الاعلى لها يساوي (موقع الوسط - 1).
اما إذا اخترنا النصف العلوي من المصفوفة فان الموقع الاعلى للمصفوفة هو ذات الموقع الاعلى لهذا النصف من المصفوفة، بينما الموقع الادنى لهذا النصف يساوي (موقع الوسط + 1).

الخطوة الثالثة: في هذه الخطة فان موقع الوسط هو 5 وهو الموقع الوحيد المتبقي

في المصفوفة والقيمة التي فيه هي (44), وهي ذات القيمة التي نبحث عنها لذلك سيتم اعادة موقع القيمة وبيان ان البحث تم بنجاح. لاحظ الشكل 3.14.



شكل 3.14: تنصيف الجزء الاخير من المصفوفة.

لنعيد العمل بالمثال السابق ولكن لنبحث (بحث غير ناجح) عن القيمة (45).

الخطوة الاولى: موقع الوسط يحتوي على القيمة 38 كما في الشكل 3.12, ونظرا لان (38 < 45) فسيتم البحث في نصف المصفوفة الاعلى.

الخطوة الثانية: نستعين بالشكل 3.13, ونلاحظ ان قيمة الوسط (45<77) لذلك سيتم البحث في الخطوة اللاحقة في النصف الاسفل من المصفوفة.

الخطوة الثالثة: يلاحظ هنا وحسب الشكل 3,14 ان القيمة في الوسط هي 44, ونظرا الى ان (45>44) لذلك فان البحث في المرحلة اللاحقة سيكون في النصف الاعلى من المصفوفة.

هنا الموقع الاعلى للمصفوفة الجديدة سيكون ذات الموقع الاعلى لها وهو (5)، بينما الموقع الادنى سيكون (5+1=6) .. من ذلك نلاحظ ان الموقع الادنى أكبر من الموقع الاعلى وهذا دلالة على انتهاء البحث دون ان نجد القيمة التي نبحث عنها لذلك تتوقف عملية البحث.

- ✓ كفاءة البحث الثنائي:

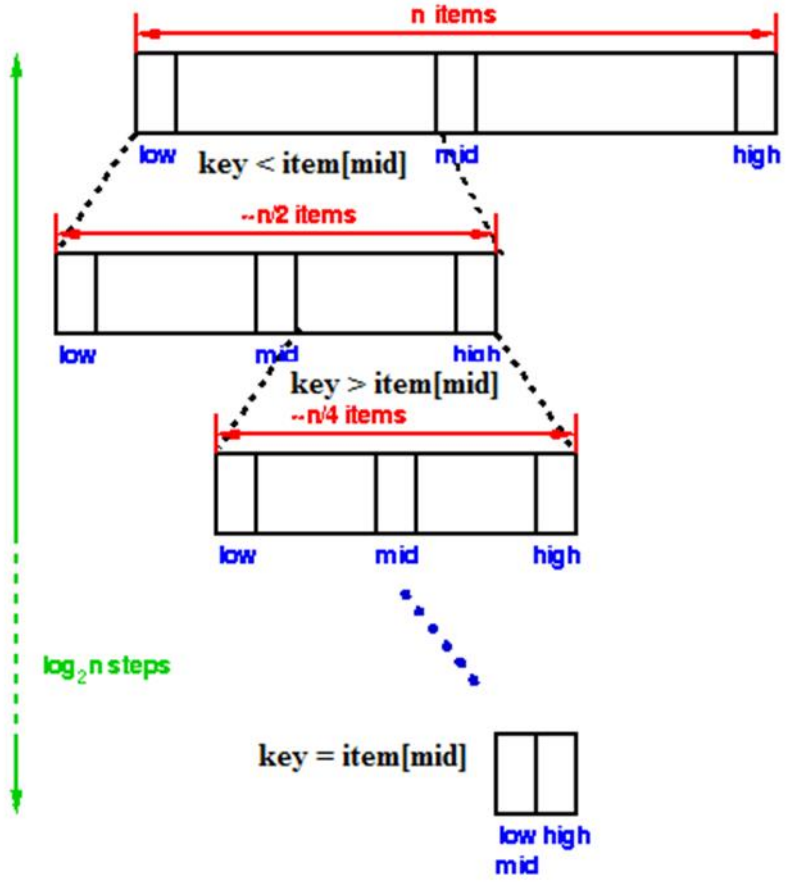
البحث الناجح: أفضل بحث يكون فيه عدد المقارنات = 1

اسوأ بحث يكون عدد المقارنات يساوي $\log_2 N$, حيث ان N يمثل حجم المصفوفة.

البحث غير الناجح: أفضل بحث لا يوجد

اسوا بحث يساوي $\log_2 N$

فقط للمقارنة إذا كان لدينا مصفوفة بحجم 1000 عنصر فان اسوا بحث بالبحث الخطي يساوي 1000 مقارنة، بينما عند البحث الثنائي فان عدد المقارنات تساوي 14.



شكل 3.15: شكل توضيحي لطريقة البحث الثنائي.

❖ دالة برمجية لطريقة البحث الثنائي هو:

```
function findIndex(values, target)
{
  return binarySearch(values, target, 0, values.length - 1);
};

function binarySearch(values, target, start, end) {
  if (start > end) { return  -1; }     // غير موجود

  int middle =  (start + end) / 2 ;
  int value = values[middle];

  if (value > target)
{ return        binarySearch (values, target, start, middle-1); }
  if (value < target)
 { return      binarySearch(values, target, middle+1, end); }
  return middle; // موجود
}
```

✓ من الممكن استدعاء الدالة مع تمرير قائمة القيم والمفتاح كما يلي:

```
 findIndex ([2, 4, 7, 9, 13, 15], 13)
```

3.11.3 البحث بالاستيفاء Interpolation Search

البحث بالاستيفاء هي طريقة بديلة لطريقة البحث الثنائي والتي تستخدم معلومات حول التوزيع الضمني للبيانات التي يتم البحث عنها. باستخدام هذه المعلومات الاضافية، فان خوارزمية البحث الاستيفائي من الممكن ان تكون أسرع بدرجة $O(log(log(n)))$, حيث ان n هو حجم المصفوفة.

في البحث الثنائي فأنها دائما تختار العنصر الذي في وسط المصفوفة لأجل المقارنة، مع اهمال نصف فضاء البحث. البحث الاستيفائي يحاول ان يتنبأ اين يوجد المفتاح الذي نبحث عنه في فضاء البحث بواسطة الاستيفاء الخطي، فهي تحسب الموقع الافتراضي للمفتاح وإذا لم يتم ايجاد المفتاح في ذلك الموقع فأنها تقلص فضاء البحث لجزء قبل او بعد الموقع الافتراضي للمفتاح. هذه الطريقة تستخدم فقط إذا كانت حسابات الفرق بين عناصر المصفوفة المتعاقبة معقولة.

✓ الموقع التقديري للمفتاح يحسب كما يلي:

نفرض ان فضاء البحث الحالي هو A[i…j] وكان البحث عن القيمة x, الان باستخدام الاستيفاء الخطي للقيم، فان x يفترض ان تكون في الموقع:

```
middle = i + ((j - i) * (x - A[i])) / (A[j] - A[i])
```

يمكن اعادة كتابة العلاقة اعلاه لغرض تفسير الرموز الواردة فيها كما يلي:

الموقع الوسط = الموقع الادنى+ ((الموقع الاعلى – الموقع الادنى) × (قيمة المفتاح – القيمة في الموقع الادنى)) ÷ (القيمة في الموقع الاعلى – القيمة في الموقع الادنى)

❖ برنامج تنفيذ طريقة البحث بالاستيفاء.

```
#include "stdio"
#include "stdlib"

#define MAX 200
using namespace std ;
int interpolation_search (int a[], int bottom, int top, int item) {

int mid;
while (bottom <= top) {
mid = bottom + (top - bottom) * ((item - a[bottom]) / (a[top] - a[bottom])) ;
  if (item == a[mid])
   return mid + 1;
  if (item < a[mid])
   top = mid - 1;
  else
   bottom = mid + 1;
 }
 return -1;
}

int main() {
 int arr[MAX];
```

```
 int i, num;
 int item, pos;

 cout<< "\n Enter total elements (num< MAX) " ;
 cin >> num ;

cout<< "Enter Values to Elements : " ;
 for (i = 0; i < num; i++)
  cin>> arr[i] ;

cout << "\n Display Elements " ;
 for (i = 0; i < num; i++)
  cout<< arr[i]<< “\t” ;

 cout<< "\n Enter value you Search For : " ;
 cin>>item ;
 pos = interpolation_search (&arr[0], 0, num, item);
 if (pos == -1)
  cout<< "\n Element” << item<< “not found \n”;
 else
  cout<< "\n Element” <<item<<” found at position \n” ;

 return 0;
}
```

3.11.4 طريقة البحث بالقفز Jump Search

هذه الطريقة تنفذ على المصفوفات المرتبة. في هذه الطريقة سنستخدم قفزات يحدد طولها بطريقة تساعد على تحقيق الوصول الى النتيجة المطلوبة (ايجاد العنصر الذي نبحث عنه) بأقل وقت ممكن. وعندما نقول قفزات هذا يعني ان المقارنة ستكون مع عناصر تبعد عن بعضها مسافة (غالبا تعتمد على عدد العناصر) تحسب مسبقا تسمى قفزة. غالبا يحدد طول القفزات بمسافة ($\sqrt{n}$) حيث ان n هي حجم المصفوفة.

عند تحديد طول القفزة فهذا يعني امكانية تحديد المواقع التي سيتم المقارنة مع قيمها. فمثلا إذا كان طول القفزة يساوي 3 فهذا يعني ان المواقع التي سيتم مقارنة عناصرها هي (3, 6, 9, 12, 15, 18 …).

يقارن المفتاح مع قيمة اول موقع فاذا كان المفتاح أكبر (في حالة كون المصفوفة مرتبة تصاعديا، اما إذا تنازلي فيكون بالعكس) ينتقل الى المقارنة مع الموقع التالي وتستمر المقارنة لحين ان يجد قيمة موقع أصغر من قيمة المفتاح. في هذه الحالة، فان قيمة المفتاح ستكون بين الموقع الذي كان اخر مقارنة معه والموقع السابق له، هنا ستكون مقارنة خطية مع القيم بين هذين الموقعين. فأما ان يتم ايجاد تطابق او لا يوجد.

مثال: في المصفوفة التالية مطلوب البحث عن القيمة (20) باستخدام البحث الثنائي، البحث الاستيفائي, والبحث بالقفز.

```
int[] a = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30}
```

- ✓ **الحل بطريقة البحث الثنائي**

المصفوفة هي بين الموقع الادنى صفر، والموقع الاعلى 15.

اذن الوسط = 2 / (15 + 0) = 7
القيمة في الموقع (7) هي (14)، والتي هي أصغر من قيمة المفتاح (20). لذلك، فان الموقع الادنى سيكون (8 = 1 + 7) والموقع الاعلى يبقى (15).
الان الموقع الوسط = 2 / (15 + 8) = 11
القيمة في الموقع (11) هي (22)، والتي هي أكبر من قيمة المفتاح (20). لذلك، فان الموقع الادنى سيكون (8) والموقع الاعلى يكون (10=1-11).
الموقع الوسط = 2 / (10 + 8) = 9
القيمة في الموقع (9) هي (18)، والتي هي أصغر من قيمة المفتاح (20). لذلك، فان الموقع الادنى سيكون (10 = 1 + 9) والموقع الاعلى يبقى (10).
الموقع الوسط = 2 / (10 + 10) = 10
القيمة في الموقع (10) تساوي (20) والذي هو يساوي قيمة المفتاح (20) لذلك فان المفتاح تم ايجاده في الموقع (10).

- ✓ **البحث الاستيفائي**

الموقع الادنى =0, الموقع الاعلى = 15, القيمة التي في الموقع الادنى = 0, القيمة التي في الموقع الاعلى = 30, المفتاح = 20. اذن الموقع الوسط يساوي

Middle =low +(high-low)*((key –a[low]) /(a[high] –a[low]))
middle = 0 +ceil(15 –0)* ((20 –0) / (30–0))= 0 +10 =10

القيمة التي في الموقع (10) تساوي (20) وهي تساوي قيمة المفتاح. بهذا يكون قد تم ايجاد قيمة المفتاح.

✓ **البحث بالقفز**

هنا سيكون طول القفزة الواحدة يساوي $(\sqrt{16} = 4)$.
بموجب هذا الطول، فان الخوارزمية ستنظر الى قيم العناصر في المواقع (4, 8, 12,…).
الان القيمة في الموقع (4) هي (6) وهذه القيمة أصغر من المفتاح (20)، لذلك سيتم النظر الى موقع القفزة التالية وهي الموقع (8) ونلاحظ ان القيمة في هذا الموقع تساوي (14) وهي ايضا اقل من قيمة المفتاح، عليه يتم الانتقال الى القفزة الاخرى ويتم المقارنة مع القيمة التي في الموقع (12)، القيمة في الموقع (12) تساوي 22 وهي أكبر من قيمة المفتاح (20)، من المقارنة الاخيرة فان قيمة المفتاح ستكون بين الموقع (8) والموقع (11) ولذلك سيتم البحث عنها هنا بحث تسلسلي ابتداءا من الموقع (8). سيتم ايجاد قيمة المفتاح في الموقع 10.

3.12 جدول التجزئة Hashing Table

هو هيكل بيانات يقوم بخزن البيانات بطريقة الترابط. في جدول التجزئة، يتم خزن البيانات في مصفوفة بحيث ان كل قيمة بيانات لها رقم فهرس (موقع في المصفوفة) وحيد. الوصول الى البيانات يصبح سريع إذا عرفنا رقم الفهرس للبيانات المطلوبة.
عليه، سيتولد هيكل بيانات بحيث ان عمليات الحشر والبحث تكون سريعة جدا نسبة لحجم البيانات. جدول التجزئة يستخدم المصفوفة كوسط خزن ويستخدم تقنية التجزئة لتوليد الفهرس الذي سيتم به حشر العنصر.

3.12.1 التجزئة Hashing

جدول التجزئة هو هيكل بيانات لخزن قيم ثنائية تمثل (المفتاح، القيم). وهي لا تشبه المصفوفات التي تستخدم ارقام الفهرسة للوصول الى العناصر، فجدول التجزئة يستخدم المفاتيح للبحث في مدخلات الجدول.

في C++ يتم تنفيذ جدول التجزئة عادة على شكل مصفوفة قوائم موصولة. فهي تخزن على شكل مصفوفة ثنائية الابعاد. بالنسبة لمصفوفة ثنائية الابعاد، فان العناصر تتكون من صفوف ثابتة الابعاد. لكن، في جدول التجزئة فان العناصر من الممكن ان تتمدد وتتقلص لتستوعب (افتراضيا) ما لانهاية من مدخلات الجدول.

من حيث الكفاءة فان جدول التجزئة يعتبر حل وسط بين المصفوفات والقوائم الموصولة. فهو يستخدم كل من الفهرسة والمرور على القائمة لخزن واسترجاع البيانات.

البحث عن العناصر باستخدام الفهرسة يجعل المصفوفة كفوءة جدا. بغض النظر عن موقع خزن العنصر في المصفوفة، هي دائما تأخذ نفس الوقت لاستعادة البيانات. حسب المصطلحات الفنية، الحصول على عنصر من مصفوفة هو ((O(1) من العمليات او (وقت ثابت).

البحث عن عناصر في القوائم الموصولة هي اقل كفاءة بكثير. لأنك لا يمكنك ان تصل بشكل مباشر الى اي عقدة في القائمة. فبدل الوصول المباشر فأنك من المفروض ان تمر على القائمة من البداية لحين الوصول الى العنصر المطلوب بطريقة البحث التسلسلي. إذا وجدت العنصر المطلوب في بداية القائمة فان استعادته تتطلب ((O (1) من العمليات حيث أنك مررت على عنصر واحد فقط في القائمة. اما إذا كان العنصر الذي تبحث عنه في ذيل القائمة فأنك تحتاج الى ((O(n) من العمليات، حيث ان n تمثل العدد الكلي للعقد في القائمة.

بشكل عام فان جدول التجزئة مشابه للمصفوفات فهو يبحث بوقت ثابت كمعدل وهو ((O(1), بغض النظر عن عدد العناصر في الجدول.

3.12.2 كيف تعمل جداول التجزئة

عندما نتحدث عن عمل جداول التجزئة، فيجب ان نتحدث عن ثلاث امور رئيسية وهي:

1. ثنائي القيمة والمفتاح: ان الثنائي (القيمة، المفتاح) هي جوهر هيكل بيانات جدول التجزئة. للحالات المناسبة، فأنها تجعل البحث حقيقة كفوء ومثالي.
في بعض الحالات ليس بالضرورة استخدام مفتاح مع القيمة وانما تستخدم ذات القيمة على انها قيمة ومفتاح بذات الوقت.
2. دالة التجزئة: دالة التجزئة تقرر اين يتم خزن او من اين يتم استعادة العناصر في جدول التجزئة. هي تأخذ مفتاح العنصر على انه معامل ادخال وتعيد رقم الفهرس او الموقع لهذا العنصر. بشكل عام، دالة التجزئة تستخدم رياضيات باقي القسمة. عمليا، قيمة المفتاح تقسم على طول الجدول لتوليد رقم فهرسة في الجدول. رقم الفهرسة هذا يشير الى الموقع في جدول التجزئة.

```
int hash( string key)
{
   int value = 0;
   for ( int i = 0; i < key.length(); i ++)
      value += key[i];
   return value % tableLength;
}
```

3. **تصادم عناصر جدول التجزئة**: التصادم يعني ان اثنان او ربما اكثر من المفاتيح لها نفس رقم الفهرس. لغرض تعظيم كفاءة خزن واسترجاع العناصر في جدول التجزئة، فاننا نحتاج الى تقليل عدد التصادم الى الحد الادنى. كل تصادم يعني عنصر اخر في القائمة، او عقدة في القائمة الموصولة. نحن نعلم بان خزن واستعادة العناصر في القوائم الموصولة تنمو أكثر في مجال عدم الكفاءة عندما تكبر القائمة. لذلك من المحبذ ان نحافظ على هذه القوائم الموصولة لتكون أقصر ما يمكن.

هناك عدة طرق لكتابة دالة تجزئة كفوءة بحيث تعالج حالات التصادم بكفاءة، منها:

A. جعل حجم الجدول رقم اولي (اي رقم لا يقبل القسمة الا على واحد او نفسه) هي طريقة جيدة لتقليل التصادم الى الحد الادنى. بشكل عام، ان دالة التجزئة الجيدة هي التي تملأ جدول التجزئة بشكل متجانس وعادل.
ان السبب باستخدام العدد الاولي، هو ان القيمة الناتجة من دالة التجزئة في حالة عدم استخدام عدد اولي لحجم المصفوفة ستكون صفر (باقي القسمة) لكل مفتاح يقبل القسمة على الحجم. لكن، إذا استخدمنا عدد اولي للحجم فأننا سنقلص فرص ان يكون المفتاح يقبل القسمة على الحجم.

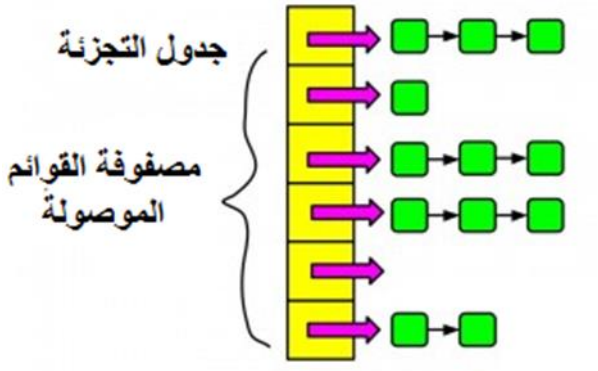
B. واحدة من الحلول المستخدمة هي عندما نستخدم دالة التجزئة ويكون الموقع الناتج محتل من قبل عنصر اخر فيتم البحث عن أقرب موقع فارغ لهذا الموقع وتحشر القيمة الجديدة فيه. هذه الطريقة لا تعتبر طريقة جيدة، من الممكن استخدامها عندما يكون حجم الجدول ثابت.
في الشكل 3.16 توضيح لكيفية تنفيذ هذه الطريقة على القيم التالية عندما يكون حجم الجدول مساوي الى 20:

(1, 2, 42, 4, 12, 14, 17, 13, 37)

C. حل اخر هو تقنية سلسلة جدول التجزئة البسيط، فكل موقع في المصفوفة يشير الى قائمة موصولة بحيث تتكون عناصر هذه القائمة من العناصر التي تتصادم في ذلك الموقع. ان عملية الحشر تتطلب ايجاد الموقع الصحيح وربط العنصر بنهاية القائمة في ذلك الموقع. اما عملية الحذف فهي تتطلب البحث في القائمة ضمن الموقع المحدد لإزالة العنصر، لاحظ الشكل 3.17.

التسلسل	المفتاح	دالة التجزئة	الفهرسة في المصفوفة	بعـــد تلافـــي التصادم
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

شكل 3.16: توضيح كيفية حساب مواقع المصفوفة ومعالجة التصادم باختيار أقرب موقع.



شكل 3.17: توضيح لمعالجة التصادم بطريقة مصفوفة القوائم.

مثال: لو فرضنا ان القيم التالية مطلوب خزنها في جدول تجزئة حجمه 20.

(1, 2, 42, 4, 12, 14, 17, 13, 37)

النتيجة ستكون كما في الشكل 3.18.

التسلسل	المفتاح	دالة التجزئة	الفهرسة في المصفوفة
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

شكل 3.18: توضيح لحساب مواقع القيم في جدول التجزئة مع معالجة التصادم بطريقة مصفوفة القوائم.

❖ برنامج لتنفيذ جداول التجزئة مع العمليات الأساسية على القيم.

```
#include <string>
using namespace std;
struct datar {

public:
int id; // لحمل تعريف وحيد لكل عنصر
```

```
int data; //بيانات كل عنصر تم هنا استخدام اعداد صحيحة
};
class hasher {
datar dt[51]; // الجدول الذي سيحمل عناصر جدول التجزئة
 int numel; //عدد العناصر في المصفوفة, لفحص انه مملوء
public:
        hasher();
        int hash(int &id);
int rehash(int &id);
 int add(datar &d);
        int remove(datar &d);
        void output();
};

/* دالة اعطاء id لجدول التجزئة
من المفضل استخدام رقم اولي لذا استخدم الرقم 11 فهو افضل لتقليل التصادم
*/
int hasher::hash(int &id) {
return (id%51);
}
/* rehash دالة في حالة حدوث تصادم فسيتم استخدام*/
int hasher::rehash(int &id) {
```

```
return ((id+1)%51);
}
hasher::hasher() {
// خلق مصفوفة
int i;
for (i=0;i<=50;i++) {
dt[i].id = -1; //ضبط قيمة ids الى -1 لبيان انها فارغة
dt[i].data = 0; //0 تكون افتراضا كل القيم تساوي
}
numel = 0;
}
int hasher::add (datar &d) {
if (numel < 51) {
// الجول له مواقع فارغة
int hashed = hash(d.id);
if (hashed >= 0 && hashed <= 50 && dt[hashed].id == -1) {
//الموقع فارغ اسناد قيمة جديدة
dt[hashed].id = d.id;
dt[hashed].data = d.data;
return 0;
} else {
// rehash يحتاج الى id
int i=0;
```

```
// حاول مع كل موقع في الجدول لإيجاد الموقع الفارغ
while(i<=50) {
hashed = rehash(hashed);
if(dt[hashed].id == -1) {
dt[hashed].id = d.id;
dt[hashed].data = d.data;
return 0;
} else if (i==50) {
//عدم امكانية ايجاد موقع فارغ
return -1; // انهاء الدالة بوجود خطأ
}
i++; // i تحديث قيمة
}
}
} else {
//الجدول مملوء
return (-1);
}
int hasher::remove (datar &d) {
int hashed = hash(d.id);
if(dt[hashed].id == d.id) {
// هذا افضل واحد للحذف
```

```
dt[hashed].id = -1;
numel -= 1;
return 0;
} else {
//نحن نحتاج الى a rehash لايجاد واحد
int i=0;
while (i<=50) {
hashed = rehash (hashed);
if (dt[hashed].id == d.id) {
dt[hashed].id = -1; //ضبط id الى -1 لانها ستصبح فارغة empty
numel -= 1; //تقليل واحد من عدد العناصر
return 0;
//success
} else if (i==50) {
return -1;
// انهاء الدالة
}
i++; // i تحديث قيمة
}
}
}
void hasher::output() {
int i;
```

```
for (i=0;i<51;i++) {
cout<<i<<" ->  "<<dt[i].id<<"    "<<dt[i].data<<endl;
}        }
int main() {
int id=45; // id اول
int ret;
ret = 0;
datar d;
d.data = 52005; //قيمة لكل القيود, ببساطة
hasher h1;
while (ret != -1) {
d.id=id;
ret = h1.add(d); // اضافة هذا القيد للجدول
//id += (id/5); //  id, تحديث قيمة id لاحظ كيف تختلف قيمة
}
d.id = 271861; //حدث قيمة واحد من id الذي تم اضافته في التكرار اعلاه
h1.remove(d); //حذف القيد من الجدول
h1.output();  //              id  =  271861 لاحظ المخرجات سيكون قيد واحد فارغ في
الفهرس او الموقع 33 وهو القيد الذي له
return 0;
}
```

الفصل الرابع
المكدس STACK

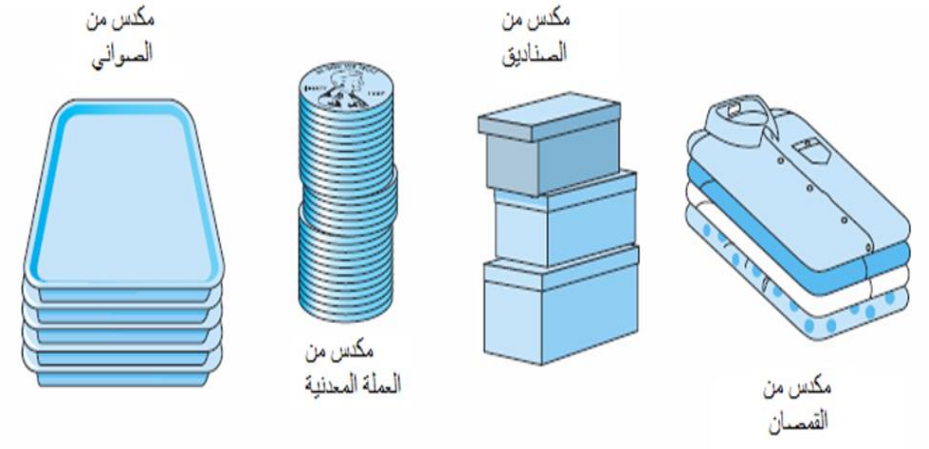
4.1 المقدمة

المكدس هو هيكل مشابه للقائمة بحيث ان عملية الاضافة والحذف تتم من نهاية واحدة. هذه القيود تجعل المكدس اقل مرونة من القوائم، لكنها ايضا تجعل المكدس اكثر كفاءة (بالنسبة للعمليات التي يقوم بها) وكذلك سهل بالتنفيذ. العديد من التطبيقات تتطلب فقط نموذج تحديدات عمليات الاضافة والحذف التي يوفرها المكدس. بالرغم من هذه التحديدات فان المكدس له الكثير من التطبيقات.

4.2 تعريف المكدس

هو طريقة خاصة لخزن البيانات والتعامل معها, وهو نسخة محددة من المصفوفات, العناصر الجديدة من الممكن ان تضاف الى المكدس او تحذف من المكدس من نهاية واحدة, فهو مستودع بيانات.

ويتبع المكدس هيكلية الداخل اولا خارج اخيرا ويمكن ان يمثل ماديا كمصفوفة ويدعى حينها هياكل البيانات المستقرة *static data structure* وعندها سيرث جميع صفات المصفوفات, او من الممكن ان يمثل كقائمة موصولة ويدعى هياكل البيانات الديناميكية *dynamic data structure* وسيمتلك جميع خواص القوائم الموصولة وبغض النظر عن طريقة تمثيل المكدس فان عمليات الاضافة والحذف تحدث في اعلى المكدس فقط.



شكل 4.1: نماذج تصف المكدس

يستخدم عادة المكدس هياكل البيانات عند كتابة الشفرة. مبدأ المكدس في الحقيقة بسيط, والذي يجعله حتى ابسط لكتابة شفرتة. افرض هذه الحالة, لديك اربعة قمصان موضوعة على طاولة فوق بعضها البعض (شكل 4.1) فاذا رغبت ان تضيف قميص خامس لها, فماذا تعمل بكل تاكيد ستضع القميص الخامس فوق القمصان الاربع, الان ماذا لو اردت ان تسحب القميص الثالث (الابيض) من القمصان الاربع التي لديك, هنا عليك ان تزيل القميص الأعلى ثم الذي بعده وهكذا لحين ان يصبح القميص الثالث في **الأعلى** ليتسنى لك الحصول عليه ومن ثم اعادة القمصان التي ازلتها الى مكانها وحسب ترتيبها اي ان تضعها قميص بعد الاخر (فوق او **أعلى** القمصان الاخرى).

لاحظ هنا, اني اشرت الى كلمة الأعلى بحروف غامقة, نعم لان أعلى المكدس هو اكثر كلمة مهمة في التعامل مع المكدس. البيانات تخزن في المكدس بحيث ان اضافة البيانات لايسمح بها الا من الأعلى فقط. ان اضافة وحذف البيانات لايسمح به الا من أعلى المكدس فقط (الاعلى كلمة افتراضية وليس حقيقية).

ملاحظة: لكي لايحدث سوء فهم عندما نقول اعلى المكدس فهو لايعني ان المكدس وعاء وله اعلى واسفل, المقصود هنا اعلى منطقيا فمثلا لو مثلنا المكدس على شكل

مصفوفة فانك لايمكنك العمل على بداية المصفوفة او الوصول الى أي عنصر في المصفوفة وانما فقط الى اخر عنصر في المصفوفة.

4.3 العمليات الأساسية التي تجرى على المكدس

هناك عدد من العمليات المحددة التي من الممكن ان نتعامل بها مع المكدس وهي:

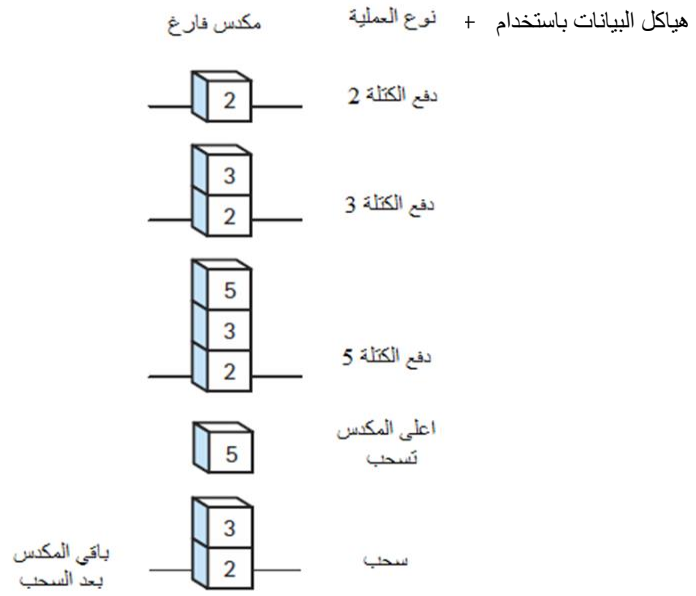
الدفع او الاضافة Push: وهي عملية اضافة عنصر جديد الى أعلى المكدس.

السحب او الحذف Pop: وهي عملية سحب عنصر من أعلى المكدس.

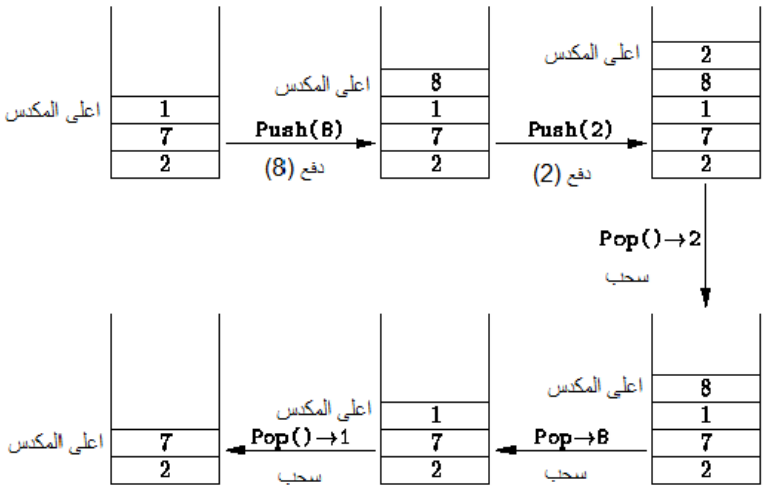
هذه العمليات (الاضافة والحذف) موضحة بالشكل 4.2, حيث نلاحظ ان الاضافة تتم من اعلى المكدس وعنصر واحد في كل مرة, وكذلك فان الحذف يتم من اعلى المكدس.. لاحظ هنا في حالة الاضافة يجب ان تحدد القيمة المضافة, مع مراعاة ان جميع عناصر المكدس هي من ذات النوع, اما في حالة الحذف فاننا نحذف العنصر الذي في اعلى المكدس بغض النظر عن قيمتة.

لاحظ وجود مؤشر يشير الى موقع اعلى عنصر بالمكدس (يمثل عنوان العنصر الذي في اعلى المكدس بالذاكرة المخصصة للمكدس) او هو يمثل موقع العنصر الاعلى في المصفوفة.

في كل عملية اضافة او حذف فان موقع المؤشر يجب ان يتغير, تغيير المؤشر يكون زيادة او نقصان بمقدار رقم واحد. فعندما تكون هناك عملية اضافة فان المؤشر يزاد بمقدار واحد, وفي عملية الحذف يتم انقاص المؤشر بمقدار واحد. في هذه الحالة فان المؤشر سيؤشر على موقع اخر في المكدس الشكل 4.3 يوضح عمليات الاضافة والحذف في المكدس.



شكل 4.2: يوضح تاثير عمليات السحب والاضافة



شكل 4.3: يوضح عمليات الاضافة والحذف من المكدس.

4.4 تطبيقات المكدس

المكدس له تطبيقات كثيرة. ادناه بعض تطبيقات المكدس:

4.4.1 عند تنفيذ اي برنامج فان المعالج سيكون له الدور الاساس بعملية التنفيذ, لذلك اذا حدث وان تم استدعاء دالة معينة في البرنامج فان المعالج سيتوقف عن اكمال تنفيذ البرنامج حسب تسلسل الايعازات او الاوامر في البرنامج ويتفرع لاكمال تنفيذ الدالة ثم العودة الى ذات الموقع الذي تفرع منه لاكمال التنفيذ حسب تسلسل الايعازات, في هذه الحالة فان المعالج يجب ان تكون له الامكانية للعودة الى البرنامج في الموقع الذي تفرع منه لتنفيذ الدالة التي تم استدعائها مع المحافظة على جميع القيم الوسطية, لذلك فان العنوان الحالي للبرنامج (اي الايعاز الذي كان من المفروض ان ينفذ قبل ان يتفرع المعالج الى تنفيذ الدالة) سيدفع الى المكدس ليتم حفظة, وعندما ينتهي المعالج من تنفيذ الدالة, فان العنوان الذي تم خزنة سيتم سحبة من المكدس ليوجه المعالج اليه لاكمال التنفيذ, نفس الشيء بالنسبة للنتائج الوسطية سيتم حفظها ودفعها الى المكدس لكي يتم العمل عليها عند العودة من تنفيذ الدالة.

4.4.2 المكدس يدعم طريقة الاستدعاء الذاتي. والمكدس يستخدم ايضا بواسطة المترجمات بعملية معالجة او تقييم التعابير وتوليد شفرة لغة الماكنة.

4.4.3 يستخدم المكدس بشكل كبير في انظمة الحواسيب الحديثة, مثال الحاسبات الشخصية الحديثة تستخدم المكدس على مستوى معمارية الحاسوب, حيث يستخدم في التصميم الاساس لنظام التشغيل لاغراض تنفيذ المقاطعة واستدعاء دوال نظام التشغيل, ومن بين الاستخدامات الاخرى, فان المكدس يستخدم لتشغيل او تنفيذ ماكنة جافا الافتراضية Java virtual machine ولغة جافا نفسها لديها صنف يسمى المكدس, والذي من الممكن استخدامة من المبرمج. المكدس في كل مكان, انظمة الحاسوب التي تعتمد على المكدس هي واحدة من التي تخزن المعلومات بشكل مؤقت في المكدس بدلا من مسجلات وحدة المعالجة المركزية.

4.4.4 تطبيق اخر للمكدس, نفرض انك تقوم بعملية جمع أعداد كبيرة جدا, على سبيل المثال افرض انك ترغب بجمع العددين

- 353,120,457,764,910,452,008,700,786
- 234,765,000,129,654,080,277,543

اولا لاحظ التالي انه من الصعب تمثيل اي من العددين كمتغير من نوع الاعداد الصحيحة (long int) لان هذا المتغير لايمكنة من استيعاب العدد الكبير اعلاه. هذه المشكلة بالامكان حلها وذلك بمعاملة العدد كسلسلة رمزية من الارقام, ويتم خزنهم بمكدسين, بعدها تجرى عملية الاضافة او الجمع بسحب ارقام من المكدسين.

4.4.5 ايجاد نتيجة تعبير حسابي: المكدس مفيد لايجاد نتيجة تعبير حسابي, افرض لديك التعبير التالي

5 * 3 +2 + 6 * 4

من الممكن ايجاد نتيجة التعبير وذلك بايجاد, اولا نتيجة حاصل ضرب 5 في 3 , بعد ذلك تخزن النتيجة في متغير مثلا A, ثم قم باضافة الرقم 2 الى المتغير A واخزن النتيجة بالمتغير A. الان اجري عملية الضرب للرقمين 6 في 4 واخزن الناتج بالمتغير B. عند ذلك ستنتهي العملية باضافة A الى B واترك النتيجة النهائية بالمتغير A .

A = 15 (5×3)

A = 15 + 2 (15 + 2)

= 17

B = 6 × 4 (6 × 4)

=24

A = 17 + 24 (17 + 24)

= 41

هذه الفكرة الاولية من الممكن انجازها بسهولة وكفاءة باستخدام المكدس بعد اجراء بعض التحويرات على التعبير الرياضي.

4.5 التعابير الجبرية Algebraic Expression

التعبير الجبري: هو ترتيب رياضي يحتوي على معاملات وعوامل رياضية, مثل

x + y * z

- ✓ المعاملات: هي كميات (وحدة بيانات) والتي تجرى عليها العمليات الرياضية.
  والمعاملات ممكن ان تكون متغيرات مثل x, y, z او ثوابت مثل 1- ,2, 34, 12 ..الخ
- ✓ العوامل الرياضية: هي رموز تدل على عمليات رياضية او منطقية بين المعاملات, مثال على بعض العوامل الشائعة +, -, *, /, ^, <=
  واقعا هناك ثلاث أنواع من التعابير الرياضية. وهي:

4.5.1 تعبير العلامة القياسية infix: وهذا هو النوع الذي اعتدنا على التعامل معه في الرياضيات منذ الايام الاولى للمدرسة, حيث يكون موضع العامل الرياضي (العلامة الرياضية) بين المعاملين الذين من المفروض ان تجرى عليهم العملية الرياضية وفقا لهذا العامل, مثل

A + B

4.5.2 تعبير العلامة اللاحقة Postfix: في هذا التعبير يكون العامل الرياضي تابع او لاحق الى المعاملين اي بعد المعاملين الذين من المفروض ان تجرى عليهم العملية الرياضية وفقا لهذا العامل, مثل

A B +

4.5.3 تعبير العلامة السابقة Prefix: هنا يكون مكان العامل الرياضي قبل المعاملين الذين من المفروض ان تجرى عليهم العملية الرياضية وفقا لهذا العامل, مثل

+ A B

لاحظ ان تعبير العلامة القياسي لايمكن ان تستخرج نتيجتة باستخدام ترتيب العوامل الرياضية من اليسار الى اليمين داخل التعبير. بينما العلامات الرياضية داخل تعبير العلامة اللاحقة دائما تكون في مكان التقييم الصحيح, عليه فان التقييم او الحساب لتعبير العلامة القياسي ينجز بخطوتين .. الاولى هو تحويل تعبير العلامة القياسية الى مايكافئه من تعبير العلامة اللاحقة, والخطوة الثانية تتضمن حساب تعبير العلامة اللاحقة. سوف نرى ذلك في هذا الفصل, كيف ان المكدس مفيد بالتعامل مع كلا الخطوتين بكفاءة, دعنا اولا نختبر العملية الاساسية للتحويل من تعبير العلامة القياسية الى تعبير العلامة اللاحقة .

4.6 تحويل تعبير العلامة القياسية الى تعبير العلامة اللاحقة

4.6.1 اسباب استخدام تعبير العلامة اللاحقة وتعبير العلامة السابقة

بالتاكيد فان القاريء ربما يتسائل لماذا نستخدم هذا الشكل الغريب من تعابير العلامة السابقة والعلامة اللاحقة.

عندما يكون لدينا تعبير علامة قياسية بسيط, فسيكون من المفاجأ ان تعرف ان تعبير العلامة القياسية ليس بسيط كما يبدو, خصوصا عندما ترغب بايجاد نتيجتة. لغرض حساب قيمة تعبير العلامة القياسية فانك تحتاج الى الاعتماد على افضلية العوامل والصفات المشتركة معة, مثال:

التعبير (4*5+3) من الممكن ايجاد نتيجته على انه يساوي 32 وهو يعني (4*(5+3)), او من الممكن لشخص اخر يجد نتيجته على انها 23 بمعنى ((4*5)+3). السؤال هو اي من هاتين النتيجتين هي الصحيحة.

ولحل هذه المشكلة فان استخدام العوامل الرياضية المرافقة للتعبير الرياضي سوف لايكون اختياريا او عشوائيا وانما يخضع لضوابط وقواعد تمثل اسبقية استخدام كل عامل رياضي في التعبير. ان اسبقية العوامل تتحكم بترتيب وتنظيم عملية حساب النتائج للعمليات الحسابية المختلفة ضمن التعبير الحسابي الواحد.

القاعدة العامة تشير الى ان العامل ذو الاسبقية الاعلى يتم حسابة او تطبيقة قبل العامل ذو الاسبقية الاوطأ.

ان استخدام هذه الاسبقيات في الحاسوب سيولد نوع من التعقيد ولذلك فان تعبير العلامة القياسي من الصعب تحليلة, لانه يحتاج الى تحديد افضلية العوامل, المحددات, وكسر التعادل (اي عندما يكون عاملين بنفس الافضلية), وهذا يجعل تقييم الحاسوب للتعبير صعب اكثر من اللازم.

هذا التعقيد من الممكن تجاوزه عند استخدام تعبير العلامة السابقة وتعبير العلامة اللاحقة لانهما لايعتمدان على افضلية العوامل, المحددات, وكسر التعادل. لذا فان تقييم التعابير بهذه الصيغ يكون اسهل.

جدول 4.1 : يوضح اسبقية العوامل الرياضية والمنطقية في الحاسوب

العوامل	الافضلية	الملاحظات
كل العوامل الاحادية, ^	6	^ تعني الرفع الى اس معين, اما العوامل الاحادية فهي تعني العوامل الرياضية التي تسبق معامل واحد فقط مثل

		.(-7)
/ , * , %	5	(%) تشير الى باقي القسمة
+ , - الثنائية	4	الثنائية تعني انها تستخدم معاملين واحد قبل العامل الرياضي والاخر بعده مثل (4 + 12).
= =, <, >, <=, >=, !=	3	تمثل العلامات المنطقية فقط
Not (!)	2	النفي في العلامات المنطقية
&&, ||	1	علامتي and, or
=	0	المساواة التي تستخدم باسناد قيمة معينة الى متغير

4.6.2 خوارزمية تحويل تعبير العلامة القياسي الى تعبير العلامة اللاحقة

1. يعامل تعبير العلامة القياسي على انه سلسلة من الرموز (معاملات وعوامل رياضية) ويتم فحصها او قراءة هذه الرموز واحد بعد الاخر من اليسار الى اليمين (يعتبر تعبير العلامة القياسية هو المدخل ويخزن بسلسلة رمزية (string)).
2. يتم استخدام المكدس كخزان وسطي (يكون المكدس ابتداءا فارغ), وسلسلة رمزية للمخرجات تكون بداية خالية من اي رمز.
3. افحص العنصر اللاحق بالمدخلات (اي ضمن تعبير العلامة القياسية).
4. اذا كان هذا العنصر معامل (متغير او ثابت) يتم وضعة في سلسلة المخرجات مباشرة.
5. اذا كان قوس مفتوح يتم دفعة الى المكدس.
6. اذا كان عامل رياضي قم بمايلي:

a) اذا كان المكدس فارغ ادفع العامل الى المكدس. اذهب الى الخطوة 7
b) اذا كان ماموجود باعلى المكدس هو قوس مفتوح ادفع العامل الى المكدس. اذهب الى الخطوة 7.
c) اذا كان هذا العامل له اسبقية اعلى من العامل في اعلى المكدس ادفع هذا العامل الى المكدس. اذهب الى الخطوة 7.
d) اذا كان هذا العامل له اسبقية اوطأ من العامل الموجود في اعلى المكدس, في هذه الحالة اسحب العامل من اعلى المكدس واخرجة ليتم وضعة في سلسلة المخرجات, اذهب الى الخطوة a في بداية الفقرة 6

7. اذا كان قوس مغلق اسحب العوامل من المكدس واحد بعد الاخر, وكل عامل يتم سحبه يوضع في سلسلة المخرجات واحد بعد الاخر, لحين تصل الى القوس المفتوح. عملية السحب ستهمل القوس المفتوح. (عملية الاضافة الى سلسلة المخرجات هي عملية جمع (concatenation))
8. اذا كان هناك مزيد من المدخلات اذهب الى الخطوة 3.
9. اذا لم يكن هناك مزيد من المدخلات, اسحب العوامل المتبقية في المكدس واضفهم الى سلسلة المخرجات (واحد بعد الاخر).

4.6.3 امثلة لتحويل تعبير العلامة القياسي الى تعبير العلامة اللاحقة (هنا باستخدام الاسبقيات وليس المكدس)

مثال 1: تعبير العلامة القياسية a + b * c

اسبقية عامل الضرب اكبر من اسبقية عامل الجمع // a + (b * c)

تحويل الضرب الى تعبير العلامة اللاحقة // a + (b c *)

تحويل الجمع الى تعبير العلامة اللاحقة // a (b c *) +

ازالة الاقواس // a b c * +

شكل تعبير العلامة اللاحقة a b c * +

ملاحظة: لاتوجد حاجة الى الاقواس في تعبير العلامة اللاحقة, أي ان تعبير العلامة اللاحقة لايحتوي اقواس.

مثال 2: تعبير العلامة القياسية (A + B) * C

لاحظ هنا ان اسبقية الاقواس اكبر من اسبقية العمليات الاخرى (اكبر من اسبقية عامل الضرب), لذلك سيكون داخل القوس له اسبقة عليا.

تحويل الجمع الى تعبير العلامة اللاحقة // (A B +) * C

تحويل الضرب الى تعبير العلامة اللاحقة // (A B +) C *

ازالة الاقواس // AB+C*

شكل تعبير العلامة اللاحقة A B + C *

لاحاجة للاقواس باي شكل من الاشكال

مثال 3: تعبير العلامة القياسية a + (b * c / d)

بداية فان الاسبقة للاقواس, داخل الاقواس هناك عمليتان الضرب والقسمة, ان اسبقية علامة الضرب وعلامة القسمة متساويتان (عند تساوي علامتين بالاسبقية فان العلامة التي على اليسار تنفذ اولا),

تحويل الضرب الى تعبير العلامة اللاحقة // a + (b c * /d)

تحويل القسمة الى تعبير العلامة اللاحقة // a + (b c * d /)

تحويل الجمع الى تعبير العلامة اللاحقة // a (b c * d /) +

ازالة الاقواس // abc*d/+

مثال 4: مثال توضيحي لكيفية تنفيذ خوارزمية تحويل تعبير العلامة القياسية باستخدام المكدس:

تعبيـــــــر العلامـــــــة القياســـــــية a+b*c-d

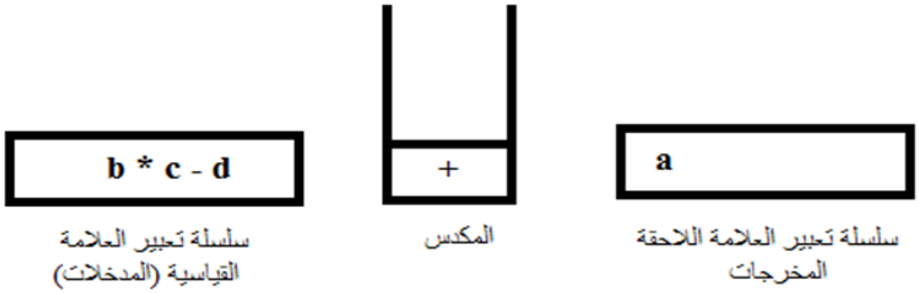
ابتداءا فان المكدس يكون فارغ وسلسلة التعبير اللاحق ليس فيها اي رموز كما في الشكل 4.4.

شكل 4.4: وضع المكدس وسلسلة المخرجات والمدخلات في بداية التنفيذ.

الان, الرمز الاول الذي يتم فحصه هو 'a' والذي ستتم اضافتة الى سلسلة تعبير العلامة اللاحقة (لانه ليس علامة رياضية) لاحظ الشكل 4.5.

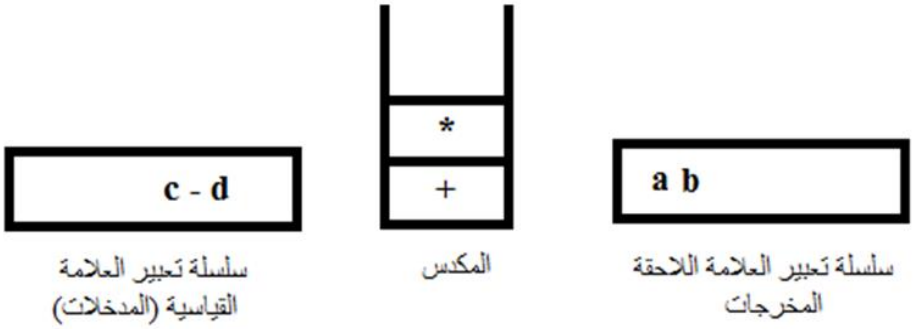
شكل 4.5: وضع المكدس بعد قراءة الرمز الاول.

الرمز الثاني الذي يتم فحصة هو علامة الجمع '+' وهذا بالطبع عامل رياضي والعوامل الرياضية تضاف الى المكدس, الشكل 4.6 يوضح النتيجة.



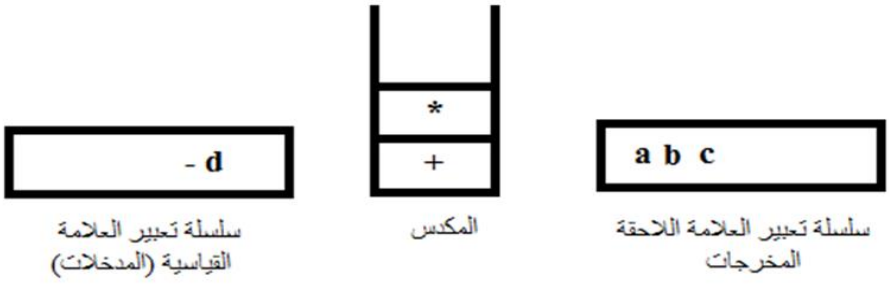
الشكل 4.6: وضع المكدس بعد قراءة الرمز الثاني

الرمز التالي الذي سيتم قرائتة هو الرمز 'b' والذي سيوضع في سلسلة تعبير العلامة اللاحقة. بعدها يكون الدور للرمز الذي سيتم قرائته هو علامة الضرب '*' والذي هو عامل رياضي. الان هذا يجب ان يودع في المكدس ويجب ان نلاحظ ان اعلى المكدس يحتوي العامل الرياضي '+' والذي هو ذو اسبقية اقل من عامل الضرب, لذلك فان عامل الضرب سيدفع الى اعلى المكدس, لاحظ وضع المكدس في الشكل 4.7.



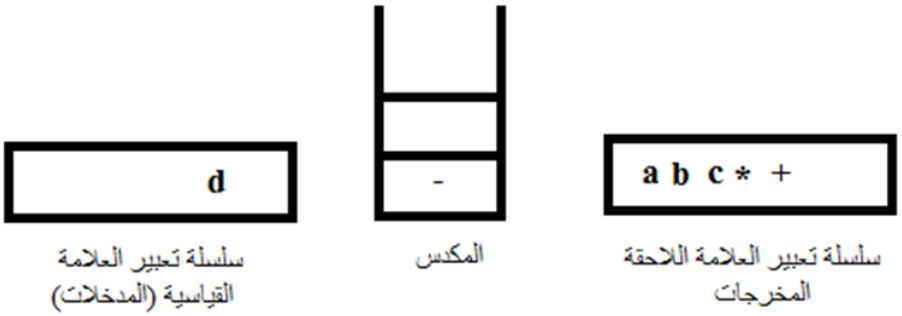
شكل 4.7: وضع المكدس وسلسلة المخرجات بعد قراءة رمزين اضافيين.

الرمز اللاحق هو 'c' والذي سيوضع في تعبير العلامة اللاحقة, شكل 4.8.



شكل 4.8: المكدس بعد اضافة الرمز c

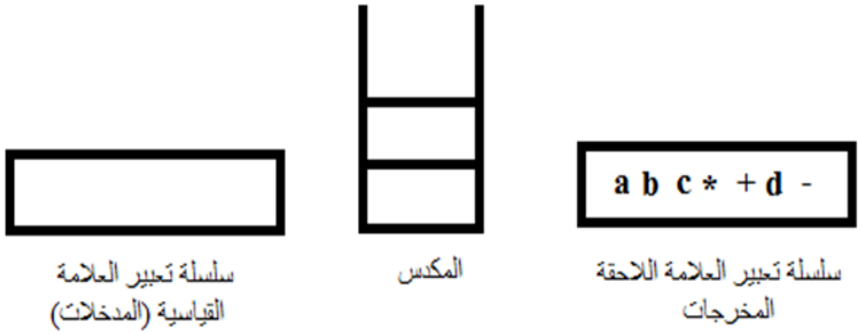
الرمز الاخر الذي سيتم قرائته هو علامة الطرح '-' وهذا عامل رياضي المفروض ان يوضع في المكدس, لاحظ ان اعلى المكدس يحتوي علامة الضرب '*' والذي هو ذو اسبقية اعلى من عامل الطرح ونظرا لعدم امكانية ان يوضع عامل رياضي ذو اسبقية دنيا فوق عامل رياضي ذو اسبقية اعلى في المكدس, لذا سيتم سحب العامل '*' من المكدس ويضاف الى سلسلة تعبير العلامة اللاحقة (المخرجات). حتى الان فان المكدس ليس فارغ. الان بعد سحب علامة الضرب فان علامة الجمع '+' هي في اعلى المكدس وهذا العامل مساوي بالاسبقية لعامل الطرح, ولذا فان عاملين متساويين بالاسبقية لايمكن ان يكونا احدهما فوق الاخر مباشرة بالمكدس, لذا فان العامل '+' يسحب ايضا من المكدس ويوضع في سلسلة تعبير العلامة اللاحقة, عند ذاك يتم دفع العامل '-' الى المكدس, لاحظ الشكل 4.9.



شكل 4.9 : وضع المكدس بعد اضافة العلامة (-)

الرمز الذي سيقرا الان هو الرمز 'd' والذي سيضاف الى تعبير العلامة اللاحقة, وبعد قراءة الرمز d نكون قد قرأنا جميع الرموز في تعبير العلامة القياسية وبهذا وصلنا الى حالة هي عدم وجود اي رمز اخر لم يتم قرائتة وعلية فان جميع العناصر الموجودة في المكدس سيتم سحبها واحد بعد الاخر, وتتم اضافتها الى

سلسلة تعبير العلامة اللاحقة بالتعاقب. فيتم هنا سحب عامل الطرح ويضاف الى سلسلة تعبير العلامة اللاحقة. لذا بعد ان يتم قراءة جميع الرموز فان المكدس وسلسلة تعبير العلامة اللاحقة سيكونان كما في الشكل 4.10.



شكل 4.10: وضع المكدس بعد قراءة اخر رمز.

النتيجة النهائية ستكون:

- abc*+d- سلسلة تعبير العلامة اللاحقة

تمرين 1: حول تعبير العلامة القياسية في الامثلة التالية الى تعبير العلامة اللاحقة باستخدام الاسبقيات واستخدام المكدس. (*الاجوبة في نهاية الفصل*)

1. (a + b) * (c – d)
2. a – b / (c + d * e)
3. ((a + b) * c – (d – e))/(f + g)

مثال 5: مطلوب تحويل تعبير العلامة القياسي 2*3/(2-1)+5*3 الى تعبير العلامة اللاحقة باستخدام المكدس؟

2*3/(2-1)+5*3

سلسلة المخرجات تعبير العلامة اللاحقة	المكدس	عناصر تعبير العلامة القياسية (المدخلات)
2	فارغ	2
2	*	*
23	*	3
23*	/	/
23*	/(	)
23*2	/(	2
23*2	-/(	-
23*21	-/(	1
23*21-	/	)
23*21-/	+	+
23*21-/5	+	5
23*21-/5	+*	*
23*21-/53	+*	3
23*21-/53*+	فارغ	
23*21-/53*+	تعبير العلامة اللاحقة	

4.7 حساب (ايجاد نتيجة) تعبير العلامة اللاحقة

بالامكان حساب نتيجة تعبير العلامة اللاحقة باستخدام المكدس. خوارزمية حساب نتيجة تعبير العلامة اللاحقة هي:

1. ابدأ بمكدس فارغ.

2. تعبير العلامة اللاحقة يتكون عادة من معاملات وعوامل رياضية, بالامكان اعتبارها كسلسلة من الرموز, يتم قرائتها او فحصها من اليسار الى اليمين تباعا.
3. يتم قراءة رمز واحد من السلسلة في كل مرة.
4. اذا كان الرمز هو معامل يتم دفعة الى داخل المكدس.
5. اذا كان الرمز عامل رياضي, عندها يتم مايلي:

a) سحب اثنين من المعاملات من اعلى المكدس (اي المعامل الاعلى في المكدس والمعامل الذي يليه).
b) اجراء العملية الحسابية على المعاملين الذين تم سحبهما ووفقا للعامل الرياضي الذي تم قرائتة.
c) يتم دفع النتيجة المستحصلة من الخطوة b الى داخل المكدس (توضع عادة في اعلى المكدس)

6. اذا لم تكن السلسلة فارغة الانتقال الى الخطوة 3.

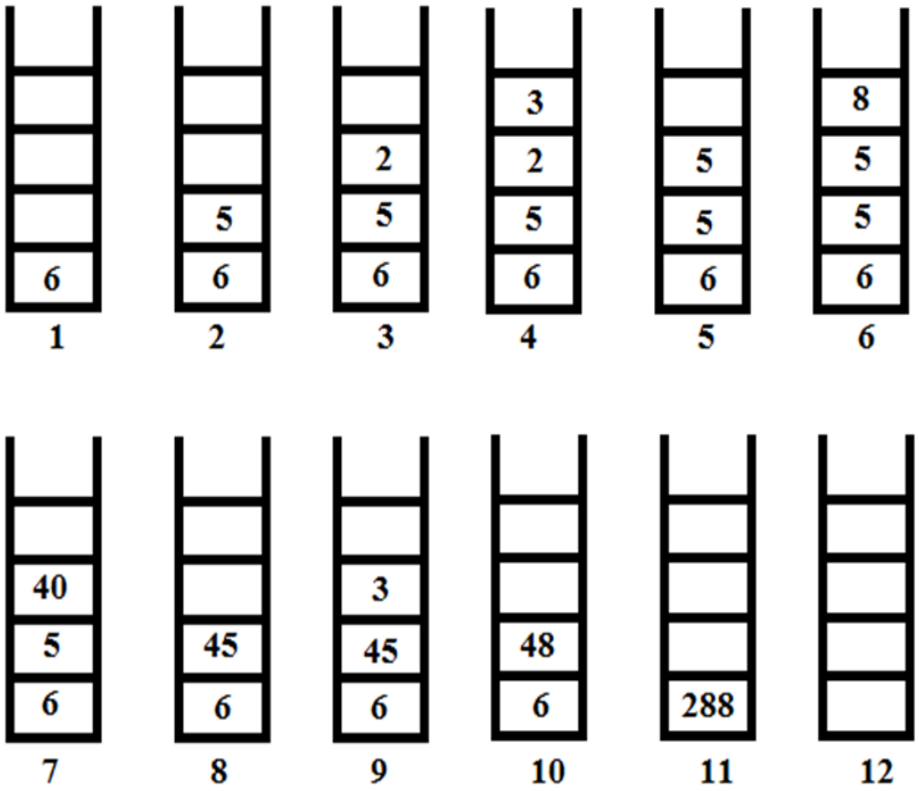
مثال 6: مثال توضيحي لايجاد نتيجة تعبير العلامة اللاحقة التالي:

6 5 2 3 + 8 * + 3 + *

الحل: يتم التعامل مع تعبير العلامة اللاحقة على انه سلسلة من الرموز (الشكل 4.11 سيساعدك على متابعة الحل), وفي كل مرة يقرأ احد هذه الرموز وذلك من اليسار الى اليمين. اولا يقرأ الرقم 6 ويدفع الى داخل المكدس الفارغ (الارقام (المعاملات) تدفع الى المكدس) بعدها ياتي الرقم 5 فيدفع الى المكدس ايضا, ثم ياتي الرقم 2 ويدفع الى المكدس ايضا حسب القاعدة, الرمز الاخر هو الرقم 3 ويدفع ايضا الى المكدس, بعد دفع الرقم 3 الى المكدس نصل الى علامة الجمع وكما اسلفنا ان العوامل الرياضية لا تدفع الى المكدس ومايحدث هو ان يتم سحب رقمين من اعلى المكدس واجراء هذه العملية الرياضية عليهما, هنا في اعلى المكدس الرقم 3 وبعدة الرقم 2 يتم سحبهما من المكدس وتجرى عليهم عملية الجمع

ليكون الناتج 5 هذا الناتج يعاد دفعة الى المكدس وطبعا سيكون في اعلى المكدس (الخطوة 5), تستمر عملية قراءة رموز تعبير العلامة اللاحقة فنقرأ الرقم 8 وهذا رقم فيدفع الى المكدس كما في الخطوة (6) في الشكل4.11, بعدها نقرأ علامة الضرب وبهذا يتم سحب رقمين من اعلى المكدس وهما الرقم 8 والذي يلية الرقم 5 لتتم عليهم عملية الضرب ويكون الناتج 40 الذي يدفع الى اعلى المكدس (خطوة 7) في الشكل 4.11 تستمر عملية قراءة رموز تعبير العلامة اللاحقة حيث علامة الجمع عندها يسحب اعلى رقمين وهم 40 والرقم 5 وتجرى عملية الجمع والناتج هو 45 يدفع الى اعلى المكدس, بعدها يتم قراءة الرقم 3 ويدفع الى اعلى المكدس ثم نقرأ علامة الجمع ونسحب الرقم 45 والرقم 3 من اعلى المكدس وتجرى عليهم عملية الجمع ليكون الناتج 48 يدفع الى اعلى المكدس (الخطوة 10) في الشكل 4.11, واخيرا تتم عملية قراءة عامل الضرب ويتم سحب الرقم 48 والرقم 6 لاجراء عملية الضرب وينتج الرقم 288 الذي يدفع الى المكدس وبهذا تنتهي العملية الرياضية نظرا لانتهاء جميع رموز تعبير العلامة اللاحقة, ولم يبقى بالمكدس سوى الناتج الذي هو 288.

العملية تتوقف عندما تنتهي جميع العوامل الرياضية والمعاملات في السلسلة (تعبير العلامة اللاحقة), نتيجة حساب التعبير المتحصلة يتم سحبها من المكدس وستكون هي الوحيدة في المكدس اي معامل واحد يبقى بالمكدس.



شكل 4.11: خطوات حل المثال 6.

مثال 7 : مثال اخر توضيحي لايجاد نتيجة تعبير العلامة اللاحقة باستخدام المكدس للتعبير:

623+-382/+*2^3+

تكون البداية باستخدام مكدس فارغ.

623+-382/+*2^3+		
الرمز	**المكدس**	**الملاحظات**
6	6	معامل يدفع الى المكدس
2	6, 2	معامل يدفع الى المكدس

3	6, 2, 3	معامل يدفع الى المكدس
+	6, 5	عامل رياضي, يتم سحب اعلى معاملين من المكدس, وتجرى عليهم العملية الرياضية وفقا للعامل الرياضي هنا عملية جمع (2+3) ثم تعاد النتيجة (5) الى اعلى المكدس
-	1	عامل, يسحب اعلى معاملين ويجرى عليهم عملية الطرح (5-6) وتدفع النتيجة (1) الى اعلى المكدس
3	1, 3	معامل يدفع الى المكدس
8	1, 3, 8	معامل يدفع الى المكدس
2	1, 3, 8, 2	معامل يدفع الى المكدس
/	1, 3, 4	عامل القسمة, يسحب اعلى معاملين بالمكدس وتجرى عليهم القسمة (8/2) وتدفع النتيجة (4) الى اعلى المكدس. **ملاحظة: المعامل الاول الذي يسحب من المكدس يوضع بعد العامل الرياضي بينما المعامل الثاني يوضع قبل العامل الرياضي.**
+	1, 7	جمع المعاملين (4+3) ودفع النتيجة (7) الى اعلى المكدس
*	7	عملية ضرب لاعلى معاملين بالمكدس (1*7) ودفع النتيجة (7) الى اعلى المكدس
2	7, 2	معامل يدفع الى المكدس
^	49	عامل الرفع (اس) لاعلى معاملين بالمكدس (7^2) وتدفع النتيجة (49) الى اعلى المكدس
3	49, 3	معامل يدفع الى المكدس
+	52	اجراء عملية الجمع على اعلى معاملين بالمكدس (49+3) ودفع النتيجة (52) الى اعلى المكدس, وهي اخر عملية وتمثل النتيجة النهائية.

	52	نتيجة حساب تعبير العلامة اللاحقة

مثال 8: تحويل تعبير العلامة القياسي 1+2*3^4/5-6 الى تعبير العلامة اللاحقة باستخدام المكدس.

1+2*3^4/5-6		
سلسلة المخرجات تعبير العلامة اللاحقة	**المكدس**	**رموز تعبير العلامةالقياسية (المدخلات)**
1		1
1	+	+
12	+	2
12	*+	*
123	*+	3
123	^*+	^
1234	^*+	4
1234	/^*+	/
1234^	/*+	
1234^*	/+	
1234^*5	/+	5
1234^*5	-/+	-
1234^*5/	-+	
1234^*5/+	-	

1234^*5/+6	-	6
1234^*5/+6-	فارغ	تعبير العلامة اللاحقة

مثال 9: حول تعبير العلامة القياسي 4*(2+3)-1 الى تعبير العلامة اللاحقة باستخدام المكدس.

4*(2+3)-1		
سلسلة المخرجات تعبير العلامة اللاحقة	**المكدس**	**رموز تعبير العلامة القياسية (المدخلات)**
1		1
1	-	-
1	-(	(
12	-(	2
12	-(+	+
123	-(+	3
123+	-(	)
123+	-	
123+	-*	*
123+4	_*	4
123+4*	-	
123+4*-	فارغ	تعبير العلامة اللاحقة

مثال 10: حول تعبير العلامة القياسي ((2+2)*(3-2))/(5-6)*(8+4) الى تعبير العلامة اللاحقة.

((2+2)*(3-2))/(5-6)*(8+4)		
سلسلة المخرجات تعبير العلامة اللاحقة	**المكدس**	**عناصر تعبير العلامة القياسي (المدخلات)**
	)	(
4	(	4
4	+)	+
48	+)	8
48+		)
48+	*	*
48+	*)	(
48+6	*)	6
48+6	*(-	-
48+65	*(-	5
48+65-	*	)
48+65-	*/	/
48+65-*	/	
48+65-*	/)	(
48+65-*	/((	(

48+65-*3	/((	3
48+65-*3	-)/	-
48+65-*32	-)/	2
48+65-*32-	/(	)
48+65-*32-	*)/	*
48+65-*32-	)*)/	(
48+65-*32-2	)*)/	2
48+65-*32-2	+)*)/	+
48+65-*32-22	+)*)/	2
48+65-*32-22+	*)/	)
48+65-*32-22+*	/	)
48+65-*32-22+*/	فارغ	تعبير العلامة اللاحقة

الان لو اردنا ان نحول تعبير العلامة اللاحقة الى تعبير العلامة القياسي. تتم بعملية بسيطة وهي مشابهة جدا الى عملية ايجاد ناتج تعبير العلامة اللاحقة, الفارق هو في ايجاد الناتج عندما نقرا علامة رياضية كنا نسحب اعلى عنصرين في المكدس وتجرى عليهم العملية الرياضية ثم يدفع الناتج الى اعلى المكدس, في هذه الحالة لغرض التحويل الى العلامة القياسة نعمل نفس الشيء عندما نقرا علامة رياضية نسحب اعلى عنصرين في المكدس ونضع العلامة بينهما ثم ندفع هذه النتيجة الى اعلى المكدس.. هذا الموضوع سيكون اكثر وضوحا عند متابعة المثال 11.

مثال 11 : تحويل تعبير العلامة اللاحقة الى تعبير العلامة القياسية

a b – c / d e + f – *

a b – c / d e + f – *		
الملاحظات	**المكدس**	**عناصر تعبير العلامـــــــة اللاحقة**
معامل يدفع الى المكدس	a	a
معامل يدفع الى المكدس	a,b	b
عامل رياضي يتم سحب اعلى عنصرين بالمكدس وهما a, b ووضع العلامة الرياضية بينهما, يدفع الناتج الى المكدس		-
تعتبر النتيجة عنصر واحد عند دفعها الى المكدس لذا توضع بين قوسين للتعامل معها على انها عنصر واحد	(a-b)	
معامل يدفع الى المكدس	(a-b),c	c
عامل رياضي, لذا يسحب اعلى عنصرين بالمكدس وهما (a-b), c وتوضع بينهم العلامة الرياضية لتكون عنصر واحد يدفع للمكدس		/
يكون العنصر الاول الذي يسحب من المكدس هو العنصر الذي يوضع بعد العلامة الرياضية بينما العنصر الثاني الذي يسحب من المكدس يوضع قبل العلامة الرياضية	(a-b)/c	
معامل يدفع الى المكدس	(a-b)/c,d	d
معامل يدفع الى المكدس	(a-b)/c,d,e	e

عامل رياضي, يسحب اعلى معاملين بالمكدس وهما d, e	(a-b)/c	+
توضع بينهم العلامة الرياضية لتكون عنصر واحد يدفع الى المكدس	(a-b)/c,(d+e)	
معامل يدفع الى المكدس	(a-b)/c,(d+e),f	f
عامل رياضي, يسحب اعلى معاملين في المكدس وهما (d+e), f	(a-b)/c	-
توضع العلامة الرياضية بينهم ليكون عنصر واحد يدفع الى المكدس	(a-b)/c,(d+e)-f	
	(a-b)/c,(d+e)-f	
عامل رياضي يسحب اعلى عنصرين بالمكدس وهما ((a-b)/c), ((d+e)-f) ويوضع بينهم العلامة الرياضية ليتكون عنصر واحد يدفع الى المكدس		*
	(a-b)/c*(d+e)-f	
النتيجة النهائية حيث لايوجد عناصر اخرى اضافية	(a-b)/c*(d+e)-f	

تمرين 2: حول كل تعبير من تعابير العلامة القياسية ادناة الى تعبير العلامة اللاحقة باستخدام المكدس (*الاجابة في نهاية الفصل*)

1. A + B * C – D * E / F
2. A + B ^ C / D * E + F – A
3. A ^ B ^ C / 2 * B + 4
4. A / B + C / D – E * F ^ 2 / B
5. (A + B) * (C – B) / 2 ^ 4

6. (A + B) * (C – D)
7. A – B / (C + D * E)
8. ((A + B) * C – (D – E))/(F + G)

4.8 خوارزمية تحويل تعبير العلامة القياسية الى تعبير العلامة السابقة

ان تحويل تعبير العلامة القياسية الى تعبير العلامة السابقة يتبع الخطوات التالية بعد ان يتم فحص تعبير العلامة القياسية من **اليمين الى اليسار** ويتم تكرار الخطوات التالية على كل عنصر من عناصر التعبير لحين ان يفرغ المكدس.

1. قراءة عنصر جديد من تعبير العلامة القياسية.
2. اذا كان العنصر هو معامل فيتم اضافتة الى سلسلة المخرجات.
3. اذا كان العنصر هو عامل رياضي فيتم اجراء مايلي:
   3.1 اذا كان العامل قوس ايمن " (" يتم وضعة في المكدس.
   3.2 اذا كان العامل قوس ايسر ") " في هذه الحالة يتم سحب جميع العوامل الموجودة في المكدس لحين الوصول الى القوس الايمن. جميع هذة العوامل التي سيتم سحبها من المكدس تكتب في سلسلة المخرجات بالتعاقب (اي واحد بعد الاخر). الاقواس تهمل.
   3.3 اذا كان العنصر عامل رياضي فيتم اجراء مايلي:
      3.3.1 اذا كان المكدس فارغ فيتم اضافة العامل الرياضي الى المكدس.
      3.3.2 اما اذا كان المكدس غير فارغ, عليك بالقيام بمايلي:
         a) ايجاد الافضلية للعامل الرياضي.
         b) ايجاد الافضلية للعامل الرياضي الموجود في اعلى المكدس.
         c) مقارنة الافضليات للعاملين, فاذا كانت افضلية العامل الرياضي المراد اضافتة اعلى من افضلية العامل الرياضي الموجود في اعلى المكدس في هذه

الحالة يتم دفع هذا العامل الى المكدس ليكون في اعلى المكدس. اما اذا كانت افضلية العامل المراد اضافتة اقل من افضلية العامل الرياضي الموجود في اعلى المكدس عندها يتم اخراج العامل الرياضي في اعلى المكدس ويضاف الى سلسلة المخرجات.. تستمر عملية المقارنة مع العامل الذي سيكون في اعلى المكدس بعد سحب العامل السابق وتطبق نفس الطريقة, وهكذا لحين الوصول الى مكدس فارغ او عامل رياضي في اعلى المكدس افضليتة اقل من العامل المراد اضافتة عندها يضاف العامل الى اعلى المكدس.

4. اذا كان هناك عناصر اخرى في تعبير العلامة القياسية, الذهاب الى الخطوة 1. في خلاف ذلك الذهاب الى الخطوة 5.
5. افراغ جميع محتويات المكدس (ازالة كل العوامل من اعلى المكدس) واضافتها الى سلسلة المخرجات.
6. بعد الانتهاء من افراغ المكدس يتم عكس رموز سلسلة المخرجات (اي يكون الرمز الايسر باليمين والرمز الذي باليمين باليسار) حيث ستكون هذه السلسلة الناتجة تمثل تعبير العلامة السابقة.

ملاحظة: بالامكان عكس رموز تعبير العلامة القياسية ابتداءا وتطبيق الخوارزمية اعلاه عليها, دون الحاجة الى عكس النتيجة النهائية, لكن هذا لايتبع في السلاسل الحرفية اثناء البرمجة.

ملاحظة: اهم الفروق بين خوارزمية التحويل الى العلامة اللاحقة والتحويل الى العلامة السابقة هو:

- في حالة التحويل الى العلامة اللاحقة فان السلسلة تقرا من اليسار الى اليمين بينما في حالة التحويل الى العلامة السابقة فان السلسلة تقرا من اليمين الى اليسار.
- في حالة العلامة اللاحقة لايجوز ان تكون اسبقيتين متساويتين احداهما فوق الاخرى في المكدس, بينما هذا الشرط غير متوفر في التحويل للعلامة السابقة

ويجوز ان تكون علامتين رياضيتين باسبقيتين متساويتين احداهما فوق الاخرى.

- لاحظ ايضا ان الاقواس تستخدم بالعكس, حيث ان القوس الايمن "(" في حالة العلامة اللاحقة هو من يفرغ المكدس لغاية القوس الايسر")" , بينما في حالة العلامة السابقة فان القوس الايسر هو من يفرغ المكدس لغاية القوس الايمن.
- النتيجة التي نحصل عليها في سلسلة الاخراج يجب ان تعكس لتكوين النتيجة النهائية.

مثال 12: حول تعبير العلامة القياسية **(a – b) / c * (d + e – f / g)** الى تعبير العلامة السابقة

(a – b) / c * (d + e – f / g)			
الملاحظات	**تعبير العلامة السابقة (المخرجات)**	**المكدس**	**رموز تعبير العلامة القياسية (المدخلات)**
قوس ايمن يدفع للمكدس		(	)
رمز يكون في المخرجات	g	(	g
علامة رياضية تدفع للمكدس	g	/(	/
رمز يكون في المخرجات	gf	/(	f
علامة رياضية تدفع للمكدس, ولكون اسبقية علامة الطرح اقل	gf	-/(	-

من القسمة فتخرج القسمة لتكون في المخرجات			
تدخل بدلا عنها في المكدس علامة الطرح	gf/	)-	
رمز يكون في المخرجات	gf/e	)-	e
علامة رياضية (الجمع) تدفع اعلى المكدس لانها مساوية للعلامة الطرح في الاسبقية	gf/e	)-+	+
رمز يرسل الى المخرجات	gf/ed	)-+	d
قوس مغلق يخرج جميع ماموجود في المكدس لغاية القوس المفتوح	gf/ed+-		(
علامة الضرب تدفع الى المكدس الفارغ	gf/ed+-	*	*
رمز يرسل الى المخرجات	gf/ed+-c	*	c
علامة القسمة تدفع الى المكدس اعلى علامة الضرب لانها متساوية في الاسبقية مع علامة الضرب	gf/ed+-c	*/	/
قوس مفتوح يدفع للمكدس	gf/ed+-c	*/)	)
رمز يرسل الى المخرجات	gf/ed+-cb	*/)	b
علامة الطرح تدفع الى المكدس اعلى القوس	gf/ed+-cb	*/)-	-
رمز يرسل الى المخرجات	gf/ed+-cba	*/)-	a
قوس مغلق يفرغ المكدس ويرسلها	gf/ed+-cba-	*/	(

الـى المخرجـات واحـد بعـد الاخـر لحين الوصول الى القوس المفتوح			
	gf/ed+-cba-/*	تعكـس هـذه النتيجـة للحصـول على تعبير العلامة السابقة	
*** / – a b c – + d e / f g**		**تعبير العلامة السابقة**	

مثال 13: حول تعبير العلامـة القياسية (4-1)*5+(2-1)/3*2 الـى تعبير العلامـة السابقة باستخدام المكدس.

2*3/(2-1)+5*(4-1)		
سلسلة المخرجات تعبير العلامة السابقة	**المكدس**	**رموز تعبير العلامة القياسية (المدخلات)**
	(	)
1	(	1
1	-(	-
14	-(	4
14-	فارغ	(
14-	*	*
14-5	*	5
14-5*	+	+
14-5*	+(	)
14-5*1	+(	1

14-5*1	-(+	-
14-5*12	-(+	2
14-5*12-	+	(
14-5*12-	/+	/
14-5*12-3	/+	3
14-5*12-3	*+/	*
14-5*12-32	*+/	2
+/*14-5*12-32	فارغ	هذة النتيجة يتم عكسها
+/*23-21*5-41	**تعبير العلامة السابقة**	

مثال 14 : تحويل تعبير العلامة السابقة * / – a b c – + d e f الى تعبير العلامة القياسية.

ملاحظة: يتم قراءة عناصر تعبير العلامة السابقة **من اليمين الى اليسار**

*** / – a b c – + d e f**		
الملاحظات	**المكدس**	**عناصر تعبير العلامة السابقة (المدخلات)**
معامل يدفع الى المكدس	f	f
معامل يدفع الى المكدس	f,e	e
معامل يدفع الى المكدس	f,e,d	d
عامل رياضي, يتم سحب اعلى معاملين بالمكدس وتوضع العلامة الرياضية بينهما, وهما d, e.. ويتم دفع النتيجة الى اعلى المكدس	f	+
ملاحظة: المعامل الاول الذي يسحب من اعلى المكدس يوضع قبل العلامة الرياضية , بينما المعامل الثاني الذي يسحب من المكدس يوضع بعد العلامة الرياضية.	f,(d+e)	
عامل رياضي		-
	(d+e)-f	
معامل يدفع الى المكدس	(d+e)-f,c	c
معامل يدفع الى المكدس	(d+e)-	b

	f,c,b	
معامل يدفع الى المكدس	(d+e)-f,c,b,a	a
عامل رياضي, يتم سحب اعلى معاملين بالمكدس وهما a, b.. وضع العلامة الرياضية بينهما ودفع النتيجة الى اعلى المكدس	(d+e)-f,c	-
	(d+e)-f,c,(a-b)	
	(d+e)-f	/
	(d+e)-f,(a-b)/c	
		*
تعبير العلامة القياسية	(a-b)/c * (d+e)-f	

مثال 15 : تحويل تعبير العلامة القياسية (a-b)/c * (d+e)-f الى تعبير العلامة السابقة.

(a-b)/c * (d+e)-f		
سلسـلة المخرجـات تعبير العلامة السابقة	المكدس	رموز تعبير العلامة القياسية (من اليمين الى اليسار) (المدخلات)
f		f
f	-	-
	-)	)
fe	-)	e
fe	+(-	+
fed	+(-	d
fed+	-	(
fed+	-*	*
fed+c	-*	c
fed+c	-*/	/
	-*/)	)
fed+cb	-*/)	b
fed+cb	-(*/)-	-
fed+cba	-(*/)-	a
fed+cba-	-*/	(
fed+cba-/	-*	
fed+cba-/*	-	
fed+cba-/*-	فارغ	
-*/-abc+def	**يتم عكس الناتج للحصول على تعبير العلامة السابقة**	

تمرين 3: حول تعابير العلامة اللاحقة التالية الى تعابير العلامة القياسية, موضحا محتويات المكدس بعد كل عملية (الاجوبة في نهاية الفصل) (**تم وضع فوارز بين القيم للتمييز**).

1. 5, 3, + , 2 , * , 6 , 9 , 7 , - , / , -
2. 3, 5, + , 6 , 4 , - , * , 4 , 1 , - , 2 , ^ , +
3. 3, 1, + , 2, ^ 7 , 4, - , 2 , * , + , 5 , -
4. 20, 45, + 20 , 10 , - , 15 , + , *

4.9 بعض البرامج اللازمة لتطبيقات المكدس باستخدام لغة C++

4.9.1 دالة باستخدام C++ لانجاز عمليتي (الدفع والسحب) (push, pop) في مكدس يمثل كمصفوفة.

```
void  PUSH ( int Stack [ ], int &Top, int Val )
{
 if ( Top = = size -1 )  // لابد من التاكد من المكدس ليس مملوء قبل الاضافة
{
   cout << " Stack Overflow " ;
   exit(0) ;    }
else
{
   Top + + ;
```

```
   Stack [ Top ] = Val ;
}
} // end PUSH function
void  POP ( int Stack [ ] , int & Top )
{
int R ;
if ( Top = = -1 ) //  لابد من التاكد من المكدس ليس فارغ قبل الحذف
{
   cout<< " Stack Underflow" ;
   exit(0) ;     }
else
{
   R = Stack [ Top ] ;
  Top - - ;
}
}  // end POP function
```

4.9.2 اكمال الدوال اعلاة لتكون صنف كامل

```
class Stack
{
int Data [ 10] ;
```

```
int Top ;
public:
Stack ( ) { Top = -1; }
void PUSH ( ) ; //  لدفع عنصر في المكدس
void POP ( ) ; //  لسحب عنصر من المكدس
};
void Stack :: PUSH ( )
{
int Val ;
cout <<"Enter the value to be push." ;
cin>> Val ;
if ( Top = = 9 )
{
cout << " Stack Overflow";
exit ( 0) ; }
else
{ Top + + ;
Data [ Top ] = Val ;
}
}
void Stack :: POP( )
{
int R ;
```

```
if ( Top = = -1 )
{ cout<<"Stack Underflow";
exit (0) ;
}
else
{ R = Data [Top ] ;
Top - - ;
}
}
```

4.9.3 برنامج يستخدم لتحويل تعبير العلامة القياسية الى تعبير العلامة اللاحقة

```
#include<iostream>
#include<stdio>
#include<conio>
#include<string>
#include<stdlib>
using namespace std ;

const int size =50;
char infix[size], postfix[size], stack[size];
int top = -1;
int precedence(char ch);          // دالة للحصول على افضليات العوامل
```

```
char pop();                    // دالة لسحب عنصر من المكدس
char topelement();             // اعادة اعلى عنصر في المكدس
void push(char ch);            // دفع عنصر ما الى المكدس
int main( )
{
char ele, elem, st[2];
int prep, pre, popped, j=0, chk=0 ;
strcpy (postfix," ") ;
gets(infix);
for (int i=0 ; infix[i] !=0 ; i++)
{
if (infix [i] != '(' && infix[i] != ')' && infix[i] != '^' && infix[i] != '*'
&& infix[i] != '/' && infix[i] != '+' && infix[i] != '-' )
postfix [j++] = infix[i] ;
else
 if (infix[i] = = '(' )
{
elem = infix [i] ;
push(elem);
}
else
if (infix[i]  = = ')' )
{
while (popped = pop( ) != '(' )
```

```
postfix [j++] = popped ;
}
else
{
elem=infix[i];
pre = precedence (elem) ;    //خزن افضلية العامل الذي ياتي من تعبير العلامة القياسية
ele = topelement ( ) ;
prep = precedence (ele) ;    //خزن الافضلية للعامل الموجود في اعلى المكدس

if(pre > prep)
push(elem);
else
{
while(prep >= pre)
{
if(ele = = '#' )
break;
popped=pop();
ele=topelement();
postfix[j++] = popped ;
prep=precedence(ele);
}
push(elem);
}
```

```
}
}

while ((popped = pop( )) != '#' )
postfix[j++] = popped ;
postfix[j]='\0';

cout<<"\n post fix :"<<postfix<<endl;
system("pause");
return 0;
}

int precedence(char ch)

{
switch(ch)
{
case '^' : return 5;
case '/' : return 4;
case '*' : return 4;
case '+' : return 3;
case '-' : return 3;
default : return 0;
```

```
}
}

char pop()        //دالة لسحب عنصر من المكدس

{
char ret;
if(top!=-1)
{ ret =stack[top];

top--;
return ret;

}
else
return '#';

}

char topelement() // دالة لاعادة العنصر الاعلى في المكدس دون ان تسحبهة من المكدس

{
char ch;
```

```
if(top!=-1)
ch=stack[top];
else
ch='#';
return ch;
}

void push(char ch) // دالة لدفع عنصر الى المكدس
{
if(top!=size-1)
{
top++;
stack[top]= ch;
}
}
```

4.9.4 حساب قيمة تعبير العلامة اللاحقة

```
#include <iostream>
#include <stdlib>
#include <math>
#include <ctype>
using namespace std ;
```

```
const int MAX = 50 ;
class postfix
{
        private :

                int stack[MAX] ;
                int top, nn ;
                char *s ;
         public :
                postfix( ) ;
                void setexpr ( char *str ) ;
                void push ( int item ) ;
                int pop( ) ;
                void calculate( ) ;
                void show( ) ;
} ;
postfix :: postfix( )
{
         top = -1 ;
}
void postfix :: setexpr ( char *str )
```

```
{
        s = str ;
}
void postfix :: push ( int item )
{
        if ( top == MAX – 1 )
                cout << endl << “Stack is full” ;
        else
        {
                top++ ;
                stack[top] = item ;
        }
}
int postfix :: pop( )
{
        if ( top == -1 )
        {
                cout << endl << “Stack is empty” ;
                return NULL ;
        }
        int data = stack[top] ;
        top-- ;
        return data ;
```

```
}
void postfix :: calculate( )
{
        int n1, n2, n3 ;
        while ( *s )
        {
            if ( *s == ' ' || *s == '\t' )
            {
                s++ ;
                continue ;
            }
            if ( isdigit ( *s ) )
            {
                nn = *s – '0' ;
                push ( nn ) ;
            }
            else
            {
                n1 = pop( ) ;
                n2 = pop( ) ;
                switch ( *s )
                {
```

```
                case '+' :
                        n3 = n2 + n1 ;
                        break ;
                case '-' :
                        n3 = n2 – n1 ;
                        break ;
                case '/' :
                        n3 = n2 / n1 ;
                        break ;
                case '*' :
                        n3 = n2 * n1 ;
                        break ;
                case '%' :
                        n3 = n2 % n1 ;
                        break ;
                case '$' :
                        n3 = pow ( n2 , n1 ) ;
                        break ;
                default :
                        cout << "Unknown operator" ;
                        exit ( 1 ) ;
        }
```

```
                    push ( n3 ) ;
                }
                s++ ;
        }
}
void postfix :: show( )
{
        nn = pop ( ) ;
        cout << "Result is: " << nn ;
}

void main( )
{
        char expr[MAX] ;
        cout << "\nEnter postfix expression to be evaluated : " ;
        cin.getline ( expr, MAX ) ;
        postfix q ;
        q.setexpr ( expr ) ;
        q.calculate( ) ;
        q.show( ) ;
}
```

تمرين 4: حول تعابير العلامة القياسية التالية الى مايكافئها من تعابير العلامة اللاحقة موضحها حالة المكدس:

1. ((A – B) * (D / E)) / (F * G * H)
2. (A + B ^ D) / (E – F) + G
3. A * (B + D) / E – F – (G + H / K)
4. A * (B + (C + D) * (E + F) / G) * H
5. A + ((B + C) + (D + E) * F) / G
6. NOT A OR NOT B AND NOT C
7. NOT (A OR B) AND C

تمرين 5: اكتب تعبير العلامة القياسية المكافيء لتعابير العلامة اللاحقة التالية:

1. 10 , 3 , * , 7 , 1 , - , * , 23 , +
2. 12, 7, 3, - , / , 2 , 1 , 5 + , * , +
3. ABCD^*E/+F-

4.10 اجوبة التمارين

تمرين 1:

1. a b + c d - *
2. a b c d e * + / -
3. a b + c * d e - - f g + /

تمرين 2:

1. A B C * + D E * F / –
2. A B C ^ D / E * + F + A –
3. A B ^ ^ C 2 / B * 4 +
4. A B / C D / + E F 2 ^ * B / –
5. A B + C B - * 2 4 ^ /
6. A B + C D – *
7. A B C D E * + / –
8. A B + C * D E – – F G + /

تمرين 3:

1. 13
2. 25
3. 17
4. 1625

تمرين 4:

1. AB – DE / * FG * H * /
2. ABD ^ + E F - / G +
3. A B D + * E / F – G H K / + -
4. A B C D + E F + * G / + * H *
5. A B C + D E + F * + G / +
6. A NOT B NOT C NOT AND OR

7. A B OR NOT C AND

تمرين 5:

1. 10 * 3 * (7 – 1) + 23
2. 12 / (7 – 3) + (1 + 5) * 2
3. A+B*C^D/E-F

هياكل البيانات باستخدام C++

الفصل الخامس

الطابور QUEUE

5.1 المقدمة

في حياتنا اليومية هناك الكثير من الحالات التي نرى فيها الناس يقفون على هيئة طوابير لشراء سلعة او انجاز عمل, هي موجودة غالبا لاي خدمة.. من دفع الفواتير في السوبرماركت الى صعود الباص وليس انتهاءا في البنوك والمصارف. فعندما يكون مجهز الخدمة غير قادر على انجاز خدمة عميل قبل ان يصل العميل اللاحق فعند ذاك نحتاج الى الطابور. تخيل مايحدث عندما يخدم مقدم الخدمة (مثلا المحاسب في البنك) شخص ما بشكل عشوائي قبل الاخرين الذين ربما وصلوا قبله دون مراعاة لزمن وصول الاشخاص للحصول على هذه الخدمة.

تشبه قوائم الانتظار في الحاسبات طوابير الانتظار في الحياة اليومية عندما يقف الناس على هيئة طابور لشراء سلعة أو إنجاز عمل، ويطبق في كلاهما مبدأ من يأتي أولاً يخدم أولاً، وفي الحاسوب يطبق نفس المبدأ.

شكل 5.1: طابور من الحياة اليومية.

5.2 ماهو الطابور

الطابور مشابهة للمكدس ماعدا امكانية اضافة البيانات من خلف الطابور والحذف من الامام. كتابة برنامج الطابور هو اكثر صعوبة من كتابة برنامج المكدس, في الطابور نحافظ على اثنين من المؤشرات من نوع الاعداد الصحيحة واحد يحدد النهاية الامامية والثاني يحدد النهاية الخلفية للطابور.

الطابور هو مثال لهياكل البيانات الخطية, هو نوع خاص لتجميع البيانات بحيث ان البيانات في التجميع يتم المحافظة عليها بترتيب يتناسب مع العمليات التي من الممكن ان تجرى على هذا الترتيب, من هذه العمليات اضافة بيانات الى نهاية الطابور وحذف البيانات من بداية الطابور. وهذا يجعل الطابور هيكل بياني يعتمد على مبدأ الداخل اولا يخرج اولا ((First In First Out (FIFO) ان هذا النظام FIFO يعني ان العنصر الاول الذي يضاف الى الطابور يخرج اولا, ولذلك فان اضافة عنصر جديد يعني ان جميع العناصر التي اضيفت قبله يجب ان تخرج قبل ان يخرج هذا العنصر الجديد.

كمثال على الطابور السيارات التي تقف واحدة خلف الاخرى في محطة البنزين لغرض التزود بالوقود, الانتظار بالدور على دفع فاتورة في محل, الانتظار بالدور لقطع تذكرة, الانتظار في الطابور في نقاط التفتيش الامني...الخ.

وفي مجال الحاسبات فان الطابور يعتبر من الهياكل المعروفة جدا في انظمة التشغيل وشبكات الحاسوب.

والطابور من الممكن ان ينمو كبرا ليصبح طويل, ولكن فقط عنصر واحد من الممكن ان يدخل في الوقت الواحد وكذلك عنصر واحد يخرج او يخدم في الوقت الواحد, ويتبع الطابور هيكلية الداخل اولا خارج اولا بشكل عام. وهناك نوعين من الطوابير هما الطابور الخطي والطابور الدائري.

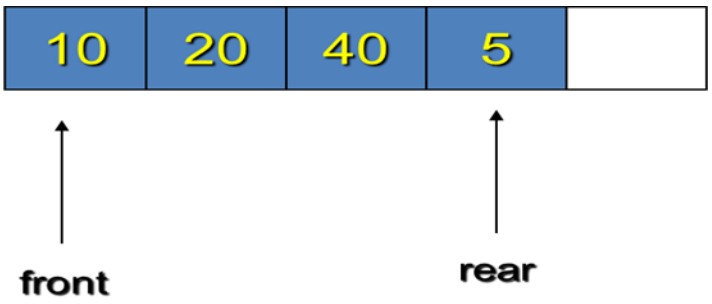
5.3 بعض تطبيقات الطابور

1. عندما يكون لدينا مصادر مشتركة مثل طابعة تستلم اوامر طباعة من اكثر من حاسوب او تستلم من حاسوب واحد اكثر من امر طباعة, في هذه الحالة فان امر طباعة واحد سيتم تنفيذه بينما الاوامر الاخرى تنتظر دورها للتنفيذ, ويتم تنفيذها حسب اسبقية وصولها (او وفقا لخوارزمية محددة بالتنفيذ).
2. اذا ماعملنا على الحاسوب وعملنا ضغط سريع بالماوس على اكثر من ملف بفارق وقت قليل جدا, وكان الملف الاول يحتاج الى بعض الوقت لانجاز عملة, فان الملف الثاني سيبقى بالانتظار لحين اكمال الملف الاول وعندها يكون بالطابور.
3. في داخل الحاسوب فان الاجهزة المختلفة لها سرع انجاز مختلفة, مثلا اجهزة الادخال بطيئة نسبة لاجهزة المعالجة لذلك فان عملية ارسال واستلام البيانات بين هذة الاجهزة يؤدي الى توقف بعض الاجهزة السريعة لانتظار البيانات من الاجهزة البطيئة, في مثل هذة الحالة فان هناك عدة طرق لمعالجة الفرق بالسرعة احدها استخدام الطابور.
4. انظمة مراكز اتصالات الهاتف تتطلب ان تستخدم الطابور لتجعل الاشخاص ينتظرون تلبية طلباتهم وتوفير الخدمة لهم وخصوصا المجانية.
5. الطلبات المختلفة على شبكة الانترنيت ممكن ان تكون ضمن طابور, وخصوصا في اجهزة التوجيه Router.
6. طلبات المقاطعة للبرامج داخل الحاسوب تحتاج الى استخدام الطوابير لتلبية اوامر المقاطعة المختلفة والتي تصل بوقت واحد او اوقات متقاربة.
7. انظمة التشغيل بشكل عام تستخدم الطوابير في مراحل مختلفة لتلبية الطلبات الكثيرة للمستخدم.

5.4 الطوابير الخطية

من الممكن تمثيل الطوابير ماديا في الذاكرة كمصفوفة وتدعى حينها هياكل البيانات المستقرة *static data structure* وعندها سترث جميع صفات المصفوفات, او من الممكن ان تمثل كقائمة موصولة وبهذا تدعى هياكل البيانات الديناميكية *dynamic data structure* وستمتلك جميع خواص وصفات القوائم الموصولة, وبغض النظر عن طريقة تمثيل الطابور فان عمليات الاضافة تحدث في ذيل الطابور بينما الحذف يحدث في راس الطابور.

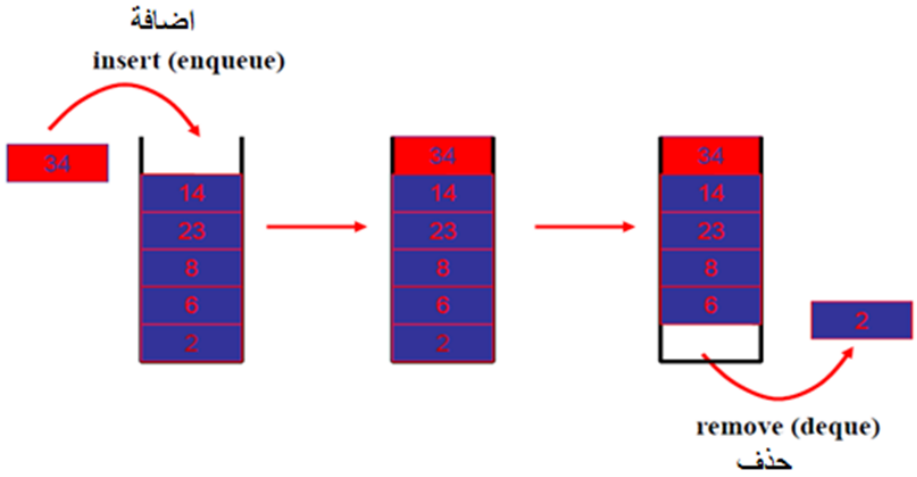
ويحتوي الطابور على مؤشرين احدهما مؤشر البداية والاخر مؤشر النهاية.



شكل 5.2: تمثيل افتراضي للطابور المستخدم في هياكل البيانات.

5.5 تنفيذ قوائم الانتظار (الطابور)

يتطلب تنفيذ قوائم الانتظار على الحاسوب حجز حيز في الذاكرة يستوعب البيانات المراد وضعها في الطابور ووفقا لحجم الطابور, مع وجود مؤشرين يتولى المؤشر الأول الإشارة إلى رأس او بداية موقع الانتظار (مؤشر الراس) Head Pointer بينما يشير الآخر إلى نهاية موقع الانتظار (مؤشر الذيل) Tail Pointer ويشير مؤشر رأس القائمة إلى عنوان أول موقع مشغول بينما يشير مؤشر ذيل القائمة إلى عنوان اخر موقع مشغول في كتلة المواقع المعنونة المخصصة للقائمة. يوضح الشكل 5.3 طريقة التعامل مع قوائم الانتظار.



شكل 5.3: شكل توضيحي لطريقة الاضافة والحذف من الطابور.

5.5.1 مميزات استخدام المصفوفة بالطابور

- تنفيذها بسيط.
- يجب ان يحدد حجم الطابور عند الاعلان.
- فضاء الذاكرة المخصص للطابور من الممكن ان لايستخدم بشكل جيد ويكون هناك فقدان بمواقع الذاكرة عند استخدام عدد قليل من العناصر مع الاعلان عن حجم كبير للطابور.
- لايمكن اضافة عناصر اضافية اكثر من عدد العناصر التي حدد بها حجم الطابور (حجم المصفوفة).

5.5.2 مميزات تنفيذ الطابور باستخدام القوائم الموصولة

- تخصيص ذاكرة بشكل ألي لكل عنصر جديد.
- ربط عناصر الطابور مع بعضها بواسطة المؤشرات.

- استخدام مؤشرين واحد يؤشر على العنصر الاول بالطابور والثاني يؤشر على العنصر الاخير بالطابور.
- تعتبر طريقة جيدة لترشيد استهلاك الذاكرة.

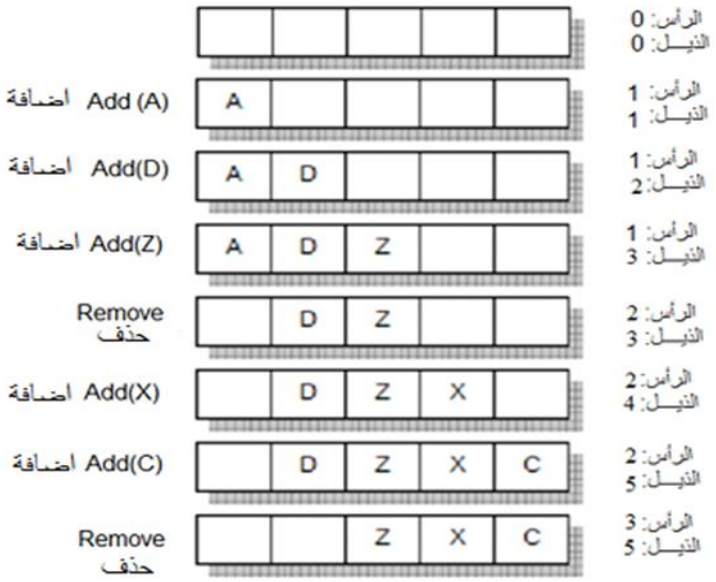
5.6 خزن الطابور في هياكل بيانات مستقرة

هي طريقة خزن الطابور على شكل مصفوفة يحدد حجمها مسبقا.

ويكون هناك مؤشران او دليلان للمصفوفة (متغيرين من نوع الاعداد الصحيحة) يمثلان رأس وذيل الطابور, ويجب ان يحافظ عليهم لتحديد وضع الطابور. رأس الطابور وذيل الطابور بداية يكون بالموقع صفر (او (1-) وهو الافضل لتجنب الموقع صفر في المصفوفة وخصوصا عند كتابة برنامج) وتتغير قيمهم عند الاضافة والحذف.

لنفرض لدينا طابور بحجم 5 عناصر لنرى كيفية الاضافة والحذف للطابور, وكيفية تغير قيم المؤشرات.

C

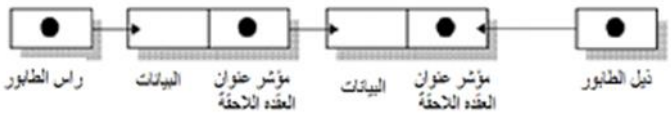


شكل 5.4: عمليات الاضافة والحذف في طابور خطي.

5.7 خزن الطابور في هياكل بيانات ديناميكية

كما هو الحال في المكدس, فان كل عقدة في هياكل البيانات الديناميكية تحتوي على بيانات مع مؤشر يشير الى عنوان العقدة اللاحقة.

الطابور ايضا يحتاج الى مؤشرين احدهما يؤشر الى العقدة الراسية ومؤشر اخر يؤشر الى عقدة الذنب. الشكل 5.5 يوضح خزن الطابورباستخدام القوائم الموصولة لطابور اسمه Queue كل عقدة تتكون من بيانات (DataItem) ومؤشر (NextNode) العقدة اللاحقة.



شكل 5.5: تمثيل الطابور على شكل قوائم موصولة.

عملية الوصول الى هذه العقد تتم من خلال استخدام المؤشرات, والتي سيتم تخصيص فصل لشرح كيفية عمل القوائم.

5.8 خوارزمية اضافة عنصر الى الطابور

1. يتم اولا فحص مؤشر النهاية للطابور, اذا كان يؤشر على الموقع الاخير للطابور (وفقا لحجم الطابور المحدد ابتداءا) (تمثيل الطابور على شكل مصفوفة), فيتم ارسال رسالة بان الطابور لايوجد به فراغ.
2. اذا لم يكن مؤشر النهاية يؤشر على الموقع الاخير للطابور, عنده يتم مايلي:
   A. يتم تغيير موقع المؤشر (مؤشر النهاية) باضافة واحد له (اي ينتقل المؤشر الى الموقع الجديد).
   B. يضاف العنصر الجديد.
3. اذا كان مؤشر النهاية يساوي (1-) عندها يتم زيادة مؤشر النهاية ومؤشر البداية بمقدار واحد (لان ذلك يعني ان مؤشر البداية يساوي (1-) ايضا اي ان الطابور فارغ ولايحتوي على اي عنصر). من الممكن اذا كان الطابور فارغ وضع المؤشرات على الموقع (1-) او (0).

❖ دالة اضافة عنصر الى الطابور الخطي, باستخدام المصفوفات.

```
int q[25], n , front= -1 , rear= -1 , item ;
void insertion( )
{

   if((rear = =n) && (front = = (rear+1)) || (front = = rear+1))
   {
    cout<< ”Queue Overflow\n” ;
   }

    else if (rear = = 0)
```

```
front = rear = 1;

else      if (rear = = n)
rear = 1;

else
rear = rear+1;

cout<< "Enter the item : " ;
cin>> item ;
q[rear] = item;
cout << " is inserted\n\n" << item ;

}
```

5.9 حذف عنصر من الطابور

1. يتم فحص مؤشر البداية للطابور, اذا كان يؤشر على الفراغ (قيمة المؤشر تساوي (1-)), يتم استصدار رسالة على ان الطابور خالي من العناصر.
2. اذا كان يؤشر على احد العناصر يتم حذف العنصر الذي يؤشر علية مؤشر البداية, ويتم ذلك كمايلي:
   A. يغير موقع المؤشر بزيادتة بمقدار واحد, (ان مجرد تغيير قيمة المؤشر بزيادتة بواحد يعني ان العنصر تم حذفة).
3. اذا كان مؤشر البداية يساوي مؤشر النهاية (هذا يعني وجود عنصر واحد في الطابور فقط) يصفر المؤشران (تسند لهما القيمة (1-)), والذي يدل على ان الطابور اصبح فارغا.

❖ دالة حذف عنصر من طابور خطي, باستخدام المصفوفات.

```
void                    deletion()
{
   if (front = = 0 )
   {
     cout<< “\n Queue Underflow\n\n” ;

    }

   item=q[front];

if ( front = = rear )
   {
          front = 0 ;
          rear = 0 ;
   }

 else if ( front = n )
          front=1;
            else
               front = front + 1 ;
cout<< “\n%d is deleted\n\n” <<  item ;

}
```

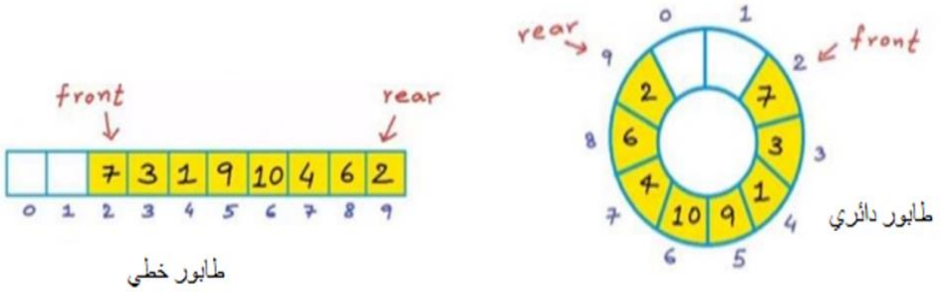
5.10 الطابور الدائري

دعنا نستخدم نفس خطوات خوارزمية الاضافة والحذف التي استخدمت في الطابور الخطي خلال هذا الفصل. بداية نبتدأ كلا النهايتين بالقيمة (1-) للدلالة على ان الطابور فارغ. فعندما يتم اضافة بيانات الى الطابور فان كلا النهايتين ستحصل على قيم جديدة. وعندما تضاف بيانات جديدة, فان النهاية الخلفية تزاد بمقدار واحد, وعندما تحذف بيانات فان النهاية الامامية تنقص بمقدار واحد. هذا العمل جيد ولكن يملك عيب اساسي. الان ماذا لو كانت القيمة العظمى لطاقة الطابور هي 5 عناصر. ونفرض ان المستخدم اضاف اربع عناصر ابتداءا, ثم حذف ثلاث عناصر واضاف اثنان اخران مرة اخرى. الطابور سوف لايسمح له باكمال الاضافة, حيث سيضيف عنصر واحد فقط ويقرر ان الطابور لايوجد به فراغ. السبب اننا نزيد ونقلل من نهايتين مختلفتين اعتمادا على الاضافة والحذف بشكل اعمى, ولاندرك ان كلا النهايتين متعلقتان ببعض. هنا سنخلص الى سؤال مهم هو لماذا تظهر رسالة بامتلاء الطابور في الوقت الذي يكون فيه الطابور في الحقيقة شبه فارغ. هذه المشكلة تقودنا الى ان نفكربطريقة جديدة تركز على البيانات اكثر من تركيزها على نهاية الاضافة او نهاية الحذف.

السؤال المهم هنا: المصفوفة لها حجم محدد, فاذا كان مؤشر الخلف في نهاية المصفوفة وكان هناك عنصر جديد يراد اضافته, فما هو الحل؟

هناك اثنان من الحلول:

- ✓ تزحيف كل العناصر الى الامام في المصفوفة (وهذه مكلفة جدا).
- ✓ الالتفاف حول المصفوفة وهذا يولد الطابور الدائري.



شكل 5.6: توضيح الطابور الدائري ومقارنته مع الطابور الخطي.

يتم تغيير قيم المؤشرات عند الاضافة والحذف باضافة واحد لهما.

rear = (rear + 1) mod n

front = (front + 1) mod n

5.10.1 مميزات الطابور الدائري

الطابور الدائري يسمح لكامل المصفوفة بخزن العناصر دون الحاجة الى تزحيف البيانات ضمن الطابور. المصفوفة بالطابور الدائري تلتف حول نفسها مما يجعل امكانية تجاور مؤشر النهاية مع مؤشر البداية (فضلا عن امكانية ان يكون مؤشر البداية اكبر من مؤشر النهاية), هذا يساعد على الوصول الى اول او اخر عنصر بشكل كفوء. بعد عملية الحذف فان المكان الفارغ في المصفوفة من الممكن ان نستفيد منه لخزن اي عنصر جديد.

في الطابور الخطي فان العناصر من الممكن ان تحشر او تضاف وذلك بزيادة مؤشر النهاية, وعندما يصل مؤشر النهاية الى الحجم الاكبر (نهاية المصفوفة) حتى وان تم حذف عناصر من مقدمة الطابور فان الطابور يعتبر مملوء. لذلك في الطابور الخطي لايمكن الاستفادة من المصفوفة بشكل كامل. من جانب اخر, في الطابور الدائري فان مؤشر البداية يتجاور مع مؤشر النهاية وعليه فان الطابور الدائري يسمح لكامل المصفوفة لخزن العناصر. الفائدة الاساسيه للطابور الدائري على الطابور الخطي هو امكانية استخدام فضاء المصفوفة بشكل اكفأ ولاتوجد ضياعات بالذاكرة.

5.10.2 خوارزمية اضافة عنصر الى الطابور الدائري

1. فحص مؤشر النهاية فيما اذا كان في موقع (1-), اي ان الطابور فارغ, عندها سيكون الاجراء هو:
   A. تغير قيمة المؤشرين وذلك باضافة واحد اليهما.

```
front = rear = 0
```

   B. اضافة البيانات في الموقع الجديد لمؤشر النهاية

```
Q[rear] = element
```

2. اذا لم تتحقق الفقرة 1, ابدأ بفحص مؤشر البداية والنهاية لمعرفة وضعية الطابور, فاذا كانت قيمة مؤشر البداية يساوي مؤشر النهاية + 1, فان هذا يعني ان الطابور الدائري مملوء, ولايوجد فراغ للاضافة, الاجراء سيكون:

```
front = (rear + 1) %  max
```

   A. يتم طباعة رسالة خطأ تشير الى امتلاء الطابور.
   B. الخروج من دالة الاضافة.
3. اذا لم يكن الطابور مملوء, ففي هذة الحالة يتم اضافة واحد الى مؤشر النهاية لتغيير موقعه.

```
rear = (rear+1) %  max
```

4. اضافة العنصر الجديد في الموقع الذي يؤشر عليه مؤشر النهاية.

```
Q[rear] = element
```

❖ دالة اضافة عنصر الى الطابور الدائري باستخدام المصفوفة

```
void insert (int data) {
        if (rear = = -1) { //    الطابور فارغ
        front = rear = 0;
```

```
        arr [front] = data;  // اضافة العنصر في بداية الطابور (اول عنصر في الطابور)
  }
  else     {
    int pos = (rear + 1) % SIZE;
    if (pos = = front) { //     الطابور مملوء
      cout << "No space in queue ..." << endl;
      return;
    }
    else {
      rear = pos; //   تحديث مؤشر النهاية
      arr[pos] = data; //   اضافة العنصر في الموقع المحدد
    }
  }     }
```

5.10.3 خوارزمية حذف عنصر من الطابور الدائري

1. بداية يتم فحص مؤشري البداية والنهاية فاذا كانت قيمة المؤشران تساوي (-1) فان ذلك يعني ان الطابور فارغ وبالتالي لايمكن حذف عنصر من طابور فارغ, في هذه الحالة:
   A. يتم طباعة رسالة خطأ للدلالة على ان الطابور فارغ.
   B. الخروج من دالة الحذف.
2. في خلاف ذلك, اي اذا كان الطابور ليس فارغ بمعنى ان العبارة في (1) خاطئة.. عندها يتم حذف العنصر باجراء ما يلي:
   A. من الممكن حفظ القيمة التي يتم حذفها اذا كان هناك ضرورة للاحتفاظ بها. عادة لاحاجة لحفظ العنصر المحذوف.

element = Q[front]

B. يتم تغيير قيمة المؤشر وذلك بانقاصها بواحد.. لاحظ هنا حالتين

a) اما ان يكون مؤشري البداية والنهاية متساويين (هذا يعني ان الطابور يحتوي على عنصر واحد فقط) عندها يتم مساواتهم للصفر او (-1).

front = rear = -1

b) اما اذا كان مؤشري البداية والنهاية غير متساويين (اي ان الطابور يحتوي على اكثر من عنصر واحد), عندها يضاف واحد الى مؤشر البداية كما في ادناه

front = (front + 1) % max

ملاحظة: في الخطوة (2) تم حجز العنصر المراد حذفة بمتغير وذلك لغرض طباعته لاحقا او اجراء اي عملية عليه, ويمكن اهمال هذه الخطوة اذا لم تكن قيمة العنصر المحذوف مهمة.

ملاحظة: ربما يسأل القاريء لماذا استخدام باقي القسمة في الخطوات التي تتطلب الاضافة الى المؤشرات كما ورد في خوارزميتي الاضافة والحذف. والجواب هنا ان (max) تعني حجم الطابور الدائري وبما اننا نستخدم طابور دائري فهناك احتمال امتلاء الطابور وحذف عناصر منه وبالتالي يتغير موقع مؤشر النهاية والبداية, فاذا امتلأ الطابور يكون مؤشر النهاية على اخر موقع في الطابور (لنتخيل الطابور ممتلأ ولكن عدد من العناصر تم حذفها هنا سيكون فراغ في بداية الطابور) فاذا اردنا اضافة جديدة فان المؤشر ستزاد قيمته بواحد وبالتالي ستكون قيمته اكبر من القيمة العليا لحجم الطابور لذلك نعمل باقي القسمة لتكون قيمة مؤشر النهاية (1) او اي قيمة اخرى ضمن مدى حجم الطابور, ونفس الشيء ينطبق على مؤشر البداية.

ملاحظة: مؤشر البداية يستخدم للحذف ومؤشر النهاية يستخدم للاضافة.

❖ دالة حذف عنصر من الطابور الدائري باستخدام المصفوفة

```
void delete ( ) {
  if (front = = -1) { //  الطابور فارغ
    cout << "Queue is empty ... " << endl;
    return;
  }
  else {
    if (front == rear) { //  فقط عنصر واحد موجود في الطابور
      front = rear = -1;
    }
    else {
      front = (front + 1) % SIZE; //  حدث مؤشر البداية بتزحيفه موقع واحد
    }
  }    }
```

❖ دالة برمجية لعرض محتويات الطابور الدائري.

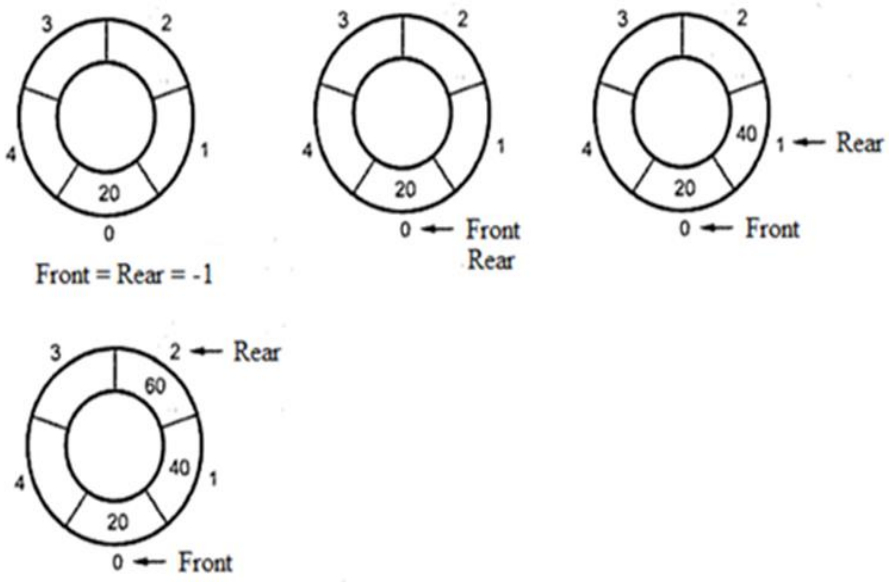
```
    void display () {
      int i;
      cout << "front : " << front << "   rear : " << rear << endl;
      cout << "Current circular Queue Elements ( front to rear ) : " <<
endl;
      if (front = = -1) {
```

```
  cout << "Queue is empty ... " << endl;
  return;
}
else {
  i = front;
  do {
    cout << arr [i] << ” ” ;
    i = (i + 1) % SIZE ;
  } while (i != rear) ;
  cout << arr[rear] ;
}
cout << endl;
  }
```

5.10.4 تنفيذ عمليات الاضافة والحذف في الطابور الدائري

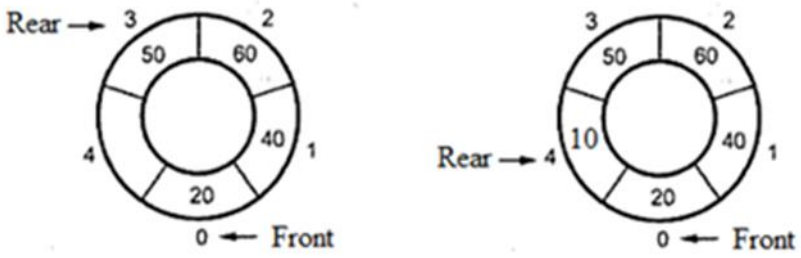
لنفرض ان لدينا طابور دائري فارغ كما في الشكل 5.7, حجمه هو 5 ولنرى كيفية الاضافة والحذف عليه.

بدءا فان الطابور خالي (اي ان مؤشر البداية والنهاية يساوي (1-)). الايعاز الاول سيكون باضافة العناصر التالية 20, 40, 60. سنتابع عملية الاضافة والحذف من خلال الاشكال الرسومية في الشكل 5.7 والتي سنوضح فيها المؤشرات ومواقعها:



شكل 5.7: يوضح الطابور الفارغ, ثم وضع الطابور بعد اضافة ثلاث قيم بالتعاقب.

الايعاز الثاني هو اضافة القيم (10 ,50) بالتعاقب الى الطابور, هذا سيتم وحسب ماموضح في الشكل 5.8.



شكل 5.8: وضع الطابور بعد اضافة عنصرين

الان لو جاء ايعاز باضافة العنصر 30 الى الطابور الدائري.

لاحظ هنا بالرجوع لخوارزمية الاضافة فاننا نفحص اذا كان الطابور مملوء ام لا, وذلك اذا تحققت العلاقة التالية

front= 0 = (Rear + 1) % size

front = (4+1) % 5 = 0

ونظرا لتحقق العلاقة, لذلك لايمكن الاضافة لان الطابور ممتليء.

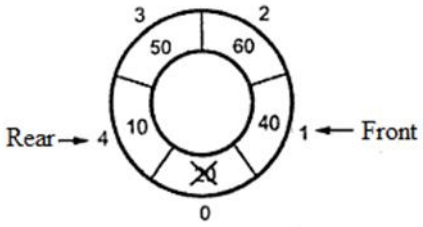
الان لو جاء الايعاز لنحذف عنصر. اولا نتاكد من وجود عناصر (اي ان الطابور ليس خاليا) ويكون الطابور فارغا اذا تساوى مؤشر البداية والنهاية وكانت قيمتهم (1-). عند الفحص نجد ان المؤشرين كل واحد يؤشر الى قيمة مختلفة.

front = 0 and rear = 4

هنا الطابور ليس فارغ لذلك بالامكان الحذف وذلك بتغيير قيمة مؤشر البداية.

front = (front + 1) % size = (0 + 1) % 5 = 1 % 5 = 1

لاحظ الشكل 5.9.

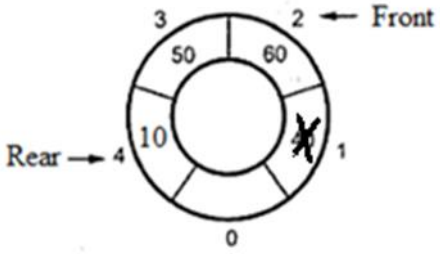


شكل 5.9: الطابور بعد حذف القيمة 20.

لنحذف عنصر اخر, نتبع نفس الطريقة اعلاه ونتاكد من ان الطابور ليس فارغ, بعد ان تاكدنا من ان الطابور ليس فارغ نقوم بعملية الحذف بتغيير قيمة مؤشر البداية.

front = (front + 1) % size = (1 +1) % 5 = 2 % 5 = 2

لاحظ الشكل 5.10.



شكل 5.10: الطابور بعد اجراء الحذف لعنصر ثان.

الفصل السادس
القوائم الموصولة LINKED LISTS

6.1 المقدمة

اذا قمت بخلق قائمة (مجموعة من القيم التي لها نفس النوع مثل قائمة او مجموعة من القيم من نوع الاعداد الصحيحة لغرض تمثيل ارقام هواتف مثلا, قائمة او مجموعة من نوع السلاسل الحرفية لحفظ اسماء الطلبة وهكذا) اعتمادا على المصفوفات فانك من الممكن ان تبدأ الاعلان عن المتغيرات اعضاء المصفوفة والتي ستحمل العناصر (البيانات), وكل عنصر من الممكن الوصول الية بواسطة رقم الفهرس المقابل له. عندما تعمل ذلك, وغالبا اثناء العمل مع المصفوفات فانك يجب ان تحدد عدد العناصر الاكبر التي تتوقع انها ستمثل بواسطة هذه المصفوفة. بدون تخطيط جيد فان البعد او الابعاد المحددة للمصفوفة من الممكن ان تكون كبيرة جدا او صغيرة جدا وهذا من الممكن ان يؤدي اما الى ضياع مساحة من الذاكرة تحجز للمصفوفة دون الاستفادة منها او ان يكون هناك حاجة الى مزيد من مساحة الذاكرة لم يتم حجزها مسبقا وبالتالي فان كلتا الحالتين ستؤديان الى قلة في كفاءة البرامج التي تستخدم المصفوفات, علما بان C++ لا تسمح بالإعلان عن مصفوفة دون تحديد الابعاد قبل ان تبدا المصفوفة. هذا يعني, عند خلق مصفوفة فانك غالبا يجب عليك ان تحدد العدد الاكبر للعناصر التي ستتضمنها المصفوفة. ملاحظة اخرى بهذا الشأن هو ان المصفوفة تخزن بمواقع متجاورة وعندما يكون حجمها كبير احيانا تبرز مشكلة عدم توفر المساحة الخزنية الكافية للمصفوفة.

القوائم الموصولة هي قوائم من الممكن ان تتوسع او تتقلص حسب رغبة المستخدم. هذا يعني عند خلق قائمة, فانك لا تحتاج الى ان تخمن او تحدد العدد الاكبر لعناصر القائمة والتي من الممكن اضافتها اثناء العمل (التنفيذ). لاستخدام هذه المرونة فان العناصر يجب ان تدار بواسطة طريقة اخرى تختلف عن طريقة ادارة المصفوفات وذلك بالاستعانة بما يسمى بالمؤشرات. مع القوائم الموصولة من السهولة اضافة وحذف عناصر في أي مكان من القائمة.

القوائم الموصولة والمصفوفات متشابهة, حيث ان كلاهما تخزن مجموعة من البيانات. كلاهما المصفوفات والقوائم الموصولة تخزن عناصر لمصلحة شفرة

(برنامج) المستخدم. نوع البيانات ليس مهم وذلك لان نفس التركيب يعمل لخزن عناصر لأي نوع من انواع البيانات. واحدة من طرق التفكير حول القوائم الموصولة هو النظر الى كيفية عمل المصفوفات والتفكير بطرق مختلفة.

ان الفرق الرئيس بين استخدام المصفوفات والقوائم الموصولة يتمثل بميزة القوائم الموصولة على اضافة عناصر جديدة وحذف عناصر من القائمة بسهولة وسرعة وهذه الميزة تعتبر صعبة في المصفوفات, كذلك فان القوائم تمتاز بانها مرنه بحجم الذاكرة التي تستخدمها حيث يمكن التمدد والتقلص بحجم الذاكرة حسب الحاجة اثناء التنفيذ في الوقت الذي لا تتوفر هذه الميزة في المصفوفات التي تمتاز بثبات الحجم وصعوبة تغييره اثناء التنفيذ, ولكن المصفوفات تمتاز بسرعة الوصول الى عناصرها (وصول مباشر) على عكس القوائم التي لا تصل الى عناصرها بشكل مباشر, وبالتالي بطأ الوصول الى عناصر القائمة مقارنة بالمصفوفات.

6.2 ابتداء القوائم الموصولة

عندما ننشأ قائمة موصولة ستكون فارغة بداية او على الاقل يجب ان تفترض ذلك قبل ان يتم اضافة اي عنصر اليها. لتحديث ذلك, فان عليك الاعلان عن متغير يعتبر العنصر الاول في القائمة ونواة بناء القائمة. بالرغم من انك يمكنك ان تدعوه باي اسم, لكن المتعارف علية ان يسمى الراس او البداية head او start.
هذا العنصر من الممكن ان يكون خاص (في حالة استخدام الصنوف classes) اذا لم ترغب ان يتم الوصول الية من خارج الصنف. اما اذا اردت ان يتم الوصول الية من قبل مستخدمي الصنف فيمكن ان تجعله عام. بالرغم من ان هذا العنصر (المتغير) يعلن عنه كمؤشر ويؤشر على بداية القائمة, يجب ان لا تخصص ذاكرة له في دالة الانشاء constructor, هذه الذاكرة تنظم عند الوصول لها. لذلك ببساطة من الممكن ابتدائها على ان تؤشر الى null.

كما وضحنا سابقا فان كل عنصر من عناصر المصفوفة يمكن الوصول له باستخدام الفهرسة (رقم الموقع) وهذا يضاف الى اسم المصفوفة التي عادة تحمل عنوان العنصر الاول بالمصفوفة. ونظرا الى ان عناصر القائمة الموصولة غير مفهرسة ولا تحتوي ارقام مواقع لذلك فعلينا ان نجد الية اخرى لغرض الوصول الى عناصر القائمة الموصولة. ان اسهل طريقة لعمل ذلك هو اولا ايجاد العنصر الاول في القائمة وبعدها يتم استخراج عنوان العنصر اللاحق له. اما اذا كانت القائمة فارغة فان هذه المعلومات ستوفر لنا null (وهي تعني لاشيء). اذا كان العنصر الاول الذي وصلت له هو اخر عنصر في القائمة, فانه ايضا سيوفر لنا لاشيء null. اي عنصر اخر في الوسط من الممكن ان يوفر لك وصول الى العنصر اللاحق في القائمة. لدعم هذه النظرية فعليك ايجاد مؤشر الى العنصر اللاحق في القائمة كمتغير عضو الى الصنف الذي يحمل العناصر. هذا المتغير العضو يمكن ان يدعى next ولكن يمكن تسميته باي اسم اخر.

6.3 ماهي القائمة الموصولة

القائمة الموصولة هي هيكل بياني يستخدم لتنظيم البيانات في الذاكرة. وكما هو واضح من التسمية فالقائمة الموصولة هي قائمة من العناصر كل عنصر يسمى (عقدة) (node) يرتبط (او يتصل) مع العناصر الاخرى باستخدام المؤشرات. والعقدة هي تركيب structure ضمن القائمة يحتوي على حقلين او اكثر (الحقول تمثل قيود في التركيب), غالبا يكون هناك حقل يسمى حقل (البيانات او المعلومات) (data or information) وهو يحتوي على البيانات او المعلومات المراد خزنها والتي يتم الاعلان عن نوعها بشكل مسبق, وحقل او حقلين لربط او ايصال العقدة بالعقد الاخرى تحتوي على عنوان ذاكرة (link or address). عليه فان القائمة الموصولة هي سلسلة من التراكيب (structures).

والقوائم الموصولة من الممكن ان تكون من اي نوع من الانواع التالية وفقا لعلاقة العقد او عناصر القائمة فيما بينها:

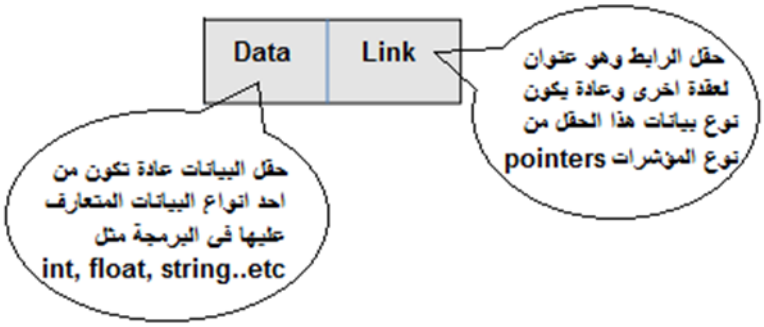
A. قوائم موصولة احادية Singly-linked Lists
B. قوائم موصولة ثنائية او قوائم موصولة بطريقين Doubly-Linked or Two Way Linked Lists
C. قوائم موصولة دائرية Circularly-Linked Lists
D. قوائم موصولة دائرية ثنائية Circularly-Doubly Linked Lists

الرجاء ملاحظة ان هناك فوارق تميز بين القوائم الموصولة الاحادية والقوائم الموصولة الدائرية والثنائية. ووفقا لهذا التعريف فان قيود التركيب ستحمل اي نوع من البيانات نحتاج اليه في العمل, فقط الانتباه ان كل قيد يجب ان يكون حالة من التركيب (structure). هذا يعني عدم امكانية ان يكون هناك تركيب بياناته حروف او رموز يؤشر الى تركيب اخر يحمل قيم من نوع short, مصفوفة حروف, او long على سبيل المثال.

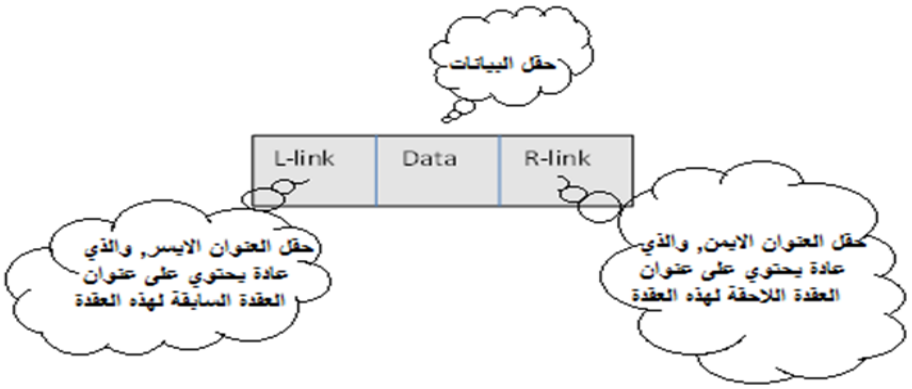
ان جميع الحالات للعقد يجب ان تكون من نفس التركيب للقائمة الواحدة. مفهوم اخر لطيف هو ان كل تركيب او عقدة من الممكن ان يتواجد في اي مكان في الذاكرة, والعقد عادة لا تكون خطية بالذاكرة (اي لا تكون بمواقع متسلسلة بالذاكرة) وانما تكون في مواقع مبعثرة داخل الذاكرة.

6.4 العقدة Node

العقدة هي بحد ذاتها موقع او مجموعة مواقع متجاورة في الذاكرة لها عنوان هو عنوان موقعها في الذاكرة (عنوان اول بايت (كلمة) في الذاكرة). ومن الممكن ان نخزن بها (العقدة) بيانات وممكن ان نخزن بها عنوان ذاكرة لموقع اخر. ولكي نضيف قيم الى هذا الموقع/المواقع سواء كانت هذه القيم بيانات او عناوين فلا بد ان نعلن عن نوع البيانات التي سيتم اضافتها لحقول العقدة (هذه العملية تفيد بتحديد حجم الذاكرة للعقدة). ان الاعلان عن نوع البيانات يتم من خلال استخدام التراكيب (structure) والتي يعرفها ويحددها المستخدم.



شكل 6.1: عقدة احادية



شكل 6.2: عقدة ثنائية

6.5 لماذا القوائم الموصولة

من فوائد استخدام القوائم هو عدم القلق حول فقدان مساحة من الذاكرة من خلال تخصيص ذاكرة اكثر من المطلوب. كلما زاد الطلب على البيانات يتم تخصيص ذاكرة اضافية وفقا لذلك. لكن الجانب الصعب هو عند الرغبة للوصول الى اي عقدة يجب علينا المرور على كل عقدة سابقة لها لحين الوصول الى العقدة

المطلوبة. هذا هو السبب لان تكون القوائم الموصولة لها اشكال مختلفة لتسهيل الوصول, مثل القوائم الموصولة الثنائية والدائرية.

القوائم الدائرية هي القوائم التي تكون فيها اخر عقدة دائما تؤشر الى العقدة الاولى. بينما الثنائية تحتوي على مؤشرين احدهما يشير الى العقدة اللاحقة بينما الثاني يؤشر الى العقدة السابقة.

6.6 المؤشرات

سنتطرق بمراجعة سريعة لبعض المصطلحات والقواعد للمؤشرات pointers والتي تعتبر الاساس في كتابة وتنفيذ برامج القوائم الموصولة. ان فهم المؤشرات يساعد بدرجة كبيرة على فهم القوائم الموصولة والعمليات التي تجرى عليها.

المؤشر هو متغير يخزن عنوان موقع ذاكرة وبالتالي فان هذا المتغير الذي هو من نوع مؤشر سيتعامل مع عناوين الذاكرة بدل ان يتعامل مع القيم او البيانات التي تحتويها مواقع الذاكرة كما في المتغيرات الاعتيادية. الموقع او العنوان الذي يشير له المتغير من نوع المؤشر بعض الاحيان يعرف باسم المؤشر عليه. احيانا فان المؤشرات ربما تكون مجموعة من القيم الخالية null والتي تفسر على انها لا تؤشر حاليا الى مؤشر عليه (في لغة C++ فان القيمة الخالية يمكن ان تستخدم القيمة المطلقة false).

لقد تعلمنا من خلال البرامج التي نكتبها بان المتغيرات ينظر لها على انها خلايا متجاورة في الذاكرة والتي من الممكن الوصول اليها من خلال المعرف الذي يمثل المتغير, وفي هذه الطريقة فان المستخدم لا يكترث او لا يهتم حول المواقع المادية للبيانات في الذاكرة, وببساطة فانه يستخدم هذه المعرفات في أي وقت يرغب فيه الاشارة الى متغيراته والعمل عليها.

ان ذاكرة الحاسبة من الممكن ان نتخيلها كمجموعة متعاقبة من خلايا الذاكرة او مواقع الخزن المتجاورة, وحجم كل خلية تخصص للمتغير هو بقدر حجم النوع

المعلن عنه لهذا المتغير, وترقم خلايا الذاكرة بشكل تسلسلي واضح. هذه الطريقة تساعد على الوصول الى أي موقع في الذاكرة وذلك باستخدام هذه الارقام التسلسلية, وهي ارقام وحيدة لكل خلية في الذاكرة, بمعنى انها لا تتكرر (تسمى هذه الارقام عناوين (addresses)). في كل مرة يتم الاعلان عن متغير ما فان حجم الذاكرة التي يحتاجها هذا المتغير ستسند او تخصص له في موقع محدد من الذاكرة (هذا الموقع له رقم يمثل عنوانه في الذاكرة) وبشكل عام فان المستخدم لا يحدد الموقع للمتغير ضمن مستوى خلايا الذاكرة. ولحسن الحظ فان هذه العملية تنجز اليا بواسطة نظام التشغيل وخلال وقت التشغيل, وعلى كل حال ففي بعض الحالات ربما يكون المستخدم راغبا بمعرفة العنوان الذي تم خزن المتغير به خلال وقت التشغيل وذلك لكي يعمل على/ او ينفذ اوامر نسبة لموقعها في الذاكرة ولكن لا يكون ذلك متاحا عادة.

بعد ان يتم الاعلان عن المؤشر المحلي وقبل ان يتم اسناد قيمة له, فان المؤشر يحتوي على قيمة غير معروفة. المؤشرات العامة تبدأ اليا بالقيمة (null) (والتي تعني الفراغ او لا شيء او ربما تفسر على انها صفر). اتفاقية مهمة هي: ان المؤشر الذي لا يؤشر حاليا الى موقع ذاكرة محدد فيجب اسناد القيمة (null) له, حيث ان أي مؤشر يحتوي على القيمة (null) فهذا يعني ان المؤشر يؤشر على لا شيء. وعلى كل حال, وبسبب ان المؤشر له القيمة (null) فان ذلك كافي لجعله غير امن. ان لغة C++ لا ترغم المؤشر لان تكون له القيمة null. المؤشرات الخالية من الممكن استخدامها لتحديد نهاية القائمة الموصولة او العقدة التي ليس لها عقدة لاحقة.

6.6.1 عوامل المؤشرات

اكثر معاملين مهمين يستخدمان مع المؤشرات هما:

عامل العنوان (&) (اذا وضع امام متغير فانه سيعيد عنوان هذا المتغير في الذاكرة).

عامل التأشير (*) (اذا وضع امام متغير فان هذا المتغير سيشير الى موقع في الذاكرة يخزن به عنوان ذاكرة).

يجب ملاحظة ان متغيرات المؤشر يجب الاعلان عن نوعها مسبقا في البرنامج ونقصد بالإعلان هو تحديد عدد المواقع التي تؤشر عليها هذه المتغيرات وهي ذات الطريقة التي كنا نتعامل بها مع المتغيرات الاعتيادية, وسبب ان نحدد نوع متغير المؤشر هو لكي يتم تحديد حجم خلايا الذاكرة التي سيتم تحديدها لعمل هذا المتغير وبالتالي سهولة اسناد واسترجاع القيم منها وفقا لنوعها.

مثال

اذا كان لدينا الاعلان التالي

```
int i, *pi;
```

في هذا الاعلان, فان i هو متغير من نوع الاعداد الصحيحة (اي انه يشير الى بيانات في الذاكرة يحتوي على بيانات من نوع الاعداد الصحيحة).

بينما pi هو مؤشر الى موقع يحتوي على الاعداد الصحيحة (اي ان pi يشير الى موقع في الذاكرة (عنوان في الذاكرة) هذا العنوان اذا وصلنا له من خلال البرنامج سنجده يحتوي على بيانات من نوع الاعداد الصحيحة). اما اذا قلنا

```
pi = &i;
```

هنا ان &i تعيد عنوان المتغير i وتسند قيمته الى المتغير pi . اما اذا اردنا ان نسند قيمة الى المتغير i فيمكننا القول:

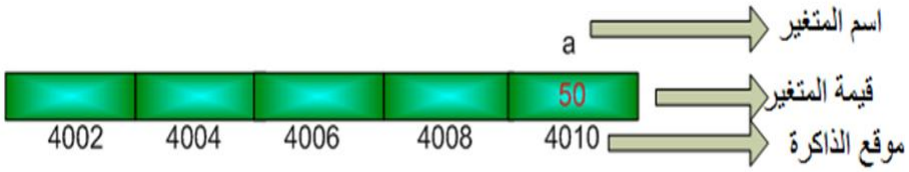
```
i = 10;    or       *pi = 10;
```

من ذلك نستنتج ان العلامة (&) اذا وضعت امام اي متغير اثناء تنفيذ البرنامج فانها ستعيد عنوان هذا المتغير في الذاكرة وليس قيمته. اما المتغير من نوع المؤشر الذي يعلن عنه كمؤشر بوضع العلامة (*) امام المتغير فانه سيحمل دائما عنوان في الذاكرة ولا يمكن ان يحمل بيانات. بينما المتغيرات الاعتيادية التي لا يوضع امامها اي من العلامتين اعلاه فأنها ستشير الى قيم او بيانات في موقع محدد في الذاكرة. لاحظ عندما نضع (*) امام متغير من نوع مؤشر فان التعامل معه سيكون مشابه

للتعامل مع المتغيرات الاعتيادية اي بالإمكان ان نسند قيمة له وتوضع بذلك الموقع كما في المثال اعلاه.

دعنا نتخيل الذاكرة على انها مصفوفة كبيرة وكل موقع في المصفوفة له عنوان ذاكرة مختلف. فاذا تم الاعلان عن متغير فان اسم المتغير سيشير الى موقع في الذاكرة يحتوي القيمة المسندة لهذا المتغير كما في الشكل 6.3.

```
int a = 50 ;
```



شكل 6.3: مواقع الذاكرة

وللتوضيح, نفرض ان اي موقع في الذاكرة من الممكن تمثيله مثل الدار الذي له عنوان ضمن المنطقة او الحي, وهذا الدار يحتوي على غرفة او اكثر (وهي تمثل النوع (افتراضا), لنفرض الغرفة هي بايت واحد وبالتالي غرفة واحدة يعني بايت واحد وهي من نوع character, وغرفتين كما لو نقول من نوع الاعداد الصحيحة وهكذا). لذلك فان الوصول الى عنوان الدار يكون من خلال اسم الدار مثلا (a), اما الشخص او الاشخاص الذين يسكنون هذا الدار فهو/هم يمثل (محتوى الدار او محتوى موقع الذاكرة), في الشكل (6.3) القيمة هي (50) ورقم الدار (حسب تسلسله في المنطقة) هو موقع الذاكرة (4010). اذا اردنا تغيير قيمة المتغير (اسم الشخص) في هذا الدار, فان الطريقة المناسبة هو ان نغير الشخص الذي يسكن هذا الدار وبالتالي نقول ان الدار يحتوي الشخص الذي سكن حديثا في الدار.. هذا من الممكن برمجيا باستخدام الايعاز التالي:

```
a = 100 ;
```

وهذا يعني اننا اخرجنا 50 من الدار ووضعنا بدلا عنه 100, دائما ان عملية المساواة تنسخ القيمة القديمة وتبدلها بالقيمة الجديدة.

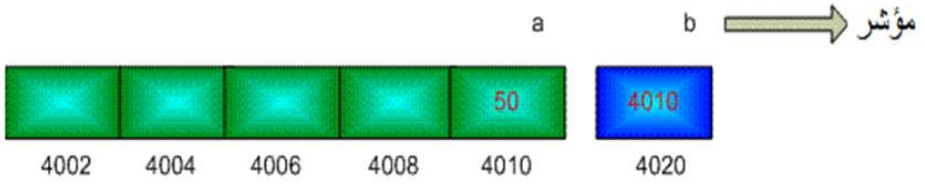
لكن لاحظ ان استخدام المؤشر يساعد على الذهاب مباشرة الى موقع الذاكرة للعنصر (a) (اي الى رقم الدار او حسب المصطلحات البرمجية موقعه في الذاكرة), ومن الممكن تغيير قيمة المتغير في ذلك الموقع من دون الحاجة الى العنصر (a) . مثال

```
int  * b ;
```

هنا تم الاعلان عن مؤشر باسم b من نوع الاعداد الصحيحة

```
b = & a;
```

بهذا فاننا مررنا عنوان العنصر (a) الى المتغير b



شكل 6.4: شكل يوضح خزن العنوان في المؤشر

الان من الممكن تغيير قيمة a دون الوصول الى a (لاحظ هنا ان المتغير b هو موقع في الذاكرة لكنه بدل ان يخزن بيانات فانه يخزن عنوان ذاكرة, وهو عنوان المتغير a).

```
*b = 100 ;
cout << a ;
```

عندما تصدر ايعاز الى الحاسوب للوصول الى b * فانه يقرا المحتويات داخل b والتي هي بالحقيقة عنوان a وبعدها يتم اتباع العنوان للوصول الى (a) او عنوان a وتتم تغيير محتويات a لتكون القيمة 100.

السؤال الان هل من الممكن قراءة وتغيير محتويات a دون الوصول الى b ؟
الجواب نعم ممكن اذا عملنا مؤشر الى مؤشر. لاحظ المثال التالي:

```
int main( )
{
    int a = 50;       // ابتداء قيمة المتغير a
    cout<<"The value of 'a': "<<a<<endl;
    int * b;          // الاعلان عن مؤشر b
    b = &a;           //  نقل عنوان  'a' الى المؤشر
    *b = 100;
     cout<<"The value of 'a' using *b: "<<a<<endl;
     int **c;         // الاعلان عن مؤشر الى المؤشر 'c'
    c = &b;
    **c = 200;        // تغيير قيمة 'a'  باستخدام مؤشر الى مؤشر
 cout<<"The value of 'a' using **c: "<<a<<endl;
    return 0;
}
```

ملاحظة: في كثير من البرامج نرى الايعاز q=q->link داخل البرنامج. هذه العبارة فقط تزحف المؤشر من عقدة الى عقدة اخرى (أي من موقع في الذاكرة الى موقع اخر).

6.7 هيكلية القائمة الموصولة

كما سبق واشرنا بان القائمة الموصولة هي عبارة عن مجموعة من العقد وبالتالي فان العقدة هي الوحدة الاساسية لبناء القائمة الموصولة, والعقدة الواحدة تمثل مجموعة من خلايا الذاكرة المتجاورة. ولذلك فان بداية العمل سيكون الاعلان عن محتويات العقدة. ان العقدة لأغراض القوائم الموصولة يعلن عنها من نوع التركيب (structure) ولذلك ستكون نقطة البداية هي ان يتم خلق تركيب باسم node (او اي اسم اخر) وهذا التركيب له على الاقل اثنان من الاعضاء او الحقول, واحد يمثل البيانات (data) وهو من نوع الاعداد الصحيحة في هذا المثال (كل انواع البيانات من الممكن استخدامها) والذي سيتم خزن المعلومات فيه, بينما الثاني هو مؤشر باسم (next) (او اي اسم اخر) عادة يؤشر الى العقدة اللاحقة (node *next) (سيحمل عنوان العقدة اللاحقة). بعد ان يتم الاعلان عن التركيب للعقدة فسيكون من الممكن ان تضاف المعلومات والبيانات الى حقول هذا التركيب.

```
struct   node
{
int       data;
node  * next;
};
```

- لاحظ هنا ان المؤشر اعلن عنه من نوع node اي من نوع التركيب وهذا يعني بانه سيؤشر الى التركيب وليس الى حقل محدد من التركيب.

ملاحظة: عند العمل على انشاء قوائم موصولة فأننا سنحتاج الى خلق عقد جديدة باستمرار وكلما دعت الحاجة, هذا من الممكن ان يتم من خلال الايعاز (new).

مثال

```
node  *Q ;

Q = new node;
```

ملاحظة: عند الانتهاء من استخدام عقدة معينة فانه يفضل حذفها واعادتها الى الذاكرة الحرة لكي يتم الاستفادة منها لاحقا, هذا يتم باستخدام الايعاز (delete).
مثال

```
delete Q ;
```

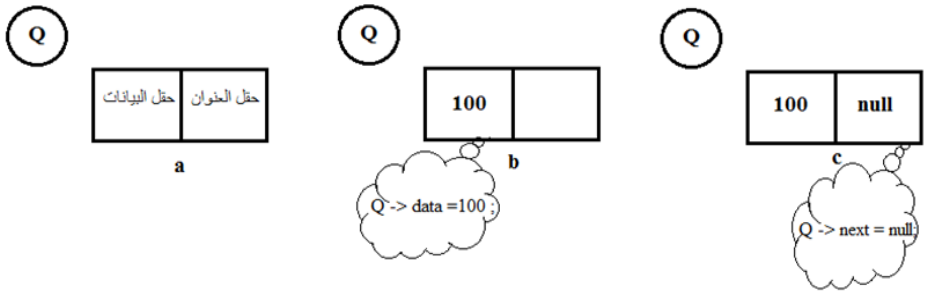
- عملية اضافة بيانات الى حقل البيانات في العقدة التي يتم خلقها تتم باستخدام الايعاز التالي

```
Q - > data = 100 ;
```

هنا تمت اضافة بيانات وهي القيمة 100 الى حقل البيانات في العقدة (Q)

- اضافة عنوان الى حقل العنوان في العقدة التي يتم خلقها, يتم من خلال الايعاز التالي مع ملاحظة ان الجانب الايمن من المساواة اما يكون عنوان عقدة اخرى او يكون (null) في حالة كون هذه العقدة التي خلقت هي العقدة الاخيرة في القائمة.

```
Q -> next = null ;
```



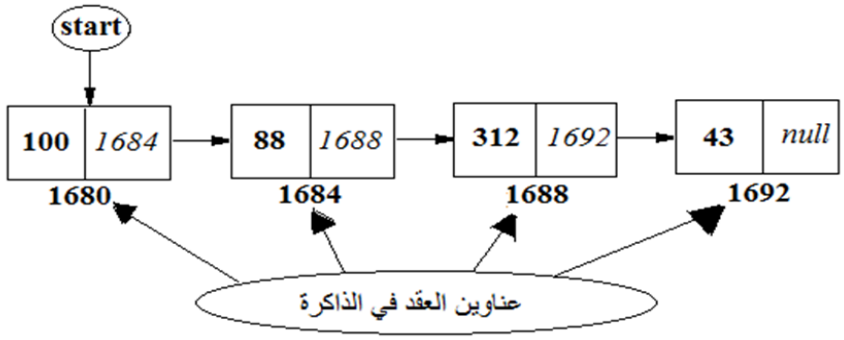
شكل 6.5: الشكل يوضح خلق عقدة جديدة واضافة بيانات لها.

6.8 المرور على عناصر القائمة الموصولة

لغرض المرور على كامل عناصر القائمة الموصولة لعرض محتوياتها مثلا.

نبدأ بخلق متغير من نوع مؤشر باسم معين مثلا (temp) وهذا المؤشر سيسند له عنوان عقدة الراس (العقدة الاولى), عادة عنوان العقدة الاولى يكون متوفر للمستخدم لكي يستخدم للوصول الى القائمة (من الممكن ان يكون عنوان العقدة الاخيرة ايضا متوفر ويعتمد ذلك على طريقة العمل والمعالجة). يمرر المؤشر الى العقدة اللاحقة وهكذا لحين الوصول الى العقدة النهائية.

لتوضيح كيفية الوصول الى العقدة اللاحقة, نفرض لدينا القائمة التالية والمكونة من اربع عقد كما في الشكل 6.6.



شكل 6.6: قائمة موصولة موضح بها القيم وعناوين العقد في الذاكرة

لاحظ عنوان العقدة الثاني موجود في حقل العنوان للعقدة الاولى, وعنوان العقدة الثالثة موجود في حقل العنوان للعقدة الثانية.. وهكذا لبقية العقد حيث يكون عنوان كل عقدة موجود في حقل عنوان العقدة السابقة.

الوصول الى العقدة اللاحقة يكون من خلال قراءة عنوانها الموجود في حقل العنوان للعقدة التي تسبقها.

فمثلا للوصول الى العقدة الثانية, يجب اولا الوصول الى العقدة الاولى من خلال عنوانها المحفوظ بالمتغير start ثم قراءة عنوان العقدة الثانية المحفوظ في حقل العنوان للعقدة الاولى باستخدام الايعاز التالي:

```
start - > next ;
```

لو اردنا ان نصل الى العقدة الثالثة مثلا نضع مؤشر على العقدة الاولى (temp) ونحركه للثالثة. ايضا من الممكن الوصول للعقدة الثالثة باستخدام العناوين في العقدة الاولى والثانية, وذلك لان عنوان العقدة الثانية موجود بالأولى وعنوان الثالثة بالثانية فيكون الايعاز التالي مفيد

```
Start -> next -> next ;
```

لاحظ هنا اولا سنصل من خلال الايعاز (start -> next) الى عنوان العقدة الثانية وهذا الايعاز يكافئ العنوان 1684(لأن حقل عنوان العقدة الثانية موجود في حقل العنوان next للعقدة الاولى start) ولذلك فان الايعاز (start -> next -> next) من الممكن ان نتخيله ليصبح (next <-1684) وهذا يعني انه يشير الى عنوان العقدة اللاحقة في العقدة 1684 (سيتم قراءة العنوان 1688 وبهذا نكون قد وصلنا الى العقدة الثالثة). الان حاول تفسير الايعاز (start -> next -> next -> next).

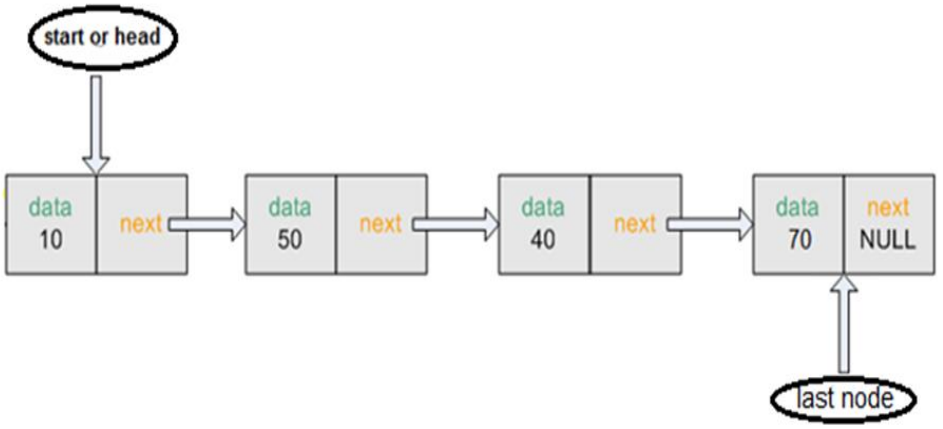
لغرض قراءة جميع بيانات القائمة الموصولة (وهي نفس فكرة الوصول لجميع العقد في القائمة) نلاحظ الكود التالي:

```
temp = start ;
while ( temp!=null )
{
cout <<  temp -> data  <<" " ;   // temp عرض بيانات العقدة المؤشر عليها بالمؤشر
```

```
temp = temp-> next ;   // نقل المؤشر (temp) الى العقدة اللاحقة
}
```

6.9 القوائم الموصولة الاحادية Singly Linked Lists

القوائم الموصولة الاحادية هي قوائم تكون فيها العقدة الواحدة متكونة من حقلين احدهم للبيانات والثاني حقل عنوان لمؤشر يؤشر على العقدة اللاحقة, واخر عقدة في القائمة تؤشر الى الفراغ (null). عادة فان عنوان العقدة الاولى يكون محفوظ لدينا ويتم الوصول للقائمة عن طريقه, ومن الممكن ان يتم حفظ عنوان العقدة الاخيرة ايضا (الشائع استخدام عنوان العقدة الاولى فقط, وهذا سنتبعه هنا).



شكل 6.7: قائمة موصولة احادية

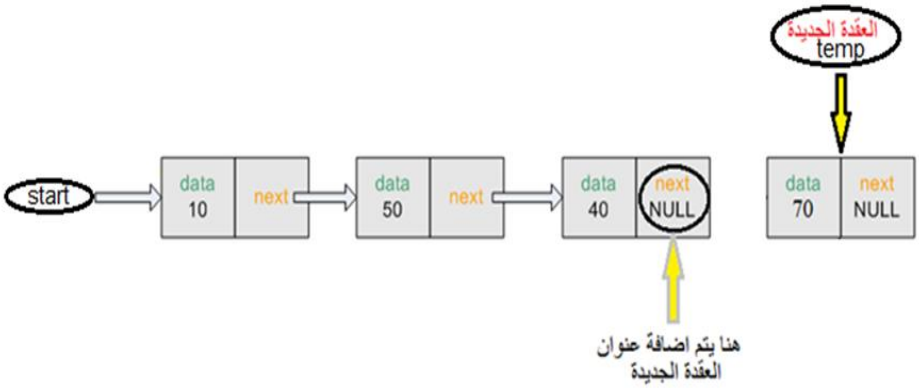
بعض العمليات الاساسية التي من الممكن انجازها على القوائم الموصولة الاحادية هي:

A. اضافة عقدة في نهاية القائمة الموصولة.
B. اضافة عقدة الى بداية القائمة الموصولة.

C. حشر عقدة في موقع ما داخل القائمة الموصولة.
D. حذف عقدة من القائمة الموصولة.
E. ترتيب عناصر القائمة الموصولة.
F. تحديث بيانات عقدة ضمن القائمة الموصولة.

A. اضافة عقدة جديدة الى نهاية القائمة الموصولة

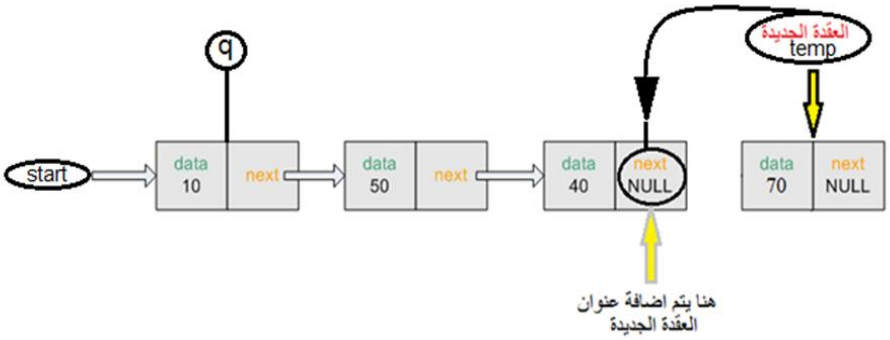
بعض الاحيان تبرز الحاجة الى اضافة عقدة جديدة في نهاية القائمة الموصولة. ولغرض الاضافة في نهاية القائمة فلابد من الوصول الى العقدة الاخيرة في القائمة ومن ثم الاضافة بعدها (العقدة الاخيرة في القوائم الاحادية يتم معرفتها وذلك بان يكون حقل العنوان في العقدة يحمل القيمة **(null)**. وحيث اننا لا نملك معلومات عن القائمة سوى معلومات عن راس القائمة او عنوان العقدة الاولى, لذلك علينا ان نبدأ من العقدة الاولى ويتم المرور على جميع العقد واحدة تلو الاخرى لحين الوصول الى العقدة الاخيرة ومن بعد ذلك تتم الاضافة بعد هذه العقدة (العقدة الاخيرة). في هذه الحالة فان من المهم ان نتبع الخطوات التالية:



شكل 6.8: شكل توضيحي لإضافة عقدة في نهاية القائمة الاحادية.

1. دائما يتم فحص القائمة للتأكد من انها ليست فارغة. نخلق عقدة جديدة (لنسميها **temp**) ثم نضيف المعلومات المطلوبة الى هذه العقدة, وهي قسمين الاول قسم البيانات اي نضيف بيانات الى حقل البيانات, وثانيا حقل العنوان وهنا يضاف عنوان العقدة اللاحقة في حقل العنوان, وبما اننا سنضيف هذه العقدة الى نهاية القائمة الموصولة, وهذا يعني ان هذه العقدة ستكون اخر عقدة في القائمة, اي ان العنوان الذي سيوضع في حقل العنوان هو لا شيء **(null)**.

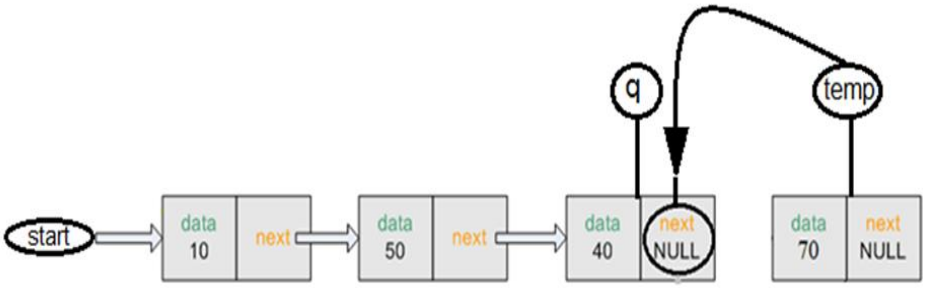
2. يتم خلق او الاعلان عن متغير من نوع المؤشر ويوضع عنوان عقدة البداية في هذا المؤشر (نفرض مؤشر باسم **q**) هذا المؤشر سيتم تحريكه من عقدة لاخرى لحين الوصول الى العقدة الاخيرة. ان السبب باستخدام مؤشر ثان يؤشر على العقدة الاولى لان العقدة الاولى دائما يؤشر عليها مؤشر يسمى مؤشر البداية او مؤشر الراس **(head or start)** ويجب ان نحافظ عليه لكي نتمكن من الوصول الى القائمة متى ما رغبنا اما اذا تم استخدام مؤشر البداية وتم تحريكه الى العقدة الثانية فأننا سنفقد عنوان العقدة الاولى والقائمة.



شكل 6.9: كيفية ربط العقدة الجديدة في نهاية القائمة

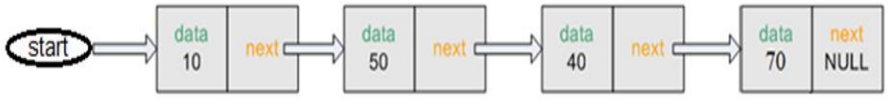
3. عند تحريك المؤشر ووصوله الى العقدة الاخيرة فأننا ببساطة سنضيف عنوان العقدة الجديدة التي يراد اضافتها الى حقل العنوان في العقدة الاخيرة, حيث في هذه الحالة فان المؤشر **q** سيكون هو من يؤشر على العقدة الاخيرة اي ان عنوان العقدة الاخيرة يكون في هذا المؤشر. تتم عملية الاضافة باستخدام الايعاز التالي:

```
q->next = temp;
```



شكل 6.10: توضيح كيفية وصول المؤشر الثاني الى العقدة الاخيرة مع الحفاظ على مؤشر البداية

4. بهذا نكون قد ربطنا عقدة جديدة بنهاية القائمة الموصولة.



شكل 6.11: القائمة الاحادية بعد الاضافة

B. اضافة عقدة الى بداية القائمة الموصولة

دعنا نحاول اضافة عقدة جديدة الى بداية القائمة او بكلام اخر وضع عقدة جديدة في بداية القائمة. ان ربط عقدة جديدة في بداية القائمة هو عمل بسيط وذلك لأنه لا يتطلب المرور على عناصر القائمة وايجاد الموقع المطلوب لحشر العقدة به. عملية الاضافة تتم بربط العقدة الجديدة بالعقدة الاولى ثم تغيير اسم العقدة الجديدة لتكن عقدة البداية.

شكل 6.12: توضيح لإضافة عقدة في بداية القائمة الموصولة الاحادية

a) اذا لم تكن القائمة فارغة, يتم خلق عقدة جديدة **(temp)** ثم نبدأ بإضافة البيانات الى حقل البيانات للعقدة الجديدة المراد اضافتها.

b) اضافة عنوان العقدة الاولى الى حقل العنوان في العقدة المراد اضافتها الى القائمة. وحسب الايعاز التالي:

```
temp -> next = start ;
```

c) وضع عنوان العقدة المضافة في متغير عنوان العقدة الاولى للقائمة **(start)** اي ان المتغير الذي يحمل اسم العقدة الجديدة **temp** سيتم تغيير تسميته بحيث يكون اسمه **start** لكي يدل على انه العقدة الاولى او عقدة البداية.

شكل 6.13: كيفية ربط العقدة الجديدة بالقائمة.

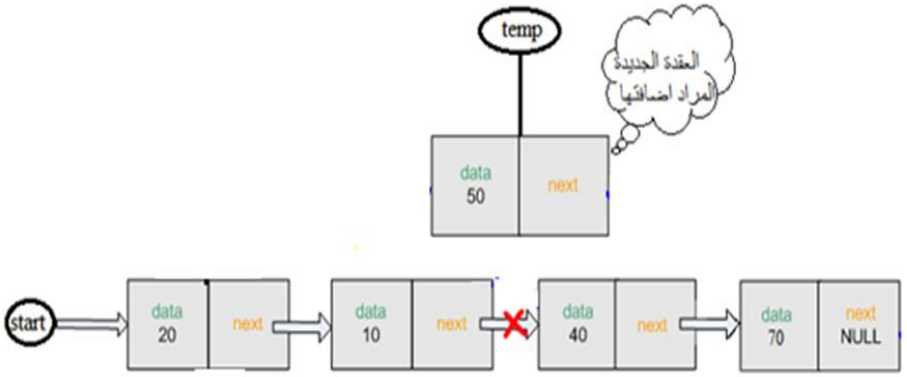
d) بهذا تكون العقدة الجديدة قد تمت اضافتها.

شكل 6.14: القائمة بعد الاضافة

C. حشر عقدة جديدة في اي مكان في القائمة

نفرض وجود الحاجة الى حشر عقدة جديدة في القائمة (في أي موقع محدد بالقائمة). هنا سيكون من المهم المرور على عناصر القائمة لإيجاد الموقع المطلوب لحشر العقدة فيه. في عملية المرور, يجب ان نحافظ على مؤشرين احدهما يشير الى العقدة السابقة والاخر يشير الى العقدة اللاحقة في القائمة (بمعنى ان يكون المؤشران على عقدتين متجاورتين). وذلك لان العقدة عندما يراد حشرها بين

عقدتين, فلابد ان تكون هناك عقدة سابقة لها واخرى لاحقة لها (أي اننا نحتاج الى عناوين عقدتين لنربط العقدة الجديدة بهما).



شكل 6.15: شكل توضيحي لقائمة تضاف لها عقدة جديدة في موقع محدد.

a) دائما نفحص ان تكون القائمة غير فارغة. فان كانت غير فارغة نستمر بعملنا.

b) يتم خلق عقدة جديدة, ومن ثم نضيف البيانات الى العقدة المراد اضافتها.

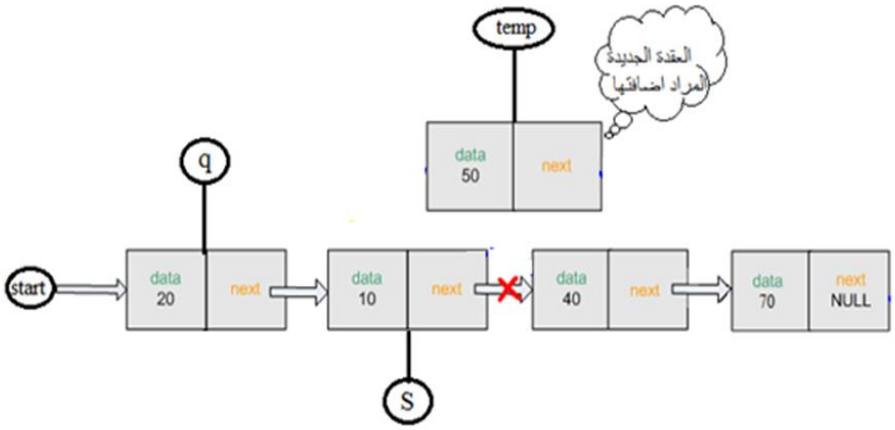
c) في هذه المسالة توجد حالتين وهما.. الاولى اضافة العقدة قبل عقدة محددة في القائمة, والثانية اضافة العقدة بعد عقدة محددة في القائمة .. ان هاتين الحالتين تختلفان بشكل بسيط في طريقة المعالجة.

d) في كلتا الحالتين سنبحث عن العقدة المحددة المراد الوصول لها, اي كأن تكون عقدة تحتوي على بيانات معينة, او بموقع معين مثلا العقدة الثالثة او الرابعة وهكذا, في هذه الحالة نتبع الخطوات التالية:

- ❖ اذا كان المطلوب اضافة العقدة قبل عقدة محددة (مثلا قبل العقدة الثالثة, او قبل العقدة التي تحتوي القيمة 40), عندها سنعمل ما يلي:

I. استحداث اثنان من المتغيرات من نوع مؤشر يسند عنوان العقدة الاولى لاحدهم, بينما يسند للأخر عنوان العقدة الثانية (يكون المؤشران على عقدتين متجاورتين).

```
q = start;

S = start -> next ;
```



شكل 6.16: بيان كيفية وضع المؤشرات على القائمة.

II. يحرك المؤشران الى العقدة اللاحقة (من خلال حلقة تكرار) بمعنى انه في كل مرة يحرك المؤشرين عقدة واحدة وبذلك نضمن انهما سيكونان على عقدتين متجاورتين دائما.

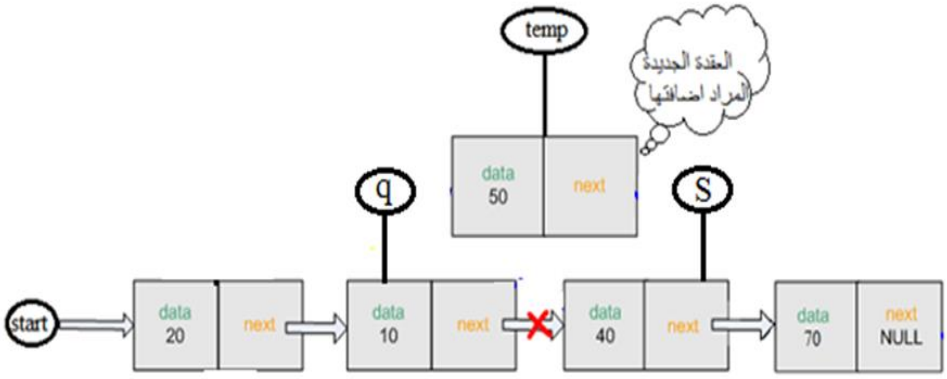
III. يتم فحص محتويات العقدة (او تسلسل العقدة اذا كان المطلوب عقدة بموقع معين في القائمة) التي يؤشر عليها المؤشر الاول (المؤشر **S**) لمعرفة اذا كانت العقدة المطلوبة ام لا.

IV. ان لم تكن العقدة المطلوبة يتم تحريك المؤشرين الاول والثاني كل منهما لعقدة واحدة (نكرر, هنا سنضمن دائما ان يكون المؤشران الاول والثاني

على عقدتين متجاورتين, بحيث المؤشر الثاني يؤشر على العقدة السابقة للمؤشر الاول).

V. التأكد بشكل مستمر ان المؤشر الاول **(S)** لم يصل الى نهاية القائمة قبل تحريكه. فان وصل الى نهاية القائمة هذا يعني عدم وجود العقدة التي نبحث عنها لغرض ان نضيف العقدة بجوارها.

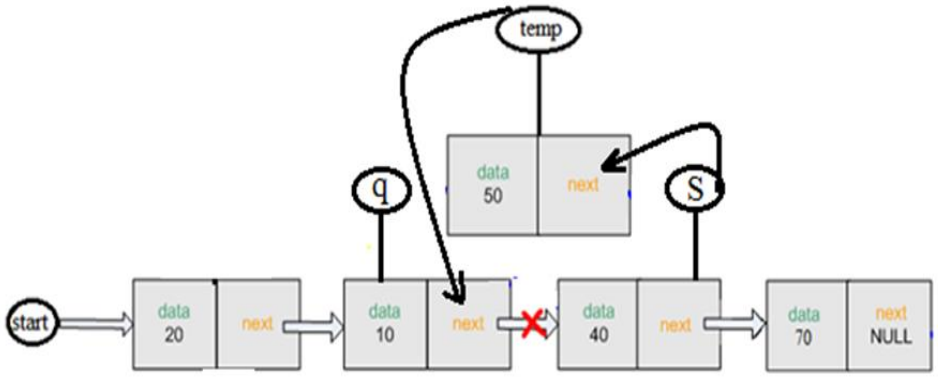
VI. في حالة الوصول الى العقدة المطلوبة سيكون المؤشر الاول **(S)** يؤشر عليها بينما المؤشر الثاني **(q)** يؤشر على العقدة السابقة لها.



شكل 6.17: وضع المؤشرات بعد وصولها الى العقدة المطلوبة.

VII. اضافة عنوان العقدة المحددة (المراد الاضافة قبلها) الى حقل العنوان في العقدة الجديدة المراد اضافتها (هذا العنوان موجود في حقل العنوان للعقدة التي يؤشر عليها المؤشر الاول **S**)

VIII. اضافة عنوان العقدة المراد اضافتها الى حقل العنوان للعقدة التي يؤشر عليها المؤشر الثاني **q**.



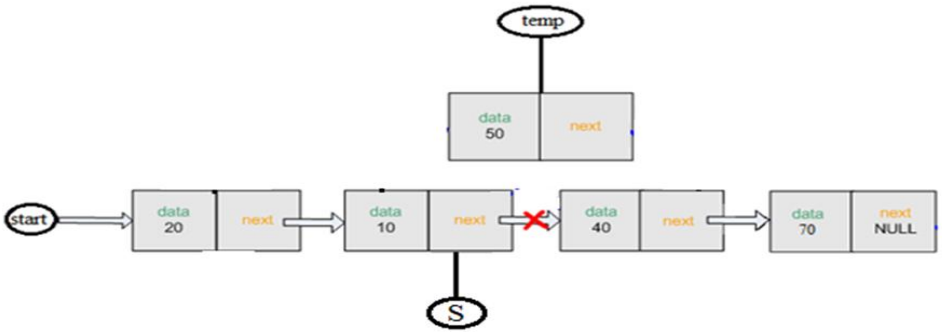
شكل 6.18: يوضح كيفية ربط العقدة الجديدة بالقائمة وطريقة وضع العناوين.

```
temp -> next = S ;

q -> next = temp ;
```

IX. بهذا تكون العقدة قد تمت اضافتها.

e) اما اذا كان المطلوب اضافة العقدة بعد عقدة محددة (نفرض الاضافة بعد العقدة الثانية التي تحتوي القيمة 10) فيكون كما يلي:

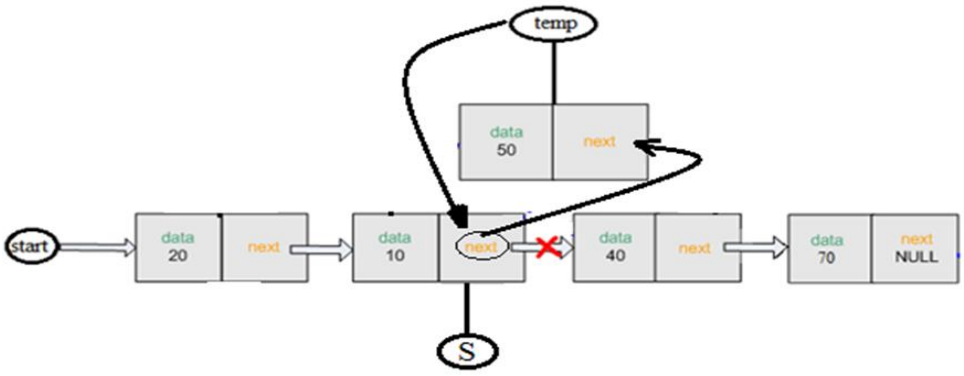
I. هنا لا نحتاج الى مؤشرين فمؤشر واحد يكفي وهو المؤشر الاول **S** الذي سيصل الى العقدة المحددة.



شكل 6.19: اضافة عقدة بعد عقدة محددة.

II. يضاف عنوان العقدة اللاحقة (اللاحقة للعقدة المراد الاضافة بعدها وهي التي يؤشر عليها المؤشر**S**, هذا العنوان موجود في حقل عنوان العقدة المراد الاضافة بعدها) الى حقل العنوان في العقدة المراد اضافتها. (ارجو الانتباه الى ان عنوان العقدة اللاحقة للعقدة التي يؤشر عليها المؤشر **S** هو موجود في الحقل **next** في الشكل 6.20).

III. يضاف عنوان العقدة المراد اضافتها الى حقل عنوان العقدة المراد الاضافة بعدها.



شكل 6.20: توضيح كيفية اضافة العقدة بعد عقدة محددة وطريقة اضافة العناوين.

```
temp -> next = S -> next ;

S -> next = temp ;
```

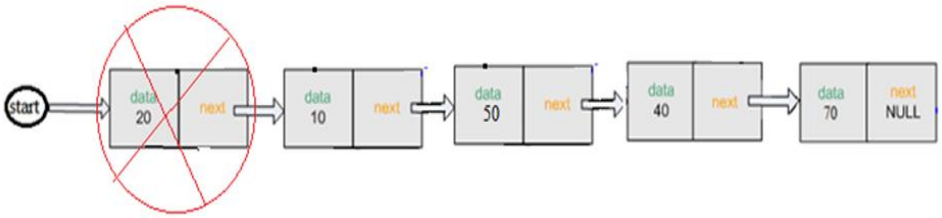
IV. تم هنا اضافة العقدة.

شكل 6.21: القائمة الموصولة بعد اضافة عقدة جديدة.

D. حذف عقدة من القائمة الموصولة

عملية حذف عقدة من قائمة موصولة تتضمن اكثر من حالة وهي:

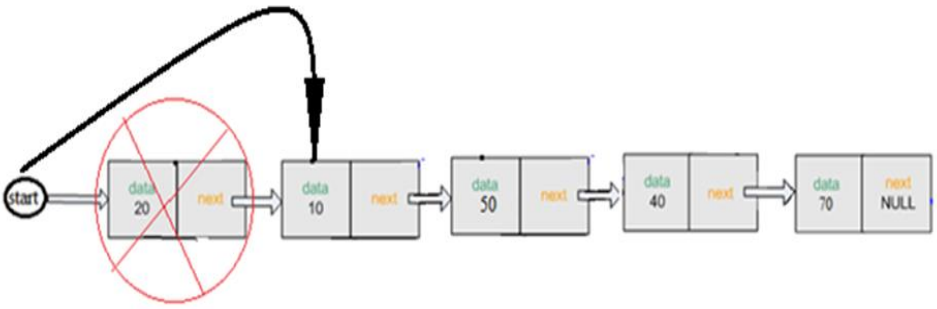
- **حذف عقدة من بداية القائمة الموصولة**



شكل 6.22: قائمة موصولة مطلوب حذف العقدة الاولى منها.

ان حذف عقدة من بداية القائمة عملية لاتحتاج الى جهد كبير, وهي تتمركز بتغيير موقع مؤشر العقدة الاولى ليوضع على العقدة الثانية (بحيث تكون هذه العقدة هي العقدة الاولى), هذه العملية من الممكن انجازها وذلك بان نضع عنوان العقدة الثانية في المتغير start.

يتم تحرير مساحة العقدة الاولى (هذا يتم بوضع مؤشر على العقدة الاولى قبل ان يتم تسمية العقدة الثانية باسم العقدة الاولى).

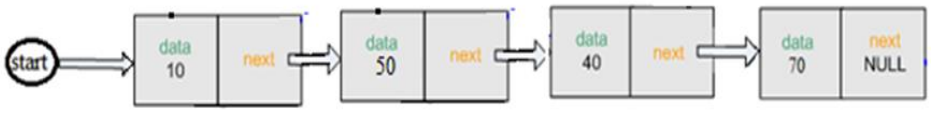


شكل 6.23: توضيح طريقه الحذف وموقع المؤشر.

Q = start ;

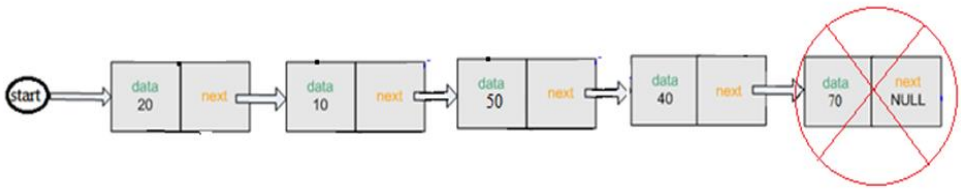
start = start->next;

free (Q) ;



شكل 6.24: القائمة بعد الحذف.

❖ حذف عقدة من نهاية القائمة الموصولة.



شكل 6.25: قائمة موصولة محدد فيها العقدة المراد حذفها.

حذف عقدة من نهاية القائمة الموصولة يتطلب الوصول الى العقدة النهائية لغرض حذفها, لاحظ هنا اذا وصلنا الى العقدة النهائية وتم حذفها فان العقدة السابقة لها هي التي ستكون العقدة النهائية وهذا يعني من المفروض اضافة null الى حقل عنوان العقدة التي تسبق العقدة النهائية الحالية.

من هذا فأننا نحتاج الى مؤشر يصل الى العقدة النهائية, وكذلك مؤشر اخر يؤشر على العقدة السابقة للعقدة النهائية لكي نحافظ عليها عند حذف العقدة النهائية.

اذن سنخلق مؤشرين احدهم يوضع على العقدة الاولى (S) والثاني (q) يوضع على العقدة الثانية, ويتم تحريك المؤشرين بمقدار عقدة واحدة في كل مرة لحين وصول المؤشر الاول (S) الذي وضع ابتداءا على العقدة الثانية الى العقدة النهائية.

شكل 6.26: القائمة الموصولة بعد وصول المؤشرات الى اماكنها المحددة.

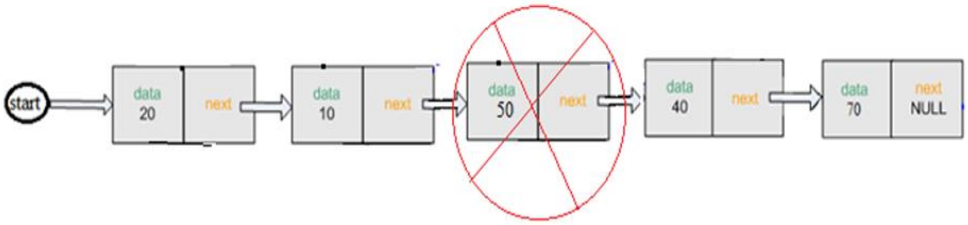
عند الوصول الى العقدة النهائية يتم وضع null في حقل العنوان للعقدة السابقة للعقدة النهائية والمؤشر عليها بالمؤشر q وبالتالي فان العقدة النهائية ستكون خارج القائمة.

يتم تحرير مساحة العقدة الاخيرة والمؤشر عليها بواسطة المؤشر S.

```
q->next = null ;
```

free (S) ;

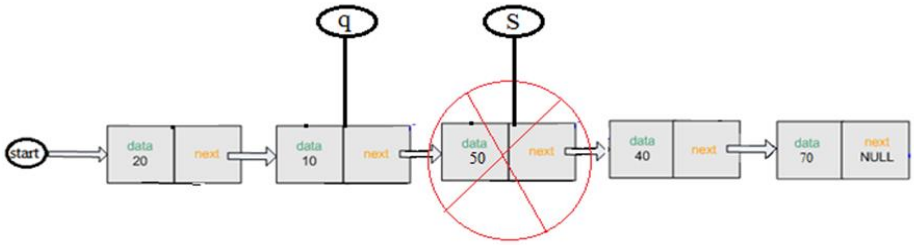
- **الحالة الثالثة هي حذف عقدة من داخل القائمة اي ان هذه العقدة توجد بين عقدتين.**



شكل 6.27: قائمة موصولة محدد بها العقدة المراد حذفها.

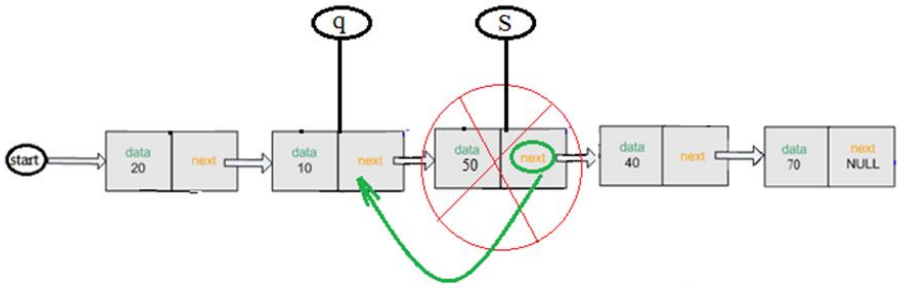
في هذه الحالة ايضا سنحتاج الى مؤشرين مؤشر سيصل الى العقدة المراد حذفها والاخر يكون على العقدة السابقة لها, لأننا بالنتيجة نحتاج ان نربط العقدة السابقة للعقدة المراد حذفها بالعقدة التي تلي العقدة المراد حذفها بحيث تكون العقدة المراد حذفها خارج القائمة.

نخلق متغيرين من نوع مؤشر ونضعهما على العقدة الاولى والثانية, ونبدأ بتحريك المؤشرين بمقدار عقدة واحدة في كل مرة لحين وصول المؤشر الاول S الى العقدة المطلوب حذفها (دائما عند تحريك المؤشرات يجب ان يكون المؤشرين على عقدتين متجاورتين).



شكل 6.28: قائمة موصولة مع المؤشرات التي وصلت الى العقدة المراد حذفها.

تتم عملية الحذف وذلك بوضع عنوان العقدة اللاحقة للعقدة التي يؤشر عليها المؤشر الاول S (وهي العقدة المراد حذفها) في حقل عنوان العقدة التي يؤشر عليها المؤشر الثاني q (لاحظ هنا ان عنوان العقدة اللاحقة دائما موجود في حقل العنوان للعقدة التي يؤشر عليها المؤشر). يتم تحرير مساحة العقدة المحذوفة.

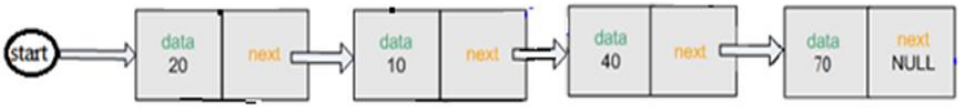


شكل 6.29: كيفية انتقال العناوين عند حذف عقدة بين عقدتين في قائمة موصولة احادية.

q-> next = S-> next ;

free (S) .

وبهذا فان العقدة المطلوب حذفها تم حذفها.



شكل 6.30: القائمة بعد اتمام عملية الحذف.

E. ترتيب عناصر قائمة موصولة

ترتيب القائمة الموصولة سهل جدا الى حد ما. بداية نخلق مؤشرين يوضع احدهم على العقدة الاولى والثاني يوضع على العقدة الثانية. هنا يجب ان نأخذ بيانات العقدة الاولى ونقارنها مع بيانات جميع العقد اللاحقة ومتى ما وجدنا بيانات اكبر يتم استبدالها (في حالة الترتيب التنازلي). بعدها ننتقل الى العقدة الثانية ونقارن بياناتها مع جميع بيانات العقد اللاحقة لها ويتم ابدال القيم الكبيرة مع الصغيرة وهكذا لحين اكمال جميع العقد.

❖ الدالة البرمجية للترتيب هي:

```
void ListSort ( ) {

node   *temp1,  *temp2 ;

int x = 0 ;

for (temp1 = start ; temp1 - > next != null ; temp1 = temp1 - > next ) {

for (temp2 = temp1 - > next; temp2 != null ; temp2 = temp2 - > next )

if ( (temp1 - > data)   <   (temp2 - > data)) {

x = temp1 - > data ;

temp1 - > data =  temp2 - > data ;
```

temp2 - > data = x ; } } }

F. تحديث بيانات عقدة ضمن القائمة الموصولة

ان تحديث بيانات أي عقدة في القائمة الموصولة هي من الاعمال السهلة وهي لا تحتاج الا لمؤشر واحد يوضع على اول عقدة في القائمة, ثم يحرك هذا المؤشر لحين الوصول الى العقدة المطلوبة (سواء كانت العقدة نبحث عنها بدلالة محتوياتها او موقعها في القائمة), بعد الوصول الى العقدة المطلوبة يتم تحديث بياناتها.

❖ البرنامج الكامل لجميع العمليات التي تجرى على القوائم الموصولة الاحادية

```
/*
 * C++ Program to Implement Singly Linked List
 */
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
/*
 * الاعلان عن العقدة تكون.. على شكل تركيب *
 */
struct node
{
   int info;
   struct node *next;
```

```
} *start ;
 /*
 * الاعلان عن الصنف
 */
class single_llist
{
   public:
      node* create_node(int);
      void insert_begin();
      void insert_pos();
      void insert_last();
      void delete_pos();
      void sort();
      void search();
      void update();
      void reverse();
      void display();
      single_llist()
      {
         start = NULL;
      }
};
```

```
main()
{
   int choice, nodes, element, position, i;
   single_llist  sl ;
   start = NULL;
   while (1)
   {
      cout<<endl<<"---------------------------------"<<endl;
      cout<<endl<<"Operations on singly linked list"<<endl;
      cout<<endl<<"---------------------------------"<<endl;
      cout<<"1.Insert Node at beginning"<<endl;
      cout<<"2.Insert node at last"<<endl;
      cout<<"3.Insert node at position"<<endl;
      cout<<"4.Sort Link List"<<endl;
      cout<<"5.Delete a Particular Node"<<endl;
      cout<<"6.Update Node Value"<<endl;
      cout<<"7.Search Element"<<endl;
      cout<<"8.Display Linked List"<<endl;
      cout<<"9.Reverse Linked List "<<endl;
      cout<<"10.Exit "<<endl;
      cout<<"Enter your choice : ";
      cin>>choice;
```

```
switch(choice)
{
case 1:
   cout<<"Inserting Node at Beginning: "<<endl;
   sl.insert_begin();
   cout<<endl;
   break;
case 2:
   cout<<"Inserting Node at Last: "<<endl;
   sl.insert_last();
   cout<<endl;
   break;
case 3:
   cout<<"Inserting Node at a given position:"<<endl;
   sl.insert_pos();
   cout<<endl;
   break;
case 4:
   cout<<"Sort Link List: "<<endl;
   sl.sort();
   cout<<endl;
   break;
```

```
case 5:
   cout<<"Delete a particular node: "<<endl;
   sl.delete_pos();
   break;
case 6:
   cout<<"Update Node Value:"<<endl;
   sl.update();
   cout<<endl;
   break;
case 7:
   cout<<"Search element in Link List: "<<endl;
   sl.search();
   cout<<endl;
   break;
case 8:
   cout<<"Display elements of link list"<<endl;
   sl.display();
   cout<<endl;
   break;
case 9:
   cout<<"Reverse elements of Link List"<<endl;
   sl.reverse();
   cout<<endl;
```

```
            break;
        case 10:
            cout<<"Exiting..."<<endl;
            exit(1);
            break;
        default:
            cout<<"Wrong choice"<<endl;
        }
    }
}

/*
 * خلق عقدة
 */
node *single_llist::create_node(int value)
{
    struct node *temp, *s;
    temp = new(struct node);
    if (temp == NULL)
    {
        cout<<"Memory not allocated "<<endl;
        return 0;
```

```
    }
else
    {
        temp->info = value;
        temp->next = NULL;
        return temp;
    }
}

/*
 * اضافة عقدة في البداية
 */
void single_llist::insert_begin()
{
    int value;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *p;
    temp = create_node(value);
    if (start == NULL)
    {
        start = temp;
        start->next = NULL;
```

```
    }
    else
    {
        p = start;
        start = temp;
        start->next = p;
    }
    cout<<"Element Inserted at beginning"<<endl;
}

/*
 *  اضافة عقدة في نهاية القائمة
 */
void single_llist::insert_last()
{
    int value;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *s;
    temp = create_node(value);
    s = start;
    while (s->next != NULL)
```

```
    {
        s = s->next;
    }
    temp->next = NULL;
    s->next = temp;
    cout<<"Element Inserted at last"<<endl;
}
/*
 * اضافة عقدة في أي موقع في القائمة
 */
void single_llist::insert_pos()
{
    int value, pos, counter = 0;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *s, *ptr;
    temp = create_node(value);
    cout<<"Enter the postion at which node to be inserted: ";
    cin>>pos;
    int i;
    s = start;
    while (s != NULL)
    {
```

```
    s = s->next;
    counter++;
}
if (pos == 1)
{
    if (start == NULL)
    {
        start = temp;
        start->next = NULL;
    }
    else
    {
        ptr = start;
        start = temp;
        start->next = ptr;
    }
}
else    if (pos > 1  && pos <= counter)
{
    s = start;
    for (i = 1; i < pos; i++)
    {
```

```
            ptr = s;
            s = s->next;
        }
        ptr->next = temp;
        temp->next = s;
    }
    else
    {
        cout<<"Positon out of range"<<endl;
    }
}
    /*
 * ترتيب القائمة الموصولة
 */
void single_llist::sort()
{
    struct node *ptr, *s;
    int value;
    if (start == NULL)
    {
        cout<<"The List is empty"<<endl;
        return;
    }
```

```
    ptr = start;
    while (ptr != NULL)
    {
        for (s = ptr->next;s !=NULL;s = s->next)
        {
            if (ptr->info > s->info)
            {
                value = ptr->info;
                ptr->info = s->info;
                s->info = value;
            }
        }
        ptr = ptr->next;
    }
}
/*
 *  حذف عنصر في موقع ما في القائمة
 */
void single_llist::delete_pos()
{
    int pos, i, counter = 0;
    if (start == NULL)
```

```
{
   cout<<"List is empty"<<endl;
   return;
}
cout<<"Enter the position of value to be deleted: ";
cin>>pos;
struct node *s, *ptr;
s = start;
if (pos == 1)
{
   start = s->next;
}
else
{
   while (s != NULL)
   {
      s = s->next;
      counter++;
   }
   if (pos > 0 && pos <= counter)
   {
      s = start;
      for (i = 1;i < pos;i++)
```

```
            {
               ptr = s;
               s = s->next;
            }
            ptr->next = s->next;
         }
         else
         {
            cout<<"Position out of range"<<endl;
         }
         free(s);
         cout<<"Element Deleted"<<endl;
      }
   }
   /*
    * تحديث بيانات عقدة
    */
   void single_llist::update()
   {
      int value, pos, i;
      if (start == NULL)
      {
```

```
    cout<<"List is empty"<<endl;
    return;
  }
  cout<<"Enter the node postion to be updated: ";
  cin>>pos;
  cout<<"Enter the new value: ";
  cin>>value;
  struct node *s, *ptr;
  s = start;
  if (pos == 1)
  {
    start->info = value;
  }
  else
  {
for (i = 0;i < pos - 1;i++)
    {
      if (s == NULL)
      {
        cout<<"There are less than "<<pos<<" elements";
        return;
      }
      s = s->next;
```

```
        }
        s->info = value;
    }
    cout<<"Node Updated"<<endl;
}

/*
 * البحث عن عنصر
 */
void single_llist::search()
{
    int value, pos = 0;
    bool flag = false;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    cout<<"Enter the value to be searched: ";
    cin>>value;
    struct node *s;
    s = start;
```

```
   while (s != NULL)
   {
      pos++;
      if (s->info == value)
      {
         flag = true;
         cout<<"Element "<<value<<" is found at position "<<pos<<endl;
      }
      s = s->next;
   }
   if (!flag)
      cout<<"Element "<<value<<" not found in the list"<<endl;
}

/*
 * عكس القائمة الموصولة
 */
void single_llist::reverse()
{
   struct node *ptr1, *ptr2, *ptr3;
   if (start == NULL)
   {
      cout<<"List is empty"<<endl;
```

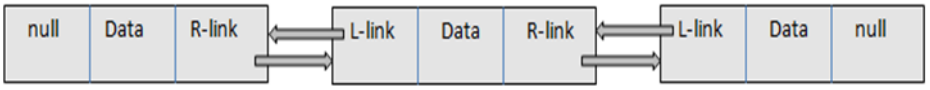
```
        return;
    }
    if (start->next == NULL)
    {
        return;
    }
    ptr1 = start;
    ptr2 = ptr1->next;
    ptr3 = ptr2->next;
    ptr1->next = NULL;
    ptr2->next = ptr1;
    while (ptr3 != NULL)
    {
        ptr1 = ptr2;
        ptr2 = ptr3;
        ptr3 = ptr3->next;
        ptr2->next = ptr1;
    }
    start = ptr2;
}

/*
```

```
 * عرض عناصر القائمة الموصولة
 */
void single_llist::display()
{
   struct node *temp;
   if (start == NULL)
   {
      cout<<"The List is Empty"<<endl;
      return;
   }
   temp = start;
   cout<<"Elements of list are: "<<endl;
   while (temp != NULL)
   {
      cout<<temp->info<<"->";
      temp = temp->next;
   }
   cout<<"NULL"<<endl;
}
```

6.10 القوائم الموصولة الثنائية Doubly Linked Lists

القوائم الموصولة الثنائية هي مجموعة من العقد (كما في القوائم الاحادية) ترتبط فيما بينها بواسطة مؤشرات مع ملاحظة ان كل عقدة في القوائم الثنائية تحتوي على ثلاث حقول وهي حقل البيانات وحقلين اخرين احدهما يشير الى العقدة اللاحقة ويسمى المؤشر الايمن, ومؤشر اخر يشير الى العقدة السابقة وهو المؤشر الايسر. ان الفرق بين القوائم الموصولة الاحادية والقوائم الموصولة الثنائية هو ان القوائم الموصولة الثنائية تحتوي على مؤشر اضافي في كل عقدة يؤشر الى العقدة السابقة.

لاحظ عندما لا توجد عقدة يؤشر عليها المؤشر فانه سيؤشر الى null وهذه الحالة ستحدث في العقدة الاخيرة حيث لا توجد عقدة لاحقة لها وبالتالي فان المؤشر الايمن سيؤشر الى null وكذلك العقدة الاولى فان المؤشر الايسر لا يؤشر الى اي عقدة سابقة للعقدة الاولى وبالتالي سيؤشر المؤشر الايسر للعقدة الاولى الى null.



شكل 6.31: قائمة موصولة ثنائية

6.10.1 العمليات التي تجرى على القوائم الموصولة الثنائية

هناك عدد من العمليات الرئيسة والاساسية التي تجرى على القوائم الموصولة الثنائية, وفي كل العمليات ودائما يجب ان يتم التأكد ان القائمة غير فارغة, العمليات التي تجرى على القوائم الموصولة الثنائية هي:

A. خلق قائمة ثنائية.
B. اضافة عقدة في نهاية القائمة الثنائية.
C. اضافة عقدة في بداية القائمة الثنائية.

D. اضافة عقدة في موقع محدد في القائمة.
E. حذف عقدة من القائمة.
F. عرض محتويات القائمة.

A. خلق قائمة

ان عملية خلق قائمة تعني اضافة عقد جديدة عندما لا تكون هناك اي عقدة في القائمة (اي لا توجد قائمة محددة), و بالتأكيد فان الخطوة الاولى في انشاء اي قائمة هو اضافة العقدة الاولى او خلق العقدة الاولى في القائمة.

بعد ان تتم اضافة العقدة الاولى يصبح بالإمكان اضافة عقد اخرى وحسب الحاجة, وهذا سيتم من خلال اضافة عقدة جديدة في نهاية القائمة التي تم خلقها سواء كانت هذه القائمة تحتوي على عقدة واحدة او اكثر ففي كل الحالات تتم الاضافة في ذيل القائمة.

ان عملية خلق عقدة تتم من خلال الايعاز الخاص بخلق عقدة جديدة بعد ان يتم تعريف هيكل العقدة structure.

```
struc node
{
int data ;
node *next ;
node *prev ;
}
```

بعد ان تتم عملية خلق العقدة (والتي سنسميها temp) تتم اضافة البيانات الى حقل البيانات, ثم نقوم بإضافة العناوين الى حقول العناوين الايمن والايسر. لاحظ هنا بما اننا نخلق عقدة جديدة لقائمة غير موجودة اصلا (اي ان هذه العقدة هي العقدة الاولى او النواة لخلق قائمة جديدة) فان حقول العناوين الايمن والايسر سوف لا

تؤشر الى اي عقدة سابقة او لاحقة لعدم وجود عقد اخرى وبالتالي فسنضع في هذه الحقول null.

```
new temp ;
temp - > data = value ;
temp -> next = null ;
if (start == null)
{
temp -> prev = null ;
start = temp ;
}
```

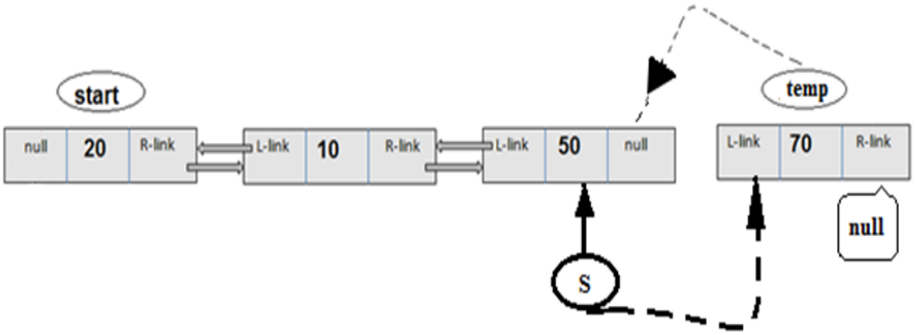
لاحظ هنا اننا فحصنا عنوان العقدة الاولى start لنتأكد من انها تساوي null للدلالة على عدم وجود قائمة, حيث انه اذا كانت هناك قائمة فان العنوان start سيكون فيه قيمة عنوان وليس null.

B. اضافة عقدة الى نهاية القائمة

خلق عقدة جديدة (نسميها temp). اضافة بيانات في حقل البيانات للعقدة الجديدة, ولما كانت هذه العقدة ستكون العقدة الاخيرة فعليه سيكون حقل العنوان الايمن لهذه العقدة يؤشر على null, اذن نضيف null الى حقل العنوان الايمن للعقدة الجديدة.

للوصول الى العقدة الاخيرة في القائمة, نضع مؤشر على العقدة الاولى ليكن تحت مسمى (S).

نعمل حلقة تكرار تبدا من العقدة الاولى وتنتهي عندما يكون عنوان العقدة اللاحقة (العنوان الايمن) يساوي null, بحيث في كل دورة يتم نقل المؤشر الى العقدة اللاحقة طالما مؤشر العنوان الايمن لايساوي null.



شكل 6.32: اضافة عقدة الى نهاية القائمة.

عندما يصل المؤشر الى العقدة الاخيرة نقوم بإضافة العقدة الجديدة بعدها وكما يلي:

يوضع عنوان العقدة الجديدة في حقل العنوان الايمن للعقدة الاخيرة, وبذلك فان العقدة الاخيرة في القائمة ستؤشر على العقدة الجديدة بحيث تكون العقدة الجديدة هي الاخيرة). يوضع عنوان العقدة الاخيرة في حقل العنوان الايسر للعقدة الجديدة, كما في الشكل 6.32.

```
s = start ;
while (s -> next != null)
{     s = s -> next ;   }
s -> next = temp ;
temp -> prev = s ;
```

C. اضافة عقدة في بداية القائمة

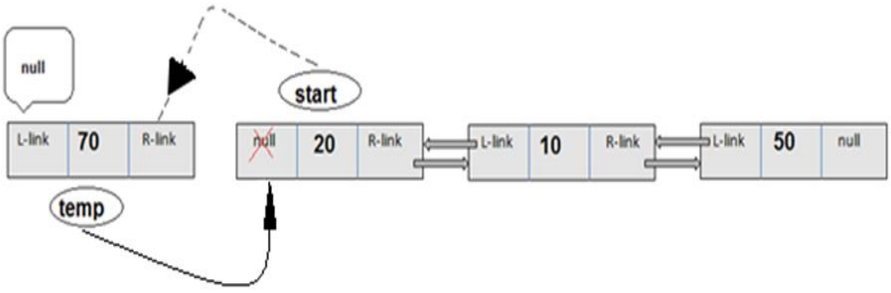
يتم خلق عقدة جديدة باسم (temp). اضافة البيانات الى حقل البيانات.

لما كانت هذه العقدة ستضاف في بداية القائمة, عليه فهي ستكون العقدة الاولى ولذلك سوف يكون عنوان العقدة السابقة null, اي يتم اضافة null في حقل العنوان الايسر.

اما مؤشر حقل العنوان الايمن للعقدة الجديدة فيجب ان يؤشر على العقدة التي هي الاولى قبل الاضافة حيث ستصبح العقدة الثانية بعد الاضافة (يوضع عنوان العقدة الاولى فيه).

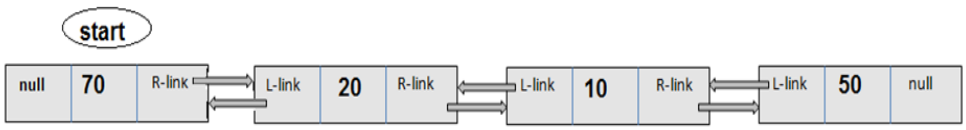
ولكي ننجز ربط العقدة الجديدة بالعقدة التي كانت هي الاولى فالمفروض ان تكون العقدة الجديدة قبلها عند الاضافة كما اسلفنا, لذلك فان حقل العنوان الايسر للعقدة التي كانت الاولى سيوضع به عنوان العقدة الجديدة (temp).

اخيرا يتم تسمية العقدة الجديدة باسم عقدة البداية start, لاحظ الشكل 6.33.



شكل 6.33: كيفية ربط العقدة الجديدة في بداية القائمة الثنائية.

```
temp -> prev = null ;
temp -> data = value ;
temp -> next = start ;
start -> prev = temp ;
start = temp ;
```



شكل 6.34: القائمة الثنائية بعد الاضافة.

D. اضافة عقدة في موقع محدد في القائمة

يتم خلق عقدة جديدة وتضاف البيانات الى حقل البيانات في العقدة الجديدة.

في هذه الحالة يتم تحديد الموقع المراد الاضافة بعده او قبله.

يوضع مؤشر على عقدة البداية او العقدة الاولى.

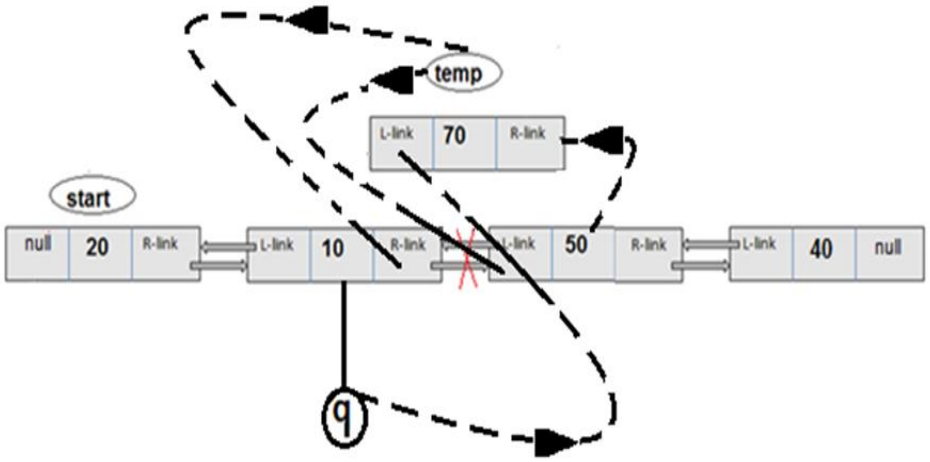
يتم عمل حلقة تكرار من العقدة الاولى الى الموقع المراد الاضافة بعده او قبله (في القوائم الثنائية نحتاج مؤشر واحد يوضع على عقدة البداية في كل الاحوال وليس كما في القوائم الاحادية التي احيانا نحتاج مؤشرين).

في كل دورة يتم نقل المؤشر الى العقدة اللاحقة. مع ملاحظة ضرورة التأكد من عدم الوصول الى العقدة الاخيرة (اذا تم الوصول الى العقدة الاخيرة قبل الوصول الى الموقع المطلوب يتم اظهار رسالة تفيد بان الموقع خارج حجم القائمة).

عند الوصل الى العقدة المراد الاضافة بعدها او قبلها, نتبع الخطوات التالية لغرض اتمام عملية الاضافة, في هذه الحالة لدينا احتمالان..

a) الاول ان تكون الاضافة بعد العقدة المحددة وهنا لدينا احتمالان:

- ❖ الاحتمال الاول.. ان تكون العقدة المراد الاضافة بعدها هي العقدة الاخيرة عندها تتم الاضافة كما سبق وان وضحنا عند الاضافة في نهاية القائمة.



شكل 6.35: اضافة عقدة بين عقدتين, الاضافة بعد عقدة محددة.

- ❖ الاحتمال الاخر ان تكون الاضافة في وسط القائمة عندها نتبع الخطوات التالية (الشكل 6.35):

نضع عنوان العقدة اللاحقة للعقدة المراد الاضافة بعدها (عنوانها موجود في حقل العنوان الايمن للعقدة q) في حقل العنوان الايمن للعقدة الجديدة.

نضع عنوان العقدة التي سنضيف بعدها في حقل العنوان الايسر للعقدة المضافة.

نضع عنوان العقدة الجديدة في حقل العنوان الايسر للعقدة التي تلي العقدة المراد الاضافة بعدها.

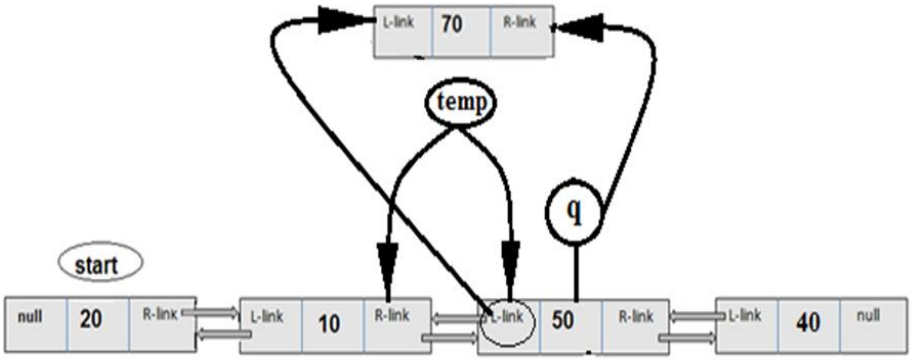
كذلك نضع عنوان العقدة الجديدة في حقل العنوان الايمن للعقدة التي سنضيف بعدها

```
temp -> next = q -> next ;
temp -> prev = q ;
q -> next -> prev = temp ;
q - > next = temp ;
```

b) الاحتمال الثاني هو ان تكون الاضافة قبل العقدة المحددة في القائمة.. وهنا لدينا احتمالان ايضا:

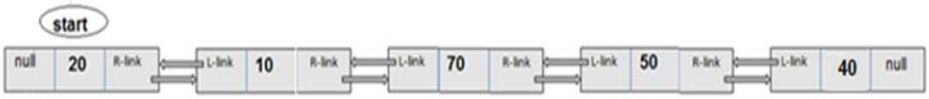
- ❖ الاحتمال الاول ان تكون الاضافة قبل العقدة الاولى وبهذا سنتبع ذات الخطوات للإضافة قبل العقدة الاولى.
- ❖ الاحتمال الثاني تكون الاضافة وسط القائمة, وسوف لا يختلف كثيرا عن الاضافة بعد عقدة محددة, بداية نحرك مؤشر ليصل الى العقدة المراد الاضافة قبلها.. ثم تتم اضافة عنوان العقدة الجديدة في حقل العنوان الايسر للعقدة المحددة المراد الاضافة قبلها, وفي حقل العنوان الايمن للعقدة السابقة للعقدة المحددة (وعنوانها موجود في حقل العنوان الايسر للعقدة المحدد الاضافة قبلها).

يتم اضافة عنوان العقدة المحدد الاضافة قبلها في حقل العنوان الايمن للعقدة الجديدة, ويضاف حقل عنوان العقدة السابقة للعقدة المحدد الاضافة قبلها في حقل العنوان الايسر للعقدة الجديدة, الشكل 6.36.



شكل 6.36: اضافة عقدة بين عقدتين, الاضافة قبل عقدة محددة.

```
temp -> next = q ;

temp -> prev = q -> prev ;

q -> prev = temp ;

q -> prev -> next = temp ;
```

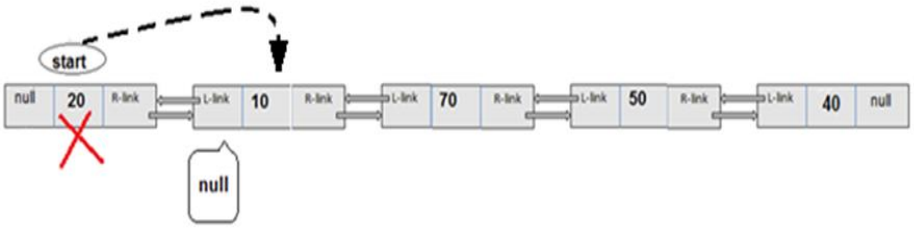


شكل 6.37: القائمة بعد عملية الاضافة في الوسط

E. حذف عنصر

عملية الحذف هنا تعتمد على ايجاد العقدة التي تحتوي قيمة معينة او محددة في موقع معين ليتم حذفها, ولإنجاز عملية الحذف علينا ان نلاحظ الاحتمالات التي تكون عليها العقدة المطلوبة وكما يلي:

a) الاحتمال الاول ان تكون العقدة المطلوب حذفها هي العقدة الاولى عندها نتبع الخطوات التالية لأجراء عملية حذف العقدة الاولى, الشكل 6.38.



شكل 6.38: حذف عقدة من بداية القائمة الموصولة الثنائية.

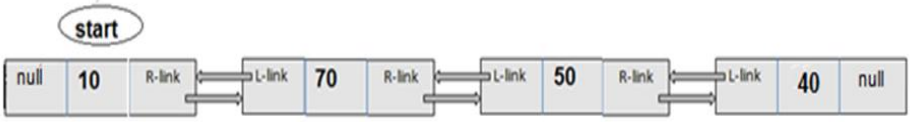
بداية نحفظ عنوان العقدة الاولى بمتغير نسميه مثلا temp (نستفيد منه لحفظ عنوان العقدة المحذوفة وامكانية تحرير مساحتها لاحقا باستخدام هذا الاسم).

نسمي العقدة الثانية باسم عقدة البداية اي نسميها start.

نضع null في حقل العنوان الايسر للعقدة الثانية (من الخطوة السابقة فان العقدة الثانية اصبحت هي عقدة البداية). عنوان العقدة الثانية موجود في حقل العنوان الايمن للعقدة الاولى.

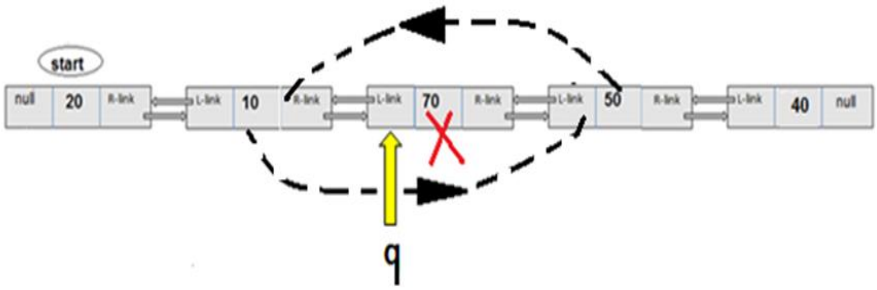
يتم تحرير مساحة العقدة المحذوفة.

```
temp = start ;
start = start -> next ;
start -> prev = null ;
free (temp) ;
```



شكل 6.39: القائمة بعد اجراء الحذف.

b) الاحتمال الثاني هو ان تكون العقدة المطلوب حذفها هي بين عقدتين عندها نتبع الخطوات التالية للحذف



شكل 6.40: حذف عقدة بين عقدتين في قائمة موصولة ثنائية.

نضع مؤشر على عقدة البداية لنسميه q.

نعمل حلقة تكرار من اول عقدة في القائمة لحين الوصول الى العقدة المطلوب حذفها, شكل 6.40.

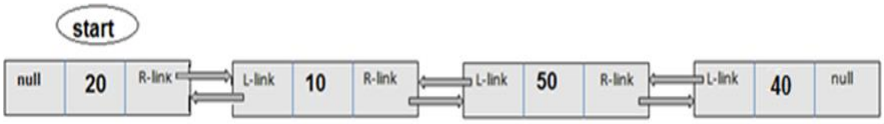
الان نبدأ عملية الحذف, وهي ببساطة تتضمن ربط العقدة السابقة للعقدة المراد حذفها مع العقدة اللاحقة للعقدة المراد حذفها وبهذا تكون هذه العقدة المراد حذفها خارج القائمة.

هذا يتم بوضع عنوان العقدة اللاحقة للعقدة المراد حذفها في حقل العنوان الايمن للعقدة السابقة للعقدة المراد حذفها (عنوان العقدة السابقة للعقدة التي نرغب بحذفها يكون موجود في حقل العنوان الايسر للعقدة المراد حذفها, وعنوان العقدة اللاحقة للعقدة المطلوب حذفها يكون في حقل العنوان الايمن لعقدة المطلوب حذفها), كما هو واضح في الشكل 6.40.

نضع عنوان العقدة السابقة للعقدة المراد حذفها في حقل العنوان الايسر للعقدة اللاحقة للعقدة المراد حذفها.

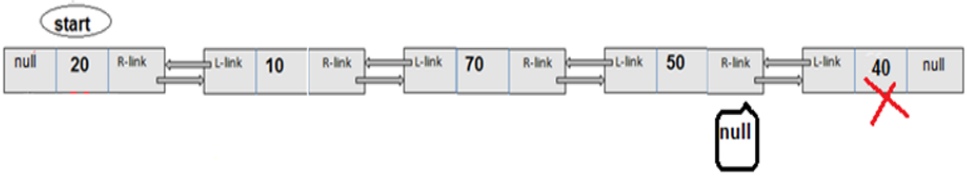
يتم تحرير مساحة العقدة المحذوفة. نفرض المطلوب حذف عقدة تحمل قيمة معينة عندها نتبع الايعازات التالية:

```
q = start ;
while (q -> next != null )
{
if (q -> data == value ) ;        {
      q -> next -> prev = q -> prev ;
      q -> prev -> next = q -> next ;
      cout << "Elemrnt Deleted \n" ;
       free (q) ;
       break ; }  }
```



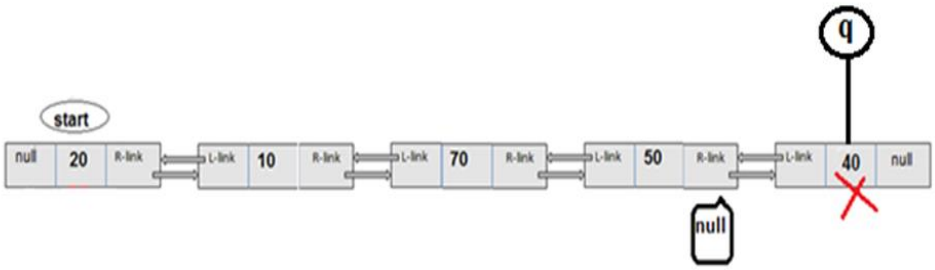
شكل 6.41: القائمة بعد حذف العقدة.

c) الاحتمال الاخير هو ان تكون العقدة المراد حذفها هي العقدة الاخيرة عند ذاك نتبع الخطوات التالية:



الشكل 6.42: حذف العقدة النهائية من القائمة الموصولة الثنائية.

كما سبق ووضحنا ان المؤشر الذي وضعناه على العقدة الاولى وتم تحريكه سيصل الى العقدة المطلوب حذفها, وفي هذه الحالة ستكون العقدة المطلوب حذفها هي العقدة الاخيرة (العقدة الاخيرة يتم التعرف عليها من فحص حقل العنوان الايمن حيث سنجد به null), شكل 6.43.



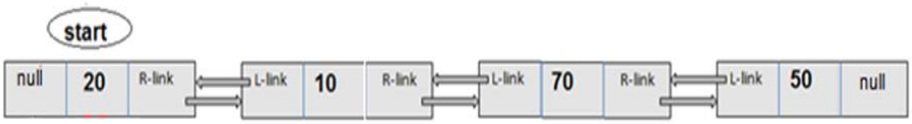
شكل 6.43: وصول المؤشر الى العقدة الاخيرة.

لكي نحذف هذه العقدة فان العقدة السابقة لها ستكون هي الاخيرة لذلك من المفروض ان يكون حقل العنوان الايمن للعقدة السابقة للعقدة الاخيرة المراد حذفها يحمل القيمة null.

عليه ضع null في حقل العنوان الايمن للعقدة السابقة للعقدة الاخيرة (لاحظ هنا ان عنوان العقدة السابقة للعقدة الاخيرة يكون موجود في حقل العنوان الايسر للعقدة الاخيرة).

تحرير مساحة العقدة الاخيرة المحذوفة.

```
q -> prev -> next = null ;

free (q) ;
```



شكل 6.44: القائمة بعد حذف العقدة الاخيرة.

F. عرض القائمة

نضع مؤشر على عقدة البداية.

نعمل حلقة تكرار من العقدة الاولى الى العقدة الاخيرة.

في كل دورة يتم عرض بيانات العقدة التي يؤشر عليها المؤشر ومن ثم ننقل المؤشر الى العقدة اللاحقة.

```
q = start ;
while ( q != null)
{
q = q -> next ;
}
```

G. حساب عدد العقد

نضع مؤشر على العقدة الاولى.

نحدد عداد وله قيمة ابتدائية مقدارها صفر.

نعمل حلقة تكرار تبدا من العقدة الاولى وتنتهي بالعقدة الاخيرة.

في كل دورة عندما ينتقل المؤشر الى العقدة اللاحقة يتم زيادة العداد بمقدار واحد.

بعد انتهاء حلقة التكرار سيكون العداد يحمل قيمة تمثل عدد العقد في القائمة.

```
int count = 0 ;
q = start ;
while (q != null)
```

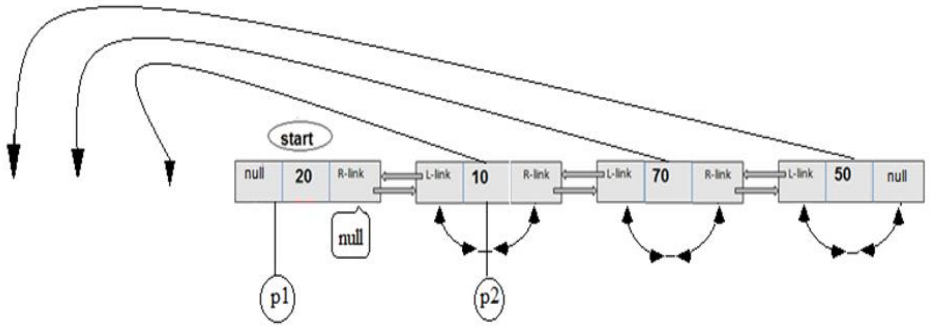
```
{
q = q -> next ;
count ++ ;
}
cout<< " Number of nodes in linked list =" << count ;
```

H. عكس القائمة

عملية عكس القائمة الموصولة الثنائية (اي ان تكون العقدة الاولى هي الاخيرة والعقدة الاخيرة هي الاولى) لا تعتبر عملية صعبة او معقدة وانما هي عملية في غاية السهولة, كل ماهو مطلوب ان تكون العقدة الاولى هي الاخيرة والعقدة الاخيرة هي الاولى, وجميع العقد يتم عكسها بعملية ابدال العناوين في حقول العناوين الايمن والايسر للعقدة (بمعنى ان يكون حقل العنوان الايمن.. ايسر وحقل العنوان الايسر.. ايمن). لاحظ هنا ان العقدة الثانية ستكون العقدة قبل الاخيرة لان العقدة الاولى اصبحت الاخيرة وهكذا.

نخلق مؤشرين ونضع احدهما على العقدة الاولى (p1) والمؤشر الثاني على العقدة الثانية (p2).

الان نجعل في حقل العنوان الايمن للمؤشر الاول القيمة null.



شكل 6.45: شكل توضيحي لطريقة عكس القائمة الموصولة الثنائية.

يوضع في حقل العنوان الايسر للعقدة التي يؤشر عليها المؤشر الاول (العقدة الاولى في هذه الحالة) عنوان العقدة التي يؤشر عليها المؤشر الثاني (اي هذه العقدة ستؤشر على المؤشر الثاني p2).

خلق حلقة تكرار يكون فيها شرط التوقف هو ان المؤشر الثاني p2 يساوي null.

داخل حلقة التكرار سيكون ما يلي

سننقل العقد الى مواقع سابقة للعقدة الاولى وحسب ترتيبها كما في الشكل 6.45, هذا يتم من خلال تغيير قيم العناوين في حقول العناوين للعقد, كما يلي:

في العقدة التي يؤشر عليها المؤشر الثاني p2, وضع العنوان الموجود في حقل العنوان الايمن في حقل العنوان الايسر.

وضع عنوان العقدة التي يؤشر عليها المؤشر p1 في حقل العنوان الايمن (اي وضع p1 في حقل العنوان الايمن).

نقل المؤشرين عقدة واحدة (دائما يبقى المؤشران متجاوران). لاحظ هنا ان المؤشر الاول تم نقله الى العقدة التي يؤشر عليها المؤشر الثاني ولا مشكلة بذلك (p1 = p2) , بينما المؤشر الثاني في الخطوة الثانية يتم نقله الى العقدة اللاحقة لها باستخدام ايعاز يبدو ظاهريا انه ينقله الى العقدة السابقة وليس اللاحقة حسب الايعاز (p2 = p2 -> prev) وسبب ذلك لأننا اساسا في خطوة سابقة غيرنا قيمة هذا الحقل (حقل عنوان العقدة السابقة) بحث وضعنا فيه عنوان العقدة اللاحقة لذل اقتضى التنويه).

بعد انتهاء او توقف التكرار يتم وضع عنوان المؤشر الاول p1 في المتغير start.

```
p1 = start ;
p2 = p1 -> next ;
p1 ->next = null ;
p1 -> prev = p2 ;
while (p2 != null )
{
p2 -> prev = p2 ->next ;
p2 -> next = p1 ;
p1 = p2 ;
p2 = p2 ->prev ;
}
```

❖ البرنامج الكامل لجميع العمليات التي تجرى على القوائم الموصولة الثنائية

```
/*
 * C++ Program to Implement Doubly Linked List
 */
#include<iostream>
#include<cstdio>
#include<cstdlib>
/*
  * الاعلان عن العقدة
```

```
 */
using namespace std;
struct  node
{
   int info;
   struct node *next;
   struct node *prev;
}*  start;

/*
الاعلان عن الصنف
 */
class double_llist
{
   public:
      void create_list(int value);
      void add_begin(int value);
      void add_after(int value, int position);
      void delete_element(int value);
      void search_element(int value);
      void display_dlist( );
      void count( );
      void reverse( );
```

```
        double_llist( ) ;
        {
           start = NULL  ;
        }
} ;

int main( )
{
   int choice, element, position;
   double_llist dl;
   while (1)
{
        cout<<endl<<"----------------------------"<<endl;
        cout<<endl<<"Operations on Doubly linked list"<<endl;
        cout<<endl<<"----------------------------"<<endl         ;
        cout<<"1.Create Node"<<endl;
        cout<<"2.Add at begining"<<endl;
        cout<<"3.Add after position"<<endl;
        cout<<"4.Delete"<<endl;
        cout<<"5.Display"<<endl;
        cout<<"6.Count"<<endl;
        cout<<"7.Reverse"<<endl;
```

```
cout<<"8.Quit"<<endl;
cout<<"Enter your choice : " ;
cin>>choice;
switch ( choice )
{
case 1:
   cout<<"Enter the element :" ;
   cin>>element;
   dl.create_list(element);
   cout<<endl;
   break;
case 2:
   cout<<"Enter the element :" ;
   cin>>element;
   dl.add_begin(element);
   cout<<endl;
   break;
case 3:
   cout<<"Enter the element :" ;
   cin>>element;
   cout<<"Insert Element after postion :" ;
   cin>>position ;
   dl.add_after(element, position) ;
```

```
    cout<<endl ;
    break;
case 4:
    if (start == NULL)
    {
       cout<<"List empty,nothing to delete"<<endl   ;
       break;
    }
    cout<<"Enter the element for deletion : " ;
    cin>>element;
    dl.delete_element(element) ;
    cout<<endl;
    break;
case 5:
    dl.display_dlist( ) ;
    cout<<endl;
    break;
case 6:
    dl.count( ) ;
    break    ;
case 7:
    if (start == NULL)
```

```
            {
               cout<<"List empty, nothing to reverse"<<endl;
               break;
            }
            dl.reverse( ) ;
            cout<<endl;
            break;
         case 8:
            exit(1) ;
         default:
            cout<<"Wrong choice"<<endl;
         }
      }
      return 0;
}

/*
* خلق قائمة موصولة ثنائية *
 */
void double_llist::create_list (int value)
{
   struct node *s, *temp;
   temp = new(struct node)  ;
```

```
    temp->info = value;
    temp->next = NULL;
    if (start == NULL)
    {
        temp->prev = NULL;
        start = temp;
    }
    else
    {
        s = start;
        while (s->next != NULL)
            s = s->next;
        s->next = temp;
        temp->prev = s;
    }
}

/*
* الاضافة في بداية القائمة *
 */
void double_llist::add_begin(int value)
{
```

```
   if (start == NULL)
   {
      cout<<"First Create the list."<<endl;
      return;
   }
   struct node *temp;
   temp = new(struct node) ;
   temp->prev = NULL;
   temp->info = value;
   temp->next = start;
   start->prev = temp;
   start = temp;
   cout<<"Element Inserted"<<endl;
}

/*
* اضافة عنصر في موقع محدد في القائمة *
 */
void double_llist::add_after(int value, int pos(
{
   if (start == NULL)
   {
      cout<<"First Create the list."<<endl;
```

```
   return;
}
struct node  *tmp, *q ;
int i;
q = start;
for (i = 0;i < pos - 1; i+)+
{
   q = q->next;
   if (q == NULL)
   {
      cout<<"There are less than " ;
      cout<<pos<<" elements."<<endl;
      return;
   }
}
tmp = new(struct node) ;
tmp->info = value;
if (q->next == NULL)
{
   q->next = tmp;
   tmp->next = NULL;
   tmp->prev = q      ;
```

```
  }
   else
   {
      tmp->next = q->next;
      tmp->next->prev = tmp;
      q->next = tmp;
      tmp->prev = q;
   }
   cout<<"Element Inserted"<<endl;
}
/*
* حذف عنصر من القائمة
 */
void double_llist::delete_element (int value)
{
   struct node *tmp, *q ;
*/   first element deletion/*
   if (start->info == value)
   {
      tmp = start;
      start = start->next  ;
      start->prev = NULL;
      cout<<"Element Deleted"<<endl;
```

```
        free(tmp) ;
        return;
    }
    q = start;
    while (q->next->next != NULL)
    {
/ *  حذف عنصر بين عقدتين  * /
        if (q->next->info == value)
        {
            tmp = q->next;
            q->next = tmp->next;
            tmp->next->prev = q;
            cout<<"Element Deleted"<<endl;
            free(tmp) ;
            return;
        }
        q = q->next;
    }
/ *   حذف العنصر الاخير   * /
    if (q->next->info == value)
    {
        tmp = q->next;
```

```
      free (tmp) ;
      q->next = NULL;
      cout<<"Element Deleted"<<endl;
      return;
   }
   cout<<"Element "<<value<<" not found"<<endl;
}

/*
* عرض عناصر القائمة الموصولة الثنائية *
 */
void double_llist::display_dlist ( )
{
   struct node *q;
   if (start == NULL)
   {
      cout<<"List empty,nothing to display"<<endl;
      return;
   }
   q = start;
   cout<<"The Doubly Link List is :"<<endl;
   while (q != NULL)
   {
```

```
      cout<<q->info>>" <-> " ;
      q = q->next;
   }
   cout<<"NULL"<<endl;
}

/*
* حساب عدد العناصر في القائمة الموصولة الثنائية *
 */
void double_llist::count ( )
{
   struct node *q = start;
   int cnt = 0;
   while (q != NULL)
   {
      q = q->next;
      cnt++ ;
   }
   cout<<"Number of elements are: "<<cnt<<endl;
}
/*
* عكس القائمة الموصولة الثنائية *
```

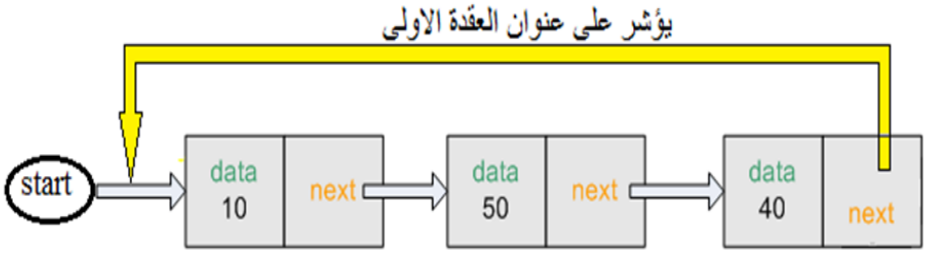
```
 */
void double_llist::reverse( )
{
   struct node *p1, *p2;
   p1 = start;
   p2 = p1->next;
   p1->next = NULL;
   p1->prev = p2;
   while (p2 != NULL)
   {
      p2->prev = p2->next;
      p2->next = p1;
      p1 = p2;
      p2 = p2->prev ;
   }
   start = p1;
   cout<<"List Reversed"<<endl ;
}
```

ملاحظة: عملية تحدث بيانات عقدة في القوائم الموصولة الثنائية لاتختلف عن عملية تحدث بيانات عقدة في القوائم الموصولة الاحادية.

6.11 القوائم الموصولة الدائرية Circular Linked Lists

القوائم الموصولة الدائرية هي مشابهة للقوائم الموصولة الاعتيادية الاحادية ماعدا ان مؤشر العقدة الاخيرة او العنصر الاخير يؤشر الى العنصر الاول في القائمة (في القوائم الاحادية كان يؤشر الى null).

يجب عليك ان تتسائل.. لماذا يريد اي شخص لعمل شيء مثل هذا ؟ حسنا ..هل تعلم ان القوائم الموصولة الدائرية تستخدم غالبا في حالات متعددة ولها تطبيقات عديدة, على سبيل المثال هي من الممكن ان تستخدم في الاعلانات الالكترونية حيث ان كل اعلان يضاف الى القائمة ويتم عرضه بعد اخر اعلان تم عرضه, وبذلك فان القائمة ستعرض بشكل الي اول اعلان في القائمة. في العقد الدائرية من الممكن ان يكون هناك مؤشرين يتم حفظهما وهما مؤشر العقدة الاولى ومؤشر العقدة الاخيرة.



شكل 6.46: قائمة موصولة ثنائية

هناك عدد من العمليات التي من الممكن اجرائها على القوائم الموصولة الدائرية كما في حالة القوائم الموصولة الاحادية (في كل العمليات التي نروم انجازها على القوائم الموصولة الثنائية علينا اولا ان نتأكد ان القائمة ليست فارغة وهذا ينجز

بفحص المؤشر last فاذا كان يؤشر الى null فان القائمة فارغة وبخلافة القائمة ليست فارغة).

6.11.1 العمليات التي تنجز على القوائم الدائرية

لاتختلف القوائم الدائرية عن باقي انواع القوائم الموصولة من حيث العمليات التي تنجز على القائمة, فقط هناك اختلاف بسيط بعملية التنفيذ وشفرة البرنامج, بشكل عام فان العمليات التي تنفذ على القوائم الدائرية هي:

A. خلق عقدة.
B. اضافة عقدة في نهاية القائمة.
C. اضافة عقدة في بداية القائمة.
D. اضافة عقدة بين عقدتين.
E. حذف عقدة.
F. عرض القائمة.
G. تحديث القائمة.
H. ترتيب عناصر القائمة.

A. خلق عقدة

يتم ابتداء خلق عقدة جديدة ويوضع لها اسم او مؤشر باسم معين يؤشر على هذه العقدة وهو يمثل اسم هذه العقدة (نفرض ان اسم المؤشر هو temp).

تضاف البيانات المحددة لهذه العقدة, تضاف الى حقل البيانات.

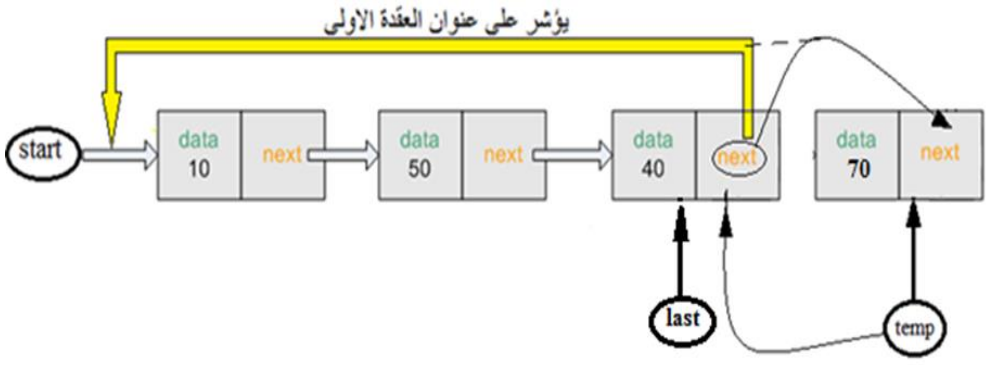
يتم فحص المؤشر last (الذي يمثل عنوان العقدة الاخيرة المسماة last) فاذا كان يؤشر على null (هذا يعني عدم وجود عقد في القائمة اي ان القائمة فارغة) عندها نجعل اسم العقدة التي تم خلقها في حقل عنوان العقدة اللاحقة للعقدة التي تم خلقها (اي انها ستؤشر على نفسها), ويعاد تسمية last بنفس اسم العقدة الاولى (اي ان start & last سيشيران الى نفس العنوان).

B. اضافة عقدة الى نهاية القائمة الموصولة الدائرية

في هذه الحالة يتم خلق عقدة جديدة كما اسلفنا وتسند البيانات في حقل البيانات للعقدة الجديدة.

بعد وضع البيانات يتم تغيير عنوان العقدة اللاحقة للعقدة التي تم خلقها بحيث يسند له ذات العنوان الموجود في حقل عنوان العقدة الاخيرة في القائمة (طالما هذه العقدة الجديدة ستكون الاخيرة لذلك لابد ان تؤشر على العقدة الاولى كما كانت العقدة الاخيرة في القائمة تؤشر).

الخطوة التالية ان يتم ربط العقدة التي هي الان اخيرة مع العقدة الجديدة لكي تكون الجديدة هي الاخيرة.



شكل 6.47: اضافة عقدة في نهاية القائمة مع توضيح لكيفية وضع العناوين وربط العقدة.

```
temp->next = last->next;
```

الان سنجعل حقل العنوان في العقدة التي هي الاخيرة قبل الاضافة يؤشر على العقدة التي تم خلقها وهي العقدة temp

```
last -> next = temp ;
```

اخيرا يتم تغير اسم العقدة التي تم خلقها لتكون باسم last بدلا من temp

```
last = temp;
```

ملاحظة: من الممكن عدم استخدام المؤشر last, في هذه الحالة نضع مؤشر على العقدة الاولى ونحركه ليصل العقدة الاخيرة (العقدة الاخيرة هي العقدة التي يكون فيها مؤشر العقدة اللاحقة يساوي عنوان العقدة الاولى وليس null كما سبق في الانواع الاخرى).

شكل 6.48: القائمة بعد اضافة العقدة.

C. اضافة عقدة في البداية

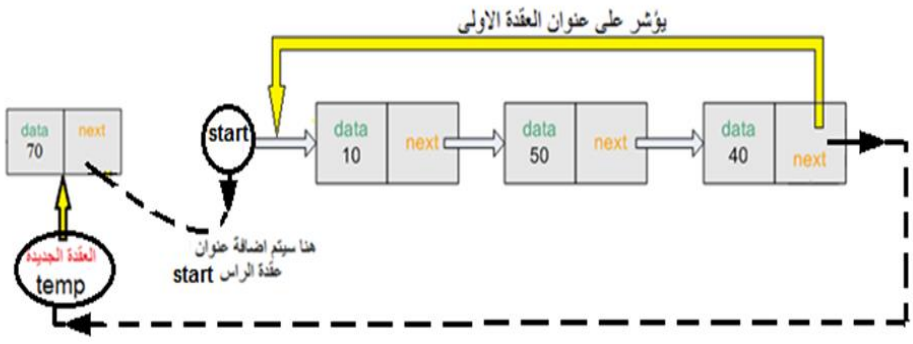
يتم خلق عقدة جديدة, واسناد قيمة الى حقل البيانـات فـي العقدة الجديـدة. ربـط العقدة التي تم خلقها الى بداية القائمة.

تتم عملية الربط بوضع عنوان العقدة الاولى في حقل العنـوان للعقدة التـي تـم خلقهـا (لان العقدة الاولى فـي القائمـة ستكون العقدة الثانيـة بعد الاضـافة), وبـذلك ستكون هـذه العقـدة التـي تـم خلقهـا سـابقة للعقـدة الاولـى فـي القائمـة اي سـتكون هـي العقـدة الاولى وكما يلي

```
temp->next = last->next ;
```

OR

```
temp -> next = start ;
```

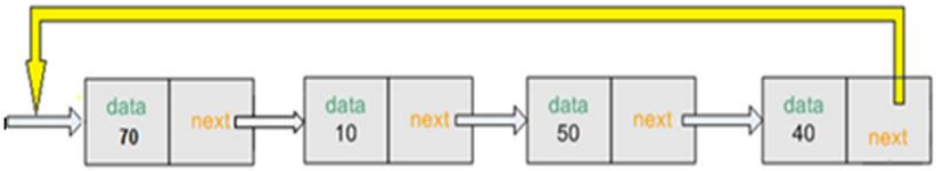


شكل 6.49: طريقة ربط العقدة في بداية القائمة وتغيير العناوين.

لاحظ هنا (شكل 6.49) ان عنوان العقدة الاولى هو موجود في حقل العنوان للعقدة الاخيرة last, لذلك من الممكن الحصول عليه من هناك واسناده الى العقدة التي تم خلقها كما وضحنا اعلاه واعادة تسمية العقدة المضافة بحيث تكون هي العقدة الاولى (start), ومن الممكن ايضا استخدامه بشكل مباشر من العقدة الاولى (start).

الخطوة الاخيرة هي جعل مؤشر العقدة الاخيرة يؤشر على العقدة التي وضعناها في البداية, واعادة تسمية العقدة الجديدة باسم عقدة البداية, وكما يلي

```
last->next = temp ;

start = temp ;
```



شكل 6.50: القائمة بعد الاضافة.

D. اضافة عقدة في موقع معين

وهي لا تختلف عن طريقة الاضافة في القوائم الاحادية, لذلك سوف لا نتطرق لها ومن الممكن الرجوع الى اضافة عقدة بين عقدتين في القوائم الاحادية.

E. حذف عقدة تحتوي على عنصر معين

اذا لم تكن القائمة فارغة فان هناك اكثر من احتمال. طبعا لابد من وضع مؤشر على العقدة الاولى, ليكن اسمه مثلا (S) لكل الحالات:

a) **الحالة الاولى**: هو ان تكون القائمة متكونة من عقدة واحدة فقط وهذا من الممكن فحصة وذلك بان يكون عنوان العقدة اللاحقة هو ذاته عنوان العقدة ولها قيمة في حقل القيمة مساوي للقيمة المطلوبة اذا تحقق هذين الشرطين فهذا يعني وجود عقدة واحدة وسيتم حذفها لنحصل على قائمة فارغة.

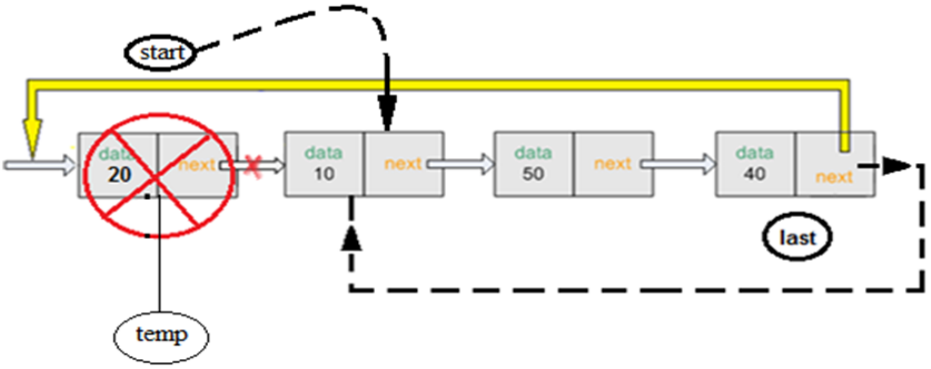
حذف هذه العقدة يكون بحفظ عنوان العقدة بمتغير, ثم حذف العقدة بمساواتها الى null ومن بعدها تحرير (المساحة) التي كانت تشغلها هذه العقدة وكما يلي

```
temp = last;
last = null;
free (temp) ;
```

b) **الحالة الثانية**: هو ان القيمة المطلوبة موجودة في العقدة الاولى لقائمة تحتوي على اكثر من عقدة, عملية الحذف تتم كما يلي:

شكل 6.51: حذف عقدة من قائمة موصولة دائرية.

بداية لابد من حفظ عنوان العقدة الاولى بمتغير من نوع مؤشر temp.



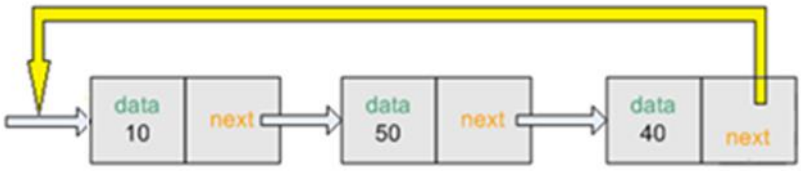
شكل 6.52: كيفية تغيير العناوين عند حذف عقدة.

يتم وضع عنوان العقدة الثانية وهو العنوان الذي يوجد في حقل العنوان للعقدة الاولى في حقل العنوان للعقدة الاخير (شكل 6.52).

تسمية العقدة الثانية باسم عقدة البداية start.

يتم تحرير المساحة الخاصة بالعقدة الاولى temp وكما يلي

```
    last->next = temp -> next;
    temp -> next = start ;
free (temp) ;
```



شكل 6.53: القائمة بعد الحذف.

c) **الحالة الثالثة:** هو حذف عقدة تكون وسط القائمة وهذا يتم بنفس الطريقة التي استخدمت بحذف عقدة في وسط قائمة موصولة احادية.

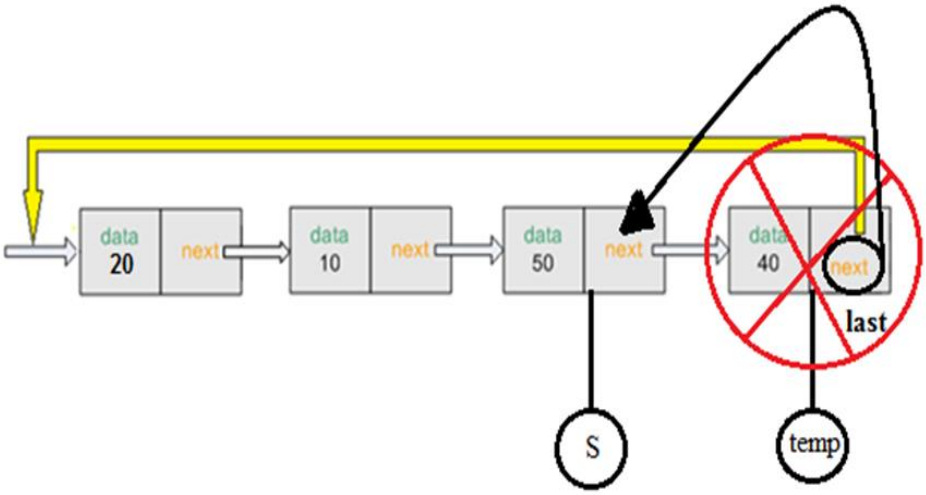
d) **الحالـة الرابعـة**: هـو ان تكـون العقدة المطلـوب حـذفها هـي العقدة الاخيـرة عندها سيتم وضع مؤشرين الاول على العقدة الاولى (S) والثاني على العقدة الثانية (temp) ويتم تحريكهم بواقع عقدة واحدة في كل مرة على ان يتم المحافظة على تموضع المؤشرين على عقدتين متجاورتين دائما, يتم التوقف عند وصول المؤشر الثاني (temp) الى العقدة النهائية.

يتم تغيير العناوين بما يضمن حذف العقدة الاخيرة وذلك بوضع عنوان العقدة الاولى في حقل العنوان للعقدة السابقة للعقدة الاخيرة والتي يؤشر عليها المؤشر الاول (S).

يعاد تسمية العقدة التي يؤشر عليها المؤشر الاول باسم last

تحرير مساحة العقدة المحذوفة

```
s->next = last->next;
last = s;
free (temp);
```



شكل 6.54: حذف العقدة الاخيرة وتوضيح كيفية تغيير العناوين.

F. عرض القائمة

يوضع مؤشر على العقدة الاولى ثم يتم نقله الى العقدة المجاورة وهكذا لحين الوصول الى نهاية القائمة.

هذه العملية تتم باستخدام حلقة تكرار تبدا من اول عقدة وتنتهي باخر عقدة.

في كل عقدة يتم المرور عليها يتم عرض القيمة الموجودة في حقل البيانات لتلك العقدة.

G. تحديث القائمة

عملية التحديث هي عملية تغيير القيمة التي في حقل البيانات لعقدة ما في موقع معين داخل القائمة.

لإنجاز ذلك نضع مؤشر على اول عقدة.

نعمل حلقة تكرار من اول عقدة الى الموقع المحدد.

ننقل المؤشر في كل مرة الى العقدة اللاحقة.

عند الوصول الى الموقع المطلوب يتم تغيير القيمة في حقل البيانات لتلك العقدة.

اذا وصل المؤشر الى العقدة الاخيرة قبل ان يتم ايجاد العقدة المطلوبة, عندها يتم عرض رسالة توضح ان الموقع المطلوب خارج حجم القائمة.

H. ترتيب القائمة

نتبع نفس الطريقة في ترتيب القوائم الاحادية.

❖ برنامج كامل لجميع العمليات التي تجرى على القوائم الموصولة الدائرية

```
/*
 * C++ Program to Implement Circular Linked List
 */
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
/*
 *  الاعلان عن العقدة
 */
struct node
{
   int info;
   struct node *next;
}*last;

/*
 * الاعلان عن الصنف
 */
class circular_llist
```

```
{
   public:
      void create_node(int value);
      void add_begin(int value);
      void add_after(int value, int position);
      void delete_element(int value);
      void search_element(int value);
      void display_list( );
      void update();
      void sort();
      circular_llist( )
      {
         last = NULL;
      }
};

int main()
{
   int choice, element, position;
   circular_llist cl;
   while (1)
   {
      cout<<endl<<"--------------------------"<<endl;
```

```
cout<<endl<<"Circular singly linked list"<<endl;
cout<<endl<<"--------------------------"<<endl;
cout<<"1.Create Node"<<endl;
cout<<"2.Add at beginning"<<endl;
cout<<"3.Add after"<<endl;
cout<<"4.Delete"<<endl;
cout<<"5.Search"<<endl;
cout<<"6.Display"<<endl;
cout<<"7.Update"<<endl;
cout<<"8.Sort"<<endl;
cout<<"9.Quit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
   cout<<"Enter the element: ";
   cin>>element;
   cl.create_node(element);
   cout<<endl;
   break;
case 2:
```

```
    cout<<"Enter the element: ";
    cin>>element;
    cl.add_begin(element);
    cout<<endl;
    break;
case 3:
    cout<<"Enter the element: ";
    cin>>element;
    cout<<"Insert element after position: ";
    cin>>position;
    cl.add_after(element, position);
    cout<<endl;
    break;
case 4:
    if (last == NULL)
    {
        cout<<"List is empty, nothing to delete"<<endl;
        break;
    }
    cout<<"Enter the element for deletion: ";
    cin>>element;
    cl.delete_element(element);
    cout<<endl;
```

```
    break;
case 5:
    if (last == NULL)
    {
        cout<<"List Empty!! Can't search"<<endl;
        break;
    }
    cout<<"Enter the element to be searched: ";
    cin>>element;
    cl.search_element(element);
    cout<<endl;
    break;
case 6:
    cl.display_list();
    break;
case 7:
    cl.update();
    break;
case 8:
    cl.sort();
    break;
case 9:
```

```
            exit(1);
            break;
        default:
            cout<<"Wrong choice"<<endl;
        }
    }
    return 0;
}

/*
 * خلق قائمة موصولة
 */
void circular_llist::create_node(int value)
{
    struct node *temp;
    temp = new(struct node);
    temp->info = value;
    if (last == NULL)
    {
        last = temp;
        temp->next = last;
    }
    else
```

```
    {
        temp->next = last->next;
        last->next = temp;
        last = temp;
    }
}

/*
 * اضافة عنصر الى بداية القائمة
 */
void circular_llist::add_begin(int value)
{
    if (last == NULL)
    {
        cout<<"First Create the list."<<endl;
        return;
    }
    struct node *temp;
    temp = new(struct node);
    temp->info = value;
    temp->next = last->next;
    last->next = temp;
```

```
}

/*
 * اضافة عنصر في موقع محدد
 */
void circular_llist::add_after(int value, int pos)
{
    if (last == NULL)
    {
        cout<<"First Create the list."<<endl;
        return;
    }
    struct node *temp, *s;
    s = last->next;
    for (int i = 0;i < pos-1;i++)
    {
        s = s->next;
        if (s == last->next)
        {
            cout<<"There are less than ";
            cout<<pos<<" in the list"<<endl;
            return;
        }
```

```
    }
    temp = new(struct node);
    temp->next = s->next;
    temp->info = value;
    s->next = temp;
    /*اضافة عنصر الى نهاية القائمة*/
    if (s == last)
    {
       last=temp;
    }
}

/*
 * حذف عنصر من القائمة
 */
void circular_llist::delete_element(int value)
{
    struct node *temp, *s;
    s = last->next;
     /*اذا كانت القائمة تحتوي فقط عنصر واحد */
    if (last->next == last && last->info == value)
    {
```

```
    temp = last;
    last = NULL;
    free(temp);
    return;
}
if (s->info == value)  /*حذف العنصر الاول*/
{
    temp = s;
    last->next = s->next;
    free(temp);
    return;
}
while (s->next != last)
{
    /*حذف عنصر بين عنصرين*/
    if (s->next->info == value)
    {
        temp = s->next;
        s->next = temp->next;
        free(temp);
        cout<<"Element "<<value;
        cout<<" deleted from the list"<<endl;
        return;
```

```
        }
        s = s->next;
    }
    /*حذف العنصر الاخير*/
    if (s->next->info == value)
    {
        temp = s->next;
        s->next = last->next;
        free(temp);
        last = s;
        return;
    }
    cout<<"Element "<<value<<" not found in the list"<<endl;
}
 /*
 * البحث عن عنصر في القائمة
 */
void circular_llist::search_element(int value)
{
    struct node *s;
    int counter = 0;
    s = last->next;
```

```
    while (s != last)
    {
        counter++;
        if (s->info == value)
        {
            cout<<"Element "<<value;
            cout<<" found at position "<<counter<<endl;
            return;
        }
        s = s->next;
    }
    if (s->info == value)
    {
        counter++;
        cout<<"Element "<<value;
        cout<<" found at position "<<counter<<endl;
        return;
    }
    cout<<"Element "<<value<<" not found in the list"<<endl;
}

/*
 * عرض محتويات القائمة الموصولة الدائرية
```

```
 */
void circular_llist::display_list()
{
   struct node *s;
   if (last == NULL)
   {
      cout<<"List is empty, nothing to display"<<endl;
      return;
   }
   s = last->next;
   cout<<"Circular Link List: "<<endl;
   while (s != last)
   {
      cout<<s->info<<"->";
      s = s->next;
   }
   cout<<s->info<<endl;
}

/*
 * تحديث القائمة الموصولة الدائرية
 */
```

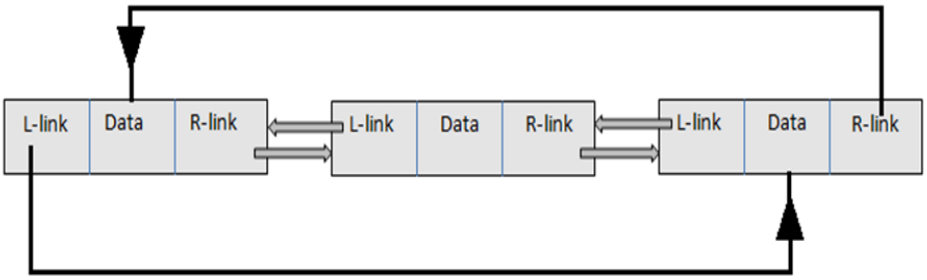
```
void circular_llist::update()
{
   int value, pos, i;
   if (last == NULL)
   {
      cout<<"List is empty, nothing to update"<<endl;
      return;
   }
   cout<<"Enter the node position to be updated: ";
   cin>>pos;
   cout<<"Enter the new value: ";
   cin>>value;
   struct node *s;
   s = last->next;
   for (i = 0;i < pos - 1;i++)
   {
      if (s == last)
      {
         cout<<"There are less than "<<pos<<" elements.";
         cout<<endl;
         return;
      }
      s = s->next;
```

```
    }
    s->info = value;
    cout<<"Node Updated"<<endl;
}

/*
 * ترتيب القائمة الموصولة الدائرية
 */
void circular_llist::sort()
{
    struct node *s, *ptr;
    int temp;
    if (last == NULL)
    {
        cout<<"List is empty, nothing to sort"<<endl;
        return;
    }
    s = last->next;
    while (s != last)
    {
        ptr = s->next;
        while (ptr != last->next)
```

```
        {
            if (ptr != last->next)
            {
                if (s->info > ptr->info)
                {
                    temp = s->info;
                    s->info = ptr->info;
                    ptr->info = temp;
                }
            }
            else
            {
                break;
            }
            ptr = ptr->next;
        }
        s = s->next;
    }
}
```

6.12 القـوائم الموصـولة الدائريــة الثنائيــة Circular Doubly Linked List

تختلف القوائم الموصولة الدائرية الثنائية عن الانواع الاخرى التي سبق وان تم التطرق لها.. حيث ان كل عقدة هنا تحتوي على حقلين للعنوان بدل من حقل واحد, احدهما يوضع به عنوان العقدة السابقة (وهو حقل العنوان الايسر), اما الثاني فهو حقل عنوان العقدة اللاحقة والذي هو (حقل العنوان الايمن).. وبالتالي فان كل عقدة من الممكن ان نعرف من خلالها العقدة السابقة لها والعقدة اللاحقة لها, مع ملاحظة ان العقدة الاولى تشير الى العقدة الاخيرة والعقدة الاخيرة تشير الى العقدة الاولى اذا كانت القائمة تحتوي على اكثر من عقدة.



شكل 6.55: قائمة موصولة دائرية ثنائية.

6.12.1 العمليات التي تجرى على القوائم الموصولة الثنائية الدائرية

هناك عدد من العمليات التي من الممكن اجرائها على القوائم الموصولة الدائرية الثنائية وهي:

A. اضافة عقدة في بداية القائمة الموصولة.
B. اضافة عقدة في نهاية القائمة.
C. اضافة عقدة في موقع محدد ضمن القائمة.
D. حذف عقدة من القائمة.
E. تحديث القائمة.
F. البحث عن عقدة داخل القائمة.
G. الترتيب.

H. عرض محتويات القائمة.

ملاحظة: في كل العمليات اعلاه والتي سيتم توضيحها لاحقا سيتم فرض اسم عقدة البداية start واسم عقدة النهاية last والعقدة التي يتم خلقها نسميها temp.

كذلك في كل العمليات علينا ابتداء ان نفحص القائمة فيما اذا كانت فارغة او لا وتكون القائمة فارغة اذا كانت عقدة البداية تساوي عقدة النهاية (اي لهم نفس الاسم) وعقدة البداية تساوي null.

المتغيرات هنا والتي ندعوها اسماء للعقد (start, last, temp) هي في حقيقتها من نوع مؤشر اي تحمل عنوان.

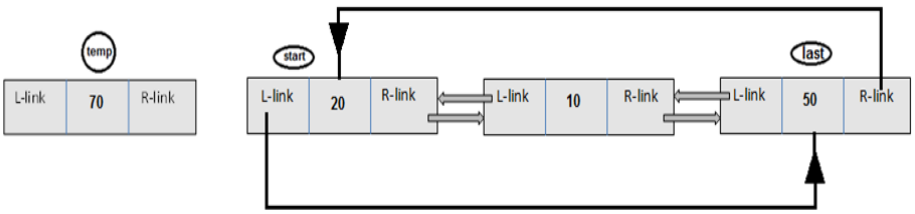
A. اضافة عقدة الى القائمة

لغرض اضافة عقدة في اي مكان من القائمة لابد بداية من خلق عقدة ثم تتم عملية الاضافة, اذن نخلق عقدة جديدة كما سبق وان شرحنا في خلق العقد الثنائية, ونضيف قيمة الى حقل البيانات في العقدة.

هناك عدد من الحالات التي تتم بها الاضافة وهي:

a) اضافة عقدة في بداية القائمة.

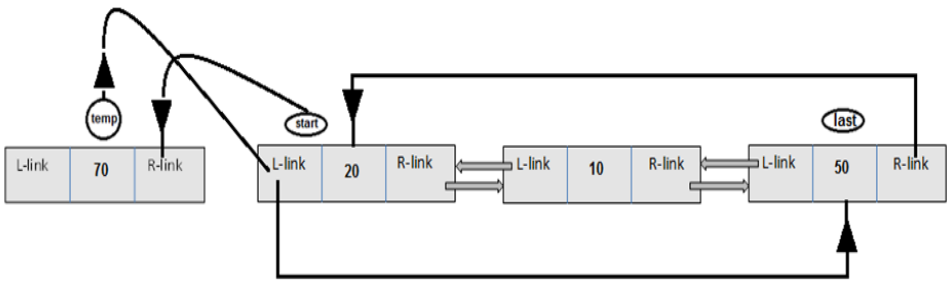
نضيف عنوان العقدة الاولى في القائمة في حقل العنوان الايمن للعقدة المراد اضافتها (نظرا الى ان العقدة ستضاف قبل العقدة الاولى في القائمة لذلك ستكون هي الاولى).



شكل 6.56: اضافة عقدة الى قائمة موصولة ثنائية دائرية.

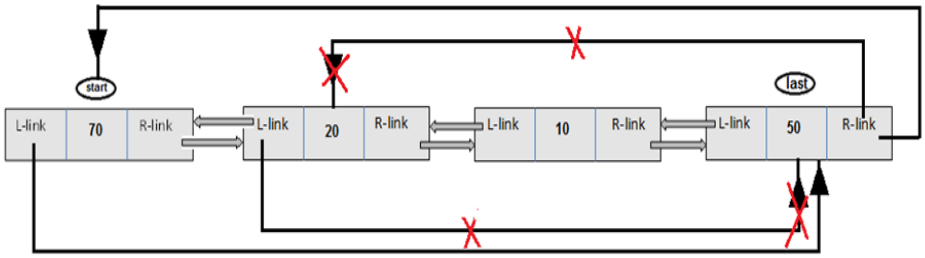
في حقل العنوان الايسر للعقدة الاولى التي في القائمة سنضع عنوان العقدة التي سيتم اضافتها (temp) (لان العقدة المضافة ستضاف قبل العقدة الاولى في القائمة وستكون لاحقا هي العقدة الاولى).

الان بعد ان اصبحت العقدة الاولى تؤشر على العقدة المضافة والعقدة المضافة تؤشر على العقدة الاولى اصبح واضحا ان العقدة المضافة اصبحت هي قبل البداية (العقدة الاولى) لذلك من المناسب ان نسميها العقدة الاولى وذلك بان نضع عنوان العقدة التي سنضيفها في المؤشر start (أي ان يتم تسمية العقدة المضافة باسم start), شكل 6.57.



شكل 6.57: توضيح لكيفية ربط العقدة المضافة في بداية القائمة.

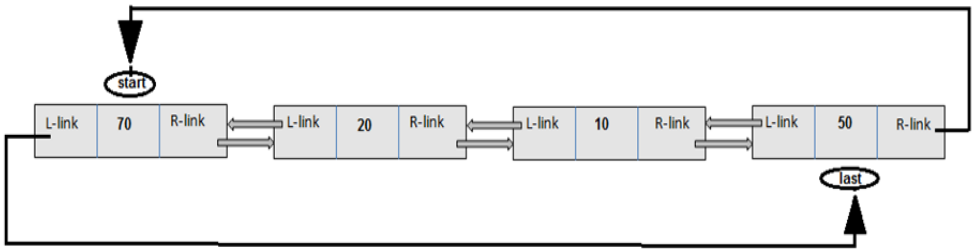
الان بقى لدينا حقل العقدة السابقة في العقدة المضافة (حقل العنوان الايسر) ليس فيه عنوان, ولما كانت القائمة دائرية فسنضع عنوان العقدة الاخيرة في حقل العنوان الايسر للعقدة المضافة.



شكل 6.58: شكل توضيحي لكيفية ربط العقدة الاولى مع الاخيرة.

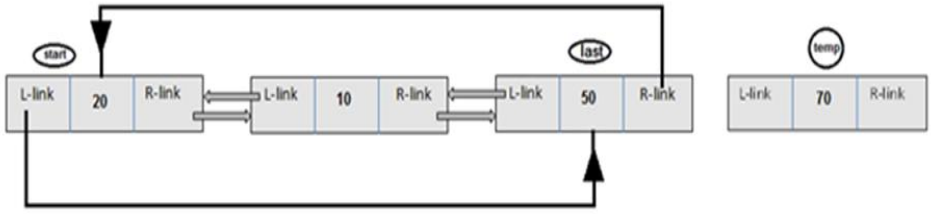
كذلك فان العقدة الاخيرة كانت تشير في حقل عنوان العقدة اللاحقة الى العقدة الاولى التي اصبحت الان ثانية بعد الاضافة, لذلك يجب تحديثه وذلك بإضافة عنوان العقدة المضافة التي هي الان اولى وغيرنا اسمها الى العقدة الاولى في حقل العنوان الايمن للعقدة الاخيرة (لان القائمة دائرية).

```
temp->next = start;
start->prev = temp;
start = temp;
start->prev = last;
last->next = start;
```



شكل 6.59: القائمة بعد الاضافة.

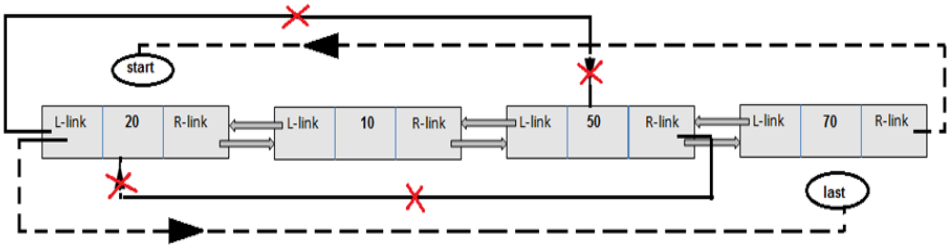
b) اضافة عقدة في نهاية القائمة



شكل 6.60: اضافة عقدة الى نهاية القائمة الموصولة الثنائية الدائرية.

ان العمل الاساس هو اضافة عناوين في حقل العناوين للعقدة المضافة وحقل العنوان الايمن للعقدة الاخيرة في القائمة وكذلك حقل العنوان الايسر للعقدة الاولى, مع ملاحظة ان العقدة الاخيرة ستكون العقدة ما قبل الاخيرة عند اضافة عقدة بعدها (لان العقدة المضافة ستكون هي الاخيرة) لذلك فان هذه العقدة التي كانت هي الاخيرة ستشير في حقل العنوان الايمن الى العقدة المضافة وليس الى العقدة الاولى كما كانت قبل الاضافة.

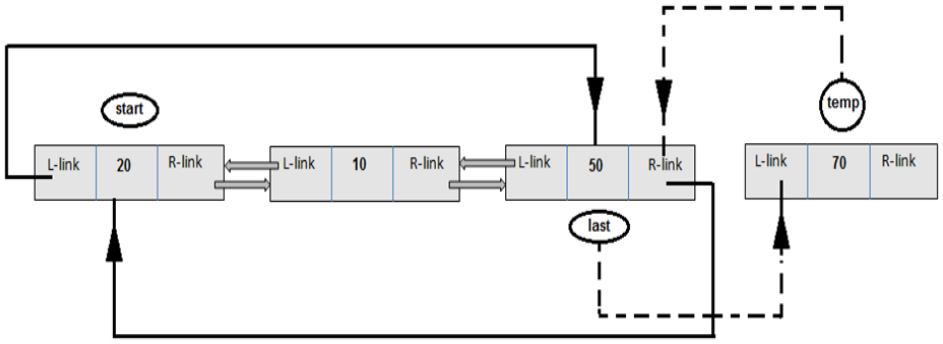
بالنسبة للعقدة المضافة سنضيف عنوان العقدة اللاحقة في حقل العنوان الايمن والتي هي العقدة الاولى نظرا لان هذه العقدة ستصبح اخر عقدة في القائمة الدائرية.



شكل 6.61: توضيح كيفية ربط العقدة المضافة مع العقدة الاولى والعقدة الاولى معها.

في حقل العنوان الايسر للعقدة المضافة سنضيف عنوان العقدة التي كانت هي اخر عقدة لانها ستكون سابقة للعقدة المضافة.

يتم ابدال اسم العقدة المضافة الى العقدة الاخيرة (last) لانها اصبحت هي العقدة الاخيرة.

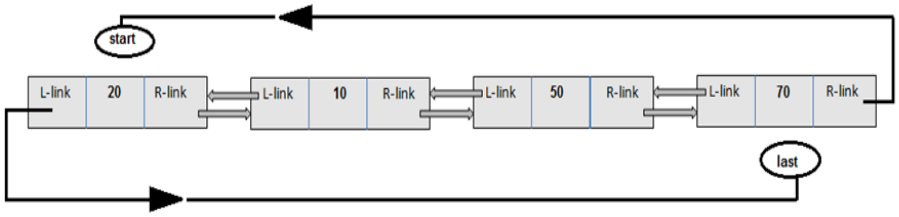


شكل 6.62: توضيح ربط العقدة الاخيرة مع العقدة المضافة.

بالنسبة للعقدة الاولى فانها كانت تشير الى عنوان العقدة الاخيرة في حقل العنوان الايسر, ولما كانت العقدة الاخيرة قد تم تغييرها لذلك يجب تحديث حقل العنوان الايسر للعقدة الاولى ليشير الى العقدة المضافة التي هي اصبحت العقدة الاخيرة.

```
last->next = temp;
temp->prev = last;
last = temp;
start->prev = last;
```

last->next = start;



شكل 6.63: القائمة بعد الاضافة.

c) اضافة عقدة في موقع محدد

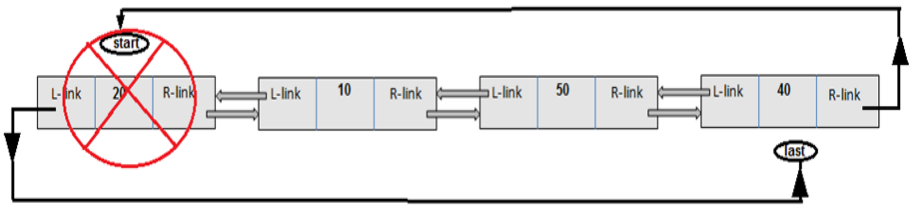
نتبع نفس خطوات الاضافة لعقدة وسط القائمة الثنائية.

B. حذف عقدة في موقع محدد

يجب تحديد الموقع المراد حذف العقدة منه ونتاكد من ان هذا الموقع هو ضمن حدود او اقل من عدد العقد في القائمة.

الان نحذف العقدة حسب الاحتمالات التالية

a) اذا كان موقع الحذف هو واحد اي انه العقدة الاولى نعمل ما يلي



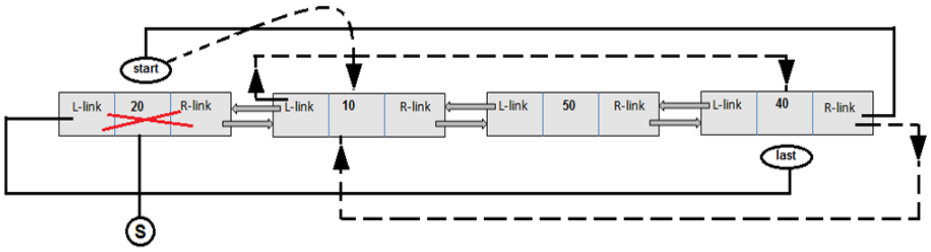
شكل 6.64: حذف العقدة الاولى في قائمة ثنائية دائرية.

بداية يتم حفظ عنوان العقدة الاولى بمتغير من نوع مؤشر ليكن S.

نغير عنوان العقدة اللاحقة في حقل العنوان الايمن للعقدة الاخيرة ليشير الى العقدة الثانية في القائمة بدلا من الاولى التي سيتم حذفها (عنوان العقدة الثانية موجود في حقل العنوان الايمن للعقدة الاولى).

نضع عنوان العقدة الاخيرة في حقل العنوان الايسر للعقدة الثانية (وبهذا ستكون العقدة الاولى خارج القائمة). يجب ان نغير تسمية العقدة الثانية لتكون العقدة الاولى start.

نحرر مساحة العقدة الاولى التي تم ازالتها والتي تم حفظ عنوانها بالمتغير S.

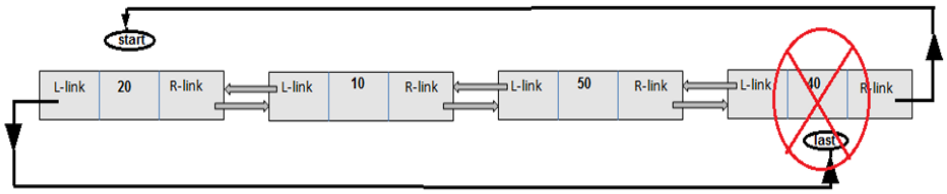


شكل 6.65: تغيير العناوين لغرض حذف العقدة الاولى

```
S = start;
last->next = S->next;
S->next-> prev = last;
start = S->next;
free (S);
```

b) اذا كانت العقدة المراد حذفها هي ليست العقدة الاولى وانما عقدة في موقع معين داخل القائمة نتبع ذات الخطوات التي تم توضيحها في حذف عقدة بين عقدتين في القوائم الثنائية.

c) اذا كانت العقدة المراد حذفها هي العقدة الاخيرة عندها نتبع الخطوات التالية



شكل 6.66: حذف العقدة الاخيرة في قائمة موصولة ثنائية دائرية.

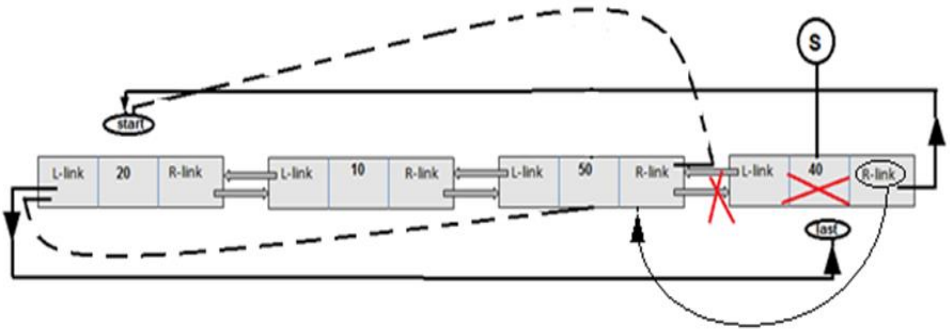
نحرك مؤشر واحد من عقدة البداية لحين الوصول الى العقدة الاخيرة (نفرض اسمه S).

يتم ربط العقدة التي تسبق العقدة الاخيرة بالعقدة الاولى وبذلك ستصبح العقدة الاخيرة التي يؤشر عليها المؤشر S خارج القائمة (في القوائم الثنائية عنوان العقدة السابقة لأي عقدة يكون موجود في حقل العنوان الايسر).

يعاد تسمية العقدة التي تسبق العقدة الاخيرة باسم last.

يتم تحرير المساحة للعقدة المحذوفة.

ملاحظة: اذا كنا نعلم مسبقا ان المطلوب هو حذف العقدة الاخيرة فمن الممكن انجاز ذلك دون الحاجة لتمرير مؤشرين وذلك بالاعتماد على عنوان العقدة الاخيرة last .



شكل 6.67: شكل توضيحي لكيفية ربط العقدة السابقة للعقدة الاخيرة مع العقدة الاولى لغرض حذف العقدة الاخيرة.

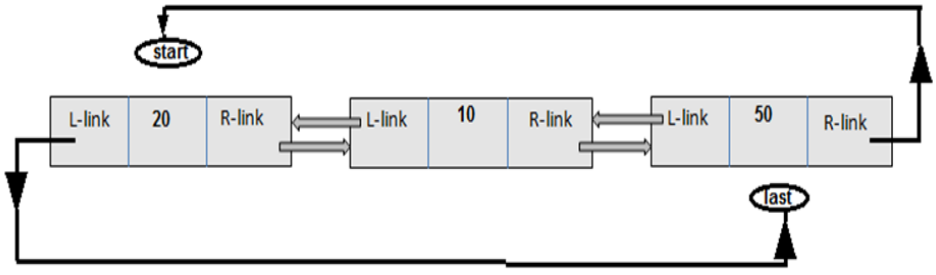
من الممكن وضع عنوان العقدة الاولى مباشرة في حقل العنوان الايسر للعقدة قبل الاخيرة كما واضح من الخط المنقط اعلى الشكل 6.67.

S -> prev -> next = start ;

ومن الممكن ايضا وضع العنوان الايمن في العقدة الاخيرة (وهو يحمل عنوان العقدة الاولى) في حقل العنوان الايمن للعقدة قبل الاخيرة كما واضح من الخط المتصل اسفل الشكل 6.67.

S -> prev -> next = S -> next ;

S -> prev = last ;



الشكل 6.68: القائمة بعد الحذف.

C. التحديث

نحدد الموقع المراد تحديث بياناته.

نحدد قيمة البيانات المراد ادخالها في حقل البيانات للعقدة المطلوب تحديث بياناتها.

اذا كان الموقع هو الموقع الاول (بمعنى العقدة الاولى) يتم تحديثها, بخلاف ذلك نضع مؤشر على العقدة الاولى ونحركه الى نهاية القائمة وفي كل مرة من البداية الى النهاية نفحص البيانات للعقدة التي يؤشر عليها المؤشر فاذا كانت هي العقدة المطلوبة يتم تحديث بياناتها والخروج من حلقة التكرار.

```
S = start;
do
{
 if (S -> data == value)
 {  S -> data = new_value ;
  cout<<"Node Updated"<<endl;
  break ;
  }
 S = S -> next ;
}   while (S != start) ;
```

D. عرض محتويات القائمة

نضع مؤشر على العقدة الاولى. نعمل حلقة تكرار من العقدة الاولى الى العقدة الاخيرة.

في كل دورة يتم اظهر او طباعة البيانات في حقل البيانات للعقدة التي يؤشر عليها المؤشر. وفي كل دورة يتم نقل المؤشر الى العقدة اللاحقة.

```
        S = start;
  while ( S != last)
   {
      S = S -> next;
      cout<< S -> data << "<->" ;
   }
   cout<< S-> data << endl ;
```

❖ البرنامج الكامل لجميع العمليات التي تجرى على القوائم الموصولة الثنائية

```
/*
 * C++ Program to Implement Circular Doubly Linked List
 */
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;

/*
```

```
 * الاعلان عن العقدة *
 */
struct node
{
   int info;
   struct node *next;
   struct node *prev;
}*start, *last;
int counter = 0;
/*
 * الاعلان عن الصنف *
 */
class double_clist
{
   public:
      node *create_node (int);
      void insert_begin();
      void insert_last();
      void insert_pos();
      void delete_pos();
      void search();
      void update();
```

```
    void display();
    void reverse();
    void sort();
    double_clist( )
    {
       start = NULL;
       last = NULL;
    }
};

int main()
{
   int choice;
   double_clist cdl;
   while (1)
   {
      cout<<"\n--------------------------------"<<endl;
      cout<<"Operations on Doubly Circular linked list"<<endl;
      cout<<"\n--------------------------------"<<endl;
      cout<<"1.Insert at Beginning"<<endl;
      cout<<"2.Insert at Last"<<endl;
      cout<<"3.Insert at Position"<<endl;
      cout<<"4.Delete at Position"<<endl;
```

```
cout<<"5.Update Node"<<endl;
cout<<"6.Search Element"<<endl;
cout<<"7.Sort"<<endl;
cout<<"8.Display List"<<endl;
cout<<"9.Reverse List"<<endl;
cout<<"10.Exit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
   cdl.insert_begin();
   break;
case 2:
   cdl.insert_last();
   break;
case 3:
   cdl.insert_pos();
   break;
case 4:
   cdl.delete_pos();
   break;
```

```
        case 5:
            cdl.update();
            break;
        case 6:
            cdl.search();
            break;
        case 7:
            cdl.sort();
            break;
        case 8:
            cdl.display();
            break;
        case 9:
            cdl.reverse();
            break;
        case 10:
            exit(1);
        default:
            cout<<"Wrong choice"<<endl;
        }
    }
    return 0;
}
```

```
/*
 *  تخصيص الذاكرة للعقد بشكل الي
 */
node* double_clist::create_node(int value)
{
   counter++;
   struct node *temp;
   temp = new(struct node);
   temp->info = value;
   temp->next = NULL;
   temp->prev = NULL;
   return temp;
}
/*
 *  اضافة عنصر في البداية
 */
void double_clist::insert_begin()
{
   int value;
   cout<<endl<<"Enter the element to be inserted: ";
   cin>>value;
```

```
    struct node *temp;
    temp = create_node (value);
    if (start == last && start == NULL)
    {
        cout<<"Element inserted in empty list"<<endl;
        start = last = temp;
        start->next = last->next = NULL;
        start->prev = last->prev = NULL;
    }
    else
    {
        temp->next = start;
        start->prev = temp;
        start = temp;
        start->prev = last;
        last->next = start;
        cout<<"Element inserted"<<endl;
    }
}

/*
 * اضافة عنصر في نهاية القائمة *
 */
```

```
void double_clist::insert_last()
{
   int value;
   cout<<endl<<"Enter the element to be inserted: ";
   cin>>value;
   struct node *temp;
   temp = create_node(value);
   if (start == last && start == NULL)
   {
      cout<<"Element inserted in empty list"<<endl;
      start = last = temp;
      start->next = last->next = NULL;
      start->prev = last->prev = NULL;
   }
   else
   {
      last->next = temp;
      temp->prev = last;
      last = temp;
      start->prev = last;
      last->next = start;
   }
```

```
}
/*
 * اضافة عنصر في موقع داخل القائمة *
 */
void double_clist::insert_pos()
{
   int value, pos, i;
   cout<<endl<<"Enter the element to be inserted: ";
   cin>>value;
   cout<<endl<<"Enter the postion of element inserted: ";
   cin>>pos;
   struct node *temp, *s, *ptr;
   temp = create_node(value);
   if (start == last && start == NULL)
   {
      if (pos == 1)
      {
         start = last = temp;
         start->next = last->next = NULL;
         start->prev = last->prev = NULL;
      }
      else
      {
```

```
        cout<<"Position out of range"<<endl;
        counter--;
        return;
    }
}
else
{
    if (counter < pos)
    {
        cout<<"Position out of range"<<endl;
        counter--;
        return;
    }
    s = start;
    for (i = 1;i <= counter;i++)
    {
        ptr = s;
        s = s->next;
        if (i == pos - 1)
        {
            ptr->next = temp;
            temp->prev = ptr;
```

```
            temp->next = s;
            s->prev = temp;
            cout<<"Element inserted"<<endl;
            break;
          }
        }
    }
}
/*
 * حذف عقدة من موقع محدد في القائمة
 */
void double_clist::delete_pos()
{
    int pos, i;
    node *ptr, *s;
    if (start == last && start == NULL)
    {
        cout<<"List is empty, nothing to delete"<<endl;
        return;
    }
    cout<<endl<<"Enter the postion of element to be deleted: ";
    cin>>pos;
    if (counter < pos)
```

```
{
   cout<<"Position out of range"<<endl;
   return;
}
s = start;
if(pos == 1)
{
   counter--;
   last->next = s->next;
   s->next->prev = last;
   start = s->next;
   free(s);
   cout<<"Element Deleted"<<endl;
   return;
}
for (i = 0;i < pos - 1;i++ )
{
   s = s->next;
   ptr = s->prev;
}
ptr->next = s->next;
s->next->prev = ptr;
```

```
    if (pos == counter)
    {
        last = ptr;
    }
    counter--;
    free(s);
    cout<<"Element Deleted"<<endl;
}
/*
 * تحديث قيمة او بيانات عقدة محددة
 */
void double_clist::update()
{
    int value, i, pos;
    if (start == last && start == NULL)
    {
        cout<<"The List is empty, nothing to update"<<endl;
        return;
    }
    cout<<endl<<"Enter the postion of node to be updated: ";
    cin>>pos;
    cout<<"Enter the new value: ";
    cin>>value;
```

```
    struct node *s;
    if (counter < pos)
    {
        cout<<"Position out of range"<<endl;
        return;
    }
    s = start;
    if (pos == 1)
    {
      s->info = value;
      cout<<"Node Updated"<<endl;
      return;
    }
    for (i=0;i < pos - 1;i++)
    {
        s = s->next;
    }
    s->info = value;
    cout<<"Node Updated"<<endl;
}
/*
 * البحث عن عنصر في القائمة *
```

```
 */
void double_clist::search()
{
   int pos = 0, value, i;
   bool flag = false;
   struct node *s;
   if (start == last && start == NULL)
   {
      cout<<"The List is empty, nothing to search"<<endl;
      return;
   }
   cout<<endl<<"Enter the value to be searched: ";
   cin>>value;
   s = start;
   for (i = 0;i < counter;i++)
   {
      pos++;
      if (s->info == value)
      {
         cout<<"Element "<<value<<" found at position: "<<pos<<endl;
         flag = true;
      }
      s = s->next;
```

```
    }
    if (!flag)
        cout<<"Element not found in the list"<<endl;
}
/*
 *  ترتيب القائمة الموصولة الثنائية الدائرية
 */
void double_clist::sort()
{
    struct node *temp, *s;
    int value, i;
    if (start == last && start == NULL)
    {
        cout<<"The List is empty, nothing to sort"<<endl;
        return;
    }
    s = start;
    for (i = 0;i < counter;i++)
    {
        temp = s->next;
        while (temp != start)
        {
```

```
            if (s->info > temp->info)
            {
                value = s->info;
                s->info = temp->info;
                temp->info = value;
            }
            temp = temp->next;
        }
        s = s->next;
    }
}
/*
 * عرض محتويات القائمة
 */
void double_clist::display()
{
    int i;
    struct node *s;
    if (start == last && start == NULL)
    {
        cout<<"The List is empty, nothing to display"<<endl;
        return;
    }
```

```
    s = start;
    for (i = 0;i < counter-1;i++)
    {
        cout<<s->info<<"<->";
        s = s->next;
    }
    cout<<s->info<<endl;
}
/*
 * عكس ترتيب القائمة الموصولة الثنائية الدائرية
 */
void double_clist::reverse()
{
    if (start == last && start == NULL)
    {
        cout<<"The List is empty, nothing to reverse"<<endl;
        return;
    }
    struct node *p1, *p2;
    p1 = start;
    p2 = p1->next;
    p1->next = NULL;
```

```
    p1->prev = p2;
    while (p2 != start)
    {
        p2->prev = p2->next;
        p2->next = p1;
        p1 = p2;
        p2 = p2->prev;
    }
    last = start;
    start = p1;
    cout<<"List Reversed"<<endl;
}
```

الفصل السابع
الاستدعاء الذاتي RECURSION

7.1 المقدمة

فكرة الاعادة (recursive) ليست معروفة بشكل عام في العالم الحقيقي, ولذلك فهي ربما تبدو مربكة للمبرمج المبتديء. عليه, فاني اتوقع بامكانية استخدامهم مباديء الاعادة بشكل تدريجي.

احيانا تكون المشكلة صعبة جدا او معقدة جدا لكي يمكن حلها وذلك لانها كبيرة جدا. فاذا كان من الممكن تجزأة المشكلة الى نسخ صغيرة مشابه للاصل, فربما يكون من الممكن ايجاد طريقة لحل واحدة من هذه النسخ الصغيرة وبهذا سنكون قادرين على بناء حل لكامل المشكلة. هذه هي الفكرة ما وراء الاعادة (recursive), خوارزمية الاعادة تجزأ المشكلة الى اجزاء صغيرة والتي ربما تكون اجابتها معروفة مسبقا, او من الممكن حلها بتطبيق نفس الخوارزمية على كل جزء صغير, وبعدها يتم دمج الحلول.

الاستدعاء الذاتي هي عملية اعادة العناصر نفسها وبنفس الطريقة. في لغات البرمجة, اذا البرنامج سمح لك لاستدعاء الدالة من داخل نفس الدالة, فعند ذلك تدعى الاستدعاء الذاتي للدالة.

لغة C++ تدعم الاستدعاء الذاتي, بمعنى دالة تستدعي نفسها. ولكن عند استخدام الاستدعاء الذاتي, المبرمجين بحاجة لان يكونوا حذرين بتعريف شرط الخروج من الدالة, في خلاف ذلك فانها ستذهب (الدالة) بحلقة تكرار لا نهائية.

7.2 الاعادة Recursion

الاعادة في علم الحاسوب هي طريقة تستند الى مفهوم حل المشكلة اعتمادا على حلول الحالات الصغيرة لنفس المشكلة (عكس التكرار). الطريقة من الممكن ان تطبق على العديد من المشاكل, والاعادة هي واحد من الافكار الرئيسية لعلوم الحاسوب.

دوال الاستدعاء الذاتي مفيدة جدا لحل العديد من مشاكل الرياضيات, مثل حساب مضروب رقم معين, توليد متوالية Fibonacci وغيرها.

ان قوة الاستدعاء الذاتي يكمن في امكانية تعريف مجموعة غير محددة من الكيانات بواسطة عبارات محددة. بنفس الطريقة, عدد غير محدد من الحسابات من الممكن وصفها بواسطة برنامج استدعاء ذاتي, حتى وان كان هذا البرنامج لا يحتوي على تكرارات خارجية. خوارزميات الاستدعاء الذاتي, هي مناسبة مبدئيا عندما تكون المشكلة قابلة للحل, او الدالة التي نبرمجها او هيكل البيانات التي تعالج هي بالاساس معرفة بدلالة الاستدعاء الذاتي.

ولكي نبسط الموضوع سنتطرق الى الدمى الروسية, فهل سبق لك أن رأيت مجموعة من الدمى الروسية؟ في البداية، ترى مجرد تمثال واحد من الخشب ويرسم عليه نقوش لتبدو كما في الشكل 7.1.

شكل 7.1: الدمى الروسية تبدو دمية واحدة.

الان لو رفعت النصف العلوي من الدمية الاولى, فماذا سترى في داخلها؟ سترى ان هناك دمية روسية اصغر منها قليلا. كما في الشكل 7.2.

شكل 7.2: الدمية الروسية بعد رفع النصف الاعلى من الدمية الاولى.

من الممكن ان تزيل هذه الدمية الاصغر وتفصل النصف العلوي والنصف السفلي للدمية الثانية. سترى دمية اخرى اصغر منها قليلا شكل 7.3.

شكل7.3: الدمى الروسية بعد فتح الدمية الثانية.

ان الاستمرار بهذا العمل (فتح كل دمية لايجاد دمية اخرى داخلها) سيؤدي الى الحصول على مجموعة من الدمى وكل واحدة هي اصغر من سابقتها قليلا (علما بانها جميعا كانت ضمن دمية واحدة رئيسية) وكما في الشكل 7.4.

الشكل 7.4: الدمى الروسية بعد فتح جميع الدمى.

لاحظ هنا اننا بدأنا بدمية كبيرة وتم التجزئة الى مجموعة من الدمى اصغر فاصغر لحين الوصول الى دمية صغير جدا بحيث لاتحتوي على دمية اصغر منها.

ما عملناه للدمى الروسية سنعمله للخوارزميات, فان المشكلة الكبيرة من الممكن ان تجزأ الى مشاكل اصغر واصغر مشابهة للمشكلة الكبيرة الاصلية لحين الوصول الى جزء صغير لايمكن تجزأته ومن الممكن حلها بشكل مباشر. هذه التقنية تدعى الاعادة (او الاستدعاء الذاتي).

7.3 كيف يستخدم الاستدعاء الذاتي

من الافضل ان يستخدم الاستدعاء الذاتي عندما تكون لدينا مهمة كبيرة (task), ومن الممكن تجزئتها الى مهمات جزئية فرعية مكررة. ولان دالة الاستدعاء تستدعي نفسها لإنجاز المهمة الجزئية, لذا في النهاية فان الدالة ستاتي على مهمة جزئية بدون ان تستدعي نفسها. وهذه تعرف الحالة الاساس (base case), ونحتاج لها لمنع الدالة من استدعاء نفسها مرة بعد مرة بدون توقف. لذلك من الممكن القول بان الحالة الاساس توقف الاستدعاء الذاتي.

في الحالة الاساس, فان الدالة لا تستدعي نفسها. ان استدعاء الدالة نفسها لغرض انجاز مهمتها الجزئية, تعرف بانها حالة الاعادة (recursive case).

هناك نوعين من الحالات عند استخدام خوارزمية الاستدعاء الذاتي (حالة الاساس, وحالة الاعادة). هذه من المهم جدا تذكرها عند استخدام الاستدعاء الذاتي, وعندما تحاول حل مشكلة عليك ان تسال نفسك: " ماهي حالتي الاساس, وماهي حالة الاعادة".

ان كلمة الاستدعاء الذاتي تعني " ان لها صفات لان تعود مرة ثانية, او تتكرر". ان عملية استدعاء الدالة لنفسها تسمى استدعاء مباشر وجميع الامثلة في هذا الفصل هي من نوع الاستدعاء المباشر. اما الاستدعاء غير المباشر فيتم عندما دالة A تستدعي دالة B ودالة B تستدعي دالة A.

7.4 الاستدعاء الذاتي باستخدام مثال

لكي نوضح عمل دالة الاستدعاء الذاتي فاننا سنشرح الموضوع باستخدام مثال, مثالنا هنا ايجاد مضروب رقم معين (factorial). بداية سنكتب البرنامج الذي يوجد نتيجة المضروب ومن ثم نشرح البرنامج خطوة خطوة.

```
#include <stdio.h>
int factorial (unsigned int i) {
  if(i <= 1) {                    // السطر 1
    return 1;                     // السطر 2
  }
  return i * factorial (i - 1);       // السطر 3
}
int  main() {
  cin>> i ;
  cout<< "Factorial of \n" <<  i << " is " <<  factorial(i) ;
  return 0;
}
```

لتوضيح عمل الاستدعاء الذاتي وحسب البرنامج اعلاه سنفرض ان المطلوب هو ايجاد مضروب الرقم 4. بداية لنشرح البرنامج ثم نوضح كيفية ايجاد مضروب

الرقم 4. لاحظ ان البداية سيتم ادخال القيمة المطلوب ايجاد المضروب لها ومن ثم تستدعى دالة ايجاد المضروب (سيتم ادخال الرقم 4). الدالة التي سيتم استدعائها في هذا البرنامج هي باسم (factorial) والتي تجد المضروب, المدخل لها قيمة الرقم الذي نحاول ايجاد المضروب له, هنا سيكون الرقم 4.

لاحظ في بداية الدالة هناك شرط, هذا الشرط يستخدم لايقاف عملية الاستدعاء الذاتي وعدم السماح بدخول الدالة الى حلقة تكرار لانهائية (حالة الاساس). طبعا شرط التوقف هو عندما نصل الى واحد فان الدالة ستصل نهايتها. الفكرة هو ان لم نصل واحد فان الدالة ستستدعي نفسها. واحدة من الطرق الرياضية لوصف (n!):

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1) * (n-2) * \cdots * 1, & \text{if } n > 0 \end{cases}$$

نفرض كما في المثال ان المطلوب هو (4!), وبسبب ان (n>0) فاننا سنستخدم الجزء الثاني من التعريف اعلاه.

4! = 4 * 3 * 2 * 1 = 24

من الممكن ايضا ان نوضح ان (n!) من الممكن كتابتها لاي قيمة (n) ليست سالبة.

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

هذا التعريف هو تعريف الاستدعاء الذاتي, لاننا وضحنا دالة المضروب بدلالة نفسها. ووفقا للتعريف في العلاقة اعلاه فان

4! = 4 * (4-1)! = 4 * 3!

بالطبع فان عملية الضرب لايمكن انجازها الان, وذلك لاننا لانعرف قيمة (!3). لذلك, لنرى كيف سيتم انجاز هذا العمل عن طريق الاستدعاء الذاتي (هنا تم تجزئة المشكلة الى اجزاء صغيرة مشابهة للمشكلة الاصلية) .

نفرض ان حسين هو احد الطلبة في قسم الحاسوب وطلب منه ايجاد (!4), حسين فهم ان

$4! = 4 * (4-1)! = 4 * 3!$

ونظرا لان حسين لا يعرف قيمة (!3), لذلك قرر الاتصال بزميلته جنى وسالها عن قيمة (!3), جنى فكرت بالمسألة وهي تملك نفس الصيغة (تعريف المضروب) وتوصلت الى ان

$3! = 3 * (3 – 1)! = 3 * 2!$

شكل 7.5: توضيح لطريقة ايجاد المضروب.

جنى لم تستطع ايجاد النتيجة النهائية حيث لا يمكنها اكمال عملية الضرب نظرا لعدم معرفتها بقيمة (!2), لذلك قررت جنى ان تجعل حسين ينتظر وتتصل بزميلها محمد لمعرفة قيمة (!2).

شكل 7.6: تجزئة مشكلة المضروب الى اجزاء اصغر.

محمد يملك نفس العلاقة الرياضية التي يملكها كل من حسين وجنى, لذلك فقد توصل الى ان

$$2! = 2 * (2 - 1)! = 2 * 1!$$

لكن محمد لم يستطع اكمال عملية الضرب لعدم معرفته نتيجة (!1), لذلك طلب من جنى الانتظار لحين ايجاد النتيجة (الان جنى وحسين بالانتظار).

محمد قرر الاتصال بوالدته والتي هي مدرسة فيزياء ليسألها عن نتيجة (!1).

شكل 7.7: ايجاد مضروب الرقم 1.

والدة محمد تعرف ذات العلاقة, لذلك فهي بسرعة توصلت الى ان

$1! = 1 * (1 - 1)! = 1 * 0!$

طبعا هي لم تتمكن من انجاز علمية الضرب وذلك لانها لم تعرف نتيجة (!0), لذلك فان والدة محمد قررت ان تجعل محمد ينتظر واتصلت بزوجها (والد محمد) مدرس الرياضيات لتطلب منه معرفة نتيجة (!0), والد محمد لم يحتاج الى وقت للتفكير فهو يعرف ذات العلاقة والتعريف, ويعلم ان الجزء الاول من التعريف يشير الى ان

$n! = 1, \quad if \quad n = 0$

لذلك فهو قد اخبرها بالنتيجة مباشرة وقال لها ان (1 = !0).

عندما علمت والدة محمد ان (1 = !0) اكملت عملية الضرب واستخرجت النتيجة وكما يلي

$1! = 1 * (1 - 1)! = 1 * 0! = 1 * 1 = 1$

والدة محمد اسرعت والفرح يغمرها باعلام محمد النتيجة وهي (1 = !1).

محمد بمجرد سماع النتيجة عوض في العلاقة التي لدية واستخرج النتيجة كما يلي

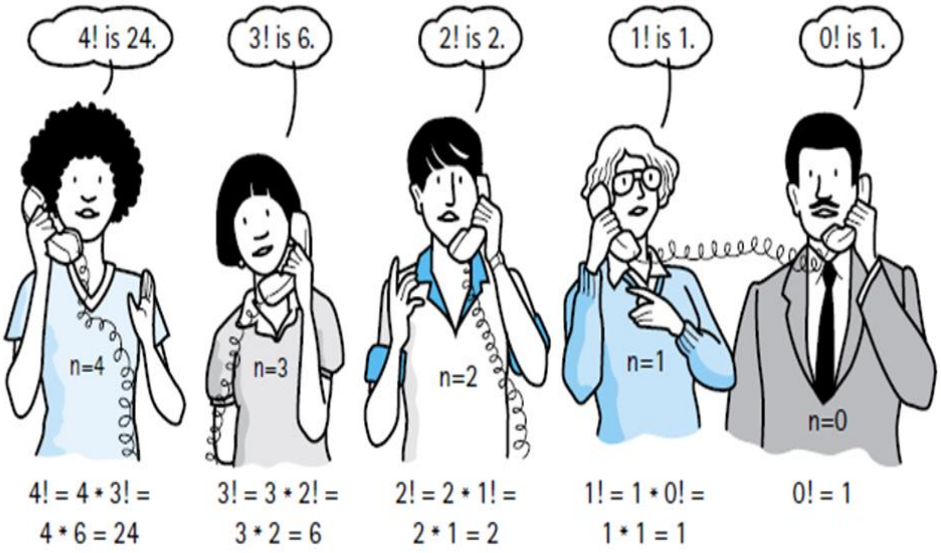
$2! = 2 * (2 - 1)! = 2 * 1! = 2 * 1 = 2$

بعد حصول محمد على النتيجة عاد الى جنى ليعلمها ان (2 = !2).
جنى والتي طال انتظارها اكملت عملية الضرب لتستخرج قيمة (!3) (هي كانت بانتظار نتيجة !2).

$3! = 3 * (3 – 1)! = 3 * 2! = 3 * 2 = 6$

بنشوة النصر اخبرت جنى زميلها حسين بان نتيجة (6 = !3) معتذرة عن التأخير لانشغالها قليلا بالمطبخ!.
واخيرا فان حسين استخدم هذه القيمة التي حصل عنها من جنى ليكمل عملية الضرب كما في ادناه وهو في داخله يزداد اعجابا بذكاء جنى.

$4! = 4 * (4-1)! = 4 * 3! = 4 * 6 = 24$



شكل 7.8: استخراج القيمة النهائية لمضروب 4, باستخدام العودة (recursion)

من المؤكد, ان استخدام العودة هي ليست حكرا على مختصي الرياضيات باستخدام الهواتف. ان لغات البرمجة مثل C++ تدعم العودة وتوفر للمبرمجين اداة قوية وفاعلة لحل نوع محدد من المشاكل بتقليل درجة التعقيد او اخفاء تفاصيل المشكلة كما في شفرة البرنامج اعلاه.

لنتابع البرنامج الخاص بمضروب الرقم (4) باستخدام الجدول وذلك لغرض تبسيط الحل.

جدول 7.1: تحليل برنامج ايجاد المضروب.

استدعاء العودة	رقم الايعاز (السطر)	الفعل
	1	4 ليست صفر, اذن نذهب الى else
	3	في هذا الجزء سيتم اعادة !(1 – 4) * 4 لاحظ هنا ان الناتج هو 4*factorial(4-1)

		وهذا يعني اعادة استدعاء الدالةfactorial ومن داخل الدالة نفسها لكن بقيمة اقل وهي (3)هذا الاستدعاء الاول بقيمة (!3)
1	1	3 ليست صفر, اذن نذهب الى else
	3	اعادة !(1 – 3) * 3 .. يتم الاستدعاء الثاني وبقيمة (!2)
2	1	2 ليست صفر, اذن نذهب الى else
	3	اعادة !(1 – 2) * 2 .. يتم الاستدعاء الثالث وبقيمة (!1)
3	1	1 ليست صفر, اذن نذهب الى else
	3	اعادة !(1 – 1) * 1 .. يتم الاستدعاء الرابع وبقيمة (!0)
4	1	(1-1) هو صفر, لذلك يتم التحول الى الايعاز 2
	2	اعادة 1
3	3	الرقم 1 يعوض استدعاء الدالة. الرقم 1. اعادة 1
2		الرقم 1 يعوض استدعاء الدالة. الرقم 2. اعادة 2
1		الرقم 2 يعوض استدعاء الدالة. الرقم 3. اعادة 6
		الرقم 6 يعوض استدعاء الدالة. الرقم 4. اعادة 24

للتاكد من ان حل العودة (الاستدعاء الذاتي) يعمل, يجب ان تكون قادر على الاجابة بنعم للاسئلة الثلاث التالية:

1. سؤال حول الحالة الاساس: هل هناك طريقة غير الاستدعاء الذاتي وبدون الدالة لايجاد النتيجة لجزء صغير من المشكلة, وهل الدالة تعمل بشكل صحيح عند القيمة الاساس.
2. سؤال الاستدعاء الاصغر: هل ان الاستدعاء الذاتي للدالة يتضمن حالة تصغير للمشكلة الاصلية, بحيث يقود الى الحالة الاساس بشكل لا مفر منه.

3. سؤال الحالة العامة: نفرض ان الاستدعاء الذاتي يعمل بشكل صحيح, فهل ان كامل الدالة تعمل بشكل صحيح.

لنجرب هذه الشروط او الاسئلة على دالة المضروب التي تطرقنا لها:

1. ان الحالة الاساس تحدث عندما تكون قيمة (n=0), حيث ان الدالة ستسند القيمة 1. والتي هي القيمة الصحيحة لمضروب الصفر (!0), وعندها لا توجد استدعاءات عودة اضافية. الجواب هو نعم.
2. لاجابة هذا السؤال, يجب ان ننظر الى المعاملات التي تمرر الى استدعاء العودة. في الدالة (factorial) فان استدعاء العودة يمرر (n-1).
في كل مرة يتم فيها الاستدعاء الذاتي للدالة ترسل قيمة معاملات اصغر للدالة, لحين ان تكون القيمة المرسلة تساوي صفر. في هذه الحالة فاننا وصلنا الى اصغر حالة, ولا توجد استدعاءات اضافية اخرى. الجواب هو نعم.
3. في حالة دالة مثل الدالة (factorial) نحتاج ان نتحقق من ان الصيغة التي نستخدمها واقعا نتيجة حل صحيح. نفرض بان الاستدعاء الذاتي (factorial (n-1)) تعطينا قيمة صحيحة للمضروب !(n-1), عبارة الاعادة ستحسب n*(n-1)! وهذا هو تعريف المضروب, حيث اننا نعلم بان الدالة تعمل لجميع القيم الموجبة. لذلك فان الاجابة نعم.

7.5 كتابة دوال الاستدعاء الذاتي

بالامكان استخدام النقاط التالية لكتابة اي دالة استدعاء ذاتي:

1. المفروض ان يكون هناك تعريف واضح لحل المشكلة. (هذه بالطبع هي الخطوة الاولى لحل اي مشكلة برمجية).
2. تحديد حجم المشكلة المطلوب حلها عند استدعاء الدالة. في الاستدعاء الاولي للدالة, فان حجم كامل المشكلة يوضح في قيمة او قيم معامل او معاملات الدالة.

3. تحديد وتعريف الحالة الاساس, والتي توضح المشكلة بدون عمليات العودة. و يتم التاكد منها عند الاجابة بنعم لسؤال الحالة الاساس.
4. تعريف وحل الحالة العامة بشكل صحيح بدلالة حالة اصغر لنفس المشكلة (استدعاء ذاتي). هذه تؤكد الاجابة نعم لسؤالي الاستدعاء الاصغر والحالة العامة.

لو حاولنا تطبيق هذه الفقرات على برنامج المضروب.

تعريف المشكلة من الممكن ان نختصره بتعريف دالة المضروب. حجم المشكلة هو عدد القيم التي من المفروض ان تضرب وهي تساوي N.

الحالة الاساس تحدث عندما تكون (N = 0), في حالة اننا سلكنا طريق عدم الاستدعاء الذاتي.

اخيرا, الحالة العامة تحدث عندما تكون (N>0) يؤدي الى استدعاء ذاتي للدالة factorial لحالة اصغر(N-1) factorial.

❖ **مثال**: البرنامج التالي يولد متوالية Fibonacci لعدد محدد.

التعريف الرياضي لارقام الفيبوناشي Fibonacci هو

fib(0) = 1

fib(1) = 1

fib(n) = fib(n-1) + fib(n-2)

```
#include <iostream>
 using namespace std;

 int fib(int n)    // 1, 1, 2, 3, 5, 8, 13, 21, ...
 {
   if (n == 0 || n == 1)
       return 1;
```

```
  return  fib(n-1) + fib(n-2);
 }

 main()
 {
  int n;

  cout << "Enter a non-negative integer: ";
  cin >> n;
  cout << "Fib(" << n << ") = " << fib(n) << "." << endl;
  return 0;
 }
```

❖ **مثال**: برنامج لايجاد مجموع ارقام موجبة بطريقة الاستدعاء الذاتي

1 + 2 + 3 + … + n

```
# include<iostream>
using namespace std;
int sum (int) ;  // اعلان عن دالة
void main (void) {
int  n  ، temp ;
cout<<"enter any integer number "<<endl ;
cin>>n ;
temp = sum (n) ;
cout<<"value = "<<n<<"and its sum="<<temp ;
}
int sum (int  n) //   دالة الاستدعاء الذاتي
{
  int value = 0 ;
```

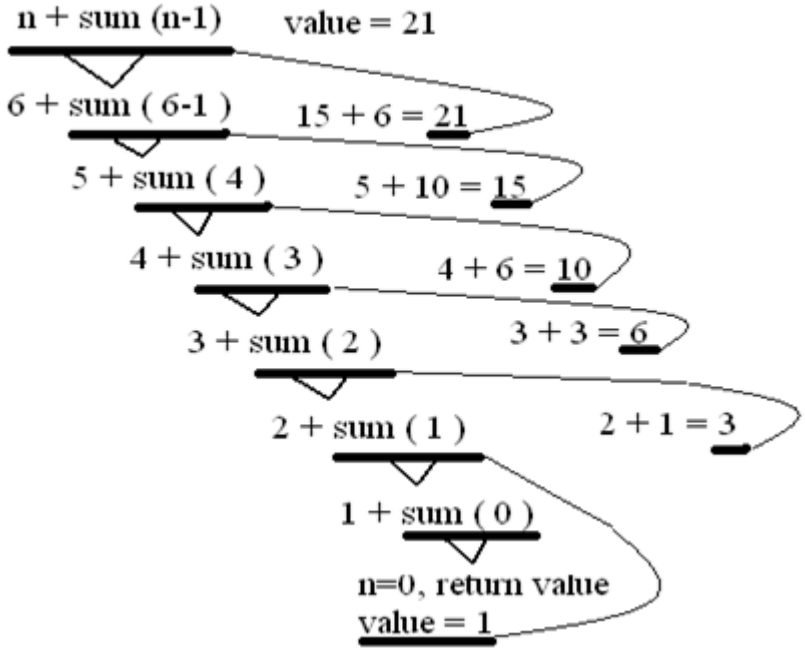
```
  if (n==0)  return (value);
    else
      value  = n + sum(n-1) ;
    return (value);
}
```

التوضيح التالي من الممكن ان يساعد على ادراك وفهم دالة الاستدعاء الذاتي.

فلو فرضنا انه تم ادخال قيمة للمتغير (n=6) لنرى كيف سيتم ايجاد المجموع، اعتمادا على قيمة العبارة

```
value = n + sum (n-1) ;
```

هنا العبارة ((sum (n-1)) هي استدعاء للدالة ((sum(int))، ونظرا لان هذا الاستدعاء من داخل الدالة نفسها اذن هو يمثل استدعاء ذاتي اي استدعاء لنفسه وسيكون عملها كما يأتي:



شكل 7.9: يبين كيفية تنفيذ دالة الاستدعاء الذاتي لبرنامج جمع متوالية

n + sum (n-1) = 6 + sum (6-1) = 6 + sum (5)

- ✓ يتم استدعاء الدالة لايجاد المجموع عندما (n=5) ويضاف الى (6)
- ✓ تستمر العملية لحين الوصول الى (n=0) فتعاد القيمة (value) والتي قيمتها (1).
- ✓ الان يكون الرجوع لتعويض القيم المستنتجة والبدء اولا بقيمة ((1) sum) حيث وجدت قيمتة (1) لتعوض بمكانها كما في اعلاه
- ✓ ستجمع نتيجة ((sum(1) مع الرقم (2) لاستخراج نتيجة ((sum(2) وتستمر العملية الى النهاية كما في الشكل 7.9.

هياكل البيانات باستخدام C++

الفصل الثامن
الأشجار الثنائية BINARY TREES

8.1 المقدمة

هناك عدد من اساسيات هياكل البيانات التي من الممكن استخدامها لحل مشاكل التطبيقات. المصفوفات تعتبر جيدة لهياكل البيانات المستقرة والتي من الممكن الوصول اليها عشوائيا وهي تنفذ بسهولة. القوائم الموصولة من جانب اخر هي ليست مستقرة وهي مثالية للتطبيقات التي تتطلب عمليات متكررة مثل الاضافة, الحذف, والتحديث. واحدة من مشاكل القوائم الموصولة هي الوصول المتسلسل للبيانات. هناك هياكل بيانات خاصة اخرى مثل المكدس والطابور التي تسمح لنا بحل المشاكل المعقدة باستخدام هياكل البيانات المحددة هذه. هيكل بيانات اخر هو جدول التجزئة (Hash table) التي تسمح للمستخدم لبرمجة التطبيقات التي تحتاج الى بحث وتحديث متكرر.

احد مساوئ استخدام المصفوفات والقوائم الموصولة لخزن البيانات هو الوقت الازم للبحث عن عنصر معين, وحيث ان كليهما المصفوفات والقوائم الموصولة هي هياكل خطية, فان الوقت المطلوب لبحث قائمة خطية يتناسب مع حجم البيانات في المجموعة. مثال, اذا كان حجم البيانات في المجموعة هو n, عليه فان عدد المقارنات التي تحتاجها لايجاد (او عدم ايجاد) عنصر معين ربما تكون من السوء لضرب n عدد من المرات. لذلك تصور انك تريد البحث في قائمة موصولة (او مصفوفة) وتحتوي مثلا $n = 10^6$ عقدة. حتى في الماكنة التي بإمكانها ان تقوم بمليون عملية مقارنة بالثانية, فان البحث عن m عنصر ستحتاج الى m من الثواني تقريبا. هذا غير مقبول في عالم اليوم حيث ان السرعة في اكمال عملية معينة هو مهم جدا. وهذا يترجم كمبالغ مالية, عليه فان هناك حاجة لهياكل بيانات افضل (اكثر كفاءة) لخزن البيانات والبحث عنها.

8.2 تعريف الاشجار

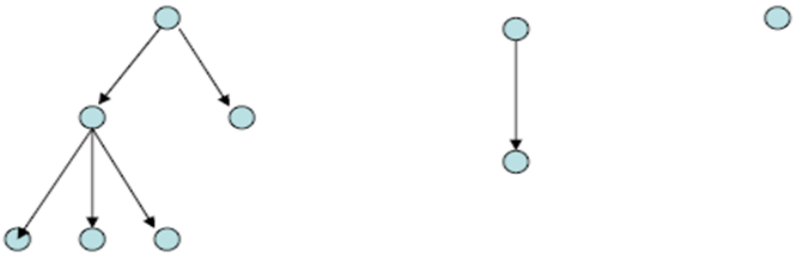
الاشجار هي هياكل بيانات غير خطية, تنظم البيانات بشكل هرمي, والاشجار اما ان تكون خالية (عدم وجود عقد) او ان تكون عبارة عن جذر فقط او جذر وفرع او اكثر من الاشجار الفرعية كما في الشكل 8.1.

من صفات الاشجار غير الخالية

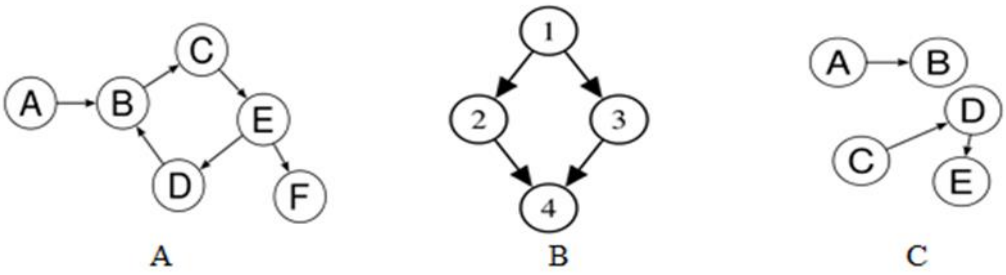
i. هناك بالضبط عقدة واحدة, تدعى الجذر لها درجة الدخول صفر (indegree)

I. اي عقده غير الجذر لها درجة الدخول واحد 1.

II. لكل عقدة a من الشجرة هناك طريق مباشر من الجذر الى العقدة a.



شكل 8.1: يوضح نماذج من الاشجار.



شكل 8.2: هذه ليست اشجار وذلك A. لانها دائرية B-C-E-D-B. B. دائرية غير مباشرة 1-2-4-3. C. فرعين غير متصلة A-B , C-D-E

8.3 العقدة Node

عقدة الشجرة تمثل المعلومات المتضمنة في هيكل مفرد. هذه العقد من الممكن ان تحتوي على قيم او شرط, او ربما تكون بمثابة هيكل بياني اخر مستقل. العقدة هي مواقع خزن متجاورة تعرف كهيكل بياني. في الاشجار الثنائية فان العقدة تتكون من ثلاث حقول, حقل البيانات (او المعلومات) والتي من الممكن ان تكون هذه البيانات: قيم رقيمة, رموز, حروف او نصوص او اي هيكل بياني مستقل. وحقلين للعناوين واحد لعنوان الابن الايمن والاخر لعنوان الابن الايسر وهي عبارة عن مؤشرات (احدهما يؤشر الى العقدة في اليمين والثاني يؤشر الى العقدة في اليسار) وهذه العقدة مشابهة للعقدة التي استخدمت في القوائم الموصولة الثنائية في فصل القوائم الموصولة لاحظ الشكل 8.3.

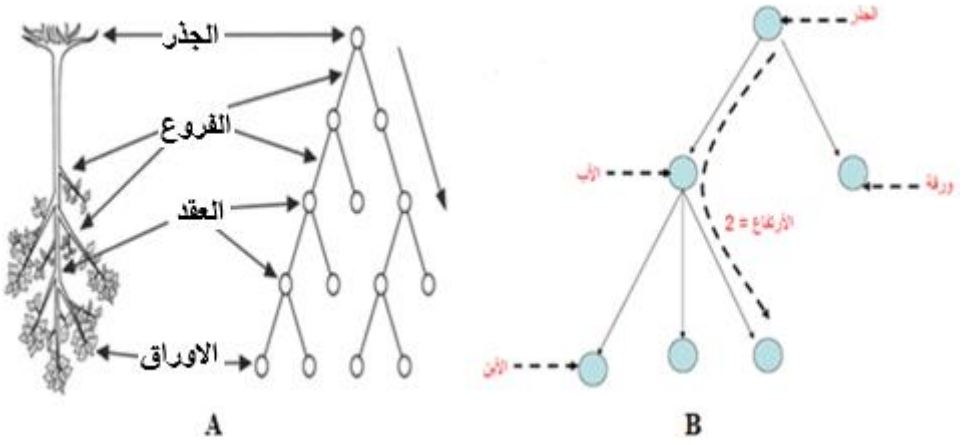
يرجى ملاحظة امكانية تمثيل العقدة بخلية مصفوفة وسناتي لاحقا عليها لكن غالبا يكون التمثيل على شكل قوائم موصولة. الشفرة الخاصة بالإعلان عن العقدة هي:

```
struct node {
int data;
struct node *leftChild;
struct node *rightChild;
};
```

شكل 8.3: عقدة القوائم الموصولة المستخدمة في الاشجار الثنائية.

8.4 مصطلحات الاشجار Trees Terminology

العديد من المصطلحات تستخدم لوصف عناصر الاشجار. نحن نحتاج لمعرفة هذه المصطلحات بحيث يكون نقاشنا مفهوم. لحسن الحظ, غالبية هذه المصطلحات تتعلق بالأشجار الحقيقية او العلاقات العائلية (مثل استخدامنا لمصطلح الاب والابن), وبذلك سوف لا يكون من الصعب تذكرهم, الشكل 8.4 يوضح الشجرة الطبيعية وشجرة هياكل البيانات.



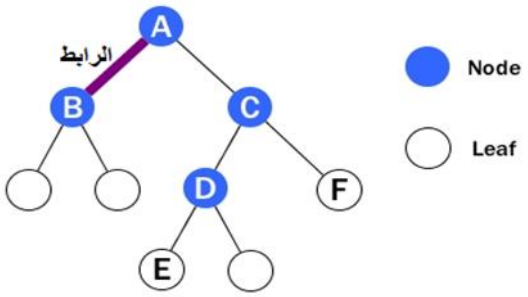
شكل 8.4: توضيح للتماثل بين الشجرة الطبيعية (A) وشجرة هيكل البيانات (B).

في التالي اهم المصطلحات التي تستخدم مع اشجار هياكل البيانات.

- **الجذر Root** : اعلى عقدة في الشجرة. العقدة في اعلى الشجرة دائما تدعى الجذر. هناك جذر واحد فقط في الشجرة. عند وجود تجمع للعقد وروابط بين العقد والتي تعرف كشجرة, فانه يجب ان يكون هناك مسار واحد وواحد فقط من الجذر الى اي عقدة اخرى. هنا نتحدث عن الجذر لكامل الشجرة. احيانا

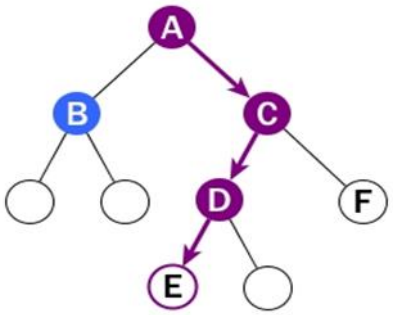
اعلى عقدة في الشجرة الفرعية تدعى جذر لتلك الشجرة الفرعية, (هنا شجرة هياكل البيانات هي شجرة مقلوبة بمعنى الجذر للاعلى والاوراق للاسفل).

- **الابن Child** : عقدة او اكثر التي تلي مباشرة عقدة اخرى. اي عقدة ربما لها رابط او اكثر ينطلق الى الاسفل الى عقد اخرى. هذه العقدة اسفل عقدة معينة مباشرة تدعى ابن.
- **الاب Parent** : الفكرة المعاكسة للابن وهي العقدة التي اعلى الابن وتتصل به مباشرة. اي عقدة (باستثناء الجذر) لها بالضبط رابط واحد ينطلق الى الاعلى, الى اي عقدة اخرى. لكل عقدة هناك عقدة واحدة فقط اعلى منها مباشرة تدعى الاب لهذه العقدة.
- **النسيب Sibling**: وهي العقد التي لها نفس الاب.
- **الورقة Leaf**: العقدة التي ليس لها ابناء تدعى العقدة الورقية او ببساطة اوراق. في الشجرة هناك جذر واحد لكن هناك عدد من الاوراق.
- **السليل Descendant**: وهي العقد التي نصل اليها بعمليات تكرارية من الاب الى الابناء.
- **السلف Ancestor**: وهي العقد التي نصل اليها بعمليات تكرارية من الابناء الى الاب.
- **العقدة الداخلية Internal node**: هي العقدة التي لها على الاقل ابن واحد.
- **العقدة الخارجية External node**: العقدة التي ليس لها ابناء.
- **الاشجار الفرعية Subtree**: اي عقدة من الممكن ان تعتبر جذر لشجرة فرعية, ان ابناء هذه العقدة وابناء ابنائها تمثل شجرة فرعية. اذا فكرت بمصطلحات العوائل فان الشجرة الفرعية تتكون من كل الاحفاد. الشجرة الفرعية من الممكن ان تكون مجموعة من العقد او عقدة واحدة.
- **الدرجة Degree**: عدد الاشجار الفرعية للعقدة.
- **الرابط (الحافة) Edge**: هو الرابط بين عقدة واخرى.



شكل 8.5: يبين العقدة, الوراق والرابط

- **المسار Path**: فكر بان احدهم يسير من عقدة الى عقدة اخرى على طول الروابط التي تربطهم. المتوالية الناتجة من العقد (الطريق الذي يوصلنا الى العقدة المطلوبة) تدعى مسار. اذن هي مجموعة من الروابط التي تربط الاب الى ابنائه والى احفاده وهكذا.

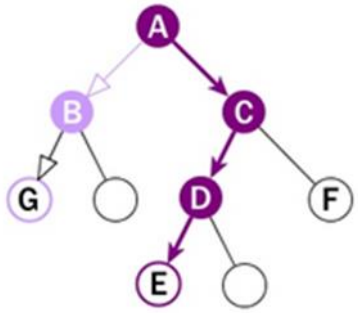


شكل 8.6: شكل توضيحي للمسار.

- **المستوى Level** :المستوى لعقدة معينة يشير الى عدد الاجيال للعقدة من الجذر. اذا فرضنا الجذر في المستوى صفر, عليه فان الابن سيكون في المستوى 1, الابن الحفيد سيكون في المستوى 2 وهكذا.
- **المفاتيح Keys:** في حقول البيانات في كيان معين توجد قيمة تسمى هذه القيمة مفتاح (اي القيمة التي في حقل البيانات للعقدة). هذه القيمة من الممكن ان تستخدم للبحث عن عنصر معين, او انجاز عمليات محددة عليها. عند تمثيل

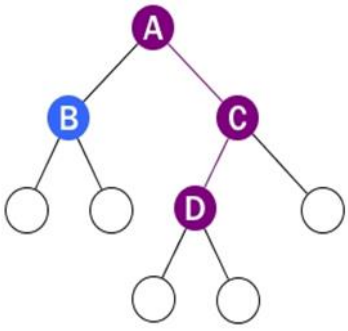
الشجرة على شكل مخطط للشجرة, سنرى دوائر تمثل العقد, وهذه الدوائر تحمل بيانات هذه البيانات تمثل المفتاح.

- ❖ **ارتفاع العقدة Node Height**: هو طول الطريق (عدد الحواف او الروابط) لأطول طريق بين العقدة والورقة.
- ❖ **ارتفاع الشجرة Tree Height**: ان اطول مسار (يتضمن الجذر والاوراق) يمثل ارتفاع الشجرة. بالنسبة للاشجار غير الخالية, فان الارتفاع هو اطول مسار في الشجرة + 1.



شكل 8.7: يوضح مفهوم الارتفاع

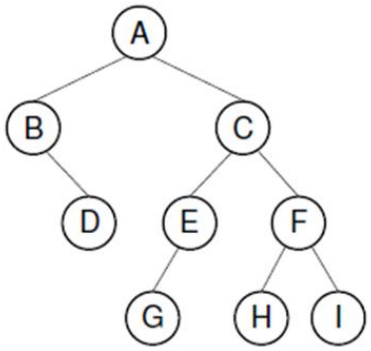
- ❖ **عمق العقدة Node Depth**: هو طول الطريق من الجذر الى العقدة.



شكل 8.8: عمق العقدة

- ❖ **عمق الشجرة Tree Depth**: هو طول الطريق من الجذر الى اعمق عقدة في الشجرة.

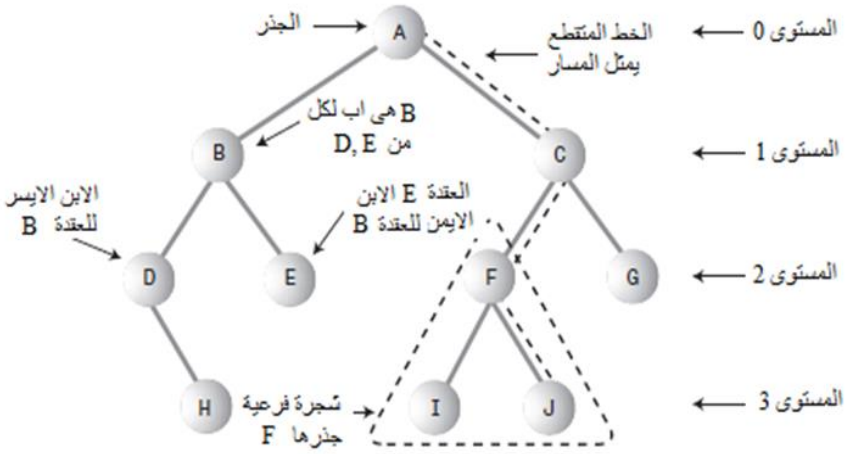
- **حجم العقدة Size of Node** : هو عدد العقد التي تمثل ابناء واحفاد العقدة (والعقدة نفسها تحسب معهم) اي ان حجم الجذر يمثل عدد العقد الكلية في الشجرة من ضمنها الجذر.
- **رتبة الشجرة Order of the Tree:** هي اكبر عدد من الابناء تحتويها عقدة من عقد الشجرة.
- **درجة الدخول In Degree**: للعقدة هي عدد الحواف التي تصل العقدة. الجذر هي العقدة الوحيدة التي لها (in degree = 0).
- **درجة الخروج Out Degree**: للعقدة هي عدد الحواف التي تترك العقدة.
- **الزيارة Visiting**: يتم زيارة العقدة عندما يصل المسيطر الى العقدة, عادة لإنجاز بعض العمليات على العقدة وليس للمرور فقط, مثل فحص القيمة لواحد من حقول البيانات او عرضها. ان المرور على عقدة في الطريق الذي ينقلنا من عقدة الى اخرى لا يعتبر زيارة للعقدة.
- **المرور Traversing**: المرور على شجرة يعني زيارة كل العقد بترتيب معين. مثال, ربما تزور كل العقد بترتيب تصاعدي حسب قيمة المفتاح. هناك العديد من طرق المرور على العقد كما سنراها لاحقا.



شكل 8.9: مثال لشجرة ثنائية.

في الشكل 8.9 فان العقدة A هي الجذر. اما العقدة B والعقدة C هم ابناء العقدة A. العقدة B مع العقدة D تكون شجرة فرعية. العقدة B لها اثنان من الابناء وهما

الابن الايسر وهنا هو شجرة خالية والابن الايمن الذي هو D. العقد A, C, F هم اسلاف العقدة H. العقد D, E, F تكون المستوى الثاني من الشجرة. العقدة A هي في المستوى صفر. المسار من A الى C الى E الى G يمثل طريق بطول 3. العقد D, G, H, I هي اوراق الشجرة. العقد A, B, C, E, F هي عقد داخلية. العقد G, H, I هي عقد خارجية. عمق العقدة I هو 3. اما ارتفاع هذه الشجرة هو 4.



شكل 8.10: عناصر الشجرة او مصطلحات الشجرة.

8.5 فوائد الأشجار

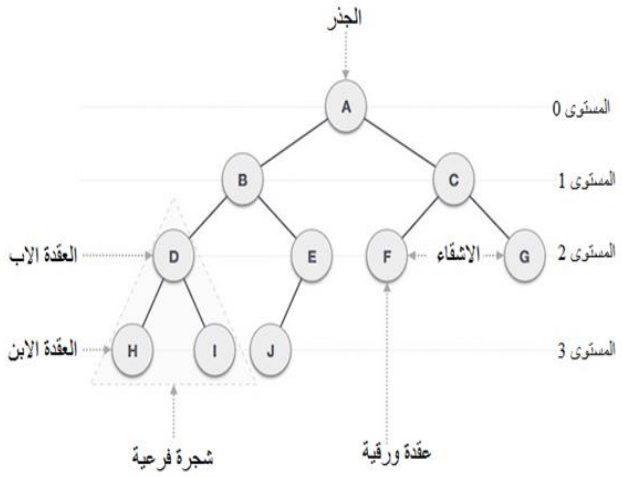
الاشجار مفيدة جدا وتستخدم دائما وذلك لانها تتمتع ببعض المميزات الجدية:

- الاشجار تعكس العلاقات الهيكلية في البيانات.
- الاشجار تستخدم لتمثيل الهرمية.
- الاشجار توفر عملية اضافة وحذف كفوءة.
- الاشجار هي بيانات مرنة جدا, تسمح بحركة الاشجار الفرعية باقل جهد.

8.6 الشجرة الثنائية Binary tree

الشجرة الثنائية هي هيكل بياني خاص يستخدم لاغراض خزن البيانات. والاشجار الثنائية لها الصفات التالية:

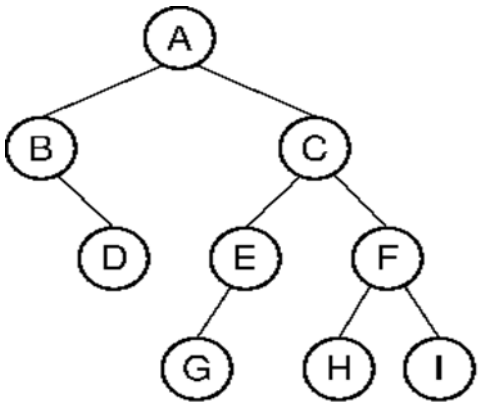
- اذا كانت كل عقدة في الشجرة لها على الاكثر اثنان من الابناء, فان هذه الشجرة تسمى ثنائية. وتعتبر الاشجار الثنائية هي الابسط والاكثر عمومية. الاشجار الثنائية تتكون من مجموعة من العقد, بحيث ان كل عقدة تحتوي على حقل مؤشر ايمن وحقل مؤشر ايسر فضلا عن حقل اخر للبيانات, العقدة الاعلى في الشجرة تسمى جذر.
- الابناء الاثنان لكل عقدة في الشجرة الثنائية تدعى الابن الايسر والابن الايمن, وفقا لموقعهم عند رسم شجرة معينة, كما في الشكل 8.11. العقدة في الشجرة الثنائية ليس من الضروري ان يكون لها اثنان من الابناء, فربما يكون لها ابن واحد سواء باليمين او باليسار, او من الممكن ان لا يكون لها اي ابن (في هذه الحالة تسمى ورقة).



شكل 8.11: شكل توضيحي للشجرة الثنائية.

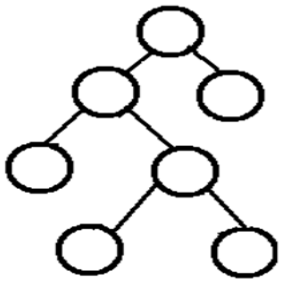
8.6.1 انواع الاشجار الثنائية

A. **شجرة ثنائية مجذرة Rooted Binary Tree:** هي شجرة بجذر حيث ان كل عقدة لها على الاكثر اثنان من الابناء كما في الشكل 8.12.



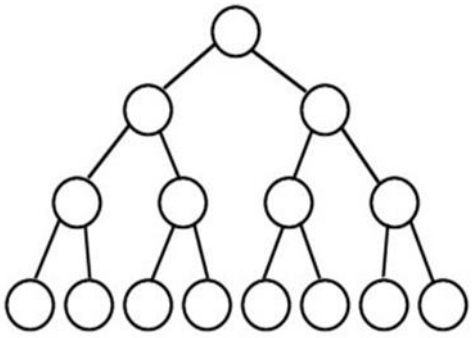
شكل 8.12: شجرة ثنائية مجذرة

B. **شجرة ثنائية مملوءة Full Binary Tree**: وتدعى احيانا (شجرة ثنائية مناسبة proper binary tree او 2-tree) وهي شجرة ثنائية بحيث ان كل عقدة عدا الاوراق لها اثنان من الابناء بالضبط, انظر الشكل 8.13.



شكل 8.13: شجرة ثنائية مملوءة

C. **شجرة ثنائية ممتازة Perfect Binary Tree**: حيث ان كل الاوراق بنفس العمق او نفس المستوى (احيانا تدعى شجرة ثنائية كاملة وتسبب التشويش), كما في الشكل 8.14.

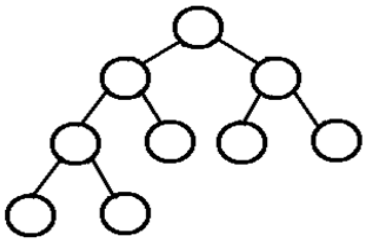


شكل 8.14: شجرة ثنائية ممتازة.

D. شجرة ثنائية كاملة Complete Binary Tree

الشجرة الثنائية الكاملة هي شجرة ثنائية بحيث:

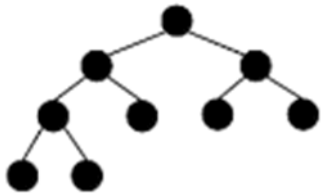
- كل المستويات عدا من المحتمل المستوى الاخير هو بالكامل مملوءة, وكذلك
- المستوى الاخير تكون جميع العقد في الجانب الايسر (بدون ان تفقد واحدة بينهم).



شكل 8.15: شجرة ثنائية كاملة.

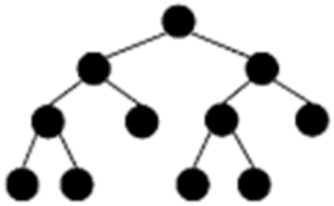
هناك حالات ممكنة للشجرة الكاملة, بحيث تكون الشجرة الثنائية

- ✓ مملوءة وكاملة



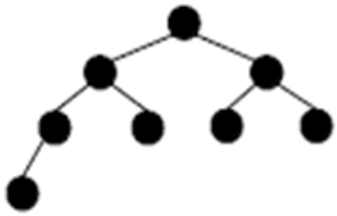
شكل 8.16: شجرة ثنائية مملوءة وكاملة.

- ✓ مملوءة وغير كاملة



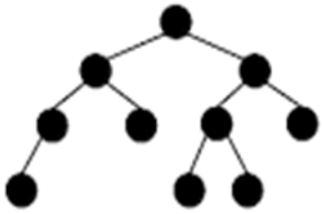
شكل 8.17: شجرة مملوءة وغير كاملة.

- ✓ كاملة ولكنها غير مملوءة



شكل 18.18: شجرة ثنائية كاملة ولكنها غير مملوءة.

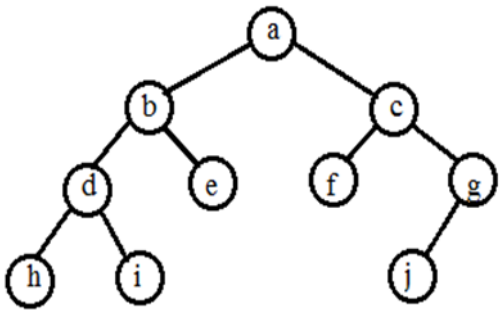
- ✓ وايضا ليست كاملة وليست مملوءة.



شكل 8.19: شجرة ثنائية كاملة وليست مملوءة.

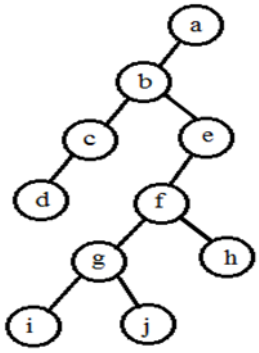
E. **شجرة ثنائية متوازنة Balance Binary Tree**: حيث ان عمق كل الاوراق يختلف بمقدار واحد على الاكثر. الاشجار المتعادلة لها عمق يمكن التنبؤ به (هي تمثل عدد العقد التي تمر عليها من الجذر الى الورقة, الجذر يحسب على انه العقدة صفروالباقية تحسب على انها العمق 1, 2, ..) هذا العمق يساوي جزء العدد الصحيح للعلاقة $log_2(n)$ حيث ان n يمثل عدد العقد في الشجرة المتعادلة. مثال الشجرة المتعادلة التي لها 3 عقد يكون عمقها $log_2(3) = 1.59$ اي يساوي 1.

تعتبر الشجرة متوازنة اذا كان كل مستوى فوق المستوى الادنى مملوء (بمعنى يحتوي على 2^nعقدة).



شكل 8.20: شجرة ثنائية متوازنة.

F. شجرة ثنائية غير متوازنة Unbalance Binary Tree



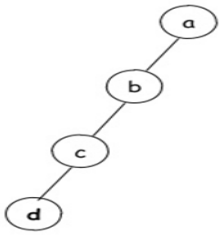
شكل 8.21: شجرة ثنائية غير متوازنة.

G. **شجرة إعادة التوليد Degeneration Tree**: وهي شجرة ثنائية بحيث ان لكل عقدة اب هناك فقط ابن يشترك معها. هذا يعني انه في قياسات الكفاءة فان الشجرة ستعامل مثل هيكل القوائم الموصولة.

H. الاشجار المنحرفة Skew Trees

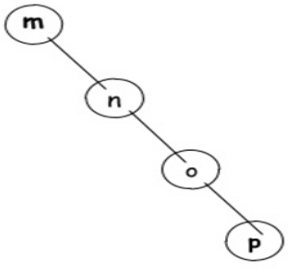
الاشجار المنحرفة هي اشجار بحيث تكون كل عقدة (عدا العقد الورقية) لها ابن واحد فقط. هناك نوعين من هذه الاشجار

✓ الاول هو الاشجار المنحرفة يسارا left skew tree
وهي شجرة منحرفة, بحيث ان كل عقدة ماعدا الجذر لها عقدة واحدة فقط في اليسار.



شكل 8.22: شجرة منحرفة يسارا

✓ النوع الثاني هي الاشجار المنحرفة يمينا Right skew tree
هي شجرة منحرفة بحيث كل عقدة عدا الجذر لها عقدة واحدة فقط في اليمين



شكل 8.23: شجرة منحرفة يمينا

8.7 اشجار البحث الثنائي Binary Search Tree

هي نوع خاص من الاشجار الثنائية. الفكرة الاساسية خلف هذا النوع من هياكل البيانات هو ان لها مستودع خزن يوفر طريقة كفوءة لترتيب البيانات, البحث, واستعادة البيانات.

اشجار البحث الثنائي تظهر سلوك خاص, هو ان العقدة التي في الجانب الايسر (الابن الايسر) يجب ان يكون له قيمة (مفتاح) اقل من قيمة الاب, والعقدة التي في اليمين (الابن الايمن) يجب ان تكون له قيمة (مفتاح) اكبر من قيمة الاب, ولا يسمح بتكرار المفتاح في الشجرة.

الشجرة الثنائية التي لها المواصفات التالية تسمى شجرة البحث الثنائي

- كل قيمة (مفتاح) في الشجرة تكرر بالضبط كحد اعلى مرة واحدة (اي لايوجد تكرار)
- العلاقات اكبر من واقل من, تعرف بشكل جيد لقيم البيانات.
- محددات الترتيب: لكل عقدة (n)

✓ كل البيانات في الشجرة الفرعية اليسرى للعقدة (n) هي اقل من البيانات في الجذر لتلك الشجرة الفرعية.

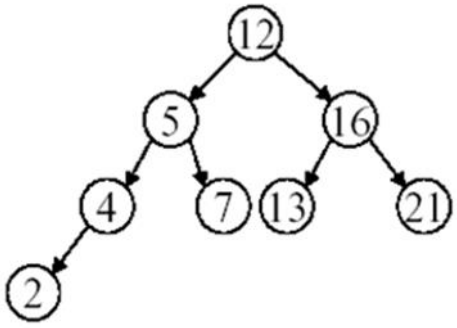
✓ كل البيانات في الشجرة الفرعية اليمنى للعقدة (n) هي اكبر من البيانات التي في الجذر لتلك الشجرة الفرعية.

8.7.1 خطوات بناء شجرة البحث الثنائي

A. الرقم الاول يعتبر جذر.

B. الارقام التالية اذا كانت اكبر من الجذر توضع في اليمين واذا اقل من الجذر توضع في اليسار.

مثال 1: ارسم شجرة بحث ثنائي للبيانات **12, 16, 5, 7, 4, 13, 21, 2**



شكل 8.24: شجرة بحث ثنائية لتمثيل القيم 12, 16, 5, 7, 4, 13, 21, 2

مثال 2: دعنا نرى المثال التالي لإضافة قائمة الارقام التالية والتي سيتم توضيح الاضافة بالتفصيل, حيث اننا سننتهي بشجرة مثل التي في الشكل 8.25:

32, 16, 34, 1, 87, 13, 7, 18, 14, 19, 23, 24, 41, 5, 53

- حيث ان القيمة 32 هي اول قيمة سيتم اضافتها, لذا فان 32 ستكون جذر الشجرة.
- القيمة التالية هي (16) والتي هي اصغر من القيمة 32, لذا فانها ستكون العقدة التي على يسار العقدة 32.
- (34) وحيث ان 34 اكبر من 32, عليه فان 34 ستكون العقدة التي توضع على يمين الجذر.
- القيمة (1) وهي اصغر من 32 لذا سيكون الاتجاه الى العقدة اليسرى للجذر. وبما ان العقدة في اليسار مشغولة بقيمة (16), لذا يتم اختبار القيمة 1 مع قيمة العقدة التي في اليسار 16, وحيث ان 1 اصغر من 16 علية فان 1 يكون الى يسار العقدة 16.

- القيمة (87) وحيث ان 87 اكبر من 32 سيكون الاتجاه الى العقدة التي على يمين الجذر, وهذه العقدة مشغولة بالقيمة (34), لذا تقارن مع 87, وحيث ان 87 اكبر من 34 لذا فان هذه القيمة ستكون على يمين العقدة 34.
- (13) وحيث ان 13 اصغر من 32, سيكون الاتجاه الى العقدة يسار الجذر, ثم تقارن مع العقدة التي على اليسار التي قيمتها 16, وحيث ان 13 اصغر من 16 تستمر الحركة الى العقدة التي على يسار العقدة 16 والتي هي مشغولة بالعقدة التي تحمل القيمة 1 وتقارن مع القيمة الجديدة, وحيث ان 13 اكبر من 1 لذا فان القيمة المضافة 13 ستكون على يمين العقدة 1.
- يستمر هذا النهج لبقية القيم **7, 18, 14, 19, 23, 24, 41, 5**.
- (53) وحيث 53 اكبر من 32 فسيكون موقعها في فرع الشجرة الايمن للجذر. كذلك 53 اكبر من 34 لذا استمر باتجاه اليمين للعقدة 34. ايضا 53 اصغر من 87 لذا يكون الاتجاه الى يسار العقدة 87. بعدها تقارن مع 41, وحيث ان 53 اكبر من 41 لذا يكون الاتجاه الى يمين العقدة 41, ونظرا الى ان يمين العقدة 41 فارغ او خالي من اي قيمة لذا فان 53 ستكون العقدة التي على يمين العقدة 41.

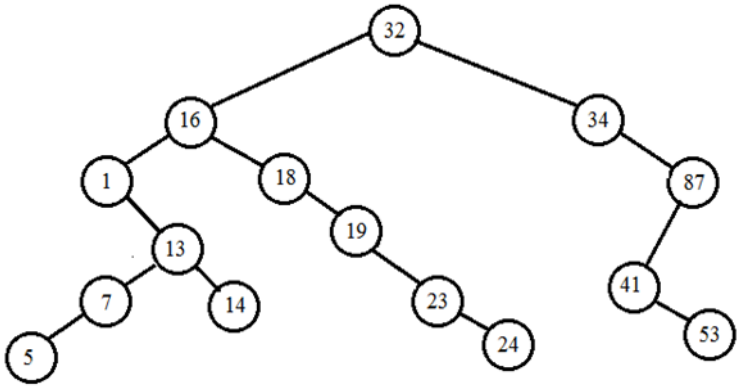
الخطوات اعلاه تعطيك فكرة عامة عن كيفية خلق الاشجار الثنائية. يجب عليك ان تعلم ان:

- ✓ ربط عقدة الى عقدة اخرى في الاشجار الثنائية هو بطبيعته واحد الى واحد.. اي ان العقدة لايؤشر عليها باكثر من عقدة واحدة.
- ✓ العقدة الواحدة بامكانها ان تؤشر الى اثنين من العقد الثانوية كحد اعلى.

هنا في الشجرة الثنائية الناتجة في الشكل 8.25, هناك كثير من العقد لديها مؤشر اليسار خالي, مثل العقد

5, 14, 1, 18, 19, 23, 24, 34, 41, 53 والتي عقدها من اليسار خالية.

هياكل



شكل 8.25: شجرة ثنائية تم بنائها لتمثل القيم (32, 16, 34, 1, 87, 13, 7, 18, 14, 19, 23, 24, 41, 5, 53)

8.8 العمليات التي تجرى على اشجار البحث الثنائي

هناك عدد من العمليات المهمة التي تجرى على شجرة البحث الثنائي, سنحاول توضيحها مع الاشارة الى شفرات البرامج الخاصة بها وهي:

A. البحث: وهي عملية البحث عن عنصر معين وحسب قيمته في الشجرة.
B. الاضافة: وهي عملية اضافة عنصر جديد للشجرة الثنائية.
C. الحذف: وهي عملية حذف احد عناصر الشجرة الثنائية.
D. المرور على عناصر الشجرة: وهي عملية زيارة جميع عناصر الشجرة الثنائية.

8.8.1 عملية البحث Search Operation

لغرض البحث عن عنصر معين عليك ان تبدا عملية البحث من الجذر, وتتم مقارنة قيمة البيانات التي تبحث عنها مع قيمة المفتاح (قيمة العقدة), وطبعا اما ان يتم البحث في فرع الشجرة الايمن اذا كانت قيمة البيانات اكبر من قيمة المفتاح, او يتم البحث في فرع الشجرة الايسر اذا كانت قيمة البيانات اصغر من قيمة المفتاح. هذا يتم تنفيذه على كل عقدة لحين الوصول الى البيانات المطلوب البحث عنها او ربما نصل الى نهاية الشجرة (الاوراق) دون ان نجد البيانات. خوارزمية البحث هي:

1. اذا تساوت بيانات الجذر مع البيانات التي نبحث عنها, عندها سيكون الجذر هو العقدة المطلوبة.
2. اذا لم تساوي الجذر, يتم البحث في العقد الاخرى.

A. اذا كانت البيانات التي نبحث عنها اكبر من بيانات العقدة (المفتاح):اذهب الى فرع الشجرة الايمن.
B. اما اذا كانت القيمة التي يتم البحث عنها اصغر من قيمة المفتاح, اذهب الى فرع الشجرة الايسر.

3. اذا تم ايجاد القيمة يتم اعادتها (طباعتها,عرضها او وضعها بمتغير ...الخ).
4. اذا لم يتم ايجاد أي بيانات مطابقة يتم عرض رسالة بان البيانات التي نبحث عنها غير موجودة.

8.8.2 عملية الاضافة Insert Operation

ان اول عملية اضافة هي عملية خلق شجرة (يتم ذلك بخلق عقدة تمثل الجذر). بعدها, امكانية اضافة عنصر اخر للشجرة. لغرض اضافة عنصر للشجرة, بداية من المفروض ايجاد الموقع المناسب لهذا العنصر. ابدأ البحث من عقدة الجذر باتباع الطريقة التي تم شرحها سابقا بحيث يقارن المفتاح (قيمة العقدة) مع بيانات العقدة المراد اضافتها فاذا كانت اقل من قيمة المفتاح يستمر البحث في فرع الشجرة الايسر, واذا كانت اكبر من قيمة المفتاح يستمر البحث في فرع الشجرة الايمن, تستمر العملية لحين الوصول الى الموقع الفارغ في فرع الشجرة وتتم اضافة البيانات. خوارزمية الاضافة هي:

1. اذا كانت الشجرة خالية (اذا كان الجذر يساوي null), يتم خلق عقدة لتكون هي الجذر.
2. اذا كان هناك جذر, تتم عملية مقارنة البيانات مع المفتاح (بيانات العقدة).
3. يتم عمل حلقة تكرار لإيجاد الموقع المطلوب لعملية الاضافة.

A. اذا كانت البيانات المطلوب اضافتها اكبر من قيمة المفتاح, يتم الذهاب الى فرع الشجرة الايمن
B. بخلاف ذلك, أي عندما تكون البيانات اصغر من قيمة المفتاح فيتم الذهاب الى فرع الشجرة الايسر.

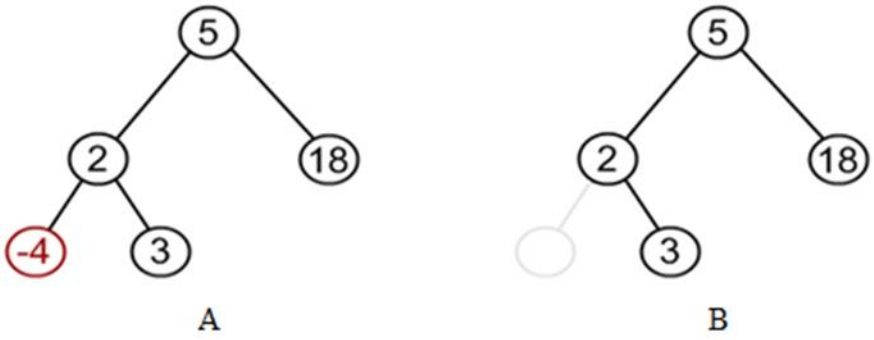
4. عند انتهاء حلقة التكرار يتم اضافة البيانات.

8.8.3 عملية حذف عنصر Removing an Element

عملية ازالة او حذف عنصر من شجرة البحث الثنائي هي عملية معقدة نسبة للعمليات الاخرى. حيث من المفروض بعد الحذف يجب ان تبقى الشجرة بشكل مرتب.

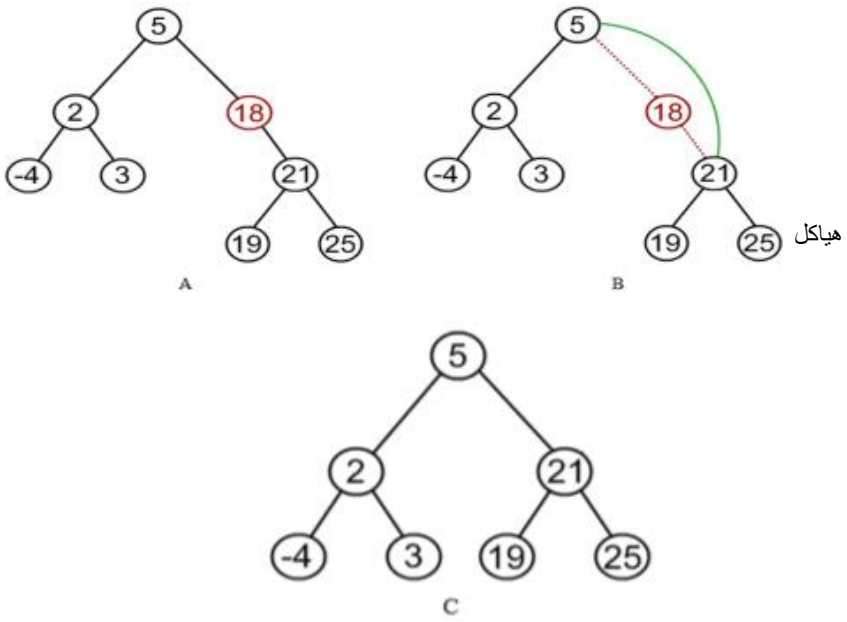
ان اول خطوة قبل عملية ازالة عنصر من الشجرة هو ايجاد هذا العنصر. وهذا يتم تحديده من خوارزمية البحث. بعد ايجاد العنصر المطلوب هناك ثلاث حالات:

A. اذا العقدة المراد حذفها هي ورقة (العقدة التي ليس لها ابناء), في هذه الحالة فان الخوارزمية ستؤشر على الرابط المقابل الذي يربط العقدة مع الاب بالقيمة null (اي ان العنوان الذي يشير للعقدة المراد حذفها في حقل العنوان للعقدة السابقة يجب ان يشير الى null). فمثلا اذا اردنا حذف القيمة (4-) من الشجرة في الشكل 8.26-A فان الناتج سيكون في الشكل 8.26-B (حيث سيتم وضع null في حقل العنوان الايسر للعقدة التي تحمل القيمة 2).



شكل 8.26: A. شجرة ثنائية مؤشر فيها العقدة المطلوب حذفها. B. الشجرة بعد اجراء عملية الحذف.

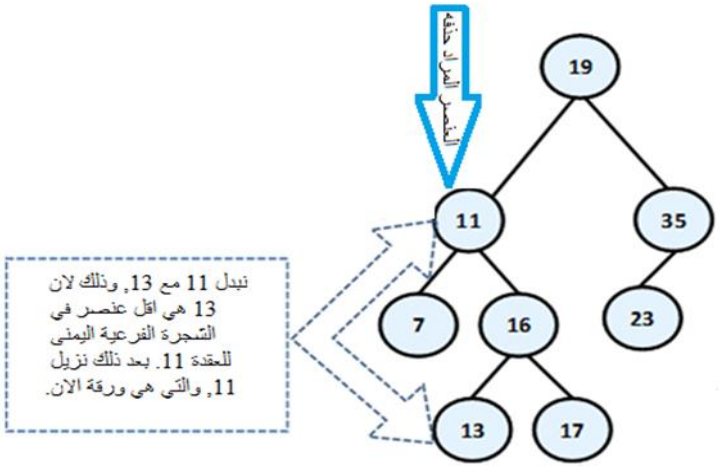
B. اذا العقدة لها فقط شجرة فرعية واحدة, شجرة فرعية يمنى او يسرى (أي لها ابن واحد), في هذه الحالة تحذف هذه العقدة من الشجرة, والخوارزمية ستربط الابن الوحيد لهذه العقدة (حتى لو كان يمثل جذر لشجرة فرعية) مع العقدة التي تمثل اب العقدة التي تم حذفها. مثال في الشكل 8.27 مطلوب حذف العقدة التي لها القيمة 18.



شكل 8.27: A. شجرة ثنائية مؤشر عليها العقدة المطلوب حذفها. B. طريقة ربط العقد بعد الحذف. C. الشجرة بعد الحذف.

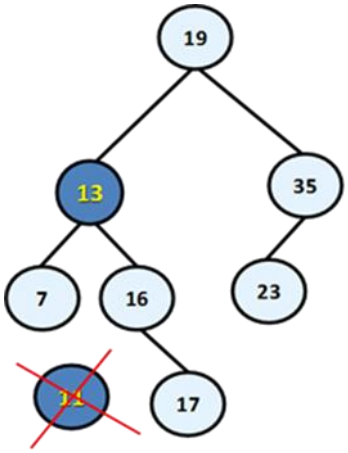
الحالة الثالثة: ان تكون العقدة المطلوب حذفها لها اثنان من الابناء (من الممكن ان يكون الابناء جذور لأشجار فرعية), في هذه الحالة يجب ان نجد العقدة الاصغر في الشجرة الفرعية اليمنى للعقدة المراد حذفها ونبدلها مع قيمة العقدة المراد حذفها. بعد هذا التبديل فان العقدة سيكون لها شجرة فرعية واحدة على الاكثر, عندها نحذف العقدة وفق القواعد او الحالات الاثنان اعلاه. هنا سنقول من الممكن عمل تبديل تناظري, أي نعمل نفس الشيء عند العمل على الشجرة الفرعية اليسرى وتكون عملية التبديل هنا مع اكبر عنصر في فرع الشجرة الايمن. في الشكل 8.28 مطلوب حذف العقدة التي تحمل القيمة 11.

يرجى ملاحظة ان العقدة المطلوب حذفها ستبقى بمكانها اثناء ابدال القيم لذلك فاننا عندما نبدل القيم لا يعني ان نحذف العقدة التي اصبحت تحمل القيمة المطلوب حذفها وانما حذف العقدة التي كانت تحمل تلك القيمة بمعنى عندما نقول مطلوب حذف العقدة التي تحمل القيمة 11 وتم ابدال القيمة 11 من هذه العقدة فان هذا لايغير الفكرة لان العقدة هي ذاتها مطلوب حذفها وان حملت قيمة اخرى.



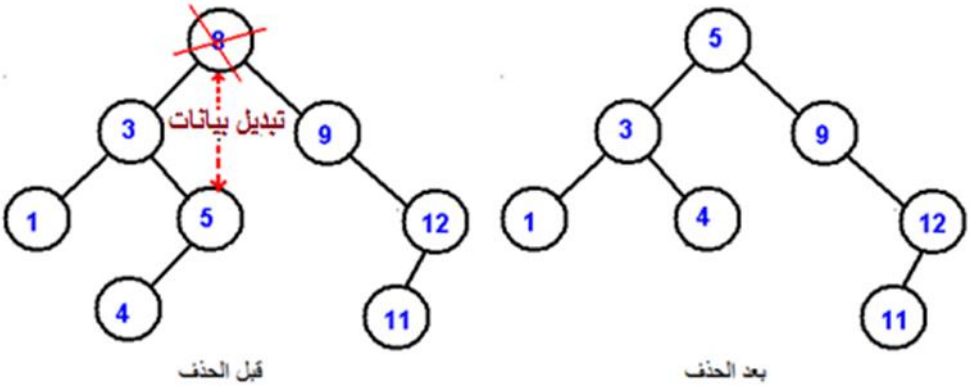
شكل 8.28: شجرة ثنائية محدد عليها العقدة المطلوب حذفها والعقدة التي يتم الابدال معها.

العقدة 11 لها شجرتين فرعيتين ووفقا لخوارزميتنا, فانه يجب ان يتم التبديل مع اصغر عنصر من الشجرة الفرعية اليمنى.. وهي القيمة 13. بعد عملية التبديل, فاننا من الممكن ان نزيل 11 (هي ورقة الان). وستكون النتيجة النهائية كما في الشكل 8.29.



شكل 8.29: الشجرة بعد اكمال عملية الحذف.

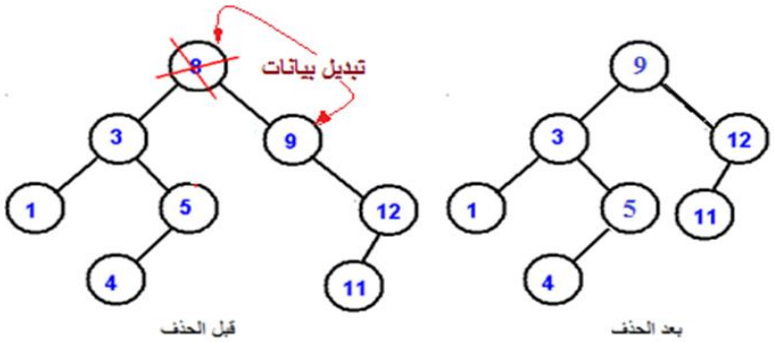
❖ مثال اخر في الشكل 8.30 مطلوب حذف العقدة التي لها القيمة 8 وهي جذر.



شكل 8.30: شجرة ثنائية قبل وبعد حذف القيمة 8 من فرع الشجرة اليسرى.

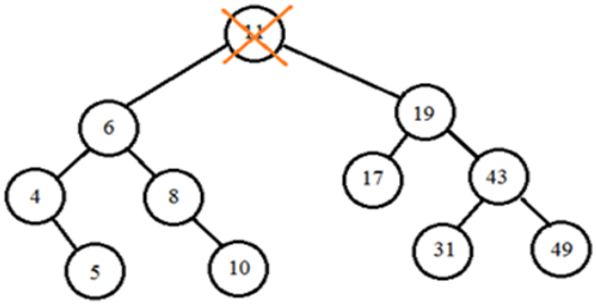
ان طريقة الحذف تتبع مايلي: ابدال العقدة المراد حذفها مع العقدة التي لها اكبر قيمة في الشجرة الفرعية اليسرى وبعدها يتم حذف العقدة التي لها القيمة الكبيرة.

من الممكن ايضا وبنفس الطريقة ان تبدل المفتاح (القيمة 8) مع العقدة التي لها اصغر قيمة في فرع الشجرة اليمنى, ويتم حذف العقدة التي لها القيمة الاصغر كما في الشكل 8.31.

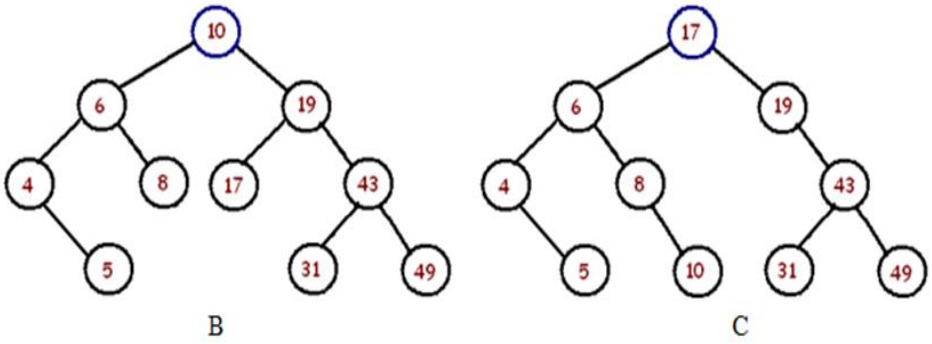


شكل 8.31: شجرة ثنائية قبل وبعد حذف القيمة 8 من فرع الشجرة اليمنى.

❖ مثال اخر في الشكل (8.32-A) مطلوب حذف العقدة التي لها القيمة 11. في هذه الحالة سيتم الابدال مع اكبر قيمة في فرع الشجرة الايسر والتي هي 10, بعدها يتم حذف العقدة كما في الشكل (8.32-B). كذلك من الممكن ان نعمل على فرع الشجرة الايمن ونبحث عن اصغر قيمة في هذا الفرع وهي القيمة 17 ويتم اجراء التبديل ثم حذف العقدة كما في الشكل (8.32-C).



A



شكل 8.32: A. شجرة ثنائية محدد بها العقدة المطلوب حذفها. B. الشجرة بعد الحذف, تم استخدام فرع الشجرة الايسر للتبديل. C. الشجرة بعد الحذف, تم استخدام فرع الشجرة الايمن للتبديل.

8.8.4 عملية المرور Traversal

هي زيارة كل عقدة في الشجرة وربما يتم طباعة قيمها. وبسبب ان كل العقد مرتبطة بروابط او مسارات, لذلك يجب ان تكون البداية من الجذر. مع ملاحظة انه من غير الممكن الوصول عشوائيا الى اي عقدة في الشجرة.

هناك ثلاث طرق تستخدم لزيارة الشجرة:

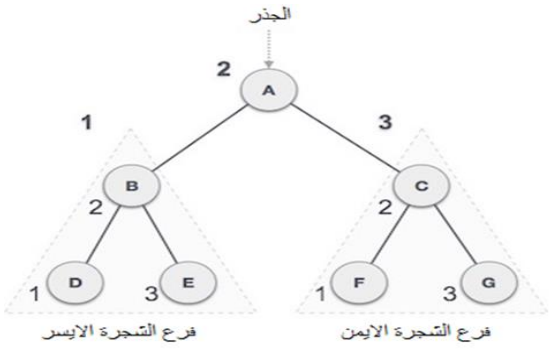
- In-order Traversal المرور بالترتيب الاعتيادي
- Pre-order Traversal المرور بالترتيب القبلي
- Post-order Traversal المرور بالترتيب البعدي

A. المرور بالترتيب الاعتيادي

في هذه الطريقة فان الزيارة تبدا بفرع الشجرة الايسر, بعدها الجذر ومن ثم فرع الشجرة الايمن. علينا ان نتذكر دائما بان كل عقدة في الشجرة تمثل فرع شجرة

بنفسها. عند المرور على الشجرة الثنائية بالترتيب الاعتيادي, فان المخرجات ستكون عبارة عن قيم (او مفاتيح) مرتبة بترتيب تصاعدي.

ايسر – جذر – ايمن



شكل 8.33: شجرة ثنائية تتم زيارتها بطريقة الترتيب الاعتيادي وحسب الترقيم الموضح في الشكل.

البداية تكون من الجذر A, وباتباع طريقة الترتيب الاعتيادي ستكون الحركة لليسار الى فرع الشجرة B. فرع الشجرة B هو ايضا جذر لذلك تستمر الحركة لليسار (لانه ترتيب اعتيادي). بعد اكمال فرع الشجرة الايسر تتم زيارة الجذر لفرع الشجرة ومن ثم زيارة فرع الشجرة الايمن لهذا الفرع. هذه العملية تستمر لحين ان تتم زيارة كل العقد. مخرجات هذه الطريقة هو

D → B → E → A → F → C → G

خوارزمية هذه الطريقة هي:

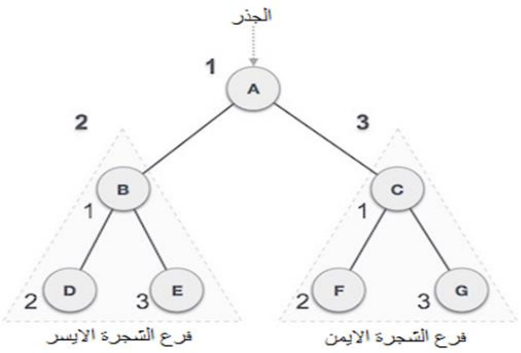
1. زيارة فرع الشجرة الايسر بطريقة التكرار والاستدعاء الذاتي.
2. زيارة الجذر.

3. زيارة فرع الشجرة الايمن بطريقة التكرار والاستدعاء الذاتي.

B. الزيارة بطريقة الترتيب القبلي Preorder Travers

في هذه الطريقة فان الجذر سيتم زيارته اولا, بعدها تتم زيارة فرع الشجرة الايسر, واخيرا زيارة فرع الشجرة الايمن.

جذر – ايسر – ايمن



شكل 8.34: شجرة ثنائية تتم زيارتها بطريقة الترتيب القبلي وحسب الترقيم الموضح في الشكل.

تكون البداية من الجذر A, وباتباع طريقة الترتيب القبلي فان الجذر سيكون اول عقدة تتم زيارتها, بعدها يتم الانتقال الى فرع الشجرة الايسر. كذلك فان العقدة B ستعتبر جذر ويتم زيارتها وبعدها يتم الانتقال الى اليسار. هذه العملية تستمر لحين زيارة جميع العقد. مخرجات هذه الطريقة هي:

A → B → D → E → C → F → G

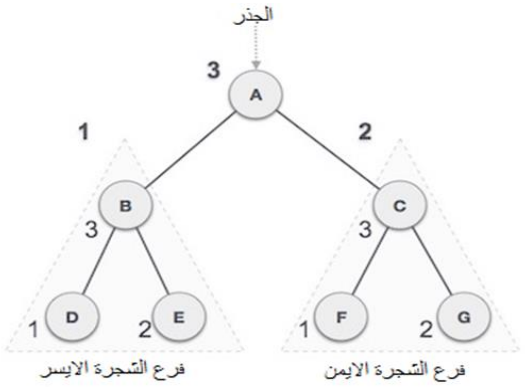
خوارزمية هذه الطريقة هي:

1. زيارة عقدة الجذر.
2. زيارة فرع الشجرة الايسر بطريقة التكرار والاستدعاء الذاتي.
3. زيارة فرع الشجرة الايمن بطريقة التكرار والاستدعاء الذاتي.

C. الزيارة بطريقة الترتيب البعدي Postorder Travers

في هذه الطريقة فان الجذر يتم زيارته اخيرا, ولذلك جاء الاسم الترتيب البعدي. في البداية تتم زيارة فرع الشجرة الايسر, ثم زيارة فرع الشجرة الايمن, واخيرا تتم زيارة الجذر.

ايسر – ايمن - جذر



شكل 8.35: شجرة ثنائية تتم زيارتها بطريقة الترتيب البعدي وحسب الترقيم الموضح في الشكل.

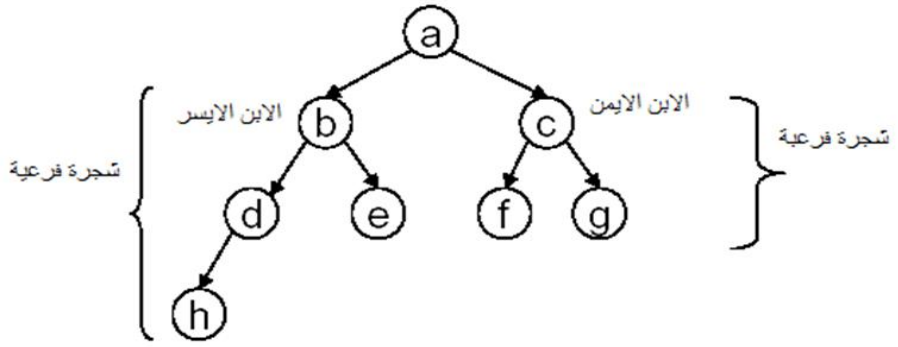
البداية من العقدة A, وباتباع طريقة الترتيب البعدي فسيكون المسار الى فرع الشجرة الايسر B. ايضا العقدة B تعتبر جذر فيتم الاستمرار الى اليسار. بعد اكمال زيارة فرع الشجرة الايسر تتم زيارة فرع الشجرة الايمن لاي جذر, تستمر هذه العملية لحين المرور على كامل العقد في الشجرة. نتيجة هذه الطريقة هي:

D → E → B → F → G → C → A

خوارزمية الترتيب البعدي هي:

1. زيارة فرع الشجرة الايسر بطريقة التكرار والاستدعاء الذاتي للدالة.
2. زيارة فرع الشجرة الايمن بطريقة التكرار والاستدعاء الذاتي للدالة.
3. زيارة الجذر.

مثال: في الشكل 8.36 شجرة ثنائية المطلوب المرور على عناصرها باستخدام طرق المرور الثلاث.



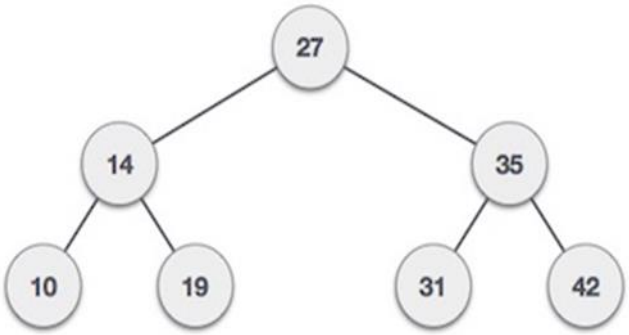
شكل 8.36: شجرة ثنائية.

Preorder الترتيب القبلي: a b d h e c f g

Inorder الترتيب الاعتيادي: h d b e a f c g

Postorder الترتيب البعدي: h d e b f g c a

مثال: في الشكل 8.37 شجرة ثنائية المطلوب المرور على عناصرها باستخدام خوارزميات الترتيب الثلاث.



شكل 8.37: شجرة ثنائية.

الترتيب الاعتيادي : 10, 14, 19, 27, 31, 35, 42

الترتيب القبلي : 27, 14, 10, 19, 35, 31, 42

الترتيب البعدي : 10, 19, 14, 31, 42, 35, 27

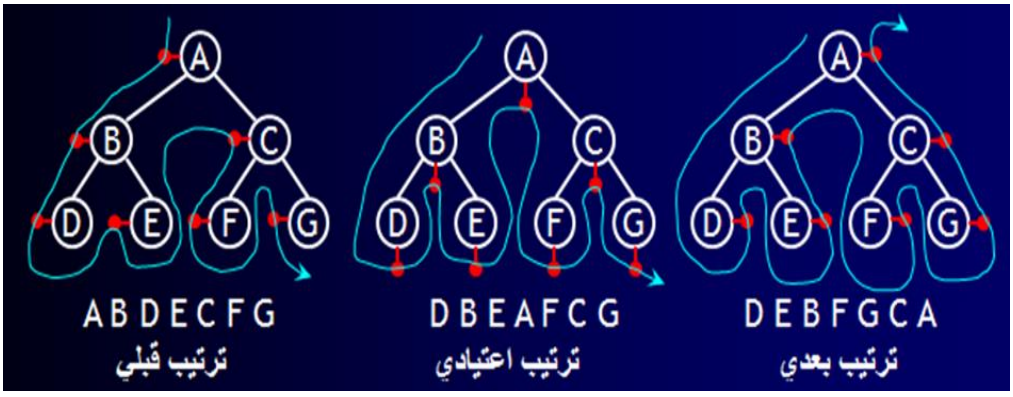
من الممكن ان نفهم او نتابع طريقة المرور من خلال الاشكال (وهي طريقة تستخدم العلامات), لا تختلف عن الطرقية السابقة الا انها ربما تسهل الفهم.

هي ببساطة ان نضع علامة على عقدة واحدة تمثل طريقة الترتيب كما في الشكل 8.38.

شكل 8.38 : عقد مفردة توضح كيفية المرور عليها باستخدام علامة تمثل الجذر.

في الشكل 8.38 نلاحظ الاشارة الحمراء والتي تشير الى الجذر والفرع من الدائرة تشير الى الابن الايسر والايمن ويكون متابعة المرور من اليسار الى اليمين لكل عقدة (العلامة الحمراء تشير الى موقع الجدر نسبة للفروع عند المرور باي ترتيب). فمثلا بالترتيب القبلي فان الاشارة الحمراء هي في اقصى اليسار وهذا يعني انها ستكون اول من يتم زيارته ثم الابن الايسر واخيرا الابن الايمن. في الترتيب الاعتيادي فان العلامة الحمراء هي في الوسط وهذا يعني ان الجذر سيكون في الوسط, حيث نبدا من اليسار وهنا سيكون الابن الايسر ثم ياتي بعدها الجذر المعلم بالعلامة الحمراء واخيرا الابن الايمن. وهكذا للترتيب البعدي حيث ان العلامة الحمراء تشير الى اليمين وهذا يعني ان الجذر سيكون باليمين ولذلك فان الترتيب يبدا بالعقدة اليسرى ثم اليمنى وبعدها الجذر.

هذه العملية يمكن اجرائها لجميع عقد الشجرة وحسب طريقة المرور فمثلا اذا كان المطلوب مرور بالترتيب القبلي فيتم تعليم جميع العقد بنفس الطريقة التي قمنا بها في الشكل 8.38 وعندها يتم المرور عليهم جميعا بسهولة من اليسار الى اليمين, كما هو موضح في الشكل 8.39.



شكل 8.39: توضيح لطريقة المرور باستخدام التعليم.

8.8.5 برنامج شامل لاغلب العمليات التي تجرى على الاشجار الثنائية

```
#include<iostream>
#include<cstdlib>
using namespace std;

struct tree {
   int info;
   tree *Left, *Right;
};

tree *root;
class Binary_tree{

   public:
      Binary_tree();
      void insert1(int);
      tree *insert2(tree *, tree *);
      void Delete(int);
      void pretrav(tree *);
      void intrav(tree *);
      void posttrav(tree *);
};
```

```
Binary_tree::Binary_tree()  {
   root = NULL;
}

tree* Binary_tree::insert2(tree *temp,tree *newnode) {
   if (temp==NULL) {
      temp=newnode;
   }

   else   if (temp->info < newnode->info) {

      insert2(temp->Right, newnode);

      if(temp->Right = =NULL)

         temp->Right = newnode;

   }

   else{
      insert2(temp->Left,newnode);
      if(temp->Left==NULL)
```

```
            temp->Left=newnode;
        }
        return temp;
}
void Binary_tree::insert1(int n){
        tree *temp=root,*newnode;
        newnode=new tree;
        newnode->Left=NULL;
        newnode->Right=NULL;
        newnode->info=n;
        root=insert2(temp,newnode);
}

void Binary_tree::pretrav(tree *t = root){
        if(root == NULL){
            cout<<"Nothing to display";
        }else

        if(t != NULL){

            cout<<t->info<<" ";
            pretrav(t->Left);
            pretrav(t->Right);
```

```
    }
}

void Binary_tree::intrav(tree *t = root){

    if(root==NULL){
        cout<<"Nothing to display";
    }       else

    if(t!=NULL){
        intrav(t->Left);
        cout<<t->info<<" ";
        intrav(t->Right);
    }
}

void Binary_tree::posttrav(tree *t = root){
    if(root==NULL){
        cout<<"Nothing to display";
    }   else
        if(t!=NULL){
        posttrav(t->Left);
```

```
        posttrav(t->Right);
        cout<<t->info<<" ";
    }
}

void Binary_tree::Delete(int key)
{
    tree *temp = root,*parent = root, *marker;
    if (temp==NULL)
        cout<<"The tree is empty"<<endl;
    else {
        while (temp!=NULL && temp->info!=key){
            parent=temp;
            if (temp->info<key){
                temp=temp->Right;
            }   else    {

                temp=temp->Left;
            }
        }
    }
    marker=temp;
    if (temp==NULL)
```

```
    cout<<"No node present";
else if (temp==root){
    if (temp->Right = =NULL && temp->Left = =NULL) {

        root=NULL;
    }
    else if (temp->Left==NULL){
        root=temp->Right;
    }
    else if (temp->Right==NULL){
        root=temp->Left;
    }
    else {
        tree *temp1;
        temp1 = temp->Right;
        while (temp1->Left!=NULL){
            temp=temp1;
            temp1=temp1->Left;
        }
        if (temp1!=temp->Right){
            temp->Left=temp1->Right;
            temp1->Right=root->Right;
```

```
        }
        temp1->Left=root->Left;
        root=temp1;
    }

}
else{
    if (temp->Right==NULL && temp->Left==NULL){
        if (parent->Right==temp)
            parent->Right=NULL;
        else
            parent->Left=NULL;
    }
    else if (temp->Left==NULL){
        if (parent->Right==temp)
            parent->Right=temp->Right;
        else
            parent->Left=temp->Right;
    }
    else if (temp->Right==NULL){
        if (parent->Right==temp)
            parent->Right=temp->Left;
        else
```

```
            parent->Left=temp->Left;
        }else    {
            tree *temp1;

            parent=temp;
            temp1=temp->Right;
            while (temp1->Left!=NULL){
                parent=temp1;
                temp1=temp1->Left;
            }
            if (temp1!=temp->Right){
                temp->Left=temp1->Right;
                temp1->Right=parent->Right;
            }
            temp1->Left=parent->Left;
            parent=temp1;
        }
    }
    delete marker;
}
int main(){
    Binary_tree bt;
```

```
int choice, n, key;
while (1){
cout<<"\n\t1. Insert\n\t2. Delete\n\t3. Preorder Traversal\n\t4. Inorder Treversal\n\t5. Postorder Traversal\n\t6. Exit" <<endl;

cout<<"Enter your choice: ";
cin>>choice;
switch(choice){
case 1:
cout<<"Enter item: ";
cin>>n;
bt.insert1(n);
break;
case 2:
cout<<"Enter element to delete: ";
cin>>key;
bt.Delete(key);
break;
case 3:
cout<<endl;
bt.pretrav();
break;
case 4:
```

```
            cout<<endl;
            bt.intrav();
            break;

        case 5:
            cout<<endl;
            bt.posttrav();
            break;
        case 6:
            exit(0);
      }
   }
   return 0;
{
```

8.9 اشجار التعابير الرياضية Expression Tree

المترجم في الحاسوب يستخدم الاشجار الثنائية لتمثيل التعابير الرياضية. هذه الاشجار الثنائية التي تستخدم العلاقات الرياضية تسمى اشجار تعابير, فهي اشجار تحتوي مجموعة من العقد وكل عقدة تحتوي على عمليات ثنائية (وهي العمليات التي تنفذ على اثنان من المعاملات مثل عمليات الجمع والطرح وغيرها) او معاملات (وهي القيم التي تنفذ عليها العمليات الرياضية).

كما سبق وان وضحنا في فصل المكدس فان العلاقات الرياضية من الممكن ان نمثلها بثلاث طرق وهي (infix, prefix, and postfix) وهذه جميعا من الممكن

ان نمثلها في شجرة ثنائية كما سبق وان وضحنا في هذا الفصل. سنحاول توضيح كيفية تحويل هذه التعابير الى اشجار ثنائية وكذلك كيف من الممكن قراءة شجرة تعبير رياضي. في الجدول 8.1 توضيح للتعابير الرياضية التي سبق وان تم شرحها في فصل المكدس.

جدول 8.1: توضيح للتعابير الرياضية باشكالها الثلاث.

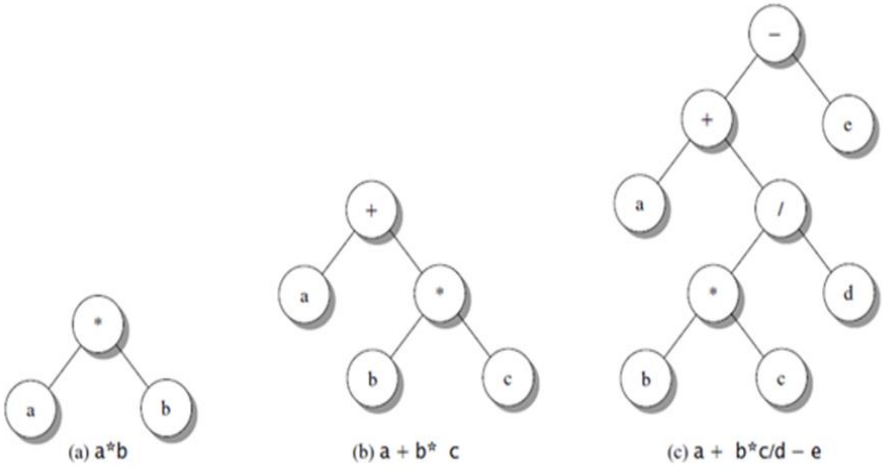
Infix	Postfix	Prefix
a*b	ab*	*ab
a+b*c	abc*+	+a*bc
a+b*c/d–e	abc*d/+e–	–+a/*bcde

نفرض ان تعبير رياضي يحتوي على عوامل الجمع (+), الطرح (-), الضرب (*), والقسمة (/) في شجرة التعبير, عليه فان كل عامل رياضي لديه اثنان من الابناء اما ان تحتوي هذه الابناء معامل, او تعبير ثانوي (شجرة فرعية).

شجرة التعبير الثنائي تتكون من :

- عقد الاوراق والتي تحتوي على معامل مفرد (من الممكن ان تكون قيم, او حروف او حتى نصوص وكلمات ...الخ).
- عقد ليست ورقية والتي تحتوي عوامل رياضية ثنائية مثل (+, -, *, /, %).
- اشجار فرعية يمين ويسار العامل الرياضي, توصف على انها تعبير فرعي او ثانوي.

لاحظ الشكل 8.40 الذي يصف التعابير الرياضية.



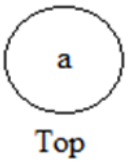
شكل 8.40: اشجار التعابير الرياضية.

ملاحظة: في شجرة التعبير فان كل عامل رياضي هو عقدة داخلية interior والتي لها ابناء, الابناء اما معاملات او تعابير ثانوية. ودائما المعاملات هي في العقد الورقية.

الخطوات التالية توضح طريقة تنفيذ خوارزمية بناء شجرة تعبير للتعبير الرياضي a+b*c على شكل ترتيب بعدي. تذكر ان التعامل مع التعبير الرياضي سيتم عن طريق المكدس كما سبق ووضحنا في فصل المكدس. هنا للسهولة سوف نمثل المكدس على شكل افقي (هو بكل الحالات تمثيل افتراضي). تذكر ان كل عنصر هو شجرة ثانوية توصف بجذرها. اذن التعبير البعدي للعلاقة يكون:

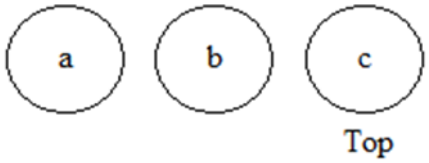
a b c * +

1. الخطوة الاولى: يتم تمييز الحرف a كمعامل. نخلق عقدة ورقية تحتوي قيمة الحرف او الرمز a, يدفع في المكدس s.



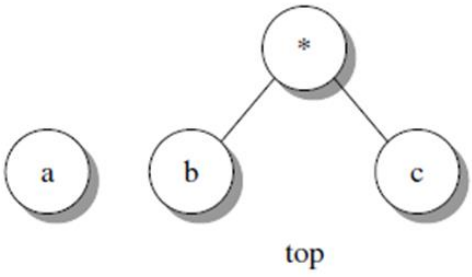
شكل 8.41: بداية المكدس.

2. الخطوة الثانية, والثالثة: يتم تمييز الحرفين b, c كمعاملات, ويتم تكوين عقد ورقية وتدفع الى المكدس.



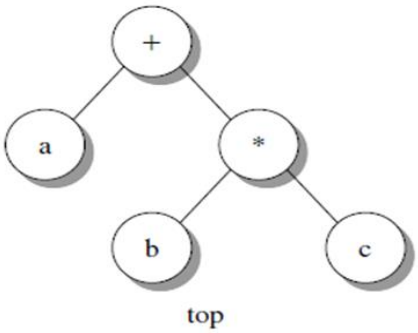
شكل 8.42: المكدس بعد اضافة اثنان من المعاملات.

3. الخطوه الرابعة: يتم تمييز * كعامل رياضي, في هذه الحالة لايدفع للمكدس لانه عامل رياضي وانما نعمل على خلق عقدة جديدة على ان تكون قيمتها الرمز (*). يتم سحب اثنان من العقد الموجودة في اعلى المكدس كما سبق وتعلمنا في فصل المكدس (العقده c والعقدة b), هذه ستكون الاشجار الثانوية اليمين واليسار للعقدة الجديدة (*) على التوالي كما في الشكل 8.43. تربط الاشجار الثانوية وتدفع الشجرة الثانوية المتكونة التي جذرها * الى المكدس.



شكل 8.43: المكدس بعد تنفيذ العامل الرياضي (*).

4. الخطوه الخامسة: يتم تمييز الرمز(+) كعامل. يتم خلق عقدة جديدة تكون قيمتها هي (+), هنا يتم سحب العقدتين الثانويتين من اعلى المكدس وتربطان الى هذه العقدة الجديدة ثم يعاد دفع العقدة المتكونة الى المكدس والتي جذرها هو (+) كما في الشكل 8.44.



شكل 8.44: المكدس بعد اضافة العامل الرياضي (+).

5. الخطوه السادسة: عند الوصول الى اخر عنصر في التعبير, فان العنصر الوحيد المتبقي في المكدس هو جذر شجرة التعبير.

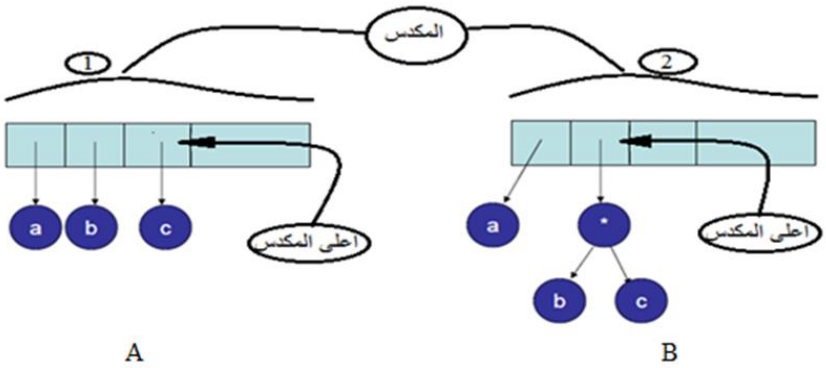
8.9.1 خوارزمية بناء شجرة التعبير الرياضي

1. اذا كان الرمز هو معامل تخلق شجرة بعقدة واحدة ويدفع مؤشرها الى المكدس.
2. اذا كان الرمز هو عامل رياضي فيعامل هذا الرمز على انه جذر, يربط هذا الجذر باثنين من الموجودات في اعلى المكدس سواء كانت (عقد او اشجار فرعية) ويتم تكوين شجرة فرعية جديدة وتعاد الى اعلى المكدس.
3. اعلى المكدس (عقدة او شجرة فرعية) سيتم سحبه اولا ويربط بالجذر في الجانب الايمن ليكون الابن الايمن, بينما الذي يليه يربط بالجذر على انه الابن الايسر.

مثال: مطلوب بناء شجرة تعبير للتعبير الرياضي (abc*+de*f+g*+)

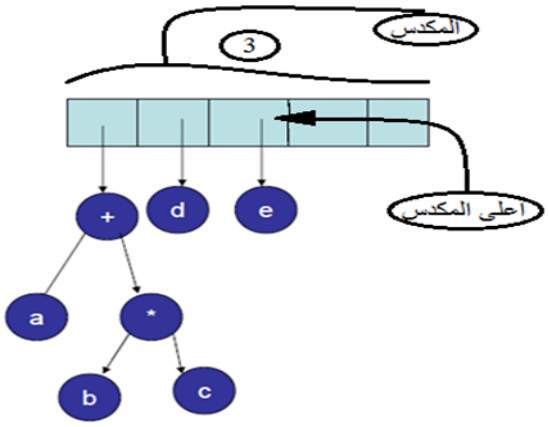
الخطوة الاولى: العوامل يتم دفعها الى المكدس, لذلك يتم اولا دفع العامل (a) الى المكدس, ثم دفع العامل (b), وبعدها يتم دفع العامل (c) الى المكدس كما في الشكل (8.45-A).

الخطوة الثانية: الان الدور للعامل الرياضي (*), في هذه الحالة سيتم سحب عنصرين من اعلى المكدس ويربطان بواسطة العامل الرياضي (*), ثم يعاد الناتج الى اعلى المكدس كما في الشكل (8.45-B).



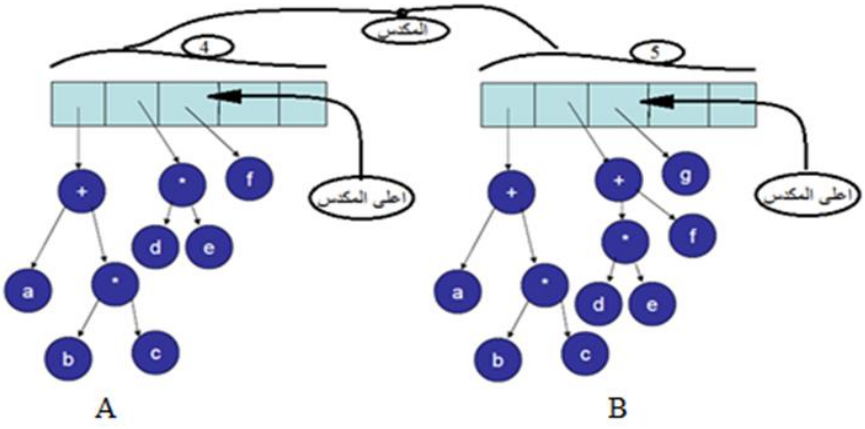
شكل 8.45: وضع المكدس بعد اضافة عدد من المعاملات والعامل (*).

الخطوة الثالثة: الان جاء دور المعامل (d) فيتم دفعه الى المكدس, ثم ياتي دور المعامل (e) فيتم ايضا دفعه الى المكدس كما في الشكل 8.46.



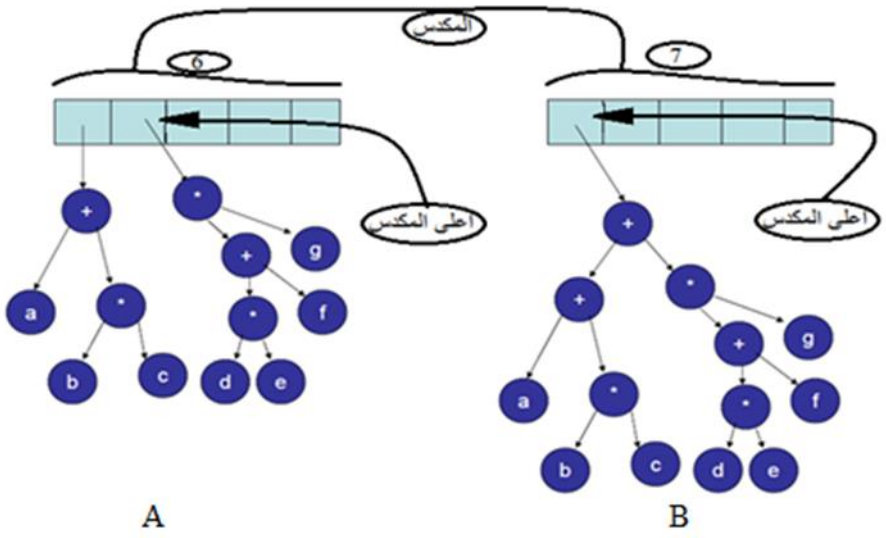
شكل 8.46: المكدس بعد اضافة عدد من المعاملات.

الخطوة الرابعة: في هذه الخطوة سيتم تنفيذ العامل الرياضي (*) , ودائما العوامل الرياضية تتطلب سحب اثنان من المعاملات في اعلى المكدس ويربطان بالعامل الرياضي (*), نؤكد على ان العنصر الذي في اعلى المكدس يكون الابن الايمن للجذر بينما الذي يليه يكون الابن الايسر للجذر, بعد انجاز الربط يعاد فرع الشجرة المتولدة الى المكدس, بعدها ياتي دور المعامل (f) والذي يدفع الى المكدس مباشرة, كما في الشكل 8.47-A. ثم ياتي دور العامل (+) والمعامل (g) شكل 8.47-B.



شكل 8.47: المكدس بعد عدد من عمليات الاضافة.

تستمر العملية لحين اكمال جميع عناصر التعبير الرياضي وبذلك سيكون الناتج من هذه العملية في المكدس وهو عبارة عن شجرة تعبير كما في الشكل 8.48-B.

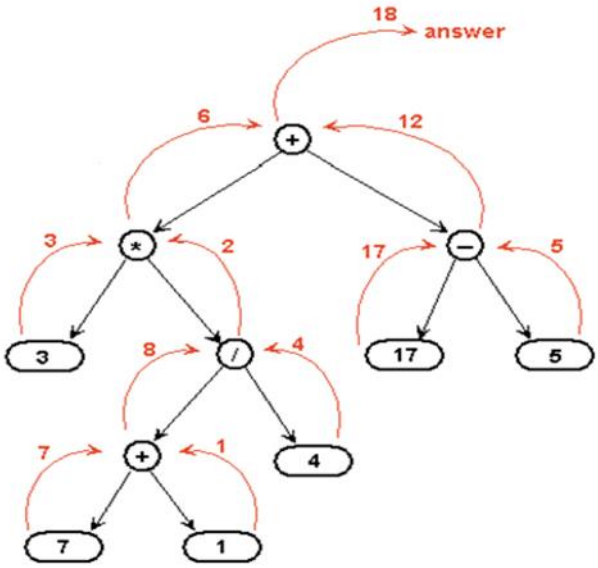


شكل :8.48 A. المكدس بعد عدد من الاضافات. B. المكدس بعد اكمال جميع عناصر التعبير الرياضي.

مثال: الشكل 8.49 هو شجرة تمثل التعبير الرياضي

3 * (7 + 1) / 4 + (17 – 5)

الاسهم الحمراء توضح كيفية حساب قيم التعبير بطريقة الترتيب الاعتيادي.



شكل 8.49: شكل توضيحي لطريقة قراءة شجرة تعبير.

8.9.2 برنامج لتنفيذ العمليات على شجرة التعبير

```
#include <iostream.h>
#include <conio.h>

        struct node {
          char data;
          node *left;
          node *right;
       };

    char postfix[35];
    int top=-1;
    node *arr[35];


    int  r (char inputchar){     //لفحص الرمز اذا كان معامل او عامل رياضي
    if (inputchar = = '+' || inputchar = = '-' || inputchar = = '*' || inputchar =
    = '/')
         return (-1);
```

```
    else if (inputchar>= 'a' || inputchar <= 'z')
     return(1);
    elseif(inputchar>= 'A' || inputchar <= 'Z')
     return(1);
    elsereturn(-99);  // في حالة الخطأ
}

//تستخدم لاضافة عنصر مفرد الى المصفوفة
void push(node *tree){
   top++;
   arr[top]=tree;
}

node *pop(){
   top--;
   return(arr[top+1]);
}


void create_expr_tree(char *suffix){
   char symbol;
   node *newl,*ptr1,*ptr2;

   int flag; // مؤشر تكون قيمته (-1) عندما يكون عامل و (1) عندما يكون معامل

   symbol = suffix[0]; //  قراءة الرمز الاول من الترتيب البعدي

  expr.for(int i=1;symbol!=NULL;i++){ // الاستمرار لغاية نهاية التعبير

  flag= r (symbol); // فحص الرمز اذا كان عامل او معامل

if (flag == 1)// اذا الرمز معامل
      {
         newl = new node;
         newl->data = symbol;
         newl->left = NULL;
         newl->right = NULL;
         push(newl);
      }
      else{    // اذا الرمز عامل سحب اثنين من المعاملات
```

```
            ptr1=pop();
            ptr2=pop();
            newl = new node;
            newl->data = symbol;
            newl->left = ptr2;
            newl->right = ptr1;
            push(newl);
         }
         symbol=suffix[i];
      }
}

void preOrder(node *tree){
   if ( tree!=NULL){
      cout<<tree->data;
      preOrder(tree->left);
      preOrder(tree->right);
   }
}

void inOrder(node *tree){
   if ( tree!=NULL){
      inOrder( tree->left);
      cout<< tree->data;
      inOrder(tree->right);
   }
}


void postOrder(node *tree){
   if ( tree!=NULL){
      postOrder( tree->left);
      postOrder( tree->right);
      cout<<tree->data;
   }
}


void main(){
```

```
    clrscr();
    cout<<"*****Expression Tree*****\n";
    cout<<"Enter Postfix Expression : ";
    cin>>postfix;

    // خلق شجرة تعبير رياضي
    create_expr_tree(postfix);

    // عملية المرور على شجرة التعبير
    cout<<"\n In-Order Traversal :   ";
    inOrder(arr[0]);

    cout<<"\n Pre-Order Traversal :  ";
    preOrder(arr[0]);

    cout<<"\n Post-Order Traversal : ";
    postOrder(arr[0]);
    getch();
}
```

8.10 تمثيل الاشجار الثنائية Binary Tree Representations

الاشجار الثنائية في هياكل البيانات تمثل بطريقتين. هذه الطرق هي:

- ❖ التمثيل على شكل مصفوفات.
- ❖ التمثيل على شكل قوائم موصولة.

8.10.1 تمثيل الشجرة الثنائية على شكل مصفوفة.

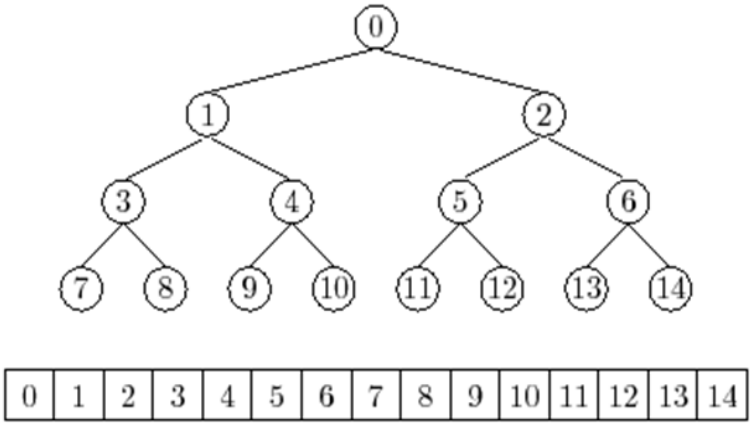
في التمثيل على شكل مصفوفة فاننا سنستخدم مصفوفة احادية البعد لتمثيل الشجرة الثنائية.

السؤال الان كيف من الممكن تمثيل الشجرة الثنائية باستخدام المصفوفات؟ في الحقيقة, هناك عدد من الطرق لعمل ذلك, سوف نناقش واحدة منها.

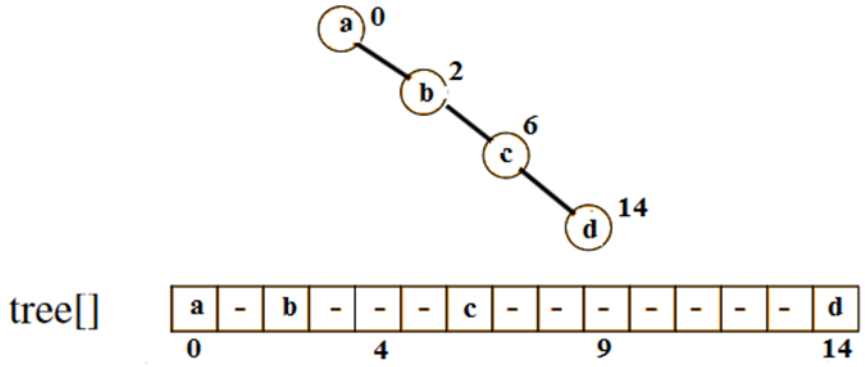
الاشجار الثنائية من الممكن ان تخزن بمصفوفة احادية في الذاكرة وفقا للقواعد التالية:

1. قيم الاشجار الثنائية تخزن في المصفوفة وتسمى المصفوفة شجرة (tree).
2. جذر الشجرة يخزن في الموقع الاول للمصفوفة (أي في الموقع 0, tree[0]).
3. الابن الايسر للشجرة يخزن في الموقع (2i + 1), بينما يخزن الابن الايمن في الموقع (2i + 2) (i يمثل الرقم التسلسلي index او موقع الاب في المصفوفة), لاحظ الشكل 8.50.
4. ان اكبر حجم لمصفوفة الشجرة هو (2^{d+1} -1), حيث ان d يمثل عمق الشجرة.
5. الشجرة الخالية او فرع الشجرة الخالي يحدد بالقيمة null. فاذا كانت الشجرة tree [0] = null فان هذا يعني ان الشجرة خالية.
6. لاحظ ان الشجرة التي لها (N) من العقد ليس دائما تشغل اول (N) من مواقع المصفوفة, كما في الشكل 8.51.
7. اذا كان موقع الابن في المصفوفة هو (i) فان موقع الاب في المصفوفة يتم حسابة باستخدام العلاقة ((i – 1) / 2) مع ملاحظة اهمال الكسر الناتج من القسمة.

من فوائد هذه الطريقة الخطية هو سهولة الوصول الى عناصر الشجرة وسهولة المرور على عناصر الشجرة, كذلك الكفاءة باستخدام المساحة الخزنية اذا كانت الشجرة كاملة. بالمقابل فان من مساوئ استخدام هذه الطريقة هو عدم الاستخدام الكفوء للمساحة الخزنية خصوصا اذا لم تكن الشجرة كاملة.



شكل 8.50: شجرة ثنائية محدد عليها ارقام المواقع لكل عقدة في المصفوفة.



شكل 8.51: شجرة منحرفة يمينا يتم خزنها في مصفوفة, لاحظ عدم خزن قيمها بمواقع متجاورة من بداية المصفوفة.

8.10.2 التمثيل على شكل قوائم موصولة.

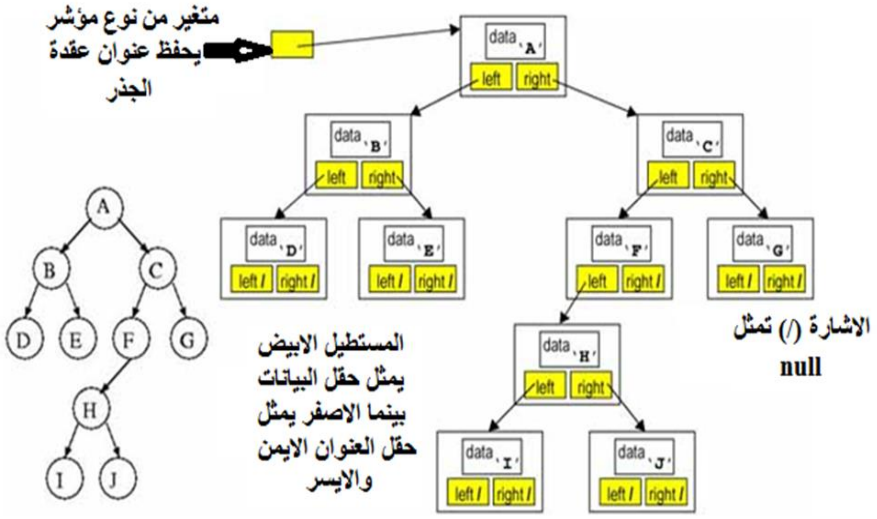
في هذه الحالة يتم استخدام قوائم موصولة ثنائية (double linked list) لتمثيل الشجرة الثنائية. في القوائم الموصولة الثنائية وكما عرفنا في فصل القوائم الموصولة فان كل عقدة تتكون من ثلاث حقول. الحقل الاول يخزن به عنوان الابن

الايسر, الحقل الثاني يخزن به البيانات الحقيقية والحقل الثالث يخزن به عنوان الابن الايمن.

في هذا التمثيل فان العقدة سيكون لها الهيكل التالي

عنوان الابن الايمن	البيانات	عنوان الابن الايسر

عملية تمثيل الشجرة على شكل قوائم موصولة موضحة بالشكل 8.52.



شكل 8.52: تمثيل الشجرة بطريقة القوائم الموصولة

8.10.3 شفرة برنامج لتنفيذ طريقة المصفوفات مع الاشجار الثنائية.

البرنامج ينفيذ اشجار البحث الثنائي بطرقة المصفوفات مع اجراء عدد من عمليات المرور التي تجرى على الاشجار.

ملاحظة: لايحبذ استخدام المصفوفات مع اشجار البحث الثنائي وذلك لانها تحتاج الى مساحة ذاكرة كبيرة. يفضل استخدام القوائم الموصولة والتي سبق وان تم كتابة شفرتها خلال هذا الفصل.

```
#include<iostream>
using namespace std;
public:
   int size;
   int* array;

   void insertElement(int x);
   void searchElement(int x);
   void inOrder(int currentIndex);
   void preOrder(int currentIndex);
   void postOrder(int currentIndex);
   void parent(int x);
   int extendSize(int x);

   BinarySearchTree (int size) {
      this -> size = extendSize(size);
      cout << this -> size << endl;
      this -> array = new int[this -> size];
      for (int x = 0; x < this -> size; x++){
         array[x] = NULL;
      }
   }
```

```
};

int BinarySearchTree::extendSize(int x) {
   int value = 0;
   for (int y = 0; y < x + 1; y++) {
      value = (2 * value) + 2;
   }
   return value;
}

void BinarySearchTree::insertElement(int x) {
   int currentIndex = 0;
   cout << "Adding: " << x;
   while(true) {
      if(array[currentIndex] == NULL){
         array[currentIndex] = x;
         cout << " Inserted at index: " << currentIndex << endl;
         break;
      }else        if(array[currentIndex] <= x) {
         if (array[currentIndex] == x){
            cout << "ERROR!-- Repeating element" << endl;
            break;
         }else
         cout << " Right ";
         currentIndex = (2 * currentIndex + 2);
      }else        if(array[currentIndex] >= x) {
```

```
        if (array[currentIndex] == x){
          cout << "ERROR!-- Repeating element" << endl;
          break;
        }else
        cout << " Left ";
        currentIndex = (2 * currentIndex + 1);
      }

   }
}

void BinarySearchTree::searchElement(int x){
   int currentIndex = 0;
   while (true) {
        if (array[currentIndex] == NULL) {
1.              cout << "Not Found" << endl;
        break;
        }
        if (array[currentIndex] == x) {
        cout << "Found at index: " << currentIndex << endl;
        break;
        }
        else      if(array[currentIndex] < x) {
        currentIndex = (2 * currentIndex + 2);
      }
        else        if(array[currentIndex] > x) {
        currentIndex = (2 * currentIndex + 1);
```

```
        }

    }

}

void BinarySearchTree::parent(int x){
    while (x != 0) {
        x = (x-1) / 2;
        cout << "---";
    }

}

void BinarySearchTree::inOrder(int currentIndex){
    if (array[currentIndex] != NULL) {
            inOrder(2 * currentIndex + 1);
            parent(currentIndex);
            cout << array[currentIndex] << endl;
            inOrder(2 * currentIndex + 2);

    }
}

void BinarySearchTree::postOrder(int currentIndex) {
    if (array[currentIndex] != NULL){
```

```
      postOrder(2 * currentIndex + 1);
      postOrder(2 * currentIndex + 2);
      parent(currentIndex);
      cout << array[currentIndex] << " " << endl;
   }

}

void BinarySearchTree::preOrder(int currentIndex) {
   if (array[currentIndex] != NULL) {
      preOrder(2 * currentIndex + 1);
      parent(currentIndex);
      cout << array[currentIndex] << " " << endl;
      preOrder(2 * currentIndex + 2);
   }
}

int main () {
   BinarySearchTree frank(5);
   frank.insertElement(4);
   frank.insertElement(6);
   frank.insertElement(9);
   frank.insertElement(3);
   frank.insertElement(2);
   frank.searchElement(1);
       frank.inOrder(0);
    };
```

الفصل التاسع
تكديس هياكل البيانات

HEAP DATA STRUCTURE

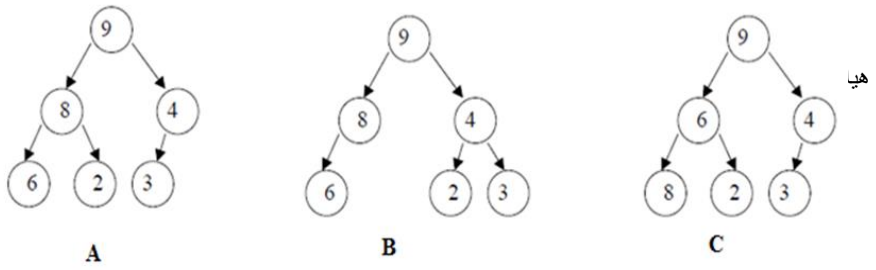
9.1 المقدمة

تطرقنا في الفصل السابق الى الاشجار الثنائية بشكل عام, والاشجار الثنائية من الممكن ان تكون بياناتها مرتبة او لا تكون مرتبة, عندما تكون بيانات الاشجار الثنائية مرتبة فان ذلك يدعى التكديس (Heap). وبالرغم من ان التكديس هو ليس مرتب بشكل كامل, الا انه يتوافق مع مباديء الترتيب. والتكديس عادة هو شجرة كاملة مرتبة البيانات.

9.2 تعريف التكديس Heap

تكديس هياكل بيانات هي شجرة ثنائية مع الخواص التالية:

1. ان تكون شجرة ثنائية كاملة (يعني كل مستوى يكون مملوء بالعقد ماعدا المستوى الاسفل احتمال ان لا يكون مملوء وبهذه الحالة يتم ملأ هذا المستوى من اليسار الى اليمين).
2. ان تكون القيم مرتبة ترتيب تصاعدي او تنازلي (اي ان تكون القيم التي تخزن في العقد مرتبة بحيث تكون قيمة عقدة الاب اكبر او تساوي قيمة الابن وهكذا لجميع العقد (هذا في الترتيب التنازلي, الترتيب التصاعدي بالعكس)).



هي

شكل 9.1: اشجار ثنائية مختلفة (A) فقط هي تكديس.

في الشكل 9.1 فقط A هو تكديس. بينما B هو ليس تكديس بسبب كونه غير كامل, وكذلك فان C هو ليس تكديس بالرغم من انه كامل ولكن لايحقق الشرط الثاني حيث ان القيم غير مرتبة.

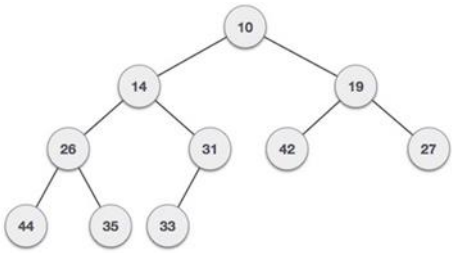
التكديس هي حالة خاصة من الاشجار الثنائية الموزونة, حيث ان عقدة الجذر تقارن مع ابنائها وتنظم او ترتيب وفقا لذلك. هناك نوعان من التكديس.

1. التكديس الاصغر Min-Heap

وهو التكديس الذي تكون فيه قيمة الجذر اصغر من او تساوي قيم ابناءه.

فلو فرضنا لدينا المدخلات التالية, فسيكون تمثيله كما في الشكل 9.2.

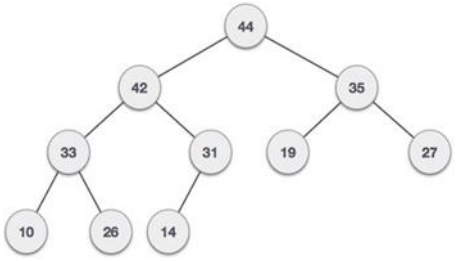
(35, 33, 42, 10, 14, 19, 27, 44, 26, 31)



شكل 9.2: تمثيل القيم على شكل تكديس اصغر.

2. التكديس الاكبر Max-Heap

حيث ان قيمة عقدة الجذر اكبر من او تساوي قيم عقد الابناء. اذا استخدمنا ذات المدخلات في مثال التكديس الاصغر فان الناتج هو الشكل 9.3.



شكل 9.3: تمثيل البيانات على شكل تكديس اكبر

9.3 خوارزمية انشاء التكديس الاكبر

بداية لابد ان نشير الى اننا سنستخدم نفس المثال الذي استخدم اعلاه لتوضيح خلق التكديس الاكبر. كذلك لابد من الاشارة الى ان خوارزمية التكديس الاصغر هي تتبع ذات الخطوات في التكديس الاكبر باستثناء ان المقارنة ستكون مع القيم الاصغر وليس القيم الاكبر.

سنشتق خوارزمية التكديس الاكبر بحشر عنصر واحد في كل مرة. في اي وقت, فان التكديس يجب ان يحافظ على خصائصه.

❖ **خوارزمية التكديس**

1. خلق عقدة جديدة, توضع في نهاية التكديس (تكون متصلة مع اخر عقدة في الشجرة).
2. اسناد قيمة جديدة لهذه العقدة.
3. تقارن قيمة هذه العقدة مع قيمة الاب لها.
4. اذا كانت قيمة العقدة الجديدة اكبر من قيمة العقدة الاب, تتم عملية الابدال بينهما.
5. تعاد الخطوات (3, 4) لحين ان يتم ضبط مواصفات التكديس (اي تكون قيمة الاب اكبر او تساوي من قيمة الابناء).

❖ دعنا نوضح هذه الخوارزمية من خلال مثال وباستخدام القيم

(35, 33, 42, 10, 14, 19, 27, 44, 26, 31)

✓ بداية ستكون القيمة الاولى (35) هي العقدة الاولى وتمثل عقدة الجذر.

✓ القيمة الثانية هي 33 وستكون في نهاية التكديس, تقارن قيمتها بقيمة الاب التي هي الان الجذر (33≤35) نلاحظ انها اصغر اذن ستكون ابن لعقدة الجذر.

شكل 9.4: ربط عقدة جديدة لشجرة التكديس.

✓ القيمة الاخرى هي (42) وهي ستكون الابن الثاني (الشجرة موزونة) وتربط مع الجذر, تقارن قيمتها مع الجذر الذي هو اب لها (35 ≤ 42), هنا القيمة 42 اكبر, لذلك يجب ان تتم عملية التبديل بين الاب والابن كما في الشكل9.5.

شكل 9.5: اضافة عقدة وتبديل.

✓ القيمة الاخرى في المجموعة هي 10 وهي ستكون في اسفل التكديس (الشجرة) وتكون العقدة (33) اب لها, تقارن القيمتين (33 ≤ 10) ونلاحظ ان 33 اكبر منها. لذا ستكون العقدة 33 اب للعقدة 10.

شكل 9.6: اضافة عقدة جديدة.

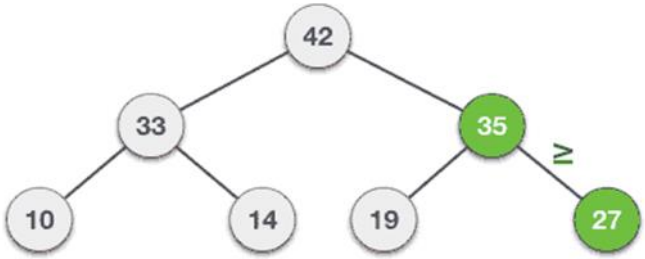
- ✓ القيمة (14) وسيكون موقعها هو الابن الثاني للعقدة (33). تقارن مع قيمة الاب لها, ($33 \leq 14$) لذا لا يوجد تبديل, وتعتبر الابن الثاني للعقدة (33).

شكل 9.7: اضافة عقدة جديدة.

- ✓ نستمر بقراءة قيم المجموعة والقيمة (19) وستكون ابن للعقدة (35), تقارن القيمتان ($35 \leq 19$) ونظرا الى ان قيمة العقدة (35) اكبر من القيمة (19) لذا لا يوجد تبديل وتكون العقدة (19) ابن للعقدة (35).

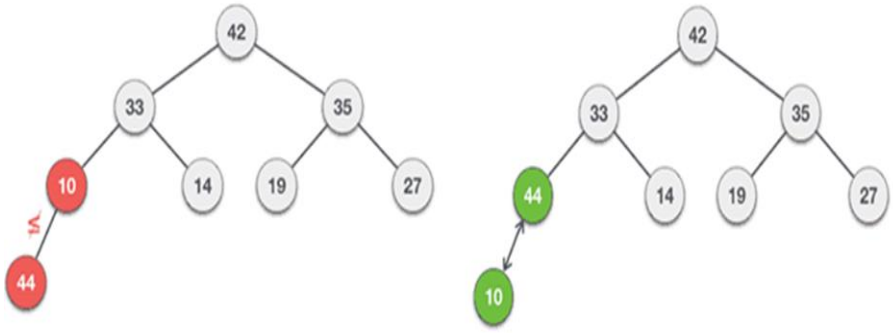
شكل 9.8: اضافة عقدة جديدة.

- ✓ القيمة التالية في المجموعة (27) وستكون الابن الثاني للعقدة (35) (دائما اضافة العقد يستمر من اليسار الى اليمين لانها شجرة كاملة), تقارن القيمتان ($35 \leq 27$), نعم نتيجة المقارنة هو ان (35) اكبر من (27) لذا لا يوجد تبديل وتربط العقدة (27) بالعقدة (35) لتكون الابن الثاني لها.



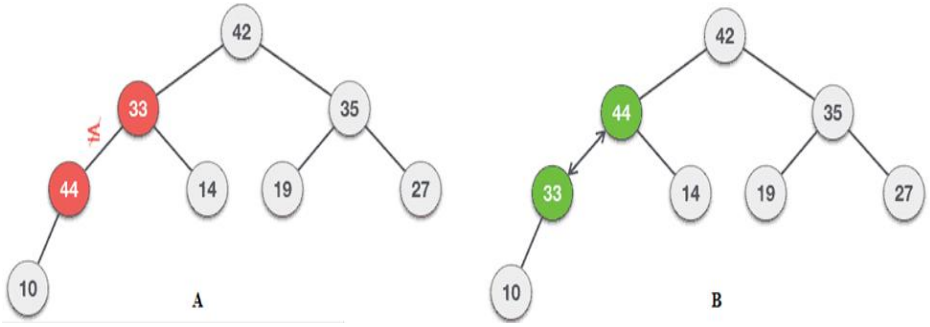
شكل 9.9: اضافة عقدة جديدة.

- ✓ القيمة التالية (44) وهي ستكون ابن العقدة (10), تقارن القيمة المضافة مع قيمة الاب ($10 \leq 44$), في هذه الحالة فان 10 اصغر من 44 لذا لابد من التبديل.



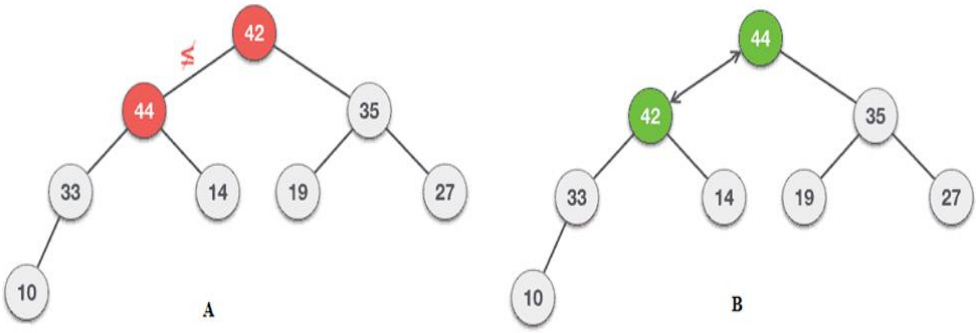
شكل 9.10: اضافة عقدة واجراء التبديل بين الابن والاب.

- ✓ بعد اجراء التبديل يلاحظ ان القيمة (44) اصبحت ابنا للعقدة (33) وعليه تقارن القيم (الابن (44) مع الاب الجديد) للمحافظة على صفات التكديس ($33 \leq 44$) ايضا هنا ان القيمة (44) هي اكبر من القيمة (33), لذا تبدل القيمتين.



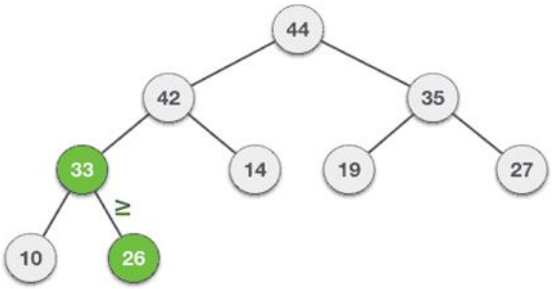
شكل 9.11: استمرار عملية التبديل.

- ✓ العقدة (44) اصبحت ابنا للعقدة (42) لذا تقارن قيمهم ($42 \leq 44$) ونلاحظ ان القيمة (44) اكبر من القيمة (42) عليه تجرى عملية التبديل.



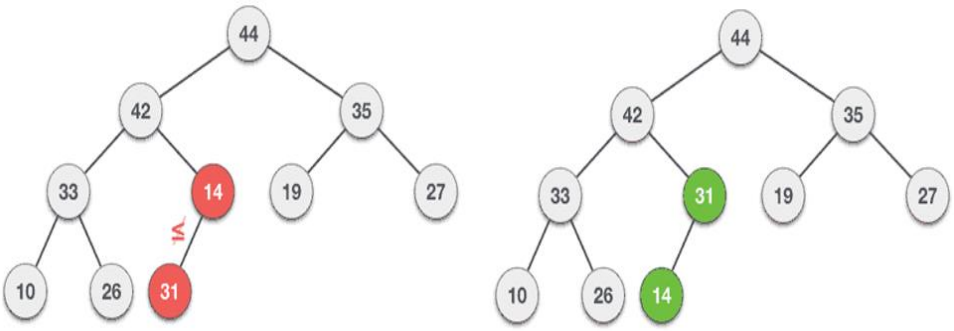
شكل 9.12: التبديل الثالث لكي نحافظ على مواصفات التكديس.

- ✓ بعد اتمام عمليات الابدال يتم قراءة القيمة الاخرى من مجموعة قيم المدخلات وهي القيمة (26), ستكون هذه القيمة الابن الثاني للعقدة (33), تقارن هذه القيم ($33 \leq 26$) وهنا القيمة (26) اصغر من القيمة (33) لذا لا يوجد تبديل.



شكل 9.13: اضافة عقدة جديدة.

- ✓ القيمة الاخيرة وهي القيمة (31) والتي ستكون ابنا للعقدة (14) وتتم المقارنة ($14 \leq 31$) ونظرا الى ان القيمة (31) اكبر من (14) لذا تتم عملية التبديل بينهما.



شكل 9.14: اضافة عقدة جديدة واجراء التبديل مع الاب.

✓ بعد التبديل ستكون العقدة (31) ابنا للعقدة (42) واذا ما قارنا القيمتين سنجد ان العقدة (31) اصغر من العقدة (42) لذا لايوجد تبديل. بهذا نكون قد خلقنا شجرة التكديس.

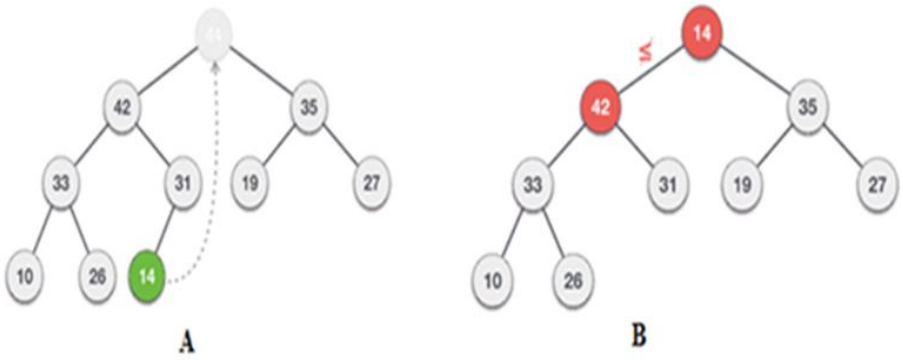
9.4 الحذف من شجرة التكديس

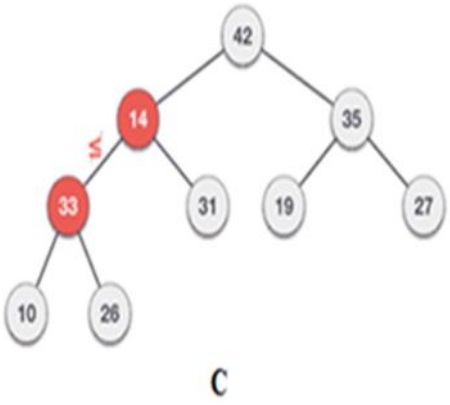
ان عملية الحذف سواء كان التكديس الاكبر او التكديس الاصغر يتم بحذف الجذر. وذلك لازالة القيمة الاكبر او القيمة الاصغر.

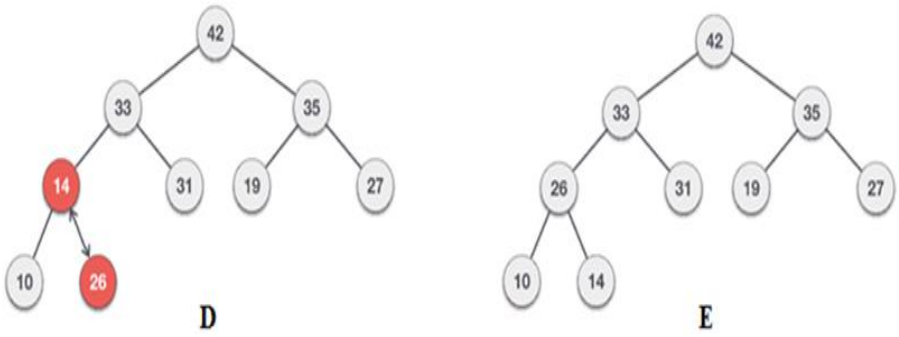
❖ **خوارزمية الحذف**

1. حذف عقدة الجذر.
2. تحريك العنصر الاخير في المستوى الادنى الى الجذر (بمعنى ستكون القيمة الاقل هي الجذر في حالة التكديس الاكبر وبالعكس في حالة التكديس الاصغر).
3. مقارنة قيمة عقدة الجذر الجديدة مع قيمة ابنها (دائما تبدا المقارنة اولا مع الابن الايسر ثم الابن الايمن).
4. اذا كانت القيم غير متوافقة وفقا لنوع التكديس فتتم عملية التبديل (في التكديس الاكبر اذا كان الابن اكبر من الاب تتم التبديل وبالعكس للاصغر).
5. اعادة الخطوات (3, 4) لحين ان توليد شجرة تكديس محافظة على خواصها.

❖ **مثـال** علـى تطبيـق خوارزميـة الحـذف علـى التكـديس الـذي تـم انشـاؤه سـابقا. موضح في الشكل 9.15.







شكل 9.15: توضيح عملية الحذف.

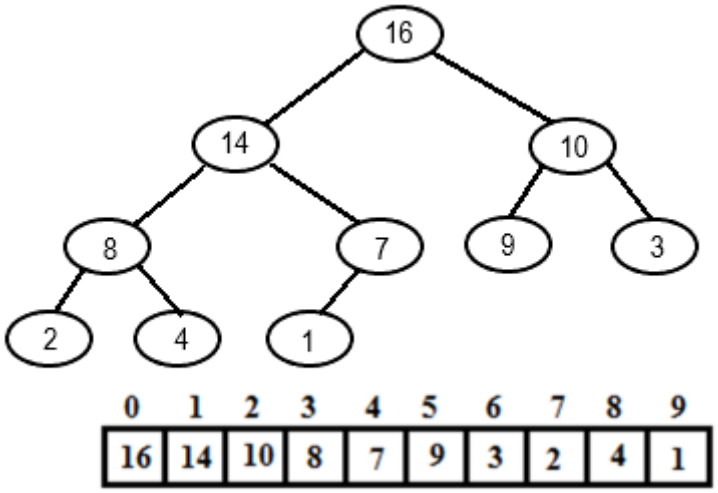
9.5 تمثيل التكديس

من الممكن تنفيذ التكديس باستخدام القوائم الموصولة كما فعلنا مع الاشجار الثنائية, ولكن من المفضل والاسهل تنفيذ التكديس باستخدام المصفوفات.

لتنفيذ ذلك يتم ترقيم شجرة التكديس من الاعلى للاسفل ابتداءا من الجذر الذي سيكون رقمه صفر (وذلك لان المصفوفات في C++ تبدأ بصفر), ترقم العقد في كل مستوى من اليسار الى اليمين. بعد الترقيم تخزن كل عقدة في الموقع المقابل للرقم الذي وضع على العقدة, حيث تخزن العقدة i في الموقع i من المصفوفة. ان حجم المصفوفة سيكون بقدر عدد العقد في التكديس.

❖ **مثال** لنلاحظ التكديس في الشكل 9.16 وكيفية تمثيله في المصفوفة.

هياكل البيا



شكل 9.16: تمثيل التكديس على شكل مصفوفة.

9.5.1 برنامج لتمثيل التكديس مع العمليات التي تجرى عليه.

```
#include <stdio>

   using namespace std;

   int array[100], n ;

   main()   {

   int choice, num ;
   n = 0 ;     /*   تمثل عدد العقد في التكديس*/
   while (1)
   {

   cout<<  "1.Insert the element \n" ;
   cout<<  "2.Delete the element \n" ;
   cout<<  "3.Display all elements \n" ;
   cout<<  "4.Quit \n" ;
   cout<<  "Enter your choice : " ;

   cin>> choice ;
```

```
switch (choice)
{
case 1:
  cout<<  "Enter the element to be inserted to the list : " ;
  cin>> num ;
  insert (num, n) ;
  n = n + 1;
  break;
case 2:
  cout<<  "Enter the elements to be deleted from the list: " ;
  cin>> num ;
  delete (num);
  break;
case 3:
  display();
  break;
case 4:
  exit(0);
default:
  cout<<  "Invalid choice \n" ;
}   /* switch   نهاية*/
}   /* while    نهاية*/
}   /* main()   نهاية*/

display()
{
 int i;
 if (n == 0)
  {
     cout<<   "Heap is empty \n" ;
     return;
   }
 for (i = 0; i < n; i++)
 cout<<   array[i] << endl ;
}  /*   display()     نهاية*/

insert (int num, int location)
{
  int parentnode;
  while (location > 0)
```

```
  {
    parentnode =(location - 1)/2;
    if (num <= array[parentnode])
   {
     array[location] = num;
     return;
    }
  array [location] = array[parentnode];
  location = parentnode;
}   /* while     نهاية*/

array [0] = num; /* اسناد رقم الى عقدة الجذر */


}     /* insert()      نهاية*/

delete(int num)
{
    int left, right, i, temp, parentnode;

    for (i = 0; i < num; i++) {
      if (num = = array[i])
        break;
       }
      if (num != array[i])
       {
         cout<<  " not found in heap list\n" <<  num ;
         return;
        }
  array [i] = array[n - 1];
  n = n - 1;
parentnode = (i - 1) / 2; /* i  ايجاد اب العقدة*/
  if (array [i] > array [parentnode])
  {
     insert (array[i], i);
     return;
  }

left = 2 * i + 1;      /* i  الابن الايسر للعقدة*/
```

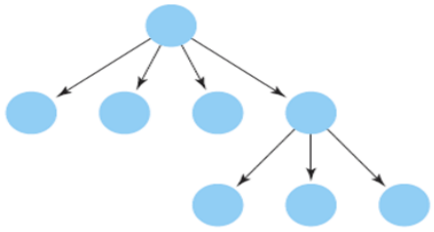
```
right = 2 * i + 2;   /* i الابن الايمن للعقدة*/
while (right < n)
    {
    if (array[i] >= array[left] && array[i] >= array[right])
    return;
    if (array[right] <= array[left])
    {
    temp = array[i];
    array [i] = array[left];
    array [left] = temp;
    i = left;
    }
    else
    {
    temp = array[i];
    array [i] = array[right];
    array [right] = temp;
    i = right;
    }
    left = 2 * i + 1;
    right = 2 * i + 2;
}/* while نهاية*/

if (left == n - 1 && array[i])    {
    temp = array[i];
    array[i] = array[left];
    array[left] = temp;
}
}
```

الفصل العاشر
المخططات GRAPHS

10.1 المقدمة

في عالم الرياضيات وعلوم الحاسوب, نظرية المخططات هي دراسة المخططات التي تتعامل مع العلاقة بين الروابط والرؤوس (العقد).

الاشجار الثنائية توفر طرق مفيدة جدا لتمثيل العلاقات عندما يكون التنظيم تنظيم هرمي, حيث ان العقدة يشار اليها بعقدة واحدة هي الاب وكل عقدة تشير الى اثنين من العقد كحد اعلى (في حال الاشجار الثنائية) كما سبق وشرحنا في فصل الاشجار. اذا تم ازالة الشرط الخاص بان يكون لكل عقدة اثنان من الابناء, فسيكون لنا شجرة عامة كما في الشكل 10.1.



شكل 10.1: شجرة عامة.

ايضا اذا تم ازالة الشرط الخاص بان يكون لكل عقدة اب واحد فسيكون لدينا شكل يسمى مخطط graph.

10.2 ماهو المخطط Graph

هو تمثيل رسومي لمجموعة من الكيانات, حيث ان بعض الازواج من الكيانات ترتبط بروابط. الكيانات المترابطة من الممكن تمثيلها بنقاط (ولكي تكون اكثر وضوحا تمثل بدوائر), وخطوط تربط هذه النقاط.

10.3 تطبيقات نظرية المخططات

نظرية المخططات لها تطبيقات بحقول مختلفة من الهندسة:

- الهندسة الكهربائية.. مفاهيم نظرية المخططات تستخدم بشكل واسع في تصميم الدوائر الكهربائية.
- علوم الحاسبات.. تستخدم نظرية المخططات لدراسة الخوارزميات. مثال
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Dijkstra's Algorithm

- شبكات الحاسوب.. العلاقة بين الحواسيب المترابطة مع بعض في الشبكة تتبع مبدأ نظرية المخططات.
- العلوم.. هيكل الجزيئات والهيكل الكيميائي للمواد, هيكل DNA كلها تمثل بالمخططات.
- اللغات.. شجرة التعبير للغة وقواعد اللغة تستخدم المخططات.
- امور عامة.. الطرق Routes بين المدن من الممكن تمثيلها بالمخططات. بالرغم من الترتيب الهرمي فان المعلومات مثل شجرة العائلة من الممكن ان تستخدم كنوع خاص من المخططات تدعى الاشجار.

10.4 مصطلحات المخططات

النقطة Point: هي موقع خاص في فضاء يتكون من بعدين او ثلاث ابعاد. لكي نفهمها اكثر فان النقطة تمثل باحد حروف الهجاء. من الممكن تمثيلها كما في الشكل 10.2.

شكل 10.2: تمثيل نقطة في المخطط

الخط Line: هو الرابط بين نقطتين. وهو يمثل كما في الشكل . 10.3

شكل 10.3: خط يربط النقطتين (a, b).

الراس او العقدة Vertex: هي النقطة التي تكون ملتقى لعدد من الخطوط. وهي تسمى عقدة. وكما في النقاط فان هذه العقد تمثل بالحروف الهجائية. كما في الشكل 10.2.

الحواف Edge: الحواف هو مصطلح رياضي للخط الرابط بين راسين (عقدتين). العديد من الحواف التي من الممكن ان نرسمها او نخلقها من عقدة مفردة. بدون العقد فان الحواف لا يمكن خلقها. يجب ان يكون هناك عقدة بداية وعقدة نهاية. هذه الحواف سنسميها روابط لكي تكون اكثر وضوحا. كما في الشكل 10.3.

مما تقدم من الممكن تعريف المخطط على انه عبارة عن مجموعة من العقد تدعى رؤوس ومجموعة من الخطوط التي تربط كل اثنين من الرؤوس تدعى روابط.

المخطط يحتوي على الاقل رابط واحد يربط مجموعة من العقد تتكون من عقدتين دون ان تكون هناك عقدة ترتبط بنفسها.

يمثل المخطط رياضيا $G = (V, E)$

حيث ان

(V) تمثل مجموعة العقد.

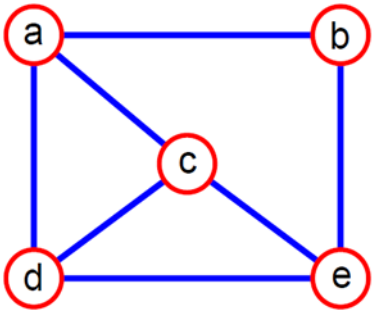
(E) مجموعة من الروابط التي تربط العقد.

e = (u, v) هو الرابط بين اثنان من العقد.

في الشكل 10.4, فان

V= {a,b,c,d,e}

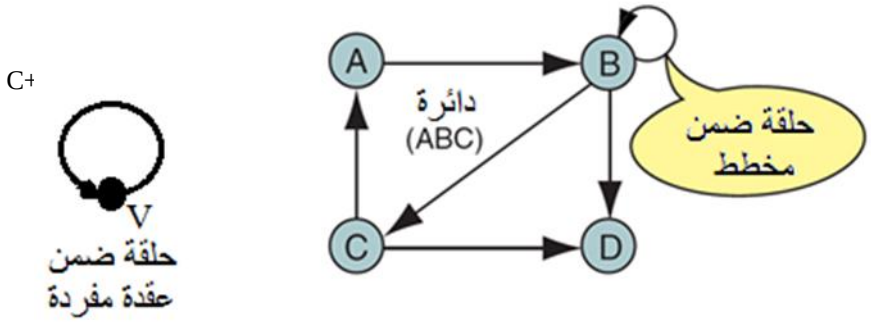
E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e),(d,e)}



شكل 10.4: مخطط يتكون من خمس عقد.

المخططات من الممكن ان تكون متجهة او غير متجهة, في الاشكال المتجهة فان كل خط والذي يسمى قوس Arc يكون له اتجاه يمثل كيفية المرور او الانتقال, اما غير المتجهة فان لها خطوط تسمى حواف edge ويمكن ان تتنقل بكلا الاتجاهين.

الحلقة Loop: في المخطط فان الرابط الذي يرسم من عقدة الى نفس العقدة, يدعى حلقة. الشكل 10.5 يوضح ذلك.



شكل 10.5: يوضح الحلقة في المخطط.

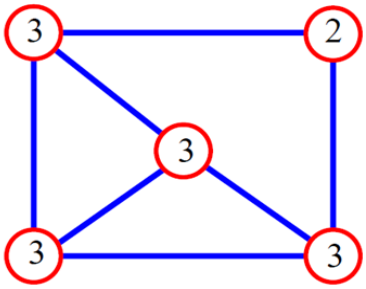
في الشكل 10.5 نلاحظ ان (V) هي عقدة ولها الرابط e = (V, V)

- **العقد المتجاورة Adjacent Vertices**: وهي اثنان من العقد بينها رابط يربطهما.
- **درجة العقدة Degree (of a vertex)**: وهي تمثل عدد الرؤوس المتجاورة في المخطط البسيط. هي عدد الروابط التي تتعلق بتلك العقدة. لاحظ الشكل 10.6 والذي يحدد درجة كل عقدة (الارقام التي داخل العقد). من الممكن ان نقول هي عدد الروابط التي ترتبط بتلك العقدة. ان درجة العقدة لاي عقدة (v) هي

$$\deg(v) \leq n - 1 \quad \forall\, v \in G$$

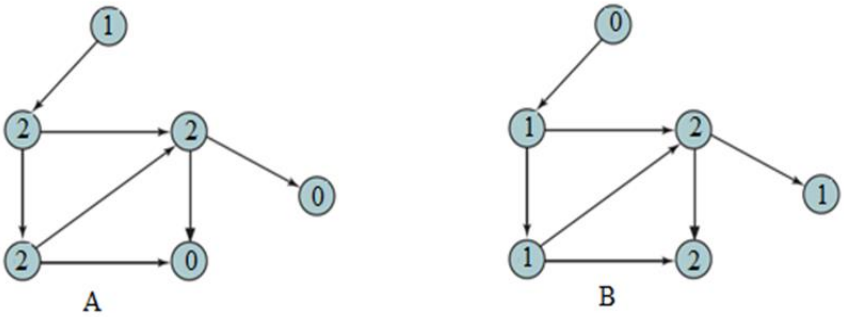
حيث ان (n) تمثل عدد العقد في المخطط. وهذا يعني ان اي عقدة بامكانها ان ترتبط مع كل عقد المخطط باستثناء نفسها. لذلك فان درجة العقدة تساوي عدد العقد في المخطط مطروح منها واحد الذي يمثل العقدة التي نحسب لها الدرجة لانه لا يمكن ان تعمل ارتباط بنفسها (اذا كان هناك حلقة ضمن المخطط فهذا لا يعتبر مخطط بسيط).

درجة العقدة من الممكن ان تحسب لحالتي المخطط المتجه والمخطط غير المتجه. في الشكل 10.6 تم تحديد الدرجة لكل عقدة في مخطط غير متجه.



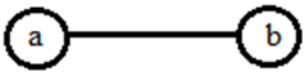
شكل 10.6: مخطط غير متجه تم تحديد درجة كل عقدة عليه.

في المخططات المتجه فان كل عقدة لها نوعين من الدرجات. درجة الدخول (indegree) وهي تمثل عدد الروابط التي تدخل الى العقدة ويرمز لها ($\deg^+(V)$), والاخرى هي درجة الخروج (outdegree) وهي تمثل عدد الروابط التي تخرج من العقدة ويرمز لها ($\deg^-(V)$).



شكل 10.7: مخطط متجه تم تحديد : A. درجة الخروج لكل عقدة. B. درجة الدخول لكل عقدة.

1. **عقدة معلقة Pendent Vertex**: العقدة التي مقدار درجتها تساوي 1 تسمى عقدة معلقة, في الشكل 10.8 فان كل من (a, b) لها درجة 1, لذلك كلاهما يعتبران معلق.

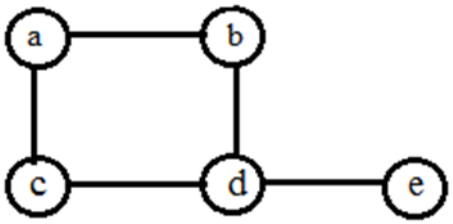


شكل 10.8: عقد pendent

2. **العقدة المعزولة Isolated Vertex**: هي العقدة التي لها درجة تساوي صفر. لاحظ الشكل 10.9 ان كل واحدة من هذه العقد لها درجة صفر, لذلك تدعى معزولة اي ليس لها ارتباط مع اي عقدة اخرى.

شكل 10.9: عقد معزولة

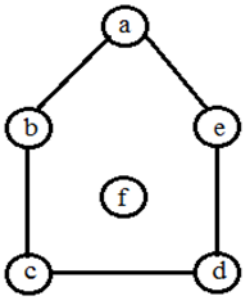
3. **درجة التوالي للمخطط Degree Sequence of A Graph**: في المخطط اذا كانت درجة جميع العقد مرتبة ترتيب تنازلي او ترتيب تصاعدي, عليه فان المتوالية الناتجة تدعى درجة التوالي للمخطط. لنلاحظ الشكل 10.10.



شكل 10.10: مخطط يوضح درجة التوالي.

في الشكل 10.10 فان العقد { d, a, b, c, e } لها درجة توالي هي:
{3, 2, 2, 2, 1}.

مثال اخر: الشكل 10.11 يحتوي على العقد {a, b, c, d, e, f} والتي لها درجة توالي هي {2, 2, 2, 2, 2, 0}



شكل 10.11: مخطط له درجة توالي.

4. **التجاور Adjacency**: التجاور نوعان:

- ✓ العقد المتجاورة: في المخططات يقال عن اثنان من العقد متجاورة, اذا كان هناك رابط بين هاتين العقدتين. ان تجاور العقد يجب ان يكون بربط عقدتين برابط مفرد.
- ✓ الرابط المتجاور: في المخططات, يقال عن اثنين من الروابط متجاورة, اذا كانت هناك عقدة مشتركة بين الرابطين. ان تجاور الرابطين يجب ان يتم بوجود عقدة مفردة مشتركة.

في الشكل 10.12 فان

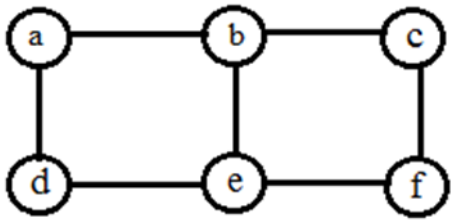
(a, b) عقدتان متجاورتان وذلك لوجود رابط مفرد مشترك بينهما وهو (ab).

العقدتين (a, d) هما ايضا عقدتين متجاورتين وذلك لوجود رابط مفرد مشترك بينهم وهو (ad).

الرابطين (ab, be) هما رابطين متجاورين وذلك لوجود عقدة مشتركة بينهم وهي العقدة (b).

الرابطين (be, de) هما رابطين متجاورين وذلك لوجود عقدة مشتركة بينهم وهي العقدة (e).

C++ ، باستخدام

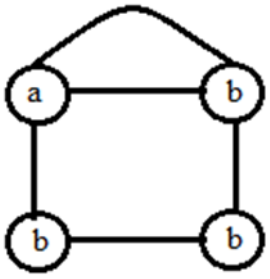


شكل 10.12: مخطط يوضح التجاور بنوعيه.

- **الروابط المتوازية Parallel Edges**: في المخططات اذا كان زوج من العقد ترتبط باكثر من رابط واحد, عليه فان هذه الروابط تسمى روابط متوازية. لاحظ الشكل 10.13 فان العقدتين (a, b) هما عقدتين ترتبطان برابطين هما (ab, ab), لذلك يسميان رابطين متوازيين.

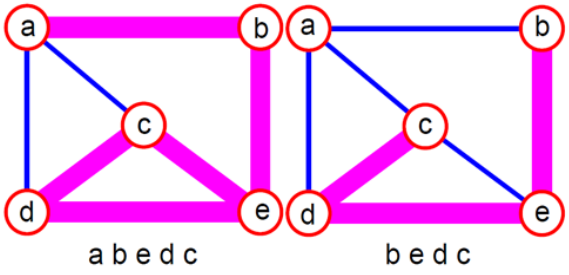
شكل 10.13: روابط متوازية.

- **مخطط متعدد Multi Graph**: المخطط الذي يحتوي على روابط متوازية اضافة الى روابط غير متوازية يسمى مخطط متعدد. لاحظ الشكل 10.14.



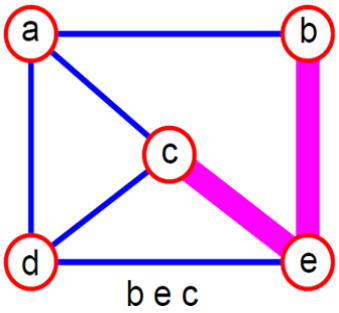
شكل 10.14: مخطط متعدد.

5. **المسار Path**: هي متوالية من العقد بحيث ان العقد المتعاقبة (v_i , v_{i+1}) تكون متجاورة. لاحظ الشكل 10.15.



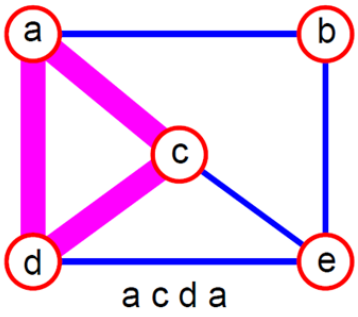
شكل 10.15: مخطط موضح عليه المسارات.

6. **المسار البسيط Simple Path**: هو المسار الذي ليس فيه عقد مكررة, كما في الشكل 10.16.



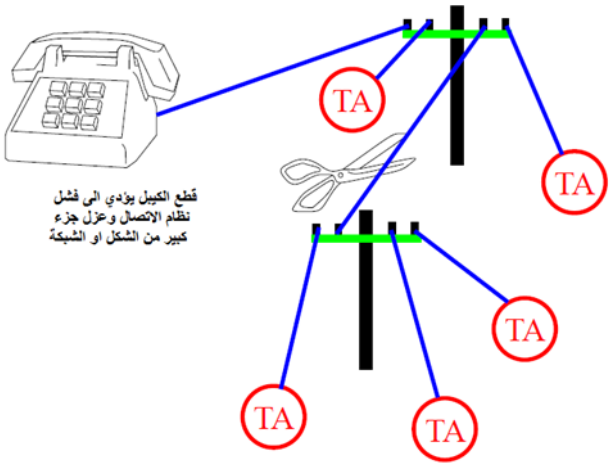
شكل 10.16: مسار بسيط

7. **دائري cycle**: مسار بسيط, ماعدا ان العقدة الاخيرة هي العقدة الاولى, لاحظ الشكل 10.17.



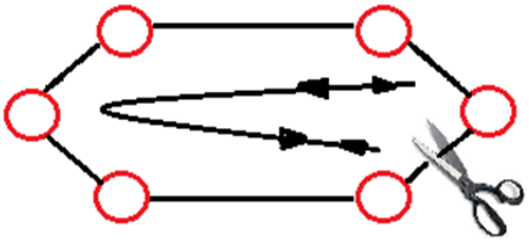
شكل 10.17:مخطط دائري.

ملاحظة: الفشل في اي رابط يؤدي الى عدم ترابط النظام (المخطط بشكل عام اقل سماح للاخطاء او الفشل) لاحظ الشكل 10.18.



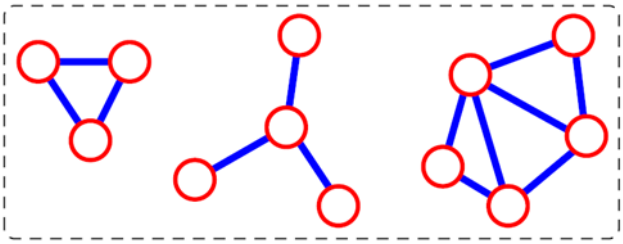
شكل 10.18: مخطط يوضح كيف يؤثر الفشل (القطع) بعزل جزء كبير من الشبكة.

في المقابل فان الشكل الدائري هو اكثر سماح للفشل او الاخطاء, هو يحتاج عدد من الروابط بقدر عدد الرؤوس او العقد, انظر الشكل 10.19.



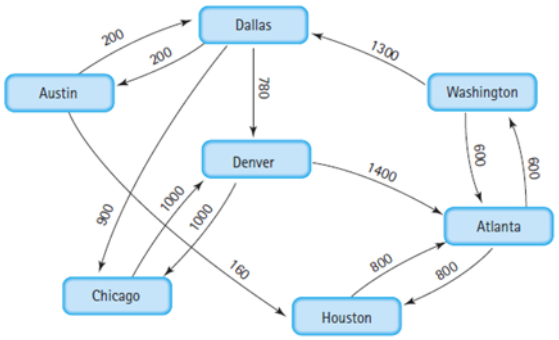
شكل 10.19: مخطط دائري يوضح كيف من الممكن تجاوز الفشل او القطع.

8. **مخطط فرعي Subgraph**: مجموعة جزئية من العقد والروابط تشكل مخطط فرعي.
9. **المكـون المتـرابط Connected Component**: هـي اكبـر عـدد مـن المخططات الفرعية المترابطة. في الشكل 10.20 ثلاث مكونات مترابطة.



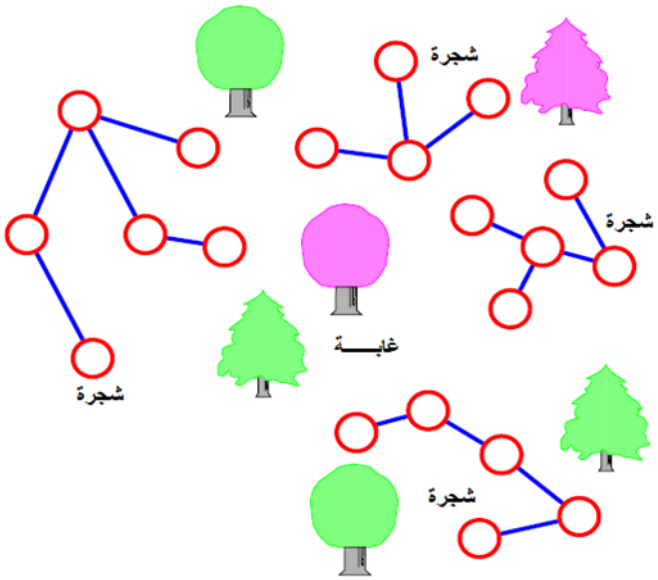
شكل 10.20: مكونات مترابطة.

1. **مخطط موزون Weighted Graph**: هو المخطط الذي يكون فيه كل رابط يحتوي على قيمة محددة.



شكل 10.21: مخطط موزون

2. **شجرة (حرة) Tree (free)**: اشكال مترابطة بدون دوائر.
3. **الغابات Forest**: تجمع للاشجار.



شكل 10.22: مخطط يوضح مفهوم الغابة والاشجار (الحرة).

- **المخطط الكامل Complete Graph**: المخطط الذي تكون فيه كل ازواج العقد متجاورة.

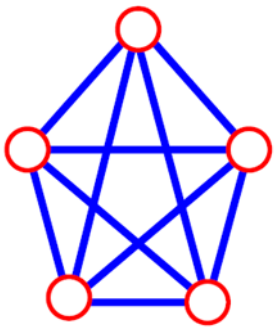
لنفرض ان (n) يمثل عدد العقد. ونفرض ان (m) يمثل عدد الروابط. عليه فان

$$\mathbf{m} = (1/2)\sum_{v \in \mathbf{V}} \deg(\mathbf{v}) = (1/2)\sum_{v \in V} (\mathbf{n} - 1) = \mathbf{n}(\mathbf{n}-1)/2$$

كل (n) من العقد سيكون لها (n-1) من الروابط. لكن نحن نحسب كل رابط مرتين لذلك فان العلاقة تكون ((n (n-1) /2

في الشكل 10.23 فان عدد العقد هي (5) ولذلك فان عدد الروابط ستكون

5 (5 – 1) / 2 = 10



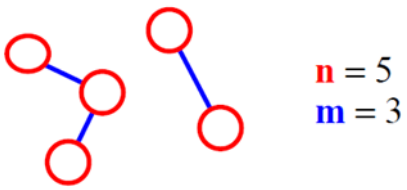
شكل 10.23: مخطط كامل.

عليه, فاذا لم يكن المخطط كامل فان عدد الروابط سيكون اقل من (n (n-1)/2).

بالنسبة للشجرة فان عدد الروابط تساوي

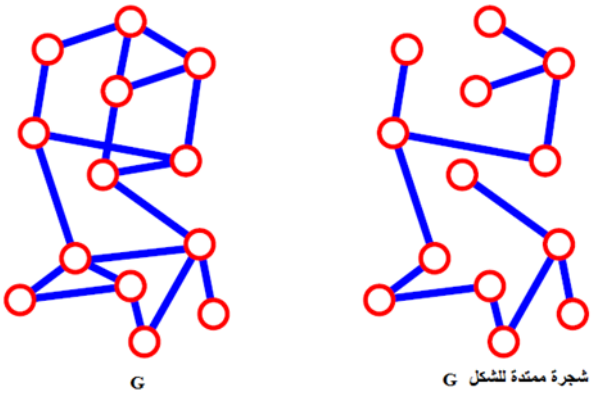
m = n – 1

ملاحظة: اذا كانت (m < n-1), فان المخطط سيكون غير مترابط كما واضح في الشكل 10.24.



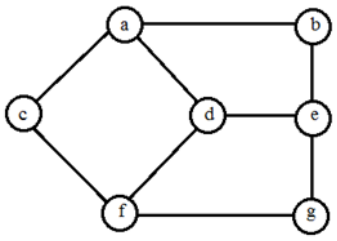
شكل 10.24: مخطط غير مترابط

- **شجرة ممتدة Spanning Tree**: الشجرة الممتدة للمخطط (G) هي مخطط فرعي له المواصفات التالية
  - ✓ هو شجرة.
  - ✓ يحتوي على كل عقد المخطط (G).



شكل 10.25: مقارنة بين المخطط والشجرة الممتدة.

- **المسافة بين عقدتين d(U, V)**: ان المسافة بين العقدة (U) والعقدة (V) هي اقصر مسار بين العقدتين. فاذا كان هناك اكثر من مسار يصل بين العقدتين فان اقصر واحد هو يمثل المسافة.



شكل 10.26: مخطط لحساب المسافة.

هنا في الشكل 10.26 فان المسافة من العقدة (d) الى العقدة (e) او ببساطة (de) هو واحد حيث يوجد رابط واحد بينهم. هناك عدة مسارات من العقدة (d) الى العقدة (e) وهي

- ✓ da, ab, be (3 = المسافة)
- ✓ df, fg, ge (3 = المسافة)
- ✓ de (1 = المسافة) هي اقصر مسافة
- ✓ df, fc, ca, ab, be (5 = المسافة)
- ✓ da, ac, cf, fg, ge (5 = المسافة)

- **انحراف العقدة Eccentricity of A Vertex**: في المخطط المتصل فان انحراف العقدة هو اكبر مسافة بين عقدة واي عقدة اخرى في المخطط, ويرمز لها e(V). بالنسبة للمخططات غير المتصلة فان جميع العقد لها انحراف يساوي ما لانهاية.

في الشكل 10.26 فان انحراف العقدة (a) هو 3.

لحساب الانحراف يتم تحديد المسافات بين العقدة (a) وجميع العقد الاخرى وتكون المسافة الاكبر هي التي تمثل الانحراف. لنحسب المسافة بين العقدة (a) وباقي العقد وهي:

الى العقدة (b) المسافة = 1 (ab)

الى العقدة (c) المسافة = 1 (ac)

المسافة الى العقدة

d = 1 (ad)

e = 2 (ab, be) OR (ad, de)

f = 2 (ac, cf) OR (ad, df)

g = 3 (ac, cf, fg) OR (ad, df, fg)

- **نصف قطر المخطط المتصل Radius of A Connected Graph**

ان الانحراف الاصغر من بين انحرافات كل العقد يعتبر نصف قطر المخطط G. المسافة الصغرى من بين كل المسافات العظمى بين عقدة وجميع العقد الاخرى يعتبر نصف قطر المخطط (G). يرمز لها r(G).

بكلام اخر من بين كل الانحرافات للعقد في المخطط, يتم اختيار الانحراف الاصغر ليمثل نصف القطر. في الشكل 10.26 فان (r(G) = 2) والتي تمثل اصغر انحراف للعقدة (d). لاحظ ان الانحراف للعقد في الشكل 10.26 هي:

e(b) = 3

e(c) = 3

e(d) = 2

e(e) = 3

e(f) = 3

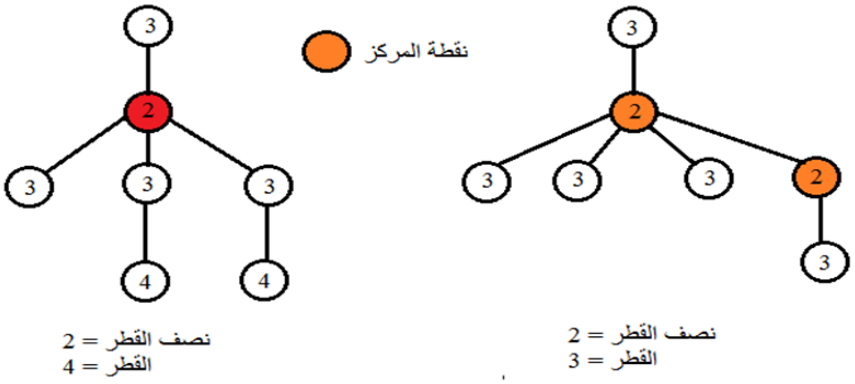
e(g) = 3

- **قطر المخطط Diameter of a Graph**

ان اكبر قيمة للانحراف لكل العقد يعتبر القطر للمخطط (هنا يختلف مفهوم القطر ونصف القطر عما تعلمناه في الرياضيات عند التعامل مع الدائرة). ويرمز لها d(G).

اذن يتم حساب الانحراف لجميع العقد كما فعلنا مع نصف القطر, والانحراف الاكبر من بين هذه القيم يعتبر هو قطر المخطط. القطر للمخطط في الشكل 10.26 هو (d(G) = 3) والتي هي اكبر قيمة انحراف.

- **نقطة التمركز Central Point**: عندما تتساوى درجة الانحراف للمخطط مع نصف قطره فان هذه النقطة تسمى نقطة التمركز. بكلام اخر عند حساب نصف القطر, وعند تطابق هذه القيمة مع انحراف عقدة ما فان هذه العقدة تعتبر المركز. في الشكل 10.26 فان نصف القطر يساوي 2, وهي تطابق درجة الانحراف للعقدة (d).
- **المركز Center**: هي مجموعة نقاط التمركز في المخطط. في مثال المخطط اعلاه فان مركز المخطط هو {d}. لاحظ الشكل 10.27.



شكل 10.27: انحرافات المخطط (الارقام داخل العقد تمثل انحراف العقدة).

- **المحيط Circumference**: عدد الروابط في اطول دائرة في المخطط (G) تسمى المحيط. في الشكل 10.26 فان المحيط يساوي (6), والذي يمثل اطول دائرة في المخطط متمثلة

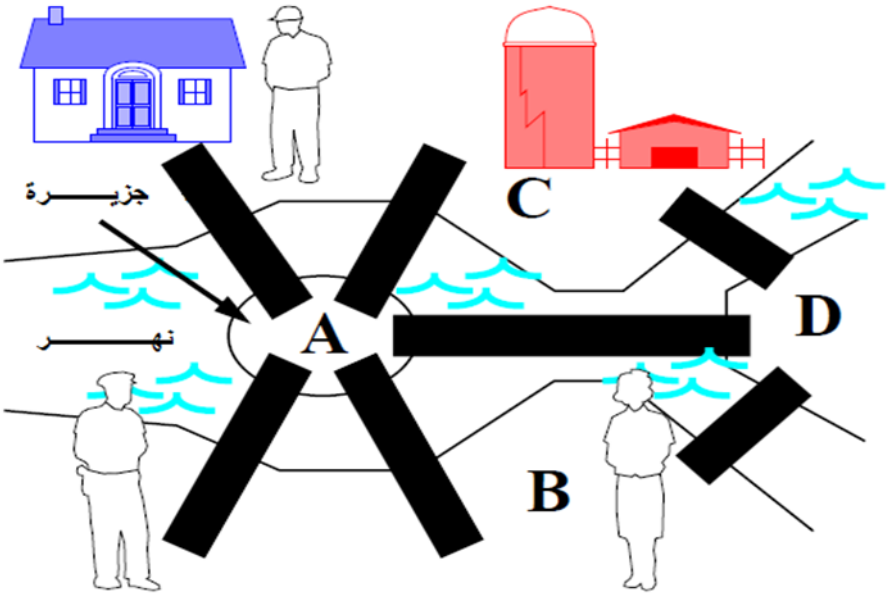
(a-c-f-g-e-b-a) او (a-c-f-d-e-b-a).

- **المقاس Girth**: عدد الروابط في اقصر دائرة في المخطط (G). ويرمز له g(G). في الشكل 10.26 فان المقاس يساوي 4 وهو يمثل

(a-c-f-d-a) or (d-f-g-e-d) or (a-b-e-d-a).

10.5 اولر وجسور كونزبيرج Euler and the Bridges of Koenigsberg

ربما الكثير منا سمع بلغز الجسور التي موضحة بالشكل 10.28, المطلوب باللغز هو المرور على كل جسر في الشكل مرة واحدة فقط والعودة الى نقطة الانطلاق (الجسور هي المستطيلات السوداء في الشكل), فهل حاولت ان تحل هذه المسالة؟, وان كنت لم تحاول فهل من الممكن ان تحل المسالة قبل ان تنتقل الى موضوع اخر في هذا الفصل. اتمنى لكم وقتا ممتعا.



شكل 10.28: لغز جسور كونزبيرج.

10.5.1 نظرية اولر Euler's Theorem

في العام 1736 اعلن اولر عن نظريته الشهيرة والتي تفيد:

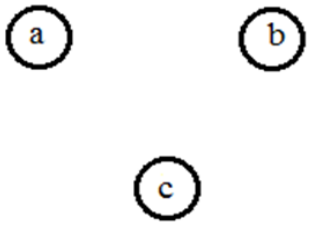
"أي مخطط من الممكن ان تمر على كل رابط فيه مرة واحدة فقط وتعود الى نقطة البداية او العقدة التي بدأت منها, اذا واذا فقط كانت كل العقد لها درجة زوجية (درجتها رقم زوجي)".

الان نعود الى اللغز, وعليك الاستعانه بنظرية اولر للاجابة عن اللغز.

10.6 انواع المخططات

هناك انواع متعددة من المخططات تعتمد على عدد العقد, عدد الروابط, الترابط, وكامل هيكل المخطط. سنتطرق لأنواع قليلة مهمة من المخططات في هذا الفصل.

1. **المخطط الفارغ Null Graph**: وهو المخطط الذي لا يحتوي على رابط. كمثال في الشكل 10.29 هناك ثلاث عقد لكن لا يوجد بينها رابط.

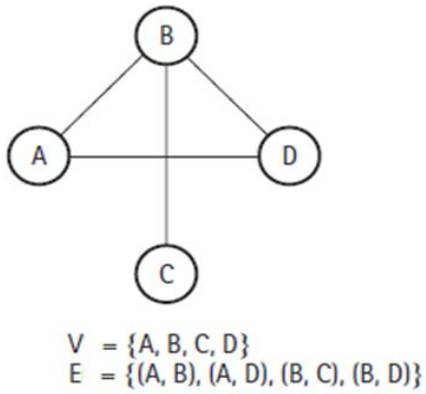


شكل 10.29: مخطط فارغ.

2. **المخطط الوضيع Trivial Graph**: المخطط الذي له عقدة واحدة يسمى المخطط الوضيع. في الشكل 10.30 فان هناك عقدة واحدة وليس لها أي رابط.

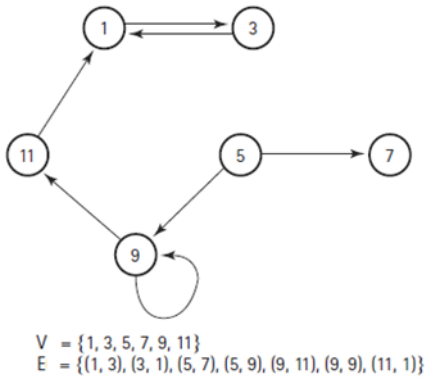
شكل 10. 30: مخطط وضيع.

3. **مخطط غير متجه Undirected Graph**: هو المخطط الذي تكون روابطه غير متجه.



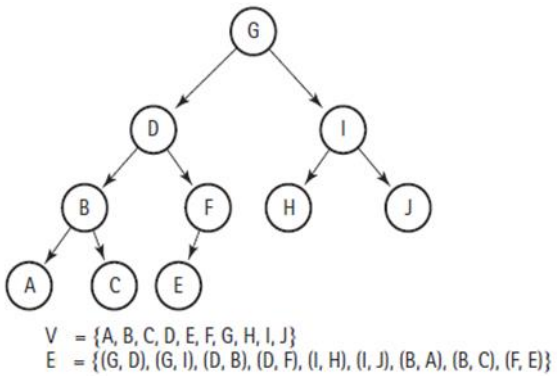
شكل 10.31: مخطط غير متجه

4. **مخطط متجه Directed Graph**: هو المخطط الذ يكون فيه كل رابط محدد اتجاهه من عقدة الى اخرى (او من الممكن نفس العقدة).



شكل 10.32: مخطط متجه

ام C++



شكل 10.33: مخطط متجه

5. **المخطط البسيط Simple Graph**: هو المخطط الذي لا يحتوي على حلقات او روابط متوازية.

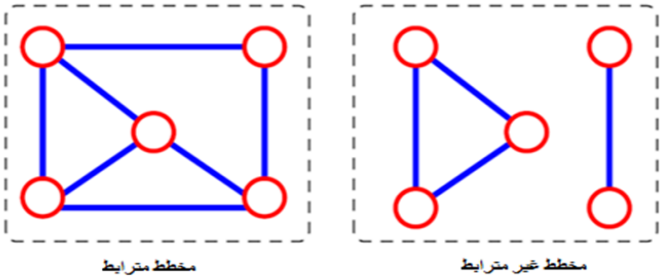
ان اكبر عدد للروابط في مخطط مفرد, عدد عقده يساوي (n) هو

$$= n (n – 1) / 2$$

كما سبق ووضحنا في هذا الفصل. ان اكبر عدد من المخططات البسيطة الممكنة عندما يكون لدينا (n) من العقد هو

$$= 2^{n(n-1)/2}$$

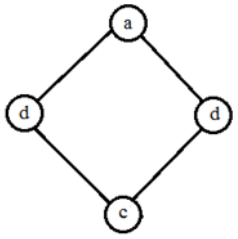
6. **المخطط المترابط Connected Graph**: اي عقدتين في المخطط تكون مترابطة بواسطة مسار, الشكل 10.34.



شكل 10.34: توضيح الفارق بين المخطط المترابط وغير المترابط.

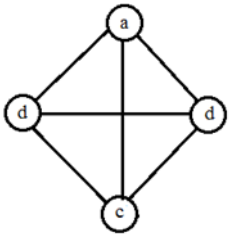
7. **المخطط غير المتصل Disconnected Graph**: يكون المخطط غير مترابط اذا احتوى على الاقل عقدتين غير مترابطتين. لاحظ الشكل 10.34.

8. **المخطط الاعتيادي Regular Graph**: هو المخطط الذي تكون جميع عقدة لها نفس الدرجة. في اي مخطط عندما تكون درجة جميع العقد فيه تساوي (k), عليه فان المخطط يسمى (مخطط اعتيادي من الدرجة k). في الشكل 10.35 فان المخطط والذي جميع عقده لها الدرجة 2 يسمى (مخطط اعتيادي – 2).



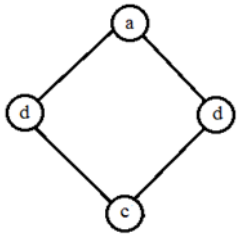
شكل 10.35: مخطط اعتيادي من الدرجة 2.

9. **المخطط الكامل Complete Graph**: هو المخطط الذي تكون فيه كل عقدة مرتبطة مع جميع العقد في المخطط. لاحظ الشكل 10.36.



شكل 10.36: مخطط كامل.

10.**مخطط دائري Cycle Graph**: المخطط البسيط الذي له (n) من العقد بحيث (n>=3), وله (n) من الروابط يسمى مخطط دائري اذا جميع روابطه عملت دائرة بطول (n). الشكل 10.37 له اربع عقد واربع روابط ويكون دائرة بطول 4.



شكل 10.37: مخطط دائري.

10.7 طرق المرور على المخطط Graph Traversal

هناك طريقتين ممكنه للمرور على المخططات:

المرور بالعمق اولا Depth First Search

المرور جانبيا اولا Breadth First Search

10.7.1 المرور بطريقة العمق اولا Depth First Traversal

خوارزمية البحث بطريقة العمق اولا تمر على المخطط بحركة باتجاه العمق ويستخدم المكدس لتحديد العقدة اللاحقة لبدا البحث عند حصول نهاية مغلقة او ميتة في اي عملية تكرار.

المرور على كل العقد ابتداء من عقدة بداية مقترحة (نحن من يحدد نقطة البداية).
عليه, فان المرور سيتبع مبدا المجاور اولا مع الاخذ بنظر الاعتبار مايلي:

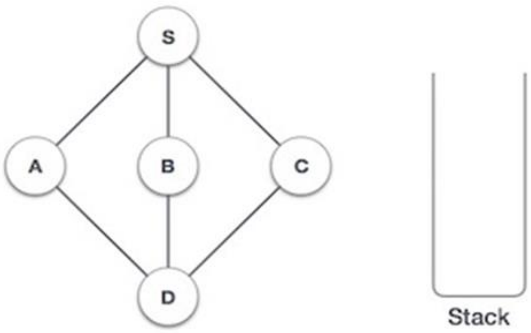
- ✓ عدم المرور على أي عقدة لأكثر من مرة واحدة فقط.
- ✓ فقط تلك العقد التي من الممكن الوصول لها يتم زيارتها.
- ✓ من المهم هنا, ان نستخدم مكدس هياكل البيانات. سيتم اضافة العقد التي يتم زيارتها الى المكدس بترتيب يتم سحبها من المكدس بحيث ان الداخل اخيرا يخرج اولا.

زيارة العقدة يتضمن اعادة البيانات التي في العقدة واضافة مجاورها الى المكدس.
على كل, فاننا لانعمل أي فعل في الحالات التالية:
اذا كان المجاور اساسا في المكدس, او اذا كان المجاور قد سبق وان تمت زيارته.

❖ **خوارزمية المرور**

القاعدة الاولى: تحديد عقدة البداية, زيارة العقد المجاورة لها التي لم يتم زيارتها.
القاعدة الثانية: اذا لم يتم ايجاد عقد مجاورة, يتم سحب عقدة من المكدس. (يتم سحب جميع العقد من المكدس التي ليس لها عقد مجاورة. اي يتم السحب لحين ان نجد عقدة لها مجاور او لحين ان يفرغ المكدس).
القاعدة الثالثة: اعد القاعدة الاولى والثانية لحين ان يفرغ المكدس.
ملاحظة: العقدة التي يتم زيارتها يفضل تعليمها او تاشيرها لتسهيل العمل.
لنتابع هذه الخوارزمية باستخدام الشكل 10.38.

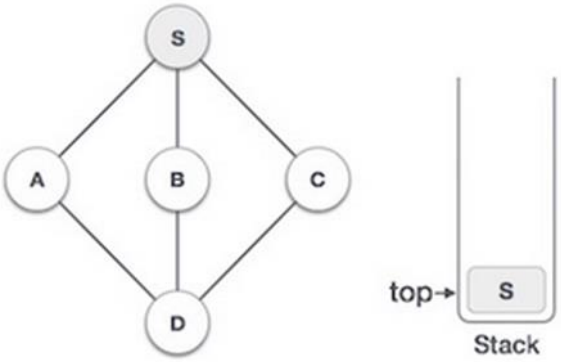
C++ ام.



الشكل 10.38: مثال لمخطط مع مكدس فارغ

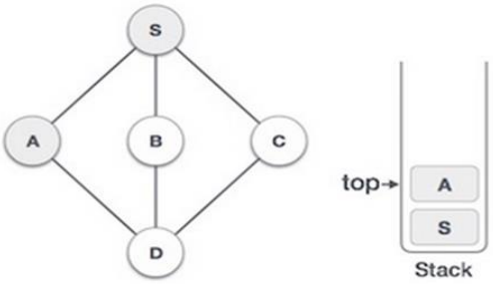
1. بداية ابدأ بمكدس فارغ.
2. يتم زيارة اول عقدة وهي العقدة (S) وندفعها في المكدس, ويتم تعليمها على انها تمت زيارتها. كما في الشكل 10.39.

نبحث عن اي عقدة مجاورة للعقدة (S). هناك ثلاث عقد مجاورة لها وهي (A, B, C) ومن الممكن ان نختار اي واحدة منهم. في هذا المثال سوف نختار العقدة حسب حروف الهجاء, وهي العقدة (A).



شكل 10.39: المكدس بعد زيارة اول عقدة.

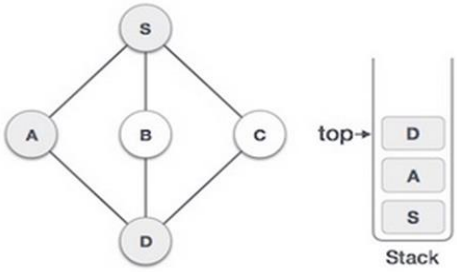
3. تؤشر العقدة (A) على انها تمت زيارتها وادفعها في المكدس كما في الشكل 10.40. ابحث عن اي عقدة مجاورة للعقدة (A). في هذه الحالة فان العقدتين (S, D) كلاهما تجاور العقدة (A) لكن من المفروض ان نختار العقدة التي لم يتم زيارتها وبالتالي سيتم اختيار العقدة (D).



شكل 10.40: المكدس بعد اضافة العقدة الثانية.

4. الان جاء دور زيارة العقدة (D) وتدفع الى المكدس بعد ان تعلم على انها تمت زيارتها, كما في الشكل 10.41.

نبحث عن العقد المحاورة لها وسنجد ان العقدتين (B, C) كلاهما تجاور العقدة (D) وكلاهما لم يتم زيارتهما. وسنختار حسب الحروف الابجدية, لتكون العقدة (B) هي العقدة المختارة.

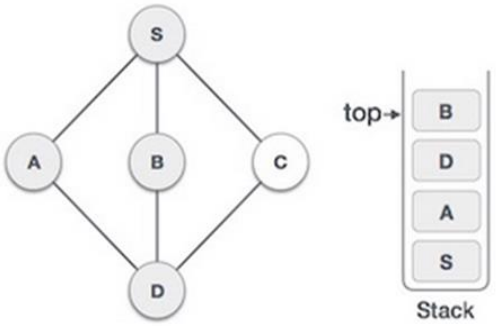


شكل 10.41: المكدس بعد عملية التحديث.

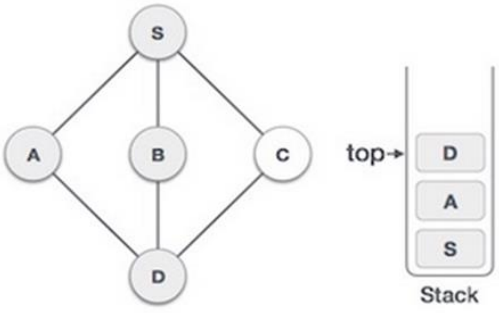
5. يتم زيارة العقدة (B) وتعلم على انها تمت زيارتها وتدفع الى المكدس فيكون شكل المكدس كما في الشكل 10.42.

هنا عند الفحص عن العقد المجاورة سوف لا نجد اي عقدة مجاورة للعقدة (B) لم يتم زيارتها سابقا. في هذه الحالة يتم سحب العقدة التي في اعلى المكدس (B), وتفحص العقدة التي بعدها (D) ان كانت لها عقدة مجاورة لم يتم

زيارتها, سنجد هنا العقدة (C) (اذا لم نجد نستمر بسحب العقد التي في اعلى المكدس). سيكون شكل المكدس بعد سحب العقدة التي في الاعلى (B) كما في الشكل 10.43.



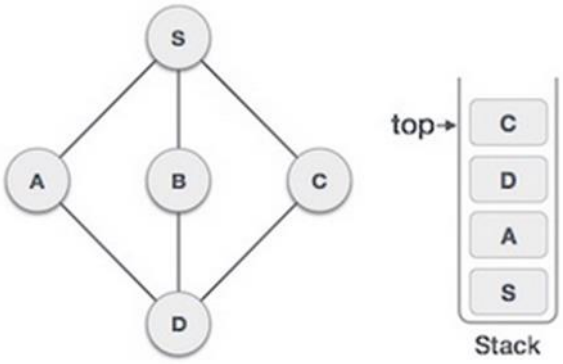
شكل 10.42: المكدس بعد اضافة عقدة



شكل 10.43: المكدس بعد سحب عقدة.

6. تتم زيارة العقدة (C) وتعلم على انها تمت زيارتها ثم تدفع الى المكدس, كما هو واضح من الشكل 10.44.

يتم البحث عن العقد المجاورة والتي لم تتم زيارتها. ونظرا لعدم وجود عقدة مجاورة للعقدة (C) لم تتم زيارتها نسحب العقدة التي في اعلى المكدس (C). نستمر بسحب العقدة التي في الاعلى واحدة بعد الاخرى (وفي كل مرة يتم فحص العقدة في اعلى المكدس ان كان لها مجاور) لحين ان يتم العثور على عقدة لها مجاور لم يتم زيارته. في مثالنا هذا سيتم سحب جميع العقد دون ان نعثر على عقدة مجاورة ولذلك متى ما فرغ المكدس فان عملية المرور على عقد المخطط تكون قد تمت.



شكل 10.44: المكدس بعد دفع العقدة (C) فيه.

✓ نتيجة المرور ستكون S A D B C

❖ برنامج المرور على المخطط بطريقة **Depth First** (يركز على المثال اعلاه).

```
#include <stdio>
#include <stdlib>
#include <stdbool>
#define MAX 5
using namespace std ;
struct Vertex {
  char label;
  bool visited;
};

// متغيرات المكدس
int stack[MAX];
int top = -1;

// مصفوفة العقد
struct Vertex* lstVertices[MAX];
```

```
// مصفوفة التجاور
int adjMatrix[MAX][MAX];

// حساب عدد العقد
int vertexCount = 0;

// دوال المكدس

void push(int item) {
  stack[++top] = item;
}

int pop() {
  return stack[top--];
}

int peek() {
  return stack[top];
}

bool isStackEmpty() {
  return top == -1;
```

```
}

// دوال المخطط

// اضافة عقدة الى قائمة العقد
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}

// اضافة رابط الى مصفوفة الروابط
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}
// عرض العقدة
void displayVertex(int vertexIndex) {
   cout<< lstVertices[vertexIndex]->label);
}

// استدعاء عقدة مجاورة لم يتم زيارتها
```

```
int getAdjUnvisitedVertex(int vertexIndex) {
  int i;

  for(i = 0; i<vertexCount; i++) {
    if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
{
      return i;
    }
  }

  return -1;
}

void depthFirstSearch() {
  int i;

  // تعليم العقدة الاولى على انها تمت زيارتها
  lstVertices[0]->visited = true;

  // عرض العقدة
  displayVertex(0);
```

```
  // دفع دليل العقدة الى المكدس
  push(0);

  while(!isStackEmpty()) {
    // استدعاء العقد المجاورة التي لم يتم زيارتها للعقدة التي في اعلى المكدس
    int unvisitedVertex = getAdjUnvisitedVertex(peek());

    // عدم وجود عقدة مجاورة
    if(unvisitedVertex == -1) {
      pop();
    }else {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      push(unvisitedVertex);
    }
  }

  // المكدس فارغ, انتهاء عملية البحث, تحديث عمليات التعليم السابق
  for(i = 0;i < vertexCount;i++) {
    lstVertices[i]->visited = false;
  }
}
```

```
int main() {
   int i, j;

   for(i = 0; i<MAX; i++) // ضبط المجاورات
{
      for(j = 0; j<MAX; j++)
         adjMatrix[i][j] = 0;
   }

   addVertex('S');   // 0
   addVertex('A');   // 1
   addVertex('B');   // 2
   addVertex('C');   // 3
   addVertex('D');   // 4

   addEdge(0, 1);    // S - A
   addEdge(0, 2);    // S - B
   addEdge(0, 3);    // S - C
   addEdge(1, 4);    // A - D
   addEdge(2, 4);    // B - D
   addEdge(3, 4);    // C - D
```

```
  cout<< "Depth First Search: " ;

  depthFirstSearch();

  return 0;
}
```

نتيجة البرنامج اعلاه هي : S A D B C

10.7.2 المرور الجانبي اولا Breadth First

ان خوارزمية البحث الجانبي اولا ستمر على المخطط بالتحرك على كل مستوى في المخطط ويستخدم الطابور للتذكير بالعقد التي تاتي لاحقا لبدا البحث عند الوصول الى نهاية ميتة.

الخوارزمية تمر على جميع العقد, بداية من عقدة بداية محددة. هنا, المرور يتبع مبدا التجاور اولا مع الاخذ بنظر الاعتبار مايلي:

- ✓ لا تتم زيارة عقدة اكثر من مرة واحدة فقط.
- ✓ فقط العقد التي من الممكن الوصول لها يتم زيارتها.

مهم هنا, ان نستخدم طابور هياكل البيانات. سيستخدم لحفظ العقد التي يتم زيارتها اخيرا بترتيب الاضافة الى الطابور بمعنى من يدخل اولا يخرج اولا.

زيارة العقدة يتضمن اعادة البيانات التي في العقدة واضافة العقدة التي تجاورها الى الطابور. على كل, سوف لا يتم اتخاذ أي فعل تحت الشروط التالية:

- ✓ اذا كان المجاور موجودا اساسا في الطابور.
- ✓ اذا المجاور سبق وان تم زيارتة.

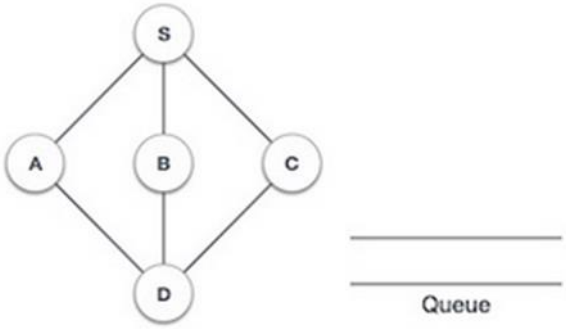
❖ **خوارزمية البحث الجانبي**

القاعدة الاولى: زيارة العقدة المجاورة. اشرها بانها تمت زيارتها. اعرضها, قم بحشرها في الطابور.

القاعدة الثانية: اذا لم تجد عقدة مجاورة, ازل اول عقدة من الطابور.

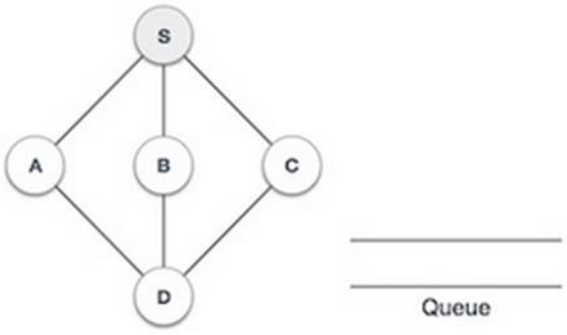
القاعدة الثالثة: كرر القاعدة الاولى والثانية لحين ان يفرغ الطابور.

لنشرح هذه الطريقة باستخدام الشكل والذي يظهر فيه المخطط الذي نرغب المرور عليه, تم انشاء طابور فارغ واضح بالشكل 10.45.



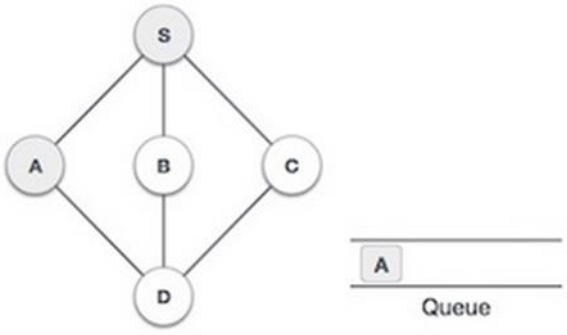
شكل 10.45: المخطط الذي سيتم زيارة عقده وكذلك تمثيل لطابور فارغ.

1. تتم عملية المرور بداية بزيارة العقدة الاولى هنا (S), وعند زيارتها تعلم (نغير لونها فقط للمتابعة, برمجيا من الممكن ان تكون علامتها true), كما في الشكل 10.46.



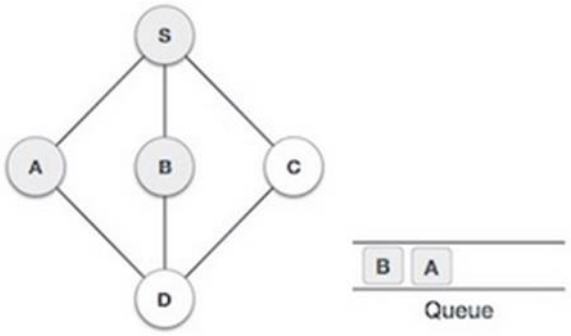
شكل 10.46: المخطط بعد زيارة العقدة الاولى.

2. يتم البحث عن العقدة المجاورة, لاحظ وجود ثلاث عقد مجاورة (A, B, C) ويتم الاختيار وفقا للحروف الابجدية فتكون العقدة التالية هي العقدة (A). يتم تعليم العقدة (A) وحشرها في الطابور كما في الشكل 10.47.



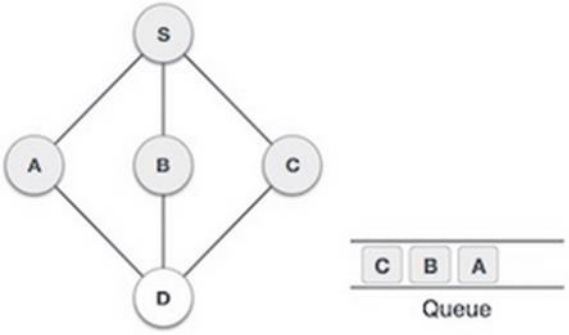
شكل 10.47: الطابور بعد التحديث.

3. العقدة اللاحقة والتي هي مجاورة للعقدة (S) وحسب التسلسل الابجدي ستكون العقدة (B), سيتم زيارتها وتعليمها على انها تمت زيارتها وتحشر في الطابور كما في الشكل 10.48 .



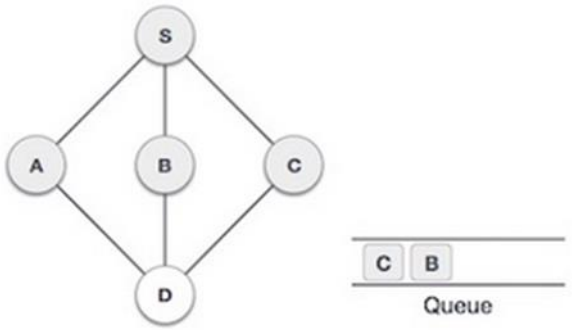
شكل 10.48: الطابور بعد التحديث

4. العقدة الاخرى المجاورة للعقدة (S) والتي لم يتم زيارتها بعد هي العقدة (C) فيصار الى زيارتها وتعلم على انها تمت زيارتها, ثم تحشر في الطابور, كما في الشكل 10.49.



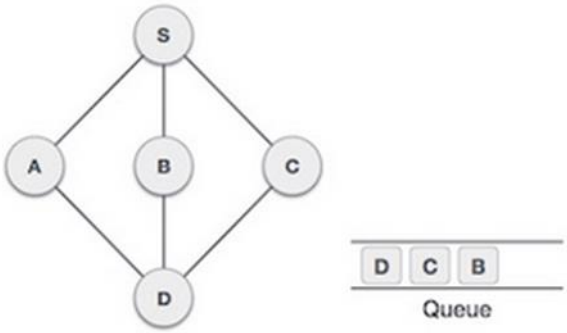
شكل 10.49: الطابور بعد حشر عقدة جديدة.

5. الان العقدة (S) ليس لها اي عقدة مجاورة لم يتم زيارتها لذلك يتم سحب عقدة من الطابور وهنا سيتم سحب العقدة (A). ويكون شكل الطابور كما في الشكل 10.50.



شكل 10.50: الطابور بعد سحب عنصر منه.

6. الان بالنسبة للعقدة التي تم سحبها من الطابور نفحص عن مجاوراتها فنجد العقدة (D) مجاورة لها, لذلك تعلم هذه العقدة على انها تمت زيارتها, وتحشر في الطابور. كما في الشكل 10.51.



شكل 10.51: الطابور بعد التحديث.

7. الان لا توجد عقد مجاورة للعقدة (A) لم يتم زيارتها, فيتم سحب عقدة من الطابور وتفحص ان كان لها مجاورات لم يتم زيارتها. هذه العملية تستمر لحين ان نجد عقدة مجاورة لم يتم زيارتها او يفرغ الطابور وبذلك تكون عملية المرور قد انتهت.
   - ✓ نتيجة عملية المرور ستكون S A B C D

❖ برنامج المرور على المخطط باستخدام طريقة **Breadth First** (سنركز على المثال اعلاه كنموذج للبرمجة).

```
#include <stdio>
#include <stdlib>
#include <stdbool>
#define MAX 5
using namespace std ;
struct Vertex {
  char label;
  bool visited;
};

متغيرات الطابور //

int queue[MAX];
int rear = -1;
```

```
int front = 0;
int queueItemCount = 0;

// متغيرات المخطط

// مصفوفة العقد
struct Vertex* lstVertices[MAX];

// مصفوفة المجاورات
int adjMatrix[MAX][MAX];

// حساب عدد العقد
int vertexCount = 0;

// دوال الطابور
void insert(int data) {
  queue[++rear] = data;
  queueItemCount++;
}

int removeData() {
  queueItemCount--;
  return queue[front++];
```

```
}

bool isQueueEmpty() {
   return queueItemCount == 0;
}

// دوال المخطط

// اضافة عقدة الى قائمة العقد
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}

// اضافة رابط الى مصفوفة الروابط
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}
```

```
// عرض العقدة
void displayVertex(int vertexIndex) {
cout<< lstVertices[vertexIndex]->label ;
}

// استدعاء العقدة المجاورة التي لم يتم زيارتها
int getAdjUnvisitedVertex(int vertexIndex) {
  int i;

  for(i = 0; i<vertexCount; i++) {
    if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
      return i;
  }

  return -1;
}
void breadthFirstSearch() {
  int i;

  // تعليم العقدة الاولى على انها تمت زيارتها
  lstVertices[0]->visited = true;

  // عرض العقدة
```

```
displayVertex(0);

// اضافة دليل العقدة الى الطابور
insert(0);
int unvisitedVertex;

while(!isQueueEmpty()) {
  // استدعاء العقد التي لم يتم زيارتها للعقدة التي في بداية الطابور
  int tempVertex = removeData();

  // عدم وجود عقد مجاورة
  while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1)
{
    lstVertices[unvisitedVertex]->visited = true;
    displayVertex(unvisitedVertex);
    insert(unvisitedVertex);
  }
}
// الطابور فارغ, انتهاء عملية المرور , ضبط اشارات التعليم
for(i = 0;i<vertexCount;i++) {
  lstVertices[i]->visited = false;
}
```

```
}
int main() {
  int i, j;

  for(i = 0; i<MAX; i++) // ضبط المجاورات
{
    for(j = 0; j<MAX; j++)
      adjMatrix[i][j] = 0;
  }
  addVertex('S');   // 0
  addVertex('A');   // 1
  addVertex('B');   // 2
  addVertex('C');   // 3
  addVertex('D');   // 4
   addEdge(0, 1);    // S - A
  addEdge(0, 2);    // S - B
  addEdge(0, 3);    // S - C
  addEdge(1, 4);    // A - D
  addEdge(2, 4);    // B - D
  addEdge(3, 4);    // C - D

   cout<< "\nBreadth First Search: " ;
   breadthFirstSearch();
```

```
  return 0;
}
```

نتيجة البرنامج هي: S A B C D

10.8 العمليات الاولية الاساسية التي تجرى على المخططات

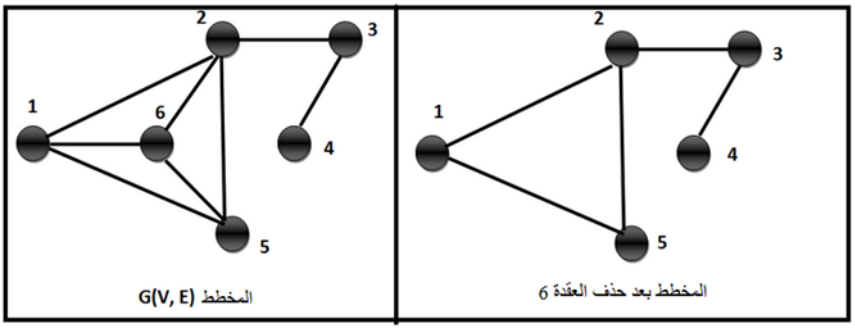
هناك عدد من العمليات التي تجرى على المخططات, غالبا هذه العمليات تؤدي الى تحوير المخططات. سنحاول ان نتطرق الى اغلب هذه العمليات. بعض العمليات ستكون سهلة جدا وواضحة عند التطرق لتمثيل المخططات.

1. حذف عقدة Delete vertex

ماذا يحدث عند حذف عقدة من المخطط G(V, E)؟

لا يمكننا ان نحذف عقدة بنجاح اذا لم يتم ازالة (حذف) كل الروابط التابعة لهذه العقدة. هذه العملية تغير مجموعة العقد وعائلة الروابط للمخطط. لذلك عند حذف عقدة فان كل الروابط المتعلقة بهذه العقدة يتم حذفها ايضا.

الشكل 10.52 يساعد في فهم حذف العقدة.

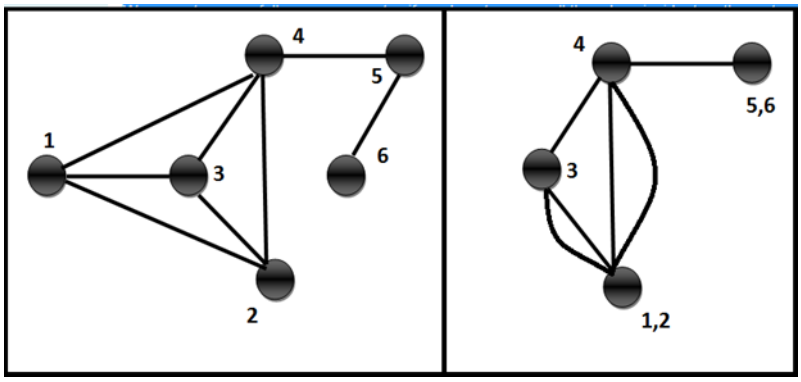


شكل 10.52: توضيح عملية حذف عقدة من المخطط.

2. دمج العقد contract vertex

واحدة من العمليات المهمة في المخططات هي دمج العقد. من الممكن انجازها بدمج اثنان او اكثر من العقد بعقدة واحدة. كذلك, من الممكن ان تنجز بدمج الروابط, والتي سنراها لاحقا.

ليس من الممكن دمج عقدة واحدة فقط, ان عملية الدمج تحتاج الى مجموعة من العقد على الاقل اثنان. وعملية الدمج تنجز عندما يكون هناك رابط بين العقدتين المراد دمجهما. العملية اساسا ستزيل كل الروابط بين العقدتين. في الشكل 10.53 فان العقدتين (1, 2) تم دمجهما, والعقدتين (5, 6) ايضا تم دمجهما. لاحظ ان الرابط الذي يربط العقدتين قبل الدمج يتم ازالته عند الدمج.



شكل 10.53: يوضح طريقة دمج العقد

3. دمج الرابط contracting an edge

دمج رابط هو مشابه بالضبط لدمج العقد. فهو يزيل الرابط من المخطط ويدمج العقد التي يربطها هذا الرابط.

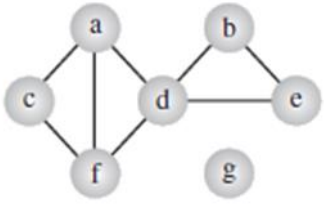
4. اضافة وحذف رابط Add and Delete Edge

اضافة رابط الى المخطط من الممكن انجازه بربط عقدتين مطلوب اضافة الرابط بينهما. حذف رابط من الممكن عمله بازالة الرابط بين عقدتين مقترحة.

10.9 تمثيل المخطط Graph Representation

هناك عدة طرق لتمثيل المخططات, نحاول ان ندرج اهم هذه الطرق والاكثر شيوعا.

1. قائمة التجاور: ان ابسط تمثيل يكون من خلال ما يسمى قائمة التجاور والتي تحدد كل العقد المجاورة لكل عقدة في المخطط. هذه القائمة من الممكن تنفيذها على شكل جدول كما في الشكل 10.54.

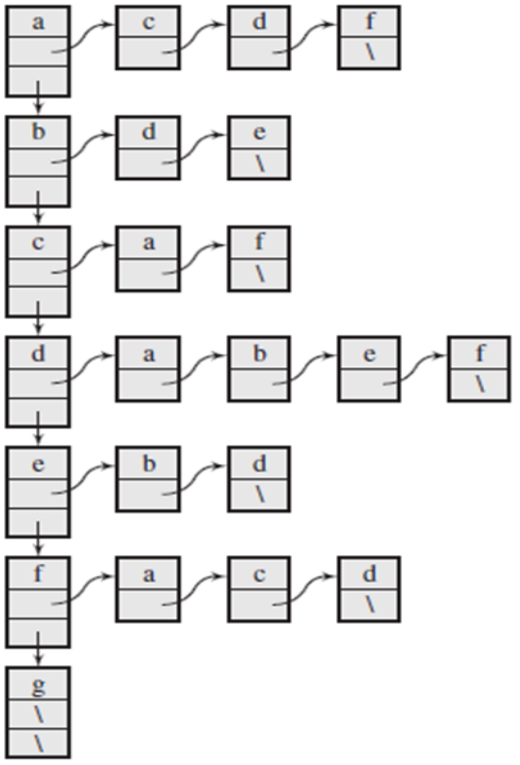


شكل 10.54: مخطط عام.

a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

شكل 10.55: يوضح قائمة التجاور للمخطط في الشكل 10.54

2. القوائم الموصولة: كذلك من الممكن تمثيلها على شكل قوائم موصولة, كما في الشكل 10.56.



هياكل البيانات

شكل 10.56: مصفوفة التجاور للمخطط في الشكل 10.54

3. مصفوفة المتجاورات Adjacency matrix

مصفوفة المتجاورات هي طريقة لتمثيل (n) من العقد في المخطط G(V, E) بمصفوفة حجمها (n × n), بحيث ان جميع مدخلاتها هي قيم منطقية (0, 1). وهي واحدة من طرق تمثيل المخططات البسيطة.

يتم ضبط (a_{ij} = true) اذا كان هناك رابط بين العقدتين (i, j).

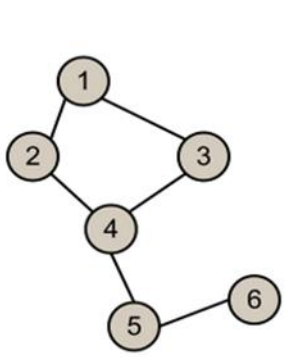
ان مدخلات المصفوفة (a[i][j]) من الممكن تعريفها وفقا لما يلي:

$$a[i][j] = \begin{cases} true & if\ (i,j) \in E \\ false & otherwise \end{cases}$$

- ✓ الروابط من الممكن ان تكون غير متجه, في حالة (a_{ij} = true) وكذلك (a_{ji} = true) اي ان ($a_{ji} = a_{ij}$).

بداية يتم خزن المصفوفة (مصفوفة ثنائية) بحجم N×N حيث ان N هو عدد العقد في المخطط. هذا يعني اذا كان هناك رابط بين العقدتين تكون قيمته 1 واذا لم يكن هناك رابط تكون القيمة صفر. لاحظ الشكل 10.57.

$$a_{ij} = \begin{cases} 1 & \text{اذا كان هناك رابط بين } (v_i, v_j) \\ 0 & \text{للحالات الاخرى} \end{cases}$$



مخطط غير متجه

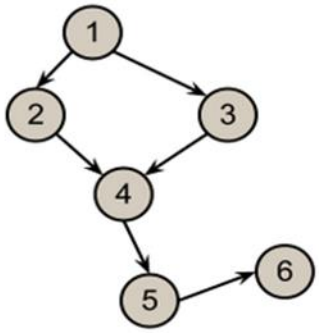
	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

مصفوفة التجاور

شكل 10.57: مصفوفة التجاور لمخطط غير المتجه

- ✓ او يكون المخطط متجه في حالة ان تكون ($a_{ij} \neq a_{ji}$).

فـي المخططـات المتجـه مـن الممكـن ان نسـتخدم 1 عنـد وجـود رابـط بـين (i, j) ونستخدم -1 بالعكس اي عندما يكون الرابط بين (j, i).



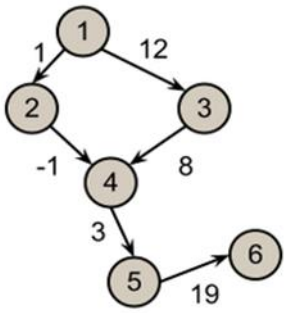
مخطط متجه

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	-1	0	0	1	0	0
3	-1	0	0	1	0	0
4	0	-1	-1	0	1	0
5	0	0	0	-1	0	1
6	0	0	0	0	-1	0

مصفوفة التجاور

شكل 10.58: مصفوفة التجاور لمخطط متجه.

- ✓ في المخططات الموزونة المتجهة, نضع الوزن على الرابط المتجه بدلا من 1.



مخطط موزون متجه

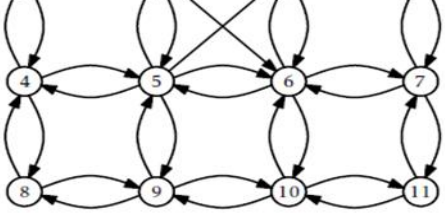
	1	2	3	4	5	6
1	0	1	12	0	0	0
2	-1	0	0	-1	0	0
3	-12	0	0	8	0	0
4	0	1	-8	0	3	0
5	0	0	0	-3	0	19
6	0	0	0	0	-19	0

مصفوفة التجاور

شكل 10.59: مصفوفة التجاور لمخطط موزون متجه.

في هذا التمثيل, فان عمليات اضافة رابط, حذف رابط, وفحص وجود رابط فقط تتضمن اضافة قيم للمصفوفة او قراءة مدخلات المصفوفة (a[i][j]) كما يلي:

```
void addEdge(int i, int j) {
a[i][j] = true;
}
void removeEdge(int i, int j) {
a[i][j] = false;
}
bool hasEdge(int i, int j) {
return a[i][j];
}
```



هياكل البيانات باستـ

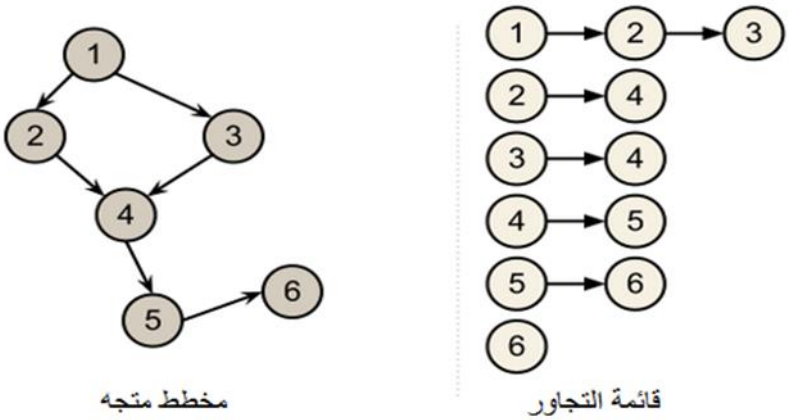
	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

شكل 10.60: مخطط متجه مع مصفوفة التجاور له

4. قوائم التجاور Adjacency Lists

هناك عد طرق لتنفيذ قوائم التجاور. في هذا الفصل سنقدم احدى الطرق البسيطة.

قوائم التجاور هي طريقة تمثيل للمخططات مفيدة, في هذه الطريقة فان المخطط G(V, E) سيمثل على شكل مصفوفة قوائم, حيث ان كل مدخل للمصفوفة يمثل احدى عقد المخطط والتي تمثل عقدة الراس لقائمة تشمل جميع العقد المرتبطة بها. نفرض ان المصفوفة التي تمثل المخطط تسمى (ad) (وهي مصفوفة قوائم), لذلك فان (ad[i]) تمثل قائمة تحتوي على جميع العقد المجاورة للعقدة (i). عليه, هي تحتوي على كل الفهارس (ارقام العقد) بحيث ان ((i, j) ∈ E) (اي وجود رابط بين (i, j)), كما في الشكل 10.61.



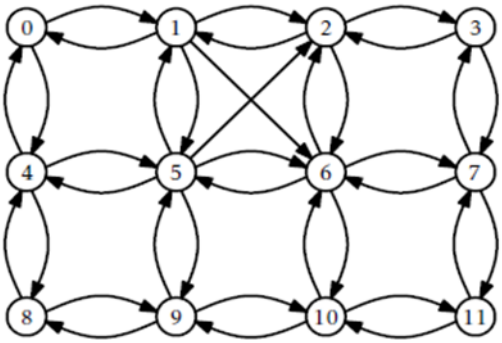
شكل 10.61: قائمة التجاور لمخطط متجه.

في طريقة قوائم التجاور فان اضافة رابط بين (i, j) تتطلب فقط اضافة العقدة (j) الى القائمة (ad[i]).

```
int n;
List *ad;
void addEdge(int i, int j) {
ad[i].add(j);
}
```

عملية ازالة رابط والذي يربط بين العقدتين (i, j) يتم باجراء عملية البحث اولا في القائمة (ad[i]) لحين ايجاد (j) وعندها يتم ازالتها.

```
void removeEdge(int i, int j) {
for (int k = 0; k < adj[i].size(); k++) {
if (adj[i].get(k) = = j) {
adj[i].remove(k);
return;
} } }
```



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

شكل 10.62: مخطط مع قوائم التجاور التي تمثله

5. النوع الاخر من المصفوفات هي مصفوفة الحدوث (incidence matrix) وهي تمثل العلاقة بين العقد والروابط ووفقا للعلاقة التالية (الشكل 10.63 يوضح ذلك):

$$a_{ij} = \begin{cases} 1 & \text{اذا كان الرابط } e_j \text{ له علاقة مع العقدة } v_i \\ 0 & \text{للحالات الاخرى} \end{cases}$$

	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

الشكل 10.63: يوضح مصفوفة الحدوث للمخطط في الشكل 10.52.

10.10 خوارزمية ديكسترا Dijkstra's algorithm

هي خوارزمية لايجاد اقصر طريق (الاقل وزنا) بين اثنان من العقد المحددة في مخطط موزون لا يحتوي على روابط لها اوزان سالبة.

- ✓ هي تحل مشكلة المسار الاقصر لعقدة واحدة
- ✓ مفاهيم الخوارزمية الاساسية: خلق جدول معلومات حول افضل الطرق المعروفة حاليا للوصول الى كل عقدة (تشمل المسافة, العقدة السابقة) وتحسينها لحين الحصول على افضل حل.
- ✓ ابسط عملية تنفيذ لهذه الخوارزمية هو بحفظ العقد في قائمة موصولة او مصفوفة.

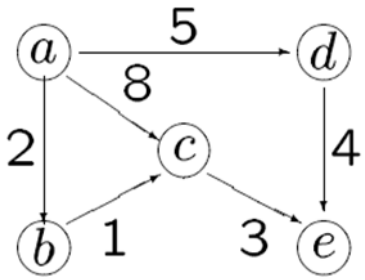
ان طول المسار ($p = (v_0, v_1, v_2 \dots v_k)$) هو مجموع الاوزان لروابطها المتتالية.

Length (p) = $\sum_{i=1}^{k} w(v_{i-1} \quad , \quad v_i \quad)$

المسافة بين (u, v) تمثل بالترميز ($\delta(u, v)$) وهو طول اقصر مسار اذا كان هناك مسار بين (u, v), وبخلاف ذلك فالطول يساوي (∞).

في الشكل 10.64, فان المسافة (p= (a, b, c, e)) تساوي (6).

وذلك لان اقصر مسافة بين (a, e) تساوي (6).



شكل 10.64: مخطط موزون.

في المخطط من الممكن ان تكون:

- ✓ العقد تمثل المدن.
- ✓ اوزان الروابط تمثل المسافة بين مدينتين (بالسيارة) (اذا كان هناك طريق يربط بين المدينتين).

خوارزمية **Dijkstra** تستخدم لايجاد الطريق الاقصر بين مدينة واي مدينة اخرى.

10.10.1 خطوات الخوارزمية :Dijkstra(V1, V2)

1. ان هدف الخوارزمية هو المحافظة على (d[V]) لان يكون بطول δ (S, V) والذي هو اقصر مسار بين عقدة المصدر وكل عقدة (V).
2. دائما (δ (S, V) ≤ d[V]) وان المسافة d[V] تساوي طول المسار المعلوم, لان (d[V] = ∞) اذا لم يكن هناك مسار.
3. بداية يتم تحديد العقدة المصدر وهي العقدة التي سيتم حساب المسافة نسبة لها (من الممكن اعتبارها عقدة البداية).
4. تحدد المسافة لكل عقد المخطط (d[V]), في البداية تكون مسافة عقدة المصدر تساوي صفر, بينما مسافة جميع العقد الاخرى تساوي ما لانهاية.
5. الخوارزمية تعالج العقد واحدة بعد الاخرى حسب ترتيب معين. هنا ان معالجة العقدة (u) مثلا تعني ايجاد المسارات الجديدة وتحديث المسافات (d[V]) لجميع العقد (v) والتي تنتمي الى مجاورات العقدة (u) اذا كان التحديث ضروري. ان عملية حساب المسافات والتحديث الذي يحدث بموجبه تسمى (relaxation). عند معالجة جميع العقد فان

(d[v] =δ (S, V)) اي اننا نكون قد حسبنا اقصر مسافة لكل العقد في المخطط.

6. عملية التحديث تتم في حالة ان يكون المسار من عقدة المصدر(S) الى العقدة (V) هو اصغر من المسافة (d[V]) الموجودة على العقدة (V), في هذه الحالة يتم وضع المسافة الجديدة للعقدة (V), بخلاف ذلك لا تتم عملية التحديث.
7. تتم عملية تنفيذ الخوارزمية وذلك بخزن العقدة في طابور الافضلية (priority queue) عند قرائتها, بحيث تكون القيمة المفتاحية لكل عقدة هي مسافتها d[V]. فتكون قيمة الافضلية للعقدة (S) تساوي صفر.

تخزن المسافة الى العقدة (S) (والتي هي صفر) وجميع العقد الاخرى (قيمتها الابتدائية ما لانهاية) في جدول خاص يسمى جدول المسافات.

8. يتم ايجاد العقدة (V) التي لها اقل قيمة (مسافة) من الطابور (سيكون موقع العقدة المختارة في بداية الطابور), ويتم ازالتها من الطابور.
9. نحدد العقد المجاورة (W) للعقدة (V). لكل عقدة من العقد المجاورة للعقدة (V) نقوم بما يلي:

A. حساب المسافة الجديدة من العقدة (V) الى العقدة المجاورة (W).

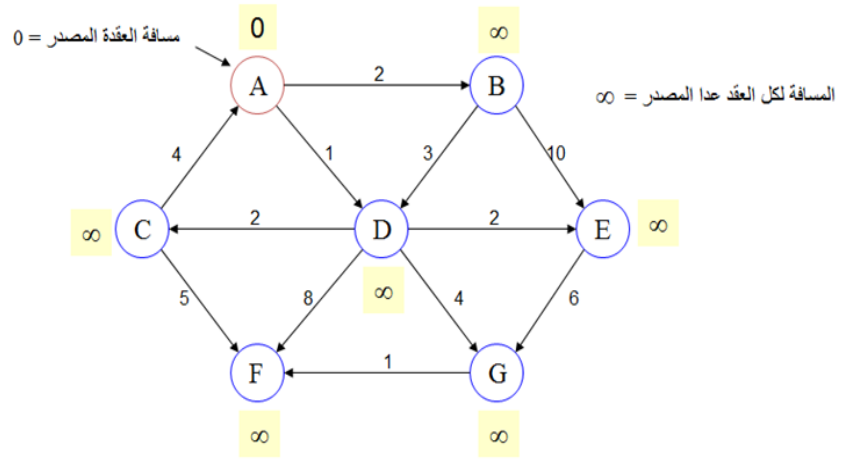
المسافة الجديدة (d) = المسافة الى العقدة (V) + وزن (V, W).

B. اذا كانت المسافة القديمة (المسافة التي خزنت في جدول المسافات) تساوي (∞) (اي انها لم تحسب سابقا) فنقوم بما يلي:
   I. خزن المسافة الجديدة (d) في جدول المسافات.
   II. حشر او خزن العقدة (W) في طابور الافضلية مع افضلية قيمتها (d).
   III. خزن (المسار = V)

C. اذا كانت المسافة التي تم حسابها (d) اصغر من المسافة القديمة, فنقوم بما يلي:
   I. تحديث المسافة القديمة للعقدة بحيث تكون قيمتها تساوي المسافة الجديدة (d).
   II. تحديث الافضلية للعقدة (W) بحيث تكون قيمتها تساوي (d).
   III. تحديث المسار الى (W) ليكون (V).

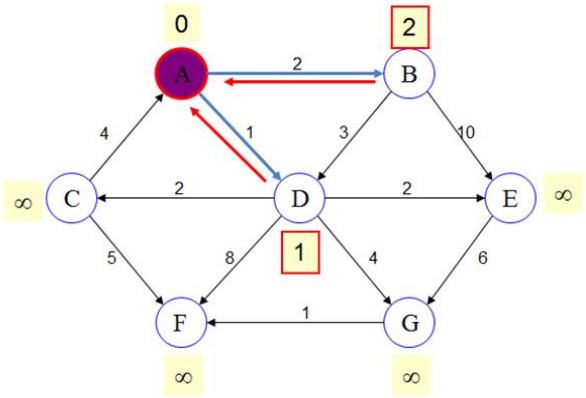
10.يتم فحص الطابور, اذا كان الطابور غير فارغ, العودة الى الخطوة (8). اما اذا كان الطابور فارغ فهذا يعني ان عملية ايجاد المسافات الاقصر قد انتهت.

مثال: مطلوب ايجاد المسافات الاقصر بين عقدة المصدر (A) وجميع العقد الاخرى في المخطط في الشكل 10.65.



شكل 10.65: مخطط موزون مطلوب ايجاد اقصر مسافة بين العقدة A وباقي العقد.

1. اختار العقدة التي لها اقل مسافة في القائمة, واضح هنا ان العقدة المصدر هي التي لها اقل مسافة وتساوي صفر. مجاورات العقدة (A) هي (B, D).



شكل 10.66: المخطط بعد اختيار العقدة المصدر وتحديد المجاورات.

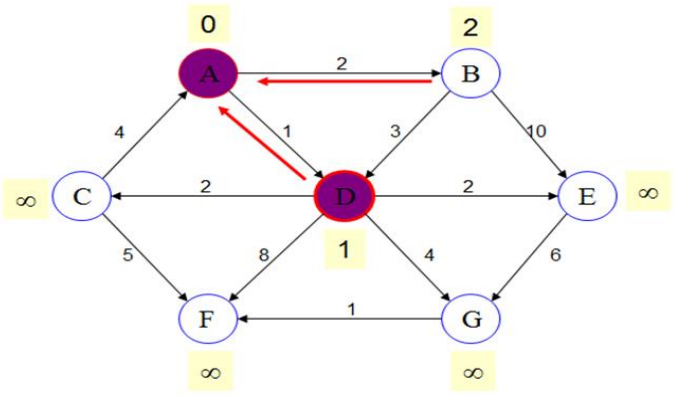
نقوم بتحديث المسافات لمجاورات هذه العقدة. كما في الشكل 10.67.

مسافة العقدة (B) = 2.

مسافة العقدة (D) = 1.

2. اختيار العقدة التي لها المسافة الاصغر من القائمة (هنا ستكون العقدة D).

مجاورات العقدة (D) هي (C, E, F, G).



شكل 10.67: اختيار العقدة ذات المسافة الاصغر.

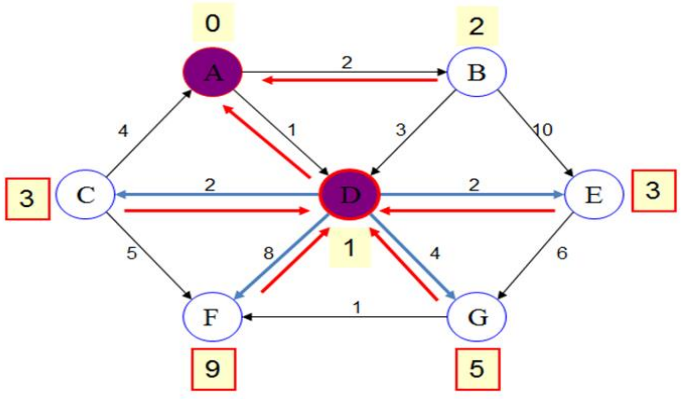
3. تحديث المسافات للعقد المجاورة للعقدة (D). لاحظ هنا ان المسافة تقاس اعتبارا من العقدة المصدر (A) مرورا بالعقدة (D) الى العقد المجاورة لها. انظر الشكل 10.68.

مسافة العقدة (C) = 1 + 2 = 3.

مسافة العقدة (E) = 1 + 2 = 3.

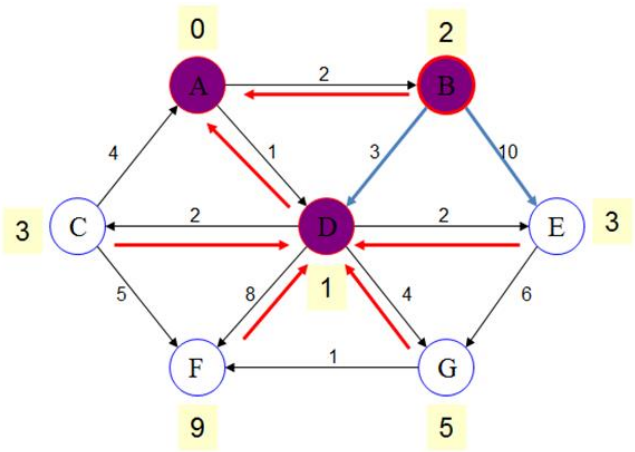
مسافة العقدة (F) = 1 + 8 = 9.

مسافة العقدة (G) = 1 + 4 = 5.



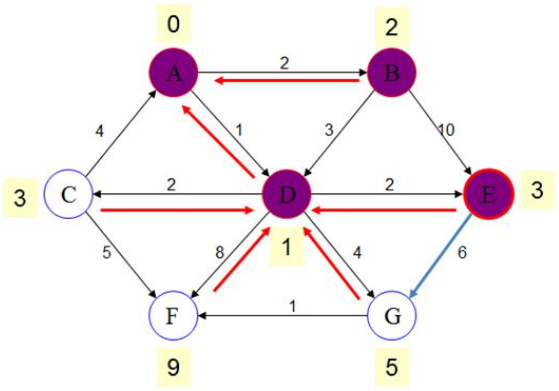
شكل 10.68: المخطط بعد حساب المسافات.

4. يتم اختيار العقدة التي لها اقل مسافة من القائمة بعد التحديث وهنا ستكون العقدة (B). يتم تحديث مسافات العقد المجاورة لها وهي العقد (D, E). لاحظ هنا ان المسافة (D) لا يتم تحديثها وذلك لانها معروفة اساسا, كذلك فان المسافة للعقدة (E) لا يتم تحديثها لان المسافة الجديدة اكبر من المسافة التي حسبت سابقا.



شكل 10.69: اختيار العقدة B وتحديث المسافات.

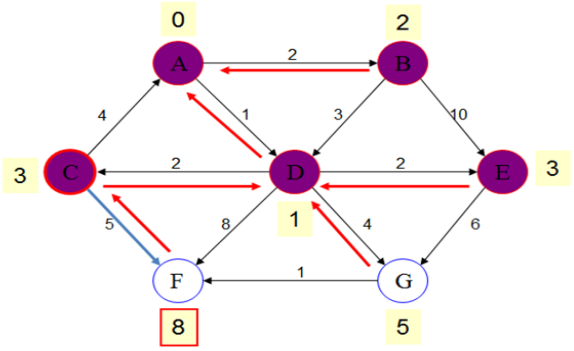
5. اختيار العقدة التي لها اقل مسافة من القائمة والتي هي العقدة (E) وتحديث مسافات المجاورات وهي العقد (G). في هذه الحالة لا يوجد تحديث لان مسافة العقدة اصغر من المسافة الجديدة. انظر الشكل 10.70.



شكل 10.70:اختيار العقد E.

6. اختيار العقدة التي لها اقل مسافة من القائمة وهي العقدة (C), وتحديث مجاوراتها. كما في الشكل 10.71.

مسافة العقدة (F) = 3 + 5 = 8.



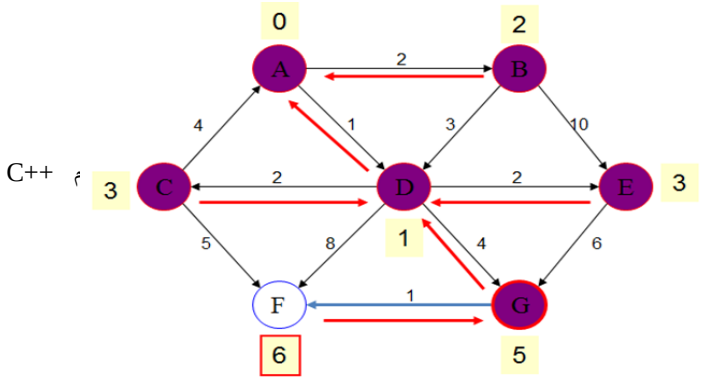
شكل 10.71: اختيار العقدة C وتحديث مجاوراتها.

7. اختيار العقدة التي لها اقل مسافة وهي العقدة (G) وتحديث مجاوراتها. الشكل 10.72.

مسافة العقدة (F) = القيمة الاصغر بين (1+5 ,8) = 6.

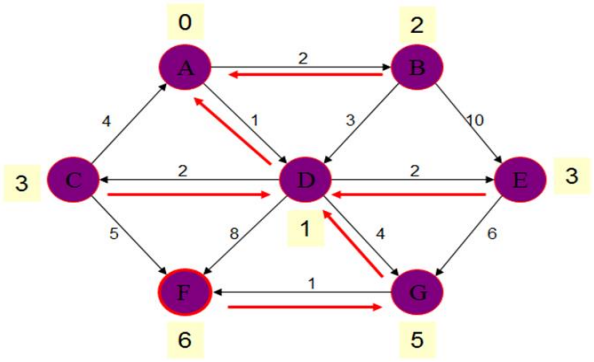
يتم اختيار المسافة الاقل بالنسبة للعقدة (F), حيث ان المسافة السابقة كانت (8) بينما المسافة الحالية هي (6).

C++ م



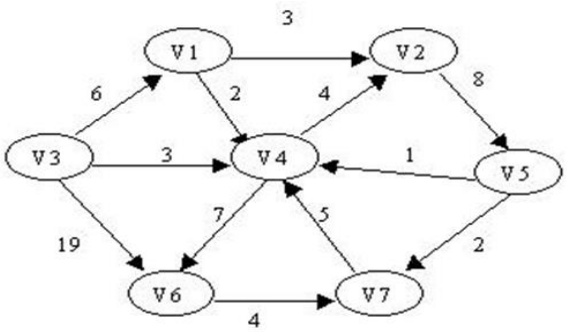
شكل 10.72: اختيار العقدة G وتحديث المسافات.

8. اختيار العقدة التي لها اقل مسافة وهي العقدة (F) وتحديث المجاورات.



شكل 10.73: اختيار العقدة F وتحديث المجاورات.

❖ **مثال اخر**: اكثر توضيحا وذلك باستخدام الجداول لتبسيط العمل وملاحظة الفارق مع المثال السابق. لدينا المخطط في الشكل 10.74 والمطلوب حساب اقصر مسار بين العقدة (V3) وجميع العقد الاخرى.



شكل 10.74: مخطط مطلوب حساب اقصر مسافة بين العقدة وباقي العقد.

رتب القيم الابتدائية على شكل جدول (وهي القيم المستسقاة من المخطط في الشكل 10.74 قبل اجراء اي عملية عليه).

جدول 10.1: القيم الابتدائية للمخطط في الشكل 10.71 .

العقد	جدول المسافات		قوائم المجاورات مع اوزانها
V1	∞	-	V2{3}, V4{2}
V2	∞	-	V5 {8}
V3	0	-	V1{6}, V4{3}, V6{19}
V4	∞	-	V2{4}, V6{7}
V5	∞	-	V4{1}, V7{2}
V6	∞	-	V7{4}
V7	∞	-	V4{5}

ملاحظة:

- ✓ سيتم استخدام DT(i, j) للاشارة الى الخلية في الصف (i) والعمود (j) في جدول المسافات (تمثل المسافة المحسوبة).
- ✓ كذلك فان (V:n) تعني ان المسافة من (V3) الى العقدة (V) تساوي (n).
- ✓ (PQ) سيرمز الى طابور الافضلية (Priority Queue) .

❖ **خطوات الحل:**

1. بداية أخزن (V3:0) في طابور الافضلية. سيكون الطابور على الشكل

PQ: V3:0

2. طالما الطابور ليس فارغا اعمل ما لي:

A. حذف العقدة التي لها اقل قيمة من الطابور وهي هنا (V3) (عادة العقدة التي لها اقل قيمة تكون في اقصى اليسار من طابور الافضلية).

قائمة المجاورات للعقدة (V3) هي ({V6{19}, V4{3}, V1{6}) كما في الجدول 10.1.

لكل واحد من هذه المجاورات أعمل ما يلي:

I. أقرأ العقدة المجاورة من القائمة V1{6} (العقدة المجاورة الاولى).

- أحسب المسافة الى العقدة V1 لانها لم تحسب من قبل (قيمتها ∞).
  - مسافة DT(1,1) = المسافة الى (V3) ..(ممثلة في DT(3,1)) + الرابط بين (V3, V1) = 6 (لاحظ هنا ان DT(3,1) تساوي صفر)
- سجل المسار, DT(1, 2) = V3 (ستمثل الصف الاول لانها V1 والعمود الثاني لانها تحديث للمسافة في جدول المسافات).
- خزن V1:6 في الطابور.

وضع الطابور سيكون

PQ: V1:6

II. أقرأ العقدة المجاورة الثانية **V4{3}**

- أحسب المسافة الى V4 لانها لم تحسب سابقا.

مسافة DT(4,1) = المسافة الى (V3) ..(متمثلة في DT(3,1)) + الرابط بين (V3, V4) = 3

- سجل المسار, DT(4, 2) = V3
- خزن V4:3 في الطابور.

وضع الطابور سيكون

PQ: V4:3, V1:6

III. أقرأ العقدة المجاورة الثالثة **V6{19}**

- أحسب المسافة الى V6 لانها لم تحسب سابقا.

مسـافة DT(6,1) = المسـافة الـى (V3) ..(متمثلـة فـي DT(3,1)) + الرابط بين (V3, V6) = 19

- سجل المسار, DT(6,2) = V3
- خزن V6:19 في الطابور.

وضع الطابور سيكون

PQ: V4:3, V1:6, V6:19

ملاحظـة: العناصـر فـي الطـابور مرتبـة حسـب الافضـلية (القيمـة الاقـل تمثـل افضـلية عليـا). لـذلك فـان عمليـة سـحب عنصـر مـن الطـابور سيكون دائمـا مـن اقصـى يسار الطابور.

جدول 10.2: جدول المسافات بعد معالجة مجاورات العقدة V3.

العقد	جدول المسافات	
V1	6	V3
V2	∞	
V3	0	
V4	3	V3
V5	∞	
V6	19	V3

V7	∞	

B. يتم سحب وازالة عقدة من بداية الطابور وستكون العقدة (**V4:3**), مجاورات هذه العقدة هي ((V2{4}, V6{7}), لكل واحد من مجاورات العقدة V4 قم بالاعمال التالية:

I. أقرأ العقدة المجاورة الاولى **V2{4}**

- أحسب المسافة الى V2 لانها لم تحسب سابقا.

مسافة DT(2,1) = المسافة الى (V4).. (متمثلة في DT(4,1)) + الرابط بين (V4, V2) = 7

- سجل المسار, DT(2,2) = V4
- خزن V2:7 في الطابور.

وضع الطابور سيكون

PQ: V1:6, V2:7, V6:19

II. أقرا العقدة المجاورة الثانية **V6{7}**

- المسافة الى V6 هي 19. وهي اكبر من المسافة الى V4 مضاف لها وزن الرابط (V4, V6), ولذلك يجب ان تحدث (عندما تكون المسافة المحسوبة اكبر من مسافة العقدة فيجب ان يتم تحديثها, اما اذا كانت اصغر فتترك بدون تحديث).

مسافة DT(6,1) = المسافة الى (V4).. (متمثلة في DT(4,1)) + الرابط بين (V4, V6) = 10

(المسافة DT(4,1) تساوي 3, لاحظ الجدول 10.2).

- سجل المسار, DT(6,2) = V4
- حدث الافضلية للعقدة V6 في طابور الافضلية لتكون قيمتها الجديدة هي 10.

وضع الطابور سيكون **PQ: V1:6, V2:7, V6:10**

جدول 10.3: جدول المسافات بعد معالجة مجاورات العقدة V4.

العقد	جدول المسافات	
V1	6	V3
V2	7	V4
V3	0	
V4	3	V3
V5	∞	
V6	10	V4
V7	∞	

C.يتم سحب وازالة عقدة من بداية الطابور وستكون العقدة (**V1:6**), مجاورات هذه العقدة هي (V2{3}, V4{2}). لكل واحد من مجاورات العقدة V1 اعمل ما يلي:

I. أقرأ العقدة المجاورة **V2{3}**

- المسافة الى العقدة V2 تم حسابها سابقا وهي تساوي 7 وهي اصغر من المسافة (الى V1 + وزن الرابط بين العقدتين (V1, V2 = (6+3), لذلك لا تعمل اي شيء.

II. أقرأ العقدة المجاورة الاخرى **V4{2}**

- المسافة الى العقدة V4 تم حسابها سابقا وهي تساوي 3, وهي اصغر من المسافة (الى V1 يضاف لها وزن الرابط بين العقدتين (V1, V4 = 6+2), لذلك لا تعمل اي شيء.

وضع الطابور سيكون **PQ: V2:7, V6:10**

D. يتم سحب وازالة عقدة من بداية الطابور وستكون العقدة (**V2:7**), مجاورات هذه العقدة هي (V5{8}). لكل واحد من مجاورات العقدة V2 تجرى الاعمال التالية:

I. أقرأ العقدة المجاورة **V5{8}**

- أحسب المسافة الى V5 لانها لم تحسب سابقا.

مسافة DT(5,1) = المسافة الى (V2).. (متمثلة في DT(2,1)) + الرابط بين (V2, V5) = 15

(المسافة DT(2,1) تساوي 7 كما في الجول 10.3).

- سجل المسار, DT(5,2) = V2
- خزن V5:15 في طابور.

وضع الطابور هو **PQ:** V6:10, V5:15

جدول 10.4: جدول المسافات بعد معالجة مجاورات العقدة V2.

العقد	جدول المسافات	
V1	6	V3
V2	7	V4
V3	0	

V3	3	V4
V2	15	V5
V4	10	V6
	∞	V7

وضع الطابور **PQ: V6:10, V5:15**

E. يتم سحب وازالة عقدة من بداية الطابور وستكون العقدة (**V6:10**), مجاورات هذه العقدة هي ({4}V7). لكل واحد من مجاورات العقدة V6 قم بالاعمال التالية:

I. أقرا العقدة المجاورة **V7{4}**

- أحسب المسافة الى V7 لانها لم تحسب سابقا.

مسافة DT(7,1) = المسافة الى (V6) .. (متمثلة في DT(6,1)) + الرابط بين (V6, V7) = 14

- سجل المسار, DT(7,2) = V6
- خزن V7:14 في طابور.

وضع الطابور سيكون **PQ: V7:14, V5:15**

جدول 10.5: جدول المسافات بعد معالجة مجاورات العقدة V6.

العقد	جدول المسافات	
V1	6	V3
V2	7	V4

	0	V3
V3	3	V4
V2	15	V5
V4	10	V6
V6	14	V7

وضع الطابور **PQ: V7:14, V5:15**

F. يتم سحب وازالة عقدة من بداية الطابور وستكون العقدة (**V7:14**), مجاورات هذه العقدة هي (V4{5}). لكل واحد من مجاورات العقدة V7 تجرى الاعمال التالية:

I. أقرا العقدة المجاورة **V4{5}**

- المسافة الى العقدة V4 تم حسابها سابقا وهي تساوي 3, وهي اصغر من المسافة الى العقدة V7 + وزن الرابط بين العقدتين (V7, V4 = 14+5 = 19), لذلك لا تعمل اي شيء.
- الجدول سوف لا يتغير

وضع الطابور سيكون PQ: **V5:15**

G. يتم سحب وازالة عقدة من بداية الطابور وستكون العقدة (**V5:15**), مجاورات هذه العقدة هي (V4{1}, V7{2}). لكل واحد من مجاورات العقدة V5 قم بالاعمال التالية:

I. أقرأ العقدة المجاورة الاولى **V4{1}**

- المسافة الى العقدة V4 تم حسابها سابقا وهي تساوي 3, وهي اصغر من المسافة الى V5 + وزن الرابط بين العقدتين (V5, V4 = 15+1=16), لذلك لا تعمل اي شيء.

II. أقرأ العقدة المجاورة الثانية **{2}V7**

- المسافة الى العقدة V7 تم حسابها سابقا وهي تساوي 14, وهي اصغر من المسافة الى V5 + وزن الرابط بين العقدتين (V5, V7 = 15+2 = 17), لذلك لا تعمل اي شيء.

3. الان الطابور فارغ, وهذه دلالة على انتهاء عمليات المعالجة.

ان اقصر المسالك بين العقدة V3 وباقي العقد في المخطط هي في الجدول 10.6.

جدول 10.6: جدول المسافات بين العقدة V3 وكل عقدة اخرى في المخطط.

العقد	جدول المسافات		المسلك الكامل
V1	6	V3	V3 V1
V2	7	V4	V3 V4 V2
V3	0		

V3 V4	V3	3	V4
V3 V4 V2 V5	V2	15	V5
V3 V4 V6	V4	10	V6
V3 V4 V6 V7	V6	14	V7

❖ برنامج خوارزمية Dijkstra's

```
/* Dijkstra's Algorithm in C */
#include<stdio>
#include<conio>
#include<process>
#include<string>
#include<math>
#define IN 99
#define N 6
using namespace std ;
int dijkstra(int cost[][N], int source, int target);
int main()
{
   int cost[N][N],i,j,w,ch,co;
   int source, target,x,y;
   for(i=1;i< N;i++)
```

```
    for(j=1;j< N;j++)
    cost[i][j] = IN;
    for(x=1; x< N; x++)
    {
        for(y=x+1; y< N; y++)
        {
        cout<< "Enter the weight of the path between nodes "<< x<< y ;
            cin>> w ;
            cost [x][y] = cost[y][x] = w;
        }
        cout << endl ;
    }
    cout<< "\nEnter the source:" ;
    cin>> source ;
    cout<< "\nEnter the target" ;
    cin>> target ;
    co = dijsktra(cost,source,target);
    cout<< "\nThe Shortest Path: " << co ;
}
int dijsktra(int cost[][N],int source,int target)
{
    int dist[N],prev[N],selected[N]={0},i,m,min,start,d,j;
```

```
char path[N];
for(i=1;i< N;i++)
{
   dist[i] = IN;
   prev[i] = -1;
}
start = source;
selected[start]=1;
dist[start] = 0;
while(selected[target] ==0)
{
   min = IN;
   m = 0;
   for(i=1;i< N;i++)
   {
      d = dist[start] +cost[start][i];
      if(d< dist[i]&&selected[i]==0)
      {
         dist[i] = d;
         prev[i] = start;
      }
      if(min>dist[i] && selected[i]==0)
      {
```

```
                min = dist[i];
                m = i;
            }
        }
        start = m;
        selected[start] = 1;
    }
    start = target;
    j = 0;
    while(start != -1)
    {
        path[j++] = start+65;
        start = prev[start];
    }
    path[j]='\0';
    strrev(path);
    cout<<  path ;
    return dist[target];
}
```

REFRENCES المصادر

1. Adam Drozdek, "**Data Structures and Algorithms in C++**", 4th edition, Published by Cengage Learning, USA, 2013.

2. Clifford A. Shaffer, "**A Practical Introduction to Data Structures and Algorithm Analysis**", 3rd edition, Published by Dover Publications, 2012.

3. D.S. Malik, "**Data Structures Using C++**", 2nd Edition, Published by Cengage Learning, 2010.

4. Harry H. Chaudhary, "**Data Structures Using C Language 2014. Perfect Beginners guide**", Createspace O-D Publishing LLC-USA, 2014.

5. James Robergé, Stefan Brandle, David Whittington, "**A Laboratory Course in C++ Data Structures**", 2nd edition, Jones and Bartlett Publishers, 2003.

6. Jörg Arndt, "Matters Computational Ideas, Algorithms, Source Code", Springer-Verlag Berlin Heidelberg , 2011, DOI**:** 10.1007/978-3-642-14764-7**.**

7. Mark Allen Weiss, "**Data Structures and Algorithm Analysis in C++**", 2nd edition, Pearson Education International, 2014.

8. Michael T. Goodrich, Roberto Tamassia, David M. Mount, "**Data Structures and Algorithms in C++",** 2nd Edition, John Wiley & Sons, 2011.

9. Neel Dale, "**C++ Plus Data Structure**", 4th edition, Jones and Bartlett Publishers, 2007.

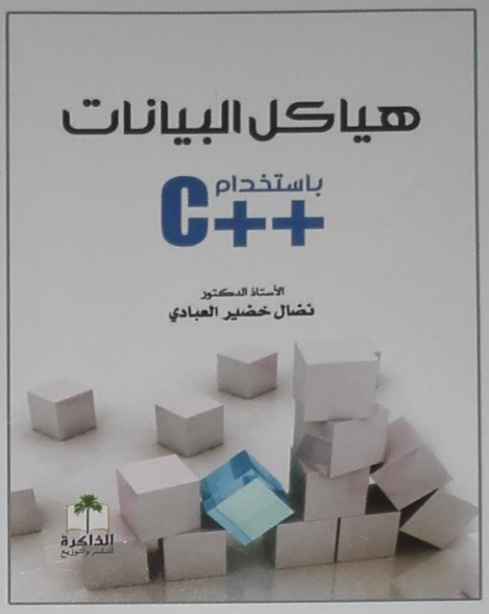
10. Radhika Raju Palagiri, Ananda Rao Akepogu, "**Data Structures and Algorithms Using C++**", Pearson India, 2010.

11. Robert Lafore, "**Data Structures & Algorithms in Java**", 2nd edition, Sams Publishing, 2003.

12. Robert Sedgewick and Kevin Wayne, "**Algorithms**", 4th edition, published by Addison-Wesley, 2011.

13. Vinu Das, "**Principles of Data Structures Using C and C++**", Published by New Age, 2006.

14. نضال العبادي, **"C++ من البداية الى البرمجة الكيانية"**, مؤسسة دار الصادق الثقافية بالاشتراك مع دار الصفاء, 2011.



9 789922 600062

الذاكرة
للنشر والتوزيع