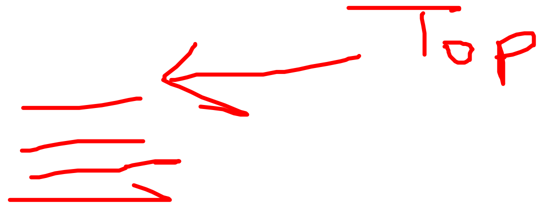
Stack Data Structure

Data Structure, second stage, Computer department, College of science for Women, University of Baghdad.

Dr. Amer Almahdawi.

What is a Stack?

- A stack is a simple data structure used for storing data.
- In a stack, the order in which the data arrives is important.
- A pile of plates in a cafeteria is a good example of a stack.
- The plates are added to the stack as they are cleaned, and they are placed on the top.
- When a plate, is required it is taken from the top of the stack.
- The first plate placed on the stack is the last one to be used.



Definition:

- A stack is an ordered list in which insertion and deletion are done at one end, called top.
- The last element inserted is the first one to be deleted.
- Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.
- Special names are given to the two changes that can be made to a stack.
- When an element is inserted in a stack, the concept is called push, and when an element is removed from the stack, the concept is called pop.

How Stacks are used

- Trying to pop out an empty stack is called underflow and trying to push an element in a full stack is called overflow.
- Generally, we treat them as exceptions.
- Consider a working day in the office. Let us assume a developer is working on a long-term project.
- The manager then gives the developer a new task which is more important.
- The developer puts the long-term project aside and begins work on the new task.

How Stacks are used

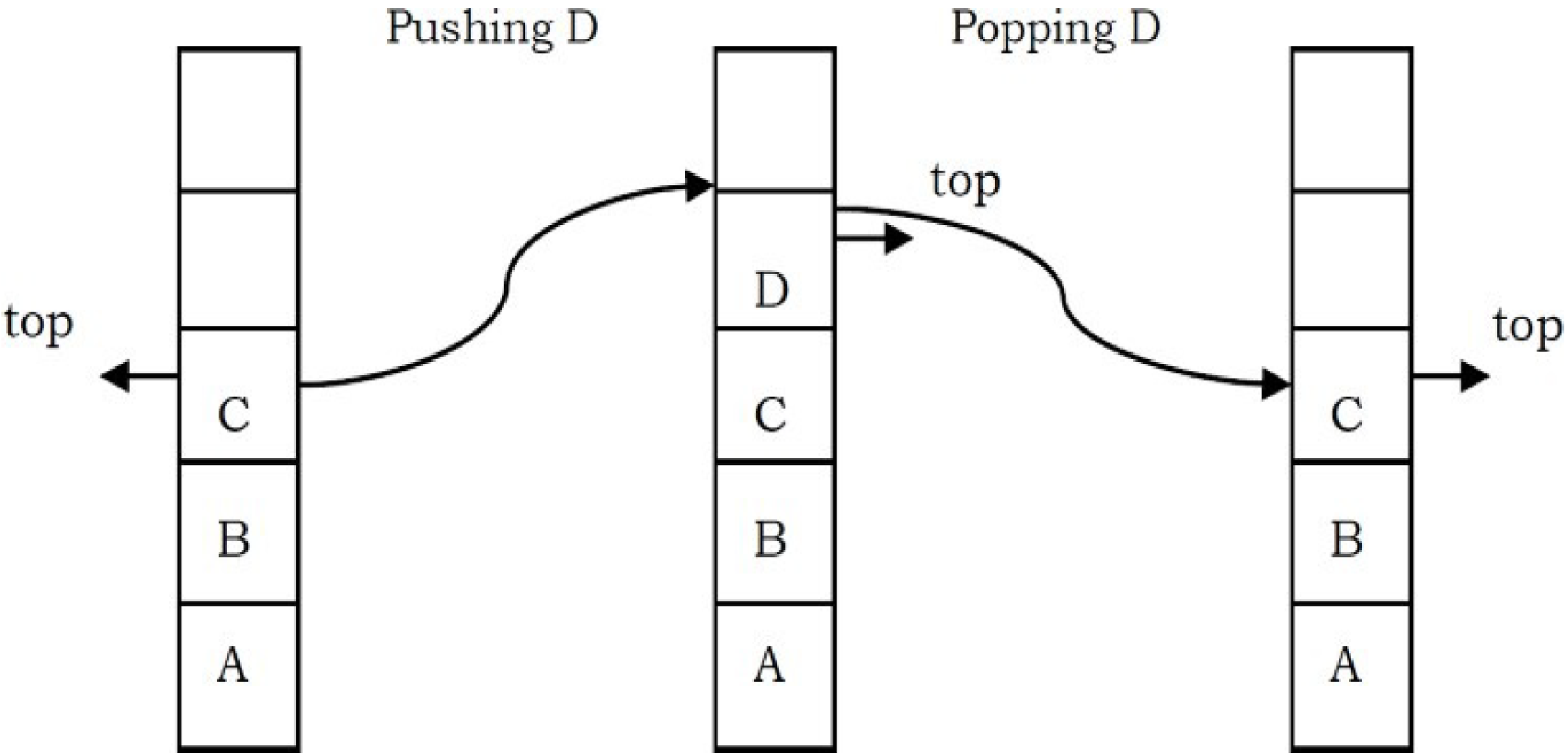
- The phone rings, and this is the highest priority as it must be answered immediately.
- The developer pushes the present task into the pending tray and answers the phone.
- When the call is complete the task that was abandoned to answer the phone is retrieved from the pending tray and work progresses.
- To take another call, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

Stack ADT (Abstracted Data Type)

- The following operations make a stack an ADT. For simplicity, assume the data is an integer type.
- **Main stack operations**
- Push (int data): Inserts data onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in the stack.
- int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

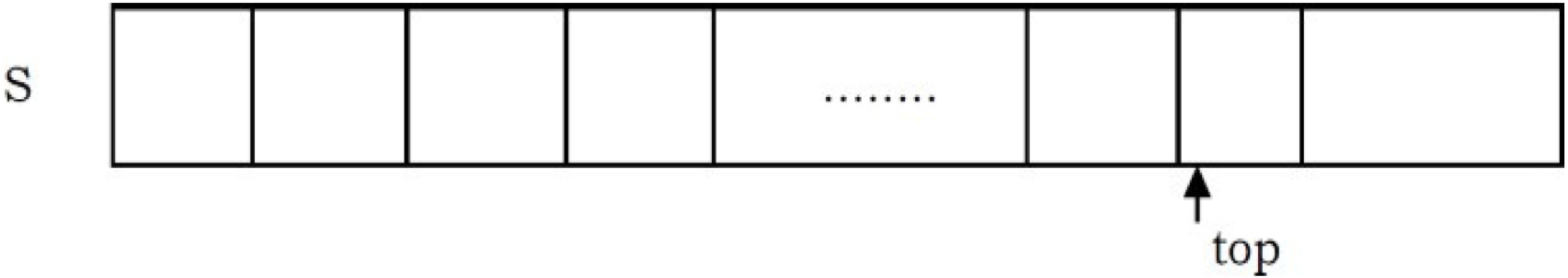

Pushing D
Popping D
top
C
B
A
top
D
C
B
A
top
C
B
A

Exceptions

- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be "thrown" by an operation that cannot be executed.
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty.
- Attempting the execution of pop (top) on an empty stack throws an exception.
- Trying to push an element in a full stack throws an exception.

Implementation

- There are many ways of implementing stack ADT; below are the commonly used methods.
- Simple array-based implementation.
- Dynamic array-based implementation.
- Linked lists implementation.



Simple Array Implementation

- This implementation of stack ADT uses an array.
- In the array, we add elements from left to right and use a variable to keep track of the index of the top element.

Simple Array Implementation

- The array storing the stack elements may become full.
- A push operation will then throw a full stack exception.
- Similarly, if we try deleting an element from an empty stack it will throw stack empty exception.

```
• class Stack(object):
•         def __init__ (self, limit = 10):
•                 self.stk = []
•                 self.limit=limit
•         def isEmpty(self):
•                 return len(self.stk)<=0
•         def push(self.item):
•                 if len(self.stk) >= self.limit:
•                         print ('Stack Overflow!’)
•                 else:
•                         self.stk.append(item)
•                 print ('Stack after Push',self.stk)
```

- def pop(self):
-         if len(self.stk) <= 0:
-                 print ('Stack Underflow!’)
-                 return 0
-         else:
-                 return self.stk.pop()
- def peek(self):
-          if len(self.stk) < 0:
-                 print ('Stack Underflow!’)
-                 return 0
-          else:
-                 return self.stk[-1]

- def size(self):
-         return len(self.stk)
- our_stack= Stack(5)
- our_stack.push(“1")
- our_stack.push(“21”)
- our_ stack.push(“14”)
- our_ stack.push(“31”)
- our_ stack.push(“19”)
- our_ stack.push(“3”)
- our_ stack.push(“99”)
- our_ stack.push("9")
- print (our_ stack.peek())
- print (our_ stack.pop())
- print (our_ stack.peek())
- print (our_ stack.pop())

Performance & Limitations

- Performance:
- Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of Push()	$O(1)$
Time Complexity of Pop()	$O(1)$
Time Complexity of Size()	$O(1)$
Time Complexity of IsEmptyStack()	$O(1)$
Time Complexity of IsFullStack()	$O(1)$
Time Complexity of DeleteStack()	$O(1)$

Limitations

- The maximum size of the stack must first be defined, and it cannot be changed.
- Trying to push a new clement into a full stack causes an implementation-specific exception.