University of Kelaniya
Sri Lanka

COSC 21063 / BECS 21223/COST 44233

Stacks

Lecturer : Ms.D.M.L.Maheshika Dissanayake

Structure of Lesson

- Stacks
  - Definition
  - Operations
  - Implementation

    - Contiguous implementation

    - Linked Implementation

Learning Outcome:

By the end of this lesson you should be able to :

- Understand the concept of stack
- Create the data structure stack and define the necessary operations
- Implement the stack using the contiguous and linked representation

Stacks

- Based on the everyday notion of a stack, such as a stack of books, or a stack of plates
- **Definition**

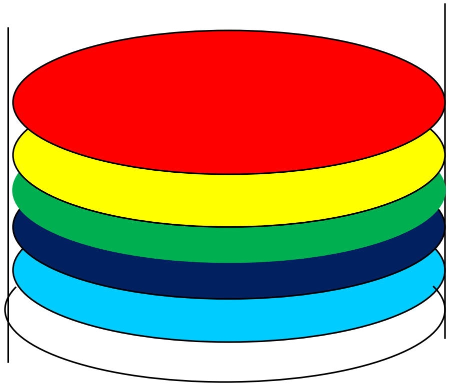
  A stack is a list of elements in which an element may be added (inserted) or removed (deleted) only at one end, called the top of the stack.
- Can only access the *top* element of the stack
- The active part of a stack is the top.

Example of a stack….

- Plates are added to the top of the plate holder
- Plates are removed from the top

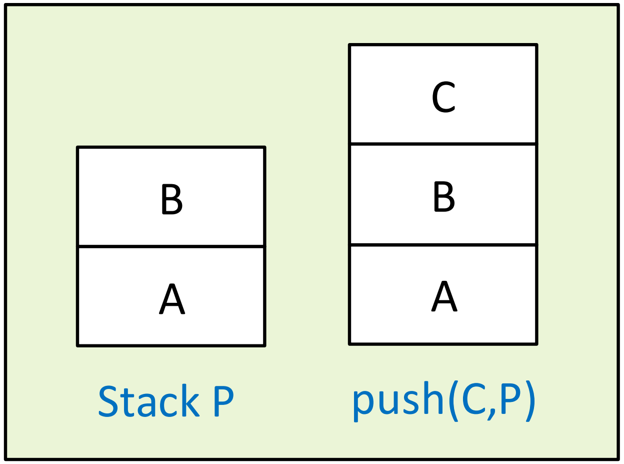
We cannot access the green plate without removing the plates above it first.

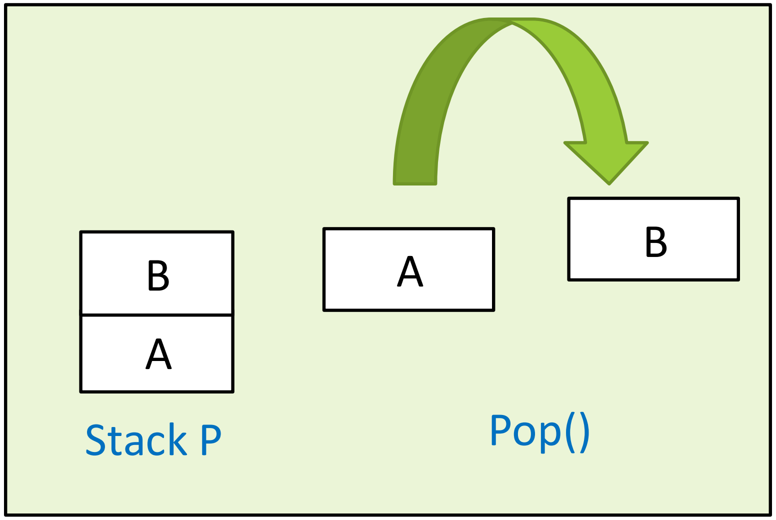
We cannot add a plate to the middle of the stack

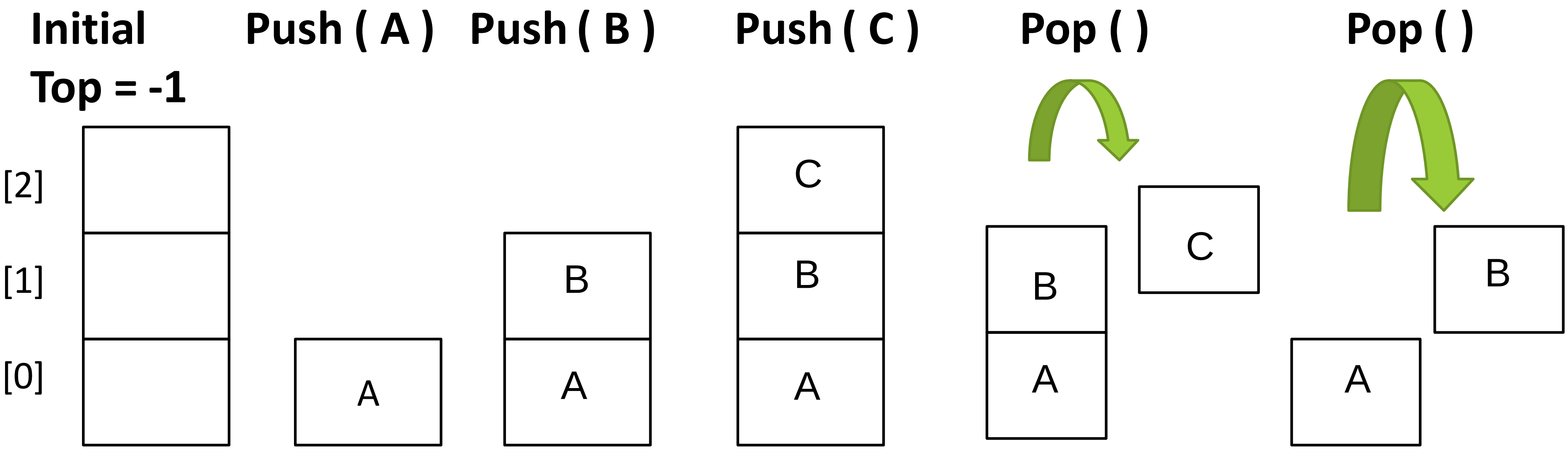


Stack Contd..

- When we add an item to a stack, we say we **push** it onto the stack
- when we remove an item, we say that we **pop** it from the stack.
- So the last item pushed onto the stack is always the first that will be popped from the stack. This property is called **last in, first out** (LIFO).






Initial
Top = -1
Push (A)
Push (B)
Push (C)
Pop ( )
Pop ( )
[2]
[1]
[0]
A
B
A
C
B
A
B
A
C
A
B

Exercise:

Add elements A B C D E into a stack

Stack Specification

- A stack is either *empty* or it consists of two parts:
  - a top element
  - a stack (the remaining elements).
- The elements in a stack may be of any type, but all the elements in a given stack must be the same type.

Pre & Post Conditions

Preconditions:

- These are properties about the inputs that are assumed by an operation.

Postconditions:

- Specify the effects of an operation. These are the *only* things you may assume have been done by the operation. They are only guaranteed to hold if the preconditions are satisfied.

Operations on the Stack

CreateStack

- Inputs: none
- Outputs: S (a stack)
- Preconditions: none
- Postconditions: S is defined (i.e. created) and empty (i.e. initialised to be empty)

Operations on the Stack Contd..

IsStackEmpty

- Inputs: S (a stack)
- Outputs: IsStackEmpty true or false
- Preconditions: The stack exists and it has been initialised
- Postconditions: IsStackEmpty is true iff S is empty.

Operations on the Stack Contd..

Top

- Inputs: S (a stack)
- Outputs: E (a stack element)
- Preconditions: S is not empty
- Postconditions: E is the top element on S (S is unchanged)

Operations on the Stack Contd..

Pop

- Inputs: S (a stack)
- Outputs: S (i.e. S is changed) and T the element removed
- Preconditions: S is not empty
- Postconditions: The top of the stack has been removed and returned in T

Exercise:

Write the specification for the following operations

- DestroyStack
- IsStackFull
- Push

Operations on the Stack Contd..

DestroyStack

- Inputs: S (a stack)
- Outputs: S’ (i.e. S changed)
- Preconditions: S exists and it has been initialised
- Postconditions: S’ is undefined. All resources (e.g. memory) allocated to S’ have been released. No stack operation can be performed on S’.

Note S’ is the updated S.

Operations on the Stack Contd..

IsStackFull

- Inputs: S (a stack)
- Outputs: IsStackFull true or false
- Preconditions: The stack exists and it has been initialised
- Postconditions: IsStackFull is true iff S is full

Operations on the Stack Contd..

Push

- Inputs: S (a stack) and V (a value)
- Outputs: S'(i.e. S changed)
- Preconditions: S is not full and V is of appropriate type for an element of S
- Postconditions: S’ has V as its top element and S as its remaining elements.

Note S’ is the updated S

- The definition of the values of type `stack' make no mention of an upper bound on the size of a stack.
- In practice, there is always an upper bound - the amount of computer storage available.
- it is an implicit precondition on all operations that there is storage available, as needed.
- The operations specified above are *core* operations - any other operation on stacks can be defined in terms of these ones

Implementation of stacks

- Contiguous implementation (Array Based)
- Linked Implementation

Contiguous implementation

- Define a stack in java
- Identify the information needed.
    - An array to hold the elements of the stack.
    - An integer variable top, the pointer of the top element in the array. This variable will also indicate the number of elements in the stack

Implement the operation: CreateStack

```
Class Stack{
    private int top;
    private int maxSize;
    private int[] stackArray;

    Stack(int size)
    {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }}
```

Implement the operation : IsStackEmpty

```
public  boolean IsStackEmpty()
   {
     return  top ==-1;
   }
```

Implement the operation : IsStackFull

```
public boolean IsStackFull() {
      return (top == maxSize - 1);
   }
```

Implement the operation :Push

```
public void push(int x) {
    if ( IsStackFull()) {
        System.out.println("Stack is full");
        System.exit(1);
    }
   else {
    System.out.println("Inserting " + x);
    stackArray[++top] = x;
    }
  }
```

Implement the operation :Pop

```
public int pop() {
    if (IsStackEmpty()) {
        System.out.println("STACK EMPTY");
        System.exit(1);
        }
        return stackArray[top--];
    }
```

An example

- Write a program for the following.

  Initialize a string. Using the operations of the stack reverse the Initialized string.

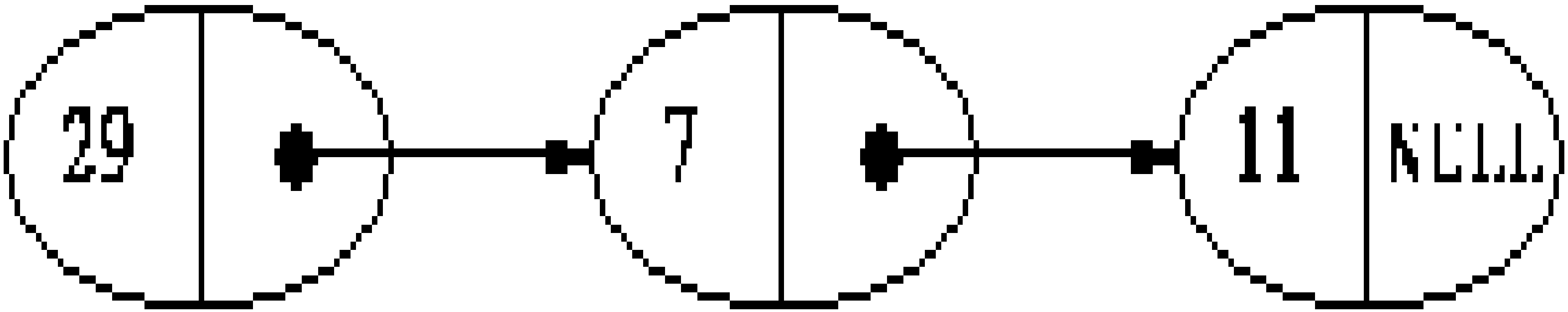
```java
public static void main (String[] args)
    {
        String str = "Reverse me";

        str = reverse(str);
        System.out.println(str);
    }
}
```

```java
public static String reverse(String str)
   {
        Stack  s1 = new Stack(30);

       // push each character in the string into the stack
       char[] chars = str.toCharArray();
       for (char c: chars) {
          s1.push(c);
       }

       // pop all characters  and  put them back to the character array
       for (int i = 0; i < str.length(); i++) {
          chars[i] =(char) s1.pop();
       }

       // convert the char array to a string and return
       return new String(chars);
   }
```

Linked Implementation

- Consider the stack of integers S = 29,7,11 (29 is the top, 11 the bottom).
- There will be one memory cell or node for each element in the stack; in this memory cell we need to store two things: the value of the element, which is an integer in this example but could be anything, and a pointer to the memory cell of the next element.

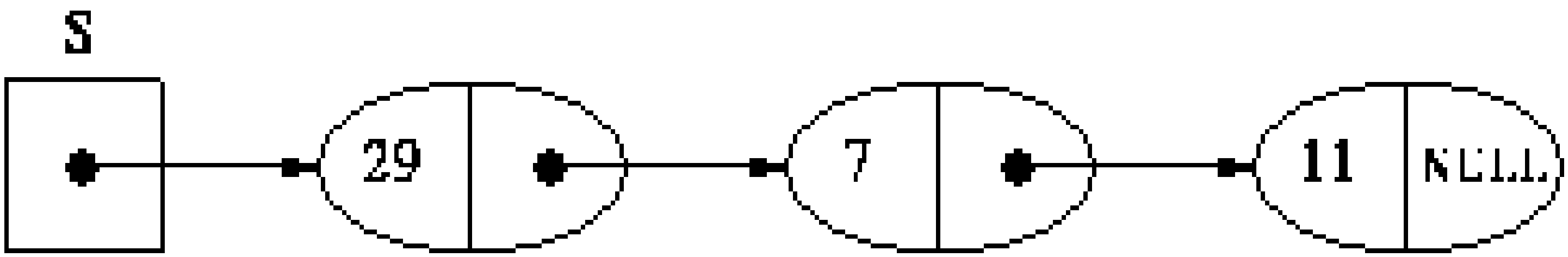
Linked Implementation Contd..



First element Second element Third element

Linked Implementation Contd..

Where is the stack ?



First element

Second element

Third element

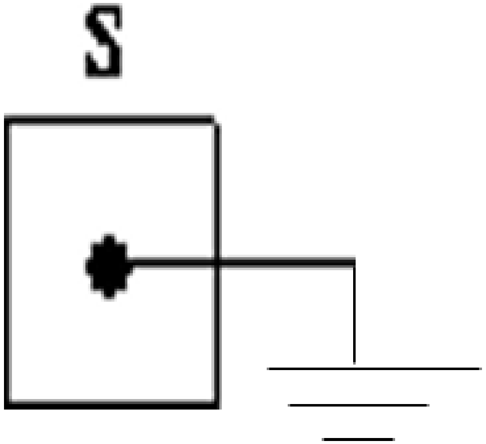
Linked Implementation Contd..

How will we represent the Empty stack?

Linked Implementation Contd..

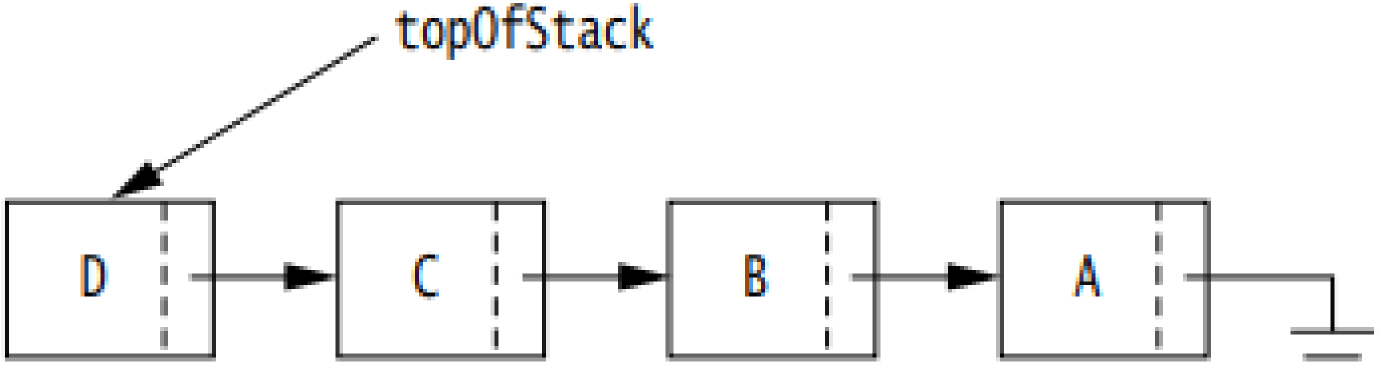
- In S we could store the *number* of elements that it contains, in which case the empty stack would be represented by setting this value to zero.
- Or we could have a boolean (logical) variable telling us whether or not S is empty.
- The simplest solution is to set S's Top pointer to NULL when S is empty (of course we are sure it will be non-NULL when S is non-empty).

Linked Implementation Contd..



Linked Implementation Contd..

- Establishing the structure of a cell or node.
- Identify the information needed for the node
  - The value of the element (in this example an integer, but could be anything)
  - A pointer to the next node



Implement the operation: CreateStack

```
class Node {
    int  entry ;
    Node next ;
}
```

Implement the operation: CreateStack

```
class Stack
{
   private Node top;
   private int no_ele;
       public Stack() {
           this.top = null;
           this.no_ele = 0;
       }
}
```

Implement the operation : IsStackEmpty

```
public  boolean IsStackEmpty()
   {
         return  top == null;
   }
```

Implement the operation : peek()

```
public int peek()
  {
     // check for an empty stack
     if (isStackEmpty()) {
        System.out.println("The stack is empty");
     }
     return top.entry;
  }
```

Implement the operation :Pop

```
public int pop() {
 if (top == null) {
        System.out.print("\n Stack is empty");
        return 0;
}
  else {
     int element = (top).entry;
     this. no_ele -= 1;
     this.top = (this.top).next;
     return element;
    }
}
```

Implement the operation :Push

```
public void push(int x) {
Node node = new Node();
  if (node == null)
     {  System.out.println("Heap Overflow");}
 else{
     System.out.println("Inserting " + x);
          node.entry = x;
          node.next = top;
          top = node;
          this. no_ele += 1; }
}
```

Summary of main points

- Definition of a stack
- Implementation of the stack operation