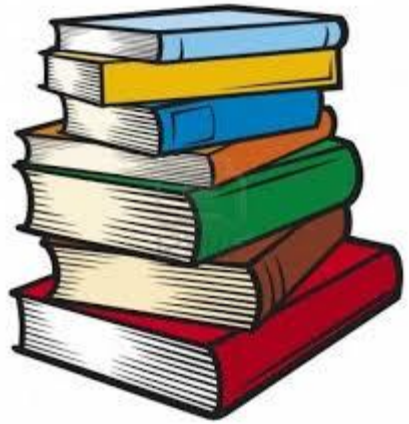
Examples of stack

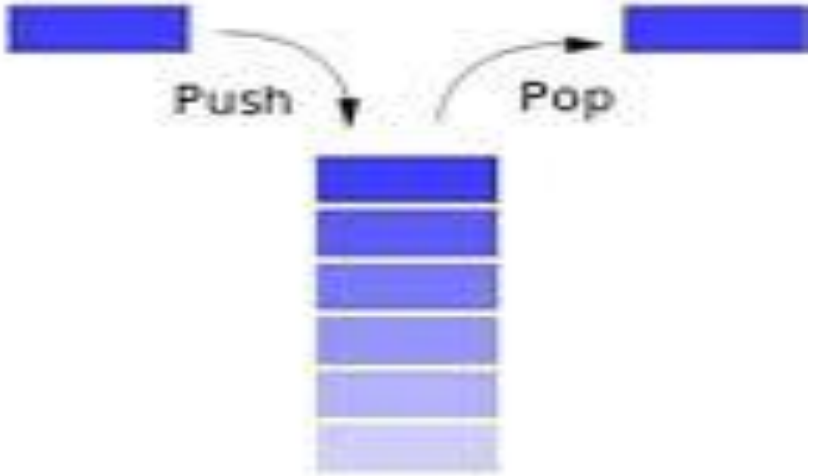
Stacks

Kiran Waghmare



Stack of books

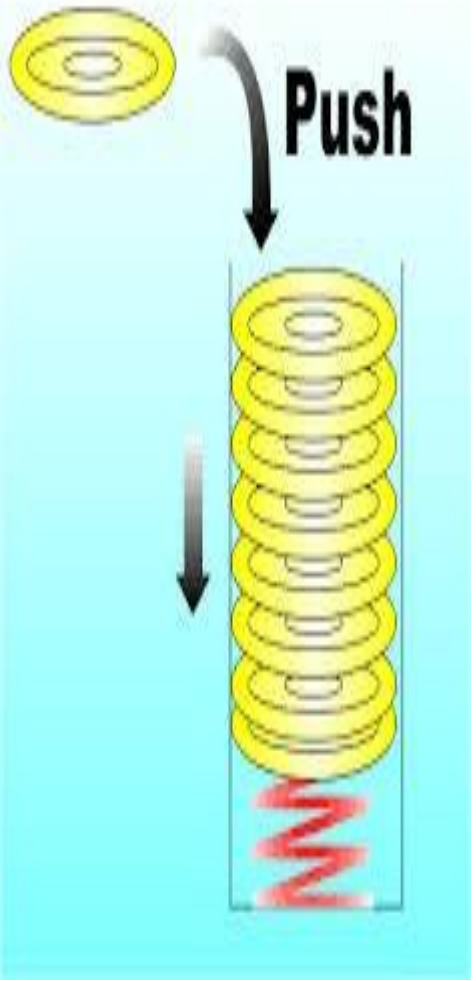
Stack of Coins



Memory stack

Stack

- Stack is an ordered list of similar data type.
- Stack is a LIFO structure. (Last in First out).
- push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

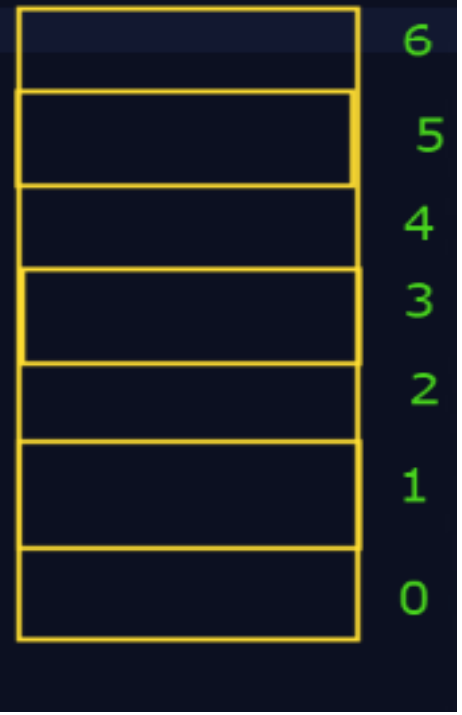


-Stack is a data structure, which is an ordered list following Last-In-First-Out

Operations on Stack:

1.Insertion : Push
2.Deletion : Pop
3.Traverse
4.Search
5.Sorting

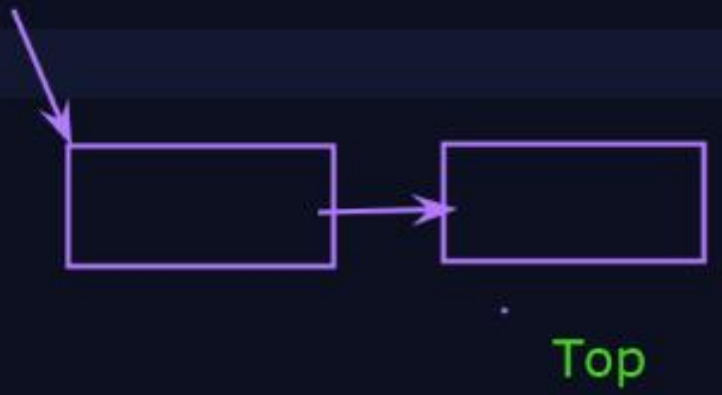
Stack:

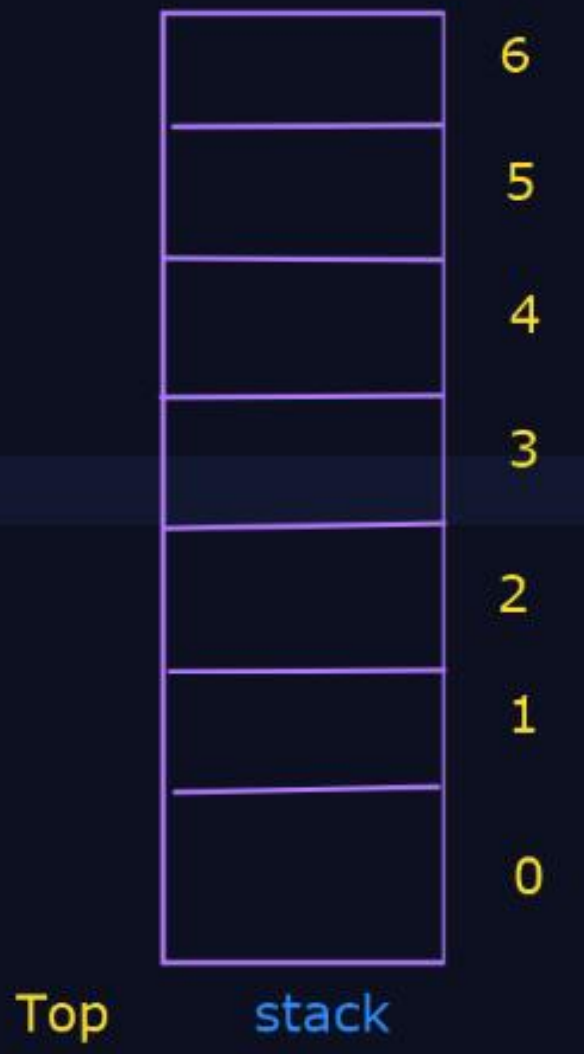


Representation of stack:

1. Fixed size stack: static : Arrays implementation
   - HAs a fixed size that cannot grow or shink.
   - Overflow occurs if stack is full.
   - Underflow occurs if stack is empty.

2. Dynamic size Stack: dynamic : Linked list Representation

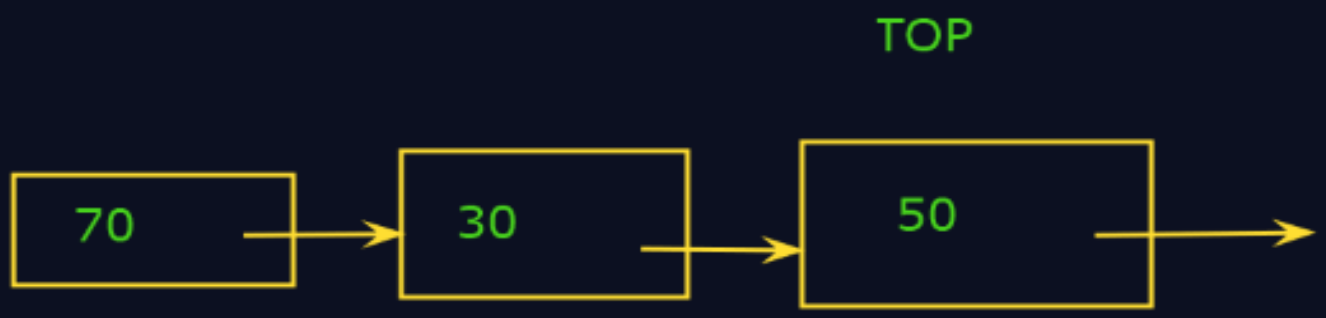




Stack: Functions used in stack:

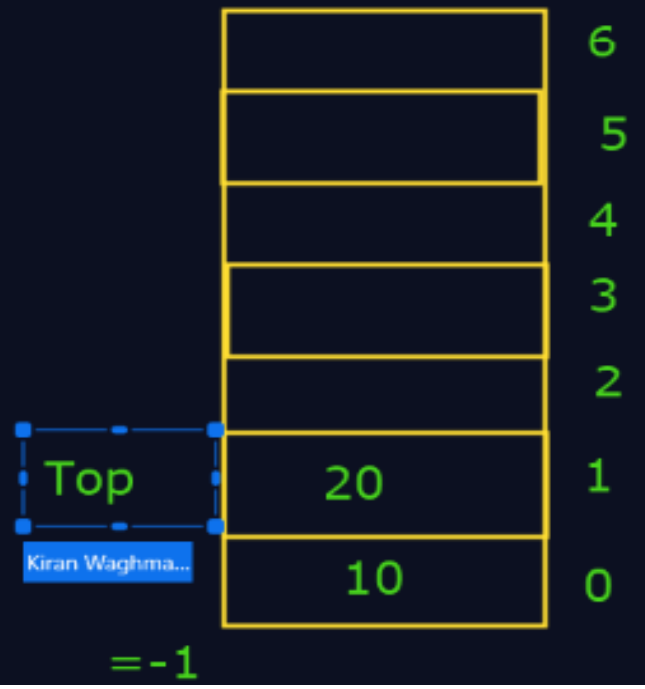
Push(): Insert an element
Pop() : Delete an element
isEmpty(): determines whether stack is empty
isFull() : determines whether stack is full
peek(): return the element at the top
count() : return the number of elemenents present in the stck
display(): print all the elements present in the stack

40
60
80



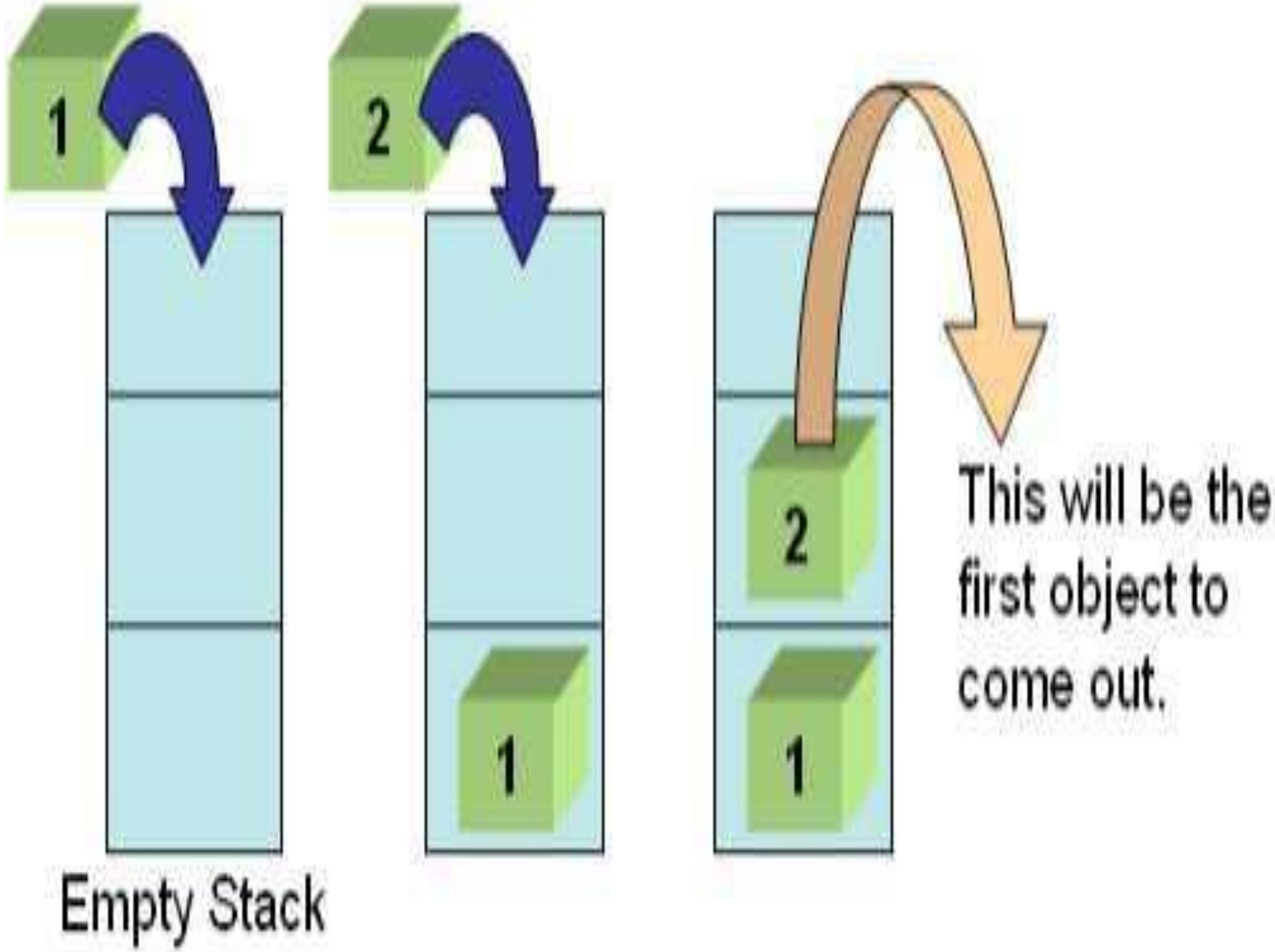
Linked List Representation of Stack

Array REpresentation of Stack



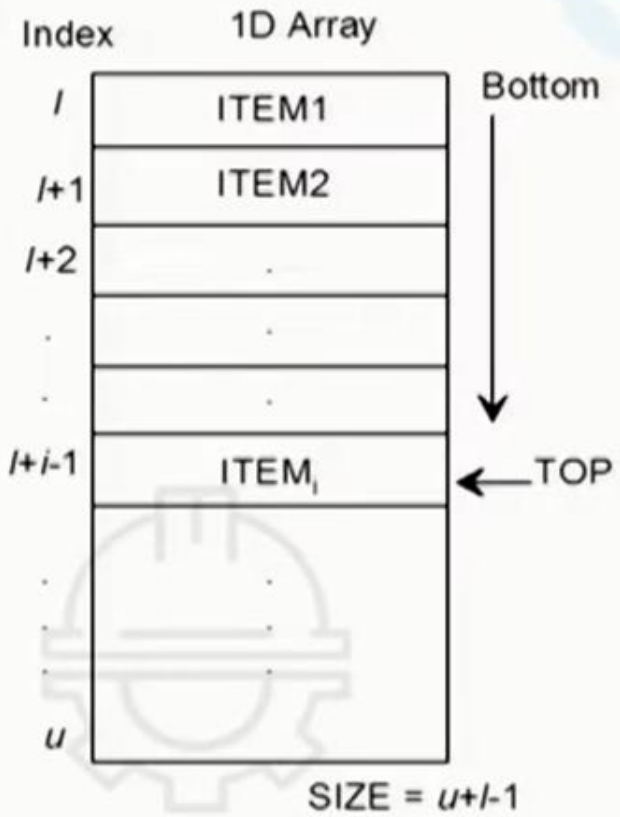
Standard Stack Operations

- **The following are some common operations implemented on the stack:**
- **push():**
  - When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():**
  - When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():**
  - It determines whether the stack is empty or not.
- **isFull():**
  - It determines whether the stack is full or not.'
- **peek():**
  - It returns the element at the given position.
- **count():**
  - It returns the total number of elements available in a stack.
- **change():**
  - It changes the element at the given position.
- **display():**
  - It prints all the elements available in the stack.

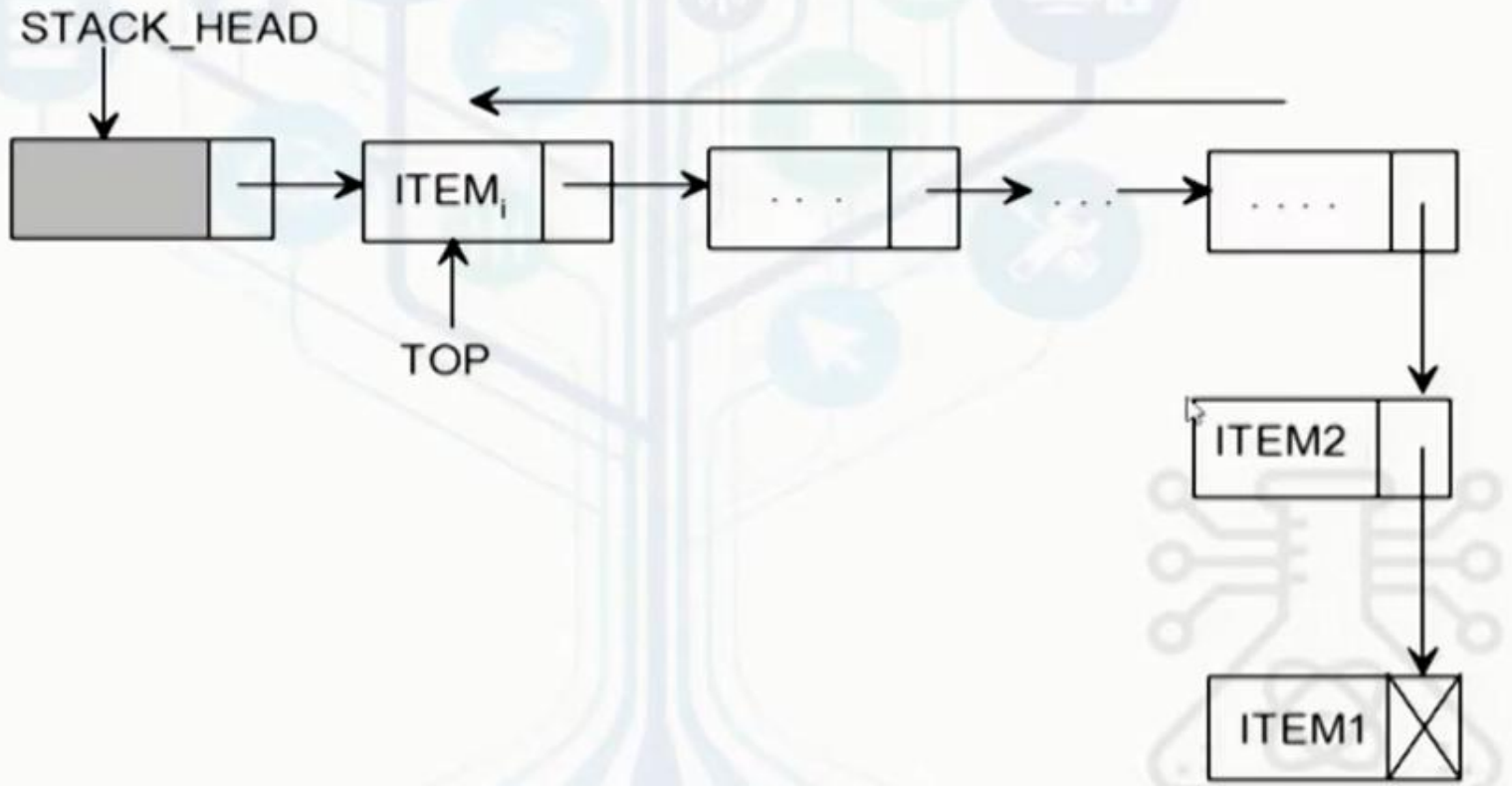

1
2
2
1
2
1
This will be the first object to come out.
Empty Stack

Memory representations

Array representation



Linked list representation



Operations on stack

Push To insert an item into the stack

Pop To remove an item from a stack

Status To know the present state of a stack

- if stack is empty,
- size of the stack,
- the element at top

Operation: Push with array

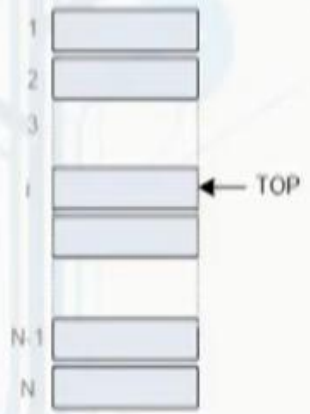
We have assumed that the array index varies from 1 to SIZE and TOP points the location of the current top-most item in the stack. The following algorithm defines the insertion of an item into a stack represented using an array A.

Input: The new item ITEM to be pushed onto it.
Output: A stack with a newly pushed ITEM at the TOP position.
Data structure: An array *A* with TOP as the pointer.

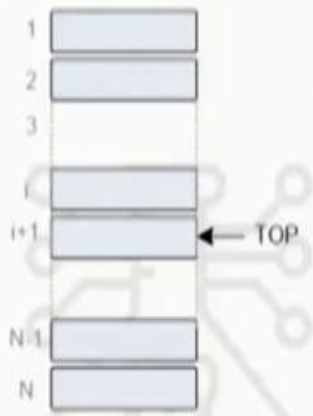
Steps:

1. **If** TOP ≥ SIZE **then**
2. + **Print** "Stack is full"
3. **Else**
4. TOP = TOP + 1
5. A[TOP] = ITEM
6. **EndIf**
7. **Stop**

Push operation is failed



Before push()



After push()

Insertion in Stack

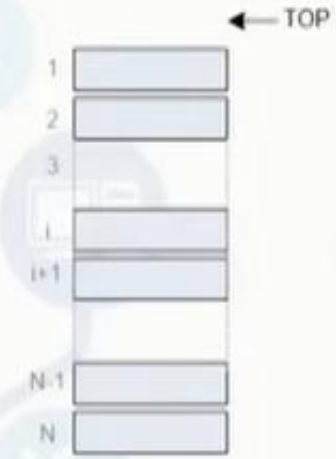
Operation: Pop with array

The following algorithm defines the deletion of an item from a stack represented using an array A.

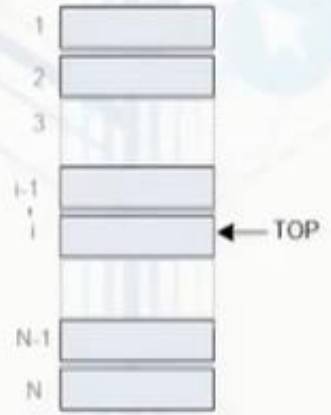
Input: A stack with elements.
Output: Removes an ITEM from the top of the stack if it is not empty.
Data structure: An array A with TOP as the pointer.

Steps:

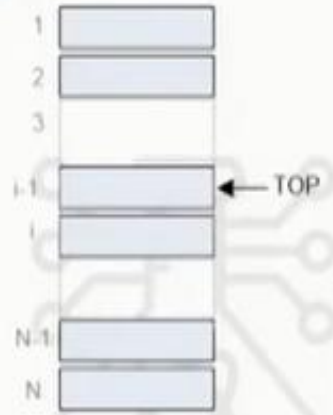
1. **If** TOP < 1 **then**
2. **Print** "Stack is empty"
3. **Else**
4. ITEM = A[TOP]
5. TOP = TOP – 1
6. **EndIf**
7. **Stop**



Pop operation is failed



Before Pop()



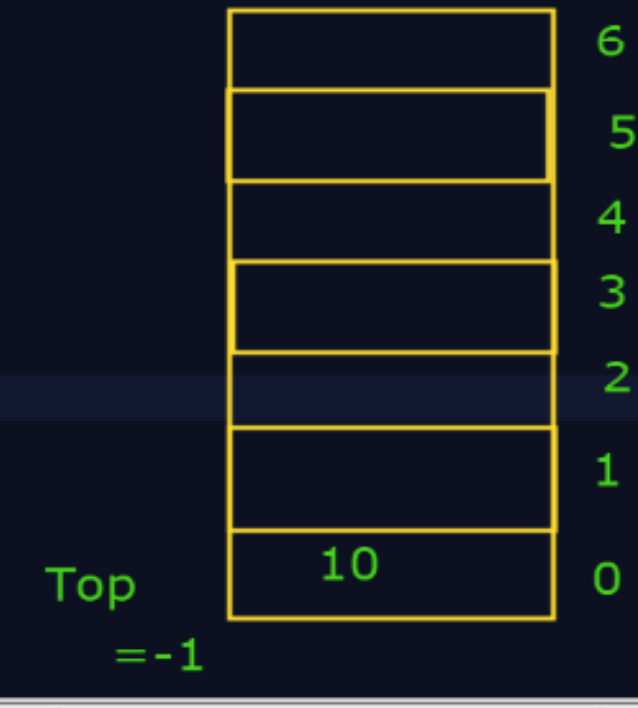
After Pop()

Deletion in Stack

```
class Stack{
    static final int MAX = 7;
    int top;
    int a[] = new int[MAX];

    Stack(){
        top = -1;
    }

    boolean isEmpty(){
        return top < 0;
    }

    boolean push(int x){
        if(top >= (MAX-1)){
            System.out.println("Stack Overflow!!!");
            return false;

        }
        else{
            a[++top] = x;
            System.out.println(x+"Element pushed!");
            return true;

        }

    }
```

40
60
20
80

Array REpresentation of Sta

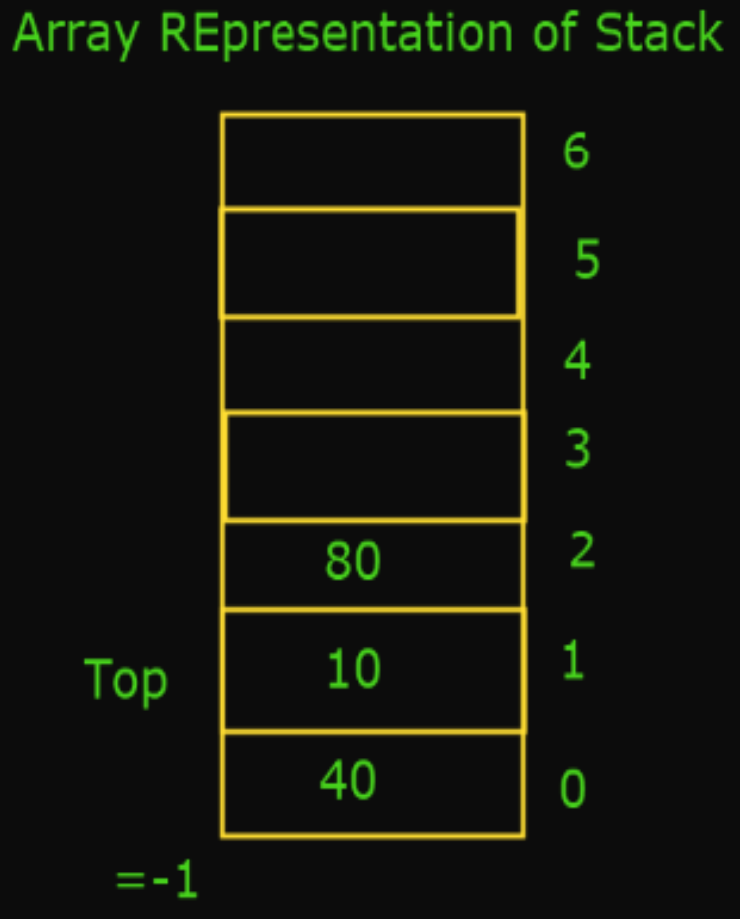


```
}

class StackDemo{
    public static void main(String[] arg

        Stack s = new Stack();
        s.push(40);
        s.push(10);
        s.push(80);
        System.out.println(s.peek());
        System.out.println(s.pop());
        System.out.println(s.peek());

    }
}
```

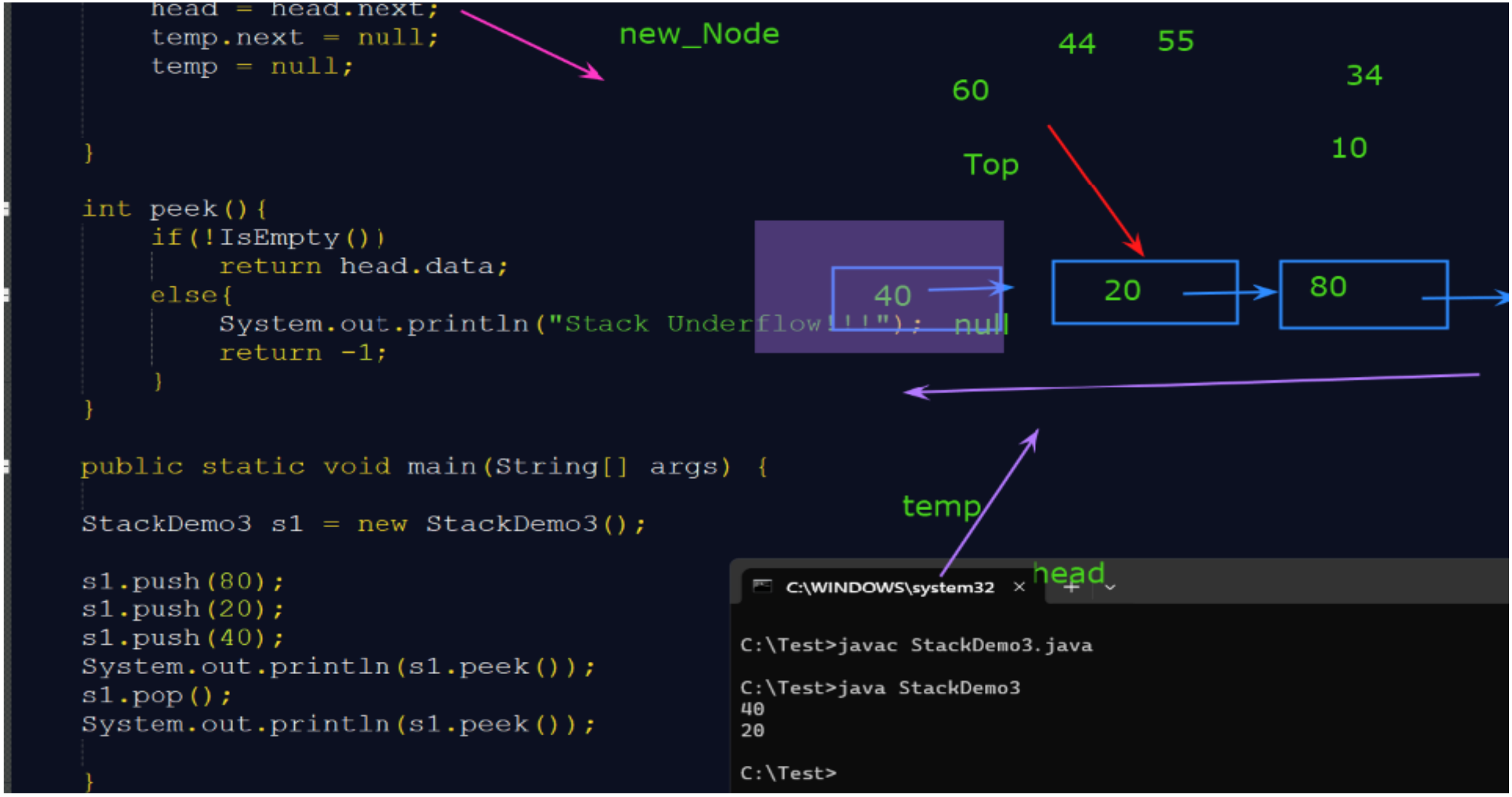
20

C:\WINDOWS\system32

```
C:\Test>javac StackDemo.java

C:\Test>java StackDemo
40Element pushed!
10Element pushed!
80Element pushed!
80
10
10

C:\Test>
```



```
        head = head.next;
        temp.next = null;
        temp = null;


    }

    int peek(){
        if(!IsEmpty())
            return head.data;
        else{
            System.out.println("Stack Underflow!!!");
            return -1;
        }
    }

    public static void main(String[] args) {

    StackDemo3 s1 = new StackDemo3();

    s1.push(80);
    s1.push(20);
    s1.push(40);
    System.out.println(s1.peek());
    s1.pop();
    System.out.println(s1.peek());

    }
```



```
isEmpty(): determines whether stack is empty
isFull() : determines whether stack is full
peek(): return the element at the top
count() : return the number of elemenents present in the stck
display(): print all the elements present in the stack

Stack Time complexity:
----------------------

using Array:
------------
Push()   : O(1)
Pop()    : O(1)
Peek()   : O(1)
isEmpty(): O(1)

using LinkedList:
-----------------
Push()   : O(1)
Pop()    : O(1)
Peek()   : O(1)
isEmpty(): O(1)
```

Worst case Time complexity

n* Push()=>O(n)

Applications of Stack

- **The following are the applications of the stack:**

- **Balancing of symbols:**
  - Stack is used for balancing a symbol. For example, we have the following program:
  - As we know, each program has an opening and closing braces;
  - when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack.
  - Therefore, the net value comes out to be zero.
  - If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:**
  - Stack is also used for reversing a string.
  - For example, we want to reverse a "javaTpoint" string, so we can achieve this with the help of a stack.
  - First, we push all the characters of the string in a stack until we reach the null character.
  - After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

- **UNDO/REDO:**
  - It can also be used for performing UNDO/REDO operations.
  - For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc.
  - So, there are three states, a, ab, and abc, which are stored in a stack.
  - There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  - If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:**
  - The recursion means that the function is calling itself again.
  - To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):**
  - This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:**
  - Suppose we have to create a path to solve a maze problem.
  - If we are moving in a particular path, and we realize that we come on the wrong way.
  - In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

- **Expression conversion:**
  - Stack can also be used for expression conversion.
  - This is one of the most important applications of stack.
  - The list of the expression conversion is given below:
    - Infix to prefix
    - Infix to postfix
    - Prefix to infix
    - Prefix to postfix
    - Postfix to infix
- **Memory management:**
  - The stack manages the memory.
  - The memory is assigned in the contiguous memory blocks.
  - The memory is known as stack memory as all the variables are assigned in a function call stack memory.
  - The memory size assigned to the program is known to the compiler.

  When the function is created, all its variables are assigned in the stack memory.

  When the function completed its execution, all the variables assigned in the stack are released.

Stack Application:

1. String Reversal
2. Balancing of parathesis
3. Recursion
4. Backtracking
5. DFS
6. Polish notation/Expression conversion

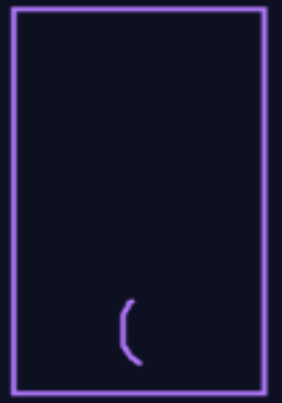
Worst case Time complexity

n* Push()=>O(n)

()
[]
{}

(()) : Balanced paraenthesis

((()): Not Balanced Parenthesis



(({}[]))

'{' || '[' || '(' : Push()

'}' || ']' || ')' : Pop ()

Parenthesis matching problem : Algorithm

1. For each character of input string
2. If character is opening parenthesis '(', put it on stack.
3. If character is closing parenthesis ')'
   a. Check top of stack, if it is '(' , pop and move to next character.
   b. If it is not '(', return false
4. After scanning the entire string, check if stack is empty. If stack is empty, return true else return false.

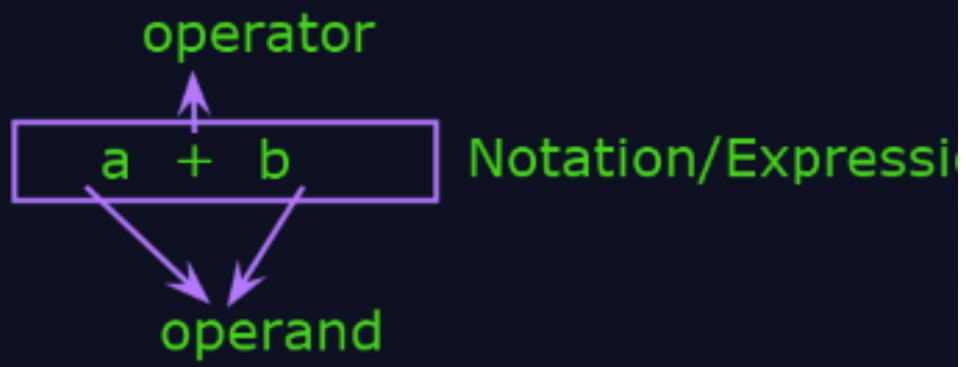
Polish Notations

1. **Infix Notation : A+B**
2. **Prefix Notation: +AB**
3. **Postfix Notation : AB+**

- **Operator Precedence:**
  1. **BODMAS Rule**
  2. **Brackets, Exponential, (* / %), (+ -)**
- **Rules: Infix to Postfix Conversion**
  1. Parenthesize the expression starting from left to right.
  2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in above expression B* C is parenthesized first before A+B.
  3. The sub-expression (part of expression) which has been converted into postfix is to be treated as single operand.
  4. Once the expression is converted to postfix from remove the parenthesis.

Stack Application:

1. String Reversal
2. Balancing of parathesis
3. Recursion
4. Backtracking
5. DFS
6. Polish notation/Expression conversion

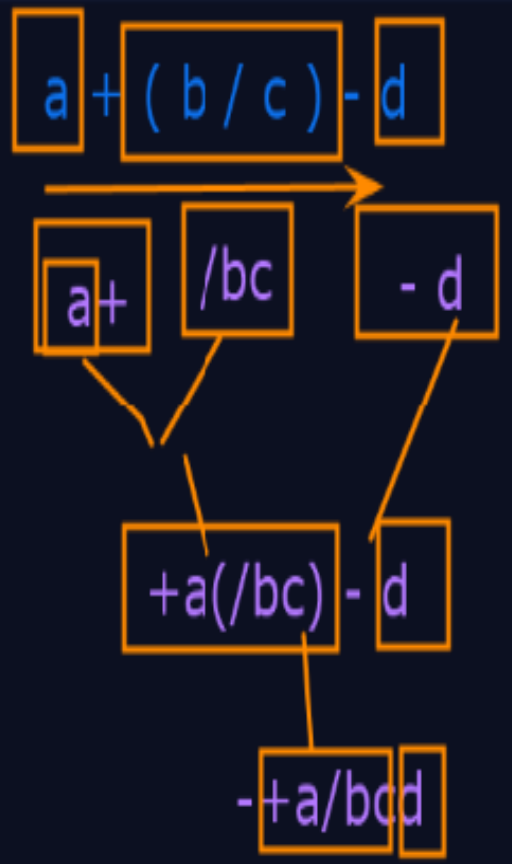


1. Infix notation/Expression
   A+B

2. Prefix notation/Expression
   +AB

3. Postfix notation/Expression
   AB+

Conversion of Postfix: a+(b*c)



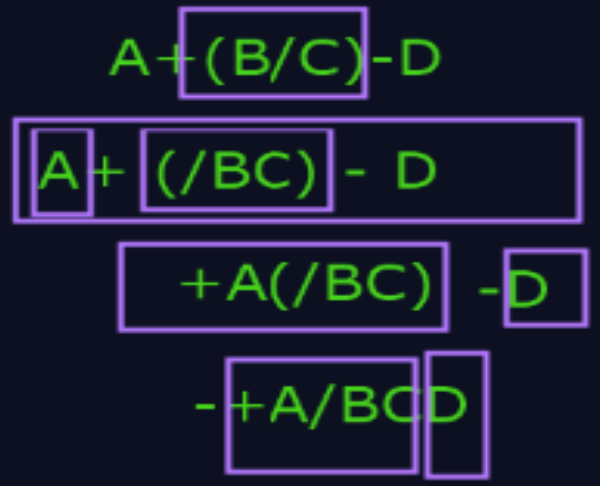
1. Infix notation/Expression
   A+B

2. Prefix notation/Expression
   +AB

3. Postfix notation/Expression
   AB+

Precedence:

Brackets
Uninary operator
Exponential (^)
*/
+-



ABC/+D-

A + B

+AB: prefix

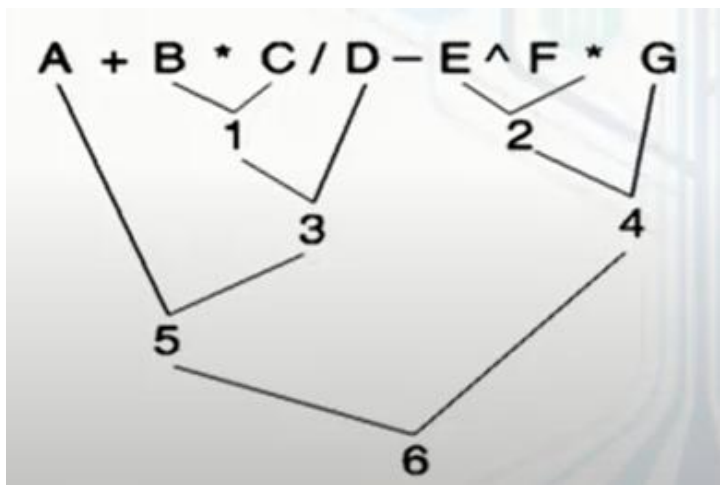
AB+:postfix

(A+B)*C/D+E^F/G

A + B * C / D – E ^ F * G

Precedence and associativity of operators

Operators	*Precedence*	*Associativity*
– (unary), +(unary), NOT	6	–
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), – (subtraction)	4	Left to right
<, <=, +, < >, >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right



ALGORITHM:

- **Scan infix expression from left to right.**
- **If there is a character as operand, output it.**
- **if not**

  1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.

  2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
- **If the scanned character is an '(', push it to the stack.**
- **If the character character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.**
- **Repeat steps 2-6 until infix expression is scanned.**
- **display the output**
- **Pop and output from the stack until it is not empty.**

ADVANTAGE OF POSTFIX:

- Any formula can be expressed without parenthesis.
- It is very convenient for evaluating formulas on computer with stacks.
- Postfix expression doesn’t has the operator precedence.
- Postfix is slightly easier to evaluate.
- It reflects the order in which operations are performed.
- You need to worry about the left and right associativity.

Application: Evaluation of a postfix expression

Steps:

1. Append a special delimiter '#' at the end of the expression
2. item = *E*.**ReadSymbol**( ) // Read the first symbol from *E*
3. **While** (item ≠ '#') **do**
4. **If** (item = operand) **then**
5. **PUSH**(item) // Operand is the first push into the stack
6. **Else**
7. op = item // The item is an operator
8. *y* = **POP**( ) // The right-most operand of the current operator
9. *x* = **POP**( ) // The left-most operand of the current operator
10. *t* = *x* op *y* // Perform the operation with operator 'op' and operands *x*, *y*
11. **PUSH**(*t*) // Push the result into stack
12. **EndIf**
13. item = *E*.**ReadSymbol**( ) // Read the next item from *E*
14. **EndWhile**
15. value = **POP**( ) // Get the value of the expression
16. **Return**(value)
17. **Stop**

Conversion of Infix to Postfix: a+(b*c)

Read character	Stack	Output
a	-	a
+	+	a
(	+(	a
b	+(	ab
*	+(*	ab
c	+(*	abc
)	+	abc*
-	-	abc*+

Algorithm:

- Declare a character stack S.
- Now traverse the expression string exp.
  - If the current character is a starting bracket (‘**(**‘ **or** ‘**{**‘ **or** ‘**[**‘) then push it to stack.
  - If the current character is a closing bracket (‘**)**’ **or** ‘**}**’ **or** ‘**]**’) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
- After complete traversal, if there is some starting bracket left in stack then “not balanced”