Data Structures

IT 205

Dr. Manish Khare

Lecture – 8,9
18-Jan-2018

Syllabus for First-In-Sem Examination

Topics covered till lecture of 30-Jan-2018

will be syllabus for First In-Sem Exam.

Stacks

Stack

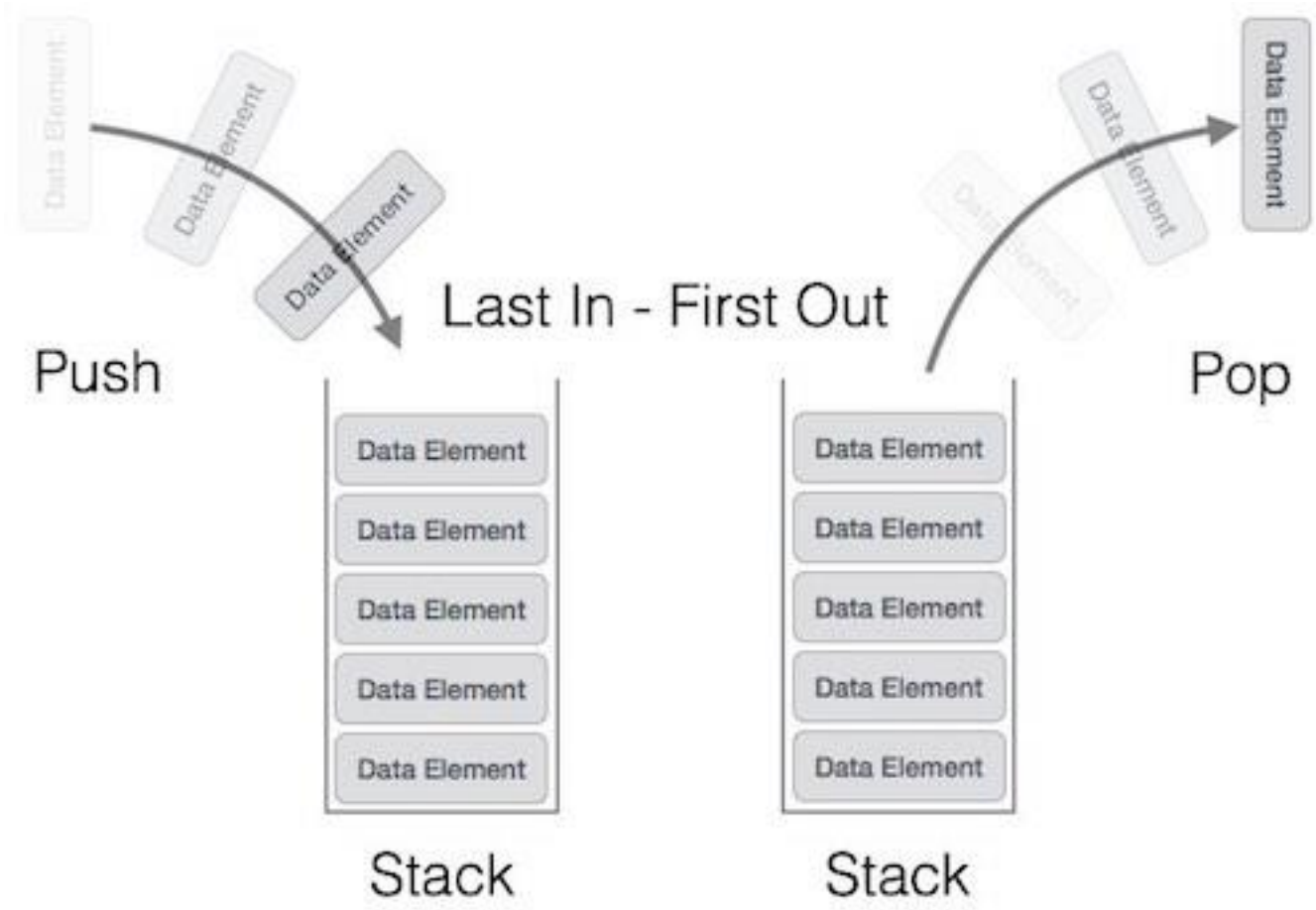
- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

- A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

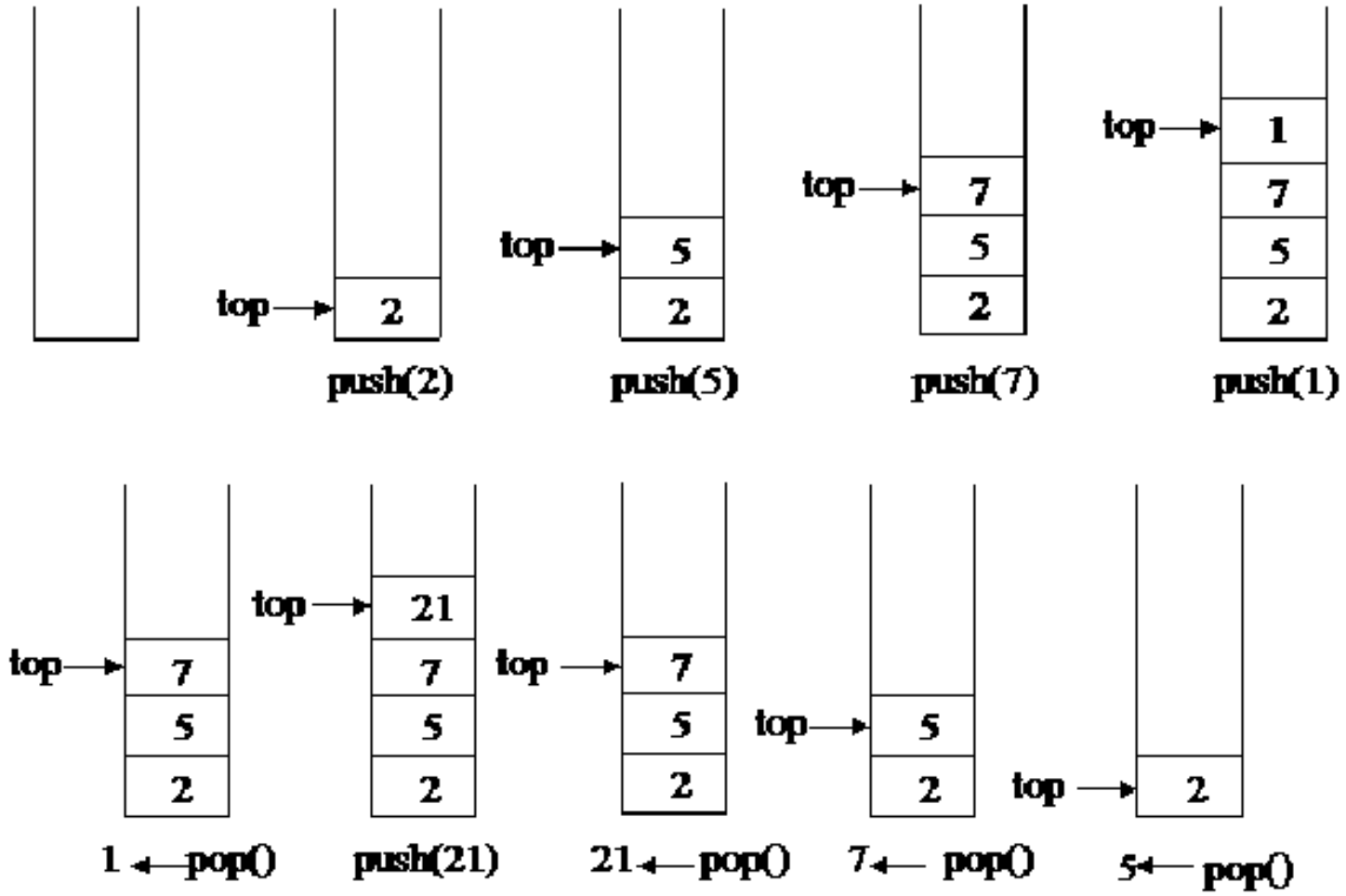
- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

- The following diagram depicts a stack and its operations −



Stack Representation



Basic Operations on Stack

- Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –
  - **push()** − Pushing (storing) an element on the stack.
  - **pop()** − Removing (accessing) an element from the stack.

- To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −
  - **peek()** − get the top data element of the stack, without removing it.
  - **isFull()** − check if stack is full.
  - **isEmpty()** − check if stack is empty.
- At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Stack Implementation

- Stack data structure can be implement in two ways. They are as follows...
  - **Using Array**
  - **Using Linked List**

- When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Stack Implementation using Array

Stack Operations using Array

- A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

- **Step 2:** Declare all the **functions** used in stack implementation.
- **Step 3:** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4:** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
- **Step 5:** In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

peek() operation

- Algorithm of peek() function –

```
begin procedure peek
   return stack[top]
end procedure
```

- Implementation of peek() function in C/C++ programming language

```
int peek() {
      return stack[top];
    }
```

isfull() operation

- Algorithm of isfull() function −

```
begin procedure isfull
    if top equals to MAXSIZE
        return true
    else
        return false
    endif
end procedure
```

isfull() operation

- Implementation of isfull() function in C/C++ programming language

```
bool isfull() {
    if(top == MAXSIZE)
        return true;
    else
        return false;
}
```

isempty() operation

- Algorithm of isempty() function −

```
begin procedure isempty()
   if top less than 1
      return true
   else
      return false
   endif
end procedure
```

isempty() operation

- Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty.

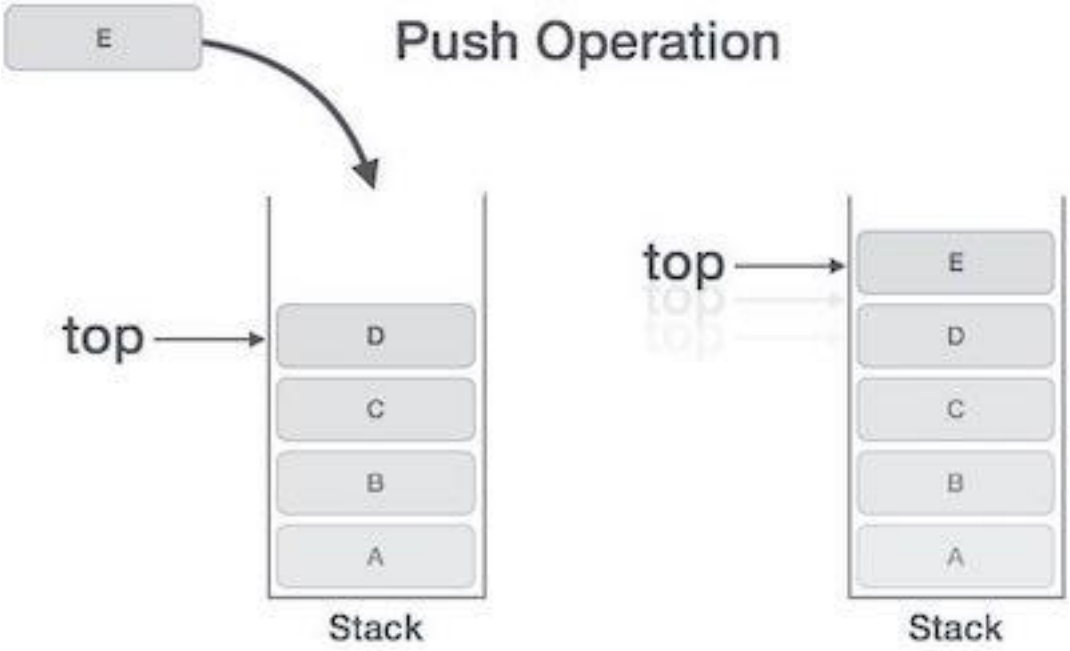
```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

- In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

  - **Step 1:** Check whether **stack** is **FULL**. (**top == SIZE-1**)
  - **Step 2:** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
  - **Step 3:** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

Push Operation



Algorithm for Push() operation

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

Implementation of push() in C/C++

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

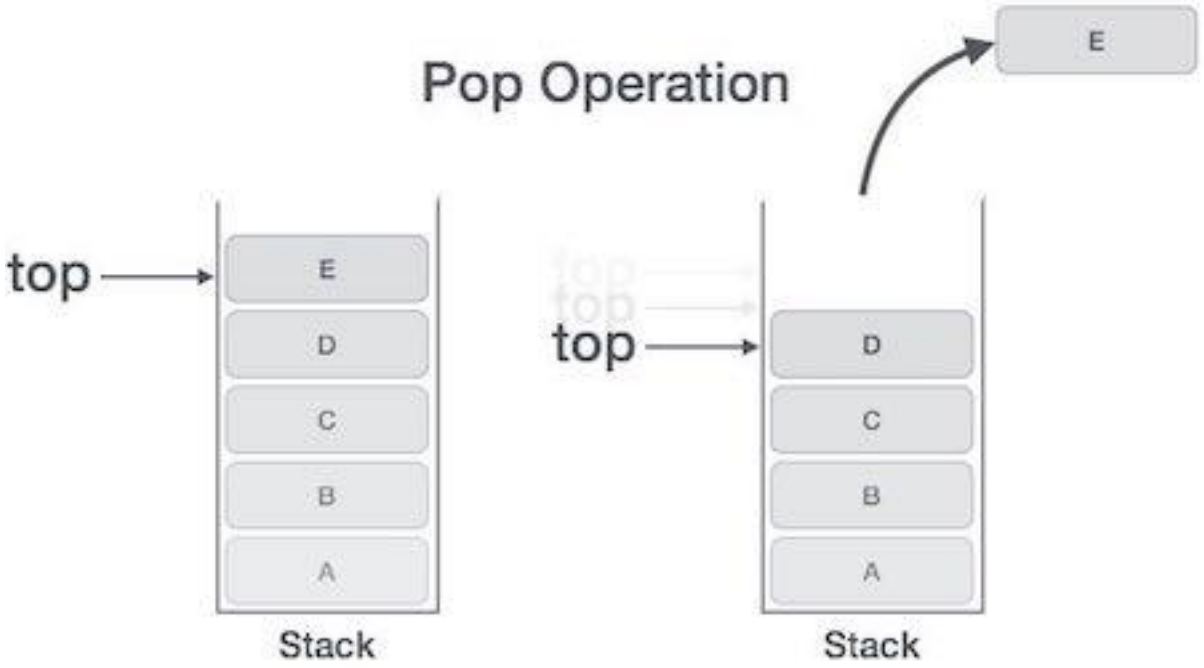
Pop Operation

- In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- **Step 1:** Check whether **stack** is **EMPTY**. (**top == -1**)

- **Step 2:** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

- **Step 3:** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).


Pop Operation
E
top
E
D
C
B
A
Stack
top
D
C
B
A
Stack

Algorithm for Pop() operation

```
begin procedure pop: stack
   if stack is empty
      return null
   endif
   data ← stack[top]
   top ← top - 1
   return data
end procedure
```

Implementation of pop() in C/C++

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Display operation

- We can use the following steps to display the elements of a stack...
- **Step 1:** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 3:** Repeat above step until **i** value becomes '0'.

Implementation of display() in C/C++

```
void display(){
  if(top == -1)
    printf("\nStack is Empty!!!");
  else{
    int i;
    printf("\nStack elements are:\n");
    for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
  }
}
```

- Program on Stack using array

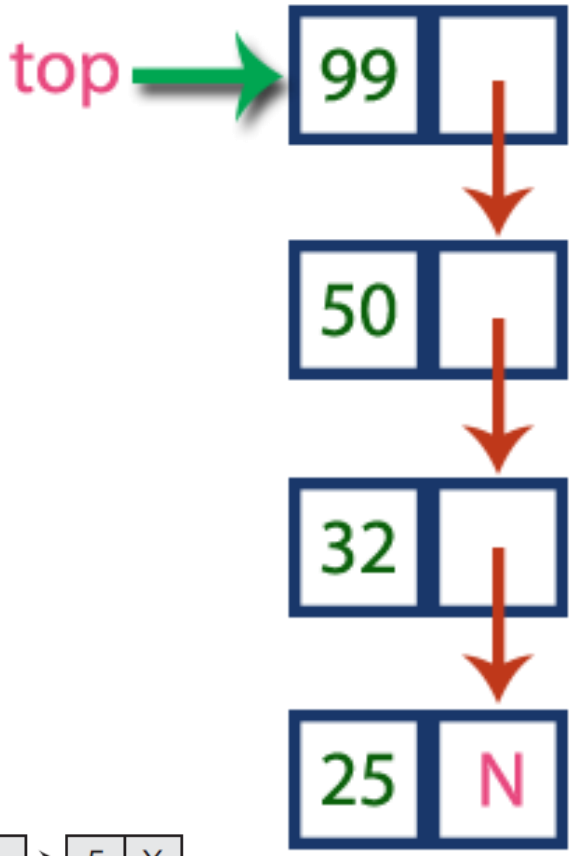
Stack Implementation using Linked List

- The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use.

- A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

- In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next** field of the first element must be always **NULL**.

- In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.



Stack Operations using Linked List

- To implement stack using linked list, we need to set the following things before implementing actual operations.

- **Step 1:** Include all the **header files** which are used in the program. And declare all the **user defined functions**.

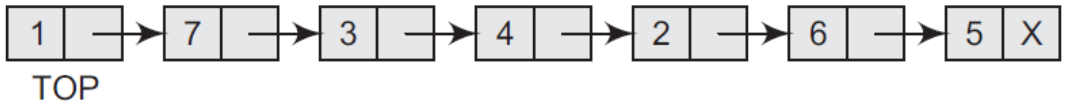
- **Step 2:** Define a '**Node**' structure with two members **data** and **next**.

- **Step 3:** Define a **Node** pointer '**top**' and set it to **NULL**.

- **Step 4:** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

Push Operation

- We can use the following steps to insert a new node into the stack...
  - **Step 1:** Create a **newNode** with given value.
  - **Step 2:** Check whether stack is **Empty** (**top** == **NULL**)
  - **Step 3:** If it is **Empty**, then set **newNode → next** = **NULL**.
  - **Step 4:** If it is **Not Empty**, then set **newNode → next** = **top**.
  - **Step 5:** Finally, set **top** = **newNode**.

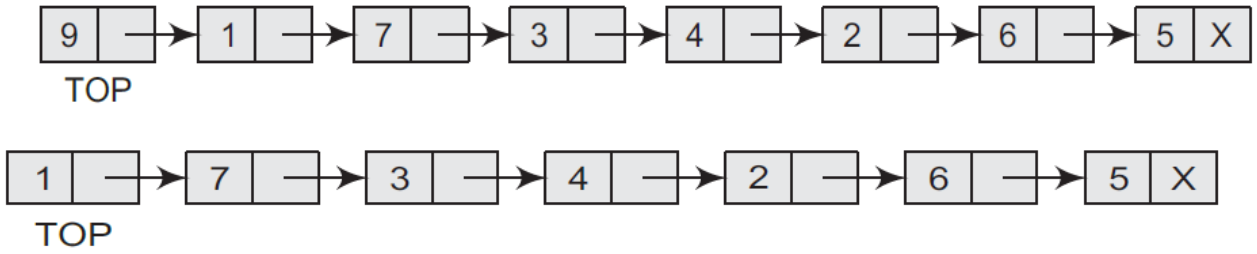


Algorithm for Push() operation

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Pop Operation

- We can use the following steps to delete a node from the stack...
  - **Step 1:** Check whether **stack** is **Empty** (**top == NULL**).
  - **Step 2:** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
  - **Step 3:** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
  - **Step 4:** Then set '**top = top → next**'.
  - **Step 7:** Finally, delete '**temp**' (**free(temp)**).



- Algorithm for Pop() operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

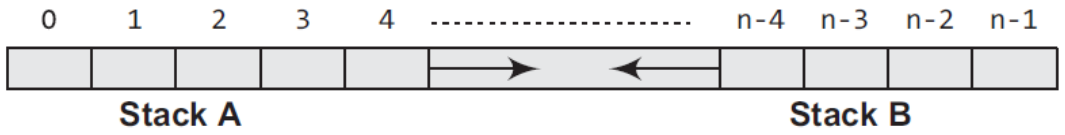
Display operation

- We can use the following steps to display the elements (nodes) of a stack...
  - **Step 1:** Check whether stack is **Empty** (**top** == **NULL**).
  - **Step 2:** If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function.
  - **Step 3:** If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.
  - **Step 4:** Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp → next** != **NULL**).
  - **Step 4:** Finally! Display '**temp → data ---> NULL**'.

- Program on Stack using linked list

Multiple Stack

- While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered.
- To deal with this problem, the code will have to be modified to reallocate more space for the array. In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory.
- Thus, there lies a trade-off between the frequency of overflows and the space allocated.
- So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size.



- An array STACK[n] is used to represent two stacks, Stack A and Stack B. The value of n is such that the combined size of both the stacks will never exceed n.
- While operating on these stacks, it is important to note one thing—Stack A will grow from left to right, whereas Stack B will grow from right to left at the same time.

- Program on multiple Stack.

Applications of Stacks

Applications of Stacks

- Stacks can be used on many applications.
  - Conversion of an infix expression into a postfix expression
  - Evaluation of a postfix expression
  - Conversion of an infix expression into a prefix expression
  - Evaluation of a prefix expression
  - Reversing a list
  - Parentheses checker
  - Recursion
  - Tower of Hanoi

Expressions

Expressions

- In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

- An expression can be defined as follows...

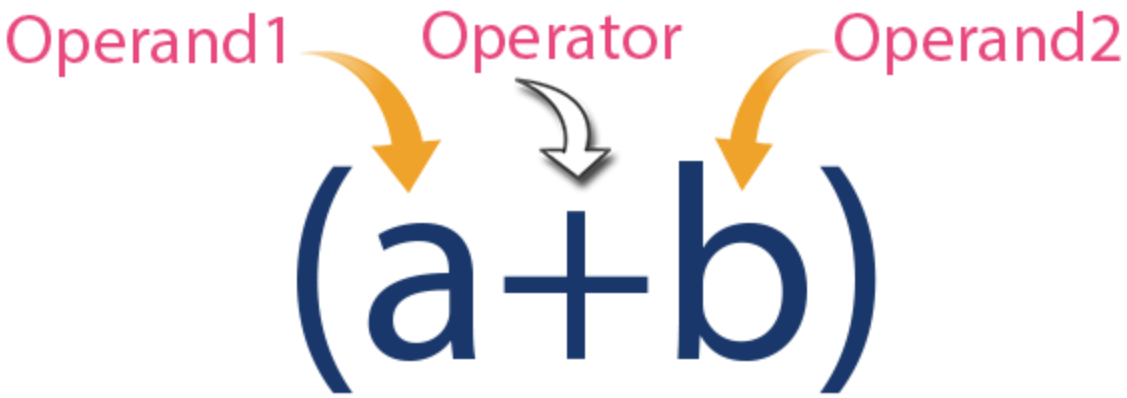
  - **An expression is a collection of operators and operands that represents a specific value.**

Expression Types

- Based on the operator position, expressions are divided into THREE types. They are as follows...
  - **Infix Expression**
  - **Postfix Expression**
  - **Prefix Expression**

Infix Expression

- In infix expression, operator is used in between operands.
- The general structure of an Infix expression is as follows...



Postfix Expression

- In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

- The general structure of Postfix expression is as follows...

Prefix Expression

- In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

- The general structure of Prefix expression is as follows...

Precedence

- When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others.
- For example −

- As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Precedence

- For example,

```
x = 7 + 3 * 2;
```

- Here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Associativity

- Associativity describes the rule where operators with the same precedence appear in an expression.
- For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.
- Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

- Precedence and associativity determines the order of evaluation of an expression.

- Following is an operator precedence and associativity table (highest to lowest)

Sr. No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

- Complete Precedence list

Level	Operators	Description	Associativity
15	() [] . new	Function Call Array Subscript Object Property Access Memory Allocation	Left to Right
14	++ -- + - ! ~ delete typeof void	Increment / Decrement Unary plus / minus Logical negation / bitwise complement Deallocation Find type of variable	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== != === !==	Equality Inequality Identity Operator Non Identity Operator	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

- The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

- In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition.

- But if we use use parenthesis for **a + b** . Then expression will be evaluated first, like **(a + b)*c**.

Expression Conversion

- To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

  - Find all the operators in the given Infix Expression.
  - Find the order of operators evaluated according to their Operator precedence.
  - Convert each operator into required type of expression (Postfix or Prefix) in the same order.

- Any expression can be represented using these three different types of expressions.
- And we can convert an expression from one form to another form like
  - **Infix to Postfix,**
  - **Infix to Prefix,**
  - **Prefix to Postfix**
  - **Prefix to infix**
  - **Postfix to prefix**
  - **Postfix to infix**

- **Example**

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

- **Step 1:** The Operators in the given Infix Expression : **= , + , ***
- **Step 2:** The Order of Operators according to their preference : *** , + , =**
- **Step 3:** Now, convert the first operator ***** ----- **D = A + B C ***
- **Step 4:** Convert the next operator **+** ----- **D = A BC* +**
- **Step 5:** Convert the next operator **=** ----- **D ABC*+ =**

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A-B)*(C+D)

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A-B)*(C+D)

    [AB–] * [CD+]

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A-B)*(C+D)

    [AB–] * [CD+]

    AB–CD+*

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A-B)*(C+D)

    [AB–] * [CD+]

    AB–CD+*

    [-AB] * [+CD]

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A-B)*(C+D)

    [AB–] * [CD+]

    AB–CD+*

    [-AB] * [+CD]

    *-AB+CD

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

    [AB+] / [CD+] – [DE*]

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

    [AB+] / [CD+] – [DE*]

    [AB+CD+/] – [DE*]

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

    [AB+] / [CD+] – [DE*]

    [AB+CD+/] – [DE*]

    AB+CD+/DE*–

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

    [AB+] / [CD+] – [DE*]

    [AB+CD+/] – [DE*]

    AB+CD+/DE*–

    [+AB] / [+CD] – [*DE]

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

    [AB+] / [CD+] – [DE*]

    [AB+CD+/] – [DE*]

    AB+CD+/DE*–

    [+AB] / [+CD] – [*DE]

    [/+AB+CD] – [*DE]

Exercise

- Convert the following infix expressions into postfix and prefix expressions
  - (A+B)/(C+D)-(D*E)

    [AB+] / [CD+] – [DE*]

    [AB+CD+/] – [DE*]

    AB+CD+/DE*–

    [+AB] / [+CD] – [*DE]

    [/+AB+CD] – [*DE]

    -/+AB+CD*DE

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

    (+AB)*C

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

    (+AB)*C

    *+ABC

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

    (+AB)*C

    *+ABC

  - **(A–B) * (C+D)**

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

    (+AB)*C

    *+ABC

  - **(A–B) * (C+D)**

    [–AB] * [+CD]

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

    (+AB)*C

    *+ABC

  - **(A–B) * (C+D)**

    [–AB] * [+CD]

    *–AB+CD

Convert the following infix expressions into prefix expressions.

- **(A + B) * C**

  (+AB)*C

  *+ABC

- **(A–B) * (C+D)**

  [–AB] * [+CD]

  *–AB+CD

- **(A + B) / (C + D) – (D * E)**

- Convert the following infix expressions into prefix expressions.

  - **(A + B) * C**

    (+AB)*C

    *+ABC

  - **(A–B) * (C+D)**

    [–AB] * [+CD]

    *–AB+CD

  - **(A + B) / (C + D) – (D * E)**

    [+AB] / [+CD] – [*DE]

Convert the following infix expressions into prefix expressions.

- **(A + B) * C**

  (+AB)*C

  *+ABC

- **(A–B) * (C+D)**

  [–AB] * [+CD]

  *–AB+CD

- **(A + B) / (C + D) – (D * E)**

  [+AB] / [+CD] – [*DE]

  [/+AB+CD] – [*DE]

Convert the following infix expressions into prefix expressions.

- **(A + B) * C**

  (+AB)*C

  *+ABC

- **(A–B) * (C+D)**

  [–AB] * [+CD]

  *–AB+CD

- **(A + B) / (C + D) – (D * E)**

  [+AB] / [+CD] – [*DE]

  [/+AB+CD] – [*DE]

  –/+AB+CD*DE