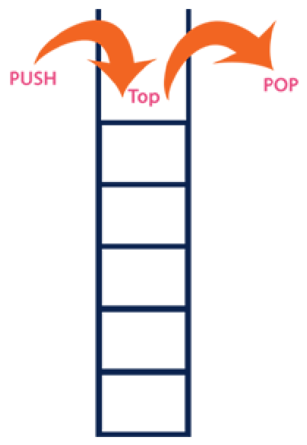
Stack

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called**"pop"**.

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows...

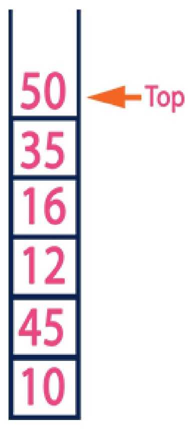
Stack is a linear data structure in which the operations are performed based on LIFO principle.

Stack can also be defined as

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below...



Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

Stack data structure can be implement in two ways. They are as follows...

1. **Using Array**
2. **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 Using Array

A stack data structure can be implemented using one dimensional array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable **'top'**. Initially top is set to -1 or 0.

Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

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

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.

push(value) - Inserting value into the stack

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.

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**).

pop() - Delete a value from the Stack

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.

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--**).

display() - Displays the elements of a Stack

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'.

```
void push(int value){
  if(top == SIZE-1)
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else{
    top++;
    stack[top] = value;
    printf("\nInsertion success!!!");
  }
}
void pop(){
  if(top == -1)
    printf("\nStack is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", stack[top]);
    top--;
  }
}
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]);
  }
}
```

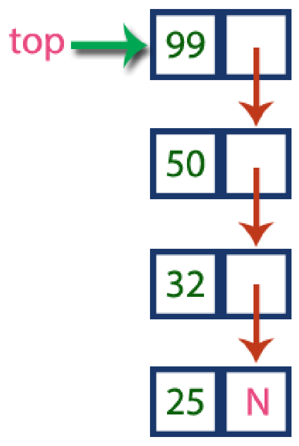
Stack using Linked List

The major problem with the stack implemented using array is the amount of data must be specified at the beginning of the implementation itself. It 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 can be used for unlimited number of values. That means, stack implemented using linked list works for variable size of data.

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**.

Example



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.

Operations

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(value) - Inserting an element into the Stack

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**.

pop() - Deleting an Element from a Stack

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)**).

display() - Displaying stack of elements

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**'.

```
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
             printf("%d--->",temp->data);
             temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   }
}
```

Expressions

An expression can be defined as

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

operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

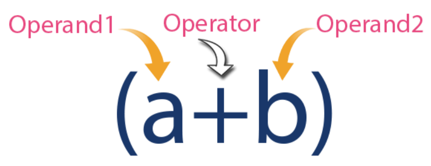
1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

Infix Expression

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as **Operand1 Operator Operand2**

Example

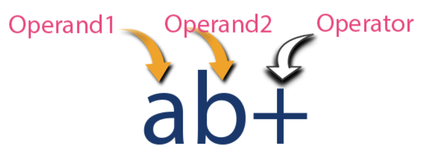


Postfix Expression

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

The general structure of Postfix expression is **Operand1 Operand2 Operator**

Example

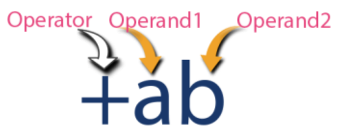


Prefix Expression

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

The general structure of Prefix expression is **Operator Operand1 Operand2**

Example



Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like **Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.**

To convert any Infix expression into Postfix or Prefix expression

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

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 **D A B C * + =**

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

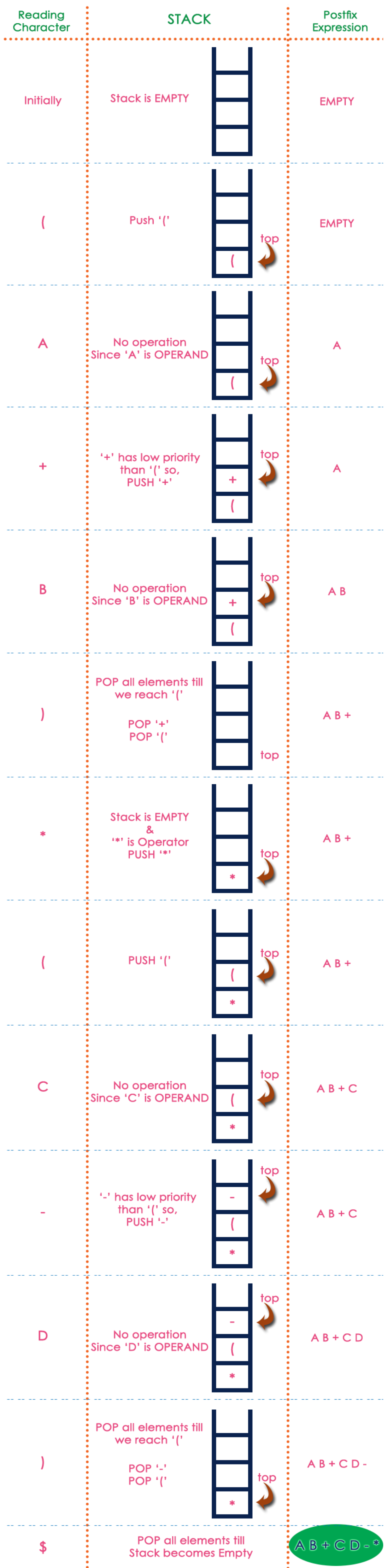
1. **Read all the symbols one by one from left to right in the given Infix Expression.**
2. **If the reading symbol is operand, then directly print it to the result (Output).**
3. **If the reading symbol is left parenthesis '(', then Push it on to the Stack.**
4. **If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.**
5. **If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.**

Example

Consider the following Infix Expression...

(A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as

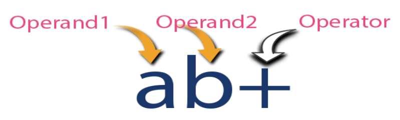


Reading Character	STACK		Postfix Expression
Initially	Stack is EMPTY		EMPTY
(	Push '('	top → (	EMPTY
A	No operation Since 'A' is OPERAND	top → (	A
+	'+' has low priority than '(' so, PUSH '+'	top → + (	A
B	No operation Since 'B' is OPERAND	top → + (	A B
)	POP all elements till we reach '(' POP '+' POP '('	top	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*'	top → *	A B +
(	PUSH '('	top → (*	A B +
C	No operation Since 'C' is OPERAND	top → (*	A B + C
-	'-' has low priority than '(' so, PUSH '-'	top → - (*	A B + C
D	No operation Since 'D' is OPERAND	top → - (*	A B + C D
)	POP all elements till we reach '(' POP '-' POP '('	top → *	A B + C D -
$	POP all elements till Stack becomes Empty		A B + C D - *

The final Postfix Expression is **A B + C D - ***

Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.



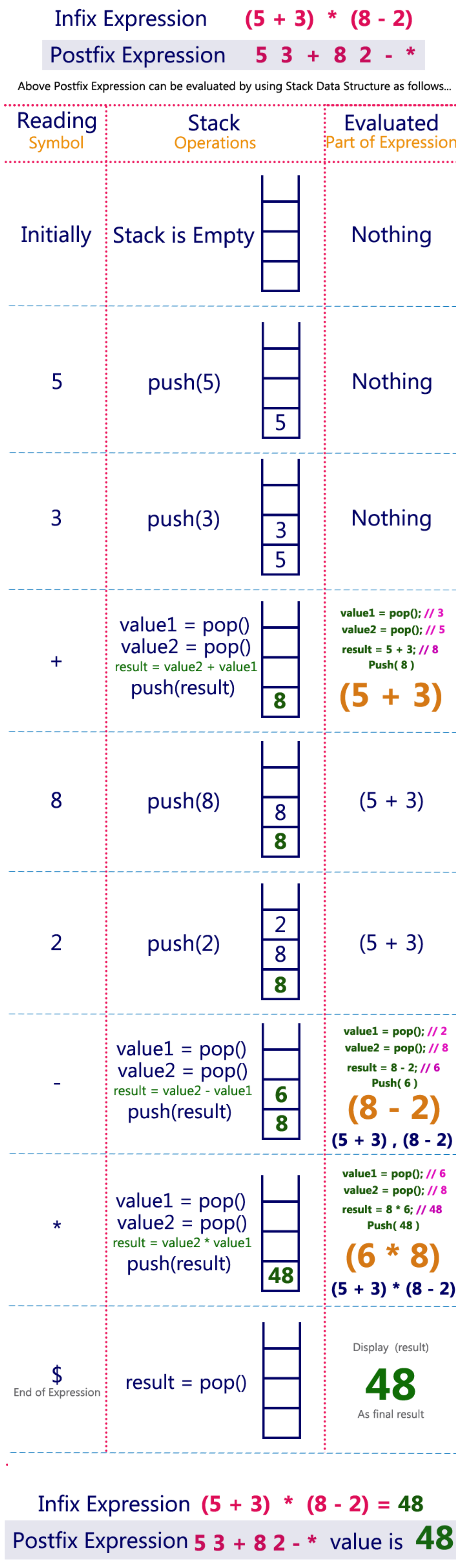
Postfix Expression has general structure **Operand1 Operand2 Operator**

Example : Consider the following Expression...

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. **Read all the symbols one by one from left to right in the given Postfix Expression**
2. **If the reading symbol is operand, then push it on to the Stack.**
3. **If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.**
4. **Finally! perform a pop operation and display the popped value as final result.**



Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing
5	push(5) 5	Nothing
3	push(3) 3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
	8
		value1 = pop(); // 3
		value2 = pop(); // 5
		result = 5 + 3; // 8
		Push(8)
		(5 + 3)

8	push(8) 8 8	(5 + 3)
2	push(2) 2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
	6
	8
		value1 = pop(); // 2
		value2 = pop(); // 8
		result = 8 - 2; // 6
		Push(6)
		(8 - 2)
		(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
	48
		value1 = pop(); // 6
		value2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3) * (8 - 2)

$ End of Expression	result = pop()	Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48

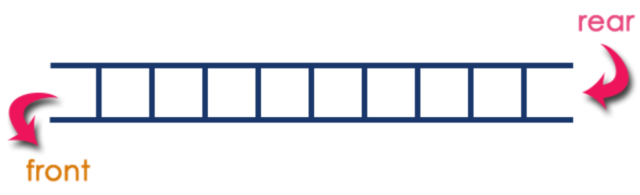
Postfix Expression 5 3 + 8 2 - * value is 48

Queue ADT

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end.

In a queue data structure, the insertion operation is performed at a posit

ion which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

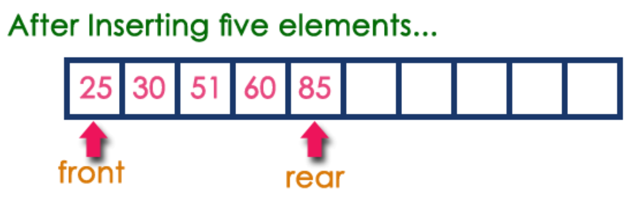
Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

A queue can also be defined as

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Example

Queue after inserting 25, 30, 51, 60 and 85.



Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways.

1. **Using Array**
2. **Using Linked List**

Queue Using Array

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values.

Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at '**front**' position as deleted element.

Queue Operations using Array

Steps to create an empty queue.

- **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 **user defined functions** which are used in queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**.

  (**int front = -1, rear = -1**)
- **Step 5:** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the queue.

steps to insert an element into the queue...

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

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter.

Steps to delete an element from the queue...

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Queue

Steps to display the elements of a queue...

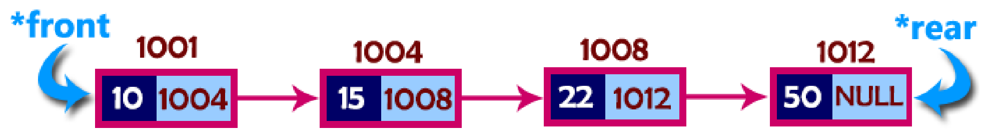
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 3:** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value is equal to **rear** (**i** <= **rear**)

Queue using Linked List

The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue 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 two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

Steps to insert a new node into the queue...

- **Step 1:** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2:** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3:** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4:** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

Steps to delete a node from the queue...

- **Step 1:** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2:** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4:** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

Steps to display the elements (nodes) of a queue...

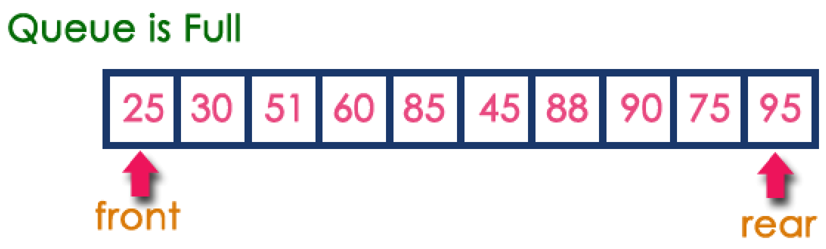
- **Step 1:** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2:** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.

- **Step 4:** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** !=NULL).
- **Step 4:** Finally! Display '**temp → data** ---> **NULL**'.

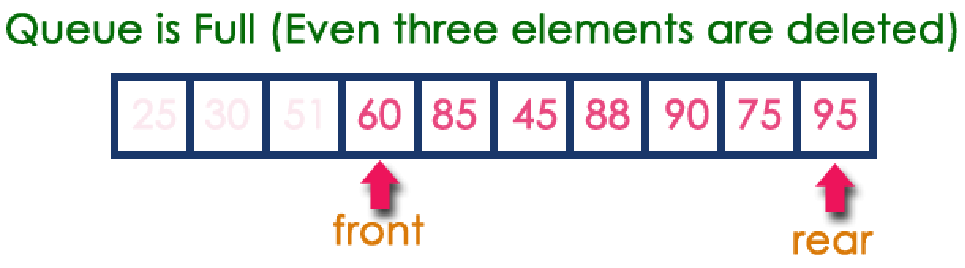
Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example consider the queue below...

After inserting all the elements into the queue.



Now consider the following situation after deleting three elements from the queue...

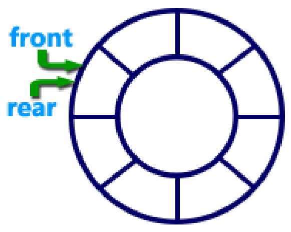


This situation also says that Queue is full and we cannot insert the new element because, '**rear**' is still at last position. Even though we have empty positions in the queue we cannot make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

What is Circular Queue?

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Implementation of Circular Queue

To implement a circular queue data structure using array, perform the following steps before we implement actual operations.

- **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 **user defined functions** used in circular queue implementation.

- **Step 3:** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**.

  (**int front = -1, rear = -1**)
- **Step 5:** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue.

Steps to insert an element into the circular queue...

- **Step 1:** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2:** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set

  rear = -1.
- **Step 4:** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**'

  if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as parameter.

Steps to delete an element from the circular queue...

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** ==**rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

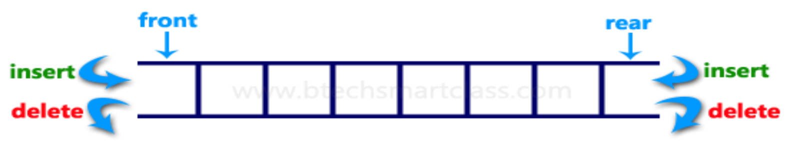
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4:** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5:** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.

- **Step 6:** Set **i** to **0**.
- **Step 7:** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until

  '**i <= rear**' becomes **FALSE**.

Double Ended Queue (Dequeue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

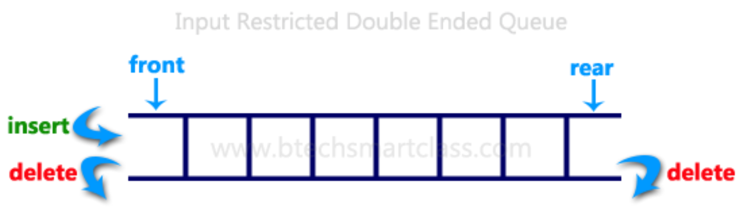


Double Ended Queue can be represented in TWO ways,

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

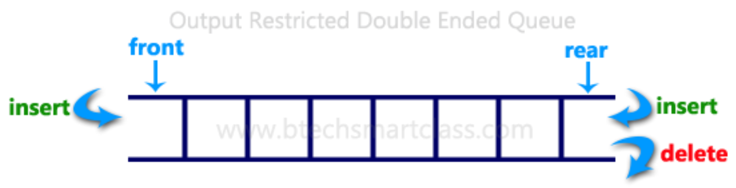
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



PUSH

Push ( ):

Description: Here STACK is an array with MAX locations. TOP points to the top most element and ITEM is the value to be inserted.

```
1. If (TOP == MAX) Then [Check for overflow]
2. Print: Overflow
3. Else
4. Set TOP = TOP + 1 [Increment TOP by 1]
5. Set STACK[TOP] = ITEM [Assign ITEM to top of STACK]
6. Print: ITEM inserted
[End of If]
7. Exit
```

POP

Pop ( ):

Description: Here STACK is an array with MAX locations. TOP points to the top most element.

```
1. If (TOP == 0) Then [Check for underflow]
2. Print: Underflow
3. Else
4. Set ITEM = STACK[TOP] [Assign top of STACK to ITEM]
5. Set TOP = TOP - 1 [Decrement TOP by 1]
6. Print: ITEM deleted
[End of If]
7. Exit
```

http://www.w3professors.com/data-structure-algorithms/delete-item-from-queue-dsa

INFIX TO POSTFIX EXPRESSION COVERSION

Description: Here I is an arithmetic expression written in infix notation and P is the equivalent postfix expression generated by this algorithm.

1. Push “(“ left parenthesis onto stack.

2. Add “)” right parenthesis to the end of expression I.

3. Scan I from left to right and repeat step 4 for each element of I until the stack becomes empty.

4. If the scanned element is:

(a) an operand then add it to P.

(b) a left parenthesis then push it onto stack.

(c) an operator then:

(i) Pop from stack and add to P each operator which has the same or higher precedence then the scanned operator.

(ii) Add newly scanned operator to stack.

(d) a right parenthesis then:

(i) Pop from stack and add to P each operator until a left parenthesis is encountered.

(ii) Remove the left parenthesis.

[End of Step 4 If]

[End of step 3 For Loop]

5.Exit.

POSTFIX EVALUATION

Evaluate ( ):

Description: Here P is a postfix expression and this algorithm evaluates it.

1. Add a “)” right parenthesis at the end of P.

2. Scan P from left to right and repeat steps 3 & 4 for each element of P until “)” is encountered.

3. If an operand is encountered, push it onto stack.

4. If an operator □ is encountered then:

(a) Pop the top two elements from stack, where A is the top element and B is the next to top element.

(b) Evaluate B □ A.

(c) Place the result of (b) back on stack.

[End of Step 4 If]

[End of step 2 For Loop]

5. Set VALUE equal to the top element on the stack.

6. Exit.

Enqueue and Dequeue in a Circular Queue

Algorithm Enqueue(QUEUE, n, front, rear, element)

/* QUEUE [0:n-1] is an array; 'front' stores the subscript value of the first element of the queue 'rear' stores the subscript value of the last element of the queue; 'front' and 'rear' are initialized to -1 when the queue is empty ; element is the new element to be inserted into the queue */

```
1. if (((front == 0) and (rear == n-1)) or (rear == front - 1)) //Check for overflow
2.    Print "Overflow"
3.    end Enqueue
3. end if
4. if (front == -1) //Inserting in an initially empty queue
5.      front = rear = 0
6.  end if
7. else if (rear == n-1) // Inserting after the last element which is at n-1
8.       rear = 0
9. end if
10. else
11.       rear = rear +1 //Increment rear
12. end else
13. QUEUE [rear]  =   element //
14.  end Enqueue
```

Algorithm Dequeue (QUEUE, n, front,rear)

/* QUEUE [0:n-1] is an array ; 'front' stores the subscript value of the first element of the queue ; 'rear' stores the subscript value of the last element of the queue ; 'front' and 'rear' are initialized to -1 when the queue is empty ; element is the element deleted from the queue */

```
1. if(front == -1) //Check for overflow
2.    Print "Underflow"
3.    end Dequeue
3. end if
4. element = QUEUE[front] // Delete the front element
5. if(front == rear) // The last element is deleted
6.     front = rear   = -1
7. end if
8. else if (front == n-1)
9.     front = 0
10. end if
11. else
12.      front = front + 1//Increment front
```

```
13. end else
14. end Dequeue
```

Insert Element into Queue

Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the QUEUE. ITEM is the value to be inserted.

```
1.  If (REAR == N) Then          [Check for overflow]
2.    Print: Overflow
3.  Else
4.   If (FRONT and REAR == 0) Then     [Check if QUEUE is empty]
(a)  Set FRONT = 1
(b)  Set REAR = 1
5.    Else
6.    Set REAR = REAR + 1       [Increment REAR by 1]
    [End of Step 4 If]
7.    QUEUE[REAR] = ITEM
8.    Print: ITEM inserted
   [End of Step 1 If]
9.  Exit
```

Algorithm to Delete Element From Queue

Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the QUEUE.

```
1.  If (FRONT == 0) Then              [Check for underflow]
2.    Print: Underflow
 3.  Else
4.    ITEM = QUEUE[FRONT]
 5.    If (FRONT == REAR) Then        [Check if only one element is left]
(a)  Set FRONT = 0
(b)  Set REAR = 0
 6.    Else
7.      Set FRONT = FRONT + 1      [Increment FRONT by 1]
[End of Step 5 If]
 8.    Print: ITEM deleted
[End of Step 1 If]
9.  Exit
```

Insert Element into Circular Queue

Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear elements of the QUEUE. ITEM is the value to be inserted.

```
1.  If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
2.    Print: Overflow
3.  Else
4.    If (REAR == 0) Then       [Check if QUEUE is empty]
 (a) Set FRONT = 1
(b) Set REAR = 1
5.    Else If (REAR == N) Then      [If REAR reaches end if QUEUE]
6.       Set REAR = 1
7.    Else
8.      Set REAR = REAR + 1     [Increment REAR by 1]
      [End of Step 4 If]
9.      Set QUEUE[REAR] = ITEM
10.     Print: ITEM inserted
       [End of Step 1 If]
11.  Exit
```

Algorithm to Delete Element From Circular Queue

Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear elements of the QUEUE.

```
1.  If (FRONT == 0) Then          [Check for Underflow]
2.    Print: Underflow
3.  Else
 4.    ITEM = QUEUE[FRONT]
5.    If (FRONT == REAR) Then       [If only element is left]

(a) Set FRONT = 0
(b) Set REAR = 0
 6.    Else If (FRONT == N) Then       [If FRONT reaches end if QUEUE]
7.       Set FRONT = 1
 8.    Else
9.      Set FRONT = FRONT + 1      [Increment FRONT by 1]
 [End of Step 5 If]
10.    Print: ITEM deleted
[End of Step 1 If]
11.  Exit
```