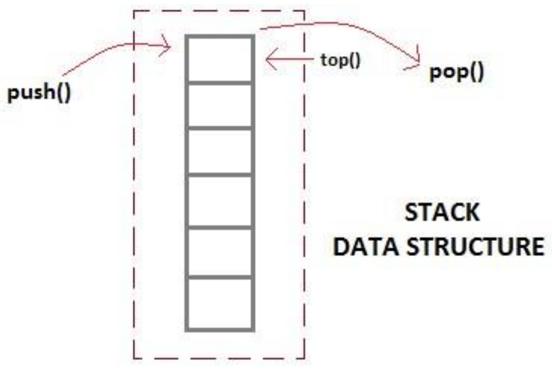
STACK AND QUEUE

STACK: STACK – Array implementation, STACK – Linked list implementation, Applications of STACK – Infix to Postfix, Evaluation of Postfix, Balancing symbols, Nested function calls, Recursion, Towers of Hanoi. QUEUE: QUEUE – Array implementation, QUEUE – Linked List implementation, Circular Queue, Applications of QUEUE – Priority queue – Double ended queue.

INTRODUCTION

Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. 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.
4. 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.

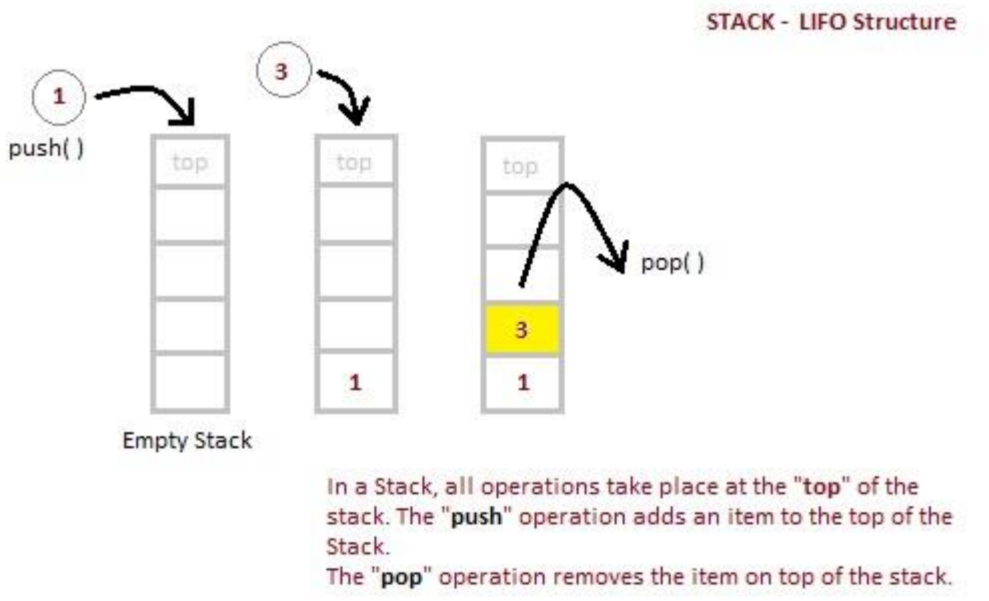
Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like : Parsing, Expression Conversion(Infix to Postfix, Postfix to Prefix etc) and many more.

Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.


STACK - LIFO Structure
1
push( )
3
top
top
top
pop( )
3
1
1
Empty Stack
In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Step 1 : Declare One Stack Structure

```
#define     size     5
struct stack
{
   int     s[size];
int top;
}st;
```

1. We have created 'stack' structure.
2. We have array of elements having size 'size'
3. To keep track of Topmost element we have declared top as structure member.

Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using 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. 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...

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.

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. 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**).

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. 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--**).

display() - Displays the elements of a Stack

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

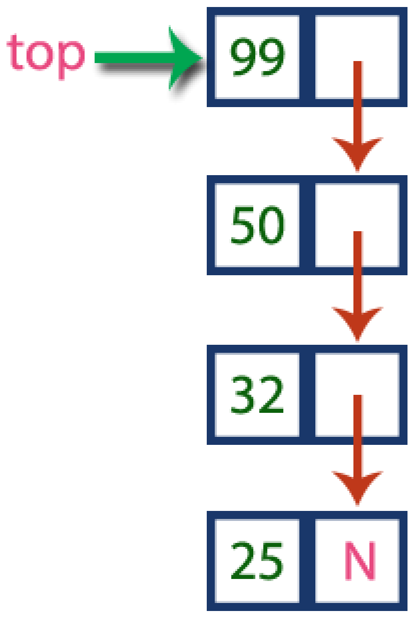
- **Step 1:** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top.
  Display **stack[i]** value and decrement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

Stack 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**.

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

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**. **pop() - Deleting an Element from a Stack**

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)**).

display() - Displaying stack of elements

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**'.

Applications of STACK

1. **Infix to Postfix**
2. **Evaluation of Postfix**
3. **Balancing symbols**
4. **Nested function calls**
5. **Recursion**
6. **Towers of Hanoi.**

1. Infix to Postfix

The way to write arithmetic expression is known as **notation.** An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

Infix Notation

We write expression in **infix** notation, e.g. **a-b+c**, where operators are used **in**-between operands. It is easy for us humans to read, write and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example **+ab**. This is equivalent to its infix notation **a+b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, operator is **postfix**ed to the operands i.e., operator is written after the operands. For example **ab+**. This is equivalent to its infix notation **a+b**.

Algorithm for Infix to Postfix

1. Examine the next element in the input.
2. If it is operand, output it.
3. If it is opening parenthesis, push it on stack.
4. If it is an operator, then
    - If stack is empty, push operator on stack.
    - If the top of stack is opening parenthesis, push operator on stack
    - If it has higher priority than the top of stack, push operator on stack.
    - Else pop the operator from the stack and output it, repeat step 4
5. If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
6. If there is more input go to step 1
7. If there is no more input, pop the remaining operators to output.

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*+ =

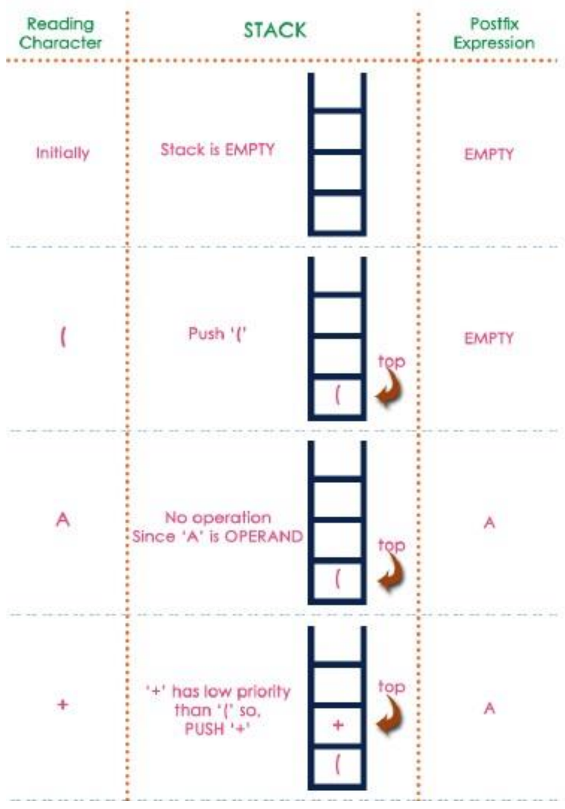
Finally, given Infix Expression is converted into Postfix Expression as follows... D A B C * + =

Example

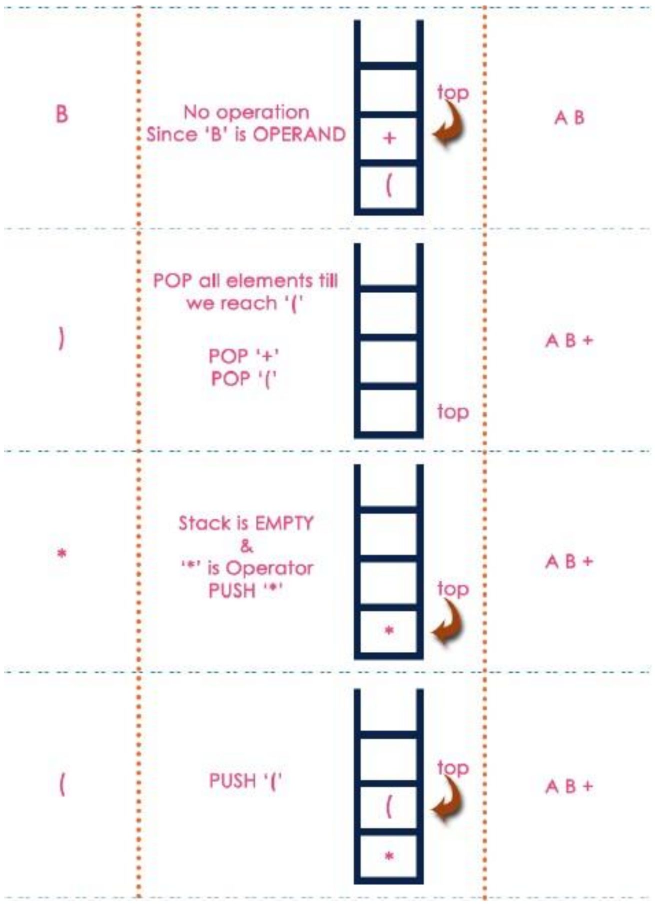
Consider the following Infix Expression...

(A + B) * (C - D)

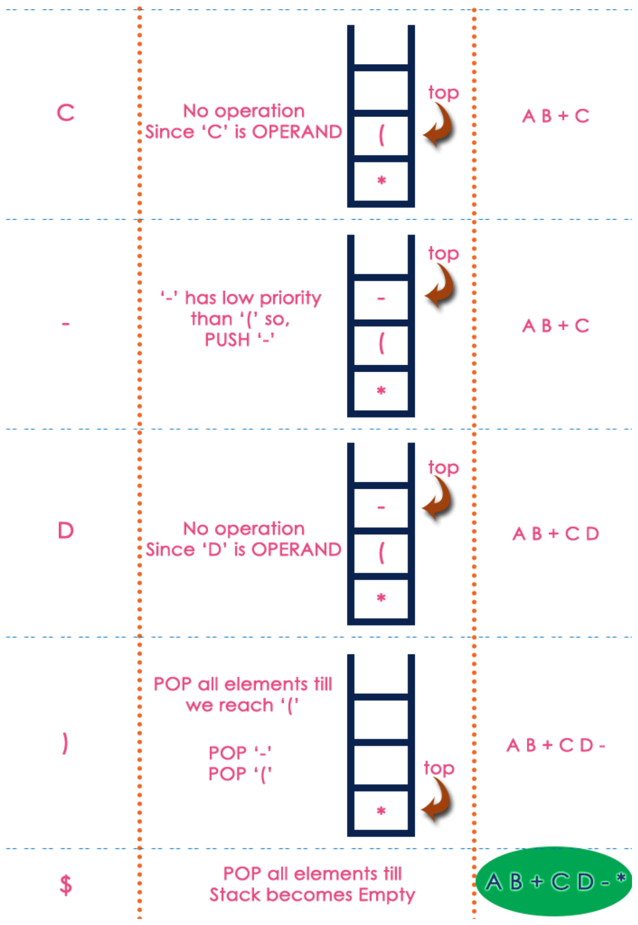
The given infix expression can be converted into postfix expression using Stack data Structure as follows...



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 as follows... A B + C D - *

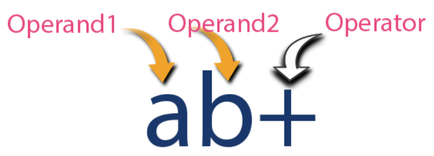
1. Evaluation of Postfix

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 following general structure...

Operand1 Operand2 Operator

Example



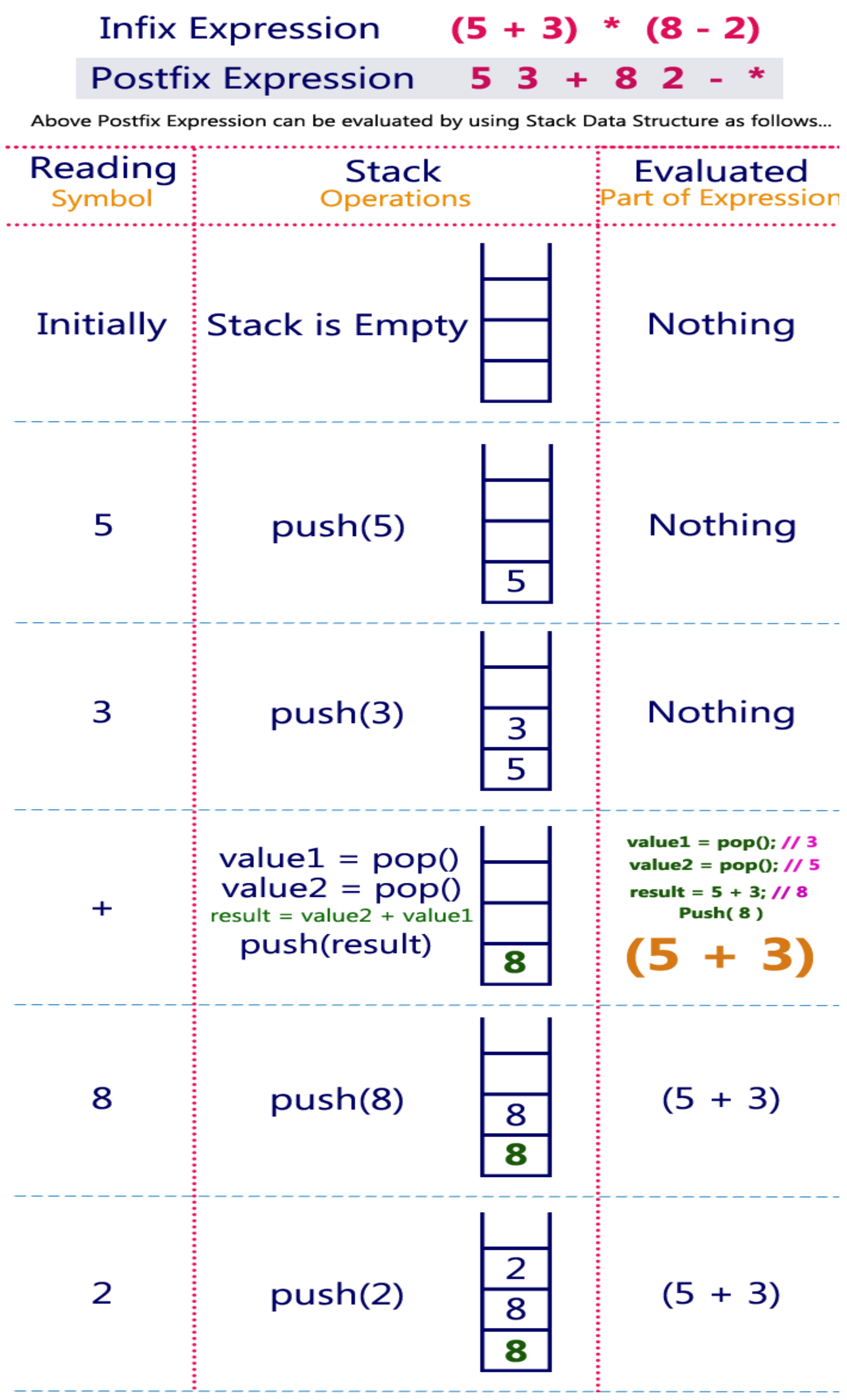
Postfix Expression Evaluation using Stack Data Structure

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.

Example

Consider the following Expression...



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

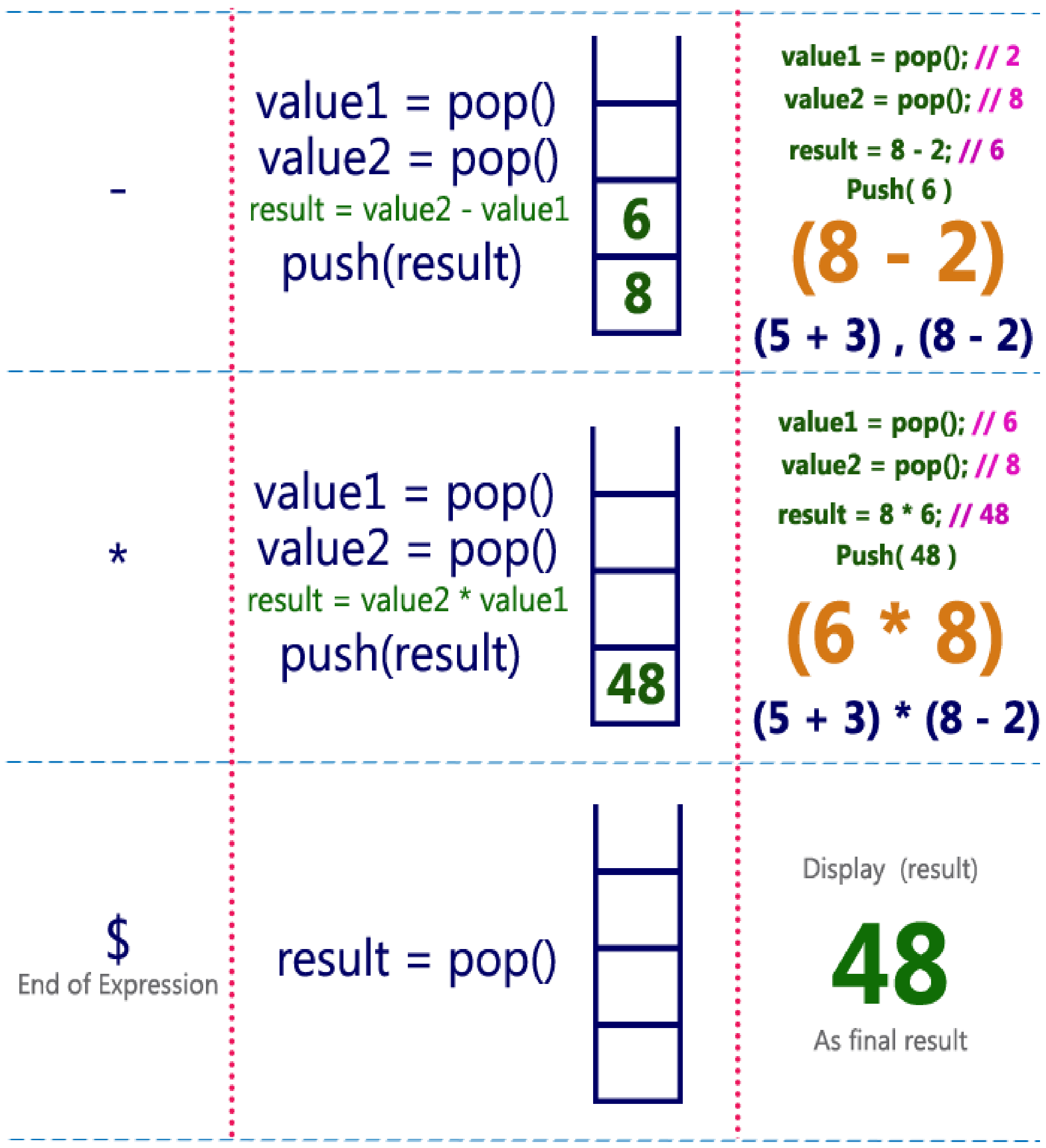
Postfix Expression 5 3 + 8 2 - *

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

Example:

Let the given expression be “2 3 1 * + 9 -“. We scan all elements one by one.

1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’
2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3′ (from bottom to top)
3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1′
4) Scan ‘*’, it’s an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result ‘3’ to stack. Stack now becomes ‘2 3′.
5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result ‘5’ to stack. Stack now becomes ‘5’.
6) Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9′. 7) Scan ‘-‘, it’s an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result ‘-4′ to stack. Stack now becomes ‘-4′.
8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack)

2. Balancing Symbols

This program uses stack to check balanced expression. The expression is parsed character by character and when opening bracket is found, it is inserted into stack. When the equivalent closing bracket is found, the stack is emptied. If the stack is empty after parsing the entire expression, then the expression is balanced expression.

Algorithm:

1. Whenever we see a opening parenthesis, we put it on stack.
2. For closing parenthesis, check what is at the top of the stack, if it corresponding opening parenthesis, remove it from the top of the stack.
3. If parenthesis at the top of the stack is not corresponding opening parenthesis, return false, as there is no point check the string further.

4. After processing entire string, check if stack is empty or not.

    4.a If the stack is empty, return true.

    4.b If stack is not empty, parenthesis do not match.

Example

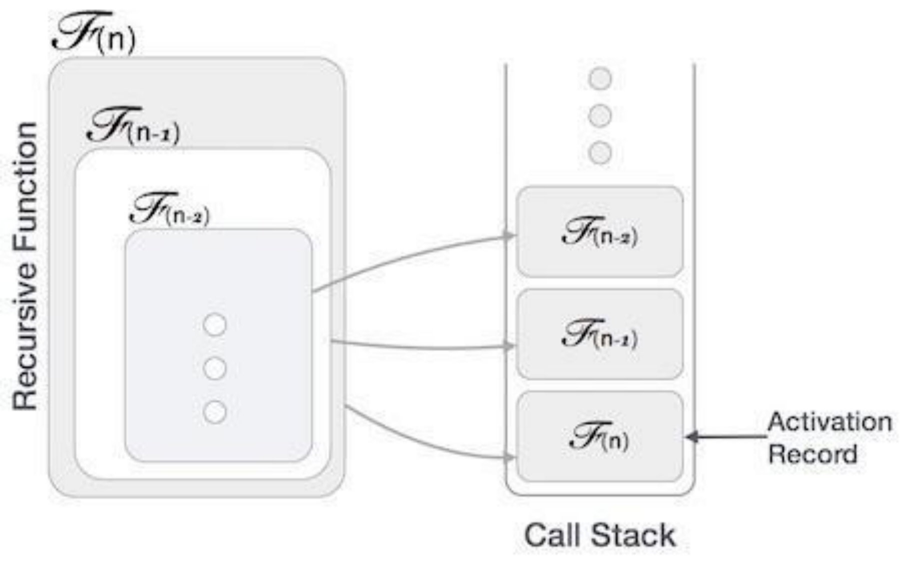
(( ) (( )))

Input	Input	Operation	Stack Symbol
(( ) (( )))	(	PUSH	(
( ) (( )))	(	PUSH	((
) (( )))	)	POP	(
(( )))	(	PUSH	((
( )))	(	PUSH	(((
)))	)	POP	((
))	)	POP	(
)	)	POP	Empty (Accepted)

3. Recursion

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to callee. This transfer process may also involve some data to be passed from caller to callee.

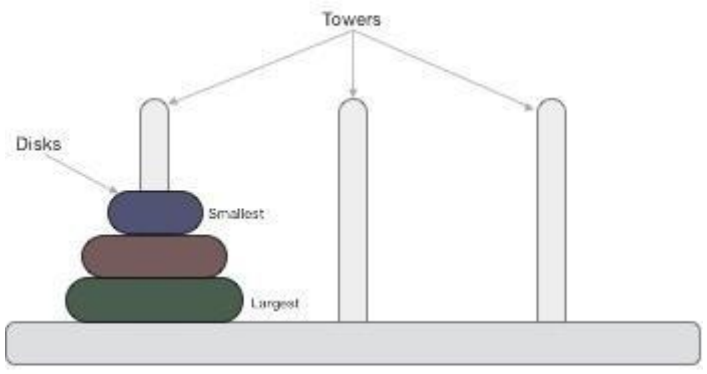
This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from callee function. Here, caller function needs to start exactly from the point of execution where it put itself on hold. It also needs the exact same data values it was working on. For this purpose an activation record (or stack frame) is created for caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to called function.

4. Towers of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three tower (pegs) and more than one rings; as depicted below –



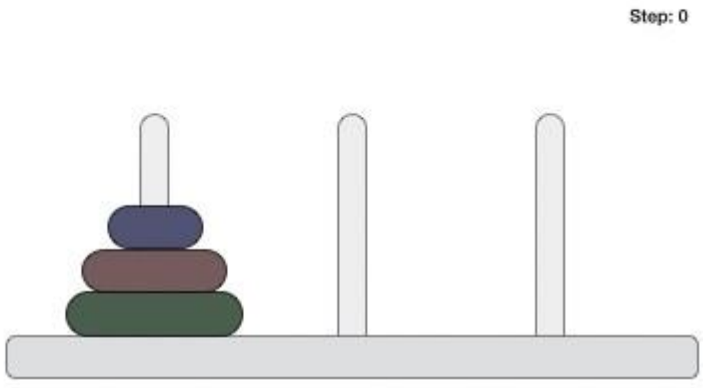
These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. The below mentioned are few rules which are to be followed for tower of hanoi –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Here is an animated representation of solving a tower of hanoi puzzle with three disks –



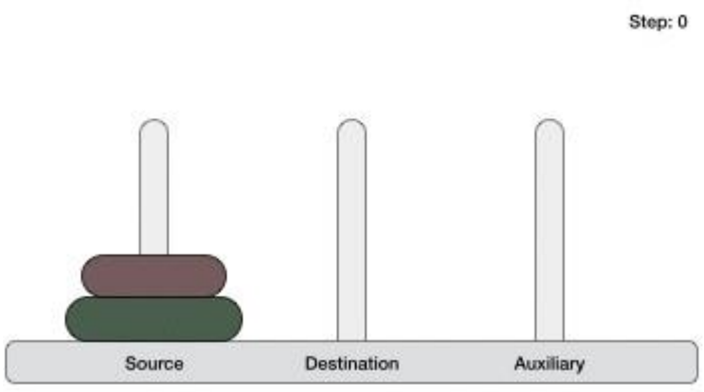
Tower of hanoi puzzle with **n** disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3-1}$ = 7 steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First we move the smaller one (top) disk to aux peg
- Then we move the larger one (bottom) disk to destination peg
- And finally, we move the smaller one from aux to destination peg.



So now we are in a position to design algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nthdisk) is in one part and all other (n-1) disks are in second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. Now we can imagine to apply the same in recursive way for all given set of disks. So steps to follow are –

Step 1 – Move n-1 disks from **source** to **aux**

Step 2 – Move nth disk from **source** to **dest**

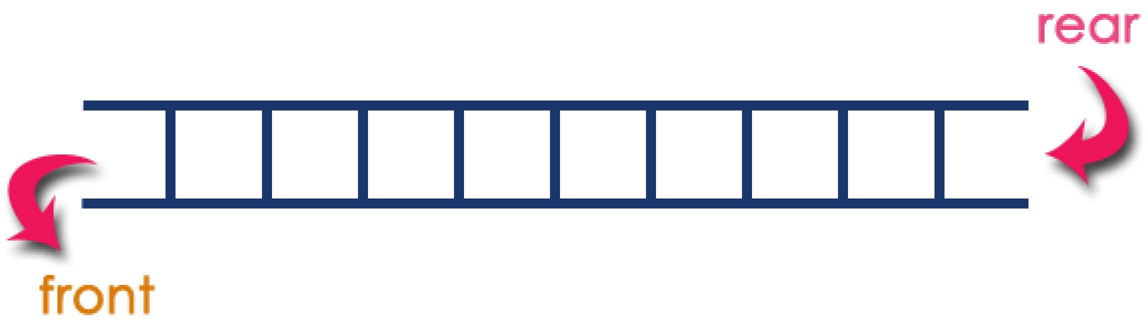
Step 3 – Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)
IF disk == 0, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF
END Procedure
STOP
```

QUEUE

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 position 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 can be defined as follows...

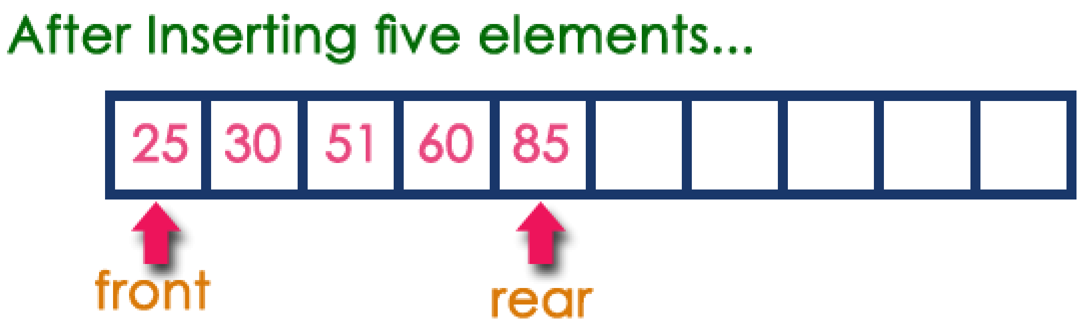
Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

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. They are as follows...

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

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

Applications of Queue:

Queue is used when things don't have to be processed immediatly, but have to be processed in First InFirst Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Array implementation Of Queue

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. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. 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

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below 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.

- **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. We can use the following steps

- **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

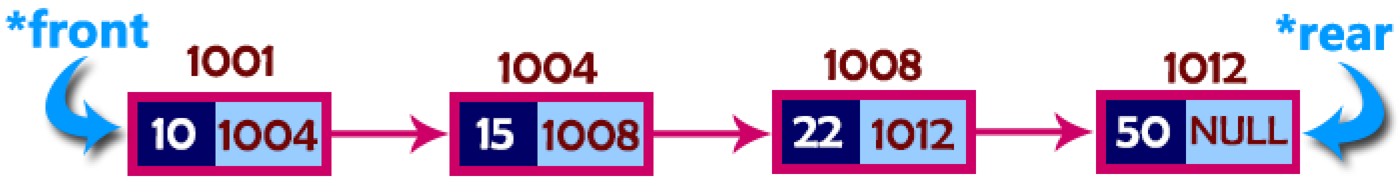
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 3:** Display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

Linked List implementation Of Queue

Queue using Linked List

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. 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

We can use the following 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

We can use the following 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

We can use the following steps to display the elements (nodes) of a queue...

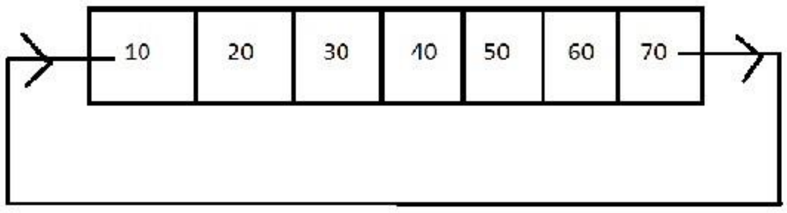
- **Step 1:** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2:** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4:** Display '**temp → 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 standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue. Circular queue is a linear data structure. It follows FIFO principle.

In circular queue the last node is connected back to the first node to make a circle.

- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".
- Items can inserted and deleted from a queue in O(1) time.



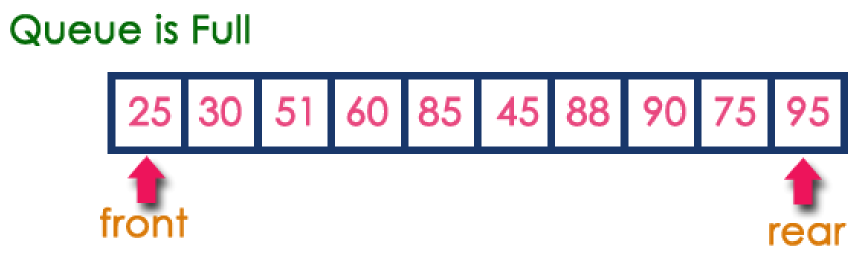
Circular Queue

Circular Queue can be created in three ways they are

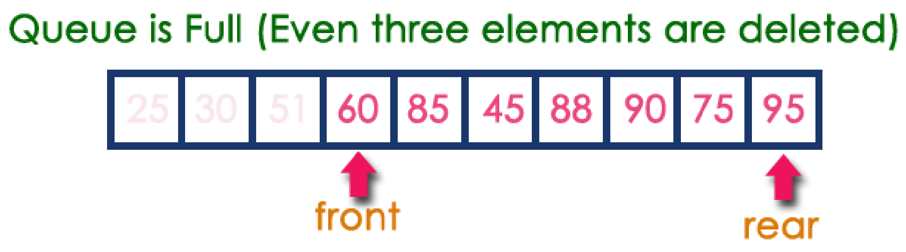
- Using single linked list
- Using double linked list
- Using arrays

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we can not 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...

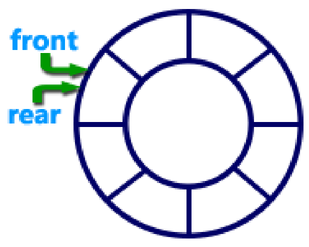


This situation also says that Queue is Full and we can not insert the new element because, 'rear' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure.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

A Circular Queue can be defined as follows...

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, we first 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

In a 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. We can use the following 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

In a 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. We can use the following 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.

Applications of QUEUE

1. Priority queue
2. Double ended queue.

Priority Queue

In normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all situations.

Consider a networking application where server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows...

Now, check waiting time for each request to be complete.

1. R1 : 20 units of time

2. R2 : 22 units of time (R2 must wait till R1 complete - 20 units and R2 itself requeres 2 units. Total 22 units)

3. R3 : 32 units of time (R3 must wait till R2 complete - 22 units and R3 itself requeres 10 units. Total 32 units)

4. R4 : 37 units of time (R4 must wait till R3 complete - 35 units and R4 itself requeres 5 units. Total 37 units)

Here, average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.

Now, consider another way of serving these requests. If we serve according to their required amount of time. That means, first we serve R2 which has minimum time required (2) then serve R4 which has second minimum time required (5) then serve R3 which has third minimum time required (10) and finnaly R1 which has maximum time required (20).

Now, check waiting time for each request to be complete.

1. R2 : 2 units of time

2. R4 : 7 units of time (R4 must wait till R2 complete 2 units and R4 itself requeres 5 units. Total 7 units)

3. R3 : 17 units of time (R3 must wait till R4 complete 7 units and R3 itself requeres 10 units. Total 17 units)

4. R1 : 37 units of time (R1 must wait till R3 complete 17 units and R1 itself requeres 20 units. Total 37 units)

Here, average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.

From above two situations, it is very clear that, by using second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the priority queue.

Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

There are two types of priority queues they are as follows...

1. **Max Priority Queue**

2. **Min Priority Queuerity Queue**

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

Priority Queue is an extension of queue with following properties.

1) Every item has a priority associated with it.

2) An element with high priority is dequeued before an element with low priority.

3) If two elements have the same priority, they are served according to their order in the queue.

The following are the operations performed in a Max priority queue...

1. isEmpty() - Check whether queue is Empty.
2. insert() - Inserts a new value into the queue.
3. findMax() - Find maximum value in the queue.
4. remove() - Delete maximum value from the queue.

Max Priority Queue Representations

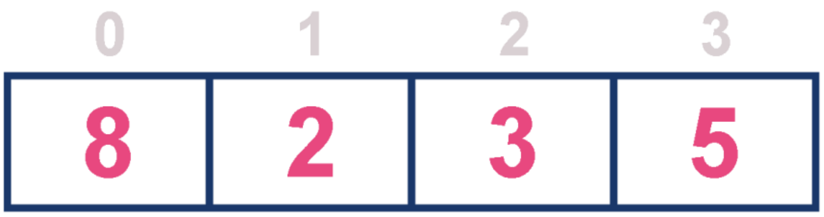
There are 6 representations of max priority queue.

1. Using an Unordered Array (Dynamic Array)
2. Using an Unordered Array (Dynamic Array) with the index of the maximum value
3. Using an Array (Dynamic Array) in Decreasing Order
4. Using an Array (Dynamic Array) in Increasing Order
5. Using Linked List in Increasing Order
6. Using Unordered Linked List with reference to node with the maximum value

#1. Using an Unordered Array (Dynamic Array)

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



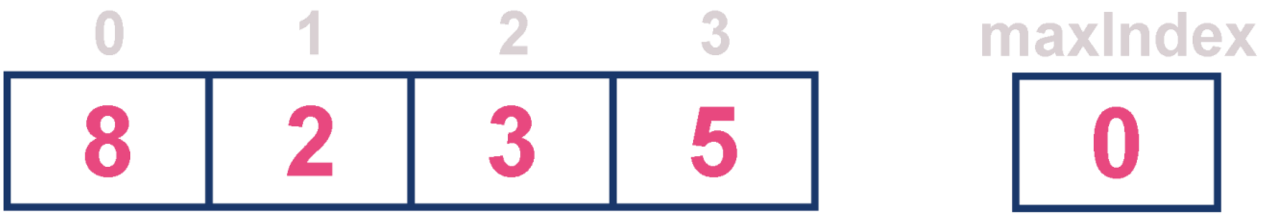
Now, let us analyse each operation according to this representation...

- isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.
- insert() - New element is added at the end of the queue. This operation requires O(1) time complexity that means constant time.
- findMax() - To find maximum element in the queue, we need to compare with all the elements in the queue. This operation requires O(n)time complexity.
- remove() - To remove an element from the queue first we need to perform findMax() which requires O(n) and removal of particular element requires constant time O(1). This operation requires O(n) time complexity.

#2. Using an Unordered Array (Dynamic Array) with the index of the maximum value

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



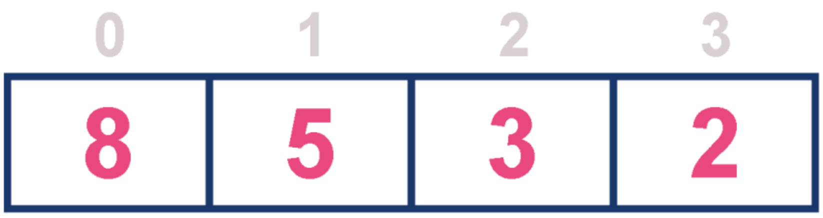
Now, let us analyse each operation according to this representation...

- isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.
- insert() - New element is added at the end of the queue with O(1) and for each insertion we need to update maxIndex with O(1). This operation requires O(1) time complexity that means constant time.
- findMax() - To find maximum element in the queue is very simple as maxIndex has maximum element index. This operation requires O(1)time complexity.
- remove() - To remove an element from the queue first we need to perform findMax() which requires O(1) , removal of particular element requires constant time O(1) and update maxIndex value which requires O(n). This operation requires O(n) time complexity.

#3. Using an Array (Dynamic Array) in Decreasing Order

In this representation elements are inserted according to their value in decreasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

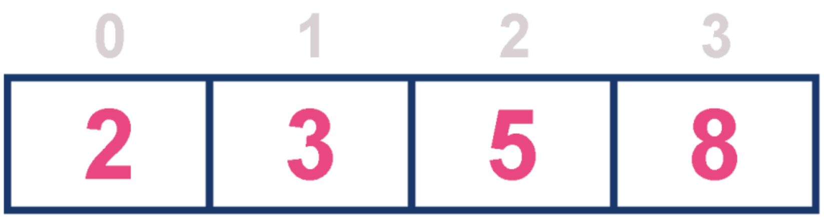
- isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.
- insert() - New element is added at a particular position in the decreasing order into the queue with O(n), because we need to shift existing elements inorder to insert new element in decreasing order. This operation requires O(n) time complexity.

- findMax() - To find maximum element in the queue is very simple as maximum element is at the beginning of the queue. This operation requires O(1) time complexity.
- remove() - To remove an element from the queue first we need to perform findMax() which requires O(1), removal of particular element requires constant time O(1) and rearrange remaining elements which requires O(n). This operation requires O(n) time complexity.

#4. Using an Array (Dynamic Array) in Increasing Order

In this representation elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

- isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.

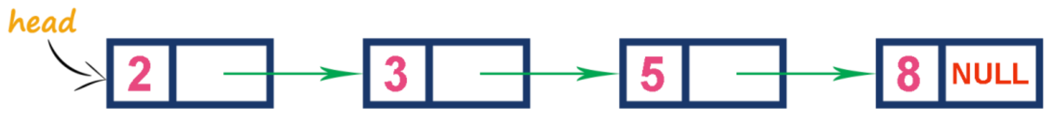
- insert() - New element is added at a particular position in the increasing order into the queue with O(n), because we need to shift existing elements inorder to insert new element in increasing order. This operation requires O(n) time complexity.
- findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requiresO(1) time complexity.
- remove() - To remove an element from the queue first we need to perform findMax() which requires O(1), removal of particular element requires constant

time O(1) and rearrange remaining elements which requires O(n). This operation requires O(n) time complexity.

#5. Using Linked List in Increasing Order

In this representation, we use a single linked list to represent max priority queue. In this representation elements are inserted according to their value in increasing order and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



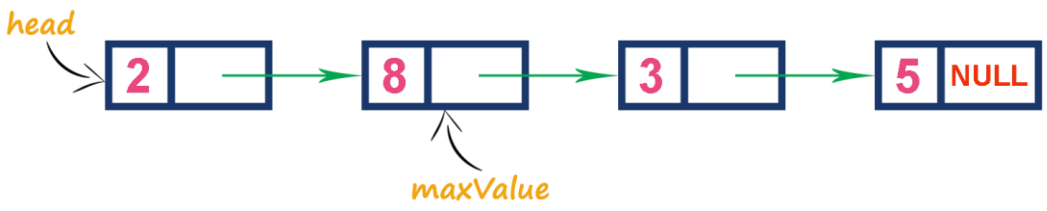
Now, let us analyse each operation according to this representation...

- isEmpty() - If 'head == NULL' queue is Empty. This operation requires O(1) time complexity that means constant time.
- insert() - New element is added at a particular position in the increasing order into the queue with O(n), because we need to the position where new element has to be inserted. This operation requires O(n) time complexity.
- findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requiresO(1) time complexity.
- remove() - To remove an element from the queue is simply removing the last node in the queue which requires O(1). This operation requires O(1) time complexity.

#6. Using Unordered Linked List with reference to node with the maximum value

In this representation, we use a single linked list to represent max priority queue. Always we maitain a reference (maxValue) to the node with maximum value. In this representation elements are inserted according to their arrival and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

- isEmpty() - If 'head == NULL' queue is Empty. This operation requires O(1) time complexity that means constant time.
- insert() - New element is added at end the queue with O(1) and update maxValue reference with O(1). This operation requires O(1) time complexity.
- findMax() - To find maximum element in the queue is very simple as maxValue is referenced to the node with maximum value in the queue. This operation requires O(1) time complexity.
- remove() - To remove an element from the queue is deleting the node which referenced by maxValue which requires O(1) and update maxValue reference to new node with maximum value in the queue which requires O(n) time complexity. This operation requires O(n)time complexity.

2. Min Priority Queue Representations

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

The following operations are performed in Min Priority Queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**

3. **findMin() - Find minimum value in the queue.**
4. **remove() - Delete minimum value from the queue.**

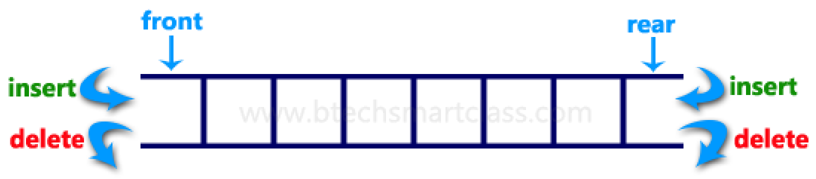
Min priority queue is also has same representations as Max priority queue with minimum value removal.

Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

Double ended Queue or Dequeue

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

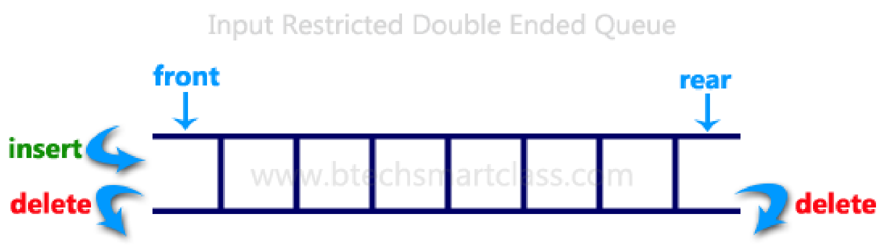


Double Ended Queue can be represented in TWO ways, those are as follows...

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

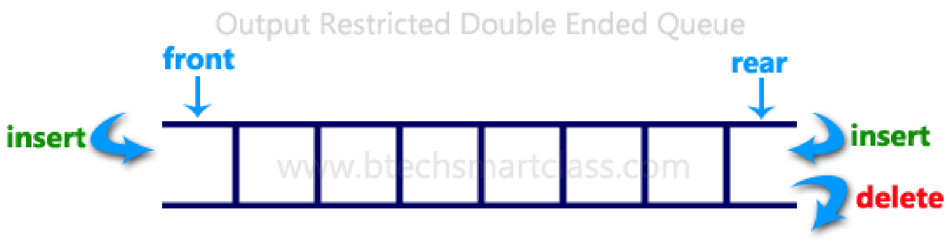
Input Restricted Double Ended Queue

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.



Applications of Dequeue:

Since Dequeue supports both stack and queue operations, it can be used as both. The Dequeue data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Dequeue. For example see Maximum of all subarrays of size k problem..

Another example of A-Steal job scheduling algorithm where Deque is used as deletions operation is required at both ends.