Introduction to Algorithm

An algorithm is a step by step procedure to solve a problem. Every algorithm must satisfy the following specifications...

- Input - Every algorithm must take zero or more number of input values from external.
- Output - Every algorithm must produce an output as result.
- Definiteness - Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation)
- Finiteness - For all different cases, the algorithm must produce result within a finite number of steps.
- Effectiveness - Every instruction must be basic enough to be carried out and it also must be feasible.

Performance Analysis

Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

- Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space
- Time required to complete the task of that algorithm (Time Complexity)

Space Complexity

Space complexity of an algorithm can be defined as the total amount of computer memory required by an algorithm to complete its execution

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

- Instruction Space: It is the amount of memory used to store compiled version of instructions.

- Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.
- Data Space: It is the amount of memory used to store all the variables and constants.

Consider the code

```
int square(int a)
{
        return a*a;
}
```

in above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.That means, totally it requires 4 bytes of memory to complete its execution.

Time Complexity

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution. Generally, running time of an algorithm depends upon the following...

- Whether it is running on Single processor machine or Multi processor machine.
- Whether it is a 32 bit machine or 64 bit machine
- Read and Write speed of the machine.
- The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
- Input data

Consider the following piece of code...

```
int sum(int a, int b)
{
  return a+b;
}
```

In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution.

Asymptotic notation

Asymptotic notation of an algorithm is a mathematical representation of its complexity

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

Big - Oh (O)
Big - Omega (Ω)
Big - Theta (Θ)

Big - Oh Notation (O)

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.

That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).

f(n) = O(g(n))

Data structure

Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy. Based on the organizing method of a data structure, data structures are divided into two types.

- Linear Data Structures
- Non - Linear Data Structures

Linear Data Structures

If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure. Example Arrays,List (Linked List),Stack,Queue

Non - Linear Data Structures

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure. Example Tree,Graph,Dictionaries,Heaps,Tries, Etc.,

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion

Traversing- It is used to access each data item exactly once so that it can be processed.

Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

Inserting- It is used to add a new data item in the given collection of data items.

Deleting- It is used to delete an existing data item from the given collection of data items.

Abstract Datatype

ADT is a set of objects together with a set of operations. “Abstract” in that implementation of in ADT definition .

E.g., List

Insert, delete, search, sort

C++ classes are perfect for ADTS

Arrays

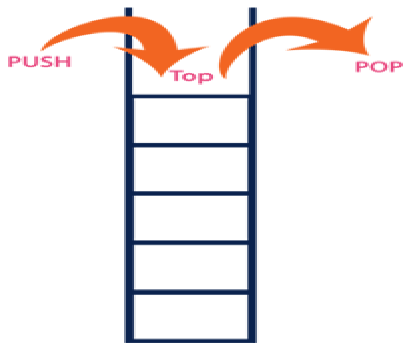
An array is a variable which can store multiple values of same data type at a time.

The individual elements of an array are identified using the combination of 'name' and 'index' as follows...

arrayName[indexValue]

STACK

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end called top of the stack. 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".

Operations on a Stack

The following operations are performed on the stack...

- Push (To insert an element on to the stack)
- Pop (To delete an element from the stack)
- Display (To display elements of the stack)

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

- Using Array
- Using Linked List

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

Stack Operations using Array

Steps to create an empty stack.

Step 1 : Create a one dimensional array with fixed size (int stack[SIZE])

Step 2: Define a integer variable 'top' and initialize with '-1'. (int top = -1)

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" 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" 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'.

<u>Expression</u>

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

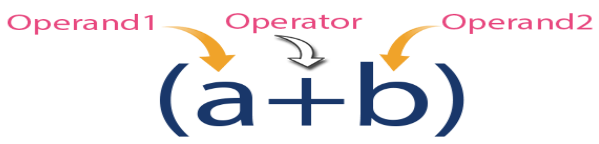
Infix Expression

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows..

Operand1 Operator Operand2

Example

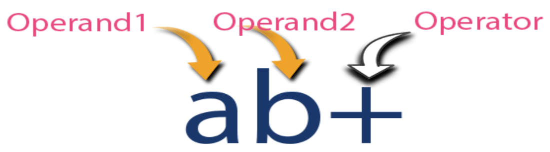


Postfix Expression

In postfix expression, operator is used after operands.The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

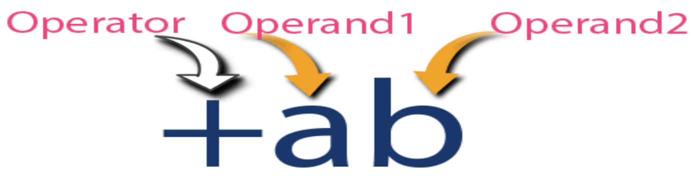
Example



Prefix Expression

In prefix expression, operator is used before operands. The general structure of Prefix expression is as follows...

Example



Applications of stack

- infix to postfix conversion
- postfix evaluation

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...

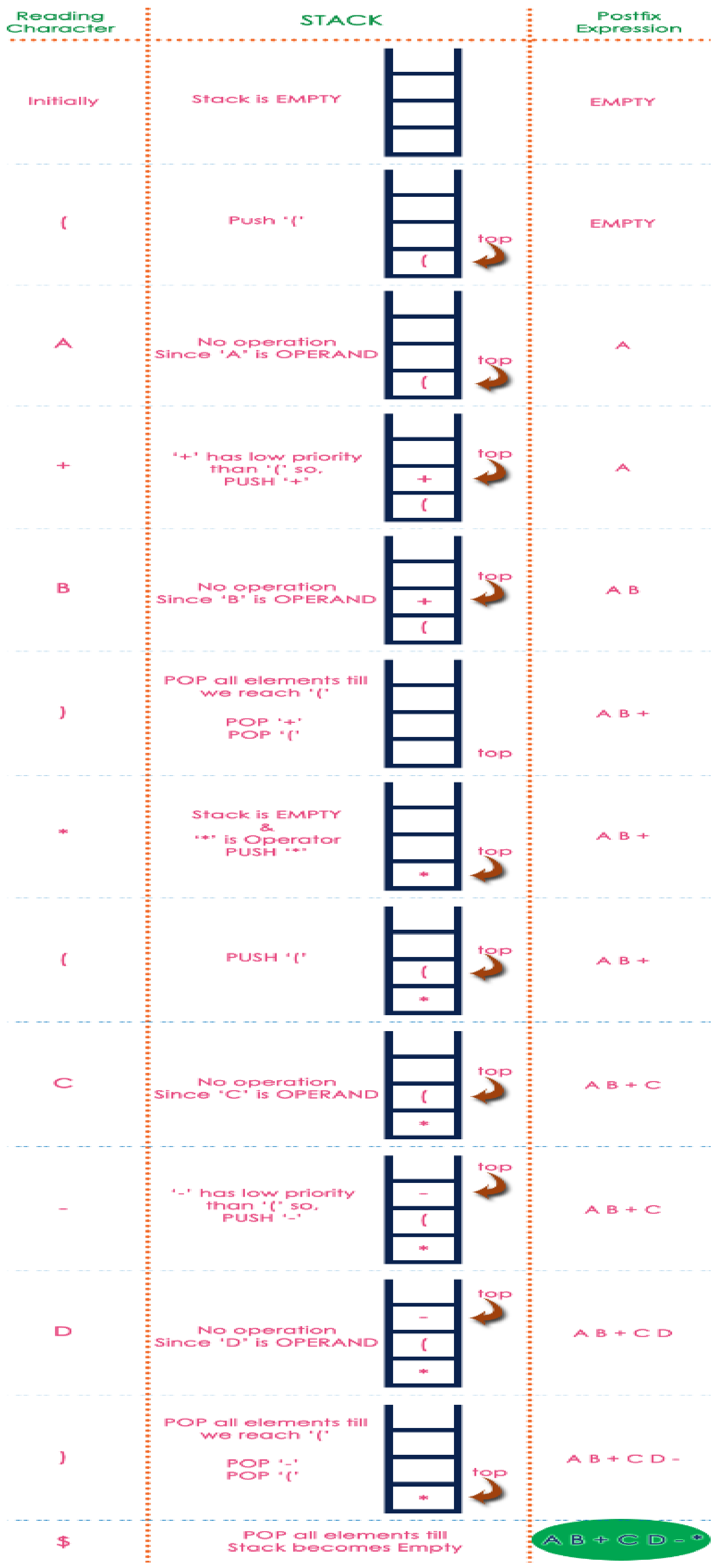
- Read all the symbols one by one from left to right in the given Infix Expression.
- If the reading symbol is operand, then directly print it to the result (Output).
- If the reading symbol is left parenthesis '(', then Push it on to the Stack.
- If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
- 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 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 - *

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...

- Read all the symbols one by one from left to right in the given Postfix Expression
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
- 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)

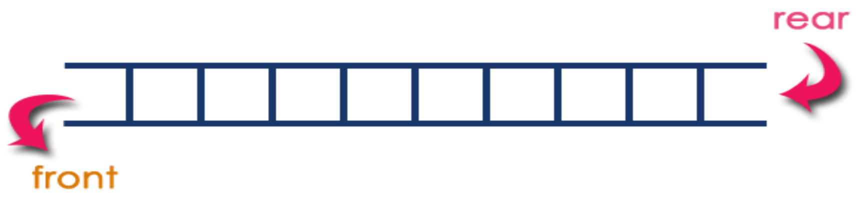
$ End of Expression	result = pop()		Display (result) **48** As final result

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

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

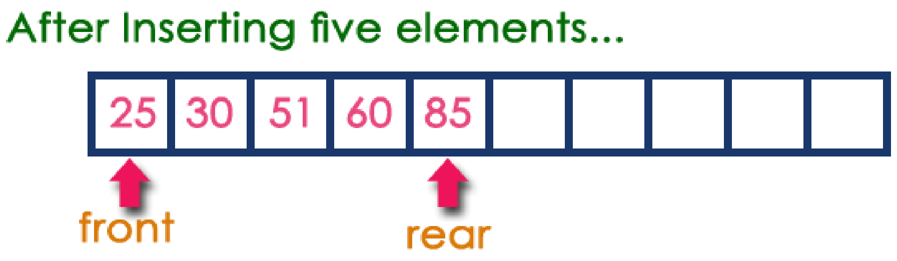
Queue ADT

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.In a queue data structure, 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 after inserting 25, 30, 51, 60 and 85.



Operations on a Queue

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

- enQueue(value) - (To insert an element into the queue)
- deQueue() - (To delete an element from the queue)
- display() - (To display the elements of the queue)

Queue data structure can be implemented in two ways. They are as follows...

- Using Array

- 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

Queue Operations using Array

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

- Create a one dimensional array with above defined SIZE (int queue[SIZE])
- Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

.**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!!!" 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.

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 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

We can use the following 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.

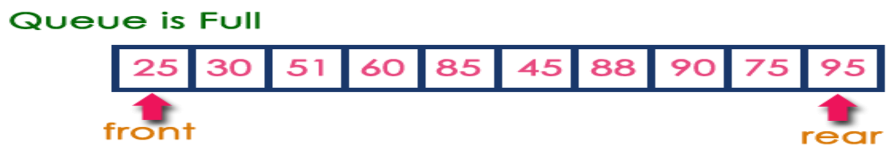
Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.

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)

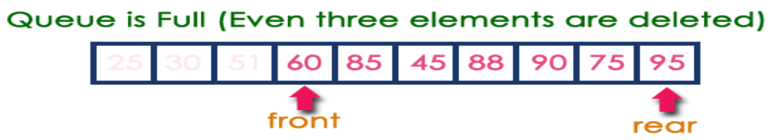
Circular Queue

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.



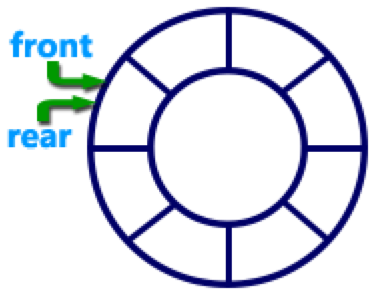
Now consider the following situation after deleting three elements from the queue...



this situation also says that Queue is Full and we cannot insert the new element because, 'rear' is still at last position. In above situation, 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**.**

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.

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

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.

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!!!" 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.

Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" 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.

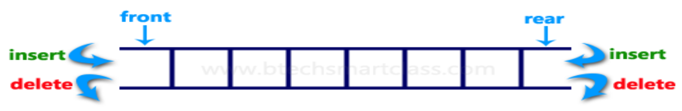
Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.

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.

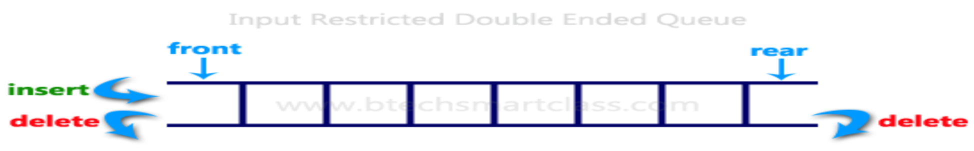


Double Ended Queue can be represented in TWO ways, those are as follows...

- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue

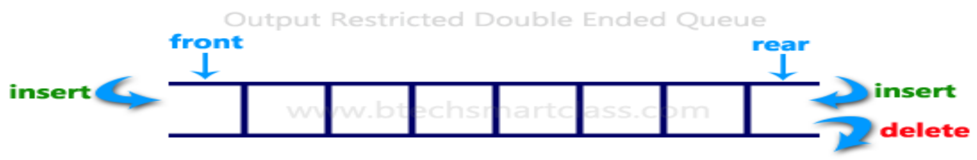
Input Restricted Double Ended Queue

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.



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...

- Max Priority Queue
- Min Priority Queue

1. Max Priority 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.

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

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

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...

- isEmpty() - Check whether queue is Empty.
- insert() - Inserts a new value into the queue.
- findMin() - Find minimum value in the queue.
- remove() - Delete minimum value from the queue.

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

MODULE II

The LIST ADT

• A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once.

A sequence of zero or more elements A1 , A2 , A3 , … AN

N: length of the list

A1 : first element

AN : last element

Advantage:

Very fast retrieval

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.