



KTU NOTES

The learning companion.

**KTU STUDY MATERIALS | SYLLABUS | LIVE
NOTIFICATIONS | SOLVED QUESTION PAPERS**

Website: www.ktunotes.in

DATA STRUCTURES

–ITT 201 (S3 IT)

MODULE -3

	Module 3: Stacks and Queues	9 hrs
3.1	Stack: Definition, Schematic Diagram of stack, Array and Linked list representation of stack , operations on stack using array and linked list (PUSH(),POP(),STATUS()) .	2
3.2	Applications of stacks: Infix to postfix conversion- post fix evaluation, string reversal, delimiter matching.	3
3.3	Queues: Definition, Schematic Diagram of queue, Array and Linked list representation of queue , operations on queue using array and linked list (ENQUEUE(),DEQUEUE(),STATUS()) .	2
3.4	Types of queue : circular queue-priority queue- doubly ended queue	2

Table of Contents

Stack	2
Stack examples	3
Stack operations.....	3
Stack using array	3
Creating stack	3
push(value) - Inserting value into the stack.....	4
pop() - Delete a value from the Stack.....	4
display() - Displays the elements of a Stack.....	4
Stack Underflow	4
Stack overflow	5
Stack Using Linked List	5
Example	6
Stack Operations using Linked List.....	6
push(value) - Inserting an element into the Stack.....	6
pop() - Deleting an Element from a Stack	7
display() - Displaying stack of elements	7
Is empty() and Isfull()	7
Expression Types.....	7
Infix Expression.....	8
Postfix Expression	8
Prefix Expression	8
Infix to Postfix Conversion using Stack Data Structure	8
Example.....	9
Example 2.....	9
Postfix evaluation.....	9
Reversal using stack	10
Parenthesis Matching using Stack	10
Queue.....	11
What is a Queue?.....	11
Example.....	11
Operations on a Queue	12
Queue Operations using Array.....	12
enqueue(value) - Inserting value into the queue	12
dequeue() - Deleting a value from the Queue.....	12

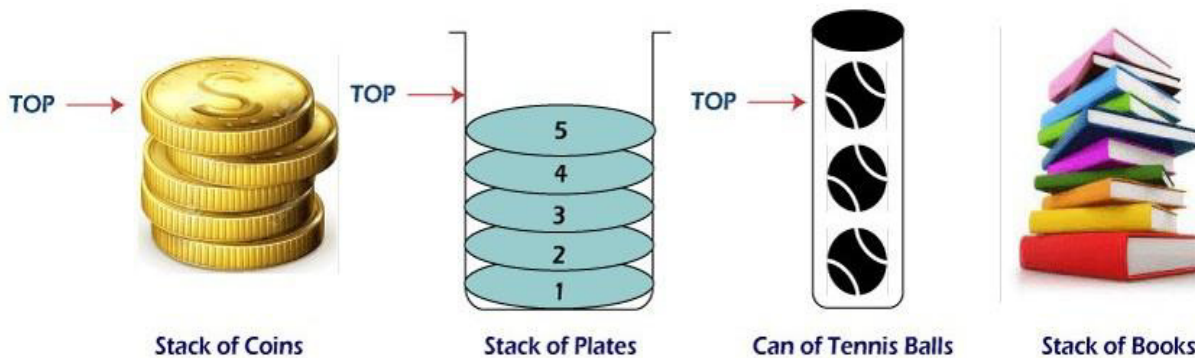
display() - Displays the elements of a Queue.....	13
Queue Using Linked List.....	13
Example.....	13
Operations	13
enQueue(value) - Inserting an element into the Queue	14
deQueue() - Deleting an Element from Queue	14
display() - Displaying the elements of Queue	14
Circular Queue Datastructure.....	15
What is Circular Queue?.....	15
Implementation of Circular Queue	16
enQueue(value) - Inserting value into the Circular Queue	16
deQueue() - Deleting a value from the Circular Queue	16
display() - Displays the elements of a Circular Queue	17
Double Ended Queue.....	17
Input Restricted Double Ended Queue.....	18
Output Restricted Double Ended Queue	18
Priority queue	18

Ktunotes.in

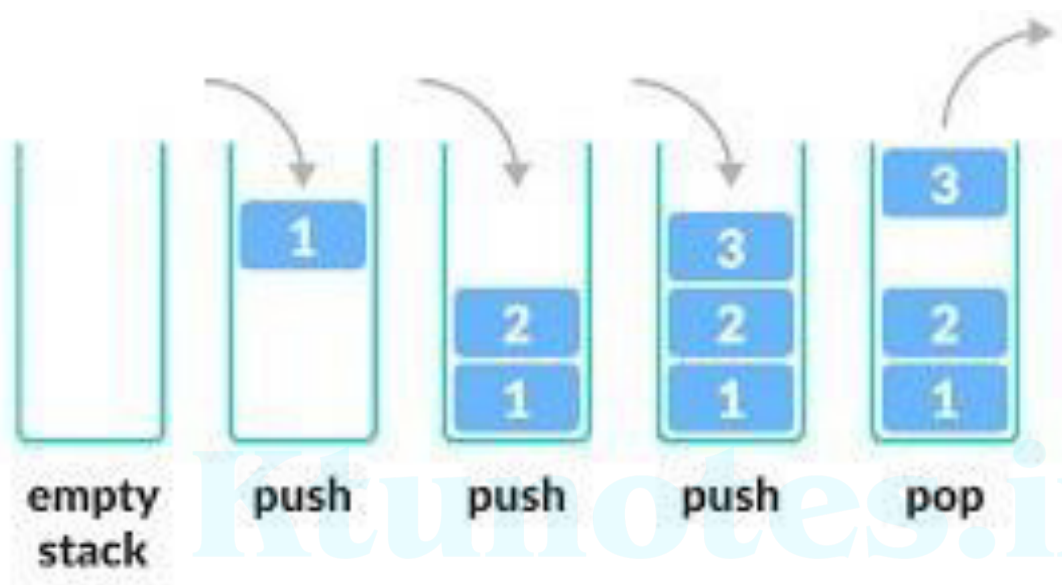
Stack

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. It is a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. The order in which elements come off a stack gives rise to its alternative name, LIFO (last in, first out). Additionally, a peek operation may give access to the top without modifying the stack.

Stack examples



Stack operations



- Insertion in Stack - Push
- Deletion in Stack - Pop

Stack using array

Stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a 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 called '**top**'. Initially, the 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.

Creating 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--**).

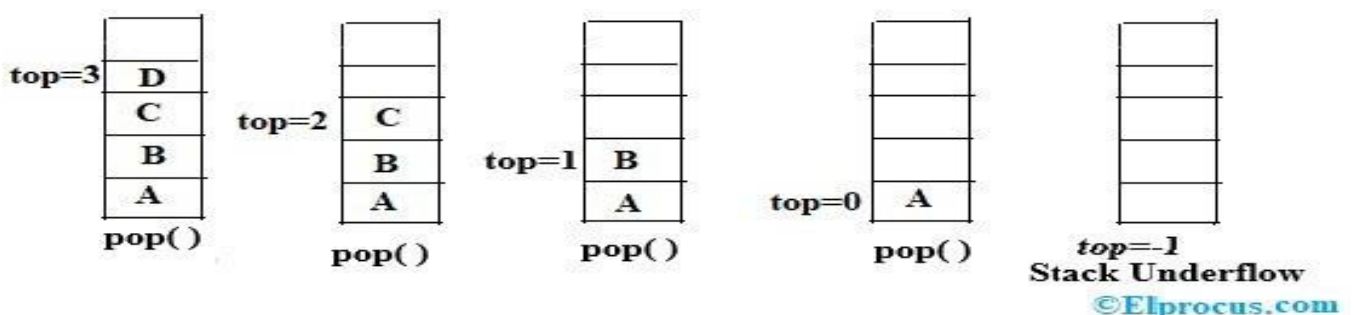
display() - Displays the elements of a Stack

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'.

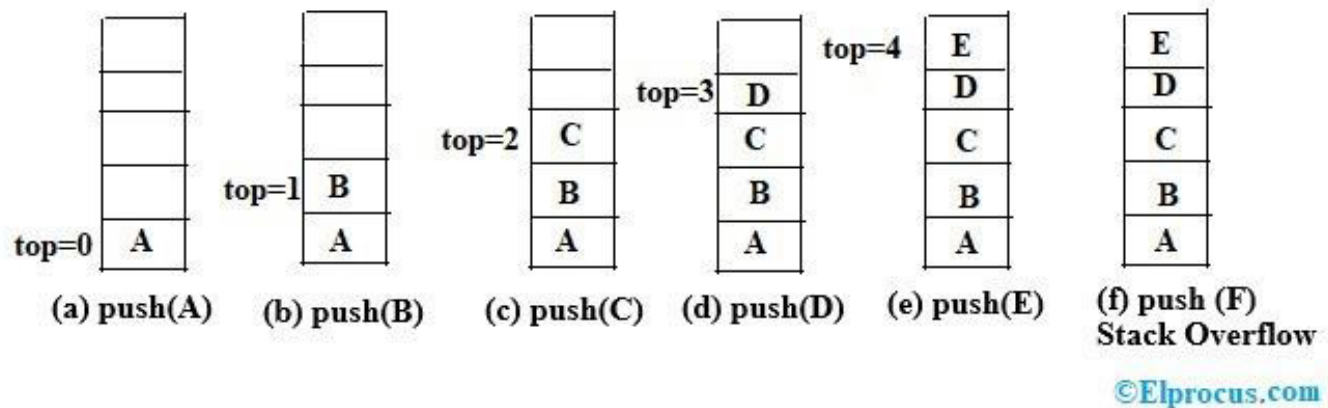
Stack Underflow

Stack Underflow happens when we try to pop an item from a stack that is empty



Stack overflow

Stack overflow happens when we try to pop an item from a stack that is already full.

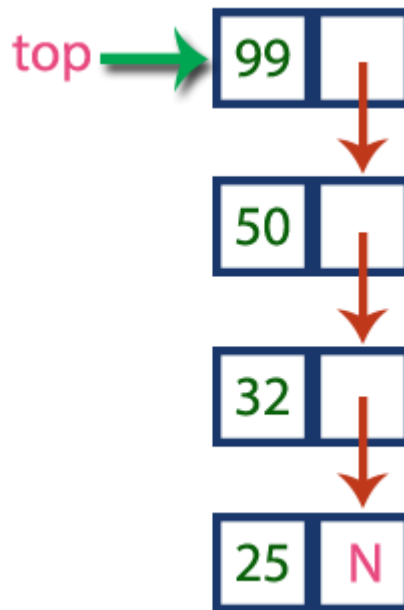


Stack Using Linked List

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an 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 a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the 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 previous node in the list. The **next** field of the first element must be always **NULL**.

Example



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

Stack Operations using Linked List

To implement a stack using a 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 5** - 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 5** - Finally! Display '**temp → data ---> NULL**'.

Is empty() and Isfull()

top is used to point to the top most element in the stack .

Top= 1, indicates that the stack is Empty

top=n-1, indicates that a stack of size n is Full

<pre>int isEmpty() { if(top == -1) return 1; else return 0; }</pre>	<pre>int isFull() { if(top == n-1) return 1; else return 0; }</pre>
--	---

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 the 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. We can say that "Operator follows the Operands".

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

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 as follows...

Operator Operand1 Operand2

Example



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.

1. If the reading symbol is operand, then directly print it to the result (Output).
2. If the reading symbol is left parenthesis '(', then Push it on to the Stack.

3. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
4. 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

	RPN	Stack	Input Expression		RPN	Stack	Input Expression
1			A+(B*(C-D)/E)	9	ABC	(* (D)/E)
2	A		+(B*(C-D)/E)	10	ABCD	(* (+) /E)
3	A	+	(B*(C-D)/E)	11	ABCD-	* (+	/E)
4	A	(+	B*(C-D)/E)	12	ABCD-*	/ (+	E)
5	AB	(+	*(C-D)/E)	13	ABCD-*E	/ (+)
6	AB	* (+	(C-D)/E)	14	ABCD-*E/	+	
7	AB	(* (+	C-D)/E)	15	ABCD-*E/+		
8	ABC	(* (+	-D)/E)				

Exa
Sup

2	*	23
/	/	23*
(/(23*
2	/(23*2
-	/(23*2
1	/(23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	++	23*21-/53
3	++	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Postfix evaluation

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 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.

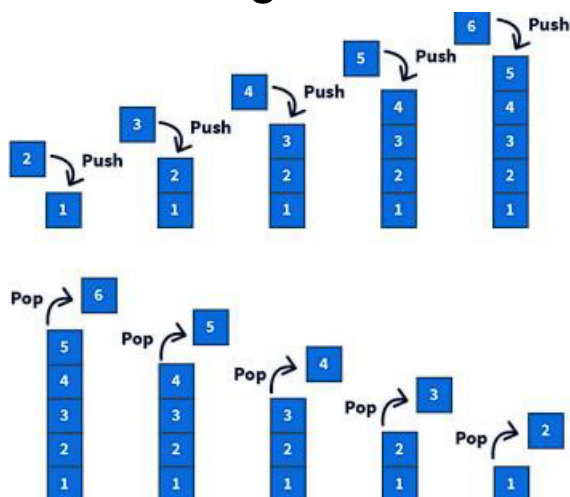
4. Finally! perform a pop operation and display the popped value as final result.

Example:

Postfix Expression: 6 2 3 + - 3 8 2 / + * 2 \$ 3 +

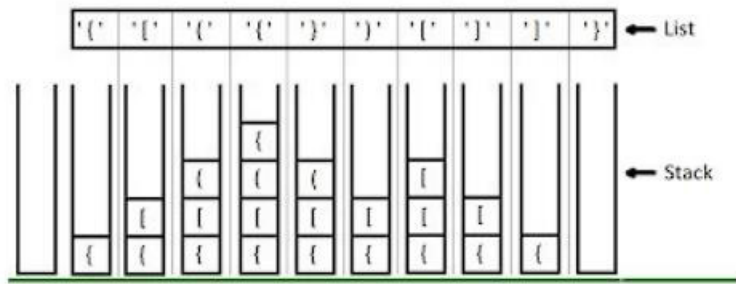
symb	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
\$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Reversal using stack



- Push all characters of string from left to right to the stack
- Pop all the items from the stack
- You will get the String as reversed

Parenthesis Matching using Stack



Push all the opening brackets in the stack.

Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature. If so, Pop the stack and continue the iteration

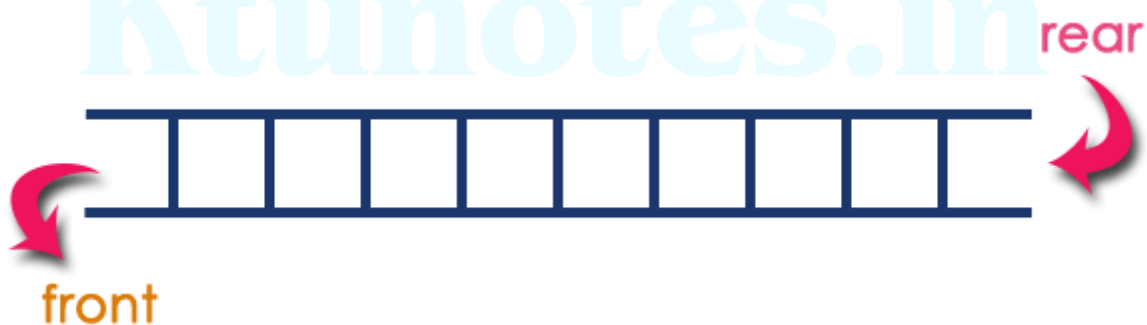
If not, then brackets are not well-ordered

in the end if the stack is empty, it means all brackets are well-formed

Queue

What is a 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 elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another 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 data structure 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.

After Inserting five elements...



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

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 a parameter and inserts that value into the queue. We can use the following 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. We can use the following 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

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+1**'.
- **Step 4** - Display '**queue[i]**' value and increment 'i' value by one (**i++**). Repeat the same until 'i' value reaches to **rear** (**i <= rear**)

Queue Using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an 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 a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the 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 5** - Finally! Display '**temp → data ---> NULL**'.

Circular Queue Datastructure

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the 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...

The queue after inserting all the elements into it is as follows...

Queue is Full



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

Queue is Full (Even three elements are deleted)



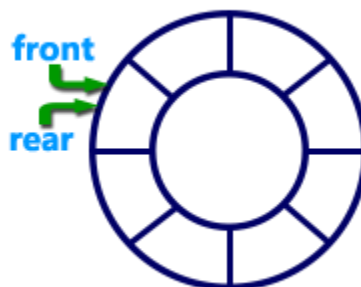
This situation also says that Queue is Full and we cannot insert the new element because 'rear' is still at last position. In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element. This is the major problem in a normal queue data structure.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

What is Circular Queue?

A Circular Queue can be defined as follows...

A 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 an 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 a 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**.

Double Ended Queue

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.

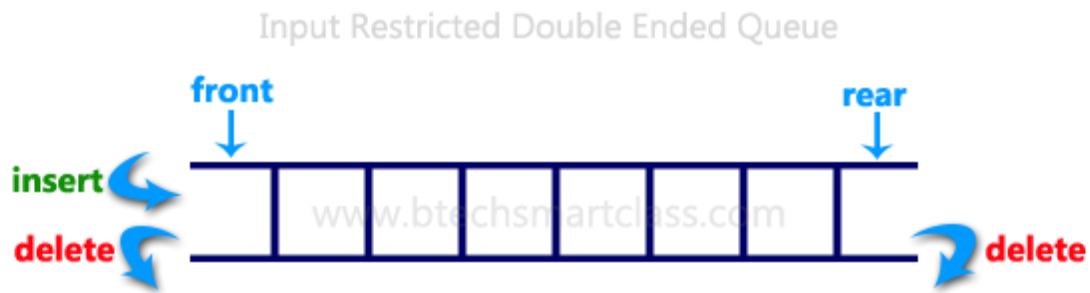


Double Ended Queue can be represented in TWO ways, those are as follows...

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

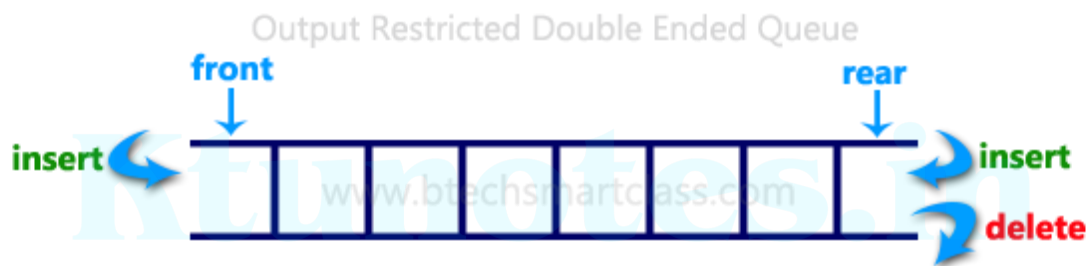
Input Restricted Double Ended Queue

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.



Priority queue

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.
- In the below priority queue, an element with a maximum ASCII value will have the highest priority. The elements with higher priority are served first.

In a priority queue, generally, the value of an element is considered for assigning the priority.

For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.