UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES

Stack ADT – Operations – Applications – Evaluating arithmetic expressions- Conversion of Infix to postfix expression – Queue ADT – Operations – Circular Queue – Priority Queue – deQueue –applications of queues.

STACK ADT

Stack is an abstract data type and it is also called linear data structure. It follows last in, first out (LIFO) strategy. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO.

The stack operations are given below.

Stack() - creates a new stack that is empty.
push(item) - adds a new item to the top of the stack.
pop() - removes the top item from the stack.
peek() - returns the top item from the stack but does not remove it.
isEmpty() - tests to see whether the stack is empty. It returns a boolean value.
size() - returns the number of items on the stack.

Stack using Array:

- **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

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'.

Program:

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10

void push(int);
void pop();
void display();

int stack[SIZE], top = -1;

void main()
{
  int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d",&value);
                    push(value);
                    break;
            case 2: pop();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void push(int value){
  if(top == SIZE-1)
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else{
    top++;
    stack[top] = value;
    printf("\nInsertion success!!!");
  }
}
void pop(){
  if(top == -1)
    printf("\nStack is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", stack[top]);
    top--;
  }
}
void display(){
  if(top == -1)
    printf("\nStack is Empty!!!");
  else{
    int i;
    printf("\nStack elements are:\n");
    for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
  }
}
```

Stack using Linked List

- The major problem with the stack implemented using array is, 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**.

OPERATIONS

Step 1: Define a '**Node**' structure with two member's **data** and **next**.
Step 2: Define a **Node** pointer '**top**' and set it to **NULL**.
Step 3: Implement the **main** function by displaying Menu with list of operations and make suitable function calls in the **main** function.

- **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**'.

Program:

```
#include<stdio.h>
#include<conio.h>

struct Node
{
  int data;
  struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();

void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Stack using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d", &value);
                    push(value);
                    break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}

void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   } }
```

APPLICATION ON STACK

i). Infix to Postfix Conversion

To convert Infix Expression into Postfix Expression using a stack data structure, Read all the symbols one by one from left to right in the given Infix Expression.

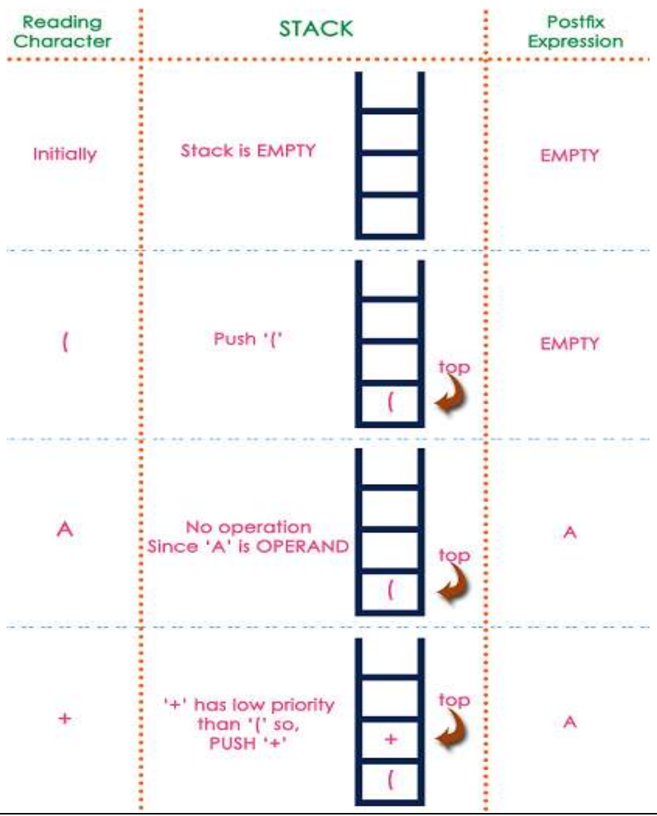
1. If the reading symbol is **operand, then directly print it to the result** (Output).
2. If the reading symbol is **left parenthesis '(', then Push it** on to the Stack.
3. If the reading symbol is **right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped 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.

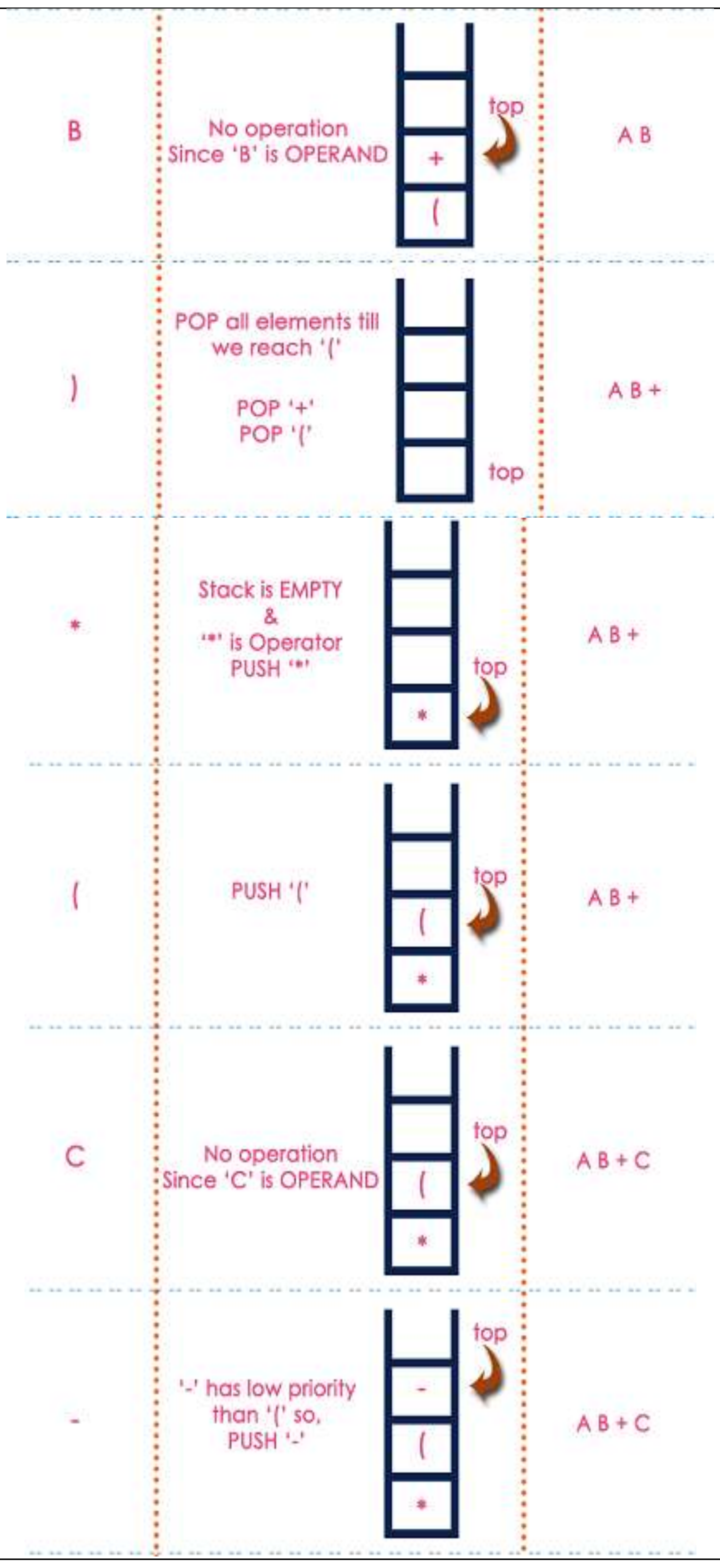
Example

Consider the following Infix Expression...

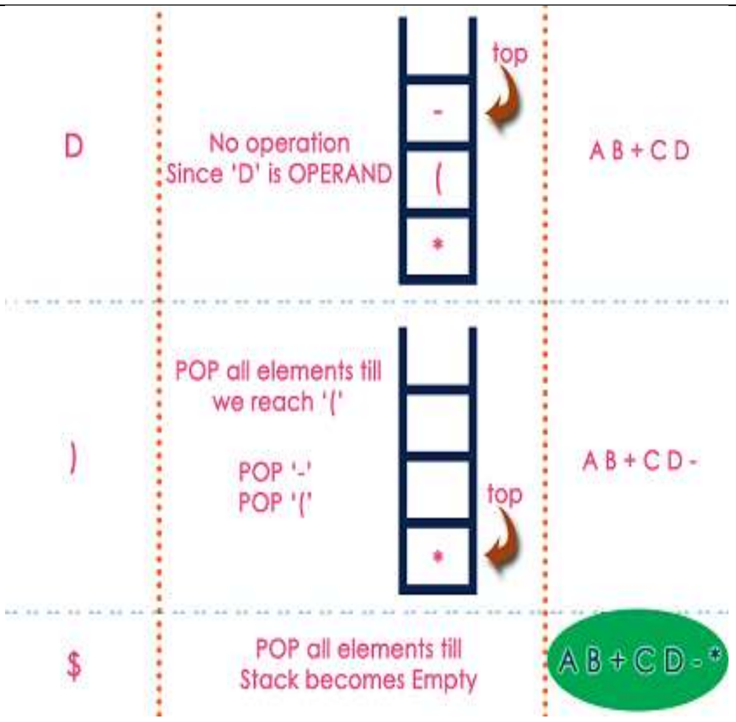
(A + B) * (C - D)

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





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

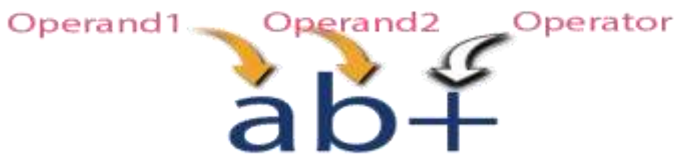


The final Postfix Expression is as follows...
A B + C D - *

Postfix Expression Evaluation

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

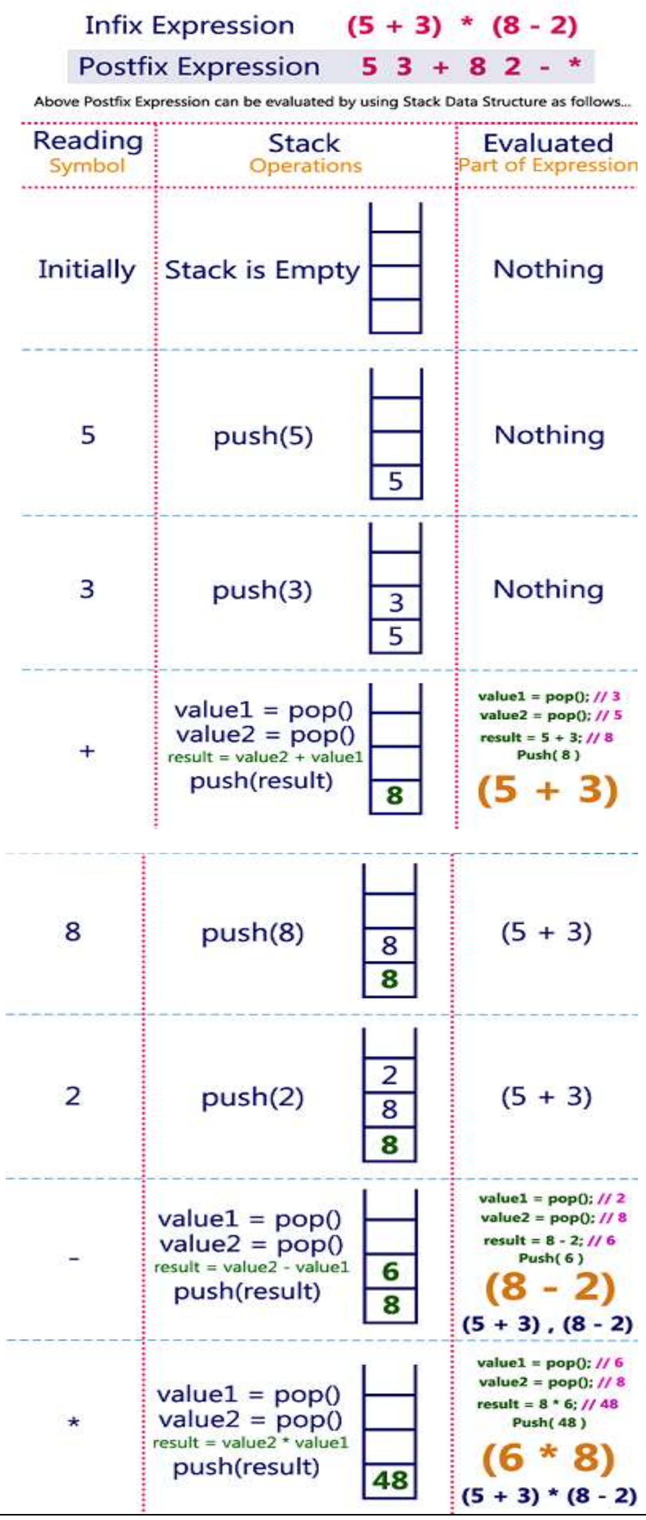
Postfix Expression has following general structure...



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

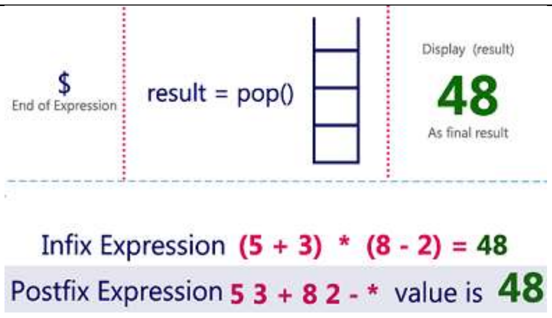


Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

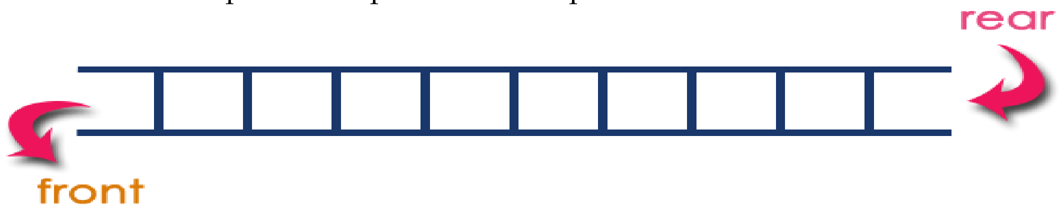
Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Stack	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

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle**.** In a queue data structure, adding and removing of elements are performed at two different positions. 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 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 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.

- **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. 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**.

- **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**).

- **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 3: Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value is equal to **rear** (**i <= rear**)

Program:
```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;

void main()
{
  int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
         case 1: printf("Enter the value to be insert: ");
                  scanf("%d",&value);
                  enQueue(value);
                  break;
         case 2: deQueue();
                  break;
         case 3: display();
                  break;
         case 4: exit(0);
```

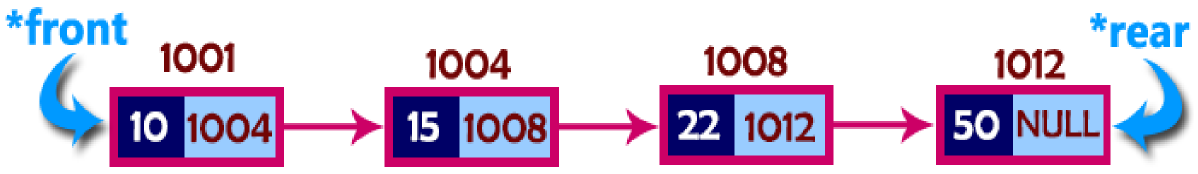
```
          default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
void enQueue(int value){
   if(rear == SIZE-1)
      printf("\nQueue is Full!!! Insertion is not possible!!!");
   else{
      if(front == -1)
          front = 0;
      rear++;
      queue[rear] = value;
      printf("\nInsertion success!!!");
   }
}
void deQueue(){
   if(front == rear)
      printf("\nQueue is Empty!!! Deletion is not possible!!!");
   else{
      printf("\nDeleted : %d", queue[front]);
      front++;
      if(front == rear)
          front = rear = -1;
   }
}
void display(){
   if(rear == -1)
      printf("\nQueue is Empty!!!");
   else{
      int i;
      printf("\nQueue elements are:\n");
      for(i=front; i<=rear; i++)
          printf("%d\t",queue[i]);
   }
}
```

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.

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**'.

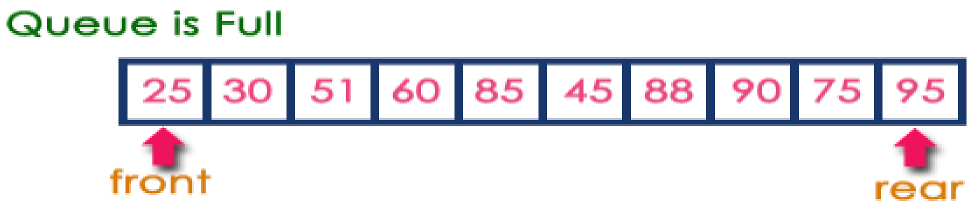
Program:

```
#include<stdio.h>
#include<conio.h>

struct Node
{
  int data;
  struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Queue Implementation using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
          case 1: printf("Enter the value to be insert: ");
                    scanf("%d", &value);
                    insert(value);
                    break;
          case 2: delete(); break;
          case 3: display(); break;
          case 4: exit(0);
          default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
void delete()
{
  if(front == NULL)
```

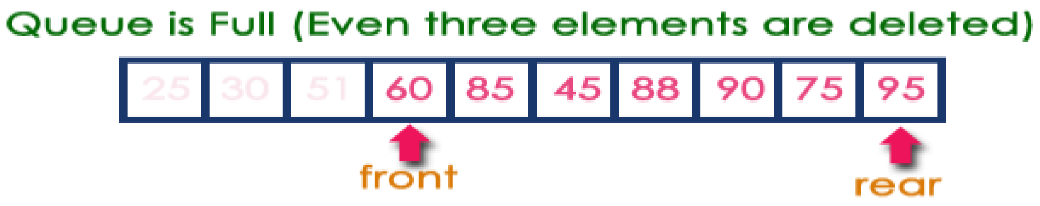
```
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
void display()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
      }
      printf("%d--->NULL\n",temp->data);
   }
}
```

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.

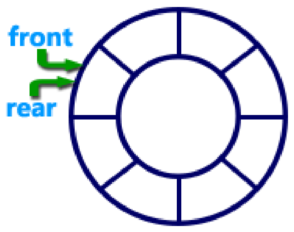


Now consider the following situation after deleting three elements from the queue...



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.

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...



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**

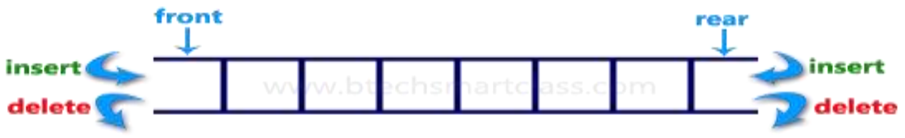
We can use the following steps to display the elements of a circular queue...
Step 1: Check whether **queue** is **EMPTY**. (**front == -1**)
Step 2: If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
Step 3: If it is **NOT EMPTY**, then define an integer variable '**i**' 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**.

Program:

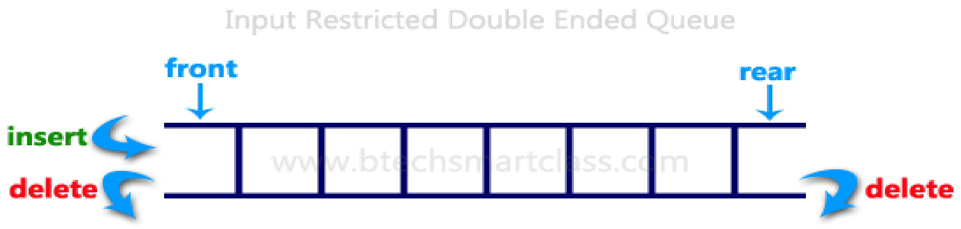
```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();
int cQueue[SIZE], front = -1, rear = -1;
void main()
{
  int choice, value;
  clrscr();
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
         case 1: printf("\nEnter the value to be insert:  ");
                  scanf("%d",&value);
                  enQueue(value);
                  break;
         case 2: deQueue();
                  break;
         case 3: display();
                  break;
         case 4: exit(0);
         default: printf("\nPlease select the correct choice!!!\n");
    }
  }
}
void enQueue(int value)
{
  if((front == 0 && rear == SIZE - 1) || (front == rear+1))
    printf("\nCircular Queue is Full! Insertion not possible!!!\n");
  else{
    if(rear == SIZE-1 && front != 0)
          rear = -1;
    cQueue[++rear] = value;
    printf("\nInsertion Success!!!\n");
    if(front == -1)
          front = 0;
  }
}
void deQueue()
{
  if(front == -1 && rear == -1)
    printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
  else{
    printf("\nDeleted element : %d\n",cQueue[front++]);
    if(front == SIZE)
          front = 0;
    if(front-1 == rear)
          front = rear = -1;
  }
}
void display()
{
  if(front == -1)
    printf("\nCircular Queue is Empty!!!\n");
  else{
    int i = front;
    printf("\nCircular Queue Elements are : \n");
    if(front <= rear){
          while(i <= rear)
            printf("%d\t",cQueue[i++]);
    }
    else{
          while(i <= SIZE - 1)
            printf("%d\t", cQueue[i++]);
          i = 0;
          while(i <= rear)
            printf("%d\t",cQueue[i++]);
    }
  }
}
```

❖ **Double Ended Queue (Dequeue)**

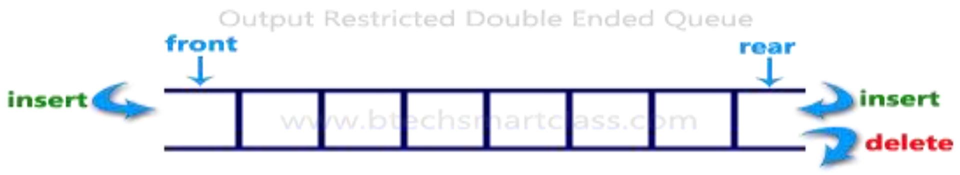
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



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.



Program

```
#include<stdio.h>
#include<conio.h>
#define SIZE 100
void enQueue(int);
int deQueueFront();
int deQueueRear();
void enQueueRear(int);
void enQueueFront(int);
void display();
```

```
int queue[SIZE];
int rear = 0, front = 0;

int main()
{
   char ch;
   int choice1, choice2, value;
   printf("\n******* Type of Double Ended Queue *******\n");
    do
    {
        printf("\n1.Input-restricted deque \n");
        printf("2.output-restricted deque \n");
        printf("\nEnter your choice of Queue Type : ");
        scanf("%d",&choice1);
        switch(choice1)
        {
            case 1:
                printf("\nSelect the Operation\n");
                printf("1.Insert\n2.Delete from Rear\n3.Delete from Front\n4. Display");
                do
                {
                  printf("\nEnter your choice for the operation in c deque: ");
                  scanf("%d",&choice2);
                  switch(choice2)
                  {
                    case 1: enQueueRear(value);
                           display();
                                       break;
                            case 2: value = deQueueRear();
                                       printf("\nThe value deleted is %d",value);
                           display();
                                       break;
                    case 3: value=deQueueFront();
                                  printf("\nThe value deleted is %d",value);
                           display();
                                  break;
                    case 4: display();
                           break;
                    default:printf("Wrong choice");
                  }
                  printf("\nDo you want to perform another operation (Y/N): ");
                  ch=getch();
                }while(ch=='y'||ch=='Y');
                getch();
                break;

            case 2 :
```

```
            printf("\n---- Select the Operation ----\n");
            printf("1. Insert at Rear\n2. Insert at Front\n3. Delete\n4. Display");
            do
            {
              printf("\nEnter your choice for the operation: ");
              scanf("%d",&choice2);
              switch(choice2)
              {
                case 1: enQueueRear(value);
                        display();
                        break;
                case 2: enQueueFront(value);
                        display();
                        break;
                case 3: value = deQueueFront();
                        printf("\nThe value deleted is %d",value);
                        display();
                        break;
                case 4: display();
                        break;
                default:printf("Wrong choice");
               }
               printf("\nDo you want to perform another operation (Y/N): ");
               ch=getch();
             } while(ch=='y'||ch=='Y');
             getch();
             break ;
        }
        printf("\nDo you want to continue(y/n):");
        ch=getch();
    }while(ch=='y'||ch=='Y');
}

void enQueueRear(int value)
{
   char ch;
   if(front == SIZE/2)
    {
        printf("\nQueue is full!!! Insertion is not possible!!! ");
        return;
    }
    do
    {
        printf("\nEnter the value to be inserted:");
        scanf("%d",&value);
        queue[front] = value;
        front++;
```

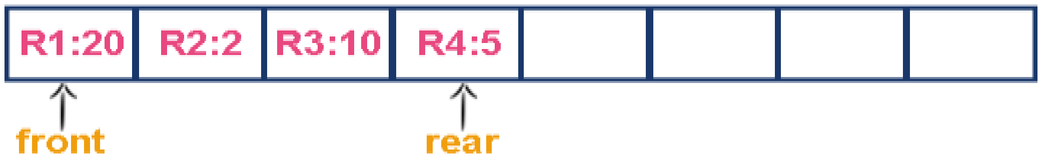
```
         printf("Do you want to continue insertion Y/N");
         ch=getch();
     }while(ch=='y');
}

void enQueueFront(int value)
{
    char ch;
    if(front==SIZE/2)
     {
         printf("\nQueue is full!!! Insertion is not possible!!!");
         return;
     }
     do
     {
         printf("\nEnter the value to be inserted:");
         scanf("%d",&value);
         rear--;
         queue[rear] = value;
         printf("Do you want to continue insertion Y/N");
         ch = getch();
     }
     while(ch == 'y');
}
int deQueueRear()
{
    int deleted;
    if(front == rear)
    {
         printf("\nQueue is Empty!!! Deletion is not possible!!!");
         return 0;
    }
    front--;
    deleted = queue[front+1];
    return deleted;
}
int deQueueFront()
{
    int deleted;
    if(front == rear)
    {
         printf("\nQueue is Empty!!! Deletion is not possible!!!");
         return 0;
    }
    rear++;
    deleted = queue[rear-1];
    return deleted;
```

```
}

void display()
{
   int i;
   if(front == rear)
     printf("\nQueue is Empty!!! Deletion is not possible!!!")
   else{
     printf("\nThe Queue elements are:");
     for(i=rear; i < front; i++)
     {
       printf("%d\t ",queue[i]);
     }
   }
}
```

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.
R1 : 20 units of time
R2 : 22 units of time (R2 must wait till R1 complete - 20 units and R2 itself requeres 2 units. Total 22 units)
R3 : 32 units of time (R3 must wait till R2 complete - 22 units and R3 itself requeres 10 units. Total 32 units)
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.
R2 : 2 units of time
R4 : 7 units of time (R4 must wait till R2 complete 2 units and R4 itself requeres 5 units. Total 7 units)
R3 : 17 units of time (R3 must wait till R4 complete 7 units and R3 itself requeres 10 units. Total 17 units)
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...
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...

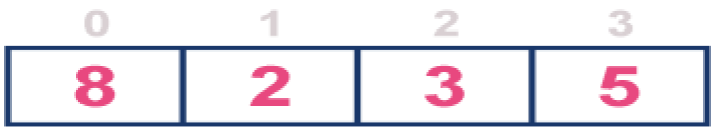
- isEmpty() - Check whether queue is Empty.
- insert() - Inserts a new value into the queue.
- findMax() - Find maximum value in the queue.
- remove() - Delete maximum value from the queue.

Max Priority Queue Representations
There are 6 representations of max priority queue.

- Using an Unordered Array (Dynamic Array)
- Using an Unordered Array (Dynamic Array) with the index of the maximum value
- Using an Array (Dynamic Array) in Decreasing Order
- Using an Array (Dynamic Array) in Increasing Order
- Using Linked List in Increasing Order
- 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.

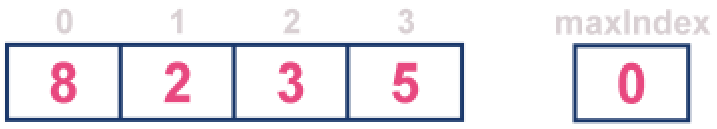


Now, let us analyse each operation according to this representation...
isEmpty() - If '**front == -1**' queue is Empty. This operation requires O(1) time complexity that means constant time.
insert() - New element is added at the end of the queue. This operation requires O(1) time complexity that means 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.

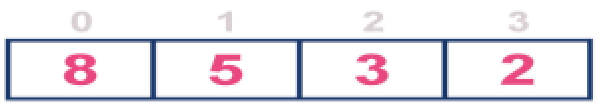


Now, let us analyse each operation according to this representation...
isEmpty() - If '**front == -1**' queue is Empty. This operation requires O(1) time complexity that means constant time.
insert() - New element is added at the end of the queue with **O(1)** and for each insertion we need to update 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.

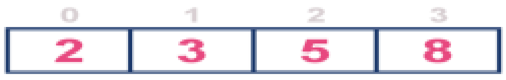


Now, let us analyse each operation according to this representation...
isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.
insert() - New element is added at a particular position in the decreasing order into the queue with **O(n)**, 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.

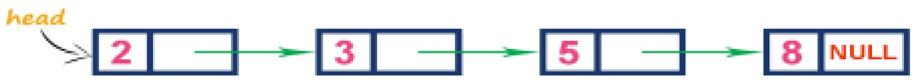


Now, let us analyse each operation according to this representation...
isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.
insert() - New element is added at a particular position in the increasing order into the queue with **O(n)**, 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 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.

#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.

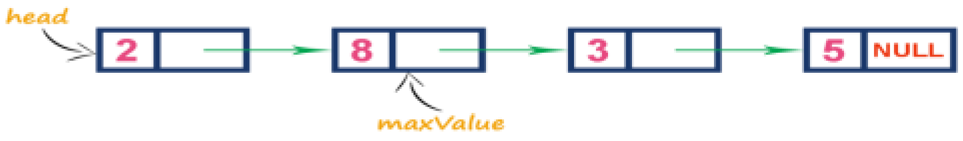
For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



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 requires **O(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.

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.