Chapter 3

The Queue

INTRODUCTION

Queue is an ordered list of items in which an item is inserted from one end called **REAR** of the queue and deleted from another end called **FRONT** of the queue.

- Queue is a linear data structure.
- **Front** points to the **beginning** of the queue and **Rear** points to the **end** of the queue.
- Queue follows the **FIFO (First - In - First Out)** structure.
- According to its FIFO structure, element inserted first will also be removed first.
- In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue), because queue is open at its both ends.
- The enqueue() and dequeue() are two basic functions used in a queue.

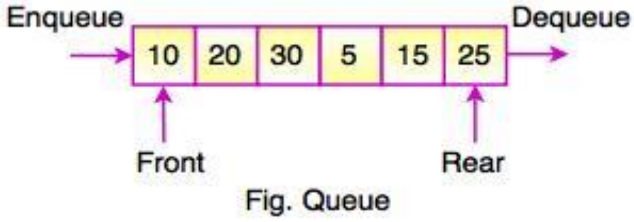


Fig. Queue

In a real-world analogy, we can imagine a bus queue where the passengers wait for the bus in a queue or a line. The first passenger in the line enters the bus first as that passenger happens to be the one who had come first.

PRIMITIVE OPERATIONS ON QUEUE

Following are the basic operations performed on a Queue.

- **EnQueue:** Adds an item to the queue. Addition of an item to the queue is always done at the rear of the queue.
- **DeQueue:** Removes an item from the queue. An item is removed or de-queued always from the front of the queue.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **peek:** Gets an element at the front of the queue without removing it.

Enqueue Operation

In this process, the following steps are performed:

- Check if the queue is full.
- If full, produce overflow error and exit.

- Else, add an element to the location pointed by 'rear'.
- Increment 'rear'.
- Return success.

Dequeue Operation

Dequeue operation consists of the following steps:

- Check if the queue is empty.
- If empty, display an underflow error and exit.
- Else, the access element is pointed out by 'front'.
- Increment the 'front' to point to the next accessible data.
- Return success.

Next, we will see a detailed illustration of insertion and deletion operations in queue.

QUEUE AS AN ADT

```
template <class KeyType>
class Queue
{
        //objects: a finite ordered list with zero or more elements.
        public:
          Queue(int MaxQueueSize=DefaultSize);
            //create an empty queue whose maximum size is MaxQueueSize
          Boolean IsFullQ();
            //if (number of elements in queue== MaxQueueSize) return TRUE
            //else return FALSE

          void Add(const Keytype& item);
            //if (IsFullQ()) then QueueFull()
            //else insert item at rear of queue

          Boolean IsEmptyQ();
            //if number of elements in the queue is equal to 0 then return TRUE
            //else return FALSE

          Keytype* Delete(KeyType&);
            //if (IsEmpty()) then QueueEmpty() and return 0
            //else remove the item at front of queue and return a pointer to it.

        ~Queue() {}
};
```

APPLICATIONS OF QUEUE

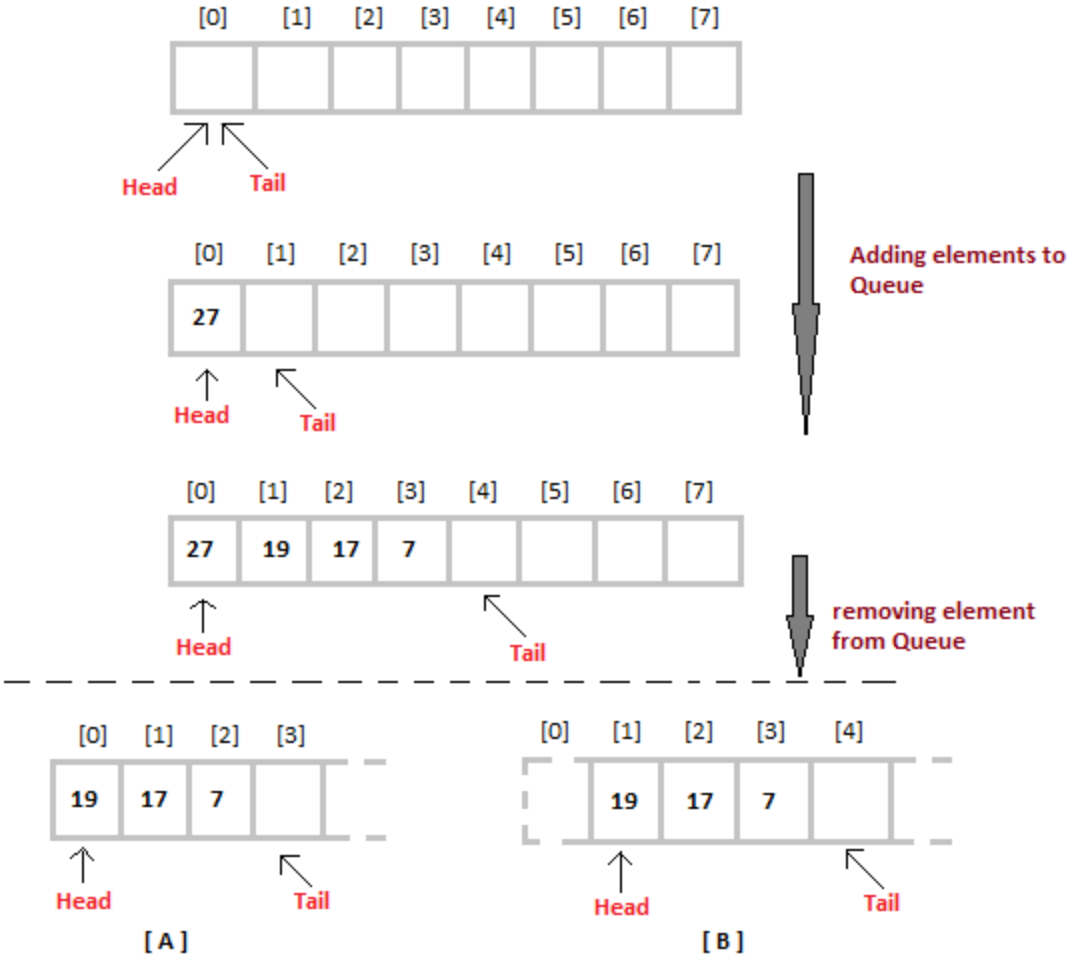
Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

IMPLEMENTATION OF QUEUE DATA STRUCTURE

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from `0`). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.
In approach [B] we remove the element from **head** position and then move **head** to the next position.
In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.
In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

/* Below program is written in C++ language */

#include<iostream>

using namespace std;

```
#define SIZE 10

class Queue
{
   int a[SIZE];
   int rear;   //same as tail
   int front;  //same as head

   public:
   Queue()
   {
      rear = front = -1;
   }

   //declaring enqueue, dequeue and display functions
   void enqueue(int x);
   int dequeue();
   void display();
};

// function enqueue - to add data to queue
void Queue :: enqueue(int x)
{
   if(front == -1) {
      front++;
   }
   if( rear == SIZE-1)
   {
      cout << "Queue is full";
   }
   else
   {
      a[++rear] = x;
   }
}

// function dequeue - to remove data from queue
int Queue :: dequeue()
{
   return a[++front];  // following approach [B], explained above
}

// function to display the queue elements
```

```
void Queue :: display()
{
   int i;
   for( i = front; i <= rear; i++)
   {
      cout << a[i] << endl;
   }
}

// the main function
int main()
{
   Queue q;
   q.enqueue(10);
   q.enqueue(100);
   q.enqueue(1000);
   q.enqueue(1001);
   q.enqueue(1002);
   q.dequeue();
   q.enqueue(1003);
   q.dequeue();
   q.dequeue();
   q.enqueue(1004);

   q.display();

   return 0;
}
```

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements by one position.

```
return a[0];    //returning first element
for (i = 0; i < tail-1; i++)    //shifting all other elements
{
   a[i] = a[i+1];
   tail--;
}
```

Drawback of Linear Queue

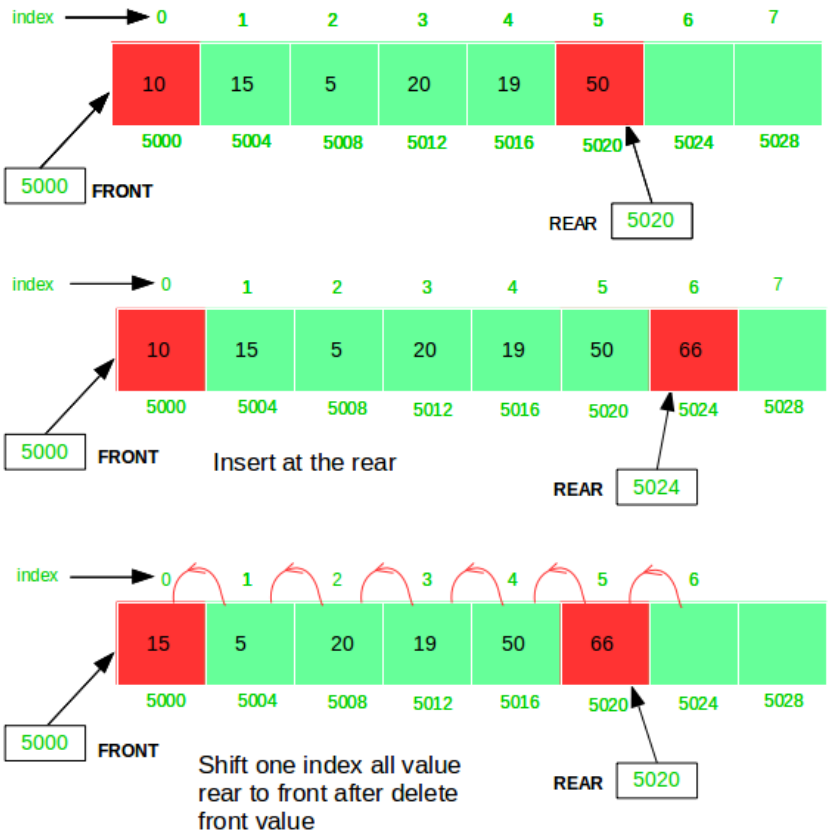
The linear queue suffers from serious drawback after performing some operations, we cannot insert items into queue, even if there is space in the queue. Suppose we have queue of 5 elements

and we insert 5 items into queue, and then delete some items, then queue has space, but at that condition we cannot insert items into queue.

LINEAR QUEUE AND ITS IMPLEMENTATION

To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to *0* which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

1. **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear < n* which indicates that the array is not full then store the element at *arr[rear]* and increment *rear* by *1* but if *rear == n* then it is said to be an Overflow condition as the array is full.
2. **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear > 0*. Now, element at *arr[front]* can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.
3. **Front:** Get the front element from the queue i.e. *arr[front]* if queue is not empty.
4. **Display:** Print all element of the queue. If the queue is non-empty, traverse and print all the elements from index *front* to *rear*.



C++ program to implement a Linear queue using an array

```
#include <iostream>
using namespace std;

struct Queue {
   int front, rear, capacity;
   int* queue;
   Queue(int c)
   {
      front = rear = 0;
      capacity = c;
      queue = new int;
   }

   ~Queue() { delete[] queue; }

   // function to insert an element
   // at the rear of the queue
   void queueEnqueue(int data)
   {
      // check queue is full or not
      if (capacity == rear) {
         printf("\nQueue is full\n");
         return;
      }

      // insert element at the rear
      else {
         queue[rear] = data;
         rear++;
      }
      return;
   }

   // function to delete an element
   // from the front of the queue
   void queueDequeue()
   {
      // if queue is empty
      if (front == rear) {
         printf("\nQueue is  empty\n");
         return;
      }

      // shift all the elements from index 2 till rear
      // to the left by one
      else {
```

```
        for (int i = 0; i < rear - 1; i++) {
            queue[i] = queue[i + 1];
        }

        // decrement rear
        rear--;
    }
    return;
}

// print queue elements
void queueDisplay()
{
    int i;
    if (front == rear) {
        printf("\nQueue is Empty\n");
        return;
    }

    // traverse front to rear and print elements
    for (i = front; i < rear; i++) {
        printf(" %d <-- ", queue[i]);
    }
    return;
}

// print front of queue
void queueFront()
{
    if (front == rear) {
        printf("\nQueue is Empty\n");
        return;
    }
    printf("\nFront Element is: %d", queue[front]);
    return;
}
};

int main(void)
{
    // Create a queue of capacity 4
    Queue q(4);

    // print Queue elements
    q.queueDisplay();

    // inserting elements in the queue
    q.queueEnqueue(20);
```

```
    q.queueEnqueue(30);
    q.queueEnqueue(40);
    q.queueEnqueue(50);

    // print Queue elements
    q.queueDisplay();

    // insert element in the queue
    q.queueEnqueue(60);

    // print Queue elements
    q.queueDisplay();

    q.queueDequeue();
    q.queueDequeue();

    printf("\n\nafter two node deletion\n\n");

    // print Queue elements
    q.queueDisplay();

    // print front of the queue
    q.queueFront();

    return 0;
}
```

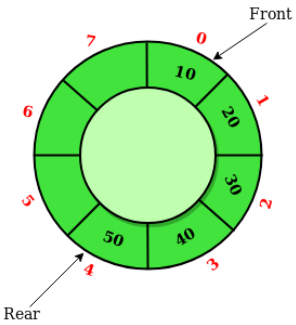
Time Complexity of Enqueue : O(1)
Time Complexity of Dequeue : O(n)

Optimizations:

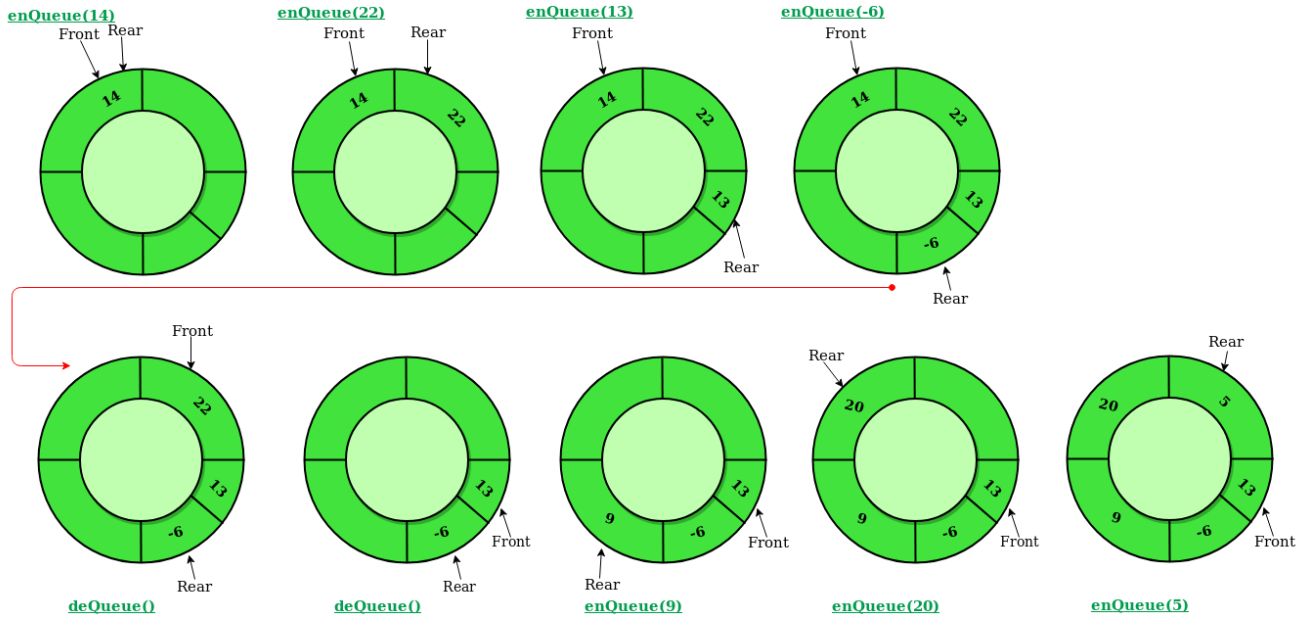
We can implement both enqueue and dequeue operations in O(1) time. To achieve this, we can either use Linked List Implementation of Queue or circular array implementation of queue.

CIRCULAR QUEUE AND ITS IMPLEMENTATION

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. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
  Steps:
  1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
  Steps:
  1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3

3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

C++ program to implement a Circular Queue using an Array

```
#include<iostream>

using namespace std;

#define SIZE 10

class CircularQueue
{
   int a[SIZE];
   int rear;   //same as tail
   int front;  //same as head

   public:
   CircularQueue()
   {
      rear = front = -1;
   }
   // function to check if queue is full
   bool isFull()
   {
      if(front == 0 && rear == SIZE - 1)
      {
         return true;
      }
      if(front == rear + 1)
      {
         return true;
      }
      return false;
   }

   // function to check if queue is empty
   bool isEmpty()
   {
      if(front == -1)
      {
         return true;
      }
      else
      {
         return false;
```

```
    }
  }

  //declaring enqueue, dequeue, display and size functions
  void enqueue(int x);
  int dequeue();
  void display();
  int size();
};

// function enqueue - to add data to queue
void CircularQueue :: enqueue(int x)
{
  if(isFull())
  {
    cout << "Queue is full";
  }
  else
  {
    if(front == -1)
    {
      front = 0;
    }
    rear = (rear + 1) % SIZE;   // going round and round concept
    // inserting the element
    a[rear] = x;
    cout << endl << "Inserted " << x << endl;
  }
}


// function dequeue - to remove data from queue
int CircularQueue :: dequeue()
{
  int y;

  if(isEmpty())
  {
    cout << "Queue is empty" << endl;
  }
  else
  {
    y = a[front];
    if(front == rear)
    {
      // only one element in queue, reset queue after removal
      front = -1;
      rear = -1;
```

```
      }
      else
      {
         front = (front+1) % SIZE;
      }
      return(y);
   }
}

void CircularQueue :: display()
{
   /* Function to display status of Circular Queue */
   int i;
   if(isEmpty())
   {
      cout << endl << "Empty Queue" << endl;
   }
   else
   {
      cout << endl << "Front -> " << front;
      cout << endl << "Elements -> ";
      for(i = front; i != rear; i= (i+1) % SIZE)
      {
         cout << a[i] << "\t";
      }
      cout << a[i];
      cout << endl << "Rear -> " << rear;
   }
}

int CircularQueue :: size()
{
   if(rear >= front)
   {
      return (rear - front) + 1;
   }
   else
   {
      return (SIZE - (front - rear) + 1);
   }
}

// the main function
int main()
{
   CircularQueue cq;
   cq.enqueue(10);
   cq.enqueue(100);
```

```
    cq.enqueue(1000);

    cout << endl << "Size of queue: " << cq.size();

    cout << endl << "Removed element: " << cq.dequeue();

    cq.display();

    return 0;
}
```

Time Complexity:

Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

Applications:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

IMPLEMENTATION OF CIRCULAR QUEUE ADT

Enqueue Operation

```
template <class KeyType>
void Queue<KeyType>::Add(const KeyType & x)
//add x to the circular queue
{
  int newrear = (rear+1)%MaxSize;
if (front == newrear) QueueFull();
  else queue[rear=newrear] = x;
}
```

Dequeue Operation

```
template <class KeyType>
```

```
KeyType* Queue<KeyType>::Delete(KeyType &x)
//Remove front element from queue
{
if (front== rear) { QueueEmpty(); return 0;}
  front= (front+ 1) % MaxSize;
  x = queue[front];
  return &x;
}
```

LINKED LIST IMPLEMENTATION OF QUEUE

```
// A CPP program to demonstrate linked list
// based implementation of queue
#include <bits/stdc++.h>
using namespace std;

// A linked list (LL) node to store a queue entry
class QNode {
public:
    int key;
    QNode* next;
};

// The queue, front stores the front node
// of LL and rear stores the last node of LL
class Queue {
public:
    QNode *front, *rear;
};

// A utility function to create
// a new linked list node.
QNode* newNode(int k)
{
    QNode* temp = new QNode();
    temp->key = k;
    temp->next = NULL;
    return temp;
}

// A utility function to create an empty queue
Queue* createQueue()
{
    Queue* q = new Queue();
    q->front = q->rear = NULL;
    return q;
```

```
}

// The function to add a key k to q
void enQueue(Queue* q, int k)
{
    // Create a new LL node
    QNode* temp = newNode(k);

    // If queue is empty, then
    // new node is front and rear both
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }

    // Add the new node at
    // the end of queue and change rear
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove
// a key from given queue q
QNode* deQueue(Queue* q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
        return NULL;

    // Store previous front and
    // move front one node ahead
    QNode* temp = q->front;
    delete(temp);

    q->front = q->front->next;

    // If front becomes NULL, then
    // change rear also as NULL
    if (q->front == NULL)
        q->rear = NULL;
    return temp;
}

// Driver code
int main()
{
    Queue* q = createQueue();
    enQueue(q, 10);
```

```
    enQueue(q, 20);
    deQueue(q);
    deQueue(q);
    enQueue(q, 30);
    enQueue(q, 40);
    enQueue(q, 50);
    QNode* n = deQueue(q);
    if (n != NULL)
        cout << "Dequeued item is " << n->key;
    return 0;
}
```

Output:

```
Dequeued item is 30
```

Time Complexity:

Time complexity of both operations enqueue() and dequeue() is O(1) as we only change few pointers in both operations. There is no loop in any of the operations.

TYPES OF QUEUE

Queue are of following types:

1. Linear queue
2. Circular queue
3. Priority queue
4. Dequeue (Double Ended Queue)

PRIORITY QUEUE

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority. For example: The element with the highest value is considered as the highest priority element. However, in other case, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priority according to our need.

Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

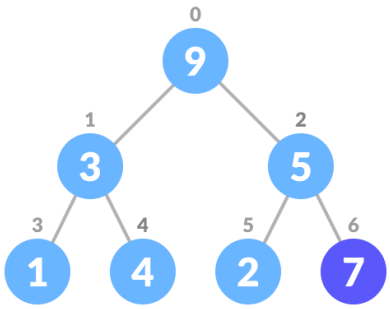
Priority Queue Operations

Basic operations of a priority queue are inserting, removing and peeking elements.

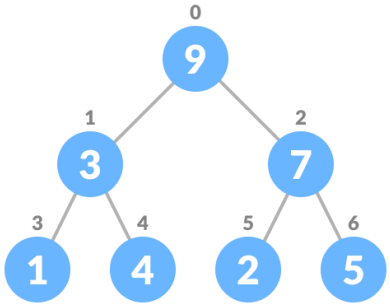
Inserting an Element from the Priority Queue

Inserting an element into a priority queue (max heap) is done by following steps.

1. Insert the new element at the end of the tree.



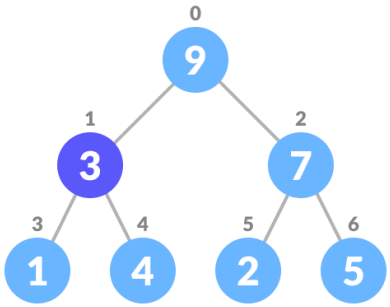
2. Heapify the tree.



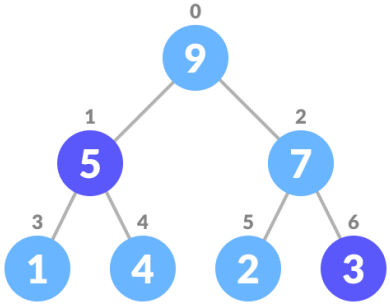
Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max heap) is done as follows:

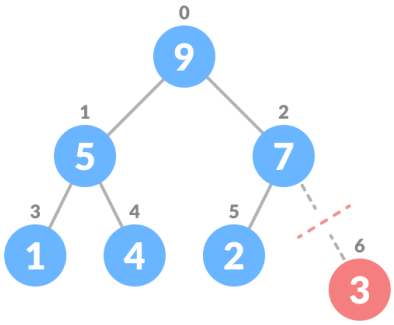
1. Select the element to be deleted.



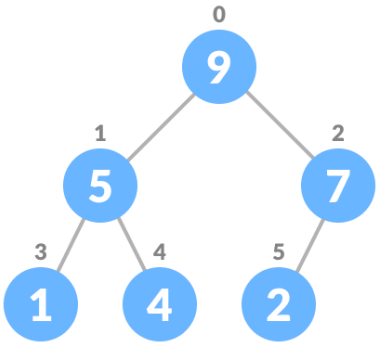
2. Swap it with the last element.



3. Remove the last element.

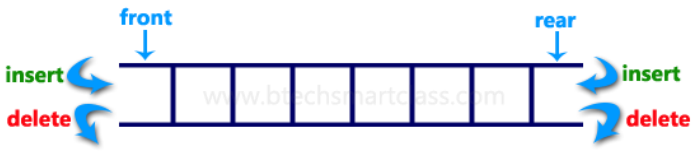


4. Heapify the tree.



DOUBLE ENDED QUEUE (DEQUE)

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.

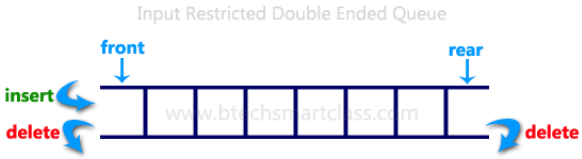


Double Ended Queue can be represented in TWO ways, those are as follows...

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.

Implementation of Double ended Queue

Here we will implement a double ended queue using a circular array. It will have the following methods:

1. **push_back :** inserts element at back
2. **push_front :** inserts element at front

3. **pop_back :** removes last element
4. **pop_front :** removes first element
5. **get_back :** returns last element
6. **get_front :** returns first element
7. **empty :** returns true if queue is empty
8. **full :** returns true if queue is full

Deque ADT

```
// Maximum size of array or Dequeue

#define SIZE 5

class Dequeue
{
    //front and rear to store the head and tail pointers
    int  *arr;
    int front, rear;

    public :

    Dequeue()
    {
      //Create the array
        arr = new int[SIZE];

        //Initialize front and rear with -1
        front = -1;
        rear = -1;
    }

    // Operations on Deque
    void  push_front(int );
    void  push_back(int );
    void  pop_front();
    void  pop_back();
    int  get_front();
    int  get_back();
    bool  full();
    bool  empty();
};
```

Implementation of Deque

Insertion at front

```
void Dequeue :: push_front(int key)
{
    if(full())
```

```
    {
        cout << "OVERFLOW\n";
    }
    else
    {
      //If queue is empty
      if(front == -1)
            front = rear = 0;

      //If front points to the first position element
         else if(front == 0)
             front = SIZE-1;

         else
            --front;

         arr[front] = key;
    }
}
```

Insertion at back

```
void Dequeue :: push_back(int key)
{
    if(full())
    {
        cout << "OVERFLOW\n";
    }
    else
    {
        //If queue is empty
        if(front == -1)
             front = rear = 0;

        //If rear points to the last element
        else if(rear == SIZE-1)
            rear = 0;

        else
           ++rear;

        arr[rear] = key;
    }
}
```

Delete from front (Delete first element)

```
void Dequeue :: pop_front()
{
    if(empty())
    {
        cout << "UNDERFLOW\n";
    }
    else
    {
```

```
      //If only one element is present
         if(front == rear)
            front = rear = -1;

         //If front points to the last element
         else if(front == SIZE-1)
            front = 0;

         else
            ++front;
    }
}
```

Delete from back (Delete last element)

```
void Dequeue :: pop_back()
{
    if(empty())
    {
        cout << "UNDERFLOW\n";
    }
    else
    {
      //If only one element is present
         if(front == rear)
            front = rear = -1;

         //If rear points to the first position element
         else if(rear == 0)
            rear = SIZE-1;

         else
            --rear;
    }
}
```

Check if deque is empty

```
bool Dequeue :: empty()
{
    if(front == -1)
      return true;
    else
      return false;
}
```

Check if deque is full

```
bool Dequeue :: full()
{
    if((front == 0 && rear == SIZE-1)  ||(front == rear + 1))
        return true;
    else
        return false;
}
```