Unit 3

Queues

	Syllabus	Guidelines	Suggested number of lectures
1	Array and linked representation of queue, deque comparison of the operations on queues in the two representations	Chapter 5, Section 5.2, 5.3 (upto 5.3.3) **Ref 2**	6
2	Applications of queues.	Chapter 6, Section 6.4.1, **Ref 1**	

References

1. Ref 1: . Drozdek, A., (2012), *Data Structures and algorithm in C++*. 3rd edition. Cengage Learning. Note: 4th edition is available. Ebook is 4th edition

2. Ref 2.: Goodrich, M., Tamassia, R., & Mount, D., (2011). *Data Structures and Algorithms Analysis in C+ +*. 2nd edition. Wiley.

3. Additional Resource 3: Sahni, S. (2011). Data Structures, Algorithms and applications in C++. 2ndEdition, Universities Press

4. Additional Resource 4: Tenenbaum, A. M., Augenstein, M. J., & Langsam Y., (2009), *Data Structures Using C and C++*. 2nd edition. PHI.

Note: Ref1, Additional resource etc. as per the LOCF syllabus for the paper.

4.2. 'Queues'

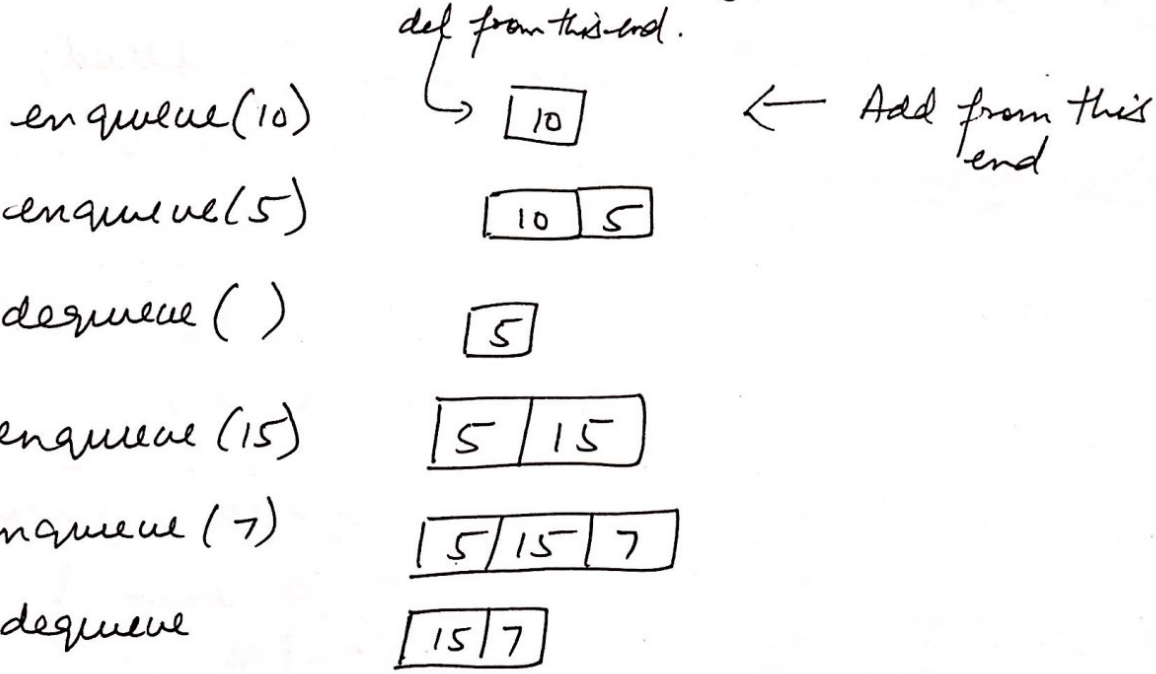
Queue is a waiting line that grows by adding elements to its one end & shrinks by removing elements from another end.

Queue follows FIFO structure : first In / first out.

Operations on Queues :-

1) clear() - clear the Queue.
2) isEmpty() - check to see if the Queue is empty.
3) enqueue(el) - Put the element 'el' at the end of Queue.
4) dequeue() - Take the first element from the Queue.
5) firstel() - Return the first element in the Queue without removing it.

Eg :-



Two types of Queues :-

1) Linear Queues
2) circular Queues

Formally, the queue abstract data type defines a container that keeps elements in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the ***front*** of the queue, and element insertion is restricted to the end of the sequence, which is called the ***rear*** of the queue. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle.

The ***queue*** abstract data type (ADT) supports the following operations:

enqueue(e): Insert element e at the rear of the queue.

dequeue(): Remove element at the front of the queue; an error occurs if the queue is empty.

front(): Return, but do not remove, a reference to the front element in the queue; an error occurs if the queue is empty.

The queue ADT also includes the following supporting member functions:

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

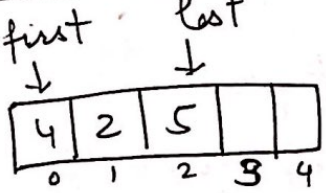
We illustrate the operations in the queue ADT in the following example.

Example 5.4: *The following table shows a series of queue operations and their effects on an initially empty queue, Q, of integers.*

Operation	***Output***	*front* $\leftarrow Q \leftarrow$ *rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	*5*	(5,3)
size()	*2*	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()

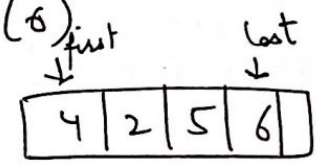
(I) Linear Queue

eg:

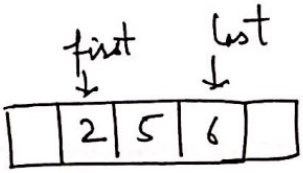


size = 5

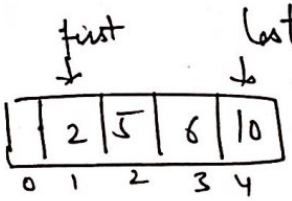
enque(6)



dequeue()



enqueue(10)



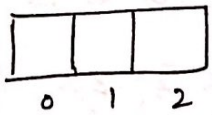
enqueue(12) → error, since no space to add element.

Disadvantage:- free space cannot be utilized (for eg. '0th' index cannot be filled).

Solution:- circular Queues.

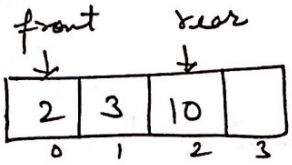
Linear Queues (using Arrays)

Initially :- front / rear (since Queue is empty)

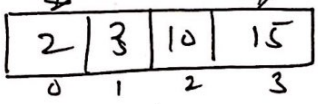


front will increase when we want to delete

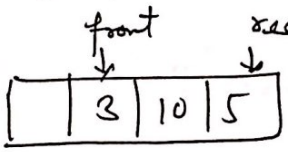
rear will increase when we want to insert

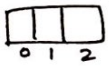


enqueue(15)



dequeue()



int size = 3 ie 


```
void insert ()
{
  do
   {
     cout<<"\n Enter Element";
     cin>>n;
     if (rear == -1)
       { front ++ ; rear++;
         a[ front ] = n;
       }
     else if ( rear < 2)
       { rear ++;
         a [rear ] = n ;
       }
     else
        cout<<" \n Rear reached last Position ";
        cout<< \n continue ??? ";
        cin>> ch;
   }while ( ch == 'y');
}
```

```
void delete()
{
    if (front == -1)
        cout<<"\n Queue is empty";
    elseif (front < 3 && front > -1)
    { cout <<"\n element deleted is: "<< a[front];
      front++;
    }
 else
     cout <<"\n Queue finished";
}

void display()
{
   cout <<"\n Queue is\n";
   if ( front == -1)
         cout <<"\n Empty";
   else
      for (int j = front; j <= rear; j++)
            cout << a[j];
}.
void main()
{

}
```

Circular Queues using Arrays

```
    first = last = -1;
    size = 10;
  Bool Is_full()
{
  return (first == 0 && last == size-1 || first == last+1);
}
  Bool Is_Empty()
{
  return (first == -1);
}
enqueue(el)
{
  If (!Isfull())
  {
    If (last == size-1 || last == -1)
    {
        a[0] = el;
        last = 0;
        If (first == -1) then first = 0;
    }
    else
        a[++last] = el;
  }
  else
      cout << "\n Queue is full";
}
 Dequeue()
{
  temp = a[first];
  If (first == last)
        last = first = -1;
  else If (first == size-1)
        first = 0;
  else first++;
} return temp;
```

circular array implementation

```
class carray
{ Public:
    carray() { first = last = -1; }
    void enqueue(); ---------
    int isfull()
    { return (first == 0 && last == size-1
                || first == last+1);
    }
    int isempty() { return first == -1; }
private:
    int a[size];
};

enqueue()
{
  if(isfull()){cout << "queue full"
                exit();}
  else if (last == -1)
        first = last = 0;
  else if (last == size-1)
        last = 0;
  else
      {last++;
       ~~a[last] = el;~~
      }
  a[last] = el;
}
dequeue()
{
    if (first == -1)
      { cout << "underflow"
        exit();}
    else
      {
        r = a[first];
        if (first == last)
              first = last = -1;
        else if (first == size-1)
              first = 0;
        else
              first++;
      }
  return r;
}
```

```
display()
{
   for(i = first; i <= last; i++)
        cout << a[i];
}
```

Implement Queue using stacks

Queue can be implemented using two stacks
Let Queue be implemented be Q & stacks used to implement Q be stack1 & stack2. Q can be implemented in two ways:-

1) By making EnQueue operation costly.

This method ensures that newly entered element is always on top of stack1 so that dequeue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

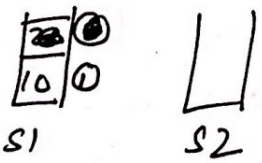
enqueue(q, x)

1. While stack1 is not empty, push all ele from stack1 & stack
2. Push x to stack1
3. Push everything back to stack1.

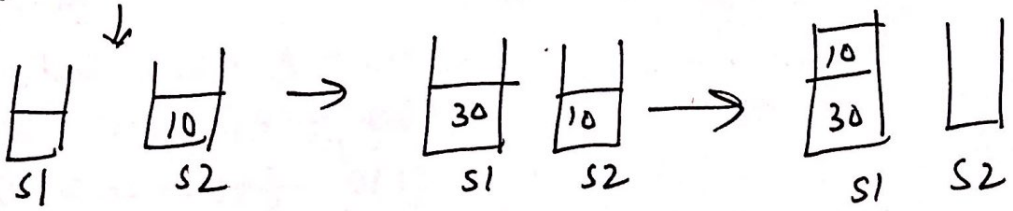
dequeue(q)

1. If stack1 is empty then error.
2. Pop an item from stack1 & return it.

28:-



enqueue(30)



a) By making dequeue operation costly.

In this method, in enqueue operation, the new element is entered at the top of stack 1.

In dequeue operation, if stack 2 is empty then all elements are moved to stack2 & finally top of stack2 is returned.

enqueue (q, x)

1) Push x to stack1

dequeue (q)

1) If both stacks are empty, then error.

2) If stack2 is empty

while stack1 is not empty, push all ele from stack1 to stack2.

3). Pop the element from stack2 & return it.

5.3 Double-Ended Queues

Consider now a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. Such an extension of a queue is called a ***double-ended queue***, or ***deque***, which is usually pronounced "deck" to avoid confusion with the dequeue function of the regular queue ADT, which is pronounced like the abbreviation "D.Q." An easy way to remember the "deck" pronunciation is to observe that a deque is like a deck of cards in the hands of a crooked card dealer—it is possible to deal off both the top and the bottom.

5.3.1 The Deque Abstract Data Type

The functions of the deque ADT are as follows, where D denotes the deque:

insertFront(e): Insert a new element e at the beginning of the deque.

insertBack(e): Insert a new element e at the end of the deque.

eraseFront(): Remove the first element of the deque; an error occurs if the deque is empty.

eraseBack(): Remove the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque includes the following support functions:

front(): Return the first element of the deque; an error occurs if the deque is empty.

back(): Return the last element of the deque; an error occurs if the deque is empty.

size(): Return the number of elements of the deque.

empty(): Return true if the deque is empty and false otherwise.

Example 5.5: *The following example shows a series of operations and their effects on an initially empty deque, D, of integers.*

Operation	***Output***	D
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	5	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	7	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

6.4.1 Breadth-First Traversal

Application of Queues.

Breadth-first traversal is visiting each node starting from the highest (or lowest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left). There are thus four possibilities, and one such possibility—a top-down, left-to-right breadth-first traversal of the tree in Figure 6.6c—results in the sequence 13, 10, 25, 2, 12, 20, 31, 29.

Implementation of this kind of traversal is straightforward when a queue is used. Consider a top-down, left-to-right, breadth-first traversal. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited. Considering that for a node on level n, its children are on level $n+1$, by placing these children at the end of the queue, they are visited after all nodes from level n are visited. Thus, the restriction that all nodes on level n must be visited before visiting any nodes on level $n+1$ is accomplished.

An implementation of the corresponding member function is shown in Figure 6.10.

FIGURE 6.10 Top-down, left-to-right, breadth-first traversal implementation.

```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                queue.enqueue(p->left);
            if (p->right != 0)
                queue.enqueue(p->right);
        }
    }
}
```