UNIT- II

Recursion:

A function which is calling itself is said to be recursion.

Ex: finding factorial of a given number using recursion.

```
#include<iostream.h>
int factorial(int n)
{ int f;
if(n==1)
return 1;
else
{
f=n*factorial (n-1);
return f;
}
}
Void main( )
{ int n;
cout<<”enter a number to find out factorial”;
cin>>n;
cout<<”the factorial of “<<n<<”is”<<factorial(n);
}
```

RECURRENCE:

A *recurrence* is a well-defined mathematical function where the function being defined is applied within its own definition. The factorial we defined as $n! = n \infty (n - 1)!$ is an example of recurrence with $1! = 1$ as the end condition. Take the *Fibonacci sequence* as an example. The Fibonacci sequence is the sequence of numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

The first two numbers of the sequence are both 1, whereas each succeeding number is the sum of the preceding two numbers (we arrived at 55 as the 10th number; it is the sum of 21 and 34, the eighth and ninth numbers). Let us define a function $F(n)$ that returns the $(n + 1)$th Fibonacci number. First, we define the *base cases* as represented by the following functions:

$F(1) = 1$ and

$F(2) = 1$

Now, we consider the other numbers. To get the $(n + 1)$th Fibonacci number, we just add the nth and the $(n - 1)$th Fibonacci numbers. $F(n) = F(n - 1) + F(n - 2)$ This function F is called *recurrence* since it computes the nth value in terms of $(n - 1)$th and $(n - 2)$th Fibonacci values. The problems that can be described using recurrence are easily expressed as recursive functions in programming.

The process of recursion occurs when a function calls itself. Recursion is useful in situations where solving one or more smaller versions of the same problem can solve the problem. Computing the value of three to the fourth power can be considered as

$$3^4 = 3 \ \mathrm{X} \ 3^3$$

Three cubed can be defined as $3^3 = 3 \ \mathrm{X} \ 3^2$

Three squared is $3^2 = 3 \ \mathrm{X} \ 3^1$

Finally, $3 = 3 \ \mathrm{X} \ 3^0 = 3 \ \mathrm{X} \ 1$

The recurrence for this computation is $\mathrm{M}^n = \mathrm{M} \ \mathrm{X} \ \mathrm{M}^{n-1}$

USE OF STACK IN RECURSION:

The stack is a special area of memory where temporary variables are stored. It acts on the LIFO principle. The following program code explains how recursive functions use the stack.

```
if(n <= 1)
return 1;
else
return n * Factorial(n - 1);
```

Let `n` = 3; that is, let us compute the value of 3!, which is 3 X 2 X 1 = 6. When the function is called f or the first time, `n` holds the value 3, so the `else` statement is executed. The function knows the value of `n` but not of `Factorial(n - 1)`, so it pushes `n` (value = 3) onto the stack and calls itself for the second time with the value 2. This time, the `else` statement is again executed, and `n` (value = 2) is pushed onto the stack as the function calls itself for the third time with the value 1. now, the `if` statement is executed and as `n` = 1, the function returns 1. Since the value of `Factorial(1)` is now known, it reverts to its second execution by popping the last value 2 from the stack and multiplying it by 1. This operation gives the value of `Factorial(2)`, so the function reverts to its first execution by popping the next value 3 from the stack and multiplying it with the factorial, giving the value 6, which the function finally returns.

From this example, we notice the following:

1. The `Factorial()` function in Program Code 4.2 runs three times for `n` = 3, out of which it calls itself two times. The number of times a function calls itself is known as the *recursive depth* of that function.
2. Each time the function calls itself, it stores one or more variables on the stack. Since stacks hold a limited amount of memory, the functions with a high recursive depth may crash because of non-

availability of memory. Such a situation is known as *stack overflow*.
3. Recursive functions usually have (and in fact should have) a *terminating* (or *end*) *condition*. The `Factorial()` function in Program stops calling itself when `n` = 1.

4. All recursive functions go through two distinct phases. The frst phase, *winding*, occurs when the function calls itself and pushes values onto the stack. The second phase, *unwinding*, occurs when the function pops values from the stack, usually after the end condition.

Variants of recursion:

The recursive functions are categorized as direct, indirect, linear, tree, and tail recursions. Recursion may have any one of the following forms:
1. A function calls itself.
2. A function calls another function which in turn calls the caller function.
3. The function call is part of the same processing instruction that makes a recursive function call.
A few more terms that are used with respect to recursion are explained in the following section.

1. Binary recursion : A *binary recursive* function calls itself twice. Fibonacci numbers computation, quick sort, and merge sort are examples of binary recursion. Program Code is an example of a binary recursion as the function `Fib()` calls itself twice.

```
int Fib(n)
{
if(n == 1 ||n == 2)
return 1;
else
return(Fib(n - 1) + Fib(n - 2));
}
```

2. Direct recursion

Recursion is when a function calls itself. Recursion is said to be *direct* when a function calls itself directly. The f`actorial()` function we discussed in Program Code is an example of direct recursion. Another example is The `Power()` function.

```
int Power(int x, int y)
{
if(y == 1)
return x;
else
return (x * Power(x, y - 1));
}
```

3. Indirect recursion

A function is said to be indirectly recursive if it calls another function, which in turn calls it. The following Program Code is an example of an indirect recursion, where the function `Fact()` calls the function `Dummy()`, and the function `Dummy()` in turn calls `Fact()`.

```
int Fact(int n)
{
if(n <= 1)
return 1;
else
return (n * Dummy(n - 1));
}
void Dummy(int n)
{
Fact(n);
}
```

4. Tail recursion

A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Tail recursion is also used to return the value of the last recursive call as the value of the function. The `Binary_Search()` function in Program Code is an example of a tail recursive function.

```
int Binary_Search(int A[], int low, int high, int key)
{
int mid;
if(low <= high)
{
mid = (low + high)/2;
if(A[mid] == key)
return mid;
else if(key < A[mid])
return Binary_Search(A, low, mid - 1, key);
else
return Binary_Search(A, mid + 1, high, key);
}
return -1;
}
```

5. Linear recursion:

Depending on the way the recursion grows, it is classified as *linear* or *tree*. A recursive function is said to be *linearly recursive* when no pending operation involves another recursive call, for example, the `Fact()` function. This is the simplest form of recursion and occurs when an action has a simple repetitive structure consisting of some basic steps followed by the action again. The `Factorial()` function in Program Code is an example of linear recursion.

6. Tree recursion

In a recursive function, if there is another recursive call in the set of operations to be completed after the recursion is over, this is called a *tree recursion.* Examples of tree recursive functions are the quick sort and merge sort algorithms, the FibSeries algorithm, and so on. The Fibonacci function FibSeries() is defined as

FibSeries(n) = 0, if $n = 0$

= 1, if $n = 1$

= FibSeries($n - 1$) + FibSeries($n - 2$), otherwise

Let $n = 5$.

FibSeries(0) = 0

FibSeries(1) = 1

FibSeries(2) = FibSeries(0) + FibSeries(1) = 1

FibSeries(3) = FibSeries(1) + FibSeries(2) = 2

FibSeries(4) = FibSeries(2) + FibSeries(3) = 3

FibSeries(5) = FibSeries(3) + FibSeries(4) = 5

Figure demonstrates this explanation for $n = 4$.

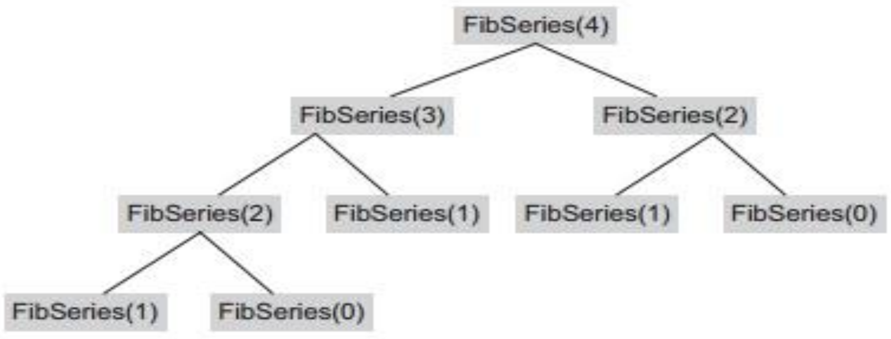


Fig. 4.1 Recursive calls in Fibonacci recursive function for $n = 4$

4.5 EXECUTION OF RECURSIVE CALLS

Let us now see how recursive calls are executed. At every recursive call, all reference parameters and local variables are pushed onto the stack along with the function value and return address. The data is conceptually placed in a *stack frame*, which is pushed onto the system stack. A stack frame contains four different elements:

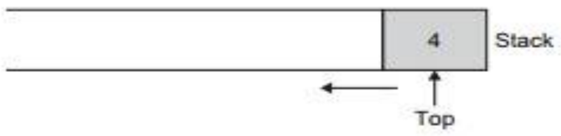
1. The reference parameters to be processed by the called function
2. Local variables in the calling function
3. The return address
4. The expression that is to receive the return value, if any

Consider the following two lines from the `Factorial()` function in Program Code 4.2:

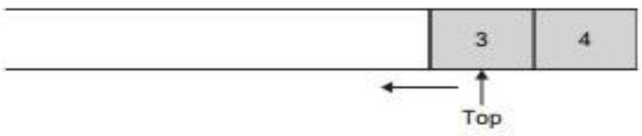
```
if(n <= 1) return 1;
else return n * Factorial(n - 1);
```

Consider the first call as `Factorial(4)`. Now,

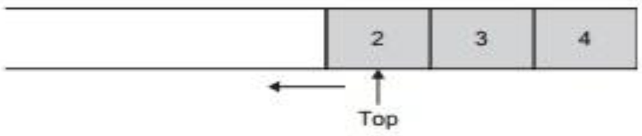
1. $n = 4$
   Hence, statement 2, which is a recursive call, is executed.
   Push 4 onto the stack and call `Factorial(4 - 1)`.



2. $n = 3$
   Hence, push 3 onto the stack and call `Factorial(2)`.



3. $n = 2$
   Hence, push 2 onto the stack and call `Factorial(1)`.



4. $n = 1$

Now execute statement 1, which returns 1.

5. Pop the contents and $n = 2$, so now the expression becomes $2 \infty 1$.

6. Now, $n = 3$ after popping the top of the stack contents. Therefore, the expression is $3 \infty 2 \infty 1$.

7. After popping the top of the stack contents applying $n = 4$, the expression is $4 \infty 3 \infty 2 \infty 1 = 24$.

8. After popping the top of the stack contents, we get to know that the stack is empty, and the answer is $4! = 24$.

At the end condition, when no more recursive calls are made, the following steps are performed:

1. If the stack is empty, then execute a normal return.
2. Otherwise, pop the stack frame, that is, take the values of all the parameters that are on the top of the stack and assign these values to the corresponding variables.
3. Use the return address to locate the place where the call was made.
4. Execute all the statements from that place (address) where the call was made.
5. Go to step 1

<u>ITERATION VERSUS RECURSION</u>:

Recursion is a top–down approach of problem solving. It divides the problem into pieces or selects one key step, postponing the rest. On the other hand, iteration is more of a bottom–up approach. It begins with what is known and from this constructs the solution step by step. It is hard to say that the non-recursive version is better than the recursive one or vice versa. However, a few languages do not support writing recursive code, such as FORTRAN or COBOL. The non-recursive version is more effcient as the overhead of parameter passing in most compilers is heavy.

Demerits of recursive algorithms

1. Many programming languages do not support recursion

2. Even though mathematical functions can be easily implemented using recursion, it is always at the cost of additional execution time and memory space.

Demerits of iterative Methods

1. Iterative code is not readable and hence not easy to understand.

2. In iterative techniques, looping of statements is necessary and needs a complex logic.

3. The iterations may result in a lengthy code.

QUEUES:

Queue is a Linear Data Structure which has two open ends. The data entered and deleted from two different ends. The end at which data is inserted is called the *rear* and that from which it is deleted is called the *front*. Queue is a first in first out (FIFO) or last in last out (LILO) structure.

Primitive operations:

1.Create This operation should create an empty queue. Here `max` is the maximum initial size that is defned.

```
#defne max 50
int Queue[max];
int Front = Rear = -1;
```

2. ***Is_Empty*** This operation checks whether the queue is empty or not. This is confirmed by comparing the values of `Front` and `Rear`. If `Front = Rear`, then `Is_Empty` returns true, else returns false.

```
bool Is_Empty()
{ if(Front == Rear)
return 1;
else
return 0;
}
```

3. Is_Full: before insertion, the queue must be checked for the `Queue_Full` state. When `Rear` points to the last location of the array, it indicates that the queue is full`bool Is_Full()`

```
{ if(Rear == max - 1)
return 1;
else
return 0;
}
```

4. Add This operation adds an element in the queue if it is not full. As `Rear` points to the last element of the queue, the new element is added at the (rear + 1)th location.

```
void Add(int Element)
{ if(Is_Full())
cout << "Error, Queue is full";
else
Queue[++Rear] = Element;
}
```

5. Delete This operation deletes an element from the front of the queue and sets `Front` to point to the next element. `Front` can be initialized to one position less than the actual front. We should first increment the value of `Front` and then remove the element.

```
int Delete()
{ if(Is_Empty())
cout << "Sorry, queue is Empty";
else
return(Queue[++Front]);
}
```

6. getFront The operation `getFront` returns the element at the front, but unlike `delete`, this does not update the value of `Front`.

```
int getFront()
{ if(Is_Empty())
cout << "Sorry, queue is Empty";
else
return(Queue[Front + 1]);
}
```

Queue ADT:

The basic operations performed on the queue include adding and deleting an element, traversing the queue, checking whether the queue is full or empty, and fnding who is at the front and who is at the rear ends.

A minimal set of operations on a queue is as follows:

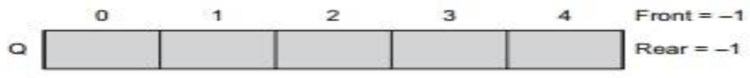
1. `create()`—creates an empty queue, *Q*
2. `add(i,Q)`—adds the element *i* to the rear end of the queue, *Q* and returns the new queue
3. `delete(Q)`—takes out an element from the front end of the queue and returns the resulting queue
4. `getFront(Q)`—returns the element that is at the front position of the queue
5. `Is_Empty(Q)`—returns true if the queue is empty; otherwise returns false

The complete specification for the queue ADT is given in Algorithm

```
class queue(element)
declare create() -> queue
add(element, queue) ->  queue
delete(queue) -> queue
getFront(queue) -> queue
Is_Empty(queue) -> Boolean;
For all Q E queue, i E  element let
Is_Empty(create()) = true
Is_Empty(add(i,Q)) = false
delete(create()) = error
delete(add(i,Q)) =
if Is_Empty(Q) then create
else add(i, delete(Q))
getFront(create) = error
getFront(add(i, Q)) =
if Is_Empty(Q) then i
else getFront(Q)
end
end queue
```

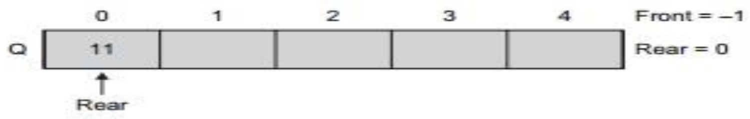
Since a queue is a linear data structure, it can be implemented using either arrays or linked lists

Let *Q* be an empty queue with `Front = Rear = –1`. Let `max = 5`.

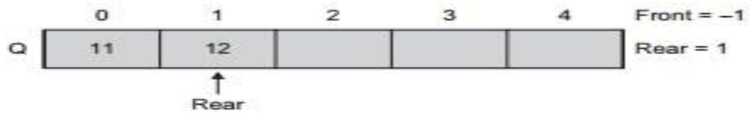


Consider the following statements:

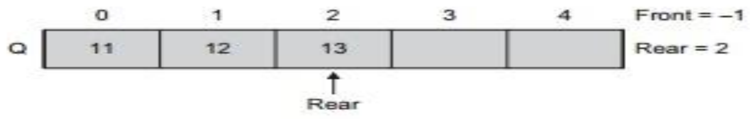
1. `Q.Add(11)`



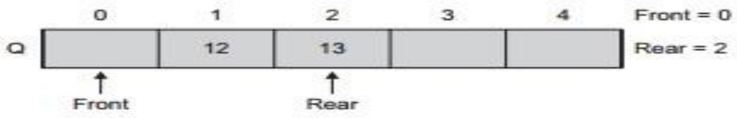
2. `Q.Add(12)`



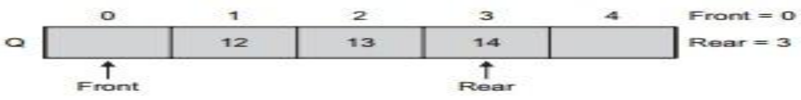
3. `Q.Add(13)`



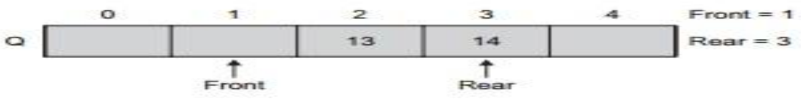
4. `A = Q.Delete()`
   Here, `A = Q[++Front] = Q[0] = 11`



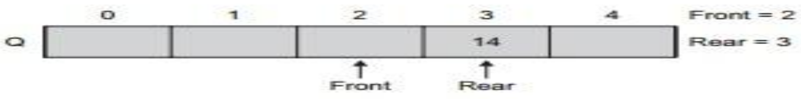
5. `Q.Add(14)`



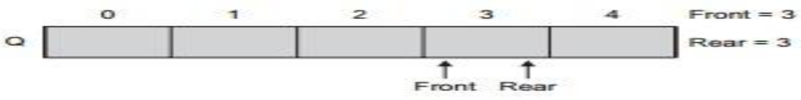
6. `A = Q.Delete()`
   `A = Q[++ Front] =   Q [1] = 12`



7. `A = Q.Delete()`
   `A = 13`



8. `A = Q.Delete()`



9. `A = Q.Delete()`
   Here we get the `Queue_empty` error condition as `Front = Rear = 3`
   Let us execute a few more statements.

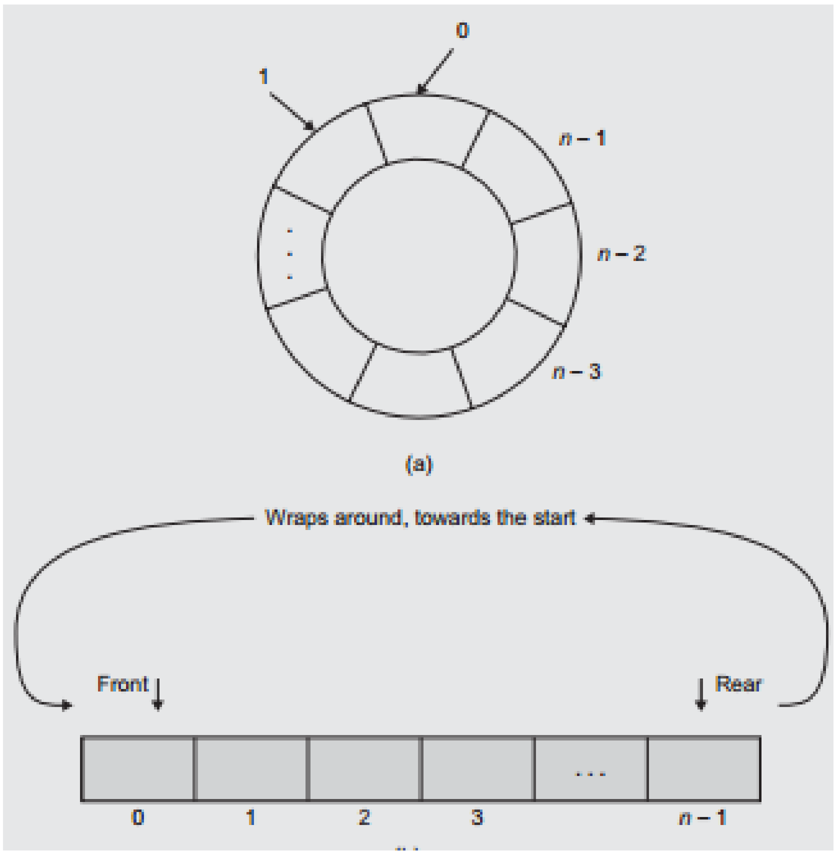
```
//Queue ADT
class queue
{
private:
int Rear, Front;
int Q[50];
int max;
int Size;
public:
queue()
{
Size = 0; max = 50;
Rear = Front = -1 ;
}
int Is_Empty();
int Is_Full();
void Add(int Element);
int Delete();
int getFront();
};
int queue :: Is_Empty()
{
if(Front == Rear)
return 1;
else
return 0;
}
int queue :: Is_Full()
{
if(Rear == max - 1)
return 1;
else
return 0;
}
void queue :: Add(int Element)
{
if(!Is_Full())
Q[++Rear] = Element;
Size++;
}
int queue :: Delete()
{
if(!Is_Empty())
{
Size--;
return(Q[++Front]);
}
}
int queue :: getFront()
{if(!Is_Empty())
return(Q[Front + 1]);
}void main(void)
{queue Q;
Q.Add(11);
Q.Add(12);
Q.Add(13);
cout << Q.Delete() << endl;
Q.Add(14);
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
Q.Add(15);
Q.Add(16);
cout << Q.Delete() << endl;
}
```

CIRCULAR QUEUES:

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.

The following are the merits of using circular queues:

1. By using circular queues, data shifting is avoided as the *front* and *rear* are modifed by using the `mod()` function. The `mod()` operation wraps the queue back to its beginning.
2. If the number of elements to be stored in the queue is fxed (i.e., if the queue size is specifc), the circular queue is advantageous.
3. Many practical applications such as printer queue, priority queue, and simulations use the circular queue.



```
#include<iostream.h>
class Cqueue
{ private:
int Rear, Front;
int Queue[50];
int Max;
int Size;
public:
Cqueue() {Size = 0; Max = 50; Rear = Front = -1;}
int Empty();
int Full();
void Add(int Element);
int Delete();
int getFront();
};
int Cqueue :: Empty()
{ if(Front == Rear)
return 1;
else
return 0;
}
int Cqueue :: Full()
{ if(Rear == Front)
return 1;
else
return 0;
}
void Cqueue :: Add(int Element)
{ if(!Full())
Rear = (Rear + 1) % Max;
Queue[Rear] = Element;
Size++;
}
int Cqueue :: Delete()
{ if(!Empty())
Front = (Front + 1) % Max;
Size--;
return(Queue[Front]);
}
int Cqueue :: getFront()
{ int Temp;
if(!Empty())
Temp = (Front + 1) % Max;
return(Queue[Temp]);
}
void main(void)
{ Cqueue Q;
Q.Add(11);
Q.Add(12);
Q.Add(13);
cout << Q.Delete() << endl;
Q.Add(14);
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
Q.Add(15);
Q.Add(16);
cout << Q.Delete() << endl;
}
```

DEQUE:

The word *deque* is a short form of double-ended queue. It is pronounced as 'deck'. *Deque* defines a data structure where elements can be added or deleted at either the front end or the rear end, but no changes can be made elsewhere in the list. Thus, *deque* is a generalization of both a stack and a queue. It supports both stack-like and queue-like capabilities. It is a sequential container that is optimized for fast index-based access and efficient insertion at either of its ends. Deque can be implemented as either a continuous deque or as a linked deque. Figure shows the representation of a deque.

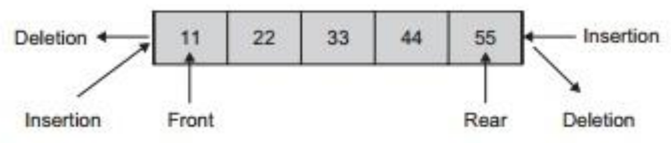


Fig. 5.5 Representation of a deque

The *deque ADT* combines the characteristics of stacks and queues. Similar to stacks and queues, a deque permits the elements to be accessed only at the ends. However, a deque allows elements to be added at and removed from either end. We can refer to the operations supported by the deque as `EnqueueFront`, `EnqueueRear`, `DequeueFront`, and `DequeueRear`. When we complete a formal description of the deque and then implement it using a dynamic, linked implementation, we can use it to implement both stacks and queues, thus achieving signifcant code reuse.

The following are the four operations associated with deque:

1. `EnqueueFront()`—adds elements at the front end of the queue
2. `EnqueueRear()`—adds elements at the rear end of the queue
3. `DequeueFront()`—deletes elements from the front end of the queue
4. `DequeueRear()`—deletes elements from the rear end of the queue

For stack implementation using deque, `EnqueueFront` and `DequeueFront` are used as `push` and `pop` functions, respectively.

LINKED LIST

A linked list is an ordered collection of data in which each element (node) contains a minimum of two values, data and link(s) to its successor (and/or predecessor). A list with one link field using which every element is associated to its successor is known as a singly linked list (SLL). In a linked list, before adding any element to the list, a memory space for that node must be allocated. A link is made from each item to the next item in the list as shown in Fig.

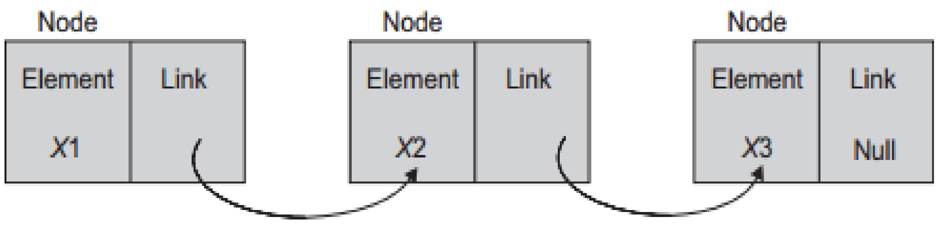


Fig. 6.3 Linked list

Each node of the linked list has at least the following two elements:

1. The data member(s) being stored in the list.s

2. A pointer or link to the next element in the list.

The last node in the list contains a null pointer

Linked List terminology

The following terms are commonly used in discussions about linked lists:

Header node: A header node is a special node that is attached at the beginning of the Linked list. This header node may contain special information (metadata) about the linked list as shown in Fig

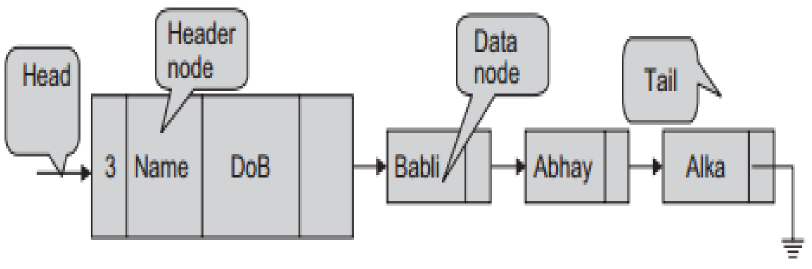


Fig. 6.4 Linked list with header node

This special information could be the total number of nodes in the list, date of creation, type, and so on.

The header node may or may not be identical to the data nodes.

Data node: The list contains data nodes that store the data members and link(s) to its predecessor (and/or successor).

Head pointer: The variable (or handle), which represents the list, is simply a pointer to the node at the head of the list. A linked list must always have at least one pointer pointing to the first node (head) of the list. This pointer is necessary because it is the only way to access the further links in the list. This pointer is often called head pointer, because a linked list may contain a dummy node attached at the start position called the header node.

Tail pointer: Similar to the head pointer that points to the first node of a linked list, we may have a pointer pointing to the last node of a linked list called the tail pointer.

Header node: Tail pointer Similar to the head pointer that points to the first node of a linked list, we may have a pointer pointing to the last node of a linked list called the tail pointer.

Primitive Operations

The following are basic operations associated with the linked list as a data structure:

1. Creating an empty list

2. Inserting a node

3. Deleting a node

4. Traversing the list

Some more operations, which are based on the basic operations, are as follows:

5. Searching a node

6. Updating a node

7. Printing the node or list

8. Counting the length of the list

9. Reversing the list

10. Sorting the list using pointer manipulation

11. Concatenating two lists

12. Merging two sorted lists into a third sorted list

In addition, operations such as merging the second sorted list into the first sorted list and many more are possible by the use of these operations.

REPRESENTATION OF LINKED LISTS USING ARRAYS:

1. REPRESENTATION OF LINKED LISTS:

Let *L* be a set of names of months of the year.

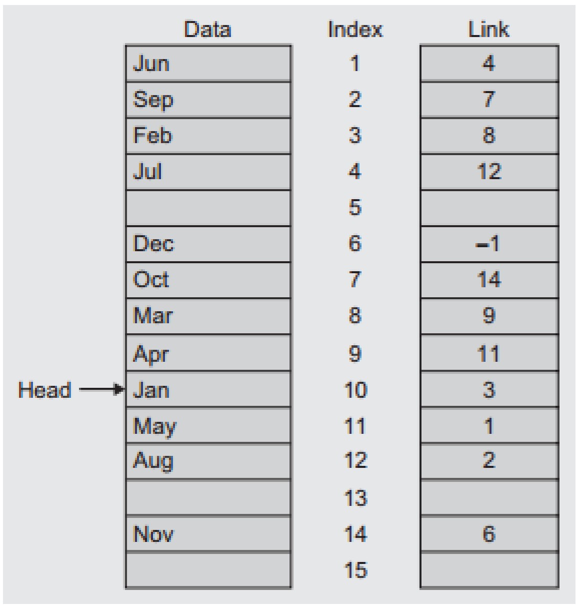
L = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}

Here, *L* is an ordered set. The linked organization of this list using arrays is shown in Fig. The elements of the list are stored in the one-dimensional array, `Data`. The elements are not stored in the same order as in the set *L*. They are also not stored in a continuous block of locations. Note that the data elements are allowed to be stored anywhere in the array, in any order.

To maintain the sequence, the second array, `Link`, is added. The values in this array are the links to each successive element. Here, the list starts at the 10th location of the array.

Let the variable `Head` denote the start of the list.

L = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}



	Data	Index	Link
	Jun	1	4
	Sep	2	7
	Feb	3	8
	Jul	4	12
		5	
	Dec	6	−1
	Oct	7	14
	Mar	8	9
	Apr	9	11
Head ⟶	Jan	10	3
	May	11	1
	Aug	12	2
		13	
	Nov	14	6
		15	

Fig. 6.5 Realization of linked list using 1D arrays

Here, `Head = 10` and `Data[Head] = Jan`.

Let us get the second element. The location where the second element is stored t is `Link[Head] = Link[10]`. Hence, `Data[Link[Head]] = Data[Link[10]] = Data[3] = Feb`. Let us get the third data element through the second element. `Data[Link[3]] = Data[8] = Mar,` and so on. Continuing in this manner, we can list all the members in the sequence. The link value of the last element is set to −1 to represent the end of the list. Figure 6.6 shows the same representation as in Fig. 6.5 but in a different manner.

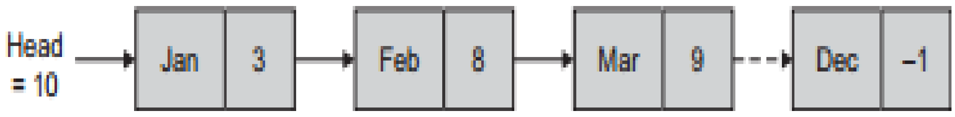


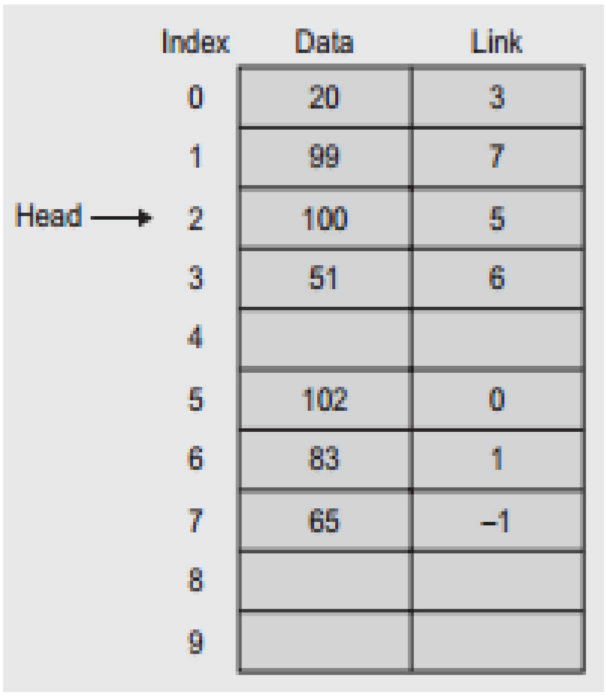
Fig. 6.6 Linked organization

Representation using 2D array:

Even though `data` and `link` are shown as two different arrays, they can be implemented using one 2D array as follows:

```
int Linked_List[max][2];
```

Figure illustrates the realization of a linked list using a 2D array where $L = \{100, 102, 20, 51, 83, 99, 65\}$, `Max = 10` and `Head = 2`.



	Index	Data	Link
	0	20	3
	1	99	7
Head →	2	100	5
	3	51	6
	4		
	5	102	0
	6	83	1
	7	65	–1
	8		
	9		

Fig. 6.7 Realization of linked list using 2D arrays

Linked List ADT:

```
#include<iostream>
using namespace std;
class Node
{
public :
int data;
Node *link;
};
class Llist
{
private:
Node *Head,*Tail;
public:
Llist()
{
Head = NULL;
}
void Create();
void Display();
Node* GetNode();
void Append(Node* NewNode);
void Insert_at_Pos( Node *NewNode, int
position);
void DeleteNode(int del_position);
void sort();
};
void Llist::sort()
{ Node *ptr,*s;
int value;
       if(Head==NULL)
       {cout<<"the list is
empty"<<endl;
       }
       ptr=Head;
           while (ptr != NULL)
    {
for (s=ptr->link;s !=NULL;s = s->link)
        {
            if (ptr->data > s->data)
            {
                value = ptr->data;
                ptr->data = s->data;
                s->data = value;
            }
        }
        ptr = ptr->link;
    }
}

void Llist :: Create()
{
char ans;
Node *NewNode;
while(1)
{
cout << "Any more nodes to be added
(Y/N)";
cin >> ans;
if(ans == 'n') break;
NewNode = GetNode();
Append(NewNode);
}
}
void Llist :: Append(Node* NewNode)
{
if(Head == NULL)
{
Head = NewNode;
Tail = NewNode;
}
else
{
Tail->link = NewNode;
Tail = NewNode;
}
}
Node* Llist :: GetNode()
{
Node *Newnode;
Newnode = new Node;
cin >> Newnode->data;
Newnode->link = NULL;
return(Newnode);
}
void Llist :: Display()
{
Node *temp = Head;
if(temp == NULL)
cout << "Empty List";
else
{
while(temp != NULL)
{
cout << temp->data << "\t";
temp = temp->link;
}
}
cout << endl;
}
void Llist :: DeleteNode(int pos)
{
int count = 1, flag = 1;
Node *curr, *temp;
temp = Head;
if(pos == 1)
{
Head = Head->link;
delete temp;
}
else
{
while(count != pos - 1)
{
temp = temp->link;
if(temp == NULL)
{
flag = 0; break;
}
count++;
}
if(flag == 1)
{
curr = temp->link;
temp->link = curr->link;
delete curr;
}
else
cout << "Position not found" << endl;
}
}
void Llist :: Insert( Node *NewNode,
int position)
{
Node *temp = Head;
int count = 1,flag = 1;
if(position == 1)
NewNode->link = temp;
Head = NewNode; // update head
}
else
```

```
{
while(count != position - 1)
{
temp = temp->link;
if(temp == NULL)
{
flag = 0; break;
}
count ++;
}
if(flag == 1)
{
NewNode->link = temp->link;
temp->link = NewNode;
}
else
cout << "Position not found" << endl;
}
}
void Llist::search()
{
        int value,pos=0;
        bool flag=false;
        if(Head==NULL)
        {
                cout<<"list is empty";
                return;
        }
        cout<<"enter the value to be
searched";
        cin>>value;
        Node *n=Head;
        while(n!=NULL)
        {

                pos++;
                if(n->data==value)
                {flag= true;

                cout<<"element"<<value<<"i
               s   found at
               position"<<pos<<endl;
                 }
        n=n->link;
}
if(!flag)
cout<<"element"<<value<<"not found in
the list"<<endl;
}
int main()
{
        Node *NewNode;
Llist L1;
L1.Create();
L1.Display();
L1.sort();
L1.Display();
NewNode=L1.GetNode();
L1.Insert(NewNode,2);
L1.Display();
L1.DeleteNode(2);
L1.Display();
}
```

LINKED LIST VARIANTS:

Linked list can be classified as follows

1. Singly linked list
2. Doubly linked list

1) **Single linked list:** A linked list in which every node has one link f eld, to provide information about where the next node of the list is, is called as *singly linked list* (SLL). It has no knowledge about where the previous node lies in the memory. In SLL, we can traverse only in one direction. We have no way to go to the *i*th node from (*i* + 1)th node, unless the list is traversed again from the first node

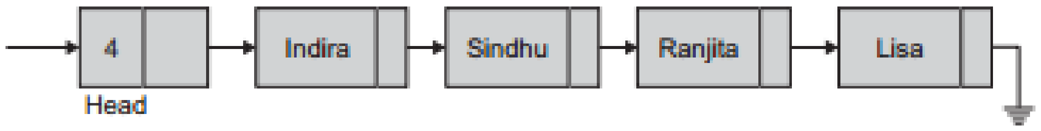


Fig. 6.25 Singly linked list

2) Double Linked List

In a doubly linked list (DLL), each node has two link fields to store information about the one to the next and also about the one ahead of the node. Hence, each node has knowledge of its successor and also its predecessor. In DLL, from every node, the list can be traversed in both the directions.

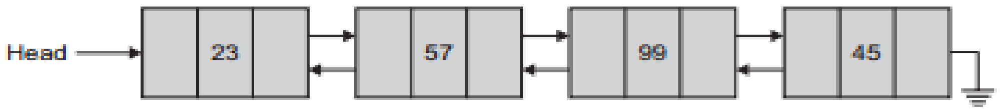


Fig. 6.26 Doubly linked list

Both SSL and DLL may or may not contain a header node. The one with a header node is explicitly mentioned in the title as a header-SLL and a header-DLL. These are also called as singly linked list with header node and doubly linked list with header node.

II. The other classification of linked lists based on their method of traversing is

1. Linear linked list
2. Circular linked list

1. Linear Linked List:

The linear linked list having only 1 way traversing all the elements in the list can be accessed by traversing from the first node of the list.

2. Circular Linked list:

In linear linked list it is not possible to traverse the list from the last node or to reach any of the nodes that precede any node. To overcome this disadvantages the link field of the last node can be set to point to the first node rather than the Null. Such a linked list is called as circular linked list.

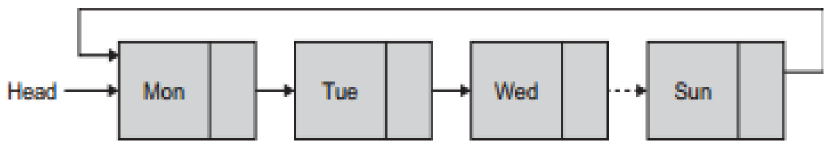


Fig. 6.27 Circular linked list

Double Linked List:

In Double linked list each node contains two links. One to its predecessor and other to its successor.

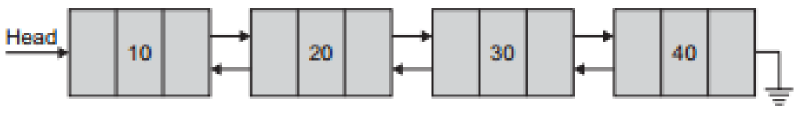


Fig. 6.28 Doubly linked list of four nodes

Each node of a DLL has three fields in general but must have at least two link fields.

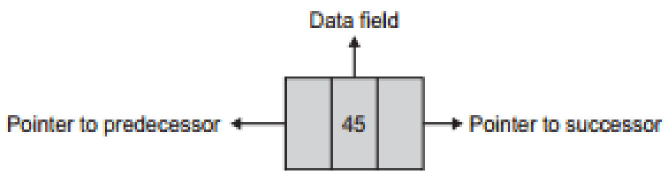


Fig. 6.29 Node structure of doubly linked list

Double Linked List ADT:

```
#include<iostream>
using namespace std;
class dllnode
{
public:
int data;
dllnode *prev, *next;
dllnode()
{
prev = next = NULL;
} };
class dlist
{ private
 dllnode *head, *tail;
 public:
 dlist()
{ head = tail = NULL; }
void create();
dllnode* getnode();
void append(dllnode* newnode);
void insert(dllnode *newnode, int pos);
void del(int val);
void search();
void display();
};
```

```
dllnode* dlist :: getnode()
{
dllnode *newnode;
newnode = new dllnode;
cout << "Enter Data";
cin >> newnode->data;
newnode->next = newnode->prev = NULL;
return(newnode);
}
void dlist :: append(dllnode* newnode)
{
if(head == NULL)
{
head = newnode;
tail = newnode;
}
else
{
tail->next = newnode;
newnode->prev = tail;
tail = newnode;
}
}
void dlist :: create()
{
char ans;
dllnode *newnode;
while(1)
{
cout << "Any more nodes to be added (Y/N)";
cin >> ans;
if(ans == 'n') break;
newnode = getnode();
append(newnode);
}
}
void dlist :: insert(dllnode* newnode, int pos)
{
dllnode *temp = head;
int count = 1,flag=1;
if(head==NULL)
head=tail=newnode;
else if(pos == 1)
{
newnode->next = head;
head->prev = newnode;
head = newnode;
}
else
{
while(count != pos)
{
temp = temp->next;
if(temp == NULL)
{
        flag=0;break;
}
count++;
}
if(flag == 1)
{

(temp->prev)->next = newnode;
newnode->prev = temp->prev;
temp->prev = newnode;
newnode->next=temp;
}
else
cout << "The node position is not found" <<
endl;
}
}
void dlist :: del(int val)
{
dllnode *curr, *temp;
curr = head;
while(curr!=NULL)
{
if(curr->data == val)
break;
curr = curr->next;
}
if(curr != NULL)
{
if(curr == head)
{
head = head->next;
head->prev = NULL;
delete curr;
}
else
{
if(temp == tail)
{
tail = temp->prev;
(temp->prev)->next = NULL;
delete temp;
}
else
{
(curr->prev)->next = curr->next;
(curr->next)->prev = curr->prev;
delete curr;
}
}
if(head == NULL)
{
tail = NULL;
}
}
else
cout << "Node to be deleted is not found \n";
}
void dlist::search()
{
            int value,pos=0;
            bool flag=false;
            if(head==NULL)
            {
                        cout<<"list is empty";
                        return;
            }
            cout<<"enter the value to be
searched";
            cin>>value;
            dllnode *n=head;
            while(n!=NULL)
            {

                        pos++;
                        if(n->data==value)
                        {flag= true;
                        cout<<"element"<<value<<"is
found at position"<<pos<<endl;
                        }
            n=n->next;
}
if(!flag)
cout<<"element"<<value<<"not found in the
list"<<endl;
            }
void dlist :: display()
{
dllnode *temp = head;
if(temp == NULL)
cout << "Empty List";
else
{
while(temp != NULL)
{
cout << temp->data << "\t";
temp = temp->next;
}
}
cout << endl;
}
int main()
{
            dllnode *newnode;
            int val;
            dlist d;
            d.create();
            d.display();
            newnode=d.getnode();
            d.insert(newnode,3);
            d.display();
            cout<<"enter element to be deleted";
            cin>>val;
            d.del(val);
            d.display();
            d.search();
}
```

CIRCULAR LINKED LIST:

In a singly linear list, the last nodes link field is set to `Null`. Instead of that, store the address of the first node of the list in that link field. This change will make the last node point to the first node of the list. Such a linked list is called *circular linked list* . From any node in such a list, it is possible to reach to any other node in the list. We need not traverse the list again right from the first node.

1. **Singly circular linked list:**

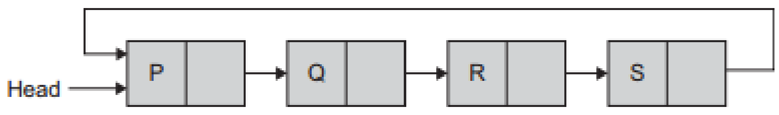


Fig. 6.35 Singly circular linked list

In a singly circular list, the pointer head points to the first node of the list. From the last node, we can access the first node. Remember that we cannot access the last node through the header node. We have access to only the first node. We need to traverse the whole list to reach to the last node.

CIRCULAR LINKED LIST WITH HEADER NODE:

Consider a circular list with a single node in the list (Fig. 6.37).

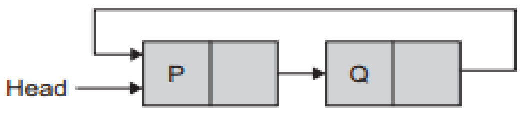


Fig. 6.37 Singly circular linked list with two nodes

Circular list with a single node has a problem of checking end of traversal as

```
(while(x->link != Head));
```

This would enter an infinite loop.

So, we can use a circular linked list with header node as shown in Fig. 6.38.

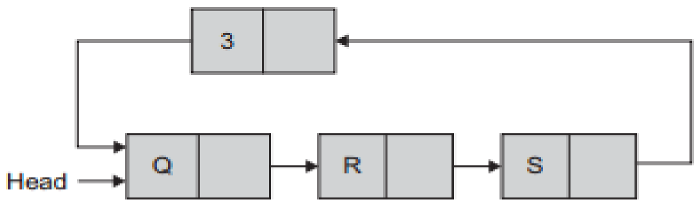
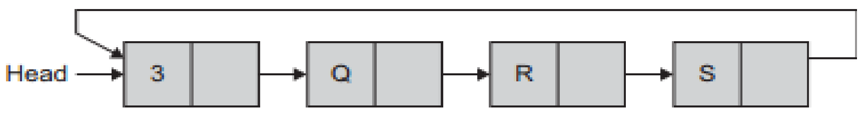


Fig. 6.38 Singly circular linked list with header node

The circular list with header node drawn in Fig. 6.38 can be redrawn as in Fig. 6.39



Suppose we want to insert a new node at the front of this list. We have to change the link field of the last node. In addition, we have to traverse the whole list to reach the last node as the link field of the last node is also to be updated. Hence, it is convenient if the head pointer points to the last node rather than the header node, which is the first node of the list.

2. **DOUBLY CIRCULAR LINKED LIST:**

In doubly circular linked list, the last node's next link is set to the first node of the list and the first node's previous link is set to the last node of the list. This gives access to the last node directly from the first node (Fig. 6.41).

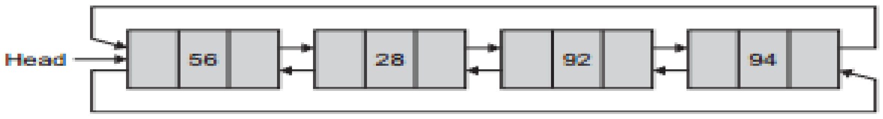


Fig. 6.41 Doubly circular list

Figure 6.41 represents the doubly circular linked list without a header node. Figure 6.42 is the doubly circular linked list with header node. Header node may store some relevant information of the list.

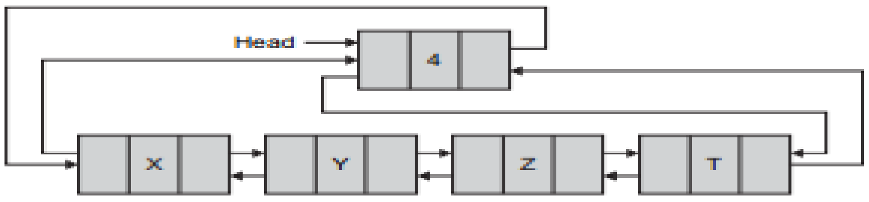


Fig. 6.42 Headed doubly circular list

The operations on circular linked list—`insert`, `delete`, `create` and `traverse`—follow the same method as that of linear list except for a few changes. We can redraw the circular list with header node as in Fig. 6.43.

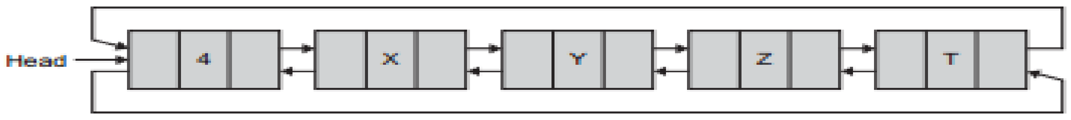


Fig. 6.43 Headed doubly circular list—representation 2

<u>**APPLICATION OF LINKED LIST–GARBAGE COLLECTION**:</u>

To be able to reuse this memory, the memory allocator will usually link the freed blocks together in a free list by writing pointers to the next free block in the block itself. An external free list pointer points to the first block in the free list. When a new block of memory is requested, the allocator will generally scan the free list looking for a free block of suitable size and delete it from the free list (relinking the free list around the deleted block).

One of the components of an operating system is the memory management module. This module maintains a list, which consists of unused memory cells. This list very often requires the operations to be performed on the list, such as `insert`, `delete`, and `search` (traversal). Such a list implemented as a linked organization is called the *list of available space*, *free storage list*, or the *free pool*. Suppose some memory block is freed by the program. The space available can be used for future use. One way to do so is to add the blocks in the free pool. For good memory utilization, the operating system periodically collects all the free blocks and inserts into the free pool. Any technique that does this collection is called *garbage collection*. Garbage collection usually takes place in two phases. In general, garbage collection takes place when either overflow or underflow occurs

Overflow Some times, a new data node is to be inserted into data structure, but there is no available space, that is, free pool is empty. This situation is called *overfl ow*.

Underfl ow This refers to the situation where the programmer wants to delete a node from the empty list.

STACK ADT BY USING LINKED LIST:

```
#include<iostream>
using  namespace   std;
class   node
{
public:
int    data;
node   *link;
};
class    stack
{
private:
node    *top;
int    Size;
int    isempty();
public:
stack()
{
top = NULL;
Size = 0;
}
int    gettop();
int     pop();
void    push( int Element);
void    display();
};
int   stack :: isempty()
{
if(top == NULL)
return    1;
else
return    0;
}
int    stack :: gettop()
{
if(!isempty())
return(top->data);
}
void   stack :: push(int value)
{
node    * newnode;
newnode = new    node;
newnode->data = value;
newnode->link = NULL;
newnode->link = top;
top = newnode;
}
int    stack :: pop()
{
node    * tmp = top;
int    data = top->data;
if(!isempty())
{
top = top->link;
delete    tmp;
return(data);
}
}
void    stack :: display()
{
node    *temp = top;
if(temp == NULL)
cout << "Empty List";
else
{
while(temp != NULL)
{
cout << temp->data << "\t";
temp = temp->link;
}
}
cout << endl;
}
int     main()
{
stack     S;
S.push(5);
S.push(6);
S.display();
cout << "top element is "<<S.gettop()<<endl;
cout << "deleted element"<<S.pop()<<endl;
S.display();
S.push(7);
S.display();
}
```

QUEUE ADT BY USING LINKED LIST

```
#include<iostream>
using  namespace  std;
class  node
{
public:
int  data;
node  *link;
};
class  queue
{
node  *front, *rear;
int  isempty();
public:
queue()
{
front = rear= NULL;
}
void  add( int element);
int  del();
int  getfront();
void  display();
};
int  queue :: isempty()
{
if(front == NULL)
return   1;
else
return  0;
}
int  queue :: getfront()
{
if(!isempty())
return(front->data);
}
Void  queue :: add(int x)
{
node   *newnode;
newnode = new     node;
newnode->data = x;
newnode->link = NULL;
if(rear == NULL)
{
front = newnode;
rear = newnode;
}
else
{
rear->link = newnode;
rear = newnode;
}
}
int   queue :: del()
{
int   temp;
node  *current = NULL;
if(!isempty())
{
temp = front->data;
current = front;
front = front->link;
delete current;
if(front == NULL)
rear = NULL;
return(temp);
}
}
void   queue::display()
{
            node    *temp= front;
            if(temp==NULL)
            cout<<"queue is empty";
            else
            {
while(temp != NULL)
{
cout << temp->data << "\t";
temp = temp->link;
}
}
cout << endl;
}
int     main()
{
queue     q;
q.add(11);
q.add(12);
q.add(13);
q.display();
cout <<"deleted element" <<q.del() << endl;
q.display();
q.add(14);
q.display();
cout << "deleted element"<<q.del() << endl;
q.display();
cout <<"front element is"<< q.getfront() <<
endl;

}
```