Chapter-4
Static and Dynamic List (8-hours)

Content

- Introduction to List
- Definition and array implementation of lists
- Queue as a List (Refer Chapter 3)
- Link List Definition and Link List as ADT
- Dynamic implementation
- Basic operation on linked list
- Doubly linked list and its advantages
- Implementation of doubly linked list
- Linked list implementation of stacks and queues.

Introduction to List

A **list** is a collection of homogeneous set of elements or objects. If the size of list is not fixed at compile time and grow or shrink at runtime then it is called **static list** however if the size of the list is fixed at compile time and grow or shrink at runtime then it is called **dynamic list.**
A list is said to be empty when it contains no elements. The number of elements currently stored is called the length of the list. The beginning of the list is called head and the end of the list is called the tail.
Example: Stack and queue are special type of list which can be implemented using array or linked list.

Operations on List /List as an ADT
Let L be a list and p indicates the position of list L, then the following basic operations can be performed on the list L.

- Create (L): Create a list L.
- Insert(X,P,L): Insert X at position P in the list L.
- Find(X,L): Return the position of first occurrence of element X in List L.
- Retrieve(P,L): Return the element at position P in List L.
- Delete(P,L): Delete the element at position P of list L.
- Next(P,L): Return the element at position P+1 in list L.
- Previous(P,L): Return the element at position P-1 in list L.
- MakeEmpty(L): Make the List L an empty list.
- First(L): Return the first position on list L.
- Print(L): Print the elements of list L.

Implementation of List

There are two ways to implement the list

1. Contiguous List-Array implementation of list (Static)
2. Linked list-Pointer implementation of list (Dynamic)

1. Array Implementation of List

An **array** is a collection of elements of same type placed in contiguous memory location and can be accessed individually using index to a unique given name. It means array can contain one type of data only, either all integer, all characters or all floating points numbers. An array can be single dimensional or multi-dimensional.
In array implementation of list, we keep the elements of list in an array and use and integer to count the number of elements in the list. The counter integer can also be used to locate the end of the list within that array. Elements of the list can be accessed with an index.

Advantage of array implementation

1. Random access is possible.
2. Array implementation is easier as compared to linked list implementation.

Disadvantage of array implementation

1. Elements in an array are always stored in contiguous memory so inserting and deleting an element in an array may require all elements to be shifted.
2. The size of array is fixed, so it does not provide good memory utilization.

The following C program shows how list can be implemented using an array.

```
#include<stdio.h>
 #include<conio.h>
 void main()
 {
        clrscr();
        int a[5]={1,2,3,4,5};
        int b[5]={2,4,6,8,10};
        int c[5];
        for( int i=0;i<=4;i++)
        {
              c[i]=a[i]+b[i];
        }

        for(int j=0;j<=4;j++)
        {
             printf("%d\t",c[j]);
        }
        getch();
}
```

i. Algorithm to insert a new element in a contiguous list

Consider we have an array A[max] of size max whose upper bound is represented by ub. It is assumed that ub is -1 for an empty array.

1. If ub = max-1
   a. Display "full"
   b. Exit
2. Read position from user
3. If position > ub+1
   a. Display "out of range"
   b. Exit
4. Read the data item to be inserted.
5. Shift element right

```
T = ub;
While (T>=position)
{
A[T+1] =A[T];
T=T-1;
}
```

6. ub=ub +1;
7. A[pos] =item;
8. Exit

ii. Algorithm for deletion of an item in a contiguous list

1. If ub = -1
   a. Display "array is empty"
   b. Exit
2. Read position to be deleted.
3. If(position >=0 && position <=ub)
   a. Shift element left

```
T=position;
While (T< ub)
{
A[T] =A[T+1];
T=T+1
}
```

   b. ub = ub -1;

Else
   a. Display "index out of range"

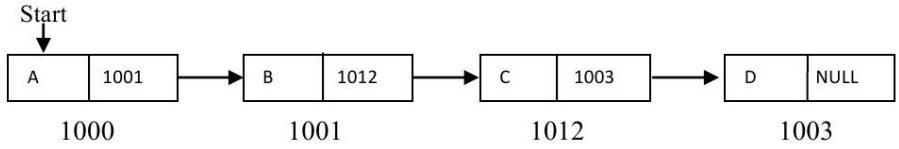
4. Exit

2. Linked List implementation of List

A linked list is a linear collection of data items called nodes, where each node consists of two parts.

a. Info: The actual element to be stored in the list. It is also called as data field.
b. Link: one or two links that point to the next and previous node in the list. It is also called as next or pointer field.

So a linked list consists of chain of nodes in which each node consists of data as well link to the next/previous node. The structure of linked list having 4 nodes is shown in the figure below.



Advantages

- Linked list are dynamic data structure i.e. they can grow or shrink during the execution of program.
- It provides easier and efficient insertions and deletions because we don't have to shift any elements in the list to insert and delete an element.
- Data can store non-contiguous memory blocks.
- Many more complex applications can be easily carried out using linked list.

Disadvantages

- Linked list require extra space to store address of next node.
- Accessing of arbitrary data items is little bit tedious and time consuming.
- Linked list allow only sequential access to its data member.
- Difficult to implement.

Operations on Linked List / Linked List as ADT

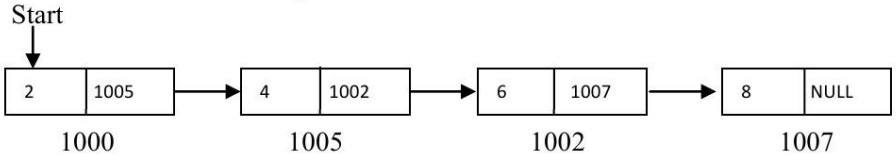
The basic operations to be performed on the linked lists are as follows.

- **Creation:** This operation is used to create a linked list.
- **Insertion:** This operation is used to insert a new node in a linked list in a specified position. Any new node may be inserted
  a. At the beginning of the linked list.
  b. At the end of the linked list.
  c. At the specified position in a linked list.
- **Deletion:** This operation is used to delete a node from the linked list. A node may be deleted from
  a. At the beginning of the linked list.
  b. At the end of the linked list.
  c. At the specified position in a linked list.
- **Traversing:** This operation is a process of going through all the nodes of the linked list from one end to the other end. Traversing can be either forward or backward.
- **Searching:** This operation is used to find an element in a linked list. Search is said to be successful if the desired element is found in the linked list otherwise unsuccessful.
- **Concatenation:** This operation is a process of joining second linked list to the end of the first linked list.

Example: Write a C program to create a linked list as below.

NOTE:

1. The malloc() function finds a free block in memory, based on the memory block size requested and then returns a pointer to it.
2. The free() function frees the allocated space.




```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
  void create();
  void traverseall();
```

```
struct node
{
        int info;
        node *next;
};

node *start; // external pointer variable

void main()
{
          clrscr();
           create();
           traverseall();
           getch();
}

void create()
{
        node *ptr,*tmp;
        ptr=(node *)malloc (sizeof(node)); // allocate memory
        ptr->info=2;
        start=ptr;

        tmp=(node *)malloc (sizeof( node));
        tmp->info=4;
        ptr->next=tmp;
        ptr=tmp;

        tmp=(node *)malloc (sizeof(node));
        tmp->info=6;
        ptr->next=tmp;
        ptr=tmp;

        tmp=(node *)malloc (sizeof(node));
        tmp->info=8;
        ptr->next=tmp;
        ptr=tmp;

        ptr->next=NULL; // End of list
}


void traverseall()
{
        node *ptr;

        ptr=start;
        //printf("%d",ptr->info);

        //ptr=ptr->next;
        //printf("%d",ptr->info);

        //ptr=ptr->next;
        //printf("%d",ptr->info);

         // ptr=ptr->next;
        //printf("%d",ptr->info);
```

```
        while(ptr!=NULL)
        {
                printf("%d",ptr->info);
                ptr=ptr->next;
        }
}
```

Example: Write a C program to create and display a linked list for N numbers

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void create();
void traverseall();


struct node
{
        int info;
        node *next;
};



node *start;



void main()
{
          clrscr();
           create();
           traverseall();
           getch();
}




void create()
    {
        int n;
        node *ptr,*tmp;
        printf("Enter the number of nodes");
        scanf("%d",&n);
        printf("Enter the first node information");
        ptr=(node *)malloc (sizeof(node));
        scanf("%d",&ptr->info);
        start=ptr;
        for (int i=0;i<n-1;i++)
          {
                printf("Enter the node information");
                tmp=(struct node *)malloc (sizeof(struct node));
                scanf("%d",&tmp->info);
                ptr->next=tmp;
                ptr=tmp;
          }

        ptr->next=NULL;  // End of list
    }
```

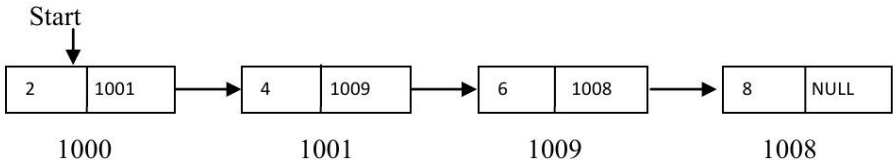
```
void traverseall()
   {
        node *ptr;
        ptr=start;
        while(ptr!=NULL)
        {
                printf("%d",ptr->info);
                ptr=ptr->next;
        }
   }
```

Types of Linked List

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List
4. Circular Doubly Linked List

1. Singly Linked List

In linear or singly linked list, there is only one link in each node which points to the next node in the list. Each node is divided into two parts. First part contain the information of the element and second part contain the link field i.e. address of next node in the list.



The linked list is accessed from an external pointer variable i.e. Start which points the first node in the list. The last node contain NULL value in the link field which indicates end of the list.

Operations on Singly Linked List

1. Creation of Singly Linked List

Algorithm to create linked list of n nodes

1. Define a node using self referential structure.

```
struct node
  {
        int info;
        node *next;
  };
```

2. Create structure type pointer variable start, ptr and tmp as

```
node *start,*ptr,*tmp;
```

3. Input the number of nodes to be created and store the value in variable N.
4. Create first node as below
   a. Read data
   b. Allocate memory and store information

```
ptr=(node *)malloc (sizeof(node));
ptr->info=data);
```

   c. Make it the start node

```
start=ptr;
```

5. Repeat the following steps for N-1 nodes.
   a. Read data
   b. Allocate memory and store information

```
tmp=(node *)malloc (sizeof(node));
tmp->info=data);
ptr->next=tmp;
```

   c. Make it the current node

```
ptr=tmp;
```

6. make the last node link value to NULL.

```
ptr->next=NULL;
```

2. Traversing a singly linked list

Traversing a linked list refers to the process of visiting each node of the list starting from the beginning. Singly linked list can be traversed only in one direction i.e. forward direction.

Algorithm:

1. Make current node point to the first node in the list.
2. Repeat step 3 and 4 until current node become null.
3. Display the information contained in the node marked as current node.
4. Make the current node point to next node in sequence.

C implementation:

```
void traverseall()
  {
        node *ptr;
        ptr=start;    // current node pointing to first node i.e. start points to first node
        while(ptr!=NULL)
        {
                printf("%d",ptr->info);
                ptr=ptr->next;
        }
  }
```

3. Counting the elements in a singly linked list

Algorithm:

1. Initialize count to 0;
2. Make current node point to first node in the list.
3. Repeat step 4 and 5 until the current node become null.
4. Increment the value of count by 1.
5. Make the current node point to next node in sequence.
6. Display count as the number of elements in the list.

C implementation:

```
void countNodes()
  {
        int count=0;
        node *ptr;
        ptr=start;
        while(ptr!=NULL)
        {
                count++;
                ptr=ptr->next;
        }
       printf("The number of nodes are %d", count);

  }
```

4. Finding an element in a singly linked list

Algorithm:

1. Input the key to be searched.
2. Make current node point to the first node in the list.
3. Repeat step 4 and 5 until current node become null.
4. Display "Search is successful" if the information in current node is equal to key and exit.
5. Make the current node point to next node in sequence.
6. Display "Search is unsuccessful"
7. Exit

C implementation:

```
Void search ()
 {
        int key;
        printf("Enter the key");
        scanf("%d", &key);
        node *ptr; // current node
        ptr=start;  // current node point to first node
        while(ptr!=NULL)
        {
                if(ptr->info==key)
                {
                printf("Search is successful");
                return;
                }
                ptr=ptr->next;
        }
       printf("Search is unsuccessful");
}
```

5. Inserting nodes in a singly linked list

A new node may be inserted

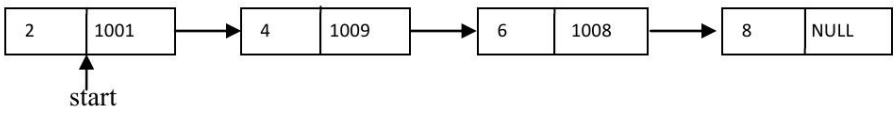
i. at the beginning of the linked list.
ii. at the end of linked list
iii. at the specified position in the linked list (between two nodes in the list)

i. Algorithm to insert a node at the beginning of the singly linked list

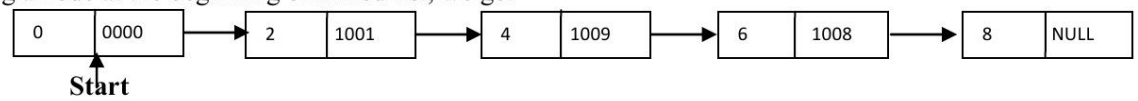
1. Allocate memory for the new node using malloc function.
   ptr=(node *)malloc (sizeof(node));
2. Assign value to the data field of the new node.
   ptr->info=value;
3. Make the next field of the new node point to the first node in the list
   ptr->next=start;
4. Make start point to new node.
   start=ptr;
5. Exit

In short, the algorithm can be written as

1. **Allocate memory for the new node.**
2. **Assign the value to the data field of the new node.**
3. **Make the next field of the new node point to the first node in the list.**
4. **Make start point to new node.**



Inserting a node at the beginning of linked list, we get



C implementation:

```
void insertatbegin(int item)
{
        Node *ptr;
        ptr=(node *)malloc (sizeof(node));
        ptr->info==item;
        if(Start==NULL)
        {
            ptr->next=NULL;
```

```
        }
        else
        {
            ptr->next=Start;
        }
            Start=ptr;

    }
```

ii. Algorithm to insert a node at the end of singly linked list

1. Allocate memory for the new node using malloc function.
   ptr=(node *)malloc (sizeof(node));
2. Assign value to the data field of the new node.
   ptr->info=value;
3. Make the next field of the new node point to NULL.
   ptr->next=NULL;
4. If start is NULL then
   a. Make start point to the new node
      start=ptr;
   b. Exit
5. Locate the last node in the list, and mark it as current node. To locate the last node in the list, perform the following steps.
   a. Make the first node as current node.
   b. Repeat step c until the successor of current node becomes NULL.
   c. Make current node point to next node in sequence.

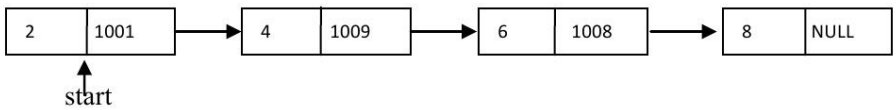
```
node *temp; // current node
temp=start;  // current node point to first node
while (temp->next!=NULL) // this loop will stuck in  last node with temp pointing to last node.
{
temp=temp->next;
}
```

6. Make the next field of current node point to the new node.
   temp->next=ptr;
7. Exit

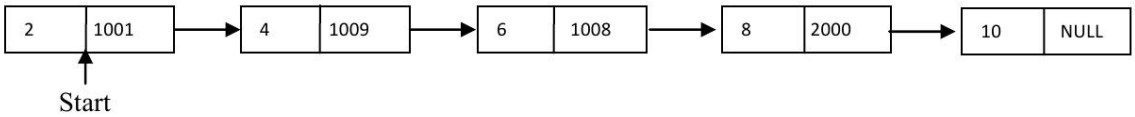
In short, the algorithm can be written as

1. **Allocate memory for the new node.**
2. **Assign the value to the data field of the new node.**
3. **Make the next field of the new node point to null.**
4. **If the list is empty, then make start point to new node and exit.**
5. **If the list is not empty, then got to the last node and then make the last node point to new node.**
6. **exit**

Example:



Inserting a node at the end of linked list, we get



C-Implementation:

```
Void insertatend(int item)
{
        Struct node *Ptr, *Tmp;
        Ptr=(node *)malloc (sizeof(node));
        Ptr->info = item;
        Ptr->next = NULL;
        if(Start == NULL)
```

```
        {
                Start =Ptr;
        }
        Else
        {
                Tmp = Start;
                While (Tmp - > next != NULL)
                {
                Tmp = tmp->next;
                }
                Tmp->next =ptr;
        }

}
```

iii. Algorithm to insert a node between two nodes in the list or inserting a node at the specified position

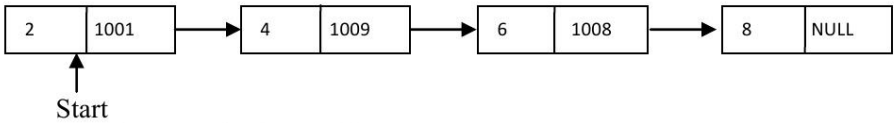
1. Input Data and position **after** which the node is to be inserted.
2. If the list is empty i.e. start=null, then display "empty list and exit".
3. Initialize temp = start and k=0
4. Repeat step 3 while (k <pos)
   a. temp =temp->next
   b. if (temp == NULL) then display "node in the list less than the position" and exit
   c. k = k+1
5. create a new node
6. new node -> info = Data
7. new node -> next = temp->next
8. temp->next=new node
9. exit

In short, the algorithm can be written as
Assuming that the new node is to be inserted between node A and node B

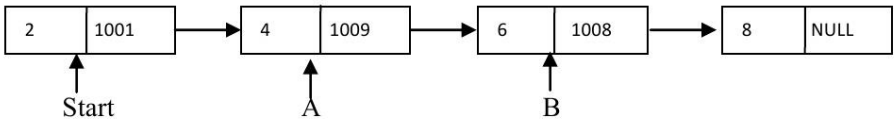
1. **Allocate memory for the new node.**
2. **Assign value to the data field of the new node.**
3. **Make the next field of the new node to point to node B.**
4. **Make the next field of the node A to point to new node.**

Example:

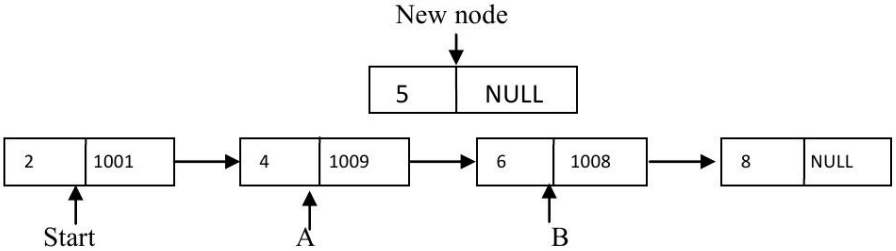
1. Let the elements in the linked list be



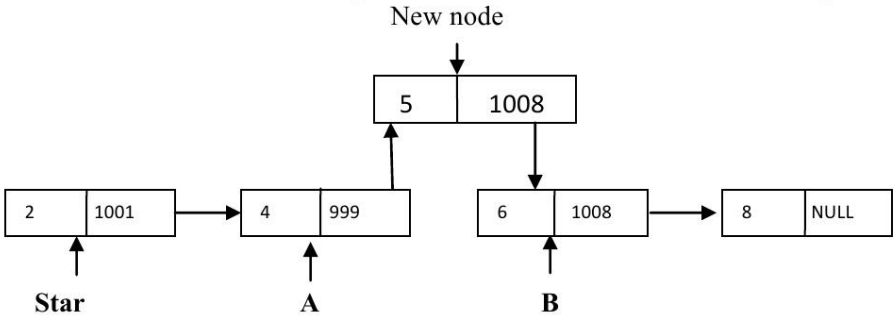
2. Let new element to be inserted is 5
3. Identifying the nodes between which the new node is to be inserted. Marking them as A and B we get



4. Allocate memory and assign value to the data field of the new node



5. Make the next field of new node point to current node (B) and next field of previous (A) point to new node



C-Implementation:

```
void insert_specified(int item,int pos)
{
        Node *ptr, *tmp;
        int k=0;
        tmp=start;
        for(k=0;k<pos;k++)
        {
                tmp=tmp->next;
                if(tmp==NULL)
                {
                        printf("Node in the list at less than the position");
                        return;
                }
        }
        Ptr=(node *)malloc (sizeof(node));
        Ptr->info=item;
        Ptr->next=tmp->next;
        tmp->next=ptr;

}
```

5. Deleting a node from a singly linked list

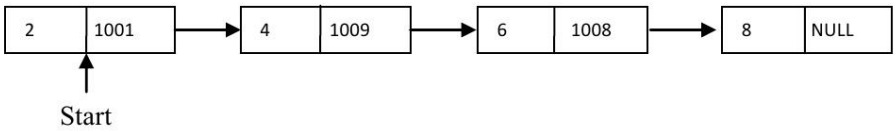
There are three cases of deleting a node in singly linked list.

i. Deleting the first node.
ii. Deleting the last node.
iii. Deleting the node from specified position.

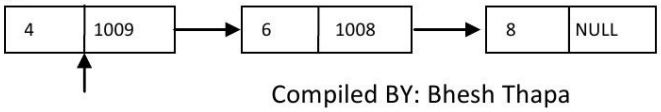
i. Algorithm to delete the first node

1. If the list is empty then display the message Deletion is not possible.
2. Else, move the start pointer to the second node.
3. Free the memory associated with first node.

Example:



After deleting the first node



Start

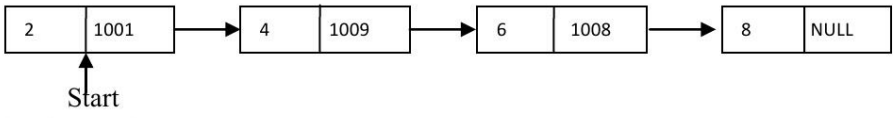
C-Implementation

```
void delete_begin()
{
        Node *ptr;
        If(start==NULL)
        {
         printf("Deletion is not possible");
         return;
        }
        else
        {
         ptr=start;
         start=start->next;
         free (ptr);
        }

}
```

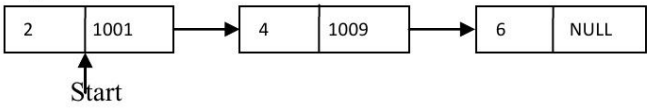
ii. Algorithm to delete the last node

1. If the linked list is empty then display message "Deletion is not possible".
2. Else, go on traversing the list till the last node.
3. Make the second last node as the last node i.e. set next field of second last node to NULL.
4. Free the memory associated with the last node.

Examle:



After deleting last node



C-Implementation

```
void delete_last()
{
        Node *ptr,*tmp;
        if(start == NULL)
        {
             Printf("Deletion is not possible");
             return;
        }
         else if(start->next ==NULL)
        {
                  ptr =start;
                  start=NULL;
                  free(ptr);
        }
        else
        {
                  tmp=start;
                  ptr=start->next;
```

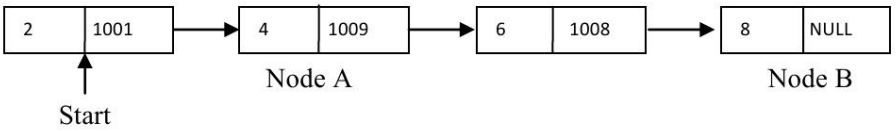
```
                while(ptr->next !=NULL)
                {
                        tmp=ptr;
                        ptr=ptr->next;
                }
                tmp->next=NULL;
                free(ptr);
        }


    }
```

iii. Deleting the node from specified position

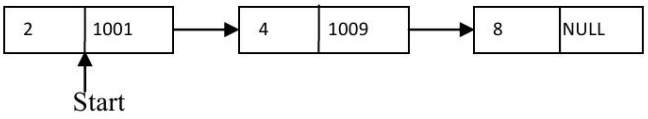
Assuming a node is to be deleted between Node A and Node B.

1. If the linked list is empty then display message "Deletion is not possible".
2. Else, make the next field of Node A point to Node B.
3. Free the node between Node A and Node B.

Example:



Suppose the node to be deleted is third one i.e. 6 then after deletion



C-Implementation

```
Void delete_specified()
    {
    Node *ptr, *tmp;
    int num;
    printf("Enter which elements you want to delete?");
    scanf("%d", &num);
    ptr=start;
    if(start==NULL)
    {
      printf("Delete not possible");
      return;
    }
    else
    {
        tmp=ptr;
        while(ptr!=NULL)
        {
            if(ptr->info==num)
            {
            tmp->next=ptr->next;
            if(ptr==start) //if only one node
            {
```

```
            start=start->next;
            }
            free(ptr);
            return;
            }
            tmp=ptr;
            ptr=ptr->next;
        }
    }

    }
```

Implementation Stack Using Linked List

Stack is an ordered collection of items in which items may be inserted or deleted at only one end. The end is called the top of stack. In stack the last item added will be the first item to be deleted so it is also called as last in first out (LIFO) data structure.

Advantage of linked list implementation of stack over array implementation

1. The stack doesn't need to be of fixed size so there can be any number of elements in the stack.
2. The insertion and deletion operation do not involve more data movements.
3. Better memory utilization

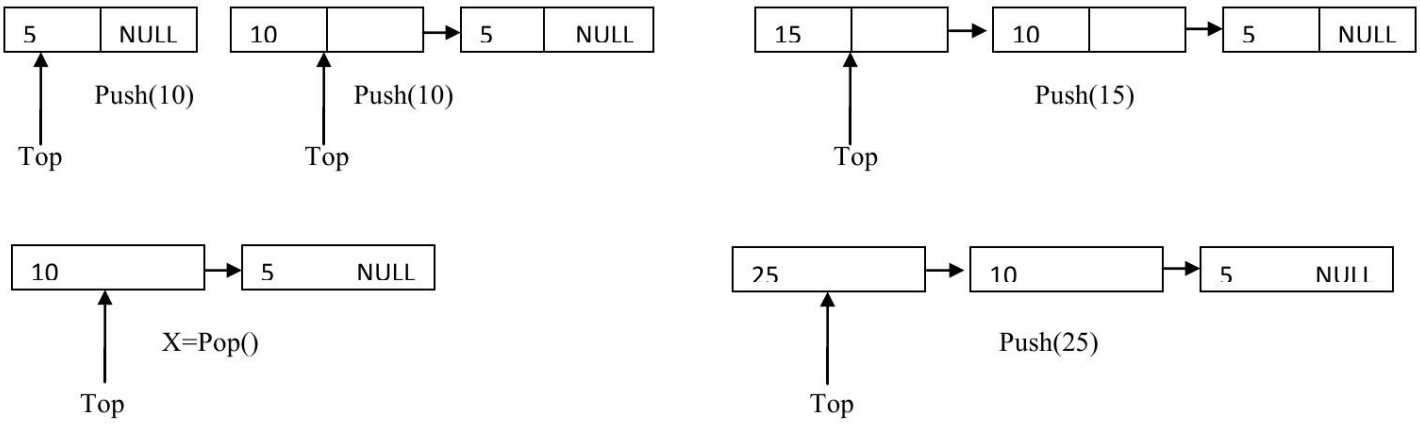
A node in a stack can be define using self-referential structure as below

```
struct Node
{
int info;
Node *next;
};
Node *top=NULL;
```

Operations on stack

1. Push() operation: The process of adding a new element to the top of stack is called push operation. To implement push, we create a new node in the list and attach it as the new first element.
2. Pop() operation: the process of deleting and existing element from the top of stack is called pop operation. To implement pop, we advance **Top** of the stack to the second item in the list.

These operations can be explained graphically as below.



i. Algorithm for Push Operation

Suppose Top is a pointer, which is pointing towards the topmost element of the stack. Top is null when stack is empty. Data is the data item to be pushed.

1. Input the data to be pushed.
2. Create a New node.
3. Newnode->info=data
4. Newnode ->next=top
5. Top=Newnode
6. Exit

C-Implementation

```
Void push(int data)
{
    Node *ptr;
    ptr=(node*)malloc(sizeof(node));
    ptr->info=data;
    ptr->next=top;
    top=ptr;
}
```

ii. Algorithm for Pop operation

1. if(top==NULL)
   a. Display "The stack is empty"
   b. Exit
2. Temp=top;
3. Display "The popped element is top->info":
4. top=top->next;
5. temp->next=NULL;
6. free the temp node;
7. exit

C-Implementation

```
void Pop()
{
   Node *tmp;
  tmp=top;
  if(tmp==NULL)
  {
   printf("Stack is empty");
  return;
  }
  else
  {
  Printf("The popped element is %d",top->info);
  top=top->next;
  free(tmp);
}
```

Implementation Queue using Linked list

Queue is an ordered collection of items from which items may be deleted at one end and inserted at other end. The end from which the item is inserted is called rear end and the end from which the item is deleted is called front end.

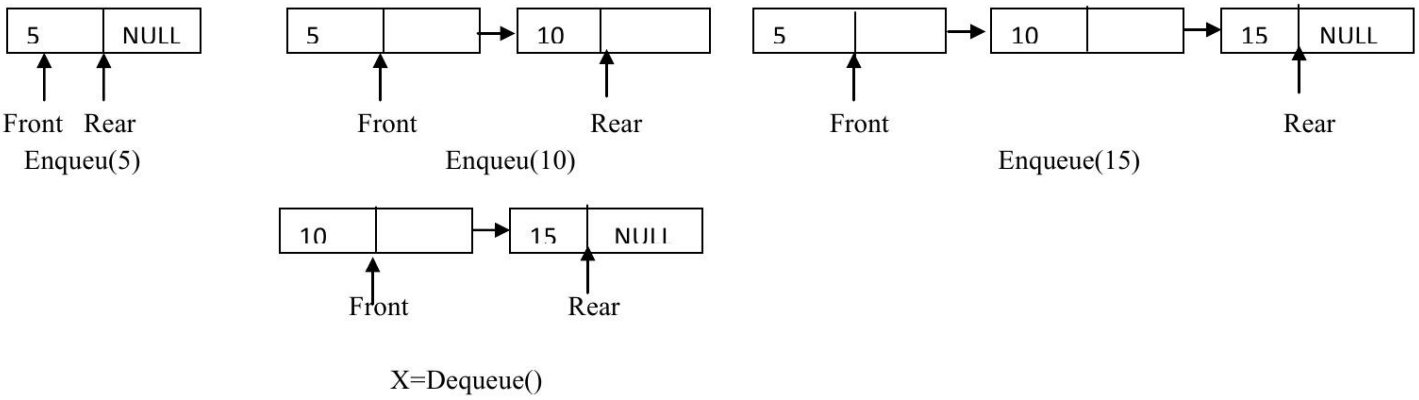
Advantage of linked list implementation of queue over array implementation

1. The queue doesn't need to be of fixed size so there can be any number of elements in the stack.
2. The insertion and deletion operation do not involve more data movements.
3. Better memory utilization

A node in a stack can be define using self-referential structure as below

```
struct Node
{
int info;
Node *next;
};
Node * front=NULL;
Node * rear=NULL;
```

These operations can be explained graphically as below.



Operations on queue

1. Enqueue() Operation: The process of adding a new element at the rear end of queue is called enqueue operation. To implement enque, we create a new node in the list and attach it as the new last element.
2. Dequeue() operation: The process of deleting and existing element from the front end of queue is called dequeue operation. To implement dequeue, we advance **Front** of the queue to the second item in the list.

i. Algorithm for Enqueue() operation

Let us assume that Rear is a pointer in queue, where the new elements are added. Front is a pointer, which is pointing to the queue where the elements are deleted. Data is an element to be inserted.

a. Input the Data to be inserted.
b. Create a New node.
c. Newnode->info=Data
d. Newnode->next=NULL
e. If(rear==NULL)
    i. Front=Newnode;
    ii. Rear=Newnode;
    Else
    i. rear->next=Newnode;
    ii. rear=Newnode;
f. Exit

C-Implementation

```
Void enqueue(int Data)
{
    Node *ptr;
    ptr=(Node*)malloc(sizeof(Node));
    ptr->info=Data;
```

```
        ptr->next=NULL;
        if(rear==NULL)
        {
                front=ptr;
                rear=ptr;
        }
        Else
        {
                rear->next=ptr;
                rear=ptr;
        }

    }
```

ii. Algorithm for dequeue() operation

Let us assume that Rear is a pointer in queue, where the new elements are added. Front is a pointer, which is pointing to the queue where the elements are deleted. Data is an element to be deleted from queue.

a. if(front==NULL)
   i. display "queue is empty "
   ii. exit
b. Data=front->info;
c. Display "The popped item is Data";
d. temp=front;
e. front=front->next;
f. free(temp)
g. Exit

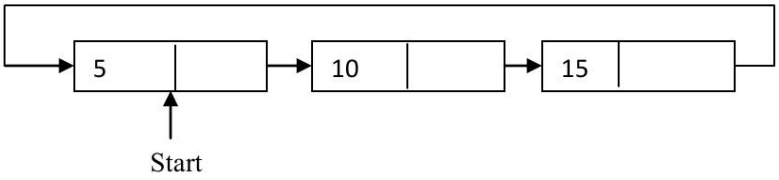
C-Implementation

```
    Void dequeue()
    {
        Node *temp;
        If(front==NULL)
        {
                printf("Queue is empty");
                return;
        }
        Else
        {
                Data=front->info;
                printf("The dequeued item is %d",Data);
                temp=front;
                front=front->next;
                free(temp);
        }

    }
```

Circular Linked List

It is just a singly linked list in which the link field(next field) of the last node contains the address of the first node of the list i.e. the link field of the last node doesn't point to NULL. So a circular linked list has no end. Circular linked list also make our

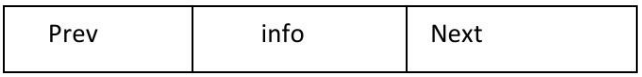
implementation easier because they eliminate the boundary conditions associated with the beginning and end of list, thus eliminating the special case codes required to handle these boundary conditions.



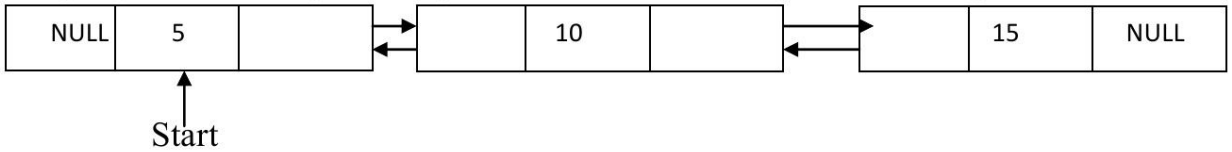
Doubly Linked List

One of the most disadvantages of singly linked list and circular linked list is the inability to traverse the list in the backward direction. In most of the real world application, it is necessary to traverse the list in both forward and backward directions. This problem can be overcome by a doubly linked list.

A doubly linked list in one in which all nodes are linked together by multiple number of link which help in accessing both the successor and predecessor nodes from the given node position. It provides bidirectional traversing. Every node in the doubly lined list has three fields: left pointer, info and right pointer as shown below.



The prev pointer is used to point to its predecessor node and the next pointer is used to point its successor node.



Doubly linked list can be represented by following declaration

```
struct Node
{
int info;
Node *prev;
Node *next;
};
```

Advantages

1. Insertion and deletion are simple as compared to other linked list.
2. Bidirectional traversing helps in efficient and easy availability of nodes.

Disadvantages

1. Difficult to implement than singly linked list and circular linked list.
2. Require more memory for extra pointer.

Insertion and deletion operation on doubly linked list

i. Insertion operation:

Any new node may be inserte

a. At the beginning of the linked list.
b. At the end of the linked list.
c. At the specified position in a linked list.

a. Algorithm for inserting a node at the beginning of the linked list

i. Allocate memory for the new node.
ii. Assign value to the data field for new node.
iii. Make the prev and right field of new node point to NULL.
iv. If the list is not empty, make the next field of new node point to start node of the list and prev field of start node point to new node.

.

b. Algorithm for inserting a node at end of linked list

i. Allocate memory for the new node.
ii. Assign value to the data field for the new node.
iii. Make prev and right field of new node point to NULL.
iv. If the list is not empty, then traverse the list till the last and make the next field of last node point to the new node and the prev field of new node point to the last node.

c. Algorithm for inserting a node at specified position.

Assuming that the new node is to be inserted between node A and node B

1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node to point to node B and prev field of node B to point new node.
4. Make the next field of the node A to point to new node and prev field of new node point to node A.

ii. Deletion Operation:

There are three cases of deleting a node in singly linked list.

a. Deleting the first node.
b. Deleting the last node.
c. Deleting the node from specified position.

a. Deleting a node from the beginning

i. If the list is empty then display the message "Empty List";
ii. Else, make the start pointer point to the second node and make second node prev field point to NULL.
iii. Free the memory associated with first node.

b. Deleting a node from the end

i. If the list is empty then display the message "Empty-List"
ii. Else, go on traversing the list till the last node.
iii. Make the second last node as the last node i.e. set next field of second last node to NULL.
iv. Free the memory associated with last node.

c. Deleting a node from specified position

Assuming a node is to be deleted between Node A and Node B.

i. If the list is empty then display the message "Deletion is not possible".
ii. Else, make the next field of Node A point to Node B, and prev field of Node B point to Node A.
iii. Free the node between Node A and Node B.

Implementation of Double Ended Queue(Dequee) using linked list

A Double Ended queue written as Dequee and pronounced Dek, is a list of emements in which insertion and deletion are performed from both end. It can be implemented both as stack and queue. If item are inserted and deleted from same end then dequee is said to be implemented as stack but if items are inserted from one end and deleted from another end then dequee is said to be implemented as queue. The possible operations that can be performed on dequee are

1. Inserting an element at rear end of dequee
2. Deleting an element at front end of dequee
2. Inserting an element at front end of dequee
4. Deleting an element at rear end of dequee

1. Algorithm for insertion from rear end

Let us assume that front and Rear is the pointers in Dequee, where the new elements can be added or deleted from either end. Rear is a pointer, which is pointing to the queue where the elements are inserted. Data is an element to be inserted.

a. Input the Data to be inserted.
b. Create a New node.
c. Newnode->info=Data
d. Newnode->next=NULL

e. Newnode->prev=NULL
f. If(rear==NULL)
    i. Front=Newnode;
    ii. Rear=Newnode;
    Else
    i. rear->next=Newnode;
    ii. Newnode->prev=rear;
    iii. rear=Newnode;
g. Exit

2. Algorithm for deletion from front end

Let us assume that front and Rear is the pointers in Dequee, where the new elements can be added or deleted from either end. Front is a pointer, which is pointing to the queue where the elements are deleted. Data is an element to be deleted from queue.

a. if(front==NULL)
    i. display "queue is empty "
    ii. rear=NULL.
    iii. Exit
b. Data=front->info;
c. Display "The popped item is Data";
d. temp=front;
e. front=front->next;
f. fron->prev=NULL;
g. free(temp)
h. Exit

3. Algorithm for insertion from front end

Let us assume that front and Rear is the pointers in Dequee, where the new elements can be added or deleted from either end. . Front is a pointer, which is pointing to the queue where the elements are inserted. Data is an element to be inserted.

a. Input the Data to be inserted.
b. Create a New node.
c. Newnode->info=Data
d. Newnode->next=NULL
e. Newnode->prev=NULL
f. If(front==NULL)
    i. Front=Newnode;
    ii. Rear=Newnode;
    Else
    i. front->prev=Newnode;
    ii. ptr->next=front;
    iii. front=Newnode;
g. Exit

4. Algorithm for deletion from rear end

Let us assume that front and Rear is the pointers in Dequee, where the new elements can be added or deleted from either end. Rear is a pointer, which is pointing to the queue where the elements are deleted. Data is an element to be deleted from queue.

i. if(rear==NULL)
    iv. display "queue is empty "
    v. Front=NULL.
    vi. Exit
j. Data=rear->info;
k. Display "The popped item is Data";
l. temp=rear;

m. rear=rear->prev;
n. rear->next=NULL;
o. free(temp)
p. Exit

C-Implementation:

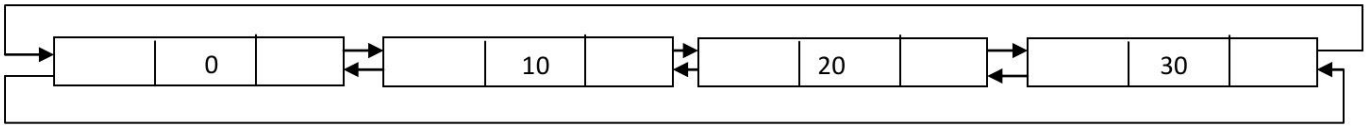
```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void insertrear();
void deletefront();
void insertfront();
void deleterear();
struct node
{
        node *prev;
        int info;
        node *next;
};
node *front=NULL, *rear=NULL;
void main()
{
        int n;
        clrscr();
        printf("Enter the choice");
        printf("\n1: insert from rear end");
        printf("\n2: delete from front end");
        printf("\n3: insert from front end");
        printf("\n4: delete from rear end");
        scanf("%d",&n);
        switch(n)
        {
                case 1:
                insertrear();
                break;
                case 2:
                deletefront();
                break;
                case 3:
                insertfront();
                break;
                case 4:
                deleterear();
                break;
        }

}
void insertrear()
{
        int data;
        node *ptr;
        printf("\nEnter the item to be inserted");
        scanf("%d",&data);
```

```
        ptr=(node *)malloc(sizeof(node));
        ptr->info=data;
        ptr->prev=NULL;
        ptr->next=NULL;
        if(rear==NULL)
        {
                front=ptr;
                rear=ptr;
        }
        else
        {
                 rear->next=ptr;
                 ptr->prev=rear;
                 rear=ptr;
        }
        printf("\nOne item inserted");
        getch();
        main();

}
void deletefront()
{
        int data;
        if(front==NULL)
        {
                printf("\nDequee is empty");
                rear=NULL;
        }
        else
        {
                node *temp;
                data=front->info;
                printf("The item deleted is %d",data);
                temp=front;
                front=front->next;
                fron->prev=NULL;
                free(temp);
        }
        getch();
        main();
}
void insertfront()
{
          int data;
        node *ptr;
        printf("\nEnter the item to be inserted");
        scanf("%d",&data);
        ptr=(node *)malloc(sizeof(node));
        ptr->info=data;
        ptr->prev=NULL;
        ptr->next=NULL;
        if(front==NULL)
        {
```

```
                front=ptr;
                rear=ptr;
        }
        else
        {
                 front->prev=ptr;
                 ptr->next=front;
                 front=ptr;
        }
        printf("\nOne item inserted");
        getch();
        main();
}
void deleterear()
{
                 int data;
                if(rear==NULL)
                {
                        printf("\nDequee is empty");
                        front=NULL;
                }
                else
                {
                        node *temp;
                        data=rear->info;
                        printf("The item deleted is %d",data);
                        temp=rear;

                        rear=rear->prev;
                        rear->next=NULL;
                        free(temp);
                }
                getch();
                main();
                 getch();
}
```

Cicular Doubly Linked List

A circular doubly linked list is the variation of Doubly linked list which has both the successor (next)and predecessors (prev)pointer in circular manner as shown in the figure below.



The objective behind considering circular doubly linked list is to simplify the insertion and deletion operations performed on doubly linked list.

Missed Topic

1. Recursive TOH program

```
#include<stdio.h>
#include<conio.h>
void transfer(int,char,char,char);
void main()
{
        int n;
        printf("Enter the number of disk");
        scanf("%d",&n);
        transfer(n,'l','r','c');  // l =left tower, r=right tower and c=center tower
        getch();
}

void transfer(int n, char from,char to ,char temp)
{
 if(n!=0)
        {
                transfer(n-1,from,temp,to);
                printf("\nmove disk %d from %c to %c",n,from,to);
                transfer(n-1,temp,to,from);
        }
 }
```

Output of above program:

```
Enter the number of disk3

move disk 1 from l to r
move disk 2 from l to c
move disk 1 from r to c
move disk 3 from l to r
move disk 1 from c to l
move disk 2 from c to r
move disk 1 from l to r_
```

2. WAP to generate Fibonacci series using recursive function where user enters the limit of series

```
#include<stdio.h>
#include<conio.h>
int fibo(int n);
void  main()
{
        int n;
        int term;
        clrscr();
        printf("Enter the limit of fibonacciseries   ");
        scanf("%d",&n);
        for(int i=1;i<=n;i++)
        {
        term=fibo(i);
        printf("\t%d",term);
        }
        getch();
}
int fibo(int n)
{
        if(n==1 || n==2)
        {
        return 1;
        }
        else
        {
        return (fibo(n-2)+fibo(n-1));
        }
        }
```

Output:

```
Enter the limit of fibonacciseries   10
1       1       2       3       5       8       13      21      34      55
```

WAP to find the sum on first n natural number using recursion.

```
#include<stdio.h>
#include<conio.h>
int sum(int);
void main()
{
        int n;
        int res;
        printf("Enter the nth number");
        scanf("%d",&n);
        res=sum(n);
        printf("The sum  is %d",res);
        getch();
}
int sum(int n)
{
```

```
        if(n==1)
        {
                return 1;
        }
        else
        {
                return (sum(n-1)+n);
        }

}
```

Output:

```
Enter the nth number 10
The sum is 55 _
```

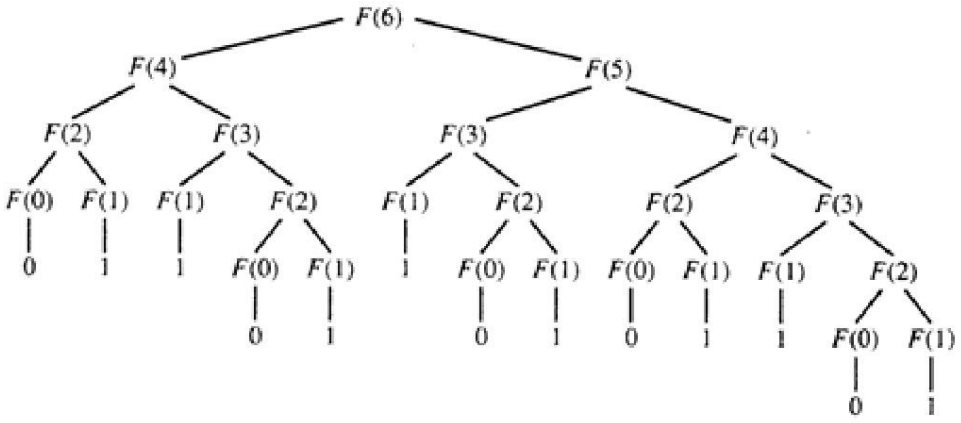
Given a recursive definition

$$\text{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{otherwise} \end{cases}$$

How many time Fib() is called for computing fib(6)?

Solution:

The tree for Fib(6) is given below.



So from the tree we see that, there are 25 calls made.