UNIT-I

.

LINEAR DATA STRUCTURES – LIST

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation –– singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operation (Insertion, Deletion, Merge, Traversal)

Abstract Data Types (ADTs)

An abstract data type is defined only by the operations that may be performed on it and by mathematical pre-conditions and constraints on the effects (and possibly cost) of those operations.

For example, an abstract stack, which is a last-in-first-out structure, could be defined by three operations: push, that inserts some data item onto the structure, pop, that extracts an item from it, and peek or top, that allows data on top of the structure to be examined without removal.

An abstract queue data structure, which is a first-in-first-out structure, would also have three operations, enqueue to join the queue; dequeue, to remove the first element from the queue; and front, in order to access and serve the first element in the queue.

An ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

Defining an abstract data type (ADT)

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Modules:

The programs can be spit in to smaller sub programs called modules. Each module is a logical unit and does a specific job or process. The size of the module is kept small by calling other modules.

Modularity has several advantages:

- Modules can be compiled separately which makes debugging process easier.
- Several modules can be implemented and executed simultaneously.
- Modules can be easily enhanced.

An ADT is a set of operations and is an extension of modular design. A useful tool for specifying the logical properties of a data type is the abstract data type. ADT refers to the basic mathematical concept that defines the data type. Eg. Objects such as list, set and graph along their operations can be viewed as ADT's. (or) Abstract Data Type is some data, associated with a set of operations and mathematical abstractions just as integer, Boolean etc.,

- Basic idea

·
  - ✓ Write once, use many.
  - ✓ Any changes are transparent to the other modules.

INTRODUCTION TO DATA STRUCTURES:

As computers become faster and faster the need for programs that can handle large amount of inputs become more acute which in turn requires choosing suitable algorithm with more efficiency, therefore the study of data structures became an important task which describes, method of organizing large amounts of data and manipulating them in a better way.

DATA:

A collection of facts, concepts, figures, observations, occurences or instructions in a formalized manner.

INFORMATION:

The meaning that is currently assigned to data by means of the conventions applied to those data(i.e. processed data)

RECORD:

Collection of related fields.

DATATYPE:

Set of elements that share common set of properties used to solve a program.

DATASTRUCTURE:

A way of organizing, storing, and retrieving data and their relationship with each other.

ALGORITHM:

An algorithm is a logical module designed to handle a specific problem relative to a particular data structure.

DESIRABLE PROPERTIES OF AN EFFECTIVE ALGORITHM:

  - it should be clear / complete and definite.
  - it should be efficient.
  - it should be concise and compact.
  - it should be less time consuming.
  - effective memory utilization.

APPLICATION OF DATA STRUCTURES:

Some of the applications of data structures are:

- operating systems

- compiler design
- statistical and numerical analysis
- database management systems
- expert systems
- network analysis

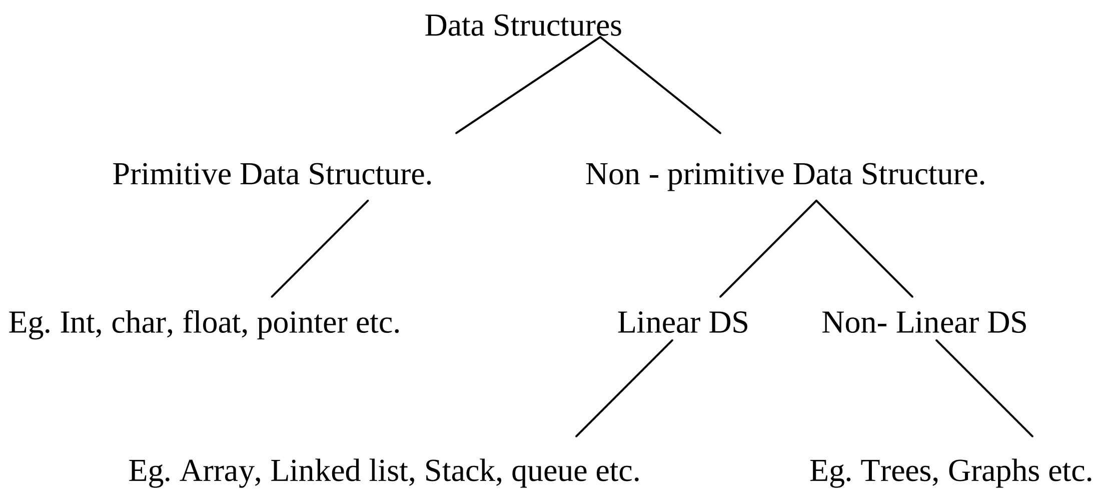
List Abstract Data Type

Data Structure is a systematic way of organizing, storing and retrieving data and their relationship with each other.

Classification of Data Structure:

There are two main types of Data Structure classification.

1. Primitive Data Structure.
2. Non-primitive Data Structure.


Data Structures
Primitive Data Structure.
Non - primitive Data Structure.
Eg. Int, char, float, pointer etc.
Linear DS
Non- Linear DS
Eg. Array, Linked list, Stack, queue etc.
Eg. Trees, Graphs etc.

Primitive Data Structure: It is the basic Data structure which can be directly operated by the machine instruction.

Non-Primitive Data Structure: It is the Data structure which emphasizes on structuring of a group of homogenous or heterogeneous data items. It is further classified into two types. They are:

- ✓ Linear Data Structures.
- ✓ Non- Linear Data Structures.

Linear Data Structures: It is a data structure which contains a linear arrangement of elements in the memory.

Non-Linear Data Structure: It is a data structure which represents a hierarchical arrangement of elements.

List ADT:

A list is a dynamic data structure. The no. of nodes on a list may vary dramatically as elements are inserted and removed. The dynamic nature of list is implemented using linked list and the static nature of list is implemented using array.

In general the list is in the form of elements A1, A2, …, AN, where N is the size of the list associated with a set of operations:

- ✓ Insert: add an element e.g. Insert(X,5)-Insert the element X after the position 5.
- ✓ Delete: remove an element e.g. Delete(X)-The element X is deleted.
- ✓ Find: find the position of an element (search) e.g. Find(X)-Returns the position of X
- ✓ FindKth: find the kth element e.g. Find(L,6)- Returns the element present in the given position of list L.
- ✓ PrintList: Display all the elements from the list.
- ✓ MakeEmpty: Make the list as empty list.

Implementation of List:

There are two methods

1. Array
2. Linked list

Array Implementation of list:

Array: A set of data elements of same data type is called array. Array is a static data structure i.e., the memory should be allocated in advance and the size is fixed. This will waste the memory space when used space is less than the allocated space. An array implementation allows the following operations.

The basic operations are:

a. Creation of a List.
b. Insertion of a data in the List
c. Deletion of a data from the List
d. Searching of a data in the list

Global Declaration:

```
int list[25], index=-1;
```

Note: The initial value of index is -1.

Create Operation:

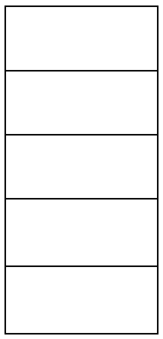
Procedure:

```
void create()

{

int n,i;
printf("\nEnter the no.of elements to be added in the list");

scanf("%d",&n);

if(n<=maxsize)

for(i=0;i<n;i++)

{

scanf("%d",&list[i]);

index++;

}

else

printf("\nThe size is limited. You cannot add data into the list");

}
```

Explanation:

- The list is initially created with a set of elements.
- Get the no. of elements (n) to be added in the list.
- Check n is less than or equal to maximum size. If yes, add the elements to the list.
- Otherwise, give an error message.
-

Diagram:



Insert Operation:

Procedure:

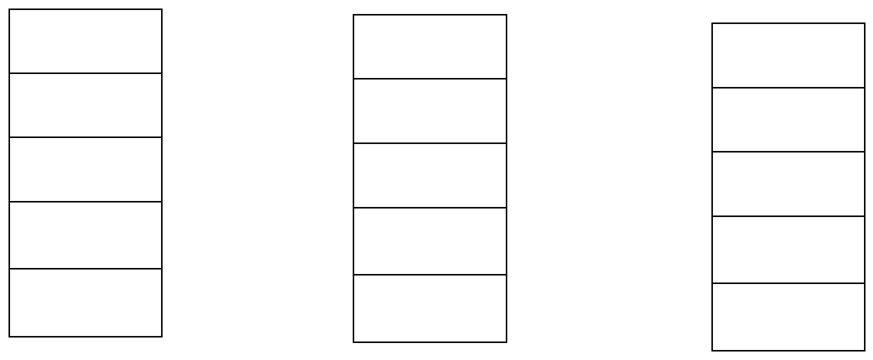
```
void insert()
{
        int i,data,pos;
        printf("\nEnter the data to be inserted");
        scanf("%d",&data);
        printf("\nEnter the position at which element to be inserted");
        scanf("%d",&pos);
        if(index<maxsize)
        {
                for(i=index;i>=pos-1;i--)
                list[i+1]=list[i];
                index++;
                list[pos-1]=data;
        }
        else
                printf("\nThe list is full");
}
```

Explanation:

- Get the data element to be inserted.
- Get the position at which element is to be inserted.
- If index is less than or equal to maxsize, then Make that position empty by altering the position of the elements in the list.
- Insert the element in the poistion.
- Otherwise, it implies that the list is empty.

Diagram:

.



Deletion Operation:

Procedure:

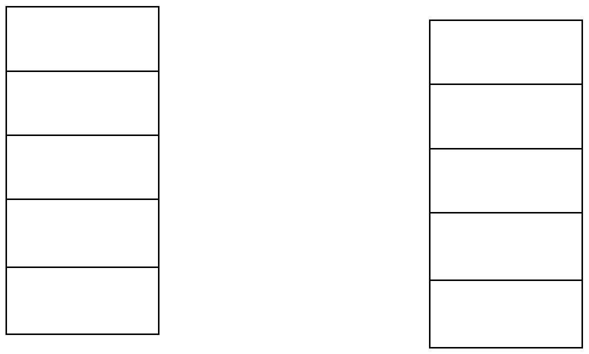
```
void del()

{

        int i,pos;

        printf("\nEnter the position of the data to be deleted");

        scanf("%d",&pos);

        printf("\nThe data deleted is %d",list[pos-1]);

        for(i=pos;i<=index;i++)

        list[i-1]=list[i];

        index--;

}
```

Explanation:

- Get the position of the element to be deleted.
- Alter the position of the elements by performing an assignment operation, list[i-1]=list[i], where i value ranges from position to the last index of the array.

Diagram:

.



Display Operation:

Procedure:

```
void display()

{

        int i;

        for(i=0;i<=index;i++)

        printf("\t%d",list[i]);

}
```

Explanation:

- Formulate a loop, where i value ranges from 0 to index (index denotes the index of the last element in the array.
- Display each element in the array.

Limitation of array implementation

- An array size is fixed at the start of execution and can store only the limited number of elements.
- Insertion and deletion of array are expensive. Since insertion is performed by pushing the entire array one position down and deletion is performed by shifting the entire array one position up.

A better approach is to use a ***Linked List.***

Linked data structure

Linked data structure is a data structure which consists of a set of data records (*nodes*) linked together and organized by references (*links* or *pointers*). The link between data can also be called a **connector**.

In linked data structures, the links are usually treated as special data types that can only be dereferenced or compared for equality. Linked data structures are thus contrasted with arrays and other data structures that require performing arithmetic operations on pointers. This distinction holds even when the nodes are actually implemented as elements of a single array, and the references are actually array indices: as long as no arithmetic is done on those indices, the data structure is essentially a linked one.

Linking can be done in two ways - Using dynamic allocation and using array index linking.

Linked lists

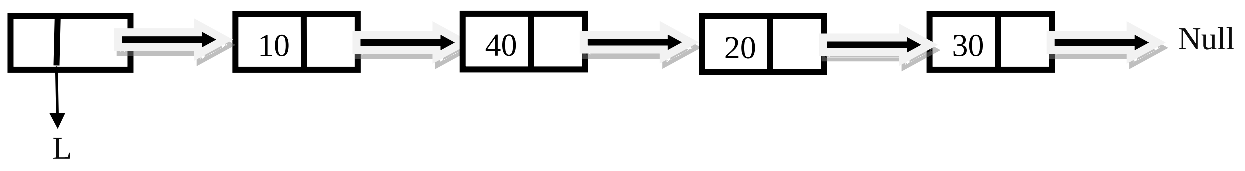
A linked list is a collection of structures ordered not by their physical placement in memory but by logical links that are stored as part of the data in the structure itself. It is not necessary that it should be stored in the adjacent memory locations. Every structure has a data field and an address field. The Address field contains the address of its successor.

A list is a linear structure. Each item except the first (front, head) has a unique predecessor. Each item except the last (end, tail) has a unique successor. First item has no predecessor, and last item has no successor. An item within a list is specified by its position in the list.

Linked list can be singly, doubly or multiply linked and can either be linear or circular.

Singly Linked List

Singly linked list is the most basic linked data structure. In this the elements can be placed anywhere in the heap memory unlike array which uses contiguous locations. Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node contains the NULL reference. It has a dynamic size, which can be determined only at run time.



Basic operations of a singly-linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list.

Algorithm:

. The node of a linked list is a structure with fields data (which stored the value of the node) and*next (which is a pointer of type node that stores the address of the next node).

Two nodes *start (which always points to the first node of the linked list) and *temp (which is used to point to the last node of the linked list) are initialized.

Initially temp = start and temp->next = NULL. Here, we take the first node as a dummy node. The first node does not contain data, but it used because to avoid handling special cases in insert and delete functions.

Functions –

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address in the next field of the new node as NULL.
2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted, If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list.Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be removed.

   Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.
3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key.Until next field of the pointer is equal to NULL, check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0,else return 1.
4. Print - function takes the start node (as pointer) as an argument. If pointer = NULL, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

Performance:

1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited number of nodes in a fragmented memory environment.
2. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.

Declaration of Linked List

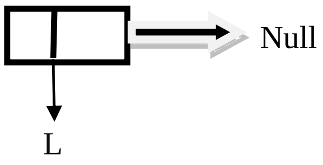
```
void insert(int X,List L,position P);
void find(List L,int X);
void delete(int x , List L);

typedef struct node *position;
position L,p,newnode;
```

```
struct node
{
        int data;
        position next;
};
```

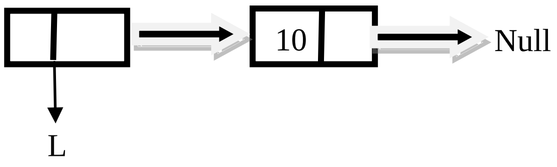
Routine to insert an element in list

```
Void insert(int X,List L,position p)
{
        position newnode;
        newnode=malloc(sizeof(struct node));
        If(newnode==NULL)
                Factal error("Out of Space");
        else
          {
                newnode->data=x;
                newnode-next=p-next;
                p->next=newnode;
          }
}
```

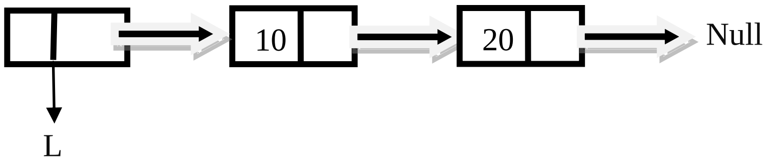
Routine to Insert an Element in the List



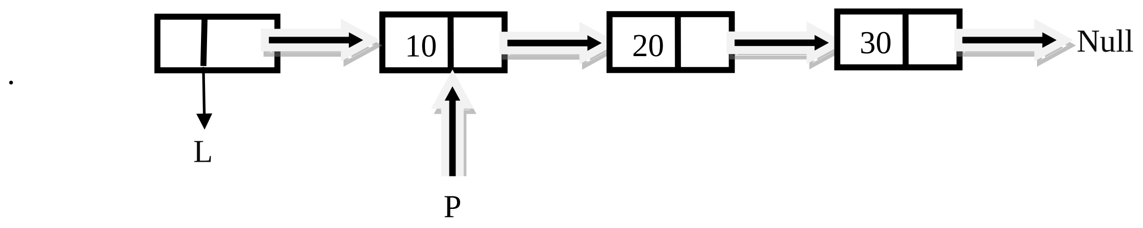
Insert(Start,10) - A new node with data 10 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



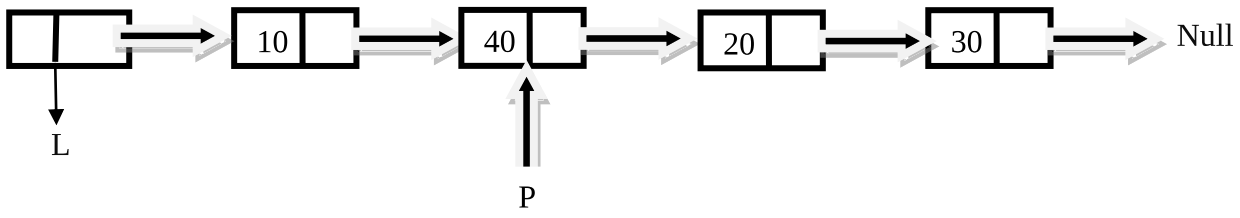
Insert(Start,20) - A new node with data 20 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



Insert(Start,30) - A new node with data 30 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.

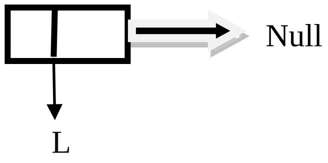


Insert(Start,40) - A new node with data 40 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



Routine to check whether a list is Empty

```
int IsEmpty(List L)
{
   if (L->next==NULL)
             return(1);
}
```



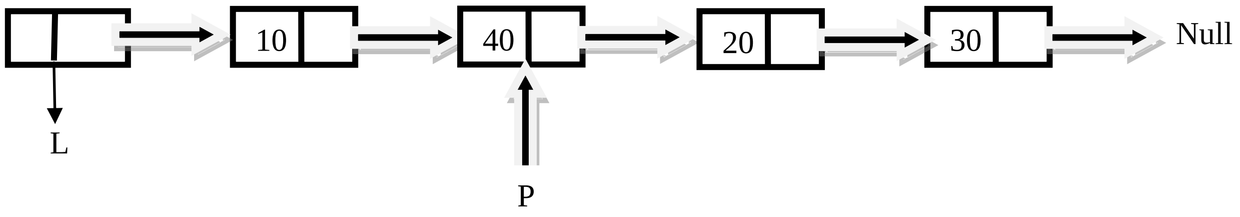
if the answer is 1 result is true

if the answer is 0 result is false

Routine to check whether the current position is last in the List

```
int IsLast(List L , position p)

{

        if(p->next==NULL)

                return(1);

}
```

.



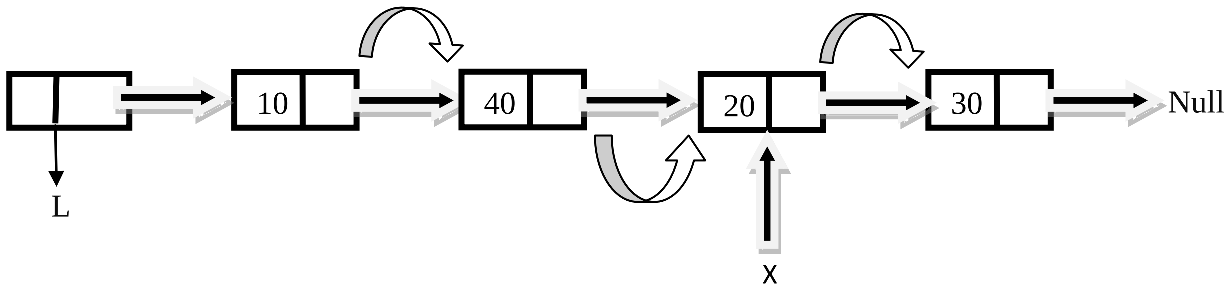
if the answer is 1 result is true

if the answer is 0 result is false

Routine to Find the Element in the List:

```
position find(List  L,int  X)
{
      position p;
      p=L->next;
      while(p!=NULL && p->data!=X)
               p=p->next;
      return(p);
}
```

Find(start,10) - To find an element start from the first node of the list and move to the next with the help of the address stored in the next field.



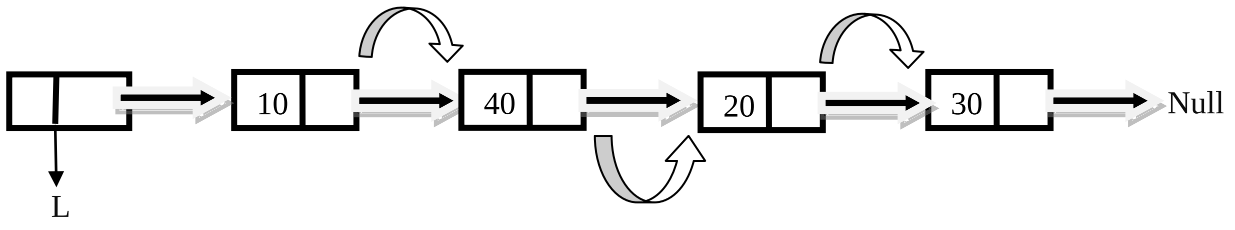
Find Previous

It returns the position of its predecessor.

```
position find previous (int X,list L)
.
{
        position p;
        p=L;
        while(p->next!=NULL && p->next->data!=X)
                p=p->next;
        return P;
}
```

Routine to Count the Element in the List:

```
void count(list L)
{
        p=L->next;
        while(p!=NULL)
        {
                count++;
                p=p->next;
        }
 print count;
}
```

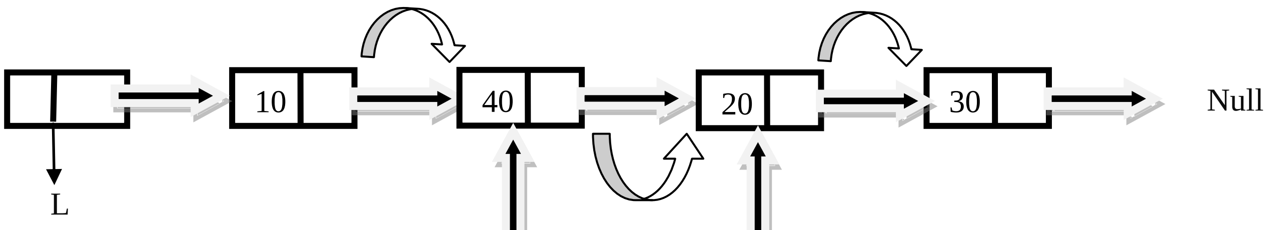


Routine to Delete an Element in the List:

Delete(start,20) - A node with data 20 is found and the next field is updated to store the NULL value. The next field of previous node is updated to store the address of node next to the deleted node.

it delete the first occurrence of element X from the list L
.

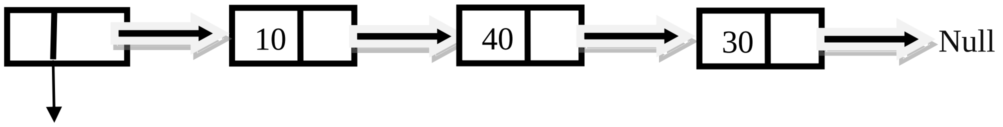
```
void delete(int x , List L)
{
        position p,Temp;
        p=find previous(X,L);
        if(!IsLast(p,L))
        {
                temp=p->next;
                p->next=temp->next;
                free(temp);
        }
}
```



P

Temp

After Deletion



Start and Temp

Routine to Delete the List

Deletet(start->next) - To print start from the first node of the list and move to the next with the help of the address stored in the next field.

```
void delete list(List L)

{

        position P,temp;

        P=L->next;

        L->next=NULL;

        while(P!=NULL)

        {

                temp=P->next;

                free(P);

                P=temp;

        }

}
```

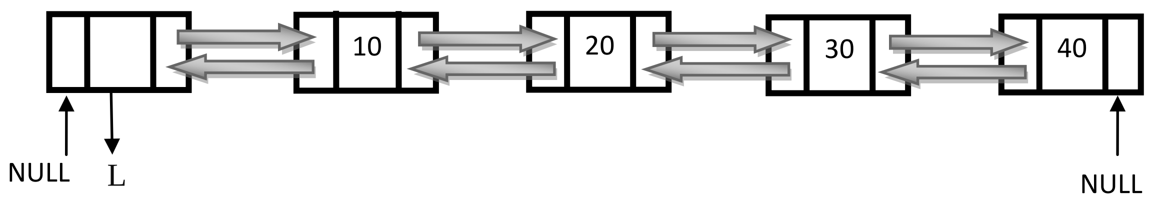
Routine to find next Element in the List

```
void findnext(int X, List L)

{

        position P;

        P=L->next;

        while(P!=NULL && P->data!=X)

                P->next;

        return P->next;

}
```

it returns its position of successor.

Doubly-Linked List

. Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.



Basic operations of a singly-linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list.

Algorithm:

The node of a linked list is a structure with fields data (which stored the value of the node), *previous (which is a pointer of type node that stores the address of the previous node) and *next (Which is a pointer of type node that stores the address of the next node)?

Two nodes *start (which always points to the first node of the linked list) and *temp (which is used to point to the last node of the linked list) are initialized.

Initially temp = start, temp->prevoius = NULL and temp->next = NULL. Here, we take the first node as a dummy node. The first node does not contain data, but it used because to avoid handling special cases in insert and delete functions.

Functions –

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address of the previous node in the previous field of the new node and address in the next field of the new node as NULL.

2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted, If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list.

   Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be removed.

   Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next) and also link the pointer and the node next to the node to be deleted (temp->prev = pointer). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

.

3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key.
   Until next field of the pointer is equal to NULL, check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.

4. Print - function takes the start node (as pointer) as an argument. If pointer = NULL, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

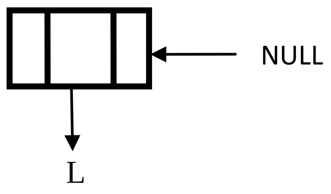
Performance:

1. The advantage of a singly linked list is that we don't need to keep track of the previous node for traversal or no need of traversing the whole list for finding the previous node.
2. The disadvantage is that more pointers needs to be handled and more link need to updated.

General declaration routine:

```
typedef struct node *position;
struct node
{
        int data;
        position prev;
        position next;
};
```

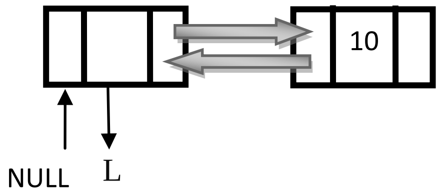
Initially



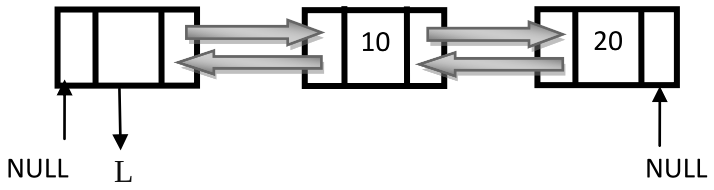
Routine for insert an element:

```
void insert (int x,List L,position p)
{
    position newnode;
    newnode = malloc(sizeof(struct node));
    if(newnode==NULL)
      Fatal error ("out of space");
      else
       {
           newnode ->data = x;
           newnode ->next =p->next;
           p->next ->prev = newnode;
           p->next= newnode;
           newnode ->prev =p;
}
```

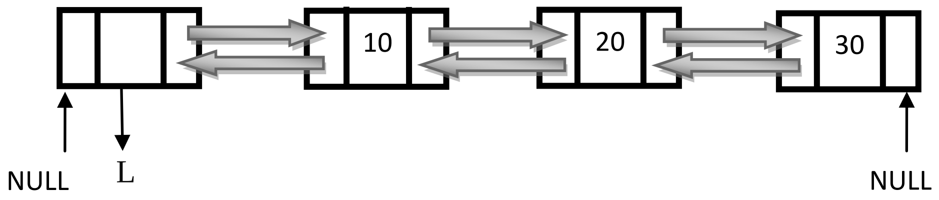
Insert(Start,10) - A new node with data 10 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.



Insert(Start,20) - A new node with data 20 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.

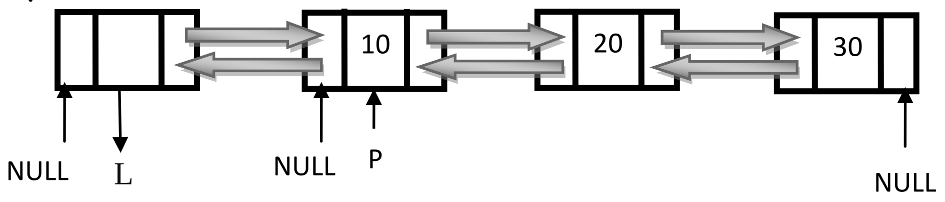


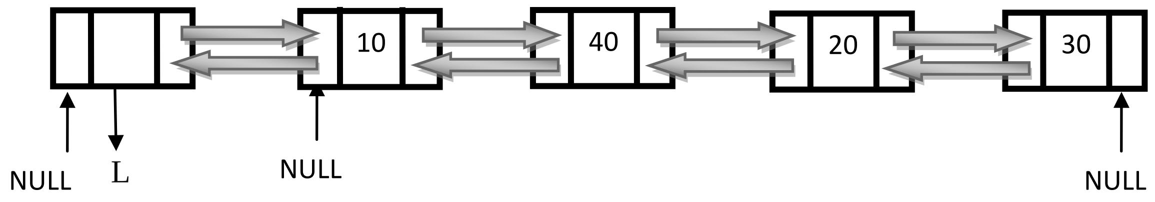
Insert(Start,30) - A new node with data 30 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.



Insert(Start,40) - A new node with data 40 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.

.



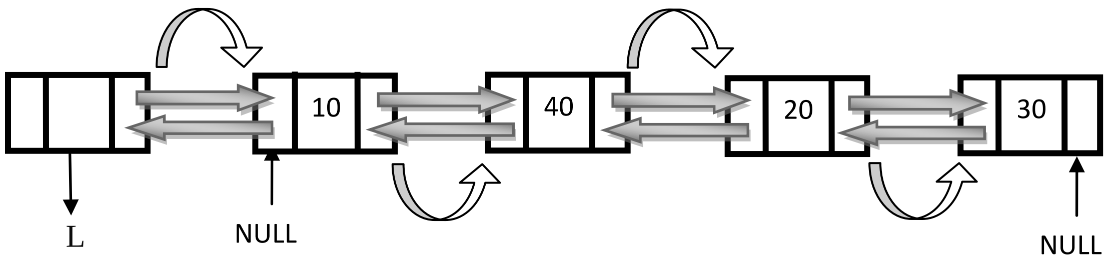


Routine to displaying an element:

```
void display(List L)
{
  p=L->next;
while (p!=NULL)
{
     print p->data;
     p=p->next;
}
print NULL
}
```

Print(start)   10, 40, 20, 30

To print start from the first node of the list and move to the next with the help of the address stored in the next field.

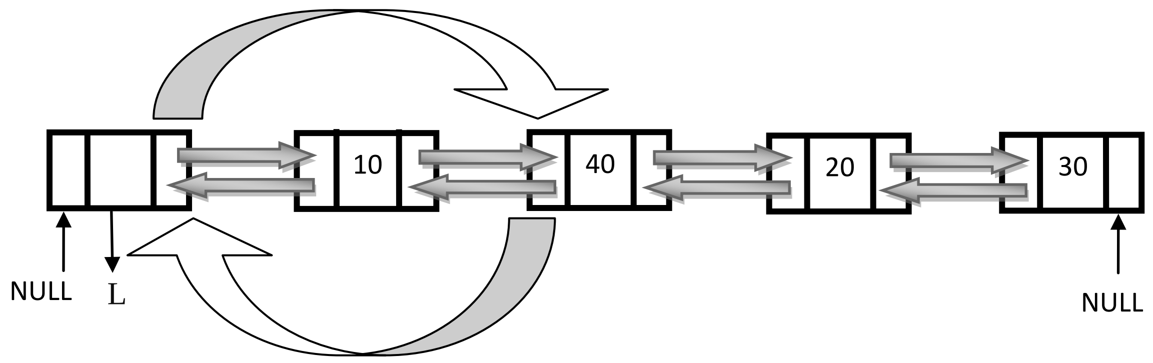


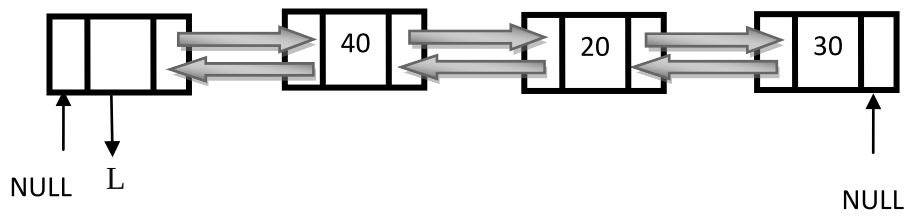
Routine for deleting an element:

.

```
Void delete (int x ,List L)
{
  Position p , temp;
  p=find(x,L);
  If(isLast(p,L))
   {
      temp =p;
       p->prev->next=NULL;
       free(temp);
   }
  else
  {
   temp = p;
   p->prev->next=p-next;
   p-next-prev = p->prev;
  free(temp);
}
```

Delete(start,10) - A node with data 10 is found, the next and previous fields are updated to store the NULL value. The next field of previous node is updated to store the address of node next to the deleted node. The previous field of the node next to the deleted one is updated to store the address of the node that is before the deleted node.

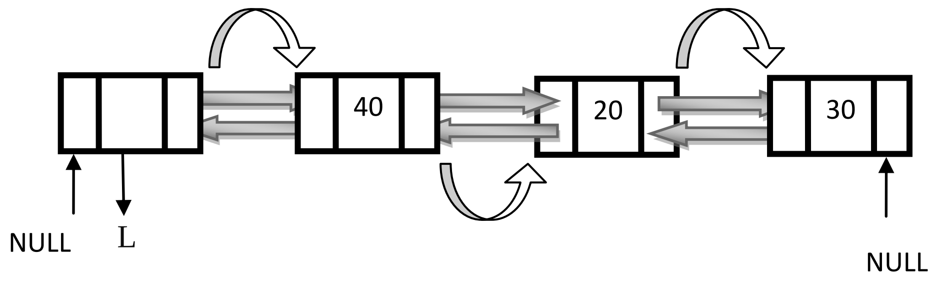




Find(start,10) ‘Element Not Found’

To print start from the first node of the list and move to the next with the help of the address stored in the next field.

.



Circular Linked List

Circular linked list is a more complicated linked data structure. In this the elements can be placed anywhere in the heap memory unlike array which uses contiguous locations. Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node points back to the first node unlike singly linked list. It has a dynamic size, which can be determined only at run time.

Basic operations of a singly-linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list.

Algorithm:

The node of a linked list is a structure with fields data (which stored the value of the node) and*next (which is a pointer of type node that stores the address of the next node).

Two nodes *start (which always points to the first node of the linked list) and *temp (which is used to point to the last node of the linked list) are initialized.

Initially temp = start and temp->next = start. Here, we take the first node as a dummy

node. The first node does not contain data, but it used because to avoid handling special cases in insert and delete functions.

Functions –

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address of the first node (start) in the next field of the new node.
2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted, If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list. Else, now pointer points to a node and the node next to it has to be removed, declare a temporary

node (temp) which points to the node which has to be removed. Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

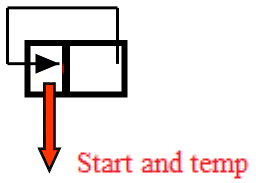
3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. A pointer start of type node is declared, which points to the head node of the list (node *start = pointer). First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key.
   Until next field of the pointer is equal to start, check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.
4. Print - function takes the start node (as start) and the next node (as pointer) as arguments. If pointer = start, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

Performance:

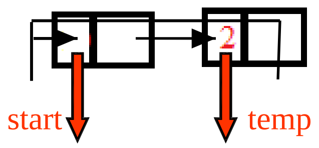
1. The advantage is that we no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found in constant time. Also, for implementing queues we will only need one pointer namely tail, to locate both head and tail.
2. The disadvantage is that the algorithms have become more complicated.

Example:

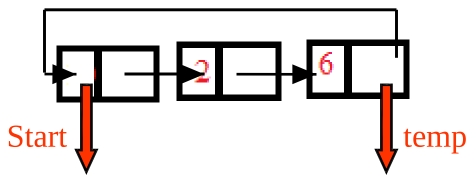
Initially



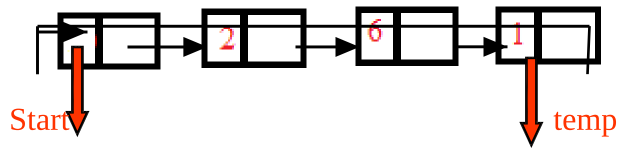
Insert(Start,2) - A new node with data 2 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.



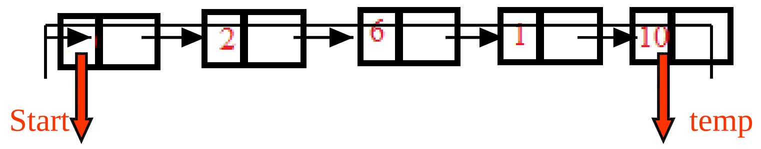
Insert(Start,6) - A new node with data 6 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.



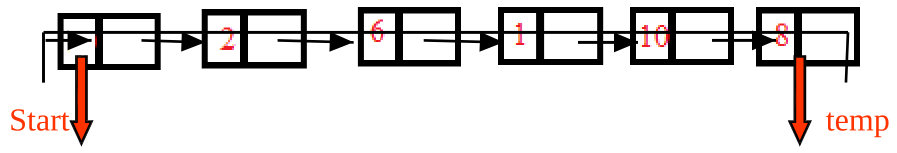
Insert(Start,1) – A new node with data 1 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.



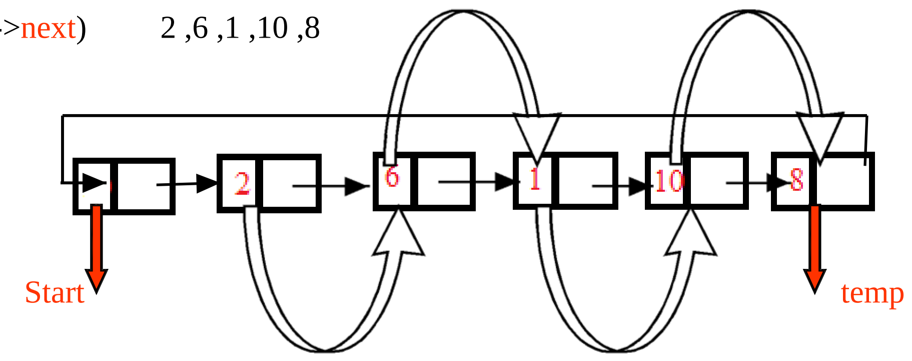
Insert(Start,10) – A new node with data 6 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.



Insert(Start,8) - A new node with data 6 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.

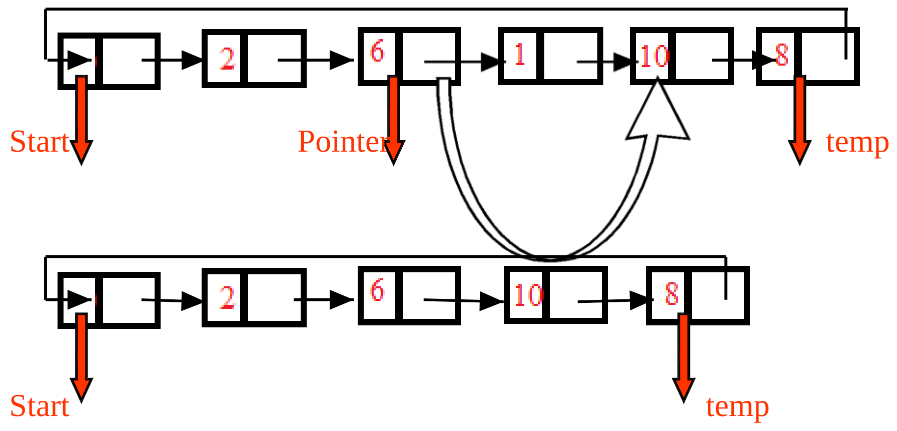


Print(start->next) 2 ,6 ,1 ,10 ,8

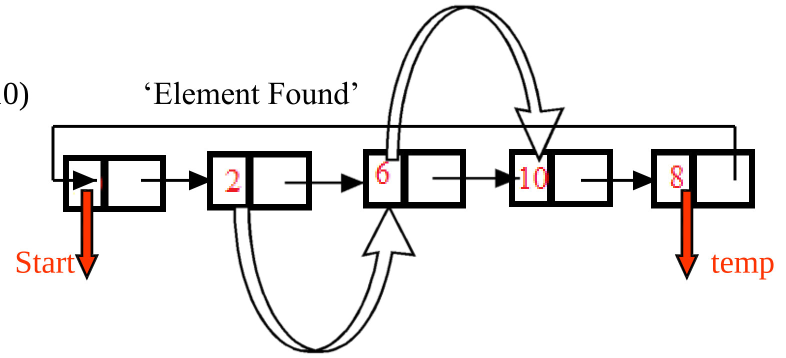


To print start from the first node of the list and move to the next with the help of the address stored in the next field.

Delete(start,1) - A node with data 1 is found and the next field is updated to store the NULL value. The next field of previous node is updated to store the address of node next to the deleted node.



Find(start,10) 'Element Found'



To find, start from the first node of the list and move to the next with the help of the address stored in the next field.

. Applications of List

Polynomial Manipulation

Polynomial manipulations such as addition, subtraction & differentiation etc.. can be performed using linked list

Declaration for Linked list implementation of Polynomial ADT

```
struct poly
{
int coeff;
int power;
struct poly * Next;
}*list1,*list2,*list3;
```

Creation of the Polynomial

```
    poly create(poly*head1,poly*newnode1)
    {
        poly *ptr;
        if(head1==NULL)
        {
                head1=newnode1;
                return (head1);
        }
        else
        {
                ptr=head1;
                while(ptr->next!=NULL)
                ptr=ptr->next;
                ptr->next=newnode1;
        }
        return(head1);
}
```

Addition of two polynomials

```
void add()
{
        poly*ptr1,*ptr2,*newnode;
        ptr1=list1;
        ptr2=list2;
        while(ptr1!=NULL&&ptr2!=NULL)
        {
                newnode=malloc(sizeof(struct poly));
if(ptr1->power==ptr2->power)
                    {
  .                           newnode->coeff=ptr1-coeff+ptr2->coeff;
                              newnode->power=ptr1->power;
                              newnode->next=NULL;
                              list3=create(list3,newnode);
                              ptr1=ptr1->next;
                              ptr2=ptr2->next;
                    }
                    else
                    {
                              if(ptr1-power>ptr2-power)
                              {
                                        newnode->coeff=ptr1->coeff;
                                        newnode->power=ptr1->power;
                                        newnode->next=NULL;
                                        list3=create(list3,newnode);
                                        ptr1=ptr1->next;
                              }
                    else
                    {
                              newnode->coeff=ptr2->coeff;
                              newnode->power=ptr2->power;
                              newnode->next=NULL;
                              list3=create(list3,newnode);
                              ptr2=ptr2->next;
                    }
          }
}
```

Subtraction of two polynomial

```
void sub()
 {
          poly*ptr1,*ptr2,*newnode;
          ptr1=list1;
          ptr2=list2;
          while(ptr1!=NULL&&ptr2!=NULL)
          {

                  newnode=malloc(sizeof(struct poly));

                  if(ptr1->power==ptr2->power)
                 {
                          newnode->coeff=(ptr1-coeff)-(ptr2->coeff);
newnode->power=ptr1->power;
                              newnode->next=NULL;
 .                            list3=create(list3,newnode);
                              ptr1=ptr1->next;
                              ptr2=ptr2->next;
                    }
                    else
                    {
                              if(ptr1-power>ptr2-power)
                              {
                                        newnode->coeff=ptr1->coeff;
                                        newnode->power=ptr1->power;
                                        newnode->next=NULL;
                                        list3=create(list3,newnode);
                                        ptr1=ptr1->next;
                              }
                    else
                    {
                              newnode->coeff=-(ptr2->coeff);
                              newnode->power=ptr2->power;
                              newnode->next=NULL;
                              list3=create(list3,newnode);
                              ptr2=ptr2->next;
                    }
          }

}
```