LECTURE NOTES

PROGRAMME – BCA

SEMESTER- II

DATA STRUCTURE USING C (BCA-204)

UNIT I

Data Structure

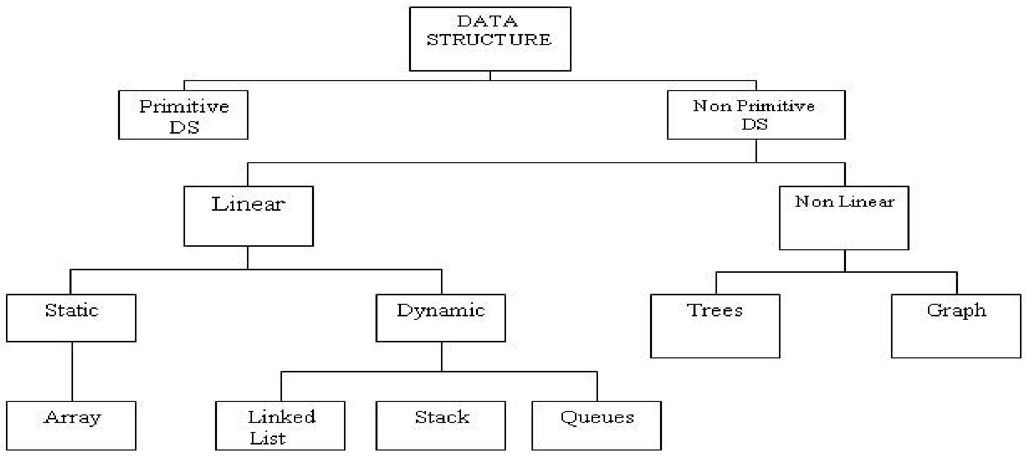
- Data Structure is representation of Data & operations allowed on the data.
- Data is represented by data values held temporarily within program data area or recorded permanently on a file often the different values are related to each other

Data Structure = Organised Data + Allowed Operations

Or

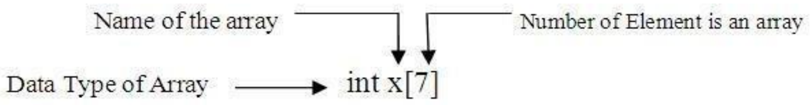
- Way of organizing and storing data in a computer system
- This organization is done under a common name.
- Depicts the logical representation of data in computer memory.

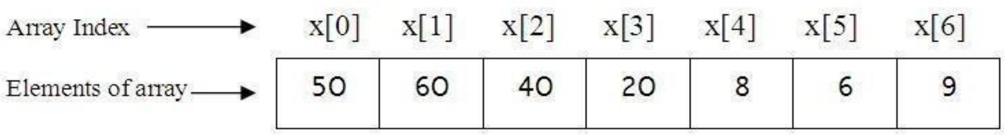
Types of Data Structure



Arrays

- Collection of similar type data elements stored at consecutive locations in the memory**.**





- Advantages

- Individual data elements can be easily referred.

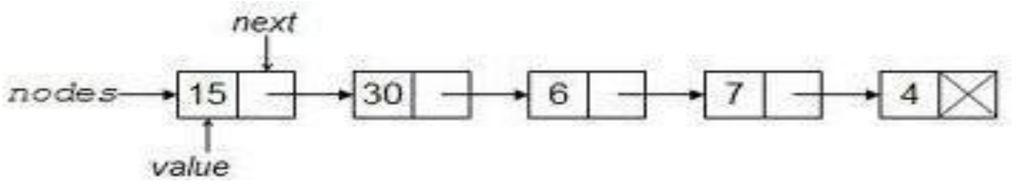
 Limitations

- Wastage of memory space.

Linked Lists

 To store data in the form of a list of nodes connected to each other through pointers.

- Each node has two parts – data and pointer to next data



 Advantages

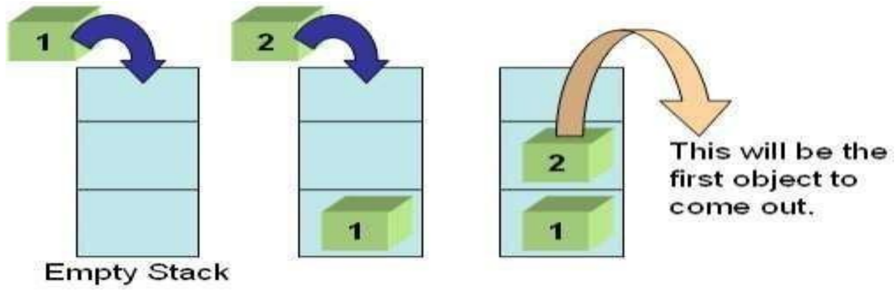
- Optimize the use of storage space.

 Limitations

- Individual elements cannot be referred directly

Stacks

 Maintains a list of elements in a manner that elements can be inserted or deleted only from one end (referred as top of the stack) of the list.



 LIFO-Last In First Out principle.

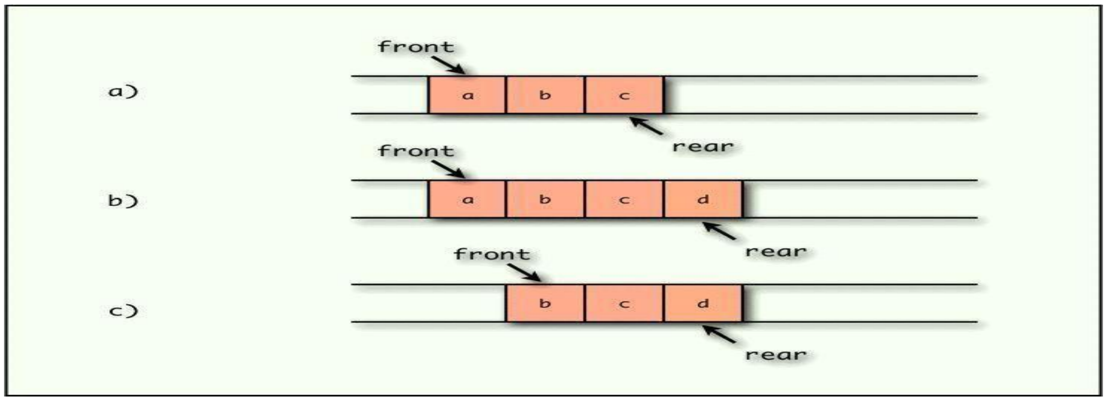
 Applications

- Implementations of system processes like program control, recursion control

Queues

 Maintains a list of elements such that insertion happens at rear end and deletion happens at front end

- FIFO – First In First Out principle



Trees

Represent data containing hierarchical relationship between elements. Example: family trees, records and table of contents.

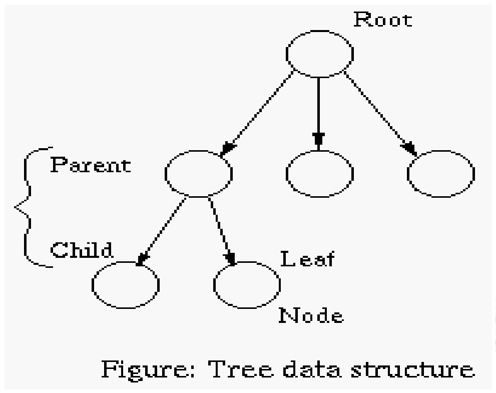
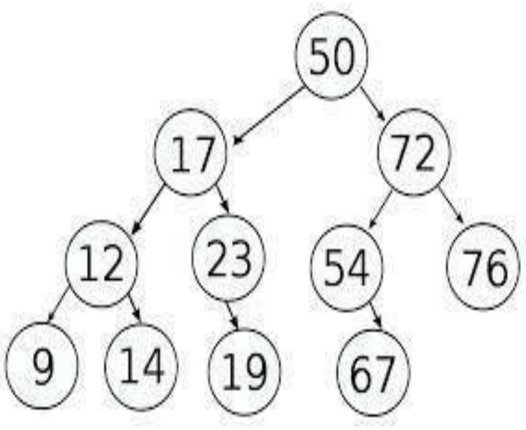


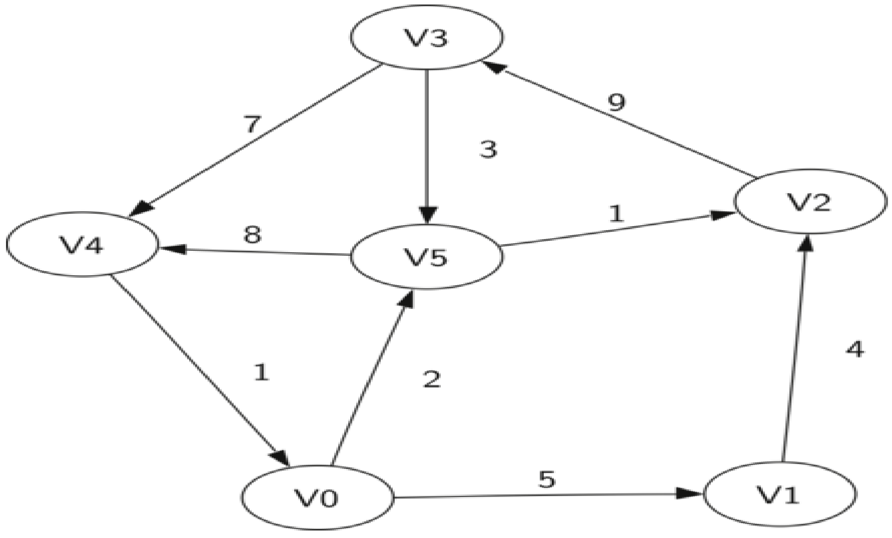
Figure: Tree data structure



- Applications
  - Implementing search algorithms

Graphs

- It is a linked data structure that comprises of vertices and a group of edges.
- Edges are associated with certain values called weights.
- Helps to compute the cost of traversing the graph through a certain path.



Abstract Data Types (ADTs) in C

- C is not object-oriented, but we can still manage to inject some object-oriented principles into the design of C code.
- For example, a data structure and its operations can be packaged together into an entity called an ADT.
- There’s a clean, simple interface between the ADT and the program(s) that use it.
- The lower-level implementation details of the data structure are hidden from view of the rest of the program.

- An abstract data type (ADT) is a set of operations and mathematical abstractions , which can be viewed as how the set of operations is implemented. Objects like lists, sets and graphs, along with their operation, can be viewed as abstract data types, just as integers, real numbers and Booleans.

Features of ADT.

- Modularity
  - Divide program into small functions
  - Easy to debug and maintain
  - Easy to modify
- Reuse
  - Define some operations only once and reuse them in future
- Easy to change the implementation

<u>List ADT</u>

LIST:

A list is a sequential data structure, ie. a collection of items accessible one after another beginning at the head and ending at the tail.

- It is a widely used data structure for applications which do not need random access
- Addition and removals can be made at any position in the list
- lists are normally in the form of a_1,a_2,a_3.......a_n. The size of this list is n.The first element of the list is a_1,and the last element is a_n.The position of element a_i in a list is i.
- List of size 0 is called as null list.

A list is a sequence of zero or more elements of a given type. The list is represented as sequence of elements separated by comma.

A1,A2,A3…..AN

where N>0 and A is of type element.

Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

1. Storing a list in a static data structure(Array List)

- This implementation stores the list in an array.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- The element with the index can be accessed in constant time (ie) the time to access does not depend on the size of the list.
- The time taken to add an element at the end of the list does not depend on the size of the list. But the time taken to add an element at any other point in the list depends on the size

of the list because the subsequent elements must be shifted to next index value.So the additions near the start of the list take longer time than the additions near the middle or end.

- Similarly when an element is removed,subsequent elements must be shifted to the previous index value. So removals near the start of the list take longer time than removals near the middle or end of the list.

Problems with Array implementation of lists:

- Insertion and deletion are expensive. For example, inserting at position 0 (a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is $O(n)$.
- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, if there are many lists of unknown size.
- Simple arrays are generally not used to implement lists. Because the running time for insertion and deletion is so slow and the list size must be known in advance

2. Storing a list in a dynamic data structure(Linked List)

- The Linked List is stored as a sequence of linked nodes which are not necessarily adjacent in memory.
- Each node in a linked list contains data and a reference to the next node
- The list can grow and shrink in size during execution of a program.
- The list can be made just as long as required. It does not waste memory space because successive elements are connected by pointers.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- The time taken to access an element with an index depends on the index because each element of the list must be traversed until the required index is found.
- The time taken to add an element at any point in the list does not depend on the size of the list,as no shifts are required
- Additions and deletion near the end of the list take longer than additions near the middle or start of the list. because the list must be traversed until the required index is found

Array versus Linked Lists

1.Arrays are suitable for

- Randomly accessing any element.
- Searching the list for a particular value
- Inserting or deleting an element at the end.

2. Linked lists are suitable for

-Inserting/Deleting an element.
-Applications where sequential access is required.
-In situations where the number of elements cannot be predicted beforehand.

Array Based Implementation

Array is a collection of specific number of data stored in consecutive memory locations.

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Fig 3.3.1 Array model

Operations on Array

- Insertion
- Deletion
- Merge
- Traversal
- Find

Insertion Operation on Array

- It is the process of adding an element into the existing array. It can be done at any position.
- Insertion at the end is easy as it is done by shifting one position towards right of last element if it does not exceeds the array size
- Example

i) Insertion at the end:

20	10	30	40		
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Insert (70,A)

20	10	30	40	70	
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

ii) Insertion at specified position

20	10	30			
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Insert(40,1,A)

20	10	30			
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

First shift the last element one position right (from location 2 to 3)

20	10		30		
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Shift the element (10) one position right (from location 1 to 2)

20		10	30		
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Now insert element (40) at location 1

20	40	10	30		
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

- ***Routine to insert an element in an array***

```
void insert(int X, int P, int A[], int N)
{
        if(P==N)
                printf("Array overflow");
        else
        {
                for(int  i=N-1;i>=P;i--)
                        A[i+1]=A[i];
                A[P]=X;
                N=N+1;
        }
}
```

Deletion operation on an Array

- It is the process of removing an element from the array at any position
- ***Routine***

```
int deletion(int P,int A[],int N)
{
        if(P==N-1)
                temp=A[P];
        else
        {
                temp=A[P];
                for(i=P;i<N-1;i++)
                        A[i]=A[i+1];
        }
        N=N-1;
        return temp;
}
```

Merge Operation

- It is the process of combining two sorted array into single sorted array.

2	4	6		
A[0]	A[1]	A[2]	A[3]	A[4]

1	3	8		
B[0]	B[1]	B[2]	B[3]	B[4]

1	2	3	4	6	8				
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]

- ***Routine to merge two sorted array***

```
void merge(int a[],int n, int b[],int m)
{
        int c[n+m];
        int i=j=k=0;
        while(i<n&&j<m)
        {
                if( a[i]<b[j])
                {
                        c[k] =a[i];
                        i++;
                        k++;
                }
                else
                {
                        c[k]=b[j];
                        j++;
                        k++;
                }
        }
        while(i<n)
        {
                c[k] =a[i];
                i++;
                k++;
        }
        while(j<m)
        {
                c[k] =a[i];
                j++;
                k++;
        }
}
```

Find operation

- It is the process of searching an element in the given array. If the element is found, it returns the position of the search element otherwise NULL.
- ***Routine***

```
int find(int x, int a[], int N)
{
        int pos,flag=0;
        for(int i=0;i<N;i++)
        {
                if(x==a[i])
                {
                        flag=1;
                        pos=i;
                        break;
                }
        }
        if(flag==1)
                printf("element %d is found at position %d",x,pos);
        else
                printf("Element not found");
        return pos;
}
```

Traversal operation

- It is the process of visiting the elements in an array.
- ***Routine***

```
void traversal(int a[],int n)
{
        for(int i=0;i<n;i++)
                printf(a[i]);
}
```

Merits and demerits of array implementation of lists

Merits

- Fast, random access of elements
- Memory efficient – very less amount of memory is required

Demerits

- Insertion and deletion operations are very slow since the elements should be moved.
- Redundant memory space – difficult to estimate the size of array.

Linked List Implementation

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to the NULL.

Data element	Next Pointer

Node

Types of linked list

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly Linked Lists

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

Doubly Linked Lists

A doubly linked list is a linked list in which each node has three fields namely data field, forward link(FLINK) and Backward Link(BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

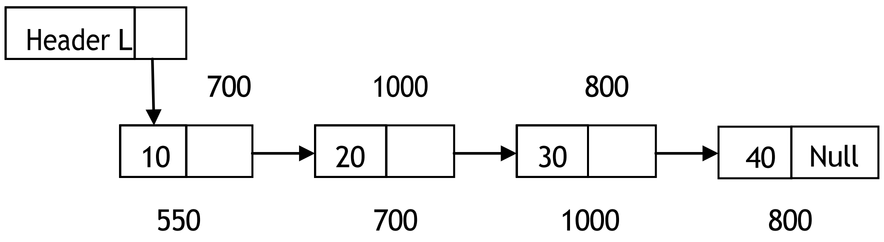
Circular Linked List

In circular linked list the pointer of the last node points to the first node. It can be implemented as

- Singly linked circular list
- Doubly linked circular list

Singly Linked Lists

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.



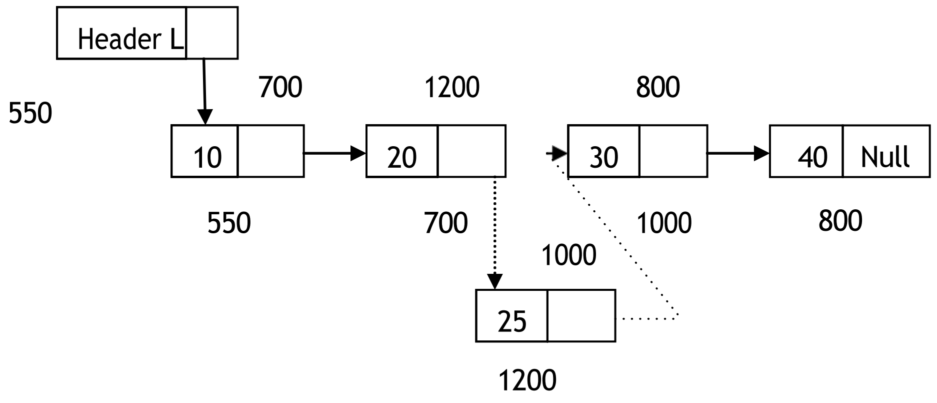
Declaration for Linked List

```
        struct node;
        typedef struct node *List;
        typedef struct node *Position;
        int isLast(List L);
        int isEmpty(List L);
        Position Find(int X,List L);
        void Delete(int X, List L);
        Position FindPrevious(int X,List L);
        Position FindNext(int X,List L);
        void insert(int X,List L, Position P);
        void DeleteList(List L);
        struct node
        {
                int Element;
                Position Next;
        };
```

Routine to insert an element in the List

```
void Insert(int X,List L, Position P)
{
        Position Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode!=NULL)
        {
                Newnode→Element=X;
                Newnode→Next=P→Next;
                P→Next=Newnode;
        }
}
```

Example : *Insert(25,L,P)*



Routine to check whether the list is empty

```
int isEmpty(List L)     /* returns 1 if L is empty */
{
        if(L→Next==NULL)
                return 1;
}
```

Routine to Check whether the current position is Last

```
int isLast(Position P,List L)
{
        if(P→Next==NULL)
                return 1;
}
```

Find Routine

```
Position Find(int X,List L)
{
        Position P;
        P=L→Next;
        While(P!=NULL && P→Element!=X)
                P= P→Next;
        return P;
}
```

FindPrevious Routine

```
Position FindPrevious(int X,List L)
{
        Position P;
        P=L;
        While(P→Next!=NULL && P→Next
                →Element!=X) P= P→Next;
        return P;
}
```

FindNext Routine

```
Position FindNext(int X,List L)
{
        Position P;
        P=L→Next;
        While(P→Next!=NULL && P→Element!=X)
                P= P→Next;
        return P→Next;
}
```

Routine to delete an element from the list

```
void Deletion(int X,List L)
{
        Position P,Temp;
        P=FindPrevious(X,L);
        if(isLast(P,L)
        {
                Temp= P→Next;
                P→Next= Temp→Next;
                Free(Temp);
        }
}
```

Routine to delete the list

```
void DeleteList(List L)
{
        Position P,Temp;
        P=L→Next;
        L→Next=NULL;
        while(P!=NULL)
        {
                Temp=P;
                P=P→Next;
                Free(Temp);
        }
}
```

Doubly Linked Lists

A doubly linked list is a linked list in which each node has three fields namely data field, forward link(FLINK) and Backward Link(BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

BLINK	Data Element	FLINK

Node in Doubly Linked List

Structure Declaration

```
struct node
{
        int Element;
        struct node *FLINK;
        struct node *BLINK;
};
```

Routine to insert an element in a doubly linked list

```
void Insert(int X, List L, Position P)
{
        struct node *Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode!=NULL)
        {
                Newnode→Element = X;
                Newnode→Flink = P→Flink;
                P→Flink→Blink = Newnode;
                P→Flink = Newnode;
                Newnode →Blink=P;
        }
}
```

Routine to delete an element in a doubly linked list

```
void Deletion(int X,List L)
{
        Position P;
        if(Find(X,L)
        {

                Temp=P;
                P→Flink→Blink = NULL;
                Free(Temp);
        }
        else
        {
                Temp=P;
                P →Blink→Flink = P→Flink ;
                P→Flink→Blink = P →Blink;
                Free(Temp);
        }
}
```

Advantages

- Deletion operation is easier
- Finding the predecessor and successor of a node is easier.

Disadvantages

- More memory space is required since it has two pointers.

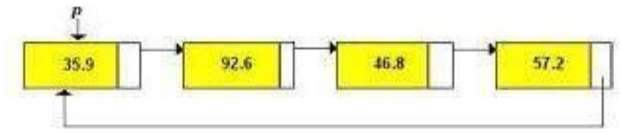
Singly Linked Circular Lists

Circular Linked List

In circular linked list the pointer of the last node points to the first node. It can be implemented as

- Singly linked circular list
- Doubly linked circular list

Singly linked circular list



Structure Declaration

```
struct node
{
        int  Element;
        struct node *Next;
};
```

Routine to insert an element in the beginning

```
void Insert_beg(int X, List L)
{
        struct node *Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode !=NULL)
        {
                Newnode→Element=x;
                Newnode→Next=L→Next;
                L→Next= Newnode;
        }
}
```

Routine to insert an element in the middle

```
void Insert_mid(int X, List L,Position P)
{
        struct node *Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode !=NULL)
        {
                Newnode→Element=x;
                Newnode→Next=P→Next;
                P→Next= Newnode;
        }
}
```

Routine to insert an element in the last

```
void Insert_last(int X, List L)
{
        struct node *Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode !=NULL)
        {
                P=L;
                while(P→Next!=L)
                        P=P→Next;
                Newnode→Element=x;
                P→Next= Newnode;
                Newnode→Next=L;
        }
}
```

Routine to delete an element from the beginning

```
void dele_First(List L)
{
        Position Temp;
        Temp= L→Next;
        L→Next= Temp→Next;
        free(Temp);
}
```

Routine to delete an element from the middle

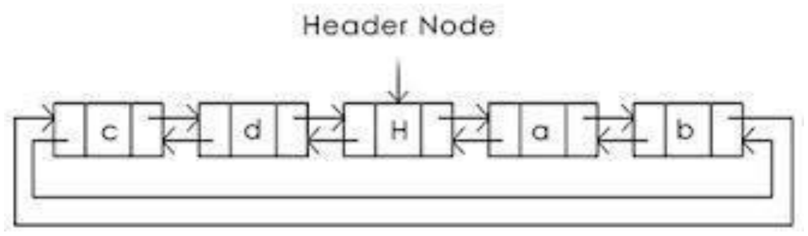
```
void dele_mid(int X,List L)
{
        Position P,Temp;
        P=FindPrevious(X,L);
        if(!isLast(P,L))
        {
                Temp= P→Next;
                P→Next= Temp→Next;
                free(Temp);
        }
}
```

Routine to delete an element in the last

```
void dele_last(int X, List L)
{
        Position P,Temp;
        P=L;
        while(P→Next→Next!=L)
                P=P→Next;
        Temp=P→Next;
        P→Next= L;
        Free(Temp);
}
```

Doubly Linked Circular Lists

A doubly linked circular list is a doubly linked list in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.



Structure Declaration

```
struct node
{
        int Element;
        struct node *FLINK;
        struct node *BLINK;
};
```

Routine to insert an element at the beginning

```
void Insert_beg(int X, List L)
{
        Position Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode!=NULL)
        {
                Newnode→Element = X;
                Newnode→Flink = L→Flink;
                L→Flink→Blink = Newnode;
                L→Flink = Newnode;
                Newnode →Blink=L;
        }
}
```

Routine to insert an element in the middle

```
void Insert_mid(int X, List L,Position P)
{

        Position Newnode;
        Newnode=malloc(sizeof(struct node));
        if(Newnode!=NULL)
        {
                Newnode→Element = X;
                Newnode→Flink = P→Flink;
                P→Flink→Blink = Newnode;
                P→Flink = Newnode;
                Newnode →Blink=P;
        }
}
```

Routine to insert an element at the last

```
void Insert_last(int X, List L)
{
        Position Newnode,P;
        Newnode=malloc(sizeof(struct node));
        if(Newnode!=NULL)
        {
                P=L;
                While(P→Flink!=NULL)
                        P= P→Flink;
                Newnode→Element = X;
                P→Flink = Newnode;
                Newnode→Flink = L;
                Newnode →Blink=P;
                L→Blink = Newnode;
        }
}
```

Routine to delete an element from the beginning

```
void dele_first(List L)
```

```
{
        Position Temp;
        if(L→Flink! = NULL)
        {
                Temp= L→Flink ;
                L→Flink = Temp→Flink;
                Temp→Flink→Blink = L;
                free(Temp);
        }
}
```

Routine to delete an element from the middle

```
void dele_mid(int X,List L)
{
        Position P,Temp;
        P=FindPrevious(X);
        if(!isLast(P,L)
        {
                Temp= P→Flink ;
                P→Flink = Temp→Flink;
                Temp→Flink→Blink = P;
                free(Temp);
        }
}
```

Routine to delete an element from the last

```
void dele_last(List L)
{
        Position Temp;
        P=L;
        While(P→Flink!= L)
                        P= P→Flink;
        Temp= P;
        P→Blink →Flink = L;
        L→Blink  = P→Blink;
        free(Temp);
}
```

Polynomial Manipulation – Insertion, Deletion

Representing a polynomial using a linked list:

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term. However, for any polynomial operation , such as addition or multiplication of polynomials , you will find that the linked list representation is more easier to deal with. First of all note that in a polynomial all the terms may not be present, especially if it is going to be a very high order polynomial.
Consider,

$5\ x^{12} + 2\ x^9 + 4x^7 + 6x^5 + x^2 + 12\ x$

Now this 12th order polynomial does not have all the 13 terms (including the constant term).
It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial.

Each node will need to store
the variable x,
the exponent and
the coefficient for each term.

It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial. Thus we need to define a node structure to hold two integers , viz. exp and coff.
Compare this representation with storing the same polynomial using an array structure. In the array we have to have keep a slot for each exponent of x, thus if we have a polynomial of order 50 but containing just 6 terms, then a large number of entries will be zero in the array. You will also see that it would be also easy to manipulate a pair of polynomials if they are represented using linked lists.

Addition of two polynomials:
Consider addition of the following polynomials
$5\ x^{12} + 2\ x^9 + 4x^7 + 6x^6 + x^3$
$7\ x^8 + 2\ x^7 + 8x^6 + 6x^4 + 2x^2 + 3\ x + 40$

The resulting polynomial is going to be
$5\ x^{12} + 2\ x^9 + 7\ x^8 + 6\ x^7 + 14x^6 + 6x^4 + x^3 + 2x^2 + 3\ x + 40$

Now notice how the addition was carried out. Let us say the result of addition is going to be

stored in a third list. We started with the highest power in any polynomial. If there was no item having same exponent , we simply appended the term to the new list, and continued with the process. Wherever we found that the exponents were matching, we simply added the coefficients and then stored the term in the new list.
If one list gets exhausted earlier and the other list still contains some lower order terms, then simply append the remaining terms to the new list. Now we are in a position to write our algorithm for adding two polynomials. Let phead1 , phead2 and phead3 represent the pointers of the three lists under consideration. Let each node contain two integers exp and coff .

Let us assume that the two linked lists already contain relevant data about the two polynomials. Also assume that we have got a function append to insert a new node at the end of the given list.p1 = phead1;

```
p2 = phead2;
 Let us call malloc to create a new node p3 to build the third list
 p3 = phead3;
 /* now traverse the lists till one list gets exhausted */
while ((p1 != NULL) || (p2 != NULL))

{
 / * if the exponent of p1 is higher than that of p2 then the next term in final list is going to be the
node of p1* /
 while (p1 ->exp > p2 -> exp )
 {
 p3 -> exp = p1 -> exp;
 p3 -> coff = p1 -> coff ;
 append (p3, phead3);
 /* now move to the next term in list 1*/
 p1 = p1 -> next;
 }
 / * if p2 exponent turns out to be higher then make p3 same as p2 and append to final list*/
while (p1 ->exp < p2 -> exp )
 {
 p3 -> exp = p2 -> exp;
 p3 -> coff = p2 -> coff ;
 append (p3, phead3);
 p2 = p2 -> next;
 }
/* now consider the possibility that both exponents are same , then we must add the coefficients
to get the term for the final list */
while (p1 ->exp = p2 -> exp )
{
```

```
p3-> exp = p1-> exp;
p3->coff = p1->coff + p2-> coff ;
 append (p3, phead3) ;
 p1 = p1->next ;
 p2 = p2->next ;
 }
}
 /* now consider the possibility that list2 gets exhausted , and there are terms remaining only in
list1. So all those terms have to be appended to end of list3. However, you do not have to do it
term by term, as p1 is already pointing to remaining terms, so simply append the pointer p1 to
phead3 */
 if ( p1 != NULL) append (p1, phead3) ;
else
 append (p2, phead3);
```

Now, you can implement the algorithm in C, and maybe make it more efficient