CS8391 - Data Structures

Regulation - 2017

UNIT-I LINEAR DATA STRUCTURES - LIST

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

* ——————————— *

Abstract Data Types (ADTs) :-

An ADT is defined to be a mathematical model of user-defined type along with a collection of all primitive operations on that model.

The definition of ADT has two parts:

(i) Description of the way in which components are related to each other.

(ii) Statements of operations that can be performed on that data type.

Objects such as lists, sets and graphs along with their operations, can be viewed as abstract data types, just as integers, reals and booleans are data types.

For example, for the set ADT, we might have the operations as union, intersection, size and complement.

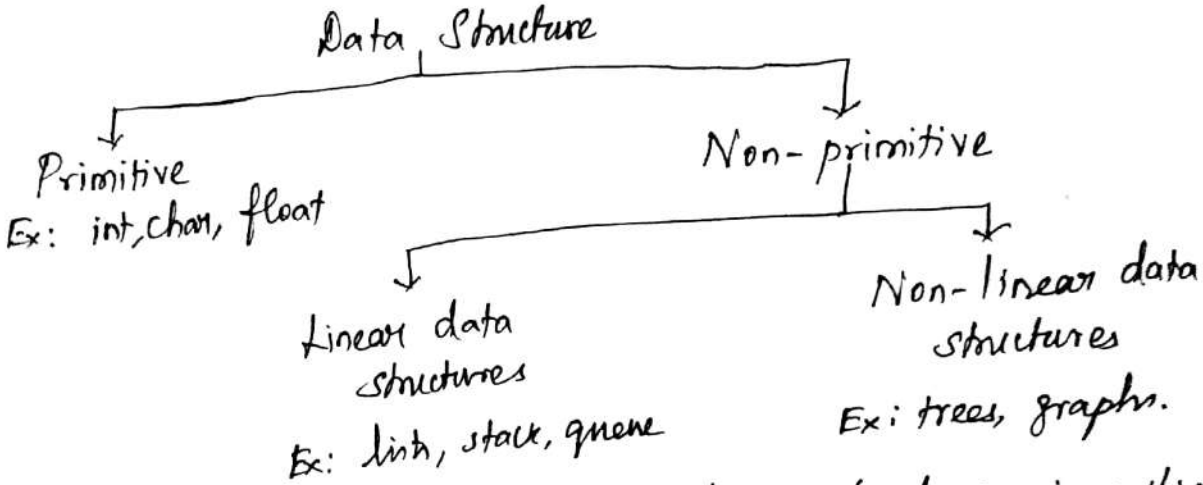
The basic idea is that the implementation of these operations is written once in the pgm, and any other part of the pgm needs to perform an operation on the ADT can be done by calling the appropriate function.

ADT = Type + Function names + Behavior of each function.

Data Structures:

A Data Structure is merely an instance of an ADT. An ADT or data structures is formally defined to be a triplet (D,F,A) where "D" stands for a set of domains, "F" denotes the set of operations and "A" represent the axioms defining the functions in "F".

Types of Data Structures:-



* Linear data structures are the data structures in which data is arranged in a list or in a straight sequence.

Ex: Array List

* Non-linear data structures are the data structures in which data may be arranged in hierarchical manner.

Ex: Trees, Graphs.

LIST ADT ;-

A list is a sequence of 0 or more elements of a given type.

We will deal with a general list of the form $A_1, A_2, A_3 \ldots, A_N$.

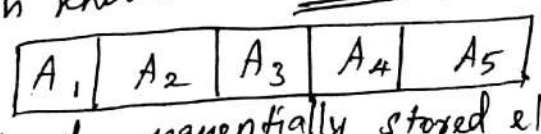
* For any list except the empty list, we say that A_{i+1} follows (or succeeds) A_i $(i < N)$ and that A_{i-1} precedes A_i $(i > 1)$.
* The first element of the list is A_1, and the last element is A_N.
* The position of element A_i in a list is i.

Operations of the LIST ADT ;-

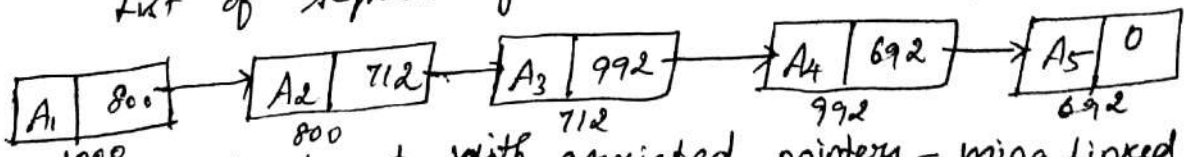
1. PrintList
2. MakeEmpty
3. Find - returns the position of the first occurrence of a key.
4. Insert - generally insert some key (value) in some position
5. Delete - " delete " " " at " "
6. FindKth - returns the element in a specified position.
7. Next & Previous - return the position of the successor and predecessor.

ARRAY IMPLEMENTATION OF LISTS :

In memory, we can store the list in two ways, one way is to store the elements (key) in sequential memory locations. This is known as arrays. The other way is to use pointers or links to associate the element sequentially. This is known as Linked Lists.



List of sequentially stored elements - using Arrays



List of elements with associated pointers - using Linked Lists

All of these instructions can be implemented jmt by ming an array. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required.

A list can be implemented using C- array. A data structure for representing a list of integers is as follows:

```
typedef struct list
{
    int data[MAX]; // MAX is a constant
    int n;
} list;
```

A set of representative functions for a list are:

1). void init (list *p); - It creates an empty list. A list will n=0 is an empty list.

2). void read (list *p); - It reads the initial set of elements and stores them in the list.

3). print (list *p); - It displays the elements of the list.

4). insert (list *p, int position, int x); - It inserts an element x at the given position in the list.

5). delete (list *p, int position); - Deletes an element from the given position is a given list.

Program:-

```
/* Array Implementation of List ADT */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 30
typedef struct list
{ int data[MAX];
  int n;
} list;
```

```
void insert (list *p, int position, int x);
void delete (list *p, int position);
int search (list *p, int x);
void print (list *p);
void read (list *p);
void init (list *l);
void main()
{
 list l;
 int x, op, position;
 clrscr();
 do
 {
  flushall();
  printf ("\n 1. create \n 2. insert \n 3. delete \n 4. search
                \n 5. print \n 6. quit ");
  printf ("\n Enter Your Choice: ");
  scanf ("%d", &op);
  switch (op)
  {
   case 1:
      read (&l);
      break;
   case 2:
      printf ("\n Enter the position and the data to be
                                          inserted: ");
      scanf ("%d %d", &position, &x);
      insert (&l, position, x);
      break;
   case 3:
      printf ("\n Enter the position: ");
      scanf ("%d", &position);
      delete (&l, position);
      break;
```

```
Case 4:
    printf("\n Enter the element to be searched: ");
    scanf("%d", &x);
    position = search(&l, x);
    if (position == -1)
        printf("\n not found");
    else
        printf("Found at the position: %d", position+1);
    break;
Case 5:
    print(&l);
    break;
default:
    break;
}
}while(op!=6);
}
void insert(list *p, int position, int x)
{
int i;
if (position < 1 || position > p->n+1)
{ printf("\n Incorrect position:");
return;
}
for (i = p->n-1; i >= position-1; i--) /* index is 1
                                        less than position*/
    p->data[i+1] = p->data[i];
    p->data[position-1] = x;
    p->n = p->n+1;
}
```

```
void deleteClist *p, int position)
{
  int i;
  if (position < 1 || position > p->n)
  {
    printf("\n Correct position");
    return;
  }
  for (i = position; i < p->n; i++)
    p->data[i-1] = p->data[i];
    p->n = p->n-1;
}
int search Clist *p, int x)
{
  int i;
    for (i=0; i<p->n; i++)
      if (x == p->data[i])
        return(i);
      return (-1);
}
void print(list *p)
{ int i;
  printf("\n");
  for (i=0; i<p->n; i++)
  printf("%d", p->data[i]);
}
void read Clist *p)
{
  int i,n;
  printf("\n Enter No. of data:");
  scanf("%d", &n);
  p->n=n;
```

```
printf("\n Enter data : ");
for (i=0; i<n; i++)
  scanf ("%d", &(p->data[i]));
}
```

Limitations of Array Implementation of List :-

1). Size of the array must be declared at the time of pgmming. This may cause underutilization or overflow.
2). It requires contignous memory
3). Insert and delete operations are time consuming.

Analysis :-

* PrintList and Find to be carried out in linear time.
* FindKth operation takes constant time.
* However, insertion and deletion are expensive.
* For example, inserting at position 0. 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. Hence the worst case of these operations is O(N).

LINKED LIST IMPLEMENTATION OF LIST ADT:

In order to avoid the linear cost of insertion & deletion, we need to ensure that the list is not stored contiguously.

The linked list consists of a series of structures, which are not necessarily adjacent in memory.

* Each structure contains the element and a pointer to a structure containing its sucessor. We call this the Next pointer.
* The last cell's Next pointer point to NULL. ANSI C specifies this NULL is zero.

Fig ① shows the general idea of a linked list. Fig ② shows the actual implementation of the list in Fig ①.

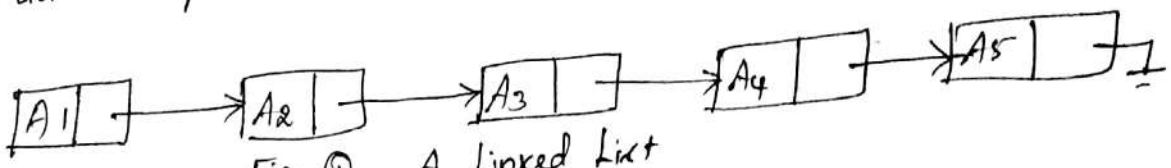


Fig ①. A Linked List

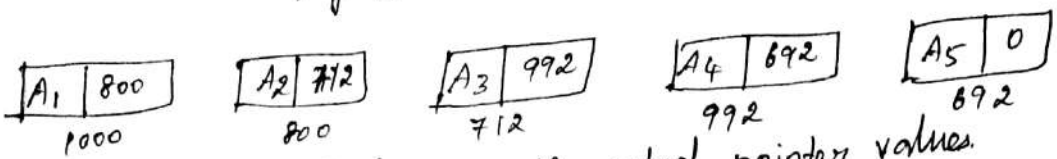
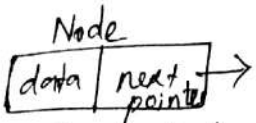


Fig ② Linked list with actual pointer values.



Single Node of a Linked List

The list contains five structures, which happen to reside in memory locations 1000, 800, 712, 992, and 692, respectively.

* The Next pointer in the first structure has the value 800, which provides the indication of where the second structure is.

To execute PrintList(L) or Find(L, key), we merely pass a pointer to the first element in the list and then traverse the list by following the Next pointers.

* This operation is clearly linear-time.

* The FindKth(L, i) takes O(i) time and works by traversing down the list. *

* The Delete command can be executed in one pointer change. Fig. ③ shows the result of deleting the third element in the original list.

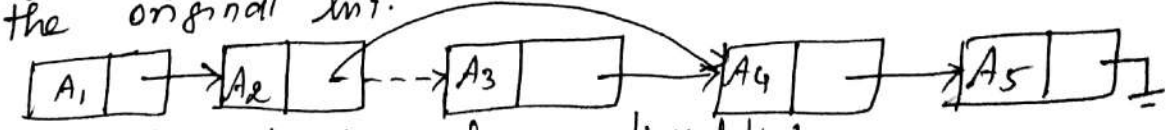


Fig ③ Deletion from a linked list

* The Insert command requires obtaining a new cell from the system by using a malloc call and then executing two pointer. The general idea is shown in Fig ④ and dashed line represent the old pointer.

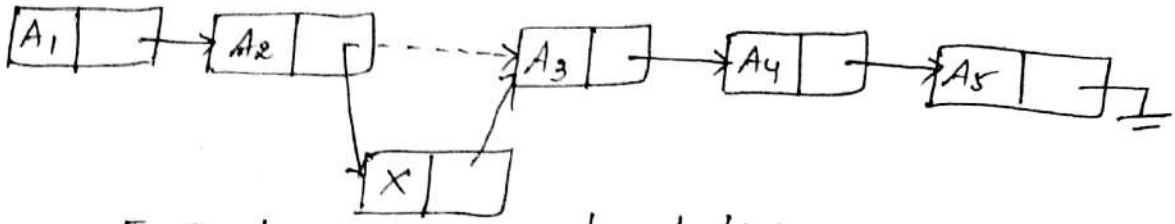


Fig ④ Insertion into a linked list

The first structure (node) in a Linked List is referred to as a header or dummy node. Fig ⑤ shows a linked list with a header representing the list $A_1, A_2, \ldots, A_5$.

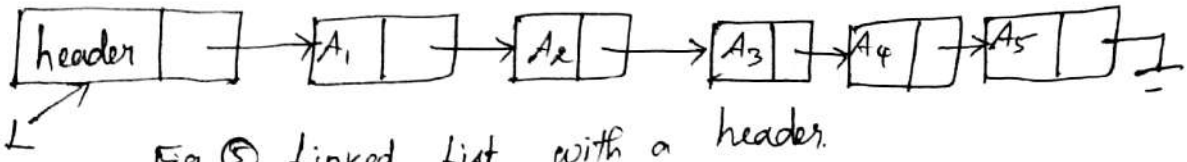


Fig. ⑤ Linked List with a header.

The Fig ⑥ shows an empty list.

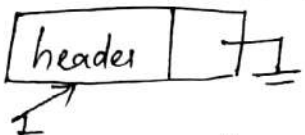


Fig ⑥ Empty list with header.

Type declarations for linked lists:

```
# ifndef _List_H

struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

List MakeEmpty(List L);
int IsEmpty(List L);
int IsLast(Position P, List L);
Position Find(ElementType X, List L);
void Delete(ElementType X, List L);
Position FindPrevious(ElementType X, List L);
```

```
void Insert(ElementType X, List L, Position P);
void DeleteList (List L);
Position Header (List L);
Position First (List L);
Position Advance (Position P);
ElementType Retrieve (Position P);
#endif
/* Place in the implementation file */
struct Node
{
  ElementType Element;
  Position Next;
};
```

To Test Linked List is empty:

```
/* Return true if L is empty
int IsEmpty (List L)
{ return L->Next == NULL;
}
```

To Test whether the current position is last:

```
/* Return true if P is the last position in list L */
int IsLast (Position P, List L)
{ return P->Next == NULL;
}
```

Find routine: - takes O(N) in the worst case

```
/* Return Position of X in L; NULL if not found */
Position Find (ElementType x, List L)
{
  Position P;
  P = L->Next;
```

```
while (P != NULL && P -> Element != x)
    P = P -> Next;
return P;
}
```

Find Previous routine: - takes O(N) in the worst case.

```
/* If X is not found, then Next field of returned
   position is NULL */  /* Assumes a header */
Position FindPrevious (ElementType x, List L)
{
  Position P;
  P = L;
  while (P -> Next != NULL && P->Next-> Element != x)
     P = P -> Next;
  return P;
}
```

Deletion routine:

```
/* Delete first occurrence of X from a list */
/* Assume use of a header node */
void Delete (ElementType x, List L)
{ Position P, TmpCell;
  P = FindPrevious( x, L);
  if (! IsLast (P, L))  /* Assumption of header use */
                        /* x is found; delete it */
  {
    TmpCell = P -> Next;
    P -> Next = TmpCell -> Next;  /* Bypass deleted cell */
    free (TmpCell);
  }
}
```

Insertion routine:

```
/* Insert (after legal position P)*/
/* Header implementation assumed */
/* Parameter L is unused in this implementation */
void Insert (ElementType x, List L, Position P)
{
  Position TmpCell;
  TmpCell = malloc (sizeof (struct Node));
  if (TmpCell == NULL)
      FatalError ("Out of Space!!!");
  TmpCell -> Element = x;
  TmpCell -> Next = P -> Next;
  P -> Next = TmpCell;
}
```

Delete a Linked List: - takes O(N log N) time to dispose of N cells.

```
void DeleteList (List L)
{
  Position P, Tmp;
  P = L -> Next;  /* Header assumed */
  L -> Next = NULL;
  while (P != NULL)
  {
    Tmp = P -> Next;
    free (P);
    P = Tmp;
  }
}
```

SINGLY LINKED LIST:-

Implementation: - creation, display, count

```
#include <conio.h>
#include <stdio.h>
typedef struct node
{
  int data;
  struct node *next;
}node;
 node *create(int);
 void print(node *);
 int count(node *);
 void main()
 {
   node * HEAD;
    int n, number;
    printf("\n No. of items: ");
    scanf("%d",&n);
    HEAD = create(n);
    print(HEAD);
    number = count(HEAD);
    printf("\n No. of nodes = %d", number);
 }
 node *create(int n)
 {
   node *head, *P;
    int i;
    head = (node *) malloc(sizeof(node));
    head -> next = NULL;
    scanf("%d", &(head -> data));
    P = head;
    // create subsequent nodes
    for(i=1; i<n; i++)
    {
      p -> next = (node *) malloc(sizeof(node));
```

```
        //new node is inserted as the next node after P
        P = P->next;
        scanf("%d", &(P->data));
        P->next = NULL;
    }
    return(head);
}
void print(node *P)
{
    while(P!=NULL)
    {
        printf("%d\t", P->data);
        P = P->next;
    }
}
int count(node *P)
{
    int i=0;
    while(P!=NULL)
    {
        P = P->next;
        i++;
    }
    return i;
}
```

Inserting an item:

It has ③ situations:

1). Insertion at the front of the list
2). Insertion in the middle of the list
3). Insertion at the end of the list.

Alg for placing a new item at the front of the list:

1). Obtain space for new node
2). Assign data to the data field of the new node.
3). Set the next field of the new node to the beginning of the linked list

4) Change the reference pointer of the linked list to point to the new node.

Alg for inserting the new data after a node N1:

1). Obtain space for new node.
2). Assign value to its data field.
3). Search for the node N1.
4). Set the next field of the new node to point to N1->next
5). Set the next field of N1 to point to the new node.

Code to insert a data in a linked list after a given data:

```
node *insert (node *head, int x, int key)
{
  /* data x in to be inserted after the key */
  /* if key is -1 then x is to be inserted as a front node*/
  node *P, *q;
    /* obtain space for the new node*/
  P = (node *) malloc (sizeof (node));
    /* store x in the new node*/
  P->data = x;
  if (key == -1)
  {
    /* insert the node at the front of the list */
    P->next = head;
    head = P;
  }
```

```
else
{ /* search for the key in the linked list */
  q = head;
  while (key != q -> data && q != NULL)
      q = q -> next;
  if (q != NULL)
  {   /* if the key is found */
      p -> next = q -> next;
      q -> next = p;
  }
}
return (head);
}
```

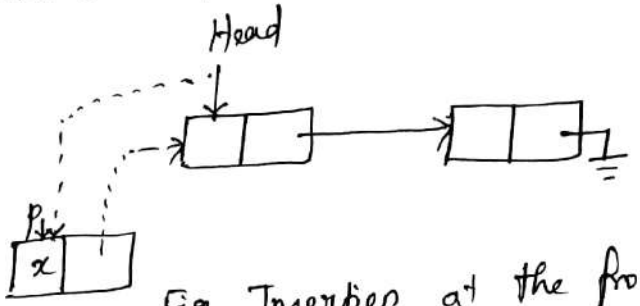


Fig. Insertion at the front

p -> data = x
p -> next = head
Head = P.

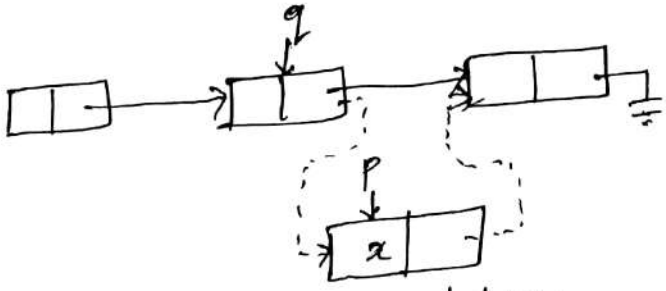


Fig. Insertion in between

p -> next = q -> next
q -> next = P

Inserting an item at the end of the linked list:

Alg:

1). Acquire memory for new node.
2). Assign value to the data field and make its 'next' field to NULL.
3). If head is NULL, then go to step 6.
4). Position a pointer q on the last node by traversing the linked list from the first node and until it reaches the last node.
5). Store the address of the newly acquired node, pointed by P, in the next field of node pointed by q.
6). Stop.

Code:

```
node *insert_end (node *head, int x)
{
  node *p, *q;
  p=(node *) malloc (sizeof (node));
  p->data = x;
  p->next = NULL;
  if (head == NULL)
     return(p);
  q = head;
  while (q->next != NULL)
     q = q->next;
  q->next = p;
  return (head);
}
```

Insertion of a data 'x' in a given location

Alg:

1). Acquire memory for new node with its address in pointer P.

2). Assign value to data field and make its next field NULL.

3). if (LOC == 1) then insert the node, pointed by P, at the beginning of the linked list. This can be done in (2) steps.
   (i) Store head in the next field of the node pointed by P.
   (ii) Move head to the newly connected node.
   · go to step 7.

4). if (LOC > 1) then position a pointer q on (LOC-1)th node.

5). if q is NULL then report "overflow" and terminate the alg. go to step 7.

6). Insert the node pointed by P, after the node pointed by q.

7). Stop

Code:

```
node *insert_LOC(node *head, int x, int LOC)
{
 node *p, *q;
 int i;
 p = (node *)malloc(sizeof(node));
 p->data = x;
 p->next = NULL;
 if (LOC == 1)
 {
  p->next = head;
  return(p);
 }
 q = head;
 for(i=1; i<LOC-1; i++)
    if (q != NULL)
       q = q->next;
    else
    {
     printf("\n Overflow");
     return(head);
    }
 p->next = q->next;
 q->next = p;
 return(head);
}
```

Deleting an Item:-

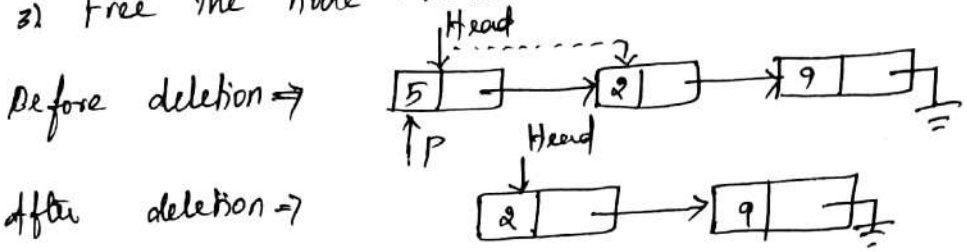
(20)

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed.

1. Deleting the first item.
2. " " last "
3. " from the middle of the list.

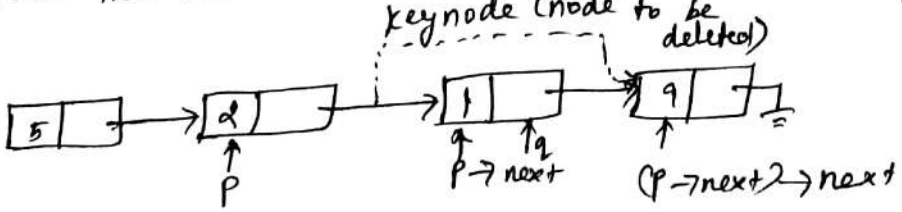
Alg for deleting the first item:

1). Store the address of the first node in a pointer variable P.
2). Move the head to the next node.
3) Free the node whose address is stored in the pointer variable P.



Alg for deleting a node from middle:

1). Store the address of the preceding node in a pointer variable P. Node to be deleted is marked as key node.
2). Store the address of the key node in a pointer variable q, so that it can be freed subsequently.
3). Make the successor of the key node as the successor of the node pointed by P.
4). Free the node whose address is stored in the pointer variable q.



code:

```
node *delete (node *head, int x)
{
  node *p, *q;
```

```
if (x == head -> data)
{   /* deleting the first item */
    p = head;
    head = head -> next;
    free (p);
}
else
{   while (x != (p -> next) -> data && p -> next != NULL)
        p = p -> next;
        if (p -> next != NULL)   /* if x exists */
        {
            q = p -> next;
            p -> next = (p -> next) -> next;
            free (q);   /* release space of the key node */
        }
    }
    return (head);
}
```

Deletion of Last node:

Alg: - Linked list is refernced by the pointer 'head'.

In order to delete the last node, we must position a pointer q on the last but one node. Address of the node to be deleted is stored in pointer P, so that the memory allocated to it can be freed.

1). If the first node itself is the last node then make the linked list empty.
2). Otherwise position a pointer q on last but one node
2). Delete the last node.
4). Stop

Code:

```
node *delete_last (node *head)
{
  node *p,*q;
  if (head->next==NULL)
  {
    free (head);
    head=NULL;
    return(head);
  }
  q=head;
  while (q->next->next!=NULL)
    q=q->next;
  p=q->next;
  free(p);
  q->next=NULL;
  return(head);
}
```

Deletion of a Node at Location LOC:

Alg: - Linked list is referenced by the pointer 'head'.

1). if (LOC==1) then the node to be deleted is the first node
   (a). Store the address of the first node in the pointer P.
   (b). Move head to the next node.
   (c). Free the memory allocated to the node to be deleted. then go to step 5.
2). Otherwise, position a pointer q on $(LOC-1)^{th}$ node.
3). if q is NULL, then report underflow and terminate the alg go to step 5.
4). Delete the desired node.
5). Stop.

Code:

```
node *delete_LOC (node *head, int LOC)
{
  node *p, *q;
  int i;
  if (LOC == 1)
  {
    p = head;
    head = head -> next;
    free (p);
    return (head);
  }
  q = head;
  for ( i = 1; i < LOC-1; i++)
      q = q -> next;
  if (q == NULL)
  {
    printf ("\n Underflow ");
    return (head);
  }
  p = q -> next;
  q -> next = p -> next;
  free(p);
  return (head);
}
```

Delete a Linked List, Referenced by pointer head:

Alg:

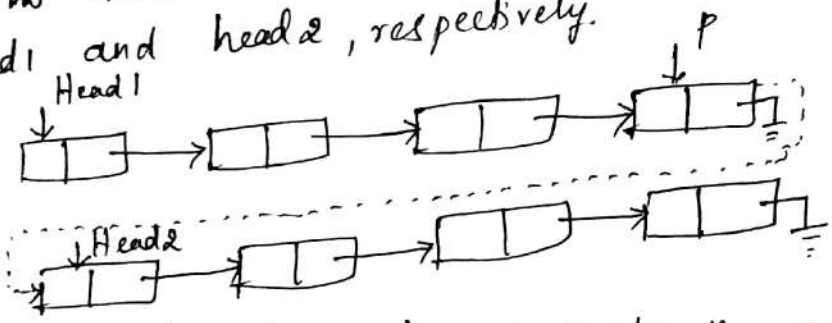
1). if head == NULL then goto step 3.
2). otherwise delete the first node
3). stop.

code:

```
node *delete_list (node *head)
{
  node *p;
  while (head != NULL)
  {
    p=head;
    head = head → next;
    free(p);
  }
  return (head);
}
```

Concatenation of Two Linked Lists:-

Alg: Let us assume that the two linked lists are referenced by head1 and head2, respectively.



1). If the first linked list is empty then return head2.
2). If the second linked list is empty then return head1.
3). Store the address of the starting node of the first linked list in a pointer variable, say P.
4). Move P to the last node of the linked list through simple linked list traversal technique.
5). Store the address of the first node of the second linked list in the next field of the node pointed by P. Return head1.

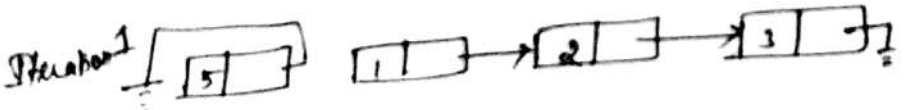
Code:

```
node * concatenate (node *head1, node *head2)
{
  node *p;
  if (head1 == NULL)   /* if the first list is empty */
    return (head2);
  if (head2 == NULL)   /* if the 2nd list is empty */
    return (head1);
  p = head1;           /* place P on the first node of 1st list */
  while (p->next != NULL) /* Move P to the last node */
    p = p->next;
  p->next = head2; /* address of the first node of the 2nd
                      list stored in the last node of the 1st
                      list */
  return (head1);
}
```

Inversion of Linked List:

A linked list can be reversed by changing the direction of the pointer iteratively.

Original linked list



Linked list after reversal of the first node.

Linked list after reversal of the first two node.

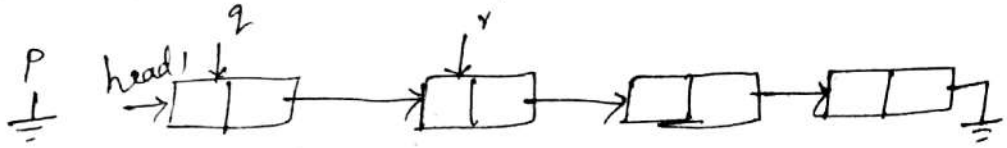
Linked list after reversal of all nodes.

Als: Let us take three pointer variables p, q, and r.

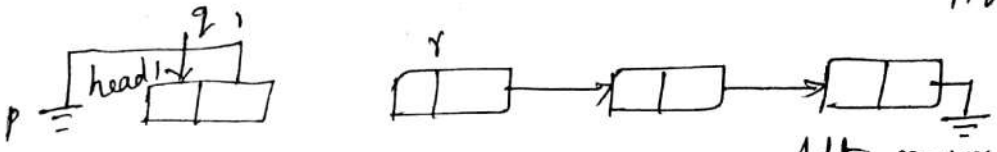
P references the linked list reversed so far. q points to the linked list to be reversed. r points to the next node q.

Initially,

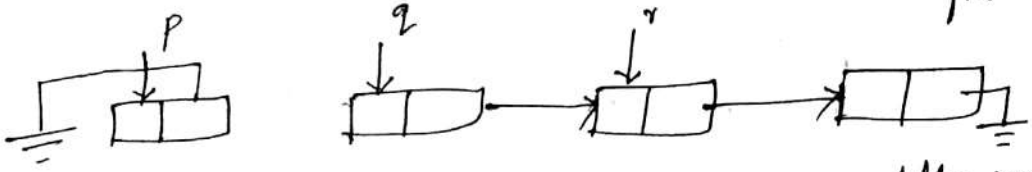
$p = NULL$, $q = head$, $r = q \rightarrow next$



initial value of p, q, r.



After reversing the first node.



After moving p, q, r forward

Code:

```
node * invert(node *head)
{
  node *p,*q,*r;
  p=NULL, q=head ; r=q->next; /* initial values of p,q,r*/
  while (q!=NULL)  /* until all nodes have been reversed*/
  {
    q->next = p;    /* move p,q,r forward by a node*/
    p=q;
    q=r;
    if (r!=NULL)
        r=r->next;
  }
  return(p);
}
```

Find routine

Alg:

In order to search an element in a linked list, we start traversing the linked list from the first node. Traversal ends with a success if the element is found. If the element is not found then search ends in a failure.

1). P = head.
2). if p → data is equal to x, then return p
3. otherwise search continues with p = p → next.
4. Stop.

Code:

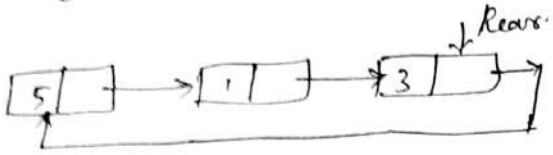
```
node find(node *head, int x)
{
  node *p;
  p = head;
  while (P != NULL && p->data != x)
      p = p->next;
  return p;
}
```

Adv. of Linked List:

1). Linked list is an example of dynamic data structure. They can grow and shrink during execution of the pgm.
2). Representation of linear data structure (polynomial, stack and queue) can easily be represented using this.
3). Efficient memory utilization - memory allocation is done by the need.
4). Insertion & deletion are easier and efficient. and it can be carried out in constant time.

CIRCULARLY LINKED LISTS:

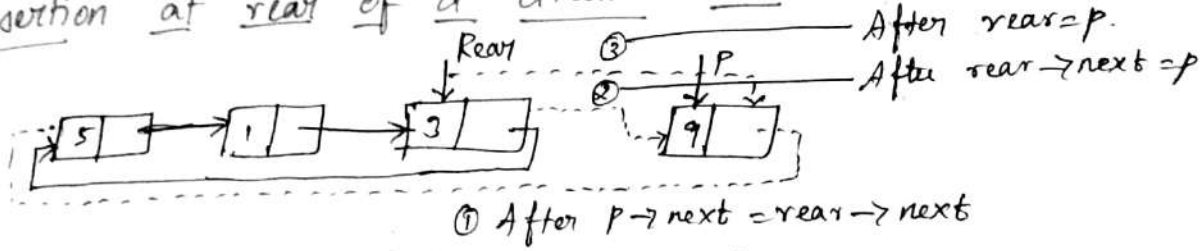
In a circular linked list, last node is connected back to the first node. In some applications, it is convenient to use circular linked list. A queue data structure can be implemented using a circular linked list, with a single pointer "rear" as the front node can be accessed through the rear node.



Insertion of a node at the start or end of a circular linked list identified by a pointer to the last node of the linked list. It takes a constant amount of time. It is irrespective of the length of the linked list.

Alg for traversing a circular list is slightly different than the same alg for a singly connected linked list because "NULL" is not encountered.

Insertion at rear of a circular linked list:




```
node *insert_rear (node *rear, int x)
{
    node *p;
    p = (node *) malloc (sizeof (node));
      /* acquire memory for the current data */
    p -> data = x;
    if (rear == NULL)
    {
```

```
    /* inserting in an empty linked list */
    rear = p;
    /* node is connected back to the same node */
    p->next = p;
    return (rear);
  }
 else
 {
   p->next = rear->next;
   rear->next = p;   /* node p is made a part of the circle */

   rear=p;
   return(rear);
 }
}
```

Insertion at the front of the circular linked list:

```
node *insert_front (node *rear, int x)
{
   node *p;
   p = (node *) malloc (sizeof (node));
   p->data = x;
   if (rear == NULL)
   {
      rear = p;
      p->next = p;
      return (rear);
   }
   else
   {
      p->next = rear->next;
      rear->next = p;
      return (rear);  /* rear is not moved after insertion */
   }}
```

Traversing a circular linked list / Display values:

```
void print(node *rear)
{
  node *p;
  if (rear != NULL)
  {
    p = rear->next; /* start traversing from the front */
    do
    {
      printf("\n %d", p->data);
      p = p->next;
    } while (p != rear->next);
  }
}
```

Couting of nodes in a circular linked list:

```
int count(node *rear)
{
  node p;
  int n=0;
  if (rear == NULL)
    return(0);
  p = rear->next;
  do
  {
    n = n+1;
    p = p->next;
  } while (p != rear->next);
  return(n);
}
```

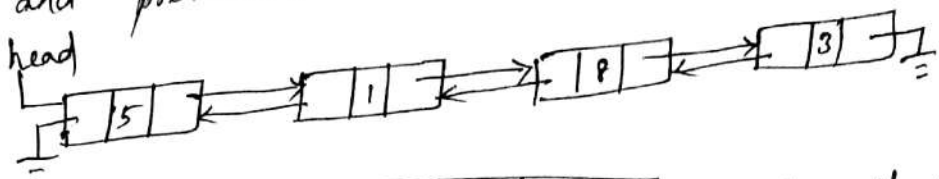
Deletion of a node:

```
node * delete_cir_loc(node *head, int loc)
{
  node *p, *q;
  int i;
```

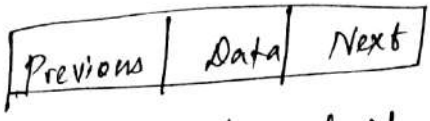
```
if (loc == 1)
{
  if ( head -> next == head)
   {
     free (p);
     return (NULL);
   }
   else
   {
     p = head -> next;
     head -> next = p -> next;
     free (p);
     return (head);
   }
}
q = head -> next;
for ( i = 1; i < loc - 1; i++)
    if (q != head)
        q = q -> next;
    else
      {
        printf ("\n  Underflow ");
        return (head);
      }
  p = q -> next;
  q -> next = p -> next;
  free (p);
  return (head);
}
```

DOUBLY LINKED LIST:

A node in a doubly linked list has 3 fields, say "data", "next" and "previous".



A doubly linked list



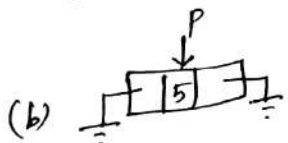
Structure of the node

* Each node has two link fields, one linking in the forward direction and the other in the backward direction.
* We can able to traverse the list in ~~either~~ both directions.

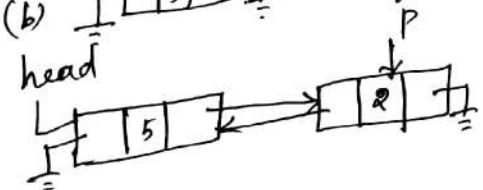
Creation :-

- Initially head is NULL

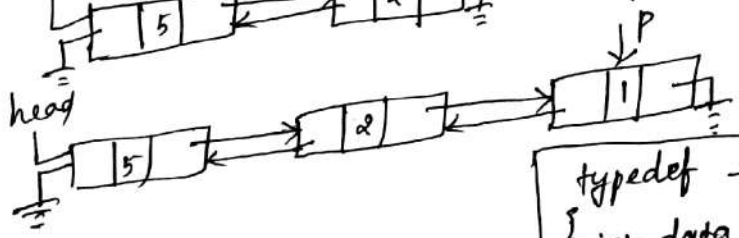
(a)



(b) - After insertion of 5



(c) - After insertion of 5 & 2.



(d)

```
typedef struct dnode
{ int data;
  struct dnode *next, *prev;
}dnode;
```

Alg:

```
dnode *p, *q;
```

1). Acquire memory for the new data

2). Store x in the newly acquired node.

3). Make previous and next field as NULL:

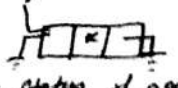


Fig. Status of node after step1 to step3

Case I: Inserting in an empty list (head == NULL)

```
if (head == NULL)
{
  head = p, q;
}
```

Case II: Inserting in an non-empty list

```
else
{
  p->next = q;
  q->prev = p;
}p=q;
```

Code to create doubly linked list & traverse forward & reverse direction

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
typedef struct dnode
{ int data;
  struct dnode *next, *prev;
}dnode;
 dnode *create();
 void print_forward(dnode *);
 void print_reverse(dnode *);
 void main()
 {
   dnode *head;
   head = NULL;   // initially the list is empty
   head = create();
   printf("\n Elements in forward direction: ");
   print_forward(head);
   printf("\n Elements in reverse direction: ");
   print_reverse(head);
 }
```

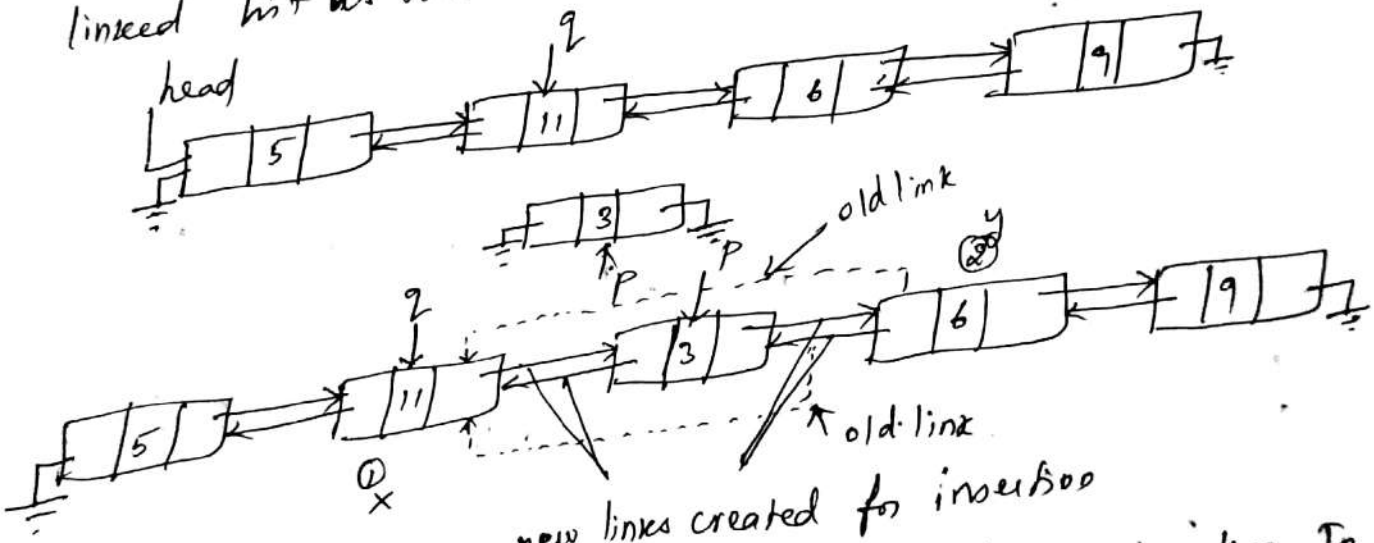
by
3.

```
dnode *create()
{
  dnode *h, *p, *q;
  int n, x;
  h = NULL;
  printf("\n Enter the no. of elements: ");
  scanf("%d", &n);
  for(i=0; i<n; i++)
  {
     printf("\n Enter the next data: ");
     scanf("%d", &x);
     q = (dnode *) malloc(sizeof(dnode));
     q->data = x;
     q->prev = q->next = NULL;
     if(h==NULL)
        p=h=q;
     else
     {
        p->next = q;
        q->prev = p;
        p=q;
     }
  }
  return(h);
}
void print_forward(dnode *h)
{
  while(h!=NULL)
  {
     printf("<- %d->", h->data);
     h = h->next;
  }
}
```

```
void print_reverse (dnode *h)
{
  while (h->next != NULL) //go to the last node
    h = h->next;          //print while moving in the reverse path
  while (h != NULL)
  {
    printf (" <- %d ->", h->data);
    h = h->prev;
  }
}
```

Insertion of node P after a node pointed to by q:

Consider a new node pointed to by 'p' is to be inserted after the node pointed to by q in a doubly linked list as shown in Fig.



Links of two nodes are altered during insertion. In the above fig., a new node pointed to by p is inserted b/n the two nodes x and y. Following links must be changed for proper insertion of node pointed by p b/n x and y.

1). Right link of x
2). Left link of y
3). Left and right links of node pointed to by p.

Instructions for making above changes:

```
p->next = q->next;      // right link of set to y
p->prev = q;            // left link of p set to x
q->next = p;            // right link of x set to p.
(p->next)->prev = p;    // left link of y set to p.
```

Code:

```
void insert (dnode *p, dnode *q)
{
    p->next = q->next;
    p->prev = q;
    q->next = p;
    if (p->next != NULL)    /* insertion at the end */
        p->next->prev = p;
}
```

code for inserting a node pointed to by p before a node pointed to by q:

```
dnode *insert(dnode *head, dnode *p, dnode *q)
{
    if (q->prev == NULL)
    {   // insert at the beginning
        p->next = q;
        p->prev = NULL;
        q->prev = p;
        return(p);
    }
    else
    {   p->prev = q->prev;
        p->next = q;
        p->prev->next = p;
        q->prev = p;
        return(head);
    }
}
```

Insertion at the beginning of DLL:-

```
dnode *insert_begin(dnode * head, int x)
{
   dnode *p;
   p=(dnode *)malloc(sizeof(dnode));
   p->data=x;
   p->prev=NULL;
   p->next=head;
   if(head!=NULL)  /* not an empty list */
      head->prev=p;
   return(p);
}
```

Insertion at the end of DLL:-

```
dnode *insert_end(dnode *head, int x)
{
   dnode *p,*q;
   p=(dnode *)malloc(sizeof(dnode));
   p->data=x;
   p->prev=p->next=NULL;
   if(head==NULL)      /*empty list*/
      return(p);
   /* go to the last node */
   q=head;
   while(q->next!=NULL)
      q=q->next;
   p->prev=q;
   q->next=p;
   return(head);
}
```

Deletion of a node in DLL:

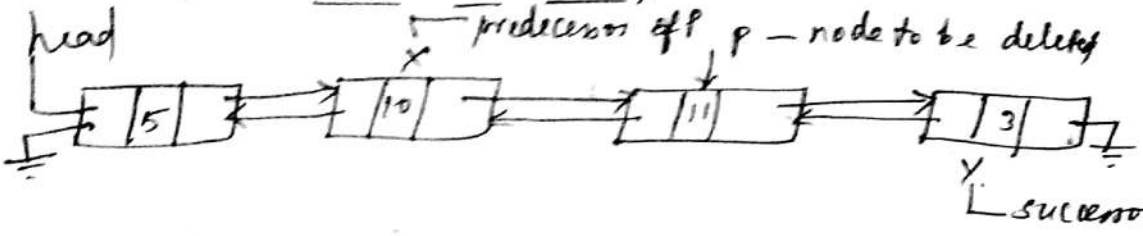


Fig. Node pointed to by P is to be deleted.

When a node, pointed to by P is to be deleted then its predecessor node x and its sucessor node y will be affected.

* Right link at x should be set to y.
* Left link at y should be set to x.
* Release the memory allocated to the node pointed to by P.

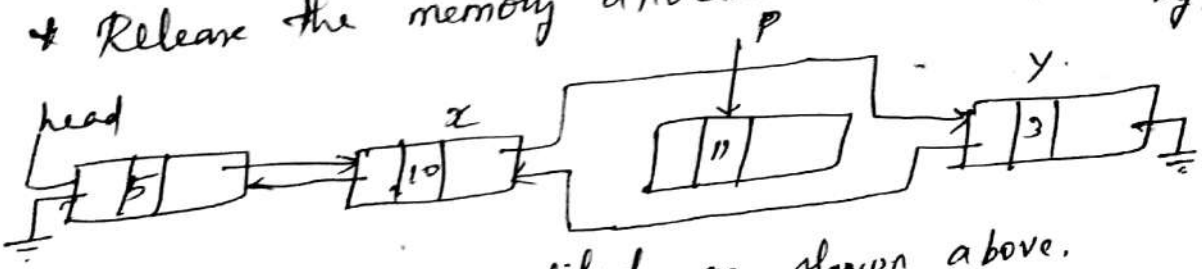


Fig. Links to be modified as shown above.

Code:

```
dnode *delete_double(dnode *head, dnode *p)
{
  if(p==head)          /* deleting the head node */
  {
    p->next->prev = NULL;
    head = p->next;
    free(p);
    return(head);
  }
```

```
p->prev->next = p->next;
if (p->next != NULL)    /* not the last node */
    p->next->prev = p->prev;
free(p);
return(head);
}
```

* Special care should be taken to delete the first and last node.
* If the node to be deleted is the first node then head should be advanced to the next node.
* If the node to be deleted is the last node then there is no right side node.

APPLICATIONS OF LISTS:-

1). Polynomial ADT:

A polynomial $P(x)$ of degree n is defined as

$$P(x) = a_0 + a_1 x^1 + a_2 x^2 + \cdots + a_n x^n.$$

where a_n is any real number and n is an integer. A polynomial can be thought of as the ordered list of non-zero terms. Each term is a 2-tuple containing power and coefficient

For example, the polynomial $5 + 6x^2 - 9x^4$ is represented as $(0,5), (2,6), (4,-9)$. Tuple $(0,5)$ stands for $5x^0$. The linked list representation of this polynomial is shown in Fig.

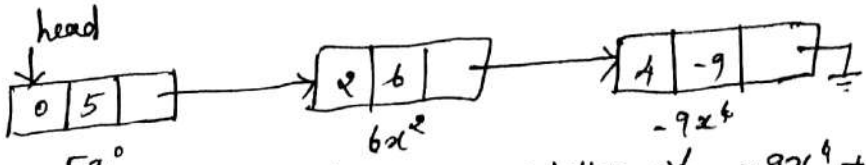


Fig. Linked list representation of $-9x^4 + 6x^2 + 5$

Operations:

1) Creation of a polynomial
2) Printing " "
3) Addition of two polynomials
4) Multiplication of " "
5) Evaluation of a polynomial.

1) Creation:

Code:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef struct pnode
{
  int pow;
  float coeff;
  struct pnode *next;
}pnode;
pnode create (int);
void print (pnode *);
pnode *addpoly (pnode *, pnode *);
pnode *multiplypoly (pnode *, pnode *);
float evaluate (pnode *, float );
void main()
{
  pnode *HEAD;
  int n;
  printf ("\n Enter the no. of terms: ");
  scanf ("%d", &n);
  HEAD = create (n);
  print (HEAD);
}
```

Creation:

Creation of a linked list representing a polynomial is similar to that of a normal linked list. Each node contains two data field, namely power and coefficient. Since, polynomial is an ordered list, data for each term must be entered in ascending order on power

```
pnode *create(int n)
{
  pnode *head, *p;
  int i;
  //Create the first node:
  head = (pnode *) malloc (sizeof (pnode));
   p = head;
   head -> next = NULL;
   printf(" Enter the power and coefficient: ");
   scanf(" %d \t %f ", &(p->pow), &(p -> coeff));
   for(i=1; i<n ; i++)
   { //create subsequent nodes
   p -> next = (pnode *) malloc (sizeof (pnode));
    p = p->next;
    p->next = NULL;
    printf("\n Enter power and coefficient: ");
    scanf (" %d \t %f ", &(p->pow), &(p -> coeff));
   }
  return (head);
}
```

Print:

```
void print(pnode *p)
{
 printf("\n");
 while(p!=NULL)
 {  printf("%.5.2fx^%d\t", p->coeff, p->pow);
    p=p->next;
 }}
```

Addition:

```
pnode *addpoly(pnode *p1, pnode *p2)
{
 pnode *p3,*r3;
 p3=NULL;
 while(p1!=NULL && p2!=NULL)
 {
   if(p3==NULL)
   {
     p3=r3=(pnode *)malloc(sizeof(pnode));
     r3->next=NULL;
   }
   else
   {
     r3->next=(pnode *)malloc(sizeof(pnode));
     r3=r3->next;
     r3->next=NULL;
   }
   if(p1->pow<p2->pow)
   {
     r3->pow=p1->pow;
     r3->coeff=p1->coeff;
     p1=p1->next;
   }
```

```
else if (p2->pow < p1->pow)
{
   r3->pow = p2->row;
   r3->coeff = p2->coeff;
   p2 = p2->next;
}
else
{
   r3->pow = p1->row;
   r3->coeff = p1->coeff + p2->coeff;
   p1 = p1->next;
   p2 = p2->next;
}
}
//insert the remaining terms of p1 into p3.
while (p1 != NULL)
{ if (p3 == NULL)
   { p3 = r3 = (pnode *)malloc (sizeof (pnode));
     r3->next = NULL;
   }
   else
   {
    r3->next = (pnode *) malloc(sizeof (pnode));
    r3 = r3->next;
    r3->next = NULL;
   }
   r3->pow = p1->pow;
   r3->coeff = p1->coeff;
   p1 = p1->next;
}
```

```
// insert the remaining terms of p2 into p3
while (p2 != NULL)
{
  if (p3 == NULL)
  {
    p3 = r3 = (pnode *) malloc (sizeof (pnode));
    r3 -> next = NULL;
  }
  else
  {
    r3 -> next = (pnode *) malloc (sizeof (pnode));
    r3 = r3 -> next;
    r3 -> next = NULL;
  }
  r3 -> pow = p2 -> pow;
  r3 -> coeff = p2 -> coeff;
  p2 = p2 -> next;
}
return (p3);
}
```

In order to add two polynomials denoted by "p" and "q", both the linked list are scanned term by term.

* Whenever the powers of the current terms of "p" and "q" are same, their coefficients are added to obtain the coefficient of the term of the new polynomial.
* The exponent part of this term is the same as that of the current term of either of "p" or "q".
* Subsequently, both pointers pointing to the current nodes are advanced to point to the next node.
* If the power part of one polynomial, say P, is less than a new node is created which is a copy of the term of "p" and this new node is added to the resultant list.

* When one of the lists of "p" or "q" is exhausted then the remaining nodes of the other list is simply copied to the resultant list.

First polynomial $5 + 9x + 10x^2 + 13x^5 + 15x^7$ (P_1)

Second " $6 + 13x^2 + 9x^3$ (P_2)

Resultant " $11 + 9x + 23x^2 + 9x^3 + 13x^5 + 15x^7$. ($P_1 + P_2$)

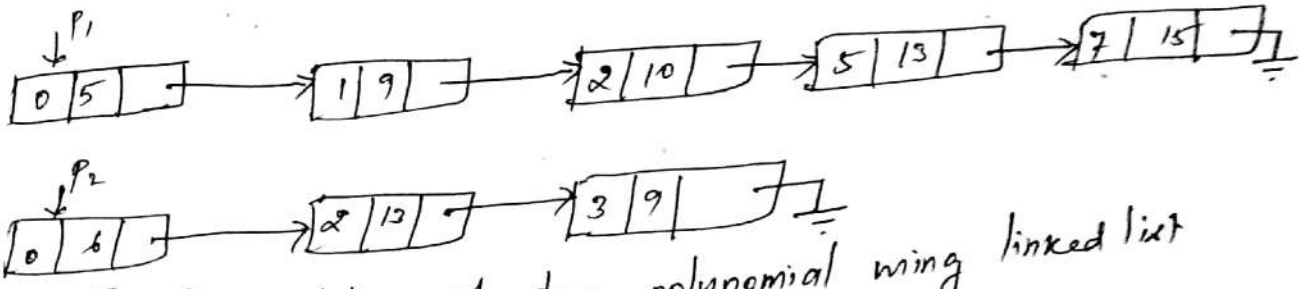
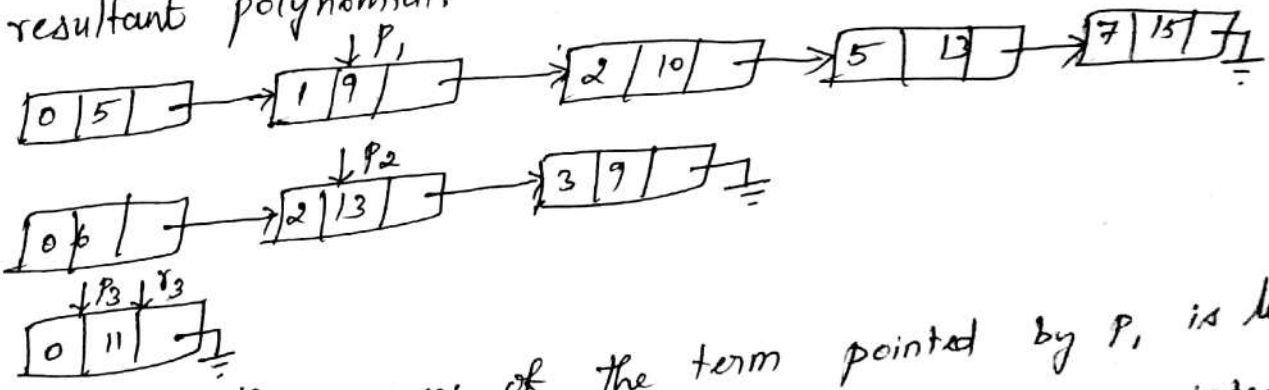
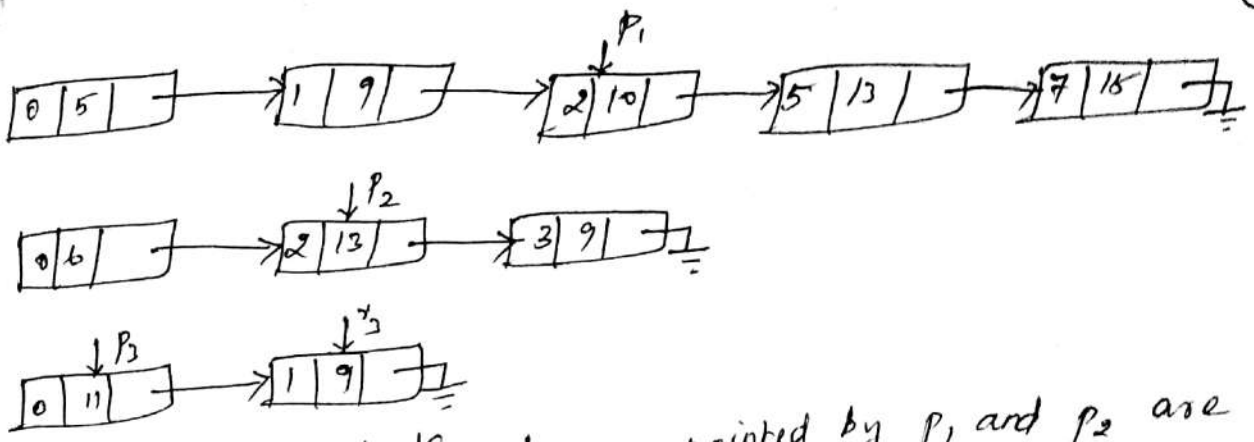


Fig. Representation of two polynomial using linked list

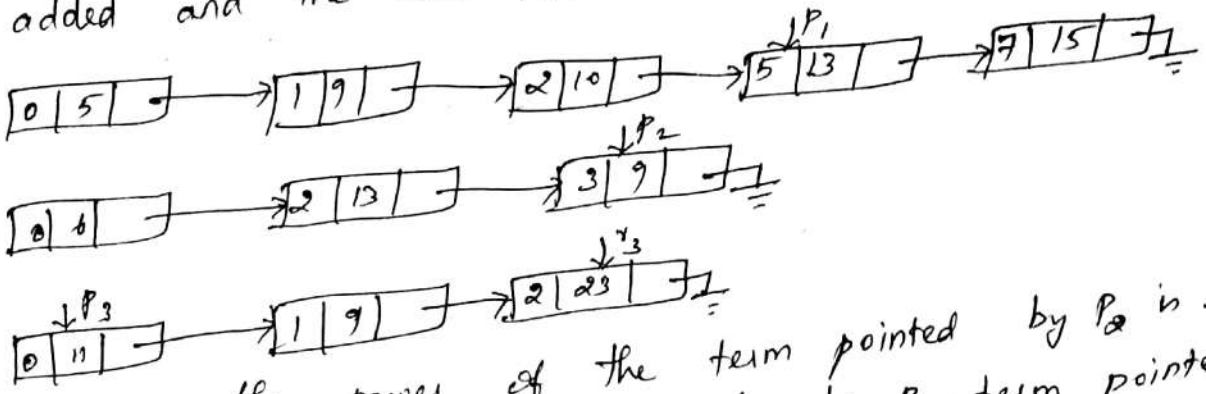
Since, the power of the terms pointed by P_1 and P_2 are same, coefficients are added and the new term is inserted into the resultant polynomial P_3. Pointer r_3 helps in inserting subsequent terms at the rear end of the resultant polynomial.



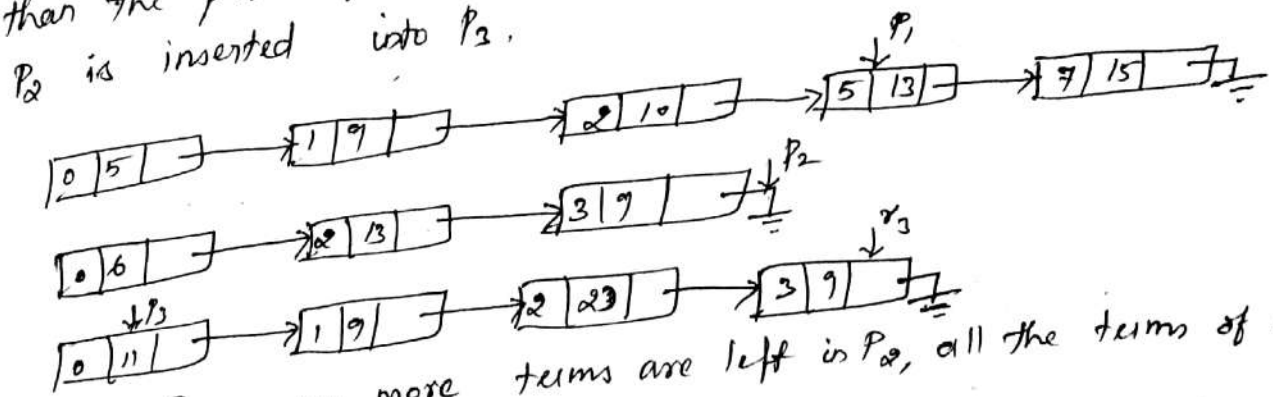
Since, the power of the term pointed by P_1 is less than the power of the term pointed by P_2, term pointed by P_1 is added to P_3. Term is added as a next node of r_3. P_1 and r_3 are advanced by a node.



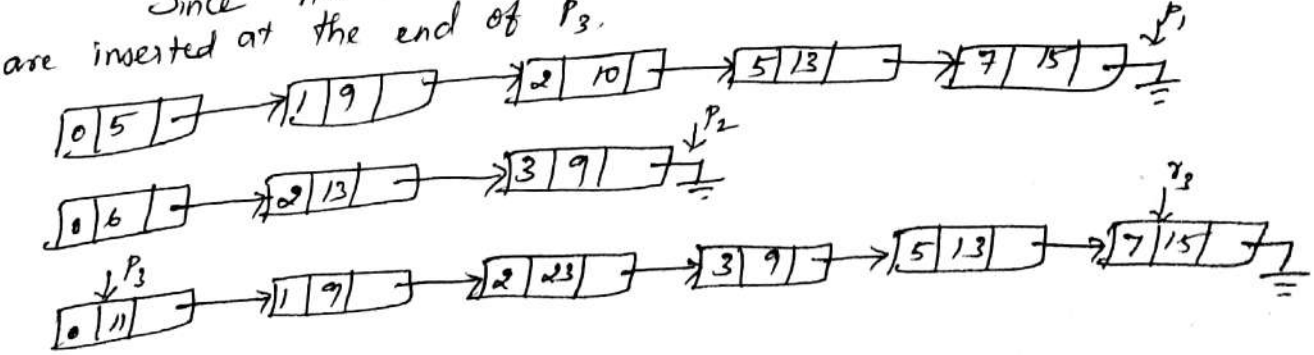
Coefficients of the terms pointed by P_1 and P_2 are added and the new term is inserted into P_3.



Since the power of the term pointed by P_2 is less than the power of the term pointed by P_1, term pointed by P_2 is inserted into P_3.



Since no more terms are left in P_2, all the terms of P_1 are inserted at the end of P_3.



Multiplication of two polynomials:

$5 + 2x + 9x^3 + 3x^5$ — P_1

$3 + x^2 + 2x^4$ — P_2

$15 + 16x + 27x^3 + 9x^5$ — P_3 [P_1 is multiplied with 3]

$5x^2 + 2x^3 + 9x^5 + 3x^7$ — P_4 [P_1 is multiplied with n^2]

$10x^4 + 4x^5 + 18x^7 + 6x^9$ — P_5 [P_1 is multiplied with $2n4$]

P_3, P_4 and P_5 are added to produce the resultant polynomial.

Alg:

1). R3 and temp are of polynomial type

2). R3 ← NULL

3). Temp ← NULL

4). while (P2 != NULL)

{

multiply the current term of P2 with the polynomial P1 with its result in temp

temp ← (polynomial P1) × (term pointed by P2)

add the polynomial "temp" to the final polynomial "R3".

R3 = R3 + temp;

P2 = (P2 → next); // skip the current term of P2

}

5). End.

Code:

```
pnode * multiply (pnode *P1, pnode *P2)
{
  pnode *temp, *r3;
  r3 = NULL;
  while (P2 != NULL)
  {
```

```
    temp = multiterm (p1, p2);
    /* function multiterm multiplies the polynomial p1
       with the current term of p2 */
    r3 = addpdy (r3, temp);
    p2 = p2 -> next;
  }
  return (r3);
}
pnode * multiterm (pnode *p1, pnode *x)
{
  pnode *new, *r1;
  new = NULL;
  while (p1 != NULL)
  {
    if (new == NULL)
    {
      r1 = new = (pnode *) malloc (sizeof (pnode));
      r1 -> next = NULL;
    }
    else
    {
      r1 -> next = (pnode *) malloc (sizeof (pnode));
      r1 = r1 -> next;
      r1 -> next = NULL;
    }
    r1 -> coeff = p1 -> coeff * x -> coeff;
    r1 -> pow = p1 -> pow + x -> pow;
    p1 = p1 -> next;
  }
  return (new);
}
```

Evaluation:

```
float evaluate (pnode *p1, float val)
{ float x;
  x = 0.00;
  while (p1 != NULL)
  {
    x = x + p1->coeff * pow (val, p1->pow);
    p1 = p1->next;
  }
  return (x);
}
```

Merits of Doubly Linked List Over Singly Linked List:

1). In a doubly linked list, one can easily find both predecessor and successor of a node.
2). A doubly linked list can be traversed in both forward and backward directions.
3). A doubly linked list can be used for memory mgmt and implementation of queue.

Differentiate doubly and circular linked list:

1). A doubly linked list can be traversed in both forward and backward directions. A circular linked list can be traversed only in the forward direction.
2). In a circular linked list insertion at the beginning and end can be done in constant time order O(1).
3). In a doubly linked list:
   (a) Insertion at the beginning can be done in O(1) time
   (b) Insertion at the end can be done in O(n) time.

Types of Linked Lists:

1). Linear Linked List
2). Doubly " "
3). Circular " "
4). Cursor based " "

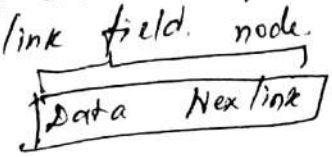
Disadv. of Linked List:

* It does not support random or direct access.
* Each data field should be supported by a link field to point to the next node.

Linked List

1). The linked list is a collection of nodes and each node is having one data field and next link field.

Ex:

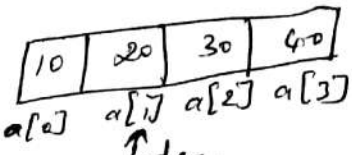


2). Any element can be accessed by sequential access only.
3). Physically the data can be deleted.
4) Insertion & deletion is easy.
5) Memory allocation is dynamic

Array

1). The array is a collection of similar types of data elements. In arrays, the data is always stored at some index of the array.

Ex:



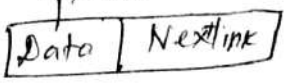
2). Any element can be accessed randomly.
3). Only logical deletion of the data is possible.
4) Insertion & deletion is difficult.
5) Memory allocation is static.

Adv of Circular Doubly Linked List:

* It allows to traverse the list starting at any point.
* It allows quick access to the first and last records.
* It allows to traverse the list in either direction.

Singly Linked List

1). It is a collection of nodes and each node is consisting of one data field and next link field. For example,



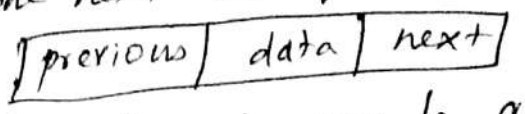
2) The elements can be accessed using next link.

3) No extra field; hence node takes less memory in SLL.

4) Less efficient access to elements

Doubly Linked Lists

1). It is a collection of nodes and each node is consisting of one data field, one previous and one next link field. For example,



2) The elements can be accessed using both previous and next link.

3) One field is required to store previous link hence node takes more memory in DLL.

4) More efficient access to elements.

Applications of Linked List:

1). Linked List can be used to implement Stack, Queue and Tree, etc. It acts as a building block for many other Data Structures.

2). Sparse Matrix

3). Radix Sort

4). Polynomial operations - Add, Subtract, Multiply, Evaluate

5) Set ADT

6) Used to implement Hash Tables.

Circular Linked List Vs Linear Linked List:

* Allows quick access to the first and last records through a single pointer (the address of the last element).
* Their main disadvantage is the complexity of iteration, which has subtle special cases.