down on spot to make free.

Delete at position zero - requires shifting all elements in the list up 1.

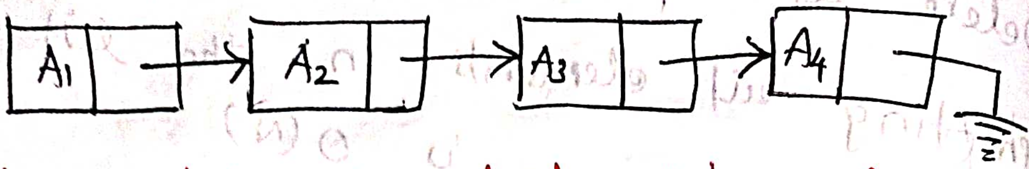
* The worst case is O(N)
* On average half of the list needs to be removed for either operation.
* Find & print list operation to be carried out in linear time.
* Find k^{th} operation takes constant time.
* The running time for insertions and deletions is so slow and the list size must be known in advance. simply arrays are generally not used to implement list.

② Linked List Implementation

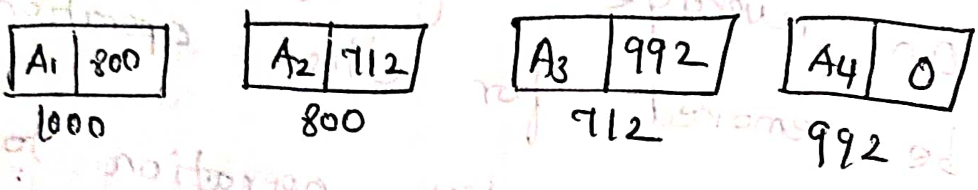
Need:

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire part of the list will need to be moved.

Linked List



Linked list with actual pointer values



The linked list consists of series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to structure containing its successor, called Next pointer.

→ The Next pointer points to NULL.

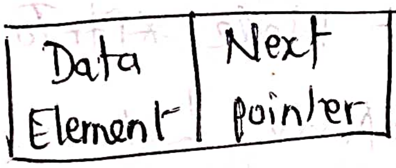
Types of Linked List

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

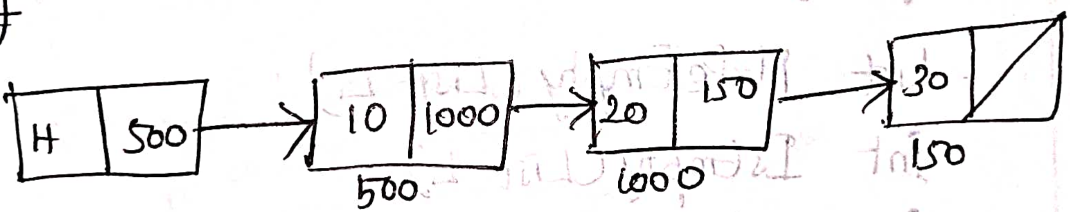
Singly Linked List :-

* Item Navigation is forward only.

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



Eg.



Basic Operations :-

* Insertion - Add an element at the beginning of the list

* Deletion - Delete an element at the beginning of the list

* Display - Displaying the complete list

* Search - Search an element using given key

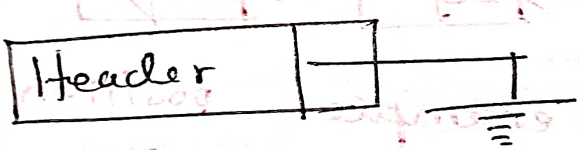
* Delete - Delete an element using given key.

Declaration for Linked List

```
#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);
```

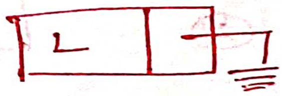
```
Position Advance (Position P);
ElementType Retrieve (Position P);

#end if

struct Node
{
  ElementType Element;
  Position Next;
};
```

Empty list with Header



Function to check whether a linked list is empty

Ex:



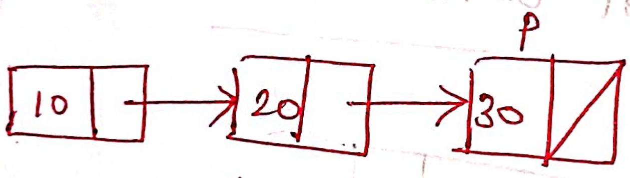

```
int IsEmpty (List L)
{
    return L→Next == NULL;
}
/* Return true if L is empty */
```

Function to test whether current position is the last in a linked list

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

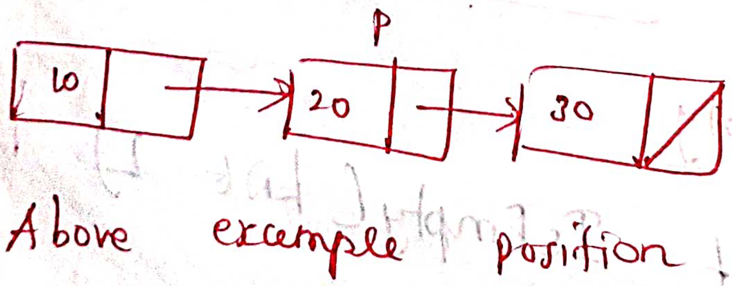
/* Return true if P is the last position in list L */

/* Parameter L is unused in this implementation */

① 

(True)

Above example position 'p' points last position. so it returns true i.e.) 1

② 

(False)

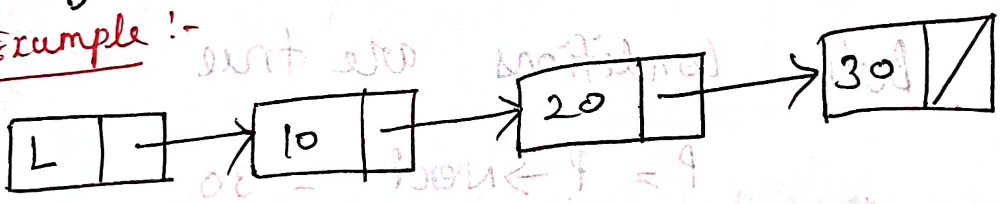
Above example position points the mid of the list. so it's not pointing last posn position. IsLast() Functions getting false.

Find Routine :-

/* 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;
}
```

Example :-

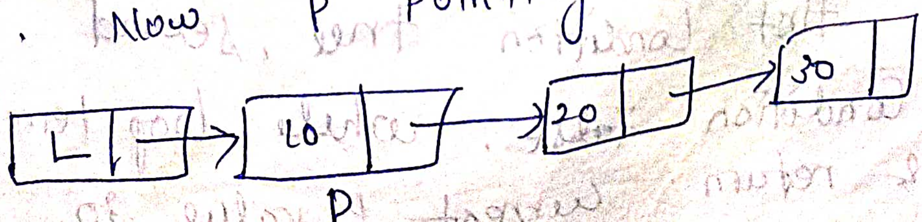


Find Routine X represents which element you want to find, L represents entire list.

Find value 30   X = 30

① P ⇒ L→Next

L→Next pointing the value 10. Now 'P' pointing value 10.



while (P != NULL && P → Element != x)

Position 'p' pointing the value 10

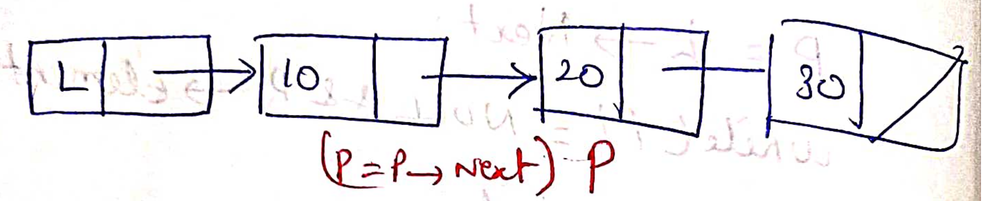
so P != NULL && P → Element != 30
(10)

both conditions are true.

P = P → Next;

P = pointing the value 20

(2)



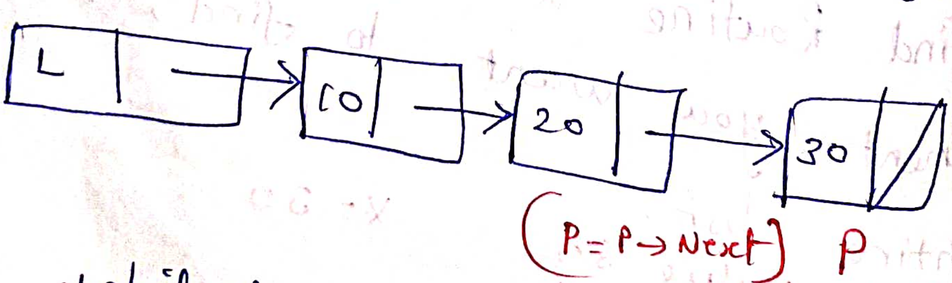
while (P != NULL && P → Element != x)
(True) 20 != 30 (True)

Both conditions are true

P = P → Next = 30

'P' pointing the value 30

(3)



while (P != NULL && P → Element != x)
(True) && 30 != 30 (False)

First condition true, second condition false. while loop terminates & return current P value 30.

Find Previous :-

Find routine is used to find previous element of given key element.

/* If X is not found then Next field of returned */

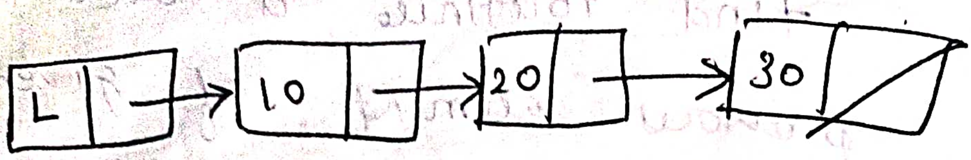
/* position is NULL */

/* Assume 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;
}
```

Above algorithm Find Previous element of 'X'.

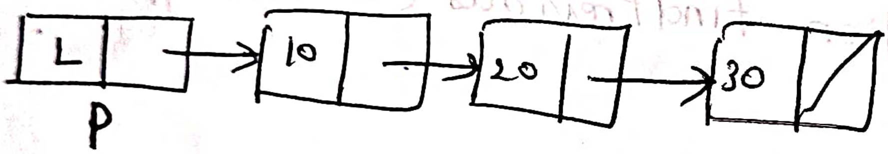
Ex



In the above list find Previous of 20 using FindPrevious routine

X = 20 & Entire list 'L' passed as an argument.

① P = L;



P→Next != NULL, P→Next pointing the value 10, so the condition is true.

P→Next→Element = 10

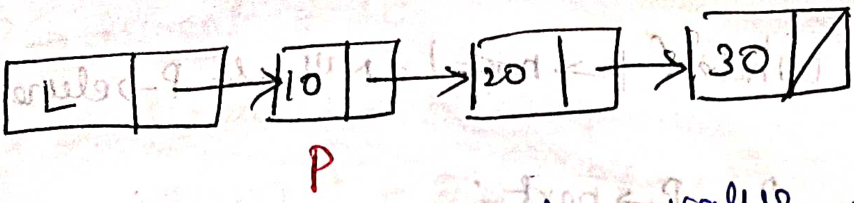
P→Next→Element != X

10 != 20

Both conditions are true. then

P = P→Next

P = 10

(1) 

P → Next pointing the value 20

so P → Next != NULL (true)

P → Next → Element = 20

20 != 20 (false)

⇒ The second condition in while loop getting false. while loop get terminated.

return P value.

→ current 'p' value 10.

⇒ previous of Element 20 returned.

Find Next:-

FindNext() routine find the next element of given key element.

```
Position FindNext(ElementType X, List L)
{
    Position P;
    P = L->Next;
```

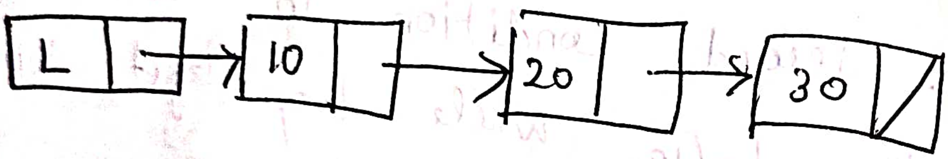
while (P→next != NULL && P→element) != x

P = P→next;

return P→next;

}

Example:-



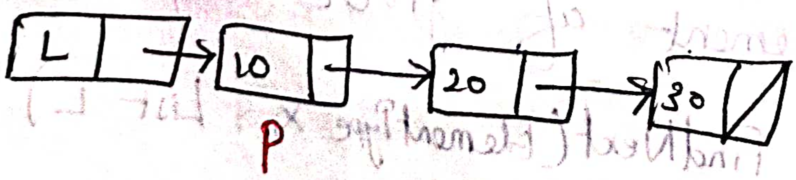
In the above list find next of 20. using FindNext() routine

X=20 & Entire list 'L' passed as an argument.

① P = L→Next

L→Next pointing the value 10

P= 10



'P' pointing value 10, P→Next pointing

P→next != NULL (true)

P→element = 10

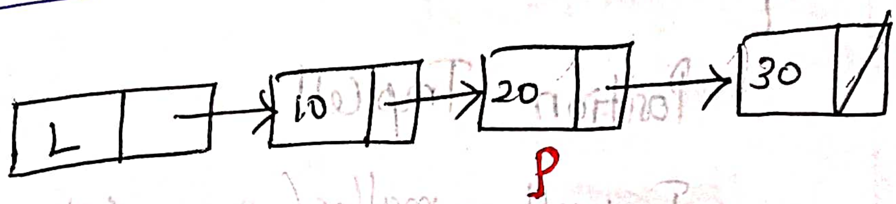
P→element != x

10 != 20 (true)

Both conditions are true [P = P→next]

[P = 20]

(2)



'p' - pointing the value 20

P→next pointing the value 30

P→next != NULL (true)

30 != NULL

p→element = 20

p→element != x

20 != 20 (False)

In while loop second condition getting false. so loop gets terminated and return P→next

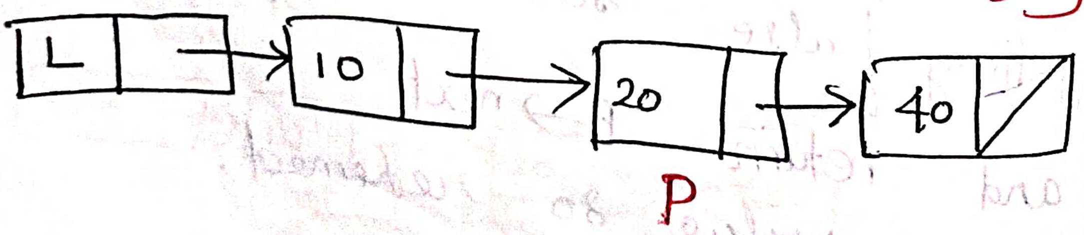
∴ value 30 returned.

Insert Routine:-

Insert an element in the given list.

```
Void Insert(Element-Type 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;
}
```

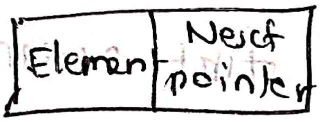
Example: Insert Element 30 [x = 30, P = 20]



Insert an element 30 in above list. currently Position 'p' pointing the value 20

① Tmpcell = malloc (sizeof (struct Node))

In declaration part struct Node contains two members called element & next pointer. Therefore using malloc function, memory dynamically allocated for Element & next pointer, that ~~with~~ can be assigned to <u>Tmpcell</u>

Tmpcell = 

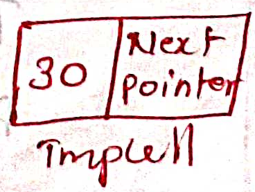
② if (Tmpcell = = NULL)

if memory is not allocated during run time, the error message can be displayed. "out of space"

③ if memory allocated

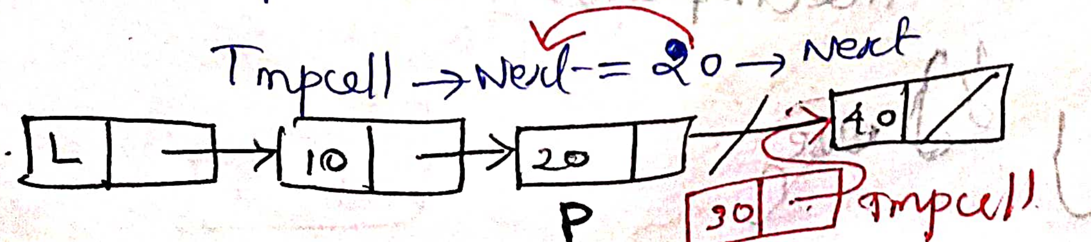
Tmpcell → Element = x;

Tmpcell → Element = 30

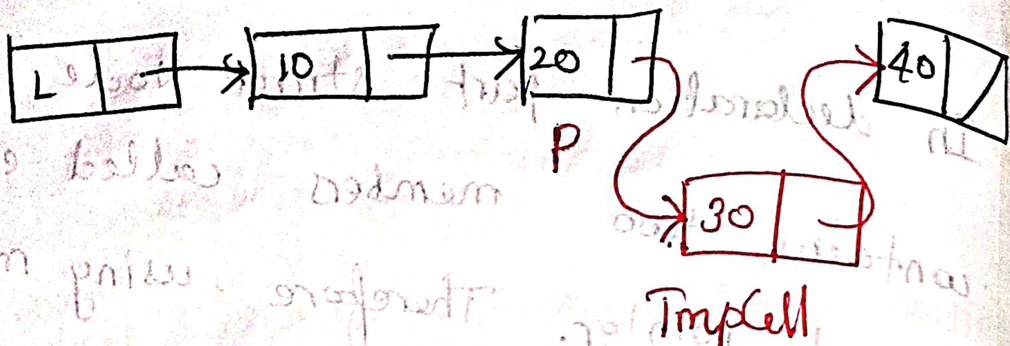


Tmpcell → Next = P → Next

Tmpcell → Next = 20 → Next



P → Next = Tmpcell;



Delete Routine :-

Delete a given element x from the list.

```
/*Delete first occurence of x from a list*/
/*Assume use of a header node */

Void Delete (ElementType X, List L)
{
    Position P, Tmpcell;
    P = Find Previous (X, L);
    if (! IsLast (P, L))
    {
        Tmpcell = P → Next;
        P → Next = Tmpcell → Next;
        free (Tmpcell);
    }} else
       {
```

Tmpcell = P → Next;

P → Next = NULL;

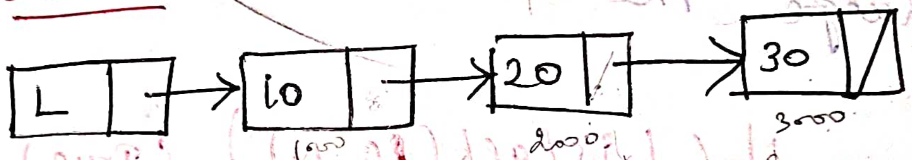
free(Tmpcell);

}

}

Example Delete an Element 20

Case (i) – Element is not a last node



X = 20

Before deleting an element in a list must follow two operations

1. FindPrevious()
2. IsLast()

(1) P = FindPrevious(X, L);

P = FindPrevious(20, L);

In the given list previous of 20 is the 10 ⇒ currently 'p' pointing position 10

P = 10

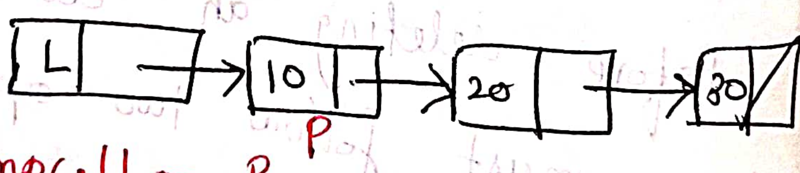
(5) check if the position 'p' is last node (or) not.

(i) if p is last node the next pointer field make it as NULL

(ii) if p is not a last node delete the pointed node and change the pointer position.

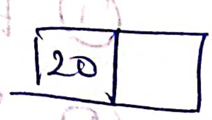
if(! IsLast(10, L)) (True)

10 is not a last node so Condition true.



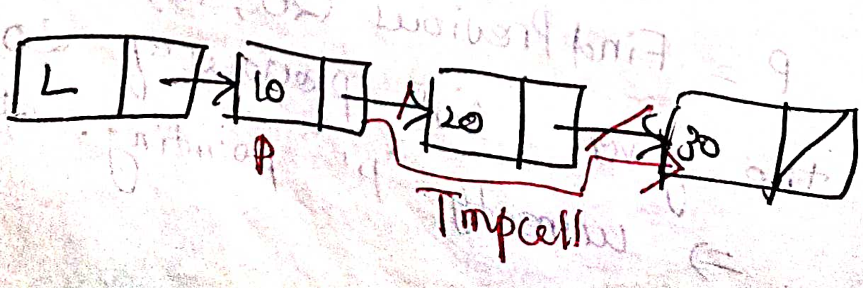
Tmpcell = P → Next

Tmpcell = 20 ⇒

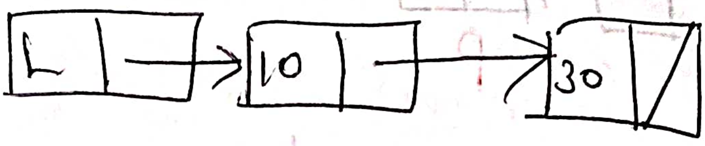


P → Next = Tmpcell → Next;

10 → Next = 20 → Next



Free (tmpCell) – Deallocating the memory allocated for tempCell.

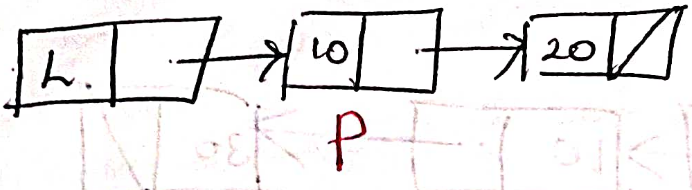


Routine to delete an Entire List

```
Void deleteList (List L)
{
  Position P, temp;
  P = L→next;
  L→next = NULL;
  while (P != NULL)
  {
    Temp = P→next;
    free(P);
    P = temp;
  }
  free(P);
}
```

```
Display (List L)
{
  P = L→next;
  while( P != NULL)
  {
    print ("P");
    P = P→next;
  }
}
```

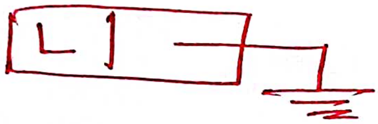
Example



P = L → next ;

P = 10

L → Next = NULL

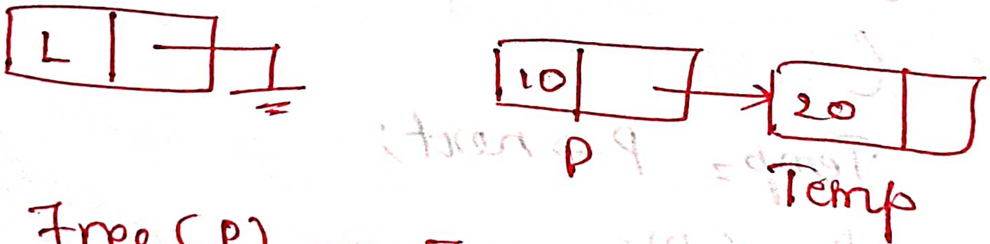


① while (P) = NULL)

while (10 ! = NULL) ✓

Temp = P → next

Temp = 20



Free (P) ⇒ Free (10)

deallocating memory of value 10

② P = temp;

P = 20

While (P ! = NULL) ✓

{

{

Temp = P → next

Temp = NULL (x)

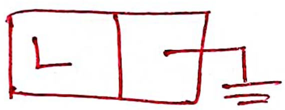
free(P) ⇒ Free(20)

P = temp ⇒ P = NULL

(3) while (P != NULL) x

(P = = NULL)

Loop terminates



Complexity Analysis

→ In Singly linked list insertion, deletion, Isempty, Islast routines takes O(1) time.

→ Find & Find Previous, Find Next routines takes O(N) as worst case and O(N) as average case.

Doubly Linked List :-

Items can be navigated forward and backward way.

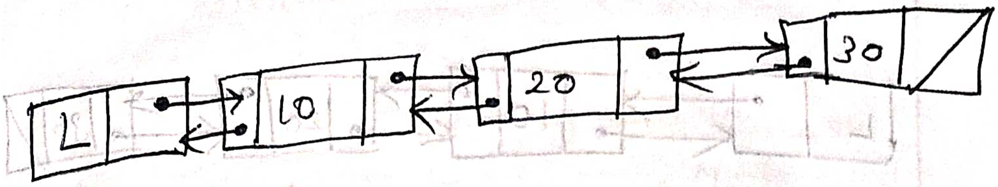
Doubly linked list is a linked list in which each node has three fields namely: data field, forward link (points to successor node), Backward link (points to predecessor node)

Backward Link	Data field	Forward Link

Structure Declaration :-

```
struct node
{
    int element;
    struct node *flink;
    struct node *Blink;
};
```

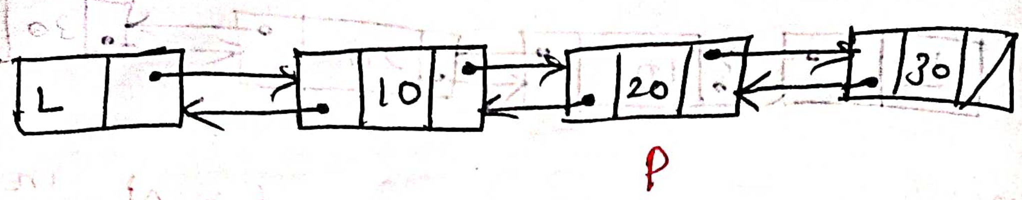
Example :-



Routine to Insert an element in 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;
Newnode = P -> flink;
newnode -> Blink = P;
}
}
```

Example :-



Insert (25)

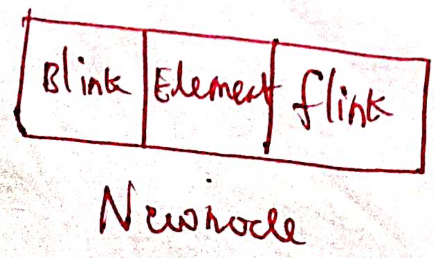
X=25, current position P=20

newnode = malloc(sizeof (struct node));

In declaration part struct node contains three members element, forward link, Backward link. Using malloc function memory dynamically allocated for above structure members that representation assigned to newnode.

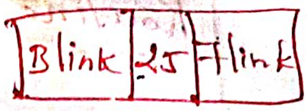
if(newnode != NULL)

if memory allocated for newnode the above condition gets true. then



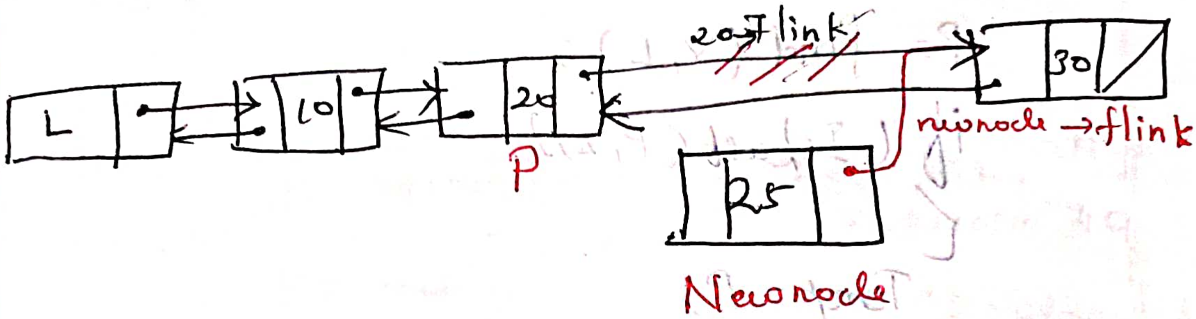
① newnode → element = x;

newnode → element = 25

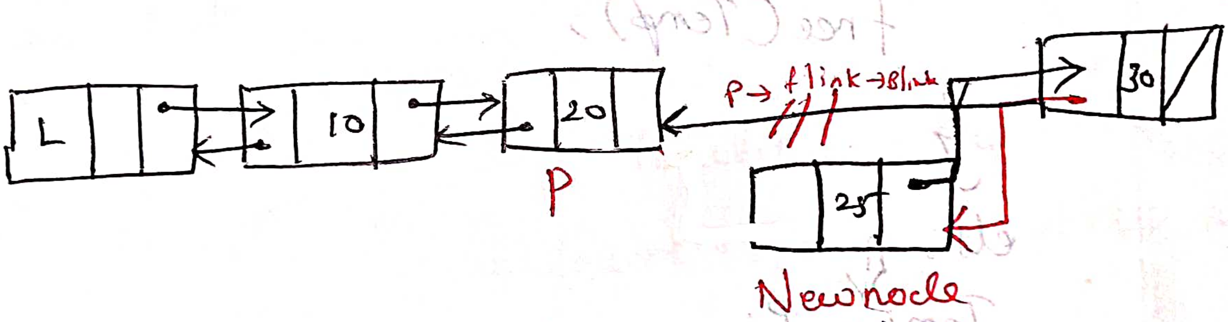


② newnode → Flink = P → flink

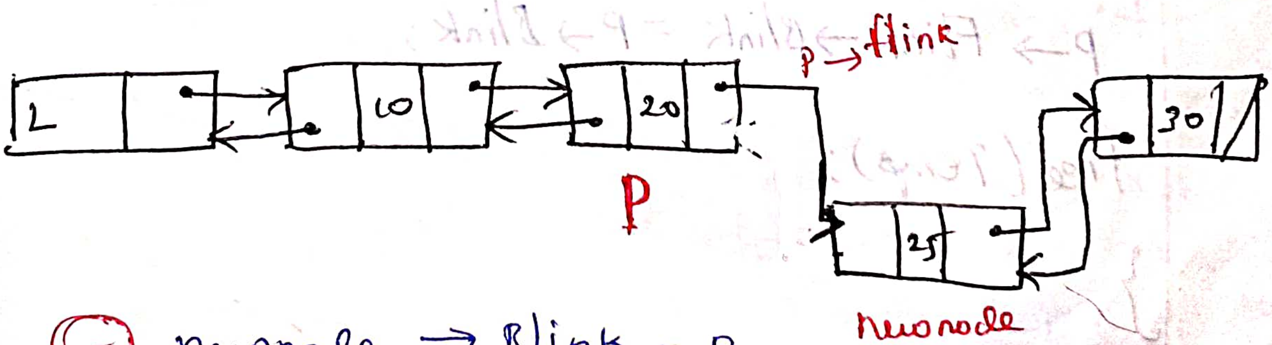
newnode → Flink = 20 → flink



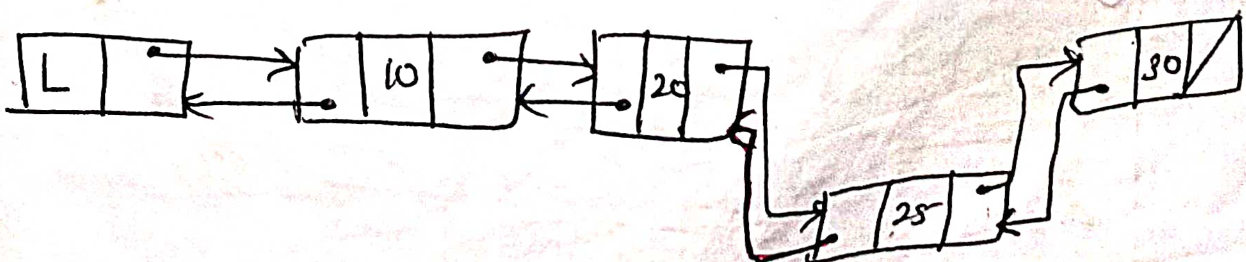
③ P → Flink → Blink = newnode;



④ newnode = P→flink;



⑤ newnode → Blink = P;

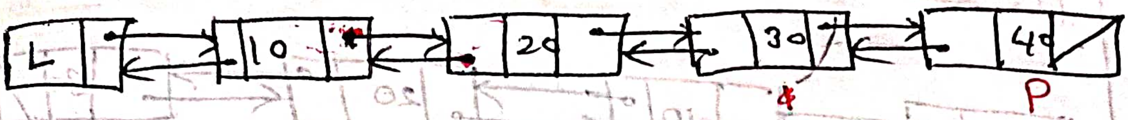


Routine to delete an Element

```
Void delete (int x, List L)
{
    Position P, Temp;
    P= find (x, L);
    if (IsLast(P, L))
    {
        Temp = P;
        P->Blink ->Flink = NULL;
        free(Temp);
    }
    else {
        Temp = P;
        P->Blink ->Flink = P->flink;
        P-> Flink->Blink = P->Blink;
        free (Temp);
    }
}
```

Example :-

Case 1 :- Delete element 40



X = 40 & List L passed as an argument.

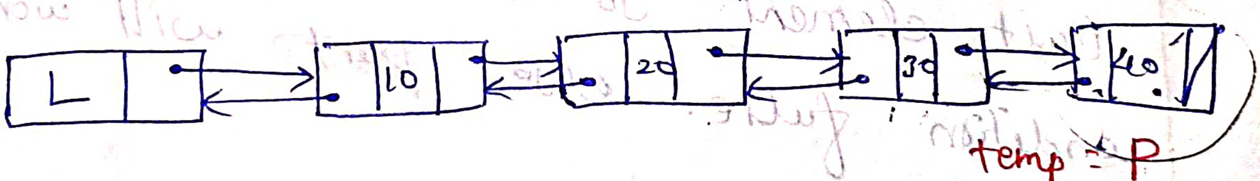
P = find (X, L) ; (40, L) find operation returns 40

P = 40 ;

⇒ check whether 40 is last element or not

if (IsLast (40, L))

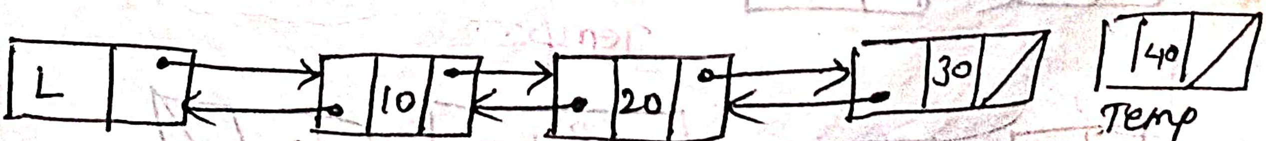
↳ Function returns true because 40 is last element



temp = 40

P → Blink → Flink = NULL

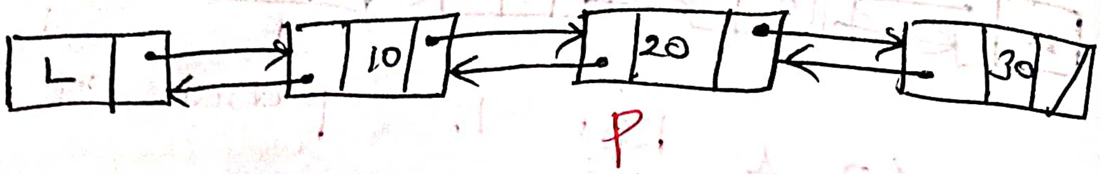
40 → Blink → Flink = NULL
30



free (temp)

Memory deallocated for the variable temp.

Case 2: Delete element 20 in the following list



[P = Find(X, L)] Find operation returns the value 20

P = 20;

⇒ Check whether the element P is last or not

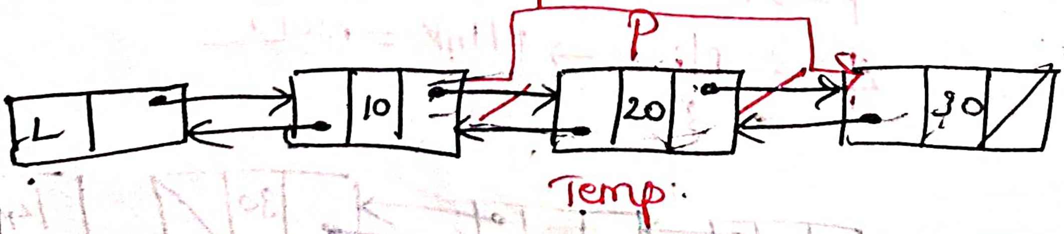
if(IsLast(P, L))

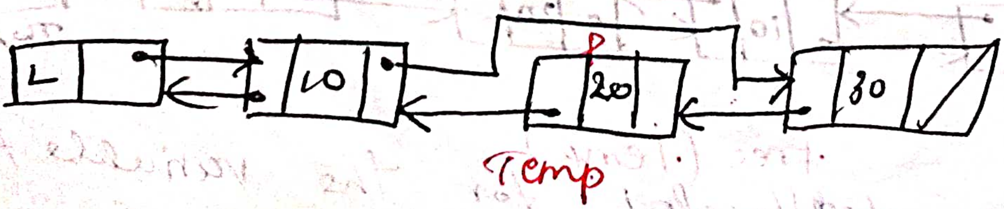
if(IsLast(20, L)) /* 20 is not a last element so it returns NULL condition false. else part will work

Temp = P;

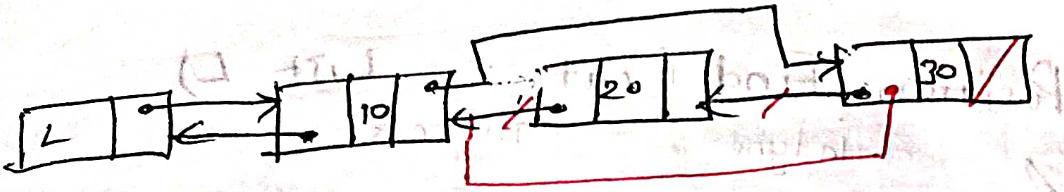
Temp = 20;

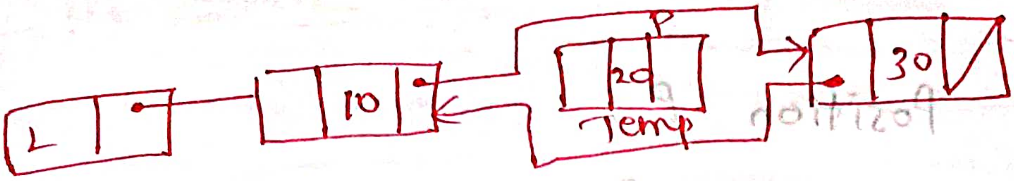
P → Blink → Flink = P → Flink



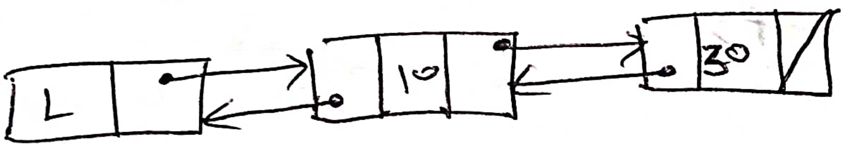


P → Flink → Blink = P → Blink





Free (Temp) ⇒ memory deallocated for the variable temp.



Routine to check whether the current position is last

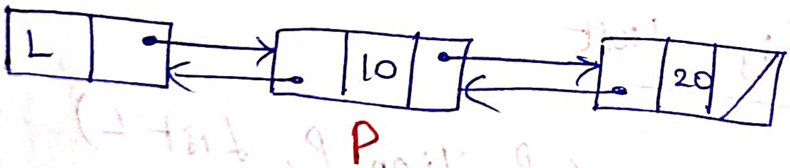
```
int IsLast ( Position P, List L)
{
  if (P → flink = = NULL)
  return (1)
}
```

Find Routine

```
Position Find (int x, List L)
```
datatype   arguments

```
{
    Position P;
    P = L→flink;
    while (P != NULL && P→element != x)
    P = P→flink;
    return P;
}
```

Example:- Find 20   x = 20



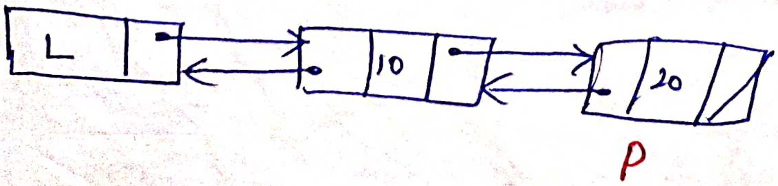
P = L→flink = 10;

while (P != NULL && P→element != 20)   (P→element = 10)

✓ ✓

Both conditions are true then

P = P→flink = 20



while (P! = NULL && P→element ! = x)

20 ! = 20

one condition true, the second condition false then while loop gets terminate then return P = 20.

Advantage of Doubly linked list

* Deletion operation easier.
* Finding successor and pre-decessor node is easier.

Disadvantage :-

* More memory is required since it has two pointers.

Circular Linked List

* In circular linked list the pointer of last node points to the first node.
* Circular linked list can be implemented as singly or doubly linked list with (or) without header.