Linked Lists

OMKARAM.

Dt: 29.04.2020

→ List is a collection of similar type of elements. There are 2 ways of maintaining a list in memory.

→ The 1st way is to store the elements of the list in an array. But arrays have some disadvantages.

→ The 2nd way of maintaining a list in memory is through linked list.

There are two types of lists:

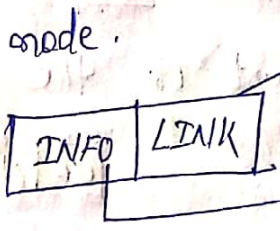
(1) Single Linked list

(2) Doubly Linked list

Single Linked list:-

→ A single linked list is made up of nodes, where each node has two parts, the 1st one is the INFO part that contains the actual data of the list.

→ The 2nd one is the link part, that points to the next node of the list or it contains the address of the next node.



→ The beginning of the list is marked by a special pointer called start.

→ This pointer points to the 1st node of the list.

→ The link part of each node points to the next node in the list.

→ But the link part of last node has no next node to points, so it is made NULL.

→ So, if we reach a node whose link part has NULL value, then we know that, we are the end of the list.

Example: A list of 4 integers: 33, 44, 55, 66.

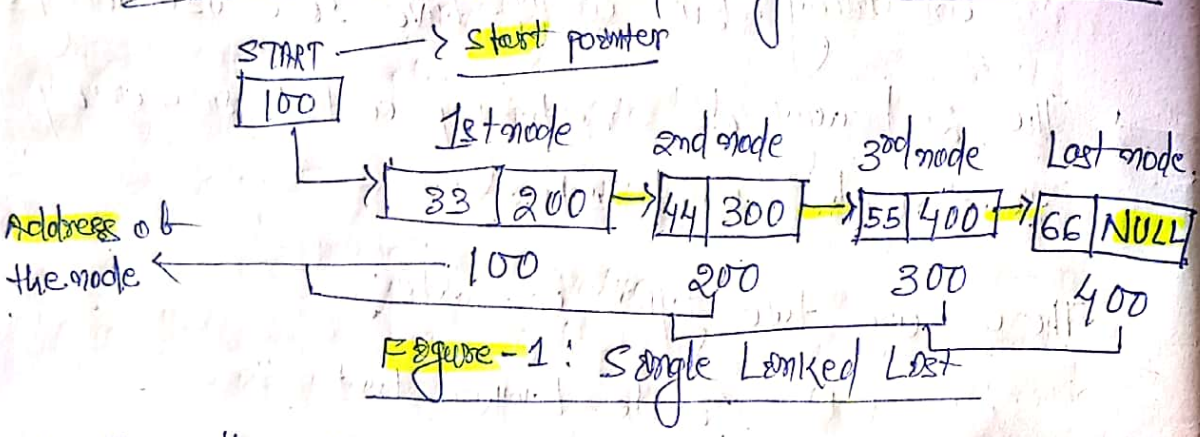


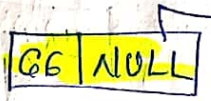
Figure-1: Single Linked List

→ From the figure-1, we have found out the following:-

(i) The address of the 1st node is contained in the pointer START, i.e 100 START.

(ii) The INFO part of each node contains the integer value, and link part contains the address of the next node.

(iii) The link part of the last node is NULL



* All operation on Linked list could start from the pointer START.
* If the pointer START contains NULL value, then list is empty.

```
struct node *start
start = NULL;
```

→ The following operations on a single linked list :-

(i) Traversing a Single Linked list.
(ii) Insertion of an element.
(iii) Deletion of an element.
(iv) Searching an element

} Basic operations of single linked list.

Traversing a Single Linked list (visiting each node)

→ If we visit the 1st node to last node of the list, it is called traversing a list.

→ Let 'P' be the pointer, which will point to the node that is currently being visited.

→ Initially, we have to visit the 1st node, so, P is assigned the value of start.

So, `P = start`

→ Now P, points to the 1st node of linked list. We can access the info part of 1st node by writing : `P -> info`

→ Now, we have to shift the pointer P, forward, so, that it points to the next node. So, the address of the next node is assigned to P.

`P = P -> Link`

→ Now P has address of the next node. So, we can visit each node of linked list, through this assignment until P has NULL value.

→ So, we reach the last node and the linked list is traversed.

Algorithm:

Step-1: START → start pointer.

Step-2: if (start == NULL)

Step-3: Printf ("List is empty")

Exit.

Step-4: else, p = start;

Step-5: if (p != NULL)

Step-6: Printf ("value")

p → info

Step-7: p = p → link → address of next node.

Step-8: Check, if (p == NULL), then visit the last node of the linked list.

Step-9: otherwise Repeat step-4 to 7.

Step-10: STOP

Example:-

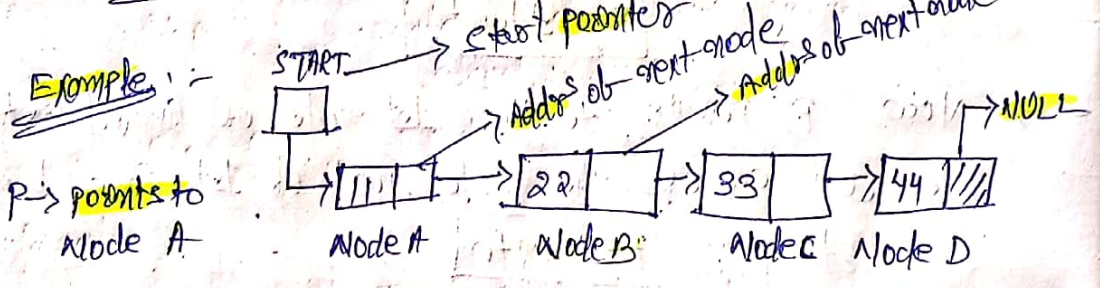


Figure-2

→ In figure-2, Node-A is 1st node, so start points to it. Node-D is the last node, so its link is NULL.

→ Initially p points to node-A, P→info = 11, and p→link points to node B.

→ After execute P = P→link, P points to node-B, so, p→info = 22, and p→link points to the node C.

→ Continue in the same process, we reach last node-D, where P→link = NULL.

Insertion on a Single Linked List.

There can be 3 cases, while inserting a node in a linked list.

(i) Insertion at the beginning.

(ii) " at the end.

(iii) " in between the list nodes (Any specific posn).

Insertion at the beginning of the list.

→ To insert a node, initially we could dynamically allocate space for that node using malloc().

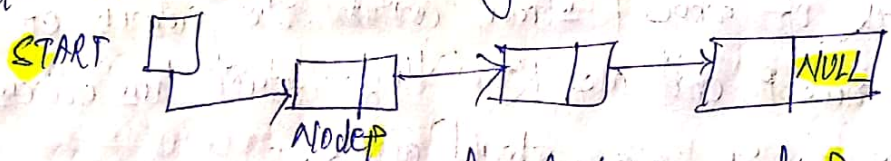
→ Let temp is a pointer that points to this dynamically allocated node.

→ In the info part of the node, we can put the data value.

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

→ The link part of the node contains garbage value. We could assign address to it in the different cases.

Let we assume the given list is as :-



Let the first node of list is node p. So, the new node T should be inserted before node p.

Before Insertion: Node p is the first node.
START points to node p.

After insertion

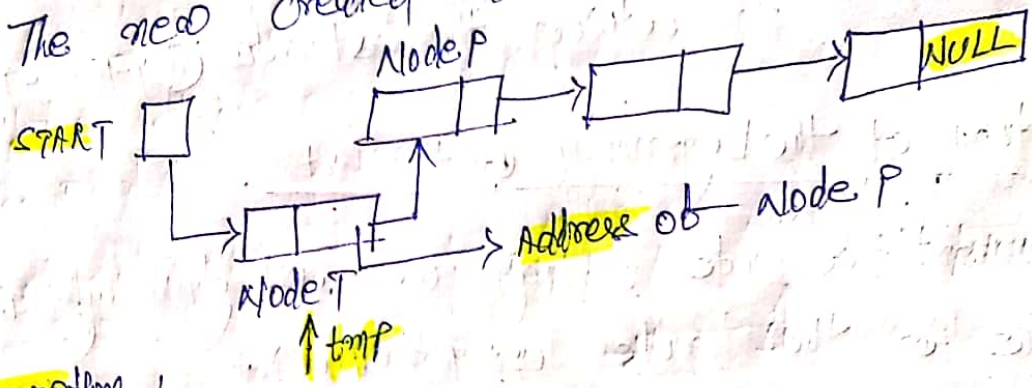
Node T is the first node

START points to node T.

Link of node T points to node P.

So, now node P is the second node of the list.

→ The new created list is :-



Algorithm :

step-1 : START

step-2 : Create a dynamically pointer "temp"

step-3 : Assign the data value to info

temp->info = data;

step-4 : Assign the start address to temp

temp->link = start;

step-5 : update the start value :

start = temp;

step-6 : Exit.

Explanation :-

(i) Link of node T should contain the address of node P, and we know that start has address of node P. So we should write :-

temp->link = start;

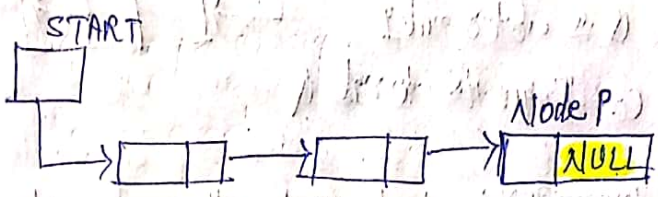
After this statement, link of node T would point to node P.

(ii) We want to make node T the first node, hence we should update start, so that now it points to node T. start = temp;

Insertion at the end of the list:

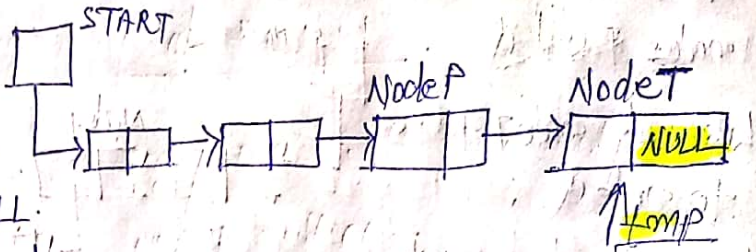
In this case, we have to insert a new node T at the end of the list. Let the last node of list is node P. So, node T should be inserted after node P.

Before Insertion:



Node P is the last node. So Link of node P is NULL.

After insertion:



Node T is last node.
Link of node T is NULL.
Node P is second last node.
Link of P points to node T.

Algorithm: Step-1: START
Step-2: Create a dynamically pointer "temp".
Step-3: Traverse the list till reach end node.
Step-4: P = START
Step-5: P -> Link = temp;
Step-6: temp -> Link = NULL;
Step-7: - End

Explanation: (i) Let we have a pointer P, pointing to the node P. So, `P -> Link = temp;`

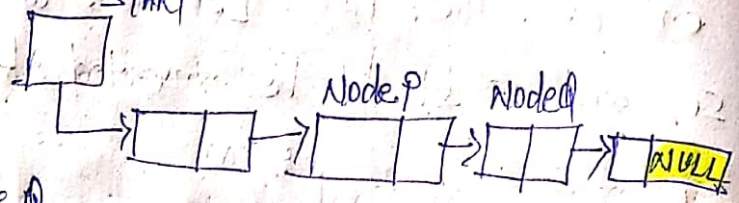
(ii) Then we assign the NULL value, to the link part of node T, which is the last node.
`temp -> link = NULL;`

(iii) Before doing this, we must traverse the linked list.

Insertion between the last node (At specific position)

Let P and Q be the two nodes on the list.

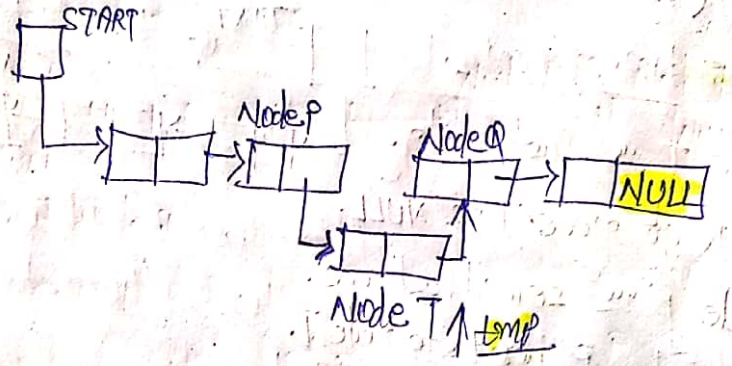
Before Insertion:



Node Q is after node P.
Link of P points to node Q

After Insertion: Let T be the node, which could insert between Node T is between P and Q.

nodes P and Q



Link of node T points to node Q.

Link of node P points to node T.

Algorithm: **Step-1:** START

Step-2:- Let P & Q be the two pointer, pointing node P and Q.

Step-3:- Let create a dynamically pointer "temp".

Step-4:- We can assign the address of Q into the new node, which is:

```
temp->link = p->link;
```

Step-5:- Also we can assign the address of new node T to its previous node.

```
p->link = temp;
```

Step-6:- Exit.

Explanation:- (i) Before insertion of node Q is on Address p->link, so, we can write:-

```
temp->link = p->link;
```

(ii) Also we can assign the address of node T to its previous node P so, `p->link = temp;`

Deletion in a Single Linked list.

→ To delete a node from the linked list, the pointers are rearranged. So, that this node is logically removed from the list.

→ To remove the node physically and return the memory occupied by it to the pool of available memory.

→ We could use the function free().

→ We could take a pointer variable 'tmp', which could point to the node being deleted.

→ To physically remove node T from the memory, we use free(tmp).

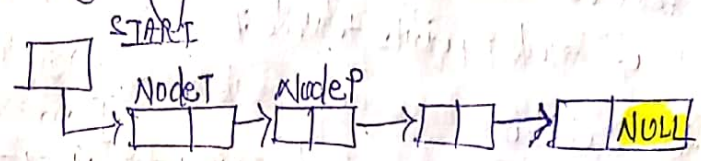
Deletion at the beginning node:

Before Deletion

Node T is the first node

start points to node T.

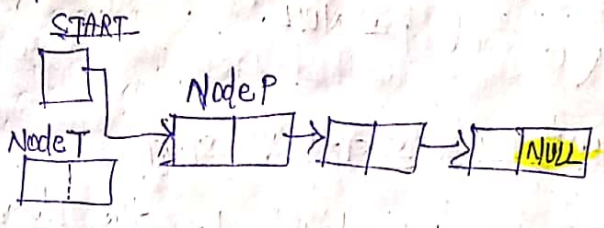
Link of node T points to node P



After Deletion

Node P is the first node

start points to node P.



Algorithm: Step-1 :- START

Step-2 :- Let "tmp" could be the dynamically pointer.

Step-3 :- tmp = start (Assign the start address to tmp)

Step-4 :- start = start->link (Assign the 1st node address to start)

Step-5 :- free(tmp)

Step-6 :- EXIT.

Explanation :- (i) First node is to be deleted, tmp could be assigned the address of first node. temp=start;

(ii) So, now tmp points to the first node, which has to be deleted. Since start points to the 1st node. So start->link could point to the 2nd node.

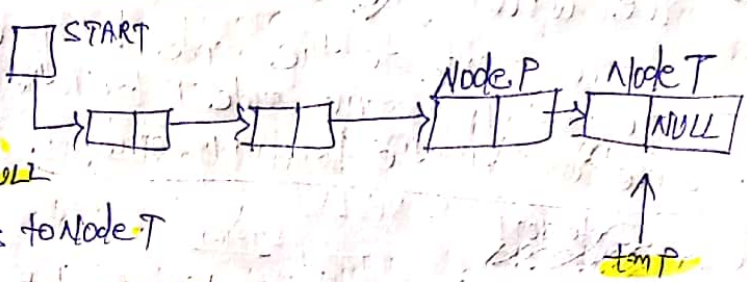
(iii) After deletion of first node, the second node (Node P) become the first one, so start should be assigned the address of node P, as:-

```
start = start->link;
```

(iv) After this statement, start could point to node P, so, now it is the first node of linked list.

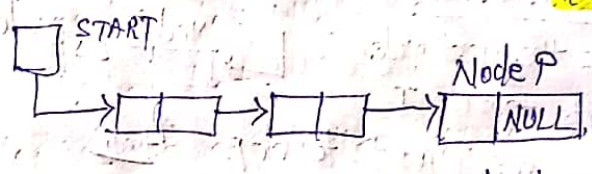
Deletion at the end of the list:

Before Deletion



Node T is the last node
So, link of node T is NULL
Link of node P points to Node T

After Deletion



Node P is the last node
Link of node P is NULL.

Algorithm :- Step-1 :- START

Step-2 :- Let "tmp" be the dynamically pointer.

Step-3 :- 'tmp' points to the node to be deleted (Node T here)

```
tmp = p->link;
```

Step-4 :- So we assign the address of last node (NULL) to its previous node.

```
p->link = tmp->link;
```

Step-5 :- free(tmp), so, memory space becomes free.

Step-6 :- Exit.

Explanation :- (i) If p is a pointer to node P, then node T can be deleted by writing this :-

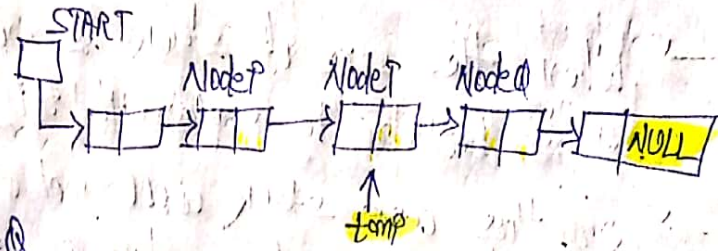
```
p->link = NULL;
```

(ii) Since link of node T is NULL, on the previous st. we can write

```
p->link = tmp->link;
```

Deletion in between the list nodes:

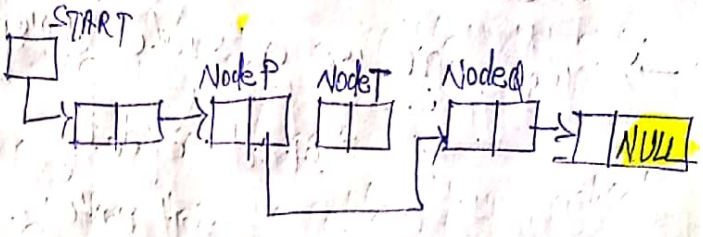
Before Deletion:



Link of P points to nodeT

Link of T points to nodeQ

After Deletion:



Node Q is after node P

Link of P points to nodeQ

Algorithm: **Step-1**: START

Step-2: Create a dynamically pointer "temp".

Step-3: "temp" points the node to be deleted (NodeT here)

Step-4: p = start;

Step-5: Now assign the address of p->link to temp.

```
temp = p->link;
```

Step-6: - Now, we can assign the addres of node to its previous node

```
p->link = temp->link;
```

Step-7: - free(temp). So, memory space becomes free.

Step8: - Exit -

Explanation: - (i) Suppose node T is to be deleted and the pointer "temp" points to it. We have pointer P & Q which points to node P & Q.

(ii) To delete nodeT, we could set link the predecessor of node T (NodeP) and successor of T (NodeQ).

```
p->link = temp->link;
```

(iii) After this, we can free memory space:

```
free(temp);
```

Linked List Implementation of Stack:

→ When the size of stack is not known in advance, it is better to implement it a linked list.

→ In this case stack could not overflow until there is space available for dynamic memory allocation.

→ We could use a single linked list to implement it.

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

→ We could take the beginning of linked list as the top of the stack.

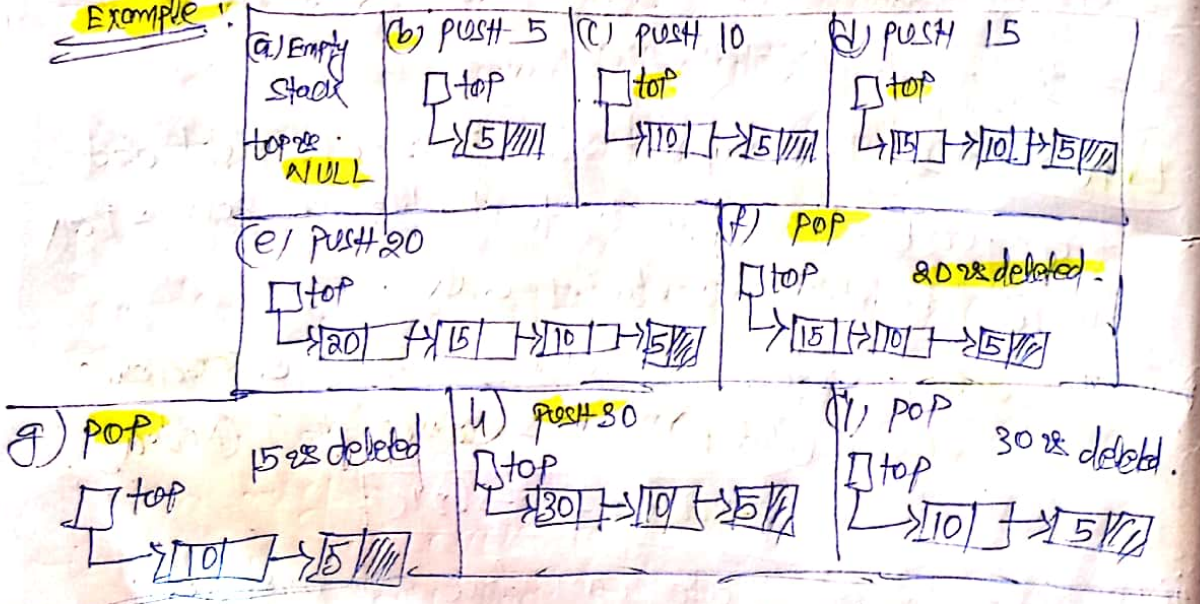
→ We could take a pointer top, that points to the first node of linked list.

→ This pointer top is same as the pointer start, that we had used in single linked list.

→ For PUSH operation, a node could be inserted in the beginning of list.

→ For POP operation, first node of the list will be deleted.

Example:



→ For pushing an element on stack, we follow the procedure of insertion on beginning of the linked list.

→ The stack will overflow, only when there is no space left for dynamic memory allocation.

→ For pop operation, we delete the first element of linked list.

→ The underflow condition will arise, when linked list is empty.

Linked List Implementation of Queue

→ Queue can also be implemented through linked list.

→ The structure of a node will be:-

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

→ Let front represent the beginning of linked list.
rear " " " end " " .

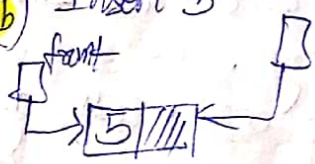
→ To insert an item on our queue, we will add a node at the end of the list.

→ To delete an item from queue, we will delete a node from the beginning of the list.

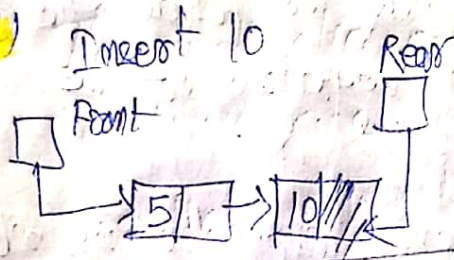
→ Let two pointers front and rear to the node of list.

Example (a) Empty Queue
Front is NULL
Rear is NULL

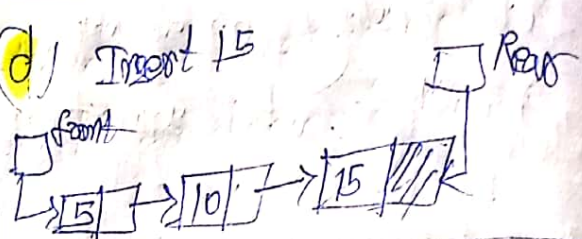
(b) Insert 5



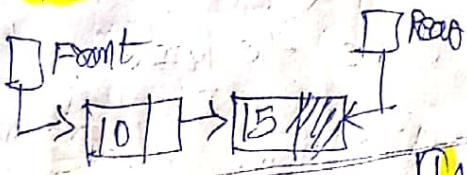
(c) Insert 10



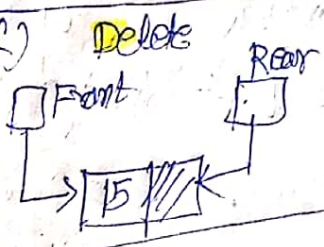
(d) Insert 15



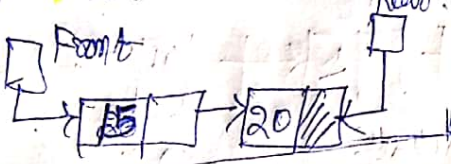
(e) Delete



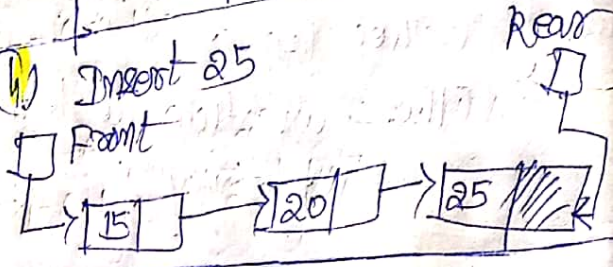
(f) Delete



(g) Insert 20



(h) Insert 25



Circular Linked list :

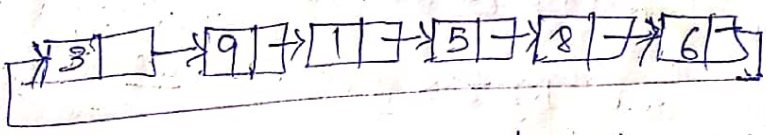
→ In a single linked list, for accessing any node, we start traversing from first node of the list.

→ Suppose we are at any node in the middle of the list, then it is not possible to access nodes, that precede the given node.

→ To overcome this problem, we may change the structure of linked list slightly.

→ This structure of the list is called circular linked list.

Example



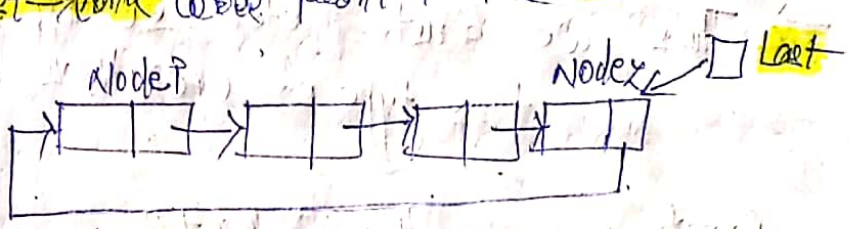
→ Each node has successor and all the nodes form a ring.

→ Now, we can access any node of the linked list, without going back and starting traversal again from the 1st node.

→ We take an external pointer, that points to the last node of the list.

→ If we have a pointer last pointing to the last node, then, last→link, could point to the next node.

Exa



→ In this figure, the pointer last points to node2 and last→link points to node1.

Polynomial arithmetic with Linked list:

→ To represent a polynomial expression, through linked list is an useful application.

→ Let us take a single variable polynomial expression.

$$5x^4 + x^3 - 6x + 2$$

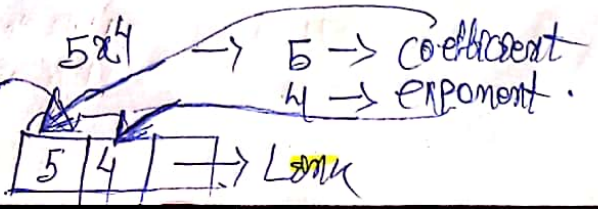
→ In each term, we have a coefficient and an exponent.

→ The whole polynomial can be represented through linked list, where each node could represent a term of expression.

```
struct node {
    float coefficient;
    int exponent;
    struct node *link;
};
```

→ Hence the Info part of the node contains coefficient and exponent, and the link part is same as before used to point to the next node of the list.

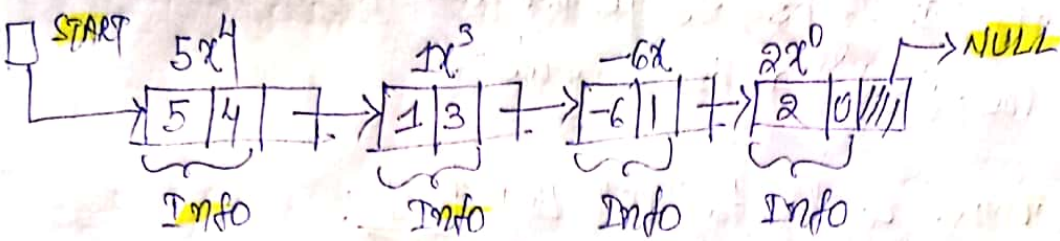
Exa



→ The arithmetic operations are easy, easier, if the terms are arranged in descending order of their exponents.

→ Thus we must prefer sorted linked list for this type of operations.

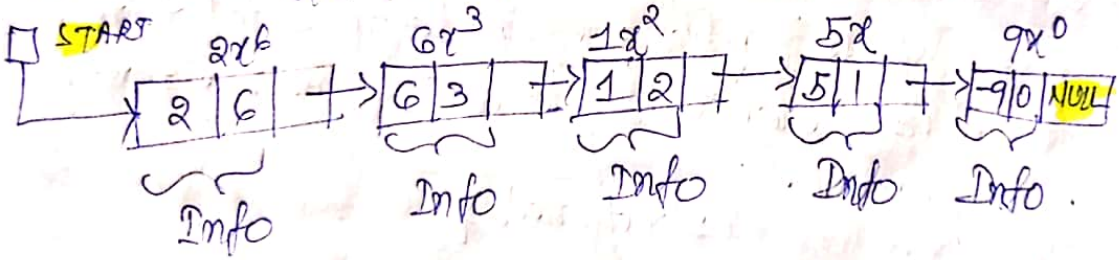
Example :- $5x^4 + x^3 - 6x + 2$



Example :- Arrange the polynomial Expression through single Linked list.

$5x + 6x^3 + x^2 - 9 + 2x^6$

Ans :- $2x^6 + 6x^3 + x^2 + 5x - 9$



→ An empty list could represent zero polynomial.

Eg $3x^3 - 4x^2 + 2x - 9$

Doubly Linked list:

→ In single linked list, we move in one direction, because each node contains the address of next node only.

→ When, we are in the middle of the list, we are unable to access its previous node. So, again we start traversing operations.

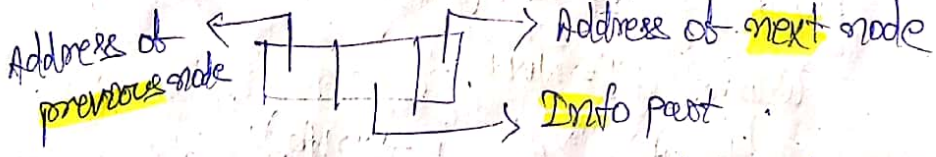
→ To overcome this problem of single linked list, we can use another datastructure called doubly linked list or two way list.

→ In this concept, each node has two pointers, one pointing to the next node and other points to the previous node.

→ The structure of doubly linked list:-

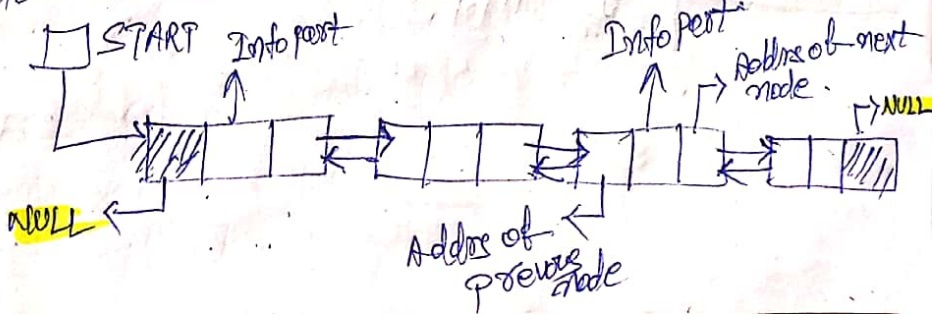
```
struct node {
    struct node *prev;
    int info;
    struct node *next;
};
```

→ Each node consists of 3 parts:-



→ The next pointer of last node and prev pointer of first node are NULL.

Example



⇒ The basic all operations are same as single linked list. But we have to do a little extra coding, because of one more extra pointer, that has to be updated each time.

→ The basic operations of Doubly linked list are:-

(i) Traversing a doubly linked list.
(ii) Insertion in a doubly linked list.
(iii) Deletion from doubly linked list.

Traversing a doubly linked list

→ The function for traversal of doubly linked list is similar to that of single linked list:

```
void display (struct node *start) {
    struct node *p;
    if (start == NULL) {
        printf ("List is empty");
        return;
    }
    p = start;
    printf ("List is :\n");
    while (p1 = NULL) {
        printf ("%d", p->info);
        p = p->next;
    }
    printf ("\n");
}
```

Insertion in a doubly linked list:-

→ Basically there are 3 cases of insertion in a doubly linked list like single linked list.

(a) Insertion at the beginning of the list.

(b) " " " end of the list.

(c) " in " between the nodes.

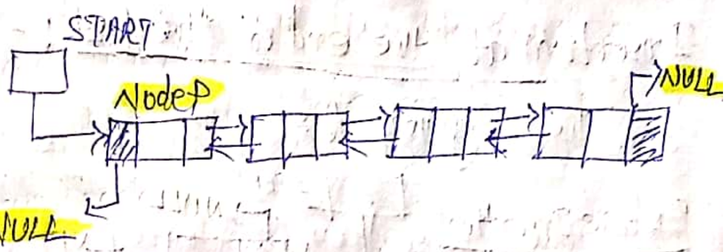
Insertion at the beginning of the list:-

The given list is:-

Before Insertion

Node P is first node

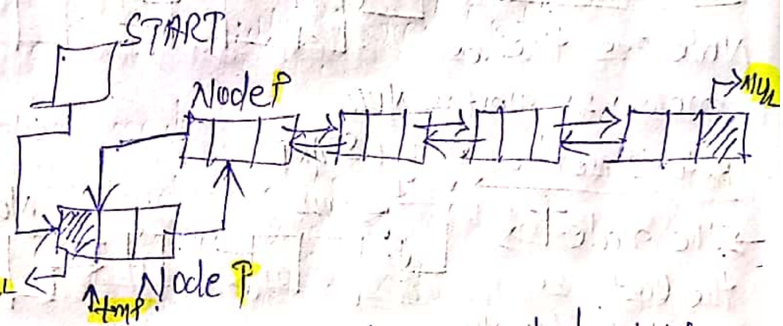
So, prev is NULL



After Insertion:

NodeT is first node

So, prev is NULL.



→ Let node T becomes the 1st node, inserted at the beginning of the list.

Algorithm: Step-1: START.

Step-2: Let 'temp' be the dynamically create pointer.

Step-3: temp → prev = NULL (1st node of the list).

Step-4: we assign the address of next insert ~~part of 2nd node point to 1st node~~

temp → next = start;

Step-5: start → prev = temp; (previous point to 1st node)

Step-6: start = temp;

Step-7: Exit.

Scanned with CamScanner

→ In this example, node T become the 1st node, so, its prev should be NULL.

```
tmp -> prev = NULL;
```

→ The next part of node T should point to node P, and address of node P is in start, so we should write:

```
tmp -> next = start;
```

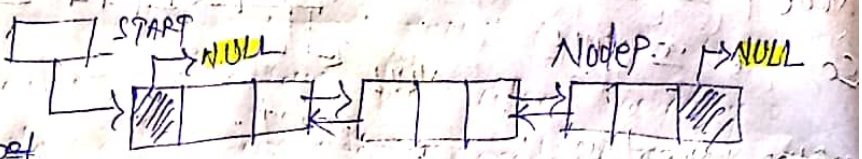
→ Now node T becomes the 1st node, so start should point to it.

```
start = tmp;
```

Insertion at the end of the list:-

The given list is:-

Before Insertion

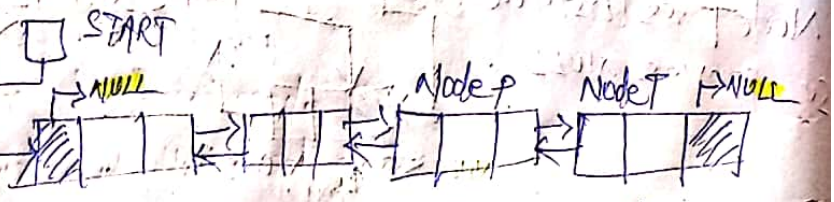


NodeP is the last node, so, next is NULL

After Insertion:

→ The node T is the last, so, next part is NULL.

→ prev part point to node P



Algorithm: Step-1: START.

Step-2:- Let "tmp" be the dynamic pointer.

Step-3:- p->next = tmp; (next part of node P points to tmp)

Step-4:- tmp->next = NULL; (Last node next part is NULL)

Step-5:- tmp->prev = P;

Step-6:- Exit.

→ nodeT becomes the last node, so next should be NULL

```
tmp->next = NULL;
```

→ next part of node P should point to node T.

```
p->next = tmp;
```

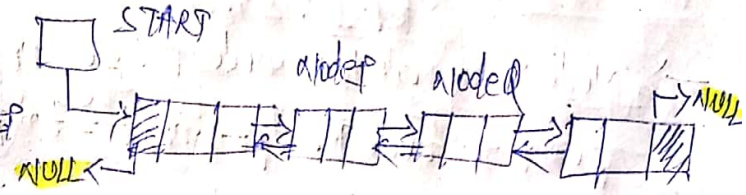
→ prev part of node T should point to node P.

```
tmp->prev = p;
```

Insertion in between the nodes:

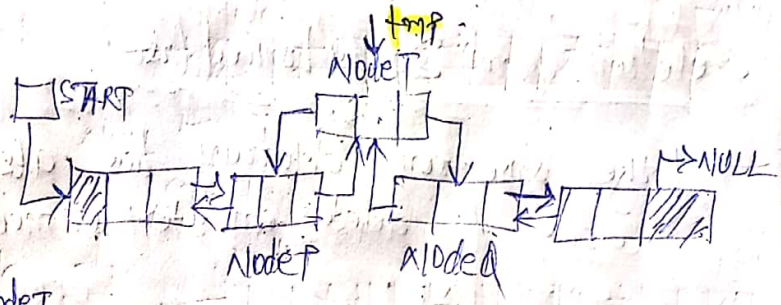
The given list is:

Before Insertion:



P & Q points to nodeP and nodeQ.

After Insertion:



nodeT is the between nodeP & nodeQ.

nodeP is prev of nodeT

nodeQ is next of nodeT.

Algorithm: Step-1: START

Step-2:- Let 'tmp' be the dynamic pointer.

Step-3:- nodeT is tmp->prev = p;

Step-4:- tmp->next = q; (or) tmp->next = p->next

Step-5:- q->prev = tmp; (or) p->next->prev = tmp;

Step-6:- p->next = tmp;

Step-7:- Exit

⇒ In this example, node T is between node P & Q.

⇒ node P is before node T, so prev of node T should point to node P.

tmp->prev = p;

⇒ Node Q is after node T, so next of node T point to node Q. tmp->next = q;

⇒ Node T is before node Q, so, prev of Q point to node T. q->prev = tmp;

⇒ Node T is after node P, so, next of node P point to node T. p->next = tmp;

Deletion from Doubly Linked List.

⇒ Like insertion, deletion has also basically 3 cases to delete an item or node from the given list.

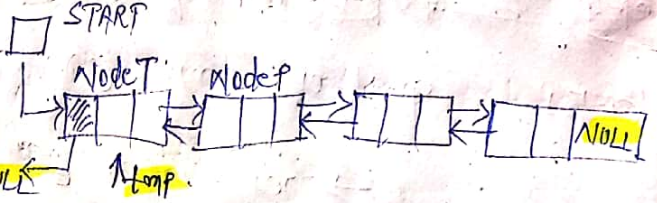
(a) Deletion of first node.
(b) Deletion at the end node.
(c) Deletion on between the nodes.

Deletion of first node from the linked list.

Before Deletion:

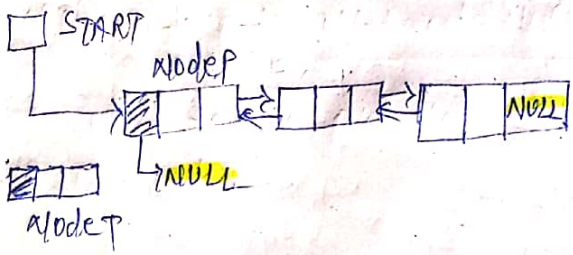
Node T is the first node.
prev is NULL.
next point to node P

Given List is :-



After Deletion:

Node P is the first node
prev is null.
next point to next node.



Algorithm : START

step-2 : Let "temp" be the dynamic pointer.

step-3 : temp will be assigned the address of first node

temp = start;

step-4 : start = start->next; (pointer to next node)

step-5 : start->prev = NULL; (1st node of list)

step-6 : free(temp) (free the occupied memory of deleted node)

step-7 : Exit

Explanation :

→ In this example, node 1 is the 1st node before deletion. so, we assign the start to temp.

temp = start;

→ start will be updated, so that now it points to node 2 (1st node after deletion)

start = start->next;

→ Node 2 is the first node, so its prev part should be NULL.

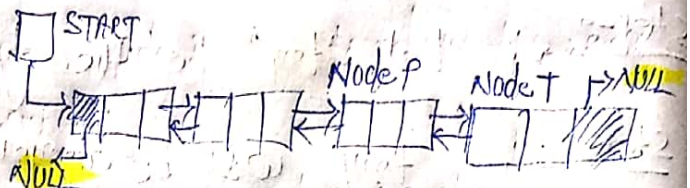
start->prev = NULL;

→ free() is used to delete the node physically from the list.

Delete the end node from the list:

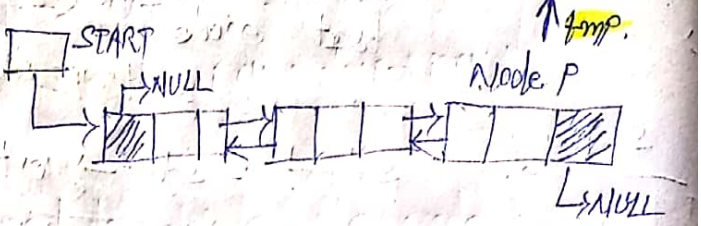
Before Deletion:

Node T is the last node next part is null.



After Deletion:

Node P is the last node. So, next part is NULL



Algorithm: Step-1: START

Step-2: Let "tmp" be the dynamic pointer.

Step-3: p->next = NULL; or

tmp->prev->next = NULL; (p = tmp->prev)

Step->4: free(tmp); (free the memory occupied by deleted node).

Step-5: EXIT

Explanation:

→ In this example nodeT is to be deleted, which is point by nodep.

So, p->next = NULL

→ The address of node p is store in temp prev.

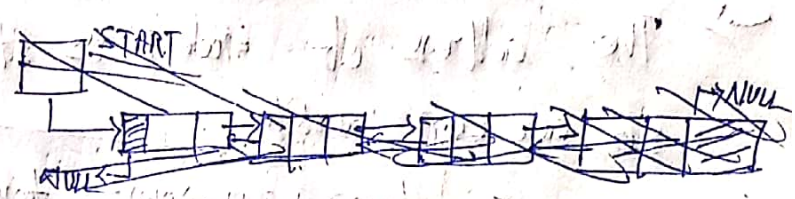
So, p = tmp->prev.

So, we can write tmp->prev->next = NULL;

→ Also free() is used to remove the node physically from the list.

Deletion an between the nodes

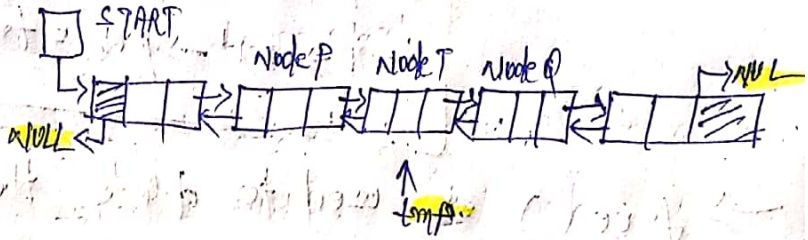
Before Deletion:



Node T is an between node P and node Q.

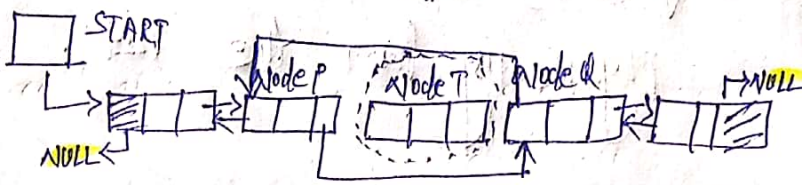
P next point to node T

Q prev " " node T.



After Deletion:

Node P points to node Q and Q prev point to node P.



Algorithm:

Step-1: START.

Step-2: Let "temp" be the dynamic pointer created.

Step-3: - `P -> next = temp -> next;` (Assign the address)

Step-4: - `q -> prev = P;` or

`temp -> next -> prev = P;` (Assig the previous node address)

Step-5: - `temp -> prev -> next = temp -> next;`

Step-6: - `temp -> next -> prev = temp -> prev;` (Assig next node)

Step-7: `free (temp)` (free the memory occupied by the deleted node).

Step-8: EXIT.

Explanation: -> In this example node T is between P & Q

So, we can write `P -> next = q;` `q -> prev = P;`

-> The address of q is in temp -> next, so, we can write `P -> next = temp -> next`

→ Also `tmp->next->prev = p;`

→ The address of node p is stored in tmp->prev. So, we can write:

```
tmp->prev->next = tmp->next;
tmp->next->prev = tmp->prev;
```

→ free() is used to delete the node physically from the list.