Unit II

Linear Data Structures using Link List Organization

Arrays
Fixed size: Resizing is expensive int a[10];
Insertions and Deletions are inefficient: Elements are usually shifted
Random access i.e., efficient indexing
No memory waste if the array is full or almost full; otherwise may result in much memory waste.
Sequential access is faster [Reason: Elements in contiguous memory locations]

Static & Dynamic Memory Management

- Static Memory Management means allocation and deallocation of memory at compilation time.
- Dynamic memory management refers to allocating and deallocating of memory while program is running(after compilation)
- Advantage of dynamic memory management in handling linklist is that we can create as many nodes as we desire and if some nodes are not required we can deallocate them. Such a deallocated memory can be reallocated for some other nodes.
- Thus the total memory utilization is possible using dynamic memory management

Sr. No.	Static Memory	Dynamic Memory
1	The Memory allocation is done at compilation time	The Memory allocation is done at c dynamic time
2	Prior to allocation of memory some fixed amount of it must be decided.	No need to know mount of memory prior to allocation.
3	Wastage of memory or shortage of memory	Memory can be allocated as per requirement.
4	E.g. Array	e.g. Linked list

Memory Model

Operating System programs and Program code
Memory for static variables
Stack memory for functions local data
Heap Memory

- Memory is divided into three parts static area, local data and heap
- Static area stores the global data.
- Stack is for local data i.e. for local variables
- Heap area is used to allocate and deallocate memory under program’s control
- Stack and heap area are the part of dynamic memory management.

Dynamic Memory Management Functions in C

1. malloc
2. calloc
3. realloc
4. free

malloc function

- **malloc**
- The malloc function is used to allocate the memory space as per the requirement
- This function does not initialize the memory allocated during execution. In fact it carries the garbage value.
- This function allocates requested block of memory and returns pointer of type void * to the start of that block.
- If function fails it returns NULL therefore it is necessary to verify if pointer returned is NOT NULL
- Syntax of malloc is
- malloc(size);
- E.g. Malloc(sizeof(int));

calloc function

- **calloc**
- The calloc is just similar to malloc but it initializes the allocated memory to zero
- The calloc () takes two arguments ,first is number of elements and second is type of elements and computes number of bytes to allocate.
- It allocates requested block and returns pointer to void pointing to beginning of block .If fails ,returns NULL.
- Syntax
- calloc(number , size)

realloc function

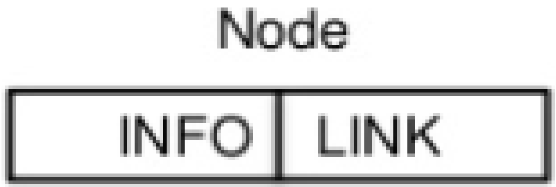
- **realloc**
- The realloc function is used to modify the size of allocated block by malloc () and calloc () functions to new size
- If enough space doesn’t exist in memory of current block to extend , new block is allocated for full size of reallocation , then copies the existing data to new block and then frees the old block
- Syntax
- realloc (pointerName , size);

Free

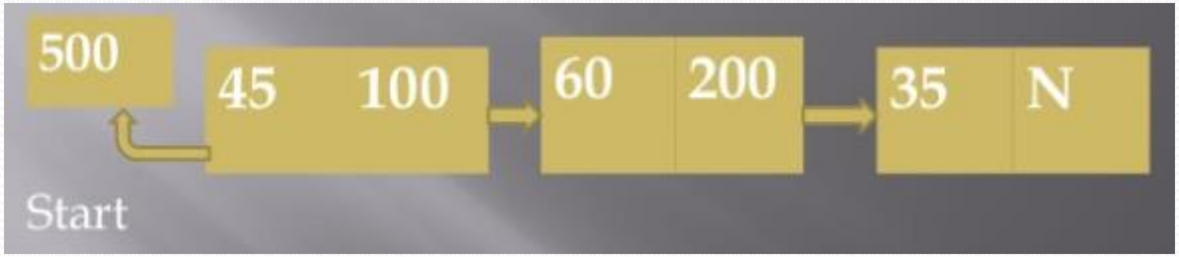
- The free function is used to deallocate the allocated memory .
- The **syntax**
- free (pointerName)

What are Linked Lists

- A linked list is a linear data structure.
- Linear collection of data elements called nodes each of which is a memory location that contains two parts. That is
- Data part
- Link part : usually the pointer to next to the next.



[Structure of a node in linked list]



- Dynamic size
- Insertions and Deletions are efficient: No shifting
- No random access
- Not suitable for operations requiring accessing elements by index such as sorting
- Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
- Sequential access is slow [Reason: Elements not in contiguous memory locations]

Linked List using dynamic memory allocatio

```
/* C implementation of Linked List */
#include<stdio.h>
#include<conio.h>
Typedef struct NODE
{
  int data;
  struct NODE *next;
}node;
Node n1,n2,n3,n4,n5;
Node *one, *temp;
void  main()
{
clrscr();
n1.data=10;
n1.next=&n2;
n2.data=20;
n2.next=&n3;
n3.data=30;
n3.next=&n4;
```

Filling data in each node and attaching the nodes to each other to form singly linked list

```
n4.data=40;
n4.next=&n5;
n5.data=50;
n5.next= NULL;
One=&n1;
Temp=one;
While(temp!=NULL)
{
Printf("\n %d", temp->data);
Temp=temp->next;
}
getch();
}
```

Output:

```
10
20
30
40
50
```

Linked List as ADT

AbstractDataType LinkedList

{

Instances: Linked listis a collection of nodes and each node is having one data filed and next link field.

Operations:

1. Create: This function creates the linked list
2. Insert: By this function any desired node value can be inserted at any desired position.
3. Delete: This function allows to delete any node present at any position.
4. Search: this function helps to search any data value present in the node.

}

Types of Linked Lists

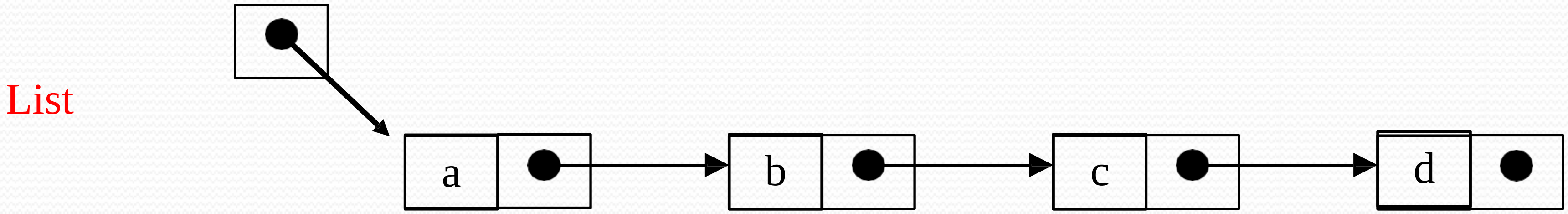
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List

Singly Linked List

- All the nodes are linked in some sequential manner
- Each node has only one link part
- Each link part contains the address of the next node in the list
- Link part of the last node contains NULL value which signifies the end of the node

Schematic representation

- Here is a singly-linked list (SLL):



- Each node contains a value(data) and a pointer to the next node in the list

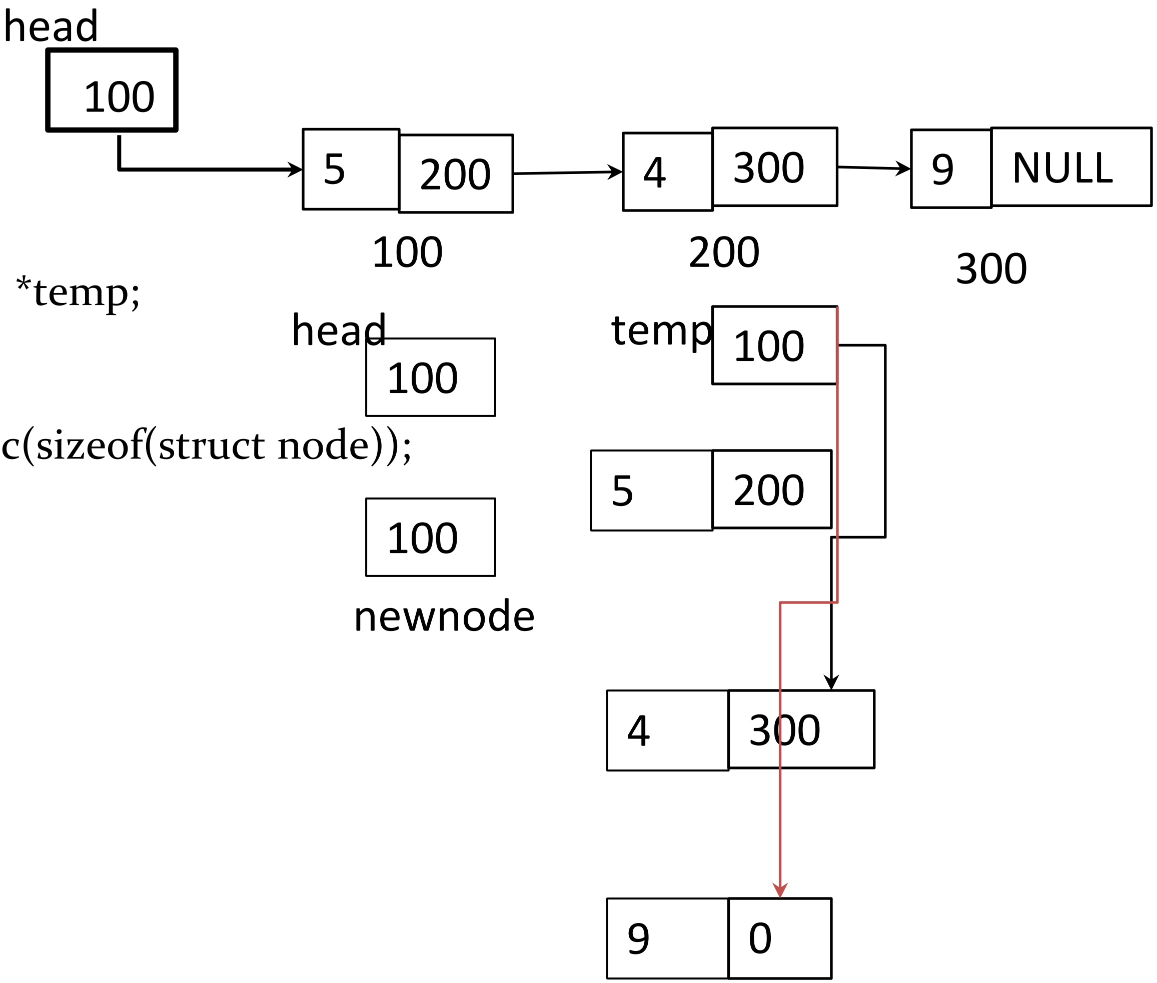
- List is the header pointer which points at the first node in the list

Basic Operations on a list

- Creating a List
- Inserting an element in a list
- Deleting an element from a list
- Searching a list
- Reversing a list

Creation of singly linklist

```
void main()
{
struct node
{
int data;
struct node *next;
};
struct node * head, *newnode, *temp;
while(choice)
{
newnode=(struct node*)malloc(sizeof(struct node));
printf(“Enter Data”);
scanf(“%d”,&newnode->data);
newnode->next=NULL;
if(head==NULL)
{
head =temp=newnode;
}
else
{
temp->next=newnode;
temp=newnode;
}
printf(“Do you want to continue(0,1)?”);
scanf(“%d”,&choice);
}
```



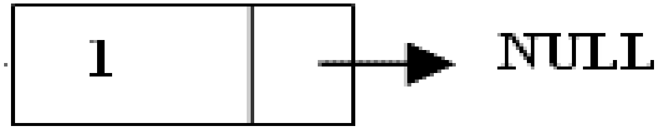
Display elements of linklist

```
temp=head
while(temp!=0)
{printf(“%d”, temp->data);
temp=temp->next;
}
getch();
}
```

Creating a node

```
struct node{
        int data;                // A simple node of a linked list
        node *next;
    }*start;                     //start   points at the first node
start=NULL ;                       initialised to NULL at beginning
```

```
node* create( int num) //say num=1 is passed from main
{
    node*ptr;
    ptr=  (struct node*)malloc(sizeof(struct node));
          //memory allocated dynamically
          if(ptr==NULL)
            ‘OVERFLOW’ // no memory available
              exit(1);
          else
          {
             ptr->data=num;
             ptr->next=NULL;
             return ptr;
          }
}
```



To be called from main() as:-

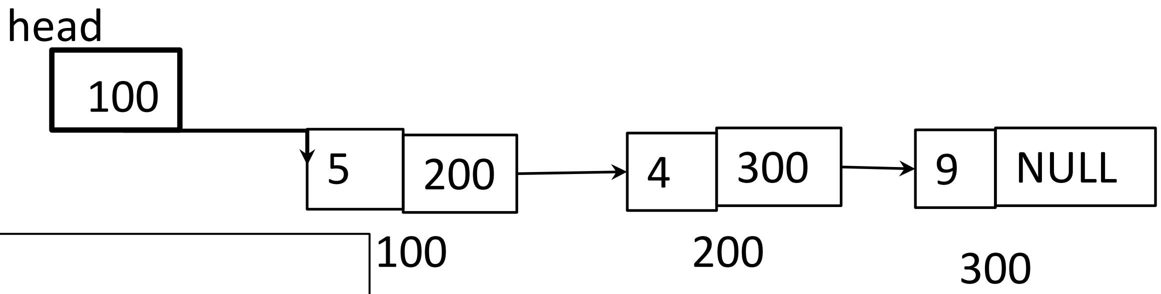
```
 void main()
{
    node* ptr;
    int data;
    printf(“enter the data to insert”);
    Scanf(“%d”, &data);
    ptr=create(data);
 }
```

Inserting the node in a SLL

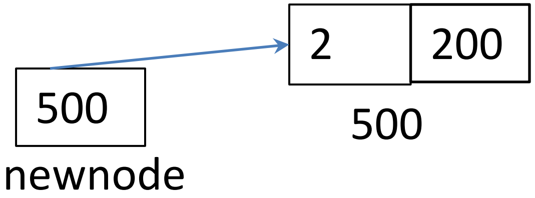
There are 3 cases here:-

- Insertion at the beginning
- Insertion at the end
- Insertion after a particular node

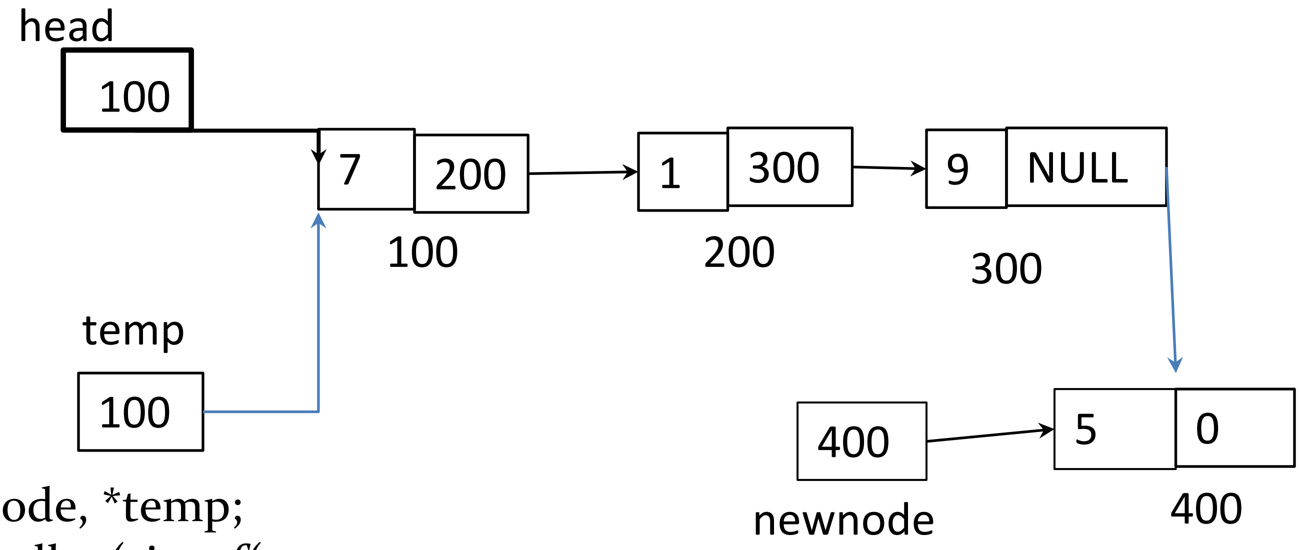
Insertion at beginning of singly linklist



```
struct node
{
int data;
struct node *next;
};
struct node * head, *newnode, *temp;

newnode=(struct
node*)malloc(sizeof(struct node));
printf(“Enter Data”);
scanf(“%d”,&newnode->data);
newnode->next=head;
head=newnode;
```

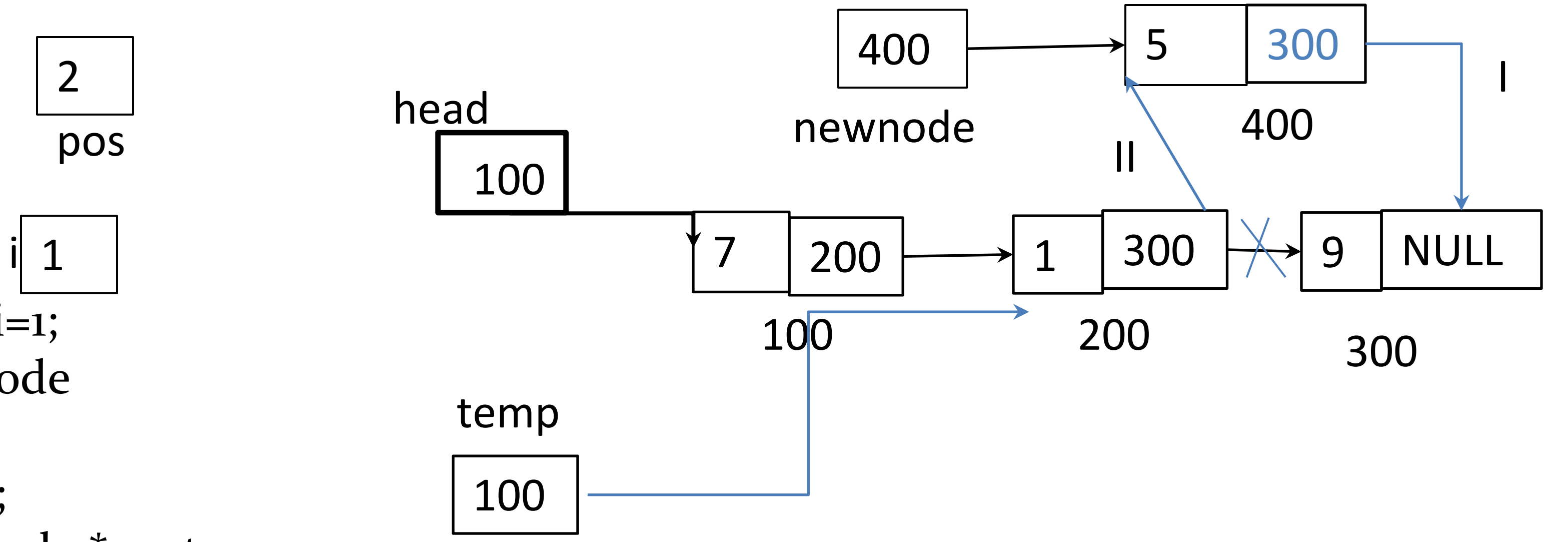


Insertion at end of singly linklist



```
struct node
{
int data;
Struct node *next;
};
struct node * head, *newnode, *temp;
newnode=(struct node*)malloc(sizeof(struct
node));
printf("Enter Data you want to insert: ");
scanf("%d",&newnode->data);
newnode->next=NULL;
temp=head;
while(temp->next !=NULL)
{
  temp – temp->next;
}
 temp->next=newnode;
```

Insertion after a given location in singly linklist



```
Int pos,i=1;
struct node
{
int data;
Struct node *next;
};
struct node * head, *newnode, *temp;
printf(“Enter Position”);
scanf(“%d”,& pos);
if (pos>count)
{
printf(“invalid Position”);
 }
else
{
temp=head;
```

```
while(i<pos)
{
 temp-temp->next;
i++;
}
newnode=(struct node*)malloc(sizeof(struct node));
printf(“Enter Data”);
scanf(“%d”,&newnode->data);
newnode->next=temp->next;
temp->next=newnode;
```

Insertion at the beginning

There are two steps to be followed:-

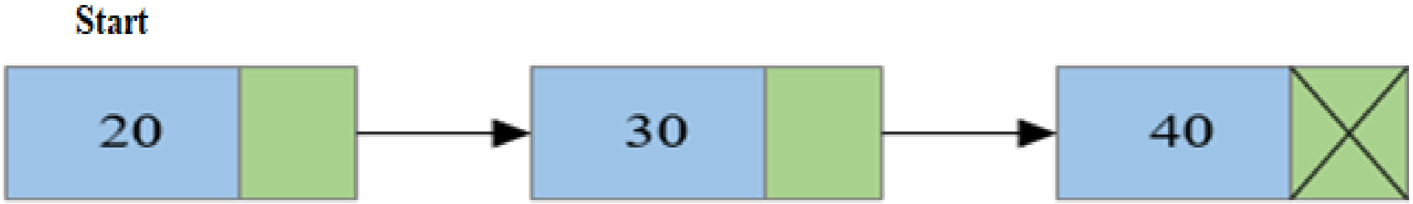
a) Make the next pointer of the node point towards the first node of the list

a) Make the start pointer point towards this new node

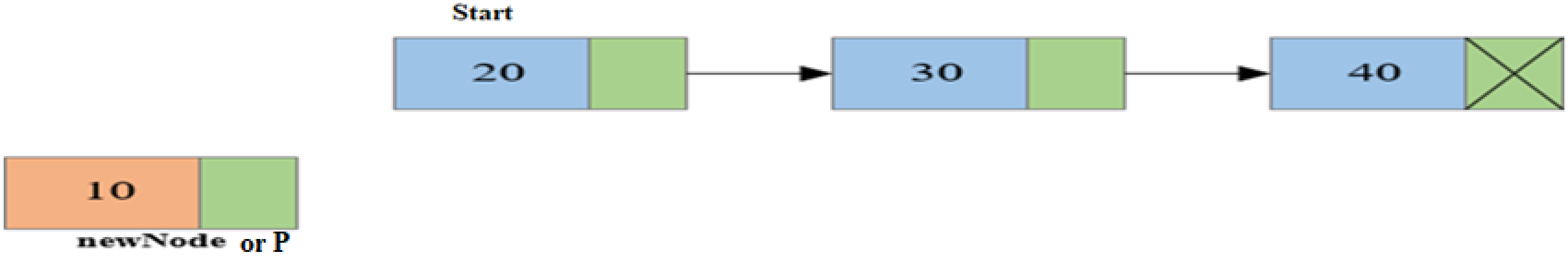
- If the list is empty simply make the start pointer point towards the new node;

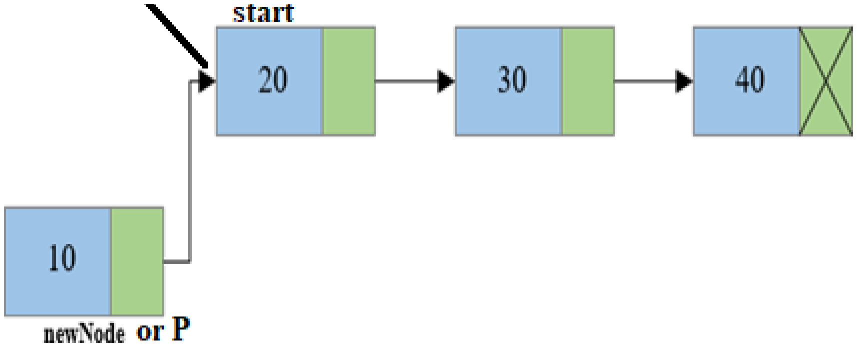
Insert node at the beginning of Singly Linked List

- Steps to insert a new node at the start of a singly linked list.



Create a new node, say newNode points to the newly created

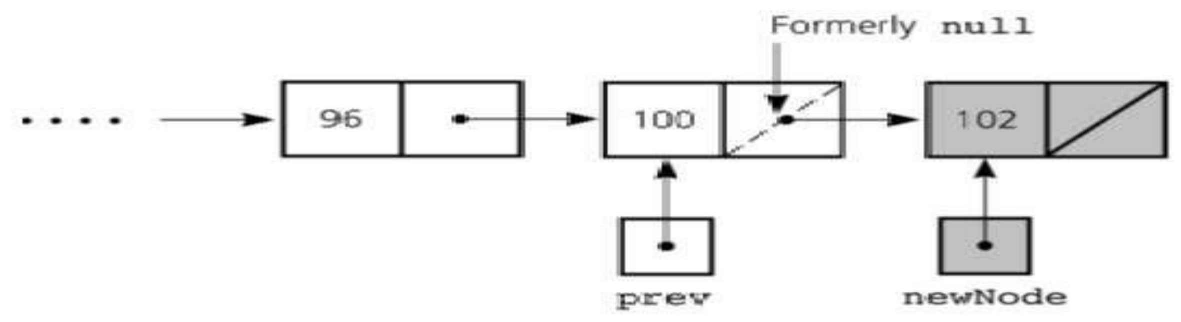




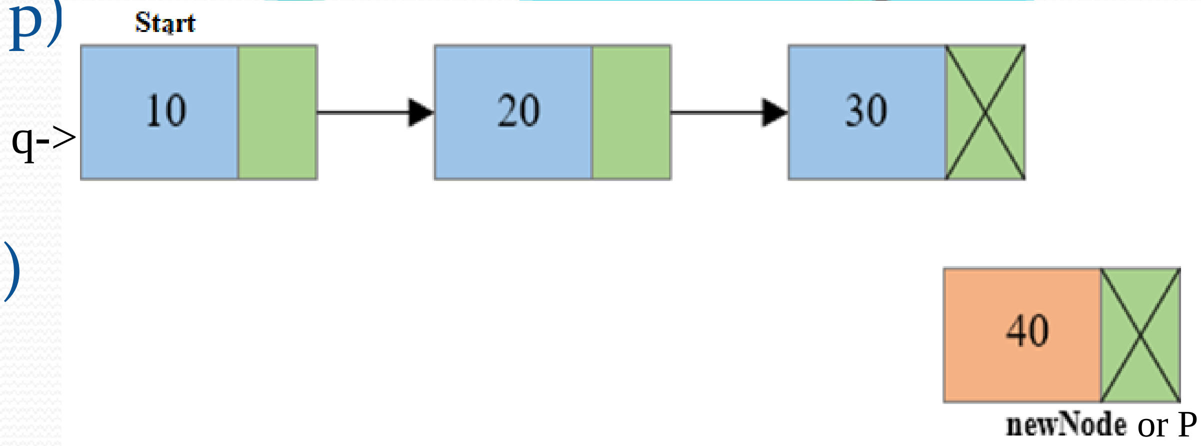
```
void insert_beg(node* p)
{
node* temp;
        if(start==NULL)
        {
           start=p;
           Printf("\nNode inserted successfully at the  beginning");
        }
        else{


              temp=start;
              start=p;
              p->next=temp;  //making new node point at
             }                    the first node of the list
   }
```

Inserting at the end

Here we simply need to make the next pointer of the last node point to the new node



```
void insert_end(node* p)
{
node *q=start;
          if(start==NULL)
         {
           start=p;
           printf(”\nNode inserted successfully at the
           end…!!!\n”);
         }
      else{
                  while(q->link!=NULL)
                    q=q->link;
             q->next=p;
            }
}
```

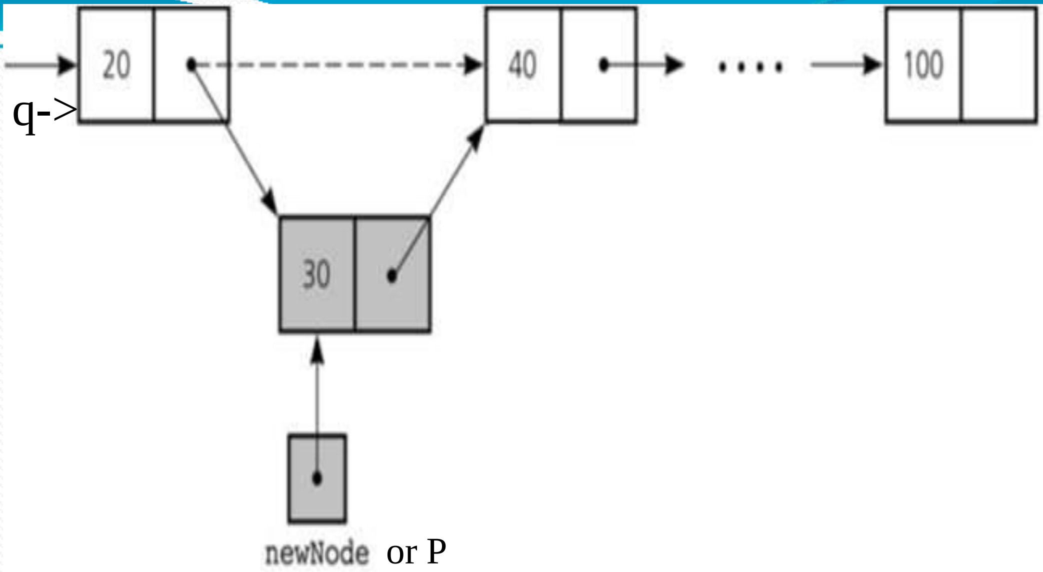


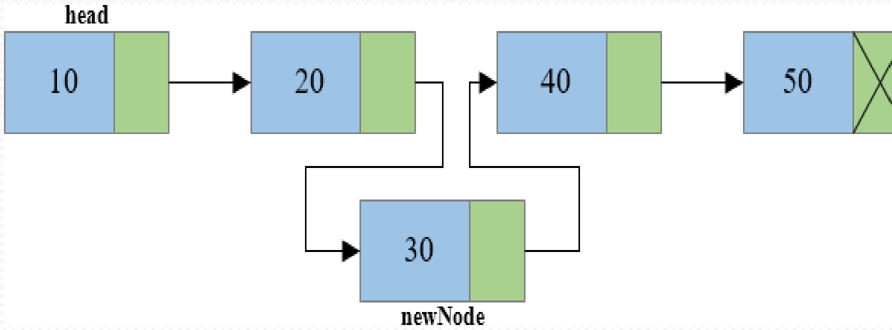
Inserting after an element

Here we again need to do 2 steps :-

- Make the next pointer of the node to be inserted point to the next node of the node after which you want to insert the node
- Make the next pointer of the node after which the node is to be inserted, point to the node to be inserted

```
void insert_after(int c,node* p)
{
node*
q;
q=start;
        for(int i=1;i<c;i++)
       {
          q=q->link;
                   if(q==NULL)
                  printf(“Less than %d”, nodes in the list…!!!”,c);
       }
 p->link=q->link;
 q->link=p;
printf(”\nNode inserted successfully”
}
```





Deleting a node in SLL

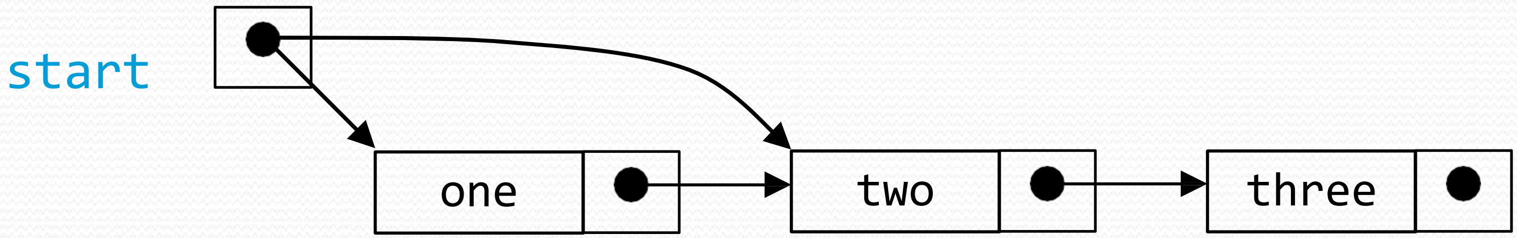
Here also we have three cases:-

- Deleting the first node
- Deleting the last node
- Deleting the intermediate node

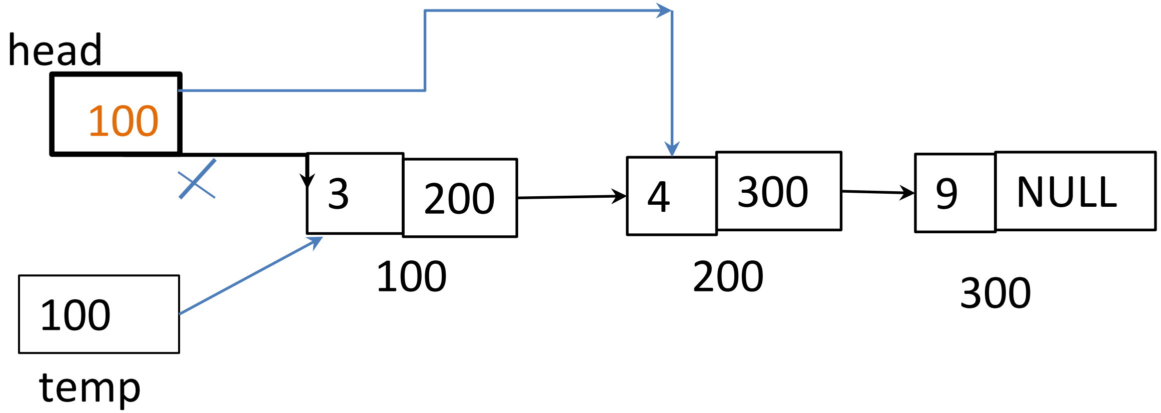
Deleting the first node

Here we apply 2 steps:-

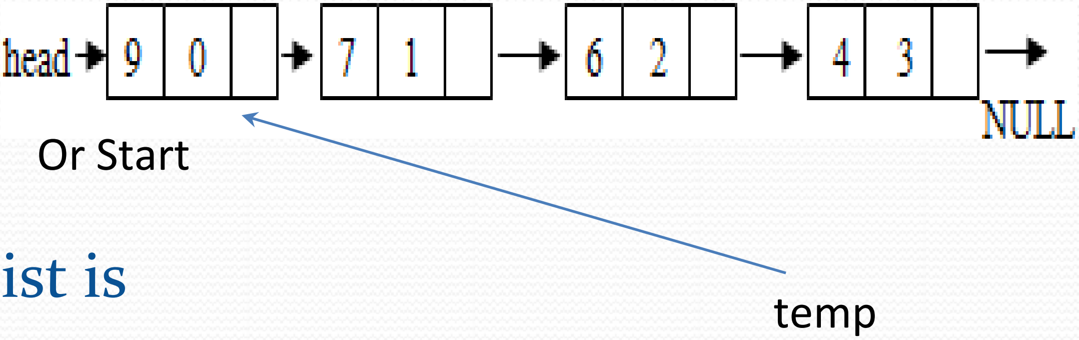
- Making the start pointer point towards the 2nd node
- Deleting the first node using delete keyword



Deletion at beginning of singly linklist



```
struct node
{
int data;
struct node *next;
};
struct node * head, *newnode, *temp;
deletefrom_beg()
{
   temp=head;
   head=head->next;
free(temp);
}
```



Or Start

temp

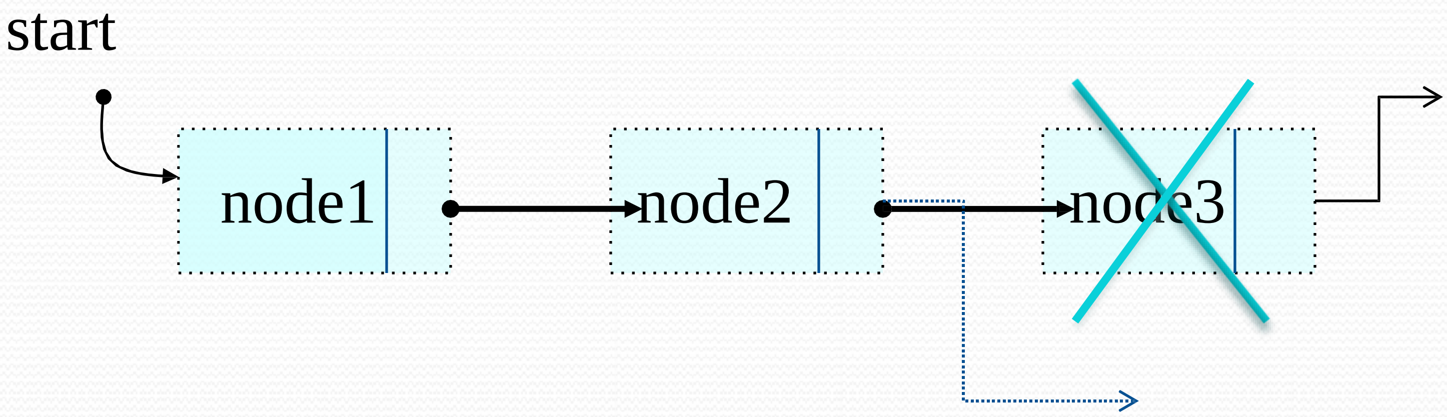
```
void del_first()
{
     if(start==NULL)
    printf(”\nError......List is
    empty\n”);  else
    {
      node* temp=start;
      start=temp->link;
      free(temp);
printf(”\nFirst              node              deleted
successfully....!!!”);
    }
}
```

Deleting the last node

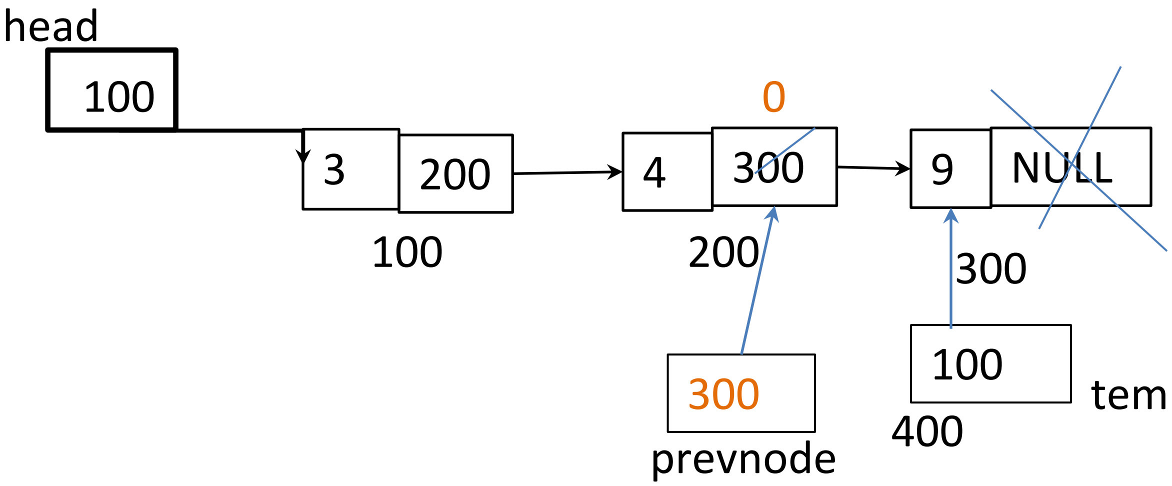
Here we apply 2 steps:-

- Making the second last node's next pointer point to NULL

- Deleting the last node via delete keyword

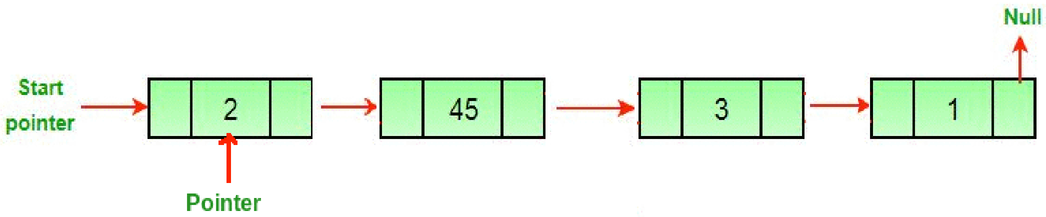


Deletion at end of singly linklist



```
struct  node
{
int data;
struct node *next;
};
struct node * head, *temp;
deletefrom_end()
{
  struct node *prevnode;
temp=head;
while(temp->next!=NULL)
{
    prevnode=temp;
    temp=temp->next;
}
```

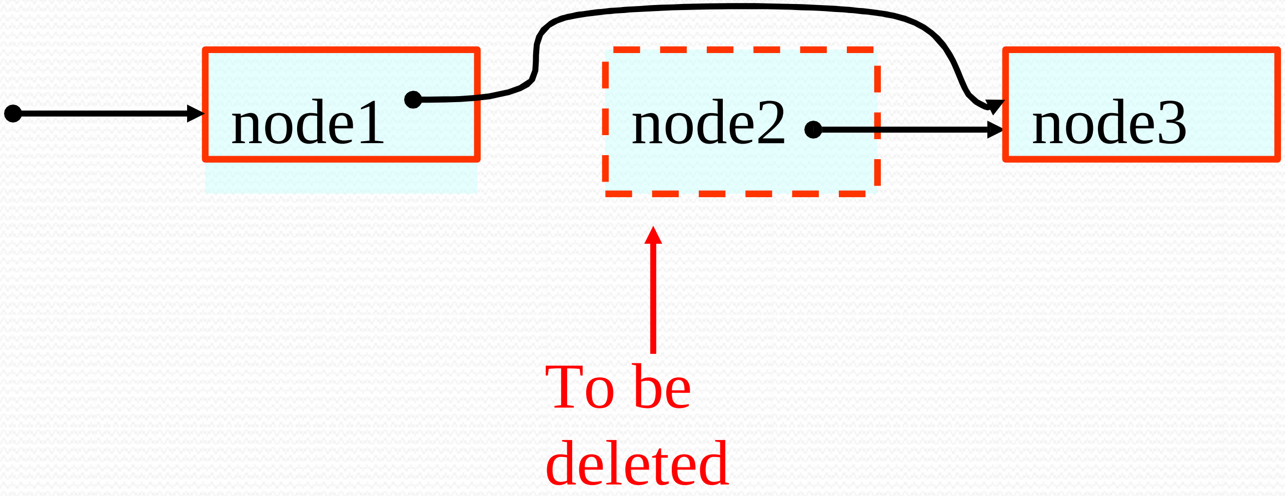
```
If(temp==head)
{
head=NULL;
Free(temp);
}
else
{
prevnode->next=NULL;
}
free(temp);
}
}
```



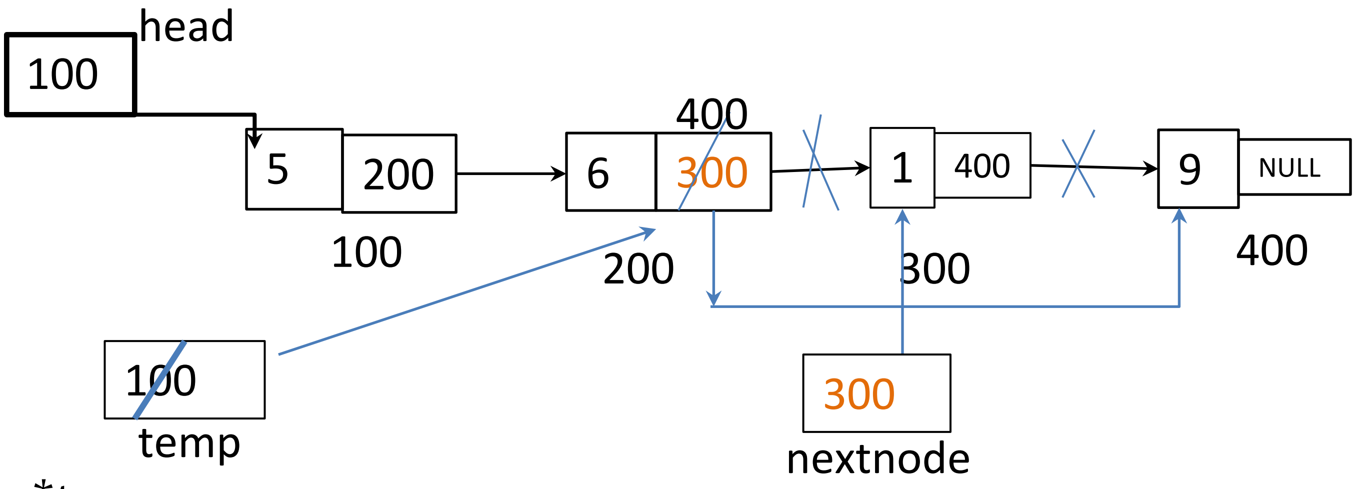
```
void del_last()
{

if(start==NULL)
printf(”\nError....List is empty”);  else
                    {
                        node* q=start;
                                 while(q->link->link!=NULL)
                                  q=q->link;
                        node* temp=q->link;
                        q->link=NULL;
                        free(temp);
                        Printf(”\nDeleted successfully…”);
                    }
        }
}
```

Deleting a particular node

Here we make the next pointer of the node previous to the node being deleted ,point to the successor node of the node to be deleted and then delete the node using delete keyword



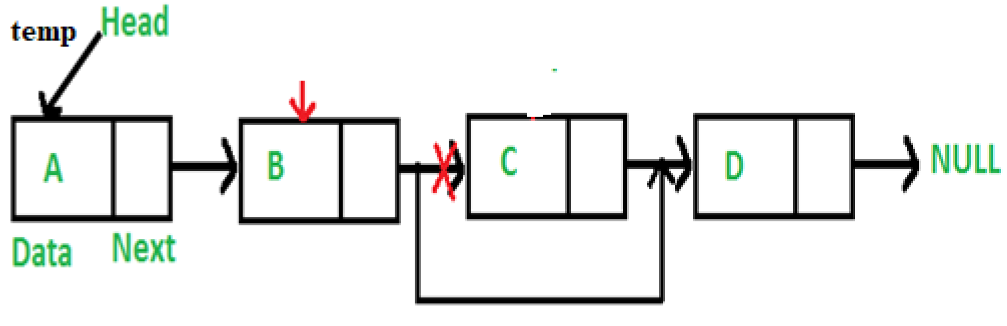
Deletion from specified position in singly linklist

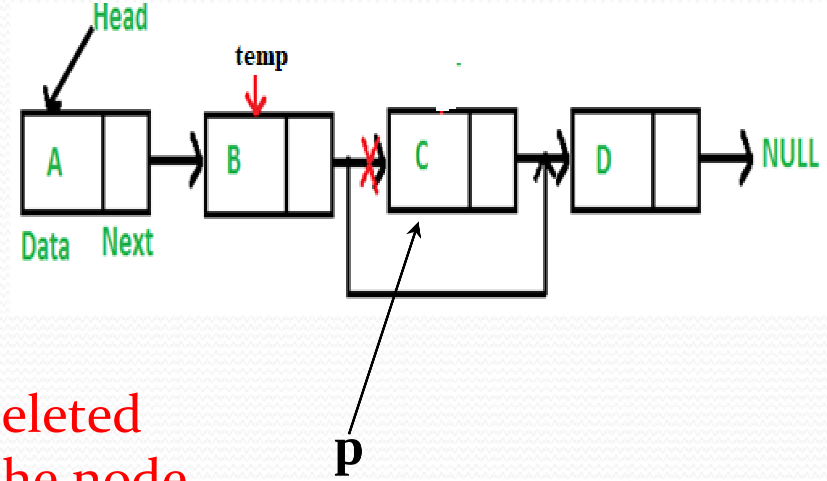


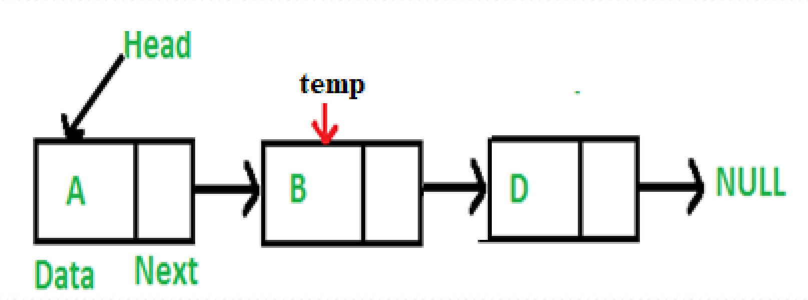
```
struct  node
{
int data;
struct node *next;
};
struct node * head, *temp;
deletefrom_position()
{
  struct node *nextnode; int pos,i=1;
temp=head;
printf(“Enter Position”);
scanf(“%d”,&pos);
while(i<pos-1)
{
    temp=temp->next;
    i++;
}
```

```
nextnode=temp->next;
temp->next = nextnode->next;
free(nextnode);}
```

```
void del(int c)
{
node* temp=head;
       for(int i=2;i<c;i++)
       {
         temp=temp->link;
                 if(temp==NULL)
                 Printf(”\nNode not
                 found\n”);
       }
       if(i==c)
       { node* p=temp->link;  //node to be deleted
        temp->link=p->link;//disconnecting the node
        free( p);                p
        printf(“Deleted Successfully”);
       }
}
```



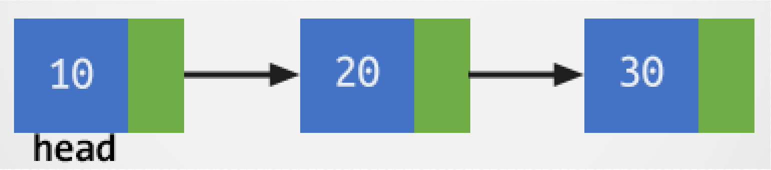


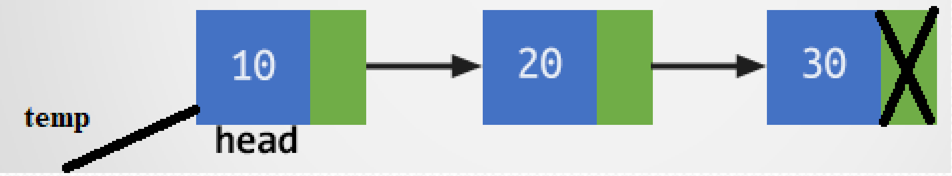


Searching a SLL

- Searching involves finding the required element in the list
- We can use various techniques of searching like linear search or binary search where binary search is more efficient in case of Arrays
- But in case of linked list since random access is not available it would become complex to do binary search in it
- We can perform simple linear search traversal

In linear search each node is traversed till the data in the node matches with the required value





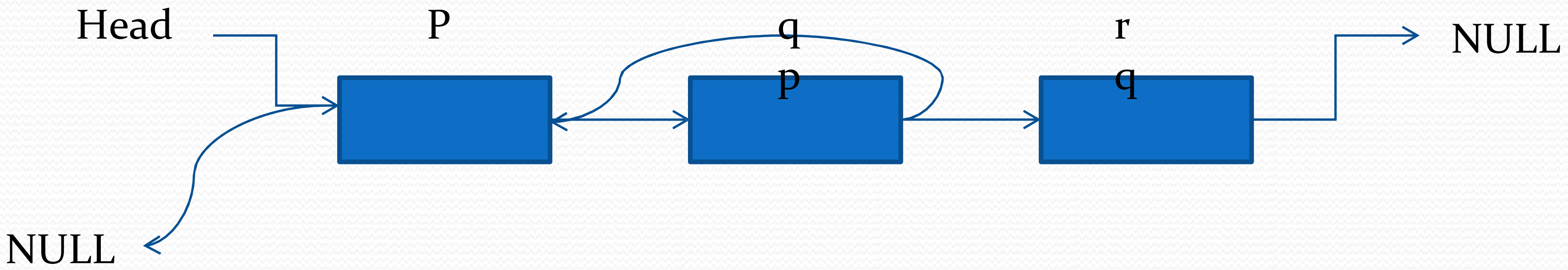
```
void search(int x) x=20
{
  node*temp=start;
     while(temp!=NULL)
        {
          if(temp->data==x)
           {
             printf(“FOUND %d”, temp->data);
             break;
           }
         temp=temp->next;
        }
}
```

Reversing a linked list

- We can reverse a linked list by reversing the direction of the links between 2 nodes

- We make use of 3 structure pointers say p,q,r

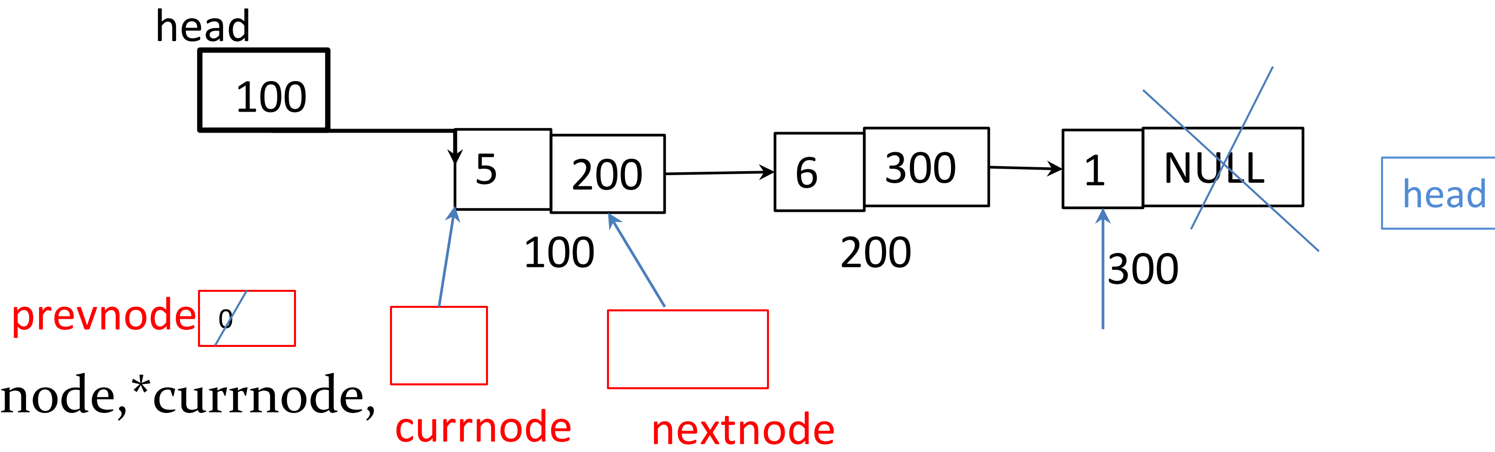
- At any instant q will point to the node next to p and r will point to the node next to q



- For next iteration p=q and q=r

- At the end we will change head to the last node

Reverse singly linklist



```
void reverse()
{
struct node *prevnode,*currnode,
   *nextnode;
 if(head==NULL)
   {
    cout<<"\nList is empty\n";
    return;
   }
prevnode=0;
currentnode=nextnode=head;
```

```
   while(nextnode!=NULL)
   {
    nextnode=nextnode->next;
    currnode->next=prevnode;
}
    prevnode=nextnode;
   }
   head=prevnode;
   cout<<"\nReversed successfully";
}
```

Inserting node to singly linked list

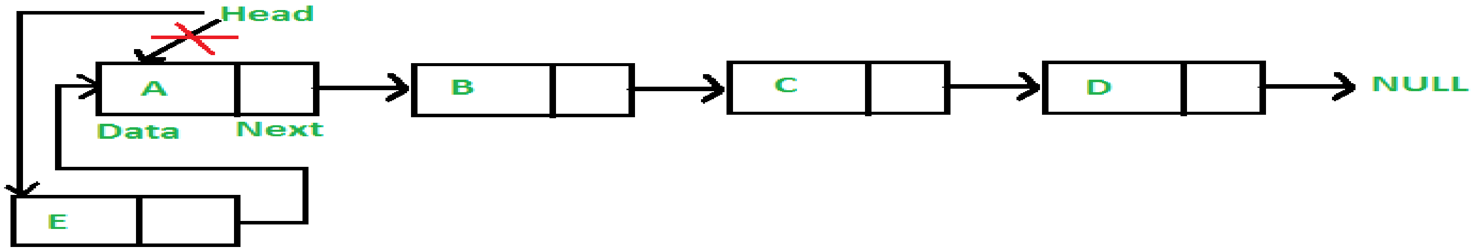
Linked List

```
class Node
{
   public:
   int data;
   Node *next;
};
```

- Inserting Node in Singly
- A node can be added in three ways
  1) At the front of the linked list
  2) After a given node.
  3) At the end of the linked list.

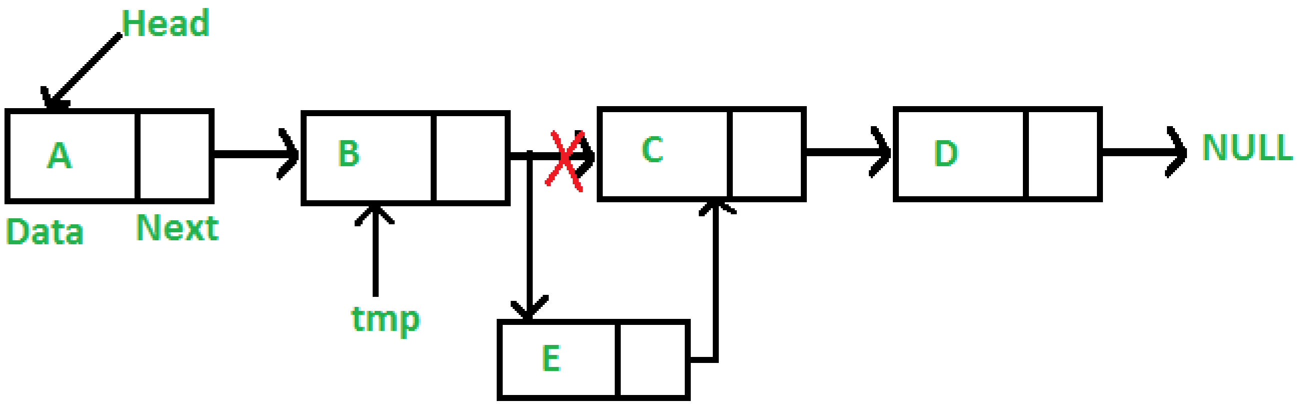
Add a node at the front: (4 steps process)

- The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



```
/* Given a reference (pointer to pointer)
to the head of a list and an int,
inserts a new node on the front of the list. */
void push(Node** head_ref, int new_data)
{
   /* 1. allocate node */
   Node* new_node = new Node();

   /* 2. put in the data */
   new_node->data = new_data;

   /* 3. Make next of new node as head */
   new_node->next = (*head_ref);

   /* 4. move the head to point to the new node */
   (*head_ref) = new_node;
}
```

- Time complexity of push() is O(1) as it does a constant amount of work.

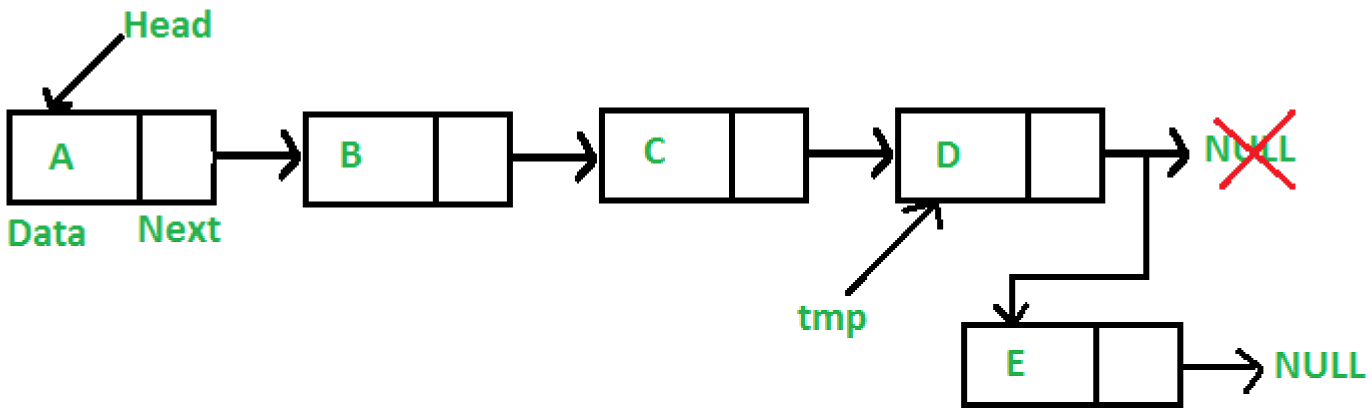
- **Add a node after a given node: (5 steps process)**
  We are given a pointer to a node, and the new node is inserted after the given node.
- 

```
//Given a node prev_node, insert a
// new node after the given
// prev_node
void insertAfter(Node* prev_node, int new_data)
{

        // 1. Check if the given prev_node is NULL
        if (prev_node == NULL)
        {
                cout << "the given previous node cannot be NULL";
                return;
        }

        // 2. Allocate new node
        Node* new_node = new Node();

        // 3. Put in the data
        new_node->data = new_data;

        // 4. Make next of new node as
        // next of prev_node
        new_node->next = prev_node->next;

        // 5. move the next of prev_node
        // as new_node
        prev_node->next = new_node;
}
```

Time complexity of insertAfter() is O(1) as it does a constant amount of work.

Add a node at the end: (6 steps process)

- The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.
  Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.



```
// Given a reference (pointer to pointer) to the head
// of a list and an int, appends a new node at the end
void append(Node** head_ref, int new_data)
{

        // 1. allocate node
        Node* new_node = new Node();

        // Used in step 5
        Node *last = *head_ref;

        // 2. Put in the data
        new_node->data = new_data;

        // 3. This new node is going to be
        // the last node, so make next of
        // it as NULL
        new_node->next = NULL;

        // 4. If the Linked List is empty,
        // then make the new node as head
        if (*head_ref == NULL)
        {
                *head_ref = new_node;
                return;
        }

        // 5. Else traverse till the last node
        while (last->next != NULL)
                last = last->next;

        // 6. Change the next of last node
        last->next = new_node;      return;
}
```

ime complexity of append is O(n) where n is the number of nodes in linked list. Since there is a loop from head to end, the function does O(n) work.

This method can also be optimized to work in O(1) by keeping an extra pointer to the tail of linked list/

Deleting a node from singly linked list

- **Iterative Method:**
  To delete a node from the linked list, we need to do the following steps.
  1) Find the previous node of the node to be deleted.
  2) Change the next of the previous node.
  3) Free memory for the node to be deleted.
- ..\link2 delete slide.docx

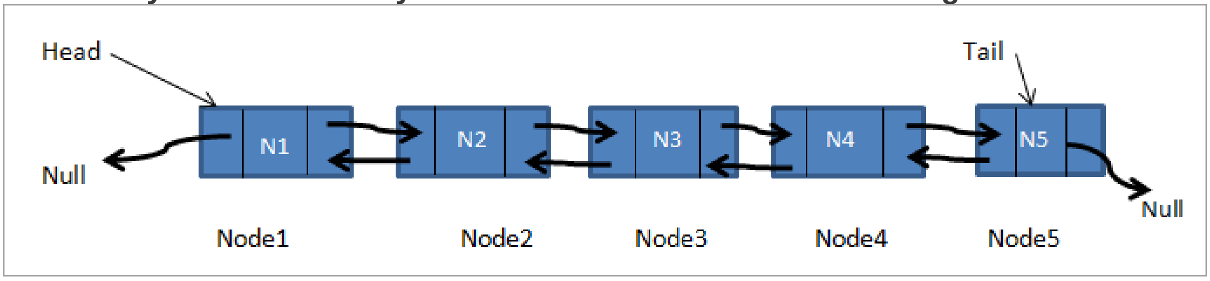
An In-Depth Tutorial On Doubly Linked List.
A doubly linked list is a variation of the singly linked list. We are aware that the singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.

A doubly linked list is also a collection of nodes. Each node here consists of a data part and two pointers. One pointer points to the previous node while the second pointer points to the next node.

Doubly Linked In C++

As in the singly linked list, the doubly linked list also has a head and a tail. The previous pointer of the head is set to NULL as this is the first node. The next pointer of the tail node is set to NULL as this is the last node.

A basic layout of the doubly linked list is shown in the below diagram.



In the above figure, we see that each node has two pointers, one pointing to the previous node and the other pointing to the next node. Only the first node (head) has its previous node set to null and the last node (tail) has its next pointer set to null.

As the doubly linked list contains two pointers i.e. previous and next, we can traverse it into the directions forward and backward. This is the main advantage of doubly linked list over the singly linked list.

Declaration

In C-style declaration, a node of the doubly linked list is represented as follows:

```
struct node

{

        struct node *prev;

        int data;

        struct node *next;

};
```

Basic Operations

Following are some of the operations that we can perform on a doubly linked list.

Insertion

Insertion operation of the doubly linked list inserts a new node in the linked list. Depending on the position where the new node is to be inserted, we can have the following insert operations.

- **Insertion at front –** Inserts a new node as the first node.
- **Insertion at the end –** Inserts a new node at the end as the last node.
- **Insertion before a node –** Given a node, inserts a new node before this node.
- **Insertion after a node –** Given a node, inserts a new node after this node.

Deletion

Deletion operation deletes a node from a given position in the doubly linked list.

- **Deletion of the first node –** Deletes the first node in the list
- **Deletion of the last node –** Deletes the last node in the list.
- **Deletion of a node given the data –** Given the data, the operation matches the data with the node data in the linked list and deletes that node.

Traversal

Traversal is a technique of visiting each node in the linked list. In a doubly linked list, we have two types of traversals as we have two pointers with different directions in the doubly linked list.

- **Forward traversal –** Traversal is done using the next pointer which is in the forward direction.
- **Backward traversal –** Traversal is done using the previous pointer which is the backward direction.

Reverse

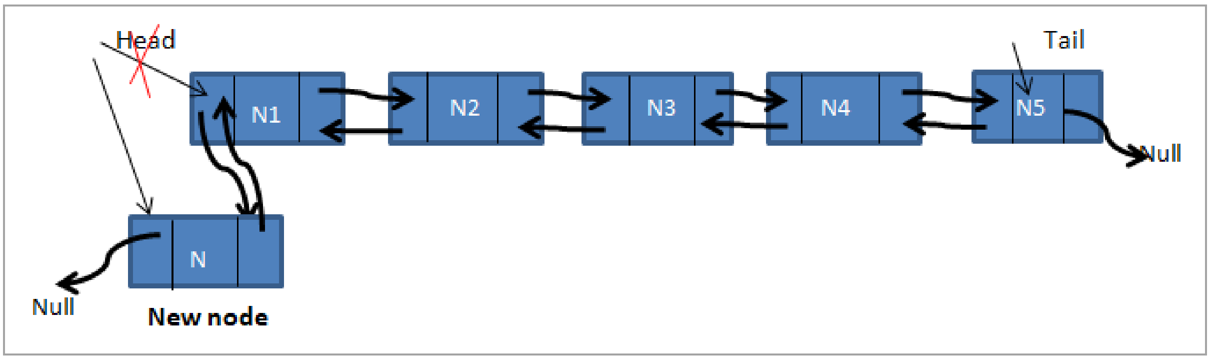
This operation reverses the nodes in the doubly linked list so that the first node becomes the last node while the last node becomes the first node.

Search

Search operation in the doubly linked list is used to search for a particular node in the linked list. For this purpose, we need to traverse the list until a matching data is found.

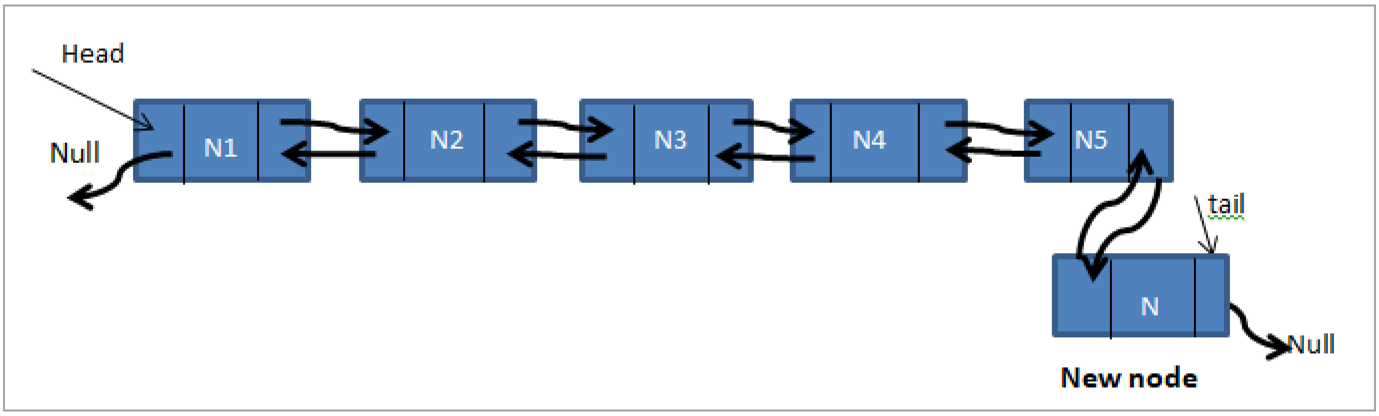
Insertion

Insert a node at the front



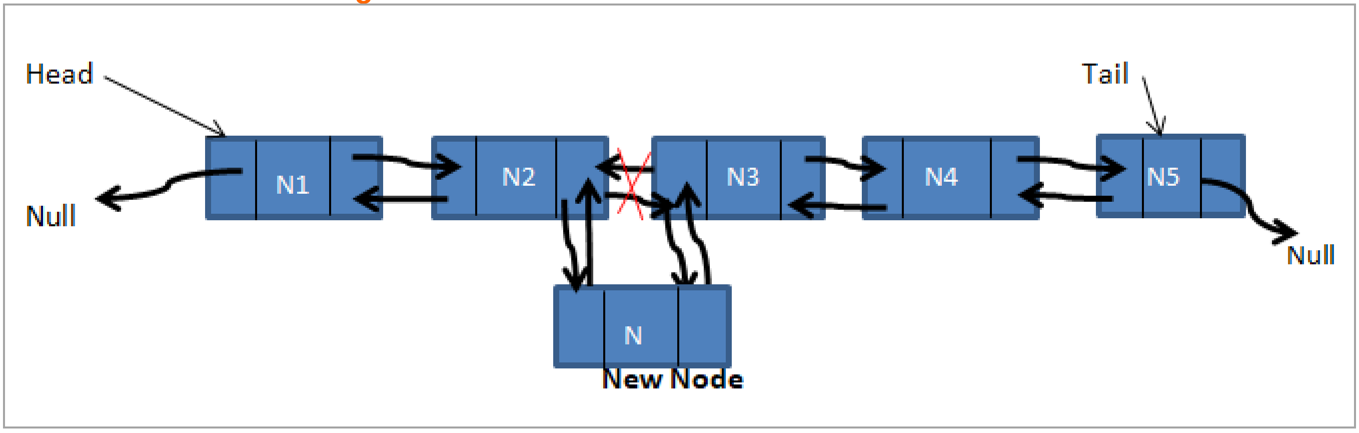
Insertion of a new node at the front of the list is shown above. As seen, the previous new node N is set to null. Head points to the new node. The next pointer of N now points to N1 and previous of N1 that was earlier pointing to Null now points to N.

Insert node at the end



Inserting node at the end of the doubly linked list is achieved by pointing the next pointer of new node N to null. The previous pointer of N is pointed to N5. The 'Next' pointer of N5 is pointed to N.

Insert node before/after given node



As shown in the above diagram, when we have to add a node before or after a particular node, we change the previous and next pointers of the before and after nodes so as to appropriately point to the new node. Also, the new node pointers are appropriately pointed to the existing nodes.

The following C++ program demonstrates all the above methods to insert nodes in the doubly linked list.

```
#include <iostream>
using namespace std;

// A doubly linked list node
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};
```

```
//inserts node at the front of the list
void insert_front(struct Node** head, int new_data)
{
   //allocate memory for New node
   struct Node* newNode = new Node;

   //assign data to new node
   newNode->data = new_data;

   //new node is head and previous is null, since we are adding at the front
   newNode->next = (*head);
   newNode->prev = NULL;

   //previous of head is new node
   if ((*head) != NULL)
   (*head)->prev = newNode;

   //head points to new node
   (*head) = newNode;
}
/* Given a node as prev_node, insert a new node after the given node */
void insert_After(struct Node* prev_node, int new_data)
{
   //check if prev node is null
   if (prev_node == NULL) {
   cout<<"Previous node is required , it cannot be NULL";
   return;
}
   //allocate memory for new node
   struct Node* newNode = new Node;

   //assign data to new node
   newNode->data = new_data;

   //set next of newnode to next of prev node
   newNode->next = prev_node->next;

   //set next of prev node to newnode
   prev_node->next = newNode;

   //now set prev of newnode to prev node
   newNode->prev = prev_node;

   //set prev of new node's next to newnode
   if (newNode->next != NULL)
   newNode->next->prev = newNode;
}

//insert a new node at the end of the list
void insert_end(struct Node** head, int new_data)
{
   //allocate memory for node
   struct Node* newNode = new Node;

   struct Node* last = *head; //set last node value to head

   //set data for new node
   newNode->data = new_data;
   //new node is the last node , so set next of new node to null
   newNode->next = NULL;

   //check if list is empty, if yes make new node the head of list
   if (*head == NULL) {
   newNode->prev = NULL;
   *head = newNode;
    return;
}

//otherwise traverse the list to go to last node
while (last->next != NULL)
last = last->next;

//set next of last to new node
last->next = newNode;

//set last to prev of new node
newNode->prev = last;
return;
}

// This function prints contents of linked list starting from the given node
void displayList(struct Node* node) {
   struct Node* last;

   while (node != NULL) {
      cout<<node->data<<"<==>";
      last = node;
      node = node->next;
   }
   if(node == NULL)
   cout<<"NULL";
   }

//main program
int main() {
   /* Start with the empty list */
   struct Node* head = NULL;

   // Insert 40 as last node
   insert_end(&head, 40);

   // insert 20 at the head
   insert_front(&head, 20);

   // Insert 10 at the beginning.
   insert_front(&head, 10);

   // Insert 50 at the end.
   insert_end(&head, 50);

   // Insert 30, after 20.
   insert_After(head->next, 30);

   cout<<"Doubly linked list is as follows: "<<endl;
   displayList(head);
```

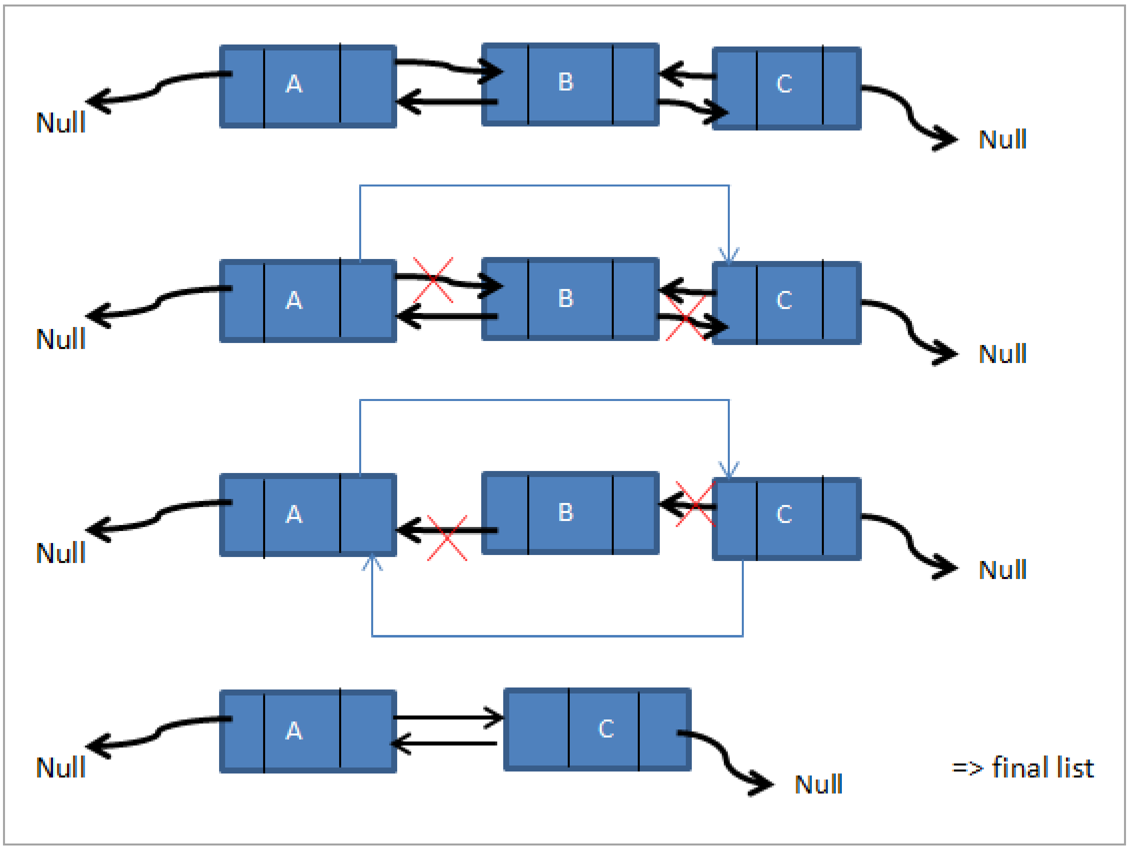
```
    return 0;
}
```

Deletion

A node can be deleted from a doubly linked list from any position like from the front, end or any other given position or given data.

When deleting a node from the doubly linked list, we first reposition the pointer pointing to that particular node so that the previous and after nodes do not have any connection to the node to be deleted. We can then easily delete the node.

Consider the following doubly linked list with three nodes A, B, C. Let us consider that we need to delete the node B.



As shown in the above series of the diagram, we have demonstrated the deletion of node B from the given linked list. The sequence of operation remains the same even if the node is first or last. The only care that should be taken is that if in case the first node is deleted, the second node's previous pointer will be set to null.

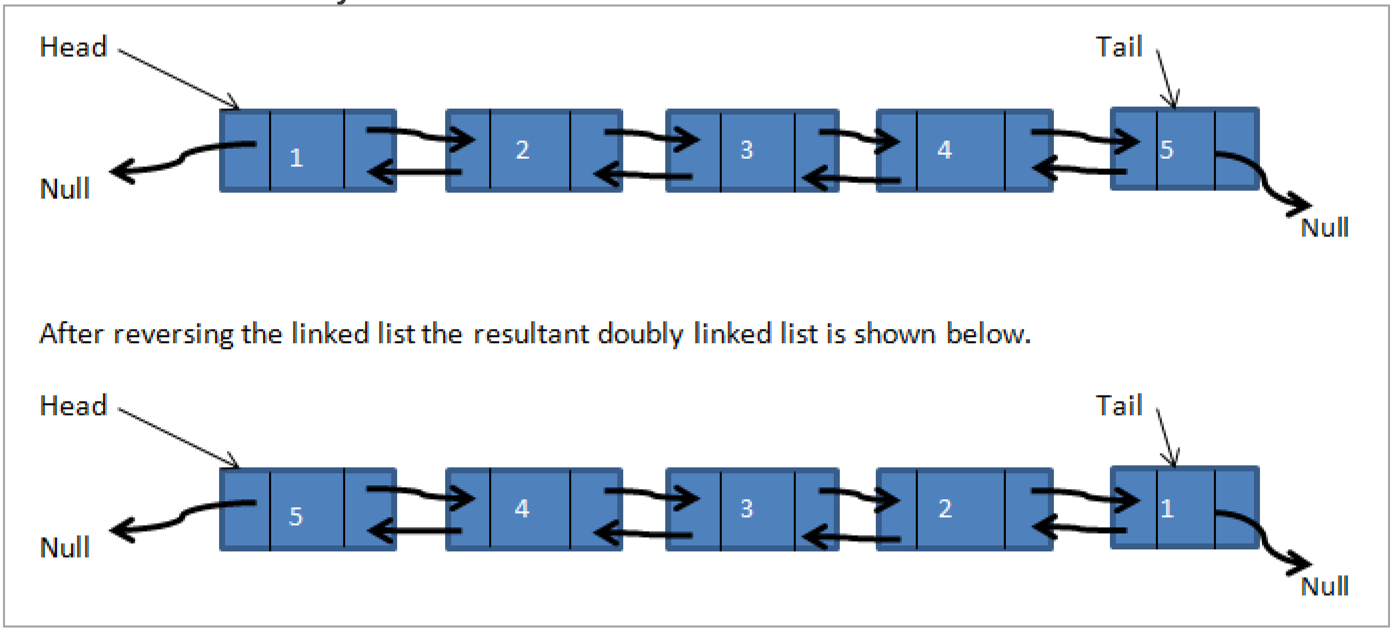
Similarly, when the last node is deleted, the next pointer of the previous node will be set to null. If in between nodes are deleted, then the sequence will be as above.

We leave the program to delete a node from a doubly linked list. Note that the implementation will be on the lines of the insertion implementation.

Reverse Doubly Linked List

Reversing a doubly linked list is an important operation. In this, we simply swap the previous and next pointers of all the nodes and also swap the head and tail pointers.

Given below is a doubly linked list:



After reversing the linked list the resultant doubly linked list is shown below.

Following C++ implementation shows the Reverse Doubly Linked List.

```
#include <iostream>

using namespace std;

//node declaration for doubly linked list

struct Node {

   int data;

   struct Node *prev, *next;

};



Node* newNode(int val)

{

   Node* temp = new Node;

   temp->data = val;

   temp->prev = temp->next = nullptr;

   return temp;

}

void displayList(Node* head)
```

```
{
   while (head->next != nullptr) {
      cout << head->data << " <==> ";
      head = head->next;
      }
   cout << head->data << endl;
}

// Insert a new node at the head of the list
void insert(Node** head, int node_data)
{
   Node* temp = newNode(node_data);
   temp->next = *head;
   (*head)->prev = temp;
   (*head) = temp;
}

// reverse the doubly linked list
void reverseList(Node** head)
{
   Node* left = *head, * right = *head;

   // traverse entire list and set right next to right
   while (right->next != nullptr)
   right = right->next;

   //swap left and right data by moving them towards each other till they meet or cross
   while (left != right && left->prev != right) {
      // Swap left and right pointer data
      swap(left->data, right->data);

      // Advance left pointer
      left = left->next;

      // Advance right pointer
      right = right->prev;
   }
}
int main()
{
   Node* headNode = newNode(5);
   insert(&headNode, 4);
   insert(&headNode, 3);
   insert(&headNode, 2);
   insert(&headNode, 1);
   cout << "Original doubly linked list: " << endl;
   displayList(headNode);
   cout << "Reverse doubly linked list: " << endl;
   reverseList(&headNode);
   displayList(headNode);

   return 0;
}
```

Advantages/Disadvantages Over Singly Linked List

Let us discuss some of the advantages and disadvantages of doubly linked list over the singly linked list.

Advantages:

- The doubly linked list can be traversed in forward as well as backward directions, unlike singly linked list which can be traversed in the forward direction only.
- Delete operation in a doubly-linked list is more efficient when compared to singly list when a given node is given. In a singly linked list, as we need a previous node to delete the given node, sometimes we need to traverse the list to find the previous node. This hits the performance.
- Insertion operation can be done easily in a doubly linked list when compared to the singly linked list.

Disadvantages:

- As the doubly linked list contains one more extra pointer i.e. previous, the memory space taken up by the doubly linked list is larger when compared to the singly linked list.
- Since two pointers are present i.e. previous and next, all the operations performed on the doubly linked list have to take care of these pointers and maintain them thereby resulting in a performance bottleneck.

Applications Of Doubly Linked List

A doubly linked list can be applied in various real-life scenarios and applications as discussed below.

- A Deck of cards in a game is a classic example of a doubly linked list. Given that each card in a deck has the previous card and next card arranged sequentially, this deck of cards can be easily represented using a doubly linked list.
- Browser history/cache – The browser cache has a collection of URLs and can be navigated using the forward and back buttons is another good example that can be represented as a doubly linked list.
- Most recently used (MRU) also can be represented as a doubly linked list.
- Other data structures like Stacks, hash table, the binary tree can also be constructed or programmed using a doubly linked list.