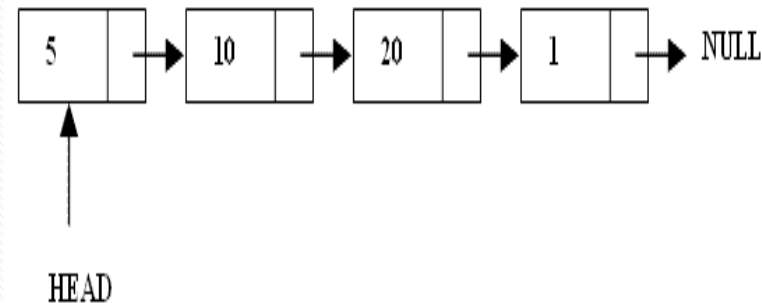


What are Linked Lists

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called next.



Arrays Vs Linked Lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Types of lists

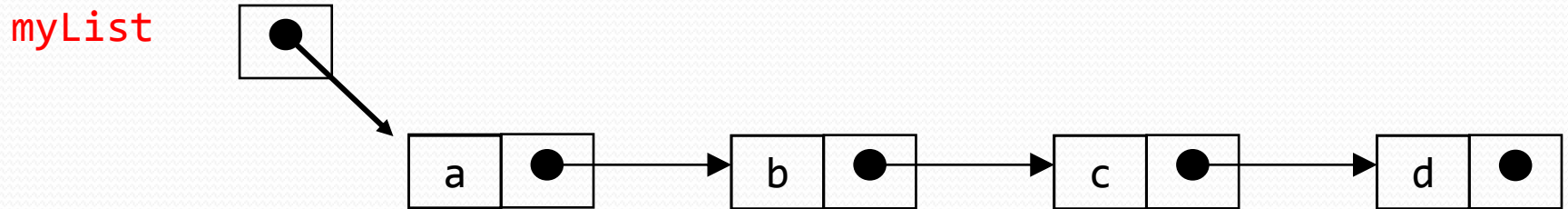
- There are two basic types of linked list
- Singly Linked list
- Doubly linked list

Singly Linked List

- 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
- **myList** 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

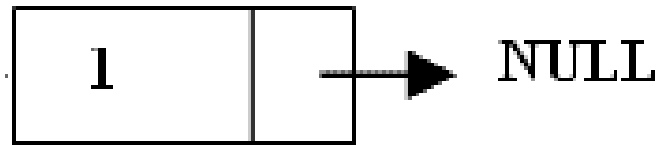
Creating a node

```
struct node{  
    int data;  
    node*next;  
}*start;  
start=NULL ;
```

// A simple node of a linked list

//start points at the first node
initialised to NULL at beginning

```
node* create( int num) //say num=1 is passed from main
{
    node*ptr;
    ptr= new 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;
    cin>>data;
    ptr=create(data);
}
```

Inserting the node in a SLL

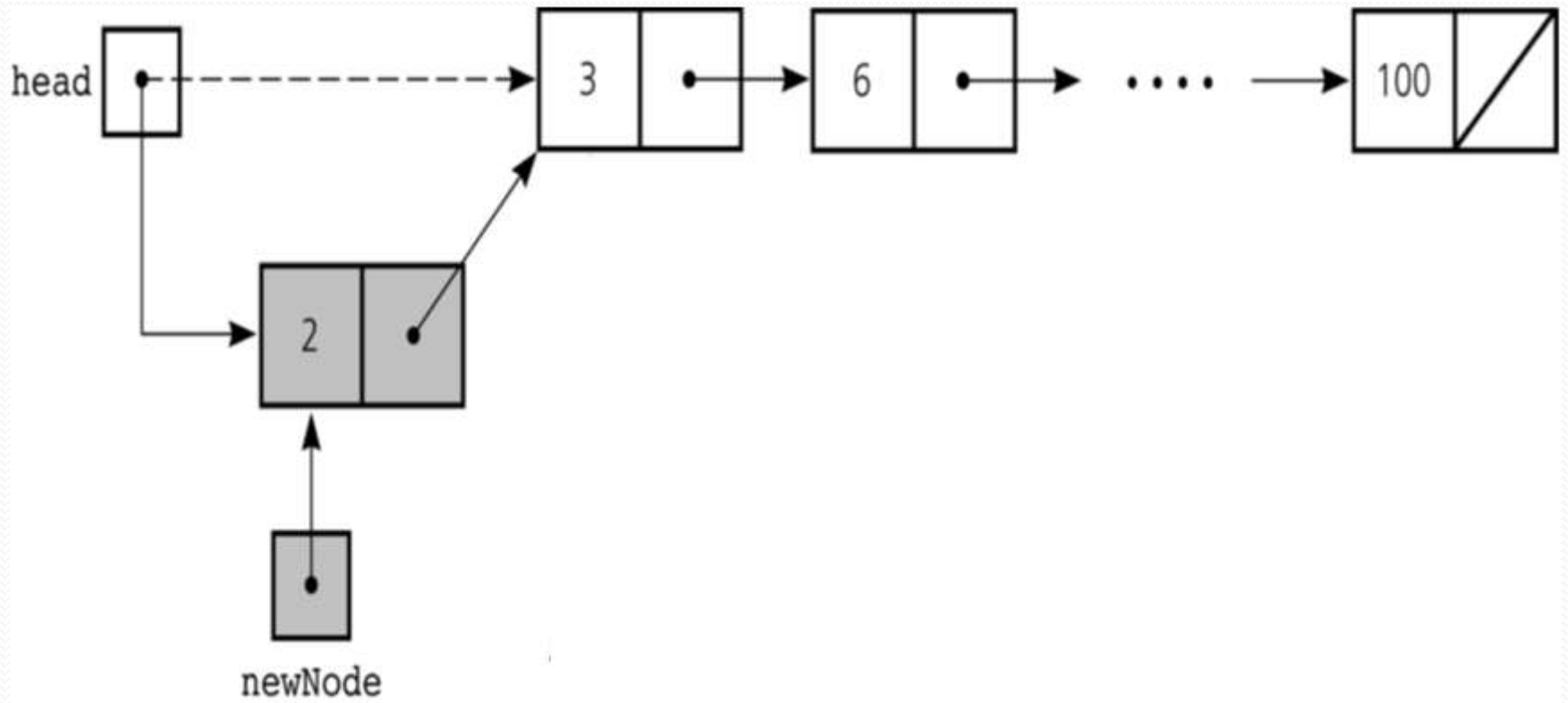
There are 3 cases here:-

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

Insertion at the beginning

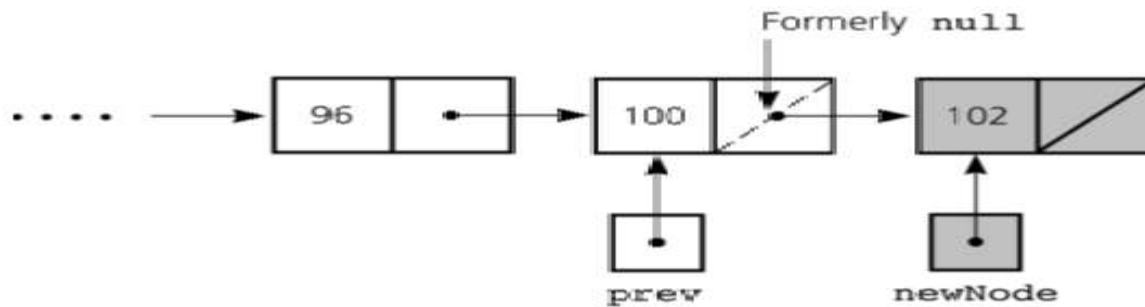
There are two steps to be followed:-

- a) Make the next pointer of the node point towards the first node of the list
- b) Make the start pointer point towards this new node
 - If the list is empty simply make the start pointer point towards the new node;



Inserting at the end

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

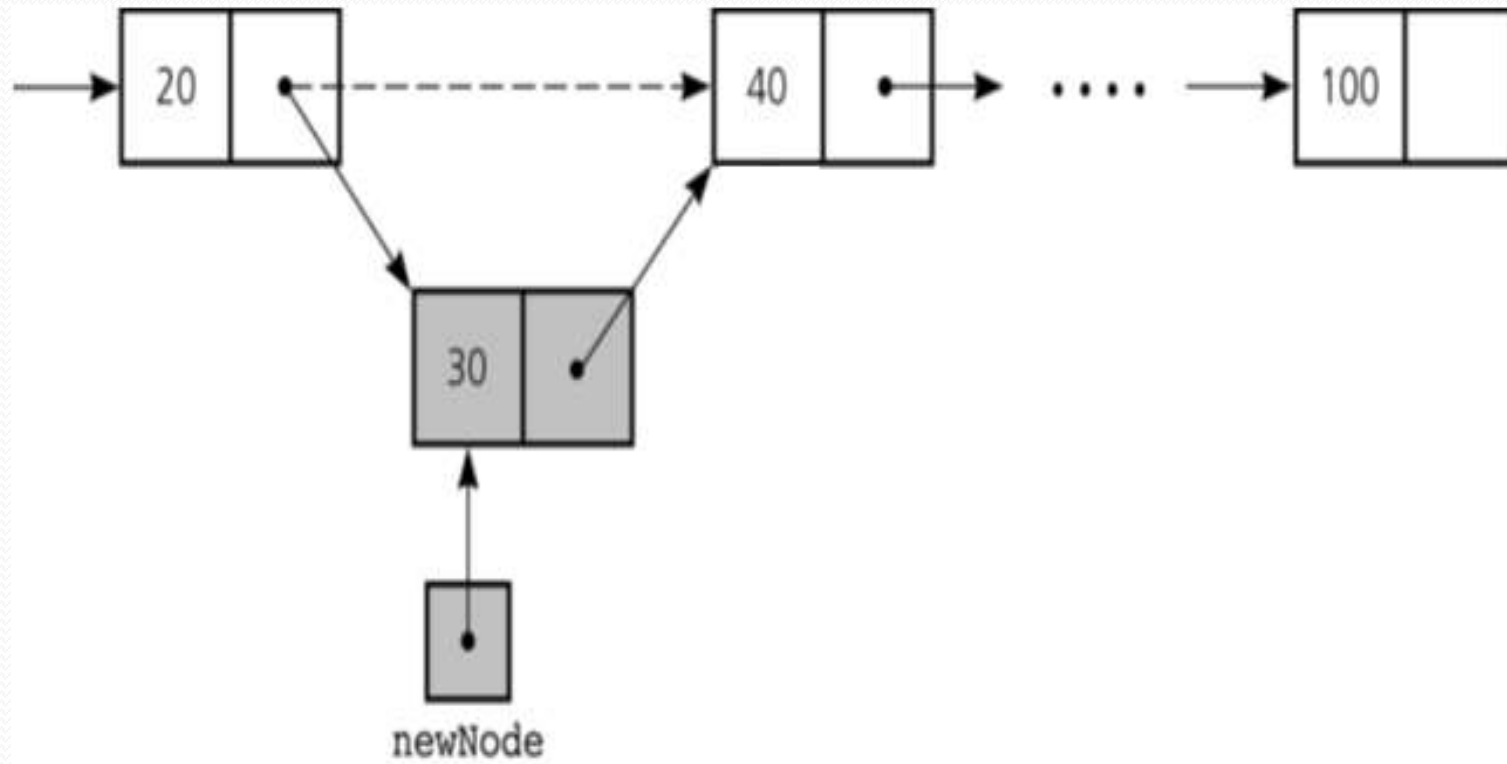


```
void insert_end(node* p)
{
    node *q=start;
    if(start==NULL)
    {
        start=p;
        cout<<"\nNode inserted successfully at the end...!!!\n";
    }
    else{
        while(q->link!=NULL)
            q=q->link;
        q->next=p;
    }
}
```

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)
            cout<<"Less than "<<c<<" nodes in the list...!!!";
    }
    p->link=q->link;
    q->link=p;
    cout<<"\nNode inserted successfully";
}
```

Deleting a node in SLL

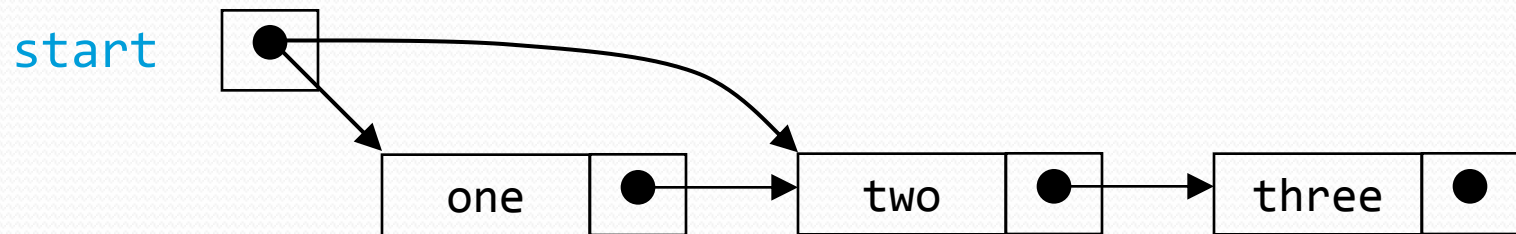
Here also we have three cases:-

- 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

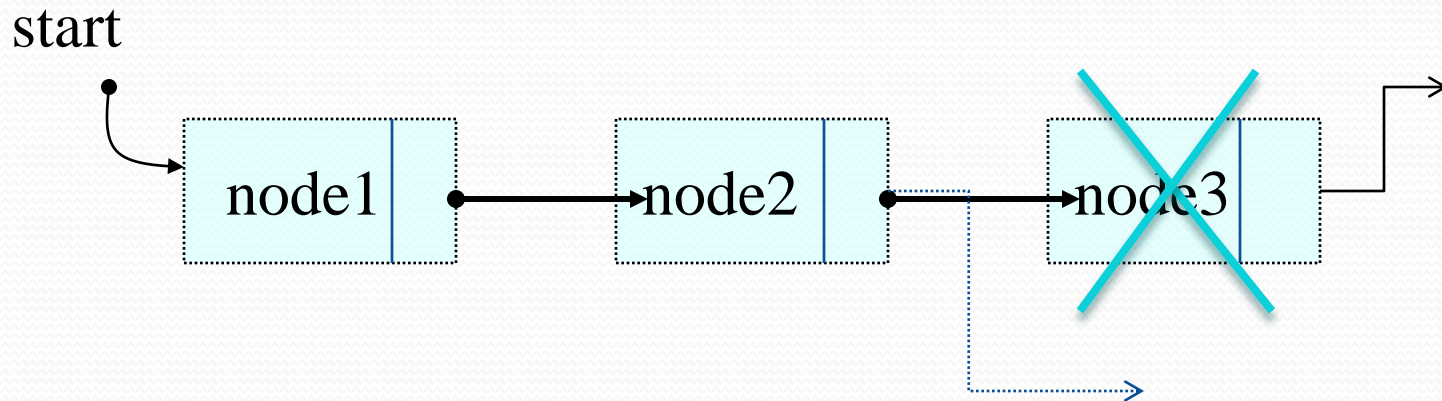


```
void del_first()
{
    if(start==NULL)
        cout<<"\nError.....List is empty\n";
    else
    {
        node* temp=start;
        start=temp->link;
        delete temp;
        cout<<"\nFirst node deleted successfully....!!!";
    }
}
```

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



```
void del_last()
```

```
{
```

```
    if(start==NULL)
```

```
        cout<<"\nError....List is empty";
```

```
    else
```

```
        {
```

```
            node* q=start;
```

```
            while(q->link->link!=NULL)
```

```
                q=q->link;
```

```
            node* temp=q->link;
```

```
            q->link=NULL;
```

```
            delete temp;
```

```
            cout<<"\nDeleted successfully...";
```

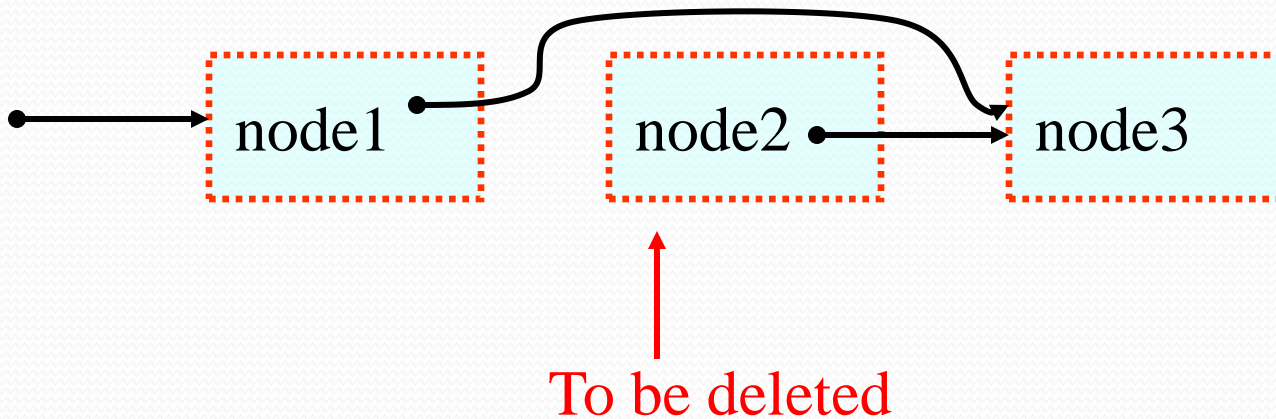
```
        }
```

```
    }
```

```
}
```

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




```
void del(int c)
{
    node* q=start;
    for(int i=2;i<c;i++)
    {
        q=q->link;
        if(q==NULL)
            cout<<"\nNode not found\n";
    }
    if(i==c)
    {
        node* p=q->link;    //node to be deleted
        q->link=p->link;    //disconnecting the node p
        delete p;
        cout<<"Deleted Successfully";
    }
}
```

Searching a SLL

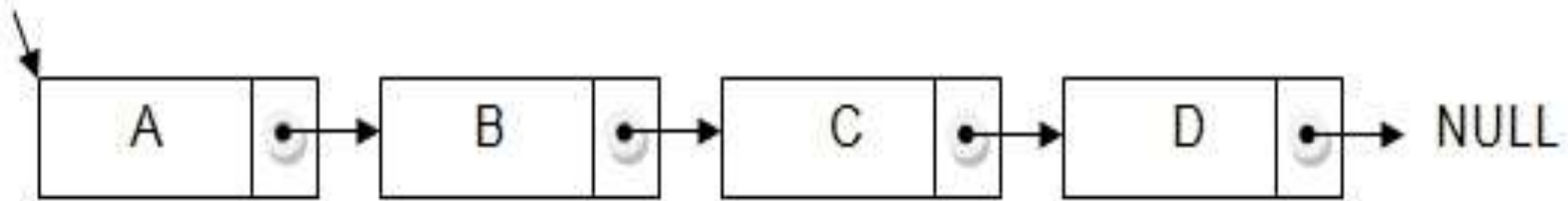
- Searching involves finding the required element in the list
- We can use various techniques of searching like linear search or binary search where binary search is more efficient in case of Arrays
- But in case of linked list since random access is not available it would become complex to do binary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

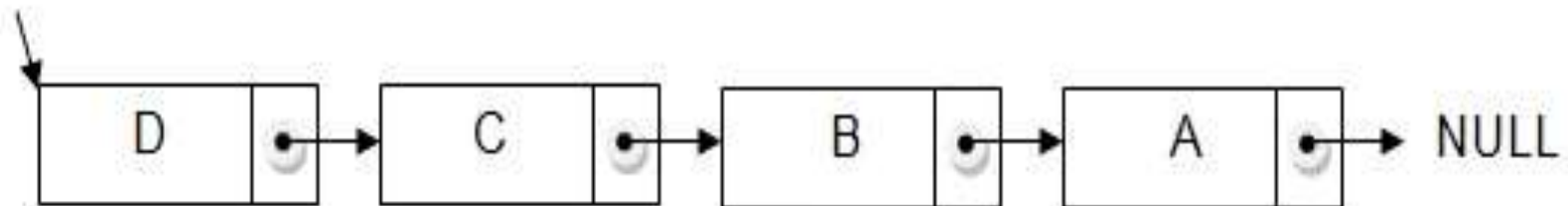
```
void search(int x)
{
    node*temp=start;
    while(temp!=NULL)
    {
        if(temp->data==x)
        {
            cout<<"FOUND "<<temp->data;
            break;
        }
        temp=temp->next;
    }
}
```

Reversing a linked list

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

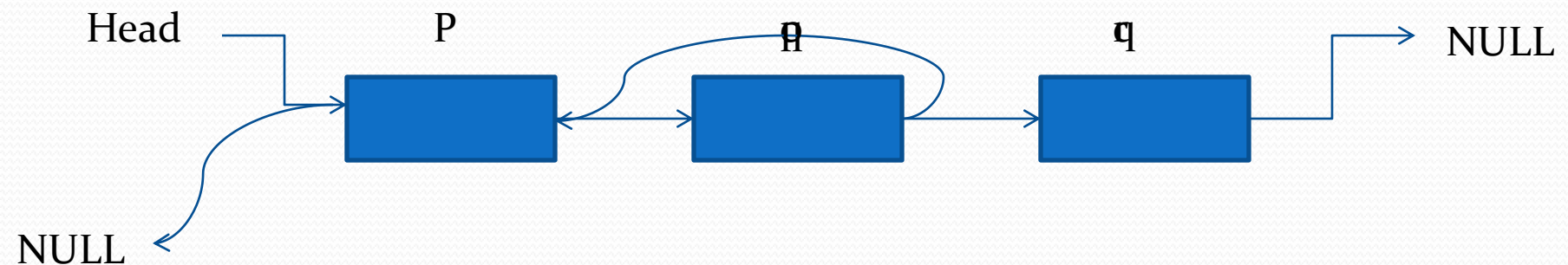


Input



Output

- 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

Code

```
void reverse()
{
    node *p,*q,*r;
    if(start==NULL)
    {
        cout<<"\nList is empty\n";
        return;
    }
    p=start;
    q=p->link;
    p->link=NULL;
    while(q!=NULL)
    {
        r=q->link;
        q->link=p;
        p=q;
        q=r;
    }
    start=p;
    cout<<"\nReversed successfully";
}
```

COMPLEXITY OF VARIOUS OPERATIONS IN ARRAYS AND SLL

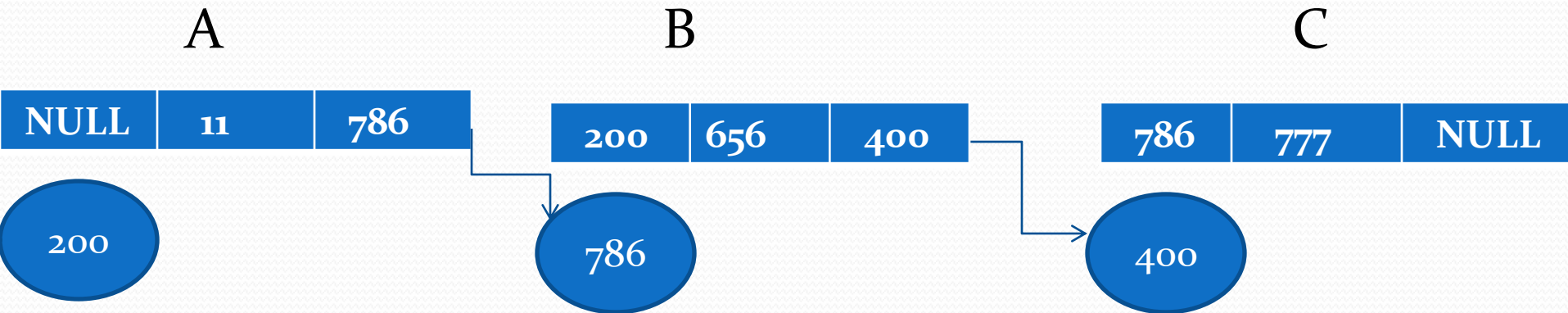
Operation	ID-Array Complexity	Singly-linked list Complexity
Insert at beginning	$O(n)$	$O(1)$
Insert at end	$O(1)$	$O(1)$ if the list has tail reference $O(n)$ if the list has no tail reference
Insert at middle	$O(n)$	$O(n)$
Delete at beginning	$O(n)$	$O(1)$
Delete at end	$O(1)$	$O(n)$
Delete at middle	$O(n)$: $O(1)$ access followed by $O(n)$ shift	$O(n)$: $O(n)$ search, followed by $O(1)$ delete
Search	$O(n)$ linear search $O(\log n)$ Binary search	$O(n)$
Indexing: What is the element at a given position k ?	$O(1)$	$O(n)$

Doubly Linked List

1. **Doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes.
2. Each node contains three fields ::
 - : one is data part which contain data only.
 - :two other field is links part that are point or references to the previous or to the next node in the sequence of nodes.
3. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null to facilitate traversal of the list.

NODE

previous	data	next
----------	------	------



A doubly linked list contains three fields: an integer value, the link to the next node, and the link to the previous node.

DLL's compared to SLL's

- Advantages:

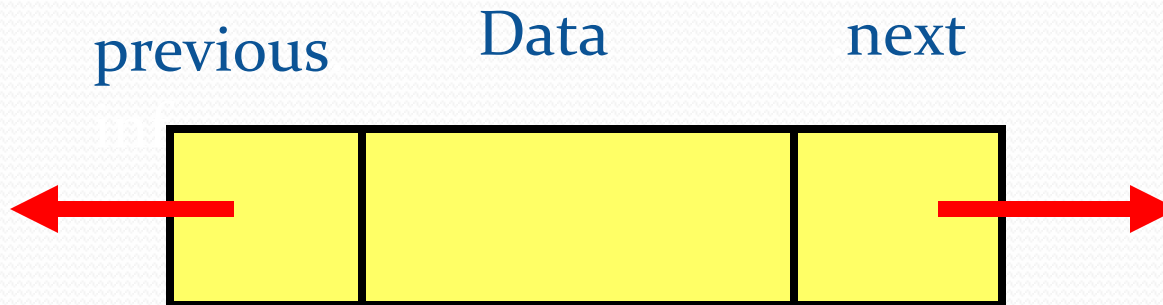
- Can be traversed in either direction (may be essential for some programs)
- Some operations, such as deletion and inserting before a node, become easier

- Disadvantages:

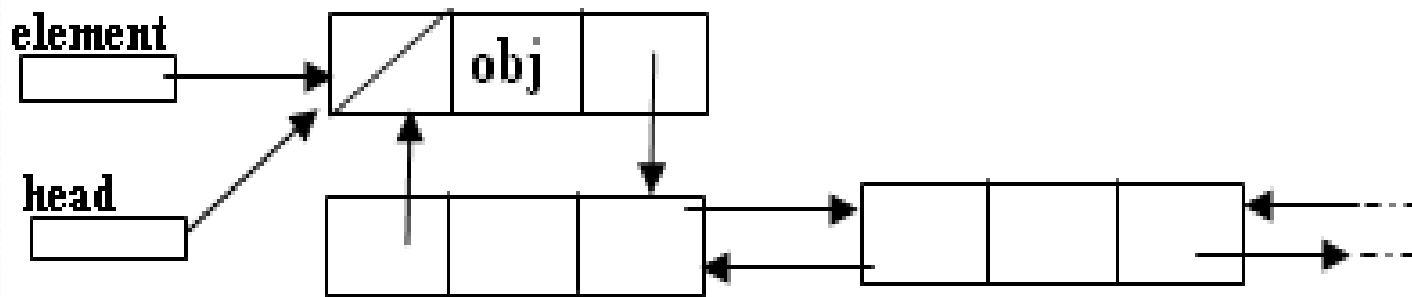
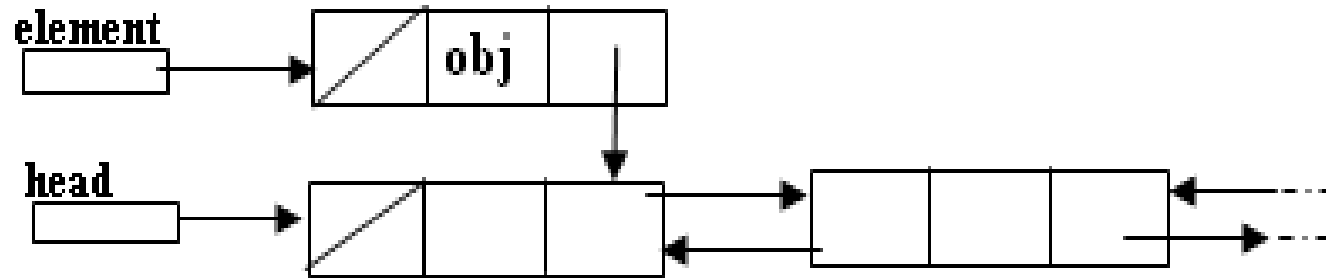
- Requires more space
- List manipulations are slower (because more links must be changed)
- Greater chance of having bugs (because more links must be manipulated)

Structure of DLL

```
struct node
{
    int data;
    node*next;
    node*previous; //holds the address of previous node
};
```

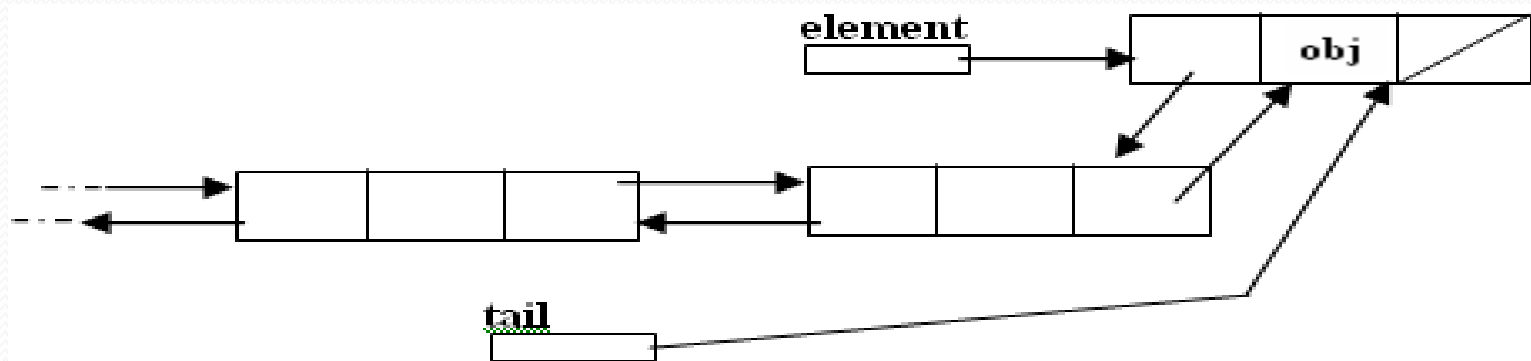
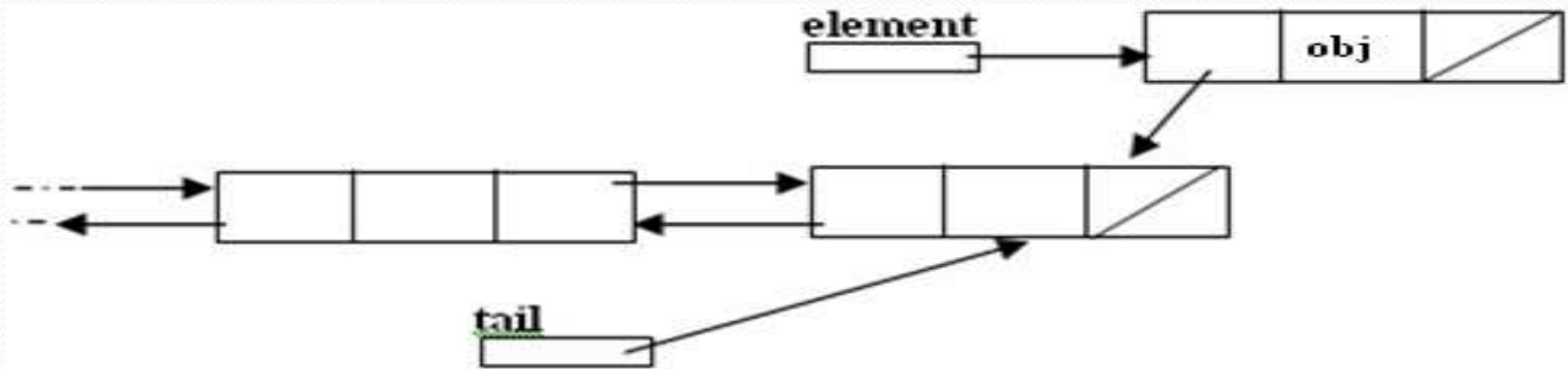


Inserting at beginning



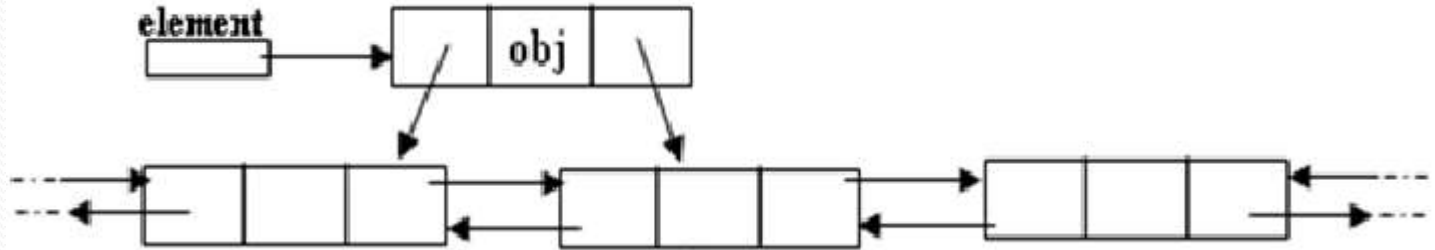
```
void insert_beg(node *p)
{
    if(start==NULL)
    {
        start=p;
        cout<<"\nNode inserted successfully at the beginning\nm";
    }
    else
    {
        node* temp=start;
        start=p;
        temp->previous=p;    //making 1st node's previous point to the
                             new node
        p->next=temp;        //making next of the new node point to the
                             1st node
        cout<<"\nNode inserted successfully at the beginning\n";
    }
}
```

Inserting at the end

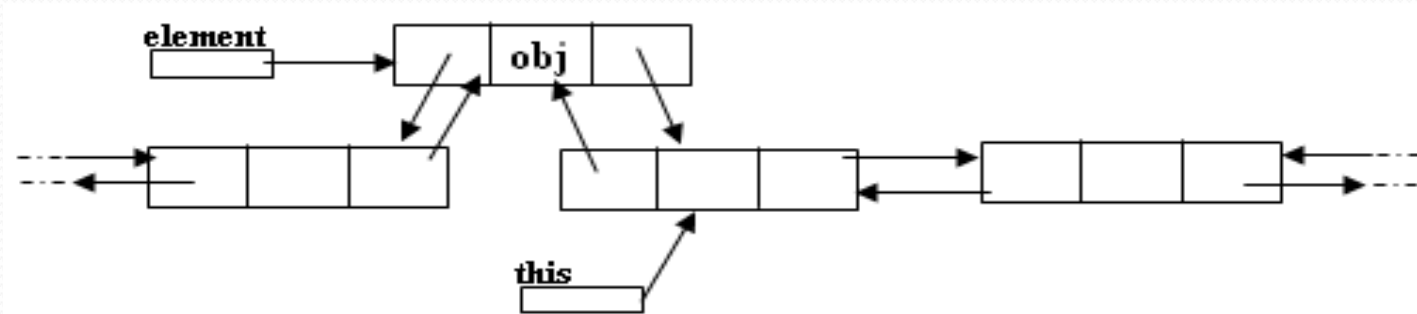


```
void insert_end(node* p)
{
    if(start==NULL)
    {
        start=p;
        cout<<"\nNode inserted successfully at the end";
    }
    else
    {
        node* temp=start;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=p;
        p->previous=temp;
        cout<<"\nNode inserted successfully at the end\n";
    }
}
```

Inserting after a node



Making next and previous pointer of the node to be inserted point accordingly

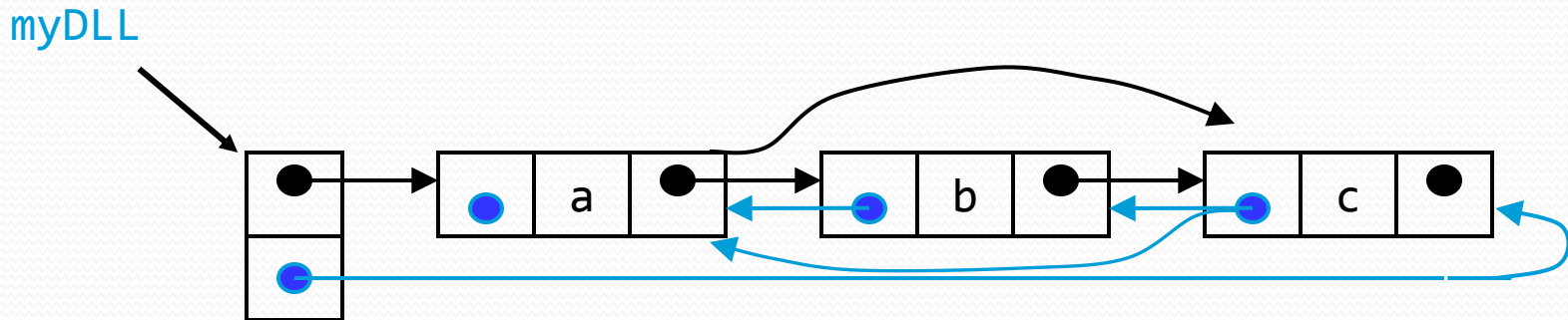


Adjusting the next and previous pointers of the nodes b/w which the new node accordingly


```
void insert_after(int c,node* p)
{
    temp=start;
    for(int i=1;i<c-1;i++)
    {
        temp=temp->next;
    }
    p->next=temp->next;
    temp->next->previous=p;
    temp->next=p;
    p->previous=temp;
    cout<<"\nInserted successfully";
}
```

Deleting a node

- Node deletion from a DLL involves changing *two* links
- In this example, we will delete node b



- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

```
void del_at(int c)
```

```
{
```

```
    node*s=start;
```

```
    {
```

```
    for(int i=1;i<c-1;i++)
```

```
    {
```

```
        s=s->next;
```

```
    }
```

```
    node* p=s->next;
```

```
    s->next=p->next;
```

```
    p->next->previous=s;
```

```
    delete p;
```

```
    cout<<"\nNode number "<<c<<" deleted successfully";
```

```
    }
```

```
}
```

```
}
```

THANK YOU