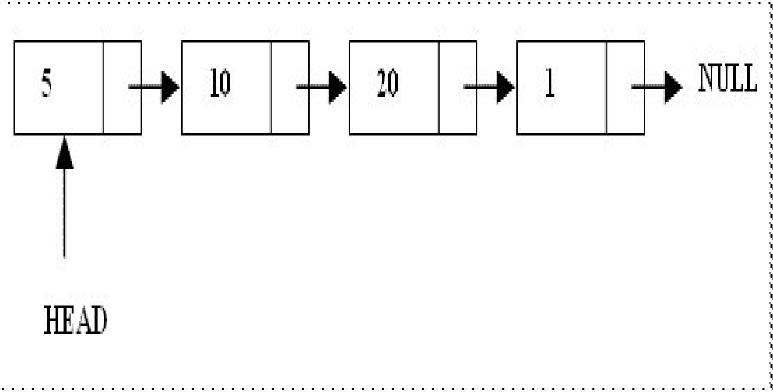
CSE225: Data Structure And Algorithms

Lecture-09: Link List

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 (according 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

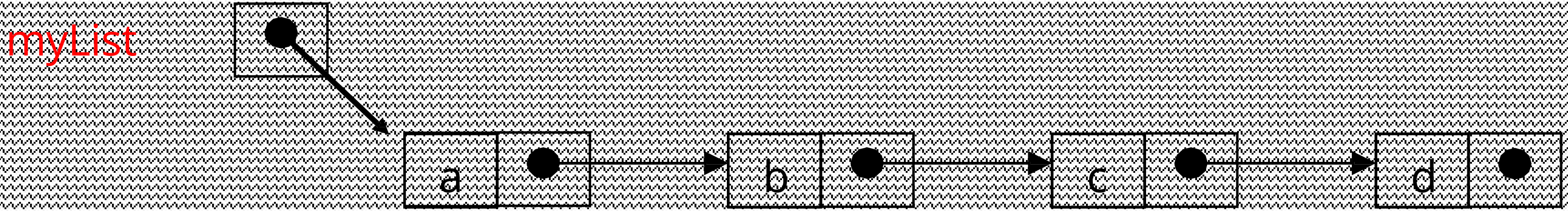
- Doubly linked list

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

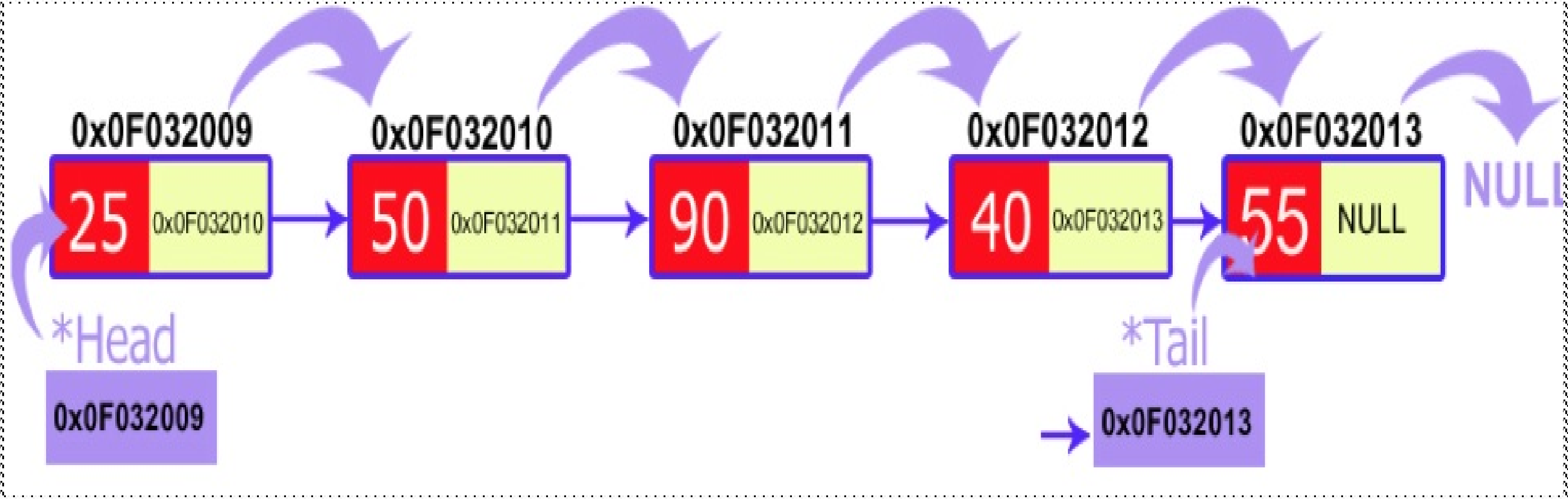
Schematic representation

- 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


0x0F032009
0x0F032010
0x0F032011
0x0F032012
0x0F032013
25
0x0F032010
50
0x0F032011
90
0x0F032012
40
0x0F032013
55
NULL
NULL
*Head
0x0F032009
*Tail
0x0F032013

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

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

Insertion Description

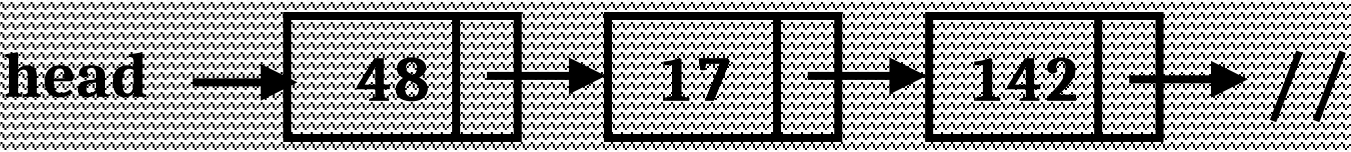
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion at the beginning

Steps:

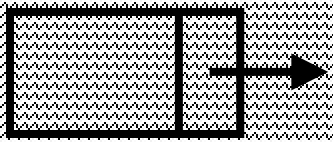
- Create a Node
- Set the node data Values
- Connect the pointers

Insertion Description

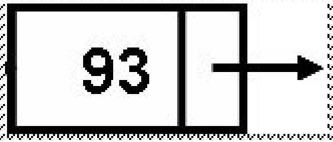


- Follow the previous steps and we get

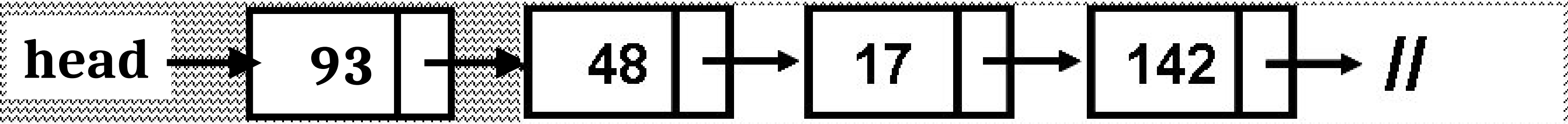
Step 1



Step 2



Step 3

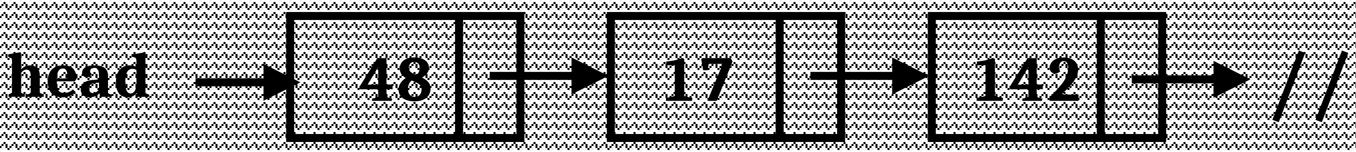


Insertion at the end

Steps:

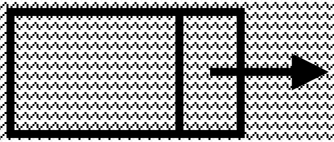
- Create a Node
- Set the node data Values
- Connect the pointers

Insertion Description

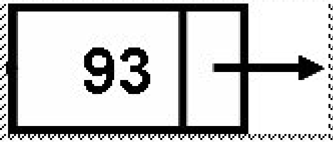


- Follow the previous steps and we get

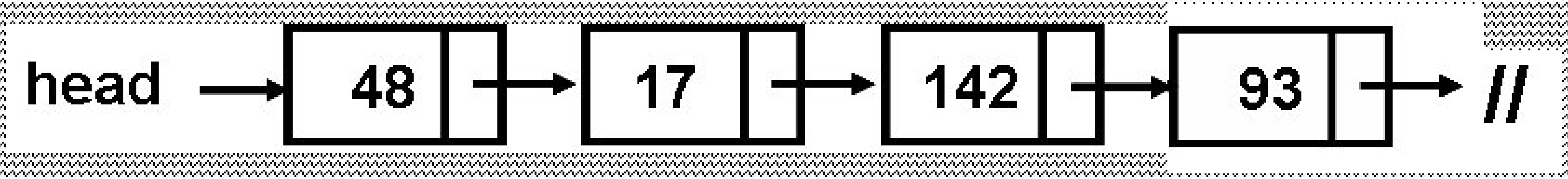
Step 1



Step 2



Step 3

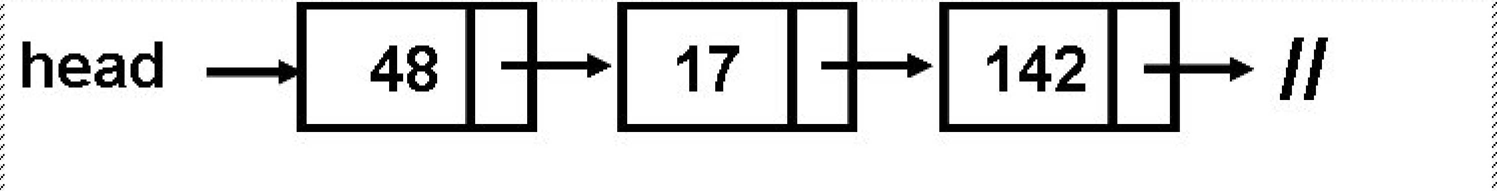


Insertion in the middle

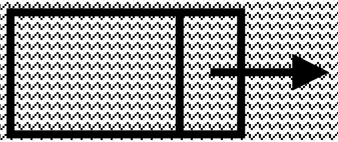
Steps:

- Create a Node
- Set the node data Values
- Break pointer connection
- Re-connect the pointers

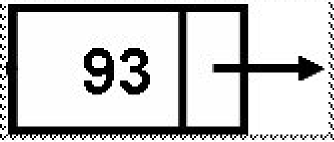
Insertion Description



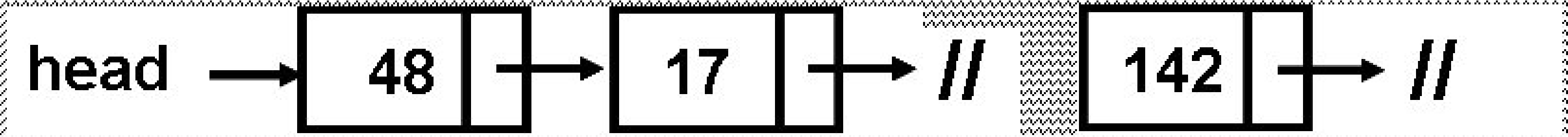
Step 1



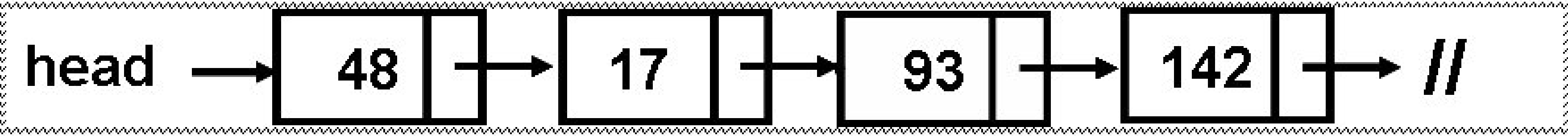
Step 2



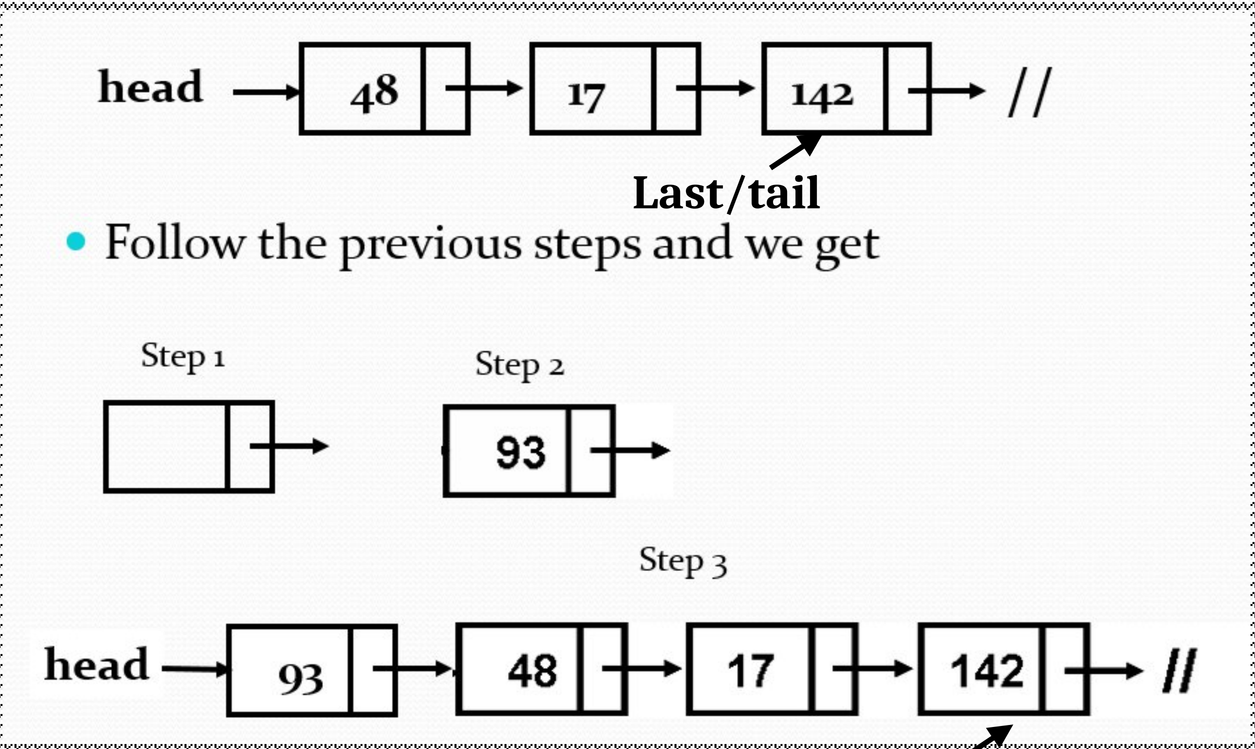
Step 3



Step 4



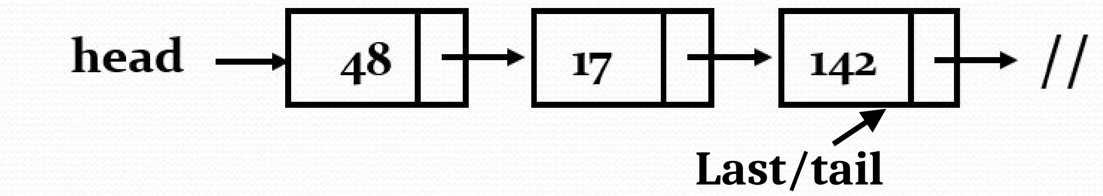
```
switch(ch)
  {
  case 1:
      temp->next=head;
      head=temp;          break;
```



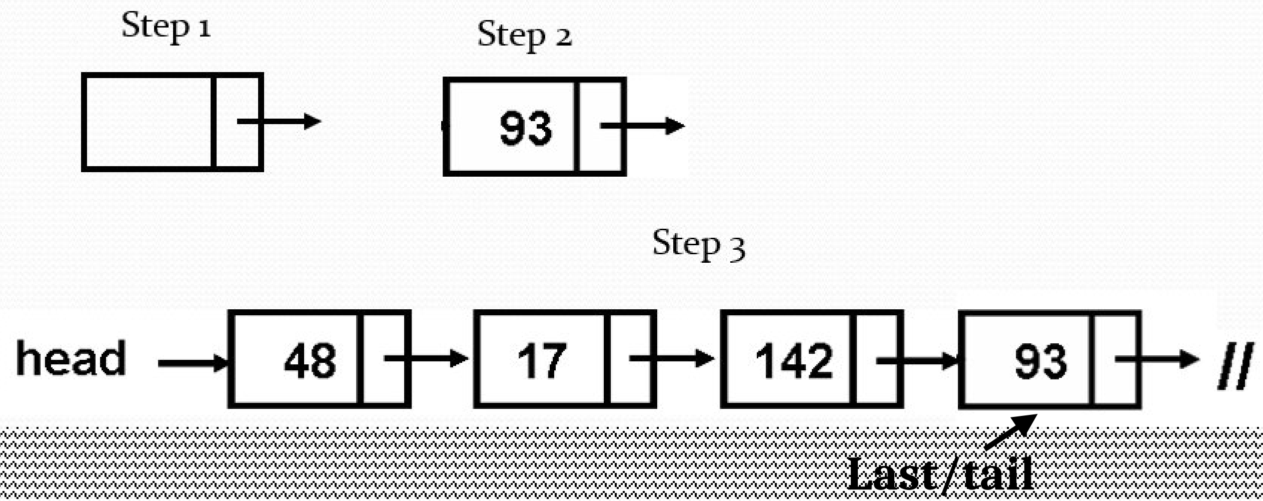
- Follow the previous steps and we get

Last/tail

```
switch(ch)
 {
 case 1:
    temp->next=head;
    head=temp;          break;
 case 2:
    last->next=temp;
    last=temp;     break;
```

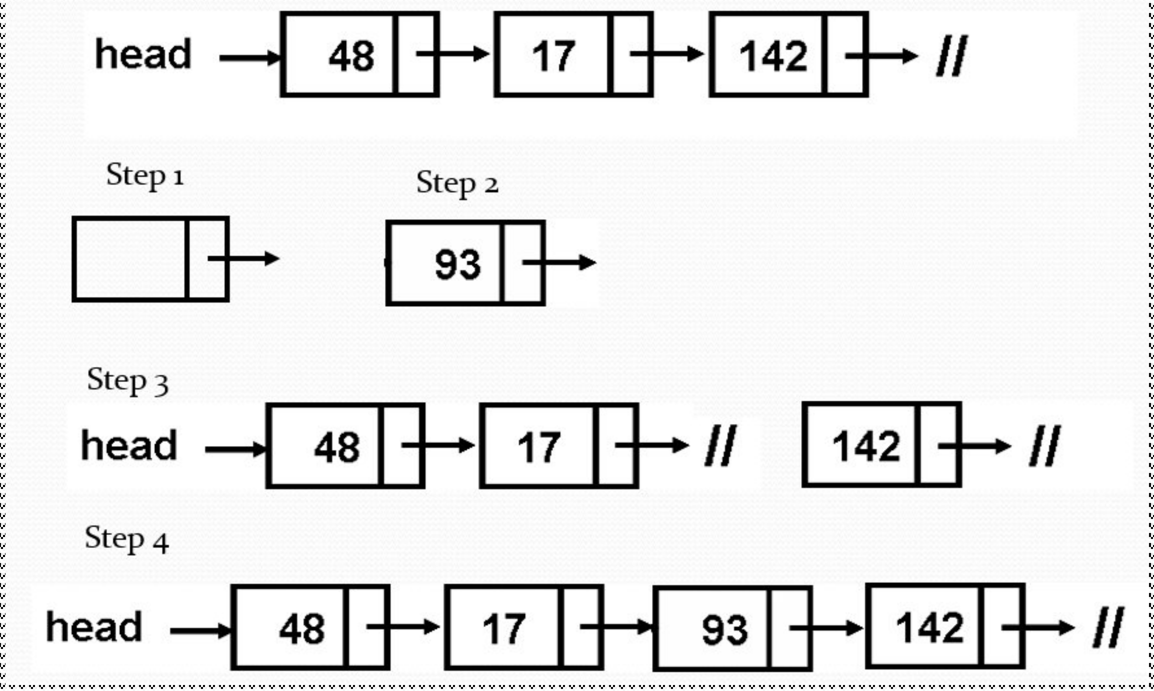


- Follow the previous steps and we get



```
switch(ch)
  {
case 3:
    cout<<"\nEnter the Position to
  Insert:";
    cin>>pos;
```

```
while(count!=pos)
{
   prev=cur;
   cur=cur->next;
   count++;
 }
   if(count==pos)
   {
      prev->next=temp;
      temp->next=cur;
   }
   else
cout<<"\nNot Able to Insert";
   break;
 }
```



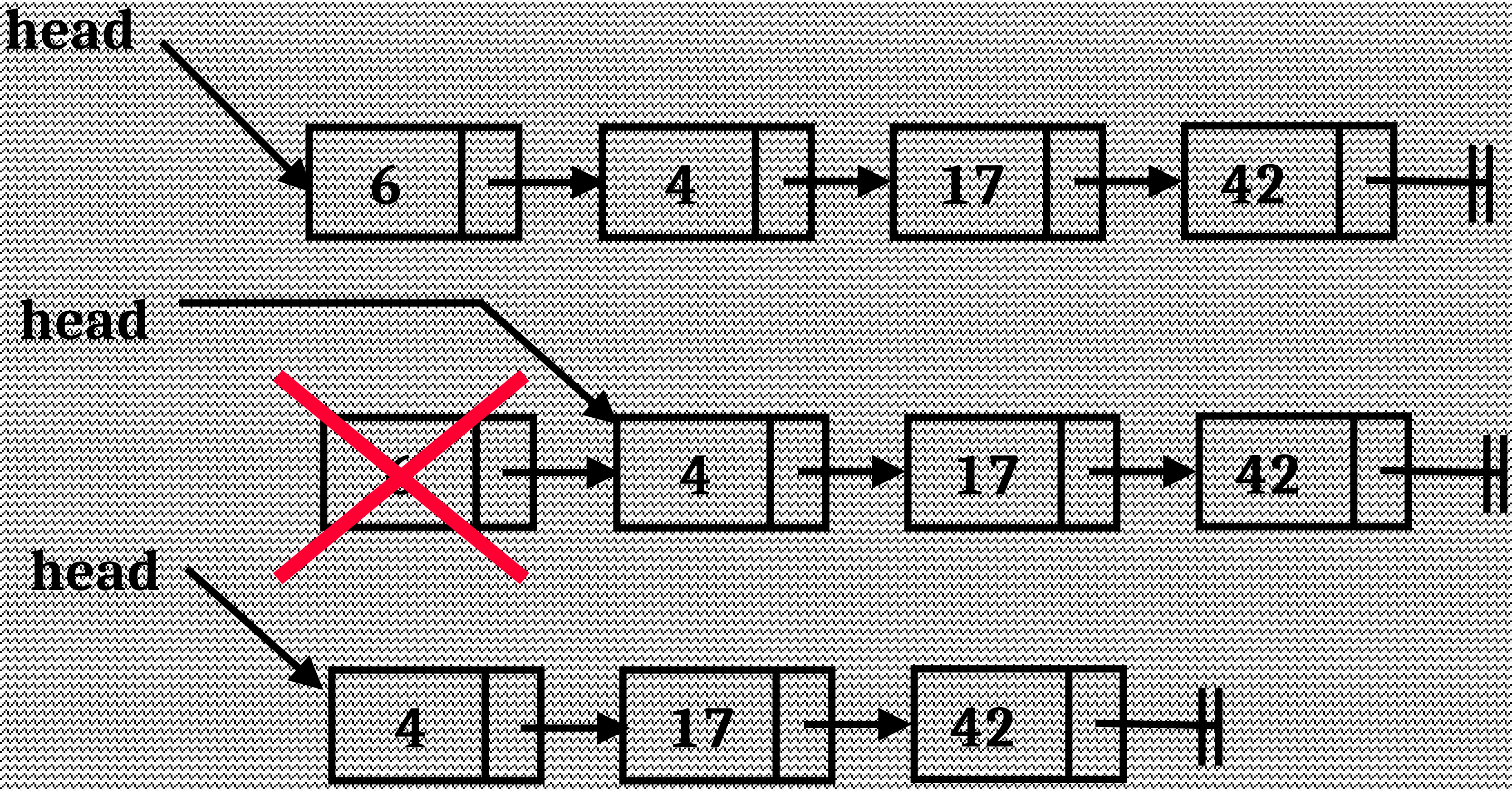
```
switch(ch)
 {
 case 1:
    temp->next=first;
    first=temp;          break;
 case 2:
    last->next=temp;
    last=temp;      break;
 case 3:
    cout<<"\nEnter the Position to
 Insert:";
    cin>>pos;
```

```
while(count!=pos)
      {
   prev=cur;
    cur=cur->next;
    count++;
      }
      if(count==pos)
      {
          prev->next=temp;
          temp->next=cur;
      }
      else
 cout<<"\nNot Able to Insert";
     break;
   }
}
```

Deletion Description

- Deleting from the beginning of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deletion Description



Deleting from the **beginning**

Steps

- Break the pointer connection and assign to temp node
- Re-connect the nodes change the starting node to next
- Delete the node

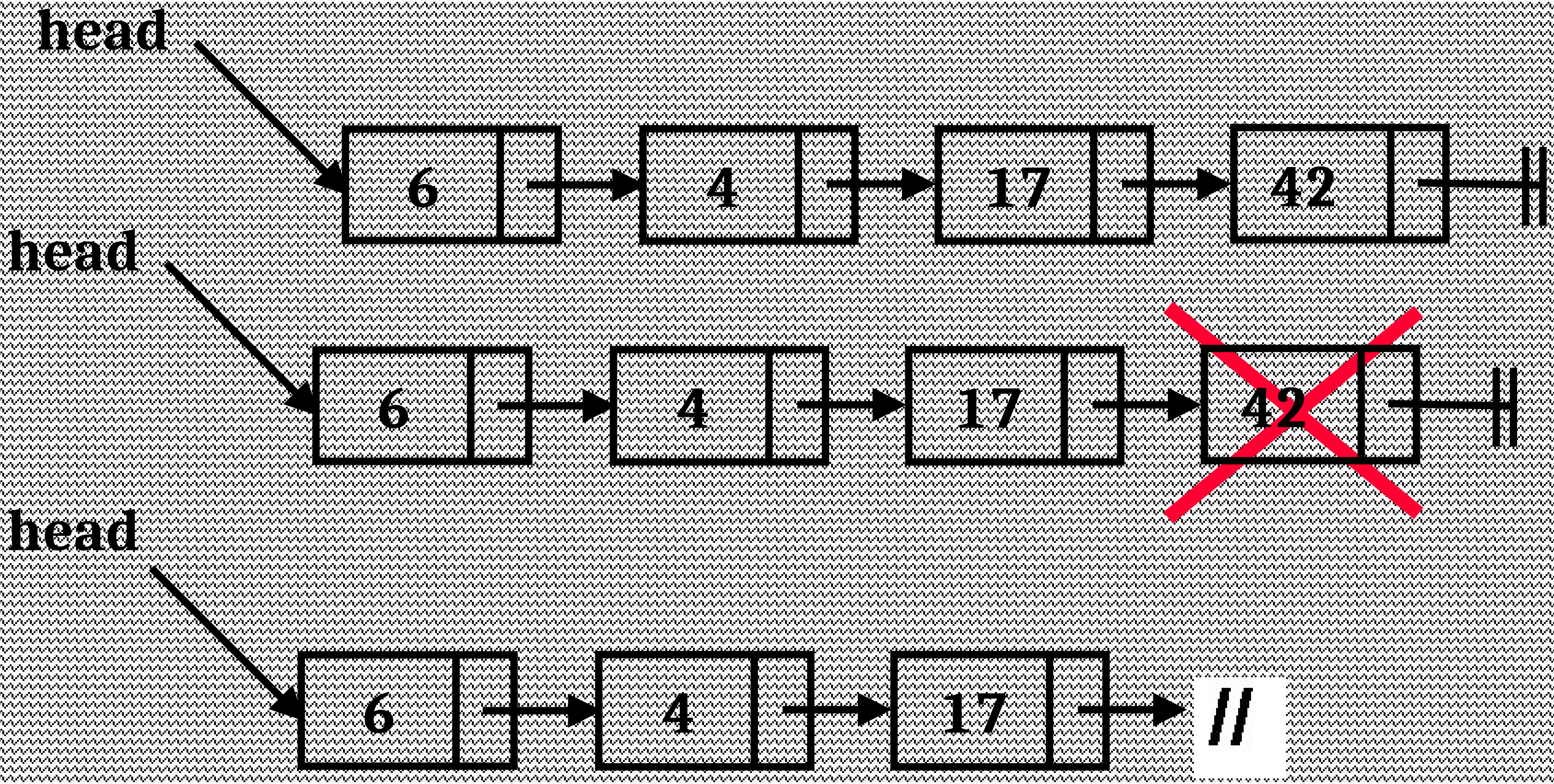
```
void delete_first()
{
    node *temp=new node;
    temp=head;
    head=head->next;
    delete temp;
}
```

Deleting from the end

Steps

- Break the pointer connection
- Set previous node pointer to NULL
- Delete the node

Deletion Description

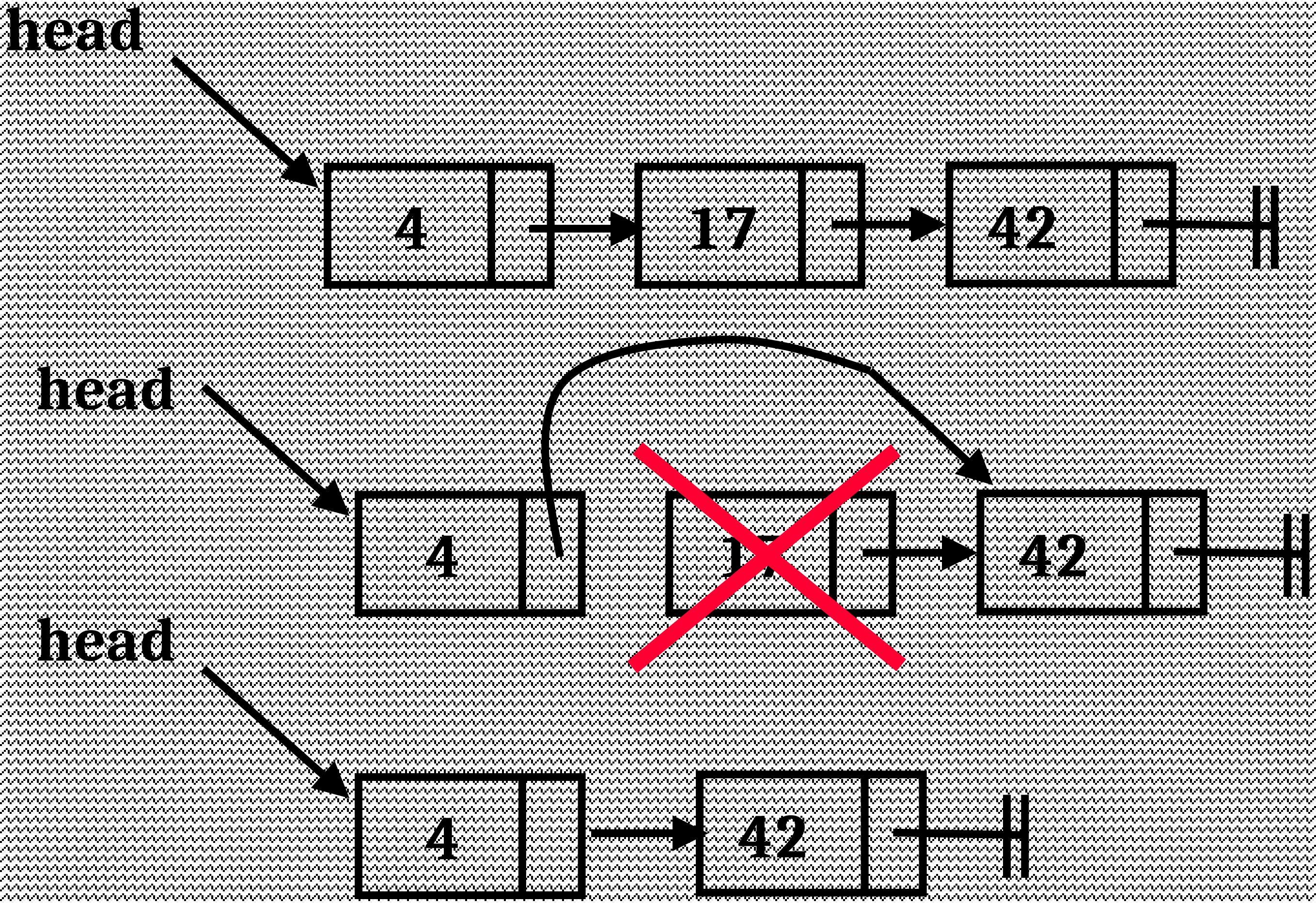


Deleting from the Middle

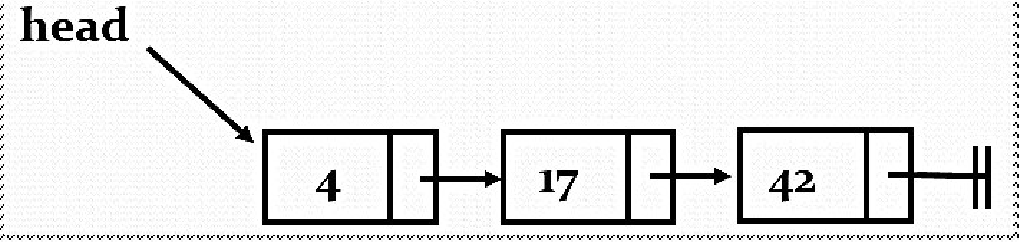
Steps

- Set previous Node pointer to next node
- Break Node pointer connection
- Delete the node

Deletion Description

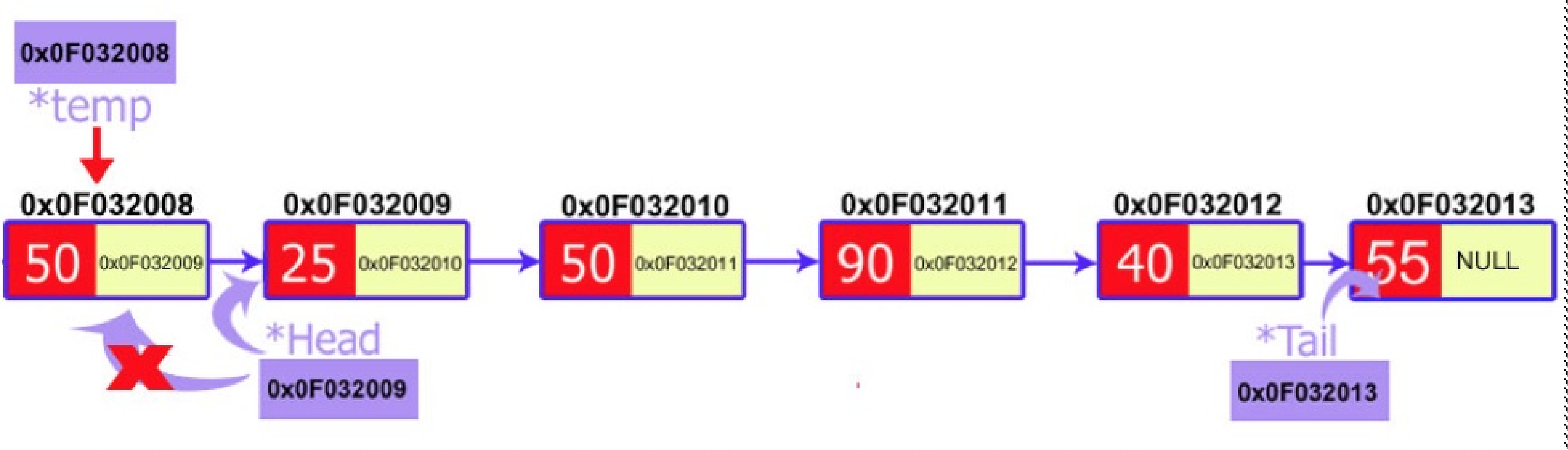


Display the list



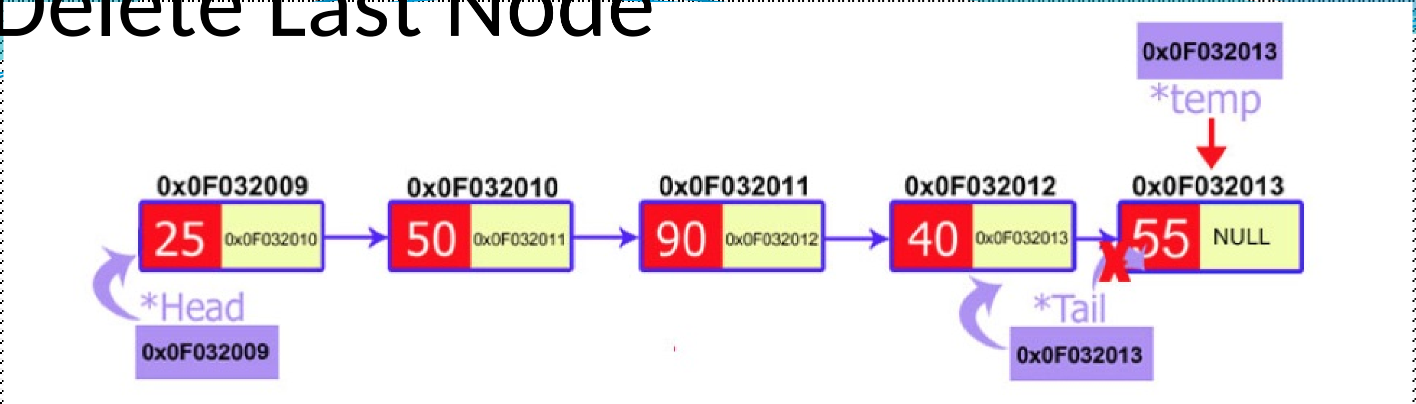
```
void display()
{
  node *temp=new node;
  temp=head;
  while(temp!=NULL)
  {
     cout<<temp->data<<"\t";
     temp=temp->next;
  }
}
```

Delete First Node



```
void delete_first()
{
   node *temp=new node;
   temp=head;
   head=head->next;
   delete temp;
}
```

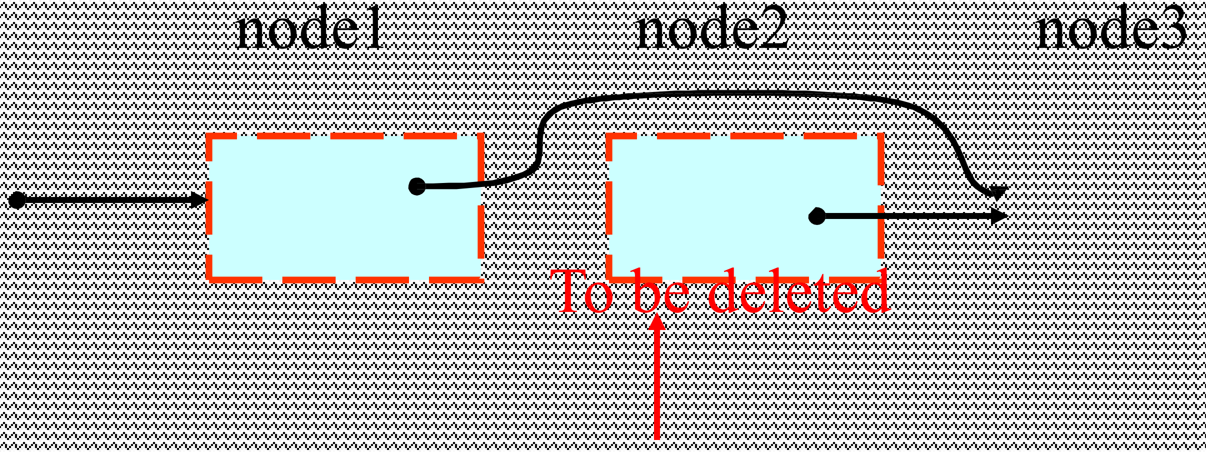
Delete Last Node

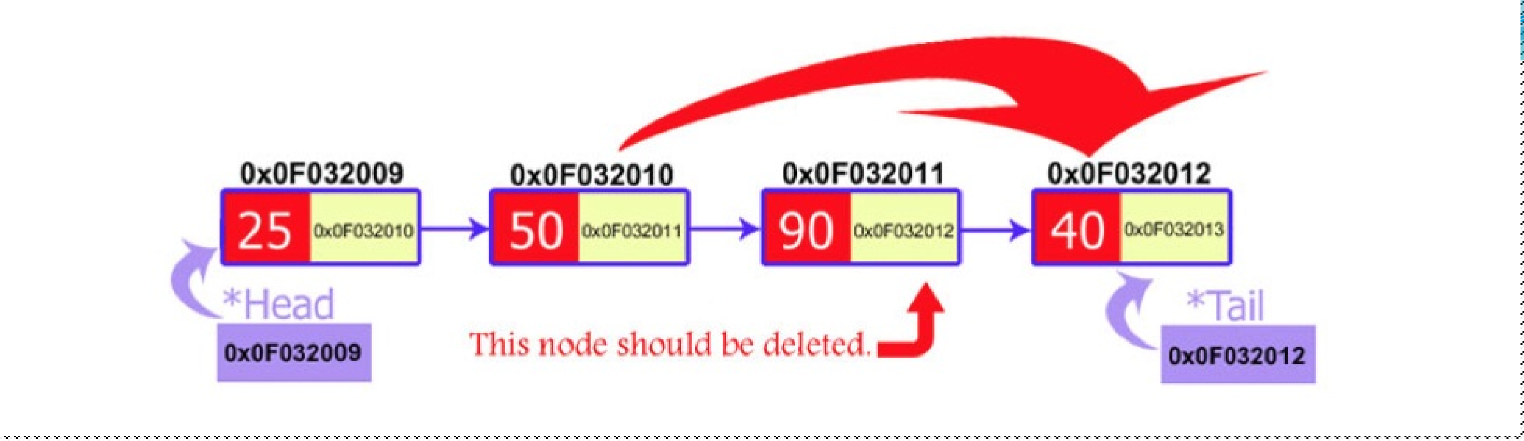


```
void delete_last()
 {
     node *current=new node;
     node *previous=new node;
     current=head;

    while(current->next!=NULL)
    {
       previous=current;
       current=current->next;
     }
}
      tail=previous;
      previous->next=NULL;
      delete current;
    }
```

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





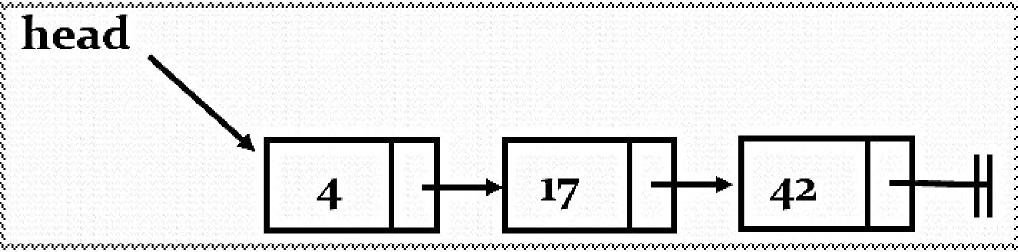
```
void delete_position(int pos)
 {
    node *current=new node;
    node *previous=new node;
    current=head;
    for(int i=1;i<pos;i++)
    {
       previous=current;
       current=current->next;
     }
     previous->next=current->next;
     delete current;
}
```

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 traversedtill the data in

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



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)

Insertion at the beginning of the Singly linked lists

```
Step 1. Create a new node and assign the address to any node
            say ptr.
Step 2. OVERFLOW,IF(PTR = NULL)
            write : OVERFLOW and EXIT.
Step 3. ASSIGN INFO[PTR] = ITEM
Step 4. IF(START = NULL)
                ASSIGN NEXT[PTR] = NULL
            ELSE
                ASSIGN NEXT[PTR] = START
Step 5. ASSIGN START = PTR
Step 6. EXIT
```

Search an element in a Linked List

Iterative Solution

bool search(Node *head, int x)

- 1) Initialize a node pointer, current = head.
- 2) Do following while current is not NULL
    - a) current->key is equal to the key being searched
        - return true.
    - b) current = current->next
- 2) Return false

Recursive Solution

- **bool search(head, x)**
- 1) If head is NULL, return false.
- 2) If head's key is same as x, return true;

  Else return search(head->next, x)

Algorithm for Deletion at beginning of the list

```
DELETE AT BEG(INFO,NEXT,START)
1.IF(START=NULL)
2.ASSIGN PTR = STRAT
3.ASSIGN TEMP = INFO[PTR]
4.ASSIGN START = NEXT[PTR]
5.FREE(PTR)
6.RETURN(TEMP)
```

Doubly Linked List

1. **Doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes.

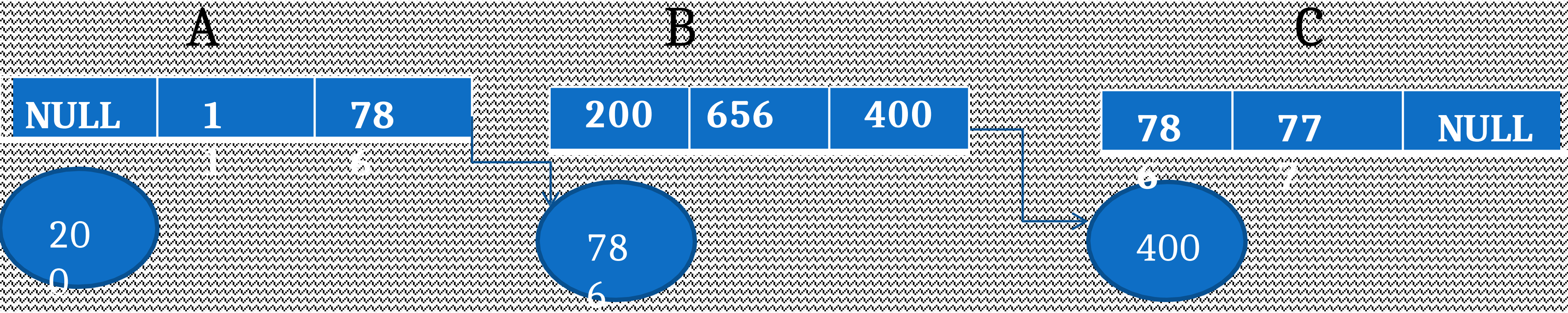
2 . **Each node contains three fields** ::

- ➢ 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 contain **three fields:** an integer value, the link to the next node, and the link to the previous node.

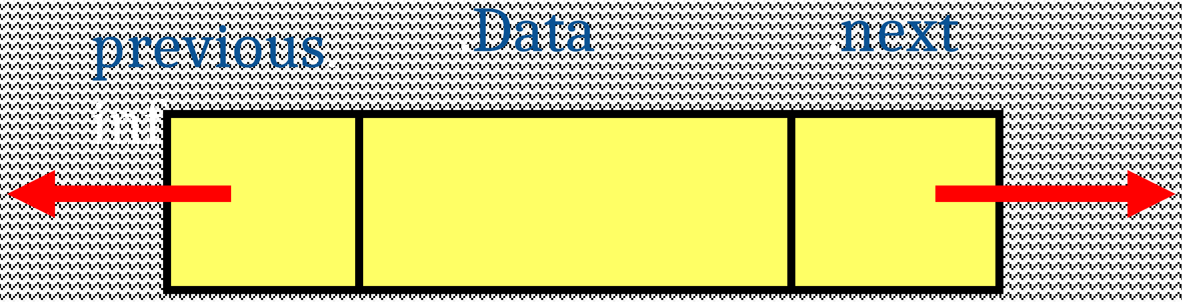
DLL's compared to SLL's

- 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

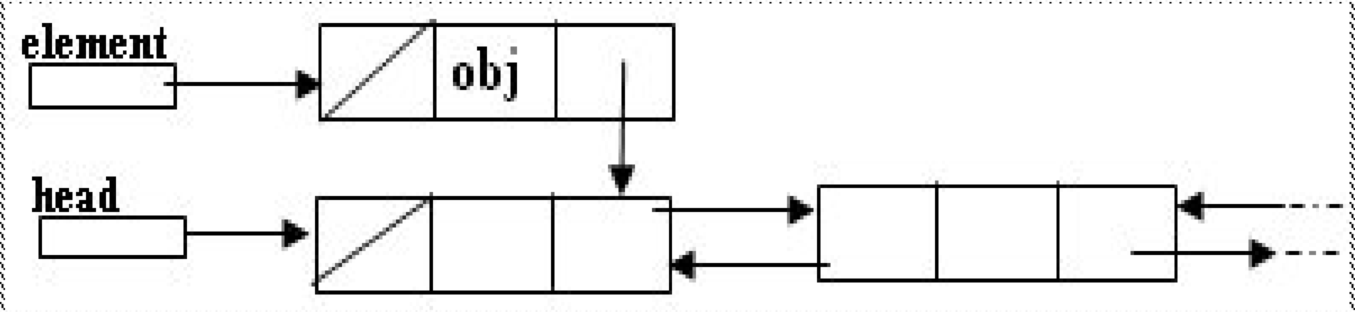
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

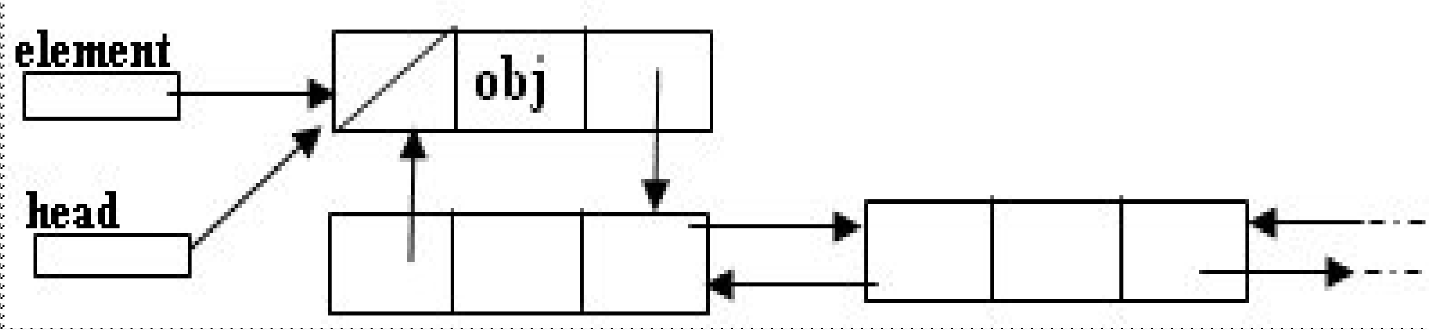
Structure of DLL

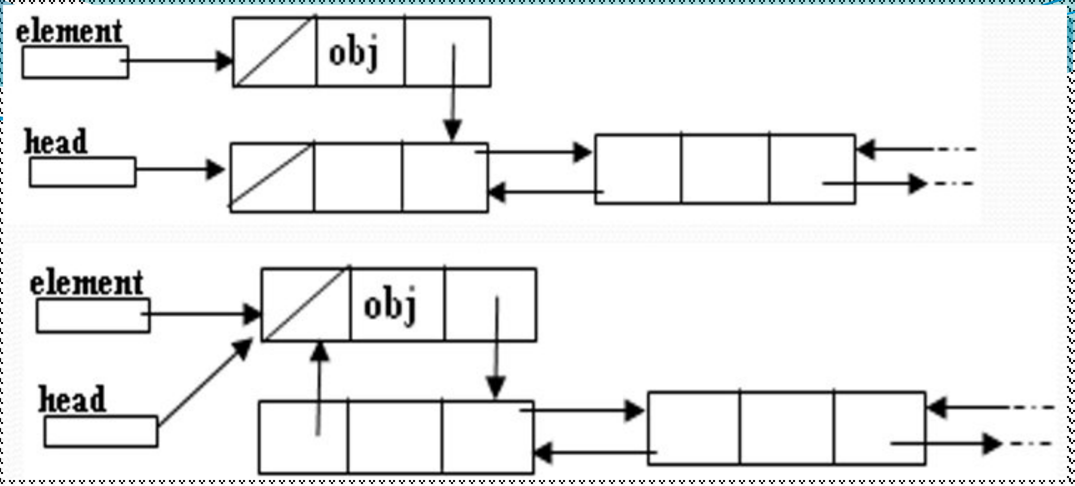
```
struct node
{
   int data;
   node*next;
   node*previous;   //holds the address of previous node
};
```



Inserting at beginning

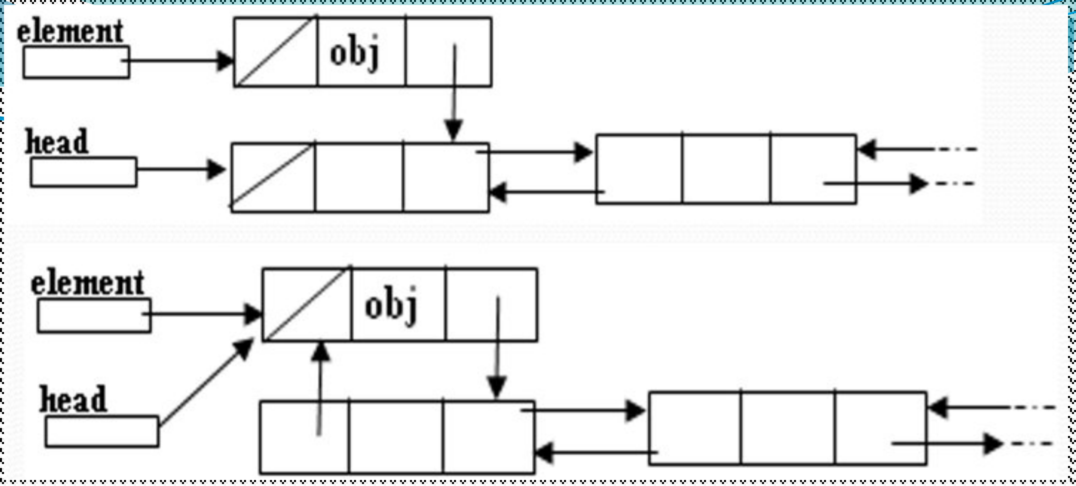






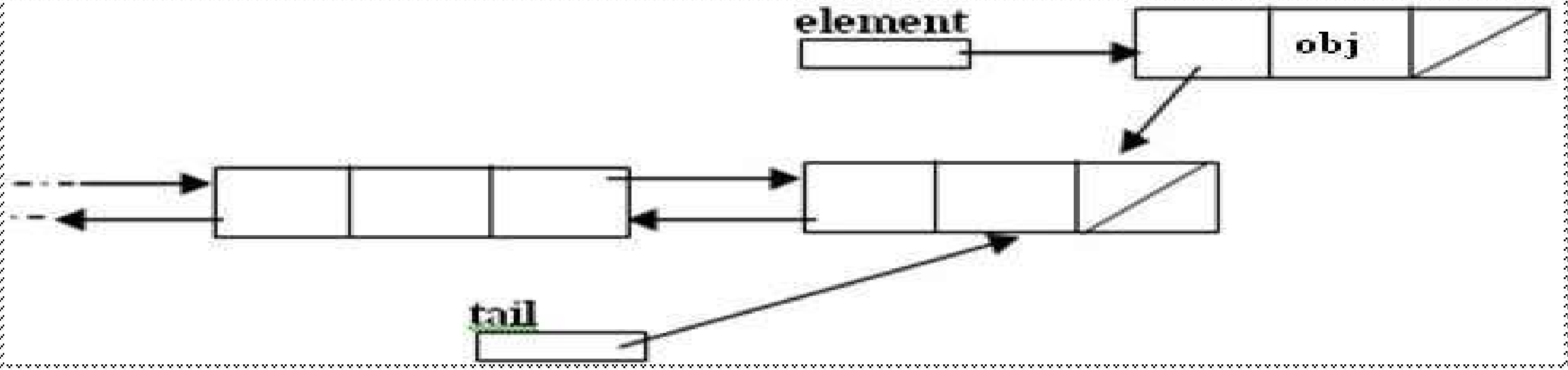
```
void insert_beg(node *p)
{
  if(start==NULL)
  {
   start=tail=p;
   cout<<"\nNode inserted successfully at the beginning\m";
  }
  else
  {
  node* temp=start;
  start=p;    //making the new node to become the head/start of the list
  temp->previous=p;  //making 1st  nodes previous point to the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

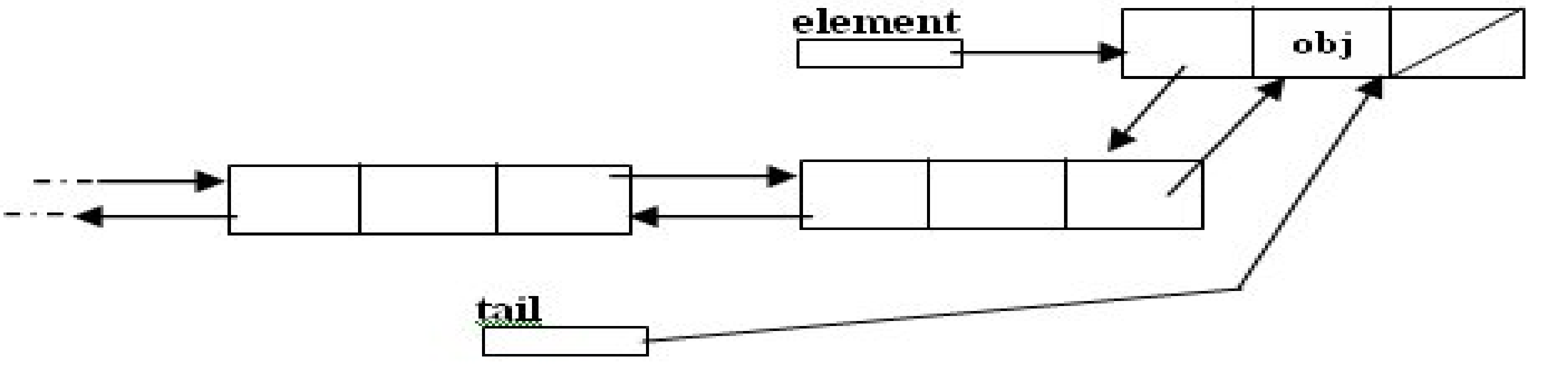
Better Version



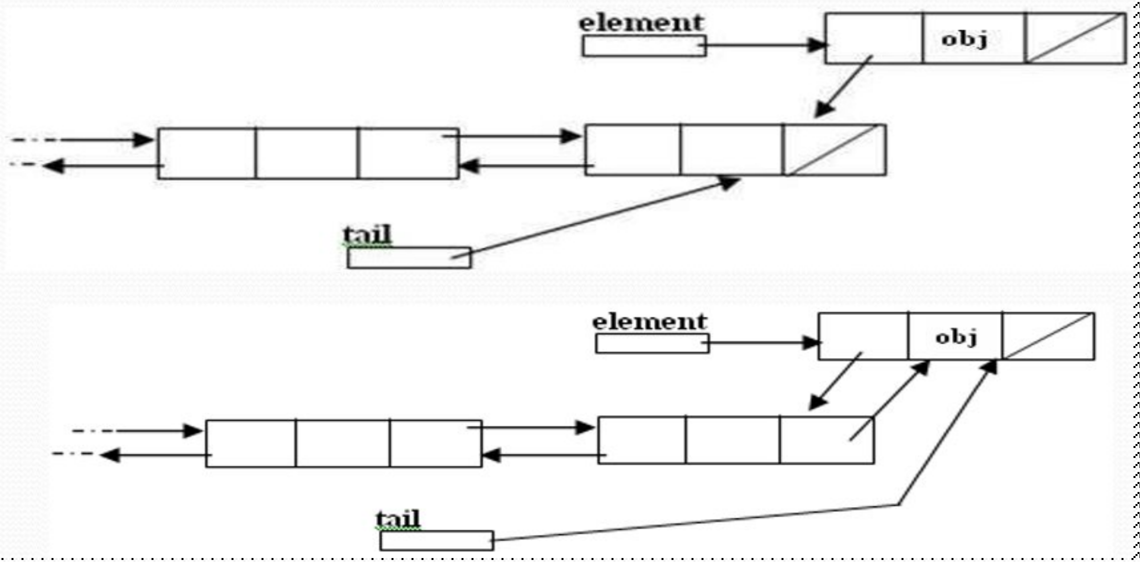
```
void insert_beg(node *p)
{
  if(start==NULL)
  {
   start=tail=p;
   cout<<"\nNode inserted successfully at the beginning\m";
  }
  else
  {
  p->next=start;   //making next of the new node point to the 1st node
  start->previous=p;   //making 1st  nodes previous point to the new node
  start = p;    //making the new node to become the head/start of the list

  cout<<"\nNode inserted successfully at the beginning\n";
  }
```

Inserting at the end



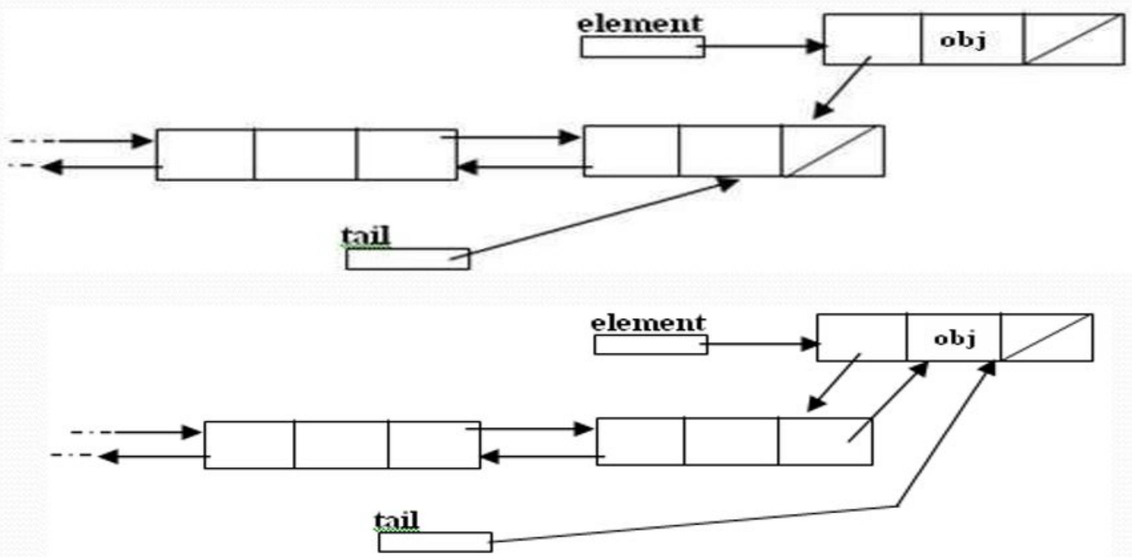


```
void insert_end(node* p)
{
    if(start==NULL)
    {
    start=p;
    cout<<"\nNode inserted successfully at the end";
    }
    else
    {
     node* temp=start;
      while(temp->next!=NULL)
      {
      temp=temp->next;
      }

     temp->next=p;
     p->previous=temp;
     cout<<"\nNode inserted successfully at the end\n";
    }
}
```

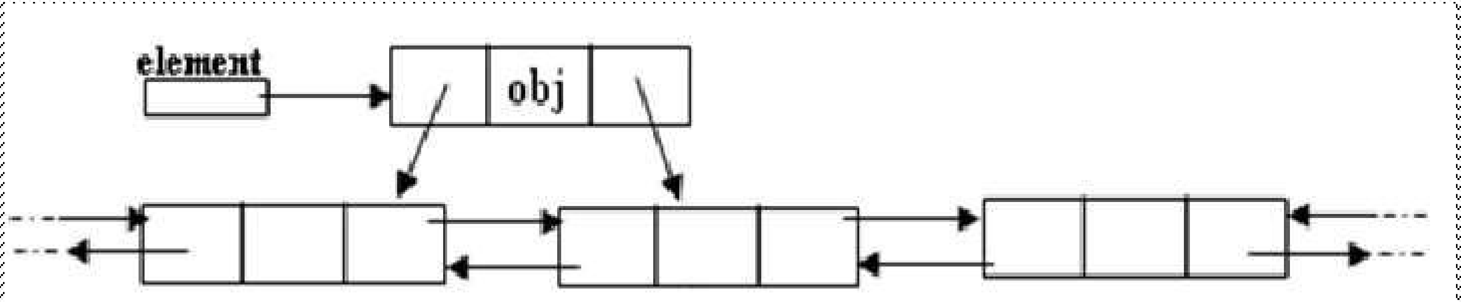


Better Version

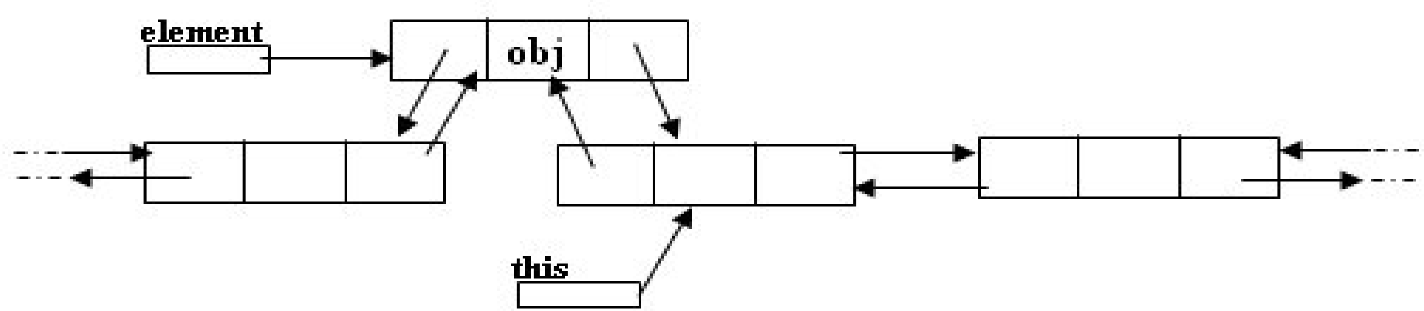
```
void insert_end(node* p)
{
    if(start==NULL)
    {
    start=tail=p;
    cout<<"\nNode inserted successfully at the end";
    }
    else
    {
     p->previous=tail;  //making previous point of the new node to the tail
     tail->next=p;    //making next of the tail node point to the new node
     tail=p;     //making the new node to become the tail of the list
     cout<<"\nNode inserted successfully at the end\n";
    }
}
```



Inserting after a node

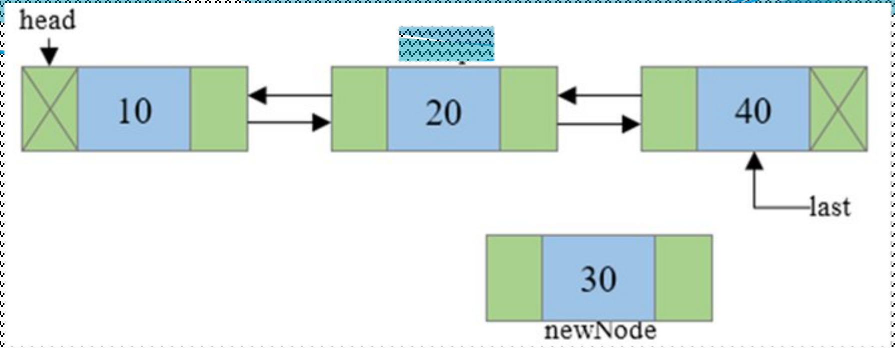


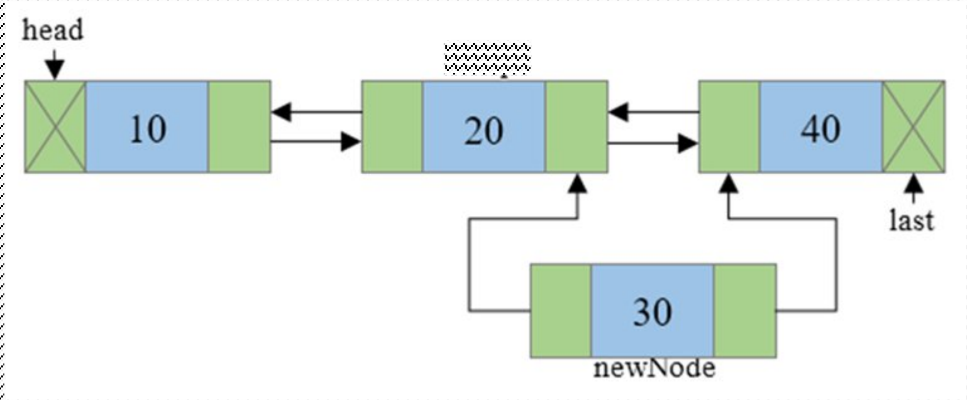
Making next and previous pointer of the node to be inserted point accordingly

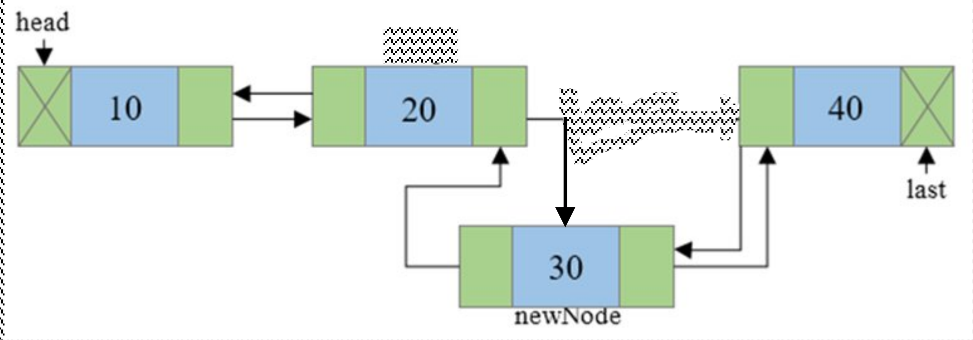


Adjusting the next and previous pointers of the nodes b/w which the new node accordingly

```
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";
}
```

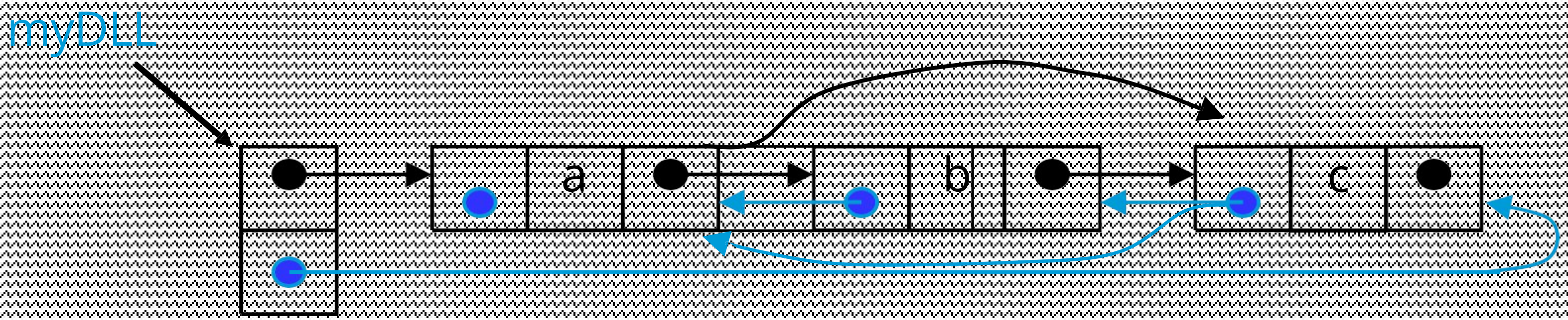






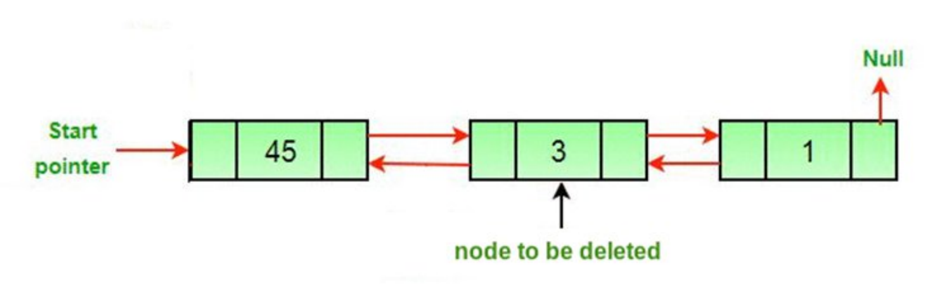
Deleting a node

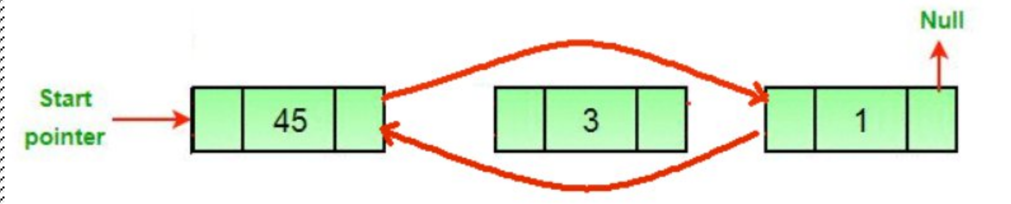
- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



- 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

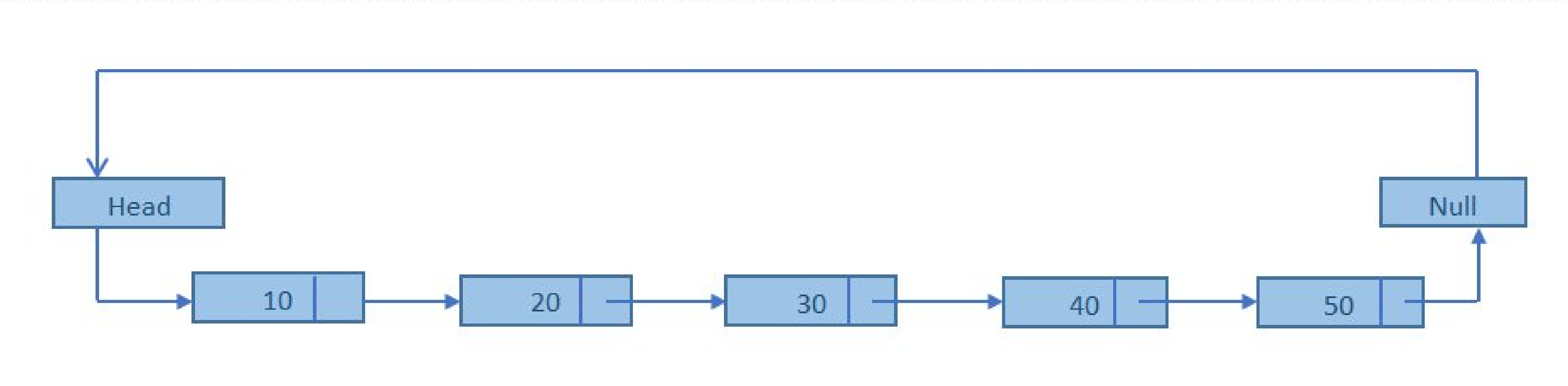
```
void del_at(int c)
{
    node*s=start;

    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";
    }
```





Circular Linked List

1. A circular linked list is a linked list in which the head element's previous pointer points to the tail element and the tail element's next pointer points to the head element.
2. A circularly linked list node looks exactly the same as a linear singly linked list.



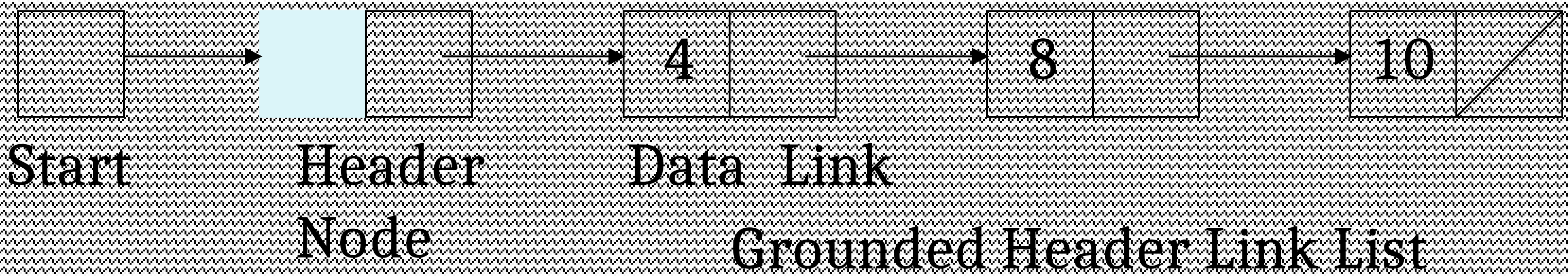
Header Linked Lists

- A Header Linked List always Contains a Special Node called Header Node
- It has Two Types:

a) Grounded Header List
Last Node Contains the NULL Pointer

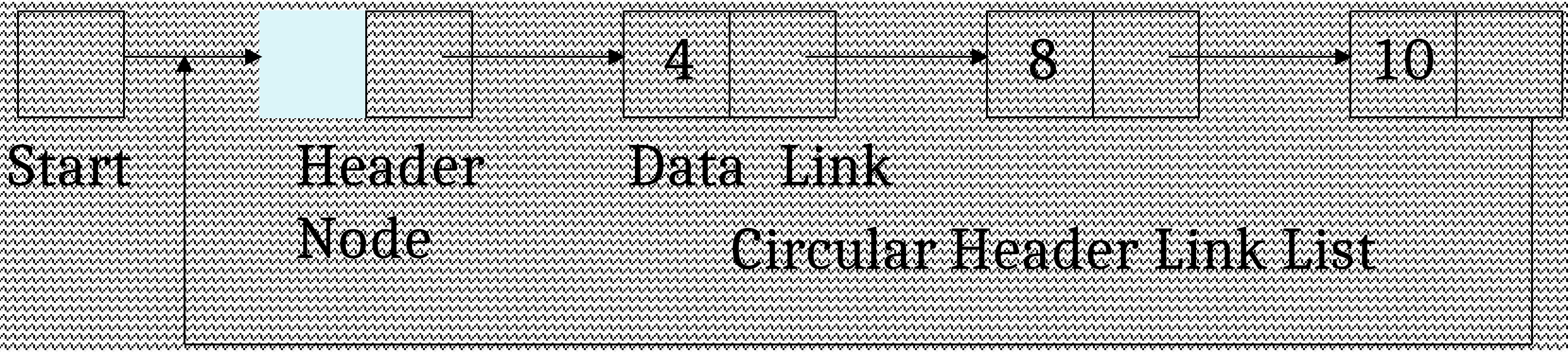
b) Circular Header List
Last Node Points Back to the Header Node

Graphical Representations



Grounded Header Link List

LINK [START] = NULL



Circular Header Link List

LINK [START] = START

APPLICATIONS OF LINKED LIST

1.Applications that have an MRU list (a linked list of file names)

2.The cache in your browser that allows you to hit the BACK button (a linked list of URLs)

3.Undo functionality in Photoshop or Word (a linked list of state)

4.A stack, hash table, and binary tree can be implemented using a doubly linked list.

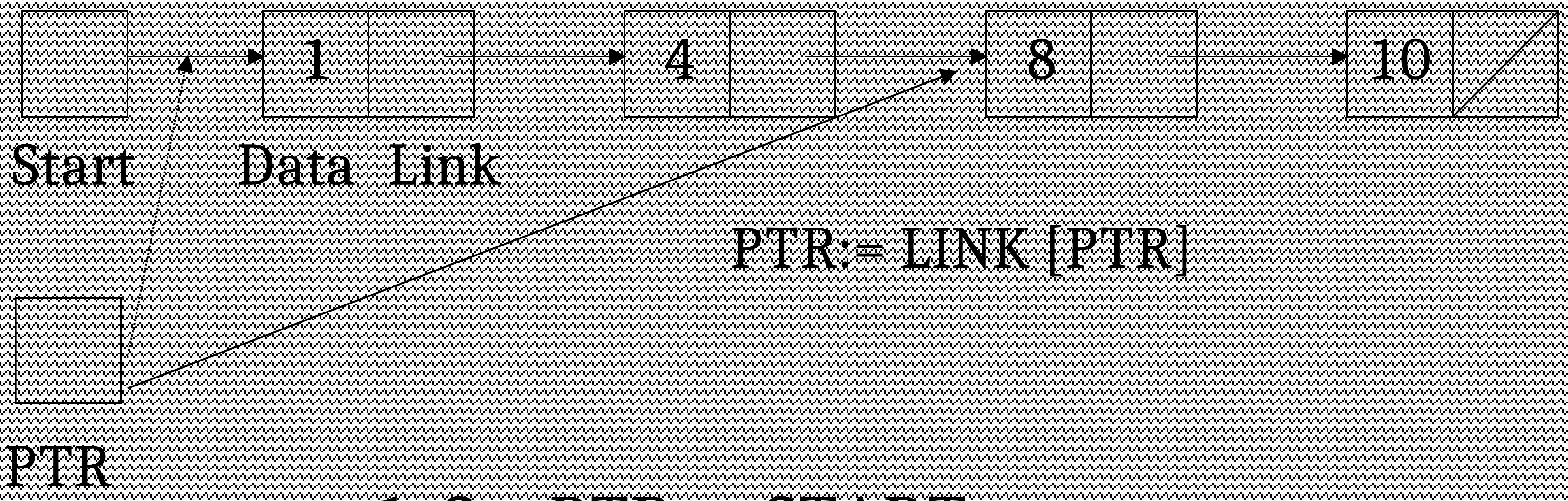
- Advantages of using an Array:
    - only good for non-sparse polynomials.
    - ease of storage and retrieval.
- Disadvantages of using an Array:
    - have to allocate array size ahead of time.
    - huge array size required for sparse polynomials.
    - Waste of space and runtime.

Reading Assignments

Arrays and Linked Lists

- An array is a list store in contiguous memory.
  - Any element of an array can be accessed quickly.
  - Insertion and deletion in the middle of an array requires the movement of many elements.
  - The size of an array is fixed.
- A linked list is a list scattered throughout memory and connected with pointers/Links of next element.
  - The elements of a linked list must be accessed in order. Linear Access Mechanism
  - Insertion and deletion only requires re-assignment of a few pointers.
  - The length of the list can change at any time, making it a dynamic data structure.

Traversing a Linked List



```
1. Set PTR := START
2. Repeat Steps 3-4 while PTR ≠ NULL
3. Apply process to INFO[PTR]
4. Set PTR:= LINK[PTR]
   [end of loop]
5. Exit
```

Searching in a Linked List

- LIST => Link List, LOC => ITEM Location or LOC => NULL

```
Search (LINK, START, ITEM, LOC)
1.  Set PTR := START, LOC:= NULL
2.  Repeat Steps 3-4 while PTR ≠ NULL
3.  If INFO[PTR] = ITEM
      Set LOC:=PTR
      [end of If Structure]
4.  Set PTR:= LINK[PTR]
5.  [end of loop]
6.  Return LOC
```

Insertion in a Linked List

1. Create a new Node and store data in that node.
2. Find the correct position in the list.
3. Assign the pointers to include the new node.

Algorithm: InsertFirst(START, ITEM)

```
1. Set New:= Create Node()
2. Set INFO [New] := ITEM and LINK [New] := NULL
3. Set START := New
4. Exit
```

Insertion In Sorted Linked List

- Steps:
    1. Find the Location of the node
    2. Use Insertion method to insert into the Linked List

1. Find the Location of the node:
    1. Search the Item after which insertion needs to be made.
    2. As the Linked List is sorted searching will result in the location of element => Given ITEM
    3. As Insertion can not be made before a Node therefore another pointer will keep track of the previous node i.e. before moving the PTR , Save := PTR
    4. PTR points to current node Element which is => ITEM while SAVE points to Element before PTR.
    5. Insertion will be made after SAVE

“Find Location in a Sorted Linked List”

Algorithm: FindLoc (START,ITEM,LOC)

```
1.  If START = NULL, then
        Set LOC := NULL and Return
2.  Else if ITEM < INFO [START] [ITEM is not in List]
    Set LOC := NULL and Return
    [ End of if ]
3.  Set SAVE := START and PTR := LINK [ START ]
4.  Repeat Steps 3 - 4 while PTR ≠ NULL
5.  If ITEM < INFO [ PTR ] then
        Set LOC := SAVE and Return
6.  Else
      Set SAVE := PTR and PTR := LINK [ PTR ]
    [ End of if ]
    [ End of Step 4 loop ]
7.  Set LOC := PTR and Return
```

Header Linked Lists

- A Header Linked List always Contains a Special Node called Header Node
- It has Two Types:

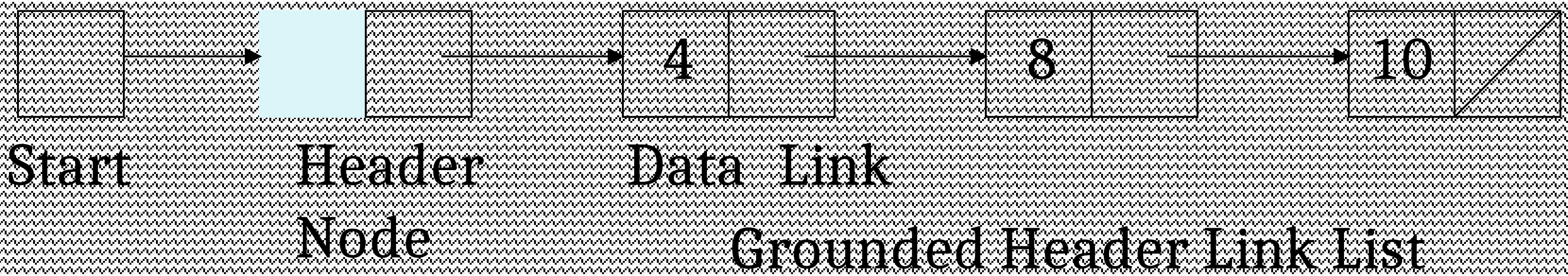
a) Grounded Header List

Last Node Contains the NULL Pointer

b) Circular Header List

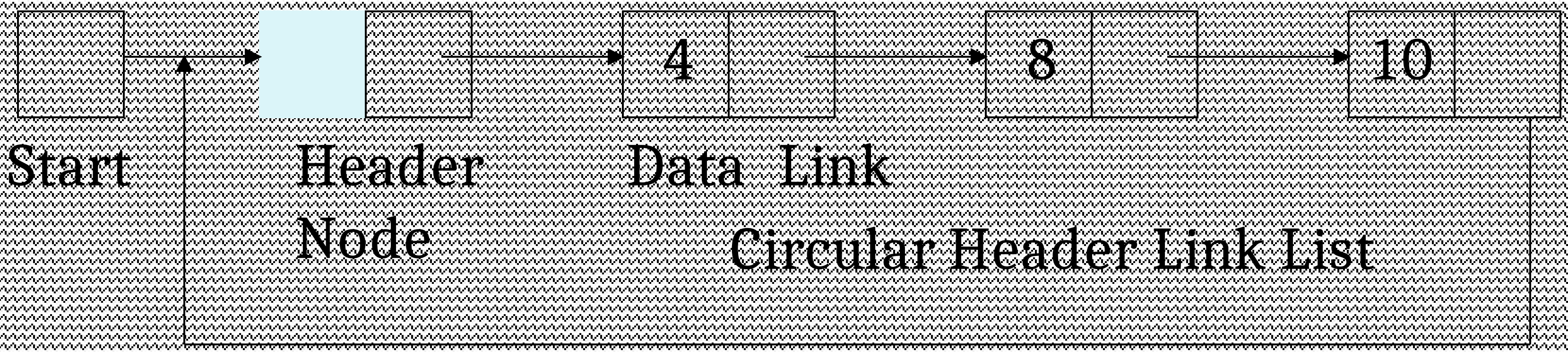
Last Node Points Back to the Header Node

Graphical Representations



Grounded Header Link List

LINK [START] = NULL



Circular Header Link List

LINK [START] = START

Header Linked Lists

- One way to simplify insertion and deletion is never to insert an item before the first or after the last item and never to delete the first node
- You can set a header node at the beginning of the list containing a value smaller than the smallest value in the data set
- You can set a trailer node at the end of the list containing a value larger than the largest value in the data set.
- These two nodes, header and trailer, serve merely to simplify the insertion and deletion algorithms and are not part of the actual list.
- The actual list is between these two nodes.

Doubly Linked Lists

- A Doubly Linked List is List in which every Node has a Next Pointer and a Back Pointer

- Every Node (Except the Last Node) Contains the Address of the Next Node, and Every Node (Except the First Node) Contains the Address of the Previous Node.

- A Doubly Linked List can be Traversed in Either Direction

Definition and Graphical Representations

The Node is Divided into 3 parts

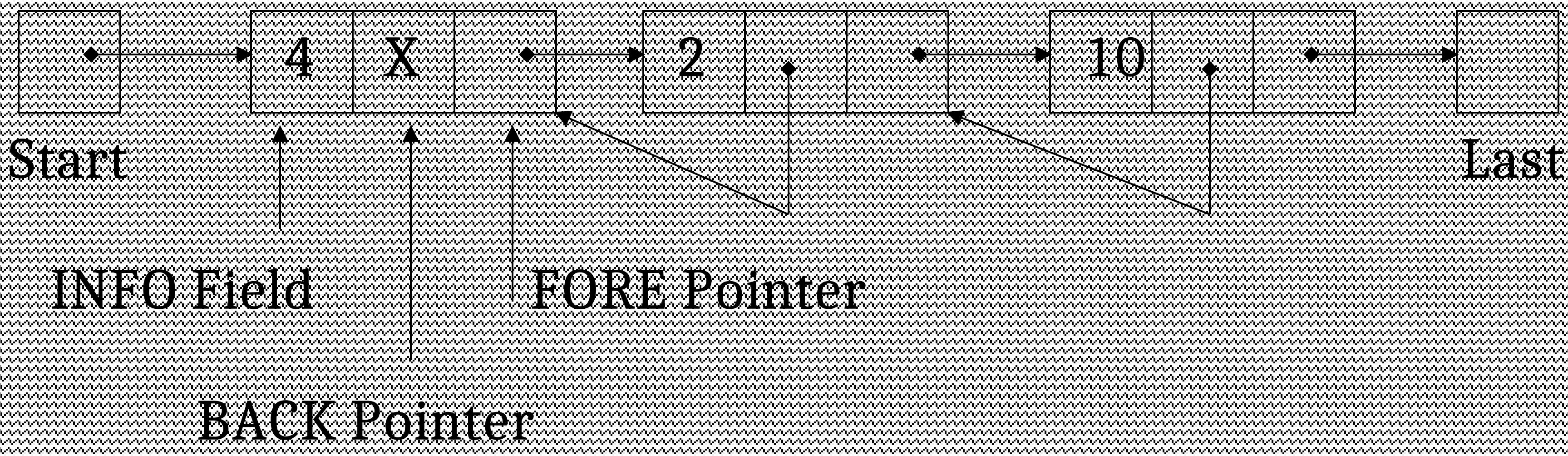
1) Information Field

2) Forward Link which Points to the Next Node

3) Backward Link which Points to the Previous Node

The starting Address or the Address of First Node is Stored in START / FIRST Pointer

Another Pointer can be used to traverse Doubly LL from End. This Pointer is Called END or LAST


4
X
2
10
Start
Last
INFO Field
FORE Pointer
BACK Pointer