CSE 207

Linked List

Linear Data Structures

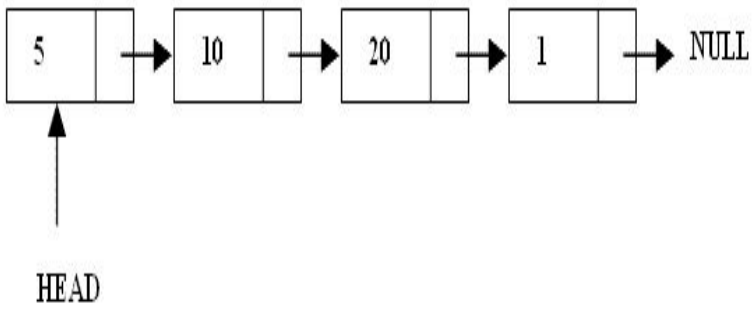
Definition

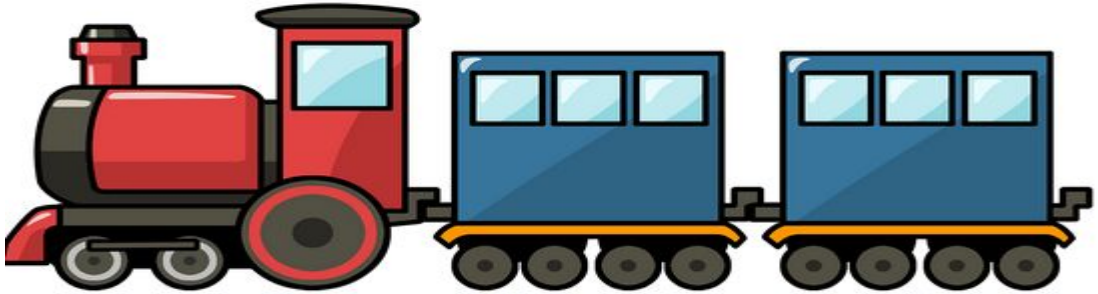
A data structure is said to be linear if its elements form a sequence or a linear list.

- Examples:
  - Array
  - Linked List
  - Stacks
  - Queues
- Operations on linear Data Structures
  - Traversal : Visit every part of the data structure
  - Search : Traversal through the data structure for a given element
  - Insertion : Adding new elements to the data structure
  - Deletion : Removing an element from the data structure.
  - Sorting : Rearranging the elements in some type of order(e.g Increasing or Decreasing)
  - Merging : Combining two similar data structures into one

What are Linked Lists

- A linked list is a linear data structure.
- A chain of structures make up linked lists called Node.
- Nodes are made up of data and a pointer to another node.
- Usually the pointer points to a structure of the same type as itself

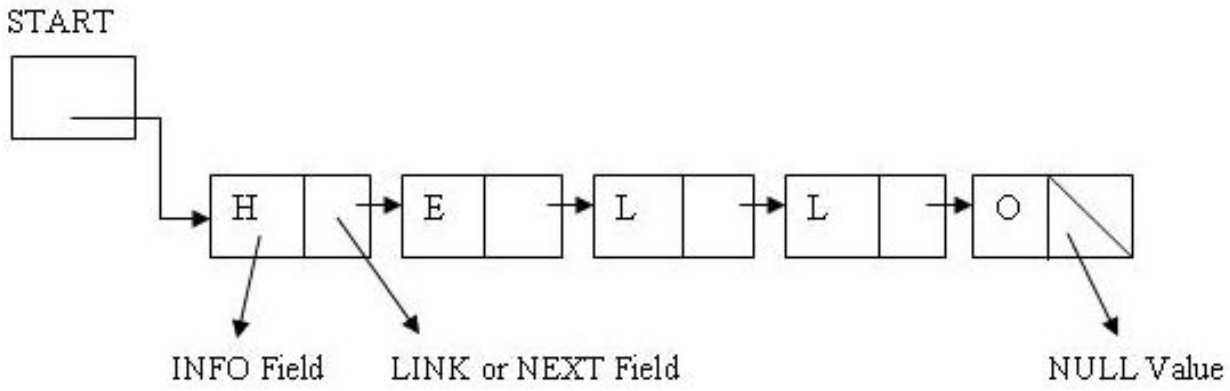




Arrays Vs Linked Lists

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]

Representation of Linked List in Memory



START = 9

	INFO	LINK
1		
2	L	8
3		
4	E	6
5		
6	L	2
7		
8	O	0
9	H	4
10		

Memory Representation of Linear Linked List

Here

START = 9	=>	INFO[9] = H is the first character.
LINK[9] = 4	=>	INFO[4] = E is the second character.
LINK[4] = 6	=>	INFO[6] = L is the third character.
LINK[6] = 2	=>	INFO[2] = L is the fourth character.
LINK[2] = 8	=>	INFO[8] = O is the fifth character.
LINK[8] = 0	=>	The NULL value, so the LIST ends here.

Types of Lists

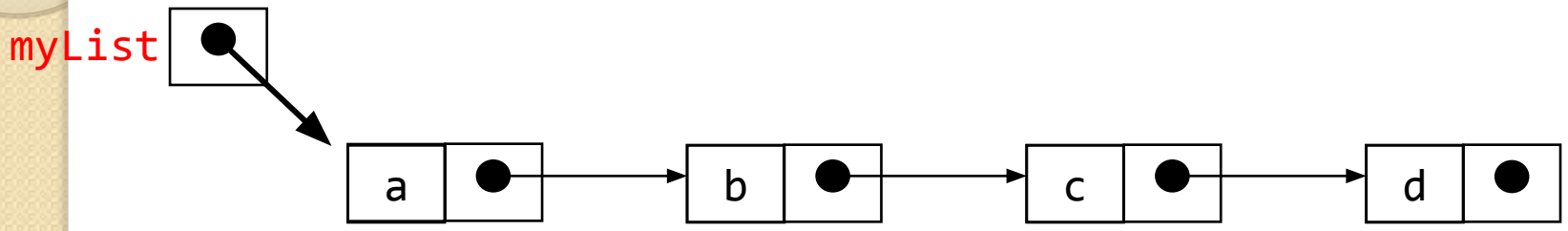
- Singly Linked list
- Doubly linked list
- Circular linked list

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

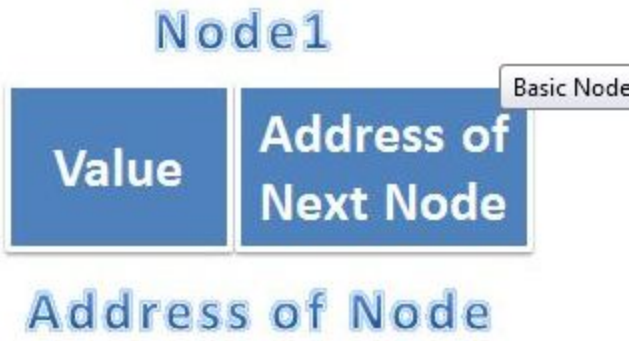
- myList is the header pointer which points at the first node in the list

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

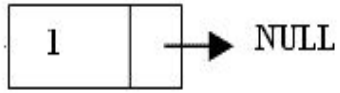
Creating a Node

```
struct node {
        int val;                        // A simple node of a linked list
        struct  node * next; // pointer to the next address
} ;
```



Creating a Node

```
void create()
{
       struct node *curr, * head,*tail;
       head = NULL;
     // beginning of loop
      curr = (struct node*)malloc(sizeof(struct node));
      curr-> val =read value;
      curr->next  = NULL;

      if (head == NULL)
      {
          head = curr;
          tail = curr;
      }
      else {
          tail->next = curr;
          tail = curr;
      }
// end of loop;
}
```



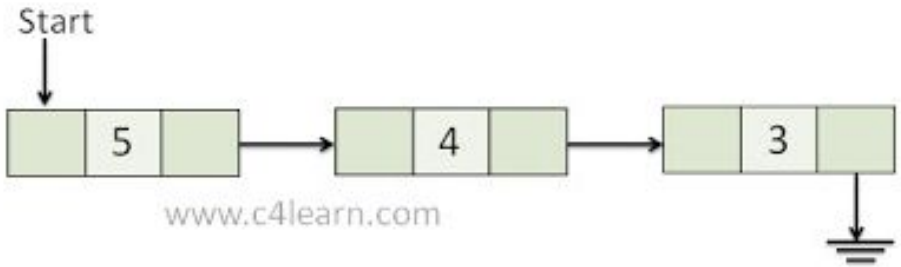
To Display

```
void display()
{
   curr = head;
    printf( "\n");
    while(curr != NULL)
    {
          printf(" %d---> ", curr->val);
          curr = curr->next ;
   }
}
```

Traversing a Linked List

Traversing linked list means visiting each and every node of the Singly linked list. Following steps are involved while traversing the singly linked list –

- Firstly move to the first node
- Fetch the data from the node and perform the operations such as arithmetic operation or any operation depending on data type.
- After performing operation, advance pointer to next node and perform all above steps on Visited node.



Inserting the Node in a SLL

There are 3 cases :-

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

Insertion at the Beginning

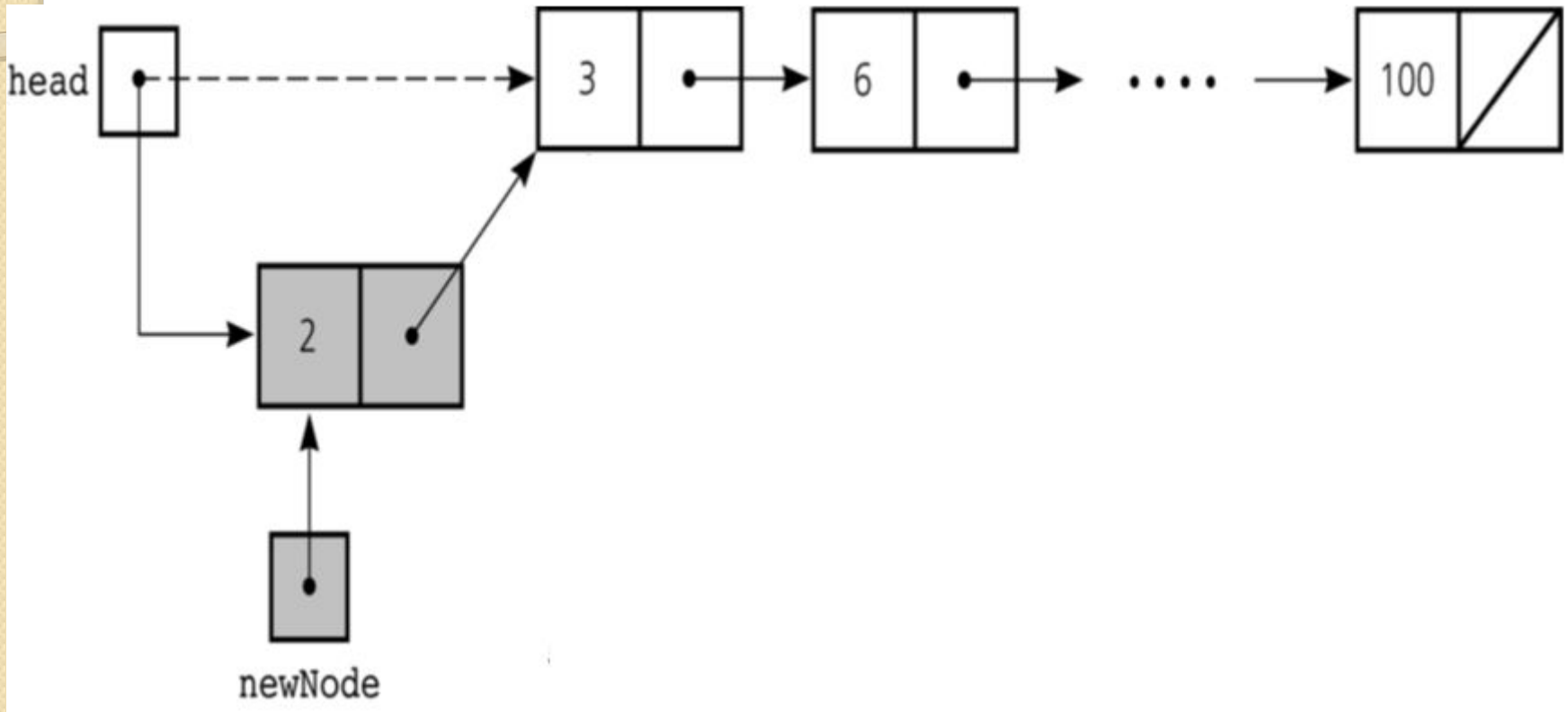
There are two steps to be followed:-

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

b) Make the head pointer point towards this new node

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

Insertion at the Beginning

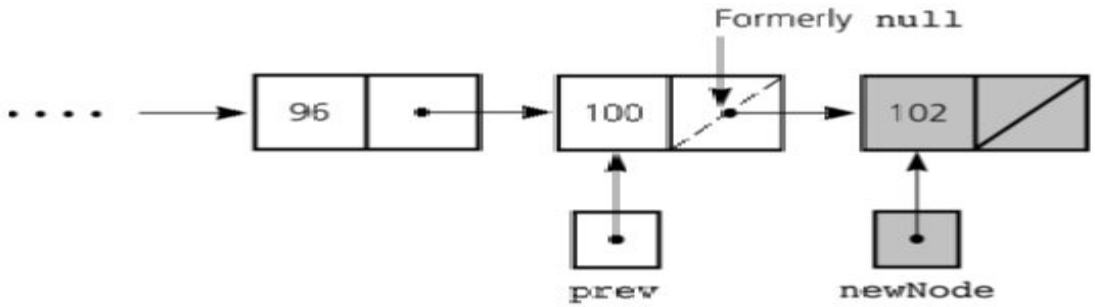


Insertion at the Beginning

```
void insert_at_beg() {
      struct node *newnode,*temp;
      newnode=(struct node *)malloc(sizeof(struct node));
       scanf("%d",&newnode->data);
       newnode->next=NULL;
        if(head==NULL) {
              head=newnode;
              tail=newnode; }
        else {
         newnode->next= head;
         head=newnode; }
   }
```

Inserting at the End

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



Inserting at the End

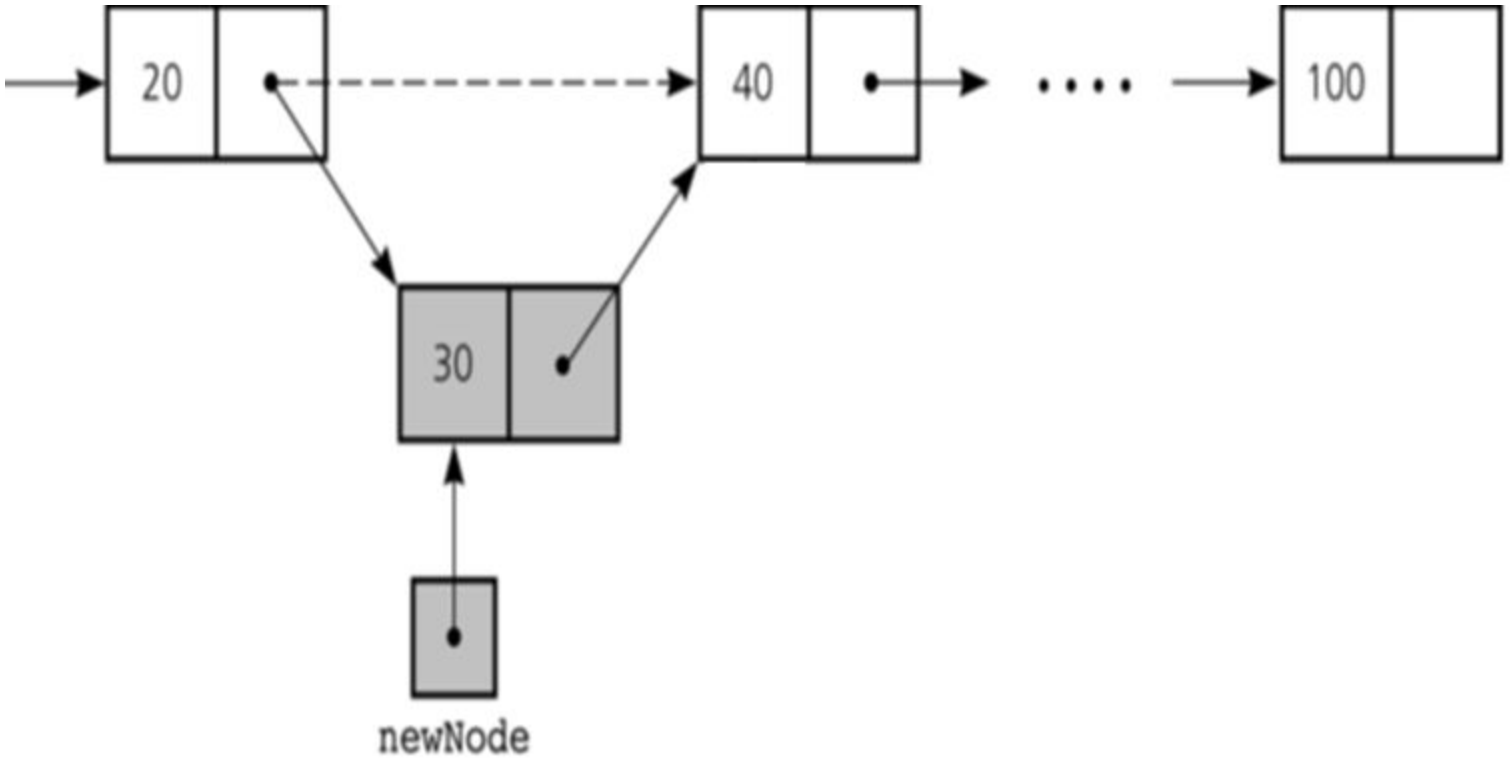
```
void insert_at_end() {
      struct node *newnode,*temp;
      newnode=(struct node *)malloc(sizeof(struct node));
       scanf("%d", &newnode->data);
       newnode->next=NULL;
       if(head==NULL) {
              head=newnode;
              tail=newnode; }
        else {
                 temp = head;
                 while(temp->next!=NULL) {
                              temp = temp->next;  }
                  temp->next = newnode;
            tail = newnode; }
}
```

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

Inserting After an Element



Inserting After an Element

```
void insert_mid()
{
    int pos,i; struct node *newnode,*curr,*temp,*temp1;
    newnode=(struct node *)malloc(sizeof(struct node));
     printf("nEnter the data : "); scanf("%d",&newnode->data);
     newnode->next=NULL;
     printf("nEnter the position : "); scanf("%d",&pos);

     if(head==NULL) { head=newnode; tail=newnode; }
     else { temp = head;
              for(i=1;i<= pos-1;i++)   {
                   temp1 = temp;
                   temp=temp->next; }
               temp1->next = newnode;
               newnode->next=temp; }
 }
```

Deleting a Node in SLL

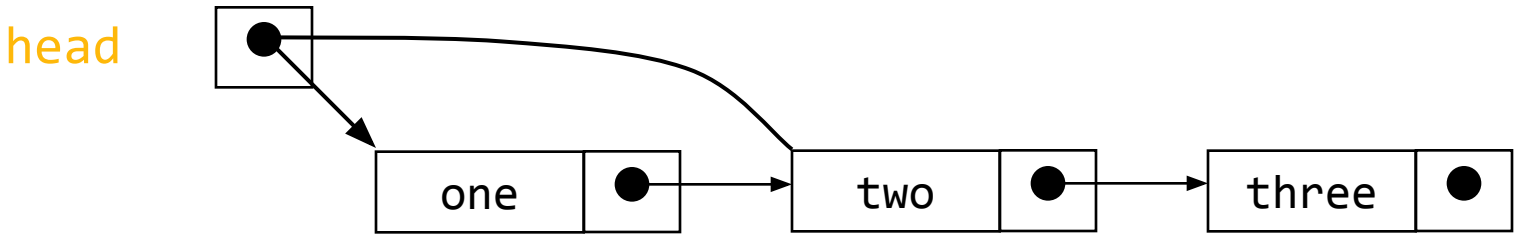
Here also we have three cases:-

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

Deleting the First Node

Here we apply 2 steps:-

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



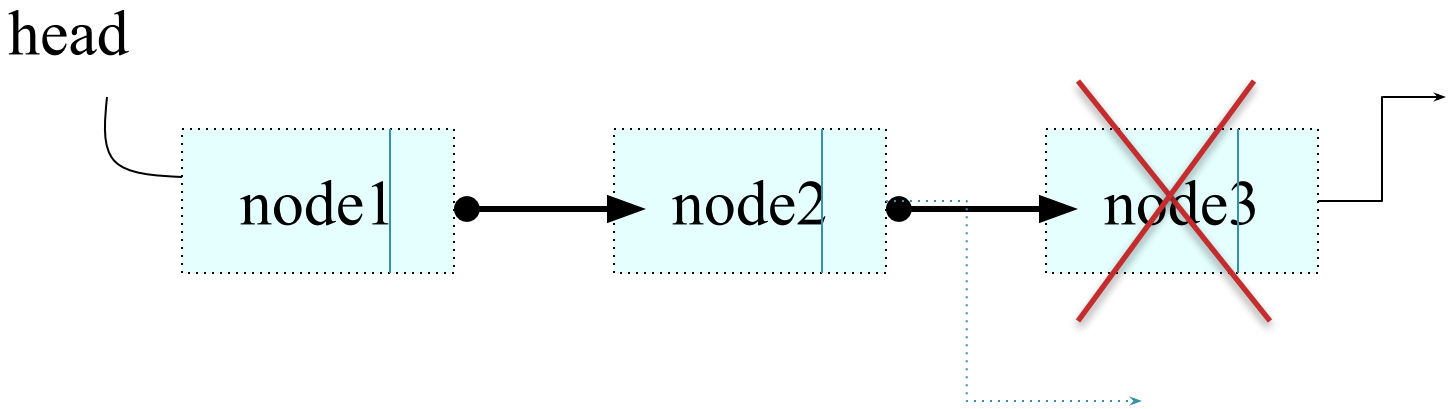
Deleting the First Node

```
void del_beg()
{
        struct node *temp;
        temp = head;
        head = head->next;
        free(temp);
}
```

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

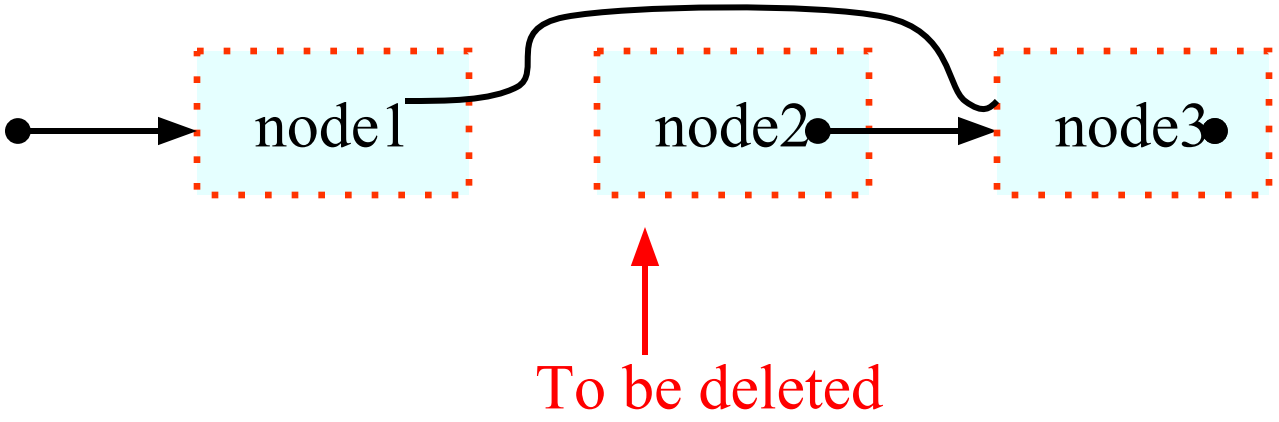


Deleting the Last Node

```
void  delete_at_end()
{
        struct node *temp,*temp1;
        if(head!=NULL) {
                temp = head;
                while(temp->next!=NULL) {
                        temp1 = temp;
                        temp = temp->next;
                 }
                 temp1->next = NULL;
                 delete(temp);
         }
}
```

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



Deleting a Particular Node

```
void  delete_any()
{
        struct node *temp,*temp1;
        int key = data to be deleted;
        if(head!=NULL) {
                temp = head;
                while(temp->next!=NULL and temp->data != key) {
                        temp1 = temp;
                        temp = temp->next;
                 }
                 if (temp->data == key){
                  temp1->next = temp->next;
                  delete(temp);
                   }
                   else "ITEM NOT FOUND"
         }
}
```

Searching a SLL

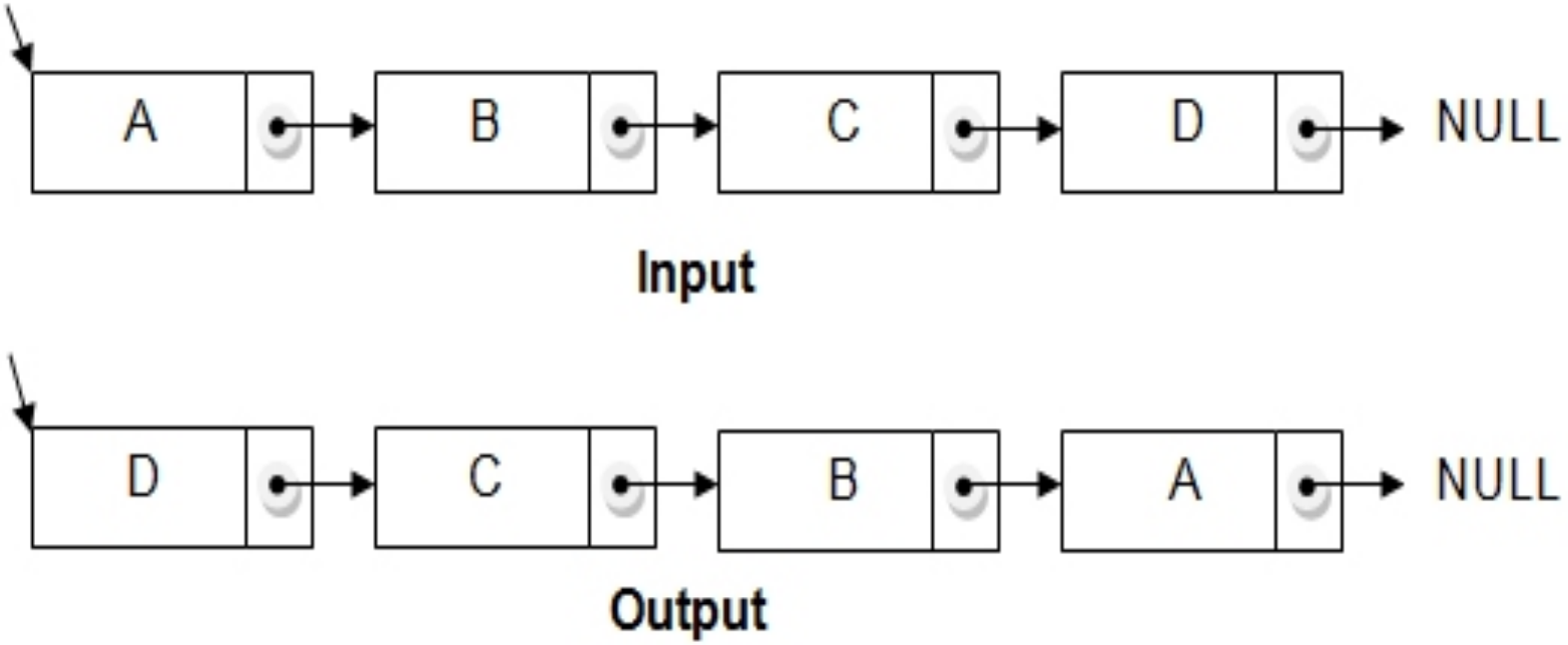
- Searching involves finding the required element in the list
- We can use various techniques of searching like linear search or binary search where binary search is more efficient in case of Arrays
- But in case of linked list since random access is not available it would become complex to do binary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

Searching a SLL

```
 void search(int x)
{
   struct node *temp=head;
      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



Doubly Linked List

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

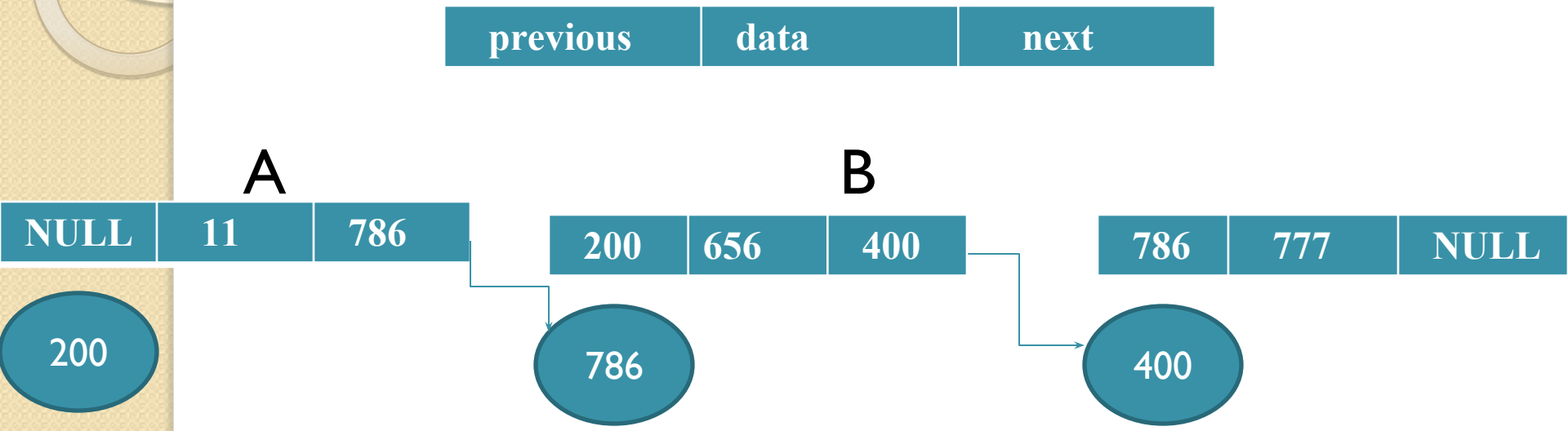
2 . Each node contains three fields ::

-: 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



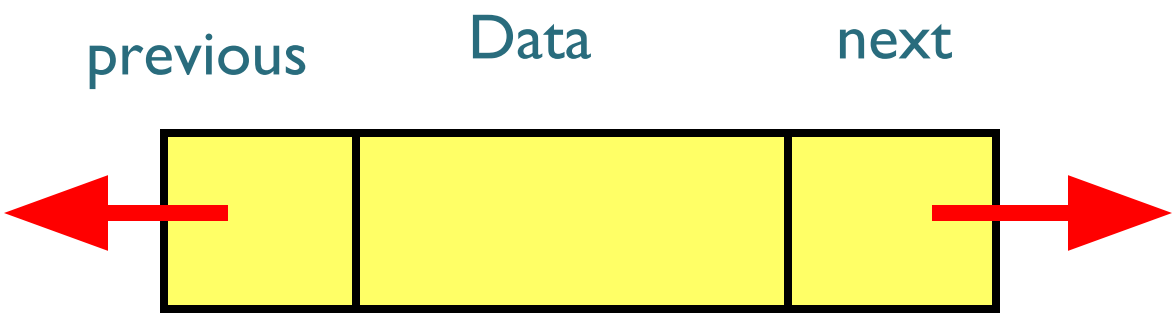
A doubly linked list contain three fields: an integer value, the link to the next node, and the link to the previous node.

DLL's compared to SLL's

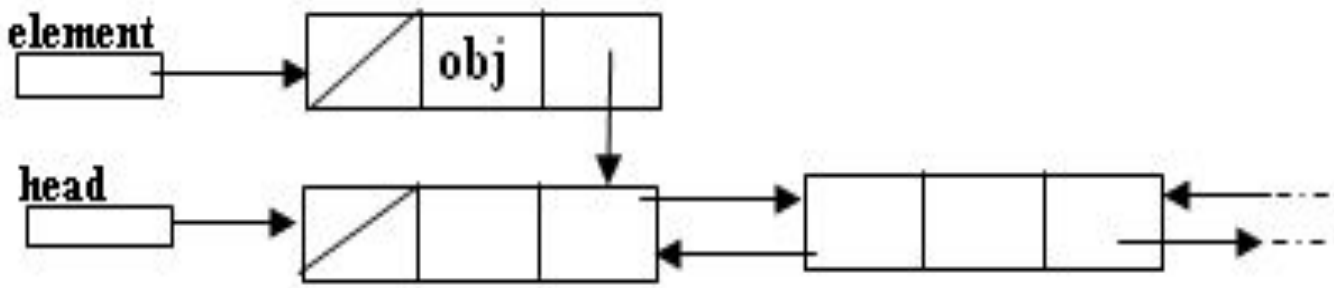
- Advantages:
  - We can traverse in both directions i.e. from heading to end and as well as from end to heading.
  - It is easy to reverse the linked list.
  - If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

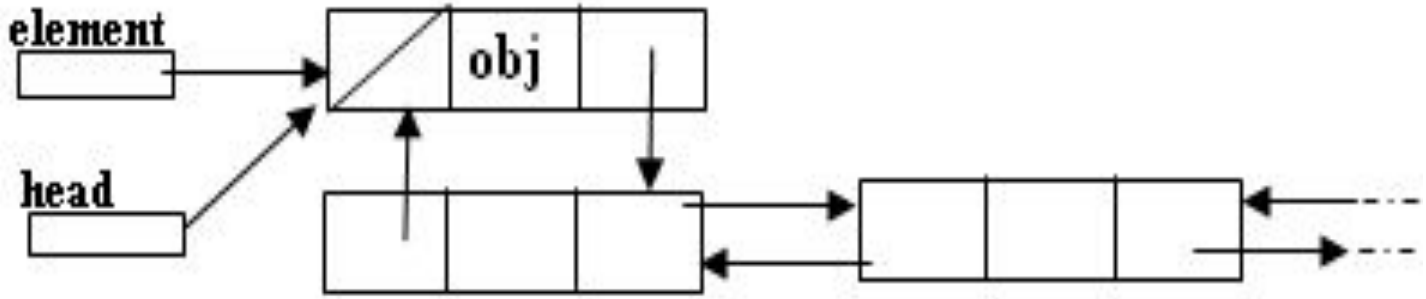
Structure of DLL

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



Inserting at Beginning

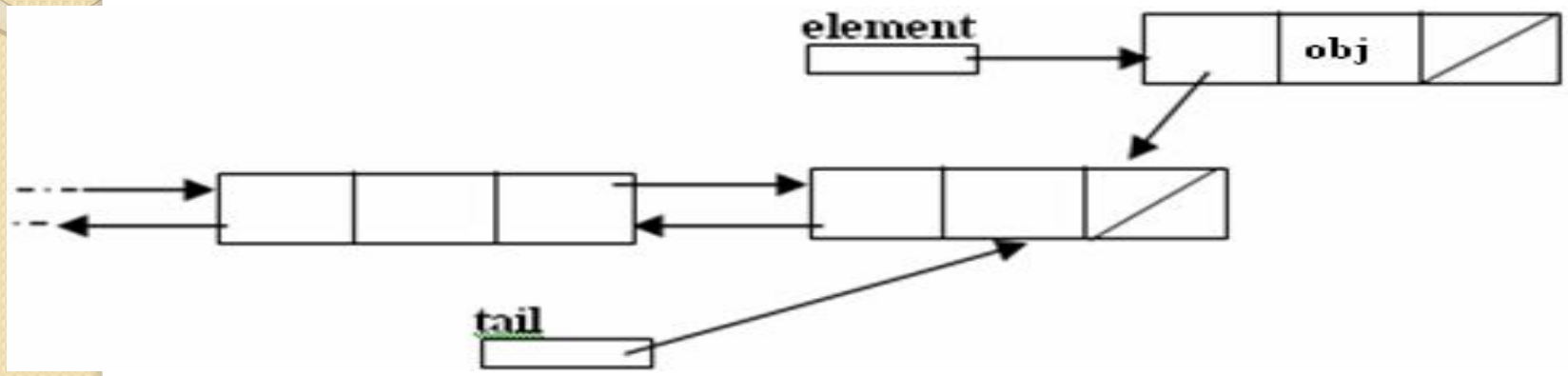


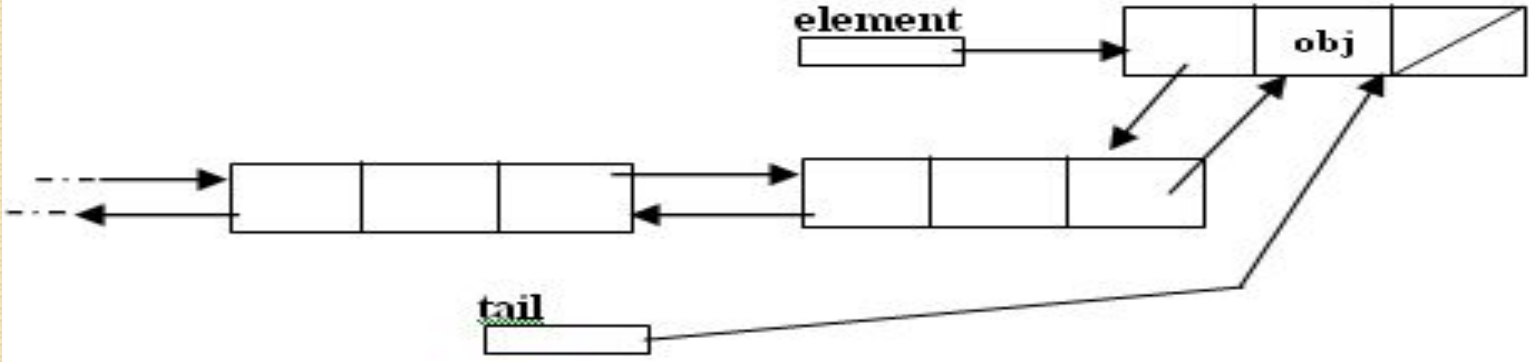


Inserting at Beginning

```
void insert_beg(node *p)
{
  if(head==NULL)
  {
    head=p;

  }
  else
  {
  node* temp=head;
  head=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

  }
```

Inserting at the End

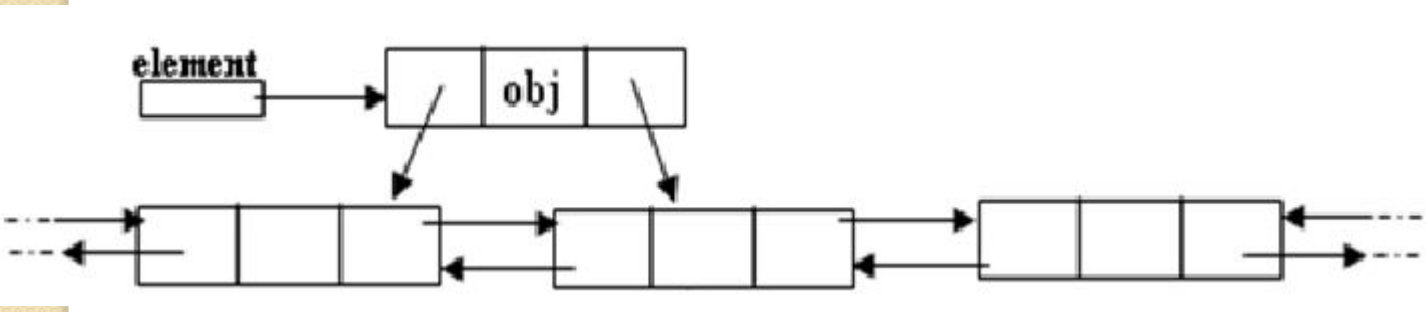




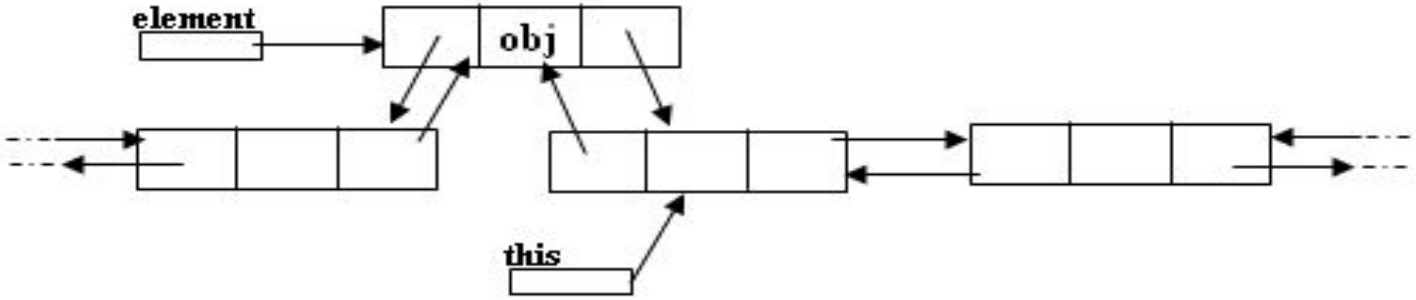
Inserting at the End

```
void insert_end(node* p)
{
    if(head==NULL)
    {
    head=p;

    }
    else
    {
     node* temp=head;
       while(temp->next!=NULL)
       {
       temp=temp->next;
       }
     temp->next=p;
     p->previous=temp;

    }
}
```

Inserting After a Node



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



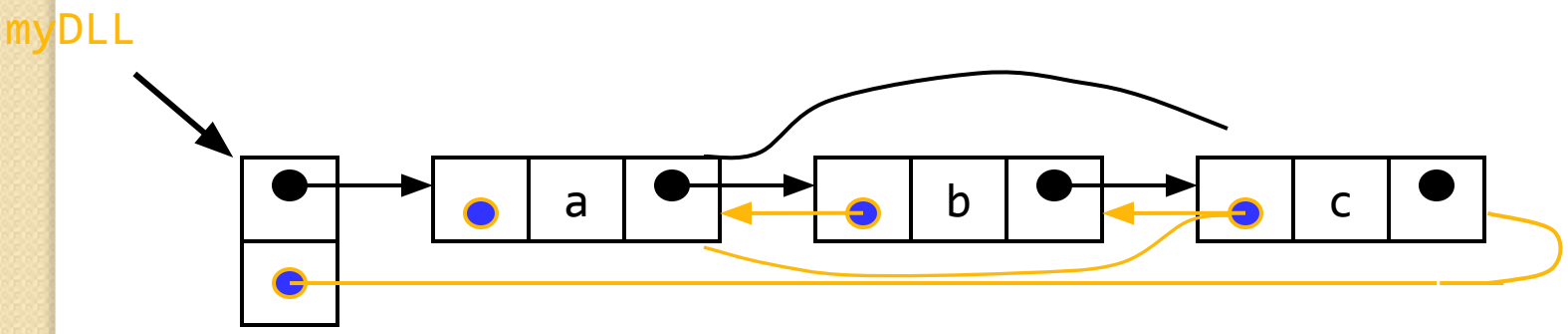
Adjusting the next and previous pointers of the nodes b/w which the new node accordingly

Inserting After a Node

```
void insert_after(int c,node* p)
{
     temp=head;
     for(int i=1;i<c-1;i++)
     {
    temp1 = temp;
     temp=temp->next;
     }
     p->next=temp;
     temp->previous=p;
     temp1->next=p;
     p->previous=temp1;
}
```

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

Deleting a Node

```
void del_at(int c)
{
    node *temp=head;
    for(int i=1;i<c-1;i++)
    {
      temp1= temp;
     temp=temp->next;
    }
    temp2=temp->next;
    temp1->next=temp2;
    temp2->previous=temp1;
    free(temp);
}
```

Applications of Linked List

1. Applications that have an MRU(most recently used) 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 linked list.