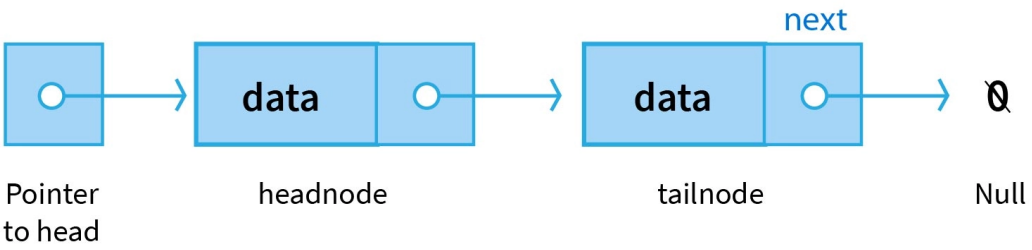
Practical – 6
Singly Linked List Operations and Applications

A **singly linked list** is a type of linked list that is *unidirectional*, that is, it can be traversed in only one direction from head to the last node (tail).
Each element in a linked list is called a **node**. A single node contains *data* and a pointer to the *next* node which helps in maintaining the structure of the list.

OR

A Linked List is a linear data structure consisting of connected nodes where each node has corresponding data and a pointer to the address of the next node. The first node of a linked list is called the Head, and it acts as an access point. On the other hand, the last node is called the Tail, and it marks the end of a linked list by pointing to a NULL value!

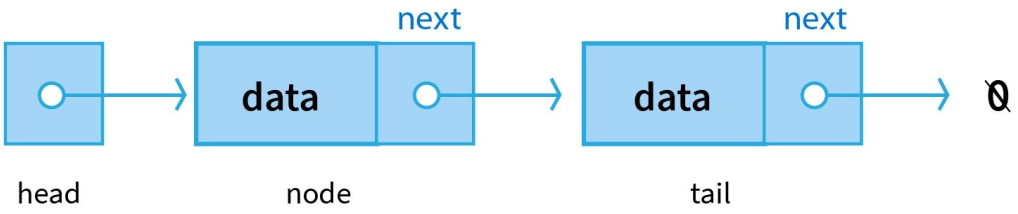


Types of Linked List

There are mainly three types of linked lists:

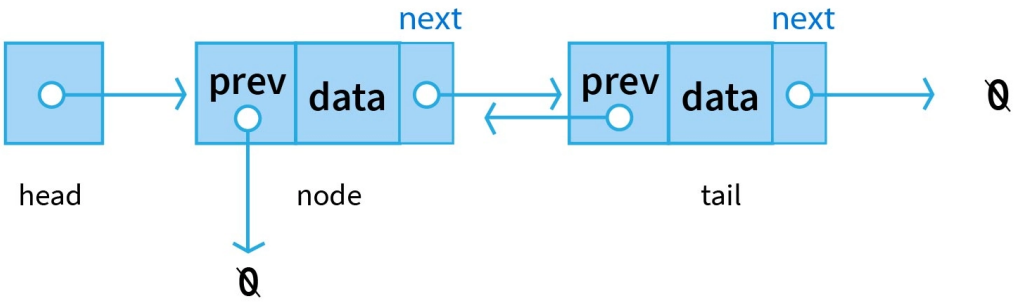
Singly Linked List

Here, we can only traverse in one direction (not the band) due to the linking of every node to its next node.



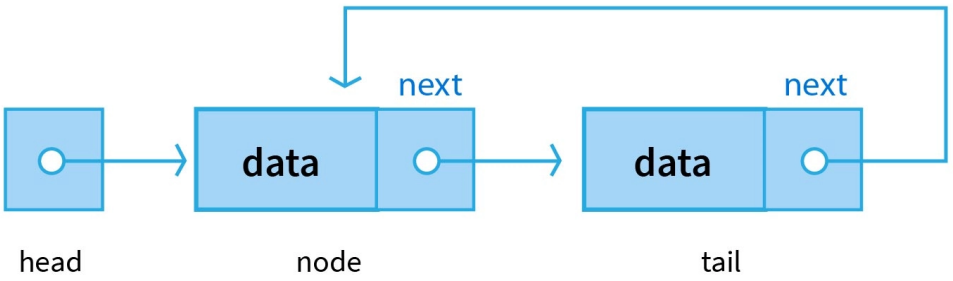
Doubly Linked List

Here, we can traverse in both directions as every node contains an additional prev pointer that points to the previous node.



Circular Linked List

Here, we can keep traversing forever and ever until the program crashes as the tail node's next pointer points to the head node instead of a NULL.



Basic LinkedList Functions & Operations

- In C programming Language, a LinkedList is a data structure consisting of nodes, nodes are connected using addresses.
- LinkedList is the most used Data Structure after the array, in fact, LinkedList has many advantages than an array, like, adding elements at any position, insertion, deletion can be performed more efficiently than an array.
- LinkedList is a collection of nodes, where every node contains two fields:
    - Data field: It stores the actual value address field.
    - Address field: It stores the reference of the next node.
- In the real world, LinkedList is like a conga line, where every person holds the hips of the person in front of them, except only those in the front and the back.

Many applications use LinkedList in computer science, let's discuss basic LinkedList functions.

- A node can be represented using structures.
- A node takes the form of a user-defined structure, a node contains two parts,i.e. to store data and to store the reference of the next node
- Basic LinkedList functions are create(), display(), insert_begin(), insert_end(), insert_pos(), delete_begin(), delete_end(), delete_pos()

create()

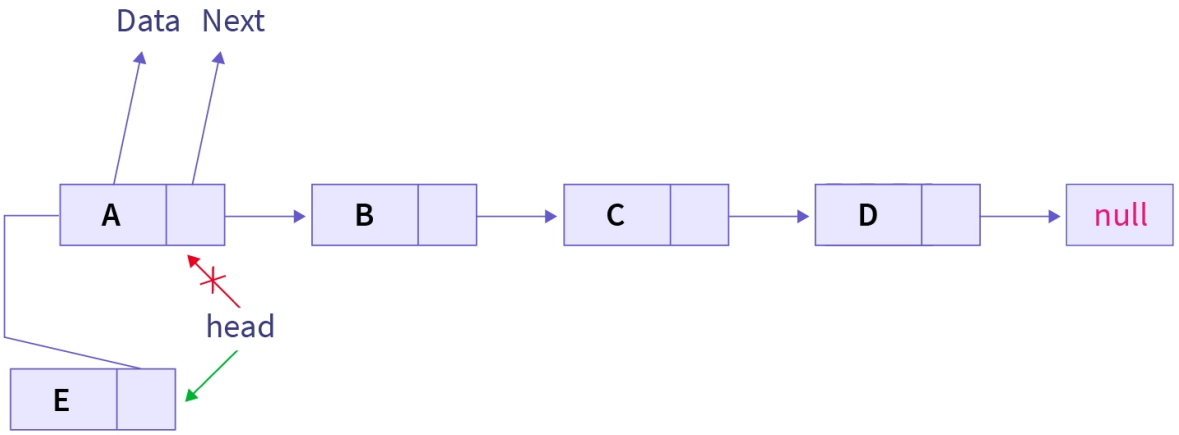
- This function is a foundation pillar for the entire linked list.
- Here, we can create a temp node to scan the value.
- Then we check if LinkedList is empty or not, if LinkedList is empty then the temp node would be the head node. (optional - can be part of insert)
- If LinkedList is not empty, then by using another node, we traverse till the end of LinkedList and add the temp node at the end of LinkedList. (optional - can be part of insert)

display()

- This function is used to display the entire LinkedList using a loop
- We first check, if the head node is pointing to NULL or not, if the head node is pointing to NULL, then it indicates that LinkedList is empty, so we return
- If LinkedList is not empty, we assign head node to a temp node and we use this temp node to traverse over the LinkedList using a loop and print them

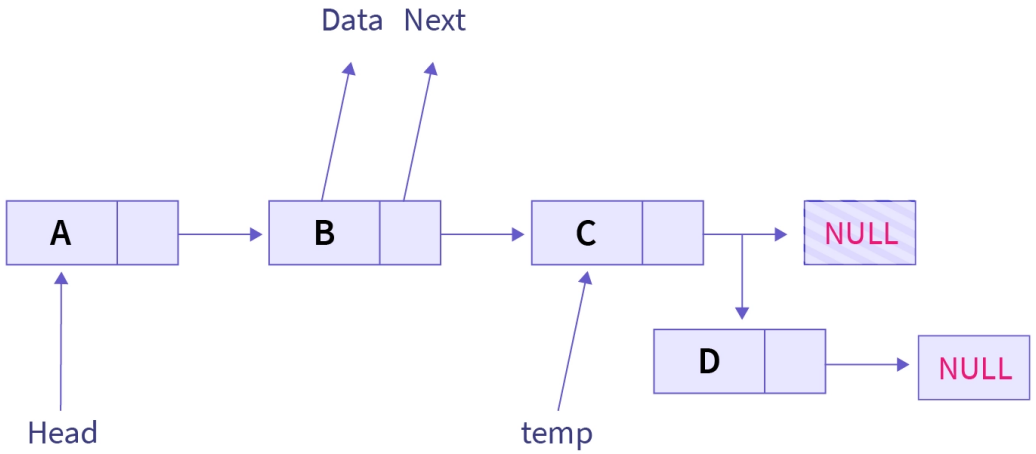
insert_begin()

- Initially, we create a temp node to scan the value then we check if LinkedList is empty or not
- If LinkedList is empty, then the newly created node would be treated as a head node
- If LinkedList is not empty, then we make the temp node point towards the current head node and the head node to point towards the newly created node



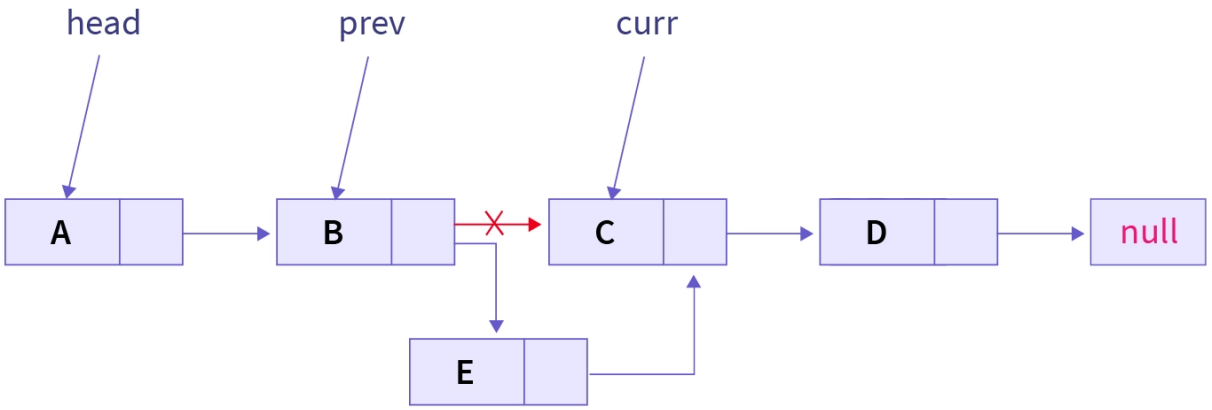
insert_end()

- Firstly, we create a temp node to scan the value then we check if LinkedList is empty or not
- If LinkedList is empty, then the newly created node would be inserted to LinkedList
- If LinkedList is not empty, then we create a new node say ptr, by using ptr we traverse till the end of LinkedList and insert the temp node at the end of LinkedList



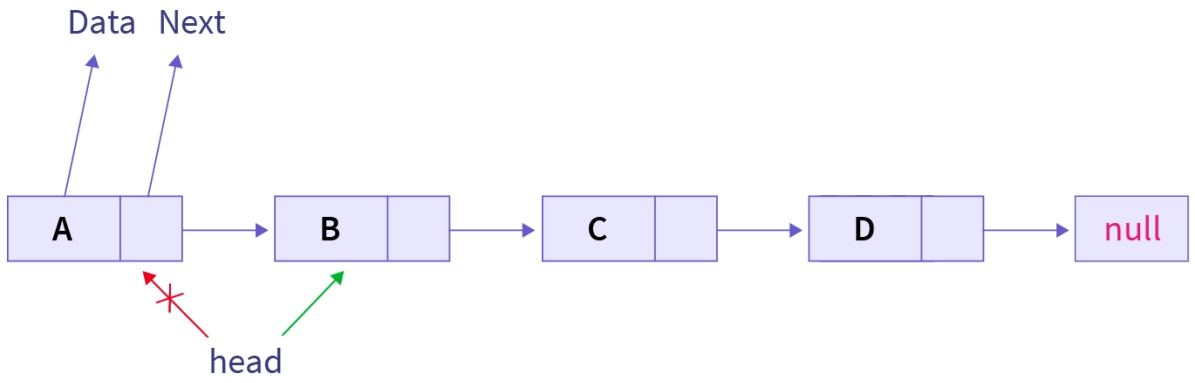
insert_pos()

- Here, we create a temp node to scan the value then we check if LinkedList is empty or not
- If LinkedList empty, then we return
- If LinkedList is not empty, then we take input of node position from the user, if the input is greater than the length of LinkedList, then we return
- If the input is in the range of length of LinkedList then, let's assume we have four nodes A, B, C, D and we need to insert a node next to B, so, we just traverse till node C and make node B point to node E and node E to point to node C.



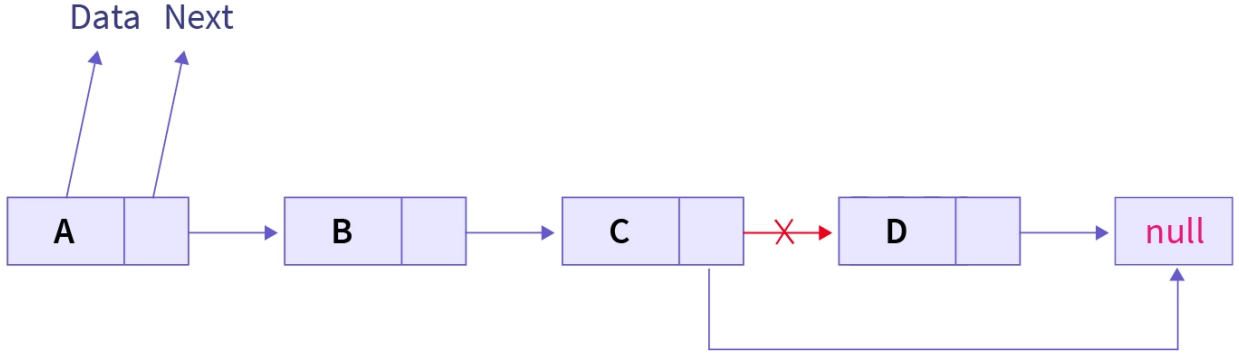
delete_begin()

- This function checks if nodes are present in LinkedList or not, if nodes are not present then we return
- If nodes are present then we make ahead node to point towards the second node and store the address of the first node in a node say, temp
- By using the address stored in temp, we delete the first node from the memory



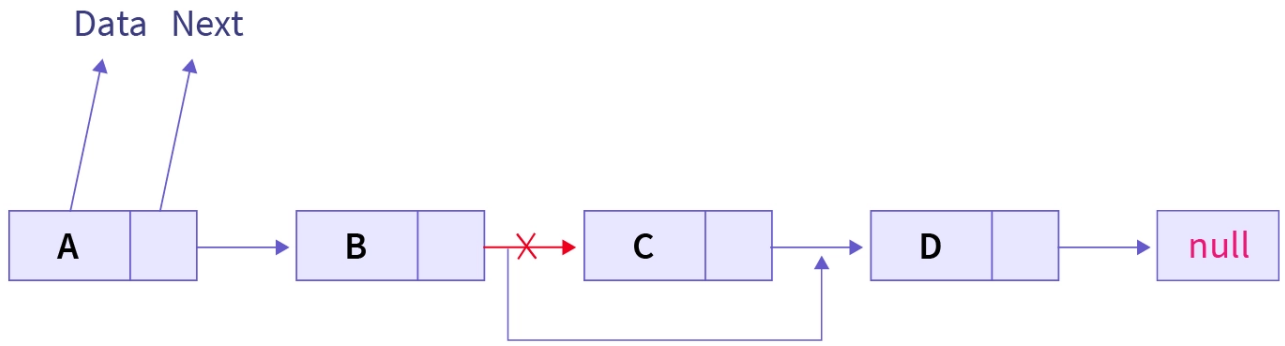
delete_end()

- This function checks if nodes are present in LinkedList or not, if nodes are not present in LinkedList, then we return
- If nodes are present in LinkedList, then we create a temp node and assign a head node value in it.
- By using this temp node, we traverse till last but one node of the LinkedList, and then we store the address present in the next field in a node say ptr.
- Now, we delete the ptr from memory, such that the last node is deleted from LinkedList



delete_pos()

- On invoking this function, we check if nodes are present in LinkedList or not, if nodes are not present then we return
- If nodes are present in LinkedList, as x,y,z and we need to delete node y
- To delete node y, we traverse till node x and make x to point towards node z, then we delete node y from memory




```
struct Node
{
    int data;
    struct Node* next;
};

struct Node* newNode(int data, struct Node* nextNode)
{
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->next = nextNode;
    return temp;
}
int main()
{
    struct Node* head = newNode(100, newNode(200, newNode(300,
newNode(400, newNode(500, NULL)))));
    struct Node* ptr = head;
    while(ptr!=NULL)
    {
      printf("%d ",ptr->data);
      ptr = ptr->next;
    }
}
```

Linked lists have a few advantages over arrays:

- Items can be added or removed from the middle of the list
- There is no need to define an initial size. They are dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

However, linked lists also have a few disadvantages:

- There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
- Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
- Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.
- Reverse Traversing is difficult in the linked list.

Why do we need pointers in Linked List?

- In the case of an array, memory is allocated in a contiguous manner, hence array elements get stored in consecutive memory locations. So when you have to access any array element, all we have to do is use the array index, for example arr[4] will directly access the 5th memory location, returning the data stored there.
- But in the case of linked lists, data elements are allocated memory at runtime, hence the memory location can be anywhere. Therefore, to be able to access every node of the linked list, the address of every node is stored in the previous node, hence forming a link between every node.
- We need this additional pointer because without it, the data stored at random memory locations will be lost. We need to store all the memory locations where elements are getting stored.
- Yes, this requires an additional memory space with each node, which means an additional space of O(n) for every n node linked list.

Singly Linked list operations

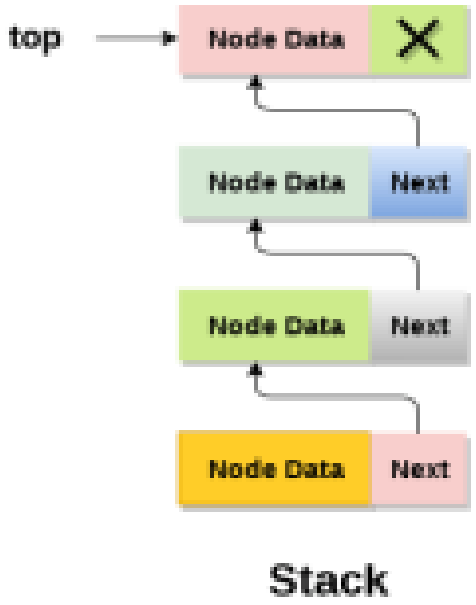
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   char data;
   struct node* next;
}*head=NULL, *temp, *newnode;
void create();
void insertFirst();
void insertEnd();
void insertSpecific(int loc);
void deleteFirst();
void deleteSpecific(int loc);
void display();
int count(struct node *q);
int main()
{
   int choice,loc;
   while(1)
   {
        printf("\n1.Insert at first \n2.Insert at end \n3.Insert
at  desired location \n4.Delete at first \n5.Delete at end
\n6.Delete at  desired location \n7.Display the list \n8.Exit
\nEnter Choice= "); scanf("%d",&choice);
        printf("\n");
        switch(choice)
        {
             case 1: insertFirst();break;
             case 2: insertEnd();break;
             case 3: printf("Enter location= ");
                  scanf("%d",&loc);
                  insertSpecific(loc);
                  break;
             case 4: deleteFirst();break;
             case 5: printf("Enter location= ");
                  scanf("%d",&loc);
                  deleteSpecific(loc);
                  break;
             case 6: display(); break;
```

```
            case 7: exit(0); break;
                        default: printf("Invalid choice");break;
        }
   }
   return 0;
}
void create()
{
   newnode=(struct node *)malloc(sizeof(struct node));
   if(newnode==NULL)
   {
        printf("No enough memory available\n");
        exit(0);
   }
   newnode->next=NULL;
   printf(" Enter Value= ");
   scanf(" %c",&newnode->data);
}
void insertFirst()
{
   create();
   if(head==NULL)
   {
        head=newnode;
        return;
   }
   newnode->next=head;
   head=newnode;
}
void insertEnd()
{ create();
   if(head==NULL)
   { head=newnode;
        return;
   }
   temp=head;
   while(temp->next!=NULL)
   { temp=temp->next;
   }
   temp->next=newnode;
}
void insertSpecific(int loc)
{
```

```
   int count=2;
   struct node *pred;
   temp=head;
   if(loc==1)
   { insertFirst();
   }
   else
   { while(temp->next!=NULL && count!=loc) {
             count++;
             //pred=temp;
             temp=temp->next;
        }
        create();
        newnode->next=temp->next;
        temp->next=newnode;
   }
}
void deleteFirst()
{
   //struct node *pred;
   if(head==NULL)
   { printf("List is empty\n");
   }
   else
   {
        temp=head;
        head=head->next;
        free(temp);
   }
}
void deleteSpecific(int loc)
{
   int count=1;
   struct node *pred;
   temp=head;
   if(head==NULL)
   {
        printf("List is empty\n");
   }
   else if(loc==1)
   {
        deleteFirst();
   }
```

```
   else
   { while(temp->next!=NULL && count!=loc) { count++;
             pred=temp;
             temp=temp->next;
        }
        pred->next=temp->next;
        free(temp);
   }
}
void display()
{
   struct node *temp;
   if(head==NULL)
   { printf("List is empty\n");
   }
   else
   {
        printf("ELements= ");
        temp=head;
        while(temp!=NULL)
        {
             printf("%c ",temp->data);
             temp=temp->next;
        }
        printf("\n");
   }
}
```

Stack implementation using singly linked list



Stack

Algorithm

push() - Inserting an element into the Stack

Step 1 - Create a newNode with a given value.

Step 2 - Check whether stack is Empty (top == NULL)

Step 3 - If it is Empty, then set newNode → next = NULL. Step 4 - If it is Not Empty, then set newNode → next = top. Step 5 - Finally, set top = newNode.

```
// C program for linked list implementation of stack

#include<stdio.h>
  #include<stdlib.h>
  struct node
  {
     int data;
     struct node *next;
  }*head=NULL,*newnode,*tmp;
  void create();
  void push();
  void pop();
  void peep(int pos);
  void display();
  int main()
  {
     char cc;
     int choice,pos,val;
     while(1)
     {
```

```
        printf("\n1.Push \n2.Pop \n3.Peep \n4.Change \n5.Display
\n6.Exit \nEnter Choice= ");
        scanf("%d",&choice);
        printf("\n");
        switch(choice)
        { case 1: push();break;
             case 2: pop();break;
             case 3: printf("Enter Position : ");
                  scanf("%d",&pos);
                  peep(pos);
                  break;
             case 4: display();break;
             case 5: exit(0);
                        default:printf("Invalid choice\n");break;
        };
   }
}
void create()
{ newnode = (struct node *) malloc(sizeof(struct node *));
   if(newnode==NULL)
   {
        printf("Memory Not Avaliable.\n");
   }
   printf("Enter value= ");
   scanf("%d",&newnode->data);
}
void push()
{
   create();
   newnode->next=head;
   head=newnode;
}
void pop()
{ if(head == NULL)
   {
        printf("STACK IS EMPTY\n");
   }
   else
   { tmp=head;
        head = tmp->next;
        free(tmp);
   }
}
```

```
void peep(int pos)
{
   int cnt = 1;
   tmp=head;
   while(tmp->next!=NULL&&cnt!=pos)
   { cnt++;
        tmp=tmp->next;
   }
   printf("Value is: %d\n",tmp->data);
}
void display()
{
   tmp = head;
   if(tmp == NULL)
   {
        printf("STACK IS EMPTY\n");
   }
   else
   { while (tmp!=NULL)
        { printf("| %d | \n",tmp->data);
             printf("|_________|\n");
             tmp = tmp -> next;
        }
        printf("\n");

   }
}
```

Exercise

1. Write program for all operations of singly link list. (store integer value in list) • Creation of List
    a. Inserting Node – as First Node,
    b. Inserting Node – as Last Node,
    c. Inserting Node – at desired location
    d. Deleting Node – at First,
    e. Deleting Node – at Last
    f. Deleting Node – a desired location
    g. Deleting Node – a specific value node
    h. Display List

2. Write a program to perform all stack operations using singly linked lists. Implement PUSH, POP, PEEP, Change and DISPLAY.

3. Write a program to perform sort on an integer linked list.