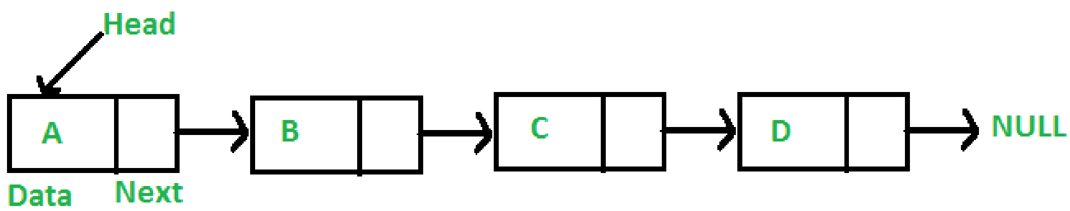
Practical 7

Aim :- Implementations of Stack Applications

What is Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Types of Linked List:-

1.Singly LinkedList

2.Doubly LinkedList

3.Circular LinkedList

1)Singly LinkedList

Complexities :-
Time Complexity:- O(1)
Space Complexity:- O(n)

Algorithm :-

Inserting At Beginning of the list

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.

- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

Deleting a Specific Node from the list

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11 -** If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Code :-

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

class node{
	public:
		int data;
		node* next;

		node(int val)
		{
			data = val;
			next=NULL;
		}
};

void insert(node* &head,int val)
{
	node* n = new node(val);

	if(head == NULL)
	{
		head = n;
		return;

	}
	node* temp = head;


	while(temp->next!=NULL)
	{
		temp = temp->next;
	}
	temp->next = n;

}

void insertatmiddle(node* &head,int val,int checkval)
{
	node* n = new node(val);
```

```
    if(head == NULL)
    {
        cout<< "List is empty"<<endl;
        return;
    }
    node* temp = head;


    while(temp->data!=checkval && temp->next!=NULL)
    {
        temp = temp->next;
    }

    if(temp->data!= checkval && temp->next==NULL)
    {
        cout<<"Value after which the new value is to be entered cannot be found"<<endl;
        return;
    }

    if(temp->next == NULL)
    {
        n->next = NULL;
        temp->next = n;
    }
    else
    {
        n->next = temp->next;
        temp->next = n;
    }
}

void display(node* head)
{
    node* temp = head;

    if(temp == NULL)
    {
        cout<<"Empty"<<endl;
    }

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

```
void reverseList(node* &head)
{
   node* prev = NULL;
   node* curr = head;
   node* next = NULL;
   while(curr != NULL)
   {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   head = prev;
}


void Delete(node* &head,int a)
{
        node* temp = head;
        if(temp != NULL)
        {
                if(temp->next == NULL)
                {
                        head = NULL;
                        return;
                }

                while(temp->next->data != a)
                {
                        temp = temp->next;
                }
                node* todelete = temp->next;
                temp->next = temp->next->next;

                delete todelete;
        }
        else
        {
                cout<<"List is empty "<<endl;
        }
}

int count(node* head)
{
        node* temp = head;
        int count = 0;
```

```
        if(temp == NULL)
        {
                cout<<"Empty"<<endl;
                return 0;
        }

        while(temp!=NULL)
        {

                temp=temp->next;
                count = count +1;
        }
        return count;
}

void search(node* head,int a)
{
                node* temp = head;
        int count = 0;

        if(temp == NULL)
        {
                cout<<"Empty"<<endl;
                return;
        }

        while(temp->data != a)
                {
                        if(temp->next!=NULL)
                        {
                        temp = temp->next;
                        }
                        else
                        {
                                cout<<"Value is not present "<<endl;
                                return;
                        }

                }
                cout<<"Element you searched is present in the list"<<endl;
}

int main()
{
node* head = NULL;

int ch;
  cout<<"1) Insert "<<endl;
```

```
cout<<"2) Insert after a specific node "<<endl;
cout<<"3) Delete"<<endl;
cout<<"4) Display "<<endl;
cout<<"5) Count"<<endl;
cout<<"6) Search"<<endl;
cout<<"7) Reverse"<<endl;
cout<<"8) Exit"<<endl;
do {
  cout<<"Enter your choice : ";
  cin>>ch;
  switch (ch) {
    case 1: {
        int a;
        cout<<"Enter your value : "<<endl;
        cin >> a;
        insert(head,a);
                    break;
              }
              case 2:
              {
                int a;
        cout<<"Enter the value after which you want to enter your new value : "<<endl;
        cin >> a;
        int b;
        cout<<"Enter your value : "<<endl;
        cin >> b;
        insertatmiddle(head,b,a);
                    break;
              }

    case 3: {
        int a;
        cout<<"Enter value to delete : "<<endl;
        cin >> a;
        Delete(head,a);
                    break;
              }
    break;
    case 4: display(head);
    break;
    case 5: cout<<"No of nodes present is "<<count(head)<<endl;
    break;
    case 6:
    {
        int a;
        cout<<"Enter value to search : "<<endl;
        cin >> a;
        search(head,a);
```

```
            break;
                        }
                        case 7:
        {
                     reverseList(head);
                     display(head);
                        }

        case 8: cout<<"Exit"<<endl;
        break;
        default: cout<<"Invalid choice"<<endl;
      }
  } while(ch!=8);

 return 1;
}
```

Output :-

```
D:\Data Structures\SinglyLinkedList.exe
1) Insert
2) Insert after a specific node
3) Delete
4) Display
5) Count
6) Search
7) Reverse
8) Exit
Enter your choice : 1
Enter your value :
5
Enter your choice : 1
Enter your value :
2
Enter your choice : 1
Enter your value :
4
Enter your choice : 1
Enter your value :
3
Enter your choice : 4
5 2 4 3
Enter your choice : 3
Enter value to delete :
2
Enter your choice : 4
5 4 3
Enter your choice : 5
No of nodes present is 3
Enter your choice : 6
Enter value to search :
4
Element you searched is present in the list
Enter your choice : 7
3 4 5
Exit
Enter your choice :
```

Doubly LinkedList

Complexities :-
Time Complexity:- O(1)
Space Complexity:- O(n)

Algorithm:
Inserting At Beginning of the list

- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At Specific location in the list (After a Node)

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

Deleting a Specific Node from the list

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5 -** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Display **'NULL <--- '**.
- **Step 5 -** Keep displaying **temp → data** with an arrow (**<===>**) until **temp** reaches to the last node
- **Step 6 -** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Code:-

```
  #include<iostream>
using namespace std;
class node{
	public:
		int data;
		node* next;
		node* prev;

		node(int val)
		{
			data = val;
			next=NULL;
			prev=NULL;
```

```
        }
};

void insertatRear(node* &head,int val)
{
    node* n = new node(val);

    if(head == NULL)
    {
        head = n;
        return;

    }
    node* temp = head;


    while(temp->next!=NULL)
    {
        temp = temp->next;
    }

    temp->next = n;
    n->prev = temp;

}

void insertatmiddle(node* &head,int val,int checkval)
{
    node* n = new node(val);

    if(head == NULL)
    {
        cout<< "List is empty"<<endl;
        return;

    }
    node* temp = head;


    while(temp->data!=checkval && temp->next!=NULL)
    {
        temp = temp->next;
    }

    if(temp->data!= checkval && temp->next==NULL)
    {
        cout<<"Value after which the new value is to be entered cannot be found"<<endl;
        return;
```

```
	}

	if(temp->next == NULL)
	{
		n->next = NULL;
		temp->next = n;
	}
	else
	{
		n->next = temp->next;
		temp->next = n;
	}

}

void insertatFront(node* &head,int val)
{
	node* n = new node(val);

	if(head == NULL)
	{
		head = n;
		return;

	}
	head->prev = n;
	n->next=head;
	head = n;
}
void Delete(node* &head,int val)
{


	if(head == NULL)
	{
		cout<<"List is empty "<<endl;
		return;

	}
	node* temp = head;


	while(temp->data!=val && temp->next != NULL)
	{
		temp = temp->next;
	}

	if(temp->next==NULL && temp->data == val && temp->prev == NULL)
```

```
        {
                head = NULL;
                return;
        }

   if(temp->data!=val && temp->next== NULL)
   {
        cout<<"Value is not present "<<endl;
        return;
        }
        else if(temp->data == val && temp->next == NULL)
        {
                node* todelete = temp;
                temp->prev->next = NULL;
                delete todelete;
        }
        else if(temp->data == val && temp->next !=NULL)
        {
                if(temp->prev == NULL)
                {
                        node* todelete = temp;
                        head = temp->next;
                   head->prev = NULL;
                        delete todelete;
                }
                else
                {
                        node* todelete = temp;
                        temp->prev->next = temp->next;
                        delete todelete;
                }
        }

}

void displayforward(node* head)
{
        node* temp = head;

        if(temp == NULL)
        {
                cout<<"Empty"<<endl;
                return;
        }

        while(temp!=NULL)
        {
                cout << temp->data<<endl;
```

```
            temp=temp->next;
        }
}
void displaybackward(node* head)
{
        node* temp = head;
            if(temp == NULL)
        {
            cout<<"Empty"<<endl;
            return;
        }
        while(temp->next!=NULL)
        {

            temp=temp->next;
        }


        while(temp!=NULL)
        {
            cout << temp->data<<endl;
            temp=temp->prev;
        }
}




int main()
{
        int ch;
        node* head = NULL;
  cout<<"1) Insert at rear "<<endl;
  cout<<"2) Insert at front"<<endl;
  cout<<"3) Insert after a specific node"<<endl;
  cout<<"4) Displayforward "<<endl;
  cout<<"5) DisplayBackward"<<endl;
  cout<<"6) Delete"<<endl;
  cout<<"7) Exit"<<endl;
  do {
    cout<<"Enter your choice : ";
    cin>>ch;
    switch (ch) {
      case 1: {
        int a;
        cout<<"Enter your value : "<<endl;
        cin >> a;
        insertatRear(head,a);
```

```
                                break;
                        }

        case 2: {
            int a;
            cout<<"Enter your value : "<<endl;
            cin >> a;
            insertatFront(head,a);
                                break;
                        }
        case 3:
                        {
                          int a;
            cout<<"Enter the value after which you want to enter your new value : "<<endl;
            cin >> a;
            int b;
            cout<<"Enter your value : "<<endl;
            cin >> b;
            insertatmiddle(head,b,a);
                                break;
                        }
        case 4: displayforward(head);
        break;
        case 5: displaybackward(head);
        break;
        case 6:
        {
            int a;
            cout<<"Enter value to delete : "<<endl;
            cin >> a;
            Delete(head,a);
            break;
                        }

        case 7: cout<<"Exit"<<endl;
        break;
        default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=7);
return 0;
}
```

Output:-

D:\Data Structures\doublelinkedlist.exe

```
1) Insert at rear
2) Insert at front
3) Insert after a specific node
4) Displayforward
5) DisplayBackward
6) Delete
7) Exit
Enter your choice : 1
Enter your value :
4
Enter your choice : 1
Enter your value :
7
Enter your choice : 1
Enter your value :
2
Enter your choice : 1
Enter your value :
9
Enter your choice : 4
4 7 2 9
Enter your choice : 3
Enter the value after which you want to enter your new value :
7
Enter your value :
1
Enter your choice : 4
4 7 1 2 9
Enter your choice : 6
Enter value to delete :
4
Enter your choice : 4
7 1 2 9
Enter your choice :
```

Circular LinkedList:

Complexities :-
Time Complexity:- O(1)
Space Complexity:- O(n)

Algorithm:
Inserting At Beginning of the list
We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)

- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
- **Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6 -** Set **temp → next** = **newNode** and **newNode → next** = **head**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Deleting a Specific Node from the list

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next** = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a circular Linked List

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **head → data**.

Code:-

```
#include<iostream>
using namespace std;
class node{
	public:
		int data;
		node* next;


		node(int val)
		{
			data = val;
			next=NULL;
		}
};

void insertatrear(node* &head,int val)
{
	node* n = new node(val);

	if(head == NULL)
	{
		head = n;
		head->next=head;

		return;
	}
	node* temp = head;
```

```
   if(head->next == head)
   {
          head->next = n;
          n->next = head;
          return;
          }
          else
          {
                 while(temp->next != head)
                 {
                        temp = temp->next;
                 }
                 temp->next= n;
                 n->next = head;

          }
}

void insertatfront(node* &head,int val)
{
          node* n = new node(val);

          if(head == NULL)
          {
                 head = n;
                 head->next=head;

                 return;
          }
          node* temp = head;

   if(head->next == head)
   {
          n->next = head;
          head->next = n;
          head = n;
          return;
          }
          else
          {
                 while(temp->next != head)
                 {
                        temp = temp->next;
                 }

                 n->next = head;
                 temp->next=n;
                 head=n;
```

```
	}
}

void Delete(node* &head,int a)
{
	node* temp = head;
	if(temp != NULL)
	{
		if(temp == head && temp->next == head && temp->data == a)
		{
			head = NULL;
			return;
		}

		while(temp->next->data != a)
		{
			temp = temp->next;
		}
		if(temp->next == head)
		{
		node* todelete = temp->next;
		 temp->next=temp->next->next;
		 head = temp->next;
		 delete todelete;
		 return;
		}
		node* todelete = temp->next;
		temp->next = temp->next->next;
		delete todelete;

	}
	else
	{
		cout<<"List is empty "<<endl;
	}
}


void display(node* head)
{
	node* temp = head;

	if(temp == NULL)
	{
		cout<<"Empty"<<endl;
		return;
	}
```

```
  do{
          cout<<temp->data<<" ";
          temp= temp->next;

  }while(temp!=head);

  cout<<endl;
}




int main()
{
          int ch;
          node* head = NULL;


          cout<<"1) Insert at rear "<<endl;
   cout<<"2) Insert at front"<<endl;
   cout<<"3) Display "<<endl;
   cout<<"4) Delete"<<endl;
   cout<<"5) Exit"<<endl;
   do {
     cout<<"Enter your choice : ";
     cin>>ch;
     switch (ch) {
       case 1: {
          int a;
          cout<<"Enter your value : "<<endl;
          cin >> a;
          insertatrear(head,a);
                              break;
                    }

       case 2: {
          int a;
          cout<<"Enter your value : "<<endl;
          cin >> a;
            insertatfront(head,a);
                              break;
                    }
       break;
       case 3: display(head);
       break;
```

```
        case 4:
        {
            int a;
            cout<<"Enter value to delete : "<<endl;
            cin >> a;
            Delete(head,a);
            break;
                        }

        case 5: cout<<"Exit"<<endl;
        break;
        default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=6);

return 0;
}
```

Output:-

D:\Data Structures\circularlinkedlist.exe

```
1) Insert at rear
2) Insert at front
3) Display
4) Delete
5) Exit
Enter your choice : 1
Enter your value :
5
Enter your choice : 1
Enter your value :
2
Enter your choice : 1
Enter your value :
8
Enter your choice : 1
Enter your value :
3
Enter your choice : 3
5 2 8 3
Enter your choice : 1
Enter your value :
4
Enter your choice : 3
5 2 8 3 4
Enter your choice : 4
Enter value to delete :
8
Enter your choice : 3
5 2 3 4
Enter your choice :
```

Conclusion :- Successfully implemented different types of linked lists.