UNIT-III
LINKED LIST & HASHING

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast.

The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak.
- Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not reallocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically, we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner.

Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore, each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked by simply storing address of the very first node in the link field of the last node.

Comparison between array an linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

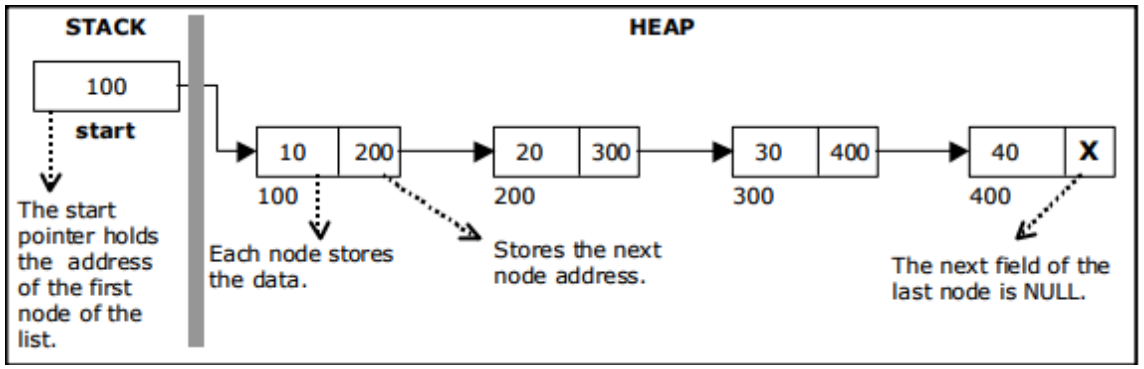
FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non-zero coefficient and exponents.
For example: P(x) = a0 Xn + a1 Xn-1 +…… + an-1 X + an
2. Represent very large numbers and operations of the large number such as addition, multiplication, and division.
3. Linked lists are to implement stack, queue, trees and graphs. 4. Implement the symbol table in compiler construction.

Single linked list:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the “start” node.



The beginning of the linked list is stored in a "start" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the start and following the next pointers.

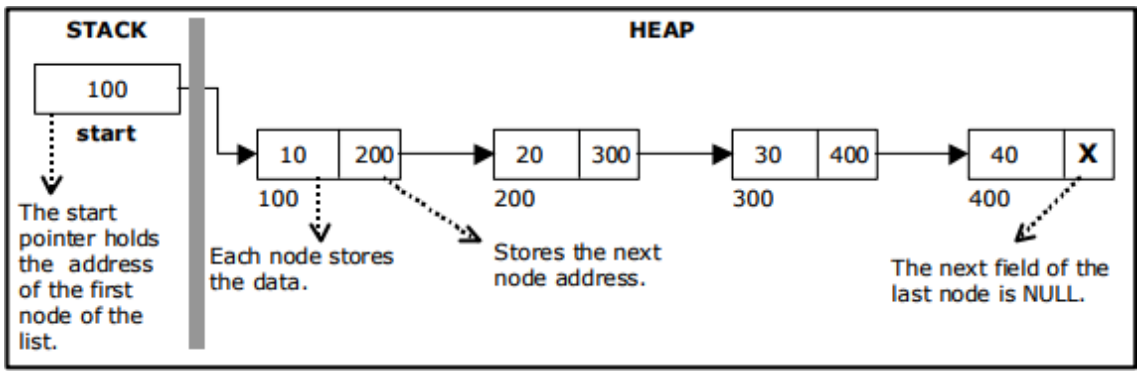
The start pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory.

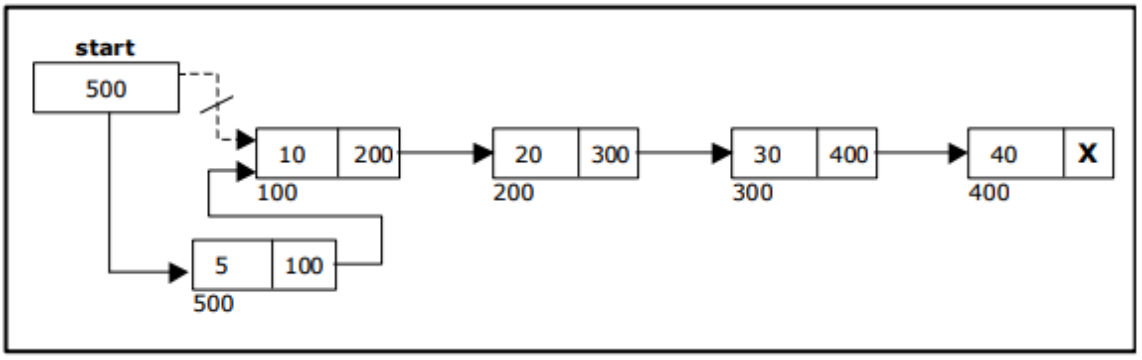


Insertion of a Node:

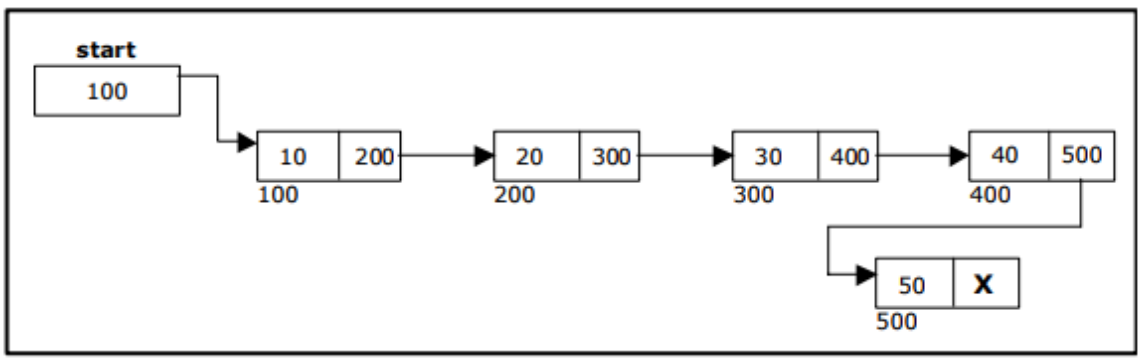
One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

- **Inserting a node at the beginning.**

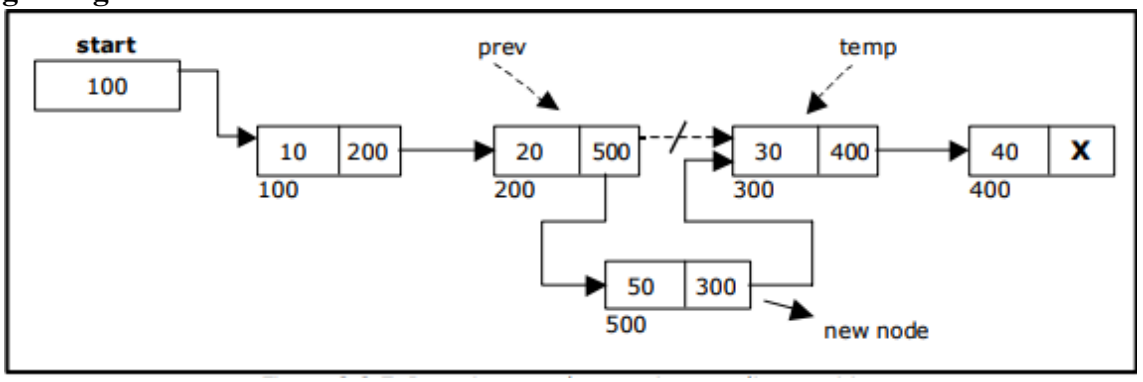


- **Inserting a node at the end:**



- **Inserting a node into the single linked list at a specified intermediate position other than**

beginning and end.

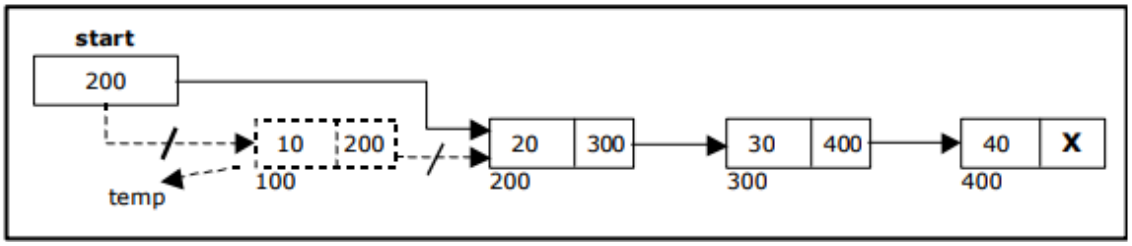


Deletion of a node:

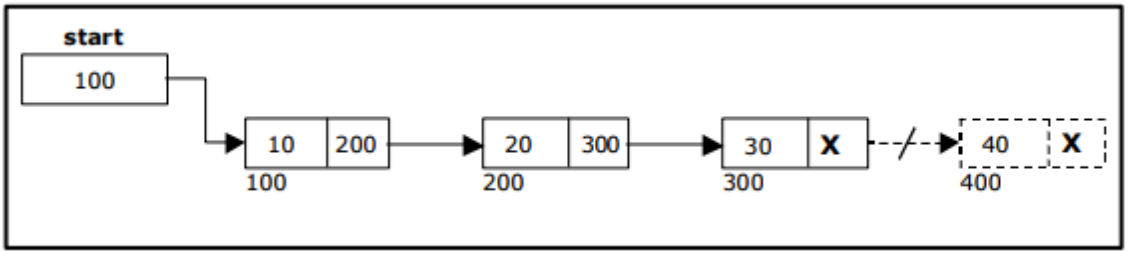
Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

- **Deleting a node at the beginning:**

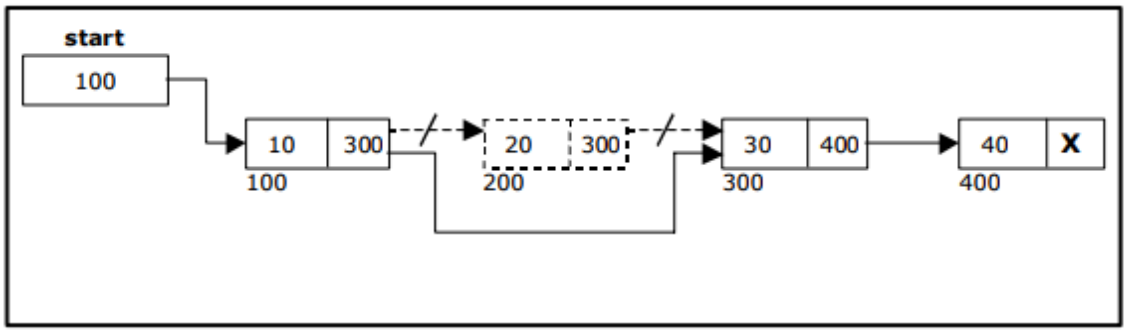


- **Deleting a node at the end:**



- **Deleting a node at Intermediate position**:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).



Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached.

Traversing a list involves the following steps:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node. The function traverse () is used for traversing and displaying the information stored in the list from left to right.

Algorithm for Traversing:

```
algorithm
1. traversing(head)
2. {
3. create Node *temp=head;
4. read temp=head;
5. if(temp==NULL) then
6. write "\nList is Empty";
7. end if;
8. while(temp!=NULL) do
9. write temp->data;
10.temp=temp->next;
11.  end of while;
12. }
```

Algorithm for Searching:

1. algorithm searching(value)
2. {
3. read pos=0,flag=0;
4. if(head==NULL) then
5. write List is Empty;
6. return;
7. endif;
8. create a Node *temp;
9. temp=head;
10. while(temp!=NULL) do
11. pos++;
12. if(temp->data==value) then
13. flag=1;
14. write element is found;
15. return;
16. endif ;

17. temp=temp->next;
18. end of while;
19. if(!flag) then
20. write element is not found;
21. endif ;
22. }

Algorithm for insertion into:

1. algorithm insertion(pos,ch,n)
2. {
3. Node *prev,*cur;
4. head=Null;
5. read prev=NULL,cur=head,count=1;
6. create a Node *temp=new Node;
7. temp->data=n;
8. temp->next=NULL;
9. write “INSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES";
10. write “Enter Your Choice:";
11. read ch;
12. switch(ch)
{

```
case 1:
   temp->next=head
   ; head=temp;
   break;
case 2:
   last->next=temp
   ; last=temp;
   break;
case 3:
   write "\nEnter the Position to Insert:";
   read pos;
   while(count!=pos)
   {
      prev=cur;
      cur=cur->next;
      count++;
   }
```

13. if(count==pos) then
14. prev->next=temp;
15. temp->next=cur;
16. else
17. write “Not Able to Insert";
18. break;
19. }

Algorithm for deletion from:

1. algorithm deletion(pos,ch)
2. Node *prev=NULL,*cur=head,count=1;

```
3.      write “DELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
        NODES";
4.     write "\nEnter Your Choice:";
5.    read ch;
6.    switch(ch)
{
case 1:
   if(head!=NULL) then
      write "\nDeleted Element is " head->data;
      head=head->next;
   else
      write "\nNot Able to Delete";
       end if;
break;

case 2:
        if(head==NULL)
        cout<<"\nNot Able to Delete";
        else
   while(cur!=last)
      prev=cur;
      cur=cur->next;
      end while;
        end if;
   if(cur==last)
      write "\nDeleted Element is: "cur->data;
      prev->next=NULL;
      last=prev;
       endif;
   break;
case 3:
        wirte “nEnter the Position of Deletion:";
        read pos;
        if(head==NULL)
        write”/nNot Able to Delete";
        else
        while(count!=pos)
         prev=cur;
        cur=cur->next;
        count++;
        end while;
        end if;
        if(count==pos)
       write "\nDeleted Element is: " cur->data;
        prev->next=cur->next;
        end if;
   break;
7. }
```

Implementation of single linked list:

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

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

   Node *head,*last;
public:
   List()
   {
      head=NULL;
      last=NULL;
   }
   void create();
   void insert();
   void delet();
   void display();
   void search();
};
void List::create()
{
   Node *temp;
   temp=new Node;
   int n;
   cout<<"\nEnter an Element:";
   cin>>n;
   temp->data=n;
   temp->next=NULL;
   if(head==NULL)
   {
      head=temp;
      last=head;
   }
    else
   {
      last->next=temp;
      last=temp;
   }
}
void List::insert()
{
   Node *prev,*cur;
   prev=NULL;
   cur=head;
   int count=1,pos,ch,n;
   Node *temp=new Node;
   cout<<"\nEnter an Element:";
   cin>>n;
   temp->data=n;
   temp->next=NULL;
   cout<<"\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES";
   cout<<"\nEnter Your Choice:";
```

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

    }
}
void List::delet()
{
    Node *prev=NULL,*cur=head;
    int count=1,pos,ch;
    cout<<"\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES";
    cout<<"\nEnter Your Choice:";
    cin>>ch;
    switch(ch)
    {
    case 1:
       if(head!=NULL)
       {
          cout<<"\nDeleted Element is "<<head->data;
          head=head->next;
       }
       else
          cout<<"\nNot Able to Delete";
       break;
    case 2:
           if(head==NULL)
           {
           cout<<"\nNot Able to Delete";
           }
```

```
            else
            {

        while(cur!=last)
        {
           prev=cur;
           cur=cur->next;
        }
        if(cur==last)
        {
           cout<<"\nDeleted Element is: "<<cur->data;
           prev->next=NULL;
           last=prev;
        }
        }
        break;
    case 3:
        cout<<"\nEnter the Position of Deletion:";
        cin>>pos;
            if(head==NULL)
            {
            cout<<"\nNot Able to Delete";
            }
            else
            {
            while(count!=pos)
            {
             prev=cur;
            cur=cur->next;
            count++;
                        }
            if(count==pos)
            {
                        cout<<"\nDeleted Element is: "<<cur->data;
            prev->next=cur->next;
            }
        }
        break;
    }
}
void List::display()
{
    Node *temp=head;
    if(temp==NULL)
    {
        cout<<"\nList is Empty";
    }
    while(temp!=NULL)
    {
        cout<<temp->data
        ; cout<<"-->";
        temp=temp->next;
    }
    cout<<"NULL";
```

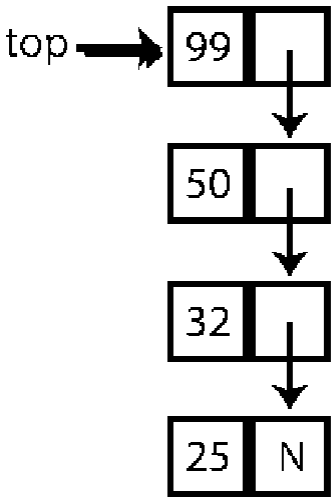
```
}
void List::search()
{
   int value,pos=0;
   bool flag=false;
   if(head==NULL)
   {
      cout<<"List is Empty";
      return;
   }
   cout<<"Enter the Value to be Searched:";
   cin>>value;
   Node *temp;
   temp=head;
   while(temp!=NULL)
   {
      pos++;
      if(temp->data==value)
      {
         flag=true;
         cout<<"Element"<<value<<"is Found at "<<pos<<" Position";
         return;
      }
      temp=temp->next;}
   if(!flag)
   {
      cout<<"Element "<<value<<" not Found in the List";
   }
}
int main()
{
   List l;
   int ch;
   while(1)
   {
      cout<<"\n**** MENU ****";
      cout<<"\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n";
      cout<<"\nEnter Your Choice:";
      cin>>ch;
      switch(ch)
      {
      case 1:
         l.c        r
         eate();
         break;
      case 2:
         l.insert();
         break;
      case 3:
         l.d        e
         let();
         break;
      case 4:
         l.search();
         break;
```

```
        case 5:
           l.display();
           break;
        case 6:
           return 0;
        }
    }
    return 0;
}
```

Linked representation of Stack:

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Implementation of stack using linked list:

```
#include <iostream>
using namespace std;
//Structure of the Node
class Node
{
public:
int data;
Node *next;
```

```
};
class Stackll: public Node
{
// top pointer to keep track of the top of the stack
Node *top = NULL;

//Function to check if stack is empty or not
public:
bool isEmpty()
{
 if(top == NULL)
 {
 return true;
 }
 else
 {
 return false;
 }
}

//Function to insert an element in stack
void push(int value)
{
  Node *temp = new Node();
  temp->data = value;
  temp->next = top;
  top = temp;
}

//Function to delete an element from the stack
void pop()
{
 if(isEmpty())
 {
  cout<<"Stack is Empty";
 }
 else
 {
  Node *temp = top;
  top = top->next;
  //delete(ptr);
 }
}
// Function to show the element at the top of the stack
void showTop()
{
 if(isEmpty())
 {
  cout<<"Stack is Empty";
 }
 else
 {
  cout<<"Element at top is : "<< top->data;
 }
```

```
}

// Function to Display the stack
void displayStack()
{
 if(isEmpty())
 {
  cout<<"Stack is Empty";
 }
 else
 {
  Node *temp=top;
  while(temp!=NULL)
  {
   cout<<temp->data<<" ";
   temp=temp->next;
  }
  cout<<"\n";
 }
 }
};
// Main function
int main()
{
Stackll sl;
 int choice,flag=1,value;
 //Menu Driven Program using Switch
 while(flag)
 {
 cout<<"\n1.Push 2.Pop 3.showTop 4.displayStack 5.exit\n";
 cin>>choice;
 switch(choice)
 {
 case 1: cout<<"Enter Value:\n";
        cin>>value;
        sl.push(value);
        break;
 case 2: sl.pop();
        break;
 case 3: sl.showTop();
        break;
 case 4: sl.displayStack();
        break;
 case 5: flag=0;
        break;
 }
 }

return 0;
}
```

Linked representation of Queue:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example:

In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

Implementation of queue using linked list:

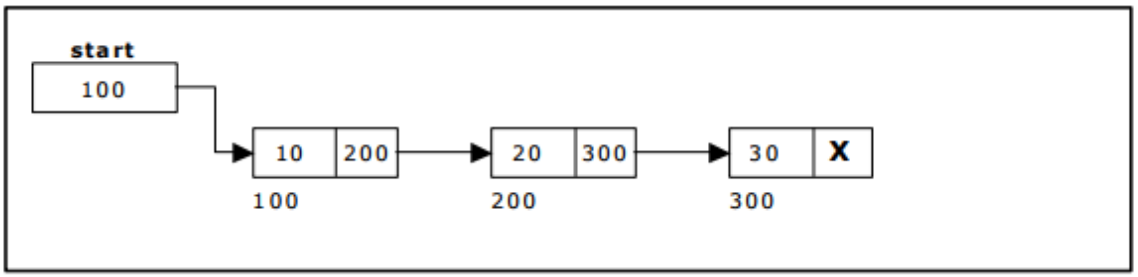
```
#include <iostream>
using namespace std;

// Structure of Node.
class Node
{
public:
int data;
Node *next;
};
class Queuell : public Node
{
Node *front,*rear;
public: Queuell()
{
front = NULL;
rear = NULL;
}
//Function to check if queue is empty or not
public:
bool isEmpty()
{
 if(front == NULL && rear == NULL)
 {
 return true;
 }
 else
 {
 return false;
 }
}
```

```
//function to enter elements in queue
void enqueue(int value)
{
 Node *temp = new Node();
 temp->data= value;
 temp->next = NULL;

 //If inserting the first element/node
 if(front == NULL)
 {
  front = temp;
  rear = temp;
 }
 else
 {
  rear ->next = temp;
  rear = temp;
 }
}

//function to delete/remove element from queue
void dequeue()
{
 if(isEmpty())
 {
 cout<<"Queue is empty\n";
 }
 else
 //only one element/node in queue.
 if( front == rear)
 {
  //free(front);
  cout<<"deleted element is:"<<front->data;
  front = rear = NULL;
 }
 else
 {
  Node *temp = front;
  cout<<"deleted element is:"<<front->data;
  front = front->next;
  //free(ptr);
 }
}

//function to show the element at front
void showfront( )
{
 if( isEmpty())
 cout<<"Queue is empty\n";
 else
 cout<<"element at front is:"<<front->data;
}

//function to display queue
```

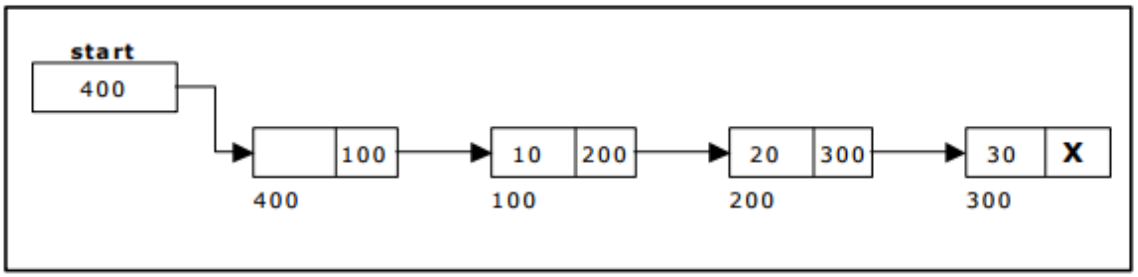
```
void displayQueue()
{
 if (isEmpty())
 cout<<"Queue is empty\n";
 else
 {
  Node *temp = front;
  while( temp !=NULL)
  {
   cout<<temp->data<<" ";
   temp= temp->next;
  }
 }
}
};
//Main Function
int main()
{
Queuell ql;
 int choice, flag=1, value;
 while( flag == 1)
 {
  cout<<"\n1.enqueue 2.dequeue 3.showfront 4.displayQueue 5.exit\n";
  cin>>choice;
  switch (choice)
  {
  case 1: cout<<"Enter Value:\n";
        cin>>value;
        ql.enqueue(value);
        break;
  case 2: ql.dequeue();
        break;
  case 3: ql.showfront();
        break;
  case 4: ql.displayQueue();
        break;
  case 5: flag = 0;
        break;
  }
 }

 return 0;
}
```

Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Single Linked List without a header node



Single Linked List with header node

Note that if your linked lists do include a header node, there is no need for the special case code given above for the remove operation; node n can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node n.

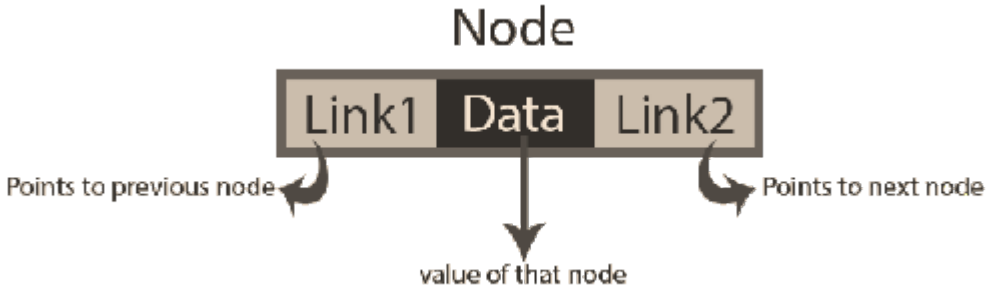
Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

Double Linked List:

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

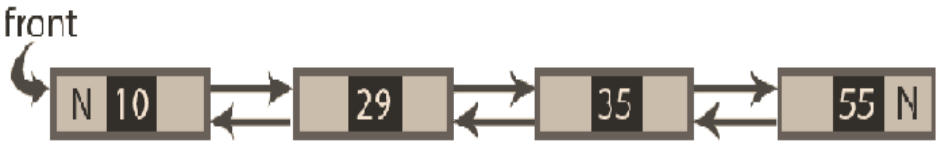
"Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence"

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

Example:



- In double linked list, the first node must be always pointed by head.
- Always the previous field of the first node must be NULL.
- Always the next field of the last node must be NULL.

Operations on Double Linked List:

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

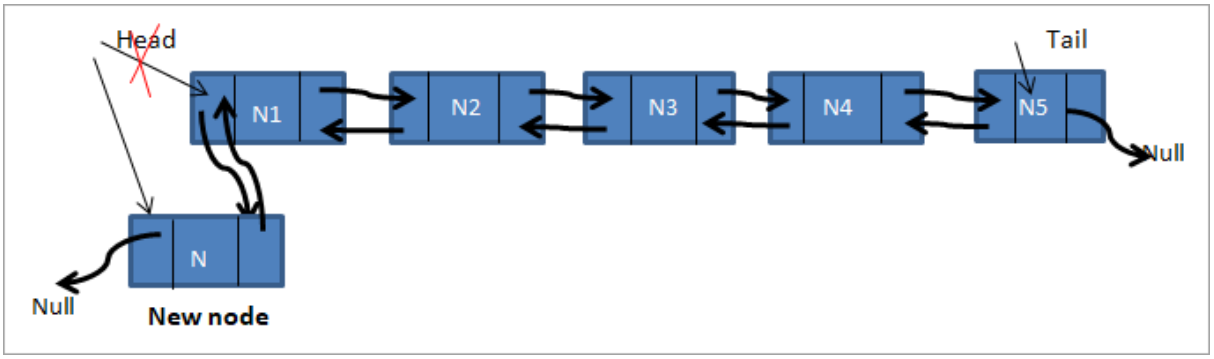
1. Insertion:

In a double linked list, the insertion operation can be performed in three ways as follows...

Inserting At Beginning of the list
Inserting At End of the list
Inserting At Specific location in the list

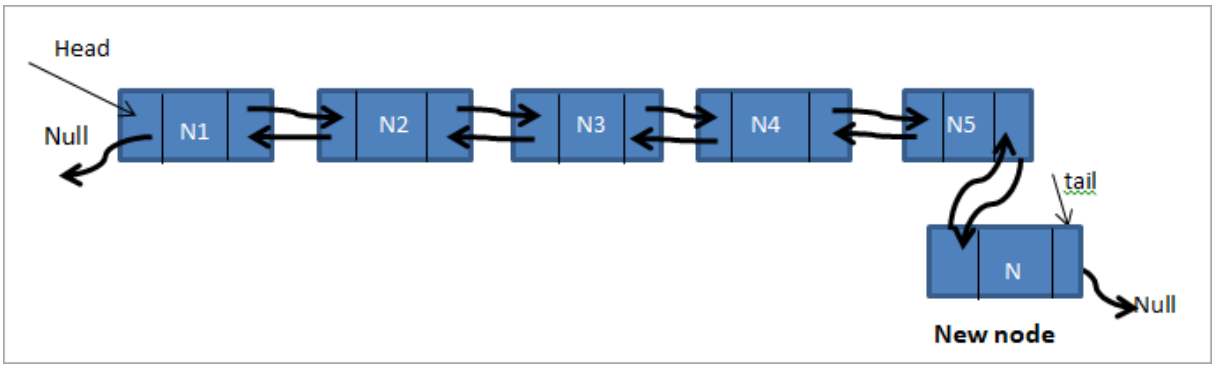
Inserting At Beginning of the list:

Insertion of a new node at the front of the list is shown above. As seen, the previous new node N is set to null. Head points to the new node. The next pointer of N now points to N1 and previous of N1 that was earlier pointing to Null now points to N.



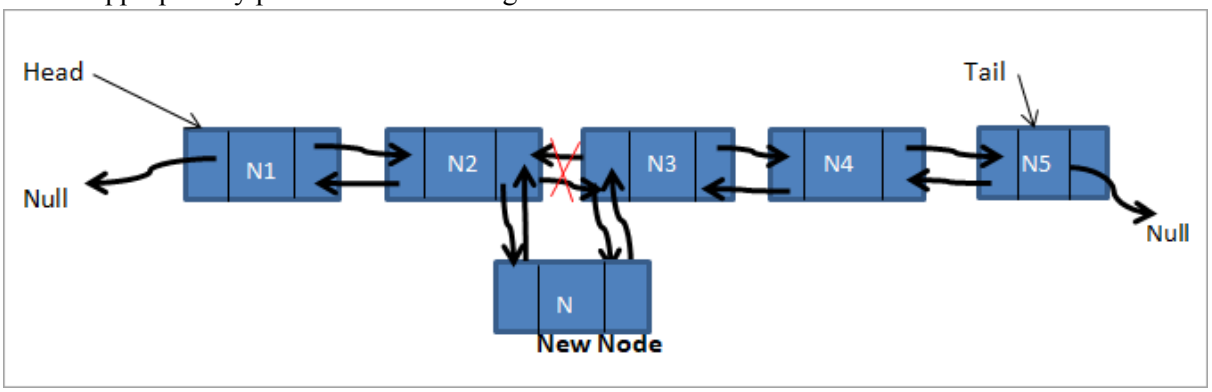
Inserting At End of the list:

Inserting node at the end of the doubly linked list is achieved by pointing the next pointer of new node N to null. The previous pointer of N is pointed to N5. The 'Next' pointer of N5 is pointed to N.



Inserting At Specific location in the list:

When we have to add a node before or after a particular node, we change the previous and next pointers of the before and after nodes so as to appropriately point to the new node. Also, the new node pointers are appropriately pointed to the existing nodes.



2. Deletion:

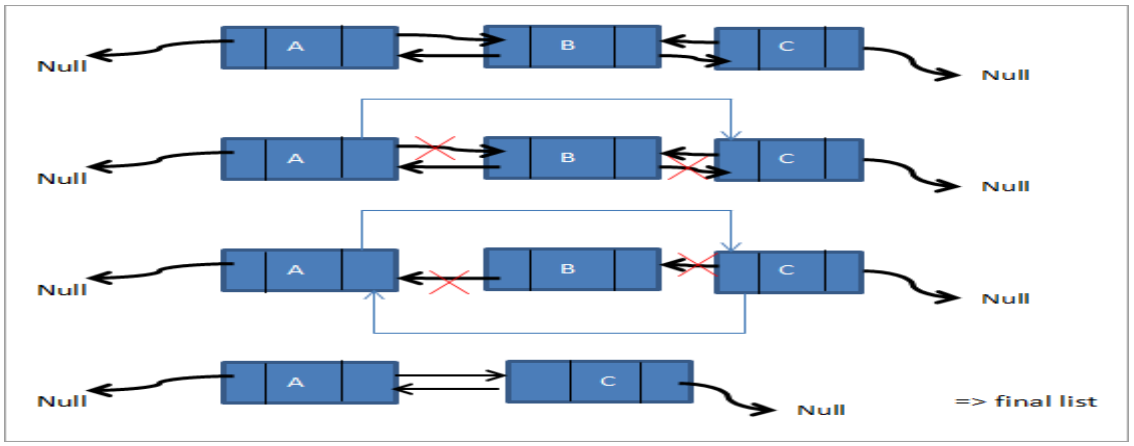
A node can be deleted from a doubly linked list from any position like from the front, end or any other given position or given data. When deleting a node from the doubly linked list, we first reposition the pointer pointing to that particular node so that the previous and after nodes do not have any connection to the node to be deleted. We can then easily delete the node.

In a double linked list, the deletion operation can be performed in three ways as follows...

Deleting from Beginning of the list
Deleting from End of the list
Deleting a Specific Node

The deletion of node B from the given linked list. The sequence of operation remains the same even if the node is first or last. The only care that should be taken is that if in case the first node is deleted, the second node's previous pointer will be set to null.

Similarly, when the last node is deleted, the next pointer of the previous node will be set to null. If in between nodes are deleted, then the sequence will be as above.



Inserting At Beginning of the list:

We can use the following steps to insert a new node at beginning of the double linked 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**.
Step4 - If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list:

We can use the following steps to insert a new node at end of the double linked 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):

We can use the following steps to insert a new node after a node in the double 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, 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 from Beginning of the list:

We can use the following steps to delete a node from beginning of the double linked 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 - Check whether list is having only one node (**temp** → **previous** is equal to **temp** → **next**)
Step 5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
Step 6 - If it is **FALSE**, then assign **temp** → **next** to **head**, **NULL** to **head** → **previous** and delete **temp**.

Deleting from End of the list:

We can use the following steps to delete a node from end of the double linked 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 - Check whether list has only one Node (**temp** → **previous** and **temp** → **next** both are **NULL**)
Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp** → **next** is equal to **NULL**)
Step 7 - Assign **NULL** to **temp** → **previous** → **next** and delete **temp**.

Deleting a Specific Node from the list:

We can use the following steps to delete a specific node from the double linked 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 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 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:

We can use the following steps to display the elements of 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**).

Implementation:

```
#include<iostream>
using namespace std;
class Node
{
public:
   int data;
   Node *prev,*next;
};
class List:public Node
{

   Node *head,*last;
public:
   List()
   {
      head=NULL;
      last=NULL;
   }
   void create();
   void insert();
```

```
    void delet();
    void display();
    void search();
};
void List::create()
{
    Node *temp;
    temp=new Node;
    int n;
    cout<<"\nEnter an Element:";
    cin>>n;
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    if(head==NULL)
    {
        head=temp;
        last=head;
    }
     else
    {
        last->next=temp;
        temp->prev=last;
        last=temp;
    }
}
void List::insert()
{
    Node *old,*cur;
    old=NULL;
    cur=head;
    int count=1,pos,ch,n;
    Node *temp=new Node;
    cout<<"\nEnter an Element:";
    cin>>n;
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    cout<<"\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES";
    cout<<"\nEnter Your Choice:";
    cin>>ch;
    switch(ch)
    {
    case 1:
        temp->next=head;
        head->prev=temp
        ; head=temp;
        break;
    case 2:
        last->next=temp;
        temp->prev=last;
        last=temp;
        break;
    case 3:
```

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

    }
}
void List::delet()
{
    Node *old=NULL,*cur=head;
    int count=1,pos,ch;
    cout<<"\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES";
    cout<<"\nEnter Your Choice:";
    cin>>ch;
    switch(ch)
    {
    case 1:
        if(head==NULL)
        {
                        cout<<"\nNot Able to Delete";
              }
              else
              {
         cout<<"\nDeleted Element is "<<head->data;
         if(head==last)
         {
         head=last=NULL;
         }
         else
         {
         Node *temp;
         temp=head;
         head=head->next;
         head->prev=NULL;
         }
        }
                                                    break;
    case 2:
                                                    if(head==NULL)
```

```
{
cout<<"\nNot Able to Delete";
```

```
            }
            else
            {

      while(cur!=last)
      {
         old=cur;
         cur=cur->next;
      }
      if(cur==last)
      {
         cout<<"\nDeleted Element is: "<<cur->data;
         if(old==NULL)
         {
         head=NULL;
         }
         else
         {
         old->next=NULL;
         last=old;
         }
       }
      }
      break;
  case 3:
      cout<<"\nEnter the Position of Deletion:";
      cin>>pos;
            if(head==NULL)
            {
            cout<<"\nNot Able to Delete";
            }
            else
            {
            while(count!=pos)
            {
                        old=cur;
            cur=cur->next;
            count++;
                         }
            if(count==pos)
            {
                        cout<<"\nDeleted Element is: "<<cur->data;
            old->next=cur->next;
            (cur->next)->prev=old;
            }
      }
      break;
   }
}
void List::display()
{
   Node *temp=head;
   if(temp==NULL)
   {
```

```
cout<<"\nList is Empty";
```

```
    }
    while(temp!=NULL)
    {
        cout<<temp->data
        ; cout<<"-->";
        temp=temp->next;
    }
    cout<<"NULL";
}
void List::search()
{
    int value,pos=0;
    bool flag=false;
    if(head==NULL)
    {
        cout<<"List is Empty";
        return;
    }
    cout<<"Enter the Value to be Searched:";
    cin>>value;
    Node *temp;
    temp=head;
    while(temp!=NULL)
    {
        pos++;
        if(temp->data==value)
        {
            flag=true;
            cout<<"Element"<<value<<"is Found at "<<pos<<" Position";
            return;
        }
        temp=temp->next;
    }
    if(!flag)
    {
        cout<<"Element "<<value<<" not Found in the List";
    }
}
int main()
{
    List l;
    int ch;
    while(1)
    {
        cout<<"\n**** MENU ****";
        cout<<"\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n";
        cout<<"\nEnter Your Choice:";
        cin>>ch;
        switch(ch)
        {
        case 1:
            l.c         r
            eate();
            break;
```

```
case 2:
   l.insert();
```

```
            break;
        case 3:
            l.d         e
            let();
            break;
        case 4:
            l.search();
            break;
        case 5:
            l.display();
            break;
        case 6:
            return 0;
        }
    }
    return 0;
}
```

Circular Linked List:

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list. A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

Example:

Operations:
In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program.
Step 2 - Declare all the **user defined** functions.
Step 3 - Define a **Node** structure with two members **data** and **next**
Step 4 - Define a Node pointer '**head**' and set it to **NULL**.

Step 5 - Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion:

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

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**.

Deletion:

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list:

We can use the following steps to delete a node from beginning of the circular linked 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 both **'temp1'** and **'temp2'** with **head**.
Step 4 - Check whether list is having only one node (**temp1 → next == head**)
Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head**)
Step 7 - Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Deleting from End of the list:

We can use the following steps to delete a node from end of the circular linked 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 - Check whether list has only one Node (**temp1 → next == head**)
Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**. Then, set **'temp2 = temp1 '** and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)
Step 7 - Set **temp2 → next** = **head** and delete **temp1**.

Deleting a Specific Node from the list:

We can use the following steps to delete a specific node from the circular linked 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 11 - 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:
We can use the following steps to display the elements of 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**.

Implementation:

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

   Node *head,*last;
public:
   List()
   {
      head=NULL;
      last=NULL;
   }
   void create();
   void insert();
   void delet();
   void display();
   void search();
};
void List::create()
{
   Node *temp;
   temp=new Node;
   int n;
   cout<<"\nEnter an Element:";
   cin>>n;
   temp->data=n;
```

```
    temp->next=NULL;
    if(head==NULL)
    {
       head=temp;
       temp->next=head;
       last=head;
    }

    else
    {
            temp->next=last->next;
       last->next=temp;
       last=temp;
    }
}
void List::insert()
{
    Node *prev,*cur;
    prev=NULL;
    cur=head;
    int count=1,pos,ch,n;
    Node *temp=new Node;
    cout<<"\nEnter an Element:";
    cin>>n;
    temp->data=n;
    temp->next=NULL;
    cout<<"\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES";
    cout<<"\nEnter Your Choice:";
    cin>>ch;
    switch(ch)
    {
    case 1:
       temp->next=head;
       head=temp;
       last->next=head;
       break;
    case 2:
       temp->next=last->next
       ; 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)
       {
          temp->next=prev->next;
          prev->next=temp;
```

```
        }
        else
           cout<<"\nNot Able to Insert";
        break;

    }
}
void List::delet()
{
    Node *prev=NULL,*cur=head;
    int count=1,pos,ch;
    cout<<"\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES";
    cout<<"\nEnter Your Choice:";
    cin>>ch;
    switch(ch)
    {
    case 1:
        if(head!=NULL)
        {
                        if(head==last)
                        {
                           cout<<"\nDeleted Element is "<<head->data;
                           head=NULL;
                        }
                                else
                                {
                                                                        cout
                                                                    <<"\nDelet
                                                                    ed Element
                                                                    is
        else                                                        "<<head->
                                                                    data;
                                                                    head=head-
                                                                    >next;
                                                                  }
                                                                 }
           cout<<"\nNot Able to Delete";
        break;
    case 2:
              if(head==NULL)
              {
              cout<<"\nNot Able to Delete";
              }
              else if(last==head)
              {
              cout<<"\nDeleted Element is "<<head->data;
                              head=NULL;
                        }
              else
              {

        while(cur!=last)
        {
           prev=cur;
```

```
    cur=cur->next;
}
if(cur==last)
{
    cout<<"\nDeleted Element is: "<<cur->data;
    prev->next=head;
```

```
            last=prev;
        }
        }
        break;
    case 3:
        cout<<"\nEnter the Position of Deletion:";
        cin>>pos;
                if(head==NULL)
                {
                cout<<"\nNot Able to Delete";
                }
                else
                {
                while(count!=pos)
                {
                 prev=cur;
                cur=cur->next;
                count++;
                                }
                if(count==pos)
                {
                                cout<<"\nDeleted Element is: "<<cur->data;
                prev->next=cur->next;
                }
        }
        break;
    }
}
void List::display()
{
    Node *temp=head;
   if(temp==NULL)
    {
        cout<<"\nList is Empty";
    }
    else
    {                                                                   while(temp!=last)
                                                                        {
                                cout<<temp->data;
                                cout<<"-->";
                                temp=temp->next;
                        }
                        cout<<last->data;
                }
}
void List::search()
{
    int value,pos=0;
    bool flag=false;
    if(head==NULL)
    {
        cout<<"List is Empty";
        return;
    }
```

```
    cout<<"Enter the Value to be Searched:";
    cin>>value;
    Node *temp;
    temp=head;
    do
    {
       pos++;
       if(temp->data==value)
       {
          flag=true;
          cout<<"Element"<<value<<"is Found at "<<pos<<" Position";
          return;
       }
       temp=temp->next;
    }while(temp!=head);
    if(!flag)
    {
       cout<<"Element "<<value<<" not Found in the List";
    }
}
int main()
{
    List l;
    int ch;
    while(1)
    {
       cout<<"\n**** MENU ****";
       cout<<"\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n";
       cout<<"\nEnter Your Choice:";
       cin>>ch;
       switch(ch)
       {
       case 1:
          l.c        r
          eate();
          break;
       case 2:
          l.insert();
          break;
       case 3:
          l.d        e
          let();
          break;
       case 4:
          l.search();
          break;
       case 5:
          l.display();
          break;
       case 6:
          return 0;
       }
    }
    return 0;
}
```

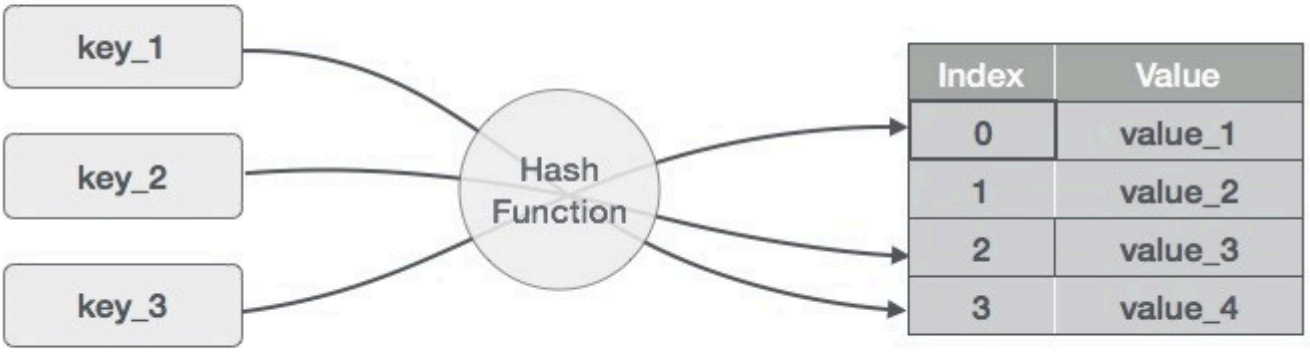
DATA STRUCTURE - HASH TABLE

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We are going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and following items are to be stored. Items are in key, value format.



- 1, 20
- 2, 70
- 42, 80
- 4, 25
- 12, 44
- 14, 32
- 17, 11
- 13, 78
- 37, 98

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12

6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are basic primary operations of a hashtable which are following.

Search − search an element in a hashtable. Insert − insert an element in a hashtable. delete − delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem { int data;
int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){ return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem * search(int key){
//get the hash
int hashIndex = hashCode(key);
```

```
//move in array until an empty while(hashArray[hashIndex] != NULL){

if(hashArray[hashIndex]->key == key) return hashArray[hashIndex];

//go to next cell
++hashIndex;

//wrap around the table hashIndex %= SIZE;
}

return NULL;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
struct DataItem * item = (struct DataItem*) malloc(sizeof(struct DataItem)); item->data = data;
item->key = key;

//get the hash
int hashIndex = hashCode(key);

//move in array until an empty or deleted cell while(hashArray[hashIndex] != NULL &&
hashArray[hashIndex]->key != -1){
//go to next cell
++hashIndex;

//wrap around the table hashIndex %= SIZE;
}

hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
struct DataItem* delete(struct DataItem* item){ int key = item->key;

//get the hash
int hashIndex = hashCode(key);

//move in array until an empty while(hashArray[hashIndex] !=NULL){

if(hashArray[hashIndex]->key == key){
struct DataItem* temp = hashArray[hashIndex];
```

```
//assign a dummy item at deleted position hashArray[hashIndex] = dummyItem;

return temp;
}

//go to next cell
++hashIndex;

//wrap around the table hashIndex %= SIZE;
}

return NULL;
}
```