linked list in c++

A linked list is a linear data structure in which each element is a separate object, known as a node.

A **linked list** is a dynamic data structure that consists of a sequence of nodes. Each node contains two parts:

1. **Data**: The value stored in the node.
2. **Pointer**: A reference (or pointer) to the next node in the sequence.

Types of Linked Lists:

1. **Singly Linked List**: Each node points to the next node.
2. **Doubly Linked List**: Each node has two pointers: one to the next node and one to the previous node.
3. **Circular Linked List**: The last node points back to the first node.

Node Structure :

```
struct Node {
   int data; // or any other data type
   Node* next; // pointer to the next node
};
```

Creating a Linked List

To create a linked list, you need to:

- Allocate memory for each node using `new` **or** `malloc`**.**
- **Initialize the** `data` **field of each node.**
- **Set the** `next` **field of each node to point to the next node in the list.**

Basic Operations

- Insertion: Insert a new node at a specific position in the list.
- Deletion: Remove a node from the list.
- Traversal: Iterate through the nodes in the list.

Reasons to Use a Linked List Over Arrays or Vectors :

Dynamic Size:

- **Linked List**: The size of a linked list is dynamic, meaning you can easily add or remove elements without worrying about memory reallocation. A linked list can grow or shrink as needed without the need to define an initial capacity.
- **Array**: Arrays have a fixed size. If you need more elements than the array can hold, you'll have to create a larger array and copy over the data, which is costly in terms of both time and memory.
- **Vector**: Vectors in C++ offer dynamic resizing, but resizing can involve expensive operations like copying and reallocating memory.

When to prefer linked lists: If your application requires frequent insertions or deletions at unknown times or unpredictable memory usage.

Efficient Insertions/Deletions:

- **Linked List**: Insertions and deletions are generally more efficient (especially at the beginning or middle). Inserting a node takes constant time (O(1)) if you have a pointer to the location, and there is no need to shift other elements like in arrays.
- **Array/Vector**: Inserting or deleting elements, especially in the middle of an array or vector, can be costly because all the subsequent elements have to be shifted. In the worst case (inserting or deleting at the beginning), this operation takes O(n) time.

When to prefer linked lists: If you need to frequently insert or delete elements at arbitrary positions (especially at the beginning or middle of the list).

Efficient Insertions/Deletions:

- **Linked List**: Insertions and deletions are generally more efficient (especially at the beginning or middle). Inserting a node takes constant time (O(1)) if you have a pointer to the location, and there is no need to shift other elements like in arrays.
- **Array/Vector**: Inserting or deleting elements, especially in the middle of an array or vector, can be costly because all the subsequent elements have to be shifted. In the worst case (inserting or deleting at the beginning), this operation takes O(n) time.

When to prefer linked lists: If you need to frequently insert or delete elements at arbitrary positions (especially at the beginning or middle of the list).

Memory Efficiency (For Large, Dynamic Structures):

- **Linked List**: Linked lists do not require a large contiguous block of memory. They allocate memory for each node as needed, which can be helpful when memory is fragmented or unpredictable.
- **Array**: Arrays require a contiguous block of memory, which can be problematic for very large arrays, especially in systems with limited memory.
- **Vector**: Vectors also need contiguous memory, though they try to manage reallocation efficiently.

When to prefer linked lists: When you have memory fragmentation or cannot predict the size of your data set, and need to avoid memory reallocation.

No Wasted Space:

- **Linked List**: Linked lists allocate exactly the amount of memory needed for each element plus a pointer to the next node. There's no wasted space unless you're concerned about the extra memory for the pointers.
- **Array**: Arrays require you to declare the size upfront, which can result in over-allocating or under-allocating space.
- **Vector**: Vectors handle resizing, but when they grow, they allocate more space than needed (usually doubling their size), which can result in unused space.

When to prefer linked lists: If you're concerned about efficient memory use when the number of elements fluctuates significantly.

singly linked list

Implementing singly linked list :

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

class Node  // Node class to represent a node of the linked list
{
   public:
   int data;      // Data field
   Node* next;    // Pointer to the next node

   Node(int data) // Creating a constructor to initialize a node
   {
      this->data=data;
      this->next=NULL;
   }
};
```

```
int main()
{
   Node* node1=new Node(10);  //Creating an object of node class dynamically using
new keyword
   cout<<node1->data<<endl;   //Displaying data members using array (->) operator .
   cout<<node1->next;
 }
```

NOTE: Here node1 is a pointer that stores the address of node1 node.
node1 contains two parts 1.data and 2.next
node1->data returns the value of a node and
node1->next returns the address of the next node
And if we try to print node1 like (cout<<node1) ,
Then it will print the address of node1 node .

Inserting data at beginning in singly linked list :

```
void insertAtBeginning(Node* &head,int newData)
{
   Node* newNode=new Node(newData);
   newNode ->next=head;
   head = newNode;
}
```

```
 //Here we are passing reference of head   (&head)  node because we want to change
the actual linked list .(don't want to make a copy)
```

```
int main()
{
   Node* node1=new Node(10);         //Created 1st node
   Node* head=node1;                 //Naming 1st node as head node
   insertAtBeginning(head,20);
   print(head);          //Print function is written on the next page
   return 0;
}
```

Printing the whole linked list

```
void print(Node* &head)
{
   Node* temp=head;  //Created a temporary node because we don't want to change

   //temp is a pointer that stores the address of 1st node (head)

   if(head==NULL)
   {
      cout<<"List is empty";
   }
   while(temp!=NULL)
   {
      cout<<temp->data<<" ";
      temp=temp->next;                    //temp->next stores the address of the next node .
   }
   cout<<endl;
}
```

NOTE : one can move forward from one node to the next node like :
temp=temp->next;
And can print the value at that next node like
cout<<temp->data;

Insert data at ending (tail) in singly linked list :

```
void insertAtTail(Node* &tail , int newData)
{
   Node* newNode=new Node(newData);
   tail ->next=newNode;
   tail = newNode;
}
```

```
int main()
{
    Node* node1=new Node(10);  //Created 1st node
    Node* head=node1;          //Naming 1st node as head node
    Node* tail=node1;
    insertAtTail(tail,20);
    print(head);
    insertAtTail(tail,30);
    print(head);
    return 0;
}
```

Insert data at any position in singly linked list :

```
void insertAtPosition(Node* &head , Node* &tail ,int position ,int newData)
{
    //insert at Start
    if(position == 1)
    {
        insertAtBeginning(head, newData);
        return;
    }

   Node* temp=head;
   int cnt=1;

   if(cnt<position-1)
   {
      temp=temp->next;
      cnt++;
   }

    //inserting at Last Position
    if(temp -> next == NULL) {
        insertAtTail(tail,newData);
        return ;
    }
```

```
    //Creating a new node
    Node* newNode=new Node(newData);

    //Inserting data at any middle position
    newNode->next=temp->next;
    temp->next=newNode;
}


int main()
{
    Node* node1=new Node(10);  //Created 1st node
    Node* head=node1;         //Naming 1st node as head node
    Node* tail=node1;
    insertAtTail(tail,20);
    insertAtTail(tail,30);
    insertAtTail(tail,40);

    cout<<"Linked list data before insertion :";
    print(head);

    cout<<"Linked list data after insertion :";
    insertAtPosition(head,tail ,3,21);
    print(head);

    //The values of head and tail must be correct after insertion
    cout<<"Head node :"<<head->data<<endl;
    cout<<"Tail node :"<<tail->data<<endl;

    return 0;
 }
```

Deleting a node in singly linked list :

```
void deleteAnyNode(Node* &head,int position)
{
  if(position==1)
  {
   //   Deleting first node
      Node* temp=head;
      head=head->next;
   //   Free memory of first node
       temp->next=NULL;
       delete temp;
  }
  else
  {
   //   Deleting any middle node
       int cnt=1;
       Node* previousNode=NULL;
       Node* current=head;
       while(cnt<position)
       {
           previousNode=current;
           current=current->next;
           cnt++;
       }

       Node* nextNode=current->next;
       previousNode->next=nextNode;

    //   Free memory of any middle node
       current->next=NULL;
       delete current;
  }
}
```

Doubly linked list

A **doubly linked list** is a type of data structure that consists of a sequence of elements (called nodes), where each node contains three parts:

1. **Data** - the actual value stored in the node.
2. **Next pointer** - a reference to the next node in the sequence.
3. **Previous pointer** - a reference to the previous node in the sequence.

In contrast to a singly linked list, where each node only knows about its next node, a doubly linked list allows traversal in both directions—forward and backward—since each node maintains references to both its previous and next nodes.

Implementing doubly linked list :

```
#include <iostream>

using namespace std;

class Node

{

   public:

   int data;

   Node* next;  //For next node

   Node* prev;  //For previous node

   Node(int data)

   {

      this->data=data;

      this->next=NULL;
```

```
        this->prev=NULL;
    }
};
```

Printing the whole doubly linked list

```
void print(Node* &head)
{
    Node* temp=head;
    while(temp !=NULL)
    {
        cout<<temp->data<<endl;
        temp=temp->next;
    }
    cout<<endl;
}
```

Length of doubly linked list

```
int getLength(Node* &head)
{
    int len=0;
    Node* temp=head;
    while(temp!=NULL)
    {
        len++;
```

```
        temp=temp->next;
    }
    return len;
}
int main()
{
    Node* node1=new Node(10);
    Node* head=node1;
    print(head);
    int len=getLength(head);
    cout<<"Length of a given linked list is :"<<len<<endl;
    return 0;
}
```

Inserting data at beginning in doubly linked list :

```
void insertAtHead(Node* &head , Node* &tail , int newData)
{
    Node* temp=new Node(newData);
    if(head==NULL)
    {
       head=temp;
       tail=temp;
    }
    else
    {
        temp->next=head;
        head->prev=temp;
        head=temp;
    } }
```

```
int main()
{
   Node* node1=new Node(10);
   Node* head=node1;
   Node* tail=node1;

   insertAtHead(head,tail,20);
   print(head);
   return 0;
}
```

Insert data at ending (tail) in doubly linked list :

```
void insertAtTail(Node* &head , Node* &tail , int newData)
{
   Node* temp=new Node(newData);
   if(tail==NULL)
   {
      tail=temp;
      head=temp;
   }
   else
   {
       tail->next=temp;
       temp->prev=tail;
       tail=temp;
   }
}
int getLength(Node* &head)
{
    int len=0;
    Node* temp=head;
   while(temp!=NULL)
   {
       len++;
       temp=temp->next;
   }
   return len;  }
```

```
int main()
{
    Node* node1=new Node(10);
    Node* head=node1;
    Node* tail=node1;

    insertAtTail(head,tail,20);
    print(head);
    return 0;
}
```

Insert data at any position in doubly linked list :

```
void insertAtPosition(Node* & head, Node* &tail, int position, int newData) {

    //insert at Start
    if(position == 1) {
        insertAtHead(tail,head, newData);
        return;
    }

    Node* temp  = head;
    int cnt = 1;

    while(cnt < position-1) {
        temp = temp->next;
        cnt++;
    }

    //inserting at Last Position
    if(temp -> next == NULL) {
        insertAtTail(tail,head,newData);
        return ;
    }

    //creating a node for d
    Node* newNode = new Node(newData);
```

```
    newNode ->next = temp -> next;
    temp -> next -> prev = newNode;
    temp -> next = newNode;
    newNode -> prev = temp;
}
```

Deleting a node in doubly linked list :

```
void deleteANode(Node* &head,int position)
{
    if(position==1)
    {
        Node* temp=head;
        temp->next->prev=NULL;
        head=temp->next;
        temp->next=NULL;
        // Deleting a node
            delete temp;
    }
    else
    {
        int cnt=1;
        Node* current=head;
        Node* previous=NULL;
        while(cnt<position)
        {
            previous=current;
            current=current->next;
            cnt++;
        }
        current->prev=NULL;
        previous->next=current->next;
        current->next=NULL;
        // Deleting a node
            delete current;
    }
}
```

Circular linked list

A **circular linked list** is a variation of the linked list in which the last node points back to the first node, forming a circular structure. This can be either a **singly circular linked list** or a **doubly circular linked list**:

In a **singly circular linked list**, each node contains data and a pointer to the next node. The last node points back to the first node instead of `nullptr`.

In a **doubly circular linked list**, each node contains three parts: data, a pointer to the next node, and a pointer to the previous node. The last node points back to the first node, and the first node's previous pointer points to the last node.

Advantages of Circular Linked List:

- It allows continuous traversal from any node.
- Useful in applications that require looping through the list multiple times without resetting to the head.

In CLL there is no need to use head node , we can access both head and tail node using tail node . because the head is next to the tail .

Inserting data in circular linked list:

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

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

   Node(int data)
   {
```

```
        this->data=data;
        this->next=NULL;
    }
};

void insertANode(Node* &tail , int element ,int data) //here data is the new inserting data and element is the list item after which we are inserting newNode
{
    //If List is empty
    if(tail == NULL)
    {
        Node* newNode=new Node(data);
        tail=newNode;
        newNode->next=newNode;
    }
    else
    {
        //For non-empty list
        //assuming the element is present in the list
        Node* temp=tail;
        while(temp->data!=element)
        {
            temp=temp->next;
        }

        //element found
        Node* newNode2=new Node(data);
        newNode2->next=temp->next;
        temp->next=newNode2;
    }
}
```

Printing circular linked list :

```
void print(Node* &tail)
{
    //For an empty list
    if(tail==NULL)
    {
        cout<<"List is empty";
    }
    Node* temp=tail; //Storing tail address into temp;
    do
    {
        cout<<tail->data<<" ";
        tail=tail->next;
    }
    while(tail!=temp);
    cout<<endl;
}

int main()
{
    Node* tail=NULL;

    //Inserting in an empty list
    insertANode(tail,1,2);

    print(tail);

    //Inserting in an non-empty list
    insertANode(tail,2,3);
    insertANode(tail,3,4);
    insertANode(tail,4,5);

    print(tail);

    insertANode(tail,3,6);
    print(tail);

    return 0;   }
```

Deleting a node from circular linked list :

```
void deleteNode(Node* &tail, int value) {

    //empty list
    if(tail == NULL) {
        cout << " List is empty, please check again" << endl;
        return;
    }
    else{
        //non-empty

        //assuming that "value" is present in the Linked List
        Node* prev = tail;
        Node* curr = prev -> next;

        while(curr -> data != value) {
            prev = curr;
            curr = curr -> next;
        }

        prev -> next = curr -> next;

        //1 Node Linked List
        if(curr == prev) {
            tail = NULL;
        }

        //>=2 Node linked list
        else if(tail == curr ) {
            tail = prev;
        }

        curr -> next = NULL;
        delete curr;

    }

}
```