One of the drawbacks of an array is that it is a static data structure, that is, the maximum capacity of an array should be known before the compilation process. Therefore, we must explicitly define its size before compilation there by reduces effective space utilization.

 Another drawback of arrays is that the elements in an array are stored a fixed distance apart, and the insertion and deletion of elements in between require a lot of data movement.

The linked list is the solution to overcome all these problems. A linked list using dynamic memory management follows

this principle—allocate and use memory when you need it and release it (free or deallocate)

A linked list is a very effective and efficient dynamic data structure for linear lists.

 Items may be added or deleted from it at any position more easily as compared to arrays.A programmer does not need to worry about how many data items a program will have to store. This enables the programmer to make effective use of the memory,

**INTRODUCTION**

Arrays and linked lists are examples of linear lists. Linear lists are those in which each member has a unique successor. Arrays contain consecutive memory locations that are a fixed

**Representing Chains in c++**

 **Defining a node in c++**

In general, a *node* is a collectionof data and link(s). *Data* is a collection of one or more items. Each item in a node is called a *field*. A field contains either a data item or a link. Every node must contain atleast one link field.

 In a linked list, before adding any element to the list, a memory space for that node must be

 allocated.

Each node of the linked list has at least the following two elements:

1. The data member(s) being stored in the list.

2. A pointer or link to the next element in the list.

The data field holds data element(s) and the linkfield(s) stores the address of its successor (and/or predecessor, if any). As the link field is a pointer to its successor, it should be a pointer variable, which should hold the addressof its successor. The successor node is of the same type as that of the node itself. Hence, every node has one member, which points to a node of the same type as itself. As every node is a group of two (or more) data elements which are of different data types, they are logically grouped using the data type, *object*. The link field of a node is a pointer that references to a node of the same type as itself. Hence, we need a *self-referential object*.

class LinkedList;

class node

{

 public:

 int data;

 node \*next;

 friend LinkedList;

};

data next

 **node**

**Designing a chain class in c++ :Designing a Linked List**

 Linked Lists

 linked lists do not necessarily contain consecutive memory locations.These data items can be stored

 anywhere in the memory in a scattered manner.

 To maintain the specific sequence of these data items, we need to maintain link(s) with a successor (and/or a predecessor). It is called as a *linked list* as each node is linked with its successor (and/or predecessor).

 A *linked list* is an ordered collection of data in which each element (node) contains a minimum of two values, *data* and *link*(*s*) to its successor (and/or predecessor). A list with one link field using which every element is associated to its successor is known as a *singlylinked list* (SLL). A link is made from each item to the next item in the list

data 1 next

data 2 next

data3 NULL

 HEAD Node1 Node2 Node3/TAIL

memory for a node is dynamically allocated. Therefore, the number of elements that may be added to a list is limited only by the amount of memory available.

Linked List Terminology

The following terms are commonly used in discussions about linked lists:

***Head pointer*** A linked list must always have at least one pointer pointing to the first node (head) of the list.This pointer is often called *head pointer*, Which contains the address of the first node of the list if list exists or NULL if list doesn’t exist

***Tail pointer*** Similar to the head pointer we may have a pointer pointing to the last node of a linked list called the *tail pointer*.

The last node in the list contains a null pointer (or a suitable value like -1) to indicate that it is the end or tail of the list

***Data node*** The list contains data nodes that store the data members and link(s) to its predecessor (and/or successor).

 Primitive Operations

The following are basic operations associated with the linked list as a data structure:

1. Creating an empty list

class LinkedList

{

 node \*Head;

 public:

 LinkedList();

 void create();

 void InBegin(int);

 void display();

 void search(int);

 void InAfter(int,int);

 void InBefore(int,int);

 void InEnd(int);

 void DelBegin();

 void DelAfter(int);

 void DelBefore(int);

 void DelEnd();

};

2. Inserting a node

3. Deleting a node

4. Traversing the list

5. Searching a node

6. Updating a node

7. Printing the node or list

8. Counting the length of the list

9. Reversing the list

10. Sorting the list using pointer manipulation

11. Concatenating two lists

12. Merging two sorted lists into a third sorted list

**PointerManipulation in C++**

**Dynamic Memory Management**

Dynamic Memory Management in C++ with new and deleteOperators

A special area of main memory, called the *heap*, is reserved for the dynamic variables.Any new dynamic variable created by a program consumes some memory in the heap. The heap is a pool of memory from which the new operator allocates memory. The memory allocated from the system heap using the new operator is de-allocated (released) using the delete operator*.*

C++ enables programmers to control the allocation and de-allocation of memory in a program. The users can dynamically allocate and de-allocate memory for any built-in or user-defined data structure.

***The new Operator***

The new operator creates a new dynamic object of a specified type and returns a pointer that points to this new object (if it fails to create the desired object, it returns 0).

Consider the following declaration and statement:

The memory is allocated dynamically and stores the address of a node to be created in newnode

node \*newnode;

newnode=new node();

If new cannot create the requested dynamic variable, then it returns a special pointer named Null.

*The Null Pointer*

Null is a special constant pointer value that is used to give a value to a pointer variable that would not otherwise have a value. It can be assigned to a pointer variable of any type.

*The delete Operator*

The object created exists till it is explicitly deleted, or till the function/program runs. To destroy a dynamically allocated variable/object and free the space occupied by the object,the delete operator is used.

delete ptr;

If we want to free a dynamically allocated array, the following is the syntax:

delete newnode;

 **Chain Manipulation Operations**

**Constructor:** Which Initialises the LinkedList Object

The LinkedList class has only one data member, the Head pointer, which points to the first node of the list, which is used to access the list.

LinkedList::LinkedList()

{

 Head=NULL;

}



 Head

The member functions including the constructor and the destructor are used to process the list. Note that the Head is private and all other member functions are public.

**Create :** Which Creates Linked List By adding nodes to the List One after Other

void LinkedList::create()

{ node \*newnode;

 node \*temp;

 while(1)

 { newnode=new node();

 cout<<"Enter data:";

 cin>>newnode->data;

 if(newnode->data==-1)break;

 if(Head==NULL)

 { newnode->next=NULL;

 Head=newnode;

 }

 Else

 { temp=Head;

 while(temp->next!=NULL)

 {

 temp=temp->next;

 }

 newnode->next=NULL;

 temp->next=newnode;

 }

 }

}

(Address of 1st node) newnode

 head 1000

1000

10 NULL

(Address of 1st node) temp newnode

 head 1000 2000

1000

10 2000

20 NULL

void LinkedList::InBegin(int val)

{

 node \*newnode;

 newnode=new node();

 newnode->data=val;

 newnode->next=Head;

 Head=newnode;

}



void LinkedList::InEnd(int val)

{

 node \*newnode;

 node \*temp;

 newnode=new node();

 temp=Head;

 newnode->data=val;

 while(temp->next!=NULL)

 {

 if(temp->next->next==NULL)

 {

 temp->next->next=newnode;

 newnode->next=NULL;

 cout<<val<<" is inserted at the end\n";

 break;

 }

 temp=temp->next;

 }

}



void LinkedList::InAfter(int bef,int val)

{

 node \*newnode;

 node \*temp;

 newnode=new node();

 newnode->data=val;

 temp=Head;

 while(temp->next!=NULL)

 {

 if(temp->data==bef)

 {

 newnode->next=temp->next;

 temp->next=newnode;

 cout<<val<<" is inserted\n";

 break;

 }

 temp=temp->next;

 }

 if(temp->next==NULL)

 cout<<bef<<" is not found\n";

}



void LinkedList::InBefore(int aft,int val)

{

 node \*newnode;

 node \*temp;

 newnode=new node();

 temp=Head;

 while(temp->next!=NULL)

 {

 if(temp->next->data==aft)

 {

 newnode->data=val;

 newnode->next=temp->next;

 temp->next=newnode;

 cout<<val<<" is inserted\n";

 break;

 }

 temp=temp->next;

 }

}





void LinkedList::search(int val)

{

 node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 {

 if(temp->data==val)

 {

 cout<<val<<" is present in the link\n";

 break;

 }

 temp=temp->next;

 }

 if(temp->next==NULL)

 cout<<val<<" is not present in the link\n";

}

void LinkedList::DelBegin()

{ node \*temp;

 temp=Head;

 Head=temp->next;

}



void LinkedList::DelAfter(int val)

{ node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 { if(temp->data==val)

 {

 temp->next=temp->next->next;

 cout<<"Node after "<<val<<" is deleted\n";

 break;

 }

 temp=temp->next;

 }

}

void LinkedList::DelBefore(int val)

{ node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 { if(temp->next->next->data==val)

 {

 temp->next=temp->next->next;

 cout<<"Node before "<<val<<" is deleted\n";

 break;

 }

 temp=temp->next;

 }

}

void LinkedList::DelEnd()

{ node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 { if(temp->next->next==NULL)

 {

 temp->next=NULL;

 cout<<"Last node is deleted\n";

 break;

 }

 temp=temp->next;

 }

}

void LinkedList::display()

{

 node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 {

 cout<<"Data="<<temp->data<<"\tAddress="<<temp->next<<endl;

 temp=temp->next;

 }

 cout<<"Data="<<temp->data<<"\tAddress="<<temp->next<<endl;

}

int main()

{

 LinkedList L;

 int ch,a,b;

 do

 {

 cout<<"1.Create 2.Display 3.Search 4.InBegin 5.InAfter 6.InBefore 7.InEnd

 8.DelBegin 9.DelAfter 10.DelBefore 11.DelEnd 12.Exit\n";

 cout<<"Enter your choice:";

 cin>>ch;

 switch(ch)

 {

 case 1:L.create();

 break;

 case 2:L.display();break;

 case 3:cout<<"Enter the element to search:";

 cin>>a;

 L.search(a);

 break;

 case 4:cout<<"Enter a value:";

 cin>>a;

 L.InBegin(a);

 cout<<a<<" is inserted\n";

 break;

 case 5:cout<<"Enter the value to insert:\n";

 cin>>a;

 cout<<"Enter value after which it should be inserted:\n";

 cin>>b;

 L.InAfter(b,a);

 break;

 case 6:cout<<"Enter a value to insert:\n";

 cin>>a;

 cout<<"Enter value before which it should be inserted:\n";

 cin>>b;

 L.InBefore(b,a);

 break;

 case 7:cout<<"Enter a value to insert:\n";

 cin>>a;

 L.InEnd(a);

 break;

 case 8:L.DelBegin();

 cout<<"Begin value is deleted\n ";

 break;

 case 9:cout<<"Enter a value after which the node is to be deleted:\n";

 cin>>a;

 L.DelAfter(a);

 break;

 case 10:cout<<"Enter a value before which the node is to be deleted:\n";

 cin>>a;

 L.DelBefore(a);

 break;

 case 11:L.DelEnd();

 break;

 case 12:exit(-1);

 }

 }while(1);

}

the nodes are made inaccessible to outside objects by declaring Head private so that the information hiding principle is not really compromised.

**Template Class Chain:**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

template <class t> class linkl;

template<class t>

class node

{

 t data;

 node<t> \*next;

 friend class linkl<t>;

};

template<class t>

class linkl

{

 node<t> \*head;

 node<t> \*temp;

 public:

 linkl();

 void create();

 void inbeg(t);

 void inaf(t);

 void display();

 int search(t);

 void inbef(t);

 void delbeg();

 void delend();

 void delaf(t);

 void delbef(t);

};

template<class t>

linkl<t>::linkl()

{

 head=NULL;

}

template<class t>

void linkl<t>::create()

{

 node<t> \*newnode;

 node<t> \*temp;

 while(1)

 {

 newnode=new node<t>();

 cout<<"enter data:"<<"\t";

 cin>>newnode->data;

 if(newnode->data==-1)

 break;

 else if(head==NULL)

 {

 newnode->next=NULL;

 head=newnode;

 }

 else

 {

 temp=head;

 while(temp->next!=NULL)

 {

 temp=temp->next;

 }

 newnode->next=NULL;

 temp->next=newnode;

 }

 }

}

template<class t>

void linkl<t>::inbeg(t x)

 {

 node<t> \*newnode;

 newnode=new node<t>();

 newnode->data=x;

 newnode->next=head;

 head=newnode;

 }

template<class t>

 void linkl<t>::inaf(t x)

{

 if(!search(x))

 {

 cout<<x<<" is nt present in the list"<<endl;

 }

 else

 {

 temp=head;

 while(temp->data!=x)

 {

 temp=temp->next;

 }

 node<t> \*newnode;

 newnode=new node<t>();

 cout<<"enter new data"<<endl;

 cin>>newnode->data;

 newnode->next=temp->next;

 temp->next=newnode;

 }

}

template<class t>

int linkl<t>::search(t x)

{

 temp=head;

 while(temp->next!=NULL)

 {

 if(temp->data==x)

 return 1;

 else

 temp=temp->next;

 }

 if(temp->data==x)

 return 1;

 else

 return 0;

}

template<class t>

void linkl<t>::display()

{

temp=head;

cout<<"the elements in the list are"<<endl;

while(temp->next!=NULL)

{

 cout<<temp->data<<" ";

 temp=temp->next;

}

cout<<temp->data<<endl;

}

template<class t>

void linkl<t>::delbeg()

{

 temp=head;

 head=temp->next;

}

template<class t>

void linkl<t>::delend()

{

 temp=head;

 while(temp->next->next!=NULL)

 {

 temp=temp->next;

 }

 temp->next=NULL;

}

template<class t>

void linkl<t>::delaf(t x)

{

 if(!search(x))

 cout<<"element nt found in the list"<<endl;

 else

 {

 temp=head;

 while(temp->data!=x)

 {

 temp=temp->next;

 }

 temp->next=temp->next->next;

 }

}

template<class t>

void linkl<t>::delbef(t x)

{

 if(!search(x))

 {

 cout<<"element not found in list"<<endl;

 }

 else

 {

 node<t> \*delnode;

 temp=head;

 while(temp->next!=NULL)

 {

 if(temp->next->next->data==x)

 {

 temp->next=temp->next->next;

 break;

 }

 else

 {

 temp=temp->next;

 }

 }

}

 }

template<class t>

void linkl<t>::inbef(t x)

{

 if(!search(x))

 {

 cout<<"element not found in the list"<<endl;

 }

 else

 {

 temp=head;

 while(temp->next->data!=x)

 {

 temp=temp->next;

 }

 node<t> \*newnode;

 newnode=new node<t>();

 t y;

 cout<<"enter new data"<<" ";

 cin>>y;

 newnode->data=y;

 newnode->next=temp->next;

 temp->next=newnode;

 }

 }

void main()

{

 clrscr();

 int ch;

 int q,r,t,m,o,p,k;

 linkl<int> l;

 do

 {

 cout<<"choose an operation"<<endl;

 cout<<"1.to create 2.insert at beginnng 3.insert after 4.insert before"<<endl;

 cout<<"5.delete at beginning 6.delete at end 7.delete after 8.delete before"<<endl;

 cout<<"9.search 10.to display 11.to exit"<<endl;

 cout<<"enter choice"<<" ";

 cin>>ch;

 switch(ch)

 {

 case 1:

 l.create();

 break;

 case 2:

 cout<<"enter an element to insert at beginning"<<endl;

 cin>>q;

 l.inbeg(q);

 break;

 case 3:

 cout<<"enter a value after which the element should be entered"<<endl;

 cin>>r;

 l.inaf(r);

 break;

 case 4:

 cout<<"enter a value before which the element should be entered"<<endl;

 cin>>t;

 l.inbef(t);

 break;

 case 5:

 l.delbeg() ;

 break;

 case 6:

 l.delend();

 break;

 case 7:

 cout<<"enter a value after which the element should be entered"<<endl;

 cin>>m;

 l.delaf(m);

 break;

 case 8:

 cout<<"enter a value before which the element should be deleted"<<endl;

 cin>>o;

 l.delbef(o);

 break;

 case 9:

 cout<<"enter an element to search"<<endl;

 cin>>k;

 p=l.search(k);

 if(p==1)

 cout<<k<<" is present in the list"<<endl;

 else

 cout<<k<<" is not in list"<<endl;

 case 10:

 l.display();

 break;

 case 11:

 exit(0);

 }

 }while(1);

}

**Circular List**

Linear and Circular Linked Lists

The other classification of linked lists based on their method of traversal is as follows:

1. Linear linked list

2. Circular linked list

*Linear Linked List*

The linked lists that we have seen so for are known as *linear linked lists*. All elements of

such a linked list can be accessed by traversing a list from the first node of the list.

Singly Linked List has a major drawback. From a specified node, it is not possible to reach any of the preceding nodes in the list. To overcome the drawback, a small change is made to the SLL so that the next field of the last node is pointing to the first node rather than NULL. Such a linked list is called a circular linked list Because it is a circular linked list, it is possible to reach any node in the list from a particular node.There is no natural first node or last node because by virtue of the list is circular.



**// Circular List**

#include<iostream>

using namespace std;

class Circular;

class node

{

 int data;

 node \*next;

 friend Circular;

};

class Circular

{

 node \*Head;

 public:

 Circular();

 void create();

 void display();

};

Circular::Circular()

{

 Head=NULL;

}

void Circular::create()

{

 node \*temp;

 node \*newnode;

 while(1)

 {

 newnode=new node();

 cout<<"Enter data:";

 cin>>newnode->data;

 if(newnode->data==-1)break;

 if(Head==NULL)

 {

 Head=newnode;

 newnode->next=Head;

 }

 else

 {

 temp=Head;

 while(temp->next!=Head)

 {

 temp=temp->next;

 }

 newnode->next=Head;

 temp->next=newnode;

 }

 }

}

void Circular::display()

{

 node \*temp;

 temp=Head;

 while(temp->next!=Head)

 {

 cout<<temp->data<<endl;

 temp=temp->next;

 }

 cout<<temp->data<<endl;

}

int main()

{

 Circular C;

 cout<<"Enter elements:";

 C.create();

 cout<<"The elements are:";

 C.display();

 return(0);

}

**Linkedstacks**

we have implemented stacks using arrays.However, an array implementation has certain limitations. One of the limitations is that such a stack cannot grow or shrink dynamically. This drawback can be overcome by using linked implementation.

A stack implemented using a linked list is also called *linked stack*.Each element of the stack will be represented as a node of the list. The addition and deletion of a node will be only at one end. The last element given is at the top of the stack, and it will be pointed to by a pointer called top. The first element is at the bottom of the stack, and its link field is set to Null. An empty stack will have Top = Null.

Operations on Linked Stack

The memory for each node is dynamically allocated on the heap. So when an item is pushed, a node for it is created, and when an item is popped, its node is freed (using delete).

The only difference is that the capacity of a linked stack is generally greater than that of a contiguous stack since a linked stack will not become full until the dynamic memory is exhausted

class Stack\_Node

{

public:

int data;

Stack\_Node \*link;

};

class Stack

{

private:

Stack\_Node \*Top;

int Size;

int IsEmpty();

public:

Stack()

{

Top = Null;

Size = 0;

}

int GetTop();

int Pop();

void Push(int Element);

int CurrSize();

};

int Stack :: IsEmpty()

{

if(Top == Null)

return 1;

else

return 0;

}

int Stack :: GetTop()

{

if(!IsEmpty())

return(Top->data);

}

void Stack :: Push(int value)

{

Stack\_Node\* NewNode;

NewNode = new Stack\_Node;

NewNode->data = value;

//NewNode->link = Null;

NewNode->link = Top;

Top = NewNode;

}

int Stack :: Pop()

{Stack\_Node\* tmp = Top;

int data = Top->data;

if(!IsEmpty())

{

Top = Top->link;

delete tmp;

return(data);

}

}

The Top is initialized to Null to indicate empty stack.

The Push() function dynamically creates a new node. After creating a new node, the

pointer variable Top should point to the newly added node in the stack.

void main()

{

Stack S;

S.Push(5);

S.Push(6);

cout << S.GetTop()<<endl;

cout << S.Pop()<<endl;

S.Push(7);

cout << S.Pop()<<endl;

cout << S.Pop()<<endl;

}

Output

6

6

7

5

**Linked Queues**

we have a circular queue of fixed size. However,there are many drawbacks of implementing queues using arrays. The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size. The declaration of arbitrarily maximum size leads to poor utilization of memory. In addition, the major

drawback is the updating of front and rear.

class Student

{

public:

int Roll\_No;

char Name[30];

int Year;

char Branch[8];

Student \*link;

};

class Queue

{

Student \*front, \*rear;

public:

Queue()

{

front = rear = Null;

}

};

Let us consider the following node structure for studying the linked queue and

operations:

class QNode

{

public:

int data;

QNode \*link;

};

class Queue

{

QNode \*Front, \*Rear;

int IsEmpty();

public:

Queue()

{

Front = Rear = Null;

}

void Add( int Element);

int Delete();

int FrontElement();

~Queue();

};

int Queue :: IsEmpty()

{

if(Front == Null)

return 1;

else

return 0;

}

int Queue :: GetFront()

{

if(!IsEmpty())

return(Front->data);

}

void Queue :: Add(int x)

{

QNode \*NewNode;

NewNode = new QNode;

NewNode->data = x;

NewNode->link = Null;

if(Rear == Null)

{

Front = NewNode;

Rear = NewNode;

}

else

{

Rear->link = NewNode;

Rear = NewNode;

}

}

int Queue :: Delete()

{

int temp;

QNode \*current = Null;

if(!IsEmpty())

{

temp = Front->data;

current = Front;

Front = Front->link;

delete current;

if(Front == Null)

Rear = Null;

return(temp);

}

}

int Queue :: FrontElement()

{

if(!IsEmpty())

return(Front->data);

}

void main()

{

Queue Q;

Q.Add(11);

Q.Add(12);

Q.Add(13);

cout << Q.Delete() << endl;

Q.Add(14);

cout << Q.Delete() << endl;

cout << Q.Delete() << endl;

cout << Q.Delete() << endl;

Q.Add(15);

Q.Add(16);

cout << Q.Delete() << endl;

cout << Q.Delete() << endl;

}

Output

11

12 // due to FrontElement

12

13

14

15

16

**Polynomial Representation**

polynomial we want to represent using a linked list be *A*(*x*). It is expanded as,

*A*(*x*) = *k*1*xm* + … + *kn*-1 *x*2 + *knx*1

where *ki* is a non-zero coefficient with exponent *m* such that *m* *m* 1... 2 1 ≥0.

A node of the linked list will represent each term. A node will have 3 fields, which represent the coefficient and exponent of a term and a pointer to the next term

****

For instance, the polynomial, say *A* 6*x*7 3*x*5 4*x*3 12 would be stored as

****

The polynomial *B* 8*x*5 9*x*4 2*x*2 10 would be stored as

class Polynomial;

class node

{

 int exp;

 int coef;

 node \*next;

};

Polynomial Addition

Let two polynomials *A* and *B* be

*A* 4*x*9 3*x*6 5*x*2 1

*B* 3*x*6 *x*2 2*x*

The polynomial *A* and *B* are to be added to yield the polynomial *C*. The assumption here is the two polynomials are stored in linked list with descending order of exponents.

The two polynomials *A* and *B* are stored in two linked lists with pointers temp1 and temp2 pointing to the first node of each polynomial, respectively.

**// Polynomial using single linked list**

#include<iostream>

#include<stdlib.h>

using namespace std;

class Polynomial;

class node

{

 int exp;

 int coef;

 node \*next;

 friend Polynomial;

};

class Polynomial

{

 node \*Head;

 int terms;

 public:

 Polynomial(int);

 void create();

 Polynomial add(Polynomial);

 void display();

};

Polynomial::Polynomial(int n)

{

 Head=NULL;

 terms=n;

}

void Polynomial::create()

{

 node \*newnode;

 node \*temp;

 int i=1;

 while(i<=terms)

 {

 newnode=new node();

 cout<<"Enter exponent:";

 cin>>newnode->exp;

 cout<<"Enter coefficent:";

 cin>>newnode->coef;

 if(Head==NULL)

 {

 newnode->next=NULL;

 Head=newnode;

 }

 else

 {

 temp=Head;

 while(temp->next!=NULL)

 {

 temp=temp->next;

 }

 newnode->next=NULL;

 temp->next=newnode;

 }

 i++;

 }

}

Polynomial Polynomial::add(Polynomial B)

{

 Polynomial C(0);

 node \*newnode;

 node \*temp1;

 node \*temp2;

 node \*temp3;

 temp1=Head;

 temp2=B.Head;

 while(temp1!=NULL && temp2!=NULL)

 {

 newnode=new node();

 if(temp1->exp==temp2->exp)

 {

 newnode->exp=temp1->exp;

 newnode->coef=temp1->coef+temp2->coef;

 C.terms+=1;

 temp1=temp1->next;

 temp2=temp2->next;

 newnode->next=NULL;

 }

 else if(temp1->exp>temp2->exp)

 {

 newnode->exp=temp1->exp;

 newnode->coef=temp1->coef;

 C.terms+=1;

 temp1=temp1->next;

 newnode->next=NULL;

 }

 else

 {

 newnode->exp=temp2->exp;

 newnode->coef=temp2->coef;

 C.terms+=1;

 temp2=temp2->next;

 newnode->next=NULL;

 }if(C.Head==NULL)

 {

 C.Head=newnode;

 }

 else

 {

 temp3=C.Head;

 while(temp3->next!=NULL)

 {

 temp3=temp3->next;

 }

 temp3->next=newnode;

 }

 }

 while(temp1!=NULL)

 {

 newnode=new node();

 newnode->exp=temp1->exp;

 newnode->coef=temp1->coef;

 C.terms+=1;

 temp1=temp1->next;

 newnode->next=NULL;

 if(C.Head==NULL)

 {

 C.Head=newnode;

 }

 else

 {

 temp3=C.Head;

 while(temp3->next!=NULL)

 {

 temp3=temp3->next;

 }

 temp3->next=newnode;

 }

 }

 while(temp2!=NULL)

 {

 newnode=new node();

 newnode->exp=temp2->exp;

 newnode->coef=temp2->coef;

 C.terms+=1;

 temp2=temp2->next;

 newnode->next=NULL;

 if(C.Head==NULL)

 {

 C.Head=newnode;

 }

 else

 {

 temp3=C.Head;

 while(temp3->next!=NULL)

 {

 temp3=temp3->next;

 }

 temp3->next=newnode;

 }

 }

 return(C);

}

void Polynomial::display()

{

 node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 {

 cout<<temp->coef<<"X^"<<temp->exp<<"+";

 temp=temp->next;

 }

 cout<<temp->coef<<"X^"<<temp->exp<<endl;

}

int main()

{

 Polynomial A(4);

 Polynomial B(5);

 Polynomial C(0);

 A.create();

 cout<<"\nPolynomial A\n";

 A.display();

 B.create();

 cout<<"\nPolynomial B\n";

 B.display();

 cout<<"\nResultant Polynomial\n";

 C=A.add(B);

 C.display();

}

**SparseMatrixRepresentation**

**Representation of sparse matrix using linked list**

We have studied the sparse matrix representation using arrays, which is a sequential allocation scheme.

Representing a sparse matrix sequentially allows faster execution of matrix operations, and it is more

storage efficient than linked allocation schemes. However, it has many shortcomings. The insertion

and deletion of elements need the movement of many other elements. In applications with frequent insertions and deletions, a linked representation can be adopted.

*Header Nodes*

1. Row field contains the number of rows.

2. Column field contains the total number of non-zero entries.

3. Row\_link field contains pointer to the header node of the first row.

4. Column\_link field contains pointer to the header node of the first column.

**Types of Linked List**

We studied that in a linked list, every node must have at least one linked fi eld. Thus, each

node provides information about its predecessor and/or successor in the list. It may also

have the knowledge about where the previous node lies in the memory. Thus, linked lists

can be classified broadly as follows:

1. Singly linked list

2. Doubly linked list

The list and operations we discussed so far had only one link pointing to its successor and

is called as singly linked list.

*Singly Linked List* A linked list in which every node has one link fi eld, to provide information about where the next node of the list is, is called as *singly linked list* (SLL). It has no knowledge about where the previous node lies in the memory. In SLL, we can traverse only in one direction. We have no way to go to the *i*th node from (*i* + 1)th node, unless the list is traversed again from the first node.

Often SLL is just referred to as a linked list.

***Doubly Linked List***

In a doubly linked list (DLL), each node has two link fields to store information about the

one to the next and also about the one ahead of the node. Hence, each node has knowledge

of its successor and also its predecessor. In DLL, from every node, the list can be

traversed in both the directions

 Doubly Linked list

In SLL, each node provides information about where the next node is. It has no knowledge

about where the previous node is. For example, if we are at the *i*th node in the list

currently, then to access the (*i* - 1)th node or (*i* - 2)th node, we have to traverse the list

right from the first node. In addition, it is not possible to delete the *i*th node given only a

pointer to the *i*th node. It is also not possible to insert a node before the *i*th node given only

a pointer to the *i*th node (there are other ways that are without link manipulations such as

using data exchange).

For handling such difficulties, we can use DLLs where each node contains two links,

one to its predecessor and other to its successor

**//Double Linked List**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

classlinkedlist;

class node

{

int data;

node \*prev;

node \*next;

friendlinkedlist;

};

classlinkedlist

{

node \*head;

intlcount;

public:

linkedlist();

void create();

voidINSatPOS(int,int);

void display();

voidDELatPOS(int);

void search(int);

};

linkedlist::linkedlist()

{

head=NULL;

lcount=0;

}

voidlinkedlist::create()

{

node \*newnode;

node \*tail;

while(1)

 {

cout<<"\n enter values:";

newnode=new node();

cin>>newnode->data;

if(newnode->data==-1)

 {

break;

 }

else if(head==NULL) (Address of

 1st node) (newnode)

 {

1000

 NULL 10 NULL

newnode->next=NULL;

newnode->prev=NULL;

head=newnode;(head) 1000

tail=newnode; (tail)

 }

else

 { (newnode)

newnode->next=NULL;

NULL 10 2000

1000 20 NULL

1000

newnode->prev=tail;

tail->next=newnode;

tail=newnode; head 1000 2000

} (tail)

lcount++;

 }

}

voidlinkedlist::INSatPOS(intpos,intval)

{

if(pos<=lcount+1)

 {

node \*newnode;

node \*temp;

newnode=new node();

newnode->data=val;

if(pos==1)

 {

temp=head;

newnode->prev=NULL;

newnode->next=head;

temp->prev=newnode;

head=newnode;

 }

else

 {

temp=head;

node \*prev;

int count=1;

while(pos!=count)

 {

 prev=temp;

 temp=temp->next;

 count++;

 }

newnode->prev=prev;

newnode->next=temp;

prev->next=newnode;

if(temp!=NULL)

 {

 temp->prev=newnode;

 }

 }

lcount++;

 }

else

 {

cout<<"not valid position";

 }

}

insertion at position 1(beginning of the list)

1000 20 NULL

3000 10 2000

NULL 30 1000

3000

Head 3000 1000 2000

 (temp)

Insertion at 3 position(middle of the list)

c

 4000 20 NULL

 1000 40 2000

 3000 10 4000

 NULL 30 1000

3000

Head 3000 1000 4000 2000

 (prev) (newnode) (temp)

 Insertion at 5th position

 (end of the list)

3000

 3000 10 4000

NULL 30 1000

 2000 50 NULL

 4000 20 5000

 1000 40 2000

 Head 3000 1000 4000 2000 5000

 (prev) newnode

voidlinkedlist::DELatPOS(intpos)

{

if(pos<=lcount)

 {

node \*temp;

node \*prev;

if(pos==1)

 {

temp=head;

temp->next->prev=NULL;

head=temp->next;

 }

else

 {

temp=head;

int count=1;

while(pos!=count)

 {

prev=temp;

temp=temp->next;

count++;

 }

prev->next=temp->next;

if(temp->next!=NULL)

 {

temp->next->prev=prev;

 }

 }

lcount--;

 }

else

 {

cout<<"invalid position";

 }

}

 Delete node at position 1(beginning of the list)

 2000 50 NULL

 4000 20 5000

 1000 40 2000

 NULL 10 4000

1000

Head 1000 4000 2000 5000

NULL 30 1000

(Deleted node)

 Delete node at position 2 (middle of the list)

 1000 20 5000

2000 50 NULL

 NULL 10 2000

1000

Head 1000 2000 5000

 (prev)

1000 40 2000

(Deleted node)

 Delete node at last position

2000 50 NULL

1000

 NULL 10 2000

1000 20 NULL

Head 1000 2000 5000

 (prev) (deleted node)

voidlinkedlist::search(intval)

{

node \*temp;

temp=head;

while(temp->next!=NULL)

 {

if(temp->data!=val)

 {

temp=temp->next;

 }

else

 {

cout<<val<<"is present in the list";

break;

 }

 }

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

 {

cout<<"\n "<<val<<" is not present in the list";

 }

}

voidlinkedlist::display()

{

node \*temp;

temp=head;

while(temp->next!=NULL)

 {

cout<<temp->data<<"\t";

temp=temp->next;

 }

cout<<temp->data;

}

main()

{

linkedlist l;

clrscr();

l.create();

intch,pos,val;

do{

cout<<"\n 1-insert at position 2-search 3-display ";

cout<<"4-delete at position 5-exit";

cout<<"\n choose operation:";

cin>>ch;

switch(ch)

 {

case 1:cout<<"enter position and value to insert a element:";

 cin>>pos;

 cin>>val;

 l.INSatPOS(pos,val);

 break;

case 2:cout<<"\n enter value to search";

 cin>>val;

 l.search(val);

 break;

case 3:cout<<"\n elements in list are:";

 l.display();

 break;

case 4:cout<<"\n enter position to delete an element:";

 cin>>pos;

 l.DELatPOS(pos);

 break;

case 5:exit(0);

 }

}while(1);

}

#include<iostream>

#include<stdlib.h>

using namespace std;

class node

{

 public:

 int data;

 node \*next;

 node \*prev;

};

class DoubleLink

{

 node \*Head;

 int Lcount;

 public:

 DoubleLink();

 void create();

 void InAtPos(int,int);

 void DeletePos(int);

 void search(int);

 void display();

};

DoubleLink::DoubleLink()

{

 Head=NULL;

 Lcount=0;

}

void DoubleLink::create()

{

 node \*temp;

 node \*newnode;

 while(1)

 {

 newnode=new node();

 cout<<"Enter data:";

 cin>>newnode->data;

 if(newnode->data==-1)break;

 if(Head==NULL)

 {

 newnode->next=NULL;

 newnode->prev=NULL;

 Head=newnode;

 Lcount+=1;

 }

 else

 {

 temp=Head;

 while(temp->next!=NULL)

 {

 temp=temp->next;

 }

 newnode->next=NULL;

 newnode->prev=temp;

 temp->next=newnode;

 Lcount+=1;

 }

 }

}

void DoubleLink::display()

{

 node \*temp;

 temp=Head;

 while(temp->next!=NULL)

 {

 cout<<temp->data<<"\t";

 temp=temp->next;

 }

 cout<<temp->data<<endl;

}

void DoubleLink::InAtPos(int pos,int val)

{

 node \*newnode;

 node \*temp;

 if(pos<=1+Lcount)

 {

 newnode=new node();

 newnode->data=val;

 if(pos==1)

 {

 temp=Head;

 temp->prev=newnode;

 newnode->next=temp;

 newnode->prev=NULL;

 Head=newnode;

 }

 else

 {

 temp=Head;

 int count=1;

 while(pos!=count)

 {

 temp->prev=temp;

 temp=temp->next;

 count++;

 }

 newnode->prev=temp->prev;

 newnode->next=temp;

 temp->prev->next=newnode;

 if(temp!=NULL)

 {

 temp->prev=newnode;

 }

 }

 }

 else

 {

 cout<<"Not a valid position\n";

 }

}

void DoubleLink::search(int val)

{

 node \*temp;

 temp=Head;

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

 {

 temp=temp->next;

 }

 if(temp->data==val)

 cout<<val<<" is found\n";

 else if((temp->next==NULL)&&(temp->data!=val))

 cout<<val<<" is not found\n";

}

void DoubleLink::DeletePos(int pos)

{

 node \*temp;

 temp=Head;

 int count=1;

 while(pos!=count)

 {

 temp->prev=temp;

 temp=temp->next;

 count++;

 }

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

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

}

int main()

{

 DoubleLink D;

 int ch,a,b;

 cout<<"Enter elements to create a list:";

 D.create();

 do

 {

 cout<<"1.InAtPos 2.Display 3.Search 4.DeletePos 5.Exit\n";

 cout<<"Enter choice:";

 cin>>ch;

 switch(ch)

 {

 case 1:cout<<"Enter value to insert:";

 cin>>a;

 cout<<"Enter position:";

 cin>>b;

 D.InAtPos(b,a);

 break;

 case 2:D.display();

 break;

 case 3:cout<<"Enter value to search:";

 cin>>a;

 D.search(a);break;

 case 4:cout<<"Enter position:";

 cin>>b;

 D.DeletePos(b);break;

 case 5:exit(-1);

 }

 }while(1);

}