string.cpp - If you don't have a string type

```
#include <string.h>
#include "mystring.h"

string::string( const char * cstring )
{
   if( cstring == NULL )
      cstring = "";
   strLength = strlen( cstring );
   bufferLength = strLength + 1;
   buffer = new char[ bufferLength ];
   strcpy( buffer, cstring );
}

string::string( const string & str )
{
   strLength = str.length( );
   bufferLength = strLength + 1;
   buffer = new char[ bufferLength ];
   strcpy( buffer,str.buffer );
}

const string & string::operator=( const string & rhs )
{
   if( this != &rhs )
   {
      if( bufferLength < rhs.length( ) + 1 )
      {
         delete [ ] buffer;
         bufferLength = rhs.length( ) + 1;
         buffer = new char[ bufferLength ];
      }
      strLength = rhs.length( );
      strcpy( buffer, rhs.buffer );
   }
   return *this;
}

const string & string::operator+=( const string & rhs )
{
   if( this == &rhs )
   {
      string copy( rhs );
      return *this += copy;
   }

   int newLength = length( ) + rhs.length( );
   if( newLength >= bufferLength )
   {
      bufferLength = 2 * ( newLength + 1 );

      char *oldBuffer = buffer;
      buffer = new char[ bufferLength ];
      strcpy( buffer, oldBuffer );
      delete [ ] oldBuffer;
   }
```

```
    strcpy( buffer + length( ), rhs.buffer );
    strLength = newLength;
    return *this;
}

char & string::operator[ ]( int k )
{
    if( k < 0 || k >= strLength )
        throw StringIndexOutOfBounds( );
    return buffer[ k ];
}

char string::operator[ ]( int k ) const
{
    if( k < 0 || k >= strLength )
        throw StringIndexOutOfBounds( );
    return buffer[ k ];
}

ostream & operator<<( ostream & out, const string & str )
{
    return out << str.c_str();
}

istream & operator>>( istream & in, string & str )
{
    char buf[ string::MAX_LENGTH + 1 ];
    in >> buf;
    if( !in.fail( ) )
        str = buf;
    return in;
}

istream & getline( istream & in, string & str )
{
    char buf[ string::MAX_LENGTH + 1 ];
    in.getline( buf, string::MAX_LENGTH );
    if( !in.fail( ) )
        str = buf;
    return in;
}

bool operator==( const string & lhs, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) == 0;
}

bool operator!=( const string & lhs, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) != 0;
}

bool operator<( const string & lhs, const string & rhs )
{
    return strcmp( lhs.c_str( ), rhs.c_str( ) ) < 0;
}
```

```
bool operator<=( const string & lhs, const string & rhs )
{
   return strcmp( lhs.c_str( ), rhs.c_str( ) ) <= 0;
}

bool operator>( const string & lhs, const string & rhs )
{
   return strcmp( lhs.c_str( ), rhs.c_str( ) ) > 0;
}

bool operator>=( const string & lhs, const string & rhs )
{
   return strcmp( lhs.c_str( ), rhs.c_str( ) ) >= 0;
}
```

A Templated Stack Data Structure Example

```
#include <dos.h>          // For sleep()
#include <iostream.h>     // For I/0
#include <windows.h>     // FOR MessageBox() API
#include <conio.h>

#define MAX 10       // MAXIMUM STACK CONTENT

template <class T>    // Using Templates so that any type of data can be
                                                                    // stored in Stack without multiple defination of
class
class stack
{

 protected:

 T arr[MAX];      // Contains all the Data

 public:
  T item,r;
  int top;       //Contains location of Topmost Data pushed onto Stack
                    stack()       //Constructor
                    {
                                  for(int i=0;i<MAX;i++)
                                                {
                                                  arr[i]=NULL;          //Initialises all Stack Contents to NULL
                                                }

                                  top=-1;     //Sets the Top Location to -1 indicating an empty stack
                    }

                   void push(T a)  // Push ie. Add Value Function
                   {
                                 top++;       // increment to by 1
                                 if(top<MAX)
                                 {
                                                arr[top]=a;        //If Stack is Vacant store Value in Array
                                 }
                                 else    // Bug the User
                                 {
```

```
                                        MessageBox(0,"STACK IS FULL","STACK
WARNING!",MB_ICONSTOP);
                                        top--;
                                }
                }

                T pop()          // Delete Item. Returns the deleted item
                {
                                if(top==-1)
                                {
                                        MessageBox(0,"STACK IS EMPTY
","WARNING",MB_ICONSTOP);
                                        return NULL;
                                }
                                else
                                {
                                        T data=arr[top];    //Set Topmost Value in data
                                        arr[top]=NULL;      //Set Original Location to NULL
                                        top--;            // Decrement top by 1
                                        return data;       // Return deleted item
                                }
                }
};


void main()
{
 stack <int>a;    // Create object of class a with int Template
 int opt=1;
 while (opt!=3)
 {
 clrscr();
 cout<<" MAX STACK CAPACITY="<<((MAX-a.top)-1)<<"



";
 cout<<"1) Push Item
";
 cout<<"2) Pop Item
";
 cout<<"3) Exit

";
 cout<<"Option?";
 cin>>opt;
 switch(opt)
 {
  case 1:
  cout<<"Which Number should be pushed?";
  cin>>a.item;
  a.push(a.item);
  break;

  case 2:
  a.r=a.pop();
  cout<<"Item popped from Stack is:"<<a.r<<endl;
```

```
 sleep(2);
 break;
}
}
}
```

Binary tree implementation

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>


struct node
{
node *left;
int value;
node *right;
};
node *curr=NULL;


int addnode(node *, node *);
int inorder(node *);
int preorder(node *);
int postorder(node *);


void main()
{
 char c;
 int v;
 clrscr();
 do
 {
 cout<<"Select any one";
 cout<<"0 ->Exit";
 cout<<"1 ->Add node";
 cout<<"2 ->Inorder traversal";
 cout<<"3 ->Preorder traversal";
 cout<<"4 ->Postorder trversal : ";

 cin>>c;

 switch(c)
  {
   case '0':
              exit(1);
   case '1':
              node *temp;
              temp = new node;
              cout<<"  Enter the value of the node : ";
```

```
                cin>>temp->value;
                if(curr==NULL)
                 {
                  curr=new node;
                  curr->value=temp->value;
                  curr->left=NULL;
                  curr->right=NULL;
                  cout<<"  The root node is added";
                 }
                else
                  v=addnode(curr,temp);
                if(v==1)
                  cout<<"  The node is added to the left";
                else if(v==2)
                  cout<<"  The node is added to the right";
                else if(v==3)
                  cout<<"  The same value exists";
                break;
    case '2':
                 v=inorder(curr);
                 if(v==0)
                 cout<<" The tree is empty";
                 break;
    case '3':
                 v=preorder(curr);
                 if(v==0)
                 cout<<" The tree is empty";
                 break;
    case '4':
                 v=postorder(curr);
                 if(v==0)
                 cout<<" The tree is empty";
                 break;
    default:
                 cout<<"Invalid entry";
                 break;
    }
   }while(c!='0');
   getch();
}


int addnode(node *fcurr, node *fnew )
{
  if(fcurr->value==fnew->value)
   {
     return 3;
   }
 else
   {
    if(fcurr->value > fnew->value)
     {
      if(fcurr->left != NULL)
              addnode(fcurr->left, fnew);
      else
              {
              fcurr->left = fnew;
```

```
            (fcurr->left)->left=NULL;
            (fcurr->left)->right=NULL;
            return 1;
            }
    }
   else
    {
     if(fcurr->right != NULL)
            addnode(fcurr->right, fnew);
     else
            {
            fcurr->right = fnew;
            (fcurr->right)->left=NULL;
            (fcurr->right)->right=NULL;
            return 2;
            }
    }
  }
}


int inorder(node *fincurr)
 {
   if(fincurr == NULL)
            return 0;
   else
    {
             if(fincurr->left != NULL)
              inorder(fincurr->left);
              cout<<fincurr->value<<"         ";
             if(fincurr->right != NULL)
              inorder(fincurr->right);
    }
 }


int preorder(node *fprcurr)
 {
   if(fprcurr == NULL)
            return 0;
   else
    {
            cout<<fprcurr->value<<"         ";
            if(fprcurr->left != NULL)
              preorder(fprcurr->left);
            if(fprcurr->right != NULL)
              preorder(fprcurr->right);
    }
 }

int postorder(node *fpocurr)
 {
   if(fpocurr == NULL)
            return 0;
   else
    {
            if(fpocurr->left != NULL)
```

```
            postorder(fpocurr->left);
           if(fpocurr->right != NULL)
            postorder(fpocurr->right);
            cout<<fpocurr->value<<"        ";
    }
 }
```

AVL tree with insertion, deletion and balancing height

```
# include <iostream.h>
# include <stdlib.h>
# include <conio.h>

struct node
{
  int element;
  node *left;
  node *right;
  int height;
};

typedef struct node *nodeptr;

class bstree
{

  public:
          void insert(int,nodeptr &);
          void del(int, nodeptr &);
          int deletemin(nodeptr &);
          void find(int,nodeptr &);
          nodeptr findmin(nodeptr);
          nodeptr findmax(nodeptr);
          void copy(nodeptr &,nodeptr &);
          void makeempty(nodeptr &);
          nodeptr nodecopy(nodeptr &);
          void preorder(nodeptr);
          void inorder(nodeptr);
          void postorder(nodeptr);
          int bsheight(nodeptr);
          nodeptr srl(nodeptr &);
          nodeptr drl(nodeptr &);
          nodeptr srr(nodeptr &);
          nodeptr drr(nodeptr &);
          int max(int,int);
          int nonodes(nodeptr);
};

//                    Inserting a node
void bstree::insert(int x,nodeptr &p)
{
  if (p == NULL)
  {
          p = new node;
          p->element = x;
          p->left=NULL;
          p->right = NULL;
          p->height=0;
```

```
        if (p==NULL)
                cout<<"Out of Space";
  }
  else
  {
        if (x<p->element)
        {
          insert(x,p->left);
          if ((bsheight(p->left) - bsheight(p->right))==2)
          {
            if (x < p->left->element)
                p=srl(p);
            else
                p = drl(p);
          }
        }
        else if (x>p->element)
        {
            insert(x,p->right);
            if ((bsheight(p->right) - bsheight(p->left))==2)
            {
                if (x > p->right->element)
                        p=srr(p);
                else
                        p = drr(p);
           }
        }
        else
                cout<<"Element Exists";
        }
        int m,n,d;
        m=bsheight(p->left);
        n=bsheight(p->right);
        d=max(m,n);
        p->height = d + 1;

}

//              Finding the Smallest
nodeptr bstree::findmin(nodeptr p)
{
        if (p==NULL)
        {
           cout<<"Empty Tree";
           return p;
        }
        else
        {
          while(p->left !=NULL)
                p=p->left;
          return p;
        }
}

//              Finding the Largest
nodeptr bstree::findmax(nodeptr p)
{
```

```
        if (p==NULL)
        {
          cout<<"Empty Tree";
          return p;
        }
        else
        {
          while(p->right !=NULL)
             p=p->right;
          return p;
        }
}

//              Finding an element
void bstree::find(int x,nodeptr &p)
{
        if (p==NULL)
          cout<<"Element not found";
        else
        if (x < p->element)
          find(x,p->left);
        else
        if (x>p->element)
          find(x,p->right);
        else
          cout<<"Element found !";

}

//              Copy a tree
void bstree::copy(nodeptr &p,nodeptr &p1)
{
        makeempty(p1);
        p1 = nodecopy(p);
}

//              Make a tree empty
void bstree::makeempty(nodeptr &p)
{
        nodeptr d;
        if (p != NULL)
        {
          makeempty(p->left);
          makeempty(p->right);
          d=p;
          free(d);
          p=NULL;
        }
}

//              Copy the nodes
nodeptr bstree::nodecopy(nodeptr &p)
{
        nodeptr temp;
        if (p==NULL)
          return p;
        else
```

```
        {
          temp = new node;
          temp->element = p->element;
          temp->left = nodecopy(p->left);
          temp->right = nodecopy(p->right);
          return temp;
        }
}

//                  Deleting a node
void bstree::del(int x,nodeptr &p)
{
        nodeptr d;
        if (p==NULL)
          cout<<"Element not found ";
        else if ( x < p->element)
          del(x,p->left);
        else if (x > p->element)
          del(x,p->right);
        else if ((p->left == NULL) && (p->right == NULL))
        {
          d=p;
          free(d);
          p=NULL;
          cout<<" Element deleted !";
        }
        else if (p->left == NULL)
        {
         d=p;
         free(d);
         p=p->right;
         cout<<" Element deleted !";
        }
        else if (p->right == NULL)
        {
         d=p;
         p=p->left;
         free(d);
         cout<<" Element deleted !";
        }
        else
         p->element = deletemin(p->right);
}

int bstree::deletemin(nodeptr &p)
{
        int c;
        cout<<"inside deltemin";
        if (p->left == NULL)
        {
         c=p->element;
         p=p->right;
         return c;
        }
        else
        {
         c=deletemin(p->left);
```

```
            return c;
          }
}

void bstree::preorder(nodeptr p)
{
          if (p!=NULL)
          {
            cout<<p->element<<"-->";
            preorder(p->left);
            preorder(p->right);
          }
}

//                    Inorder Printing
void bstree::inorder(nodeptr p)
{
          if (p!=NULL)
          {
            inorder(p->left);
            cout<<p->element<<"-->";
            inorder(p->right);
     }
}

//                    PostOrder Printing
void bstree::postorder(nodeptr p)
{
     if (p!=NULL)
     {
            postorder(p->left);
            postorder(p->right);
            cout<<p->element<<"-->";
          }
}

int bstree::max(int value1, int value2)
{
          return ((value1 > value2) ? value1 : value2);
}

int bstree::bsheight(nodeptr p)
{
          int t;
          if (p == NULL)
                    return -1;
          else
          {
                    t = p->height;
                    return t;
          }
}

nodeptr bstree:: srl(nodeptr &p1)
{
          nodeptr p2;
          p2 = p1->left;
```

```
        p1->left = p2->right;
        p2->right = p1;
        p1->height = max(bsheight(p1->left),bsheight(p1->right)) + 1;
        p2->height = max(bsheight(p2->left),p1->height) + 1;
        return p2;
}

nodeptr bstree:: srr(nodeptr &p1)
{
        nodeptr p2;
        p2 = p1->right;
        p1->right = p2->left;
        p2->left = p1;
        p1->height = max(bsheight(p1->left),bsheight(p1->right)) + 1;
        p2->height = max(p1->height,bsheight(p2->right)) + 1;
        return p2;
}


nodeptr bstree:: drl(nodeptr &p1)
{
        p1->left=srr(p1->left);
        return srl(p1);
}

nodeptr bstree::drr(nodeptr &p1)
{
        p1->right = srl(p1->right);
        return srr(p1);
}

int bstree::nonodes(nodeptr p)
{
        int count=0;
        if (p!=NULL)
        {
                nonodes(p->left);
                nonodes(p->right);
                count++;
        }
        return count;

}



int main()
{
        clrscr();
        nodeptr root,root1,min,max;//,flag;
        int a,choice,findele,delele,leftele,rightele,flag;
        char ch='y';
        bstree bst;
        //system("clear");
        root = NULL;
        root1=NULL;
        cout<<"          AVL Tree";
```

```
	cout<<"		========";
	do
	{
		cout<<"		1.Insertion
		2.FindMin
		";
		cout<<"3.FindMax
		4.Find
		5.Copy
		";
		cout<<"6.Delete
		7.Preorder
		8.Inorder
";
		cout<<"		9.Postorder
		10.height
";
		cout<<"
Enter the choice:
";
		cin>>choice;
		switch(choice)
		{
		case 1:
			cout<<"New node's value ?";
			cin>>a;
			bst.insert(a,root);
			break;
		case 2:
			if (root !=NULL)
			{
			min=bst.findmin(root);
			cout<<"
Min element :   "<<min->element;
			}
			break;
		 case 3:
		if (root !=NULL)
		{
			max=bst.findmax(root);
			cout<<"
Max element :   "<<max->element;
			}
			break;
		case 4:
			cout<<"
Search node :
";
			cin>>findele;
			if (root != NULL)
				bst.find(findele,root);
			break;
		case 5:
			bst.copy(root,root1);
			bst.inorder(root1);
			break;
		case 6:
```

```
                        cout<<"Delete Node ?";
                        cin>>delele;
                        bst.del(delele,root);
                        bst.inorder(root);
                        break;

                case 7:
                        cout<<" Preorder Printing... :";
                        bst.preorder(root);
                        break;

                case 8:
               cout<<" Inorder Printing.... :";
                bst.inorder(root);
                break;

                case 9:
                cout<<" Postorder Printing... :";
                bst.postorder(root);
                break;
                case 10:
                        cout<<" Height and Depth is";
                        cout<<bst.bsheight(root);
                        cout<<"No. of nodes:- "<<bst.nonodes(root);
                        break;

                }
                cout<<" Do u want to continue (y/n) ?";
                cin>>ch;
        }while(ch=='y');


        return 0;
}
```

Quick Sort Implementation

```
#include<process.h>
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>

int Partition(int low,int high,int arr[]);
void Quick_sort(int low,int high,int arr[]);

void main()
{
int *a,n,low,high,i;
clrscr();
cout<<"/*************************Quick Sort Algorithm
Implementation*****************/

";
cout<<"Enter number of elements:
";
cin>>n;
```

```
a=new int[n];
/* cout<<"enter the elements:";
for(i=0;i<n;i++)
cin>>a;*/
for(i=0;i<n;i++)
a[i]=rand()%100;
clrscr();
cout<<"Initial Order of elements";
 for(i=0;i<n;i++)
  cout<<a[i]<<"   ";
  cout<<"";

high=n-1;
low=0;
Quick_sort(low,high,a);
cout<<"Final Array After Sorting:";

  for(i=0;i<n;i++)
  cout<<a[i]<<"   ";

getch();
}

/*Function for partitioning the array*/

int Partition(int low,int high,int arr[])
{ int i,high_vac,low_vac,pivot/*,itr*/;
   pivot=arr[low];
   while(high>low)
{ high_vac=arr[high];

  while(pivot<high_vac)
  {
    if(high<=low) break;
    high--;
    high_vac=arr[high];
  }

  arr[low]=high_vac;
  low_vac=arr[low];
  while(pivot>low_vac)
  {
    if(high<=low) break;
    low++;
    low_vac=arr[low];
  }
  arr[high]=low_vac;
}
  arr[low]=pivot;
   return low;
}

void Quick_sort(int low,int high,int arr[])
{
  int Piv_index,i;
  if(low<high)
```

```
 {
  Piv_index=Partition(low,high,arr);
  Quick_sort(low,Piv_index-1,arr);
  Quick_sort(Piv_index+1,high,arr);
 }
}
```

Sorted Doubly Linked List with Insertion and Deletion

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

class Dllist
{
  private:
     typedef struct Node
     {
       string name;
       Node* next;
       Node* prev;
     };
     Node* head;
     Node* last;
  public:
     Dllist()
     {
      head = NULL;
      last = NULL;
     }
     bool empty() const { return head==NULL; }
     friend ostream& operator<<(ostream& ,const Dllist& );
     void Insert(const string& );
     void Remove(const string& );
};

void Dllist::Insert(const string& s)
{
   // Insertion into an Empty List.
   if(empty())
   {
     Node* temp = new Node;
     head = temp;
     last = temp;
     temp->prev = NULL;
     temp->next = NULL;
     temp->name = s;
   }
   else
   {
     Node* curr;
     curr = head;
     while( s>curr->name && curr->next != last->next) curr = curr->next;

     if(curr == head)
     {
       Node* temp = new Node;
```

```
        temp->name = s;
        temp->prev = curr;
        temp->next = NULL;
        head->next = temp;
        last = temp;
      //  cout<<" Inserted "<<s<<" After "<<curr->name<<endl;
       }
       else
       {
       if(curr == last && s>last->name)
       {
         last->next = new Node;
         (last->next)->prev = last;
         last = last->next;
         last->next = NULL;
         last->name = s;
      //  cout<<" Added "<<s<<" at the end "<<endl;
       }
       else
       {
         Node* temp = new Node;
         temp->name = s;
         temp->next = curr;
         (curr->prev)->next = temp;
         temp->prev = curr->prev;
         curr->prev = temp;
      //  cout<<" Inserted "<<s<<" Before "<<curr->name<<endl;
       }
      }
    }
}

ostream& operator<<(ostream& ostr, const Dllist& dl )
{
   if(dl.empty()) ostr<<" The list is empty. "<<endl;
   else
   {
      Dllist::Node* curr;
      for(curr = dl.head; curr != dl.last->next; curr=curr->next)
        ostr<<curr->name<<" ";
      ostr<<endl;
      ostr<<endl;
      return ostr;
   }
}

void Dllist::Remove(const string& s)
{
   bool found = false;
   if(empty())
   {
    cout<<" This is an empty list! "<<endl;
    return;
   }
   else
   {
    Node* curr;
```

```
    for(curr = head; curr != last->next; curr = curr->next)
    {
       if(curr->name == s)
       {
         found = true;
         break;
       }
    }
    if(found == false)
    {
     cout<<" The list does not contain specified Node"<<endl;
     return;
    }
    else
    {
      // Curr points to the node to be removed.
      if (curr == head && found)
      {
        if(curr->next != NULL)
        {
         head = curr->next;
         delete curr;
         return;
        }
        else
        {
         delete curr;
         head = NULL;
         last = NULL;
         return;
        }
      }
      if (curr == last && found)
      {
       last = curr->prev;
       delete curr;
       return;
      }
     (curr->prev)->next = curr->next;
     (curr->next)->prev = curr->prev;
      delete curr;
    }
  }
}

int main()
{
   Dllist d1;
   int ch;
   string temp;
   while(1)
   {
     cout<<endl;
     cout<<" Doubly Linked List Operations "<<endl;
     cout<<" ------------------------------"<<endl;
     cout<<" 1. Insertion "<<endl;
     cout<<" 2. Deletion "<<endl;
```

```
    cout<<" 3. Display "<<endl;
    cout<<" 4. Exit "<<endl;
    cout<<" Enter your choice : ";
    cin>>ch;
    switch(ch)
    {
      case 1: cout<<" Enter Name to be inserted : ";
            cin>>temp;
            d1.Insert(temp);
            break;
      case 2: cout<<" Enter Name to be deleted : ";
            cin>>temp;
            d1.Remove(temp);
            break;
      case 3: cout<<" The List contains : ";
            cout<<d1;
            break;
      case 4: system("pause");
            return 0;
            break;
    }
  }
```

Graphs program

```
#include <iostream.h>
#include <conio.h>

class graph
{
        private:int n;
                int **a;
                int *reach;
                int *pos;
        public:graph(int k=10);
                void create();
                void dfs();
                void dfs(int v,int label);
                int begin(int v);
                int nextvert(int v);
};
void graph::graph(int k)
{
        n=k;
        a=new int *[n+1];
        reach=new int[n+1];
        pos=new int [n+1];
        for(int i=1;i<=n;i++)
                pos[i]=0;
        for(int j=1;j<=n;j++)
                a[j]=new int[n+1];
}
void graph::create()
{
        for(int i=1;i<=n;i++)
        {
                cout<<"Enter the "<<i<<"th row of matrix a:
";
```

```
                for(int j=1;j<=n;j++)
                        cin>>a[i][j];
        }
        for(int k=1;k<=n;k++)
                reach[k]=0;
}
void graph::dfs()
{
        int label=0;
        for(int i=1;i<=n;i++)
                if(!reach[i])
                {
                        label++;
                        dfs(i,label);
                }
        cout<<"
The contents of the reach array is:
;
        for(int j=1;j<=n;j++)
                cout<<reach[j]<<"       ";
}
void graph::dfs(int v,int label)
{
        cout<<v<<"      ";
        reach[v]=label;
        int u=begin(v);
        while(u)
        {
                if(!reach[u])
                        dfs(u,label);
                u=nextvert(v);
        }
}
int graph::begin(int v)
{
        if((v<1)&&(v>n))
                cout<<"Bad input
";
        else
                for(int i=1;i<=n;i++)
                        if(a[v][i]==1)
                        {
                                pos[v]=i;
                                return i;
                        }
        return 0;
}
int graph::nextvert(int v)
{
        if((v<1)&&(v>n))
                cout<<"Bad input
";
        else
                for(int i=pos[v]+1;i<=n;i++)
                        if(a[v][i]==1)
                        {
                                pos[v]=i;
```

```
                                        return i;
                                }
        return 0;
}
void main()
{
        clrscr();
        int x;
        cout<<"Enter the no of vertices:
";
        cin>>x;
        graph g(x);
        g.create();
        cout<<"dfs is.....";
        g.dfs();
        getch();
}
```

Binary Search tree with insertion, deletion, finding an element,
min element, max element, left child, right child, recursive and nonrecursive traversals,
finding the number of nodes, leaves, fullnodes, ancestors, descendants

```
# include <conio.h>
# include <process.h>
# include <iostream.h>
# include <alloc.h>

struct node
{
        int ele;
        node *left;
        node *right;
};

typedef struct node *nodeptr;
static int nodes=0;
static int leaves=0;
static int full=0;
class stack
{
        private:
                struct snode
                {
                        nodeptr ele;
                        snode *next;
                };
                snode *top;
        public:
                stack()
                {
                        top=NULL;
                }
                void push(nodeptr p)
                {
                        snode *temp;
                        temp = new snode;
                        temp->ele = p;
```

```
                temp->next = top;
                top=temp;
        }

        void pop()
        {
                if (top != NULL)
                {
                nodeptr t;
                snode *temp;
                temp = top;
                top=temp->next;
                delete temp;
                }
        }

        nodeptr topele()
        {
                if (top !=NULL)
                        return top->ele;
                else
                        return NULL;
        }

        int isempty()
        {
        return ((top == NULL) ? 1 : 0);
        }

};


class bstree
{
        public:
                void insert(int,nodeptr &);
                void del(int,nodeptr &);
                int deletemin(nodeptr &);
                void find(int,nodeptr &);
                nodeptr findmin(nodeptr);
                nodeptr findmax(nodeptr);
                void copy(nodeptr &,nodeptr &);
                void makeempty(nodeptr &);
                nodeptr nodecopy(nodeptr &);
                void nonodes(nodeptr);
                void fullnodes(nodeptr);
                void ances(int,nodeptr &);
                void desc(int,nodeptr &);
                void allleaves(nodeptr);
                void noleaves(nodeptr);
                void preorder(nodeptr);
                void inorder(nodeptr);
                void postorder(nodeptr);
                void preordernr(nodeptr);
                void inordernr(nodeptr);
                void postordernr(nodeptr);
                void leftchild(int,nodeptr &);
```

```
                void rightchild(int,nodeptr &);
};

void bstree::insert(int x,nodeptr &p)
{
        if (p==NULL)
        {
                p = new node;
                p->ele=x;
                p->left=NULL;
                p->right=NULL;
        }
        else
        {
                if (x < p->ele)
                        insert(x,p->left);
                else if (x>p->ele)
                        insert(x,p->right);
                else
                        cout<<"Element already Exits !";
        }
}

void bstree:: del(int x,nodeptr &p)
{
        nodeptr d;
        if (p==NULL)
                cout<<"Element not found ";
        else if (x < p->ele)
                del(x,p->left);
        else if (x > p->ele)
                del(x,p->right);
        else if ((p->left == NULL) && (p->right ==NULL))
        {
                d=p;
                free(d);
                p=NULL;
        }
         else if (p->left == NULL)
        {
                d=p;
                free(d);
                p=p->right;
        }
        else if (p->right ==NULL)
        {
                d=p;
                p=p->left;
                free(d);
        }
        else
        p->ele=deletemin(p->right);
}

int bstree::deletemin(nodeptr &p)
{
        int c;
```

```
        if (p->left == NULL)
        {
                c=p->ele;
                p=p->right;
                return c;
        }
        else
                c=deletemin(p->left);
                return c;
}

void bstree::copy(nodeptr &p,nodeptr &p1)
{
        makeempty(p1);
        p1=nodecopy(p);
}

void bstree::makeempty(nodeptr &p)
{
        nodeptr d;
        if (p!=NULL)
        {
                makeempty(p->left);
                makeempty(p->right);
                d=p;
                free(d);
                p=NULL;
        }
}

nodeptr bstree::nodecopy(nodeptr &p)
{
        nodeptr temp;
        if (p == NULL)
                return p;
        else
        {
                temp = new node;
                temp->ele=p->ele;
                temp->left = nodecopy(p->left);
                temp->right = nodecopy(p->right);
                return temp;
        }
}


nodeptr bstree::findmin(nodeptr p)
{
        if (p==NULL)
        {
                cout<<"Tree is empty !";
                return p;
        }
        else
        {
                while (p->left !=NULL)
                        p=p->left;
```

```
                return p;
        }
}



nodeptr bstree::findmax(nodeptr p)
{
        if (p==NULL)
        {
                cout<<"Tree is empty !";
                return p;
        }
        else
        {
                while (p->right !=NULL)
                        p=p->right;
                return p;
        }
}

void bstree::find(int x,nodeptr &p)
{
        if (p==NULL)
                cout<<"Element not found  !";
        else
        {
                if (x <p->ele)
                        find(x,p->left);
                else if ( x> p->ele)
                        find(x,p->right);
                else
                        cout<<"Element Found !";
        }
}
void bstree::desc(int x,nodeptr &p)
{
        if (p==NULL)
                cout<<"Element not found  !";
        else
        {
                if (x <p->ele)
                        desc(x,p->left);
                else if ( x> p->ele)
                        desc(x,p->right);
                else
                {
                        if (p->left !=NULL)
                        preorder(p->left);
                        else
                        preorder(p->right);
                }
                        //cout<<"Element Found !";
        }
}

void bstree::ances(int x,nodeptr &p)
```

```
{
	if (p==NULL)
		cout<<"Element not found  !";
	else
	{
		if (x <p->ele)
			{
			cout<<p->ele<<"-->";
			ances(x,p->left);
			}
		else if ( x> p->ele)
			{
			cout<<p->ele<<"-->";
			ances(x,p->right);
			}
		else
			cout<<"Element Found !";
	}
}

void bstree::nonodes(nodeptr p)
{
	if (p!=NULL)
	{
		nodes++;
		nonodes(p->left);
		nonodes(p->right);
	}
}
void bstree::noleaves(nodeptr p)
{
	if (p!=NULL)
	{
		noleaves(p->left);
		if ((p->left == NULL) && (p->right == NULL))
			leaves++;
		noleaves(p->right);
	}
}
void bstree::allleaves(nodeptr p)
{
	if (p!=NULL)
	{
		allleaves(p->left);
		if ((p->left == NULL) && (p->right == NULL))
			cout<<p->ele<<"-->";
		allleaves(p->right);
	}
}

void bstree::fullnodes(nodeptr p)
{
	if (p!=NULL)
	{
		fullnodes(p->left);
		if ((p->left != NULL) && (p->right != NULL))
			full++;
```

```
                fullnodes(p->right);
        }
}

void bstree::preorder(nodeptr p)
{
        if (p!=NULL)
        {
                cout<<p->ele<<"-->";
                preorder(p->left);
                preorder(p->right);
        }
}
void bstree::inorder(nodeptr p)
{
        if (p!=NULL)
        {
                inorder(p->left);
                cout<<p->ele<<"-->";
                inorder(p->right);
        }
}

void bstree::postorder(nodeptr p)
{
        if (p!=NULL)
        {
                postorder(p->left);
                postorder(p->right);
                cout<<p->ele<<"-->";
        }
}



void bstree::preordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
        {
                cout<<p->ele<<"-->";
                s.push(p);
                p=p->left;
        }
        else
        if (s.isempty())
        {
                cout<<"Stack is empty";
                return;
        }
        else
        {
                nodeptr t;
                t=s.topele();
                p=t->right;
```

```
                s.pop();
        }
        }
}


void bstree::inordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
        {
                s.push(p);
                p=p->left;
        }
        else
        {
        if (s.isempty())
        {
                cout<<"Stack is empty";
                return;
        }
        else
        {
                p=s.topele();
                cout<<p->ele<<"-->";
        }
        s.pop();
        p=p->right;
        }
        }
}


void bstree::postordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
        {
                s.push(p);
                p=p->left;
        }
        else
        {
                if (s.isempty())
                {
                        cout<<"Stack is empty";
                        return;
                }
                else
                if (s.topele()->right == NULL)
                {
                        p=s.topele();
                        s.pop();
```

```
                        cout<<p->ele<<"-->";
                        if (p==s.topele()->right)
                        {
                                cout<<s.topele()->ele<<"-->";
                                s.pop();
                        }
                        if (!s.isempty())
                                p=s.topele()->right;
                        else
                                p=NULL;
                }
        }
        }
}

void bstree::leftchild(int q,nodeptr &p)
{
        if (p==NULL)
                cout<<"The node does not exists ";
        else
        if (q < p->ele )
                leftchild(q,p->left);
        else
        if (q > p->ele)
                leftchild(q,p->right);
        else
        if (q == p->ele)
        {
                if (p->left != NULL)
                        cout<<"Left child of "<<q<<"is "<<p->left->ele;
                else
                        cout<<"No Left child !";
        }
}

void bstree::rightchild(int q,nodeptr &p)
{
        if (p==NULL)
                cout<<"The node does not exists ";
        else
        if (q < p->ele )
                rightchild(q,p->left);
        else
        if (q > p->ele)
                rightchild(q,p->right);
        else
        if (q == p->ele)
        {
                if (p->right != NULL)
                        cout<<"Right child of "<<q<<"is "<<p->right->ele;
                else
                        cout<<"No Right Child !";
        }
}

int main()
{
```

```
int ch,x,leftele,rightele;
bstree bst;
char c='y';
nodeptr root,root1,min,max;
root=NULL;
root1=NULL;
do
{
//      system("clear");
        clrscr();
        cout<<"
                        Binary Search Tree
";
        cout<<"                 ----------
";
        cout<<"         1.Insertion
                2.Deletion
                3.NodeCopy
";
        cout<<"         4.Find
                5.Findmax
                6.Findmin
";
        cout<<"         7.Preorder
                8.Inorder
                9.Postorder";
        cout<<"
                10.Leftchild
                11.Rightchild
                12.Counting
";
        cout<<"
Enter your choice :";
        cin>>ch;

        switch(ch)
        {
        case 1:
                cout<<"
                1.Insertion
";
                cout<<"Node Element ?
";
                cin>>x;
                bst.insert(x,root);
                cout<<"Inorder traversal is :
";
                bst.inorder(root);
                break;

        case 2:
                cout<<"
                2.Deletion
";
                cout<<"Delete Element ?
";
                cin>>x;
```

```
                bst.del(x,root);
                bst.inorder(root);
                break;

        case 3:
                cout<<"
                3.Nodecopy
";
                bst.copy(root,root1);
                cout<<"
The new tree is :
";
                bst.inorder(root1);
                break;

        case 4:
                cout<<"
                4.Find
";
                cout<<"Search Element ?
";
                cin>>x;
                bst.find(x,root);
                cout<<"
 The ancestors are :
";
                bst.ances(x,root);
                cout<<"
 The descendants are :
";
                bst.desc(x,root);
                break;

        case 5:
                cout<<"
                5.Findmax
";
                if (root == NULL)
                        cout<<"
Tree is empty";
                else
                {
                max=bst.findmax(root);
                cout<<"Largest element is :        "<<max->ele<<endl;
                }
                break;

        case 6:
                cout<<"
                6.Findmin
";
                if (root == NULL)
                        cout<<"
Tree is empty";
                else
                {
                min=bst.findmin(root);
```

```
                cout<<"Smallest element is :      "<<min->ele<<endl;
                }
                break;

        case 7:
                cout<<"
                7.Preorder
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                cout<<"
Preorder traversal (Non-Recursive) is :
";
                bst.preordernr(root);
                cout<<"
Preorder traversal (Recursive) is :
";
                bst.preorder(root);
                }
                break;

        case 8:
                cout<<"
                8.Inorder
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                cout<<"
Inorder traversal (Non-Recursive) is :
";
                bst.inordernr(root);
                cout<<"
Inorder traversal (Recursive) is :
";
                bst.inorder(root);
                }
                break;

        case 9:
                cout<<"
                9.Postorder
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                cout<<"
Postorder traversal (Non-Recursive) is :
";
                bst.postordernr(root);
```

```
                cout<<"
Postorder traversal (Recursive) is :
";
                bst.postorder(root);
                }
                break;

        case 10:
                cout<<"
                10.Finding the left Child
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                cout<<"Parent of the left child ?
";
                cin>>leftele;
                bst.leftchild(leftele,root);
                }
                break;

        case 11:
                cout<<"
                11.Finding the Right Child
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                cout<<"Parent of the right child ?
";
                cin>>rightele;
                bst.rightchild(rightele,root);
                }
                break;
        case 12:
                bst.nonodes(root);
                cout<<"Number of nodes : "<<nodes<<endl;
                nodes=0;
                bst.noleaves(root);
                cout<<"Number of leaves :"<<leaves<<endl;
                leaves=0;
                bst.fullnodes(root);
                cout<<"Number of fullnodes :"<<full<<endl;
                full=0;
                cout<<"All leaf nodes are :
";
                bst.allleaves(root);
                break;
        }
        cout<<"
Continue (y/n) ?
";
        cin>>c;
```

```
        }while (c=='y' || c == 'Y');
        return 0;
}
```

Binary Search Tree with non-recursive traversals

This program includes the inserting a node, deleting a node,recursive tree traversal,non-recursive tree traversal,finding the
minimum,maximum,leftchild,rightchild,copy a tree to another,making a tree null.

Code :

```
# include <conio.h>
# include <process.h>
# include <iostream.h>
# include <alloc.h>

struct node
{
        int ele;
        node *left;
        node *right;
};

typedef struct node *nodeptr;
class stack
{
        private:
                struct snode
                {
                        nodeptr ele;
                        snode *next;
                };
                snode *top;
        public:
                stack()
                {
                        top=NULL;
                }
                void push(nodeptr p)
                {
                        snode *temp;
                        temp = new snode;
                        temp->ele = p;
                        temp->next = top;
                        top=temp;
                }

                void pop()
                {
                        if (top != NULL)
                        {
                        nodeptr t;
                        snode *temp;
                        temp = top;
                        top=temp->next;
                        delete temp;
                        }
```

```
		}

		nodeptr topele()
		{
			if (top !=NULL)
				return top->ele;
			else
				return NULL;
		}



		int isempty()
		{
		return ((top == NULL) ? 1 : 0);
		}

};


class bstree
{
	public:
		void insert(int,nodeptr &);
		void del(int,nodeptr &);
		int deletemin(nodeptr &);
		void find(int,nodeptr &);
		nodeptr findmin(nodeptr);
		nodeptr findmax(nodeptr);
		void copy(nodeptr &,nodeptr &);
		void makeempty(nodeptr &);
		nodeptr nodecopy(nodeptr &);
		void preorder(nodeptr);
		void inorder(nodeptr);
		void postorder(nodeptr);
		void preordernr(nodeptr);
		void inordernr(nodeptr);
		void postordernr(nodeptr);
		void leftchild(int,nodeptr &);
		void rightchild(int,nodeptr &);


};

void bstree::insert(int x,nodeptr &p)
{
	if (p==NULL)
	{
		p = new node;
		p->ele=x;
		p->left=NULL;
		p->right=NULL;
	}
	else
	{
		if (x < p->ele)
			insert(x,p->left);
```

```
                else if (x>p->ele)
                        insert(x,p->right);
                else
                        cout<<"Element already Exits !";
        }
}

void bstree:: del(int x,nodeptr &p)
{
        nodeptr d;
        if (p==NULL)
                cout<<"Element not found ";
        else if (x < p->ele)
                del(x,p->left);
        else if (x > p->ele)
                del(x,p->right);
        else if ((p->left == NULL) && (p->right ==NULL))
        {
                d=p;
                free(d);
                p=NULL;
        }
         else if (p->left == NULL)
        {
                d=p;
                free(d);
                p=p->right;
        }
        else if (p->right ==NULL)
        {
                d=p;
                p=p->left;
                free(d);
        }
        else
        p->ele=deletemin(p->right);
}

int bstree::deletemin(nodeptr &p)
{
        int c;
        if (p->left == NULL)
        {
                c=p->ele;
                p=p->right;
                return c;
        }
        else
                c=deletemin(p->left);
                return c;
}

void bstree::copy(nodeptr &p,nodeptr &p1)
{
        makeempty(p1);
        p1=nodecopy(p);
}
```

```
void bstree::makeempty(nodeptr &p)
{
	nodeptr d;
	if (p!=NULL)
	{
		makeempty(p->left);
		makeempty(p->right);
		d=p;
		free(d);
		p=NULL;
	}
}

nodeptr bstree::nodecopy(nodeptr &p)
{
	nodeptr temp;
	if (p == NULL)
		return p;
	else
	{
		temp = new node;
		temp->ele=p->ele;
		temp->left = nodecopy(p->left);
		temp->right = nodecopy(p->right);
		return temp;
	}
}


nodeptr bstree::findmin(nodeptr p)
{
	if (p==NULL)
	{
		cout<<"Tree is empty !";
		return p;
	}
	else
	{
		while (p->left !=NULL)
			p=p->left;
		return p;
	}
}



nodeptr bstree::findmax(nodeptr p)
{
	if (p==NULL)
	{
		cout<<"Tree is empty !";
		return p;
	}
	else
	{
		while (p->right !=NULL)
```

```
                        p=p->right;
                return p;
        }
}

void bstree::find(int x,nodeptr &p)
{
        if (p==NULL)
                cout<<"Element not found  !";
        else
        {
                if (x <p->ele)
                        find(x,p->left);
                else if ( x> p->ele)
                        find(x,p->right);
                else
                        cout<<"Element Found !";
        }
}

void bstree::preorder(nodeptr p)
{
        if (p!=NULL)
        {
                cout<<p->ele<<"-->";
                preorder(p->left);
                preorder(p->right);
        }
}
void bstree::inorder(nodeptr p)
{
        if (p!=NULL)
        {
                inorder(p->left);
                cout<<p->ele<<"-->";
                inorder(p->right);
        }
}

void bstree::postorder(nodeptr p)
{
        if (p!=NULL)
        {
                postorder(p->left);
                postorder(p->right);
                cout<<p->ele<<"-->";
        }
}



void bstree::preordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
```

```
	{
		cout<<p->ele<<"-->";
		s.push(p);
		p=p->left;
	}
	else
	if (s.isempty())
	{
		cout<<"Stack is empty";
		return;
	}
	else
	{
		nodeptr t;
		t=s.topele();
		p=t->right;
		s.pop();
	}
	}
}


void bstree::inordernr(nodeptr p)
{
	stack s;
	while (1)
	{
	if  (p != NULL)
	{
		s.push(p);
		p=p->left;
	}
	else
	{
	if (s.isempty())
	{
		cout<<"Stack is empty";
		return;
	}
	else
	{
		p=s.topele();
		cout<<p->ele<<"-->";
	}
	s.pop();
	p=p->right;
	}
	}
}


void bstree::postordernr(nodeptr p)
{
	stack s;
	while (1)
	{
	if  (p != NULL)
```

```
	{
		s.push(p);
		p=p->left;

	}
	else
	{
		if (s.isempty())
		{
			cout<<"Stack is empty";
			return;
		}
		else
		if (s.topele()->right == NULL)
		{
			p=s.topele();
			s.pop();
			cout<<p->ele<<"-->";
			if (p==s.topele()->right)
			{
				cout<<s.topele()->ele<<"-->";
				s.pop();
			}
		}
		if (!s.isempty())
			p=s.topele()->right;
		else
			p=NULL;
	}
	}
}

void bstree::leftchild(int q,nodeptr &p)
{
	if (p==NULL)
		cout<<"The node does not exists ";

	else
	if (q < p->ele )
		leftchild(q,p->left);
	else
	if (q > p->ele)
		leftchild(q,p->right);
	else
	if (q == p->ele)
	{
		if (p->left != NULL)
			cout<<"Left child of "<<q<<"is "<<p->left->ele;
		else
			cout<<"No Left child !";
	}
}

void bstree::rightchild(int q,nodeptr &p)
{
	if (p==NULL)
		cout<<"The node does not exists ";
```

```
        else
        if (q < p->ele )
                rightchild(q,p->left);
        else
        if (q > p->ele)
                rightchild(q,p->right);
        else
        if (q == p->ele)
        {
                if (p->right != NULL)
                        cout<<"Right child of "<<q<<"is "<<p->right->ele;
                else
                        cout<<"No Right Child !";
        }
}




int main()
{
int ch,x,leftele,rightele;
bstree bst;
char c='y';
nodeptr root,root1,min,max;
root=NULL;
root1=NULL;
do
{
//      system("clear");
        clrscr();
        cout<<"
Binary Search Tree
";
        cout<<"------------------------
";
        cout<<"         1.Insertion
                2.Deletion
                3.NodeCopy
";
        cout<<"         4.Find
                5.Findmax
                6.Findmin
";
        cout<<"         7.Preorder
                8.Inorder
                9.Postorder
";
        cout<<"
                10.Leftchild
                11.Rightchild
                0.Exit
";
        cout<<"
Enter your choice :";
        cin>>ch;
```

```
	switch(ch)
	{
	case 1:
		cout<<"
		1.Insertion
";
		cout<<"Enter the new element to get inserted :
";
		cin>>x;

		bst.insert(x,root);
		cout<<"Inorder traversal is :
";
		bst.inorder(root);
		break;

	case 2:
		cout<<"
		2.Deletion
";
		cout<<"Enter the element to get deleted :
";
		cin>>x;
		bst.del(x,root);
		bst.inorder(root);
		break;

	case 3:
		cout<<"
		3.Nodecopy
";
		bst.copy(root,root1);
		cout<<"
The new tree is :
";
		bst.inorder(root1);
		break;

	case 4:
		cout<<"
		4.Find
";
		cout<<"Enter the element to be searched :
";
		cin>>x;
		bst.find(x,root);
		break;

	case 5:
		cout<<"
		5.Findmax
";
		if (root == NULL)
			cout<<"
Tree is empty";
		else
```

```
                {
                        max=bst.findmax(root);
                        cout<<"Largest element is :       "<<max->ele<<endl;
                }
                break;

        case 6:
                cout<<"
                6.Findmin
";
                if (root == NULL)
                        cout<<"
Tree is empty";
                else
                {
                        min=bst.findmin(root);
                        cout<<"Smallest element is :      "<<min->ele<<endl;
                }
                break;

        case 7:
                cout<<"
                7.Preorder
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                        cout<<"
Preorder traversal (Non-Recursive) is :
";
                        bst.preordernr(root);
                        cout<<"
Preorder traversal (Recursive) is :
";
                        bst.preorder(root);
                }
                break;

        case 8:
                cout<<"
                8.Inorder
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                        cout<<"
Inorder traversal (Non-Recursive) is :
";
                        bst.inordernr(root);
                        cout<<"
Inorder traversal (Recursive) is :
";
                        bst.inorder(root);
```

```
                }
                break;

        case 9:
                cout<<"
                9.Postorder
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                        cout<<"
Postorder traversal (Non-Recursive) is :
";
                        bst.postordernr(root);
                        cout<<"
Postorder traversal (Recursive) is :
";
                        bst.postorder(root);
                }
                break;

        case 10:
                cout<<"
                10.Finding the left Child
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                        cout<<"Enter the node for which the left child is to be found :
";
                        cin>>leftele;
                        bst.leftchild(leftele,root);
                }
                break;

        case 11:
                cout<<"
                11.Finding the Right Child
";
                if (root==NULL)
                        cout<<"
Tree is empty";
                else
                {
                        cout<<"Enter the node for which the Right child is to be found :
";
                        cin>>rightele;
                        bst.rightchild(rightele,root);
                }
                break;


        case 0:
```

```
                    exit(0);
            }
            cout<<"
Continue (y/n) ?
";
            cin>>c;
            }while (c=='y' || c == 'Y');
            return 0;
}
```

Sorted Doubly Linked List with Insertion and Deletion

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

class Dllist
{
  private:
     typedef struct Node
     {
       string name;
       Node* next;
       Node* prev;
     };
     Node* head;
     Node* last;
  public:
     Dllist()
     {
      head = NULL;
      last = NULL;
     }
     bool empty() const { return head==NULL; }
     friend ostream& operator<<(ostream& ,const Dllist& );
     void Insert(const string& );
     void Remove(const string& );
};

void Dllist::Insert(const string& s)
{
   // Insertion into an Empty List.
   if(empty())
   {
     Node* temp = new Node;
     head = temp;
     last = temp;
     temp->prev = NULL;
     temp->next = NULL;
     temp->name = s;
   }
   else
   {
     Node* curr;
     curr = head;
```

```
      while( s>curr->name && curr->next != last->next) curr = curr->next;

      if(curr == head)
      {
        Node* temp = new Node;
        temp->name = s;
        temp->prev = curr;
        temp->next = NULL;
        head->next = temp;
        last = temp;
     //  cout<<" Inserted "<<s<<" After "<<curr->name<<endl;
      }
      else
      {
      if(curr == last && s>last->name)
      {
        last->next = new Node;
        (last->next)->prev = last;
        last = last->next;
        last->next = NULL;
        last->name = s;
     //  cout<<" Added "<<s<<" at the end "<<endl;
      }
      else
      {
        Node* temp = new Node;
        temp->name = s;
        temp->next = curr;
        (curr->prev)->next = temp;
        temp->prev = curr->prev;
        curr->prev = temp;
     //  cout<<" Inserted "<<s<<" Before "<<curr->name<<endl;
      }
     }
    }
}

ostream& operator<<(ostream& ostr, const Dllist& dl )
{
   if(dl.empty()) ostr<<" The list is empty. "<<endl;
   else
   {
      Dllist::Node* curr;
      for(curr = dl.head; curr != dl.last->next; curr=curr->next)
        ostr<<curr->name<<" ";
      ostr<<endl;
      ostr<<endl;
      return ostr;
   }
}

void Dllist::Remove(const string& s)
{
   bool found = false;
   if(empty())
   {
     cout<<" This is an empty list! "<<endl;
```

```
    return;
   }
   else
   {
    Node* curr;
    for(curr = head; curr != last->next; curr = curr->next)
    {
       if(curr->name == s)
       {
         found = true;
         break;
       }
    }
    if(found == false)
    {
     cout<<" The list does not contain specified Node"<<endl;
     return;
    }
    else
    {
      // Curr points to the node to be removed.
      if (curr == head && found)
      {
       if(curr->next != NULL)
       {
        head = curr->next;
        delete curr;
        return;
       }
       else
       {
        delete curr;
        head = NULL;
        last = NULL;
        return;
       }
      }
      if (curr == last && found)
      {
       last = curr->prev;
       delete curr;
       return;
      }
     (curr->prev)->next = curr->next;
     (curr->next)->prev = curr->prev;
      delete curr;
    }
  }
}

int main()
{
   Dllist d1;
   int ch;
   string temp;
   while(1)
   {
```

```
    cout<<endl;
    cout<<" Doubly Linked List Operations "<<endl;
    cout<<" ------------------------------"<<endl;
    cout<<" 1. Insertion "<<endl;
    cout<<" 2. Deletion "<<endl;
    cout<<" 3. Display "<<endl;
    cout<<" 4. Exit "<<endl;
    cout<<" Enter your choice : ";
    cin>>ch;
    switch(ch)
    {
      case 1: cout<<" Enter Name to be inserted : ";
            cin>>temp;
            d1.Insert(temp);
            break;
      case 2: cout<<" Enter Name to be deleted : ";
            cin>>temp;
            d1.Remove(temp);
            break;
      case 3: cout<<" The List contains : ";
            cout<<d1;
            break;
      case 4: system("pause");
            return 0;
            break;
    }
  }
```

To find the BFS and DFS of the given graph

```
#include<stdio.h>
#include<conio.h>

#define size 20

int a[10][10],vertex[10],n,e;

/*STACK FUNCTIONS*/
#define bottom -1

int stack[size],top=bottom;
int stackempty()
{
        return(top=bottom) ? 1:0;
}
int stackfull()
{
        return(top==size-1) ? 1:0;
}

void push(int item)
{
        if(stackfull())
                printf("7

 STACK IS FULL");
        else
```

```
                stack[++top]=item;
}

int pop()
{
        if(stackempty())
        {
                printf("7

 STACK IS EMPTY");
                return -1;
        }
        else
                return stack[top--];
}

int peep()
{
        if(stackempty())
        {
                printf("7

 STACK IS EMPTY");
                return -1;
        }
        else
                return stack[top];
}

/* QUEUE FUNCTIONS */
#define start -1

int q[size];
int f=start,r=start;

int qempty(){ return(f==r)?1:0; }
int qfull(){ return(r==size-1)?1:0;}

void addq(int c)
{
if(qfull())
        printf("7

QUEUE IS FULL");
else
        q[++r]=c;
}

int delq()
{
if(qempty())
        {
        printf("7

QUEUE IS EMPTY");
        return -1;
        }
```

```
else
        return q[++f];
}
// j is unvisited adjecent vertex to i
int adjvertex(int i)
{
int j;
for(j=0;j<n;j++)
        if(a[i][j]==1&&vertex[j]==0)
                return j;
        return n;
}

int  visitall()
{
int i;
for(i=0;i<n;i++)
        if(vertex[i]==0)
                return 0;
        return 1;
}

/*FUNCTION FOR BFS*/
void bfs()
{
int i,j,k,cur=0;//current vertex is startting vertex
for(i=0;i<n;i++)
        vertex[i]=0;//not visited
printf("

BFS path => V%d ",cur+1);
addq(cur);
vertex[cur]=1;//marking visited vertex
        while(!visitall())
                {
                if(qempty())
                        {
                        printf("7

GRAPH IS DISCONNECTED");
                        break;
                        }
                cur=delq();
                //visit all vertices which are adjecent to current vertex
                for(j=0;j<n;j++)
                        {
                        //adjecent are not visited
                        if(a[cur][j]==1&&vertex[j]==0)
                                {
                                printf("V%d ",j+1);
                                addq(j);
                                //marking visited vertex
                                vertex[j]=1;
                                }
                        }
                }
}
```

```
/*FUNCTION FOR DFS*/
void dfs()
{
int i,j,k,cur=0;//current vertex is startting vertex
for(i=0;i<n;i++)
        vertex[i]=0;//not visited
printf("

DFS path => V%d ",cur+1);
push(cur);
vertex[cur]=1;//marking visited vertex
        while(!visitall())
                {
                do
                        {
                        cur=adjvertex(peep());
                        if(cur==n) pop();
                        }
                while(cur==n&&!stackempty());
                if(stackempty())
                        {
                        printf("7

GRAPH IS DISCONNECTED");
                        break;
                        }
                if(cur!=n)
                        {
                        printf(" V%d ",cur+1);
                        push(cur);
                        vertex[cur]=1;//marking visited vertex
                        }
                }
}

/*MAIN PROGRAM*/
void main()
{
        int i,j,k;
        clrscr();
        for(i=0;i<10;i++)
        for(j=0;j<10;j++)
        a[i][j]=0;

        printf("
ENTER NO OF VERTICES & EDGES OF UNDIRECTED GRAPH : ");
        scanf("%d%d",&n,&e);

        printf("
ENTER EDGES AS PAIR OF VERTICES

");
        for(k=1;k<=e;k++)
                {
                printf("EDGE %d =>",k);
                scanf("%d%d",&i,&j);
```

```
                //for undirected graph
                a[i-1][j-1]=1;
                }

        dfs();
        bfs();
        getch();
}
```

Single linked list

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>


class list
{
        struct node
        {
                int data;
                node *link;
        }*p;
public:
        void inslast(int);
        void insbeg(int);
        void insnext(int,int);
        void delelement(int);
        void delbeg();
        void dellast();
        void disp();
        int seek(int);
        list(){p=NULL;}
        ~list();
};

void list::inslast(int x)
{
        node *q,*t;
        if(p==NULL)
        {
                p=new node;
                p->data=x;
                p->link=NULL;
        }
        else
        {
                q=p;
                while(q->link!=NULL)
                        q=q->link;
                t=new node;
                t->data=x;
                t->link=NULL;
                q->link=t;
        }
```

```
	cout<<"
 Inserted successfully at the end..
";
	disp();
}

void list:: insbeg(int x)
{
	node *q;
	q=p;
	p=new node;
	p->data=x;
	p->link=q;
	cout<<"
 Inserted successfully at the begining..
";
	disp();
}


void list::delelement(int x)
{
	node *q,*r;
	q=p;
	if(q->data==x)
	{
		p=q->link;
		delete q;
		return;
	}
	r=q;
	while(q!=NULL)
	{
		if(q->data==x)
		{
			r->link=q->link;
			delete q;
			return;
		}
		r=q;
		q=q->link;
	}
	cout<<"
 Element u entered   "<<x<<"    is not found..
";
}

void list:: delbeg()
{
	cout<<"
 The list before deletion:
";
	disp();
	node *q;
	q=p;
	if(q==NULL)
	{
```

```
                cout<<"
 No data is present..
";
                return;
        }
        p=q->link;
        delete q;
        return;
}


void list:: dellast()
{
        cout<<"
 The list before deletion:
";
        disp();
        node *q,*t;
        q=p;
        if(q==NULL)
        {
                cout<<"
 There is no data in the list..
";
                return;
        }
        if(q->link==NULL)
        {
                p=q->link;
                delete q;
                return;
        }

        while(q->link->link!=NULL)
                q=q->link;
        q->link=NULL;
        return;
}

list::~list()
{
        node *q;
        if(p==NULL) return;
        while(p!=NULL)
        {
                q=p->link;
                delete p;
                p=q;
        }
}

void list::disp()
{
        node *q;
        q=p;
        if(q==NULL)
        {
```

```
                cout<<"
 No data is in the list..
";
                return;
        }
        cout<<"
 The items present in the list are :
";
        while(q!=NULL)
        {
                cout<<" "<<q->data;
                q=q->link;
        }
}

void list :: insnext(int value,int position)
{
        node *temp,*temp1;
        temp=p;
        if(temp1==NULL)
        {
                temp1= new node;
                temp1->data=value;
                temp1->link=NULL;
                p=temp1;
                return;
        }
        for(int i=0;((i<position)&&(temp->link!=NULL)) ;i++)
        {
                if(i==(position-1))
                {
                        temp1= new node;
                        temp1->data= value;
                        temp1->link=temp->link;
                        temp->link=temp1;
                }
                temp=temp->link;
        }
        //cout<<"
 Inserted successfully at the position.."<<position;
        disp();
}


int list::seek(int value)
{
        node *temp;
        temp=p;
        int position=0;
        while(temp!=NULL)
        {
                if(temp->data==value)
                        return position+1;
                else
                {
                        temp=temp->link;
                        position=position+1;
```

```
                }
        }
        cout<<"

 Element "<<value<<" not found";
        return 0;
}


void main()
{
list l;
int ch,v,p,ps;
do
{
        clrscr();
        cout<<"
 Operations on List..
";
        cout<<"
1.Insertion
2.Deletion
3.Display
4.Seek
5.Exit";
        cout<<"
 Enter ur choice:";
        cin>>ch;

        switch(ch)
        {
        case 1:
                cout<<"
1.Insertion at begining
2.Insertion at the end
";
                cout<<"3.Insertion after the mentioned position
";
                cout<<"
 Enter ur choice:";
                cin>>ps;
                cout<<"
 Enter the value to insert:";
                cin>>v;
                switch(ps)
                {
                        case 1:
                                l.insbeg(v);
                                break;
                        case 2:
                                l.inslast(v);
                                break;
                        case 3:
                                cout<<"
 Enter the position to insert the value:";
                                cin>>p;
                                l.insnext(v,p);
```

```
                                        break;

                                default:
                                        cout<<"
 The choice is invalid
";
                                        return;
                        }
                break;

                case 2:
                        cout<<"
1.Delete the first element
2.Delete the last element";
                        cout<<"
3.Enter the element to delete from the list";
                        cout<<"
 Enter ur choice:";
                        cin>>ps;
                        switch(ps)
                        {
                                case 1:
                                        l.delbeg();
                                        cout<<"
 The list after deletion:
";l.disp();
                                        break;
                                case 2:
                                        l.dellast();
                                        cout<<"
 The list after deletion:
";l.disp();
                                        break;
                                case 3:
                                        l.disp();
                                        cout<<"
 Enter the element to delete :   ";
                                        cin>>v;
                                        l.delelement(v);
                                        cout<<"
 The list after deletion:
";l.disp();
                                        break;

                                default:
                                        cout<<"
 The option is invalid...
";
                                        break;
                        }
                break;

                case 3:
                        l.disp();
                        break;

                case 4:
```

```
                    l.disp();
                    cout<<"
 Enter the element to search:";
                    cin>>v;
                    cout<<"
 The position of the element "<< v<<"  is "<<l.seek(v);
                    getch();
                    break;

          case 5:
                    exit(1);

          default:
                    cout<<"
 The option is invalid...
";
                    return;
          }
          getch();
}while(ch!=5);
getch();
return;
}
```

Stack implementation as a class.

```
# include<iostream.h>
# include<process.h>
# include<conio.h>
# define SIZE 20

class stack
{
int a[SIZE];
int tos; // Top of Stack
public:
          stack();
          void push(int);
          int pop();
          int isempty();
          int isfull();
};
stack::stack()
{
tos=0; //Initialize Top of Stack
}

int stack::isempty()
{
return (tos==0?1:0);
}
int stack::isfull()
{
return (tos==SIZE?1:0);
}

void stack::push(int i)
{
```

```
if(!isfull())
{
a[tos]=i;
tos++;
}
else
{
 cerr<<"Stack overflow error !
Possible Data Loss !";
}
}
int stack::pop()
{
if(!isempty())
{
return(a[--tos]);
}
else
{
cerr<<"Stack is empty! What to pop...!";
}
return 0;
}

void main()
{
stack s;
int ch=1,num;
while(ch!=0)
{
	cout<<"Stack Operations Mani Menu
1.Push
2.Pop
3.IsEmpty
4.IsFull
0.Exit

";
	cin>>ch;
	switch(ch)
	{
	case 0:
		exit(1); //Normal Termination of Program
	case 1:
		cout<<"Enter the number to push";
		cin>>num;
		s.push(num);
		break;
	case 2:
		cout<<"Number popped from the stack is: "<<s.pop()<<endl;
		break;
	case 3:
		(s.isempty())?(cout<<"Stack is empty.
"):(cout<<"Stack is not empty.
");
		break;
	case 4:
```

```
                    (s.isfull())?(cout<<"Stack is full.
"):(cout<<"Stack is not full.
");
                    break;
            default:
                    cout<<"Illegal Option.
Please try again
";
            }
}//end of while
getch();
}
```

Sorted Doubly Linked List with Insertion and Deletion

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

class Dllist
{
  private:
     typedef struct Node
     {
       string name;
       Node* next;
       Node* prev;
     };
     Node* head;
     Node* last;
  public:
     Dllist()
     {
      head = NULL;
      last = NULL;
     }
     bool empty() const { return head==NULL; }
     friend ostream& operator<<(ostream& ,const Dllist& );
     void Insert(const string& );
     void Remove(const string& );
};

void Dllist::Insert(const string& s)
{
   // Insertion into an Empty List.
   if(empty())
   {
     Node* temp = new Node;
     head = temp;
     last = temp;
     temp->prev = NULL;
     temp->next = NULL;
     temp->name = s;
   }
   else
   {
     Node* curr;
```

```
      curr = head;
      while( s>curr->name && curr->next != last->next) curr = curr->next;

      if(curr == head)
      {
        Node* temp = new Node;
        temp->name = s;
        temp->prev = curr;
        temp->next = NULL;
        head->next = temp;
        last = temp;
      //  cout<<" Inserted "<<s<<" After "<<curr->name<<endl;
      }
      else
      {
      if(curr == last && s>last->name)
      {
        last->next = new Node;
        (last->next)->prev = last;
        last = last->next;
        last->next = NULL;
        last->name = s;
      //  cout<<" Added "<<s<<" at the end "<<endl;
      }
      else
      {
        Node* temp = new Node;
        temp->name = s;
        temp->next = curr;
        (curr->prev)->next = temp;
        temp->prev = curr->prev;
        curr->prev = temp;
      //  cout<<" Inserted "<<s<<" Before "<<curr->name<<endl;
      }
      }
    }
}

ostream& operator<<(ostream& ostr, const Dllist& dl )
{
   if(dl.empty()) ostr<<" The list is empty. "<<endl;
   else
   {
      Dllist::Node* curr;
      for(curr = dl.head; curr != dl.last->next; curr=curr->next)
        ostr<<curr->name<<" ";
      ostr<<endl;
      ostr<<endl;
      return ostr;
   }
}

void Dllist::Remove(const string& s)
{
   bool found = false;
   if(empty())
   {
```

```
    cout<<" This is an empty list! "<<endl;
    return;
   }
   else
   {
    Node* curr;
    for(curr = head; curr != last->next; curr = curr->next)
    {
       if(curr->name == s)
       {
         found = true;
         break;
       }
    }
    if(found == false)
    {
     cout<<" The list does not contain specified Node"<<endl;
     return;
    }
    else
    {
      // Curr points to the node to be removed.
      if (curr == head && found)
      {
        if(curr->next != NULL)
        {
         head = curr->next;
         delete curr;
         return;
        }
        else
        {
         delete curr;
         head = NULL;
         last = NULL;
         return;
        }
      }
     if (curr == last && found)
     {
      last = curr->prev;
      delete curr;
      return;
     }
    (curr->prev)->next = curr->next;
    (curr->next)->prev = curr->prev;
     delete curr;
   }
  }
}

int main()
{
   Dllist d1;
   int ch;
   string temp;
   while(1)
```

```
{
  cout<<endl;
  cout<<" Doubly Linked List Operations "<<endl;
  cout<<" ----------------------------"<<endl;
  cout<<" 1. Insertion "<<endl;
  cout<<" 2. Deletion "<<endl;
  cout<<" 3. Display "<<endl;
  cout<<" 4. Exit "<<endl;
  cout<<" Enter your choice : ";
  cin>>ch;
  switch(ch)
  {
    case 1: cout<<" Enter Name to be inserted : ";
          cin>>temp;
          d1.Insert(temp);
          break;
    case 2: cout<<" Enter Name to be deleted : ";
          cin>>temp;
          d1.Remove(temp);
          break;
    case 3: cout<<" The List contains : ";
          cout<<d1;
          break;
    case 4: system("pause");
          return 0;
          break;
  }
}
```

Program for Circular Queue Implementation using Arrays

Code :

```
/* Circular Queues */
#include<iostream.h>
#include<conio.h>

const int MAX = 5;
class cqueue
{
        int a[MAX],front,rear;
      public :
        cqueue()
        {
                front=rear=-1;
        }
        void insert(int );
        int deletion();
        void display();
};
```

```
void cqueue :: insert(int val)
{
        if((front==0 && rear==MAX-1) || (rear+1==front))
                cout<<" Circular Queue is Full
";
        else
        {
         if(rear==MAX-1)
                rear=0;
         else
                rear++;
         a[rear]=val;
        }
        if(front==-1)
         front=0;
}
int cqueue :: deletion()
{
        int k;
        if(front==-1)
                cout<<"Circular Queue is Empty
";
        else
        {
                k=a[front];
                if(front==rear)
                  front=rear=-1;
                else
                {
                  if(front==MAX-1)
                        front=0;
                  else
                        front++;
                }
        }
        return k;
}
void cqueue :: display()
{
         int i;
         if(front==-1)
                cout<<"Circular Queue is Empty
";
         else
         {
                if(rear < front)
                {
                        for(i=front;i<=MAX-1;i++)
                          cout<<a[i]<<"   ";
                        for(i=0;i<=rear;i++)
                          cout<<a[i]<<"   ";
                }
                else
                {
                        for(i=front;i<=rear;i++)
                          cout<<a[i]<<"   ";
                        cout<<endl;
```

```
                        }
                    }
            }

            void main()
            {
                    cqueue c1;

                    int ch,val;
                    char op;
                    do
                    {
                     clrscr();
                     cout<<"-----------Menu-------------
";
                     cout<<"1.Insertion
2.Deletion
3.Display
4.Exit
";
                     cout<<"Enter Your Choice <1..4> ?";
                     cin>>ch;
                     switch(ch)
                     {
                             case 1 : cout<<"Enter Element to Insert ?";
                                              cin>>val;
                                              c1.insert(val);
                                              break;
                             case 2 : val=c1.deletion();
                                              cout<<"Deleted Element :"<<val<<endl;
                                              break;
                             case 3 : c1.display();
                                              break;
                            }
                            cout<<"Do you want to continue<Y/N> ?";
                            cin>>op;
                     }while(op=='Y' || op=='y');
                     getch();
                    }
```

Stack implementation as a class.

```
# include<iostream.h>
# include<process.h>
# include<conio.h>
# define SIZE 20

class stack
{
int a[SIZE];
int tos; // Top of Stack
public:
        stack();
        void push(int);
        int pop();
        int isempty();
        int isfull();
```

```
};
stack::stack()
{
tos=0; //Initialize Top of Stack
}

int stack::isempty()
{
return (tos==0?1:0);
}
int stack::isfull()
{
return (tos==SIZE?1:0);
}

void stack::push(int i)
{
if(!isfull())
{
a[tos]=i;
tos++;
}
else
{
 cerr<<"Stack overflow error !
Possible Data Loss !";
}
}
int stack::pop()
{
if(!isempty())
{
return(a[--tos]);
}
else
{
cerr<<"Stack is empty! What to pop...!";
}
return 0;
}

void main()
{
stack s;
int ch=1,num;
while(ch!=0)
{
        cout<<"Stack Operations Mani Menu
1.Push
2.Pop
3.IsEmpty
4.IsFull
0.Exit

";
        cin>>ch;
        switch(ch)
```

```
        {
        case 0:
                exit(1); //Normal Termination of Program
        case 1:
                cout<<"Enter the number to push";
                cin>>num;
                s.push(num);
                break;
        case 2:
                cout<<"Number popped from the stack is: "<<s.pop()<<endl;
                break;
        case 3:
                (s.isempty())?(cout<<"Stack is empty.
"):(cout<<"Stack is not empty.
");
                break;
        case 4:
                (s.isfull())?(cout<<"Stack is full.
"):(cout<<"Stack is not full.
");
                break;
        default:
                cout<<"Illegal Option.
Please try again
";
        }
}//end of while
getch();
}
```

Implementing Queue as a Class

```
# include<iostream.h>
# include<conio.h>
# define SIZE 20

class queue
{
        int a[SIZE];
        int front;
        int rear;
public:
        queue();
        ~queue();
        void insert(int i);
        int remove();
        int isempty();
        int isfull();
};

queue::queue()
{
front=0;
rear=0;
}
queue::~queue()
{
```

```
delete []a;
}
void queue::insert(int i)
{
if(isfull())
{
        cout<<"

******
Queue is FULL !!!
No insertion allowed further.
******
";
        return;
}
a[rear] = i;
rear++;
}
int queue::remove()
{
if(isempty())
{
        cout<<"

******
Queue Empty !!!
Value returned will be garbage.
******
";
        return (-9999);
}

return(a[front++]);
}
int queue::isempty()
{
if(front == rear)
        return 1;
else
        return 0;
}
int queue::isfull()
{
if(rear == SIZE)
        return 1;
else
        return 0;
}

void main()
{
clrscr();
queue q;
q.insert(1);
q.insert(2);
cout<<"
"<<q.remove();
```

```
cout<<"
"<<q.remove();
cout<<"
"<<q.remove();
getch();
}
```