**1024: 图的邻接矩阵**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 58  Solved: 37  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1024)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1024)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1024)]

**Description**

**利用邻接矩阵存储结构，完成深度优先遍历和广度优先遍历算法。已知部分代码如下（勿改动），只需完成深度遍历DFSTraverse和广度遍历BFSTraverse即可。**

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

//循环队列类  
const int QueueSize=100;  //循环队列的最大长度  
template <class T>        //定义模板类CirQueue  
class CirQueue  
{  
public:  
    CirQueue( );                 //构造函数，置空队  
    ~ CirQueue( );               //析构函数  
    void EnQueue(T x);           //将元素x入队  
    T DeQueue( );                //将队头元素出队  
    T GetQueue( );               //取队头元素（并不删除）  
    bool Empty( );               //判断队列是否为空  
 bool Full();  
private:  
    T data[QueueSize];  //存放队列元素的数组  
    int front, rear;    //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置  
};

//功    能：初始化空队列  
 template <class T>  
CirQueue<T>::CirQueue( )   
{  
 front=rear=0;  
}

// 功    能：销毁队列  
template <class T>  
CirQueue<T>::~CirQueue( )  
{  
}

//功    能：元素x入队  
template <class T>   
void CirQueue<T>::EnQueue(T x)  
{  
    if (Full()) throw "Overflow";  
    rear=(rear+1) % QueueSize;   //队尾指针在循环意义下加1  
    data[rear]=x;                //在队尾处插入元素  
}

//功    能：队头元素出队，返回值为出队元素  
template <class T>   
T CirQueue<T>::DeQueue( )  
{  
    if (Empty()) throw "Downflow";   
    front=(front+1) % QueueSize;    //队头指针在循环意义下加1  
    return data[front];             //读取并返回出队前的队头元素，注意队头指针  
}  
                                   //指向队头元素的前一个位置  
// 功    能：获取队头元素  
template <class T>  
T CirQueue<T>::GetQueue( )  
{     
    int i;  
    if (Empty()) throw "Downflow";   
    i=(front+1) % QueueSize;  //注意不要给队头指针赋值  
    return data[i];  
}

// 功    能：判断队列是否为空,若空返回true，否则返回false  
template <class T>   
bool CirQueue<T>::Empty( )   
{  
    return front==rear;   
}  
//功   能：判断队列是否满，若满返回true，否则false  
template <class T>  
bool CirQueue<T>::Full()  
{  
 return (rear+1)%QueueSize==front;  
}

//Class MGraph  
const int MaxSize=20;                     //图中最多顶点个数

int visited[MaxSize];                    //访问标志数组（0表示未访问，1表示已访问）

template <class T>  
class MGraph  
{  
public:  
    MGraph(T a[],int n,int e);   //构造函数，初始化具有n个顶点,e条边的图  
    void DispMGraph();           //输出图中顶点值和矩阵值                 
      
 void DFSTraverse(int v);             从v顶点出发深度优先遍历图（一个连通子图）

 void BFSTraverse(int v);              从v顶点出发广度优先遍历图（一个连通子图）

private:  
    T  vertex[MaxSize];             //存放图中顶点信息的数组  
    int arc[MaxSize][MaxSize];      //存放图中顶点关系（边）的数组  
    int vertexNum,arcNum;           //图的顶点数和边数  
 };

//构造一个邻接矩阵存储的无向图  
template <class T>  
MGraph<T>::MGraph(T a[], int n, int e)   
{  
 vertexNum=n;                 //顶点数  
 arcNum=e;                    //边数  
 int i,j,k;  
 for (i=0; i<vertexNum; i++)   
  vertex[i]=a[i];          //顶点值  
 for (i=0; i<vertexNum; i++)    //初始化邻接矩阵值为0（顶点间无边）  
  for (j=0; j<vertexNum; j++)  
   arc[i][j]=0;               
 for (k=0; k<arcNum; k++)   //依次输入e条边  
 {  
  cin>>i>>j;             //输入各条边依附的两个顶点的序号  
  arc[i][j]=1;           //置有边标志  
  arc[j][i]=1;      
 }  
}

//输出图中所有顶点信息，和边信息  
template <class T>  
void MGraph<T>::DispMGraph( )               
{                                   
 int i,j;  
    cout<<"  ";  
    for(i=0;i<vertexNum;i++)               
  cout<<vertex[i]<<" ";//输出图中所有的顶点  
    cout<<endl;  
 for(i=0;i<vertexNum;i++)  
 {  
  cout<<vertex[i]<<" ";  
  for(j=0;j<vertexNum;j++)  
   cout<<arc[i][j]<<" ";  //输出边值（0/1）  
  cout<<endl;  
 }  
}  
//在此处实现深度优先遍历算法

//在此处实现广度优先遍历算法  
  
int main()  
{  
 string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息  
 int i,n,e;  
 cin>>n>>e;  //输入顶点个数和边个数  
    MGraph<string> G(a,n,e);  
 G.DispMGraph();  
 for(i=0;i<n;i++)  
  visited[i]=0;  
 cout<<"DFS:";  
 G.DFSTraverse(0);  
 cout<<endl;  
    for(i=0;i<n;i++)  
  visited[i]=0;  
 cout<<"BFS:";  
 G.BFSTraverse(0);  
 return 0;  
}

**Input**

第一行为顶点个数和边的条数：6 7  
第二行为各条边所对应的顶点序号：0 1 0 3 1 2 1 5 2 4 3 5 4 5

**Output**

**Sample Input**

6 7

0 1 0 3 1 2 1 5 2 4 3 5 4 5

**Sample Output**

A B C D E F

A 0 1 0 1 0 0

B 1 0 1 0 0 1

C 0 1 0 0 1 0

D 1 0 0 0 0 1

E 0 0 1 0 0 1

F 0 1 0 1 1 0

DFS:A B C E F D

BFS:A B D C F E

#include <iostream>

#include <string>

using namespace std;

//循环队列类

const int QueueSize=100;  //循环队列的最大长度

template <class T>        //定义模板类CirQueue

class CirQueue

{

public:

    CirQueue( );                 //构造函数，置空队

    ~ CirQueue( );               //析构函数

    void EnQueue(T x);           //将元素x入队

    T DeQueue( );                //将队头元素出队

    T GetQueue( );               //取队头元素（并不删除）

    bool Empty( );               //判断队列是否为空

 bool Full();

private:

    T data[QueueSize];  //存放队列元素的数组

    int front, rear;    //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置

};

//功    能：初始化空队列

 template <class T>

CirQueue<T>::CirQueue( )

{

 front=rear=0;

}

// 功    能：销毁队列

template <class T>

CirQueue<T>::~CirQueue( )

{

}

//功    能：元素x入队

template <class T>

void CirQueue<T>::EnQueue(T x)

{

    if (Full()) throw "Overflow";

    rear=(rear+1) % QueueSize;   //队尾指针在循环意义下加1

    data[rear]=x;                //在队尾处插入元素

}

//功    能：队头元素出队，返回值为出队元素

template <class T>

T CirQueue<T>::DeQueue( )

{

    if (Empty()) throw "Downflow";

    front=(front+1) % QueueSize;    //队头指针在循环意义下加1

    return data[front];             //读取并返回出队前的队头元素，注意队头指针

}

                                   //指向队头元素的前一个位置

// 功    能：获取队头元素

template <class T>

T CirQueue<T>::GetQueue( )

{

    int i;

    if (Empty()) throw "Downflow";

    i=(front+1) % QueueSize;  //注意不要给队头指针赋值

    return data[i];

}

// 功    能：判断队列是否为空,若空返回true，否则返回false

template <class T>

bool CirQueue<T>::Empty( )

{

    return front==rear;

}

//功   能：判断队列是否满，若满返回true，否则false

template <class T>

bool CirQueue<T>::Full()

{

 return (rear+1)%QueueSize==front;

}

//Class MGraph

const int MaxSize=20;                     //图中最多顶点个数

int visited[MaxSize];                    //访问标志数组（0表示未访问，1表示已访问）

template <class T>

class MGraph

{

public:

    MGraph(T a[],int n,int e);   //构造函数，初始化具有n个顶点,e条边的图

    void DispMGraph();           //输出图中顶点值和矩阵值

 void DFSTraverse(int v);           //  从v顶点出发深度优先遍历图（一个连通子图）

 void BFSTraverse(int v);            //  从v顶点出发广度优先遍历图（一个连通子图）

private:

    T  vertex[MaxSize];             //存放图中顶点信息的数组

    int arc[MaxSize][MaxSize];      //存放图中顶点关系（边）的数组

    int vertexNum,arcNum;           //图的顶点数和边数

 };

//构造一个邻接矩阵存储的无向图

template <class T>

MGraph<T>::MGraph(T a[], int n, int e)

{

 vertexNum=n;                 //顶点数

 arcNum=e;                    //边数

 int i,j,k;

 for (i=0; i<vertexNum; i++)

  vertex[i]=a[i];          //顶点值

 for (i=0; i<vertexNum; i++)    //初始化邻接矩阵值为0（顶点间无边）

  for (j=0; j<vertexNum; j++)

   arc[i][j]=0;

 for (k=0; k<arcNum; k++)   //依次输入e条边

 {

  cin>>i>>j;             //输入各条边依附的两个顶点的序号

  arc[i][j]=1;           //置有边标志

  arc[j][i]=1;

 }

}

//输出图中所有顶点信息，和边信息

template <class T>

void MGraph<T>::DispMGraph( )

{

 int i,j;

    cout<<"  ";

    for(i=0;i<vertexNum;i++)

  cout<<vertex[i]<<" ";//输出图中所有的顶点

    cout<<endl;

 for(i=0;i<vertexNum;i++)

 {

  cout<<vertex[i]<<" ";

  for(j=0;j<vertexNum;j++)

   cout<<arc[i][j]<<" ";  //输出边值（0/1）

  cout<<endl;

 }

}

//在此处实现深度优先遍历算法

template <class T>

void MGraph<T>::DFSTraverse(int v)

{

        int i;

        cout<<vertex[v]<<" ";

        visited[v]=1;

        for( i = 0 ; i < vertexNum ; i++)

            if( arc[v][i]==1 && visited[i]==0 )  DFSTraverse(i);

}

//在此处实现广度优先遍历算法

template <class T>

void MGraph<T>::BFSTraverse(int v)

{

        CirQueue<int> C;

        C.EnQueue( v );

        visited[0]=1;

        int i;

        while( !C.Empty() )

        {

            for( i = 0 ; i < vertexNum ; i++ )

            if( arc[ C.GetQueue() ][i] == 1 && visited[i] == 0 )

            {

                C.EnQueue( i );

                visited [ i ] = 1;

            }

            cout<< vertex [ C.DeQueue( ) ] <<" ";

        }

//         while(1);

        cout<<endl;

}

int main()

{

 string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息

 int i,n,e;

 cin>>n>>e;  //输入顶点个数和边个数

    MGraph<string> G(a,n,e);

 G.DispMGraph();

 for(i=0;i<n;i++)

  visited[i]=0;

 cout<<"DFS:";

 G.DFSTraverse(0);

 cout<<endl;

    for(i=0;i<n;i++)

  visited[i]=0;

 cout<<"BFS:";

 G.BFSTraverse(0);

 return 0;

}

**1025: 邻接表（1）**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 50  Solved: 36  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1025)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1025)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1025)]

**Description**

**已知有向图采用邻接表存储，部分代码如下，请补充完成计算各个顶点的入度，出度算法。**

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

const int MaxSize=20;     // 顶点个数的最大值  
int visited[MaxSize];     //访问标志数组（0表示未访问，1表示已访问）

//定义边表结点  
struct ArcNode                   
{  
 int adjvex;                    //邻接点的序号  
 ArcNode \*next;                 //指向下一个边结点的指针  
};

 //定义顶点表结点  
template <class T>  
struct VertexNode                
{  
 T vertex;                 //顶点的名称  
 ArcNode \*firstedge;        //指向第一个边表结点的头指针  
};

//邻接表类  
template <class T>  
class ALGraph  
{  
public:  
   ALGraph(T a[ ], int n, int e);   //构造函数，初始化一个有n个顶点e条边的图  
   ~ALGraph();                      //析构函数，释放邻接表中各边表结点的存储空间  
   void DispALGraph();              //输出邻接表  
   void CountInD(int ind[]);     //计算各个顶点的入度,存储在ind中  
   void CountOutD(int outd[]);   //计算各个顶点的出度,存储在outd中  
private:  
   VertexNode<T> adjlist[MaxSize];  //存放顶点表的数组  
   int vertexNum, arcNum;           //图的顶点数和边数  
};

/\*  
 \*前置条件：图不存在  
 \*输    入：无   
 \*功    能：图的初始化  
 \*输    出：无  
 \*后置条件：得到一个无向图  
 \*/  
template <class T>  
ALGraph<T>::ALGraph(T a[ ], int n, int e)  
{  
    arcNum = e;                             //边数  
 vertexNum=n;                            //顶点数  
    int i,j;  
 for (i=0; i<vertexNum; i++)  
 {          
    adjlist[i].vertex = a[i];  
    adjlist[i].firstedge = NULL;       
 }

 for (int k=0; k<arcNum; k++)    //依次输入每一条边，并在相应边表中插入结点  
    {   
  cin>>i>>j;                         //输入边所依附的两个顶点的序号  
        ArcNode \*s=new ArcNode;   
  s->adjvex=j;  //生成一个边表结点s  
     s->next=adjlist[i].firstedge;      //将结点s插入到i号表的头结点之后    
        adjlist[i].firstedge=s;  
 }  
}  
/\*   前置条件：图已存在  
 \*   输    入：无   
 \*   功    能：销毁图  
 \*   输    出：无  
 \*   后置条件：释放图所占用的存储空间  
 \*/  
template <class T>  
ALGraph<T>::~ALGraph( )  
{  
  for (int i=0; i<vertexNum; i++)  
  {  
    ArcNode \* p=adjlist[i].firstedge;  
 while (p!=NULL)                                              //循环删除  
 {  
  adjlist[i].firstedge=p->next;  
  delete p;                                                 //释放结点空间  
  p=adjlist[i].firstedge;  
 }  
  }  
}

/\*     
 \*前置条件：图已存在  
 \*输    入：无  
 \*功    能：输出图中所有顶点及边的数据信息  
 \*输    出：图中所有顶点及边的数据信息  
 \*后置条件：图保持不变  
 \*/  
template <class T>  
void ALGraph<T>::DispALGraph( )                       
{                                    
 int i;  
 ArcNode \*p;  
 cout<<"图的邻接表:\n";  
    for(i=0;i<vertexNum;i++){        
  cout<<i<<" "<<adjlist[i].vertex<<" "; //输出图中顶点的序号i及值  
  for(p=adjlist[i].firstedge;p;p=p->next)  
   cout<<p->adjvex<<" ";  //输出i号顶点的邻接点的序号   
  cout<<endl;  
 }  
}

//计算各个顶点的入度  
template <class T>  
void ALGraph<T>::CountInD(int ind[])  
{  
}  
//计算各个顶点的出度  
template <class T>  
void ALGraph<T>::CountOutD(int outd[])  
{  
 }  
int main()  
{  
 string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息  
 int i,n,e;  
 cin>>n>>e;  //输入顶点个数和边个数  
    ALGraph<string> G(a,n,e);  
 G.DispALGraph();  
 int ind[MaxSize];  
 cout<<"Indegree:";  
 G.CountInD(ind);   
    for(i=0;i<n;i++)  
  cout<<ind[i]<<" ";// 输出各个顶点的入度  
 cout<<endl;  
    int outd[MaxSize];  
 cout<<"OutDegree:";  
 G.CountOutD(outd);  
    for(i=0;i<n;i++)  
  cout<<outd[i]<<" "; // 输出各个顶点的出度  
 return 0;  
}

**Input**

**Output**

**Sample Input**

6 7

0 1 0 3 1 2 1 5 2 4 3 5 4 5

**Sample Output**

图的邻接表:

0 A 3 1

1 B 5 2

2 C 4

3 D 5

4 E 5

5 F

Indegree:0 1 1 1 1 3

OutDegree:2 2 1 1 1 0

#include <iostream>

#include <string>

using namespace std;

const int MaxSize=20; // 顶点个数的最大值

int visited[MaxSize]; //访问标志数组（0表示未访问，1表示已访问）

//定义边表结点

struct ArcNode

{

int adjvex; //邻接点的序号

ArcNode \*next; //指向下一个边结点的指针

};

//定义顶点表结点

template <class T>

struct VertexNode

{

T vertex; //顶点的名称

ArcNode \*firstedge; //指向第一个边表结点的头指针

};

//邻接表类

template <class T>

class ALGraph

{

public:

ALGraph(T a[ ], int n, int e); //构造函数，初始化一个有n个顶点e条边的图

~ALGraph(); //析构函数，释放邻接表中各边表结点的存储空间

void DispALGraph(); //输出邻接表

void CountInD(int ind[]); //计算各个顶点的入度,存储在ind中

void CountOutD(int outd[]); //计算各个顶点的出度,存储在outd中

private:

VertexNode<T> adjlist[MaxSize]; //存放顶点表的数组

int vertexNum, arcNum; //图的顶点数和边数

};

/\*

\*前置条件：图不存在

\*输 入：无

\*功 能：图的初始化

\*输 出：无

\*后置条件：得到一个无向图

\*/

template <class T>

ALGraph<T>::ALGraph(T a[ ], int n, int e)

{

arcNum = e; //边数

vertexNum=n; //顶点数

int i,j;

for (i=0; i<vertexNum; i++)

{

adjlist[i].vertex = a[i];

adjlist[i].firstedge = NULL;

}

for (int k=0; k<arcNum; k++) //依次输入每一条边，并在相应边表中插入结点

{

cin>>i>>j; //输入边所依附的两个顶点的序号

ArcNode \*s=new ArcNode;

s->adjvex=j; //生成一个边表结点s

s->next=adjlist[i].firstedge; //将结点s插入到i号表的头结点之后

adjlist[i].firstedge=s;

}

}

/\* 前置条件：图已存在

\* 输 入：无

\* 功 能：销毁图

\* 输 出：无

\* 后置条件：释放图所占用的存储空间

\*/

template <class T>

ALGraph<T>::~ALGraph( )

{

for (int i=0; i<vertexNum; i++)

{

ArcNode \* p=adjlist[i].firstedge;

while (p!=NULL) //循环删除

{

adjlist[i].firstedge=p->next;

delete p; //释放结点空间

p=adjlist[i].firstedge;

}

}

}

/\*

\*前置条件：图已存在

\*输 入：无

\*功 能：输出图中所有顶点及边的数据信息

\*输 出：图中所有顶点及边的数据信息

\*后置条件：图保持不变

\*/

template <class T>

void ALGraph<T>::DispALGraph( )

{

int i;

ArcNode \*p;

cout<<"图的邻接表:\n";

for(i=0;i<vertexNum;i++){

cout<<i<<" "<<adjlist[i].vertex<<" "; //输出图中顶点的序号i及值

for(p=adjlist[i].firstedge;p;p=p->next)

cout<<p->adjvex<<" "; //输出i号顶点的邻接点的序号

cout<<endl;

}

}

//计算各个顶点的入度

template <class T>

void ALGraph<T>::CountInD(int ind[])

{

int i,count;

for( i = 0 ; i < vertexNum ; i++ )

ind[i]=0;

for( i = 0 ; i < vertexNum ; i++ )

{

ArcNode \*p;

for( p = adjlist[i].firstedge ; p ; p = p->next )

ind[p->adjvex]++;

}

}

//计算各个顶点的出度

template <class T>

void ALGraph<T>::CountOutD(int outd[])

{

int i,count;

for( i = 0 ; i < vertexNum ; i++ )

{

ArcNode \*p;

count = 0 ;

for( p = adjlist[i].firstedge ; p ; p = p->next , count++) ;

outd[i]=count;

}

}

int main()

{

string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息

int i,n,e;

cin>>n>>e; //输入顶点个数和边个数

ALGraph<string> G(a,n,e);

G.DispALGraph();

int ind[MaxSize];

cout<<"Indegree:";

G.CountInD(ind);

for(i=0;i<n;i++)

cout<<ind[i]<<" ";// 输出各个顶点的入度

cout<<endl;

int outd[MaxSize];

cout<<"OutDegree:";

G.CountOutD(outd);

for(i=0;i<n;i++)

cout<<outd[i]<<" "; // 输出各个顶点的出度

return 0;

}

**1026: 邻接表（2）**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 47  Solved: 37  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1026)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1026)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1026)]

**Description**

**已知邻接表存储的无向图的部分代码如下，请补充完成深度优先、广度优先遍历算法。**

//graphmain.cpp  
#include <iostream>  
#include <string>  
using namespace std;

//循环队列类  
const int QueueSize=100;  //循环队列的最大长度  
template <class T>        //定义模板类CirQueue  
class CirQueue  
{  
public:  
    CirQueue( );                 //构造函数，置空队  
    ~ CirQueue( );               //析构函数  
    void EnQueue(T x);           //将元素x入队  
    T DeQueue( );                //将队头元素出队  
    T GetQueue( );               //取队头元素（并不删除）  
    bool Empty( );               //判断队列是否为空  
 bool Full();  
private:  
    T data[QueueSize];  //存放队列元素的数组  
    int front, rear;    //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置  
};

//功    能：初始化空队列  
 template <class T>  
CirQueue<T>::CirQueue( )   
{  
 front=rear=0;  
}

// 功    能：销毁队列  
template <class T>  
CirQueue<T>::~CirQueue( )  
{  
}

//功    能：元素x入队  
template <class T>   
void CirQueue<T>::EnQueue(T x)  
{  
    if (Full()) throw "Overflow";  
    rear=(rear+1) % QueueSize;   //队尾指针在循环意义下加1  
    data[rear]=x;                //在队尾处插入元素  
}

//功    能：队头元素出队，返回值为出队元素  
template <class T>   
T CirQueue<T>::DeQueue( )  
{  
    if (Empty()) throw "Downflow";   
    front=(front+1) % QueueSize;    //队头指针在循环意义下加1  
    return data[front];             //读取并返回出队前的队头元素，注意队头指针  
}  
                                   //指向队头元素的前一个位置  
// 功    能：获取队头元素  
template <class T>  
T CirQueue<T>::GetQueue( )  
{     
    int i;  
    if (Empty()) throw "Downflow";   
    i=(front+1) % QueueSize;  //注意不要给队头指针赋值  
    return data[i];  
}

// 功    能：判断队列是否为空,若空返回true，否则返回false  
template <class T>   
bool CirQueue<T>::Empty( )   
{  
    return front==rear;   
}  
//功   能：判断队列是否满，若满返回true，否则false  
template <class T>  
bool CirQueue<T>::Full()  
{  
 return (rear+1)%QueueSize==front;  
}

const int MaxSize=20;     // 顶点个数的最大值  
int visited[MaxSize];     //访问标志数组（0表示未访问，1表示已访问）

//定义边表结点  
struct ArcNode                   
{  
 int adjvex;                    //邻接点的序号  
 ArcNode \*next;                 //指向下一个边结点的指针  
};

 //定义顶点表结点  
template <class T>  
struct VertexNode                
{  
 T vertex;                 //顶点的名称  
 ArcNode \*firstedge;        //指向第一个边表结点的头指针  
};

//邻接表类  
template <class T>  
class ALGraph  
{  
public:  
   ALGraph(T a[ ], int n, int e);   //构造函数，初始化一个有n个顶点e条边的图  
   ~ALGraph();                      //析构函数，释放邻接表中各边表结点的存储空间  
   void DispALGraph();              //输出邻接表  
   int Count( );                    //计算连通分量个数，返回值为连通分量的个数（请大家自己测试）  
   void DFSTraverse(int v);         //深度优先遍历图  
   void BFSTraverse(int v);          //深度优先遍历图  
private:  
   VertexNode<T> adjlist[MaxSize];  //存放顶点表的数组  
   int vertexNum, arcNum;           //图的顶点数和边数  
};

/\*  
 \*前置条件：图不存在  
 \*输    入：无   
 \*功    能：图的初始化  
 \*输    出：无  
 \*后置条件：得到一个无向图  
 \*/  
template <class T>  
ALGraph<T>::ALGraph(T a[ ], int n, int e)  
{  
    arcNum = e;                             //边数  
 vertexNum=n;                            //顶点数  
    int i,j;  
 for (i=0; i<vertexNum; i++)  
 {          
    adjlist[i].vertex = a[i];  
    adjlist[i].firstedge = NULL;       
 }

 for (int k=0; k<arcNum; k++)    //依次输入每一条边，并在相应边表中插入结点  
    {   
  cin>>i>>j;                         //输入边所依附的两个顶点的序号  
        ArcNode \*s=new ArcNode;   
  s->adjvex=j;  //生成一个边表结点s  
     s->next=adjlist[i].firstedge;      //将结点s插入到i号表的头结点之后    
        adjlist[i].firstedge=s;  
        s=new ArcNode;   
  s->adjvex=i;  //生成一个边表结点s  
     s->next=adjlist[j].firstedge;      //将结点s插入到j号表的头结点之后    
        adjlist[j].firstedge=s;  
 }  
}  
/\*   前置条件：图已存在  
 \*   输    入：无   
 \*   功    能：销毁图  
 \*   输    出：无  
 \*   后置条件：释放图所占用的存储空间  
 \*/  
template <class T>  
ALGraph<T>::~ALGraph( )  
{  
  for (int i=0; i<vertexNum; i++)  
  {  
    ArcNode \* p=adjlist[i].firstedge;  
 while (p!=NULL)                                              //循环删除  
 {  
  adjlist[i].firstedge=p->next;  
  delete p;                                                 //释放结点空间  
  p=adjlist[i].firstedge;  
 }  
  }  
}

/\*     
 \*前置条件：图已存在  
 \*输    入：无  
 \*功    能：输出图中所有顶点及边的数据信息  
 \*输    出：图中所有顶点及边的数据信息  
 \*后置条件：图保持不变  
 \*/  
template <class T>  
void ALGraph<T>::DispALGraph( )                       
{                                    
 int i;  
 ArcNode \*p;  
 cout<<"图的邻接表:\n";  
    for(i=0;i<vertexNum;i++){        
  cout<<i<<" "<<adjlist[i].vertex<<" "; //输出图中顶点的序号i及值  
  for(p=adjlist[i].firstedge;p;p=p->next)  
   cout<<p->adjvex<<" ";  //输出i号顶点的邻接点的序号   
  cout<<endl;  
 }  
}

template <class T>  
int ALGraph<T>::Count( )                       
{                                    
 int i,n=0;

 for(i=0;i<vertexNum;i++) visited[i]=0;  
 for(i=0;i<vertexNum;i++){         
  if(!visited[i]){  
   n++;  
   BFSTraverse(i);   //利用深度优先或广度优先遍历算法均可  
  }    
 }  
 return n;  
}

//在下面完成深度优先、广度优先遍历算法

int main()  
{  
 string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息  
 int i,n,e;  
 cin>>n>>e;  //输入顶点个数和边个数  
    ALGraph<string> G(a,n,e);  
 G.DispALGraph();  
 for(i=0;i<n;i++)  
  visited[i]=0;  
 cout<<"DFS:";  
 G.DFSTraverse(0);  
 cout<<endl;  
    for(i=0;i<n;i++)  
  visited[i]=0;  
 cout<<"BFS:";  
 G.BFSTraverse(0);

//  cout<<"\n连通分量个数:";  
// cout<<G.Count()<<endl;  
 return 0;  
}

**Input**

第一行为顶点个数和边的条数：6 7  
第二行为各条边所对应的顶点序号：0 1 0 3 1 2 1 5 2 4 3 5 4 5

**Output**

因为构造函数中各条边所对应的边表结点采用头插法插入到表头结点之后，因此输入0 1 0 3，0号链表后为3 1序列，其它亦如此。

图的邻接表:  
0 A 3 1   
1 B 5 2 0   
2 C 4 1   
3 D 5 0   
4 E 5 2   
5 F 4 3 1   
DFS:A D F E C B   
BFS:A D B F C E

**Sample Input**

6 7

0 1 0 3 1 2 1 5 2 4 3 5 4 5

**Sample Output**

图的邻接表:

0 A 3 1

1 B 5 2 0

2 C 4 1

3 D 5 0

4 E 5 2

5 F 4 3 1

DFS:A D F E C B

BFS:A D B F C E

#include <iostream>

#include <string>

using namespace std;

//循环队列类

const int QueueSize=100; //循环队列的最大长度

template <class T> //定义模板类CirQueue

class CirQueue

{

public:

CirQueue( ); //构造函数，置空队

~ CirQueue( ); //析构函数

void EnQueue(T x); //将元素x入队

T DeQueue( ); //将队头元素出队

T GetQueue( ); //取队头元素（并不删除）

bool Empty( ); //判断队列是否为空

bool Full();

private:

T data[QueueSize]; //存放队列元素的数组

int front, rear; //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置

};

//功 能：初始化空队列

template <class T>

CirQueue<T>::CirQueue( )

{

front=rear=0;

}

// 功 能：销毁队列

template <class T>

CirQueue<T>::~CirQueue( )

{

}

//功 能：元素x入队

template <class T>

void CirQueue<T>::EnQueue(T x)

{

if (Full()) throw "Overflow";

rear=(rear+1) % QueueSize; //队尾指针在循环意义下加1

data[rear]=x; //在队尾处插入元素

}

//功 能：队头元素出队，返回值为出队元素

template <class T>

T CirQueue<T>::DeQueue( )

{

if (Empty()) throw "Downflow";

front=(front+1) % QueueSize; //队头指针在循环意义下加1

return data[front]; //读取并返回出队前的队头元素，注意队头指针

}

//指向队头元素的前一个位置

// 功 能：获取队头元素

template <class T>

T CirQueue<T>::GetQueue( )

{

int i;

if (Empty()) throw "Downflow";

i=(front+1) % QueueSize; //注意不要给队头指针赋值

return data[i];

}

// 功 能：判断队列是否为空,若空返回true，否则返回false

template <class T>

bool CirQueue<T>::Empty( )

{

return front==rear;

}

//功 能：判断队列是否满，若满返回true，否则false

template <class T>

bool CirQueue<T>::Full()

{

return (rear+1)%QueueSize==front;

}

const int MaxSize=20; // 顶点个数的最大值

int visited[MaxSize]; //访问标志数组（0表示未访问，1表示已访问）

//定义边表结点

struct ArcNode

{

int adjvex; //邻接点的序号

ArcNode \*next; //指向下一个边结点的指针

};

//定义顶点表结点

template <class T>

struct VertexNode

{

T vertex; //顶点的名称

ArcNode \*firstedge; //指向第一个边表结点的头指针

};

//邻接表类

template <class T>

class ALGraph

{

public:

ALGraph(T a[ ], int n, int e); //构造函数，初始化一个有n个顶点e条边的图

~ALGraph(); //析构函数，释放邻接表中各边表结点的存储空间

void DispALGraph(); //输出邻接表

int Count( ); //计算连通分量个数，返回值为连通分量的个数（请大家自己测试）

void DFSTraverse(int v); //深度优先遍历图

void BFSTraverse(int v); //深度优先遍历图

private:

VertexNode<T> adjlist[MaxSize]; //存放顶点表的数组

int vertexNum, arcNum; //图的顶点数和边数

};

/\*

\*前置条件：图不存在

\*输 入：无

\*功 能：图的初始化

\*输 出：无

\*后置条件：得到一个无向图

\*/

template <class T>

ALGraph<T>::ALGraph(T a[ ], int n, int e)

{

arcNum = e; //边数

vertexNum=n; //顶点数

int i,j;

for (i=0; i<vertexNum; i++)

{

adjlist[i].vertex = a[i];

adjlist[i].firstedge = NULL;

}

for (int k=0; k<arcNum; k++) //依次输入每一条边，并在相应边表中插入结点

{

cin>>i>>j; //输入边所依附的两个顶点的序号

ArcNode \*s=new ArcNode;

s->adjvex=j; //生成一个边表结点s

s->next=adjlist[i].firstedge; //将结点s插入到i号表的头结点之后

adjlist[i].firstedge=s;

s=new ArcNode;

s->adjvex=i; //生成一个边表结点s

s->next=adjlist[j].firstedge; //将结点s插入到j号表的头结点之后

adjlist[j].firstedge=s;

}

}

/\* 前置条件：图已存在

\* 输 入：无

\* 功 能：销毁图

\* 输 出：无

\* 后置条件：释放图所占用的存储空间

\*/

template <class T>

ALGraph<T>::~ALGraph( )

{

for (int i=0; i<vertexNum; i++)

{

ArcNode \* p=adjlist[i].firstedge;

while (p!=NULL) //循环删除

{

adjlist[i].firstedge=p->next;

delete p; //释放结点空间

p=adjlist[i].firstedge;

}

}

}

/\*

\*前置条件：图已存在

\*输 入：无

\*功 能：输出图中所有顶点及边的数据信息

\*输 出：图中所有顶点及边的数据信息

\*后置条件：图保持不变

\*/

template <class T>

void ALGraph<T>::DispALGraph( )

{

int i;

ArcNode \*p;

cout<<"图的邻接表:\n";

for(i=0;i<vertexNum;i++){

cout<<i<<" "<<adjlist[i].vertex<<" "; //输出图中顶点的序号i及值

for(p=adjlist[i].firstedge;p;p=p->next)

cout<<p->adjvex<<" "; //输出i号顶点的邻接点的序号

cout<<endl;

}

}

template <class T>

int ALGraph<T>::Count( )

{

int i,n=0;

for(i=0;i<vertexNum;i++) visited[i]=0;

for(i=0;i<vertexNum;i++){

if(!visited[i]){

n++;

BFSTraverse(i); //利用深度优先或广度优先遍历算法均可

}

}

return n;

}

//在下面完成深度优先、广度优先遍历算法

template <class T>

void ALGraph<T>::DFSTraverse(int v) //深度优先遍历图

{

ArcNode \*p;

cout<<adjlist[v].vertex<<" ";

visited[v] = 1;

for( p = adjlist[v].firstedge ; p ; p = p->next )

if( visited[p->adjvex] == 0 )

{

visited[p->adjvex] = 1;

DFSTraverse( p->adjvex );

}

}

template <class T>

void ALGraph<T>::BFSTraverse(int v) //广度优先遍历图

{

ArcNode \*p;

CirQueue<int> C;

C.EnQueue( v );

visited[v] = 1;

while( !C.Empty() )

{

for( p = adjlist[C.GetQueue( )].firstedge ; p ; p = p->next )

if( visited[p->adjvex] == 0 )

{

C.EnQueue( p->adjvex );

visited[ p -> adjvex ] = 1;

}

cout<<adjlist[C.DeQueue( )].vertex<<" ";

}

}

int main()

{

string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息

int i,n,e;

cin>>n>>e; //输入顶点个数和边个数

ALGraph<string> G(a,n,e);

G.DispALGraph();

for(i=0;i<n;i++)

visited[i]=0;

cout<<"DFS:";

G.DFSTraverse(0);

cout<<endl;

for(i=0;i<n;i++)

visited[i]=0;

cout<<"BFS:";

G.BFSTraverse(0);

// cout<<"\n连通分量个数:";

// cout<<G.Count()<<endl;

return 0;

}

**1027: 拓扑排序**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 29  Solved: 19  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1027)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1027)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1027)]

**Description**

**利用邻接表存储结构实现一个有向图的数据输出，计算各顶点的入度，再输出拓扑排序序列，若拓扑排序失败，输出Failure。已知部分代码如下，勿改动。请完成CountInD，TopSort函数。**

//graphmain.cpp  
#include <iostream>  
#include <string>  
using namespace std;  
//栈类  
const int StackSize=100;  //10只是示例性的数据，可以根据实际问题具体定义  
template <class T>       //定义模板类SeqStack  
class SeqStack  
{  
public:  
    SeqStack( ) ;            //构造函数，栈的初始化  
 ~SeqStack( );            //析构函数  
    void Push(T x);          //将元素x入栈  
    T Pop( );                //将栈顶元素弹出  
    T GetTop( );          //取栈顶元素（并不删除）  
 bool Empty( );           //判断栈是否为空  
 bool Full();             //判断栈是否为满  
private:  
    T data[StackSize];      //存放栈元素的数组  
    int top;                //栈顶指针，指示栈顶元素在数组中的下标  
};

//初始化空栈  
template <class T>  
SeqStack<T>::SeqStack( )  
{  
 top=-1;  
}

//功    能：销毁栈  
template <class T>  
SeqStack<T>::~SeqStack( )  
{  
}

//功    能：元素x压栈  
template <class T>   
void SeqStack<T>::Push(T x)  
{  
    if (Full()) throw "Overflow";  
    top++;  
    data[top]=x;  
}  
// 功    能：栈顶元素弹栈  
template <class T>  
T SeqStack<T>::Pop( )  
{   
    T x;  
    if (Empty()) throw "Downflow";  
    x=data[top--];  
    return x;  
}

// 功    能：读取当前的栈顶元素  
template <class T>   
T SeqStack<T>::GetTop( )  
{  
 if (Empty())throw"Downflow";    
    return data[top];  
}

//功    能：判断栈是否为空  
template <class T>   
bool SeqStack<T>::Empty( )  
{  
 if(top==-1) return 1;  
 else return 0;  
}

//功    能：判断栈是否为满  
template <class T>   
bool SeqStack<T>::Full( )  
{  
 return top==StackSize-1;  
}   
//图  
const int MaxSize=20;     // 顶点个数的最大值  
int visited[MaxSize];     //访问标志数组（0表示未访问，1表示已访问）

//定义边表结点  
struct ArcNode                   
{  
 int adjvex;                    //邻接点的序号  
 ArcNode \*next;                 //指向下一个边结点的指针  
};

 //定义顶点表结点  
template <class T>  
struct VertexNode                
{  
 int in;     //各顶点的入度  
 T vertex;                 //顶点的名称  
 ArcNode \*firstedge;        //指向第一个边表结点的头指针  
};

//邻接表类  
template <class T>  
class ALGraph  
{  
public:  
   ALGraph(T a[ ], int n, int e);   //构造函数，初始化一个有n个顶点e条边的图  
   ~ALGraph();                      //析构函数，释放邻接表中各边表结点的存储空间  
   void DispALGraph();              //输出邻接表  
   void CountInD();     //计算各个顶点的入度,存储在adjlist[i].in中  
   void DispInD();     //输出各顶点的入度  
   void TopSort();     //拓扑排序，失败，抛出异常：Failure  
private:  
   VertexNode<T> adjlist[MaxSize];  //存放顶点表的数组  
   int vertexNum, arcNum;           //图的顶点数和边数  
};

/\*  
 \*前置条件：图不存在  
 \*输    入：无   
 \*功    能：图的初始化  
 \*输    出：无  
 \*后置条件：得到一个无向图  
 \*/  
template <class T>  
ALGraph<T>::ALGraph(T a[ ], int n, int e)  
{  
    arcNum = e;                             //边数  
 vertexNum=n;                            //顶点数  
    int i,j;  
 for (i=0; i<vertexNum; i++)  
 {          
    adjlist[i].vertex = a[i];  
    adjlist[i].firstedge = NULL;       
 }

 for (int k=0; k<arcNum; k++)    //依次输入每一条边，并在相应边表中插入结点  
    {   
  cin>>i>>j;                         //输入边所依附的两个顶点的序号  
        ArcNode \*s=new ArcNode;   
  s->adjvex=j;  //生成一个边表结点s  
     s->next=adjlist[i].firstedge;      //将结点s插入到i号表的头结点之后    
        adjlist[i].firstedge=s;  
 }  
}  
/\*   前置条件：图已存在  
 \*   输    入：无   
 \*   功    能：销毁图  
 \*   输    出：无  
 \*   后置条件：释放图所占用的存储空间  
 \*/  
template <class T>  
ALGraph<T>::~ALGraph( )  
{  
  for (int i=0; i<vertexNum; i++)  
  {  
    ArcNode \* p=adjlist[i].firstedge;  
 while (p!=NULL)                                              //循环删除  
 {  
  adjlist[i].firstedge=p->next;  
  delete p;                                                 //释放结点空间  
  p=adjlist[i].firstedge;  
 }  
  }  
}

/\*     
 \*前置条件：图已存在  
 \*输    入：无  
 \*功    能：输出图中所有顶点及边的数据信息  
 \*输    出：图中所有顶点及边的数据信息  
 \*后置条件：图保持不变  
 \*/  
template <class T>  
void ALGraph<T>::DispALGraph( )                       
{                                    
 int i;  
 ArcNode \*p;  
 cout<<"图的邻接表:\n";  
    for(i=0;i<vertexNum;i++){        
  cout<<i<<" "<<adjlist[i].vertex<<" "; //输出图中顶点的序号i及值  
  for(p=adjlist[i].firstedge;p;p=p->next)  
   cout<<p->adjvex<<" ";  //输出i号顶点的邻接点的序号   
  cout<<endl;  
 }  
}

//计算各个顶点的入度  
template <class T>  
void ALGraph<T>::CountInD()  
{

}  
//输出各个顶点的入度  
template <class T>  
void ALGraph<T>::DispInD()  
{  
 int i;  
 cout<<"In:";  
    for(i=0;i<vertexNum;i++)  
  cout<<adjlist[i].in<<" ";  
 cout<<endl;  
}  
//topsort  
template <class T>  
void ALGraph<T>::TopSort()  
{

}

int main()  
{  
 string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息  
 int n,e;  
 cin>>n>>e;  //输入顶点个数和边个数  
    ALGraph<string> G(a,n,e);  
 G.DispALGraph();  
 G.CountInD();  
 G.DispInD();  
 try{  
  G.TopSort();  
 }  
 catch(const char \*ms)  
 {  
  cout<<ms<<endl;  
 }  
    return 0;  
}

**Input**

**Output**

**Sample Input**

6 7

0 1 0 3 1 2 1 5 2 4 3 5 4 5

**Sample Output**

图的邻接表:

0 A 3 1

1 B 5 2

2 C 4

3 D 5

4 E 5

5 F

In:0 1 1 1 1 3

TopSort:A B C E D F

#include <iostream>

#include <string>

using namespace std;

//栈类

const int StackSize=100; //10只是示例性的数据，可以根据实际问题具体定义

template <class T> //定义模板类SeqStack

class SeqStack

{

public:

SeqStack( ) ; //构造函数，栈的初始化

~SeqStack( ); //析构函数

void Push(T x); //将元素x入栈

T Pop( ); //将栈顶元素弹出

T GetTop( ); //取栈顶元素（并不删除）

bool Empty( ); //判断栈是否为空

bool Full(); //判断栈是否为满

private:

T data[StackSize]; //存放栈元素的数组

int top; //栈顶指针，指示栈顶元素在数组中的下标

};

//初始化空栈

template <class T>

SeqStack<T>::SeqStack( )

{

top=-1;

}

//功 能：销毁栈

template <class T>

SeqStack<T>::~SeqStack( )

{

}

//功 能：元素x压栈

template <class T>

void SeqStack<T>::Push(T x)

{

if (Full()) throw "Overflow";

top++;

data[top]=x;

}

// 功 能：栈顶元素弹栈

template <class T>

T SeqStack<T>::Pop( )

{

T x;

if (Empty()) throw "Downflow";

x=data[top--];

return x;

}

// 功 能：读取当前的栈顶元素

template <class T>

T SeqStack<T>::GetTop( )

{

if (Empty())throw"Downflow";

return data[top];

}

//功 能：判断栈是否为空

template <class T>

bool SeqStack<T>::Empty( )

{

if(top==-1) return 1;

else return 0;

}

//功 能：判断栈是否为满

template <class T>

bool SeqStack<T>::Full( )

{

return top==StackSize-1;

}

//图

const int MaxSize=20; // 顶点个数的最大值

int visited[MaxSize]; //访问标志数组（0表示未访问，1表示已访问）

//定义边表结点

struct ArcNode

{

int adjvex; //邻接点的序号

ArcNode \*next; //指向下一个边结点的指针

};

//定义顶点表结点

template <class T>

struct VertexNode

{

int in; //各顶点的入度

T vertex; //顶点的名称

ArcNode \*firstedge; //指向第一个边表结点的头指针

};

//邻接表类

template <class T>

class ALGraph

{

public:

ALGraph(T a[ ], int n, int e); //构造函数，初始化一个有n个顶点e条边的图

~ALGraph(); //析构函数，释放邻接表中各边表结点的存储空间

void DispALGraph(); //输出邻接表

void CountInD(); //计算各个顶点的入度,存储在adjlist[i].in中

void DispInD(); //输出各顶点的入度

void TopSort(); //拓扑排序，失败，抛出异常：Failure

private:

VertexNode<T> adjlist[MaxSize]; //存放顶点表的数组

int vertexNum, arcNum; //图的顶点数和边数

};

/\*

\*前置条件：图不存在

\*输 入：无

\*功 能：图的初始化

\*输 出：无

\*后置条件：得到一个无向图

\*/

template <class T>

ALGraph<T>::ALGraph(T a[ ], int n, int e)

{

arcNum = e; //边数

vertexNum=n; //顶点数

int i,j;

for (i=0; i<vertexNum; i++)

{

adjlist[i].vertex = a[i];

adjlist[i].firstedge = NULL;

}

for (int k=0; k<arcNum; k++) //依次输入每一条边，并在相应边表中插入结点

{

cin>>i>>j; //输入边所依附的两个顶点的序号

ArcNode \*s=new ArcNode;

s->adjvex=j; //生成一个边表结点s

s->next=adjlist[i].firstedge; //将结点s插入到i号表的头结点之后

adjlist[i].firstedge=s;

}

}

/\* 前置条件：图已存在

\* 输 入：无

\* 功 能：销毁图

\* 输 出：无

\* 后置条件：释放图所占用的存储空间

\*/

template <class T>

ALGraph<T>::~ALGraph( )

{

for (int i=0; i<vertexNum; i++)

{

ArcNode \* p=adjlist[i].firstedge;

while (p!=NULL) //循环删除

{

adjlist[i].firstedge=p->next;

delete p; //释放结点空间

p=adjlist[i].firstedge;

}

}

}

/\*

\*前置条件：图已存在

\*输 入：无

\*功 能：输出图中所有顶点及边的数据信息

\*输 出：图中所有顶点及边的数据信息

\*后置条件：图保持不变

\*/

template <class T>

void ALGraph<T>::DispALGraph( )

{

int i;

ArcNode \*p;

cout<<"图的邻接表:\n";

for(i=0;i<vertexNum;i++){

cout<<i<<" "<<adjlist[i].vertex<<" "; //输出图中顶点的序号i及值

for(p=adjlist[i].firstedge;p;p=p->next)

cout<<p->adjvex<<" "; //输出i号顶点的邻接点的序号

cout<<endl;

}

}

//计算各个顶点的入度

template <class T>

void ALGraph<T>::CountInD()

{

int i;

ArcNode \*p;

for( i = 0 ; i < vertexNum ; i++ )

adjlist[i].in = 0;

for( i = 0 ; i < vertexNum ; i++ )

for( p = adjlist[i].firstedge ; p ; p = p -> next )

adjlist[ p -> adjvex ].in++;

}

//输出各个顶点的入度

template <class T>

void ALGraph<T>::DispInD()

{

int i;

cout<<"In:";

for(i=0;i<vertexNum;i++)

cout<<adjlist[i].in<<" ";

cout<<endl;

}

//topsort

template <class T>

void ALGraph<T>::TopSort()

{

cout<<"TopSort:";

SeqStack<int> s;

int count=0,i,j;

ArcNode \*p;

for( i = 0 ; i < vertexNum ; i++ )

if( adjlist[i].in == 0 ) s.Push( i );

while( !s.Empty() )

{

cout<<adjlist[ s.GetTop() ].vertex<<" ";

count++;

p=adjlist[ s.Pop() ].firstedge;

while( p != NULL )

{

j=p -> adjvex;

adjlist[ j ].in--;

if( adjlist[j].in == 0 )

s.Push( j );

p = p -> next ;

}

}

if( count < vertexNum )

cout<<"有回路";

}

int main()

{

string a[]={"A","B","C","D","E","F","G","H","I","J","K","L","M","N"}; //顶点信息

int n,e;

cin>>n>>e; //输入顶点个数和边个数

ALGraph<string> G(a,n,e);

G.DispALGraph();

G.CountInD();

G.DispInD();

try{

G.TopSort();

}

catch(const char \*ms)

{

cout<<ms<<endl;

}

return 0;

}

**1028: 折半查找（1）**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 64  Solved: 33  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1028)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1028)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1028)]

**Description**

**已知有序顺序表类SortList类，实现折半查找。部分代码如下，勿改动，请补充Bin\_Search和DispList函数。**

const int MaxSize=100; //顺序表的最大长度  
//有序表类  
class SortList{  
 public:  
  SortList(){n=0;}  
  ~SortList(){}  
  void Insert(int x);//有序表的插入，使序列仍有序  
   void DispList(); //输出表  
  int Bin\_Search(int key);   //查找成功，函数返回key所在数组下标（位置），否则查找失败，返回0  
 private:  
  int r[MaxSize+1]; //存储元素(r[1]~r[n]存储元素)  
      int n;   //顺序表实际长度  
};

//在有序表中插入元素x，使序列仍有序  
void SortList::Insert(int x){   
    int i;      
 if (n>=MaxSize)         //表满不能插入   
  throw "Overflow";  
 r[0]=x;  
 for(i=n;r[i]>x;i--)  
  r[i+1]=r[i];//将i位置元素后移  
    r[i+1]=x;                 //在位置i+1插入元素x  
    n++;                    //线性表长度增1      
}

int main(){  
 SortList A;  //空表A   
 int x,key;  
 //利用插入函数创建有序表，以0结束  
 while(1){  
  cin>>x;  
  if(!x)break;  
  try{  
   A.Insert(x);  
  }  
  catch(char \*wrong){  
   cout<<wrong<<endl;  
  }  
 }  
 A.DispList();  
 int pos;  
 cin>>key;  
 pos=A.Bin\_Search(key);  
 if(!pos)//查找失败  
  cout<<"Find "<<key<<" failure\n";  
 else cout<<"Find "<<key<<" success,position:"<<pos<<endl;  
 return 0;  
}

**Input**

**Output**

**Sample Input**

43 53 1 25 2 426 324 345 423 34 0 25

**Sample Output**

Data:1 2 25 34 43 53 324 345 423 426

Find 25 success,position:3

#include <iostream>

#include <string>

using namespace std;

const int MaxSize=100; //顺序表的最大长度

//有序表类

class SortList{

public:

SortList(){n=0;}

~SortList(){}

void Insert(int x);//有序表的插入，使序列仍有序

void DispList(); //输出表

int Bin\_Search(int key); //查找成功，函数返回key所在数组下标（位置），否则查找失败，返回0

private:

int r[MaxSize+1]; //存储元素(r[1]~r[n]存储元素)

int n; //顺序表实际长度

};

//在有序表中插入元素x，使序列仍有序

void SortList::Insert(int x){

int i;

if (n>=MaxSize) //表满不能插入

throw "Overflow";

r[0]=x;

for(i=n;r[i]>x;i--)

r[i+1]=r[i];//将i位置元素后移

r[i+1]=x; //在位置i+1插入元素x

n++; //线性表长度增1

}

void SortList::DispList() //输出表

{

int i;

cout<<"Data:";

for( i=1 ; i<=n ; i++ )

cout<<r[i]<<" ";

cout<<endl;

}

int SortList::Bin\_Search(int key)

{

int a=1,b=n;

while( a<=b )

{

if( key == r[(a+b)/2] )

return (a+b)/2;

else

{

if( key < r[(a+b)/2] )

b=(a+b)/2-1;

if( key > r[(a+b)/2] )

a=(a+b)/2+1;

}

}

return 0;

}

int main(){

SortList A; //空表A

int x,key;

//利用插入函数创建有序表，以0结束

while(1){

cin>>x;

if(!x)break;

try{

A.Insert(x);

}

catch(char \*wrong){

cout<<wrong<<endl;

}

}

A.DispList();

int pos;

cin>>key;

pos=A.Bin\_Search(key);

if(!pos)//查找失败

cout<<"Find "<<key<<" failure\n";

else cout<<"Find "<<key<<" success,position:"<<pos<<endl;

return 0;

}

**1029: 折半查找（2）**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 27  Solved: 26  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1029)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1029)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1029)]

**Description**

**已知有序顺序表类SortList类，计算折半查找等概论查找条件下的ASL值。部分代码如下，勿改动，请补充ASL\_Bin\_Search和DispList函数。**

#include <iostream>  
using namespace std;  
const int MaxSize=100; //顺序表的最大长度  
//有序表类  
class SortList{  
 public:  
  SortList(){n=0;}  
  ~SortList(){}  
  void Insert(int x);//有序表的插入，使序列仍有序  
   void DispList(); //输出表  
  int Bin\_Search(int key);  //返回值为查找key值所需的比较次数  
  double ASL\_Bin\_Search();  
 private:  
  int r[MaxSize+1]; //存储元素(r[1]~r[n]存储元素)  
      int n;   //顺序表实际长度  
};

//在有序表中插入元素x，使序列仍有序  
void SortList::Insert(int x){   
    int i;      
 if (n>=MaxSize)         //表满不能插入   
  throw "Overflow";  
 r[0]=x;  
 for(i=n;r[i]>x;i--)  
  r[i+1]=r[i];//将i位置元素后移  
    r[i+1]=x;                 //在位置i+1插入元素x  
    n++;                    //线性表长度增1      
}

//计算ASL  
double SortList::ASL\_Bin\_Search()  
{  
 int i,ASL=0;  
 for(i=1;i<=n;i++)  
  ASL+=Bin\_Search(r[i]);  //累加各个元素所需的比较次数  
    return 1.0\*ASL/n;  
}

void SortList::DispList( ){ //输出线性表  
    int i;  
 cout<<"Data:";  
    for (i=1;i<=n;i++)  
  cout<<r[i]<<" ";   
    cout<<endl;  
}

int main(){  
 SortList A;  //空表A   
 int x;  
 //利用插入函数创建有序表，以0结束  
 while(1){  
  cin>>x;  
  if(!x)break;  
  try{  
   A.Insert(x);  
  }  
  catch(char \*wrong){  
   cout<<wrong<<endl;  
  }  
 }  
 A.DispList();  
 double ASL;  
 ASL=A.ASL\_Bin\_Search();  
 cout<<"ASL:"<<ASL<<endl;  
 return 0;  
}

**Input**

**Output**

**Sample Input**

43 53 1 25 2 426 324 345 423 34 0

**Sample Output**

Data:1 2 25 34 43 53 324 345 423 426

ASL:2.9

#include <iostream>

using namespace std;

const int MaxSize=100; //顺序表的最大长度

//有序表类

class SortList{

public:

SortList(){n=0;}

~SortList(){}

void Insert(int x);//有序表的插入，使序列仍有序

void DispList(); //输出表

int Bin\_Search(int key); //返回值为查找key值所需的比较次数

double ASL\_Bin\_Search();

private:

int r[MaxSize+1]; //存储元素(r[1]~r[n]存储元素)

int n; //顺序表实际长度

};

//在有序表中插入元素x，使序列仍有序

void SortList::Insert(int x){

int i;

if (n>=MaxSize) //表满不能插入

throw "Overflow";

r[0]=x;

for(i=n;r[i]>x;i--)

r[i+1]=r[i];//将i位置元素后移

r[i+1]=x; //在位置i+1插入元素x

n++; //线性表长度增1

}

//计算ASL

double SortList::ASL\_Bin\_Search()

{

int i,ASL=0;

for(i=1;i<=n;i++)

ASL+=Bin\_Search(r[i]); //累加各个元素所需的比较次数

return 1.0\*ASL/n;

}

void SortList::DispList( ){ //输出线性表

int i;

cout<<"Data:";

for (i=1;i<=n;i++)

cout<<r[i]<<" ";

cout<<endl;

}

int SortList::Bin\_Search(int key) //返回值为查找key值所需的比较次数

{

int i=0,a=1,b=n;

while(a<=b)

{ i++;

if( key == r[(a+b)/2] )

return i;

else {

if( key < r[(a+b)/2] ) b=(a+b)/2-1;

else a=(a+b)/2+1;

}

}

}

int main(){

SortList A; //空表A

int x;

//利用插入函数创建有序表，以0结束

while(1){

cin>>x;

if(!x)break;

try{

A.Insert(x);

}

catch(char \*wrong){

cout<<wrong<<endl;

}

}

A.DispList();

double ASL;

ASL=A.ASL\_Bin\_Search();

cout<<"ASL:"<<ASL<<endl;

return 0;

}

**1030: 二叉排序树（1）-递归方式**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 42  Solved: 34  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1030)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1030)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1030)]

**Description**

**已知二叉排序树的部分代码如下，勿改动。请补充实现递归查找算法SearchBST和InsertBST。**

//BiNode  
template <class T>  
struct BiNode  
{  
 T data;  
 BiNode \*lchild, \*rchild;  
};

//BiSortTree  
template <class T>  
class BiSortTree     
{  
public:  
 BiSortTree(T r[],int n);    //建立关键字序列r[0]~r[n-1]的二叉排序树  
 ~BiSortTree( );       //析构函数，释放二叉排序树中所有结点，同二叉链表的析构函数  
    void InsertBST(BiNode<T>\*&root,BiNode<T> \*s);    //插入结点\*s  
 BiNode<T>\* SearchBST(BiNode<T> \*root,T k);   //查找值为k的结点，返回值为k所在结点的地址  
 void InOrderBST(BiNode<T> \*root);   //中序遍历二叉树（递归）  
    BiNode<T> \*GetRoot()  //获取root值  
 {  
  return root;  
 }  
private:  
    BiNode<T> \*root;    //二叉排序树（即二叉链表）的根指针  
    void Release(BiNode<T> \*&root);   //析构函数调用  
};

//构造函数，将r[0]~r[n-1]各个元素依次插入，生成一棵二叉排序树  
template <class T>  
BiSortTree<T>::BiSortTree(T r[],int n)//构造函数  
{  
 root=NULL;  //初始化空二叉树  
 int i;  
 for(i=0;i<n;i++){   //进行n次插入  
  BiNode<T> \*s;  
  s=new BiNode<T>;  
  s->data=r[i];  
  s->lchild=s->rchild=NULL;  
  InsertBST(root,s);//将结点s插入到二叉排序树中  
 }  
}

//析构函数，调用Release释放内存  
template <class T>  
BiSortTree<T>::~BiSortTree( )  
{   
    Release(root);   
}

//释放二叉排序树的存储空间，调用析构函数实现  
template <class T>  
void BiSortTree<T>::Release(BiNode<T>\* &root)  
{  
 if (root){                    
  Release(root->lchild);   //释放左子树  
  Release(root->rchild);   //释放右子树  
  delete root;  
 }    
}

//二叉排序树的主函数

int main( ){  
    int a[10],n;  
 n=0;  
 while(1)  
 {  
  cin>>a[n];  
  if(!a[n])break;  //输入数据以0结束  
  n++;  
 }   
 BiSortTree<int> bst(a,n);//构造二叉排序树  
    BiNode<int> \*root;  
 root=bst.GetRoot();  
 cout<<"Inorder:";  
 bst.InOrderBST(root);//中序遍历二叉排序树（得到递增有序序列）  
 cout<<endl;  
 int x;  
 cin>>x;  
 BiNode<int> \*s;  
    s=bst.SearchBST(root,x);  
 if(!s) cout<<"Failure";  
 else cout<<"Success";  
 cout<<endl;  
 return 0;  
}

**Input**

**Output**

**Sample Input**

3 42 4 32 42 42 32 56 37 0 32

**Sample Output**

Inorder:3 4 32 37 42 56

Success

#include <iostream>

using namespace std;

template <class T>

struct BiNode

{

T data;

BiNode \*lchild, \*rchild;

};

//BiSortTree

template <class T>

class BiSortTree

{

public:

BiSortTree(T r[],int n); //建立关键字序列r[0]~r[n-1]的二叉排序树

~BiSortTree( ); //析构函数，释放二叉排序树中所有结点，同二叉链表的析构函数

void InsertBST(BiNode<T>\*&root,BiNode<T> \*s); //插入结点\*s

BiNode<T>\* SearchBST(BiNode<T> \*root,T k); //查找值为k的结点，返回值为k所在结点的地址

void InOrderBST(BiNode<T> \*root); //中序遍历二叉树（递归）

BiNode<T> \*GetRoot() //获取root值

{

return root;

}

private:

BiNode<T> \*root; //二叉排序树（即二叉链表）的根指针

void Release(BiNode<T> \*&root); //析构函数调用

};

//构造函数，将r[0]~r[n-1]各个元素依次插入，生成一棵二叉排序树

template <class T>

BiSortTree<T>::BiSortTree(T r[],int n)//构造函数

{

root=NULL; //初始化空二叉树

int i;

for(i=0;i<n;i++){ //进行n次插入

BiNode<T> \*s;

s=new BiNode<T>;

s->data=r[i];

s->lchild=s->rchild=NULL;

InsertBST(root,s);//将结点s插入到二叉排序树中

}

}

//析构函数，调用Release释放内存

template <class T>

BiSortTree<T>::~BiSortTree( )

{

Release(root);

}

//释放二叉排序树的存储空间，调用析构函数实现

template <class T>

void BiSortTree<T>::Release(BiNode<T>\* &root)

{

if (root){

Release(root->lchild); //释放左子树

Release(root->rchild); //释放右子树

delete root;

}

}

//二叉排序树的主函数

template <class T>

void BiSortTree<T>::InOrderBST(BiNode<T> \*root) //中序遍历二叉树（递归）

{

if( root )

{

InOrderBST( root->lchild );

cout<<root->data<<" ";

InOrderBST( root->rchild );

}

}

template <class T>

void BiSortTree<T>::InsertBST(BiNode<T>\*&root,BiNode<T> \*s) //插入结点\*s

{

if(root)

{

if( s->data < root->data )

InsertBST( root->lchild , s);

if( s->data > root->data )

InsertBST( root->rchild , s);

}

else root=s;

}

template <class T>

BiNode<T>\* BiSortTree<T>::SearchBST(BiNode<T> \*root,T k) //查找值为k的结点，返回值为k所在结点的地址

{

if( root )

{

if( k==root->data ) return root;

if( k < root->data )

SearchBST(root->lchild ,k);

else

SearchBST(root->rchild, k);

}

else return NULL;

}

int main( ){

int a[10],n;

n=0;

while(1)

{

cin>>a[n];

if(!a[n])break; //输入数据以0结束

n++;

}

BiSortTree<int> bst(a,n);//构造二叉排序树

BiNode<int> \*root;

root=bst.GetRoot();

cout<<"Inorder:";

bst.InOrderBST(root);//中序遍历二叉排序树（得到递增有序序列）

cout<<endl;

int x;

cin>>x;

BiNode<int> \*s;

s=bst.SearchBST(root,x);

if(!s) cout<<"Failure";

else cout<<"Success";

cout<<endl;

return 0;

}

**1031: 二叉排序树（2）-非递归方式**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 29  Solved: 25  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1031)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1031)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1031)]

**Description**

**已知二叉排序树部分代码如下，勿改动。请实现非递归函数InsertBST和SearchBST。**

//BiNode  
template <class T>  
struct BiNode  
{  
 T data;  
 BiNode \*lchild, \*rchild;  
};

//BiSortTree  
template <class T>  
class BiSortTree     
{  
public:  
 BiSortTree(T r[],int n);    //建立关键字序列r[0]~r[n-1]的二叉排序树  
 ~BiSortTree( );       //析构函数，释放二叉排序树中所有结点，同二叉链表的析构函数  
    void InsertBST(BiNode<T> \*s);    //插入结点\*s  
 BiNode<T>\* SearchBST(T k);   //查找值为k的结点，返回值为k所在结点的地址  
 void InOrderBST()   //中序遍历  
 {  
  InOrderBST(root);  //调用私有中序递归函数  
 }  
private:  
    BiNode<T> \*root;    //二叉排序树（即二叉链表）的根指针  
    void Release(BiNode<T> \*&root);   //析构函数调用  
 void InOrderBST(BiNode<T> \*root);   
};

//构造函数，将r[0]~r[n-1]各个元素依次插入，生成一棵二叉排序树  
template <class T>  
BiSortTree<T>::BiSortTree(T r[],int n)//构造函数  
{  
 root=NULL;  //初始化空二叉树  
 int i;  
 for(i=0;i<n;i++){   //进行n次插入  
  BiNode<T> \*s;  
  s=new BiNode<T>;  
  s->data=r[i];  
  s->lchild=s->rchild=NULL;  
  InsertBST(s);//将结点s插入到二叉排序树中  
 }  
}

//中序遍历叉排序树,通过中序遍历是否为递增序列来验证构造的二叉排序树是否正确  
template <class T>  
void BiSortTree<T>::InOrderBST(BiNode<T> \*root){  
 if (root){  
  InOrderBST(root->lchild);  
  cout<<root->data<<" ";  
  InOrderBST(root->rchild);  
 }  
}

//析构函数，调用Release释放内存  
template <class T>  
BiSortTree<T>::~BiSortTree( )  
{   
    Release(root);   
}

//释放二叉排序树的存储空间，调用析构函数实现  
template <class T>  
void BiSortTree<T>::Release(BiNode<T>\* &root)  
{  
 if (root){                    
  Release(root->lchild);   //释放左子树  
  Release(root->rchild);   //释放右子树  
  delete root;  
 }    
}

//二叉排序树的主函数，文件名为bisorttreemain.cpp  
int main( ){  
    int a[10],n;  
 n=0;  
 while(1)  
 {  
  cin>>a[n];  
  if(!a[n])break;  //输入数据以0结束  
  n++;  
 }   
 BiSortTree<int> bst(a,n);//构造二叉排序树  
    cout<<"Inorder:";  
 bst.InOrderBST();//中序遍历二叉排序树（得到递增有序序列）  
 cout<<endl;  
 int x;  
 cin>>x;  
 BiNode<int> \*s;  
    s=bst.SearchBST(x);  
 if(!s) cout<<"Failure";  
 else cout<<"Success";  
 cout<<endl;  
 return 0;  
}

**Input**

**Output**

**Sample Input**

3 42 4 32 42 42 32 56 37 0 32

**Sample Output**

Inorder:3 4 32 37 42 56

Success

#include <iostream>

using namespace std;

template <class T>

struct BiNode

{

T data;

BiNode \*lchild, \*rchild;

};

//BiSortTree

template <class T>

class BiSortTree

{

public:

BiSortTree(T r[],int n); //建立关键字序列r[0]~r[n-1]的二叉排序树

~BiSortTree( ); //析构函数，释放二叉排序树中所有结点，同二叉链表的析构函数

void InsertBST(BiNode<T> \*s); //插入结点\*s

BiNode<T>\* SearchBST(T k); //查找值为k的结点，返回值为k所在结点的地址

void InOrderBST() //中序遍历

{

InOrderBST(root); //调用私有中序递归函数

}

private:

BiNode<T> \*root; //二叉排序树（即二叉链表）的根指针

void Release(BiNode<T> \*&root); //析构函数调用

void InOrderBST(BiNode<T> \*root);

};

//构造函数，将r[0]~r[n-1]各个元素依次插入，生成一棵二叉排序树

template <class T>

BiSortTree<T>::BiSortTree(T r[],int n)//构造函数

{

root=NULL; //初始化空二叉树

int i;

for(i=0;i<n;i++){ //进行n次插入

BiNode<T> \*s;

s=new BiNode<T>;

s->data=r[i];

s->lchild=s->rchild=NULL;

InsertBST(s);//将结点s插入到二叉排序树中

}

}

template <class T>

void BiSortTree<T>::InsertBST(BiNode<T> \*s) //插入结点\*s

{

if( !this->root ) this->root = s;

else {

BiNode<T> \* root = this->root;

while( root )

{

if( root->data < s->data ) {

if( !root -> rchild ) {

root->rchild = s; break;}

else root=root->rchild;

}

else

if( root ->data > s->data ){

if( !root->lchild ) {

root->lchild = s; break;}

else root=root->lchild;

}

else break;

}

}

}

template <class T>

BiNode<T>\* BiSortTree<T>::SearchBST(T k) //查找值为k的结点，返回值为k所在结点的地址

{

BiNode<T>\* root = this->root ;

while( root )

{

if( root -> data == k)

return root;

else

if(root -> data < k)

root = root -> rchild;

else

root = root -> lchild;

}

return NULL;

}

//中序遍历叉排序树,通过中序遍历是否为递增序列来验证构造的二叉排序树是否正确

template <class T>

void BiSortTree<T>::InOrderBST(BiNode<T> \*root){

if (root){

InOrderBST(root->lchild);

cout<<root->data<<" ";

InOrderBST(root->rchild);

}

}

//析构函数，调用Release释放内存

template <class T>

BiSortTree<T>::~BiSortTree( )

{

Release(root);

}

//释放二叉排序树的存储空间，调用析构函数实现

template <class T>

void BiSortTree<T>::Release(BiNode<T>\* &root)

{

if (root){

Release(root->lchild); //释放左子树

Release(root->rchild); //释放右子树

delete root;

}

}

//二叉排序树的主函数，文件名为bisorttreemain.cpp

int main( ){

int a[10],n;

n=0;

while(1)

{

cin>>a[n];

if(!a[n])break; //输入数据以0结束

n++;

}

BiSortTree<int> bst(a,n);//构造二叉排序树

cout<<"Inorder:";

bst.InOrderBST();//中序遍历二叉排序树（得到递增有序序列）

cout<<endl;

int x;

cin>>x;

BiNode<int> \*s;

s=bst.SearchBST(x);

if(!s) cout<<"Failure";

else cout<<"Success";

cout<<endl;

return 0;

}