1.函数名: bsearch

功 能: 二分法搜索

用 法: void \*bsearch(const void \*key, const void \*base, size\_t \*nelem,

size\_t width, int(\*fcmp)(const void \*, const \*));

#include <stdio.h>

#include <stdlib.h>

#define NUM 8

int compare(const void \*p, const void \*q)

{

return (\*(int \*)p - \*(int \*)q);

}

int main(int argc, char \*argv[])

{

int array[NUM] = {9, 2, 7, 11, 3, 87, 34, 6};

int key = 3;

int \*p;

qsort(array, NUM, sizeof(int), compare);

p = (int \*)bsearch(&key, array, NUM, sizeof(int), compare);

(p == NULL) ? puts("not found") : puts("found");

return 0;

}

2.最值元素

\*min\_element(myints,myints+7)

3.复制函数

#include <iostream>

using namespace std;

int main ()

{

int src[]= {1, 2, 3, 4, 5, 6, 7};

// 将数组向左移动两位

copy(src + 2, src + 7, src);

//3456767

//copy(复制开始，复制结束，起点)

copy\_backward(src+2,src+7,src+2);

//6734567

//copy(复制开始，复制结束，终点)

return 0;

}

4.交换一个区间

int a[]={1,1,1,1,1};

int b[]={2,2,2,2,2};

swap\_ranges(a+1, a+4, b);

for(int i=0;i<5;i++)

cout<<a[i];

return 0;

5.replace替换函数

#include <iostream>

#include <algorithm>

using namespace std;

/\*用法一：

\*用str替换指定字符串从起始位置pos开始长度为len的字符

\*string& replace (size\_t pos, size\_t len, const string& str);

\*/

int main()

{

string line = "this@ is@ a test string!";

line = line.replace(line.find("@"), 1, ""); //从第一个@位置替换第一个@为空

cout << line.c\_str() << endl;

/\*用法二：

\*用str替换 迭代器起始位置 和 结束位置 的字符

\*string& replace (const\_iterator i1, const\_iterator i2, const string& str);

\*/

string line = "this@ is@ a test string!";

line = line.replace(line.begin(), line.begin()+6, ""); //用str替换从begin位置开始的6个字符

cout << line << endl;

/\*用法三：

\*用substr的指定子串（给定起始位置和长度）替换从指定位置上的字符串

\*string& replace (size\_t pos, size\_t len, const string& str, size\_t subpos, size\_t sublen);

\*/

string line = "this@ is@ a test string!";

string substr = "12345";

line = line.replace(0, 5, substr, substr.find("1"), 3); //用substr的指定子串（从1位置数共3个字符）替换从0到5位置上的line

cout << line << endl;

return 0;

}

6、rotate(begin,middle,end)

#include<stdio.h>

#include<algorithm>

using namespace std;

int main()

{

char str[12]="this is acm";

rotate(str,str+5,str+11);

printf("%s\n",str);

return 0;

}

#include<iostream>

using namespace std;

const int MaxSize=100;

template<class DataType>//这个叫做模板类，它的作用是可以在之后创建对象的时候再确定所使用的参数类型，也就是可以创建各种类型，如果使用模板类，属于这个类的函数方法必须在每个函数之前都如：template<class DataType>这样声明一遍

class SeqList

{

public:

SeqList(){length=0;MaxSize=100;}//这个为无参构造函数，一般采用 类名+对象名()下面的为有参构造函数，这是类的一种特殊的函数，

SeqList(DataType a[],int n);//这是有参构造函数，采用 类名+对象名(参数 参数)进行初始化，在实例化过程中，采用SeqList i0(),调用的就是上面的无参构造函数，采用SeqList i1(数组名，数组长度)调用这个有参构造函数

~SeqList(){}

void Upsidedown(DataType a[],int n);

int getlength(){return length;}

DataType Get(int n);

int Locate(DataType x);

void Insert(int i,DataType x);

DataType Delete(int i);

void PrintList();

private:

DataType data[MaxSize];

int length;

};

//顺序表

template<class DataType>

SeqList<DataType>::SeqList(DataType a[],int n)//构造函数

{

if(n>MaxSize)throw"参数非法";

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

data[i]=a[i];

length=n;

}

template<class DataType>

void SeqList<DataType>::Upsidedown(DataType a[],int n)//倒置函数

{

if(n>MaxSize)throw"参数非法";

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

data[i]=a[n-i-1];

length=n;

}

template<class DataType>

DataType SeqList<DataType>::Get(int i)//按位查找

{

if(i<1&&i>length)throw"查找位置非法";

else return data[i-1];

}

template<class DataType>

int SeqList<DataType>::Locate(DataType x)//按值查找

{

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

if(data[i]==x)return i+1;

return 0;

}

template<class DataType>

void SeqList<DataType>::Insert(int i,DataType x)//插入算法

{

if(length>=MaxSize)throw"上溢";

if(i<1||i>length+1)throw"位置";

for(int j=length;j>=i;j--)

data[j]=data[j-1];//注意这个算法不要把这三句加大括号

data[i-1]=x;

length++;

}

template<class DataType>

DataType SeqList<DataType>::Delete(int i)//删除算法

{

if(length==0)throw"下溢";

if(i<1||i>length)throw"位置";

DataType x=data[i-1];

for(int j=i;j<length;j++)

data[j-1]=data[j];

length--;

return x;

}

template<class DataType>

void SeqList<DataType>::PrintList()

{

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

cout<<data[i]<<" ";

cout<<endl;

}

//单链表,单链表的使用方法是书上的源码，书上未对一部分指针进行定义，见谅

template<class DataType>

struct Node

{

DataType data;

Node<DataType> \*next;

};//这里是一个节点结构体，结构体中给出了一个DataType类型的数据，一个DataType类型的指针

template<class DataType>

class LinkList

{

public:

LinkList();

LinkList(DataType a[],int n);

~LinkList();

int getlength();

DataType Get(int i);

int Locate(DataType x);

void Insert(int i,DataType x);

DataType Delete(int i);

void Delete\_2(DataType x);

void PrintList();

private:

Node<DataType> \*first;

Node<DataType> \*p;

};

template<class DataType>

void LinkList<DataType>::PrintList()//遍历算法

{

p=first->next;

while(p!=NULL)

{

cout<<p->data<<" ";

p=p->next;

}

}

template<class DataType>

int LinkList<DataType>::getlength()//求长度

{

p=first->next;

int count=0;

while(p!=NULL)

{

p=p->next;

count++;

}

return count;

}

template<class DataType>

DataType LinkList<DataType>::Get(int i)//按位查找

{

p=first->next;int count=1;

while(p!=NULL&&count<i)

{

p=p->next;

count++;

}

if(p==NULL)throw"位置";

else return p->data;

}

template<class DataType>

int LinkList<DataType>::Locate(DataType x)//按值查找

{

p=first->next;

int count=1;

while(p!=NULL)

{

if(p->data==x)return count;

p=p->next;

count++;

}

return 0;

}

template<class DataType>

void LinkList<DataType>::Insert(int i,DataType x)//插入算法

{

Node<DataType> \*s;//这里有必要对每一个用到的指针都进行定义，否则程序将会报错

p=first;

int count=0;

while(p!=NULL&&count<i-1)

{

p=p->next;

count++;

}

if(p==NULL)throw"位置";

else

{

s=new Node<DataType>;s->data=x;

s->next=p->next;p->next=s;

}

}

template<class DataType>

LinkList<DataType>::LinkList(DataType a[],int n)//尾插法创建单链表

{

Node<DataType> \*r;

Node<DataType> \*s;

first=new Node<DataType>;

r=first;

for(int i=0;i<n;i++)//记得这种函数里面i不可以没有定义！

{

s=new Node<DataType>;s->data=a[i];

r->next=s;r=s;

}

r->next=NULL;

}

template<class DataType>

DataType LinkList<DataType>::Delete(int i)//删除算法

{

DataType x;//这里尤其要注意这个x不是指针，只是一个DataType类型的数据，因为下面x指向的是---指针q所指向的数据

Node<DataType> \*q;

p=first;int count=0;

while(p!=NULL&&count<i-1)

{

p=p->next;

count++;

}

if(p==NULL||p->next==NULL)

throw"位置";

else{

q=p->next;x=q->data;

p->next=q->next;

delete q;

return x;

}

}

template<class DataType>

void LinkList<DataType>::Delete\_2(DataType x)//删除（按值）算法

{

int count;

Node<DataType> \*q;

p=first->next;

count=0;

while(p!=NULL&&p->data!=x)

{

p=p->next;

count++;

}

if(p==NULL||p->next==NULL)

throw"位置";

else

{

q=p->next;

x=q->data;

p->next=q->next;

delete q;

}

}

template<class DataType>

LinkList<DataType>::~LinkList()//析构函数

{

cout<<"i3的析构函数被调用"<<endl;

Node<DataType> \*q;

while(first!=NULL)

{

q=first;

first=first->next;

delete q;

}

}

int main()

{

//这里仅仅根据顺序表给出相应的应用方法,单链表与此相似

int a[3]={1,2,3};//创建一个数组

SeqList<int> i1(a,3);//1.以int型创建实例为例，在这里制定<DataType>的类型，用了构造函数对顺序表进行初始化，创建了1,2,3在顺序表中，相当于SeqList类的一个对象i1，如果类和对象不懂，请参考面向对象程序设计，数据结构和他分不开

cout<<"表长为:"<<i1.getlength()<<endl;//利用对象i1调用getlength()函数来求表长

cout<<"第二位是:"<<i1.Get(2)<<endl;//按位查找表的第二位

cout<<"查找元素3:在第"<<i1.Locate(3)<<"位"<<endl;//查找值“3”在第几位

i1.Upsidedown(a,3);

i1.PrintList();

i1.Insert(2,4);//在第二位插入4

cout<<"第二位是:"<<i1.Get(2)<<endl;

cout<<"i1表为:";

i1.PrintList();//输出表

cout<<endl;

i1.Delete(2);//删除第二位

cout<<"此时的第二位是:"<<i1.Get(2)<<endl;

cout<<"i1表为:";

i1.PrintList();//输出表

cout<<endl;

float b[4]={0.0,1.1,2.2,3.3};

SeqList<float> i2(b,4);//2.以double型创建实例为例，这个i2是另外一个实例，和i1无任何关系

cout<<"i2表为:";

i2.PrintList();//输出表i2

cout<<endl;

//考虑良久还是写出了单链表给出的应用方法，这里采用输入的方式给出应用实例

cout<<endl;

double c[100];

int n;

cout<<"请输入单链表的长度:";

cin>>n;

for(int j=0;j<n;j++)

{

cout<<"请输入第"<<j+1<<"个元素";

cin>>c[j];

}

LinkList<double> i3(c,n);//采用尾插法建立单链表，这里有必要说一下，建立i1，i2，i3的过程同理，这里用到的都是类的一种特殊函数，即构造函数

i3.PrintList();//输出单链表

cout<<endl;

i3.Insert(4,5.5);//在第四个位置插入数据

i3.PrintList();//输出单链表

cout<<endl;

cout<<"删除的是:"<<i3.Delete(4)<<endl;//删除第四个数据并输出删除的内容，这里很多人会问删除算法中return x的作用，其实仅仅是求出删除的内容

cout<<"删除后的数据是:";

i3.PrintList();//输出单链表

cout<<endl;

cout<<"第二个数为:"<<i3.Get(2)<<endl;

//最后说下析构函数，程序结束执行，对象存储空间被释放，析构函数自动被调用\*/

return 0;

}

#include<iostream>

using namespace std;

//循环队列的实现

const int QueueSize=5;

template <class DataType>

class CirQueue

{

public:

CirQueue(){front=rear=QueueSize-1;}

~CirQueue(){}

void EnQueue(DataType x);

DataType DeQueue();

DataType GetQueue();

DataType GetRear();

int Empty()

{

if(front==rear&&flag==0)

return 1;

else

return 0;

}

private:

DataType data[QueueSize];

int front,rear;

bool flag;

};

//入队算法

template <class DataType>

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

{

rear=(rear+1)%QueueSize;

flag=1;

data[rear]=x;

cout<<data[rear]<<"入队"<<endl;

if(rear==front&&flag==1)cout<<"队满"<<endl;

}

template <class DataType>//出队算法

DataType CirQueue<DataType>::DeQueue()

{

if(rear==front&&flag==0)cout<<"队空"<<endl;

else

{

front=(front+1)%QueueSize;

flag=0;

return data[front];

}

}

template <class DataType>//读取队头元素

DataType CirQueue<DataType>::GetQueue()

{

// if(rear==front)throw"下溢";

int i=(front+1)%QueueSize;

return data[i];

}

template <class DataType>//读取队尾元素

DataType CirQueue<DataType>::GetRear()

{

// if(rear==front)throw"下溢";

int i=rear%QueueSize;

return data[i];

}

//链队列

template <class DataType>

struct Node

{

DataType data;

Node<DataType> \*next;

};

template <class DataType>

class LinkQueue

{

public:

LinkQueue();

~LinkQueue(){};

void EnQueue(DataType x);

DataType DeQueue();

DataType GetQueue();

DataType GetRear();

int Empty()

{

if(front==rear)return 1;

else return 0;

}

Node<DataType> \*front,\*rear;

};

template <class DataType>//构造函数

LinkQueue<DataType>::LinkQueue()

{

Node<DataType> \*s;

s=new Node<DataType>;s->next=NULL;

front=rear=s;

}

template <class DataType>//入队算法

void LinkQueue<DataType>::EnQueue(DataType x)

{

Node<DataType> \*s;

s=new Node<DataType>;s->data=x;

s->next=NULL;

rear->next=s;

rear=s;

}

template <class DataType>//取队头元素

DataType LinkQueue<DataType>::GetQueue()

{

if(rear==front)throw"下溢";

DataType x=front->next->data;

return x;

}

template <class DataType>//取队尾元素

DataType LinkQueue<DataType>::GetRear()

{

if(rear==front)throw"下溢";

DataType x=rear->data;

return x;

}

template <class DataType>//出队算法

DataType LinkQueue<DataType>::DeQueue()

{

if(rear==front)throw"下溢";

Node<DataType> \*p;

DataType x;

p=front->next;x=p->data;

front->next=p->next;

if(p->next==NULL)rear=front;

delete p;

return x;

}

int main()

{

cout<<"循环队列"<<endl;

CirQueue<int> i1;

cout<<"1,2,3依次入队"<<endl;

i1.EnQueue(1);i1.EnQueue(2);i1.EnQueue(3);i1.EnQueue(4);i1.EnQueue(5);//1,2,3依次入队

cout<<"现在的队头元素为:"<<i1.GetQueue()<<endl;

cout<<"现在的队尾元素为:"<<i1.GetRear()<<endl;

cout<<"队头元素"<<i1.DeQueue()<<"出队"<<endl;

cout<<"队头元素"<<i1.DeQueue()<<"出队"<<endl;

cout<<"队头元素"<<i1.DeQueue()<<"出队"<<endl;

cout<<"队头元素"<<i1.DeQueue()<<"出队"<<endl;

cout<<"队头元素"<<i1.DeQueue()<<"出队"<<endl;

i1.DeQueue();

/\* if(i1.Empty())

cout<<"队列i1空"<<endl;

else

cout<<"队列i1非空"<<endl;\*/

cout<<endl;

cout<<"链队列"<<endl;

LinkQueue<char> i2;

cout<<"a,b,c依次入队"<<endl;

i2.EnQueue('a');i2.EnQueue('b');i2.EnQueue('c');

cout<<"现在的队头元素为:"<<i2.GetQueue()<<endl;

cout<<"现在的队尾元素为:"<<i2.GetRear()<<endl;

cout<<"队头元素"<<i2.DeQueue()<<"出队"<<endl;

cout<<"队头元素"<<i2.DeQueue()<<"出队"<<endl;

cout<<"队头元素"<<i2.DeQueue()<<"出队"<<endl;

if(i2.Empty())

cout<<"队列i2空"<<endl;

else

cout<<"队列i2非空"<<endl;

return 0;

}

#include<iostream>

using namespace std;

//栈

const int StackSize=100;

template<class DataType>

class SeqStack

{

public:

SeqStack(){top=-1;}

~SeqStack(){}

void Push(DataType x);

DataType Pop();

DataType GetTop(){if(top!=-1) return data[top];}

int Empty()

{

if(top==-1)return 1;

else return 0;

}

//private:

DataType data[StackSize];

int top;

};

template<class DataType>//顺序栈入栈算法

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

{

if(top==StackSize-1)throw"上溢";

data[++top]=x;

}

template<class DataType>//顺序栈出栈算法

DataType SeqStack<DataType>::Pop()

{

if(top==-1)throw"下溢";

DataType x=data[top--];

return x;

}

//两栈共享空间

template<class DataType>

class BothStack

{

public:

BothStack(){top1=-1;top2=StackSize;}

~BothStack(){}

void Push(int i,DataType);

DataType Pop(int i);

DataType GetTop(int i);

int Empty(int i);

private:

DataType data[StackSize];

int top1,top2;

};

template<class DataType>//两栈共享空间入栈算法

void BothStack<DataType>::Push(int i,DataType x)

{

if(top1==top2-1)throw"上溢";

if(i==1)data[++top1]=x;

if(i==2)data[--top2]=x;

}

template<class DataType>//出栈算法

DataType BothStack<DataType>::Pop(int i)

{

if(i==1){

if(top1==-1)throw"下溢";

return data[top1--];

}

if(i==2){

if(top2==StackSize)throw"下溢";

return data[top2++];

}

}

template<class DataType>//取栈顶元素

DataType BothStack<DataType>::GetTop(int i)

{

if(i==1){ if(top1!=-1) return data[top1];}

if(i==2){ if(top2!=-1) return data[top2];}

}

template<class DataType>//判空操作

int BothStack<DataType>::Empty(int i)

{

if(i==1)

{

if(top1==-1)

{

cout<<"栈1空"<<endl;

return 1;

}

else

{

cout<<"栈1非空"<<endl;

return 0;

}

}

if(i==2)

{

if(top2==StackSize)

{

cout<<"栈2空"<<endl;

return 1;

}

else

{

cout<<"栈2非空"<<endl;

return 0;

}

}

else

return 0;

}

//(书上没有取栈顶元素和判空操作的代码

//链栈

template<class DataType>

struct Node

{

DataType data;

Node<DataType> \*next;

};

template<class DataType>

class LinkStack

{

public:

LinkStack(){top=NULL;}

~LinkStack(){};

void Push(DataType x);

DataType Pop();

DataType GetTop(){if(top!=NULL)return top->data;}

int Empty();

private:

Node<DataType> \*top;

};

template<class DataType>//链栈入栈算法

void LinkStack<DataType>::Push(DataType x)

{

Node<DataType> \*s;

s=new Node<DataType>;s->data=x;

s->next=top;top=s;

}

template<class DataType>//链栈出栈算法

DataType LinkStack<DataType>::Pop()

{

Node<DataType> \*p;

p=new Node<DataType>;

if(top==NULL)throw"下溢";

DataType x=top->data;p=top;

top=top->next;

delete p;

return x;

}

template<class DataType>//判空操作

int LinkStack<DataType>::Empty()

{

if(top==NULL)

{

cout<<"栈为空"<<endl;

return 1;

}

else

{

cout<<"栈非空"<<endl;

return 0;

}

}

int main()

{

//顺序栈

cout<<"顺序栈"<<endl;

SeqStack<int> i1;//创建顺序栈i1

cout<<"top="<<i1.top<<endl;

i1.Push(3);

i1.Push(2);

i1.Push(1);//依次压入3,2,1

cout<<"i1的栈顶元素是:"<<i1.GetTop()<<endl;

cout<<"取走i1中元素"<<i1.Pop()<<endl;//取出栈顶元素 1，现在的栈顶元素应该是2

cout<<"i1的栈顶元素是:"<<i1.GetTop()<<endl;

cout<<endl;

//两栈共享空间

cout<<"两栈共享"<<endl;

BothStack<char> i2;//创建两栈共享的栈i2

i2.Push(1,'c');

i2.Push(1,'b');

i2.Push(1,'a');//向1中压入c，b，a

i2.Push(2,'C');

i2.Push(2,'B');

i2.Push(2,'A');//向2中压入C,B,A

cout<<"栈1的栈顶元素是:"<<i2.GetTop(1)<<endl;

cout<<"栈2的栈顶元素是:"<<i2.GetTop(2)<<endl;

cout<<"取走栈1中元素"<<i2.Pop(1)<<endl;

cout<<"取走栈1中元素"<<i2.Pop(1)<<endl;

cout<<"取走栈1中元素"<<i2.Pop(1)<<endl;

i2.Empty(1);i2.Empty(2);

cout<<endl;

//链栈

cout<<"链栈"<<endl;

LinkStack<float> i3;//创建链栈i3

i3.Push(3);

i3.Push(2);

i3.Push(1);//依次压入3,2,1

cout<<"i3的栈顶元素是:"<<i3.GetTop()<<endl;

i3.Empty();//判断栈是否为空

cout<<"取走i3中元素"<<i3.Pop()<<endl;

cout<<"取走i3中元素"<<i3.Pop()<<endl;

cout<<"取走i3中元素"<<i3.Pop()<<endl;

i3.Empty();

return 0;

}

#include<iostream>

using namespace std;

//循环链表

template<class DataType>

struct DulNode

{

DataType data2;

DulNode<DataType> \*next2,\*prior2;

};

template<class DataType>

class DulList

{

public:

DulList();

DulList(DataType a[],int n);

int getlength();

void Insert(int i,DataType x);

DataType Delete(int i);

void PrintList();

private:

DulNode<DataType> \*first2;

DulNode<DataType> \*p2;

};

template<class DataType>

void DulList<DataType>::PrintList()//遍历算法

{

p2=first2->next2;

while(p2!=first2)

{

cout<<p2->data2<<" ";

p2=p2->next2;

}

cout<<endl;

}

template<class DataType>

int DulList<DataType>::getlength()//求长度

{

p2=first2->next2;

int count=0;

while(p2!=first2)

{

p2=p2->next2;

count++;

}

return count;

}

template<class DataType>

void DulList<DataType>::Insert(int i,DataType x)//插入算法

{

DulNode<DataType> \*s2;//这里有必要对每一个用到的指针都进行定义，否则程序将会报错

p2=first2->next2;

int count=1;

if(i==1)

{

s2=new DulNode<DataType>;s2->data2=x;

s2->prior2=first2;s2->next2=first2->next2;

s2->next2->prior2=s2;first2->next2=s2;

}

else

{

while(p2!=first2&&count<i-1)

{

p2=p2->next2;

count++;

}

if(p2==first2)throw"位置";

else

{

s2=new DulNode<DataType>;s2->data2=x;

s2->prior2=p2;s2->next2=p2->next2;

s2->next2->prior2=s2;p2->next2=s2;

}

}

}

template<class DataType>

DulList<DataType>::DulList(DataType a[],int n)//尾插法创建循环链表

{

DulNode<DataType> \*r;

DulNode<DataType> \*s;

first2=new DulNode<DataType>;

r=first2;

for(int i=0;i<n;i++)//记得这种函数里面i不可以没有定义！

{

s=new DulNode<DataType>;

s->data2=a[i];

r->next2=s;s->prior2=r;r=s;

}

r->next2=first2;

first2->prior2=r;

}

template<class DataType>

DataType DulList<DataType>::Delete(int i)//删除算法

{

DataType x;//这里尤其要注意这个x不是指针，只是一个DataType类型的数据，因为下面x指向的是---指针q所指向的数据

DulNode<DataType> \*q2;

p2=first2->next2;int count=0;

while(p2!=first2&&count<i-1)

{

p2=p2->next2;

count++;

}

if(p2==first2)throw"位置";

else{

p2->next2->prior2=p2->prior2;p2->prior2->next2=p2->next2;

delete p2;

return x;

}

}

int main()

{

int a[4]={1,2,3,4};

DulList<int> i1(a,4);

i1.PrintList();

i1.Insert(1,5);//在第一位插入5

i1.PrintList();

i1.Delete(5);//删除第五个元素

i1.PrintList();//输出链表

return 0;

}

#include<iostream>

using namespace std;

template<class DataType>

struct BiNode

{

DataType data;

BiNode<DataType> \*lchild,\*rchild;

};//定义的是二叉链表，左孩子指针，右孩子指针，data域

template<class DataType>

class BiTree

{

public:

BiTree(){count=0;cout<<"请输入根结点值:";root=Create(root);}//构造函数，构建一棵二叉树

~BiTree(){Release(root);}//析构函数，释放各节点的储存空间

void PreOrder(){cout<<"前序遍历二叉树为:";PreOrder(root);cout<<endl;}//前序遍历二叉树

void InOrder(){cout<<"中序遍历二叉树为:";InOrder(root);cout<<endl;}//中序遍历二叉树

void PostOrder(){cout<<"后序遍历二叉树为:";PostOrder(root);cout<<endl;}//后序遍历二叉树

void LeverOrder();//层序遍历二叉树

int count;//定义的是树中的结点个数

private:

BiNode<DataType> \*root;//指向根节点的头指针

BiNode<DataType>\* Create(BiNode<DataType> \*bt);

/\*构造函数调用，这里需强调一下，很多人不理解这里为什么在函数返回的数据类型后加一个'\*',

这个函数返回的是一个BiNode类型的指针，指针指向的是一个地址值，也就是说，函数返回了一个地址

这个地址值又赋给了root，指针root便有了指向，指向了根节点的位置\*/

void Release(BiNode<DataType> \*bt);//析构函数的递归删除函数

void PreOrder(BiNode<DataType> \*bt);//前序遍历函数

void InOrder(BiNode<DataType> \*bt);//中序遍历函数

void PostOrder(BiNode<DataType> \*bt);//后序遍历函数

};

template<class DataType>

BiNode<DataType>\* BiTree<DataType>::Create(BiNode<DataType> \*bt)

/\*这个函数返回的是一个BiNode类型的bt指针，传入的参数也是BiNode类型的bt指针，最开始调用构造函数的时候传入的是根节点root，

然后分别对root的左、右孩子节点调用Create()函数，传入的参数是root->lchild,root->rchild，然后继续递归调用直到 输入结束符 使那一步返回的指针bt为NUll为止

这时候，结束的那步会返回到父结点并执行父结点的右孩子结点的Create()函数，

如此循环直到返回的指针是根节点root的bt指针为止，此时，二叉树构造完毕

由于在构造一个结点时，它的父结点必须已经被构造，所以应该采用前序遍历的方式，即根左右的方式递归构造\*/

{

DataType ch;

cin>>ch;

if(ch=='#')bt=NULL;//如果输入为‘#’，则树为空，这里只针对<char>类型，如果数据为int类型，可以通过判断输入是否为0等来进行空树的输入

else

{

count++;

bt=new BiNode<DataType>;

bt->data=ch;//根节点

cout<<"请输入结点"<<ch<<"所对应的左孩子结点的值:";

bt->lchild=Create(bt->lchild);//递归建立左子树

cout<<"请输入结点"<<ch<<"所对应的右孩子结点的值:";

bt->rchild=Create(bt->rchild);//递归建立右子树

}

return bt;

}

template<class DataType>

void BiTree<DataType>::Release(BiNode<DataType> \*bt)

/\*这个函数和上面的构造函数机理相似，只不过没有递归返回那步，由于在释放一个结点时，它的左右孩子结点必须也被释放，所以采用后序遍历，即左右根的方式释放结点\*/

{

if(bt!=NULL)

{

Release(bt->lchild);//递归删除左子树

Release(bt->rchild);//递归删除右子树

delete bt;//删除根节点

}

}

template<class DataType>

void BiTree<DataType>::PreOrder(BiNode<DataType> \*bt)//前序遍历函数

/\*这个函数和上面的构造函数机理相似，它是顺序为根左右的递归输出元素，\*/

{

if(bt==NULL)return;//如果bt为空就返回

else

{

cout<<bt->data;//根

PreOrder(bt->lchild);//向根的左孩子递归

PreOrder(bt->rchild);//向根的右孩子递归

}

}

template<class DataType>

void BiTree<DataType>::InOrder(BiNode<DataType> \*bt)//中序遍历函数

/\*这个函数和上面的构造函数机理相似，它是顺序为左根右的递归输出元素，\*/

{

if(bt==NULL)return;//如果bt为空就返回

else

{

InOrder(bt->lchild);//向根的左孩子递归

cout<<bt->data;//根

InOrder(bt->rchild);//向根的右孩子递归

}

}

template<class DataType>

void BiTree<DataType>::PostOrder(BiNode<DataType> \*bt)//后序遍历函数

/\*这个函数和上面的构造函数机理相似，它是顺序为左右根的递归输出元素，\*/

{

if(bt==NULL)return;//如果bt为空就返回

else

{

PostOrder(bt->lchild);//向根的左孩子递归

PostOrder(bt->rchild);//向根的右孩子递归

cout<<bt->data;//根

}

}

template<class DataType>

void BiTree<DataType>::LeverOrder()

{

cout<<"层序遍历二叉树为:";

int front=-1,rear=-1;//初始化一个队列

BiNode<DataType>\* \*Queue=new BiNode<DataType>\* [count];//创建一个数据类型为 BiNode指针型 的数组Queue，这个数组即为队列

BiNode<DataType>\* q;//创建一个数据类型为BiNode指针型的变量q

if(root==NULL)return;//如果根节点为空，退出函数

Queue[++rear]=root;//根结点入队

while(front!=rear)//当队列非空时

{

q=Queue[++front];

cout<<q->data;

if(q->lchild!=NULL)Queue[++rear]=q->lchild;

if(q->rchild!=NULL)Queue[++rear]=q->rchild;

}

cout<<endl;

}

int main()

{

BiTree<char> tree;//调用构造函数

cout<<"树中结点个数为:"<<tree.count<<endl;

tree.PreOrder();//前序遍历函数调用

tree.InOrder();//中序遍历函数调用

tree.PostOrder();//后序遍历函数调用

tree.LeverOrder();//层序遍历函数调用

return 0;

}

迪杰斯特拉

#include<stdio.h>

#include<math.h>

const double r=7.5;

const double edge=50;

const double MAX=1000;

int startnum,endnum;

int n;

double mindist[102]={0};

int minstep[102]={0};

double finaldist[102]={0};

int finalstep[102]={0};

int Finalstep[102]={0};

struct point{

double x;

double y;

}points[102];

int end[102]={0};

double graph[102][102]={0};

double min(double a,double b)

{

return a<b?a:b;

}

double min(double a,double b,double c,double d)

{

return min(a,min(b,(min(c,d))));

}

double getdistance(point a,point b)

{

return sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2));

}

void dijkstra(int start)

{

int i,j;

double dist[102];

bool S[102];

int num=1;

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

{

dist[i]=graph[start][i];

S[i]=0;

if(dist[i]!=MAX)

minstep[i]++;

}

dist[start]=0;

S[start]=1;

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

{

double min=MAX;

int k=start;

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

{

if(S[i]==0&&dist[i]!=0&&dist[i]<=min)

{

k=i;

min=dist[i];

}

}

S[k]=1;

mindist[k]=min;

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

if(S[j]==0&&graph[k][j]<MAX)

{

if(dist[j]>dist[k]+graph[k][j])

{

dist[j]=dist[k]+graph[k][j];

minstep[j]=minstep[k]+1;

}

}

}

}

int main()

{

double resultdis;

int resultstep=0;

int Resultstep=0;

point origin={0,0};

int i,j;

double ptdis,egdis;

double d;

while(scanf("%d%lf",&n,&d)!=EOF)

{

points[0].x=points[0].y=0;

int k=0;

int p=0;

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

{

scanf("%lf%lf",&points[i].x,&points[i].y);

}

if(d>=edge-r)

{

printf("%.2lf 1\n",edge-r);

continue;

}

startnum=endnum=0;

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

{

egdis=min(points[i].x+edge,edge-points[i].x,points[i].y+edge,edge-points[i].y);

if(egdis<=d)

{

end[endnum]=i;

endnum++;

}

}

if(endnum==0)

{

printf("can\'t be saved\n");

continue;

}

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

{

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

{

ptdis=getdistance(points[i],points[j]);

if(((i==0||j==0)&&ptdis<=r+d)||ptdis<=d)

graph[i][j]=graph[j][i]=ptdis;

else

graph[i][j]=graph[j][i]=MAX;

}

}

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

minstep[i]=0;

dijkstra(0);

for(j=0;j<endnum;j++)

{

finalstep[k]=minstep[end[j]]+1;

finaldist[k]=mindist[end[j]]+min(points[end[j]].x+edge,edge-points[end[j]].x,points[end[j]].y+edge,edge-points[end[j]].y)-r;

k++;

}

if(k==0)

{

printf("can\'t be saved\n");

}

else

{

resultdis=finaldist[0];

resultstep=finalstep[0];

for(i=0;i<k-1;i++)

{

if(resultdis>finaldist[i+1])

{

resultdis=finaldist[i+1];

resultstep=finalstep[i+1];

}

}

if(resultdis+r>=MAX)

{

printf("can\'t be saved\n");

continue;

}

else

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

if(finaldist[i]==resultdis)

{

Finalstep[p]=finalstep[i];

p++;

}

Resultstep=Finalstep[0];

for(i=0;i<p-1;i++)

{

if(Resultstep>Finalstep[i+1])

{

Resultstep=Finalstep[i+1];

}

}

printf("%.2lf %d\n",resultdis,Resultstep);

}

}

return 0;

}

栈应用

#include <iostream>

#include <stack>

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

using namespace std;

stack<char>number;

char start[10000],end[10000];

int len1,len2;

char output[20000];

int l1,l2;

void dfs(int k)

{

if(l2==len1)

{

for(int i=0;i<2\*len1;i++)

cout<<output[i]<<' ';

cout<<endl;

}

if(l1<len1)

{

number.push(start[l1]);

output[k]='i';

l1++;

dfs(k+1);

l1--;

output[k]='\0';

number.pop();

}

if(!number.empty()&&number.top()==end[l2])

{

l2++;

output[k]='o';

number.pop();

dfs(k+1);

l2--;

number.push(end[l2]);

output[k]=0;

}

}

int main()

{

while(scanf("%s %s",start,end)!=EOF)

{

len1=strlen(start);

len2=strlen(end);

if(len1!=len2)

{

cout<<'['<<endl;

cout<<']'<<endl;

}

else

{

l1=0;

l2=0;

cout<<'['<<endl;

dfs(0);

cout<<']'<<endl;

}

}

return 0;

}

#include<stdio.h>

#include<algorithm>

#include<cstdlib>

using namespace std;

int cmp(const void \*a,const void \*b)

{

return (\*(long long\*)a-\*(long long\*)b)>0?1:-1;//这是从小到大排序，若是从大到小改成： return \*(int \*)b-\*(int \*)a;

}

int n;

int m;

int r;

struct Student{

long long id;

int coursenum;

int tnum;

long long time[100];

}stu[50];

struct Course{

long long id;

int capacity;

int tnum;

long long time[100];

}cour[50];

int main()

{

int i,j,k,stuno,courno,acnum,casenum=0;

long long stuid,courid;

while(scanf("%d%d%d",&n,&m,&r)!=EOF)

{

casenum++;

acnum=0;

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

for(j=0;j<100;j++)

{

stu[i].time[j]=0;

cour[i].time[j]=0;

}

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

scanf("%lld",&stu[i].id);

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

{

scanf("%lld%d%d",&cour[i].id,&cour[i].capacity,&cour[i].tnum);

for(j=0;j<cour[i].tnum;j++)

scanf("%lld",&cour[i].time[j]);

qsort(cour[i].time,cour[i].tnum,sizeof(cour[i].time[0]),cmp);

}

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

{

scanf("%lld%lld",&stuid,&courid);

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

{

if(stu[stuno].id==stuid)

break;

}

for(courno=0;courno<m;courno++)

{

if(cour[courno].id==courid)

break;

}//找到对应的学生和课时

if(cour[courno].capacity==0)

{

//printf("%lld无余量\n",cour[courno].id);

continue;

}

for(j=0,k=0;j<stu[stuno].tnum&&k<cour[courno].tnum;)

{

if(stu[stuno].time[j]==cour[courno].time[k])

break;

if(stu[stuno].time[j]<cour[courno].time[k])

j++;

else if(stu[stuno].time[j]>cour[courno].time[k])

k++;

}

if(j<stu[stuno].tnum&&k<cour[courno].tnum)

{

//printf("%lld %lld时间冲突\n",stu[stuno].id,cour[courno].id);

continue;

}

for(j=stu[stuno].tnum;j<stu[stuno].tnum+cour[courno].tnum;j++)

{

stu[stuno].time[j]=cour[courno].time[j-stu[stuno].tnum];

}

stu[stuno].tnum+=cour[courno].tnum;

qsort(stu[stuno].time,stu[stuno].tnum,sizeof(stu[stuno].time[0]),cmp);

cour[courno].capacity--;

acnum++;

}

printf("Case %d: %d\n",casenum,acnum);

}

return 0;

}