小组成员：李树臻 李鹏辉 李冠润 刘宪铎 杜中驭

目 录

[第2章 线性表 3](#_Toc128043512)

[2.1 顺序表基本操作的实现 3](#_Toc128043513)

[2.2 单链表基本操作的实现 6](#_Toc128043514)

[2.3 单链表原地逆置 11](#_Toc128043515)

[2.4 单链表排序 14](#_Toc128043516)

[2.5 两个有序表合并 18](#_Toc128043517)

[2.6 利用单链表求两个集合的交集 21](#_Toc128043518)

[2.7 利用单链表判断两个集合是否相等 25](#_Toc128043519)

[2.8 删除单链表中的重复元素 29](#_Toc128043520)

[2.9 使用单链表实现一元多项式相加 32](#_Toc128043521)

[第3章 栈和队列 36](#_Toc128043522)

[3.1 顺序栈基本操作的实现 36](#_Toc128043523)

[3.2 链栈基本操作的实现 39](#_Toc128043524)

[3.3 循环队列基本操作的实现 43](#_Toc128043525)

[3.4 链队列的基本操作的实现 46](#_Toc128043526)

[3.5 Hanio塔问题 50](#_Toc128043527)

[3.6 利用顺序栈实现进制转换 52](#_Toc128043528)

[3.7 表达式括号匹配问题 56](#_Toc128043529)

[3.8 后缀表达式求值问题 60](#_Toc128043530)

[3.9 中缀表达式求值问题 64](#_Toc128043531)

[3.10 中缀表达式转换成后缀表达式问题 69](#_Toc128043532)

[第4章 字符串和多维数组 75](#_Toc128043533)

[4.1 字符串的BF匹配算法的实现 75](#_Toc128043534)

[4.2 字符串的KMP匹配算法的实现 76](#_Toc128043535)

[4.3 对称矩阵的压缩存储 78](#_Toc128043536)

[4.4 上三角矩阵的压缩存储 81](#_Toc128043537)

[4.5 下三角矩阵的压缩存储 84](#_Toc128043538)

[4.6 三对角矩阵的压缩存储 86](#_Toc128043539)

[4.7 稀疏矩阵的压缩存储 89](#_Toc128043540)

[4.8 约瑟夫环问题 92](#_Toc128043541)

[4.9 求解矩阵的马鞍点 94](#_Toc128043542)

[4.10 螺旋方阵问题 98](#_Toc128043543)

[4.11 幻方问题 100](#_Toc128043544)

[第5章 树和二叉树 104](#_Toc128043545)

[5.1 二叉链表基本操作的实现 104](#_Toc128043546)

[5.2 二叉树遍历的非递归实现 109](#_Toc128043547)

[5.3 求二叉树的最小深度 115](#_Toc128043548)

[5.4 判断二叉树是否是完全二叉树 117](#_Toc128043549)

[5.5 判断二叉树的结构是否对称 122](#_Toc128043550)

[5.6 判断二叉树是否对称 125](#_Toc128043551)

[5.7 求二叉树中叶子结点个数及结点个数 129](#_Toc128043552)

[5.8 中序线索二叉树的构造 131](#_Toc128043553)

[5.9 哈夫曼算法的实现 134](#_Toc128043554)

[5.10 由前序遍历序列和中序遍历序列构造二叉树 137](#_Toc128043555)

[5.11 由后序遍历序列和中序遍历序列构造二叉树 139](#_Toc128043556)

[5.12 求二叉树第k层的结点个数和叶子结点个数 142](#_Toc128043557)

[5.13 打印二叉树第k层的结点和叶子结点 146](#_Toc128043558)

[5.14 求二叉树最大的结点距离 149](#_Toc128043559)

[第6章 图 153](#_Toc128043560)

[6.1 邻接矩阵存储无向图的实现 154](#_Toc128043561)

[6.2 邻接表存储有向图的实现 156](#_Toc128043562)

[6.3 Prim算法的实现 163](#_Toc128043563)

[6.4 Kruscal算法的实现 165](#_Toc128043564)

[6.5 Dijkstra算法的实现 168](#_Toc128043565)

[6.6 Floyd算法的实现 171](#_Toc128043566)

[6.7 拓扑排序的实现 173](#_Toc128043567)

[6.8 选址问题 176](#_Toc128043568)

[6.9 七巧板涂色问题 180](#_Toc128043569)

[6.10 五叉路口交通灯问题 183](#_Toc128043570)

[6.11 农夫过河问题 186](#_Toc128043571)

[第7章 查找 190](#_Toc128043572)

[7.1 线性查找的实现 191](#_Toc128043573)

[7.2 折半查找的实现 192](#_Toc128043574)

[7.3 二叉排序树基本操作的实现 194](#_Toc128043575)

[7.4 散列查找的实现 199](#_Toc128043576)

[第8章 排序 202](#_Toc128043577)

[8.1 直接插入排序的实现 202](#_Toc128043578)

[8.2 Shell排序的实现 204](#_Toc128043579)

[8.3 冒泡排序的实现 205](#_Toc128043580)

[8.4 快速排序的实现 206](#_Toc128043581)

[8.5 简单选择排序的实现 207](#_Toc128043582)

[8.6 堆排序的实现 209](#_Toc128043583)

[8.7 非递归实现二路归并排序 210](#_Toc128043584)

# 第2章 线性表

## 2.1 顺序表基本操作的实现

说明：实现SeqList类模板，基本操作包括无参构造函数、有参构造函数、求顺序表的表长、按位查找、按值查找、插入、删除、遍历等，并在主函数中对各种操作进行验证。

### 实验代码

* 1. SeqList.h

const int MaxSize = 100; /\*顺序表的最大容量为100\*/

template <class ElemType> /\*定义模板类SeqList\*/

class SeqList{

public:

SeqList(); /\*建立空的顺序表\*/

SeqList(ElemType a[ ], int n); /\*建立一个长度为n的顺序表\*/

~SeqList(); /\*析构函数\*/

int Length(); /\*返回顺序表的表长\*/

ElemType Get(int i); /\*按位查找\*/

int Locate(ElemType x); /\*按值查找\*/

void Insert(int i, ElemType x); /\*插入操作\*/

ElemType Delete(int i); /\*删除操作\*/

void PrintList(); /\*遍历操作\*/

private:

ElemType data[MaxSize]; /\*存放数据元素的数组\*/

int length; /\*顺序表的长度\*/

};

* 1. SeqList.cpp

#include "SeqList.h"

#include <iostream>

using namespace std;

/\*SeqList.h中定义的SeqList类的详细实现\*/

template <class ElemType>

SeqList<ElemType>::SeqList() {

length = 0;

}

template <class ElemType>

SeqList<ElemType>::SeqList(ElemType a[], int n) {

if(n > MaxSize)

throw "参数非法，数组参数的长度超过了顺序表的最大容量";

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

data[i] = a[i];

/\*顺序表的长度为数组参数的长度\*/

length = n;

}

template <class ElemType>

SeqList<ElemType>::~SeqList() {

}

template <class ElemType>

int SeqList<ElemType>::Length() {

return length;

}

template <class ElemType>

ElemType SeqList<ElemType>::Get(int i) {

if(i < 1 || i > length)

throw "查找位置非法，合法位置应在1~length之间";

else

return data[i-1];

}

template <class ElemType>

int SeqList<ElemType>::Locate(ElemType x) {

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

if(data[i] == x)

return i+1; /\*若data[i]=x，返回其逻辑位置i+1\*/

return 0; /\*查找失败，返回0\*/

}

template <class ElemType>

void SeqList<ElemType>::Insert(int i, ElemType x) {

if (length >= MaxSize)

throw "顺序表已满，不能再插入元素，上溢";

if (i < 1|| i > length+1)

throw "插入位置不合法，合法的插入位置应为1~length+1";

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

data[j] = data[j-1]; /\*元素data[i-1]...data[length-1]后移，方向从后向前\*/

data[i-1] = x; /\*新元素位置\*/

length++;

}

template <class ElemType>

ElemType SeqList<ElemType>::Delete(int i) {

if(length == 0)

throw "顺序表为空，不能删除元素，下溢";

if(i < 1 || i > length)

throw "删除元素的逻辑位置非法，合法位置应为1~length";

ElemType x = data[i-1]; /\*取出位置i的元素\*/

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

data[j-1] = data[j]; /\*元素data[i]...data[length-1]前移，方向从前向后\*/

length--;

return x;

}

template <class ElemType>

void SeqList<ElemType>::PrintList() {

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

cout<<data[i]<<" "; /\*依次输出线性表的元素值\*/}

cout<<endl;

}

* 1. Main.cpp

#include <iostream>

using namespace std; /\*使用std命名空间，避免命名冲突\*/

#include "SeqList.cpp"

int main() {

int r[10] = {3,6,9,15,17,13,12,15,11,20};

SeqList<int> L(r, 10);

cout<<"执行插入操作前数据为："<<endl;

L.PrintList(); /\*输出所有元素\*/

try{

L.Insert(6, 16);

}

catch(char \*str) {

cout<<str<<endl;

}

cout<<"执行插入操作后数据为："<<endl;

L.PrintList(); /\*输出所有元素\*/

cout<<"值为16的元素位置为:";

cout<<L.Locate(11)<<endl; /\*查找元素11，并返回在顺序表中的位置\*/

cout<<"执行删除第7个元素操作，删除前数据为："<<endl;

L.PrintList(); /\*输出所有元素\*/

try{

L.Delete(9); /\*删除第9个元素\*/

}

catch(char \*str) {

cout<<str<<endl;

}

cout<<"删除后数据为："<<endl;

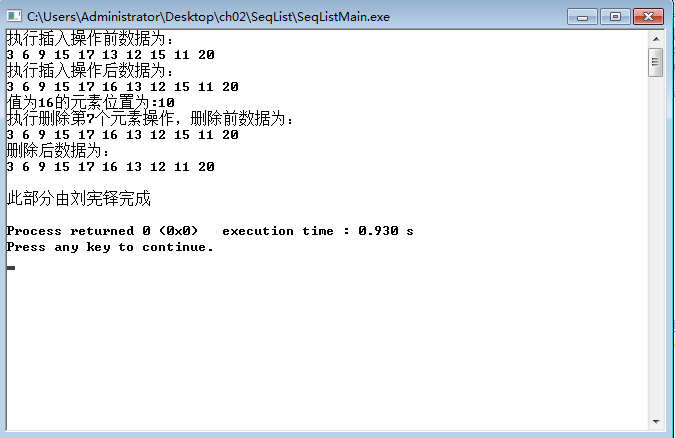
L.PrintList(); /\*输出所有元素\*/

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果



### 遇到的问题及解决方法

1. 顺序表是基本的数据结构，以数组的形式保存，通过前面的学习我们对此较为熟悉，但在本次实训中，需要采用类模板的形式，分为三个文件来编写，组员在操作初期不太适应这种方式，在头文件的书写时经常出错。
2. 通过第一个项目，我对顺序表有了更深刻的理解，注意到在引用一个符号之前，一定要确保它已经声明或者已经定义。
3. 加深记忆顺序表的存储特点：只要确定了起始位置，表中任一元素的地址都通过下列公式得到：LOC（ai）=LOC（a1）+（i-1）\*L （1≤i≤n） 其中，L是元素占用存储单元的长度。

## 2.2 单链表基本操作的实现

说明：实现LinkList类模板，基本操作包括无参构造函数、有参构造函数、析构函数，求单链表的表长，按位查找，按值查找、插入元素、删除元素、遍历整个单链表等，并在主函数中对各操作进行验证，本项目需要使用多文件实现。

### 实验代码

* 1. LinkList.h

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList(); /\*建立只有头结点的空单链表\*/

LinkList(ElemType a[], int n); /\*尾插法建立单链表\*/

~LinkList(); /\*析构函数\*/

int Length(); /\*返回单链表的表长\*/

ElemType Get(int i); /\*按位查找，返回第i个元素的值\*/

int Locate(ElemType x); /\*按值查找\*/

void Insert(int i, ElemType x); /\*插入操作\*/

ElemType Delete(int i); /\*删除操作\*/

void PrintList(); /\*遍历操作\*/

private:

Node<ElemType> \*first; /\*头指针\*/

};

* 1. LinkList.cpp

#include "LinkList.h"

#include <iostream>

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first = new Node<ElemType>; /\*生成头结点\*/

first->next = NULL; /\*头结点的指针域置为空\*/

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[], int n) {

Node<ElemType> \*r, \*s;

first = new Node<ElemType>; /\*生成头结点\*/

r = first; /\*尾指针初始化\*/

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

s = new Node<ElemType>; /\*申请新结点\*/

s->data = a[i]; /\*给新结点的数据域赋值\*/

r->next = s; /\*将新结点插入到单链表末尾\*/

r = s; /\*更新尾指针r\*/

}

r->next=NULL; /\*将终端结点的指针域置空\*/

}

template <class ElemType>

LinkList<ElemType>::~LinkList() {

Node<ElemType> \*q;

while(first != NULL) { /\*当单链表不为空时\*/

q = first; /\*暂存被释放结点\*/

first = first->next; /\*继续处理下一个结点\*/

delete q; /\*释放q\*/

}

}

template <class ElemType>

int LinkList<ElemType>::Length() {

Node<ElemType> \*p = first->next;

int count = 0;

while(p != NULL) {

count++;

p = p->next;

}

return count;

}

template <class ElemType>

ElemType LinkList<ElemType>::Get(int i) {

Node<ElemType> \*p = first->next;

int count = 1;

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

p = p->next;

count++;

}

if(p!=NULL)

return p->data;

else

throw "参数非法";

}

template <class ElemType>

void LinkList<ElemType>::Insert(int i, ElemType x)

{

Node<ElemType> \*p = first,\*s;

int j = 0; /\*工作指针p应指向头结点\*/

while(p != NULL && j < i-1) { /\*查找第i-1个结点\*/

p = p->next; /\*工作指针p后移\*/

j++; /\*计数器加1\*/

}

/\*没有找到第i-1个结点\*/

if (p == NULL) throw "参数不合法，不存在第i-1个结点";

else {

s = new Node<ElemType>; /\*申请新结点\*/

s->data = x; /\*数据域赋值\*/

s->next = p->next;

p->next = s; /\*将结点s插入到结点p之后\*/

}

}

template <class ElemType>

ElemType LinkList<ElemType>::Delete(int i) {

Node<ElemType> \*p,\*q;

ElemType x;

int j = 0; /\*计数器初始化\*/

p = first; /\*注意工作指针p要指向头结点\*/

while(p != NULL && j < i-1) { /\*查找第i-1个结点\*/

p = p->next;

j++;

}

if(p == NULL || p->next == NULL) /\*结点p不存在或p存在但p的后继结点不存在\*/

throw "参数i不合法";

else {

q = p->next;

x = q->data; /\*暂存被删结点的数据域\*/

p->next = q->next; /\*更改p结点的指针域\*/

delete q; /\*释放q\*/

return x;

}

}

template <class ElemType>

int LinkList<ElemType>::Locate(ElemType x) {

Node<ElemType> \*p;

p = first->next; /\*工作指针p初始化\*/

int j = 1; /\*累加器j初始化\*/

while(p != NULL) {

if(p->data == x)

return j; /\*查找成功，返回其序号\*/

p = p->next;

j++;

}

return 0; /\*查找失败，返回0\*/

}

template <class ElemType>

void LinkList<ElemType>::PrintList() {

Node<ElemType> \*p;

p = first->next; /\*工作指针p初始化\*/

while(p != NULL) {

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

p = p->next; /\*工作指针p后移\*/

}

cout<<endl;

}

* 1. MAin.cpp

#include <iostream>

#include "LinkList.cpp"

using namespace std;

int main() {

int r[10] = {3,6,8,9,1,12,16,18,20,19};

LinkList<int> L(r, 10);

cout<<"执行插入操作前的单链表数据为："<<endl;

L.PrintList(); /\*显示单链表中所有的元素\*/

try{

L.Insert(3, 0);

}

catch (char \*str) {

cout<<str<<endl;

}

cout<<"执行插入操作后的单链表数据为："<<endl;

L.PrintList(); /\*显示单链表中所有元素\*/

cout<<"值为18的元素位置为:";

cout<<L.Locate(18)<<endl; /\*查找元素18，并返回在单链表中位置\*/

int length = L.Length();

cout<<"当前单链表的表长为："<<length<<endl;

cout<<"执行删除操作前数据为："<<endl;

L.PrintList(); /\*显示单链表中所有的元素\*/

try{

L.Delete(8); /\*删除第8个元素结点\*/

}

catch (char \*str) {

cout<<str<<endl;

}

cout<<"执行删除操作后数据为："<<endl;

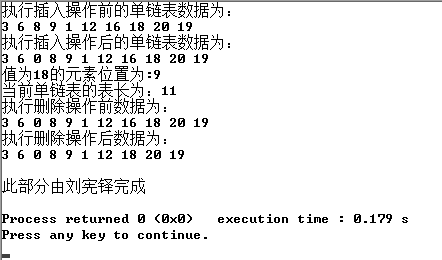
L.PrintList(); /\*显示单链表中所有元素\*/

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果



### 遇到的问题和解决的办法

1. 链表是以节点的方式存储的，各节点不一定连续存放，在操作中对添加节点不够熟悉，选择了直接添加到链表尾部，先创建一个head头结点，作用就是表示单链表的头，后面每添加一个节点，就直接加入到链表的最后。

## 2.3 单链表原地逆置

说明：借助于LinkList类模板，实现单链表的构造、原地逆置、分别遍历逆置前后的单链表。原地逆置指的是不能申请新的结点空间。

### 实验代码

* 1. LinkList.h

#ifndef LINKLIST\_H /\*避免重复引用头文件\*/

#define LINKLIST\_H

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList(); /\*建立只有头结点的空单链表\*/

LinkList(ElemType a[],int n); /\*尾插法建立单链表\*/

~LinkList(); /\*析构函数\*/

void Reverse(); /\*逆置操作\*/

void PrintList(); /\*遍历操作\*/

private:

Node<ElemType> \*first; /\*单链表的头指针\*/

};

#endif

* 1. LinkList.cpp

#include "LinkList.h"

#include<iostream>

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first = new Node<ElemType>; /\*生成头结点\*/

first->next = NULL; /\*头结点的指针域置为空\*/

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[], int n) {

Node<ElemType> \*r, \*s;

first = new Node<ElemType>; /\*生成头结点\*/

r = first; /\*尾指针初始化\*/

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

s = new Node<ElemType>; /\*申请新结点\*/

s->data = a[i]; /\*给新结点的数据域赋值\*/

r->next = s; /\*将新结点插入到单链表末尾\*/

r = s; /\*更新尾指针r\*/

}

r->next = NULL; /\*将终端结点的指针域置空\*/

}

template <class ElemType>

LinkList<ElemType>::~LinkList() {

Node<ElemType> \*q;

while(first != NULL) { /\*当单链表不为空时\*/

q = first; /\*暂存被释放结点\*/

first = first->next; /\*继续处理下一个结点\*/

delete q; /\*释放q\*/

}

}

template <class ElemType>

void LinkList<ElemType>::Reverse() {

Node<ElemType> \*p,\*q;

p = first->next;

first->next = NULL;

while(p != NULL) {

q = p->next;

p->next = first->next;

first->next = p;

p = q;

}

}

template <class ElemType>

void LinkList<ElemType>::PrintList( ) {

Node<ElemType> \*p;

p = first->next; /\*工作指针p初始化\*/

while(p != NULL) {

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

p = p->next; /\*工作指针p后移\*/

}

cout<<endl;

}

* 1. Main.cpp

#include<iostream>

using namespace std;

#include "LinkList.cpp"

int main() {

int r[10] = {1,2,5,6,7,8,9,11,12,15};

LinkList<int> L(r, 10);

cout<<"执行逆置操作前的单链表数据为："<<endl;

L.PrintList(); /\*显示单链表中所有的元素\*/

/\*逆置\*/

L.Reverse();

cout<<"执行逆置操作后的单链表数据为："<<endl;

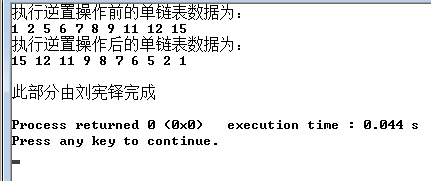
L.PrintList();

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### **实验结果**



### 遇到的问题及解决方法

* 1. 利用模板实现链表操作，这一节中我们的问题较少，加深了对类模板使用的理解，定义自己的类及其成员函数的实现，然后调用。

## 2.4 单链表排序

说明：借助于LinkList类模板，实现单链表的构造、用直接插入法对单链表进行排序、遍历排序前后的单链表。

### 实验代码

1. LinkList.h

#ifndef LINKLIST\_H

#define LINKLIST\_H

template <class ElemType>

struct Node{

ElemType data;

Node \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList();

LinkList(ElemType a[],int n);

~LinkList();

void PrintList();

int IsOrdering(); /\*判断单链表是否升序排序\*/

void Sort(); /\*将单链表按升序排序\*/

private:

Node<ElemType> \*first;

};

#endif

1. LinkList.cpp

#include <iostream>

#include "LinkList.h"

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first = new Node<ElemType>;

first->next = NULL;

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[], int n) {

Node<ElemType> \*r,\*s;

first = new Node<ElemType>;

r = first;

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

s = new Node<ElemType>;

s->data = a[i];

r->next = s;

r = s;

}

r->next = NULL;

}

template <class ElemType>

LinkList<ElemType>::~LinkList() {

Node<ElemType> \*q = NULL;

while(first != NULL) {

q = first;

first = first->next;

delete q;

}

}

template <class ElemType>

void LinkList<ElemType>::PrintList() {

Node<ElemType> \*p = first->next;

while(p != NULL) {

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

p = p->next;

}

cout<<endl;

}

template <class ElemType>

int LinkList<ElemType>::IsOrdering() {

Node<ElemType> \*p,\*q = NULL;

p = first->next;

int flag = 1;

if(p) {

q = p->next;

}

/\*比较相邻的结点是否逆序\*/

while((p != NULL) && (q != NULL)) {

if(p->data > q->data) {

flag = 0;

break;

}

else {

p = q;

q = q->next;

}

}

return flag;

}

template <class ElemType>

void LinkList<ElemType>::Sort() {

/\*使用直接插入排序对单链表进行排序\*/

Node<ElemType> \*p,\*q,\*r,\*s;

p = first->next;

/\*如果单链表非空\*/

if(p != NULL) {

/\*在首元结点后断开\*/

q = p->next;

p->next = NULL;

/\*依次将q插入到有序单链表中\*/

while(q) {

r = q->next;

/\*在单链表中查找结点q的插入位置\*/

s = first;

/\*若s的后继结点存在，并且其数据域小于q的数据域时后移s\*/

while((s->next) && ((s->next->data) < (q->data))) {

s = s->next;

}

/\*将q插入到s之后\*/

q->next = s->next;

s->next = q;

/\*q后移\*/

q = r;

}

}

}

1. Main.cpp

#include<iostream>

using namespace std;

#include "LinkList.cpp"

int main() {

int r1[10] = {3,4,5,6,7,8,9,10,11,12};

LinkList<int> L1(r1, 10);

cout<<"单链表L1为："<<endl;

L1.PrintList(); /\*显示单链表中所有的元素\*/

cout<<"L1是否已有序？";

if(L1.IsOrdering()) {

cout<<"有序"<<endl;

}

else {

cout<<"无序"<<endl;

}

int r2[8]={0,-1,2,-3,4,-5,6,-7};

LinkList<int> L2(r2, 8);

cout<<"单链表L2为："<<endl;

L2.PrintList();

cout<<"L1是否已有序？";

if(L2.IsOrdering()) {

cout<<"有序"<<endl;

}

else {

cout<<"无序"<<endl;

L2.Sort();

cout<<"排序之后的L2为："<<endl;

L2.PrintList();

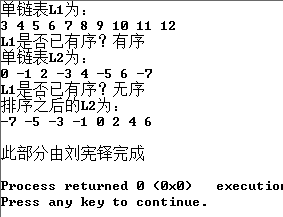
}

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果：



### 遇到的问题及解决方法

1. 单链表的快排序基本思想是基于划分，但单链表不支持基于下标的访问，所以我们把待排序的链表拆分为2个子链表，选择链表的第一个节点作为基准，然后进行比较，比基准小得节点放入左面的子链表，比基准大的放入右边的子链表。在对待排序链表扫描一遍之后，左边子链表的节点值都小于基准的值，右边子链表的值都大于基准的值，然后把基准插入到链表中，并作为连接两个子链表的桥梁。然后分别对左、右两个子链表进行递归快速排序。
2. 在支点的选取上，由于不能随机访问第K个元素，因此每次选择支点时取待排序那部分链表的头指针。
3. 遍历量表方式，由于不能从单链表的末尾向前遍历，因此我们使用两个指针分别向前向后遍历的策略实效。
4. 交换数据方式，直接交换链表数据指针指向的部分，不交换链表节点本身。

## 2.5 两个有序表合并

说明：借助于LinkList类模板，构造两个有序表，将两个有序表合并成一个有序表，分别遍历两个有序表，再遍历合并后的有序表。

### 实验代码

1. LinkList.h

#ifndef LINKLIST\_H /\*避免重复引用头文件\*/

#define LINKLIST\_H

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList(); /\*建立只有头结点的空单链表\*/

LinkList(ElemType a[],int n); /\*尾插法建立单链表\*/

void PrintList(); /\*遍历操作\*/

Node<ElemType> \*GetFirst(); /\*返回头指针\*/

//friend void Merge(LinkList<ElemType> &, LinkList<ElemType> &);

private:

Node<ElemType> \*first; /\*单链表的头指针\*/

};

#endif

1. LinkList.cpp

#include "LinkList.h"

#include<iostream>

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first=new Node<ElemType>; /\*生成头结点\*/

first->next=NULL; /\*头结点的指针域置为空\*/

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[],int n) {

Node<ElemType> \*r, \*s;

first=new Node<ElemType>; /\*生成头结点\*/

r=first; /\*尾指针初始化\*/

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

s=new Node<ElemType>; /\*申请新结点\*/

s->data=a[i]; /\*给新结点的数据域赋值\*/

r->next=s; /\*将新结点插入到单链表末尾\*/

r=s; /\*更新尾指针r\*/

}

r->next=NULL; /\*将终端结点的指针域置空\*/

}

template <class ElemType>

void LinkList<ElemType>::PrintList() {

Node<ElemType> \*p;

p=first->next; /\*工作指针p初始化\*/

while(p!=NULL) {

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

p=p->next; /\*工作指针p后移\*/

}

cout<<endl;

}

template <class ElemType>

Node<ElemType> \*LinkList<ElemType>::GetFirst() {

return first;

}

1. Main.cpp

#include<iostream>

using namespace std;

#include "LinkList.cpp"

template <class ElemType>

void Merge(LinkList<ElemType> &L1, LinkList<ElemType> &L2) {

/\*将单链表L1和L2合并至L1\*/

Node<ElemType> \*first1, \*first2;

first1 = L1.GetFirst(); /\*获取单链表L1的头指针\*/

first2 = L2.GetFirst(); /\*获取单链表L2的头指针\*/

Node<ElemType> \*p,\*q,\*r;

r = first1; /\*单链表L1的尾指针\*/

p = first1->next;

q = first2->next;

while(p != NULL && q != NULL) {

if(p->data <= q->data) {

r->next = p;

r = p;

p = p->next;

}

else {

r->next = q;

r = q;

q = q->next;

}

}

while(p != NULL) {

r->next = p;

r = p;

p = p->next;

}

while(q != NULL) {

r->next = q;

r = q;

q = q->next;

}

r->next = NULL;

}

int main() {

int r1[5] = {2,4,6,8,10};

int r2[8] = {1,3,5,7,9,11,13,15};

/\*第一个单链表的表长为5\*/

LinkList<int> L1(r1, 5);

/\*第二个单链表的表长为8\*/

LinkList<int> L2(r2, 8);

cout<<"第一个单链表为："<<endl;

L1.PrintList();

cout<<"第二个单链表为："<<endl;

L2.PrintList();

cout<<"合并以后的单链表为："<<endl;

Merge(L1, L2);

L1.PrintList();

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

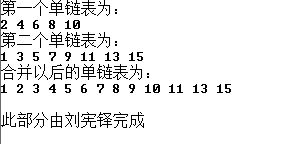
}

两个有序表：

2 , 4 , 6 , 8 , 10

1 , 3 , 5 , 7 , 9 , 11 , 13 , 15

### 实验结果



### 遇到的问题及解决方法

1. 有序表的合并和顺序表的插入方法是类似的，根据前面的经验，我们创建了两个数组，实现两个有序表，逐个将第二个顺序表中的数据与第一个数据表中的数据对比大小，并按大小顺序排列合并，生成最后的表。

## 2.6 利用单链表求两个集合的交集

说明：借助于LinkList类模板，构造两个单链表分别存储两个集合，求两个单链表都包含的元素结点，遍历两个单链表，输出两个单链表都包含的元素结点。

### 实验代码

* 1. LinkList.h

#ifndef LINKLIST\_H /\*避免重复引用头文件\*/

#define LINKLIST\_H

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList(); /\*无参构造函数，建立只有头结点的空单链表\*/

LinkList(ElemType a[], int n); /\*有参构造函数，使用尾插法建立有n个元素结点的单链表\*/

int Locate(ElemType x); /\*在单链表中按值查找，若存在，返回序号\*/

void PrintList(); /\*遍历操作，按序号依次输出各元素\*/

Node<ElemType> \*GetFirst(); /\*返回头指针\*/

private:

Node<ElemType> \*first; /\*单链表的头指针\*/

};

#endif

* 1. LikList.app

#include "LinkList.h"

#include<iostream>

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first = new Node<ElemType>; /\*生成头结点\*/

first->next = NULL; /\*头结点的指针域置为空\*/

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[], int n) {

Node<ElemType> \*r,\*s;

first = new Node<ElemType>; /\*生成头结点\*/

r = first; /\*尾指针初始化\*/

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

s = new Node<ElemType>; /\*申请新结点\*/

s->data = a[i]; /\*给新结点的数据域赋值\*/

r->next = s; /\*将新结点插入到单链表末尾\*/

r = s; /\*更新尾指针r\*/

}

r->next = NULL; /\*将终端结点的指针域置空\*/

}

template <class ElemType>

int LinkList<ElemType>::Locate(ElemType x) {

Node<ElemType> \*p;

p = first->next; /\*工作指针p初始化\*/

int j = 1; /\*累加器j初始化\*/

while(p != NULL) {

if(p->data == x)

return j; /\*查找成功，返回其序号\*/

p = p->next;

j++;

}

return 0; /\*查找失败，返回0\*/

}

template <class ElemType>

void LinkList<ElemType>::PrintList() {

Node<ElemType> \*p;

p = first->next; /\*工作指针p初始化\*/

while(p != NULL) {

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

p = p->next; /\*工作指针p后移\*/

}

cout<<endl;

}

* 1. Main.cpp

#include<iostream>

using namespace std;

#include "LinkList.cpp"

template <class ElemType>

void Intersection(LinkList<ElemType> &L1, LinkList<ElemType> &L2) {

/\*将单链表L1和L2存储的集合求交集并输出\*/

Node<ElemType> \*first1,\*p;

first1 = L1.GetFirst();

p = first1->next;

while(p != NULL) {

/\*在L2中查找p的数据域是否已存在\*/

int i = L2.Locate(p->data);

/\*找到，输出p->data\*/

if(i != 0) {

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

}

p = p->next;

}

cout<<endl;

}

int main() {

int r1[7] = {1, 2, 3, 4, 5, 6, 7};

int r2[8] = {5,6,7,8,9,10,11,12};

/\*第一个单链表的表长为7\*/

LinkList<int> L1(r1, 7);

/\*第二个单链表的表长为8\*/

LinkList<int> L2(r2, 8);

cout<<"第一个集合为："<<endl;

L1.PrintList();

cout<<"第二个集合为："<<endl;

L2.PrintList();

cout<<"两个集合的交集为："<<endl;

Intersection(L1, L2);

char r3[5] = {'A','B','C','D','E'};

char r4[6] = {'C','D','E','F','G','H'};

LinkList<char> L3(r3, 5);

LinkList<char> L4(r4, 6);

cout<<"第三个集合为："<<endl;

L3.PrintList();

cout<<"第四个集合为："<<endl;

L4.PrintList();

cout<<"两个集合的交集为："<<endl;

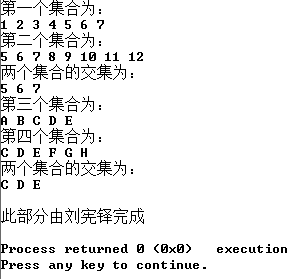
Intersection(L3, L4);

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果



### 遇到的问题及解决方法

（1）SeqList.cpp文件总是报错error: 'cout' was not declared in this scope无法使用cout输出，刚开始使用printf代替，后发现可以使用添加using namespace std;命令解决

## 2.7 利用单链表判断两个集合是否相等

说明：借助于LinkList类模板，构造两个单链表分别存储两个集合，判断两个单链表是否相等。可先将单链表按元素结点值排序，然后再判断所包含的元素值是否相等。分别遍历两个单链表，给出相等及不相等的实验数据。

### 实验代码

* 1. LinkList.h

#ifndef LINKLIST\_H

#define LINKLIST\_H

template <class ElemType>

struct Node{

ElemType data;

Node \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList();

LinkList(ElemType a[], int n);

~LinkList();

void PrintList();

Node<ElemType>\* GetFirst();

private:

Node<ElemType> \*first;

};

#endif

* 1. LinkList.app

#include <iostream>

#include "LinkList.h"

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first = new Node<ElemType>;

first->next = NULL;

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[], int n) {

Node<ElemType> \*r,\*s;

first = new Node<ElemType>;

r = first;

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

s = new Node<ElemType>;

s->data = a[i];

r->next = s;

r = s;

}

r->next = NULL;

}

template <class ElemType>

LinkList<ElemType>::~LinkList() {

Node<ElemType> \*q = NULL;

while(first != NULL) {

q = first;

first = first->next;

delete q;

}

}

template <class ElemType>

Node<ElemType>\* LinkList<ElemType>::GetFirst() {

return first;

}

template <class ElemType>

void LinkList<ElemType>::PrintList() {

Node<ElemType> \*p = first->next;

while(p!=NULL) {

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

p = p->next;

}

cout<<endl;

}

* 1. Main.app

#include <iostream>

using namespace std;

#include "LinkList.cpp"

template <class ElemType>

/\*判断两个集合是否相等，相等返回1，不相等返回0\*/

int SetIsEqual(LinkList<ElemType> &L1, LinkList<ElemType> &L2) {

Node<ElemType> \*p,\*q;

int flag = 1;

p = L1.GetFirst()->next;

q = L2.GetFirst()->next;

/\*两个链表都未结束时判断\*/

while(p != NULL && q != NULL) {

/\*如果数据域相等则继续比较\*/

if(p->data == q->data) {

p = p->next;

q = q->next;

}

/\*相同位置的数据域不相等\*/

else {

flag = 0;

break;

}

}

/\*一个链表结束，另一个链表没结束\*/

/\*并且已比较的部分相等\*/

if((flag == 1) && (p || q)) {

flag = 0;

}

return flag;

}

void outResult(int flag) {

if(flag) {

cout<<"相等";

}

else {

cout<<"不相等";

}

cout<<endl;

}

int main() {

int s1[] = {1,2,3,4,5};

int s2[] = {1,2,3,4,5};

int s3[] = {6,7,8,9,10};

int s4[] = {6,7,8,9,10,11,12,13};

int s5[] = {3,4,5,6,7};

LinkList<int> L1(s1, 5);

LinkList<int> L2(s2, 5);

LinkList<int> L3(s3, 6);

LinkList<int> L4(s4, 8);

LinkList<int> L5(s5, 5);

cout<<"第一个集合：";

L1.PrintList();

cout<<"第二个集合：";

L2.PrintList();

cout<<"第三个集合：";

L3.PrintList();

cout<<"第四个集合：";

L4.PrintList();

cout<<"第五个集合：";

L5.PrintList();

cout<<"第一、二个集合是否相等？";

outResult(SetIsEqual(L1,L2));

cout<<"第三、四个集合是否相等？";

outResult(SetIsEqual(L3,L4));

cout<<"第二、五个集合是否相等？";

outResult(SetIsEqual(L2,L5));

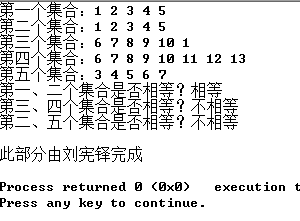
cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果

实验数据可参考下图：



### 遇到的问题及解决方法

1. 初始在编写LinkList.cpp文件时，遇到多次error: 'NULL' was not declared in this scope空值未定义的错误，后通过查询csdn发现LinkLIst.cpp文件只使用了LinkLIst..h做头文件，无法对c++语法中基础定义的值使用，后添加#include <iostream>头文件解决问题

## 2.8 删除单链表中的重复元素

说明：借助于LinkList类模板，构造单链表，将单链表中的所有重复元素删除，有多个值相等的，只保留一个，分别遍历删除前后的单链表。构造单链表的值需要输入。

### 实验代码

1. LinkList.h

#ifndef LINKLIST\_H /\*避免重复引用头文件\*/

#define LINKLIST\_H

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkList{

public:

LinkList(); /\*建立只有头结点的空单链表\*/

LinkList(ElemType a[], int n); /\*尾插法建立单链表\*/

~LinkList(); /\*析构函数\*/

void LinkListDelNormalDup(); /\*删除有普通单链表中的重复元素\*/

void PrintList(); /\*遍历操作\*/

private:

Node<ElemType> \*first; /\*单链表的头指针\*/

};

#endif

1. LinkList.app

#include "LinkList.h"

#include<iostream>

using namespace std;

template <class ElemType>

LinkList<ElemType>::LinkList() {

first = new Node<ElemType>; /\*生成头结点\*/

first->next = NULL; /\*头结点的指针域置为空\*/

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType a[], int n) {

Node<ElemType> \*r, \*s;

first = new Node<ElemType>; /\*生成头结点\*/

r = first; /\*尾指针初始化\*/

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

s = new Node<ElemType>; /\*申请新结点\*/

s->data = a[i]; /\*给新结点的数据域赋值\*/

r->next = s; /\*将新结点插入到单链表末尾\*/

r = s; /\*更新尾指针r\*/

}

r->next = NULL; /\*将终端结点的指针域置空\*/

}

template <class ElemType>

LinkList<ElemType>::~LinkList() {

Node<ElemType> \*q;

while(first != NULL) { /\*当单链表不为空时\*/

q = first; /\*暂存被释放结点\*/

first = first->next; /\*继续处理下一个结点\*/

delete q; /\*释放q\*/

}

}

template <class ElemType>

void LinkList<ElemType>::LinkListDelNormalDup() {

Node<ElemType> \*p,\*q,\*r;

p = first->next;

while(p != NULL) {

q = p;

while(q->next != NULL) {

if(p->data == q->next->data) {

r = q->next;

q->next = r->next;

delete r;

}

else {

q = q->next;

}

}

p = p->next;

}

}

template <class ElemType>

void LinkList<ElemType>::PrintList() {

Node<ElemType> \*p;

p = first->next; /\*工作指针p初始化\*/

while(p != NULL) {

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

p = p->next; /\*工作指针p后移\*/

}

cout<<endl;

}

1. Main.app

#include<iostream>

using namespace std;

#include "LinkList.cpp"

int main() {

int r[10] = {7,6,8,5,2,5,8,5,4,9};

LinkList<int> L(r, 10);

cout<<"单链表原来的数据为："<<endl;

L.PrintList();

L.LinkListDelNormalDup();

cout<<"删除重复的元素之后的单链表数据为："<<endl;

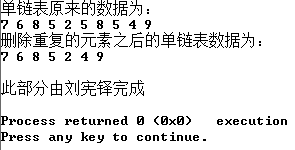
L.PrintList();

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果



### 遇到的问题及解决方法

1. 在单链表表尾，虽然被删节点不存在，但是前去节点存在，仅当被删节点的前去节点存在且p不是终端节点时，才能确定被删节点的存在。
2. 建立三个工作指针p,q,r，然后p遍历全表。p每到一个结点，q就从这个结点往后遍历，并与p的数值比较，相同的话就free掉那个结点。

## 2.9 使用单链表实现一元多项式相加

说明：借助于LinkList类模板，实现两个一元多项式相加。使用单链表分别存储一元多项式，将两个一元多项式相加，分别输出两个一元多项式及合并以后的多项式。所有数据需要输入。需要注意一元多项式的输出格式，例如：A(x)=12X6-7X3-8X2+100，多项式中的指数可放到变量后，最高的指数项前的+号省略，其它项的+号不能省略。

### 实验代码

* 1. Linklistpoly.h

#ifndef LINKLISTPOLY\_H

#define LINKLISTPOLY\_H

struct Node{

float coef; /\*系数\*/

int exp; /\*指数\*/

Node \*next;

};

class LinkListPoly{

public:

LinkListPoly();

LinkListPoly(int c[], int e[], int n);

~LinkListPoly();

void PrintList();

Node \*GetFirst();

private:

Node \*first;

};

#endif

* 1. LinkListPoly.cpp

#include <iostream>

using namespace std;

#include "LinkListPoly.h"

LinkListPoly::LinkListPoly() {

first=new Node;

first->next=NULL;

}

LinkListPoly::LinkListPoly(int c[], int e[], int n) {

Node \*r,\*s;

first = new Node;

r = first;

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

s = new Node;

s->coef = c[i];

s->exp = e[i];

r->next = s;

r = s;

}

r->next = NULL;

}

LinkListPoly::~LinkListPoly() {

Node \*q = NULL;

while(first != NULL) {

q = first;

first = first->next;

delete q;

}

}

Node \*LinkListPoly::GetFirst() {

return first;

}

void LinkListPoly::PrintList() {

Node \*p = first->next;

int count = 0;

while(p != NULL) {

if(count != 0 && p->coef > 0) {

cout<<"+";

}

cout<<p->coef;

if(p->exp != 0) {

cout<<"X";

if(p->exp != 1) {

cout<<p->exp;

}

}

p = p->next;

count++;

}

cout<<endl;

}

* 1. Main.cpp

#include <iostream>

#include "LinkListPoly.h"

using namespace std;

/\*实现一元多项式相加，结果存入LA\*/

void PolyAdd(LinkListPoly &LA, LinkListPoly &LB) {

/\*p指向单链表LA的首元结点\*/

/\*q指向单链表LB的首元结点\*/

Node \*p, \*q;

/\*p\_pre为p的前驱结点\*/

/\*q\_pre为q的前驱结点\*/

Node \*p\_pre,\*q\_pre;

Node \*tmp;

p\_pre = LA.GetFirst();

q\_pre = LB.GetFirst();

p = p\_pre->next;

q = q\_pre->next;

while(p != NULL && q != NULL) {

/\*p后移,q不动\*/

if(p->exp < q->exp) {

p\_pre = p;

p = p->next;

}

/\*p不动，q插入到p之前，p\_pre指向q，q指向原位置的下一个结点\*/

else if(p->exp > q->exp) {

tmp = q->next;

q\_pre->next = q->next;

q->next = p;

p\_pre->next = q;

p\_pre = q;

q = tmp;

}

/\*指数相等，合并结点p和q的系数\*/

else if(p->exp == q->exp) {

p->coef = p->coef + q->coef;

if(p->coef == 0) {

/\*系数为0时，删除p\*/

tmp = p;

p = p->next;

p\_pre->next = p;

delete tmp;

}

/\*删除q\*/

tmp = q;

q = q->next;

q\_pre->next = q;

delete tmp;

}

}

/\*p已为空，如果q不为空，则将q连接到p\_pre的后面\*/

/\*此时p为空，p\_pre不为空\*/

if(q != NULL) {

p\_pre->next = q;

}

}

int main() {

int c1[4] = {1, 2, -100, 4};

int p1[4] = {0, 2, 4, 6};

LinkListPoly LA(c1, p1, 4);

cout<<"第一个多项式为："<<endl;

LA.PrintList();

int c2[6]={7, 20, -9, 5, 30, 40};

int p2[6]={0, 1, 2, 4, 6, 10};

cout<<"第二个多项式为："<<endl;

LinkListPoly LB(c2, p2, 6);

LB.PrintList();

PolyAdd(LA, LB);

cout<<"相加以后的多项式为："<<endl;

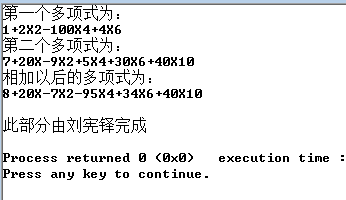
LA.PrintList();

cout<<"\n此部分由刘宪铎完成"<<endl;

return 0;

}

### 实验结果



### 遇到的问题及解决方法

* 1. 初次编译时，在LinkListPoly .cpp文件中出现“multiple definition of... first defined here”问题，过后查询发现是应为多次时嵌套出现多个cpp文件先分别被编译为.o格式的目标文件，两个目标文件再被链接器链接起来，相当于分别进行了一次include，于是就出现了“multiple definition of... first defined here”问题，解决方法取消了多次嵌套，直接均连接LinkListPoly.h文件。
  2. 多项式指数可能很高且变化很大，在表示多项式的线性表中会存在很多零元素。我们选择只储存非零项，在存储非零项系数的同时存储相应的指数。

# 第3章 栈和队列

## 3.1 顺序栈基本操作的实现

说明：实现SeqStack类模板，基本操作包括构造函数、取栈顶、出栈、入栈、判空、判满，并在主函数中验证。

### 实验代码

1. 顺序栈类

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType> /\*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(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop(); /\*返回栈顶元素\*/

int Empty(); /\*判断栈是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

1. 顺序栈操作类

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top == -1)

return 1;

else

return 0;

}

1. 顺序栈操作主函数类

#include <iostream>

using namespace std;

#include "SeqStack.cpp"

int main() {

SeqStack<int> S; /\*创建类实例\*/

if(S.Empty() == 1)

cout<<"初始栈为空"<<endl;

else

cout<<"初始栈不为空"<<endl;

cout<<"8入栈"<<endl;

S.Push(8);

cout<<"9入栈"<<endl;

S.Push(9);

cout<<"10入栈"<<endl;

S.Push(10);

cout<<"栈顶元素为:"<<endl; /\*取栈顶元素\*/

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

cout<<"栈顶出栈"<<endl;

S.Pop(); /\*出栈\*/

cout<<"新的栈顶元素为:"<<endl;

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

cout<<" -杜中驭"<<endl;

return 0;

}

### 实验运行结果

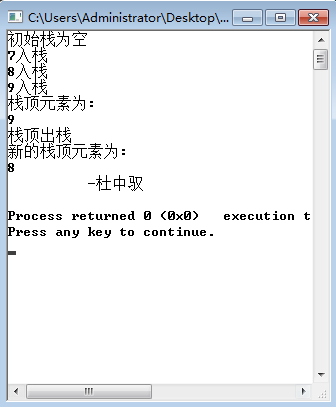


图3-1顺序栈基本操作实现

### 遇到问题及解决方法

1. 执行入栈操作时，容易导致栈溢出，只有当栈的容量足够大时才能避免此情况。

2. 出栈操作时，要确保栈不为空，对栈进行非空判断，否则会出现栈空的情况。

3. 对栈中元素的访问，需要从栈顶开始进行查找，而栈顶元素可能是最先入栈的，因此需要维护一个指针结构p。

## 3.2 链栈基本操作的实现

说明：实现LinkStack类模板，基本操作包括构造函数、析构函数、取栈顶、出栈、入栈、判空，并在主函数中验证。

### 实验代码

链栈基本操作类：

#include "LinkStack.h"

template <class ElemType>

LinkStack<ElemType>::LinkStack() {

top = NULL;

}

template <class ElemType>

LinkStack<ElemType>::~LinkStack() {

Node<ElemType> \*q;

while(top != NULL) {

q = top;

top = top->next;

delete q;

}

}

template <class ElemType>

void LinkStack<ElemType>::Push(ElemType x) {

Node<ElemType> \*s;

s = new Node<ElemType>;

s->data = x;

s->next = top;

top = s;

}

template <class ElemType>

ElemType LinkStack<ElemType>::Pop() {

if(top == NULL)

throw "链栈为空";

Node<ElemType> \*q;

ElemType x;

x = top->data;

q = top;

top = top->next;

delete q;

return x;

}

template <class ElemType>

ElemType LinkStack<ElemType>::GetTop() {

if(top != NULL)

return top->data;

else

throw "栈为空";

}

template <class ElemType>

int LinkStack<ElemType>::Empty() {

if(top == NULL) {

return 1;

}

else {

return 0;

}

}

链表操作主函数：#include <iostream>

using namespace std;

#include "LinkStack.cpp"

int main() {

LinkStack<int> S; /\*创建类实例\*/

if(S.Empty() == 1)

cout<<"初始栈为空"<<endl;

else

cout<<"初始栈不为空"<<endl;

cout<<"7入栈"<<endl;

S.Push(7);

cout<<"8入栈"<<endl;

S.Push(8);

cout<<"9入栈"<<endl;

S.Push(9);

cout<<"栈顶元素为:"<<endl; /\*取栈顶元素\*/

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

cout<<"栈顶出栈"<<endl;

S.Pop(); /\*出栈\*/

cout<<"新的栈顶元素为:"<<endl;

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

cout<<" -杜中驭"<<endl;

return 0;

}

链表类及头文件：

#ifndef LINKSTACK\_H /\*避免重复引用头文件\*/

#define LINKSTACK\_H

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkStack{

public:

LinkStack(); /\*构造函数，初始化空的链栈\*/

~LinkStack(); /\*析构函数，释放链栈中所有的结点\*/

void Push(ElemType x); /\*将x入栈\*/

ElemType Pop(); /\*返回栈顶元素，并出栈\*/

ElemType GetTop(); /\*取栈顶值\*/

int Empty(); /\*判断是否为空栈\*/

private:

Node<ElemType> \*top; /\*栈顶指针\*/

};

#endif

### 实验运行结果

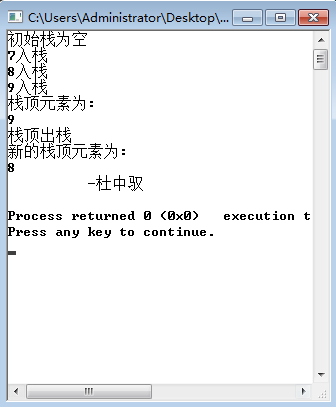


图3-2链栈基本操作的实现

### 遇到问题及解决方法

1. 多个文件互相调用同一个头文件时，会出现重复调用头文件问题，需要加上#ifndef SEQSTACK\_H头文件。
2. 需要理解指针结构p，来灵活对栈进行出栈，入站操作。

## 3.3 循环队列基本操作的实现

说明：实现CirQueue类模板，基本操作包括构造函数、入队、出队、取队头、判空、判满，并在主函数中验证。

### 实验代码

循环队列操作类：

#include <iostream>

using namespace std;

#include "CirQueue.h"

template <class ElemType>

CirQueue<ElemType>::CirQueue() {

front = QueueSize - 1;

rear = QueueSize - 1;

}

template <class ElemType>

CirQueue<ElemType>::~CirQueue() {

}

template <class ElemType>

void CirQueue<ElemType>::EnQueue(ElemType x) {

if((rear + 1) % QueueSize == front)

throw "循环队列已满，上溢！";

rear = (rear + 1) % QueueSize;

data[rear] = x;

}

template <class ElemType>

ElemType CirQueue<ElemType>::DeQueue() {

if(rear == front)

throw "循环队列为空！";

front = (front + 1) % QueueSize;

return data[front];

}

template <class ElemType>

ElemType CirQueue<ElemType>::GetQueue() {

if(rear == front)

throw "循环队列为空！";

return data[(front + 1) % QueueSize];

}

template <class ElemType>

int CirQueue<ElemType>::Length() {

return (rear - front + QueueSize) % QueueSize;

}

template <class ElemType>

int CirQueue<ElemType>::Empty() {

if(front == rear)

return 1;

else

return 0;

}

循环队列操作主函数：

#include <iostream>

using namespace std;

#include "CirQueue.cpp"

int main() {

CirQueue<int> Q; /\*创建类实例\*/

if(Q.Empty() == 1)

cout<<"初始循环队列为空"<<endl;

else

cout<<"初始循环队列不为空"<<endl;

cout<<"7入队"<<endl;

Q.EnQueue(7);

cout<<"8入队"<<endl;

Q.EnQueue(8);

cout<<"9入队"<<endl;

Q.EnQueue(9);

cout<<"10入队"<<endl;

Q.EnQueue(10);

cout<<"队头元素为："<<endl; /\*取队头元素\*/

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

cout<<"队列中的元素个数为："<<endl;

cout<<Q.Length()<<endl; /\*取元素个数\*/

cout<<"队头出队"<<endl;

Q.DeQueue();

cout<<"队列中的元素个数为："<<endl;

cout<<Q.Length()<<endl; /\*取元素个数\*/

cout<<"新的队头元素为："<<endl; /\*取队头元素\*/

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

cout<<" -杜中驭"<<endl;

return 0;

}

循环队列操作中循环队列类定义：

#ifndef CIRQUEUE\_H

#define CIRQUEUE\_H

const int QueueSize = 100; /\*定义循环队列的容量\*/

template <class ElemType> /\*定义模板类CirQueue\*/

class CirQueue{

public:

CirQueue(); /\*构造函数，初始化循环队列\*/

~CirQueue(); /\*析构函数\*/

void EnQueue(ElemType x); /\*入队操作\*/

ElemType DeQueue(); /\*出队操作，返回出队的元素\*/

ElemType GetQueue(); /\*取队头操作\*/

int Length(); /\*返回循环队列的元素个数\*/

int Empty(); /\*判断循环队列是否为空，空返回1，否则返回0\*/

private:

ElemType data[QueueSize]; /\*存放循环队列元素的数组\*/

int front,rear; /\*队头，队尾指针\*/

};

#endif

### 实验运行结果

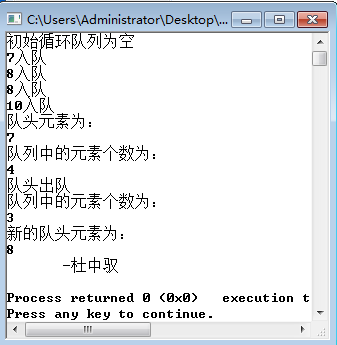


图3-3循环队列基本操作的实现

### 遇到问题及解决方法

1.实现循环队列，需要理解循环，即实现循环主要通过记录头尾指针和指示当前队列容量的引用变量，使循环队列实现循环。

2.如何判断循环队列已满或为空？设置头指针front及尾指针rear，当(rear + 1) % QueueSize =front时，循环队列已满，当rear=front时，循环队列为空。

## 3.4 链队列的基本操作的实现

说明：实现LinkQueue类模板，基本操作包括构造函数、析构函数、入队、出队、取队头、判空，并在主函数中验证。

### 实验代码

链队列操作类：

#include <iostream>

using namespace std;

#include "LinkQueue.h"

template <class ElemType>

LinkQueue<ElemType>::LinkQueue() {

front = new Node<ElemType>;

front->next = NULL;

rear = front;

}

template <class ElemType>

LinkQueue<ElemType>::~LinkQueue() {

Node<ElemType> \*p;

while(front != NULL) {

p = front;

front = front->next;

delete p;

}

rear = NULL;

}

template <class ElemType>

void LinkQueue<ElemType>::EnQueue(ElemType x) {

Node<ElemType> \*s;

s = new Node<ElemType>;

s->data = x;

s->next = NULL;

rear->next = s;

rear = s;

}

template <class ElemType>

ElemType LinkQueue<ElemType>::DeQueue() {

Node<ElemType> \*p;

ElemType x;

if(rear == front)

throw "链队列为空！";

/\*p指向队头\*/

p = front->next;

x = p->data;

front->next = p->next;

/\*队列长度为1时，需要更改rear指针\*/

if(p->next == NULL)

rear = front;

delete p;

return x;

}

template <class ElemType>

ElemType LinkQueue<ElemType>::GetQueue() {

if(front == rear)

throw "链队列为空！";

return front->next->data;

}

template <class ElemType>

int LinkQueue<ElemType>::Empty() {

if(front == rear)

return 1;

else

return 0;

}

链队列操作主函数实现：

#include <iostream>

using namespace std;

#include "LinkQueue.cpp"

int main() {

LinkQueue<int> Q; /\*创建类实例\*/

if(Q.Empty()==1)

cout<<"初始链队列为空"<<endl;

else

cout<<"初始链队列不为空"<<endl;

cout<<"0入队"<<endl;

Q.EnQueue(7);

cout<<"8入队"<<endl;

Q.EnQueue(8);

cout<<"9入队"<<endl;

Q.EnQueue(9);

cout<<"10入队"<<endl;

Q.EnQueue(10);

cout<<"队头元素为："<<endl; /\*取队头元素\*/

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

cout<<"队头出队"<<endl;

Q.DeQueue();

cout<<"新的队头元素为："<<endl; /\*取队头元素\*/

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

cout<<" -杜中驭"<<endl;

return 0;

}

定义链队列类：

#ifndef LINKQUEUE\_H /\*避免重复引用头文件\*/

#define LINKQUEUE\_H

template <class ElemType>

struct Node{

ElemType data;

Node<ElemType> \*next;

};

template <class ElemType>

class LinkQueue{

public:

LinkQueue(); /\*创建只包含一个头结点的链队列\*/

~LinkQueue(); /\*释放链队列中所有的元素结点\*/

void EnQueue(ElemType x); /\*入队操作\*/

ElemType DeQueue(); /\*出队操作\*/

ElemType GetQueue(); /\*取队头\*/

int Empty(); /\*判断链队列是否为空\*/

private:

Node<ElemType> \*front, \*rear; /\*链队列的头指针和尾指针\*/

};

#endif

### 实验运行结果

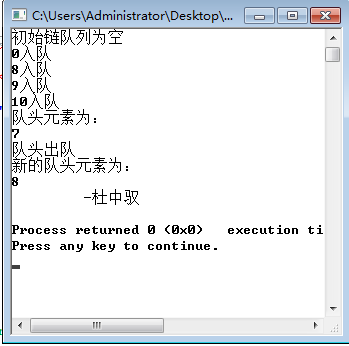


图3-4链队列基本操作的实现

### 遇到问题及解决方法

1. 在链队列中，如何实现元素入队出队操作？新元素的节点入队，令原来的尾结点的next指向新节点，新节点的next指向头结点，更新尾结点指向新节点，效率也比较高。 -出队操作：令头结点的next指向头结点的next指向的节点，然后再令尾结点指向头结点。

## 3.5 Hanio塔问题

说明：递归实现Hanio塔问题，对于n个盘子，给出详细的解决方案，盘子的个数n=3，4，5分别作为测试数据。

### 实验代码

Hanio函数：

#include <iostream>

using namespace std;

void Hanoi(int n, char A, char B, char C) {

if(n == 1) {

cout<<A<<" 塔移到 "<<C<<" 塔"<<endl;

}

else {

Hanoi(n - 1, A, C, B);

cout<<A<<" 塔移到 "<<C<<" 塔"<<endl;

Hanoi(n - 1, B, A, C);

}

}

int main() {

int n;

cout<<"请输入A塔上金碟的个数：";

cin>>n;

if(n<1)

throw "参数不合法，n必须是大于0的整数！";

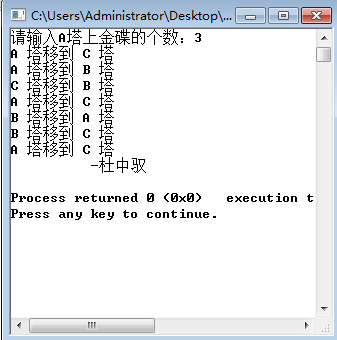
Hanoi(n, 'A', 'B', 'C');

cout<<" -杜中驭"<<endl;

return 0;

}

### 实验运行结果





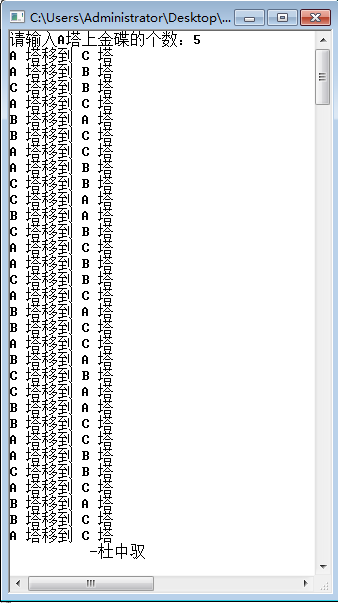


图3-5Hanio塔问题

### **遇到问题及解决方法**

应如何减少移动次数? 解决方法：利用递归算法将要移动的n层汉尼塔看作两个问题，一个是移动n-1层汉尼塔，将最后一个盘子从A柱移动到B柱；另外一个是将n-1层汉尼塔从A柱移动到C柱，此时将最后一个盘子从B柱移动到C柱

## 3.6 利用顺序栈实现进制转换

说明：利用SeqStack类模板，将一个非负的十进制数转换成2-16进制之间的数。输入十进制数，以及转换以后的进制，例如十进制数200，转换成16进制数。需要考虑转换为大于10进制的数时，大于等于10的数如何转化成A-F，例如转换成16进制时，如何将10-15对应转换为A-F。

### 实验代码

顺序表操作类：

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top == -1)

return 1;

else

return 0;

}

顺序表表类：

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType> /\*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(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop(); /\*返回栈顶元素\*/

int Empty(); /\*判断栈是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

进制转换主函数：

#include <iostream>

using namespace std;

#include "SeqStack.cpp"

void convert(int N, int n) {

SeqStack<int> S;

while(N != 0) {

S.Push(N % n);

N = N / n;

}

int e;

while(!S.Empty()) {

e = S.Pop();

if(e > 9) {

/\*当余数大于9时，用大写字母表示\*/

e = e + 55;

cout<<(char)e;

}

else

cout<<e;

}

}

int main() {

int n,N;

cout<<"请输入需要转化进制的十进制数N(N>0)：";

cin>>N;

if(N <= 0) {

cout<<"N必须大于0!"<<endl;

return 0;

}

cout<<"请输入想要转化的进制(2...16)：";

cin>>n;

if(n < 2 || n > 16) {

cout<<"请输入2...16的整数！"<<endl;

return 0;

}

cout<<"转化后的结果为：";

convert(N, n);

cout<<endl;

cout<<" -杜中驭"<<endl;

return 0;

}

### 实验运行结果

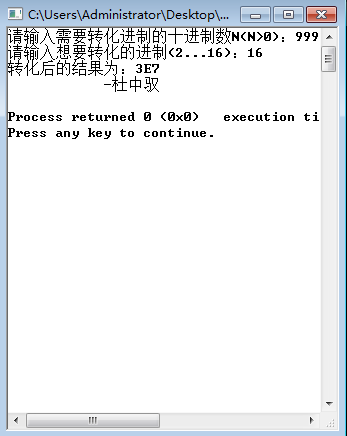


图3-6利用顺序栈实现进制转换

### 遇到问题及解决方法

1. 识别操作符：需要确定每个位置的字符是操作符还是数字。
2. 优先级处理：在求解中缀表达式时，通常需要根据不同操作符的优先级来调整求解顺序，这需要已经预先定义好每种操作符的优先级与结合规则。

3、运算顺序：在求解中缀表达式时，也需要考虑操作符之间的结合顺序，如加减是从左往右的结合，乘除是从右往左的结合。

4、括号处理：将括号包含的部分作为一个整体处理，需要考虑括号的嵌套情况以及不合法括号包含情况

## 3.7 表达式括号匹配问题

说明：利用SeqStack类模板，实现中缀表达式中括号的匹配问题。假设表达式中的括号只包括(、[、{、}、]、)三种括号，输入表达式，给出匹配或不匹配的结果。

### 实验代码

顺序栈操作类：

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top == -1)

return 1;

else

return 0;

}

顺序栈类定义：

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType> /\*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(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop(); /\*返回栈顶元素\*/

int Empty(); /\*判断栈是否为空，若为空返回1,否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

括号匹配主函数：

#include "SeqStack.cpp"

#include <iostream>

#include <string.h>

using namespace std;

/\*中缀表达式中包括{}、[]、()，以'#'结束，判断各种括号是否匹配\*/

int IsBracketMatching() {

SeqStack<char> S; /\*初始化空顺序栈\*/

int i = 0;

char c;

c = getchar();

while(c != '#') {

switch(c) {

case '{':

case '[':

case '(':

S.Push(c);

break;

case '}':

if(S.Pop() != '{')

return 0;

break;

case ']':

if(S.Pop() != '[')

return 0;

break;

case ')':

if(S.Pop() != '(')

return 0;

break;

default:

break;

}

c = getchar();

}

if(!S.Empty()) {

return 0;

}

return 1;

}

int main() {

cout<<"请输入表达式（以'#'结束）："<<endl;

int r = IsBracketMatching();

if(!r) {

cout<<"括号不匹配！"<<endl;

}

else {

cout<<"括号匹配！"<<endl;

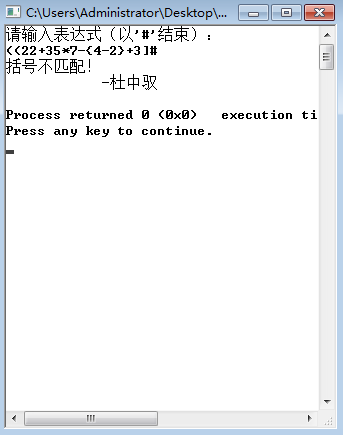
}

cout<<" -杜中驭"<<endl;

return 0;

}

### 实验运行结果



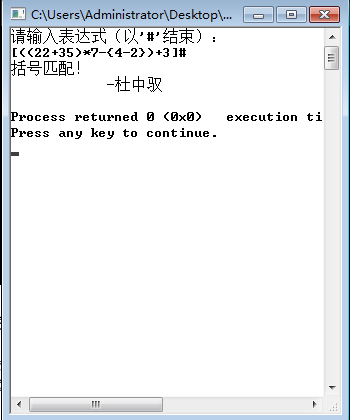


图3-7表达式括号匹配问题

### 遇到问题及解决方法

问题：如何实现括号匹配？  
将最后遇到的“（”，“【”，“{“放入顺序栈中，最后一个为栈顶，再匹配“}”,”】”,”）”,若“}”,”】”,”）”全部与栈顶元素一一匹配，则匹配成功。

## 3.8 后缀表达式求值问题

说明：利用SeqStack类模板，实现后缀表达式求值。假设表达式中只包括+、-、\*、/四种运算。分别考虑操作数为一位整数，操作数为多位整数的不同情况。输入后缀表达式，输出表达式结果。

### 实验代码

顺序表操作类：

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top == -1)

return 1;

else

return 0;

}

顺序表类：

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType> /\*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(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop(); /\*返回栈顶元素\*/

int Empty(); /\*判断栈是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

后缀表达式求值函数：  
#include <iostream>

using namespace std;

#include "SeqStack.cpp"

float PostExpression(string postexp) {

SeqStack<float> S; /\*初始化空顺序栈\*/

int i = 0;

float a, b;

while(postexp[i] != '\0') {

switch(postexp[i]) {

/\*运算符+ \*/

case '+':

a = S.Pop();

b = S.Pop();

S.Push(a + b);

break;

/\*运算符- \*/

case '-':

a = S.Pop();

b = S.Pop();

S.Push(b - a);

break;

/\*运算符\* \*/

case '\*':

a = S.Pop();

b = S.Pop();

S.Push(a \* b);

break;

/\*运算符/ \*/

case '/':

a = S.Pop();

b = S.Pop();

if(a != 0) {

S.Push(b / a);

}

else {

throw "除零错误！";

}

break;

default:

/\*处理数字字符\*/

float d = 0;

while(postexp[i] >= '0' && postexp[i] <= '9') {

d = 10 \* d + postexp[i] - '0';

i++;

}

S.Push(d);

break;

}

i++;

}

return S.GetTop();

}

int main() {

cout<<"请输入表达式"<<endl;

string exp ;//= "89#60#-12#8#-\*";

cin>>exp;

float result = PostExpression(exp);

cout<<result;

cout<<" -杜中驭"<<endl;

return 0;

}实验数据可参考：

89#60#-12#8#-\*

输出结果为：

116

### 实验运行结果

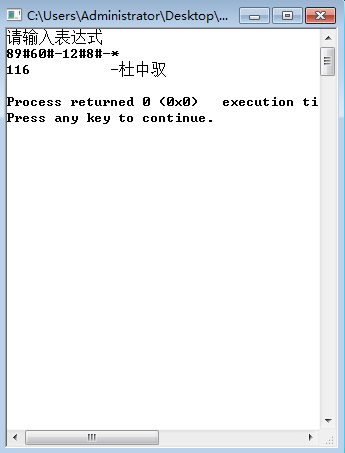


图3-8后缀表达式求值问题

### 遇见问题及解决方法

问题：如何将数字与运算符匹配进行运算？

解决方法：设置顺序栈，当扫描到数字时，将数字入栈，若扫描到运算符，取栈顶两个数字进行运算，这样最后栈内只存有一个数字，即运算结果。

## 3.9 中缀表达式求值问题

说明：利用栈求解中缀表达式，假设表达式中只包括+、-、\*、/四种运算。操作数为一位整数或者多位整数。输入中缀表达式，输出表达式结果。难点在于判断各种运算符的优先级次序。

### 实验代码

顺序表操作类：

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top == -1)

return 1;

else

return 0;

}

顺序表类：

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType> /\*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(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop(); /\*返回栈顶元素\*/

int Empty(); /\*判断栈是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

中缀表达式求值函数：

#include "SeqStack.cpp"

#include <iostream>

#include <stdlib.h>

#include <string.h>

#define OperaterSetSize 7

using namespace std;

char OperaterSet[OperaterSetSize] = {'+', '-', '\*', '/', '(', ')', '#'};

/\*运算符间的优先关系\*/

unsigned char Prior[7][7] = {

'>', '>', '<', '<', '<', '>', '>',

'>', '>', '<', '<', '<', '>', '>',

'>', '>', '>', '>', '<', '>', '>',

'>', '>', '>', '>', '<', '>', '>',

'<', '<', '<', '<', '<', '=', ' ',

'>', '>', '>', '>', ' ', '>', '>',

'<', '<', '<', '<', '<', ' ', '='

};

/\*判断c是否是运算符\*/

int IsOperator(char c) {

int flag = 0;

for(int i = 0; i < OperaterSetSize; i++) {

if(c == OperaterSet[i]) {

flag = 1;

break;

}

}

return flag;

}

/\*返回运算符oper在运算符数组中的序号\*/

int ReturnOpOrd(char oper) {

for(int i = 0; i < OperaterSetSize; i++) {

if (oper == OperaterSet[i]) {

return i;

}

}

return -1;

}

/\*比较两个运算符的优先级，返回字符>,<,=\*/

char Priority(char c1, char c2) {

int i,j;

i = ReturnOpOrd(c1);

j = ReturnOpOrd(c2);

return Prior[i][j];

}

/\*符号运算函数，只有+,-,\*,/ \*/

double Operate(double a, unsigned char c, double b) {

switch(c) {

case '+':

return a + b;

case '-':

return a - b;

case '\*':

return a \* b;

case '/':

return a / b;

default:

return 0;

}

}

/\*算术表达式求值的算符优先算法\*/

float EvaluateInExpression() {

SeqStack<char> OPTR; /\*运算符栈\*/

SeqStack<float> OPND; /\*操作数栈\*/

char tmp[20]; /\*临时数据，用于将数字字符串转化成整数数值\*/

float data, a, b;

char oper, c, cton[2];

OPTR.Push('#');

/\*将tmp置为空\*/

strcpy(tmp, "\0");

cin>>c;

while (c != '#' || OPTR.GetTop() != '#') {

/\*c是操作数\*/

if(!IsOperator(c)) {

cton[0] = c;

cton[1] = '\0'; /\*存放单个数\*/

strcat(tmp, cton); /\*将单个数连到tmp中，形成字符串\*/

cin>>c;

/\*如果遇到运算符，则将字符串tmp转换成浮点数，入栈，并重新置空\*/

if(IsOperator(c)) {

data = (float)atof(tmp);

OPND.Push(data);

strcpy(tmp, "\0");

}

}

/\*c是运算符\*/

else {

switch(Priority(OPTR.GetTop(), c)) {

case '<': /\*栈顶元素优先权低\*/

OPTR.Push(c);

cin>>c;

break;

case '=': /\*脱括号并接收下一字符\*/

OPTR.Pop();

cin>>c;

break;

case '>': /\*退栈并将运算结果入栈\*/

oper = OPTR.Pop();

b = OPND.Pop();

a = OPND.Pop();

OPND.Push(Operate(a, oper, b));

break;

default:

break;

}

}

}

return OPND.GetTop();

}

int main() {

cout<<"请输入中缀表达式：（以'#'结束）："<<endl;

cout<<EvaluateInExpression()<<endl;

cout<<" -杜中驭"<<endl;

}

实验数据可参考：

(89-60)\*(12-8)#

输出结果：

116

### **实验运行结果**

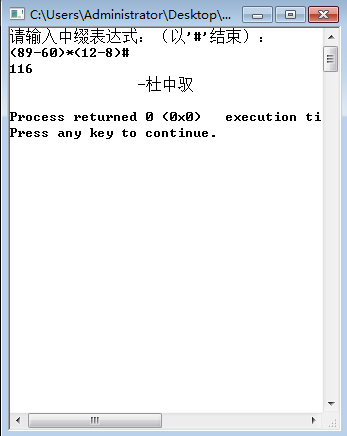


图3-9中缀表达式求值问题

### 遇见问题及解决方法

问题：中缀表达式的计算如何判定运算数字的先后顺序？

解决方法：将第一个数字入栈后，将其转化为浮点数，当检测到运算符后，计算该运算符与上一个运算符优先级，若运算符优先级高于栈顶运算符，则与栈顶数字进行运算，若运算符优先级相同，则去括号，若运算符优先级低，则接受下一字符。

## 3.10 中缀表达式转换成后缀表达式问题

说明：利用栈将中缀表达式转换成后缀表达式，假设表达式中只包括+、-、\*、/四种运算。操作数为一位整数或者多位整数。输入中缀表达式，输出其对应的后缀表达式。

### 实验代码

顺序表操作类：

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top==-1)

return 1;

else

return 0;

}

顺序表类定义：

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType> /\*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(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop(); /\*返回栈顶元素\*/

int Empty(); /\*判断栈是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

中缀表达式转换为后缀表达式主函数：

#include "SeqStack.cpp"

#include <iostream>

#include <stdlib.h>

#include <string.h>

#define OperaterSetSize 7

using namespace std;

char OperaterSet[OperaterSetSize] = {'+', '-', '\*', '/', '(', ')', '#'};

/\*运算符间的优先关系\*/

unsigned char Prior[7][7] = {

'>', '>', '<', '<', '<', '>', '>',

'>', '>', '<', '<', '<', '>', '>',

'>', '>', '>', '>', '<', '>', '>',

'>', '>', '>', '>', '<', '>', '>',

'<', '<', '<', '<', '<', '=', ' ',

'>', '>', '>', '>', ' ', '>', '>',

'<', '<', '<', '<', '<', ' ', '='

};

/\*判断c是否是运算符\*/

int IsOperator(char c) {

int flag = 0;

for(int i = 0; i < OperaterSetSize; i++) {

if(c == OperaterSet[i]) {

flag = 1;

break;

}

}

return flag;

}

/\*返回运算符oper在运算符数组中的序号\*/

int ReturnOpOrd(char oper) {

for(int i = 0; i < OperaterSetSize; i++) {

if (oper == OperaterSet[i]) {

return i;

}

}

return -1;

}

/\*比较两个运算符的优先级，返回字符>,<,=\*/

char Priority(char c1, char c2) {

int i,j;

i = ReturnOpOrd(c1);

j = ReturnOpOrd(c2);

return Prior[i][j];

}

/\*符号运算函数，只有+,-,\*,/ \*/

double Operate(double a, unsigned char c, double b) {

switch (c) {

case '+':

return a + b;

case '-':

return a - b;

case '\*':

return a \* b;

case '/':

return a / b;

default:

return 0;

}

}

/\*算术表达式求值的算符优先算法\*/

void InToPostfix() {

SeqStack<char> OPTR;

OPTR.Push('#');

char c, oper, cton[2];

char tmp[10];

float data;

cin>>c;

/\*将tmp置为空\*/

strcpy(tmp, "\0");

while (c != '#' || OPTR.GetTop() != '#') {

/\*c是操作数,将操作数用#分隔并输出\*/

if(!IsOperator(c)) {

cton[0] = c;

cton[1] = '\0'; /\*存放单个数\*/

strcat(tmp, cton); /\*将单个数连到tmp中，形成字符串\*/

cin>>c;

/\*如果遇到运算符，则将字符串tmp转换成浮点数，入栈，并重新置空\*/

if(IsOperator(c)) {

data = (float)atof(tmp);

cout<<data;

cout<<"#";

strcpy(tmp, "\0");

}

}

/\*c是运算符\*/

else {

switch(Priority(OPTR.GetTop(), c)) {

case '<': /\*栈顶元素优先权低\*/

OPTR.Push(c);

cin>>c;

break;

case '=': /\*脱括号并接收下一字符\*/

OPTR.Pop();

cin>>c;

break;

case '>': /\*退栈并输出，继续处理当前字符\*/

oper = OPTR.Pop();

cout<<oper;

break;

default:

break;

}

}

}

}

int main() {

cout<<"请输入中缀表达式（以'#'结束）："<<endl;

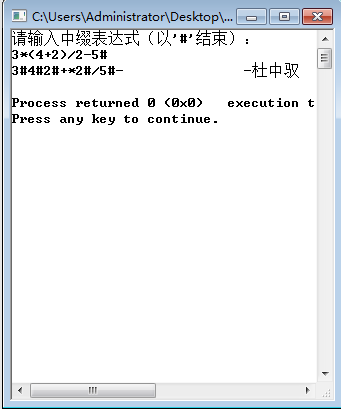
InToPostfix();

cout<<" -杜中驭"<<endl;

return 0;

}

### 实验运行结果



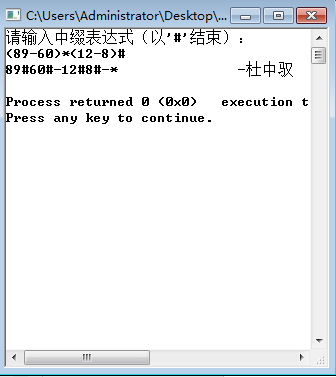


图3-10中缀表达式转化为后缀表达式

### 遇到问题及解决方法：

1.使用getchar（）输入字符时，需使用c头文件，c++头文件中不含getchar()函数。  
2.需要计算每个运算符优先级并进行比较，并把给定的字符串进行字符串转化，转化为浮点数。

# 第4章 字符串和多维数组

## 4.1 字符串的BF匹配算法的实现

说明：使用字符数组存储字符串S和T，输入S和T，使用BF算法进行模式匹配，输出T在S中第一次出现的逻辑位置。

### 实验代码

#include<iostream>

using namespace std;

int BF(char s[], char t[]);

int main()

{

char TargetString[1005], PatternString[1005];

cout << "Input the target string." << endl;

cin >> TargetString;

cout << "Input the pattern string." << endl;

cin >> PatternString;

int ans = BF(TargetString, PatternString);

cout << "The first occurrence of " << PatternString << " in " << TargetString << " is " << ans << endl;

cout << "By 李冠润" << endl;

return 0;

}

int BF(char s[], char t[])

{

int i = 0, j = 0;

while (s[i] != '\0' && t[j] != '\0')

{

if(s[i] == t[j])

{

i++;

j++;

}

else

{

i = i - j + 1;

j = 0;

}

}

if(t[j] == '\0')

return i - j + 1;

else

return 0;

}

### 实验结果

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## 4.2 字符串的KMP匹配算法的实现

说明：使用字符数组存储字符串S和T，输入S和T，使用KMP算法进行模式匹配，输出T在S中第一次出现的逻辑位置。

### 实验代码

#include <iostream>

#include <cstring>

using namespace std;

void GetNext(string T, int next[]);

int KMP(string s, string t);

int main()

{

string TargetString, PatternString;

cout << "Input the target string." << endl;

cin >> TargetString;

cout << "Input the pattern string." << endl;

cin >> PatternString;

int ans = 0;

ans = KMP(TargetString, PatternString);

if(ans)

cout << "The first occurrence of " << PatternString << " in " << TargetString << " is " << ans << endl;

else

cout << "No " << PatternString << " in " << TargetString << endl;

cout << "By 李冠润" << endl;

return 0;

}

void GetNext(string T, int next[])

{

int j = 0, k = -1;

next[0] = -1;

while (j < (int)T.length())

{

if (k == -1 || T[j] == T[k])

{

j++, k++;

if (T[j] == T[k])

next[j] = next[k];

else

next[j] = k;

}

else

k = next[k];

}

}

int KMP(string s, string t)

{

int Next[1005], i = 0, j = 0;

GetNext(t, Next);

while (i < (int)s.length() && j < (int)t.length())

{

if (j == -1 || s[i] == t[j])

i++, j++;

else

j = Next[j];

}

if (j >= (int)t.length())

return i - j + 1;

else

return 0;

}

### 实验结果

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## 4.3 对称矩阵的压缩存储

说明：随机生成nxn阶对称矩阵，将对称矩阵按行优先压缩存下三角的方式压缩存储到一维数组中，根据矩阵的行列下标在一维数组中存取元素。

### 实验代码

#include<iostream>

#include<cstdlib>

#include<ctime>

using namespace std;

#define MAX 105

class Matrix

{

private:

int A[MAX][MAX];

int SA[MAX \* (MAX + 1) / 2];

public:

int n;

Matrix(int n);

void compressMatrix();

int getElement(int i, int j);

void PrintMatrix();

};

Matrix::Matrix(int n)

{

this->n = n;

int a = 100;

int b = 500;

srand((unsigned)time(NULL));

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

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

{

A[i][j] = (rand() % (b - a)) + a;

A[j][i] = A[i][j];

}

}

void Matrix::compressMatrix()

{

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

{

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

{

SA[i \* (i + 1) / 2 + j] = A[i][j];

}

}

}

int Matrix::getElement(int i, int j)

{

if (i < 0 || i >= n || j < 0 || j >= n)

throw "参数非法";

if (i >= j)

return SA[i \* (i + 1) / 2 + j];

else

return SA[j \* (j + 1) / 2 + i];

}

void Matrix::PrintMatrix()

{

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

{

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

{

cout.width(3);

cout << A[i][j] << " ";

}

cout << endl;

}

}

int main()

{

int n;

cout << "Input the number of rows and columns of the matrix" << endl;

cin >> n;

Matrix A(n);

cout << "The constructed random matrix is as follows" << endl;

A.PrintMatrix();

A.compressMatrix();

cout << "Input the coordinates to query" << endl;

int x, y;

cin >> x >> y;

int ans;

ans = A.getElement(x - 1, y - 1);

cout << "The value of " << x << "rows and " << y << " columns is " << ans << endl;

cout << "By 李冠润" << endl;

}

### 实验结果

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## 4.4 上三角矩阵的压缩存储

说明：随机生成nxn阶上三角矩阵，将上三角矩阵按行优先压缩存储的方式存储到一维数组中，根据矩阵的行列下标在一维数组中存取元素。

### 实验代码

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

#define MAX 105

class Matrix

{

private:

int A[MAX][MAX];

int SA[MAX \* (MAX + 1) / 2];

public:

int n;

Matrix(int n);

void compressMatrix();

int getElement(int i, int j);

void PrintMatrix();

};

Matrix::Matrix(int n)

{

this->n = n;

int a = 100;

int b = 500;

srand((unsigned)time(NULL));

int c = (rand() % (b - a)) + a;

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

{

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

A[i][j] = c;

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

A[i][j] = (rand() % (b - a)) + a;

}

}

void Matrix::compressMatrix()

{

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

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

SA[i \* (2 \* n - i + 1) / 2 + j - 1] = A[i][j];

SA[n \* (n + 1) / 2] = A[n - 1][0];

}

int Matrix::getElement(int i, int j)

{

if (i < 0 || i >= n || j < 0 || j >= n)

throw "参数非法";

if (i <= j)

return SA[i \* (2 \* n - i + 1) / 2 + j - i];

else

return SA[n \* (n + 1) / 2];

}

void Matrix::PrintMatrix()

{

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

{

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

{

cout.width(3);

cout << A[i][j] << " ";

}

cout << endl;

}

}

int main()

{

int n;

cout << "Input the number of rows and columns of the matrix" << endl;

cin >> n;

Matrix A(n);

cout << "The constructed random matrix is as follows" << endl;

A.PrintMatrix();

A.compressMatrix();

cout << "Input the coordinates to query" << endl;

int x, y;

cin >> x >> y;

int ans;

ans = A.getElement(x - 1, y - 1);

cout << "The value of " << x << "rows and " << y << " columns is " << ans << endl;

cout << "By 李冠润" << endl;

}

### 实验结果

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## 4.5 下三角矩阵的压缩存储

说明：随机生成nxn阶下三角矩阵，将下三角矩阵按行优先压缩存储的方式存储到一维数组中，根据矩阵的行列下标在一维数组中存取元素。

### 实验代码

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

#define MAX 105

class Matrix

{

private:

int A[MAX][MAX];

int SA[MAX \* (MAX + 1) / 2];

public:

int n;

Matrix(int n);

void compressMatrix();

int getElement(int i, int j);

void PrintMatrix();

};

Matrix::Matrix(int n)

{

this->n = n;

int a = 10;

int b = 50;

srand((unsigned)time(NULL));

int c = (rand() % (b - a)) + a;

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

{

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

A[i][j] = (rand() % (b - a)) + a;

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

A[i][j] = c;

}

}

void Matrix::compressMatrix()

{

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

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

SA[i \* (i + 1) / 2 + j] = A[i][j];

SA[n \* (n + 1) / 2] = A[0][n - 1];

}

int Matrix::getElement(int i, int j)

{

if (i < 0 || i >= n || j < 0 || j >= n)

throw "参数非法";

if (i >= j)

return SA[i \* (i + 1) / 2 + j];

else

return SA[n \* (n + 1) / 2];

}

void Matrix::PrintMatrix()

{

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

{

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

{

cout.width(3);

cout << A[i][j] << " ";

}

cout << endl;

}

}

int main()

{

int n;

cout << "Input the number of rows and columns of the matrix" << endl;

cin >> n;

Matrix A(n);

cout << "The constructed random matrix is as follows" << endl;

A.PrintMatrix();

A.compressMatrix();

cout << "Input the coordinates to query" << endl;

int x, y;

cin >> x >> y;

int ans;

ans = A.getElement(x - 1, y - 1);

cout << "The value of " << x << "rows and " << y << " columns is " << ans << endl;

}

### 实验结果

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## 4.6 三对角矩阵的压缩存储

说明：随机生成nxn阶三对角矩阵，将三对角矩阵按行优先压缩存储的方式存储到一维数组中，根据矩阵的行列下标在一维数组中存取元素。

### 实验代码

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

#define MAX 105

class Matrix

{

private:

int A[MAX][MAX];

int SA[MAX \* (MAX + 1) / 2];

public:

int n;

Matrix(int n);

void compressMatrix();

int getElement(int i, int j);

void PrintMatrix();

};

Matrix::Matrix(int n)

{

this->n = n;

int a = 10;

int b = 50;

srand((unsigned)time(NULL));

int i, j;

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

{

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

{

if (abs(i - j) <= 1)

A[i][j] = (rand() % (b - a)) + a;

else

A[i][j] = 0;

}

}

}

void Matrix::compressMatrix()

{

int k = 0;

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

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

{

if (A[i][j] != 0)

{

SA[k++] = A[i][j];

}

}

}

int Matrix::getElement(int i, int j)

{

if (i < 0 || i >= n || j < 0 || j >= n)

throw "参数非法";

if (abs(i - j) <= 1)

return SA[2 \* i + j];

else

return 0;

}

void Matrix::PrintMatrix()

{

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

{

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

{

cout.width(3);

cout << A[i][j] << " ";

}

cout << endl;

}

}

int main()

{

int n;

cout << "Input the number of rows and columns of the matrix" << endl;

cin >> n;

Matrix A(n);

cout << "The constructed random matrix is as follows" << endl;

A.PrintMatrix();

A.compressMatrix();

cout << "Input the coordinates to query" << endl;

int x, y;

cin >> x >> y;

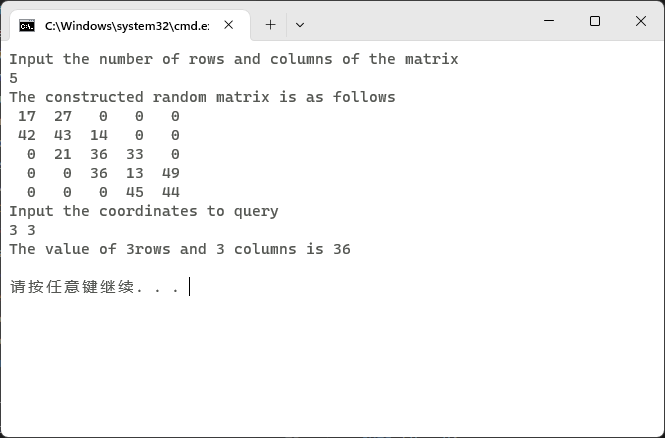
int ans;

ans = A.getElement(x - 1, y - 1);

cout << "The value of " << x << "rows and " << y << " columns is " << ans << endl;

}

### 实验结果



## 4.7 稀疏矩阵的压缩存储

说明：输入nxm阶的稀疏矩阵A，将稀疏矩阵按行优先压缩存储到三元组顺序表中，按照矩阵的行列下标在三元组顺序表中进行元素的存取。分别使用普通转置算法和快速转置算法求矩阵A的转置矩阵B，分别输出A和B的三元组顺序表。

### 实验代码

#include<iostream>

#define MAX 10005

using namespace std;

class Triple

{

public:

int x, y;

int value;

};

class tripleTable

{

private:

Triple data[MAX];

int maxx, maxy, num;

public:

void inputTripleTable();

void printTripleTable();

tripleTable transposeTripleTable();

tripleTable fastTransposeTripleTable();

};

void tripleTable::inputTripleTable()

{

cin >> maxx >> maxy >> num;

for (int i = 1; i <= num; i++)

{

cin >> this->data[i].x >> this->data[i].y >> this->data[i].value;

}

}

void tripleTable::printTripleTable()

{

cout << "The triplet table you entered is " << endl;

for (int i = 1; i <= this->num; i++)

{

cout << this->data[i].x << " " << this->data[i].y << " " << this->data[i].value << endl;

}

}

tripleTable tripleTable::transposeTripleTable()

{

tripleTable after;

after.maxx = this->maxy;

after.maxy = this->maxx;

after.num = this->num;

if(after.num)

{

int q = 1;

for (int i = 1; i <= this->maxy; i++)

for (int j = 1; j <= this->maxx; j++)

if (this->data[j].y == i)

{

after.data[q].x = this->data[j].y;

after.data[q].y = this->data[j].x;

after.data[q].value = this->data[i].value;

q++;

}

}

return after;

}

tripleTable tripleTable::fastTransposeTripleTable()

{

tripleTable after;

after.maxx = this->maxy;

after.maxy = this->maxx;

after.num = this->num;

int cnum[MAX];

int cpot[MAX];

if (after.num)

{

for (int i = 1; i <= this->maxy; i++)

cnum[i] = 0;

for (int i = 1; i <= this->num; i++)

cnum[this->data[i].y]++;

cpot[1] = 1;

for (int i = 2; i <= this->maxx; i++)

cpot[i] = cpot[i - 1] + cnum[i - 1];

for (int i = 1; i <= this->num; i++)

{

int col = this->data[i].y;

int q = cpot[col];

after.data[q].x = this->data[i].y;

after.data[q].y = this->data[i].x;

after.data[q].value = this->data[i].value;

cpot[col]++;

}

}

return after;

}

int main()

{

tripleTable a;

a.inputTripleTable();

a.printTripleTable();

tripleTable transposedTripleTable;

// transposedTripleTable = a.fastTransposeTripleTable();

transposedTripleTable = a.transposeTripleTable();

transposedTripleTable.printTripleTable();

cout << "By 李冠润" << endl;

}

### 实验数据：

6 7 8

1 2 12

1 3 9

3 1 -3

3 6 14

4 3 24

5 2 18

6 1 15

6 4 -7

说明：

第一行三个数分别为稀疏矩阵的行数maxx、列数maxy、非零元素数num；

接下来的num行，每行分别是一个三元组

### 实验结果

图形用户界面

描述已自动生成

## 4.8 约瑟夫环问题

说明：输入n和密码m，使用一维数组实现约瑟夫环问题，输出序列值。

实验数据可参考下图：

### 实验代码

#include<iostream>

using namespace std;

#define MAX 1005

int main()

{

int n, m;

cin >> n >> m;

int r[MAX];

int i = 0, k = 0, j = 0;

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

r[i] = i + 1;

i = 0;

cout << "出列的顺序为" << endl;

while(i < n)

{

if(r[i] != 0)

k++;

if(k == m)

{

k = 0;

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

r[i] = 0;

j++;

}

if(j == n)

break;

i = (i + 1) % n;

}

cout << endl;

cout << "By 李冠润" << endl;

return 0;

}

### 实验数据

9 5

输入的两个数分别为总人数n和密码m。

### 实验结果

图形用户界面, 应用程序, Word

描述已自动生成

## 4.9 求解矩阵的马鞍点

说明：输入nxm阶的矩阵，输出矩阵的马鞍点。注意需要分别使用不存在马鞍点、存在一个马鞍点、存在多个马鞍点三组测试数据，

实验数据可分别参考下列各图：

### 实验代码

#include <iostream>

using namespace std;

#define MAXN 105

int main()

{

int Matrix[MAXN][MAXN];

int m, n, minData, flag = 0;

cin >> m >> n;

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

{

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

{

cin >> Matrix[i][j];

}

}

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

{

minData = Matrix[i][0];

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

{

if (Matrix[i][j] < minData)

minData = Matrix[i][j];

}

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

{

if (Matrix[i][j] == minData)

{

int k;

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

if (Matrix[k][j] > minData)

break;

if (k == m)

{

cout << "The saddle point is " << Matrix[i][j] << " of " << i + 1 << " row and " << j + 1 << " columns" << endl;

flag = 1;

}

}

}

}

if (!flag)

cout << "This matrix has no saddle point" << endl;

cout << "By 李冠润" << endl;

}

### 实验数据与结果

#### 4.9.2.1 数据1-只有一个鞍点

4 4

1 2 3 4

3 4 5 6

5 6 7 8

6 7 8 9

第一行两个数分别为矩阵的行数m和列数n。

接下来的m行，每行n个数，代表矩阵的m\*n个元素。

图形用户界面, 文本, 应用程序, Word

描述已自动生成

#### 4.9.2.2 数据2-不存在鞍点

4 4

1 2 3 4

3 4 5 6

4 5 6 7

9 8 7 6

第一行两个数分别为矩阵的行数m和列数n。

接下来的m行，每行n个数，代表矩阵的m\*n个元素。

图形用户界面, 应用程序, Word

描述已自动生成

#### 4.9.2.3 数据3-有多个鞍点

4 4

1 2 3 4

3 4 5 2

5 5 8 5

4 1 3 3

第一行两个数分别为矩阵的行数m和列数n。

接下来的m行，每行n个数，代表矩阵的m\*n个元素。

图形用户界面, 文本, 应用程序

描述已自动生成

## 4.10 螺旋方阵问题

说明：生成任意nxn阶的螺旋方阵并输出。

### 实验代码

#include<iostream>

using namespace std;

int calculate(int n, int i, int j);

int main()

{

int n;

cout << "Enter the number of rows and columns of the matrix" << endl;

cin >> n;

cout << "The spiral square matrix of order "<< n << " is" << endl;

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

{

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

{

cout.width(5);

cout << calculate(n - 1, i, j);

}

cout << endl;

}

cout << "By 李冠润" << endl;

}

int calculate(int n, int i, int j)

{

int k = 0;

int mini = i < n - i ? i : n - i;

int minj = j < n - j ? j : n - j;

int min = mini < minj ? mini : minj;

int h;

for (h = 0; h < min; ++h)

{

k += (n - 2 \* h) \* 4;

}

if (i == min)

{

k += j - min + 1;

}

else if (j == n - min)

{

k += (n - 2 \* min) + (i - min) + 1;

}

else if (i == n - min)

{

k += (n - 2 \* min) \* 2 + (n - min - j) + 1;

}

else if (j == min)

{

k += (n - 2 \* min) \* 3 + (n - min - i) + 1;

}

return k;

}

### 实验结果

表格

描述已自动生成

## 4.11 幻方问题

说明：生成奇数阶幻方，并输出；生成双偶阶幻方，并输出。

实验数据可分别参考下列各图：

### 生成双偶阶幻方

#include <iostream>

#include <iomanip>

using namespace std;

#define MAXN 105

int judge(int i, int j)

{

int flag = 0;

if (i == j)

return 1;

else if (i > j && (i - j) % 4 == 0)

return 1;

else if (j > i && (j - i) % 4 == 0)

return 1;

else if ((i + j + 1) % 4 == 0)

return 1;

else

return 0;

}

void MagicSquresDoubleEven(int a[][MAXN], int n)

{

int i, j;

int k = 1;

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

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

a[i][j] = k++;

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

{

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

{

if (judge(i, j) == 1)

a[i][j] = n \* n + 1 - a[i][j];

}

}

}

void PrintMagicSquares(int a[][MAXN], int n)

{

int i, j;

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

{

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

{

cout << setw(5) << a[i][j];

}

cout << endl;

}

}

int main()

{

int a[MAXN][MAXN];

int i, j;

int n;

cout << "Enter the order of magic square" << endl;

cin >> n;

MagicSquresDoubleEven(a, n);

cout << "The magic square of order " << n << " is" << endl;

PrintMagicSquares(a, n);

cout << "By 李冠润" << endl;

return 0;

}

图形用户界面, 表格

中度可信度描述已自动生成

### 生成奇数阶幻方

#include <iostream>

#include <iomanip>

#define MAXN 100

using namespace std;

void MagicSquresOdd(int a[][MAXN], int n)

{

int i, j, k;

i = 0;

j = n / 2;

a[i][j] = 1;

for (k = 2; k <= n \* n; k++)

{

int ti, tj;

ti = i;

tj = j;

i = (i - 1 + n) % n;

j = (j - 1 + n) % n;

if (a[i][j] > 0)

{

i = (ti + 1) % n;

j = tj;

}

a[i][j] = k;

}

}

void PrintMagicSquares(int a[][MAXN], int n)

{

int i, j;

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

{

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

{

cout << setw(5) << a[i][j];

}

cout << endl;

}

}

int main()

{

int a[MAXN][MAXN];

int i, j;

int n;

cout << "Enter the order of magic square" << endl;

cin >> n;

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

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

a[i][j] = 0;

MagicSquresOdd(a, n);

cout << "The magic square of order " << n << " is" << endl;

PrintMagicSquares(a, n);

cout << "By 李冠润" << endl;

return 0;

}

图形用户界面, 应用程序

描述已自动生成

# 第5章 树和二叉树

提示：本章所有的实训项目都需要输入，大部分是需要输入结点序列构造二叉链表。

5.10，5.11需要输入序列，类似于字符数组。

## 5.1 二叉链表基本操作的实现

说明：实现类模板BiTree，基本操作包括构造函数、递归前序遍历、递归中序遍历、递归后序遍历、层序遍历，并在主函数中验证，测试数据参照教材图5-19，即下图。



### 试验代码

1. Bitree.h

#ifndef BITREE\_H

#define BITREE\_H

/\*二叉树的结点结构\*/

template <class ElemType>

struct BiNode{

ElemType data;

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

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

/\*析构函数，释放各结点\*/

~BiTree() {

Release(root);

}

/\*前序遍历二叉树\*/

void PreOrder() {

PreOrder(root);

}

/\*中序遍历二叉树\*/

void InOrder() {

InOrder(root);

}

/\*后序遍历二叉树\*/

void PostOrder() {

PostOrder(root);

}

/\*层序遍历二叉树\*/

void LevelOrder() {

LevelOrder(root);

};

private:

BiNode<ElemType> \*root; /\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt); /\*构造函数调用\*/

void Release(BiNode<ElemType> \*bt); /\*析构函数调用\*/

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

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

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

void LevelOrder(BiNode<ElemType> \*bt); /\*层序遍历函数调用\*/

};

#endif

1. Bitree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

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

/\*按照后序遍历的顺序释放二叉树\*/

if(bt != NULL) {

Release(bt->lchild); /\*释放左子树\*/

Release(bt->rchild); /\*释放右子树\*/

delete bt; /\*删除根结点\*/

}

}

template <class ElemType>

void BiTree<ElemType>::PreOrder(BiNode<ElemType> \*bt) {

if(bt == NULL) /\*递归调用的边界条件\*/

return;

else {

cout<<bt->data<<" "; /\*访问根结点\*/

PreOrder(bt->lchild); /\*前序递归遍历左子树\*/

PreOrder(bt->rchild); /\*前序递归遍历右子树\*/

}

}

template <class ElemType>

void BiTree<ElemType>::InOrder(BiNode<ElemType> \*bt) {

if(bt == NULL)

return; /\*递归调用的边界条件\*/

else {

InOrder(bt->lchild); /\*中序递归遍历左子树\*/

cout<<bt->data<<" "; /\*访问根结点\*/

InOrder(bt->rchild); /\*中序递归遍历的右子树\*/

}

}

template <class ElemType>

void BiTree<ElemType>::PostOrder(BiNode<ElemType> \*bt) {

if(bt == NULL)

return; /\*递归调用的边界条件\*/

else {

PostOrder(bt->lchild); /\*后序递归遍历左子树\*/

PostOrder(bt->rchild); /\*后序递归遍历右子树\*/

cout<<bt->data<<" "; /\*访问根结点\*/

}

}

template <class ElemType>

void BiTree<ElemType>::LevelOrder(BiNode<ElemType> \*bt) {

const int MaxSize = 100;

/\*采用顺序队列，假设不会发生溢出\*/

int front = -1, rear = -1;

BiNode<ElemType> \*Q[MaxSize], \*q;

if(bt == NULL)

return;

else {

Q[rear++] = bt; /\*bt入队\*/

/\*队列非空时循环\*/

while(front != rear) {

q = Q[front++]; /\*队头出队\*/

cout<<q->data<<" "; /\*访问队头\*/

if(q->lchild != NULL) /\*如果队头有左孩子，左孩子入队\*/

Q[rear++] = q->lchild;

if(q->rchild != NULL) /\*如果队头有右孩子，右孩子入队\*/

Q[rear++] = q->rchild;

}

}

}

1. Bitree Main.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T;

cout<<"------前序遍历------ "<<endl;

T.PreOrder();

cout<<endl;

cout<<"------中序遍历------ "<<endl;

T.InOrder( );

cout<<endl;

cout<<"------后序遍历------ "<<endl;

T.PostOrder( );

cout<<endl;

cout<<"------层序遍历------ "<<endl;

T.LevelOrder();

cout<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验运行结果

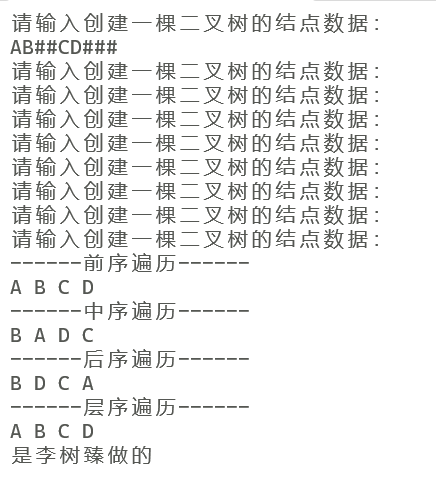


图5-1 二叉链表基本操作的实现

### 遇到的问题和解决的办法

1. 问题：如何判断一个二叉树是否是平衡的？

解决方法：利用递归：判断根节点的左右子树深度之差是否小于等于1(这里需要用到求深度的方法)，如果是，根节点就是平衡的；然后，在判断根节点的左孩子和右孩子是否是平衡的。如此继续下去，直到遇见叶子节点。一旦不是，立刻返回false。

1. 问题：怎样编写一个程序，把一个有序整数数组放到二叉树中？

解决方法：利用递归。

## 5.2 二叉树遍历的非递归实现

说明：实现类模板BiTree，基本操作包括构造函数、非递归前序遍历、非递归中序遍历、非递归后序遍历，并在主函数中验证，测试数据参照教材图5-19，即下图。



### 实验代码

1. SeqStack.h

#ifndef SEQSTACK\_H

#define SEQSTACK\_H

const int StackSize = 100; /\*定义顺序栈的容量\*/

template <class ElemType>

class SeqStack

{

public:

SeqStack(); /\*构造函数，栈的初始化\*/

~SeqStack(); /\*析构函数\*/

void Push(ElemType x); /\*元素x入栈\*/

ElemType Pop(); /\*返回栈顶元素的值并出栈\*/

ElemType GetTop( ); /\*返回栈顶元素\*/

int Empty( ); /\*判断栈是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[StackSize]; /\*存放栈元素的数组\*/

int top; /\*栈顶指针\*/

};

#endif

1. SeqStack.cpp

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class ElemType>

SeqStack<ElemType>::SeqStack() {

top = -1;

}

template <class ElemType>

SeqStack<ElemType>::~SeqStack() {

}

template <class ElemType>

void SeqStack<ElemType>::Push(ElemType x) {

if(top == StackSize-1)

throw "顺序栈已满，上溢！";

top++;

data[top] = x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::Pop() {

ElemType x;

if(top == -1)

throw "顺序栈为空，下溢！";

x = data[top--];

return x;

}

template <class ElemType>

ElemType SeqStack<ElemType>::GetTop() {

if(top != -1)

return data[top];

else

throw "顺序栈为空！";

}

template <class ElemType>

int SeqStack<ElemType>::Empty() {

if(top == -1)

return 1;

else

return 0;

}

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

/\*二叉树的结点结构\*/

struct BiNode{

char data;

BiNode \*lchild, \*rchild;

};

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

/\*析构函数，释放各结点\*/

~BiTree() {

Release(root);

}

/\*前序非递归遍历二叉树\*/

void PreOrder() {

PreOrder(root);

}

/\*中序非递归遍历二叉树\*/

void InOrder() {

InOrder(root);

}

/\*后序非递归遍历二叉树\*/

void PostOrder() {

PostOrder(root);

}

private:

BiNode \*root; /\*指向根结点的头指针\*/

BiNode \*Creat(BiNode \*bt); /\*构造函数调用\*/

void Release(BiNode \*bt); /\*析构函数调用\*/

void PreOrder(BiNode \*bt); /\*前序遍历函数调用\*/

void InOrder(BiNode \*bt); /\*中序遍历函数调用\*/

void PostOrder(BiNode \*bt); /\*后序遍历函数调用\*/

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h" /\*引用二叉树类定义头文件\*/

#include "SeqStack.cpp" /\*引用顺序栈类实现源文件\*/

/\*后序遍历时使用的顺序栈的元素类型\*/

struct Element{

BiNode \*ptr;

int flag;

};

BiNode \*BiTree::Creat(BiNode \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

void BiTree::Release(BiNode \*bt) {

/\*按照后序遍历的顺序释放二叉树\*/

if(bt != NULL) {

Release(bt->lchild); /\*释放左子树\*/

Release(bt->rchild); /\*释放右子树\*/

delete bt;

}

}

/\*前序非递归遍历\*/

void BiTree::PreOrder(BiNode \*bt) {

SeqStack<BiNode \*> S;

while(bt != NULL||S.Empty() != 1) {

while(bt != NULL) {

cout<<bt->data<<" ";

S.Push(bt);

bt = bt->lchild;

}

if(S.Empty() != 1) {

bt = S.Pop();

bt = bt->rchild;

}

}

}

/\*中序非递归遍历\*/

void BiTree::InOrder(BiNode \*bt) {

SeqStack<BiNode \*> S;

while(bt != NULL || S.Empty() != 1) {

while(bt != NULL) {

S.Push(bt);

bt = bt->lchild;

}

if(S.Empty() != 1) {

bt = S.Pop();

cout<<bt->data<<" ";

bt = bt->rchild;

}

}

}

/\*后序非递归遍历\*/

void BiTree::PostOrder(BiNode \*bt) {

SeqStack<Element> S;

Element e;

/\*bt不为空或者栈不为空\*/

while(bt != NULL || S.Empty() == 0) {

while(bt != NULL) {

e.ptr = bt;

e.flag = 1;

S.Push(e);

bt = bt->lchild;

}

/\*栈不为空并且栈顶的flag为2时，出栈并访问\*/

while((S.Empty() == 0)&&(S.GetTop()).flag == 2) {

e = S.Pop();

cout<<e.ptr->data<<" ";

}

/\*栈不为空，并且栈顶的flag为1时，将栈顶的flag更改为2，并访问其右孩子\*/

if(S.Empty() == 0) {

e = S.Pop();

bt = e.ptr->rchild;

e.flag = 2;

S.Push(e);

}

}

}

1. BiTreeNoRecurMain.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree T;

cout<<"------前序非递归遍历------ "<<endl;

T.PreOrder();

cout<<endl;

cout<<"------中序非递归遍历------ "<<endl;

T.InOrder( );

cout<<endl;

cout<<"------后序非递归遍历------ "<<endl;

T.PostOrder( );

cout<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 试验结果

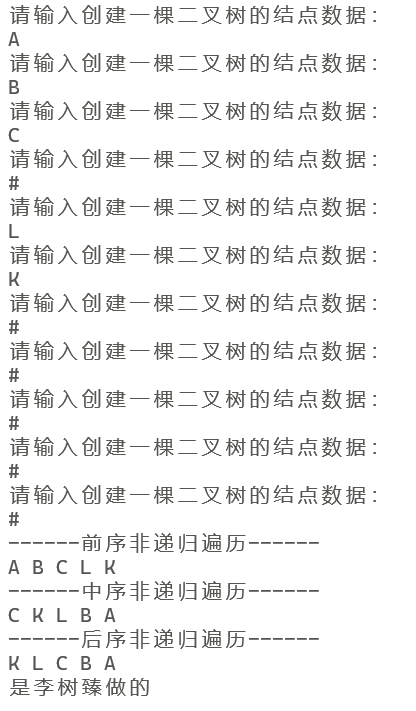


图3-2 二叉树遍历的非递归实现

### 遇到问题和解决办法

1. 问题：如何实现二叉树的非递归遍历？
2. 解决办法：对二叉树进行先序、中序、后序遍历都是从根结点开始，且在遍历的过程中，经过的节点路线都是一样的，只不过访问的顺序不同。. 先序遍历是深入时遇到结点就访问，中序遍历是深入时从左子树返回时遇到结点就访问，而后序遍历是从右子树反回时遇到根结点就访问，在这一过程中，反回结点的顺序与深入结点的顺序相反，即先深入再反回，这不禁让人联想到栈，而想要实现二叉树的非递归遍历，就需要用栈的思想来实现。其中，非递归版的中序遍历写法最简单，后序遍历最难。

## 5.3 求二叉树的最小深度

说明：利用类模板BiTree，实现二叉链表的构造、求二叉树的最小深度，并在主函数中验证，测试数据参照教材图5-13，即下图。



### 实验代码

（1）Depth.h

#include <iostream>

#include <stdio.h>

#include <malloc.h>

using namespace std;

typedef struct BiTNode

{

char data; /\*结点的数据域\*/

struct BiTNode \*lchild , \*rchild; /\*指向左孩子和右孩子\*/

} BiTNode , \*BiTree;

/\*创建一棵二叉树\*/

（2）Depth.cpp

/\*队列基本操作头文件\*/

#include"Depth.h"

void CreatBiTree(BiTree \*T)

{

char c;

cin>>c;

if(c == '#')

\*T = NULL;

else

{

\*T = (BiTNode \* )malloc(sizeof(BiTNode)); /\*创建根结点\*/

(\*T)->data = c; /\*向根结点中输入数据\*/

CreatBiTree(&((\*T)->lchild)); /\*递归地创建左子树\*/

CreatBiTree(&((\*T)->rchild)); /\*递归地创建右子树\*/

}

}

//计算二叉树的深度

int getBitreeDepth(BiTree T)

{

int leftHeight, rightHeight, maxHeight;//左子树，右子树，最大深度

if (T != NULL) //如果为空树

{

leftHeight = getBitreeDepth(T->lchild);//左子树深度

rightHeight = getBitreeDepth(T->rchild);//右子树深度

maxHeight = leftHeight>rightHeight?leftHeight:rightHeight;//最大深度

return maxHeight+1;//二叉树深度=最大深度+1

}

else

{

return 0;

}

}

（3）DepthMain.cpp

#include" Depth.cpp"

int main()

{

BiTree T = NULL; /\*最开始T指向空\*/

cout<<"请您输入一个二叉树（以#为空子树）：";

CreatBiTree(&T); /\*创建二叉树\*/

cout<<endl;

cout<<"二叉树的深度为"<<getBitreeDepth(T)<<endl;

getchar() ;

getchar() ;

}

### 实验运行结果

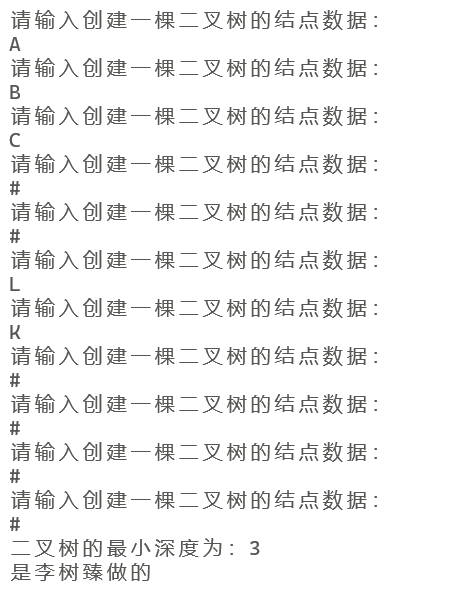


图5-3 求二叉树的最小深度

### 遇到的问题和解决的办法

（1）问题：如何求二叉树的最小深度？

解决方法：给定一个二叉树，二叉树的最小深度是从根节点到最近叶子结点距离。（叶子结点是没有子节点的结点），当二叉树为空时，返回0；当二叉树的左子树为空时，返回右子树的最小深度+1；当二叉树的右子树为空时，返回左子树的最小深度+1；当二叉树都不为空时，返回左右子树最小深度较小值+1。

（2）问题：怎样编写一个程序，把一个有序整数数组放到二叉树中？

解决方法：利用递归。

## 5.4 判断二叉树是否是完全二叉树

说明：利用类模板BiTree，实现二叉链表的构造，判断二叉树是否是完全二叉树，并在主函数中进行验证。要求分别使用教材图5-19所示的二叉树和教材图5-32所示的二叉树作为测试数据，即下列各图。





### 实验代码：

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

/\*二叉树的结点结构\*/

template <class ElemType>

struct BiNode{

ElemType data;

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

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root= Creat(root);

}

BiNode<ElemType> \*GetRoot() {

return root;

}

int IsCompleteBiTree() {

return IsCompleteBiTree(root);

}

private:

BiNode<ElemType> \*root; /\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt); /\*构造函数调用\*/

int IsCompleteBiTree(BiNode<ElemType> \*bt); /\*判断二叉树是否是完全二叉树\*/

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

#include "CirQueue.cpp"

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

int BiTree<ElemType>::IsCompleteBiTree(BiNode<ElemType> \*bt) {

CirQueue<BiNode<ElemType> \*> Q;

BiNode<ElemType> \*p;

if(bt == NULL) return 1; /\*空二叉树\*/

Q.EnQueue(bt);

/\*当出队的队头指针不为空时，将其左右指针入队\*/

while((p = Q.DeQueue()) != NULL) {

Q.EnQueue(p->lchild);

Q.EnQueue(p->rchild);

}

/\*当遇到空指针时，判断队列中是否有非空指针\*/

/\*如果有，则不是完全二叉树\*/

/\*没有，则为完全二叉树\*/

while(!Q.Empty()) {

p = Q.DeQueue();

if(p != NULL)

return 0;

}

return 1;

}

1. CirQueue.h

#ifndef CIRQUEUE\_H

#define CIRQUEUE\_H

const int QueueSize = 100; /\*定义循环队列的容量\*/

template <class ElemType> /\*定义模板类CirQueue\*/

class CirQueue{

public:

CirQueue(); /\*构造函数，循环队列的初始化\*/

void EnQueue(ElemType x); /\*入队操作\*/

ElemType DeQueue(); /\*出队操作\*/

int Empty(); /\*判断队列是否为空，若为空返回1，否则返回0\*/

private:

ElemType data[QueueSize]; /\*存放队列元素的数组\*/

int front, rear; /\*队头，队尾指针\*/

};

#endif

1. CirQueue.cpp

#include <iostream>

using namespace std;

#include "CirQueue.h"

template <class ElemType>

CirQueue<ElemType>::CirQueue() {

front = QueueSize-1;

rear = QueueSize-1;

}

template <class ElemType>

void CirQueue<ElemType>::EnQueue(ElemType x) {

if((rear + 1) % QueueSize == front)

throw "循环队列已满，溢出！";

rear = (rear + 1) % QueueSize;

data[rear] = x;

}

template <class ElemType>

ElemType CirQueue<ElemType>::DeQueue() {

if(rear == front)

throw "循环队列为空！";

front = (front + 1) % QueueSize;

return data[front];

}

template <class ElemType>

int CirQueue<ElemType>::Empty() {

if(front == rear)

return 1;

else

return 0;

}

1. JudgeCompleteBiTreeMain.cpp

#include <iostream>

#include <cmath>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T;

int flag=T.IsCompleteBiTree();

if(flag==true)

cout<<"二叉树是完全二叉树！";

else

cout<<"二叉树不是完全二叉树！";

return 0;

}

### 实验结果

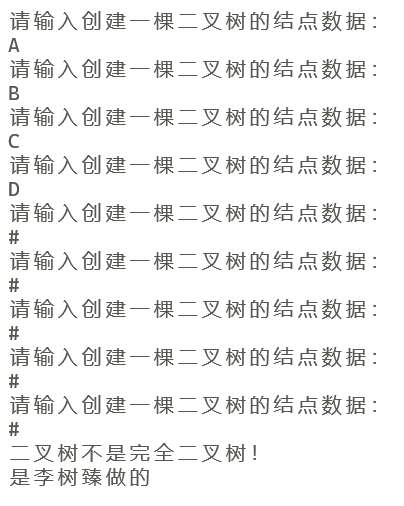


图5-4 判断二叉树是否是完全二叉树

### **遇到的问题和解决办法**：

1. 问题：如何判断一棵二叉树是完全二叉树？

解决办法：在层序遍历一棵非完全二叉树时，非空结点之前会出现空节点。而层序遍历一棵完全二叉树时，在非空节点之前不会出现空节点，即一旦层序遍历中出现空节点，则之后不会再出现非空节点。如果在遍历二叉树时遇到空节点，整棵二叉树的遍历已经完成，则二叉树是完全二叉树；如果遇到空节点时，整棵二叉树的遍历还没有结束，则二叉树不是完全二叉树。

## 5.5 判断二叉树的结构是否对称

说明：利用类模板BiTree，实现二叉链表的构造，并判断二叉树是否结构对称，并在主函数中进行验证。要求分别使用教材图5-34和图5-35所示的二叉树作为测试数据，即下列各图。





### 实验代码：

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

template <class ElemType>

/\*二叉树的结点结构\*/

struct BiNode{

ElemType data;

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

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

/\*判断二叉树是否结构性对称\*/

int IsStructureSymmetric() {

return IsStructureSymmetric(root);

}

private:

BiNode<ElemType> \*root; /\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt); /\*构造函数调用\*/

int IsStructureSymmetric(BiNode<ElemType> \*bt);

int IsCheck(BiNode<ElemType> \*lchild, BiNode<ElemType> \*rchild);

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

int BiTree<ElemType>::IsStructureSymmetric(BiNode<ElemType> \*bt) {

if(bt == NULL)

return 1;

return IsCheck(bt->lchild, bt->rchild);

}

template <class ElemType>

int BiTree<ElemType>::IsCheck(BiNode<ElemType> \*lchild, BiNode<ElemType> \*rchild) {

if(lchild == NULL && rchild == NULL)

return 1;

if(lchild == NULL || rchild == NULL)

return 0;

return IsCheck(lchild->lchild, rchild->rchild) && IsCheck(lchild->rchild, rchild->lchild);

}

1. JudgeStructureSymmetricMain.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T;

bool flag = T.IsStructureSymmetric();

if(flag)

cout<<"二叉树是结构性对称的！"<<endl;

else

cout<<"二叉树不是结构性对称的！"<<endl;

return 0;

}

### 实验结果：



图5-5-1 判断二叉树结构是否对称

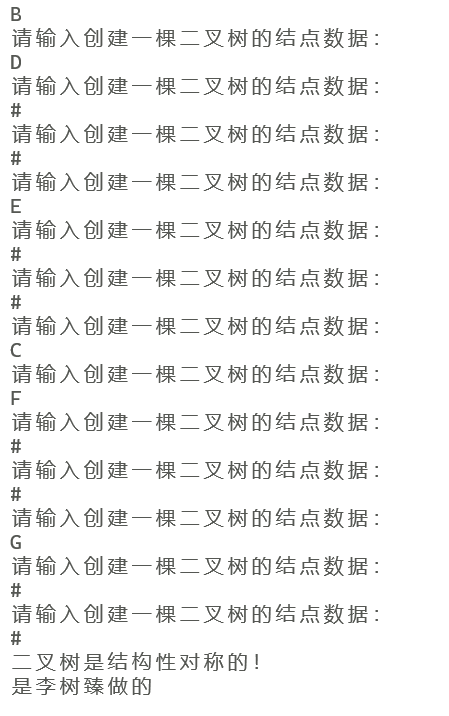


图5-5-2 判断二叉树结构是否对称

### 遇到问题和解决办法：

1. 问题：如何判断二叉树结构对称？

解决办法：二叉树结构性对称指的是不考虑结点的数据值，对于二叉树中任意一个非空的结点，左右孩子都不存在，或者左右孩子都存在；如果结点存在子树，则左右子树的结构也分别对称。如果二叉树为空，则是结构对称的。如果二叉树不为空，则判断其左右子树是否是结构对称的。

## 5.6 判断二叉树是否对称

说明：利用类模板BiTree，实现二叉链表的构造，并判断二叉树是否是对称的，并在主函数中进行验证。要求分别使用教材图5-38和图5-39所示的二叉树作为测试数据，即下列各图。





### 实验代码

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

template <class ElemType>

/\*二叉树的结点结构\*/

struct BiNode{

ElemType data;

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

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

/\*判断二叉树是否结构性对称\*/

bool IsSymmetric() {

return IsSymmetric(root);

}

private:

BiNode<ElemType> \*root; /\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt); /\*构造函数调用\*/

int IsSymmetric(BiNode<ElemType> \*bt); /\*判断二叉树是否对称\*/

int IsCheck(BiNode<ElemType> \*lchild, BiNode<ElemType> \*rchild); /\*判断非空二叉树的左右子树是否对称\*/

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

int BiTree<ElemType>::IsSymmetric(BiNode<ElemType> \*bt) {

if(bt == NULL)

return 1;

return IsCheck(bt->lchild, bt->rchild);

}

template <class ElemType>

int BiTree<ElemType>::IsCheck(BiNode<ElemType> \*lchild, BiNode<ElemType> \*rchild) {

if(lchild == NULL && rchild == NULL)

return 1;

if(lchild == NULL||rchild == NULL)

return 0;

/\*此处与判断结构性对称不同\*/

if(lchild->data != rchild->data)

return 0;

return IsCheck(lchild->lchild,rchild->rchild) && IsCheck(lchild->rchild,rchild->lchild);

}

1. JudgeSymmetricMain.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T;

bool flag = T.IsSymmetric();

if(flag)

cout<<"二叉树是对称的！"<<endl;

else

cout<<"二叉树不是对称的！"<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验结果

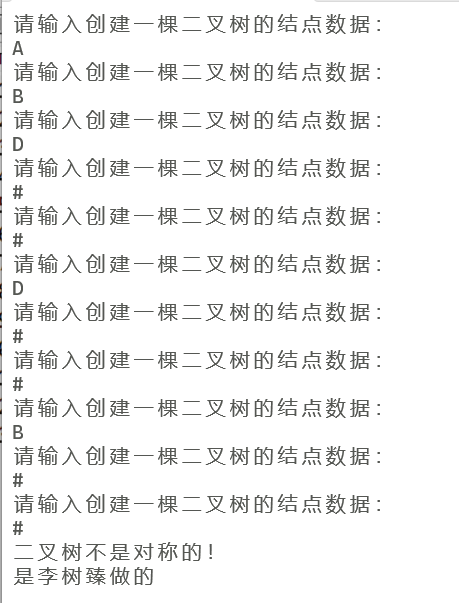


图5-6-1 判断二叉树是否对称

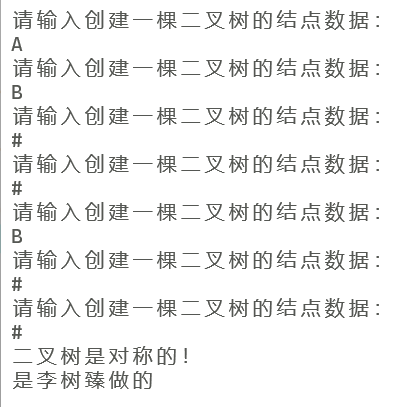


图5-6-2 判断二叉树是否对称

### 遇到问题和解决办法

1. 问题：如何判断二叉树对称？

解决办法：在考虑结点值的情况下，二叉树的任意一个非空节点，如果结点存在左右孩子，则左右孩子相同。并且左右孩子也都是关于中轴线对称的。与判断二叉树是否结构对称的算法非常类似，区别是结构性对称不需要考虑结点的值，而对称需要考虑结点的值。

（2）问题：怎样编写一个程序，把一个有序整数数组放到二叉树中？

解决方法：利用递归。

## 5.7 求二叉树中叶子结点个数及结点个数

说明：利用类模板BiTree，实现二叉链表的构造，并计算二叉树中叶子结点的个数及结点个数，并在主函数中输出。要求使用教材图5-13所示的二叉树作为测试数据，即下图。



### 实验代码

（1）BitreeNum.h

#include<iostream>

#include "stdio.h"

#include "stdlib.h"

using namespace std;

typedef struct BiTNode

{

char data;//树中节点的数据

struct BiTNode \*lchild,\*rchild;//节点的左孩子指针和右孩子指针

}BiTNode,\*BiTree;//二叉树节点

（2）BitreeNum.cpp

#include" BitreeNum.h"

//创建树

void CreateBiTree(BiTree &T)

{

char ch;

cin>>ch;

if(ch == '#')

{

T = NULL;

return;

}

T = (BiTNode\*)malloc(sizeof(BiTNode));

T->data = ch;

CreateBiTree(T->lchild);

CreateBiTree(T->rchild);

}

//计算树中的叶子节点数

int LeafCount(BiTree T)

{

if(NULL == T)

{

return 0;

}

else

{

if(NULL == T->lchild && NULL == T->rchild)

{

return 1;

}

else

{

return LeafCount(T->lchild) + LeafCount(T->rchild);

}

}

}

//树的节点个数

int Nodes(BiTree T)

{

if(NULL == T)

return 0;

else

{

if(NULL == T->lchild && NULL == T->rchild)

return 1;

else

{

return Nodes(T->lchild) + Nodes(T->rchild) + 1;

}

}

}

（3）BitreeNum Main.cpp

#include" BitreeNum.cpp"

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

{

BiTree T = NULL;

CreateBiTree(T);

int leaf = LeafCount(T);

cout<<"二叉树中叶子结点个数："<<leaf<<endl;

int nodes = Nodes(T);

cout<<"二叉树中结点个数："<<nodes<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验运行结果

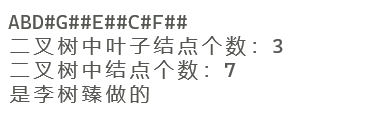


图5-7 二叉树中结点个数和叶子结点个数

### 遇到问题和解决办法

（1）问题：求二叉树叶子结点个数及结点个数可以用什么算法实现？

解决方法：使用递归算法或非递归算法。

（2）问题：计算二叉树叶子结点个数的思路是什么？

解决方法：1. 使用递归创建并初始化二叉树。当输入的数据不为“#”时，将该元素视为一个有效的元素，否则置为null。每次递归返回当前位置的子树。2. 计算二叉树的所有叶子节点的数量。当一个节点的左孩子和右孩子都为空时。他是叶子节点。使用递归如果能找到就返回1，如果节点为NULL返回0，否则返回count(t->lchild)+ count(t->rchild)。

## 5.8 中序线索二叉树的构造

说明：利用InThrBiTree类模板，需要依次完成以下工作：

（1）声明和定义InThrBiTree类模板；

（2）构造初始的未线索化的中序线索二叉链表；

（3）完成求已知结点p的后继结点的算法；

（4）完成对中序线索二叉树进行非递归的中序遍历的算法；

（5）在主函数中对遍历算法进行验证，输出中序遍历序列。

测试数据参考下图。



### 实验代码

（1）MiddleOrder.h

#include <bits/stdc++.h>

using namespace std;

typedef struct BiTNode{

char data;//数据域

struct BiTNode \*lchild,\*rchild;//左右孩子指针

}BiTNode,\*BiTree;

stack<BiTree> s;//创建栈用于中序遍历

（2）MiddleOrder.cpp

#include" MiddleOrder.h "

void InitBiTree(BiTree \*T){

\*T = NULL;

}

/\*

创建树，以先序方式插入树节点

\*/

void createBiTree(BiTree \*T){

char ch;

cin>>ch;

if(ch =='#') \*T=NULL;

else{

\*T = (BiTree)malloc(sizeof(BiTNode));

if(!(\*T)) exit(-1);

//按照先序的方式插入生成树

(\*T)->data = ch;

createBiTree(&((\*T)->lchild));

createBiTree(&((\*T)->rchild));

}

}

/\*

利用栈，对二叉树进行非递归中序遍历

\*/

void InOrder(BiTree T){

BiTree p = T;//p是遍历指针

while(p||!s.empty()){//当栈不空或p不空时进入下面循环

if(p){ //如果p不为空就一路向左并将所有左孩子入栈

s.push(p);

p = p->lchild;

}else{ //当碰到最后一个左孩子的左节点为空时，进入该循环，进行出栈

p=s.top();

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

s.pop();

p = p->rchild;//出完栈后，转到右孩子

}

}

cout<<endl;

}

（3）MiddleOrderMain.cpp

#include" MiddleOrder.cpp"

int main(){

BiTree T;

InitBiTree(&T);

cout<<"请按照先序地方式输入二叉树(输入#代表空)："<<endl;

createBiTree(&T);

cout<<"该二叉树的中序遍历是：";

InOrder(T);

return 0;

}

### 实验运行结果

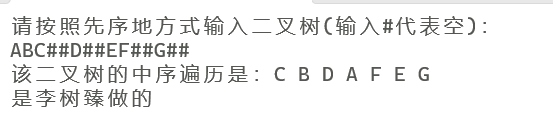


图5-8 中序线索二叉树的构造

### **遇到的问题和解决的办法**

（1）问题：中序线索二叉树的构造方法是什么？

解决方法：直接利用递归的中序遍历算法来完成，中间需要一个指向前驱结点的指针，初值为空。1. 递归中序遍历左子树。2. 访问某个结点时，同时修改前驱指向自己（如果有）和自己指向前驱（如果有）的线索与标志完了后，前驱移动指向当前结点。3. 递归中序遍历右子树。

1. 问题如何对中序线索二叉树进行非递归的中序遍历？

解决办法：主要应用了栈。

## 5.9 哈夫曼算法的实现

说明：使用三叉链表的静态链表模板存储二叉树的各个结点，实现哈夫曼算法。以教材图5-27的数据为例，即下图。



### 实验代码

#include<iostream>

#include <vector>

#include <algorithm>

using namespace std;

class huffumanNode

{

public:

int weight;

huffumanNode \*left;

huffumanNode \*right;

//用来构造叶子节点

huffumanNode(int setWeight)

{

this->weight = setWeight;

this->left = NULL;

this->right = NULL;

}

//用来构造中间的节点

huffumanNode(int setWeight, huffumanNode \*setLeft, huffumanNode \*setRight)

{

this->weight = setWeight;

this->left = setLeft;

this->right = setRight;

}

//判断是否是叶子节点

bool isLeaf()

{

return this->left == NULL && this->right == NULL;

}

};

//降序排序比较函数，以wieght的大小降序排列

bool myHuffumanComepare(huffumanNode \*node1, huffumanNode \*node2)

{

return node1->weight > node2->weight;

}

/\*得到带权路径长度。所谓树的带权路径长度，就是树中所有的叶结点

的权值乘上其到根结点的 路径长度（若根结点为0层，叶结点到根结点的路径长度

为叶结点的层数）\*/

void getHuffmanSum(huffumanNode \*root,int length,int &sum)

{

if (!root)

{

return;

}

if (root->isLeaf())

{

sum += root->weight\*length;

}

length++;

getHuffmanSum(root->left, length, sum);

getHuffmanSum(root->right, length, sum);

}

//前序遍历打印，提交时不要调用这个哈，仅为调试

void prePrint(huffumanNode \*root)

{

if (!root)

{

return;

}

cout << root->weight << " ";

prePrint(root->left);

prePrint(root->right);

}

int main()

{

cout<<"请输入叶子节点的个数："<<endl;

int n;

while (cin >> n)

{

vector<huffumanNode\*> weightArray;

cout<<"请输入所有叶子节点对应的权值："<<endl;

while (n--)

{

int tempWeight;

cin >> tempWeight;

huffumanNode \*newNode = new huffumanNode(tempWeight);

weightArray.push\_back(newNode);

}

sort(weightArray.begin(), weightArray.end(), myHuffumanComepare);

while ((int)weightArray.size() > 1)

{

huffumanNode \*minNode1 = weightArray[weightArray.size() - 1];

huffumanNode \*minNode2 = weightArray[weightArray.size() - 2];

int fatherWeight = minNode1->weight + minNode2->weight;

/\*根据最小的两个权重节点，构造新节点\*/

huffumanNode \*fatherNode = new huffumanNode(fatherWeight, minNode1, minNode2);

weightArray.pop\_back();

weightArray.pop\_back();

weightArray.push\_back(fatherNode);

sort(weightArray.begin(), weightArray.end(), myHuffumanComepare);

}

cout<<"先序遍历此哈夫曼树的权值："<<endl;

prePrint(weightArray[0]);

cout << endl;

int length = 0;

int sum = 0;

getHuffmanSum(weightArray[0], length, sum);

cout << "树的所有叶结点的带权路径长度之和(WPL): " << sum << endl;

}

return 0;

}

### 实验运行结果

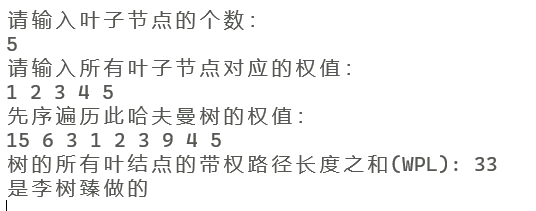


图5-9 哈夫曼算法

### 遇到的问题和解决的办法

（1）问题：如何使用已知的叶子节点来构造哈夫曼树

解决方法：每次都选择频率最小的两个子树，然后固定把最小两个中较大的一个放在左子树，较小的放在右边来合并成一个新的树，根节点的频率为两子树频率相加。

（2）问题：如何判断是否存在左右叶子节点

解决方法：构造bool函数进行判断

## 5.10 由前序遍历序列和中序遍历序列构造二叉树

说明：输入二叉树的前序遍历序列和中序遍历序列，构造二叉树，并对构造的二叉树进行前序、中序、后序遍历。要求使用教材图5-51所示的二叉树作为测试数据，即下图。



### 实验代码

ConBitree.h

#include <iostream>

#include <string>

using namespace std;

struct node

{

char data;

node \* lchild;

node \* rchild;

};

node \*createNode(int Xb, int Xe, int Zb, int Ze,string xianxu,string zhongxu);

void houxu(node \*Node);

（2）ConBitree.cpp

#include" ConBitree.h"

node \*createNode(int Xb, int Xe, int Zb, int Ze,string xianxu,string zhongxu)

{

if(Xb >Xe)

return NULL;

node \*Node = new node;

Node->data = xianxu[Xb];

int k=0;

for (int i = Zb; i <Ze+1; i++)

{

if (xianxu[Xb] == zhongxu[i])

{

k = i;

break;

}

}

Node->lchild = createNode(Xb + 1, Xb+k-Zb, Zb, k - 1, xianxu, zhongxu);

Node->rchild = createNode(Xb+k-Zb + 1, Xe, k + 1, Ze, xianxu, zhongxu);

return Node;

}

void houxu(node \*Node)

{

if (Node == NULL)

return;

houxu(Node->lchild);

houxu(Node->rchild);

cout << Node->data;

}

（3）ConBitreeMain.cpp

#include" ConBitree.cpp"

int main()

{

string xianxu;

string zhongxu;

cout<<"请输入前序遍历序列：";

while (cin >> xianxu) {

cout<<"请输入中序遍历序列：";

cin >> zhongxu;

int Xb, Xe, Zb, Ze;//Xb表示先序开始字符位置Xe表示先序结束字符位置 Zb表示中序开始字符位置,Ze表示中序结束字符位置

Xb = Zb = 0;

Xe = xianxu.length() - 1;

Ze = zhongxu.length() - 1;

node \*Node = new node;

Node = createNode(Xb, Xe, Zb, Ze, xianxu, zhongxu);

cout<<"后序遍历序列为：";

Cout << “是李树臻做的“ << endl;

houxu(Node);

}

system("pause");

return 0;

}

### 实验运行结果

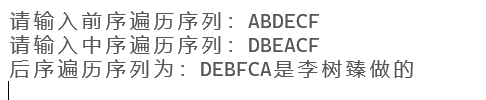


图5-10 由前序遍历序列和中序遍历序列构造二叉树

### 遇到的问题和解决的办法

（1）问题：由前序遍历序列和中序遍历序列构造二叉树的解题思路是什么？

解决方法：二叉树的先序遍历先访问根结点，其次遍历根节点的左子树，然后遍历根节点的右子树。中序遍历，先遍历左子树，然后遍历根结点，最后遍历右子树。因此中序遍历序列被根节点分为两部分：根结点之前的部分为左子树结点中序序列，根结点之后的为右子树结点中序序列。

（2）问题：怎样编写一个程序，把一个有序整数数组放到二叉树中？

解决方法：利用递归。

## 5.11 由后序遍历序列和中序遍历序列构造二叉树

说明：输入二叉树的后序遍历序列和中序遍历序列，构造二叉树，并对构造的二叉树进行前序、中序、后序遍历。要求使用教材图5-51所示的二叉树作为测试数据，即下图。



### 实验代码

（1）ConBitree2.h

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include<math.h>

#include <iostream>

using namespace std;

#define MAXSIZE 100

typedef struct Node

{

char data;

struct Node \*lchild, \*rchild;

}BitNode, \*BiTree;

（2）ConBitree2.cpp

#include"ConBitree2.h"

void CreateBiTree(BiTree \*T, char \*in, char \*post, int len);

void PrintLevel(BiTree T);

void PreTraverse(BiTree T);

void PrintLevel(BiTree T)

{

BiTree Queue[MAXSIZE];

int front, rear;

if (T==NULL)

{

return;

}

front = -1;

rear = 0;

Queue[rear] = T;

while (front!=rear)

{

front++;

printf("%4c",Queue[front]->data);

if (Queue[front]->lchild!=NULL)

{

rear++;

Queue[rear] = Queue[front]->lchild;

}

if (Queue[front]->rchild!=NULL)

{

rear++;

Queue[rear] = Queue[front]->rchild;

}

}

}

void PreTraverse(BiTree T)

{

if (T!=NULL)

{

printf("%4c", T->data);

PreTraverse(T->lchild);

PreTraverse(T->rchild);

}

}

void CreateBiTree(BiTree \*T, char \*in, char \*post, int len)

{

int k;

char \*temp;

if (len<=0)

{

\*T = NULL;

return;

}

for (temp = in; temp < in + len;temp++)

{

if (\*(post+len-1)==\*temp)

{

k = temp - in;

(\*T) = (BitNode \*)malloc(sizeof(BitNode));

(\*T)->data = \*temp;

break;

}

}

CreateBiTree(&((\*T)->lchild), in, post, k);

CreateBiTree(&((\*T)->rchild), in + k + 1, post + k, len - k - 1);

}

void TreePrint(BiTree T, int level)

{

int i;

if (T==NULL)

{

return;

}

TreePrint(T->rchild, level + 1);

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

{

printf(" ");

}

printf("%c\n", T->data);

TreePrint(T->lchild, level + 1);

}

（3）ConBitreeMain2.cpp

#include "ConBitree2.cpp "

int main()

{

BiTree T;

int len;

char in[MAXSIZE], post[MAXSIZE];

cout<<"由中序序列和后序序列构造二叉树： "<<endl;

cout<<"请输入中序的字符串序列：";//DBGEACF

gets(in);

printf("请输入后序的字符串序列：");//DGEBFCA

gets(post);

len = strlen(post);

CreateBiTree(&T, in, post, len);

TreePrint(T, 1);

printf("\n建立的二叉树先序遍历结果是： \n");

PreTraverse(T);

printf("\n建立的二叉树层序遍历结果是： \n");

PrintLevel(T);

printf("\n");

system("pause");

}

### 实验运行结果

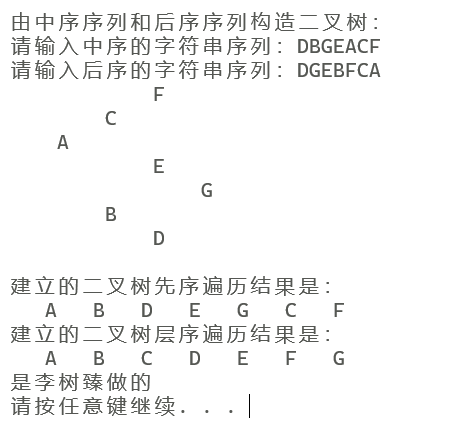


图5-11 由后序遍历序列和中序遍历序列构造二叉树

### 遇到的问题和解决的办法

（1）问题：由后序遍历序列和中序遍历序列构造二叉树的解题思路是什么？

解决方法：根据后序遍历，先找到这棵树的根节点的值，也就是数组中最后一个节点(数组长度-1)的位置，由此创建根节点。然后在中序遍历中找到根的值所在的下标，切出左右子树的前序和中序。

## 5.12 求二叉树第k层的结点个数和叶子结点个数

说明：利用BiTree类模板

实验数据参考下图。



可以求第3层的结点个数和叶子结点个数。

### 实验代码

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

/\*二叉树的结点结构\*/

template <class ElemType>

struct BiNode{

ElemType data;

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

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

int GetNodesNumberOfKthLevel(int k) {

return GetNodesNumberOfKthLevel(root, k);

}

int GetLeafNodesNumberOfKthLevel(int k) {

return GetLeafNodesNumberOfKthLevel(root, k);

};

private:

BiNode<ElemType> \*root;/\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt);/\*构造函数调用\*/

int GetNodesNumberOfKthLevel(BiNode<ElemType> \*bt, int k); /\*获取第k层的结点数\*/

int GetLeafNodesNumberOfKthLevel(BiNode<ElemType> \*bt, int k); /\*获取第k层的叶子结点数\*/

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

int BiTree<ElemType>::GetNodesNumberOfKthLevel(BiNode<ElemType> \*bt, int k) {

if(bt == NULL || k < 1)

return 0;

if(k == 1)

return 1;

return GetNodesNumberOfKthLevel(bt->lchild, k - 1) + GetNodesNumberOfKthLevel(bt->rchild, k - 1);

}

template <class ElemType>

int BiTree<ElemType>::GetLeafNodesNumberOfKthLevel(BiNode<ElemType> \*bt, int k) {

if(bt == NULL || k < 1)

return 0;

if(bt != NULL) {

/\*判断bt是否是叶子\*/

if(k == 1) {

if(bt->lchild == NULL && bt->rchild == NULL)

return 1;

else

return 0;

}

/\*递归求左右子树中的叶子数的和\*/

if(k > 1) {

return GetLeafNodesNumberOfKthLevel(bt->lchild, k - 1) + GetLeafNodesNumberOfKthLevel(bt->rchild, k - 1);

}

}

}

1. GetNodesNumberOfKthLevelMain.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T;

int nodes,leafNodes;

int k;

cout<<"请输入层数k：";

cin>>k;

nodes = T.GetNodesNumberOfKthLevel(k);

leafNodes = T.GetLeafNodesNumberOfKthLevel(k);

cout<<"二叉树的第"<<k<<"层的结点个数为"<<nodes<<"个。"<<endl;

cout<<"二叉树的第"<<k<<"层的叶子结点个数为"<<leafNodes<<"个。"<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验结果

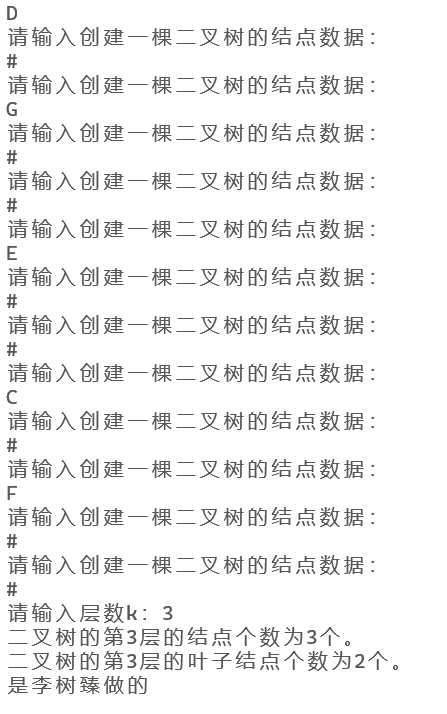


图5-12 求二叉树第k层的节点个数和叶子结点个数

### 遇到的问题和解决办法

1. 问题：如何求二叉树第k层的结点个数和叶子结点个数？

解决办法：采用递归。

（2）问题：求二叉树叶子结点个数及结点个数可以用什么算法实现？

解决方法：使用递归算法或非递归算法。

## 5.13 打印二叉树第k层的结点和叶子结点

实验数据参考下图。



可以打印第3层的结点和叶子结点。

### 实验代码

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

/\*二叉树的结点结构\*/

template <class ElemType>

struct BiNode{

ElemType data;

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

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

void PrintNodesOfKthLevel(int k) {

PrintNodesOfKthLevel(root, k);

}

void PrintLeafNodesOfKthLevel(int k) {

PrintLeafNodesOfKthLevel(root, k);

}

private:

BiNode<ElemType> \*root; /\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt); /\*构造函数调用\*/

void PrintNodesOfKthLevel(BiNode<ElemType> \*bt, int k); /\*打印第k层的结点\*/

void PrintLeafNodesOfKthLevel(BiNode<ElemType> \*bt, int k); /\*打印第k层的叶子结点\*/

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

void BiTree<ElemType>::PrintNodesOfKthLevel(BiNode<ElemType> \*bt, int k) {

if(bt == NULL || k < 1)

return;

if(k == 1)

cout<<bt->data<<" ";

PrintNodesOfKthLevel(bt->lchild, k-1);

PrintNodesOfKthLevel(bt->rchild, k-1);

}

template <class ElemType>

void BiTree<ElemType>::PrintLeafNodesOfKthLevel(BiNode<ElemType> \*bt, int k) {

if(bt == NULL || k < 1)

return;

if(bt != NULL) {

/\*判断bt是否是叶子\*/

if(k == 1) {

if(bt->lchild == NULL && bt->rchild == NULL)

cout<<bt->data<<" ";

}

else if(k > 1) {

/\*递归打印左子树第k-1层的叶子结点\*/

PrintLeafNodesOfKthLevel(bt->lchild, k - 1);

/\*递归打印右子结第k-1层的叶子结点\*/

PrintLeafNodesOfKthLevel(bt->rchild, k - 1);

}

}

}

1. PrintNodesOfKthLevelMain.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T;

int k;

cout<<"请输入层数k：";

cin>>k;

cout<<"二叉树第"<<k<<"层的结点为：";

T.PrintNodesOfKthLevel(k);

cout<<endl;

cout<<"二叉树第"<<k<<"层的叶子结点为：";

T.PrintLeafNodesOfKthLevel(k);

cout<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验结果

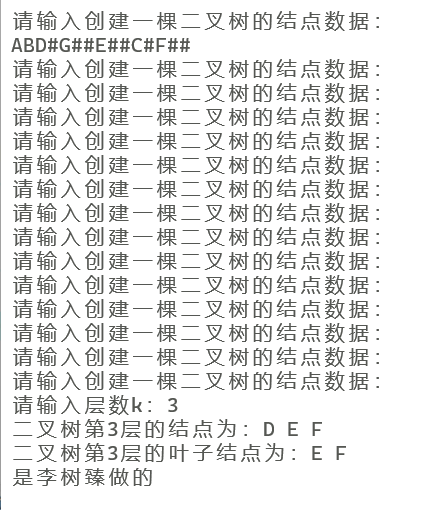


图5-13 打印二叉树第k层的结点和叶子结点

### 遇到问题和解决办法

1. 如何打印二叉树第k层的结点和叶子结点？

解决办法：运用递归算法。想要输出叶子结点，需要计算出结点的左右子树是不是同时为空，如果是就输出。

## 5.14 求二叉树最大的结点距离

实验数据可分别参考以下各图。







### 实验代码

1. BiTree.h

#ifndef BITREE\_H

#define BITREE\_H

template <class ElemType>

/\*二叉树的结点结构\*/

struct BiNode{

ElemType data;

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

int maxLeft; /\*左子树结点的最大距离\*/

int maxRight; /\*右子树结点的最大距离\*/

};

template <class ElemType>

class BiTree{

public:

/\*构造函数，建立一棵二叉树\*/

BiTree() {

root = Creat(root);

}

void GetMaxPathLength() {

GetMaxPathLength(root);

}

private:

BiNode<ElemType> \*root; /\*指向根结点的头指针\*/

BiNode<ElemType> \*Creat(BiNode<ElemType> \*bt); /\*构造函数调用\*/

void GetMaxPathLength(BiNode<ElemType> \*bt); /\*求二叉树的结点的最大距离\*/

};

#endif

1. BiTree.cpp

#include <iostream>

using namespace std;

#include "BiTree.h"

/\*全局变量，记录二叉树的最大的结点距离\*/

int maxLen = 0;

template <class ElemType>

BiNode<ElemType> \*BiTree<ElemType>::Creat(BiNode<ElemType> \*bt) {

char ch;

cout<<"请输入创建一棵二叉树的结点数据："<<endl;

cin>>ch;

/\*'#'代表空二叉树\*/

if(ch == '#')

return NULL;

else {

bt = new BiNode<ElemType>; /\*生成新结点\*/

bt->data = ch;

bt->lchild = Creat(bt->lchild); /\*递归创建左子树\*/

bt->rchild = Creat(bt->rchild); /\*递归创建右子树\*/

}

return bt;

}

template <class ElemType>

void BiTree<ElemType>::GetMaxPathLength(BiNode<ElemType> \*bt) {

/\*二叉树为空，空操作返回\*/

if(bt == NULL) {

return;

}

/\*二叉树左子树为空，则左子树的最大结点距离为0\*/

if(bt->lchild == NULL) {

bt->maxLeft = 0;

}

/\*二叉树右子树为空，则右子树的最大结点距离为0\*/

if(bt->rchild == NULL) {

bt->maxRight = 0;

}

/\*如果左子树不为空，递归寻找左子树的最大结点距离\*/

if(bt->lchild != NULL) {

GetMaxPathLength(bt->lchild);

}

/\*如果右子树不为空，递归寻找右子树的最大结点距离\*/

if(bt->rchild != NULL) {

GetMaxPathLength(bt->rchild);

}

/\*计算左子树最大结点距离\*/

if(bt->lchild != NULL) {

int temp = 0;

int ll = bt->lchild->maxLeft;

int lr = bt->lchild->maxRight;

temp = ll > lr ? ll : lr;

bt->maxLeft = temp + 1;

}

/\*计算右子树最大结点距离\*/

if(bt->rchild != NULL) {

int temp = 0;

int rl = bt->rchild->maxLeft;

int rr = bt->rchild->maxRight;

temp = rl > rr ? rl : rr;

bt->maxRight = temp + 1;

}

/\*更新最大结点距离\*/

if(bt->maxLeft + bt->maxRight > maxLen) {

maxLen = bt->maxLeft + bt->maxRight;

}

}

1. GetMaxPathLength.cpp

#include <iostream>

using namespace std;

#include "BiTree.cpp"

int main() {

BiTree<char> T; /\*创建一棵二叉树\*/

T.GetMaxPathLength();

cout<<"二叉树的最长距离为：";

cout<<maxLen<<endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验结果



图5-14 求二叉树最大结点距离

### **遇到问题和解决方案**

1. 问题：如何求二叉树最大结点距离？

解决办法：如果把二叉树看成图，父子结点之间的连线看做是双向的，则两个结点之间的距离定义为两个结点之间的边的条数。二叉树中结点之间的最大距离也称为二叉树的最长路径。

从以上例子可以看出，相距最远的两个结点，或者是两个叶子结点，或者是一个叶子结点到根节点，如果最长路径经过根节点，则距离最远的结点都是左右子树中距离根节点最远的叶子结点，如果路径不经过根节点，则肯定是根的左子树中的最大结点距离或者是根的右子树的最大的结点距离。

# 第6章 图

图这一章全部需要输入，图的类型包括：有向图，无向图，有向网，无向网，根据具体情况决定。

图需要输入图的顶点个数，边数，边依附的顶点以及可能的权值。

## 6.1 邻接矩阵存储无向图的实现

说明：实现类模板MGraph，基本操作包括构造函数、深度优先遍历、广度优先遍历，并验证。

实验数据参考下图。



### 实验代码

#include<bits/stdc++.h>

using namespace std;

#define MAX 105

template <class MGraph\_entry>

class MGraph

{

private:

int edg[MAX][MAX];//存边

int n,e;//分别为顶点数和边数

int visit[MAX];

public:

MGraph(); //构造函数

void DFS(MGraph \*G,int v);

void BFS(MGraph \*G,int v);

};

template <class MGraph\_entry>

MGraph<MGraph\_entry>::MGraph(){

for(int i=0;i<MAX;i++){

for(int j=0;j<MAX;j++){

edg[i][j]=0;

}

}

memset(visit,0,sizeof(visit));

cout<<"请输入无向图的顶点数和边数："<<endl;

cin>>n>>e;

cout<<"输入该图的"<<e<<"条边的两个端点(1 ~ m)："<<endl;

int x,y;

for(int i=0;i<e;i++){

cin>>x>>y;

edg[x][y]=edg[y][x]=1;

}

}

template <class MGraph\_entry>

void MGraph<MGraph\_entry>::DFS(MGraph \*G,int v){//深度优先遍历

if(!G->visit[v]){

G->visit[v]=1;

cout<<v<<" ";

}

for(int j=1;j<=n;j++){

if(G->edg[v][j]&&!G->visit[j]){//edg[v][j]是与v相邻的节点，visit[j]判断该节点是否被访问

DFS(G,j);

}

}

}

template <class MGraph\_entry>

void MGraph<MGraph\_entry>::BFS(MGraph \*G,int v){//广度优先遍历

memset(G->visit,0,sizeof(G->visit));

queue<MGraph\_entry> q;

cout<<v<<" ";

G->visit[v]=1;

q.push(v);

int x;

while(!q.empty()){

x=q.front();

q.pop();

for(int i=1;i<=G->n;i++){

if(G->edg[x][i]&&!visit[i]){

cout<<i<<" ";

visit[i]=1;

q.push(i);

}

}

}

memset(G->visit,0,sizeof(G->visit));

}

int main(){

MGraph<int> mgraph;

cout<<"通过深度优先遍历以此访问的节点如下："<<endl;

mgraph.DFS(&mgraph,1);

cout<<endl<<"通过广度优先遍历以此访问的节点如下："<<endl;

mgraph.BFS(&mgraph,1);

cout<<endl<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

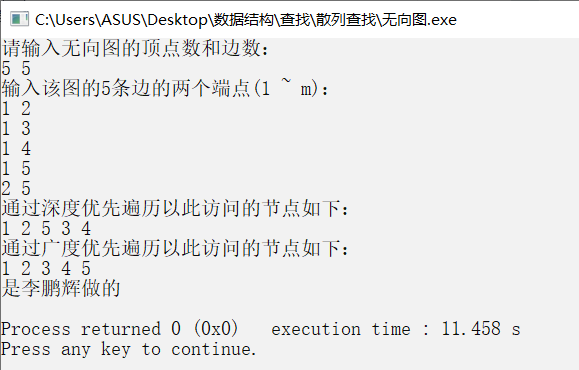


图6-1 邻接矩阵存储无向图的实现

### 遇到的问题和解决的办法

（1）无向图的边的矩阵一定是一个对称矩阵，因为无向图只关心边是否存在，而不关心方向，V0和V1有边，那么V1和V0也有边。

（2）因为这里不研究有圈图，所以主对角线都是0，输入V0和V1边的关系后，就不必输入V1和V0的关系了。

（3）广度优先遍历类似于树的层序遍历，深度优先遍历类似于树的先序遍历，在树中没有回路，而图中可能存在回路，在搜索相邻点时可能搜到已访问的点，所以我们通过visit数组来存放各个点的状态。

## 6.2 邻接表存储有向图的实现

说明：实现类模板ALGraph，基本操作包括构造函数、深度优先遍历、广度优先遍历，并验证。

### 实验代码

#include<bits/stdc++.h>

#define MAX 105

using namespace std;

// 邻接表

class ListDG

{

private: // 内部类

// 邻接表中表对应的链表的顶点

class ENode

{

public:

int ivex; // 该边所指向的顶点的位置

ENode \*nextEdge; // 指向下一条弧的指针

};

// 邻接表中表的顶点

class VNode

{

public:

char data; // 顶点信息

ENode \*firstEdge; // 指向第一条依附该顶点的弧

};

private: // 私有成员

int mVexNum; // 图的顶点的数目

int mEdgNum; // 图的边的数目

VNode mVexs[MAX];

public:

// 创建邻接表对应的图(自己输入)

ListDG();

// 创建邻接表对应的图(用已提供的数据)

ListDG(char vexs[], int vlen, char edges[][2], int elen);

~ListDG();

// 深度优先搜索遍历图

void DFS();

// 广度优先搜索（类似于树的层次遍历）

void BFS();

// 打印邻接表图

void print();

private:

// 返回ch的位置

int getPosition(char ch);

// 深度优先搜索遍历图的递归实现

void DFS(int i, int \*visited);

// 将node节点链接到list的最后

void linkLast(ENode \*list, ENode \*node);

};

ListDG::ListDG()

{

char c1, c2;

int v, e;

int i, p1, p2;

ENode \*node1, \*node2;

cout<<"请输入有向图的顶点数和边数："<<endl;

cin>>mVexNum>>mEdgNum;

cout<<"初始化邻接表的顶点："<<endl;

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

{

cout << "顶点(" << i << "): ";

cin>>mVexs[i].data;

mVexs[i].firstEdge = NULL;

}

// 初始化"邻接表"的边

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

{

// 读取边的起始顶点和结束顶点

cout << "edge(" << i << "): ";

cin>>c1>>c2;

p1 = getPosition(c1);

p2 = getPosition(c2);

// 初始化node1

node1 = new ENode();

node1->ivex = p2;

// 将node1链接到"p1所在链表的末尾"

if(mVexs[p1].firstEdge == NULL)

mVexs[p1].firstEdge = node1;

else

linkLast(mVexs[p1].firstEdge, node1);

}

}

/\*

\* 创建邻接表对应的图(用已提供的数据)

\*/

ListDG::ListDG(char vexs[], int vlen, char edges[][2], int elen)

{

char c1, c2;

int i, p1, p2;

ENode \*node1, \*node2;

// 初始化"顶点数"和"边数"

mVexNum = vlen;

mEdgNum = elen;

// 初始化"邻接表"的顶点

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

{

mVexs[i].data = vexs[i];

mVexs[i].firstEdge = NULL;

}

// 初始化"邻接表"的边

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

{

// 读取边的起始顶点和结束顶点

c1 = edges[i][0];

c2 = edges[i][1];

p1 = getPosition(c1);

p2 = getPosition(c2);

// 初始化node1

node1 = new ENode();

node1->ivex = p2;

// 将node1链接到"p1所在链表的末尾"

if(mVexs[p1].firstEdge == NULL)

mVexs[p1].firstEdge = node1;

else

linkLast(mVexs[p1].firstEdge, node1);

}

}

/\*

\* 析构函数

\*/

ListDG::~ListDG()

{

}

/\*

\* 将node节点链接到list的最后

\*/

void ListDG::linkLast(ENode \*list, ENode \*node)

{

ENode \*p = list;

while(p->nextEdge)

p = p->nextEdge;

p->nextEdge = node;

}

/\*

\* 返回ch的位置

\*/

int ListDG::getPosition(char ch)

{

int i;

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

if(mVexs[i].data==ch)

return i;

return -1;

}

/\*

\* 深度优先搜索遍历图的递归实现

\*/

void ListDG::DFS(int i, int \*visited)

{

ENode \*node;

visited[i] = 1;

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

node = mVexs[i].firstEdge;

while (node != NULL)

{

if (!visited[node->ivex])

DFS(node->ivex, visited);

node = node->nextEdge;

}

}

/\*

\* 深度优先搜索遍历图

\*/

void ListDG::DFS()

{

int i;

int visited[MAX]; // 顶点访问标记

// 初始化所有顶点都没有被访问

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

visited[i] = 0;

cout << "DFS: ";

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

{

if (!visited[i])

DFS(i, visited);

}

cout << endl;

}

/\*

\* 广度优先搜索（类似于树的层次遍历）

\*/

void ListDG::BFS()

{

int head = 0;

int rear = 0;

int queue[MAX]; // 辅组队列

int visited[MAX]; // 顶点访问标记

int i, j, k;

ENode \*node;

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

visited[i] = 0;

cout << "BFS: ";

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

{

if (!visited[i])

{

visited[i] = 1;

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

queue[rear++] = i; // 入队列

}

while (head != rear)

{

j = queue[head++]; // 出队列

node = mVexs[j].firstEdge;

while (node != NULL)

{

k = node->ivex;

if (!visited[k])

{

visited[k] = 1;

cout << mVexs[k].data << " ";

queue[rear++] = k;

}

node = node->nextEdge;

}

}

}

cout << endl;

}

/\*

\* 打印邻接表图

\*/

void ListDG::print()

{

int i,j;

ENode \*node;

cout << "List Graph:" << endl;

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

{

cout << i << "(" << mVexs[i].data << "): ";

node = mVexs[i].firstEdge;

while (node != NULL)

{

cout << node->ivex << "(" << mVexs[node->ivex].data << ") ";

node = node->nextEdge;

}

cout << endl;

}

}

int main()

{

ListDG\* pG;

pG = new ListDG();

pG->print(); // 打印图

pG->DFS(); // 深度优先遍历

pG->BFS(); // 广度优先遍历

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

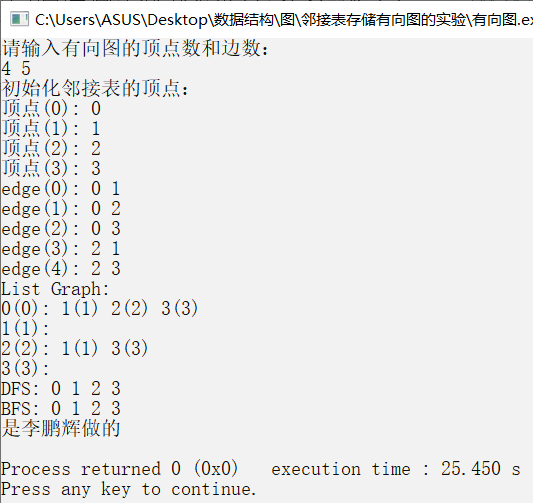


图6-2 邻接表存储有向图的实现

### 遇到的问题和解决的办法

用邻接表存储的有向图，因为在建立邻接表的时候，建的链表都是头插入的，所以遍历出来的顺序和原先设想的是相反的，就是从下往上的。

## 6.3 Prim算法的实现

说明：实现Prim算法，对于给定的连通网，输出最小生成树中的各边。

### 实验代码

#include <iostream>

#include <cstdio>

#include <vector>

#include <queue>

#include <cstring>

using namespace std;

const int maxn=1005;

struct Edge{

int to,dist;

Edge(int t,int d):to(t),dist(d){}

bool operator<(const Edge& e)const{

return dist>e.dist; }

};

int n,m;

bool vis[maxn];

vector<Edge> g[maxn];

priority\_queue<Edge> que;

void prim(){

memset(vis,0,sizeof(vis));

while(que.size()) que.pop();

for(int i=0;i<g[0].size();++i) que.push(g[0][i]);

vis[0]=true;

int ans=0;

for(int cnt=1;cnt<n;++cnt){

while(que.size() && vis[que.top().to]) que.pop();

Edge e=que.top();

ans+=e.dist;

int v=e.to;

vis[v]=true;

for(int i=0;i<g[v].size();++i){

if(!vis[g[v][i].to]) que.push(g[v][i]);

}

}

cout<<"prim算法生成的结果为"<<ans<<endl;

}

int main(){

cout<<"请分别输入顶点数和边数："<<endl;

cin>>n>>m;

cout<<"请输入对应的边和权值："<<endl;

for(int i=0;i<n;++i) g[i].clear();

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

int u,v,w;

scanf("%d%d%d",&u,&v,&w);

g[u-1].push\_back(Edge(v-1,w));

g[v-1].push\_back(Edge(u-1,w));

}

prim();

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

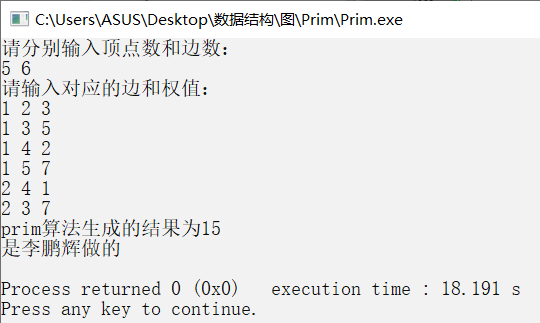


图6-3 Prim算法的实现

### 遇到的问题和解决的办法

Prim算法是求最小生成树的算法，其基本思想是：从一定点出发，找出此点邻接边中最短的边加入对应顶点，再次寻找这两条边的最短邻接边，依次类推，加入所有没有被访问过的边。我们通过这种类比的方法写出了相应代码。

## 6.4 Kruscal算法的实现

说明：实现Kruscal算法，对于给定的连通网，输出最小生成树中的各边。

### 实验代码

#include<iostream>

#include<cstdio>

#include<cstring>

#include<algorithm>

#include<cmath>

#include<queue>

#include<stack>

#include<map>

#include<sstream>

using namespace std;

typedef long long ll;

const int maxn = 3e5 + 10;

const int INF = 1 << 30;

int dir[4][2] = {1,0,0,1,-1,0,0,-1};

int T, n, m, x;

struct edge

{

int u, v, w;

bool operator <(const edge& a)const

{

return w < a.w;

}

};

edge a[maxn];

int par[600], high[600];

//初始化n个元素

void init(int n)

{

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

{

par[i] = i;

high[i] = 0;

}

}

//查询树的根

int Find(int x)

{

return par[x] == x ? x : par[x] = Find(par[x]);//路径压缩

}

void unite(int x, int y)

{

x = Find(x);

y = Find(y);

if(x == y)return;

if(high[x] < high[y])par[x] = y;//y的高度高，将x的父节点设置成y

else

{

par[y] = x;

if(high[x] == high[y])high[x]++;

}

}

bool same(int x, int y)

{

return Find(x) == Find(y);

}

void kruskal(int n, int m)//点数n，边数m

{

int sum\_mst = 0;//mst权值

int num= 0;//已经选择的边的边数

sort(a, a + m);//边进行排序

init(n);//初始化并查集

cout<<"选取的边分别为："<<endl;

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

{

int u = a[i].u;

int v = a[i].v;

if(Find(u - 1) != Find(v - 1))//图最开始的下标是1，并查集是0

{

printf("%d %d %d\n", u, v, a[i].w);

sum\_mst += a[i].w;

num++;

unite(u - 1, v - 1);

}

if(num >= n - 1)break;

}

cout<<"kruskal算法生成的结果为"<<sum\_mst<<endl;

}

int main()

{

cout<<"请分别输入顶点数和边数："<<endl;

cin >> n >> m;

cout<<"请输入对应的边和权值："<<endl;

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

{

cin >> a[i].u >> a[i].v >> a[i].w;

}

kruskal(n, m);

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

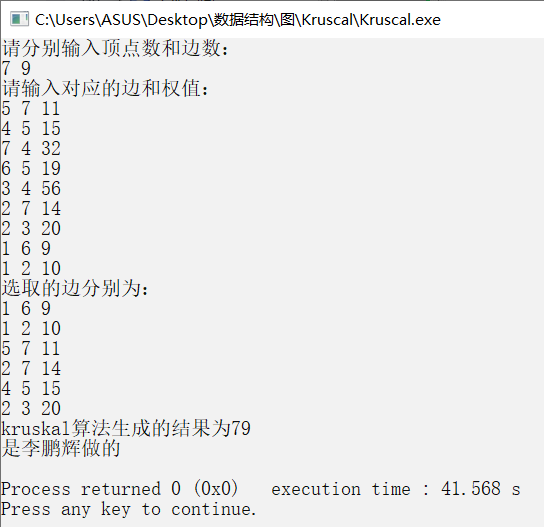


图6-4 Kruscal算法的实现

### 遇到的问题和解决的办法

（1）根据权值从小到大遍历每一条边，把原图分为两个集合，每一次判断这一条边的两个端点是否在同一集合，为真，则表明这个点已经在MST中，由于贪心，所以最优，故跳过；为假，则表明这个点并未在集合之中，所以将该点加入集合中（运用并查集来实现）。

（2）因为一个有n个点的图，最小生成树只需n-1条边，所以标记一下，如果已经选出了n-1条边，break；若最后遍历了整个图，仍为选出n-1条边，输出无解。

## 6.5 Dijkstra算法的实现

说明：实现Dijkstra算法，求源点V0到其它顶点的最短路径长度。

### 实验代码

#include <iostream>

#include <limits>

using namespace std;

struct Node { //定义表结点

int adjvex; //该边所指向的顶点的位置

int weight;// 边的权值

Node \*next; //下一条边的指针

};

struct HeadNode{ // 定义头结点

int nodeName; // 顶点信息

int inDegree; // 入度

int d; //表示当前情况下起始顶点至该顶点的最短路径,初始化为无穷大

bool isKnown; //表示起始顶点至该顶点的最短路径是否已知,true表示已知，false表示未知

int parent; //表示最短路径的上一个顶点

Node \*link; //指向第一条依附该顶点的边的指针

};

//G表示指向头结点数组的第一个结点的指针

//nodeNum表示结点个数

//arcNum表示边的个数

void createGraph(HeadNode \*G, int nodeNum, int arcNum) {

//初始化头结点

for (int i = 0; i < nodeNum; i++) {

G[i].nodeName = i+1; //位置0上面存储的是结点v1,依次类推

G[i].inDegree = 0; //入度为0

G[i].link = NULL;

}

for (int j = 0; j < arcNum; j++) {

int begin, end, weight;

cin >> begin >> end >> weight;

// 创建新的结点插入链接表

Node \*node = new Node;

node->adjvex = end - 1;

node->weight = weight;

++G[end-1].inDegree; //入度加1

//插入链接表的第一个位置

node->next = G[begin-1].link;

G[begin-1].link = node;

}

}

//得到begin->end权重

int getWeight(HeadNode \*G, int begin, int end) {

Node \*node = G[begin-1].link;

while (node) {

if (node->adjvex == end - 1) {

return node->weight;

}

node = node->next;

}

}

//从start开始，计算其到每一个顶点的最短路径

void Dijkstra(HeadNode \*G, int nodeNum, int start) {

//初始化所有结点

for (int i = 0; i < nodeNum; i++) {

G[i].d = INT\_MAX; //到每一个顶点的距离初始化为无穷大

G[i].isKnown = false; // 到每一个顶点的距离为未知数

}

G[start-1].d = 0; //到其本身的距离为0

G[start-1].parent = -1; //表示该结点是起始结点

while(true) {

//==== 如果所有的结点的最短距离都已知, 那么就跳出循环

int k;

bool ok = true; //表示是否全部ok

for (k = 0; k < nodeNum; k++) {

//只要有一个顶点的最短路径未知,ok就设置为false

if (!G[k].isKnown) {

ok = false;

break;

}

}

if (ok) return;

//==========================================

//==== 搜索未知结点中d最小的,将其变为known

//==== 这里其实可以用最小堆来实现

int i;

int minIndex = -1;

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

if (!G[i].isKnown) {

if (minIndex == -1)

minIndex = i;

else if (G[minIndex].d > G[i].d)

minIndex = i;

}

}

//===========================================

G[minIndex].isKnown = true; //将其加入最短路径已知的顶点集

// 将以minIndex为起始顶点的所有的d更新

Node \*node = G[minIndex].link;

while (node != NULL) {

int begin = minIndex + 1;

int end = node->adjvex + 1;

int weight = getWeight(G, begin, end);

if (G[minIndex].d + weight < G[end-1].d) {

G[end-1].d = G[minIndex].d + weight;

G[end-1].parent = minIndex; //记录最短路径的上一个结点

}

node = node->next;

}

}

}

//打印到end-1的最短路径

void printPath(HeadNode \*G, int end) {

if (G[end-1].parent == -1) {

cout << "v" << end;

} else if (end != 0) {

printPath(G, G[end-1].parent + 1); // 因为这里的parent表示的是下标，从0开始，所以要加1

cout << " -> v" << end;

}

}

int main() {

HeadNode \*G;

int nodeNum, arcNum;

cout<<"请分别输入顶点数和边数："<<endl;

cin >> nodeNum >> arcNum;

cout<<"请输入对应的边和权值："<<endl;

G = new HeadNode[nodeNum];

createGraph(G, nodeNum, arcNum);

Dijkstra(G, nodeNum, 1);

cout << "打印从v1开始所有的最短路径" << endl;

for (int k = 2; k <= nodeNum; k++) {

cout << "v1到v" << k << "的最短路径为" << G[k-1].d << ": ";

printPath(G, k);

cout << endl;

}

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

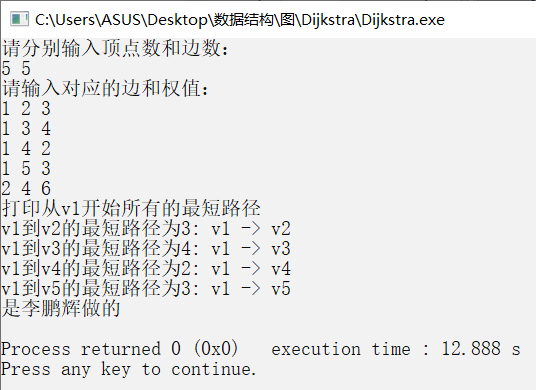


图6-5 Dijkstra算法的实现

### 遇到的问题和解决的办法

（1）首先，引进一个辅助向量D，它的每个分量D[i]表示当前所找到的从始点v到每个终点vi的最短路径的长度。如D[3]=2表示从始点v到终点3的路径相对最小长度为2。

（2）这里强调相对就是说在算法过程中D[i]的值是在不断逼近最终结果但在过程中不一定就等于最短路径长度。它的初始状态为：若从v到vi有弧，则D[i]为弧上的权值；否则置D[i]为∞。显然，长度为 D[j]=Min{D[i]|vi∈V}的路径就是从v出发的长度最短的一条最短路径。此路径为(v,vj)。

（3）Dijkstra不能直接解决带负权值的图，可以用SPFA算法。

（4）用下标为1开始写可以防止溢出。

## 6.6 Floyd算法的实现

说明：实现Floyd算法，对于给定的有向网，求每一对顶点之间的最短路径长度。

### 实验代码

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cmath>

using namespace std;

int main() {

int e[10][10],k,i,j,n,m,t1,t2,t3;

int inf=99999999; //用inf(infinity的缩写)存储一个我们认为的正无穷值

//读入n和m，n表示顶点个数，m表示边的条数

cout<<"请分别输入顶点数和边数："<<endl;

scanf("%d %d",&n,&m);

//初始化

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

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

if(i==j)

e[i][j]=0;

else

e[i][j]=inf;

//读入边

cout<<"请输入对应的边和权值："<<endl;

for(i=1; i<=m; i++) {

scanf("%d %d %d",&t1,&t2,&t3);

e[t1][t2]=t3;

}

//Floyd-Warshall算法核心语句

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

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

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

if(e[i][j]>e[i][k]+e[k][j] )

e[i][j]=e[i][k]+e[k][j];

//输出最终的结果

cout<<"最终矩阵为："<<endl;

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

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

printf("%10d",e[i][j]);

}

printf("\n");

}

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

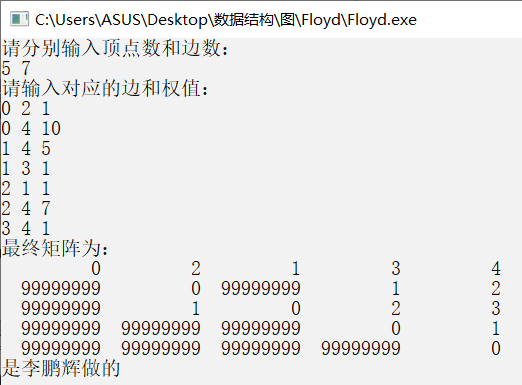


图6-6 Floyd算法的实现

### 遇到的问题和解决的办法

（1）首先记录两点间无其他中间顶点的距离（Vi---Vj)，加一个顶点为中间点，记录加一个顶点后两点间的最短距离，以此类推，加完N个顶点后两点间的最短距离即可求出。

（2）用三重循环实现。用二维数组表示顶点到顶点的距离。如D[i][j]表示从i到j点的距离。第一层循环依次加顶点，第二层循环遍历Vi点，第三层循环遍历Vj点。

（3）Dijkstra不能解决带有负权值的图，Floyd可以解决带有负权值的图，但不能解决带有负权回路的图。

（4）可以增加一个path二维数组存储路径，利用递归的方法生成最短路径。

## 6.7 拓扑排序的实现

说明：对于有向图，求解拓扑序列，不存在拓扑序列的，给出说明。

### 实验代码

#include <iostream>

#include <stack>

using namespace std;

#define maxnum 20

struct node { //边表节点的定义

int vex;

node\* next; //指向下一个边表节点

};

struct vernode {

char vex; //存放顶点信息

node\* first;//指向第一个边表节点

int indegree;

};

struct Graph {

vernode v[maxnum];

int vnums,enums;

};

void CreateGraph(Graph &g,int n,int e) { //创建有向图的邻接表

int i,j,k;

g.vnums=n;

g.enums=e;

cout<<"依次输入顶点的编号："<<endl;

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

cin>>g.v[i].vex;

g.v[i].first=NULL; //初始化为空

}

cout<<"依次输入边的起始点和终止点:"<<endl;

for(k=1; k<=e; k++) {

node \*p;

cin>>i>>j;

p=new node();

p->vex=j;

p->next=g.v[i].first;

g.v[i].first=p; //头插法

}

}

void TopoSort(Graph &g) {

std::stack<int> s;

node \*p;

int counts=0;

int i,j;

for(i=1; i<=g.vnums; i++)

g.v[i].indegree=0; //初始化入度为0

for(i=1; i<=g.vnums; i++) { //计算各个顶点的入度

p=g.v[i].first;

while(p) {

g.v[p->vex].indegree++; //计算入度

p = p->next;

}

}

for(i=1; i<=g.vnums; i++)

if(g.v[i].indegree==0)

s.push(i); //将度为 0 的顶点入栈,这里进栈的是入度为 0 的顶点在数组的位置

cout<<"拓扑排序的结果："<<endl;

while(!s.empty()) {

j=s.top();

s.pop(); //出栈

cout<<g.v[j].vex<<" "; //将栈顶的元素出栈且输出

counts++;

p=g.v[j].first; //让p指向入度为0 的第一个节点

while(p) {

g.v[p->vex].indegree--;

if(g.v[p->vex].indegree==0)

s.push(p->vex);

p=p->next;

}

}

if(counts<g.vnums)

cout<<"图中有环"<<endl;

}

int main() {

Graph g;

int n,e;

cout<<"输入顶点个数和边数: "<<endl;

cin>>n>>e;

CreateGraph(g,n,e);

TopoSort(g);

cout<<endl;

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

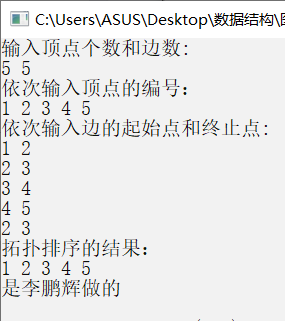


图6-7 拓扑排序的实现

### 遇到的问题和解决的办法

（1）从有向图中选取一个没有前驱的顶点，并输出之；  
（2）从有向图中删去此顶点以及所有以它为尾的弧；  
（3）重复上述两步，直至图空，或者图不空但找不到无前驱的顶点为止。没有前驱 -- 入度为零，删除顶点及以它为尾的弧-- 弧头顶点的入度减1。

## 6.8 选址问题

说明：图中的四个顶点为四个小区，有向边上的权值表示小区之间的距离，现在要在四个小区中选择一个小区新建一所活动中心。怎么选取合适？

### 实验代码

#include <iostream>

using namespace std;

#define MAXV 50

#define INF 32767

typedef int InfoType;

//邻接矩阵存储方法

typedef struct {

int no;

InfoType info;

} VertexType;

typedef struct {

int edges[MAXV][MAXV];

int n, e;

VertexType vexs[MAXV];

} MGraph;

void Ppath(int path[], int i, int v) {

int k;

k = path[i];

if (k == v) return;

Ppath(path, k, v);

}

int dis[5][5]= {

{0,32767,32767,32767},

{32767,0,32767,32767},

{32767,32767,0,32767},

{32767,32767,32767,0}

} ;

int biaoji1 = 0, biaoji2 = 0;

void Dispath(int dist[], int path[], int s[], int n, int v) {

int i;

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

if (i == v) continue;

if (s[i] == 1) {

// cout << "从" << v << "到" << i << "的最短路径为：" << dist[i] << " "<<endl;

dis[v][i]=dist[i];

Ppath(path, i, v);

if (biaoji1 != 5) {

biaoji2 += dist[i];

biaoji1++;

} else {

// cout << "和为：" << " " << biaoji2;

biaoji1 = 0;

biaoji2 = 0;

}

}

// else

// cout << "从" << v << "到" << i << "不存在的路径" << endl;

}

}

void Dijkstra(MGraph g, int v) {

int dist[MAXV], path[MAXV];

int s[MAXV];

int mindis, i, j, u;

for (i = 0; i < g.n; i++) {

dist[i] = g.edges[v][i];

s[i] = 0;

if (g.edges[v][i] < INF) path[i] = v;

else path[i] = -1;

}

s[v] = 1;

path[v] = 0;

for (i = 0; i < g.n; i++) {

mindis = INF;

for (j = 0; j < g.n; j++) {

if (s[j] == 0 && dist[j] < mindis) {

u = j;

mindis = dist[j];

}

}

s[u] = 1;

for (j = 0; j < g.n; j++) {

if (s[j] == 0) {

if (g.edges[u][j] < INF && dist[u] + g.edges[u][j] < dist[j]) {

dist[j] = dist[u] + g.edges[u][j];

path[j] = u;

}

}

}

}

Dispath(dist, path, s, g.n, v);

}

//主函数

int main() {

int i, j, n=4;

MGraph g;

cout << "带权有向图的顶点个数为4"<<endl;

cout << "带权有向图的邻接矩阵：" << endl;

printf("\tA\tB\tC\tD\n");

printf("A\t0\t5\tINF\t7\n");

printf("B\tINF\t0\t4\t2\n");

printf("C\t3\t3\t0\t2\n");

printf("D\tINF\tINF\t1\t0\n");

g.edges[0][0]=0;

g.edges[0][1]=5;

g.edges[0][2]=32767;

g.edges[0][3]=7;

g.edges[1][0]=32767;

g.edges[1][1]=0;

g.edges[1][2]=4;

g.edges[1][3]=2;

g.edges[2][0]=3;

g.edges[2][1]=3;

g.edges[2][2]=0;

g.edges[2][3]=2;

g.edges[3][0]=32767;

g.edges[3][1]=32767;

g.edges[3][2]=1;

g.edges[3][3]=0;

g.n = n;

cout << "采用狄克斯特拉算法得到的最短路径矩阵为：" << endl;

for (i = 0; i < n; i++) Dijkstra(g, i);

printf("\tA\tB\tC\tD\t\n") ;

for(int i=0; i<4; i++) {

printf("%c",'A'+i);

for(int j=0; j<4; j++) {

printf("\t%d",dis[i][j]);

}

cout<<endl;

}

int mmin=100000,index=-1;

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

int sum=dis[i][0]+dis[i][1]+dis[i][2]+dis[i][3]+dis[0][i]+dis[1][i]+dis[2][i]+dis[3][i];

if(sum<mmin) {

mmin=sum;

index=i;

}

}

printf("应该建在%c上\n",'A'+index);

cout<<endl;

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

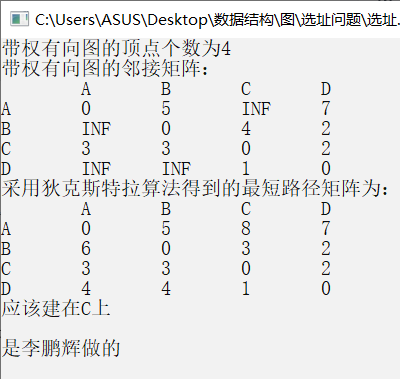


图6-8 选址问题

### 遇到的问题和解决的办法

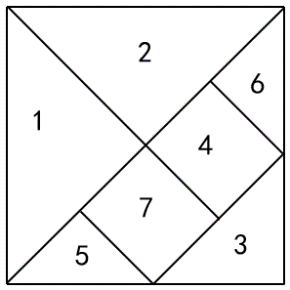
（1）题目已给条件，所以没有输入操作，根据所给图可以写出有向图的邻接矩阵。

（2）使用dijkstra算法算出有向图中任意两点之间的最短距离，用dis数组存起来。

（3）对于最合适的方案我们理解为某点到其他点的距离之和与其他点通往该点的距离之和的和，所以需要将矩阵中的对应行和对应列的值求和，算出距离和最短的点即为最合适的点。

## 6.9 七巧板涂色问题

说明：对于如下图所示的七巧板涂色，要求相邻的板颜色不同，并且总的颜色数尽可能少。



### 实验代码

#include <iostream>

using namespace std;

/\*七巧板块数\*/

const int n = 7;

/\*邻接矩阵，1表示相邻，0表示不相邻\*/

const int data[n][n] = {

{0, 1, 0, 0, 1, 0, 1},

{1, 0, 0, 1, 0, 1, 0},

{0, 0, 0, 0, 1, 0, 1},

{0, 1, 1, 0, 0, 1, 1},

{1, 0, 0, 0, 0, 0, 1},

{0, 1, 0, 1, 0, 0, 0},

{1, 0, 1, 1, 1, 0, 0}

};

/\*七巧板的颜色，0表示没有涂色\*/

int color[n]= {0, 0, 0, 0, 0, 0, 0};

/\*记录涂色方案的种数\*/

static int total;

/\*输出涂色方案\*/

void PrintColor() {

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

cout<<color[i]<<" ";

}

total++;

cout<<endl;

}

/\*判断与周围的七巧板颜色是否相同\*/

int ColorSame(int s) {

int flag = 0;

for(int i = 0; i < s; i++) {

/\*判断相邻的块颜色是否相同\*/

if(data[i][s] == 1 && color[i] == color[s])

flag = 1;

}

return flag;

}

/\*递归查找涂色方案\*/

void Painting(int s) {

/\*s=0~6，如果s=7说明已经涂完\*/

if(s == n)

PrintColor();

else {

/\*1、2、3、4代表四种颜色\*/

for(int i = 1; i <= 4; i++) {

color[s] = i;

/\*如果七巧板s和相邻的七巧板颜色不同，则递归涂七巧板s+1\*/

if(ColorSame(s) == 0)

Painting(s + 1);

}

}

}

int main() {

total = 0;

Painting(0);

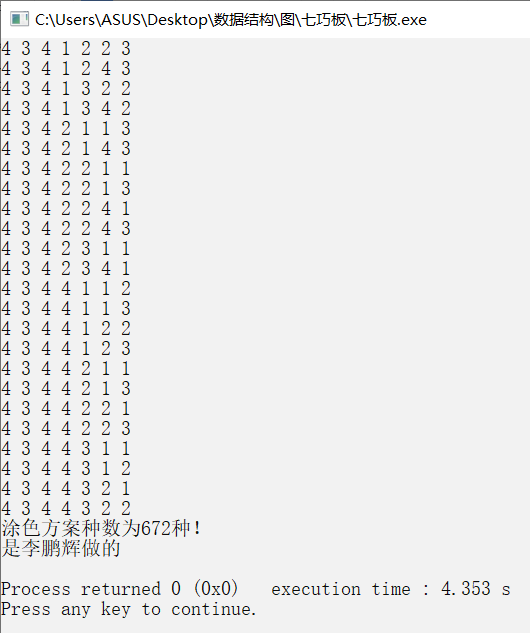
cout<<"涂色方案种数为"<<total<<"种！"<<endl;

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果



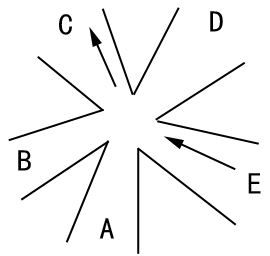
### 遇到的问题和解决的办法

（1）将实际问题的图抽象成无向边图，用邻接矩阵存储。

（2）利用递归查找涂色方案。

## 6.10 五叉路口交通灯问题

说明：对于如下图所示的五叉路口，设计交通灯方案，总体可行的方案较多，只选择其中的六种输出即可。



说明：不能从C到其它路口，C路口为单行道，只出不进。即和C路口相关的可通行路线包括BC、AC、EC、DC，但是C到其它的路口是不通的。

### 实验代码

#include <iostream>

using namespace std;

/\*路线条数\*/

const int n = 13;

/\*路线\*/

const string path[n] = {"AB", "AC", "AD", "BA", "BC", "BD", "DA", "DB", "DC", "EA", "EB", "EC", "ED"};

/\*邻接矩阵表示各路线之间的关系，1表示路线不能同时通行，0表示路线可以同时通行\*/

const int data[n][n] = {

{0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0},

{0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0},

{1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0},

{1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0},

{0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0},

{0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

};

/\*路线对应的交通灯颜色，0表示没有对应颜色\*/

int color[n] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

/\*记录交通灯方案的种数\*/

static int total;

/\*输出交通灯颜色方案\*/

void PrintColor() {

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

cout<<path[i]<<" ";

cout<<color[i]<<" ";

}

total++;

if(total == 10) {

cout<<endl;

cout<<"是李鹏辉做的"<<endl;

exit(0);

}

cout<<endl;

}

/\*判断与不能同时通行的路线的颜色是否相同\*/

int ColorSame(int s) {

int flag = 0;

for(int i = 0; i < s; i++) {

if(data[i][s] == 1 && color[i] == color[s])

flag = 1;

}

return flag;

}

/\*递归查找交通灯方案\*/

void Painting(int s) {

/\*s = 0~12，如果s = 13说明已经涂完\*/

if(s == n)

PrintColor();

else {

/\*1、2、3、4代表四种颜色\*/

for(int i = 1; i <= 4; i++) {

color[s] = i;

/\*如果第s条路线的交通灯颜色符合要求，则递归处理第s+1条\*/

if(ColorSame(s) == 0)

Painting(s + 1);

}

}

}

int main() {

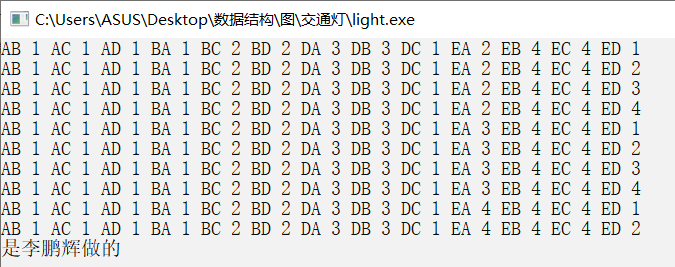
total = 0;

Painting(0);

return 0;

}

### 实验运行结果



### 遇到的问题和解决的办法

（1）交通方案太多导致运行崩溃，我们只计算其中十种方案。

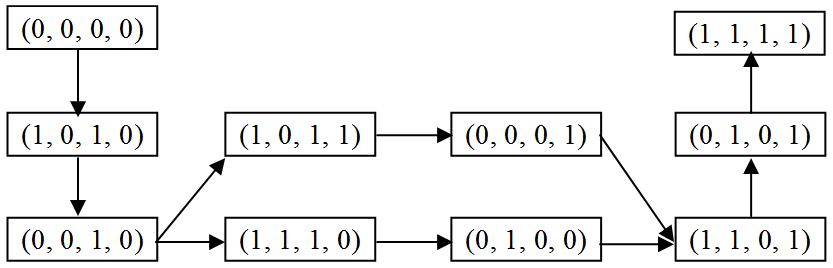
（2）将本问题与七巧板涂色问题类比去解决。

（3）利用递归查找交通灯方案。

## 6.11 农夫过河问题

说明：一位农夫带着一只狼、一只羊和一棵白菜，身处河的南岸，他要把这些东西全部运到河的北岸。他面前只有一条小船，船只能容下他和一件物品，只有农夫自己才能撑船。如果农夫在场，则狼不能吃羊，羊不能吃白菜；如果农夫不在场，则狼会吃羊，羊会吃白菜。所以在任何情况下，农夫不能留下狼和羊单独离开，也不能留下羊和白菜单独离开，按要求设计过河方案。

**说明：教材P208图6-64有误，正确的如下图所示。**



### 实验代码

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

typedef int DataType;//

//顺序队列：类型和界面函数声明

struct SeqQueue {

// 顺序队列类型定义

int MAXNUM; // 队列中最大元素个数

int f, r;//定义头结点front和尾结点rear

DataType \*q;

};

typedef struct SeqQueue \*PSeqQueue; // 顺序队列类型的指针类型

//创建一个空队列

PSeqQueue createEmptyQueue\_seq(int m) {

PSeqQueue queue = (PSeqQueue)malloc(sizeof(struct SeqQueue));

if (queue != NULL) {

queue->q = (DataType\*)malloc(sizeof(DataType) \*m);

if (queue->q) {

queue->MAXNUM = m;

queue->f = 0;

queue->r = 0;

return (queue);

} else

free(queue);

}

printf("Out of space!!\n"); // 存储分配失败

return NULL;

}

//判断队列是否为空

int isEmptyQueue\_seq(PSeqQueue queue) {

return (queue->f == queue->r);

}

// 在队尾插入元素x，此队列队尾一个元素空出来

void enQueue\_seq(PSeqQueue queue, DataType x) {

if ((queue->r + 1) % queue->MAXNUM == queue->f)

printf("Full queue.\n");

else {

queue->q[queue->r] = x;

queue->r = (queue->r + 1) % queue->MAXNUM;

}

}

// 删除队列头部元素

void deQueue\_seq(PSeqQueue queue) {

if (queue->f == queue->r)

printf("Empty Queue.\n");

else

queue->f = (queue->f + 1) % queue->MAXNUM;

}

DataType frontQueue\_seq(PSeqQueue queue) {

if (queue->f == queue->r)

printf("Empty Queue.\n");

else

return (queue->q[queue->f]);

}

//个体状态判断函数

int farmer(int location) {

//判断农夫的位置

return (0 != (location &0x08));

}

int wolf(int location) {

//判断狼的位置

return (0 != (location &0x04));

}

int cabbage(int location) {

//判断白菜的位置

return (0 != (location &0x02));

}

int goat(int location) {

//判断羊的位置

return (0 != (location &0x01));//

}

//安全状态的判断函数

// 若状态安全则返回true

int safe(int location) {

// 羊吃白菜

if ((goat(location) == cabbage(location)) && (goat(location) != farmer(location)))

return 0;

// 狼吃羊

if ((goat(location) == wolf(location)) && (goat(location) != farmer(location)))

return 0;

return 1; // 其他状态是安全的

}

void bin\_print(int num) {

char tmp[4];

int i;

for (i = 0; i < 4; ++i) {

tmp[i] = num & 0x01;

num >>= 1;

}

for (i = 3; i >= 0; --i)

putchar((tmp[i] == 0)?'0':'1');

return;

}

int main() {

int i, movers, location, newlocation;

int route[16]; //用于记录已考虑的状态路径

PSeqQueue moveTo; //用于记录可以安全到达的中间状态

moveTo = createEmptyQueue\_seq(20); //创建空队列

enQueue\_seq(moveTo, 0x00); //初始状态进队列,0x00入队

for (i = 0; i < 16; i++)

route[i] = -1;

//准备数组route初值

route[0] = 0;

//该算法仅打印一种路径，起思想是dfs

while (!isEmptyQueue\_seq(moveTo) && (route[15] == - 1)) {

location = frontQueue\_seq(moveTo); //取队头状态为当前状态

deQueue\_seq(moveTo);//删除moveTo结点，moveTo结点出栈

//考虑各种物品移动

for (movers = 1; movers <= 8; movers <<= 1)

//农夫与移动的物品在同一侧

if ((0 != (location & 0x08)) == (0 != (location & movers))) { //判断农夫与移动的物品是否在同一侧

newlocation = location ^ (0x08 | movers); //计算新状态，异或代表把船上的（0x08|movers）从一个岸移到另一个岸；（0x08|movers）代表船上有农夫和movers代表的东西（movers代表且仅代表一个东西）

//新状态安全且未处理

if (safe(newlocation) && (route[newlocation] == -1)) {

route[newlocation] = location; //记录新状态的前驱

enQueue\_seq(moveTo, newlocation); //新状态入队

}

}

}

// 打印出路径

if (route[15] != -1)

//到达最终状态

{

printf("The reverse path is : \n");

for (location = 15; location >= 0; location = route[location]) {

printf("The location is : %2d, ", location);

bin\_print(location);

putchar('\n');

if (location == 0) {

cout<<"是李鹏辉做的"<<endl;

exit(0);

}

}

} else

printf("No solution.\n");

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

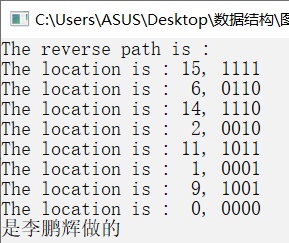


图6-9 农夫过河问题

### 遇到的问题和解决的办法

（1）需要列举的所有状态(二进制0000 ~ 1111)一共16种，所以可以构造一个包含16个元素的整数顺序表来满足以上的要求

（2）从初始状态二进制0000(全部在河的南岸) 出发，寻找一种全部由安全状态构成的状态序列，它以二进制1111(全部到达河的北岸) 为最终目标，并且在序列中的每一个状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达。

（3）要求在序列中不应该出现重复的状态。为了实现广度优先搜索，算法中需要使用了一

个整数队列moveTo，它的每个元素表示一个可以安全到达的中间状态。另外还需要一个数据结构记录已被访问过的各个状态，以及已被发现的能够到达当前这个状态的路径。

# 第7章 查找

输入可有可无，如果没有输入，需要在main()函数中设置好数据。

## 7.1 线性查找的实现

### 实验代码

#include<iostream>

using namespace std;

#define MAXSIZE 100

typedef int ElemType;

typedef struct

{

ElemType \*R; //表基址

int length; //表长

}SSTable;

void Create\_Seq(SSTable &ST,ElemType key)

{

cout<<"请输入建表元素个数:";

int n;

cin>>n;

ST.length=n;

ST.R= new ElemType[MAXSIZE];

cout<<"请输入"<<n<<"个建表元素:";

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

{

cin>>ST.R[i];

}

}

int Search\_Seq(SSTable ST,ElemType key)

{

//若成功返回其位置信息，否则返回0

ST.R[0]=key;

int i;

for(i=ST.length;ST.R[i]!=key;--i);

return i;

}

int main()

{

SSTable ST;

int key,pos;

Create\_Seq(ST,key);//建表

cout<<"请输入要查询的数:";

cin>>key;

//查找

pos=Search\_Seq(ST,key);

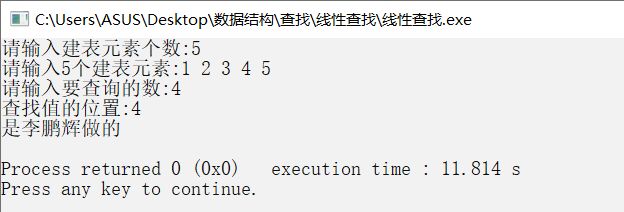
cout<<"查找值的位置:"<<pos<<endl;

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果



### 遇到的问题和解决的办法

（1）问题：线性查找的思路是什么？

解决方法：核心：从数据的最后一个元素开始，依次比较，直到找到目标数据或查找失败。

1. 从表中的最后一个元素开始，依次与关键字比较。

2. 若某个元素匹配关键字，则查找成功。

3. 若查找到第一个元素还未匹配关键字，则查找失败，返回0。

（2）问题：线性查找的时间复杂度是多少？

解决方法：顺序查找平均关键字匹配次数为表长的一半，其时间复杂度为O(n)。

（3）问题：如果顺序表中有重复的数据怎么办？

解决方法：由于该查找方法是从后往前查找，因此返回的位置是关键字在在顺序表中最后一次出现的位置。

## 7.2 折半查找的实现

### 实验代码

#include<iostream>

using namespace std;

int Bisearch(int data[],int x,int low ,int high)

{

if(low>high)

{

return -1;

exit(0);

}

int mid=(low+high)/2;

if (x == data[mid])

{

return mid;

exit(0);

}

if ( x < data[mid] )

{

Bisearch(data,x,low,mid-1);

}

else

{

Bisearch(data,x,mid+1,high);

}

}

int main()

{ int i=0;

int length=10;

int data[10];

cout<<"请输入10个元素：";

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

cin>>data[i];

int x;

cout<<"请输入你要查找的元素：";

cin>>x;

int loaction=Bisearch(data,x,0,9);

if( loaction==-1)

{

cout<<"查找失败，没有你要查找的值"<<endl;

}

else

{

cout<<"你要查找的值"<<x<<"的位置在第"<<loaction+1<<"个位置"<<endl;

}

cout<<"是李鹏辉做的"<<endl;

return 0;

}

### 实验运行结果

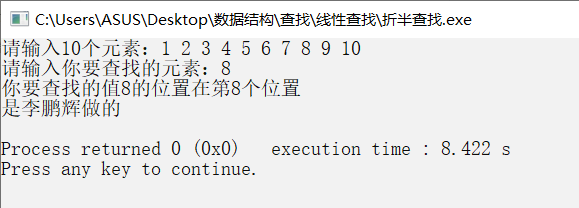


图7-2折半查找的实现

### 遇到的问题和解决的办法

（1）问题：折半查找的算法要求是什么？

解决方法：1.必须采用顺序存储结构。2.必须按关键字大小有序排列。

（2）问题：折半查找的思路是什么？

解决方法：折半查找的思路是，在有序表中取中间元素作为比较对象，若给定值与中间元素的关键码相等，则查找成功；若给定值小于中间元素的关键码，则在中间元素的左半区继续查找；同理，如果找不到在右半部查找。不断重复上述的查找过程，直到查找成功，或所查找的区域无数据元素，查找失败。

（3）问题：使用折半查数据怎么才能避免数组不溢出？

解决方法：数据地址会变化，在固定范围内不要继续往下地址递增就不会溢出，如果自身申请的数组过小也会溢出。

## 7.3 二叉排序树基本操作的实现

说明：实现类BiSortTree，基本操作包括构造函数、插入元素、查找、中序遍历，并验证（选做二叉排序树中结点的删除）。

### 实验代码

#include<iostream>

#include<stdlib.h>

using namespace std;

#define N 10000

typedef int Status; // Status 相当于 int

typedef int TElemType; //TElemType 相当于 int

typedef struct BiTNode{

TElemType data; //结点数据域

struct BiTNode \*lchild,\*rchild; //左右孩子指针

}BiTNode,\*BiTree;

Status SearchBST(BiTree T,int key,BiTree f,BiTree \*p) //二叉排序树的查找

{

if(!T) //查找不成功

{

\*p = f;

return 0;

}

else if(T->data == key) //查找成功

{

\*p = T;

return 1;

}

else if (key < T->data)

{

return SearchBST(T->lchild, key, T, p); //在左子树继续查找

}

else

{

return SearchBST(T->rchild, key, T, p); //在右子树继续查找

}

}

Status InsertBST (BiTree \*T,int key) //二叉排序树的插入

{

BiTree p, s;

if(!SearchBST(\*T, key, NULL, &p)) //查找二叉排序树中是否已经存在key值

{

s = new BiTNode;

s->data = key;

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

if(!p) //若二叉排序树为空，则插入根节点

{

\*T = s;

}

else if(key < p->data)

p->lchild = s;

else

p->rchild = s;

return 1;

}

else

return 0;

}

Status Delete(BiTree \*p) //二叉排序树的删除

{

BiTree q,s;

if( (\*p) -> rchild==NULL) //无右，以左子树替换

{

q=\*p;

\*p=(\*p)->lchild;

free(q);

}

else if((\*p)->lchild==NULL) //无左

{

q=\*p;

\*p=(\*p)->rchild;

free(q);

}

else

{

q=\*p;

s=(\*p)->lchild; //左一下

while(s->rchild) //右到底

{

q=s;

s=s->rchild;

}

(\*p)->data=s->data; //前驱替换删除节点

if(q!=\*p)

q->rchild=s->lchild;

else

q->lchild=s->lchild;

free(s);

}

return 1;

}

Status DeleteBST(BiTree \*T,int key)

{

if(!\*T) //不存在与key相等的元素

return 0;

else

{

if(key==(\*T)->data)

return Delete(T);

else if(key<(\*T)->data)

return DeleteBST(&(\*T)->lchild,key);

else

return DeleteBST(&(\*T)->rchild,key);

}

}

Status InOrderTraverse(BiTree T) //递归中序遍历

{

if(T==NULL) return 1;

else{

InOrderTraverse(T->lchild); //中序遍历左子树

cout << T->data << " "; //访问根结点

InOrderTraverse(T->rchild); //中序遍历右子树

}

}

int main()

{

int i,num;

BiTree T=NULL;

int a[N];

int x;

cout << "请输入要建立二叉排序树的个数:" ;

int n;

cin >> n;

cout << endl << "请输入" << n << "个数字：" << endl;

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

{

scanf("%d",&a[i]);

}

cout << endl;

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

{

InsertBST(&T,a[i]);

}

cout << "遍历二叉树：";

InOrderTraverse(T); cout << endl << endl;

cout << "请输入你要插入的元素:";

cin >> x;

if(InsertBST(&T,x))

cout << "插入元素成功！！！"<< endl;

else

cout << "插入元素失败，因为树中已经有目标元素！！" << endl;

cout << "遍历二叉树：";

InOrderTraverse(T); cout << endl << endl;

cout << "请输入你要删除的元素：";

cin >> x;

if(DeleteBST(&T,x))

cout << "删除元素成功！！！"<< endl;

else

cout << "删除元素失败，因为树中无目标元素！！" << endl;

cout << "遍历二叉树：";

InOrderTraverse(T); cout << endl << endl;

return 0;

}

### 实验运行结果



图7-3 二叉排序树基本操作的实现

### 遇到的问题和解决的办法

（1）问题：二叉排序树查找的思路是什么？

解决方法：1.查找最小值的思路即一路向左。2.查找最大值的思路即一路向右。3.查找某个数的位置的思路，根节点不为空：1）如果根节点key==查找key，返回true。2）如查找key<根节点key，在左子树查找。3）如查找key>根节点key，在右子树查找。

（2）问题：二叉树排序插入的思路是什么？

解决方法：1. 如果树为空的话，直接插入该元素。2. 若节点不为空，按照查找的方式，查找应该插入的位置。

（3）问题：二叉树排序删除有几种情况？

解决方法：1. 要删除的是叶节点：直接删除，再修改其父节点指向叶节点的指针，置为NULL。2. 要删除的节点只有一个子节点：将其父节点指向待删除节点的指针指向待删除节点的子节点。3. 要删除的节点有左右节点：用另一个节点替代被删除节点，如左边树的最大元素，或者右边树的最小元素。而且替代节点是一定没有两个子节点的，所以可以将问题简单化。

## 7.4 散列查找的实现

### 实验代码

#include<iostream>

#include <stdio.h>

#include <stdlib.h>

#define OK 1

#define ERROR 0

#define SUCCESS 1

#define UNSUCCESS 0

#define HASHSIZE 7 //定义散列表表未数组的长度

#define NULLKEY -32768

typedef struct

{

int \*elem; //数据元素存储基地址，动态分配数组

}HashTable;

int m = HASHSIZE; //散列表长，全局变量

//初始化散列表

int InitHashTable(HashTable \*h)

{

int i;

h->elem = (int \*)malloc(sizeof(int) \* m);

if (h->elem == NULL)

{

fprintf(stderr, "malloc() error.\n");

return ERROR;

}

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

{

h->elem[i] = NULLKEY;

}

return OK;

}

//散列函数

int Hash(int key)

{

return key % m; //除留余数法

}

//插入关键字进散列表

void InsertHash(HashTable \*h, int key)

{

int addr = Hash(key); //求散列地址

int i;

for (i = 1; h->elem[addr] != NULLKEY;i++) //如果不为空，则冲突

{

addr = (addr + i) % m; //开放地址法的线性探测

}

h->elem[addr] = key; //直到有空位后插入关键字

}

//散列表查找关键字

int SearchHash(HashTable h, int key)

{

int addr = Hash(key); //求散列地址

int i;

for (i = 1; h.elem[addr] != key;i++) //如果所查关键字不在除留余数法的列表中，则冲突

{

addr = (addr + i) % m; //开放地址法的线性探测

if (h.elem[addr] == NULLKEY || addr == Hash(key))

{

//如果位置为空或循环回原点

printf("查找失败, %d 不在Hash表中.\n", key);

return UNSUCCESS;

}

}

printf("查找成功，%d 在Hash表第 %d 个位置.\n", key, addr+1);

return SUCCESS;

}

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

{

int i,j= 0;

HashTable h;

//初始化Hash表

InitHashTable(&h);

//未插入数据之前，打印Hash表

printf("\n未插入数据之前，Hash表中内容为:\n");

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

{

printf("%d ", h.elem[i]);

}

//插入数据

printf("\n现在插入数据，请输入(A代表结束哦).\n");

while (scanf("%d", &i) == 1)

{

if (i == 'a')

{

break;

}

InsertHash(&h, i);

}

//打印插入数据后Hash表的内容

printf("插入数据后Hash表的内容为:\n");

printf("散列地址：\t");

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

printf("%d\t", i);

}

printf("\n散列表：\t");

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

{

printf("%d\t", h.elem[i]);

}

printf("\n现在进行查询.\n");

SearchHash(h, 22);

SearchHash(h, 3);

system("pause");

return 0;

}

### 实验运行结果

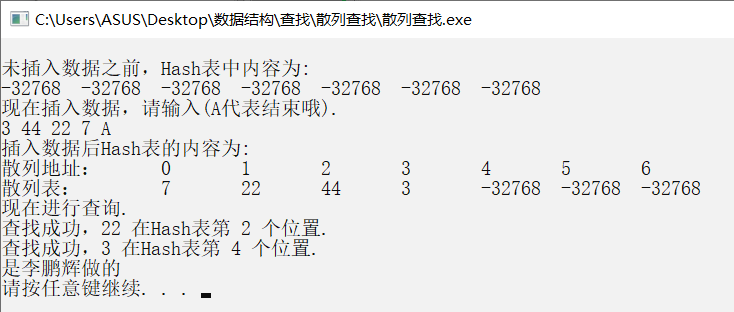


图7-4 散列查找的实现

### 遇到的问题和解决的办法

（1）问题：如何判断散列函数的定义域？

解决方法：散列函数的定义域必须包括需要存储的全部数据元素的关键字，而如果散列表允许有m个地址时，其值域必须在0~m-1。

（2）问题：散列函数计算出来的地址应如何分布？

解决方法：散列函数计算出来的地址应能均匀分布在整个地址空间中，若kcy是从关键字集合中随机抽取的一个关键字，散列函数应能以同等概串取0到m-1中的每一个值。

（3）问题：出现散列冲突时应如何解决？

解决方法：使用开放寻址法或链表法。

# 第8章 排序

输入数据可有可无，如果没有输入数据，需要在main()函数中设置好。

也可以采用随机数生成待排序序列。

## 8.1 直接插入排序的实现

### 实验代码

#include <iostream>

using namespace std;

//创建数组

void CreateArry(int arry[], int len){

    cout << "输入六位数组的元素:" << endl<<endl;

    int key;

    for (int i = 0; i < len; i++){

        cin >> key;

        arry[i] = key;

    }

}

//直接插入排序

void Inert\_Sort(int arry[], int len){

    int i, j;

    for (i = 1; i < len; ++i){

        int temp = arry[i];

        for (j = i - 1; j >= 0; --j){

            if (temp < arry[j]){

                arry[j + 1] = arry[j];

            }

            else { break; }//temp大于有序表中的最后一位则不需要移动

        }

        arry[j+1] = temp;//跳出内层循环后插入在合适的位置

    }

}

//打印数组

void Print(int arry[], int len){

 cout<<endl;

 cout<<"排序后数组：";

    for (int i = 0; i < len; i++){

        cout << arry[i] << " ";

    }

    cout << endl<<endl;

}

void test01(){

    int arry[6];

    int len = sizeof(arry) / sizeof(arry[0]);

    CreateArry(arry, len);

    Inert\_Sort(arry, len);

    Print(arry, len);

}

int main(){

  test01();

cout << "是李树臻做的" << endl;

system("pause");

  return 0;

}

### 实验运行结果

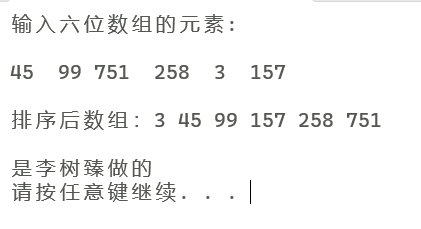


图8-1 直接插入排序

### 遇到的问题和解决的办法

（1）初始状态下，将序列的第一个记录看做一个有序的子序列。从第二个记录开逐个将待排序记录插入到已排好序的序列中，从而得到一个新的长度+1的有序序列，直到整个序列有序为止。直接插入排序算法思想简单，容易实现，适用于当待排序序列中的记录个数较小或者待排序序列已基本有序的情况。当待排序序列中的记录个数较多，并且序列不是基本有序时，算法的效率极低。

## 8.2 Shell排序的实现

### 实验代码

#include <iostream>

using namespace std;

void shellsort(int a[],int left,int right){

    int d = (left+right)>>1;//d步长

    while(d>0){

        for(int i = 0; i<right-d+1;i++)

        if(a[i]>a[i+d]){

            int temp = a[i+d];

            a[i+d] = a[i];

            a[i] = temp;

        }

        d=d>>1;

    }

}

int main(){

 int a[10];

    cout<<"请输入10个数据：";

    for(int i = 0;i<10;i++)

    cin>>a[i];

    shellsort( a,0,9);

    cout<<"排列后的数据：";

    for(int i = 0;i<10;i++)

    cout<<a[i]<<" ";

    getchar();

    return 0;

}

### 实验运行结果

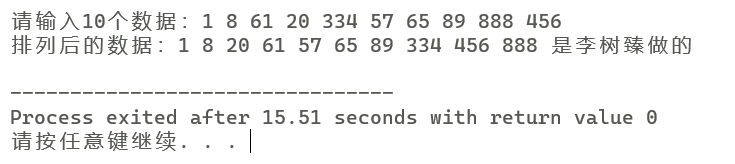


图8-2 Shell排序

### 遇到的问题和解决的办法

（1）先取一个小于n的整数d1作为第一个增量，把文件的全部记录分成d1个组。所有距离为d1的倍数的记录放在同一个组中。先在各组内进行直接插入排序；然后，取第二个增量d2<d1重复上述的分组和排序，直至所取的增量dt=1(dt<dt-l<；…<d2<d1），即所有记录放在同一组中进行直接插入排序为止。

## 8.3 冒泡排序的实现

### 实验代码

#include<iostream>

#include<cmath>

using namespace std;

int main() {

 double a[100]; //定义数组，大小100

 int N;

 int i = 0, j = 0;

 cout<<"输入元素个数：";

 cin >>N;

 cout<<endl;

 cout<<"输入数据：";

 for (i = 0; i<N; i++) //输入N个元素

  cin >> a[i];

 //-------排序---------------

 for (i = 0; i<N - 1; i++) {

  for (j = 0; j<N - 1 - i; j++){

   if (a[j]>a[j + 1]) {

    int tmp;

    tmp = a[j];

    a[j] = a[j + 1];

    a[j + 1] = tmp;

   }

  }

 }

 //--------输出----------

 cout<<endl;

 cout<<"输出数据：";

 for (i = 0; i<N; i++) {

  cout << a[i] << " ";

 }

 cout << endl;

 return 0;

}

### 实验运行结果

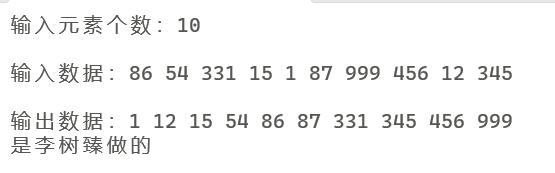


图8-3 冒泡排序

### 遇到的问题和解决的办法

问题：冒泡排序是怎样排序的？

解决办法：比较相邻的元素。如果第一个比第二个大，就交换他们两个。 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

## 8.4 快速排序的实现

### 实验代码

#include<iostream>

using namespace std;

void quickSort(int a[], int m,int n);

int partion(int a[], int m, int n);

void quickSort(int a[], int m, int n){

 if (m < n){

  int q = partion(a, m, n);

  quickSort(a, m, q );

  quickSort(a, q + 1, n);

 }

}

int partion(int a[], int m, int n){

 int key=m;

 int j= n,i=m;

 int temp1, temp2;

 while (i != j){

  while (a[j] > a[key] && i < j){

   --j;

  }

  while ((a[i] < a[key]) && (i < j)){

   ++i;

  }if (i < j){

   temp1 = a[j];

   a[j] = a[i];

   a[i] = temp1;

  }

 }

 temp2 = a[key];

 a[key] = a[i];

 a[i] = temp2;

 return i;

}

int main(){

 int a[] = { 4,3,7,9,1,2,0,5,8,6 };

 cout << "输入数据：4 3 7 9 1 2 0 5 8 6 " << endl;

 int m = 0;

 int n = (sizeof(a) / 4)-1;

 quickSort(a, m,n);

 cout<<"输出数据：";

 for (int i = 0; i < 10; i++){

cout << a[i] << " ";

}

cout << endl;

cout << "是李树臻做的" << endl;

}

### 实验运行结果

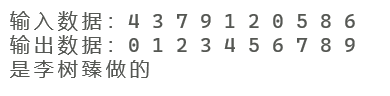


图 8-4 快速排序

### 遇到的问题和解决的办法

（1）快速排序之所以比较快，是因为与冒泡排序相比，每次的交换时跳跃式的，每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，交换的距离就大的多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。

## 8.5 简单选择排序的实现

### 实验代码：

#include <iostream>

using namespace std;

int main(){

int i,N;

cout<<"请输入数据个数："<<endl;

cin>>N;

int a[N];

cout<<"输入数据：";

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

cin>>a[i];

int k,t;

for(int i=0;i<N-1;i++){

k=i;

for(int j=i+1;j<N;j++)

if(a[j]<a[k])

k=j;

if(k != i){

t = a[k];

a[k] = a[i];

a[i] = t;

}

}

cout<<"输出数据：" ;

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

cout<<a[i]<<" ";

cout << endl;

cout << "是李树臻做的" << endl;

}

### 实验运行结果

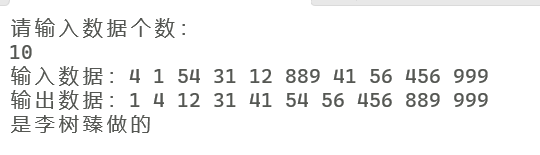


图8-5 简单选择排序

### 遇到的问题和解决的办法

问题：当出现最坏情况时的排序方法？

解决方法：最坏情况下，即待排序记录初始状态是按第一条记录最小，之后的记录从小到大顺序排列，则需要移动记录的次数最多为3（n-1）。简单选择排序过程中需要进行的比较次数与初始状态下待排序的记录序列的排列情况无关。当i=1时，需进行n-1次比较；当i=2时，需进行n-2次比较；依次类推，共需要进行的比较次数是(n-1)+(n-2)+…+2+1=n(n-1)/2，即进行比较操作的时间复杂度为O(n^2)，进行移动操作的时间复杂度为O(n)。

## 8.6 堆排序的实现

### 实验代码

#include <iostream>

using namespace std;

void adjust\_heap(int\* a, int node, int size){

int left = 2\*node + 1;

int right = 2\*node + 2;

int max = node;

if( left < size && a[left] > a[max])

max = left;

if( right < size && a[right] > a[max])

max = right;

if(max != node){

swap( a[max], a[node]);

adjust\_heap(a, max, size);

}

}

void heap\_sort(int\* a, int len){

for(int i = len/2 -1; i >= 0; --i)

adjust\_heap(a, i, len);

for(int i = len - 1; i >= 0; i--){

swap(a[0], a[i]); // 将当前最大的放置到数组末尾

adjust\_heap(a, 0 , i); // 将未完成排序的部分继续进行堆排序

}

}

int main(){

int i,N;

cout<<"请输入数据个数：";

cin>>N;

int a[N];

cout<<"输入数据：";

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

cin>>a[i];

int len= sizeof(a) / sizeof(int);

heap\_sort(a, len);

cout<<"输出数据：";

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

cout << a[i] << ' ';

cout << endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验运行结果

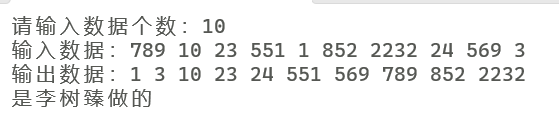


图8-6 堆排序的实现

### 遇到的问题和解决的办法

（1）对于一个完全二叉树，在填满的情况下（非叶子节点都有两个子节点），每一层的元素个数是上一层的二倍，根节点数量是1，所以最后一层的节点数量，一定是之前所有层节点总数+1，所以，我们能找到最后一层的第一个节点的索引，即节点总数/2（根节点索引为0），这也就是第一个叶子节点，所以第一个非叶子节点的索引就是第一个叶子结点的索引-1。那么对于填不满的二叉树呢？这个计算方式仍然适用，当我们从上往下，从左往右填充二叉树的过程中，第一个叶子节点，一定是序列长度/2，所以第一个非叶子节点的索引就是arr.length / 2 -1。

## 8.7 非递归实现二路归并排序

### 实验代码：

#include <iostream>

#include <stdlib.h>

using namespace std;

void Merge( int A[], int TmpA[], int L, int R, int RightEnd ){ /\* 将有序的A[L]~A[R-1]和A[R]~A[RightEnd]归并成一个有序序列 \*/

int LeftEnd, NumElements, Tmp;

int i;

LeftEnd = R - 1; /\* 左边终点位置 \*/

Tmp = L; /\* 有序序列的起始位置 \*/

NumElements = RightEnd - L + 1;

while( L <= LeftEnd && R <= RightEnd ) {

if ( A[L] <= A[R] )

TmpA[Tmp++] = A[L++]; /\* 将左边元素复制到TmpA \*/

else

TmpA[Tmp++] = A[R++]; /\* 将右边元素复制到TmpA \*/

}

while( L <= LeftEnd )

TmpA[Tmp++] = A[L++]; /\* 直接复制左边剩下的 \*/

while( R <= RightEnd )

TmpA[Tmp++] = A[R++]; /\* 直接复制右边剩下的 \*/

for( i = 0; i < NumElements; i++, RightEnd -- )

A[RightEnd] = TmpA[RightEnd]; /\*将有序的TmpA[]复制回A[] \*/

}

/\* length = 当前有序子列的长度\*/

void Merge\_pass( int A[], int TmpA[], int N, int length ){ /\* 两两归并相邻有序子列 \*/

int i, j;

for ( i=0; i <= N-2\*length; i += 2\*length )

Merge( A, TmpA, i, i+length, i+2\*length-1 );

if ( i+length < N ) /\* 归并最后2个子列\*/

Merge( A, TmpA, i, i+length, N-1);

}

void Merge\_Sort( int A[], int N ){

int length;

int \*TmpA;

length = 1; /\* 初始化子序列长度\*/

TmpA = (int\*)malloc( N \* sizeof( int ) );

if ( TmpA != NULL ) {

while( length < N ) {

Merge\_pass( A, TmpA, N, length );

length \*= 2;

//Merge\_pass( TmpA, A, N, length );

//length \*= 2;

}

free( TmpA );

}

else printf( "空间不足" );

}

int main(){

int i,N;

cout<<"请输入数据个数：";

cin>>N;

int a[N];

cout<<"输入数据：";

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

cin>>a[i];

Merge\_Sort(a,N);

cout<<"输出数据：";

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

cout<<a[i]<<" ";

cout << endl;

cout << "是李树臻做的" << endl;

return 0;

}

### 实验运行结果

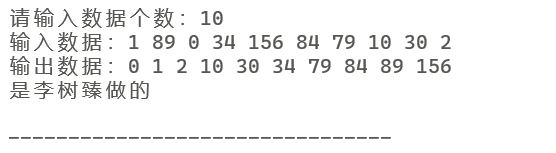


图8-7 非递归实现二路归并排序

### 遇到的问题和解决的办法

（1）问题：二路归并的思想是什么？

解决方法：将若干个有序序列两两归并，直到形成一个有序序列为止。将一个长度为n的无序序列看作是n个长度为1的有序序列的集合。然后两两归并，直到整个序列有序。

（2）在归并的过程中，可能会破坏原序列的有序性，所以需要一个新的数组在存储归并后的结果。一次归并的算法是从开始同时遍历两个序列，将较小的值放入结果序列中，直到遍历结束，成为一个有序序列。