中国地质大学（北京）

课程设计报告

**课程名称**：数据结构课程设计

**学 号**：1004191211

**姓 名**：郎文鹏

**完成时间**：2021年6月15日

# 内容摘要

本实验共涵盖了线性表实验、栈&队列实验、串&数组实验、树实验、图实验、查表表实验、内排序实验共七章数据结构实验。每个实验均以一个例题引入，先介绍实验引用的主要知识点，再搭配本人参考实验代码，最后是对实验的说明及反思。

# 实验环境

|  |  |
| --- | --- |
| 编程语言 | C++11 |
| IDE配置 | Qt Creator 4.3.1 |
| 制图软件 | processon绘图 |
| 编译环境 | Windows10x86.MinGW 32bit |

# 目 录

目录

[内容摘要 1](#_Toc74543018)

[实验环境 1](#_Toc74543019)

[目 录 2](#_Toc74543020)

[实验一 线性表实验 4](#_Toc74543021)

[题目一 顺序表基本操作 4](#_Toc74543022)

[题目二 单链表基本操作 15](#_Toc74543023)

[题目三 双链表基本操作 27](#_Toc74543024)

[题目四 循环单链表基本操作 41](#_Toc74543025)

[实训一 单链表的分割 51](#_Toc74543026)

[实训二 单链表基础上模拟集合操作 53](#_Toc74543027)

[实训三 单链表基础上模拟多项式相加运算【待填】 59](#_Toc74543028)

[实训四 单链表基础上模拟大数运算【待填】 59](#_Toc74543029)

[实训五 循环单链表解决约瑟夫环问题 60](#_Toc74543030)

[实训六 列车时刻表管理系统 66](#_Toc74543031)

[实验二 栈&队列实验 74](#_Toc74543032)

[题目一 顺序栈基本操作 74](#_Toc74543033)

[题目二 循环顺序队列基本操作 82](#_Toc74543034)

[实训一（1） 借助栈实现单链表翻转 91](#_Toc74543035)

[实训一（2）递归求值 93](#_Toc74543036)

[实训一（3）递归求全排列 95](#_Toc74543037)

[实训二 利用栈表达式求值 97](#_Toc74543038)

[实训三 利用栈求解迷宫路径 100](#_Toc74543039)

[实训四 八皇后问题 104](#_Toc74543040)

[实训五 病人看病模拟程序 108](#_Toc74543041)

[实验三 串&数组实验 111](#_Toc74543042)

[题目一 串的简单匹配算法 111](#_Toc74543043)

[题目二 串的算法 114](#_Toc74543044)

[题目三 矩阵的三元组 118](#_Toc74543045)

[题目四 广义表基本运算 123](#_Toc74543046)

[题目五 求矩阵马鞍点 128](#_Toc74543047)

[实验四 树实验 131](#_Toc74543048)

[题目一 二叉树基本运算 131](#_Toc74543049)

[题目二 前中序构造二叉树 150](#_Toc74543050)

[题目三 二叉树查找结点祖先【待填】 160](#_Toc74543051)

[题目四 线索化二叉树基本操作 160](#_Toc74543052)

[题目五 哈弗曼树编码 173](#_Toc74543053)

[实验五 图实验 179](#_Toc74543054)

[题目一 图基本运算 179](#_Toc74543055)

[题目二 图的遍历算法 184](#_Toc74543056)

[实验六 查找表实验 190](#_Toc74543057)

[题目一 顺序查找 190](#_Toc74543058)

[题目二 二分查找 191](#_Toc74543059)

[题目三 哈希表基本操作 194](#_Toc74543060)

[实验七 内排序实验 199](#_Toc74543061)

[题目一 直接插入排序 199](#_Toc74543062)

[题目二 希尔排序 201](#_Toc74543063)

[题目三 快速排序 205](#_Toc74543064)

[参考文献 209](#_Toc74543065)

# 实验一 线性表实验

## 题目一 顺序表基本操作

### 实验内容

实现顺序表的各种基本运算，并在此基础上设计一个主程序完成如下功能：

1. 初始化顺序表L。
2. 依次采用尾插法插入a,b,c,d,e元素。
3. 输出顺序表L及L的长度。
4. 判断顺序表L是否为空。
5. 输出顺序表L的第3个元素。
6. 输出元素d的位置。
7. 在第4个元素位置上插入f元素。
8. 删除L的第3个元素。
9. 输出顺序表L。
10. 释放顺序表L。

### 实验运用主要知识点



图1 顺序表存储结构

**语法基础**：、指针与引用、类和对象、模板与泛型编程

**概念基础**：线性表因为顺序存储又称为顺序表，它是用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的元素在物理位置上也相邻。数组一般通过动态分配空间，一旦数据空间占满就另外开辟一块更大的存储空间用以替换原来的存储空间的从而达到扩充存储数组的目的而不需要为线性表一次性地划分所有空间

**特点分析**：

1. 最主要的特点是随机访问，即通过首地址和元素需要可以在时间内
2. 顺序表的存储密度高，每个结点只存储数据元素
3. 顺序表逻辑上相邻的元素物理也相邻，所以插入和删除元素需要移动大量元素

**考察内容**：初始化顺序表、尾插法插入元素、顺序表属性查询、查询指定位置元素、指定位置插入元素、删除指定位置元素、释放顺序表

### 实验说明及源码

#### 操作说明

为了便于后续实验能够复用顺序表我参考的设计思路将顺序表每个操作的函数只返还一个需要的数据结果，具体不同场景需要输出的结果则由使用者根据函数返还结果在主程序进一步进行判断处理

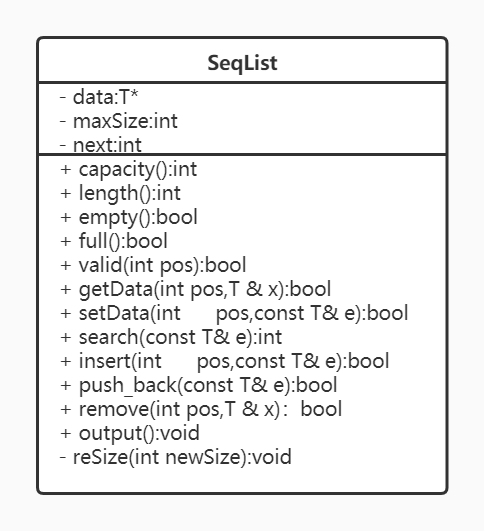


图2 顺序表类图

|  |  |
| --- | --- |
| **SeqList成员** | **作用** |
| data | 动态数组 |
| maxSize | 记录当前动态数组能防止的最大元素个数 |
| next | 记录下一个新元素需要放置在数组的下标位置 |

表2 顺序表类成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名称** | SeqList | | | |
| **类描述** | 生成顺序表对象的模板类，包含顺序表的需要用到的操作 | | | |
| capacity | capacity():int | | | |
| **功能描述** | 获取顺序表的容量 | | |
| **返回值** | 整数表示能够存储的最大元素个数 | | |
| length | length():int | | | |
| **功能描述** | 获取顺序表的长度 | | |
| **返回值** | 整数表示顺序表内当前元素的总个数 | | |
| empty | empty():bool | | | |
| **功能描述** | 判断顺序表是否为空 | | |
| **返回值** | true：顺序表为空 false：顺序表不为空 | | |
| full | full():bool | | | |
| **功能描述** | 判断顺序表为否已满 | | |
| **返回值** | true：顺序表已满 false：顺序表未满 | | |
| valid | valid(int pos):bool | | | |
| **功能描述** | 判断访问的下标是否合法 | | |
| **返回值** | true：访问下标合法 false：访问下标非法 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 访问的下标 |
| getData | getData(int pos,T &x):bool | | | |
| **功能描述** | 获取指定下标元素并返还是否成功 | | |
| **返回值** | true：获取成功 false：获取石板 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定的下标 |
| x | T& | 获取的元素 |
| setData | setData(int pos,const T &e):bool | | | |
| **功能描述** | 设置指定下标的元素值并返回是否成功 | | |
| **返回值** | true：设置成功 false：设置失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| e | T& | 欲修改的值 |
| search | search(cont T &e):int | | | |
| **功能描述** | 在顺序表中查找值为e的元素并返还对应下标 | | |
| **返回值** | -1：没有找到 非负数：找到的元素下标 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T& | 欲查找的值 |
| insert | insert(int pos,const T &e):bool | | | |
| **功能描述** | 在指定下标位置插入一个新的元素并返回是否成功 | | |
| **返回值** | true：插入成功 false：插入失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| e | T& | 新元素的值 |
| push\_back | push\_back(const T& e):bool | | | |
| **功能描述** | 在顺序表尾部插入一个元素并返回是否成功 | | |
| **返回值** | true：插入成功 false：插入失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T& | 插入元素的值 |
| remove | remove(int pos,T &x):bool | | | |
| **功能描述** | 删除顺序表中指定下标的元素同时保存在x中并返回是否成功 | | |
| **返回值** | true：删除成功 false：删除失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| x | T&x | 删除的元素值 |
| output | output():void | | | |
| **功能描述** | 按照指定格式输出顺序表中元素 | | |
| **返回值** | 无 | | |
| reSize | reSize(int newSize):void | | | |
| **功能描述** | 扩容辅助函数防止顺序表溢出 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| newSize | int | 新的扩容空间 |

表3 顺序表类接口文档

#### 程序源码

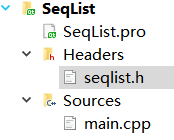


图3 顺序表源码文件树

seqlist.h文件

1. #ifndef SEQLIST\_H
2. #define SEQLIST\_H
3. #include <iostream>
4. template<class T>
5. class SeqList {
6. public:
7. SeqList(int sz = 100);                  *// 初始构造函数*
8. SeqList(const SeqList<T>& L) = default; *// 拷贝构造函数*
9. ~SeqList();                             *// 析构函数*
10. int  capacity() const;                  *// 顺序表的最大容量*
11. int  length() const;                    *// 顺序表当前长度*
12. bool empty() const;                     *// 判断顺序表是否为空*
13. bool full() const;                      *// 判断顺序表是否已满*
14. bool valid(int pos) const;              *// 判断访问下标是否合法*
15. bool getData(int pos,
16. T & x)const;               *// 获取下标对应元素并保存在x*
17. bool setData(int      pos,
18. const T& e);               *// 设置下标对应元素*
19. int  search(const T& e) const;          *// 寻找元素在表中的下标*
20. bool insert(int      pos,
21. const T& e);                *// 在指定下标插入元素*
22. bool push\_back(const T& e);             *// 尾插法插入元素*
23. bool remove(int pos,
24. T & x);                     *//删除指定下标元素并将删除元素保存在x*
25. void output() const;                    *// 遍历表中元素*
26. private:
27. T \*data;                  *// 动态数组指针*
28. int maxSize;              *// 动态数组的最大空间*
29. int next;                 *//下一个元素下标*
30. void reSize(int newSize); *// 辅助动态数组重新分配空间的函数*
31. };
32. template<class T>
33. void SeqList<T>::reSize(int newSize) {
34. if (newSize <= 0) std::cerr << "reSize falied." << std::endl;
35. else if (newSize == maxSize) return;
36. else {
37. if (newSize > this->maxSize) {
38. T \*temp = new T[newSize];
39. for (int i = 0; i < next; ++i) temp[i] = this->data[i];
40. delete[] this->data;
41. this->data = temp;
42. }
43. else next = newSize;
44. this->maxSize = newSize;
45. }
46. }
47. template<class T>
48. SeqList<T>::SeqList(int sz) {
49. this->data = new T[sz];
50. this->maxSize = sz; this->next = 0;
51. std::cout << "SeqList initialized." << std::endl;
52. }
53. template<class T>
54. SeqList<T>::~SeqList() {
55. delete[]data;
56. std::cout << "SeqList destructed." << std::endl;
57. }
58. template<class T>
59. int SeqList<T>::capacity() const {
60. return this->maxSize;
61. }
62. template<class T>
63. int SeqList<T>::length() const {
64. return this->next;
65. }
66. template<class T>
67. bool SeqList<T>::empty() const {
68. return this->next == 0;
69. }
70. template<class T>
71. bool SeqList<T>::full() const {
72. return this->next == this->maxSize;
73. }
74. template<class T>
75. bool SeqList<T>::valid(int pos) const {
76. return pos >= 0 && pos < this->next;
77. }
78. template<class T>
79. bool SeqList<T>::getData(int pos, T& x) const {
80. if (valid(pos)) {
81. x = this->data[pos];
82. return true;
83. }
84. return false;
85. }
86. template<class T>
87. bool SeqList<T>::setData(int pos, const T& e) {
88. if (valid(pos)) {
89. this->data[pos] = e;
90. return true;
91. }
92. return false;
93. }
94. template<class T>
95. int SeqList<T>::search(const T& e) const {
96. for (int i = 0; i < this->next; ++i)
97. if (this->data[i] == e) return i;
98. return -1;
99. }
100. template<class T>
101. bool SeqList<T>::insert(int pos, const T& e) {
102. *//    if ((pos == 0) && (this->next == 0)) { // 特判头插法第一个元素*
103. *//        this->data[0] = e;*
104. *//        ++this->next;*
105. *//        return true;*
106. *//    }*
107. if ((pos < 0) || (pos > this->next)) return false;  *//不能用valid函数判断因为可以在最后一个元素的后一个位置插入*
108. if (this->next + 1 > this->maxSize) reSize(this->maxSize \* 2);
109. for (int i = this->next; i >= pos + 1; --i) this->data[i] = this->data[i - 1];
110. this->data[pos] = e;
111. ++this->next;
112. return true;
113. }
114. template<class T>
115. bool SeqList<T>::push\_back(const T& e) { *// 函数复用*
116. return insert(this->next, e);
117. }
118. template<class T>
119. bool SeqList<T>::remove(int pos, T& x) {
120. if (!valid(pos)) return false;
121. x = this->data[pos];
122. for (int i = pos; i < this->next - 1; ++i) this->data[i] = this->data[i + 1];
123. --this->next;
124. return true;
125. }
126. template<class T>
127. void SeqList<T>::output() const {
128. for (int i = 0; i < this->next; ++i) {
129. std::cout << this->data[i];
130. if (i != this->next - 1) std::cout << "->";
131. }
132. std::cout << std::endl;
133. }
134. #endif *// SEQLIST\_H*

main.cpp文件

1. #include <iostream>
2. #include "seqlist.h"
3. using namespace std;
4. signed main()
5. {
6. char ch;
7. cout << "opt-1: Initialize sequence list L" << endl;
8. SeqList<char> L(1);
9. cout << endl;
10. cout
11. <<"opt-2: Inserts elements a,b,c,d, and e in sequence by using tail interpolation"
12. << endl;
13. for (char i = 'a'; i <= 'e'; ++i)
14. if (L.push\_back(i)) cout << "insert " << i << " succeed." << endl;
15. else cerr << "insert " << i << " failed." << endl;
16. cout << endl;
17. cout << "opt-3: Traverse sequence list L and the length of L" << endl;
18. L.output();
19. cout << L.length() << endl;
20. cout << "opt-4: Check whether the sequence list L is empty" << endl;
21. if (L.empty()) cout << "Yes" << endl;
22. else cout << "No" << endl;
23. cout << endl;
24. cout << "opt-5: Output the third element in the sequence list L" << endl;
25. if (L.getData(2, ch)) cout << ch << endl;
26. else cerr << "getData failed." << endl;
27. cout << endl;
28. cout << "opt-6: Output the position of element d" << endl;
29. cout << L.search('d') << endl;
30. cout << endl;
31. cout << "opt-7: Insert the f element at the fourth element position" << endl;
32. if (L.insert(3, 'f')) cout << "insert f succeed." << endl;
33. else cout << "insert f failed." << endl;
34. cout << endl;
35. cout << "opt-8: Delete the third element of L" << endl;
36. if (L.remove(2, ch)) cout << "remove " << ch << " succeed." << endl;
37. else cerr << "remove " << ch << " failed." << endl;
38. cout << endl;
39. cout << "opt-9: Output sequence list L" << endl;
40. L.output();
41. cout << endl;
42. cout << "opt-10: Release sequence list L" << endl;
43. return 0;
44. }

### 实验运行截图及说明体会

#### 运行截图

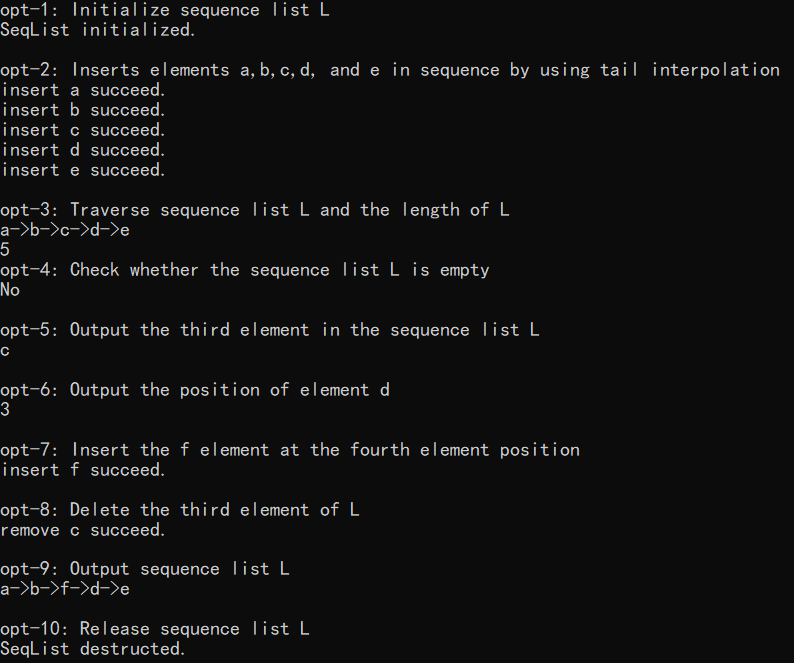


图4 顺序表实验运行结果

#### 程序说明

**扩容操作**：每次插入新元素的时候会先判断是否需要进行扩容操作，每次扩容操作会将当前的容量扩大一倍作为新的容量

**插入元素操作&尾插元素操作**：顺序表从0开始存储元素，尾插操作借助插入操作实现但是需要使用者指明插入位置，由函数自行补全参数

**删除元素操作&获取元素操作**：两个操作都是先判断下标是否合法然后返回是否成功的布尔值，另外内部参数返回引用类型的元素，这样完成两个结果的反馈供后续操作进行使用

**复杂度分析**：线性表的插入操作、删除、按值查找操作平均时间复杂度为，修改或者查询指定位置元素的操作平均时间复杂度为

#### 体会反思

1. 算法的健壮性——起初没有考虑操作进行时下标合法性的判断问题，在调试过程中出现多次以外退出的情况，查看了书籍后添加了作为一个单独的操作供内外部使用
2. 左值引用传值的问题——想要通过传入参数的方式获取函数返回的结果需要保证传入的变量是一个左值而且采用引用传递，调试时发现自己没有引用传参
3. 封装的思想——扩容操作作为数据结构自身的空间调整机理不应该对外公开，仅作为一种辅助函数供内部使用所以函数选择放在中，数据成员同理使用者只能通过类提供好的函数进行操作，这样既对用户操作透明便于使用类而且也保障了数据的安全性

## 题目二 单链表基本操作

### 实验内容

编写一个程序，实现单链表的各种基本运算，并在此基础上设计一个主程序完成如下功能：

1. 初始化单链表H。
2. 依次采用尾插法插入a,b,c,d,e元素。
3. 输出单链表H及H的长度。
4. 判断单链表H是否为空。
5. 输出单链表H的第3个元素。
6. 输出元素d的位置。
7. 在第4个元素位置上插入f元素。
8. 删除H的第3个元素。
9. 输出单链表L。
10. 释放单链表L。

### 实验运用主要知识点



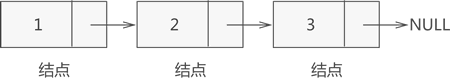


图5 单链表存储结构

**语法基础**：、指针与引用、类和对象、模板与泛型编程

**概念基础**：线性表的链式存储又称为单链表，它是指通过一组任意的存储单元来存储线性表中的数据元素。为了建立数据元素之间的线性关系，对每个链表结点处存放元素自身的信息外，还需要存放一个执行其后继的指针

**特点分析**：利用单链表可以解决顺序表需要大量连续存储单元的特点，但单链表附加指针域也存在浪费存储空间的缺点。由于单链表的元素离散的分布在存储空间中，所以单链表是非随机存取的存储结构，即不能直接找到表中某个特定的结点。查找某个结点的时候需要从表头开始遍历一次查找。通常用头结点来标识一个单链表，头指针为是表示一个空表。此外为了操作上的方便在单链表第一个结点之前附加一个结点成为头结点。头结点的数据与可以不设任何信息，也可以记录表长等信息。头结点的指针域指向线性表的第一个元素结点

**考察内容**：初始化单链表、尾插法插入元素、单链表属性查询、查询指定位置元素、指定位置插入元素、删除指定位置元素、释放单链表

### 实验说明及源码

#### 操作说明

由于测试数据量较小同时为了避免头指针在各操作中需要特判头尾的缺点我选择使用带有头结点的单链表进行实验。链表和链表结点分开封装，链表存储多个链表结点并包含一系列对于多个链表结点操作的函数

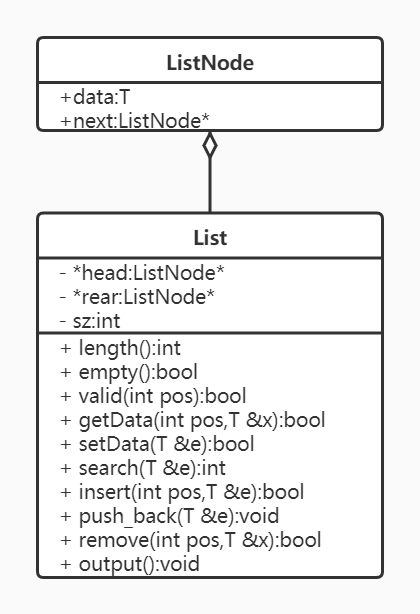


图6 单链表类图

|  |  |
| --- | --- |
| **ListNode成员** | **作用** |
| data | 存储单个元素的值 |
| next | 存储下一个结点物理地址 |

表4 单链表结点成员

|  |  |
| --- | --- |
| **List成员** | **作用** |
| head | 记录头结点物理位置 |
| rear | 记录尾结点物理地址 |
| sz | 记录链表内结点个数 |

表5 单链表成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名称** | List | | | |
| **类描述** | 生成单链表对象的模板类，包含单链表的需要用到的操作 | | | |
| length | length():int | | | |
| **功能描述** | 获取顺序表的长度 | | |
| **返回值** | 整数表示顺序表内当前元素的总个数 | | |
| empty | empty():bool | | | |
| **功能描述** | 判断顺序表是否为空 | | |
| **返回值** | true：顺序表为空 false：顺序表不为空 | | |
| valid | valid(int pos):bool | | | |
| **功能描述** | 判断访问的下标是否合法 | | |
| **返回值** | true：访问下标合法 false：访问下标非法 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 访问的下标 |
| getData | getData(int pos,T &x):bool | | | |
| **功能描述** | 获取指定下标元素并返还是否成功 | | |
| **返回值** | true：获取成功 false：获取石板 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定的下标 |
| x | T& | 获取的元素 |
| setData | setData(int pos,const T &e):bool | | | |
| **功能描述** | 设置指定下标的元素值并返回是否成功 | | |
| **返回值** | true：设置成功 false：设置失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定的下标 |
| e | T& | 欲修改的值 |
| search | search(cont T &e):int | | | |
| **功能描述** | 在顺序表中查找值为e的元素并返对应下标 | | |
| **返回值** | -1：没有找到 非负数：找到的元素下标 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T& | 欲查找的值 |
| insert | insert(int pos,const T &e):bool | | | |
| **功能描述** | 在指定下标位置插入一个新的元素并返回是否成功 | | |
| **返回值** | true：插入成功 false：插入失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| e | T& | 新元素的值 |
| push\_back | push\_back(const T& e):bool | | | |
| **功能描述** | 在顺序表尾部插入一个元素并返回是否成功 | | |
| **返回值** | true：插入成功 false：插入失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T& | 插入元素的值 |
| remove | remove(int pos,T &x):bool | | | |
| **功能描述** | 删除顺序表中指定下标的元素同时保存在x中并返回是否成功 | | |
| **返回值** | true：删除成功 false：删除失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| x | T&x | 删除的元素值 |
| output | output():void | | | |
| **功能描述** | 按照指定格式输出顺序表中元素 | | |
| **返回值** | 无 | | |

表6 单链表接口文档

#### 程序源码

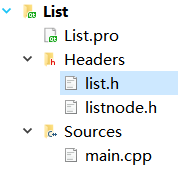


图7 单链表文件树

listnode.h文件

1. #ifndef LISTNODE\_H
2. #define LISTNODE\_H
3. template<class T>
4. class ListNode {
5. public:
6. T data;
7. ListNode \*next;
8. ListNode() = default;
9. ListNode(T data, ListNode \*next = nullptr) : data(data), next(next) {}
10. };
11. #endif *// LISTNODE\_H*

list.h文件

1. #ifndef LIST\_H
2. #define LIST\_H
3. #include "listnode.h"
4. template<class T>
5. class List {
6. public:
7. List();
8. List(const List<T>& L) = default;
9. ~List();
10. int  length() const;           *// 顺序表当前长度*
11. bool empty() const;            *// 判断顺序表是否为空*
12. bool valid(int pos) const;     *// 判断访问下标是否合法*
13. bool getData(int pos,
14. T & x)const;      *// 获取下标对应元素并保存在x*
15. bool setData(int      pos,
16. const T& e);      *// 设置下标对应元素*
17. int  search(const T& e) const; *// 寻找元素在表中的下标*
18. bool insert(int      pos,
19. const T& e);       *// 在指定下标插入元素*
20. void push\_back(const T& e);    *// 尾插法插入元素*
21. bool remove(int pos,
22. T & x);            *//删除指定下标元素并将删除元素保存在x*
23. void output() const;           *// 遍历表中元素*
24. private:
25. ListNode<T> \*head, \*rear; *// 指向头结点的头指针和尾插法用的指针*
26. int sz = 0;               *// 记录元素的数量*
27. };
28. template<class T>
29. List<T>::List() {
30. this->head = new ListNode<T>; *// 引入空结点*
31. this->head->next = nullptr;
32. this->rear = this->head;
33. this->sz = 0;
34. std::cout << "List initialized." << std::endl;
35. }
36. template<class T>
37. List<T>::~List() {
38. ListNode<T> \*temp1 = this->head, \*temp2;
39. while (temp1 != nullptr) {
40. temp2 = temp1;
41. temp1 = temp1->next;
42. delete temp2;
43. }
44. this->sz = 0;
45. std::cout << "List destructed." << std::endl;
46. }
47. template<class T>
48. int List<T>::length() const {
49. return this->sz;
50. }
51. template<class T>
52. bool List<T>::empty() const {
53. return this->sz == 0;
54. }
55. template<class T>
56. bool List<T>::valid(int pos) const {
57. return pos >= 1 && pos <= this->sz;
58. }
59. template<class T>
60. bool List<T>::getData(int pos, T& x) const {
61. if (valid(pos)) {
62. ListNode<T> \*temp = this->head;
63. for (int i = 0; i < pos; ++i) temp = temp->next;
64. x = temp->data;
65. return true;
66. }
67. return false;
68. }
69. template<class T>
70. bool List<T>::setData(int pos, const T& e) {
71. if (valid(pos)) {
72. ListNode<T> \*temp = this->head;
73. for (int i = 0; i < pos; ++i) temp = temp->next;
74. temp->data = e;
75. return true;
76. }
77. return false;
78. }
79. template<class T>
80. int List<T>::search(const T& e) const {
81. ListNode<T> \*temp = this->head->next;
82. int res = 1;
83. while (temp != nullptr) {
84. if (temp->data == e) return res;
85. temp = temp->next, ++res;
86. }
87. return -1;
88. }
89. template<class T>
90. bool List<T>::insert(int pos, const T& e) {
91. *//    if ((pos == 1) && (this->sz == 0)) {*
92. *//        ListNode<T> \*temp = new ListNode<T>;*
93. *//        temp->next = nullptr, this->head->next = temp;*
94. *//        ++this->sz;*
95. *//        return true;*
96. *//    }*
97. if ((pos < 1) || (pos > this->sz + 1)) return false;
98. ListNode<T> \*temp1 = this->head, \*temp2 = new ListNode<T>; *// 前驱结点和插入结点*
99. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
100. temp2->data = e, temp2->next = temp1->next;
101. temp1->next = temp2;
102. if (temp2->next == nullptr) this->rear = temp2;
103. ++this->sz;
104. return true;
105. }
106. template<class T>
107. void List<T>::push\_back(const T& e) {
108. ListNode<T> \*temp = new ListNode<T>{ e };
109. this->rear->next = temp;
110. this->rear = temp;
111. ++this->sz;
112. *//    return insert(this->sz + 1, e); 可以复用上面的代码进行尾结点插入但是太慢*
113. }
114. template<class T>
115. bool List<T>::remove(int pos, T& x) {
116. if (!valid(pos)) return false;
117. ListNode<T> \*temp1 = head, \*temp2; *// 前驱结点和删除结点*
118. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
119. temp2 = temp1->next;
120. x = temp2->data;
121. if (temp2->next == nullptr) this->rear = temp1;
122. else temp1->next = temp2->next;
123. delete temp2;
124. --this->sz;
125. return true;
126. }
127. template<class T>
128. void List<T>::output() const {
129. ListNode<T> \*temp = head->next;
130. while (temp != nullptr) {
131. std::cout << temp->data;
132. if (temp->next != nullptr) std::cout << "->";
133. temp = temp->next;
134. }
135. std::cout << std::endl;
136. }
137. #endif *// LIST\_H*

main.cpp文件

1. #include <iostream>
2. #include "list.h"
3. using namespace std;
4. signed main()
5. {
6. char ch;
7. cout << "opt-1: Initialize list H" << endl;
8. List<char> H;
9. cout << endl;
10. cout <<
11. "opt-2: Inserts elements a,b,c,d, and e in sequence by using tail interpolation"
12. << endl;
13. for (char i = 'a'; i <= 'e'; ++i) {
14. H.push\_back(i);
15. cout << "insert " << i << " succeed." << endl;
16. }
17. cout << endl;
18. cout << "opt-3: Traverse list H and the length of L" << endl;
19. H.output();
20. cout << H.length() << endl;
21. cout << endl;
22. cout << "opt-4: Check whether the list H is empty" << endl;
23. if (H.empty()) cout << "Yes" << endl;
24. else cout << "No" << endl;
25. cout << endl;
26. cout << "opt-5: Output the third element in the list H" << endl;
27. if (H.getData(3, ch)) cout << ch << endl;
28. else cerr << "getData failed." << endl;
29. cout << endl;
30. cout << "opt-6: Output the position of element d" << endl;
31. cout << H.search('d') << endl;
32. cout << endl;
33. cout << "opt-7: Insert the f element at the fourth element position" << endl;
34. if (H.insert(4, 'f')) cout << "insert f succeed." << endl;
35. else cout << "insert f failed." << endl;
36. cout << endl;
37. cout << "opt-8: Delete the third element of H" << endl;
38. if (H.remove(3, ch)) cout << "remove " << ch << " succeed." << endl;
39. else cerr << "remove " << ch << " failed." << endl;
40. cout << endl;
41. cout << "opt-9: Output list H" << endl;
42. H.output();
43. cout << endl;
44. cout << "opt-10: Release list H" << endl;
45. return 0;
46. }

### 实验运行截图及说明体会

#### 运行截图

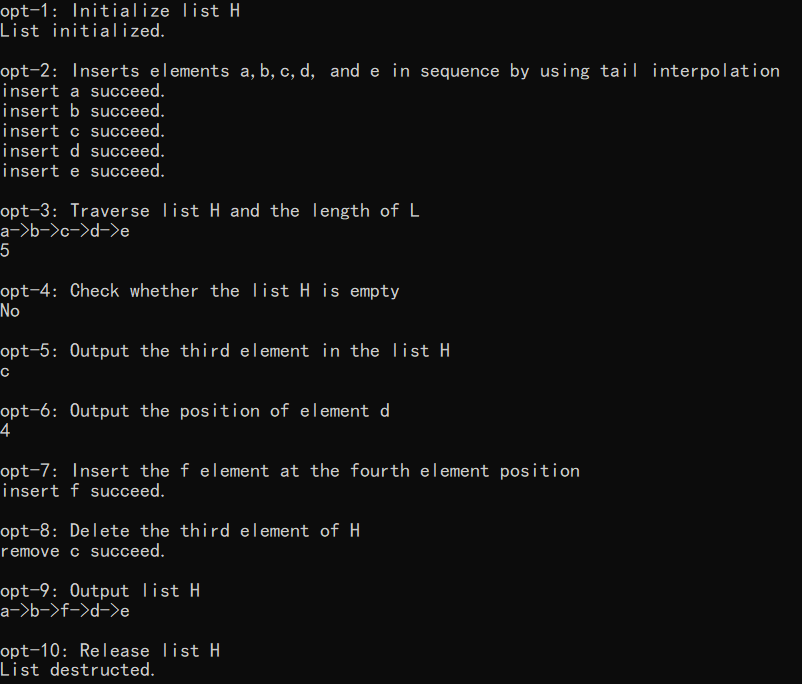


图8 单链表实验运行结果

#### 程序说明

**初始构造操作**：尾指针需要指向头结点，头指针始终指向头结点

**删除元素操作&释放操作**：由于都是从堆中申请的空间所以两个操作都需要进行手动释放空间

**插入元素操作&删除元素操作**：涉及到结点之间关系的变动的时候，由于每个结点都是记录后面结点的信息所以为了防止修改过程中关系的丢失始终遵循先处理靠后结点后处理靠前结点

**遍历操作&查找操作**：为了保证不损失表头和表尾的信息所以不能直接对指针进行操作而是进行复制一份临时指针使用

**复杂度分析**：单链表的插入操作、删除操作平均时间复杂度为，按值查找、修改或者查询指定位置元素的操作平均时间复杂度为

#### 体会反思

1. 链表结点之间的关系——调试过程中发现多出由于没有理解链表结点之间关系而导致操作过程中是表发生断开，其实最好的一种方法就是从后向前依次处理发生变化的结点并画图帮助自己理解
2. 代码优化——起初尾插元素操作使用的还是像上述顺序表的对插入元素的复用操作，后来发现单链表直接表尾插入效率更高，复用函数反而多了遍历查找的时间
3. 表成员属性的维护——调试过程中还发现自己经常忘记结点增删过程中对于表成员的修改，这是致命的错误不仅会影响上述函数自身也会导致查找遍历等操作过程中出现越界访问
4. 引入头结点的好处：一方面在链表的第一个位置上的操作和在表的其他位置上的操作一致无序进行特殊处理，另一方面无论链表是否为空其头指针都指向头结点的非空指针，因此空表和非空表的处理也得到了统一

## 题目三 双链表基本操作

### 实验内容

编写一个程序，实现双链表的各种基本运算，并在此基础上设计一个主程序完成如下功能：

1. 初始化双链表H。
2. 依次采用尾插法插入a,b,c,d,e元素。
3. 输出双链表H及H的长度。
4. 判断双链表H是否为空。
5. 输出双链表H的第3个元素。
6. 输出元素d的位置。
7. 在第4个元素位置上插入f元素。
8. 删除H的第3个元素。
9. 输出双链表L。
10. 释放双链表L。

### 实验运用主要知识点

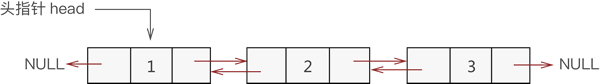


图9 双链表存储结构

**语法基础**：、指针与引用、类和对象、模板与泛型编程

**概念基础**：单链表结点中只有一个指向其后继的指针，使得单链表只能从头结点依次顺序的向后遍历。要访问到某个结点的前驱结点只能从头开始遍历，由此访问后继结点的时间复杂度为，访问前驱结点的时间复杂度为。为了克服单链表的缺点引入了双链表，其结点中有两个指针和分别指向前驱结点和后继结点。

**特点分析：**双链表在单链表的结点中增加了一个指向其前驱的指针因此双链表中的按值查找和按位查找的操作与单链表的相同。但双链表在插入和删除操作的实现上与单链表有较大的不同，这是因为“链”变化时也需要对prior指针做出修改，其关键是保证在修改的过程中不断链。此外，双链表可以很方便的找到其前驱结点，因次插入、删除操作的时间复杂度仅为

**考察内容:** 初始化双链表、尾插法插入元素、单链表属性查询、查询指定位置元素、指定位置插入元素、删除指定位置元素、释放双链表

### 实验说明及源码

#### 操作说明

仍然采用带有头结点的双链表进行实验，双链表可以看做是双链表结点以及存储处理多个双链表结点关系的两部分组成

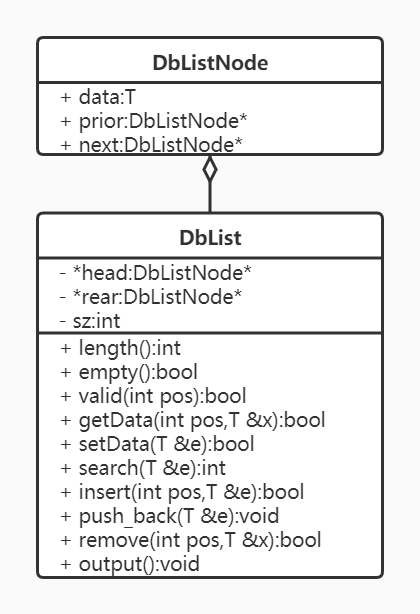


图10 双链表类图

|  |  |
| --- | --- |
| **DbListNode成员** | **作用** |
| data | 存储单个元素的值 |
| prior | 存储上一个结点物理地址 |
| next | 存储下一个结点物理地址 |

表7 双链表结点成员

|  |  |
| --- | --- |
| **DbList成员** | **作用** |
| head | 记录头结点物理位置 |
| rear | 记录尾结点物理地址 |
| sz | 记录链表内结点个数 |

表8 双链表成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名称** | DbList | | | |
| **类描述** | 生成双链表对象的模板类，包含双链表的需要用到的操作 | | | |
| length | length():int | | | |
| **功能描述** | 获取顺序表的长度 | | |
| **返回值** | 整数表示顺序表内当前元素的总个数 | | |
| empty | empty():bool | | | |
| **功能描述** | 判断顺序表是否为空 | | |
| **返回值** | true：顺序表为空 false：顺序表不为空 | | |
| valid | valid(int pos):bool | | | |
| **功能描述** | 判断访问的下标是否合法 | | |
| **返回值** | true：访问下标合法 false：访问下标非法 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 访问的下标 |
| getData | getData(int pos,T &x):bool | | | |
| **功能描述** | 获取指定下标元素并返还是否成功 | | |
| **返回值** | true：获取成功 false：获取石板 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定的下标 |
| x | T& | 获取的元素 |
| setData | setData(int pos,const T &e):bool | | | |
| **功能描述** | 设置指定下标的元素值并返回是否成功 | | |
| **返回值** | true：设置成功 false：设置失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定的下标 |
| e | T& | 欲修改的值 |
| search | search(cont T &e):int | | | |
| **功能描述** | 在顺序表中查找值为e的元素并返还对应下标 | | |
| **返回值** | -1：没有找到 非负数：找到的元素下标 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T& | 欲查找的值 |
| insert | insert(int pos,const T &e):bool | | | |
| **功能描述** | 在指定下标位置插入一个新的元素并返回是否成功 | | |
| **返回值** | true：插入成功 false：插入失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| e | T& | 新元素的值 |
| push\_back | push\_back(const T& e):bool | | | |
| **功能描述** | 在顺序表尾部插入一个元素并返回是否成功 | | |
| **返回值** | true：插入成功 false：插入失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T& | 插入元素的值 |
| remove | remove(int pos,T &x):bool | | | |
| **功能描述** | 删除顺序表中指定下标的元素同时保存在x中并返回是否成功 | | |
| **返回值** | true：删除成功 false：删除失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| pos | int | 指定下标 |
| x | T&x | 删除的元素值 |
| output | output():void | | | |
| **功能描述** | 按照指定格式输出顺序表中元素 | | |
| **返回值** | 无 | | |

表9 双链表接口文档

#### 程序源码

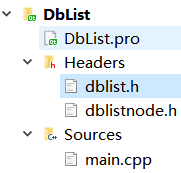


图11 双链表文件树

dblistnode.h文件

1. #ifndef DBLISTNODE\_H
2. #define DBLISTNODE\_H
3. template<class T>
4. class DbListNode
5. {
6. public:
7. T data;
8. DbListNode \*prior, \*next;
9. DbListNode() = default;
10. DbListNode(T data, DbListNode \*prior = nullptr, DbListNode \*next = nullptr) : data(data),
11. prior(prior),
12. next(next)
13. {
14. }
15. };
16. #endif *// DBLISTNODE\_H*

dblist.h文件

1. #ifndef DBLIST\_H
2. #define DBLIST\_H
3. #include "dblistnode.h"
4. template<class T>
5. class DbList
6. {
7. public:
8. DbList();
9. DbList(const DbList<T> &L) = default;
10. ~DbList();
11. int  length() const;           *// 顺序表当前长度*
12. bool empty() const;            *// 判断顺序表是否为空*
13. bool valid(int pos) const;     *// 判断访问下标是否合法*
14. bool getData(int pos, T &x)const;      *// 获取下标对应元素并保存在x*
15. bool setData(int pos, const T &e);      *// 设置下标对应元素*
16. int  search(const T &e) const; *// 寻找元素在表中的下标*
17. bool insert(int pos, const T &e);       *// 在指定下标插入元素*
18. void push\_back(const T &e);    *// 尾插法插入元素*
19. bool remove(int pos, T &x);            *// 除指定下标元素并将删除元素保存在x*
20. void output() const;           *// 遍历表中元素*
21. private:
22. DbListNode<T> \*head, \*rear; *// 指向头结点的头指针*
23. int sz = 0;                 *// 记录元素的数量*
24. };
25. template<class T>
26. DbList<T>::DbList()
27. {
28. head = new DbListNode<T>;
29. this->head->next = nullptr, this->head->prior = nullptr, this->rear = head;
30. this->sz = 0;
31. std::cout << "DbList initialized." << std::endl;
32. }
33. template<class T>
34. DbList<T>::~DbList()   *// 改用从后向前遍历释放*
35. {
36. DbListNode<T> \*temp1 = this->rear, \*temp2;
37. while (temp1 != nullptr) {
38. temp2 = temp1;
39. temp1 = temp1->prior;
40. delete temp2;
41. }
42. this->sz = 0;
43. std::cout << "DbList destructed." << std::endl;
44. }
45. template<class T>
46. int DbList<T>::length() const
47. {
48. return this->sz;
49. }
50. template<class T>
51. bool DbList<T>::empty() const
52. {
53. return this->sz == 0;
54. }
55. template<class T>
56. bool DbList<T>::valid(int pos) const
57. {
58. return pos >= 1 && pos <= this->sz;
59. }
60. template<class T>
61. bool DbList<T>::getData(int pos, T &x) const
62. {
63. if (valid(pos)) {
64. if (pos <= this->sz / 2) { *// 正向遍历找更快*
65. DbListNode<T> \*temp = this->head;
66. for (int i = 0; i < pos; ++i) temp = temp->next;
67. x = temp->data;
68. } else { *// 反向遍历找更快*
69. DbListNode<T> \*temp = this->rear;
70. for (int i = 0; i < this->sz - pos; ++i) temp = temp->prior;
71. x = temp->data;
72. }
73. return true;
74. }
75. return false;
76. }
77. template<class T>
78. bool DbList<T>::setData(int pos, const T &e)
79. {
80. if (valid(pos)) {
81. if (pos <= this->sz / 2) { *// 正向遍历找更快*
82. DbListNode<T> \*temp = this->head;
83. for (int i = 0; i < pos; ++i) temp = temp->next;
84. temp->data = e;
85. } else { *// 反向遍历找更快*
86. DbListNode<T> \*temp = this->rear;
87. for (int i = 0; i < this->sz - pos; ++i) temp = temp->prior;
88. temp->data = e;
89. }
90. return true;
91. }
92. return false;
93. }
94. template<class T>
95. int DbList<T>::search(const T &e) const   *// 正向遍历和反向遍历好坏一样*
96. {
97. DbListNode<T> \*temp = this->head->next;
98. int res = 1;
99. while (temp != nullptr) {
100. if (temp->data == e) return res;
101. temp = temp->next, ++res;
102. }
103. return -1;
104. }
105. template<class T>
106. bool DbList<T>::insert(int pos, const T &e)
107. {
108. if ((pos < 1) || (pos > this->sz + 1)) return false;
109. DbListNode<T> \*temp1, \*temp2;
110. if (pos <= this->sz + 1 / 2) {                     *// 正向遍历找更快*
111. temp1 = this->head, temp2 = new DbListNode<T>; *// 前驱结点和插入结点*
112. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
113. temp1->next->prior = temp2;
114. temp2->data = e, temp2->prior = temp1, temp2->next = temp1->next;
115. temp1->next = temp2;
116. } else {                                           *// 反向遍历找更快*
117. temp1 = this->rear, temp2 = new DbListNode<T>; *// 前驱结点和插入结点*
118. for (int i = 0; i < this->sz - pos + 1; ++i) temp1 = temp1->prior;
119. temp1->next->prior = temp2;
120. temp2->data = e, temp2->prior = temp1, temp2->next = temp1->next;
121. temp1->next = temp2;
122. }
123. if (temp2->next == nullptr) this->rear = temp2;
124. ++this->sz;
125. return true;
126. }
127. template<class T>
128. void DbList<T>::push\_back(const T &e)
129. {
130. DbListNode<T> \*temp = new DbListNode<T> { e, this->rear };
131. this->rear->next = temp;
132. this->rear = temp;
133. ++this->sz;
134. }
135. template<class T>
136. bool DbList<T>::remove(int pos, T &x)
137. {
138. if (!valid(pos)) return false;
139. DbListNode<T> \*temp1, \*temp2;
140. if (pos <= this->sz / 2) { *// 正向遍历找更快*
141. temp1 = this->head;    *// 前驱结点和插入结点*
142. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
143. temp2 = temp1->next;
144. } else {                *// 反向遍历找更快*
145. temp1 = this->rear; *// 前驱结点和插入结点*
146. for (int i = 0; i < this->sz - pos + 1; ++i) temp1 = temp1->prior;
147. temp2 = temp1->next;
148. }
149. x = temp2->data;
150. if (temp2->next == nullptr) this->rear = temp1;
151. else temp2->next->prior = temp1, temp1->next = temp2->next;
152. --this->sz;
153. delete temp2;
154. return true;
155. }
156. template<class T>
157. void DbList<T>::output() const
158. {
159. DbListNode<T> \*temp = head->next;
160. std::cout << "Positive traversal: ";
161. while (temp != nullptr) {
162. std::cout << temp->data;
163. if (temp->next != nullptr) std::cout << "->";
164. temp = temp->next;
165. }
166. std::cout << std::endl;
167. std::cout << "Reverse traversal: ";
168. temp = this->rear;
169. while (temp != head) {
170. std::cout << temp->data;
171. if (temp->prior != head) std::cout << "->";
172. temp = temp->prior;
173. }
174. std::cout << std::endl;
175. }
176. #endif *// DBLIST\_H*

main.cpp文件

1. #include <iostream>
2. #include "dblist.h"
3. using namespace std;
4. signed main()
5. {
6. char ch;
7. cout << "opt-1: Initialize double list H" << endl;
8. DbList<char> H;
9. cout << endl;
10. cout<<"opt-2: Inserts elements a,b,c,d, and e in sequence by using tail interpolation"
11. << endl;
12. for (char i = 'a'; i <= 'e'; ++i) {
13. H.push\_back(i);
14. cout << "insert " << i << " succeed." << endl;
15. }
16. cout << endl;
17. cout << "opt-3: Traverse double list H and the length of L" << endl;
18. H.output();
19. cout << H.length() << endl;
20. cout << endl;
21. cout << "opt-4: Check whether the double list H is empty" << endl;
22. if (H.empty()) cout << "Yes" << endl;
23. else cout << "No" << endl;
24. cout << endl;
25. cout << "opt-5: Output the third element in the double list H" << endl;
26. if (H.getData(3, ch)) cout << ch << endl;
27. else cerr << "getData failed." << endl;
28. cout << endl;
29. cout << "opt-6: Output the position of element d" << endl;
30. cout << H.search('d') << endl;
31. cout << endl;
32. cout << "opt-7: Insert the f element at the fourth element position" << endl;
33. if (H.insert(4, 'f')) cout << "insert f succeed." << endl;
34. else cout << "insert f failed." << endl;
35. cout << endl;
36. cout << "opt-8: Delete the third element of H" << endl;
37. if (H.remove(3, ch)) cout << "remove " << ch << " succeed." << endl;
38. else cerr << "remove " << ch << " failed." << endl;
39. cout << endl;
40. cout << "opt-9: Output double list H" << endl;
41. H.output();
42. cout << endl;
43. cout << "opt-10: Release double list H" << endl;
44. return 0;
45. }

### 实验运行截图及说明体会

#### 运行截图

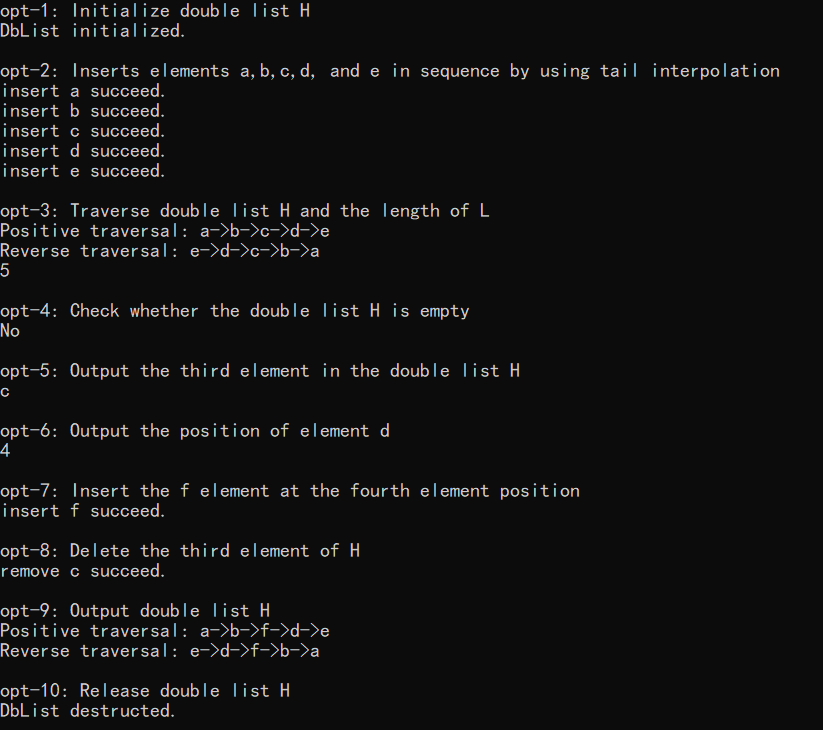


图12 双链表实验运行结果

#### 程序说明

**初始构造操作**：头指针永远指向头结点，头结点的前去后继均为，尾指针指向头结点

**插入元素操作&删除元素操作**：需要修改操作结点的前驱与后继两个结点的属性，代码书写顺序不唯一

**查找元素操作**：因为可以正序也可以倒序遍历，所以在按位查找元素的时候可以先判断一下位置在表的前部分还是后部分从而决定是采用正序遍历还是倒序遍历查找以加快查找的效率

**复杂度分析**：双链表的插入操作、删除操作平均时间复杂度为，按值查找、修改或者查询指定位置元素的操作平均时间复杂度为

#### 体会反思

1. 画图重要性：因为需要维护的信息更多了所以先后顺序也更容易混乱，借助画图可以更直观的理解
2. 多角度看待问题：插入元素操作、删除元素操作看起来好像因为需要维护更多俄关系从而更加复杂，但是在遍历的过程中因为有更多的信息所以不用再维护一个前驱结点而是找到需要操作的结点后直接借助指针就可以方便的获得它的前驱结点

## 题目四 循环单链表基本操作

### 实验内容

编写一个程序，实现循环单链表的各种基本运算，并在此基础上设计一个主程序完成如下功能：

1. 初始化循环单链表H。
2. 依次采用尾插法插入a,b,c,d,e元素。
3. 输出循环单链表H及H的长度。
4. 判断循环单链表H是否为空。
5. 输出循环单链表H的第3个元素。
6. 输出元素d的位置。
7. 在第4个元素位置上插入f元素。
8. 删除H的第3个元素。
9. 输出循环单链表L。

释放循环单链表L。

### 实验运用主要知识

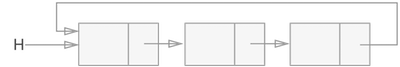


图13 循环单链表存储结构

**语法基础**：、指针与引用、类和对象、模板与泛型编程

**概念基础**：循环单链表和单链表的区别在于表中最后一个结点的指针不是而改为指向头结点，从而整个链表形成一个环

**特点分析：**在循环单链表中表尾结点的指向表头故表中没有指针域为的结点，因此循环单链表的判空条件不是头结点的指针是否为空，而是它是否等于头指针。循环单链表的插入、删除算法与单链表的几乎一样，所不同的是若操作是在表尾进行则执行的操作不同以让单链表继续保持循环的性质。正式因为循环单链表是一个“环”所以在任何一个位置上的插入和删除操作都是等价的，无序判断是否是表尾。在单链表中只能从表头结点开始往后顺序遍历整个链表而循环单链表可以从表中的任意一个结点开始遍历整个链表。有时对单链表常做的操作是在表头和表尾进行，此时对循环单链表不设头指针而仅设尾指针，从而使得操作的效率更高。

**考察内容:** 初始化循环单链表、尾插法插入元素、单链表属性查询、查询指定位置元素、指定位置插入元素、删除指定位置元素、释放循环单链表

### 实验说明及源码

#### 操作说明

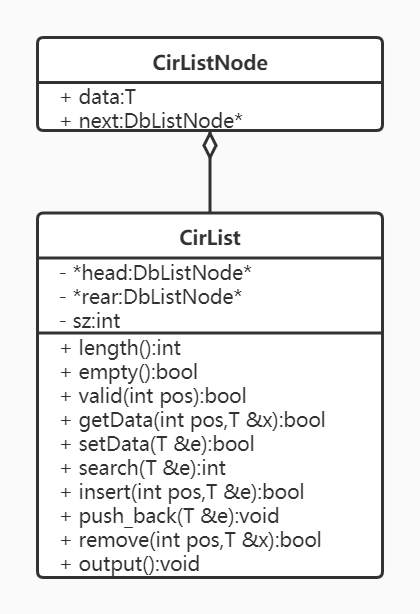


图14 循环单链表类图

由于循环单链表与单链表的差别不大所以这里不再列出成员与类的接口文档表

#### 程序源码

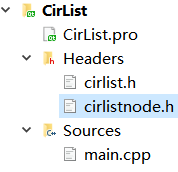


图15 循环单链表文件树

cirlistnode.h文件

1. #ifndef CIRLISTNODE\_H
2. #define CIRLISTNODE\_H
3. #include "cirlistnode.h"
4. template<class T>
5. class CirListNode
6. {
7. public:
8. T data;
9. CirListNode \*next;
10. CirListNode() = default;
11. CirListNode(T data, CirListNode \*next = nullptr) : data(data),
12. next(next)
13. {
14. }
15. };
16. #endif *// CIRLISTNODE\_H*

cirlist.h文件

1. #ifndef CIRLIST\_H
2. #define CIRLIST\_H
3. #include "cirlistnode.h"
4. template<class T>
5. class CirList
6. {
7. public:
8. CirList();
9. CirList(const CirList<T> &L) = default;
10. ~CirList();
11. int  length() const;           *// 顺序表当前长度*
12. bool empty() const;            *// 判断顺序表是否为空*
13. bool valid(int pos) const;     *// 判断访问下标是否合法*
14. bool getData(int pos, T &x)const;      *// 获取下标对应元素并保存在x*
15. bool setData(int pos, const T &e);      *// 设置下标对应元素*
16. int  search(const T &e) const; *// 寻找元素在表中的下标*
17. bool insert(int pos, const T &e);       *// 在指定下标插入元素*
18. void push\_back(const T &e);    *// 尾插法插入元素*
19. bool remove(int pos, T &x);            *// 除指定下标元素并将删除元素保存在x*
20. void output() const;           *// 遍历表中元素*
21. private:
22. CirListNode<T> \*head, \*rear;
23. int sz = 0;
24. };
25. template<class T>
26. CirList<T>::CirList()
27. {
28. this->head = new CirListNode<T>;
29. this->head->next = head;
30. this->rear = this->head;
31. this->sz = 0;
32. std::cout << "CirList initialized." << std::endl;
33. }
34. template<class T>
35. CirList<T>::~CirList()
36. {
37. CirListNode<T> \*temp1 = this->head->next, \*temp2; *// 考虑释放空表情况*
38. delete head;
39. while (temp1 != head) {
40. temp2 = temp1;
41. temp1 = temp1->next;
42. delete temp2;
43. }
44. this->sz = 0;
45. std::cout << "CirList destructed." << std::endl;
46. }
47. template<class T>
48. int CirList<T>::length() const
49. {
50. return this->sz;
51. }
52. template<class T>
53. bool CirList<T>::empty() const
54. {
55. return this->sz == 0;
56. }
57. template<class T>
58. bool CirList<T>::valid(int pos) const
59. {
60. return pos >= 1 && pos <= this->sz;
61. }
62. template<class T>
63. bool CirList<T>::getData(int pos, T &x) const
64. {
65. if (valid(pos)) {
66. CirListNode<T> \*temp = this->head;
67. for (int i = 0; i < pos; ++i) temp = temp->next;
68. x = temp->data;
69. return true;
70. }
71. return false;
72. }
73. template<class T>
74. bool CirList<T>::setData(int pos, const T &e)
75. {
76. if (valid(pos)) {
77. CirListNode<T> \*temp = this->head;
78. for (int i = 0; i < pos; ++i) temp = temp->next;
79. temp->data = e;
80. return true;
81. }
82. return false;
83. }
84. template<class T>
85. int CirList<T>::search(const T &e) const
86. {
87. CirListNode<T> \*temp = this->head->next;
88. int res = 1;
89. while (temp != head) {
90. if (temp->data == e) return res;
91. temp = temp->next, ++res;
92. }
93. return -1;
94. }
95. template<class T>
96. bool CirList<T>::insert(int pos, const T &e)
97. {
98. if ((pos < 1) || (pos > this->sz + 1)) return false;
99. CirListNode<T> \*temp1 = this->head, \*temp2 = new CirListNode<T>;
100. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
101. temp2->data = e, temp2->next = temp1->next;
102. temp1->next = temp2;
103. if (temp2->next == head) this->rear = temp2;
104. ++this->sz;
105. return false;
106. }
107. template<class T>
108. void CirList<T>::push\_back(const T &e)
109. {
110. CirListNode<T> \*temp = new CirListNode<T> { e, head };
111. this->rear->next = temp;
112. this->rear = temp;
113. ++this->sz;
114. }
115. template<class T>
116. bool CirList<T>::remove(int pos, T &x)
117. {
118. if (!valid(pos)) return false;
119. CirListNode<T> \*temp1 = head, \*temp2;
120. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
121. temp2 = temp1->next;
122. temp1->next = temp2->next;
123. x = temp2->data;
124. if (temp2->next == head) this->rear = temp1;
125. delete temp2;
126. --this->sz;
127. return true;
128. }
129. template<class T>
130. void CirList<T>::output() const
131. {
132. CirListNode<T> \*temp = head->next;
133. while (temp != head) {
134. std::cout << temp->data;
135. if (temp->next != head) std::cout << "->";
136. temp = temp->next;
137. }
138. std::cout << std::endl;
139. }
140. #endif *// CIRLIST\_H*

main.cpp文件

1. #include <iostream>
2. #include "cirlist.h"
3. using namespace std;
4. signed main()
5. {
6. char ch;
7. cout << "opt-1: Initialize circular list H" << endl;
8. CirList<char> H;
9. cout << endl;
10. cout
11. <<"opt-2: Inserts elements a,b,c,d, and e in sequence by using tail interpolation"
12. << endl;
13. for (char i = 'a'; i <= 'e'; ++i) {
14. H.push\_back(i);
15. cout << "insert " << i << " succeed." << endl;
16. }
17. cout << endl;
18. cout << "opt-3: Traverse circular list H and the length of L" << endl;
19. H.output();
20. cout << H.length() << endl;
21. cout << endl;
22. cout << "opt-4: Check whether the circular list H is empty" << endl;
23. if (H.empty()) cout << "Yes" << endl;
24. else cout << "No" << endl;
25. cout << endl;
26. cout << "opt-5: Output the third element in the circular list H" << endl;
27. if (H.getData(3, ch)) cout << ch << endl;
28. else cerr << "getData failed." << endl;
29. cout << endl;
30. cout << "opt-6: Output the position of element d" << endl;
31. cout << H.search('d') << endl;
32. cout << endl;
33. cout << "opt-7: Insert the f element at the fourth element position" << endl;
34. if (H.insert(4, 'f')) cout << "insert f succeed." << endl;
35. else cout << "insert f failed." << endl;
36. cout << endl;
37. cout << "opt-8: Delete the third element of H" << endl;
38. if (H.remove(3, ch)) cout << "remove " << ch << " succeed." << endl;
39. else cerr << "remove " << ch << " failed." << endl;
40. cout << endl;
41. cout << "opt-9: Output circular list H" << endl;
42. H.output();
43. cout << endl;
44. cout << "opt-10: Release circular list H" << endl;
45. return 0;
46. }

### 实验运行截图及说明体会

#### 运行截图

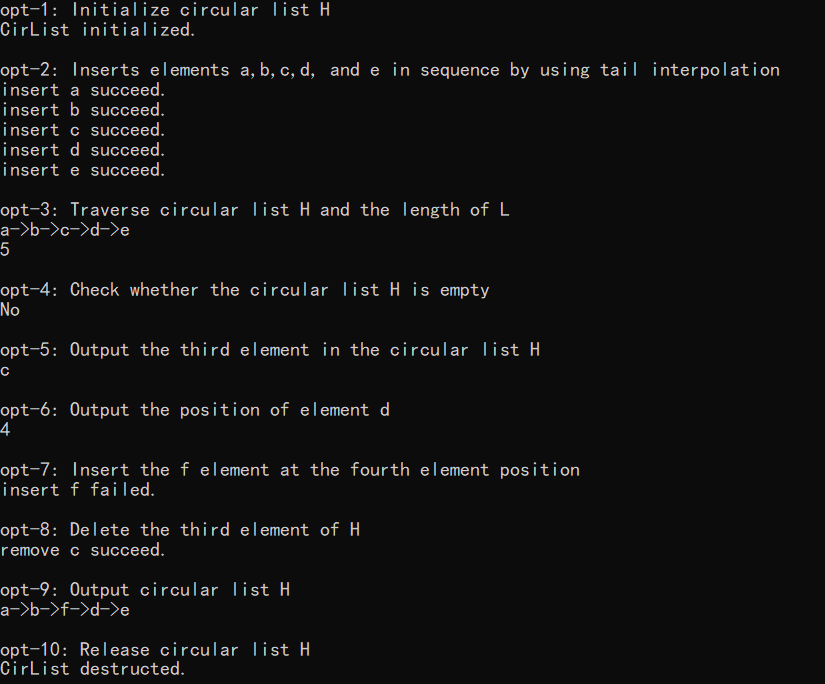


图16 循环单链表实验运行结果

#### 程序说明

**判空操作**：循环单链表的判空条件不是头结点的指针为空，而是它是否等于头指针

**复杂度分析**：循环单链表的插入操作、删除操作平均时间复杂度为，按值查找、修改或者查询指定位置元素的操作平均时间复杂度为

#### 体会反思

1. 循环双链表——根据前面所学的链表组合可以推出循环双链表，不同的是在循环双联表中头结点的指针还要指向表尾结点，在循环双链表中判空的条件是头结点的和的域都等于头结点
2. 静态链表——在一些不支持指针的高级语言中静态链表是一种折中的方案，虽然没有单链表使用起来简便但也实现了插入、删除元素操作并可以进行动态的内存分配。它借助数组来表述线性表的链式存储结构，结点也有数据域和指针域，与前面所讲的链表中的指针不同的是这里的指针是结点的相对地址（数组下标）

## 实训一 单链表的分割

### 实训题目

将单链表按某个基准划分。编写一个程序，以给定值x为基准将单链表分割为两部分，所有小于x的结点排在大于或等于x的结点之前。并分析时间复杂度

### 实训思路与源码

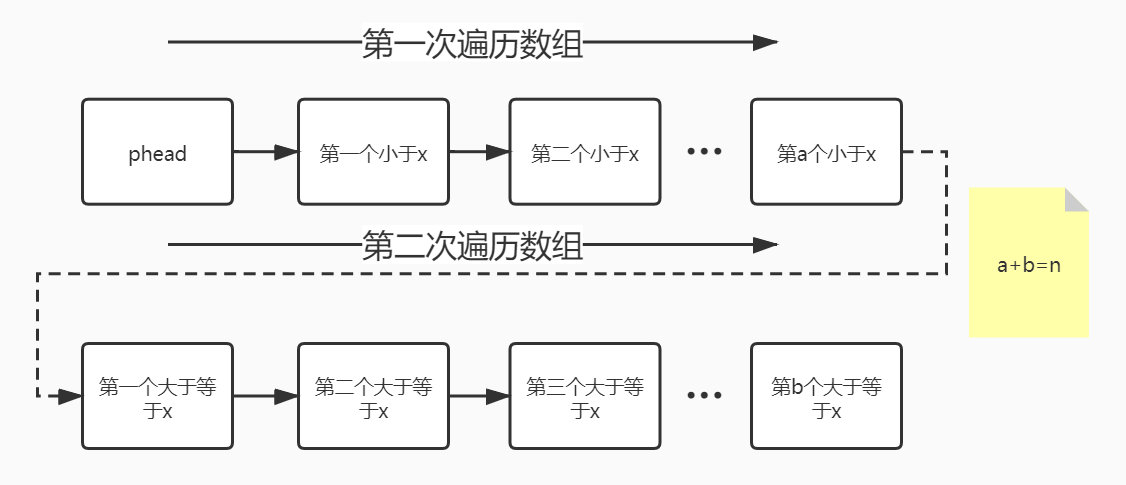


图18 实训一朴素思路

朴素思路：建立一个空表然后先遍历一遍数组将小于x的元素先用尾插法插入表中然后在遍历一遍数组将大于等于x的元素尾插法插入表中，时间复杂度为

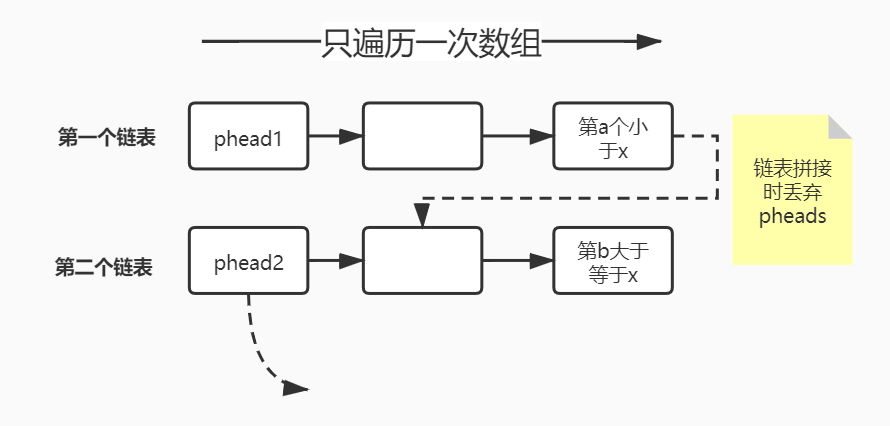


图19 实训一优化思路

优化思路：分别创建两个空表，只用遍历一遍数组然后将小于x的元素插入一个表中，将大于等于x的元素插入另一个表中，最后将两个表拼接，时间复杂度为

源码采用实验二中完成的单链表操作借助暴力操作完成

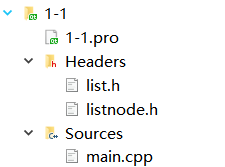


图20 实训一文件树

listnode.h&listnode.h文件略

main.cpp文件

1. #include <iostream>
2. #include "list.h"
3. using namespace std;
4. signed main()
5. {
6. List<int> L, res;
7. int a[10] = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };
8. for (auto e:a) L.push\_back(e);
9. int x = 5;
10. int temp;
11. for (int i = 1; i <= 10; ++i) {
12. L.getData(i, temp);
13. if (temp < x) res.push\_back(temp);
14. }
15. for (int i = 1; i <= 10; ++i) {
16. L.getData(i, temp);
17. if (temp >= x) res.push\_back(temp);
18. }
19. L.output();
20. cout << "x: " << x << endl;
21. res.output();
22. return 0;
23. }

### 实训运行截图

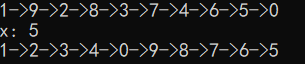


图21 实训一运行结果

## 实训二 单链表基础上模拟集合操作

### 实训题目

用单链表表示的两个集合（假设同一个集合中不存在重复的元素），求它们的并、交和差运算

### 实训思路与源码

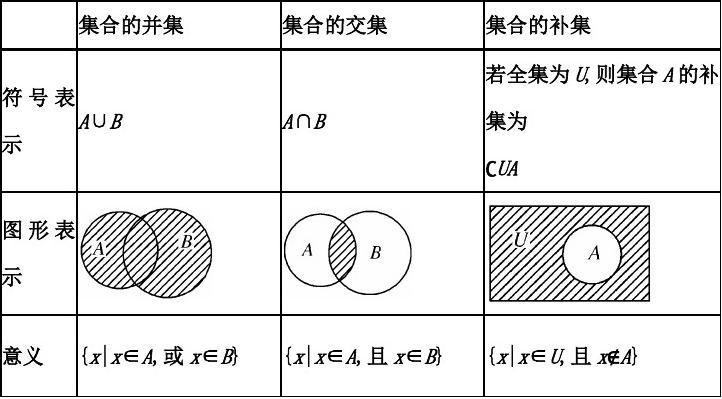


图22 实训二思路

单链表的基础上公有继承派生出集合类，定义该集合的交、并、补等相关操作以及辅助的去重操作

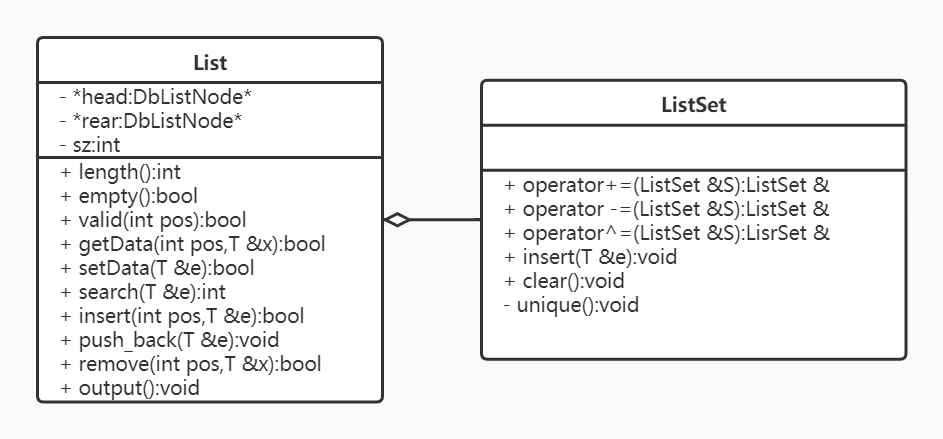


图23 集合类图

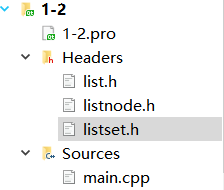


图24 实训二文件树

listnode.h&list.h文件略

listset.h文件

1. #ifndef LISTSET\_H
2. #define LISTSET\_H
3. #include "list.h"
4. #include <map>
5. template<class T>
6. class ListSet : public List<T>
7. {
8. public:
9. ListSet() = default;
10. ListSet(const ListSet<T> &S);
11. ListSet<T> &operator+=(const ListSet<T> &S);
12. ListSet<T> &operator-=(const ListSet<T> &S);
13. ListSet<T> &operator^=(const ListSet<T> &S);
14. ListSet<T> &operator=(const ListSet<T> &S);
15. void        insert(const T &e); *// 覆盖父类函数*
16. void        clear();
17. private:
18. void unique(); *// 辅助函数用来内部去重*
19. };
20. template<class T>
21. void ListSet<T>::unique()
22. {
23. std::map<T, bool> mp;
24. ListNode<T> \*temp1 = this->head, \*temp2; *// 前驱结点和判断结点*
25. while (temp1->next != nullptr) {         *// 直至最后一个元素也被去重*
26. temp2 = temp1->next;
27. if (mp[temp2->data]) {
28. temp1->next = temp2->next;
29. delete temp2;
30. if (temp2 == this->rear) this->rear = temp1;
31. --this->sz;
32. } else mp[temp2->data] = true;
33. temp1 = temp1->next;
34. }
35. }
36. template<class T>
37. ListSet<T>::ListSet(const ListSet<T> &S) : List<T >::List()
38. {
39. ListNode<T> \*temp = S.head->next;
40. while (temp != nullptr) {
41. this->push\_back(temp->data);
42. temp = temp->next;
43. }
44. }
45. template<class T>
46. void ListSet<T>::insert(const T &e)
47. {
48. if (this->search(e) == -1) this->push\_back(e);
49. }
50. template<class T>
51. ListSet<T> &ListSet<T>::operator+=(const ListSet &S)
52. {
53. ListNode<T> \*temp = S.head->next;
54. while (temp != nullptr) {
55. this->push\_back(temp->data);
56. temp = temp->next;
57. }
58. unique();
59. return \*this;
60. }
61. template<class T>
62. ListSet<T> &ListSet<T>::operator-=(const ListSet<T> &S)
63. {
64. std::map<T, bool> mp;
65. ListNode<T> \*temp = S.head->next;
66. while (temp != nullptr) {
67. mp[temp->data] = true;
68. temp = temp->next;
69. }
70. ListNode<T> \*temp1 = this->head, \*temp2; *// 前驱结点和删除结点*
71. while (temp1->next != nullptr) {
72. temp2 = temp1->next;
73. if (mp[temp2->data]) {
74. temp1->next = temp2->next;
75. if (temp2 == this->rear) this->rear = temp1;
76. delete temp2;
77. --this->sz;
78. } else temp1 = temp1->next;
79. }
80. return \*this;
81. }
82. template<class T>
83. ListSet<T> &ListSet<T>::operator^=(const ListSet<T> &S)
84. {
85. std::map<T, int> mp;
86. ListNode<T> \*temp = this->head->next;
87. while (temp != nullptr) {
88. mp[temp->data] = true;
89. temp = temp->next;
90. }
91. clear();
92. temp = S.head->next;
93. while (temp != nullptr) {
94. if (mp[temp->data]) insert(temp->data);
95. temp = temp->next;
96. }
97. return \*this;
98. }
99. template<class T>
100. ListSet<T> &ListSet<T>::operator=(const ListSet<T> &S)
101. {
102. this->clear();
103. ListNode<T> \*temp = S.head->next;
104. while (temp != nullptr) {
105. this->push\_back(temp->data);
106. temp = temp->next;
107. }
108. return \*this;
109. }
110. template<class T>
111. void ListSet<T>::clear()
112. {
113. ListNode<T> \*temp;
114. while (this->head->next != nullptr) {
115. temp = this->head->next;
116. this->head->next = temp->next;
117. delete temp;
118. }
119. this->sz = 0;
120. this->rear = this->head;
121. *// std::cout << "ListSet has been cleared." << std::endl;*
122. }
123. #endif *// LISTSET\_H*

main.cpp文件

1. #include <iostream>
2. #include "listset.h"
3. using namespace std;
4. signed main()
5. {
6. ListSet<int> s1, s2;
7. int a[5] = { 1, 2, 3, 4, 5 }, b[5] = { 3, 4, 5, 6, 7 };
8. for (auto e:a) s1.insert(e);
9. for (auto e:b) s2.insert(e);
10. cout << "s1: ";
11. s1.output();
12. cout << "s2: ";
13. s2.output();
14. ListSet<int> s3(s1);
15. s3 += s2;
16. cout << "s1+s2: ";
17. s3.output();
18. s3 = s1;
19. s3 ^= s2;
20. cout << "s1^s2: ";
21. s3.output();
22. s3 = s1;
23. s3 -= s2;
24. cout << "s1-=s2: ";
25. s3.output();
26. return 0;
27. }

### 实训运行截图

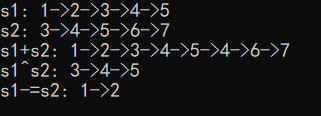


图25 实训二运行结果

## 实训三 单链表基础上模拟多项式相加运算【待填】

### 实训题目

实现两个多项式相加运算

## 实训四 单链表基础上模拟大数运算【待填】

### 实训题目

用单链表实现两个大整数相加运算。要求：

1. 将用户输入的十进制整数字符串转化为带头结点的单链表，每个结点存放一个整数位
2. 求两个整数单链表相加的结果单链表
3. 求结果单链表的中间位，如234的中间位是“3”，2345的中间位是“3”

## 实训五 循环单链表解决约瑟夫环问题

### 实训题目

利用循环单链表的操作求解下述问题：用整数序列表示顺序围坐在圆桌周围的人。然后使用以及或者作为输入数据，检查你的程序的正确性和健壮性

### 实训思路与源码

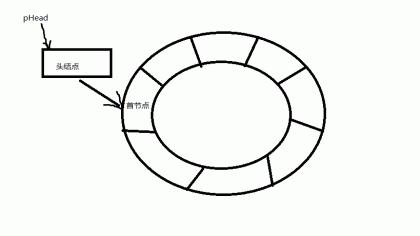


图26 实训五思路图

给实验四完成的循环单链表定义一个新的函数——边循环遍历边统计，统计到指定位置的结点删除并重新统计直至链表为空即获得了约瑟夫环的出队顺序

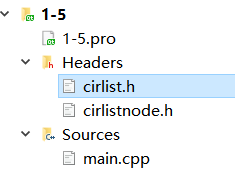


图27 实训五运行结果

cirlistnode.h文件略

cirlist.h文件

1. #ifndef CIRLIST\_H
2. #define CIRLIST\_H
3. #include "cirlistnode.h"
4. template<class T>
5. class CirList
6. {
7. public:
8. CirList();
9. CirList(const CirList<T> &L) = default;
10. ~CirList();
11. int  length() const;           *// 顺序表当前长度*
12. bool empty() const;            *// 判断顺序表是否为空*
13. bool valid(int pos) const;     *// 判断访问下标是否合法*
14. bool getData(int pos, T &x)const;      *// 获取下标对应元素并保存在x*
15. bool setData(int pos, const T &e);      *// 设置下标对应元素*
16. int  search(const T &e) const; *// 寻找元素在表中的下标*
17. bool insert(int pos, const T &e);       *// 在指定下标插入元素*
18. void push\_back(const T &e);    *// 尾插法插入元素*
19. bool remove(int pos, T &x);            *// 除指定下标元素并将删除元素保存在x*
20. void output() const;           *// 遍历表中元素*
21. void josephus(int s, int m);
22. private:
23. CirListNode<T> \*head, \*rear;
24. int sz = 0;
25. };
26. template<class T>
27. CirList<T>::CirList()
28. {
29. this->head = new CirListNode<T>;
30. this->head->next = head;
31. this->rear = this->head;
32. this->sz = 0;
33. *// std::cout << "CirList initialized." << std::endl;*
34. }
35. template<class T>
36. CirList<T>::~CirList()
37. {
38. CirListNode<T> \*temp1 = this->head->next, \*temp2; *// 考虑释放空表情况*
39. delete head;
40. while (temp1 != head) {
41. temp2 = temp1;
42. temp1 = temp1->next;
43. delete temp2;
44. }
45. this->sz = 0;
46. *// std::cout << "CirList destructed." << std::endl;*
47. }
48. template<class T>
49. int CirList<T>::length() const
50. {
51. return this->sz;
52. }
53. template<class T>
54. bool CirList<T>::empty() const
55. {
56. return this->sz == 0;
57. }
58. template<class T>
59. bool CirList<T>::valid(int pos) const
60. {
61. return pos >= 1 && pos <= this->sz;
62. }
63. template<class T>
64. bool CirList<T>::getData(int pos, T &x) const
65. {
66. if (valid(pos)) {
67. CirListNode<T> \*temp = this->head;
68. for (int i = 0; i < pos; ++i) temp = temp->next;
69. x = temp->data;
70. return true;
71. }
72. return false;
73. }
74. template<class T>
75. bool CirList<T>::setData(int pos, const T &e)
76. {
77. if (valid(pos)) {
78. CirListNode<T> \*temp = this->head;
79. for (int i = 0; i < pos; ++i) temp = temp->next;
80. temp->data = e;
81. return true;
82. }
83. return false;
84. }
85. template<class T>
86. int CirList<T>::search(const T &e) const
87. {
88. CirListNode<T> \*temp = this->head->next;
89. int res = 1;
90. while (temp != head) {
91. if (temp->data == e) return res;
92. temp = temp->next, ++res;
93. }
94. return -1;
95. }
96. template<class T>
97. bool CirList<T>::insert(int pos, const T &e)
98. {
99. if ((pos < 1) || (pos > this->sz + 1)) return false;
100. CirListNode<T> \*temp1 = this->head, \*temp2 = new CirListNode<T>;
101. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
102. temp2->data = e, temp2->next = temp1->next;
103. temp1->next = temp2;
104. if (temp2->next == head) this->rear = temp2;
105. ++this->sz;
106. return false;
107. }
108. template<class T>
109. void CirList<T>::push\_back(const T &e)
110. {
111. CirListNode<T> \*temp = new CirListNode<T> { e, head };
112. this->rear->next = temp;
113. this->rear = temp;
114. ++this->sz;
115. }
116. template<class T>
117. bool CirList<T>::remove(int pos, T &x)
118. {
119. if (!valid(pos)) return false;
120. CirListNode<T> \*temp1 = head, \*temp2;
121. for (int i = 0; i < pos - 1; ++i) temp1 = temp1->next;
122. temp2 = temp1->next;
123. temp1->next = temp2->next;
124. x = temp2->data;
125. if (temp2->next == head) this->rear = temp1;
126. delete temp2;
127. --this->sz;
128. return true;
129. }
130. template<class T>
131. void CirList<T>::output() const
132. {
133. CirListNode<T> \*temp = head->next;
134. while (temp != head) {
135. std::cout << temp->data;
136. if (temp->next != head) std::cout << "->";
137. temp = temp->next;
138. }
139. std::cout << std::endl;
140. }
141. template<class T>
142. void CirList<T>::josephus(int s, int m)
143. {
144. if ((s > this->sz) || (this->sz == 0)
145. || (m == 0)) {
146. std::cout << "failed to solve josephus." << std::endl;
147. return;
148. }
149. std::cout << "josephus answer: ";
150. CirListNode<T> \*temp1 = this->head, \*temp2; *// 前驱结点和删除结点*
151. for (int i = 0; i < s - 1; ++i) temp1 = temp1->next;
152. int cnt = 0;
153. while (this->sz) {
154. ++cnt;
155. if (cnt == m) {
156. temp2 = temp1->next;
157. cnt = 0;
158. std::cout << temp2->data << ' ';
159. temp1->next = temp2->next;
160. if (temp2 == rear) rear = temp1, temp1 = this->head;
161. delete temp2;
162. --this->sz;
163. continue;
164. }
165. temp1 = temp1->next;                    *// 前驱结点往后走*
166. if (temp1 == rear) temp1 = this->head;  *// 到了尾结点循环回来*
167. }
168. std::cout << std::endl;
169. }
170. #endif *// CIRLIST\_H*

main.cpp文件

1. #include <iostream>
2. #include "cirlist.h"
3. using namespace std;
4. void Josephus(int n, int s, int m)
5. {
6. CirList<int> L; *// 单循环链表写个函数就可以了*
7. for (int i = 1; i <= n; ++i) L.push\_back(i);
8. L.josephus(s, m);
9. }
10. signed main()
11. {
12. Josephus(9, 1, 5);
13. Josephus(9, 1, 0);
14. Josephus(9, 1, 10);
15. return 0;
16. }

### 实训运行截图



图28 实训五运行结果

## 实训六 列车时刻表管理系统

### 实训题目

案例描述：一个火车要对进出本站的列车信息进行计算机管理，包括建立、增加、删除、查询、修改车次信息等。列车信息有车次、开点、到点、始发站、终点站等。已知进出该站的列车车次变化较多

### 实训思路与源码

分析案例可以发现列车数量多而且增删操作比较多，适合使用链表式存储，我为了加快系统的运行效率选择利用实验三完成的双链表，定义一个新类作为基础的操作数据单元，一个复用双链表操作的类作为系统管理类。在主函数中创建一个系统管理类对象通过调用自身封装好的函数模拟案例的操作

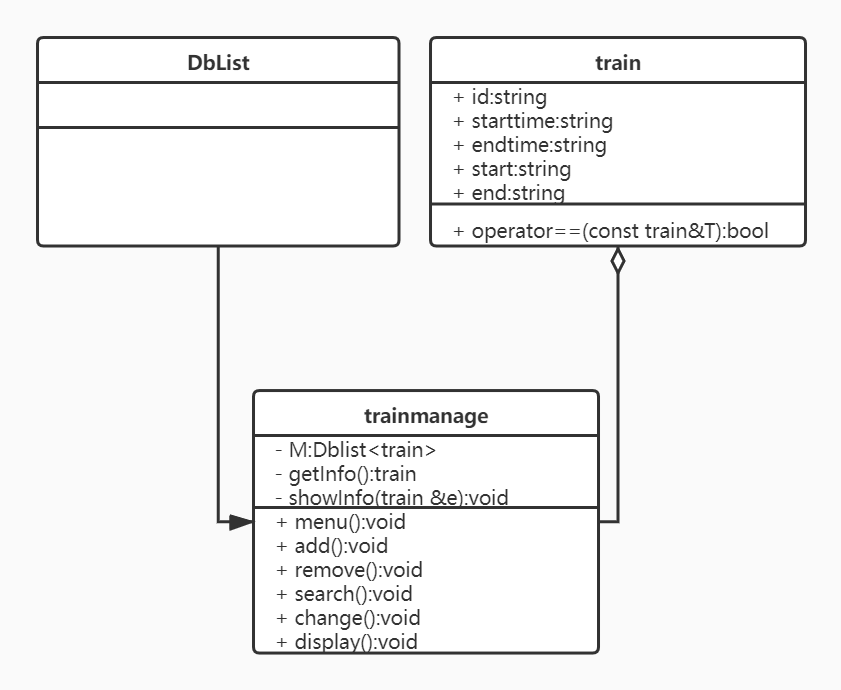


图29 列车时刻表管理系统类图

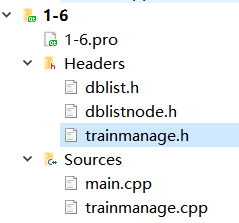


图30 实训六文件树

dblistnode.h&dblist.h文件略

trainmanage.h文件

1. #ifndef TRAINMANAGE\_H
2. #define TRAINMANAGE\_H
3. #include "dblist.h"
4. #include <iostream>
5. #include <string>
6. class train
7. {
8. public:
9. train() = default;
10. train(std::string id, std::string starttime = "00:00", std::string endtime = "00:00",
11. std::string start = "", std::string end = "") :
12. id(id),
13. starttime(starttime),
14. endtime(endtime),
15. start(start),
16. end(end)
17. {
18. }
19. bool operator==(const train &T) const
20. {
21. return this->id == T.id;
22. }
23. std::string id, starttime, endtime, start, end;
24. };
25. class trainmanage
26. {
27. public:
28. trainmanage();
29. ~trainmanage();
30. void menu();
31. void add();
32. void remove();
33. void search();
34. void change();
35. void display();
36. private:
37. DbList<train> M;
38. train getInfo(); *// 辅助函数*
39. void  showInfo(train &e);
40. };
41. #endif *// TRAINMANAGE\_H*

trainmanege.cpp文件

1. #include "trainmanage.h"
2. trainmanage::trainmanage()
3. {
4. std::cout << "Welcome to the train management system." << std::endl;
5. }
6. trainmanage::~trainmanage()
7. {
8. std::cout << "Thanks for using the train management system." << std::endl;
9. }
10. void trainmanage::search()
11. {
12. std::cout << "Input train id: ";
13. std::string s;
14. std::cin >> s;
15. train temp(s);
16. int pos = M.DbList::search(temp);
17. if (pos == -1) {
18. std::cout << "failed to search train." << std::endl;
19. return;
20. }
21. M.DbList::getData(pos, temp);
22. std::cout << "id\tstarttime\tendtime\tstart\tend" << std::endl;
23. showInfo(temp);
24. std::cout << "search train succeed." << std::endl;
25. }
26. void trainmanage::change()
27. {
28. std::cout << "Input train id: ";
29. std::string s;
30. std::cin >> s;
31. train temp(s);
32. int pos = M.DbList::search(temp);
33. if (pos == -1) {
34. std::cout << "failed to change train." << std::endl;
35. return;
36. }
37. temp = getInfo();
38. M.setData(pos, temp);
39. std::cout << "change train succeed." << std::endl;
40. }
41. void trainmanage::display()
42. {
43. std::cout << "id\tstarttime\tendtime\tstart\tend" << std::endl;
44. train temp;
45. for (int i = 1; i <= M.length(); ++i) { *// 效率有点低了其实重载输出运算符更好*
46. M.getData(i, temp);
47. showInfo(temp);
48. }
49. }
50. train trainmanage::getInfo()
51. {
52. std::string id, starttime, endtime, start, end;
53. std::cout << "Input id: ";
54. std::cin.ignore();
55. std::cin >> id;
56. std::cout << "Input starttime: ";
57. std::cin.ignore();
58. std::cin >> starttime;
59. std::cout << "Input endtime: ";
60. std::cin.ignore();
61. std::cin >> endtime;
62. std::cout << "Input start: ";
63. std::cin.ignore();
64. std::cin >> start;
65. std::cout << "Input end: ";
66. std::cin.ignore();
67. std::cin >> end;
68. return train { id, starttime, endtime, start, end };
69. }
70. void trainmanage::showInfo(train &e)
71. {
72. std::cout << e.id << "\t" << e.starttime << "\t\t" << e.endtime << "\t"
73. <<e.start << "\t" << e.end << std::endl;
74. }
75. void trainmanage::menu()
76. {
77. std::cout << "================================" << std::endl;
78. std::cout << "\*\*\*\*  train manage system   \*\*\*\*" << std::endl;
79. std::cout << "\*\*\*\*    [0] exit            \*\*\*\*" << std::endl;
80. std::cout << "\*\*\*\*    [1] add train       \*\*\*\*" << std::endl;
81. std::cout << "\*\*\*\*    [2] remove train    \*\*\*\*" << std::endl;
82. std::cout << "\*\*\*\*    [3] search train    \*\*\*\*" << std::endl;
83. std::cout << "\*\*\*\*    [4] change train    \*\*\*\*" << std::endl;
84. std::cout << "\*\*\*\*    [5] all train       \*\*\*\*" << std::endl;
85. std::cout << "================================" << std::endl;
86. std::cout << "Input your choice: ";
87. }
88. void trainmanage::add()
89. {
90. std::cout << "Input train information: " << std::endl;
91. train temp = getInfo();
92. if (M.search(temp) != -1) {
93. std::cout << "Add train failed." << std::endl;
94. return;
95. }
96. M.push\_back(temp);
97. std::cout << "Add train succeed." << std::endl;
98. }
99. void trainmanage::remove()
100. {
101. std::cout << "Input train id: ";
102. std::cin.ignore();
103. std::string s;
104. std::cin >> s;
105. train temp(s);
106. int pos = M.DbList::search(temp);
107. if (pos == -1) {
108. std::cout << "failed to remove train." << std::endl;
109. return;
110. }
111. M.DbList::remove(pos, temp);
112. std::cout << "id\tstarttime\tendtime\tstart\tend" << std::endl;
113. showInfo(temp);
114. std::cout << "remove train succeed." << std::endl;
115. }

main.cpp文件

1. #include <iostream>
2. #include "trainmanage.h"
3. using namespace std;
4. signed main()
5. {
6. trainmanage system;
7. while (1) {
8. system.menu();
9. int opt;
10. cin >> opt;
11. if (opt == 0) break;
12. else if (opt == 1) system.add();
13. else if (opt == 2) system.remove();
14. else if (opt == 3) system.search();
15. else if (opt == 4) system.change();
16. else if (opt == 5) system.display();
17. else cout << "Unknown operation." << endl;
18. cout << endl;
19. }
20. return 0;
21. }

### 实训运行截图

图31 实训六运行结果

# 实验二 栈&队列实验

## 题目一 顺序栈基本操作

### 实验内容

编写一个程序，实现顺序栈的各种基本运算，并在此基础上设计一个主程序完成如下功能：

1. 初始化栈S。
2. 判断栈S是否非空。
3. 依次进栈元素a,b,c,d,e 。
4. 输出栈的长度。
5. 输出从栈顶到栈底的元素。
6. 输出出栈序列。

释放栈。

### 实验运用主要知识点

栈结构存储 {1,2,3,4}

图31 顺序栈存储结构

**语法基础**：、指针与引用、类和对象、模板与泛型编程、运算符重载

**概念基础**：采用顺序存储的栈被称为顺序栈，它利用一组地址连续的存储单元存放自栈底到栈顶的数据元素，同时附设一个指针指示当前栈顶元素的位置

**特点分析**：

1. 栈满足后进先出LIFO规则，存储的数据元素只能从一端进出
2. 由于顺序栈的入栈操作受数组上界的约束，当对栈的最大使用空间估计不足时有可能发生栈溢出的情况，此时应该及时向使用者报告消息以便做出及时的处理。或者采用动态分配空间实现顺序栈的动态扩容操作

**考察内容**：初始化顺序栈、插入元素、顺序栈属性查询、入栈操作、出栈操作

### 实验源码及备注

#### 操作说明

由于输出出栈序列可以看做是多个出栈操作，所以将顺序栈的基本操作并没有专门设置函数，改在程序中完成

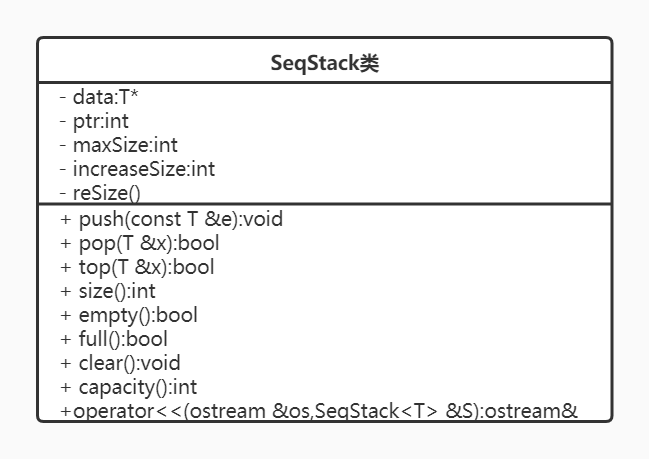


图32 顺序栈类图

|  |  |
| --- | --- |
| **成员** | **作用** |
| data | 存储数据 |
| ptr | 顺序栈中指向栈顶的指针 |
| maxSize | 记录顺序栈最大容量 |
| increaseSize | 每次扩容的大小 |

表10 顺序栈类成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名称** | SeqStack | | | |
| **类描述** | 生成顺序栈对象的模板类，包含顺序栈的需要用到的操作 | | | |
| push | push(const T&e):void | | | |
| **功能描述** | 将元素入栈 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T | 存储元素 |
| pop | pop(T &x):bool | | | |
| **功能描述** | 将栈顶元素出栈 | | |
| **返回值** | true:出栈元素成功 false:出栈元素失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| x | T | 记录出栈元素 |
| top | top(T &x):bool | | | |
| **功能描述** | 获取栈顶元素 | | |
| **返回值** | true:获取栈顶元素成功 false:获取栈顶元素失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| x | T | 记录栈顶元素 |
| size | size():int | | | |
| **功能描述** | 获取顺序栈内元素的个数 | | |
| **返回值** | 整数表示顺序栈大小 | | |
| empty | empty():bool | | | |
| **功能描述** | 检查顺序栈是否为空 | | |
| **返回值** | true：栈空 false：栈非空 | | |
| full | full():bool | | | |
| **功能描述** | 检查顺序栈是否为满 | | |
| **返回值** | true:栈满 false:栈未满 | | |
| clear | clear():void | | | |
| **功能描述** | 清空顺序栈内元素 | | |
| **返回值** | 无 | | |
| capacity | capacity():void | | | |
| **功能描述** | 获取当前顺序栈的最大容量 | | |
| **返回值** | 整数表示顺序栈的最大容量 | | |
| operator<< | operator<<(ostream &os,SeqStack<T> &S):ostream& | | | |
| **功能描述** | 重载的输出函数 | | |
| **返回值** | ostream& | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| os | ostream& | 输出流 |
| S | SeqStack<T> | 调用函数的顺序栈 |
| reSize | reSize():void | | | |
| **功能描述** | 顺序栈扩容辅助操作 | | |
| **返回值** | 无 | | |

表11 顺序栈类接口说明文档

#### 程序源码

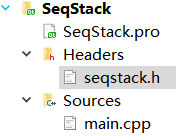


图33 顺序栈文件树

SeqStack.h文件

1. #ifndef SEQSTACK\_H
2. #define SEQSTACK\_H
3. template<class T>
4. class SeqStack
5. {
6. public:
7. SeqStack(int sz = 100);
8. ~SeqStack();
9. void push(const T &e); *// 元素压入栈*
10. bool pop(T &x);        *// 弹出元素并保存在x*
11. bool top(T &x);        *// 获取栈顶元素并保存在x*
12. int size() const;
13. bool empty() const;
14. bool full() const;
15. void clear();
16. int capacity() const;
17. friend std::ostream &operator<<(std::ostream &os, SeqStack<T> &S)   *// 放在模板类内随着实例化而实例化*
18. {
19. os << "top: " << S.data[S.ptr] << std::endl;
20. for (int i = 0; i < S.size();
21. ++i) os << i << ": " << S.data[i] << std::endl;
22. return os;
23. }
24. private:
25. T \*data;          *// 存储数据的顺序数组*
26. int ptr;          *// 栈顶遍历指针*
27. int maxSize;      *// 栈容量*
28. int increaseSize; *// 每次扩容的大小*
29. void reSize();    *// 扩容*
30. };
31. template<class T>
32. SeqStack<T>::SeqStack(int sz) : ptr(-1),
33. maxSize(sz),
34. increaseSize(sz / 2)
35. {
36. this->data = new T[maxSize];
37. std::cout << "SeqStack initialized." << std::endl;
38. }
39. template<class T>
40. SeqStack<T>::~SeqStack()
41. {
42. delete[] this->data;
43. std::cout << "SeqStack destructed." << std::endl;
44. }
45. template<class T>
46. void SeqStack<T>::push(const T &e)
47. {
48. if (this->maxSize - 1 == this->ptr) reSize();
49. data[++ptr] = e;
50. }
51. template<class T>
52. bool SeqStack<T>::pop(T &x)
53. {
54. if (this->ptr == -1) return false;
55. x = data[ptr--];
56. return true;
57. }
58. template<class T>
59. bool SeqStack<T>::top(T &x)
60. {
61. if (this->ptr == -1) return false;
62. x = this->data[this->ptr];
63. return true;
64. }
65. template<class T>
66. int SeqStack<T>::size() const
67. {
68. return this->ptr + 1;
69. }
70. template<class T>
71. bool SeqStack<T>::empty() const
72. {
73. return this->ptr == -1;
74. }
75. template<class T>
76. bool SeqStack<T>::full() const
77. {
78. return this->ptr == this->maxSize - 1;
79. }
80. template<class T>
81. void SeqStack<T>::clear()
82. {
83. this->ptr = -1;
84. }
85. template<class T>
86. int SeqStack<T>::capacity() const
87. {
88. return this->maxSize;
89. }
90. template<class T>
91. void SeqStack<T>::reSize()
92. {
93. T \*temp = new T[maxSize + increaseSize];
94. for (int i = 0; i < ptr; ++i) temp[i] = this->data[i];
95. maxSize = maxSize + increaseSize;
96. delete this->data;
97. this->data = temp;
98. }
99. #endif *// SEQSTACK\_H*

main.cpp文件

1. #include <iostream>
2. #include "seqstack.h"
3. using namespace std;
4. signed main()
5. {
6. char ch;
7. cout << "opt-1: Initialize the sequence stack S" << endl;
8. SeqStack<char> S;
9. cout << endl;
10. cout << "opt-2: Check whether the sequence stack S is empty" << endl;
11. if (S.empty()) cout << "Yes" << endl;
12. else cout << "No" << endl;
13. cout << endl;
14. cout << "opt-3: Push elements a,b,c,d and e onto the stack" << endl;
15. for (char i = 'a'; i <= 'e'; ++i) {
16. S.push(i);
17. cout << "push " << i << " succeed." << endl;
18. }
19. cout << endl;
20. cout << "opt-4: Output the size of stack" << endl;
21. cout << S.size() << endl;
22. cout << endl;
23. cout << "opt-5: Outputs elements from the bottom to the top of the stack" <<
24. endl;
25. cout << S << endl;
26. cout << "opt-6: Output stack sequence" << endl;
27. while (!S.empty()) {
28. S.pop(ch);
29. cout << ch << ' ';
30. }
31. cout << endl << endl;
32. cout << "opt-7: Release SeqStack S" << endl;
33. return 0;
34. }

### 实验运行截图及说明

#### 运行截图

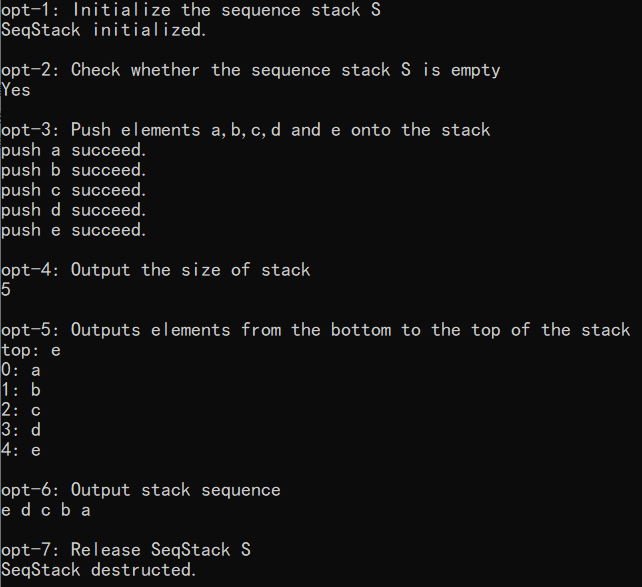


图33 顺序栈实验运行结果

#### 程序说明

**栈顶指针**：初始设置为-1说明没有元素存在，第一个元素存在数组的下标0位置

**进栈操作**：顺序栈不满时栈顶指针先加1再送元素到栈顶元素

**出栈操作**：顺序栈非空的时候先取出栈顶元素再将栈顶指针减1

**栈空条件**：当栈顶指针值为-1时说明顺序栈为空

**栈满操作**：当栈顶指针指向当前栈内最大容量-1时说明顺序栈满

**读取栈顶元素操作**：仅为读取栈顶元素并没有出栈操作因此原栈顶元素依然保存在栈中

**输出栈顶到栈底操作**：通过重载输出运算符规定输出流的输出格式，需要注意的是由于当前为类的模板泛型编程所以该操作函数需要放到类内部定义

#### 体会反思

1. 模板泛型编程的原理——与平常的代码编译原理不同，泛型编程作为模板只有在需要用到的时候才进行编译，因此代码必须在定义时规定好内部用到成员的在编译时选择的数据类型
2. 栈的扩容机制——vector一般采用二倍扩容机制，但是网络上并没有查到vector的扩容方法，这里暂时采用线性扩容存储空间

## 题目二 循环顺序队列基本操作

### 实验内容

编写一个程序，实现循环顺序队列的各种基本运算，并在此基础上设计一个主程序完成如下功能：

1. 初始化队列Q。
2. 判断队列Q是否非空。
3. 依次进队列元素a,b,c。
4. 出队一个元素，输出该元素。
5. 输出队列Q的元素个数。
6. 依次进入队列元素d,e,f。
7. 输出出队序列。
8. 释放队列。

规定采用下述空间调整方法：队列插入函数中当队列满并需要插入新元素时将数组空间扩大一倍使新元素得以插入，队列删除函数中当队列元素少于数组空间的1/4时将数组空间自动缩减一半。

### 实验运用主要知识点

**语法基础**：、指针与引用、类和对象、模板与泛型编程、运算符重载

**概念基础**：顺序队列是指分配一块连续的存储单元存放队列中的元素，并附设两个指针：队头指针front指向队头元素和队尾指针rear指向队尾元素的下一个位置

**特点分析**：

1. 队列只允许在表的一端进行插入元素操作，而在另一端进行删除元素操作
2. 操作的特性是先进先出FIFO规则
3. 队尾元素的下一个位置存放的是队首元素

**考察内容**：初始化循环顺序队列、插入元素、循环顺序队列属性查询、入队操作、出队操作

### 实验源码及备注

#### 操作说明

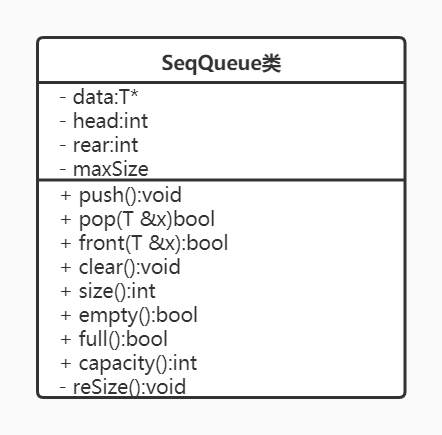


图33 循环顺序队列类图

|  |  |
| --- | --- |
| **成员** | **作用** |
| data | 存储数据 |
| ptr | 顺序栈中指向栈顶的指针 |
| maxSize | 记录顺序栈最大容量 |
| increaseSize | 每次扩容的大小 |

表11 循环顺序队列类成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名称** | SeqQueue | | | |
| **类描述** | 生成顺序队列对象的模板类，包含顺序队列的需要用到的操作 | | | |
| push | push(const T&e):void | | | |
| **功能描述** | 将元素出队 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T | 存储入队的元素 |
| pop | pop(T &x):bool | | | |
| **功能描述** | 将元素出队 | | |
| **返回值** | true：出队元素成功 false:出队元素失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| x | T | 存储出队的元素 |
| front | front(T &x):bool | | | |
| **功能描述** | 获取队首元素 | | |
| **返回值** | true：获取队首元素成功 false：获取队首元素失败 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| x | T | 存储获取的队首元素 |
| clear | clear():void | | | |
| **功能描述** | 清空循环顺序队列内元素 | | |
| **返回值** | 无 | | |
| size | size():int | | | |
| **功能描述** | 获取循环顺序队列内元素个数 | | |
| **返回值** | 整数表示循环顺序队列大小 | | |
| empty | empty():bool | | | |
| **功能描述** | 判断循环顺序队列是否为空 | | |
| **返回值** | true:循环顺序队列为空 false:循环顺序队列非空 | | |
| full | full():bool | | | |
| **功能描述** | 判断循环顺序队列是否为满 | | |
| **返回值** | true:循环顺序队列已满 false:循环顺序队列未满 | | |
| capacity | capacaity():int | | | |
| **功能描述** | 获取循环顺序队列的最大容量 | | |
| **返回值** | 整数为循环顺序队列的最大容量 | | |
| reSize | reSize(int newSize):void | | | |
| **功能描述** | 循环顺序队列的辅助扩容操作 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| newSize | int | 想要扩容的大小 |

表12 循环顺序队列类接口文档

#### 程序源码

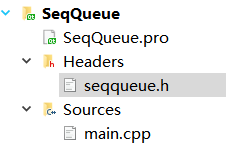


图35 循环顺序队列文件树

seqqueue.h文件

1. #ifndef SEQQUEUE\_H
2. #define SEQQUEUE\_H
3. template<class T>
4. class SeqQueue {
5. public:
6. SeqQueue(int sz = 4);
7. ~SeqQueue();
8. void push(const T& e);
9. bool pop(T& x);
10. bool front(T& x);
11. void clear();
12. int  size() const;
13. bool empty() const;
14. bool full() const;
15. int capacity() const;
16. private:
17. T \*data;
18. int head, rear, maxSize; *// 一个元素的下标,最后一元素后一个位置的下标,最大空间*
19. void reSize(int newSize);
20. };
21. template<class T>
22. SeqQueue<T>::SeqQueue(int sz) : head(0), rear(0), maxSize(sz) {
23. this->data = new T[maxSize];
24. std::cout << "SeqQueue initialized." << std::endl;
25. }
26. template<class T>
27. void SeqQueue<T>::push(const T& e) {
28. if (this->full()) reSize(maxSize \* 2);
29. this->data[this->rear] = e;
30. this->rear = (this->rear + 1) % maxSize;
31. }
32. template<class T>
33. SeqQueue<T>::~SeqQueue() {
34. delete[]this->data;
35. std::cout << "SeqQueue destructed." << std::endl;
36. }
37. template<class T>
38. bool SeqQueue<T>::pop(T& x) {
39. if (this->empty()) return false;
40. x = this->data[head];
41. head = (head + 1) % maxSize;
42. if ((this->maxSize > 4) && (this->size() < this->maxSize / 4)) reSize(
43. this->maxSize / 2);
44. return true;
45. }
46. template<class T>
47. bool SeqQueue<T>::front(T& x) {
48. if (this->empty()) return false;
49. x = this->data[head];
50. return true;
51. }
52. template<class T>
53. void SeqQueue<T>::clear() {
54. head = rear;
55. reSize(4);
56. }
57. template<class T>
58. int SeqQueue<T>::size() const {
59. return (rear - head + maxSize) % maxSize;
60. }
61. template<class T>
62. bool SeqQueue<T>::empty() const {
63. return this->rear == this->head;
64. }
65. template<class T>
66. bool SeqQueue<T>::full() const {
67. return (this->rear + 1) % maxSize == this->head;
68. }
69. template<class T>
70. int SeqQueue<T>::capacity() const {
71. return this->maxSize;
72. }
73. template<class T>
74. void SeqQueue<T>::reSize(int newSize) {
75. if (size() > this->maxSize) return;
76. T  \*temp = new T[newSize];
77. int p = 0;
78. while (this->head != this->rear) {
79. temp[p++] = this->data[this->head];
80. this->head = (this->head + 1) % this->maxSize;
81. }
82. delete[]this->data;
83. this->data = temp;
84. this->head = 0, this->rear = p;
85. this->maxSize = newSize;
86. }
87. #endif *// SEQQUEUE\_H*

main.cpp文件

1. #include <iostream>
2. #include "seqqueue.h"
3. using namespace std;
4. signed main()
5. {
6. char ch;
7. cout << "opt-1: Initialize the sequence queue Q" << endl;
8. SeqQueue<char> Q;
9. cout << endl;
10. cout << "opt-2: Check whether the sequence queue Q is empty" << endl;
11. if (Q.empty()) cout << "Yes" << endl;
12. else cout << "No" << endl;
13. cout << endl;
14. cout << "opt-3: Push elements a,b,c into the queue" << endl;
15. for (char i = 'a'; i <= 'c'; ++i) {
16. Q.push(i);
17. cout << "push " << i << " succeed." << endl;
18. }
19. cout << endl;
20. cout << "opt-4: Extract an element and print the element" << endl;
21. Q.pop(ch);
22. cout << ch << endl;
23. cout << endl;
24. cout << "opt-5: Out the size of queue" << endl;
25. cout << Q.size() << endl;
26. cout << endl;
27. cout << "opt-6: Push elements d,e,f into the queue" << endl;
28. for (char i = 'd'; i <= 'f'; ++i) {
29. Q.push(i);
30. cout << "push " << i << " succeed." << endl;
31. }
32. cout << endl;
33. cout << "opt-7: Output queue sequence" << endl;
34. while (!Q.empty()) {
35. Q.pop(ch);
36. cout << ch << ' ';
37. }
38. cout << endl << endl;
39. cout << "opt-8: Release SeqQueue Q" << endl;
40. return 0;
41. }

### 实验运行截图及说明

#### 运行截图

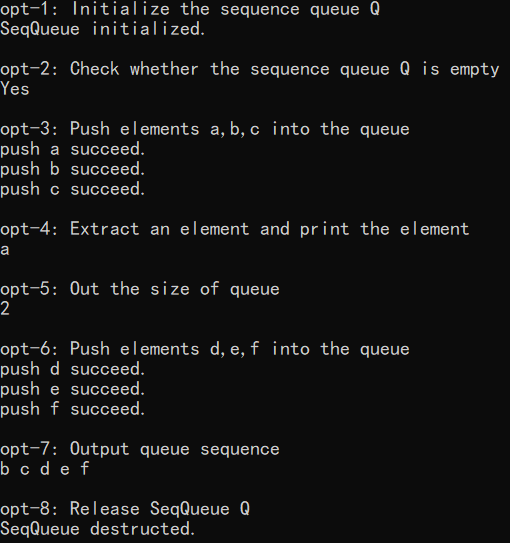


图36 循环顺序队列实验运行结果

#### 程序说明

**初始化操作**：将队首指针和队尾指针指向同一个位置

**入队操作&出队操作**：将队尾指针加1或者减1后对最大容量maxSize取模实现队列的循环存储

**判断队空操作**：当队首指针的指针域指向自己的时候说明队空

**判断队满操作**：当队尾指针的指针域指向自己的时候说明队满

**获取队列大小操作**：队尾指针和队首指针记录的下标相减后的最小正数值

**解决判断操作冲突**：为了区分队空还是队满的情况有如下三种处理按时：

一是牺牲一个单元来区分队空和队满（也就是本实验采用的方法），入队是少用一个队列单元，这是一种较为普遍的做法，约定“队首指针再队尾指针的下一个位置作为队满的操作”

二是类型中增设表示元素个数的数据成员

三是类型中增设一个标记成员以区分队满还是队空

#### 体会反思

1. 注意区分循环链表和循环顺序队列的区别——在循环顺序队列中指针操作实际上是数组下标，不同于循环链表直接指向物理地址因此在判断队满&队空操作以及求队列大小操作的方法不同于之前的循环链表
2. 扩容方法的理解——一个好的扩容方法可以提高数据结构的处理效率并更加合理的分配空间

## 实训一（1） 借助栈实现单链表翻转

### 实训题目

借助栈实现单链表的逆置运算

### 实训思路与源码

由于栈的LIFO规则所以很容易想到将链表元素全部入栈后出栈时建表即可

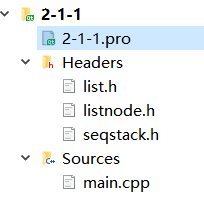


图37 实训一（1）文件树

listnode.h&list.h&seqstack.h略

main.cpp文件

1. #include <iostream>
2. #include <string>
3. #include "seqstack.h"
4. #include "list.h"
5. using namespace std;
6. signed main()
7. {
8. string str = "hello\_world";
9. List<char> L, RL;
10. for (auto e:str) L.push\_back(e);
11. SeqStack<char> S;
12. cout << "before: "; L.output();
13. char ch;
14. for (int i = 1; i <= L.length(); ++i) {
15. L.getData(i, ch);
16. S.push(ch);
17. }
18. while (!S.empty()) {
19. S.pop(ch);
20. RL.push\_back(ch);
21. }
22. cout << "after: "; RL.output();
23. return 0;
24. }

### 实训运行截图

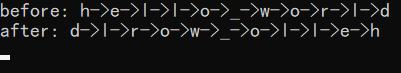


图38 实训一（1）运行结果

## 实训一（2）递归求值

### 实训题目

已知A[n]为整数驻足，试写出实现下述算法的递归算法：

（1）求数组A中的最大整数

（2）求n个整数的和

（3）求n个整数的平均值

### 实训思路与源码

通过分治的思想将大问题先分解为小的问题递归求解，小问题解决完后回溯供大问题进行求解

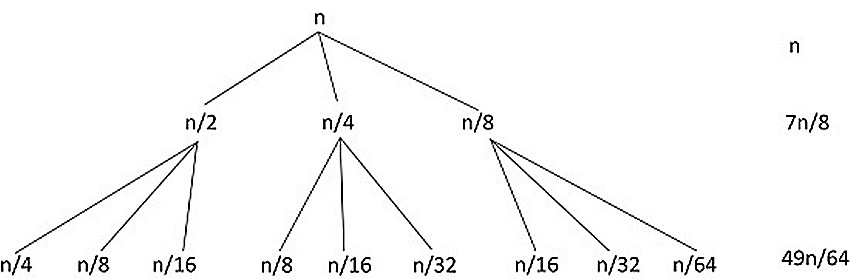


图39 实训一（2）思路

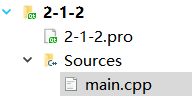


图40 实训一（2）文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. int A[5] = { 5, 33, 26, 78, 25 }; *// 分治*
4. int getmax(int l, int r)
5. {
6. if (l == r) return A[l];
7. int mid = (l + r) >> 1;
8. return max(getmax(l, mid), getmax(mid + 1, r));
9. }
10. int getsum(int l, int r)
11. {
12. if (l == r) return A[l];
13. int mid = (l + r) >> 1;
14. return getsum(l, mid) + getsum(mid + 1, r);
15. }
16. double getaverage(int l, int r)
17. {
18. if (l == r) return A[l];
19. int mid = (l + r) >> 1;
20. return (getaverage(l,
21. mid) \* (mid - l + 1)+getaverage(mid + 1, r) \* (r - mid)) \* 1.0 / (r - l + 1);
22. }
23. signed main()
24. {
25. for (auto e:A) cout<<e<<' ';
26. cout<<endl;
27. cout << "max element: " << getmax(0, 4) << endl;
28. cout << "sum: " << getsum(0, 4) << endl;
29. cout << "average: " << getaverage(0, 4) << endl;
30. return 0;
31. }

### 实训运行截图

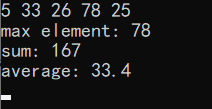


图41 实训一（2）运行结果

## 实训一（3）递归求全排列

### 实训题目

编写一个递归算法输出区间内这n个元素的全排列

### 实训思路与源码

每次尝试选择一个当前符合条件的数，函数递归调用至n次后说明找到全排列的一种情况

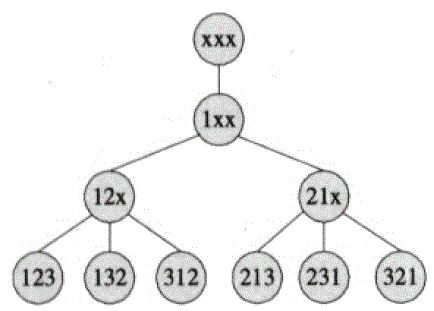


图42 实训一（3）思想

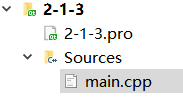


图43 实训一（3）文件树

main.cpp文件

1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. using namespace std;
5. const int maxn = 1e3 + 5;
6. int  n, cnt, temp[maxn];
7. bool vis[maxn];
8. void dfs(int pos) {
9. if (pos == n + 1) {
10. ++cnt;
11. for (int i = 1; i <= n; ++i) cout << temp[i] << ' ';
12. cout << endl;
13. return;
14. }
15. for (int i = 1; i <= n; ++i) {
16. if (!vis[i]) {
17. vis[i] = true;
18. temp[pos] = i;
19. dfs(pos + 1);
20. vis[i] = false;
21. }
22. }
23. }
24. signed main()
25. {
26. cout << "Input n: "; cin >> n;
27. dfs(1);
28. cout << "Total :" << cnt << endl;
29. return 0;
30. }

### 实训运行截图



图44 实训一（3）运行结果

## 实训二 利用栈表达式求值

### 实训题目

利用栈对只含二目运算符的中缀算术表达式求值，并将该中缀表达式转换为后缀表达式。算术运算符包括：\*、/、+、-，优先级从高到低。

### 实训思路与源码

中缀表达式即数学中常用写法，程序想要进行计算需要现利用栈将表达式转换为后缀表达式（栈用来进出运算的符号），并再借助栈完成后缀表达式的求值（栈用来进出运算的数字）

中缀表达式转换为后缀表达式思想：从左到右遍历中缀表达式的每个数字和符号，若是数字就输出，即成为后缀表达式的一部分；若是符号，则判断其与栈顶符号的优先级，是右括号或优先级低于找顶符号（乘除优先加减）则栈顶元素依次出栈并输出，并将当前符号进栈，一直到最终输出后缀表达式为止

后缀表达式求值的思想：设置一个栈，开始时，栈为空，然后从左到右扫描后缀表达式，若遇操作数，则进栈；若遇运算符，则从栈中退出两个元素，先退出的放到运算符的右边，后退出的 放到运算符左边，运算后的结果再进栈，直到后缀表达式扫描完毕。此时，栈中仅有一个元素，即为运算的结果

为了减少字符判断的操作，本实训输入采用空格分隔每个字符主要实现应用栈的核心操作

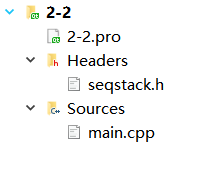


图45 实训二文件树

seqstack.h文件略

main.cpp文件

1. #include <iostream>
2. #include <map>
3. #include "seqstack.h"
4. using namespace std;
5. map<char, int> val;
6. char op[] = "()-+/\*"; *// 优先级*
7. SeqStack<char> S;
8. SeqStack<int> ans;
9. string function;
10. inline bool check(string input)
11. {
12. return input[0] >= '0' && input[0] <= '9';
13. }
14. void getsum(char op)
15. {
16. int a, b, x;
17. ans.top(b);
18. ans.pop(x);
19. ans.top(a);
20. ans.pop(x);
21. int sum = 0;
22. if (op == '\*') sum = a \* b;
23. else if (op == '/') sum = a / b;
24. else if (op == '+') sum = a + b;
25. else sum = a - b;
26. ans.push(sum);
27. }
28. signed main()
29. {
30. for (int i = 0; i < 6; ++i) val[op[i]] = i;
31. *// freopen("in.txt", "r", stdin);*
32. string input;
33. char ch, x;
34. while (cin >> input) { *// 1 \* 2 + 5 - ( 6 + 7 - ( 5 + 71 ) \* 19 ) - 15 \* 14*
35. if (check(input)) {
36. function += input;
37. int num = stoi(input);
38. *// cout << num << ' ';*
39. ans.push(num); *// 数字压入栈*
40. continue;
41. }
42. ch = input[0];
43. switch (ch) {
44. case '(':
45. S.push(ch);
46. break;
47. case ')':
48. S.top(x);
49. while (!S.empty() && x != '(') {
50. function += x;
51. *// cout << x << ' ';*
52. getsum(x);
53. S.pop(x);
54. S.top(x);
55. }
56. S.pop(x);
57. break;
58. default:
59. S.top(x);
60. while (!S.empty() && val[x] >= val[ch]) {
61. function += x;
62. *// cout << x << ' ';*
63. getsum(x);
64. S.pop(x);
65. S.top(x);
66. }
67. S.push(ch);
68. break;
69. }
70. }
71. while (!S.empty()) {
72. S.top(x);
73. function += x;
74. *// cout << x << ' ';*
75. getsum(x);
76. S.pop(x);
77. }
78. int res;
79. ans.top(res);
80. cout << function << endl;
81. cout << res << endl;
82. return 0;
83. }

### 实训运行截图

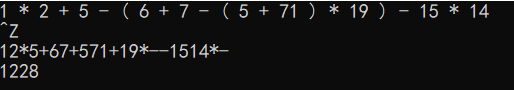


图46 实训二运行结果

## 实训三 利用栈求解迷宫路径

### 实训题目

用栈求解下图所示迷宫的所有路径，并输出最短路径长度及该最短路径。

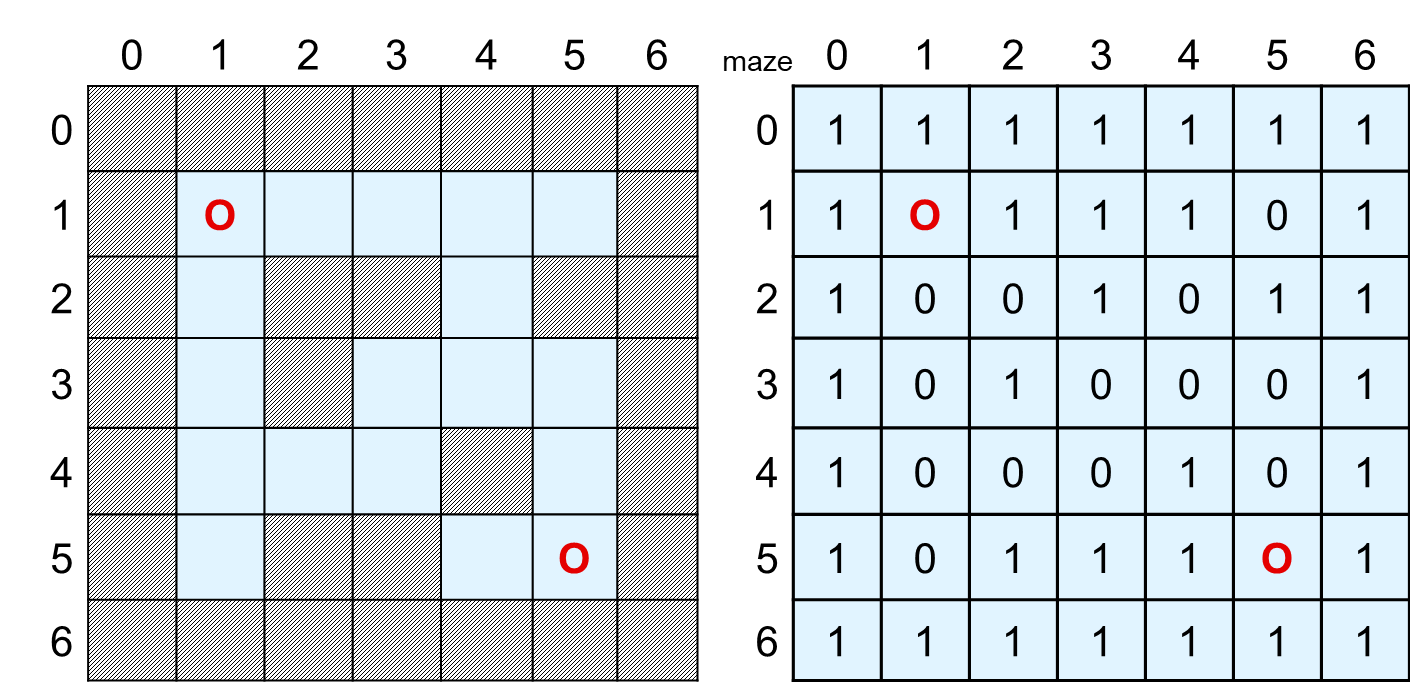


图47 实训三题目插图

### 实训思路与源码

利用栈模拟广度优先搜索，以下述一道例题说明

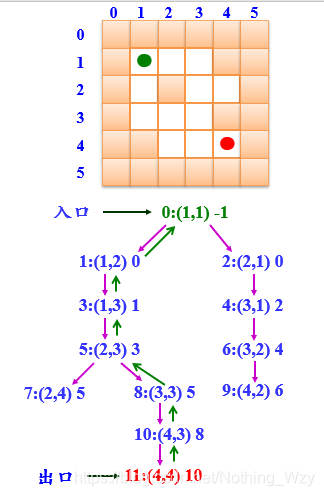


图48 实训三思想

利用栈存储每个尝试走的状态，走完一个状态后回退，当走到终点的时候即找到一个路径并输出路径

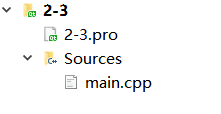


图49 实训三文件树

main.cpp文件

1. #include <iostream>
2. #include <stack>
3. #include <vector>
4. #include <algorithm>
5. using namespace std;
6. const int maxn = 15;
7. const int dir[][2] = { { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 } };
8. struct node {
9. int x, y;
10. node() = default;
11. node(int x, int y) : x(x),
12. y(y)
13. {
14. }
15. node operator=(const node &A)
16. {
17. this->x = A.x, this->y = A.y;
18. return \*this;
19. }
20. bool operator==(const node &A)
21. {
22. return this->x == A.x && this->y == A.y;
23. }
24. };
25. int minn = 15;
26. stack<node> ans;
27. bool vis[maxn][maxn];
28. node pre[maxn][maxn]; *// 前序结点*
29. node st(1, 1), ed(5, 5);
30. bool mz[maxn][maxn] = {
31. { 0, 0, 0, 0, 0, 0, 0 },
32. { 0, 1, 1, 1, 1, 1, 0 },
33. { 0, 1, 0, 0, 1, 0, 0 },
34. { 0, 1, 0, 1, 1, 1, 0 },
35. { 0, 1, 1, 1, 0, 1, 0 },
36. { 0, 1, 0, 0, 1, 1, 0 },
37. { 0, 0, 0, 0, 0, 0, 0 }
38. };
39. bool check(node now)
40. {
41. return mz[now.x][now.y] == 1 && vis[now.x][now.y] == 0;
42. }
43. void trv(stack<node> s)
44. {
45. node temp = s.top();
46. cout << "(" << temp.x << "," << temp.y << ")";
47. s.pop();
48. while (s.size()) {
49. temp = s.top();
50. cout << "->(" << temp.x << "," << temp.y << ")";
51. s.pop();
52. }
53. cout << endl;
54. }
55. void dfs(node now, int depth)
56. {
57. vis[now.x][now.y] = 1;
58. if (now == ed) {
59. stack<node> temp;
60. node a = now;
61. while (!(a == st)) {
62. temp.push(a);
63. a = pre[a.x][a.y];
64. }
65. temp.push(st);
66. trv(temp);
67. if (depth < minn) minn = depth, ans = temp;
68. return;
69. }
70. for (int i = 0; i < 4; ++i) {
71. node nex(now.x + dir[i][0], now.y + dir[i][1]);
72. if (check(nex)) {
73. pre[nex.x][nex.y] = now;
74. dfs(nex, depth + 1);
75. vis[nex.x][nex.y] = 0;
76. }
77. }
78. }
79. signed main()
80. {
81. dfs(st, 0);
82. cout << "Min len: " << minn << endl;
83. cout << "Min route: ";
84. trv(ans);
85. return 0;
86. }

### 实训运行截图

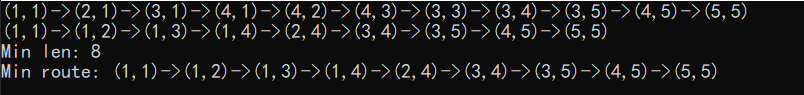


图50 实训三运行结果

## 实训四 八皇后问题

### 实训题目

设在初始状态下在国际象棋棋盘上没有任何棋子（皇后）。然后顺序在第1行，第2行……第8行上布放棋子。在每一行中有8个可选择位置，但在任一时刻，棋盘的合法布局都必须满足3个限制条件，即任何两个棋子不得放在棋盘上的同一行、或者同一列、或者同一斜线上。是编写一个递归算法，求解并输出此问题的所有合法布局。（提示：用回溯法。在第i行第j列安放一个棋子时，需要记录在行方向、列方向、正斜线方向、反斜线方向的安放状态，若当前布局合法，可想下一行递归求解，否则可移走这个棋子，恢复安放该棋子的前的状态，试探本行的第j+1列）

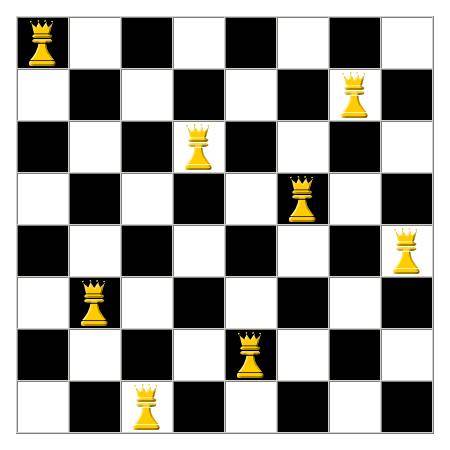


图51 实训四题目插图

### 实训思路与源码

在题目给出的思路上选择对行进行递归求解，在递归到的行上尝试放在各列直至放完所有的行，则请找到一个答案并进行记录

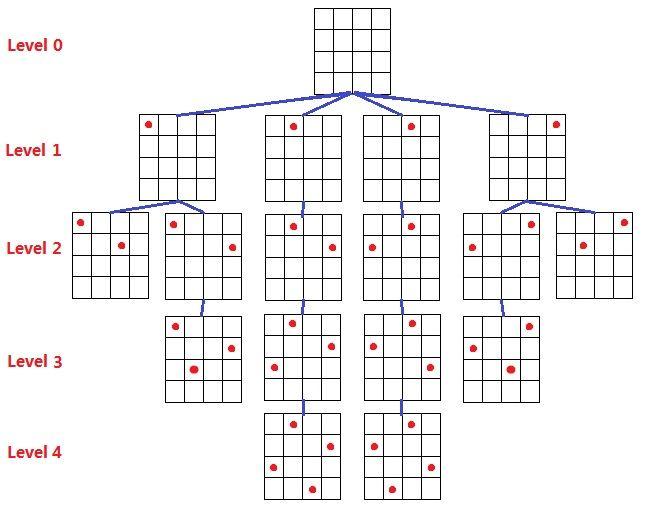


图52 实训四思路

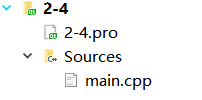


图53实训四文件树

main.cpp文件

1. #include <bits/stdc++.h>
2. using namespace std;
3. const int N = 15;
4. struct node {
5. int x, y;
6. node(int x, int y) : x(x),
7. y(y)
8. {
9. }
10. };
11. vector<node> vec;
12. vector<vector<node> > ans;
13. */\*皇后肯定是每一行放一个，我要考虑怎末放列数\*/*
14. int n;
15. int sum = 0; *// 记录器*
16. int x[N]; *// 标记皇后放置的列数*
17. bool place(int k)   *// 判断此列数是否可以存放*
18. {
19. for (int i = 1; i < k; i++)
20. if (abs(k-i) == abs(x[k]-x[i]) || x[k] == x[i])
21. *// abs是C++的绝对值函数，任意两行的皇后不能够相邻或者同列*
22. return 0;
23. return 1;
24. }
25. int queen(int t)
26. {
27. if (t > n) { *// 若是皇后全部能够摆完，即当放置的皇后超过n时，可行解个数加1，此时n必须大于0*
28. sum++;
29. ans.push\_back(vec);
30. } else
31. for (int i = 1; i <= n; i++) {*// 通过枚举来确定一行皇后可以放的位置*
32. x[t] = i; *// 标明第t个皇后放在第i列*
33. if (place(t)) { *// 如果可以放在某一位置，则继续放下一皇后*
34. vec.push\_back(node(t, i));
35. queen(t+1);
36. vec.pop\_back();
37. }
38. }
39. return sum;
40. }
41. signed main()
42. {
43. while (cin>>n)
44. {
45. ans.clear();
46. vec.clear();
47. memset(x, 0, sizeof x);
48. sum = 0;
49. if (n == 0) break;
50. queen(1);
51. cout<<sum<<endl;
52. for (auto e1:ans) {
53. for (auto e2:e1)
54. printf("(%d, %d) ", e2.x, e2.y);
55. cout<<endl;
56. }
57. }
58. return 0;
59. }

### 实训运行截图

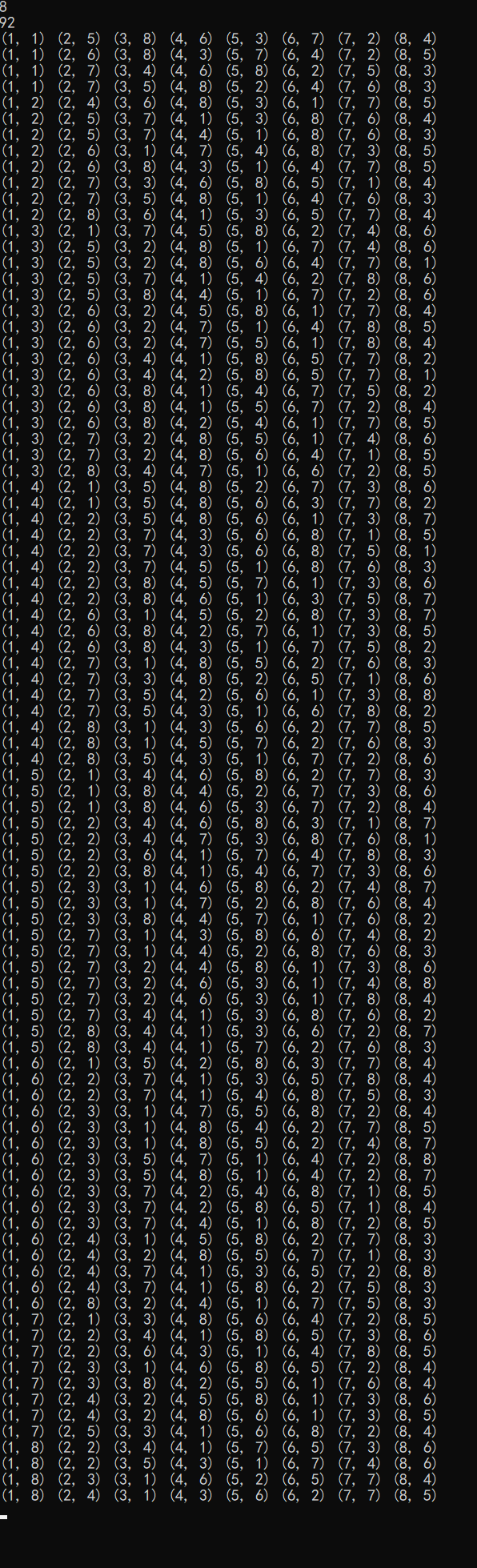
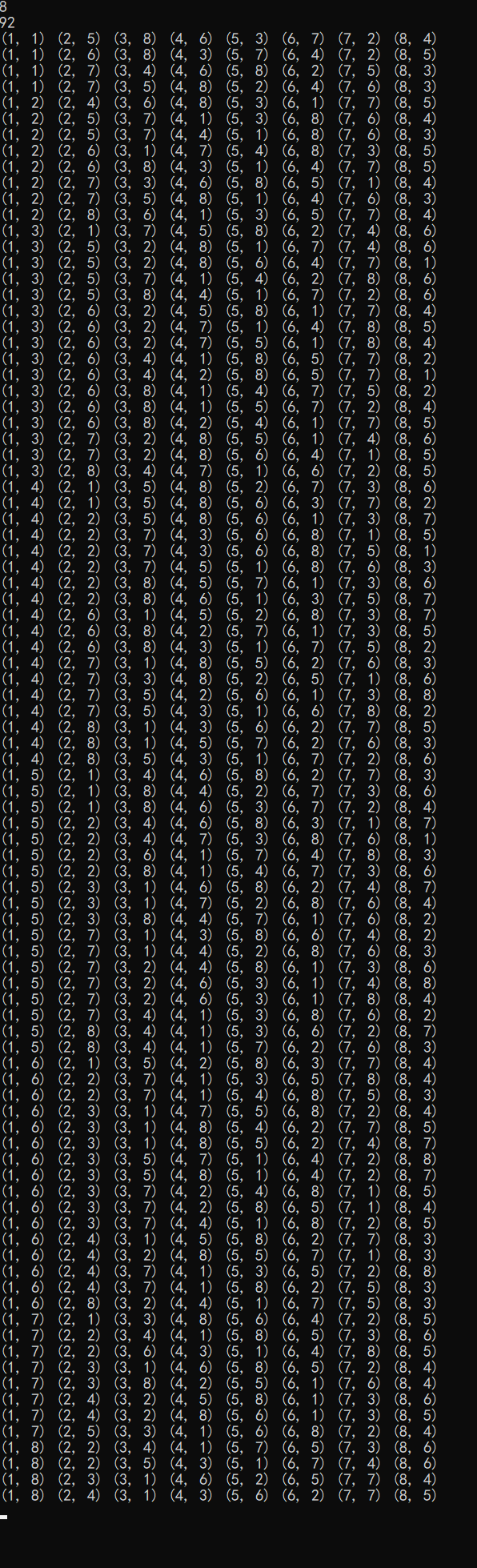


图54 实训四运行结果

## 实训五 病人看病模拟程序

### 实训题目

编写一个程序，反映病人到医院看病，排队看医生的情况。在病人排队过程中，主要重复两件事：

1. 病人到达诊室，将病历本交给护士，排到等待队列中候诊
2. 护士从等待队列中取出下一位病人的病历，该病人进入诊室就诊

要求模拟病人等待就诊这一过程。程序采用菜单方式，其选项及功能说明如下：

* + 排队――输入排队病人的病历号，加入病人排队队列中；
  + 就诊――病人排队队列中最前面的病人就诊，并将其从队列中删除；
  + 查看排队――从对首到队尾列出所有的排队病人的病历号；
  + 不再排队，余下依次就诊――从对首到队尾列出所有的排队病人的病历号，并退出运行；
  + 下班――退出运行。

### 实训思路与源码

直接main主程序借助实验完成的顺序栈对应场景完成操作

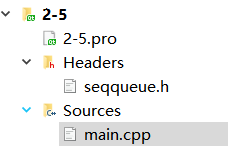


图55 实训五文件树

seqqueue.h文件略

main.cpp文件

1. #include <iostream>
2. #include "seqqueue.h"
3. using namespace std;
4. void menu()
5. {
6. std::cout << "================================" << std::endl;
7. std::cout << "\*\*\*\* hospital manage system \*\*\*\*" << std::endl;
8. std::cout << "\*\*\*\*    [0] exit            \*\*\*\*" << std::endl;
9. std::cout << "\*\*\*\*    [1] line up         \*\*\*\*" << std::endl;
10. std::cout << "\*\*\*\*    [2] see a doctor    \*\*\*\*" << std::endl;
11. std::cout << "\*\*\*\*    [3] check queue     \*\*\*\*" << std::endl;
12. std::cout << "\*\*\*\*    [4] in order        \*\*\*\*" << std::endl;
13. std::cout << "================================" << std::endl;
14. std::cout << "Input your choice: ";
15. }
16. signed main()
17. {
18. SeqQueue<string> Q;
19. string temp;
20. while (1) {
21. menu();
22. int opt;
23. cin >> opt;
24. if (opt == 0) {
25. cout << "Thanks for using!" << endl;
26. break;
27. } else if (opt == 1) {
28. cout << "Input your Medical record: ";
29. cin >> temp;
30. Q.push(temp);
31. cout << temp << " in queue" << endl;
32. } else if (opt == 2) {
33. if (Q.empty()) cout << "No queues." << endl;
34. else {
35. Q.pop(temp);
36. cout << temp << " see a doctor." << endl;
37. }
38. } else if (opt == 3) {
39. SeqQueue<string> q(Q);
40. while (!q.empty()) {
41. q.pop(temp);
42. cout << temp << " in queue." << endl;
43. }
44. } else if (opt == 4) {
45. while (!Q.empty()) {
46. Q.pop(temp);
47. cout << temp << " see a doctor." << endl;
48. }
49. } else cout << "Invalid operation." << endl;
50. cout << endl;
51. }
52. return 0;
53. }

### 实训运行截图

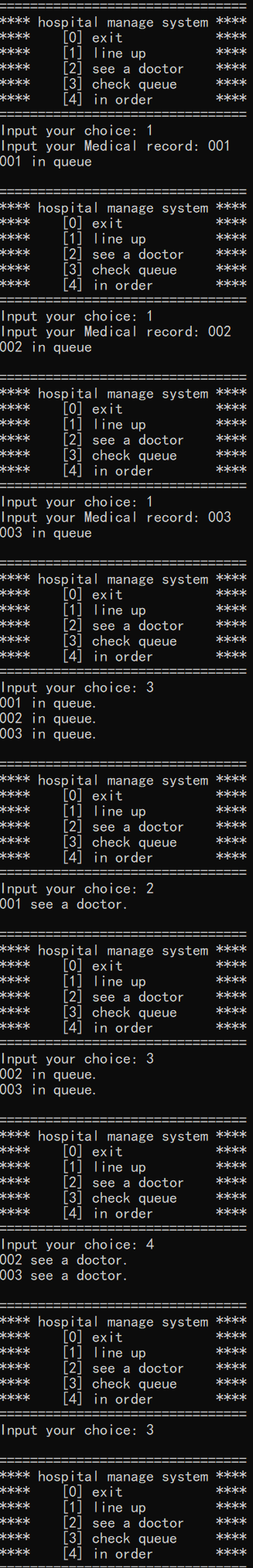
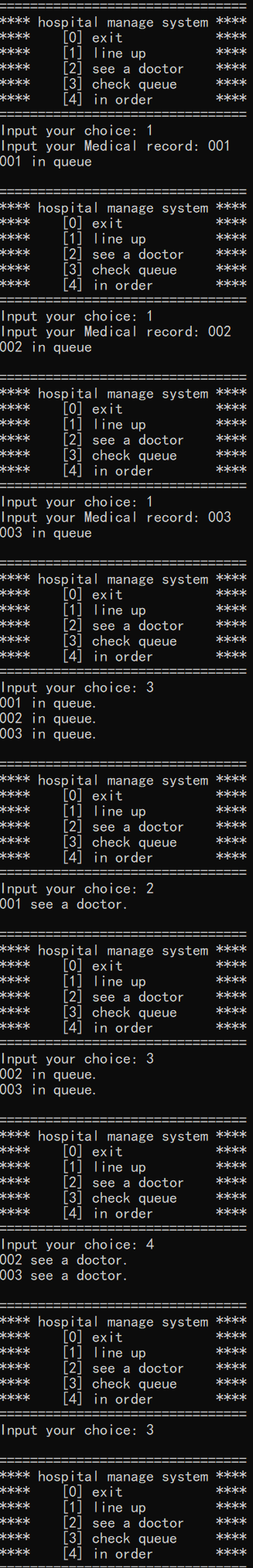


图56 实训五运行结果

# 实验三 串&数组实验

## 题目一 串的简单匹配算法

### 实验题目

采用顺序存储结构存储串，编写一个程序，采用简单模式匹配方法求串s中出现的第一个最长重复子串的下标和长度。例如：s=“aababcabcdabcde”，最长重复子串为：abcd

### 实验运用主要知识点

**语法基础**：、指针与引用、标准模板库、函数

**概念基础**：BF算法，即暴力算法，是普通的模式匹配算法，BF算法的思想就是将目标串S的第一个字符与模式串T的第一个字符进行匹配，若相等，则继续比较S的第二个字符和 T的第二个字符；若不相等，则比较S的第二个字符和T的第一个字符，依次比较下去，直到得出最后的匹配结果。BF算法是一种蛮力算法

### 实验源码及备注

#### 操作说明

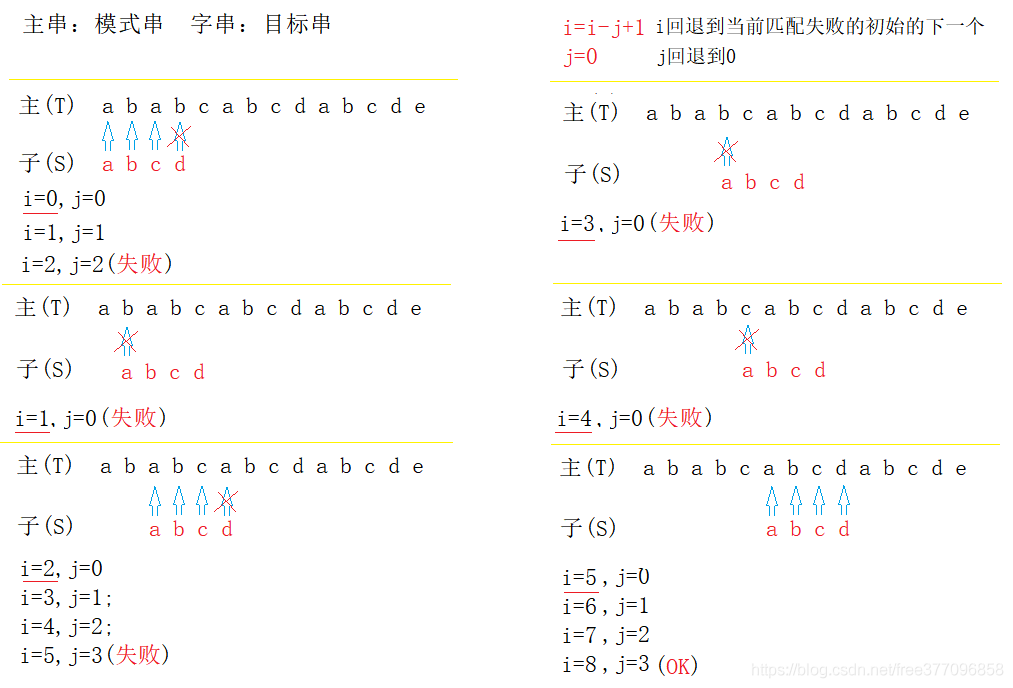


图57 BF算法说明

#### 程序源码

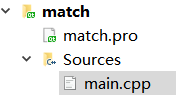


图58 题目一文件树

main.cpp文件

1. #include <iostream>
2. #include <string>
3. #include <vector>
4. #include <algorithm>
5. using namespace std;
6. typedef pair<string, int> P;
7. int count(string &str, string &patt)
8. {
9. int cnt = 0;
10. for (int i = 0; i < str.size() - patt.size() + 1; ++i) {
11. bool flag = 1;
12. for (int j = 0; j < patt.size(); ++j) {
13. if (patt[j] != str[j + i]) {
14. flag = 0;
15. break;
16. }
17. }
18. if (flag) ++cnt;
19. }
20. return cnt;
21. }
22. signed main()
23. {
24. string s = "aababcabcdabcde";
25. int len = s.size();
26. vector<P> ans;
27. for (int l = 1; l <= len; ++l) {
28. for (int i = 0; i + l < len; ++i) {
29. string temp = s.substr(i, l);
30. int cnt = count(s, temp);
31. if (cnt >= 2) ans.push\_back(P(temp, cnt));
32. }
33. }
34. sort(ans.begin(), ans.end(), [=](P a, P b) {
35. if (a.first.size() == b.first.size()) return a.second > b.second;
36. return a.first.size() > b.first.size();
37. });
38. ans.erase(unique(ans.begin(), ans.end()), ans.end()); *// 假去重+移除*
39. for (auto e:ans) cout << e.first << ' ' << e.second << endl;
40. cout << "Result: " << ans[0].first << ' ' << ans[0].second << endl;
41. return 0;
42. }

### 实验运行截图及说明

#### 运行截图

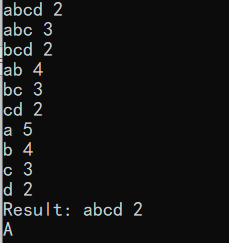


图59 题目一运行截图

#### 程序说明

**字典序:** 在字典中，单词是按照首字母在字母表中的顺序进行排列的，比如在之前。而第一个字母相同时，会去比较两个单词的第二个字母在字母表中的顺序，比如在之前，以此类推

**函数**:捕获外界的函数并进行函数内部的运算

**结果说明:**为了更清晰的看到匹配的结果，本实验的代码选择将暴力匹配尝试的字符串均记录统计并按照字典序将这些字符串排序输出

**时间复杂度**：

#### 体会反思

算法的引入——之前经常忘记算法的思路，经过这次手写匹配算法了解了匹配算法效率低的原因，对于算法减少回溯的思想有了更深的理解，其实每个优秀算法的诞生都是在前人算法的缺点基础上做出的进一步改进从而加快效率

## 题目二 串的算法

### 实验题目

编写一个程序，利用KMP算法求子串t在主串是中出现的次数，并以s=”aaabbdaabbde”,t=“aabbd”为例，显示其匹配过程。（匹配过程的显示选做）。

### 实验运用主要知识点

**语法基础**：、指针与引用

**概念基础**：KMP 算法是 D.E.Knuth、J,H,Morris 和 V.R.Pratt 三位共同提出的，称之为 Knuth-Morria-Pratt 算法，简称KMP算法。该算法相对于Brute-Force（暴力）算法有比较大的改进，主要是消除了主串指针的回溯，从而使算法效率有了某种程度的提高。主要步骤分为两个：求解数组以及匹配匹配串和模式串

**特点分析**：

1. 数组处理对象是模式串自身，在三分法中有很多关于数组的定义与求法，本实验选择如下定义：若，则为满足以下条件的最大值：相同，不同。若不存在这样的，则

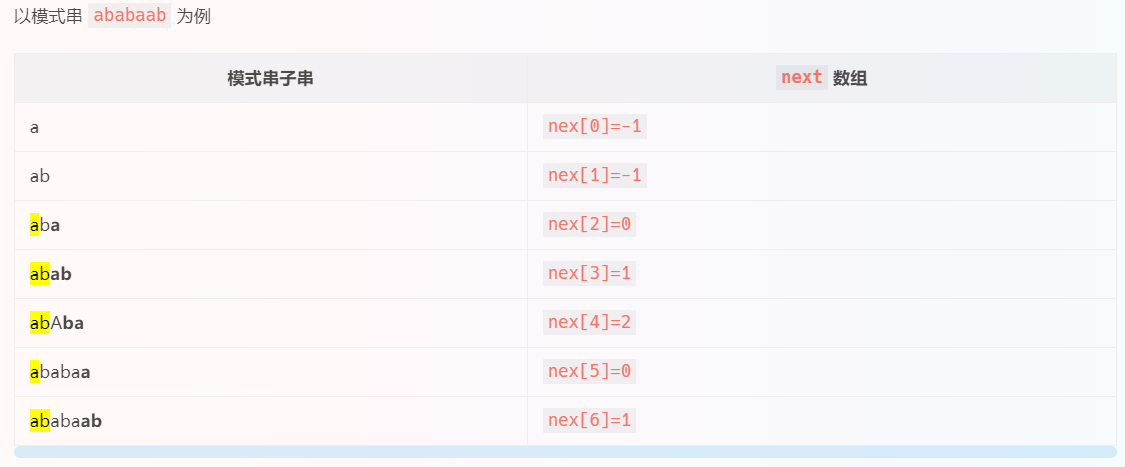


图60 next数组举例

1. 借助求出的关于模式串nex数组进行如下操作：利用已经部分匹配这个有效信息，保持i指针不回溯，通过修改j指针，让模式串尽量地移动到有效的位置

### 实验源码及备注

#### 操作说明

由于本人对于KMP算法理解不深，匹配情况仅做简单的统计，运行结果将每次匹配的位置与字符进行输出

#### 程序源码

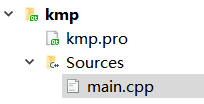


图61 题目二文件树

main.cpp文件

1. #include <bits/stdc++.h>
2. using namespace std;
3. typedef long long ll;
4. const int N = 1e6 + 10;
5. int nex[N];
6. void getnex(char str[], int len)
7. {
8. nex[0] = -1;
9. for (int i = 1, j = -1; i < len; i++) {
10. while (j != -1 && str[i] != str[j + 1]) j = nex[j];  *// 符合回退*
11. if (str[i] == str[j + 1]) j++;                       *// 符合前进*
12. nex[i] = j;                                          *// 记录当前位置的最长前缀的最后一个字符的位置*
13. }
14. }
15. int kmp(char tex[], char patt[])
16. {
17. int len1 = strlen(tex), len2 = strlen(patt), cnt = 0;
18. getnex(patt, len2);
19. for (int i = 0, j = -1; i < len1; i++) {
20. while (j != -1 && tex[i] != patt[j + 1]) {
21. printf("unmatch. s[%d]: %c, t[%d]: %c\n",
22. i,
23. tex[i],
24. j + 1,
25. patt[j + 1]);
26. j = nex[j];
27. }
28. if (patt[j + 1] == tex[i]) {
29. printf("match. s[%d]: %c, t[%d]: %c\n", i, tex[i], j + 1,
30. patt[j + 1]);
31. j++;
32. }
33. *// 匹配成功后次数+1，同时j回推继续匹配*
34. if (j == len2 - 1) {
35. printf("match succeed. match begin at pos %d of s\n", i);
36. cnt++, j = nex[j];
37. }
38. }
39. return cnt;
40. }
41. signed main()
42. {
43. *// ios::sync\_with\_stdio(false), cin.tie(0);*
44. char a[] = "aaabbdaabbde", b[] = "aabbd";
45. getnex(b, strlen(b));
46. cout << "Result: " << kmp(a, b) << endl;
47. *// system("pause");*
48. return 0;
49. }

### 实验运行截图及说明

#### 运行截图

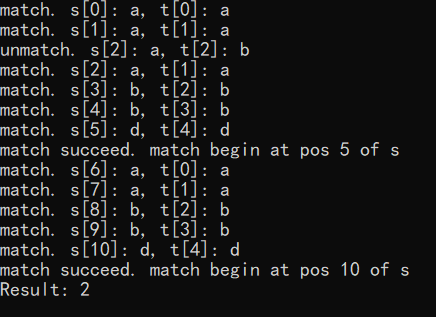


图62 题目二运行结果

#### 程序说明

**时间复杂度**：BF算法的时间复杂度为而算法成功的降到

#### 体会反思

1. 数组的定义：数组若是从-1开始一般记录的是匹配前缀末尾元素在模式串中的位置，若是从0开始一般记录的就是匹配的前缀的长度
2. 查找出现次数问题：如果题目允许模式串在匹配串中重复出现那么肯定是让指针回溯到上次的增加匹配的机会，反之只能不回溯从头重新进行匹配

## 题目三 矩阵的三元组

### 实验题目

实现稀疏矩阵的基本运算。假设n\*n的稀疏矩阵A采用三元组表示，设计一个程序，实现如下功能：

* 1. 生成如下两个稀疏矩阵的三元组a和b；



图63 题目三插图

* 1. 输出a转置矩阵的三元组
  2. 输出a+b的三元组

### 实验运用主要知识点

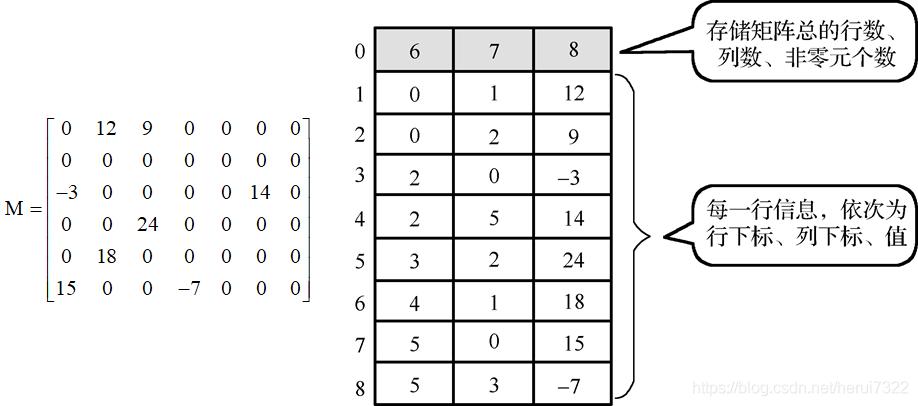


图64 三元组存储结构

**语法基础**：、指针与引用、线性代数、STL标准模板库

**概念基础**：按照压缩存储的概念，只存储稀疏矩阵的非零元素。因此，除了存储非零元的值之外，还必须同时记录下它所在的行和列的位置（i，j）。反之，一个三元组（i，j，aij） 唯一确定了矩阵A的一个非零元素。由此，稀疏矩阵可以用表示非零元的三元组及其行列数和非零元的个数唯一确定

**考察内容**：三元组表示法、矩阵的转置、矩阵内的计算

### 实验源码及备注

#### 操作说明

矩阵可以看做由多个有序的三元组组成，所有代码采用分层设为两个结构体组成关联关系，node类作为一个三元组，matrix统计矩阵的行数、列数以及非零元素的个数（即三元组的个数）

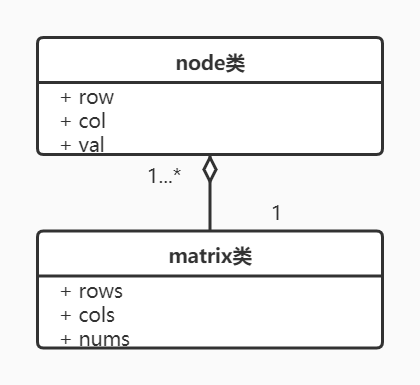


图65 题目三类图

#### 程序源码

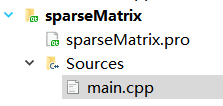


图66 题目三文件树

main.cpp文件

1. #include <iostream>
2. #include <algorithm>
3. #include <map>
4. using namespace std;
5. const int maxn = 25;
6. typedef pair<int, int> P;
7. int n = 4, m = 4;
8. int a[][4] = {
9. {1, 0, 3, -0},
10. {0, 1, -0, 0},
11. {0, -0, 1, 0},
12. {-0, 0, 1, 1}
13. }, b[][4] = {
14. {3, 0, 0, 0},
15. {0, 4, 0, 0},
16. {0, 0, 1, 0},
17. {0, 0, 0, 2}
18. };
19. struct node {
20. int row, col, val;
21. };
22. struct matrix {
23. int rows, cols, nums;
24. node vals[maxn];
25. };
26. bool cmp(node x, node y)
27. {
28. if (x.row < y.row) return x.row < y.row;
29. return x.col < y.col;
30. }
31. void Creat(matrix &t, int arr[][4])
32. {
33. t.rows = n, t.cols = m, t.nums = 0;
34. for (int i = 0; i < n; ++i)
35. for (int j = 0; j < m; ++j)
36. if (arr[i][j]) {
37. t.vals[t.nums].row = i;
38. t.vals[t.nums].col = j;
39. t.vals[t.nums].val = arr[i][j];
40. ++t.nums;
41. }
42. }
43. void Reverse(matrix &t)
44. {
45. for (int i = 0; i < t.nums; ++i)
46. swap(t.vals[i].col, t.vals[i].row);
47. }
48. void Display(matrix &t)
49. {
50. sort(t.vals, t.vals+t.nums, cmp);
51. for (int i = 0; i < t.nums; ++i)
52. printf("(%d, %d): %d\n", t.vals[i].row+1, t.vals[i].col+1, t.vals[i].val);
53. printf("\n");
54. }
55. void add(matrix &res, matrix &x, matrix &y)
56. {
57. map<P, int> mp;
58. for (int i = 0; i < x.nums; ++i)
59. mp[P(x.vals[i].col, x.vals[i].row)] += x.vals[i].val;
60. for (int i = 0; i < y.nums; ++i)
61. mp[P(y.vals[i].col, y.vals[i].row)] += y.vals[i].val;
62. res.cols = x.cols, res.rows = x.rows, res.nums = 0;
63. for (auto e:mp) {
64. res.vals[res.nums].val = e.second;
65. res.vals[res.nums].row = e.first.first;
66. res.vals[res.nums].col = e.first.second;
67. ++res.nums;
68. }
69. }
70. signed main()
71. {
72. matrix A, B, C;
73. Creat(A, a), Creat(B, b);
74. Reverse(A), Display(A);
75. Reverse(A), add(C, A, B), Display(C);   *// 再倒置回来*
76. return 0;
77. }

### 实验运行截图及说明

#### 运行截图

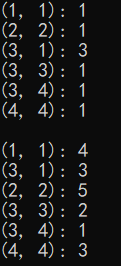


图67 题目三运行结果

#### 程序说明

**统计次数**：采用容器进行统计

**时间复杂度**：

#### 体会反思

矩阵转置：矩阵的两次转置等于本身，实验中可以选择复制矩阵但是需要重载矩阵的赋值构造函数，而利用上述定理可以减少代码量

## 题目四 广义表基本运算

### 实验题目

实现广义表的基本运算。编写一个程序，实现广义表的以下操作，完成以下功能：

1. 建立广义表g=（b,(b,a,( ),d),((a,b),c,(( )))）的链式存储结构。
2. 输出广义表g的长度

输出广义表g的深度。

### 实验运用主要知识点

**语法基础**：、指针与引用、STL标准模板库

**概念基础**：广义表，又称列表，也是一种线性存储结构。同数组类似，广义表中既可以存储不可再分的元素，也可以存储广义表，记作：。其中，代表广义表的名称，表示广义表存储的数据。广义表中每个既可以代表单个元素，也可以代表另一个广义表

通常，广义表中存储的单个元素称为 "原子"，而存储的广义表称为"子表"。  
例如创建一个广义表LS={1,{1,2,3}}，我们可以这样解释此广义表的构成：广义表LS存储了一个原子1和子表{1,2,3}  
以下是广义表存储数据的一些常用形式：

1. A=()：A表示一个广义表，只不过表是空的
2. B=(e)：广义表B中只有一个原子e
3. C=(a,(b,c,d))：广义表C中有两个元素，原子a和子表(b,c,d)。
4. D=(A,B,C)：广义表D中存有3个子表，分别是A、B和C。这种表示方式等同于D=((),(e),(b,c,d))
5. E=(a,E)：广义表E中有两个元素，原子a和它本身。这是一个递归广义表，等同于：E=(a,(a,(a,…)))

注意，A=()和A=(())是不一样的。前者是空表，而后者是包含一个子表的广义表，只不过这个子表是空表

**特性分析**：

1. 当广义表不是空表时，称第一个数据（原子或子表）为"表头"，剩下的数据构成的新广义表为"表尾"
2. 除非广义表为空表，否则广义表一定具有表头和表尾，且广义表的表尾一定是一个广义表

**考察内容**：广义表的链式存储、求广义表长度以及深度

### 实验源码及备注

#### 操作说明

由于空表的特殊性建表时需要特殊处理，为了简便本次实验输入空表时中间加入一个空格。采用递归传递引用指针进行操作，出现异常可能是程序存在溢出问题

#### 程序源码

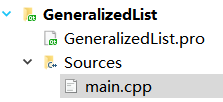


图68 题目四的文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. struct GlNode {
4. bool tag;   *// 标记是否为表的头结点*
5. union { *// 组合*
6. char data;
7. struct GlNode \*sublist;
8. } val;
9. struct GlNode \*next;    *// 一个元素结点位置*
10. };
11. GlNode \*creat(char \* &s)
12. {
13. GlNode \*head;
14. char ch = \*s;   *// 获取一个字符*
15. ++s;
16. if (ch != '\0') {
17. head = new GlNode();
18. if (ch == '(') {*// 创建一个新的结点*
19. head->tag = 1;*// 新结点为表头结点*
20. head->val.sublist = creat(s);   *// 递归构造*
21. } else if (ch == ')') head = nullptr;
22. else {
23. head->tag = 0;  *// 新结点为原子结点*
24. head->val.data = ch;
25. }
26. } else head = nullptr;  *// 串结束子表为空*
27. ch = \*s;
28. ++s;
29. if (head != nullptr) {
30. if (ch == ',') head->next = creat(s);
31. else head->next = nullptr;
32. }
33. return head;
34. }
35. void display(GlNode \*g)
36. {
37. if (g != nullptr) {
38. if (g->tag) {
39. cout<<"(";
40. if (g->val.sublist == nullptr) cout<<"";
41. else display(g->val.sublist);
42. } else cout<<g->val.data;
43. }
44. if (g->tag) cout<<")";
45. if (g->next != nullptr) {
46. cout<<",";
47. display(g->next);
48. }
49. }
50. int getLength(GlNode \*g)
51. {
52. int cnt = 0;
53. g = g->val.sublist;
54. while (g != nullptr) {
55. ++cnt;
56. g = g->next;
57. }
58. return cnt;
59. }
60. int getDepth(GlNode \*g)
61. {
62. int maxx = 0, depth;
63. if (!g->tag) return 0;
64. g = g->val.sublist;
65. if (g == nullptr) return 1;
66. while (g != nullptr) {
67. if (g->tag) {
68. depth = getDepth(g);
69. if (depth > maxx) maxx = depth;
70. }
71. g = g->next;
72. }
73. return maxx + 1;
74. }
75. signed main()
76. {
77. char s[] = "(b,(b,a,( ),d),((a,b),c,(( ))))";
78. char \*ps = (char \*)&s;
79. GlNode \*ghead = creat(ps);
80. display(ghead);
81. cout<<endl;
82. cout<<"Length: "<<getLength(ghead)<<endl;
83. cout<<"Depth: "<<getDepth(ghead)<<endl;
84. return 0;
85. }

### 实验运行截图及说明

#### 运行截图

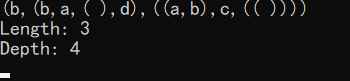


图69 题目四运行结果

#### 体会反思

表头与表尾：将一个表拆成一个表头和表尾时需要将表尾元素外部套上一组括号，因为是将表分为两组所以前面的表头可以看做是一个新的子表，而后面剩余的元素看做另一个子表

## 题目五 求矩阵马鞍点

### 实验题目

求一个矩阵中的马鞍点。如果矩阵A中存在这样的一个元素，满足条件：A[i][j]是第i行中值最小的元素，并且是第j列中值最大的元素，则称之位该矩阵的一个马鞍点。设计一个程序，求矩阵中所有的马鞍点

### 实验运用主要知识点

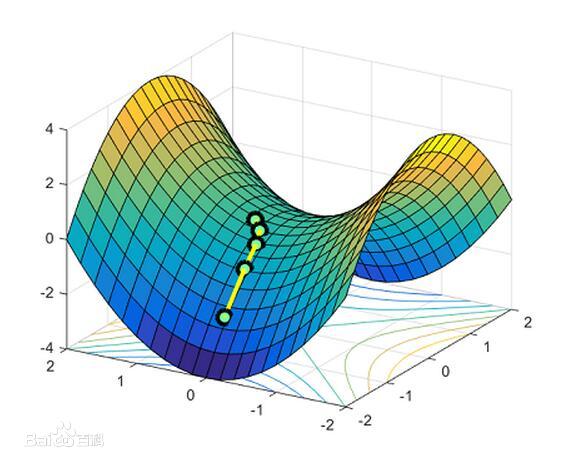


图70 鞍点示意图

**语法基础**：、线性代数

**概念基础**：鞍点在[微分方程](https://baike.baidu.com/item/%E5%BE%AE%E5%88%86%E6%96%B9%E7%A8%8B/4763)中，沿着某一方向是稳定的，另一条方向是不稳定的奇点，叫做鞍点。在[泛函](https://baike.baidu.com/item/%E6%B3%9B%E5%87%BD/9410786)中，既不是极大值点也不是极小值点的临界点，叫做鞍点。在[矩阵](https://baike.baidu.com/item/%E7%9F%A9%E9%98%B5/18069)中，一个数在所在行中是[最大值](https://baike.baidu.com/item/%E6%9C%80%E5%A4%A7%E5%80%BC/774514)，在所在列中是[最小值](https://baike.baidu.com/item/%E6%9C%80%E5%B0%8F%E5%80%BC/774537)，则被称为鞍点。在物理上要广泛一些，指在一个方向是极大值，另一个方向是极小值的点

### 实验源码及备注

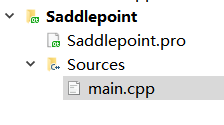


图71 题目五文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. const int maxn = 25;
4. int n, m;
5. int A[maxn][maxn];
6. int row[maxn], col[maxn];
7. int main()
8. {
9. cin>>n>>m;
10. for (int i = 0; i < n; ++i)
11. for (int j = 0; j < m; ++j)
12. cin>>A[i][j];
13. int temp;
14. for (int i = 0; i < n; ++i) {
15. temp = INT\_MAX;
16. for (int j = 0; j < m; ++j)
17. if (A[i][j] < temp) temp = A[i][j], row[i] = j;
18. }
19. for (int j = 0; j < m; ++j) {
20. temp = INT\_MIN;
21. for (int i = 0; i < n; ++i)
22. if (A[i][j] > temp)
23. temp = A[i][j], col[j] = i;
24. }
25. for (int i = 0; i < n; ++i) {
26. if (col[row[i]] == i)
27. printf("Result--(%d, %d): %d\n", i, row[i], A[i][row[i]]);
28. }
29. return 0;
30. }
31. */\**
32. 4 4
33. 9 7 6 8
34. 20 26 22 29
35. 90 44 25 30
36. 12 4 2 6
37. \*/

### 实验运行截图及说明

#### 运行截图

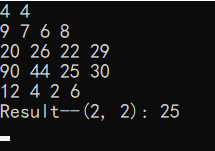


图71 题目五运行结果

#### 程序说明

**寻找马鞍点**：现在一行中找出最小的点，再检验其是否为列中的最大点

# 实验四 树实验

## 题目一 二叉树基本运算

### 实验内容

编写一个程序，实现二叉树的各种运算，并在此基础上设计一个主程序完成如下功能：

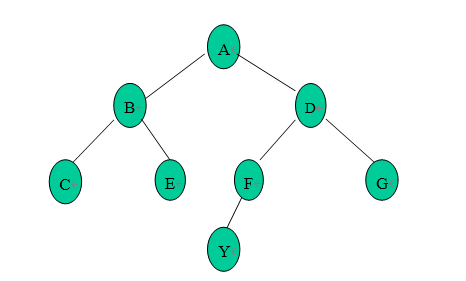


图72 题目一配图

1. 由如图一所示的二叉树创建其对应的二叉链表存储结构（提示：由二叉树的扩展前序序列构造二叉链表）
2. 输出二叉树的中序、先序、后序遍历序列的递归和非递归算法（中序必做，先序和后序选做）；
3. 输出二叉树B（树形结构或者广义表形式）；（选做）
4. 输出二叉树中指定结点值（假设所有结点值不同）的左右孩子结点；
5. 输出二叉树B的叶子结点个数；
6. 输出该二叉树的高度；
7. 输出二叉树中指定结点值（假设所有结点值不同）的结点所在的层次；（选做）

释放二叉树B。

### 实验运用主要知识点

**语法基础**：、指针与引用、类和对象、模板与泛型编程

**概念基础**：一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别成为左子树和右子树的、互不相交的二叉树组成

**特性分析**：

1. 二叉树结点的层次从1开始，则在二叉树的第i层最多有个结点
2. 深度为k的二叉树最少有k个结点，最多有个结点
3. 对任何一个二叉树，如果其叶结点有个，度为2的非叶结点有个则存在关系
4. 具有n个结点的完全二叉树的深度为
5. 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号为1，2，……，n，则有以下关系：

（1）若，则i无双亲

（2）若，则i的双亲为

（3）若,则i的左子女为，若，则i的有子女为

（4）若i为奇数且，则其左兄弟为，若i为偶数且i!=n则其右兄弟为

**考察内容**：二叉链表存储二叉树、二叉树的遍历、二叉树的广义表、查找二叉树结点、统计二叉树的结点个数、查询二叉树的高度、查找二叉树结点的层次、释放二叉树

### 实验源码及备注

#### 操作说明

二叉树可以看作是由多个二叉树结点按照一定关系组成的，所以本实验程序仍然选择分层封装形成组合关系

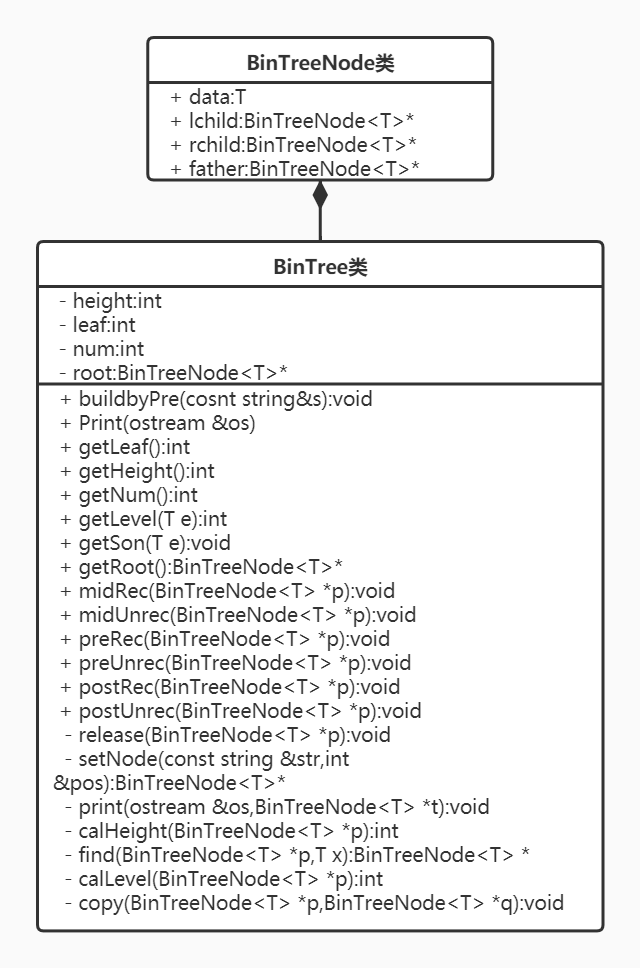


图73 二叉树类图

|  |  |
| --- | --- |
| **成员** | **作用** |
| data | 记录结点的权值 |
| lchild | 记录该结点的左子结点 |
| rchild | 记录该结点的右子结点 |

表13 BinTreeNode类成员

|  |  |
| --- | --- |
| **成员** | **作用** |
| height | 记录二叉树的深度 |
| leaf | 记录二叉树的叶结点个数 |
| num | 记录二叉树的结点个数 |
| root | 记录二叉树的根结点位置 |

表14 BinTree类成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名称** | BinTree | | | |
| **类描述** | 生成二叉树对象的模板类，包含二叉树基本运算用到的操作 | | | |
| buildbyPre | buildbyPre(const string&s):void | | | |
| **功能描述** | 根据前序遍历序列建树 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| s | string | 前序遍历的序列字符串 |
| Print | Print(ostream &os):void | | | |
| **功能描述** | 遍历二叉树 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| os | ostream | 输出流 |
| getLeaf | getLeaf():int | | | |
| **功能描述** | 获取叶子结点的个数 | | |
| **返回值** | 整数为叶子结点的个数 | | |
| getHeight | getHeight():int | | | |
| **功能描述** | 获取二叉树的高度 | | |
| **返回值** | 整数为二叉树的高度 | | |
| getNum | getNum():int | | | |
| **功能描述** | 获取二叉树结点个数 | | |
| **返回值** | 整数为二叉树结点个数 | | |
| getLevel | getLevel(T e):int | | | |
| **功能描述** | 获取二叉树中指定结点的层数 | | |
| **返回值** | 整数为指定结点的层数 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| e | T | 查找结点权值 |
| getSon | getSon(T e):void | | | |
| **功能描述** | 获取指定结点的子结点 | | |
| **返回值** | 无 | | |
| getRoot | getRoot():BinTreeNode<T>\* | | | |
| **功能描述** | 获取二叉树根结点 | | |
| **返回值** | 二叉树结点指针表示根结点 | | |
| midRec | midRec(BinTreeNode<T> \*p):void | | | |
| **功能描述** | 二叉树的中序递归遍历 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下遍历的结点 |
| midUnRec | midUnRec(BinTreeNode<T> \*p):void | | | |
| **功能描述** | 二叉树的中序非递归遍历 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下遍历的结点指针 |
| preRec | preRec(BinTeeeNode<T> \*p):void | | | |
| **功能描述** | 二叉树的前序递归遍历 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下遍历的结点指针 |
| preUnrec | preUnrec(BinTreeNode<T> \*p):void | | | |
| **功能描述** | 二叉树的前序非递归遍历 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下遍历的结点指针 |
| postRec | postRec(BinTreeNode<T> \*p):void | | | |
| **功能描述** | 二叉树的后序递归遍历 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下遍历的结点指针 |
| postUnrec | postUnrec(BinTreeNode<T> \*p):void | | | |
| **功能描述** | 二叉树的后序非递归遍历 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下遍历的结点指针 |
| release | release(BinTreeNode<T> \*p):void | | | |
| **功能描述** | 从指定结点位置开始向下释放的辅助函数 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 开始向下释放的结点指针 |
| setNode | setNode(const string &str,int &pos) | | | |
| **功能描述** | 根据当前串的指定位置获取结点权值并接力新的结点辅助函数 | | |
| **返回值** | BinTreeNode<T>\* | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| str | string& | 前序遍历字符串 |
| pos | int& | 获取权值对应字符串的位置 |
| print | print(ostream &os,BinTreeNode<T> \*t):void | | | |
| **功能描述** | 输出从二叉树当前结点向下递归广义表辅助函数 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| os | ostream& | 输入流 |
| t | BinTreeNode<T>\* | 向下递归的结点 |
| calHeight | calHeight(BinTreeNode<T> \*p):int | | | |
| **功能描述** | 计算从当前结点向下的深度辅助函数 | | |
| **返回值** | 整数为当前结点的向下深度 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 指向向下计算深度结点的指针 |
| find | find(BinTreeNode<T> \*p,T x):BinTreeNode<T>\* | | | |
| **功能描述** | 从当前结点向下查找权值为x的结点辅助函数 | | |
| **返回值** | BinTreeNode<T>\* | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 向下查找的结点指针 |
| x | T | 查找结点的权值 |
| calLevel | calLevel(BinTreeNode<T> \*p):int | | | |
| **功能描述** | 向上递归计算当前结点的层数辅助函数 | | |
| **返回值** | 整数为结点的层数 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 计算层数的目标结点 |
| copy | copy(BinTreeNode<T> \*p,BinTreeNode<T> \*q):void | | | |
| **功能描述** | 结点之间进行深复制的递归辅助函数 | | |
| **返回值** | 无 | | |
| **参数说明** | **参数名称** | **参数类型** | **参数说明** |
| p | BinTreeNode<T>\* | 进行复制的结点 |
| q | BinTreeNode<T>\* | 被复制的结点 |

表15 二叉树类接口文档

#### 程序源码

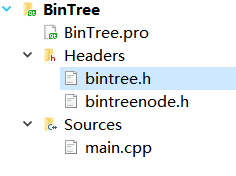


图74 二叉树文件树

bintreenode.h文件

1. #ifndef BINTREENODE\_H
2. #define BINTREENODE\_H
3. template<class T>
4. class BinTreeNode
5. {
6. public:
7. T data;
8. BinTreeNode<T> \*lchild, \*rchild, \*father;
9. BinTreeNode(T data, BinTreeNode<T> \*lchild = nullptr, BinTreeNode<T> \*rchild = nullptr,
10. BinTreeNode<T> \*father = nullptr);
11. BinTreeNode(const BinTreeNode<T> &N);
12. ~BinTreeNode();
13. };
14. template<class T>
15. BinTreeNode<T>::BinTreeNode(T data, BinTreeNode<T> \*lchild, BinTreeNode<T> \*rchild,
16. BinTreeNode<T> \*father) :
17. data(data),
18. lchild(lchild),
19. rchild(rchild),
20. father(father)
21. {
22. }
23. template<class T>
24. BinTreeNode<T>::BinTreeNode(const BinTreeNode<T> &N) :
25. data(N.data),
26. lchild(N.lchild),
27. rchild(N.rchild),
28. father(N.father)
29. {
30. }
31. template<class T>
32. BinTreeNode<T>::~BinTreeNode()
33. {
34. delete lchild;
35. delete rchild;
36. delete father;
37. }
38. #endif *// BINTREENODE\_H*

bintree.h文件

1. #ifndef BINTREE\_H
2. #define BINTREE\_H
3. #include <iostream>
4. #include <string>
5. #include <algorithm>
6. #include <stack>
7. #include "bintreenode.h"
8. template<class T>
9. class BinTree
10. {
11. public:
12. BinTree();
13. BinTree(const BinTree<T> &A);   *// 拷贝构造函数*
14. ~BinTree();
15. void buildbyPre(const std::string &s);   *// 前序遍历结果建树*
16. void Print(std::ostream &os);   *// 输出二叉树*
17. int getLeaf();*// 获取叶子数量*
18. int getHeight();  *// 获取高度*
19. int getNum();  *// 获取结点数量*
20. int getLevel(T e);    *// 获取结点层次*
21. void getSon(T e); *// 获取结点孩子*
22. BinTreeNode<T> \*getRoot();    *// 获取该树的根(后面线索化二叉树复用需要)*
23. *// 非递归用栈进行模拟*
24. void midRec(BinTreeNode<T> \*p);    *// 从结点递归中序*
25. void midUnrec(BinTreeNode<T> \*p);  *// 从结点非递归中序*
26. void preRec(BinTreeNode<T> \*p);    *// 从结点递归前序*
27. void preUnrec(BinTreeNode<T> \*p);  *// 从结点非递归前序*
28. void postRec(BinTreeNode<T> \*p);   *// 从结点递归后续*
29. void postUnrec(BinTreeNode<T> \*p); *// 从结点非递归后续*
30. private:
31. int height, leaf, num;  *// 高度,叶子数量,结点数量*
32. BinTreeNode<T> \*root;   *// 根结点*
33. void release(BinTreeNode<T> \*p); *// 从结点递归向下释放结点*
34. BinTreeNode<T> \*setNode(const std::string &str, int &pos); *// 从串中根据指定位置生成结点并继续递推*
35. void print(std::ostream &os, BinTreeNode<T> \*t);    *// 广义表形式输出*
36. int calHeight(BinTreeNode<T> \*p);   *// 从结点递归向下计算高度(定义上来说是深度)*
37. BinTreeNode<T> \*find(BinTreeNode<T> \*p, T x);   *// 辅助函数从结点递归向下找值为x的结点*
38. int calLevel(BinTreeNode<T> \*p);    *// 从结点向上遍历计算层次*
39. void copy(BinTreeNode<T> \*p, BinTreeNode<T> \*q);    *// 结点之间深复制*
40. };
41. template<class T>
42. BinTree<T>::BinTree() : height(0),
43. leaf(0),
44. num(0),
45. root(nullptr)
46. {
47. std::cout<<"BinTree builed succeed."<<std::endl;
48. }
49. template<class T>
50. BinTree<T>::BinTree(const BinTree<T> &A) : height(A.height),
51. leaf(A.leaf),
52. num(A.num)
53. {
54. root = new BinTreeNode<T>(A.root);
55. copy(this->root, A.root);
56. }
57. template<class T>
58. BinTree<T>::~BinTree()
59. {
60. std::cout<<"BinTree has been released!"<<std::endl;
61. release(root);
62. }
63. template<class T>
64. void BinTree<T>::release(BinTreeNode<T> \*p)
65. {
66. if (p == nullptr) return;
67. release(p->lchild);
68. release(p->rchild);
69. delete p;
70. }
71. template<class T>
72. void BinTree<T>::copy(BinTreeNode<T> \*p, BinTreeNode<T> \*q)
73. {
74. if (q->lchild != nullptr) {
75. p->lchild = new BinTreeNode<T>(q->lchild);
76. copy(p->lchild, q->lchild);
77. p->lchild->father = p;
78. }
79. if (q->rchild != nullptr) {
80. p->rchild = new BinTreeNode<T>(q->rchild);
81. copy(p->rchild, q->rchild);
82. p->rchild->father = p;
83. }
84. }
85. template<class T>
86. void BinTree<T>::buildbyPre(const std::string &str)
87. {
88. height = leaf = num = 0;
89. if (str[0] != '#') {
90. int pos = 0;
91. root = setNode(str, pos);
92. root->father = nullptr;
93. height = calHeight(root);
94. } else std::cerr<<"Empty BinTree"<<std::endl;
95. }
96. template<class T>
97. BinTreeNode<T> \*BinTree<T>::setNode(const std::string &str, int &pos)   *// 一定是要引用传值的*
98. {
99. BinTreeNode<T> \*temp = new BinTreeNode<T>(str[pos]);
100. ++pos;
101. if (str[pos] != '#') {
102. temp->lchild = setNode(str, pos);
103. temp->lchild->father = temp;
104. } else {
105. temp->lchild = nullptr;
106. ++pos;
107. }
108. if (str[pos] != '#') {
109. temp->rchild = setNode(str, pos);
110. temp->rchild->father = temp;
111. } else {
112. temp->rchild = nullptr;
113. ++pos;
114. }
115. if (temp->lchild == nullptr && temp->rchild == nullptr) ++leaf;
116. ++num;
117. return temp;
118. }
119. template<class T>
120. void BinTree<T>::Print(std::ostream &os)
121. {
122. print(os, this->root);
123. os<<std::endl;
124. }
125. template<class T>
126. int BinTree<T>::getLeaf()
127. {
128. return this->leaf;
129. }
130. template<class T>
131. int BinTree<T>::getHeight()
132. {
133. return this->height;
134. }
135. template<class T>
136. int BinTree<T>::getNum()
137. {
138. return num;
139. }
140. template<class T>
141. int BinTree<T>::getLevel(T e)
142. {
143. return calLevel(find(root, e));
144. }
145. template<class T>
146. void BinTree<T>::print(std::ostream &os, BinTreeNode<T> \*t)
147. {
148. os<<t->data;
149. if (t->lchild != nullptr || t->rchild != nullptr) {
150. os<<"(";
151. if (t->lchild != nullptr) print(os, t->lchild);
152. os<<",";
153. if (t->rchild != nullptr) print(os, t->rchild);
154. os<<")";
155. }
156. }
157. template<class T>
158. int BinTree<T>::calHeight(BinTreeNode<T> \*p)
159. {
160. if (p == nullptr) return 0;
161. else return std::max(calHeight(p->lchild), calHeight(p->rchild))+1;
162. }
163. template<class T>
164. BinTreeNode<T> \*BinTree<T>::find(BinTreeNode<T> \*p, T x)
165. {
166. if (p == nullptr) return nullptr;
167. if (p->data == x) return p;
168. BinTreeNode<T> \*temp;
169. if ((temp = find(p->lchild, x)) != nullptr) return temp;
170. if ((temp = find(p->rchild, x)) != nullptr) return temp;
171. return nullptr;
172. }
173. template<class T>
174. int BinTree<T>::calLevel(BinTreeNode<T> \*p)
175. {
176. if (p == nullptr) { *// 空指针没有结点*
177. std::cerr<<"element not exist."<<std::endl;
178. return 0;
179. }
180. int level = 1;  *// 向上遍历确定层数*
181. while (p != this->root) {
182. p = p->father;
183. ++level;
184. }
185. return level;
186. }
187. template<class T>
188. void BinTree<T>::getSon(T e)
189. {
190. BinTreeNode<T> \*temp = find(root, e);
191. if (temp != nullptr)
192. printf("lchild: %c rchild: %c\n", temp->lchild->data, temp->rchild->data);
193. }
194. template<class T>
195. void BinTree<T>::midRec(BinTreeNode<T> \*t)
196. {
197. if (t == nullptr) return;
198. midRec(t->lchild);
199. std::cout<<t->data;
200. midRec(t->rchild);
201. }
202. template<class T>
203. void BinTree<T>::midUnrec(BinTreeNode<T> \*t)
204. {
205. std::stack<BinTreeNode<T> \*> S;
206. std::string ans;
207. while (t != nullptr) {
208. S.push(t);
209. t = t->lchild;
210. }
211. while (!S.empty()) {
212. t = S.top();
213. S.pop();
214. ans += t->data;
215. if (t->rchild != nullptr) {
216. t = t->rchild;
217. while (t != nullptr) {
218. S.push(t);
219. t = t->lchild;
220. }
221. }
222. }
223. std::cout<<ans;
224. }
225. template<class T>
226. void BinTree<T>::preRec(BinTreeNode<T> \*p)
227. {
228. if (p == nullptr) return;
229. std::cout<<p->data;
230. preRec(p->lchild);
231. preRec(p->rchild);
232. }
233. template<class T>
234. void BinTree<T>::preUnrec(BinTreeNode<T> \*p)
235. {
236. std::stack<BinTreeNode<T> \*> S;
237. std::string ans;
238. S.push(p);
239. while (!S.empty()) {
240. p = S.top();
241. S.pop();
242. ans += p->data;
243. if (p->rchild != nullptr) S.push(p->rchild);
244. if (p->lchild != nullptr) S.push(p->lchild);
245. }
246. std::cout<<ans;
247. }
248. template<class T>
249. void BinTree<T>::postRec(BinTreeNode<T> \*p)
250. {
251. if (p == nullptr) return;
252. postRec(p->lchild);
253. postRec(p->rchild);
254. std::cout<<p->data;
255. }
256. template<class T>
257. void BinTree<T>::postUnrec(BinTreeNode<T> \*p)
258. {
259. std::stack<BinTreeNode<T> \*> S;
260. BinTreeNode<T> \*lastroot = nullptr;
261. std::string ans;
262. while (p != nullptr) {
263. S.push(p);
264. p = p->lchild;
265. }
266. while (!S.empty()) {
267. p = S.top();
268. if (p->rchild != nullptr && p->rchild != lastroot) {
269. p = p->rchild;
270. while (p != nullptr) {
271. S.push(p);
272. p = p->lchild;
273. }
274. } else {
275. ans += p->data;
276. lastroot = p;
277. S.pop();
278. }
279. }
280. std::cout<<ans;
281. }
282. template<class T>
283. BinTreeNode<T> \*BinTree<T>::getRoot()
284. {
285. return this->root;
286. }
287. #endif *// BINTREE\_H*

main.cpp文件

1. #include <iostream>
2. #include <string>
3. #include "BinTree.h"
4. using namespace std;
5. string str = "ABC##E##DFY###G##";
6. signed main()
7. {
8. cout<<"opt-1: Build binary tree"<<endl;
9. BinTree<char> B;
10. B.buildbyPre(str);
11. cout<<endl;
12. cout<<"opt-2: Traverse the binary tree"<<endl;
13. cout<<"Prerec traversal: ";
14. B.preRec(B.getRoot());
15. cout<<endl;
16. cout<<"PreUnrec traversal: ";
17. B.preUnrec(B.getRoot());
18. cout<<endl;
19. cout<<"Midrec traversal: ";
20. B.midRec(B.getRoot());
21. cout<<endl;
22. cout<<"MidUnrec traversal: ";
23. B.midUnrec(B.getRoot());
24. cout<<endl;
25. cout<<"Postrec traversal: ";
26. B.postRec(B.getRoot());
27. cout<<endl;
28. cout<<"PostUnrec traversal: ";
29. B.postUnrec(B.getRoot());
30. cout<<endl<<endl;
31. cout<<"opt-3: Output binary tree generalized table"<<endl;
32. B.Print(cout);
33. cout<<endl;
34. cout<<"opt-4: Output the children of node D"<<endl;
35. B.getSon('D');
36. cout<<endl;
37. cout<<"opt-5: Output the number of leaf nodes in the binary tree"<<endl;
38. cout<<B.getLeaf()<<endl;
39. cout<<endl;
40. cout<<"opt-6: Output the height of the binary tree"<<endl;
41. cout<<B.getHeight()<<endl;
42. cout<<endl;
43. cout<<"opt-7: The number of levels of output node D"<<endl;
44. cout<<B.getLevel('D')<<endl;
45. cout<<endl;
46. cout<<"opt-8: Release the binary tree";
47. cout<<endl;
48. return 0;
49. }

### 实验运行截图及说明

#### 运行截图

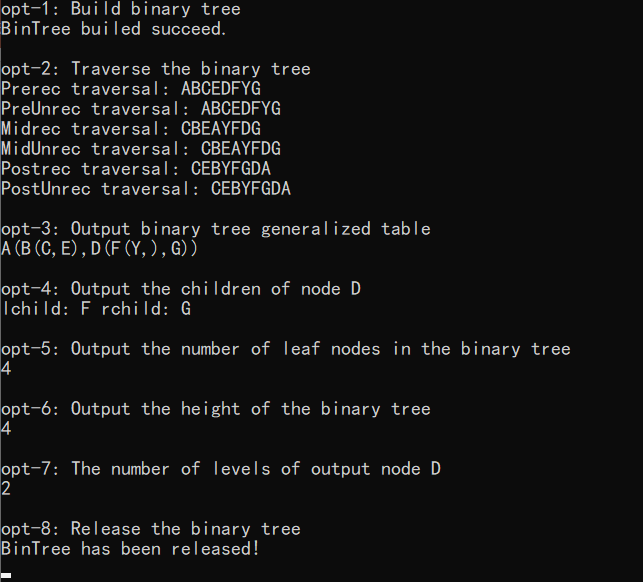


图74 二叉树实验运行结果

#### 程序说明

**二叉链表存储二叉树**：因为前序遍历就是递归遍历的结果所以生成二叉树就仿照前序遍历的路径进行结点的生成与连接，每次从字符串的指定位置获取一个权值

**二叉树的遍历**：三种递归遍历实现方法均相同只是操作的顺序存在差异，三种非递归遍历实现方法均借助栈模拟递归的顺序

**二叉树的广义表**：可以将每个结点看做根结点一个新的广义子表的生成，因此递归方法进行遍历，先访问左子树在用逗号间隔后访问右子树

**查找二叉树结点**：外部函数调用内部的查找辅助函数，从指定的结点向下递归查找权值为e的结点

**统计二叉树的结点个数**：维护一个变量在每次结点发生变化的时候统计

**查询二叉树的高度**：维护一个变量在每次结点发生变化的时候统计

**查找二叉树结点的层次**:先通过查找辅助函数确定是否存在指定的结点然后再调用计算层数的辅助函数

**释放二叉树**：析构函数调用辅助的递归结点释放函数

#### 体会反思

1. 函数分层调用——外部函数的操作实际上借助内部提供的各种基础函数完成，如此不仅逻辑上更加清晰而且相似的操作可以通过复用同一个函数减少代码的冗余
2. 引用传值——对于需要修改传入函数的参数需要采用引用传值保证局部函数修改
3. 二叉树的深度与高度的区别：深度是从上向下递归计算，高度是从下向上递归计算

## 题目二 前中序构造二叉树

### 实验内容

已知一棵树的前序遍历序列（ABECDFGHIJ）和中序遍历序列（EBCDAFHIGJ），试构造这棵二叉树

### 实验运用主要知识点

**语法基础**：、指针与引用、类和对象、模板与泛型编程、STL标准模板库

**概念基础**：

1. 前序遍历：对于当前结点，先输出该结点，然后输出他的左孩子，最后输出他的右孩子
2. 中序遍历：对于当前结点，先输出它的左孩子，然后输出该结点，最后输出它的右孩子
3. 后序遍历：对于当前结点，先输出它的左孩子，然后输出它的右孩子，最后输出该结点

**特性分析**：

1. 前序遍历序列第一个是根结点，中序遍历根结点两侧分为左子树与右子树上的结点，后续遍历最后一个是根结点
2. 根是相对的，对于整棵树而言只有一个根，但对于每棵子树而言，又有自己的根
3. 有了前序遍历，首先我们可以知道根结点的值，也就是数组中下标为0的位置，由此创建根结点。  
   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

**考察内容**：中序遍历与前序遍历构造二叉树

### 实验源码及备注

#### 操作说明

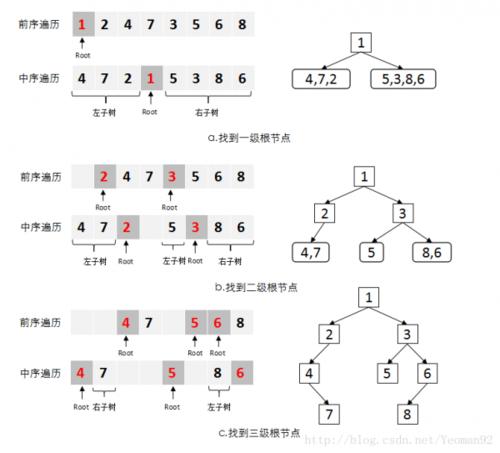


图75 题目二思路

#### 程序源码

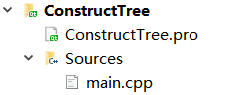


图76 题目二文件树

main.cpp文件

1. #ifndef BINTREE\_H
2. #define BINTREE\_H
3. #include <iostream>
4. #include <string>
5. #include <algorithm>
6. #include <stack>
7. #include "bintreenode.h"
8. template<class T>
9. class BinTree
10. {
11. public:
12. BinTree();
13. BinTree(const BinTree<T> &A);   *// 拷贝构造函数*
14. ~BinTree();
15. void buildbyPre(const std::string &s);   *// 前序遍历结果建树*
16. void Print(std::ostream &os);   *// 输出二叉树*
17. int getLeaf();*// 获取叶子数量*
18. int getHeight();  *// 获取高度*
19. int getNum();  *// 获取结点数量*
20. int getLevel(T e);    *// 获取结点层次*
21. void getSon(T e); *// 获取结点孩子*
22. BinTreeNode<T> \*getRoot();    *// 获取该树的根(后面线索化二叉树复用需要)*
23. *// 非递归用栈进行模拟*
24. void midRec(BinTreeNode<T> \*p);    *// 从结点递归中序*
25. void midUnrec(BinTreeNode<T> \*p);  *// 从结点非递归中序*
26. void preRec(BinTreeNode<T> \*p);    *// 从结点递归前序*
27. void preUnrec(BinTreeNode<T> \*p);  *// 从结点非递归前序*
28. void postRec(BinTreeNode<T> \*p);   *// 从结点递归后续*
29. void postUnrec(BinTreeNode<T> \*p); *// 从结点非递归后续*
30. private:
31. int height, leaf, num;  *// 高度,叶子数量,结点数量*
32. BinTreeNode<T> \*root;   *// 根结点*
33. void release(BinTreeNode<T> \*p); *// 从结点递归向下释放结点*
34. BinTreeNode<T> \*setNode(const std::string &str, int &pos); *// 从串中根据指定位置生成结点并继续递推*
35. void print(std::ostream &os, BinTreeNode<T> \*t);    *// 广义表形式输出*
36. int calHeight(BinTreeNode<T> \*p);   *// 从结点递归向下计算高度(定义上来说是深度)*
37. BinTreeNode<T> \*find(BinTreeNode<T> \*p, T x);   *// 辅助函数从结点递归向下找值为x的结点*
38. int calLevel(BinTreeNode<T> \*p);    *// 从结点向上遍历计算层次*
39. void copy(BinTreeNode<T> \*p, BinTreeNode<T> \*q);    *// 结点之间深复制*
40. };
41. template<class T>
42. BinTree<T>::BinTree() : height(0),
43. leaf(0),
44. num(0),
45. root(nullptr)
46. {
47. std::cout<<"BinTree builed succeed."<<std::endl;
48. }
49. template<class T>
50. BinTree<T>::BinTree(const BinTree<T> &A) : height(A.height),
51. leaf(A.leaf),
52. num(A.num)
53. {
54. root = new BinTreeNode<T>(A.root);
55. copy(this->root, A.root);
56. }
57. template<class T>
58. BinTree<T>::~BinTree()
59. {
60. std::cout<<"BinTree has been released!"<<std::endl;
61. release(root);
62. }
63. template<class T>
64. void BinTree<T>::release(BinTreeNode<T> \*p)
65. {
66. if (p == nullptr) return;
67. release(p->lchild);
68. release(p->rchild);
69. delete p;
70. }
71. template<class T>
72. void BinTree<T>::copy(BinTreeNode<T> \*p, BinTreeNode<T> \*q)
73. {
74. if (q->lchild != nullptr) {
75. p->lchild = new BinTreeNode<T>(q->lchild);
76. copy(p->lchild, q->lchild);
77. p->lchild->father = p;
78. }
79. if (q->rchild != nullptr) {
80. p->rchild = new BinTreeNode<T>(q->rchild);
81. copy(p->rchild, q->rchild);
82. p->rchild->father = p;
83. }
84. }
85. template<class T>
86. void BinTree<T>::buildbyPre(const std::string &str)
87. {
88. height = leaf = num = 0;
89. if (str[0] != '#') {
90. int pos = 0;
91. root = setNode(str, pos);
92. root->father = nullptr;
93. height = calHeight(root);
94. } else std::cerr<<"Empty BinTree"<<std::endl;
95. }
96. template<class T>
97. BinTreeNode<T> \*BinTree<T>::setNode(const std::string &str, int &pos)   *// 一定是要引用传值的*
98. {
99. BinTreeNode<T> \*temp = new BinTreeNode<T>(str[pos]);
100. ++pos;
101. if (str[pos] != '#') {
102. temp->lchild = setNode(str, pos);
103. temp->lchild->father = temp;
104. } else {
105. temp->lchild = nullptr;
106. ++pos;
107. }
108. if (str[pos] != '#') {
109. temp->rchild = setNode(str, pos);
110. temp->rchild->father = temp;
111. } else {
112. temp->rchild = nullptr;
113. ++pos;
114. }
115. if (temp->lchild == nullptr && temp->rchild == nullptr) ++leaf;
116. ++num;
117. return temp;
118. }
119. template<class T>
120. void BinTree<T>::Print(std::ostream &os)
121. {
122. print(os, this->root);
123. os<<std::endl;
124. }
125. template<class T>
126. int BinTree<T>::getLeaf()
127. {
128. return this->leaf;
129. }
130. template<class T>
131. int BinTree<T>::getHeight()
132. {
133. return this->height;
134. }
135. template<class T>
136. int BinTree<T>::getNum()
137. {
138. return num;
139. }
140. template<class T>
141. int BinTree<T>::getLevel(T e)
142. {
143. return calLevel(find(root, e));
144. }
145. template<class T>
146. void BinTree<T>::print(std::ostream &os, BinTreeNode<T> \*t)
147. {
148. os<<t->data;
149. if (t->lchild != nullptr || t->rchild != nullptr) {
150. os<<"(";
151. if (t->lchild != nullptr) print(os, t->lchild);
152. os<<",";
153. if (t->rchild != nullptr) print(os, t->rchild);
154. os<<")";
155. }
156. }
157. template<class T>
158. int BinTree<T>::calHeight(BinTreeNode<T> \*p)
159. {
160. if (p == nullptr) return 0;
161. else return std::max(calHeight(p->lchild), calHeight(p->rchild))+1;
162. }
163. template<class T>
164. BinTreeNode<T> \*BinTree<T>::find(BinTreeNode<T> \*p, T x)
165. {
166. if (p == nullptr) return nullptr;
167. if (p->data == x) return p;
168. BinTreeNode<T> \*temp;
169. if ((temp = find(p->lchild, x)) != nullptr) return temp;
170. if ((temp = find(p->rchild, x)) != nullptr) return temp;
171. return nullptr;
172. }
173. template<class T>
174. int BinTree<T>::calLevel(BinTreeNode<T> \*p)
175. {
176. if (p == nullptr) { *// 空指针没有结点*
177. std::cerr<<"element not exist."<<std::endl;
178. return 0;
179. }
180. int level = 1;  *// 向上遍历确定层数*
181. while (p != this->root) {
182. p = p->father;
183. ++level;
184. }
185. return level;
186. }
187. template<class T>
188. void BinTree<T>::getSon(T e)
189. {
190. BinTreeNode<T> \*temp = find(root, e);
191. if (temp != nullptr)
192. printf("lchild: %c rchild: %c\n", temp->lchild->data, temp->rchild->data);
193. }
194. template<class T>
195. void BinTree<T>::midRec(BinTreeNode<T> \*t)
196. {
197. if (t == nullptr) return;
198. midRec(t->lchild);
199. std::cout<<t->data;
200. midRec(t->rchild);
201. }
202. template<class T>
203. void BinTree<T>::midUnrec(BinTreeNode<T> \*t)
204. {
205. std::stack<BinTreeNode<T> \*> S;
206. std::string ans;
207. while (t != nullptr) {
208. S.push(t);
209. t = t->lchild;
210. }
211. while (!S.empty()) {
212. t = S.top();
213. S.pop();
214. ans += t->data;
215. if (t->rchild != nullptr) {
216. t = t->rchild;
217. while (t != nullptr) {
218. S.push(t);
219. t = t->lchild;
220. }
221. }
222. }
223. std::cout<<ans;
224. }
225. template<class T>
226. void BinTree<T>::preRec(BinTreeNode<T> \*p)
227. {
228. if (p == nullptr) return;
229. std::cout<<p->data;
230. preRec(p->lchild);
231. preRec(p->rchild);
232. }
233. template<class T>
234. void BinTree<T>::preUnrec(BinTreeNode<T> \*p)
235. {
236. std::stack<BinTreeNode<T> \*> S;
237. std::string ans;
238. S.push(p);
239. while (!S.empty()) {
240. p = S.top();
241. S.pop();
242. ans += p->data;
243. if (p->rchild != nullptr) S.push(p->rchild);
244. if (p->lchild != nullptr) S.push(p->lchild);
245. }
246. std::cout<<ans;
247. }
248. template<class T>
249. void BinTree<T>::postRec(BinTreeNode<T> \*p)
250. {
251. if (p == nullptr) return;
252. postRec(p->lchild);
253. postRec(p->rchild);
254. std::cout<<p->data;
255. }
256. template<class T>
257. void BinTree<T>::postUnrec(BinTreeNode<T> \*p)
258. {
259. std::stack<BinTreeNode<T> \*> S;
260. BinTreeNode<T> \*lastroot = nullptr;
261. std::string ans;
262. while (p != nullptr) {
263. S.push(p);
264. p = p->lchild;
265. }
266. while (!S.empty()) {
267. p = S.top();
268. if (p->rchild != nullptr && p->rchild != lastroot) {
269. p = p->rchild;
270. while (p != nullptr) {
271. S.push(p);
272. p = p->lchild;
273. }
274. } else {
275. ans += p->data;
276. lastroot = p;
277. S.pop();
278. }
279. }
280. std::cout<<ans;
281. }
282. template<class T>
283. BinTreeNode<T> \*BinTree<T>::getRoot()
284. {
285. return this->root;
286. }
287. #endif *// BINTREE\_H*

### 实验运行截图及说明

#### 运行截图



图77 题目二运行结果

#### 程序说明

**树的输出**：采用广义表形式输出

**构造方法**：每次从前序遍历中获取结点权值然后在中序遍历中查找将其分为两个子序列再次递归进行构造，前序遍历的划分就是中序遍历找到的位置，因为划分的位置就是起初的位置加上左子树结点的个数

**字符串截取**：借助函数进行分隔

## 题目三 二叉树查找结点祖先【待填】

### 实验内容

设二叉树采用二叉链表表示，指针root指向根结点，试编写一个非递归程序在二叉树中查找值为x的结点，并打印该结点所有祖先结点的算法。在此算法中，假设值为x的结点不多于一个

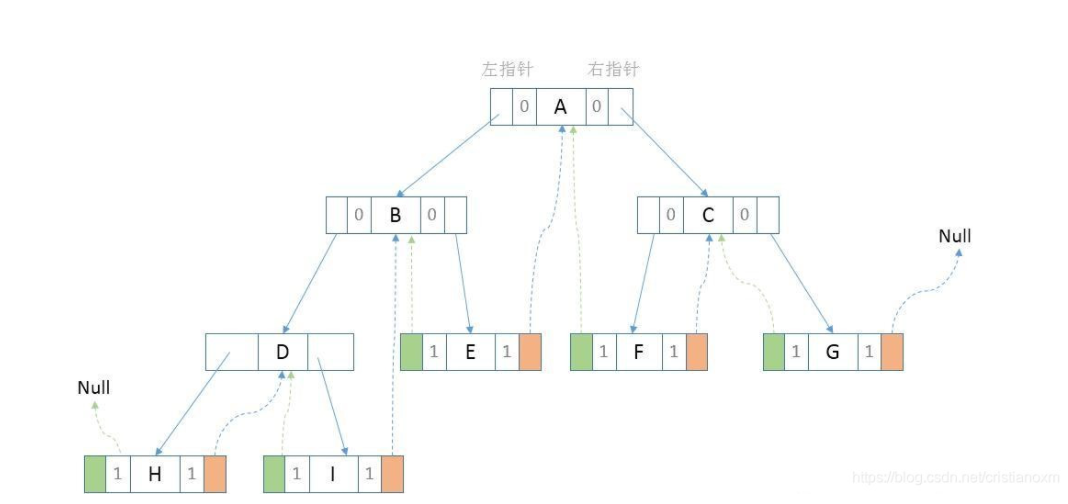
## 题目四 线索化二叉树基本操作

### 实验内容

线索化二叉树的操作：

1. 编写一个程序实现二叉树的中序线索化，采用递归和非递归方式输出中序线索二叉树的中序序列，并以题目一实验所示的二叉树B对程序进行验证；
2. 求出该线索化二叉树中给定结点的直接前驱和直接后继结点。
3. 试编写一个算法把一个新结点l作为结点s的左子女插入到一棵线索化二叉树中，s原来的左子女变成l的左子女

### 实验运用主要知识点



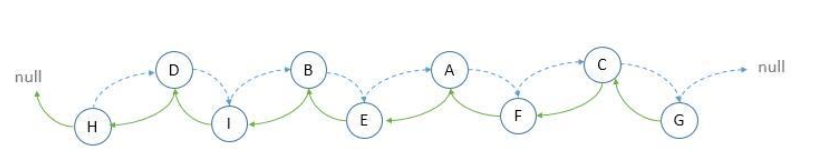


图78 线索化二叉树存储结构

**语法基础**：、指针与引用、类和对象、模板与泛型编程

**概念基础**：遍历二叉树是以一定的规则将二叉树中的结点排成一个线性序列，从而得到几种遍历序列，使得序列中的每个结点（第一个结点和最后一个结点除外）都有一个直接前驱和直接后继。传统的二叉链表存储仅能体现父子关系不能直接得到结点在遍历中的前驱和后继。前面提到含n个结点的二叉树中，有n+1个空指针。由此可以利用这些空指针来存放指向其前驱或后继的指针，加快查找结点前驱和后继的速度。规定若无左子树令lchild指向其前驱结点，若无右子树另rchild指向其后继结点。还需要增加两个标识域标志指针域是指向左（右）孩子还是指向前驱（后继）

**特点分析**：

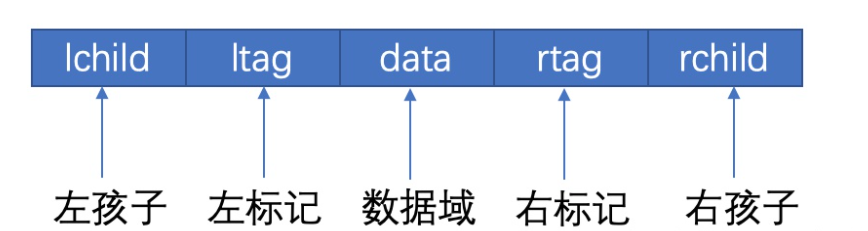


图79 线索二叉树结点

1. 线索二叉树的标志域含义如下：对于来说，0代表域指示结点的左孩子，1代表域指示结点的前驱；对于来说，0代表域指示结点的右孩子，1代表域指示结点的后继
2. 线索化二叉树是将表中的空指针改为指向前驱和后继的线索。而前驱和后继的信息只有在遍历时才能得到，因此线索化的实质就是遍历一次二叉树
3. 以中序线索化二叉树为例。附设指针指向刚刚访问过的结点，指针指向正在访问的结点，即指向的前驱。在中序遍历的过程中，检查的左指针是否为空，若为空就将它指向；检查的右指针是否为空，若为空就将它指向p

**考察内容**：构建线索化二叉树、查询二叉树结点的前去后继、左子树插入新结点

### 实验源码及备注

#### 操作说明

线索化二叉树可以看做是在二叉树基础上进一步做出的修改，因为本实验代码先复用二叉树代码生成二叉树，再利用二叉树生成线索二叉树。线索化二叉树基本操作与二叉树类似，只是在结点变化部分做出了小的修改

#### 程序源码

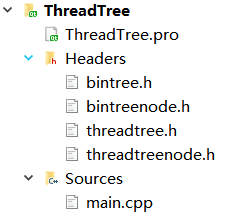


图80 线索化二叉树文件树

bintreenode.h&bintree.h文件略

threadtreenode.h文件

1. #ifndef THREADTREENODE\_H
2. #define THREADTREENODE\_H
3. template<class T>
4. class ThreadTreeNode
5. {
6. public:
7. T data;
8. ThreadTreeNode<T> \*lchild, \*rchild, \*father;
9. bool ltag, rtag;
10. ThreadTreeNode(T data, ThreadTreeNode<T> \*lchild = nullptr, ThreadTreeNode<T> \*rchild = nullptr,
11. ThreadTreeNode<T> \*father = nullptr, bool ltag = 0, bool rtag = 0);
12. ThreadTreeNode(const ThreadTreeNode<T> &N);
13. ~ThreadTreeNode();
14. };
15. template<class T>
16. ThreadTreeNode<T>::ThreadTreeNode(const ThreadTreeNode<T> &N) :
17. lchild(N.lchild),
18. rchild(N.rchild),
19. father(N.father),
20. ltag(N.ltag),
21. rtag(N.rtag)
22. {
23. }
24. template<class T>
25. ThreadTreeNode<T>::~ThreadTreeNode()
26. {
27. delete(lchild);
28. delete(rchild);
29. delete(father);
30. }
31. template<class T>
32. ThreadTreeNode<T>::ThreadTreeNode(T data, ThreadTreeNode<T> \*lchild, ThreadTreeNode<T> \*rchild,
33. ThreadTreeNode<T> \*father, bool ltag, bool rtag) :
34. data(data),
35. lchild(lchild),
36. rchild(rchild),
37. father(father),
38. ltag(ltag),
39. rtag(rtag)
40. {
41. }
42. #endif *// THREADTREENODE\_H*

threadtree.h文件

1. #ifndef THREADTREE\_H
2. #define THREADTREE\_H
3. #include "threadtreenode.h"
4. #include "bintree.h"
5. template<class T>
6. class ThreadTree
7. {
8. public:
9. ThreadTree();
10. ThreadTree(const BinTree<T> &A);  *// 注意必须将函数声明为const因为内部调用了常函数getRoot()*
11. ~ThreadTree();
12. void findPreAndNext(T x);   *// 找到值为x结点的前驱和后继结点*
13. void InsertLchild(T x, T y);    *// 将值为x结点插入到值为y当前结点的左子树(前提建立在了结点的权值唯一)*
14. *// 非递归用栈进行模拟*
15. void midRec(ThreadTreeNode<T> \*p);    *// 从结点递归中序*
16. void midUnrec(ThreadTreeNode<T> \*p);  *// 从结点非递归中序*
17. void preRec(ThreadTreeNode<T> \*p);    *// 从结点递归前序*
18. void preUnrec(ThreadTreeNode<T> \*p);  *// 从结点非递归前序*
19. void postRec(ThreadTreeNode<T> \*p);   *// 从结点递归后续*
20. void postUnrec(ThreadTreeNode<T> \*p); *// 从结点非递归后续*
21. ThreadTreeNode<T> \*getRoot();
22. private:
23. ThreadTreeNode<T> \*root;    *// 根结点*
24. void release(ThreadTreeNode<T> \*p);
25. void threadInit(ThreadTreeNode<T> \*now, ThreadTreeNode<T> \* &pre);
26. ThreadTreeNode<T> \*find(ThreadTreeNode<T> \*p, T x);
27. void copy(ThreadTreeNode<T> \*p, const BinTreeNode<T> \*q);
28. ThreadTreeNode<T> \*pre(ThreadTreeNode<T> \*p);
29. ThreadTreeNode<T> \*post(ThreadTreeNode<T> \*p);
30. ThreadTreeNode<T> \*last(ThreadTreeNode<T> \*p);
31. ThreadTreeNode<T> \*first(ThreadTreeNode<T> \*p);
32. };
33. template<class T>
34. ThreadTree<T>::ThreadTree() : root(nullptr)
35. {
36. std::cout<<"ThreadTree builed succeed."<<std::endl;
37. }
38. template<class T>
39. ThreadTree<T>::ThreadTree(const BinTree<T> &A)
40. {
41. if (A.getRoot() == nullptr) {
42. std::cerr<<"ThreadTree build failed."<<std::endl;
43. return;
44. }
45. this->root = new ThreadTreeNode<T>(A.getRoot()->data);
46. copy(root, A.getRoot());
47. ThreadTreeNode<T> \*prep = nullptr;    *// 空结点*
48. threadInit(this->root, prep);
49. prep->rtag = 1;
50. std::cout<<"ThreadTree builed succeed."<<std::endl;
51. }
52. template<class T>
53. ThreadTree<T>::~ThreadTree()
54. {
55. release(root);
56. std::cout<<"ThreadTree has been released!"<<std::endl;
57. }
58. template<class T>
59. void ThreadTree<T>::findPreAndNext(T x)
60. {
61. ThreadTreeNode<T> \*temp = find(root, x);
62. if (temp != nullptr) {
63. if (pre(temp) != nullptr) std::cout<<"pre data: "<<pre(temp)->data<<std::endl;
64. else std::cout<<"pre data: #"<<std::endl;
65. if (post(temp) != nullptr) std::cout<<"post data: "<<post(temp)->data<<std::endl;
66. else std::cout<<"post data: #"<<std::endl;
67. }
68. }
69. template<class T>
70. void ThreadTree<T>::InsertLchild(T x, T y)
71. {
72. ThreadTreeNode<T> \*temp = find(this->root, y);
73. if (temp == nullptr) {
74. std::cout<<"dont find the prenode you want to insert."<<std::endl;
75. return;
76. }
77. ThreadTreeNode<T> \*prep = pre(temp);
78. ThreadTreeNode<T> \*newnode = new ThreadTreeNode<T>(x, temp->lchild, temp, temp);*// 该结点的后继就是temp*
79. newnode->rtag = 1, newnode->ltag = temp->ltag;
80. if (temp->ltag == 0) temp->lchild->father = newnode; *// 设置结点原左子女的信息*
81. if (prep != nullptr && prep->rtag) prep->rchild = newnode;
82. temp->lchild = newnode;  *// 设置当前结点的左侧信息*
83. temp->ltag = 0;
84. }
85. template<class T>
86. void ThreadTree<T>::midRec(ThreadTreeNode<T> \*p)
87. {
88. if (p == nullptr) return;
89. if (p->ltag != 1) midRec(p->lchild);
90. std::cout<<p->data;
91. if (p->rtag != 1) midRec(p->rchild);
92. }
93. template<class T>
94. void ThreadTree<T>::midUnrec(ThreadTreeNode<T> \*p)
95. {
96. p = first(p);
97. while (p) {
98. std::cout<<p->data;
99. p = post(p);
100. }
101. }
102. template<class T>
103. void ThreadTree<T>::preRec(ThreadTreeNode<T> \*p)
104. {
105. if (p == nullptr) return;
106. std::cout<<p->data;
107. if (p->ltag != 1) preRec(p->lchild);
108. if (p->rtag != 1) preRec(p->rchild);
109. }
110. template<class T>
111. void ThreadTree<T>::preUnrec(ThreadTreeNode<T> \*p)
112. {
113. while (p != nullptr) {
114. std::cout<<p->data;
115. if (p->ltag != 1) p = p->lchild;
116. else if (p->ltag != 1) p = p->rchild;
117. else {
118. while (p && p->rtag) p = p->rchild;
119. if (p != nullptr) p = p->rchild;
120. }
121. }
122. }
123. template<class T>
124. void ThreadTree<T>::postUnrec(ThreadTreeNode<T> \*p)
125. {
126. while (p->rtag == 0 || p->ltag == 0) {
127. if (p->ltag == 0) p = p->lchild;
128. else p = p->rchild;
129. }
130. std::cout<<p->data;
131. while (p->father != nullptr) {
132. if (p == p->father->rchild || p->father->rtag == 1) p = p->father;
133. else {
134. p = p->father->rchild;
135. while (p->rtag == 0 || p->ltag == 0) {
136. if (p->ltag == 0) p = p->lchild;
137. else p = p->rchild;
138. }
139. }
140. std::cout<<p->data;
141. }
142. }
143. template<class T>
144. void ThreadTree<T>::postRec(ThreadTreeNode<T> \*p)
145. {
146. if (p == nullptr) return;
147. if (p->ltag != 1) preRec(p->lchild);
148. if (p->rtag != 1) preRec(p->rchild);
149. std::cout<<p->data;
150. }
151. template<class T>
152. ThreadTreeNode<T> \*ThreadTree<T>::getRoot()
153. {
154. return this->root;
155. }
156. template<class T>
157. void ThreadTree<T>::release(ThreadTreeNode<T> \*p)
158. {
159. if (p->lchild != nullptr && p->ltag == 0) release(p->lchild);
160. if (p->rchild != nullptr && p->rtag == 0) release(p->rchild);
161. delete(p);
162. }
163. template<class T>
164. void ThreadTree<T>::threadInit(ThreadTreeNode<T> \*now, ThreadTreeNode<T> \* &pre)
165. {
166. if (now == nullptr) return;
167. if (now->lchild != nullptr) threadInit(now->lchild, pre);
168. else now->lchild = pre, now->ltag = 1;
169. if (pre != nullptr && pre->rchild == nullptr) pre->rchild = now, pre->rtag = 1;
170. pre = now;
171. threadInit(now->rchild, pre);
172. }
173. template<class T>
174. ThreadTreeNode<T> \*ThreadTree<T>::find(ThreadTreeNode<T> \*p, T x)
175. {
176. if (p == nullptr) return nullptr;
177. if (x == p->data) return p;
178. ThreadTreeNode<T> \*temp;
179. if (p->ltag == 0 && (temp = find(p->lchild, x)) != nullptr) return temp;
180. if (p->rtag == 0 && (temp = find(p->rchild, x)) != nullptr) return temp;
181. return nullptr;
182. }
183. template<class T>
184. void ThreadTree<T>::copy(ThreadTreeNode<T> \*p, const BinTreeNode<T> \*q)
185. {
186. if (q->lchild != nullptr) {
187. p->lchild = new ThreadTreeNode<T>(q->lchild->data);
188. copy(p->lchild, q->lchild);
189. p->lchild->father = p;
190. }
191. if (q->rchild != nullptr) {
192. p->rchild = new ThreadTreeNode<T>(q->rchild->data);
193. copy(p->rchild, q->rchild);
194. p->rchild->father = p;
195. }
196. }
197. template<class T>
198. ThreadTreeNode<T> \*ThreadTree<T>::pre(ThreadTreeNode<T> \*p)
199. {
200. if (p->ltag == 1) return p->lchild;
201. else return last(p->lchild); *// 说明需要从头重新遍历找前驱结点*
202. }
203. template<class T>
204. ThreadTreeNode<T> \*ThreadTree<T>::post(ThreadTreeNode<T> \*p)
205. {
206. if (p->rtag == 1) return p->rchild;
207. else return first(p->rchild);   *// 说明需要从头重新遍历找后继结点*
208. }
209. template<class T>
210. ThreadTreeNode<T> \*ThreadTree<T>::last(ThreadTreeNode<T> \*p)
211. {
212. while (p->rtag != 1) p = p->rchild;
213. return p;
214. }
215. template<class T>
216. ThreadTreeNode<T> \*ThreadTree<T>::first(ThreadTreeNode<T> \*p)
217. {
218. while (p->ltag != 1) p = p->lchild;
219. return p;
220. }
221. #endif *// THREADTREE\_H*

main.cpp文件

1. #include <iostream>
2. #include "threadtree.h"
3. using namespace std;
4. BinTree<char> B;
5. string str = "ABC##E##DFY###G##";
6. signed main()
7. {
8. B.buildbyPre(str);
9. cout<<"opt-1: Build thread tree"<<endl;
10. ThreadTree<char> T(B);
11. cout<<"opt-2: Traverse the thread tree"<<endl;
12. cout<<"Prerec traversal: ";
13. T.preRec(T.getRoot());
14. cout<<endl;
15. cout<<"PreUnrec traversal: ";
16. T.preUnrec(T.getRoot());
17. cout<<endl;
18. cout<<"Midrec traversal: ";
19. T.midRec(T.getRoot());
20. cout<<endl;
21. cout<<"MidUnrec traversal: ";
22. T.midUnrec(T.getRoot());
23. cout<<endl;
24. cout<<"Postrec traversal: ";
25. T.postRec(T.getRoot());
26. cout<<endl;
27. cout<<"PostUnrec traversal: ";
28. T.postUnrec(T.getRoot());
29. cout<<endl<<endl;
30. cout<<"opt-3: Find the pre and post node of B"<<endl;
31. T.findPreAndNext('B');
32. cout<<endl;
33. cout<<"opt-4: Insert character I at pos of B lchild"<<endl;
34. cout<<"Insert before the pre and post node of B"<<endl;
35. T.findPreAndNext('B');
36. T.InsertLchild('I', 'B');
37. cout<<"Insert after the pre and post node of B"<<endl;
38. T.findPreAndNext('B');
39. cout<<endl;
40. cout<<"opt-5: Release the thread tree";
41. cout<<endl;
42. return 0;
43. }

### 实验运行截图及说明

#### 运行截图

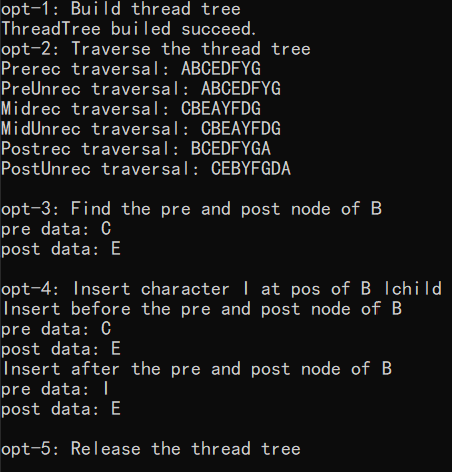


图81 线索化二叉树运行结果

#### 程序说明

**线索化操作**：建立前驱结点的条件是当前结点的左指针为空，建立后继结点的条件是当前结点的有指针为空并且前驱结点不为空

**插入结点操作**：先判断要插入的父结点是否存在，然后对于父结点右指针设置后继，新结点左指针指向父结点的左子结点，右指针指向父结点作为后继。进而再判断是否还需要设置父结点的父结点后继

#### 体会反思

代码能力有待提高——仅仅在二叉树的基础上做出了一些修改，起初并没有想到复用二叉树的操作。后来调试也花了大概仅4个小时，深深感受到自己编程能力的不足

## 题目五 哈弗曼树编码

### 实验内容

编写一个程序，构造一棵哈夫曼树，输出对应的哈夫曼编码和平均查找长度，并对表1所示的数据进行验证

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 单词 | The | of | a | to | and | In | that | he | is | at | on | for | his | are | be |
| 频率 | 1192 | 677 | 541 | 518 | 462 | 450 | 242 | 195 | 190 | 181 | 174 | 157 | 138 | 124 | 123 |

表16 哈弗曼树编码单词统计

### 实验运用主要知识点

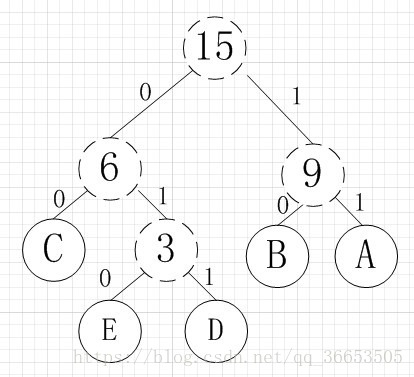


图82 利用哈夫曼树进行编码

**语法基础**：、指针与引用、类和对象、模板与泛型编程、STL标准模板库

**概念基础**：

1. 哈弗曼树：在许多应用中树中结点常常被赋予一个表示某种意义的数值称为该结点的权。从树的根到任意结点的路径长度（经过的边数）与该结点上权值的乘积成为该结点的带权路径长度。树中所有叶结点的带权路径长度之和成为该树的带权路径长度，记为WPL。在含有n个带权叶结点的二叉树中，其中带权路径长度最小的树称为哈弗曼树，也称为最优二叉树。
2. 哈夫曼编码：[哈夫曼](https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC)编码又称霍夫曼编码，是一种编码方式，哈夫曼编码是可变[字长](https://baike.baidu.com/item/%E5%AD%97%E9%95%BF/97660)编码(VLC)的一种。Huffman于1952年提出一种编码方法，该方法完全依据[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6/4768913)出现概率来构造异字头的平均长度最短的码字，有时称之为最佳编码，一般就叫做Huffman编码（有时也称为霍夫曼编码）

**特点分析**：

1. 哈夫曼所构造的码字不是唯一的，对于同一个信息源，无论上述的前后顺序如何排列，它的平均码长是不会改变的，所以他的优点是编码效率唯一性
2. 哈夫曼编码的稳定性比较差。如果改变其中一位数据就会产生改变

**考察内容**：哈弗曼树的构造、哈弗曼树编码

### 实验源码及备注

#### 程序源码

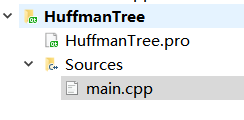


图83 哈弗曼树编码文件树

main.cpp文件

1. #include <iostream>
2. #include <queue>
3. #include <vector>
4. #include <algorithm>
5. using namespace std;
6. struct node {
7. string data; *// date非空的就是叶结点*
8. int val;
9. string code;
10. node \*lchild, \*rchild;
11. node() = default;
12. node(const node &A) :
13. data(A.data),
14. val(A.val),
15. code(A.code),
16. lchild(A.lchild),
17. rchild(A.rchild)
18. {
19. }
20. node(string data, int val, string code = "", node \*lchild = nullptr, node \*rchild = nullptr) :
21. data(data),
22. val(val),
23. code(code),
24. lchild(lchild),
25. rchild(rchild)
26. {
27. }
28. bool operator <(const node y) const
29. {
30. return this->val > y.val;   *// 设置val大的优先级高*
31. }
32. };
33. vector<node> vec;
34. void encode(node \*now, string str)
35. {
36. now->code = str;
37. if (now->data != "") vec.push\_back(node(\*now));
38. if (now->lchild != nullptr) encode(now->lchild, now->code+'0');
39. if (now->rchild != nullptr) encode(now->rchild, now->code+'1');
40. }
41. bool cmp(node x, node y)
42. {
43. if (x.code.size() != y.code.size()) return x.code.size() < y.code.size();
44. for (int i = 0; i < (int)x.code.size(); ++i) {
45. if (x.code[i] != y.code[i])
46. return x.code[i] < y.code[i];
47. }
48. return 0;
49. }
50. void release(node \*rt)
51. {
52. if (rt->lchild != nullptr || rt->rchild != nullptr) {
53. if (rt->lchild != nullptr) release(rt->lchild);
54. if (rt->rchild != nullptr) release(rt->rchild);
55. } else delete rt;
56. }
57. signed main()
58. {
59. priority\_queue<node> que;
60. int n;
61. cin>>n;
62. node \*temp;
63. for (int i = 0; i < n; ++i) {
64. string d;
65. int f;
66. cin>>d>>f;
67. que.push(node(d, f));
68. }
69. while (que.size() > 1) {    *// 建立好二叉树*
70. temp = new node;
71. node \*p = new node(que.top());
72. temp->lchild = p;
73. que.pop();
74. p = new node(que.top());
75. temp->rchild = p;
76. que.pop();
77. temp->val = temp->lchild->val+temp->rchild->val;
78. temp->code = temp->data = "";
79. que.push(\*temp);
80. }
81. temp = new node(que.top());
82. encode(temp, "");
83. sort(vec.begin(), vec.end(), cmp);
84. int avg = 0;
85. cout<<endl<<endl<<"Huffman Code as follows: "<<endl;
86. for (auto e:vec) {
87. avg += e.code.size();
88. cout<<e.data<<'\t'<<e.code<<endl;
89. }
90. cout<<"ASL: "<<avg\*1.0/vec.size()<<endl;    *// sum\_{查找每个元素概率\*比较次数}*
91. release(temp);
92. return 0;
93. }
94. */\*15*
95. The 1192
96. of 677
97. a 541
98. to 518
99. and 462
100. In 450
101. that 242
102. he 195
103. is 190
104. at 181
105. on 174
106. for 157
107. his 138
108. are 124
109. be 123
110. \*/

### 实验运行截图及说明

#### 运行截图

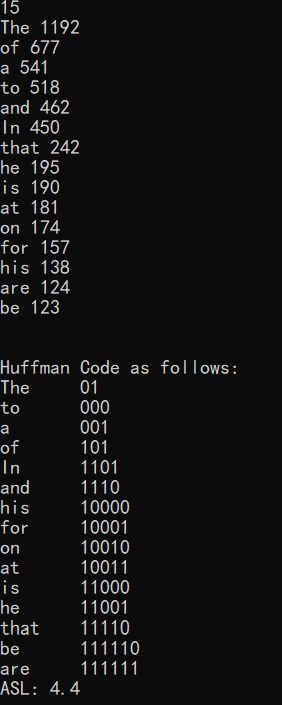


图84 哈夫曼树编码运行结果

#### 程序说明

**哈夫曼树编码**：每次选择连个最小的权值结点生成新的权值结点并重新放入集合，递归建树直至集合中只剩下一个结点。此时规定左子树路径为0，右子树路径为1，每个叶子结点上单词的编码就是从根结点到当前位置路径上所有字符的加和

#### 体会反思

1. 优先队列重载——本次实验了解了STL中的优先队列是利用向量容器实现的，想要自定义优先级的方法就是在结构体中将小于号重载
2. ASL——平均查找长度考虑所有单词被查询的可能相同的情况下查询路径的平均数

# 实验五 图实验

## 题目一 图基本运算

### 实验内容

编写一个程序，实现图的相关运算，并在此基础上设计一个主程序完成如下功能：

1. 建立图（二）所示的有向图G的邻接矩阵，并输出之。
2. 由有向图G的邻接矩阵产生邻接表，并输出之。
3. 再由（2）的邻接表产生对应的邻接矩阵，并输出之。

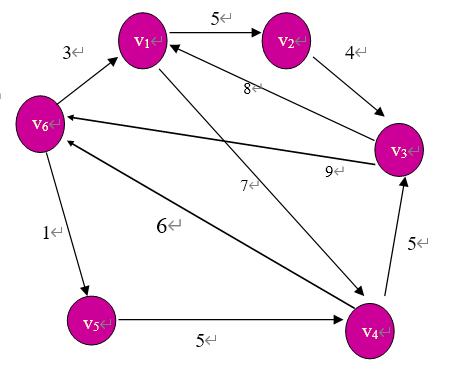


图85 题目一插图

### 实验运用主要知识点

**语法基础**：、指针与引用、标准模板库

**概念基础**：图的存储必要完整、准确地反应顶点集和边集的信息。根据不同图的结构和算法，采用不同的存储方式对程序的效率产生相当大的影响，因此所选的存储结构应适合于带求解的问题。

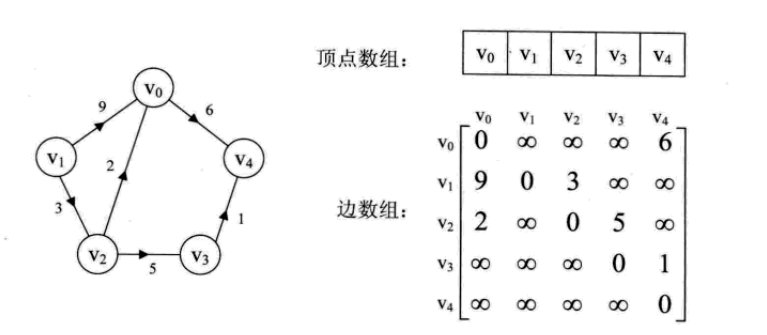


图86 有向带权图邻接矩阵

1. 邻接矩阵法：所谓邻接矩阵存储，是指用一个一位数组存储图中顶点的信息，用一个二维数组存储图中边的信息（即各顶点之间的邻接关系），存储顶点之间邻接关系的二维数组称为邻接矩阵。结点为n的图的邻接矩阵A是n\*n的。将G的顶点编号为v1,v2,…,vn。对于带权图而言，若顶点vi和vj之间有边相连则邻接矩阵中对应项存放着该边对应的权值，若顶点vi和vj不相连则用来代表这两个顶点之间不存在边

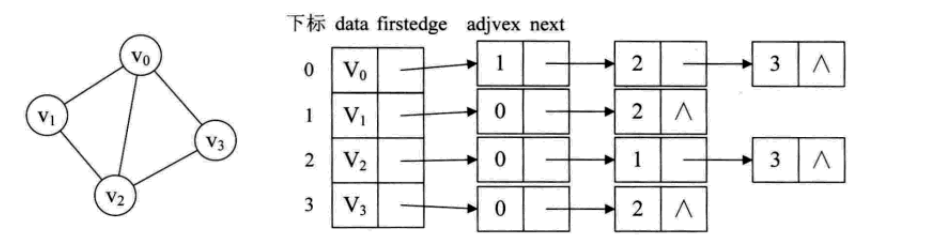


图87 无向图邻接表

1. 邻接表法：当一个图为稀疏图是，使用邻接矩阵法显然要浪费大量的存储空间，而图的邻接法结合了顺序存储和链式存储方法，大大减少了这种不必要的浪费。所谓邻接表，是指对图G中每个顶点vi建立一个单链表，第i个链表中的结点表示依附于顶点vi的边（对于有向图则是一顶点vi为尾的弧），这个单链表就称为顶点vi的边表（对于有向图则称为出边表）。边表的头指针和顶点的数据信息采用顺序存储，所以邻接表中存在两种结点：顶点表和边表结点
2. 对于有向图还有一种十字链表，无向图还有邻接多重表

**考察内容**：建立邻接矩阵、建立邻接表、邻接矩阵与邻接表转换

### 实验源码及备注

#### 操作说明

本实验数据均在int范围内，因此采用十六进制0x3f3f3f3f作为无穷大

#### 程序源码

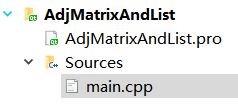


图88 题目一文件树

main.cpp文件

1. #include <bits/stdc++.h>
2. using namespace std;
3. const int inf = 0x3f3f3f3f;
4. const int maxn = 15;
5. int n, m;
6. int matrix[maxn][maxn];
7. struct edge {
8. int v, c;
9. edge(int v, int c) : v(v),
10. c(c)
11. {
12. }
13. };
14. vector<edge> G[maxn];
15. void init()
16. {
17. memset(matrix, 0x3f, sizeof matrix);
18. for (int i = 0; i < maxn; ++i) {
19. matrix[i][i] = 0;
20. G[i].clear();
21. }
22. }
23. int main()
24. {
25. cin>>n>>m;
26. init();
27. while (m--) {
28. int u, v, w;
29. cin>>u>>v>>w;
30. matrix[u][v] = w;
31. }
32. cout<<"opt-1: Generate adjacency matrix from graph."<<endl;
33. for (int i = 1; i <= n; ++i) {
34. for (int j = 1; j <= n; ++j)
35. if (matrix[i][j] != inf)
36. printf("(%d,%d,%d) \t", i, j, matrix[i][j]);
37. else printf("(%d,%d,%s)\t", i, j, "non");
38. cout<<endl;
39. }
40. cout<<endl;
41. cout<<"opt-2: Generate adjacency list from adjacency matrix."<<endl;
42. for (int i = 1; i <= n; ++i)
43. for (int j = 1; j <= n; ++j) {
44. if (matrix[i][j] != inf) G[i].push\_back(edge(j, matrix[i][j]));
45. }
46. for (int i = 1; i <= n; ++i) {
47. cout<<"start at "<<i<<"--> ";
48. for (auto e:G[i])
49. if (e.v != i)
50. printf("(%d,%d) ", e.v, e.c);
51. cout<<endl;
52. }
53. cout<<endl;
54. cout<<"opt-3: Generate adjacency matrix from adjacency list."<<endl;
55. for (int i = 1; i <= n; ++i)
56. for (int j = 1; j <= n; ++j) {
57. if (i == j) matrix[i][j] = 0;
58. else matrix[i][j] = inf;
59. }
60. for (int i = 1; i <= n; ++i)
61. for (auto e:G[i])
62. matrix[i][e.v] = e.c;
63. for (int i = 1; i <= n; ++i) {
64. for (int j = 1; j <= n; ++j)
65. if (matrix[i][j] != inf)
66. printf("(%d,%d,%d) \t", i, j, matrix[i][j]);
67. else printf("(%d,%d,%s)\t", i, j, "non");
68. cout<<endl;
69. }
70. return 0;
71. }
72. */\**
73. 6 10
74. 1 2 5
75. 2 3 4
76. 4 3 5
77. 5 4 5
78. 6 5 1
79. 6 1 3
80. 3 1 8
81. 1 4 7
82. 3 6 9
83. 4 6 6
84. \*/

### 实验运行截图及说明

#### 运行截图

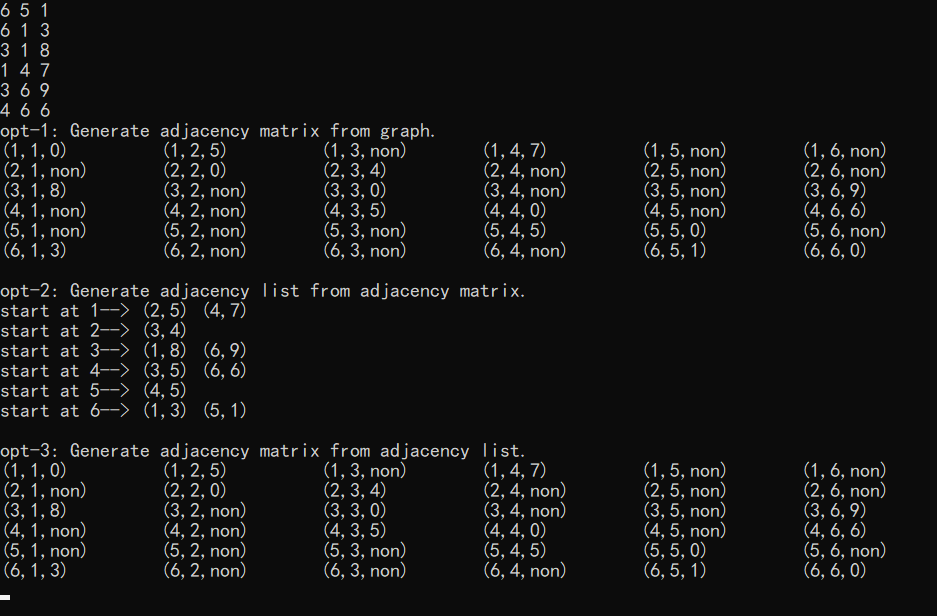


图89 题目一运行结果

#### 程序说明

**时间复杂度**：都是枚举两个端点确定一条边因此复杂度为

## 题目二 图的遍历算法

### 实验内容

题目二：编写一个程序，实现图的遍历算法，并在此基础上设计一个主程序完成如下功能：

1. 输出图（二）的有向图G从顶点V1开始的深度优先遍历序列序列（递归算法）
   1. 输出如图（二）的有向图G从顶点v1开始的深度优先遍历序列（非递归算法）
   2. 输出如图（二）的有向图G从顶点v1开始的广度优先遍历序列（选做）

### 实验运用主要知识点

**语法基础**：、指针与引用、标准模板库

**概念基础**：

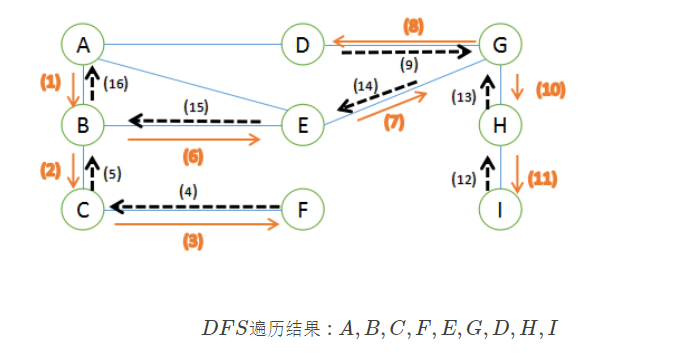


图90 图的DFS

1. 图的深度优先搜索：深度优先搜索在搜索过程中每当访问到某一个顶点后，需要**递归地访问此顶点的所有未访问过的相邻顶点**。因而，这种搜索将尽可能深地持续探索，直到无法继续为止。这种策略使得深度优先搜索在很多应用中非常有用，包括环检测以及拓扑排序

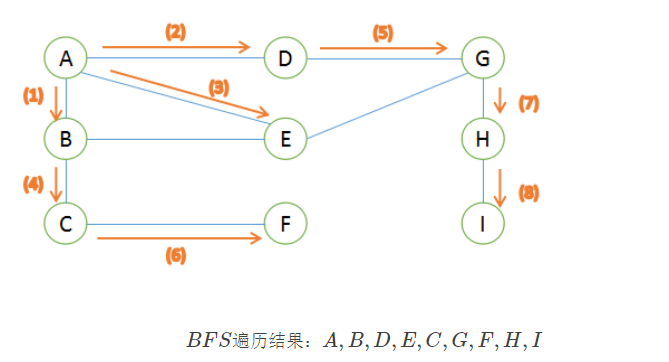


图91 图的BFS

1. 图的广度优先搜索：广度优先搜索在进一步探索图中的顶点之前，先访问当前顶点的所有邻接顶点。这种查找方法在很多应用中都非常有用，包括查找出最小生成树和最短路径问题

### 实验源码及备注

#### 操作说明

本实验采用从结点1开始进行搜索

#### 程序源码

1. #include <bits/stdc++.h>
2. using namespace std;
3. const int maxn = 15;
4. bool vis[maxn];
5. vector<int> vec, G[maxn];
6. int pre[maxn];
7. void dfsRec(int st)
8. {
9. for (auto e:G[st]) {
10. if (!vis[e]) {
11. cout<<st<<"->"<<e<<endl;
12. vec.push\_back(e);
13. vis[e] = 1;
14. dfsRec(e);
15. }
16. }
17. }
18. void dfsUnrec()
19. {
20. stack<int> sta;
21. sta.push(1);
22. vis[1] = 1;
23. while (sta.size()) {
24. int now = sta.top();
25. if (now != 1) cout<<pre[now]<<"->"<<now<<endl;
26. vec.push\_back(now);
27. sta.pop();
28. vis[now] = 1;
29. for (auto e: G[now])
30. if (!vis[e]) {
31. pre[e] = now;
32. sta.push(e);
33. }
34. }
35. }
36. void bfs()
37. {
38. vis[1] = 1;
39. vec.push\_back(1);
40. queue<int> que;
41. que.push(1);
42. while (que.size()) {
43. int now = que.front();
44. que.pop();
45. for (auto e:G[now]) {
46. if (!vis[e]) {
47. cout<<now<<"->"<<e<<endl;
48. vec.push\_back(e);
49. vis[e] = 1;
50. que.push(e);
51. }
52. }
53. }
54. }
55. int main()
56. {
57. int n, m;
58. cin>>n>>m;
59. while (m--) {
60. int u, v;
61. cin>>u>>v;
62. G[u].push\_back(v);
63. }
64. cout<<"opt-1: dfsRec from node 1."<<endl;
65. memset(vis, 0, sizeof vis);
66. vec.push\_back(1);
67. vis[1] = 1;
68. dfsRec(1);
69. cout<<"Result: ";
70. for (auto e:vec) cout<<e<<' ';
71. cout<<endl<<endl;
72. cout<<"opt-2: dfsUnrec from node 1."<<endl;
73. memset(vis, 0, sizeof vis);
74. vec.clear();
75. dfsUnrec();
76. for (auto e:vec) cout<<e<<' ';
77. cout<<endl<<endl;
78. cout<<"opt-3: bfs from node 1."<<endl;
79. memset(vis, 0, sizeof vis);
80. vec.clear();
81. bfs();
82. cout<<"Result: ";
83. for (auto e:vec) cout<<e<<' ';
84. cout<<endl<<endl;
85. return 0;
86. }
87. */\**
88. 6 10
89. 1 2
90. 1 4
91. 2 3
92. 3 1
93. 3 6
94. 4 3
95. 4 6
96. 5 6
97. 6 5
98. 6 1
99. \*/

### 实验运行截图及说明

#### 运行截图

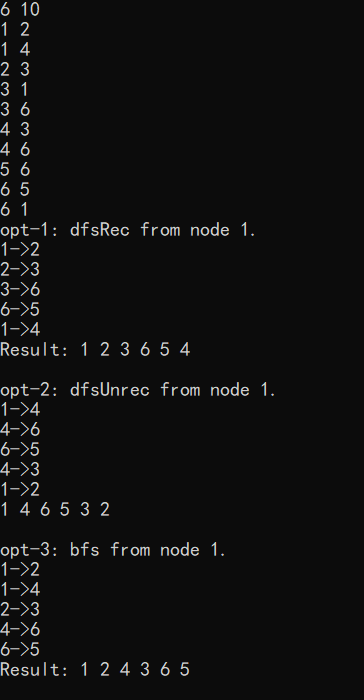


图92 题目二运行结果

# 实验六 查找表实验

## 题目一 顺序查找

### 实验内容

编写一个程序，输出在顺序表｛3,6,2,10,1,8,5,7,4,9｝中采用顺序方法查找关键字5的过程。

### 实验运用主要知识点

**语法基础**：

**概念基础**：将数组暴力遍历一遍对每个元素尝试匹配目标值

### 实验源码及备注

#### 操作说明

匹配情况采用按位输出

#### 程序源码

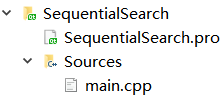


图93 顺序查找文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. signed main()
4. {
5. int x = 5;
6. int a[] = {3, 6, 2, 10, 1, 8, 5, 7, 4, 9};
7. bool flag = false;
8. for (int i = 0; i < 10; ++i) {
9. if (a[i] == x) {
10. cout<<"match succeed at pos: "<<i<<endl;
11. flag = true;
12. break;
13. } else cout<<"match failed at pos: "<<i<<endl;
14. }
15. if (flag) cout<<"Yes"<<endl;
16. else cout<<"No"<<endl;
17. return 0;
18. }

### 实验运行截图及说明

#### 运行截图

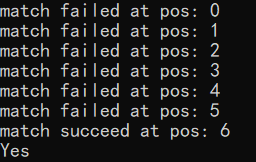


图94 顺序查找运行结果

## 题目二 二分查找

### 实验内容

编写一个程序，输出在顺序表｛1,2,3,4,5,6,7,8,9,10｝中采用二分查找方法查找关键字5的过程。

### 实验运用主要知识点

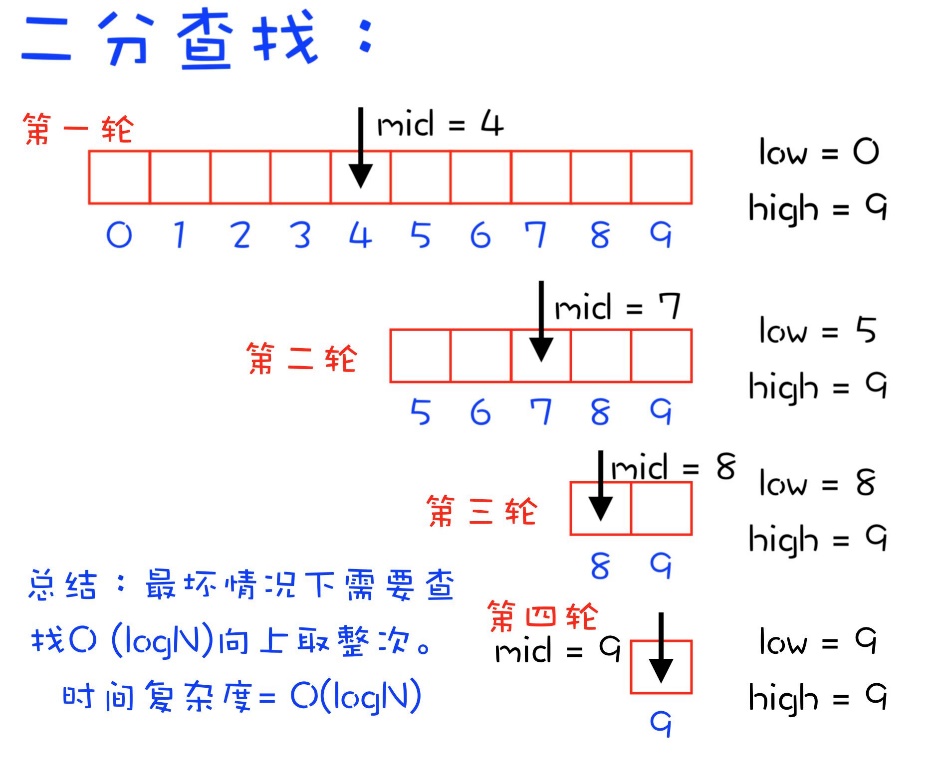


图94 二分法思路

**语法基础**：

**概念基础**：二分查找又称折半查找，仅适用于有序的顺序表。基本思想是首先将给定值key与表中中间位置的元素比较，若相等则查找成功，返回该元素的存储位置；若不等，则所需查找的元素只能在中间元素以外的前半部分或后半部分。然后在缩小的范围内继续进行同样的查找，如此重复直至找到为止，或确定表中没有所需要查找的元素，则查找不成功，返回查找失败的信息

**特性分析**：由二分查找实际上可以衍生出找上界与找下界等操作，对于整数与浮点数的查找方法也存在区别

**考察内容**：顺序表的整数二分查找

### 实验源码及备注

#### 操作说明

采用左闭右开区间的二分查找，数组元素从下标0开始

#### 程序源码

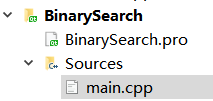


图95 二分查找文件树

main.cpp文件

1. #include <iostream>
2. #include <algorithm>
3. using namespace std;
4. bool binarySearch(int nums[], int len, int target)
5. {
6. int left = 0;
7. int right = len - 1;  *// 注意查找区间为全闭区间*
8. while (left <= right) {  *// 注意二分查找终止时为查找区间left>right*
9. *// cout<<"dubug: "<<left<<' '<<right<<endl;*
10. int mid = left + (right - left) / 2;
11. if (nums[mid] == target) {*// 锁定到某个target时返回*
12. cout<<"match succeed at pos: "<<mid<<endl;
13. return true;
14. } else if (nums[mid] < target)
15. left = mid + 1;
16. else if (nums[mid] > target)
17. right = mid - 1;
18. cout<<"match failed at pos: "<<mid<<endl;
19. }
20. return false;
21. }
22. signed main()
23. {
24. int x = 5;
25. int a[] = {3, 6, 2, 10, 1, 8, 5, 7, 4, 9};
26. cout<<"sort before: ";
27. for (auto e: a) cout<<e<<' ';
28. cout<<endl;
29. sort(a, a + 10);
30. cout<<"sort  after: ";
31. for (auto e:a) cout<<e<<' ';
32. cout<<endl;
33. bool flag = binarySearch(a, 10, x);
34. if (flag) cout<<"Yes"<<endl;
35. else cout<<"No"<<endl;
36. return 0;
37. }

### 实验运行截图及说明

#### 运行截图

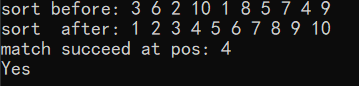


图96 二分查找运行结果

#### 程序说明

**时间复杂度**：因为是每次查找区间都会减半所以复杂度为

## 题目三 哈希表基本操作

### 实验内容

编写一个程序实现哈希表的相关运算，并在此基础上完成如下功能：

1. 建立｛16,74,60,43,54,90,46,31,29,88,77｝哈希表A[0……12]，哈希函数为：H(k)＝key％p，并采用线性探测再散列法解决冲突
2. 在上述哈希表中查找关键字为29的记录

在上述哈希表中删除关键字为77的记录，再将其插入。

### 实验运用主要知识点

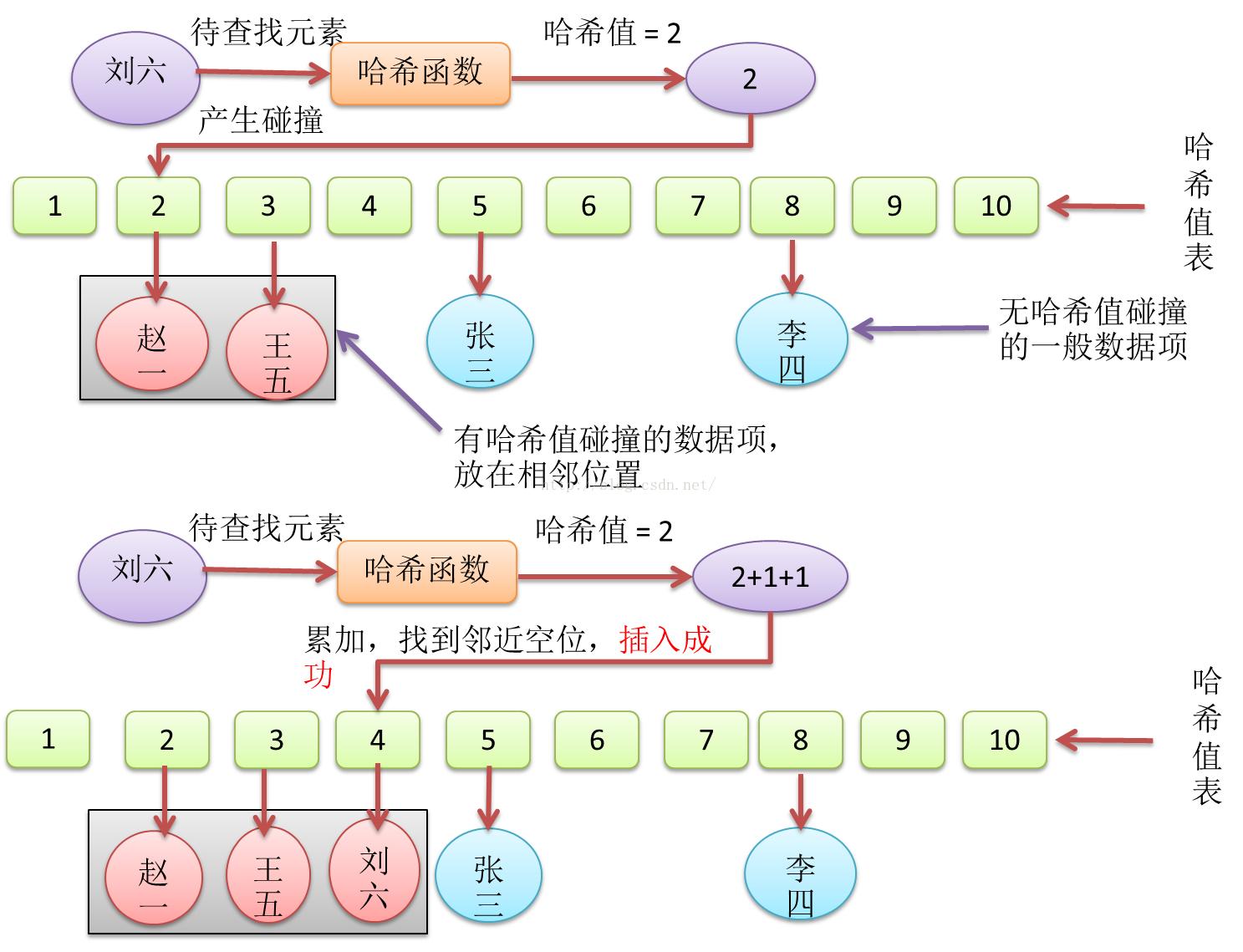


图97 哈希表思路

**语法基础**：

**概念基础**：线性表与树表的查找中记录表中的位置与记录的关键字之间不存在确定关系，因此在这些表中查找记录是需要进行一系列的关键字比较。这类查找方法在“比较”的基础上查找的效率取决于比较的次数。

1. 散列函数是一个把查找表中的关键字映射成该关键字对应的地址的函数，散列函数可能会把两个及两个以上的不同关键字映射到同一个地址，称这种情况为冲突；另一方面由于这样的冲突是不可避免的所以还要设计好处理冲突的方法
2. 散列表是根据关键字而进行直接访问的数据结构。也就是说散列表建立了关键字和存储地址之间的一种直接映射关系
3. 除数余留法：最简单也是最常用的方法，假定散列表表长为m，取一个不大于m但是最接近或等于m的质数p，利用以下公式把关键字转换成散列地址，除留余数法的关键是选好p，使得每个关键字通过该函数转换后等概率的映射到散列空间上的任一地址从而尽可能减少冲突的可能性
4. 开放定制法中的线性探测法即d=0,1,2,3,…,m-1

**特点分析**：

1. 理想情况下对散列表进行查找的时间复杂度为，即与表中元素的个数无关
2. 构造散列函数的时候应该注意：散列函数的定义域必须包含全部需要存储的关键字，而至于的范围则依赖于散列表的大小或地址范围；散列函数计算出来的地址应该能等概率、均匀分布在整个空间地址，从而减少冲突的发生；散列函数应该尽量简单能够在短时间内计算出任一关键字对应的散列地址
3. 线性探测法的特点是冲突发生是，顺序查看表中下一个单元直到找到一个空闲单元或查边全表

**考察内容**：哈希表线性探测法解决冲突、查找关键字、删除关键字、插入关键字

### 实验源码及备注

#### 程序源码

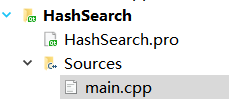


图98 哈希表文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. int val[] = {16, 74, 60, 43, 54, 90, 46, 31, 29, 88, 77}, A[13];
4. void hashinsert(int key)    *// 将key插入hash*
5. {
6. int pos = key%13;
7. while (A[pos]) {
8. pos = (pos+1)%13;
9. if (pos == key%13) {
10. cout<<"Table is full."<<endl;
11. return;
12. }
13. }
14. A[pos] = key;
15. cout<<"insert "<<key<<" at pos: "<<pos<<endl;
16. }
17. int hashfind(int key)   *// 在hash中查找key所处的下标*
18. {
19. int pos = key%13;
20. while (A[pos] != key) {
21. pos = (pos+1)%13;
22. if (pos == key%13) {
23. cout<<"Did not find it."<<endl;
24. return -1;
25. }
26. }
27. return pos;
28. }
29. void hasherase(int key)
30. {
31. int pos = hashfind(key);
32. if (pos != -1) {
33. A[pos] = 0;
34. cout<<"erase "<<key<<" at pos: "<<pos<<endl;
35. return;
36. }
37. }
38. signed main()
39. {
40. cout<<"opt-1: Construct the hash table."<<endl;
41. for (int i = 0; i < 11; ++i) hashinsert(val[i]);
42. for (auto e:A) cout<<e<<' ';
43. cout<<endl;
44. cout<<endl;
45. cout<<"opt-2: Find the keyword 29."<<endl;
46. cout<<hashfind(29)<<endl;
47. cout<<endl;
48. cout<<"opt-3: Delete 77 and insert it again."<<endl;
49. hasherase(77);
50. hashinsert(77);
51. cout<<endl;
52. cout<<"Final table."<<endl;
53. for (auto e:A) cout<<e<<' ';
54. return 0;
55. }

### 实验运行截图及说明

#### 运行截图

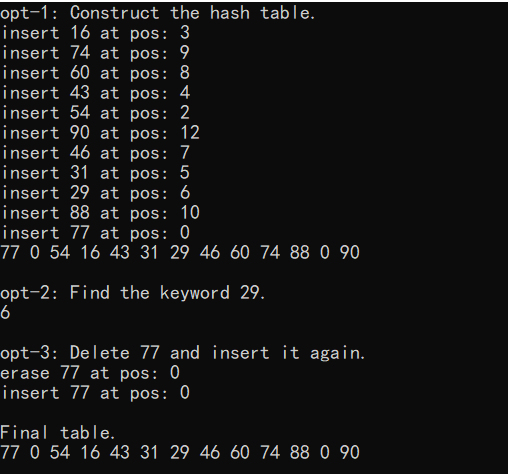


图99 哈希表运行结果

# 实验七 内排序实验

## 题目一 直接插入排序

### 实验内容

编写一个程序实现直接插入排序过程，并输出｛9,8,7,6,5,4,3,2,1,0｝的排序过程。

### 实验运用主要知识点

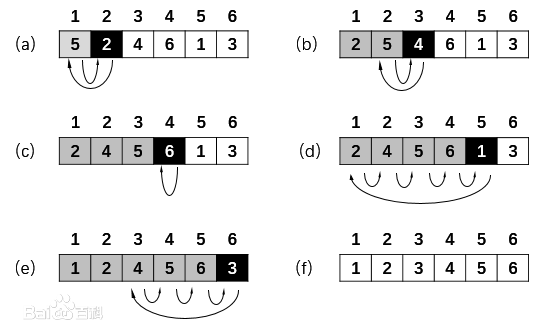


图100 直接插入排序思路

**语法基础**：

**概念基础**：把待排序的记录按其关键码值的大小逐个插入到一个已经排好序的有序序列中，直到所有的记录插入完为止，得到一个新的有序序列

**特点分析**：因为如果只有一个数的话是不用排序的，所以 n 个数只用给 n-1 个数找位置，所以总共需要循环 n-1 次。每一次循环过后有序序列变长，无序序列变短

**考察内容**：直接插入排序

### 实验源码及备注

#### 操作说明

元素从数组下标1开始，数组下标0位置作为哨兵位用来记录当前排序的元素供交换使用。为了便于清楚看到排序的效果每次排序后进行序列的输出

#### 程序源码

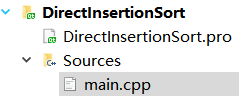


图101 直接插入排序

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. *// 标从1开始,0作为哨兵位*
4. const int n = 10;
5. int arr[n+1] = {-1, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
6. void dirInsertSort()
7. {
8. for (int i = 2; i <= n; i++) {
9. if (arr[i] <= arr[i-1]) {
10. arr[0] = arr[i];
11. int j;
12. for (j = i-1; arr[0] < arr[j]; --j)
13. arr[j+1] = arr[j];
14. arr[j+1] = arr[0];
15. cout<<"confirm element "<<arr[j+1]<<" pos--> ";
16. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
17. cout<<endl;
18. }
19. }
20. }
21. int main()
22. {
23. cout<<"sort before: ";
24. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
25. cout<<endl;
26. dirInsertSort();
27. cout<<"sort after: ";
28. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
29. cout<<endl;
30. return 0;
31. }

### 实验运行截图及说明

#### 运行截图

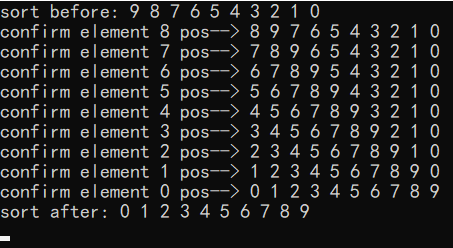


图102 直接插入排序运行结果

#### 程序说明

**时间复杂度**：因为每次查找最值需要遍历一次序列，总共需要给n-1个元素排序故总的迭代次数为次时间复杂度为

## 题目二 希尔排序

### 实验内容

编写一个程序实现希尔排序过程，并输出｛9,8,7,6,5,4,3,2,1,0｝的排序过程。

### 实验运用主要知识点

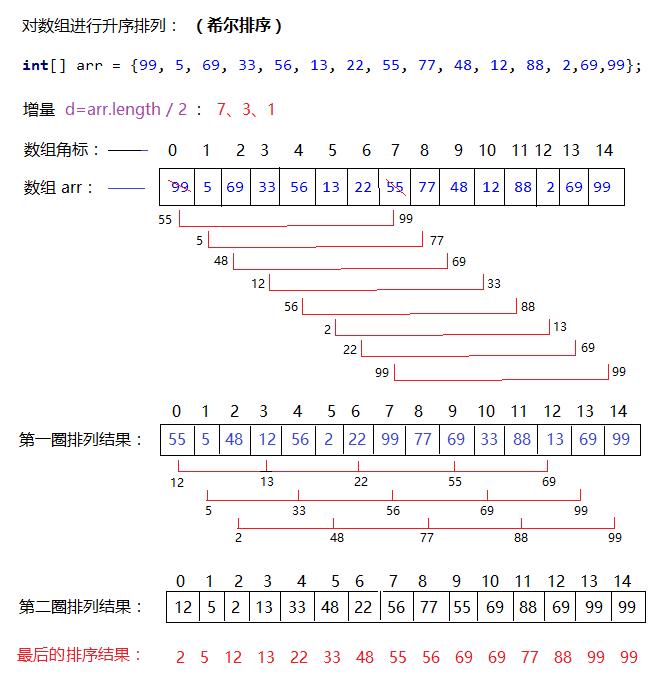


图103 希尔排序思路

**语法基础**：

**概念基础**：希尔排序是直接插入排序另一种更改思路，插入排序在对几乎已经排好序的数列操作时，效率高（即可以达到线性排序的效率）。但插入排序一般是低效的，因为插入排序每次只能将数据移动一位，所以希尔排序不再加快查找效率而是加快移动效率，通过设置不同增量可以让特别靠后的小数向前“跳”着到序列前面，又称“缩小增量排序”

**特点分析**：

1. 希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序。随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个数列恰被分成一组形成有序序列
2. 希尔排序是第一个将排序的时间复杂度降至以下的，该方法实质上是一种分组插入方法
3. 步长的选择并没有一个固定的科学方法，发明作者推荐每次步长减半，最终进行一次步长为1的排序

**考察内容**：希尔排序

### 实验源码及备注

#### 操作说明

希尔排序可以看做是多次不同步长的分组插入排序

#### 程序源码

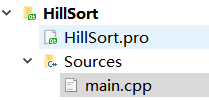


图104 希尔排序文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. const int n = 10, t = 4;
4. int arr[n+1] = {-1, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}, dlta[4] = {5, 3, 2, 1};
5. void HillInsert(int k)
6. {
7. for (int i = k+1; i <= n; i++) {
8. if (arr[i] < arr[i-k]) {
9. arr[0] = arr[i];
10. int j;
11. for (j = i-k; j > 0 && arr[0] < arr[j]; j -= k)
12. arr[j+k] = arr[j];
13. arr[j+k] = arr[0];
14. }
15. }
16. }
17. void HillSort()
18. {
19. for (int i = 0; i < t; ++i) {
20. HillInsert(dlta[i]);
21. cout<<"swap element at dlta: "<<dlta[i]<<"--> ";
22. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
23. cout<<endl;
24. }
25. }
26. signed main()
27. {
28. cout<<"sort before: ";
29. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
30. cout<<endl;
31. HillSort();
32. cout<<"sort after: ";
33. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
34. cout<<endl;
35. return 0;
36. }

### 实验运行截图及说明

#### 运行截图

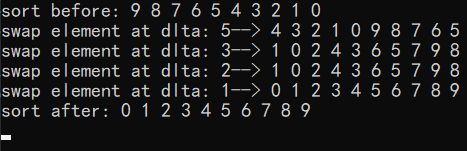


图105 希尔排序运行结果

#### 程序说明

**时间复杂度**：一般认为希尔排序的平均时间复杂度为

## 题目三 快速排序

### 实验内容

编写一个程序实现快速排序过程，并输出｛6,8,7,9,0,1,3,2,4,5｝的排序过程。

### 实验运用主要知识点

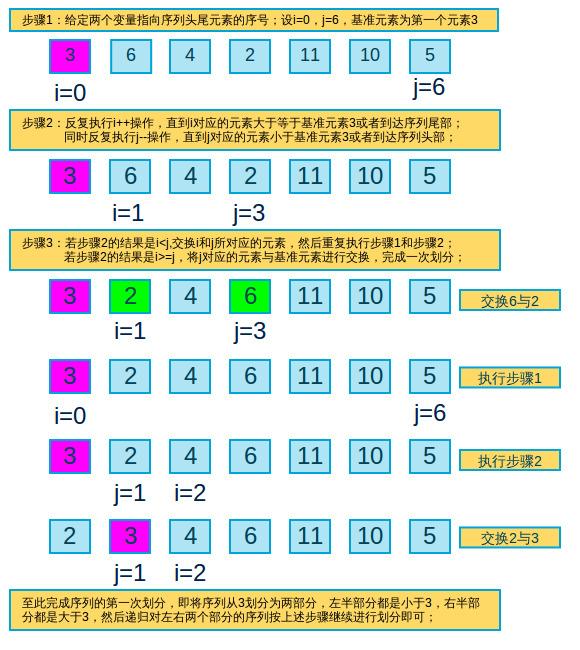


图106 快速排序思路

**语法基础**：

**概念基础**：基于分治的思想，是冒泡排序的改进型。首先在数组中选择一个基准点（该基准点的选取可能影响快速排序的效率，一般选择序列的开头元素），然后设两个指针（指向起始位置，指向末尾）分别从数组的两端扫描数组，首先从后半部分扫描，如果发现有元素比该基准点的值小，就交换和位置的值，然后从前半部分扫描，发现有元素大于基准点的值，就交换和位置的值，如此往复循环，直到，然后把基准点的值放到这个位置。一次排序就完成了。以后采用递归的方式再分别对前半部分和后半部分排序，当前半部分和后半部分均有序时该数组就自然有序了

**特点分析**：它是基于关键字比较的内部排序算法中速度最快者，快速排序因此而得名。快速排序主要消耗在区间的划分操作上，每一次划分（循环）结束也就确定了基准点在最终排好序的数列中的绝对位置，所以划分次即完成了对个数的排序。快速排序是对[冒泡排序](https://baike.baidu.com/item/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F/4602306)算法的一种改进

**考察内容**：快速排序

### 实验源码及备注

#### 操作源码

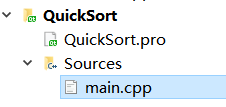


图107 快速排序文件树

main.cpp文件

1. #include <iostream>
2. using namespace std;
3. const int n = 10;
4. int arr[n+1] = {-1, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
5. void quickSort(int low, int high)
6. {
7. int i, j, key;
8. if (low > high) return;
9. i = low;
10. j = high;
11. key = arr[low];
12. while (i != j) {
13. while (j > i && arr[j] >= key) j--;
14. while (i < j && arr[j] <= key) i++;
15. if (i != j) {
16. int middle = arr[i];
17. arr[i] = arr[j];
18. arr[j] = middle;
19. }
20. }
21. arr[low] = arr[i];
22. arr[i] = key;
23. cout<<"confirm element "<<arr[i]<<" pos-->";
24. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
25. cout<<endl;
26. quickSort(low, i-1);
27. quickSort(i+1, high);
28. }
29. signed main()
30. {
31. cout<<"sort before: ";
32. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
33. cout<<endl;
34. quickSort(1, 10);
35. cout<<"sort after: ";
36. for (int i = 1; i <= n; ++i) cout<<arr[i]<<' ';
37. cout<<endl;
38. return 0;
39. }

### 实验运行截图及说明

#### 运行截图

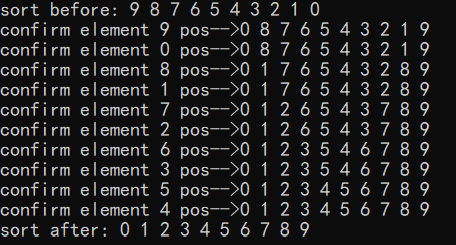


图108 快速排序运行结果

#### 程序说明

**时间复杂度**：快速排序的平均时间复杂度为

**合并操作**：合并操作将两个已经排好序的子序列一次从头部取较小的元素放入大序列，如果最后有一方剩余则直接全部放到大序列尾部。需要注意的是放入大序列的时候是从下标开始存

# 参考文献

[1]王道论坛.2022年数据结构考研复习指导[M].电子工业出版社:北京,2021.1:1.

[2]殷人昆.数据结构（用面向对象方法与C++语言描述）[M].清华大学出版社:北京,2007.6:1.

[3]喻梅、于瑞国.ACM/ICPC算法基础训练教程[M].清华大学出版社:北京,2015:1.

[4]王振华.算法与数据结构PPT[Z].中国地质大学（北京）:中国地质大学（北京）,2021.

[5]解学武.数据结构与算法教程，数据结构C语言版教程[EB/OL].http://data.biancheng.net/,2001-05-21.

[6]无名氏.数据结构与算法[EB/OL].https://www.runoob.com/data-structures/data-structures-tutorial.html,2013.

[7]Xander.数据结构[EB/OL].https://gitee.com/xander23333,2019.

[8]Emily静.线索化二叉树（前序线索化，中序线索化）[EB/OL].https://blog.csdn.net/qq\_39295755/article/details/79535406,2018-03-13.

[9]Liu Zhian.三元组（稀疏矩阵的转置）[EB/OL].https://blog.csdn.net/qq\_37174526/article/details/78244152?utm\_source=blogxgwz5,2017-10-15.

[10]kong\_xz.广义表[EB/OL].https://blog.csdn.net/kong\_xz/article/details/79484843,2018-03-08.

[11]薛定谔的猫ovo.二叉树遍历应用之根据前序遍历建树[EB/OL].https://blog.csdn.net/weixin\_44162361/article/details/115581781,2021-04-11.

[12]胖媛.二叉树的前序遍历（两种方法）[EB/OL].https://blog.csdn.net/weixin\_44919969/article/details/101107608,2019-09-21.

[13]jfkidear.树的高度和深度的区别[EB/OL].https://blog.csdn.net/jfkidear/article/details/52916712,2016-10-24.

[14]Jivan2233.二叉树——根据先序（后序）和中序遍历建树[EB/OL].https://blog.csdn.net/q982151756/article/details/78711844,2017-12-04.

[15]依然有清风.C++优先队列的重载（最小堆、最大堆）[EB/OL].https://blog.csdn.net/sinat\_37205608/article/details/82460301?utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-1.control&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-1.control,2018-09-06.

[16]成长的小牛233.赫夫曼编码长度计算问题？[EB/OL].https://blog.csdn.net/dreamzuora/article/details/53450611,2016-12-04.

[17]hellobigorange.顺序表[EB/OL].https://blog.csdn.net/qq\_34229228/article/details/101378722,2019-09-25.

[18]万里归来少年心.泛型编程[EB/OL].https://blog.csdn.net/liyazhen2011/article/details/82349890,2018-09-03.

[19]王卓.数据结构与算法基础[EB/OL].https://www.bilibili.com/video/BV1nJ411V7bd?from=search&seid=11496931962275287291,2020-01-10.

[20]部分图片与文字来源于网络