《数据结构与算法》
实验教学指导书

适用于 2025-2026 学年第 1 学期

编写： 王艳艳、李振华、肖际伟、高洪峰、田广野、蒋超亚

山东大学控制科学与工程学院

目　录

适用教材：

1. 《数据结构与算法》C++版，清华大学出版社，游洪跃等著。

参考书：

1. 《数据结构与算法实验和课程设计》（第 2 版），清华大学出版社，游洪跃等著。

2. 数据结构算法与应用：C++语言描述。[美] 萨特吉·萨尼（Sartaj Sahni） 著，王立柱，刘志红译

3. 殷人昆 编著.数据结构(用面向对象方法与 C++描述)(第 2 版).清华大学出版社。

4. 张铭王腾蛟 赵海燕编著.数据结构与算法.高等教育出版社。

实验一、线性表和栈的操作

一、实验目的

实验 1.1、线性表的操作

1、掌握线性表的顺序存储表示和链式存储表示。

2、掌握顺序表和链表的基本操作算法，包括创建、取值、查找、插入、删除等基本操作的实现。

3、了解线性表两种不同存储结构的特点，会灵活运用线性表解决某些实际问题。

实验 1.2、栈的操作

1、掌握栈的顺序存储和链式存储结构的实现方法。

2、掌握栈的基本操作算法，包括初始化、入栈、出栈、取栈顶元素、判空等操作。

3、理解栈"后进先出"（LIFO）的特性，并能够灵活运用栈解决实际问题。

4、掌握栈在括号匹配和表达式求值中的应用。

二、实验内容及要求

实验 1.1、线性表的操作

分别利用线性表顺序存储结构和链式存储结构结构实现教材信息的管理，教材信息包含：ISBN、出版社、作者、价格等信息。

1、线性表顺序存储结构下基本操作的实现（初始化、建表、取值、查找、插入、删除、两个非递减有序链表的归并等）。

2、线性表链式存储结构下基本操作的实现（初始化、建表、取值、查找、插入、删除、两个非递减有序链表的归并等）

实验 1.2、栈的操作

实现一个简单的表达式计算器，包含以下功能：

1、基本栈操作的实现

- 实现顺序栈的基本操作（初始化、入栈、出栈、取栈顶、判空）
- 实现链式栈的基本操作（初始化、入栈、出栈、取栈顶、判空）

2、括号匹配检查

- 检查表达式中的小括号()是否正确匹配
- 扩展：检查多种括号（()、[]、{}）

3、个位数表达式转换与计算

- 实现个位数中缀表达式转后缀表达式
- 实现个位数后缀表达式的计算
- 支持四则运算（+、-、*、/）

三、核心概念详解

1、什么是中缀表达式?

中缀表达式就是我们日常使用的数学表达式，运算符位于两个操作数中间。

例如：

- 2 + 3 （加号在 2 和 3 中间）
- 5 * 6 （乘号在 5 和 6 中间）
- (2 + 3) * 4 （复杂一点的表达式）
- 2 + 3 * 4 - 5 （需要考虑运算优先级）

特点：

- 人类容易理解和书写
- 需要考虑运算符优先级（先乘除后加减）
- 需要括号来改变运算顺序
- 计算机直接计算比较困难

2、什么是后缀表达式（逆波兰表达式）?

后缀表达式是将运算符放在操作数后面的表达式。

对应关系举例：

中缀表达式	后缀表达式	说明
2 + 3	2 3+	先写 2 和 3，再写+
5 * 6	5 6 *	先写 5 和 6，再写*
2 + 3 * 4	2 3 4 * +	先算 3 和 4* 结果再与 2 相加
(2 + 3) * 4	2 3 + 4 *	先算括号内 2 3 +得 5，再 5 4 *

特点：

- 不需要括号
- 不需要考虑优先级
- 计算机容易计算（用栈即可）
- 人类不太容易理解

3、为什么要转换?

- 中缀表达式：人类友好，但计算机处理困难
- 后缀表达式：计算机友好，用栈就能轻松计算

4、后缀表达式如何计算?

使用栈，规则非常简单：

1、从左到右扫描后缀表达式

2、遇到数字就入栈

3、遇到运算符就从栈中弹出两个数进行运算，结果入栈

4、最后栈中剩下的就是结果

示例：计算 2 3 4 * +（对应中缀：2 + 3 * 4）

1. 读到 2，入栈　　栈：[2]
2. 读到 3，入栈　　栈：[2, 3]
3. 读到 4，入栈　　栈：[2, 3, 4]
4. 读到*，弹出 4 和 3，计算 3*4=12，结果入栈　　栈：[2, 12]
5. 读到+，弹出 12 和 2，计算 2+12=14，结果入栈　　栈：[14]

结果: 14

四、算法思想

实验 1.1、线性表的操作

1、初始化

初始化操作主要涉及到为顺序表分配内存空间，并设置其初始状态。通过定义一个顺序表的结构体，并为其动态分配一个足够大小的数组来完成。算法上，是内存分配的过程。

2、建表

建表操作是在初始化后的顺序表中填充数据元素。这是通过遍历一个给定的数据集合（如数组），并将每个元素依次存入顺序表的数组中来实现。算法上，这是一个简单的遍历过程。

3、取值

取值操作是根据给定的位置索引，从顺序表中获取对应位置上的数据元素。由于顺序表的数据元素在内存中连续存放，因此可以通过直接计算偏移地址来获取所需元素。算法上，这是一个简单的数组访问操作。

4、查找

查找操作是在顺序表中搜索具有给定值的元素，并返回其位置索引。通过遍历整个顺序表来查找元素

5、插入

插入操作是在顺序表的指定位置插入一个新的数据元素。这涉及到将指定位置及其之后的所有元素向后移动一个位置，为新元素腾出空间。算法上，这涉及到元素的移动操作。

6、删除

删除操作是从顺序表的指定位置移除一个数据元素。这涉及到将指定位置之后的所有元素向前移动一个位置，以填补被删除元素留下的空白。算法上，这涉及到元素的移动操作。

7、两个非递减有序链表的归并

归并算法通过遍历两个链表，比较当前结点的值，选择较小的结点接入新链表中，并移动指针继续比较，直到所有结点都被遍历完。这涉及到创建一个新的顺序表或数组来存储归并后的结果，并通过双指针技术来遍历两个原顺序表或数组。

实验 1.2、栈的操作

1. 括号匹配算法

算法步骤：

1. 创建一个空栈
2. 从左到右扫描表达式
3. 遇到左括号'('，入栈
4. 遇到右括号')'：
   - 如果栈空，说明右括号多了，不匹配
   - 如果栈不空，出栈一个左括号
5. 扫描结束后，如果栈空则匹配，否则左括号多了

2. 中缀转后缀算法

算法步骤：

1. 创建一个空栈（注意：此栈用于存放运算符!）
2. 从左到右扫描中缀表达式
3. 遇到操作数，直接输出
4. 遇到运算符：
   - 如果栈空，直接入栈
   - 如果栈顶运算符优先级低，直接入栈
   - 否则，弹出栈顶运算符并输出，重复比较后续新的栈顶运算符
5. 遇到左括号，直接入栈
6. 遇到右括号，弹出运算符直到遇到左括号
7. 扫描结束，弹出栈中所有运算符

五、源代码示例

1、线性表顺序存储结构简化代码示例

```
#include <iostream>        // 输入输出流头文件
#include <fstream>    // 文件操作头文件
#include <string>      // 字符串操作头文件
#include <iomanip>   // 输入输出操纵运算头文件
using namespace std; // 调用命名空间 std 中所有标识符，简化标准库函数的调用
 #define MAXSIZE 1000 // 定义数组的最大容量
// 预定义常量及预定义类型
#define OK 1             // 操作成功时的返回值
#define ERROR 0             // 操作失败时的返回值
#define OVERFLOW -2 // 内存溢出错误的返回值
typedef int Status; // 定义 Status 为返回值类型，用于表示函数的执行状态
 // 定义 Book 结构体来存储书籍的信息
typedef struct
{
```

```
    int id;             // 书籍的 ISBN 编号
    double price; // 书籍的价格
} Book;
 // 定义顺序表（顺序存储的线性表）结构体，用于存储一组书籍
typedef struct
{
    Book *elem; // 指向存储书籍的动态数组
    int length; // 顺序表的当前长度
} SqList;
 // 1.初始化
Status initList(SqList &L)
{
    L.elem = new Book[MAXSIZE]; // 动态分配最大容量的数组空间
    if (!L.elem)                    // 如果内存分配失败
        exit(OVERFLOW);                 // 退出程序并返回溢出错误
    L.length = 0;                   // 初始化顺序表长度为 0
    return OK;                          // 返回成功状态
}

// 3.取值
Status GetElem(SqList L, int i, Book &e)
{
    if (i < 1 || i > L.length) // 检查 i 是否超出有效范围
        return ERROR;               // 返回错误状态
    e = L.elem[i - 1];          // 获取第 i 个元素（数组下标从 0 开始）
    return OK;                          // 返回成功状态
}
// 4.查找
int LocateElem(SqList L, Book e)
{
    for (int i = 0; i < L.length; i++) // 遍历顺序表中的每本书
    {
        if (L.elem[i].id == e.id) // 如果找到匹配的 ISBN 编号
            return i + 1;           // 返回位置（从 1 开始）
    }
    return 0; // 未找到，返回 0
}
//5.插入
```

```
Status ListInsert(SqList &L, int i, Book e)
{
    if ((i < 1) || (i > L.length + 1)) // 判断要插入的位置是否合法。
        return ERROR;
    if (L.length == MAXSIZE) // 判断当前列表是否已经满了，如果满了就返回 ERROR
        return ERROR;
    for (int j = L.length - 1; j >= i - 1; j--)
    {
        L.elem[j + 1] = L.elem[j];
    }
    L.elem[i - 1] = e;
    ++L.length;
    return OK;
}
//6. 删除
Status ListDelete(SqList &L, int i)
{
    // 在顺序表中删除第 i 个元素, i 值的合法范围是 1<=i<=L.length
    if ((i < 1) || (i > L.length))
        return ERROR;                    // 不合法，返回错误状态

    // 从位置 i 开始，将后面的元素依次向前移动
    for (int j = i; j < L.length; j++)
    {
        L.elem[j - 1] = L.elem[j]; // 将位置 j 的元素赋值给位置 j-1
    }
    --L.length; // 顺序表长度减少 1
    return OK;   // 返回成功状态
}
//7. 归并

/*给定两个非递减有序的顺序表 A 和 B，将它们合并成一个新的非递减有序的顺序表 C。
合并步骤：
    创建一个新的顺序表 C，它的长度是 A 和 B 的长度之和。
    设置两个指针 i 和 j，分别指向顺序表 A 和 B 的起始位置。
    比较 A[i] 和 B[j]：
        如果 A[i] <= B[j]，将 A[i] 放入 C，然后指针 i 右移。
        否则，将 B[j] 放入 C，然后指针 j 右移。
```

当其中一个顺序表（A 或 B）的所有元素都被处理完后，将另一个顺序表剩余的元素直接放入 C。

返回合并后的顺序表 C。 */

```
// 归并两个非递减有序的顺序表 L 和 A 到 B
Status MergeList(SqList L, SqList A, SqList &B)
{
    int i = 0, j = 0, k = 0;       // 初始化指针
    B.length = A.length + L.length; // B 的长度为 A 和 L 之和
    B.elem = new Book[B.length];      // 为 B 分配空间
    if (!B.elem)                          // 检查内存分配
        return OVERFLOW;
    // 归并过程：当 L 和 A 都还有元素时进行比较
    while (i < L.length && j < A.length)
    {
        if (L.elem[i].id <= A.elem[j].id)
        {
            B.elem[k++] = L.elem[i++]; // L 的元素较小，放入 B
        }
        else
        {
            B.elem[k++] = A.elem[j++]; // A 的元素较小，放入 B
        }
    }

    // 如果 L 还有剩余元素，直接放入 B
    while (i < L.length)
    {
        B.elem[k++] = L.elem[i++];
    }

    // 如果 A 还有剩余元素，直接放入 B
    while (j < A.length)
    {
        B.elem[k++] = A.elem[j++];
    }

    return OK; // 返回成功状态
}
```

```
// 9. 清空数据表
// 功能：释放顺序表的动态数组并将其长度重置为 0
Status ClearList(SqList &L)
{
    if (L.elem) // 如果顺序表已经有元素
    {
        delete[] L.elem;   // 释放动态分配的内存
        L.elem = nullptr; // 将指针设为 nullptr，防止悬空指针
    }
    L.length = 0; // 将顺序表长度重置为 0
    return OK;
}
// 函数：将顺序表 B 的值赋给 L
// 功能：清空 L 后，将 B 的值复制到 L 中
Status AssignList(SqList &L, SqList B)
{
    ClearList(L);                    // 先清空 L
    L.elem = new Book[B.length]; // 为 L 分配与 B 相同长度的内存
    if (!L.elem)                     // 检查内存分配是否成功
        return OVERFLOW;

    for (int i = 0; i < B.length; i++) // 复制 B 中的每个元素到 L
    {
        L.elem[i] = B.elem[i];
    }
    L.length = B.length; // 更新 L 的长度
    return OK;
}
 // 函数：显示菜单
// 功能：输出用户可执行的操作菜单
void showMenu()
{
    cout << "*****************************\n";
    cout << "****        1. 初始化       ****\n";
    cout << "****        2. 赋值         ****\n";
    cout << "****        3. 取值         ****\n";
    cout << "****        4. 查找         ****\n";
    cout << "****        5. 插入         ****\n";
```

```
    cout << "****         6. 删除          ****\n";
    cout << "****         7. 归并          ****\n";
    cout << "****         8. 输出          ****\n";
    cout << "****         9. 清空数据表    ****\n";
    cout << "****         0. 退出          ****\n";
    cout << "*****************************\n";
}
// 主函数
int main()
{
    SqList L, A, B;          // 定义顺序表 L A B
    Book e;                  // 定义书籍 e
    int count, i, ps; // count 用于存储书籍的数量，i 用于索引,ps 为元素的位置
    int choose = -1;     // 用户选择的操作，初始化为-1
    showMenu();              // 显示操作菜单

    while (choose != 0) // 当用户选择不是 0（退出）时，继续操作
    {
        cout << "请选择操作（0-8）:"; // 提示用户输入操作
        cin >> choose;                  // 读取用户选择
        switch (choose)                 // 根据选择执行对应的操作
        {
        case 1:                                 // 初始化顺序表
            initList(L);                        // 调用初始化函数
            cout << "初始化顺序表成功：\n"; // 输出提示
            break;

        case 2:                                                 // 赋值
            cout << "请输入图书的个数：";                       // 提示用户输入书籍数量
            cin >> count;                                               // 读取书籍数量
            cout << "请输入图书的信息（ISBN 编号 价格）：\n"; // 输入书籍信息
            for (i = 0; i < count; i++)                 // 循环输入每本书籍的信息
                cin >> L.elem[i].id >> L.elem[i].price;
            L.length = count;                           // 更新顺序表的长度
            break;

        case 3:                                                 // 取值
            cout << "请输入位置：";                             // 提示用户输入位置
```

```
        cin >> i;                                   // 读取位置
        GetElem(L, i, e);                           // 调用取值函数
        cout << e.id << "\t" << e.price << endl; // 输出书籍信息
        break;
    case 4:                                         // 查找
        cout << "请输入 ISBN 号码：";       // 提示用户输入要查找的 ISBN 编号
        cin >> e.id;                            // 读取 ISBN 编号
        cout << LocateElem(L, e) << endl; // 调用查找函数并输出结果
        break;
    case 5:
        // 插入
        cout << "请输入要插入的位置\n";
        cin >> ps;                                          // 读取插入位置
        cout << "请输入图书的信息（ISBN 编号  价格）： "; // 提示输入书籍信息
        cin >> e.id >> e.price;                             // 读取书籍的信息
    if (ListInsert(L, ps, e) == OK)
        {
            cout << "插入成功!\n";
        }
        else
            cout << "插入失败!\n";
        break;
    case 6:
        cout << "请输入要删除的元素的位置\n";
        cin >> ps;
        ListDelete(L, ps);
        break;
    case 7:                 // 归并
        initList(A); // 调用初始化函数
        initList(B);
        cout << "请输入要与 L 归并的顺序表 A\n";
        cout << "请输入图书的个数：";                  // 提示用户输入书籍数量
        cin >> count;                                       // 读取书籍数量
        cout << "请输入图书的信息（ISBN 编号  价格）：\n"; // 提示输入书籍信息
        for (i = 0; i < count; i++)                         // 输入每本书籍的信息
            cin >> A.elem[i].id >> A.elem[i].price;
        A.length = count;
        if (MergeList(L, A, B) == OK)
```

```
            {
                AssignList(L, B); // 将归并后的 B 赋给 L
                cout << "归并成功，并将 B 的值赋给 L!\n";
            }
            else
            {
                cout << "归并失败!\n";
            }
            break;

        case 8:
        cout << "输出顺序表内容：\n// 提示即将输出顺序表内容
            for (i = 0; i < L.length; i++)
                // 遍历顺序表中的每本书
            cout << left << setw(15) << L.elem[i].id << "\t" << left << setw(5) <<
L.elem[i].price << endl; // 输出每本书的 ISBN 编号和价格，格式化对齐
            break;
        case  9:
            ClearList(L);
            if (ClearList(L) == OK)
            {
                cout << "清空顺序表成功\n";
            }
            else
                cout << "清楚数据成功";
        }
    }
    return 0; // 程序正常结束
}
```

2、线性表链式存储结构简化代码示例

```
#include <iostream>        //输入输出流头文件
#include <fstream>    //文件操作头文件
#include <string>      //字符串操作头文件
#include <iomanip>   //输入输出操纵运算头文件
using namespace std; // 调用命名空间 std 中所有标识符
// 预定义常量及预定义类型
#define OK 1
```

```
#define ERROR 0
#define OVERFLOW -2
typedef int Status; // Status 是函数返回值类型，其值是函数结果状态代码。

// 表示部分：
typedef struct LNode
{    int data;            // 结点的数据域
     struct LNode *next; // 结点的指针域
} LNode, *LinkList;          // LinkList 为指向结构体 LNode 的指针类型
 // 实现部分：
// 1.初始化
Status InitList_L(LinkList &L)
{
     // 构造一个空的单链表 L
     L = new LNode; // 生成新结点作为头结点，用头指针 L 指向头结点
     L->next = NULL; // 头结点的指针域置空
     return OK;
}

// 2.头插法创建链表
void CreateList_H(LinkList &L, int n)
{    // 逆位序输入 n 个元素的值，建立到头结点的单链表 L
     LinkList p;
     // LNode *p;
     for (int i = 0; i < n; ++i)
     {
          p = new LNode; // 生成新结点*p
          cout << "Please input the data：" << i + 1 << endl;
          cin >> p->data; // 输入元素值赋给新结点*p 的数据域
          p->next = L->next;
          L->next = p; // 将新结点*p 插入到头结点之后
     }
} // CreateList_H

// 3.尾插法创建链表
int CreateList_T(LinkList &L, int n)
{
     LNode *p, *tail = L; // 尾指针
```

```
    for (int i = 0; i < n; i++)
    {
        p = new LNode;
        cout << "请输入数据：" << i + 1 << ": ";
        cin >> p->data;
        p->next = NULL;
        tail->next = p; // 将新结点加入尾部
        tail = p;       // 更新尾指针
    }
    return OK;
}

// 4. 取值（获取链表中的第 i 个元素）
Status GetElem(LinkList L, int i, int &e)
{
    LNode *p = L->next; // 指向第一个结点
    int j = 1;
    while (p && j < i) // 查找第 i 个结点
    {
        p = p->next;
        j++;
    }
    if (!p || j > i)
        return ERROR; // 第 i 个元素不存在
    e = p->data;          // 取出元素
    return OK;
}

// 5.查找链表中指定元素的位置
Status LocateElem(LinkList L, int e)
{
    LNode *p = L->next; // 指向第一个结点
    int j = 1;
    while (p)
    {
        if (p->data == e) // 如果找到匹配的元素
            return j;
        p = p->next;
```

```
        j++;
    }
    return ERROR; // 未找到，返回 0
}

//6.插入
Status ListInsert_L(LinkList &L, int i, int e)
{   // 在带头结点的单链线性表 L 的第 i 个元素之前插入元素 e
    LinkList p, s;
    p = L;
    int j = 0;
    while (p && j < i - 1)
    { // 寻找第 i-1 个结点
        p = p->next;
        ++j;
    }
    if (!p || j > i - 1)
        return ERROR;                          // i 小于 1 或者大于表长
    s = (LinkList)malloc(sizeof(LNode)); // 生成新结点
    s->data = e;
    s->next = p->next; // 插入 L 中
    p->next = s;
    return OK;
} // LinstInsert_L

//7. 删除元素
Status ListDelete(LinkList &L, int i)
{
    LNode *p = L; // 指向头结点
    int j = 0;
    while (p->next && j < i - 1) // 查找第 i-1 个结点
    {
        p = p->next;
        j++;
    }
    if (!(p->next) || j > i - 1)
        return ERROR; // 删除位置不合法
    LNode *q = p->next; // 要删除的结点
```

```
    p->next = q->next;  // 将 q 结点从链表中断开
    delete q;           // 释放 q 结点的内存
    return OK;
}

// 8.遍历
void TraverseList_L(LinkList L)
{
    LNode *p;
    p = L->next;
    while (p)
    {
        cout << p->data << "    ";
        p = p->next;
    }
    cout << endl;
}

//9.归并
Status MergeList_L(LinkList La, LinkList Lb, LinkList &Lc)
{
    // 已知单链线性表 La 和 Lb 的元素按值非递减排列。
    // 归并 La 和 Lb 得到新的单链线性表 Lc，Lc 的元素也按值非递减排列。
    LinkList pa, pb, pc;
    pa = La->next;
    pb = Lb->next;
    Lc = pc = La; // 用 La 的头结点作为 Lc 的头结点
    while (pa && pb)
    {
        if (pa->data <= pb->data)
        {
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        }
        else
        {
            pc->next = pb;
```

```
            pc = pb;
            pb = pb->next;
        }
    }
    pc->next = pa ? pa : pb; // 插入剩余段
    free(Lb);                // 释放 Lb 的头结点
    return OK;
} // MergeList_L

// 清空链表
Status AssignList(LinkList &L)
{
    LNode *p = L->next;
    while (p)
    {
        LNode *temp = p;
        p = p->next;
        delete temp; // 释放结点内存
    }
    L->next = NULL; // 重新初始化链表为空
    cout << "链表已清空！\n";
    return OK;
}

void showMenu()
{
    cout << "*****************************\n";
    cout << "****      1. 初始化      ****\n";
    cout << "****      2. 头插法创建  ****\n";
    cout << "****      3. 尾插法创建  ****\n";
    cout << "****      4. 取值        ****\n";
    cout << "****      5. 查找        ****\n";
    cout << "****      6. 插入        ****\n";
    cout << "****      7. 删除        ****\n";
    cout << "****      8. 遍历        ****\n";
    cout << "****      9. 归并        ****\n";
    cout << "****      0. 退出        ****\n";
    cout << "*****************************\n";
```

```
}

int main()
{
    LinkList L,La,Lb;
    int n,ps,count1,count2,i,i1,i2,e;
    int choose = -1;
    showMenu();
    while (choose != 0)
    {
        cout << "Please select（0-9）:";
        cin >> choose;
        switch (choose)
        {
        case 1: // 1.初始化
            InitList_L(L);
            cout << "Init successfully：\n";
            break;
        case 2: // 2.头插法创建单链表
            cout << "Please input the number of LNode：";
            cin >> n;
            CreateList_H(L, n);
            break;
        case 3: // 3.尾插法创建单链表
            cout << "Please input the number of LNode：";
            cin >> n;
            CreateList_T(L, n);
            break;
        case 4: // 4.取值
            cout << "请输入位置：";
            cin >> i ;                                  // 读取位置
            GetElem(L, i, e);                          // 调用取值函数
            cout << e << endl;
            break;
        case 5: // 5.查找
            cout << "请输入元素：";          // 提示用户输入查找
            cin >> e;                          // 读取
            cout << LocateElem(L, e) << endl; // 调用查找函数并输出结果
```

```
        break;
    case 6: // 6.插入
        cout << "请输入要插入的位置\n";
        cin >> ps;                                          // 读取插入位置
        cout << "请输入信息： "; // 提示用户输入书籍信息
        cin >> e;                                           // 读取
        if (ListInsert_L(L, ps, e) == OK)
        {
            cout << "插入成功!\n";
        }
        else
            cout << "插入失败!\n";
        break;
    case 7: // 7.删除
        cout << "请输入要删除的元素的位置\n";
        cin >> ps;
        ListDelete(L, ps);
        break;
    case 8: // 8.遍历
        TraverseList_L(L);
        break;
    case 9: // 9.归并
        InitList_L(La); // 初始化 La
        InitList_L(Lb); // 初始化 Lb
        cout << "请输入 La 链表的元素个数： ";
        cin >> count1;
        cout << "请输入 La 链表的元素：\n";
        CreateList_T(La, count1); // 使用尾插法创建 La 链表
        cout << "请输入 Lb 链表的元素个数： ";
        cin >> count2;
        cout << "请输入 Lb 链表的元素：\n";
        CreateList_T(Lb, count2); // 使用尾插法创建 Lb 链表

        // 合并两个链表，Lc 存储合并结果
        if (MergeList_L(La, Lb, L) == OK)
        {
            cout << "归并成功，结果链表为：\n";
            TraverseList_L(L); // 遍历输出归并后的结果链表
```

```
                }
                else
                {
                    cout << "归并失败！\n";
                }
                break;
            }
        }
        return 0;
}
```

3、栈的操作简化代码示例

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
using namespace std;

// 顺序栈的实现
class Stack {
private:
    int* data;          // 存储数据的数组
    int top;            // 栈顶指针
    int maxSize;        // 栈的最大容量

public:
    // 构造函数：初始化栈
    Stack(int size = 100) {
        maxSize = size;
        data = new int[maxSize];
        top = -1;   // -1 表示栈空
    }

    // 析构函数：释放内存
    ~Stack() {
        delete[] data;
    }
```

```
// 判断栈是否为空
bool isEmpty() {
    return top == -1;
}

// 判断栈是否已满
bool isFull() {
    return top == maxSize - 1;
}

// 入栈操作
bool push(int elem) {
    if (isFull()) {
        cout << "栈已满，无法入栈！" << endl;
        return false;
    }
    data[++top] = elem;  // 先移动指针，再存入数据
    return true;
}

// 出栈操作
bool pop(int& elem) {
    if (isEmpty()) {
        cout << "栈为空，无法出栈！" << endl;
        return false;
    }
    elem = data[top--];  // 先取出数据，再移动指针
    return true;
}

// 获取栈顶元素（不出栈）
bool getTop(int& elem) {
    if (isEmpty()) {
        return false;
    }
    elem = data[top];
    return true;
```

```
    }

    // 显示栈中所有元素
    void display() {
        cout << "栈中元素（从底到顶）： ";
        for (int i = 0; i <= top; i++) {
            cout << data[i] << " ";
        }
        cout << endl;
    }
};

// 字符栈（用于存储运算符和括号）
class CharStack {
private:
    char* data;
    int top;
    int maxSize;

public:
    CharStack(int size = 100) {
        maxSize = size;
        data = new char[maxSize];
        top = -1;
    }

    ~CharStack() {
        delete[] data;
    }

    bool isEmpty() {
        return top == -1;
    }

    bool push(char elem) {
        if (top == maxSize - 1) return false;
        data[++top] = elem;
        return true;
```

```
    }

    bool pop(char& elem) {
        if (isEmpty()) return false;
        elem = data[top--];
        return true;
    }

    bool getTop(char& elem) {
        if (isEmpty()) return false;
        elem = data[top];
        return true;
    }
};

// 括号匹配检查
class BracketChecker {
public:
    // 检查括号是否匹配
    bool checkBrackets(string expr) {
        CharStack stack(1000);

        for (int i = 0; i < expr.length(); i++) {
            char c = expr[i];

            // 遇到左括号，入栈
            if (c == '(' || c == '[' || c == '{') {
                stack.push(c);
            }
            // 遇到右括号，检查匹配
            else if (c == ')' || c == ']' || c == '}') {
                if (stack.isEmpty()) {
                    cout << "错误：右括号多余，位置：" << i << endl;
                    return false;
                }

                char top;
                stack.pop(top);
```

```
                // 检查是否匹配
                if ((c == ')' && top != '(') ||
                    (c == ']' && top != '[') ||
                    (c == '}' && top != '{')) {
                    cout << "错误：括号类型不匹配，位置：" << i << endl;
                    return false;
                }
            }
        }

        // 检查是否还有未匹配的左括号
        if (!stack.isEmpty()) {
            cout << "错误：左括号多余" << endl;
            return false;
        }

        return true;
    }
};

// 简单计算器（支持个位数的四则运算）
class SimpleCalculator {
private:
    // 判断字符是否为运算符
    bool isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/';
    }

    // 获取运算符优先级
    int getPriority(char op) {
        if (op == '+' || op == '-') return 1;
        if (op == '*' || op == '/') return 2;
        if (op == '(') return 0;   // 左括号优先级最低
        return -1;
    }

    // 执行运算
```

```
    int calculate(int a, int b, char op) {
        switch(op) {
            case '+': return a + b;
            case '-': return a - b;
            case '*': return a * b;
            case '/':
                if (b == 0) {
                    cout << "错误：除零！" << endl;
                    return 0;
                }
                return a / b;
            default: return 0;
        }
    }

public:
    // 中缀转后缀（简化版：只支持个位数）
    string infixToPostfix(string infix) {
        CharStack opStack(100);   // 运算符栈
        string postfix = "";        // 后缀表达式结果

        cout << "\n 转换过程：" << endl;
        cout << "步骤\t 当前字符\t 操作\t\t 运算符栈\t 后缀表达式" << endl;
        cout << "----------------------------------------------------------" << endl;

        for (int i = 0; i < infix.length(); i++) {
            char c = infix[i];

            // 跳过空格
            if (c == ' ') continue;

            cout << i+1 << "\t" << c << "\t\t";

            // 如果是数字，直接加入后缀表达式
            if (isdigit(c)) {
                postfix += c;
                postfix += " ";
                cout << "数字,直接输出\t";
```

```
        }
        // 如果是左括号，入栈
        else if (c == '(') {
            opStack.push(c);
            cout << "左括号,入栈\t";
        }
        // 如果是右括号
        else if (c == ')') {
            cout << "右括号,弹出至(\t";
            char top;
            while (!opStack.isEmpty()) {
                opStack.pop(top);
                if (top == '(') break;
                postfix += top;
                postfix += " ";
            }
        }
        // 如果是运算符
        else if (isOperator(c)) {
            cout << "运算符,比较优先级\t";
            char top;
            // 弹出优先级高于或等于当前运算符的运算符
            while (!opStack.isEmpty() &&
                   opStack.getTop(top) &&
                   getPriority(top) >= getPriority(c)) {
                opStack.pop(top);
                postfix += top;
                postfix += " ";
            }
            opStack.push(c);
        }

        // 显示当前状态
        displayStack(opStack);
        cout << "\t" << postfix << endl;
    }

    // 弹出栈中剩余的运算符
```

```
    char top;
    while (!opStack.isEmpty()) {
        opStack.pop(top);
        postfix += top;
        postfix += " ";
    }

    cout << "\n 最终后缀表达式： " << postfix << endl;
    return postfix;
}

// 计算后缀表达式
int evaluatePostfix(string postfix) {
    Stack numStack(100);   // 操作数栈

    cout << "\n 计算过程： " << endl;
    cout << "步骤\t 当前元素\t 操作\t\t 栈状态" << endl;
    cout << "----------------------------------------" << endl;

    int step = 1;
    for (int i = 0; i < postfix.length(); i++) {
        char c = postfix[i];

        // 跳过空格
        if (c == ' ') continue;

        cout << step++ << "\t" << c << "\t\t";

        // 如果是数字，入栈
        if (isdigit(c)) {
            numStack.push(c - '0');   // 字符转数字
            cout << "数字,入栈\t";
        }
        // 如果是运算符，弹出两个数计算
        else if (isOperator(c)) {
            int b, a;
            numStack.pop(b);   // 注意顺序
            numStack.pop(a);
```

```
                int result = calculate(a, b, c);
                numStack.push(result);
                cout << a << c << b << "=" << result << ",入栈\t";
            }

            numStack.display();
        }

        int result;
        numStack.pop(result);
        return result;
    }

private:
    // 辅助函数：显示字符栈内容
    void displayStack(CharStack& s) {
        // 这里简化处理，实际可以实现一个查看栈内容的方法
        cout << "[栈内容]";
    }
};

// 主函数
int main() {
    cout << "========== 栈的应用实验 ==========" << endl;

    // 1. 测试基本栈操作
    cout << "\n【实验 1：基本栈操作】" << endl;
    cout << "------------------------" << endl;
    Stack stack(5);

    cout << "执行入栈操作：push(10), push(20), push(30)" << endl;
    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.display();

    int elem;
    stack.pop(elem);
```

```
    cout << "执行出栈操作，弹出元素：" << elem << endl;
    stack.display();

    // 2. 测试括号匹配
    cout << "\n【实验 2：括号匹配检查】" << endl;
    cout << "------------------------" << endl;
    BracketChecker checker;

    string expr1 = "((a+b)*c)";
    string expr2 = "((a+b)*c))";
    string expr3 = "{[()]}";
    string expr4 = "{[(])}";

    cout << "表达式: " << expr1 << endl;
    cout << "检查结果: " << (checker.checkBrackets(expr1) ? "匹配" : "不匹配") <<
endl;

    cout << "\n 表达式: " << expr2 << endl;
    cout << "检查结果: " << (checker.checkBrackets(expr2) ? "匹配" : "不匹配") <<
endl;

    cout << "\n 表达式: " << expr3 << endl;
    cout << "检查结果: " << (checker.checkBrackets(expr3) ? "匹配" : "不匹配") <<
endl;

    cout << "\n 表达式: " << expr4 << endl;
    cout << "检查结果: " << (checker.checkBrackets(expr4) ? "匹配" : "不匹配") <<
endl;

    // 3. 测试表达式转换和计算
    cout << "\n【实验 3：表达式转换与计算】" << endl;
    cout << "------------------------" << endl;
    SimpleCalculator calc;

    // 测试用例 1：简单表达式
    string infix1 = "2+3*4";
    cout << "中缀表达式: " << infix1 << endl;
    string postfix1 = calc.infixToPostfix(infix1);
```

```
    int result1 = calc.evaluatePostfix(postfix1);
    cout << "计算结果: " << result1 << endl;
    cout << "预期结果: 14 (2+3*4 = 2+12 = 14)" << endl;

    cout << "\n------------------------" << endl;

    // 测试用例 2：带括号的表达式
    string infix2 = "(2+3)*4";
    cout << "中缀表达式: " << infix2 << endl;
    string postfix2 = calc.infixToPostfix(infix2);
    int result2 = calc.evaluatePostfix(postfix2);
    cout << "计算结果: " << result2 << endl;
    cout << "预期结果: 20 ((2+3)*4 = 5*4 = 20)" << endl;

    return 0;
}
```

实验二、数组和矩阵的操作

一、实验目的

1、掌握稀疏矩阵三元组的存储结构及其基本算法设计。

2、参照给定的程序样例，实现有关的操作。

二、实验内容及要求

提交实验报告，报告内容包括：目的、内容及要求、算法描述、程序结构、主要变量说明、程序清单、调试情况、设计技巧、心得体会。

1. 创建稀疏矩阵类，提供操作：两个稀疏矩阵相加、两个稀疏矩阵相乘、稀疏矩阵的转置、输出矩阵。矩阵元素的数据类型请使用 int。

2. 键盘输入矩阵的行数、列数；并按行优先顺序（先按行号升序排列，行号相同的元素再按列号升序排列）输入矩阵的各元素值，建立矩阵。

3. 对建立的矩阵执行相加、相乘、转置的操作，输出操作的结果矩阵。

4. 各操作需在稀疏矩阵上进行，充分考虑数据的稀疏性，不得直接或间接转换为二维数组形式计算，否则取消成绩。

三、操作要求

需要实现的相关操作，演示如下：

----------操作演示----------

1. 重置矩阵 P.
2. 矩阵乘法 P=P*Q.
3. 矩阵加法 P=P+Q.
4. 输出操作.
5. 转置操作 P=P^T.
6. 退出

选择功能(1~6):1

请输入矩阵 P 的行数 n、列数 m.两数中间用空格分开.

5 5

请输入矩阵 Q 中非零元素个数 t.

7

请依次输入矩阵中非零元素 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始.

1 1 2
1 5 3
2 3 4
3 1 4
3 2 1
4 4 7
5 2 2

矩阵 P 已重置.

选择功能(1~6):4
5 5
2 0 0 0 3
0 0 4 0 0
4 1 0 0 0
0 0 0 7 0
0 2 0 0 0

选择功能(1~6):2

为计算 P=P*Q，请输入矩阵 Q 的行数 n、列数 m.
5 5

请输入矩阵 Q 中非零元素个数 t.
4

请依次输入矩阵中非零元素 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始.

1 1 3
2 2 4
3 3 7
4 4 9

矩阵相乘已完成.

选择功能(1~6):4

5 5
6 0 0 0 0
0 0 28 0 0
12 4 0 0 0
0 0 0 63 0
0 8 0 0 0

选择功能(1~6):3

为计算 P=P+Q，请输入矩阵 Q 的行数 n、列数 m.
5 5

请输入矩阵 Q 中非零元素个数 t.
3

请依次输入矩阵中非零元素 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始.
2 3 12
4 5 11
5 5 7

矩阵相加已完成.

选择功能(1~6):4
5 5
6 0 0 0 0
0 0 40 0 0
12 4 0 0 0
0 0 0 63 11
0 8 0 0 7

选择功能(1~6):5

选择功能(1~6):4
5 5
6 0 12 0 0
0 0 4 0 8
0 40 0 0 0

0 0 0 63 0
0 0 0 11 7

选择功能(1~6):
----------操作演示----------

为方便操作描述，我们假设存在一个矩阵 P。

重置矩阵 P：

需要提示用户输入矩阵 P 的信息：
矩阵的行数 矩阵的列数
矩阵中非零元素个数 t
[t 行 矩阵中非零元素]
矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。

矩阵乘法 P=P*Q：

需要提示用户输入矩阵 Q 的信息：
矩阵的行数 矩阵的列数
矩阵中非零元素个数 t
[t 行 矩阵中非零元素]
矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。
设输入的矩阵为 Q，若 P×Q 运算合法，则将 P×Q 的结果矩阵赋给 P，若不合法，则将 Q 赋给 P，同时输出-1。

矩阵加法 P=P+Q：

需要提示用户输入矩阵 Q 的信息：
矩阵的行数 矩阵的列数
矩阵中非零元素个数 t
[t 行 表示矩阵中非零元素]
矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。
设输入的矩阵为 Q，若 P+Q 运算合法，则将 P+Q 的结果矩阵赋给 P，若不合法，则将 Q 赋给 P，同时输出-1。

输出操作：

设矩阵 P 的尺寸为 n 行 m 列，第一行输出矩阵 P 的行数和列数，随后 n 行按行优先顺序输出矩阵 P，每行 m 个数字，来表示当前的矩阵内容，每行数字之间用空格分隔。

转置操作 P=P^T：

设矩阵 P 的尺寸为 n 行 m 列，将其转置为 m 行 n 列的矩阵，无需输出。

四、算法思想

（1）创建算法

从输入获取矩阵的行数、列数和非零元素个数，然后依次输入每个非零元素的行索引、列索引和值，按照插入算法的逻辑将元素插入到三元组数组中。

（2）输出算法

遍历三元组数组，按照矩阵的格式输出非零元素，对于非零元素位置输出 0。

（3）相加算法

同时遍历两个稀疏矩阵的三元组数组，根据行索引和列索引判断元素是否对应。若对应，则将元素值相加，若结果不为 0 则存入结果矩阵的三元组数组；若不对应，则将较小索引对应的元素直接存入结果矩阵。

（4）相乘算法

对于矩阵 P 的每一个非零元素，遍历矩阵 Q 中相同列索引（即 Q 元素的列索引等于 P 元素的行索引）的非零元素，将它们的值相乘并累加到结果矩阵对应位置。

（6）转置算法

交换原矩阵三元组数组中元素的行索引和列索引，同时更新矩阵的行数和列数。为了保持三元组数组按行优先顺序存储，需要对交换后的数组进行重新排序。

五、关键代码示例

```
struct matrixterm {
    int row, col, value;
    void operator=(matrixterm& a) {//重载=
        row = a.row;
        col = a.col;
        value = a.value;
    }
};
class sparseMatrix {
private:
    int rows, cols;//行、列
    int num_value = 0;//非零元素的个数初始化为0
    matrixterm* element;//存非零元素的结构体数组
public:
    sparseMatrix(int n, int m) {
```

```
        rows = n;
        cols = m;
        element = new matrixterm[(n + 13) * (m + 13)];
    }
    void operator=(sparseMatrix& a) {//重载=，将一个矩阵赋给另一个矩阵
        rows = a.rows;
        cols = a.cols;
        num_value = a.num_value;
        for (int i = 0; i < num_value; i++) element[i] = a.element[i];
    }
    void input(int x, int y, int v);
    void reset(int n, int m);
    void multiply(sparseMatrix& b);
    void add(sparseMatrix& b);
    void output();
    void transpose();
    ~sparseMatrix() { delete[] element; }
};
void sparseMatrix::input(int x, int y, int v) {//输入 x 行序号，y 列序号,v 非零元素值
    element[num_value].row = x;
    element[num_value].col = y;
    element[num_value].value = v;
    num_value++;
}


void sparseMatrix::reset(int n, int m) {//重置矩阵
    rows = n;
    cols = m;
    delete[] element;//释放原来申请的空间
    element = new matrixterm[(n + 13) * (m + 13)];//重新分配内存
}


void sparseMatrix::multiply(sparseMatrix& b) {//乘法
    if (cols != b.rows) {
        *this = b;
        cout << -1 << endl;
    }
    else {
        sparseMatrix c(rows, b.cols);
        int* num_row_nz = new int[b.rows + 1];//b 第 i 行非零元素的个数
        int* row_nz_index = new int[b.rows + 1];//b 第 i 行第一个非零元素的索引(在 b.element
中的位置)
```

```
        int* result = new int[b.cols + 1];//乘后中每一行的结果
        for (int i = 1; i <= b.rows; i++) num_row_nz[i] = 0;//初始化 b 每行非零个数为 0
        for (int i = 0; i < b.num_value; i++) num_row_nz[b.element[i].row]++;//统计 b 每行非零元素的个数
        row_nz_index[1] = 0;//保证 b 第一行索引从第一个开始
        for (int i = 2; i <= b.rows; i++) row_nz_index[i] = row_nz_index[i - 1] + num_row_nz[i - 1];//统计 b 每行第一个非零元素的索引
        int id = 0;
        for (int i = 1; id < num_value; i++)//i 为结果的行序号，循环停止条件为 a 没有了非零元素了
        {
            for (int j = 1; j <= b.cols; j++) result[j] = 0;//初始化第 i 行每列的结果为 0
            while (element[id].row == i) {//该 while 循环求出第 i 行所有结果
                if (num_row_nz[element[id].col] != 0) {//a 列与 b 对应的所在行，非零元素个数不为 0
                    for (int x = row_nz_index[element[id].col]; x < row_nz_index[element[id].col] + num_row_nz[element[id].col]; x++)
                        result[b.element[x].col] += element[id].value * b.element[x].value;//该 a 列的值与 b 对应行的素有元素依次相乘，并加到结果与之对应的列中
                }
                id++;//b 该行乘完一遍，换下一个
            }
            for (int p = 1; p <= b.cols; p++) {//将该行的结果传给矩阵 c
                if (result[p] != 0) {
                    c.element[c.num_value].value = result[p];
                    c.element[c.num_value].row = i;
                    c.element[c.num_value].col = p;
                    c.num_value++;
                }
            }
        }
        *this = c;//求的结果在给 P(a)
        delete[] num_row_nz;
        delete[] row_nz_index;
        delete[] result;
    }
}
void sparseMatrix::add(sparseMatrix& b) {//加法
    if (cols != b.cols || rows != b.rows) {
        *this = b;
        cout << -1 << endl;
    }
    else {
```

```
        sparseMatrix c(rows, cols);
        int an = 0;
        int bn = 0;
        while (an < num_value && bn < b.num_value) {
            int aindex = (element[an].row - 1) * cols + element[an].col;//a 非零元素在整个矩阵中的索引
            int bindex = (b.element[bn].row - 1) * cols + b.element[bn].col;//b 非零元素在整个矩阵中的索引
            if (aindex < bindex) {//a 非零，b 为零
                c.element[c.num_value] = element[an];
                an++;
                c.num_value++;
            }
            else if (aindex == bindex) {//a,b 都为非零
                if (element[an].value + b.element[bn].value != 0) {//相加结果非零时
                    c.element[c.num_value].col = element[an].col;
                    c.element[c.num_value].row = element[an].row;
                    c.element[c.num_value].value = element[an].value + b.element[bn].value;
                    c.num_value++;
                }
                an++;
                bn++;
            }
            else {//a 为零，b 非零
                c.element[c.num_value] = b.element[bn];
                bn++;
                c.num_value++;
            }
        }
        //将剩余的非零元素赋给 c
        while (an != num_value) {
            c.element[c.num_value] = element[an];
            c.num_value++;
            an++;
        }
        while (bn != b.num_value) {
            c.element[c.num_value] = b.element[bn];
            c.num_value++;
            bn++;
        }
        *this = c;//将 c 给 P(a)
    }
}
```

```
void sparseMatrix::output() {//输出
    cout << rows << " " << cols << endl;
    int t = 0;
    for (int i = 1; i <= rows; i++) {
        for (int j = 1; j <= cols; j++) {
            if (t < num_value && element[t].row == i && element[t].col == j) {
                cout << element[t].value << " ";
                t++;
            }
            else {
                cout << 0 << " ";
            }
        }
        cout << endl;
    }
}
void sparseMatrix::transpose() {//转置
    sparseMatrix b(cols, rows);
    b.cols = rows;
    b.rows = cols;
    b.num_value = num_value;
    int* num_col_nz = new int[cols + 1];//矩阵每列的非零元素个数
    int* col_nz_index = new int[cols + 1];//矩阵每列第一个非零元素索引，因为 b 为行映射所以 a 要按列索引
    for (int i = 1; i <= cols; i++) num_col_nz[i] = 0;
    for (int i = 0; i < num_value; i++) num_col_nz[element[i].col]++;
    col_nz_index[1] = 0;
    for (int i = 2; i <= cols; i++) col_nz_index[i] = col_nz_index[i - 1] + num_col_nz[i - 1];
    for (int i = 0; i < num_value; i++) {
        int j = col_nz_index[element[i].col];//因为 a 为按行映射，将 a 中元素对应列的第一个非零数所在索引赋给 j
        b.element[j].col = element[i].row;
        b.element[j].row = element[i].col;
        b.element[j].value = element[i].value;
        col_nz_index[element[i].col]++;//该列的索引到下一个非零元素
    }
    *this = b;
}
```

实验三、树和二叉树的操作

一、实验目的

1、深入理解二叉树的基本概念和递归程序设计方法。

2、熟练掌握二叉树在二叉链表存储结构中的常用遍历方法：先序、中序、后序递归遍历，了解先序、中序和后序非递归遍历及层序遍历。

二、实验内容及要求

编写一个程序，用二叉树表示表达式，表达式只包含=、+、-、*、/、(、）和用字母表示的数且没有错误。例如“(a+b) * c - e/f =”表达式对应的二叉树如图 2.11.1 所示。

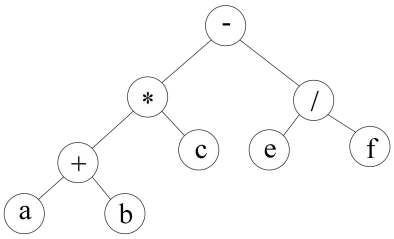


图 2.11.1 “(a+b) * c - e/f =”表达式对应的二叉树

三、算法思想

FigureExprByBiTree 类声明如下：

```
// 表达式的二叉树表示类
class FigureExprByBiTree
{
private:
// 辅助函数
    static bool IsOperator(char ch);    // 判断 ch 是否为运算符
    static int LeftPri(char op);        // 左边运算符的优先级
    static int RightPri(char op);       // 右边运算符的优先级

public:
// 接口方法声明
    FigureExprByBiTree() {};            // 无参数的构造函数
    virtual ~FigureExprByBiTree() {};   // 析构函数
```

```
    static void Run();                    // 将从键盘输入的中缀表达式用二叉树表示
};
```

函数 Run() 将从键盘中输入的中缀表达式转换为后缀表达式，用二叉树表示，并显示二叉树，需要考虑各操作符的优先级，为方便起见定义两个函数 LeftPri(ch) 和 RightPri(ch) 分别表示左边操作符与右边操作符的优先级，定义如表 2.11.1 所示。

表 2.11.1 定义函数优先级

操作符 ch	=	(	* 和 /	+ 和 -	)
LeftPri(ch)	0	1	5	3	6
RightPri(ch)	0	6	4	2	1

在表示中缀表达式时,可采用中缀表达式求值的算法思想,只是操作数栈(即下面算法思想中的二叉树栈 biTr)中用存储指向结点的指针来代替存储操作数。具体算法思想如下:

(1) 定义二叉树栈 biTr(即操作数栈),操作符栈 optr,将结束符‘=’进 optr 栈,然后读入中缀表达式字符流的首字符 ch。

(2) 重复执行以下步骤,直到 ch 等于‘=’,同时栈顶的操作符也是‘=’,停止循环。

若 ch 是操作数,则用 ch 生成只含一个结点的二叉树,并入 biTr 栈,读入下一个字符 ch。

若 ch 是操作符,判断 ch 的优先级 RightPri(ch)和位于栈顶的操作符 optrTop 的优先级 LeftPri(optrTop):

若 LeftPri(optrTop)小于 RightPri(ch),令 ch 进栈 optr,读入下一个字符 ch。

若 LeftPri(optrTop)大于 RightPri(ch),从 biTr 中退出 rightT 与 leftT,从 optr 中退出 theta,由 theta,rightT,leftT 生成新二叉树,并入 biTr 栈。

若 LeftPri(optrTop)等于 RightPri(ch),若退出的是 ch= ')' ,则退栈,读入下一个字符 ch。

(3) 取出 biTr 栈顶 t,输出相应二叉树。

四、源代码示例

1. 建立项目 figure_expr_by_bitree。

2. 将链栈与二叉树需要的头文件 node.h, lk_stack.h, bin_tree_node.h, binary_tree.h 和 lk_queue.h 复制到 figure_expr_by_bitree 文件夹中, 并将其加入项目中。

3. 建立头文件 figure_expr_by_bitree.h, 声明并实现表达式的二叉树表示类 FigureExprByBiTree。具体内容如下:

```
// 文件路径名：figure_expr_by_bitree\figure_expr_by_bitree.h
#ifndef   __FIGURE_EXPR_BY_BI_TREE_H__
#define   __FIGURE_EXPR_BY_BI_TREE_H__

#include <iostream>                    // 编译预处理命令
#include <cstdlib>    // 含 C 函数 exit() 的声明(stdlib.h 与 cstdlib 是 C 的头文件)
```

```
using namespace std;                    // 使用命名空间 std
#include "lk_stack.h"                   // 链栈
#include "bin_tree_node.h"              // 二叉树结点类
#include "binary_tree.h"                // 二叉树类

// 表达式的二叉树表示类
class FigureExprByBiTree
{
private:
// 辅助函数
    static bool IsOperator(char ch);            // 判断 ch 是否为运算符
    static int LeftPri(char op);                // 左边运算符的优先级
    static int RightPri(char op);               // 右边运算符的优先级

public:
// 接口方法声明
    FigureExprByBiTree() {};                    // 无参数的构造函数
    virtual ~FigureExprByBiTree() {};           // 析构函数
    static void Run();                          // 将从键盘输入的中缀表达式用二叉树表示

};

// 表达式的二叉树表示类的实现部分
bool FigureExprByBiTree:: IsOperator(char ch)
// 操作结果: 如果 ch 是运算符, 则返回 true, 否则返回 false
{
    if (ch == '=' || ch == '*' || ch == '/' || ch == '+' || ch == '-' || ch == '(' || ch == ')')
return true;
    else return false;
}

int FigureExprByBiTree:: LeftPri(char op)
// 操作结果: 左边运算符的优先级
{
    int result;                             // 优先级
    if (op == '=') result = 0;
    else if (op == '(') result = 1;
    else if (op == '*' || op == '/') result = 5;
    else if (op == '+' || op == '-') result = 3;
    else if (op == ')') result = 6;
    return result;                          // 返回优先级
}

int FigureExprByBiTree:: RightPri(char op)
// 操作结果: 右边运算符的优先级
{
```

```
    int result;                                  // 优先级
    if (op == '=') result = 0;
    else if (op == '(') result = 6;
    else if (op == '*' || op == '/') result = 4;
    else if (op == '+' || op == '-') result = 2;
    else if (op == ')') result = 1;
    return result;                               // 返回优先级
}

Void FigureExprByBiTree:: Run()
// 操作结果: 将从键盘中输入的中缀表达式转换为后缀表达式,用二叉树表示,并显示二叉树
{
    LinkStack<BinTreeNode<char>*> biTr;          // 二叉树栈
    LinkStack<char> optr;                        // 操作符栈
    char ch, optrTop, theta;   //输入的字符 ch, 操作符栈 optr 栈顶操作符, 操作符 theta
    BinTreeNode<char>* r;      // 为编程方便起见, 使 optr 的栈底用^表示
    optr.Push('=');                              // 取出操作符栈 optr 的栈顶
    optr.Top(optrTop);                           // 从输入流 cin 中取出一个字符
    cin >> ch;
    while (optrTop != '=' || ch != '=')
    {   // 当 optrTop 等于'=' 且 ch 等于'=' 不成立时, 表达式运算未结束
        if (!IsOperator(ch))
        {   // ch 为操作数
            r = new BinTreeNode<char>(ch);       // 生成只含一个结点的二叉树
            biTr.Push(r);                        // r 进 optr 栈
            cin >> ch;                           // 从输入流 cin 中取出一个字符
        }
        else
        {   // ch 为操作符
            if (LeftPri(optrTop) < RightPri(ch))
            {   // ch 优先级更高
                optr.Push(ch);                   // ch 进 optr 栈
                cin >> ch;                       // 从输入流 cin 中取出一个字符
            }
            else if (LeftPri(optrTop) > RightPri(ch))
            {   // optrTop 优先级更高
                BinTreeNode<char>* leftT, * rightT;   // 二叉树

                if (!biTr.Pop(rightT))
                {   // 出现异常
                    cout << "表达式有错!" << endl;    // 提示信息
                    exit(1);                         // 退出程序
                }
                if (!biTr.Pop(leftT))
                {   // 出现异常
```

```
                cout << "表达式有错!" << endl;        // 提示信息
                exit(2);                              // 退出程序
            }

            optr.Pop(theta);                          // 从 optr 栈退出 theta
            r = new BinTreeNode<char>(theta, leftT, rightT);   // 生成新二叉树
            biTr.Push(r);                             // r 进 biTr 栈
        }
        else if (LeftPri(optrTop) == RightPri(ch) && ch == ')')
        {   // 表示 optrTop 等于'（'与等于'）'
            optr.Pop(ch);                             // 从 optr 栈退出栈顶的'('
            cin >> ch;                                // 从输入流 cin 中取出一个字符
        }
    }
    optr.Top(optrTop);                                // 取出操作符栈 optr 的栈顶
}
biTr.Pop(r);                                          // 取生成的二叉树的根
BinaryTree<char> bt(r);                               // 生成二叉树
DisplayBTWithTreeShape(bt);                           // 显示二叉树
}

#endif
```

4. 建立源程序文件 main.cpp, 实现 main() 函数, 具体代码如下:

```
// 文件路径名：figure_expr_by_bitree\main.cpp
#include <iostream>                 // 编译预处理命令
#include <cstdlib> //   含 C 函数 system() 的声明(stdlib.h 与 cstdlib 是 C 的头文件)
#include <cctype>   //含 C 函数 tolower() 的声明(ctype.h 与 cctype 是 C 的头文件)
using namespace std;                // 使用命名空间 std
#include "figure_expr_by_bitree.h"     // 表达式求值类的头文件

int main()
{
    char select;                                      // 接收用户是否继续的回答
    do
    {
        cout << "输入表达式: " << endl;
        FigureExprByBiTree::Run();                    // 用二叉树表示从键盘输入的
中缀表达式
        cout << "是否继续(y/n)?";
        cin >> select;                                // 输入用户的选择
        select = tolower(select);                     // 大写字母转换为小写字母
        while (select != 'y' && select != 'n')
        {   // 输入有错
            cout << "输入有错,请重新输入(y/n): ";
```

```
            cin >> select;                          // 输入用户的选择
            select = tolower(select);               // 大写字母转换为小写字母
        }
    } while (select == 'y');

    system("PAUSE");                                // 调用库函数 system()
    return 0;                                       // 返回值 0, 返回操作系统
}
```

5. 编译及运行程序。

实验四、图的操作

一、实验目的

参照给定的图类和广义表类的程序样例，验证给出的图的常见算法，并实现有关的操作。

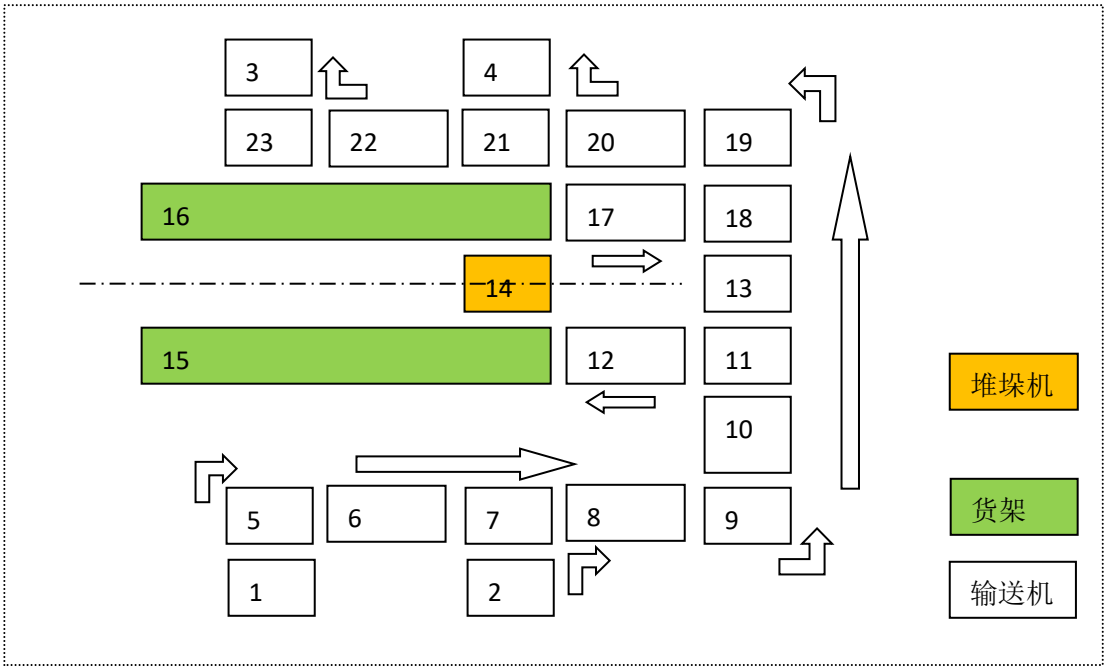
二、实验要求

1、掌握图的特点。掌握它们的常见算法；

2、提交实验报告，报告内容包括：目的、要求、算法描述、程序结构、主要变量说明、程序清单、调试情况、设计技巧、心得体会。

三、实验内容

图是一个适应性很强的数据结构，能对应现实中很多的多对多关系的系统。例如物流中的地面输送系统。下面是一个地面输送系统的示意图，{1,2}为入库站台，{3,4}为出库站台：



结点：1~23（整形）

弧：

<1,5>,<5,6>,<6,7>,<2,7>,<7,8>,<8,9>,<9,10>,<10,11>,<11,12>,<12,14>,<14,15>,<15,14>,<14,16>,<16,14>,<14,17>,<11,13>,<13,18>,<17,18>,<18,19>,<19,20>,<20,21>,<21,4>,<21,22>,<22,23>,<23,3>

稀疏图，以邻接表的形式建立图结构。

要求：

1. 以数据文件的形式，输入数据和保存数据。

2. 能够求任意设备之间的任务路径。例如：入库 1-15，得到 <1,5,6,7,8,9,10,11,12,14,15>

四、源代码示例

```
//DirectedGraph.h
#include <iostream>
#include "SqList.h"
#include "LinkList.h"

void print_int(int p)//用于遍历输出
{
    std::cout << p;
}

template <typename ElemType>
class Vertex {
public:
    ElemType data;
    LinkList<int> adjLink; // 出边邻接表
};
template <typename ElemType>
class DirectedGraph {
private:
    SqList<Vertex<ElemType>> vexTable;
    int vexNum = 0;   // 顶点数
    int edgeNum = 0; // 边数
public:
    // 添加顶点
    void addVertex(const ElemType& data) {
        vexTable.AddTail(data);
        vexNum++;
    }
    // 添加有向边 v→w
    void addEdge(int v, int w) {
        if (v >= 0 && v < vexNum && w >= 0 && w < vexNum) {
            vexTable[v].adjLink.Insert(w);
            edgeNum++;
```

```
            }
        }
        // 获取顶点数
        int getVexNum() const { return vexNum; }
        // 获取边数
        int getEdgeNum() const { return edgeNum; }
        // 打印邻接表
        void print() const {
            for (int i = 0; i < vexNum; ++i) {
                std::cout << "顶点" << i << "(" << vexTable[i].data << ")的出边: ";
                adjLink.Traverse(print_int);
                std::cout << std::endl;
            }
        }
};
//main.cpp
#include "directed_graph.h"
int main() {
    DirectedGraph<std::string> g;
    g.addVertex("1");
    g.addVertex("2");
    g.addVertex("3");
    g.addVertex("4");
    ......
    g.addEdge(0, 4); // 1→5
    g.addEdge(4, 5); // 5→6
    g.addEdge(5, 6); // 6→7
    g.addEdge(1, 6); // 2→7
    g.addEdge(6, 7); // 7→8
    ......
    std::cout << "顶点数: " << g.getVexNum() << std::endl;
    std::cout << "边数: " << g.getEdgeNum() << std::endl;
    g.print();
    return 0;
}
```

自行添加有关图的深度或广度搜索有关的内容。

实验五、查找

一、实验目的

1、掌握顺序查找、二分查找的基本思想和算法实现。
2、理解不同查找算法的适用场景和性能差异。
3、掌握哈希表的基本概念和简单哈希查找的实现。
4、学会分析和比较不同查找算法的时间复杂度。
5、能够根据实际问题选择合适的查找方法。

二、实验内容及要求

设计并实现一个学生成绩管理系统，该系统需要管理学生的基本信息（学号、姓名、成绩），并提供多种查找功能：

基本功能要求

（1）学生信息管理
- 添加学生信息
- 显示所有学生信息
- 按成绩排序（为二分查找做准备）

（2）查找功能实现
- 顺序查找：按学号查找学生
- 二分查找：按成绩查找学生（需要先排序）
- 哈希查找：基于学号的快速查找（简单哈希表）

（3）性能比较
- 记录并比较不同查找方法的比较次数
- 分析不同数据量下的查找效率

三、算法思想

1、顺序查找（线性查找）

基本思想：

从第一个元素开始，逐个比较，直到找到目标元素或遍历完所有元素。

特点：

- 适用于：无序数据、数据量小
- 时间复杂度：O(n)

- 优点：简单，不要求数据有序
- 缺点：效率低，数据量大时很慢

算法步骤：

1. 从第一个元素开始
2. 逐个比较是否等于目标值
3. 找到则返回位置
4. 遍历完都没找到则返回-1

2、二分查找（折半查找）

基本思想：

在有序数组中，每次比较中间元素，根据比较结果缩小查找范围一半。

特点：

- 适用于：有序数据
- 时间复杂度：O(log n)
- 优点：效率高，适合大数据量
- 缺点：要求数据必须有序

算法步骤：

1. 设置查找范围：low=0, high=n-1
2. 计算中间位置：mid=(low+high)/2
3. 比较中间元素与目标值：
    - 相等：找到，返回位置
    - 目标值较小：在左半部分继续查找，high=mid-1
    - 目标值较大：在右半部分继续查找，low=mid+1
4. 重复步骤 2-3，直到找到或 low>high

3、哈希查找

基本思想：通过哈希函数将关键字映射到存储位置，实现快速查找。

特点：

- 时间复杂度：理想情况 O(1)
- 优点：查找速度极快
- 缺点：需要额外空间，可能有冲突

简单哈希函数：

位置 = 学号 % 表长

四、源代码示例

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iomanip>
using namespace std;

// 学生信息结构体
struct Student {
    int id;                 // 学号
    string name;            // 姓名
    double score;           // 成绩

    // 构造函数
    Student() : id(0), name(""), score(0.0) {}
    Student(int i, string n, double s) : id(i), name(n), score(s) {}

    // 显示学生信息
    void display() const {
        cout << "学号: " << setw(8) << id
             << " | 姓名: " << setw(10) << name
             << " | 成绩: " << setw(6) << fixed << setprecision(1) << score <<
endl;
    }
};

// 学生管理系统类
class StudentManager {
private:
    vector<Student> students;           // 存储所有学生
    vector<Student> sortedByScore;      // 按成绩排序的学生列表
    int compareCount;                   // 记录比较次数

public:
```

```
// 添加学生
void addStudent(int id, string name, double score) {
    students.push_back(Student(id, name, score));
    cout << "添加成功：" << name << endl;
}

// 显示所有学生
void displayAll() {
    if (students.empty()) {
        cout << "暂无学生信息！" << endl;
        return;
    }

    cout << "\n===== 所有学生信息 =====" << endl;
    cout << "------------------------" << endl;
    for (int i = 0; i < students.size(); i++) {
        cout << i+1 << ". ";
        students[i].display();
    }
    cout << "------------------------" << endl;
    cout << "总人数: " << students.size() << endl;
}

// 按成绩排序（为二分查找做准备）
void sortByScore() {
    sortedByScore = students;   // 复制到新数组

    // 使用冒泡排序（教学目的，让过程更清晰）
    for (int i = 0; i < sortedByScore.size() - 1; i++) {
        for (int j = 0; j < sortedByScore.size() - 1 - i; j++) {
            if (sortedByScore[j].score > sortedByScore[j+1].score) {
                swap(sortedByScore[j], sortedByScore[j+1]);
            }
        }
    }

    cout << "\n===== 按成绩排序结果 =====" << endl;
    for (int i = 0; i < sortedByScore.size(); i++) {
```

```
            cout << i+1 << ". ";
            sortedByScore[i].display();
        }
    }

    // 1. 顺序查找（按学号）
    int sequentialSearch(int targetId) {
        compareCount = 0;

        cout << "\n【顺序查找过程】" << endl;
        cout << "目标学号: " << targetId << endl;

        for (int i = 0; i < students.size(); i++) {
            compareCount++;
            cout << "第" << compareCount << "次比较: "
                 << students[i].id << " vs " << targetId;

            if (students[i].id == targetId) {
                cout << " -> 找到了！" << endl;
                return i;  // 返回位置
            }
            cout << " -> 不匹配，继续" << endl;
        }

        cout << "查找结束，未找到该学生" << endl;
        return -1;  // 未找到
    }

    // 2. 二分查找（按成绩）
    int binarySearch(double targetScore) {
        if (sortedByScore.empty()) {
            cout << "请先执行按成绩排序！" << endl;
            return -1;
        }

        compareCount = 0;
        int low = 0;
        int high = sortedByScore.size() - 1;
```

```
        cout << "\n【二分查找过程】" << endl;
        cout << "目标成绩: " << targetScore << endl;
        cout << "查找范围: [0, " << high << "]" << endl;

        while (low <= high) {
            int mid = (low + high) / 2;
            compareCount++;

            cout << "\n 第" << compareCount << "次比较:" << endl;
            cout << "   范围[" << low << ", " << high << "], 中间位置: " << mid << endl;
            cout << "   比较: " << sortedByScore[mid].score << " vs " << targetScore;

            if (sortedByScore[mid].score == targetScore) {
                cout << " -> 找到了! " << endl;
                return mid;
            }
            else if (sortedByScore[mid].score < targetScore) {
                cout << " -> 目标在右半部分" << endl;
                low = mid + 1;
            }
            else {
                cout << " -> 目标在左半部分" << endl;
                high = mid - 1;
            }
        }

        cout << "\n 查找结束，未找到该成绩的学生" << endl;
        return -1;
    }

    // 3. 查找结果处理
    void handleSearchResult(int index, const string& method) {
        cout << "\n===== " << method << "查找结果 =====" << endl;

        if (index != -1) {
```

```
            cout << "查找成功！" << endl;
            if (method == "顺序" || method == "哈希") {
                cout << "找到学生信息：" << endl;
                students[index].display();
            } else if (method == "二分") {
                cout << "找到学生信息：" << endl;
                sortedByScore[index].display();
            }
        } else {
            cout << "查找失败，未找到符合条件的学生" << endl;
        }

        cout << "比较次数: " << compareCount << " 次" << endl;
    }

    // 获取比较次数
    int getCompareCount() {
        return compareCount;
    }
};

// 简单哈希表实现
class SimpleHashTable {
private:
    static const int TABLE_SIZE = 13;  // 哈希表大小（选择质数）
    Student* table[TABLE_SIZE];              // 哈希表数组
    int compareCount;

    // 哈希函数：除留余数法
    int hashFunction(int key) {
        return key % TABLE_SIZE;
    }

public:
    // 构造函数
    SimpleHashTable() {
        for (int i = 0; i < TABLE_SIZE; i++) {
            table[i] = nullptr;
```

```
        }
        compareCount = 0;
    }

    // 析构函数
    ~SimpleHashTable() {
        for (int i = 0; i < TABLE_SIZE; i++) {
            if (table[i] != nullptr) {
                delete table[i];
            }
        }
    }

    // 插入学生
    void insert(const Student& student) {
        int index = hashFunction(student.id);
        int originalIndex = index;

        cout << "插入学号 " << student.id << "，哈希值: " << index;

        // 线性探测解决冲突
        while (table[index] != nullptr) {
            cout << " -> 位置" << index << "已占用";
            index = (index + 1) % TABLE_SIZE;

            // 如果回到原位置，表示哈希表已满
            if (index == originalIndex) {
                cout << " -> 哈希表已满！" << endl;
                return;
            }
        }

        table[index] = new Student(student);
        cout << " -> 存储在位置: " << index << endl;
    }

    // 哈希查找
    Student* search(int id) {
```

```
    compareCount = 0;
    int index = hashFunction(id);
    int originalIndex = index;

    cout << "\n【哈希查找过程】" << endl;
    cout << "目标学号: " << id << "，哈希值: " << index << endl;

    while (table[index] != nullptr) {
        compareCount++;
        cout << "第" << compareCount << "次比较: 位置" << index
             << "的学号" << table[index]->id;

        if (table[index]->id == id) {
            cout << " -> 找到了！" << endl;
            return table[index];
        }

        cout << " -> 不匹配，继续探测" << endl;
        index = (index + 1) % TABLE_SIZE;

        // 如果回到原位置，说明没找到
        if (index == originalIndex) {
            break;
        }
    }

    cout << "查找结束，未找到" << endl;
    return nullptr;
}

// 显示哈希表
void display() {
    cout << "\n===== 哈希表内容 =====" << endl;
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "位置[" << setw(2) << i << "]: ";
        if (table[i] != nullptr) {
            cout << "学号=" << table[i]->id
                 << "，姓名=" << table[i]->name;
```

```
            } else {
                cout << "空";
            }
            cout << endl;
        }
    }

    int getCompareCount() {
        return compareCount;
    }
};

// 性能测试函数
void performanceTest() {
    cout << "\n========== 查找性能对比测试 ==========" << endl;

    StudentManager mgr;

    // 生成测试数据
    int sizes[] = {10, 50, 100};

    for (int size : sizes) {
        cout << "\n 数据量: " << size << " 个学生" << endl;
        cout << "--------------------" << endl;

        // 清空并重新生成数据
        mgr = StudentManager();
        for (int i = 0; i < size; i++) {
            mgr.addStudent(1000 + i, "Student" + to_string(i), 60 + rand() % 40);
        }

        // 查找中间位置的元素
        int targetId = 1000 + size/2;

        // 顺序查找
        mgr.sequentialSearch(targetId);
        int seqCompare = mgr.getCompareCount();
```

```
        // 二分查找（先排序）
        mgr.sortByScore();
        mgr.binarySearch(80.0);   // 查找成绩 80 分
        int binCompare = mgr.getCompareCount();

        cout << "\n 性能比较： " << endl;
        cout << "顺序查找比较次数: " << seqCompare << endl;
        cout << "二分查找比较次数: " << binCompare << endl;
        cout << "效率提升: " << (seqCompare > binCompare ? "二分查找更快" : "顺
序查找更快") << endl;
    }
}

// 主函数
int main() {
    cout << "========== 查找算法实验 ==========" << endl;

    StudentManager manager;
    SimpleHashTable hashTable;

    // 初始化测试数据
    cout << "\n【初始化测试数据】" << endl;
    manager.addStudent(1001, "张三", 85.5);
    manager.addStudent(1003, "李四", 72.0);
    manager.addStudent(1005, "王五", 90.0);
    manager.addStudent(1002, "赵六", 68.5);
    manager.addStudent(1004, "钱七", 85.5);
    manager.addStudent(1006, "孙八", 78.0);

    // 同时插入哈希表
    cout << "\n【构建哈希表】" << endl;
    hashTable.insert(Student(1001, "张三", 85.5));
    hashTable.insert(Student(1003, "李四", 72.0));
    hashTable.insert(Student(1005, "王五", 90.0));
    hashTable.insert(Student(1002, "赵六", 68.5));
    hashTable.insert(Student(1004, "钱七", 85.5));
    hashTable.insert(Student(1006, "孙八", 78.0));
```

```
int choice;
do {
    cout << "\n========== 功能菜单 ==========" << endl;
    cout << "1. 显示所有学生信息" << endl;
    cout << "2. 顺序查找（按学号）" << endl;
    cout << "3. 二分查找（按成绩）" << endl;
    cout << "4. 哈希查找（按学号）" << endl;
    cout << "5. 按成绩排序" << endl;
    cout << "6. 显示哈希表" << endl;
    cout << "7. 性能对比测试" << endl;
    cout << "0. 退出" << endl;
    cout << "请选择: ";
    cin >> choice;

    switch(choice) {
        case 1:
            manager.displayAll();
            break;

        case 2: {
            int id;
            cout << "请输入要查找的学号: ";
            cin >> id;
            int index = manager.sequentialSearch(id);
            manager.handleSearchResult(index, "顺序");
            break;
        }

        case 3: {
            double score;
            cout << "请先确保已按成绩排序！" << endl;
            cout << "请输入要查找的成绩: ";
            cin >> score;
            int index = manager.binarySearch(score);
            manager.handleSearchResult(index, "二分");
            break;
        }
```

```
            case 4: {
                int id;
                cout << "请输入要查找的学号: ";
                cin >> id;
                Student* student = hashTable.search(id);

                cout << "\n===== 哈希查找结果 =====" << endl;
                if (student != nullptr) {
                    cout << "查找成功！" << endl;
                    student->display();
                } else {
                    cout << "查找失败！" << endl;
                }
                cout << "比较次数: " << hashTable.getCompareCount() << " 次"
<< endl;
                break;
            }
            case 5:
                manager.sortByScore();
                break;

            case 6:
                hashTable.display();
                break;

            case 7:
                performanceTest();
                break;

            case 0:
                cout << "退出程序" << endl;
                break;

            default:
                cout << "无效选择！" << endl;
        }
    } while (choice != 0);
```

```
    return 0;
}
```

五、实验要求

1. 基本要求
   - 实现三种查找算法（顺序查找、二分查找、哈希查找）
   - 能够正确处理查找成功和失败的情况
   - 记录并显示每次查找的比较次数
2. 扩展要求
   - 实现查找算法的性能比较
   - 处理哈希冲突
   - 实现批量数据的查找测试

六、测试用例

测试用例 1：顺序查找
- 输入：学号 1003
- 预期：找到李四，比较次数 2 次

测试用例 2：二分查找
- 输入：成绩 85.5
- 预期：找到对应学生，比较次数≤$\log_2 n$ 次

测试用例 3：哈希查找
- 输入：学号 1005
- 预期：找到王五，比较次数 1-2 次

七、思考题

1. 在什么情况下顺序查找比二分查找更合适？
2. 哈希查找的平均时间复杂度是 O(1)，为什么有时候比较次数不止 1 次？
3. 如果要查找的是字符串类型的姓名，应该使用哪种查找方法？为什么？
4. 如何设计一个好的哈希函数？哈希冲突有哪些解决方法？

实验六、排序

一、实验目的

1、掌握堆排序算法；
2、掌握归并排序算法；

二、实验内容及要求

1、堆排序算法要求：

（1）编写函数模板 InputData，输入参数为数组名 elem 和元素个数 n，读入需要排序的数据，并将数据保存到数组中；

（2）编写函数模板 SiftAdjust，输入参数为数组名 elem 和数组的起始下标 low 和终止下标 high，输入数据 elem[low...high]中的记录除 elem[low]以外都满足大顶堆定义；将数组调整为以 elem[low]为顶的大顶堆；

（3）编写函数模板 HeapSort，输入参数为数组名 elem 和元素个数 n；首先，循环调用 SiftAdjust 将 elem 调整为大顶堆；再逐一将大顶和对应的元素互换和调用 SiftAdjust 构建大顶堆，完成对 elem 的排序；

（4）编写函数模板 OutputData，输入参数为数组名 elem 和元素个数 n，向标准输出设备输出数组中的数据；

（5）编写主函数，在主函数中读入需要排序的元素个数 n，建立长度为 n，类型为 int 的动态数组 elem，调用 InputData 函数读入待排序数据，调用 HeapSort 实现对 elem 的排序，调用 OutputData 输出 elem 中的数据，回收 elem 的空间；

2、归并排序算法要求：

（1）编写函数模板 InputData，输入参数为数组名 elem 和元素个数 n，读入需要排序的数据，并将数据保存到数组中；

（2）编写函数模板 SimpleMerge，输入参数为数组名 elem 和数组的起始下标 low、中间下标 mod、终止下标 high，输入数组中 elem[low...mid]和 elem[mid+1...high]的记录都是已排好序的记录；将数组 elem[low...mid]和 elem[mid+1...high]进行归并排序；

（3）编写函数模板 SimpleMergeSort，输入参数为数组名 elem 和数组的起始下标 low 和终止下标 high；采用递归调用的方式调用 SimpleMergeSort 和 SimpleMerge，完成对 elem 的排序；

（4）编写函数模板 OutputData，输入参数为数组名 elem 和元素个数 n，向标准输出设备输出数组中的数据；

（5）编写主函数，在主函数中读入需要排序的元素个数 n，建立长度为 n，类型为 int 的动态数组 elem，调用 InputData 函数读入待排序数据，调用 SimpleMergeSort 实现对 elem 的排序，调用 OutputData 输出 elem 中的数据，回收 elem 的空间；

三、算法思想

1、堆排序算法思想：

（1）将存于数组中的序列看作是一棵完全二叉树的顺序存储；

（2）按照堆的概念对数组进行调整，使数组成为一个大顶堆；

（3）摘取大顶，将其换到数组的最后位置；

（4）对不含最后一个元素的新数组进行调整，使数组成为一个大顶堆，得到次大元素，将次大元素换到新数组的最后位置，直到数组中元素全部有序；

2、归并排序算法思想：

通过归并两个、三个或多个有序子序列，逐步增加有序序列长度的方式实现排序。通常采用的是 2 路归并排序。即：将两个位置相邻的有序子序列归并为一个有序序列。

（1）将每个元素都看作是 1 个有序序列，对每两个序列进行归并；

（2）为归并后的序列新开空间，其长度为两个序列长度之和；

（3）将两个序列中的第 0 个元素标记为当前需要比较的两个元素；

（4）取出序列 1 和序列 2 所标记的元素，将较大元素加入到新序列中，并将较大元素所在序列的标记后移，如较大元素已是该序列的最后 1 个元素，则把另 1 个序列的所剩元素都依次转移到新序列中，完成两个序列的归并排序，否则重复该步骤 d)；

（5）完成所有序列的归并后，再对所生成的新序列进行两两归并，直到只剩一个序列；完成整个序列的排序；

四、源代码示例

1、堆排序算法关键代码示例

```
template <class ElemType>
void SiftAdjust(ElemType elem[], int low, int high)
//  操作结果:elem[low .. high]中记录除 elem[low]以外都满足
//  堆定义,调整 elem[low]使其 elem[low .. high]成为一个大顶堆
{
    for (int f = low, i = 2 * low + 1; i <= high; i = 2 * i + 1)
    {      // f 为被调整结点，i 为 f 的最大孩子
           if (i < high && elem[i] < elem[i + 1])
        {     //  右孩子更大，i 指向右孩子
```

```
            i++;
        }
        if (elem[f] >= elem[i])
        {    // 已成为大顶堆
            break;
        }
        ElemType temp = elem[f]; elem[f] = elem[i]; elem[i] = temp;
        // 交换 elem[f]与 elem[i]
        f = i;      // 成为新的调整结点
    }
}

template <class ElemType>
void HeapSort(ElemType elem[], int n)
// 操作结果:对数组 elem 进行堆排序
{
    int i;
    for (i = (n - 2) / 2; i >= 0; --i)
    {    // 将 elem[0 .. n - 1]调整成大顶堆
        SiftAdjust(elem, i, n - 1);
    }
    for (i = n - 1; i > 0; --i)
    {    // 第 n - i 趟堆排序
        ElemType temp = elem[0];
            elem[0] = elem[i];
            elem[i] = temp;
        // 将堆顶元素和当前未经排序的子序列
            //elem[0 .. i]中最后一个元素交换
        SiftAdjust(elem, 0, i - 1);
            // 将 elem[0 .. i - 1]重新调整为大顶堆
    }
}
```

2、归并排序算法关键代码示例

```
template <class ElemType>
void Merge(ElemType elem[], int low, int mid, int high)
// 操作结果:将有序子序列 elem[low .. mid]和 elem[mid + 1 .. midhigh]归并为新的有序
序列 elem[low .. high]
```

```
{
    ElemType * temElem = new ElemType[high+1];
    int i, j, k;            // 临时变量
    for (i = low, j = mid + 1, k = low; i <= mid && j <= high; k++)
    {   //i 为归并时 elem[low..mid]当前元素的下标，j 为归并时 elem[mid+1..high]
        // 当前元素的下标，k 为 temElem 中当前元素的下标
        if (elem[i] <= elem[j])
        {   // elem[i]较小,先归并
            temElem[k] = elem[i];
            i++;
        }
        else
        {   // elem[j]较小,先归并
            temElem[k] = elem[j];
            j++;
        }
    }
    for (; i <= mid; i++, k++)
    {   // 归并 elem[low .. mid]中剩余元素
        temElem[k] = elem[i];
    }
    for (; j <= high; j++, k++)
    {   // 归并 elem[mid + 1 .. high]中剩余元素
        temElem[k] = elem[j];
    }

    for (i = low; i <= high; i++)
    {   // 将 temElem[low .. high]复制到 elem[low .. high]
        elem[i] = temElem[i];
    }
    delete []temElem;
}

template <class ElemType>
void MergeSortHelp(ElemType elem[], int low, int high)
// 操作结果:对序列 elem[low .. high]拆分，进行归并排序
{
    if (low < high)
```

```
    {
        // 将 elem[low .. high]平分为 elem[low .. mid]和 elem[mid + 1 .. high]
        int mid = (low + high) / 2;
        // 对 elem[low .. mid]进行归并排序
        MergeSortHelp(elem, low, mid);
        // 对 elem[mid + 1 .. high]进行归并排序
        MergeSortHelp(elem, mid + 1, high);
        // 对 elem[low .. mid]和 elem[mid + 1 .. high]进行归并
        Merge(elem, low, mid, high);
    }
}
```