**作业 HW5 实验报告**

日期：2024年12月18日

# 实验报告格式要求按照模板（使用Markdown等也请保证报告内包含模板中的要素）

# 对字体大小、缩进、颜色等不做强制要求（但尽量代码部分和文字内容有一定区分，可参考vscode配色）

# 实验报告要求在文字简洁的同时将内容表示清楚

# 报告内不要大段贴代码，尽量控制在20页以内

1. **涉及数据结构和相关背景**

* 顺序查找：最简单的查找方式，从头到尾逐个比较。
* 二分查找：针对有序数据，每次将范围缩小一半的查找方法。
* 二叉排序树：一种树形结构，左小右大的特点使查找更高效。
* 二叉平衡树：在二叉排序树基础上保持平衡，防止性能退化。
* 哈希表：通过映射关系实现快速查找的数据结构。

**2. 实验内容**

**2.1 和有限的最长子序列**

**2.1.1 问题描述**

给定一个长度为n的整数数组nums和一个长度为m的整数数组queries，需要找出nums中元素之和小于等于queries[i]的子序列的最大长度，返回一个长度为m的数组answer作为结果。其中子序列可以通过删除原数组中的某些元素（也可以不删除）得到。

**2.1.2 基本要求**

 输入格式：

* 第一行：两个整数n和m，表示数组长度
* 第二行：n个整数，表示nums数组元素
* 第三行：m个整数，表示queries数组元素

 数据范围：

* 20%数据：1≤n,m≤10
* 40%数据：1≤n,m≤100
* 100%数据：1≤n,m≤1000
* 所有数据：1≤nums[i],queries[i]≤106

 输出格式： 一行m个整数，表示answer结果

**2.1.3 数据结构设计**

主要使用vector数组存储数据：

vector<int> nums(n);    // 存储输入的原始数组

vector<int> queries(m); // 存储查询数组

vector<int> ad(n);      // 存储前缀和数组

**2.1.4功能说明（函数、类）**

1.排序函数mysort：使用归并排序对数组进行排序

static void mysort(vector<int> &a, int start, int end)

{

    // 递归终止条件

    if (end - start <= 1)

        return;

    // 分治排序

    int mid = (start + end) / 2;

    mysort(a, start, mid); // 左半部分排序

    mysort(a, mid, end);   // 右半部分排序

    // 合并两个有序数组

    // ...合并过程代码...

}

1. 主要处理流程：

// 1. 对原数组进行排序

mysort(nums, 0, n);

// 2. 计算前缀和

ad[0] = nums[0];

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

    ad[i] = ad[i-1] + nums[i];

// 3. 对每个查询进行处理

// 找到最大的前缀和小于等于查询值的位置

**2.1.5 调试分析（遇到的问题和解决方法）**

 前缀和的边界处理

* 需要特别注意前缀和数组的第一个元素初始化
* 在查询过程中需要考虑前缀和数组的边界判断，即t+1 == n的情况

 查询处理的注意事项

* 当查询值大于所有元素和时，需要返回数组长度n
* 当查询值小于最小元素时，需要返回0
* 其他情况需要在前缀和数组中准确定位合适的位置

 代码优化分析

* 可以使用二分查找代替线性查找，提高查询效率
* 预处理时的排序和前缀和计算是必要的，不能省略
* 可以考虑使用更高效的排序算法

**2.1.6 总结和体会**

 算法设计心得

* 排序后求解是解决此类问题的关键策略
* 前缀和的应用使得子序列和的计算变得高效
* 理解了贪心策略在此类问题中的应用

 实现要点总结

* 数据预处理（排序、前缀和）对效率提升很关键
* 合理的数据结构选择可以简化代码实现
* 边界情况的处理需要特别注意

 可改进之处

* 查询过程可以使用二分查找优化
* 可以考虑使用更高效的排序方法
* 内存使用还可以进一步优化

 收获与思考

* 加深了对前缀和技巧的理解
* 体会到了预处理在算法设计中的重要性
* 学会了如何处理类似的子序列问题

**2.2 二叉排序树**

**2.2.1 问题描述**

实现一个可以维护整数集合的二叉排序树(BST)，支持重复关键字，并需要实现以下功能：

* 插入整数
* 删除整数
* 查询某数出现次数
* 查询最小值
* 查询某数的前驱

**2.2.2 基本要求**

输入格式：

* 第1行：整数n，表示操作数量
* 接下来n行：每行包含操作类型op和操作数x（除op=4外）
  + op=1：插入x
  + op=2：删除一个x
  + op=3：查询x的出现次数
  + op=4：查询最小值
  + op=5：查询x的前驱

输出格式：

* 除插入操作外，每个操作输出一行
* 对于不存在的情况输出"None"

**2.2.3 数据结构设计**

1.节点结构设计：

struct Node

{

    int key;     // 节点关键字值

    int count;   // 重复次数统计

    Node \*left;  // 左子树指针

    Node \*right; // 右子树指针

};

2. 二叉排序树类设计：

class BST

{

private:

    Node \*root; // 根节点指针

    // 五个基本操作的私有成员函数

    void insert();

    void delete\_num();

    void count\_num();

    void cout\_min();

    void cout\_pri();

public:

    void dispose(const int n); // 操作分发函数

};

**2.2.4功能说明（函数、类）**

1. 插入操作(insert)：

void BST::insert()

{

    // 1.读取待插入值

    // 2.若树为空，创建根节点

    // 3.否则查找插入位置

    // 4.如果找到相同值，count++

    // 5.否则创建新节点

}

2. 删除操作(delete\_num)：

void BST::delete\_num()

{

    // 查找阶段

    /\* 查找待删除节点并记录其父节点 \*/

    // 处理重复元素

    if (!now->count)

        /\* 如果count>1，直接count--并返回 \*/

    // 删除处理

    if (now->left == nullptr && now->right == nullptr)

        /\* 情况1：叶子节点直接删除 \*/

    else if (now->left == nullptr)

        /\* 情况2：只有右子树，右子树替代当前节点 \*/

    else if (now->right == nullptr)

        /\* 情况3：只有左子树，左子树替代当前节点 \*/

    else

        /\* 情况4：有两个子树，寻找左子树最大值作为替代 \*/

}

3. 查询操作(count\_num、cout\_min、cout\_pri)：

// 统计元素个数

void BST::count\_num()

{

    /\* 查找目标节点并返回count值，不存在返回0 \*/

}

// 查找最小值

void BST::cout\_min()

{

    /\* 一直向左遍历到最左节点 \*/

    Node \*now = root;

    while (now->left != nullptr)

        now = now->left;

    cout << now->key << endl;

}

// 查找前驱

void BST::cout\_pri()

{

    /\* 记录遍历过程中小于目标值的最大值 \*/

    int num, out = -999; // out记录当前找到的最大前驱

    cin >> num;

    Node \*now = root;

    while (now != nullptr)

    {

        if (now->key < num)

            out = max(out, now->key);

        if (now->key >= num)

            now = now->left;

        else

            now = now->right;

    }

}

推荐使用代码+注释的形式，清晰描述函数功能、输入内容和输出/返回内容

# 不要直接贴函数实现代码，请选择关键代码/伪代码展示并用注释辅助说明

# 例如

/\*\*

 \* @brief           二叉树指令构造

 \* @param   order   构造指令列表

 \* @param   index   当前节点序号

 \* @return          子树根节点指针

 \*/

TreeNode \*buildTreeDFS(const std::vector<int> &order, int &index) {

    if (子树为空) {

        返回空指针 // 返回空指针给父节点

}

// 构造以当前节点为根节点的子树并返回

    return new TreeNode(order[index ++], buildTreeDFS(order, index), buildTreeDFS(order, index));

}

**2.2.5 调试分析（遇到的问题和解决方法）**

1. 关键问题：

* 删除操作的各种情况处理
* 前驱查找的完整性
* 重复元素的计数维护

1. 解决方案：

* 分类讨论删除节点的子树情况
* 使用辅助变量记录遍历路径
* 通过count字段维护重复元素

**2.2.6 总结和体会**

1. 实现要点：

* 完整的二叉排序树操作集
* 重复元素的处理策略
* 指针操作的准确性

1. 收获：

* 加深了对BST基本操作的理解
* 掌握了处理重复元素的方法
* 提高了指针操作的熟练度

1. 优化建议：

* 可以添加平衡维护机制
* 考虑使用智能指针管理内存
* 可以增加异常处理机制

**2.3 换座位**

**2.3.1 问题描述**

在期末考试中，由于监考老师拿错了座位表，需要通过交换学生位置的方式，将错误的座位安排调整为正确的座位安排。每次交换可以选择两个学生互换位置，要求找出完成调整所需的最少交换次数。

**2.3.2 基本要求**

输入：

* 两个n×m的字符串数组，表示错误座位表(old\_chart)和正确座位表(new\_chart)
* 数据范围：1≤n,m≤200
* 学生名字由小写字母组成，长度≤5

输出：

* 一个整数，表示最少需要的交换次数

**2.3.3 数据结构设计**

class Solution

{

private:

    // 学生位置映射

    map<string, pair<int, int>> student\_pos;

    // 访问标记数组

    vector<vector<bool>> visited;

public:

    int solve(vector<vector<string>> &old\_chart,

              vector<vector<string>> &new\_chart);

};

**2.3.4功能说明（函数、类）**

1. 构建位置映射

// 将学生名字映射到其在原座位表中的位置

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

{

    for (int j = 0; j < m; j++)

    {

        student\_pos[old\_chart[i][j]] = make\_pair(i, j);

    }

}

2. 寻找置换环

// 追踪置换环的过程

while (!visited[curr\_i][curr\_j])

{

    visited[curr\_i][curr\_j] = true;

    cycle\_length++;

    // 找到下一个位置

    string target\_student = new\_chart[curr\_i][curr\_j];

    pair<int, int> next\_pos = student\_pos[target\_student];

    // 更新位置

    curr\_i = next\_pos.first;

    curr\_j = next\_pos.second;

}

3. 计算交换次数

// 对于长度为k的置换环，需要k-1次交换

if (cycle\_length > 0)

{

    total\_swaps += (cycle\_length - 1);

}

**2.3.5 调试分析（遇到的问题和解决方法）**

1. 关键问题：

* 置换环的识别和构建
* 最小交换次数的计算
* 重复位置的处理

1. 解决方案：

* 使用标记数组避免重复访问
* 通过map快速定位学生位置
* 基于置换环理论计算交换次数

**2.3.6 总结和体会**

1. 算法思路：

* 问题可以转化为置换环问题
* 每个置换环需要的交换次数是确定的
* 通过贪心策略寻找最优解

1. 实现要点：

* 高效的数据结构选择
* 清晰的循环控制逻辑
* 正确的访问标记处理

1. 优化建议：

* 可以考虑使用并查集
* 优化内存使用
* 添加输入验证

1. 收获：

* 深入理解了置换环理论
* 掌握了最小交换次数的计算方法
* 提高了复杂问题的分析能力

**2.4 家谱查询**

**2.4.1 问题描述**

实现一个家谱树系统，根据缩进格式的家谱信息构建家族关系，并能够回答关于家族成员关系的查询。支持判断父子关系、兄弟关系、祖先后代关系等多种关系查询。

**2.4.2 基本要求**

输入：

* 第一行：两个整数n,m表示家谱人数和查询数量
* 接下来n行：家谱信息，通过缩进表示层级关系
* 然后m行：关系查询语句

输出：

* 对每个查询输出True或False
* 每组测试用例后输出空行

数据范围：

* 20%数据：0<n≤20, 0<m≤40
* 40%数据：0<n≤100, 0<m≤200
* 100%数据：0<n≤5000, 0<m≤10000

**2.4.3 数据结构设计**

1. 树节点结构：

struct Node

{

    string name;             // 节点名字

    Node \*parent;            // 父节点指针

    vector<Node \*> children; // 子节点列表

    Node(string n) : name(n), parent(nullptr) {}

};

2．家谱树类：

class FamilyTree

{

private:

    Node \*root;                  // 根节点

    map<string, Node \*> nameMap; // 名字到节点的映射

    void cleanTree(Node \*node);                              // 清理树

    bool isAncestorHelper(Node \*ancestor, Node \*descendant); // 辅助函数

public:

    // 主要接口函数

    void buildTree(vector<string> &lines); // 构建树

    bool isChild(const string &child, const string &parent);

    bool isParent(const string &parent, const string &child);

    bool isSibling(const string &name1, const string &name2);

    bool isDescendant(const string &descendant, const string &ancestor);

    bool isAncestor(const string &ancestor, const string &descendant);

};

**2.4.4功能说明（函数、类）**

1. 树的构建过程：

void buildTree(vector<string> &lines)

{

    // 处理根节点

    root = new Node(lines[0]);

    nameMap[lines[0]] = root;

    // 记录每层最后节点和缩进

    Node \*lastNodes[71] = {nullptr};

    int lastIndents[71] = {0};

    lastNodes[0] = root;

    // 逐行处理，根据缩进构建关系

    for (size\_t i = 1; i < lines.size(); i++)

    {

        // 计算缩进确定层级

        // 创建新节点并建立父子关系

        // 更新层级信息

    }

}

2. 关系判断实现：

// 判断是否为子节点

bool isChild(const string &child, const string &parent)

{

    /\* 检查child是否为parent的直接子节点 \*/

}

// 判断是否为兄弟节点

bool isSibling(const string &name1, const string &name2)

{

    /\* 检查两个节点是否共享同一个父节点 \*/

}

// 判断祖先关系

bool isAncestor(const string &ancestor, const string &descendant)

{

    /\* 通过向上遍历判断祖先关系 \*/

}

**2.4.5 调试分析（遇到的问题和解决方法）**

1. 关键问题：

* 树的正确构建和层级关系维护
* 缩进处理与父子关系确定
* 特殊情况的处理（自身关系判断）

1. 解决方案：

* 使用数组记录层级信息
* 通过map实现快速节点查找
* 清晰的关系判断逻辑

**2.4.6 总结和体会**

1. 实现要点：

* 树形结构的设计与维护
* 高效的节点查找机制
* 多种关系判断的实现

1. 优化建议：

* 可以使用智能指针管理内存
* 优化字符串处理效率
* 改进树的构建算法

1. 收获：

* 加深了对树结构的理解
* 掌握了家谱关系的表示方法
* 提高了字符串处理能力

**2.5 哈希表**

**2.5.1 问题描述**

设计一个针对字符串的哈希函数系统，处理包含英文字母的人名列表。系统需要将字符串映射到哈希表中，并使用平方探测法解决冲突。

**2.5.2 基本要求**

 哈希函数计算：

* 公式：h(key) = ((...(key[0] \* 37+key[1]) \* 37+...)\*37+key[n-1])
* key[i]为字符ASCII码值

 映射规则：

* h(key)=h(key)%M
* M为大于等于P的最小素数

 数据范围：

* 1≤P≤10000
* 1≤N≤2P
* 字符串长度：1≤n≤100

**2.5.3 数据结构设计**

// 核心数据结构

vector<bool> used; // 标记位置是否被使用

// 关键函数声明

bool isPrime(int n);                                                              // 判断素数

int getNextPrime(int n);                                                          // 获取下一个素数

unsigned long long int getHash(const string &key);                                // 计算哈希值

int findPosition(vector<bool> &used, unsigned long long int hash, int tableSize); // 查找位置

**2.5.4功能说明（函数、类）**

1. 素数相关处理：

bool isPrime(int n)

{

    if (n < 2)

        return false;

    for (int i = 2; i \* i <= n; i++)

    {

        if (n % i == 0)

            return false;

    }

    return true;

}

int getNextPrime(int n)

{

    if (n <= 2)

        return 2;

    if (n % 2 == 0)

        n++; // 优化：从奇数开始检查

    while (!isPrime(n))

        n += 2;

    return n;

}

2. 哈希值计算：

unsigned long long int getHash(const string &key)

{

    unsigned long long int hash = 0;

    for (char c : key)

    {

        hash = hash \* 37ULL + (unsigned long long int)c;

    }

    return hash;

}

3. 位置查找：

int findPosition(vector<bool> &used, unsigned long long int hash, int tableSize)

{

    int base = hash % tableSize;

    // 检查基础位置

    if (!used[base])

        return base;

    // 平方探测

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

    {

        // 正向探测

        long long offset = ((long long)i \* i) % tableSize;

        int pos = (base + offset) % tableSize;

        if (!used[pos])

            return pos;

        // 负向探测

        pos = ((base - offset) % tableSize + tableSize) % tableSize;

        if (!used[pos])

            return pos;

    }

    return -1; // 未找到可用位置

}

**2.5.5 调试分析（遇到的问题和解决方法）**

1. 关键问题：

* 大数值处理（unsigned long long int）
* 平方探测的正确实现
* 哈希冲突的有效处理

1. 特别注意：

* 溢出不需要特殊处理
* 探测次数限制为表长
* 正负方向的平方探测

**2.5.6 总结和体会**

1. 技术要点：

* 哈希函数的设计与实现
* 冲突解决策略的选择
* 大数运算的处理方法

1. 优化方向：

* 素数判断的效率优化
* 平方探测的实现优化
* 内存使用的优化

1. 收获：

* 深入理解哈希表原理
* 掌握平方探测技术
* 提高数值处理能力

**2.6 最大频率栈**

**2.6.1 问题描述**

设计一种特殊的栈数据结构，在常规的压栈操作基础上，弹出操作需要移除并返回出现频率最高的元素，如果有多个频率最高的元素，则返回最靠近栈顶的元素。

**2.6.2 基本要求**

实现FreqStack类的三个方法：

1. FreqStack()：构造函数，初始化空栈
2. void push(int val)：将元素压入栈中
3. int pop()：弹出并返回频率最高的元素

数据范围：

* 操作数：1≤n≤20000
* 元素值：0≤val≤10^9

**2.6.3 数据结构设计**

class FreqStack

{

private:

    // 记录元素频率

    unordered\_map<int, int> freq;

    // 按频率分组的元素栈

    unordered\_map<int, vector<int>> group;

    // 当前最大频率

    int maxFreq;

public:

    FreqStack();

    void push(int val);

    int pop();

};

**2.6.4功能说明（函数、类）**

1. 构造函数：

FreqStack()

{

    maxFreq = 0; // 初始化最大频率为0

}

2. 压栈操作：

void push(int val)

{

    // 更新元素频率

    freq[val]++;

    // 更新最大频率

    maxFreq = max(maxFreq, freq[val]);

    // 将元素加入对应频率组

    group[freq[val]].push\_back(val);

}

3. 弹栈操作：

int pop()

{

    // 获取最高频率组的栈顶元素

    int val = group[maxFreq].back();

    // 移除该元素

    group[maxFreq].pop\_back();

    // 更新元素频率

    freq[val]--;

    // 如果当前最高频率组为空，降低maxFreq

    if (group[maxFreq].empty())

    {

        maxFreq--;

    }

    return val;

}

**2.6.5 调试分析（遇到的问题和解决方法）**

1. 关键问题：

* 频率统计的准确性
* 最大频率的维护
* 同频率元素的顺序保持

1. 解决方案：

* 使用哈希表快速统计频率
* 动态更新最大频率
* 使用栈保持元素顺序

**2.6.6 总结和体会**

1. 设计思路：

* 组合使用哈希表和栈
* 按频率分组存储元素
* 维护全局最大频率

1. 实现要点：

* 高效的频率统计
* 正确的元素顺序维护
* 合理的内存使用

1. 优化建议：

* 可以考虑使用优先队列
* 优化内存占用
* 添加边界检查

1. 收获：

* 掌握了复合数据结构的设计
* 理解了频率统计的实现方法
* 提高了数据结构组合使用的能力

**3. 实验总结**

本次实验实现了多个数据结构和算法问题，通过动手实践加深了对数据结构的理解。主要涉及以下内容：

1.数据结构应用

二叉排序树：实现了插入、删除、查找等基本操作，处理了平衡问题和重复元素

哈希表：设计了字符串哈希函数，使用平方探测法解决冲突

家谱树：使用树结构表示家族关系，实现多种关系查询

频率栈：组合哈希表和栈实现特殊的弹出规则

2.算法设计要点

换座位问题：通过置换环理论最小化交换次数

前缀和技术：在最长子序列问题中优化查询

平衡维护：在二叉排序树中处理节点平衡

冲突处理：在哈希表中实现高效的探测策略

3.编程技巧总结

处理大数据时注意数据类型选择

善用函数指针实现操作分发

注重代码可维护性和复用性

4.关键难点突破

树结构的遍历和关系判断

哈希冲突的有效处理

置换环理论的实践应用

复杂数据结构的组合使用

5.实验收获

加深了对各类数据结构的理解

提高了算法设计和实现能力

学会了处理复杂工程问题的方法

培养了调试和优化代码的能力

通过本次实验，不仅巩固了课堂知识，也提高了实际编程能力。尤其是在处理哈希表、二叉树等复杂数据结构时，通过实践加深了理解。同时，在解决具体问题的过程中，也学会了如何选择合适的数据结构和算法，为今后解决更复杂的实际问题打下了良好基础。