# 作业 HW5* 实验报告

姓名：刘彦 学号：2352018 日期：2024 年 12 月 18 日

## 1. 涉及数据结构和相关背景

### 1.1 查找的基本概念

查找操作是指在一个数据结构中寻找某个元素的位置，通常表现为检索和搜索。查找操作的时间复杂度直接影响到程序的效率。常见的查找操作包括：

- 线性查找：顺序遍历数据结构，逐个元素检查是否为目标元素。
- 二分查找：在已排序的数据结构中，采用分治法逐步缩小查找范围。
- 哈希查找：通过哈希函数计算出元素的索引位置，以常数时间快速定位。

### 1.2 常见的查找数据结构

#### 1.2.1 数组和链表

数组：数组是最简单的线性数据结构，查找一个元素通常需要遍历整个数组，时间复杂度为 O(n)（线性查找）。在已排序数组中，可以使用二分查找将查找时间降低到 O(logn)。链表：链表是另一种常见的线性数据结构，和数组类似，普通链表的查找时间复杂度是 O(n)，但链表不需要连续的内存空间，适合动态大小的数据集。

#### 1.2.2 哈希表（Hash Table）

哈希表（或哈希映射）是一种通过哈希函数将元素映射到特定位置的数据结构。哈希表的查找、插入和删除操作平均时间复杂度是 O(1)，非常适合需要高效查找的数据密集型应用场景。哈希表的基本原理如下：

哈希函数：将数据映射到数组中的某个位置。理想的哈希函数能将输入数据均匀地分配到哈希表中，以减少冲突。

冲突解决：当不同的输入映射到相同的索引时，需要一种策略来处理冲突。常见的冲突

#### 1.2.3 二叉搜索树（BST）

二叉搜索树是一种二叉树，其中每个节点的值大于其左子树的所有节点值，小于其右子树的所有节点值。查找一个元素时，通过比较节点的值与当前节点的值，递归向左或向右子树查找。由于 BST 的结构，查找的平均时间复杂度为 O(logn)，但是最坏情况下，如果树不平衡，时间复杂度可能退化为 O(n)。

平衡二叉搜索树：为了避免最坏情况下退化为线性结构，可以使用平衡树，如 AVL 树、红黑树等，它们通过旋转操作保持树的平衡，确保查找操作的时间复杂度始终保持在 O(logn)。

## 2. 实验内容

### 2.1 和有限的最长子序列

#### 2.1.1 问题描述

给你一个长度为 n 的整数数组 nums 和一个长度为 m 的整数数组 queries，返回一个长度为 m 的数组 answer，其中 answer[i]是 nums 中元素之和小于等于 queries[i]的子序列的最大长度。

子序列是由一个数组删除某些元素(也可以不删除)但不改变元素顺序得到的一个数组。

### 2.1.2 基本要求

输入：第一行包括两个整数 n 和 m，分别表示数组 nums 和 queries 的长度。第二行包括 n 个整数，为数组 nums 中元素。第三行包含 m 个整数，为数组 queries 中元

对于 20%的数据，有 1<=n,m<=10

对于 40%的数据，有 1<=n,m<=100

对于 100%的数据，有 1<=n,m<=1000

对于所有数据，1<=nums[i],queries[i]<=106

输出：输出一行，包括 m 个整数，为 answer 中元素。

### 2.1.3 数据结构设计

nums 存储输入的整数数据，queries 存储查询值，prefix_sum 存储前缀和，result 存储查询结果。

```
int nums[1000], queries[1000], prefix_sum[1001], result[1000];
```

### 2.1.4 功能说明（函数、类）

实现二分查找的函数

```
/**
 * @brief           二分查找函数
 * @param prefix_sum 前缀和数组
 * @param n          数组长度
 * @param target     目标值
 * @return          返回满足条件的最大长度
 */
int binarySearch(int prefix_sum[], int n, int target)
{
    int left = 0, right = n;
    while (left < right)
    {
        int mid = (left + right + 1) / 2;
        if (prefix_sum[mid] <= target)
            left = mid;  // 尝试扩大范围
        else
            right = mid - 1;
    }
    return left; // 返回满足条件的最大长度
}
```

main 函数中关键部分

```
// Step 1: 对 nums 排序
    bubbleSort(nums, n);
```

```
    // Step 2: 计算前缀和
    prefix_sum[0] = 0;
    for (int i = 1; i <= n; ++i)
        prefix_sum[i] = prefix_sum[i - 1] + nums[i - 1];
    // Step 3: 处理每个 query，使用二分查找
    for (int i = 0; i < m; ++i)
        result[i] = binarySearch(prefix_sum, n, queries[i]);
```

### 2.1.5 调试分析（遇到的问题和解决方法）

2.1.5.1 冒泡排序效率问题

在处理较大的数据集时，冒泡排序的效率较低，导致程序运行时间过长，尤其在 n 较大的情况下。我尝试使用一些优化方法，比如在每轮排序过程中，如果没有交换发生，提前终止排序，增加了一个标志位 swapped，在每一轮排序时，如果没有进行交换，就提前退出循环。

```
bool swapped = false;  // 添加标志位
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        if (!swapped) break;  // 没有交换，提前退出
```

2.1.5.2 二分查找返回值不正确

在使用二分查找时，结果并没有按预期工作。具体来说，查询目标值的结果并不总是准确，尤其在某些边界条件下，返回的索引错误.我检查了 binarySearch 函数，发现 left 和 right 的更新逻辑有些问题，导致在某些情况下查询的边界不够准确，最终结果出错。通过逐步打印出 left、right 和 mid 的值，发现 mid 的计算没有准确选择合适的位置，尤其在循环终止时。我更新了 binarySearch 的条件，确保每次更新时，能够准确找到满足条件的最大值。

### 2.1.6 总结和体会

通过二分查找优化了查询的过程，能够在 O(logn)的时间内快速定位目标位置。特别是在前缀和数组中，二分查找能够迅速给出答案。通过调试，我意识到对于边界的细致处理是至关重要的，确保 left 和 right 的更新能够准确定位。

## 2.2 二叉排序树

### 2.2.1 问题描述

二叉排序树 BST（二叉查找树）是一种动态查找表，基本操作集包括：创建、查找，插入，删除，查找最大值，查找最小值等。

本题实现一个维护整数集合（允许有重复关键字）的 BST，并具有以下功能：1. 插入一个整数 2.删除一个整数 3.查询某个整数有多少个 4.查询最小值 5. 查询某个数字的前驱（集合中比该数字小的最大值）。

### 2.2.2 基本要求

输入：第 1 行一个整数 n，表示操作的个数；接下来 n 行，每行一个操作，第一个数字 op 表示操作种类：

- 若 op=1，后面跟着一个整数 x，表示插入数字 x
- 若 op=2，后面跟着一个整数 x，表示删除数字 x（若存在则删除，否则输出 None，若有多个则只删除一个），
- 若 op=3，后面跟着一个整数 x，输出数字 x 在集合中有多少个（若 x 不在集合中则输出 0）
- 若 op=4，输出集合中的最小值（保证集合非空）
- 若 op=5，后面跟着一个整数 x，输出 x 的前驱（若不存在前驱则输出 None，x 不一定在集合中）

输出：一个操作输出 1 行（除了插入操作没有输出）

### 2.2.3 数据结构设计

```
struct TreeNode {
    int val;  int count;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x)
    {
        val = x;
        count = 1;
        left = nullptr;
        right = nullptr;
    }
};
class BST {
private:
    TreeNode* root;
public:
    BST();
    void insert(int x); // 插入操作
    void remove(int x); // 删除操作
    void count(int x); // 查询某个数字的计数
    void findMinValue(); // 查询最小值
    void findPredecessor(int x); // 查询某个数字的前驱
private:
    void insert(TreeNode*& node, int x); // 内部实现：插入值 x
    void remove(TreeNode*& node, int x); // 内部实现：删除值 x
    TreeNode* find(TreeNode* node, int x); // 内部实现：查找节点 x
    TreeNode* findMin(TreeNode* node); // 内部实现：查找树的最小值
    TreeNode* findMax(TreeNode* node); // 内部实现：查找树的最大值
    TreeNode* findPredecessor(TreeNode* node, int x);//内部实现：查找 x 的前驱
```

```
};
```

### 2.2.4 功能说明（函数、类）

实现插入值的函数

```
/**
 * @brief           内部实现：插入值 x
 */
void BST::insert(TreeNode*& node, int x)
{
    if (node == nullptr) {
        node = new TreeNode(x);
        return;
    }
    if (x < node->val)  insert(node->left, x);
    else if (x > node->val)  insert(node->right, x);
    else  node->count++;
}
```

实现删除值的函数

```
/**
 * @brief           内部实现：删除值 x
 */
void BST::remove(TreeNode*& node, int x)
{
    if (node == nullptr)  return;
    if (x < node->val)  remove(node->left, x);
    else if (x > node->val)  remove(node->right, x);
    else {
        if (node->count > 1)  node->count--;
        else {
            if (node->left == nullptr && node->right == nullptr) {
                delete node;
                node = nullptr;
            }
            else if (node->left == nullptr) {
                TreeNode* temp = node;
                node = node->right;
                delete temp;
            }
            else if (node->right == nullptr) {
                TreeNode* temp = node;
                node = node->left;
                delete temp;
            }
            else {
```

```
                TreeNode* successor = findMin(node->right);
                node->val = successor->val;
                node->count = successor->count;
                remove(node->right, successor->val);
            }
        }
    }
}
```

实现查找节点的函数

```
/**
 * @brief           内部实现：查找节点 x
 */
TreeNode* BST::find(TreeNode* node, int x)
{
    if (node == nullptr)  return nullptr;
    if (x < node->val)  return find(node->left, x);
    else if (x > node->val)  return find(node->right, x);
    else  return node;
}
```

实现查找树的最小值的函数

```
/**
 * @brief           内部实现：查找树的最小值
 */
TreeNode* BST::findMin(TreeNode* node)
{
    while (node && node->left)  node = node->left;
    return node;
}
```

实现查找树的最大值的函数

```
/**
 * @brief           内部实现：查找树的最大值
 */
TreeNode* BST::findMax(TreeNode* node)
{
    while (node && node->right)  node = node->right;
    return node;
}
```

查找前驱的函数

```
/**
 * @brief           内部实现：查找 x 的前驱
 */
TreeNode* BST::findPredecessor(TreeNode* node, int x)
{
    if (node == nullptr)  return nullptr;
```

```
    if (x <= node->val)  return findPredecessor(node->left, x);
    else {
        TreeNode* temp = findPredecessor(node->right, x);
        if (temp)  return temp;
        return node;
    }
}
```

### 2.2.5 调试分析（遇到的问题和解决方法）

2.2.5.1 处理重复元素的插入

在 BST 中，通常插入操作会保证树的二叉搜索特性，即左子树的所有节点值小于根节点值，右子树的所有节点值大于根节点值。但如果允许重复元素出现，如何处理就显得尤为重要。当插入相同的元素时，BST 会默认将新元素插入到树中的某一位置，这可能会破坏树的结构。

因此，需要一个机制来处理重复元素。我在节点中维护一个计数器来记录该元素出现的次数。当插入一个重复的元素时，应该增加该节点的计数，而不是重新插入相同的元素。

2.1.5.2 高效查找前驱节点

在 AVL 树中，查找前驱节点是一个常见操作，尤其是在实现 `find` 功能时。前驱节点是比当前节点值小的最大节点。我们可以通过以下方式提高查找效率

在 AVL 树中，查找某个节点的前驱可以分为两个情境：

- 当前节点有左子树：在这种情况下，前驱是当前节点的左子树中最大的节点（即最右节点）。
- 当前节点没有左子树：前驱节点是当前节点的祖先中第一个其值大于当前节点值的节点。

通过对树的结构和递归进行优化，可以提高查找前驱节点的效率。

### 2.2.6 总结和体会

在许多应用中，特别是在需要频繁插入、删除、查找的场景下，普通的二叉搜索树（BST）会因为树的高度可能过高而导致性能下降（最坏情况时间复杂度可能是 `O(n)`）。而 AVL 树通过自我平衡的机制，确保了每次插入和删除操作的时间复杂度为 `O(logn)`，大大提高了树的操作效率。

## 2.3 换座位

### 2.3.1 问题描述

期末考试，监考老师粗心拿错了座位表，学生已经落座，现在需要按正确的座位表给学生重新排座。假设一次交换你可以选择两个学生并让他们交换位置，给你原来错误的座位表和正确的座位表，问给学生重新排座需要最少的交换次数。

### 2.3.2 基本要求

输入：两个 n*m 的字符串数组，表示错误和正确的座位表 old_chart 和 new_chart，old_chart[i][j]为原来坐在第 i 行第 j 列的学生名字。

对于 `100%`的数据，`1<=n,m<=200`

人名为仅由小写英文字母组成的字符串，长度不大于 5。

输出：一个整数，表示最少交换次数。

提交要求：本题需要提交类 Solution 的实现，类中需要包含一个名为 slove 的函数，下载 tamplate.cpp 查看详细信息，完成类的定义，提交时仅提交类的定义相关代码

### 2.3.3 数据结构设计

使用 old_chart 和 new_chart 两个二维数组来表示旧的和新的排序。我们需要将它们转换为一维数组 old_vec 和 new_vec，以便进行后续的操作。

用一个 map<string, int> position_map 来记录每个学生在 new_vec 中的目标位置。通过这个映射关系，我们可以快速查找 old_vec 中每个学生在 new_vec 中的位置，从而构建一个目标位置数组 target_pos，该数组表示每个学生在目标排序中应该去的位置。

使用一个布尔数组 vector<bool> visited 来记录每个学生是否已经在正确的位置，避免重复计算。通过环的方式计算最小交换次数：每个学生从 old_vec 出发，通过目标位置数组 target_pos 找到新的位置，直到回到起始位置。每次访问一个环中的学生时，都标记为已访问。环的长度为 k 时，需要 k-1 次交换来将环中的所有元素放到正确位置。

### 2.3.4 功能说明（函数、类）

```
class Solution {
public:
    int solve(vector<vector<string>> &old_chart, vector<vector<string>>
&new_chart) {
        int n = old_chart.size();
        int m = old_chart[0].size();
        int total_size = n * m;
        vector<string> old_vec, new_vec;  // 将二维数组转为一维数组
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++) {
                old_vec.push_back(old_chart[i][j]);
                new_vec.push_back(new_chart[i][j]);
            }
        // 使用 map 来记录 new_vec 中学生的目标位置
        map<string, int> position_map;
        for (int i = 0; i < total_size; i++)
            position_map[new_vec[i]] = i;
        vector<int> target_pos(total_size);  // 创建目标位置数组
        for (int i = 0; i < total_size; i++) {
            target_pos[i] = position_map[old_vec[i]];
        }
        // 使用 visited 数组来检查每个学生是否已经在正确的位置
        vector<bool> visited(total_size, false);
        int swap_count = 0;
        for (int i = 0; i < total_size; i++) {  // 计算最小交换次数
            // 如果该位置已经访问过，或者已经在正确的位置，跳过
            if (visited[i] || target_pos[i] == i)
                continue;
```

```
            // 计算环的长度
            int cycle_length = 0;
            int current = i;
            while (!visited[current]) {
                visited[current] = true;
                current = target_pos[current];
                cycle_length++;
            }
            // 如果环的长度大于 1，那么需要进行 `cycle_length - 1` 次交换
            if (cycle_length > 1)
                swap_count += cycle_length - 1;
        }
        return swap_count;
    }
};
```

### 2.3.5 调试分析（遇到的问题和解决方法）

2.3.5.1 目标位置数组的构建错误

构建目标位置数组时，`target_pos` 数组的值不正确，导致最终的交换次数计算错误。特别是如果 `position_map` 中的映射关系没有正确建立，`target_pos` 可能会指向错误的位置。解决方法是在构建 `position_map` 时，确保每个学生在 `new_vec` 中的位置正确。可以在调试时打印出 `position_map` 中的值，确保它们与预期一致。

2.3.5.2 环的长度小于 2 时没有更新交换次数

在计算最小交换次数时，如果环的长度为 1，应该跳过该环，因为不需要进行任何交换。如果环的长度大于 1，交换次数应该是 `cycle_length - 1`。如果这部分逻辑不正确，会导致交换次数计算错误。确保只有当环的长度大于 1 时，才更新交换次数。

### 2.3.6 总结和体会

本题的核心思想是通过环的长度来计算最小交换次数。这类问题实际上涉及到图的环和排列的概念。每次计算环的长度时，我们需要识别出一个“环”并在其中交换元素，直到每个元素都处于正确的位置。我在计算交换次数时采用了环排序的思想。每次计算环的长度并交换，这样可以避免不必要的重复交换，从而减少交换次数。通过这种方式，解决方案的时间复杂度得到了优化。

## 2.4 最大频率栈

### 2.4.1 问题描述

设计一个类似堆栈的数据结构，将元素推入堆栈，并从堆栈中弹出出现频率最高的元素。实现 FreqStack 类：

- FreqStack() 构造一个空的堆栈。
- void push(int val) 将一个整数 val 压入栈顶。
- int pop() 删除并返回堆栈中出现频率最高的元素。如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素。

### 2.4.2 基本要求

输入：第一行包含一个整数 n。接下来 n 行每行包含一个字符串（push 或 pop）表示一个操作，若操作为 push，则该行额外包含一个整数 val，表示压入堆栈的元素。

对于 100%的测试数据，1<=n<=20000，0<=val<=10^9

且当堆栈为空时不会输入 pop 操作

输出：包含若干行，每有一个 pop 操作对应一行，为弹出堆栈的元素

提交要求：本题需要提交类 FreqStack 的实现，类中需要包含三个名为 FreqStack、push、pop 的函数，下载 tamplate.cpp 查看详细信息，完成类的定义，提交时仅提交类的定义相关代码。

### 2.4.3 数据结构设计

```
    map<int, int> frequency_map;  // 存储元素的频率
    map<int, stack<int>> freq_map;  // 存储频率到元素的映射
    int max_freq;  // 当前栈中最大的频率
```

### 2.4.4 功能说明（函数、类）

```
class FreqStack {
private:
    map<int, int> frequency_map;  // 存储元素的频率
    map<int, stack<int>> freq_map;  // 存储频率到元素的映射
    int max_freq;  // 当前栈中最大的频率
public:
    FreqStack() : max_freq(0) {}
    void push(int val) { // 将元素 val 压入栈中
        int freq = ++frequency_map[val]; // 更新频率
        max_freq = max(max_freq, freq); // 更新最大频率
        freq_map[freq].push(val); // 将元素添加到频率栈中
    }
    int pop() { // 从栈中弹出元素
        int val = freq_map[max_freq].top();
        freq_map[max_freq].pop(); // 获取当前最大频率栈中的顶部元素
        frequency_map[val]--; // 更新频率
        if (freq_map[max_freq].empty()) // 如果该频率的栈为空，更新最大频率
            max_freq--;
        return val;
    }
};
```

### 2.4.5 调试分析（遇到的问题和解决方法）

2.4.5.1 栈为空时未更新最大频率

在 pop()操作中，如果当前最大频率的栈已经为空(即该频率的所有元素都已被弹出)，需要更新 max_freq。如果没有正确更新，后续的 pop()操作可能会尝试从空的栈中弹出元素，导致错误。最终我在 pop()函数中中，在弹出元素之后，检查当前频率栈是否为空。如果为空，则需要更新 max_freq，减少当前最大频率。

2.4.5.2 频率栈的元素顺序不正确

在 freq_map 中存储频率栈时，可能会遇到栈内的元素顺序不符合预期。例如，如果在高频栈中，应该弹出的是最后一个压入的元素，但由于某些原因栈的顺序不正确。解决方法是每次 push(val)时，将元素压入到对应频率的栈中，并确保栈顶始终是最后压入的元素。确保在每次 pop()时，栈顶元素就是最后压入的频率最高的元素。

### 2.4.6 总结和体会

在实现和调试过程中，逐步验证每个操作的正确性非常重要。例如，通过检查 max_freq 的更新是否正确，验证 push 和 pop 操作是否按照预期的顺序进行，可以帮助我们在程序出现问题时快速定位错误。

## 3. 实验总结

### 3.1 实验目标和内容回顾

实验的主要目标是：

- 理解和掌握常见的查找数据结构，如哈希表、二叉搜索树等
- 在实际场景中应用这些数据结构解决查找问题
- 设计并实现相应的查找算法，测试其性能和效率

实验内容涉及：

- 设计并实现了哈希表和二叉搜索树（BST）的基本查找操作
- 实现了哈希冲突的处理方法
- 进行了多种查询操作的测试，评估了不同数据结构在查找操作中的性能

### 3.2 收获与体会

通过本次实验，我深刻体会到了以下几个方面的内容：

①选择合适的数据结构非常重要

在不同的应用场景中，选择合适的数据结构对提升程序性能至关重要。例如，对于频繁查找的应用场景，哈希表是一个理想的选择，而对于需要顺序遍历或范围查询的场景，二叉搜索树会更合适。理解这些数据结构的优缺点，有助于在实际开发中做出更合理的选择。

②数据结构的实际应用：

通过实现哈希表、二叉搜索树等数据结构，我不仅加深了对它们的理解，还能够将它们应用于实际问题中。这让我对如何通过不同的数据结构解决实际问题有了更深刻的认识，特别是在性能优化和资源管理方面。

③调试与优化

在实验中，我不仅实现了数据结构的基本功能，还进行了性能调优和内存优化。特别是在面对大规模数据时，如何合理管理内存、如何选择合适的算法来优化查找操作的效率，是我最大的收获。