HW5 实验报告

1. 涉及数据结构和相关背景

- 顺序查找
- 二分查找
- 二叉排序树查找
- 二叉平衡树
- 哈希表

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中元素

  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

  下载编译并运行p126_data.cpp以生成随机测试数据
- 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

- 将数组中的数进行排序，计算排序后数组中每一位数字的前缀和，将每一位的前缀和与answer[i]进行比较，记录最大遍历到的位置即可
- 原数组与前缀和数组

```
int s[1010], sumer[1010];
```

2.1.4 功能说明

- 输入数据与前缀和记录

```
cin >> n >> m;
for (int i = 1; i <= n; i++)
{
    cin >> s[i];
}

//将原数组进行排序
sort(s + 1, s + n + 1);

//记录前缀和数组
for (int i = 1; i <= n; i++)
{
    sumer[i] = sumer[i - 1] + s[i];
}
```

- 进行比较，记录最远序号

```
for (int j = 1; j <= m; j++)
{
    cin >> tar;

    if (tar > sumer[n])
    {
        cout << n << " ";
        continue;
    }

    for (int i = 0; i <= n; i++)
    {
        if (sumer[i] > tar)
        {
            cout << i - 1 << " ";
            break;
        }

    }

}
```

2.1.5 调试分析

- 注意为了保证前缀和数组的操作一致性，前缀和以及原数组的下标开始值均为1，而0下标下对应的值为0，

2.1.6 总结和体会

- 涉及到无定序的子序列时，考虑进行排序解决问题

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

- 将BST优化为AVL进行查找
- 节点类
- ```
  class Node
  {
  public:
      //construction
      Node(int ele, Node* parent)
      {
          this->ele = ele;            //存储元素
          this->parent = parent;      //父节点
          this->left = NULL;          //左孩子
  ```
  ```

```
        this->right = NULL;          //右孩子
        this->cnt = 1;               //记录重复元素的数量，避免重复插入
        this->height = 1;            //当前子树的高度
    }

    //balance factor
    int finderBalanceFactor();       //计算平衡因子


    //update the height
    void updateHeight();             //更新子树高度

    bool finder(int ele, Node*& pre, Node*& thiss, Node*& next, bool&
found);

    Node* left, * right, * parent;
    int ele, cnt, height;
};
```

- AVL树的根节点

```
//the root of AVL tree
Node* root = NULL;
```

2.2.4 功能说明

- 节点的性质与方法
- 计算当前节点的平衡因子

```
int finderBalanceFactor()
{
    //the height of right subtree minus the height of left subtree
    return (this->right ? this->right->height : 0) - (this->left ? this-
>left->height : 0);
}
```

> 计算方法是右子树高度减去左子树的高度，注意判断左右子树为空的情况

- 更新当前子树高度

```
 void updateHeight()
 {
     //dont forget to add up the height of itself
     this->height = max((this->left ? this->left->height : 0), (this->right
? this->right->height : 0)) + 1;
 }
```

> 更新当前子树高度为左右子树的高度的最大值再加上本身的高度1，作为当前子树的高度

- 查找插入位置

```
bool finder(int ele, Node*& pre, Node*& thiss, Node*& next, bool& found)
{
    if (this->left)
        if (this->left->finder(ele, pre, thiss, next, found))
            return true;
    if (found) { // 已经找到，该节点即为后继节点，所有节点都找到，遍历退出
        next = this;
        return true;
    }
    if (this->ele > ele) { // 没有对应节点，找到前驱和后继，遍历退出
        thiss = NULL;
        next = this;
        return true;
    }
    if (this->ele == ele) { // 找到对应节点，需要遍历右子树找后继
        thiss = this;
        found = true;
    }
    else
        pre = this; // 没有找到对应节点，当前节点值小于查找元素，暂时设为前驱
    if (this->right)
        return this->right->finder(ele, pre, thiss, next, found);
    return false; // 查找还未结束
}
```

> 利用递归的形式进行插入节点位置的查找

- AVL树方法
- 向AVL树中插入节点

```
void insert(int ele)
{
    if (!root)
    {
        root = new Node(ele, NULL);
        return;
    }

    Node* l = NULL, * p = NULL, * n = NULL;
    bool found = false;
    root->finder(ele, l, p, n, found);

    if (p)                                  //找到了，说明原树种已经有当前元素，计数++不用重复插入
    {
        p->cnt++;
        return;
    }

    if (l && !l->right)                     //左子树不空，右子树空，向右子树插入
    {
        p = (l->right = new Node(ele, l));
    }
    else
```

```
    {
        p = (n->left = new Node(ele, n));    //相反情况，向左子树插入节点
    }
    keepBalance(p);                          //更新平衡因子

}
```

> 注意重复元素直接将对应节点计数++即可，不用重复插入
>
> 最后要进行平衡因子的更新

- 删除元素

```
bool deleteItem(int ele)
{
    if (!root)
        return false;
    Node* l = NULL, * p = NULL, * n = NULL;
    bool found = false;
    root->finder(ele, l, p, n, found);
    if (!p) // 没找到
        return false;
    if (p->cnt > 1) { // 待删除元素不止一个，计数减一即可
        p->cnt--;
        return true;
    }
    // 待删除元素只有一个，需要删除节点
    if (!p->left && !p->right) { // 待删除节点为叶子节点
        if (!p->parent) {
            root = NULL;
        }
        else {
            ChangeParent(p, NULL);
            keepBalance(p->parent);
        }
    }
    else if (!p->left) { // 没有左子树，无法用前驱代替自己，用右子树代替自己
        // 如果没有右子树，也可以直接用左子树代替自己
        ChangeParent(p, p->right);
        keepBalance(p->parent);
    }
    else { // 有左子树，则前驱一定在左子树中，用前驱代替自己
        if (p->left == l) { // 前驱就是左孩子
            l->right = p->right;
            if (p->right)
                p->right->parent = l;
            ChangeParent(p, l);
            keepBalance(l);
        }
        else { // 找到前驱，用前驱代替自己，处理亲子关系
            Node* t = l->parent;
            t->right = l->left;
            if (l->left)
                l->left->parent = t;

            l->left = p->left;
```

```
            p->left->parent = l;
            l->right = p->right;
            if (p->right)
                p->right->parent = l;
            ChangeParent(p, l);
            keepBalance(t);
        }
    }
    p->left = p->right = NULL;
    delete p;
    return 1;
}
```

> 删除元素需要进行树的平衡调整，同样注意删除具有多个重复元素的节点，直接将计数器即可

- 返回某节点出现次数

```
int counter(int ele)
{
    if (!root)
        return 0;
    Node* l = NULL, * p = NULL, * n = NULL;
    bool found = false;
    root->finder(ele, l, p, n, found);
    if (p)
        return p->cnt;                          //直接返回计数器的值即可
    return 0;
}
```

> 每一个节点都有相应的计数器的值，若能找到该节点则直接返回即可

- 寻找最小值

```
int finderMin()
{
    if (!root)
        return -1;
    Node* p = root;
    while (p->left)
        p = p->left;
    return p->ele;
}
```

- 寻找该节点的父节点

```
int finderParent(int ele)
{
    if (!root)
        return -1;
    Node* l = NULL, * p = NULL, * n = NULL;
    bool found = false;

    root->finder(ele, l, p, n, found);
    if (l)
        return l->ele;
    return -1;
}
```

- 进行平衡操作
  - 右旋转
  - ```
    void RotateRight(Node* p) {
        Node* t = p->left;
        p->left = t->right;
        if (t->right)
            t->right->parent = p;
        t->right = p;
        UpdateRelation(p, t);
    }
    ```
  - 左旋转

    ```
    void RotateLeft(Node* p) {
        Node* t = p->right;
        p->right = t->left;
        if (t->left)
            t->left->parent = p;
        t->left = p;
        UpdateRelation(p, t);
    }
    ```
  - 进行旋转调整

    ```
    void keepBalance(Node* p)
    {
        while (p) {
            p->updateHeight();
            if (-2 >= p->finderBalanceFactor() || p->finderBalanceFactor()
    >= 2) {
                // 旋转
                if (p->finderBalanceFactor() < 0) { //LL型
                    if (p->left->finderBalanceFactor() <= 0) {
                        RotateRight(p);
                    }
                    else {                          //LR型
                        RotateLeft(p->left);
                        RotateRight(p);
                    }
    ```
    ```

```
            }
            else {
                if (p->right->finderBalanceFactor() >= 0) {
                    RotateLeft(p);              //RR型
                }
                else {
                    RotateRight(p->right);      //RL型
                    RotateLeft(p);
                }
            }
        }
        p = p->parent;                          //从当前插入元素向上找到第一个不满足平衡树条件的节点，进行平衡
    }
}
```

- 更新旋转后的亲子关系

```
void ChangeParent(Node* p, Node* t) {       //更新指针域
    if (t)
        t->parent = p->parent;
    if (p->parent) {
        if (p == p->parent->left)
            p->parent->left = t;
        else
            p->parent->right = t;
    }
    else
        root = t;
}

void UpdateRelation(Node* p, Node* t) {     //更新当前子树更新后的高度
    ChangeParent(p, t);
    p->parent = t;
    p->updateHeight();
    t->updateHeight();
}
```

- 主函数调用

```
int main()
{
    int n;
    cin >> n;
    while (n)
    {
        int option = 0, ele = 0, last = 0;
        cin >> option;
        if (option != 4)
            cin >> ele;

        if (option == 1)
            insert(ele);
```

```
        else if (option == 2)
        {
            if (!deleteItem(ele))
                cout << "None" << endl;
        }
        else if (option == 3)
            cout << counter(ele) << endl;
        else if (option == 4)
            cout << finderMin() << endl;
        else if (option == 5)
        {
            last = finderParent(ele);
            if (last == -1)
                cout << "None" << endl;
            else
                cout << last << endl;
        }
        n--;
    }
    return 0;
}
```

2.2.5 调试分析

- 要特别注意插入元素有重复元素的情况，考虑到题目中有取出此元素的数量，则可以在节点处添加一共计数器，记录当前元素重复的数量，防止重复插入，同时满足取出数量的要求
- AVL中进行的旋转操作需要进行左旋右旋，分为四种情况讨论
- 再进行旋转时需要找到第一个不满足平衡条件的节点进行平衡

2.2.6 总结和体会

- 进行了AVL树的构建，熟悉了BST的构建、查找方法
- 可以改进：将AVL的方法在类中进行封装，对外界提供相应接口给i偶见AVL树结构

2.3哈希表

2.3.1 问题描述

- 本题针对字符串设计哈希函数。假定有一个班级的人名名单，用汉语拼音（英文字母）表示。
- 要求：
    - 首先把人名转换成整数，采用函数h(key)=((...(key[0] * 37+key[1]) * 37+...)*37+key[n-2]*) 37+key[n-1]，其中key[i]表示人名从左往右的第i个字母的ascii码值(i从0计数,字符串长度为n，1<=n<=100)。
    - 采取除留余数法将整数映射到长度为P的散列表中，h(key)=h(key)%M，若P不是素数，则M是大于P的最小素数，并将表长P设置成M。
    - 采用平方探测法（二次探测再散列）解决冲突。（有可能找不到插入位置，当探测次数>表长时停止探测）

2.3.2 基本要求

- 输入：
- 第1行输入2个整数N、P，分别为待插入关键字总数、散列表的长度。若P不是素数，则取大于P的最小素数作为表长。
- 第2行给出N个字符串，每一个字符串表示一个人名
- 输出：
- 在1行内输出每个字符串插入到散列表中的位置，以空格分割，若探测后始终找不到插入位置，输出一个'-'

2.3.3 数据结构设计

- 首先利用素数筛（欧拉筛）寻找不小于p的最小素数

```
int primes[N];
```

- 用哈希函数进行地址的映射

```
int hashTable[N] = { 0 };
```

- 用访问数组记录当前位置是否被访问过

```
int hasher(int mod, string s);
```

- 探测方法-二次探测法解决哈希重冲突

```
int findIt(string& str);
```

2.3.4 功能说明

- 欧拉筛进行素数筛选

```
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (!st[i]) primes[cnt++] = i;
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] * i] = true;               //所有素数的倍数的数均为合数
            if (i % primes[j] == 0) break;
        }
    }
}
```

> 欧拉筛素数的主要思想就是所有素数的倍数的数均为合数，将那些数筛去即可

- 哈希函数

```
int findIt(string& str) {
    /*平方探测法进行相应位置观察*/
    int count = hasher(m, str);
    for (int i = 0; i <= m; i++) {
        int index = ((count + Square((i + 1) / 2 % m) * (i % 2 ? 1 : -1)) %
m + m) % m;
        if (vis[index] == 0) {
            return index;
        }
    }
    return -1;
}
```

  - 平法探测法：每次前后观察距离当前位置为平方距离的位置，观察是否那个位置为空，即观察$i+1^2, i-1^2, i+2^2, i-2^2 \cdots$位置是否为空，若为空即可返回相应位置的映射值
  - *根据相关资料，当哈希表的长度为素数 $4n+3$ 的形式时才可以有效减少哈希冲突的产生*

- 主函数调用

```
int main()
{
    get_primes(10010);
    cin >> n >> p;
    for (int i = 0; i < 10000; i++)
    {
        if (primes[i] >= p)
        {
            m = primes[i];
            break;
        }
    }


    string temp;
    while (n)
    {
        cin >> temp;
        int res = findIt(temp);
        if (res == -1)
            cout << "- ";
        else
        {
            cout << res << " ";
            vis[res] = 1;
        }
        n--;
    }
    return 0;
```

```
}
```

2.3.5 调试分析

- 关于平方定址法
- ```
  int finder(int oriKey, int mod)
  {
      if (vis[oriKey] == 0)
          return oriKey;
      else
      {
          int found = 0;
          int base = 1;
          int times = 0;
          while (!found)
          {
              if (times > mod)
                  break;

              if (vis[(oriKey + base * base) % mod] == 0)
              {
                  found = 1;
                  return (oriKey + base * base) % mod;
              }
              else if (vis[(oriKey - base * base) % mod] == 0)
              {
                  found = 1;
                  return (oriKey - base * base) % mod;
              }
              else
              {
                  base += 1;
                  times += 1;
              }
          }

          return -1;
      }
  }
  ```

2.3.6 总结和体会

- 练习用哈希表进行数据存储
- 用平方定址法进行哈希冲突的解决

2.4换座位
  ```

2.4.1 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。

2.4.2 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

2.4.3 数据结构设计

- 该题目和一维数组最少交换次数是相同的，
- 可以证明当遇到与标准排布不同的位置时，循环执行将当前位置的元素与其正确位置交换，直到当前位置正确。
- 这样交换的次数将会是最小的

证明：

- 其本质就是进行交换环移动的过程，这个交换环同时需要自己构造，结论
- **本文试图证明**：对于长度为 $N \geq 1$ 任意数列，若该数列有 $M(1 \leq M \leq N)$ 个交换环，则最小交换数 $F_N(M) = N - M$。
- 交换环
  - **交换环**：对于元素a_{ij}，其中 i 表示该元素**排序前的下标**，j 表示**排序后的下标**，若存在一个 $n(n > 0)$ 个元素序列 $\{a_{ij}\}$，满足：1. $j_n = i_1$；2. $j_k = i_{k+1}(1 \leq k < n)$，则称序列 $\{a_{ij}\}$ 为**交换环**。
- 引理

  引理1：对于 $N_{node} \geq 2$ 的交换环对任意两个不同节点进行交换后则成为两个交换环。

  引理2: 对于两个交换环，在每个交换环中各取一个节点，将两个节点进行交换后, 两个交换环合并为一个交换环。
- **引理3**: 对于有 N 个节点的交换环, 最少交换次数$F_N(1) = N - 1$。

  引理4：对于长度为 $N(N > 2)$ 的任意数列，若该数列有 $M(1 \leq M \leq N)$ 个交换环，则最少交换数 $F_N(M) = N - M$。
- 相关证明
- 该题目中进行移动时相当于构造了一个位置相差一次移动的交换环

2.4.4 功能说明

- 用`map`记录正确的位置，用访问数组记录该点是否访问过

```
 vector<vector<int>> vis;
vis.resize(210);
for (int i = 0; i < 200; i++)
{
    vis[i].resize(210);
}



int step = 0;
int n = old_chart.size();
int m = old_chart[0].size();


map<string, pair<int, int>> mp;


for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        mp.insert({ new_chart[i][j], {i, j} });
    }
}
```

> 即将相应字符映射到相应坐标

- 进行交换

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        while (old_chart[i][j] != new_chart[i][j])      //若当前位置不对，则一直交换
        {
            string temp = old_chart[i][j];

            int x = mp[temp].first;
            int y = mp[temp].second;
            swap(old_chart[x][y], old_chart[i][j]);
            step+;
        }
    }
}

return step;
```

2.4.5 调试分析

- 该题目与一维数组相同，均是求将数组恢复成有序所需的最小交换次数，注意进行交换的相关规则

3. 实验总结

- 顺序查找
- 二分查找
- 二叉排序树查找
- 二叉平衡树
    - 树的构建
    - 基础操作
        - 插入删除查看
    - 旋转操作
    - 元素重复次数
- 哈希表
    - 哈希冲突的解决
    - 平方寻址法