《数据结构与算法》实验报告

实验题目	二叉查找树、散列表实践		
实验时间	2023/12/15 9:00-12:00	**实验地点**	DS3402
实验成绩		**实验性质**	**☐验证性 √设计性 ☐综合性**

教师评价：

☐算法/实验过程正确； ☐源程序/实验内容提交 ☐程序结构/实验步骤合理；

☐实验结果正确； ☐语法、语义正确； ☐报告规范；

其他：

评价教师签名：

实验目的

1. 掌握 BST、AVL 树及散列表的基本概念和基本原理
2. 训练使用 BTS、AVL 和散列函数，通过编程解决不同难度问题的实践能力

二、实验项目内容

注：每道题按下面的格式分别描述

实验题目 1：寻找签到题

题目内容：在 ACM 程序设计竞赛赛场，当某个队伍 AC（正确解答）一道题目后就会在其前面升起 1 个彩色气球。而且每种颜色的气球只能用在一道题目上，所以不同颜色的气球不能相互替代。在某次比赛中，有 1 道最简单的题目（签到题），显然该题是被 AC 最多的。已知比赛过程中已送出的气球数量以及每个气球的颜色，请判断哪道题是签到题。若有多道题目的已送气球数相同，则认为字典序最小的颜色对应着签到题。

输入格式:

首先输入一个正整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试先输入一个整数 n（1≤n<1000000），代表已经送出的气球总数，然后输入 n 个已送出气球的颜色（由长度不超过 6 且不包含空格的英文字母组成），数据之间间隔一个空格。注意，统计时，忽略气球颜色的大小写。

报告创建时间：2023/12/24

输出格式:

对于每组测试，在一行上输出一个由小写字母构成的字符串，表示签到题对应的气球颜色。

代码：

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

int main() {
	int n , m;
	cin >> n;
	while (n--) {
		cin >> m;
		map<string, int> qq;
		while (m--) {
			string s;
			cin >> s;
			transform(s.begin(), s.end(), s.begin(), ::tolower);
			qq[s] += 1;
		}
		string ans;
		int max = 0;
		for (auto i = qq.begin(); i != qq.end(); i++) {
			if (i->second > max ) {
				max = i->second;
				ans = i->first;
			}
		}
		cout << ans << endl;
```

```
            qq.clear();
        }
        return 0;
    }
```

实验题目 2：航空公司 VIP 客户查询

题目内容：不少航空公司都会提供优惠的会员服务，当某顾客飞行里程累积达到一定数量后，可以使用里程积分直接兑换奖励机票或奖励升舱等服务。现给定某航空公司全体会员的飞行记录，要求实现根据身份证号码快速查询会员里程积分的功能。

输入格式:

输入首先给出两个正整数 N（≤10^5）和 K（≤500）。其中 K 是最低里程，即为照顾乘坐短程航班的会员,航空公司还会将航程低于 K 公里的航班也按 K 公里累积。随后 N 行，每行给出一条飞行记录。飞行记录的输入格式为：18 位身份证号码（空格）飞行里程。其中身份证号码由 17 位数字加最后一位校验码组成，校验码的取值范围为 0~9 和 x 共 11 个符号；飞行里程单位为公里，是(0, 15 000]区间内的整数。然后给出一个正整数 M（≤10^5），随后给出 M 行查询人的身份证号码。

输出格式:

对每个查询人，给出其当前的里程累积值。如果该人不是会员，则输出 No Info。每个查询结果占一行。

代码：

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

```
long long idtonum(string s) {
	long long res = 0;
	for (int i = 0; i < 17; i++)
		res = 10 * res + (s[i] - '0');
	if(s[17] == 'x') res = 10 * res + 10;
	else res = 10 * res + (s[17] - '0');
	return res;
}

int main() {
	map<long long, int>id;
	int n, k; cin >> n >> k;
	string sid; long long st;
	int num;
	for (int i = 0; i < n; i++) {
		cin >> sid;
		cin >> num;
		if (num < k)num = k;
		st = idtonum(sid);
		id[st] += num;
	}
	int m;
	cin >> m;
	for (int i = 0; i < m; i++) {
		cin >> sid;
		st = idtonum(sid);
		if (id.count(st)) {
			cout << id[st] << endl;
		}
		else cout << "No Info\n";
	}
```

}

实验题目 3：二叉搜索树的四种遍历

题目内容：这是一道模板题！

这里重新定义一下二叉搜索树：

所有节点的左孩子及其子树小于该节点，所有节点的右孩子及其子树大于等于该节点。

考虑到你可能对如何建立一颗二叉搜索树、如何对其插入节点、如何实现三种遍历不那么熟练，我们会在下面给出从零开始如何建立一颗二叉搜索树的完整 C++代码。

首先是结构体的定义，对于一颗 BST 节点，我们会在结构体里定义三个信息，分别是它的数据域和两个指针，分别指向左右孩子！

```
struct node {
    int data;
    node * left=NULL;
    node * right=NULL;
};
```

接下来是一个一个把节点插入到搜索树中！

```
void create (node * &root,int data) {
    if (root==NULL) {
        root=new node;
        root->data=data;
        root->left=NULL;
        root->right=NULL;
        return;
    }
    if (data<root->data) create( &root->left,data);
    else create( &root->right,data);
}
```

然后，我们只需要先定义一个根节点为空，再依次把要插入的元素作为参数运行这个 create 函数，就能得到一颗二叉搜索树啦！

接下来，我们对这颗 BST，做前序、中序、后序遍历，可以发现，这三种遍历只有输出的位置不同！

```
void preOrder (node * root) {
    if (root==NULL) return;
    pre.push_back(root->data);
    preOrder(root->left);
    preOrder(root->right);
}
void inOrder (node * root) {
    if (root==NULL) return;
    inOrder(root->left);
    in.push_back(root->data);
    inOrder(root->right);
}
void postOrder (node * root) {
    if (root==NULL) return;
    postOrder(root->left);
    postOrder(root->right);
    post.push_back(root->data);
}
```

下面展示层序遍历，这和另外三种遍历不同，层序遍历需要调用一个之前的数据结构——队列，这里的队列直接调用了 C++的 STL 库，对于擅长英语的你一定是很简单的！

```
void bfs (node * root) {
    queue<node *> q;
    q.push(root);
    while (!q.empty()) {
        node * u=q.front();
        q.pop();
        lev.push_back(u->data);
        if (u->left!=NULL) q.push(u->left);
        if (u->right!=NULL) q.push(u->right);
    }
}
```

上面的代码片段展示了从零开始，建立一颗二叉搜索树，并对其做四种遍历的过程！

现在给出一个二叉搜索树的插入序列，请你把序列中的数字依次插入到一颗空的二叉搜索树中，并输出它的前序、中序、后序、层序遍历！

输入格式:

第一行包含一个正整数 N（N<=1000）表示插入序列种元素的个数。

第二行包含 N 个整数，表示插入序列。(题目保证所有元素在 int 的范围内)

输出格式:

第一行输出 N 个整数，表示它的前序序列。

第二行输出 N 个整数，表示它的中序序列。

第三行输出 N 个整数，表示它的后序序列。

第四行输出 N 个整数，表示它的层序序列。

所有整数之间用一个空格分开，每行的末尾没有多余空格。

代码：

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

typedef struct BinarySortTree {
    int data;                                   //存放数据的数据域
    struct BinarySortTree* left;    //指针域 左指针
    struct BinarySortTree* right;  //指针域 右指针
    BinarySortTree(int x) : data(x), left(NULL), right(NULL) {}
}Node, * tree;
Node* root;//根节点
vector<int> pre;
vector<int> in;
vector<int> post;
vector<int> lev;

void insert(int key)
{
    Node* temp = root;//一个临时指针 用于移动
    Node* prev = NULL;//前一个结点
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        if (key < temp->data)
            temp = temp->left;
```

```
            else //if (key >= temp->data)
                temp = temp->right;
        }
        //此时就找到了待插入节点
        temp = new BinarySortTree(key);
        if (prev == NULL) {
            root = temp;
            return;
        }
        if (key < prev->data)
            prev->left = temp;
        else
            prev->right = temp;
    }
    void creat(tree &root, int key) {
        if (root == NULL) {
            root = new BinarySortTree(key);
            return;
        }
        if (key < root->data)
            creat(root->left, key);
        else
            creat(root->right, key);
    }
    void preorder(tree Tree) {
        if (Tree == NULL) return;
        pre.push_back(Tree->data);
        preorder(Tree->left);
        preorder(Tree->right);
    }
    void inorder(tree Tree) {
```

```
        if (Tree == NULL) return;
        inorder(Tree->left);
        in.push_back(Tree->data);
        inorder(Tree->right);
    }
    void postorder(tree Tree) {
        if (Tree == NULL) return;
        postorder(Tree->left);
        postorder(Tree->right);
        post.push_back(Tree->data);
    }
    void bfs(tree Tree) {
        queue<tree> q;
        q.push(Tree);
        while (!q.empty()) {
            tree u = q.front();
            q.pop();
            lev.push_back(u->data);
            if (u->left != NULL)  q.push(u->left);
            if (u->right != NULL)       q.push(u->right);
        }
    }
    void print(vector<int>& a) {
        int n = a.size();
        if(n)  cout << a[0];
        for (int i = 1; i < n; i++) {
            cout <<" " << a[i];
        }
        cout << "\n";
    }
```

```
int main() {
    int N;
    cin >> N;
    int temp;
    while (N--) {
        cin >> temp;
        //insert(temp);
        creat(root, temp);
    }
    preorder(root);
    inorder(root);
    postorder(root);
    bfs(root);
    print(pre); print(in); print(post); print(lev);
}
```

实验题目 4：平均查找长度之线性再散列

题目内容：对于指定长度为 N 的整数数组，存储于指定长度为 M 的散列表中，使用指定的散列函数（使用简单的模素数 P 运算），若使用线性再散列（这里指的是：若冲突找下一个空位置）处理冲突，编程计算查找成功时平均查找长度和查找失败时的平均查找长度。

代码：

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

int main() {
```

```
//n 数字个数，m 数组大小，p 给定素数
int n, m, p;
cin >> n >> m >> p;
int* a = new int[m];
bool* fd = new bool[m];
int b;
for (int i = 0; i < n; i++) {
	a[i] = 0;
	fd[i] = 0;
}
int sum1=0;
for (int i = 0; i < n; i++) {
	cin >> b;
	sum1++;
	if (!fd[b % p] && a[b % p] != b) {
		a[b % p] = b;
		fd[b % p] = 1;
	}
	else {
		int j = b % p;
		while (fd[j] && a[j] != b) { j = (j + 1) % m; sum1++; }
		a[j] = b;
		fd[j] = 1;
	}
}
int sum2 = 0;
for (int i = 0; i < p; i++) {
	for (int j = i; j < i + m;) {
		if (fd[j % m]) {
			sum2++;
			j++;
```

```
                }
                else {
                    sum2++;
                    break;
                }
            }
        }
        cout << sum1 << "/" << n << endl;
        cout << sum2 << "/" << p << endl;
    }
```

实验题目 5：insertion order（插入排序）

题目内容：A friend of yours is currently taking a class on algorithms and data structures. Just last week he learned about binary search trees and the importance of using self-balancing trees in order to keep the tree height low and guarantee fast access to every node.

你的一个朋友正在上一门算法和数据结构的课。就在上周，他学习了二叉搜索树和使用自平衡树的重要性保持树的高度较低，并保证快速访问每个节点。

Recall that a binary search tree is a binary tree with each node storing a key, and the property that the key of each node is greater than all keys in the left subtree of that node and less than all keys in the right subtree. A new key is inserted into the tree by adding a new leaf node with that key in the only position such that the property is maintained, as seen in the figure below.

回想一下，二叉搜索树是一棵二叉树，每个节点存储一个键和属性每个节点的键值大于该节点左子树中的所有键值小于所有键值键在右子树中。通过添加一个新的叶节点，将一个新键插入到树中在保持属性的唯一位置键入，如下图所示。

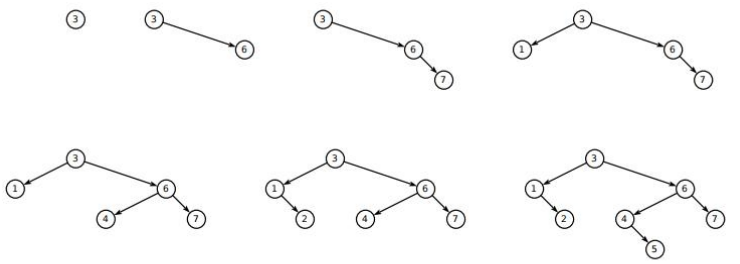

Figure 1: Illustration of the first sample case.

To illustrate to him just how bad things can get without self-balancing, you want to show him that it is possible to build trees of nearly any height by carefully choosing an insertion order.

为了向他说明，如果没有自我平衡，事情会变得多么糟糕，你要展示给他看通过仔细选择插入顺序，可以构建几乎任何高度的树。

You are given two integers n and k and want to construct a binary search tree with n nodes of height k (the height of a tree is the maximal number of nodes on a path from the root to a leaf).

To do so, you need to find a permutation of the integers from 1 to n such that, when they are inserted into an empty binary search tree in that order (without self-balancing), the resulting tree has height k.

已知两个整数 n 和 k 想要构造一个有 n 个结点的二叉搜索树高度 k(树的高度是从根到叶的路径上的最大节点数)。要做到这一点，你需要找到一个从 1 到 n 的整数的排列，当它们是按照这个顺序(不需要自平衡)插入到一个空的二叉搜索树中，生成的树有高度 k。

代码：

```
#include<iostream>
using namespace std;
int main(){
	int n,m;
	cin>>n>>m;
	if(n==7&&m==4){
		cout<<"4 7 6 5 3 2 1";
	}
	else if(n==1&&m==1)
		cout<<"1";
	else
		cout<<"impossible";
}
```