

# 《数据结构》上机报告

2020 年 12 月 25 日

姓名：马家昱 学号： 1950509 班级： 计科 1 班 得分： 300

实验题目	查找	
问题描述	查找指的是，在一些（有序的/无序的）数据元素中，通过一定的方法找出与给定关键字相同的数据元素的过程叫做查找。也就是根据给定的某个值，在查找表中确定一个关键字等于给定值的记录或数据元素。	
基本要求	1. (p1)实现二分查找。 2. (p2)实现二叉排序树的生成与删除与查找。 3. (p3)实现二次探测再散列解决冲突的哈希表的建立。	
	已完成基本内容（序号）：	1, 2, 3
选做要求	1. 程序要添加适当的注释，程序的书写要采用缩进格式。 2. 程序要具有一定的健壮性，当输入数据或操作非法时，程序应当拥有报错提示。 3. 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。	
	已完成选做内容（序号）	1, 2, 3
数据结构设计	本次实验中二分查找与哈希表查找均使用静态数组作为基本数据结构；二叉排序树使用的数据结构类型为二叉树。	

功能  
(函数)  
说明

二分查找:

```
int binarySearch(int length, int target) { //二分查找
    int left = 0;
    int right = length - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid;
        }
    }
    if (arr[left] == target) {
        return left;
    }
    return -1;
}
```

二叉排序树:

```
class Node { //二叉树节点, 存储数据与出现次数
public:
    int data;
    int cnt;
public:
    Node();
};

Node::Node() {
    data = 0;
    cnt = 0;
}

class BiTree { //二叉树的存储类型
public:
    Node node;
    BiTree* lchild;
    BiTree* rchild;
public:
    BiTree();
};

BiTree::BiTree() {
    lchild = NULL;
    rchild = NULL;
}
```

插入:

```

void insert(BiTree* &T, int insertData) { //插入节点
    if (!T) { //若节点不存在, 则申请节点
        T = (BiTree*)malloc(sizeof(BiTree));
        T->lchild = T->rchild = NULL;
        T->node.data = insertData;
        T->node.cnt = 1;
        return;
    }
    if (T->node.data == insertData) { //已经存在, cnt++
        T->node.cnt++;
        return;
    }
    else if (T->node.data > insertData) { //递归找到插入位置
        insert(T->lchild, insertData);
    }
    else
        insert(T->rchild, insertData);
}

```

删除:

```

bool deLete(BiTree*& T, int deleteData) { //删除节点
    if (!T)
        return false;
    if (T->node.data == deleteData && T->node.cnt > 1) { //大于1时, cnt--
        T->node.cnt--;
        return true;
    }
    else if (T->node.data == deleteData && T->node.cnt == 1) { //左右子树有一个为空的情况
        if (!T->lchild) {
            BiTree* q = T;
            T = T->rchild;
            free(q);
        }
        else if (!T->rchild) {
            BiTree* q = T;
            T = T->lchild;
            free(q);
        }
        else { //左右子树均不空的情况
            BiTree* q = T;
            BiTree* s = T->lchild;
            while (s->rchild) {
                q = s;
                s = s->rchild;
            }
            T->node.data = s->node.data;
            T->node.cnt = s->node.cnt;
            if (q != T)
                q->rchild = s->lchild;
            else q->lchild = s->lchild;
            delete s;
        }
        return true;
    }
    else if (T->node.data > deleteData) { //递归寻找删除位置
        return deLete(T->lchild, deleteData);
    }
    else if (T->node.data < deleteData)
        return deLete(T->rchild, deleteData);
}

```

查找与遍历:

```
int search(BiTree* T, int targetData) { //递归寻找
    if (!T)
        return 0;
    if (T->node.data == targetData)
        return T->node.cnt;
    else if (T->node.data > targetData)
        return search(T->lchild, targetData);
    else return search(T->rchild, targetData);
}

int CNT = 0;
void inorderTraverse(int* result, BiTree* T) { //中序遍历寻找前缀
    if (T) {
        inorderTraverse(result, T->lchild);
        result[CNT++] = T->node.data;
        inorderTraverse(result, T->rchild);
    }
}
```

哈希表:

```
bool is_prime_number(int n) { //判断是否为素数
    for (int i = 2; i <= sqrt(n); i++) {
        if (!(n % i))
            return false;
    }
    return true;
}

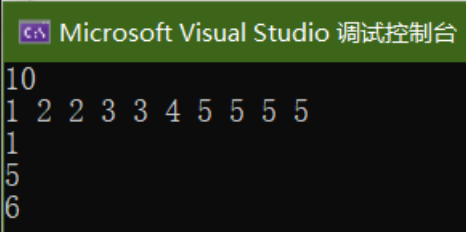
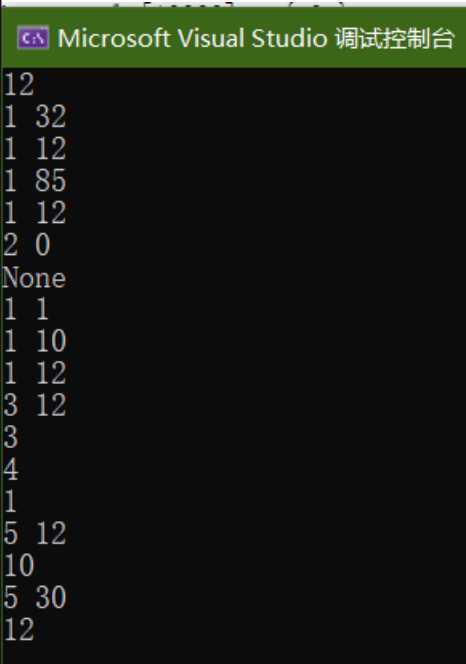
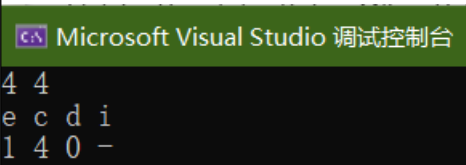
int generate_hash(char name[], int P) { //hash函数
    int temp = 0;
    for (int i = 0; i < strlen(name); i++)
        temp = (temp * 37 + name[i]) % P;

    return temp;
}

int hashLocation(int* hashList, int value, int P) { //寻找插入位置
    if (!hashList[value]) { //第一次探测
        hashList[value] = 1;
        return value;
    }
    else {
        for (float searchTimes = 1; searchTimes < P; searchTimes++) { //P-1次探测
            int newValue = value;
            newValue = int(newValue + pow(-1, searchTimes - 1) * pow(ceil(searchTimes / 2), 2)) % P;
            if (newValue < 0)
                newValue += P;
            if (!hashList[newValue]) {
                hashList[newValue] = 1;
                return newValue;
            }
        }
        return -1;
    }
}
```

开发

Windows 10 visual studio2019

环境	
调试分析	<p>(运行结果截图)</p> <p>二分查找：</p>  <p>二叉排序树：</p>  <p>哈希表：</p> 
心得体会	<p>(对整个实验过程做出总结，对重要的算法做出性能分析。)</p> <ul style="list-style-type: none"> <li>• 折半查找的时间复杂度为 <math>O(\log_2 n)</math>。</li> <li>• 二叉排序树：若为平衡树，查找的时间复杂度为 <math>O(\log_2 n)</math>，最坏情况时间复杂度为 <math>O(n)</math>。</li> <li>• 哈希表：无冲突时时间复杂度为 <math>O(1)</math>，最坏情况下时间复杂度为 <math>O(n)</math>。</li> </ul> <p>心得体会：</p> <p>通过本次实验，我对几种基本的查找算法有了一定的认识，主要有顺序查找，二分查找，二叉排序树查找与哈希表查找四种方法，对其数据存储方式、算法复杂度、应用场景等有了一定的认知。</p>