《数据结构》上机报告

2020 年 12 月 24 日

姓名： 王上游　　学号：1850767　班级：　　19 计科 2 班　　得分：

实验题目	查找算法实验报告
实验目的	1、 掌握各种查找算法的实现； 2、掌握各种查找算法的特点，时间复杂度、稳定性。
问题描述	（1） 折半查找 二分法将所有元素所在区间分成两个子区间，根据计算要求决定下一步计算是在左区间还是右区间进行；重复该过程，直到找到解为止。二分法的计算效率是 O(logn),在很多算法中都采用了二分法，例如：折半查找，快速排序，归并排序等。 折半查找要求查找表是有序排列的，本题给定已排序的一组整数，包含重复元素，请改写折半查找算法，找出关键字 key 在有序表中出现的第一个位置（下标最小的位置），保证时间代价是 O(logn)。若查找不到，返回-1。 （2） 二叉排序树 二叉排序树 BST（二叉查找树）是一种动态查找表，或者是一棵空树，或者是具有下列性质的二叉树： 1. 每个结点都有一个作为查找依据的关键字(key)，所有关键字的键值互不相等。 2. 左子树(若非空)上所有结点的键值都小于它的根结点的键值。 3. 右子树(若非空)上所有结点的键值都大于它的根结点的键值。 4. 左子树和右子树也是二叉排序树。 二叉排序树的基本操作集包括：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。

<table>
<tr><td></td><td>（3） 哈希表

哈希表（hash table，散列表）是一种用于以常数平均时间执行插入、删除和查找的查找表，其基本思想是：找到一个从关键字到查找表的地址的映射 h（称为散列函数），将关键字 key 的元素存到 h(key)所指示的存储单元中。当两个不相等的关键字被散列到同一个值时称为冲突，产生冲突的两个（或多个）关键字称为同义词，冲突处理的方法主要有：开放定址法，再哈希法，链地址法。

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：
1. 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)。

2. 采取除留余数法将整数映射到长度为 P 的散列表，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。

3. 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）</td></tr>
<tr><td>基本要求</td><td>1. 程序要添加适当的注释，程序的书写要采用 缩进格式 。

2. 程序要具在一定的 健壮性，即当输入数据非法时， 程序也能适当地做出反应，如 插入删除时指定的位置不对 等等。

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。

4. 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。</td></tr>
<tr><td>选做要求</td><td>无
<table><tr><td>已完成选做内容（序号）</td><td>无</td></tr></table></td></tr>
</table>

<table>
<tr>
<td>数据结构设计</td>
<td>
（1）折半查找
直接采用 int 数组

（2）二叉排序树
建立了链式二叉树，在此基础上实现二叉排序树
<pre>
struct node
{
    int value;
    int count;
    node* lchild;
    node* rchild;
};

typedef struct node* BST;
</pre>
（3）哈希表
不需要记录数据，仅需要记录空间是否冲突，因此int数组记录哈希表相应位置是否已经被占据
<pre>
int HashTable[10008];
</pre>
</td>
</tr>
<tr>
<td>功能（函数）说明</td>
<td>
（1）折半查找

题目要求简单，因此直接在 main 函数中实现二分查找。

（2）二叉排序树

<pre>
void insert(BST& root, int newvalue);
//以root为根的树中插入新值newvalue

node* GetPri(BST root, int v);
//以root为根的树中寻找比v小的最大值

bool erase(BST& root, int v);
//在以root为根的二叉排序树中删除值为v的节点

node* Getmin(BST root);
//在以root为根的二叉排序树中取最小值

int GetCount(BST root, int v);
//在以root为根的二叉排序树中取v的出现次数
</pre>
（3）哈希表
不需要记录数据，仅需要记录空间是否冲突，因此int数组记录哈希表相应
</td>
</tr>
</table>

	位置是否已经被占据
	`int myhash(const char s[10005]);`
	`//根据规则生成串s的散列值，冲突检测，输出答案`
	
	`bool isprime(int n);`
	`//检测素性`

界面设计和使用说明	OJ 练习题，无界面，略
调试分析	（1） 折半查找 注意 mid 的计算方法以及移动 l,r 缩小一半范围的方法。防止死循环。 （2） 二叉排序树 二叉排序树删除节点的操作有一定难度。可以采用删去直接前驱的办法。待删节点不是物理上的被删节点。将中序下直接前驱结点的数据复制到待删结点，物理上删除前驱，重接子树。 （3） 哈希表 采用开放定址法，二次探测再散列方法处理冲突。注意对哈希值取模加模，防止散列值越界或为负值。
心得体会	（对整个实验过程做出总结，对重要的算法做出性能分析。） 三道题遇到了一些 bug，不过很快解决了。这次作业让我对查找算法有了较好的掌握。 （1） 折半查找 二分法查找，复杂度 O(logn)。 （2） 二叉排序树

	查找、插入、删除的复杂度与树高成一次线性。因此查找、插入、删除的最优复杂度和平均复杂度均为 O(logn)。若不做优化，最坏情况下二叉排序树退化为有序链表，其插入、删除、查找的最坏复杂度即为 O(n)。 （3） 哈希表 开放定址法，二次探测再散列方法处理冲突的平均查找长度为$-\frac{1}{a}\ln(1-a)$。其中 a 是装填因子，a = N/P。因此，若设计合理，哈希表的最优和平均插入和查找复杂度为常数级。哈希表的最坏复杂度是 O(n)。此时所有散列值全部冲突，退化为线性表。本题中由于对字符串每一位操作，复杂度为 O(n)，n 为字符串长度。

代码实现	（1）折半查找
	`#define _CRT_SECURE_NO_WARNINGS`
	`#include <iostream>`
	`#include <cstdio>`
	`using namespace std;`
	
	`#ifdef _MSC_VER`
	`#pragma warning(disable:6031)`
	`#endif`
	
	`int a[100005];`
	
	`int main()`
	`{`
	`  int n, k;`
	`  scanf("%d", &n);`
	`  for (int i = 0; i < n; i++)`
	`  scanf("%d", a + i);`
	`  scanf("%d", &k);`
	`  for (int i = 0; i < k; i++)`
	`  {`
	`  int x;`
	`  scanf("%d", &x);`
	`  int l = 0;`
	`  int r = n - 1;`
	`  while (l < r)`
	`  {`
	`  int mid = l + (r - l) / 2;`
	`  if (a[mid] >= x)`
	`  r = mid;`
	`  else`
	`  l = mid + 1;`

```
        }
        printf("%d\n", a[l] == x ? l : -1);
    }
    return 0;
    }
```

（2）二叉排序树

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

#ifdef _MSC_VER
#pragma warning(disable:6031)
#endif

struct node
{
    int value;
    int count;
    node* lchild;
    node* rchild;
};

typedef struct node* BST;

void insert(BST& root, int newvalue)
{
    if (!root)//建立新结点
    {
        root = new node;
        root->value = newvalue;
        root->count = 1;
        root->lchild = root->rchild = NULL;
    }
    else
    {
        if (newvalue < root->value)//在左子树插入
            insert(root->lchild, newvalue);
        else if (newvalue > root->value)//在右子树插入
            insert(root->rchild, newvalue);
        else//该值已有节点
        {
```

```
                root->count++;
            }
        }
    }

node* GetPri(BST root, int v)//以root为根的树中寻找比v小的最大值
{
    if (!root)return NULL;

    if (root->value >= v)//在左子树寻找
        GetPri(root->lchild, v);
    else
    {
        node* temp = GetPri(root->rchild, v);//在右子树寻找
        if (temp)
            return temp;
        else//右子树没找到，当前节点就是答案
            return root;
    }
}

bool erase(BST& root, int v)//在以root为根的二叉排序树中删除值为v的
节点
{
    if (!root)
        return false;

    if (root->value > v)//左子树中删除
        return erase(root->lchild, v);
    else if (root->value < v)//右子树中删除
        return erase(root->rchild, v);
    else
    {
        if (root->count > 1)//数量大于1，不是真删除节点，数量-1
        {
            root->count--;
        }
        else//要删除节点
        {
            node* t = root;
            if (!root->rchild)//右子树空，接左子树即可
            {
                root = root->lchild;
                delete t;
```

```
                }
                else if (!root->lchild)//左子树空，接右子树即可
                {
                        root = root->rchild;
                        delete t;
                }
                else//都不空
                {
                        node* s = root->lchild;
                        while (s->rchild)
                        {
                                t = s;
                                s = s->rchild;
                        }//s：在左子树中找被删节点的直接前驱，s是一个没有右子树的节点
                        root->count = s->count;
                        root->value = s->value;
                        //s的数据复制到被删除节点的位置
                        if (t != root)//t是原本s的双亲
                                t->rchild = s->lchild;
                        else//s就是当初被删节点的左子树，直接接上
                                t->lchild = s->lchild;

                        delete s;//s数据已经移动到被删除位置代替，原s被删
                }
        }
        return true;
    }
}

node* Getmin(BST root)//在以root为根的二叉排序树中取最小值
{
    node* p = root;
    while (p->lchild)
        p = p->lchild;
    return p;
}

int GetCount(BST root, int v)//在以root为根的二叉排序树中取v的出现次数
{
    if (!root)
        return 0;
```

```
    if (root->value == v)
        return root->count;
    else if (v < root->value)
        return GetCount(root->lchild, v);
    else
        return GetCount(root->rchild, v);
}

int main()
{
    BST bst = NULL;
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        int op;
        scanf("%d", &op);
        int x;
        if (op != 4)
            scanf("%d", &x);
        if (op == 1)
        {
            insert(bst, x);
        }
        else if (op == 2)
        {
            if (!erase(bst, x))
                printf("None\n");
        }
        else if (op == 3)
            printf("%d\n", GetCount(bst, x));
        else if (op == 4 && bst)
            printf("%d\n", Getmin(bst)->value);
        else if (op == 5)
        {
            node* t = GetPri(bst, x);
            if (t)
                printf("%d\n", t->value);
            else
                printf("None\n");
        }
    }
    return 0;
}
```

（3）哈希表

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

#ifdef _MSC_VER
#pragma warning(disable:6031)
#endif

int N, P, M, K;
int HashTable[10008];
char s[10005];

bool isprime(int n)
{
	for (int i = 2; i * i <= n; i++)
		if (n % i == 0)
			return false;
	return true;
}

int myhash(const char s[10005])
{
	int h = 0;
	int i = 0;
	while (s[i])
	{
		h = h * 37 + int(s[i++]);
		h %= M;
	}
	if (!HashTable[h])
	{
		HashTable[h] = 1;
		return h;
	}
	else
	{
		int newh;
		for (int i = 1; i <= K; i++)
		{
			newh = (h + i * i) % M;
```

```
                if (!HashTable[newh])
                {
                    HashTable[newh] = 1;
                    return newh;
                }
                newh = (h - i * i + M * i) % M;
                if (!HashTable[newh])
                {
                    HashTable[newh] = 1;
                    return newh;
                }
            }
            return -1;
        }

}

int main()
{
    ios::sync_with_stdio(false);
    cin >> N >> P;
    M = P;
    while (!isprime(M))M++;
    K = M / 2;
    memset(HashTable, 0, sizeof(HashTable));
    for (int i = 0; i < N; i++)
    {
        cin >> s;
        int ans = myhash(s);
        if (ans == -1)
            cout << "-";
        else
            cout << ans;
        cout << " ";
    }
    return 0;
}
```