

**实验报告**



**题目： 实验二：树-查找综合应用实验**

**班 级： 2023211314**

**学 号： 2023211304**

**姓 名： 唐术洋**

**学 院： 计算机学院**

**2024年 11 月 21 日**

1. 实验目的
2. 练习二叉树的实现及操作，学习用二叉树解决实际问题；
3. 练习查找树的实现及操作，学习用查找树解决实际问题；
4. 锻炼递归算法编写及程序调试的能力；
5. 学习自己查找相关资料以解决实际问题的能力。

二、实验环境

Windows,VsCode,gcc

三、实验内容

**场景设定：**

假设某超市需要实现一个商品库存管理系统。每种商品有唯一的ID，初始的10k种商品样例数据如附件csv所示。文件第一行为表头，后续每行为一种商品，包含Uniq Id（唯一id）,Product Name（商品名称）,Inventory（库存）三列。因为有大量的查询、添加、删除操作，计划采用查找树作为其核心数据结构。请在此场景设定下完成以下要求。

**要求：**

1. 读取marketing\_sample\_10k\_data.csv文件，依次插入10k项商品，构造出对应的二叉查找树。
2. 求出该二叉查找树的高度。
3. 求针对这10k项商品的平均查找长度。
4. 采用非递归的方式中序遍历该二叉树，依次输出商品的Uniq Id。
5. 读取drop sample\_1k\_data.csv中的1000项需要删除的商品id清单，依次从二叉查找树中将其删除。
6. 求删除后的二叉查找树的高度。

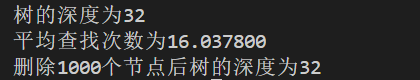
**附加要求**：将要求第1步中构造二叉查找树，优化成构造平衡二叉树。不要求实现平衡二叉树结点的删除。

1. 实验步骤（基础80分+附加10分）

结果如图：（图为要求2，3，6）（要求4见文本文件“ID1.txt”，要求5见文本文件“ID2.txt”）

（删除树的节点后将右子树的最左叶节点上移作为被删除节点的替换）

（排序的依据为id字符串的大小）



**递归算法设计思想：**

1. **递归求深度：**

一棵树的深度==MAX{其左子树的深度，其右子树的深度}+1。

如果当前节点为叶子节点，则其深度为1.

若为空节点（NULL），则深度为0.

1. **递归求总查找次数：**

总查找次数**==**（根节点层次为1）

每访问一个节点时，将其深度加到 totalSearchTimes 中，直到遍历完整棵树。

当前节点为空节点（NULL）时返回 0。递归步骤是分别处理左子树和右子树，并将深度逐层增加。

1. **递归查找关键值为key的节点：**

前提：结合实验文件可以得到树中必存在我们需要查找的关键值为key的节点。

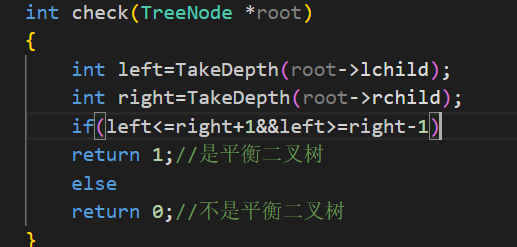
若根节点不是我们要查找的节点，那么其必存在于其左子树或者右子树中，结合二叉排序树的性质（小在左，大在右），将问题分解为分别在其左子树和右子树中查找，查找到便返回指向该节点的指针。

1. **删除二叉排序树节点中的递归：**

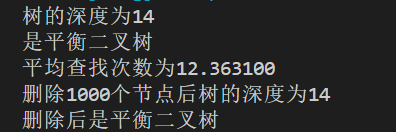
得到要删除的节点的关键值后，若目标节点关键值小于当前节点关键值，则进入其左子树删除。若大于当前节点关键值，则进入其右子树删除。由此递归调用deleteNode函数，分别传参左孩子和右孩子。

**附加要求：（因为在文件的输出结果与基础要求完全一致，故不再另外提交）**

**平衡二叉树判断函数截图：**



**运行结果截图：**



**将二叉排序树优化为平衡二叉树的优化思路：**

**核心：分治思想，递归构造**

因为平衡二叉树的任意一个节点的左右子树高度差不超过1，因此构造时将已排序好的treenode型数组从中间隔断，最中间的为根节点，左侧部分用于构造左子树，右侧部分用于构造右子树，由此开始递归。当左侧数组下标大于右侧数组下标时返回NULL。（表示数组为空）

五、实验分析和总结（10分）

实验思路大致为：先构造出所需的二叉排序树，再分部分编写功能函数，最后将功能函数的反馈整合为最终结果。

设想及效果：

编写程序的时候我一直认为我写了这么多递归函数那么程序运行的速度一定比较慢（因为我认为10k的数据量算大，而且递归函数的运行时间随操作对象数增长而增长的速度非常快），结果运行时间不算很慢（不到1s）。

完全没想到平衡二叉树能够优化那么多，树的深度减了超过一半，平均查找长度也减了差不多4。对于平衡二叉树的优化作用有了更清晰的认识。

总结心得：本次实验让我从实践层面熟悉了二叉排序树的相关操作，以及增强了我编写递归算法程序与调试程序的能力。

1. 程序源代码（10分）

基础要求代码：

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

typedef struct node

{

char id[35];

char name[250];

int inventory;

struct node \*lchild;

struct node \*rchild;

} TreeNode;

typedef struct stack

{

TreeNode \*data[1000]; // 栈的最大深度为1000

int top;

} Stack;

void InitStack(Stack \*s) // 初始化栈

{

s->top = -1;

}

int IsEmpty(Stack \*s) // 判断栈是否为空

{

return s->top == -1;

}

void push(Stack \*s, TreeNode \*node) // 入栈

{

if (s->top >= 999)

{

fprintf(stderr, "Stack overflow\n");

exit(1);

}

s->data[++(s->top)] = node;

}

TreeNode \*pop(Stack \*s) // 出栈

{

if (IsEmpty(s))

{

fprintf(stderr, "Stack underflow\n");

exit(1);

}

return s->data[(s->top)--];

}

int TakeDepth(TreeNode \*root) // 统计高度/深度

{

if (root == NULL)

return 0;

else

{

int left = TakeDepth(root->lchild);

int right = TakeDepth(root->rchild);

return left > right ? left + 1 : right + 1;

}

}

void NonRecursiveInorderTraversal(TreeNode \*root, FILE \*fp) // 非递归中序遍历二叉树

{

Stack s;

InitStack(&s);

TreeNode \*current = root;

fprintf(fp, "Uniq ID:\n");

while (current != NULL || !IsEmpty(&s))

{

// 将当前节点的所有左子节点入栈

while (current != NULL)

{

push(&s, current);

current = current->lchild;

}

// 出栈并访问节点

current = pop(&s);

fprintf(fp, "%s\n", current->id);

// 访问右子节点

current = current->rchild;

}

}

// int check(TreeNode \*root)

// {

// int left=TakeDepth(root->lchild);

// int right=TakeDepth(root->rchild);

// if(left<=right+1&&left>=right-1)

// return 1;

// else

// return 0;

// }

TreeNode \*ReadCSVandCreateNode(FILE \*fp) // 读CSV数据返回生成的节点指针

{

TreeNode \*p = (TreeNode \*)malloc(sizeof(TreeNode));

p->lchild = NULL;

p->rchild = NULL;

char line[1024];

char \*h = fgets(line, 1024, fp);

int i = 0;

if (h != NULL)

{

strcpy(p->id, strtok(line, ","));

strcpy(p->name, strtok(NULL, ","));

p->inventory = 100;

return p;

}

else

return NULL;

}

TreeNode \*insert(TreeNode \*root, TreeNode \*p)

{

if (root == NULL)

{

return p; // 如果当前子树为空，创建新节点

}

if (strcmp(root->id, p->id) > 0)

{

root->lchild = insert(root->lchild, p); // 插入左子树

}

else

{

root->rchild = insert(root->rchild, p); // 插入右子树

}

return root;

}

TreeNode \*SearchBST(TreeNode \*T, char \*key)

// 在二叉排序树T中查找关键字值为 key 的结点，

// 找到返回该结点的地址，否则返回空。

{

if ((!T) || strcmp(key, T->id) == 0)

return (T); // T为空或者查找成功

else if (strcmp(key, T->id) < 0)

return (SearchBST(T->lchild, key));

else

return (SearchBST(T->rchild, key));

}

TreeNode \*FindMin(TreeNode \*node)

{

while (node->lchild != NULL)

{

node = node->lchild;

}

return node;

}

TreeNode \*DeleteNode(TreeNode \*root, char \*key)

{

// 首先查找要删除的节点

TreeNode \*target = SearchBST(root, key);

if (target == NULL)

{

printf("Node with key %s not found.\n", key);

return root; // 节点不存在

}

// 找到要删除的节点后，开始删除操作

if (root == NULL)

{

return root; // 树为空或未找到节点

}

// 如果要删除的节点在左子树中

if (strcmp(key, root->id) < 0)

{

root->lchild = DeleteNode(root->lchild, key);

}

// 如果要删除的节点在右子树中

else if (strcmp(key, root->id) > 0)

{

root->rchild = DeleteNode(root->rchild, key);

}

// 找到了要删除的节点

else

{

// 节点是叶子节点

if (root->lchild == NULL && root->rchild == NULL)

{

free(root);

root = NULL;

}

// 节点有一个子节点

else if (root->lchild == NULL)

{

TreeNode \*temp = root;

root = root->rchild;

free(temp);

}

else if (root->rchild == NULL)

{

TreeNode \*temp = root;

root = root->lchild;

free(temp);

}

// 节点有两个子节点

else

{

TreeNode \*temp = FindMin(root->rchild); // 找到右子树最左节点

strcpy(root->id, temp->id);

strcpy(root->name, temp->name);

root->inventory = temp->inventory;

root->rchild = DeleteNode(root->rchild, temp->id);

}

}

return root;

}

int TakeNodeTimes(TreeNode \*root, int depth, int \*totalSearchTimes) // 统计总查找次数

{

if (root == NULL)

return 0;

\*totalSearchTimes += depth;

TakeNodeTimes(root->lchild, depth + 1, totalSearchTimes);

TakeNodeTimes(root->rchild, depth + 1, totalSearchTimes);

return \*totalSearchTimes;

}

double TakeTheAverageTimes(TreeNode \*root) // 计算平均次数

{

int totalSearchTimes = 0;

int NodeCount = 10000;

return (double)TakeNodeTimes(root, 1, &totalSearchTimes) / NodeCount;

}

TreeNode \*CreateBinaryTree(FILE \*fp)

{ // 递归构建二叉排序树

TreeNode \*root = NULL;

TreeNode \*p = NULL;

root = ReadCSVandCreateNode(fp);

while ((p = ReadCSVandCreateNode(fp)) != NULL)

{

root = insert(root, p);

}

return root;

}

int main()

{

FILE \*fp = fopen("marketing\_sample\_10k\_data.csv", "r");

if (fp == NULL)

{

printf("Error opening file.\n");

return 1;

}

TreeNode \*root = CreateBinaryTree(fp);

fclose(fp);

FILE \*f1 = fopen("ID1.txt", "w");

NonRecursiveInorderTraversal(root, f1);

fclose(f1);

printf("树的深度为%d\n", TakeDepth(root));

double average = TakeTheAverageTimes(root);

printf("平均查找次数为%lf\n", average);

FILE \*pf = fopen("drop sample\_1k\_data.csv", "r");

char line[35];

while (fgets(line, 35, pf) != NULL)

{

line[strlen(line) - 1] = '\0';

root = DeleteNode(root, line);

}

FILE \*f2 = fopen("ID2.txt", "w");

NonRecursiveInorderTraversal(root, f2);

fclose(f2);

printf("删除1000个节点后树的深度为%d\n", TakeDepth(root));

return 0;

}

**附加要求代码：（仅提交与基础部分代码不同的部分，重写main函数和部分构造树的函数）**

TreeNode\* CreateNode(TreeNode p)

{

TreeNode\* h=(TreeNode\*)malloc(sizeof(TreeNode));

h->lchild=NULL;

h->rchild=NULL;

strcpy(h->id,p.id);

strcpy(h->name,p.name);

h->inventory=100;

return h;

}

TreeNode\* CreateBalancedBST(TreeNode\* a, int start, int end) {

if (start > end) {

return NULL; // 子数组为空，返回NULL

}

// 取中间元素作为根节点

int mid = (start + end) / 2;

TreeNode\* root = CreateNode(a[mid]);

// 递归构建左子树

root->lchild = CreateBalancedBST(a, start, mid - 1);

// 递归构建右子树

root->rchild = CreateBalancedBST(a, mid + 1, end);

return root;

}

int cmp(const void \*a, const void \*b)//qsort库函数的比较函数

{

return strcmp(((TreeNode \*)a)->id, ((TreeNode \*)b)->id);

}

int main()

{

FILE \*fp = fopen("marketing\_sample\_10k\_data.csv", "r");

if (fp == NULL)

{

printf("Error opening file.\n");

return 1;

}

TreeNode\* a=(TreeNode\*)malloc(sizeof(TreeNode)\*10000);

for(int i=0;i<=9999;i++)

{

char line[512];

char\* h=fgets(line,512,fp);

strcpy(a[i].id,strtok(line,","));

strcpy(a[i].name,strtok(NULL,","));

a[i].inventory=100;

}

//排序部分

qsort(a,10000,sizeof(TreeNode),cmp);

TreeNode \*root = CreateBalancedBST(a,0,9999);

fclose(fp);

FILE \*f1 = fopen("ID1.txt", "w");

NonRecursiveInorderTraversal(root, f1);

fclose(f1);

printf("树的深度为%d\n", TakeDepth(root));

if(check(root))

printf("是平衡二叉树\n");

else

printf("不是平衡二叉树\n");

double average = TakeTheAverageTimes(root);

printf("平均查找次数为%lf\n", average);

FILE \*pf = fopen("drop sample\_1k\_data.csv", "r");

char line[35];

while (fgets(line, 35, pf) != NULL)

{

line[strlen(line) - 1] = '\0';

root = DeleteNode(root, line);

}

FILE \*f2 = fopen("ID2.txt", "w");

NonRecursiveInorderTraversal(root, f2);

fclose(f2);

printf("删除1000个节点后树的深度为%d\n", TakeDepth(root));

if(check(root))

printf("删除后是平衡二叉树\n");

else

printf("删除后不是平衡二叉树\n");

return 0;

}