北京郵電大學

实验报告

题目：实验二：树-查找综合应用实验

班　　级：2023211XXX

学　　号：2023XXXXXXX

姓　　名：Yokumi

学　　院：计算机学院

2024 年 11 月 22 日

一、实验目的

1、 练习二叉树的实现及操作，学习用二叉树解决实际问题；

2、 练习查找树的实现及操作，学习用查找树解决实际问题；

3、 锻炼递归算法编写及程序调试的能力；

4、 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

操作系统为 MacOS 14.5 (23F79)，处理器架构为 arm64。C 标准采用 c17，编译器为 clang-1500.3.9.4。

```
(base)  yokumi@YokumideMacBook-Air  ~  sw_vers
ProductName:            macOS
ProductVersion:         14.5
BuildVersion:           23F79
```

图 1 操作系统

```
(base)  yokumi@YokumideMacBook-Air  ~  clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: arm64-apple-darwin23.5.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

图 2 编译器环境

工作目录结构如下：

```
├─DS
    ├─Lab2
        ├─lab2.c    // 源代码
        ├─lab2_extra.c    // 附加内容源代码
        ├─marketing_sample_10k_data.csv    // 商品目录
        ├─drop sample_1k_data.csv    // 删除商品清单
```

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。

四、实验步骤（基础 80 分+附加 10 分）

4.1 高层数据结构设计

4.1.1 二叉查找树

二叉搜索树（Binary Search Tree），也称为二叉查找树、有序二叉树（Ordered Binary Tree）或排序二叉树（Sorted Binary Tree），是指一棵空树或者具有下列性质的二叉树：

- 若任意节点的左子树不为空，则左子树上所有节点的值均小于它的根节点的值；
- 若任意节点的右子树不为空，则右子树上所有节点的值均大于或等于 它的根节点的值；
- 任意节点的左、右子树也分别为二叉搜索树；
- 没有键值相等的节点；

二叉搜索树的优点是，即便在最坏的情况下，也允许你在 O(h) 的时间复杂度内执行所有的搜索、插入、删除操作，其中 h 为二叉搜索树的高度。

通常来说，如果需要有序地存储数据或者需要同时执行搜索、插入、删除等多步操作，二叉搜索树是一个很好的选择。

对于实验中的场景，商品库存管理系统包含大量商品，且商品 ID 不重复，对于库存管理需要进行查找、更新、插入、删除等操作，所以，二叉搜索树这种数据结构非常合适。

观察实验提供的商品信息文件可以发现，商品 ID 是一个字符串，且长度均为 32；商品名称不固定，但长度均不超过 255；库存则为一个整型变量。

针对本题设计的二叉搜索树的树节点（BstNode）结构体如下：

```
1. typedef struct BstNode {
2.     char ID[33];                    // 商品 ID
3.     char Name[256];                 // 商品名称
4.     int inventory;                  // 商品库存
5.     struct BstNode *left, *right;  // 左右子树指针
```

```
6. } BstNode;
```

涉及对于二叉搜索树的操作将在下一部分展开。

4.1.2 平衡二叉树

对于二叉查找树，由于插入顺序不同，可能导致其查找效率较差（比如，均插在右子树上，最坏情况下需要查找 N 次，即此时二叉查找树高度等于节点个数），所以，如果能让二叉查找树的高度达到平衡，那么，一个有 N 个节点的平衡二叉搜索树，一个具有 n 个结点的平衡二叉树的高度 h 满足 $\log_2(n+1) \le h \le \log_a(\sqrt{5}(n+1)) - 2$，$a = \frac{1+\sqrt{5}}{2}$。

在最坏情况下，AVL 树的高度约为$1.44\log_2 n$，而完全平衡的二叉树高度约为$\log_2 n$，因此 AVL 树是接近最优的。

具有 N 个节点的二叉搜索树的高度在 logN 到 N 区间变化。也就是说，搜索操作的时间复杂度可以从 logN 变化到 N。这会造成巨大的性能差异。因此高度平衡的二叉搜索树对提高性能起着重要作用。

本实验中将采用 AVL 树对查找效率进行优化，以下是 AVL 树的树节点的结构体定义：

```
1. typedef struct AVLNode {
2.     char ID[33];                    // 商品 ID
3.     char Name[256];                 // 商品名称
4.     int inventory;                  // 商品库存
5.     int height;                     // 节点高度
6.     struct AVLNode* left;           // 左右子树指针
7.     struct AVLNode* right;
8. } AVLNode;
```

平衡因子(balance factor），定义为该节点的左子树的高度 - 右子树的高度。

以下分别是计算和更新节点高度、计算平衡因子的代码：

```
1. // 获取节点高度
2. int height(AVLNode* node) {
3.     // 空节点高度为 -1 ，叶节点高度为 0
4.     if (node != NULL) {
5.         return node->height;
6.     }
7.     return -1;
8. }
9.
10. // 更新节点高度
11. void updateHeight(AVLNode *node) {
12.     int lh = height(node->left);
13.     int rh = height(node->right);
14.     // 节点高度等于最高子树高度 + 1
```

```
    if (lh > rh) {
        node->height = lh + 1;
    } else {
        node->height = rh + 1;
    }
}

// 计算平衡因子
int balanceFactor(AVLNode *node) {
    // 空节点平衡因子为 0
    if (node == NULL) {
        return 0;
    }
    // 节点平衡因子 = 左子树高度 - 右子树高度
    return height(node->left) - height(node->right);
}

```

关于 AVL 树的构建和插入，见下一部分。

4.1.3 栈

栈的特点是“后进先出”，在树的遍历时，往往不能直接输出当前遍历到的节点，而是需要先存放，当遍历到的节点为叶子结点等时，输出该节点，再进行回溯，此时，栈的特点满足了这一特性。本实验中，采用栈这一数据结构，用于非递归实现树的遍历等操作时存放树节点。

对栈节点的定义如下，采用链栈来实现，方便自由压栈、弹栈，注意链栈的 next 指针由栈顶指向栈底。

```
// 定义栈节点
typedef struct StackNode {
    BstNode* treeNode;      // 栈中存储的二叉树节点
    struct StackNode* next; // 指向下一个栈节点的指针
} StackNode;
```

本实验中，栈的基本操作主要创建、压栈、弹栈和判断栈是否为空，实现代码分别如下：

```
// 创建一个栈节点
StackNode* createStackNode(BstNode* treeNode) {
    StackNode* newStackNode = (StackNode*)malloc(sizeof(StackNode));
    newStackNode->treeNode = treeNode;
    newStackNode->next = NULL;
    return newStackNode;
}

// 压栈
void push(StackNode** top, BstNode* treeNode) {
```

```
    StackNode* newStackNode = createStackNode(treeNode);
    newStackNode->next = *top;
    *top = newStackNode;
}

// 出栈
BstNode* pop(StackNode** top) {
    if (*top == NULL) {
        return NULL;
    }
    StackNode* temp = *top;
    BstNode* treeNode = temp->treeNode;
    *top = (*top)->next;
    free(temp);
    return treeNode;
}

// 判断栈是否为空
int isEmpty(StackNode* top) {
    return top == NULL;
}
```

4.2 函数/高层算法设计

4.2.1 文件 I/O 操作

实验所需文件 marketing_sample_10k_data.csv，drop sample_1k_data.csv 均存放在源代码的工作目录下。

对于 csv 文件，每一行的不同数据项用“,”分隔，在进行读取时，可以按行读取，按“,”分隔，注意跳过表头。

```
// 从文件中读取数据并构建二叉查找树
BstNode* buildTreeFromFile(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        perror("文件打开失败");
        exit(EXIT_FAILURE);
    }

    BstNode* root = NULL;
    char line[300];

    // 跳过表头
    fgets(line, sizeof(line), file);

```

```
    // 读取每一行数据
    while (fgets(line, sizeof(line), file)) {
        char ID[33];
        char Name[256];
        int inventory;

        sscanf(line, "%32[^,],%255[^,],%d", ID, Name, &inventory);
        root = insertBST(root, ID, Name, inventory);
    }

    fclose(file);
    return root;
}
```

在进行文件读取时，要检查文件指针，即文件是否打开成功。

4.2.2 二叉查找树的构造/插入

单独定义创建二叉树节点的函数如下，此函数实现了创建二叉查找书节点-分配内存-根据输入参数更新节点内容-返回指向该节点的指针。

```
// 创建新节点
BstNode* createNode(const char* ID, const char* Name, int inventory) {
    BstNode* newNode = (BstNode*)malloc(sizeof(BstNode));
    strcpy(newNode->ID, ID);
    strcpy(newNode->Name, Name);
    newNode->inventory = inventory;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

此函数得到一个孤立的节点，接下来需要将其插入到二叉查找树中。

对于插入，为保证二叉查找树的结构和性质，一种插入思路如下：

1. 根据节点值与目标节点值的关系，搜索左子树或右子树；
2. 重复步骤 1 直到到达叶子节点；
3. 根据节点的值与目标节点的值的关系，将新节点添加为其左侧或右侧的子节点。

代码实现上，递归比较简洁：

```
// 插入节点到 BST
BstNode* insertBST(BstNode* root, const char* ID, const char* Name, int inventory) {
    if (root == NULL) {
        return createNode(ID, Name, inventory);
    }
    if (strcmp(ID, root->ID) < 0) {
        root->left = insertBST(root->left, ID, Name, inventory);
    } else if (strcmp(ID, root->ID) > 0) {
```

```
        root->right = insertBST(root->right, ID, Name, inventory);
    }
    return root;
}
```

这种递归插入的实现思路是，如果待插入节点的 ID 小于当前节点的 ID，则继续在该节点的左子树中进行比较，反之，则在该节点的右子树中进行比较。由于商品的 ID 无重复，所以这种方法可行。

4.2.3 计算二叉查找树的高度

树的高度定义为从根节点到树中最深叶节点的路径上节点数（包括根节点）减一（因为根节点的高度为 0）。一种计算树的高度的方法是递归，思路如下：

1. 对于一个节点，如果为空，则返回深度 -1 ，表示为空树；
2. 如果不为空，该节点的高度应该是它的左子树和右子树中高度的较大者（该节点的高度 = 较大者 + 1 – 1 = 较大者）；
3. 如此递归计算出每个节点的高度，得到根节点的高度。

一种实现代码如下：

```
// 计算树的高度
int calculateHeight(BstNode* root) {
    if (root == NULL) {
        return -1; // 空树高度为 -1
    }
    int leftHeight = calculateHeight(root->left);
    int rightHeight = calculateHeight(root->right);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}
```

4.2.4 计算二叉查找树的平均查找长度

平均查找长度（Average Search Length, ASL）是衡量查找树效率的重要指标，定义为所有节点的查找路径长度的加权平均值。

公式如下：

$$\text{ASL} = \frac{\sum_{i=1}^{n}(D_i \times F_i)}{\sum_{i=1}^{n} F_i}$$

其中：

- D_i：节点 i 的深度（从根节点到该节点的路径长度）。
- F_i：节点 i 的频率。
- n：节点总数。

若假设每个节点的查找频率相等，公式简化为：

$$\mathrm{ASL} = \frac{\sum_{i=1}^{n} D_i}{n}$$

对于实验中的场景，由于提供的数据未给出商品的销量等信息，且商品库存相同，我们认为商品的查找频率均相等。

设计代码如下，仍采用递归的思想，同时，参数以指针传递，这样递归的公式更加清晰：

```
// 计算所有节点深度的总和和节点总数
void calculateTotalDepth(BstNode* root, int depth, int* totalDepth, int* totalNodes) {
    if (root == NULL) {
        return;
    }

    // 从根节点开始遍历，根节点的深度为 0

    *totalDepth += depth; // 总查找长度 += 该节点的深度
    (*totalNodes)++; // 节点数量++

    calculateTotalDepth(root->left, depth + 1, totalDepth, totalNodes); // 递归左子树，其中左孩子的深度 = 该节点的深度 + 1
    calculateTotalDepth(root->right, depth + 1, totalDepth, totalNodes); // 递归左子树，其中右孩子的深度 = 该节点的深度 + 1
}
```

当树非空时，ASL 按照如下形式进行计算：

```
double averageSearchLength = (double)totalDepth / totalNodes;
```

4.2.5 中序遍历（非递归形式）

中序遍历的顺序为：左-中-右，非递归形式实现中序遍历的大致思路如下：

1. 向左走到底，同时将路径上的节点压栈；
2. 若走到左下角，打印当前节点，即将当前节点出栈并访问；
3. 转向该节点的右子树（其位于左下角，一定无左子树）；

按照上述思路，编写代码如下：

```
// 非递归中序遍历
void inOrderTraversal(BstNode* root) {
    StackNode* stack = NULL; // 初始化空栈
    BstNode* cPtr = root;    // 初始指向根节点

    while (cPtr != NULL || !isEmpty(stack)) { // 当前节点不为空或栈非空
        while (cPtr != NULL) { // 当前节点不为空，则将当前节点入栈，然后继续往左直至走到底
            push(&stack, cPtr);
            cPtr = cPtr->left;
        }

```

```
        cPtr = pop(&stack); // 已经走到左下角，出栈，将栈顶节点赋给 p
        printf("%s\n", cPtr->ID); // 输出该节点

        cPtr = cPtr->right; // 该节点作为根节点，位于整棵树的左下角，无左子树，检查右子树
    }
}
```

4.2.6 查找节点

该操作虽然实验没有要求，但我们将其独立出来，为删除节点作准备。对于一般的二叉树，查找节点需要按照一定顺序遍历完所有节点（最差情况下）。

但对于二叉查找树，由于其性质（左子树节点值小于根节点值，右子树节点值大于根节点值），可以通过比较当前节点值与目标值的大小来决定查找方向，对于实验中商品数量较大的情况，能极大提升查找效率。

根据以上思路，采用递归思路设计查找代码如下：

```
// 搜索节点
BstNode* searchNode(BstNode* root, const char* ID) {
    // 如果节点为空或找到目标节点
    if (root == NULL || strcmp(root->ID, ID) == 0) {
        return root;
    }
    // 根据目标值选择查找方向
    if (strcmp(ID, root->ID) < 0) {
        return searchNode(root->left, ID);
    } else {
        return searchNode(root->right, ID);
    }
}
```

4.2.6 删除节点

从 BST 中删除节点，如果待删除的节点没有子节点，那么非常简单，但如果删除包含左子节点或者右子节点，特别是删除包含两个节点的节点，因为删除节点后需要仍保持二叉查找树的结构和性质，其思路大致如下：

1. 如果要删除的节点没有子节点，可以直接移除该节点；
2. 如果要删除的节点有一个子节点，我们可以用其子节点作为替换；
3. 如果要删除的节点有两个子节点，此时，对于该节点，左子树上所有节点的值均小于该节点的值，右子树上所有节点的值均大于该节点的值。
   我们希望删除该节点后，该树仍保持上述性质，那么需要用其中一个节点来替换，分析可知，该节点需要是左子树中的最大值或右子树中的最小值，结合中序遍历可知，该节点为**中序后继节点或者前驱节点**。

综合以上思路，设计代码如下（对于第三种情况，采取用其左子树的最大值替换）：

```c
// 删除节点
BstNode* deleteBST(BstNode* root, const char* ID, int* isFound) {
    if (root == NULL) {
        *isFound = 0;  // 节点未找到
        return NULL; // 空树，无法进行删除操作
    }
    if (strcmp(ID, root->ID) < 0) { // 往左子树查找
        root->left = deleteBST(root->left, ID, isFound);
    } else if (strcmp(ID, root->ID) > 0) { // 往右子树查找
        root->right = deleteBST(root->right, ID, isFound);
    } else { // 找到该节点
        *isFound = 1;
        // 情况 1：叶子节点
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }

        // 情况 2：只有一个子节点
        if (root->left == NULL) {
            BstNode* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            BstNode* temp = root->left;
            free(root);
            return temp;
        }

        // 情况 3：有两个子节点
        // 找到右子树中的最小节点
        BstNode* successor = root->right;
        while (successor->left != NULL) {
            successor = successor->left;
        }
        // 用后继节点的数据替换当前节点
        strcpy(root->ID, successor->ID);
        strcpy(root->Name, successor->Name);
        root->inventory = successor->inventory;

        // 删除后继节点
        root->right = deleteBST(root->right, successor->ID, isFound);
    }
    return root;
```

```
45. }
```

4.2.7 AVL 树的构建和插入

AVL 树插入的方法与二叉查找树类似，唯一的区别在于，在 AVL 树中插入节点后，从该节点到根节点的路径上可能会出现一系列失衡节点（平衡因子绝对值大于 1 的节点称为“失衡节点”。）。因此，我们需要从这个节点开始，自底向上执行旋转操作，使所有失衡节点恢复平衡。

根据节点失衡情况的不同，旋转操作分为四种：右旋、左旋、先右旋后左旋、先左旋后右旋。

1. 右旋：从底至顶看，找到二叉树中首个失衡节点，将该节点记为 node ，其左子节点记为 child ，执行“右旋”操作，以 child 为原点，将 node 向右旋转，再以 child 替代 node 原来的位置，注意如果 child 有右孩子，需要将右孩子作为 node 的左孩子，代码实现如下：

```
1. // 右旋操作
2. AVLNode *rightRotate(AVLNode *node) {
3.     AVLNode *child, *grandChild;
4.     child = node->left;
5.     grandChild = child->right;
6.     // 以 child 为原点，将 node 向右旋转
7.     child->right = node;
8.     node->left = grandChild;
9.     // 更新节点高度
10.    updateHeight(node);
11.    updateHeight(child);
12.    // 返回旋转后子树的根节点
13.    return child;
14. }
```

2. 左旋：就是“右旋”的镜像，同理可得：

```
1. // 左旋操作
2. AVLNode* leftRotate(AVLNode *node) {
3.     AVLNode *child, *grandChild;
4.     child = node->right;
5.     grandChild = child->left;
6.     // 以 child 为原点，将 node 向左旋转
7.     child->left = node;
8.     node->right = grandChild;
9.     // 更新节点高度
10.    updateHeight(node);
11.    updateHeight(child);
12.    // 返回旋转后子树的根节点
13.    return child;
14. }
```

3. 先左旋后右旋：先对 child 执行左旋，再对 node 执行右旋；
4. 先右旋后左旋：先对 child 执行右旋，再对 node 执行左旋；

针对不同情况，采用相应的操

失衡节点的平衡因子	子节点的平衡因子	旋转方法
> 1	>= 0	右旋
> 1	< 0	先左旋再右旋
< -1	<= 0	左旋
< -1	> 0	先右旋再左旋

其中失衡节点的平衡因子 > 1 即为左偏树，< 1 即为右偏树。
将上述情况的判断分装为函数如下：

```
// 旋转操作，使该子树重新恢复平衡
AVLNode* Rotate(AVLNode *node) {
    // 获取节点 node 的平衡因子
    int bf = balanceFactor(node);
    // 左偏树
    if (bf > 1) {
        if (balanceFactor(node->left) >= 0) {
            // 右旋
            return rightRotate(node);
        } else {
            // 先左旋后右旋
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
    }
    // 右偏树
    if (bf < -1) {
        if (balanceFactor(node->right) <= 0) {
            // 左旋
            return leftRotate(node);
        } else {
            // 先右旋后左旋
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
    }
    // 平衡树，无须旋转，直接返回
    return node;
}
```

插入节点的过程与二叉查找树类似，但由于再插入后需要进行旋转操作保证

平衡，采用递归思路进行需要额外变量或传入更多参数以处理旋转和返回值，所以编写代码时，采用辅助函数简化了递归逻辑：主函数 insertAVL 提供接口，传入根节点指针（root），而辅助函数 insertHelper 负责递归逻辑和树平衡操作。这样递归逻辑较为简洁。编写代码如下：

```
// 插入节点
void insertAVL(AVLNode** root, const char* ID, const char* Name, int inventory) {
    *root = insertHelper(*root, ID, Name, inventory);
}

// 递归插入节点（辅助函数）
AVLNode* insertHelper(AVLNode* node, const char* ID, const char* Name, int inventory)
{
    if (node == NULL) {
        return createNode(ID, Name, inventory);
    }
    /* 1. 查找插入位置并插入节点 */
    if (strcmp(ID, node->ID) < 0) {
    node->left = insertHelper(node->left, ID, Name, inventory);
    } else if (strcmp(ID, node->ID) > 0) {
        node->right = insertHelper(node->right, ID, Name, inventory);
    } else {
        // 重复节点不插入，直接返回
        return node;
    }
    // 更新节点高度
    updateHeight(node);
    /* 2. 执行旋转操作，使该子树重新恢复平衡 */
    node = Rotate(node);
    // 返回子树的根节点
    return node;
}
```

4.3 具体实验内容实现及运行结果

4.3.1 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。

将树的信息导出为 dot 文件，在用 dot 工具生成可视化的树如下（由于节点太多，结果保存在 bst_graph.pdf 中，以下仅为部分）：

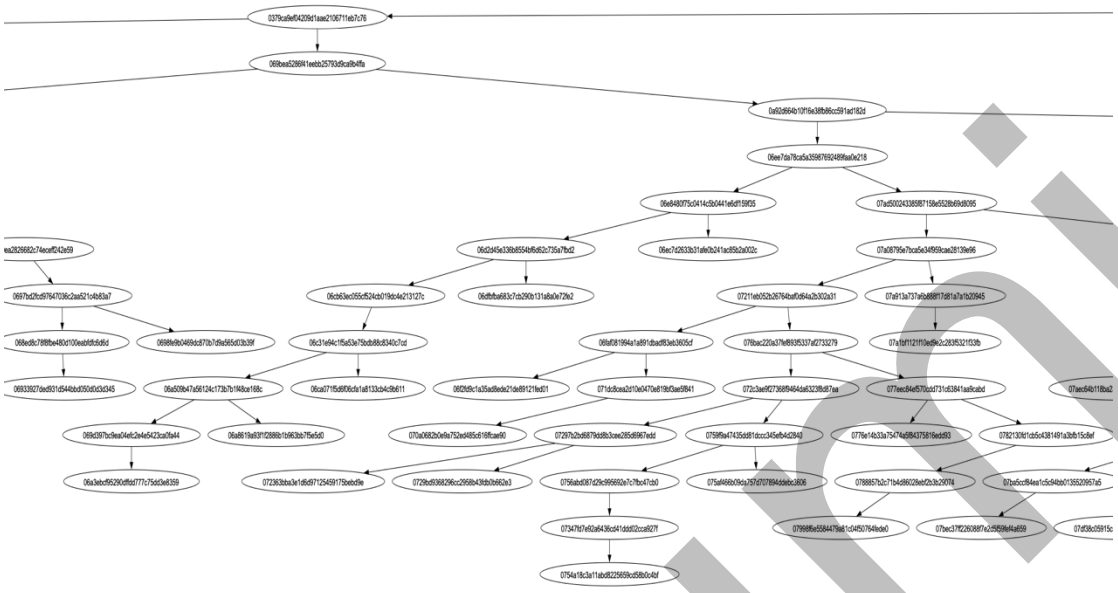

图 1 BST 树可视化结果（局部）

4.3.2 求出该二叉查找树的高度

调用函数，得到结果如下：

```
(base) yokumi@YokumideMacBook-Air ~/Documents/Code-Repo/DS/Lab2 main ±
 lab2 && "/Users/yokumi/Documents/Code-Repo/DS/Lab2/"lab2
二叉查找树的高度：31
```

图 2 BST 树高度

4.3.3 求针对这 10k 项商品的平均查找长度

根据以上定义，设计函数如上，运行，得到结果如下：

```
(base) yokumi@YokumideMacBook-Air ~/Documents/Code-Repo/DS/Lab2 main ±
 lab2 && "/Users/yokumi/Documents/Code-Repo/DS/Lab2/"lab2
二叉查找树的高度：31
二叉查找树的平均查找长度：15.038
```

图 3 平均查找长度（保留三位小数）

由结果可见，二叉查找树极大程度的提高了查找效率，对于 10k 个商品，平均查找长度仅需约 15，极大地提升了查找效率。

4.3.4 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

由于输出含 10k 个商品，不通过命令行输出，而是将结果保存至“中序遍历结果.txt”文件中，文件部分内容如下：

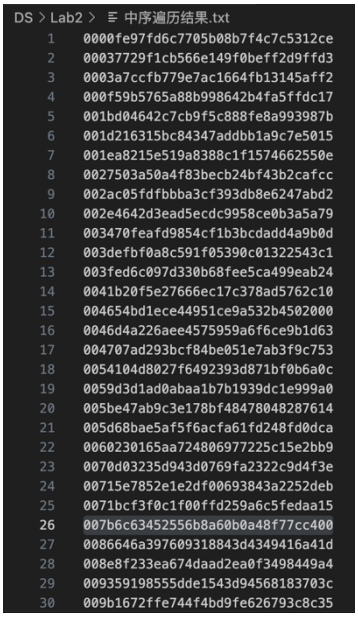

图 4 BST 树中序遍历结果

可以发现，BST 树的中序遍历结果严格按照字符串从小到大顺序排列，这是因为中序遍历的顺序为 左 – 中 – 右，正符合 BST 树小的插入左子树，大的插入右子树的插入顺序。

4.3.5 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

运行代码，将待删除商品删除后，再输出中序遍历结果如下：

图 5 删除后的中序遍历结果

可以验证，删除后，树仍保持 BST 的良好性质，并且所有待删除节点均已找到并成功删除。

4.3.6 求删除后的二叉查找树的高度

同样运行代码，计算删除后的二叉树高度，得到结果如下：

```
(base) ✘ yokumi@YokumideMacBook-Air ~/Documents/Code-Repo/DS/Lab2 ᛘ main ±
-o lab2 && "/Users/yokumi/Documents/Code-Repo/DS/Lab2/"lab2
二叉查找树的高度：31
二叉查找树的平均查找长度：15.038
中序遍历结果已保存到文件：中序遍历结果.txt
中序遍历结果已保存到文件：中序遍历结果_删除后.txt
删除商品后，二叉平衡树的高度：31
删除后的二叉查找树的平均查找长度：14.812
```

图 6 删除后的的二叉树高度

由结果可以发现，删除 1k 个节点对于二叉树平衡树的高度和平均查找次数均无显著影响。

4.3.7 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

同第一步，将构造的二叉平衡树的信息导出为 dot 文件，在用 dot 工具生成可视化的树如下（由于节点太多，结果保存在 avl_graph.pdf 中，以下仅为部分）：

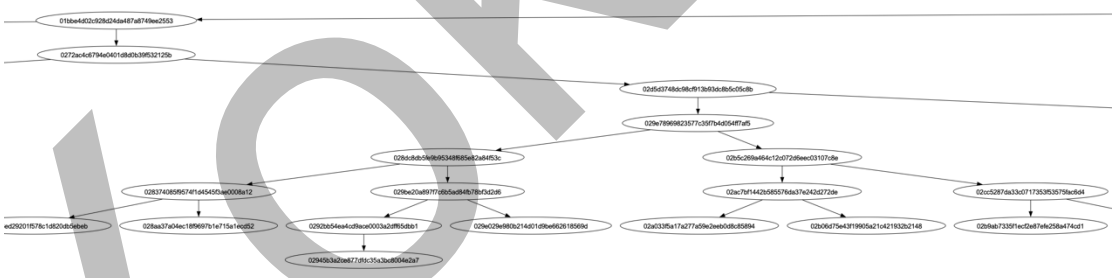

图 7 AVL 树可视化结果（局部）

从图中明显可以看出，AVL 树的高度比普通的 BST 树要低很多，我们输出其高度和平均查找次数如下：

```
● (base) yokumi@YokumideMacBook-Air ~/Documents/Code-Repo/DS/Lab2 ᛘ main ±
  a.c -o lab2_extra && "/Users/yokumi/Documents/Code-Repo/DS/Lab2/"lab2_extra
  AVL树的高度：15
  AVL树的高度 = 1.13logN
  中序遍历结果已保存到文件：中序遍历结果_AVL.txt
  二叉查找树的平均查找长度：11.530
```

图 8 AVL 树相关结果

由图可见，AVL 树的高度仅为 BST 树的一半，查找效率为 logN 级别，对于含有大量商品的实际情况，极大程度地提升了查找效率。不过缺点是，如果需要

频繁删除商品，其删除操作较为繁琐，不过效率比 BST 树有显著提升。

五、实验分析和总结（10 分）

本实验总体来说完成较为顺利，难点主要在于涉及树的操作的递归实现，如计算节点高度、递归插入节点等操作，较难直接想明白递归过程，所以采取了辅助函数或传递指针等方法，简化了递归函数的设计，使得过程更加清晰直观。

对于附加要求，课堂上只讲述了其原理和一些基本操作，像旋转等操作的实现等在本实验之前均未实际上手练习过，在其插入函数的设计上遇到了一些困难，比如在旋转时，并未考虑该节点有 grandchild 的情况，导致在代码调试过程中遇到段错误，通过设置断点定位问题位置，发现是未考虑上述情况，成功解决，并熟悉了 AVL 树维持平衡的操作。

本实验通过一个商品管理系统的实例，直观地展现了二叉查找树以及二叉平衡树对于查找效率地提升，并熟练了树的基本操作设计，尤其是递归函数的思想。

六、程序源代码（10 分）

由于实验中，基本要求和附加要求的实现放在两个代码文件中分别实现，篇幅较长，本实验的完整源代码和运行结果均存放在压缩包中上传至教学云平台。