**第二次研讨课报告**

**一、树形结构在文件管理中的应用**

**1. 问题**

1. 如何利用树形结构管理文件目录，并区分文件和文件夹。
2. 如何统计某节点下文件夹和文件的数量。
3. 如何实现文件或文件夹的删除、复制、移动操作。
4. 如何在目录树和路径之间进行映射，实现地址路径与树节点的互相定位。

**2. 对问题的理解与分析**

* **树形结构表示文件系统**：
  + 文件系统是一个典型的树形结构。
  + 每个文件夹是树的节点，每个文件夹或文件是节点的子节点。
  + 根节点表示顶层目录（如 C:\ 或 /）。
* **节点的组成**：
  + 每个节点包含：
    - 名称：文件夹或文件的名称。
    - 类型：区分文件夹（目录）和文件。
    - 子节点列表：存储子目录和文件的指针列表。
* **目录树的操作**：
  + **统计数量**：递归遍历节点的子节点，分别统计文件和文件夹的数量。
  + **删除**：删除目标节点及其子节点。
  + **复制**：递归复制节点及其所有子节点。
  + **移动**：将目标节点从当前父节点移除，并插入到新父节点中。
  + **路径映射**：利用递归或迭代实现路径到节点、节点到路径的双向映射。

#### 3. 所采用的逻辑结构、存储结构及算法思想描述

**3.1 数据结构**

struct TreeNode{

    string name         // 节点名称

    bool isDirectory    // true: 文件夹; false: 文件

    List<TreeNode\*> children // 子节点列表

};

##### ****3.2 树形结构管理文件目录****

1. **根节点表示顶层目录**：
   * 根节点为树的入口，表示文件系统的顶层目录。
2. **区分文件和文件夹**：
   * 通过 isDirectory 字段区分文件夹（true）和文件（false）。

##### ****3.3 算法****

###### (1) ****统计文件和文件夹数量****

FUNCTION CountFilesAndFolders(TreeNode\* node):

    IF node IS NULL:

        RETURN (0, 0)

    fileCount ← 0

    folderCount ← 0

    IF node.isDirectory:

        folderCount += 1

    ELSE:

        fileCount += 1

    FOR child IN node.children:

        (childFiles, childFolders) ← CountFilesAndFolders(child)

        fileCount += childFiles

        folderCount += childFolders

    RETURN (fileCount, folderCount)

###### (2) ****删除节点****

FUNCTION DeleteNode(TreeNode\* parent, TreeNode\* target):

    IF parent IS NULL OR target IS NULL:

        RETURN false

    FOR child IN parent.children:

        IF child == target:

            REMOVE child FROM parent.children

            DELETE child AND ITS CHILDREN

            RETURN true

    RETURN false

###### (3) ****复制节点****

FUNCTION CopyNode(TreeNode\* source):

    IF source IS NULL:

        RETURN NULL

    newNode ← CREATE NEW NODE

    newNode.name ← source.name

    newNode.isDirectory ← source.isDirectory

    FOR child IN source.children:

        newNode.children.ADD(CopyNode(child))

    RETURN newNode

###### (4) ****移动节点****

FUNCTION MoveNode(TreeNode\* sourceParent, TreeNode\* targetParent, TreeNode\* target):

    IF sourceParent IS NULL OR targetParent IS NULL OR target IS NULL:

        RETURN false

    DeleteNode(sourceParent, target) // 从原父节点移除

    targetParent.children.ADD(target) // 插入到新父节点

    RETURN true

###### (5) ****地址路径与目录树的映射****

**(a) 地址路径 → 树节点：**

FUNCTION LocateNodeByPath(TreeNode\* root, string path):

    parts ← SPLIT path BY "/"

    current ← root

    FOR part IN parts:

        FOUND ← false

        FOR child IN current.children:

            IF child.name == part:

                current ← child

                FOUND ← true

                BREAK

        IF NOT FOUND:

            RETURN NULL // 路径不存在

    RETURN current

**(b) 树节点 → 地址路径：**

FUNCTION GetPathFromNode(TreeNode\* node):

    IF node IS NULL:

        RETURN ""

    path ← node.name

    WHILE node.parent IS NOT NULL:

        node ← node.parent

        path ← node.name + "/" + path

    RETURN path

**4. 时间复杂度**

统计文件和文件夹：O(n)，n为节点总数

删除节点：O(k)，kkk为目标节点子节点数

复制节点：O(k)，kkk为复制节点子节点数

移动节点：O(1)+O(k)

路径映射：O(d)，d为路径深度

#### 实验总结

1. 树形结构是文件系统管理的核心，可以高效地组织和操作文件及文件夹。
2. 统计文件和文件夹数量、删除、复制和移动操作可通过递归遍历实现，算法简单高效。
3. 地址路径和目录树之间的映射需要结合字符串操作，适合大多数文件管理应用。

**二、从海量短信中找出重复出现最多的前10条**

#### 1. 问题

给定一个包含一千万条短信的文本文件（ASCII 格式，每行一条短信），文件中可能有重复短信。任务是找出重复出现次数最多的前 10 条短信，并输出结果。

#### 2. 对问题的理解与分析

**输入与输出：**

* **输入：**一个包含一千万条短信的文本文件，文件中的每行表示一条短信。
* **输出：**出现频率最高的前 10 条短信及其出现次数。

**3. 所采用的逻辑结构、存储结构及算法思想描述**

### ****方法一：基于字符串比较的二叉树****

##### ****逻辑结构****

1. **树节点定义**：
   * 每个节点包含：
     + key：存储字符串内容。
     + count：该字符串的出现次数。
     + left 和 right：左右子树指针。
2. **操作**：
   * **插入**：
     + 使用 strcmp 比较新字符串与当前节点的字符串。
     + 如果返回 1，插入左子树；如果返回 -1，插入右子树；如果返回 0，计数加一。
   * **中序遍历**：
     + 按照节点的频率排序，并动态维护出现次数最多的前 10 条。

结构体：

STRUCTURE TreeNode:

    string key          // 存储短信内容

    int count           // 记录该短信的出现次数

    TreeNode\* left      // 指向左子树

    TreeNode\* right     // 指向右子树

1. 插入节点

FUNCTION Insert(TreeNode\* root, string key):

    // 如果当前节点为空，创建一个新节点并初始化

    IF root IS NULL:

        root ← CREATE NEW NODE

        root.key ← key

        root.count ← 1

        root.left ← NULL

        root.right ← NULL

        RETURN root

    // 比较新字符串与当前节点的字符串

    COMPARE ← strcmp(key, root.key)

    IF COMPARE == 0:

        // 如果字符串相同，增加计数

        root.count += 1

    ELSE IF COMPARE < 0:

        // 如果字符串小于当前节点，插入左子树

        root.left ← Insert(root.left, key)

    ELSE:

        // 如果字符串大于当前节点，插入右子树

        root.right ← Insert(root.right, key)

    RETURN root

1. 中序遍历

FUNCTION Inorder(TreeNode\* root, PriorityQueue top10Heap):

    IF root IS NOT NULL:

        // 遍历左子树

        Inorder(root.left, top10Heap)

        // 将当前节点的频率和字符串加入最小堆

        ADD (root.count, root.key) TO top10Heap

        // 如果堆中元素超过 10 个，移除频率最小的

        IF top10Heap.size > 10:

            REMOVE smallest element FROM top10Heap

        // 遍历右子树

        Inorder(root.right, top10Heap)

##### 时间复杂度

* 插入：平均 O(log⁡N)O(\log N)O(logN)，最坏 O(N)O(N)O(N)。
* 遍历：O(N)O(N)O(N)。
* 总复杂度：O(Nlog⁡N)O(N \log N)O(NlogN)。

**方法二：基于字典树（Trie）的统计方法**

**Trie 树节点定义**：

* 每个节点包含：
  + children：指向子节点的指针数组（大小为字符集大小，假设为 ASCII 256）。
  + count：记录字符串出现次数（仅在叶节点使用）。
* 每个字符串按字符逐级插入，路径唯一。

**字典树思想**：

1. 每个节点表示短信的一个字符。
2. 判断当前字符是否存在对应的子节点：
   * 若存在，进入对应节点；
   * 若不存在，创建新节点。
3. 重复上述步骤，直到处理到字符串的最后一个字符。
4. 最后一个字符的节点存储该字符串的出现次数。

**解决思路**：

1. 使用逐字节比较构建字典树。
2. 每条短信按字符插入到字典树，最后一个字符对应的节点计数加一。
3. 深度优先搜索（DFS）遍历字典树，找到出现次数最多的前 10 条短信。

**结构体**

STRUCTURE TrieNode:

    TrieNode\* children[256]  // 每个节点最多有 256 个子节点（ASCII 字符集）

    int count                // 记录字符串出现次数

**插入字符串到字典树**

FUNCTION Insert(TrieNode\* root, string key):

    TrieNode\* current ← root

    FOR char IN key:

        // 如果当前字符的子节点不存在，则创建新节点

        IF current.children[char] IS NULL:

            current.children[char] ← CREATE NEW TrieNode

        // 移动到对应子节点

        current ← current.children[char]

    // 字符串结束后，更新计数

    current.count += 1

深度优先搜索（DFS）遍历字典树

FUNCTION DFS(TrieNode\* node, string currentPath, PriorityQueue top10Heap):

    // 如果节点为空，直接返回

    IF node IS NULL:

        RETURN

    // 如果节点的计数大于 0，说明该路径对应一个完整字符串

    IF node.count > 0:

        ADD (node.count, currentPath) TO top10Heap

        // 如果堆中元素超过 10 个，移除频率最小的

        IF top10Heap.size > 10:

            REMOVE smallest element FROM top10Heap

    // 遍历所有可能的子节点

    FOR i IN RANGE(256):

        IF node.children[i] IS NOT NULL:

            // 递归访问子节点，并在路径中追加当前字符

            DFS(node.children[i], currentPath + CHAR(i), top10Heap)

**主函数**

FUNCTION Main():

    INITIALIZE TrieNode\* root ← CREATE NEW TrieNode  // 初始化字典树根节点

    INITIALIZE PriorityQueue top10Heap              // 用于存储前 10 条高频短信（最小堆）

    // Step 1: 打开文件并逐行读取短信内容

    OPEN file FOR READING

    WHILE NOT END OF file:

        line ← READ NEXT LINE

        Insert(root, line)  // 插入短信到 Trie

    CLOSE file

    // Step 2: 深度优先搜索 Trie 找到前 10 条高频短信

    DFS(root, "", top10Heap)

    // Step 3: 输出结果

    CONVERT top10Heap TO sorted list BY frequency DESC

    FOR each (frequency, message) IN sorted list:

        PRINT frequency, "-", message

#### 4. 时间复杂度

1. **插入字符串到字典树**：
   * 每条短信按字符插入，时间复杂度为 O(L)O(L)O(L)，其中 LLL 是字符串的平均长度。
   * 总插入复杂度为 O(L⋅N)O(L \cdot N)O(L⋅N)，NNN 是字符串总数（1,000 万条）。
2. **DFS 遍历字典树**：
   * 遍历每个节点的复杂度为 O(L⋅N)O(L \cdot N)O(L⋅N)，因为树的总节点数与字符串的字符总数成正比。
3. **总体复杂度**：
   * 插入和遍历的总复杂度为 O(L⋅N)O(L \cdot N)O(L⋅N)。