山东大学 计算机科学与技术 学院

数据结构与算法课程设计 课程实验报告

学号：202200130039	姓名：张向荣	班级：22.3
上机学时：4 小时	实验日期：2024.5.5	

课程设计题目：

N-ary 树的实现与分析

软件开发环境：

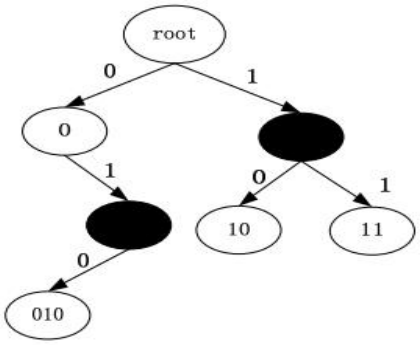
Dev-c++

报告内容：

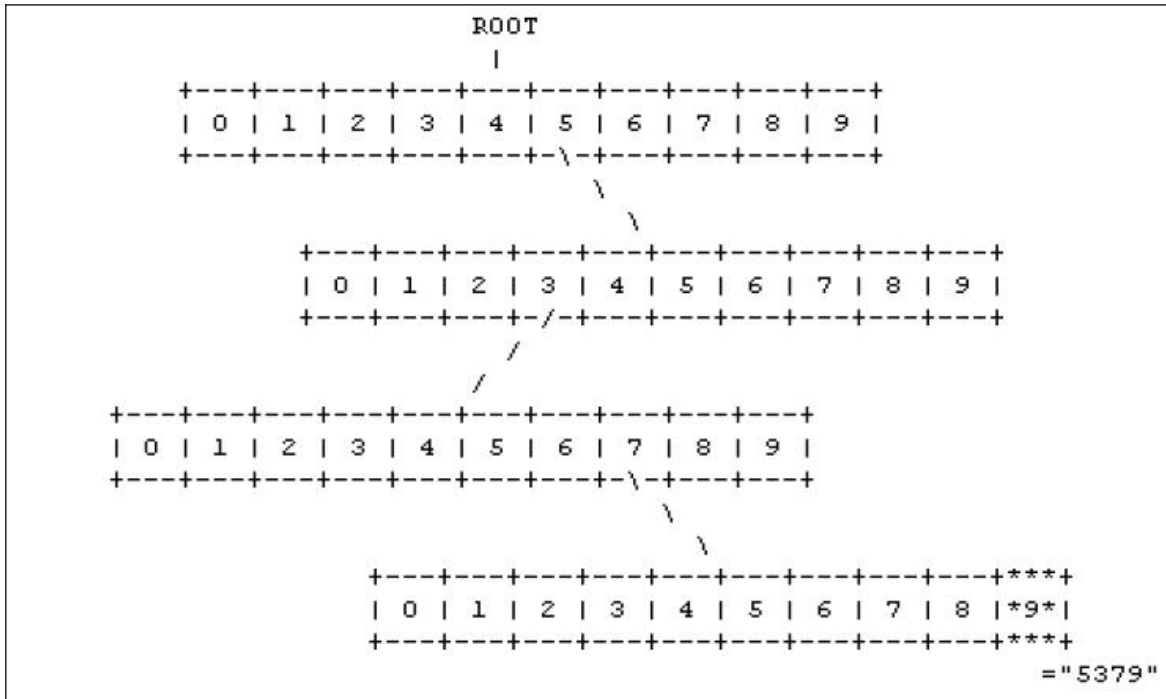
1.需求描述

1.1 问题描述

哈夫曼算法输出的结果就是一个二元 Trie，在二元 Trie 中，每个左子树分支用 0 表示，右子树分支用 1 表示。如下图就是就是一个二元 Trie 的示例图。



可以将二元 Trie 扩展到 N-ary Trie。在 N-ary Trie 中，每个结点都有 0~N 之间任何个儿子结点,其中每个分支都用一个相应的符号表示(在 N-ary Trie 中有 N 个不同的符号)。例如一个电话本可用一个 N-ary Trie 表示，其中 N=10，分别表示 0~9 的十个数字，具体情况如下图所示：



1.2 基本要求

① 设计并实现 N-ary Trie 的 ADT（N=26，建立在英语上的 Trie），该 ADT 包括 Trie 的组织存储以及其上的基本操作：包括初始化，查找，插入，删除等。

② 应用 Trie 结构实现文本文档的索引化。首先扫描文本文档（存放在 txt 文件中），然后在此基础上用 Trie 登记单词出现的位置（行号），最后在 Trie 上实现查询。

③ 用户输入的查询可以是针对一个单词的，也可是针对一个字符序列的（以某几个字母开头的单词）。

1.3 输入说明

输入界面设计

首先是一个菜单，选择要实现的功能。

1. 从文件初始化树
2. 查找
3. 插入
4. 删除
5. 退出

输入 1 实现树的初始化，也就是构建树的过程。2、3、4 即在树上进行相应的操作功能。5.退出循环。

输入样例

1.4 输出说明

输出界面设计

当输入 1，如果没有找到相应的文件，则重新输入。如果找到，提示“树已从文件初始化”。

当输入 2，继续输入要查找的单词或字符序列，然后将单词或者以其为前缀的单词或字符序列以及所在的行号全部输出。

当输入 3，继续输入要插入的单词，以及单词要插入的行号。插入成功则输出“单词已经插入”。

当输入 4，继续输入要删除的单词或字符序列，删除成功输出“单词已被删除”。

当输入 5，输出“感谢使用，再见！”。

当输出其他数字，输出“无效的选择，请重新输入。”

输出样例

2.分析与设计

2.1 问题分析

题目中有三个要求，要求 1 是实现树的功能，且 N=26，这样这棵树就变成了一棵单词查找树，用类实现并要实现相关功能。

要求 2 是需要我们实现文本文档的索引化，可以使用 getline()函数扫描文本文档，然后使用插入函数将文本中的内容插入到树中。

要求 3 需要实现针对单词与字符序列为前缀的查找。

首先要了解单词查找树的性质：

1. 单词查找树由字符串键中的所有字符构造而成，允许使用被查找键中的字符进行查找。

2. 每个节点都只可能有一个指向它的节点，称为它的父节点（只有一个节点除外，即根节点，没有任何节点指向根节点）

3. 每个节点都有 R 条链接，其中 R 为字母表大小，一般很多都是空链接，空链接一般省略。

4. 每个结点含有一个相应的值，可以是空也可以是符号表中的某个键所关联的值（值为空的节点在符号表中没有对应的键，它们的存在是为了简化单词查找树中的查找操作）

5. 将每个键所关联的值保存在该键的最后一个字母所对应的节点中。

2.2 主程序设计

在 main.cpp 中，定义一个用 trie 结构实现文本文档索引化功能的一个函数，使用 getline（）函数扫描文本的每一行，假如其中有空格，使用 string 中 erase()函数将空格删除。将该行的单词或者字符序列以及行号使用树中的 insert（）函数插入到树中。

然后再 main()函数中，通过输入相应的选择执行不同的函数功能。

2.3 设计思路

首先 N=26，正好对应 26 个字母，可以转化为单词查找树。每个节点的孩子节点都可以有 26 种情况的字母，则在结构体节点中定义一个指针结构体数组，长度为 26，表示 26 个孩子节点。通过类成员函数实现相关操作。初始化可以将文本中的内容用 insert（）函数插入到树中，在前缀查找中，可以根据查找步骤找到前缀，然后遍历尾字符的所有节点，并使用递归将所有可能的情况全部遍历。

2.4 数据及数据类型定义

定义树的节点结构体，定义一个结构体类型的指针，int 类型的变量表示行号，bool 类型标记是否为一个单词或序列的最后一个字母。在树的类中定义一个结构体类型的指针，表示树的根节点。

2.5.算法设计及分析

设计一个 trie 类，实现查询、插入、删除、初始化等操作。

首先定义一个结构体树节点，定义数组孩子节点，长度为 26。

在结构体中定义一个记录单词行号的变量。一个 bool 类型的变量，用来确定是否是一个单词的最后一个字母，便于后序操作。如果是最后一个字母则说明单词已经遍历结束，返回 true,否则需要继续遍历，返回 false。

在插入操作 insert()中，首先需要确定树是否存在该单词。从根节点开始遍历，将字母转换为数字索引查找对应的孩子节点。

插入操作中有两种情况：
1. 在到达单词的尾字符值之前就遇到了一个空结点，说明字符序列不存在。如果不存在对应节点，则创建孩子节点，直到遍历到最后一个节点，记录行号，并标记最后一个孩子节点为 true。
2. 在遇到空节点之前就到达了单词的尾字符，说明字符序列已经存在，此时直接设置尾字符对应标记为 true，并记录行号。

在删除操作 erase（）中，首先需要确定树是否存在该单词。先从根节点开始遍历，将字母转换为数字索引查找对应的孩子节点。
删除操作有两种情况：
1. 在到达单词的尾字符之前就遇到了一个空节点，说明字符序列不存在。无需删除，直接返回。
2. 在遇到空节点之前就到达了单词的尾字符，此时将单词尾字符标记为 false,并清除行号。

在初始化操作中，首先清空现有的 trie 的数据。用 trie 结构实现文本文档的索引化，使用 getline（）函数读取文本的每行内容，使用 string::erase()将单词中的空格删除，将每行的单词插入 trie 中。

应用 trie 结构实现文本文档的索引化，首先要扫描文本，将文本中的单词存到（插入操作）单词树中，并记录每个单词的行号，用 trie 查询功能实现对单词的查找。

在查询操作中，用户输入查询可以是针对一个单词的、也可以是针对一个字符序列。
查找操作分为两种：1、针对单词或字符序列的查找。2、以其为前缀和的查找。

在针对单词或字符序列的查找中，首先要确定输入的单词或字符序列是否存在。通过遍历输入的字符序列，查找的结果有 3 种：
1. 单词的尾字符所对应的节点中标记值为 true，查找成功。
2. 单词的尾字符所对应的节点中标记值为 false，查找失败。
3. 查找结束于一条空链接，查找失败。

在查找时如果在 trie 中存在该字符序列，（无论单词尾字符的节点中标记为 false 还是 true），然后开始遍历序列后面的孩子节点确定是否存在以此为前缀的序列。遍历尾字符节点的非空孩子节点，通过递归继续向下遍历，每遍历一次验证节点的标记，将标记为 true 的字符序列与其行号输出，直至全部为空节点。
如果不存在，无序继续遍历。

3. 测试

```
菜单：
1. 从文件初始化树
2. 查找
3. 插入
4. 删除
5. 退出
请选择操作： 1
请输入文件名： dc.txt
树已从文件初始化。

菜单：
1. 从文件初始化树
2. 查找
3. 插入
4. 删除
5. 退出
请选择操作： 2
请输入要查找的单词或字符序列： apple
apple 在第 1 行出现。
appleada 在第 4 行出现。
appleafdagsa 在第 5 行出现。
apples 在第 2 行出现。
applesda 在第 3 行出现。
```

```
菜单：
1. 从文件初始化树
2. 查找
3. 插入
4. 删除
5. 退出
请选择操作： 3
请输入要插入的单词： blue
请输入单词的行号： 3
单词 'blue' 已插入。

菜单：
1. 从文件初始化树
2. 查找
3. 插入
4. 删除
5. 退出
请选择操作： 4
请输入要删除的单词： blue
单词 'blue' 已删除。

菜单：
1. 从文件初始化树
2. 查找
3. 插入
4. 删除
5. 退出
请选择操作： 5
感谢使用，再见！
```

4. 分析与探讨

单词查找树，特点是以空间换时间，在查找字符串时有极大的时间优势，查找的时间复杂度与键的数量无关，在找到的情况下，最大的时间复杂度为键长度+1，找不到的情况下可以小于键的长度。优点是利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较。

5. 附录：实现源代码

```
#include<string>
#include"trie.h"

using namespace std;
```

```
Trie::Trie(){
    root=new TrieNode();//创建根接点
}


Trie::~Trie()
{
    deleteNode(root);
}

void Trie::deleteNode(TrieNode* node)
{
        if(node==NULL)
        {
            return;
        }

        for(int i=0;i<NODE_NUM;++i)
        {
            deleteNode(node->children[i]);//通过递归将 node 的孩子结点全部删除
        }

        delete node;//删除 node 结点
}

void Trie::searchNode(TrieNode* node,const string& currentWord,int currentLine)
{
        if(node==NULL) return;

        if(node->isEndOfWord)//存在单词
        {
            cout<<currentWord<<" 在第 "<<node->positions<<" 行出现。\n";
        }

        for(int i=0;i<NODE_NUM;++i)
        {
            if(node->children[i]!=NULL) //该字母存在
            {
                char ch='a'+i;//转换为字符
                searchNode(node->children[i],currentWord+ch,currentLine);// 寻找下一个
存在字母的位置
            }
        }
}

    // 查询以指定字符串或者指定字符串开头的所有单词
```

```
void Trie::searchWithPrefix(const string& prefix)
{
            TrieNode* current=root;
            for(char ch : prefix) //遍历前缀序列
           {
                  int index=ch-'a';
                  if(!current->children[index])
                {
                        return; //  前缀不存在，无需继续搜索
                  }
                  current=current->children[index];
            }
            //  开始搜索以指定字符串或者指定字符串开头的所有单词
            searchNode(current,prefix,current->positions);
}


      //  获取单词的位置信息
int Trie::getWordPositions(const string& word)
{
            TrieNode* current=root;
            for(char ch : word)
           {
                  int index=ch-'a';//转化成孩子节点的索引
                  if(!current->children[index])
                {
                        return -1; //  单词不存在
                  }
                  current=current->children[index];
            }
            if(current->isEndOfWord)
           {//最后一个字母为 true,则单词存在。
                  return current->positions;//返回单词位置
            }
            else
           {
                  return -1; //  单词不存在
            }
}

void Trie::insert(const string& word,int position)
{
            TrieNode* current=root;//从根节点开始遍历
            for(char ch : word)
           {
                  int index=ch-'a';//转换成每个字母的数字索引
```

```
                if(!current->children[index])
                {//如果孩子不存在该字母
                        current->children[index]=new TrieNode();//创建结点
                }
                current=current->children[index];
        }
        current->isEndOfWord=true;
        current->positions=position;//行号
}

void Trie::erase(const string& word)
{
        TrieNode* current=root;
        for(char ch : word)
        {
                int index=ch-'a';
                if(!current->children[index])
                {
                        return; //  单词不存在，无需删除
                }
                current=current->children[index];//查看单词的下一个字母是否存在
        }
        current->isEndOfWord=false;
        current->positions=0;
}

    //  查询单词或字符序列
bool Trie::search(const string& word)
{
        TrieNode* current=root;
        for(char ch : word)
        {
                int index=ch-'a';
                if(!current->children[index])
                {//未找到
                        return false;//直接结束
                }
                current=current->children[index];//指向下一个字母
        }
        return current->isEndOfWord;//最后一个单词是否 true or false
}

   //  清空  Trie
void Trie::clear()
{
    deleteNode(root);
```

```
        root = new TrieNode();
}
```