天津大学

数据结构实验报告

实验名称：树

学院名称　　　　智能与计算学部

专　　业　　　　软件工程

学生姓名　　　　陈昊昆

学　　号　　　　3021001196

年　　级　　　　2021 级

班　　级　　　　软工 3 班

时　　间　　2023 年 5 月 18 日

1. 实验内容

1 实现树的节点类 TreeNode（支持线索二叉树）和树类 MyTree（支持线索二叉树），采用二叉链表存储，完成以下功能：

- 树节点类的初始化和销毁
- 树节点类的打印，void printNode()
- 树初始化 MyTree()，初始化一棵空树
- 树初始化 MyTree(const char[])，根据二叉树的先序序列，生成二叉树的二叉链表存储,使用@表示 NULL
- 树的复制构造函数 MyTree(const MyTree&),复制参数中的树
- 树销毁~MyTree()支持普通二叉树与线索二叉树
- 先序遍历二叉树并打印，void preOrderTraverse()//仅支持普通二叉树，不考虑线索化，该函数不可直接递归调用，可添加必要的函数支持，对节点的访问操作为打印该节点
- 中序遍历二叉树并打印，void inOrderTraverse()//支持普通二叉树与线索二叉树，该函数不可直接递归调用，可添加必要的函数支持，对节点的访问操作为打印该节点
- 后序遍历二叉树并打印，void postOrderTraverse()//仅支持普通二叉树，不考虑线索化，该函数不可直接递归调用，可添加必要的函数支持，对节点的访问操作为打印该节点
- 定位二叉树中的节点，TreeNode& locateNode(const char& v); //在树中找到值为 v 的节点则返回该节点，否则返回 NULL，支持普通二叉树与线索二叉树
- 计算二叉树的叶子结点数，int countLeaf()//仅支持普通二叉树，不考虑线索化
- 计算二叉树的深度，int countHeight()//仅支持普通二叉树，不考虑线索化
- 当前树是否是线索二叉树，bool isThreadedTree()，是线索二叉树返回 true，否则 false
- 为二叉树生成中序线索二叉树 bool inOrderThreading()
- 寻找中序线索二叉树中某节点的前驱节点 TreeNode& preNode(const TreeNode&);//仅支持线索二叉树
- 寻找中序线索二叉树中某节点的后继节点，TreeNode& nextNode(const TreeNode&);//仅支持线索二叉树

2 实现霍夫曼树 HuffmanTree，输出对应的霍夫曼编码

- 树初始化 HuffmanTree(const int&, const int[])，根据输入创建一棵

霍夫曼树，第一个参数为节点个数，第二个参数为节点数组，节点值为节点重要度，越大代表越重要，要求树构建时偏小的值放入左子树，偏大的值放入右子树

- 树销毁~HuffmanTree()
- 输出霍夫曼编码 void printHuffmanCodes()//格式：节点值：编码，节点排序递减
- 其他必要的函数

2. 程序实现

```
#include "myTree.h"
#include <bits/stdc++.h>
using namespace std;


static TreeNode nullNode('@', nullptr,nullptr,Link,Link);
// 树节点类的初始化和销毁
TreeNode::TreeNode(char value, TreeNode* left, TreeNode* right, NodeTag
lTag, NodeTag rTag)
{
    data = value;
    leftChild = left;
    rightChild = right;
    this->lTag = lTag;
    this->rTag = rTag;
}

TreeNode::TreeNode(){

}

TreeNode::~TreeNode()
{
    if (leftChild != nullptr && lTag == Link) {
        delete leftChild;
        leftChild = nullptr;
    }
    if (rightChild != nullptr && rTag == Link) {
        delete rightChild;
        rightChild = nullptr;
    }
}
```

```
// 树节点类的打印
void TreeNode::printNode()
{
    cout << this->data;
}


char TreeNode::getdata() const{
    return data;
}


NodeTag TreeNode::getltag() const {
    return lTag;
}

NodeTag TreeNode::getrtag() const{
    return rTag;
}

TreeNode* TreeNode::getlchild() const{
    return leftChild;
}

TreeNode* TreeNode::getrchild() const{
    return rightChild;
}

// 树初始化 MyTree()，初始化一棵空树
MyTree:: MyTree()
{

}


// 树初始化 MyTree(const char[])，根据二叉树的先序序列，生成二叉树的二叉链表存储,使用@表示 NULL
MyTree::MyTree(const char* preorder)
{
    if (preorder == nullptr) {
        return;
    }

    int index = 0;
    root = constructTree(preorder, index);
    isThread = false;
```

```
}

TreeNode* MyTree::constructTree(const char* preorder, int& index)
{
    if (preorder[index] == '@') {
        index++;
        return nullptr;
    }

    TreeNode* node = new TreeNode(preorder[index], nullptr, nullptr,
Link, Link);
    index++;
    node->leftChild = constructTree(preorder, index);
    node->rightChild = constructTree(preorder, index);

    return node;
}


// 树的复制构造函数 MyTree(const MyTree&),复制参数中的树
MyTree::MyTree(const MyTree& other) {
    if (other.root == nullptr) {
        root = nullptr;
        return;
    }

    root = new TreeNode(other.root->data, nullptr, nullptr,
other.root->lTag, other.root->rTag);
    isThread = false;

    if (other.root->leftChild != nullptr) {
        root->leftChild = copySubtree(other.root->leftChild);
    }
    if (other.root->rightChild != nullptr) {
        root->rightChild = copySubtree(other.root->rightChild);
    }
}

TreeNode* MyTree::copySubtree(TreeNode* node) {
    if (node == nullptr) {
        return nullptr;
    }

    TreeNode* copyNode = new TreeNode(node->data, nullptr, nullptr,
node->lTag, node->rTag);
```

```cpp
    if (node->leftChild != nullptr) {
        copyNode->leftChild = copySubtree(node->leftChild);
    }
    if (node->rightChild != nullptr) {
        copyNode->rightChild = copySubtree(node->rightChild);
    }

    return copyNode;
}


// 树销毁~MyTree()支持普通二叉树与线索二叉树
MyTree::~MyTree() {
    deleteSubtree(root);

    clearThread(root);
}

void MyTree::deleteSubtree(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    deleteSubtree(node->leftChild);
    deleteSubtree(node->rightChild);

    delete node;
}

void MyTree::clearThread(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    if (node->lTag == Thread) {
        node->lTag = Link;
        node->leftChild = nullptr;
    } else {
        clearThread(node->leftChild);
    }

    if (node->rTag == Thread) {
        node->rTag = Link;
        node->rightChild = nullptr;
```

```
    } else {
        clearThread(node->rightChild);
    }
}


// 先序遍历二叉树
void MyTree::preOrderTraverse() {
    if(isThread == 0){
        if (root == nullptr) {
            return;
        }

        stack<TreeNode*> stk;
        stk.push(root);

        while (!stk.empty()) {
            TreeNode* node = stk.top();
            stk.pop();

            cout << node->data ;

            if (node->rightChild != nullptr) {
                stk.push(node->rightChild);
            }
            if (node->leftChild != nullptr) {
                stk.push(node->leftChild);
            }
        }
    }
}


// 中序遍历二叉树(普通二叉树和线索二叉树)
void MyTree::inOrderTraverse() {
    if (root == nullptr) {
        return;
    }
    stack<TreeNode*> stk;
    TreeNode* node = root;
    if(isThread == 0){
        while (node != nullptr || !stk.empty()) {
            while (node != nullptr) {
                stk.push(node);
                node = node->leftChild;
            }
```

```
            node = stk.top();
            stk.pop();

            cout << node->data;
            node = node->rightChild;
        }
    }
    else{
        while (node->lTag == Link) {
            node = node->leftChild;
        }

        TreeNode* firstNode = node;

        while (node != nullptr) {
            cout << node->data ;
            node = getNextInOrderNode(node);
            if (node->rTag == Thread && node->rightChild == firstNode)
{
                break;
        }
        }

    }

}

TreeNode* MyTree::getNextInOrderNode(TreeNode* node) {
    if (node->rTag == Link) {
        node = node->rightChild;
        while (node->lTag == Link) {
            node = node->leftChild;
        }
    } else {
        node = node->rightChild;
    }

    return node;
}

// 后序遍历二叉树
void MyTree::postOrderTraverse() {
    if (root == nullptr) {
        return;
```

```
    }

    stack<TreeNode*> stk;
    TreeNode* node = root;
    TreeNode* lastVisited = nullptr;

    while (node != nullptr || !stk.empty()) {
        while (node != nullptr) {
            stk.push(node);
            node = node->leftChild;
        }

        node = stk.top();
        if (node->rightChild == nullptr || node->rightChild ==
lastVisited) {
            cout << node->data;
            lastVisited = node;
            stk.pop();
            node = nullptr;
        } else {
            node = node->rightChild;
        }
    }
}


// 定位二叉树中的节点
TreeNode& MyTree::locateNode(const char& v) {
    stack<TreeNode*> stk;
    TreeNode* node = root;
    if(isThread == 0){
            if (root == nullptr) {
            return nullNode;
        }

        while (node != nullptr || !stk.empty()) {
            while (node != nullptr) {
                stk.push(node);
                node = node->leftChild;
            }

            node = stk.top();
            stk.pop();

            if (node->data == v) {
```

```
                return *node;
            }

            node = node->rightChild;
        }

        return nullNode;
    }

    else{
        while (node->leftChild != nullptr && node-> lTag == Link) {
            node = node->leftChild;

        }
        TreeNode* firstNode = node;

        while (node != nullptr) {
            if (node->data == v) {
                return *node;
            }
            if (node->rTag == Thread) {
                node = node->rightChild;
            } else {
                node = node->rightChild;
                while (node->leftChild != nullptr && node-> lTag ==
Link) {
                node = node->leftChild;
                }
            }
            if ( node == firstNode) {
                break;
        }
        }
        return nullNode;
    }

}


// 计算二叉树的叶子结点数
int MyTree::countLeaf() {
    if (root == nullptr) {
        return 0;
    }
```

```cpp
    stack<TreeNode*> stk;
    TreeNode* node = root;
    int count = 0;

    while (node != nullptr || !stk.empty()) {
        while (node != nullptr) {
            stk.push(node);
            node = node->leftChild;
        }

        node = stk.top();
        stk.pop();

        if (node->leftChild == nullptr && node->rightChild == nullptr) {
            count++;
        }

        node = node->rightChild;
    }

    return count;
}


// 计算二叉树的深度
int MyTree::countHeight() {
    if (root == nullptr) {
        return 0;
    }

    queue<TreeNode*> q;
    q.push(root);
    int height = 0;

    while (!q.empty()) {
        int size = q.size();

        while (size--) {
            TreeNode* node = q.front();
            q.pop();

            if (node->leftChild != nullptr) {
                q.push(node->leftChild);
            }
```

```
            if (node->rightChild != nullptr) {
                q.push(node->rightChild);
            }
        }

        height++;
    }

    return height;
}


// 当前树是否是线索二叉树
bool MyTree::isThreadedTree(){
    if(isThread == 0) return false;
    else return true;
}


// 为二叉树生成中序线索二叉树
bool MyTree::inOrderThreading() {
    if (root == nullptr) {
        return false;
    }

    isThread = true;

    TreeNode* head = new TreeNode();
    head->leftChild = root;

    TreeNode* pre = head;

    inOrderThreadingHelper(root, pre);

    pre->rightChild = head;
    pre->rTag = Thread;

    head->leftChild = pre;
    head->lTag = Thread;

    return true;
}

void MyTree::inOrderThreadingHelper(TreeNode* node, TreeNode*& pre) {
    if (node == nullptr) {
        return;
```

```
    }

    inOrderThreadingHelper(node->leftChild, pre);

    if (node->leftChild == nullptr) {
        node->lTag = Thread;
        node->leftChild = pre;
    }

    if (pre->rightChild == nullptr) {
        pre->rTag = Thread;
        pre->rightChild = node;
    }

    pre = node;

    inOrderThreadingHelper(node->rightChild, pre);
}


// 寻找中序线索二叉树中某节点的前驱节点
TreeNode& MyTree::preNode(const TreeNode& node) {
    if(node.getdata() == '@') return nullNode;
    if (node.getltag() == Thread) {
        TreeNode* p = node.getlchild();
        return *p;
    }
    else {
        TreeNode* p = node.getlchild();
        if (p != nullptr) {
            while (p->getrtag() == Link) {
                    p = p->getrchild();
                }
                return *p;
            } else {
                p = node.getrchild();
                return *p;
            }
    }
}


// 寻找中序线索二叉树中某节点的后继节点
TreeNode& MyTree::nextNode(const TreeNode& node) {
    if(node.getdata() == '@') return nullNode;
    if (node.getrtag() == Thread) {
```

```
        return *node.getrchild();
    } else {
        TreeNode* p = node.getrchild();
        while (p->getltag() == Link) {
            p = p->getlchild();
        }
        return *p;
    }
}

HuffmanTreeNode::HuffmanTreeNode(int value, HuffmanTreeNode* left,
HuffmanTreeNode* right)
{
    data = value;
    leftChild = left;
    rightChild = right;
}

HuffmanTreeNode::HuffmanTreeNode(){

}

HuffmanTreeNode::~HuffmanTreeNode()
{
    if (leftChild != nullptr ) {
        delete leftChild;
        leftChild = nullptr;
    }
    if (rightChild != nullptr ) {
        delete rightChild;
        rightChild = nullptr;
    }
}


int HuffmanTreeNode::getdata() const{
    return data;
}



HuffmanTreeNode* HuffmanTreeNode::getlchild() const{
    return leftChild;
}

HuffmanTreeNode* HuffmanTreeNode::getrchild() const{
```

```
    return rightChild;
}

// 哈夫曼树初始化
HuffmanTree:: HuffmanTree(const int& n, const int* weight) {
    priority_queue<HuffmanTreeNode*, vector<HuffmanTreeNode*>, Compare>
pq;
    for (int i = 0; i < n; i++) {
        pq.push(new HuffmanTreeNode(weight[i], nullptr, nullptr));

    }


class Compare {
public:
    bool operator()(const HuffmanTreeNode* a, const HuffmanTreeNode* b)
const {
        return (*a).getdata() > (*b).getdata();
    }
};


    // TreeNode* top = pq.top();
    //     cout << top->getdata() << endl;



    while (pq.size() > 1) {
        HuffmanTreeNode* left = pq.top();
        pq.pop();
        // cout << (*left).getdata() << endl;
        HuffmanTreeNode* right = pq.top();
        pq.pop();
        // cout << right->getdata() << endl;
        HuffmanTreeNode* parent = new HuffmanTreeNode((*left).getdata()
+ (*right).getdata(), left, right);
        // cout << parent->getdata() << endl;
        pq.push(parent);
    }

    root = pq.top();
    // cout << (*root).getdata() << endl;
}


class Compare {
```

```
public:
    bool operator()(const HuffmanTreeNode* a, const HuffmanTreeNode* b)
const {
        return (*a).getdata() > (*b).getdata();
    }
};



// 哈夫曼树销毁
HuffmanTree:: ~HuffmanTree(){
    destroyHuffmanTree(root);
}

void HuffmanTree::destroyHuffmanTree(HuffmanTreeNode* root) {
    if (root == nullptr) {
        return;
    }

    destroyHuffmanTree((*root).getlchild());
    destroyHuffmanTree((*root).getrchild());

    delete root;
}



void HuffmanTree::printHuffmanCodes() {
    string code = "";
    printHuffmanCodesHelper(root, code);
}

void HuffmanTree::printHuffmanCodesHelper(HuffmanTreeNode* node, string
code) {
    if (node == nullptr) {
        return;
    }
    if ((*node).getlchild() == nullptr && (*node).getrchild() ==
nullptr) {
        cout << (*node).getdata() << ":" << code  << endl;
    }

    printHuffmanCodesHelper((*node).getrchild(), code + "1");
    printHuffmanCodesHelper((*node).getlchild(), code + "0");

}
```

3. 实验结果

树

```
ABDGHJKCEFI
GDJHKBAECFI
GJKHDBEIFCA
5
5
I
ABCDE
CDEBA
EDCBA
1
5
0
1
A
B
E
E
B
```

哈夫曼树

```
56:1
9:011
8:010
7:001
3:0001
1:0000
```

4. 实验中遇到的问题及解决方法

理解了二叉树的存储结构。通过结点的左孩子和右孩子指针实现树的连接。结点包含值和两个指针。

实现了二叉树的基本操作,包括构造、复制构造、析构、定位结点、计算叶子结点数和树高等。这些都是理解二叉树很重要的基础。

实现了三种遍历方式:先序、中序和后序遍历。理解了它们的实现原理和区别。这三种遍历方式适用于不同的场景和应用。

实现了线索二叉树,理解了线索化的原理。线索二叉树在原树结构的基础上,对空指针进行了 Thread 标记,并指向其前驱或后继,从而实现空间优化和快速访问。

实现了哈夫曼树,理解了哈夫曼编码的原理。哈夫曼树是一种带权路径长度最短

的二叉树,用于数据的压缩编码。

实现过程中也体会到一些细节,如递归结束条件、空树的判断等。这些在实现树结构时需要注意。