

2023—2024 学年第二学期

《数据结构与算法导论》实验报告



班级： 2023211119

姓名： XXX

学号： 2023210XXX

班内序号： 07

报告日期： 2024-06-01

数据结构实验报告

实验名称： 实验 1——二叉树

学生姓名： XXX

班 级： 2023211119

班内序号： 07

学 号： 2023210XXX

日 期： 2024 年 6 月 1 日

1. 实验要求

[正文格式要求]

字体： 汉字宋体、英文 Times New Roman

字号： 五号

颜色： 黑色

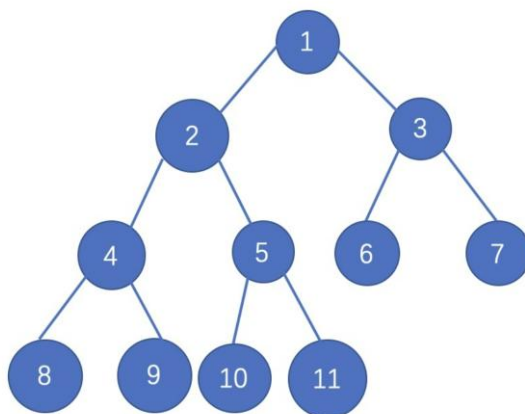
行距： 单倍行距

[内容要求]

- 1、二叉树的建立
- 2、前序遍历二叉树
- 3、中序遍历二叉树
- 4、后序遍历二叉树
- 5、按层序遍历二叉树
- 6、求二叉树的深度
- 7、求指定结点到根的路径

2. 程序分析

2.1 存储结构



2.2 关键算法分析

1. 关键算法:

a. 构造函数

a.1. 构造函数 BiTree()

```
BiTree() {  
    root = NULL;  
}
```

a.2. 从数组生成

```
BiTree(data[], n) {  
    root = NULL;  
    Create(root, data, 1, n); // 从数组的第一个元素开始创建二叉树  
}
```

b. 析构函数

```
~BiTree() {  
    Release(root); // 递归释放每个节点  
}
```

c. 递归创建二叉树

```
void Create(R, data[], i, n) {  
    if (i <= n and data[i-1] != '0') { // 0 代表空节点  
        R = new Node<T>; // 创建新节点  
        R.data = data[i-1];  
        R.lch = R.rch = NULL;  
        Create(R.lch, data, 2*i, n); // 创建左子树  
        Create(R.rch, data, 2*i+1, n); // 创建右子树  
    }  
}
```

d. 递归释放二叉树

```
void Release(R) {  
    if (R != NULL) {  
        Release(R.lch); // 释放左子树  
        Release(R.rch); // 释放右子树  
        delete R; // 删除当前节点  
    }  
}
```

e. 前序遍历

```
void PreOrder(R) {  
    if (R != NULL) {  
        cout << R.data << " "; // 访问根节点  
        PreOrder(R.lch); // 遍历左子树  
        PreOrder(R.rch); // 遍历右子树  
    }  
}
```

f. 中序遍历

```
void InOrder(R) {
    if (R != NULL) {
        InOrder(R.lch); // 遍历左子树
        cout << R.data << " "; // 访问根节点
        InOrder(R.rch); // 遍历右子树
    }
}
```

g.后序遍历

```
void PostOrder(R) {
    if (R != NULL) {
        PostOrder(R.lch); // 遍历左子树
        PostOrder(R.rch); // 遍历右子树
        cout << R.data << " "; // 访问根节点
    }
}
```

h.层序遍历

```
void LevelOrder(R) {
    queue<Node<T>*> q;
    q.push(R); // 根节点入队
    while (!q.empty()) {
        Node<T>* p = q.front(); // 队首元素出队
        q.pop();
        cout << p.data << " "; // 访问节点
        if (p.lch != NULL) q.push(p.lch); // 左子节点入队
        if (p.rch != NULL) q.push(p.rch); // 右子节点入队
    }
}
```

i.节点总数

```
int GetTotalNode(R) {
    if (R == NULL) return 0; // 空树节点数为 0
    return GetTotalNode(R.lch) + GetTotalNode(R.rch) + 1; // 左子树节点数 + 右子
    树节点数 + 根节点
}
```

k.二叉树高度

```
int getheight(R) {
    if (R == NULL) return 0; // 空树的高度为 0
    int m = getheight(R.lch); // 左子树高度
    int n = getheight(R.rch); // 右子树高度
    return (m > n) ? m + 1 : n + 1; // 左右子树高度较大者加 1
}
```

l.二叉树叶子节点数

```
int countleaf(R) {
    if (R == NULL) return 0; // 空树叶子节点数为 0
    if (R.lch == NULL and R.rch == NULL) return 1; // 叶子节点
}
```

```

        return countleaf(R.lch) + countleaf(R.rch); // 左子树叶子节点数 + 右子树叶子
        节点数
    }

```

m. 寻找目标节点到根节点的路径

```

void noderootPath(target) {
    stack<Node<T>*> stk;
    searchPath(target, root, stk); // 搜索路径
    if (stk.empty()) {
        cout << "No path" << endl; // 没有找到路径
    } else {
        cout << "Path: ";
        while (!stk.empty()) {
            Node<T>* out = stk.top();
            if (stk.size() == 1) {
                cout << out->data;
            } else {
                cout << out->data << "->";
            }
            stk.pop();
        }
        cout << endl;
    }
}

```

2. 代码详细分析

a. 构造函数

a.1. 构造函数 BiTree()

初始化根节点 root 为 NULL。

a.2. 从数组生成

带参数的构造函数，使用数组 data 来创建二叉树。数组中的元素按照层序遍历的顺序排列，n 是数组的长度。

b. 析构函数

递归释放二叉树的内存。

c. 递归创建二叉树

用于根据数组 data 创建二叉树。i 代表当前处理的数组元素的位置，n 是数组的长度。如果 data[i-1] 不等于 '0'（空节点），则创建新的节点 R 并递归创建其左右子树。

d. 递归释放二叉树

用于释放二叉树的内存。首先释放左右子树，然后删除当前节点。

e. 前序遍历

首先访问根节点，然后递归地前序遍历左子树和右子树。

f. 中序遍历

首先递归地中序遍历左子树，然后访问根节点，最后递归地中序遍历右子树。

g. 后序遍历

首先递归地后序遍历左子树和右子树，然后访问根节点。

h.层序遍历

层次遍历（广度优先遍历）二叉树。使用队列来逐层访问节点。

i.节点总数

递归计算左子树和右子树的节点数，然后加 1。

k.二叉树高度

递归计算左子树和右子树的高度，返回较大的一个加 1。

l.二叉树叶子节点数

如果节点是叶子节点，则返回 1，否则返回左子树和右子树叶子节点数量的和。

m.寻找目标节点到根节点的路径

寻找从根节点到目标节点 `target` 的路径。使用 `searchPath` 函数来找到路径，并将节点压入栈中，然后从栈中弹出节点并打印路径。

3.时间和空间复杂度（n 是数组的大小）**a.构造函数****a.1.构造函数 `BiTree()`**

时间复杂度： $O(1)$ ；空间复杂度： $O(1)$

a.2.从数组生成

时间复杂度： $O(n)$ ；空间复杂度： $O(\log n) \sim O(n)$

b.析构函数

时间复杂度： $O(n)$ ；空间复杂度： $O(\log n) \sim O(n)$

c.递归创建二叉树

时间复杂度： $O(n)$ ；空间复杂度： $O(\log n) \sim O(n)$

d.递归释放二叉树

时间复杂度： $O(n)$ ；空间复杂度： $O(\log n) \sim O(n)$

e.前序遍历

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

f.中序遍历

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

g.后序遍历

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

h.层序遍历

时间复杂度： $O(n)$ ；空间复杂度： $O(w)$ ，其中 w 是树的宽度

i.节点总数

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

k.二叉树高度

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

l.二叉树叶子节点数

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

m.寻找目标节点到根节点的路径

时间复杂度： $O(n)$ ；空间复杂度： $O(h)$ ，其中 h 是树的高度

3. 程序运行结果

测试主函数流程：

```

int main()
{
    char data[]="ABCDE0F0G0H00IJ";//0 代表空节点,从 1 开始存储,从上到下, 从左到
    右,根节点为 A,左孩子为 B,右孩子为 C,左孩子的左孩子为 D,右孩子的右孩子为 J.
    BiTree<char> tree(data,15);//15 代表节点个数
    cout<<"PreOrder: ";//前序遍历
    tree.PreOrder(tree.GetRoot());
    cout<<endl;
    cout<<"InOrder: ";//中序遍历
    tree.InOrder(tree.GetRoot());
    cout<<endl;
    cout<<"PostOrder: ";//后序遍历
    tree.PostOrder(tree.GetRoot());
    cout<<endl;
    cout<<"LevelOrder: ";//层次遍历
    tree.LevelOrder(tree.GetRoot());
    cout<<endl;
    cout<<"Total Node: "<<tree.GetTotalNode(tree.GetRoot())<<endl;//节点个数
    cout<<"Height: "<<tree.getheight(tree.GetRoot())<<endl;//树的高度
    cout<<"Leaf Node: "<<tree.countleaf(tree.GetRoot())<<endl;//叶子节点个数
    tree.noderootPath('H');//节点 H 到根节点的路径
    return 0;
}

```

测试结果:

```

PreOrder: A B D G E H C F I J
InOrder: D G B E H A C I F J
PostOrder: G D H E B I J F C A
LevelOrder: A B C D E F G H I J
Total Node: 10
Height: 4
Leaf Node: 4
Path: A->B->E->H

```

```

PS C:\Users\18055\Desktop\Data_Structur_Algo>
PS C:\Users\18055\Desktop\Data_Structur_Algo> & 'c:\Users\18055\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-ptiwhn3z.vrt' '--stdout=Microsoft-MIEngine-Out-15v4m3eu.4ta' '--stderr=Microsoft-MIEngine-Error-n3r3klml.g2d' '--pid=Microsoft-MIEngine-Pid-fsqouul2.14c' '--dbgExe=C:\Program Files (x86)\mingw64\bin\gdb.exe' '--interpreter=mi'
PreOrder: A B D G E H C F I J
InOrder: D G B E H A C I F J
PostOrder: G D H E B I J F C A
LevelOrder: A B C D E F G H I J
Total Node: 10
Height: 4
Leaf Node: 4
Path: A->B->E->H
PS C:\Users\18055\Desktop\Data_Structur_Algo>

```

测试结论: 功能实现完全。

实验名称： 实验 2——哈夫曼编/解码器

学生姓名： XXX

班 级： 2023211119

班内序号： 07

学 号： 2023210XXX

日 期： 2024 年 6 月 1 日

1. 实验要求

[正文格式要求]

字体：汉字宋体、英文 Times New Roman

字号：五号

颜色：黑色

行距：单倍行距

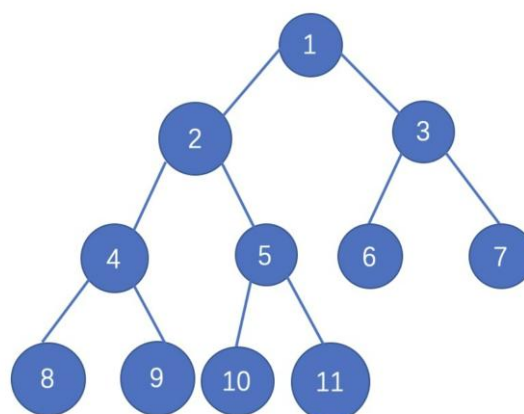
[内容要求]

- 1、初始化(Init): 能够对输入的任意长度的字符串 s 进行统计，统计每个字符的频度，并建立哈夫曼树
- 2、建立编码表(CreateTable): 利用已经建好的哈夫曼树进行编码，并将每个字符的编码输出。
- 3、编码(Encoding): 根据编码表对输入的字符串进行编码，并将编码后的字符串输出。
- 4、译码(Decoding): 利用已经建好的哈夫曼树对编码后的字符串进行译码，并输出译码结果。

2. 程序分析

2.1 存储结构

栈



2.2 关键算法分析

1.关键算法:

a.构造函数

```
// 构造函数，接受一个字符串并构建 Huffman 树
HuffmanCodec(const string& s) {
    buildTree(s);
}
```

b.析构函数

```
~HuffmanCodec(const string& s) {
    delete root;
}
```

c.buildTrees 函数

```
// 辅助函数，用于构建 Huffman 树
void buildTree(const string& s) {
    // 初始化一个大小为 256 的向量，用于存储字符频率
    vector<int> frequency(256, 0);

    // 遍历字符串，统计每个字符出现的频率
    for each char c in s {
        frequency[c]++;
    }

    // 创建一个优先队列，用于存储 Huffman 树节点
    // 队列中的节点按照字符频率排序
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;
    for each int i from 0 to 255 {
        // 如果字符频率大于 0，则创建节点并加入优先队列
        if (frequency[i] > 0) {
            pq.push(new HuffmanNode(i, frequency[i]));
        }
    }

    // 构建 Huffman 树的过程
    // 当队列中有多于一个节点时，重复以下步骤：
    while (pq.size() > 1) {
        // 取出频率最小的两个节点
        HuffmanNode* left = pq.top(); pq.pop();
        HuffmanNode* right = pq.top(); pq.pop();

        // 创建一个新的内部节点，其频率为两个节点频率之和
        HuffmanNode* newNode = new HuffmanNode('$', left->frequency +
right->frequency);
        // 新节点的左右子节点分别指向这两个节点
        newNode->left = left;
        newNode->right = right;
    }
}
```

```
    // 将新节点加入优先队列
    pq.push(newNode);
}

// 最后一个留在队列中的节点即为 Huffman 树的根节点
root = pq.top();
}
```

d.生成 Huffman 编码表

```
void generateCodes(HuffmanNode* node, string code, vector<string>& codes) {
    // 如果节点为空，直接返回
    if (node == nullptr) {
        return;
    }

    // 如果节点是叶子节点（即存储了实际的字符），则将编码添加到 codes 向量中
    if (node->data != '$') {
        codes[node->data] = code;
    }

    // 递归调用 generateCodes 函数，为左子树和右子树生成编码
    // 左子树编码在当前编码后添加 "0"，右子树编码添加 "1"
    generateCodes(node->left, code + "0", codes);
    generateCodes(node->right, code + "1", codes);
}
```

e.打印 Huffman 树中每个字符的编码

```
// 公开函数，用于打印 Huffman 树中每个字符的编码
void printCodes(HuffmanNode* node, string str) {
    // 如果节点为空，直接返回
    if (node == nullptr) {
        return;
    }

    // 如果节点是叶子节点，则打印字符和对应的编码
    if (node->data != '$') {
        cout << node->data << ": " << str << "\n";
    }

    // 递归打印左子树和右子树的编码
    printCodes(node->left, str + "0");
    printCodes(node->right, str + "1");
}
```

f.将字符串编码成 Huffman 编码

```
// 公开函数，用于将字符串编码为 Huffman 编码
string encode(const string& s) {
```

```
// 初始化一个向量，用于存储每个字符对应的 Huffman 编码
vector<string> codes(256, "");
// 生成 Huffman 编码
generateCodes(root, "", codes);

// 初始化编码后的字符串
string encodedString = "";
// 遍历输入字符串，将每个字符对应的 Huffman 编码添加到编码后的字符串中
for each char c in s {
    encodedString += codes[c];
}

// 返回编码后的字符串
return encodedString;
}

g. 将 Huffman 编码解码成字符串
// 公开函数，用于将 Huffman 编码解码回原始字符串
string decode(const string& encodedString) {
    // 初始化解码后的字符串
    string decodedString = "";
    // 初始化当前节点为根节点
    HuffmanNode* current = root;

    // 遍历编码字符串中的每个字符
    for each char c in encodedString {
        // 根据当前字符是 '0' 还是 '1'，移动到相应的子节点
        if (c == '0') {
            current = current->left;
        } else {
            current = current->right;
        }

        // 如果当前节点是叶子节点（即存储了实际的字符），则将其添加到解码后的字符串中
        // 并将当前节点重置为根节点，以便继续解码剩余的编码
        if (current->data != '$') {
            decodedString += current->data;
            current = root;
        }
    }

    // 返回解码后的字符串
    return decodedString;
}
```

```
};
```

2. 代码详细分析

a. 构造函数

使用输入字符串 `s` 调用 `buildTrees` 函数初始化 Huffman 树。

b. 析构函数

释放 Huffman 树的内存。

c. `buildTrees` 函数

使用 `vector<int>` 统计输入字符串 `s` 中每个字符的频率。

创建一个优先队列 `pq`，存储按照频率排序的 `HuffmanNode` 指针。

遍历优先队列，每次取出两个频率最小的节点，创建一个新的内部节点（用 '\$' 表示），其频率为两个节点频率之和，并将新节点加入优先队列。

最后，优先队列中剩下的节点即为 Huffman 树的根节点。

d. 生成 Huffman 编码表

如果节点为空，返回。

如果节点是叶子节点（即存储了实际字符），则将当前编码添加到 `codes` 向量中。

递归地为左子节点和右子节点生成编码，左子节点的编码在当前编码后添加 "0"，右子节点添加 "1"。

e. 打印 Huffman 树中每个字符的编码

如果节点是叶子节点，则打印字符和对应的编码。递归打印左子树和右子树的编码。

f. 将字符串编码成 Huffman 编码

遍历输入字符串，将每个字符的编码拼接成编码后的字符串。

g. 将 Huffman 编码解码成字符串

初始化解码后的字符串 `decodedString`。从根节点开始，根据编码字符串中的 "0" 和 "1"，递归遍历 Huffman 树。当到达叶子节点时，将字符添加到解码字符串中，并重置为根节点继续解码。

3. 时间和空间复杂度（ n 是数组的大小）

a. 构造函数

时间复杂度： $O(k \log k)$ ；空间复杂度： $O(k)$ k 是不同字符的数量

b. 析构函数

时间复杂度： $O(k)$ ；空间复杂度： $O(1)$ k 是不同字符的数量

c. `buildTrees` 函数

时间复杂度： $O(k \log k)$ ；空间复杂度： $O(k)$ k 是不同字符的数量

d. 生成 Huffman 编码表

时间复杂度： $O(m)$ ；空间复杂度： $O(\log m)$ m 是 Huffman 树中节点的数量

e. 打印 Huffman 树中每个字符的编码

时间复杂度： $O(m)$ ；空间复杂度： $O(\log m)$ m 是 Huffman 树中节点的数量

f. 将字符串编码成 Huffman 编码

时间复杂度： $O(m * n)$ ；空间复杂度： $O(n)$ n 是字符串长度， m 是平均编码长度。

g. 将 Huffman 编码解码成字符串

时间复杂度： $O(m * n)$ ；空间复杂度： $O(n)$ n 是字符串长度， m 是树的高度。

3. 程序运行结果

测试主函数流程:

```
int main() {
    string inputString = "";
    getline(cin, inputString);
    HuffmanCodec huffmanCodec(inputString);

    string encodedString = huffmanCodec.encode(inputString);
    cout << "Encoded string: " << encodedString << endl;

    string decodedString = huffmanCodec.decode(encodedString);
    cout << "Decoded string: " << decodedString << endl;

    huffmanCodec.printCodes(huffmanCodec.getroot(), "");
    return 0;
}
```

测试结果:

I love data Structure, I love Computer. I will try my best to study data Structure.

Encoded string:
10100111010100000011011011111010101001000100111001001001100101101101100101111001
1010011101111010011101010000001101101111011000000000100010111011100110111000001
1111101001110110010011110101010111110011000111111000100111111001010110101110100
111100000011011101001011101010111111101010100100010011100100100110010110110110
010111100110100011

Decoded string: I love data Structure, I love Computer. I will try my best to study data Structure.

o: 0000

m: 00010

.: 00011

S: 00100

b: 001010

p: 001011

v: 00110

,: 001110

i: 001111

a: 0100

l: 0101

C: 011000

w: 011001

c: 01101

s: 01110

y: 01111

t: 100

I: 10100

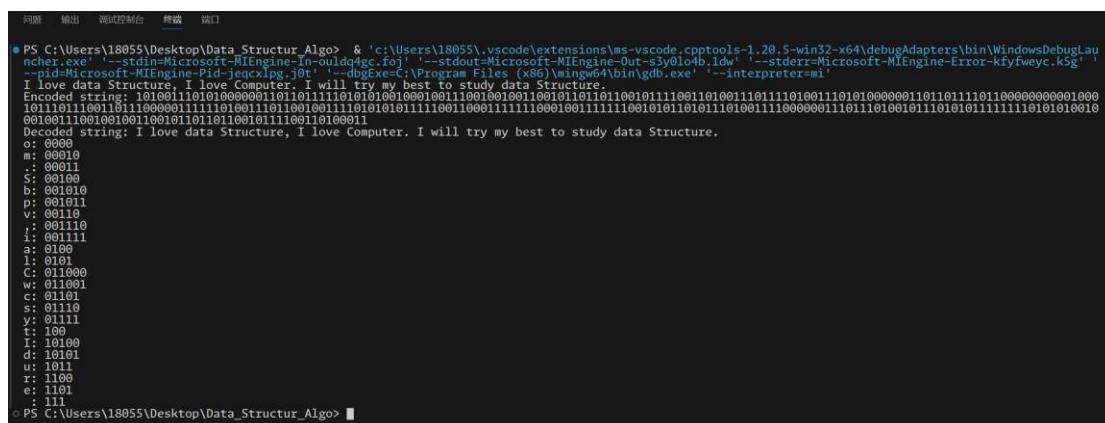
d: 10101

u: 1011

r: 1100

e: 1101

: 111



```
PS C:\Users\18055\Desktop\Data_Structur_Algo> & 'c:\Users\18055\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugAdapters\bin\WindowsDebugLaunche
r.exe' '--stdin=Microsoft-MIEngine-In-build\mc.foj' '--stdout=Microsoft-MIEngine-Out-s3y0lo4b.ldw' '--stderr=Microsoft-MIEngine-Error-kfyfweyc.k5g' '--pid=Microsoft-MIEngine-Pid-jeacklpg.j0t' '--dbgExe=c:\Program Files (x86)\mingw64\bin\gdb.exe' '--interpreter=mi'
I love data Structure, I love Computer. I will try my best to study data Structure.
Encoded string: 101001110101000000110110111101010100010011001001001100101101101100101110011010011010000001101101110110000000001000
101110111001101110000011110100111011001111010101110011000111110001001111100100111110010101011010111000000110110100111010101111101010010
00100111001001001001010101100101110010100011
Decoded string: I love data Structure, I love Computer. I will try my best to study data Structure.
o: 0000
e: 00010
.: 00011
S: 00100
b: 001010
p: 001011
v: 00110
.: 001110
i: 001111
a: 0100
l: 0101
C: 011000
w: 011001
c: 01101
s: 01110
y: 01111
t: 100
I: 10100
d: 10101
u: 1011
r: 1100
e: 1101
.: 111
PS C:\Users\18055\Desktop\Data_Structur_Algo>
```

测试结论：功能实现完全。

4. 总结

心得体会：

通过分析这两个文件，我主要了解了：

二叉树的多种遍历方法及其实现。

递归在二叉树构建和遍历中的应用。

哈夫曼编码的原理和实现，以及它在数据压缩中的应用。

如何使用优先队列来管理节点，以及自定义比较函数的重要性。