2023—2024 学年第二学期

《数据结构与算法导论》实验报告

严禁抄袭！

班级：_____

姓名：_____

学号：_____

班内序号：_____

报告日期：_____

数据结构实验报告

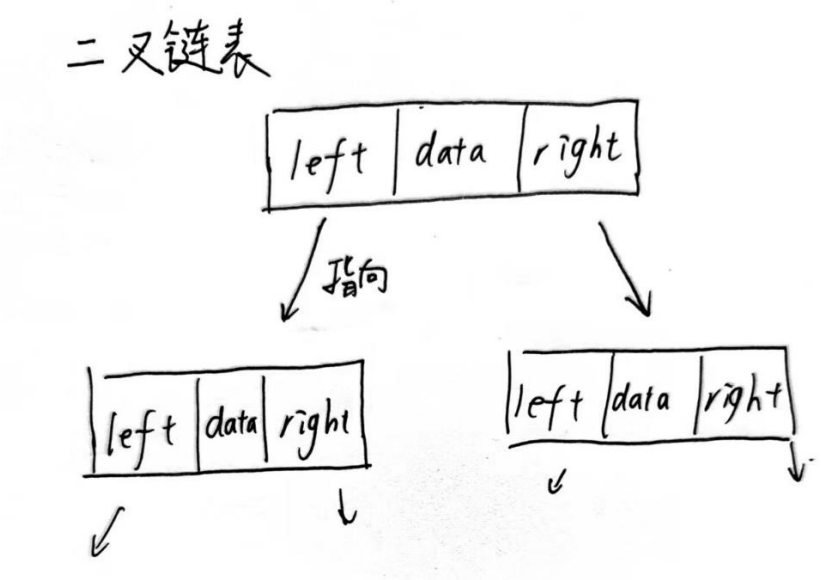
1．实验要求

根据二叉树的抽象数据类型的定义，使用二叉链表实现一个二叉树。
二叉树的基本功能：
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
8、二叉树的销毁　　9、其他：自定义操作
编写测试 main()函数测试二叉树的正确性

2. 程序分析

2.1 存储结构

使用二叉链表存储，每一个节点由三个域构成，分别为左右子树指针域和数据域



2.2 关键算法分析

关键算法过程描述

1. 创建树（CreateTree）：

自然语言描述：
该算法递归地从数组中创建二叉树。对于每个节点，如果它的索引在数组范围内且不为#，则创建一个新节点，并递归地为其左子节点和右子节点创建子树。

伪代码：
如果 i 大于 n 或 data[i-1] 为 #，则返回 NULL。
创建一个新节点 R，并将 data[i-1] 赋值给 R->data。
递归调用 CreateTree 为 R->lchlid 传递 2*i 作为索引。
递归调用 CreateTree 为 R->rchlid 传递 2*i+1 作为索引。

代码步骤：

```
template<class T>
void BiTree<T>::CreateTree(BiNode<T>* &R, T data[], int n, int i) {
    if (i <= n && data[i-1] != '#') {
        R = new BiNode<T>;
        R->data = data[i-1];
        R->lchlid = NULL;
        R->rchlid = NULL;
        CreateTree(R->lchlid, data, n, 2 * i);
        CreateTree(R->rchlid, data, n, 2 * i + 1);
    } else {
        R = NULL;
    }
}
```

2. 前序遍历（PreOrder）：

自然语言描述：
该算法从根节点开始，首先访问根节点，然后递归地访问左子树和右子树。

伪代码：
如果 R 不为空，执行以下操作：
    输出 R->data。
    递归调用 PreOrder 访问 R->lchlid。
    递归调用 PreOrder 访问 R->rchlid。

代码步骤：

```
template<class T>
void BiTree<T>::PreOrder(BiNode<T>* R) {
    if (R != NULL) {
        cout << R->data << " ";
        PreOrder(R->lchlid);
        PreOrder(R->rchlid);
    }
}
```

3. 中序遍历（InOrder）：

严禁抄袭！

自然语言描述：
该算法从根节点开始，首先递归地访问左子树，然后访问根节点，最后递归地访问右子树。

伪代码：
如果 R 不为空，执行以下操作：
递归调用 InOrder 访问 R->lchlid。
输出 R->data。
递归调用 InOrder 访问 R->rchlid。

代码步骤：

```
template<class T>
void BiTree<T>::InOrder(BiNode<T>* R) {
    if (R != NULL) {
        InOrder(R->lchlid);
        cout << R->data << " ";
        InOrder(R->rchlid);
    }
}
```

4. 后序遍历（PostOrder）：

自然语言描述：
该算法从根节点开始，首先递归地访问左子树，然后访问右子树，最后访问根节点。

伪代码：
如果 R 不为空，执行以下操作：
递归调用 PostOrder 访问 R->lchlid。
递归调用 PostOrder 访问 R->rchlid。
输出 R->data。

代码步骤：

```
template<class T>
void BiTree<T>::PostOrder(BiNode<T>* R) {
    if (R != NULL) {
        PostOrder(R->lchlid);
        PostOrder(R->rchlid);
        cout << R->data << " ";
    }
}
```

5. 层序遍历（LevelOrder）：

自然语言描述：
该算法使用一个队列从根节点开始进行层序遍历。首先将根节点入队，然后每次从队列中取出一个节点，

严禁抄袭！

访问它并将它的左子节点和右子节点依次入队，直到队列为空。

伪代码：

如果 R 不为空，执行以下操作：
    将 R 入队。
    当队列不为空时，执行以下循环：
        将队列的前端节点 temp 出队。
        输出 temp->data。
        如果 temp->lchlid 不为空，将其入队。
        如果 temp->rchlid 不为空，将其入队。

代码步骤：

```
template<class T>
void BiTree<T>::LevelOrder(BiNode<T>* R) {
    queue<BiNode<T>*> OrderQueue;
    if (R != NULL) {
        OrderQueue.push(R);
        while (!OrderQueue.empty()) {
            BiNode<T>* temp = OrderQueue.front();
            OrderQueue.pop();
            cout << temp->data << " ";
            if (temp->lchlid != NULL) OrderQueue.push(temp->lchlid);
            if (temp->rchlid != NULL) OrderQueue.push(temp->rchlid);
        }
    }
}
```

6. 获取树的深度（GetTreeDepth）：

自然语言描述：

该算法递归地计算左子树和右子树的深度，并返回两者中的较大者加 1。

伪代码：

如果 R 为空，返回 0。
递归调用 GetTreeDepth 计算左子树的深度 leftsize。
递归调用 GetTreeDepth 计算右子树的深度 rightsize。
返回 max(leftsize, rightsize) + 1。

代码步骤：

```
template<class T>
int BiTree<T>::GetTreeDepth(BiNode<T>* R) {
    if (R == NULL) return 0;
    int leftsize = GetTreeDepth(R->lchlid);
    int rightsize = GetTreeDepth(R->rchlid);
```

严禁抄袭！

```
    return max(leftsize, rightsize) + 1;
}
```

7. 查找路径（FindPath）：

自然语言描述：
该算法使用辅助函数 isfind 从根节点开始查找目标节点，并将从根节点到目标节点的路径存储在一个栈中。如果找到了目标节点，则打印路径，否则输出未找到。

伪代码：
初始化一个栈 Path。
调用 isfind，如果返回 true，则输出路径 Path。
如果返回 false，则输出未找到路径。

代码步骤：

```
template<class T>
void BiTree<T>::FindPath(BiNode<T>* R, T x) {
    stack<T> Path;
    bool IsFind = isfind(R, x, Path);
    if (!IsFind) cout << "未找到路径";
    else {
        cout << "找到路径，如下：" << endl;
        PrintStack(Path);
    }
}
```

辅助函数 isfind 自然语言描述：
该算法递归地查找目标节点。如果当前节点是目标节点，则将其入栈并返回 true。否则，将当前节点入栈并递归地查找左子树和右子树。如果找到目标节点，则返回 true。如果未找到，则将当前节点出栈并返回 false。

伪代码：
如果 R 为空，返回 false。
如果 R->data 等于 x，将 R->data 入栈，返回 true。
将 R->data 入栈。
初始化 flag 为 false。
如果 flag 为 false 且 R->lchlid 不为空，递归调用 isfind 查找左子树。
如果 flag 为 false 且 R->rchlid 不为空，递归调用 isfind 查找右子树。
如果 flag 仍为 false，则将当前节点出栈。
返回 flag。

代码步骤：

```
template<class T>
bool BiTree<T>::isfind(BiNode<T>* R, T x, stack<T> &Path) {
```

```
    if (R == NULL) return false;
    if (R->data == x) {
        Path.push(R->data);
        return true;
    }
    Path.push(R->data);
    bool flag = false;
    if (!flag && R->lchlid != NULL) {
        flag = isfind(R->lchlid, x, Path);
    }
    if (!flag && R->rchlid != NULL) {
        flag = isfind(R->rchlid, x, Path);
    }
    if (!flag) Path.pop();
    return flag;
}
```

8. 释放树的内存（Release）：

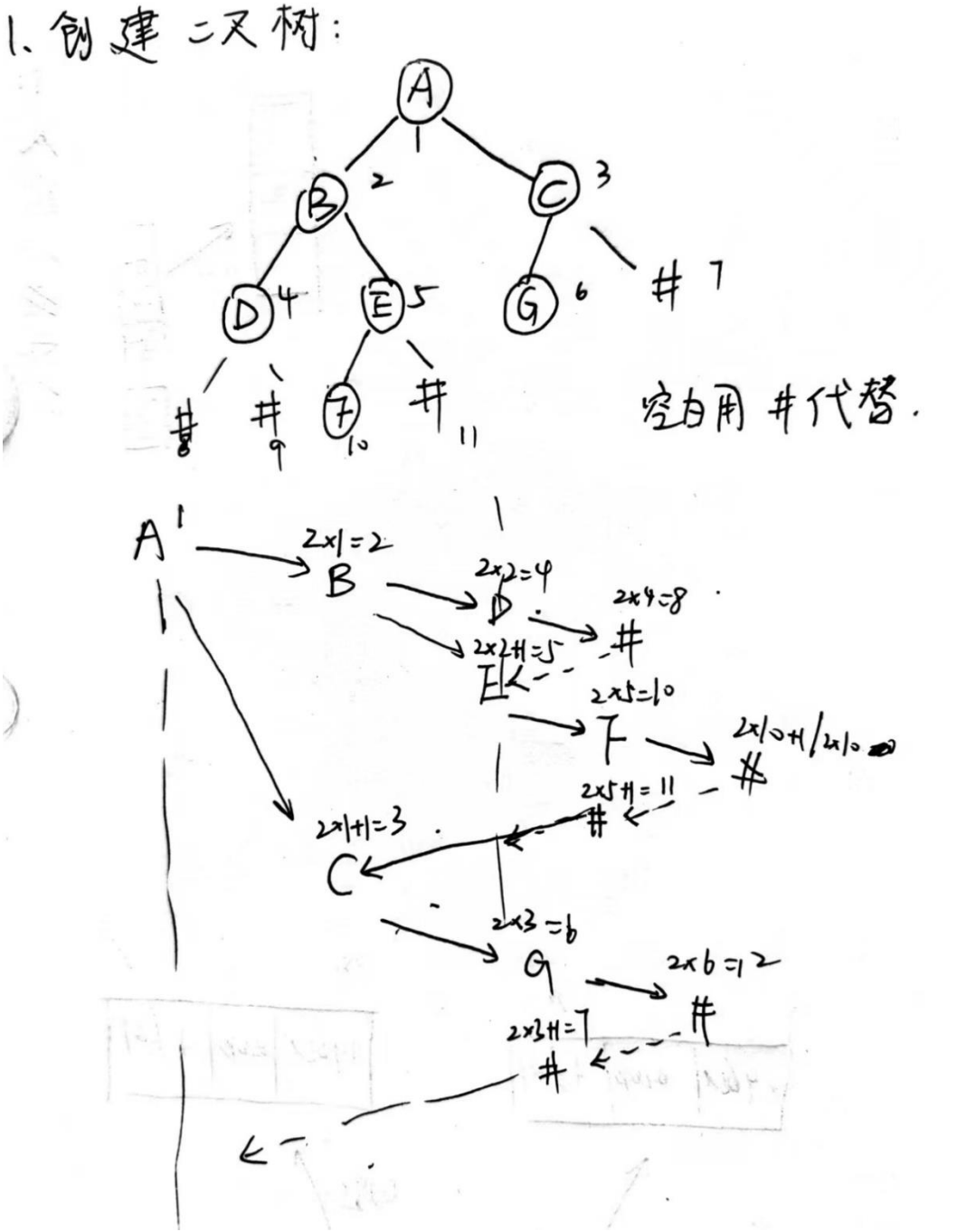
自然语言描述：
该算法递归地删除树中的所有节点，从叶子节点开始，逐步向上删除。

伪代码：
如果 R 不为空，执行以下操作：
递归调用 Release 释放 R->lchlid。
递归调用

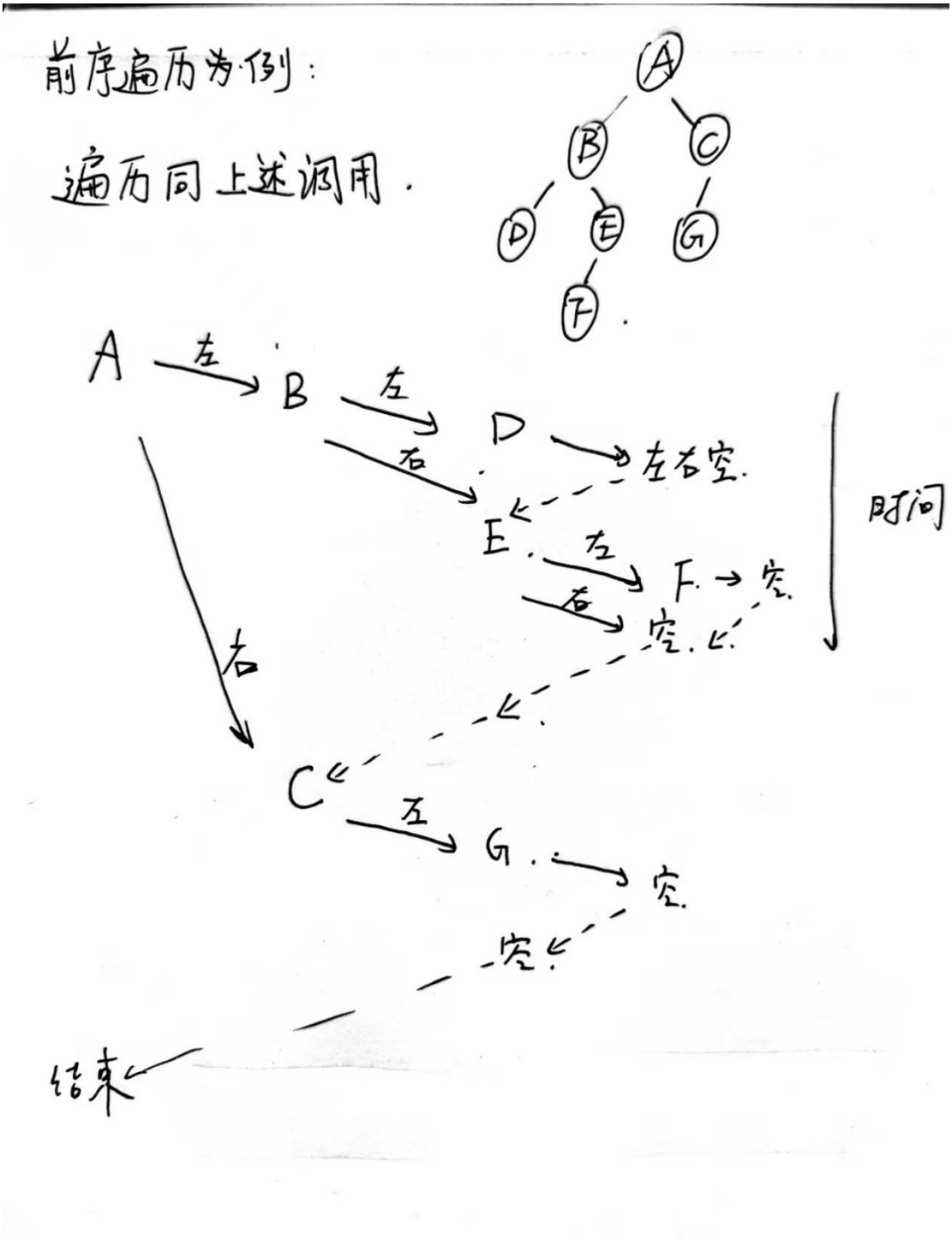
手绘流程图如下：

1、 创建二叉树：

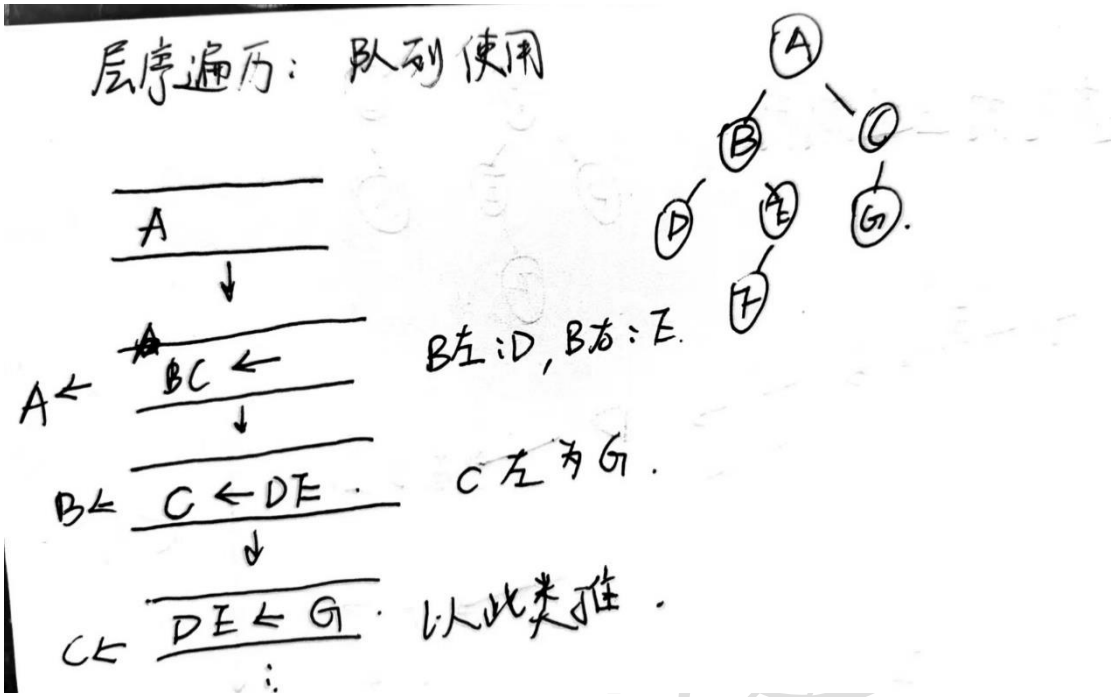
严禁抄袭！


1、创建二叉树：
A
B
C
D
E
G
#
空白用#代替.
2x1=2
2x2=4
2x4=8
2x2+1=5
2x5=10
2x10+1/2x10
2x5+1=11
2x1+1=3
2x3=6
2x6=12
2x3+1=7

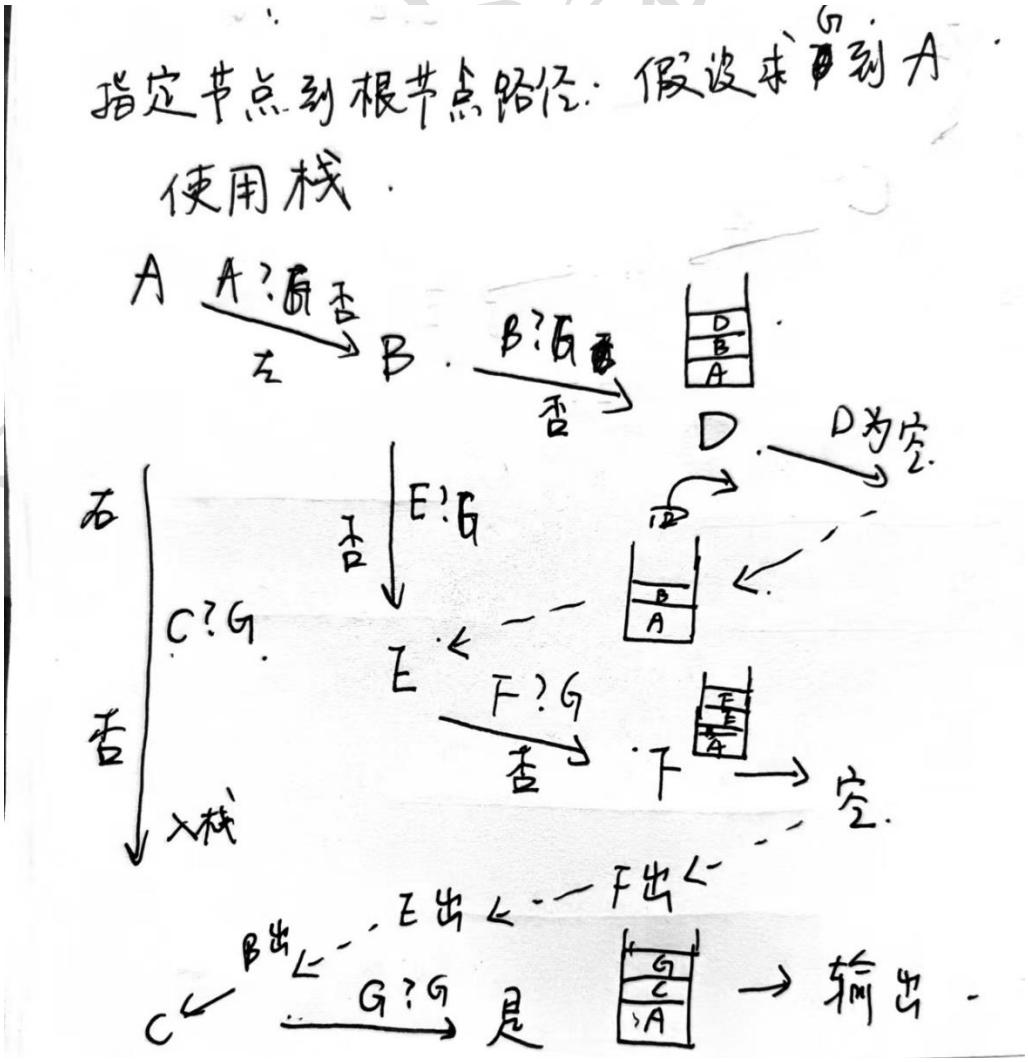
2、 前序遍历二叉树：



3、 层序遍历：


层序遍历：队列使用
A
BC
C←DE
DE←G
B左:D, B右:E.
C左为G.
以此类推.

4、 寻找路径：


指定节点到根节点路径：假设求G到A
使用栈
A
A?G
否
左
B
B?G
否
D
D为空
右
C?G
否
入栈
E?G
否
E
F?G
否
下
空
F出
E出
B出
C
G?G
是
输出

5、

2.3 其他

3. 程序运行结果

1. 用户输入字符序列构建二叉树。
2. 输出提示信息，开始构建二叉树。
3. 用户输入字符序列。
4. 计算输入字符序列的长度。
5. 调用 CreateTree 方法构建二叉树。
6. 输出提示信息，开始遍历二叉树。
7. 调用 PreOrder 方法进行前序遍历，输出遍历结果。
8. 调用 PostOrder 方法进行后序遍历，输出遍历结果。
9. 调用 InOrder 方法进行中序遍历，输出遍历结果。
10. 调用 LevelOrder 方法进行层序遍历，输出遍历结果。
11. 输出提示信息，计算二叉树的深度。
12. 调用 GetTreeDepth 方法获取二叉树的深度，输出深度值。
13. 输出提示信息，开始查找路径。
14. 用户输入要查找的节点。
15. 调用 FindPath 方法查找从指定节点到根节点的路径，输出路径信息。
16. 输出提示信息，进行析构测试。
17. 调用析构函数释放二叉树内存。
18. 程序暂停，等待用户按下任意键结束程序。、

测试条件：

输入字符序列："ABD#CEF"。
期望构建一棵如下所示的二叉树：

测试结论：

成功构建了上述二叉树。
各种遍历方法得到的结果符合预期。
二叉树的深度为 3。
查找从指定节点到根节点的路径结果正确。
测试结果如图

严禁抄袭！

4. 总结

1、在传参时，根节点应该以什么形式传入函数是需要考虑的，有时需要以引用的方式传入，以改变根节点的值，传入参数不合适，有时候找不到根节点本体

2、多次使用递归方法，这个方法自己想并不容易想到，而且在调试的时候，递归也不好跟踪，这一次学会了递归的思路，而且学会了调试递归函数。

3、递归调用时候很多需要在函数外面写一个辅助函数，这样不仅调用清楚，逻辑更加清晰，可读性也更好

严禁抄袭！

实验名称： 题目二：哈夫曼编/解码器

学生姓名： 赵明远

班　　级： 2023211118

班内序号： 33

学　　号： 2023210249

日　　期： 2024 年 6 月 5 日

1．实验要求

利用二叉树结构实现哈夫曼编/解码器。

基本要求：

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。

5、打印(Print)：以直观的方式打印哈夫曼树（选作）

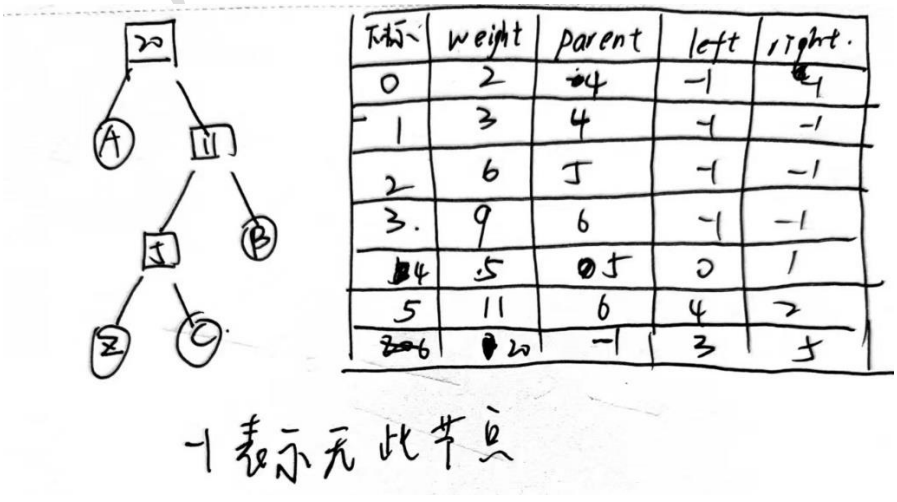
6、计算输入的字符串编码前和编码后的长度，并进行分析，讨论赫夫曼编码的压缩效果。

7、可采用二进制编码方式（选作）

2. 程序分析

2.1 存储结构

使用静态三叉链表实现，存储结构为数组，包含三个指针域和一个数据域，指针域保存的是双亲节点和孩子节点的数组下标。



2.2 关键算法分析

关键算法分析：

详细步骤：

1. CountWeight：

CountWeight(weight[], num, ch[], content[])

// 输入：权重数组 weight，字符数量 num，输入字符序列 ch，字符数组 content

// 输出：无

1. 初始化数组 temp[]，将所有元素置零
2. 初始化变量 numnow = 0
3. 遍历输入字符序列 ch：
    1. 获取当前字符 ch[i]
    2. 将字符转换为整数 temp[ch[i]]++
    3. 如果当前字符不在字符数组 content 中，则将其加入，并增加 numnow
4. 将 temp[] 中的频率复制到权重数组 weight[] 中
5. 将 content[] 最后一位置零

2. CreateHTree：

CreateHTree(weight[], category, content[])

// 输入：权重数组 weight，字符种类数目 category，字符数组 content

// 输出：哈夫曼树 HTree

1. 初始化哈夫曼树 HTree[]
2. 初始化编码表 HCodeTable[]
3. 对于每个字符种类：
    1. 设置叶子节点信息：HTree[i].weight = weight[i]，HTree[i].lchild = HTree[i].rchild = HTree[i].parent = -1
    2. 设置编码表信息：HCodeTable[i].data = content[i]，HCodeTable[i].code = nullptr
4. 选择最小的两个节点 x 和 y：
    1. 初始化 x 和 y 为 -1
    2. 对于每个节点：
        1. 如果该节点的父节点为 -1 且权重小于 x 或 x 为 -1，则更新 x
        2. 如果该节点的父节点为 -1 且权重小于 y 或 y 为 -1 且不等于 x，则更新 y
5. 创建非叶子节点：
    1. 设置新节点的父节点为 i
    2. 更新权重为 x.weight + y.weight
    3. 设置左孩子和右孩子为 x 和 y
    4. 将 x 和 y 的父节点设置为 i
6. 重复步骤 4 和 5，直到只剩下一个根节点

3. CreateCodeTable：

CreateCodeTable()

// 输入：哈夫曼树 HTree，编码表 HCodeTable

// 输出：编码表 HCodeTable
1. 从根节点开始遍历哈夫曼树：
    1. 如果当前节点为叶子节点：
        1. 初始化空字符串 code = ""
        2. 递归地向上回溯到根节点，记录路径上的编码：
            1. 如果当前节点是其父节点的左孩子，则在 code 后面添加 '0'，否则添加 '1'
        3. 将生成的编码反转后存储到编码表中

4. Encode：

Encode(word, code)
// 输入：输入字符串 word，编码结果 code
// 输出：编码结果 code
1. 初始化空字符串 code = ""
2. 对于输入字符串中的每个字符：
    1. 在编码表中查找对应的编码，将其拼接到 code 中
3. 返回编码结果 code

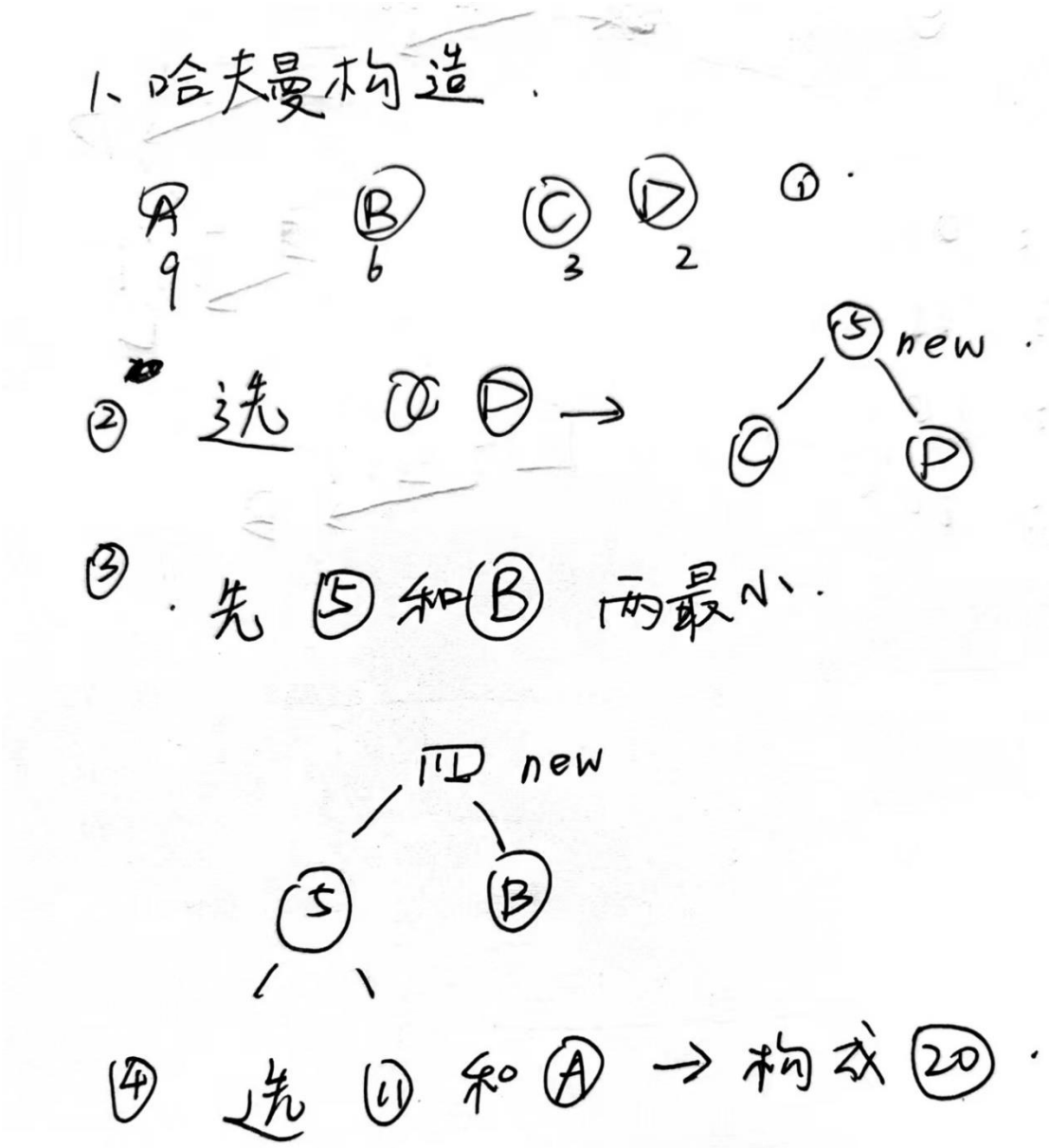
5. Decode：

Decode(code, word)
// 输入：哈夫曼编码 code，解码结果 word
// 输出：解码结果 word
1. 初始化当前节点为根节点
2. 对于每个编码位 code 中的字符：
    1. 如果当前节点有左孩子

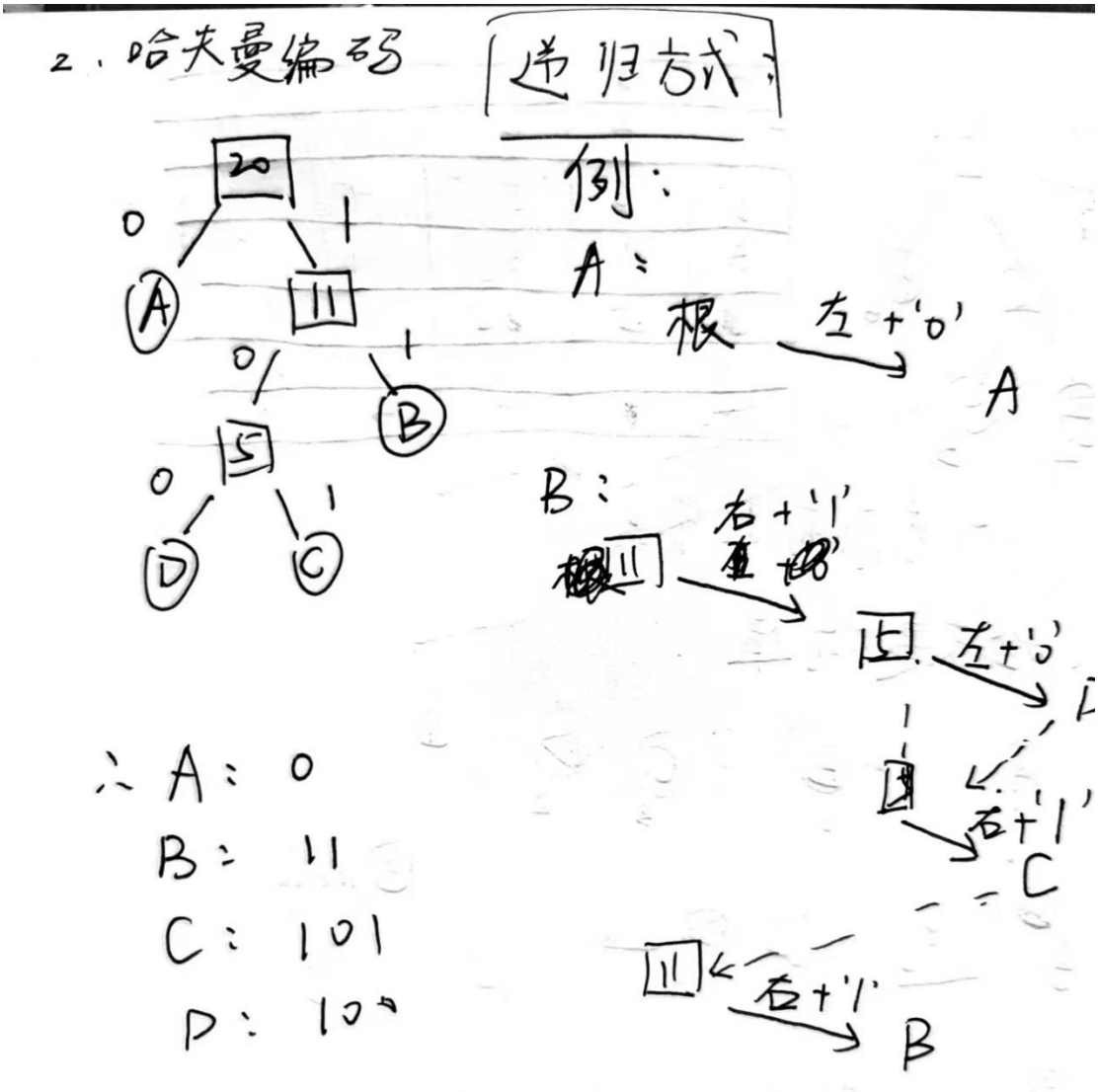
代码的时间复杂度：
- CountWeight：O(n)
- CreateHTree：O(n^2)
- CreateCodeTable：O(n)
- Encode：O(m)
- Decode：O(m)

其中，n 表示字符种类数，m 表示输入字符串的长度。
以下为手绘流程图：
1、 哈夫曼构造：

严禁抄袭！


1、哈夫曼构造.
A 9
B 6
C 3
D 2
①.
②选 C D →
5 new
C D
③.先 5 和 B 两最小.
11 new
5 B
④选 11 和 A → 构成 20.

2、 哈夫曼编解码：



2.3 其他

3. 程序运行结果

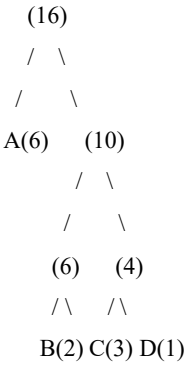
主函数流程：

1. 用户输入一个字符串，作为待编码的字符串。
2. 创建 Huffman 类的实例 tree。
3. 初始化数组 weight 和字符数组 content。
4. 调用 CountWeight 方法统计输入字符串中各个字符的出现频率，并将结果保存在 weight 数组中，同时将字符存储到 content 数组中。
5. 调用 GetCate 方法获取字符种类数。
6. 调用 CreateHTree 方法构建哈夫曼树。
7. 调用 CreateCodeTable 方法生成编码表。
8. 用户输入一个待编码的字符串 word。
9. 调用 Encode 方法对待编码的字符串进行编码。

10. 输出编码结果 code。
11. 调用 Decode 方法对编码结果进行解码。
12. 输出解码结果。

测试条件：
- 输入字符串：AAAAAABBBBCCCD
- 权重数组：{'A': 6, 'B': 4, 'C': 3, 'D': 1}
- 字符数组：['A', 'B', 'C', 'D']
- 待编码字符串：ABCD
- 编码结果：010111110
- 解码结果：ABCD

预测结果：



测试结论：

对输入字符串 "ABCD" 进行哈夫曼编码后，得到编码结果为 "010111110"。
对编码结果 "010111110" 进行解码后，得到解码结果为 "ABCD"。
因此，哈夫曼编码和解码的结果与预期一致。

4. 总结

1、首先哈夫曼这次运用的是一种静态链表的思想，是用数组下标来表示位置的，而且我的程序里设置的是哈夫曼树和数据节点的下标一一对应。

2、本次比较不好写的细节就是在编解码时，传入的字符串类型，我这里用的是指针，而且在拼接字符串时，要注意前后的数据类型，string 和 char 类型的转化和字符拼接的函数一定要清楚，有哪些可以用常字符串，有哪些不可以，需要分清楚。

3、还有本次调试了比较久的就是在解码过程中，最后一位总是解不到，这里我的代码中也

给出了一些叙述。

严禁抄袭！