组号: 12

图片包含 游戏机, 画

描述已自动生成

上海大学计算机工程与科学学院

**实 验 报 告**

（数据结构1）

学 期：2023-2024年冬季

组 长： 孔馨怡

学 号： 22122128

指导教师： 朱能军

成绩评定： （教师填写）

二〇二四年一月十六日

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **小组信息** | | | | |
| 登记序号 | 姓名 | 学号 | 贡献比 | 签名 |
| 1 | 孔馨怡 | 22122128 | 33.3% | 孔馨怡 |
| 2 | 高语涵 | 22120828 | 33.3% | 高语涵 |
| 3 | 杨利亚 | 22122418 | 33.3% | 杨利亚 |

|  |  |
| --- | --- |
| **实验概述** | |
| 实验零 | （熟悉上机环境、进度安排、评分制度；确定小组成员） |
| 实验一 | Chapter 3---双向链表的应用：发简历 |
| 实验二 | Chapter 4---队列/栈的应用：车厢调度问题 |
| 实验三 | Chapter 5---模式匹配的KMP算法：文学研究助手 |
| 实验四 | Chapter 6---二叉树拓展及标记二叉树 |

实验四

一、**实验题目**

（一）、左右子树交换

在二叉链表类模板中增加函数成员Revolute()，实现二叉树中所有结点的左右子树交换。

（二）、标记二叉树

1、问题描述

一颗二叉树，根结点标记为（1，1），规定：如果一个结点标记为（a，b），则它的左孩子（如果存在）标记为（a+b，b），它的右孩子（如果存在）标记为（a，a+b）。现在已知某个结点的标记为（a，b），求从根结点开始需要经过多少次左分支和多少次右分支才能达到结点（a，b）。

2、输入文件

输入文件第一行只有一个整数n，表示测试的数据组数。

接下来n行（第2～n+1）行，每行包括两个整数a和b。

3、输出文件

输出文件有n行，每行包括两个整数，分别表示从根结点开始到（a，b）需要经过的左分支数和右分支数。

4、输入样例

2

42 1

3 4

5、输出样例

41 0

2 1

二、**实验内容**

本次实验是实现左右子树交换和二叉树的标记，主要涉及二叉树的算法设计和实现。具体包括:二叉树的先序、中序、后序遍历的递归和非递归实现，二叉树的层次遍历，二叉树的复制、高度计算、结点个数计算、叶子结点个数计算和二叉树的翻转。

三、**解决方案**

1、算法设计（*主要描述数据结构、算法思想、主要操作、用例分析、改进方法等*）

（1）数据结构

本次实验主要数据结构是二叉树，它由节点构成，每个节点包括数据、左子树引用和右子树引用。这种数据结构自然地反映了树形结构的特点。

（2）算法思想与主要操作

**左右子树的交换：**

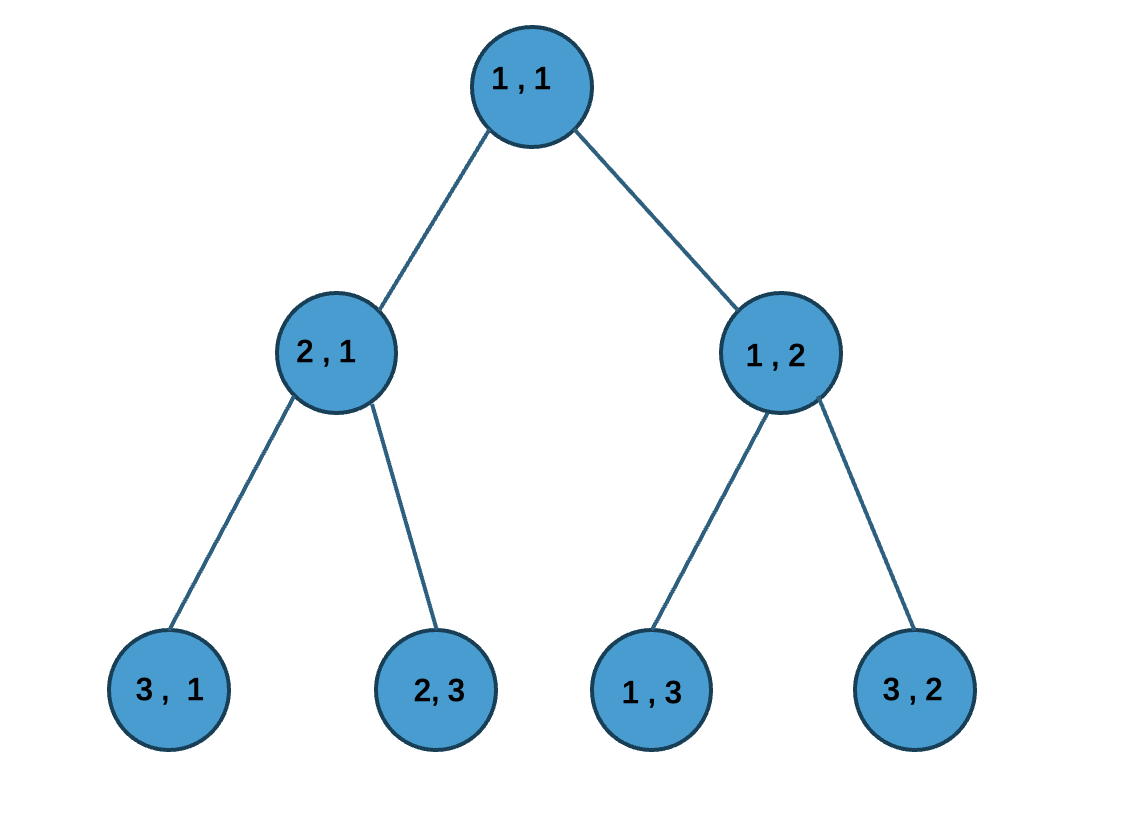
* 首先处理空指针的情况，如果输入的根节点‘r’为空，则直接返回空指针。
* 接着创建一个新的二叉树节点’newRoot’，其数据与原始节点’r’的数据相同。
* 之后采用调用递归函数的方式，对原始节点’r’的右子树进行左右子树交换，并把结果赋给新节点’newRoot’的左孩子。
* 对于’r’的左子树进行同样的操作。
* 最后返回新创建的节点’newRoot’。

**标记二叉树：**

先根据题意梳理一遍标记过程，如果一个结点标记为（a，b），则它的左孩子（如果存在）标记为（a+b，b），它的右孩子（如果存在）标记为（a，a+b）。那么从（1，1）根节点开始的树每一个标记的点的数值都是固定的，就如右图。

图示

描述已自动生成图片包含 图示

描述已自动生成在实现每一步标记的时候图片包含 户外, 钟表

描述已自动生成，都是将一边加到另一边，所以我们可以非常容易的根据只一个节点的标记数字，来判断它上一步实现的操作，近而可以还原至上一节点状态，以此类推，一直到根节点。代码思路梳理如下左图：(以2,3为例)

进而可以用循环来实现该操作，每一次的循环思路如上右图

2、源程序代码（*要求有必要注释、格式整齐、命名规范，利于阅读*）

（1）左右子树的交换

template <class **ElemType**>

**BinTreeNode**<**ElemType**>\* **BinaryTree**<**ElemType**>::**Revolute**(const **BinTreeNode**<**ElemType**>\* r) const

{

if (r == NULL)

return NULL;

**BinTreeNode**<**ElemType**>\* newRoot = new **BinTreeNode**<**ElemType**>(r->data);

newRoot->leftChild = **Revolute**(r->rightChild);

newRoot->rightChild = **Revolute**(r->leftChild);

return newRoot;

}

**BinTreeNode**<char>\* p;

char e;

e = 'A';

**BinaryTree**<char> **bt**(e); *// 建立二叉树*

e = 'B';

p = bt.**GetRoot**();

bt.**InsertLeftChild**(p, e); *// 插入左孩子*

e = 'C';

bt.**InsertRightChild**(p, e); *// 插入右孩子*

p = bt.**LeftChild**(p);

e = 'D';

bt.**InsertLeftChild**(p, e); *// 插入右孩子*

p = bt.**GetRoot**();

p = bt.**RightChild**(p);

e = 'E';

bt.**InsertLeftChild**(p, e); *// 插入左孩子*

e = 'F';

bt.**InsertRightChild**(p, e); *// 插入右孩子*

*// 显示原始二叉树*

cout << "原树:" << endl;

**DisplayBTWithTreeShape**(bt);

**BinTreeNode**<char>\* newRoot = bt.**Revolute**(bt.**GetRoot**());*// 调用Revolute函数进行左右子树交换*

**BinaryTree**<char> **newTree**(newRoot);*// 构造一个新的二叉树对象，以新的根结点为根*

cout << "\n交换后的树:" << endl;*// 显示交换左右子树后的二叉树*

**DisplayBTWithTreeShape**(newTree);

（2）标记二叉树

int n, a, b, i;

cout << "输入计算行数："<<endl;

cin >> n;

int left[100], right[100];*//构建数组储存每一组计算结果*

for (i = 0; i < n; i++)

left[i] = 0, right[i] = 0;*//初始化数组*

cout << "请输入数据：" << endl;

for (i = 0; i < n; i++) {

cin >> a >> b;

while (1) {

if (a > b) {

a = a - b;

left[i]++;*//如果a>b，说明该结点为左子树*

}

if (a < b) {

b = b - a;

right[i]++; *//如果a<b，说明该结点为右子树*

}

if (a == b) break; }

}

cout << "结果为：" << endl;

for (i = 0; i < n; i++) {

cout << left[i] << " " << right[i] << endl;

}

文本

描述已自动生成图形用户界面, 文本

描述已自动生成图形用户界面, 文本, 应用程序

描述已自动生成3、实验结果（*展示实验结果、测试情况、结果分析等*）

**4、算法分析（*对算法空间、时间效率进行必要分析，可能的改进建议等*）**

（1）空间复杂度

①构造函数和析构函数

构造函数 BinaryTree() 和析构函数 ~BinaryTree() 中没有使用额外的辅助空间，因此它们的空间复杂度都是 O(1)。

②遍历算法

遍历算法中通常使用递归或栈来辅助实现。这可能占用 O(h) 的栈空间，其中 h 为二叉树的高度。因此，遍历算法的空间复杂度为 O(h)。

③复制二叉树

复制构造函数 BinaryTree(const BinaryTree<ElemType> &t) 和赋值运算符重载 operator= 中的递归调用可能占用 O(h) 的栈空间。此外，递归深度与二叉树的高度 h 相关，因此复制二叉树的空间复杂度为 O(h)。

综合来看，该二叉树类的主要空间复杂度来源于递归调用和栈空间。在极端情况下，当二叉树为一个链式结构时，高度 h 可以达到 n（节点数），此时空间复杂度可能达到 O(n)。在平衡二叉树的情况下，高度 h 可以近似为 log(n)，因此空间复杂度为 O(log(n))。

（2）时间复杂度

①构造函数和析构函数：

构造函数 BinaryTree() 和析构函数 ~BinaryTree() 中的操作主要是一些基本的初始化和资源释放，其时间复杂度都是 O(1)。

②遍历算法：

遍历算法的时间复杂度取决于访问每个节点的次数。如果使用递归方式实现，对于每个节点，都需要递归访问其左右子树。因此，遍历算法的时间复杂度为 O(n)，其中 n 是二叉树的节点数量。

③复制二叉树：

复制构造函数 BinaryTree(const BinaryTree<ElemType> &t) 和赋值运算符重载 operator= 中的操作涉及到复制二叉树的每个节点。由于每个节点只复制一次，时间复杂度为 O(n)，其中 n 是二叉树的节点数量。

④获取树的高度：

获取树的高度的操作通常需要遍历整个树，访问每个节点一次。因此，获取树的高度的时间复杂度也是 O(n)。

5. 查找节点：

如果考虑二叉搜索树的特性，查找节点的平均时间复杂度为 O(log(n))，其中 n 是二叉树的节点数量。在最坏情况下，如果二叉搜索树不平衡，时间复杂度可能退化为 O(n)。

综合来看，该二叉树类的主要操作的时间复杂度主要集中在遍历和复制操作上，都是 O(n) 级别的。在涉及树的高度的操作上也是 O(n) 级别。需要注意，如果使用平衡二叉树的数据结构，查找节点的平均时间复杂度可以优化为 O(log(n))。

**5、总结与心得（*主要描述实验过程中存在的问题、原因、解决方法、收获、对实验内容的其他应用思考等*）**

可能改进的建议：

关于左右子树的交换功能，本实验代码采用在二叉树模板中添加Revolute函数，可以考虑一下几点进行改进：

①成员变量私有化：将BinTreeNode<ElemType> \*root声明为private，通过公共的GetRoot()函数进行访问。这有助于封装和保护数据。

②采用递归或迭代方式实现Revolute:如果Revolute函数是递归的，确保添加适当的基准情况，避免无限递归。如果使用迭代方式，可以考虑使用循环或者栈来实现。

③错误处理:考虑在Revolute函数中添加错误处理机制，处理可能的异常情况，例如空指针等。

以下是简单的改进代码:

template <class **ElemType**>

class **BinaryTree**

{

private:

**BinTreeNode**<**ElemType**> \*root;

*// 其他私有成员和函数*

public:

*// 公共接口和其他函数*

void **SwapLeftAndRight**(**BinTreeNode**<**ElemType**> \*node);

*// 交换结点的左右子树*

*// 其他函数的实现*

};

template <class **ElemType**>

void **BinaryTree**<**ElemType**>::**SwapLeftAndRight**(**BinTreeNode**<**ElemType**> \*node)

{

if (node == nullptr)

{

*// 处理空指针的情况*

return;

}

*// 交换左右子树*

BinTreeNode<ElemType> \*temp = node->leftChild;

node->leftChild = node->rightChild;

node->rightChild = temp;

*// 递归处理左右子树*

if (node->leftChild != nullptr)

{

**SwapLeftAndRight**(node->leftChild);

}

if (node->rightChild != nullptr)

{

**SwapLeftAndRight**(node->rightChild);

}

}