**Nankai University**

南 开 大 学

计算机学院

数据结构实验四

**实验四**实验报告

冯思程2112213

年级：2021级

专业：计算机科学与技术

指导教师：王玮

目录

**一、实验题目**

[（一）作业要求](#bookmark51" \o "Current Document)

[（二）问题描述](#bookmark60" \o "Current Document)

**[二、实验方案](#bookmark79" \o "Current Document)**

[（一）实验工具](#bookmark84" \o "Current Document)

[（二） 实验结果分析](#bookmark106" \o "Current Document)

[三](#bookmark147" \o "Current Document)**[、算法设计及分析](#bookmark147" \o "Current Document)**

[i代码展示与分析](#bookmark147" \o "Current Document)

[四](#bookmark313" \o "Current Document)**[、复杂度分析](#bookmark313" \o "Current Document)**

1. **[时间复杂度分析](#bookmark313" \o "Current Document)**
2. **[空间复杂度分析](#bookmark313" \o "Current Document)**

[五](#bookmark338" \o "Current Document)**[、实验总结](#bookmark338" \o "Current Document)**

—' 实验题目

（一） 作业要求

给定一个非空二叉树，关键字值不重复，使用链式存储完成下面操作：

1. 判断给定二叉树是否为完全二叉树

2. 给定节点值，返回该节点的高度

3. 若不是完全二叉树，将该树转为完全二叉树并输出。

需要自己定义二叉树结构，并根据输入构建二叉树进行上述操作，输入为层次遍历的顺序，若该位置无节点则为#。

（二） 问题描述

主要算法逻辑部分：

1. 首先构建结点结构，每个结点有data和它指向的左右孩子指针组成。
2. 按层序遍历构建二叉树，这里要利用队列，首先将跟结点入队，然后开始循环存数据，如果碰到了回车符号代表已经存完全部的数据了，跳出循环。然后分别对左孩子和右孩子存入数据，如果碰到#号则代表置空，其他情况正常存入，对于每个双亲，左右孩子分配完则双亲出列。循环直到都遍历。
3. 判断是不是完全二叉树，层次遍历查找空结点，如果空结点的下一个不是空结点则说明不是完全二叉树。还是利用队列，先跟结点入队（跟结点不可能是空）然后执行：

0.访问节点的左孩子和右孩子

1.左右孩子都没有 出队 判断队伍中的节点是否都没有孩子，如果都没有 是完全二叉树，如果有的右孩子 不是完全二叉树

2.如果少左孩子但是有右孩子 一定不是完全二叉树

3.如果有左孩子少右孩子，同1

4.如果左右孩子都有 出队并且把左右孩子入队。

（4）层次遍历找到一个结点的深度，新设置一个标识结点，标识每一行的结束，同时定义一个level，如果找到目标结点，则直接返回level值，找到了标志节点 说明一行结束 标志出列，如果啥都不剩了 结束循环，如果还有下一行 那就深度加一 再把标志放进去，其他普通节点 把孩子入列之后双亲出列。

（5）转换完全二叉树：新建一个char类的数组存储所有数据，并将索引与数据对应，再重新层序遍历新建一个二叉树存储。首先获得了按照层次遍历顺序存储所有元素的数组和数组下标的范围0-index-1。然后再层序遍历新建完全二叉树，每次循环分配左孩子和右孩子，要有一个判断有没有元素分给右孩子的if语句，因为最后一个可能不是成对的，没有右孩子。

（6）层序遍历输出二叉树，类似前文的逻辑，通过标识结点一行一行的输出，每行结束换行。

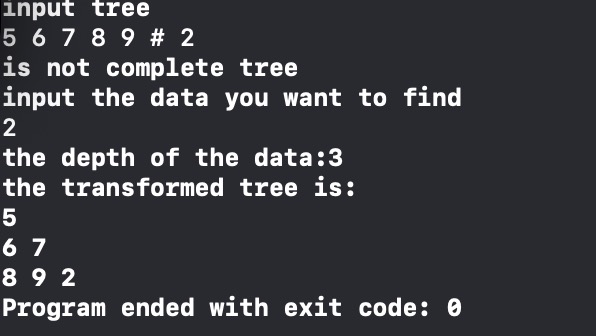
二、 实验方案

（一）实验工具

1.由于本人的电脑是mac下的macos系统，在本次实验用采用在xcode编译器下使用c++语言完成本次实验。

（二）实验结果分析步骤

1.在编写完程序后，输入多组数据进行输出测试，发现可以成功实现所需的功能。如下图：



1. 算法设计及分析

代码实现和分析：

#include<iostream>

#include<queue>

#include<cmath>

#include<cstring>

**using** **namespace** std;

**typedef** **char** ElemType;

**typedef** **struct** BiNode//定义了结点结构

{

    ElemType data;

**struct** BiNode\* lchild;

**struct** BiNode\* rchild;

    BiNode()//初始化，data为0，指针为空

    {

        data=0;

        lchild=**nullptr**;

        rchild=**nullptr**;

    }

}BiNode, \* BiTree;

**void** CreateBiTreeLevel(BiTree &T)//层序遍历创建二叉树

{

**char** rootdata;

    cin >>rootdata;

    T->data = rootdata;

    queue<BiNode\*>Q;//创建队列

    Q.push(T);//跟结点入列

**while** (!Q.empty())

    {

**char** lchilddata;

**if** (cin.get() == '\n')//遇到回车结束循环

        {

**break**;

        }

        cin >> lchilddata;

**if** (lchilddata == '#')//遇到#号指针置空

        {

            Q.front()->lchild =**nullptr**;

        }

**Else//其他情况，将数据存入，在把左孩子入队**

        {

            Q.front()->lchild = **new** BiNode;

            Q.front()->lchild->data = lchilddata;

            Q.push(Q.front()->lchild);

        }

**char** rchilddata;//对于右孩子的处理同理上文左孩子

**if** (cin.get() == '\n')

        {

**break**;

        }

        cin >> rchilddata;

**if** (rchilddata == '#')

        {

            Q.front()->rchild =**nullptr**;

        }

**else**

        {

            Q.front()->rchild = **new** BiNode;

            Q.front()->rchild->data = rchilddata;

            Q.push(Q.front()->rchild);

        }

        Q.pop();

    }

}

**bool** IsCompleteBiTree(BiTree& T)//判断是不是完全二叉树，分别对应算法时候的情况来进行处理。

{

    queue<BiNode\*>Q;

    Q.push(T);//跟结点入队

**while** (!Q.empty())

    {

**if** (Q.front()->lchild ==**nullptr** && Q.front()->rchild != **nullptr**)

        {

**return** 0;

        }

**else** {

**if** (Q.front()->lchild !=**nullptr** && Q.front()->rchild != **nullptr**)

        {

            Q.push(Q.front()->lchild);

            Q.push(Q.front()->rchild);

            Q.pop();

        }

**else**

        {

**if** (Q.front()->lchild !=**nullptr**)

            {

                Q.push(Q.front()->lchild);

            }

            Q.pop();

**while** (!Q.empty())

            {

**if** (Q.front()->lchild !=**nullptr** || Q.front()->rchild != **nullptr**)

                {

**return** 0;

                }

**else**{}

                Q.pop();

            }

**return** 1;

            }

        }

    }

**return** 0;

}

**int** GetLevel(BiTree& T, **char** x)//返回目标结点的深度

{

    queue<BiNode\*>Q;

    BiNode\* flag = **new** BiNode;//创建新的标识结点

    flag->data = '^';

    Q.push(T);

    Q.push(flag);//跟结点和标识结点入队

**int** level = 1;//深度初始值是1

**while** (!Q.empty())

{

//找到了标志节点 说明一行结束 标志出列

//如果啥都不剩了 结束循环

//如果还有下一行 那就深度加一 再把标志放进去

**if** (Q.front()->data == x){

**return** level;

        }

**else** **if** (Q.front()->data == '^'){

                Q.pop();

**if** (Q.empty()){

**break**;

            }

**else**{

                level++;

                Q.push(flag);

            }

        }

**else**{//其他普通结点，把孩子入队后双亲出队

**if** (Q.front()->lchild != **nullptr**)

            {

                Q.push(Q.front()->lchild);

            }

**if** (Q.front()->rchild != **nullptr**)

            {

                Q.push(Q.front()->rchild);

            }

            Q.pop();

        }

    }

**return** 1;

}

BiTree& Transform(BiTree& T)//转化为完全二叉树

{

**char** arr[100];

    queue<BiNode\*>Q;

    Q.push(T);

**int** index = 0;

**while** (!Q.empty())

    {

        arr[index] = Q.front()->data;

        index++;

**if** (Q.front()->lchild !=**nullptr**)

        {

            Q.push(Q.front()->lchild);

        }

**if** (Q.front()->rchild !=**nullptr**)

        {

            Q.push(Q.front()->rchild);

        }

        Q.pop();

    }//获得了按照层次遍历顺序存储所有元素的数组和数组下标的范围0-index-1

    BiTree TransformationTree = **new** BiNode;//创建新二叉树

    TransformationTree->data = arr[0];

    queue<BiNode\*>QQ;

    QQ.push(TransformationTree);

**for** (**int** i = 1; i <= index - 1 && !QQ.empty(); i = i + 2)

    {

        QQ.front()->lchild = **new** BiNode;

        QQ.front()->lchild->data = arr[i];

        QQ.push(QQ.front()->lchild);

**if** (i >= index - 1)

        {

            QQ.front()->rchild = **nullptr**;

        }

**else**

        {

            QQ.front()->rchild = **new** BiNode;

            QQ.front()->rchild->data = arr[i + 1];

        }

        QQ.push(QQ.front()->rchild);

        QQ.pop();

    }

**return** TransformationTree;

}

**void** LevelOrderByQueue(BiTree& T)//层次遍历输出二叉树

{

    queue<BiNode\*>Q;

    BiNode\* flag = **new** BiNode;//定义标识结点

    flag->data = '^';

    Q.push(T);

    Q.push(flag);

**while** (!Q.empty())

{

//访问队头元素并输出

//如果是标志节点 出列 换行 入列

//其他情况 如果有左孩子和有孩子

//把左孩子和右孩子入队

//把原来的双亲出队

**if** (Q.front()->data == '^')

        {

            Q.pop();

            cout << endl;

**if** (!Q.empty())

            {

                Q.push(flag);

            }

        }

**else**

        {

            cout << Q.front()->data << " ";

**if** (Q.front()->lchild != **NULL**)

            {

                Q.push(Q.front()->lchild);

            }

**if** (Q.front()->rchild != **NULL**)

            {

                Q.push(Q.front()->rchild);

            }

            Q.pop();

        }

    }

}

**int** main()

{

    BiTree MyTree = **new** BiNode;

    cout << "input tree " << endl;

    CreateBiTreeLevel(MyTree);

**if** (IsCompleteBiTree(MyTree))

    {

        cout<< "is complete tree" <<endl;

    }

**else**

    {

        cout<< "is not complete tree " <<endl;

    }

    cout << "input the data you want to find" << endl;

**char** inputdata;

    cin >> inputdata;

    cout << "the depth of the data:" << GetLevel(MyTree, inputdata) << endl;

**if** (!IsCompleteBiTree(MyTree))

    {

        cout << "the transformed tree is:" << endl;

        MyTree = Transform(MyTree);

        LevelOrderByQueue(MyTree);

    }

**return** 0;

}

1. 复杂度分析

首先来说一下时间复杂度和空间复杂度的表示方法：算法的执行效率，粗略地讲，就是算法代码的执行时间。我们假设每行代码执行的时间都一样，都是1个单位时间，从而算出一段代码总的执行时间为多少个单位时间，然后将公式中的低阶、常量、系数这三个不左右增长趋势的部分忽略，只记录最大量级的表示法。例：T(n)=2n+2就可以记为T(n)=O(n)，T(n)=n2+2就可以记为T(n)=O(n2)。这就是大O时间复杂度表示法，大O时间复杂度实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，也叫做渐进时间复杂度，简称时间复杂度。

1. 时间复杂度分析：

首先分析创建二叉树函数，只有一个while循环来遍历，中间没有出现递归和循环，时间复杂度是O（n），然后是对于判断是不是完全二叉树函数，while遍历循环中有一个while循环，没有递归，但是在两者是重叠的，即是一个分情况分开的一重循环，时间复杂度也是O（n），然后是获得深度函数，只有一个while遍历循环，时间复杂度是O（n），然后是转化完全二叉树函数，分为两个部分，前一部分是用数组获取数据和索引，只有一个while循环，后一部分是创建新二叉树，只有一个while循环，综上这个函数时间复杂度也是O（n）。

然后是层序遍历输出二叉树，只有一个while循环，时间复杂度也是O（n）。主函数中分别调用了上述函数，没有出现别的循环或者递归，综上，程序的时间复杂度是O（n）

1. 空间复杂度分析：

首先分析创建二叉树函数，只创建n个结点和一个队列，都是一次线性的空间，空间复杂度是O（n），然后是对于判断是不是完全二叉树函数，创建了队列，空间复杂度也是O（n），然后是获得深度函数，没有二重创建空间，都是一次线性，空间复杂度是O（n），然后是转化完全二叉树函数，分为两个部分，前一部分是用数组获取数据和索引，创建空间也是一次线性的，后一部分是创建新二叉树，也是一样的，综上这个函数空间复杂度也是O（n）。

然后是层序遍历输出二叉树，空间复杂度也是O（n）。主函数中分别调用了上述函数，没有多创建新的多重空间，综上，程序的时间复杂度是O（n）。

五、实验总结和程序改进

1.首先是程序流畅度的思考，在上次实验中，发现我在面对一些特殊情况时候，程序无法作出及时的反应，后来在助教学长的提醒下，进行了程序的优化。在这次实验中，我有意的关注了程序应对特殊情况的能力。对于本次程序由于已经声明过输入的二叉树非空，而且没有相同元素，节省了不少对特殊情况的应对处理。

2.因为这个程序涉及了很多对于指针的操作，很容易出错，我在不断的debug过程中，提高了自己的代码调试能力。