实验四实验报告

2112514 辛浩然

实验内容

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#。

实验代码

```
#include <iostream>
using namespace std;
class OutOfBounds
{
public:
    OutOfBounds() {}
};
template <class T>
class QNode
{
public:
    T data;
    QNode<T> *next;
};
template <class T>
class Queue
{
    QNode<T> *front;
    QNode<T> *rear;

public:
    Queue()
    {
        front = rear = nullptr;
    }
    ~Queue()
    {
        QNode<T> *n;
        while (front)
        {
            n = front->next;
            delete front;
            front = n;
        }
    }
    bool empty()
    {
```

```
        return (!front);
    }
    void push(T x)
    {
        QNode<T> *s = new QNode<T>();
        s->data = x;
        s->next = nullptr;
        if (front)
        {
            rear->next = s;
        }
        else
        {
            front = s;
        }
        rear = s;
    }
    T poll()
    //从队列头部获取元素
    {
        if (empty())
            throw OutOfBounds();
        T tmp = front->data;
        QNode<T> *p = front;
        front = front->next;
        delete p;
        return tmp;
    }
    void pop()
    {
        if (empty())
            throw OutOfBounds();
        QNode<T> *p = front;
        front = front->next;
        delete p;
    }
    T getFront()
    {
        if (empty())
            throw OutOfBounds();
        T tmp = front->data;
        return tmp;
    }
    T popRear()
    //从队列尾部获取元素
    {
        if (empty())
            throw OutOfBounds();
        T tmp = rear->data;
        if (rear == front)
        {
            return poll();
        }
        QNode<T> *p = front;
        while (p->next != rear && p->next)
        {
            p = p->next;
        }
```

```
        p->next = nullptr;
        rear = p;
        return tmp;
    }
};
class BinaryTree;
class BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
    friend class BinaryTree;

public:
    BinaryTreeNode()
    {
        data = 0;
        left = nullptr;
        right = nullptr;
    }
    BinaryTreeNode(int x)
    {
        data = x;
        left = nullptr;
        right = nullptr;
    }
    BinaryTreeNode(int x, BinaryTreeNode *l, BinaryTreeNode *r)
    {
        data = x;
        left = l;
        right = r;
    }
};

class BinaryTree
{
private:
    BinaryTreeNode *root;
    void Remove(BinaryTreeNode *p);
    void Add(BinaryTreeNode *p, BinaryTreeNode *q);

public:
    BinaryTree() { root = new BinaryTreeNode; }
    void CreateBinaryTree(int *num, bool *b, int len);
    bool isComplete();
    int getHeight(int x);
    void OutPut();
    void toComplete();
};

void BinaryTree::CreateBinaryTree(int *num, bool *b, int len)
{
    if (b[0])
        return;
    root->data = num[0];
    Queue<BinaryTreeNode *> s;
    s.push(root);
    int i = 1;
    while (i < len && (!s.empty()))
```

```
    //层序创建
    {
        BinaryTreeNode *head = s.poll();
        if (!head)
        {
            s.push(nullptr);
            s.push(nullptr);
            i += 2;
            continue;
        }
        if (b[i]) //空节点
        {
            head->left = nullptr;
        }
        else //非空节点
        {
            head->left = new BinaryTreeNode;
            head->left->data = num[i];
        }
        s.push(head->left);
        if (++i >= len)
            break;
        if (b[i])
        {
            head->right = nullptr;
        }
        else
        {
            head->right = new BinaryTreeNode;
            head->right->data = num[i];
        }
        s.push(head->right);
        if (++i >= len)
            break;
    }
}
bool BinaryTree::isComplete() //由上到下，由左到右层序遍历，如果前面有过叶子节点，那么后面遍历到的就不能有孩子
{
    if (!root)
    {
        return true;
    }
    Queue<BinaryTreeNode *> s;
    s.push(root);
    bool preLeaf = 0; //未遇到过叶子节点
    while (!s.empty())
    {
        BinaryTreeNode *head = s.poll();
        if (head->left)
        {
            if (preLeaf)
            {
                return false;
            }
            s.push(head->left);
        }
        else
```

```
        {
            preLeaf = 1; //遇到了 叶子节点，那后面遍历到的就不能有孩子了
        }
        if (head->right)
        {
            if (preLeaf)
            {
                return false;
            }
            s.push(head->right);
        }
        else
        {
            preLeaf = 1;
        }
    }
    return true;
}
int BinaryTree::getHeight(int x)
{
    int height = 1, curNum = 1, nextNum = 0;
    // curnum是本层节点数，nextnum是下层节点数
    if (root)
    {
        Queue<BinaryTreeNode *> s;
        s.push(root);
        while (!s.empty())
        {
            BinaryTreeNode *head = s.poll();
            curNum--;
            if (head->data == x)
                break;
            //遍历到目标时，目标上面的层肯定都遍历完了
            if (head->left)
            {
                s.push(head->left);
                nextNum++;
            }
            if (head->right)
            {
                s.push(head->right);
                nextNum++;
            }
            if (!curNum)
            { //遍历完一层
                height++;
                curNum = nextNum;
                nextNum = 0;
                //变下层
            }
        }
    }
    return height;
}
void BinaryTree::OutPut()
{
    int curNum = 1, nextNum = 0;
    if (root)
```

```
    {
        Queue<BinaryTreeNode *> s;
        s.push(root);
        while (!s.empty())
        {
            BinaryTreeNode *head = s.poll();
            cout << head->data << ' ';
            curNum--;
            if (head->left)
            {
                s.push(head->left);
                nextNum++;
            }
            if (head->right)
            {
                s.push(head->right);
                nextNum++;
            }
            if (!curNum)
            { //本层的所有节点输出完毕
                cout << endl;
                curNum = nextNum;
                nextNum = 0;
            }
        }
    }
}
void BinaryTree::toComplete()
{
    Queue<BinaryTreeNode *> s;
    Queue<BinaryTreeNode *> tree;
    //先把所有节点存到队列tree中
    s.push(root);
    while (!s.empty())
    {
        BinaryTreeNode *head = s.poll();
        tree.push(head);
        if (head->left)
        {
            s.push(head->left);
        }
        if (head->right)
        {
            s.push(head->right);
        }
    }
    //接下来，按层遍历二叉树，遇到空节点，tree队列从尾部出一个元素，把这个节点从树中取下来，加在该位置。直到tree队列空。
    //这样二叉树前面的空缺被后面的节点补全。
    s.push(root);
    tree.poll();
    while (!tree.empty())
    {
        BinaryTreeNode *head = s.poll();
        if (!head->left) //如果该节点无左孩子
        {
            BinaryTreeNode *toDelete = tree.popRear();
            Remove(toDelete);
```

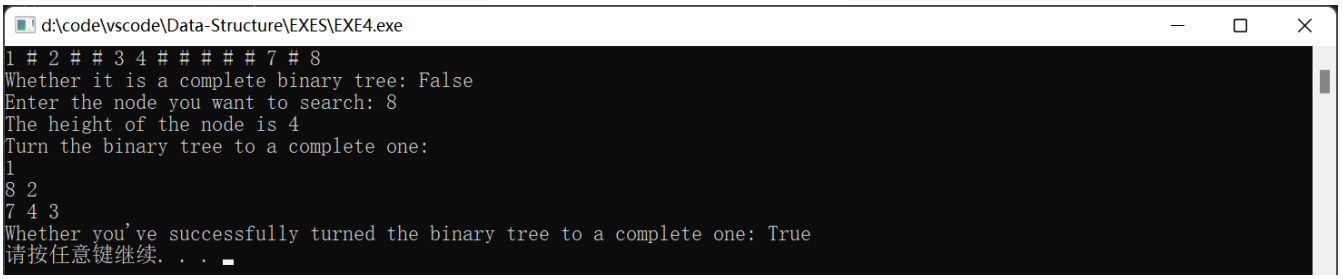
```
            Add(head, toDelete);
            //把最后的节点取下来，补过来
        }
        else
        {
            tree.poll();
        }
        s.push(head->left);
        if (tree.empty())
            break;
        if (!head->right)
        {
            BinaryTreeNode *toDelete = tree.popRear();
            Remove(toDelete);
            Add(head, toDelete);
        }
        else
        {
            tree.poll();
        }
        s.push(head->right);
    }
}
void BinaryTree::Remove(BinaryTreeNode *p)
{ //在转换的条件下，删除的节点一定没孩子
    Queue<BinaryTreeNode *> s;
    s.push(root);
    //找到删除节点的上一个节点
    BinaryTreeNode *head = new BinaryTreeNode;
    while (!s.empty())
    {
        head = s.poll();
        if (head->left == p || head->right == p)
        {
            break;
        }
        if (head->left)
        {
            s.push(head->left);
        }
        if (head->right)
        {
            s.push(head->right);
        }
    }
    if (head->left == p)
    {
        head->left = nullptr;
    }
    else
    {
        head->right = nullptr;
    }
}
void BinaryTree::Add(BinaryTreeNode *p, BinaryTreeNode *q)
{
    if (!p->left)
    {
```

```
            p->left = q;
        }
        else
        {
            p->right = q;
        }
    }
    int main()
    {
        try
        {
            BinaryTree a;
            int num[100];
            bool symbol[100] = {0};
            int i = 0;
            char k;
            while (1)
            {
                if (!(cin >> num[i]))
                {
                    cin.clear();
                    cin >> k;
                    symbol[i] = 1;
                }
                i++;
                if (cin.get() == '\n')
                    break;
            }
            a.CreateBinaryTree(num, symbol, i);
            cout << "Whether it is a complete binary tree: ";
            cout << (a.isComplete() ? "True" : "False") << endl;
            cout << "Enter the node you want to search: ";
            int x;
            cin >> x;
            cout << "The height of the node is ";
            cout << a.getHeight(x) << endl;
            if (!a.isComplete())
            {
                cout << "Turn the binary tree to a complete one: " << endl;
                a.toComplete();
                a.OutPut();
                cout << "Whether you've successfully turned the binary tree to a complete one: ";
                cout << (a.isComplete() ? "True" : "False") << endl;
            }
        }
        catch (OutOfBounds)
        {
            cerr << "Queue is empty" << endl;
        }
        system("pause");
    }
```

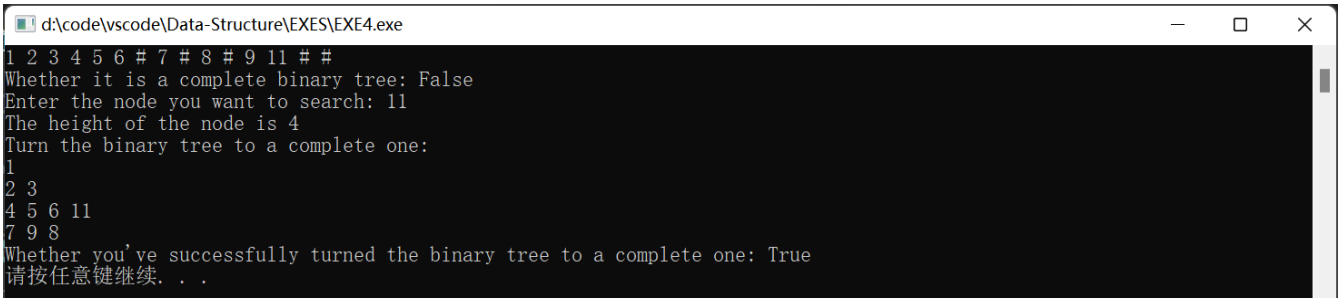
实验测试

测试用例一

测试用例二



测试用例三



思路分析

层序创建二叉树

- 获取输入的元素：
  - 定义数组储存输入的二叉树节点数值。
  - 另外定义`bool`型数组储存每个节点是否为空，如果输入`#`时，对应的`bool`型数组里的元素置`1`.
- 层序创建二叉树：
  - 首先创建根节点。
    - 如果第一个输入的为`#`，那么根节点为空，二叉树为空；
    - 如果不是，创建根节点，其`data`值为数组的第一个元素。创建一个队列，将根节点压入队列。
  - 如果队列不为空，且数组索引小于数组长度，从队列拿出一个节点。
    - 如果出的这个节点是空节点，那么它的左右孩子肯定都为空。将它的左右孩子压入队列。数组索引加2。跳过本次循环。（空节点也入队列是为了保证数组索引的正确性）
    - 在数组中读取下一个数据。
      - 如果`bool`型数组对应元素值为`1`，即输入的为`#`，队列中弹出的节点的左孩子为空；
      - 如果输入的不是`#`，为队列中弹出的节点的左孩子分配空间，将其`data`值置为读取的数据。
      - 将节点的左孩子（无论是否是空节点）入队列。
    - 判断数组索引是否小于数组长度，如果不是，说明所有数据读完，结束循环。
    - 在数组中读取下一个数据。
      - 如果`bool`型数组对应元素值为`1`，即输入的为`#`，队列中弹出的节点的右孩子为空；
      - 如果输入的不是`#`，为队列中弹出的节点的右孩子分配空间，将其`data`值置为读取的数据。
      - 将节点的右孩子（无论是否是空节点）入队列。
  - 循环上步操作，直到队列为空或数组索引不小于数组长度。

完全二叉树判断

- 基本思想：完全二叉树，从上到下，从左到右遍历的时候，在遍历到第一个没有某个孩子的节点前，所有的遍历到的节点肯定都有孩子；在此之后，所有的节点肯定都没有孩子。
- 实现过程：
  - 如果根节点为空，肯定是完全二叉树；
  - 如果根节点非空，创建一个队列，将根节点压入队列。并定义一个`bool`型变量`preLeaf`，初始化为`0`，其表示有无遇到过只有一个孩子或无孩子的节点。
    - 如果队列不为空，从队列拿出一个节点。
      - 如果该节点的左孩子为空，`preLeaf`置`1`；
      - 如果该节点的左孩子不为空
        - 如果`preLeaf`为`0`，说明之前所有的节点都有2个孩子，目前为止满足完全二叉树的条件。
        - 如果`preLeaf`为`1`，说明之前有个节点至少无一个孩子，那如果是完全二叉树的话该节点就不应该有孩子，说明不是完全二叉树，返回`false`；
      - 如果该节点的右孩子为空，`preLeaf`置`1`；
      - 如果该节点的右孩子不为空
        - 如果`preLeaf`为`0`，说明之前所有的节点都有2个孩子，该节点也有左孩子，目前为止满足完全二叉树的条件。
        - 如果`preLeaf`为`1`，说明之前有个节点至少无一个孩子，那如果是完全二叉树的话本节点就不应该有孩子，说明不是完全二叉树，返回`false`；
    - 循环上步操作，直到队列空。如果最终队列空，说明遍历完二叉树都未出现异常节点，则该二叉树为完全二叉树，返回`true`。

得到节点高度

- 基本思想：从上到下，从左到右遍历二叉树，直到遇到该节点为止。此时，该节点上面的层都已经完全遍历完。节点高度就是遍历完的层数加1.
- 实现过程：
  - 如果根节点非空，创建一个队列，将根节点压入队列。定义两个变量，分别表示本层节点数和下层节点数。定义高度，初始化为`1`.
  - 如果队列不为空，从队列拿出一个节点，本层节点数减一。
    - 如果该节点的数值等于目标数，则已经遇到该节点，结束循环。
    - 如果该节点有左孩子，将左孩子放入队列，下层节点数加一。
    - 如果该节点有右孩子，将有孩子放入队列，下层节点数加一。
    - 如果本层节点数为0，说明本层节点遍历完，下层变本层。高度加一，下层节点数赋值给本层节点数，下层节点数清零。
  - 循环上步操作，直到队列空或者找到目标节点。返回高度值。

不完全二叉树转化为完全二叉树

- 基本思想：层序遍历二叉树，如果遇到空节点，从二叉树最后取下节点接在该位置。
- 实现过程：
  - 首先保存二叉树所有节点入队列。使用两个队列，队列`s`和队列`t`，将根节点压入队列`s`。
    - 如果队列`s`不为空，从队列中拿出一个节点，把这个节点保存到队列`t`中。
      - 如果该节点有左孩子，把左孩子压入队列`s`。
      - 如果该节点有右孩子，把右孩子压入队列`s`。
    - 循环上步操作，直到队列`s`为空。
  - 经过上述操作，队列`s`为空，队列`t`保存二叉树所有节点。

- 将根节点存至队列`s`，从队列`t`中拿出一个节点。
- 如果队列`s`不为空，如果队列`t`不为空，从队列`s`中拿出一个节点。
  - 如果该节点没有左孩子，从队列t的**尾部**出一个节点（在队列的结构中自己写了从尾部取元素的函数，变成双向队列），把这个节点从原有的地方取下来，插入到该位置（具体实现下面再详细写）。把插入的左孩子压入队列`s`。
  - 如果这个节点有左孩子，把左孩子压入队列`s`，从队列`t`头部出一个节点。
  - 如果队列`t`为空，结束循环。
  - 如果该节点没有右孩子，从队列`t`的**尾部**出一个节点，把这个节点从原有的地方取下来，插入到该位置。把插入的右孩子压入队列`s`。
  - 如果这个节点有右孩子，把右孩子压入队列`s`，从队列`t`头部出一个节点。
- 循环上步操作，直至队列`t`为空。因为在上述操作中，每次遇到非空节点，队列`t`就从头部出一个节点，每次遇到空节点，队列`t`就从尾部出一个节点。这样保证的非空节点的数目不变。
- 在二叉树中取下某节点：首先通过层序遍历找到该节点父母。在本程序情况下，倒着取节点，取下的节点肯定没孩子。那么直接将该节点父母的对应指向改为空就可以了。
- 在二叉树中插入某节点：通过层序遍历找到要插入的位置的父母。将父母的左/右指向改为插入的节点。

输出二叉树

- 如果根节点非空，创建一个队列，将根节点放入队列。定义两个变量，分别表示本层节点数和下层节点数。
  - 如果队列不为空
    - 从队列拿出一个节点，输出该节点的数值。本层节点数减一。
    - 如果该节点有左孩子，将左孩子放入队列，下层节点数加一。
    - 如果该节点有右孩子，将右孩子放入队列，下层节点数加一。
    - 如果本层节点数为0，说明本层节点遍历完，下层变本层。输出换行。下层节点数赋值给本层节点数，下层节点数清零。
  - 循环上步操作，直到队列空。

复杂度

时间复杂度O(n)，空间复杂度O(n)

心得体会

1. 通过本次实验熟练了二叉树的操作，尤其是层序遍历，对二叉树的理解更加深刻；
2. 在实验过程中，遇到了一些问题：
   - 在创建二叉树时，由于设定的int型数组，无法读取到输入的`#`，是通过查找相关资料，发现字母导致输入状态异常，需要使用 `cin.clear();` 重置`cin`状态，还需要把缓存区内的`#`清除。
   - 如何将不完全二叉树转化为完全二叉树，一开始并没有清晰的思路。后来在经过反复思考后找到思路。在实现的过程中，一开始对于取下节点的`remove`操作，没有意识到本实验中的特殊情况，即取下的节点都没孩子，而是分情况进行，很是复杂。
   - 等等.....
3. 对于实验的操作，为降低时间复杂度，都采用了非递归的方式实现。