20190408 左士海-作业 7

6. 试编写算法求二叉树中双分支节点的个数。

```cpp
template<class T>
int BiTree<T>::doubleBranchCount(BiNode<T> *p) {
    if (p == nullptr) {
        return 0;
    }
    int left = doubleBranchCount(p->leftChild);
    int right = doubleBranchCount(p->rightChild);
    return left + right + ( (p->leftChild && p->rightChild) ? 1 : 0);
}
```

方法一：类似于求高度。若当前节点为双分支节点，则该二叉树的双分支节点数为左子树 + 右子树双分支节点数 + 1。

```cpp
template<class T>
void BiTree<T>::doubleBranchCount(BiNode<T> *p, int &cnt) {
    if (p) {
        if (p->leftChild && p->rightChild) {
            cnt++;
        }
        doubleBranchCount(p->leftChild, cnt);
        doubleBranchCount(p->rightChild, cnt);
    }
}
```

方法二：通过传递引用参数的形式，在每次递归调用，通过判断当前是否双分支节点来对参数实时更新。

7. 试编写算法求二叉树中各个结点的平衡因子（左右子树高度之差）

```cpp
template<class T>
int BiTree<T>::balanceFactor(BiNode<T> *p) {
    if (p == nullptr) {
        return 0;
    }
    int left = balanceFactor(p->leftChild);
    int right = balanceFactor(p->rightChild);
    cout << p->data << ": " << abs(left - right) << endl;
    return max(left, right) + 1;
}
```

参考递归计算高度，当前节点的平衡因子等于左右子树高度差的绝对值，这里直接对每个节点的平衡因子进行输出，也可以增加一个引用参数来对数据进行保存。

8. 一棵二叉树以二叉链表来表示，求其指定的某一层 k(k>1)上的叶子结点的个数。

```cpp
template<class T>
void BiTree<T>::leafCountOfLevel(BiNode<T> *p, int k, int level, int
&cnt) {
    if (p == nullptr) {
        return;
    }
    if(level == k) {
        if (!p->leftChild && !p->rightChild)
            cnt++;
        return; // 已经到达第 k 层了 没有必要再深入下去
    }
    leafCountOfLevel(p->leftChild, k, level + 1, cnt);
    leafCountOfLevel(p->rightChild, k, level + 1, cnt);
}
```

递归版。原理类似于求双分支节点数目方法二，通过参数记录当前状态。Level 为当前递归的层数，cnt 为引用类型，记录叶子节点个数。

9. 试编写算法输出一棵二叉树中根结点到各个叶子结点的路径。

```cpp
template<class T>
void BiTree<T>::pathToLeaf(BiNode<T> *p, string path) {
    if (p == nullptr) return;
    if (p->leftChild == nullptr && p->rightChild == nullptr) {
 // 是叶节点  输出路径 并返回
        cout << path << endl;
        return;
    }
    path += " -> ";      // 说明当前节点至少有一个孩子  "->" 指向下一个节点
    // 因为节点可能只有一个孩子 所以要先判断某个孩子存不存在
    if (p->leftChild) {      // 如果左孩子存在 进入左子树
        path += p->leftChild->data;
        pathToLeaf(p->leftChild, path);
        path.pop_back();         // 回溯  恢复到没有走这条路径的情况

    }
    if (p->rightChild) {     // 如果右孩子存在 进入右子树
        path += p->rightChild->data;
        pathToLeaf(p->rightChild, path);
        path.pop_back();      // 回溯
    }
}
```

深度优先搜索的思路，依次输出根节点到每个叶节点的路径。这里用单个 string 类型变量来储存路径并打印。

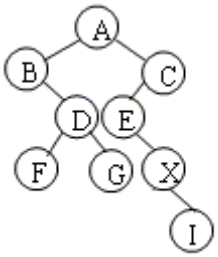
10. 设计一个算法，求二叉树中两个给定结点的最近公共祖先。

```cpp
template<class T>
BiNode<T> *BiTree<T>::lowestCommonAncestor(BiNode<T> *p, BiNode<T> *x,
BiNode<T> *y) {
    if (p == nullptr) {
        return nullptr;
    }
    if (p == x || p == y) {
        return p;
    }
    BiNode<T> *left = lowestCommonAncestor(p->leftChild, x, y);     //
在左子树中查找
    BiNode<T> *right = lowestCommonAncestor(p->rightChild, x, y);  // 在
右子树中查找
    if(left != nullptr && right != nullptr){
        return p;    // x, y 节点 分别在两个子树中 返回当前的根节点
    }
    if(left == nullptr && right == nullptr){
        return nullptr; // x, y 均不在两个子树中 返回 null
    }
    return left == nullptr ? right : left;  // 在一边有一边没有 到有的一边
}
```

对于两个节点，其最小公共祖先，存在三种情况：1. 不存在；2. 其中一个节点是另外一个节点的祖先；3. 两个节点分别位于不同的子树中。

补充作业（选做题）：

11. 若一棵二叉树中没有数据域值相同的结点，试设计算法打印二叉树中数据域值为 x 的结点的所有祖先结点的数据域。如果根结点的数据域值为 x 或不存在数据域值为 x 的结点，则什么也不打印。例如对下图所示的二叉树，则打印结点序列为 A、C、E。




```cpp
template<class T>
string BiTree<T>::allAncestorOfX(T x) {
    string res, path;
    if (root) {
        path += root->data;
    } else {
        return "";
    }
    allAncestorOfX(root, x, path, res);
    return res;
}
```

```cpp
template<class T>
void BiTree<T>::allAncestorOfX(BiNode<T> *p, T x, string path,
string &res) {
    if (p == nullptr) return;
    if (p->data == x) {
        path.pop_back();    // 定义自己不是自己的祖先
        res = path;     // 把结果赋给引用传递出去
        return; // 无需深入
    }
    path += " ";
    if (p->leftChild) {     // 左子树存在进入左子树
        path += p->leftChild->data;
        allAncestorOfX(p->leftChild, x, path, res);
        path.pop_back();    // 回溯
    }
    if (p->rightChild) {    // 右子树存在进入右子树
        path += p->rightChild->data;
        allAncestorOfX(p->rightChild, x, path, res);
        path.pop_back();    // 回溯
    }
}
```

思路：深度优先搜索，该题类似于输出根节点到每个叶子的路径。由于该题主需要输出一条结果，所以定义一个引用来接收，再通过接口函数返回出去。

12. 已知二叉树存于二叉链表中，试编写一个算法，判断给定二叉树是否为完全二叉树。

```cpp
template<class T>
bool BiTree<T>::isCompleteBiTree_2() {
    if (root == nullptr) return true;       // 空树也是完全二叉树
    queue<BiNode<T> *> queue;
    bool flag = false;  // 状态变量 记录是否遇到为 nullptr 的节点
    queue.push(root);
    while (!queue.empty()) {
        BiNode<T> * p = queue.front();
        if (p != nullptr) {
            if (flag) {
                return false;
            }
            // 无论有没有左右孩子都入队 为判断状态做准备
            queue.push(p->leftChild);
            queue.push(p->rightChild);
        } else {
            flag = true;
        }
        queue.pop();
    }
    return true;
}
```

思路：对于完全二叉树，倒数第二层往上一定是满二叉树，对于最后一层，可满可不满。若一节点存在，则其层序遍历上的前驱一定存在，也就是说，若当前节点不存在，往后再出现节点存在，就不可能是完全二叉树。

13. 已知二叉树存于二叉链表中，编写一个递归算法，利用叶结点中空的右链指针域 rchild，将所有叶结点自左至右链接成一个单链表，算法返回最左叶结点的地址（链头）

```cpp
template<typename T>
void BiTree<T>::getLeafTable(BiNode<T> *p, BiNode<T> *&head, BiNode<T> *&rear)
{
    if (p == nullptr) return;
    if (!p->leftChild && !p->rightChild) {  // 如果是叶节点
        if (rear) { // 叶子链表不为空
            rear->rightChild = p;
            rear = p;
            rear->rightChild = nullptr;
        } else {    // 叶子链表为空 定义头结点
            head = rear = p;
        }
    }
    getLeafTable(p->leftChild, head, rear); // 从左到右 左孩子优先
    getLeafTable(p->rightChild, head, rear);
}
// 接口函数
template<typename T>
BiNode<T> * BiTree<T>::getLeafTable() {
    BiNode<T> *head = nullptr, *temp = nullptr;
    getLeafTable(root, head, temp);
    return head;
}
```

思路：深度优先搜索，以实现叶节点从左到右的顺序，使用返回 head 指针引用，为叶子链表的头结点。rear 指针始终指向链表尾结点以完成新叶子节点的插入。

14. 已知二叉树存于二叉链表中，试编写一个算法计算二叉树的宽度，即同一层中结点数的最大值。

```cpp
template<class T>
int BiTree<T>::width(BiNode<T> *p) {
    if (p == nullptr) return 0;
    queue<BiNode<T> *> queue;
    queue.push(p);
    int width = 1;
    int len;    // 记录每一层的个数
    while (!queue.empty()) {
        len = queue.size();
        while (len) {
            BiNode<T> * t = queue.front();
            queue.pop();
            // 下一层入队
            if (t->leftChild) queue.push(t->leftChild);
            if (t->rightChild) queue.push(t->rightChild);
            len--;
        }
        width = max(width, (int)queue.size());
    }
    return width;
}
```

思路：层序遍历，一层一层地遍历，完成一层后，对宽度 width 进行更新。

课堂补充题：

由中序序列和后序序列构造二叉树：

```cpp
template<class T>
BiNode<T> *BiTree<T>::createByMidPost(vector<T> &mid, vector<T> &post, int imid,
int ipost, int n) {
    if (0 == n) return nullptr;
    auto *p = new BiNode<T>;
    p->data = post[ipost];   // 后序序列的根节点在最后一位
    int i;
    for (i = 0; i < n; ++i) {   // // 在中序序列中定位根节点
        if (post[ipost] == mid[imid + i]) break;
    }
    p->leftChild = createByMidPost(mid, post, imid, ipost - n + i, i);
    p->rightChild = createByMidPost(mid, post, imid + i + 1, ipost - 1, n - i - 1);
    return p;
}
```