实验题第 1 题

第 1 小题

(1) 设计思路

本题要求根据二叉树的先序遍历序列建立二叉树。

- 根据测试程序给出的形式(tree.CreateBinaryTree())可知，需要设计一个内部函数作为辅助函数，再用一个接口函数去调用，从而完成二叉树的构建。
- 由题，如果出现 #，则表明子树为空树。分两种情况讨论：
  - 1. 先序序列第一个元素为 #
    此时，整棵树为空树，r（根节点）赋值为 NULL。
  - 2. 先序序列第一个元素不为 #
    则沿根节点左孩子一直往下创建子节点，直到第一次读取到 #，表示触底，此时终止创建左孩子节点，读取下一位数据赋给右孩子，并以之为新的根节点，重新开始此过程。
- 总结规律可知，整个过程类似于先序遍历，大致步骤有三如下：
  - （递归的基本操作）先判断是否为空树，
    - 如果空，则向后一位继续判断，并返回；
    - 如果不空，则创建新节点并赋值，然后向后一位继续判断。
  - 递归构建左子树；
  - 递归构建右子树；

  根据以上步骤，基本可以形成思路，构建出如下代码。

(2) 源代码

void CreateBinaryTree(BTNode<DataType> * &r, DataType pre[], int &preStart){};
//内部私有辅助函数

```
//根据二叉树的有占位符的先序遍历序列创建 n 个节点的二叉树
template <class DataType>
void BinaryTree<DataType>::CreateBinaryTree(BTNode<DataType> * &r, DataType pre[],
int &preStart){
  if(pre[preStart] == '#')
  {                                          //如果先序遍历序列第一位为 #，则是空（子）树
    r = NULL;                                //设置树为空树
    preStart++;                              //向后一位扫描
    return;                                  //调用返回指令，结束函数
  }
  else
  {
    r = new BTNode<DataType>(pre[preStart]);  //创建新节点，赋值为先序序列第 prestart 位
    preStart++;                              //向后一位扫描
  }

  CreateBinaryTree(r->lChild, pre, preStart); //递归构建左子树
  CreateBinaryTree(r->rChild, pre, preStart); //递归构建右子树
}
```

void CreateBinaryTree(){}; **//接口函数**

```
template <class DataType>
void BinaryTree<DataType>::CreateBinaryTree() {
    string pre;                                          //用字符串表示先序序列
    cin >> pre;                                          //输入先序序列

    int preStart = 0;                                    //初始化先序序列的起始位置
    char *preArray = new char[pre.length() + 1];         //创建一个字符数组用于读取先序序列
    strcpy(preArray, pre.c_str());                       //将字符串内容复制到字符数组中

    CreateBinaryTree(root, preArray, preStart);          //调用内部函数构建二叉树

    delete preArray;                                     //使用完数组后，释放内存
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

利用这种方法构建二叉树，只需要带占位符的先序遍历即可，优点是直观、逻辑简单，且二叉树越“满”越节省空间。缺点是当二叉树比较稀疏时，会浪费很大的占用空间，空间利用率上不如经典的先序+中序序列创建法。

因此，在已知二叉树为满二叉树、完全二叉树，或接近以上两者时，可以采用这种方法来创建左右孩子链式存储结构的二叉树。

第 2 小题

(1) 设计思路

本题要求计算二叉树的最大宽度。

- 由对宽度的定义（二叉树的最大宽度是指二叉树所有层中节点个数的最大值），可知需要对二叉树进行层次性的操作，即分别获取每一层的节点数，作为本层的宽度，取其中最大为二叉树的宽度。

  因此，可以自然地联想到二叉树的层次遍历，类比地进行层次性操作。
- 但层次遍历存在无法区分每个层次的问题，只能整体性地遍历二叉树所有节点。于是需要寻找方法，在层次遍历的基础上去进行“分层遍历”。一可行方案如下：
  - 仍使用`!q.IsEmpty()`的判断条件去保证遍历完二叉树所有节点；
  - 在 while 循环中嵌套一个 for 循环来进行当前层的遍历，保证每一轮 while 循环只遍历一层的节点。
  - 在 while 循环中进行获取、交换、迭代操作，在 for 循环中进行出、入队操作。

根据以上方案，构造出代码如下。

(2) 源代码

int Width(BTNode<DataType> *r){};　　//求二叉树最大宽度

```
template <class DataType>
int BinaryTree<DataType>::Width(BTNode<DataType> *r)
{
    if (!root) return 0;                              //如果根节点为空，返回 0
    LinkQueue<BTNode<DataType>*> q;                   //创建链队列 q 来辅助获取每一层宽度
    q.EnQueue(root);                                  //根节点入队
    int maxWidth = 0;                                 //初始化 maxWidth 为 0

    while (!q.IsEmpty()) {                            //仿照层次遍历判断 q 非空
        int width = q.GetLength();                    //利用 Getlength 获取当前层的宽度
        if (width > maxWidth) maxWidth = width;       //如果当前层宽度更大，则更新 maxWidth
        BTNode<DataType> *p;                          //创建临时节点 p 来存放上一层的最左节点
        // 处理下一层的节点
        for (int i = 0; i < width; ++i){              //仿照层次遍历进行出入队
            q.DelQueue(p);                            //队头节点出队
            if (p->lChild) q.EnQueue(p->lChild);      //左孩子存在则入队
            if (p->rChild) q.EnQueue(p->rChild);      //右孩子存在则出队
        }
    }
  return maxWidth;
}
```

int Width(){}; **//求二叉树最大宽度(接口)**

```
//接口函数
template <class DataType>
int BinaryTree<DataType>::Width() {
        return Width(root);
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

需要注意的是，根据题目要求，本操作仍需要一个接口函数配合一个辅助函数来实现。

可以得到一个普适性的方法：

如果需要在 while 循环中根据数据特征（数据特征随机，且可以通过具体数据得到）进行有规律的操作，可嵌套一个 for 循环来实现分步性的循环。

第 3 小题

(1) 设计思路

本题要求编写递归函数求二叉树的节点数目。

- 要求二叉树的节点数目，只要求左子树的数目和又子树的数目并加和即可。因此，需要构建一个程序，递归调用加和函数。
- 从先、中、后序遍历中可以总结出规律：
  只要在函数中两次调用自身，变量分别为左、右孩子（左、右子树），就可以通过递归，遍历到以函数参数为根的二叉树的所有节点。
- 根据以上规律，可以直接在 return 返回的的内容中递归调用自身，通过递归返回来实现对所有节点的计数和加和。由此得到代码如下。

(2) 源代码

int getNodeCount(BTNode<DataType> *r);　//递归求二叉树节点数

```
template <class DataType>
int BinaryTree<DataType>::getNodeCount(BTNode<DataType> *r) {
  if (!r) return 0;            //如果节点为空，返回 0
  return                       //如果节点非空，继续递归返回
  1 + getNodeCount(r->lChild) + getNodeCount(r->rChild);
  //递归调用 getNodeCount，计算总节点数为 1 + 左子树节点数 + 右子树节点数
}
```

int NodeCount();　//递归求节点数目（接口）

```
template <class DataType>
int BinaryTree<DataType>::NodeCount(){
  return getNodeCount(root);
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

实验题第 2 题

本题要求树以孩子兄弟链表为存储结构，设计算法完成题目。
则先根据 BTNode.h 编写头文件如下。并对 BinaryTree 作相应修改，改写为 Tree.h。

TreeNode.h　　//节点头文件

```
#pragma once
template <class DataType>
class TreeNode {
public:
  DataType data;                              //数据域
  TreeNode<DataType> *Child;                  //左孩子指针域
  TreeNode<DataType> *Brother;                //右孩子指针域
  TreeNode() { Child = Brother = NULL; }  //无参构造函数，构造空节点
  //有参构造函数，构造数据元素节点
  TreeNode(DataType &e, TreeNode<DataType> *child = NULL, TreeNode<DataType> *brother
= NULL)
  {
    data = e;  Child = child;  Brother = brother;
  }
};
```

第 1 小题

(1) 设计思路

本题要求求树的深度。

- 树的深度即高度。
- 由于确定一个节点的高度，要比较各子树的高度，则需要先递归到叶节点并计算其高度，再逐层向上返回并累积。这种特性使得本题适合用后序遍历来解决。
- 需要注意的是，每一层的兄弟节点深度相同，子树的深度不一定相同，要从左向右对每个兄弟节点递归计算子树深度，取子树中深度最大的加上 1，即为以其父节点为根的树的深度。可以用 while 循环遍历同一层的兄弟节点，同时递归计算其子树深度，再取最大深度。

(2) 源代码

int Height(TreeNode<DataType> *r){};　　//求树的深度

```
template <class DataType>
int Tree<DataType>::Height(TreeNode<DataType> *r) {
    if (r == NULL) return 0;              //空树深度为 0

    int Height = 0;                       //初始化 Height 为 0
    TreeNode<DataType>* p = r->Child;     //指针 p 指向 r 的孩子节点

    while (p!= NULL) {                    //如果孩子/兄弟节点存在
        int pHeight = this->Height(p);    //递归计算孩子深度(用 this->Height 是为了防止与
BTNode.Height 引发歧义)
        p = p->Brother;                   //访问下一个兄弟节点
        if (pHeight > Height)
          Height = pHeight;               //如果孩子的深度大于 Height，更新 Height 为 pHeight
    }
```

```
    //如果 r 的孩子/兄弟节点为空，递归返回 1+Height
    return 1 + Height;
}
```

int Height(){};　　//求树的深度(接口)

```
template <class DataType>
int Tree<DataType>::Height() {          //接口
    return Height(root);
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

孩子-兄弟存储结构的树和一般的二叉树逻辑存在很大的不同，有时不能套用二叉树的操作逻辑和思维模式。 对于孩子-兄弟树而言，求深度的基本思路是从根节点出发，递归计算每个子节点的深度，沿兄弟指针逐步遍历同层节点，再返回最大深度，这符合后序遍历思维。

第 2 小题

(1) 设计思路

本题要求求树的度。

- 树的度即度数最大的节点的度。
- 计算树的度，可以从根节点的孩子节点开始，利用 while 循环横向遍历兄弟节点，逐次加 1，直到兄弟指针为空，终止循环。过程中递归计算以每一个兄弟节点为根的树的度数，取其中最大的暂作为 maxDegree，即树的度。循环结束后，得到根节点的度数 Dgree，再将 Dgree 和子树中最大的 maxDegree 作比，择其大者而返回之。

(2) 源代码

int Degree(TreeNode<DataType>* r){}; **//求树的度**

```
template <class DataType>
int Tree<DataType>::Degree(TreeNode<DataType>* r) {
    if (r == NULL) return 0;                    //空树度为 0

    int Degree = 0;                             //初始化 Degree(当前节点的度数)
    int maxDegree = 0;                          //初始化 maxDegree(最大度数，树的度)
    TreeNode<DataType>* p = r->Child;           //指针 p 指向 r 的孩子节点

    //递归计算当前节点的度（即第一个孩子节点及其兄弟节点的总数量）
    while (p != NULL) {                         //如果孩子/兄弟节点存在
        Degree++;                               //孩子/兄弟节点存在，度数+1
        int childDegree = this->Degree(p);      //递归计算子节点的度
        if (childDegree > maxDegree)
         maxDegree = childDegree;               //更新最大度
        p = p->Brother;                         //访问下一个兄弟节点
    }

    // 返回当前节点度和子节点度中的最大值
    if (Degree > maxDegree)                     //比较记录的节点度数目和 maxDegree，更大则替换
        maxDegree = Degree;
  return maxDegree;
}
```

int Degree(){}; **//求树的度(接口)**

```
template <class DataType>
int Tree<DataType>::Degree() {
  return Degree(root);
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

上面两题都用到了递归的思想，对抽象思维能力是个很大的挑战。总结规律，遇到这类问题，要从一般步骤开始考虑：

1. 明确问题的递归特性

判断子问题的解是否可以用于构建原问题的解，且解决方法一致或类似。如果可以，则符合递归特性。

2. 确定递归的判空单位操作

如 `if (r == NULL) return 0`，此类最基本的操作可以保证递归调用函数时，函数遇到这种清空会直接回溯到上一层递归。因为这种操作在逻辑上一般比较简单，先写下来可以帮助思考和理解接下来要处理的递归问题。

3. 定义递归关系

将问题转化为较小规模的子问题，并明确递归关系。递归关系应表达当前问题如何通过调用自身来解决更小规模的子问题。如在二叉树中计算高度的递归关系为：

$$Height(root) = 1 + \max(Height(left), Height(right))$$

4. 合成子问题的解

确定如何按照逻辑，用适合的操作将子问题的解组合为原问题的解。

5. 书写递归函数并进行验证

按照递归关系和组合方式，实现递归函数。检查是否覆盖了所有边界情况。