专题：根据遍历序列重建二叉树

1.由先序序列、中序序列得后序序列（无需建树）

```
void solve(int preL, int inL, int postL, int n)
{
    if (n <= 0)
        return;
    if (n == 1)
    {
        post[postL] = pre[preL];
        return;
    }

    int root = pre[preL];
    post[postL + n - 1] = root;

    int i;
    for (i = 0; i < n; i++)
        if (in[inL + i] == root)
            break;

    int numL = i;
    int numR = n - numL - 1;
    solve(preL + 1, inL, postL, numL);
    solve(preL + numL + 1, inL + numL + 1, postL + numL, numR);
}
```

在这个方法中，我们需要传入的参数有：先序、中序、后序序列的左端点，需要处理数组的规模n。

根据二叉树三种遍历的规律：先序序列的根结点出现在区间左端点preL，后序序列的根结点出现在区间右端点$postR=postL+n-1$。接下来，只需要根据中序序列去寻找根结点，即可划分左右子树。然后分别对左右子树进行递归即可。

2. 后序与中序序列重建这棵树

```
struct node {
    int data;
    node* lchild, *rchild;
    node() {
        this->lchild = NULL;
        this->rchild = NULL;
    }
};

node* create(int postL, int postR, int inL, int inR) {
    if(postL > postR) return NULL;
```

```
    node* root = new node;
    root -> data = post[postR];
    int k;
    for(k = inL; k <= inR; k++)
        if(in[k] == post[postR]) break;
    int numLeft = k - inL;
    root -> lchild = create(postL, postL + numLeft - 1, inL, k - 1);
    root -> rchild = create(postL + numLeft, postR - 1, k + 1, inR);
    return root;
}
```

传参跟1中有所不同，但本质是一样的，都是通过左端点和偏移量确定所要寻找的数组下标，只是这里换成了借助右端点。代码实现和递归的原理一模一样。

3. 先序与中序序列重建这棵树

```
node* create(int preL, int preR, int inL, int inR) {
    if(preL > preR) return NULL;
    node* root = new node;
    root -> data = pre[preL];
    int k;
    for (k = inL; k <= inR; k++)
        if (in[k] == pre[preL])
            break;
    int numLeft = k - inL;
    root -> lchild = create(preL + 1, preL + numLeft, inL, k - 1);
    root -> rchild = create(preL + 1 + numLeft, preR, k + 1, inR);
    return root;
}
```

与2的实现大同小异。

4. 先序与后序序列转中序（可能不唯一）

由于没有中序序列的支持，这时可能会出现不唯一情况：即无法确定某个结点是左孩子还是右孩子。

```
void getIn(int preL, int preR, int postL, int postR) {
    if(preL == preR) {
        in.push_back(pre[preL]);
        return;
    }

    int k;
    for(k = preL + 1; k <= preR; k++)
        if(pre[k] == post[postR - 1]) break;
    int numL = k - preL - 1;

    if(k == preL + 1) unique = false; //既是左子树根又是右子树根，矛盾，说明这棵树
无法唯一确定。
```

```
    else
        getIn(preL + 1, k - 1, postL, postL + numL - 1);
    in.push_back(post[postR]);
    getIn(k, preR, postL + numL, postR - 1);
}
```

5. 总结

- 时刻牢记三种遍历顺序：

  > 先序：根 -> 左子树 -> 右子树，中序：左子树 -> 根 -> 右子树，后序：左子树 -> 右子树 -> 根

- 先序（后序）序列用于确定根结点，然后拿着根结点去中序序列里找左右子树。
- 想不清楚时可以画画图，因为三种序列里，确定左右子树区间端点略有区别（其实也就差个+1，-1之类的）。