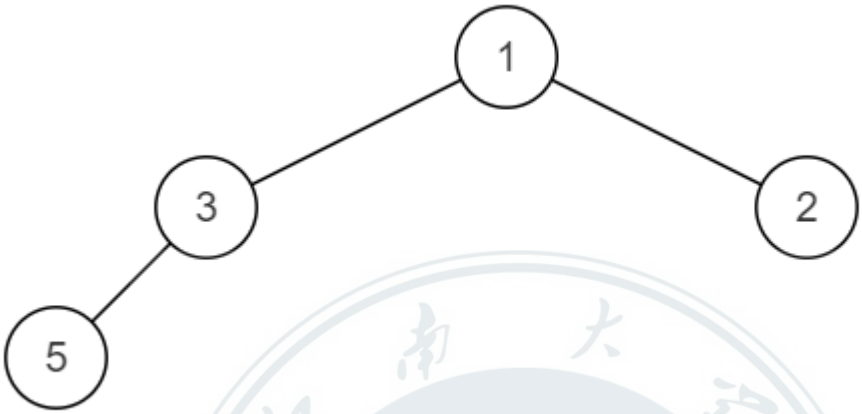
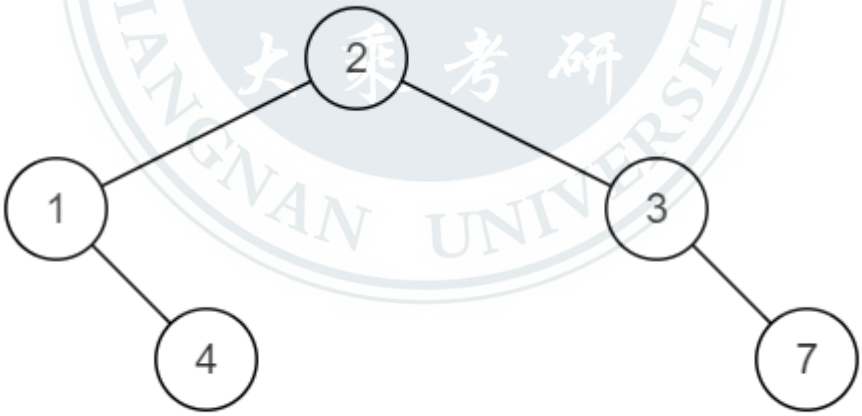
1. 已知两颗二叉树，将它们合并成一颗二叉树。合并规则是：都存在的结点，就将结点值加起来，否则空的位置就由另一个树的结点来代替。例如： 两颗二叉树是:

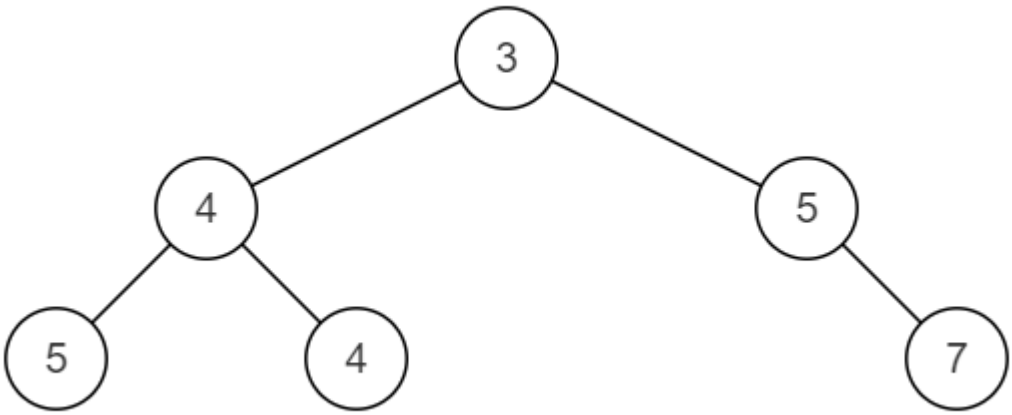
```
树1
```



树2



合并后:

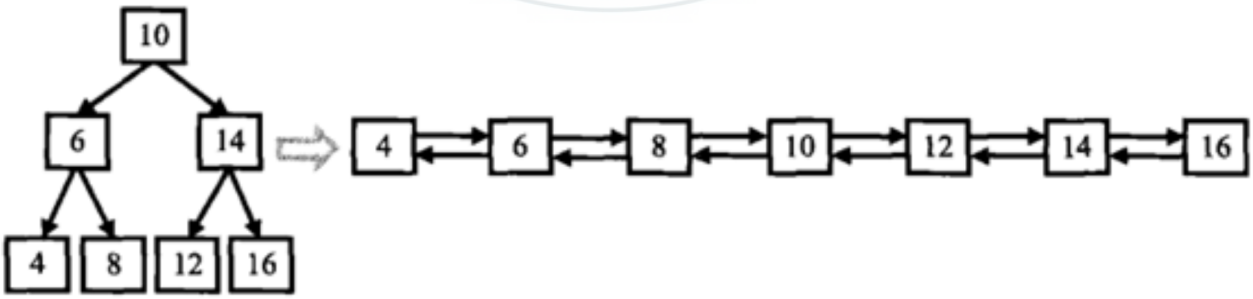


思路：要将一棵二叉树的节点与另一棵二叉树相加合并，肯定需要遍历两棵二叉树，那我们可以考虑同步遍历两棵二叉树，这样就可以将每次遍历到的值相加在一起。遍历的方式有多种，这里推荐前序递归遍历。

代码：

```
BTNode *mergeTrees(BTNode *t1, BTNode *t2)
{
    if (!t1)
        return t2;
    if (!t2)
        return t1;
    t1->data += t2->data;
    t1->lchild = mergeTrees(t1->lchild, t2->lchild);
    t1->rchild = mergeTrees(t1->rchild, t2->rchild);
    return t1;
}
```

2. 输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。如下图所示



要求：空间复杂度O(1)（即在原树上操作），时间复杂度 O(n)

注意：要求不能创建任何新的结点，只能调整树中结点指针的指向。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继, 返回链表中的第一个节点的指针

思路：二叉搜索树最左端的元素一定最小，最右端的元素一定最大，符合“左中右”的特性，因此二叉搜索树的中序遍历就是一个递增序列，我们只要对它中序遍历就可以组装称为递增双向链表。

代码：

```
BTNode *head, *pre;
void inorder(BTNode *cur)
{
    if (cur)
    {
        inorder(cur->lchild);
        if (pre == NULL)
            head = cur;
        else
        {
            pre->rchild = cur;
            cur->lchild = pre;
        }
        pre = cur;
        inorder(cur->rchild);
    }
}
BTNode *Convert(BTNode *pRootOfTree)
{
    if (!pRootOfTree)
        return NULL;
    pre = NULL;
    inorder(pRootOfTree);
    return head;
}
```

3. 假设一棵平衡二叉树的每个结点都表明了平衡因子b，试设计一个算法，求平衡二叉树的高度。

分析：因为二叉树各结点已标明了平衡因子b，故从根结点开始记树的层次。根结点的层次为1，每下一层，层次加1，直到层数最大的叶子结点，这就是平衡二叉树的高度。当结点的平衡因子b为0时，任选左右一分枝向下查找，若b不为0，则沿左（当b=1时）或右（当b=-1时）向下查找。

代码

```
int Height(BiTree t)
// 求平衡二叉树t的高度
{
    int level = 0;
    BTNode *p = t;
    while (p)
    {
        level++; // 树的高度增1
        if (p->bf < 0)
            p = p->rchild; // bf=-1 沿右分枝向下
                           // bf是平衡因子，是二叉树t结点的一个域，
                           //因篇幅所限，没有写出其存储定义
        else
            p = p->lchild; // bf>=0 沿左分枝向下
    }

    return level;
```

```
}
```

4. 已知二叉树T的结点形式为（lchild,data,count,rchild），在树中查找值为X的结点，若找到，则记数（count）加1，否则，作为一个新结点插入树中，插入后仍为二叉排序树，写出其非递归算法。

代码:

```
void BSTInsert2(BiTree &T, int x)
{
    BTNode *pre = NULL; // pre是p的父节点
    BTNode *p = T, *q;
    while (p)
    {
        if (p->data == x)
        {
            ++(p->count);
            return;
        }
        else if (p->data > x)
        {
            pre = p;
            p = p->lchild;
        }
        else
        {
            pre = p;
            p = p->rchild;
        }
    }
    q = (BTNode *)malloc(sizeof(BTNode));
    q->lchild = q->rchild = NULL;
    q->data = x;
    q->count = 1;
    if (pre->data > x)
        pre->lchild = q;
    else
        pre->rchild = q;
}
```