实验报告

一、数据结构设计

结点的数据结构设计如下，其中 val 表示该节点存储的值，size 表示以该节点为根节点的子树的节点数量（包括该节点本身），lchild 是指向左子树的指针，rchild 是指向右子树的指针。

```
typedef struct TreeNode{
    int val;
    int size;
    TreeNode *lchild;
    TreeNode *rchild;
}TreeNode, *BiTree;
```

二、代码说明

本实验在普通二叉搜索树（BST）的基础上进行，所以具有插入、删除等基本功能。基本功能添加了注释，由于不是本实验的重点因此不再详细说明。需要注意之处只有插入和删除结点时会对查找路径上的结点的 size 进行相应处理。代码详见“BST.cpp”

1.插入结点

```
int InsertNode(BiTree* BSTTree, int e){  //插入节点
    if(*BSTTree == NULL){//此树为空，则创建根节点
        TreeNode* tmpnode = (TreeNode*)malloc(sizeof(TreeNode));
        tmpnode->val = e;
        tmpnode->size = 1;
        tmpnode->lchild = NULL;
        tmpnode->rchild = NULL;
        *BSTTree = tmpnode;
        return 1;//插入成功
    }
    if((*BSTTree)->val == e){  //关键值相同，插入失败
        return 0;
    }else if((*BSTTree)->val > e){  //插入值小于根节点的值，插入左子树
        (*BSTTree)->size++;
        return InsertNode(&(*BSTTree)->lchild, e);
    }else if((*BSTTree)->val < e){  //否则，插入右子树
        (*BSTTree)->size++;
        return InsertNode(&(*BSTTree)->rchild, e);
    }
}
```

2.创建 BST

```
void CreateTree(BiTree* BSTTree, int n){  //创建BST,其中n为节点总数
    int tmp;
    for(int i = 0; i < n; i++){
        cin >> tmp;
        InsertNode(BSTTree, tmp);
    }

}
```

3.查找结点

该函数中，为了便于在 AuxBST_Find 中的使用，添加了一个参数 esize，以求递归查找过程中，除所求节点的左子树以外，小于所求节点值的结点总数

```
TreeNode* SearchNode(BiTree BSTTree, int e, int* esize){
//查找值为e的结点，esize返回除当前节点的左子树以外，小于当前节点值的结点总数
    if(BSTTree == NULL)
        return NULL;
    if(BSTTree->val == e){
        (*esize) += 1;
        return BSTTree;
    }else if(BSTTree->val > e){
        return SearchNode(BSTTree->lchild, e, esize);
    }else if(BSTTree->val < e){
        (*esize) += BSTTree->lchild->size+1;
        return SearchNode(BSTTree->rchild, e, esize);
    }
```

4. 删除节点

```
TreeNode* DeleteNode(BiTree* BSTTree, int e){  //删除节点
    if(*BSTTree == NULL)
        return NULL;
    else if((*BSTTree)->val > e){  //找左节点
        (*BSTTree)->size--;
        (*BSTTree)->lchild = DeleteNode(&(*BSTTree)->lchild, e);
    }else if((*BSTTree)->val < e){  //找右节点
        (*BSTTree)->size--;
        (*BSTTree)->rchild = DeleteNode(&(*BSTTree)->rchild, e);
    }else{  //找到了，分情况处理
        if((*BSTTree)->lchild == NULL){  //没有左节点
            TreeNode* rnode = (*BSTTree)->rchild;
            free(*BSTTree);
            return rnode;
        }else if((*BSTTree)->rchild == NULL){  //没有右节点
            TreeNode* lnode = (*BSTTree)->lchild;
            free(*BSTTree);
            return lnode;
        }else if((*BSTTree)->lchild->rchild == NULL){  //左节点没有右节点
            TreeNode* lnode = (*BSTTree)->lchild;
            lnode->rchild = (*BSTTree)->rchild;
            free(*BSTTree);
            return lnode;
        }else{  //找左子树的最大值来替换当前节点
            TreeNode* p = (*BSTTree)->lchild;
            for(p; p->rchild->rchild != NULL; p = p->rchild);
            TreeNode* lnode = p->lchild;
            p->rchild = lnode->rchild;
            lnode->lchild = (*BSTTree)->lchild;
            lnode->rchild = (*BSTTree)->rchild;
            free(p);
            return lnode;
        }
    }
    return *BSTTree;
}
```

5.查找第 k 小的数

由于为结点添加了 size 属性，所以在查找第 k 小的数时，就不需要遍历二叉搜索树了，

只需要根据当前节点及其左右子树的 size 值来判断第 k 小的数的位置，然后在对应的子树中继续查找即可。即如果左子树的结点数小于 k，那么第 k 个数就在右子树中，继续查找右子树即可（注意在查找右子树时，k 中需要去掉左子树的部分）；否则查找左子树。

```
int AuxBST_Select(BiTree BSTTree, int k){  //查找第k小的数
    if(BSTTree->size < k)  //总结点数小于k，查找失败
        return -1;
    int num = 0;
    if(BSTTree->lchild){
        num = BSTTree->lchild->size;  //num是该树中小于当前节点值的结点数
    }
    if(num == k - 1)  //找到了
        return BSTTree->val;
    else if(num < k - 1){  //需要在右子树中继续查找
        return AuxBST_Select(BSTTree->rchild, k - num - 1);
    }else if(num > k - 1){ //需要在左子树中继续查找
        return AuxBST_Select(BSTTree->lchild, k);
    }
}
```

6. 找比 z 大的第 k 个数

先用查找功能找到 z 值对应的结点，然后求出所有小于该节点的结点的总数，即找到 z 值的排序 rank(z)，然后调用 Aux_Select 函数找到第 rank(z)+k 个数，就是比 z 大的第 k 个数。

```
int AuxBST_Find(BiTree Tree, int z, int k){ //找比z大的第k个数
    int zsize = 0;
    TreeNode* tmp = SearchNode(Tree, z, &zsize);
    if(tmp == NULL){//查找错误，没有为z的值
        return -1;
    }else{
        if(tmp->lchild)
            return AuxBST_Select(Tree, zsize + tmp->lchild->size + k);
            //zsize加左子树结点数，就是所有值小于z所在结点值的结点总数
        else
            return AuxBST_Select(Tree, zsize + k);
    }
}
```

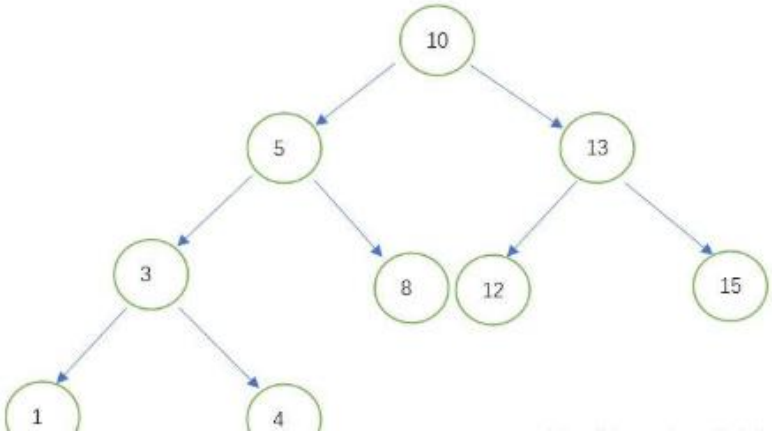
7.中序遍历 BST

比较特别的是输出格式都以“**结点存储值：结点 size 值**”的形式输出。

```
void ShowTree(BiTree BSTTree){  //中序遍历
    if(BSTTree){
        ShowTree(BSTTree->lchild);
        printf("%d:%d ", BSTTree->val, BSTTree->size);
        ShowTree(BSTTree->rchild);
    }
```

三、测试结果

按照 10 5 13 3 8 12 15 1 4 的顺序输入九个结点，构造如下 BST 树：



按照所给的要求，对应的输出为：

```
Please input the number of the nodes
9
Please input the nums in order
10 5 13 3 8 12 15 1 4
1:1 3:3 4:1 5:5 8:1 10:9 12:1 13:3 15:1
Size of the left child of root: 5
Size of the right child of root: 3
Now you input k to get the kth num in BST
5
The 5th num is:8
Now you input z&k to get the kth num greater than z in BST
8 3
The 3th num greater than 8 is:13
PS D:\exam\ics\schedlab\sched\cpp\src>
```

经检验（还使用了其他数据检验，在此不再赘述），所给的结果都与实际符合，证明程序正确无误。

四、复杂度分析

1. k-select 查找：

由于 AuxBST_Select 不需要遍历二叉搜索树，在树的每一层只需要访问一棵子树，所以对应的复杂度应该与 BST 上的查找相同，均为 O(logn)。而非强化的 BST 需要递归计数，直到找到第 k 个数为止，时间复杂度为 O(n)。因此，强化后的 BST 在时间复杂度上有所提高。

2. 找比 z 大的第 k 个数：

该功能分别调用 SearchNode 和 AuxBST_Select，而这两个函数的复杂度都是 O(logn)，所以 AuxBST_Find 的时间复杂度也是 O(logn)。而非强化的 BST 仍然要遍历找到 z 之后继续向后找 k 个，复杂度还是 O(n)。所以强化后的 BST 在时间复杂度上也有所提高。