动态查找表

表结构本身是在查找过程中动态生成的，即对于给定值key，若表中存在其关键字等于key的记录，则查找成功返回，否则插入关键字为key的记录

二叉排序树（Binary Sort Tree）

二叉排序树（二叉查找树），或者是一颗空树；

或者是具有下列性质的二叉树：

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值
- 它的左右子树也分别为二叉排序树

查找

```
/**
 * 功能：在二叉排序树中查找有指定关键字的结点；查找成功返回父结点
 * 参数：root 根结点；key 要查找的关键字；parent 双亲；returnTree 通过此参数返回指针
 * 返回值：查找成功返回TRUE，否则返回FALSE
*/
BinarySortTree SearchBST(BinarySortTree bst, KeyType key){
    //如果空则返回空
    if(bst == NULL){
        return NULL;
    //如果要查找的关键字和当前结点关键字相同，那么返回
    }else if(bst->data->key == key){
        return bst;
    //如果要查找的关键字大于结点关键字，那么继续查找右子树，否则继续查找左子树
    }else if(bst->data->key > key){
        return SearchBST(bst->leftChild, key);
    }else{
        return SearchBST(bst->rightChild, key);
    }
}
```

插入

二叉排序树是一种动态树表，树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入；新插入的结点一定是一个新添加的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或者右孩子结点；

```
/**
 * 功能：向二叉排序树中插入结点
 * 参数：bst 二叉排序树；value 要插入的值
 * 返回值：插入成功返回TRUE，否则返回FALSE
```

```
*/
Status insertBST(BinarySortTree &bst, ElementType value){
    BinarySortTree parentNode;
    //如果指定结点已经存在，那么插入失败
    if(searchBST(bst, value, NULL, parentNode)){
        return FALSE;
    }
    //具有指定值的结点不存在，构造一个新结点
    else{
        //申请一个新结点并且设置其数据域和指针域
        BinaryNode *node = (BinaryNode *)malloc(sizeof(BinaryNode));
        node->data = value;
        node->leftChild = node->rightChild = NULL;

        //如果具有指定值的结点的双亲也不存在，那么说明这个是一个空树
        if(parentNode == NULL){
            bst = node;
        }
        //根据二叉排序树的性质
        //左子树的值均小于根结点，右子树的值均大于根节点
        else if(value < parentNode->data){
            parentNode->leftChild = node;
        }else{
            parentNode->rightChild = node;
        }

        return TRUE;
    }
}
```

删除

```
/**
 * 功能：删除二叉排序树中的结点
 * 参数：tree 二叉树     左根右
 * 返回值：遍历结果
*/
Status deleteBST(BinarySortTree &bst, ElementType value){
    BinarySortTree parent, node;

    //查找该节点指针以及父结点指针
    Status res = searchBST(bst, value, NULL, parent);
    node = searchBST(bst, value);

    //情况1：结点为空代表找不到；删除失败
    if(node == NULL){
        return FALSE;
    }
    //情况2：该结点为叶子结点
    else if((node->leftChild == NULL) && (node->rightChild == NULL)){
        //没有父结点，又为叶子结点
```

```
        //说明该树只有一个结点，那么直接置为空即可
        if(parent == NULL){
            bst = NULL;
        }
        //否则直接删除（将父结点的对应指针设置为空）
        else if(parent->leftChild == node){
            parent->leftChild = NULL;
        }else{
            parent->rightChild = NULL;
        }
        return TRUE;
    }
    //情况3：该结点只有左孩子
    else if((node->leftChild != NULL) && (node->rightChild == NULL)){
        //没有父结点，又只有左孩子
        //说明这棵树只有左子树
        if(parent == NULL){
            bst = node->leftChild;
        }
        //直接把被删除结点的左孩子接到被删除结点的父结点上
        //爷爷收养内（左）孙子
        else if(parent->leftChild == node){
            parent->leftChild = node->leftChild;
        }else{
            parent->rightChild = node->leftChild;
        }
        return TRUE;
    }
    //情况4：该结点只有右孩子
    else if((node->leftChild == NULL) && (node->rightChild != NULL)){
        //没有父结点，又只有右孩子
        //说明这棵树只有右子树
        if(parent == NULL){
            bst = node->rightChild;
        }
        //直接把被删除结点的右孩子接到被删除结点的父结点上
        //爷爷收养外（右）孙子
        else if(parent->leftChild == node){
            parent->leftChild = node->rightChild;
        }else{
            parent->rightChild = node->rightChild;
        }
        return TRUE;
    }
    //情况5：该结点既有左孩子又有右孩子
    else{
        //找到该结点的中序遍历的前驱或者后继顶替其位置

        //前驱；左子树一直找到最右结点；左子树中最大的
        //后继；右子树一直找到最左结点；右子树中最小的
        //这里用的是前驱顶替其位置
        BinarySortTree priorNode = node->leftChild;
        while(priorNode->rightChild != NULL){
            priorNode = priorNode->rightChild;
```

```
        }

        //替换被删除的结点；直接修改数据域，可以不用重新修改关系
        node->data = priorNode->data;

        //删除顶替顶点的原来位置
        deleteBST(node->leftChild ,priorNode->data);
        return TRUE;
    }
}
```

平衡二叉树（Balanced Binary Tree）

平衡二叉树又称AVL树

它或者是一颗空树，或者是具有下列性质的树

- 他的左子树和右子树都是平衡二叉树
- 且左子树和右子树的深度之差的绝对值不超过1

B-树和B+树

B-树是一种平衡的多路查找树，它在文件系统中很有用