第四节：二叉搜索树

4.1 二叉搜索树

二叉搜索树（BST），也称二叉排序树和二叉查找树。一棵这个树，可以为空。如果不为空，满足以下性质：

（1）非空左子树的所有键值**小于**其根结点的键值。

（2）非空右子树的所有键值**大于**其根结点的键值。

（3）左、右子树都是二叉搜索树。

1.二叉搜索树操作的函数：

Position Find(ElementType X, BinTree BST)：从二叉搜索树 BST 中查找元素 X，并返回其结点地址；

Position FindMin(ElementType X, BinTree BST)：从二叉搜索树 BST 中查找最小元素 X，并返回其结点地址；

Position FindMax(ElementType X, BinTree BST)：从二叉搜索树 BST 中查找最大元素 X，并返回其结点地址；

BinTree Insert(ElementType X, BinTree BST)

BinTree Delete(ElementType X, BinTree BST)

2.二叉搜索树的查找操作：Find

查找从根结点开始，如果树为空，返回 **NULL**。

若树非空，则根结点关键字与 X 进行比较，并进行以下处理：

（1）若 X 小于根结点键值，则在左子树中继续搜索；

（2）若 X 大于根节点键值，则在右子树中搜索；

（3）若两者相等，搜索完成，返回指向此结点的指针。

代码如下：

```
Position Find(ElementType X, BinTree BST)
{
    if(!BST)
        return NULL;
    if(X>BST->Data)
        return Find(X,BST->Right);
    else if(X<BST->Data)
        return Find(X, Bst->Left);
    else
        return BST;
}
```

由于非递归函数的执行效率高，可将尾递归函数修改为迭代函数：

```
Position Find(ElementType X, BinTree BST)
{
    while(BST){
        if(X>BST->Data)
```

```
                BST= BST->Right;
            else if(X<BST->Data)
                BST= BST->Left;
        else
            return BST;
        }
        return NULL;
}
```

3.查找最大和最小元素

最大元素一定在树的最右分支的端结点上；最小元素在最左分支的端结点上。

```
/*查找最小元素的递归函数*/
Position FindMin(BinTree BST)
{
        if(!BST) return NULL;
        else if(!BST->Left)
            return BST;
        else
            return FindMin(BST->Left);
}
/*查找最大元素的迭代函数*/
Position FindMax(BinTree BST)
{
        if(BST)
            while(BST->Right)
                BST = BST->Right;
        return BST;
}
```

4.二叉搜索树的插入

分析：关键是要找到元素应该插入的位置，可以采用与 Find 类型的方法。

```
BinTree Insert(ElementType X, BinTree BST)
{
        if(!BST){
            BST = malloc(sizeof(struct TreeNode));
            BST->Data =X;
            BST->Left = BST->Right =NULL;
        }
        else if(X<BST->Data)
            BST->Left = Insert(X, BST->Left);
        else if(X>BST->Data)
            BST->Right = Inser(X,BST->Right);
        return BST;
}
```

4.二叉搜索树的删除

我们需要考虑三种情况：

（1）要删除的是叶结点：直接删除，并再修改其父结点指针为 NULL。

（2）要删除的结点只有一个孩子结点：要将其**父结点的指针**指向要删除结点的**孩子结点**。

（3）要删除的结点有左、右两棵子树：用另一结点代替被删除的结点：右子树的最小元素或者左子树的最大元素。如图 2 所示。

图 2

代码如下所示：

```
BinTree Delete( ElementType X, BinTree BST )
{     Position Tmp;
      if( !BST ) printf("要删除的元素未找到");
      else if( X < BST->Data )
              BST->Left = Delete( X, BST->Left); /*  左子树递归删除  */
      else if( X > BST->Data )
              BST->Right = Delete( X, BST->Right); /*  右子树递归删除  */
      else /*找到要删除的结点  */
          if( BST->Left && BST->Right ) { /*被删除结点有左右两个子结点  */
              Tmp = FindMin( BST->Right );
                            /*在右子树中找最小的元素填充删除结点*/
              BST->Data = Tmp->Data;
              BST->Right = Delete( BST->Data, BST->Right);
                                    /*在删除结点的右子树中删除最小元素*/
          } else { /*被删除结点有一个或无子结点*/
              Tmp = BST;
              if( !BST->Left ) /*  有右孩子或无子结点*/
                  BST = BST->Right;
              else if( !BST->Right ) /*有左孩子或无子结点*/
                  BST = BST->Left;
              free( Tmp );
          }
      return BST;
}
```

4.2 平衡二叉树

1.平衡二叉树基本介绍

查找二叉搜索树（BST）的时间复杂度（最坏情况下）用查找过程中的比较次数来衡量，它取决于树的深度。

对于二叉树中任一结点 T，其“平衡因子（Balance Factor，简称 BF）”定义为 BF(T)=hL-hR。（其中 hL 和 hR 分别为 T 的左、右子树的高度。）

“平衡二叉树（Balanced Binary Tree）”又称为“AVL 树”。 AVL 树或者是一棵空树，或者是具有下列性质的非空二叉搜索树：任一结点左、右子树高度差的绝对值不超过 1。”即 **|BF(T) |≤ 1**。

平衡二叉树的高度能达到 logn 吗？

设 nh 高度为 h 的平衡二叉树的最少结点树。结点数最少时:

nh=n(h-1)+n(h-2)+1

具体见图 2.

图 2

1.平衡二叉树的调整

（1）右单旋

如图 3 所示，当只将 Mar 加上 May 的时候，其平衡因子为-1，但是当增加了一个 Nov 的时候，其平衡引子变成了-2，不符合平衡二叉树的要求，因此很显然我们会想到将其调整到图 3 右侧的结构，这样其三个结点的平衡因子都是 0 了。

图 3

首个不平衡的“发现者”是 Mar（未必是根结点），它是调整起点位置。而“麻烦结点”Nov 在发现者右子树的右边，因而叫 RR 插入，需要 RR 旋转（右单旋）；一般情况（设 A 是首个发现者）的调整方式如下:

图 4

（2）左单旋

图 5

（3）左-右双旋

图 6

（4）右-左双旋

图 7