9 第四讲 树（中）

笔记本：	浙江大学《数据结构》
创建时间：	2025/4/5 17:34
更新时间：	2025/4/6 11:58
作者：	panhengye@163.com
URL：	https://www.doubao.com/chat/2609993353268738

二叉搜索树

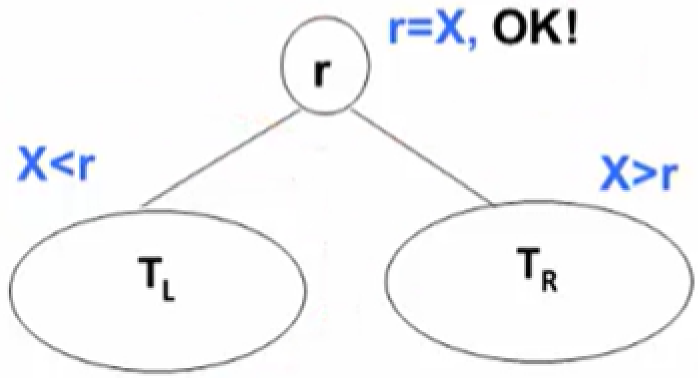
什么是二叉搜索树?

- 查找有静态与动态查找两种类型
- 静态查找可以用二分法
  - 得益于事先有效地组织了数据
  - 查找效率就是树的高度
  - 那么引发思考：**有没有可能直接把数据放在树上?**
- 这样动态查找效率会高于存在线性表中

定义：二叉搜索树（BST，Binary Search Tree），也称二叉排序树或者二叉查找树

- 可以为空
- 如果不空，则：
  - 非空左子树的所有键值小于其根结点的键值
  - 非空右子树的所有键值大于其根结点的键值
  - 左、右子树都是二叉搜索树

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
- 如果非空，则根结点关键字和X进行比较
    - X < 根结点键值，只需在左子树中继续搜索
    - X > 根结点键值，只需在左子树中继续搜索
    - 如果两者比较结果是相等，返回指向此结点的指针
- 

- 递归实现方案

```
Position Find( ElementType X, BinTree BST )
{
    if( !BST ) return NULL; /*查找失败*/
    if( X > BST->Data )
        return Find( X, BST->Right ); /*在右子树中继续查找*/
    Else if( X < BST->Data )
        return Find( X, BST->Left ); /*在左子树中继续查找*/
    else /* X == BST->Data */
        return BST; /*查找成功，返回结点的找到结点的地址*/
}
```

- 迭代实现方案

```
Position IterFind( ElementType X, BinTree BST )
{
    while( BST ) {
        if( X > BST->Data )
            BST = BST->Right; /*向右子树中移动，继续查找*/
        else if( X < BST->Data )
            BST = BST->Left;  /*向左子树中移动，继续查找*/
        else /* X == BST->Data */
            return BST; /*查找成功，返回结点的找到结点的地址*/
    }
    return NULL; /*查找失败*/
}
```

思考:

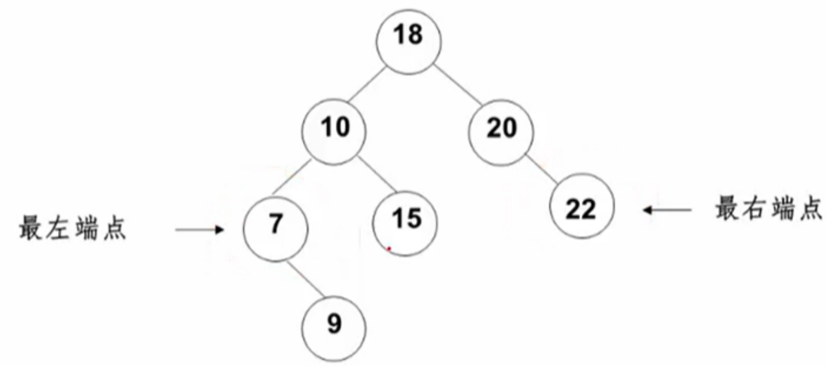
- 二叉搜索树的搜索效率取决于树的高度(depth），如果树的结构不好，全部聚集到右侧形成一条长链，那么它的效率是n-1，而不是$\log_2 n$

- 因此如何存放数据很重要，引出了后续的议题：平衡二叉树

二叉搜索树的查找：最小值和最大值

查找最大和最小元素

- 最大元素一定是在树的最右分枝的端结点上
- 最小元素一定是在树的最左分枝的端结点上



-
- 实现方法

```
Position FindMin( BinTree BST )
{
    if( !BST ) return NULL; /*空的二叉搜索树，返回NULL*/
    else if( !BST->Left )
        return BST;   /*找到最左叶结点并返回*/
    else
        return FindMin( BST->Left ); /*沿左分支继续查找*/
}
```

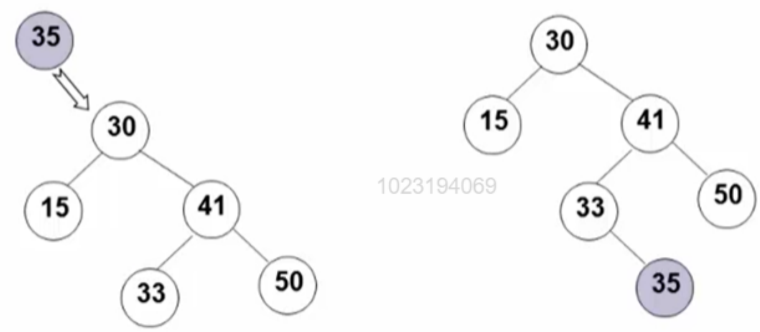
查找最小元素的递归函数

```
Position FindMax( BinTree BST )
{
    if(  BST )
        while( BST->Right )   BST = BST->Right;
              /*沿右分支继续查找，直到最右叶结点*/
    return BST;
}
```

查找最大元素的迭代函数

二叉搜索树的插入

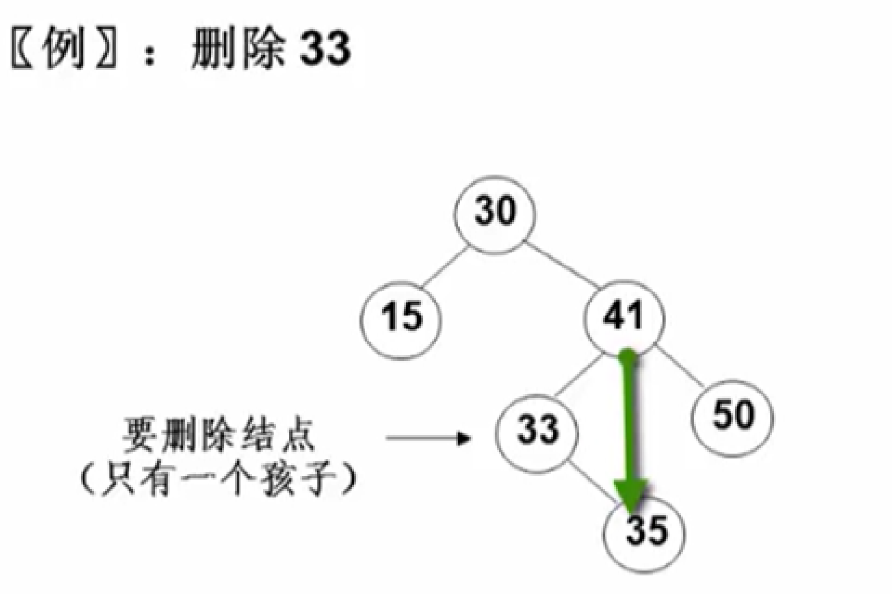
〖分析〗关键是要找到元素应该插入的位置，
可以采用与Find类似的方法



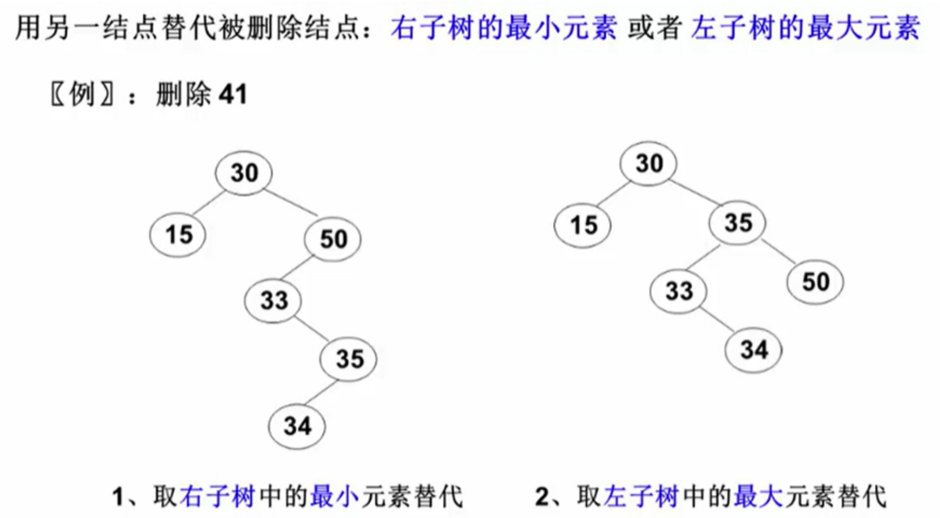

```
BinTree Insert( ElementType X, BinTree BST )
{
    if( !BST ){
            /*若原树为空，生成并返回一个结点的二叉搜索树*/
        BST = malloc(sizeof(struct TreeNode));
        BST->Data = X;
        BST->Left = BST->Right = NULL;
    }else  /*开始找要插入元素的位置*/
        if( X < BST->Data )
            BST->Left = Insert( X, BST->Left);
                        /*递归插入左子树*/
        else  if( X > BST->Data )
            BST->Right = Insert( X, BST->Right);
                        /*递归插入右子树*/
        /* else X已经存在，什么都不做 */
    return BST;
}
```

二叉树的删除

- 要删除的是叶结点：直接删掉，父结点指向Null
- 要删除的结点只有一个孩子结点
    - 将父结点的指针指向要删除的孩子结点



- 要删除的结点有左、右两棵子树

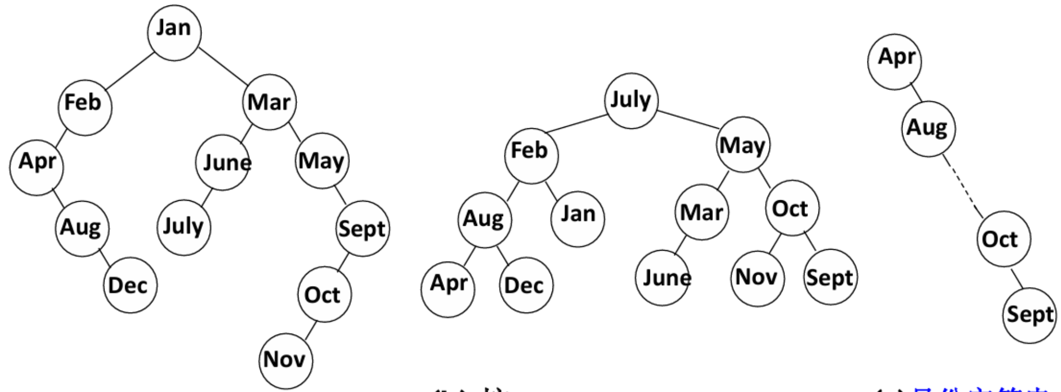


  - 因为这两个结点一定不会有两个“儿子”——相当于将问题简化了

```
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;
```

平衡二叉树

〖例〗搜索树结点不同插入次序，将导致不同的深度和平均查找长度**ASL**



(a) 自然月份序列

ASL(a)=(1+2×2+3×3+4×3+5×2+6×1)/12 = 3.5

(b) 按**July, Feb, May, Mar, Aug, Jan, Apr, Jun, Oct, Sept, Nov, Dec**

ASL(b)=3.0

(c)月份字符串大小顺序

ASL(c)= 6.5

什么是平衡二叉树

平衡因子（Balance Factor, 简称BF）： BF(T) = h_L - h_R

平衡二叉树（Balanced Binary Tree）(AVL树）—— Adelson - Velsky and Landis Tree，以发明者的名字命名

- 空树
- 或者，对**任一结点**平衡因子绝对值 ≤ 1

平衡二叉树的搜索效率是 $\log_2 n$

平衡二叉树的调整

命名规律

旋转模式	插入位置	命名含义
LL	左子树的左子树	Left-Left
RR	右子树的右子树	Right-Right
LR	左子树的右子树	Left-Right
RL	右子树的左子树	Right-Left

调整方向规律

旋转模式	插入位置	调整方向	规律
LL	左子树过重	右旋	只转父亲，方向相反

RR	右子树过重	左旋	只转父亲，方向相反
LR	左子树的右子树过重	先左旋再右旋	先子再父，旋向同名
RL	右子树的左子树过重	先右旋再左旋	先子再父，旋向同名

调整后的树结构

- 单旋调整后，树的高度减少一层。
- 双旋调整后，树的高度可能保持不变或减少一层。

图示：

LL旋

```
调整前：
    A
   /
  B
/
C


调整后：
    B
   / \
  C   A
```

RR旋

```
调整前：
  A
   \
    B
     \
      C


调整后：
    B
   / \
  A   C
```

LR旋

```
调整前：
    A
   /
  B
   \
    C


先左旋 B：
    A
   /
  C
/
B
```

```
再右旋 A:
    C
   / \
  B   A
```

RL旋

```
调整前:
  A
   \
    B
   /
  C
/
D


先右旋 B:
  A
   \
    C
   / \
  D   B


再左旋 A:
    C
   / \
  A   B
 /
D
```

注意：有时候插入元素即便不需要调整结构，也可能需要重新计算平衡因子

判别是否为同一棵二叉搜索树

如何比较两棵二叉搜索树是否相同?

思路1：建两棵树，先比较根结点，然后用递归去对比左子节点、右子节点

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_same_tree(p, q):
    if not p and not q:
        return True
    if not p or not q:
        return False
    if p.val != q.val:
        return False
    return is_same_tree(p.left, q.left) and is_same_tree(p.right, q.right)
```

思路2：不建树

3 1 2 4	vs	3 4 1 2
{1 2} 3 {4}		{1 2} 3 {4}

3 1 2 4	vs	3 2 4 1
{1 2} 3 {4}		{2 1} 3 {4}

实现该思路的代码demo

```
def compare_trees(list1, list2):
    # 找到根节点(3)的位置
    root1_idx = list1.index(3)
    root2_idx = list2.index(3)

    # 分离小于根节点和大于根节点的元素（保持原始顺序）
    left1 = [x for x in list1 if x < 3]
    right1 = [x for x in list1 if x > 3]

    left2 = [x for x in list2 if x < 3]
    right2 = [x for x in list2 if x > 3]

    # 比较左子树和右子树的元素及顺序
    return left1 == left2 and right1 == right2


# 测试示例(预期是数1为真，树2为假)
tree1_1 = [3, 1, 2, 4]
tree1_2 = [3, 4, 1, 2]
tree2_1 = [3, 1, 2, 4]
tree2_2 = [3, 2, 4, 1]


print(compare_trees(tree1_1, tree1_2))
print(compare_trees(tree2_1, tree2_2))
```

思路3

建立一棵树，然后让它和一个序列比较

要思考的三个问题

- 3.1 搜索树表示
- 3.2 建搜索树T
- 3.3. 判断一序列是否与搜索树T一致

3.1 伪代码（flag标识有没有被访问过）

```
typedef struct TreeNode *Tree;
struct TreeNode {
        int v;
        Tree Left, Right;
        int flag;
};
```

代码框架

```
int main()
{ 对每组数据
    ● 读入N和L
    ● 根据第一行序列建树T
    ● 依据树T分别判别后面的
      L个序列是否能与T形成
      同一搜索树并输出结果

    return 0;
}
```

```
int main()
{      int N, L, i;
       Tree T;

       scanf("%d", &N);
       while (N) {
            scanf("%d", &L);
            T = MakeTree(N);
            for (i=0; i<L; i++) {
                 if (Judge(T, N)) printf("Yes\n");
                 else   printf("No\n");
                 ResetT(T);        /*清除T中的标记flag*/
            }
            FreeTree(T);
            scanf("%d", &N);
       }

    return 0;
}
```

3.2 建树

如何建搜索树

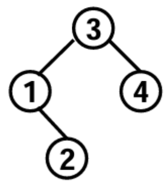
```
Tree MakeTree( int N )
{          Tree T;
           int i, V;

           scanf("%d", &V);
           T = NewNode(V);
           for (i=1; i<N; i++) {
                     scanf("%d", &V);
                     T = Insert(T, V);
           }
           return T;
}
```

```
Tree Insert( Tree T, int V )
{
   if ( !T ) T = NewNode(V);
   else {
     if ( V>T->v )
         T->Right = Insert( T->Right, V );
     else
         T->Left = Insert( T->Left, V );
   }
   return T;
}
```

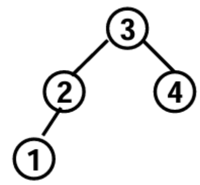
```
Tree NewNode( int V )
{   Tree T = (Tree)malloc(sizeof(struct TreeNode));
    T->v = V;
    T->Left = T->Right = NULL;
    T->flag = 0;
    return T;
}
```

3.3 判别

通过3 1 4 2构造的T



3 2 4 1对应的树



- **如何判别序列3 2 4 1是否 与树T一致？**

  方法：在树T中按顺序搜索序列3 2 4 1中的每个数

  - 如果每次搜索所经过的结点在前面均出现过，则一致
  - 否则（某次搜索中遇到前面未出现的结点），则不一致