树

2023年7月16日　　20:41

树的定义：n个结点构成的有限集合

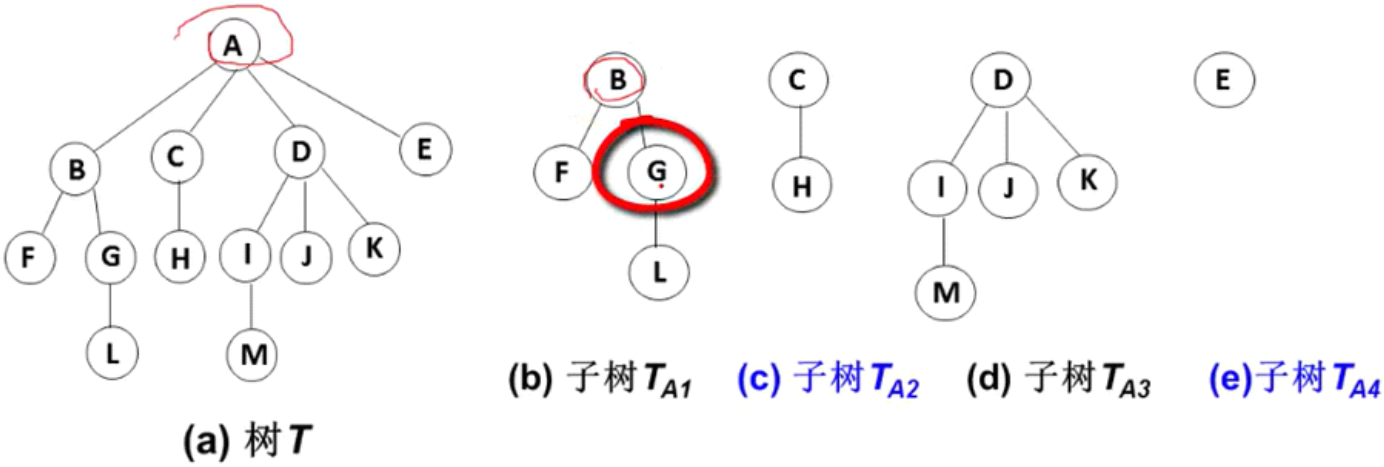
树的定义

树（Tree）：n（n≥0）个结点构成的有限集合。

当n=0时，称为空树；

对于任一棵非空树（n> 0），它具备以下性质：

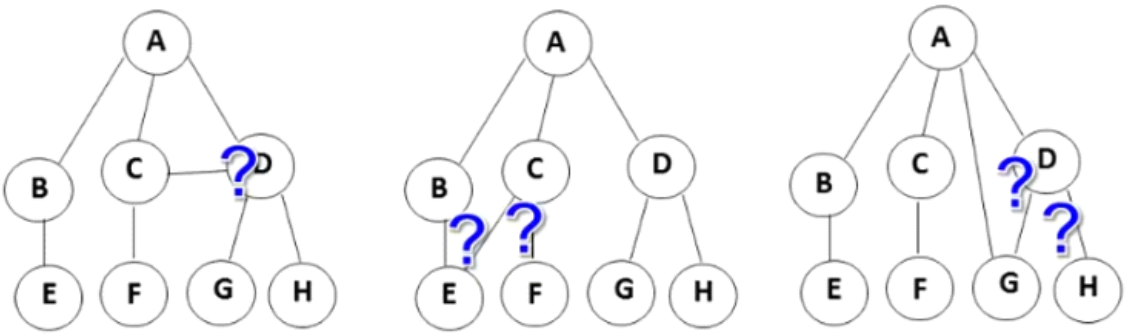
- 树中有一个称为“根（Root）”的特殊结点，用 r 表示；
- 其余结点可分为m(m>0)个互不相交的有限集T_1，T_2，…，T_m，其中每个集合本身又是一棵树，称为原来树的“子树（SubTree）”



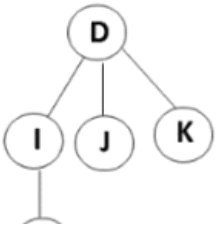
(a) 树 T　(b) 子树 T_{A1}　(c) 子树 T_{A2}　(d) 子树 T_{A3}　(e) 子树 T_{A4}

屏幕剪辑的捕获时间: 2023/7/21 18:06

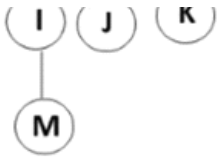
❖ 树与非树？



- 子树是不相交的；
- 除了根结点外，每个结点有且仅有一个父结点；
- 一棵N个结点的树有N-1条边。



➢ 一棵N个结点的树有N-1条边。



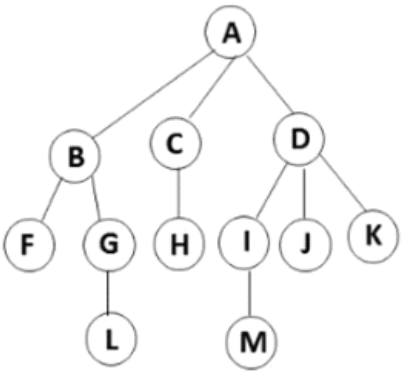
屏幕剪辑的捕获时间: 2023/7/23 19:49

树是保证所有节点联通的最小的连接方式

一些基本术语:

❖ 树的一些基本术语

1. 结点的度（Degree）：结点的子树个数
2. 树的度：树的所有结点中最大的度数
3. 叶结点（Leaf）：度为0的结点
4. 父结点（Parent）：有子树的结点是其子树的根结点的父结点
5. 子结点（Child）：若A结点是B结点的父结点，则称B结点是A结点的子结点；子结点也称孩子结点。
6. 兄弟结点（Sibling）：具有同一父结点的各结点彼此是兄弟结点。



屏幕剪辑的捕获时间: 2023/7/23 19:51

❖ 树的一些基本术语

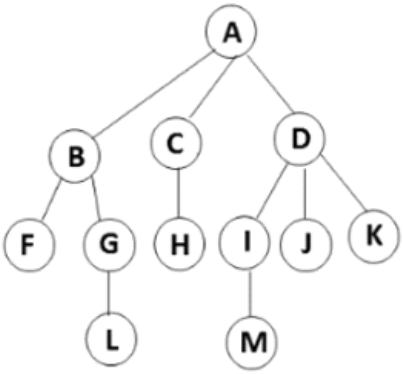
7. 路径和路径长度：从结点n_1到n_k的路径为一个结点序列$n_1, n_2, \ldots, n_k$，n_i是 n_{i+1}的父结点。路径所包含边的个数为路径的长度。

9. 祖先结点(Ancestor)：沿树根到某一结点路径上的所有结点都是这个结点的祖先结点。

10. 子孙结点(Descendant)：某一结点的子树中的所有结点是这个结点的子孙。

11. 结点的层次（Level）：规定根结点在1层，其它任一结点的层数是其父结点的层数加1。

12. 树的深度（Depth）：树中所有结点中的最大层次是这棵树的深度。



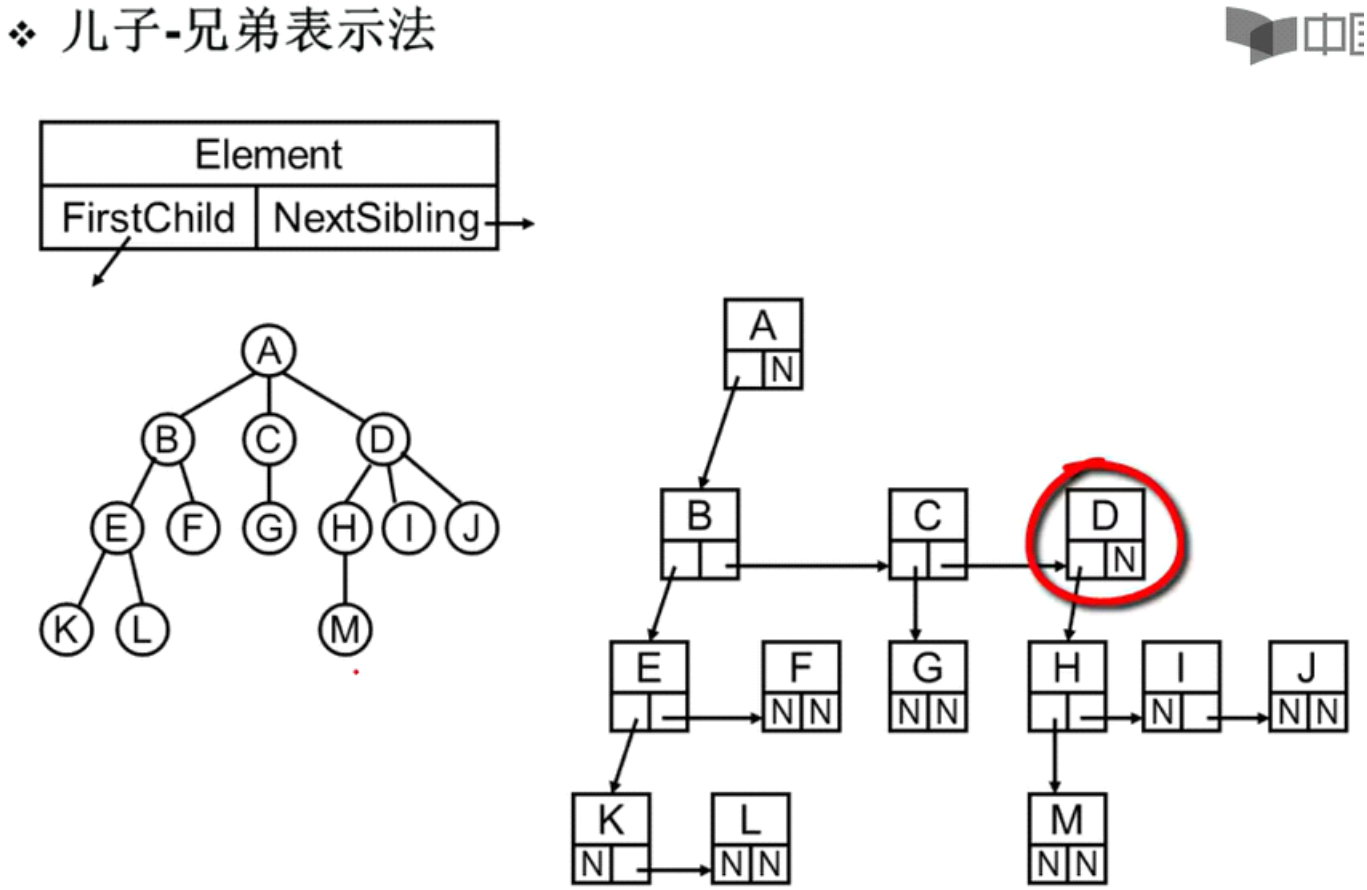
屏幕剪辑的捕获时间: 2023/7/23 19:53

树的表示

2023年7月23日 19:55

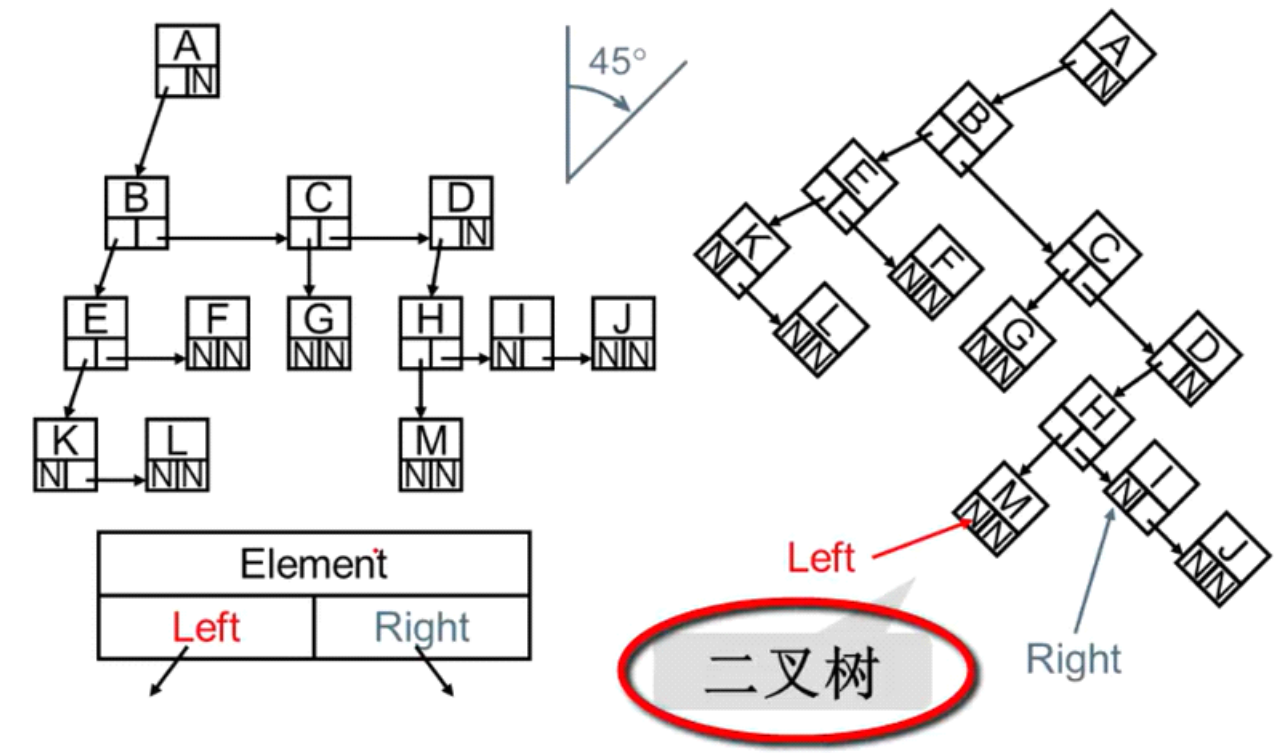
单纯的数组与普通链表都是很难建立一个树的（普通的链表可以实现，但是程序不好设计，也不好改进）
一种方法是把树的每一个结点的指针数量设置为一样的，但是会造成大量的空间浪费

比较好的实现方法：
1.Child-Sibling表示法



屏幕剪辑的捕获时间: 2023/7/23 20:44

此即**二叉树**



屏幕剪辑的捕获时间: 2023/7/23 21:00

二叉树的性质

2023年7月23日　21:05

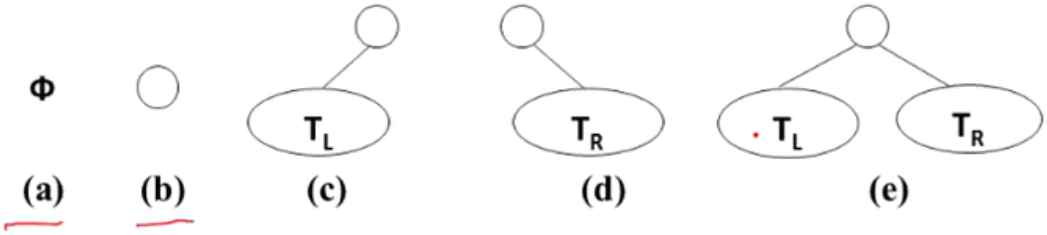
二叉树的定义

二叉树T：一个有穷的结点集合。

这个集合可以为空

若不为空，则它是由根结点和称为其左子树T_L和右子树T_R的两个不相交的二叉树组成。

- 二叉树具体五种基本形态

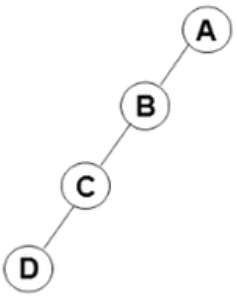


- 二叉树的子树有左右顺序之分

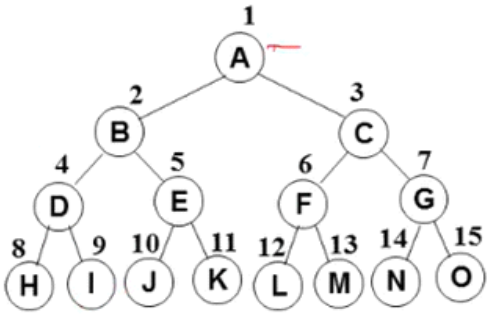
屏幕剪辑的捕获时间: 2023/7/23 21:06

特殊二叉树

- 斜二叉树(Skewed Binary Tree)



- 完美二叉树(Perfect Binary Tree)
  满二叉树(Full Binary Tree)



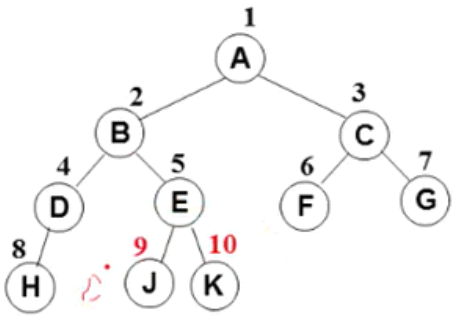
- 完全二叉树
  (Complete Binary Tree)

有n个结点的二叉树，对树中结点按从上至下、从左到右顺序进行编号，编号为i（$1 \le i \le n$）结点与满二叉树

从上至下、从左到右顺序进行编号，
编号为i（1≤i≤n）结点与满二叉树
中编号为 i 结点在二叉树中位置相同

屏幕剪辑的捕获时间: 2023/7/23 21:09

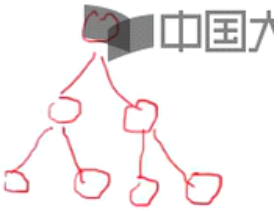
完全二叉树相对于满二叉树，可以缺掉最底层最右边的一些结点，下图就不是完全二叉树



屏幕剪辑的捕获时间: 2023/7/23 21:10

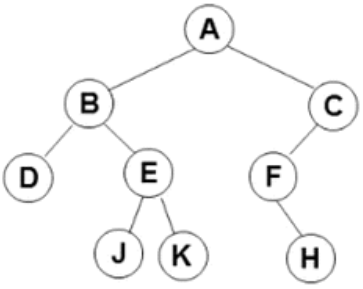
二叉树的性质:

二叉树几个重要性质



- 一个二叉树第 i 层的最大结点数为：2^{i-1}，$i \geq 1$。
- 深度为k的二叉树有最大结点总数为：2^k-1，$k \geq 1$。
- 对任何非空二叉树T，若n_0表示叶结点的个数、n_2是度为2的非叶结点个数，那么两者满足关系$n_0 = n_2 + 1$。

$1+2^1+2^2+\cdots+2^{k-1}=2^k-1$



- $n_0 = 4$，$n_1 = 2$
- $n_2 = 3$;
- $n_0 = n_2 + 1$

屏幕剪辑的捕获时间: 2023/7/23 21:16

对于第三条性质的证明，考虑对每个结点，从下往上看与从上往下看所得到的边总数相同建立等式

屏幕剪辑的捕获时间: 2023/7/23 21:17

屏幕剪辑的捕获时间: 2023/7/23 21:18

二叉树的存储形式

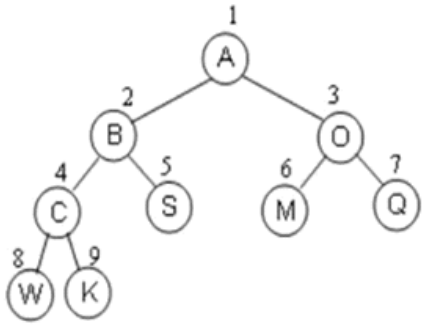
2023年7月23日　　21:22

1.顺序存储结构（数组）

完全二叉树可以使用顺序存储方式，因为各个结点的编号都是固定的

完全二叉树：按从上至下、从左到右顺序存储

n个结点的完全二叉树的结点父子关系：



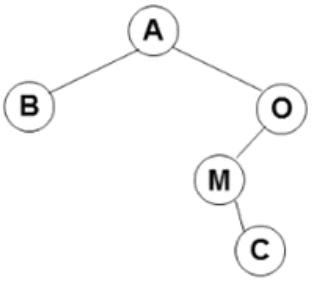
- 非根结点（序号 i > 1）的父结点的序号是⌊i / 2⌋;
- 结点（序号为 i ）的左孩子结点的序号是 2i，（若2 i <= n，否则没有左孩子）；
- 结点（序号为 i ）的右孩子结点的序号是 2i+1，（若2 i +1<= n，否则没有右孩子）；

结点	A	B	O	C	S	M	Q	W	K
序号	1	2	3	4	5	6	7	8	9

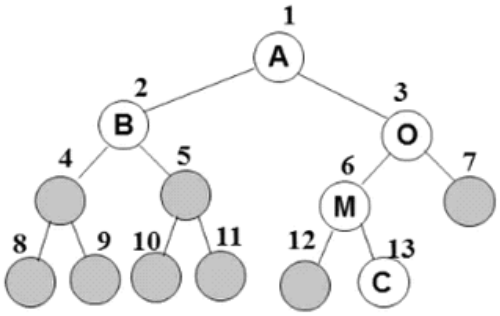
屏幕剪辑的捕获时间: 2023/7/23 21:25

一般二叉树实际可以通过补全为完全二叉树的方式进行存储，但是会造成空间浪费

- 一般二叉树也可以采用这种结构，但会造成空间浪费.....



(a)一般二叉树



(b) 对应的完全二叉树

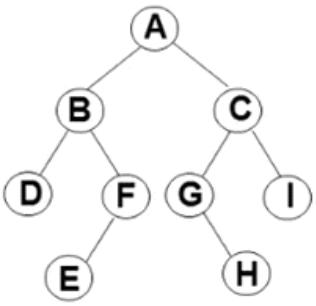
结点	A	B	O	∧	∧	M	∧	∧	∧	∧	∧	∧	C
序号	1	2	3	4	5	6	7	8	9	10	11	12	13

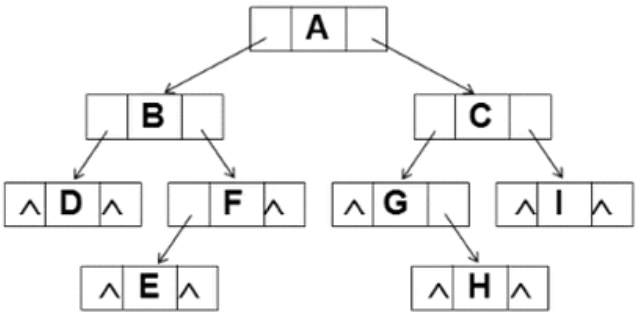
屏幕剪辑的捕获时间: 2023/7/23 21:27

2.链式存储

创建结构体TreeNode，每个结点有三个域（数据、左子树，右子树）

```
typedef  struct  TreeNode *BinTree;
typedef  BinTree Position;
struct  TreeNode{
        ElementType  Data;
        BinTree  Left;
        BinTree  Right;
}
```





屏幕剪辑的捕获时间: 2023/7/23 21:29

二叉树的递归遍历

2023年7月23日　　21:38

1.先序遍历

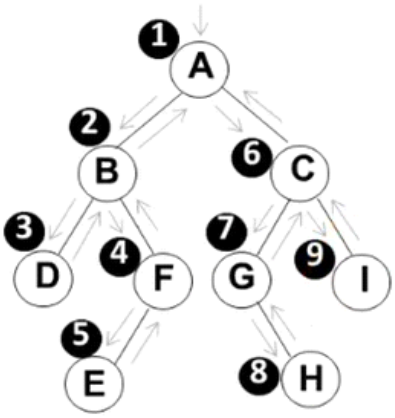
先对左边递归，再对右边递归

遍历过程为：
① 访问根结点；
② 先序遍历其左子树；
③ 先序遍历其右子树。

A（B D F E）（C G H I）

先序遍历=>　A B D F E C G H I

```
void PreOrderTraversal( BinTree BT )
{
    if( BT ) {
        printf("%d", BT->Data);
        PreOrderTraversal( BT->Left );
        PreOrderTraversal( BT->Right );
    }
}
```



屏幕剪辑的捕获时间: 2023/7/23 21:39

2.中序遍历

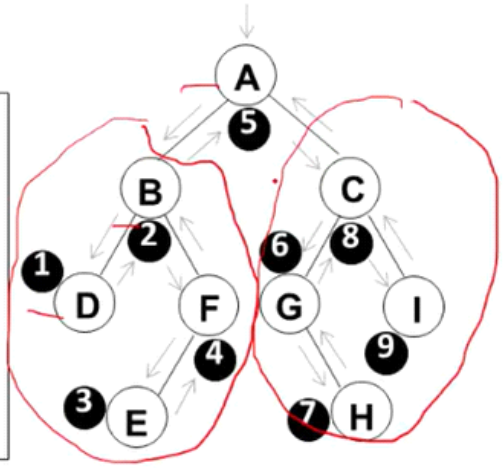
先递归遍历左树，再访问根节点，再递归遍历右树

遍历过程为：
① 中序遍历其左子树；
② 访问根结点；
③ 中序遍历其右子树。

（D B E F）A（G H C I）

中序遍历=>　D B E F A G H C I

```
void InOrderTraversal( BinTree BT )
{
    if( BT ) {
        InOrderTraversal( BT->Left );
        printf("%d", BT->Data);
        InOrderTraversal( BT->Right );
    }
}
```



屏幕剪辑的捕获时间: 2023/7/23 21:43

3.后序遍历

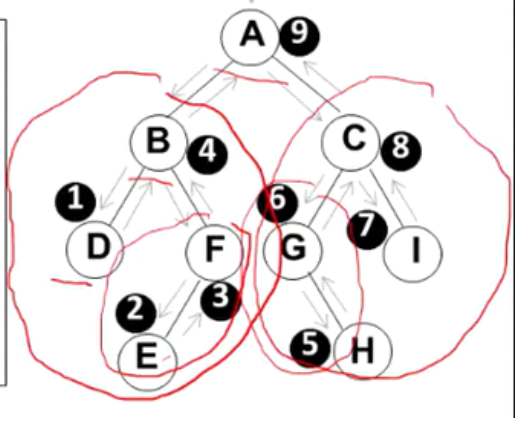
先左边遍历，再右边遍历，再访问根节点

遍历过程为：
① 后序遍历其左子树；
② 后序遍历其右子树；
③ 访问根结点。

（D E F B ）（H G I C）A

后序遍历=> D E F B H G I C A

```
void PostOrderTraversal( BinTree BT )
{
    if( BT ) {
        PostOrderTraversal( BT->Left );
        PostOrderTraversal( BT->Right);
        printf("%d", BT->Data);
    }
}
```

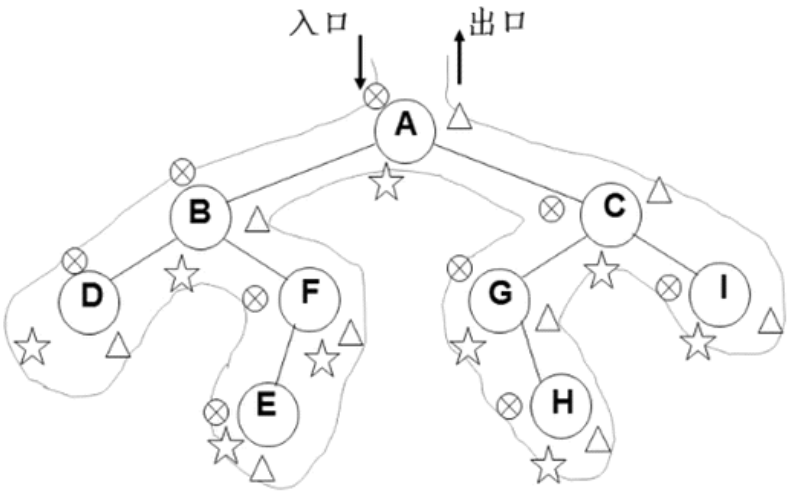


屏幕剪辑的捕获时间: 2023/7/23 21:45

遍历的路径都是相同的！！

❖ 先序、中序和后序遍历过程：遍历过程中经过结点的路线一样，只是访问各结点的时机不同。

❖ 图中在从入口到出口的曲线上用⊗、☆□ 和△三种符号分别标记出了先序、中序和后序访问各结点的时刻



屏幕剪辑的捕获时间: 2023/7/23 21:48

二叉树的非递归遍历

2023年7月23日 21:50

中序遍历非递归算法

基本思路：使用堆栈

- 遇到一个结点，就把它压栈，并去遍历它的左子树；
- 当左子树遍历结束后，从栈顶弹出这个结点并访问它；
- 然后按其右指针再去中序遍历该结点的右子树。

```
void InOrderTraversal( BinTree BT )
{   BinTree T=BT;
    Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
    while( T || !IsEmpty(S) ){
        while(T){    /*一直向左并将沿途结点压入堆栈*/
            Push(S,T);
            T = T->Left;
        }
        if(!IsEmpty(S)){
            T = Pop(S); /*结点弹出堆栈*/
            printf("%5d", T->Data); /*（访问）打印结点*/
            T = T->Right; /*转向右子树*/
        }
    }
}
```

屏幕剪辑的捕获时间: 2023/7/23 22:06

❖先序遍历的非递归遍历算法？

```
void InOrderTraversal( BinTree BT )
{    BinTree T BT;
    Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
    while( T || !IsEmpty(S) ){
        while(T){    /*一直向左并将沿途结点压入堆栈*/
            Push(S,T);
            T = T->Left;
        }
        if(!IsEmpty(S)){
            T = Pop(S); /*结点弹出堆栈*/
            printf("%5d", T->Data); /*（访问）打印结点*/
            T = T->Right; /*转向右子树*/
        }
    }
```

```
                t = t->right; //转向右子树
            }
        }
    }
```

屏幕剪辑的捕获时间: 2023/7/23 22:08

层序遍历

2023年7月24日 9:49

层序基本过程：先根结点入队，然后：

① 从队列中取出一个元素；
② 访问该元素所指结点；
③ 若该元素所指结点的左、右孩子结点非空，则将其左、右孩子的指针顺序入队。

```
void LevelOrderTraversal ( BinTree BT )
{   Queue Q;  BinTree T;
    if ( !BT ) return; /* 若是空树则直接返回 */
    Q = CreatQueue( MaxSize ); /*创建并初始化队列Q*/
    AddQ( Q, BT );
    while ( !IsEmptyQ( Q ) ) {
        T = DeleteQ( Q );
        printf("%d\n", T->Data); /*访问取出队列的结点*/
        if ( T->Left )  AddQ( Q, T->Left );
        if ( T->Right )  AddQ( Q, T->Right );
    }
}
```

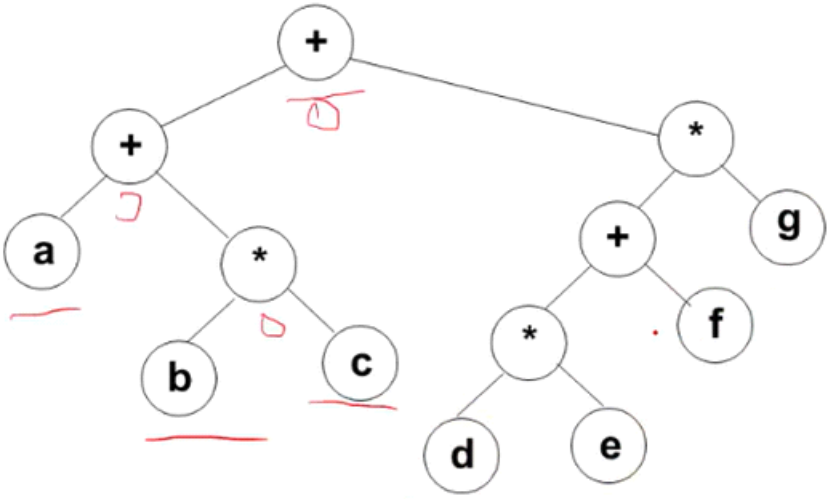
屏幕剪辑的捕获时间: 2023/7/24 9:53

一些例子

2023年7月24日　　10:40

可以通过不同的遍历顺序得到前缀、后缀表达式，但是中缀表达式会受到运算优先级影响
解决方法是输出左树时先打印一个"（"，左树遍历结束后打印一个"）"

【例】 二元运算表达式树及其遍历

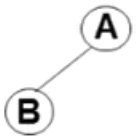


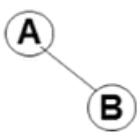
- 三种遍历可以得到三种不同的访问结果：
- 先序遍历得到前缀表达式：**+ + a * b c * + * d e f g**
- 中序遍历得到中缀表达式：**a + b * c + d * e + f * g**
- 后序遍历得到后缀表达式：**a b c * + d e * f + g * +**

屏幕剪辑的捕获时间: 2023/7/24 10:40

【例】 由两种遍历序列确定二叉树

没有中序的困扰：
- 先序遍历序列：A B
- 后序遍历序列：B A



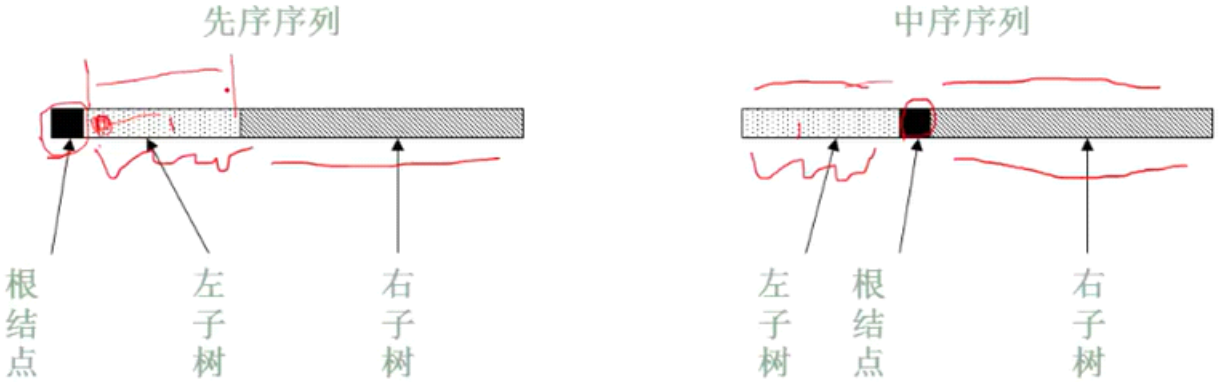


屏幕剪辑的捕获时间: 2023/7/24 10:44

❖ 先序和中序遍历序列来确定一棵二叉树

〖分析〗

- ◆ 根据先序遍历序列第一个结点确定根结点；
- ◆ 根据根结点在中序遍历序列中分割出左右两个子序列
- ◆ 对左子树和右子树分别递归使用相同的方法继续分解。



屏幕剪辑的捕获时间: 2023/7/24 10:46

查找&二叉搜索树

2023年7月16日 16:31

查找：给定某个关键词，从一个集合里把与这个关键字相等的记录找到

静态查找：找的集合本身是不动的，不向集合中增加新元素，也不删除老的元素
动态查找：除了查找，本身还可能发生插入与删除

二叉查找树的一些功能：

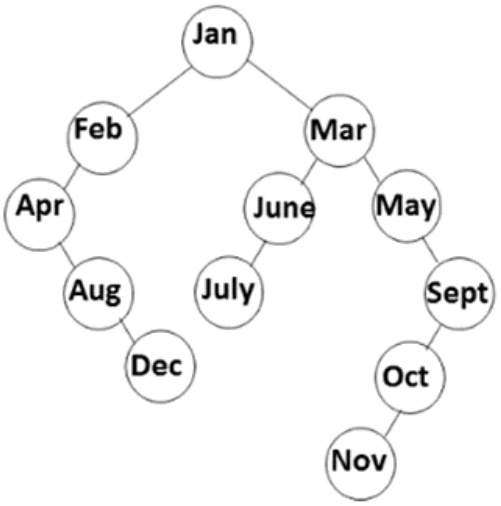
屏幕剪辑的捕获时间: 2023/7/26 10:26

平衡二叉树

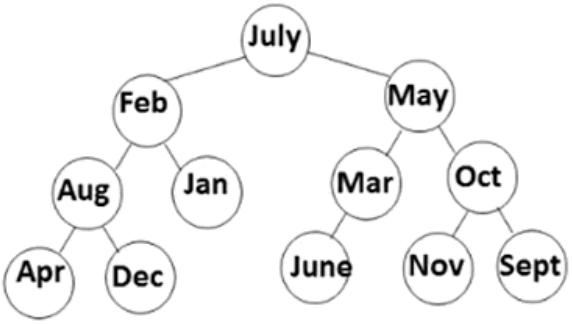
2023年7月26日 12:54

什么是平衡二叉树

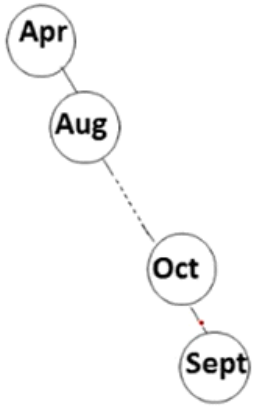
〖例〗搜索树结点不同插入次序，将导致不同的深度和平均查找长度ASL



(a) 自然月份序列



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

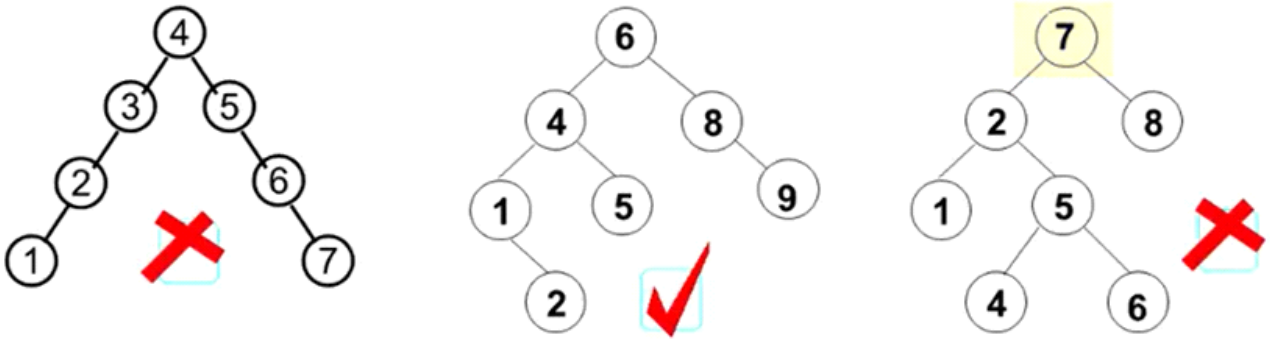


(c)月份字符串大小顺序

什么是平衡二叉树

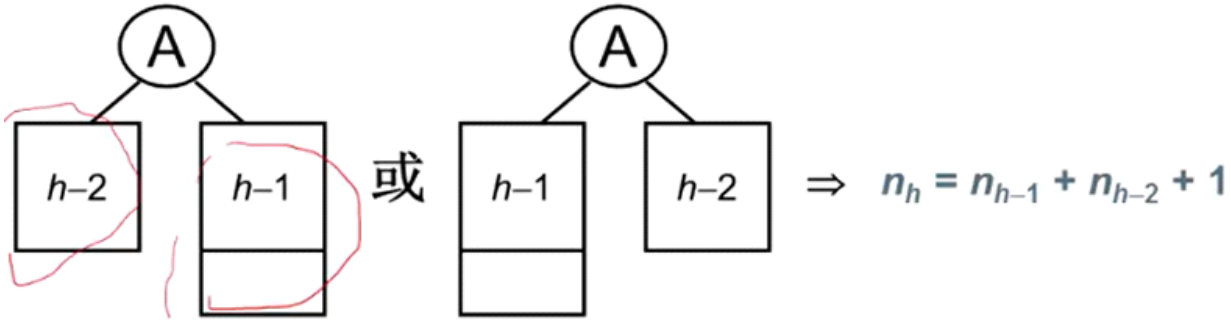
“平衡因子（Balance Factor，简称BF）：$BF(T) = h_L - h_R$，
其中h_L和h_R分别为T的左、右子树的高度。

平衡二叉树（Balanced Binary Tree）（AVL树）
空树，或者
任一结点左、右子树高度差的绝对值不超过1，即$|BF(T)| \le 1$



给定结点个数时，平衡二叉树的层数问题：

平衡二叉树的高度能达到$\log_2 n$吗？
设 n_h 高度为h的平衡二叉树的最少结点数。结点数最少时：



斐波那契序列：
$F_0 = 1,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2}$ for $i > 1$

屏幕剪辑的捕获时间: 2023/7/26 13:07

设 n_h 是高度为h的平衡二叉树的最小结点数.

h	n_h	F_h
0	1	1
1	2	1
2	4	2
3	7	3
4	12	5
5	20	8
6	33	13
7	54	21
8	88	34
9	……	

$\Rightarrow n_h = n_{h-1} + n_{h-2} + 1$

$\Rightarrow n_h = F_{h+2} - 1$, （对 $h \geq 0$）

$$F_i \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i$$

$$\Rightarrow n_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 1$$

$$\Rightarrow h = O(\log_2 n)$$

□ 给定结点数为 n的AVL树的最大高度为$O(\log_2 n)$!

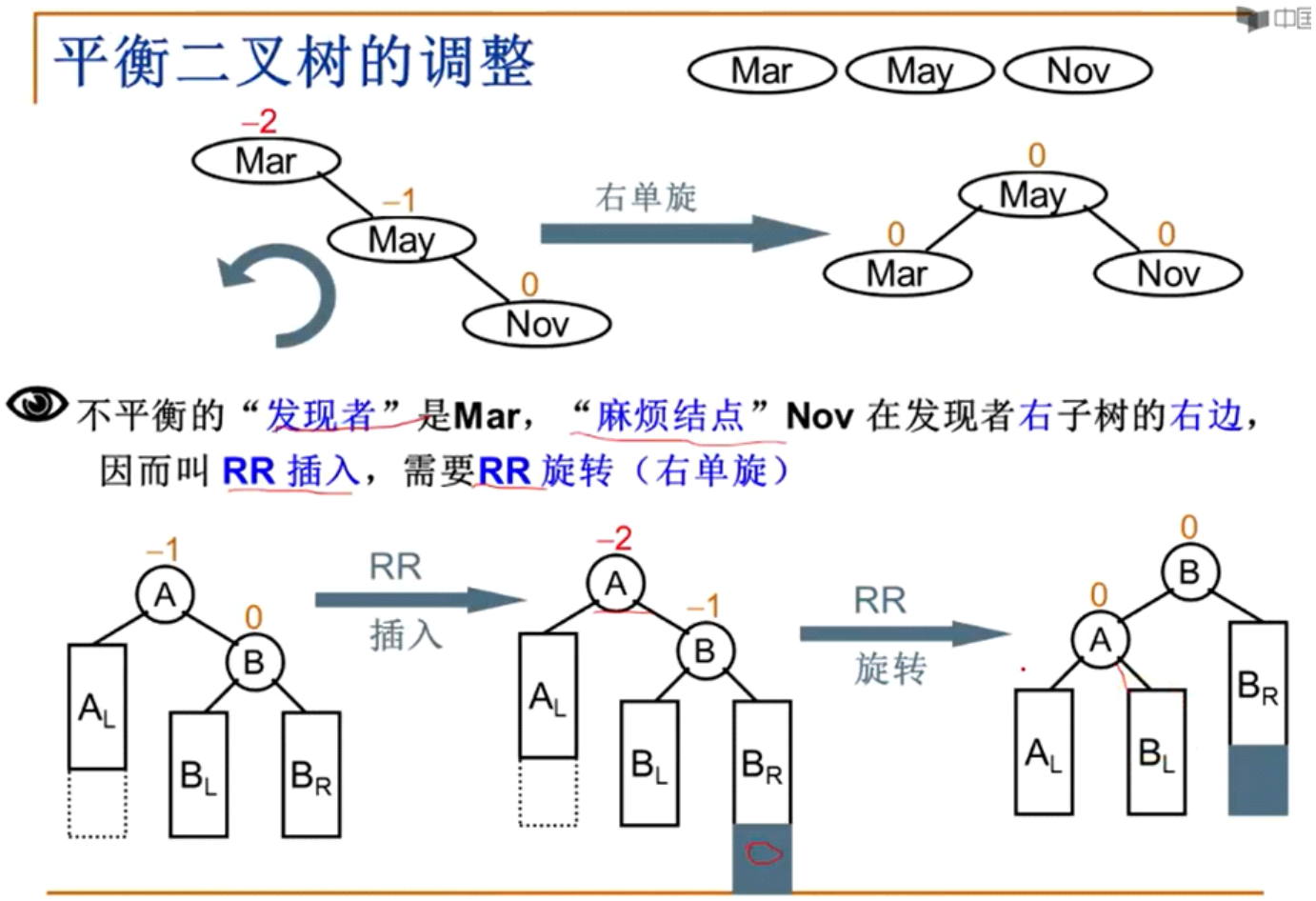
平衡二叉树的调整

2023年7月26日　　13:09

※文后有重要注记

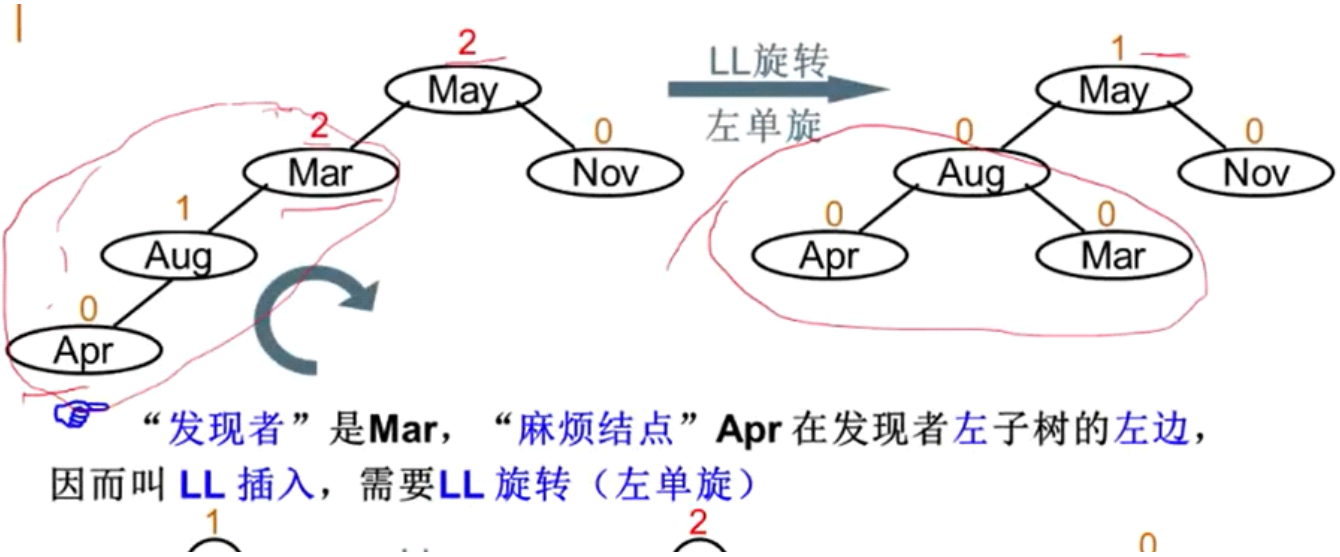
向平衡二叉树内插入、删除数据时，可能会让树变得不平衡，如何调整?

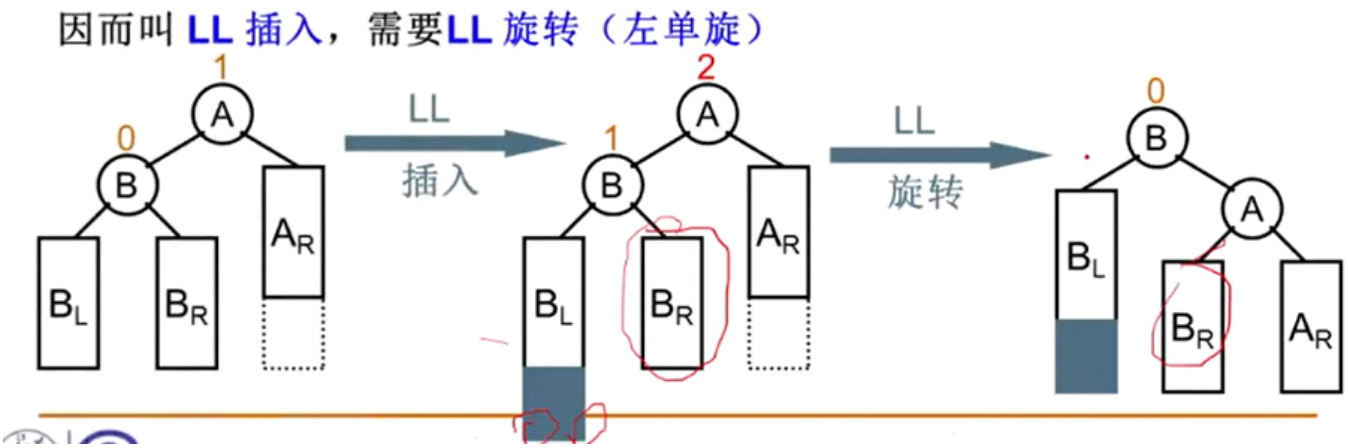
1.RR旋转



屏幕剪辑的捕获时间: 2023/7/26 13:13

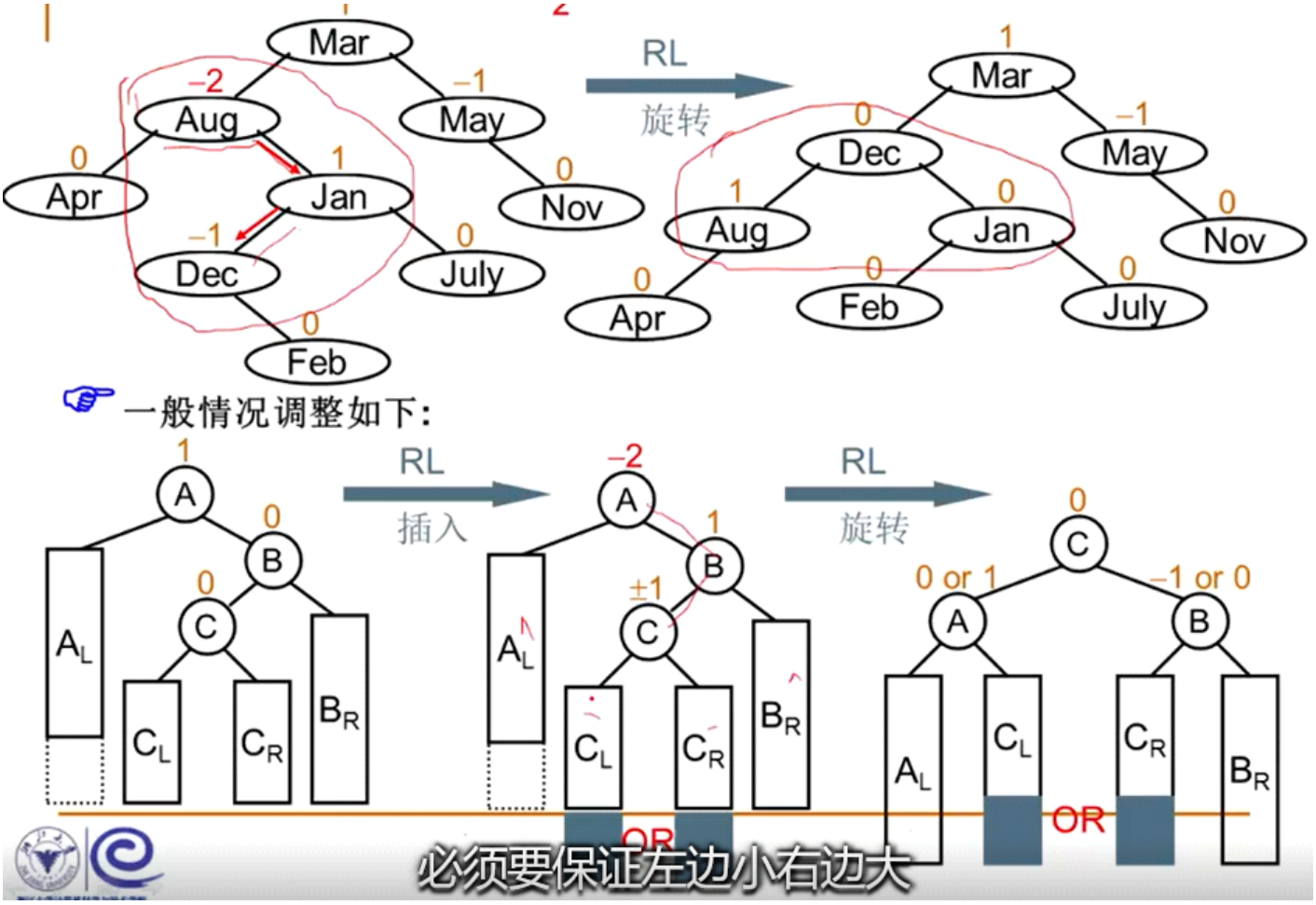
2.LL旋转



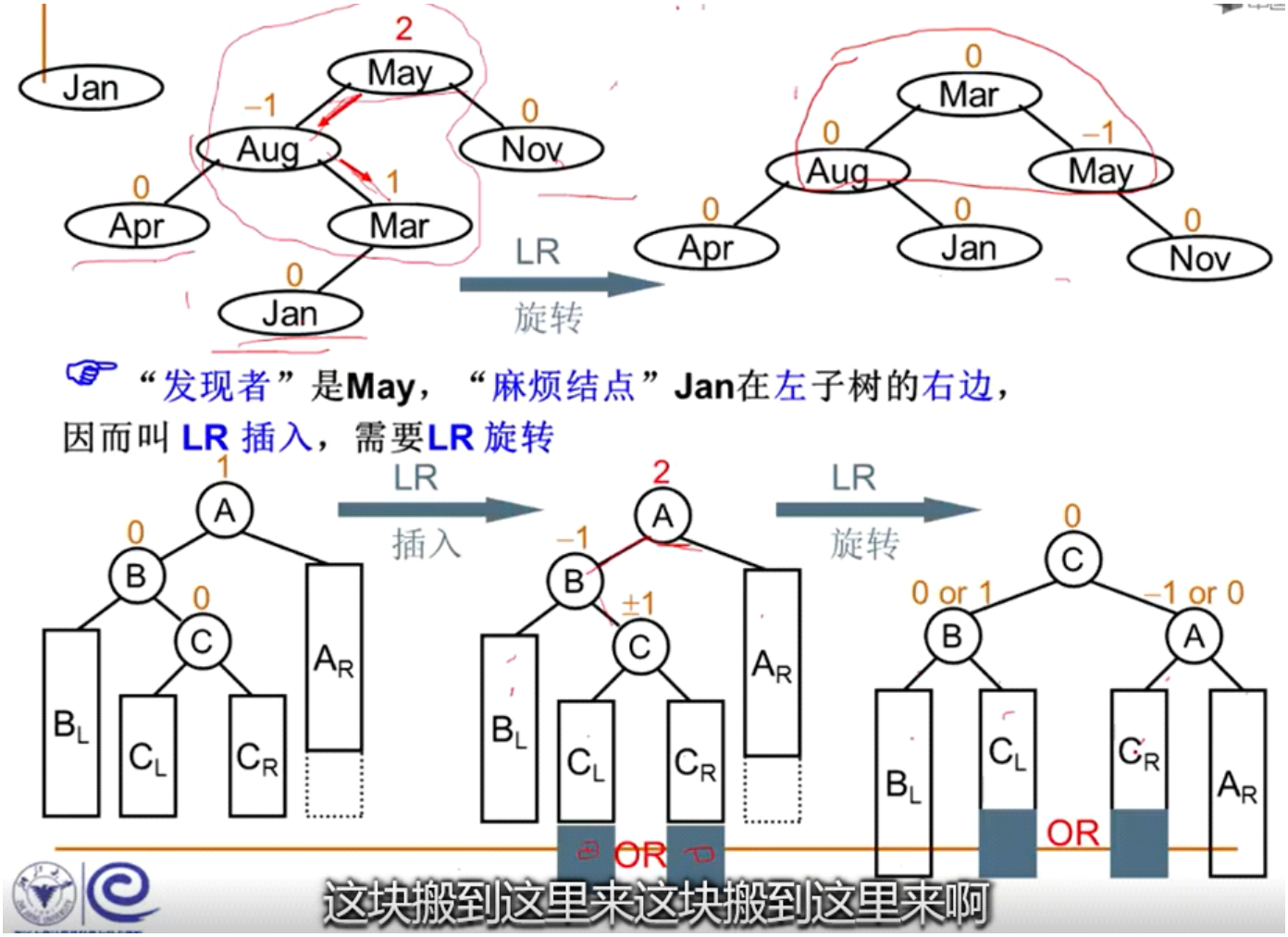


屏幕剪辑的捕获时间: 2023/7/26 13:18

3.LR旋转与RL旋转



屏幕剪辑的捕获时间: 2023/7/26 13:33



屏幕剪辑的捕获时间: 2023/7/26 13:33

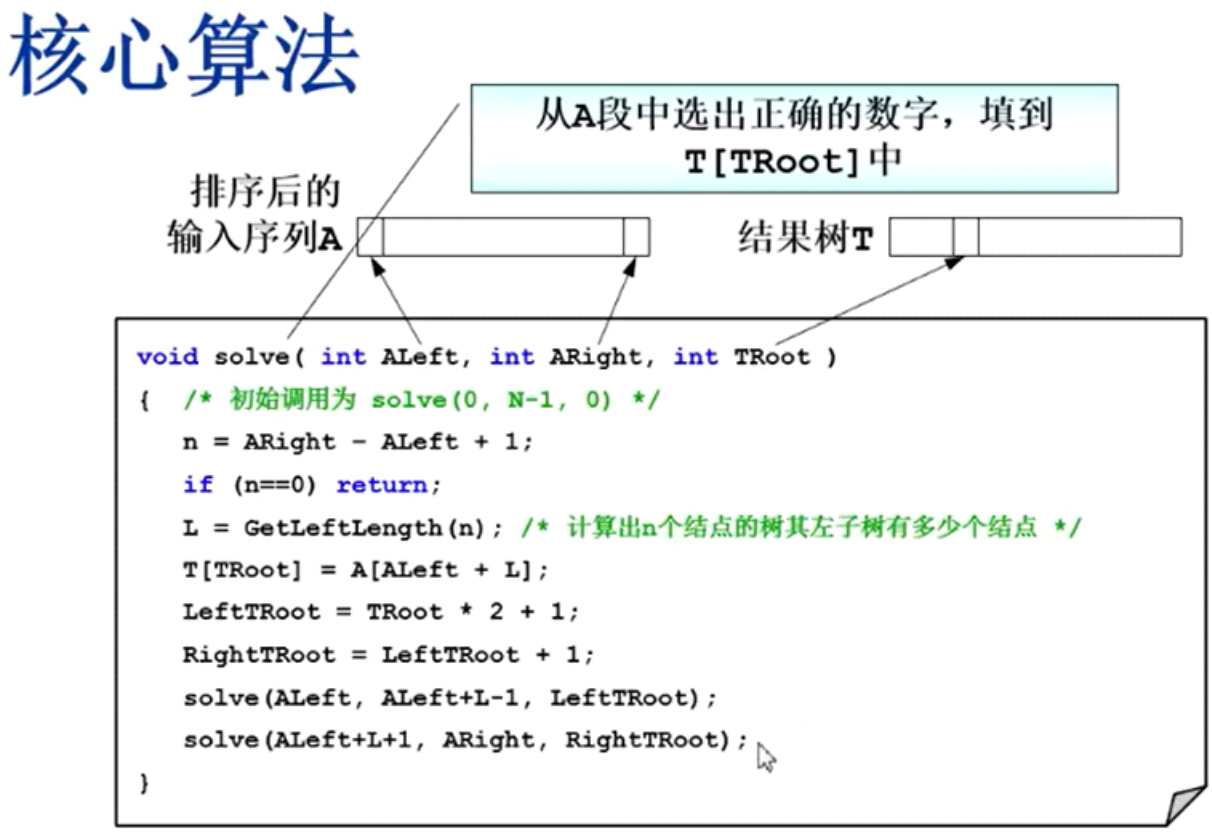
注：LR与RL旋转实质是做了两次单旋，以LR旋转为例：
记A->left=B,B->right=C,实际上是对B做一次右旋，旋转后的A->left为C，B->right为C->left，之后对A做一次左旋，这样A->left=C->right，C->right=A，C->left=B，完成.

算例：完全二叉搜索树算法

2023年7月27日　　17:23

[查看对应的算法cpp文件，点击这里]

本题选择树的存储形式也很重要，之所以选用数组，是为了充分利用完全二叉树的性质，即每一个结点都有元素（除了最后一层），所以存储空间上与链表相差不大，而数组搜索各个结点要比链表方便得多（实际就是按层序遍历的顺序来存储的），所以在本题的要求下，数组显然全方位爆鲨链表



屏幕剪辑的捕获时间: 2023/7/27 17:25

数列排序以及查找左边子树结点个数的问题不多表述，重要的是本题的递归实现方法

在实现递归的时候，由于最后两步是分别对左半边子列和右半边子列做递归，看似简单，具体操作时会发现确定每一种递归中赋予TRoot的值不易确定，要理解TRoot的含义

TRoot是当前需要插入根结点的位置，由于我们一直在对左子列做递归，递归未结束时，树结点一直在向->left的方向走，根结点从0开始编号，所以在递归左侧时，易发现每递归一次，都有LeftRoot=Troot*2+1，至于RightRoot，我们将其设置为RightRoot=LeftRoot+1后，并未参与左列的递归，直到左侧递归全部完成后，再按照**每一次递归时生成的RightRoot**将数据填入各个右结点（数组T对应的RightRoot位置），最终正序输出得到的T数组即为层序遍历结果