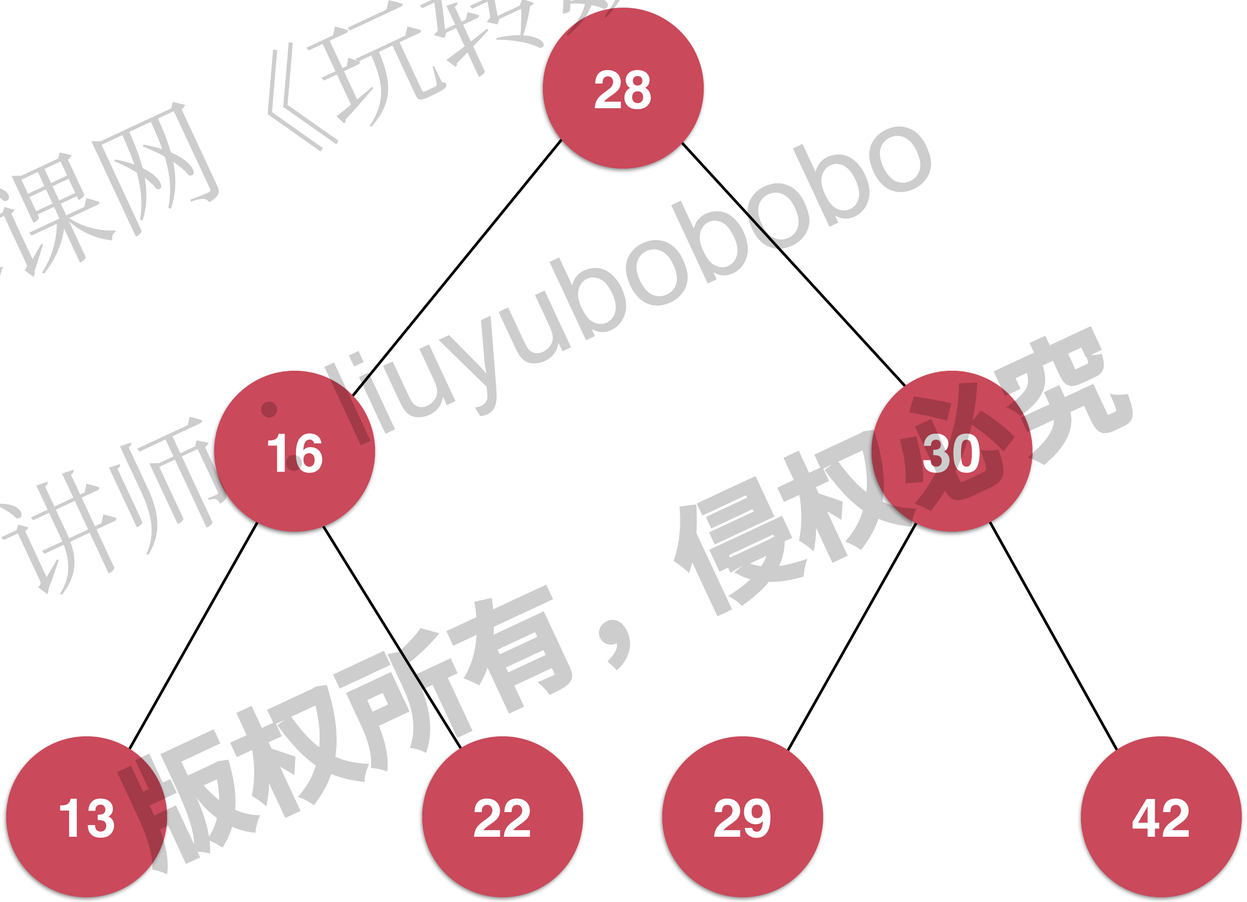
玩儿转数据结构

liuyubobobo

二分搜索树

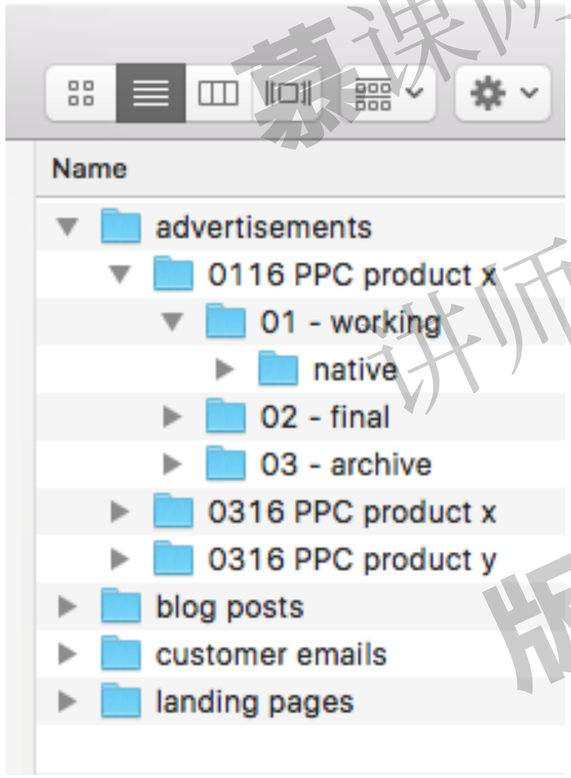
为什么要有（要学习）树结构

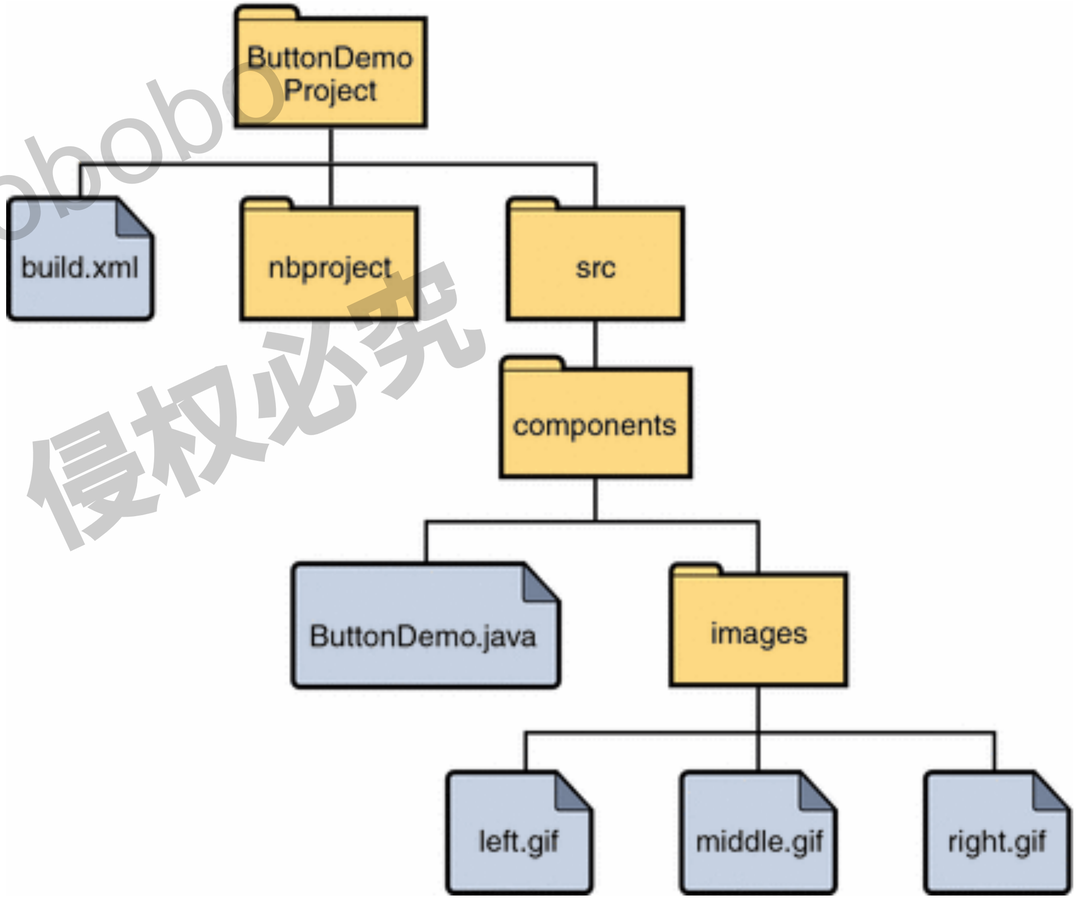
树结构



为什么要有树结构?

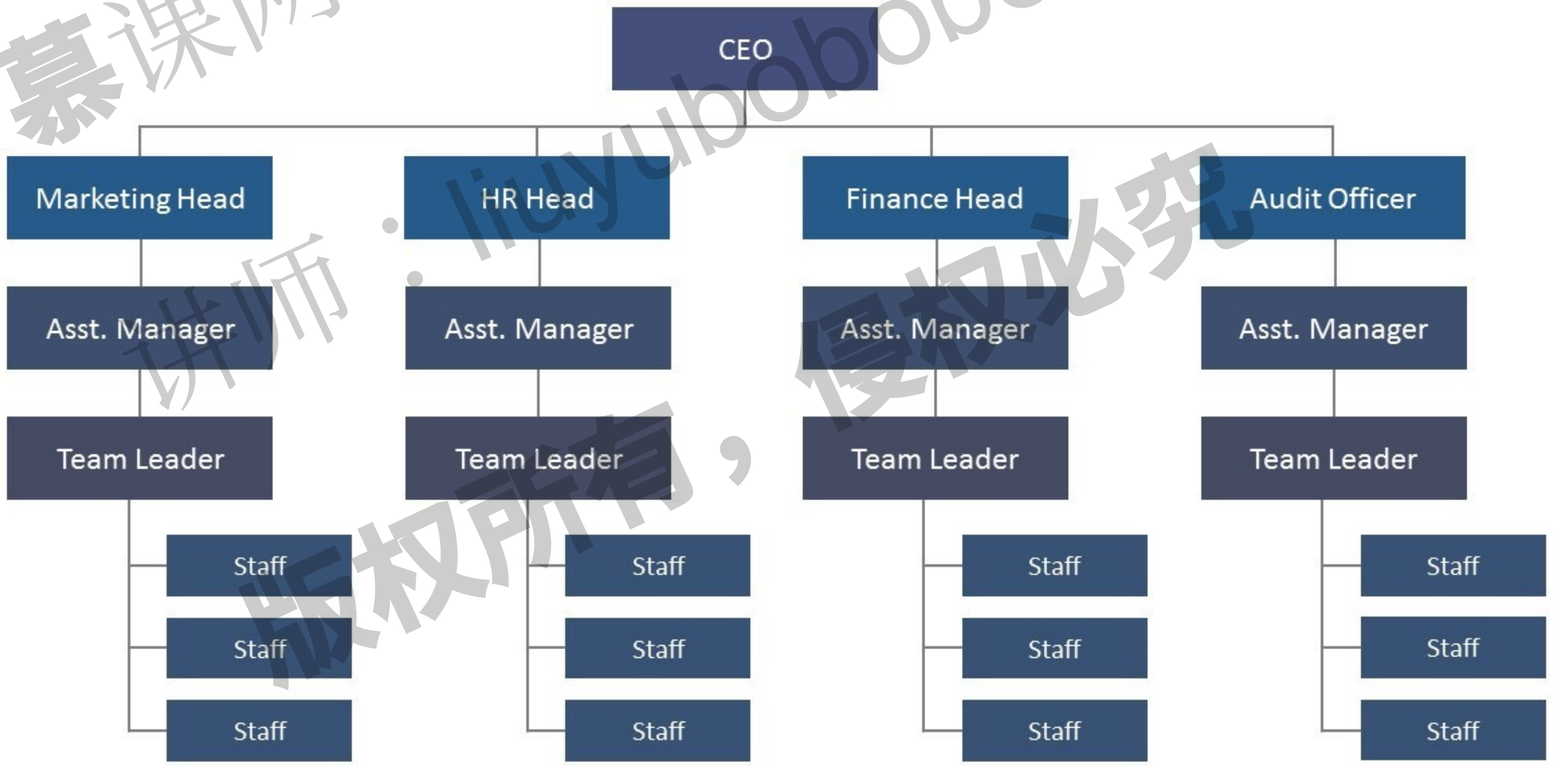
- 树结构本身是一种天然的组织结构





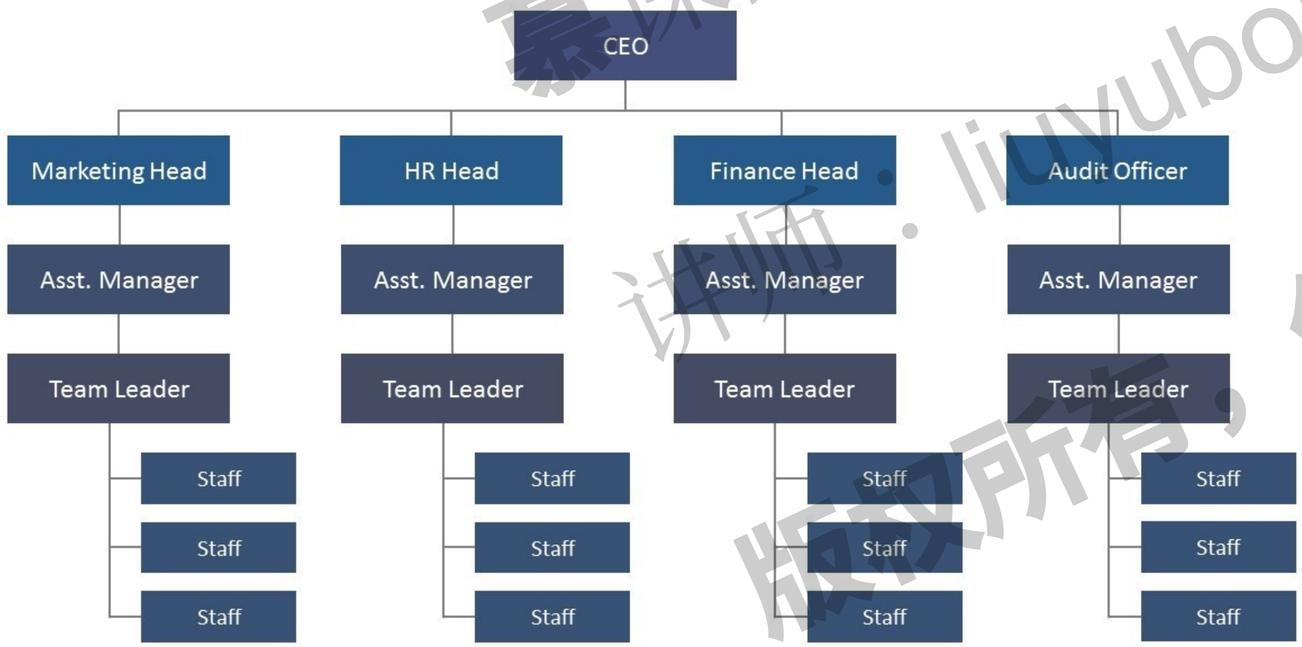
为什么要有树结构?

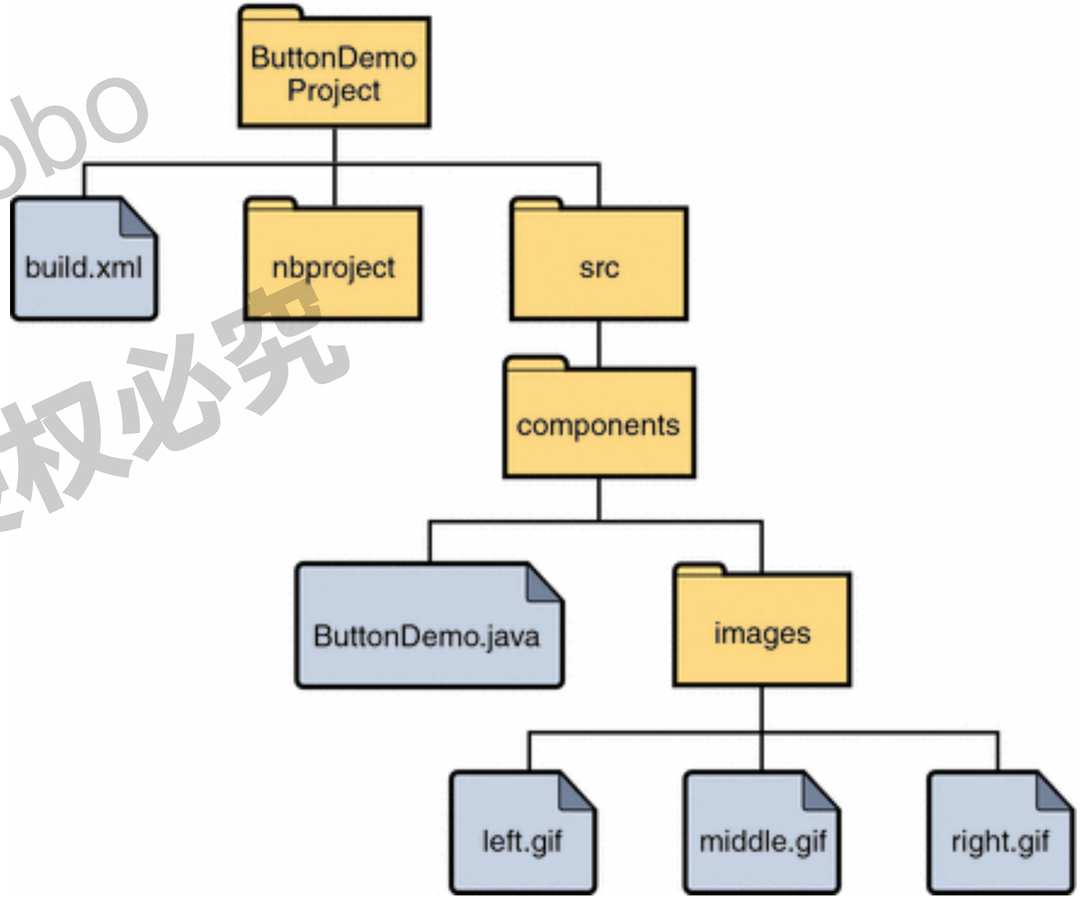
- 树结构本身是一种天然的组织结构



为什么要有树结构?

- 高效





为什么要有树结构?

- 将数据使用树结构存储后，出奇的高效

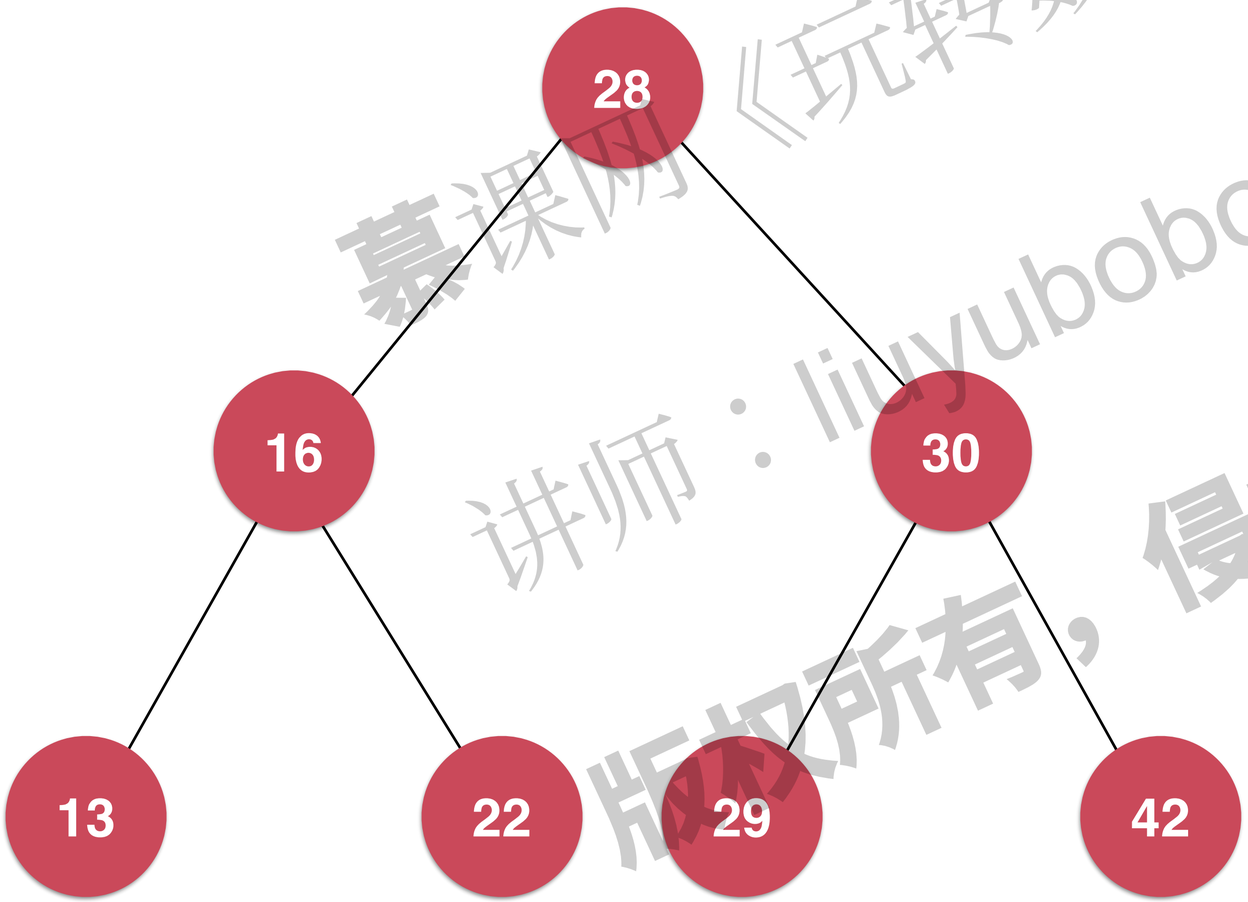
  二分搜索树（Binary Search Tree）

  平衡二叉树：AVL；红黑树

  堆；并查集

  线段树；Trie (字典树，前缀树)

二叉树

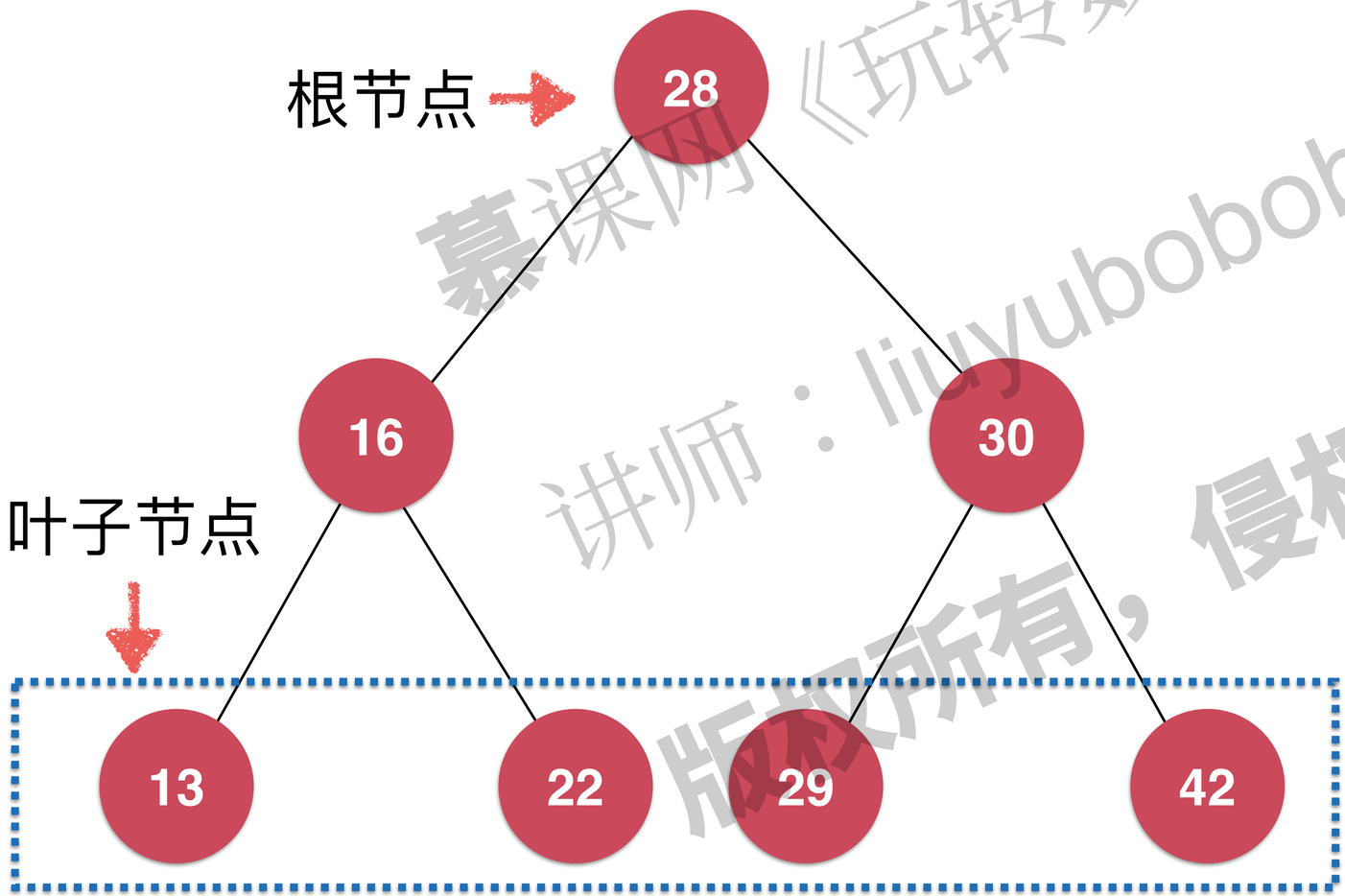


- 和链表一样，动态数据结构

```
class Node {
    E e;
    Node left;
    Node right;
}
```

- 二叉树（多叉树）

二叉树

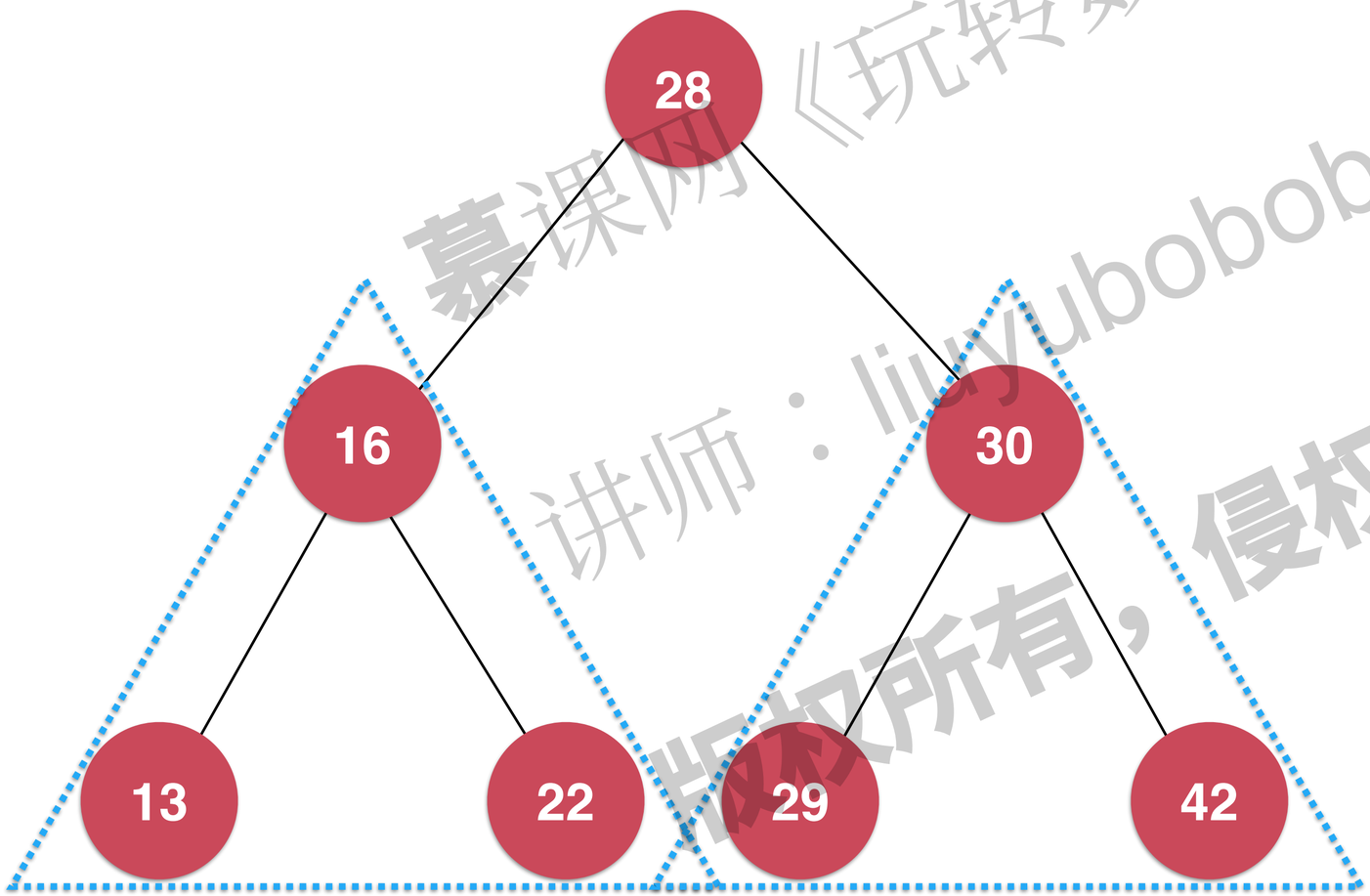


- 二叉树具有具有唯一根节点

```
class Node {
    E e;
    Node left;  // 左孩子
    Node right; // 右孩子
}
```

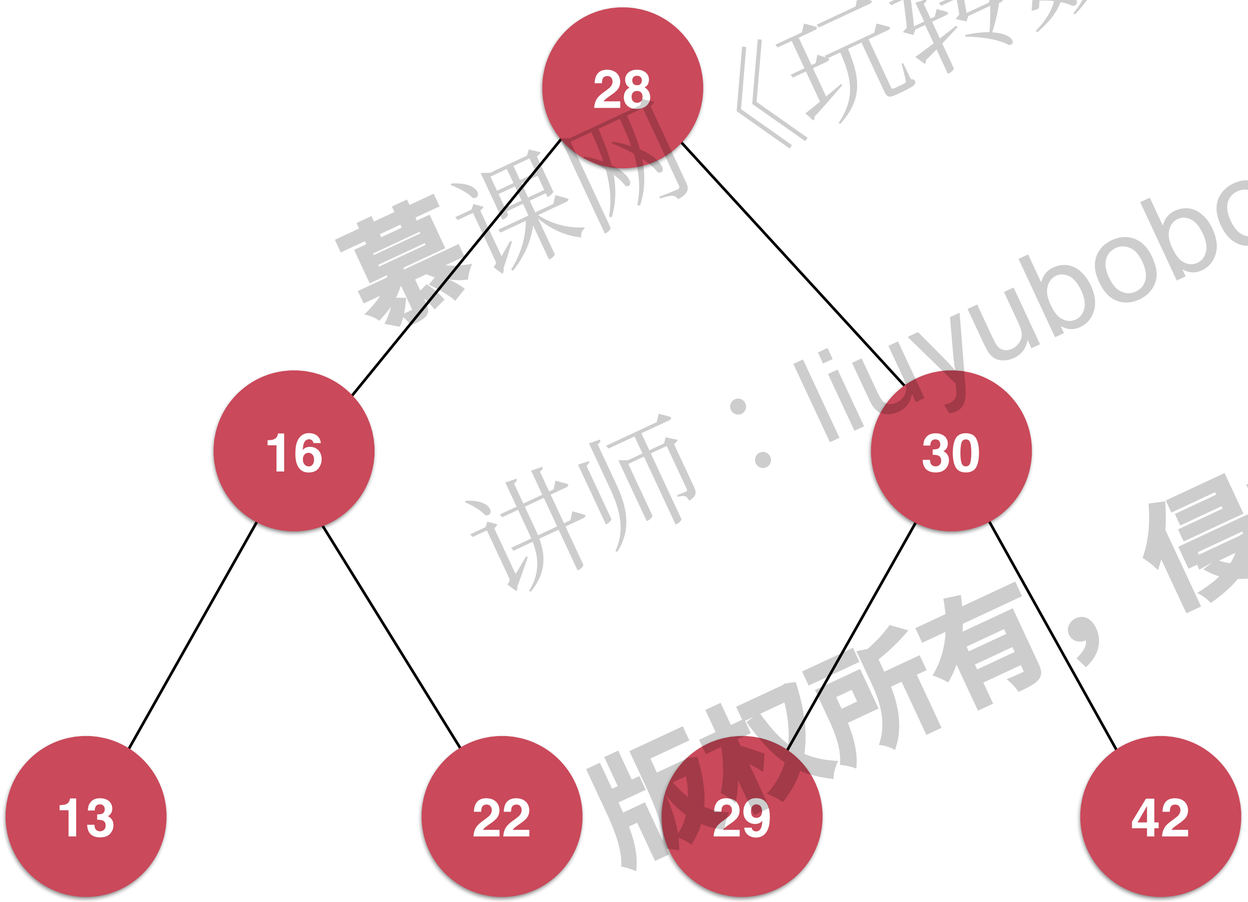
- 二叉树每个节点最多有两个孩子
- 二叉树每个节点最多有一个父亲

二叉树



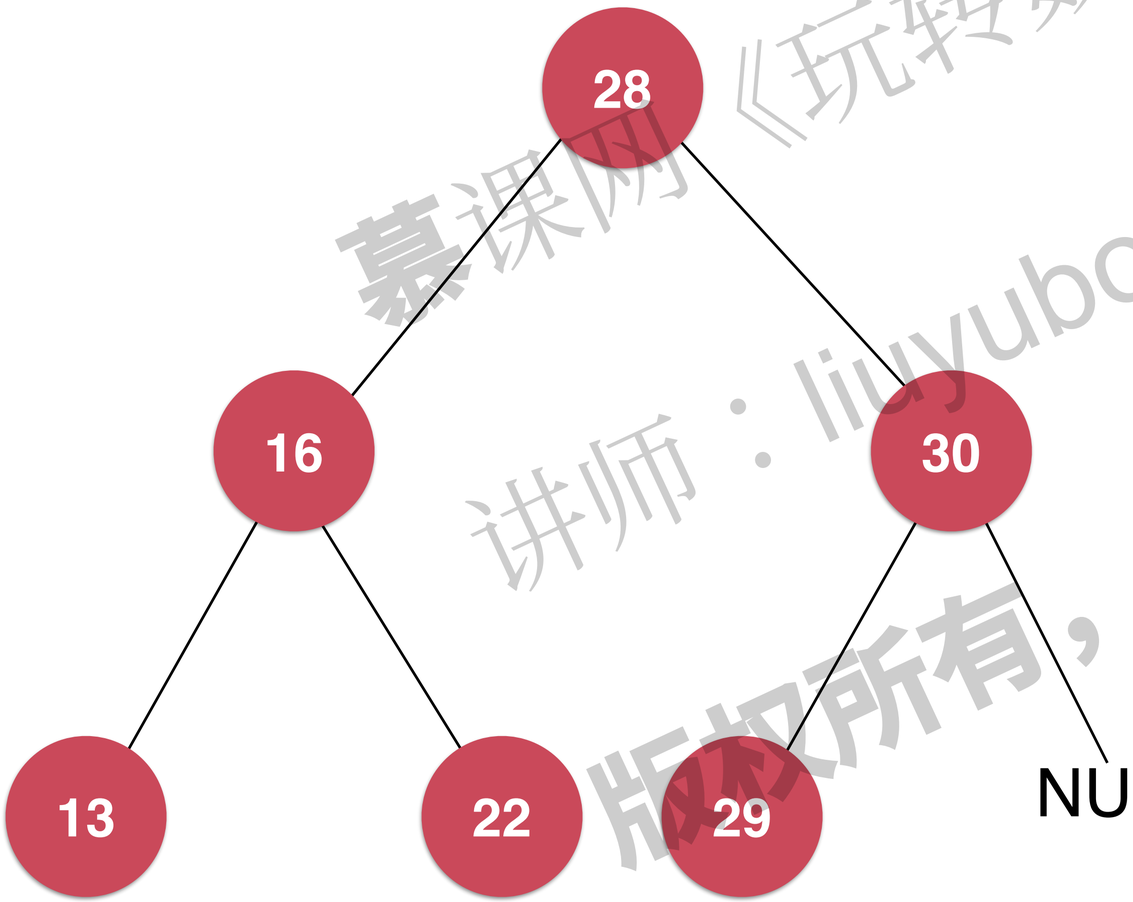
- 二叉树具有天然递归结构
  - 每个节点的左子树也是二叉树
  - 每个节点的右子树也是二叉树

二叉树



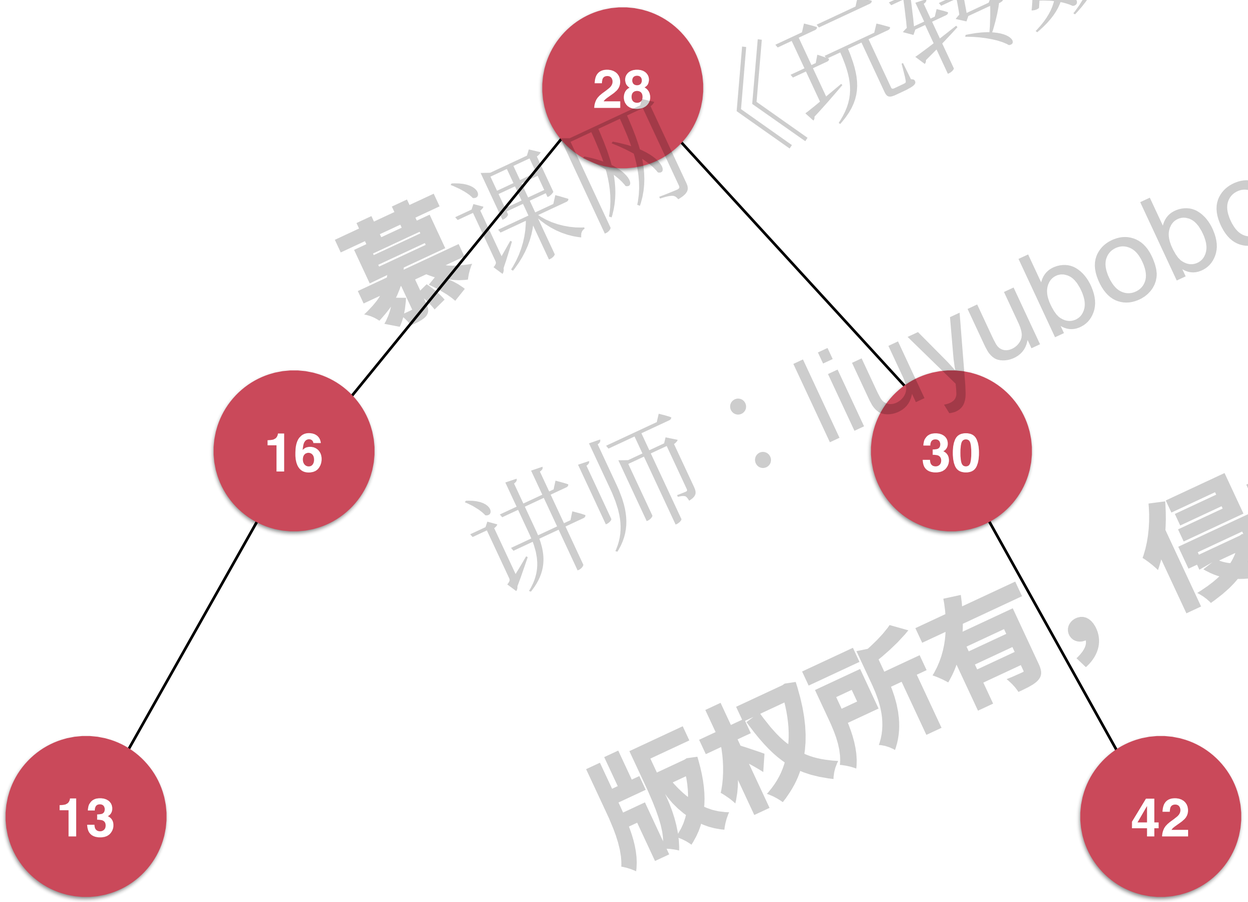
- 二叉树不一定是“满”的

二叉树



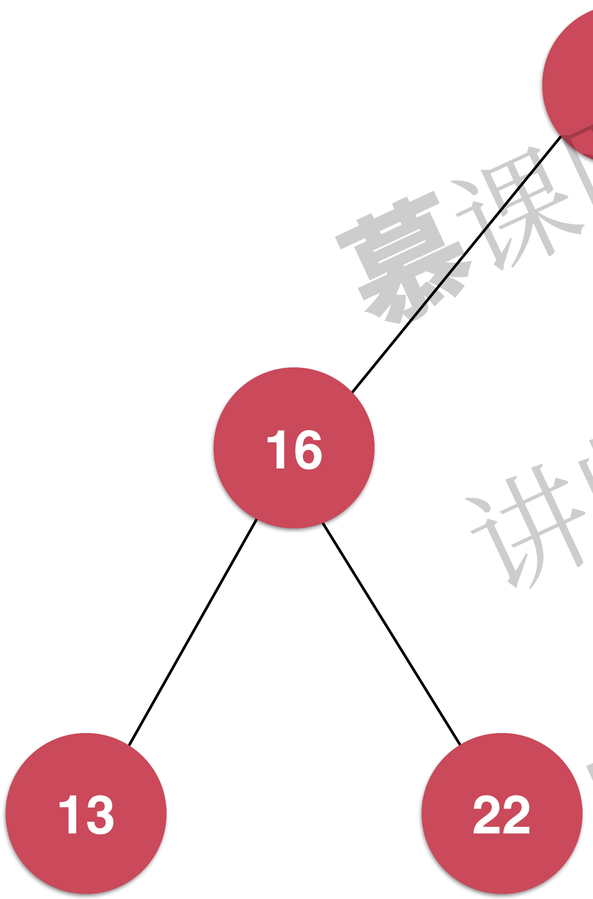
- 二叉树不一定是“满”的

二叉树



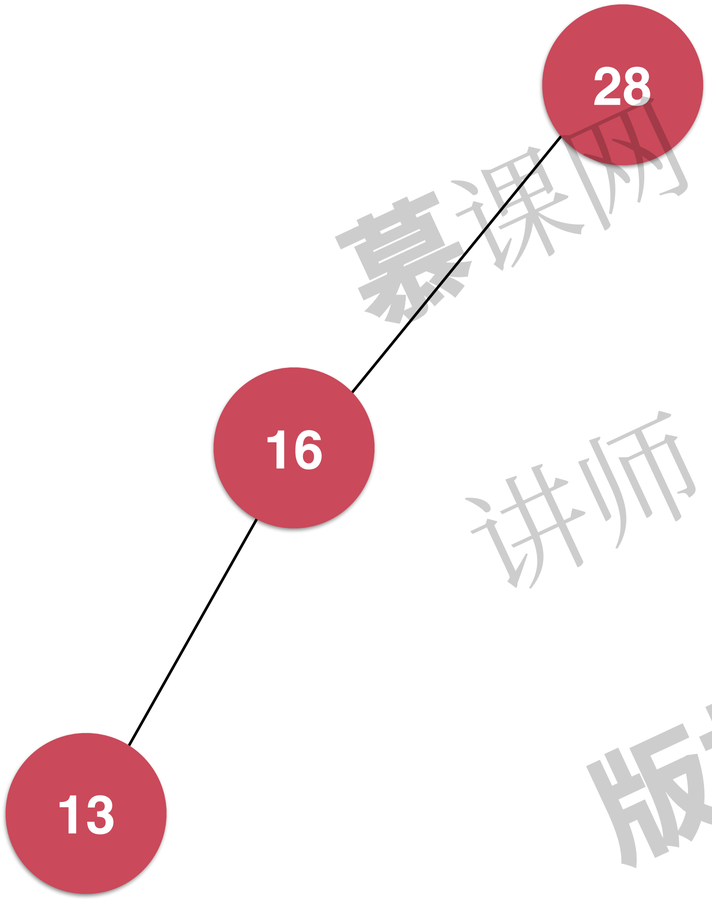
- 二叉树不一定是“满”的

二叉树



- 二叉树不一定是“满”的

二叉树



- 二叉树不一定是“满”的

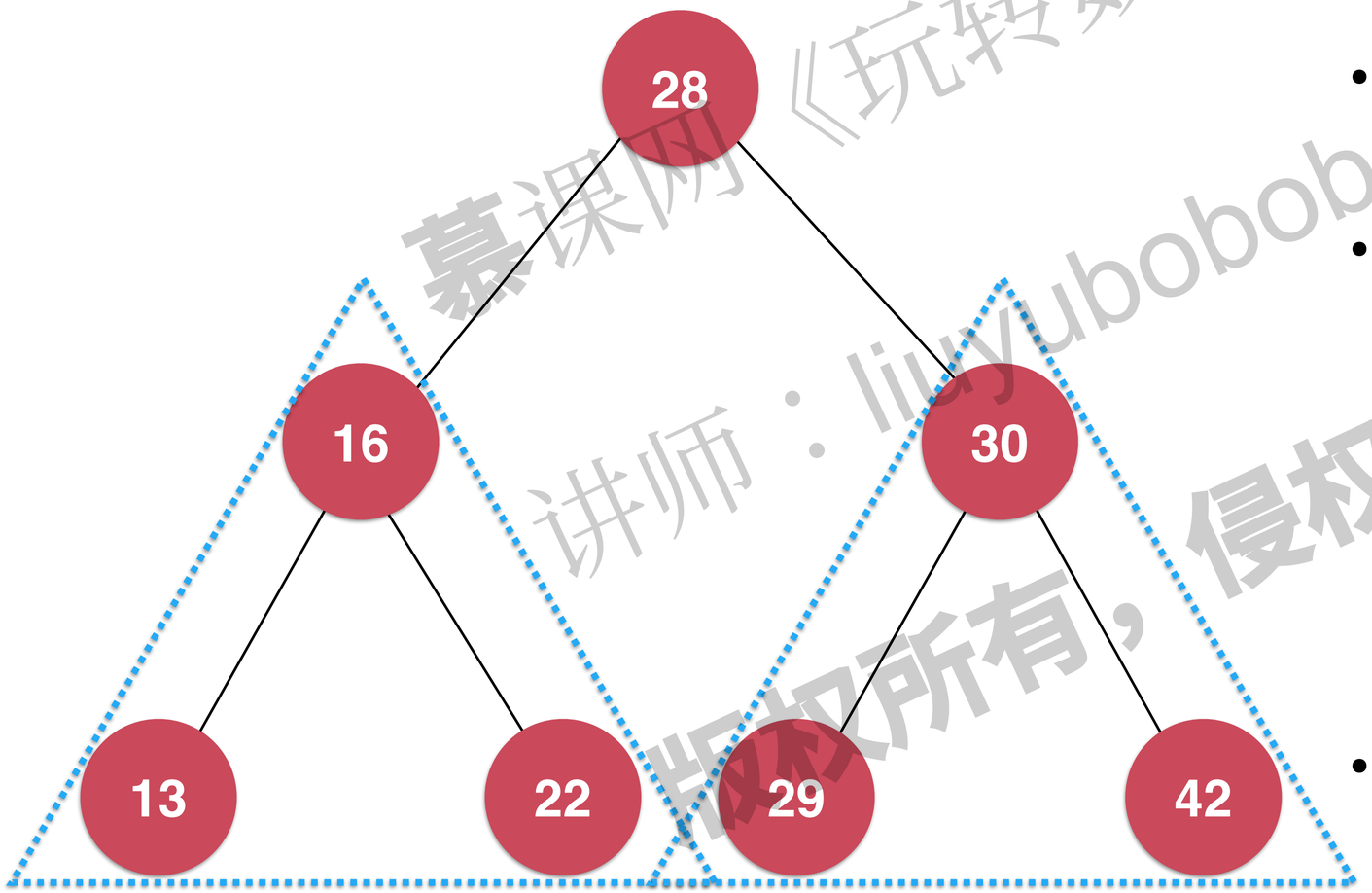
二叉树

- 二叉树不一定是“满”的

一个节点也是二叉树

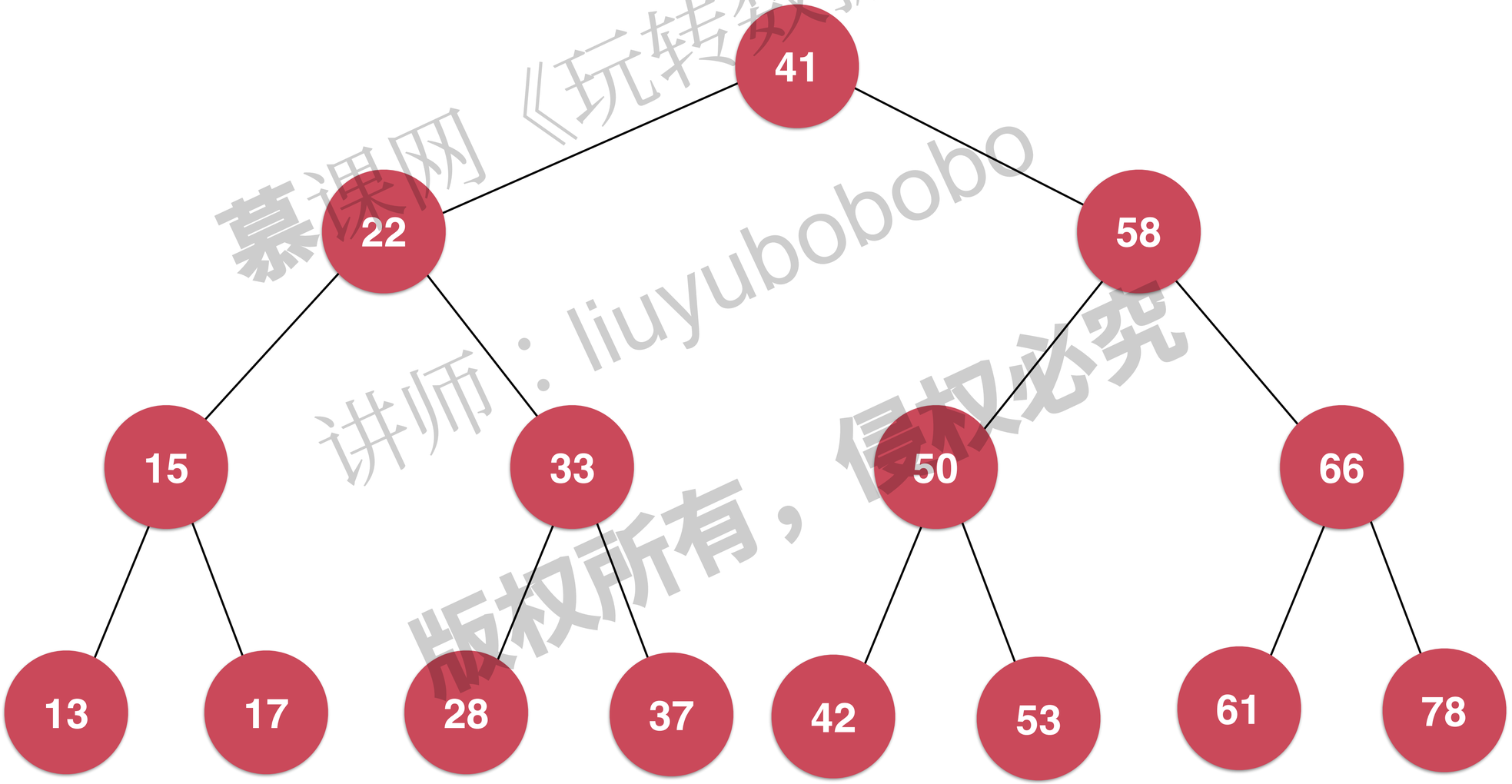
NULL 空也是二叉树

二分搜索树 Binary Search Tree

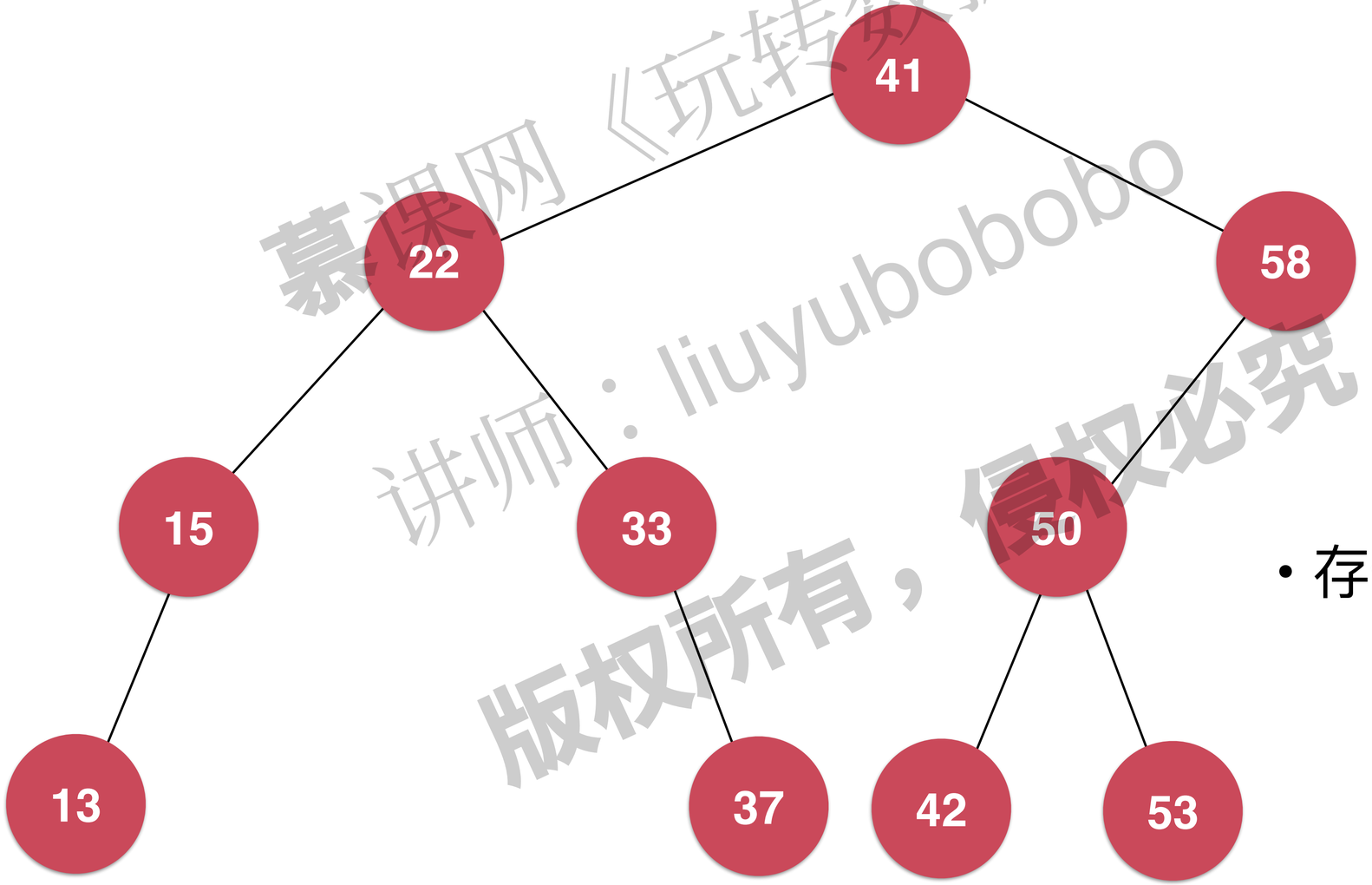


- 二分搜索树是二叉树
- 二分搜索树的每个节点的值：
  - 大于其左子树的所有节点的值
  - 小于其右子树的所有节点的值
- 每一棵子树也是二分搜索树

二分搜索树 Binary Search Tree



二分搜索树 Binary Search Tree



- 存储的元素必须有可比较性

实践：二分搜索树基础结构

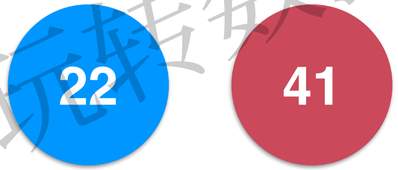
二分搜索树添加新元素

二分搜索树添加新元素

二分搜索树添加新元素

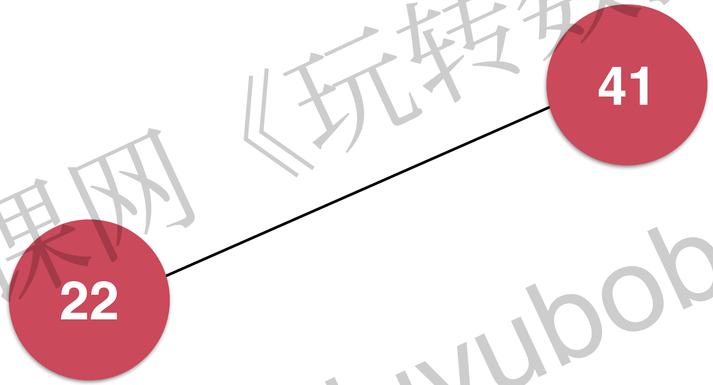
41

二分搜索树添加新元素

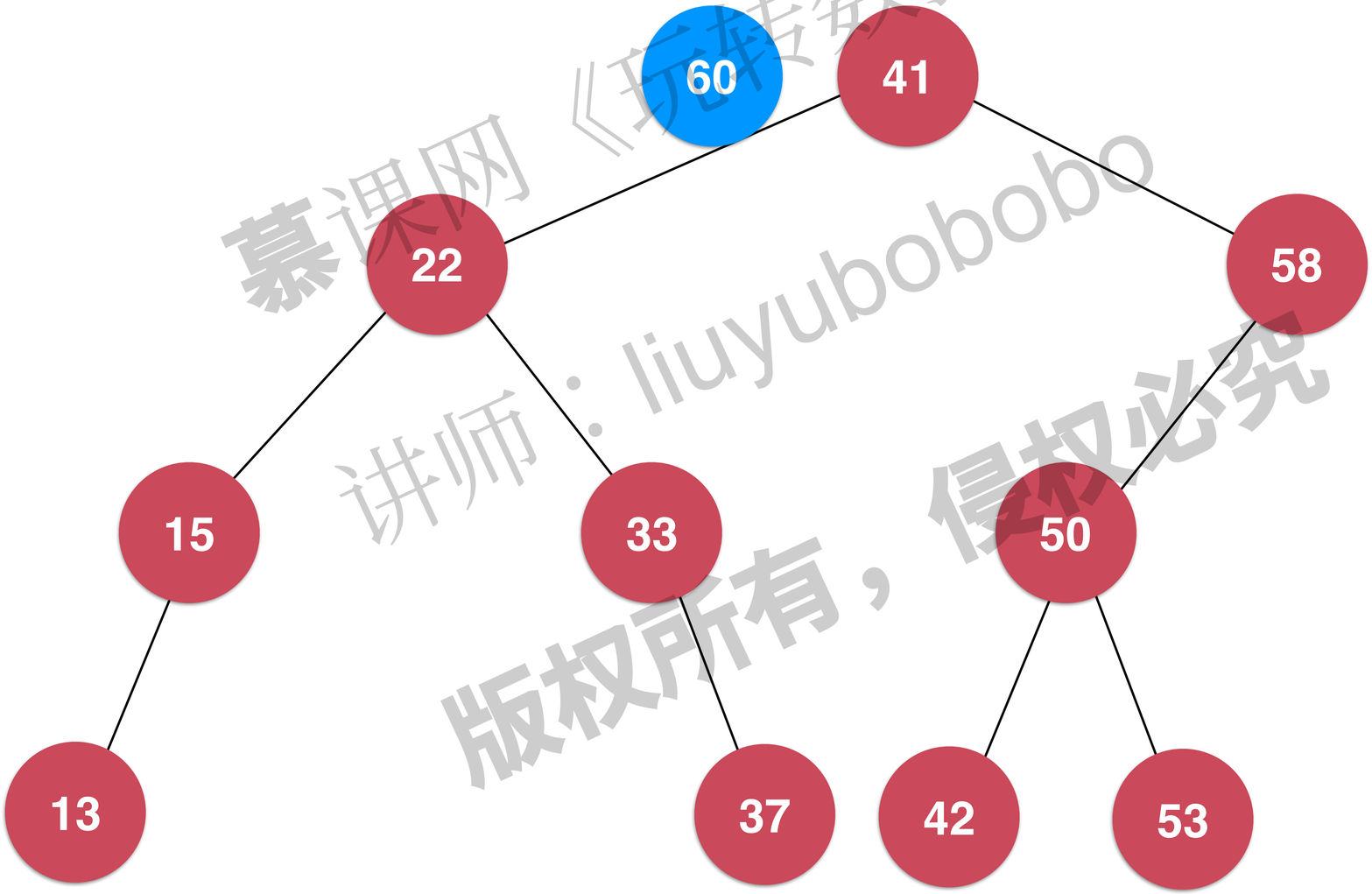


二分搜索树添加新元素

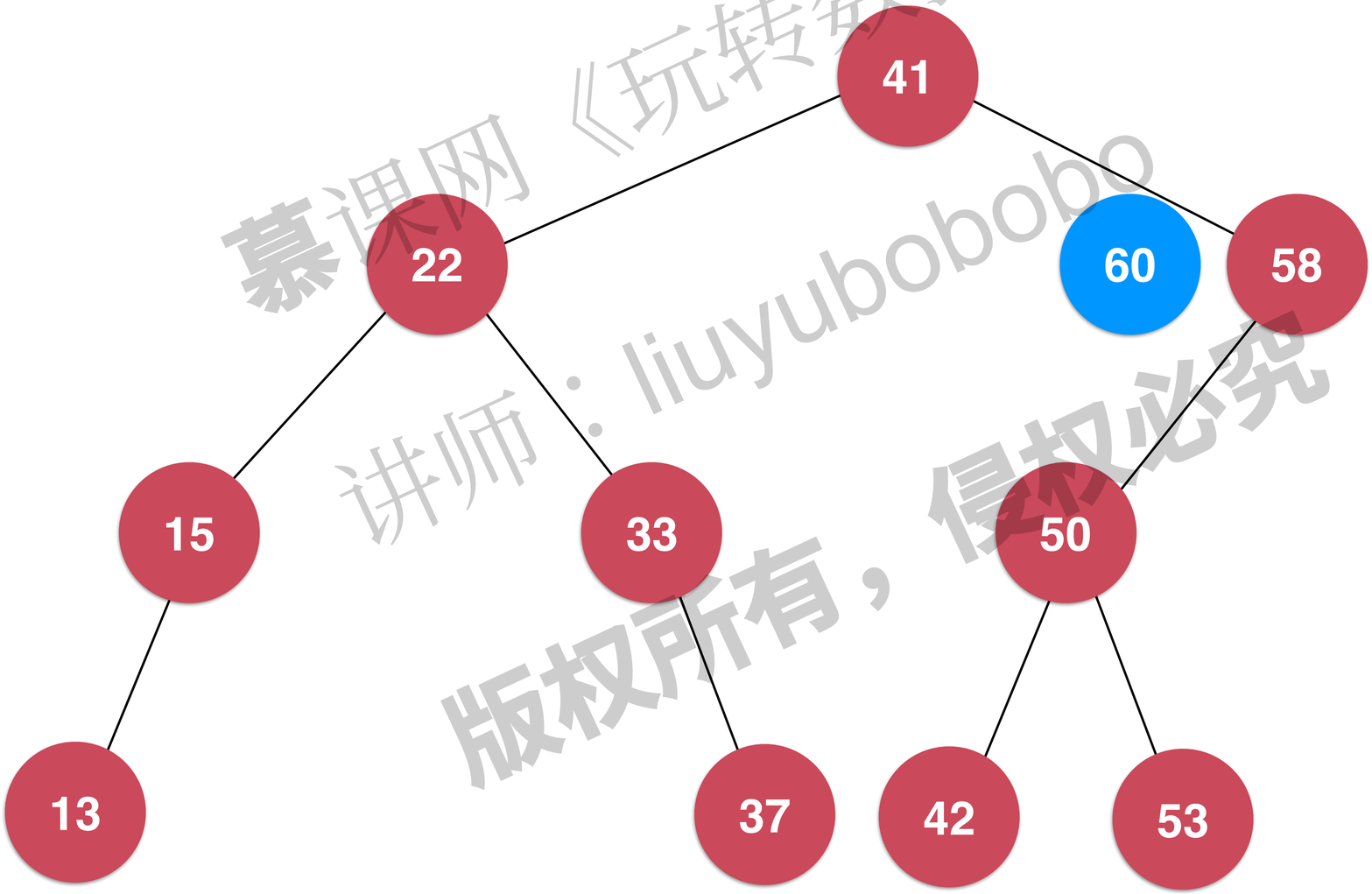
二分搜索树添加新元素



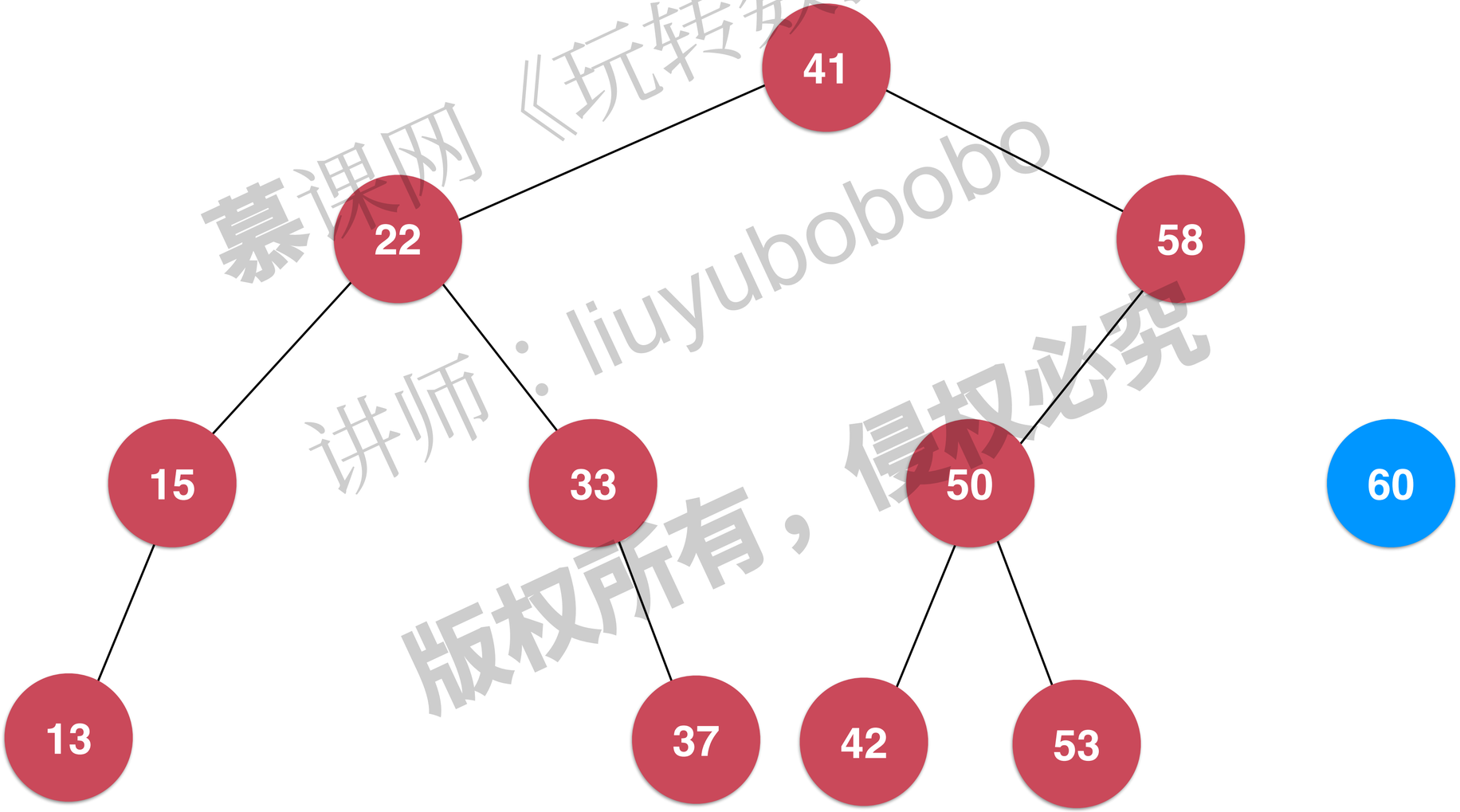
二分搜索树添加新元素



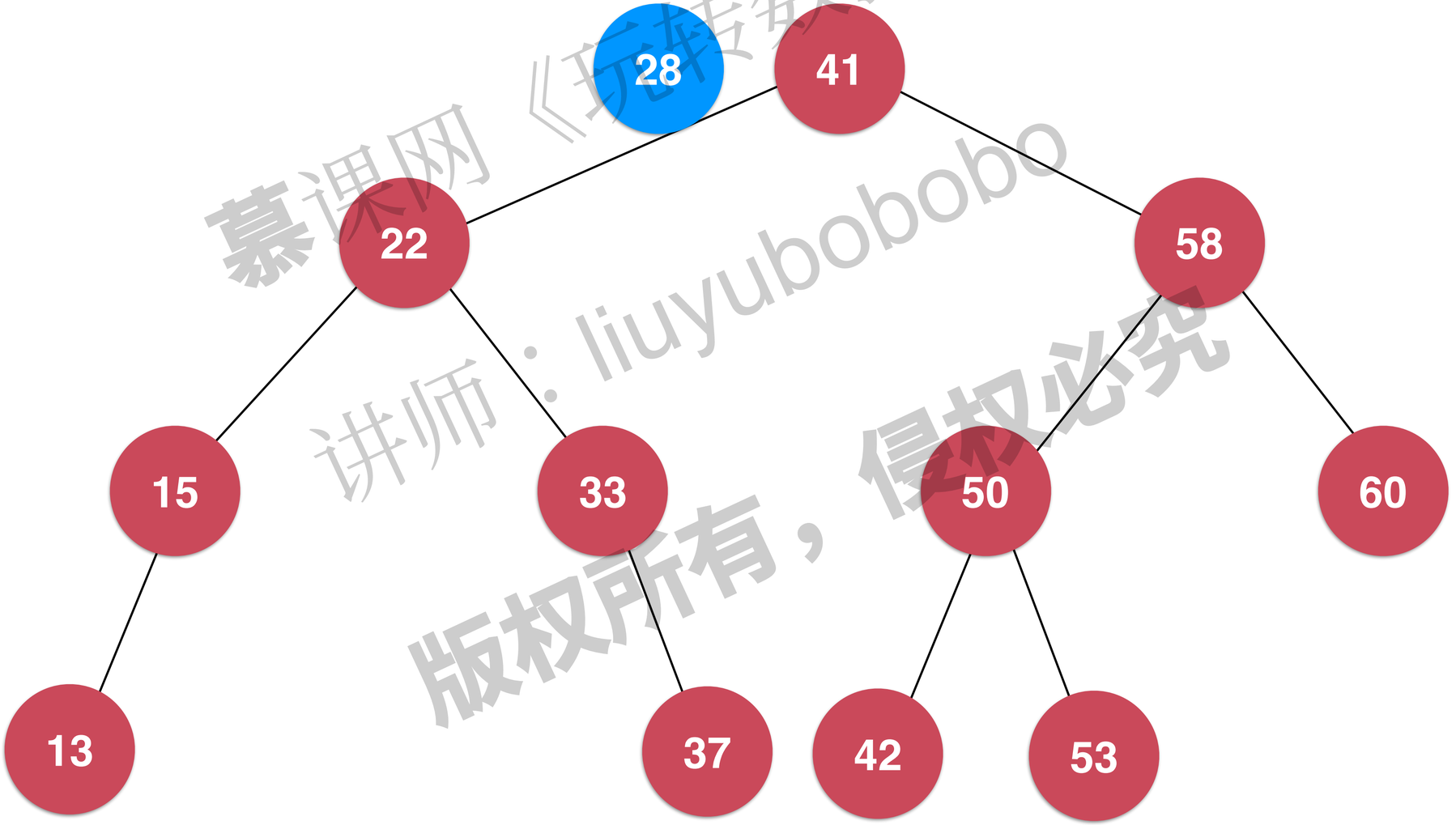
二分搜索树添加新元素



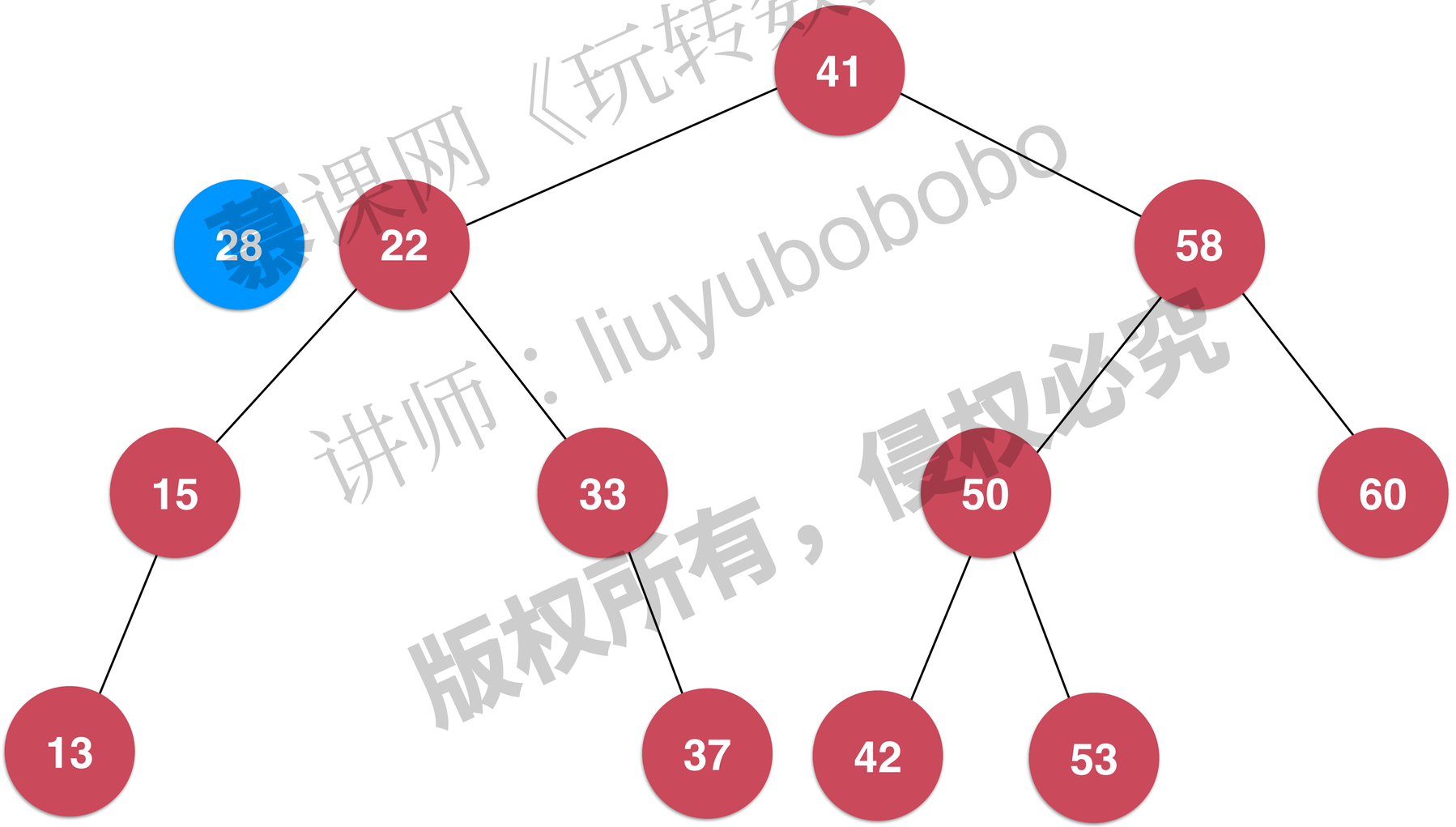
二分搜索树添加新元素



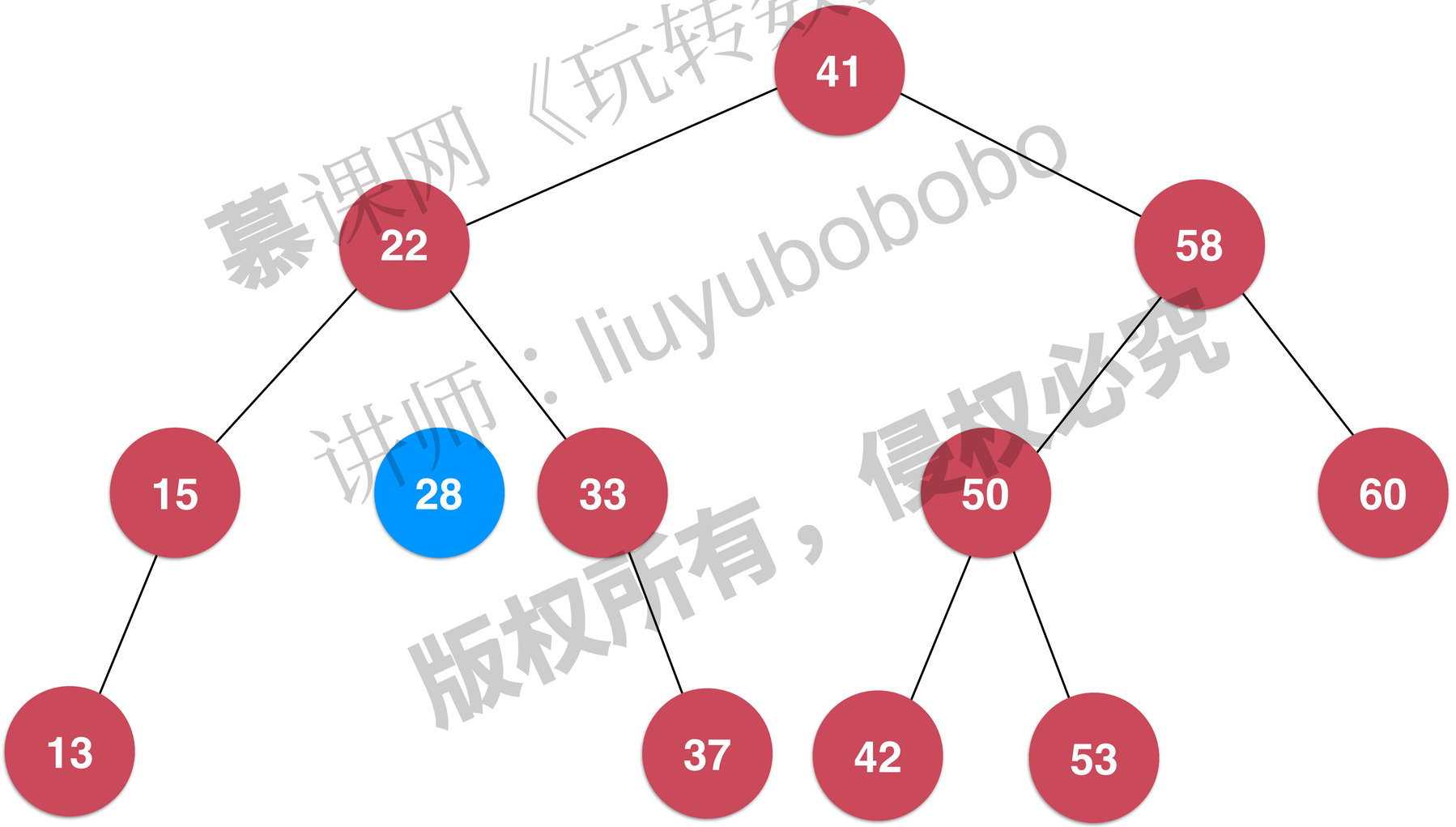
二分搜索树添加新元素



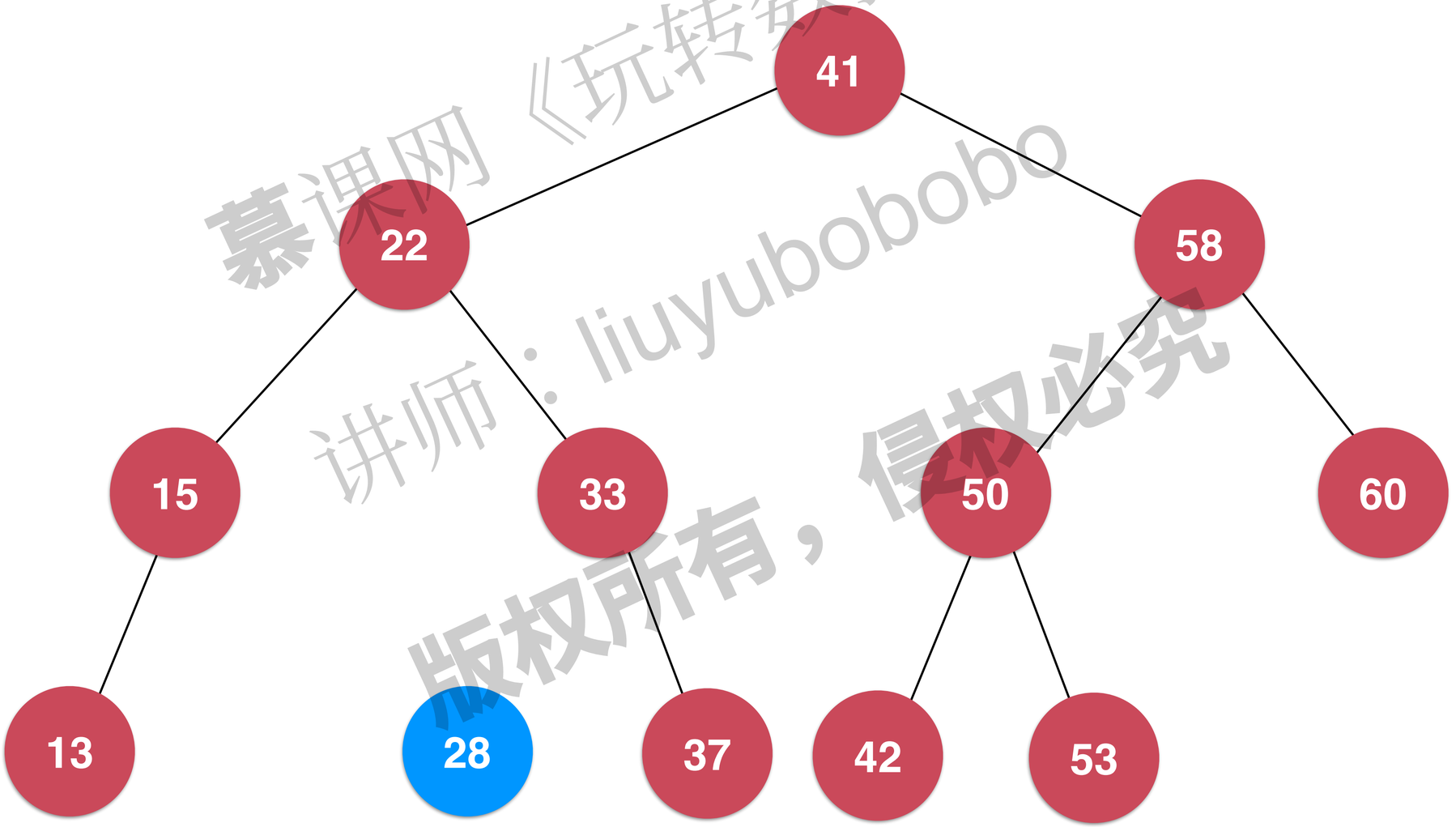
二分搜索树添加新元素



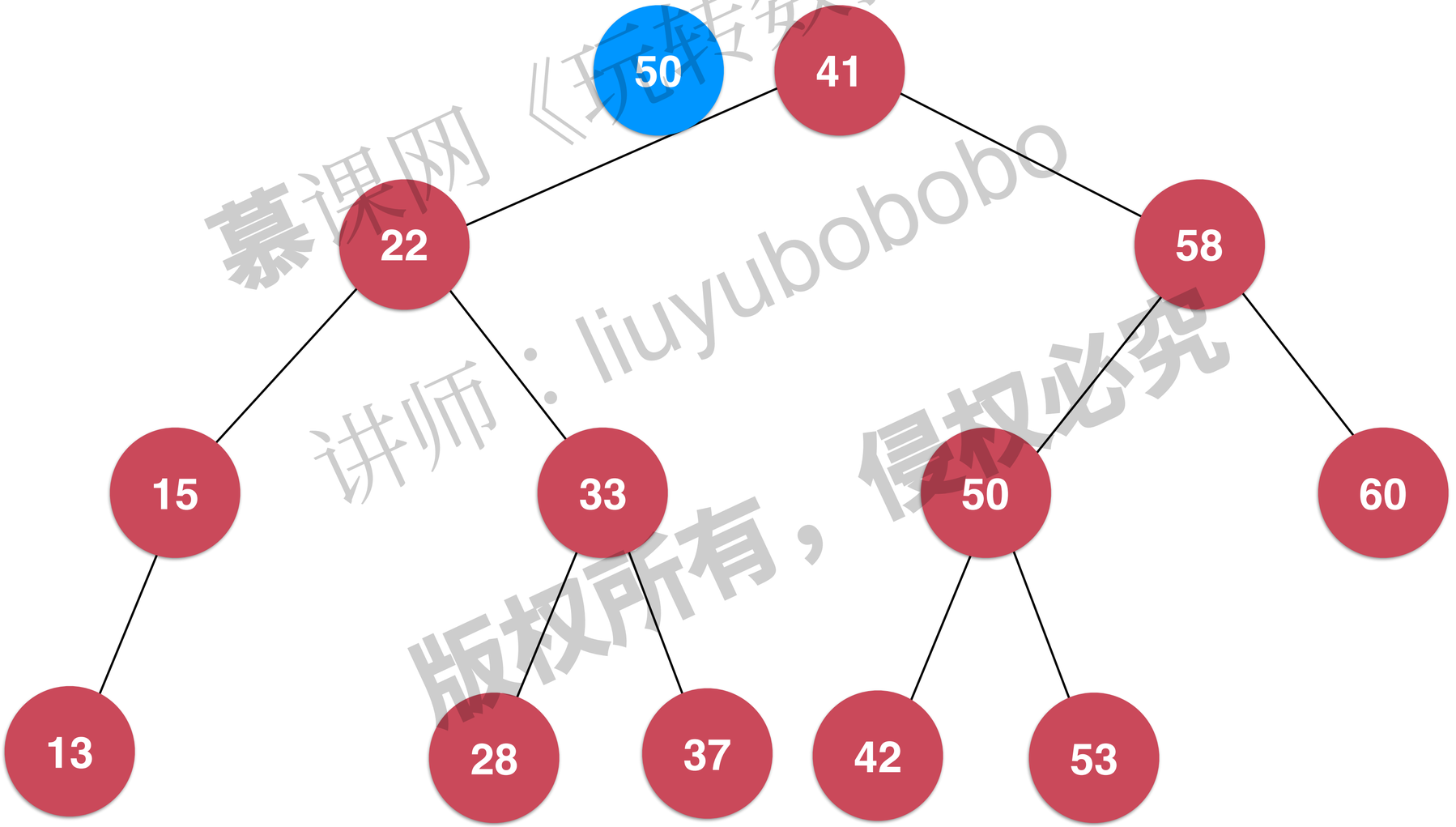
二分搜索树添加新元素



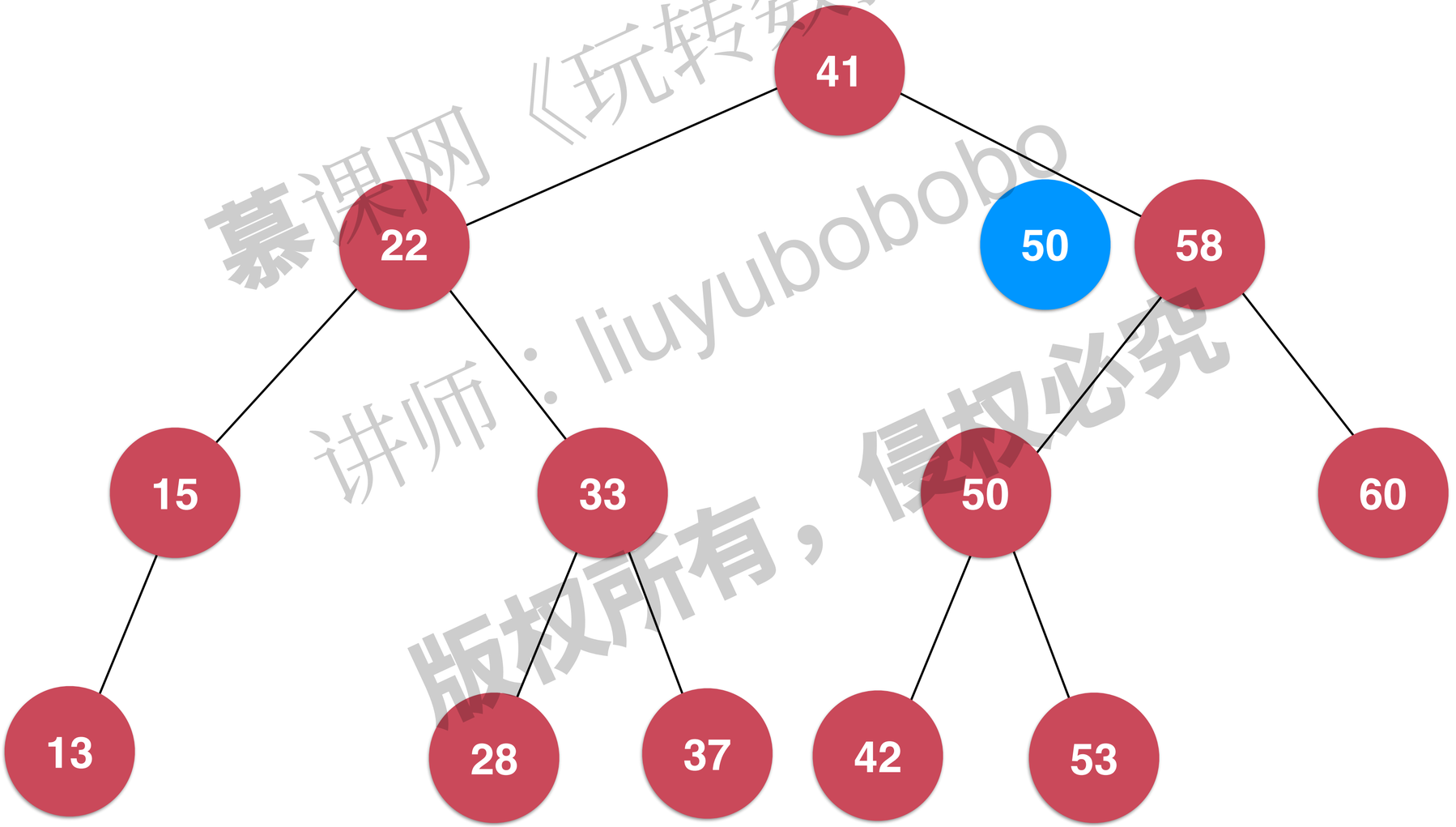
二分搜索树添加新元素



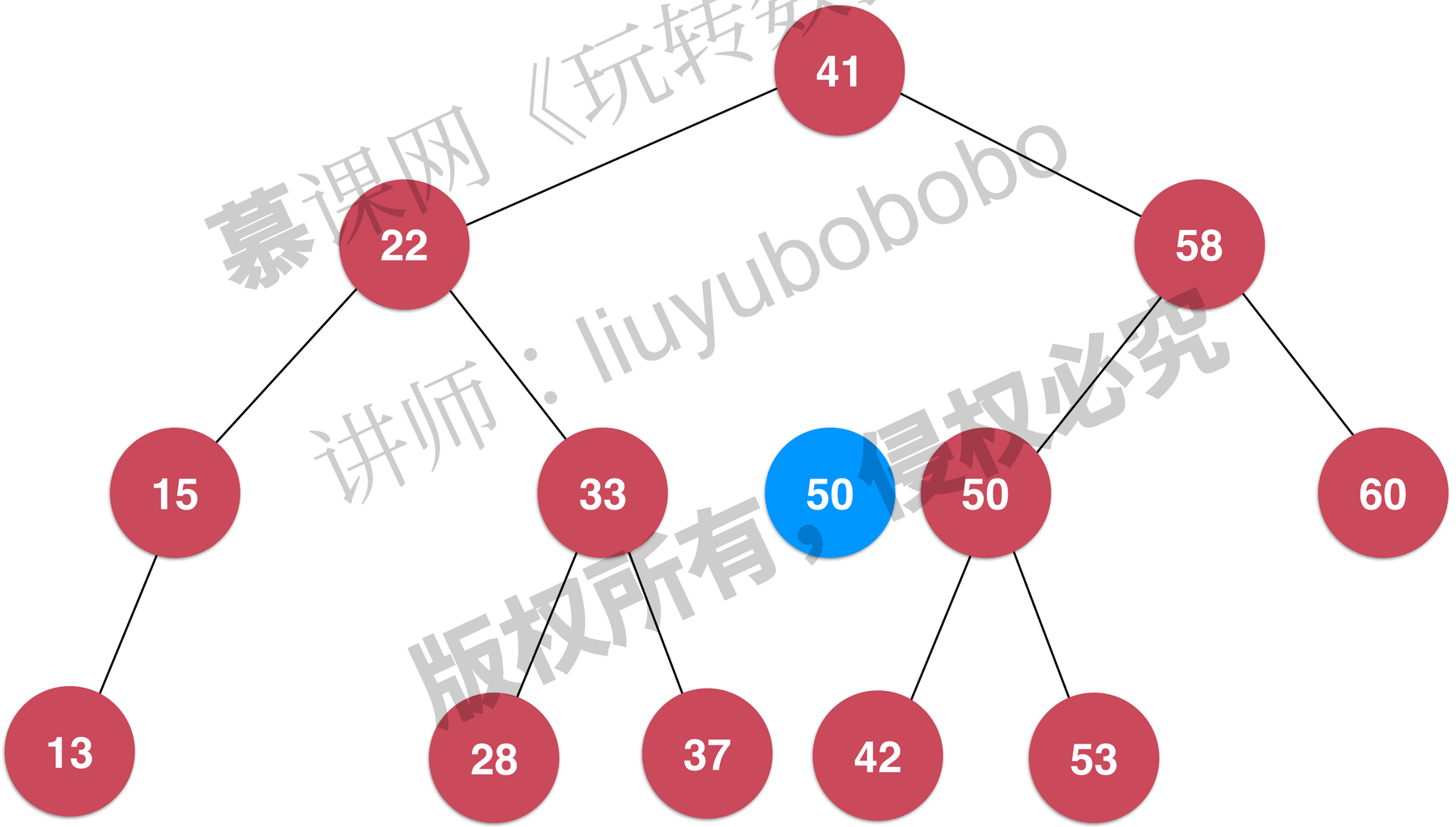
二分搜索树添加新元素



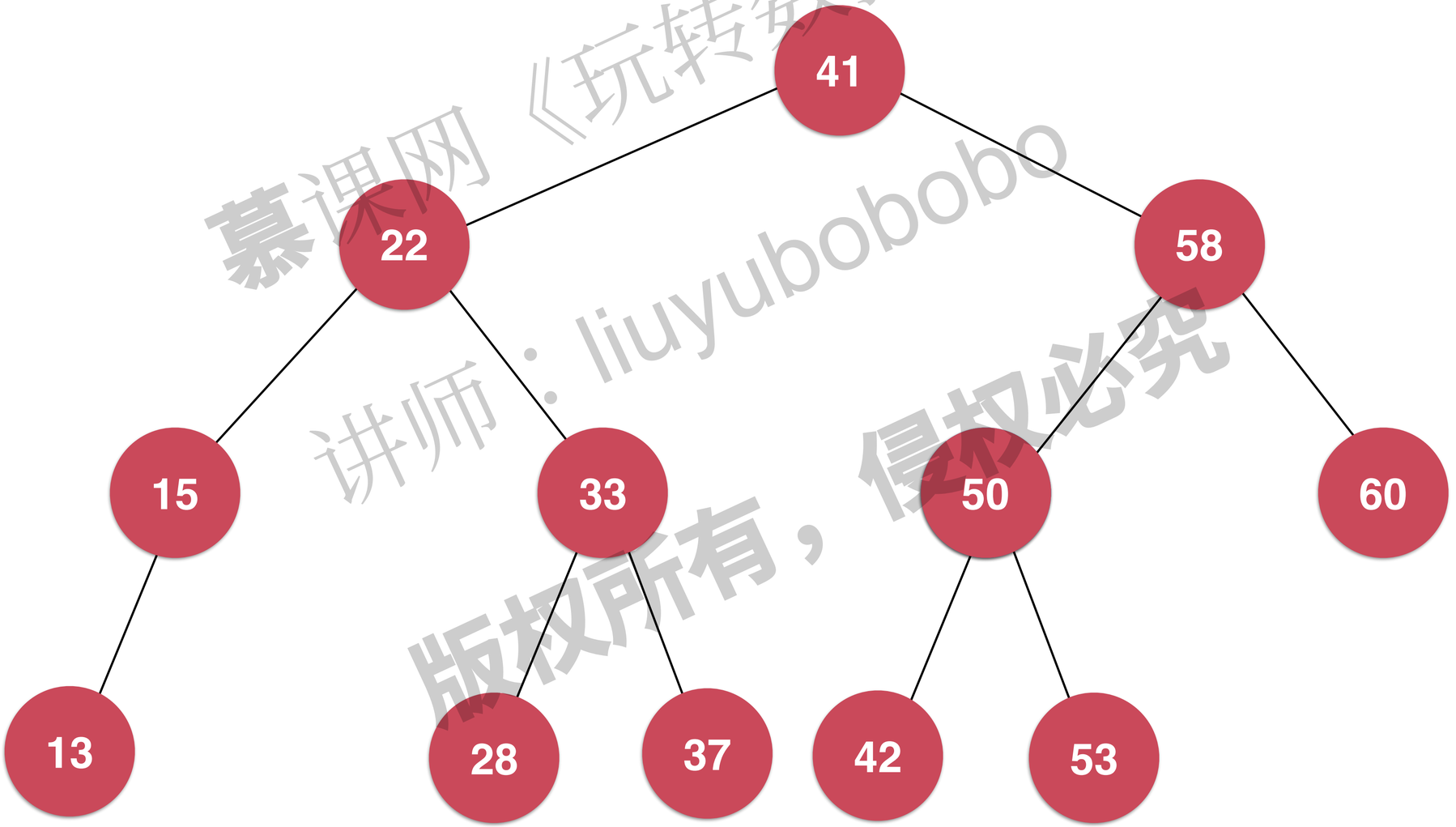
二分搜索树添加新元素



二分搜索树添加新元素



二分搜索树添加新元素



二分搜索树添加新元素

- 我们的二分搜索树不包含重复元素

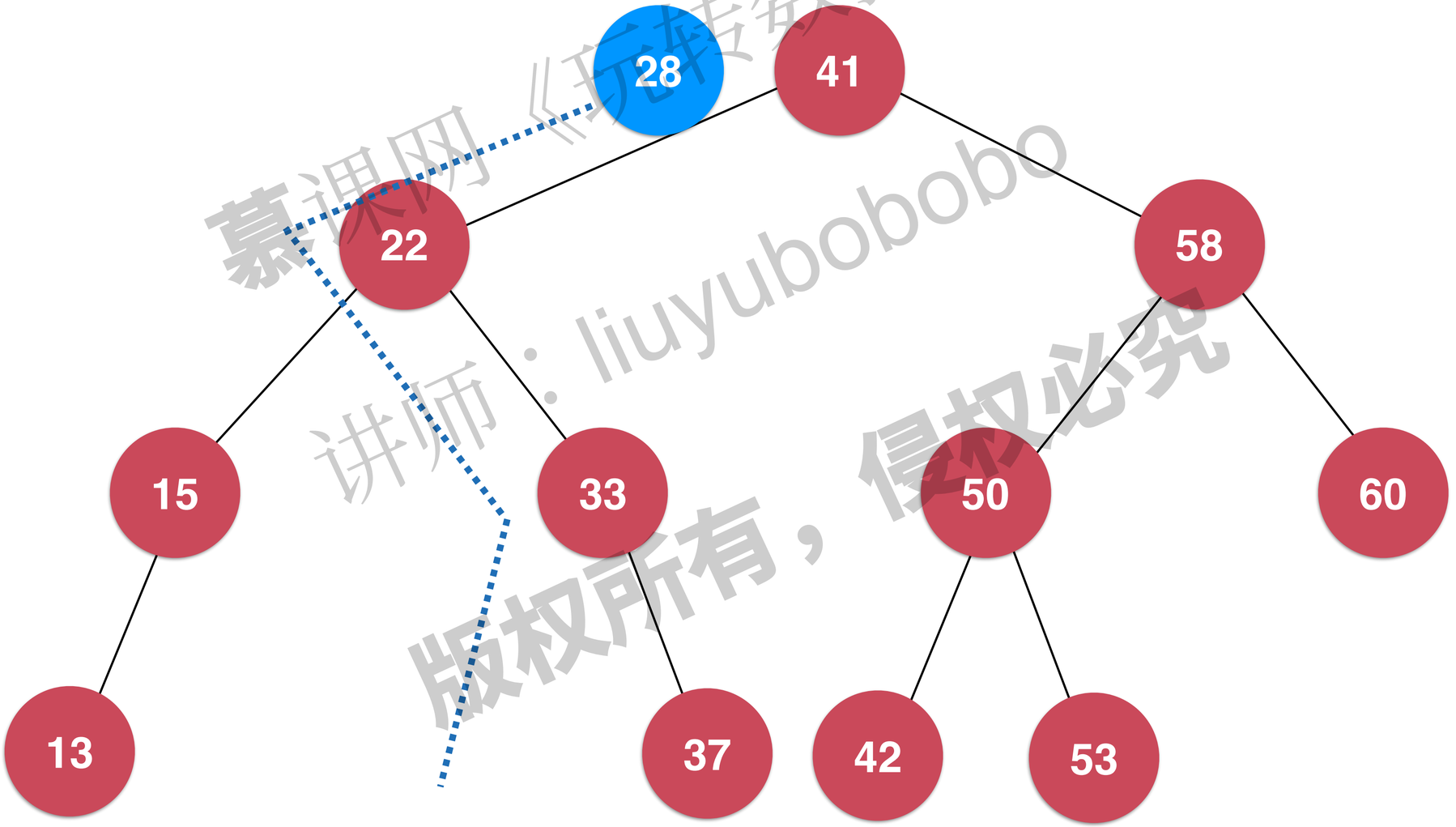
  如果想包含重复元素的话，只需要定义：

  左子树小于等于节点；或者右子树大于等于节点

  注意：我们之前讲的数组和链表，可以有重复元素

- 二分搜索树添加元素的非递归写法，和链表很像

二分搜索树添加新元素



二分搜索树添加新元素

- 二分搜索树添加元素的非递归写法，和链表很像
- 这个课程在二分搜索树方面的实现，关注递归实现
- 二分搜索树一些方法的非递归实现，留做练习
- 在二分搜索树方面，递归比非递归实现简单：）

实践：二分搜索树添加新元素

递归的终止条件

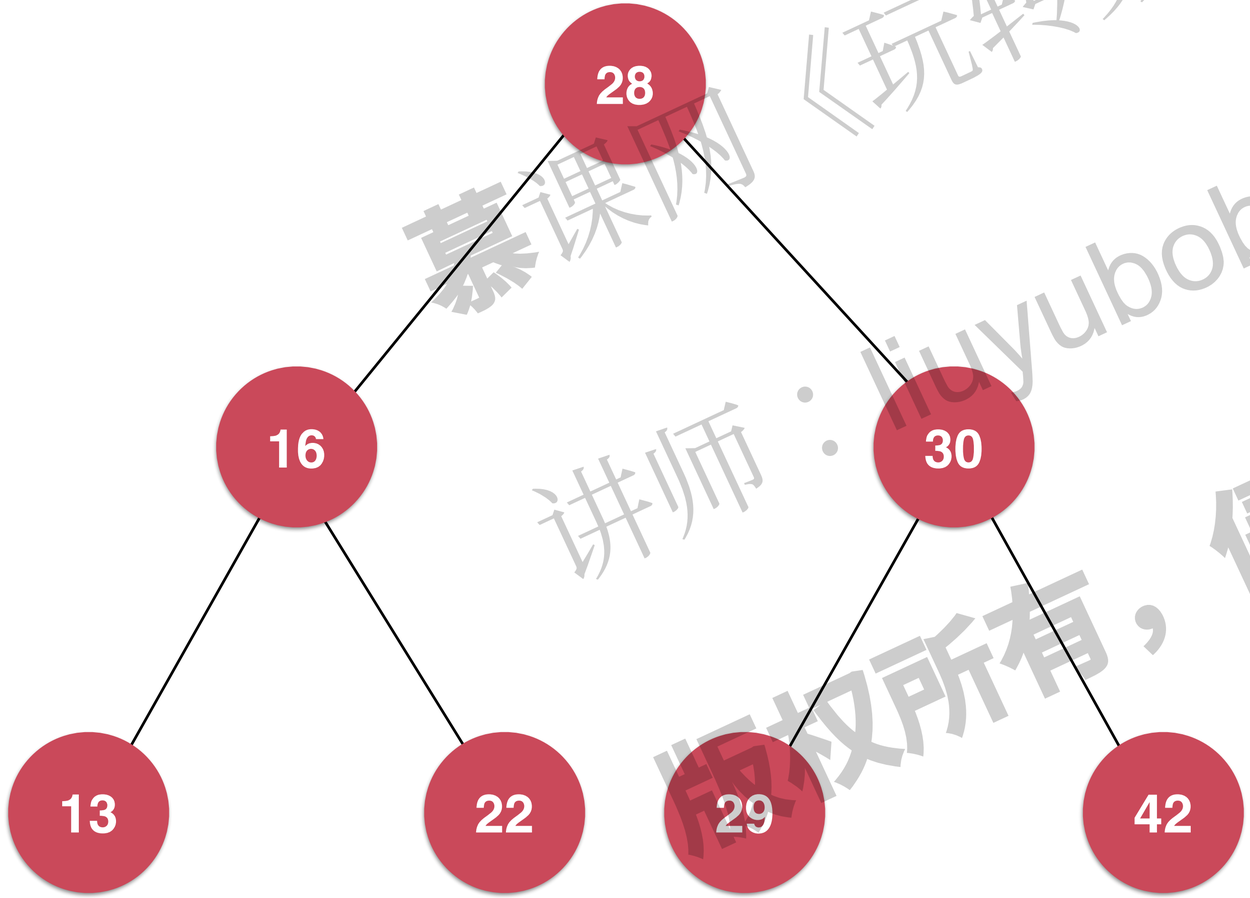
实践：另一种方法递归实现二分搜索树添加新元素

二分搜索树的查询

实践：二分搜索树的查询

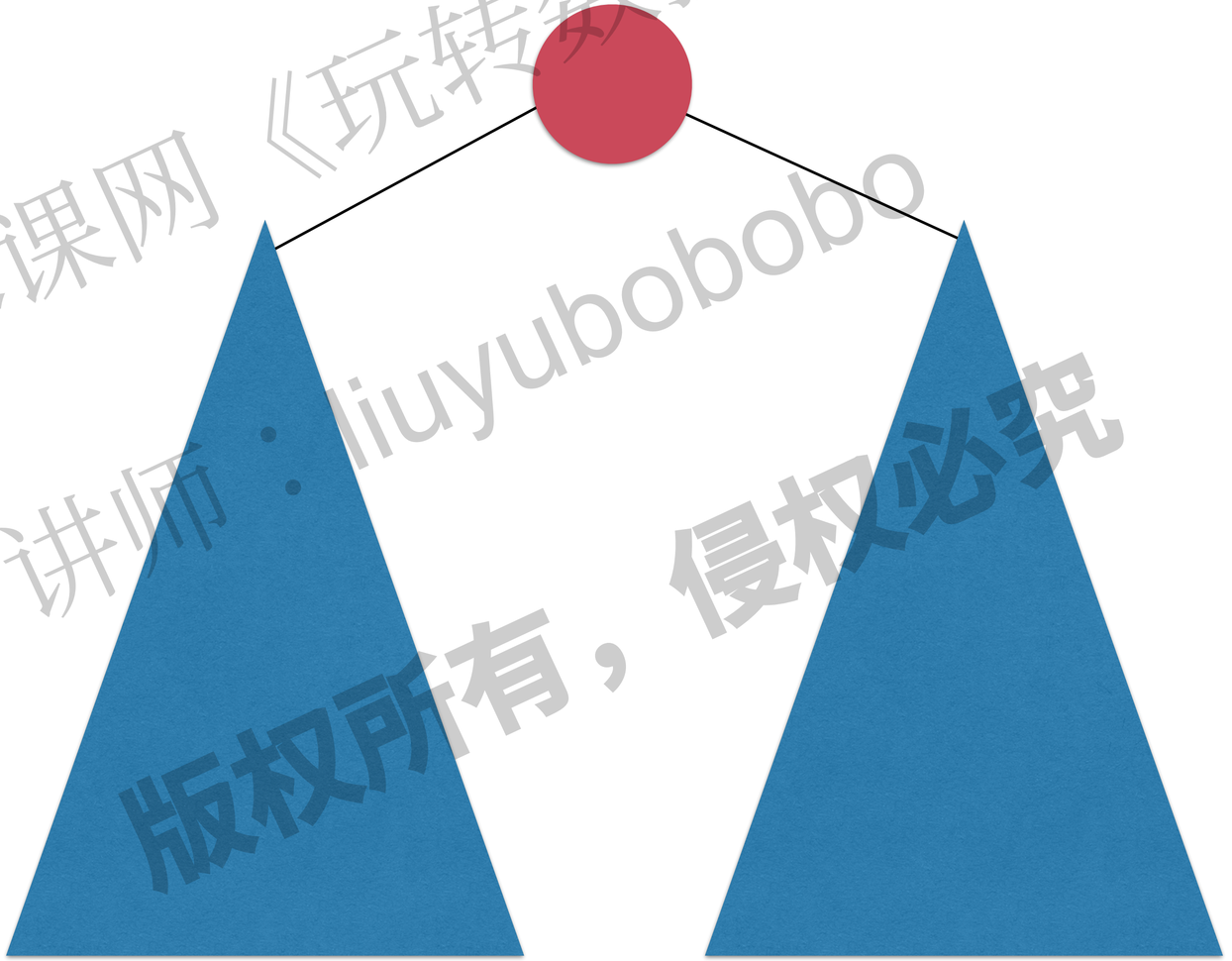
二分搜索树的遍历

什么是遍历操作

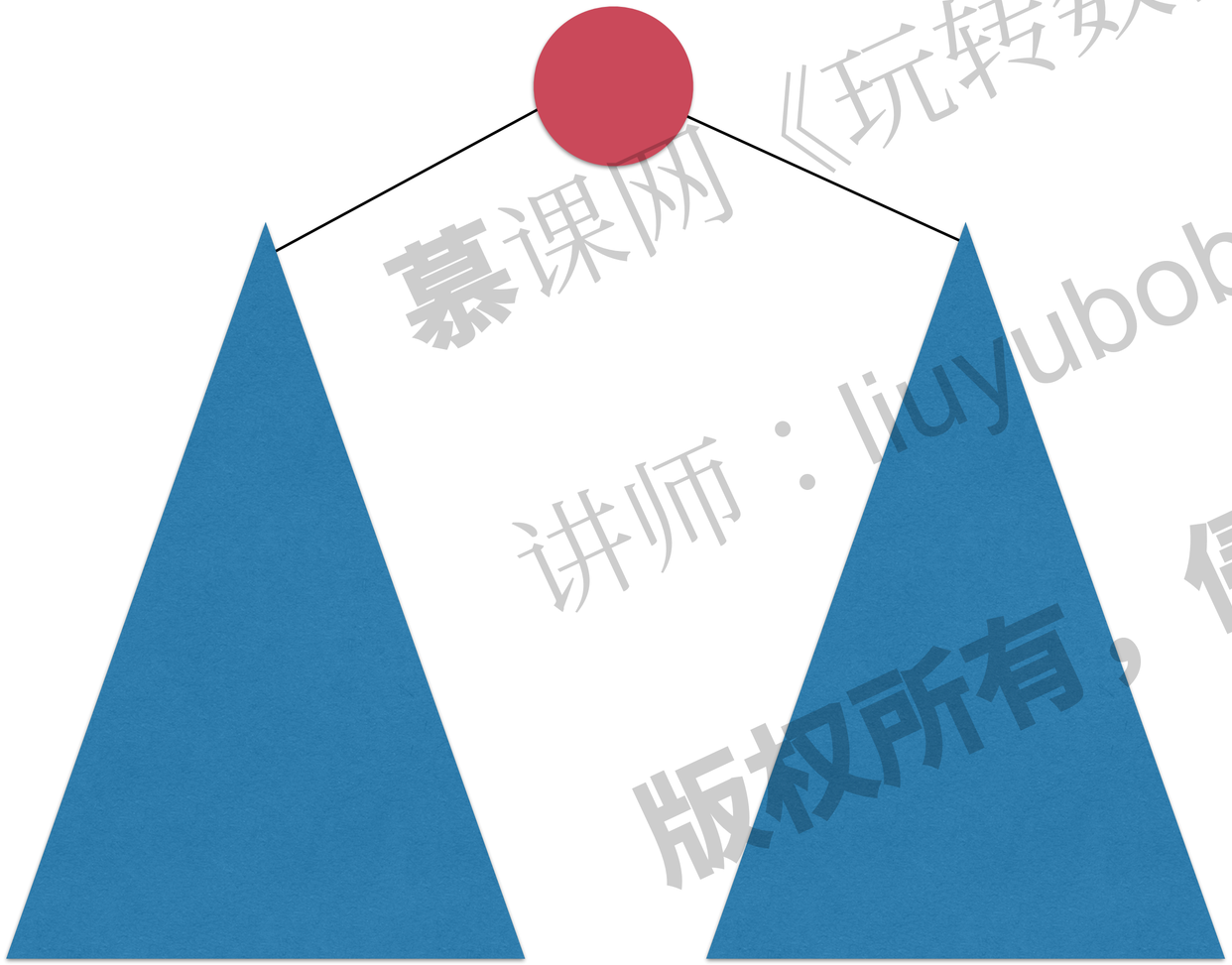


- 遍历操作就是把所有节点都访问一遍
- 访问的原因和业务相关
- 在线性结构下，遍历是极其容易的
- 在树结构下，也没那么难：）

二分搜索树的递归操作



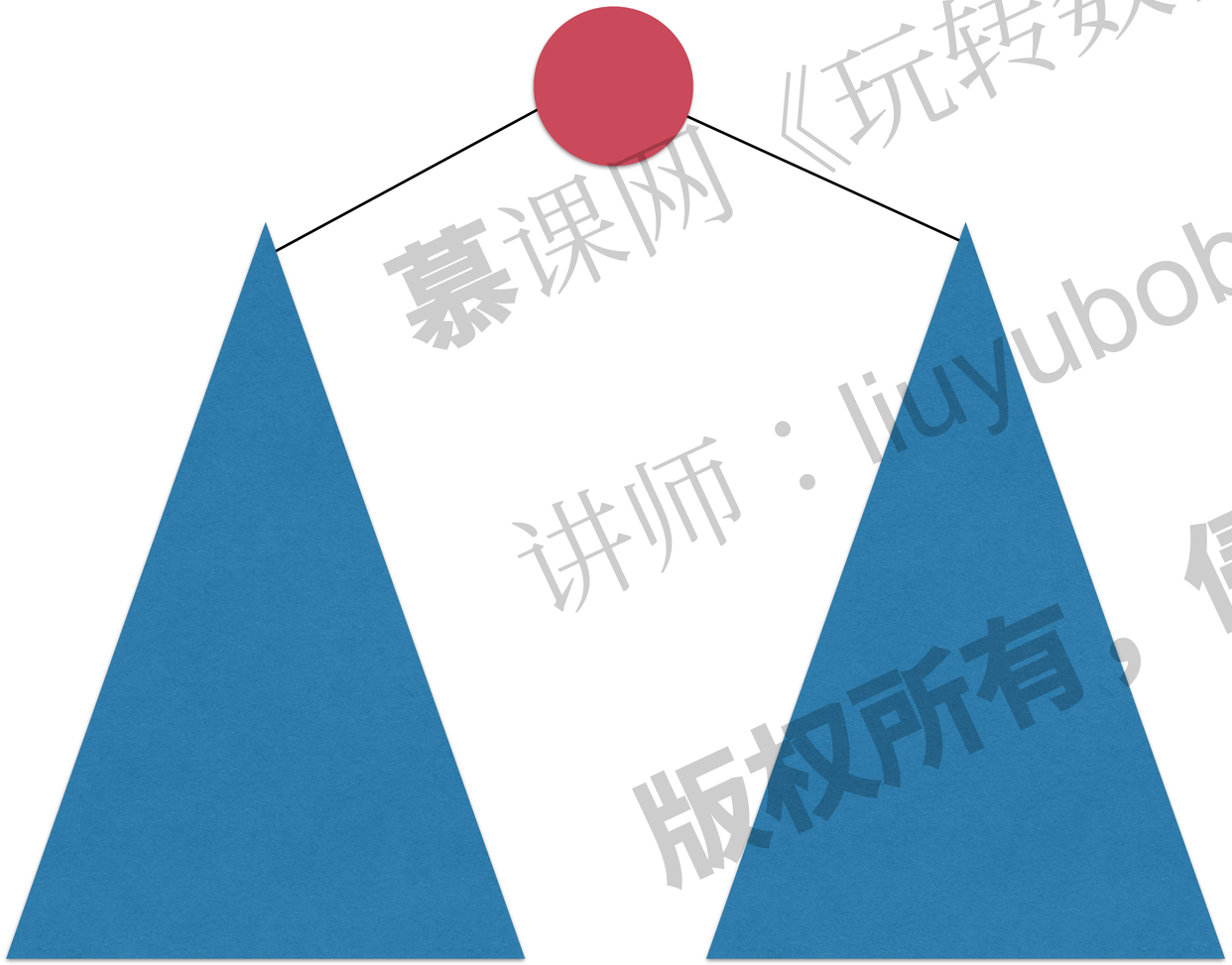
二分搜索树的递归操作



- 对于遍历操作，两棵子树都要顾及

```
function traverse(node):
    if(node == null)
        return;

访问该节点

traverse(node.left)
traverse(node.right)
```

二分搜索树的前序遍历



- 对于遍历操作，两棵子树都要顾及

```
function traverse(node):
    if(node == null)
        return;

访问该节点

traverse(node.left)
traverse(node.right)
```

实践：二分搜索树的前序遍历

实践：二分搜索树的toString

二分搜索树的前中后序遍历

前序遍历

```
function traverse(node):
    if(node == null)
        return;

    访问该节点
    traverse(node.left)
    traverse(node.right)
```

前序遍历

```
function traverse(node):
    if(node == null)
        return;

    访问该节点
    traverse(node.left)
    traverse(node.right)
```

前序遍历

```
function traverse(node):
    if(node == null)
        return;

    访问该节点

    traverse(node.left)
    traverse(node.right)
```

- 最自然的遍历方式
- 最常用的遍历方式

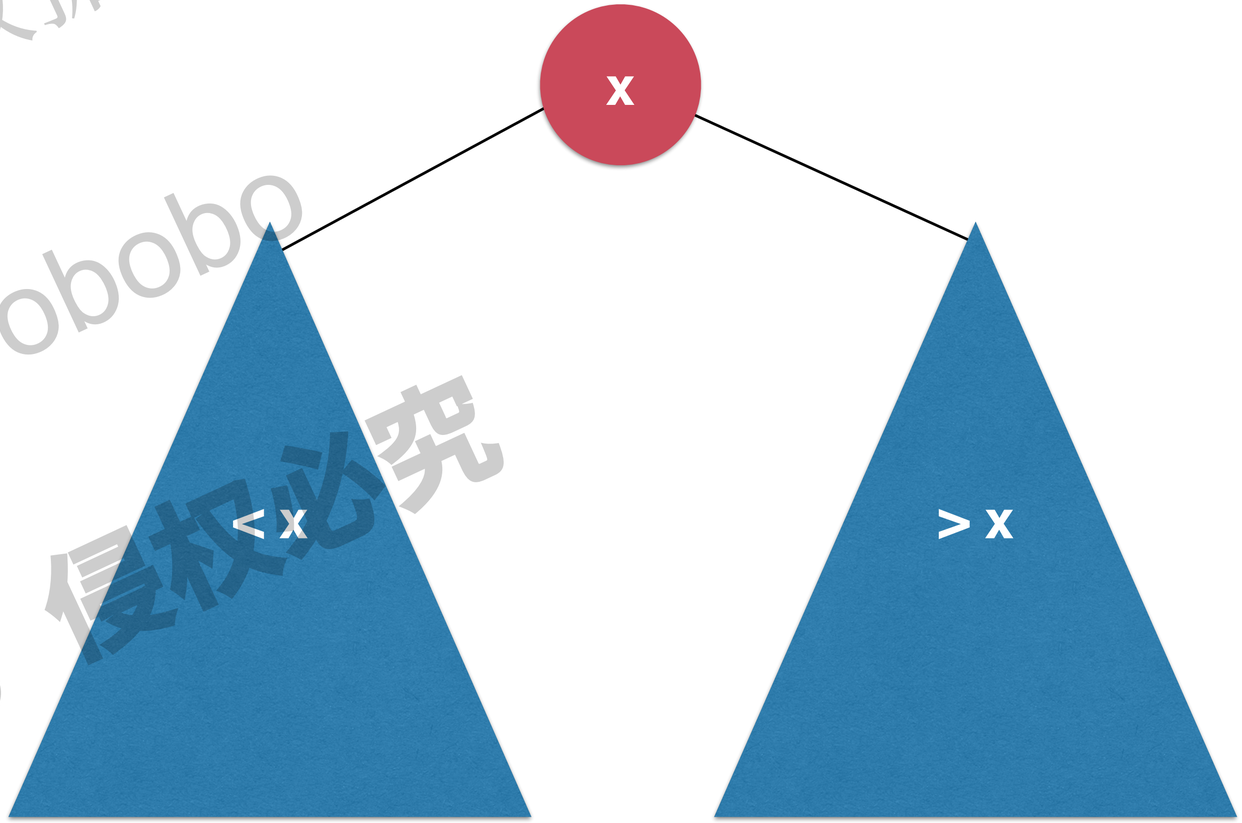
中序遍历

```
function traverse(node):
    if(node == null)
        return;

    traverse(node.left)
    访问该节点
    traverse(node.right)
```

实践：二分搜索树的中序遍历

中序遍历

```
function traverse(node):
    if(node == null)
        return;

    traverse(node.left)
    访问该节点
    traverse(node.right)
```



- 二分搜索树的中序遍历结果是顺序的

后序遍历

```
function traverse(node):
    if(node == null)
        return;

    traverse(node.left)
    traverse(node.right)
    访问该节点
```

实践：二分搜索树的后序遍历

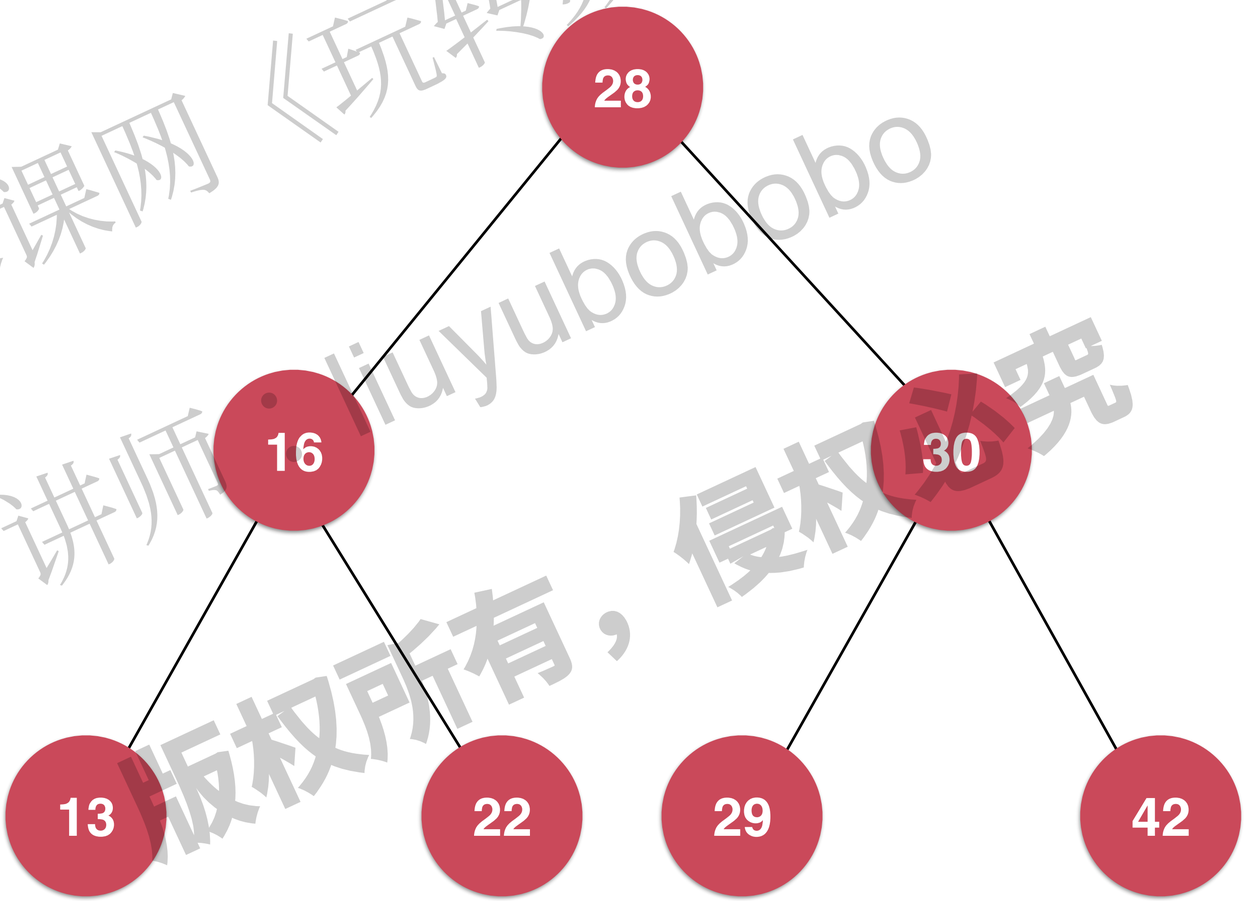
后序遍历

```
function traverse(node):
    if(node == null)
        return;

    traverse(node.left)
    traverse(node.right)
    访问该节点
```

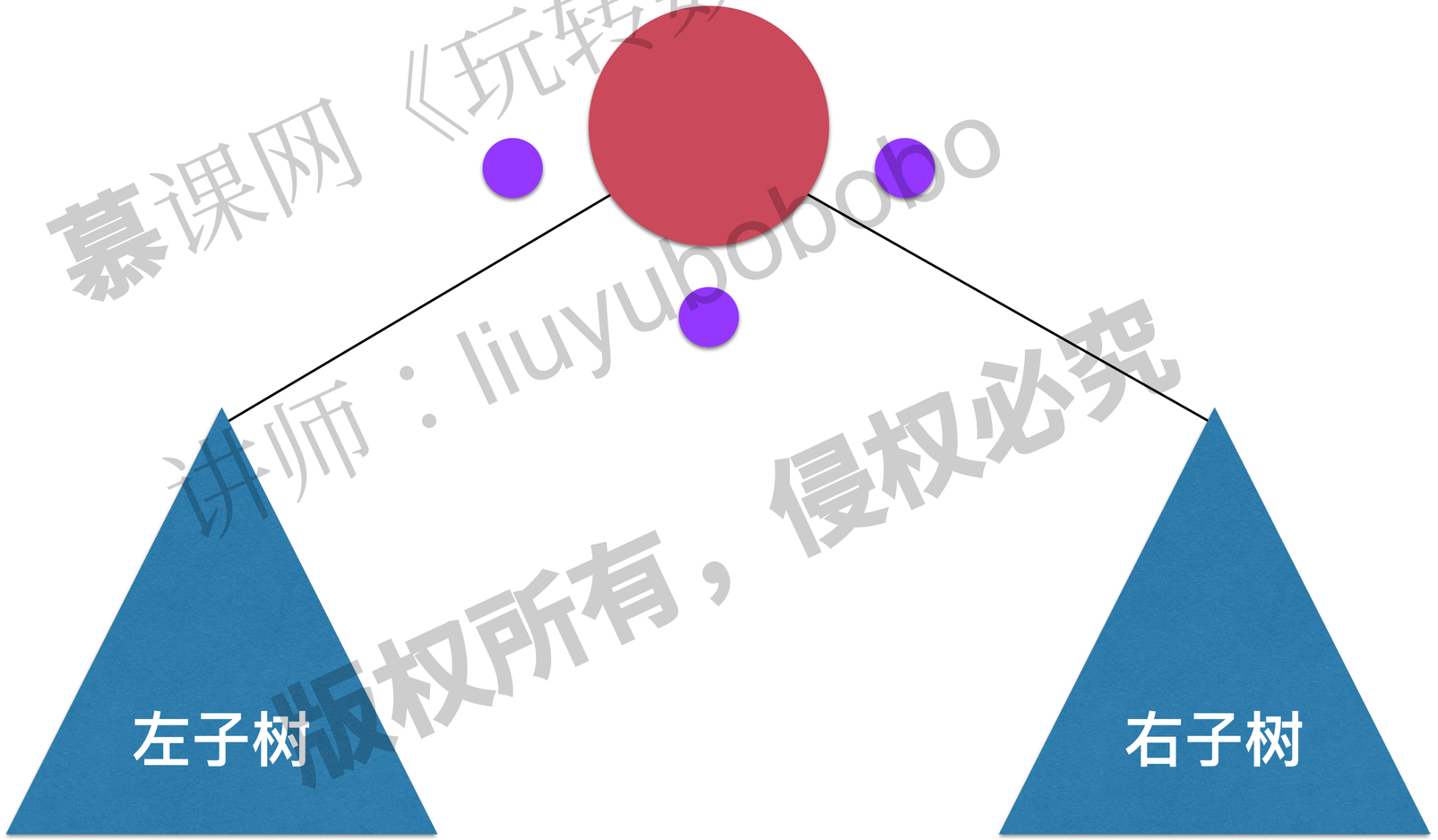
- 后序遍历的一个应用：
- 为二分搜索树释放内存

再看二分搜索树的遍历

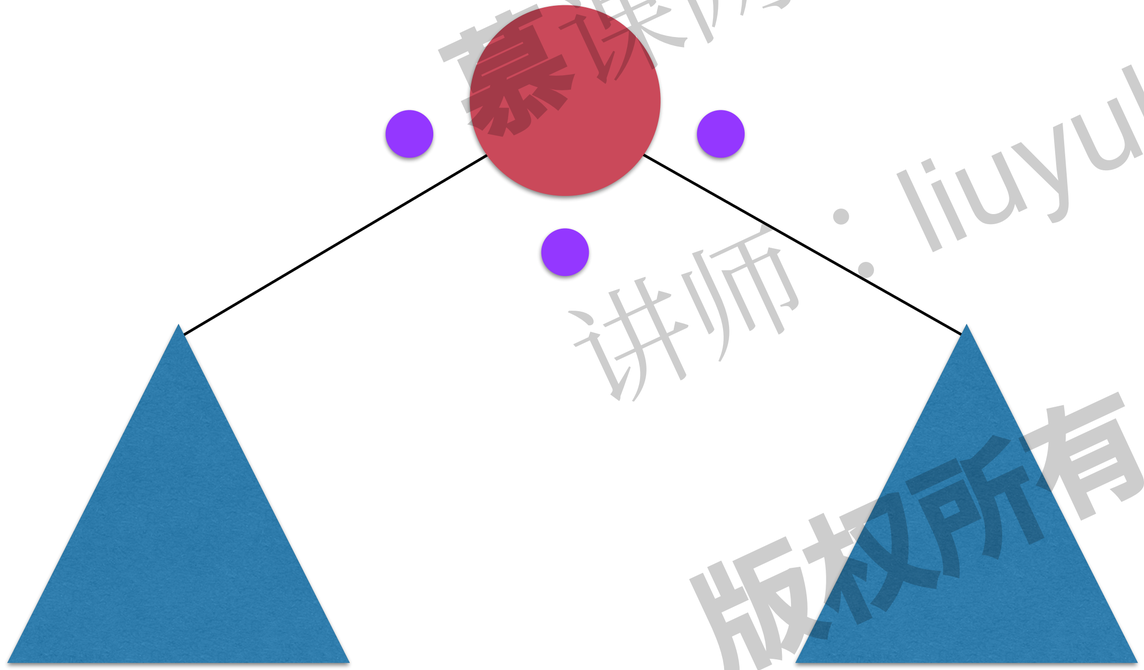
二分搜索树的遍历



二分搜索树的遍历



二分搜索树的遍历



```
function traverse(node):
    if(node == null) return;
```

- 访问该节点?

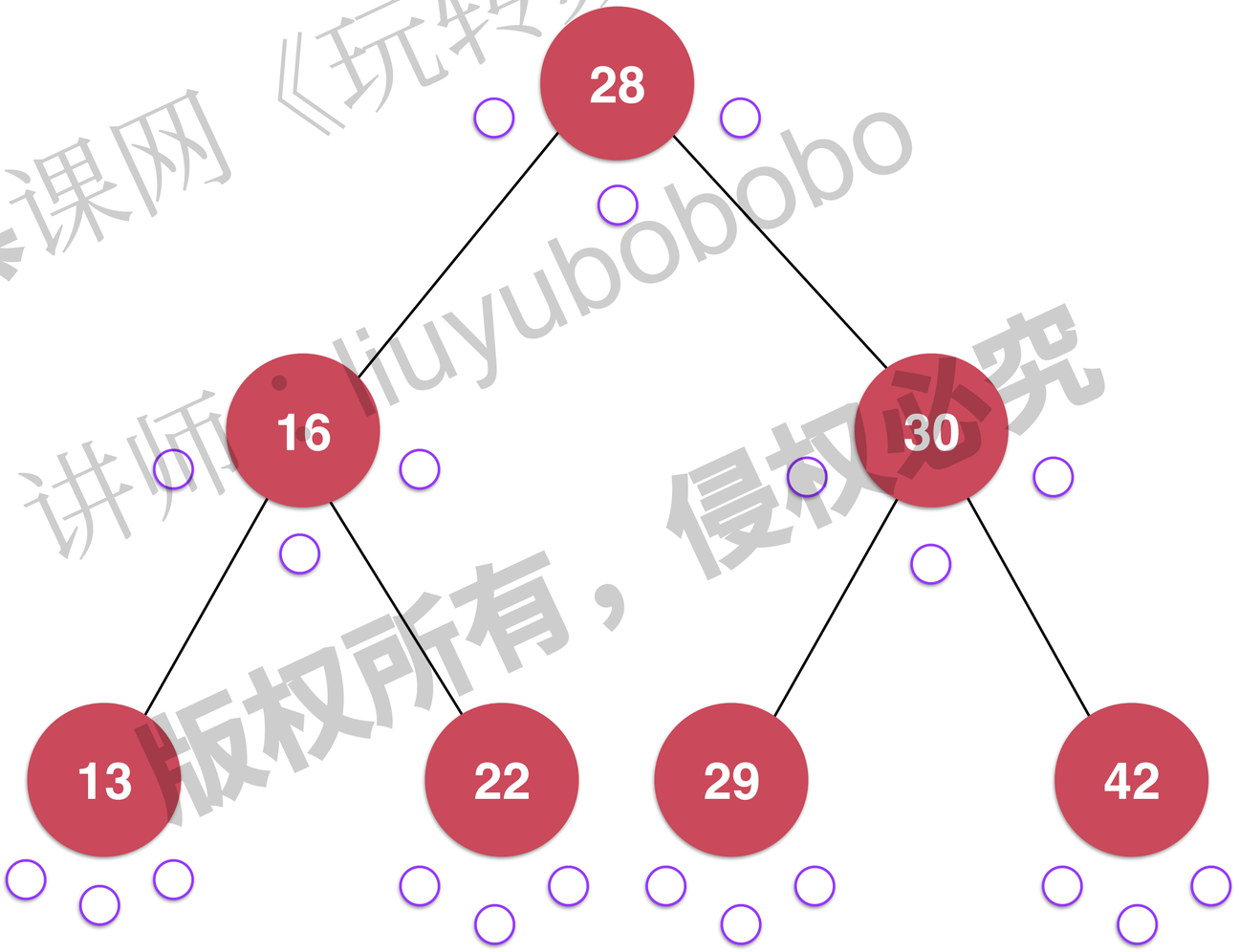
  traverse(node.left)

- 访问该节点?

  traverse(node.right)

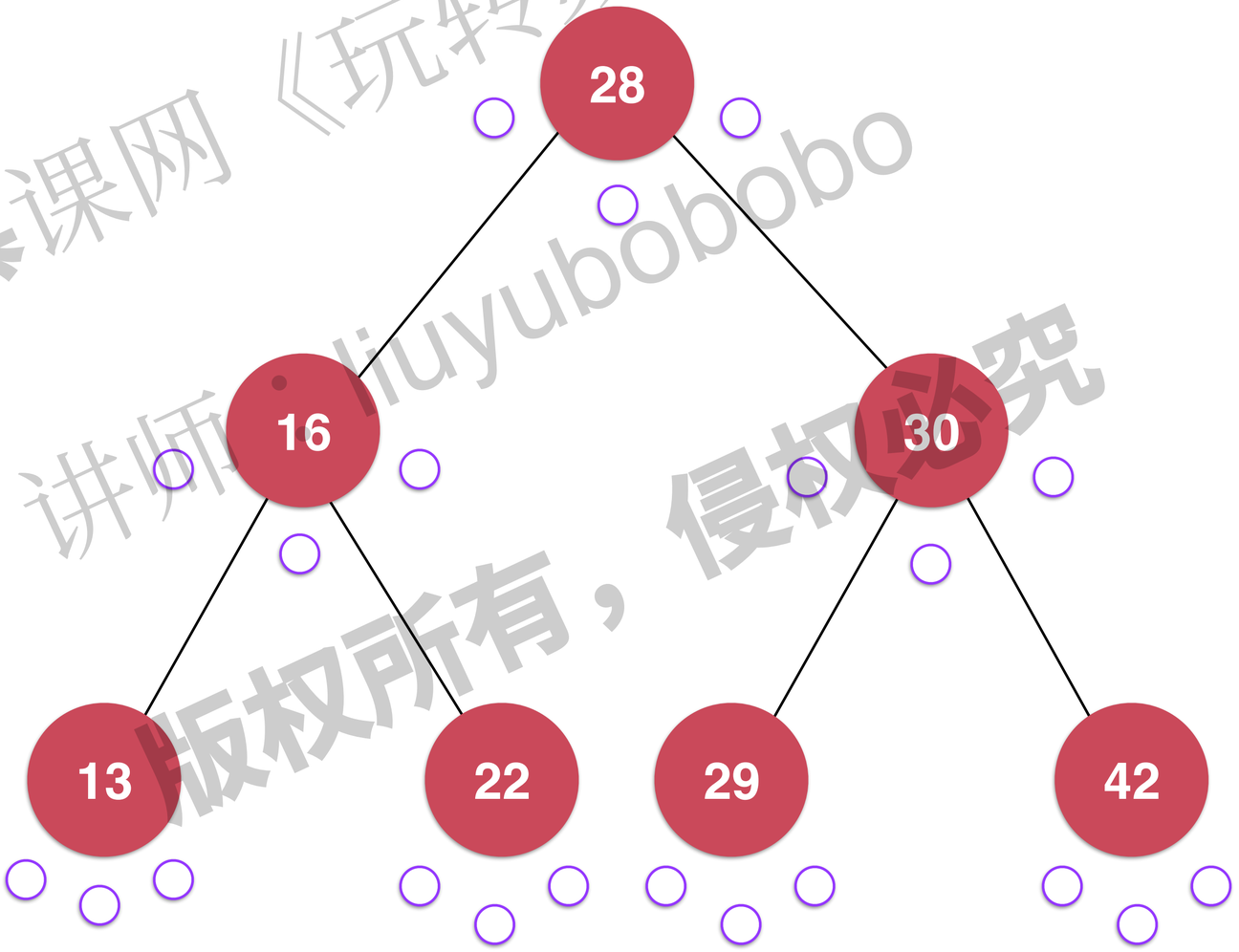
- 访问该节点?

二分搜索树的遍历

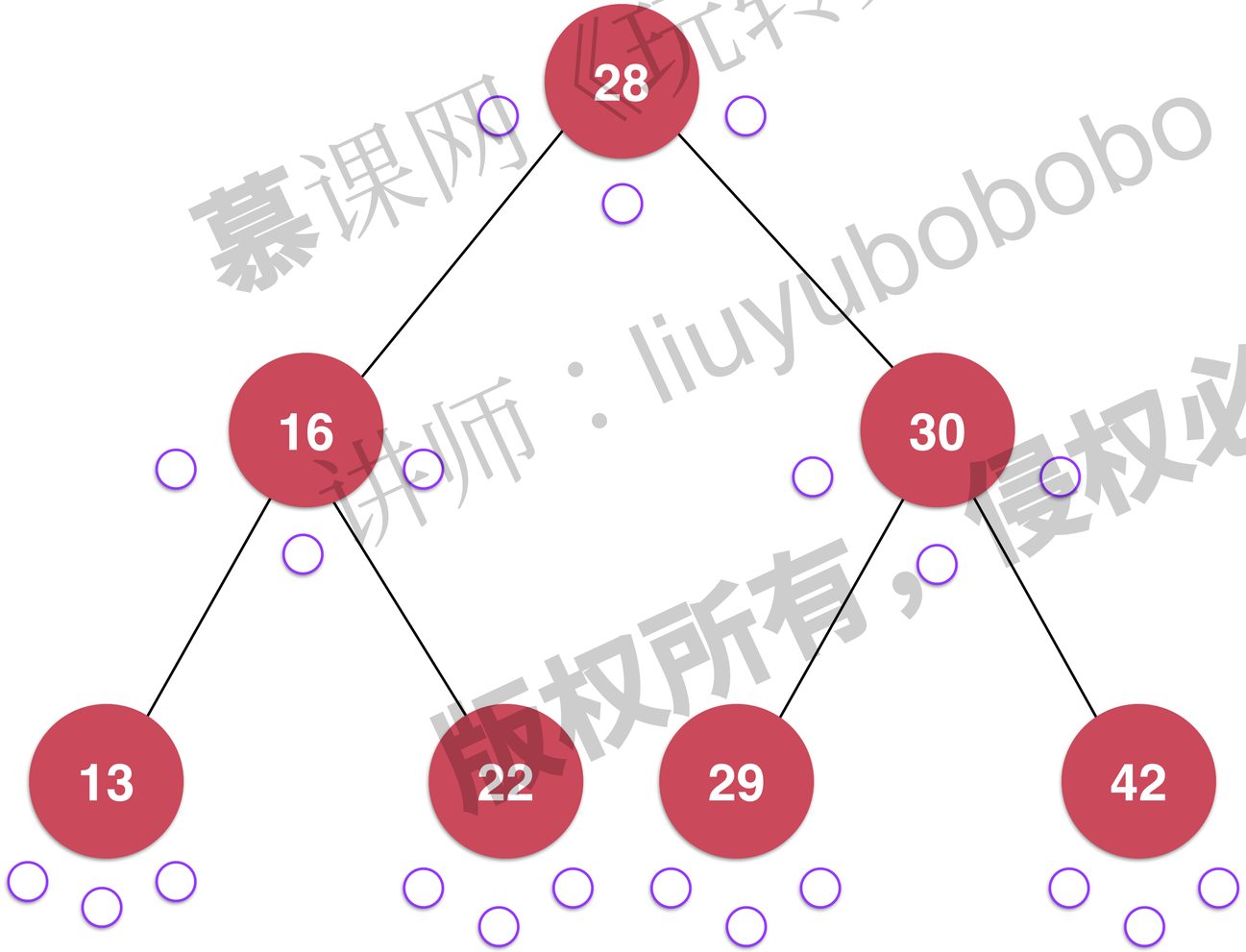


再看二分搜索树的前序遍历

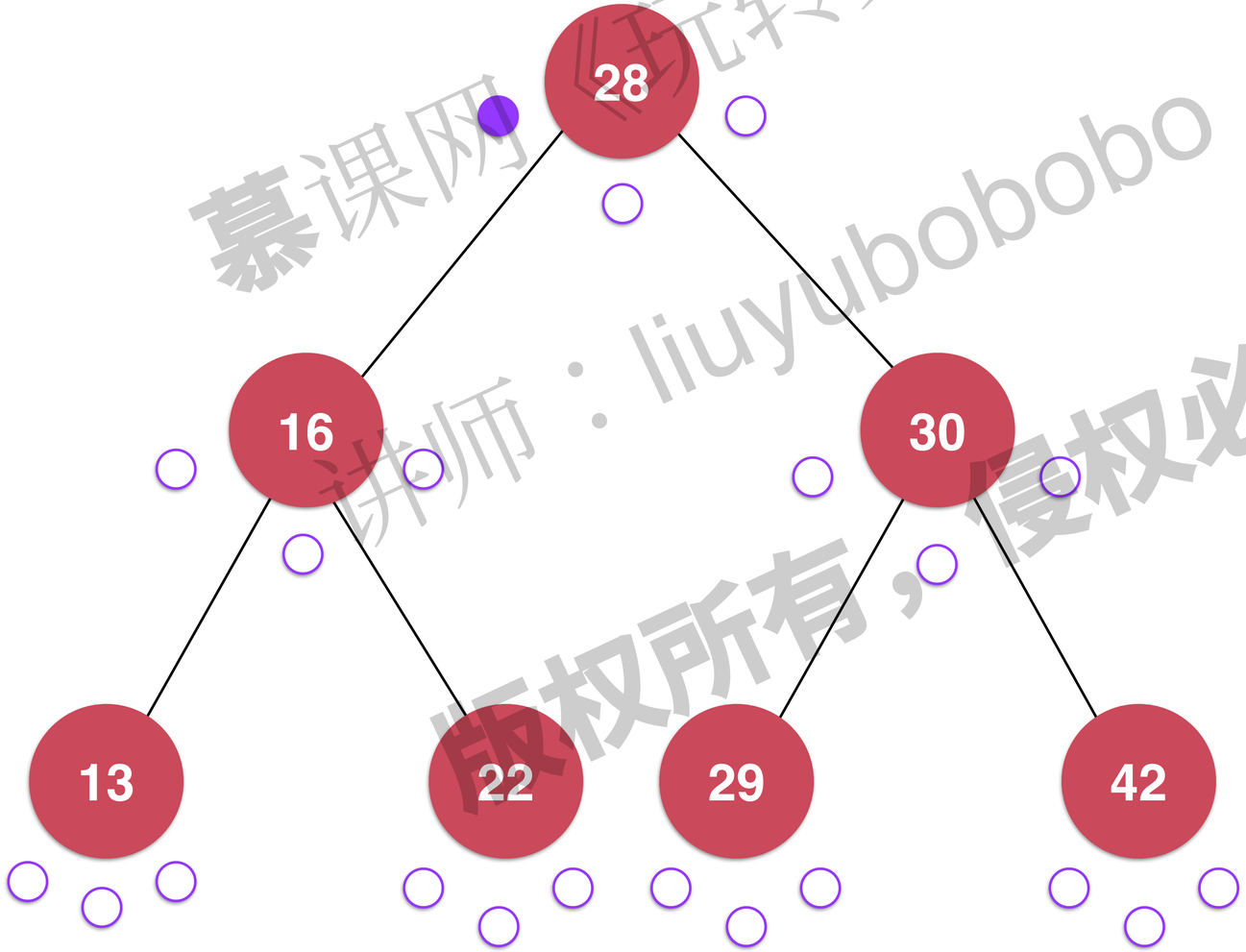
二分搜索树的遍历



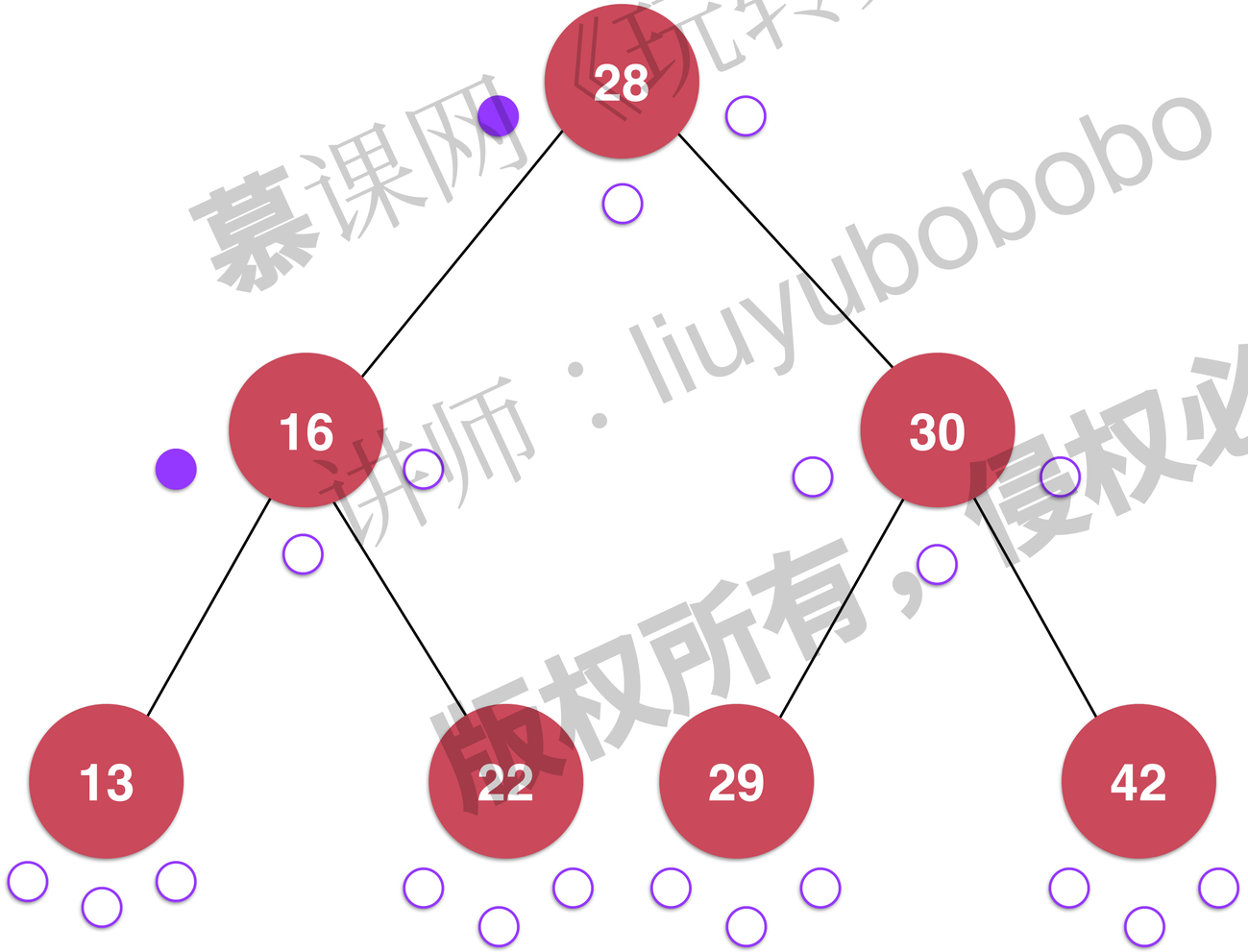
二分搜索树的前序遍历



二分搜索树的前序遍历



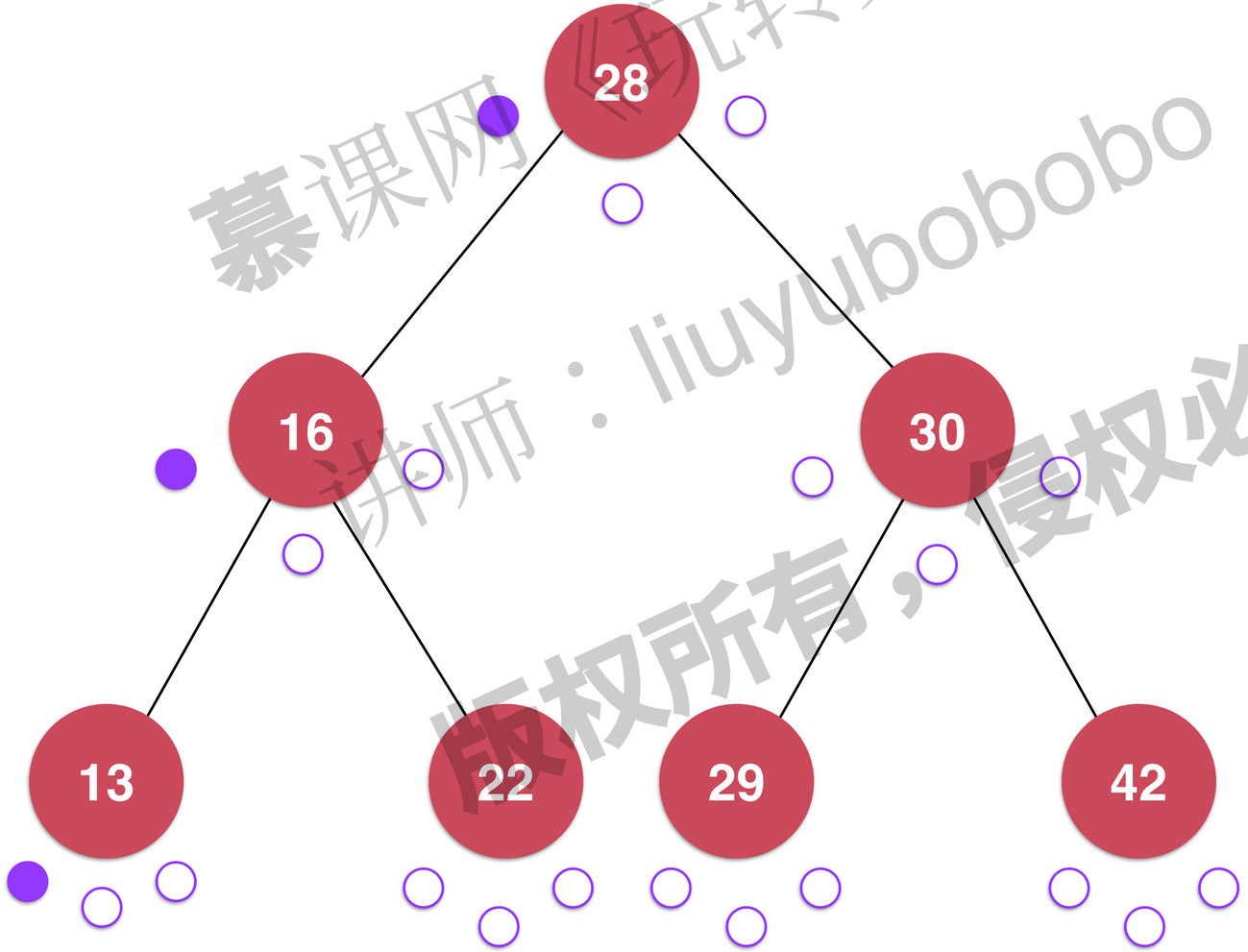
二分搜索树的前序遍历



28

16

二分搜索树的前序遍历

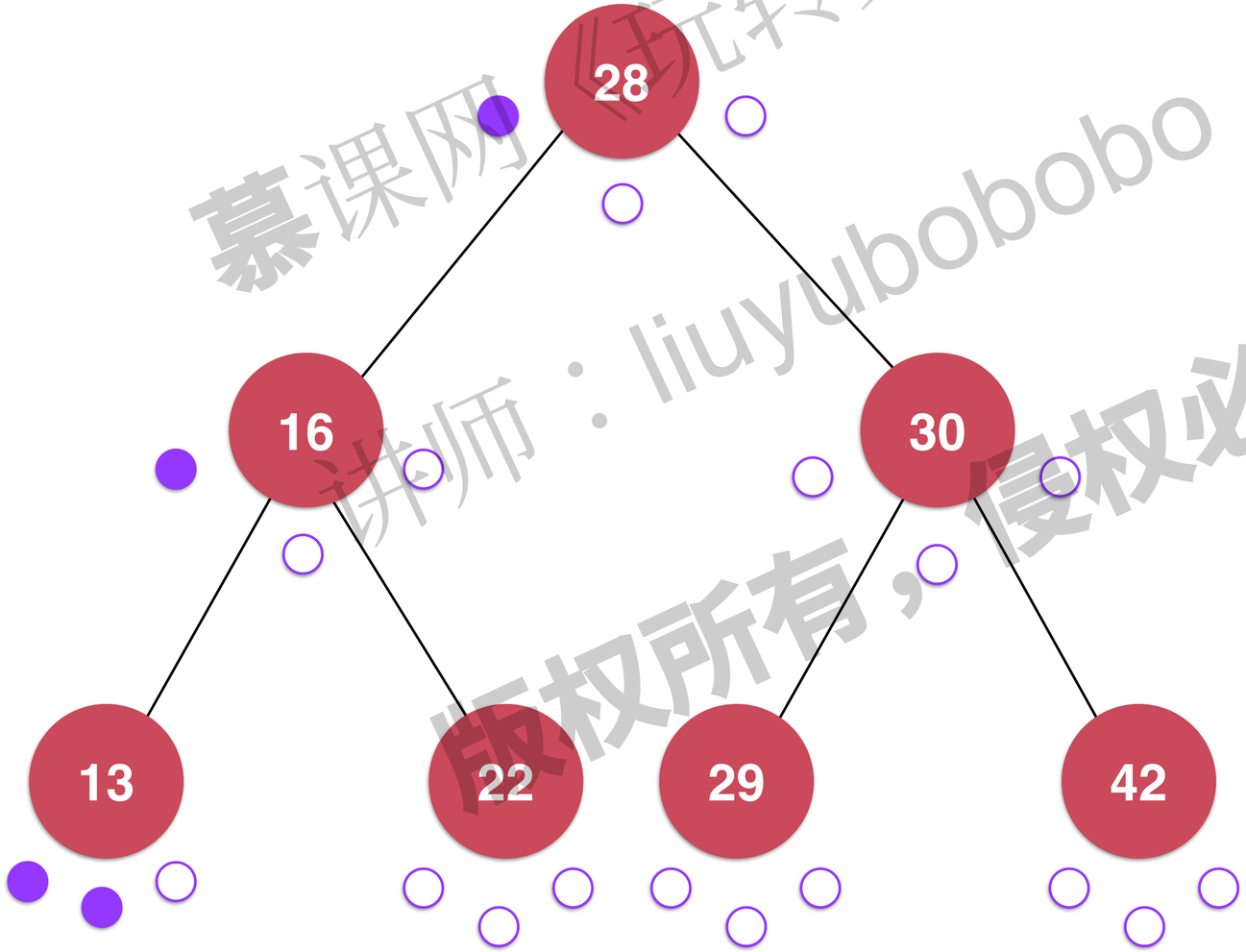


28

16

13

二分搜索树的前序遍历

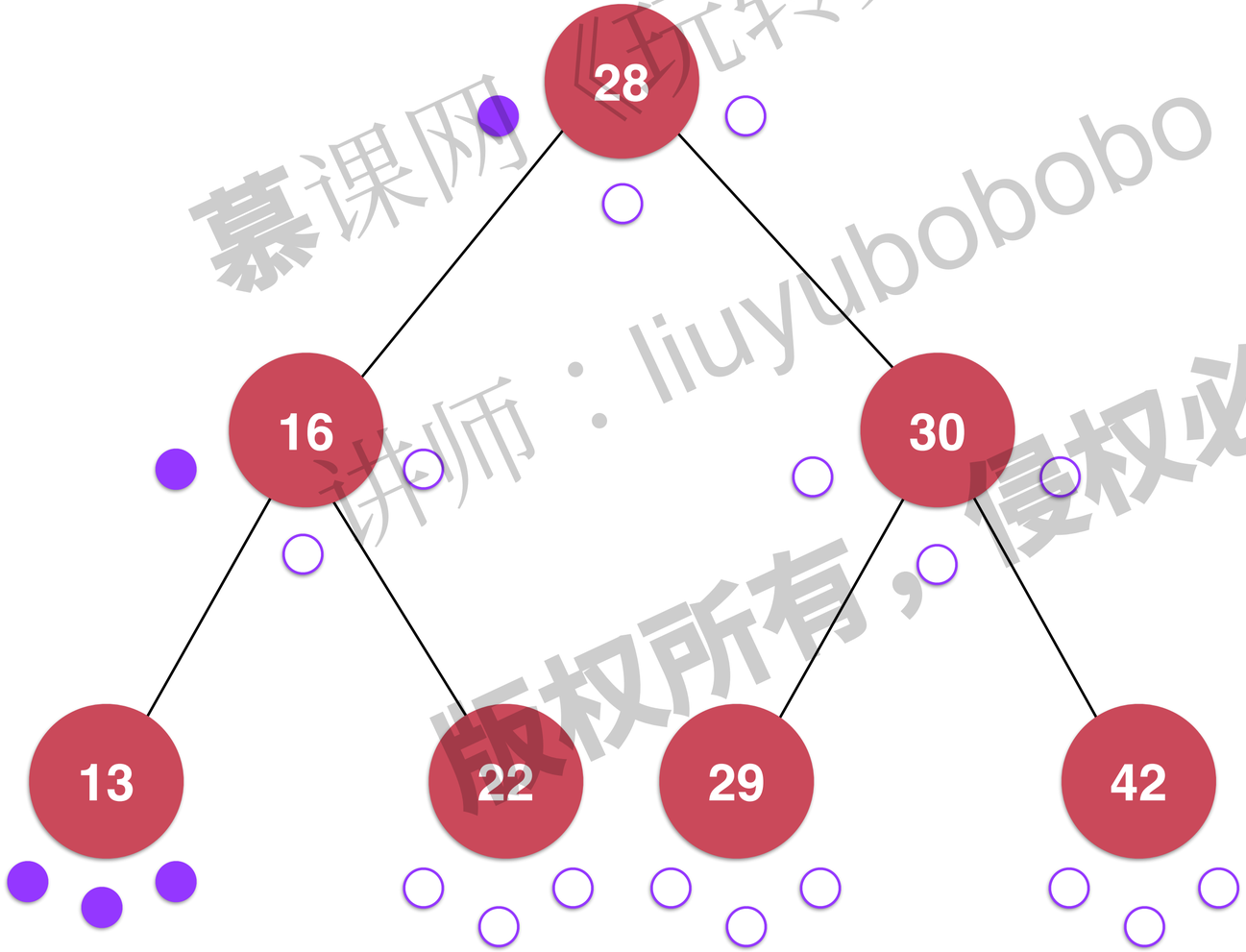


28

16

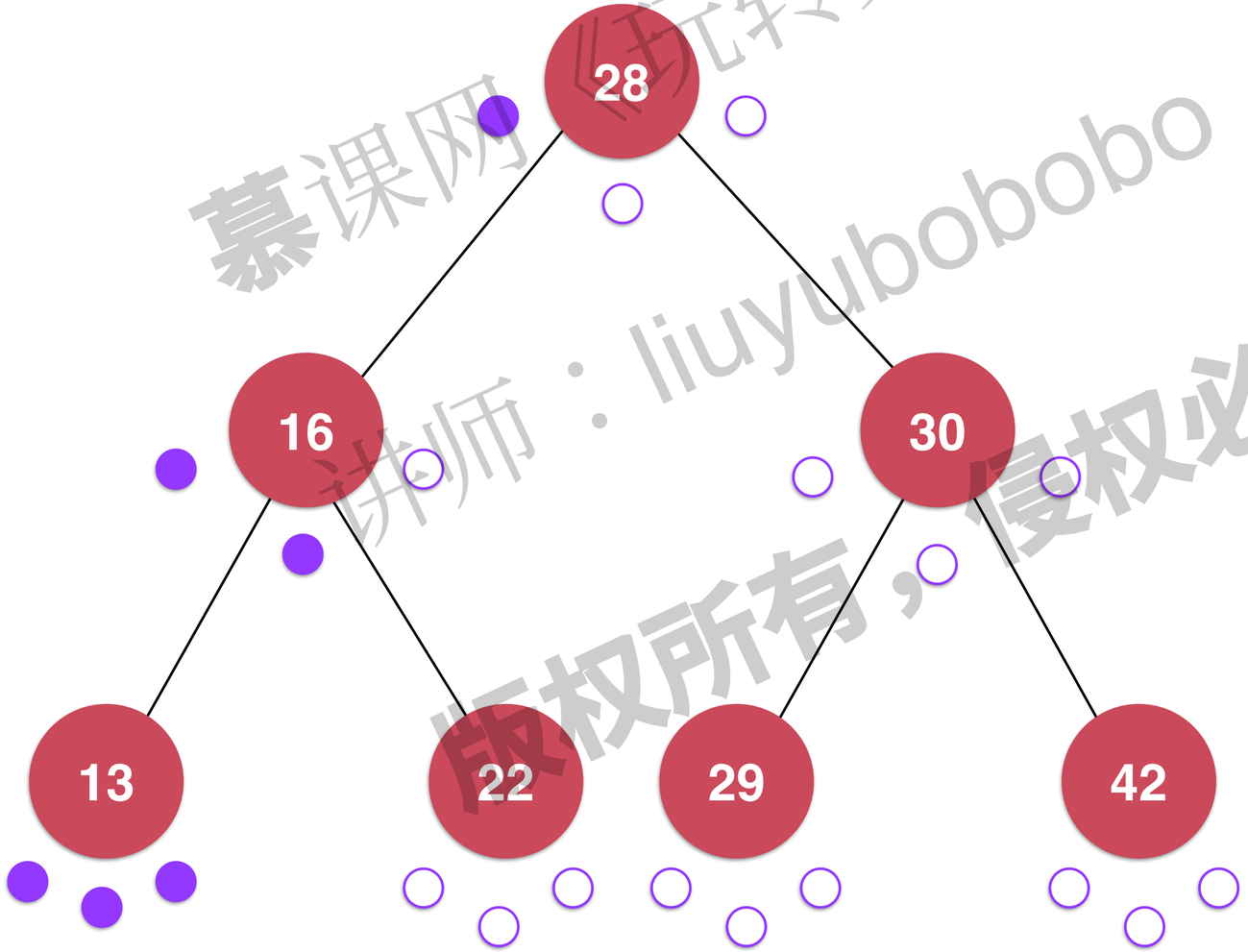
13

二分搜索树的前序遍历



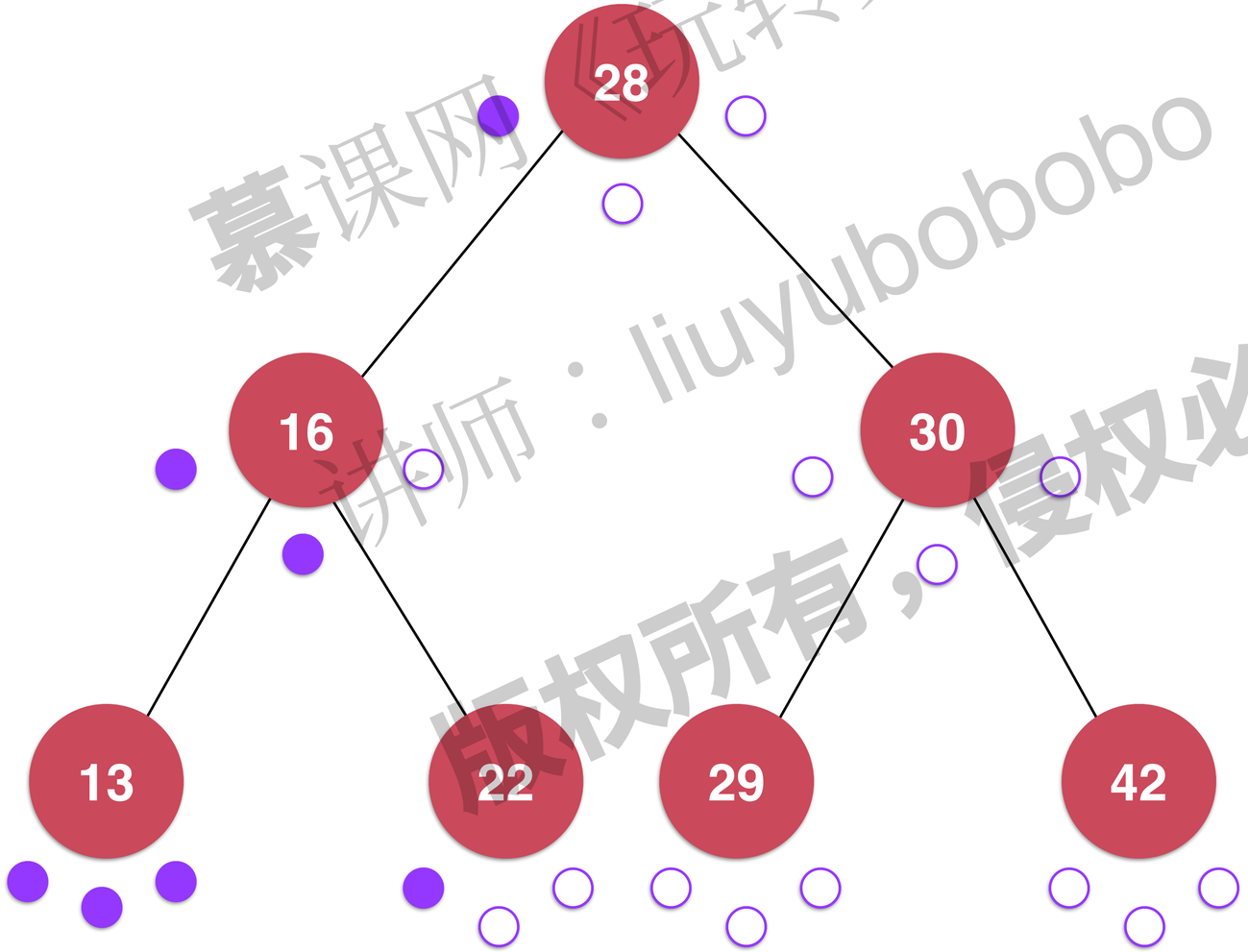
28
16
13

二分搜索树的前序遍历



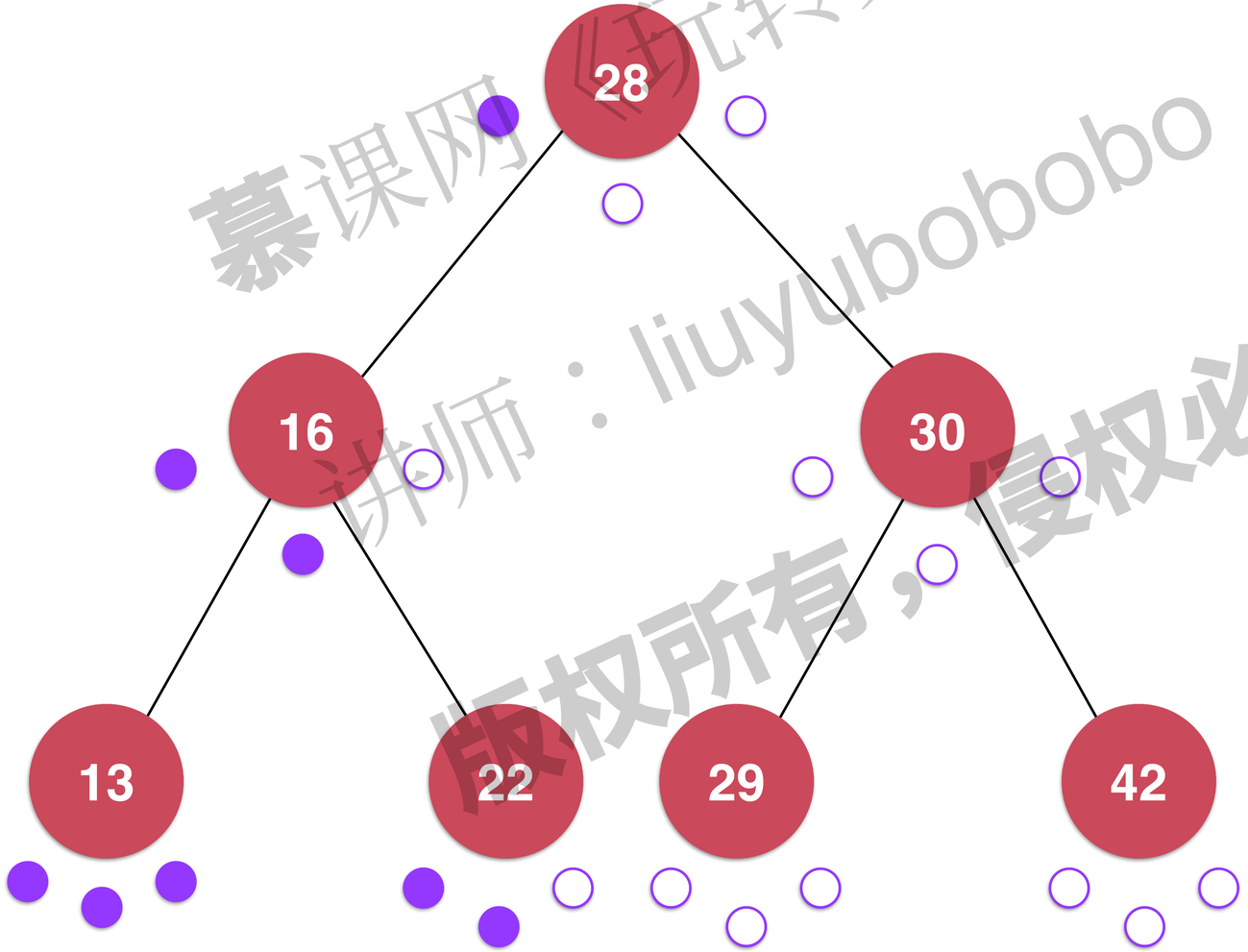
28
16
13

二分搜索树的前序遍历



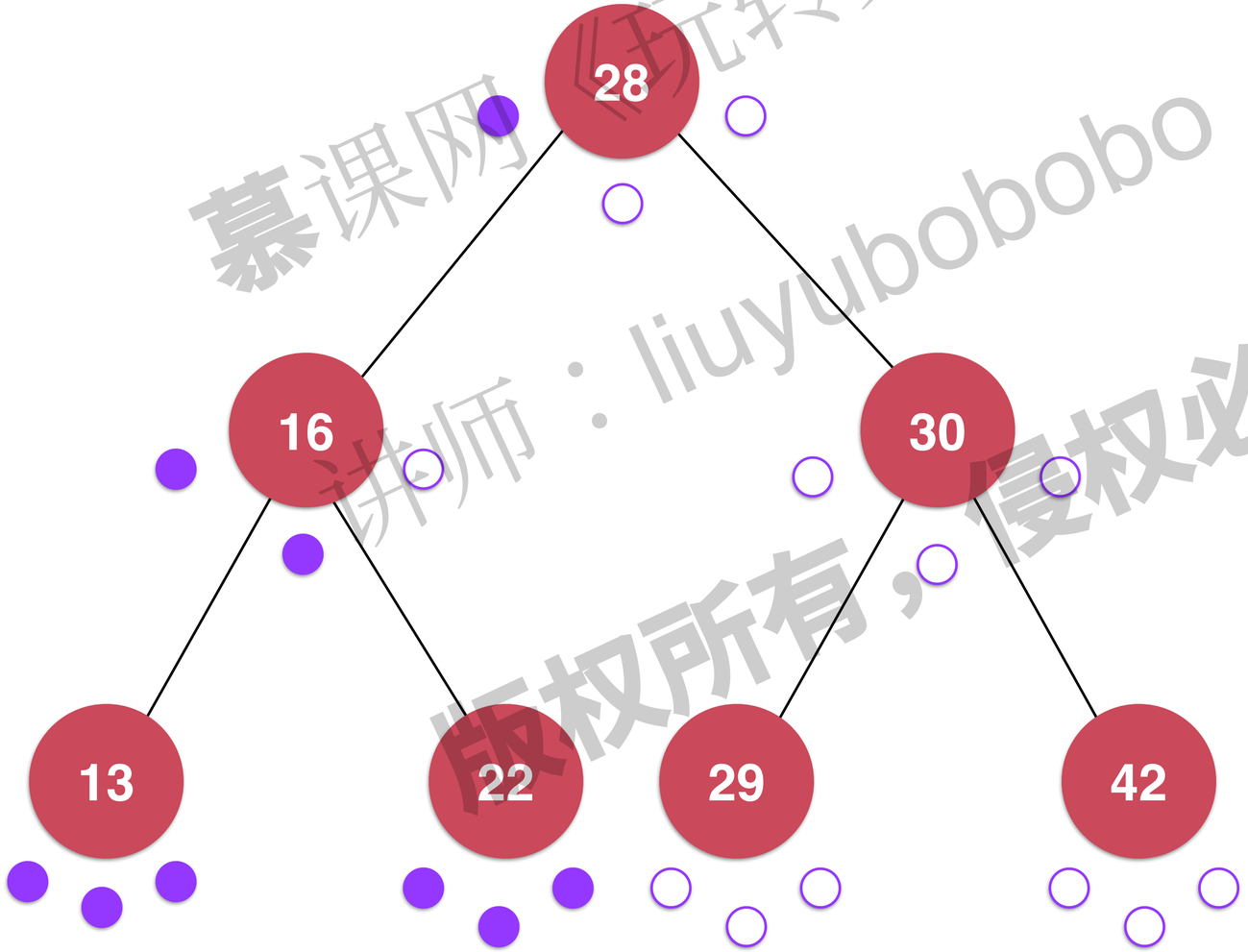
28
16
13
22

二分搜索树的前序遍历



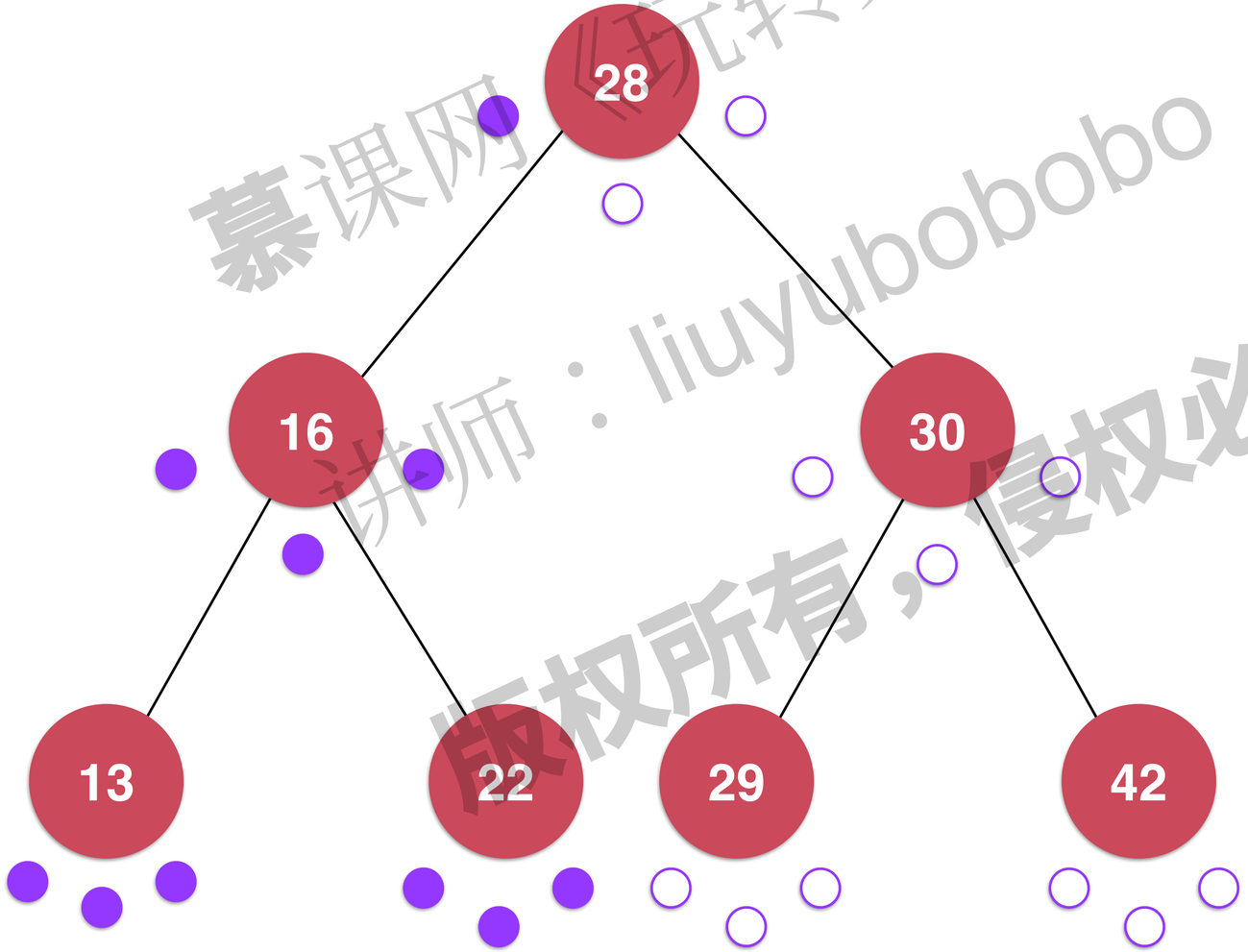
28
16
13
22

二分搜索树的前序遍历



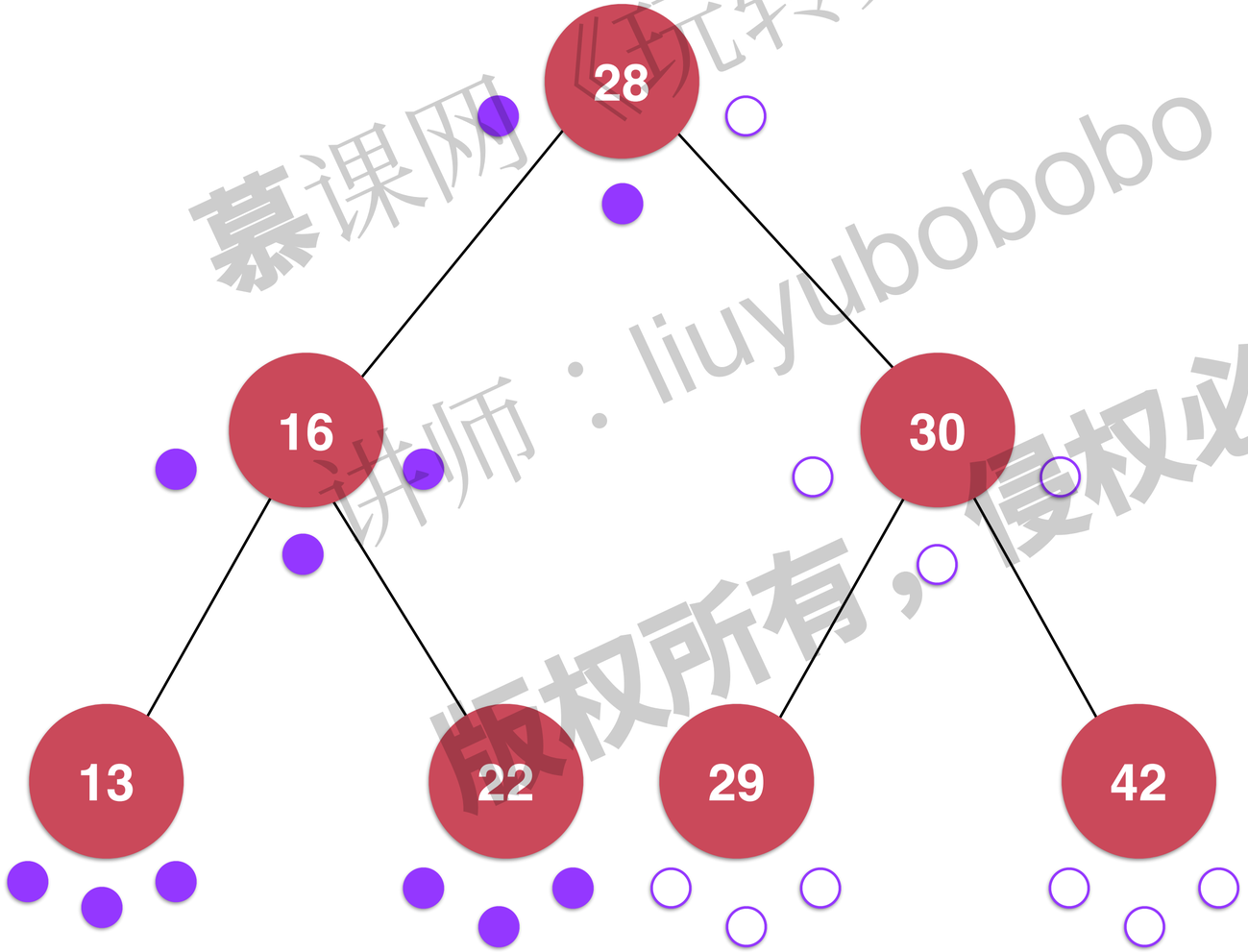
28
16
13
22

二分搜索树的前序遍历



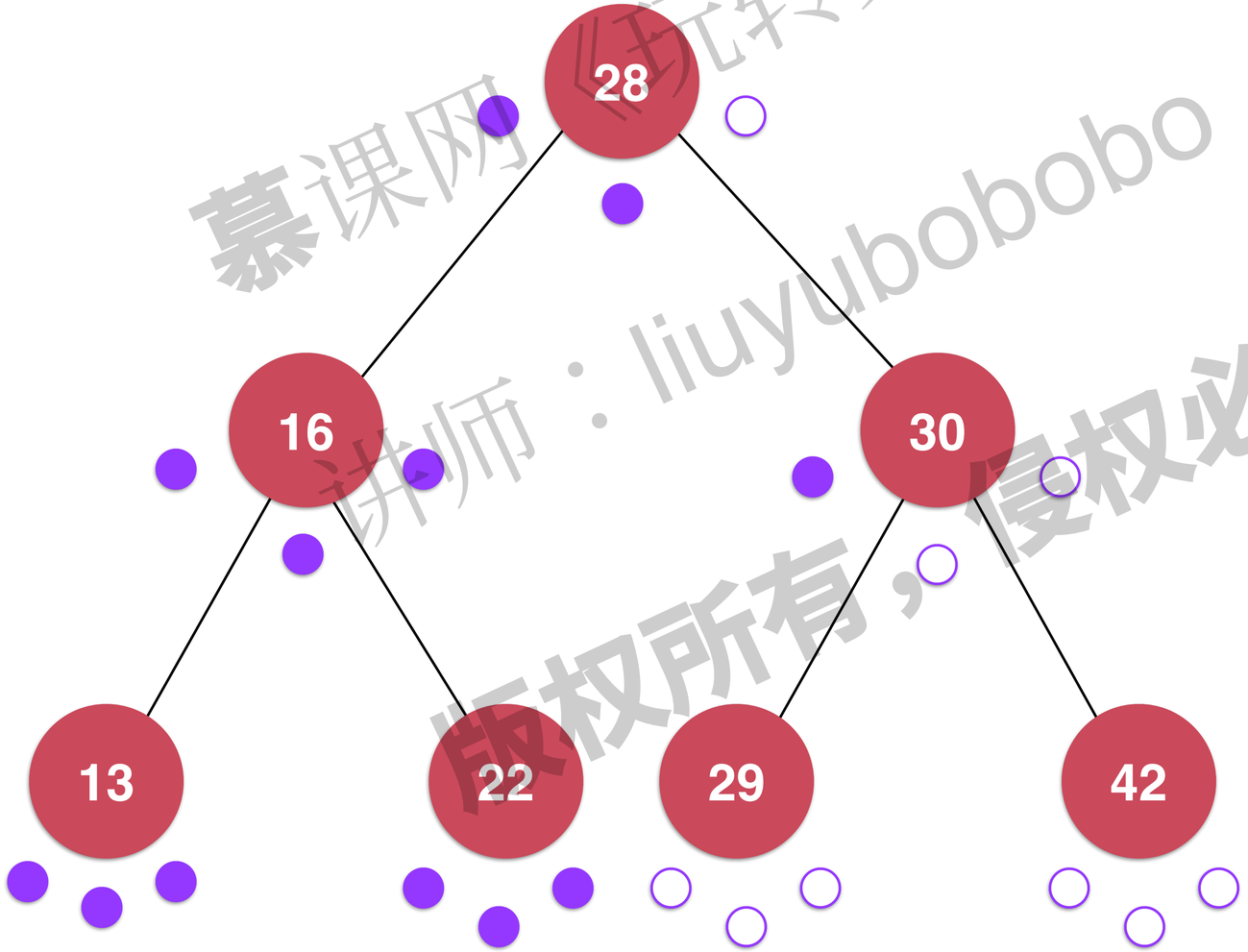
28
16
13
22

二分搜索树的前序遍历



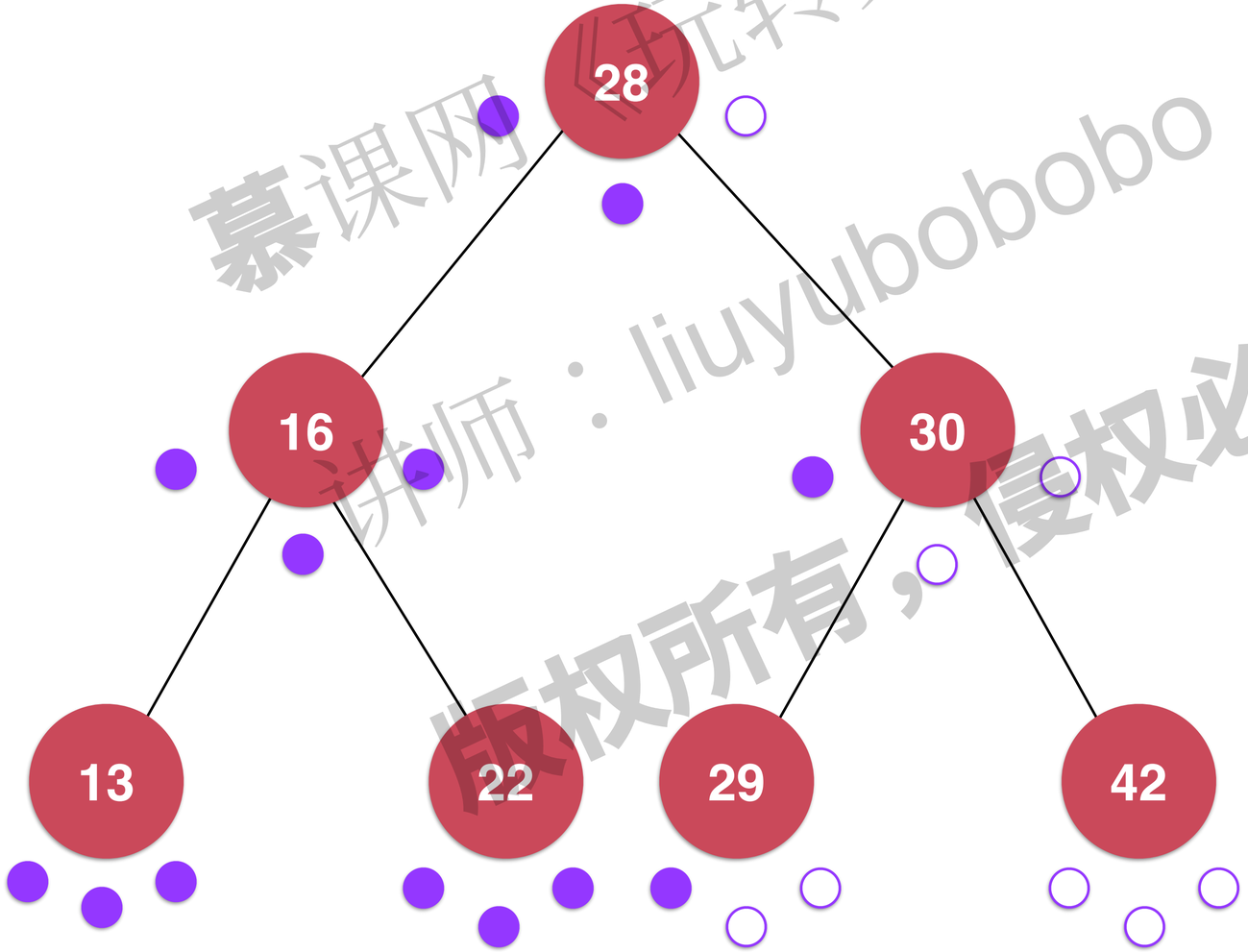
28
16
13
22

二分搜索树的前序遍历



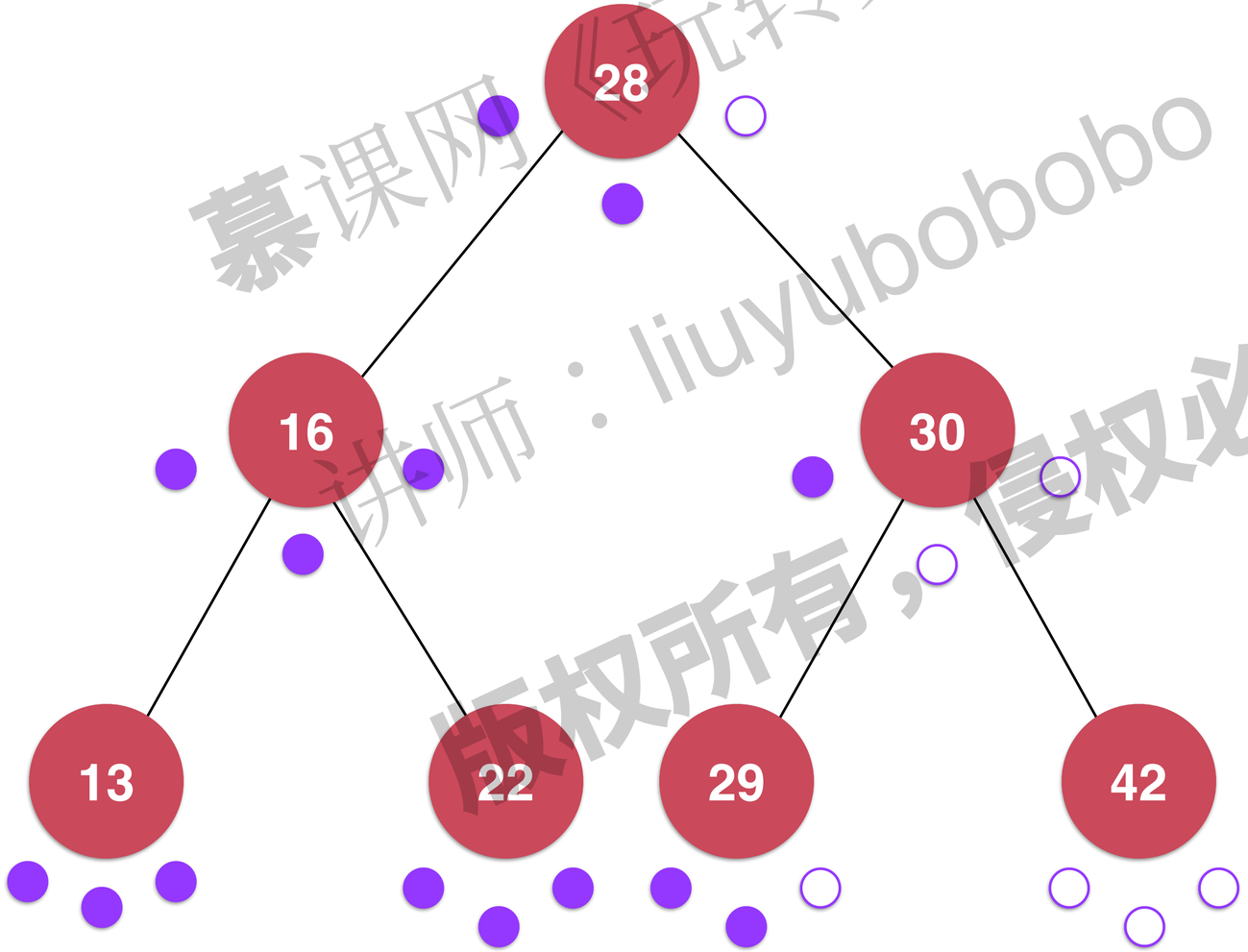
28
16
13
22
30

二分搜索树的前序遍历



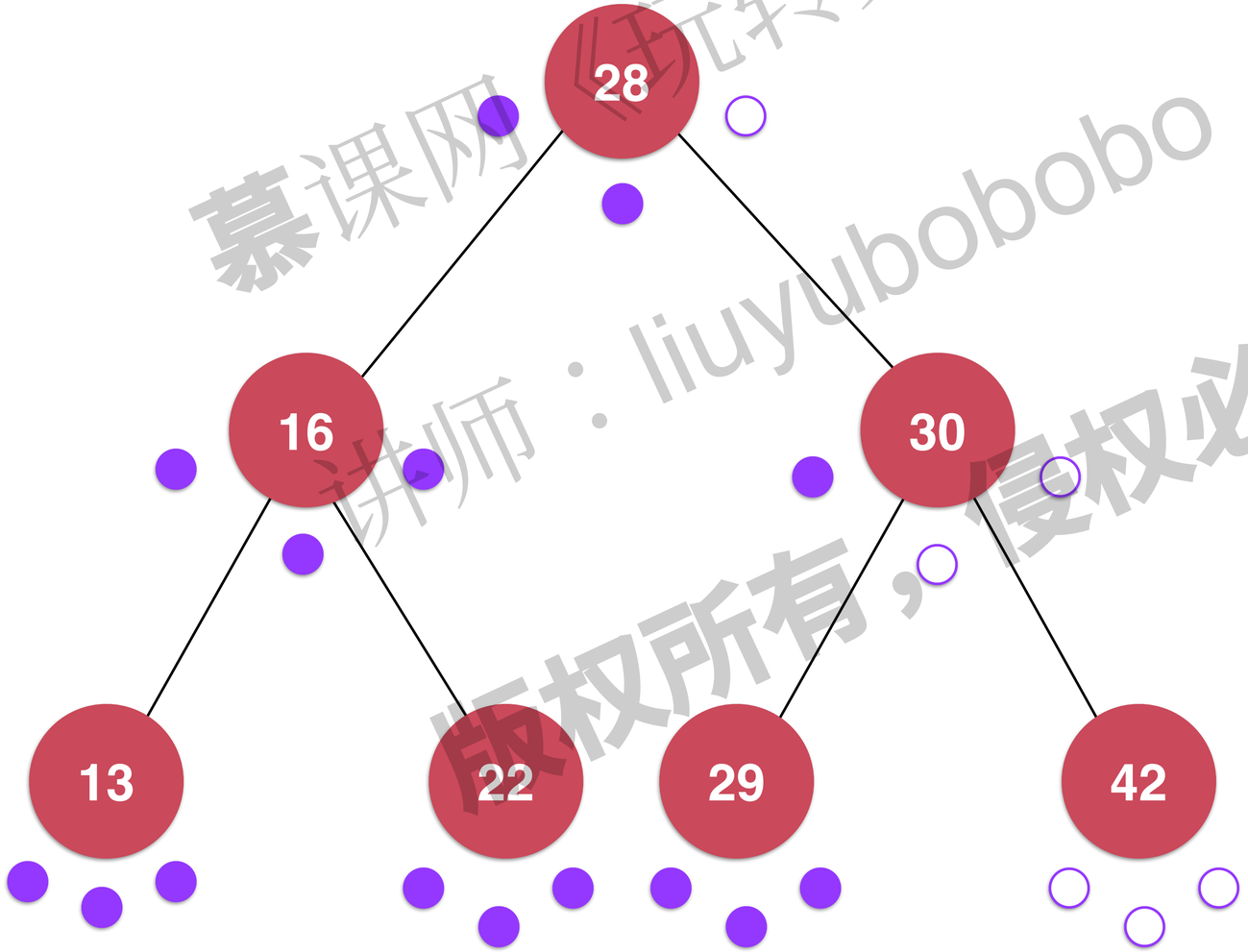
28
16
13
22
30
29

二分搜索树的前序遍历



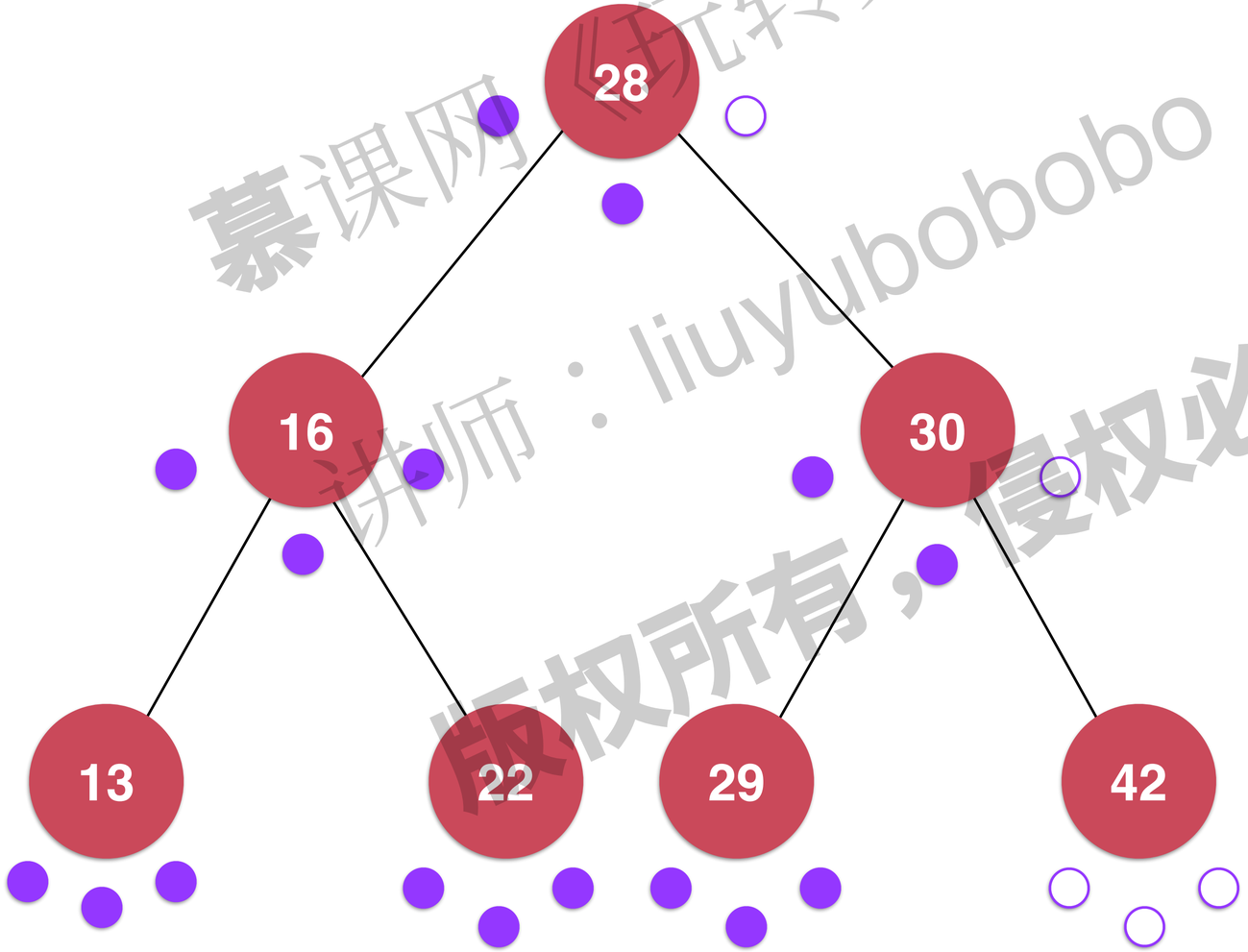
28
16
13
22
30
29

二分搜索树的前序遍历



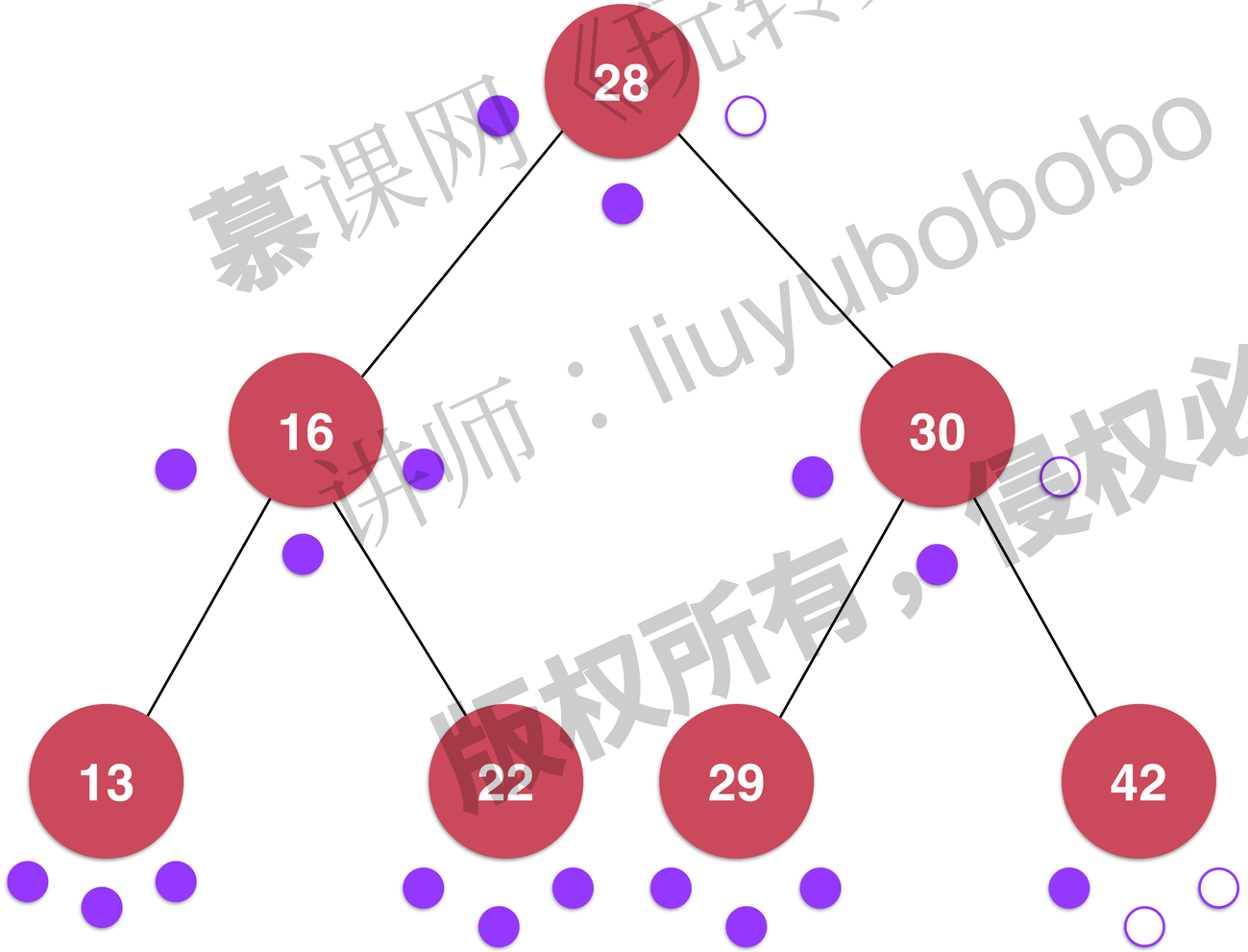
28
16
13
22
30
29

二分搜索树的前序遍历



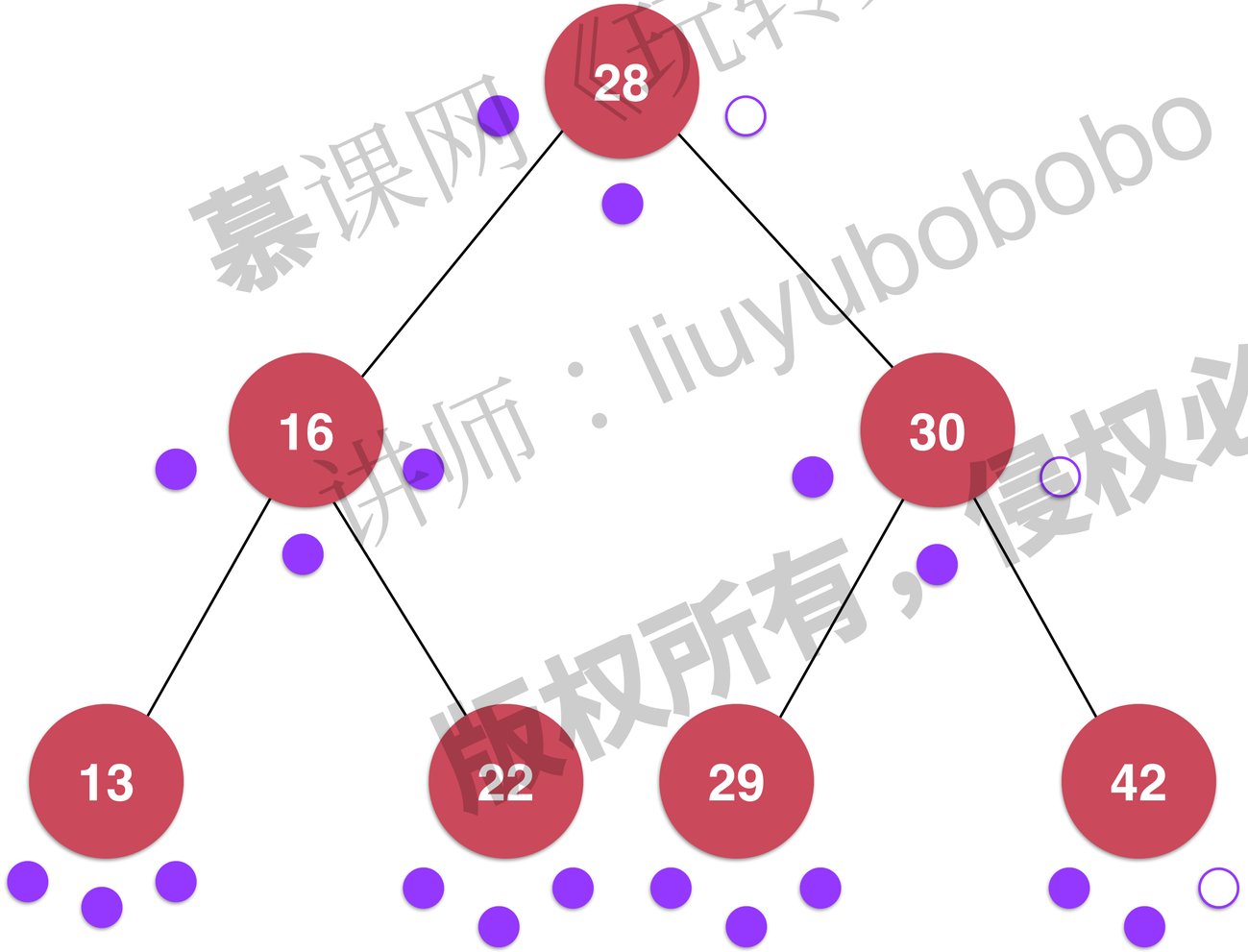
28
16
13
22
30
29

二分搜索树的前序遍历



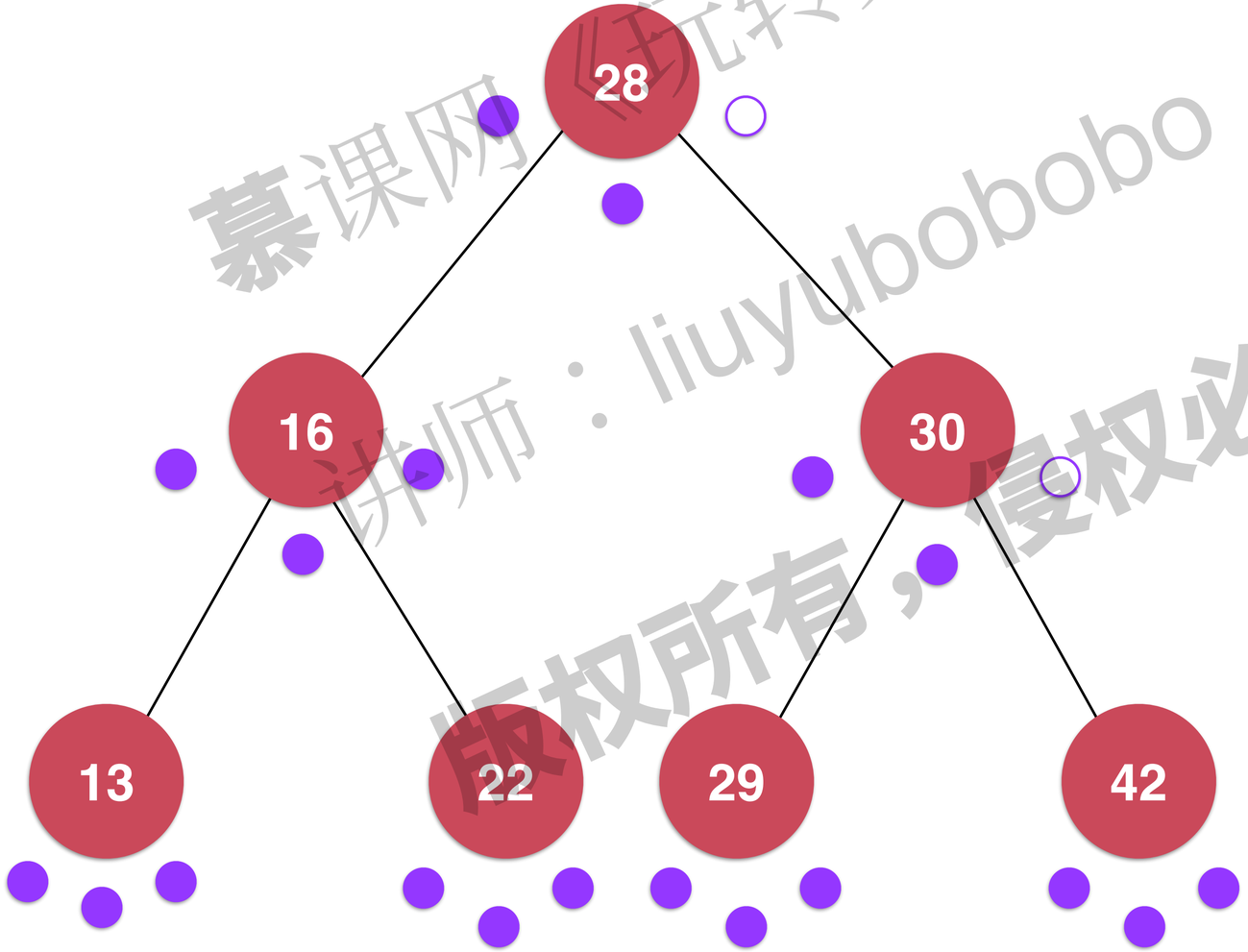
28
16
13
22
30
29
42

二分搜索树的前序遍历



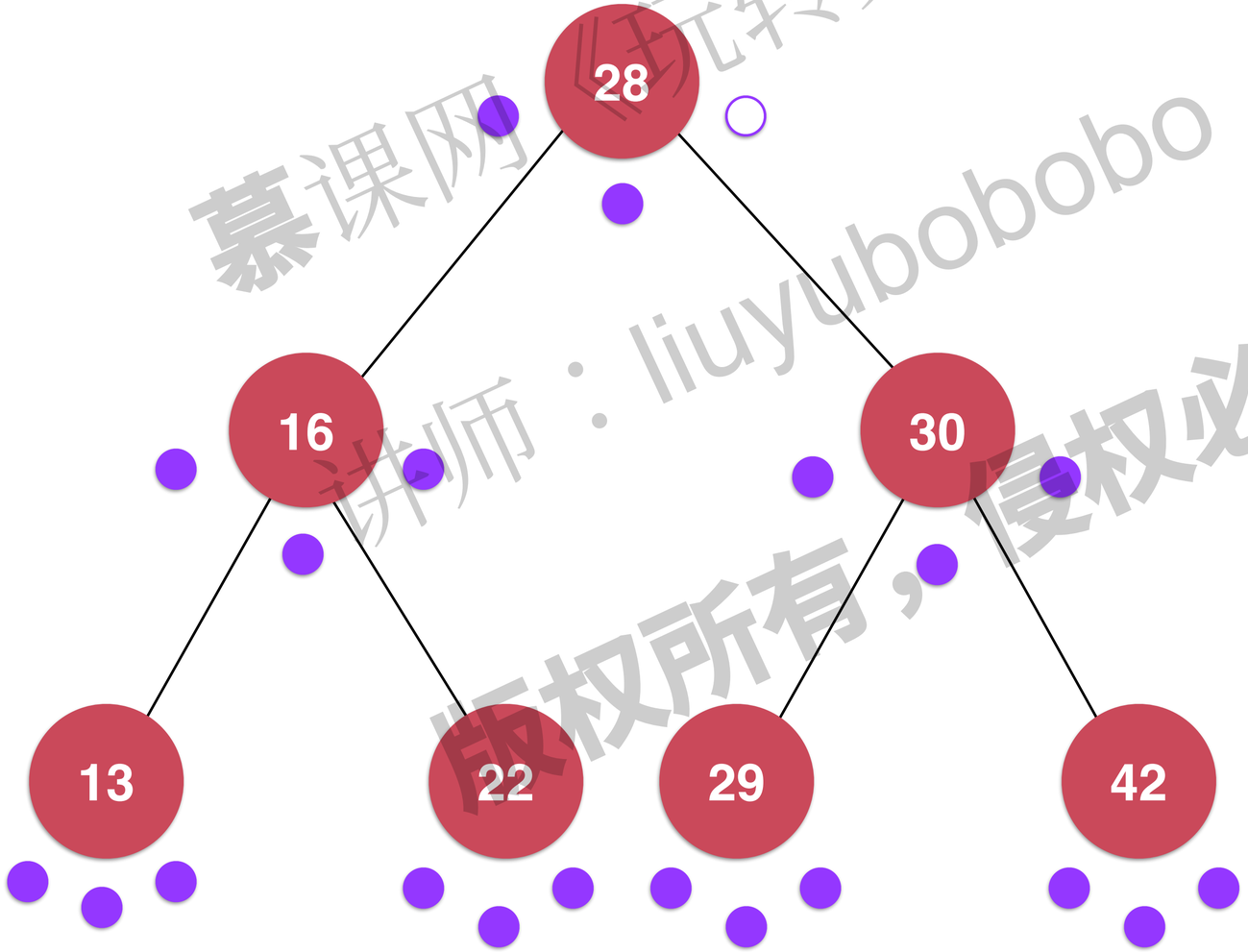
28
16
13
22
30
29
42

二分搜索树的前序遍历



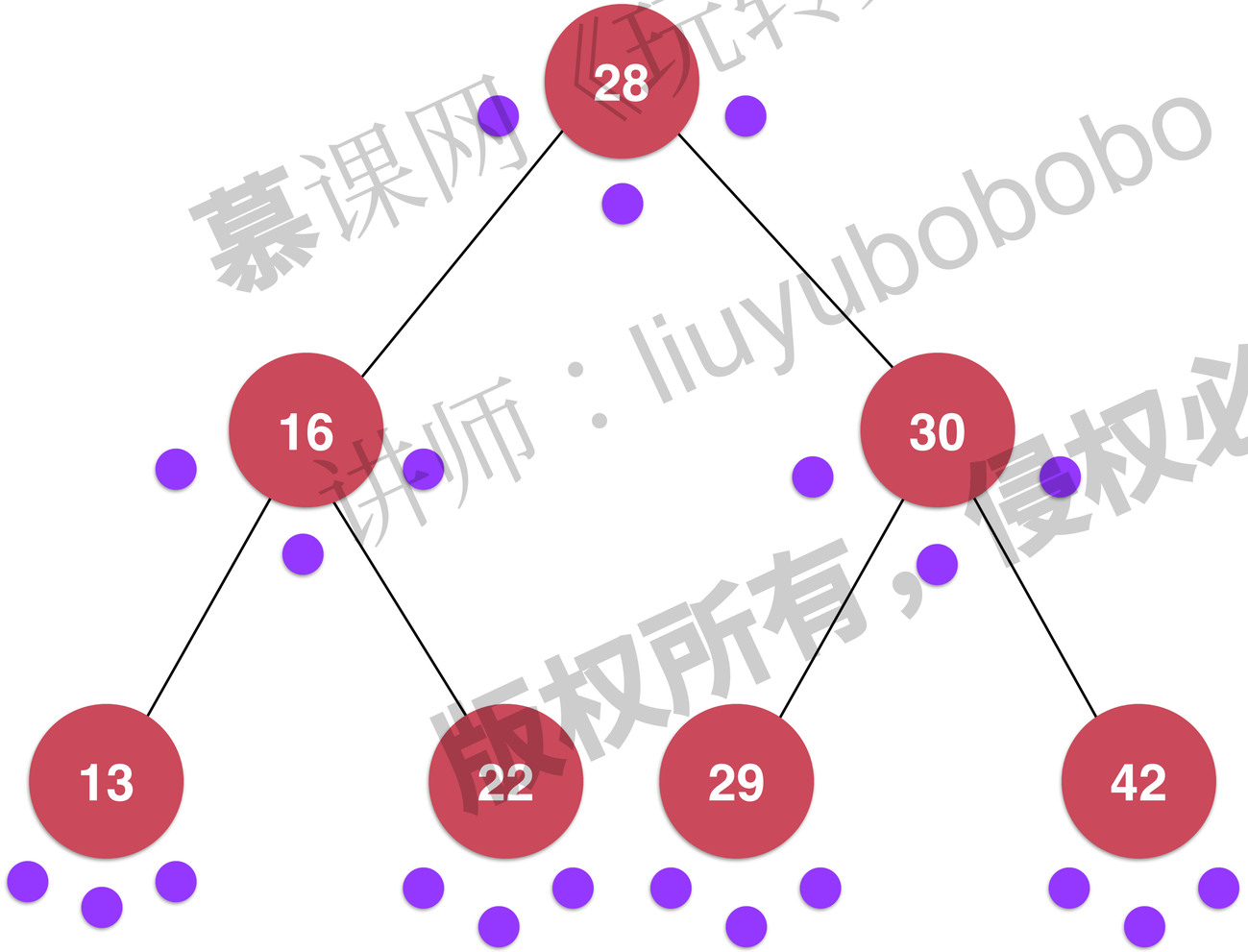
28
16
13
22
30
29
42

二分搜索树的前序遍历



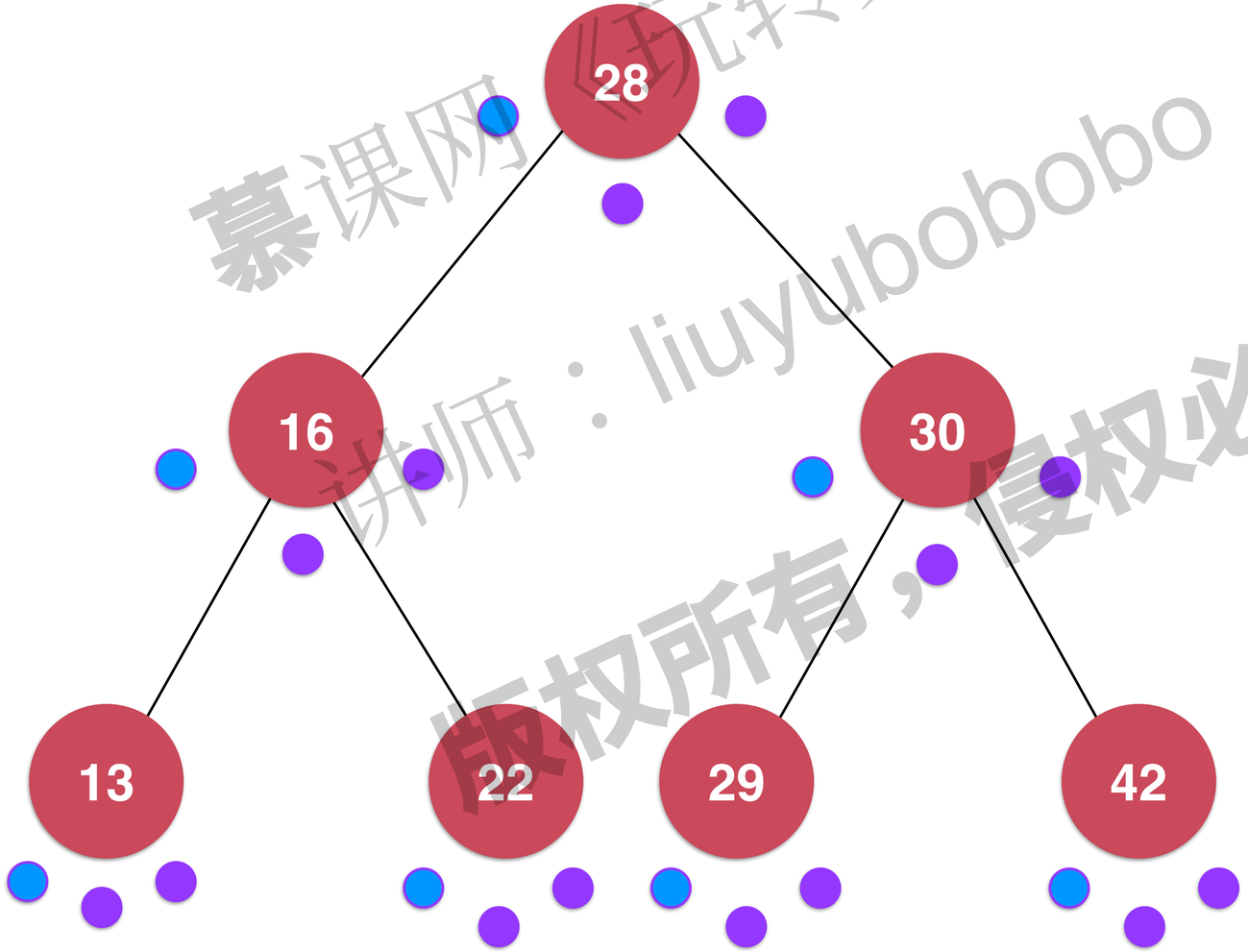
28
16
13
22
30
29
42

二分搜索树的前序遍历



28
16
13
22
30
29
42

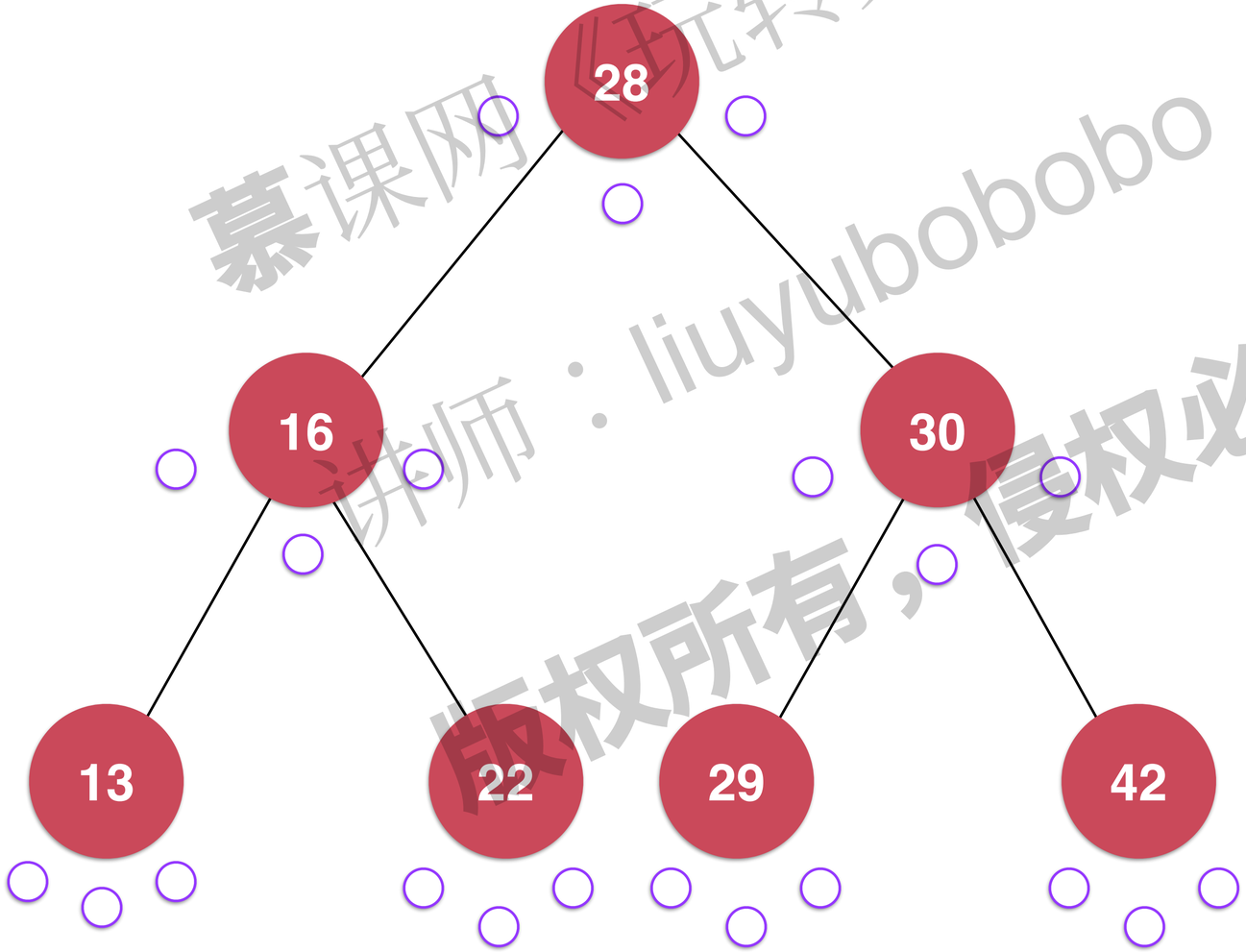
二分搜索树的前序遍历



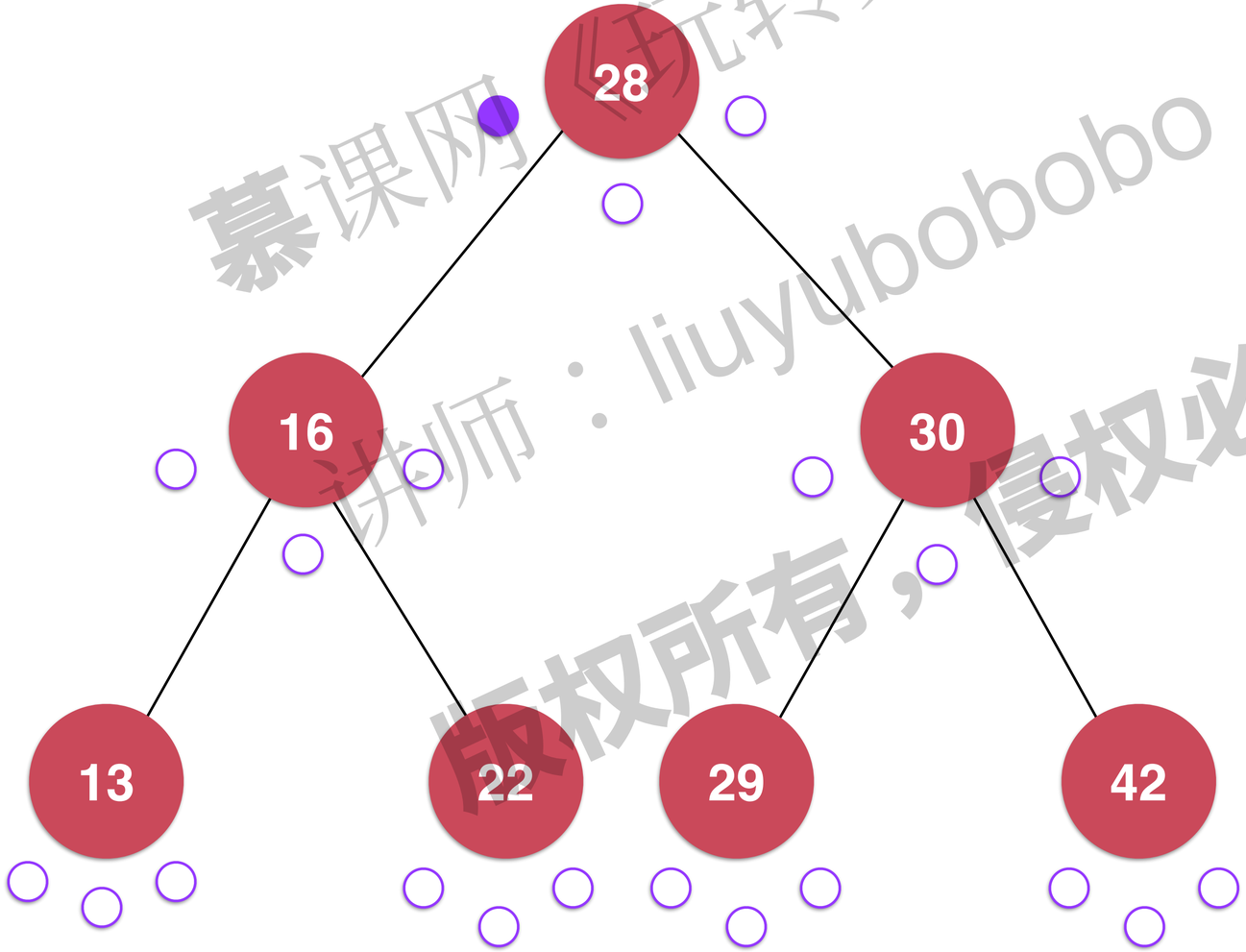
28
16
13
22
30
29
42

再看二分搜索树的中序遍历

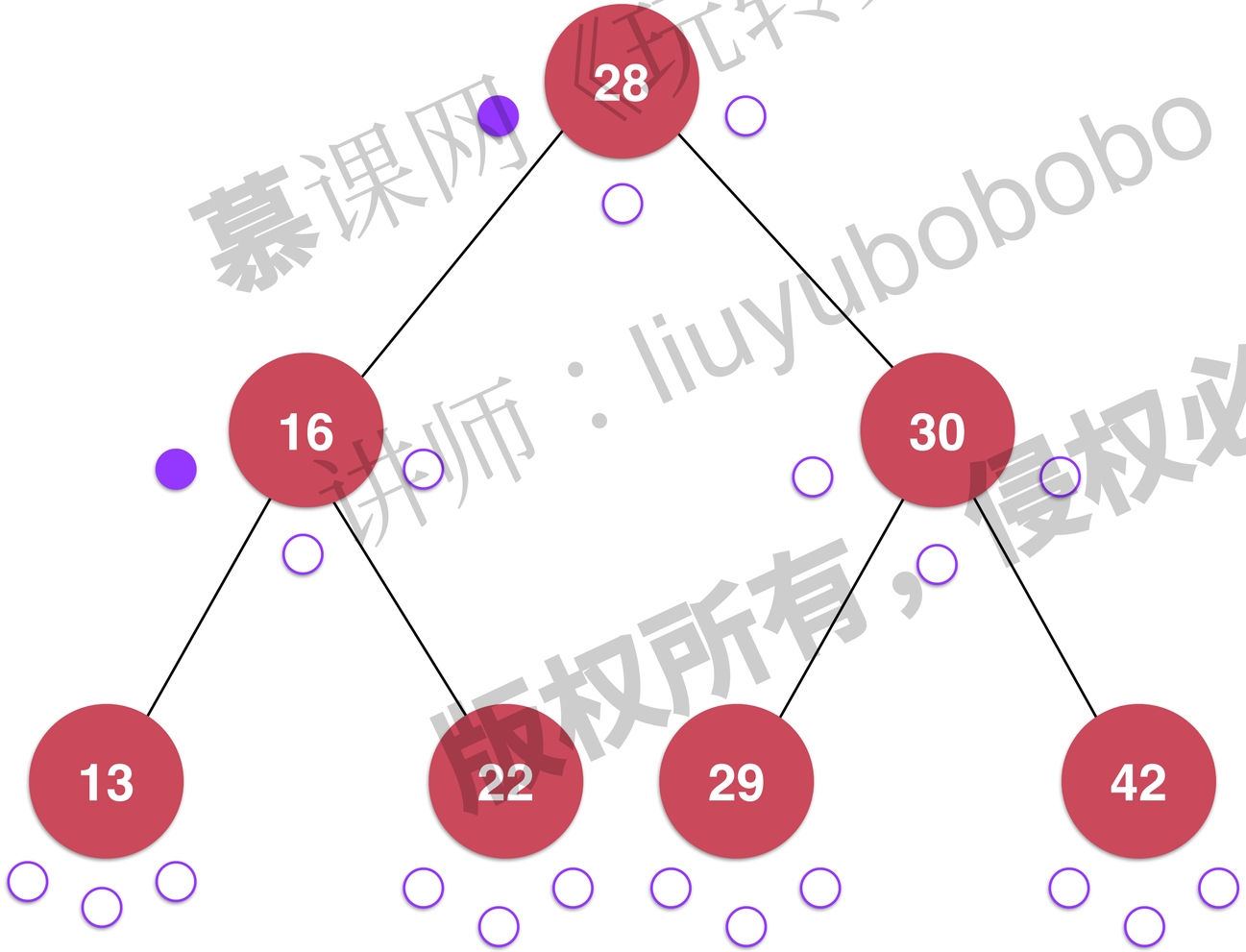
二分搜索树的中序遍历



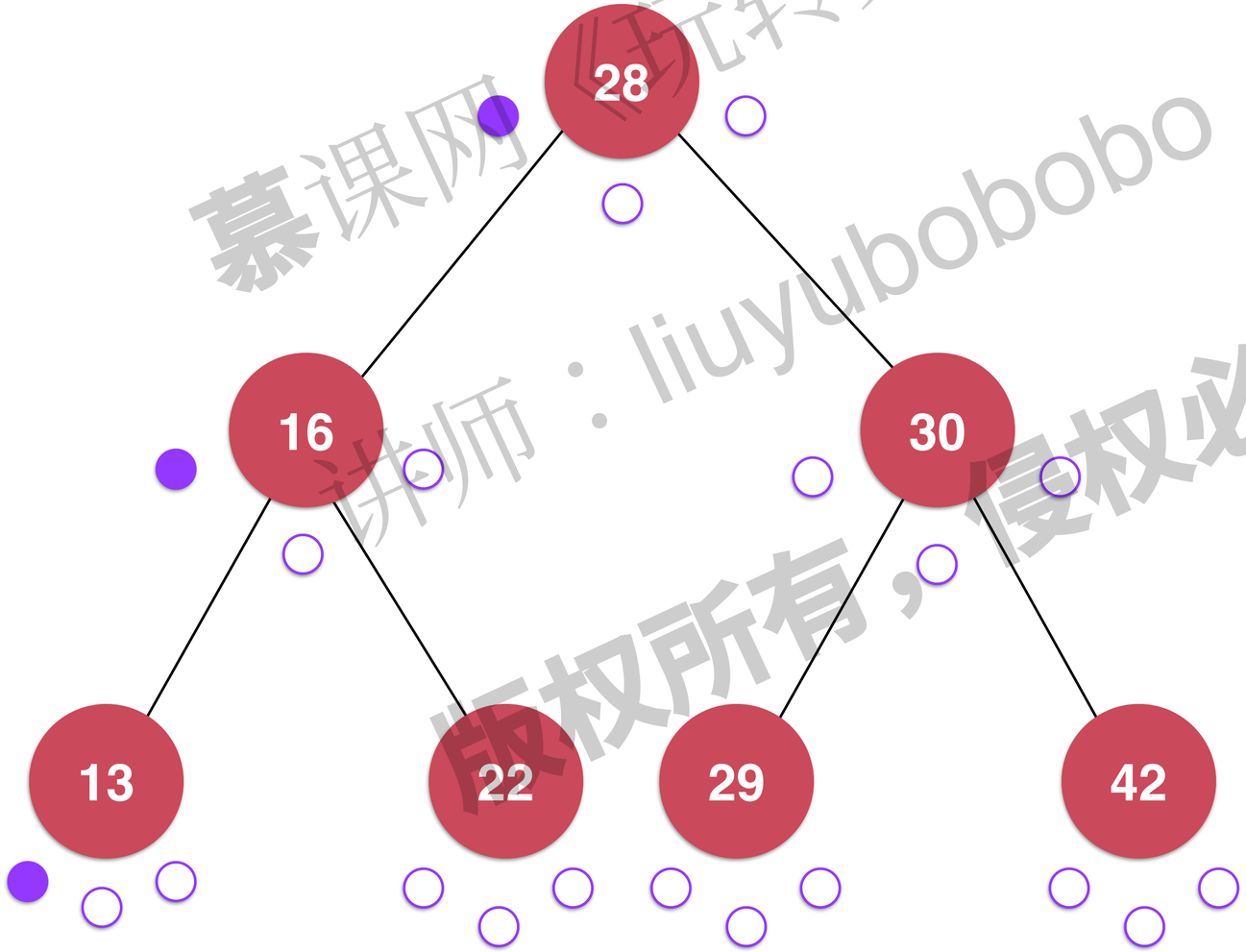
二分搜索树的中序遍历



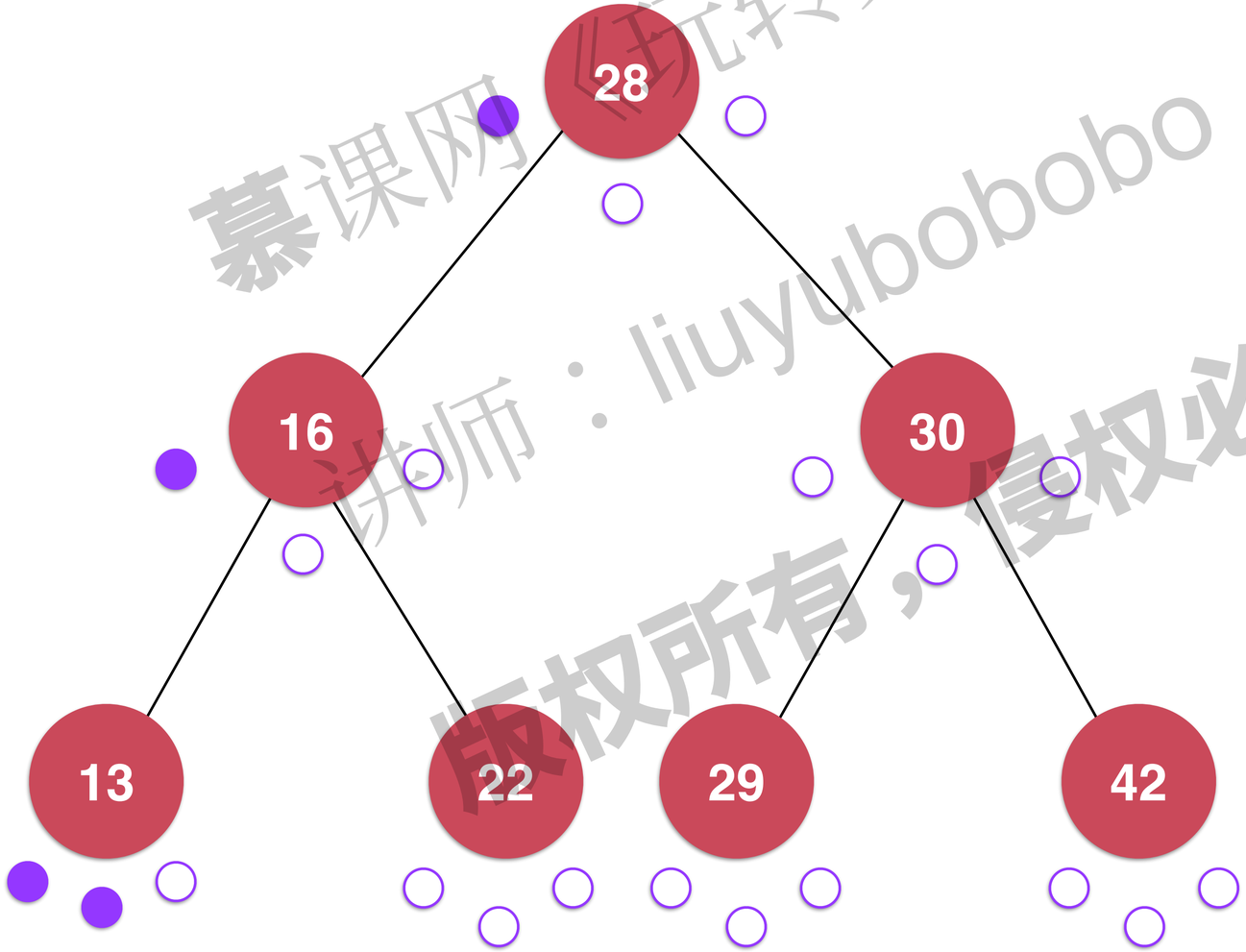
二分搜索树的中序遍历



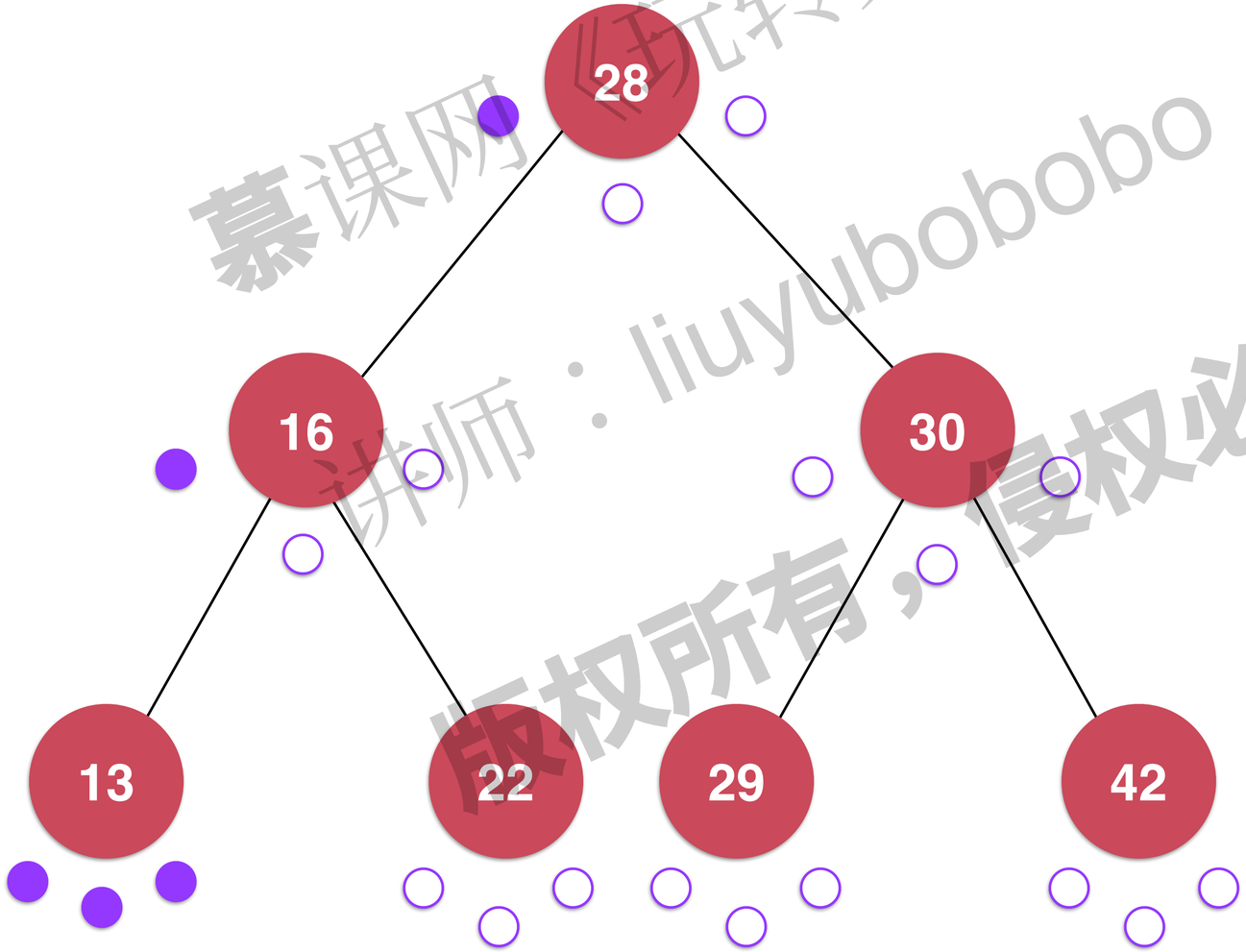
二分搜索树的中序遍历



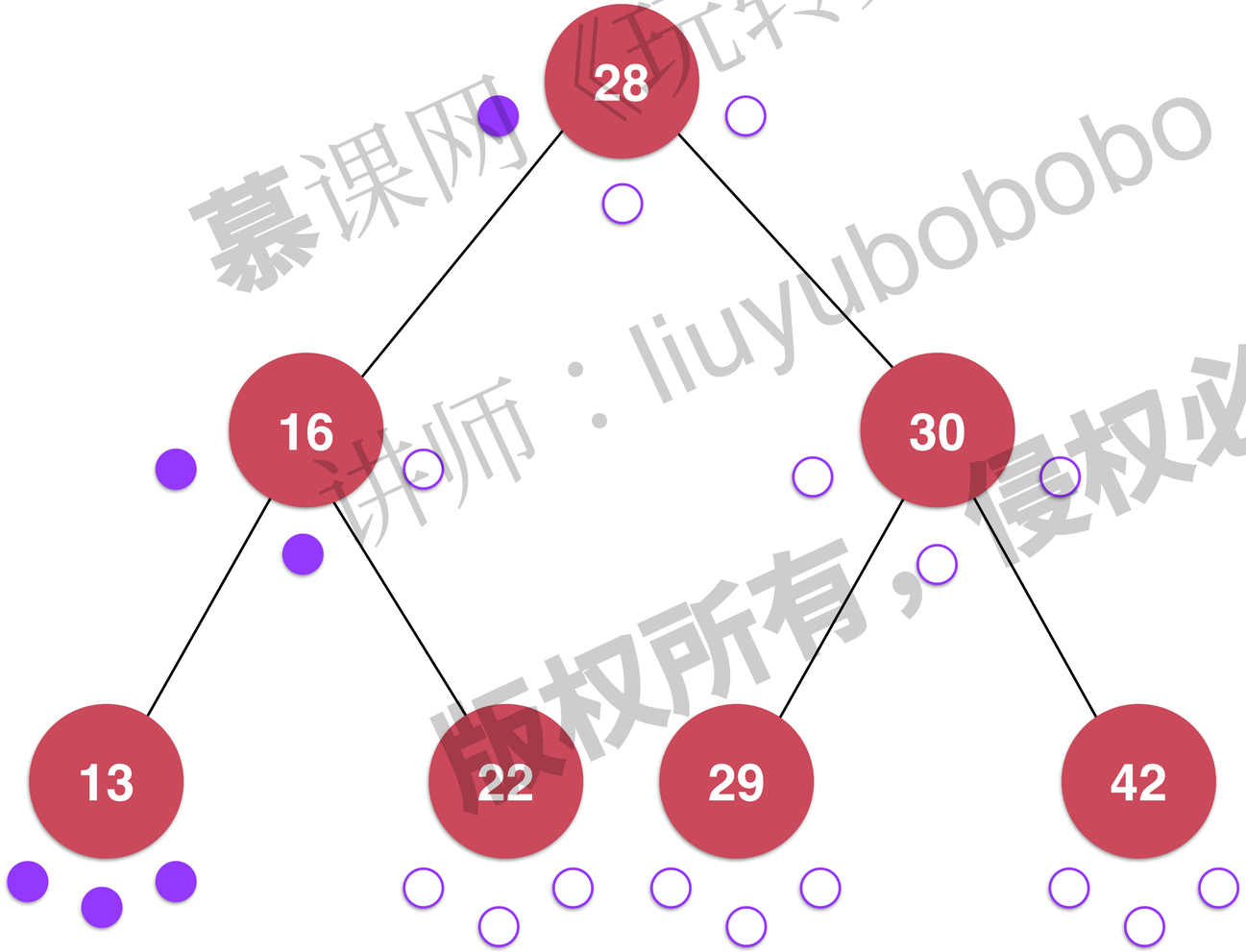
二分搜索树的中序遍历



二分搜索树的中序遍历



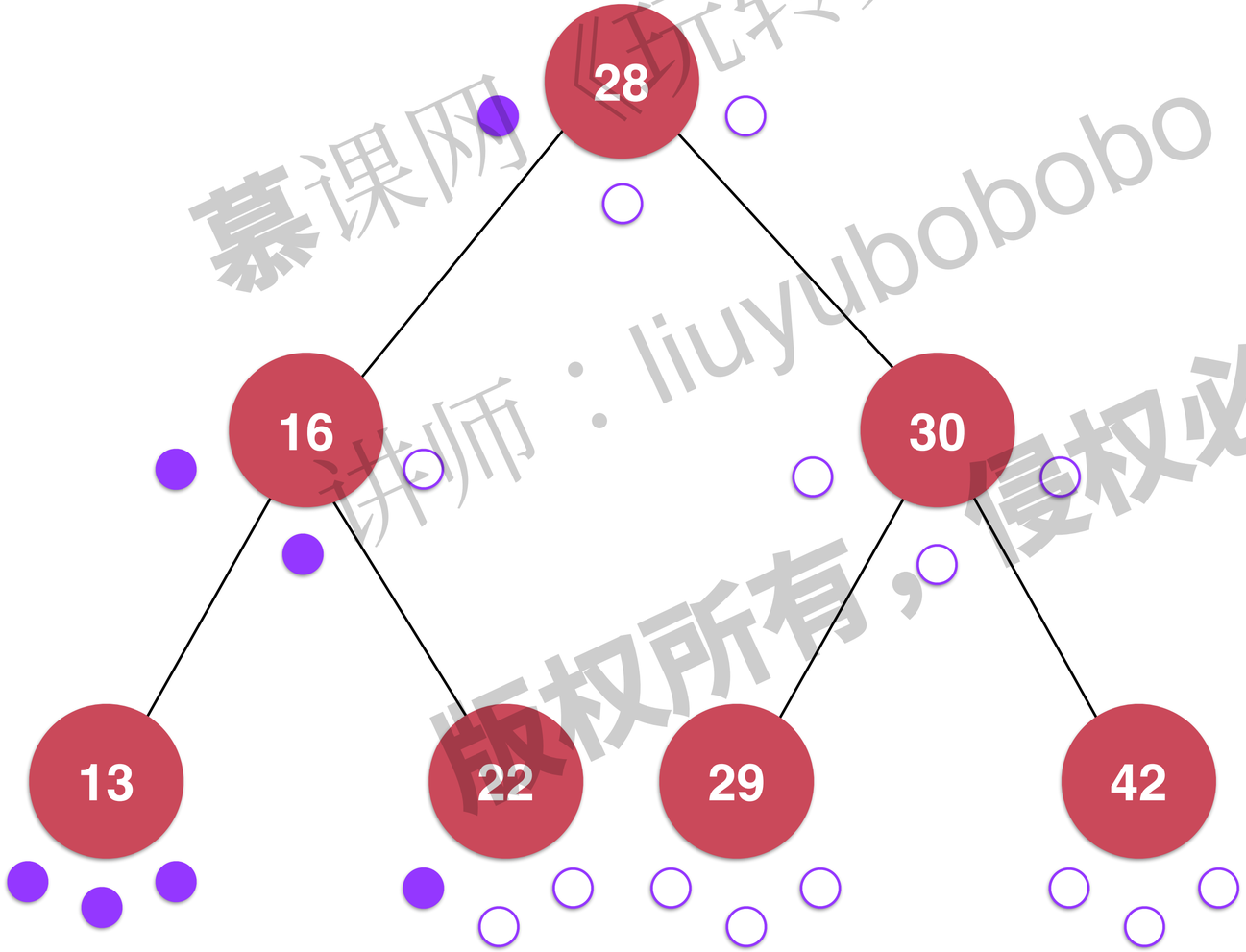
二分搜索树的中序遍历



13

16

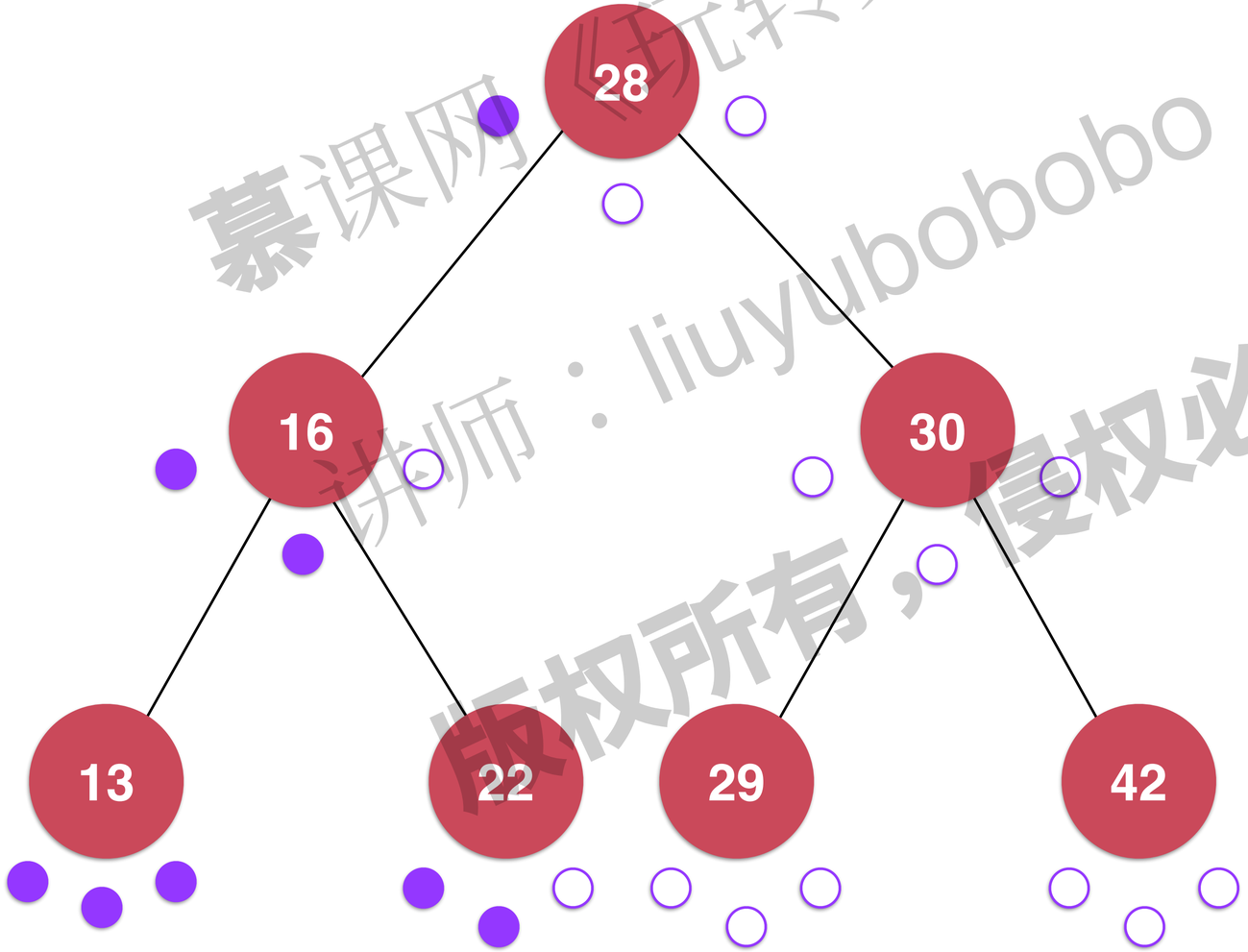
二分搜索树的中序遍历



13

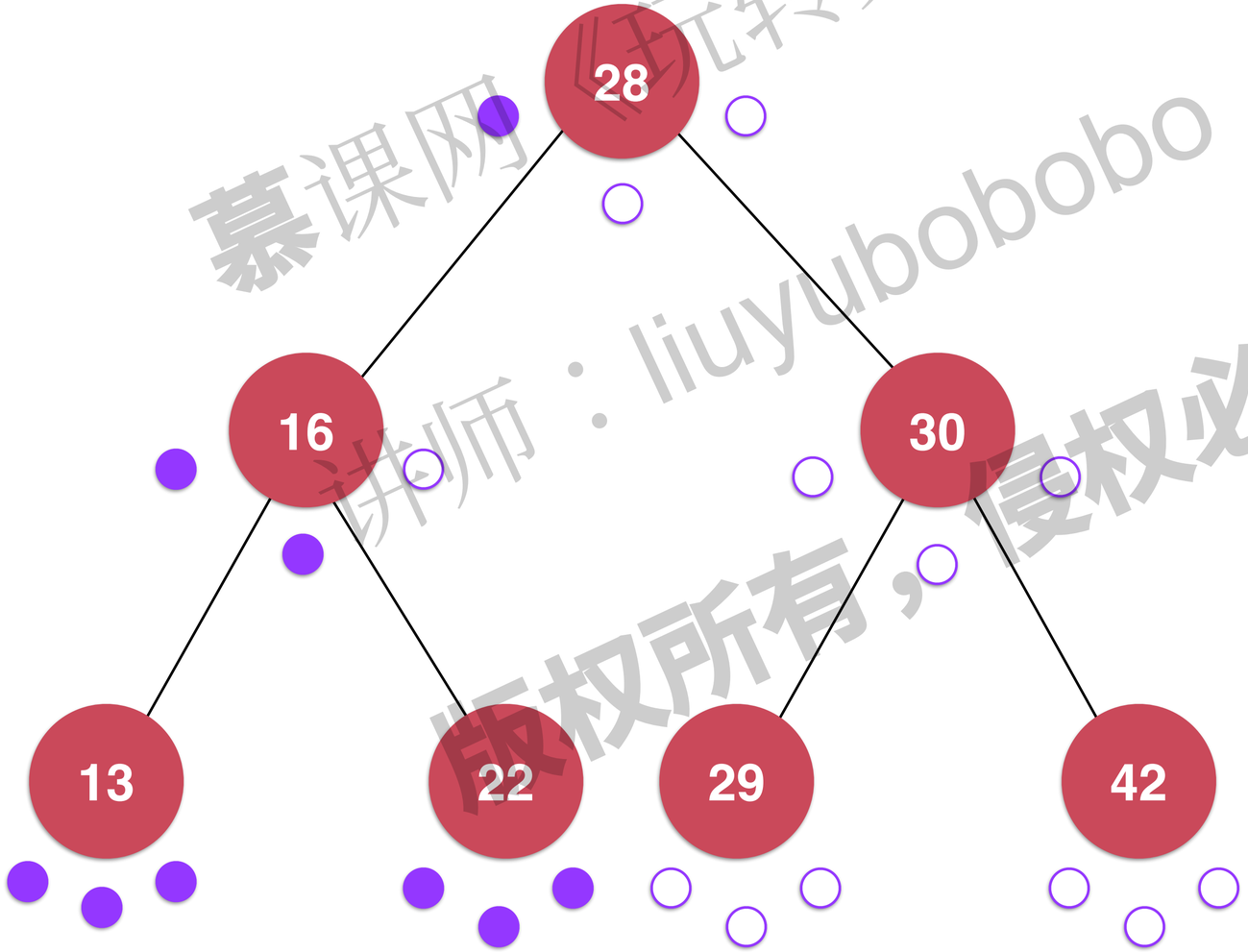
16

二分搜索树的中序遍历



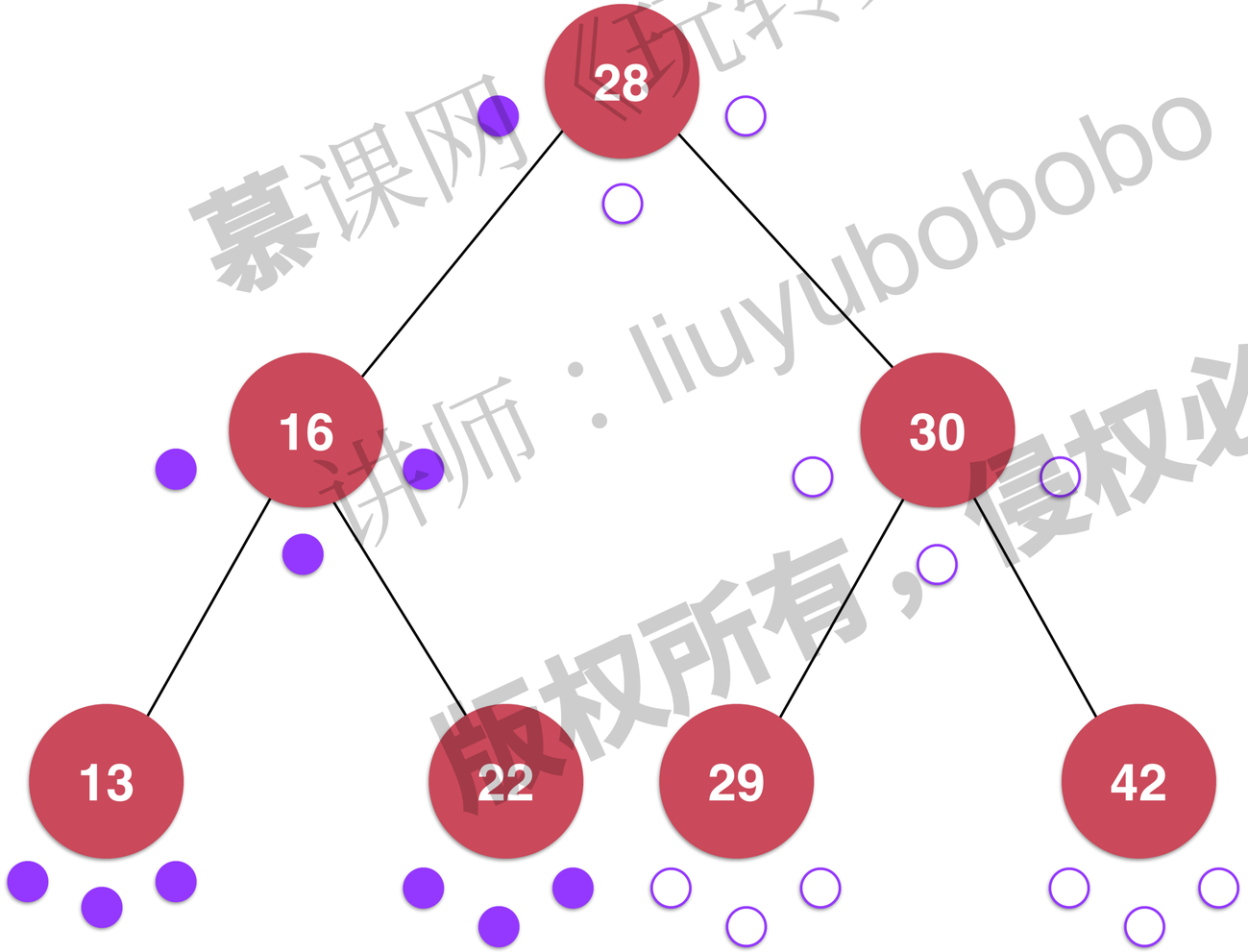
13
16
22

二分搜索树的中序遍历



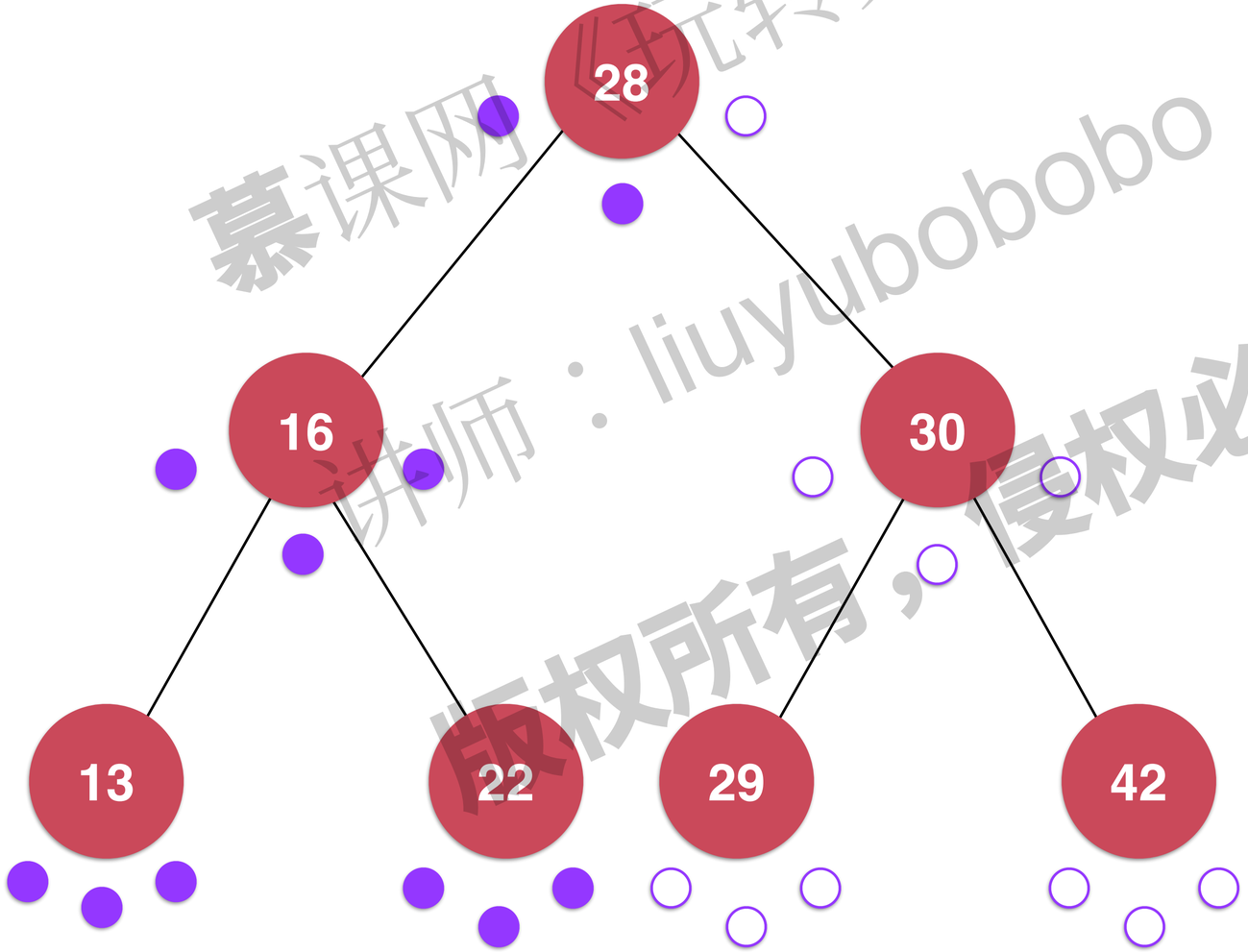
13
16
22

二分搜索树的中序遍历



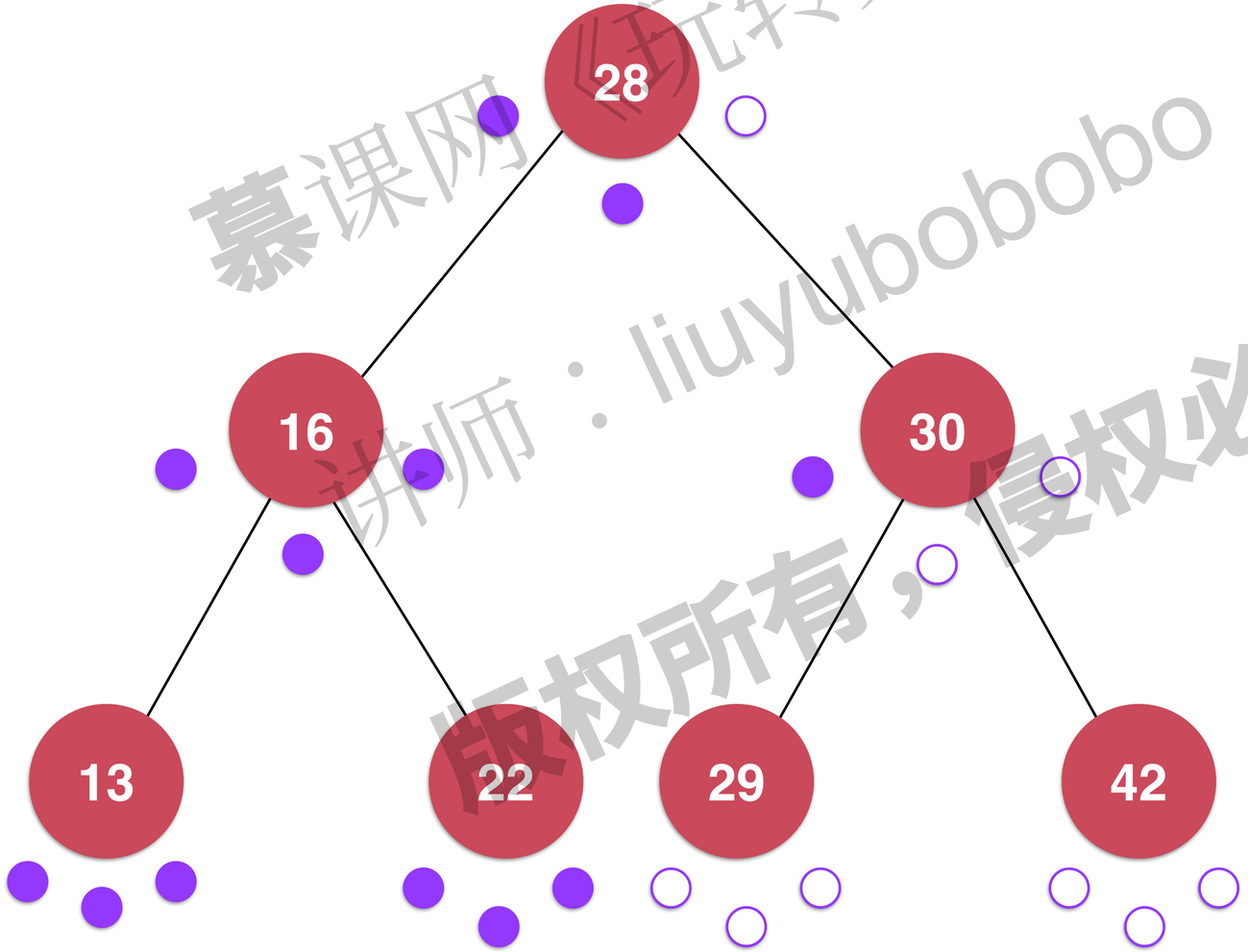
13
16
22

二分搜索树的中序遍历



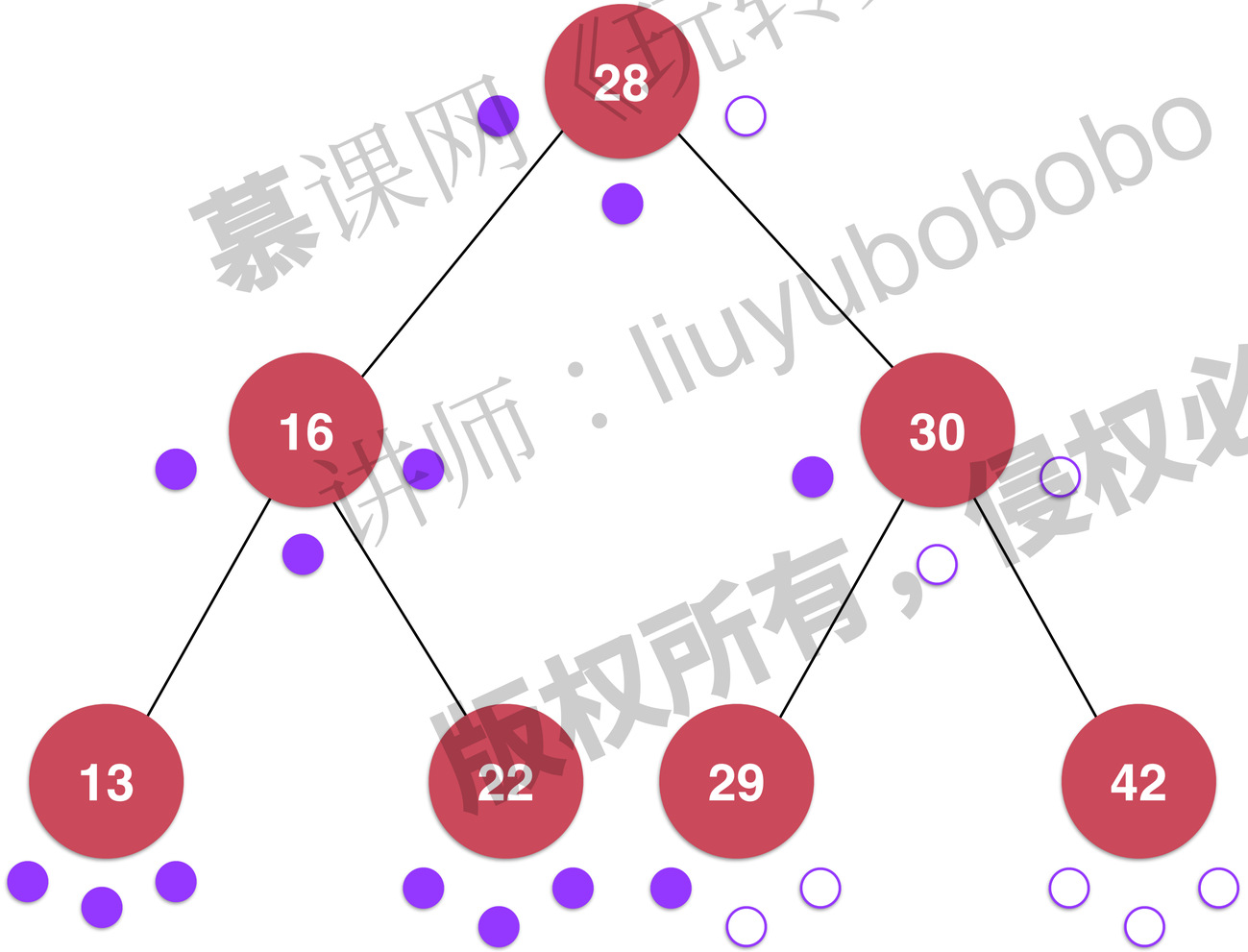
13
16
22
28

二分搜索树的中序遍历



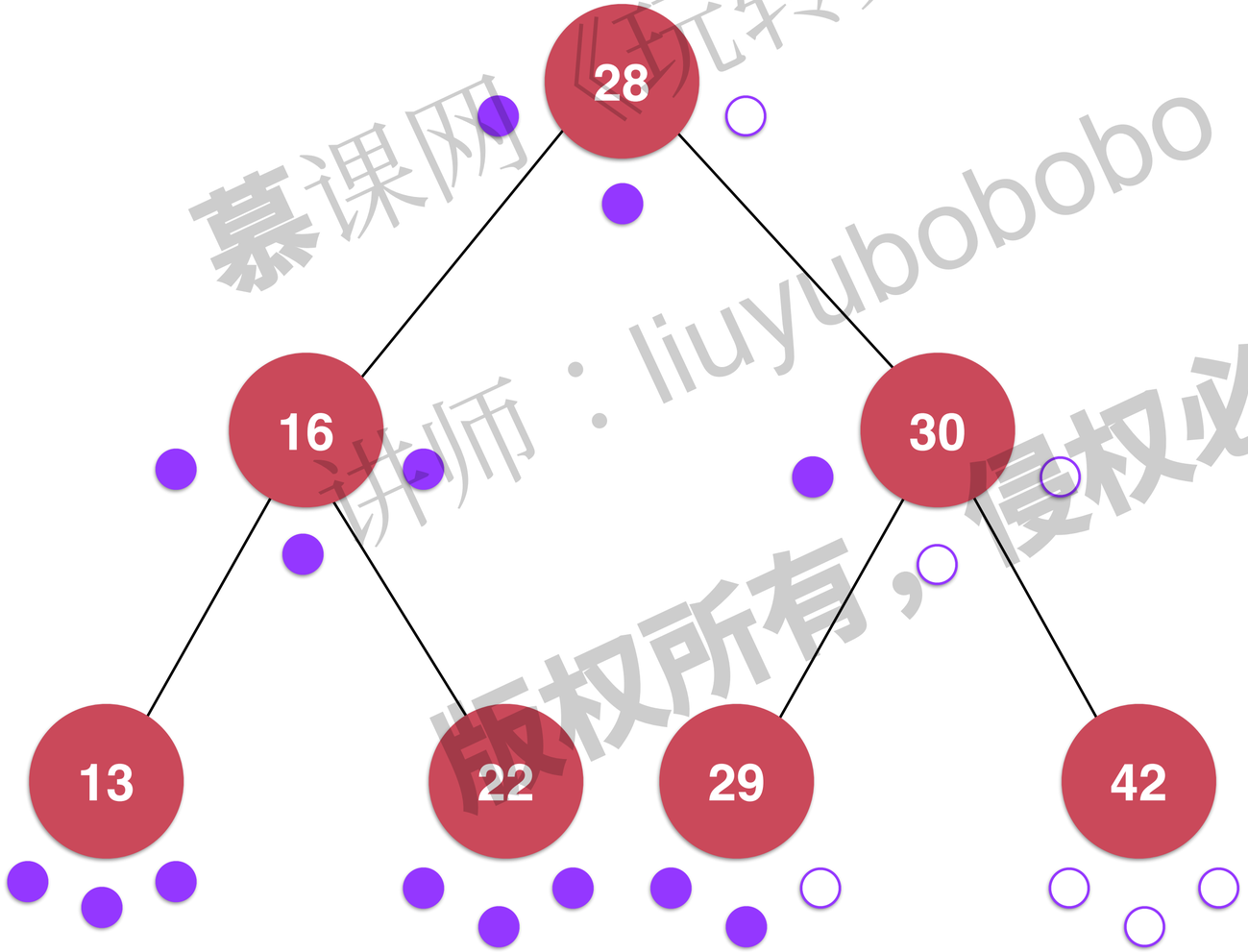
13
16
22
28

二分搜索树的中序遍历



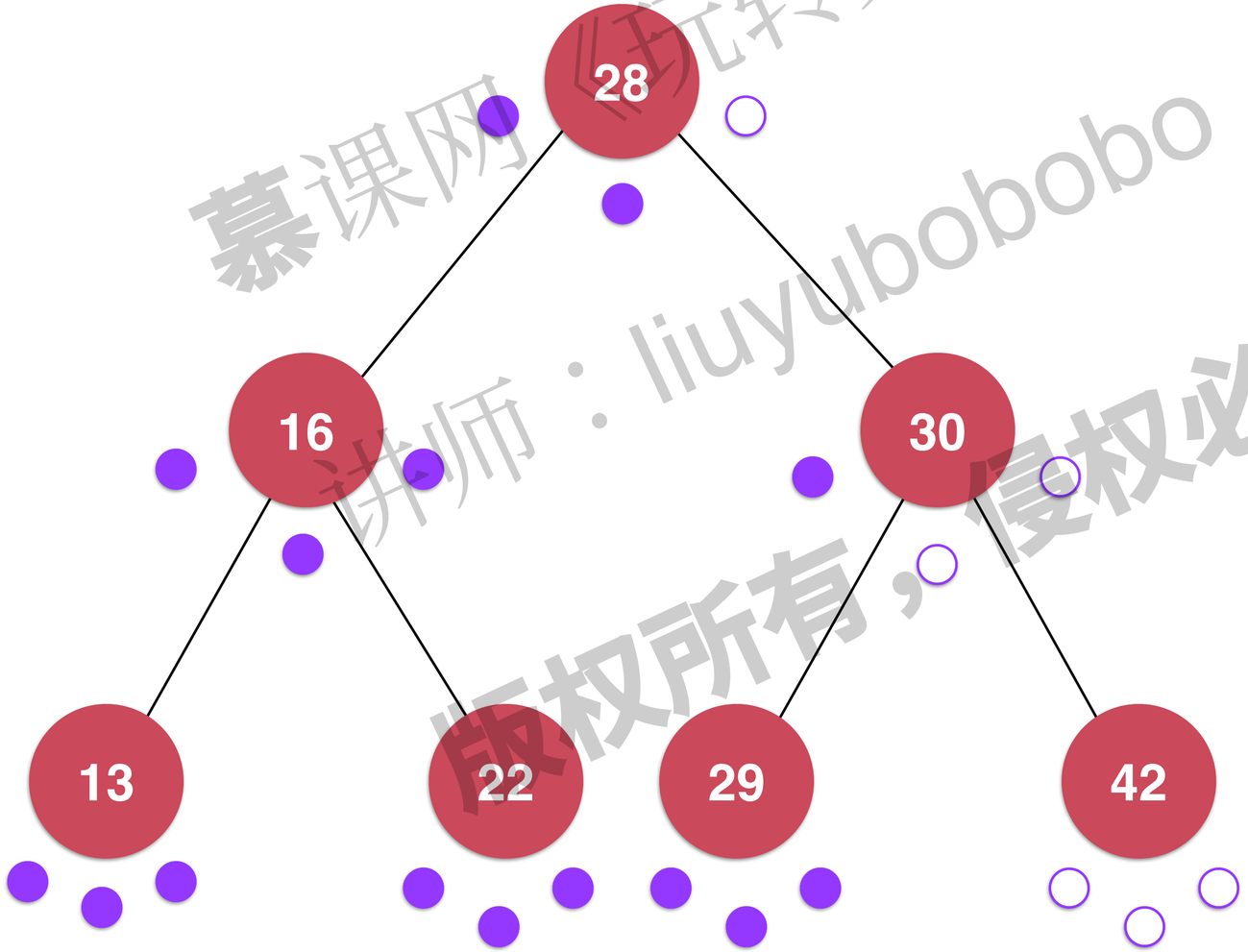
13
16
22
28

二分搜索树的中序遍历



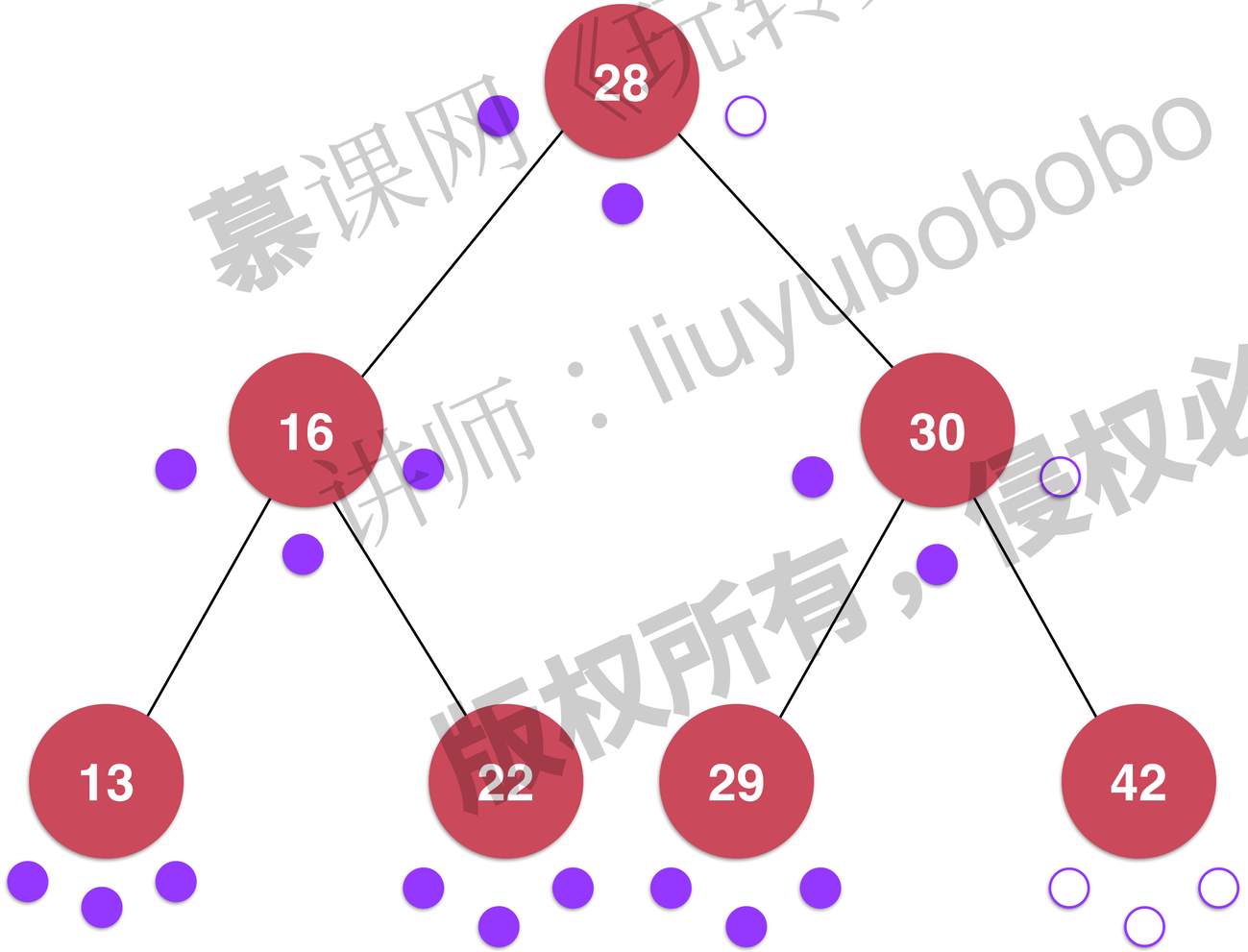
13
16
22
28
29

二分搜索树的中序遍历



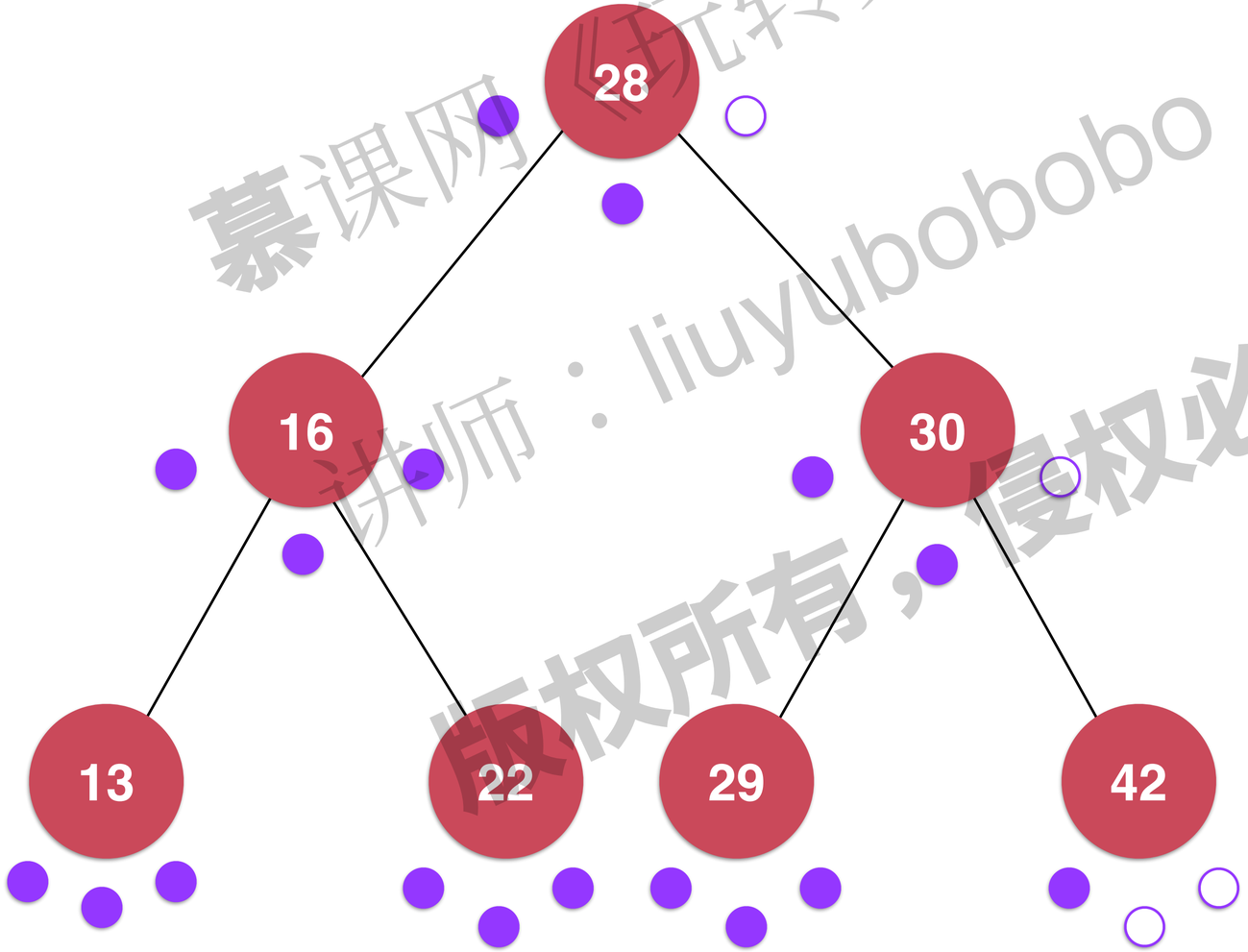
13
16
22
28
29

二分搜索树的中序遍历



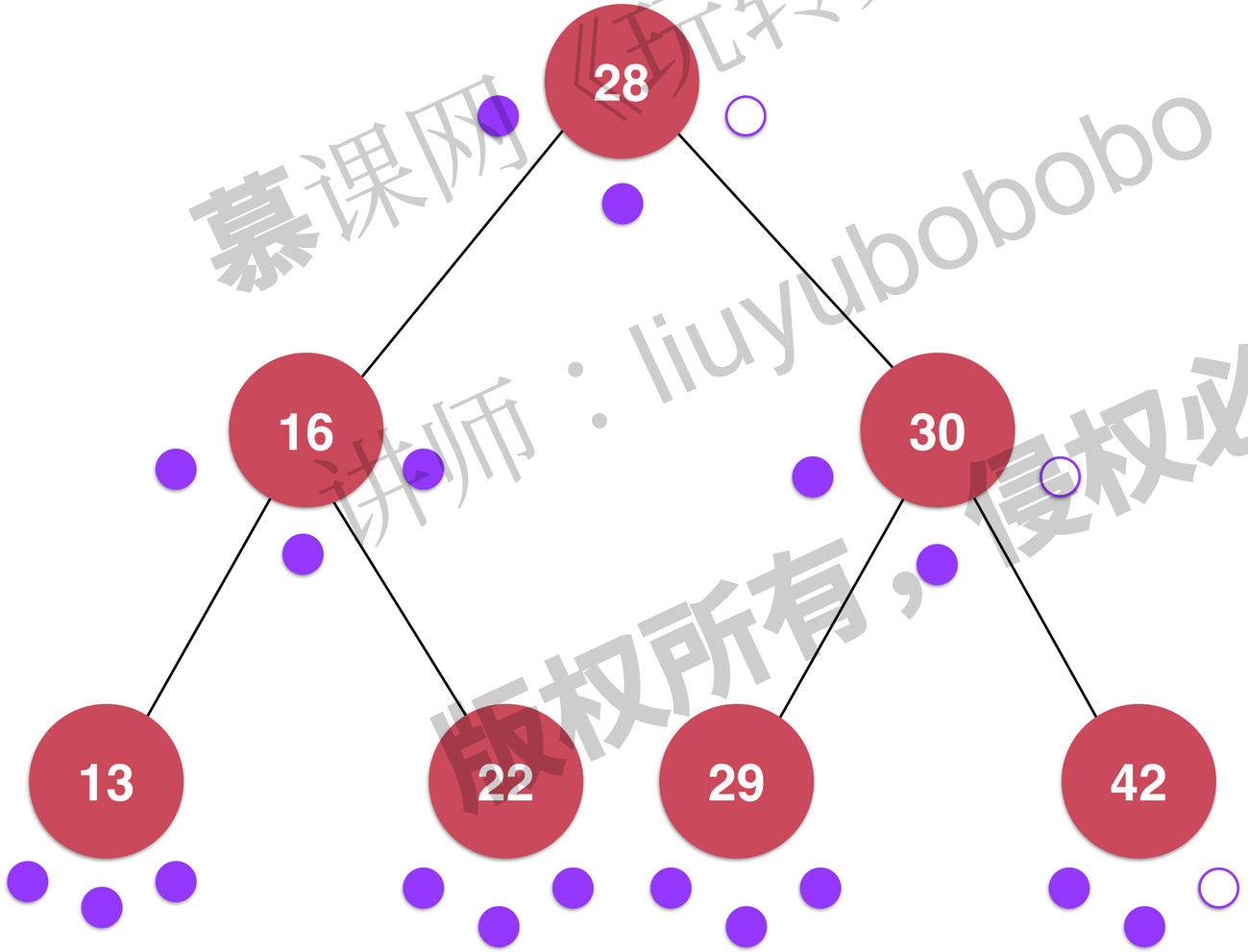
13
16
22
28
29
30

二分搜索树的中序遍历



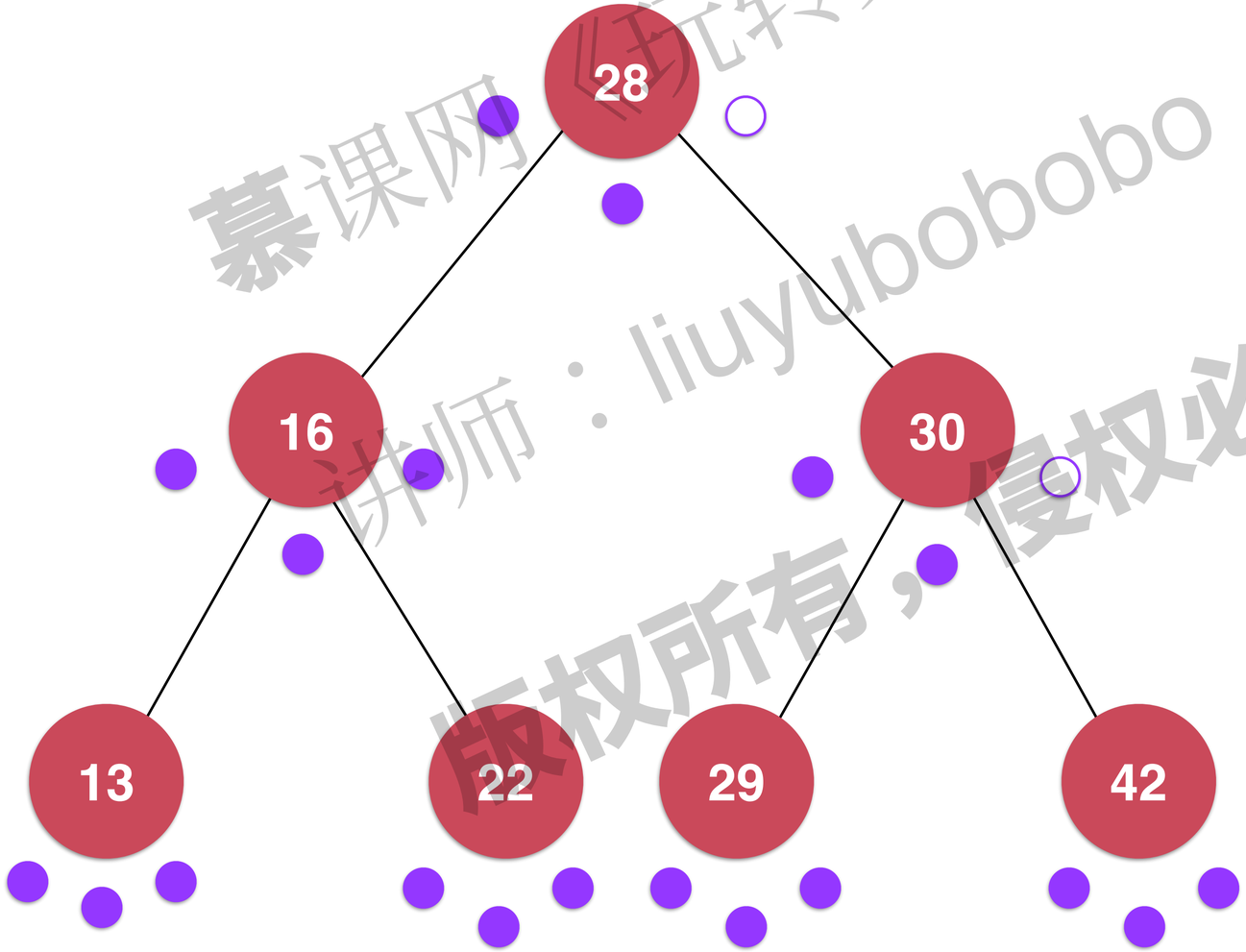
13
16
22
28
29
30

二分搜索树的中序遍历



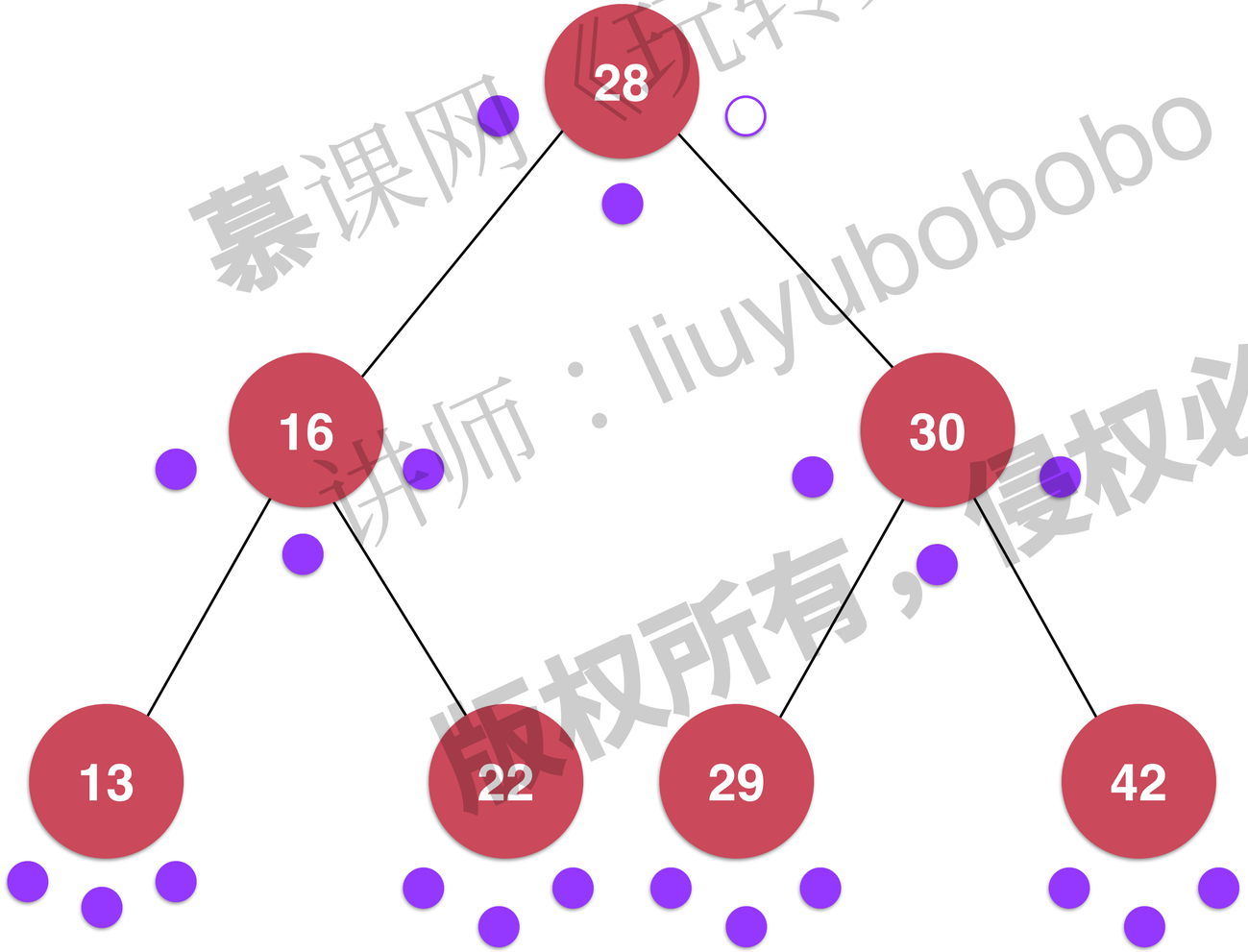
13
16
22
28
29
30
42

二分搜索树的中序遍历



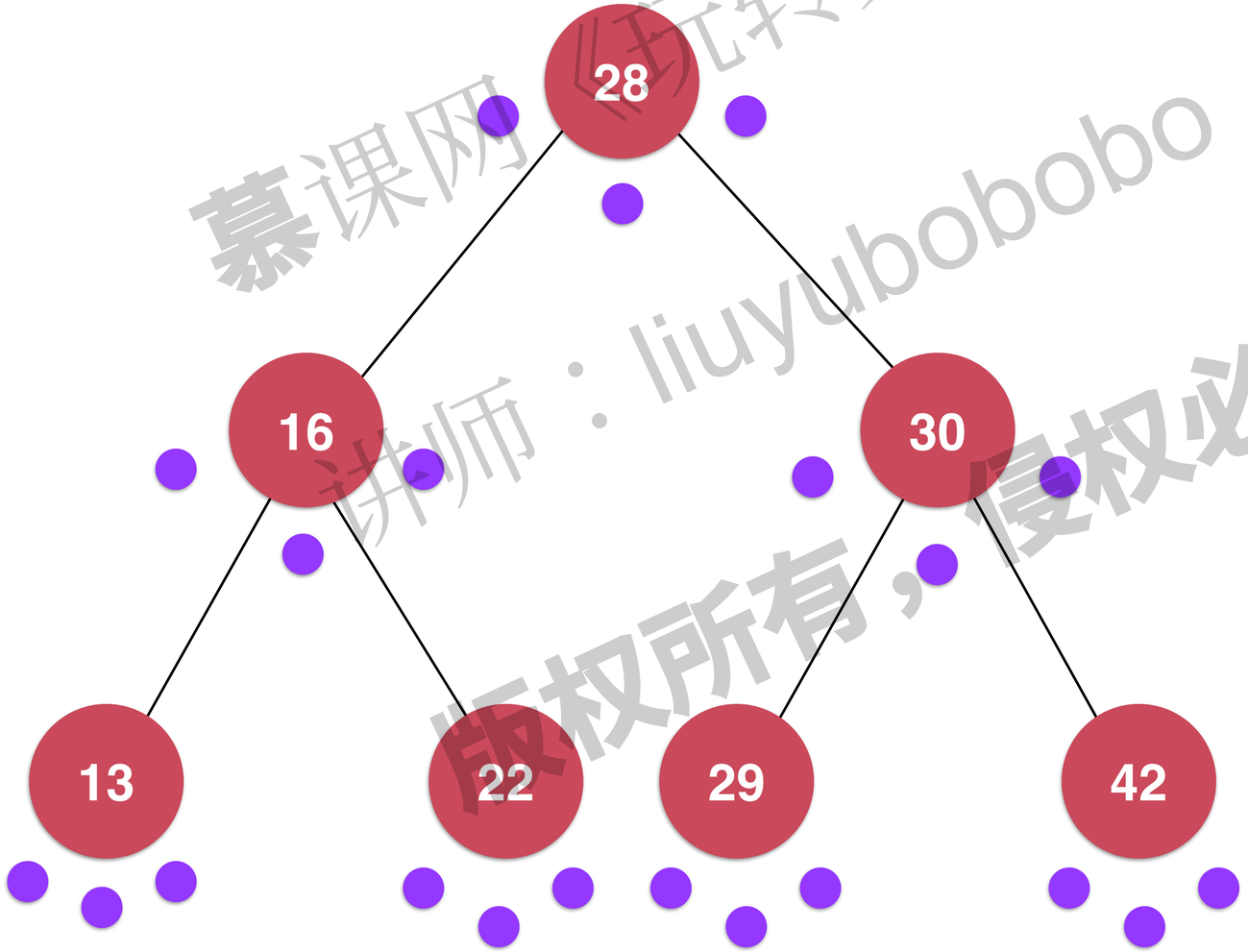
13
16
22
28
29
30
42

二分搜索树的中序遍历



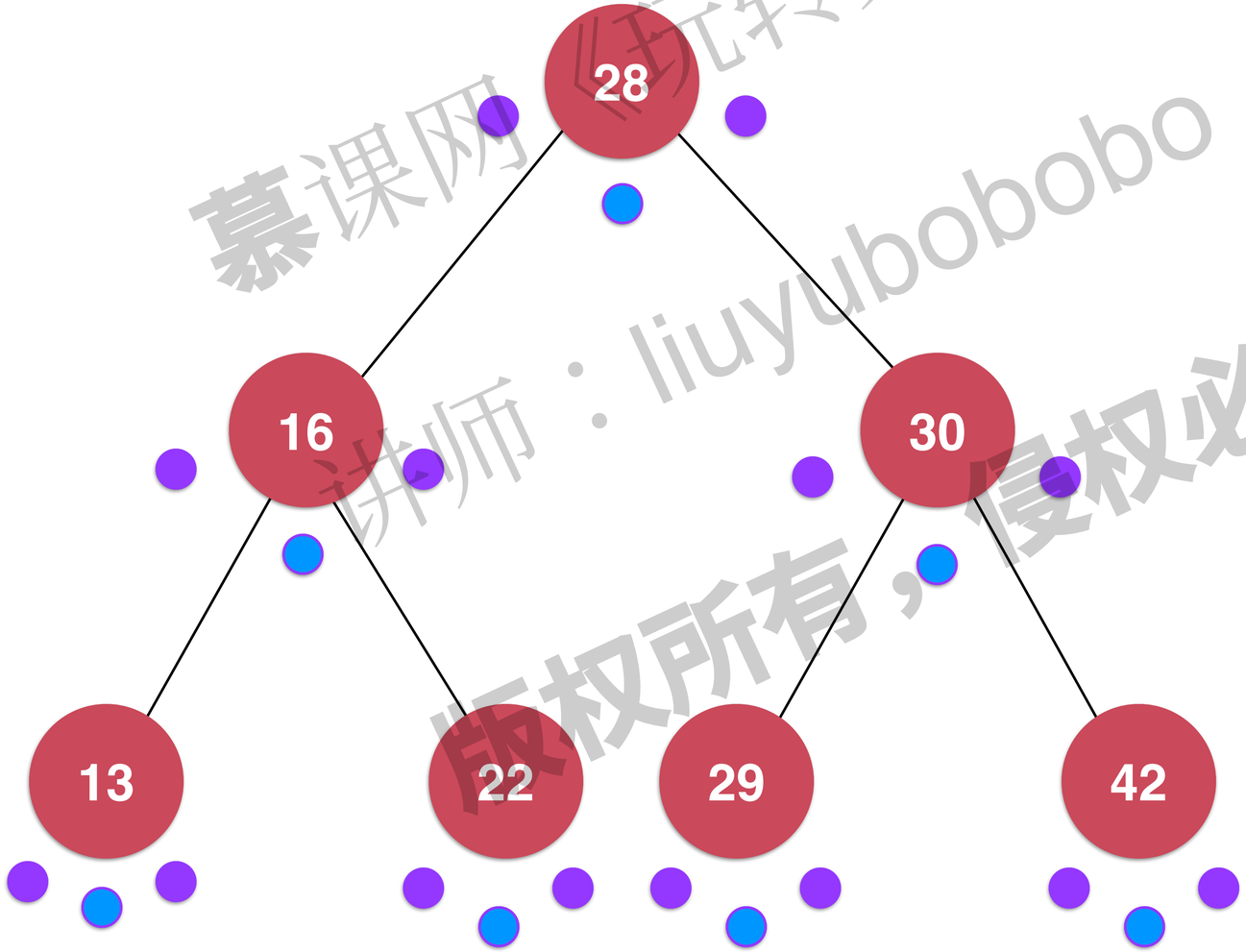
13
16
22
28
29
30
42

二分搜索树的中序遍历



13
16
22
28
29
30
42

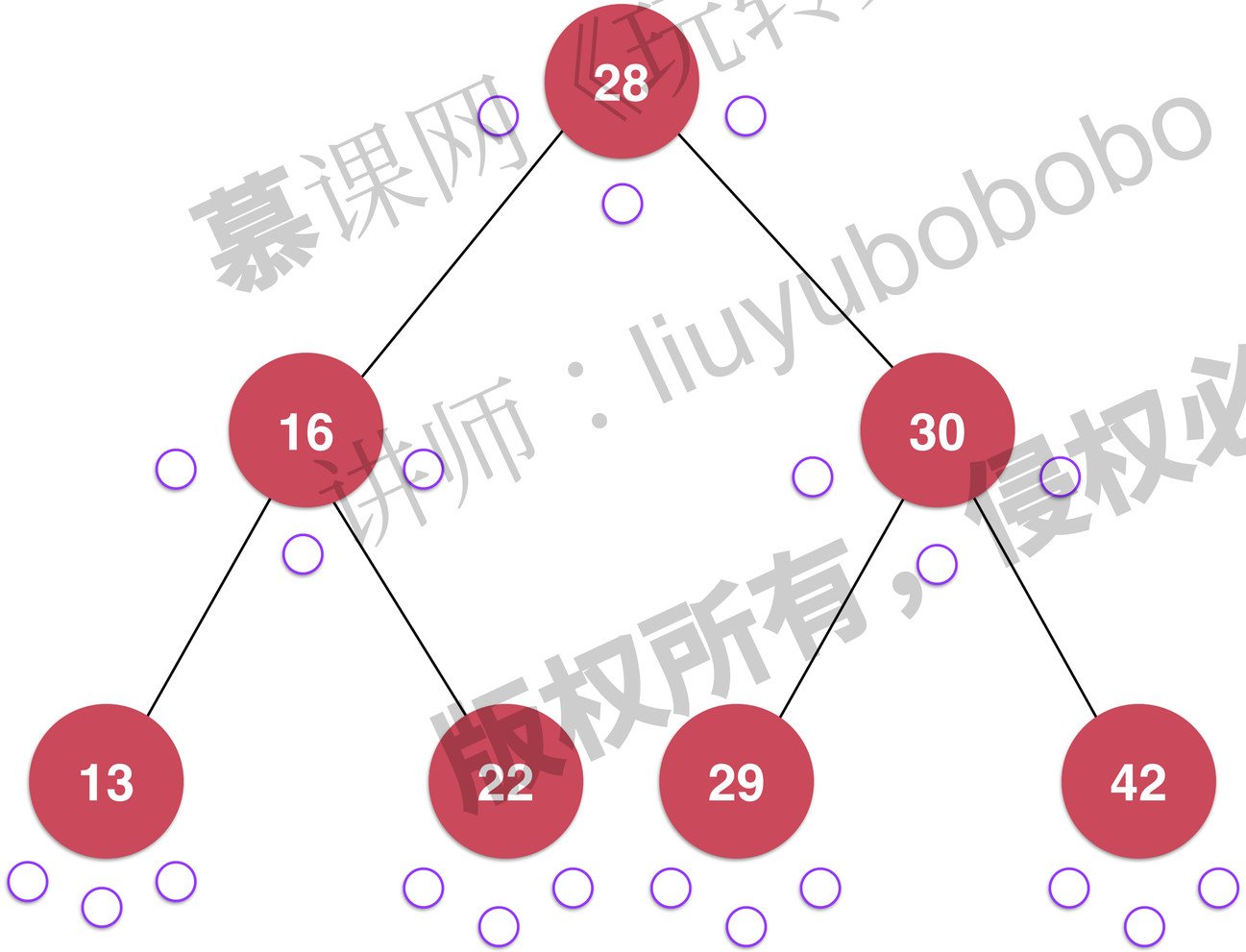
二分搜索树的中序遍历



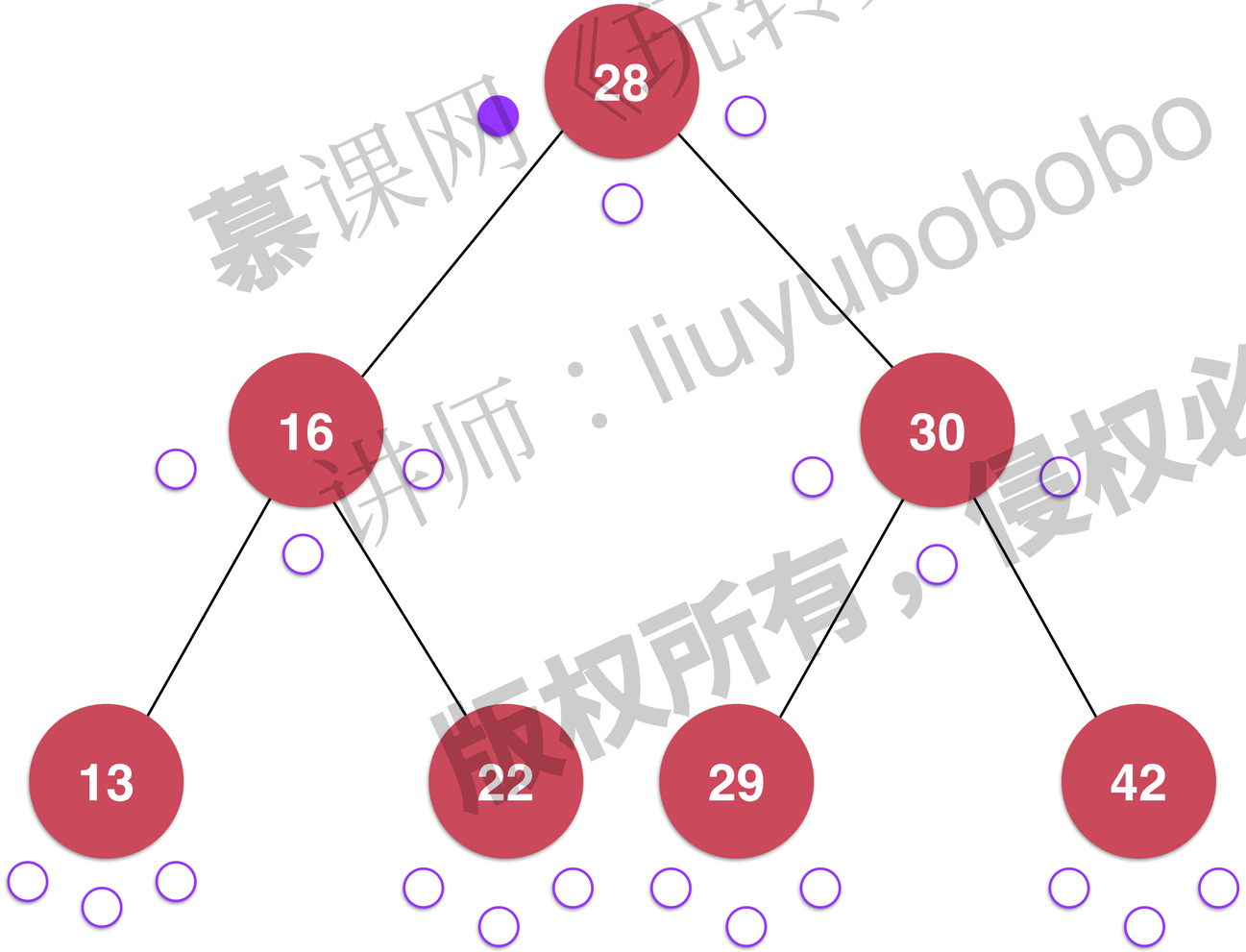
13
16
22
28
29
30
42

再看二分搜索树的后序遍历

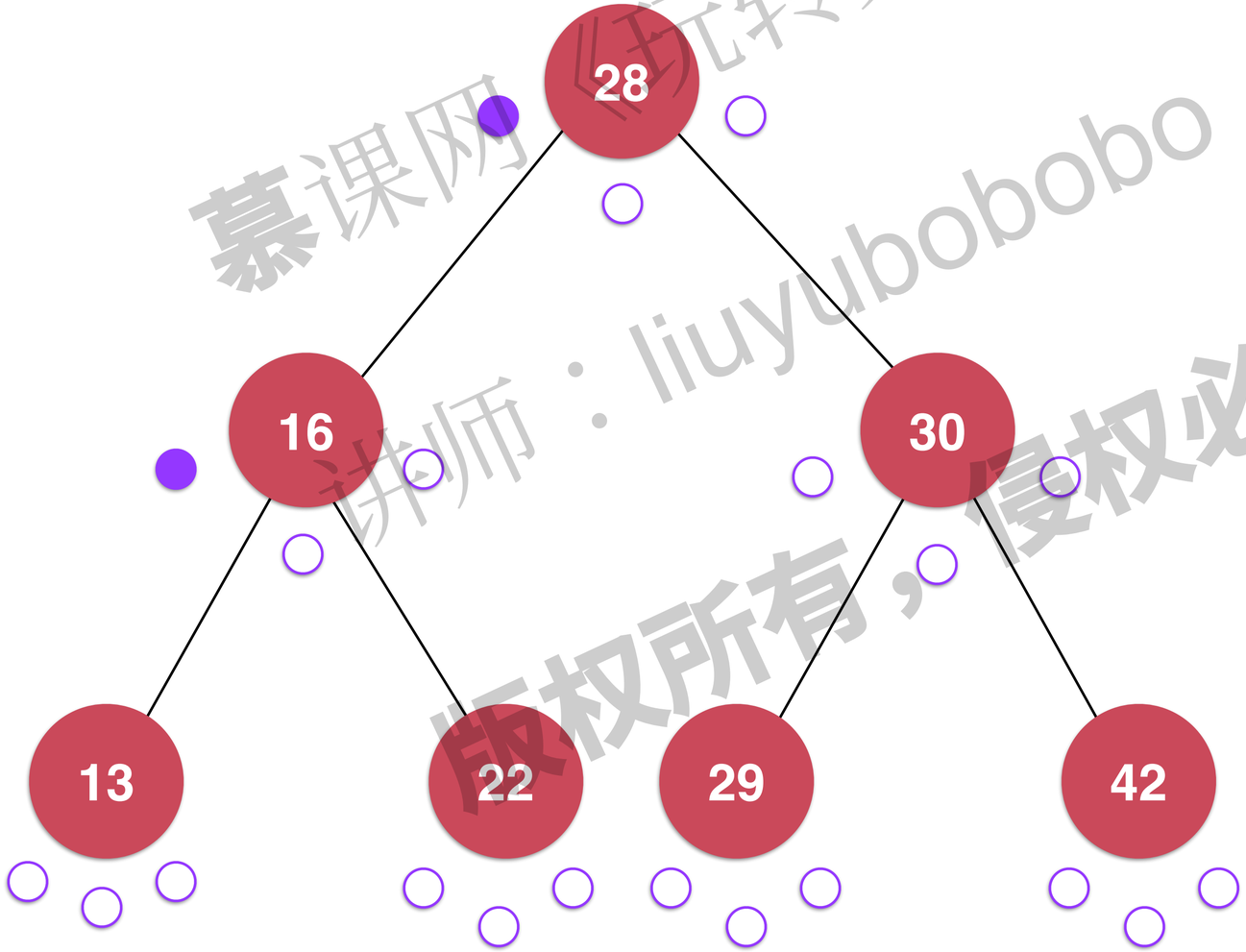
二分搜索树的后序遍历



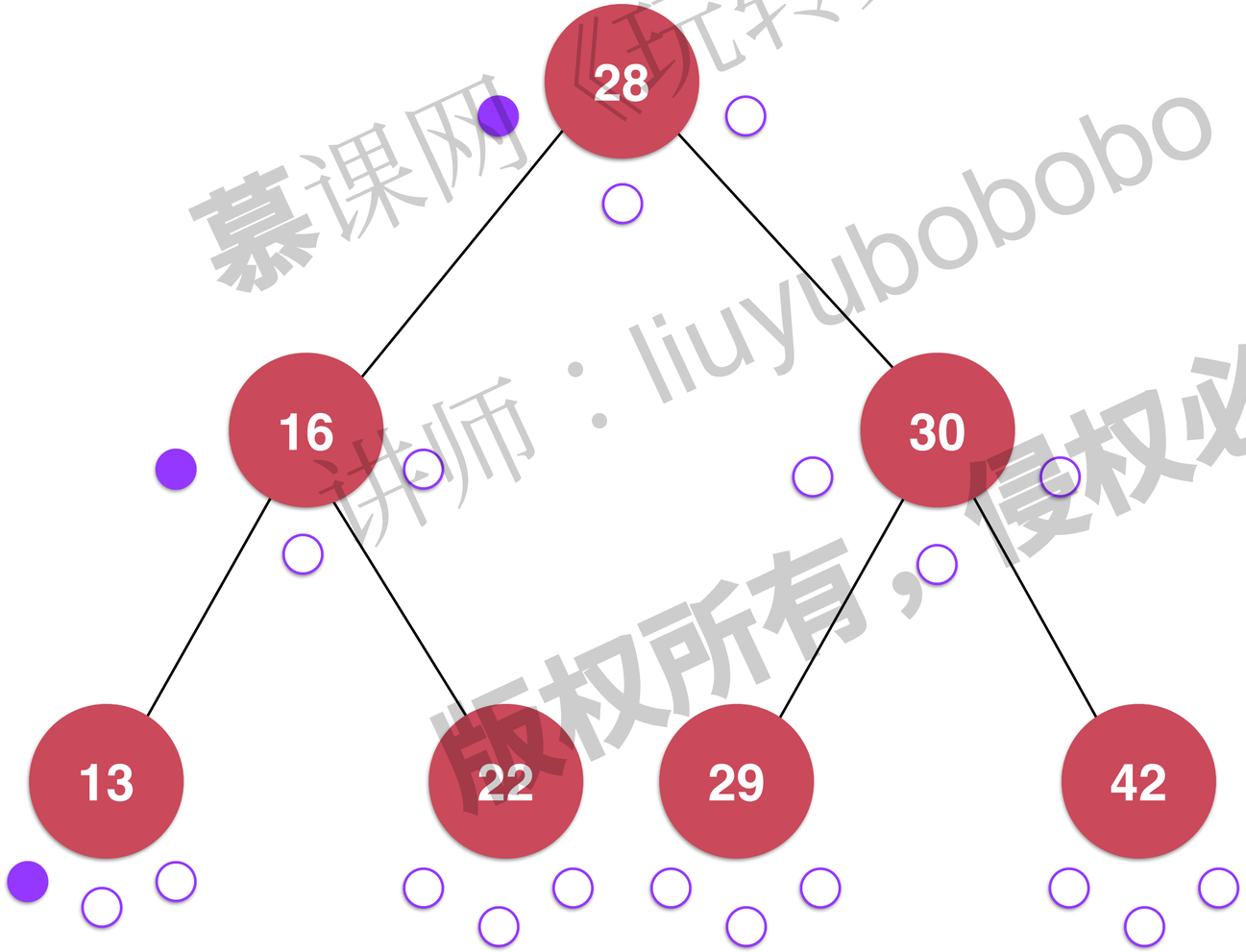
二分搜索树的后序遍历



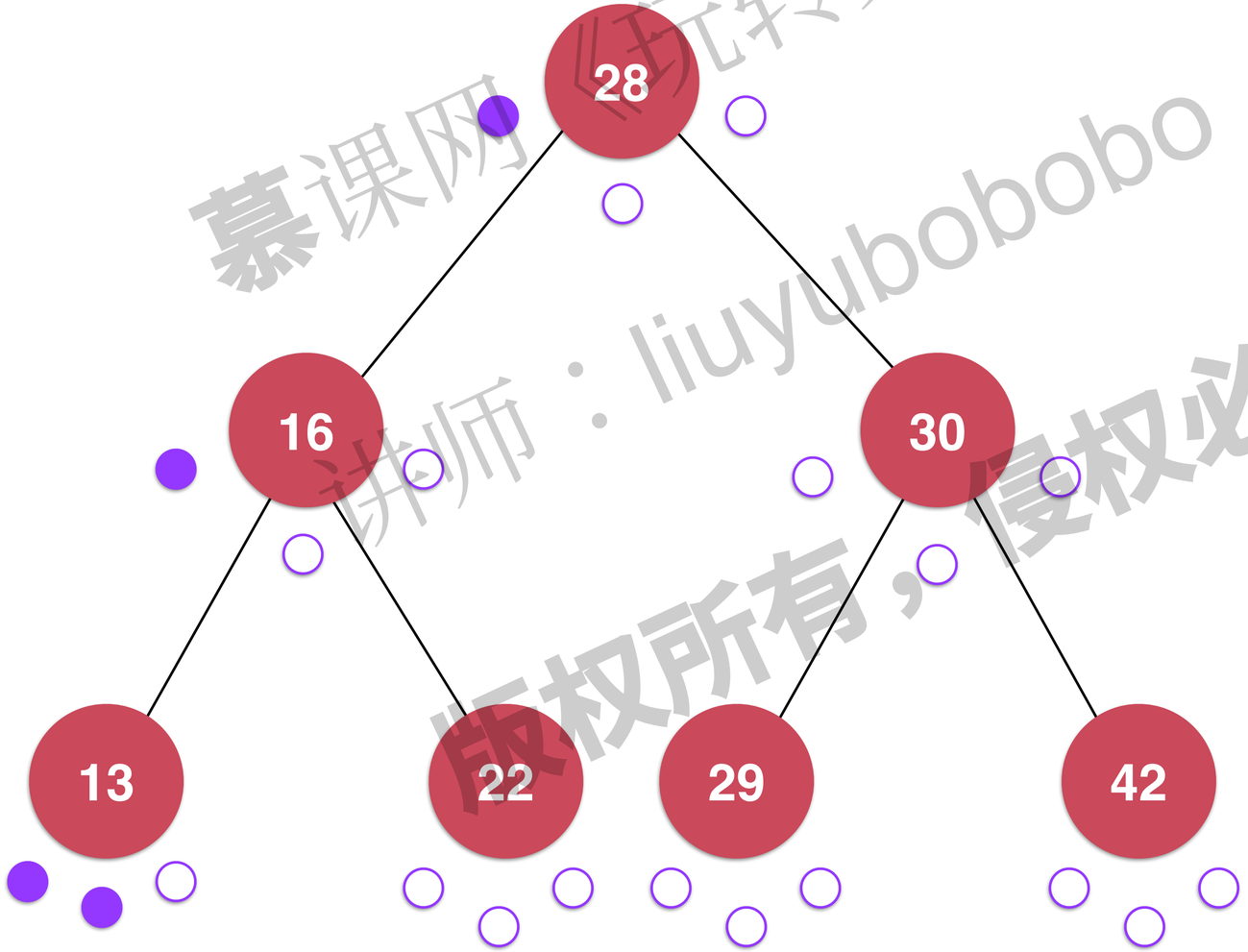
二分搜索树的后序遍历



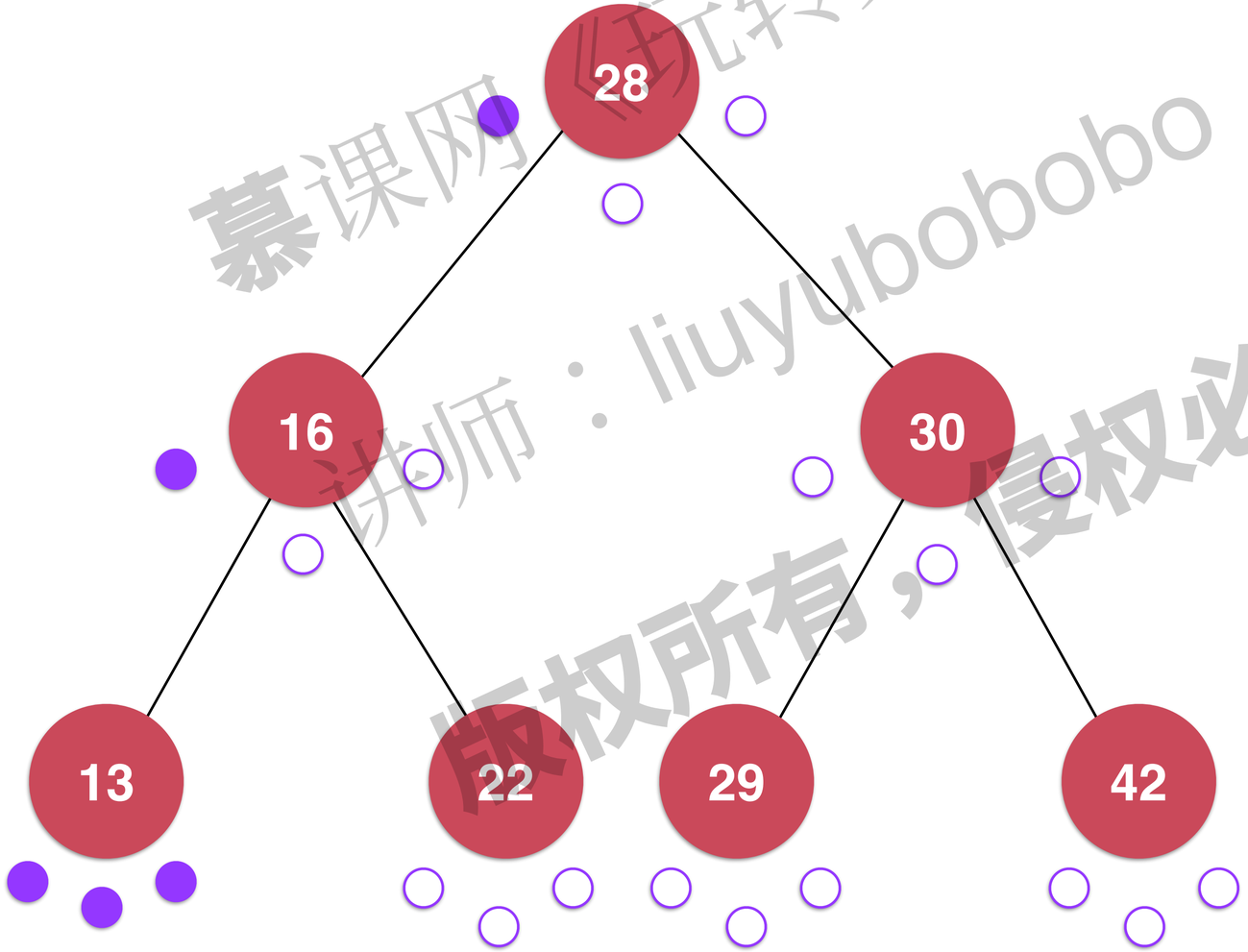
二分搜索树的后序遍历



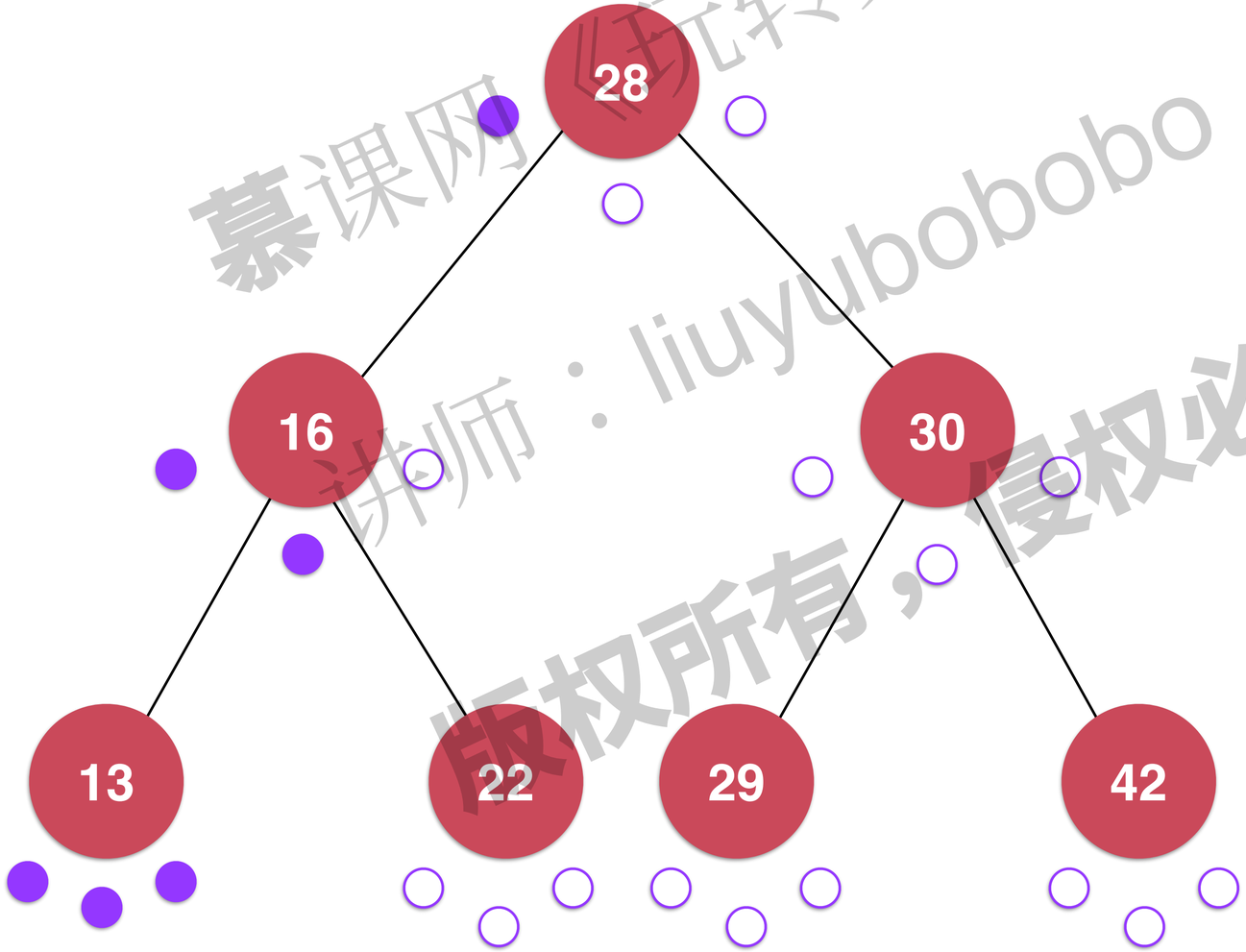
二分搜索树的后序遍历



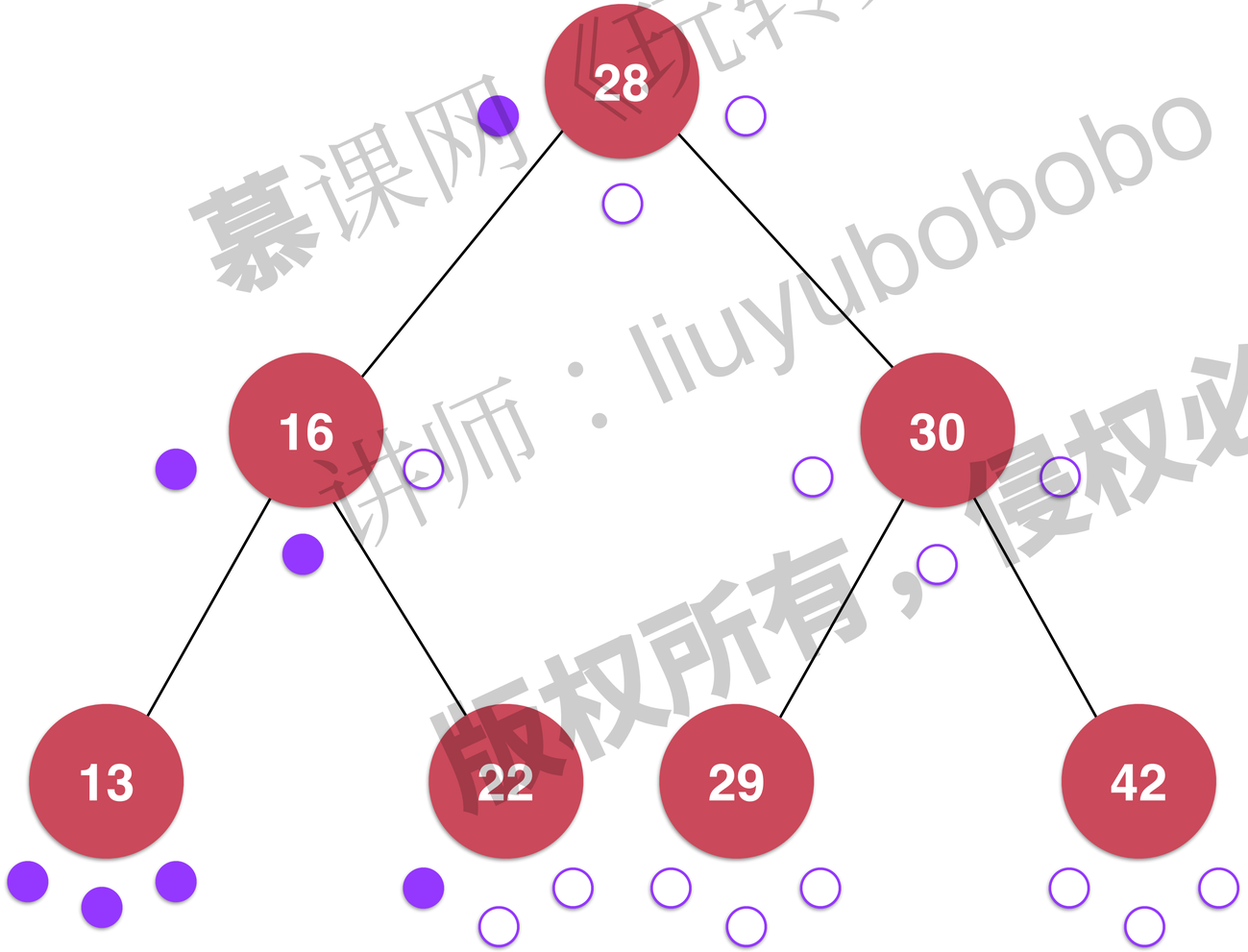
二分搜索树的后序遍历



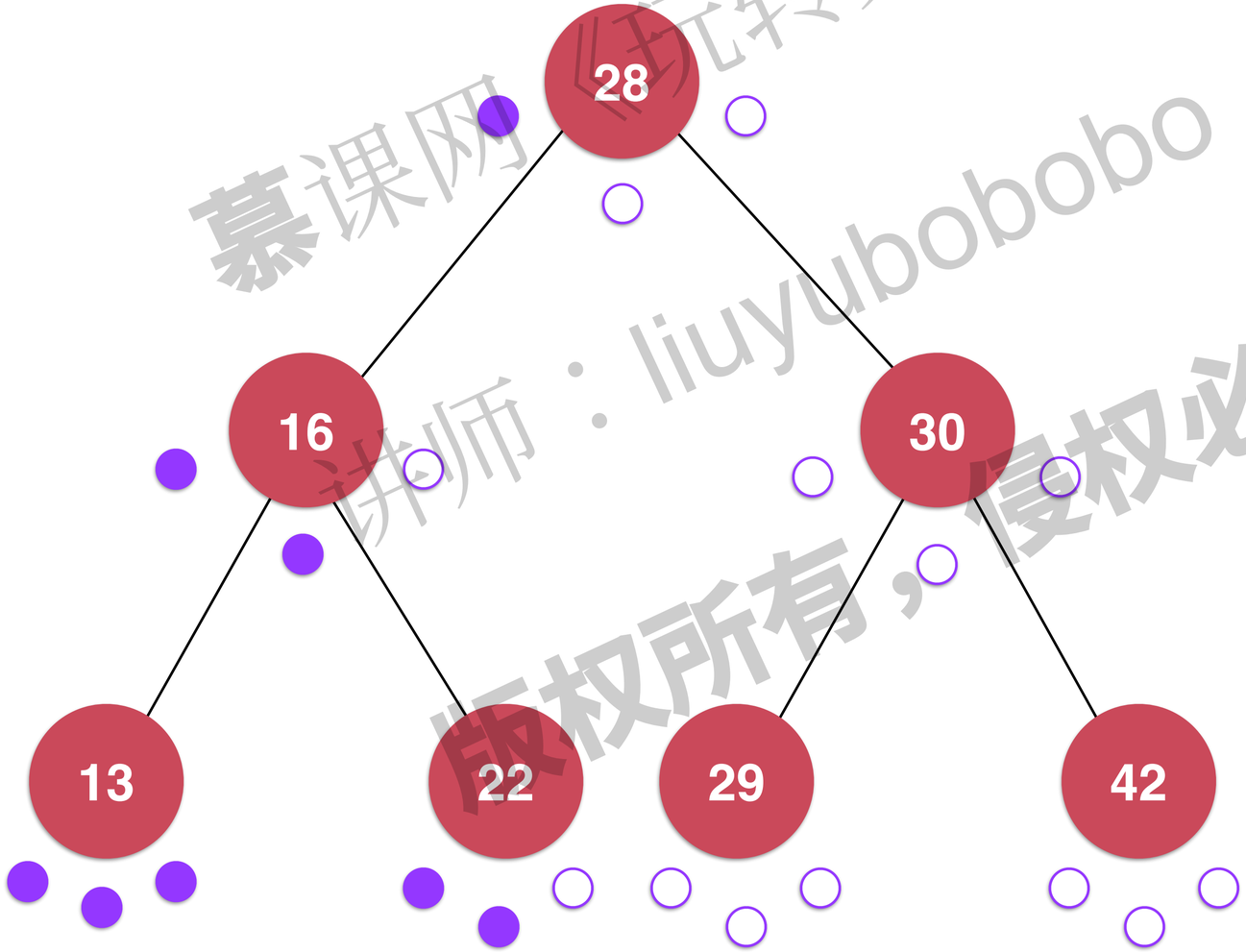
二分搜索树的后序遍历



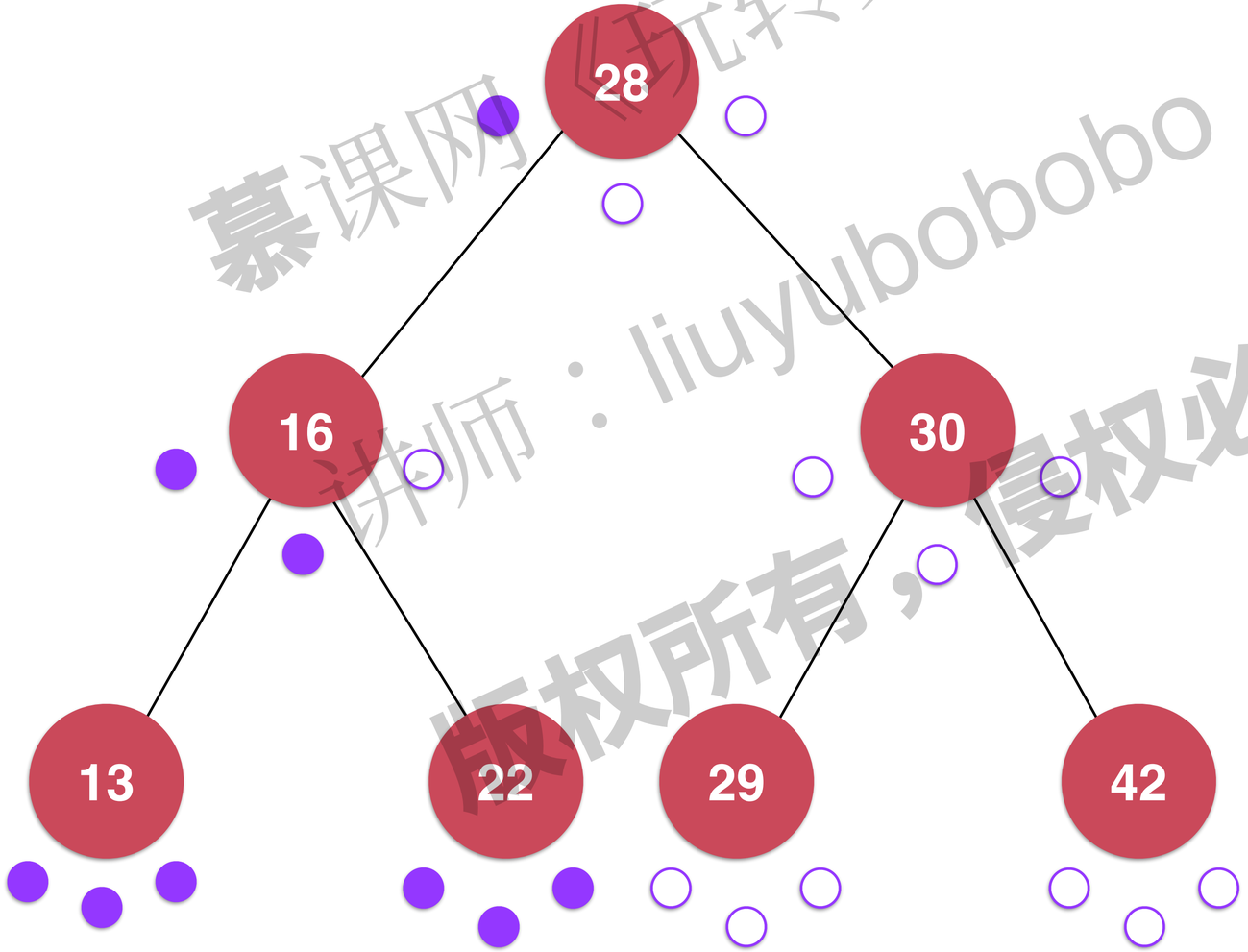
二分搜索树的后序遍历



二分搜索树的后序遍历



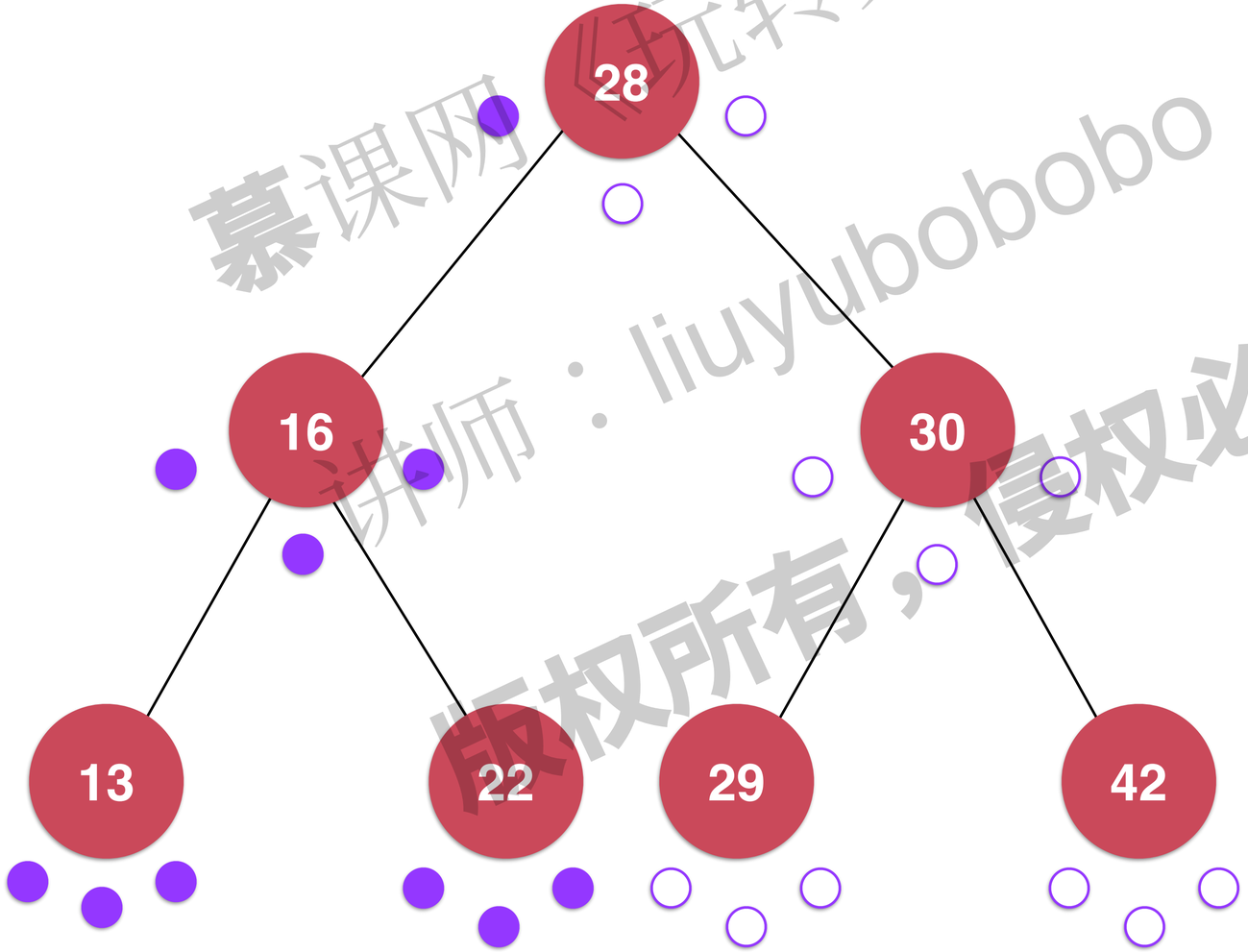
二分搜索树的后序遍历



13

22

二分搜索树的后序遍历

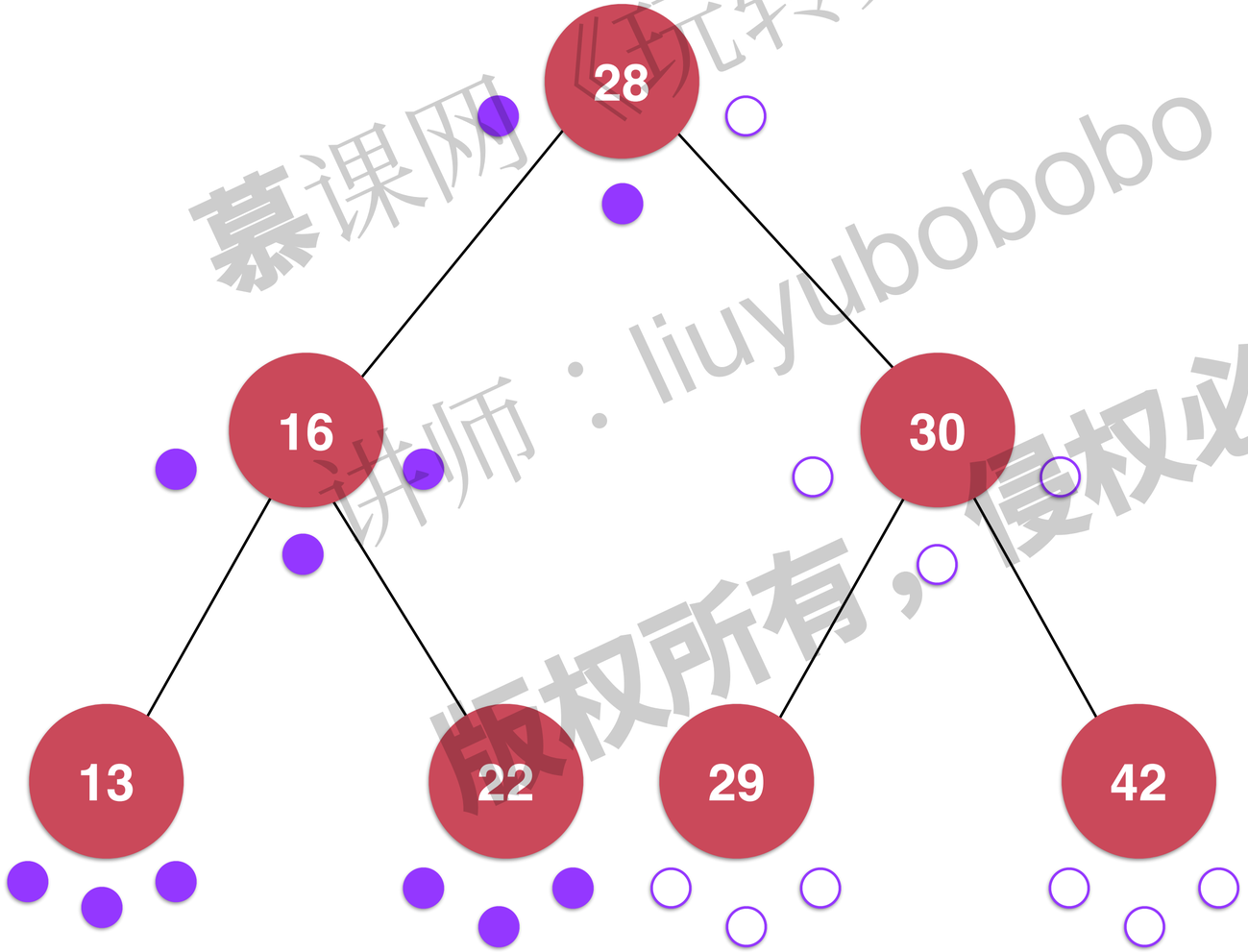


13

22

16

二分搜索树的后序遍历

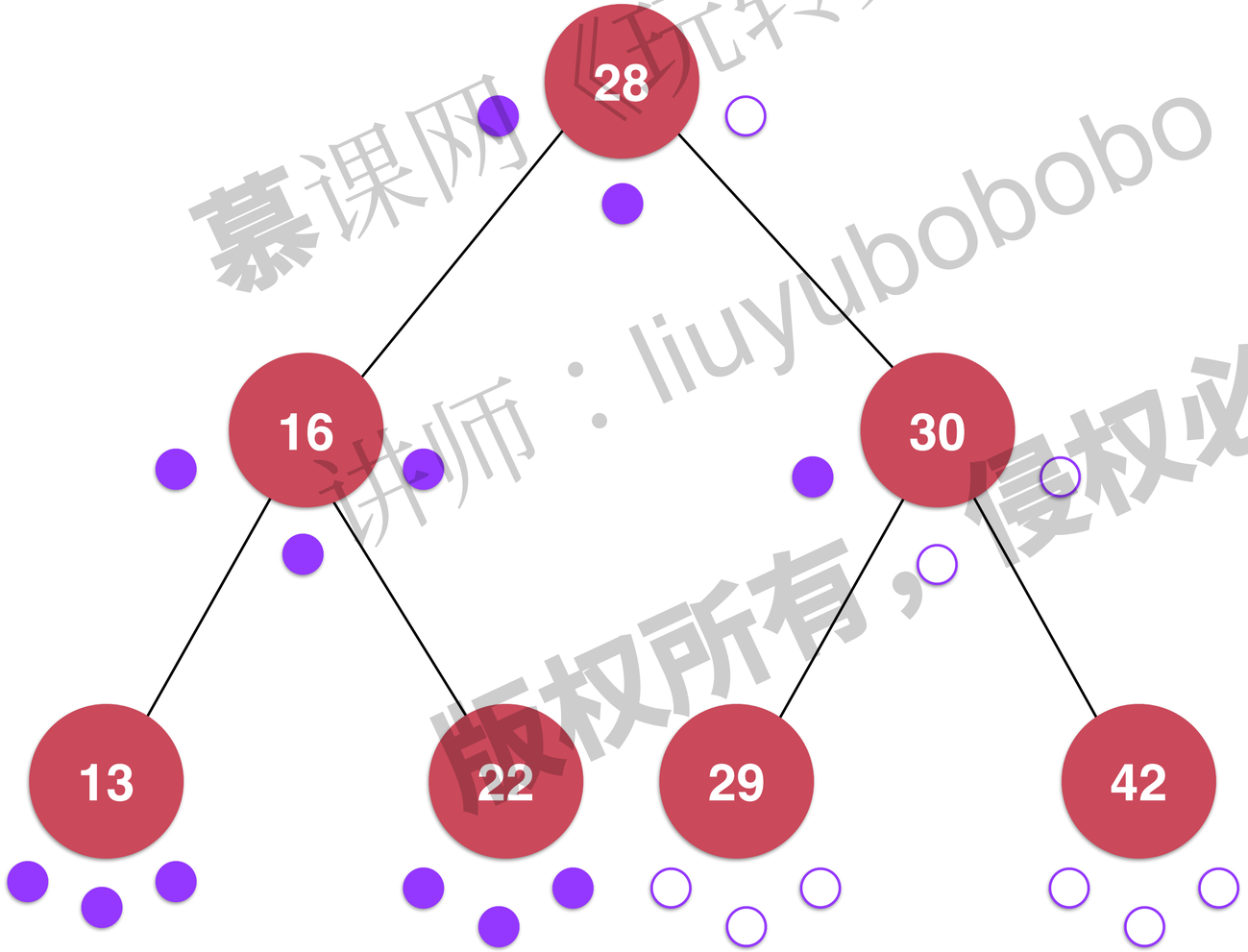


13

22

16

二分搜索树的后序遍历

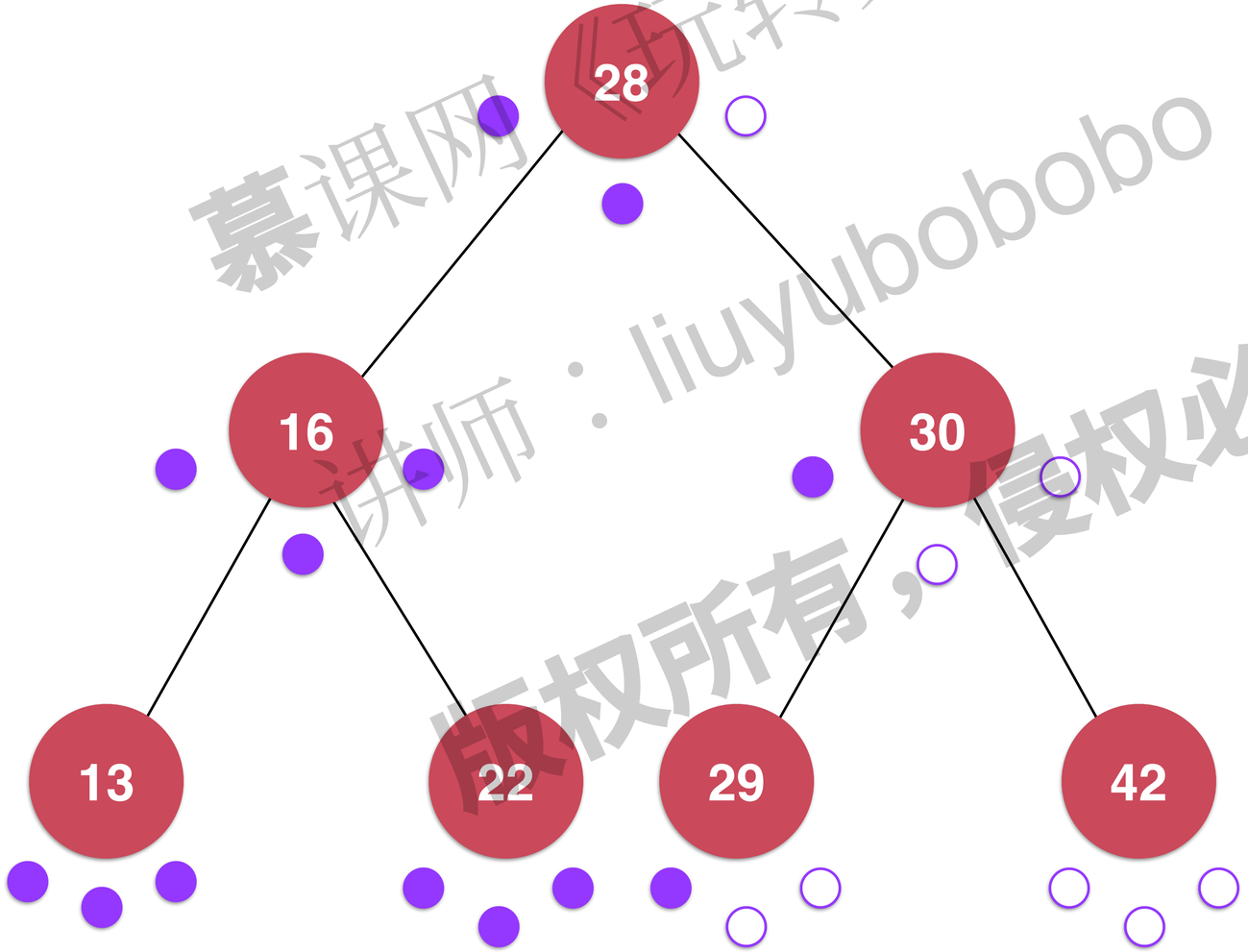


13

22

16

二分搜索树的后序遍历

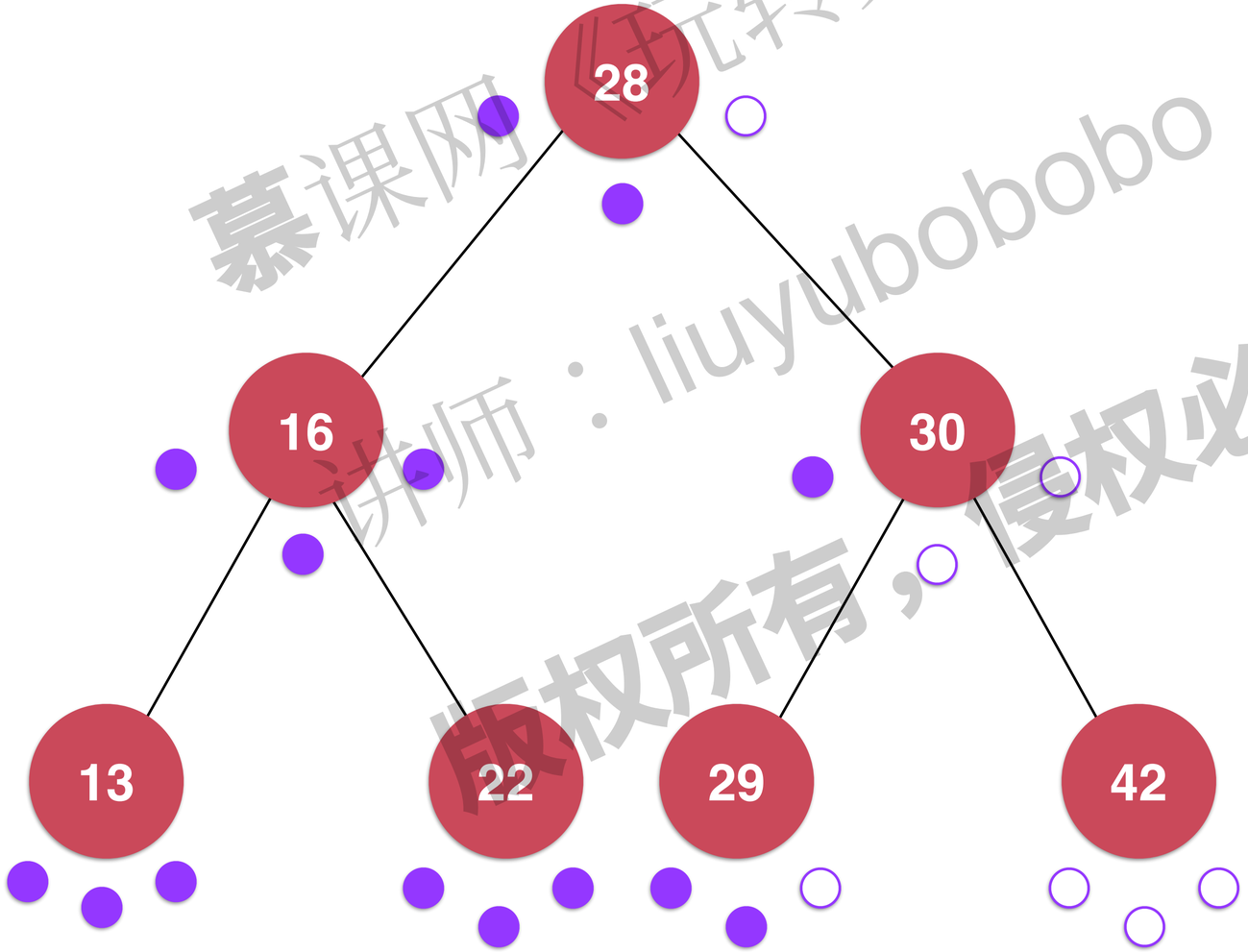


13

22

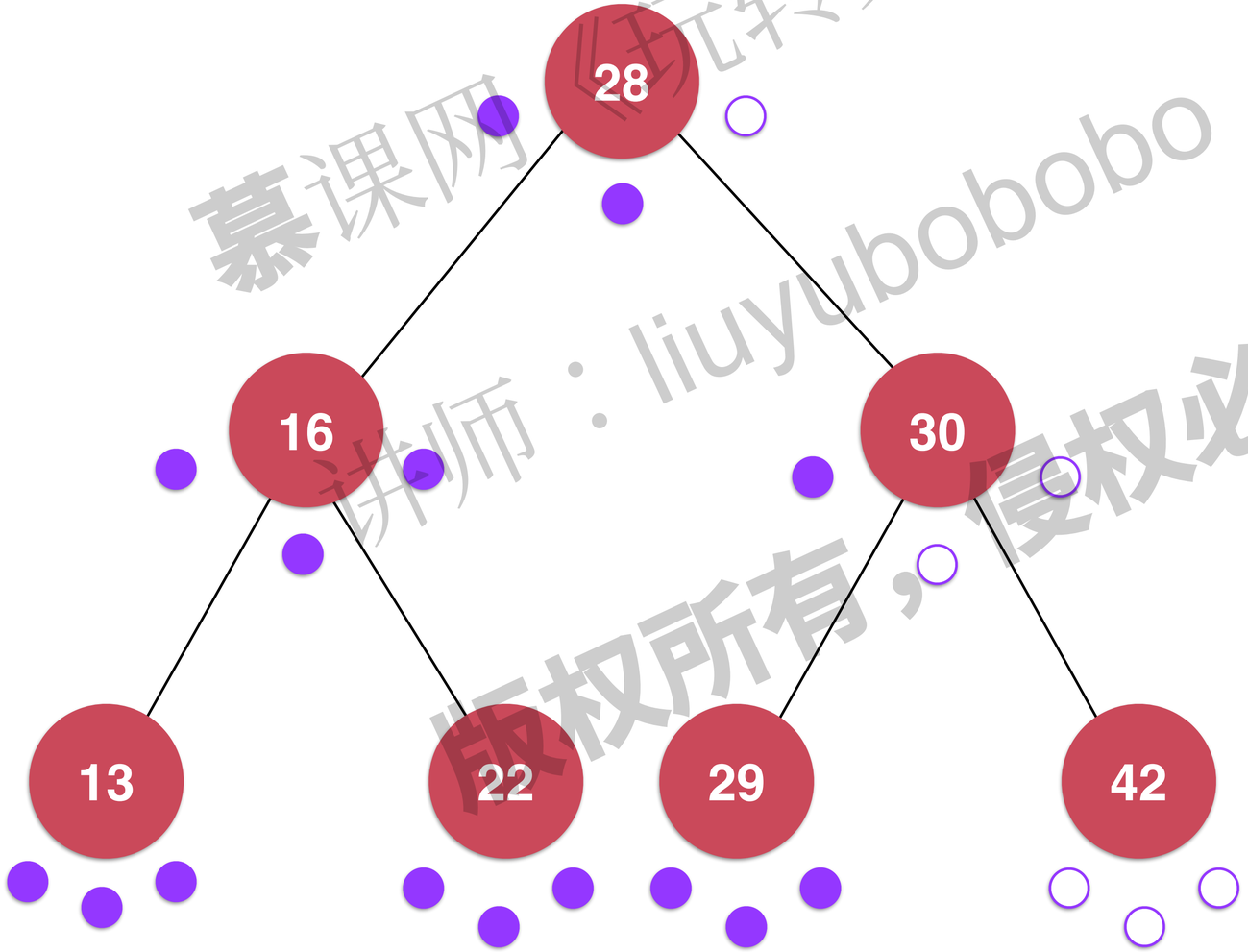
16

二分搜索树的后序遍历



13
22
16

二分搜索树的后序遍历



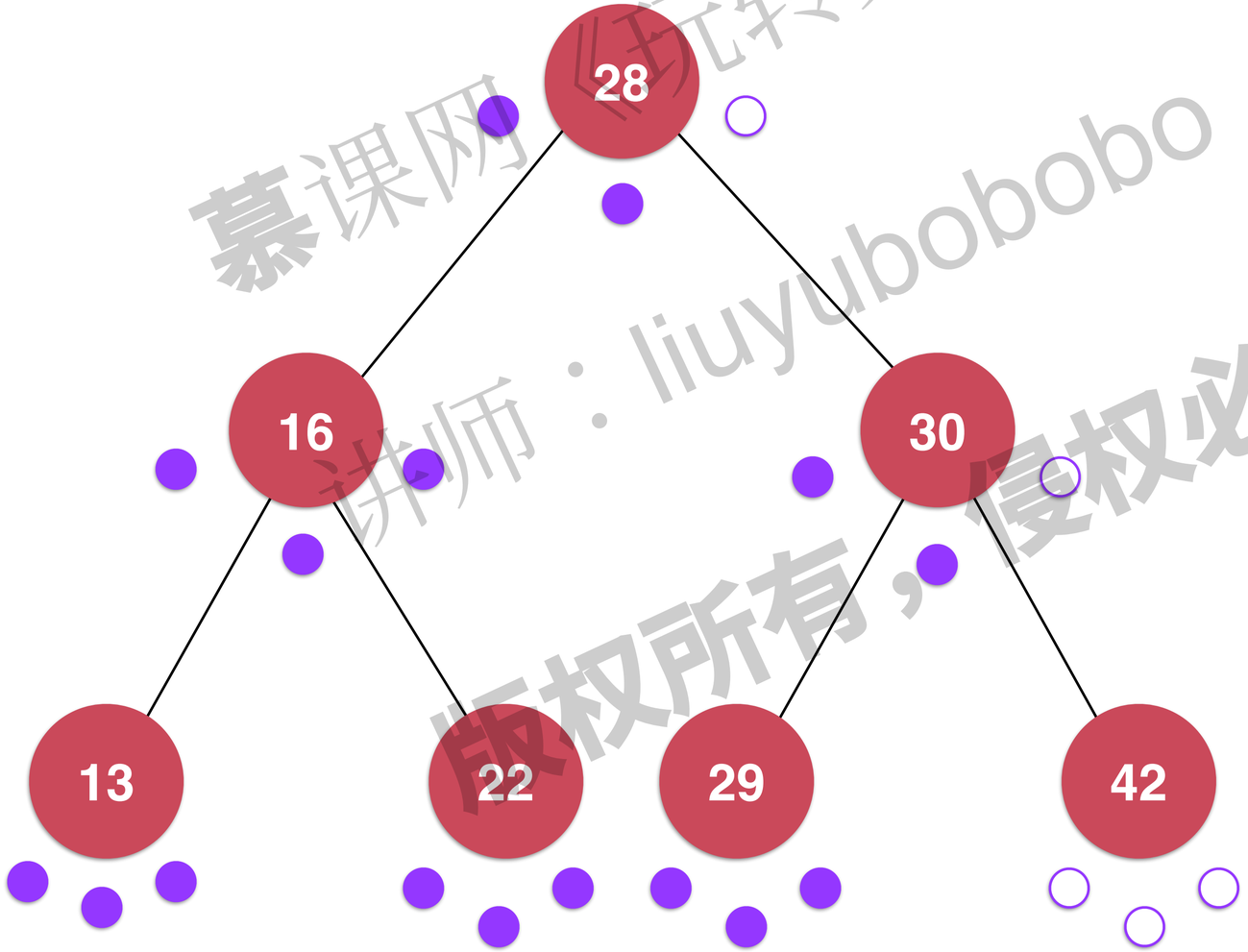
13

22

16

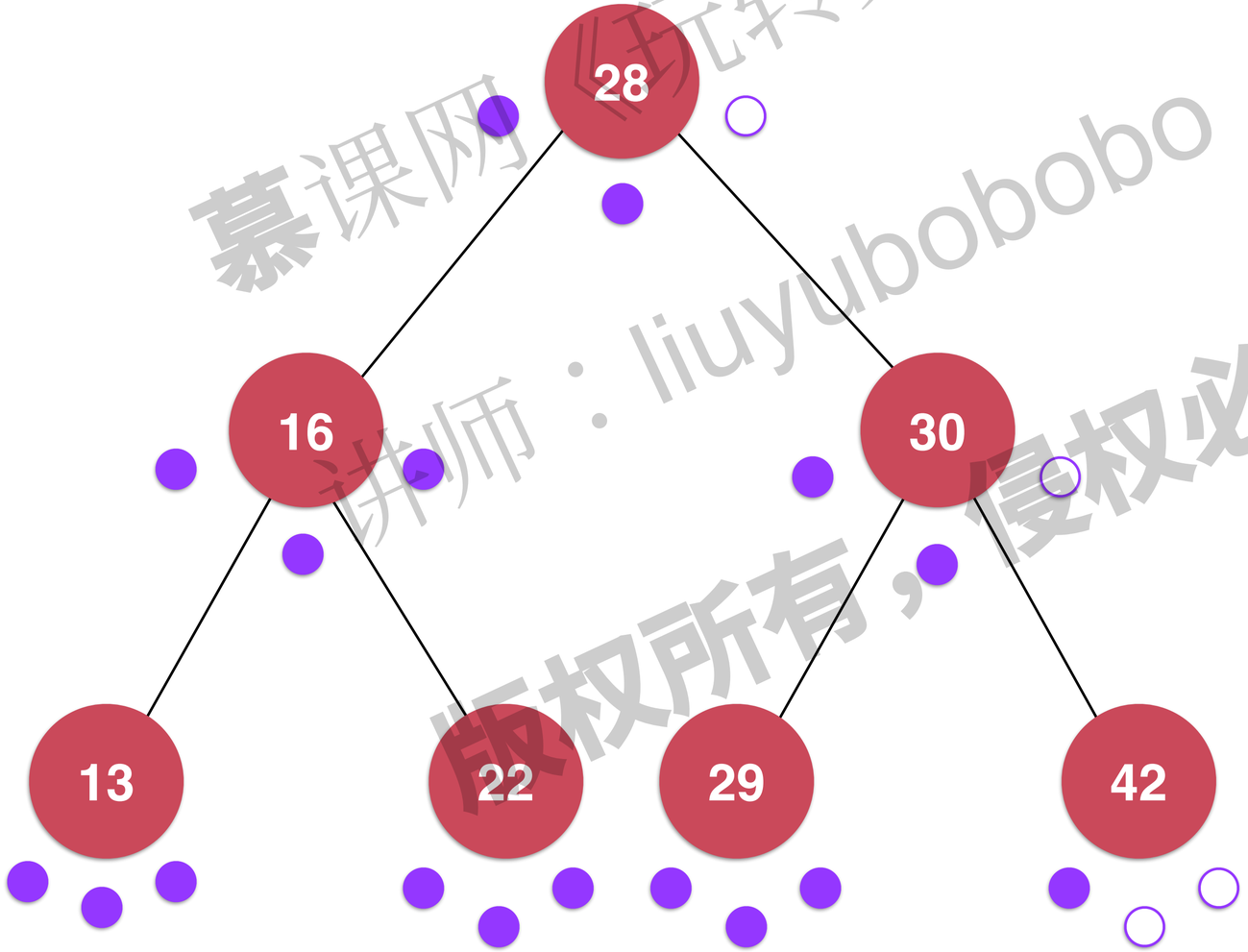
29

二分搜索树的后序遍历



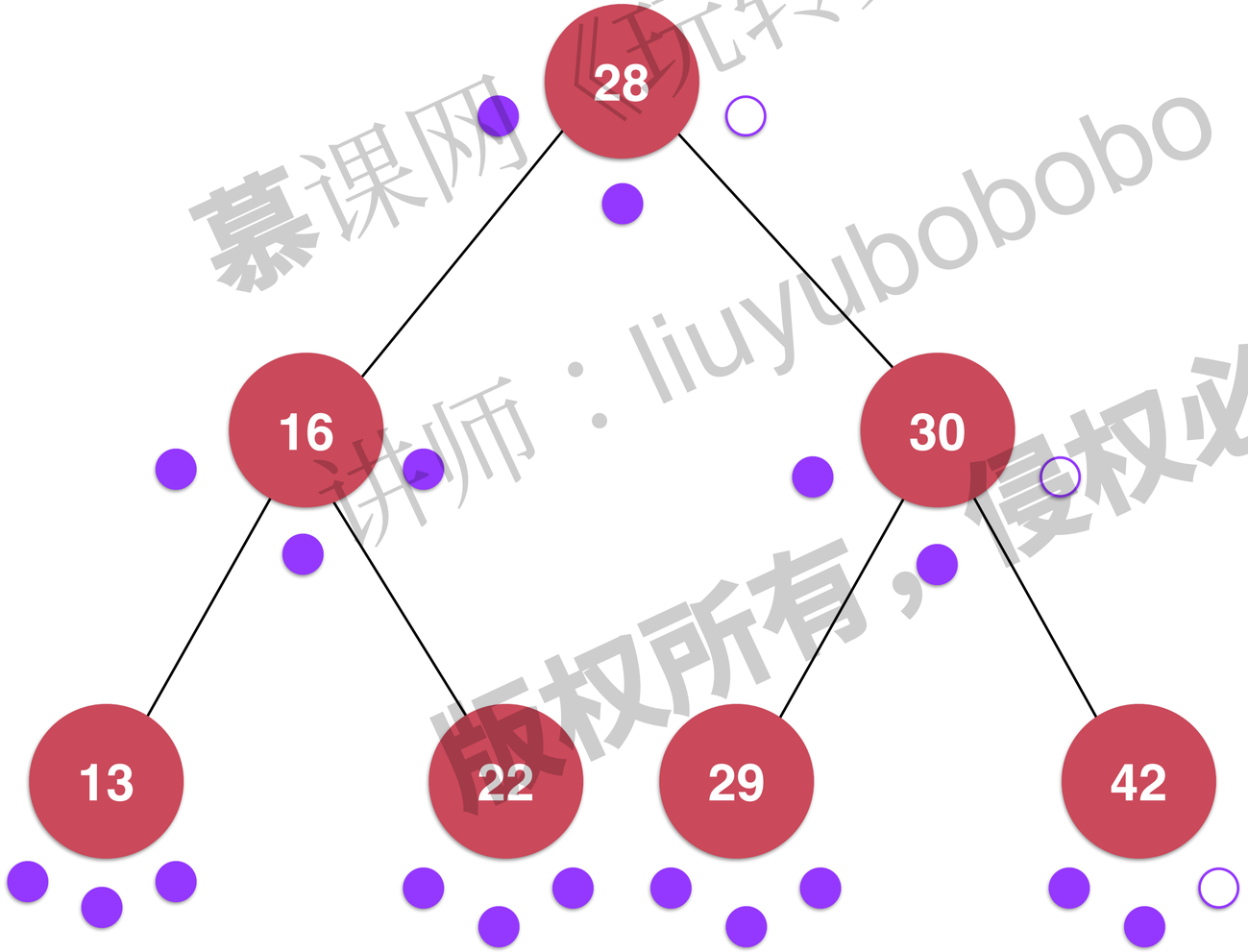
13
22
16
29

二分搜索树的后序遍历



13
22
16
29

二分搜索树的后序遍历



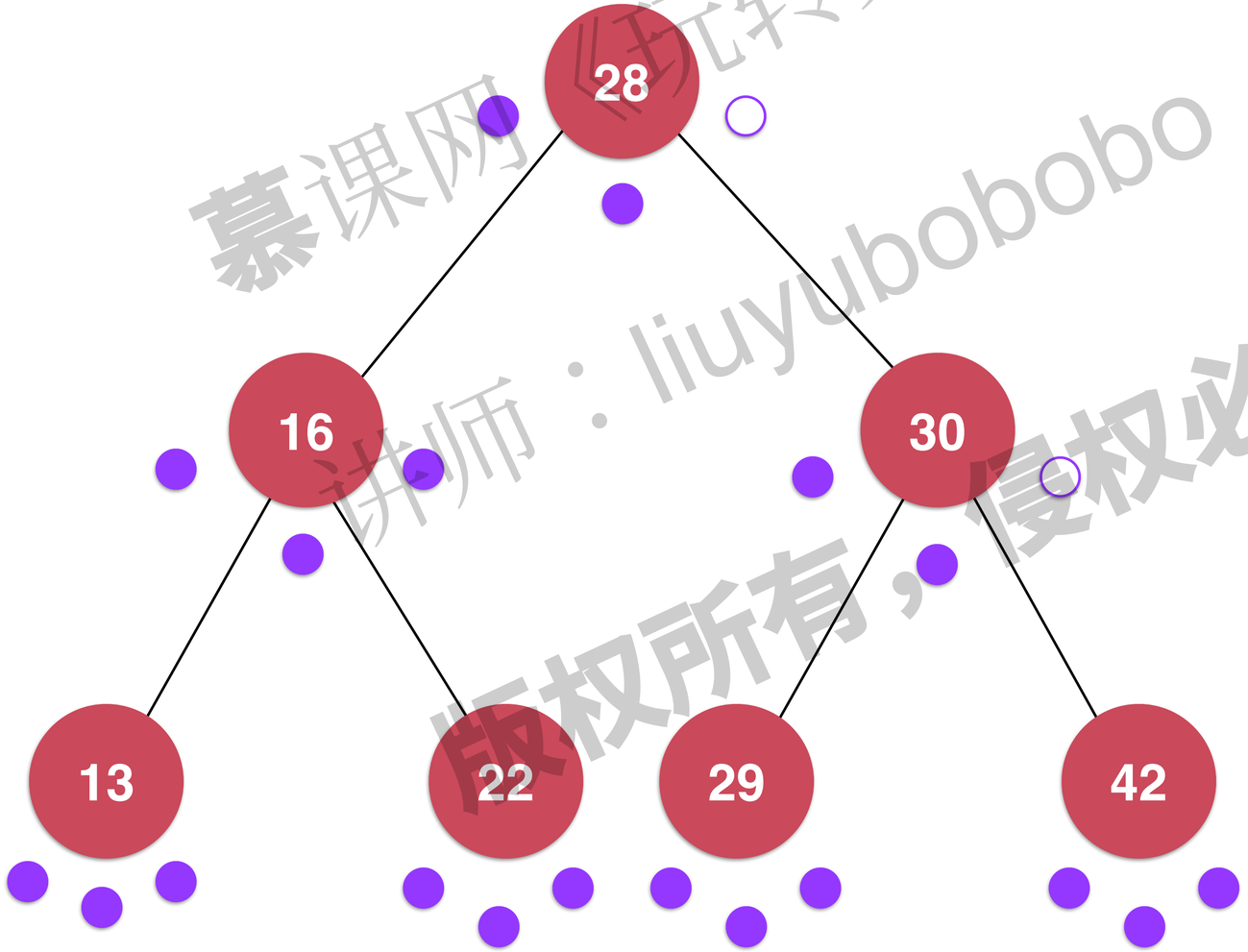
13

22

16

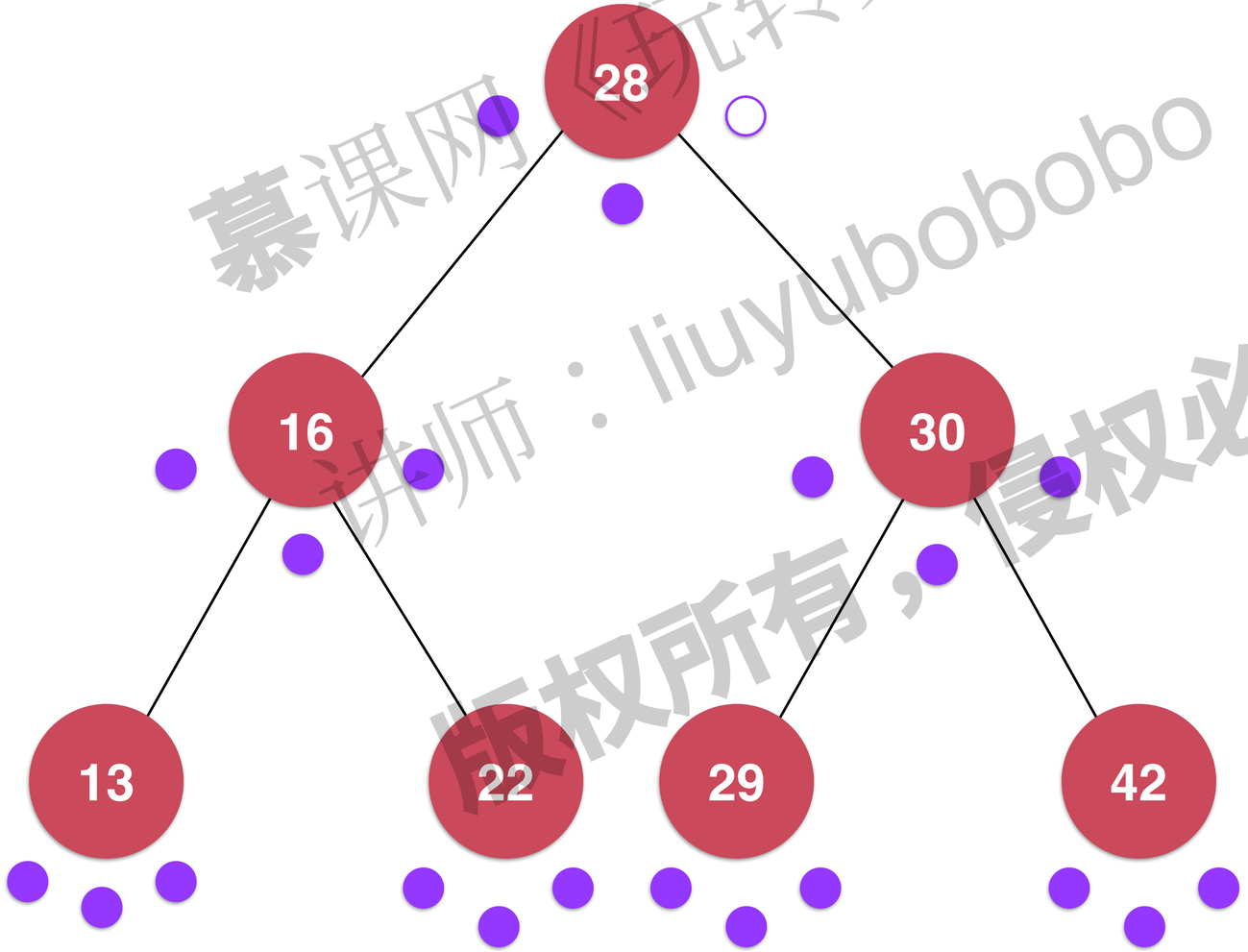
29

二分搜索树的后序遍历



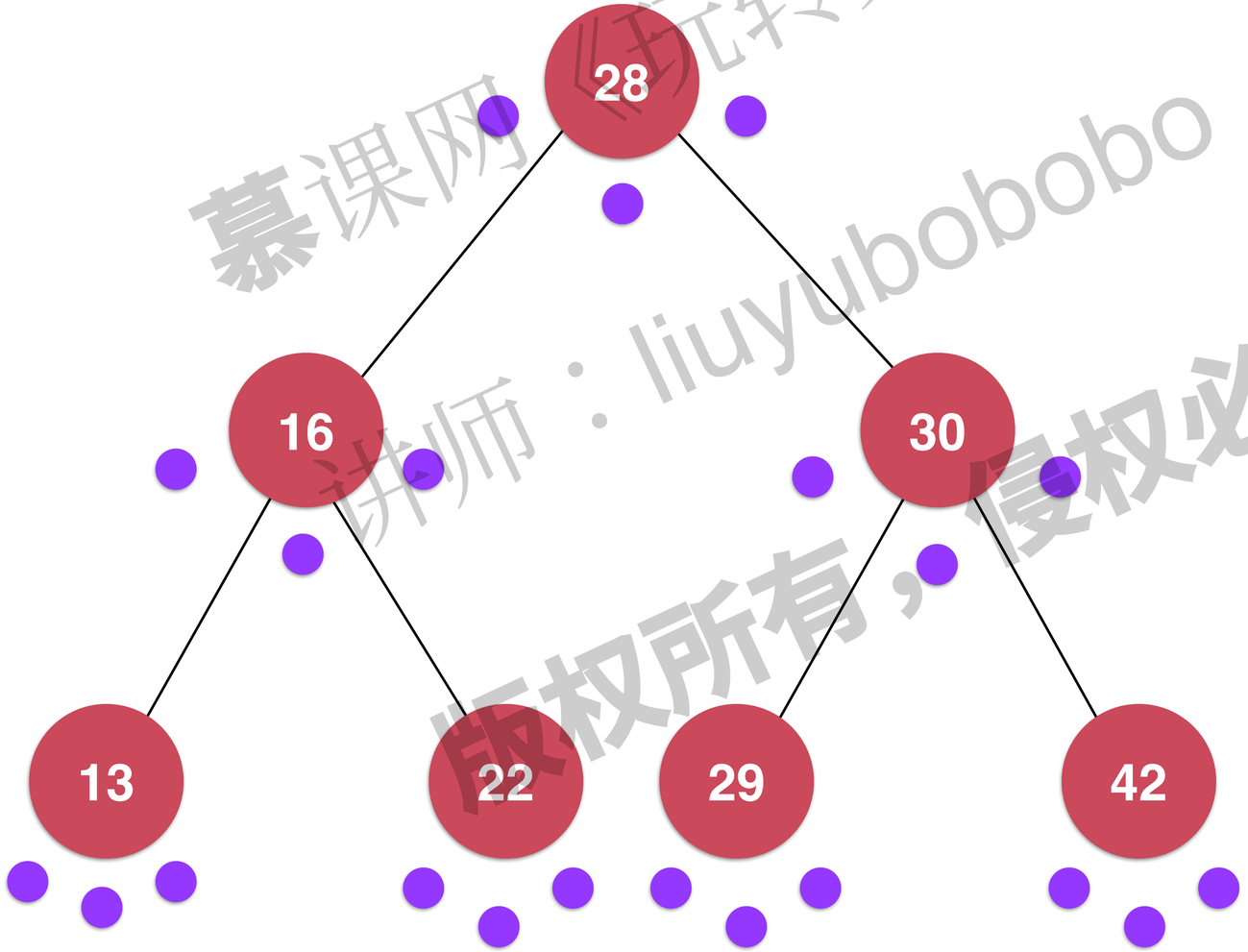
13
22
16
29
42

二分搜索树的后序遍历



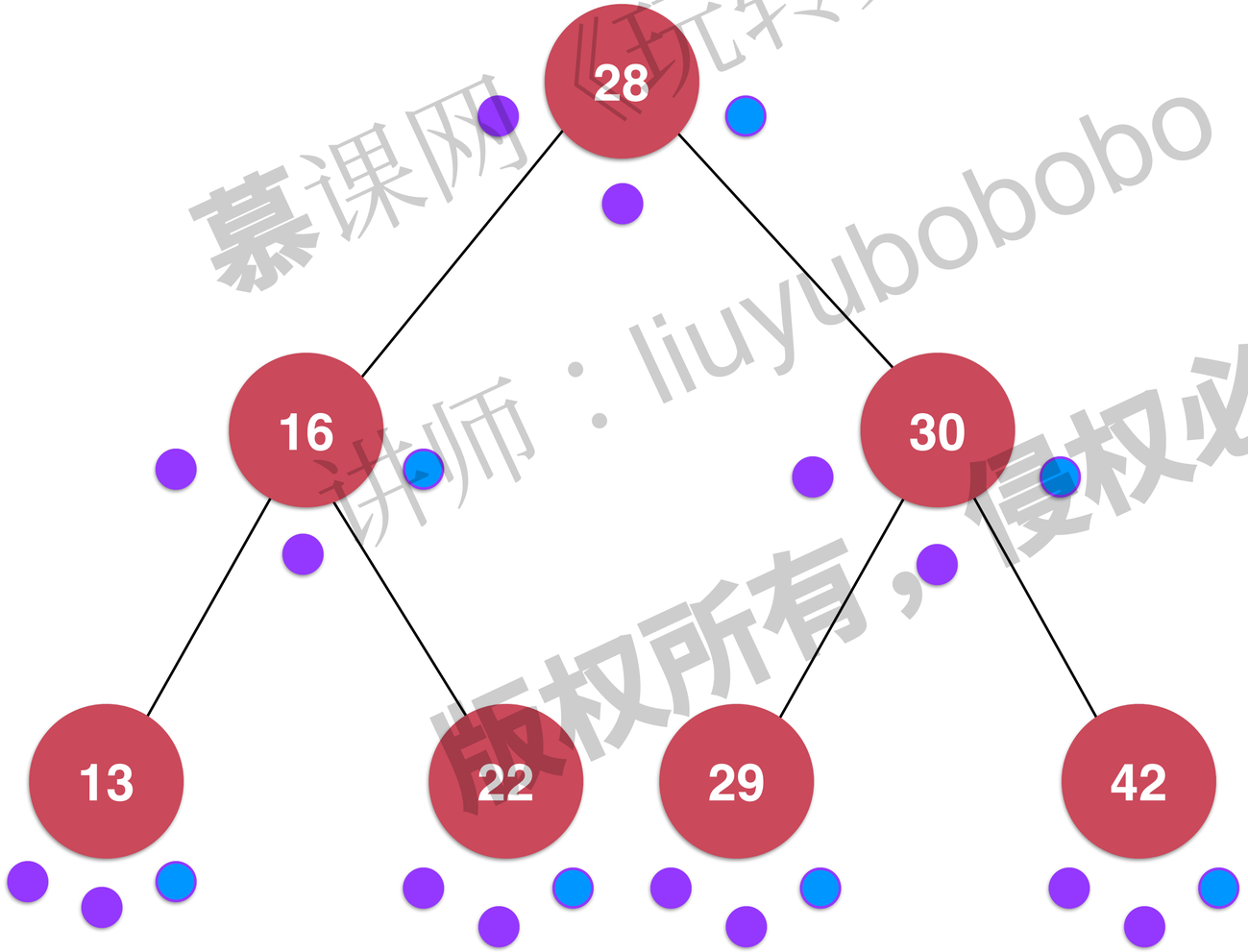
13
22
16
29
42
30

二分搜索树的后序遍历



13
22
16
29
42
30
28

二分搜索树的后序遍历



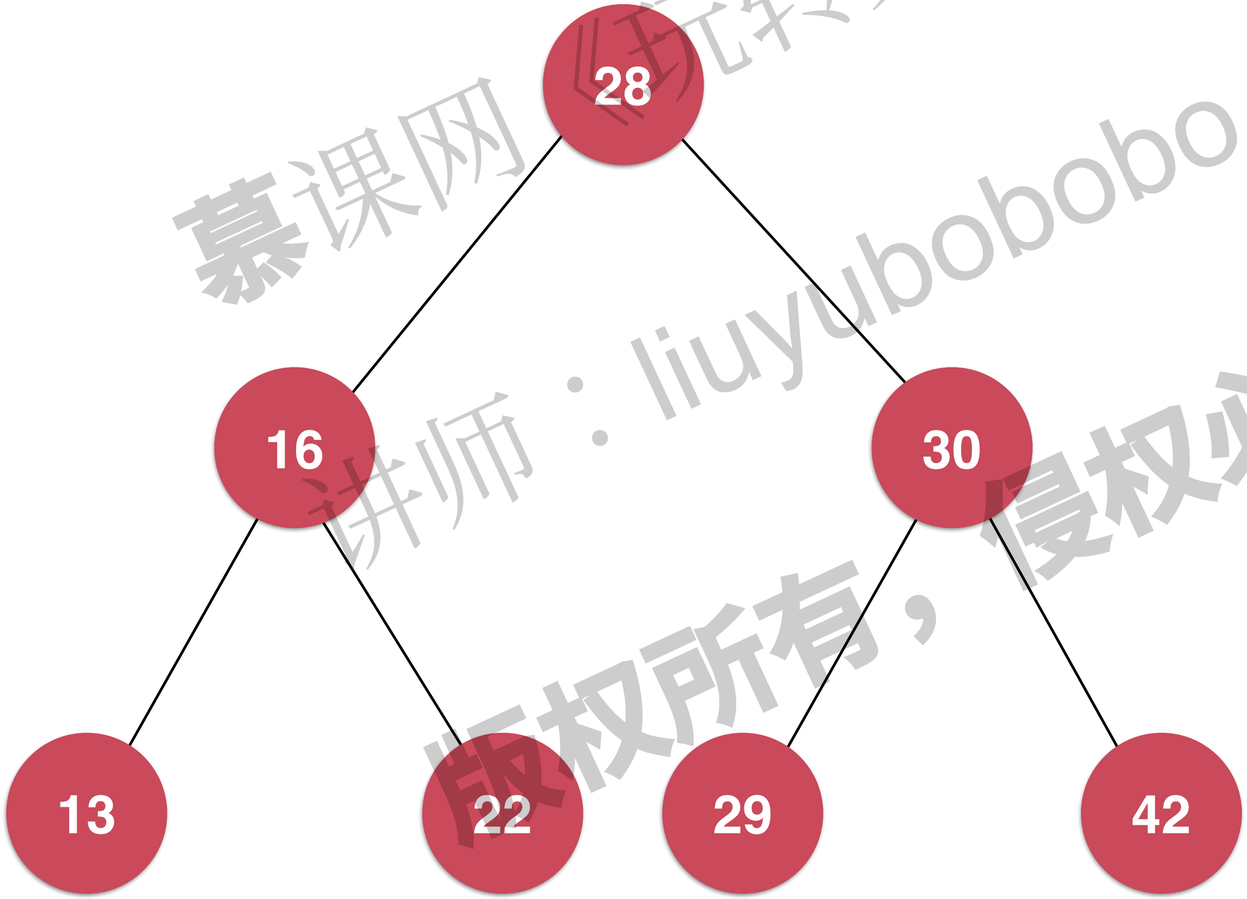
13
22
16
29
42
30
28

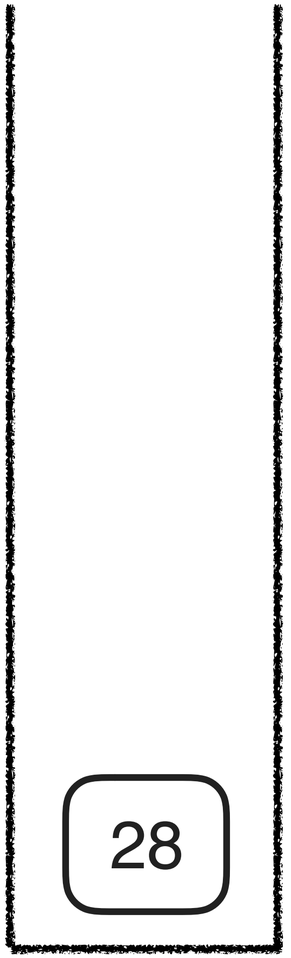
二分搜索树前序遍历的非递归写法

前序遍历

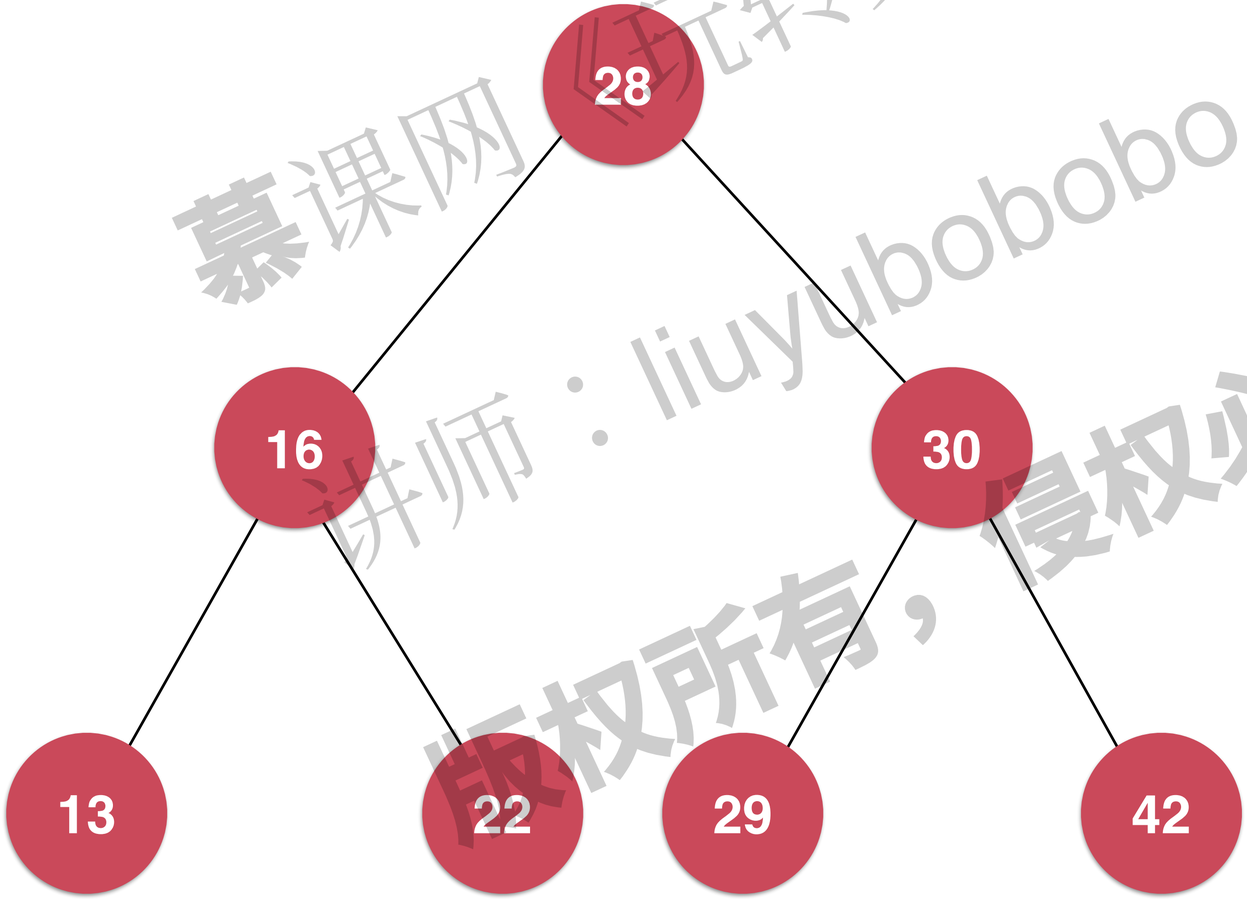
```
function traverse(node):
    if(node == null)
        return;

    访问该节点
    traverse(node.left)
    traverse(node.right)
```

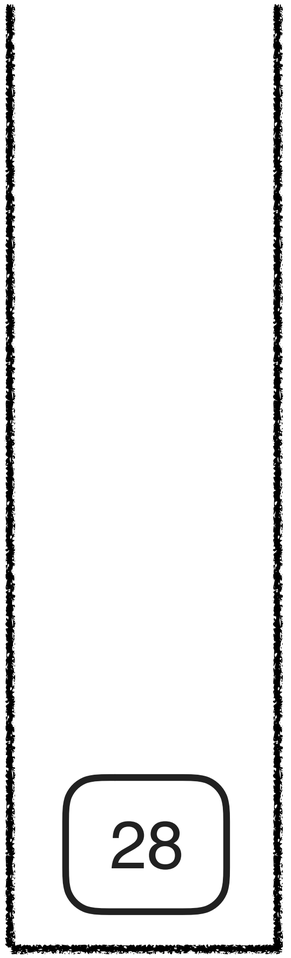
二分搜索树前序遍历的非递归写法



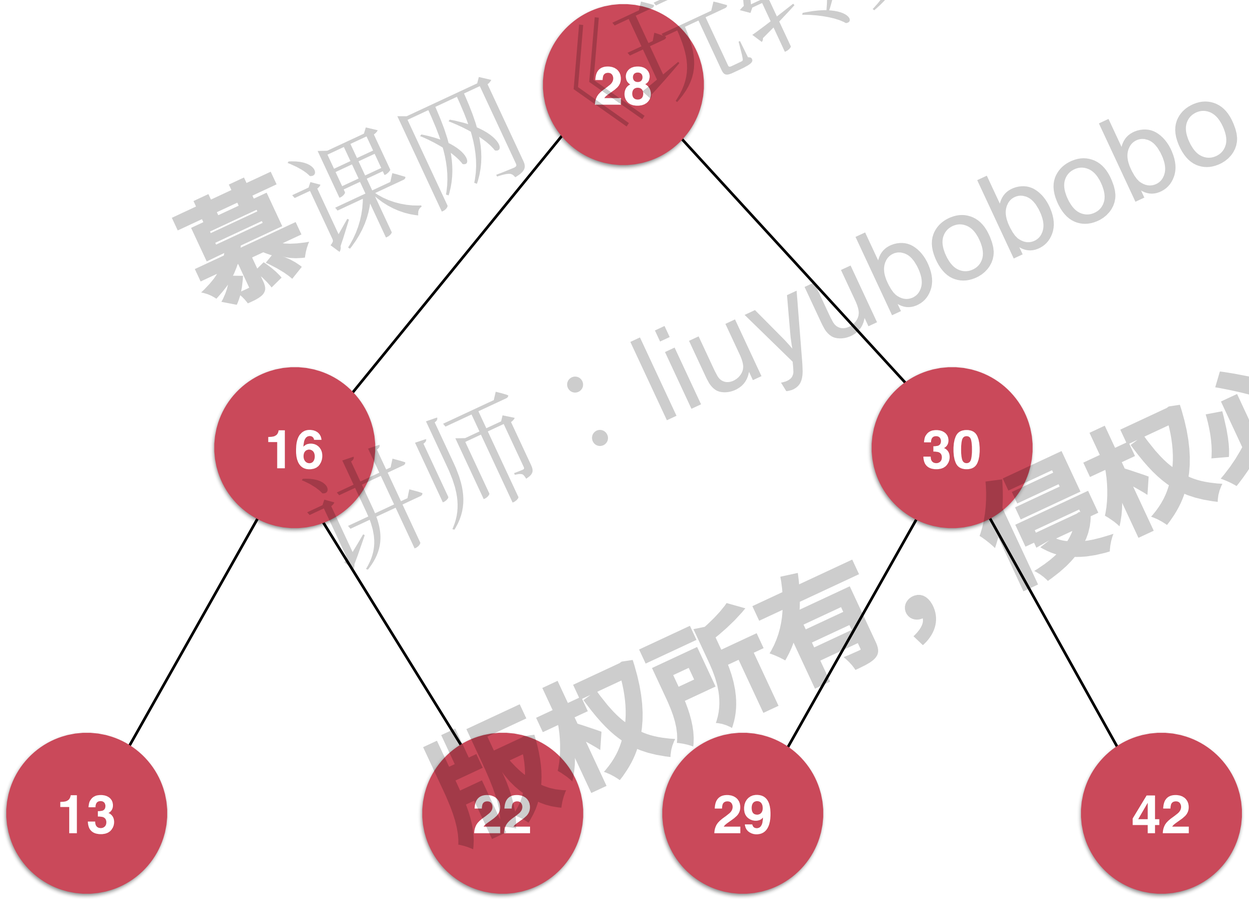


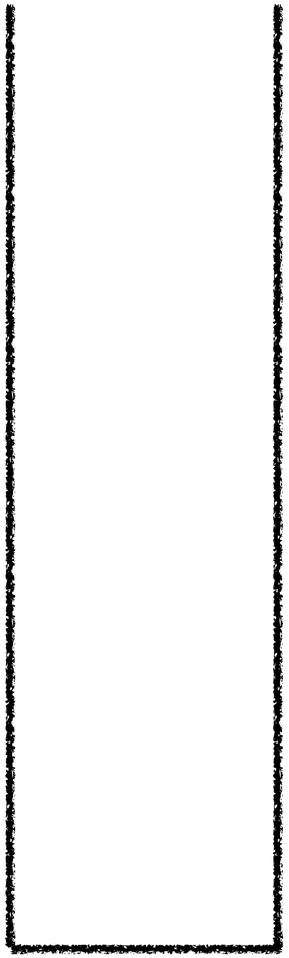
二分搜索树前序遍历的非递归写法



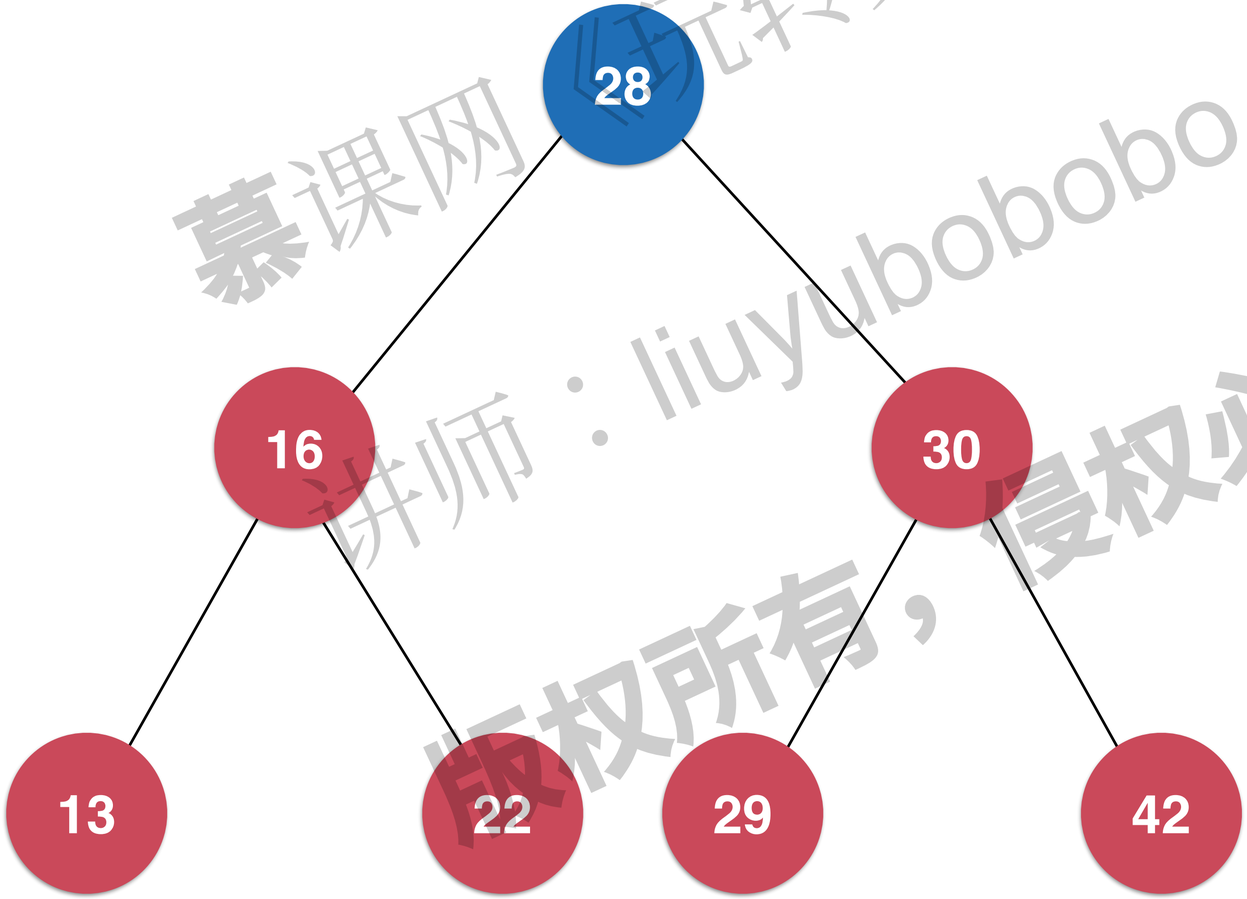


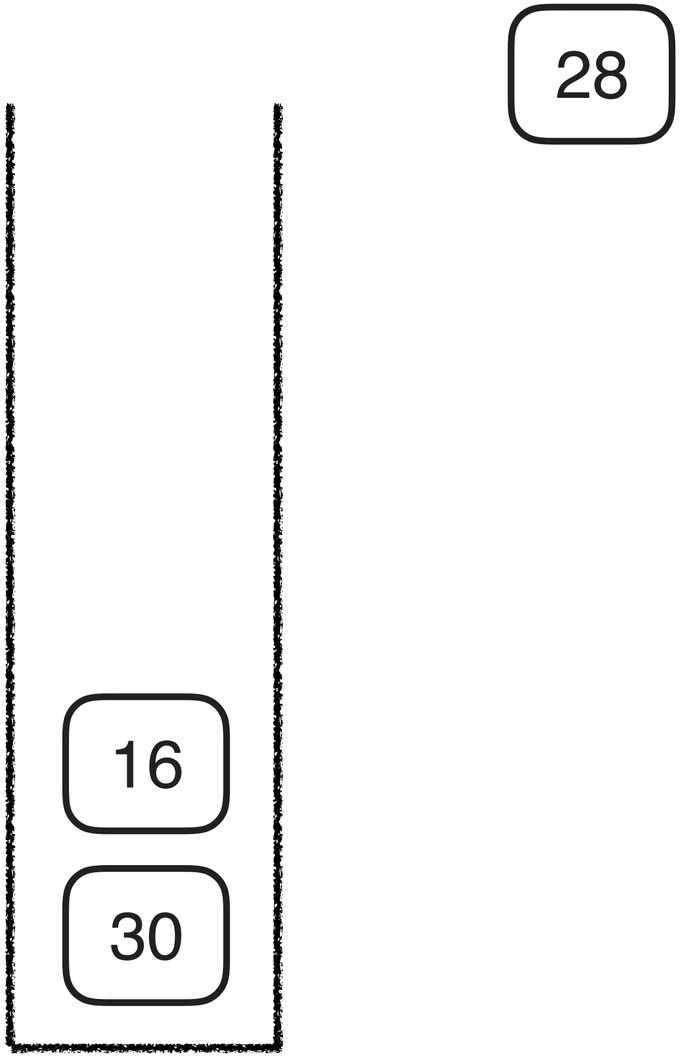
二分搜索树前序遍历的非递归写法



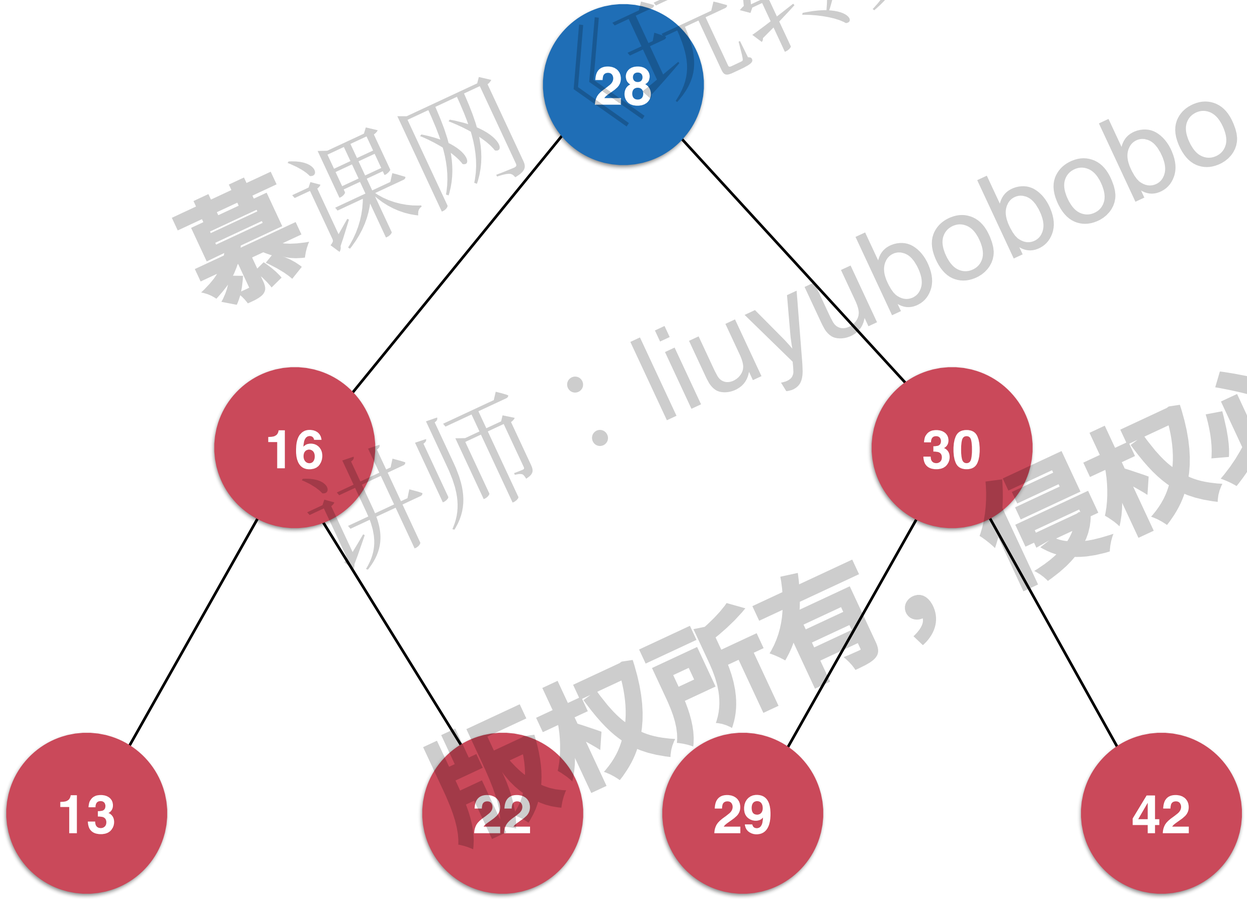


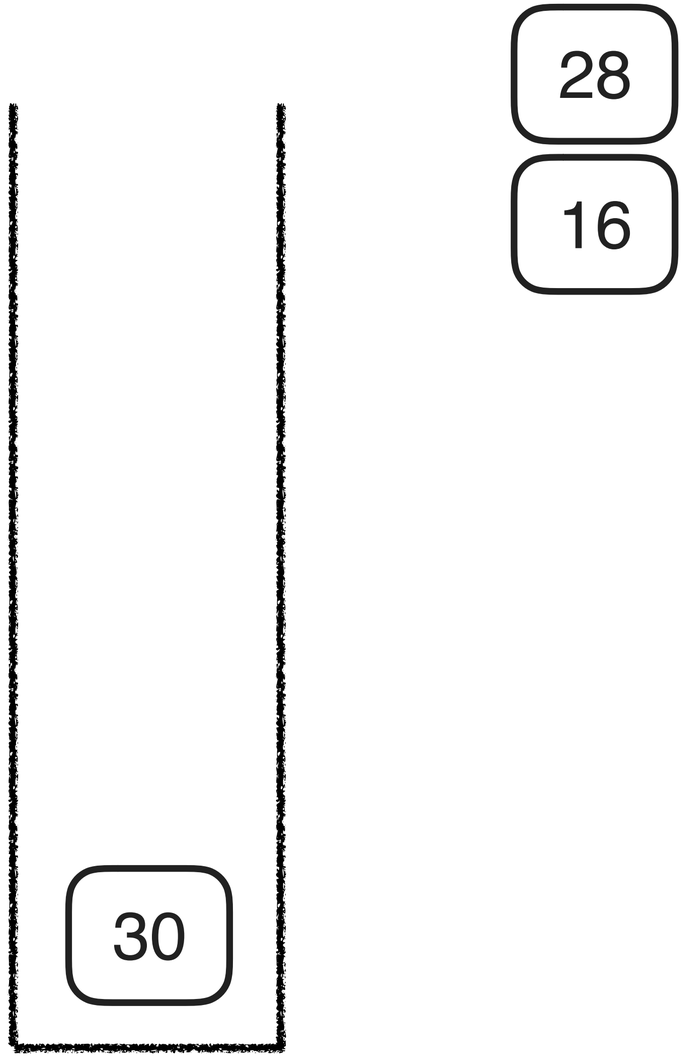
二分搜索树前序遍历的非递归写法



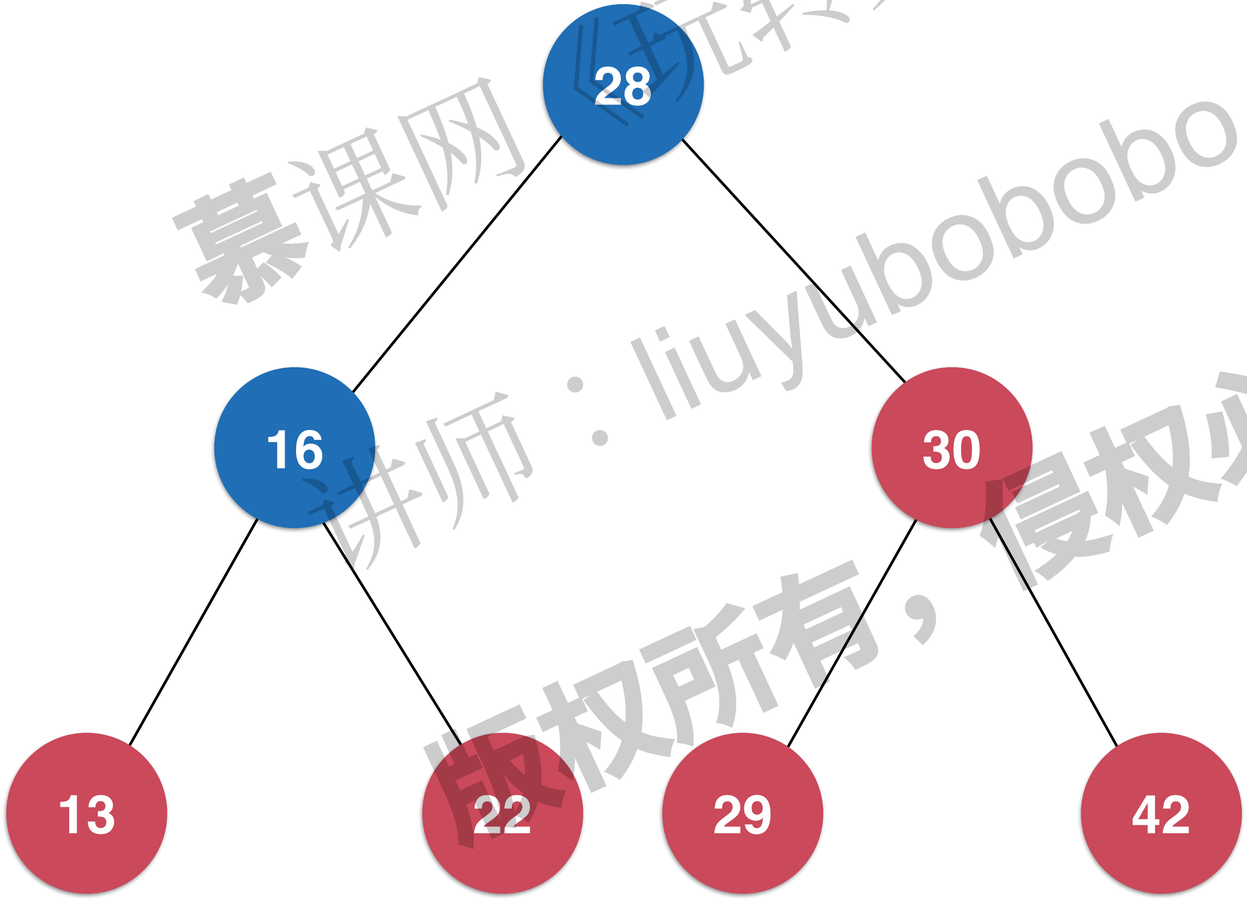


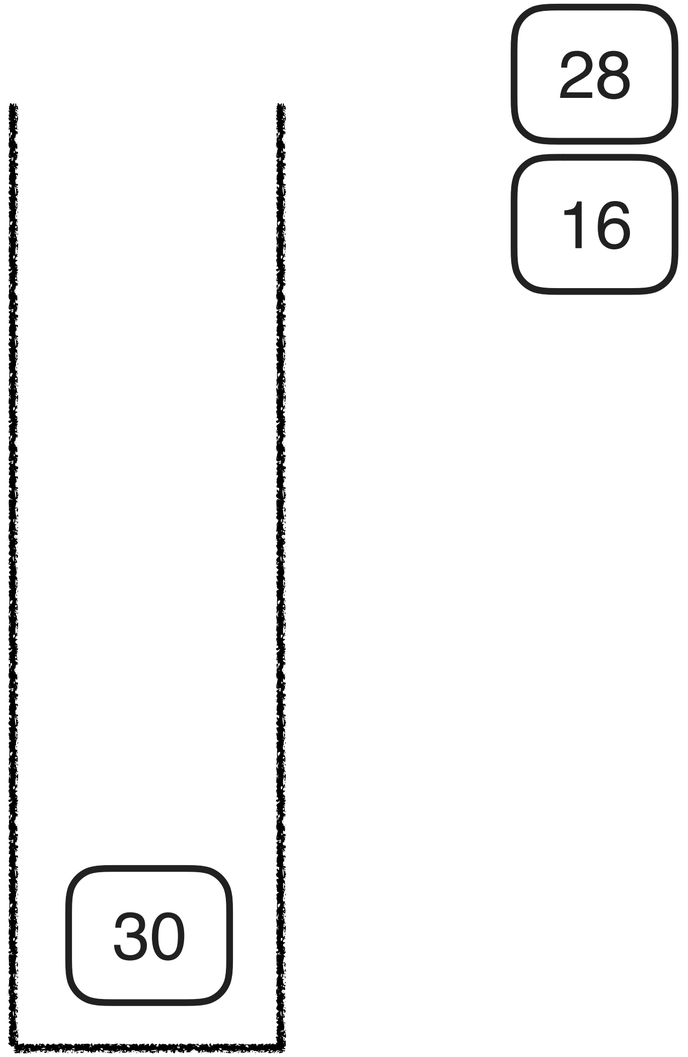
二分搜索树前序遍历的非递归写法



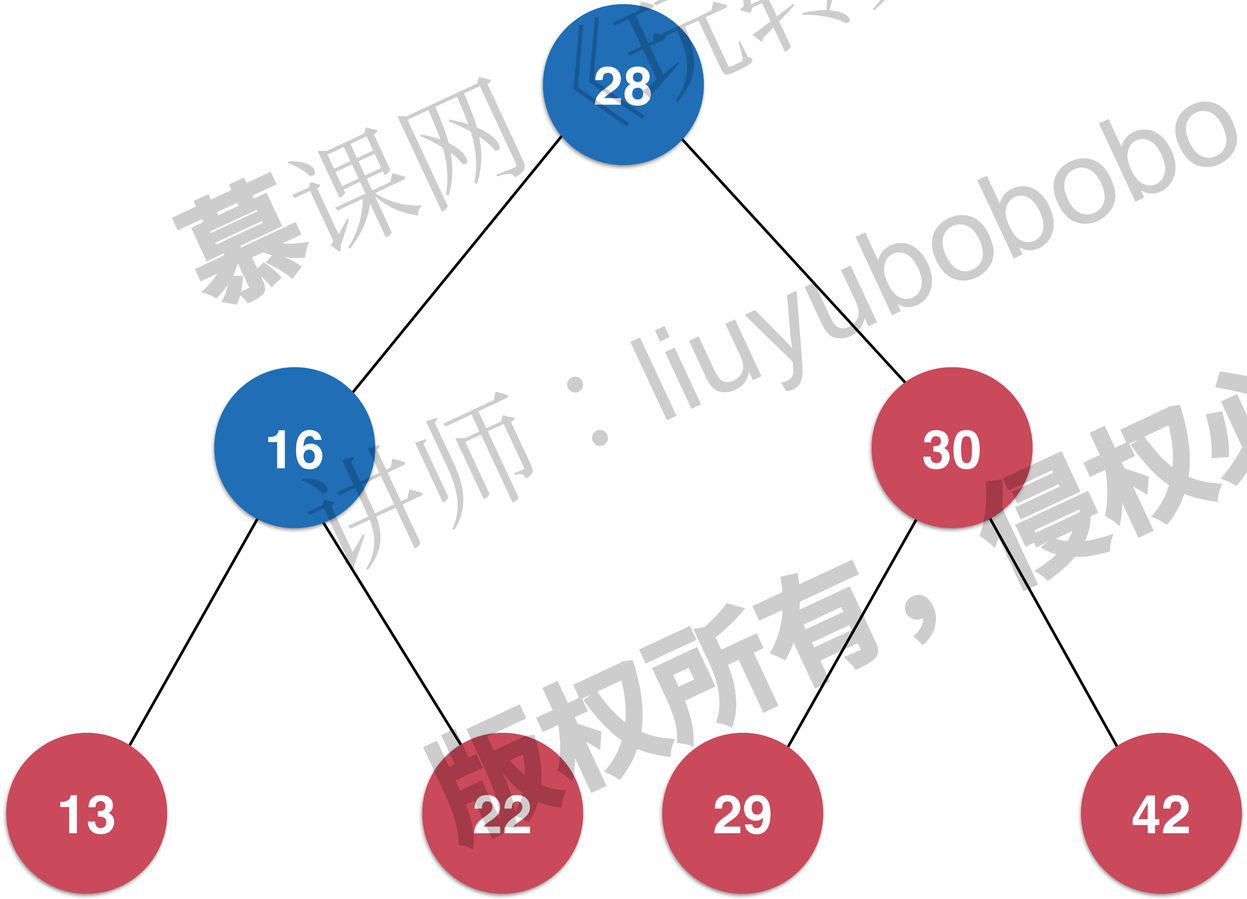


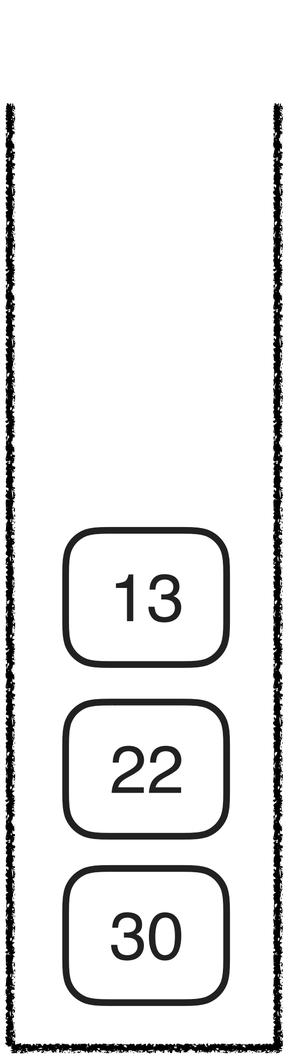
二分搜索树前序遍历的非递归写法





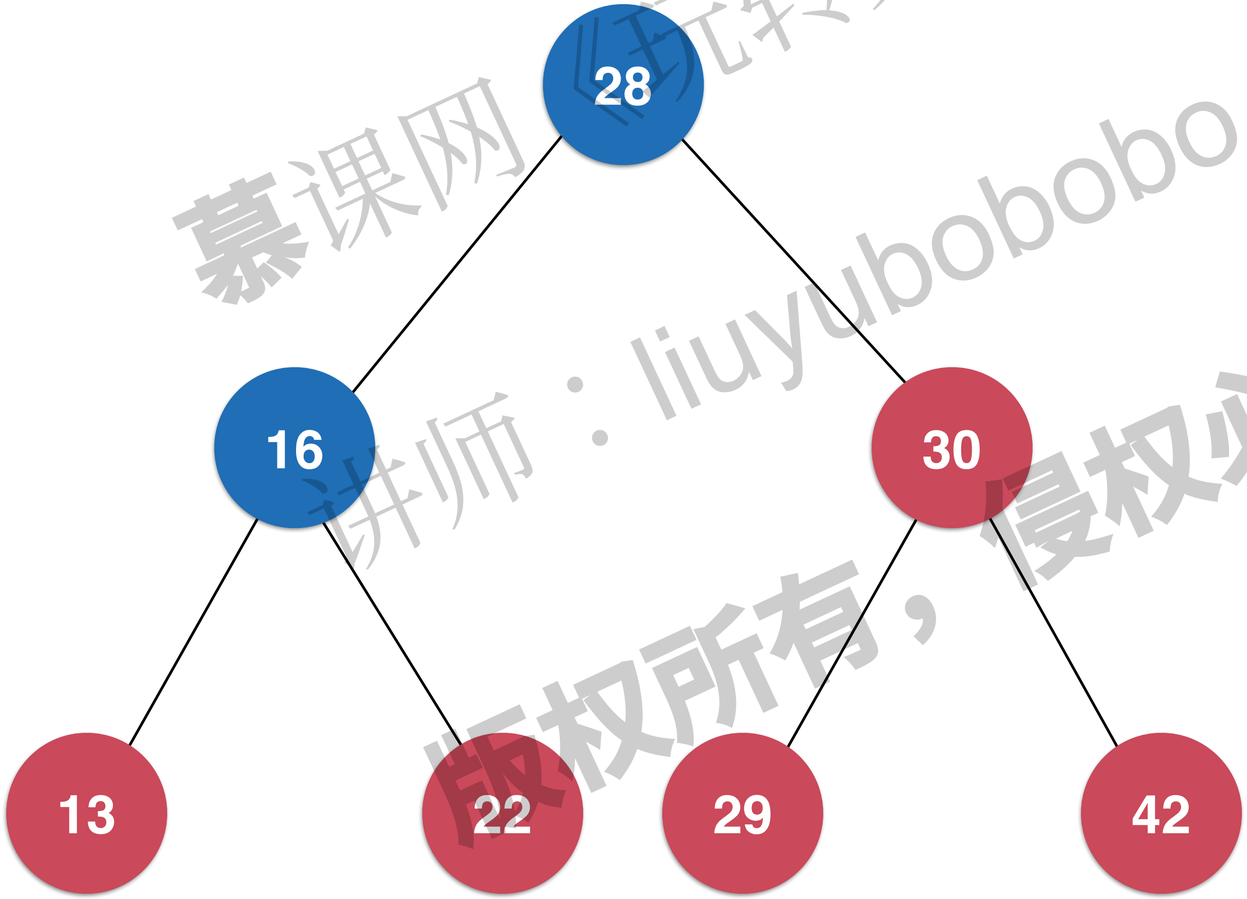
二分搜索树前序遍历的非递归写法

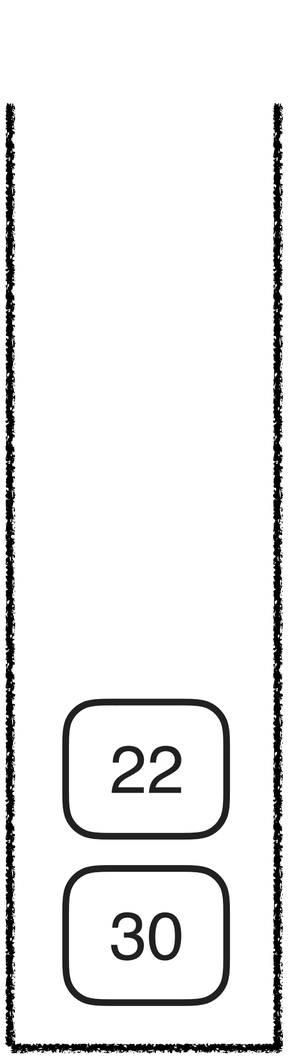


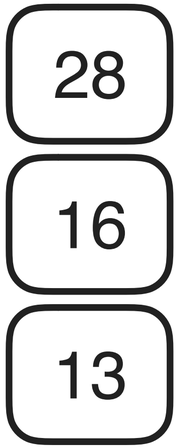


28
16

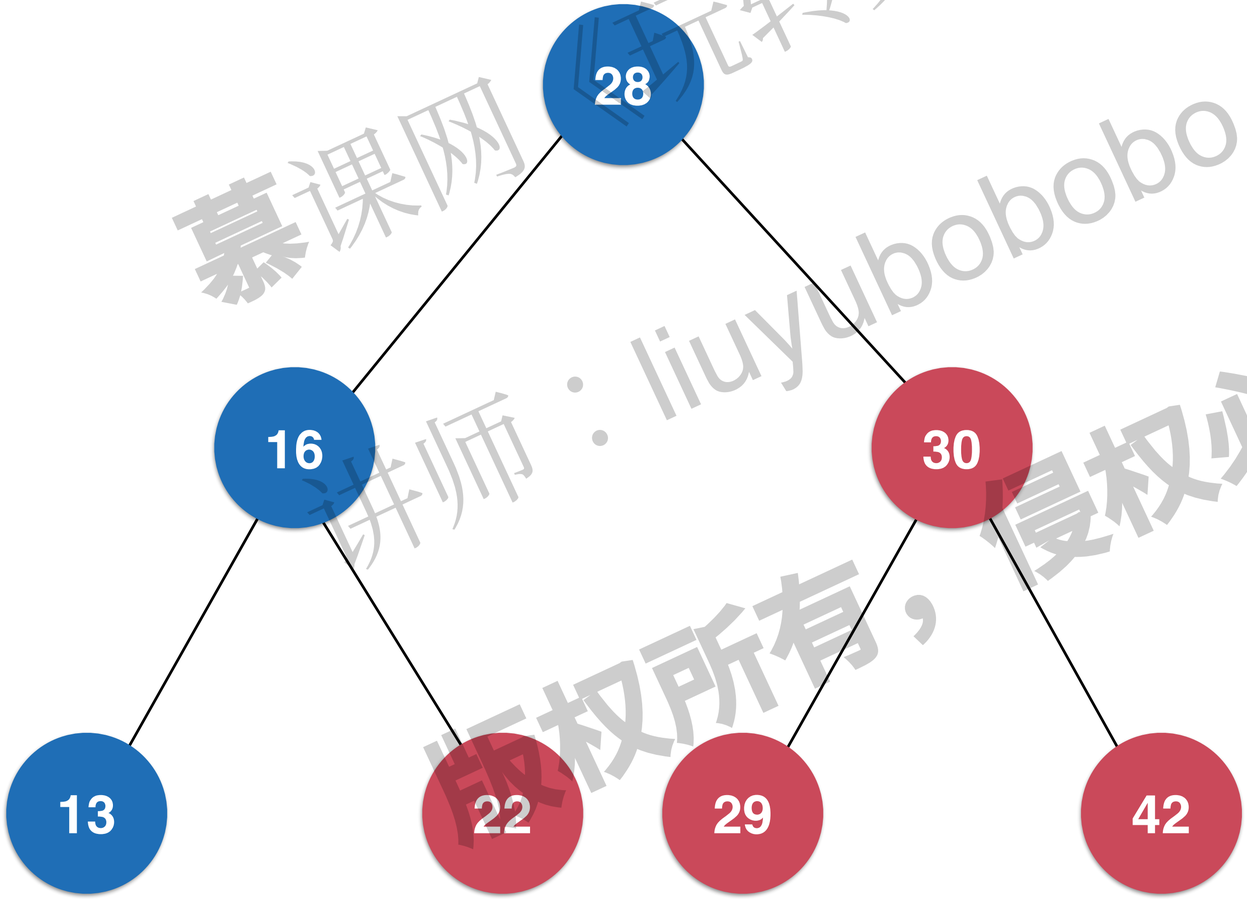
二分搜索树前序遍历的非递归写法

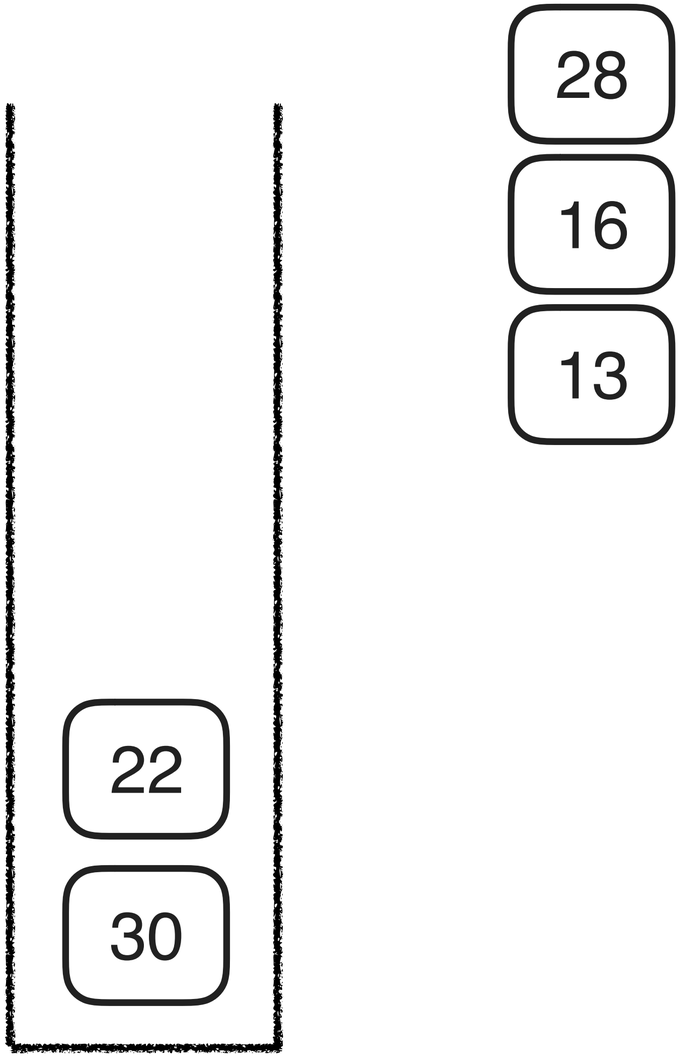




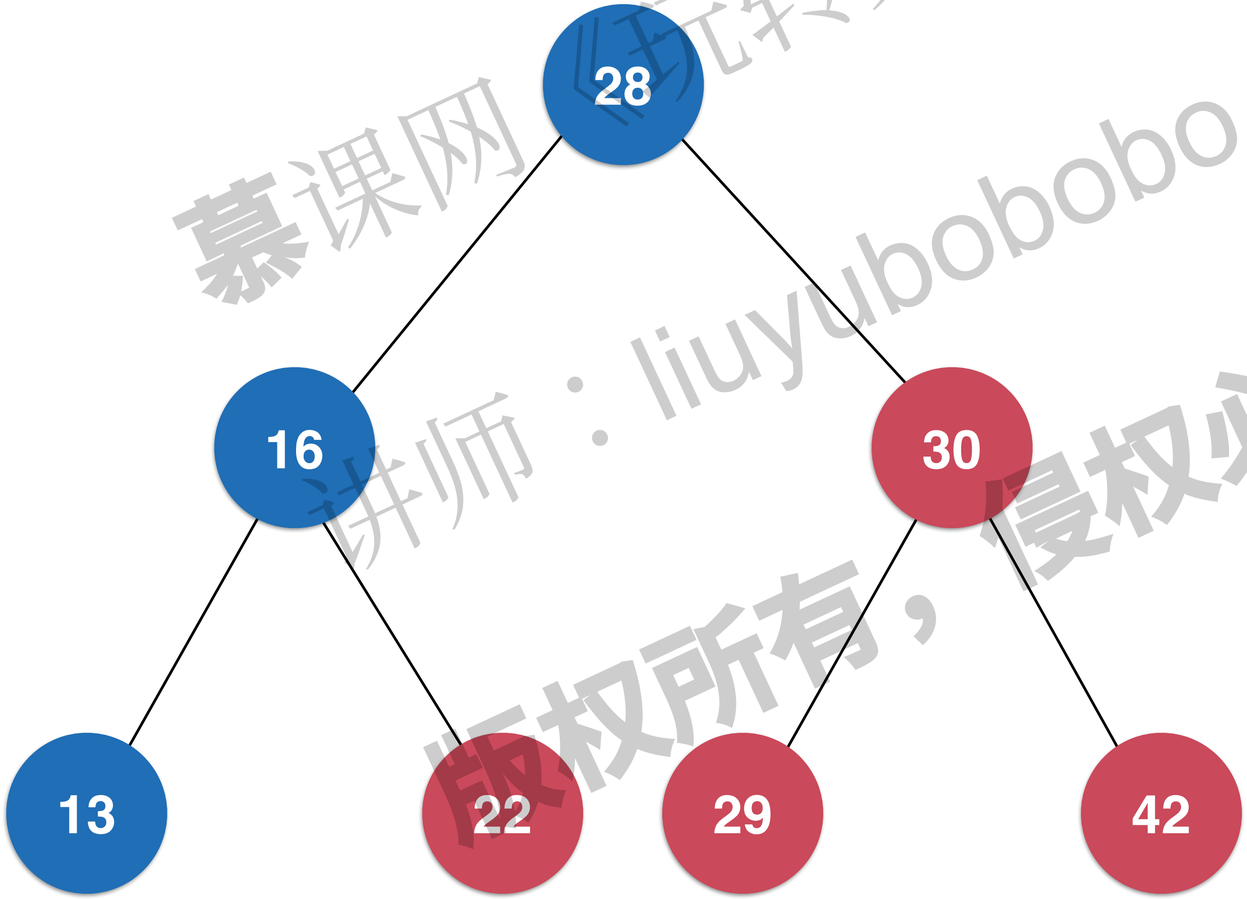


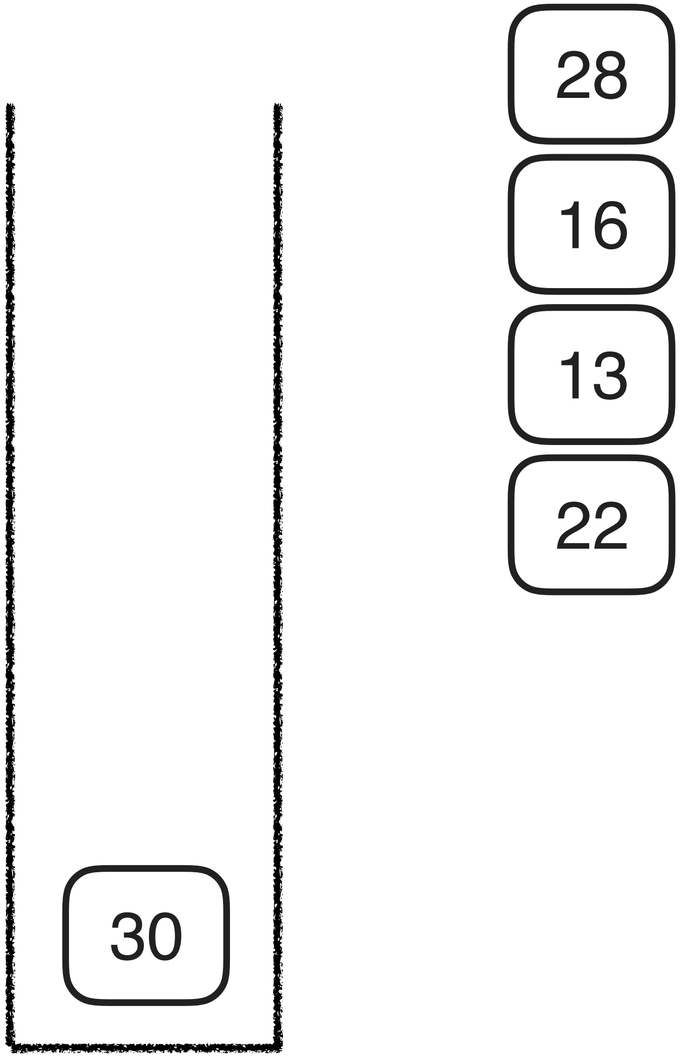
二分搜索树前序遍历的非递归写法



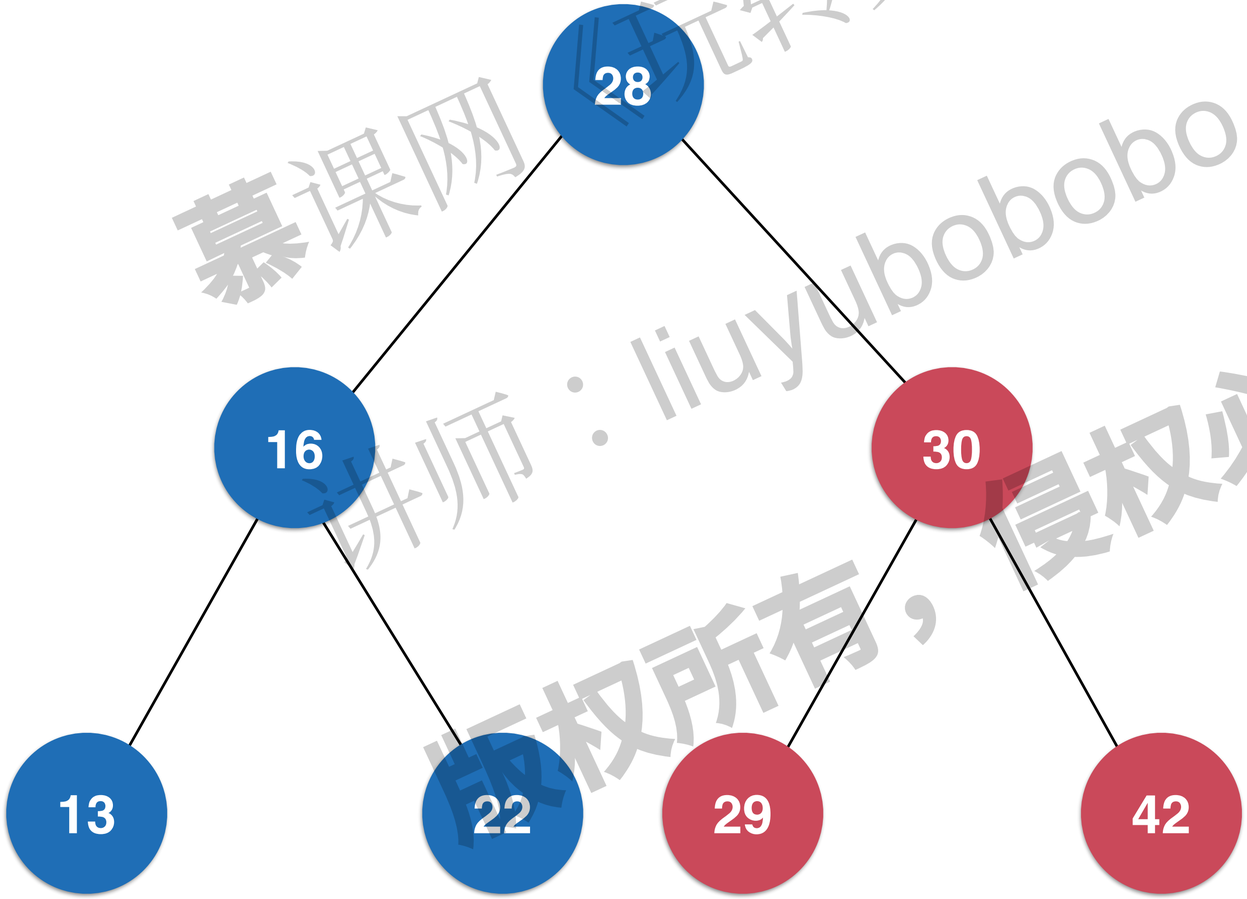


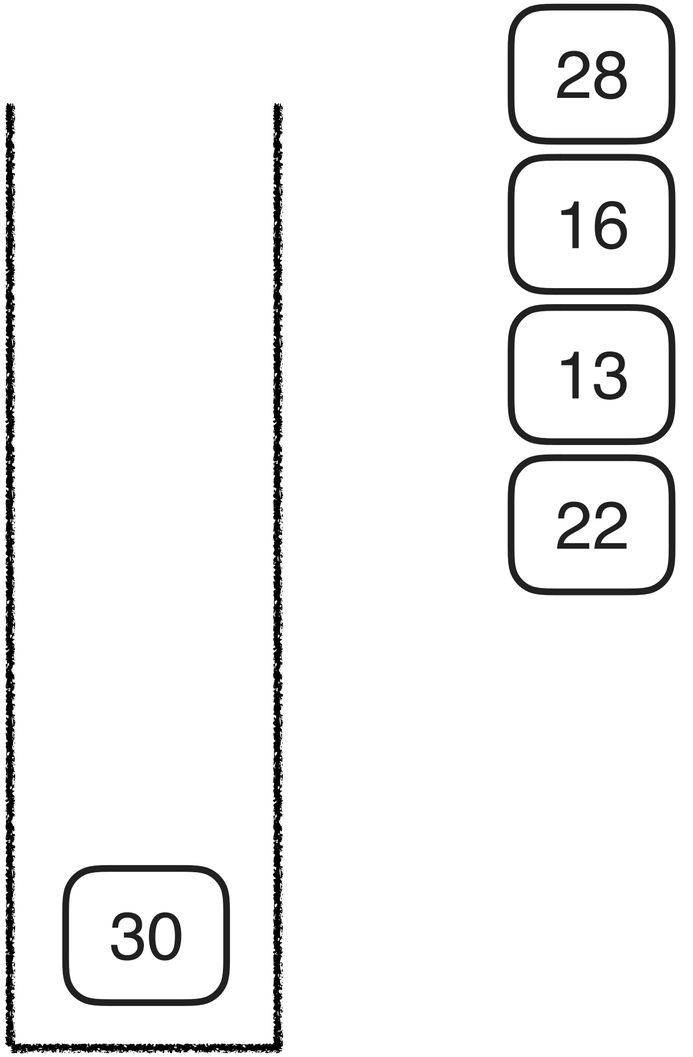
二分搜索树前序遍历的非递归写法



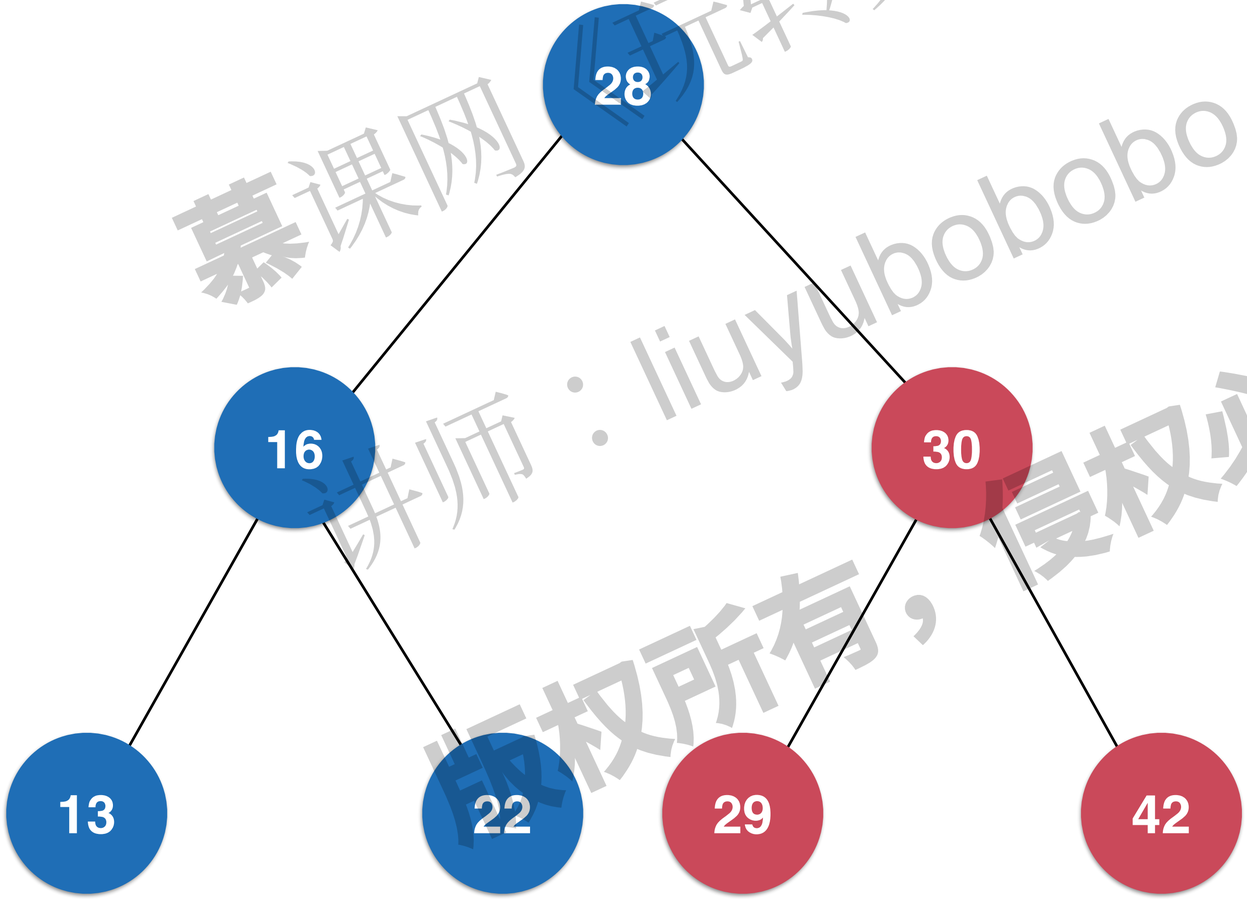


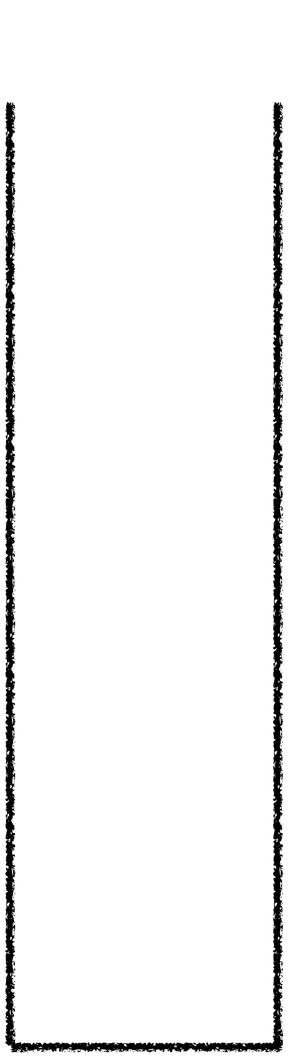
二分搜索树前序遍历的非递归写法



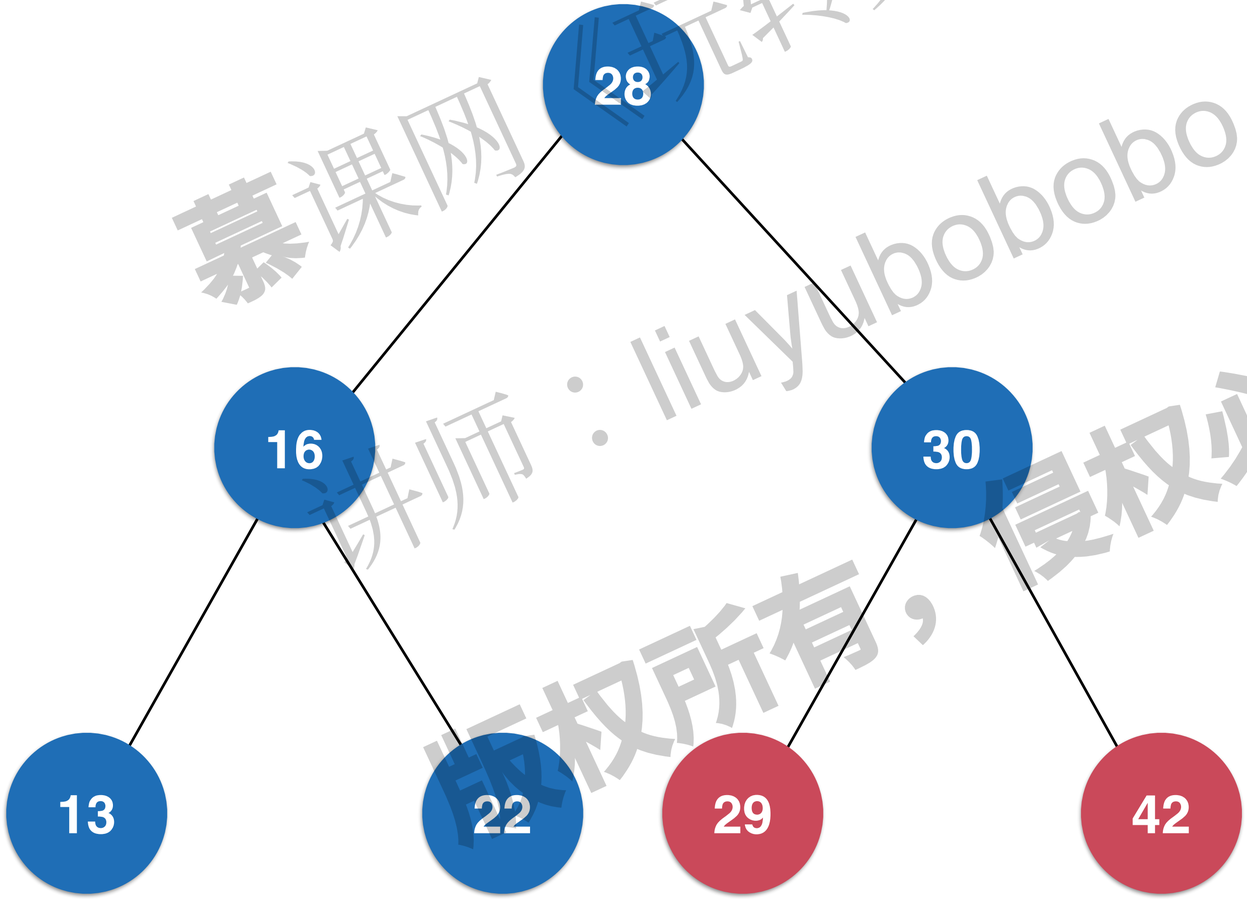


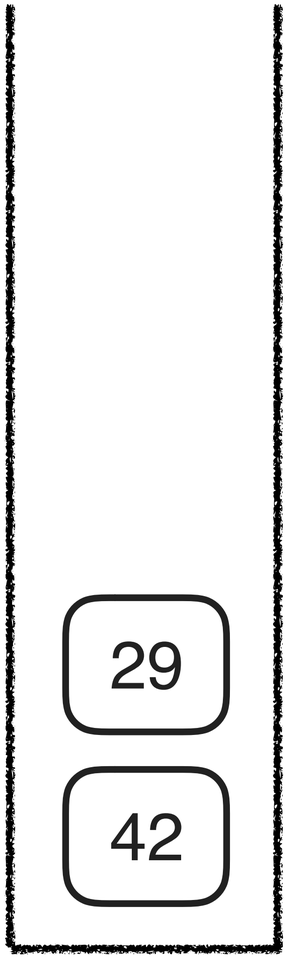
二分搜索树前序遍历的非递归写法



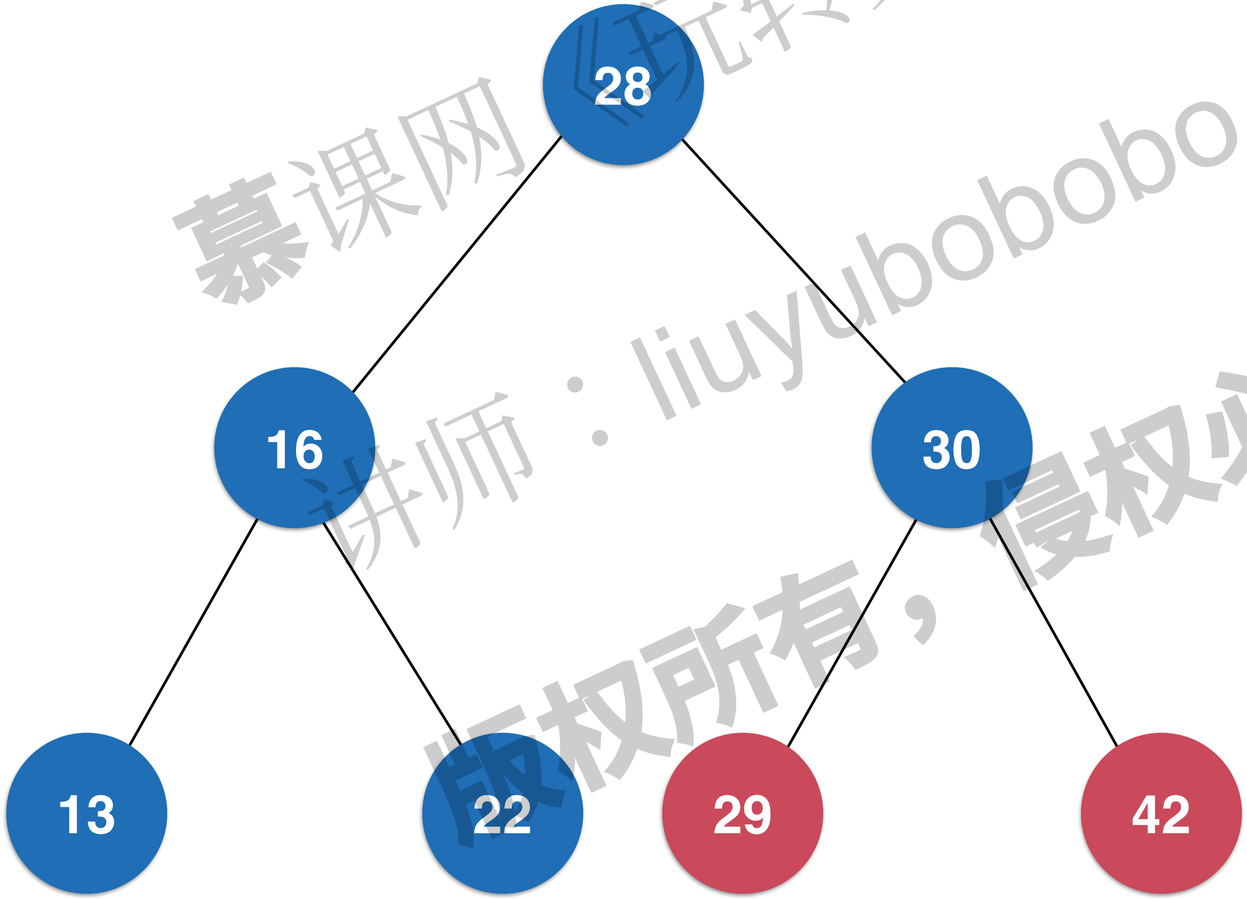


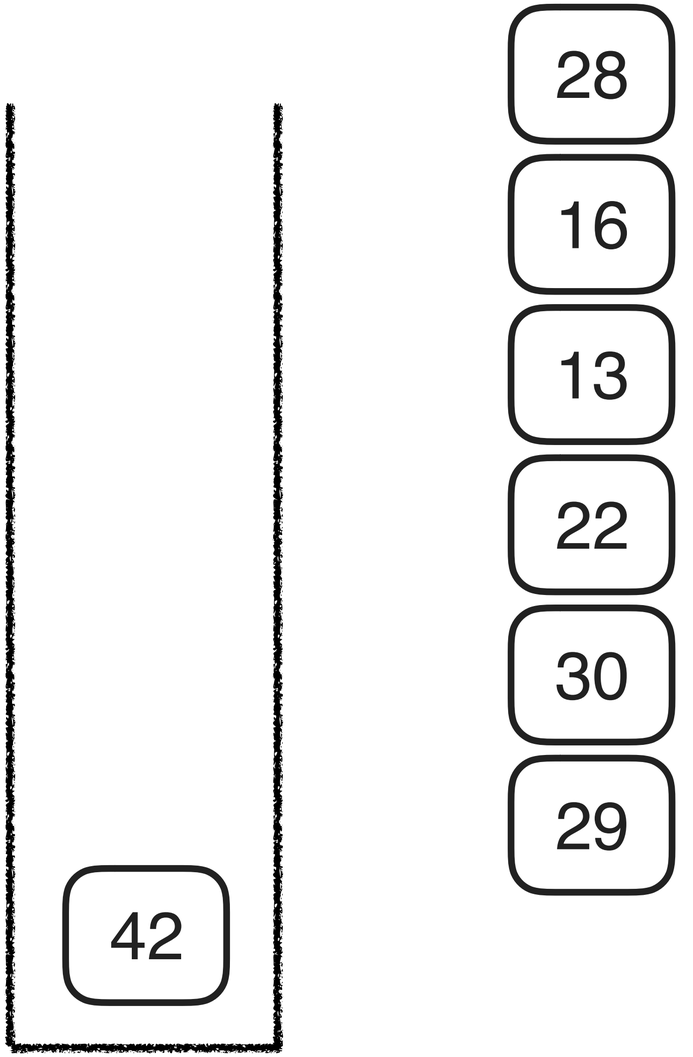
二分搜索树前序遍历的非递归写法



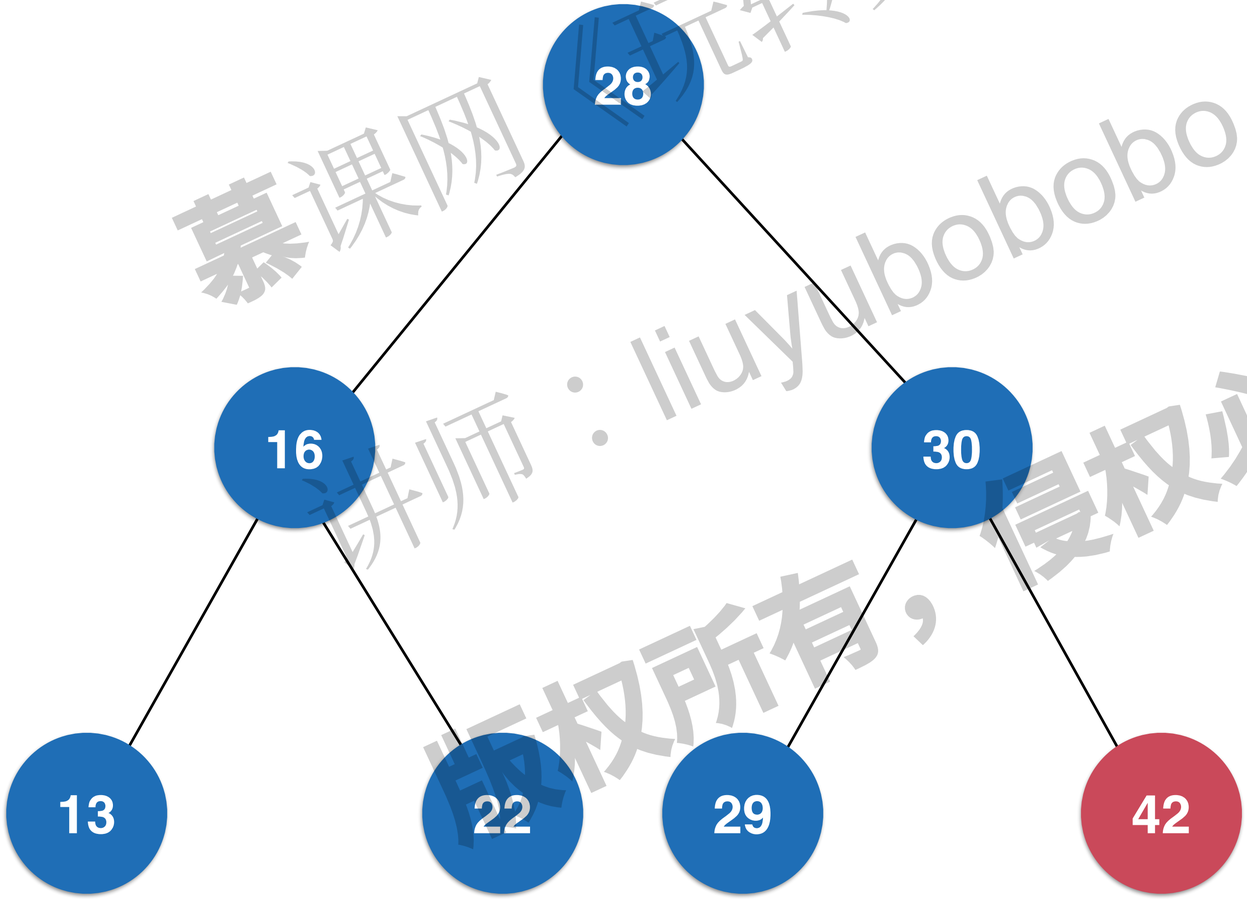


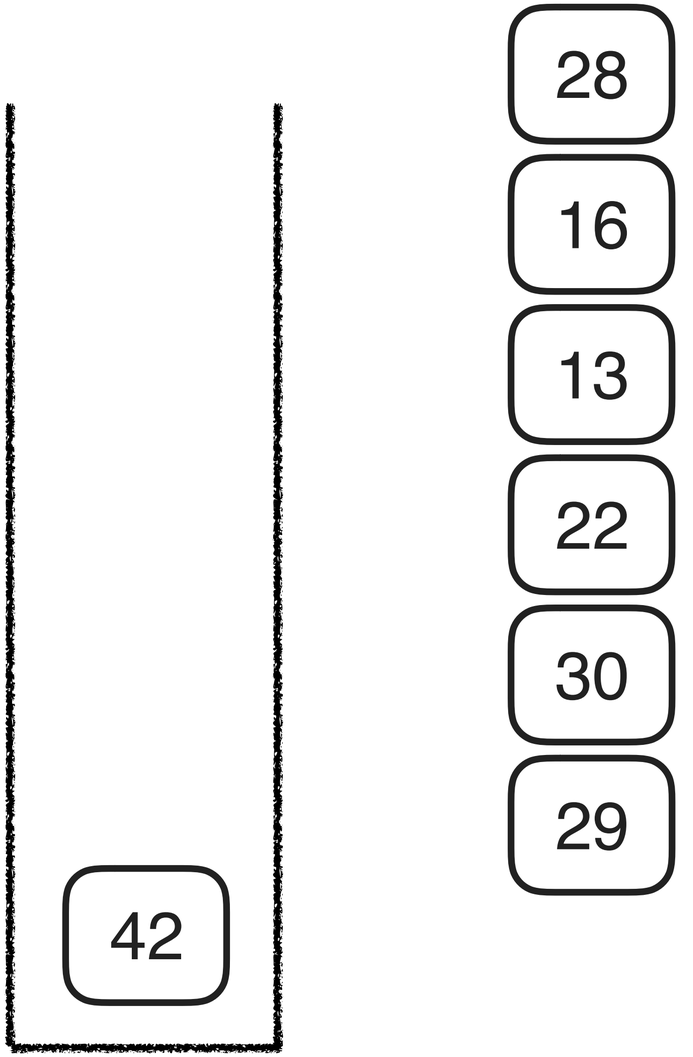
二分搜索树前序遍历的非递归写法



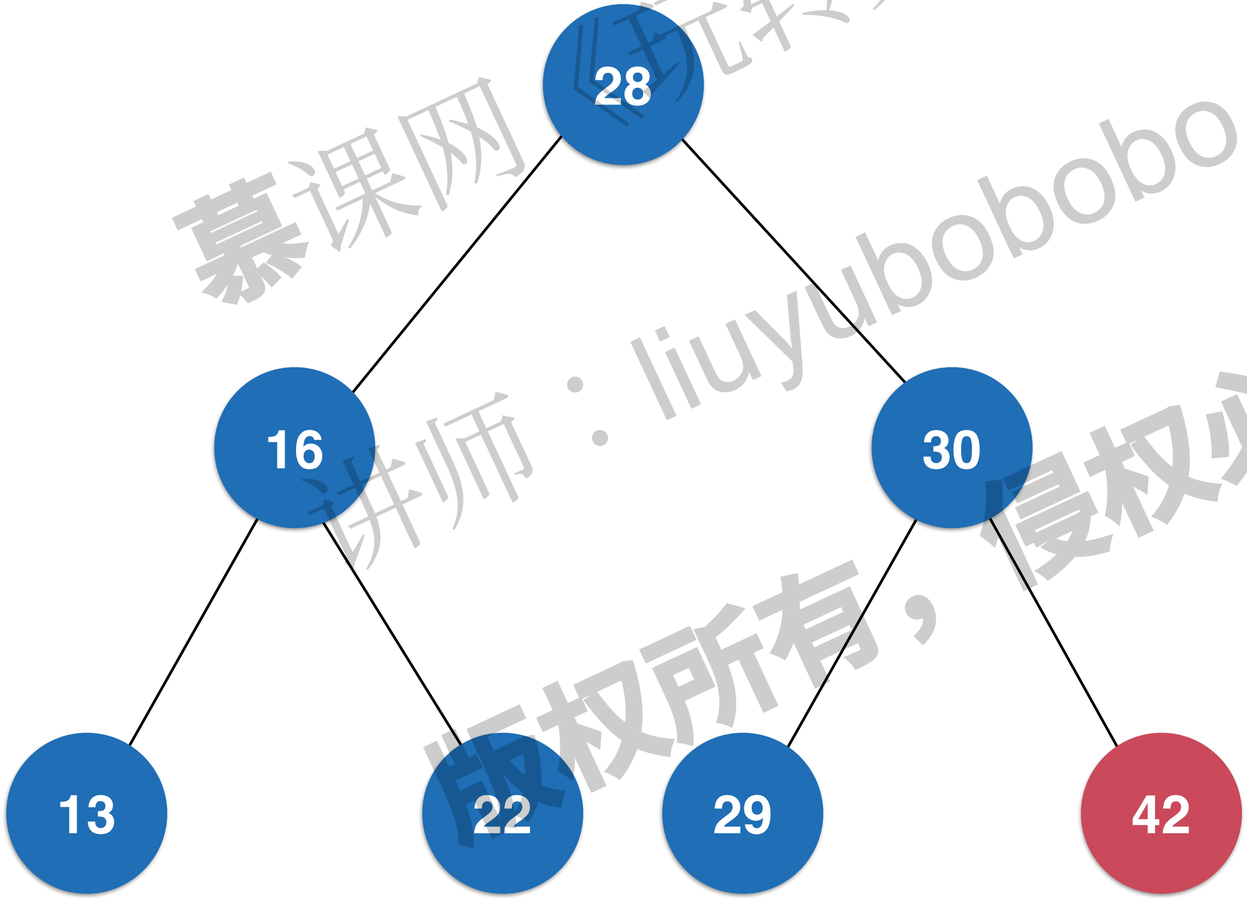


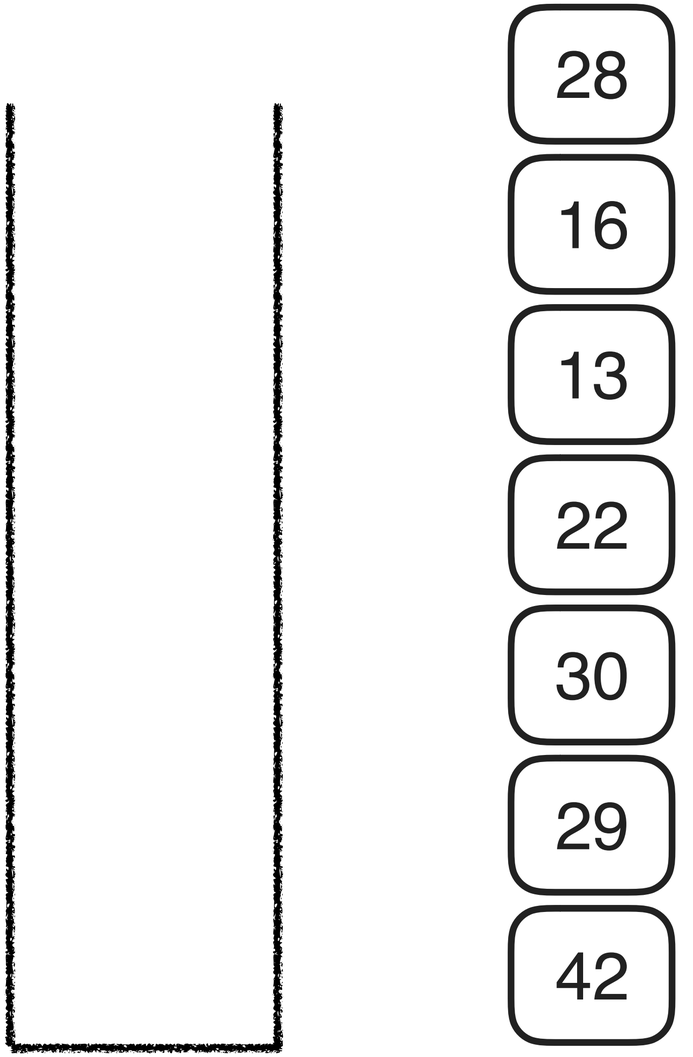
二分搜索树前序遍历的非递归写法



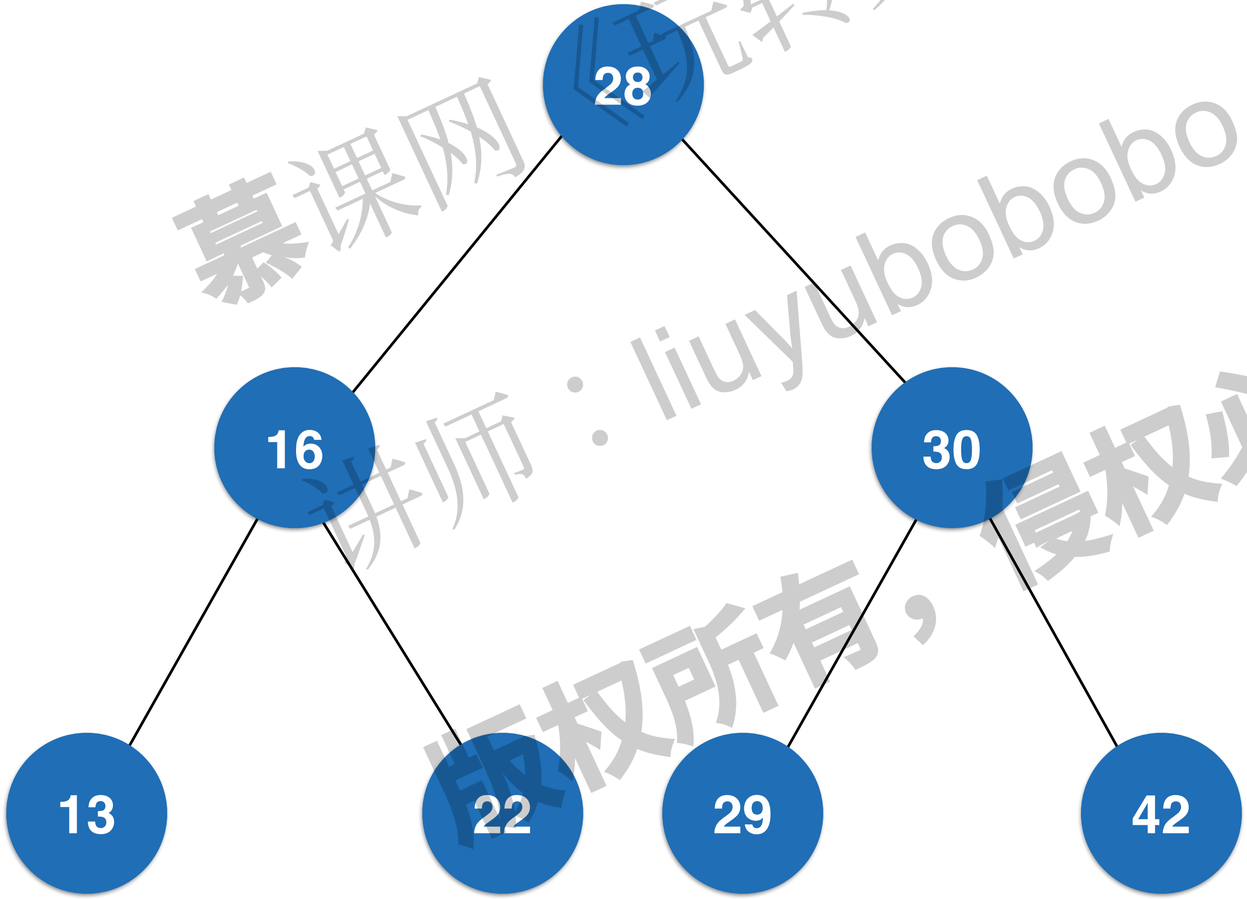


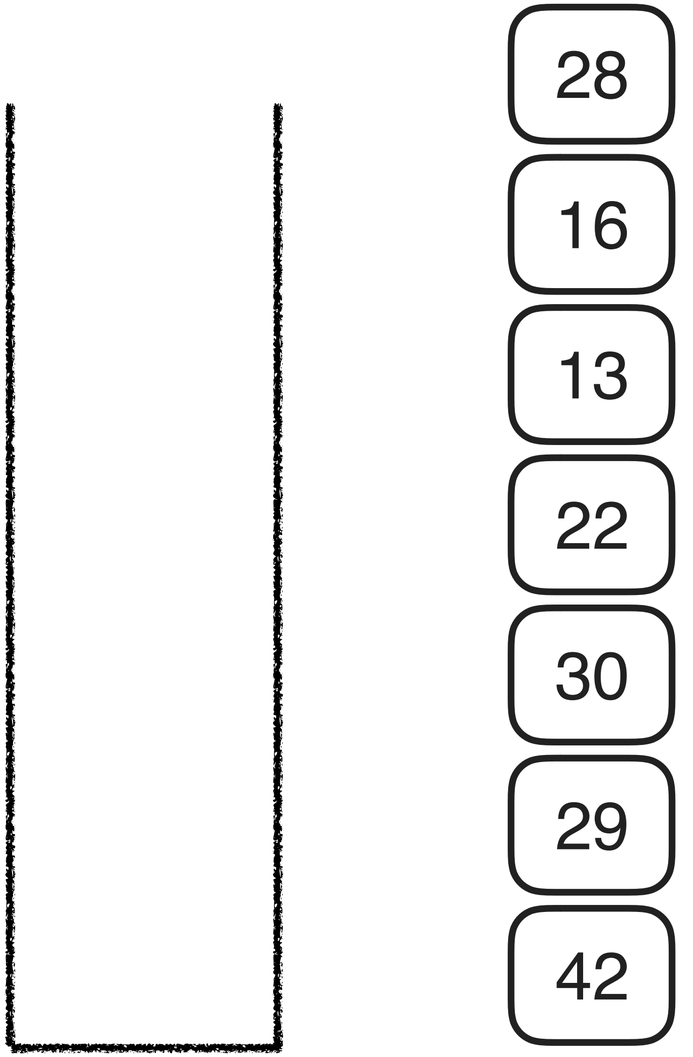
二分搜索树前序遍历的非递归写法





二分搜索树前序遍历的非递归写法





实践：二分搜索树前序遍历的非递归写法

二分搜索树遍历的非递归实现

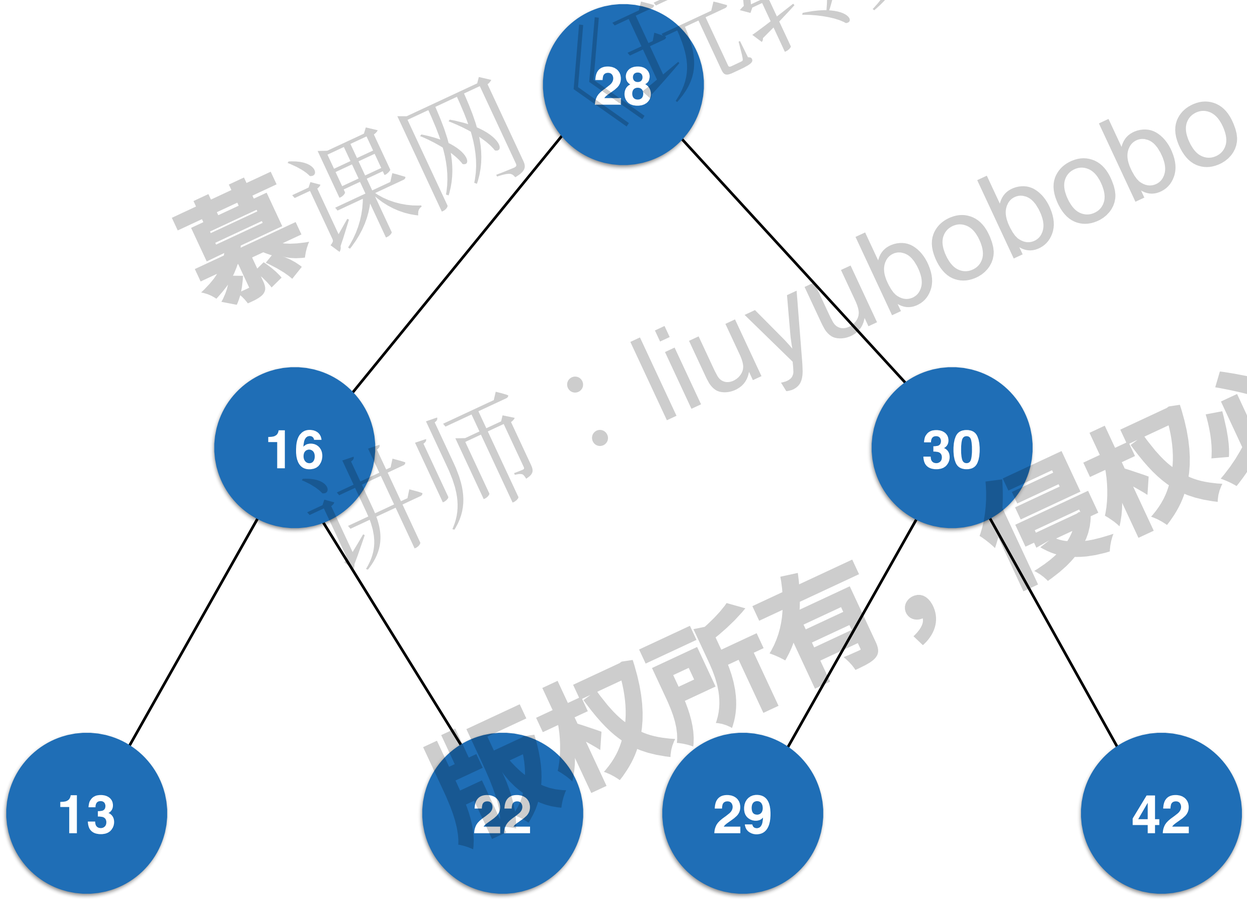
- 二分搜索树遍历的非递归实现，比递归实现复杂很多
- 中序遍历和后序遍历的非递归实现更复杂
- 中序遍历和后序遍历的非递归实现，实际应用不广
- 中序遍历和后序遍历的非递归实现留做练习

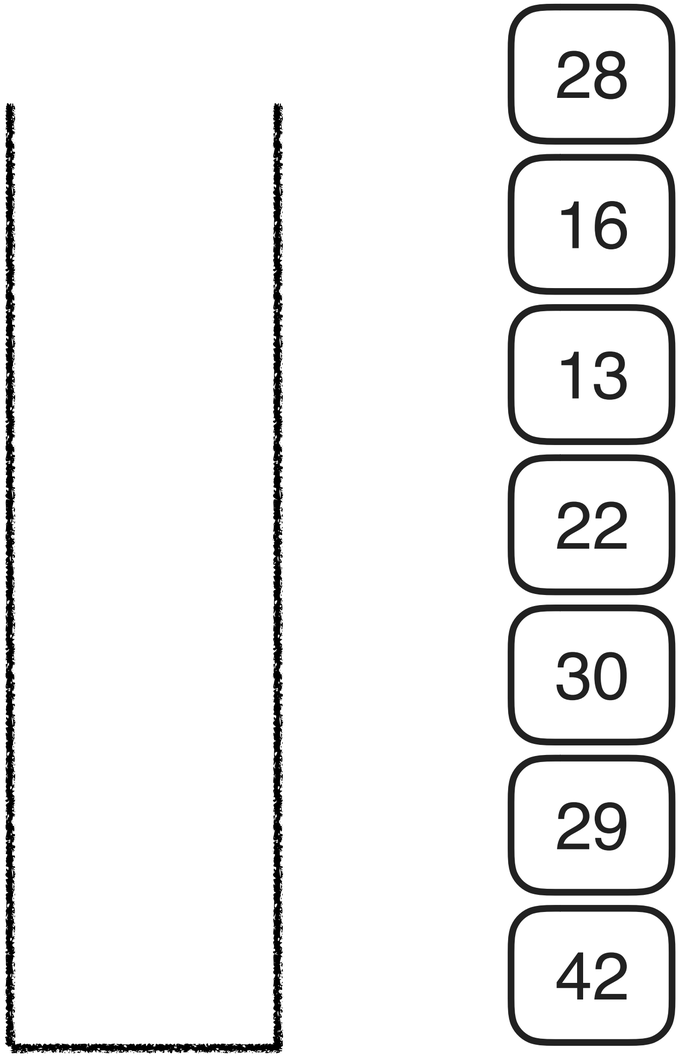
二分搜索树遍历的非递归实现

- 经典教科书的写法

- 《玩转算法面试》中完全模拟系统栈的写法

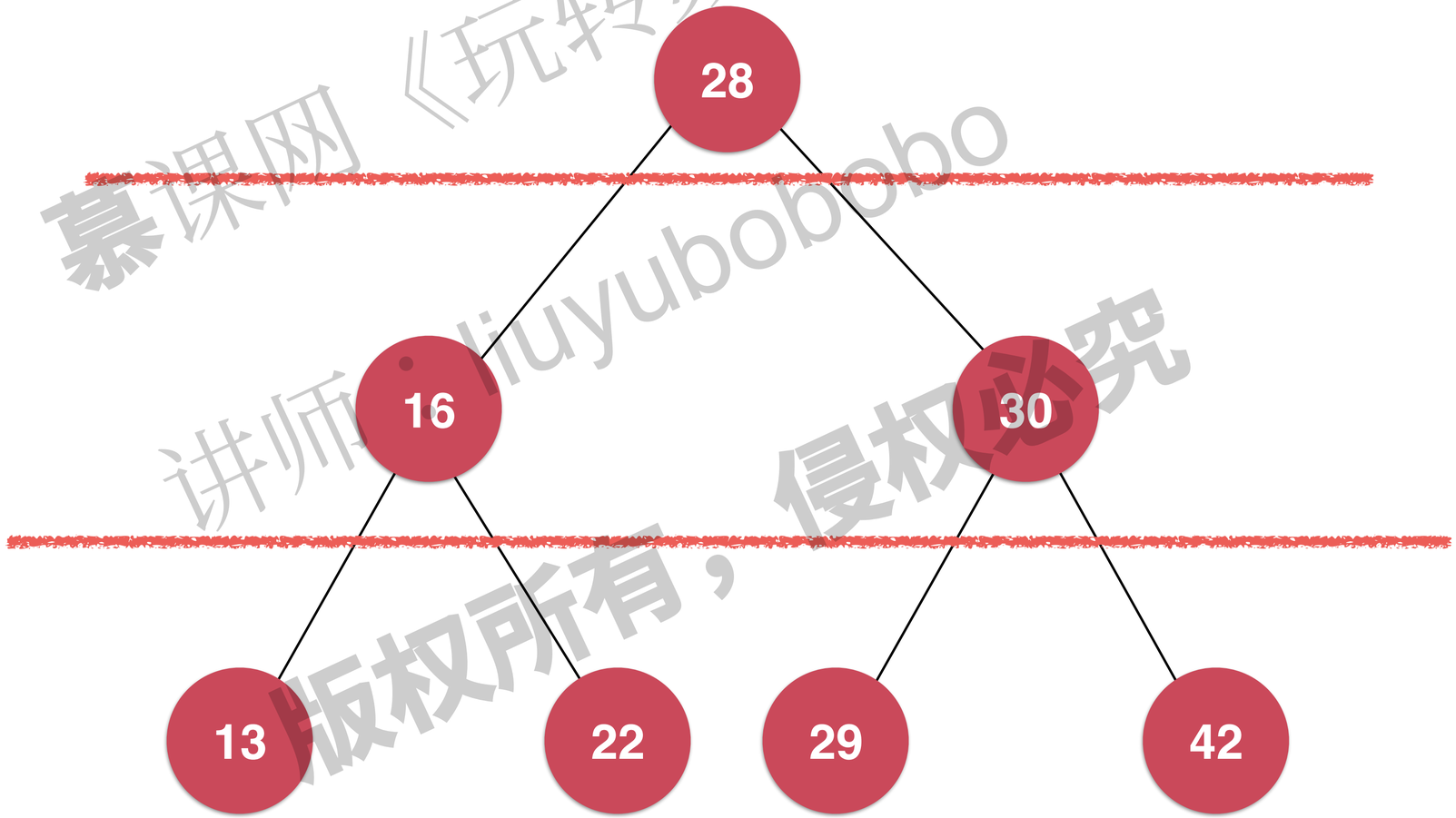
二分搜索树前序遍历的非递归写法



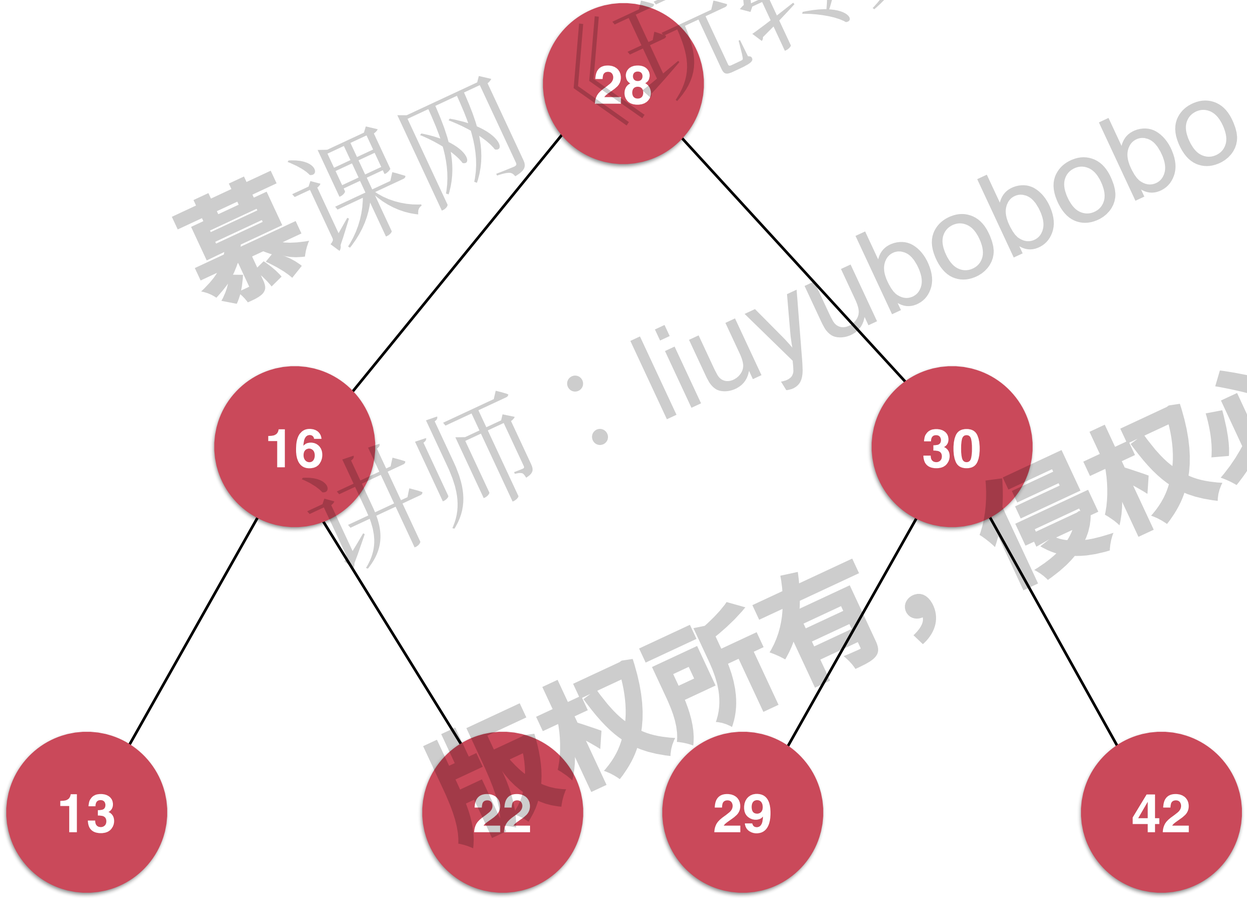


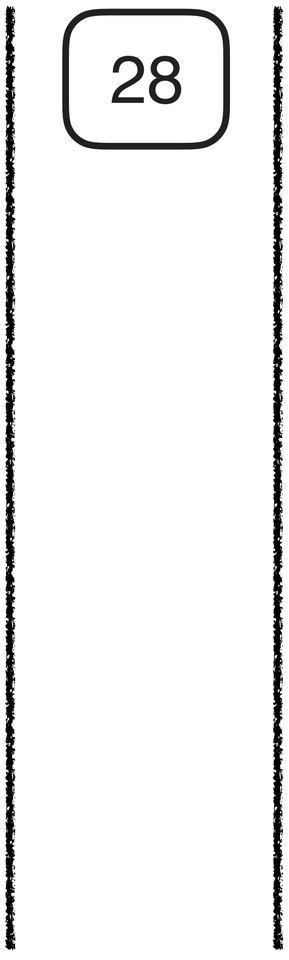
二分搜索树的层序遍历

二分搜索树的层序遍历

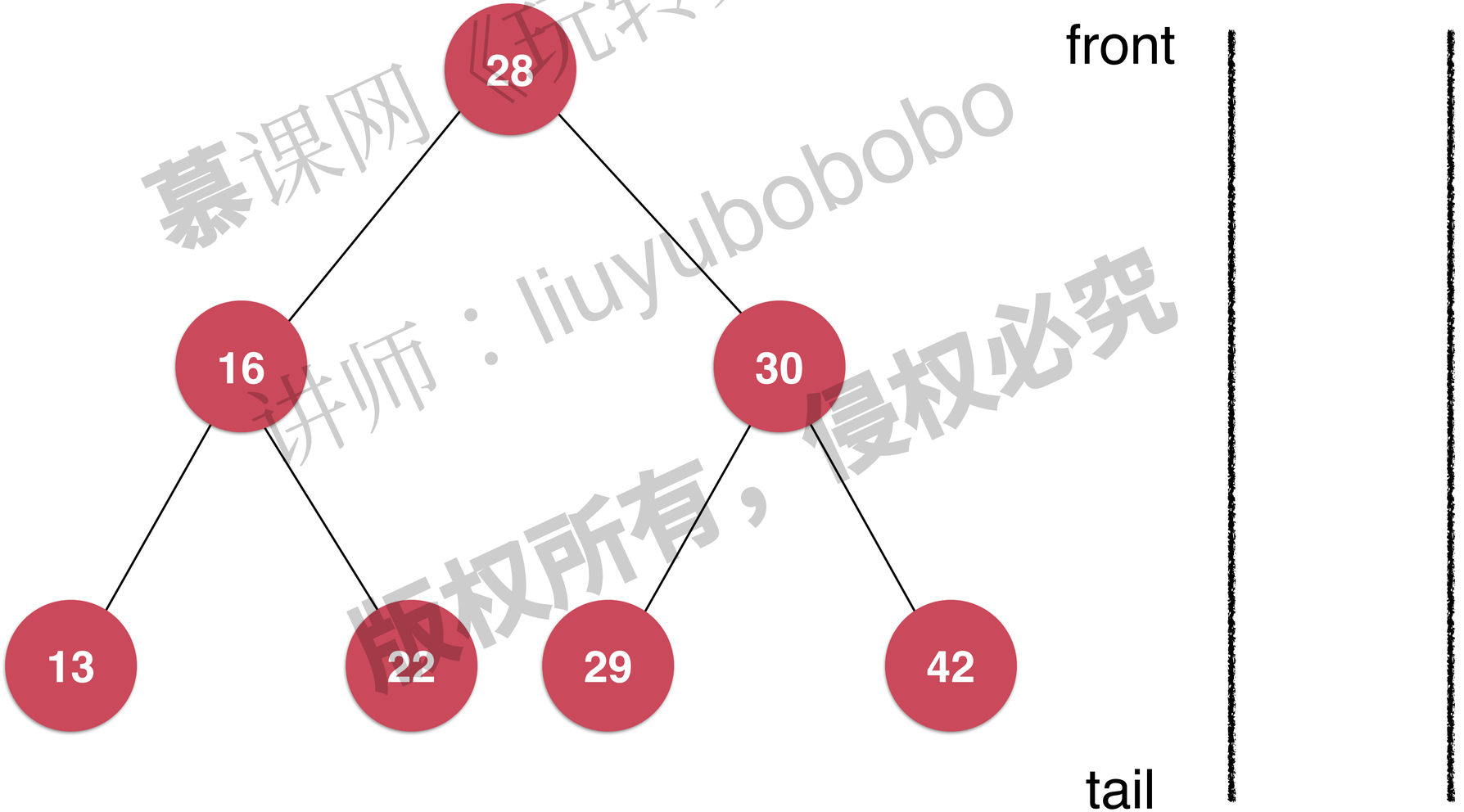


二分搜索树的层序遍历

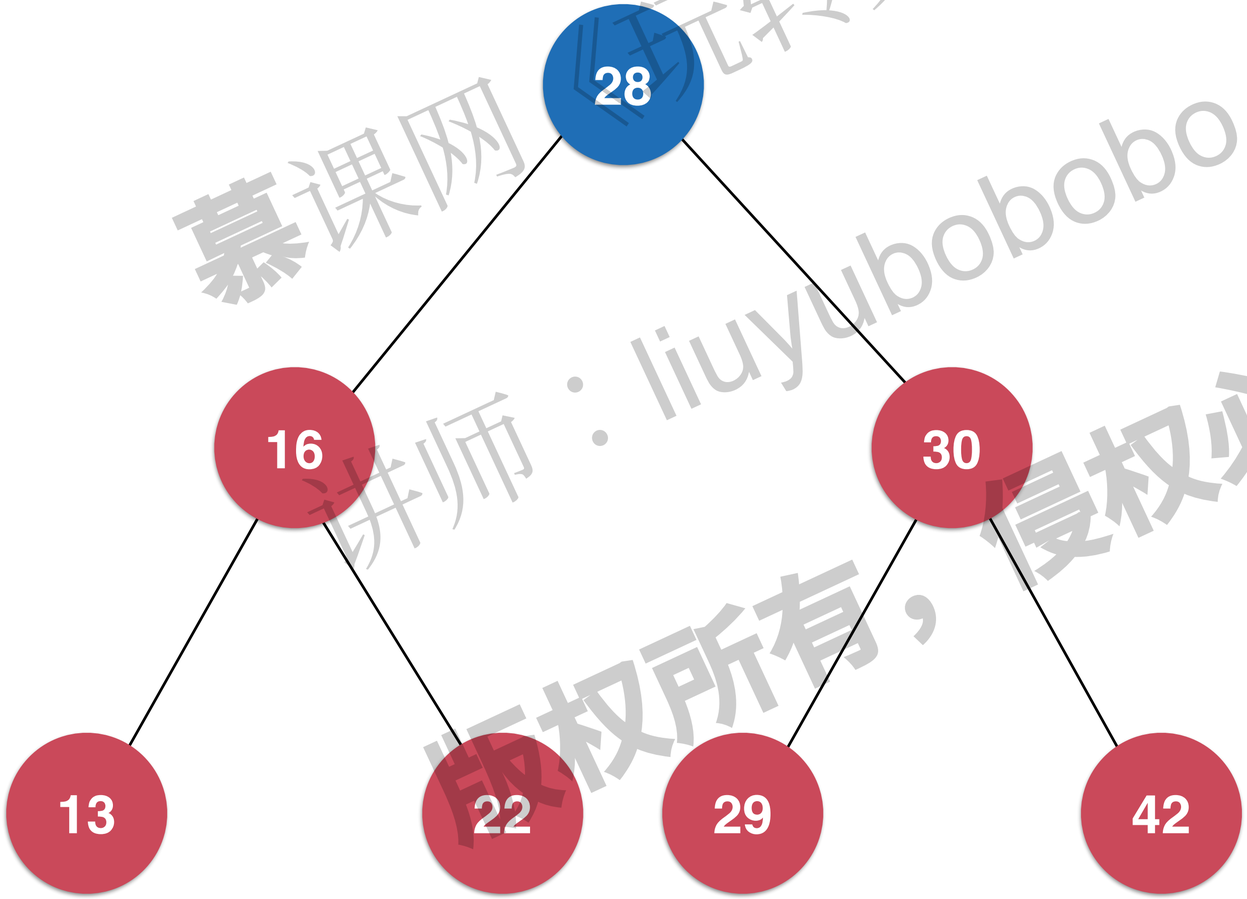


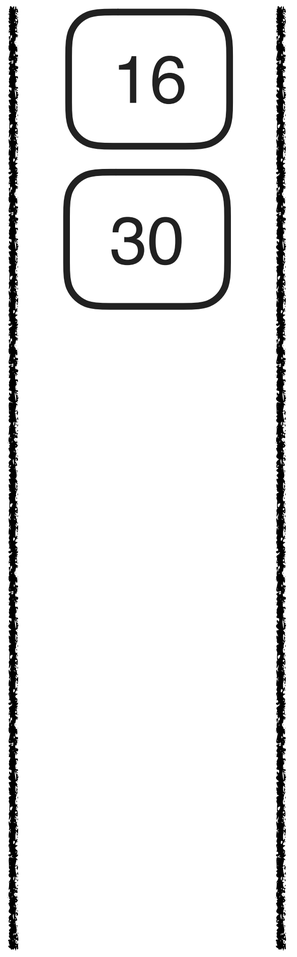


二分搜索树的层序遍历



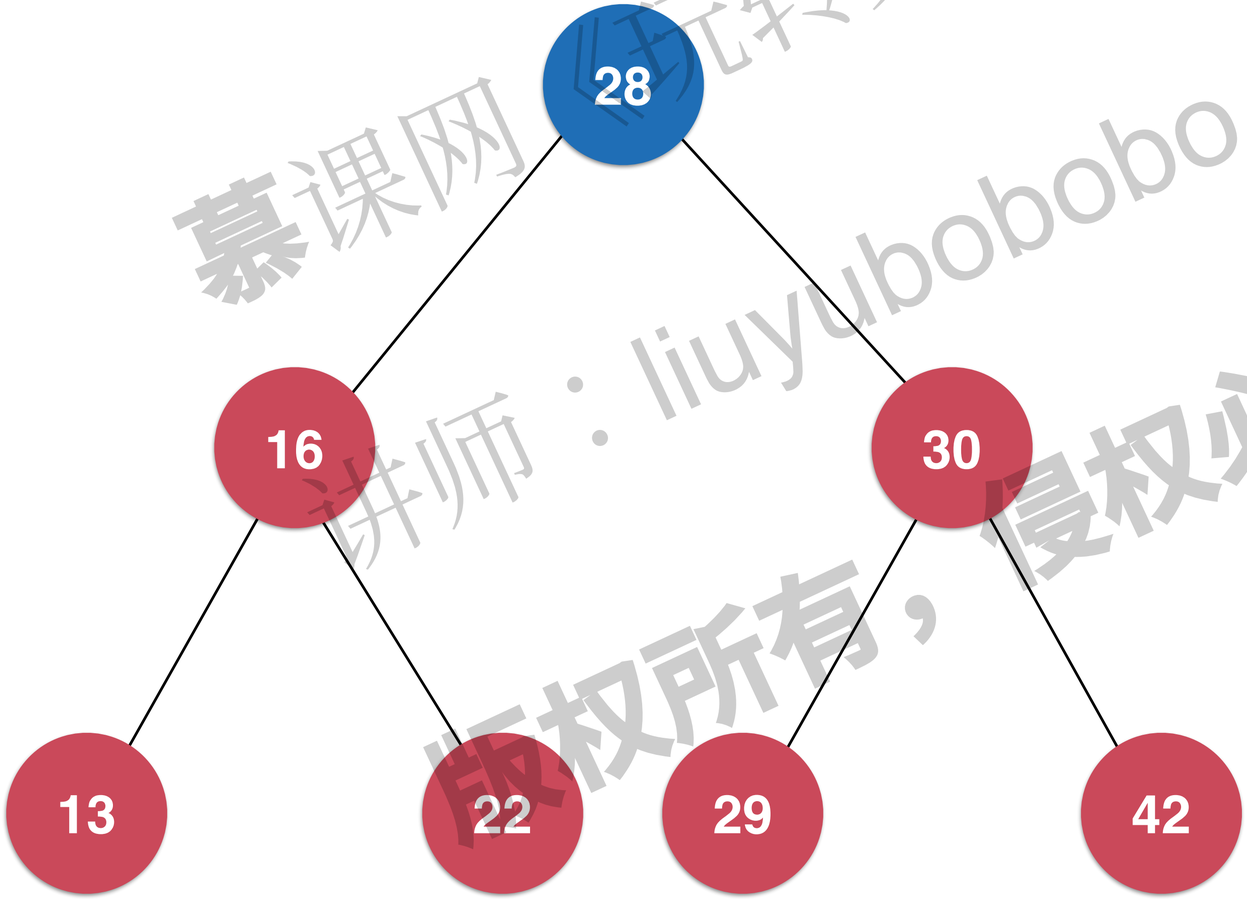
二分搜索树的层序遍历





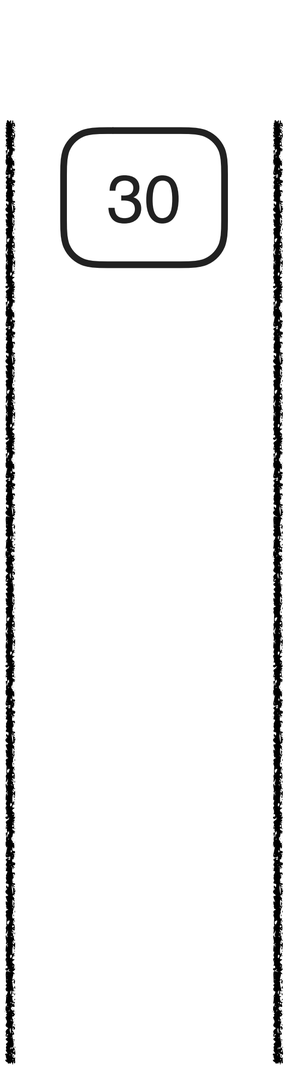
28

二分搜索树的层序遍历



front

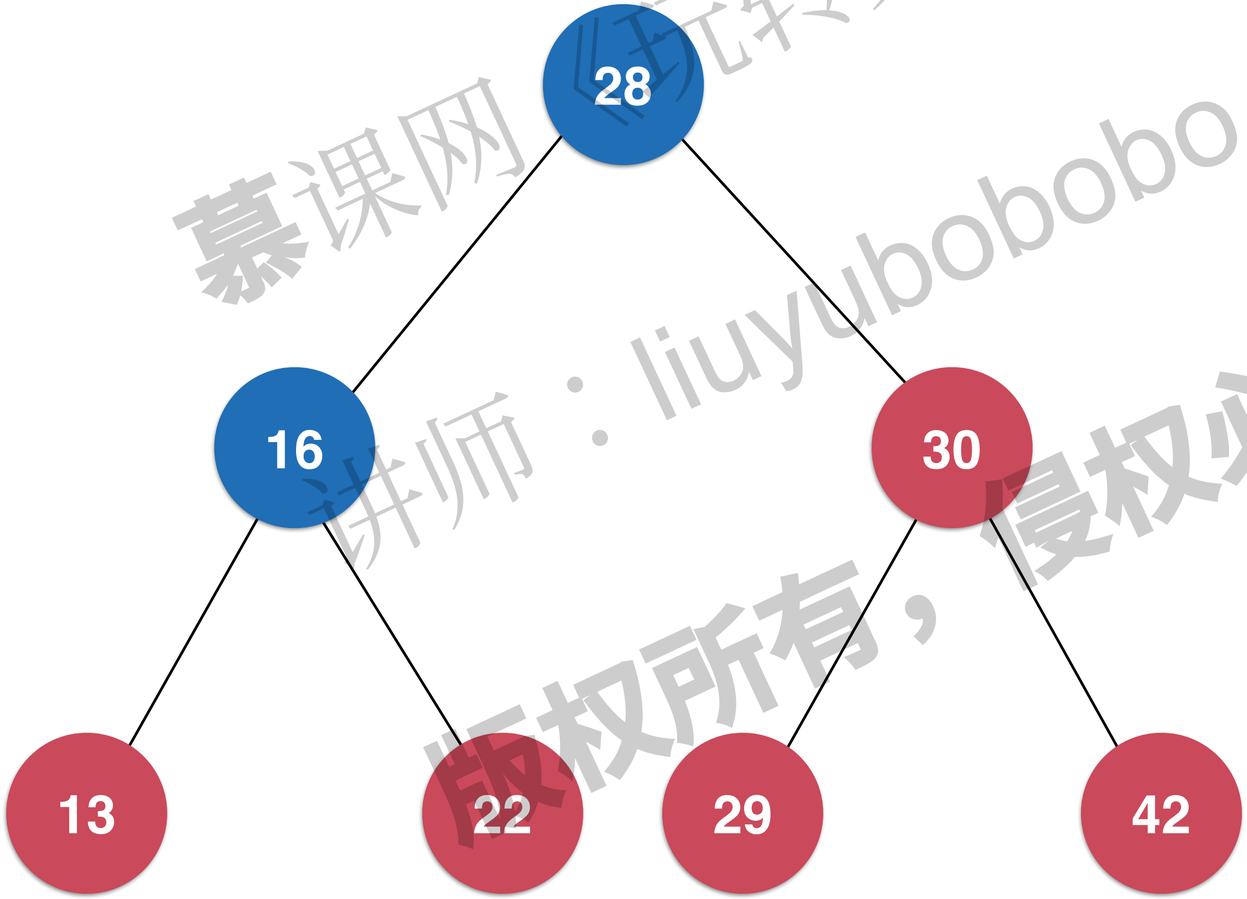
tail

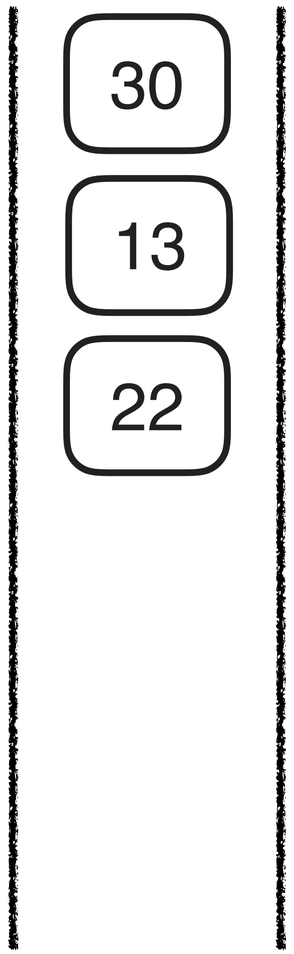


28

16

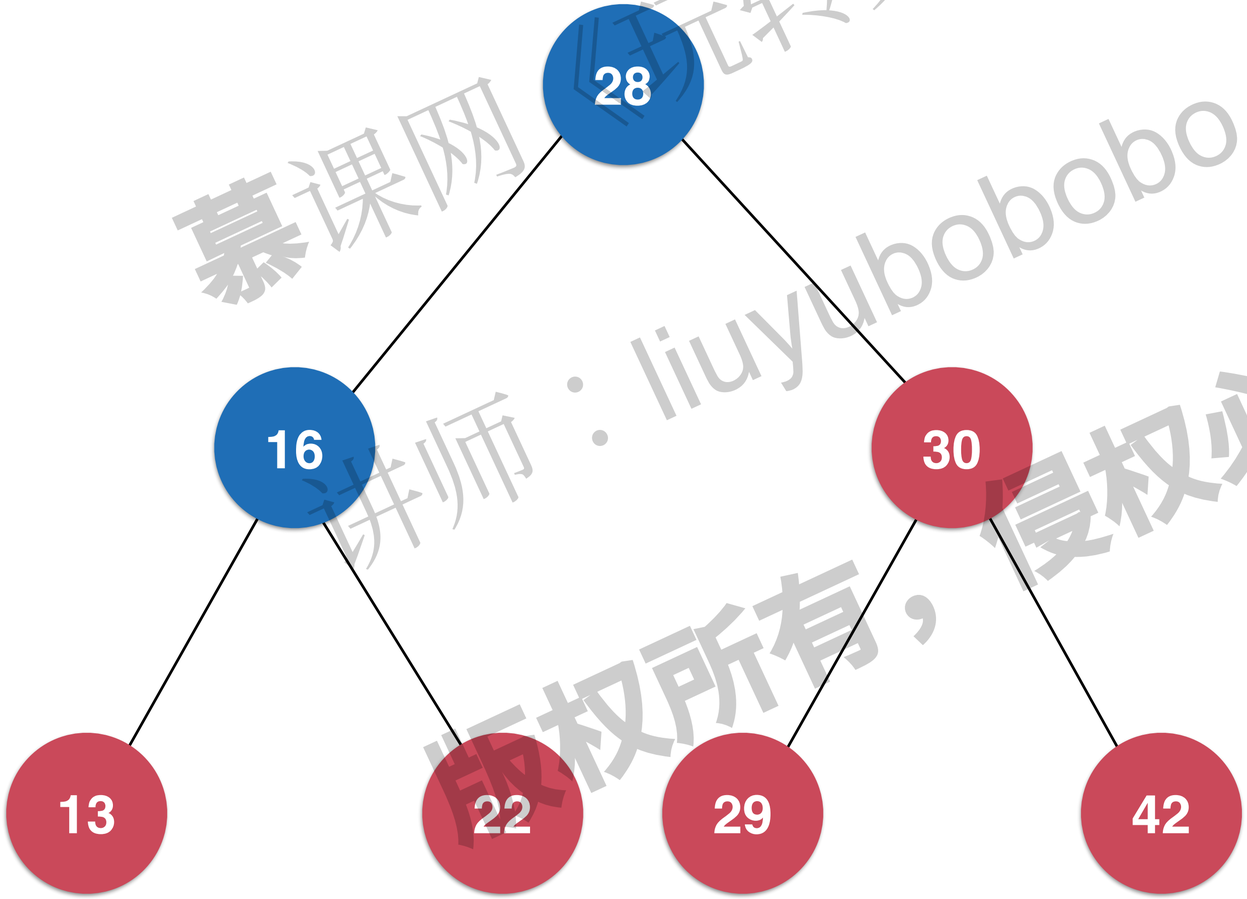
二分搜索树的层序遍历

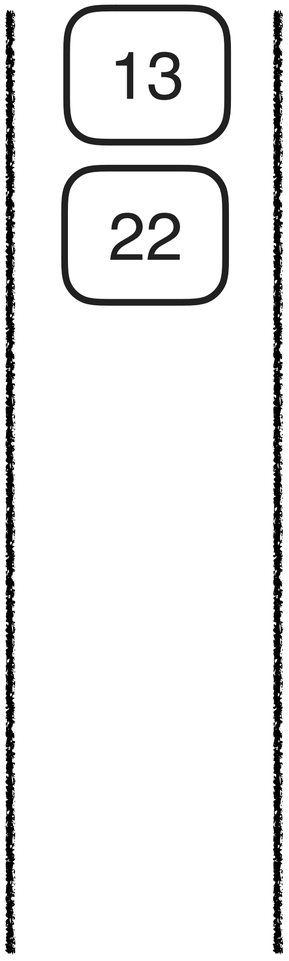


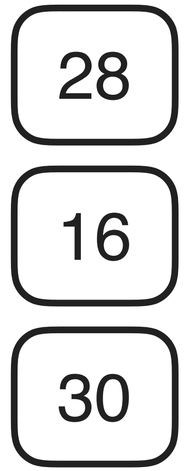


28
16

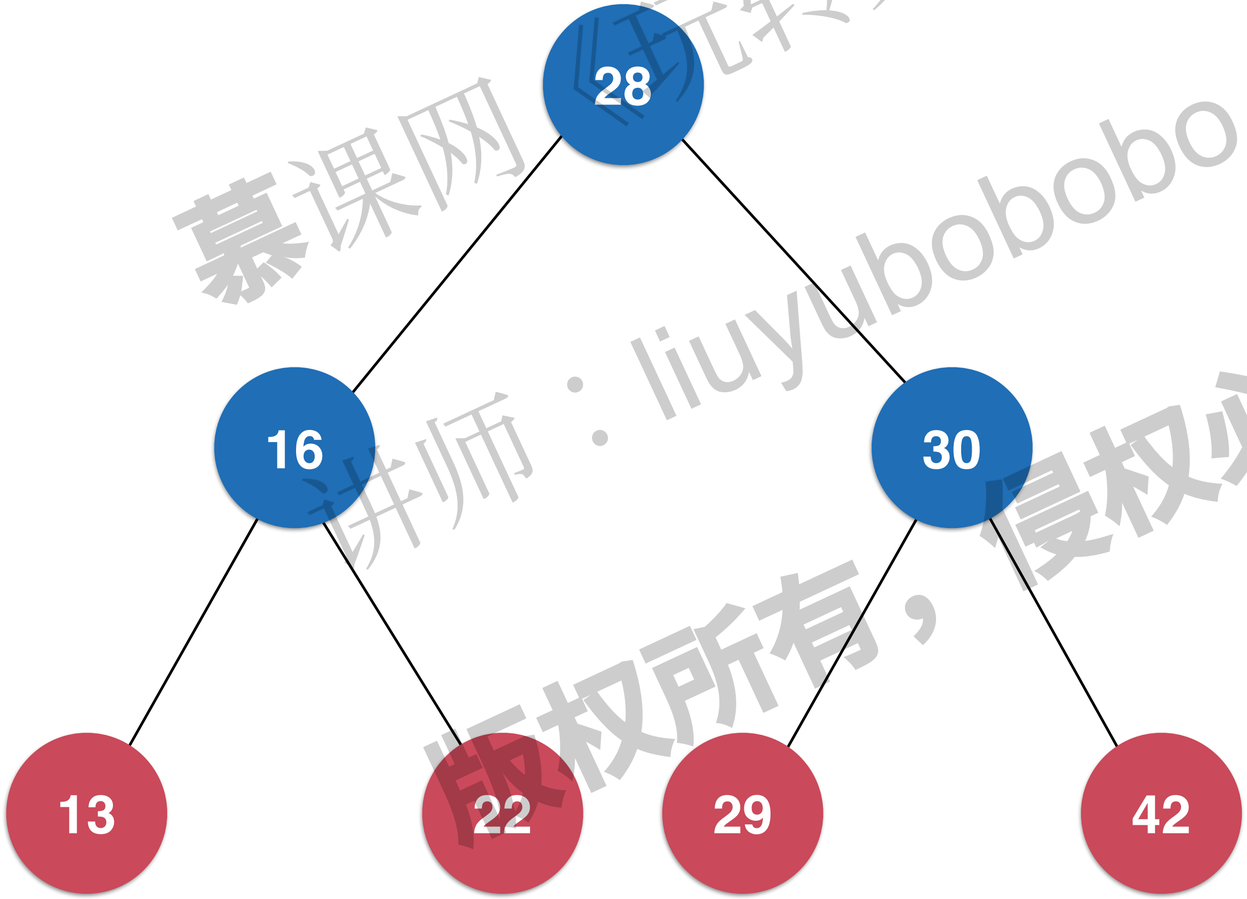
二分搜索树的层序遍历

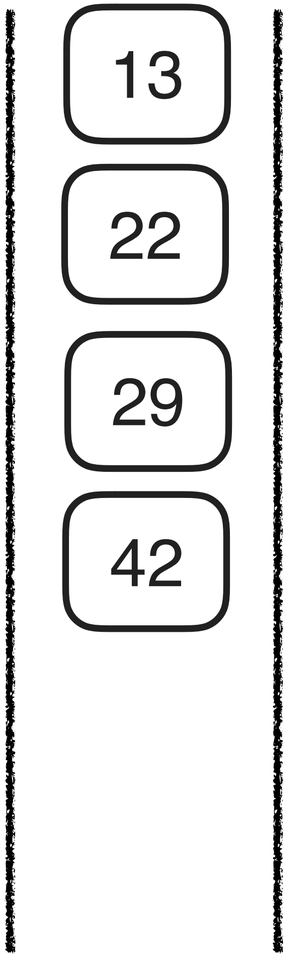


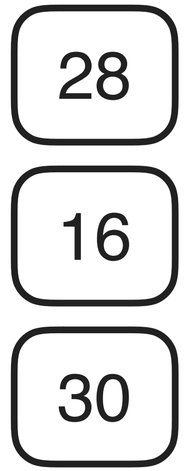




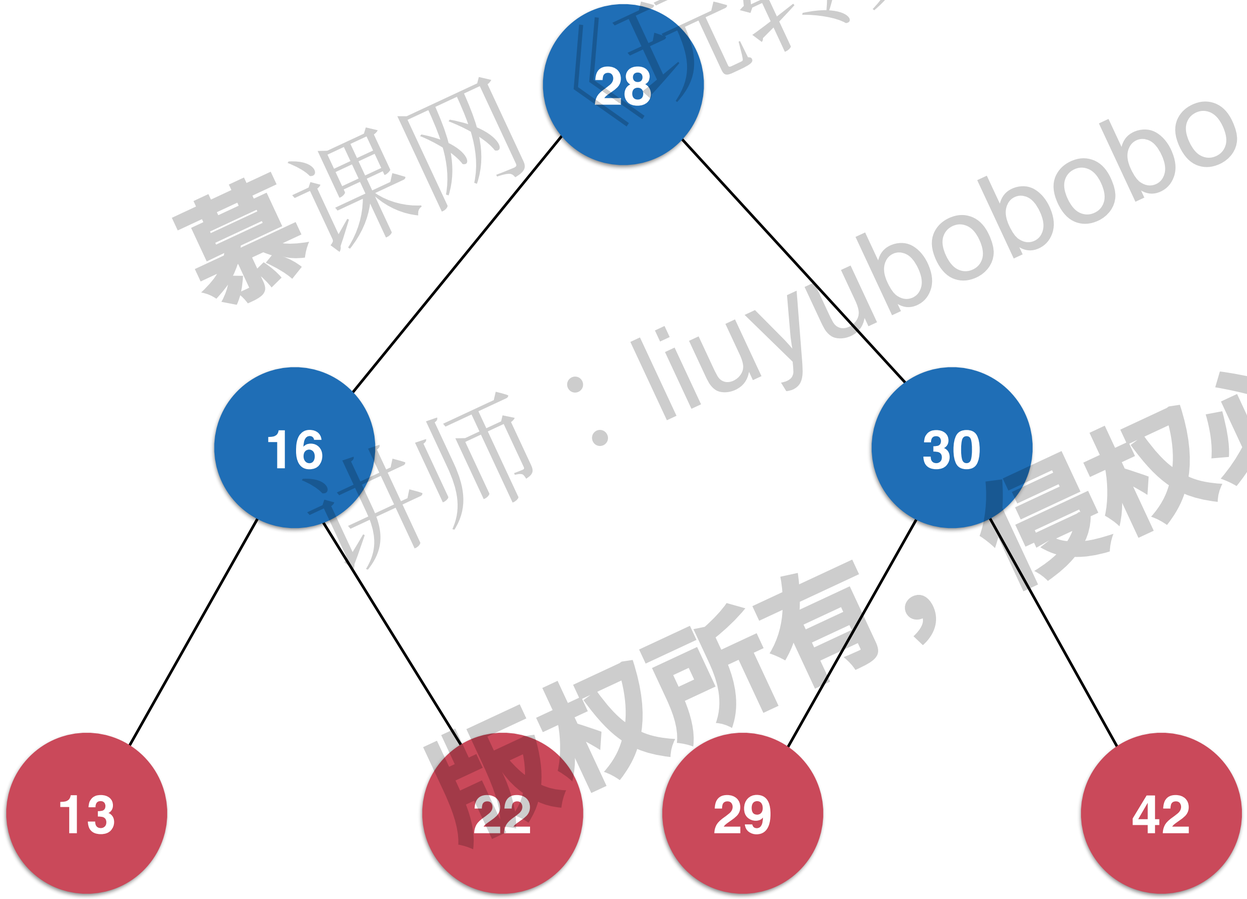
二分搜索树的层序遍历





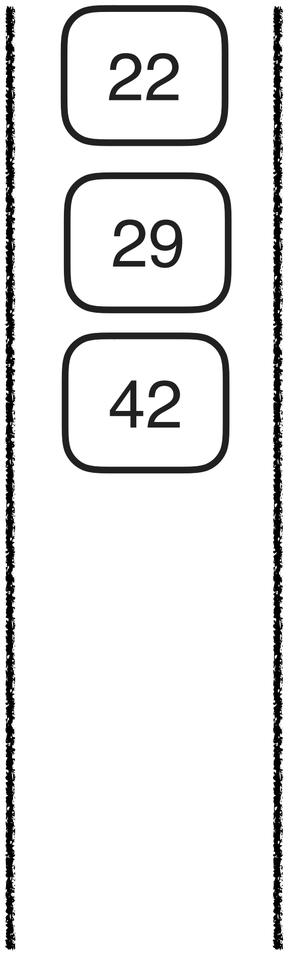


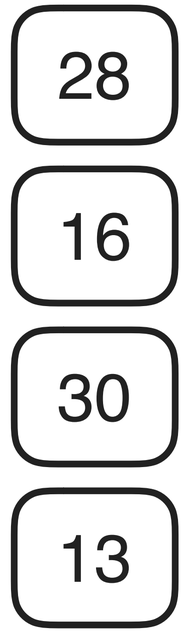
二分搜索树的层序遍历



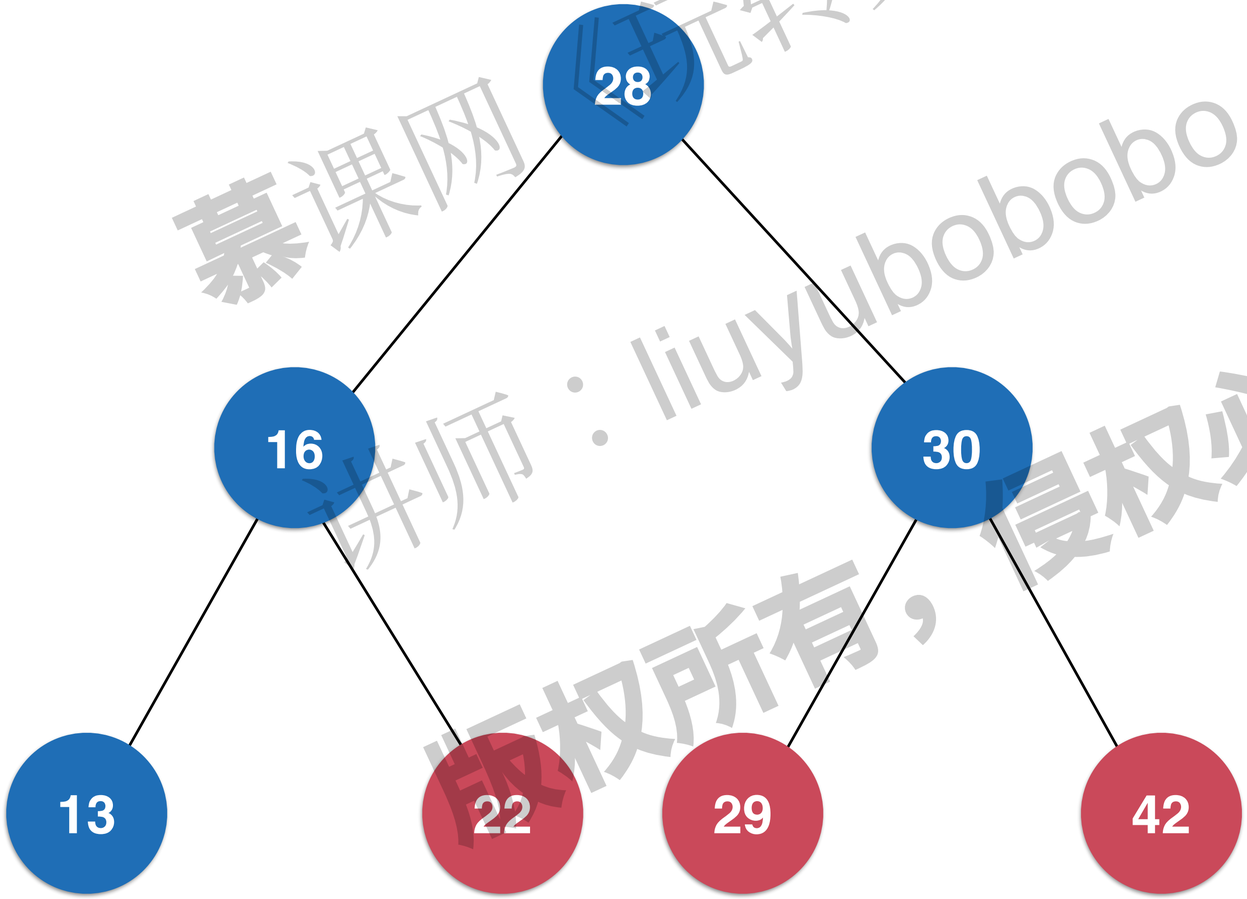
front

tail



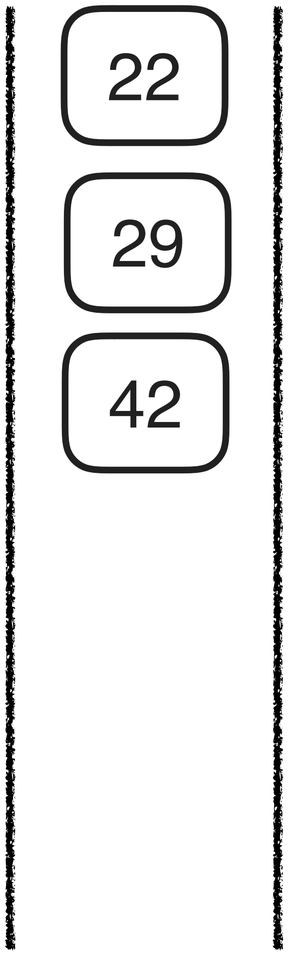


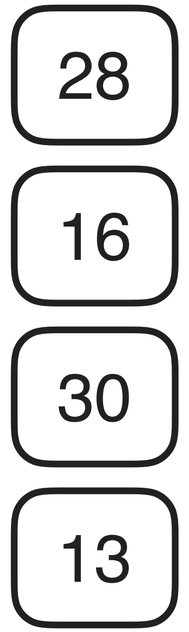
二分搜索树的层序遍历



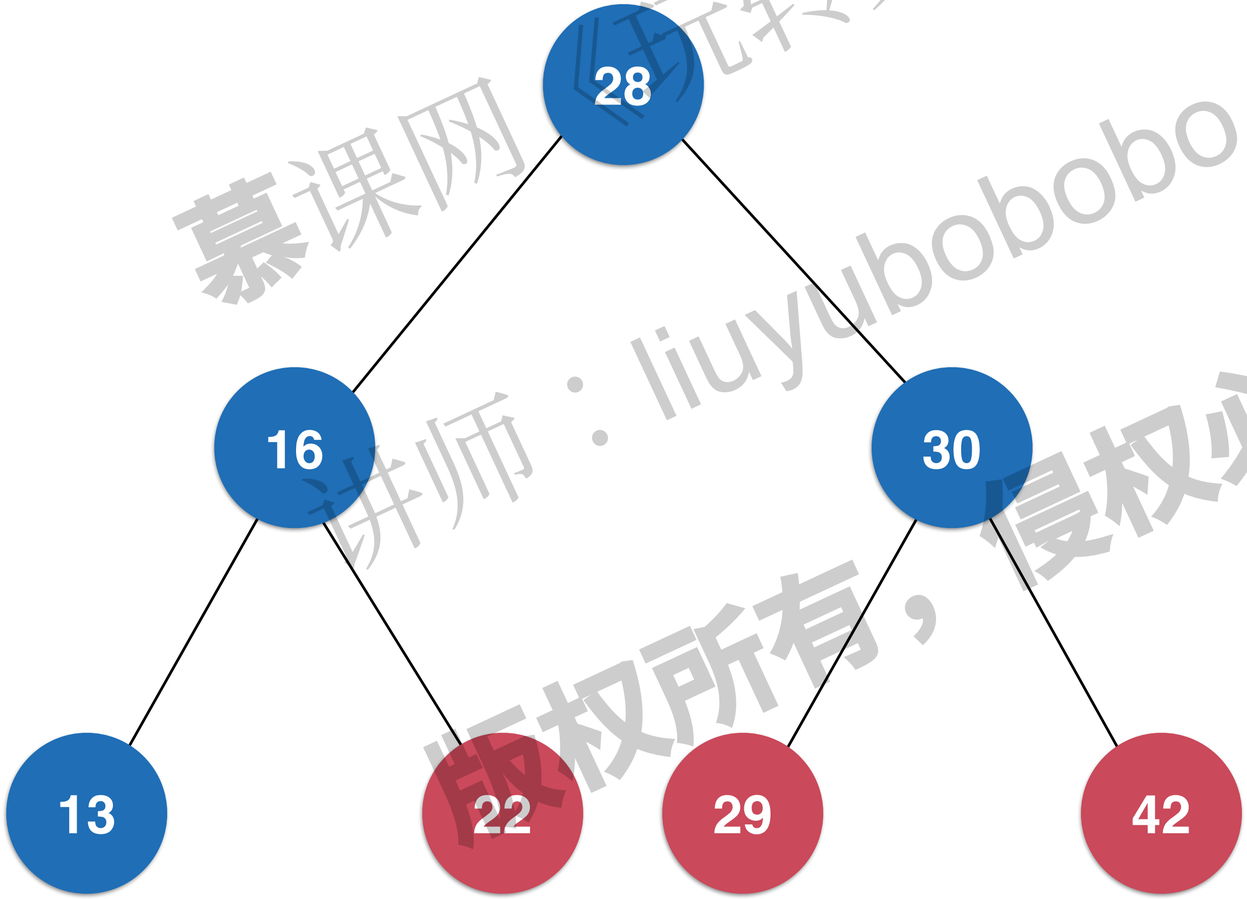
front

tail

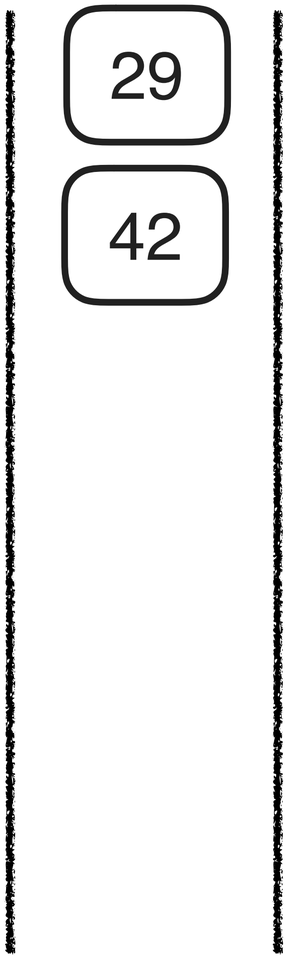




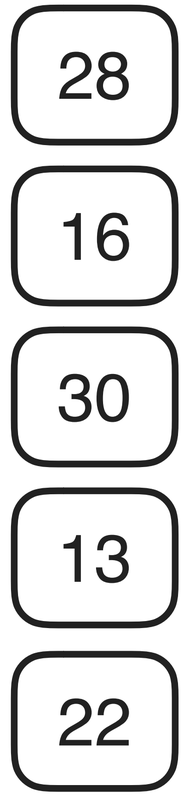
二分搜索树的层序遍历



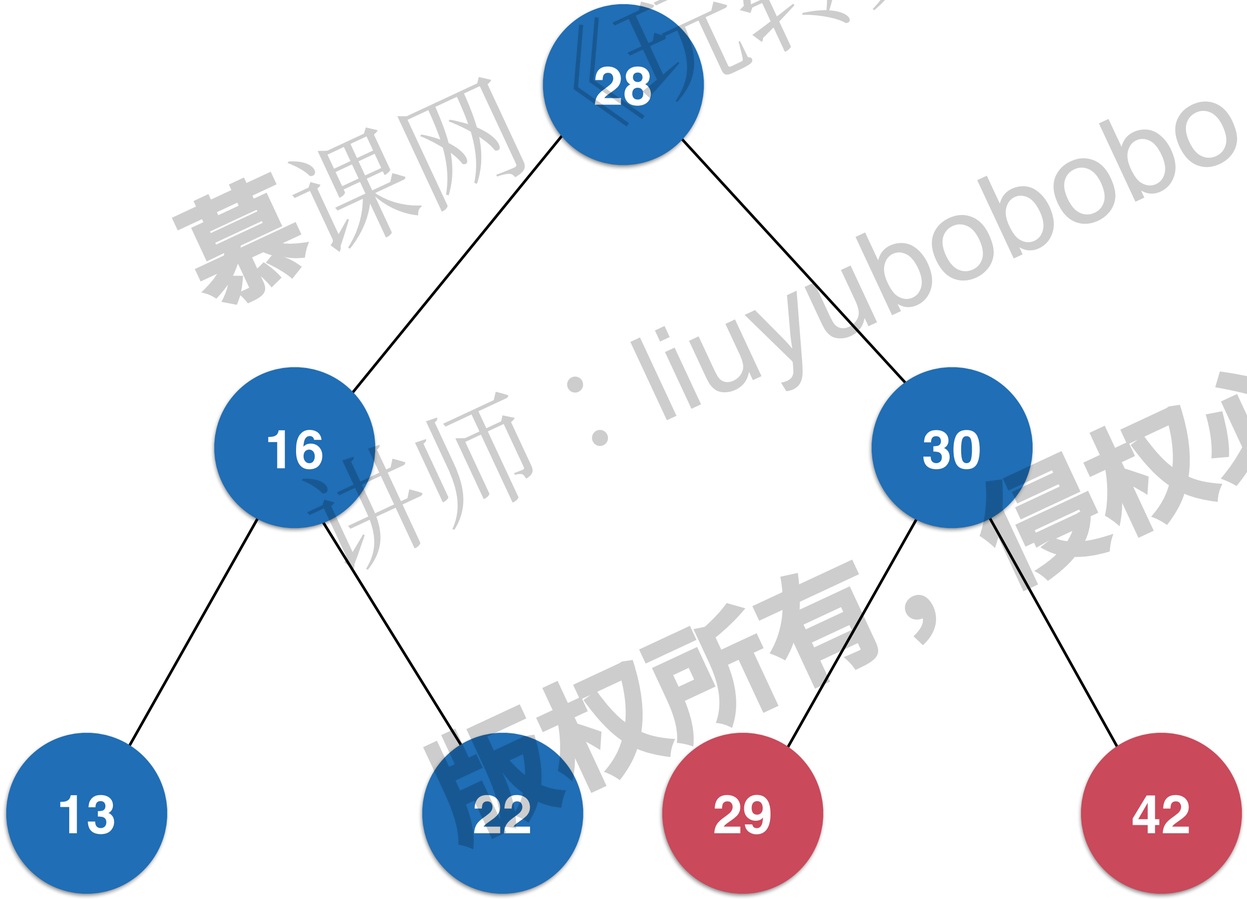
front

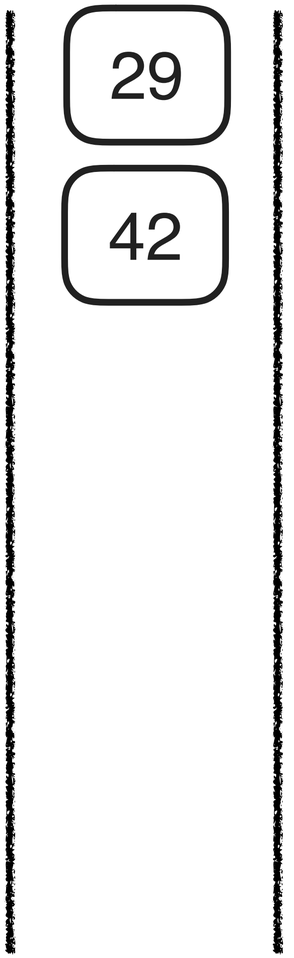


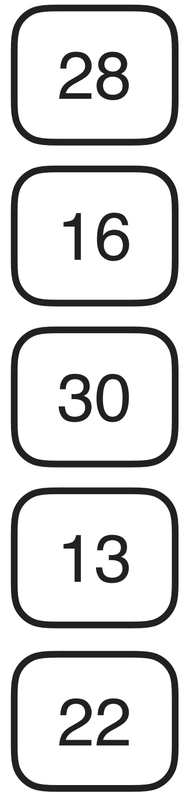
tail



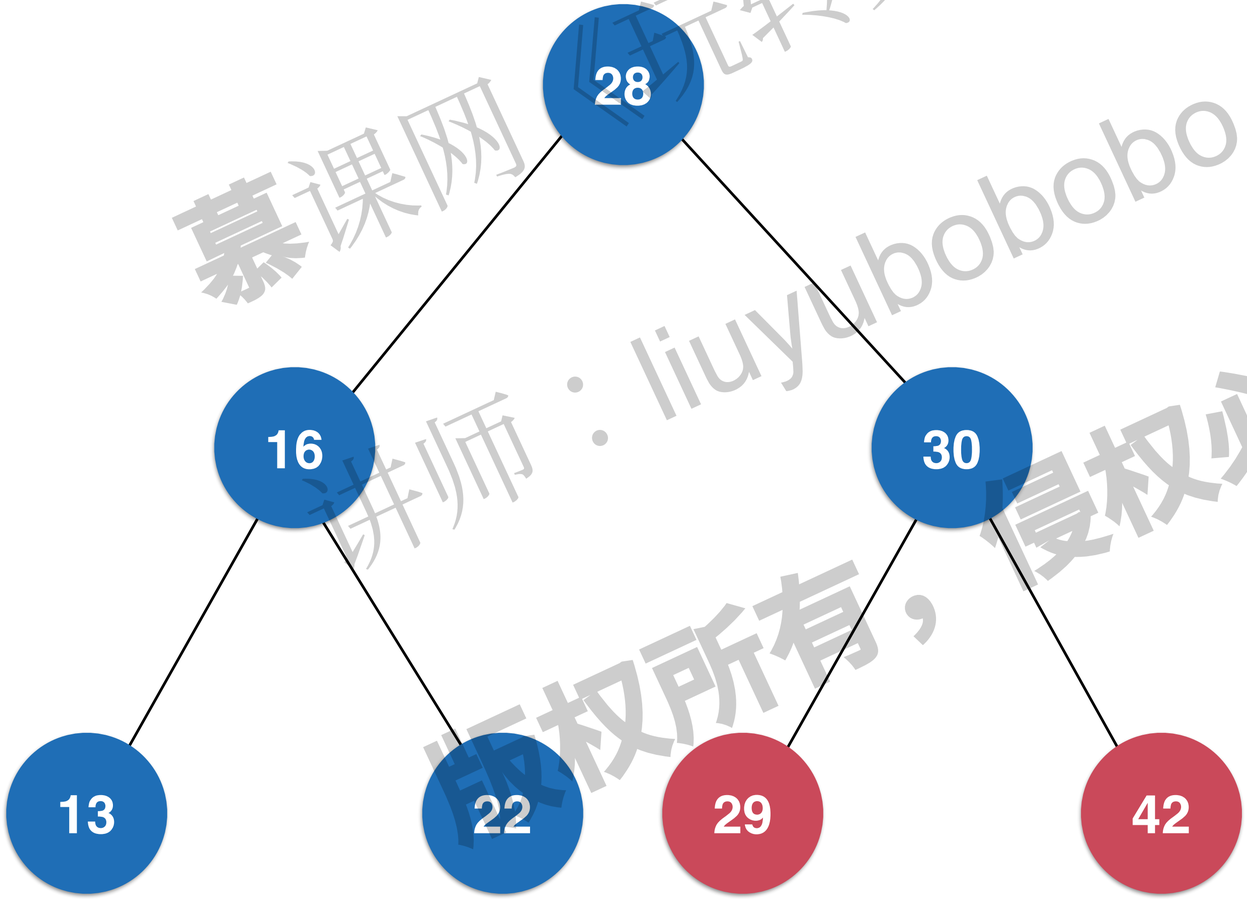
二分搜索树的层序遍历

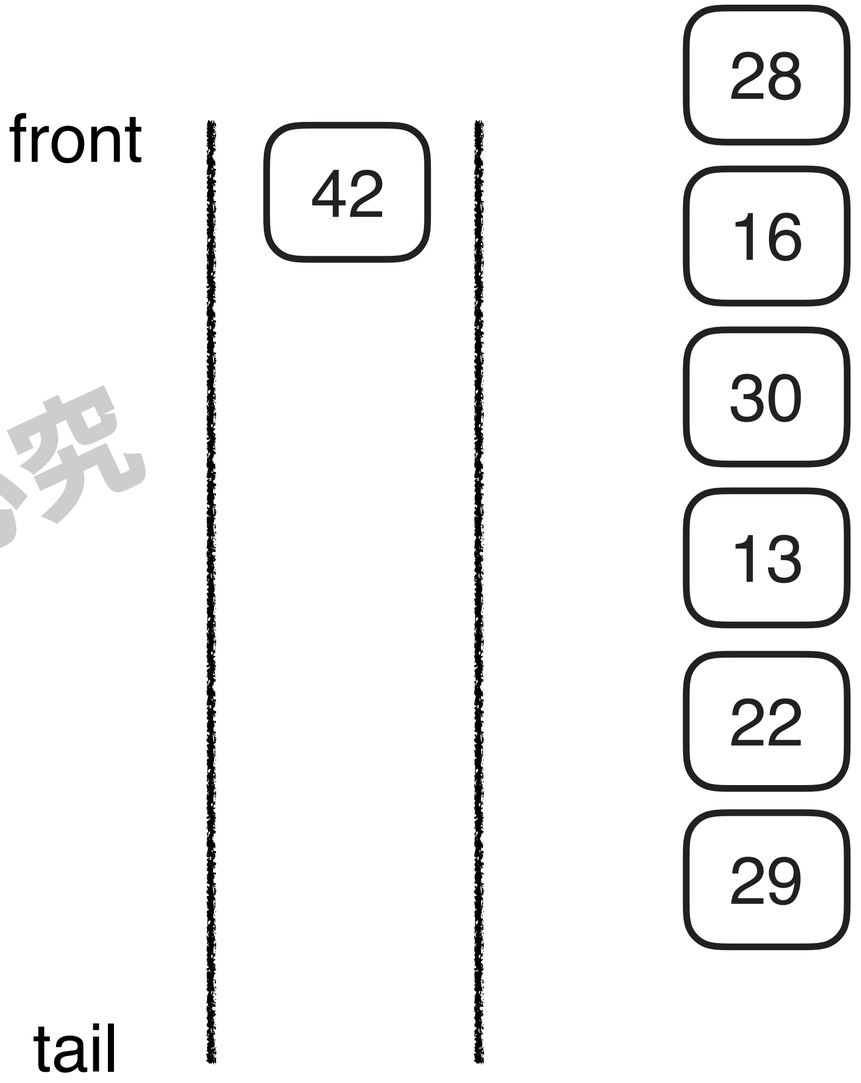




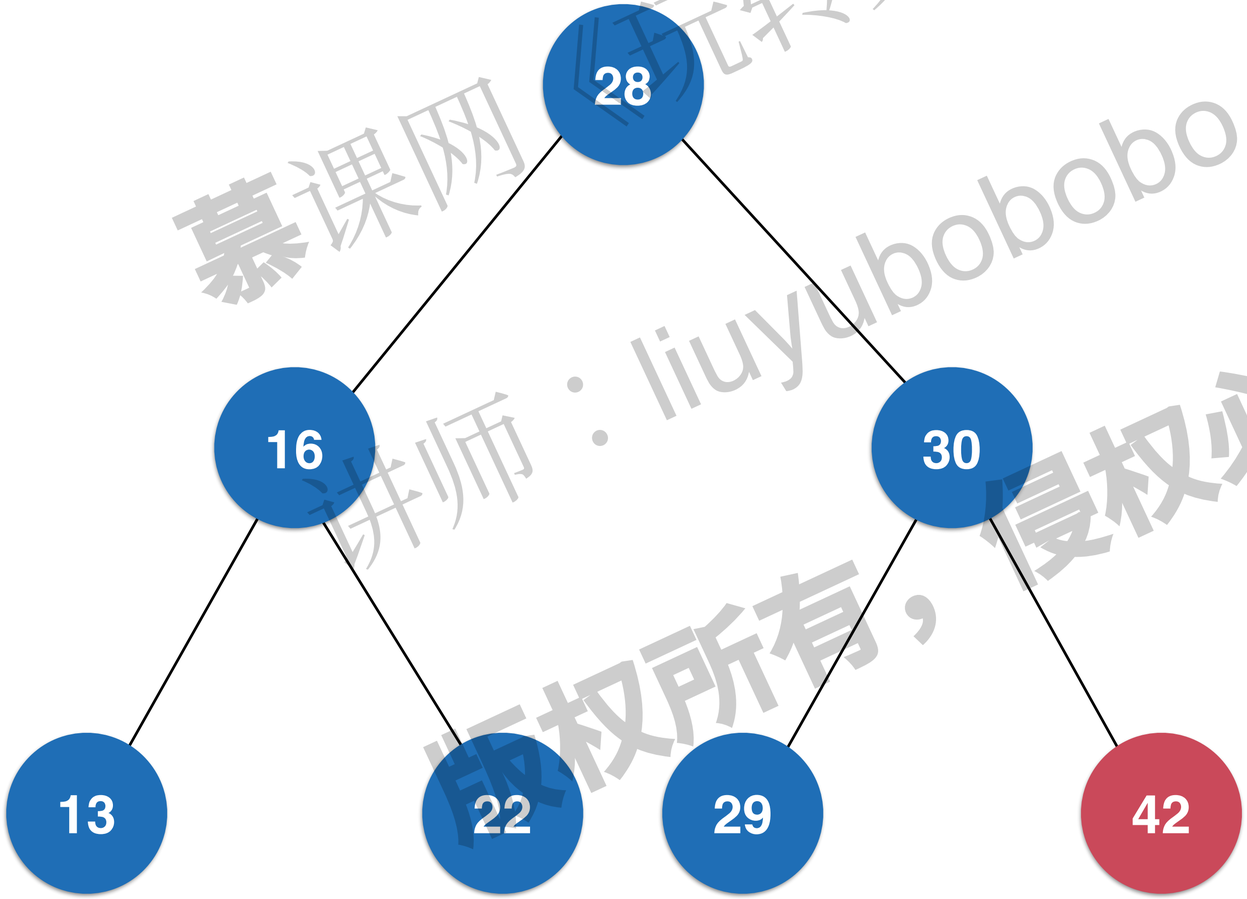


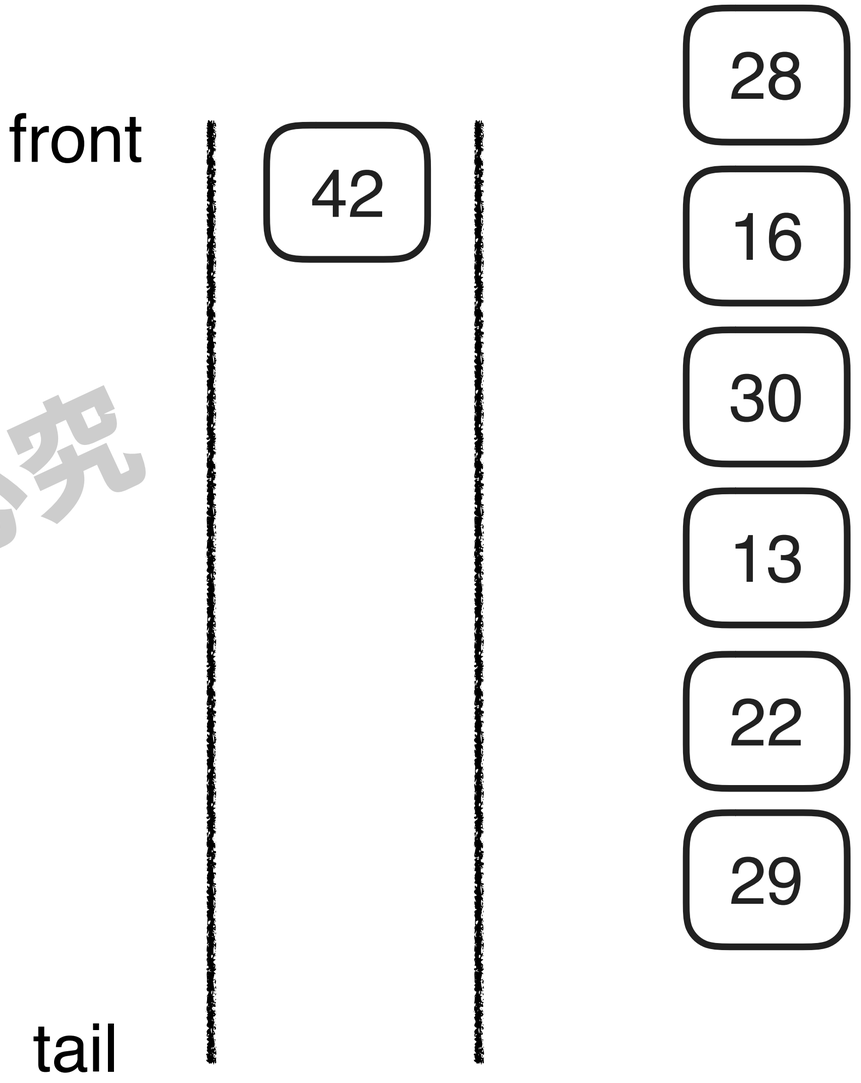
二分搜索树的层序遍历



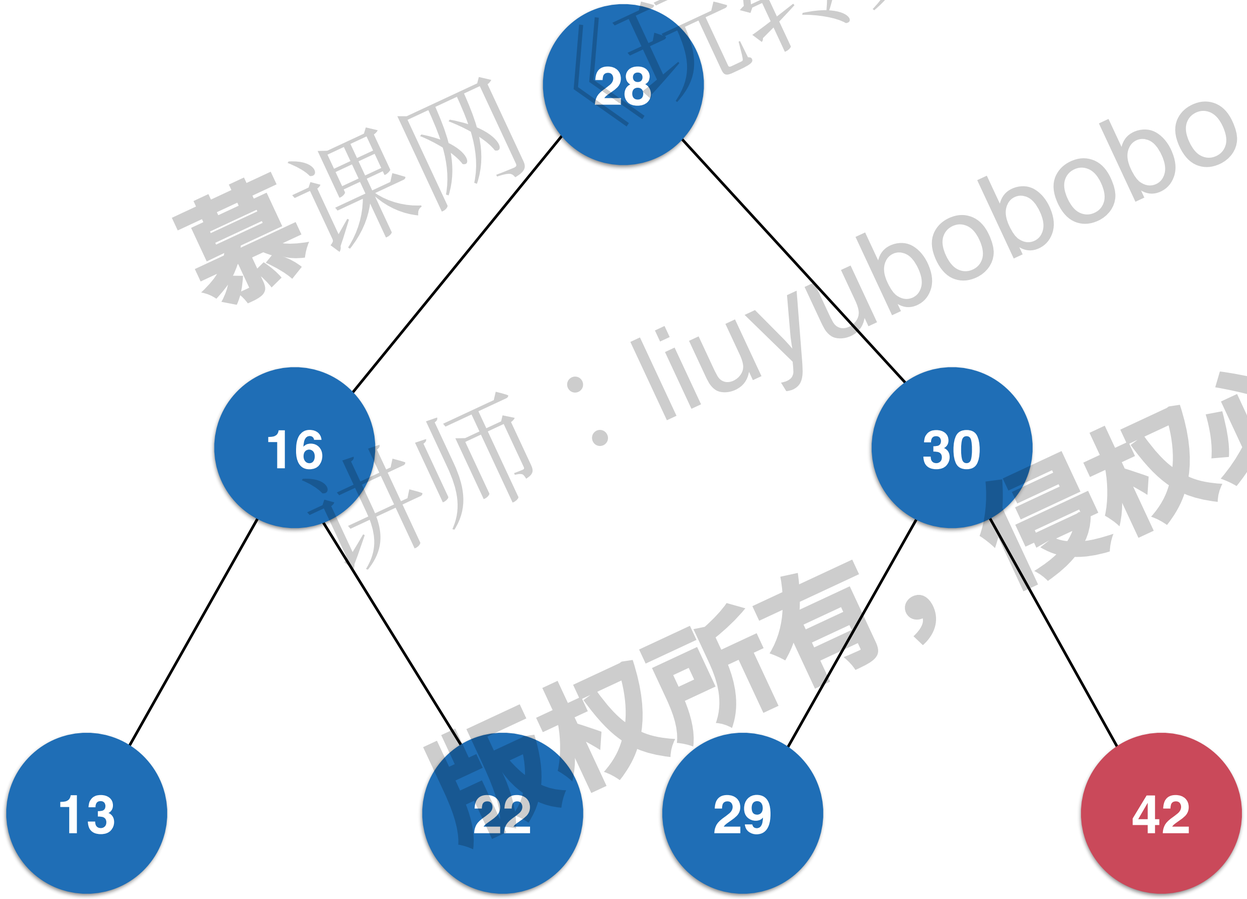


二分搜索树的层序遍历



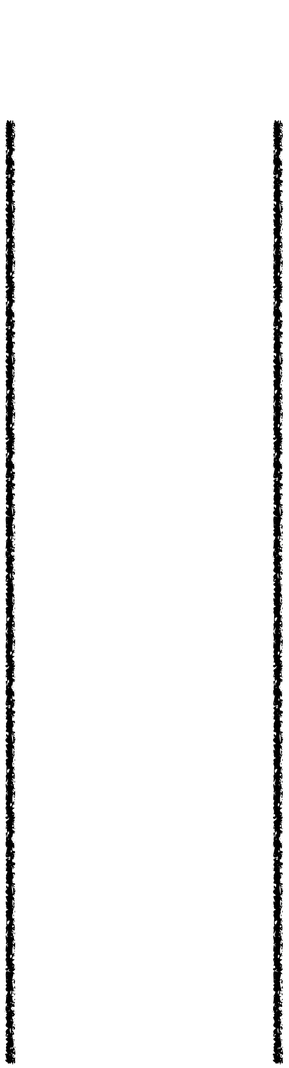


二分搜索树的层序遍历

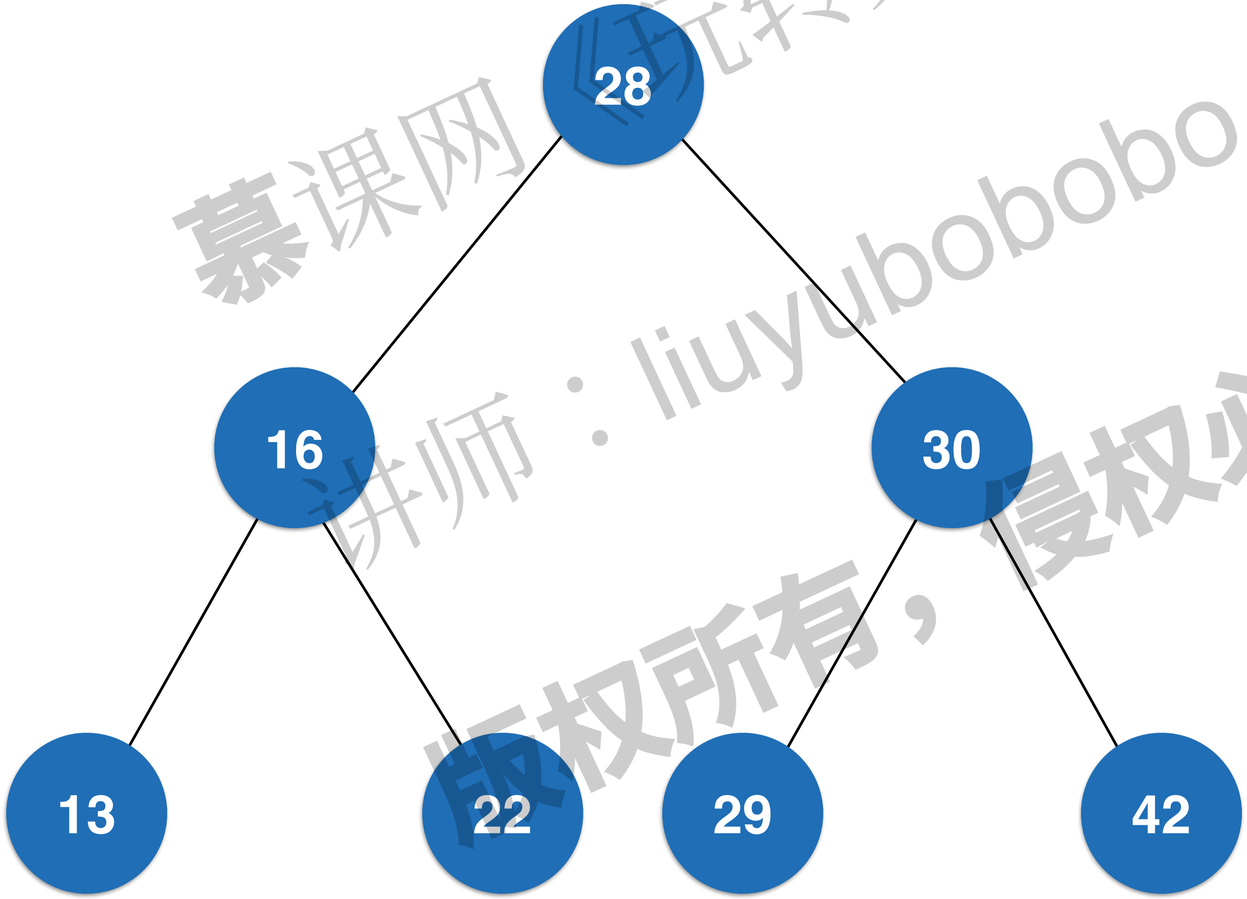


front

tail

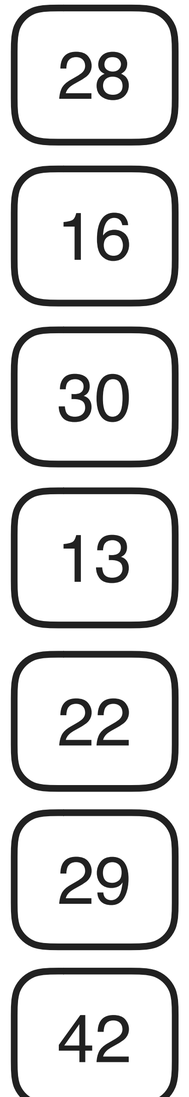


二分搜索树的层序遍历



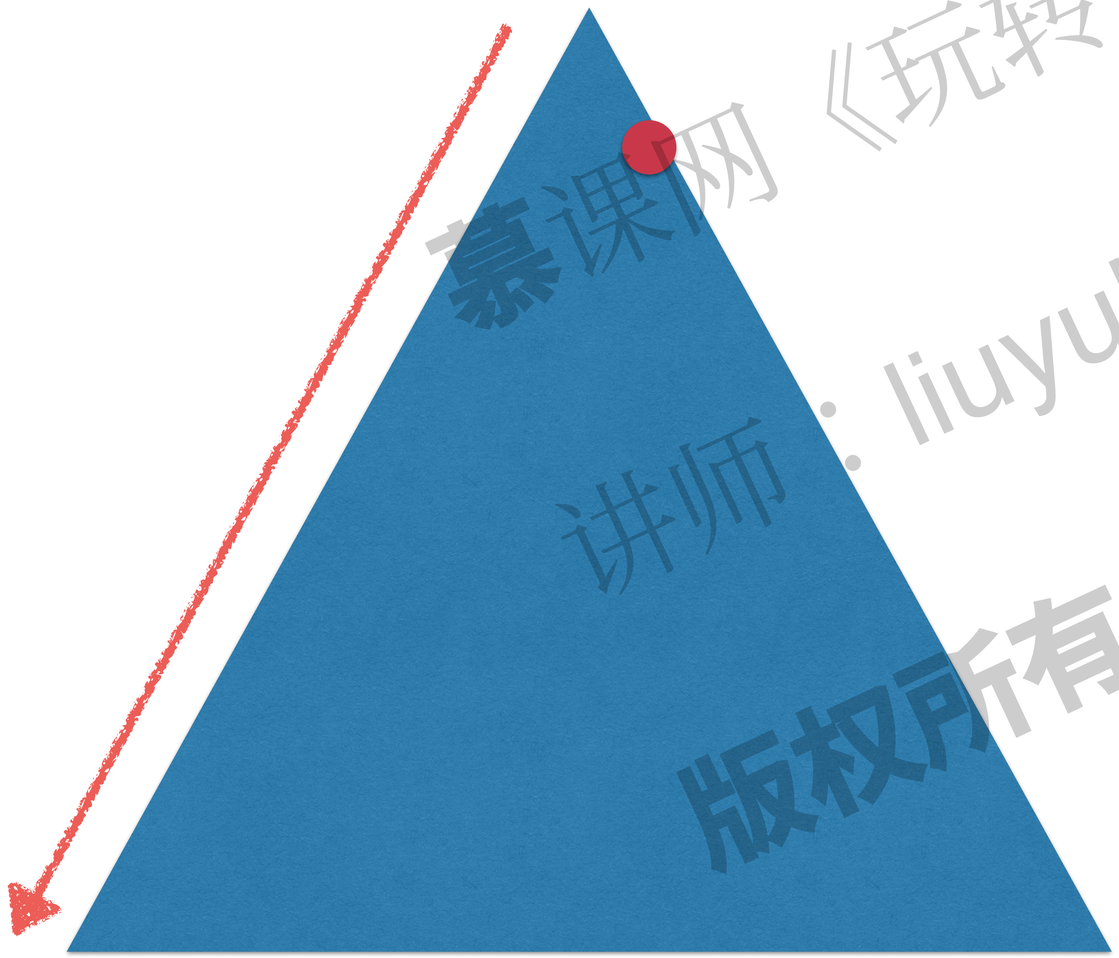
front

tail



实践：二分搜索树的层序遍历

广度优先遍历的意义

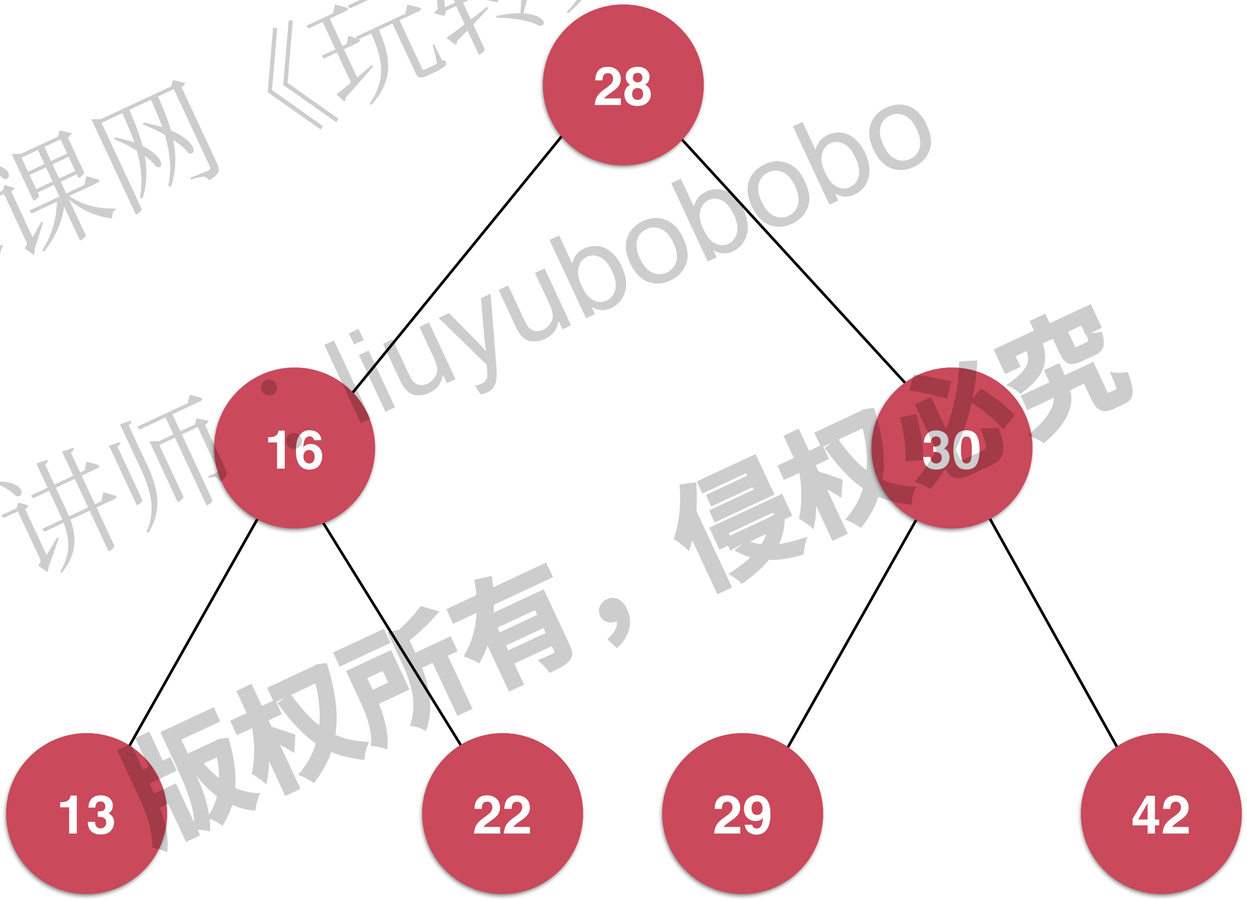


- 更快的找到问题的解
- 常用于算法设计中 - 最短路径
- 图中的深度优先遍历和广度优先遍历

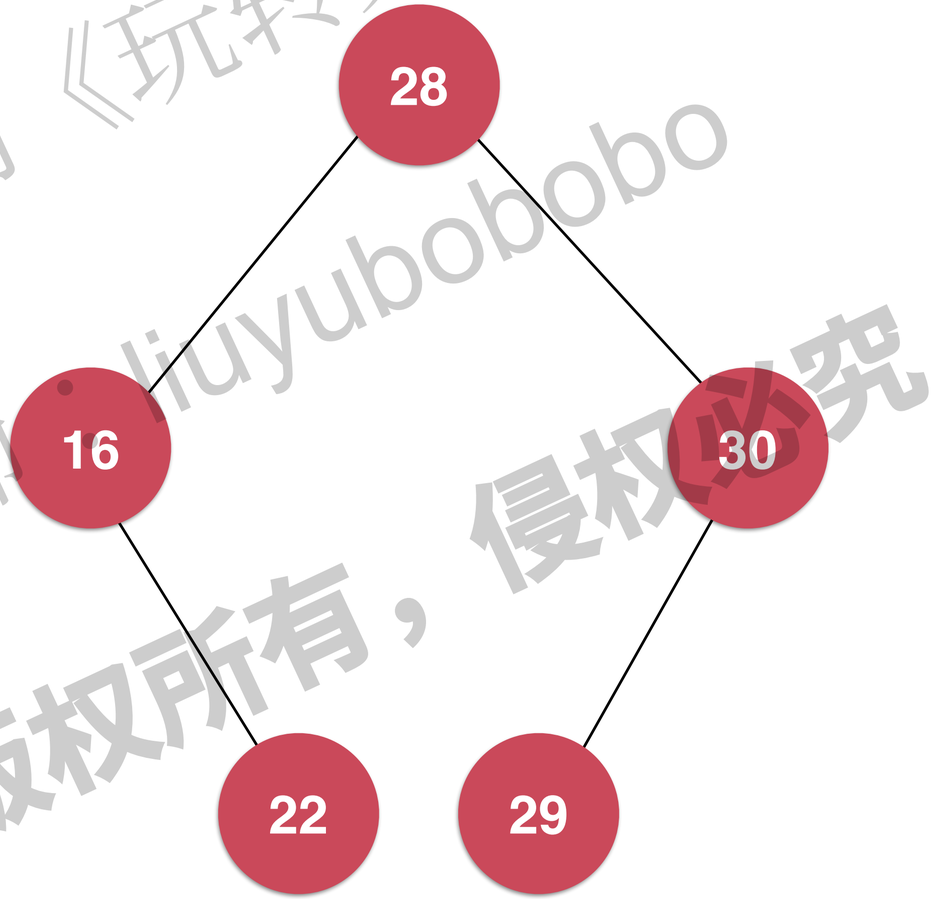
二分搜索树 删除节点

从最简单的，删除二分搜索树的最小值和最大值开始

二分搜索树的最小值和最大值

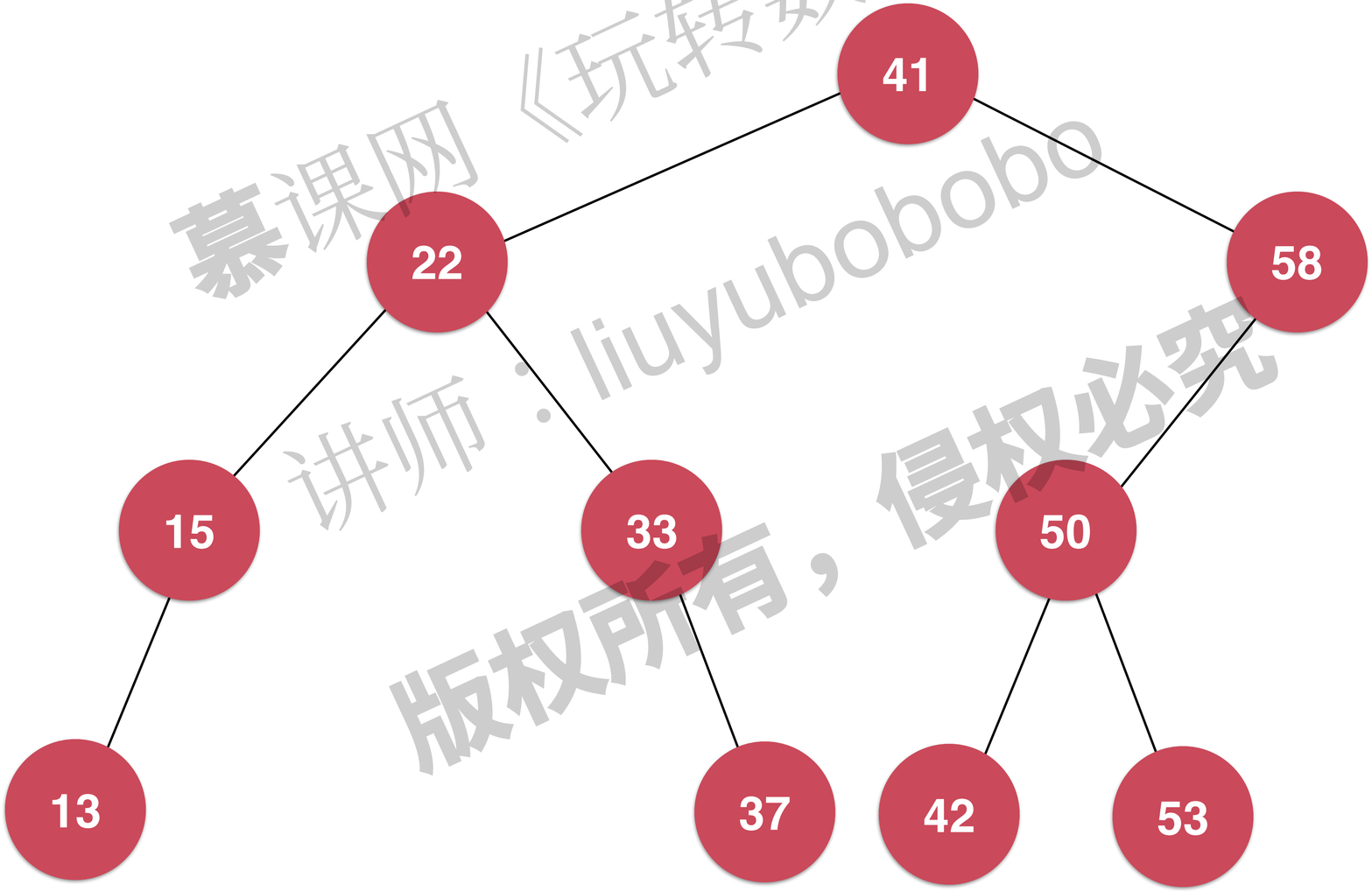


二分搜索树的最小值和最大值

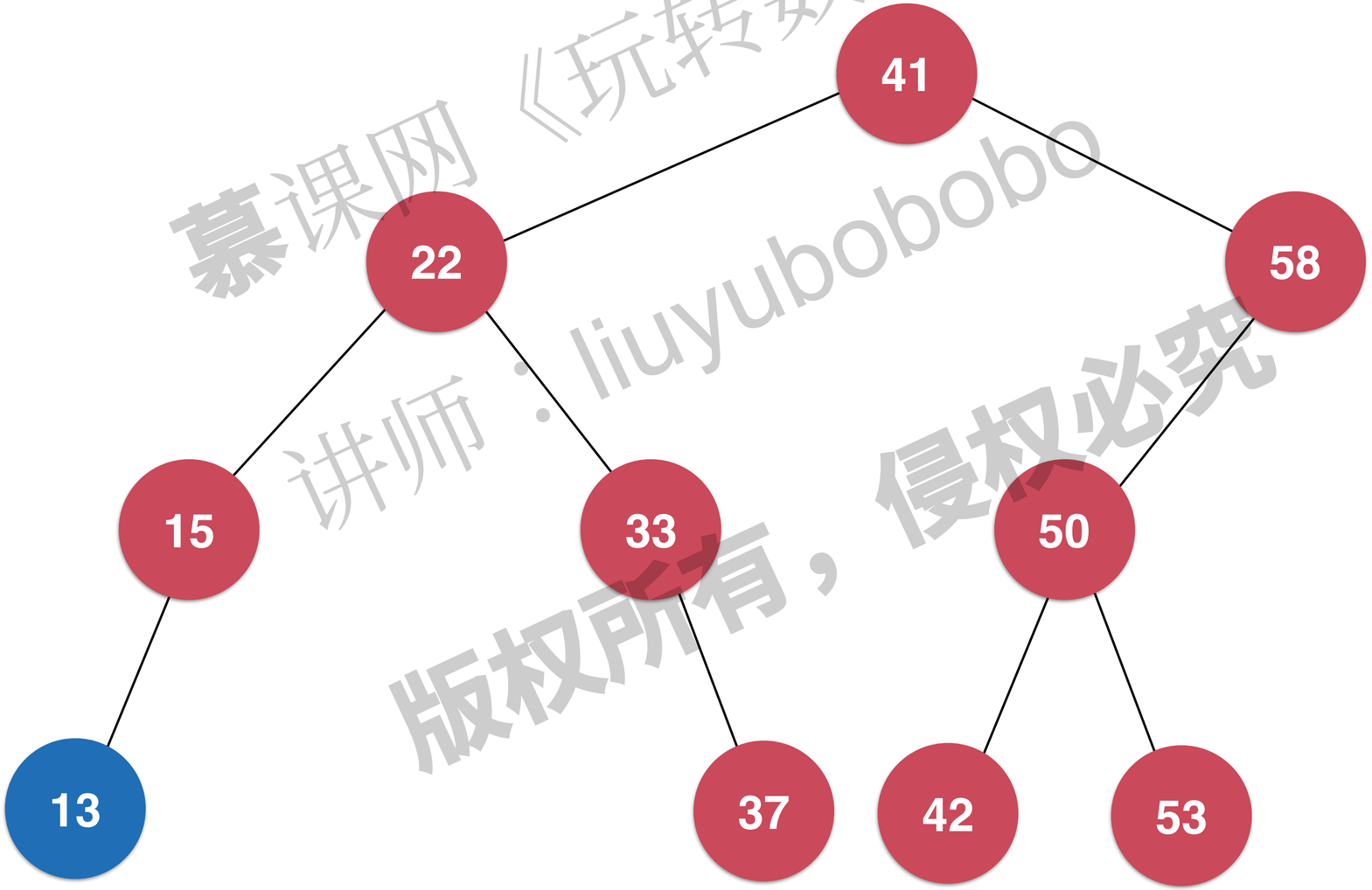


实践：求二分搜索树的最小值和最大值

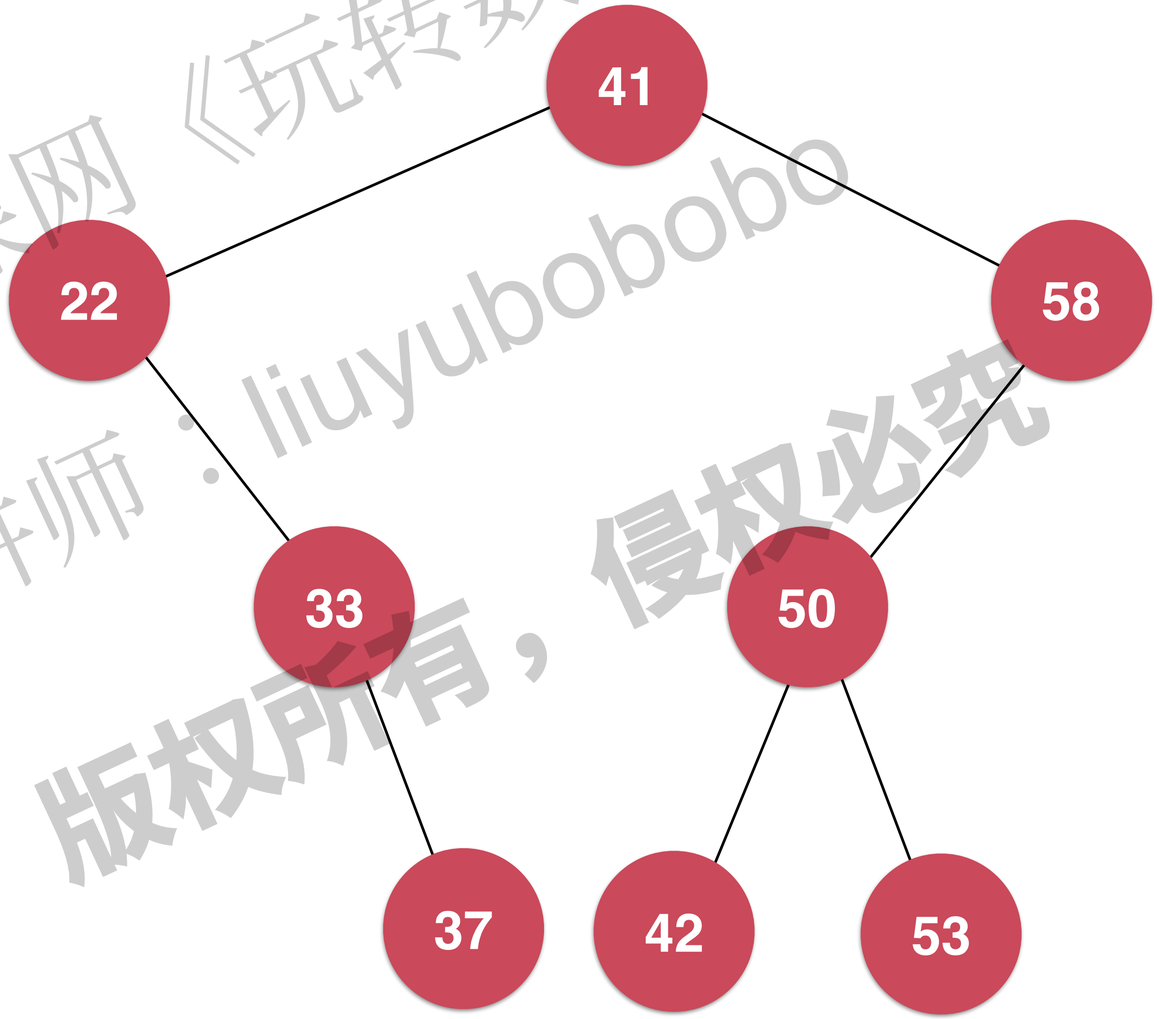
删除二分搜索树的最小值



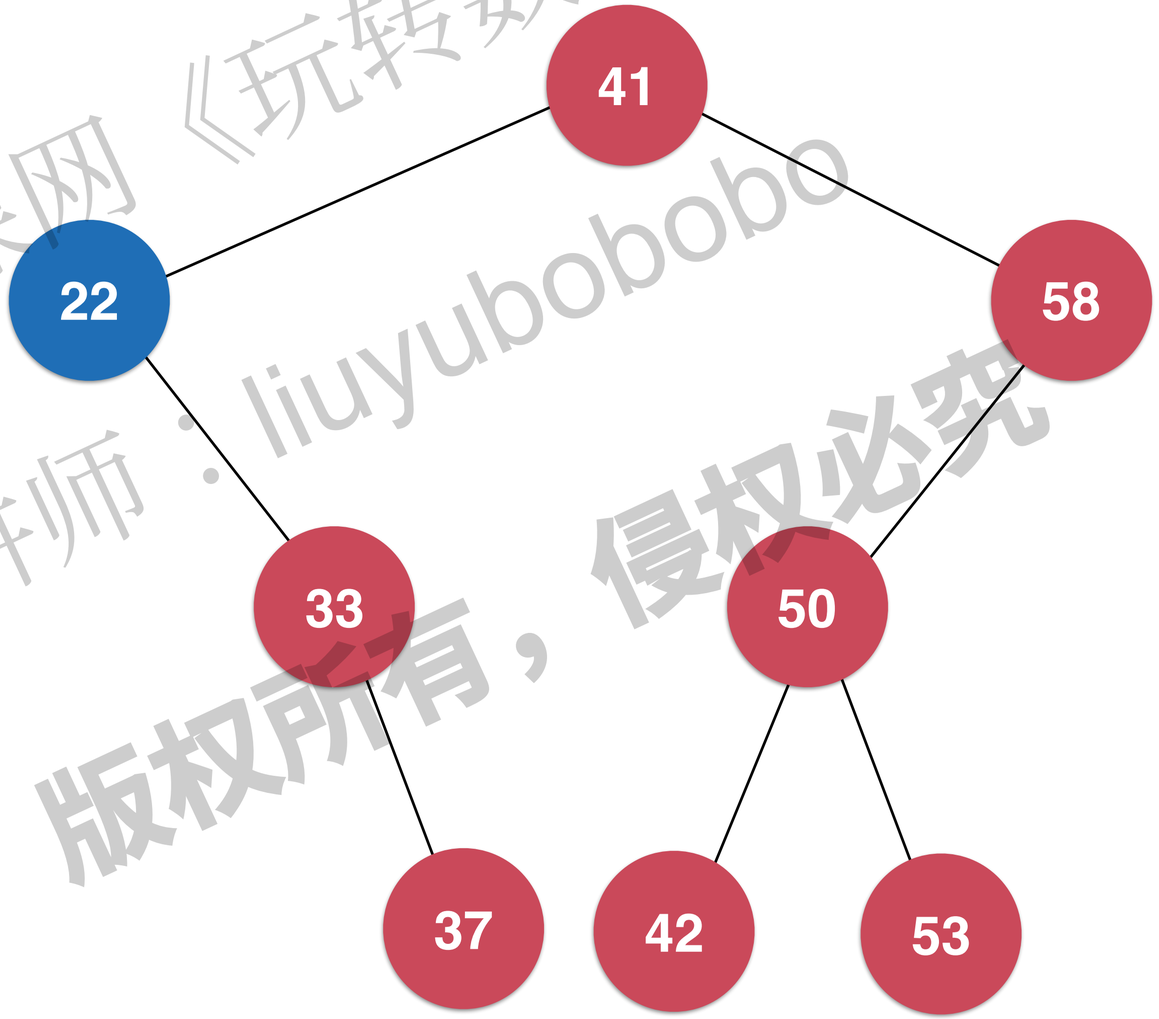
删除二分搜索树的最小值



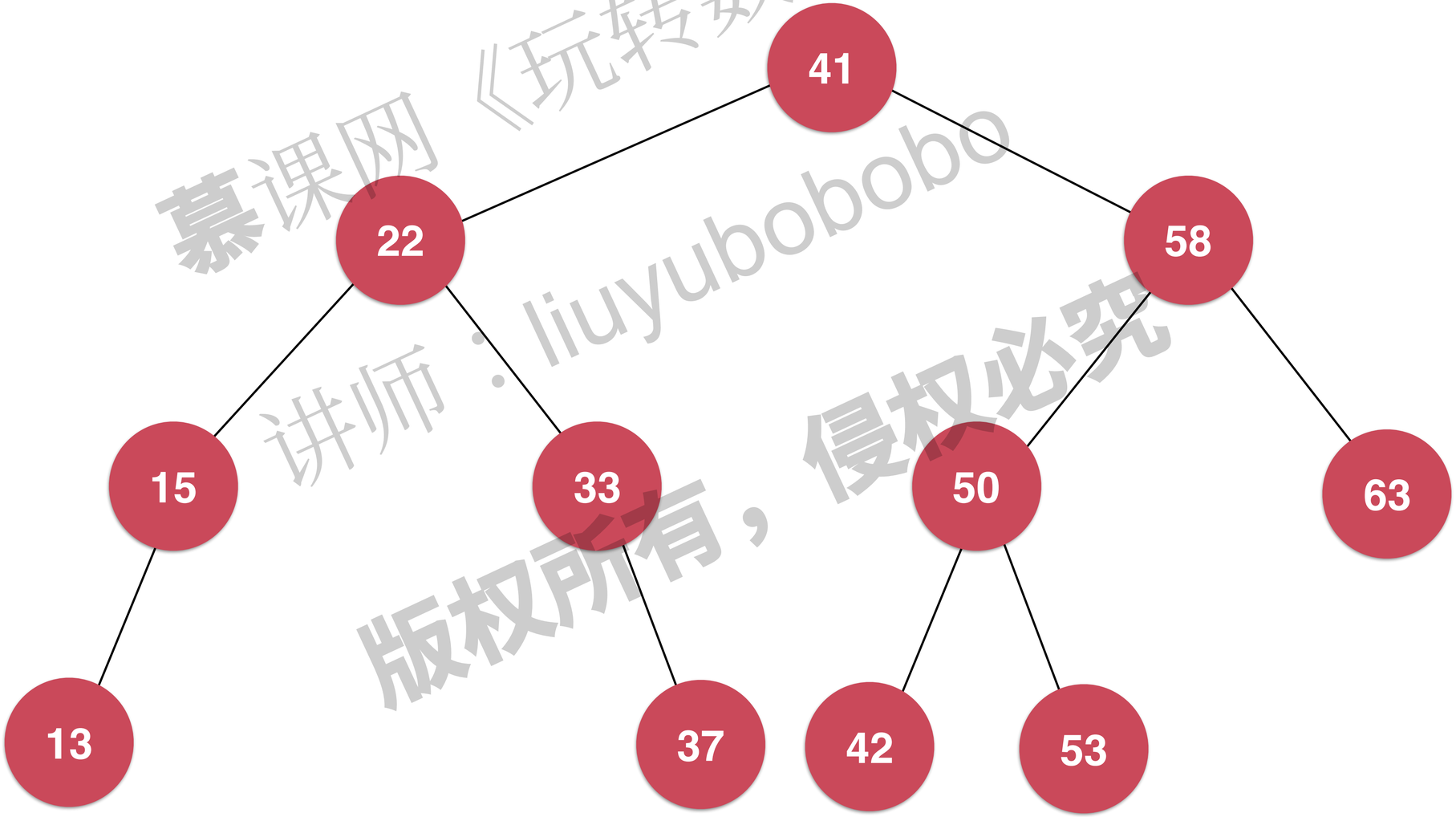
删除二分搜索树的最小值



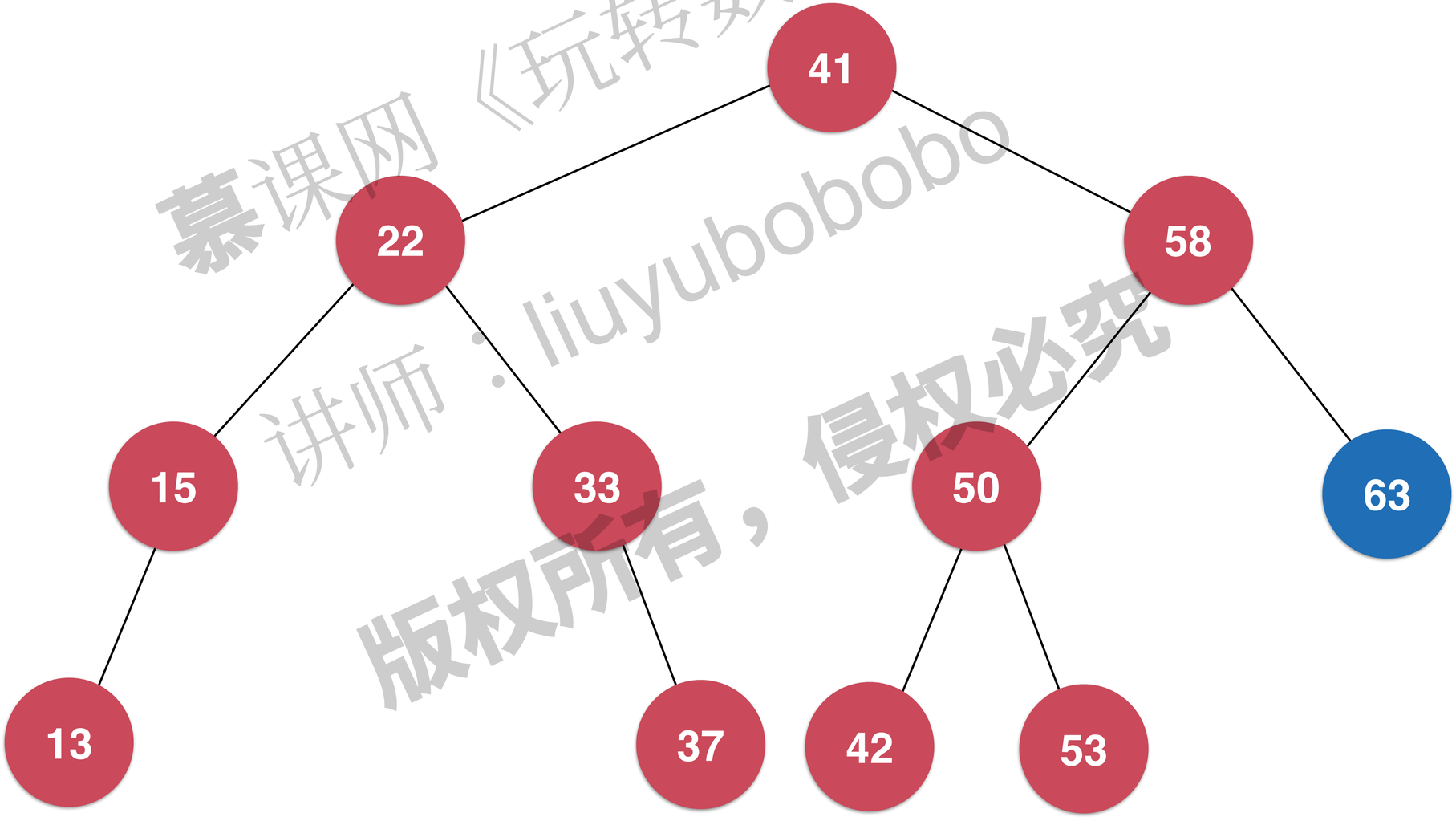
删除二分搜索树的最小值



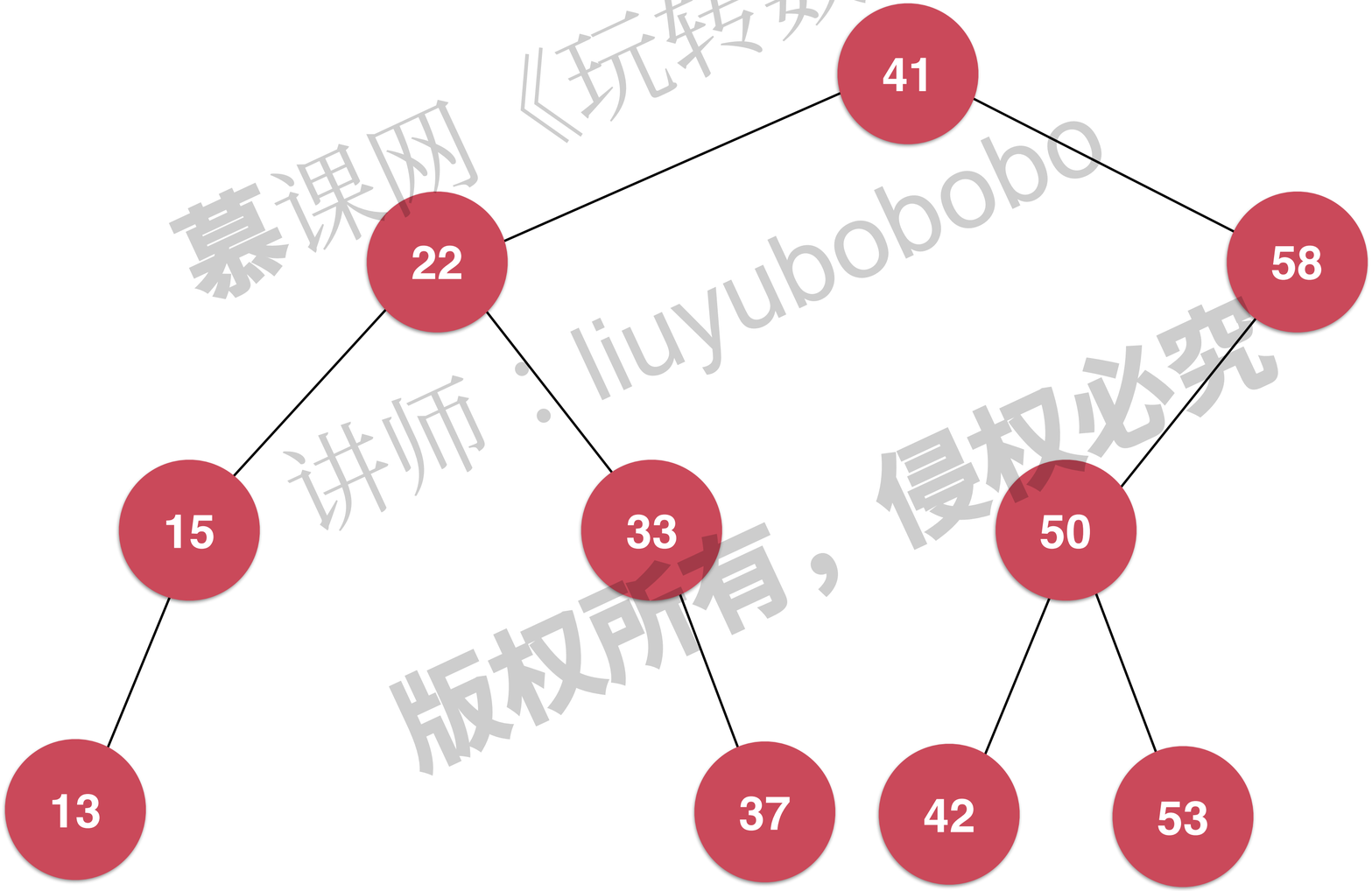
删除二分搜索树的最大值



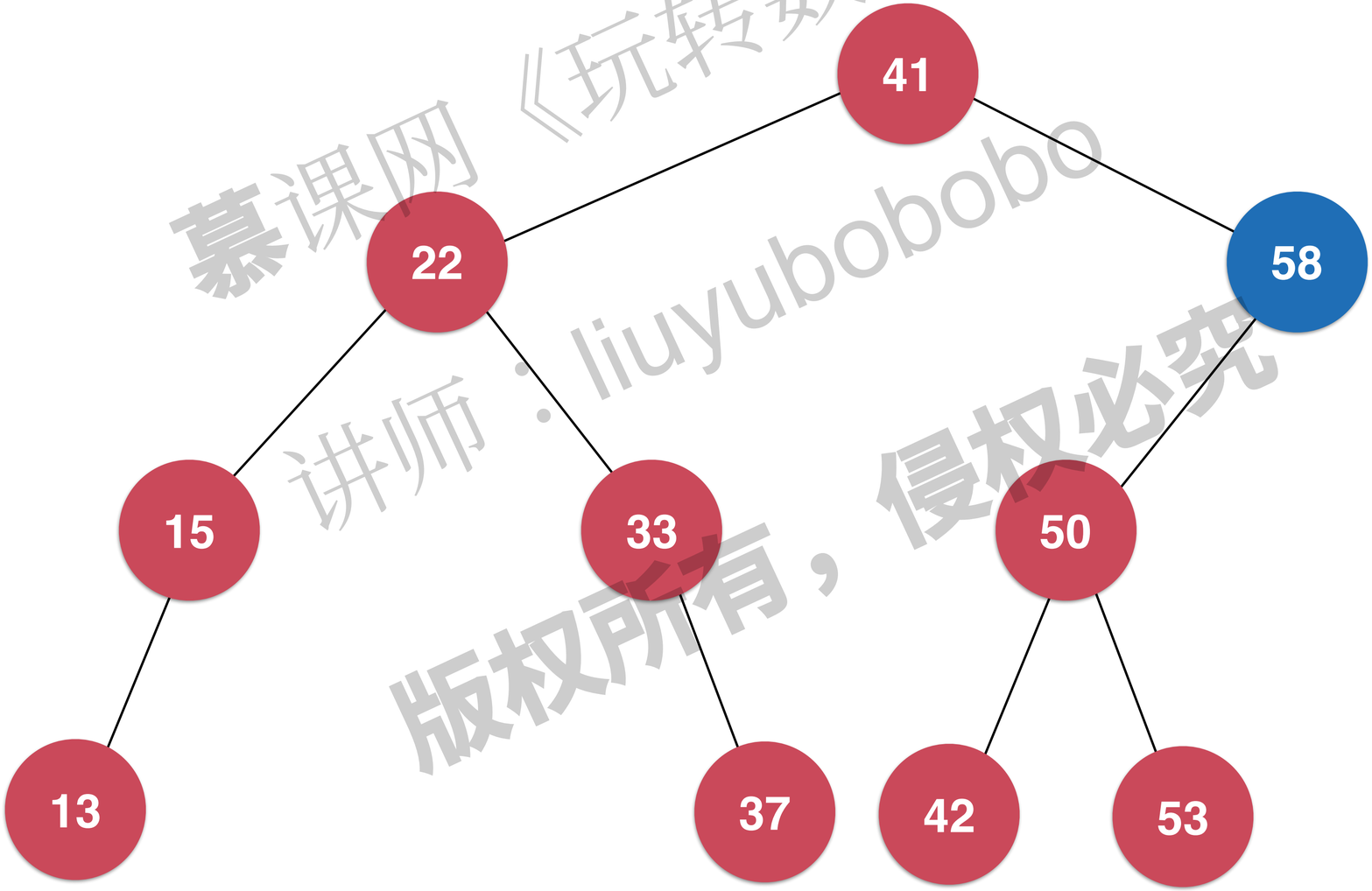
删除二分搜索树的最大值



删除二分搜索树的最大值



删除二分搜索树的最大值

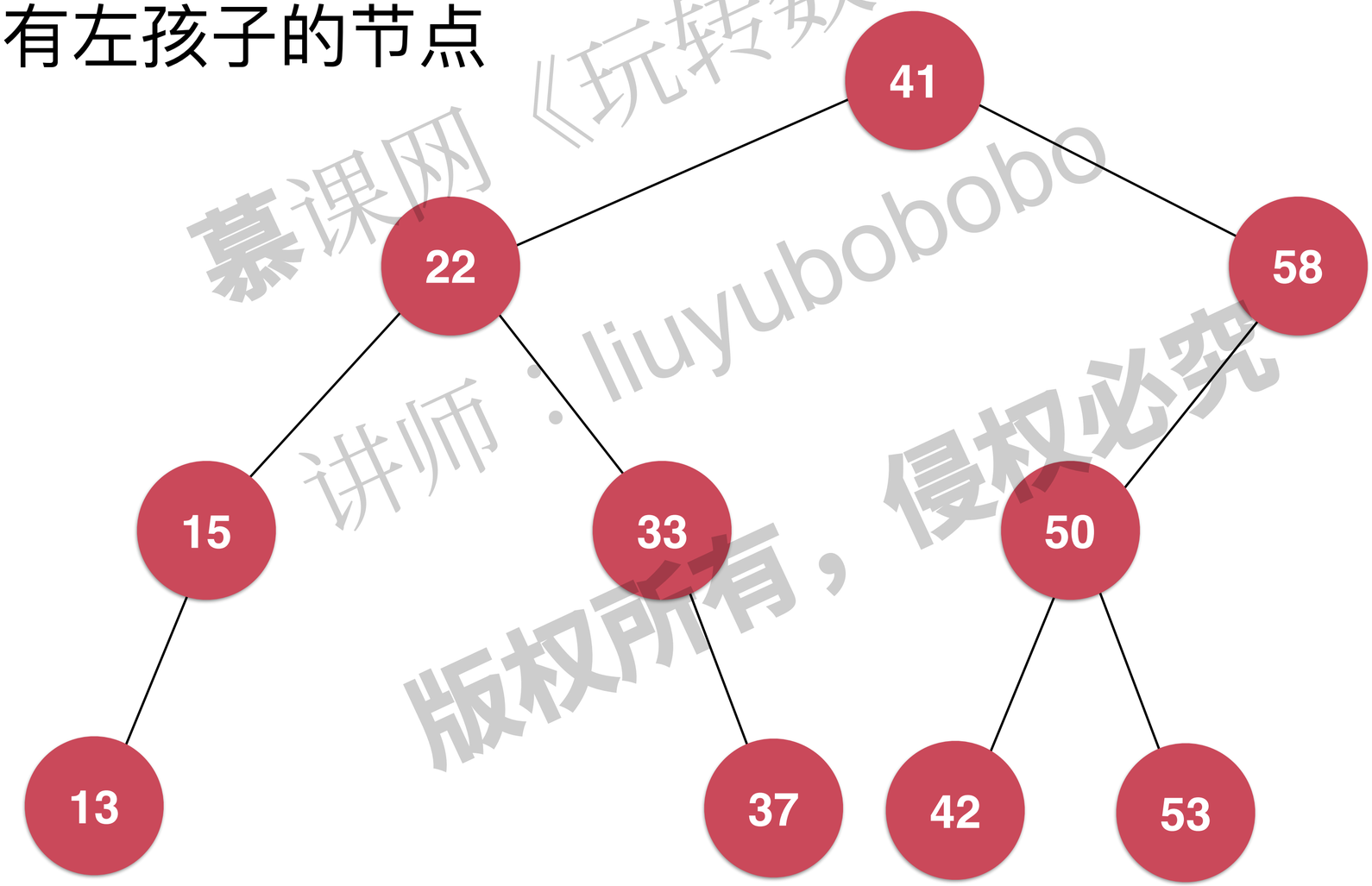


实践：删除二分搜索树的最小值和最大值

二分搜索树删除节点

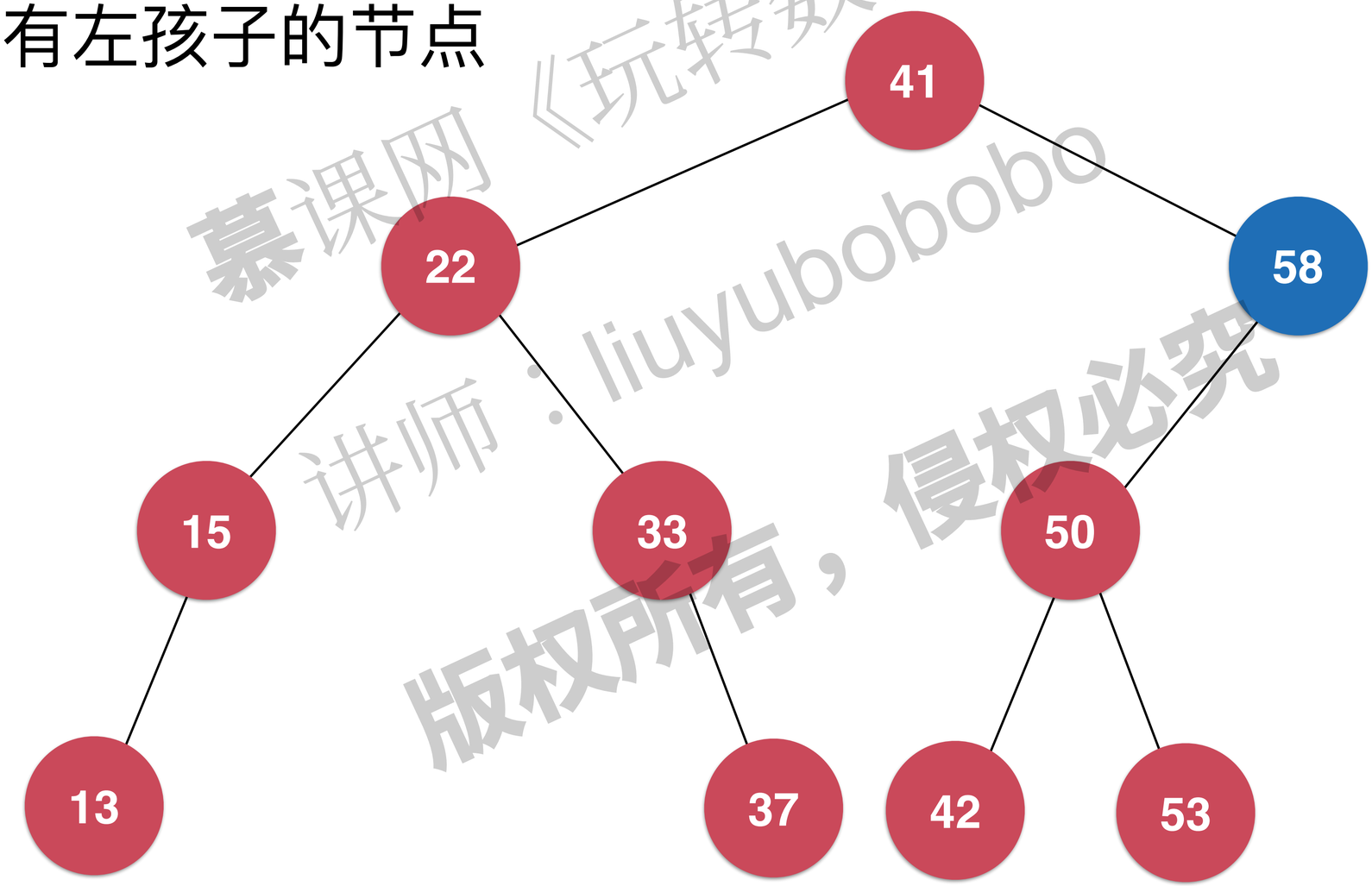
二分搜索树删除节点

删除只有左孩子的节点



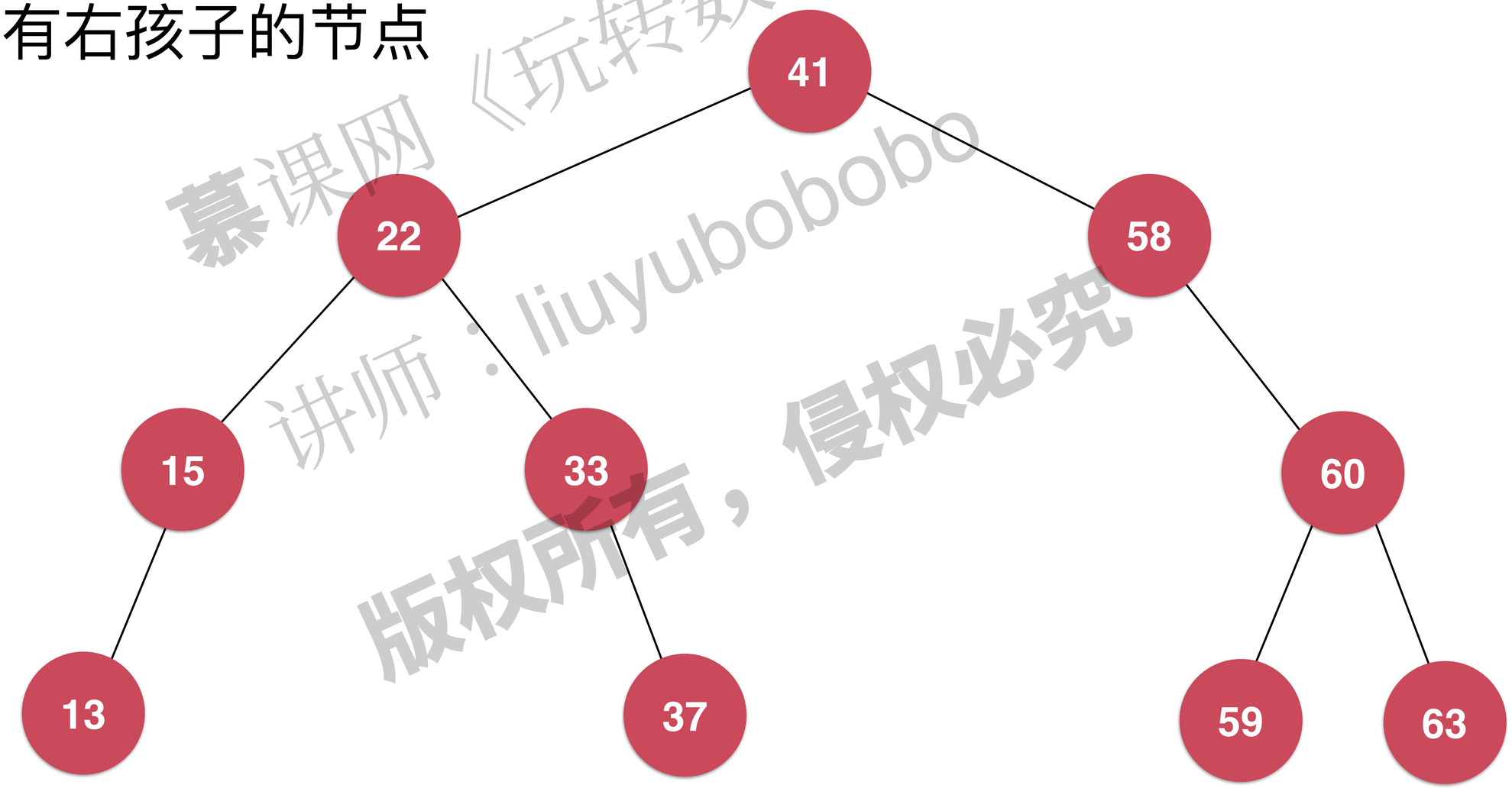
二分搜索树删除节点

删除只有左孩子的节点



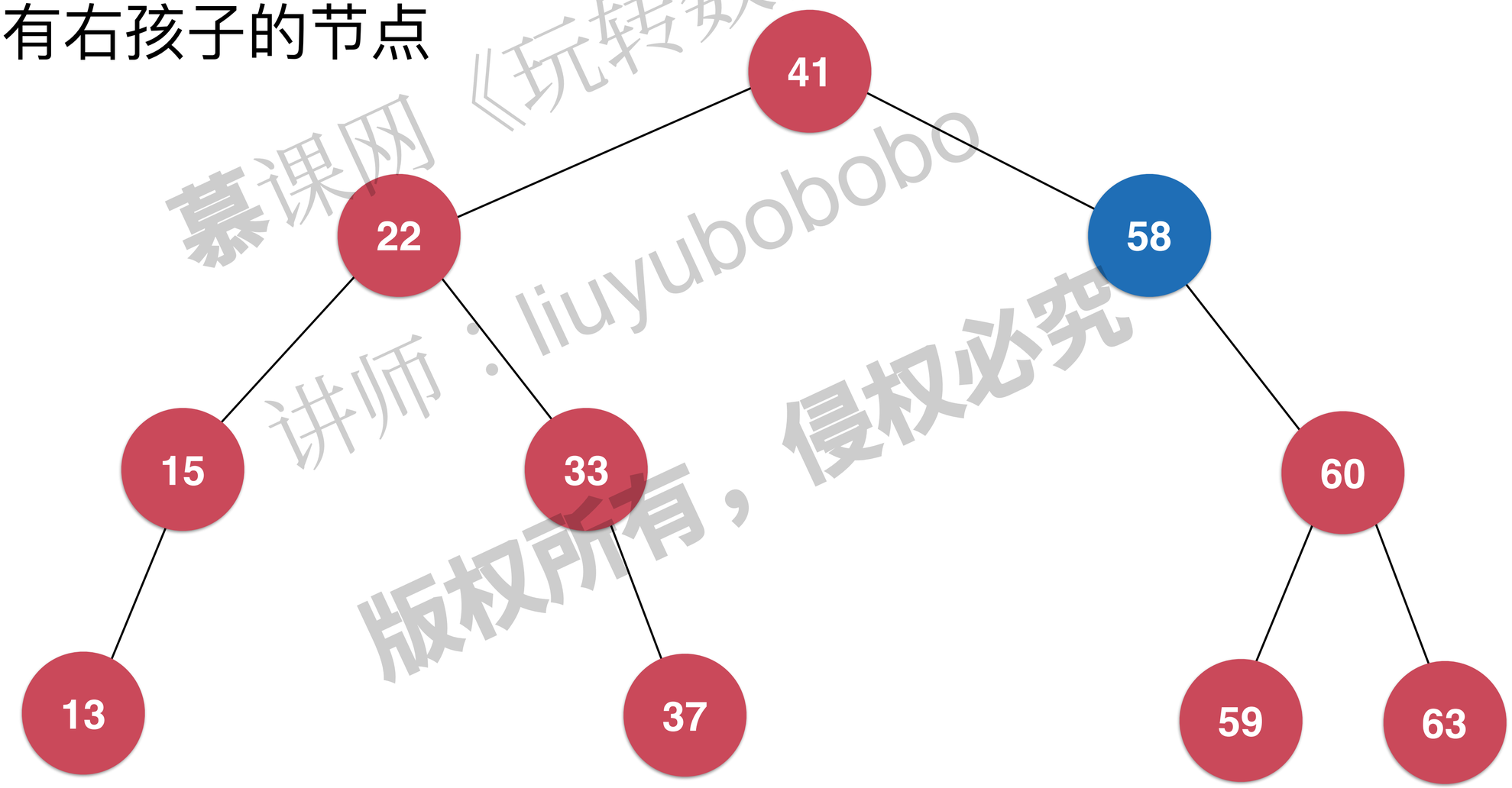
二分搜索树删除节点

删除只有右孩子的节点



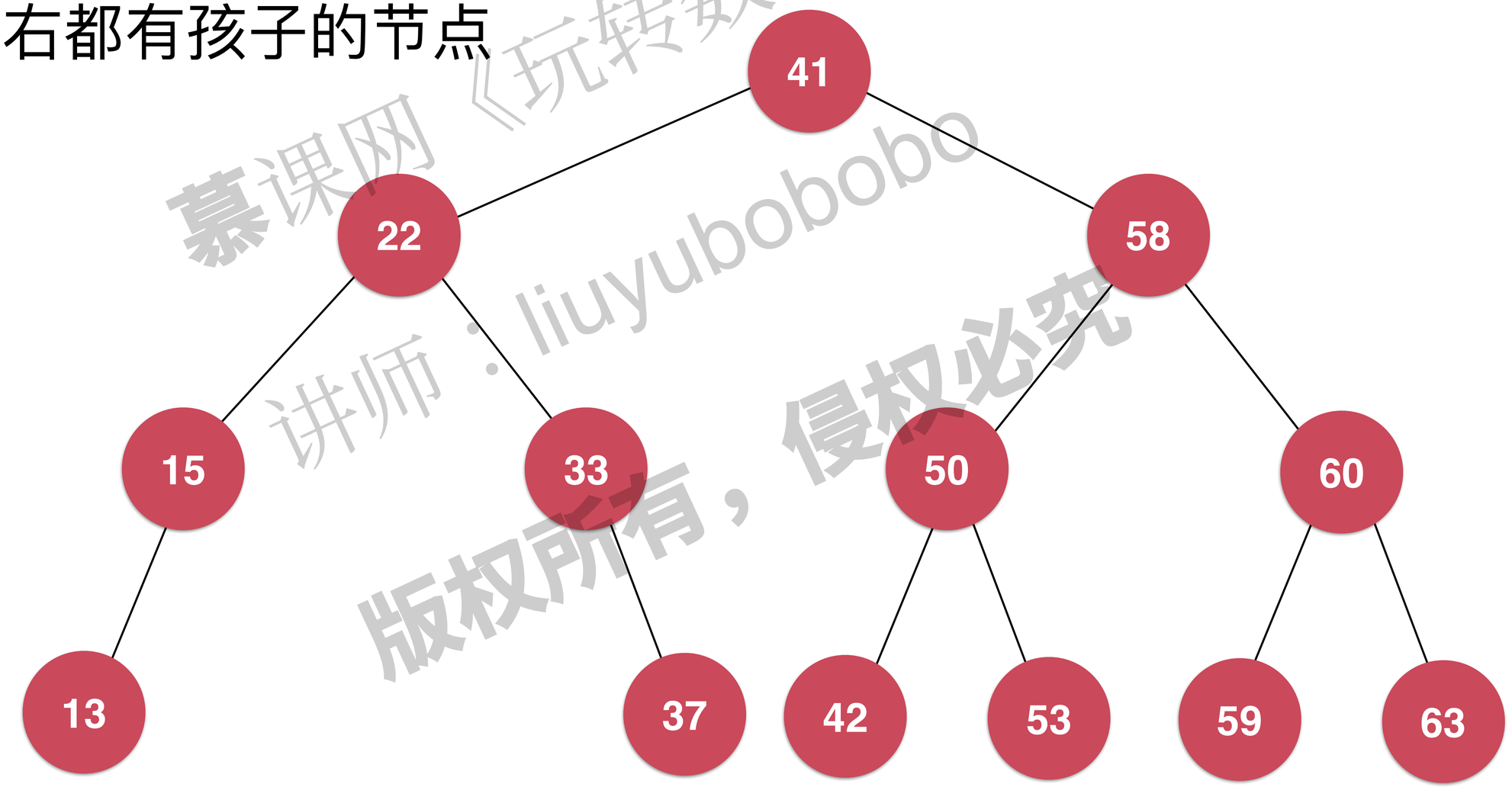
二分搜索树删除节点

删除只有右孩子的节点



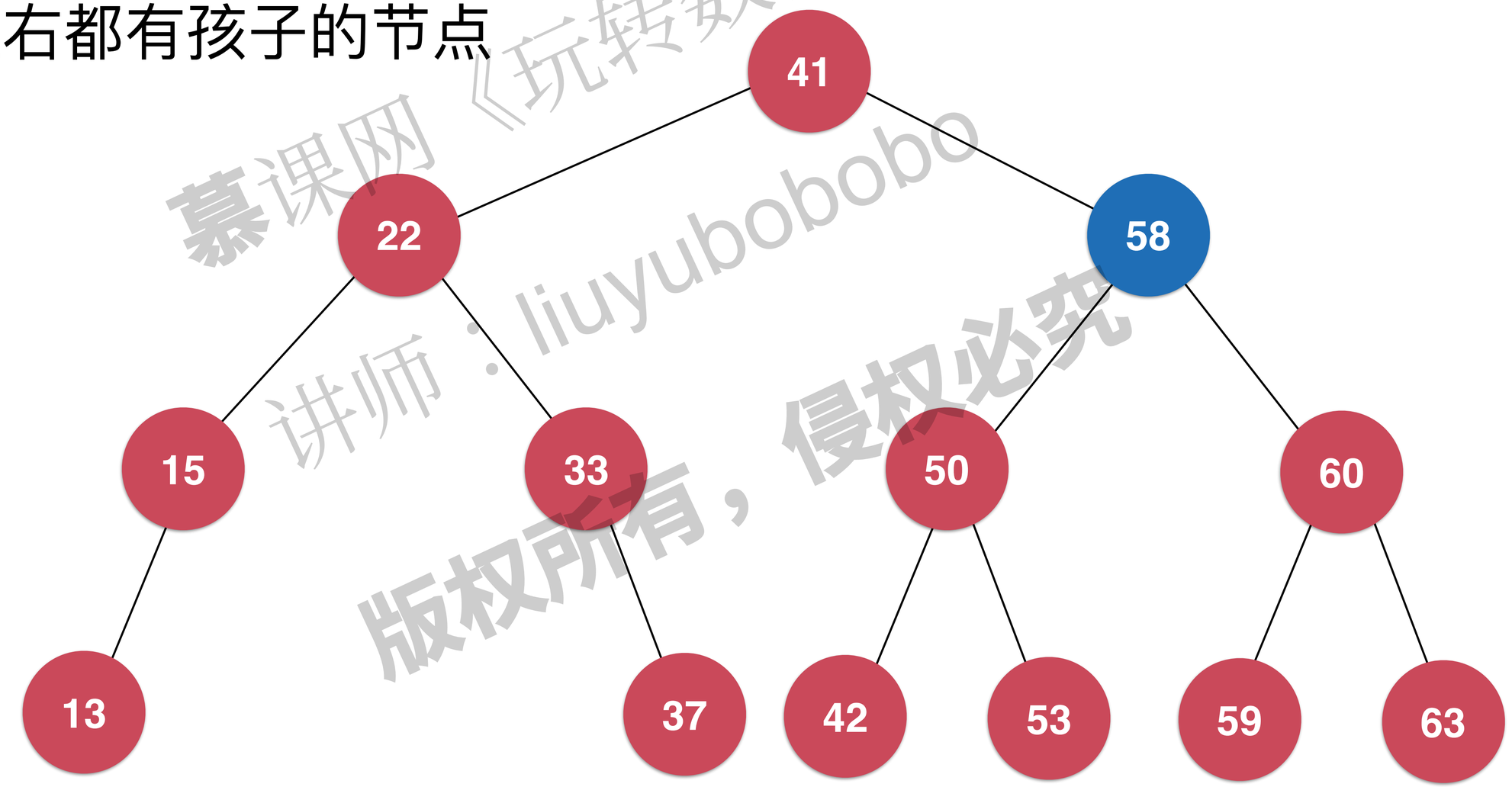
二分搜索树删除节点

删除左右都有孩子的节点



二分搜索树删除节点

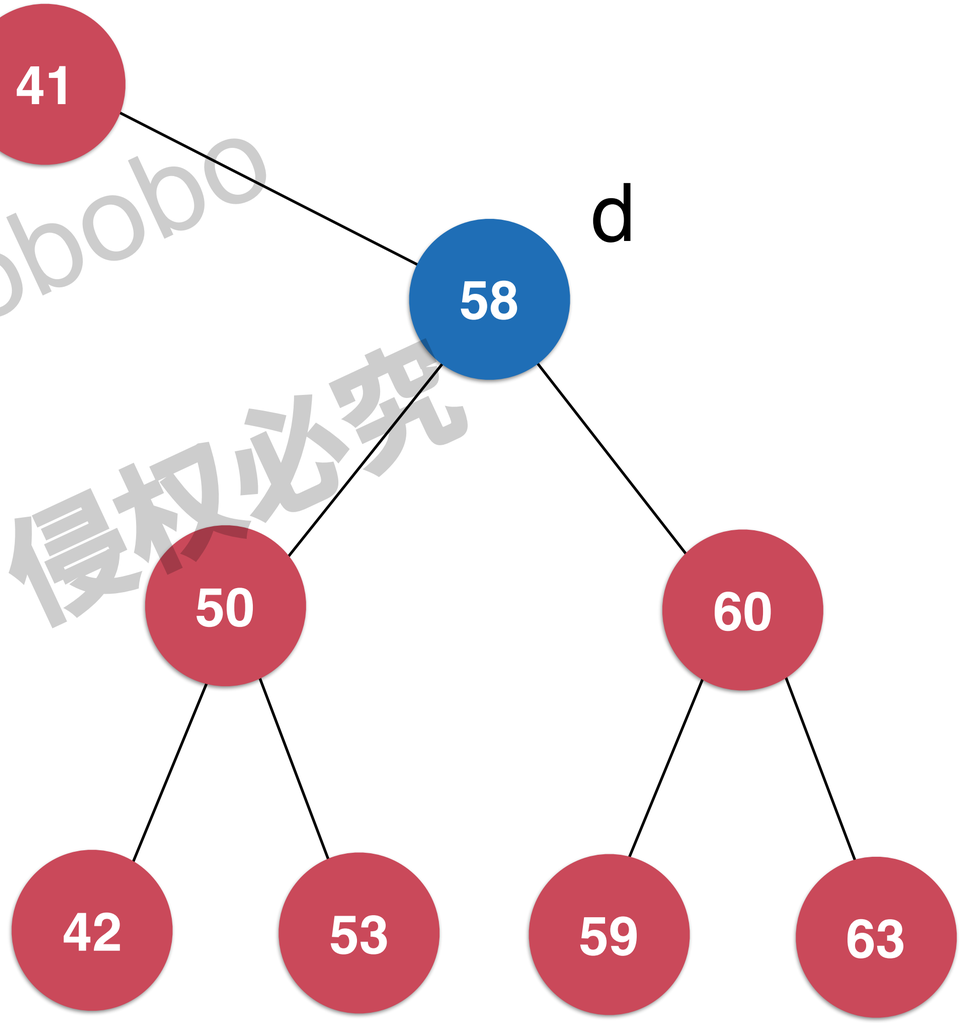
删除左右都有孩子的节点



1962年，Hibbard提出 - Hibbard Deletion

二分搜索树删除节点

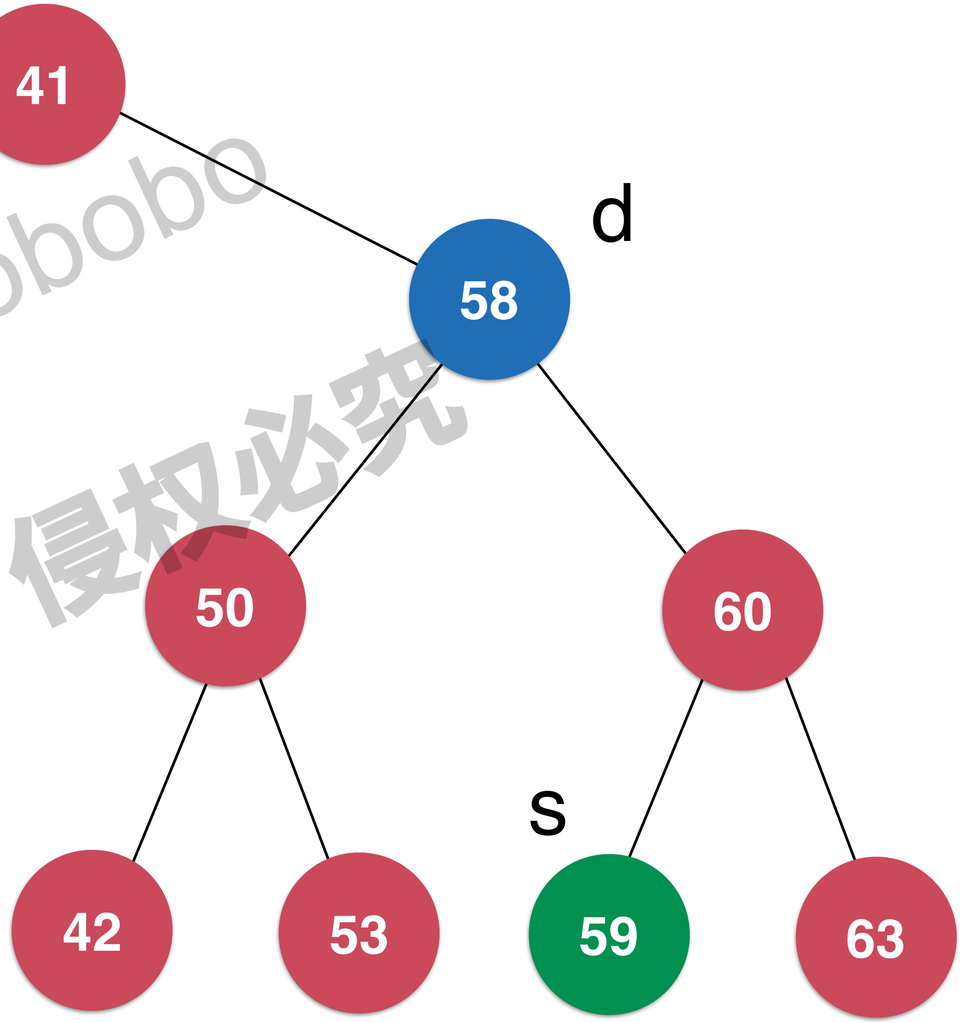
删除左右都有孩子的节点 d



二分搜索树删除节点

删除左右都有孩子的节点 d

找到 s = min(d->right)

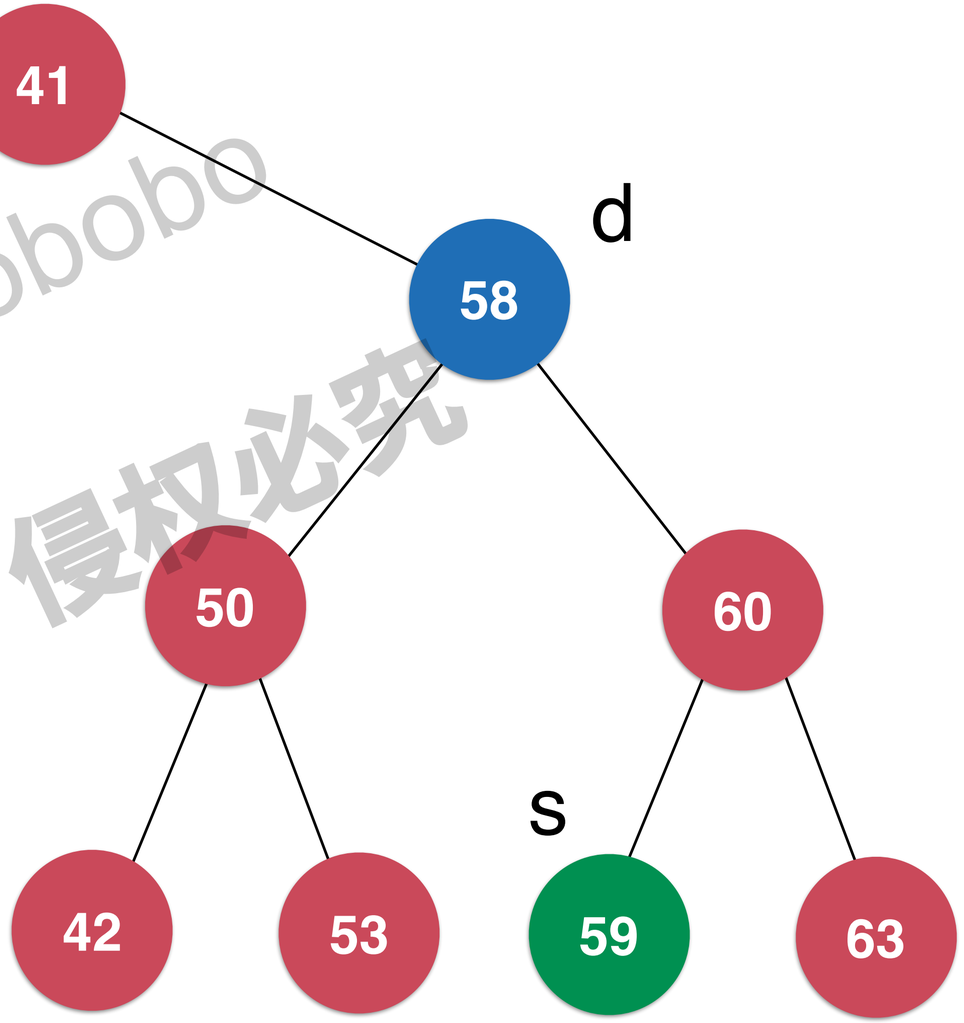


二分搜索树删除节点

删除左右都有孩子的节点 d

找到 s = min(d->right)

s 是 d 的后继



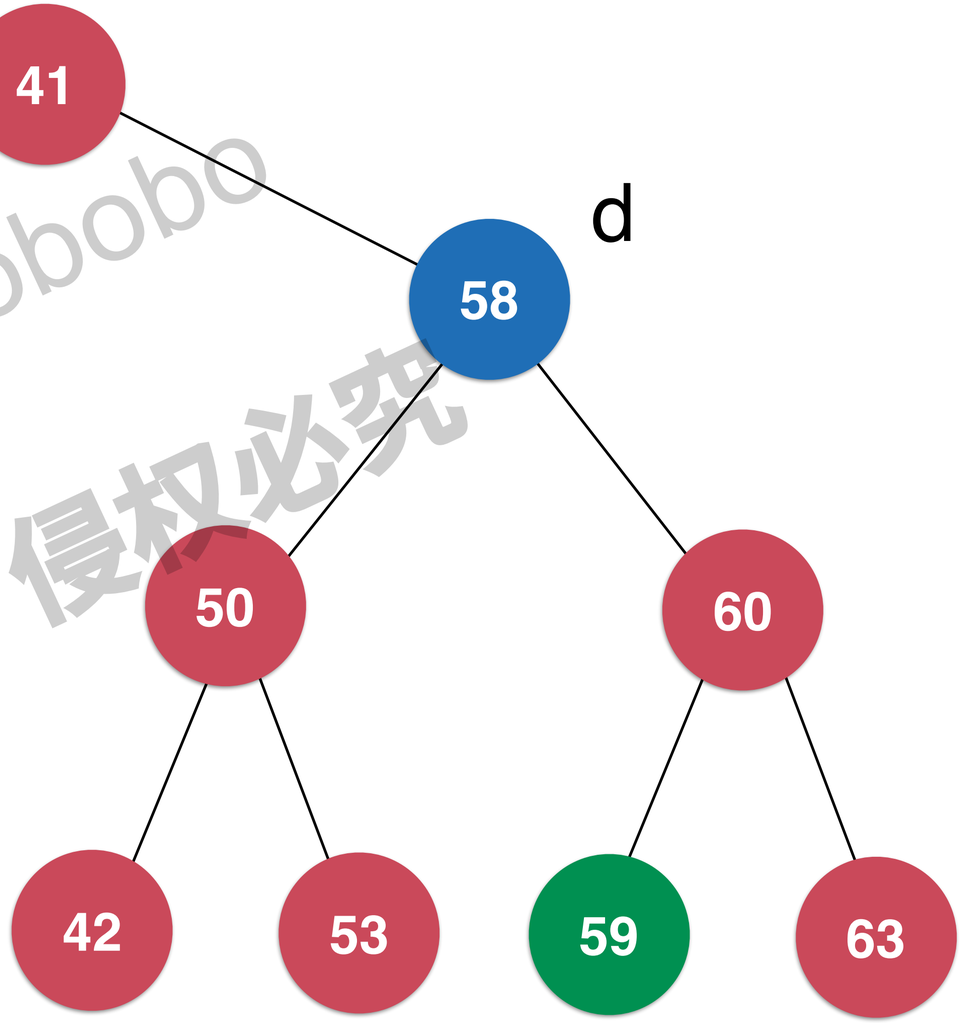
二分搜索树删除节点

删除左右都有孩子的节点 d

找到 s = min(d->right)

s 是 d 的后继

s->right = delMin(d->right)



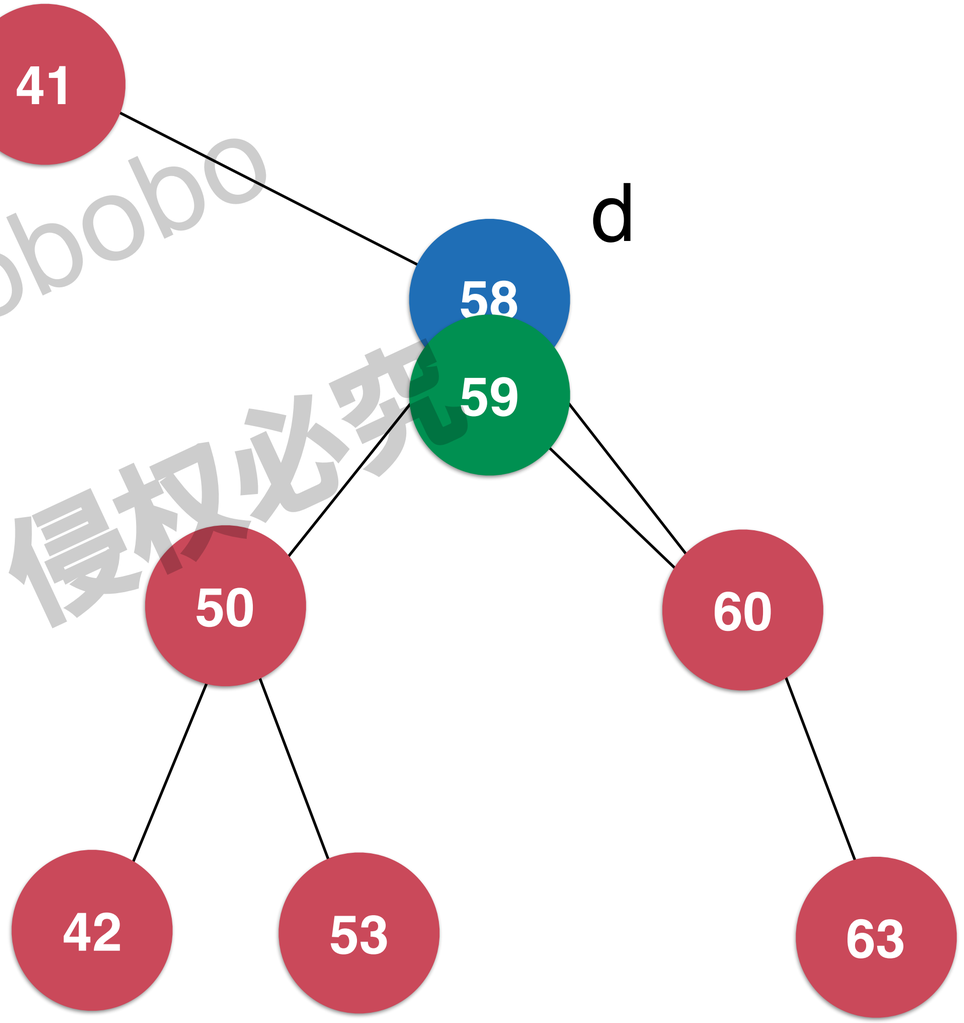
二分搜索树删除节点

删除左右都有孩子的节点 d

找到 s = min(d->right)

s 是 d 的后继

s->right = delMin(d->right)



二分搜索树删除节点

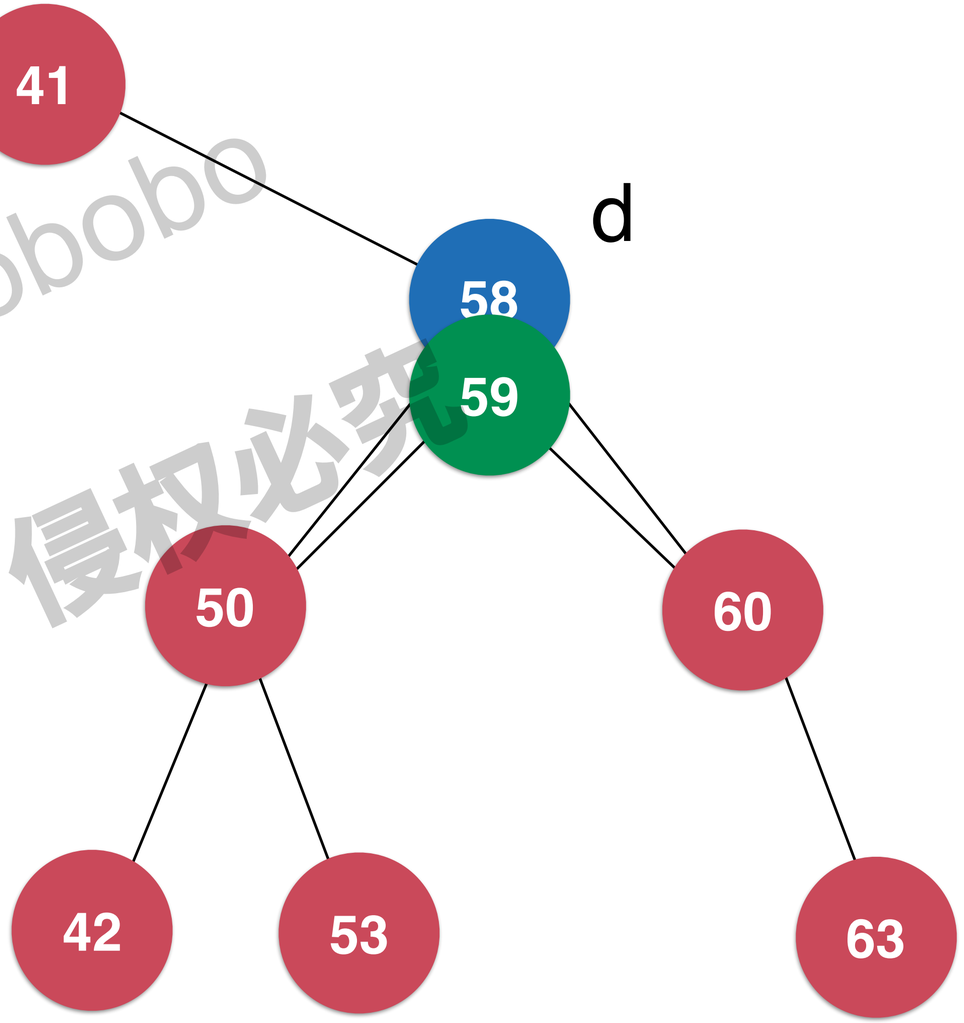
删除左右都有孩子的节点 d

找到 s = min(d->right)

s 是 d 的后继

s->right = delMin(d->right)

s->left = d->left



二分搜索树删除节点

删除左右都有孩子的节点 d

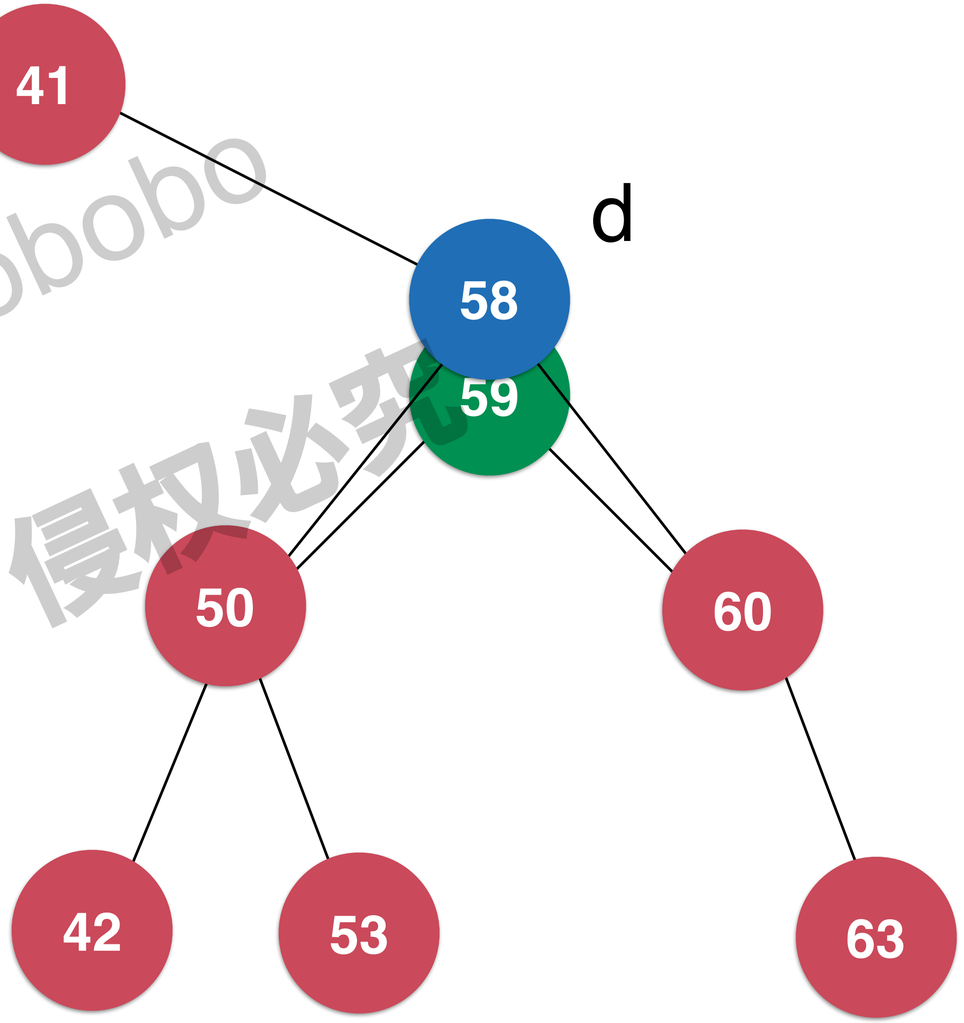
找到 s = min(d->right)

s 是 d 的后继

s->right = delMin(d->right)

s->left = d->left

删除d，s是新的子树的根



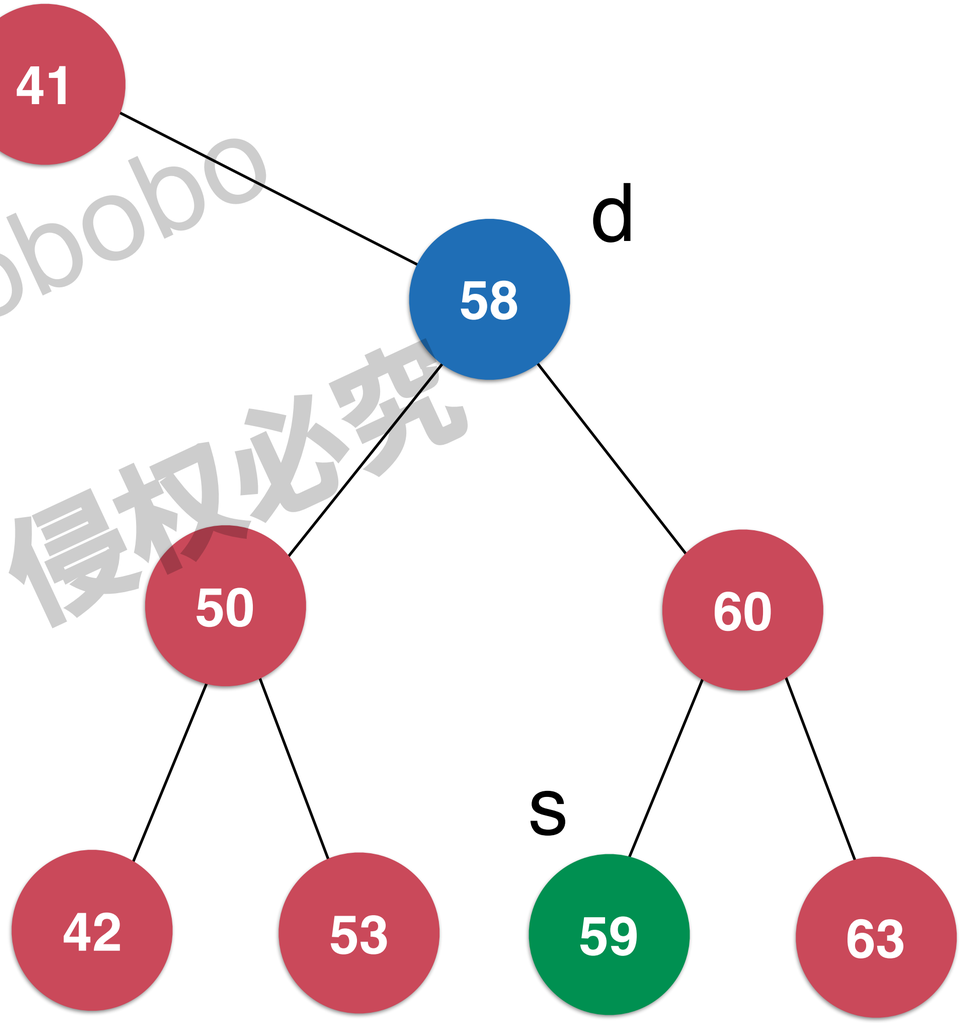
实践：删除二分搜索树的任意一个节点

二分搜索树删除节点

删除左右都有孩子的节点 d

找到 s = min(d->right)

s 是 d 的后继

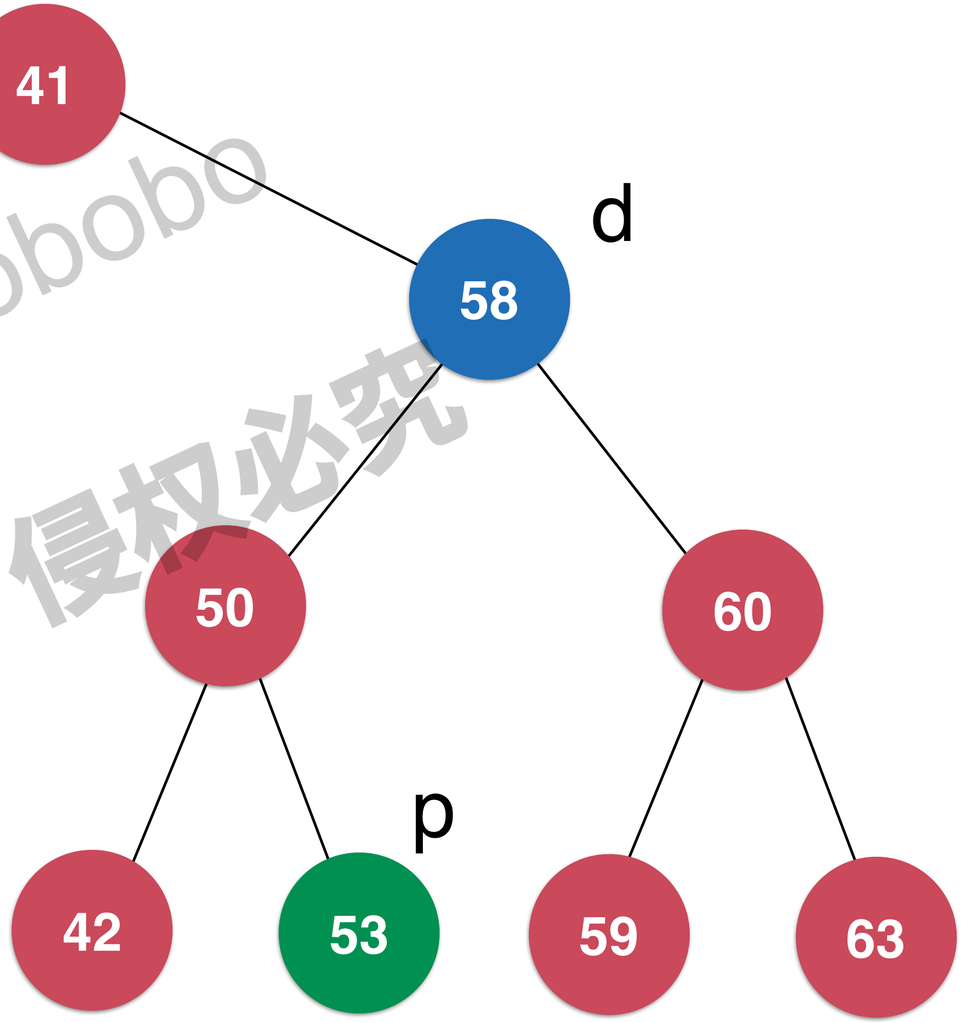


二分搜索树删除节点

删除左右都有孩子的节点 d

找到 p = max(d->left)

p 是 d 的前驱



更多二分搜索树相关的问题

二分搜索树的顺序性

二分搜索树的顺序性

minimum，maximum

二分搜索树的顺序性

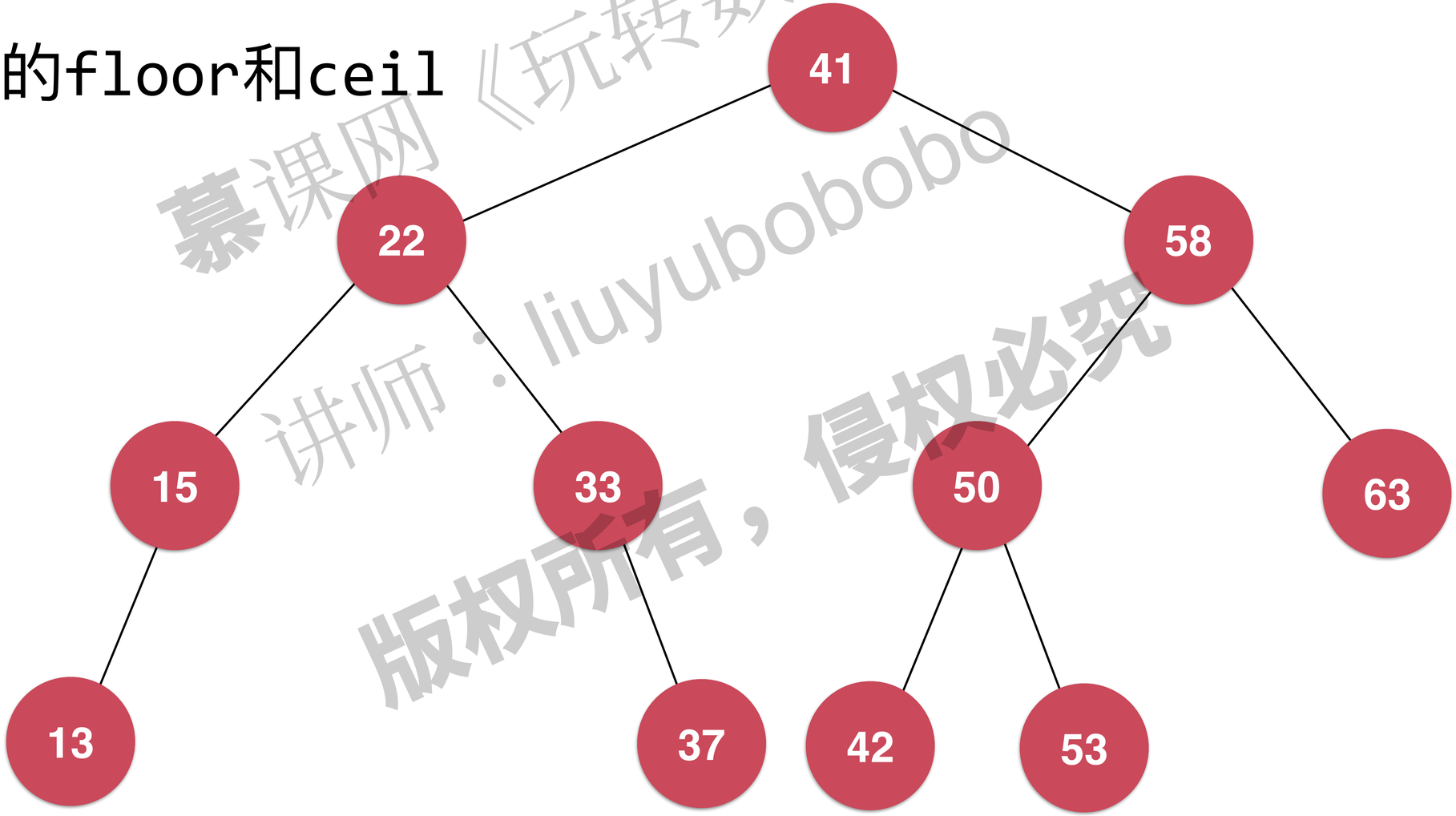
successor，predecessor

二分搜索树的顺序性

floor，ceil

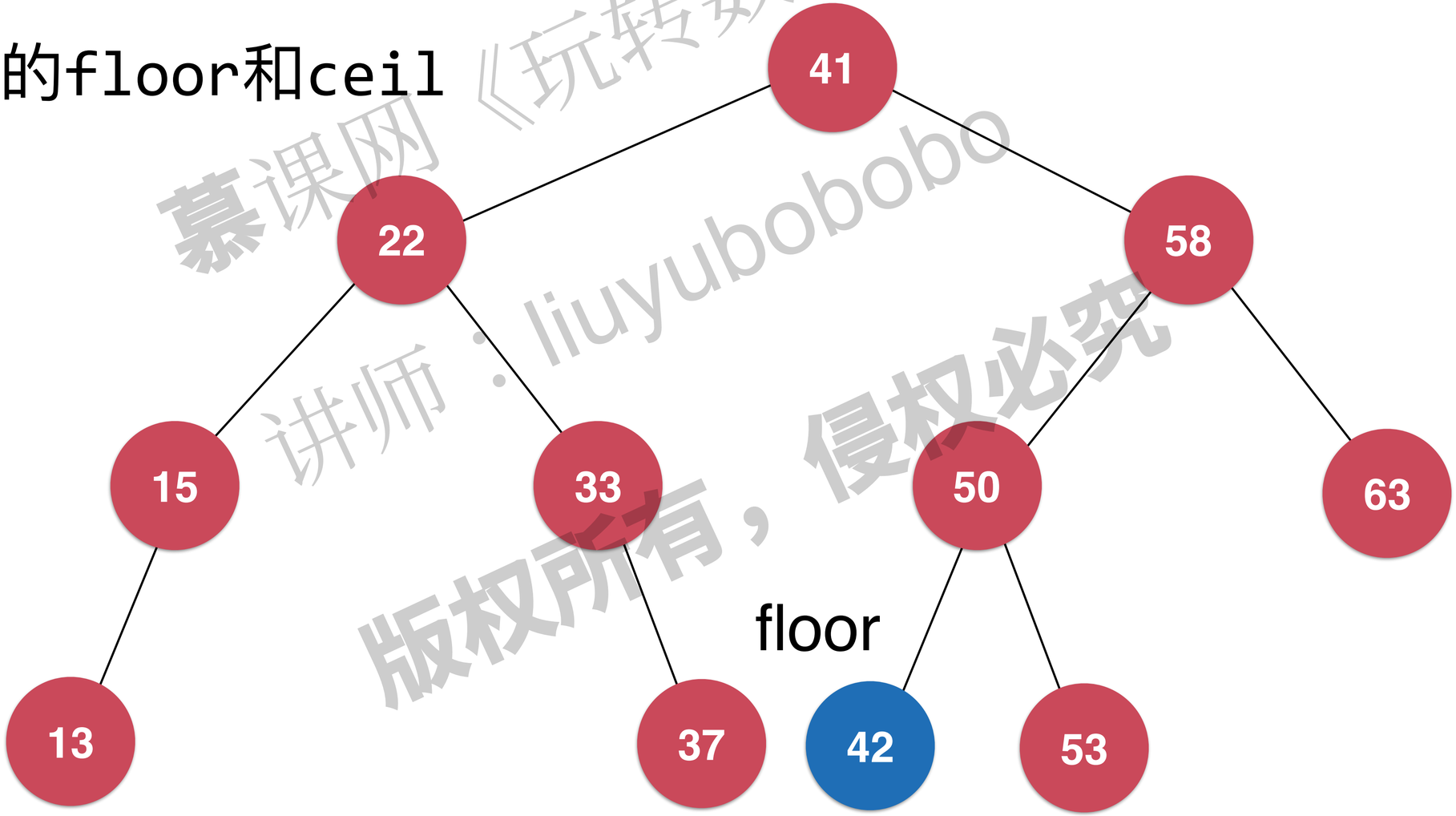
二分搜索树的floor和ceil

寻找45的floor和ceil



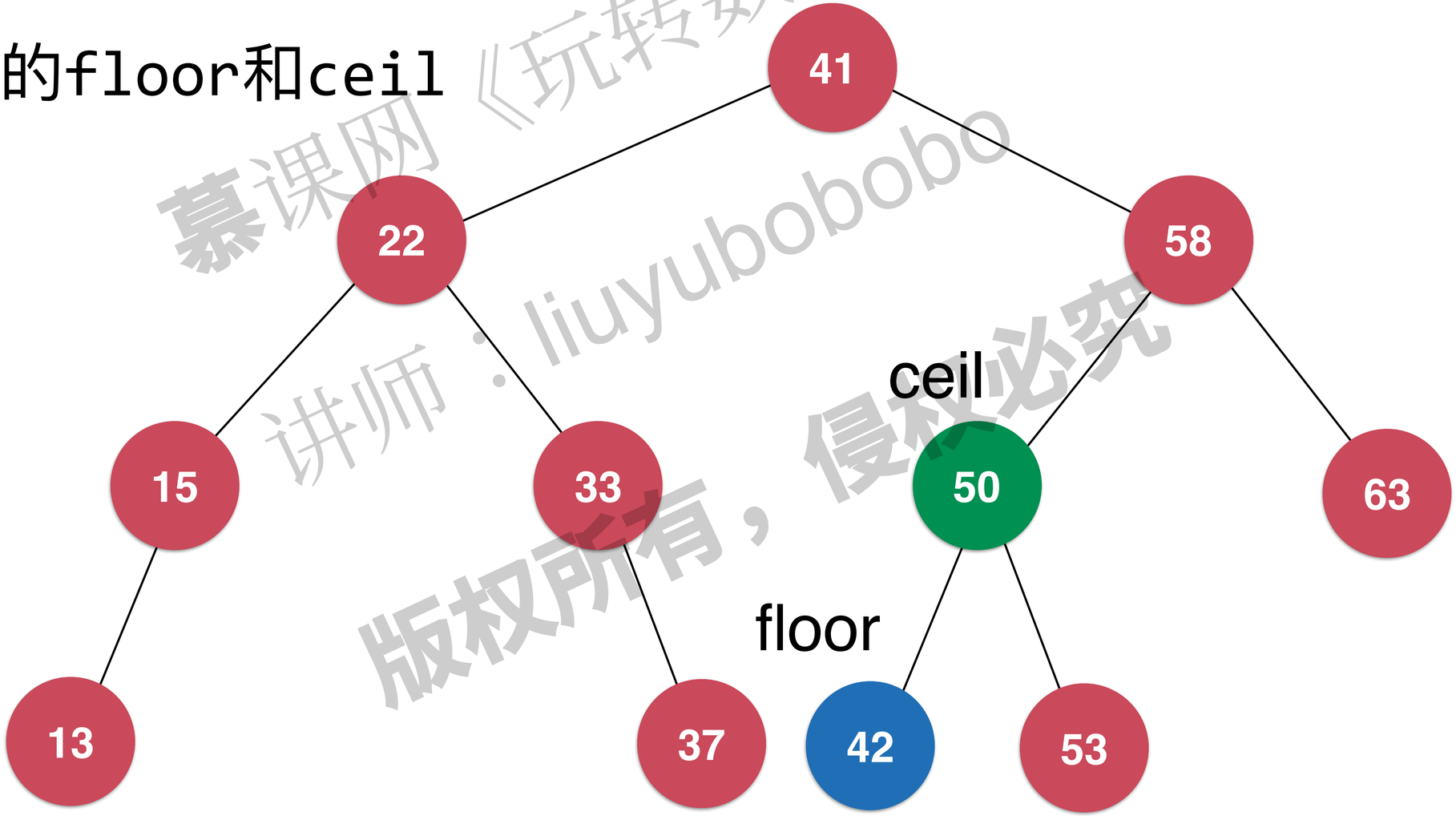
二分搜索树的floor和ceil

寻找45的floor和ceil



二分搜索树的floor和ceil

寻找45的floor和ceil

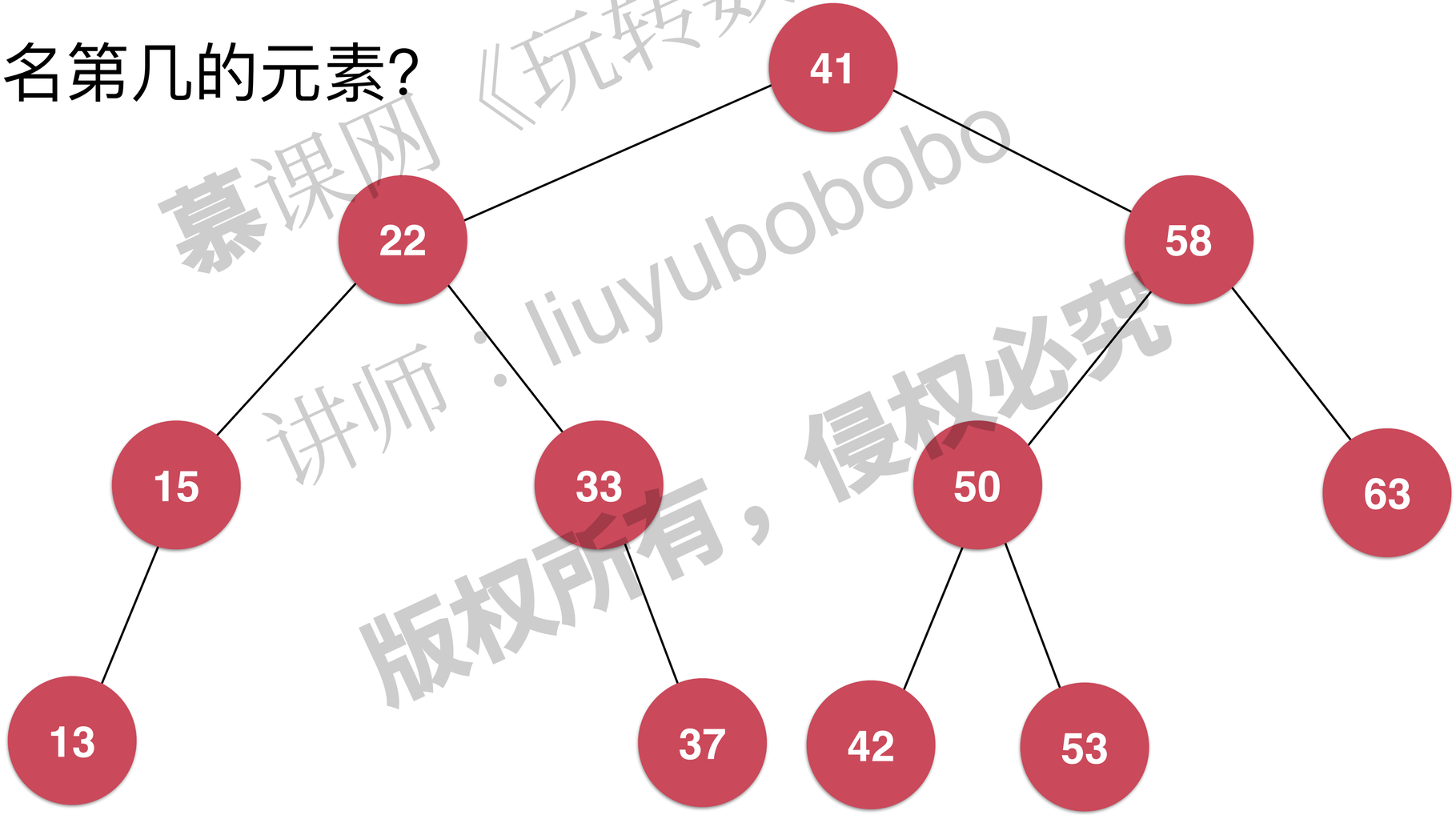


二分搜索树的顺序性

rank，select

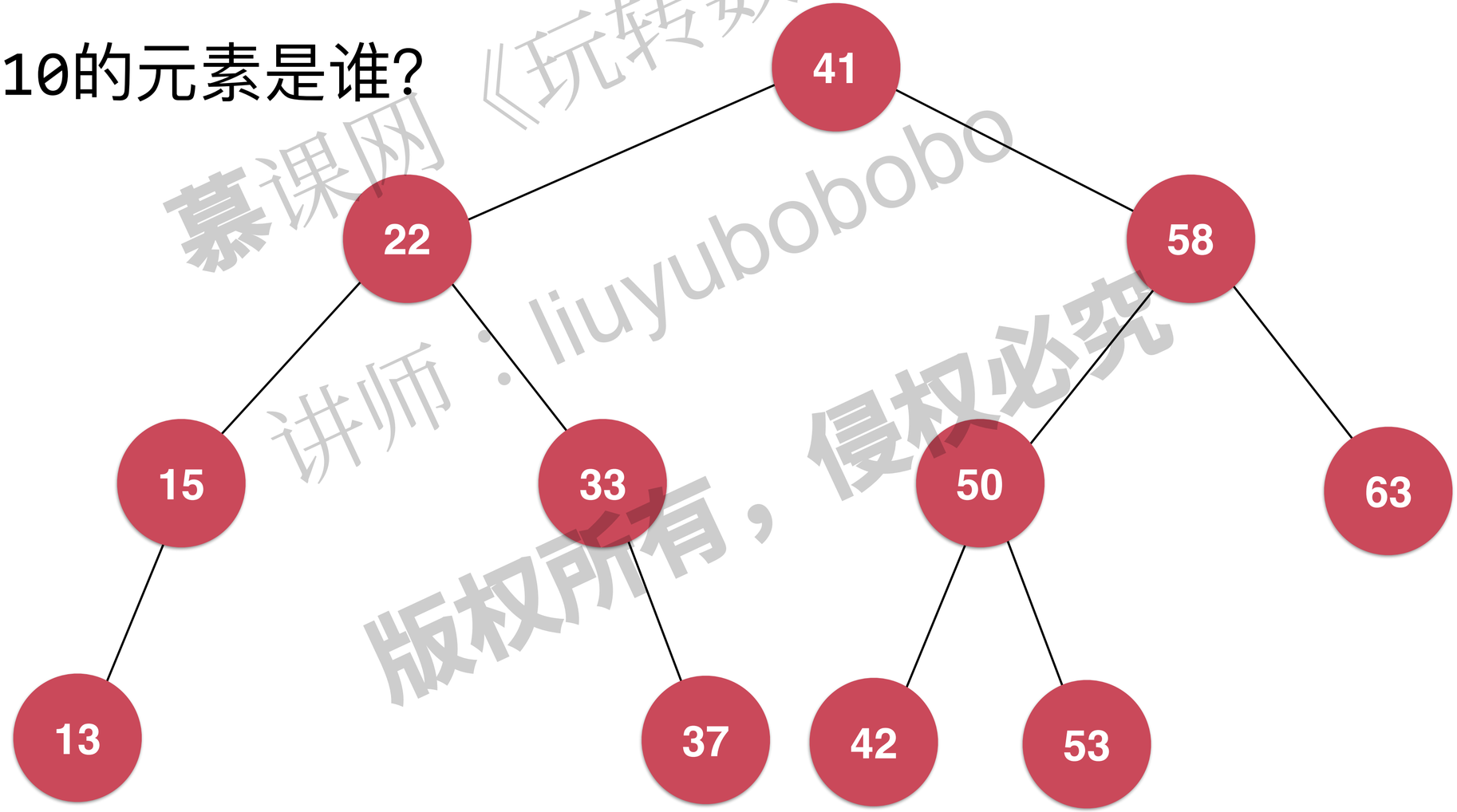
二分搜索树的rank

58是排名第几的元素?

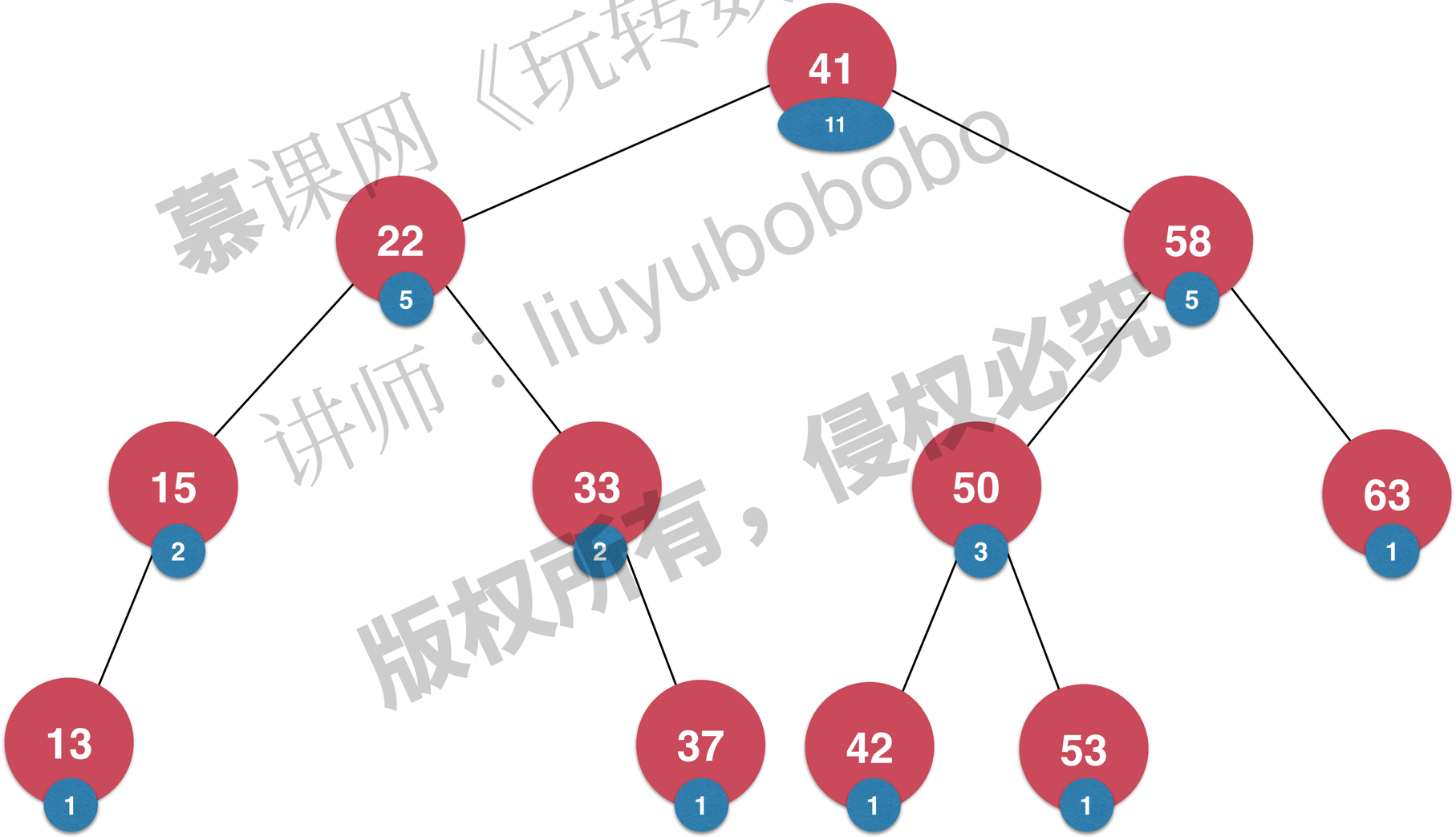


二分搜索树的select

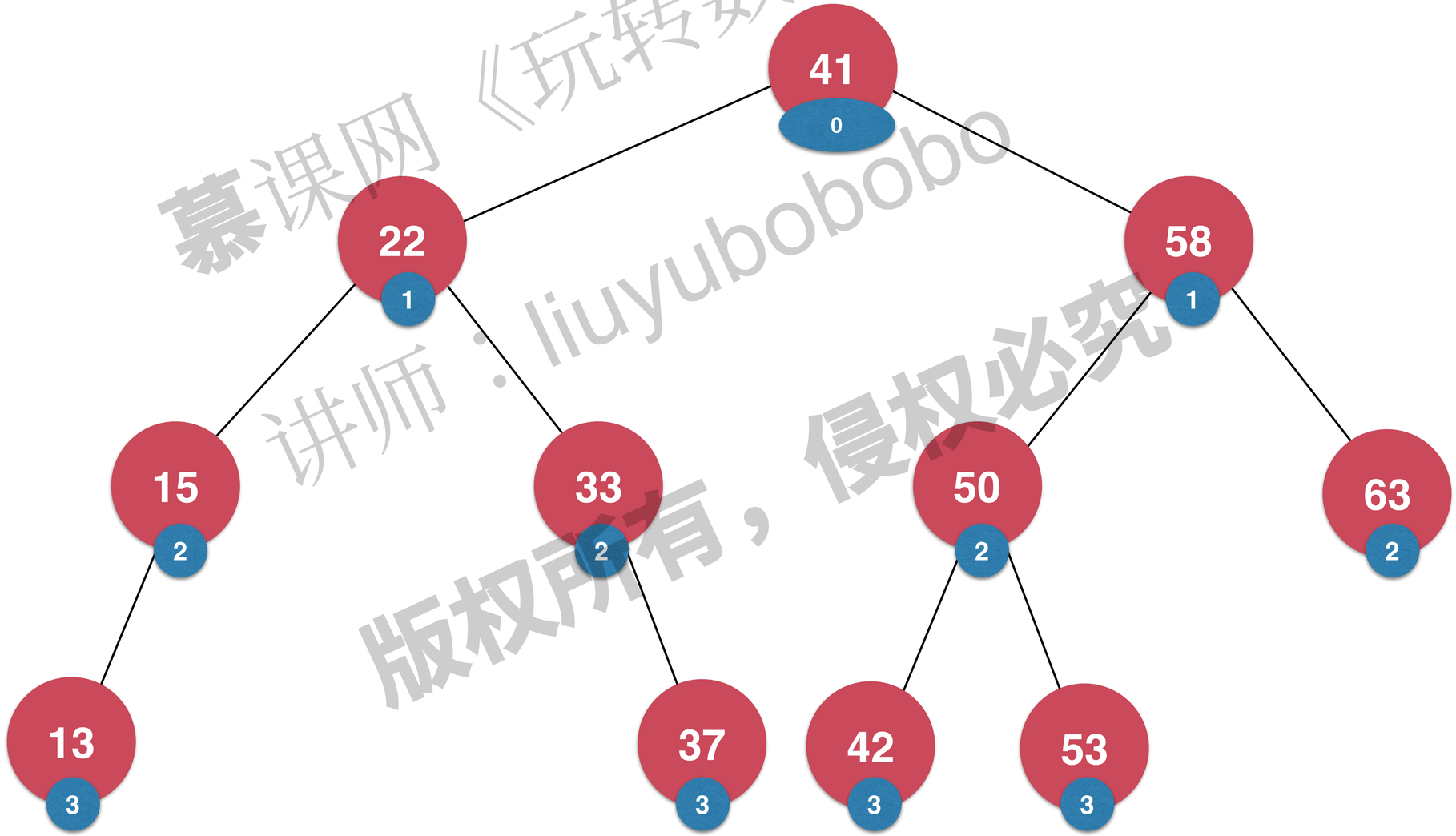
排名第10的元素是谁?



维护size的二分搜索树

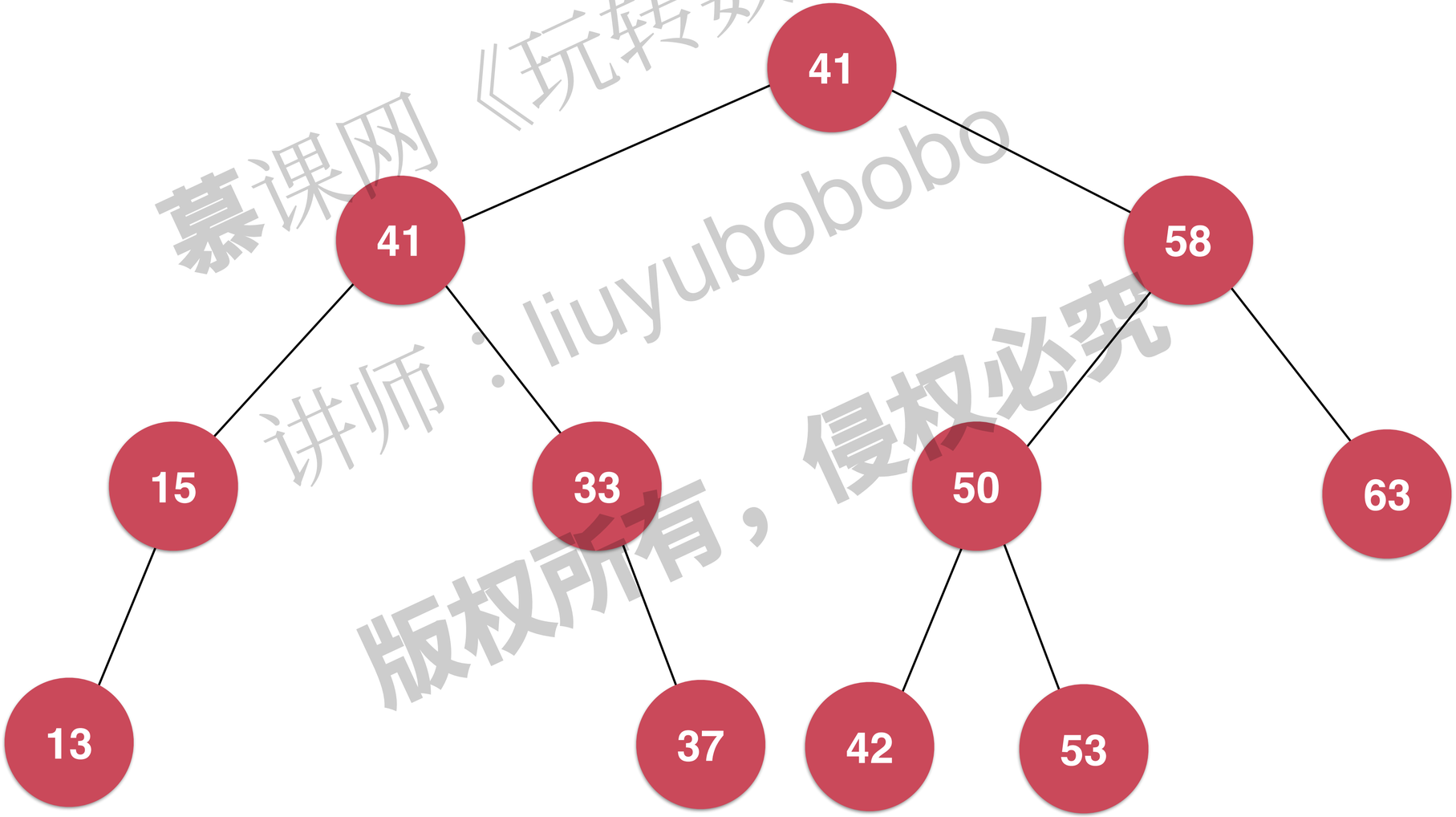


维护depth的二分搜索树

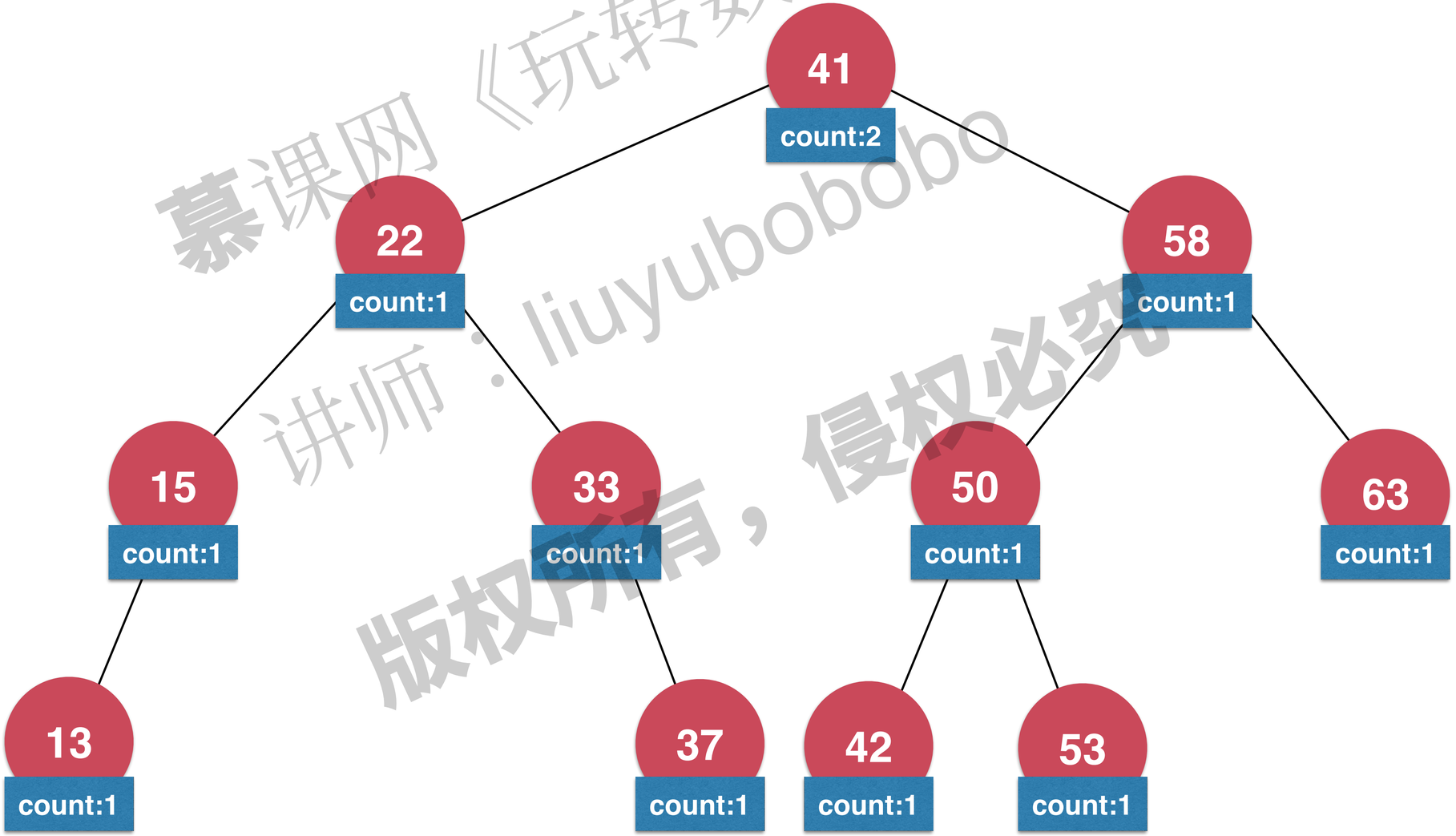


支持重复元素的二分搜索树

支持重复元素的二分搜索树



支持重复元素的二分搜索树



Leetcode上和二分搜索树相关的问题

二分搜索树 Binary Search Tree

其他

欢迎大家关注我的个人公众号：是不是很酷

玩儿转数据结构

liuyubobobo