

# FSE598 前沿计算技术

## 模块 3    算法设计与分析 单元 2    排序和搜索算法 第 3 讲    二叉搜索树

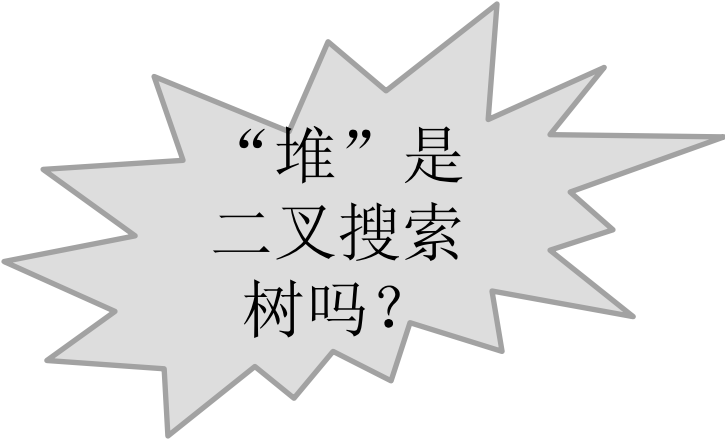
本课程的部分内容是基于 Thomas H. Cormen、Charles E. Leiserson 等人的  
“算法简介”教材

## 学习内容

- ❑ 二叉搜索树的定义
- ❑ 树的遍历：中序、前序和后序
- ❑ 按树的中序遍历进行排序
- ❑ 树的查询：搜索、最小值、最大值、后继节点、前驱节点
- ❑ 插入和删除
- ❑ 树高度

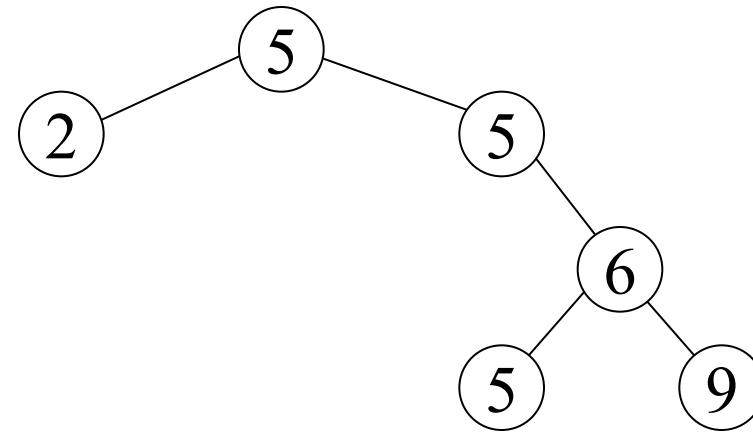
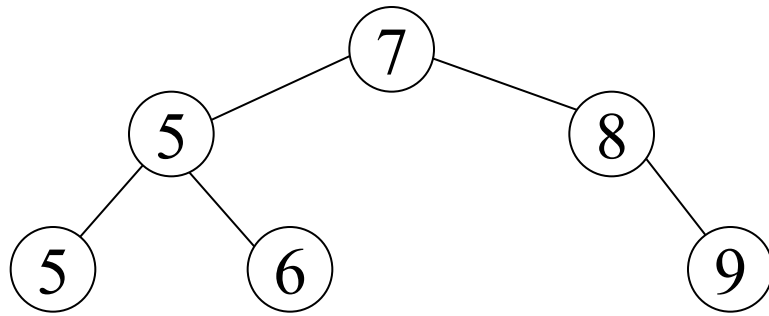
# 二叉搜索树

- **搜索树**是一种支持字典和优先级队列操作的数据结构：
  - 最小值、最大值
  - 前驱节点、后继节点
  - 搜索、插入、删除
  - 排序
- 二叉搜索的基本操作
- 树的运行时间与其高度成正比。
- 对于完整的二叉树，其高度为  $\Theta(\lg n)$
- 定义：**二叉搜索树**是指具有以下属性的二叉树：
  - 对于树中的任意节点  $i$ ，如果  $x$  是其左侧子树中的任意节点， $y$  是其右侧子树中的任意节点，则
$$key[x] \leq key[i] \leq key[y] \quad // \quad x \text{ 的密钥} \leq i \text{ 的密钥} \leq y \text{ 的密钥}$$



“堆”是  
二叉搜索  
树吗？

# 示例



树的遍历:

□ 前序: 根节点 — 左侧子树 — 右侧子树

7 5 5 6 8 9

5 2 5 6 5 9

□ 中序: 左侧子树 — 根节点 — 右侧子树

5 5 6 7 8 9

2 5 5 5 6 9

□ 后序: 左侧子树 — 右侧子树 — 根节点

5 6 5 9 8 7

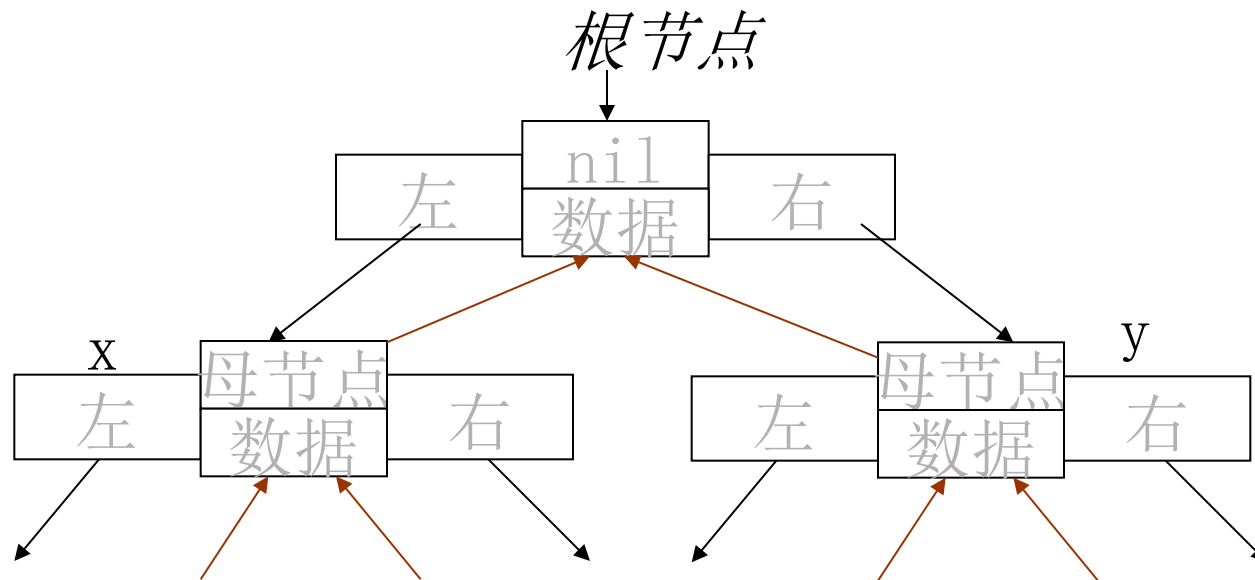
2 5 9 6 5 5

# 二叉树的表示方法

树节点由至少包含四个字段的对象进行定义：  
树节点

`data: string`    // 也称为“密钥”

母节点、左节点、右节点：指向树节点的指针



```
root.left = x
root.right = y
x.parent = root
y.parent = root
```

# 树的中序遍历算法

Inorder-Tree-Walk(x)

1. if  $x \neq \text{nil}$  then
  2.     Inorder-Tree-Walk(x.left)     // 左[x]
  3.     print(x.data)                 // 数据[x]
  4.     Inorder-Tree-Walk(x.right)    // 右[x]
- 

Preorder-Tree-Walk(x)

1. if  $x \neq \text{nil}$  then
2.     print(x.data)
3.     Preorder-Tree-Walk(x.left)
4.     Preorder-Tree-Walk(x.right)

# 示例

按前序打印树:

$*x+*abc$  {前缀表示}

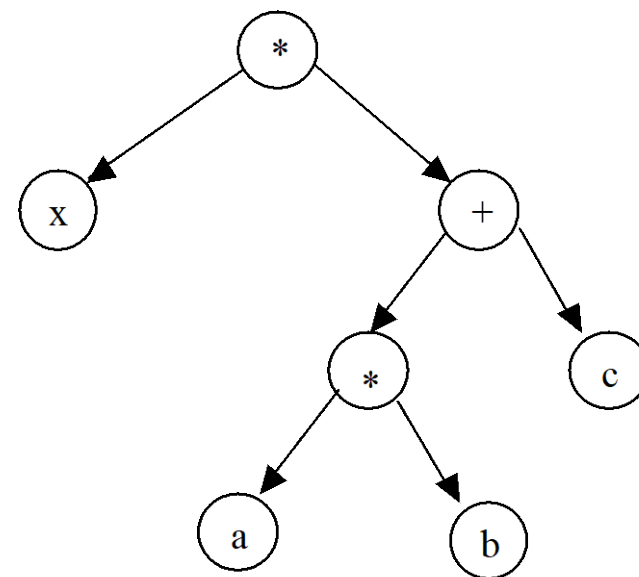
按中序打印树:

$x*a*b + c$  {中缀表示}

$(x*(a*b + c))$

按后序打印树:

$xab*c+*$  {后缀表示}



注意，前缀和后缀表示中无需括号：计算顺序仅由符号序列指定。

**Tree-Search**(root, key)

1. **if** root == nil **or** root.data == key **then**
2.     **return** root
3. **if** key < root.data
4.     **then return** **Tree-Search**(root.left, key)
5.     **else return** **Tree-Search**(root.right, key)

---

**Iterative-Tree-Search**(root, key)

1. **while** root ≠ nil **and** root.data ≠ key **do**
2.     **if** key < root.data
3.         **then** root := root.left
4.         **else** root := root.right
5. **return** root

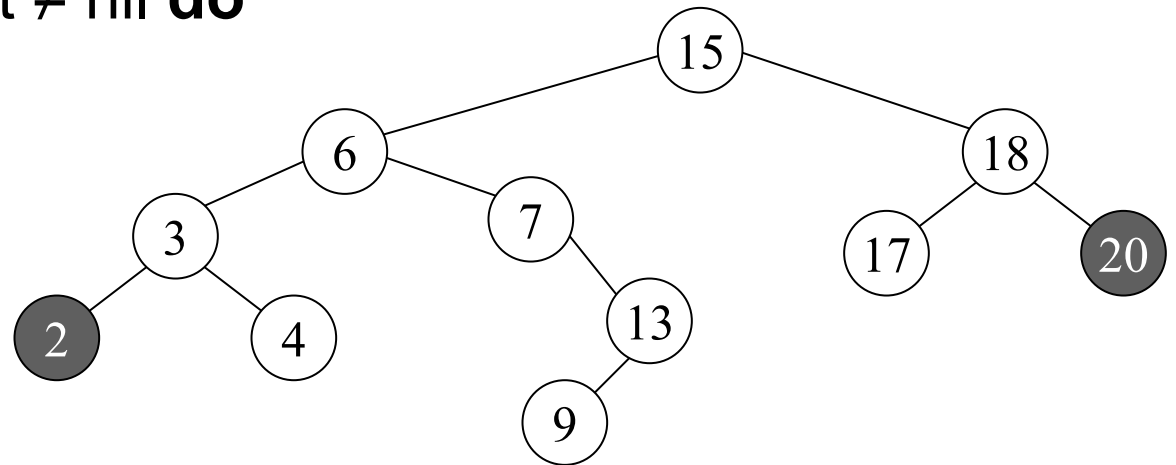


## Tree-Minimum(root)

1. **while** root  $\neq$  nil **and** root.left  $\neq$  nil **do**
2.     root := root.left
3. **return** root

## Tree-Maximum(root)

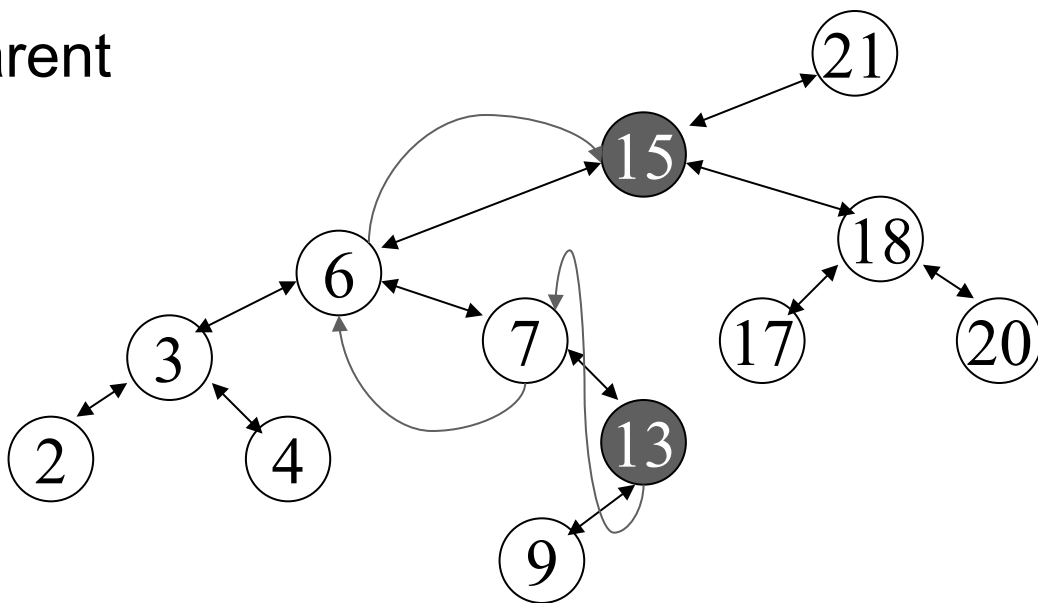
1. **while** root  $\neq$  nil **and** root.right  $\neq$  nil **do**
2.     root := root.right
3. **return** root



# 二叉搜索的操作（接上页）

## Successor(x)

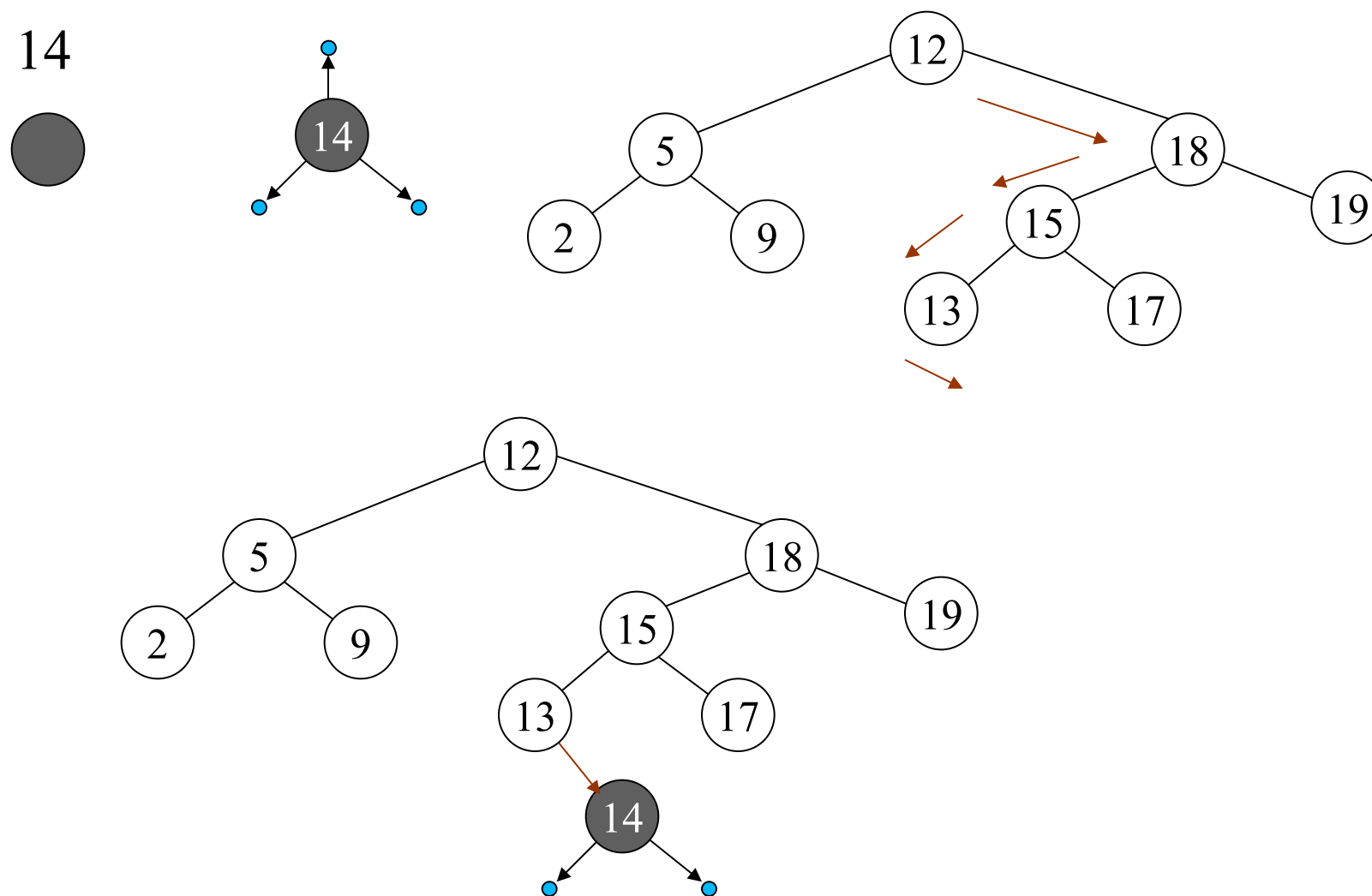
1. **if**  $x.\text{right} \neq \text{nil}$  **then**
2.      $\text{return Tree-Minimum}(x.\text{right})$      //e.g., 15 --> 17
3.      $y := x.\text{parent}$      // e.g., node 13
4.     **while**  $y \neq \text{nil}$  **and**  $x == y.\text{right}$  **do**     // 找到“右侧”的母节点: x 是 y 的左子节点
5.          $x := y$
6.          $y := y.\text{parent}$
7.     **return**  $y$



$x = \&(13)$   $y = \&(7)$   
 $x = \&(7)$   $y = \&(6)$   
 $x = \&(6)$   $y = \&(15)$   
此时  $x \neq y.\text{right}$

运行时间为  $O(h)$ : 向上或向下移动

# 示例：如何插入节点？



# 二叉搜索的操作（接上页）

## Tree-Insertion( $T$ , key)

1.  $z := \text{new-treenode}(\text{key}, \text{nil}, \text{nil}, \text{nil})$
2.  $x := \text{root}[T]$
3. **if**  $x = \text{nil}$  **then**  $\text{root}[T] := z$  **return**  
**else**
4.     **while**  $x \neq \text{nil}$  **do**
5.          $y := x$
6.         **if**  $\text{key} < x.\text{data}$
7.             **then**  $x := x.\text{left}$
8.             **else**  $x := x.\text{right}$
9.      $z.\text{parent} = y$
10.    **if**  $\text{key} < y.\text{data}$
11.       **then**  $y.\text{left} := z$
12.       **else**  $y.\text{right} := z$

用搜索找到位置  $p$



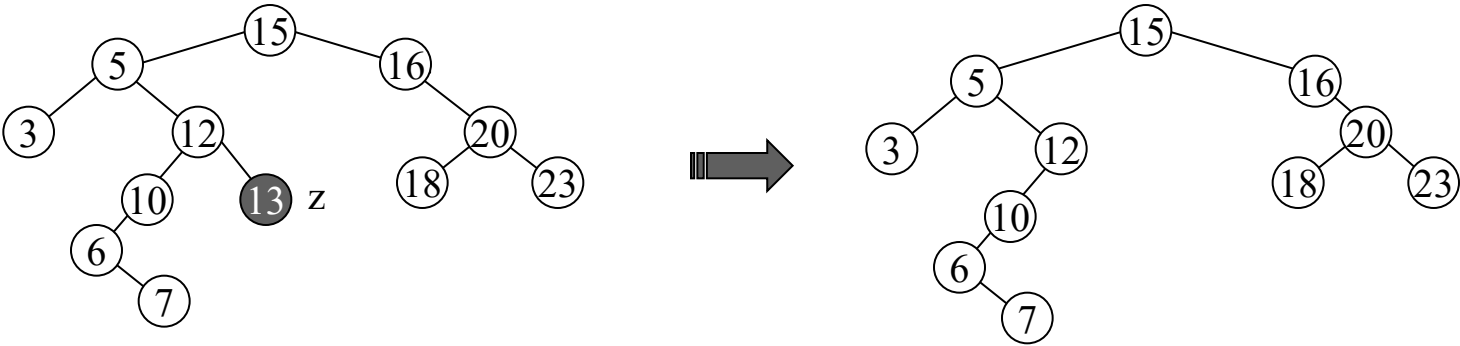
```
z := new-treenode(k, p, l, r)
z.data := k
z.parent := p
z.left := l
z.right := r
```

运行时间为  $O(h)$ : 从根节点向下移动。

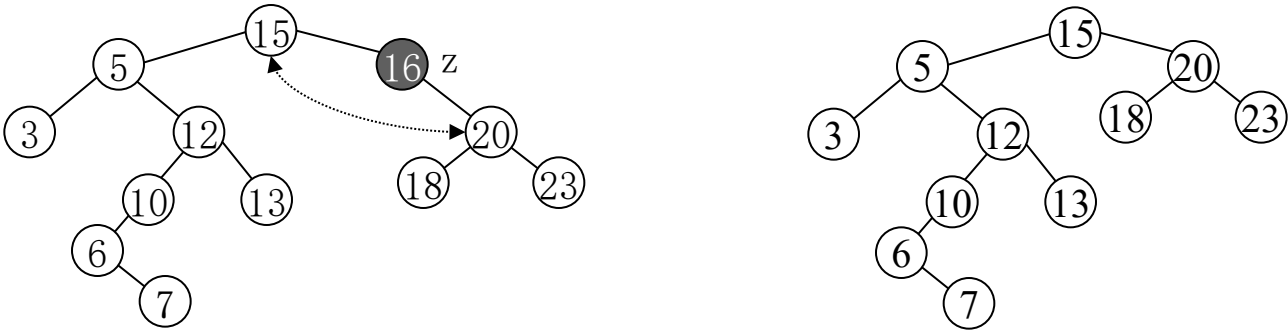
# 示例：如何删除节点？

考虑三种情况：  
(还有很多其他情况)

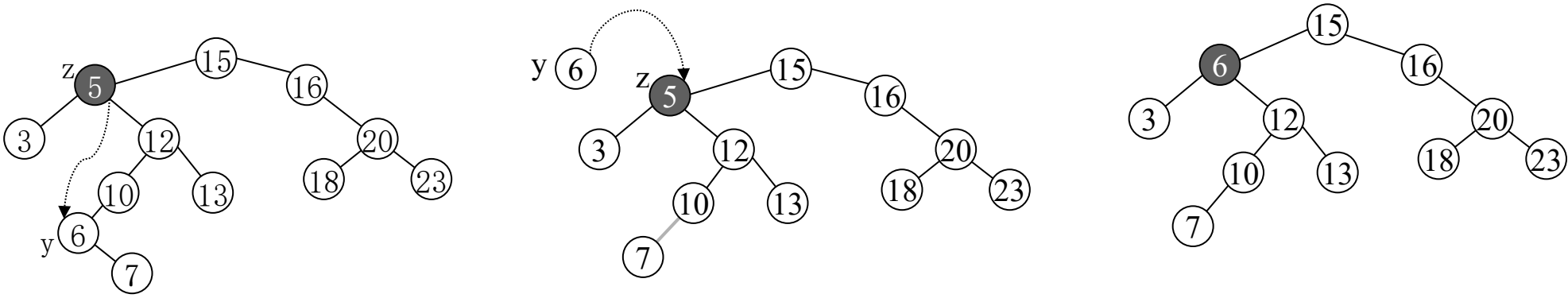
(1)  
无子节点



(2)  
有一个子节点



(3)  
有两个子节点



# 二叉搜索的操作（接上页）

// 该算法仅考虑在前面的示例中的三种情况

Tree-Deletion( $T, z$ ) //  $z$  是要删除的节点的索引

1.     **if**  $z.\text{left} == \text{nil}$  **or**  $z.\text{right} == \text{nil}$

2.         **then**  $y := z$

3.         **else**  $y := \text{Tree-Successor}(z)$

4.     **if**  $y.\text{left} \neq \text{nil}$

5.         **then**  $x == y.\text{left}$

6.         **else**  $x == y.\text{right}$

7.     **if**  $x \neq \text{nil}$

8.         **then**  $x.\text{parent} := y.\text{parent}$

9.     **if**  $y.\text{parent} == \text{nil}$

10.         **then**  $\text{root}[T] := x$

11.         **else if**  $y == y.\text{parent}.\text{left}$      //  $y$  是其母节点的左侧子节点

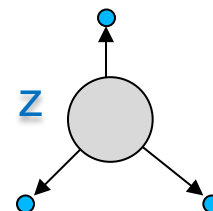
12.             **then**  $y.\text{parent}.\text{left} := x$

13.             **else**  $y.\text{parent}.\text{right} := x$

14.     **if**  $y \neq z$

15.         **then**  $z.\text{data} := y.\text{data}$      // 如果节点有其他数据字段，应全部复制

16.     **return**  $y$



$O(h)$

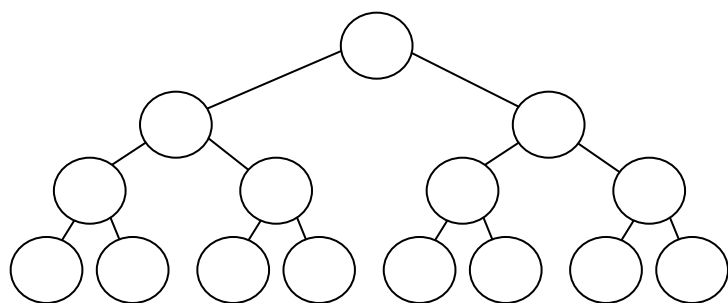
运行时间为  
 $O(h)$   
不存在其他循环

# 树的高度

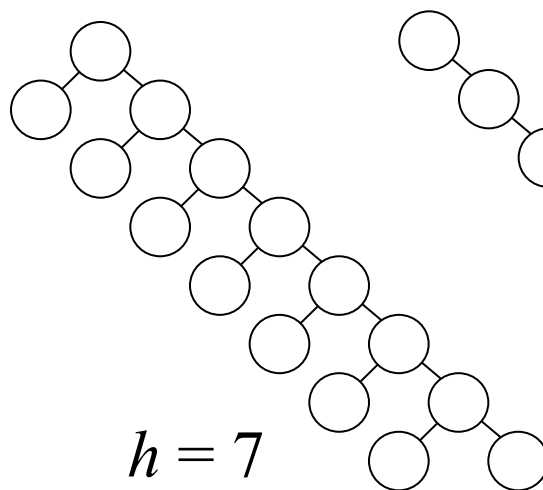
二叉搜索树上的所有动态操作都是  $O(h)$

$h$  和  $n$  之间的关系是什么？

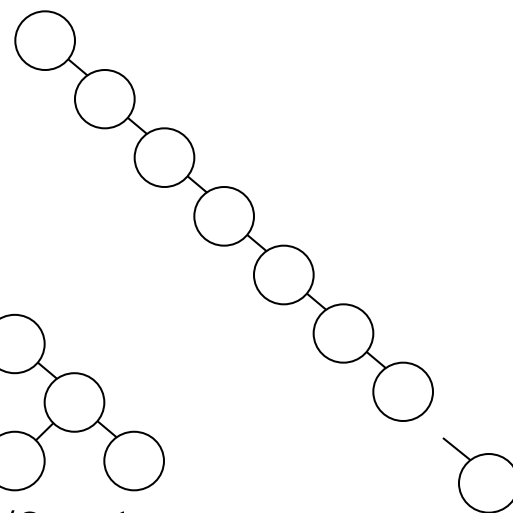
存储在树中的密钥数是多少？如果  $n = 15$ :



$$\begin{aligned}h &= 3 \\&= \lg(n+1) - 1 \\&= O(\lg n)\end{aligned}$$



$$\begin{aligned}h &= 7 \\&= (n+1)/2 - 1 \\&= O(n)\end{aligned}$$



$$\begin{aligned}h &= n - 1 \\&= O(n)\end{aligned}$$

**定理：**一个根据  $n$  个不同密钥随机构建（插入）的二叉搜索树的平均高度为  $O(\lg n)$

# 二叉搜索树的总结

我们探讨了

- 二叉搜索树的定义

$$\text{key}[\text{in left subtree}] \leq \text{key}[\text{root}] \leq \text{key}[\text{right subtree}]$$

- 树的遍历：中序、前序和后序

- 按树的中序遍历进行排序

- 树的查询：搜索、最小值、最大值、后继节点、前驱节点，运行时间均为  $O(h)$ 。

- 插入和删除的运行时间为  $O(h)$ 。

- 树的高度非常重要：一个根据  $n$  个不同密钥随机构建的二叉搜索树的平均高度为  $O(\lg n)$