

数据结构与算法

DATA STRUCTURE

第十五讲 二叉查找树

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 二叉查找树

顺序查找

- 二分法查找
- 插值查找
- Fibonacci查找

二叉查找树

Binary Search Tree

内容

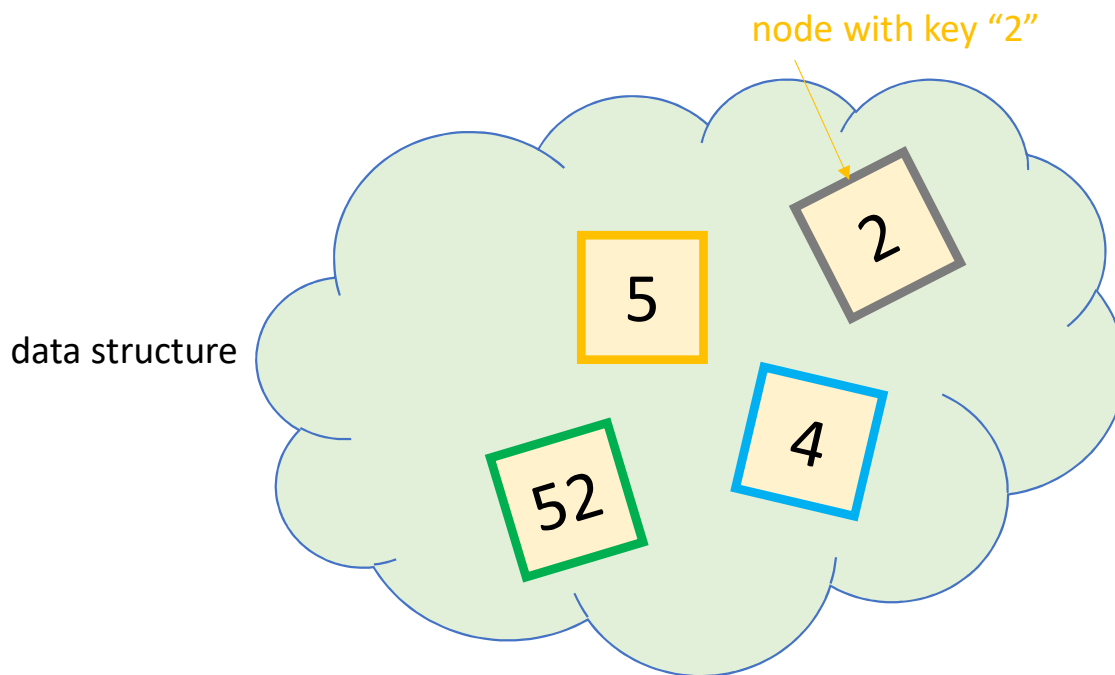
- 为什么使用二叉查找树
- 什么是二叉查找树 **Binary Search Tree**
- 怎么维护二叉查找树

内容

- 为什么使用二叉查找树
- 什么是二叉查找树Binary Search Tree
- 怎么维护二叉查找树

问题:

- 假设有几百万数据+需要维护，如何设计数据结构能够支持高效的查找/插入/删除.



回忆之前的动态数据结构

1. Sorted linked lists:



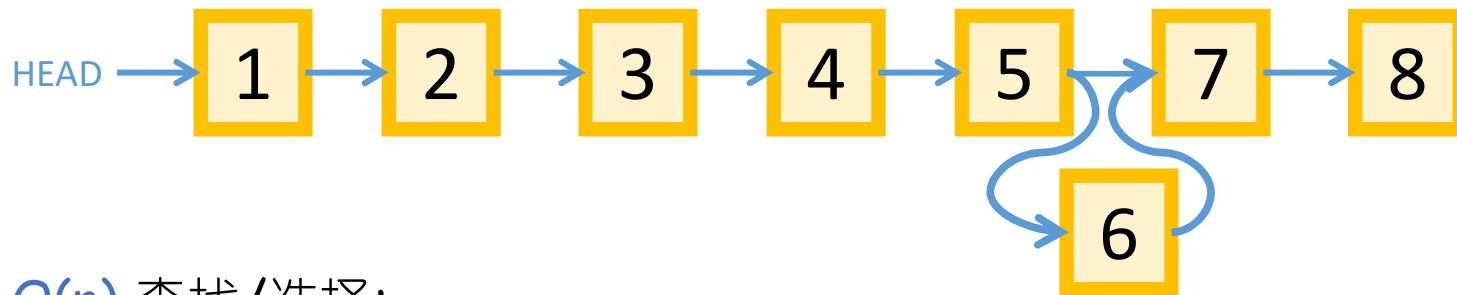
2. Sorted array:



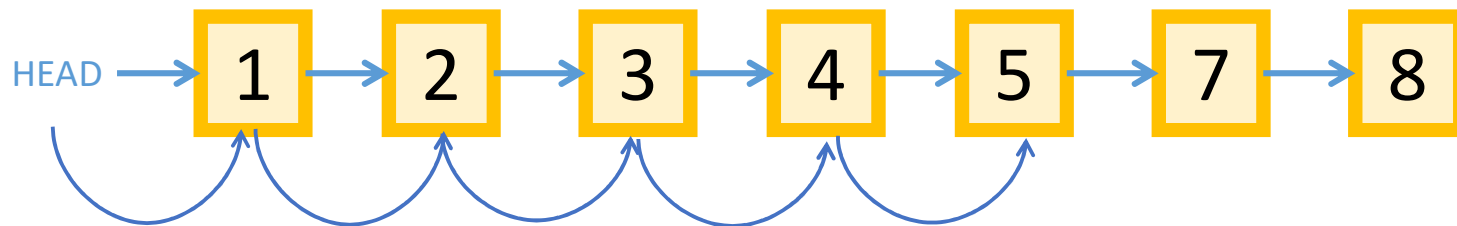
Stack, queue, heap都不能满足要求，比array，linklist还差

Sorted linked lists

- $O(1)$ 插入/删除 (假设已经知道需要插入/删除的节点):



- $O(n)$ 查找/选择:



Sorted Arrays

- $O(n)$ 插入/删除 :



- $O(\log(n))$ 查找, $O(1)$ 选择:









Search: 二分法查找3.

Select: 用下标3直接寻址A[3].

The best of both worlds

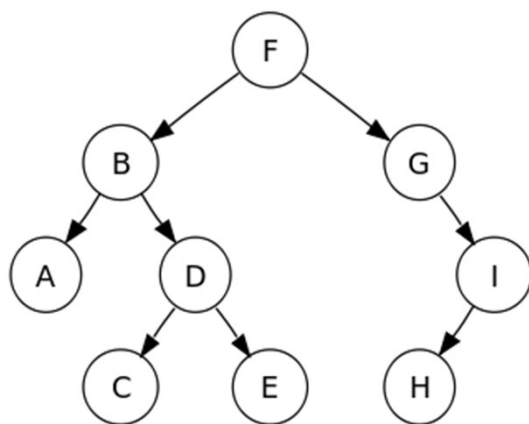
TODAY!

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$ 
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$ 

内容

- 为什么使用二叉查找树
- 什么是二叉查找树 **Binary Search Tree**
- 怎么维护二叉查找树

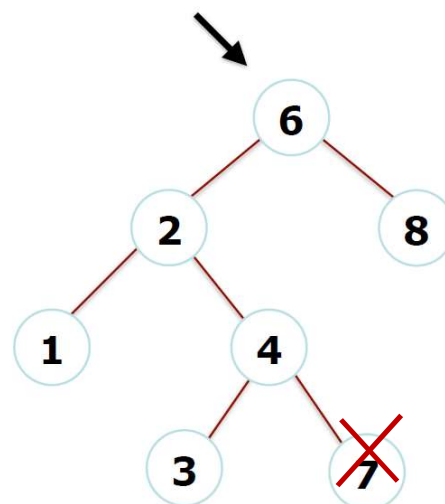
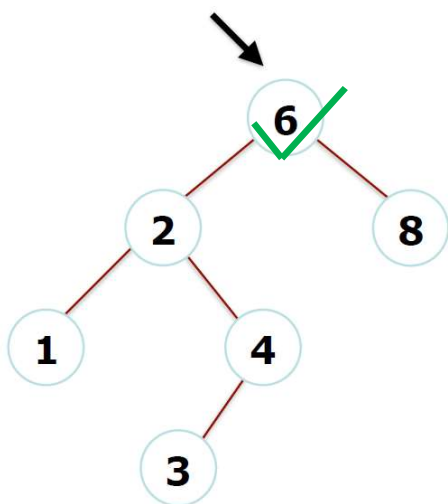
首先是个二叉树



二叉查找树

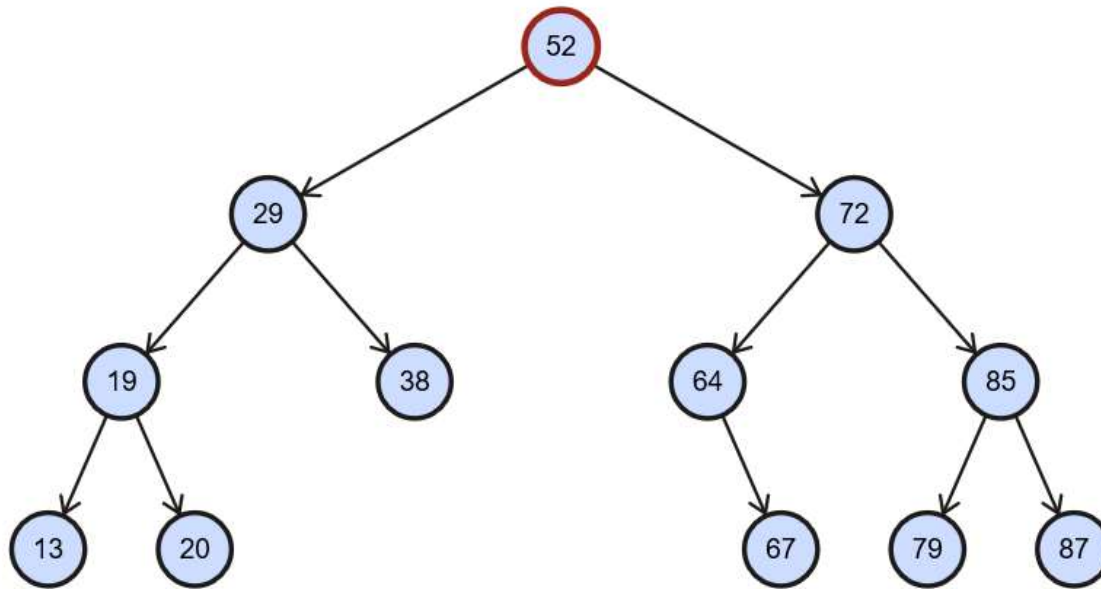
满足下面性质的二叉树: 对任何一个节点元素X

- 它左子树里的任何一个节点元素都小于X.
- 它右子树里的任何一个节点元素都大于X.



- 注意任意子树都是BST

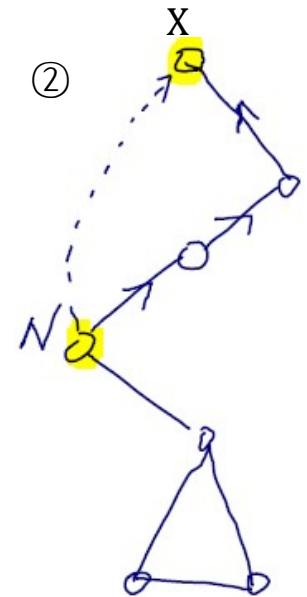
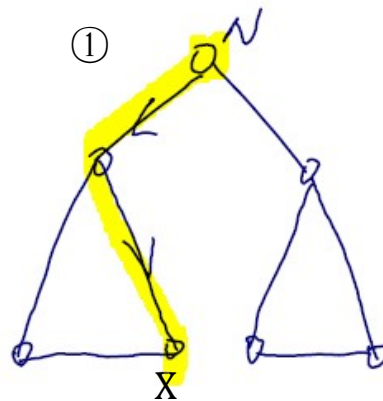
最大值/最小值



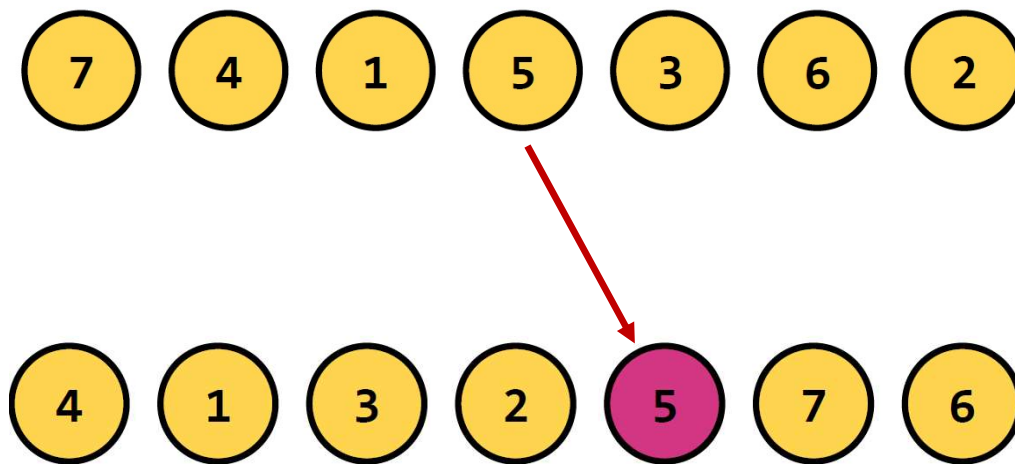
```
// Utility function to find leftmost  
// node in a tree rooted with root  
Node* Leftmost(Node * root)  
{  
    while (root && root->left)  
    {  
        root = root->left;  
    }  
    return root;  
}
```

前驱predecessor/后继successor

- 我们在线索化二叉树里，涉及到了前驱/后继，按中序遍历
- 在BST里，前驱/后继有了更明确的意义
 - 前驱就是所有元素里刚好比它小的元素
 - 后继就是所有元素里刚好比它大的元素
- BST的中序遍历，正好是从小到大的排序

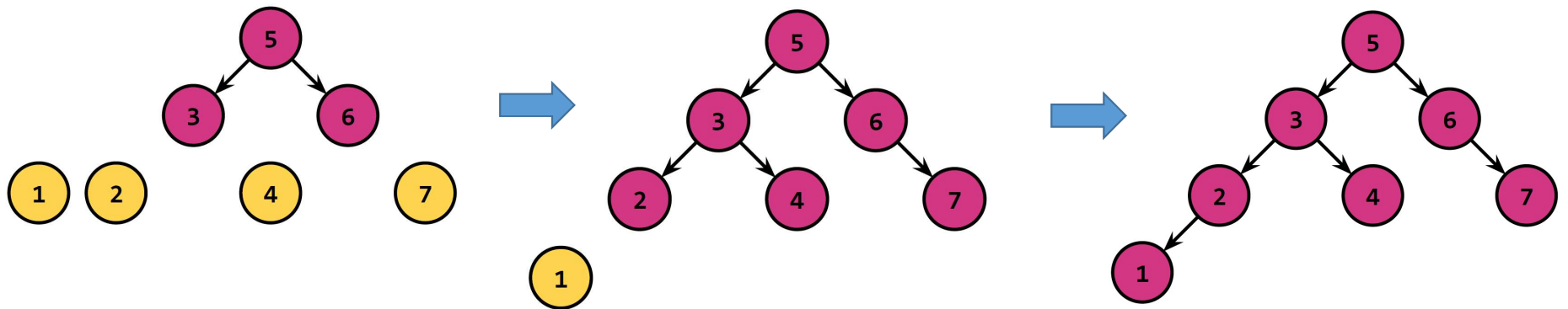


如何建立二叉查找树

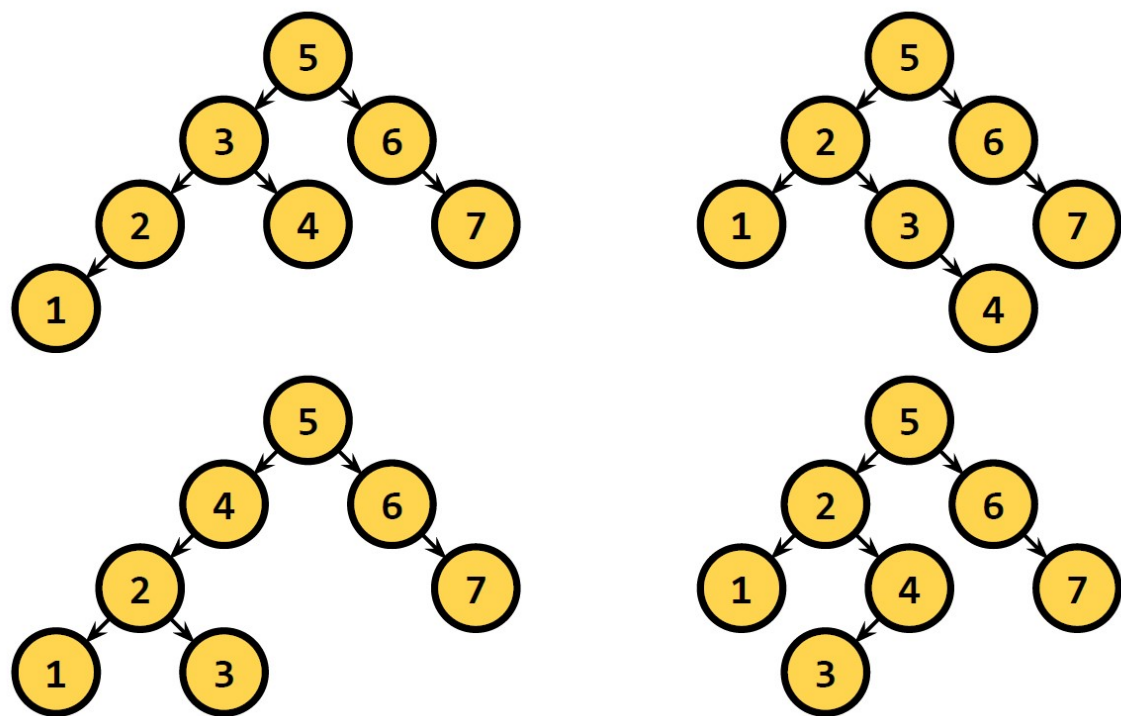


选定一个数据元素作为根节点，并以它为基轴，所有小的元素放左边，大的元素在右边

如何建立二叉查找树



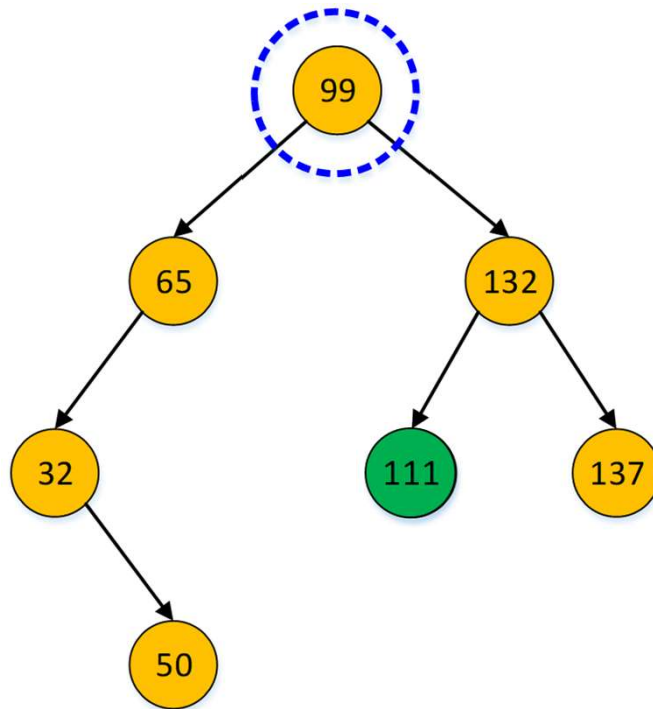
对同一批数据元素，可以有很多二叉树



内容

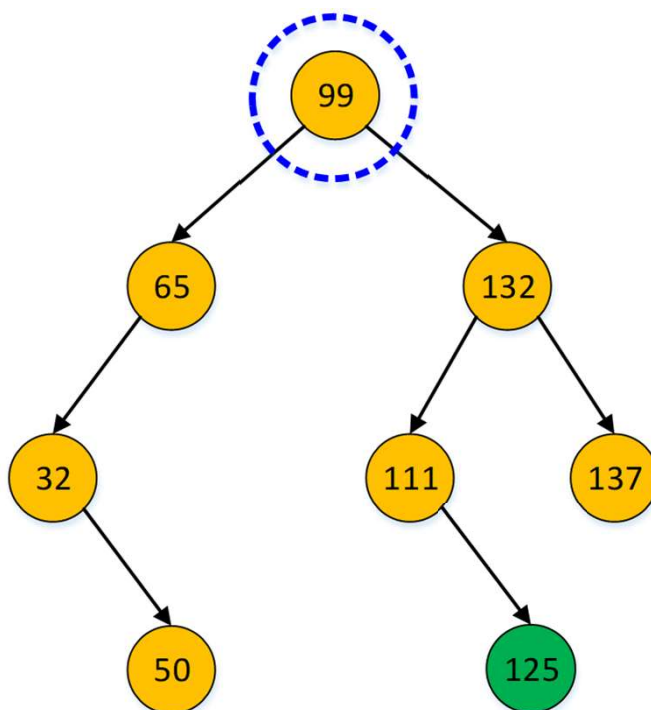
- 为什么使用二叉查找树
- 什么是二叉查找树Binary Search Tree
- 怎么动态维护二叉查找树：
查找/插入/删除

查找算法(111)



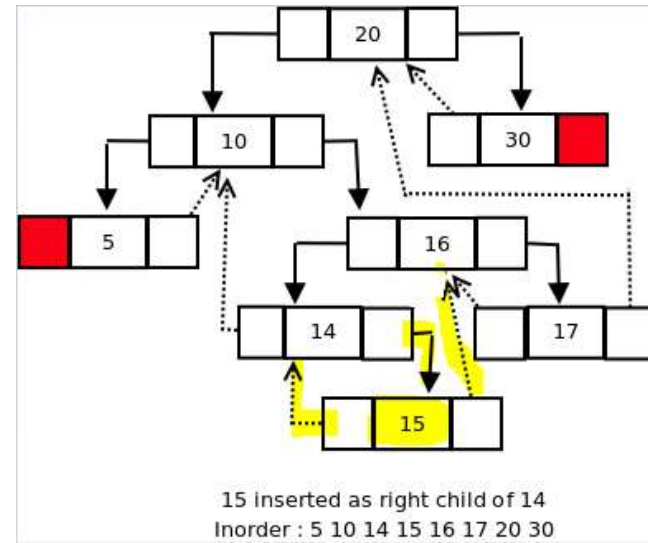
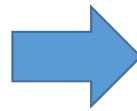
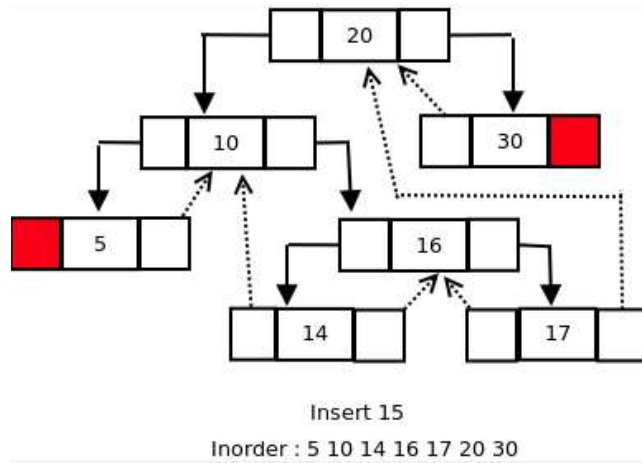
因为对查找特别友好，所以称为二叉查找树

插入算法(125)



先查找到叶节点，然后根据大小插入为左/右子节点

*Threaded BST插入算法



先按正常BST插入新元素

然后按插入节点是左子节点/右子结点，调节相应的thread指针

Randomly built BST

- 对给定的一组数，
 - 随机选出一个数 x ，利用插入算法，把 x 插入初始为空的BST
 - 继续上面步骤，直到数组变空
-
- 树的平均高度是 $O(\log(n))$
 - 对于Threaded BST，只需要改用Threaded BST插入算法

删除算法

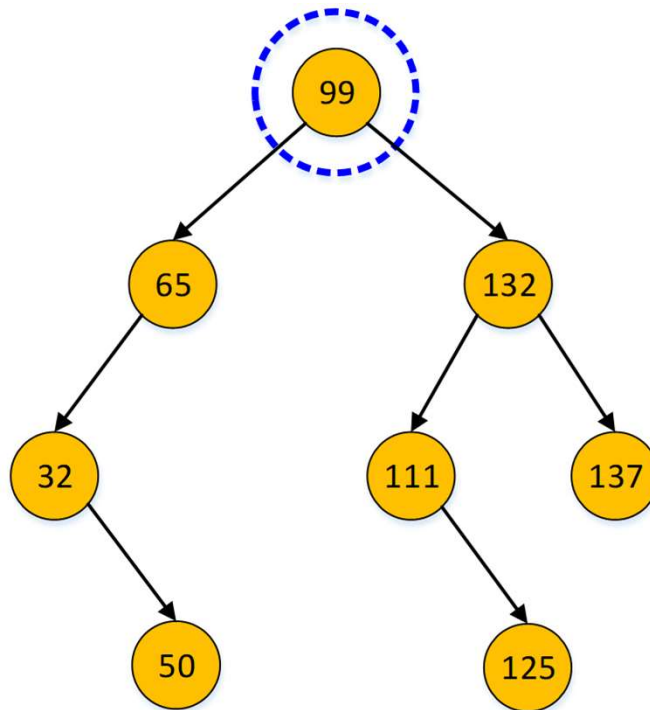
比插入算法要复杂:

Case 1: 被删节点有0个子节点

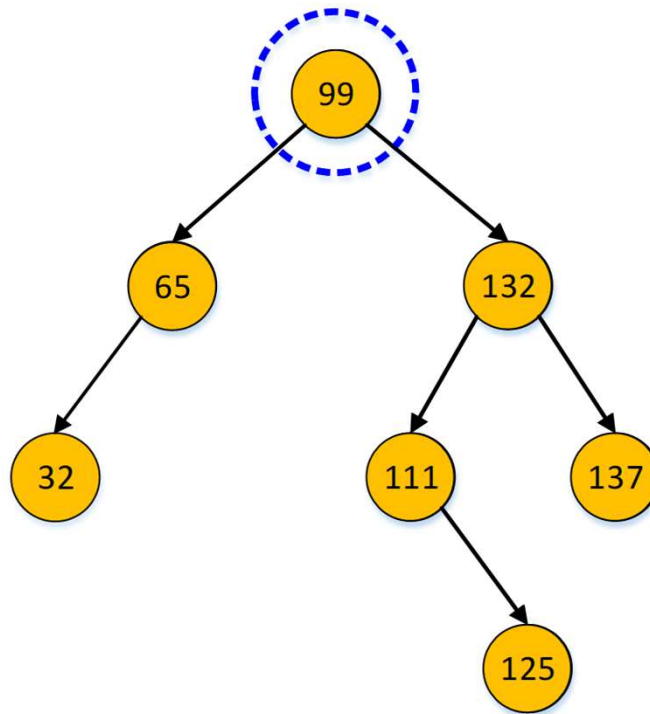
Case 2: 被删节点有1个子节点

Case 3: 被删节点有2个子节点

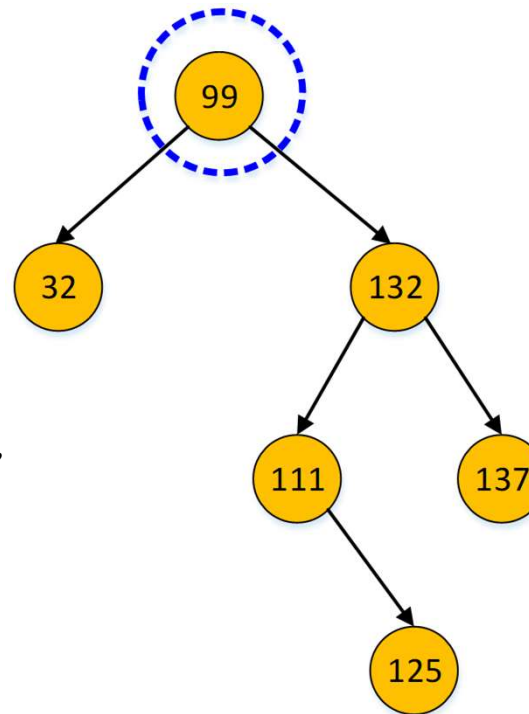
Case 1: Delete Key (50)



Case 2: Delete Key (65)

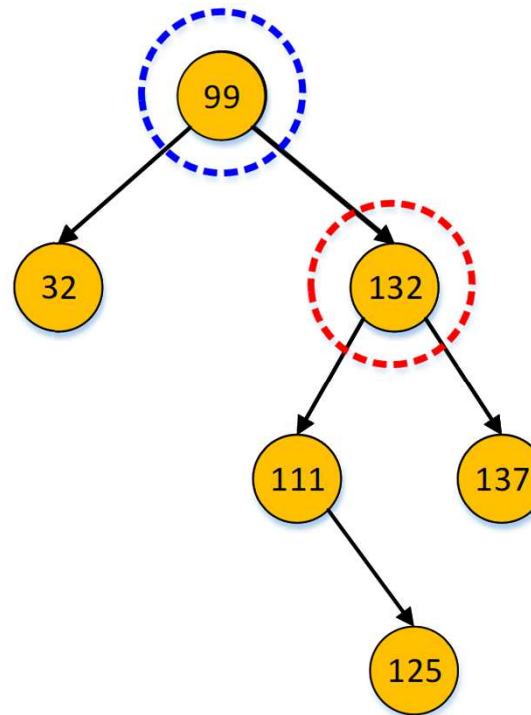


Case 3: Delete Key (99)



如何直接删除99，会剩余两个单独子树，
不好处理

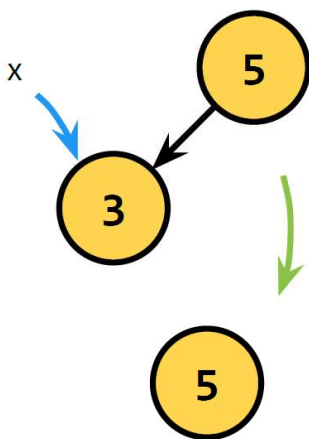
Case 3: Delete Key (99) .



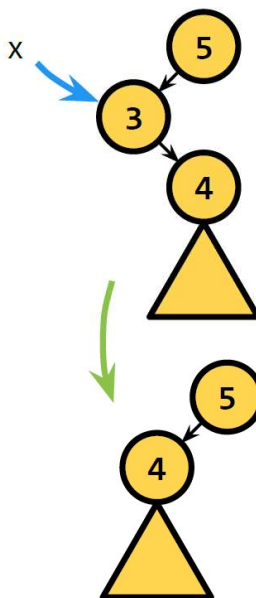
- 先查找99的直接后继111
- 然后用111替换99，111的新位置合法
- 最后删除原来的111（最多1个子节点）

删除算法总结

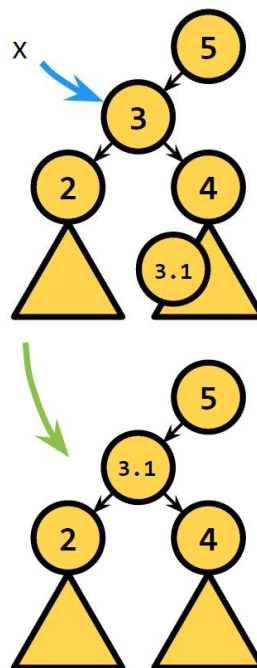
Case 1: x是叶节点,
直接删除



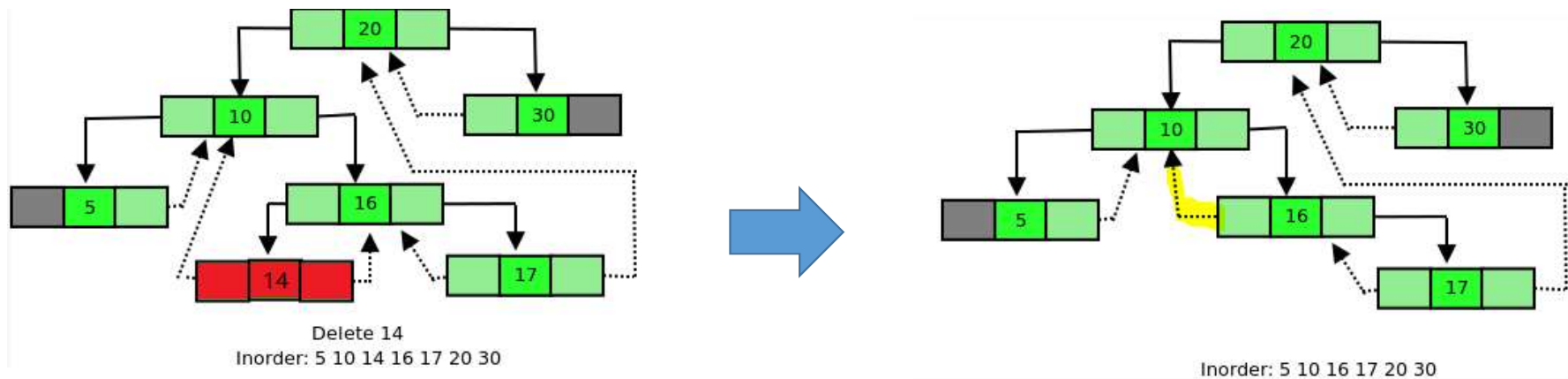
Case 2: x有一个子节点,
直接把子节点的子节点
接到父节点



Case 3: x有两个子节点,
用x的直接后继代替, 然
后删除后继原来的节点
(最多有一个子节点)



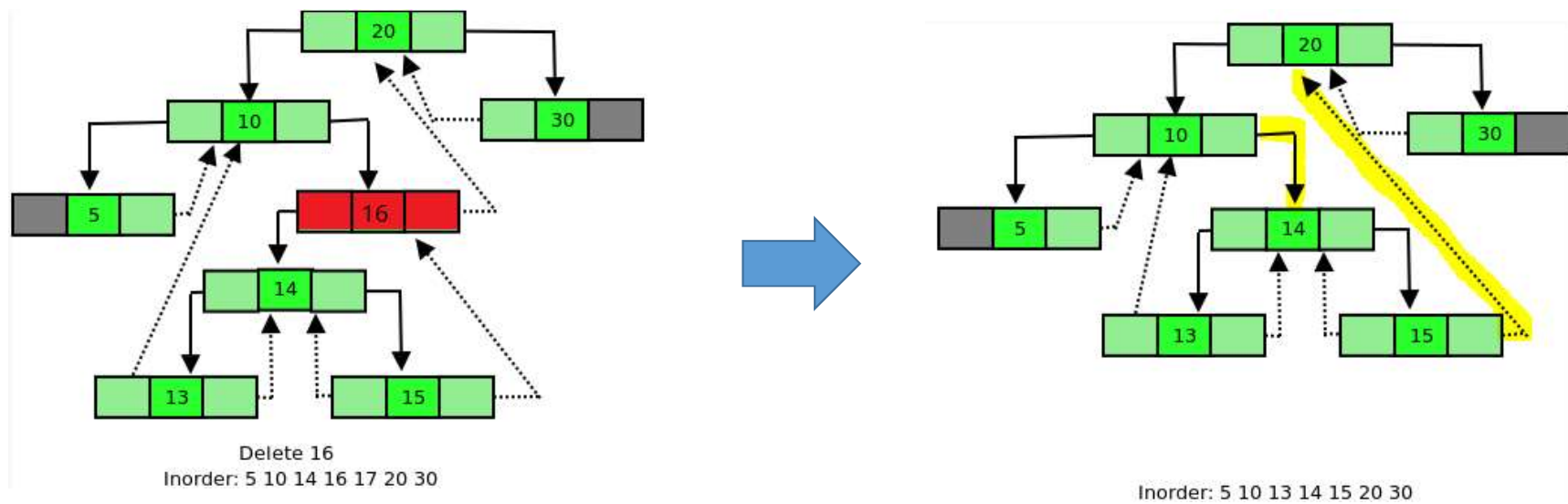
*Threaded BST删除算法 (1)



先按正常BST删除叶结点

然后按删除节点是左子节点/右子结点，调节父节点相应的thread指针

*Threaded BST删除算法 (2)



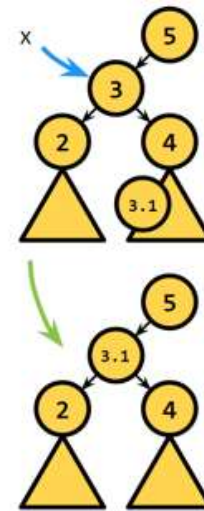
先按正常BST删除结点16，它只有左子节点

然后按删除节点是左子节点/右子结点，调节父节点相应的指针以及14子树中的rightmost结点的thread指针

*Threaded BST删除算法 (3)

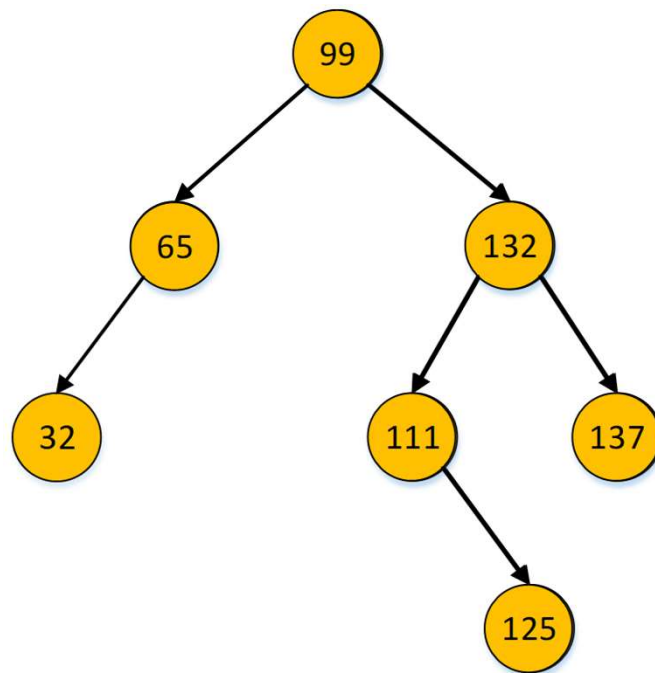
两个子节点的情况：

- 按正常**BST**删除算法找到被删节点**3**的直接后继**3.1**
- 把**3.1**复制到节点**3**（不用复制指针）
- 然后删除原来的**3.1**节点（最多一个子节点）



查找/插入/删除时间复杂度

- 查找是主要的操作
- 插入/删除都是先查找，然后再花费 $O(1)$ 额外操作.

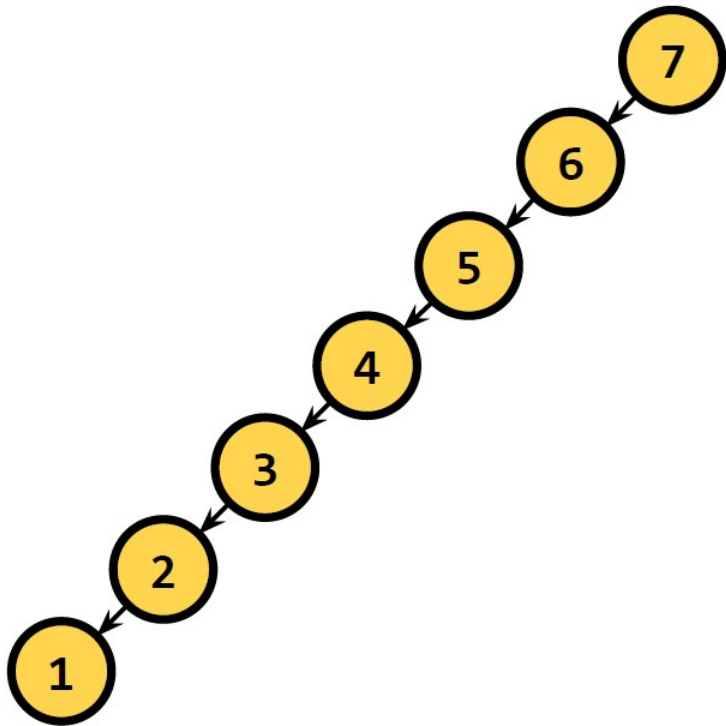


Time = $O(\text{depth of tree})$
 $= O(\log n) ???$

问题是.....

- 如果**BST**里的元素的插入顺序是**有序**的
- 你会得到一个什么样子的**BST**？

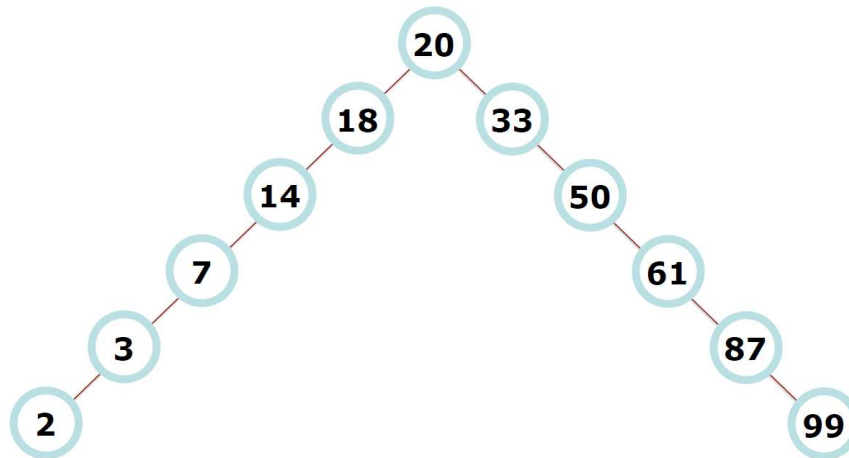
问题是.....



- 这还是一个有效的**BST**
- 但是树的高度是 n , 不再是 $O(\log(n))$
- 其实退化成了链表

维持 $O(\log n)$ 复杂度的思路

- 保持二叉查找树的左右节点个数平衡
- 不过不是下面这种平衡，即只有根节点是平衡的



维持 $O(\log n)$ 复杂度的思路

- 另一方面，如果要求所有节点的左右子树完全平衡
- 却是太难达到要求，只有满的完全的二叉树才是
- 思路是我们只需要维持大体平衡，
- 希望所有节点的左右子树的节点个数只差常数倍

