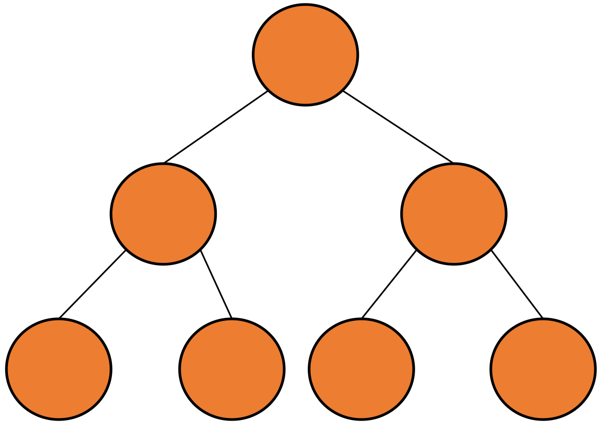
湖南大學
HUNAN UNIVERSITY

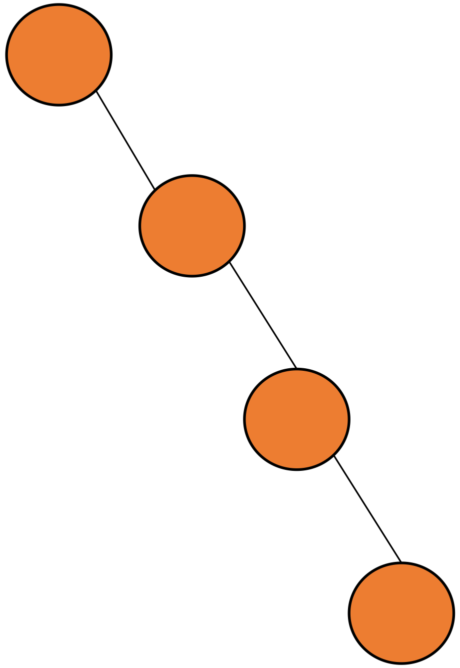
第五章 树

湖南大学信息科学与工程学院

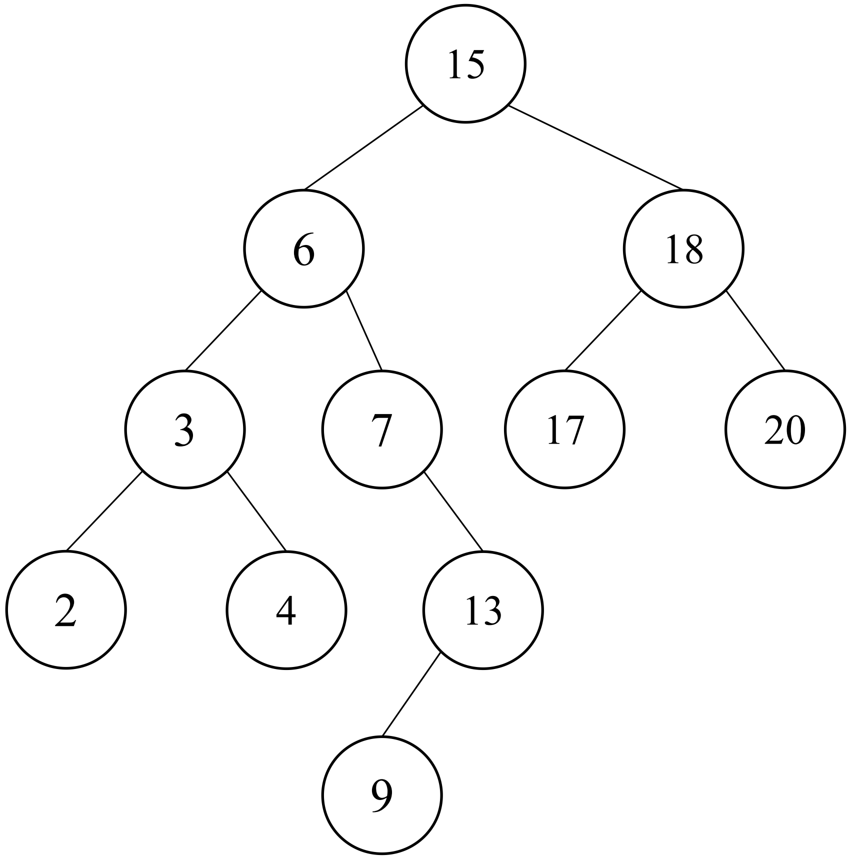
二叉搜索树

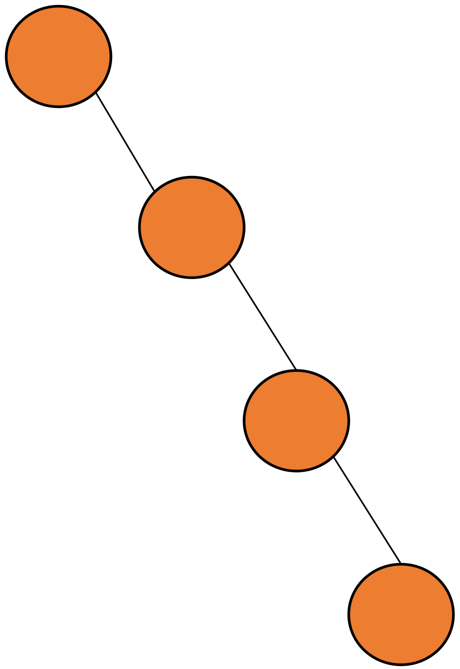


一颗好的二叉搜索树，
高度为 log n



不好的二叉搜索树高度可
能为 n

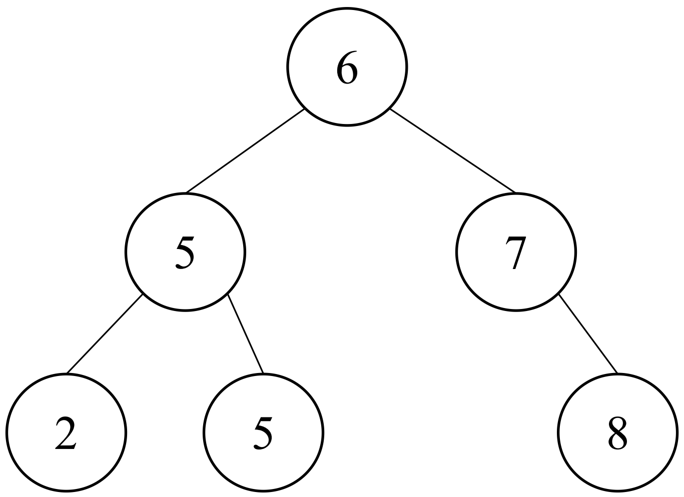

15
6
18
3
7
17
20
2
4
13
9



二叉搜索树

二叉搜索树的每个结点都包含一个关键字key，同时包含属性left、right和p，它们分别指向结点的左孩子、右孩子和双亲。如果某个孩子结点的父结点不存在，则相应属性值为NIL。根结点是树中唯一父指针为NIL的结点。

二叉搜索树



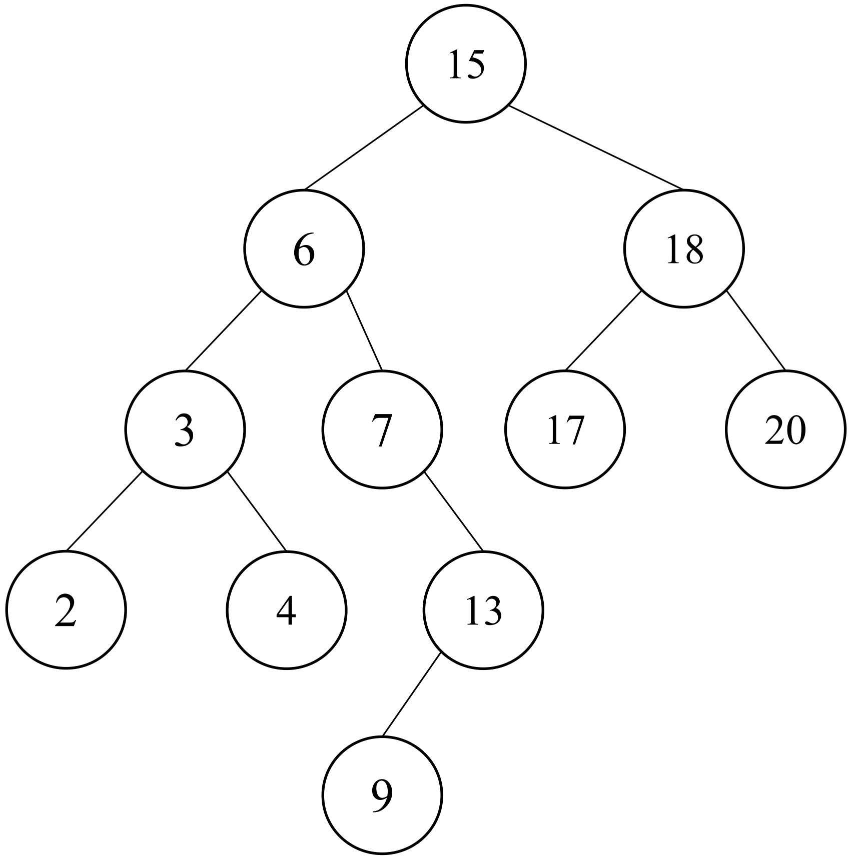
对于任何结点x，其左子树中的关键字最大不超过x.key，其右子树中的关键字最小不低于x.key。上图展示了一颗包含6个结点、高度为2的二叉搜索树。

查询二叉搜索树

在一颗二叉搜索树中查找一个具有给定关键字的结点，输入一个指向树根的指针和一个关键字k，如果这个结点存在，TREE-SEARCH返回一个指向关键字为k的结点的指针，否则返回NIL。

```
TREE-SEARCH(x,k)
If x ==NIL or k == x.key
    return x
If k < x.key
    return TREE-SEARCH(x.left,k)
else return TREE-SEARCH(x.right,k)
```

查询二叉搜索树



查询关键字为13的结点，从树根开始，13<15，故沿着left指针查找左子树；13>6，故沿着right指针查找右子树；13>7，故沿着right指针查找右子树；最终找到13。

查询二叉搜索树

```
ITERATIVETREE-SEARCH(x,k)
while x ≠ NIL ,and k ≠ x.key
If k< x.key
        x = x.left
 else  x = x.right
 return x
```

我们可以采用while循环来展开递归，用一种迭代方式重写这个过程，对于大多数计算机，迭代版本的效率要高得多。

最大关键字元素

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

二叉搜索树性质保证了TREE-MAXIMUM是正确的。如果结点没有右子树，那么由于x左子树中的每个关键字都至少小于或等于x.key，则以x为根的子树中的最大关键字是x.key。如果结点有右子树，那么由于其左子树中没有关键字大于x.key，且在右子树中的每个关键字不小于x.key，则以x为根的子树中的最大关键字一定在以x.right为根的子树中。

最小关键字元素

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

二叉搜索树性质保证了TREE-MINIMUM是正确的。如果结点没有左子树，那么由于x右子树中的每个关键字都至少大于或等于x.key，则以x为根的子树中的最小关键字是x.key。如果结点有左子树，那么由于其右子树中没有关键字小于x.key，且在左子树中的每个关键字不大于x.key，则以x为根的子树中的最小关键字一定在以x.left为根的子树中。

查询二叉搜索树

这两个过程在一颗高度为h的树上均能在O(h)的时间内执行完，因为与TREE-SEARCH一样，它们所遇到的结点均形成了一条从树根向下的简单路径。

后继和前驱

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x,right)
y = x.p
while y ≠NIL and x == y.right
x = y
y = y.p
return y
```

给定一颗二叉搜索树中的一个结点，有时候需要按中序遍历的次序查找它的后继。如果所有的关键字互不相同，则一个结点x的后继是大于x.key的最小关键字的结点。一颗二叉搜索树的结构允许我们通过没有任何关键字的比较来确定一个结点的后继。如果后继存在，下面的过程将返回一颗二叉搜索树中的结点x的后继；如果x是这棵树中的最大关键字，则返回NIL。

构造一颗二叉搜索树

插入10

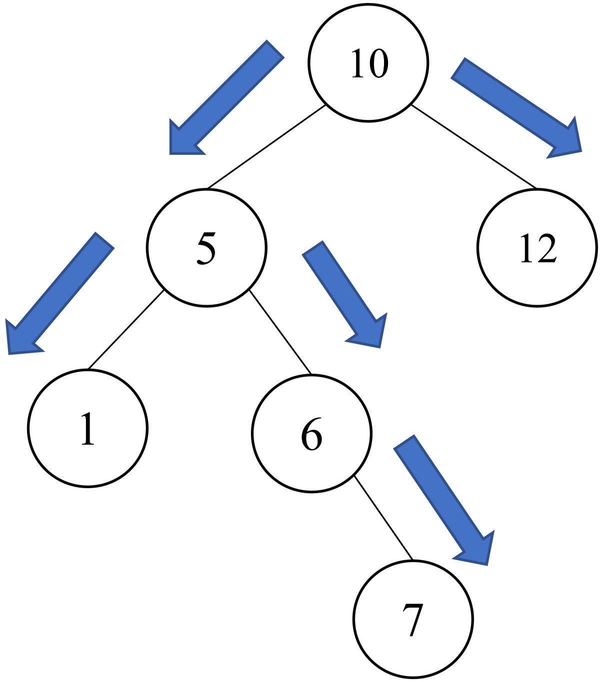
插入12

插入5

插入1

插入6

插入7

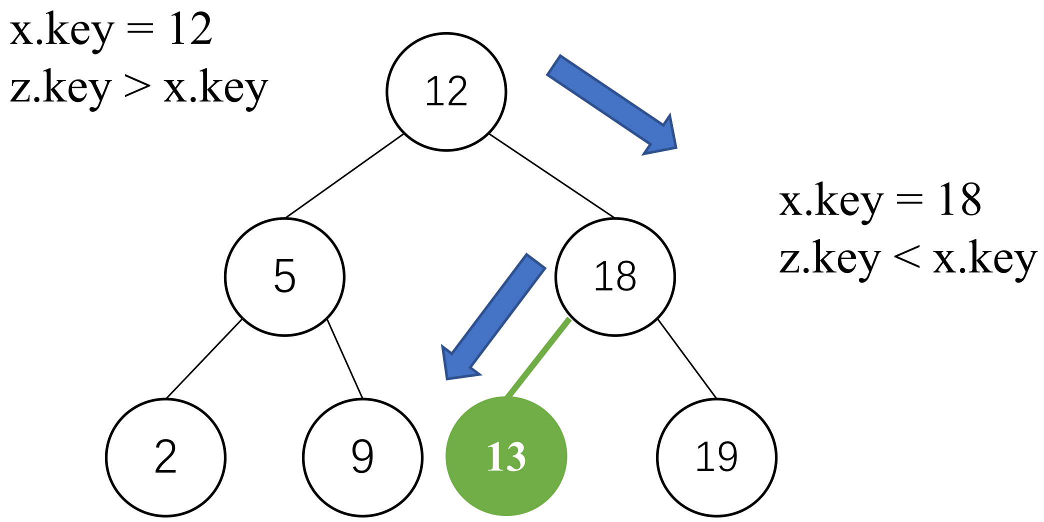


二叉搜索树的插入

```
TREE-SEARCH(T,z)
y == NIL
x == T.root
While x ≠ NIL
    y = x
    If z.key < x.key
        x = x.left
    else  x = x.right
z.p =y
If y == NIL
    T.root = z        //tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

要将一个新值v插入到一颗二叉搜索树T中，需要调用过程TREE-INSERT。该过程以结点z作为输入，其中z.key = v, z.left = NIL, z.right = NIL。这个过程要修改T和z的某些属性，来把z插入到树中的相应位置上。

二叉搜索树的插入



插入关键字为13的结点z,首先与根结点12进行比较，13>12，转向右子树，13<18，转向左子树，左子树为空，则插入结点13。

从一棵二叉搜索树T中删除一个结点z

从一棵二叉搜索树T中删除一个结点z的整个策略分为三种基本情况(如下所述)，但只有第三种情况有点棘手。

- 如果z没有孩子结点，那么只是简单地将它删除，并修改它的父结点，用 NIL 作为孩子来替换z。

从一棵二叉搜索树T中删除一个结点z

从一棵二叉搜索树T中删除一个结点z

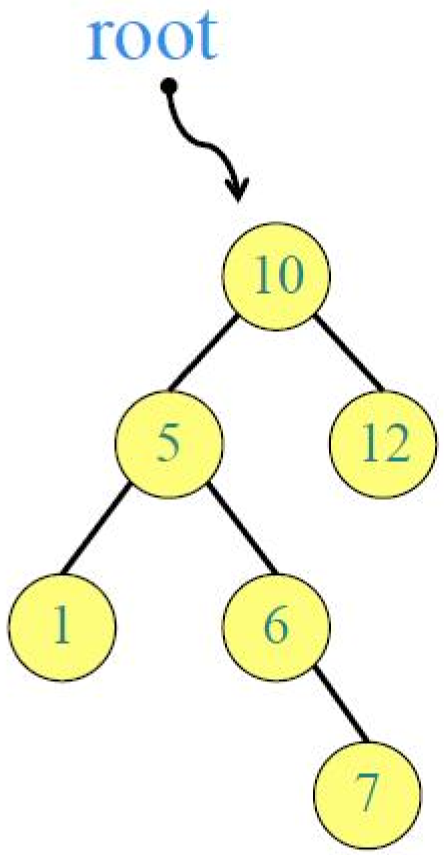
- 如果z只有一个孩子，那么将这个孩子提升到树中z的位置上，并修改z的父结点，用z的孩子來替换z。
- 如果z有两个孩子，那么找z的后继y(一定在z的右子树中），并让y占据树中z的位置。z的原来右子树部分成为y的新的右子树，并且z的左子树成为y的新的左子树。

从一棵二叉搜索树T中删除一个结点z

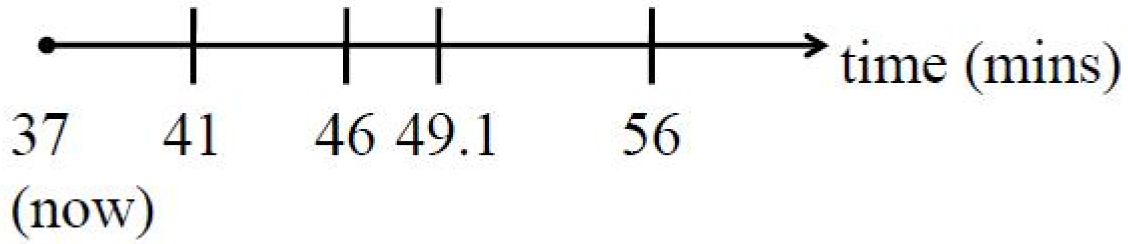
- 如果z有两个孩子，那么找z的后继y(一定在z的右子树中），并让y占据树中z的位置。z的原来右子树部分成为y的新的右子树，并且z的左子树成为y的新的左子树。

Growing BSTs 树的成长

- Insert 10
- Insert 12
- Insert 5
- Insert 1
- Insert 6
- Insert 7



BST as a data structure



- Operations:

数据结构的操作

  - insert(k): inserts key k

插入：插入给定键值k的节点

  - search(k): finds the node containing key k (if it exists)

搜索：搜索包含键值k的节点（如果存在）

  - next-larger(x): finds the next element after element x

下一个最大节点：找出当前节点x的下一个最大节点

  - findmin(x): finds the minimum of the tree rooted at x

找最小节点：找出当前节点x为根节点的子树的最小节点

  - delete(x): deletes node x

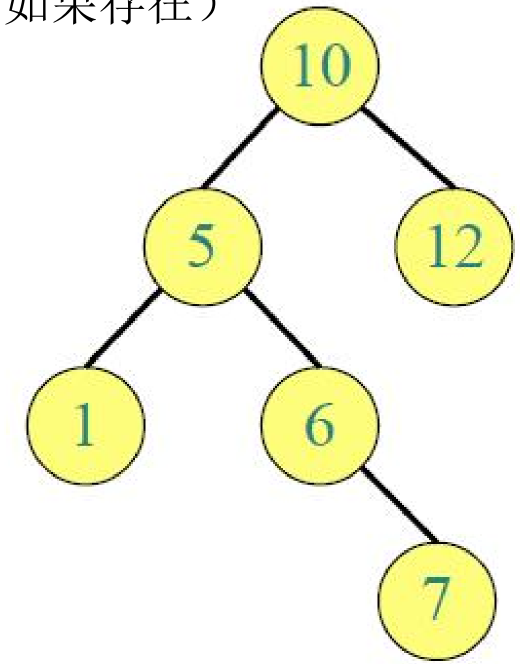
删除节点x

Search

Search(k): 搜索：搜索包含键值k的节点（如果存在）

- Recurse left or right until you find k, or get NIL

  递归查找左节点或由节点，
  直到找到k或者不在列表中为止



Search(7)

Search(8)

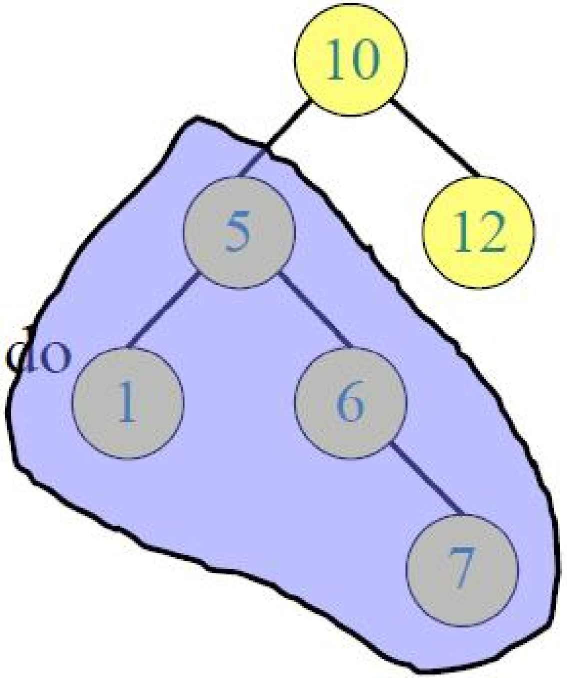
Next-larger

next-larger(x):

下一个最大节点：找出当前节点x的下一个最大节点

- If right[x] $\neq$ NIL then
  return minimum(right[x])
- Otherwise
  y $\leftarrow$ p[x]
  While y$\neq$NIL and x=right[y] do
  - x $\leftarrow$ y
  - y $\leftarrow$ p[y]

  Return y



next-larger(5)

next-larger(7)

Minimum

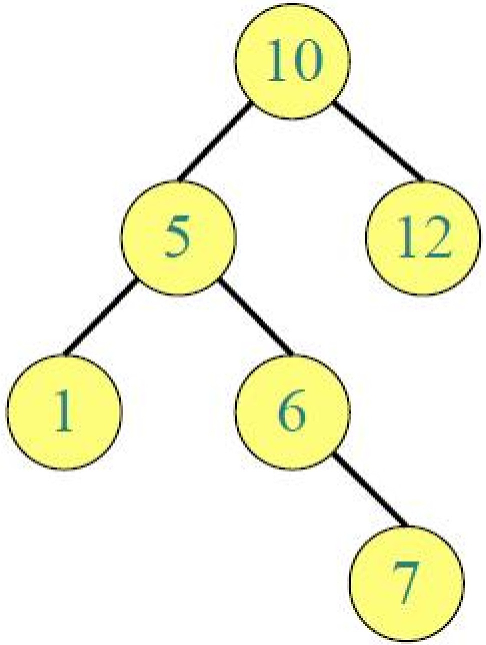
找最小节点：找出当前节点x为根节点
的子树的最小节点

Minimum(x)

- While left[x]≠NIL do

  x ← left[x]

- Return x



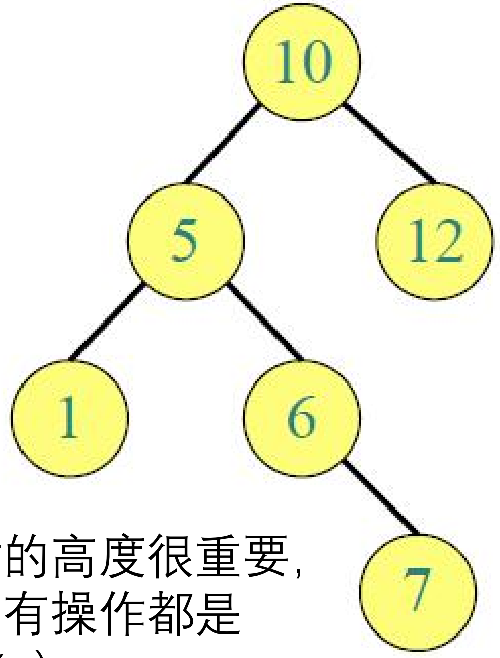
minimum(5)

Analysis

- We have seen insertion, search, minimum, etc.
- How much time does any of this take ?
- Worst case: O(height)

  => height really important
- After we insert n elements, what is the worst possible BST height ?

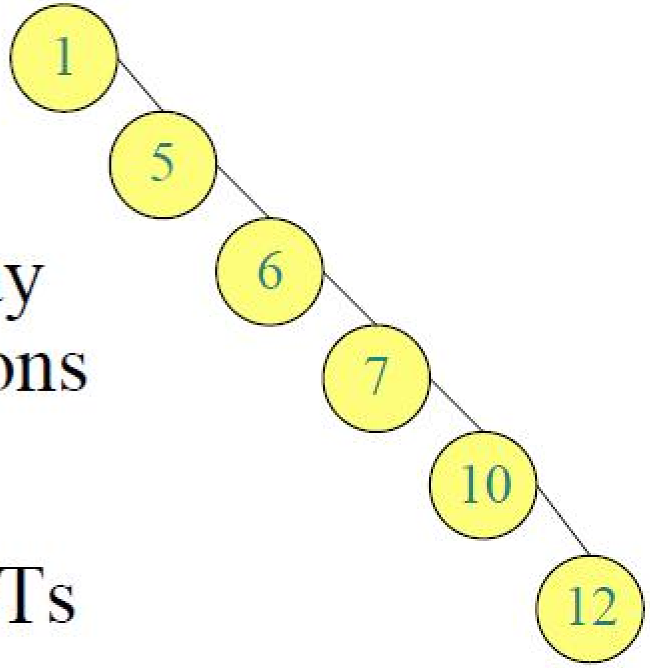
当插入完n个元素后，可能的最差二叉搜索树的高度是多少？



树的高度很重要，所有操作都是O(n)

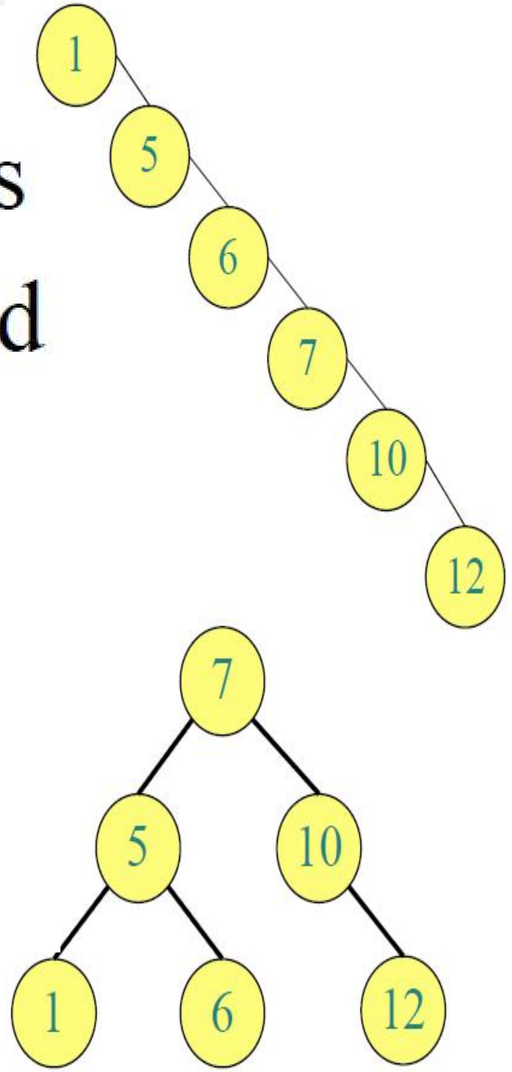
Analysis

- n-1
- So, still O(n) for the runway reservation system operations
- Next lecture: **balanced** BSTs
- **Readings: CLRS 13.1-2**



Lecture Overvie

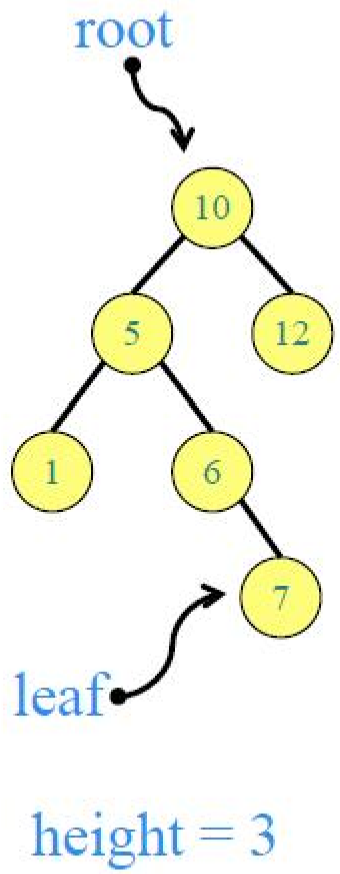
- Review: Binary Search Trees
- Importance of being balanced
- Balanced BSTs
  - AVL trees
    - definition
    - rotations, insert



Binary Search Trees (BSTs)

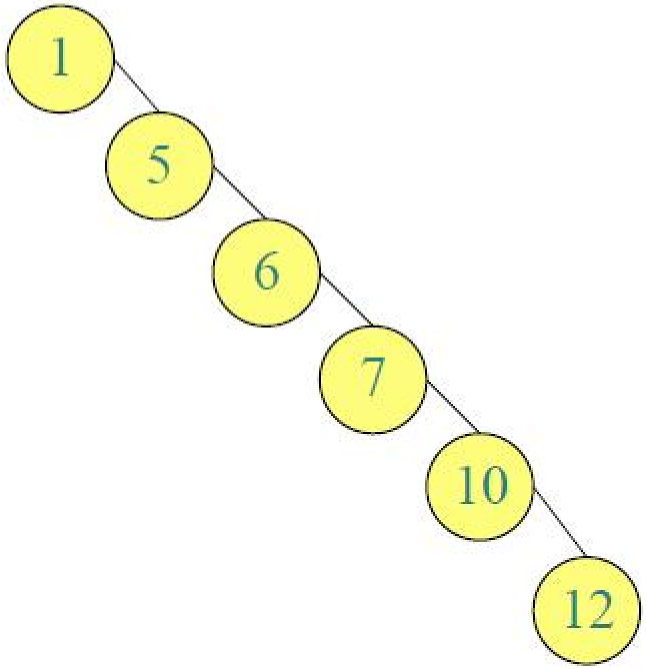
- Each node x has:
  - key[x]
  - Pointers: left[x], right[x], p[x]

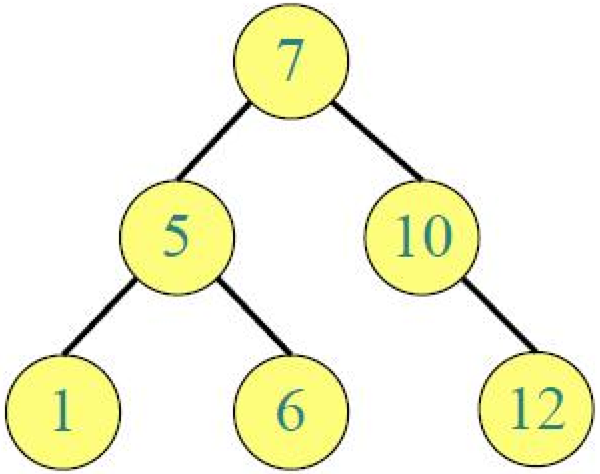
- Property: for any node x:
  - For all nodes y in the left subtree of x:
    $$key[y] \leq key[x]$$
  - For all nodes y in the right subtree of x:
    $$key[y] \geq key[x]$$



The importance of being balanced

for n nodes:　　平衡树的高度非常的关键



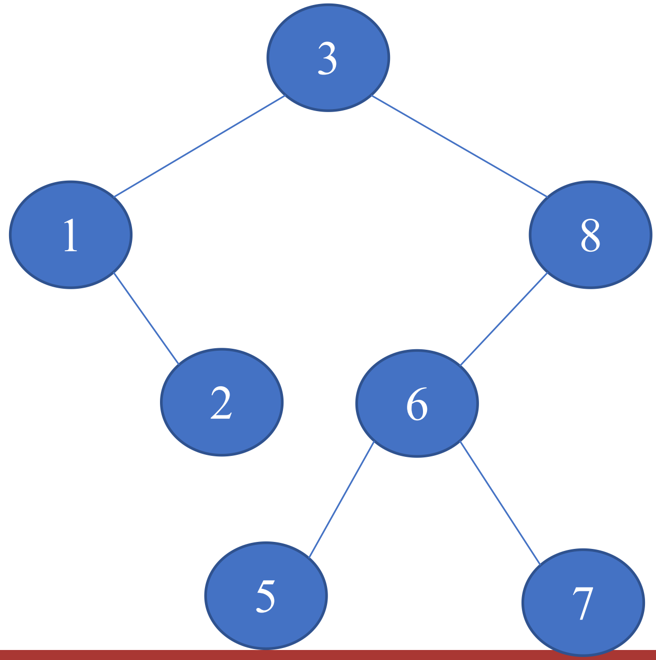


$h = \Theta(\log n)$　　　　$h = \Theta(n)$

BST Sort

- Given an array A, build a BST for A
- Do an inorder tree walk(中序遍历)

3	1	8	2	6	7	5



Time Complexity

- Given an array A, build a BST for A ($\Omega(n\log n)$)
- Do an inorder tree walk ($O(n)$)

Relation to Quick Sort

- Comparisons in BST Sort are the same to comparisons in Quick Sort

3	1	8	2	6	7	5

- We can randomize BST Sort
- Randomized BST Sort have the same time complexity to randomized Quick Sort

Balanced BST Strategy

平衡的二叉搜索树的策略

给所有的节点增加一些额外的信息

- Augment every node with some data
- Define a local invariant on data

给每个本地节点的信息定义一个不变式

- Show (prove) that invariant guarantees $\Theta(\log n)$ height

证明每个本地节点的不变式可以保证树的高度为$\Theta(\log n)$

- Design algorithms to maintain data and the invariant

设计算法来维持额外的节点信息和不变式

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

湖南大學
HUNAN UNIVERSITY

HNU

谢谢观赏

实 事 求 是　敢 为 人 先