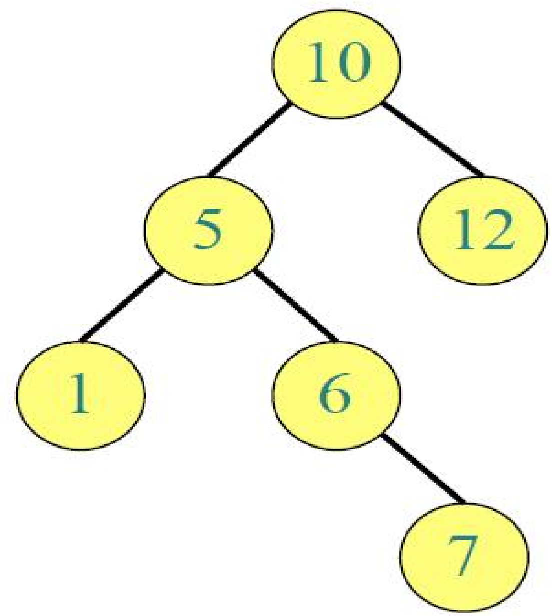
# 第四章 树

湖南大学信息科学与工程学院

# Overview

- Runway reservation system （机场跑道预定系统）：
  - Definition
  - How to solve with lists（由一系列的飞机起飞时间节点组成）
- Binary Search Trees
  - Operations
- Readings: CLRS 10, 12.1-3

http://izismile.com/tags/Gibraltar/

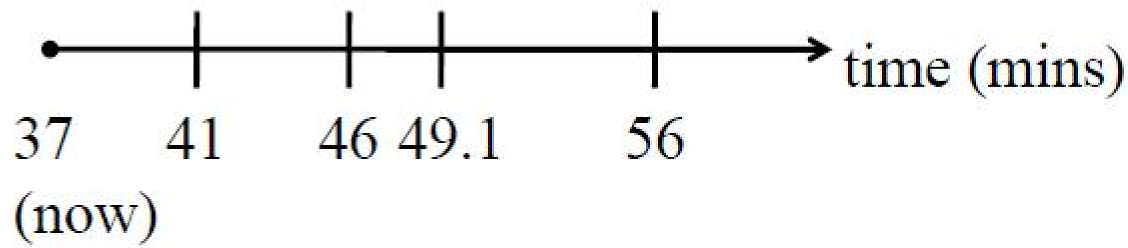


# Runway reservation system

- Problem definition:
  - Single (busy) runway 单一跑道
  - Reservations for landings 预定起飞时间
    - maintain a set of future landing times 维护一个集合的起飞时间节点
    - a new request to land at time t 给定一个新的起飞时间节点申请t
    - add t to the set if no other landings are scheduled within < 3 minutes from t 把t加入到集合中：新的时间节点t与其他所有时间节点的间隔小于3分钟
    - when a plane lands, removed from the set 当飞机起飞，则把它的时间节点t从集合中删除

# Runway reservation system

- Example




  - R = (41, 46, 49.1, 56)　时间节点集合
  - requests for time:　新时间节点请求
    - 44 => reject (46 in R)　拒绝
    - 53 => ok　允许
    - 20 => not allowed (already past)　不允许，超过边界
- Ideas for efficient implementation ?

# Some options

- Keep R as an unsorted list
  - Bad: takes linear time to search for collisions
  - Good: can insert t in O(1) time

缺点：需要线性时间来找冲突

优点：插入为常量时间

- Keep R as a sorted array (resort after each insertion)
  - Bad: takes "a lot of" time to insert elements
  - Good: 3 minute check can be done in O(log n) time:
    - Using binary search, find* the smallest i such that R[i]>=t (next larger element)
    - Compare t to R[i] and R[i-1]

缺点：需要很多时间来插入时间节点

下一个最大元素

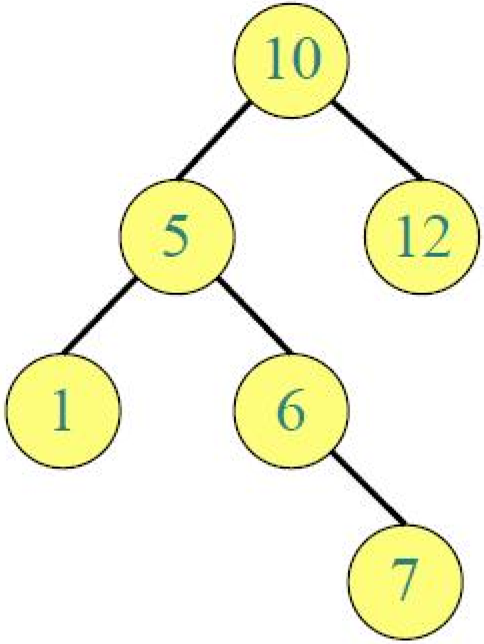
优点：检查3分钟时间冲突可以在对数时间内解决

**Need: *fast* insertion into *sorted* list (sort of)**

数组插入效率低，需要更快的插入算法及数据结构

# Binary Search Tree (BSTs)

- Each node x has:
  - key[x]　键值
  - Pointers:　节点指针、引用或索引
    - left[x]　左节点
    - right[x]　右节点
    - p[x]　父节点



# Binary Search Tree (BSTs)

- 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:

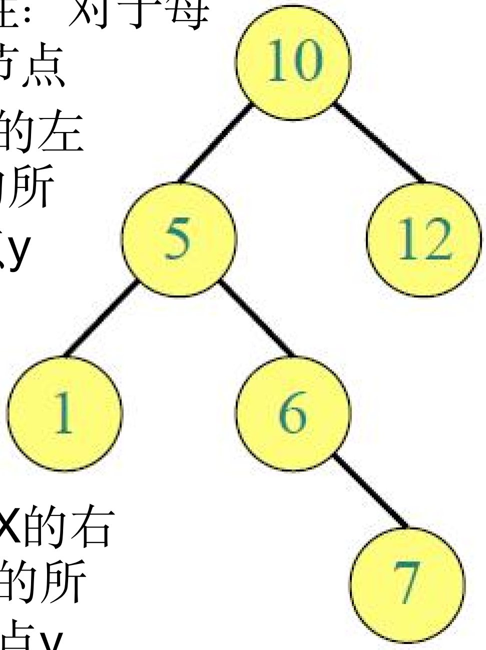
    $$key[y] \geq key[x]$$

- How are BSTs made ?

属性：对于每个节点

对于X的左子树的所有节点y

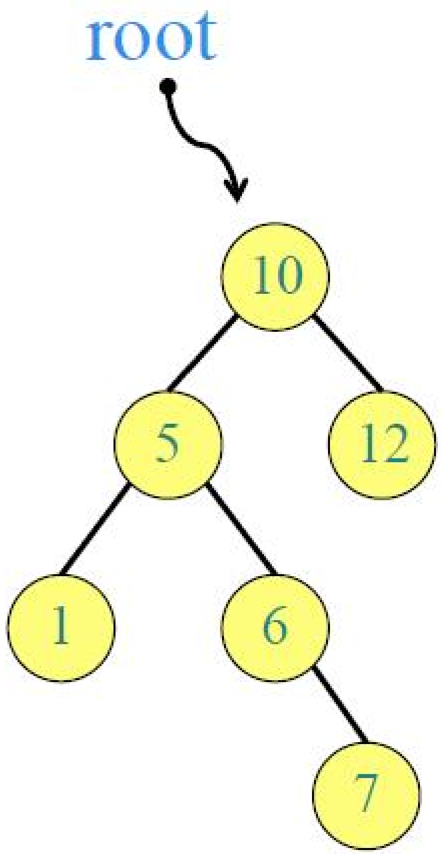
对于X的右子树的所有节点y



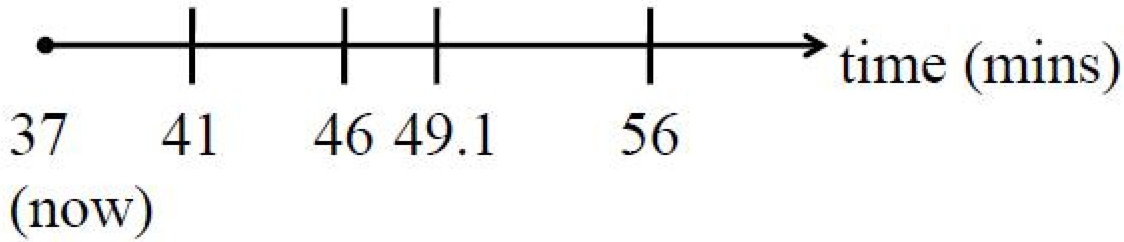

怎么构建二叉搜索树?

# Growing BSTs
# 树的成长

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



# BST as a data structure



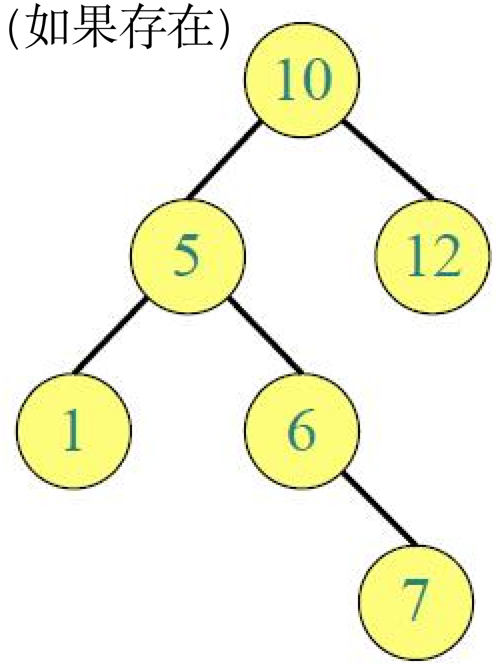

- Operations: 数据结构的操作
  - insert(k): inserts key k 插入：插入给定键值k的节点
  - search(k): finds the node containing key k (if it exists) 搜索：搜索包含键值k的节点（如果存在）
  - next-larger(x): finds the next element after element x 下一个最大节点：找出当前节点x的下一个最大节点
  - minimum(x): finds the minimum of the tree rooted at x 找最小节点：找出当前节点x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)

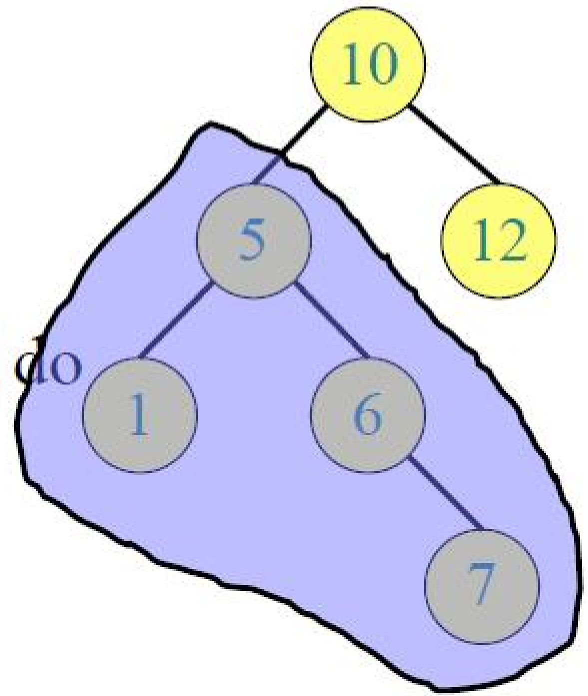
Search(8)

# 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] ≠ NIL then
  return minimum(right[x])
- Otherwise
  y ← p[x]
  While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]

  Return y




next-larger(5)

next-larger(7)

# Minimum

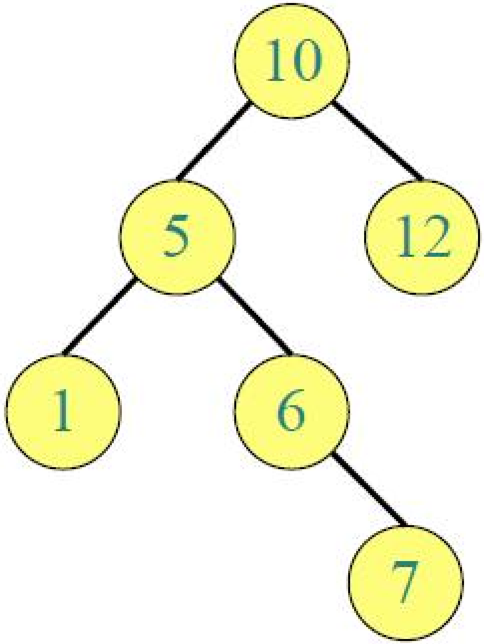
找最小节点：找出当前节点x为根节点
的子树的最小节点

Minimum( x )

- While left[x]≠NIL do

  $x \leftarrow left[x]$

- Return x

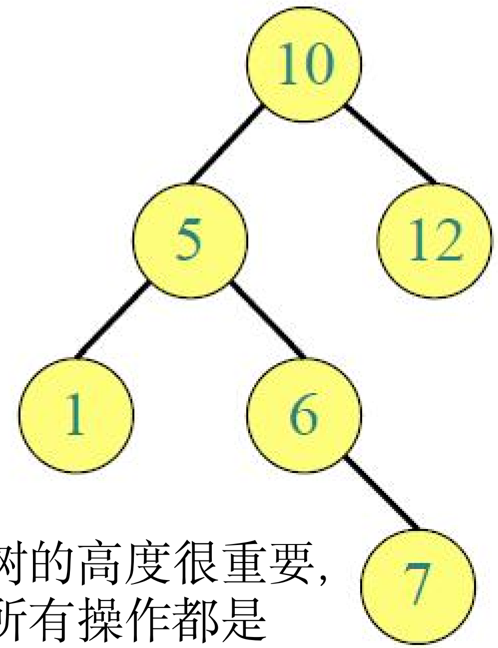



minimum( 5)

# 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 ?

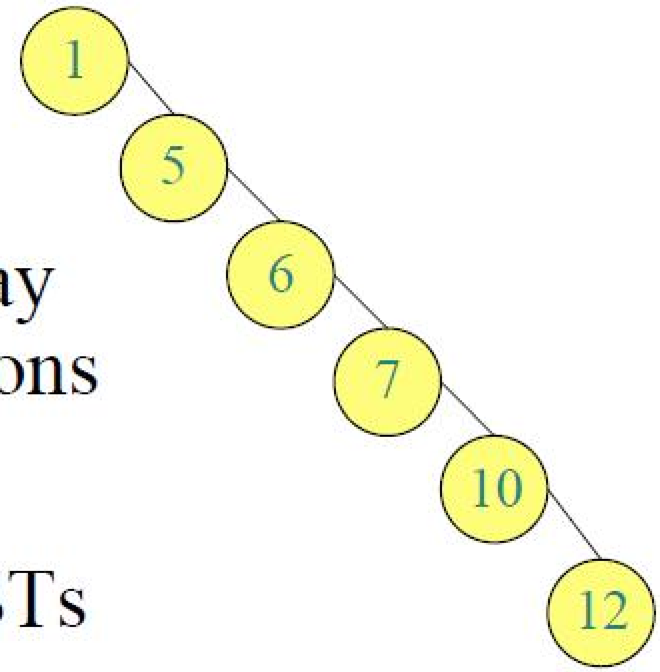



树的高度很重要，所有操作都是O(n)

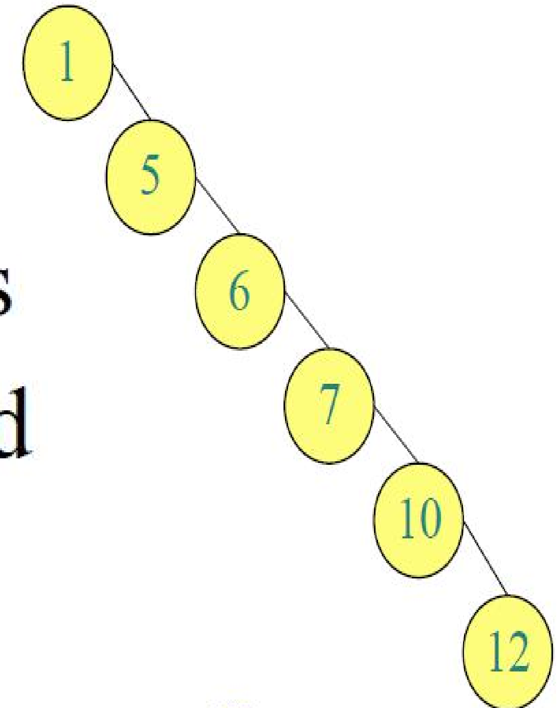
当插入完n个元素后，可能的最差二叉搜索树的高度是多少?

# Analysis

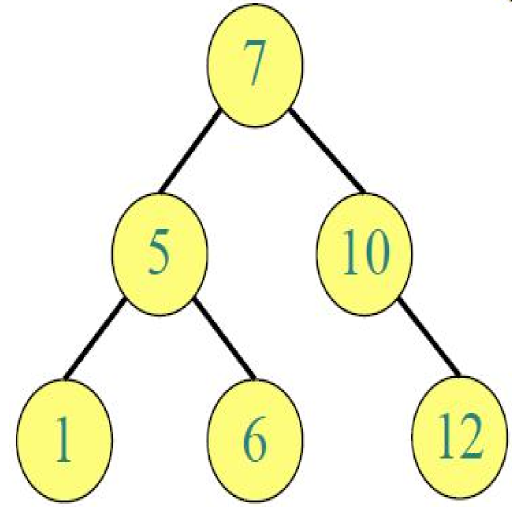
- n-1
- So, still O(n) for the runway reservation system operations
- Next lecture: **balanced** BSTs
- **Readings: CLRS 13.1-2**



# Lecture Overvie

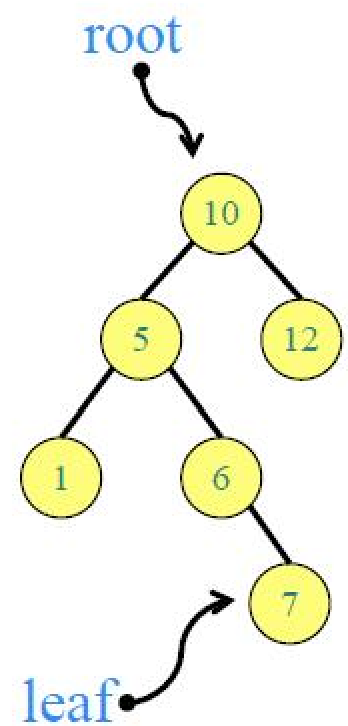



- Review: Binary Search Trees
- Importance of being balanced
- Balanced BSTs
  - AVL trees
    - definition
    - rotations, insert



# Binary Search Trees (BSTs)

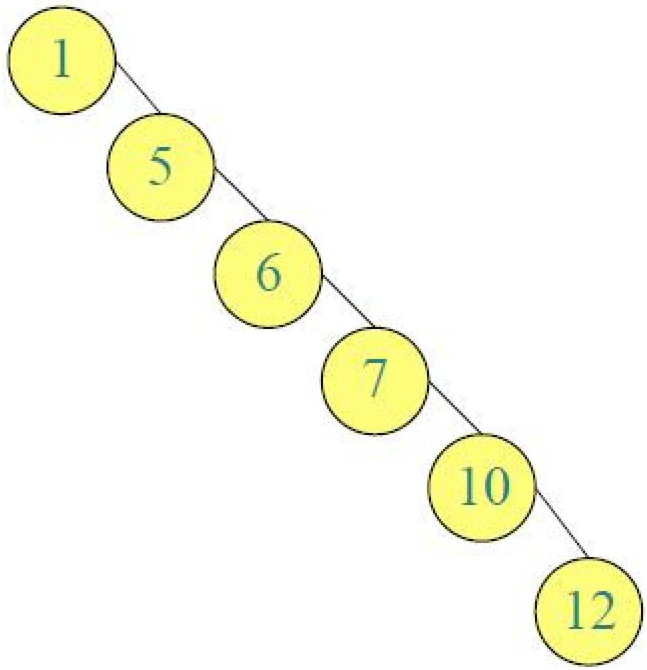
- Each node x has:
  - key[x]
  - Pointers: left[x], right[x], p[x]
- Property: for any node x:
  - For all nodes y in the left subtree of x:
    $key[y] \le key[x]$
  - For all nodes y in the right subtree of x:
    $key[y] \ge key[x]$

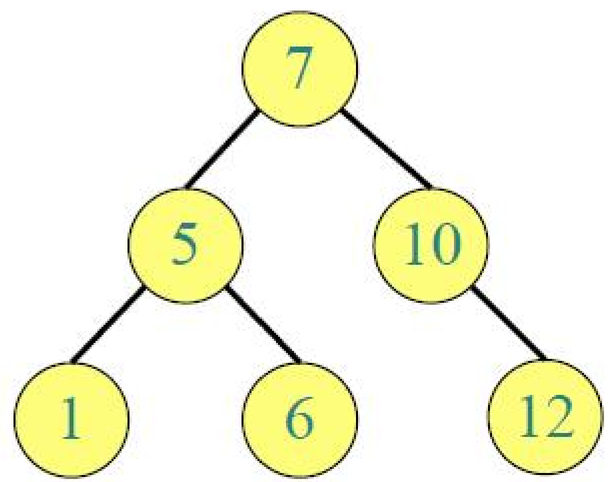



height = 3

# The importance of being balanced

for n nodes:　　平衡树的高度非常的关键



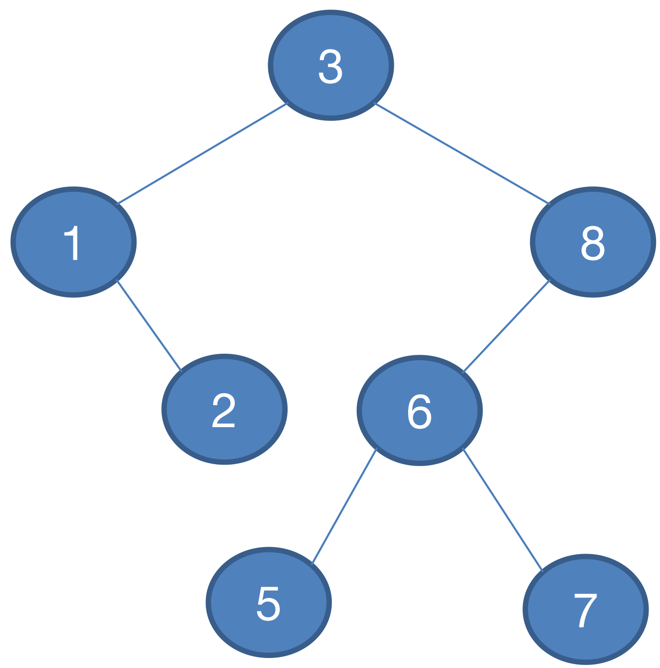




$h = \Theta(n)$　　$h = \Theta(\log n)$

# 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 ($\Omega$(nlogn))
- 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$\Theta(\log n)$

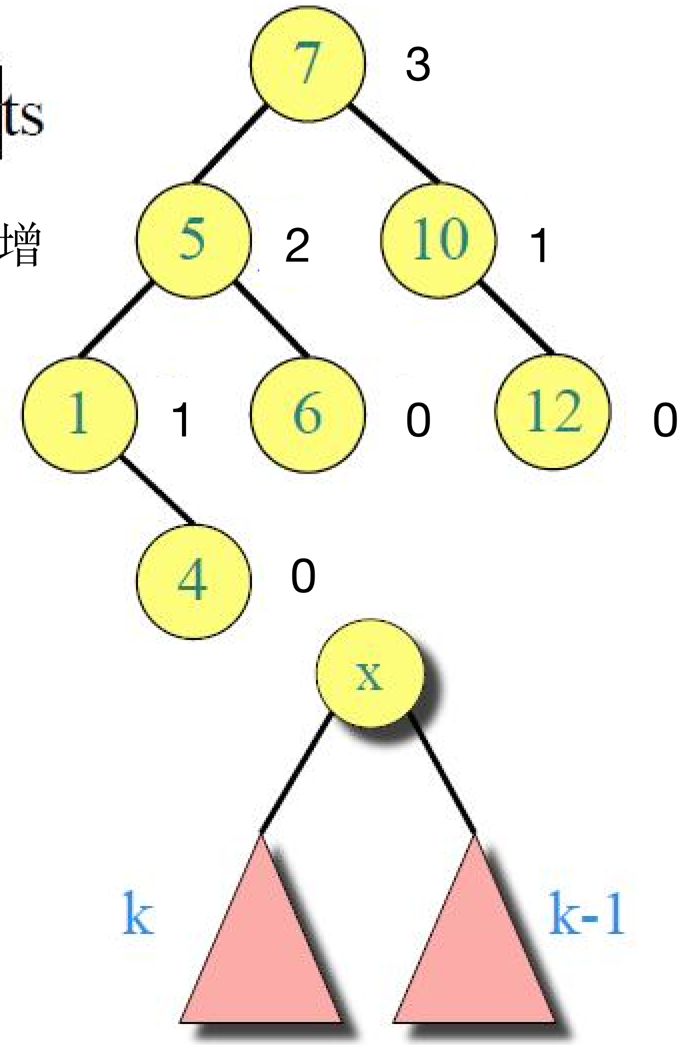
- Design algorithms to maintain data and the invariant

设计算法来维持额外的节点信息和不变式

# AVL Trees: Definition

[Adelson-Velskii and Landis'62]

- **Data**: for every node, maintain its height ("augmentation")
  信息：对于每一个节点，维护它的高度（“增加物”）
  – Leaves have height 0叶节点高度为0
  – NIL has "height" -1
    空节点高度为-1
  不变式：对于每一个节点，左子树与右子树的高度差为1
- **Invariant**: for every node x, the heights of its left child and right child differ by at most 1



# AVL trees have height Θ(*log n*)

**Invariant**: for every node $x$, the heights of its left child and right child differ by at most $1$

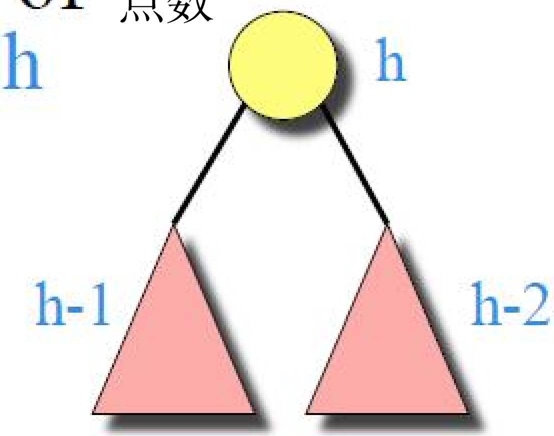
一系列高度为h的子平衡树当中的最小的节点数

- Let $n_h$ be the minimum number of nodes of an AVL tree of height $h$
- We have $n_h \geq 1 + n_{h-1} + n_{h-2}$

$$\Rightarrow n_h > 2n_{h-2}$$

$$\Rightarrow n_h > 2^{h/2}$$

$$\Rightarrow h < 2 \lg n_h$$

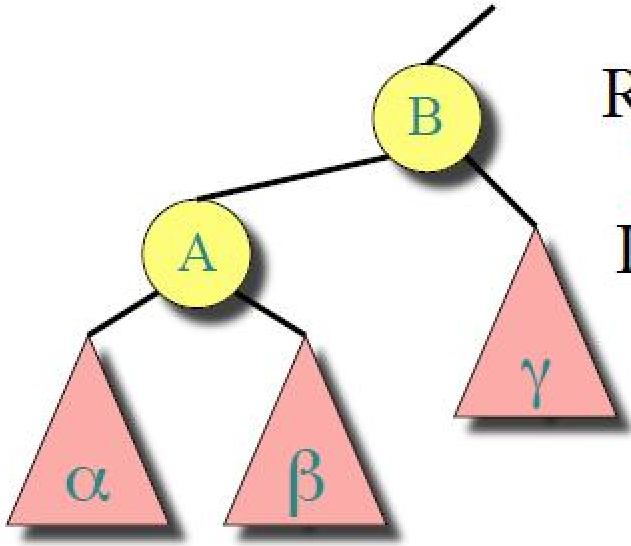



- The constant "2" can be improved

每个节点满足不变式的话，上面证明了整棵树的高度必然是Θ(log *n*),下一步是怎样来维护每个节点的不变式？

How can we maintain the invariant ?

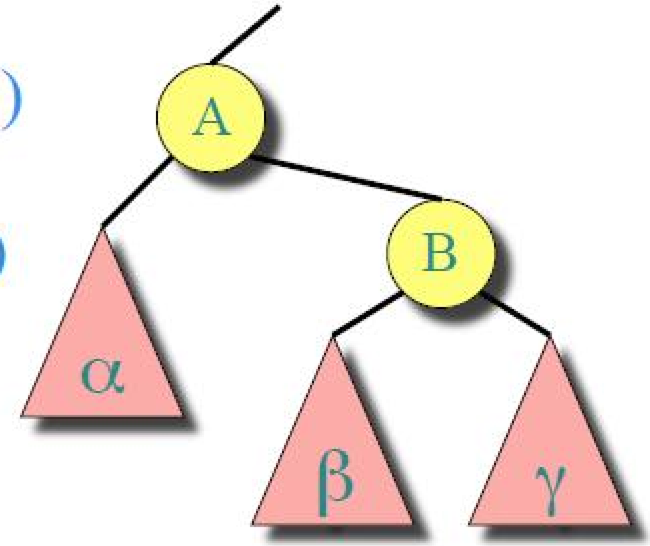
# Rotations




RIGHT-ROTATE(B)

LEFT-ROTATE(A)

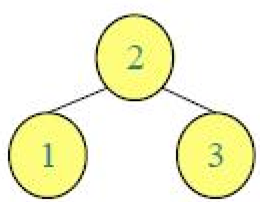
右旋转(B)
左旋转(A)



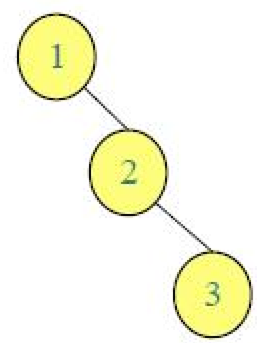

Rotations maintain the inorder ordering of keys:

- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \le A \le b \le B \le c.$

旋转前后节点的键值的排序没有变化

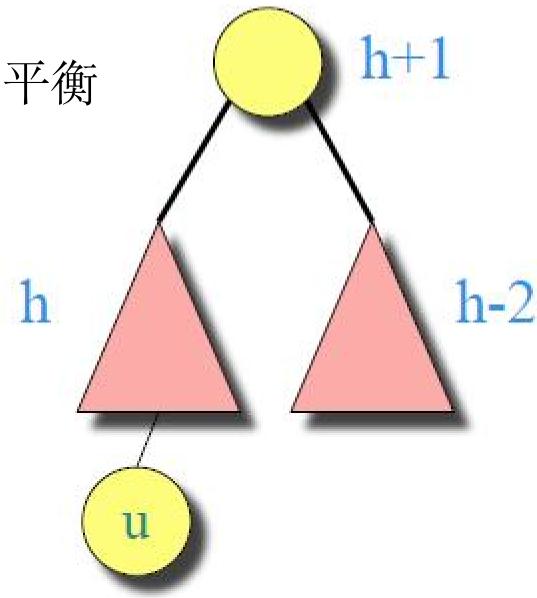



LEFT-ROTATE(1)



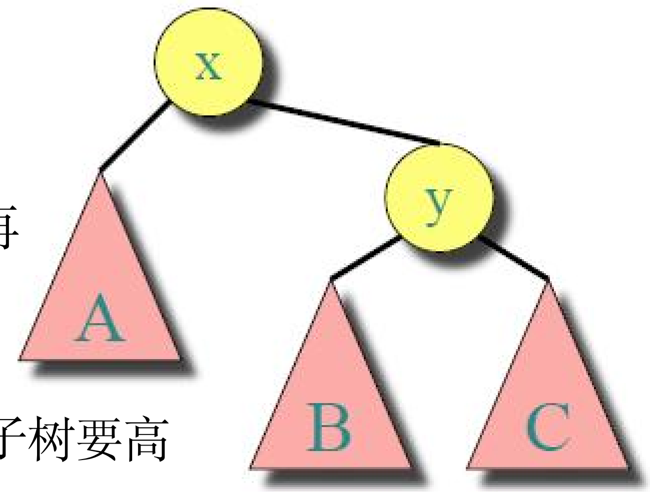
# Insertions

- Insert new node u as in the simple BST 插入一个新的节点可能导致不平衡
  - Can create imbalance
- Work your way up the tree, restoring the balance
- Similar issue/solution when deleting a node

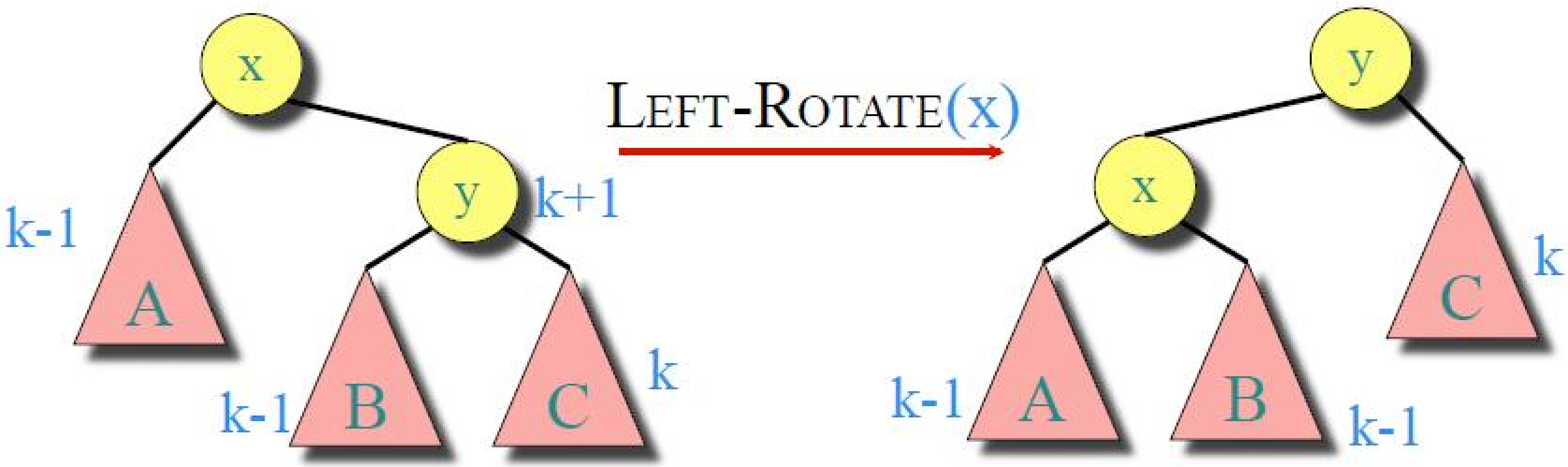


# Balancing

- Let x be the lowest "violating" node 使X节点是最低的违反属性的节点
  - We will fix the subtree of x and move up 先修复X为根节点的子树，再一步步往上修复
- Assume the right child of x is deeper than the left child of x (x is "right-heavy") 右高：假设x右子树比左子树要高
- Scenarios: 可能出现3种情况
  - Case 1: Right child y of x is right-heavy X的右节点y是右高
  - Case 2: Right child y of x is balanced X的右节点y是平衡的
  - Case 3: Right child y of x is left-heavy X的右节点y是左高

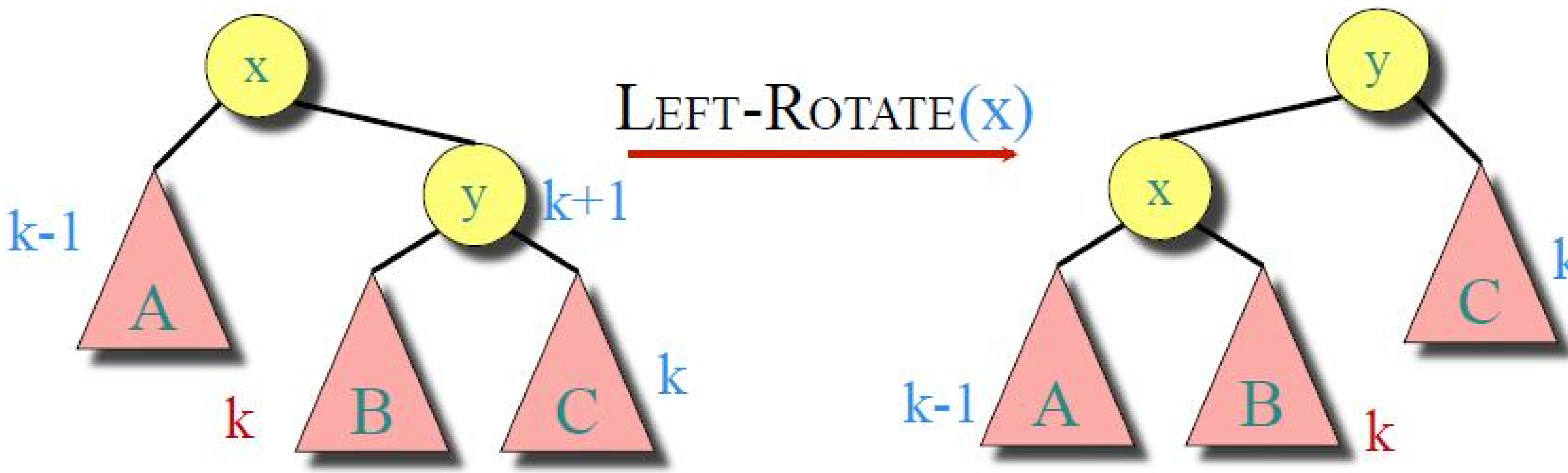


# Case 1: y is right-heavy




X的右节点y是右高

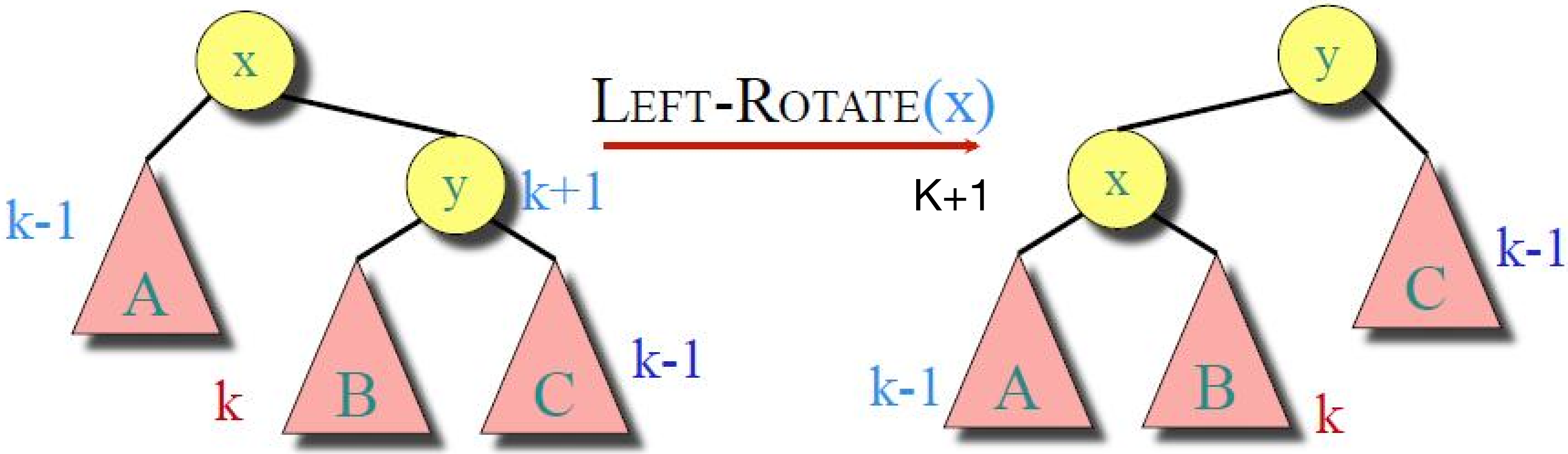
# Case 2: y is balanced




X的右节点y是平衡的

Same as Case 1

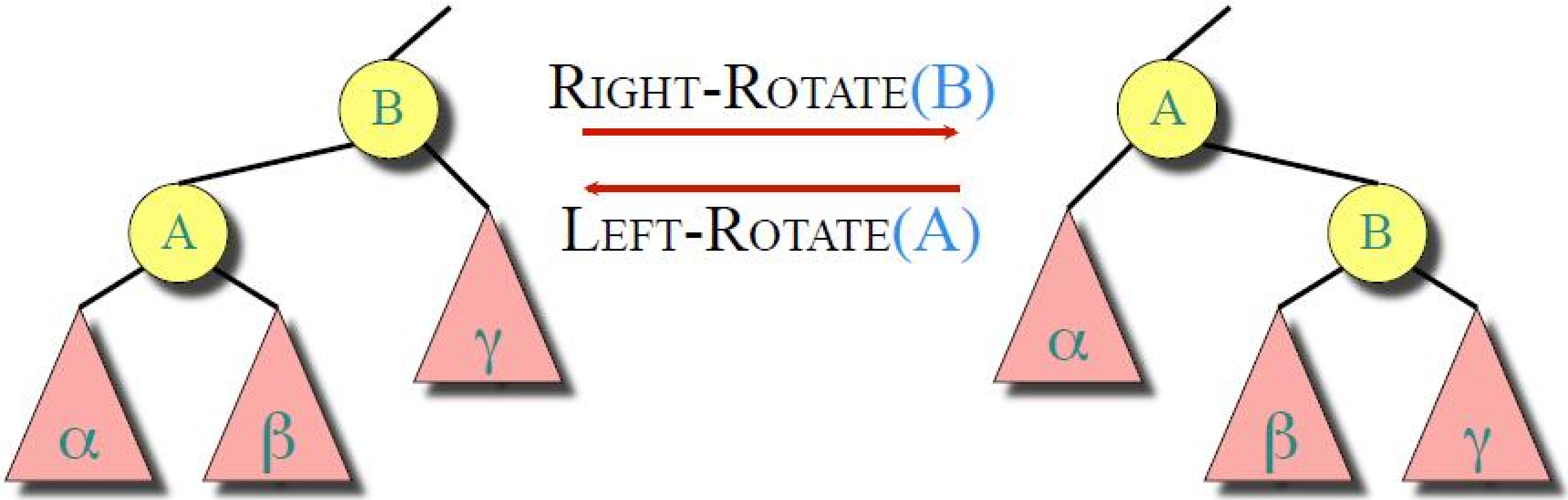
# Case 3: y is left-heavy




X的右节点y是左高

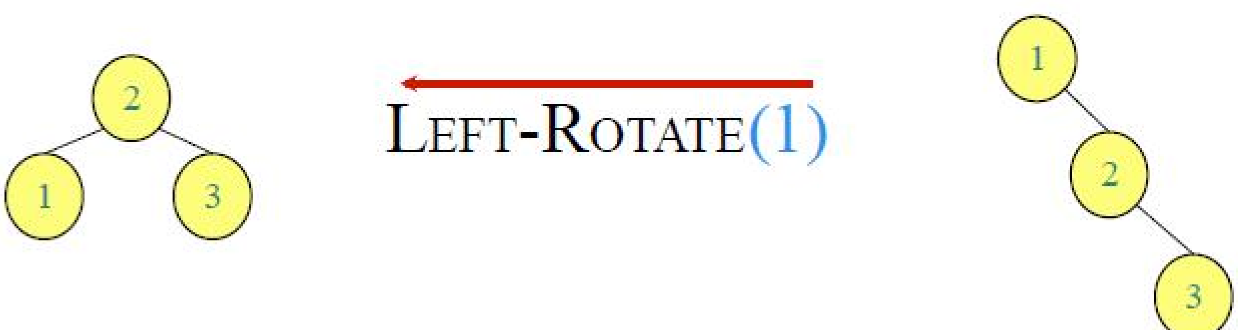
Need to do more …

# Rotations

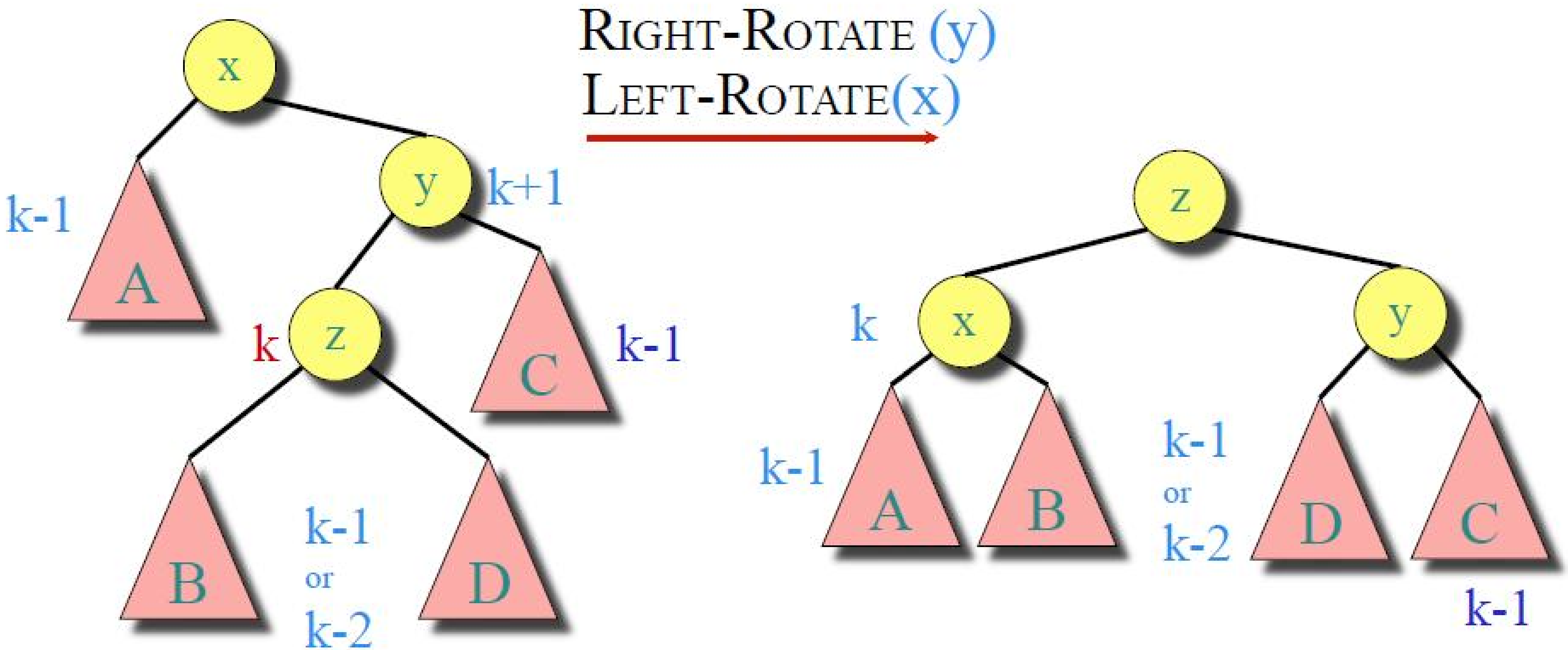



Rotations maintain the inorder ordering of keys:

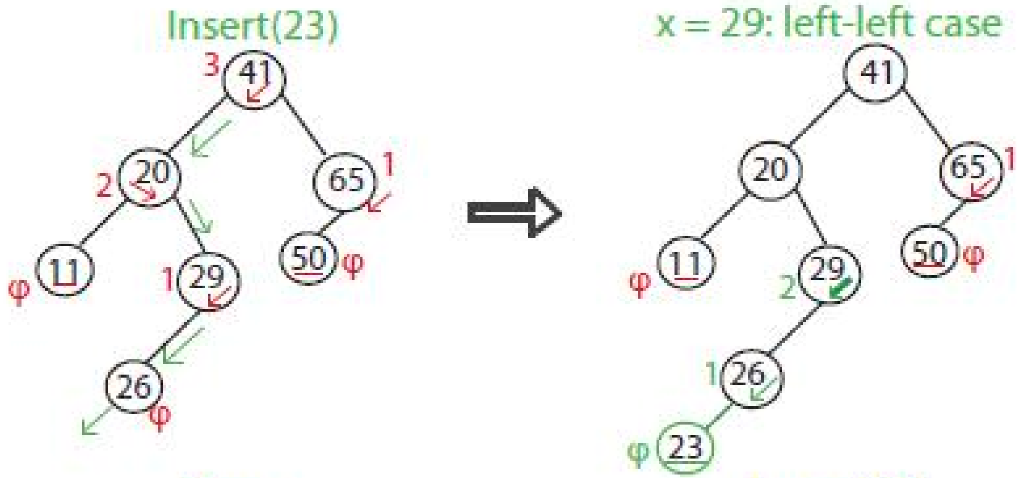
- $a \in \alpha, b \in \beta, c \in \gamma \;\Rightarrow\; a \leq A \leq b \leq B \leq c$.



# Case 3: y is left-heavy




And we are done!

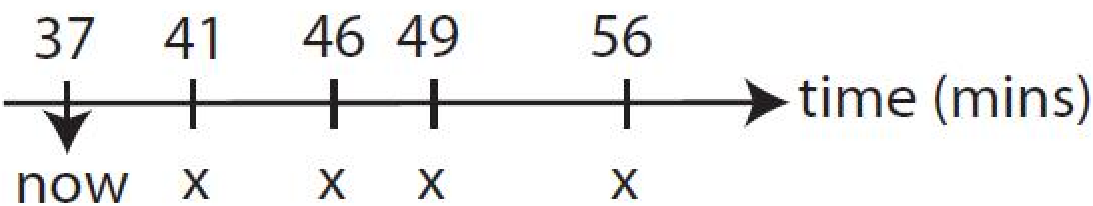

Insert(23)
41
20
65
11
29
50
26
x = 29: left-left case
41
20
65
11
29
50
26
23

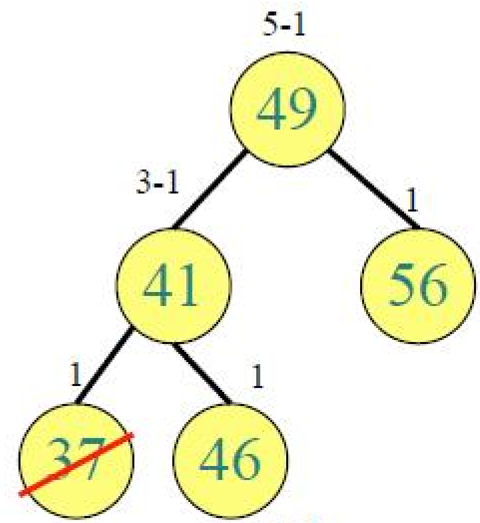
# Conclusions

- Can maintain balanced BSTs in O(log n) time per insertion
- Search etc take O(log n) time

# BST for runway reservation system

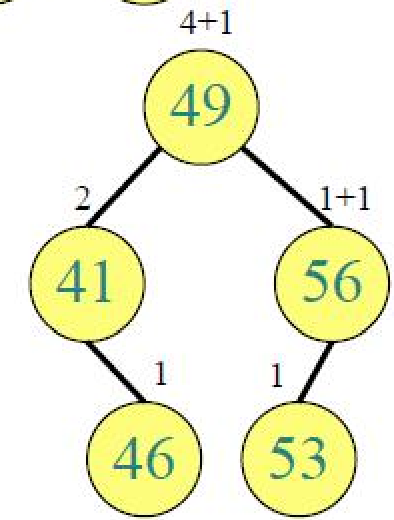
- R = (37, 41, 46, 49, 56) current landing times




- remove t from the set when a plane lands

  R = (41, 46, 49, 56)




- add new t to the set if no other landings are scheduled within < 3 minutes from t
  - 44 => reject (46 in R)
  - 53 => ok
- delete, insert, conflict checking take O(h), where h is the height of the tree

# Balanced Search Trees …

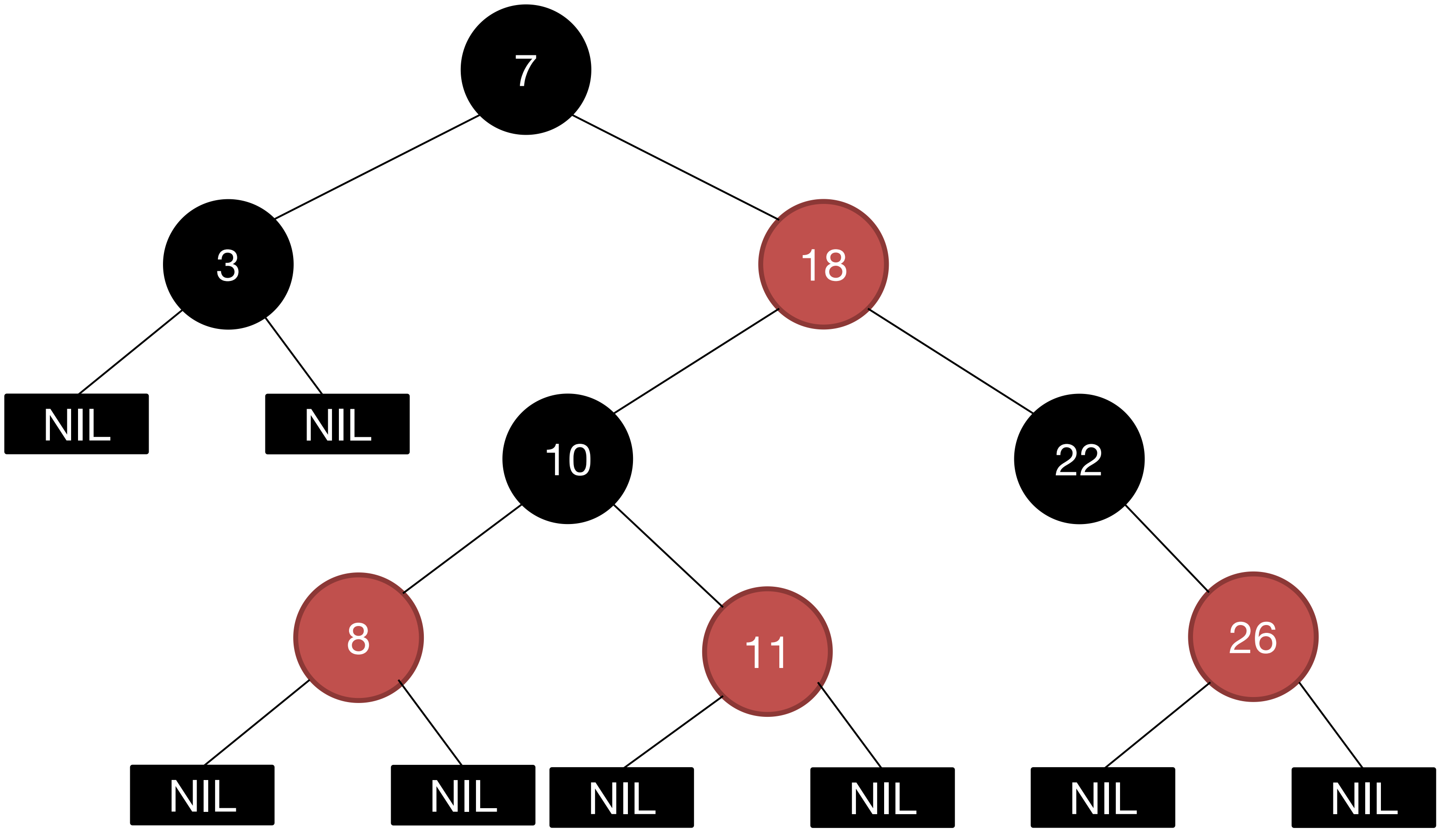
- AVL trees (Adelson-Velsii and Landis 1962)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Scapegoat trees (Galperin and Rivest 1993)
- Treaps (Seidel and Aragon 1996)
- ….

# Red-black Tree

- BST structure with extra color, satisfying :

1. Every node is either black or red;
2. Root and leaves are all black, and all leaves are NIL;
3. Red nodes' parents are black;
4. The paths from a node x to all its descendant leaves have the same number of black nodes.

# Example



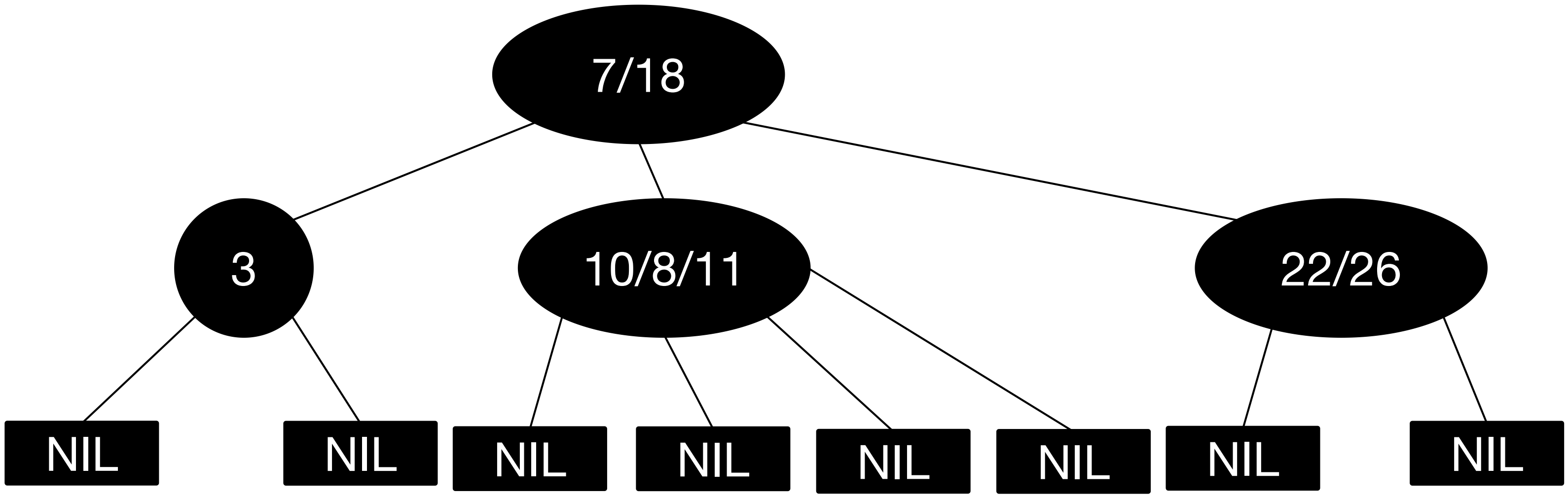
# Red-black Tree

- In Red-black tree, the same number of black nodes in all paths from a node x to all its descendant leaves have are called the *black height* of x
- Supposed that there are n nodes, there are n+1 leaves in a red-black tree(Proved in induction)

# Height of Red-Black Tree

- The height of a red-black tree is smaller than 2log(n+1)=O(log n)
- Proof: Merge red node with its black parent. Then, the tree becomes a 2-3-4 tree, where each node have 2, 3 or 4 children and all leaves have the same depth that is black height h'.

# Example for Proof

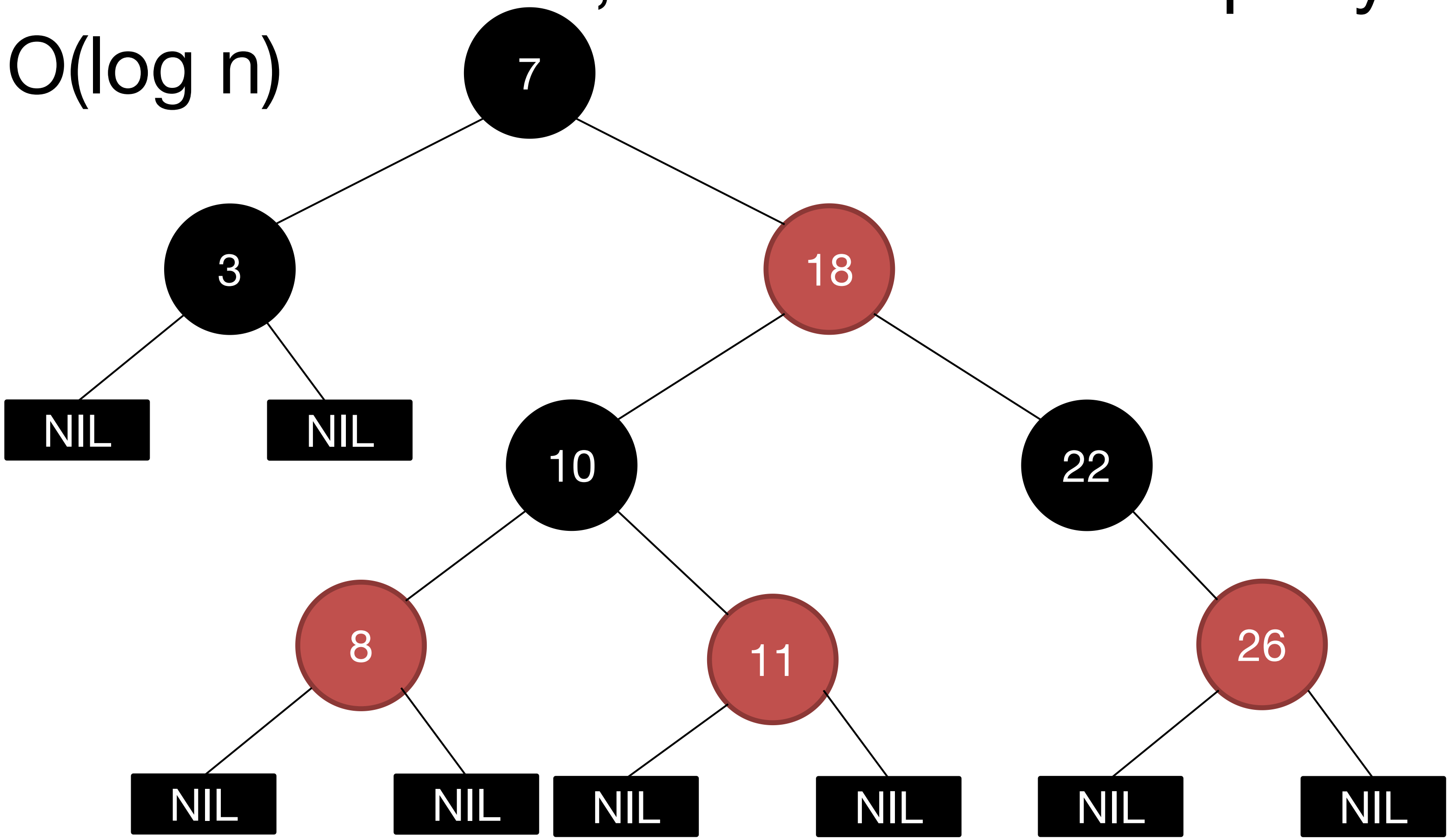


# Height of Red-Black Tree

- The height of a red-black tree is smaller than 2log(n+1)=O(log n)
- Proof: $2^{h'} \leq \#leaves \leq 4^{h'} \Rightarrow 2^{h'} \leq n+1 \Rightarrow$ $h' \leq log(n+1)$
- Hence, the height of a red-black tree h is samller than 2h', so $h \leq 2log(n+1)$

# Search Red-black Tree

- Seach as in BST, and the cost of query is O(log n)



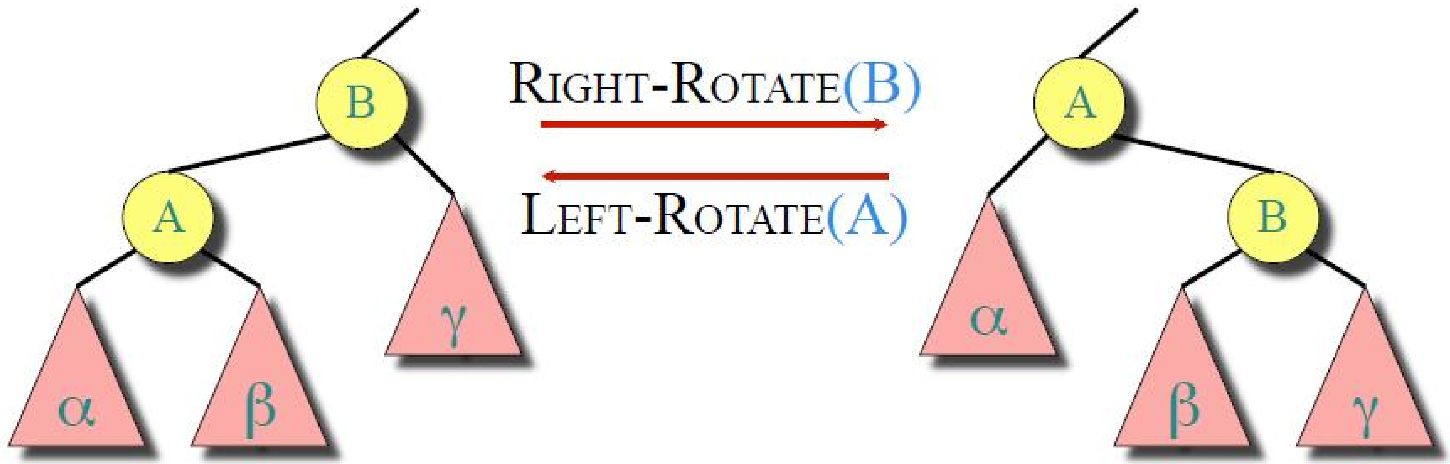
# Insert Red-black Tree

- The cost of update is O(log n)

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
    x = x.left
    else x = x.right
z.p = y
if y == T.nil
        T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                    // case 1
            y.color = BLACK                      // case 1
            z.p.p.color = RED                    // case 1
            z = z.p.p                            // case 1
        else if z == z.p.right
                z = z.p                          // case 2
                LEFT-ROTATE(T, z)                // case 2
            z.p.color = BLACK                    // case 3
            z.p.p.color = RED                    // case 3
            RIGHT-ROTATE(T, z.p.p)               // case 3
    else(same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

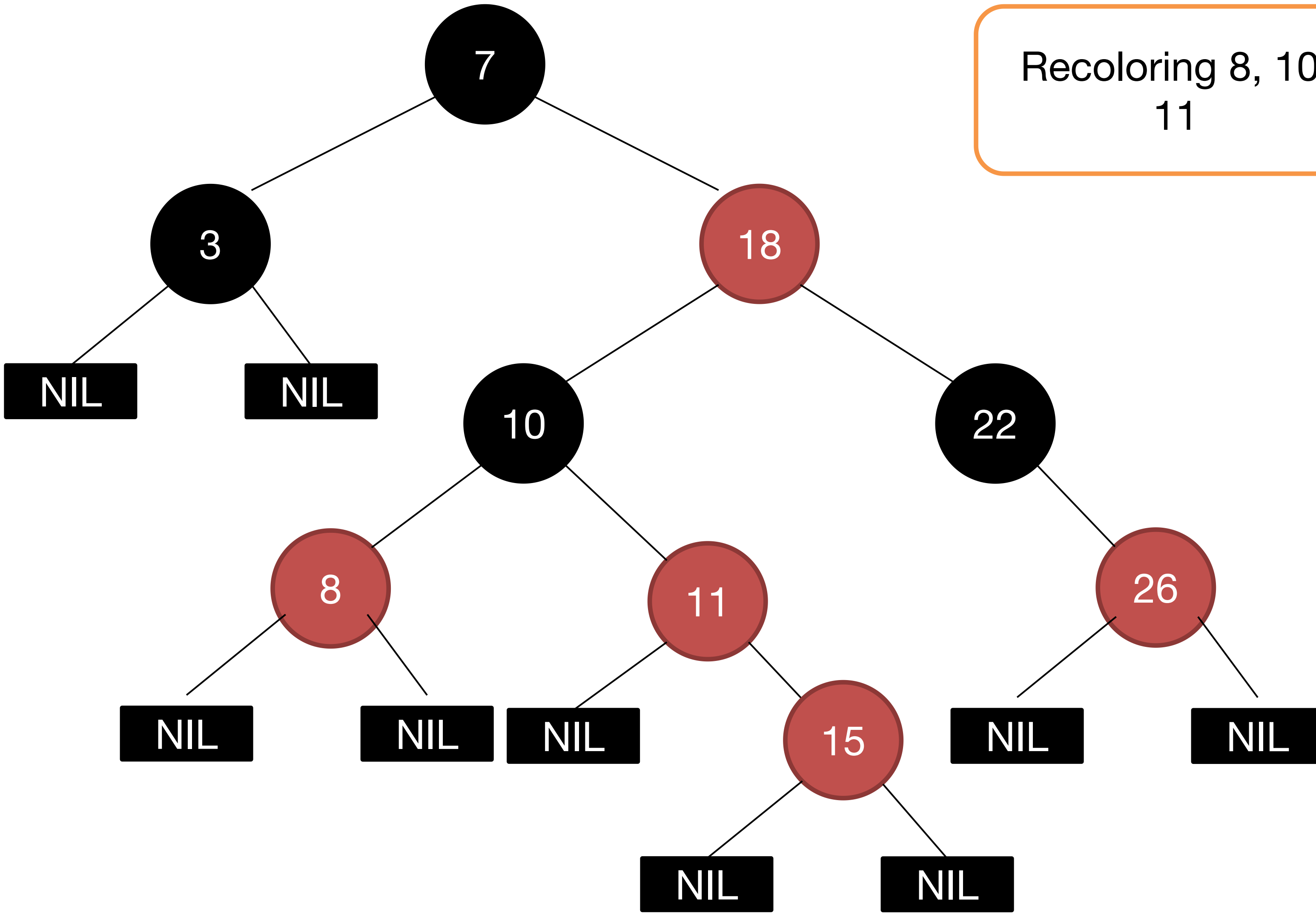
# Left and Right Rotations




Rotations maintain the inorder ordering of keys:

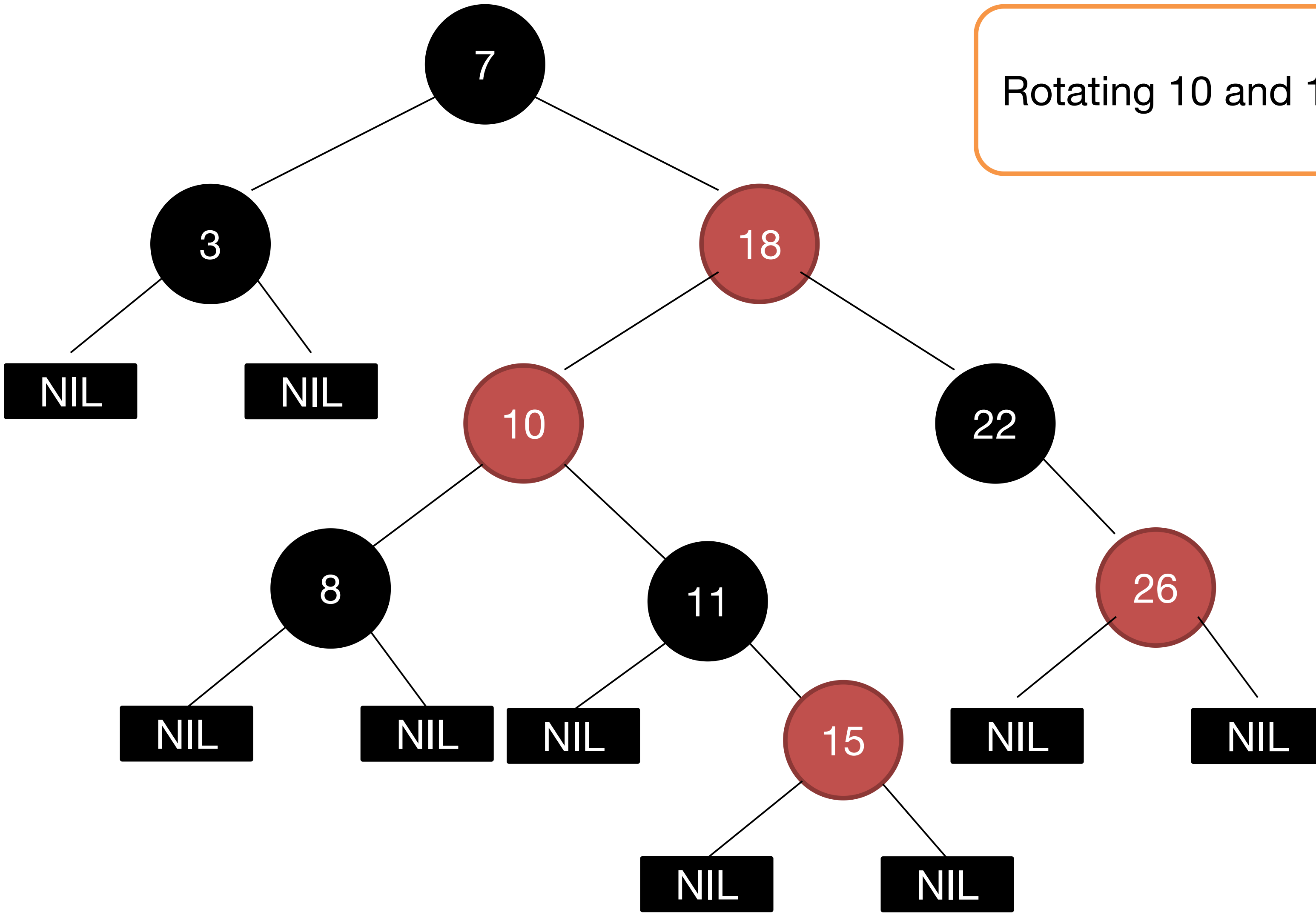
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \le A \le b \le B \le c.$

# Example Insert 15



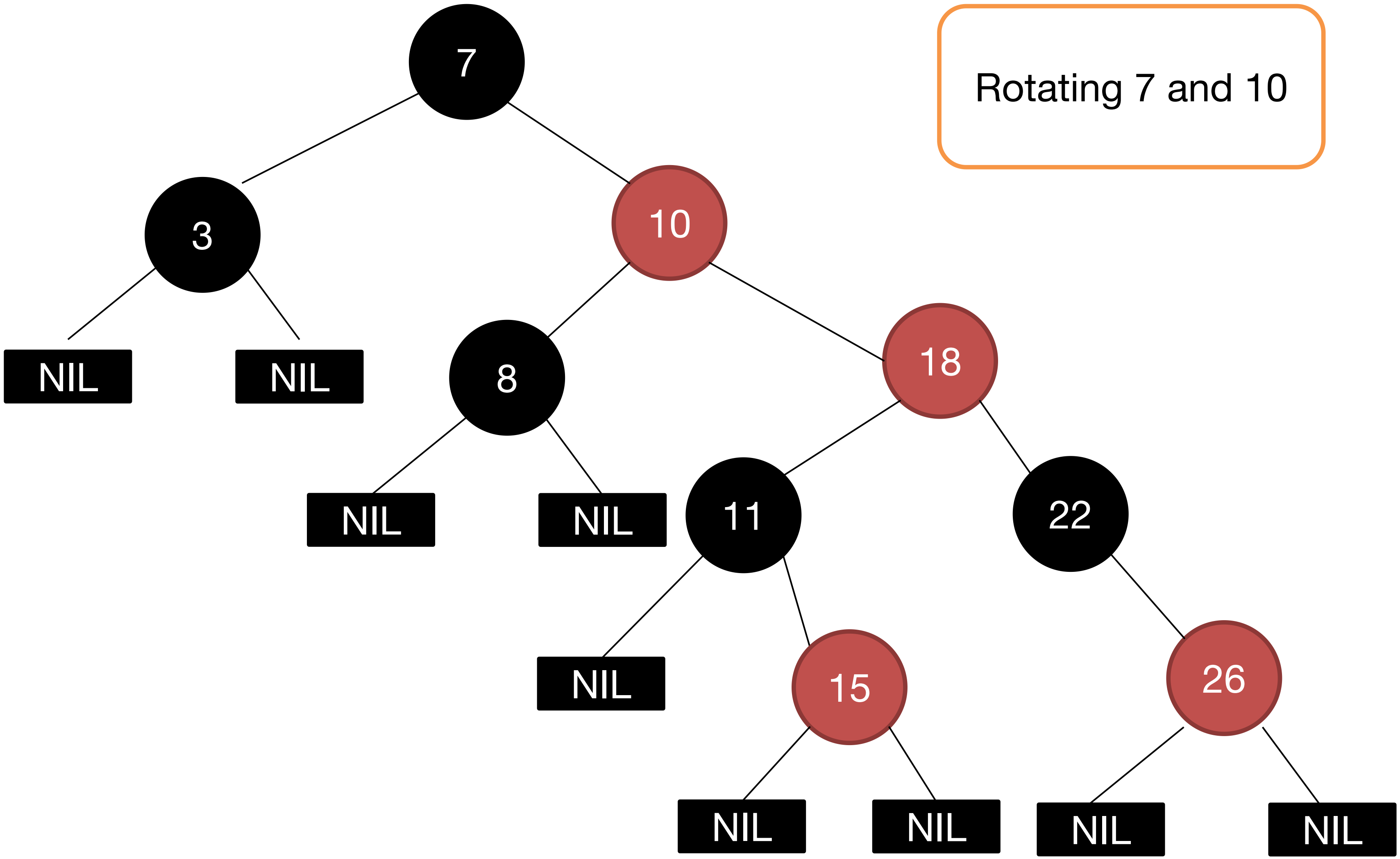

Recoloring 8, 10, 11

# Example Insert 15



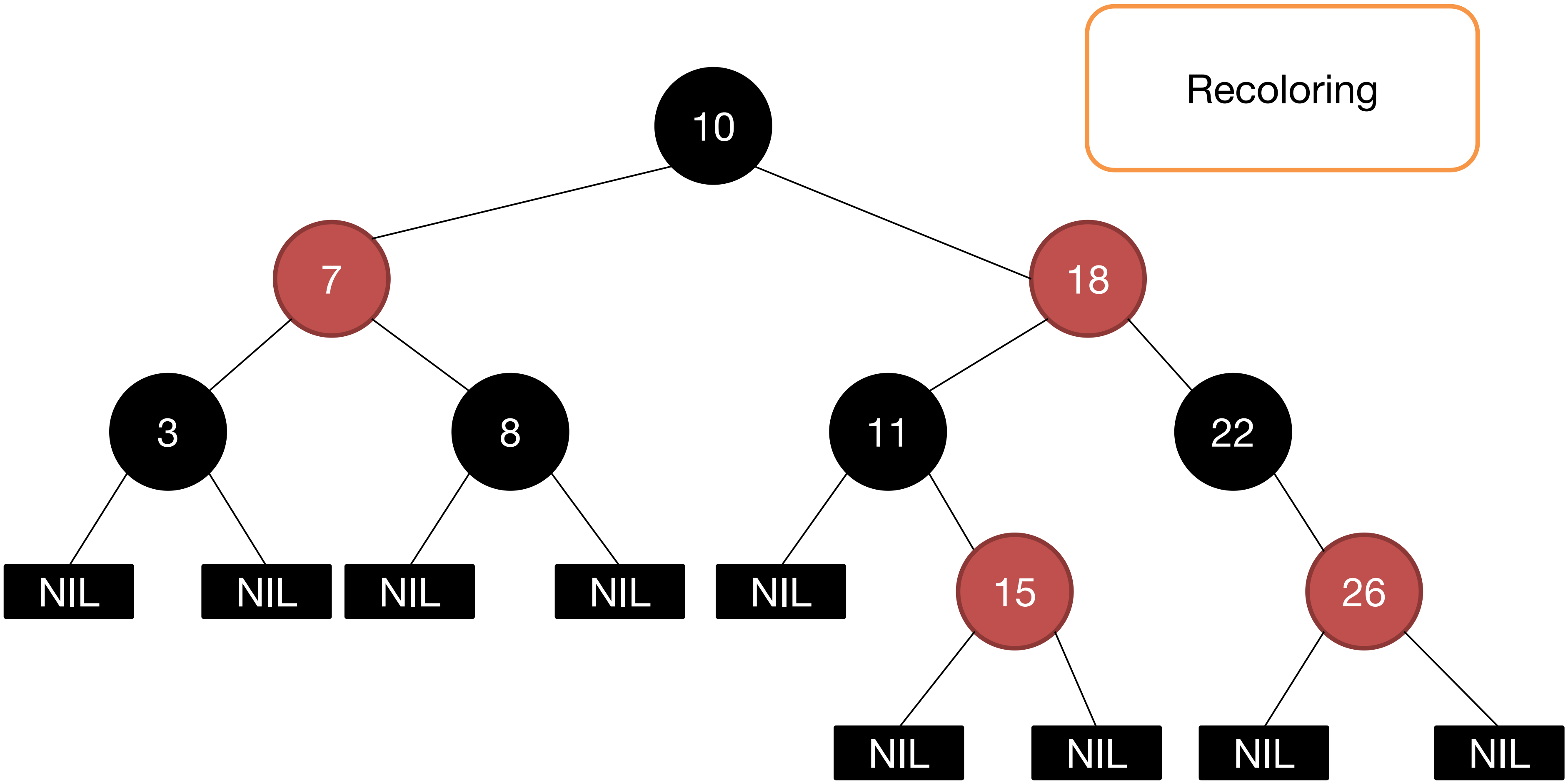

Rotating 10 and 18

# Example Insert 15



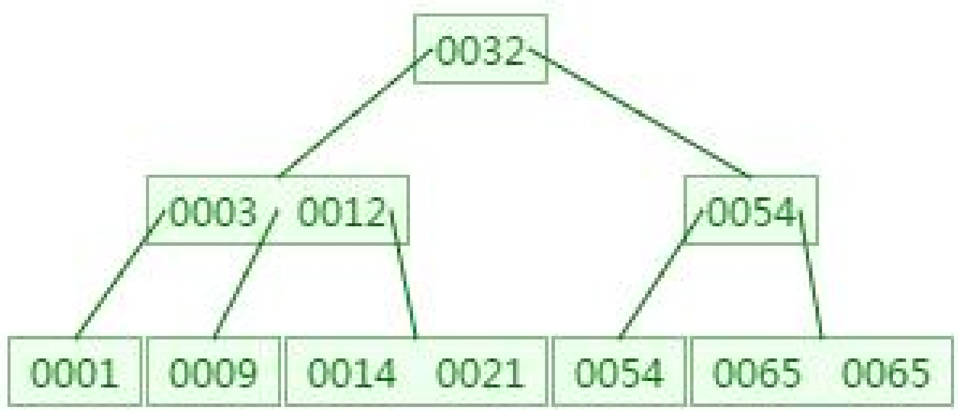

Rotating 7 and 10

# Example Insert 15



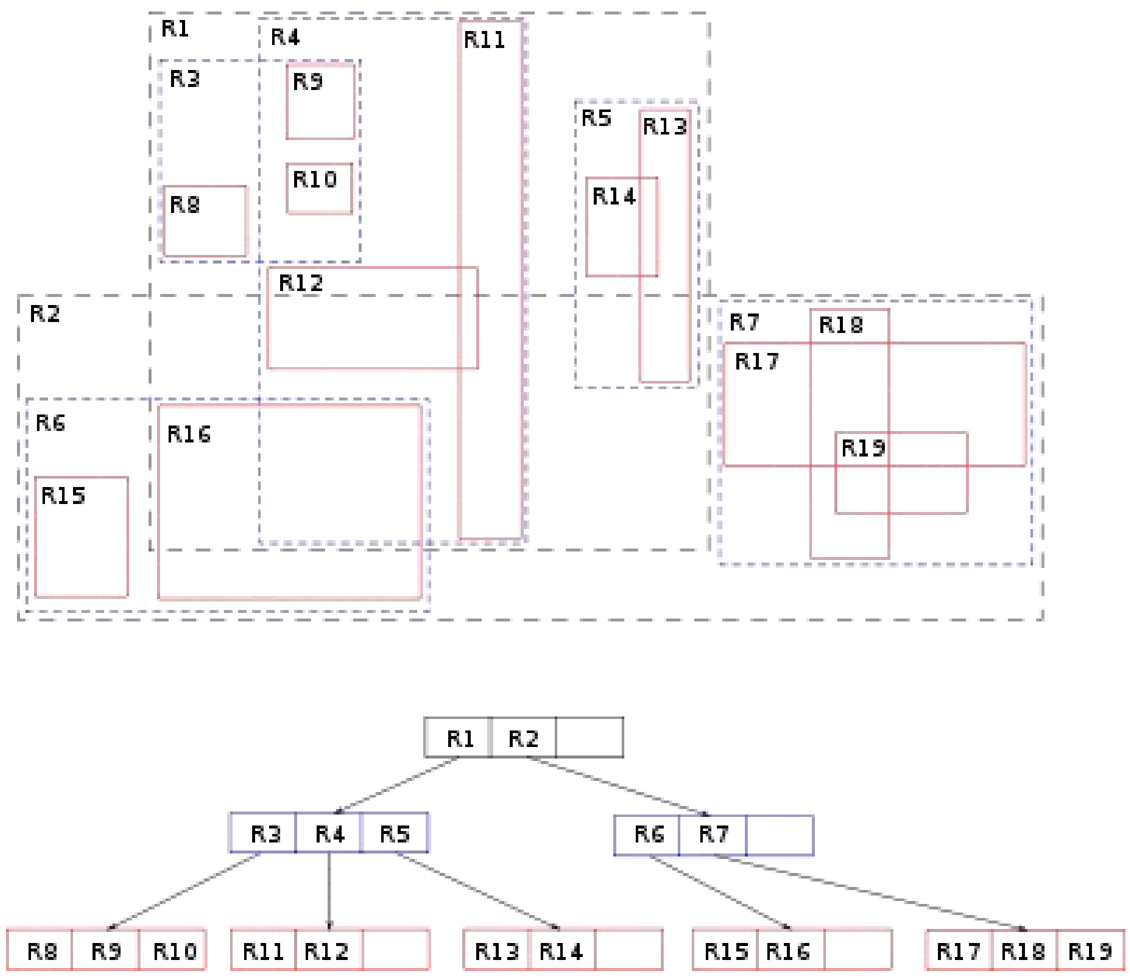
# B-Tree

- In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within the pre-defined range [m/2,m].



# R-Tree

- R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons.


R1
R4
R11
R3
R9
R5
R13
R10
R14
R8
R12
R2
R7
R18
R17
R6
R16
R19
R15
R1 R2
R3 R4 R5
R6 R7
R8 R9 R10
R11 R12
R13 R14
R15 R16
R17 R18 R19

# Signature Tree

- signature pointed to by the corresponding leaf node.