底层基本数据结构：

- 链表（LinkedList/List）-线性
- 树（Tree）
- 图（Graph）

<u>研究的问题：一切希望相关操作的算法运行时间尽量缩短，尽管代码可能复杂</u>

对一系列数进行 search 某一个数
单纯只对一系列数进行某一个数的一次 search，wc time complexity : O(n)
但如果要多次对多个系列数进行对应的数的 search，就要进行先排序再按照某种搜索算法进行搜索（老方法，O(n^2)）
排序 – O(nlgN)
涉及：排序、搜索/查询 （新方法，O(n^lgn) < O(n^2)）

比如说：Binary Search Algorithm

Search/查找/查询

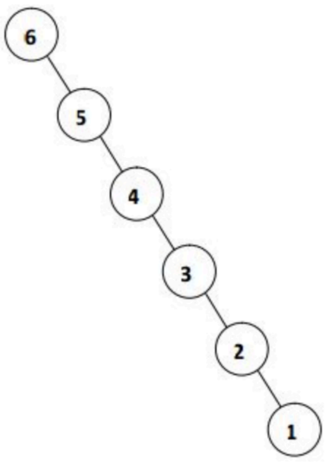
对数据进行基本的增（Insert）、删（Delete）、改、查（Search）

有一些数据库服务器（例如 postgreSQL）的底层是通过树（Tree）来存储数据的

AVL 树存在的目的是什么？
AVL Tree 是自平衡二叉查找树（Self Balancing Binary Search Tree）的一种实现方式

为什么会有自平衡二叉查找树（Self Balancing Binary Search Tree）？

主要是因为 Binary Search Tree（二叉查找树）
在遇到有序数列的情况下会退化成「链表 List」



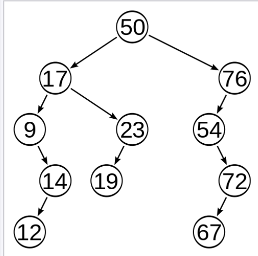
而「链表」查找元素的时间复杂度为O(n)。所以为了避免出现这种情况，就需要「自平衡」。

Self-balancing binary search tree

From Wikipedia, the free encyclopedia

> This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(November 2010) (Learn how and when to remove this template message)*

In computer science, a **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.[1]

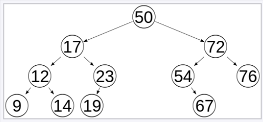


An example of an **unbalanced** tree; following the path from the root to a node takes an average of 3.27 node accesses

These structures provide efficient implementations for mutable ordered lists, and can be used for other abstract data structures such as associative arrays, priority queues and sets.

The red–black tree, which is a type of self-balancing binary search tree, was called symmetric binary B-tree[2] and was renamed but can still be confused with the generic concept of **self-balancing binary search tree** because of the initials.

Contents [hide]



The same tree after being height-balanced; the average path effort decreased to 3.00 node accesses

Overview [edit]

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. A binary tree with height *h* can contain at most $2^0+2^1+\cdots+2^h = 2^{h+1}-1$ nodes. It follows that for a tree with *n* nodes and height *h*:

$$n \le 2^{h+1} - 1$$

通俗一些，就是将原来位置随意琐碎乱套的二叉搜索树（ Binary Search Tree），有 n 个结点（Node），通过某种方式，尽量将树的高度变矮、两边饱满（尽量都有 subtree, leaf），高度控制在 floor(lgN)

实现自平衡二叉查找树（Self Balancing Binary Search Tree）的几种数据结构（包括了 AVL 树）
AVL 树更严格，不仅要遵循基本 bst 性质，又要每个结点（Node）的平衡因子的值在-1 和 1 之间

Implementations [edit]

Popular data structures implementing this type of tree include:

- 2-3 tree
- AA tree
- AVL tree
- B-tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap
- Weight-balanced tree

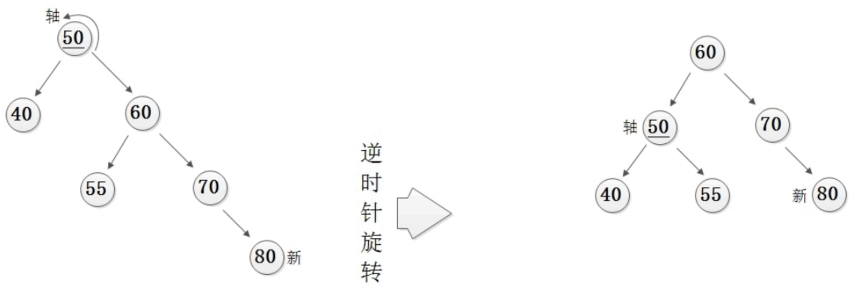
对 AVL 树旋转方式的误解（与常态下的旋转完全是两个不同的概念）
AVL 树旋转在哪里出现？在 bst insert 之后的 fix imbalance 里出现

- 所谓左“旋转”：50 这个结点受影响，是让 60 结点做 root，50 做 60 的左子树（node70 是新 insert 的，insert 到 60 的右子树）
-右旋转同理

情况 3

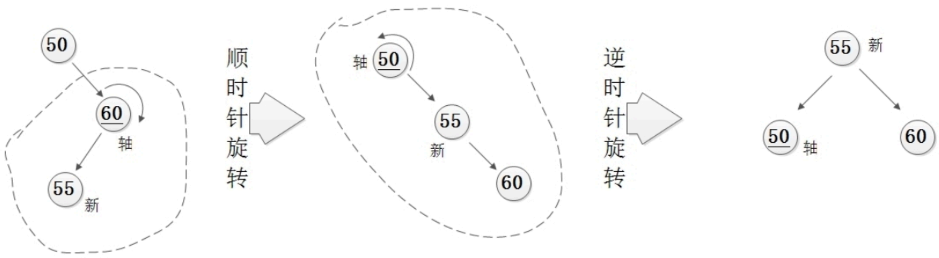
-同时左旋转还包含一种情况：（node80 是新 insert 的，insert 到 70 的右子树）同样是 50 这个结点受影响，是让 60 结点做 root，让 60 左侧的所有结点统一做 60 的左子树

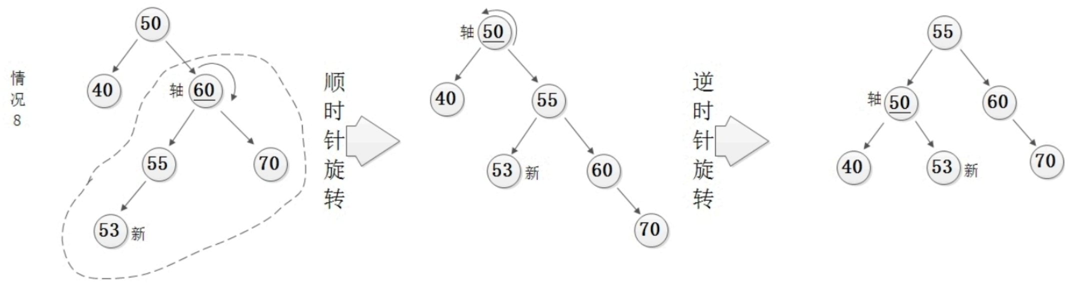
情况 4



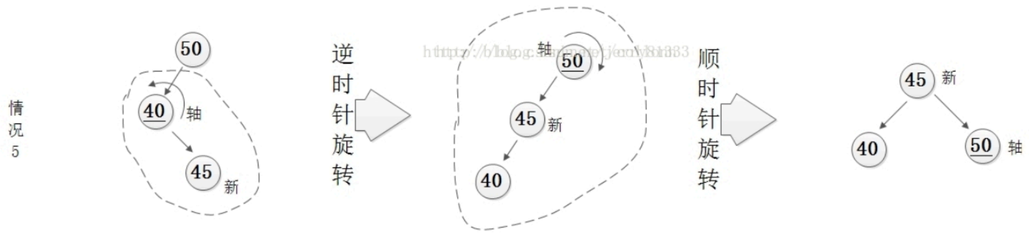
右左（**RL**）- 都要经历两次所谓的“旋转”（先右后左）

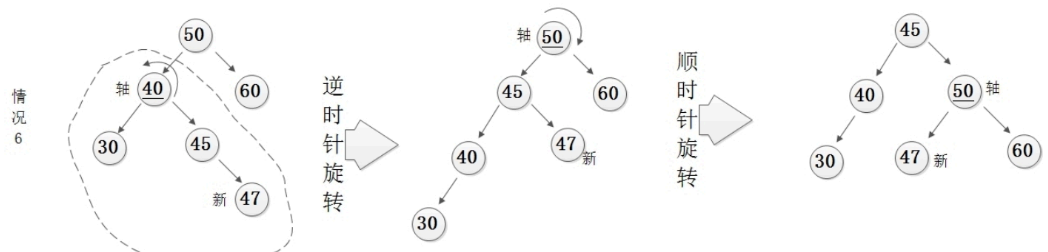
情况 7





左右（**LR**）- 都要经历两次所谓的“旋转”（先左后右）





- 自下往上进行所谓的“旋转”（递归）

AVL 树相关操作算法的 precondition 和 postcondition（依然都要保证是 AVL 树）

Augmented Data Structure
往原有 bst/avl 里的树结点加新的 field/attribute，使一些操作/method 方法运行时间保持不变或更快。（比如说 node.height，如果没有 height, 则还要在算法里单独写 height 的 helper function）

但加了新的 field/attribute 之后，每一次基本的 insert、delete 都要伴随实时更新每一个树结点的新的 field/attribute，再之后，去进行相应的操作/method（查询等）

Tutorial 3

	select	rank	insert/delete	search
AVL	O(n)	O(n)	O(lgN)	O(lgN)
AVL node.rank	O(lgN)	O(lgN)	O(n)	O(lgN)
AVL node.size	O(lgN)	O(lgN)	O(lgN)	O(lgN)
sort	0(1)	O(lgN)	O(nlgN)	O(lgN)

O(lgN)基本上代表着以高度 height 从 root 至下进行遍历
相比于传统的 traversal 遍历 preOrder, postOrder, inOrder 的 O(n)，即遍历每一个节点更节省时间