

{ "熵码匠艺": "Software Craftsmanship" }

博客园

首页

新随笔

联系

订阅

管理

随笔 - 205

文章 - 0

评论 - 2088

阅读 - 168万

二叉查找树

在文章《常用数据结构及复杂度》中，介绍了一些计算机程序设计常用的线性数据结构，包括 Array、ArrayList、LinkedList<T>、List<T>、Stack<T>、Queue<T>、Hashtable 和 Dictionary<T> 等。并简单介绍了这些数据结构的内部实现原理和常用操作的运算复杂度，以及如何选择合适的数据结构。本篇文章中，我们将介绍常见的树形结构及其常用操作的运算复杂度。

- 二叉树 (Binary Tree)
 - 完全二叉树和满二叉树
- 二叉查找树 (Binary Search Tree)
 - 插入节点
 - 删除节点
 - 遍历节点

我们知道像家谱族谱、公司组织结构等都是以树结构的形式组织数据。例如下图中所展示的公司组织结构图。



树 (Tree) 是由多个节点 (Node) 的集合组成，每个节点又有多个与其关联的子节点 (Child Node)。子节点就是直接处于节点之下的节点，而父节点 (Parent Node) 则位于节点直接关联的上方。树的根 (Root) 指的是一个没有父节点的单独的节点。

所有的树都呈现了一些共有的性质：

1. 只有一个根节点；
2. 除了根节点，所有节点都有且只有一个父节点；
3. 无环。将任意一个节点作为起始节点，都不存在任何回到该起始节点的路径。（正是前两个性质保证了无环的成立。）

树中使用的术语

- **根 (Root)**：树中最顶端的节点，根没有父节点。
- **子节点 (Child)**：节点所拥有子树的根节点称为该节点的子节点。
- **父节点 (Parent)**：如果节点拥有子节点，则该节点为子节点的父节点。
- **兄弟节点 (Sibling)**：与节点拥有相同父节点的节点。
- **子孙节点 (Descendant)**：节点向下路径上可达的节点。
- **叶节点 (Leaf)**：没有子节点的节点。
- **内节点 (Internal Node)**：至少有一个子节点的节点。
- **度 (Degree)**：节点拥有子树的数量。
- **边 (Edge)**：两个节点中间的链接。
- **路径 (Path)**：从节点到子孙节点过程中的边和节点所组成的序列。
- **层级 (Level)**：根为 Level 0 层，根的子节点为 Level 1 层，以此类推。
- **高度 (Height) /深度 (Depth)**：树中层的数量。比如只有 Level 0,Level 1,Level 2 则高度为 3。

类别	树名称
----	-----

公告

昵称：sangmado

园龄：10年11个月

荣誉：推荐博客

粉丝：2685

关注：50

+加关注

积分与排名

积分 - 568925

排名 - 742

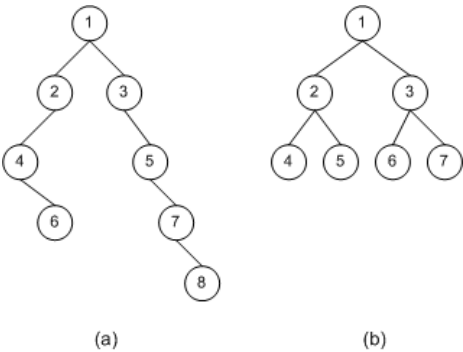
阅读排行榜

1. C# 高性能 TCP 服务的多种实现...
2. 算法复杂度分析(53747)
3. 常用数据结构及复杂度(51389)
4. Bellman-Ford 单源最短路径算法(...
5. 二叉查找树(47176)
6. WinDbg 命令三部曲：（一）Win...
7. Ford-Fulkerson 最大流算法(40596)
8. C# 异步 TCP 服务器完整实现(37...
9. 字符串匹配算法(36501)
10. 哈希表和完美哈希(33451)
11. 编写更好的C#代码(32027)
12. 后缀树(31714)
13. Dijkstra 单源最短路径算法(30701)
14. 比较排序算法(28214)
15. 开源倾情奉献：基于.NET打造I...
16. 代码的印象派：写点好代码吧(2...
17. Lock-Free 编程(21338)
18. K-Means 聚类算法(20858)
19. Johnson 全源最短路径算法(200...
20. 那些年黑了你的微软BUG(19443)
21. 软件质量模型(18618)
22. Service Locator 模式(18055)
23. C#开源实现JPEG流传输(17742)
24. 开放封闭原则（Open Closed Pr...
25. EntityFramework 中支持 BulkIn...
26. Scrum 是什么？(14858)
27. C# 对 TCP 客户端的状态封装(1...
28. 获取机器安装.NET版本的几种...
29. 秒懂C#通过Emit动态生成代码(1...
30. ConcurrentDictionary 对决 Dicti...
31. 人人都是 DBA (I) SQL Server ...
32. 里氏替换原则（Liskov Substitut...
33. Boyer-Moore 字符串匹配算法(1...
34. 深度优先搜索检测有向图有无环...
35. 设计模式之美(13150)
36. 我是一个垃圾程序员(12369)
37. 单一职责原则（Single Respons...
38. Floyd-Warshall 全源最短路径算...
39. WinDbg 命令三部曲：（二）Wi...

二叉查找树 (Binary Search Tree)	二叉查找树，笛卡尔树，T 树
自平衡二叉查找树 (Self-balancing Binary Search Tree)	AA 树，AVL 树， 红黑树 (Red-Black Tree) ， 伸展树 (Splay Tree)
B 树 (B-Tree)	2-3 树， 2-3-4 树， B 树，B+ 树，B* 树
字典树 (Trie-Tree)	后缀树，基数树，三叉查找树，快速前缀树
空间数据分割树 (Spatial Data Partitioning Tree)	R 树，R+ 树，R* 树， 线段树，优先 R 树

二叉树 (Binary Tree)

二叉树 (Binary Tree) 是一种特殊的树类型，其每个节点最多只能有两个子节点。这两个子节点分别称为当前节点的左孩子 (left child) 和右孩子 (right child) 。



上图中，二叉树 (a) 包含 8 个节点，其中节点 1 是它的根节点。节点 1 的左孩子为节点 2，右孩子为节点 3。注意，并没有要求一个节点同时具有左孩子和右孩子。例如，二叉树 (a) 中，节点 4 就只有一个右孩子 6。此外，节点也可以没有孩子节点。例如，二叉树 (b) 中，节点 4、5、6、7 都没有孩子节点。

没有孩子的节点称为叶节点 (Leaf Node)，有孩子的节点则称为内节点 (Internal Node)。如上图中，二叉树 (a) 中节点 6、8 为叶节点，节点 1、2、3、4、5、7 为内节点。

完全二叉树和满二叉树

完全二叉树 (Complete Binary Tree)：深度为 h，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 h 的满二叉树中，序号为 1 至 n 的节点对应时，称之为完全二叉树。

满二叉树 (Full Binary Tree)：一棵深度为 h，且有 $2^h - 1$ 个节点称之为满二叉树。

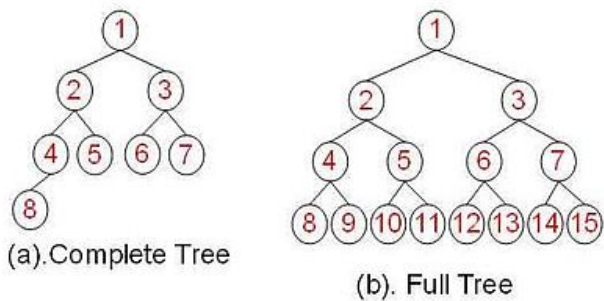
A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A **full binary tree** is a tree in which every node other than the leaves has two children.

40. Cowboy 开源 WebSocket 网络...

最新评论

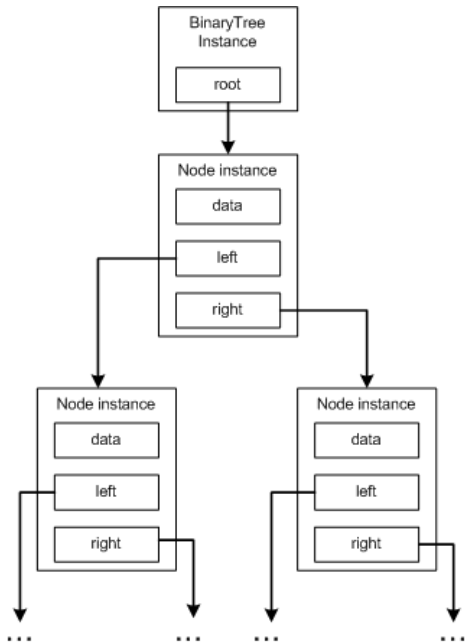
1. Re:接口分离原则 (Interface Seg...
牛皮
--苍迹
2. Re:字符串匹配算法
@sangmado 哈哈哈哈哈，博主太实在了...
--flameking
3. Re:人人都是 DBA (VII) B 树和 ...
感觉B树的删除那里没有将清楚呀
--沾三焚
4. Re:算法范式
赞
--弈心者
5. Re:二叉查找树
感谢分享
--身轻如阿宝



	完全二叉树	满二叉树
总节点数 k	$2^{h-1} \leq k < 2^h - 1$	$k = 2^h - 1$
树高 h	$h = \log_2 k + 1$	$h = \log_2(k + 1)$

.NET 中并没有直接提供二叉树的实现，我们需要自己来实现 `BinaryTree` 类。在《你曾实现过二叉树吗》一文中，实现了一个基于泛型的简单的二叉树模型。

我们已经了解了 数组 是将元素连续地排列在内存当中，而二叉树却不是采用连续的内存存放。实际上，通常 `BinaryTree` 类的实例仅包含根节点（Root Node）实例的引用，而根节点实例又分别指向它的左右孩子节点实例，以此类推。所以关键的不同之处在于，组成二叉树的节点对象实例可以分散到 CLR 托管堆中的任何位置，它们没有必要像数组元素那样连续的存放。



如果要访问二叉树中的某一个节点，通常需要逐个遍历二叉树中的节点，来定位那个节点。它不象数组那样能对指定的节点进行直接的访问。所以查找二叉树的渐进时间是线性的 $O(n)$ ，在最坏的情况下需要查找树中的所有节点。也就是说，随着二叉树节点数量增加时，查找任一节点的步骤数量也将相应地增加。

那么，如果一个二叉树的查找时间是线性的，定位时间也是线性的，那相比 数组 来说到底哪里有优势呢？毕竟数组的查找时间虽然是线性 $O(n)$ ，但定位时间却是常量 $O(1)$ 啊？的确是这样，通常来说普通的二叉树确实不能提供比数组更好的性能。然而，如果我们按照一定的规则来组织排列二叉树中的元素时，就可以很大程度地改善查询时间和定位时间。

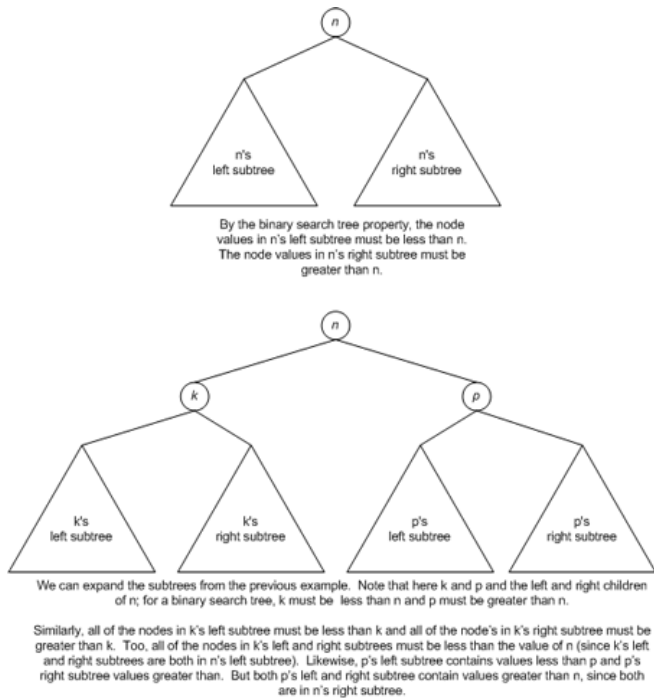
二叉查找树 (Binary Search Tree)

二叉查找树 (BST: Binary Search Tree) 是一种特殊的二叉树，它改善了二叉树节点查找的效率。二叉查找树有以下性质：

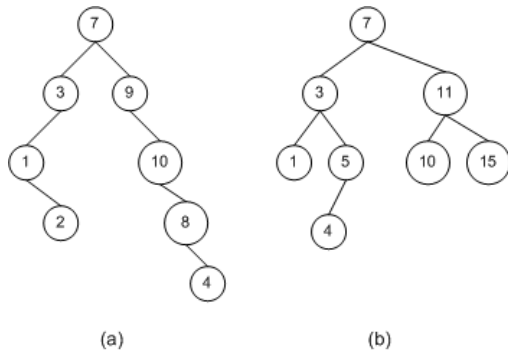
对于任意一个节点 n ,

- 其左子树 (left subtree) 下的每个后代节点 (descendant node) 的值都小于节点 n 的值；
- 其右子树 (right subtree) 下的每个后代节点的值都大于节点 n 的值。

所谓节点 n 的子树，可以将其看作是以节点 n 为根节点的树。子树的所有节点都是节点 n 的后代，而子树的根则是节点 n 本身。



下图中展示了两个二叉树。二叉树 (b) 是一个二叉查找树 (BST)，它符合二叉查找树的性质规定。而二叉树 (a)，则不是二叉查找树。因为节点 10 的右孩子节点 8 小于节点 10，但却出现在节点 10 的右子树中。同样，节点 8 的右孩子节点 4 小于节点 8，但出现在了它的右子树中。无论是在哪个位置，只要不符合二叉查找树的性质规定，就不是二叉查找树。例如，节点 9 的左子树只能包含值小于节点 9 的节点，也就是 8 和 4。



从二叉查找树的性质可知，BST 各节点存储的数据必须能够与其他的节点进行比较。给定任意两个节点，BST 必须能够判断这两个节点的值是小于、大于还是等于。

假设我们要查找 BST 中的某一个节点。例如在上图中的二叉查找树 (b) 中，我们要查找值为 10 的节点。

我们从根开始查找。可以看到，根节点的值为 7，小于我们要查找的节点值 10。因此，如果节点 10 存在，必然存在于其右子树中，所以应该跳到节点 11 继续查找。此时，节点值 10 小于节点 11 的值，则节点 10 必然存在于节点 11 的左子树中。在查找节点 11 的左孩子，此时我们已经找到了目标节点 10，定位于此。

如果我们要查找的节点在树中不存在呢？例如，我们要查找节点 9。重复上述操作，直到到达节点 10，它大于节点 9，那么如果节点 9 存在，必然存在于节点 10 的左子树中。然而我们看到节点 10 根本就没有左孩子，因此节点 9 在树中不存在。

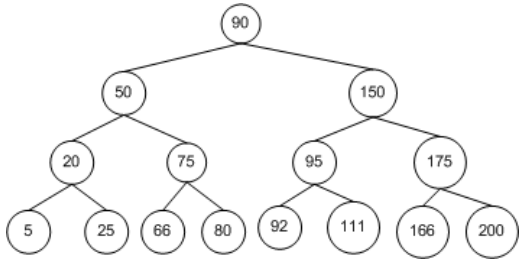
总结来说，我们使用的查找算法过程如下：

假设我们要查找节点 n ，从 BST 的根节点开始。算法不断地比较节点值的大小直到找到该节点，或者判定不存在。每一步我们都要处理两个节点：树中的一个节点，称为节点 c ，和要查找的节点 n ，然后并比较 c 和 n 的值。开始时，节点 c 为 BST 的根节点。然后执行以下步骤：

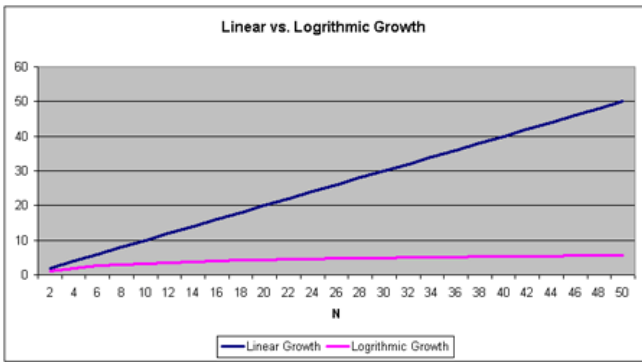
1. 如果 c 值为空，则 n 不在 BST 中；
2. 比较 c 和 n 的值；
3. 如果值相同，则找到了指定节点 n ；
4. 如果 n 的值小于 c ，那么如果 n 存在，必然在 c 的左子树中。回到第 1 步，将 c 的左孩子作为 c ；

5. 如果 n 的值大于 c ，那么如果 n 存在，必然在 c 的右子树中。回到第 1 步，将 c 的右孩子作为 c ；

通过 BST 查找节点，理想情况下我们需要检查的节点数可以减半。如下图中的 BST 树，包含了 15 个节点。从根节点开始执行查找算法，第一次比较决定我们是移向左子树还是右子树。对于任意一种情况，一旦执行这一步，我们需要访问的节点数就减少了一半，从 15 降到了 7。同样，下一步访问的节点也减少了一半，从 7 降到了 3，以此类推。

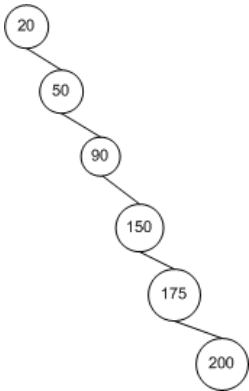


根据这一特点，查找算法的时间复杂度应该是 $O(\log_2 n)$ ，简写为 $O(\lg n)$ 。我们在文章《算法复杂度分析》中有一些关于时间复杂度的描述。可知， $\log_2 n = y$ ，相当于 $2^y = n$ 。即，如果节点数量增加 n ，查找时间只缓慢地增加到 $\log_2 n$ 。下图中显示了 $O(\log_2 n)$ 和线性增长 $O(n)$ 的增长率之间的区别。时间复杂度为 $O(\log_2 n)$ 的算法运行时间为下面那条线。



从上图可以看出， $O(\log_2 n)$ 曲线几乎是水平的，随着 n 值的增加，曲线增长十分缓慢。举例来说，查找一个具有 1000 个元素的数组，需要查询 1000 个元素，而查找一个具有 1000 个元素的 BST 树，仅需查询不到 10 个节点 ($\log_2 1024 = 10$)。

而实际上，对于 BST 查找算法来说，其十分依赖于树中节点的拓扑结构，也就是节点间的布局关系。下图描绘了一个节点插入顺序为 20, 50, 90, 150, 175, 200 的 BST 树。这些节点是按照递升顺序被插入的，结果就是这棵树没有广度 (Breadth) 可言。也就是说，它的拓扑结构其实就是将节点排在一条线上，而不是以扇形结构散开，所以查找时间也为 $O(n)$ 。



当 BST 树中的节点以扇形结构散开时，对它的插入、删除和查找操作最优的情况下可以达到亚线性的运行时间 $O(\log_2 n)$ 。因为当在 BST 中查找一个节点时，每一步比较操作后都会将节点的数量减少一半。尽管如此，如果拓扑结构像上图中的样子时，运行时间就会退减到线性时间 $O(n)$ 。因为每一步比较操作后还是需要逐个比较其余的节点。也就是说，在这种情况下，在 BST 中查找节点与在数组 (Array) 中查找就基本类似了。

因此，BST 算法查找时间依赖于树的拓扑结构。最佳情况是 $O(\log_2 n)$ ，而最坏情况是 $O(n)$ 。

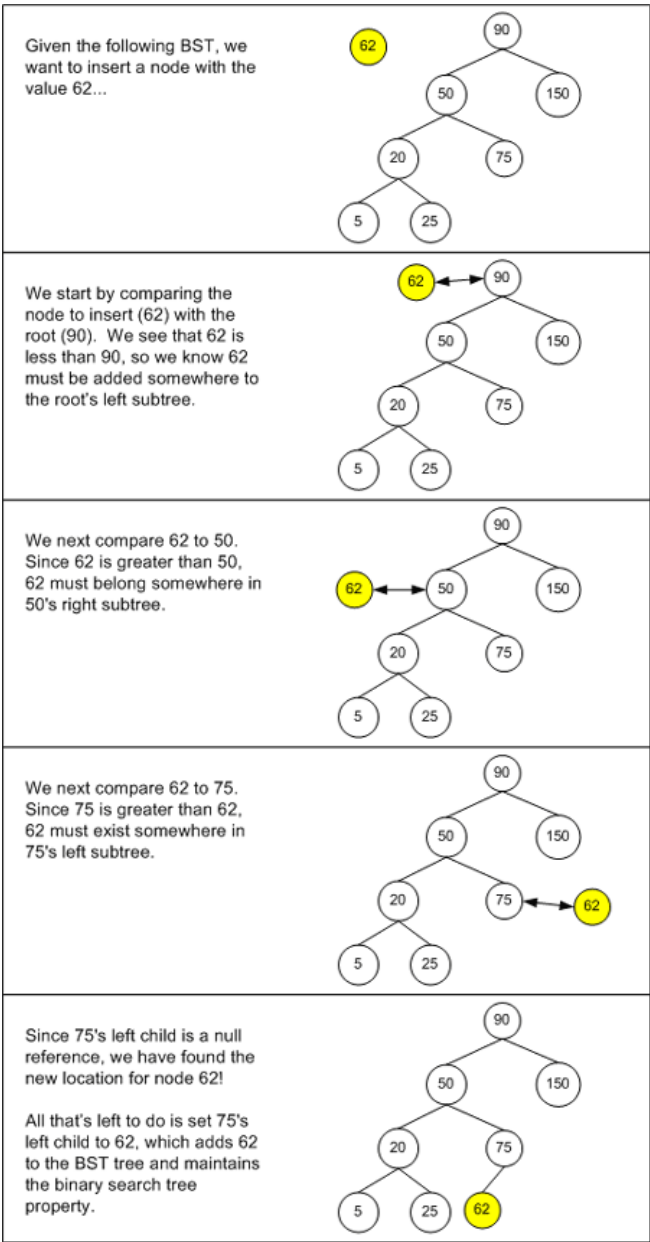
插入节点

我们不仅需要了解如何在二叉查找树中查找一个节点，还需要知道如何在二叉查找树中插入和删除一个节点。

当向树中插入一个新的节点时，该节点将总是作为叶子节点。所以，最困难的地方就是如何找到该节点的父节点。类似于查找算法中的描述，我们将这个新的节点称为节点 n ，而遍历的当前节点称为节点 c 。开始时，节点 c 为 BST 的根节点。则定位节点 n 父节点的步骤如下：

1. 如果节点 c 为空，则节点 c 的父节点将作为节点 n 的父节点。如果节点 n 的值小于该父节点的值，则节点 n 将作为该父节点的左孩子；否则节点 n 将作为该父节点的右孩子。
2. 比较节点 c 与节点 n 的值。
3. 如果节点 c 的值与节点 n 的值相等，则说明用户在试图插入一个重复的节点。解决办法可以是直接丢弃节点 n ，或者可以抛出异常。
4. 如果节点 n 的值小于节点 c 的值，则说明节点 n 一定是在节点 c 的左子树中。则将父节点设置为节点 c ，并将节点 c 设置为节点 c 的左孩子，然后返回至第 1 步。
5. 如果节点 n 的值大于节点 c 的值，则说明节点 n 一定是在节点 c 的右子树中。则将父节点设置为节点 c ，并将节点 c 设置为节点 c 的右孩子，然后返回至第 1 步。

当合适的节点找到时，该算法结束。从而使新节点被放入 BST 中成为某一父节点合适的孩子节点。



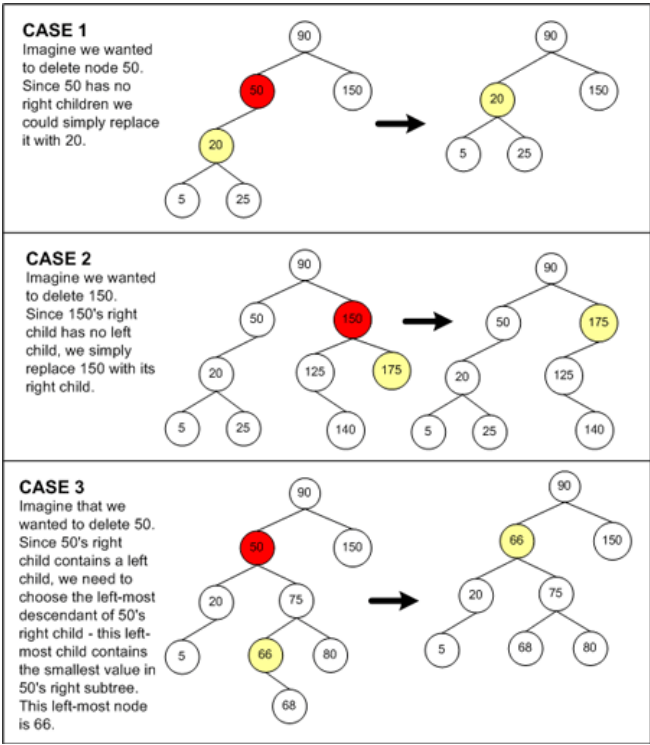
BST 的插入算法的复杂度与查找算法的复杂度是一样的：最佳情况是 $O(\log_2 n)$ ，而最坏情况是 $O(n)$ 。因为它们对节点的查找定位策略是相同的。

删除节点

从 BST 中删除节点比插入节点难度更大。因为删除一个非叶子节点，就必须选择其他节点来填补因删除节点所造成的树的断裂。如果不选择节点来填补这个断裂，那么就违背了 BST 的性质要求。

删除节点算法的第一步是定位要被删除的节点，这可以使用前面介绍的查找算法，因此运行时间为 $O(\log_2 n)$ 。接着应该选择合适的节点来代替删除节点的位置，它共有三种情况需要考虑。

- **情况 1:** 如果删除的节点没有右孩子，那么就选择它的左孩子来代替原来的节点。二叉查找树的性质保证了被删除节点的左子树必然符合二叉查找树的性质。因此左子树的值要么都大于，要么都小于被删除节点的父节点的值，这取决于被删除节点是左孩子还是右孩子。因此用被删除节点的左子树来替代被删除节点，是完全符合二叉搜索树的性质的。
- **情况 2:** 如果被删除节点的右孩子没有左孩子，那么这个右孩子被用来替换被删除节点。因为被删除节点的右孩子都大于被删除节点左子树的所有节点，同时也大于或小于被删除节点的父节点，这同样取决于被删除节点是左孩子还是右孩子。因此，用右孩子来替换被删除节点，符合二叉查找树的性质。
- **情况 3:** 如果被删除节点的右孩子有左孩子，就需要用被删除节点右孩子的左子树中的最下面的节点来替换它，就是说，我们用被删除节点的右子树中最小值的节点来替换。



我们知道，在 BST 中，最小值的节点总是在最左边，最大值的节点总是在最右边。因此替换被删除节点右子树中最小的一个节点，就保证了该节点一定大于被删除节点左子树的所有节点。同时，也保证它替代了被删除节点的位置后，它的右子树的所有节点值都大于它。因此这种选择策略符合二叉查找树的性质。

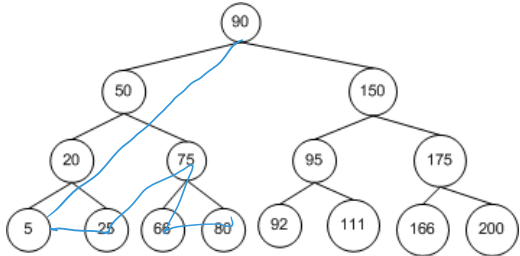
和查找、插入算法类似，删除算法的运行时间也与 BST 的拓扑结构有关，最佳情况是 $O(\log_2 n)$ ，而最坏情况是 $O(n)$ 。

遍历节点

对于线性的连续的数组来说，遍历数组采用的是单向的迭代法。从第一个元素开始，依次向后迭代每个元素。而 BST 则有三种常用的遍历方式：

- 前序遍历 (Perorder traversal)
- 中序遍历 (Inorder traversal)
- 后序遍历 (Postorder traversal)

当然，这三种遍历方式的工作原理是类似的。它们都是从根节点开始，然后访问其子节点。区别在于遍历时，访问节点本身和其子节点的顺序不同。




前序遍历 (Perorder traversal)

前序遍历从当前节点（节点 c）开始访问，然后访问其左孩子，再访问右孩子。开始时，节点 c 为 BST 的根节点。算法如下：

1. 访问节点 c;

2. 对节点 c 的左孩子重复第 1 步;

3. 对节点 c 的右孩子重复第 1 步;


- 则上图中树的遍历结果为：90, 50, 20, 5, 25, 75, 66, 80, 150, 95, 92, 111, 175, 166, 200。


中序遍历 (Inorder traversal)

中序遍历是从当前节点（节点 c）的左孩子开始访问，再访问当前节点，最后是其右节点。开始时，节点 c 为 BST 的根节点。算法如下：

1. 访问节点 c 的左孩子;

2. 对节点 c 重复第 1 步;

3. 对节点 c 的右孩子重复第 1 步。


- 则上图中树的遍历结果为：5, 20, 25, 50, 66, 75, 80, 90, 92, 95, 111, 150, 166, 175, 200。

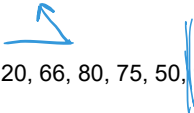
后序遍历 (Postorder traversal)

后序遍历首先从当前节点（节点 c）的左孩子开始访问，然后是右孩子，最后才是当前节点本身。开始时，节点 c 为 BST 的根节点。算法如下：

1. 访问节点 c 的左孩子;

2. 对节点 c 的右孩子重复第1 步;

3. 对节点 c 重复第 1 步;


- 则上图中树的遍历结果为：5, 25, 20, 66, 80, 75, 50, 92, 111, 95, 166, 200, 175, 150, 90。

参考资料

- *An Extensive Examination of Data Structures Using C# 2.0*
- 考察数据结构 - 第三部分：二叉树和BSTs[译]
- *Red-black tree*
- *Red/Black Tree Algorithm Visualization*
- *Left-Leaning Red-Black Trees*
- *Red-Black Trees*
- *Introduction to Algorithms : LECTURE 10 Balanced Search Trees*
- 教你透彻了解红黑树

本文《二叉查找树》由 Dennis Gao 发表自博客园博客，任何未经作者本人允许的人为或爬虫转载均为耍流氓。

 博客园

App开发者高效成长

 增长变现闭环

 收入提升 28%

立即注册



编辑推荐：
· 技术管理进阶——管人还是管事？

- 以终为始：如何让你的开发符合预期
- 五个维度打造研发管理体系
- 不会SQL也能做数据分析？浅谈语义解析领域的机会与挑战
- Spring IoC Container 原理解析

最新新闻：

- 宁德时代的鱿鱼游戏（2021-10-21 22:38）
 - 法拉第未来中国前员工收到补薪！网友：贾跃亭还钱了（2021-10-21 22:33）
 - 登顶中国互联网新首富 对张一鸣意味着什么（2021-10-21 22:32）
 - 蚂蚁反诈治理报告：已在全国100个县城、3200个社区开展防骗教育（2021-10-21 22:28）
 - "腾讯清理大师"等14款移动应用存在隐私不合规行为（2021-10-21 22:20）
- » 更多新闻...