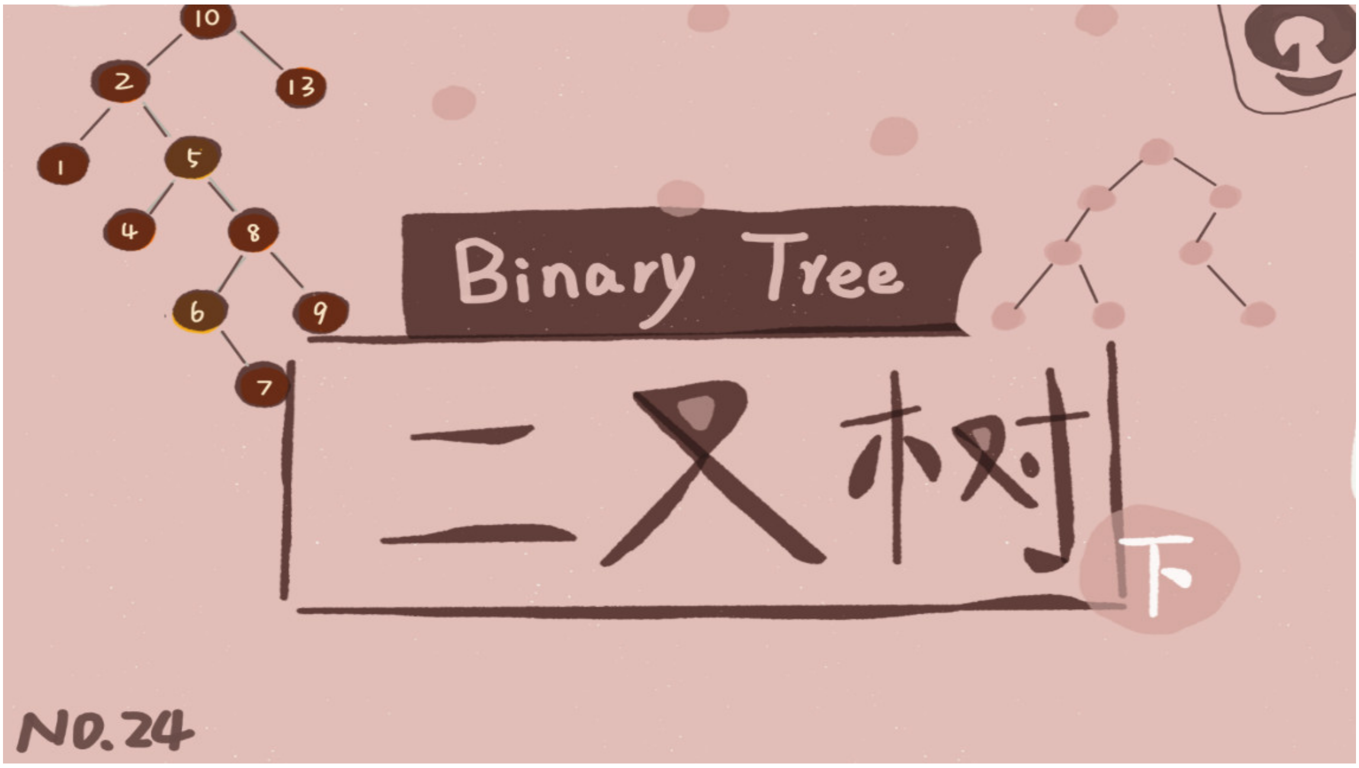
24 | 二叉树基础（下）：有了如此高效的散列表，为什么还需要二叉树?

2018-11-14 王争



24 | 二叉树基础（下）：有了如此高效的散列表，为什么还需要二叉树?
朗读人：修阳 12′21′′ | 5.66M

上一节我们学习了树、二叉树以及二叉树的遍历，今天我们再来学习一种特殊的的二叉树，二叉查找树。二叉查找树最大的特点就是，支持动态数据集合的快速插入、删除、查找操作。

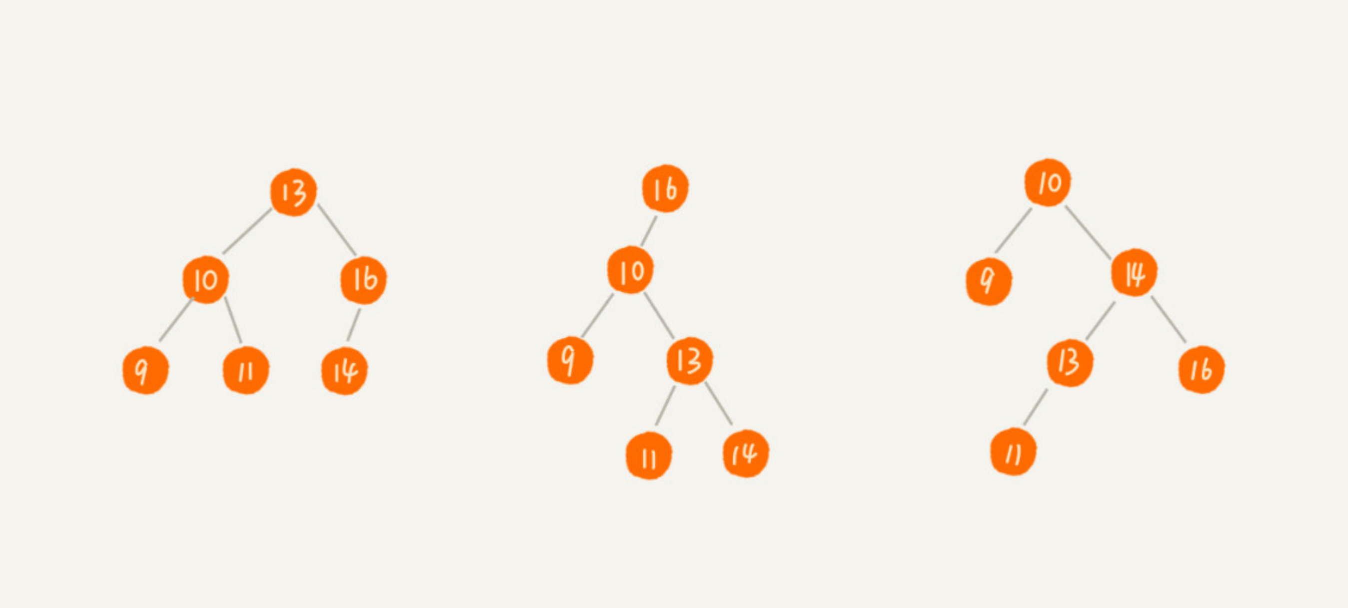
我们之前说过，散列表也是支持这些操作的，并且散列表的这些操作比二叉查找树更高效，时间复杂度是 O(1)。**既然有了这么高效的散列表，使用二叉树的地方是不是都可以替换成散列表呢? 有没有哪些地方是散列表做不了，必须要用二叉树来做的呢?**

带着这些问题，我们就来学习今天的内容，二叉查找树!

二叉查找树（Binary Search Tree）

二叉查找树是二叉树中最常用的一种类型，也叫二叉搜索树。顾名思义，二叉查找树是为了实现快速查找而生的。不过，它不仅仅支持快速查找一个数据，还支持快速插入、删除一个数据。它是怎么做到这些的呢?

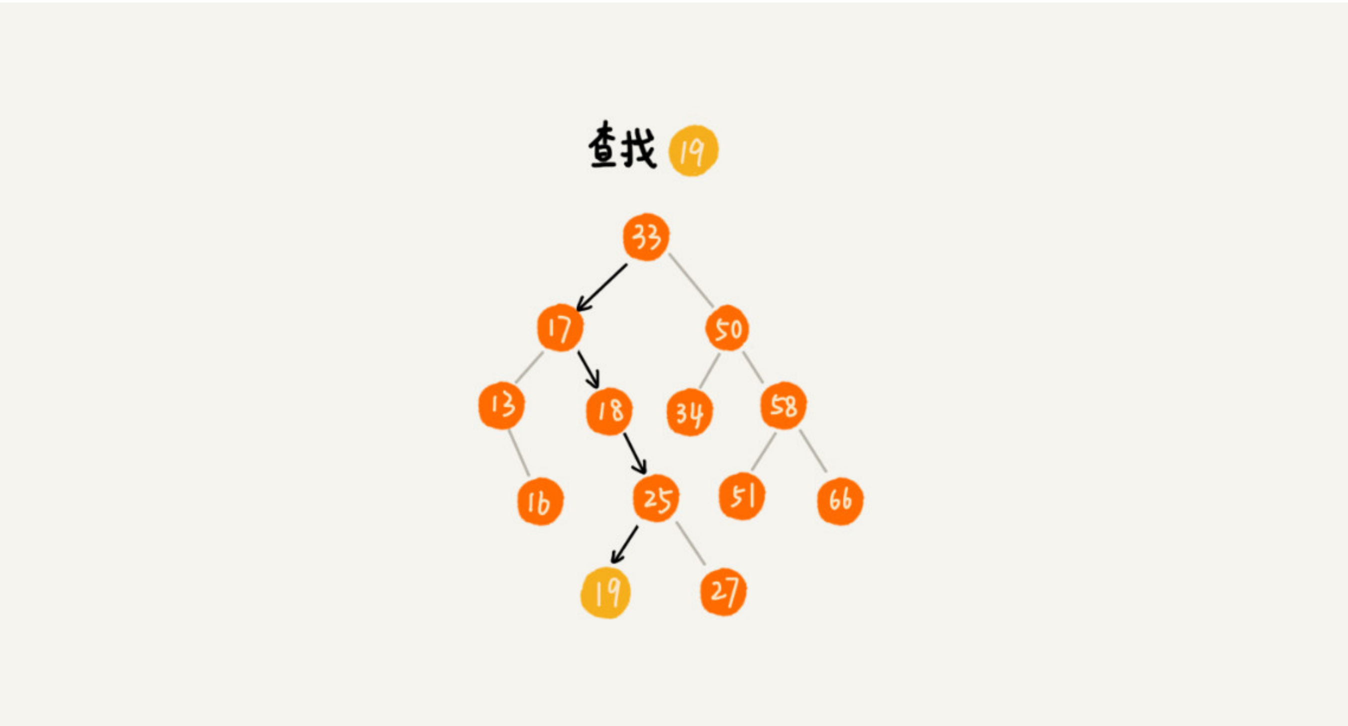
这些都依赖于二叉查找树的特殊结构。**二叉查找树要求，在树中的任意一个节点，其左子树中的每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值。** 我画了几个二叉查找树的例子，你一看应该就清楚了。



前面我们讲到，二叉查找树支持快速查找、插入、删除操作，现在我们就依次来看下，这三个操作是如何实现的。

1. 二叉查找树的查找操作

首先，我们看如何在二叉查找树中查找一个节点。我们先取根节点，如果它等于我们要查找的数据，那就返回。如果要查找的数据比根节点的值小，那就在左子树中递归查找；如果要查找的数据比根节点的值大，那就在右子树中递归查找。



这里我把查找的代码实现了一下，贴在下面了，结合代码，理解起来会更加容易。

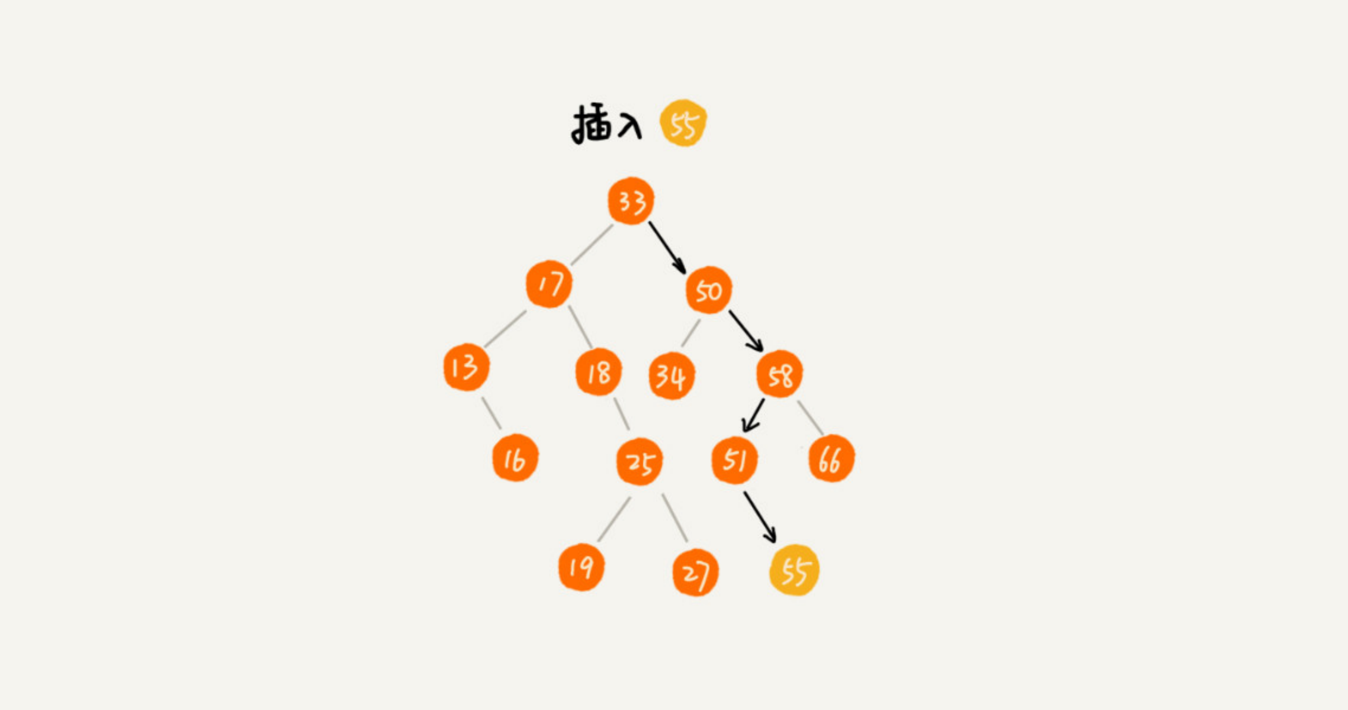
复制代码

```
public class BinarySearchTree {
  private Node tree;

  public Node find(int data) {
    Node p = tree;
    while (p != null) {
      if (data < p.data) p = p.left;
      else if (data > p.data) p = p.right;
      else return p;
    }
    return null;
  }

  public static class Node {
    private int data;
    private Node left;
    private Node right;

    public Node(int data) {
      this.data = data;
    }
  }
}
```

2. 二叉查找树的插入操作

二叉查找树的插入过程有点类似查找操作。新插入的数据一般都是在叶子节点上，所以我们只需要从根节点开始，依次比较要插入的数据和节点的大小关系。

如果要插入的数据比节点的数据大，并且节点的右子树为空，就将新数据直接插到右子节点的位置；如果不为空，就再递归遍历右子树，查找插入位置。同理，如果要插入的数据比节点数值小，并且节点的左子树为空，就将新数据插入到左子节点的位置；如果不为空，就再递归遍历左子树，查找插入位置。



同样，插入的代码我也实现了一下，贴在下面，你可以看看。

复制代码

```
public void insert(int data) {
  if (tree == null) {
    tree = new Node(data);
    return;
  }

  Node p = tree;
  while (p != null) {
    if (data > p.data) {
      if (p.right == null) {
        p.right = new Node(data);
        return;
      }
      p = p.right;
    } else { // data < p.data
      if (p.left == null) {
        p.left = new Node(data);
        return;
      }
      p = p.left;
    }
  }
}
```

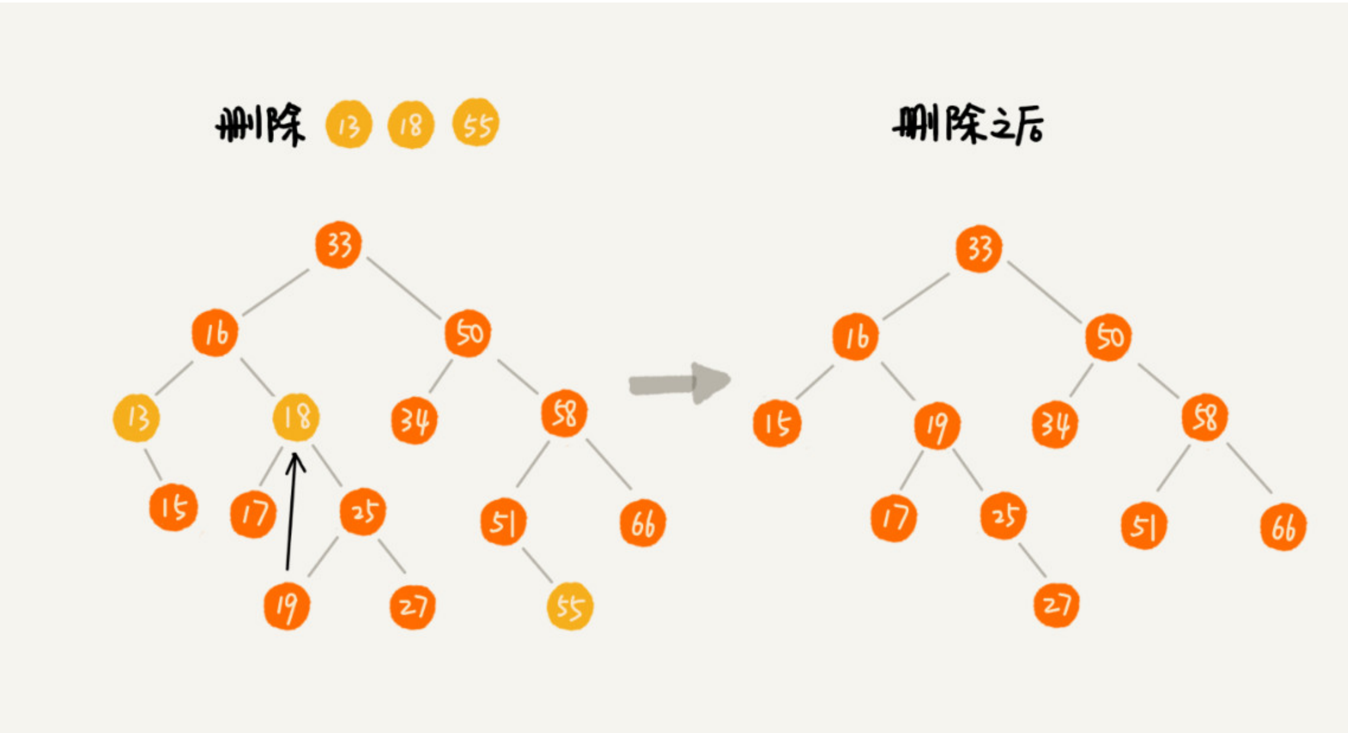
3. 二叉查找树的删除操作

二叉查找树的查找、插入操作都比较简单易懂，但是它的删除操作就比较复杂了 。针对要删除节点的子节点个数的不同，我们需要分三种情况来处理。

第一种情况是，如果要删除的节点没有子节点，我们只需要直接将父节点中，指向要删除节点的指针置为 null。比如图中的删除节点 55。

第二种情况是，如果要删除的节点只有一个子节点（只有左子节点或者右子节点），我们只需要更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点就可以了。比如图中的删除节点 13。

第三种情况是，如果要删除的节点有两个子节点，这就比较复杂了。我们需要找到这个节点的右子树中的最小节点，把它替换到要删除的节点上。然后再删除掉这个最小节点，因为最小节点肯定没有左子节点（如果有左子结点，那就不是最小节点了），所以，我们可以应用上面两条规则来删除这个最小节点。比如图中的删除节点 18。



老规矩，我还是把删除的代码贴在这里。

复制代码

```
public void delete(int data) {
  Node p = tree; // p 指向要删除的节点，初始化指向根节点
  Node pp = null; // pp 记录的是 p 的父节点
  while (p != null && p.data != data) {
    pp = p;
    if (data > p.data) p = p.right;
    else p = p.left;
  }
  if (p == null) return; // 没有找到

  // 要删除的节点有两个子节点
  if (p.left != null && p.right != null) { // 查找右子树中最小节点
    Node minP = p.right;
    Node minPP = p; // minPP 表示 minP 的父节点
    while (minP.left != null) {
```

```
      minPP = minP;
      minP = minP.left;
    }
    p.data = minP.data; // 将 minP 的数据替换到 p 中
    p = minP; // 下面就变成了删除 minP 了
    pp = minPP;
  }

  // 删除节点是叶子节点或者仅有一个子节点
  Node child; // p 的子节点
  if (p.left != null) child = p.left;
  else if (p.right != null) child = p.right;
  else child = null;

  if (pp == null) tree = child; // 删除的是根节点
  else if (pp.left == p) pp.left = child;
  else pp.right = child;
}
```

实际上，关于二叉查找树的删除操作，还有个非常简单、取巧的方法，就是单纯将要删除的节点标记为“已删除”，但是并不真正从树中将这个节点去掉。这样原本删除的节点还需要存储在内存中，比较浪费内存空间，但是删除操作就变得简单了很多。而且，这种处理方法也并没有增加插入、查找操作代码实现的难度。

4. 二叉查找树的其他操作

除了插入、删除、查找操作之外，二叉查找树中还可以支持**快速地查找最大节点和最小节点、前驱节点和后继节点**。这些操作我就不一一展示了。我会将相应的代码放到 GitHub 上，你可以自己先实现一下，然后再去上面看。

二叉查找树除了支持上面几个操作之外，还有一个重要的特性，就是**中序遍历二叉查找树，可以输出有序的数据序列，时间复杂度是 O(n)，非常高效**。因此，二叉查找树也叫作二叉排序树。

支持重复数据的二叉查找树

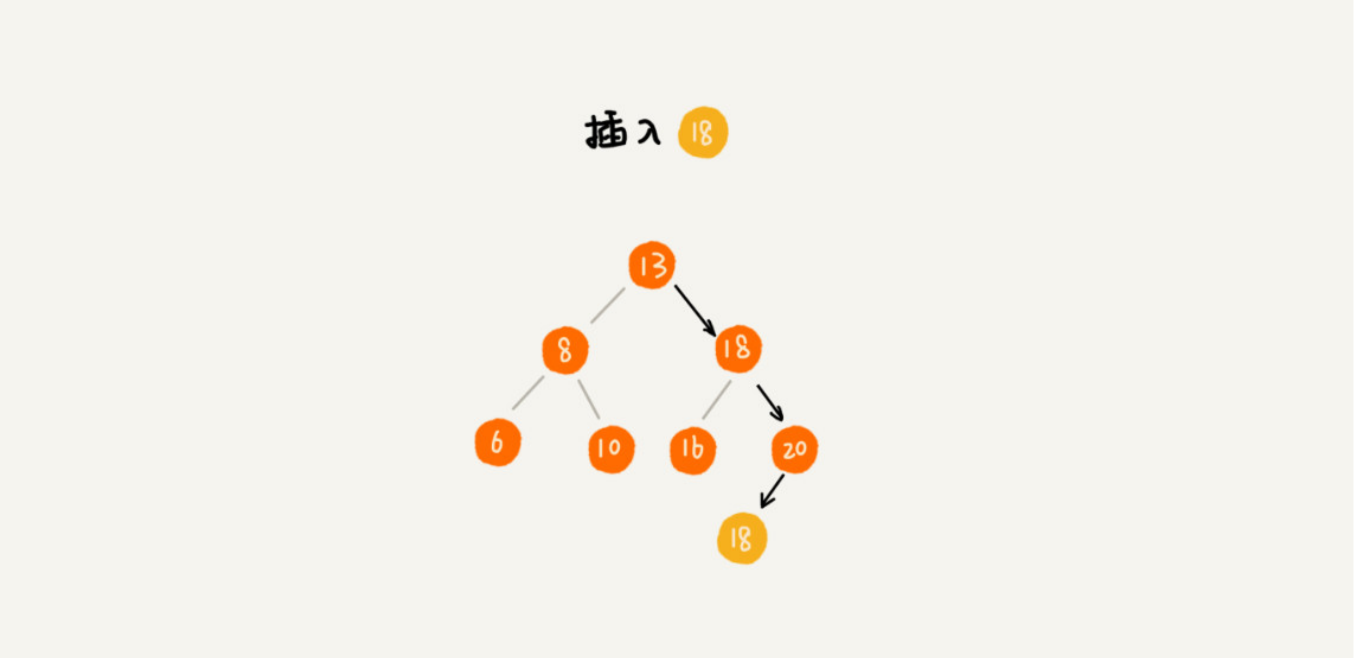
前面讲二叉查找树的时候，我们默认树中节点存储的都是数字。很多时候，在实际的软件开发中，我们在二叉查找树中存储的，是一个包含很多字段的对象。我们利用对象的某个字段作为键值（key）来构建二叉查找树。我们把对象中的其他字段叫作卫星数据。

前面我们讲的二叉查找树的操作，针对的都是不存在键值相同的情况。那如果存储的两个对象键值相同，这种情况该怎么处理呢？我这里有两种解决方法。

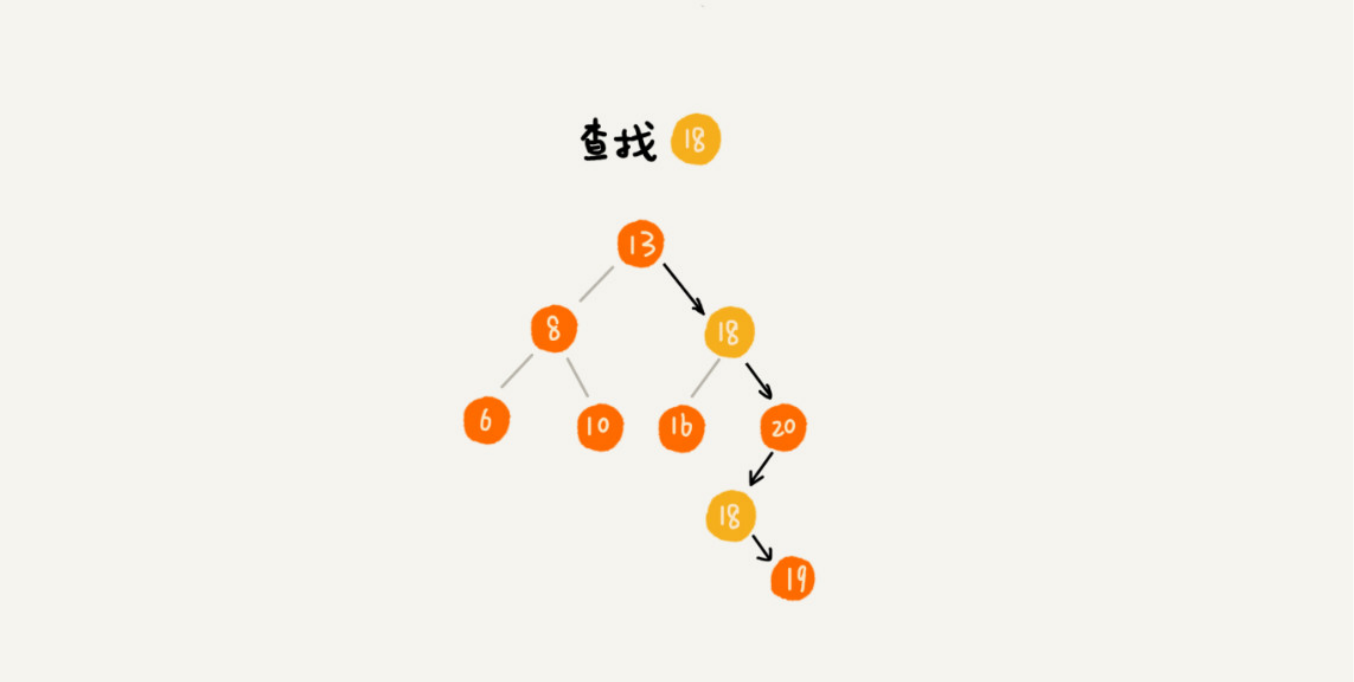
第一种方法比较容易。二叉查找树中每一个节点不仅会存储一个数据，因此我们通过链表和支持动态扩容的数组等数据结构，把值相同的数据都存储在同一个节点上。

第二种方法比较不好理解，不过更加优雅。

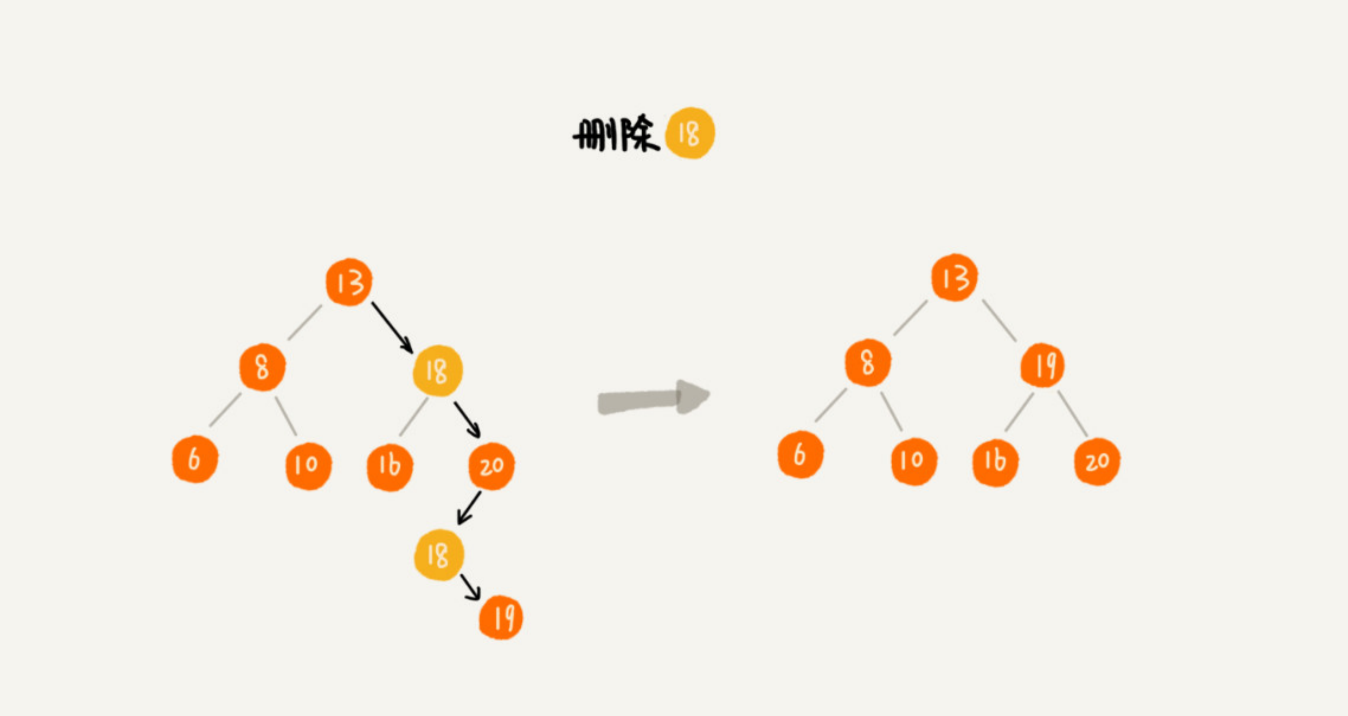
每个节点仍然只存储一个数据。在查找插入位置的过程中，如果碰到一个节点的值，与要插入数据的值相同，我们就将这个要插入的数据放到这个节点的右子树，也就是说，把这个新插入的数据当作大于这个节点的值来处理。



当要查找数据的时候，遇到值相同的节点，我们并不停止查找操作，而是继续在右子树中查找，直到遇到叶子节点，才停止。这样就可以把键值等于要查找值的所有节点都找出来。



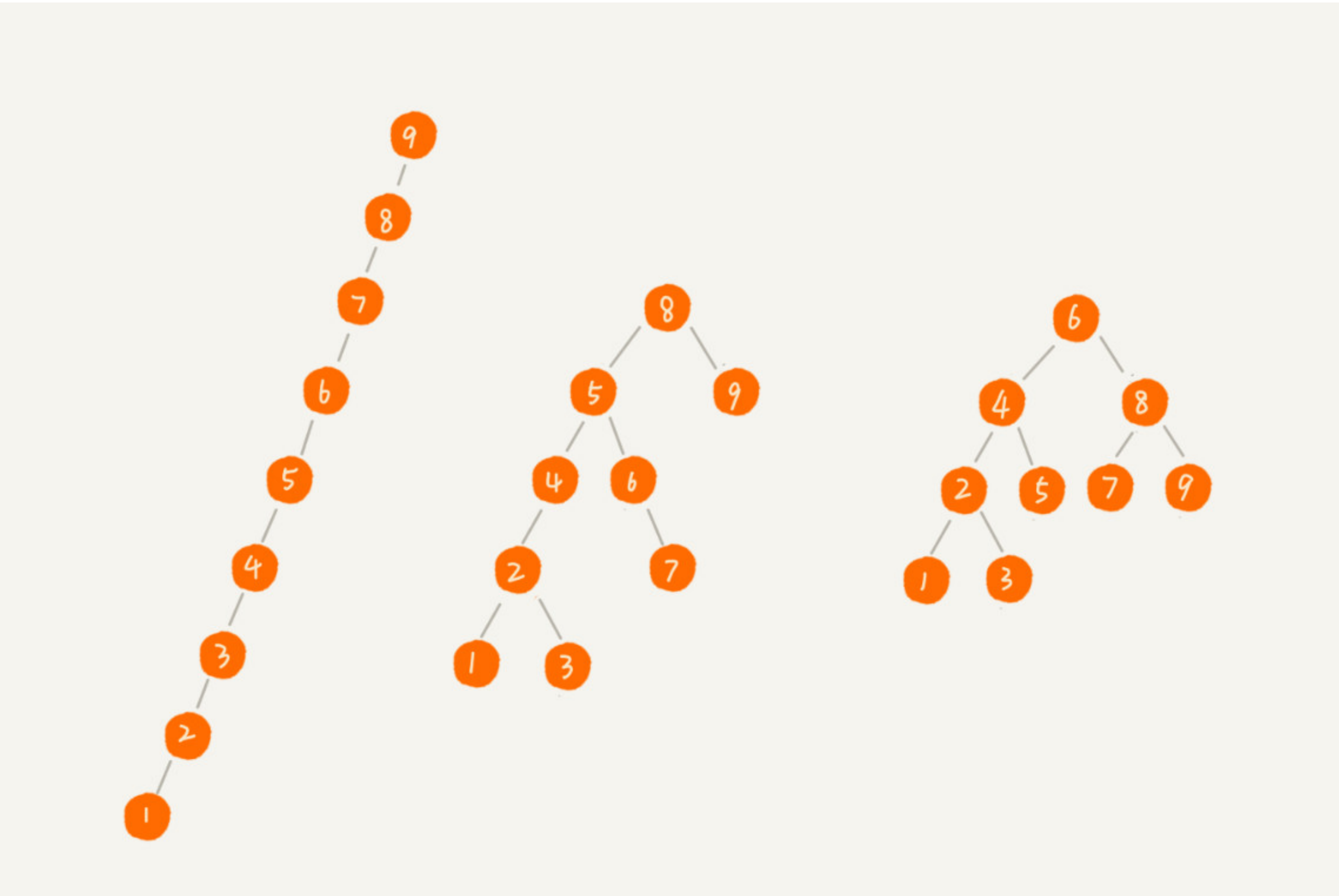
对于删除操作，我们也需要先查找到每个要删除的节点，然后再按前面讲的删除操作的方法，依次删除。



二叉查找树的时间复杂度分析

好了，对于二叉查找树常用操作的实现方式，你应该掌握得差不多了。现在，我们来分析一下，二叉查找树的插入、删除、查找操作的时间复杂度。

实际上，二叉查找树的形态各式各样。比如这个图中，对于同一组数据，我们构造了三种二叉查找树。它们的查找、插入、删除操作的执行效率都是不一样的。图中第一种二叉查找树，根节点的左右子树极度不平衡，已经退化成了链表，所以查找的时间复杂度就变成了 O(n)。



我刚刚其实分析了一种最糟糕的情况，我们现在来分析一个最理想的情况，二叉查找树是一棵完全二叉树（或满二叉树）。这个时候，插入、删除、查找的时间复杂度是多少呢？

从我前面的例子、图，以及还有代码来看，不管操作是插入、删除还是查找，**时间复杂度其实都跟树的高度成正比，也就是 O(height)**。既然这样，现在问题就转变成另外一个了，也就是，如何求一棵包含 n 个节点的完全二叉树的高度？

树的高度就等于最大层数减一，为了方便计算，我们转换成层来表示。从图中可以看出，包含 n 个节点的完全二叉树中，第一层包含 1 个节点，第二层包含 2 个节点，第三层包含 4 个节点，依次类推，下面一层节点个数是上一层的 2 倍，第 K 层包含的节点个数就是 2^(K-1)。

不过，对于完全二叉树来说，最后一层的节点个数有点儿不遵守上面的规律了。它包含的节点个数在 1 个到 2^(L-1) 个之间（我们假设最大层数是 L）。如果我们把每一层的节点个数加起来就是总的节点个数 n。也就是说，如果节点的个数是 n，那么 n 满足这样一个关系：

复制代码

```
n >= 1+2+4+8+...+2^(L-2)+1
n <= 1+2+4+8+...+2^(L-2)+2^(L-1)
```

借助等比数列的求和公式，我们可以计算出，L 的范围是 [$\log_2(n+1)$, $\log_2 n$ +1]。完全二叉树的层数小于等于 $\log_2 n$ +1，也就是说，完全二叉树的高度小于等于 $\log_2 n$。

显然，极度不平衡的二叉查找树，它的查找性能肯定不能满足我们的需求。我们需要构建一种不管怎么删除、插入数据，在任何时候，都能保持任意节点左右子树都比较平衡的二叉查找树，这就是我们下一节课要详细讲的，一种特殊的二叉查找树，平衡二叉查找树。平衡二叉查找树的高度接近 logn，所以插入、删除、查找操作的时间复杂度也比较稳定，是 O(logn)。

解答开篇

我们在散列表那节中讲过，散列表的插入、删除、查找操作的时间复杂度可以做到常量级的 O(1)，非常高效。而二叉查找树在比较平衡的情况下，插入、删除、查找操作时间复杂度才是 O(logn)，相对散列表，好像并没有什么优势，那我们为什么还要用二叉查找树呢？

我认为有下面几个原因：

第一，散列表中的数据是无序存储的，如果要输出有序的数据，需要先进行排序。而对于二叉查找树来说，我们只需要中序遍历，就可以在 O(n) 的时间复杂度内，输出有序的数据序列。

第二，散列表扩容耗时很多，而且当遇到散列冲突时，性能不稳定，尽管二叉查找树的性能不稳定，但是在工程中，我们最常用的平衡二叉查找树的性能非常稳定，时间复杂度稳定在 O(logn)。

第三，笼统地来说，尽管散列表的查找等操作的时间复杂度是常量级的，但因为哈希冲突的存在，这个常量不一定比 logn 小，所以实际的查找速度可能不一定比 O(logn) 快。加上哈希函数的耗时，也不一定就比平衡二叉查找树的效率高。

第四，散列表的构造比二叉查找树要复杂，需要考虑的东西很多。比如散列函数的设计、冲突解决办法、扩容、缩容等。平衡二叉查找树只需要考虑平衡性这一个问题，而且这个问题的解决方案比较成熟、固定。

最后，为了避免过多的散列冲突，散列表装载因子不能太大，特别是基于开放寻址法解决冲突的散列表，不然会浪费一定的存储空间。

综合这几点，平衡二叉查找树在某些方面还是优于散列表的，所以，这两者的存在并不冲突。我们在实际的开发过程中，需要结合具体的需求来选择使用哪一个。

内容小结

今天我们学习了一种特殊的二叉树，二叉查找树。它支持快速地查找、插入、删除操作。

二叉查找树中，每个节点的值都大于左子树节点的值，小于右子树节点的值。不过，这只是针对没有重复数据的情况。对于存在重复数据的二叉查找树，我介绍了两种构建方法，一种是让每个节点存储多个值相同的数据；另一种是，每个节点中存储一个数据。针对这种情况，我们只需要稍加改造原来的插入、删除、查找操作即可。

在二叉查找树中，查找、插入、删除等很多操作的时间复杂度都跟树的高度成正比。两个极端情况的时间复杂度分别是 O(n) 和 O(logn)，分别对应二叉树退化成链表的情况和完全二叉树。

为了避免时间复杂度的退化，针对二叉查找树，我们又设计了一种更加复杂的树，平衡二叉查找树，时间复杂度可以做到稳定的 O(logn)，下一节我们具体来讲。

课后思考

今天我讲了二叉树高度的理论分析方法，给出了粗略的数量级。如何通过编程，求出一棵给定二叉树的确切高度呢？

欢迎留言和我分享，我会第一时间给你反馈。

我已将本节内容相关的详细代码更新到 GitHub，戳此即可查看。

上一篇 23 | 二叉树基础（上）：什么样的二叉树适合用数组来存储?

写留言

精选留言

失火的夏天 29

确定二叉树高度有两种思路：第一种是深度优先思想的递归，分别求左右子树的高度。当前节点的高度就是左右子树中较大的那个+1；第二种可以采用层次遍历的方式，每一层记录都记录下当前队列的长度，这个是队尾，每一层队头从0开始。然后每遍历一个元素，队头下标+1。直到队头下标等于队尾下标。这个时候表示当前层遍历完成。每一层刚开始遍历的时候，树的高度+1。最后队列为空，就能得到树的高度。

2018-11-14

作者回复

👍 大家可以看看这条留言

2018-11-14

拉欧 8

递归法，根节点高度=max(左子树高度，右子树高度)+1

2018-11-14

作者回复

👍 精髓

2018-11-14

莫弹弹 4

在sf的微信公众号上刚好看到二叉树相关的文章，二叉树常规操作都有了，基本思路是：

- 只有一个根结点时，二叉树深度为 1
- 只有左子树时，二叉树深度为左子树深度加 1
- 只有右子树时，二叉树深度为右子树深度加 1
- 同时存在左右子树时，二叉树深度为左右子树中深度最大者加 1

https://mp.weixin.qq.com/s/ONKJyusGCIE2ctwT9uLv9g

2018-11-14

作者回复

👍

2018-11-14

一般社员 2

老师，不理解删除有两个子节点那段代码，最后删除minp，不是minpp.left =null,minp =null吗

2018-11-14

等风来 1

老师:删除示例的25节点的右节点[21]错误;
删除节点有两个节点
p = minP; // 下面就变成了删除 minP 了...
pp = minPP;
是不是应该改成: minPP.Left = minP.Right;

2018-11-14

Ricky 0

老师，请问二叉搜索树如果删除仅设置状态的话，难道删除节点后树的结构一直保持不变吗?那插入数值时该怎么插呢，如果遍历到已删除节点，那直接继续往下遍历吗？或者当删除节点是叶子节点，插入位置正好在该节点，直接替换?

2018-11-14

张远 0

怎么没有python代码呢老师

2018-11-14

JerryLuo 0

为什么我算出来的L的范围是[log(n+1) +1, logn +2]，找不出哪里算错了，老师可以贴下计算步骤吗?

2018-11-14

三个石头 0

应该只能遍历一遍吧

2018-11-14

qpm 0

hi，老师。一直都每天在专栏里学习，我希望可以向你提点课程设计上的建议。
算法的学习过程整体来说还是由浅入深的，从线性结构到非线性结构，从树概念深入学习二叉树等等，我觉得文章末尾的习题可以有一道和下一篇文章有所关联的问题，方便我们思考过后可以更容易地学习下一篇文章，也算是一个链表的思维方式。
专栏至此非常有用，深入浅出，谈及了很多算法书本上没说到的点。感谢老师

2018-11-14

计科一班 0

老师你好，对于删除两个节点的最后两行代码：p = minP; pp = minPP;
不是很理解这两句意思，说是删除，但是不太明白，能否解释一下。

2018-11-14

牵手约定 0

有点懵了，

2018-11-14

Sharry 0

```
template<typename T>
int getTreeHeight(TreeNode<T> *node) {
if (node == NULL) {
return -1;
}
int leftHeight = getTreeHeight(node->left);
int rightHeight = getTreeHeight(node->right);
return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}
```

2018-11-14

thsai 0

删除的第三种情况里，如果找到最小的叶子节点是有右子节点的呢，比如删除前的那棵树中删除50，此时找到最小叶子节点是51，那55这个节点怎么办？代码里面好像没有体现出来

2018-11-14

徐凯 0

```
int ret_height（treenode* T)
 {
if（! T)
return 0；
int Lh=0,Rh=0;
```

if（T->left)
Lh=ret_height(T->left);
If(T->right)
Rh =ret_height(T->right);
return max(Lh,Rh)+1;
}
刚起床写的一段 不知道漏没漏东西。zz

2018-11-14


王小李 0
为什么第二种实现更优雅?

2018-11-14

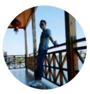

陈毓飞 0
我觉得不需要定义内部类Node，BinarySearchTree本身就是Node。这样既能体现出BinarySearchTree的左右子树也都是BinarySearchTree，又能方便做一些递归操作。

```
public class BinarySearchTree {
private int value;
private BinarySearchTree left;
private BinarySearchTree right;


...
}
```

2018-11-14


Ryan-Hou 0
平衡树相比于哈希表，保存了节点数据间的顺序信息，所以操作的时间复杂度上会比哈希表大(因为额外的提供了顺序性，对应的会有代价)。也正因为保存了顺序性，平衡树可以方便的实现min, max, ceil, floor 等操作，所以个人认为这两种数据结构最大的不同在于这里，有不同的取舍

2018-11-14


Dream. 0
是否可以当标记删除的节点到了一定的阀值时，再一次性删除?
一次性删除又可能会导致在某次触发删除时会非常慢。
是否可以在每次查询的时候，顺带将标记已删除的节点给删除掉?
但是这样的操作，与直接删除元素相比，还影响查询速度。
所以只能舍弃空间换取时间了吗?

2018-11-14


杨伟 0

遍历一遍

2018-11-14