

1 修改后 remove 函数实现的阐述

首先，若对象是空树，则直接 return 其次，若对象比当前节点小，则继续向左找，比当前大，向右找。找到对象后，如果该待删除对象只有一侧子树甚至没有子树，都很容易实现，按照原版代码即可。我们只关心当待删除对象的左右子树均非空时的情况。

此时，需要先把它的继任者找出来，这个找继任者的操作就在 detachMin 里完成。

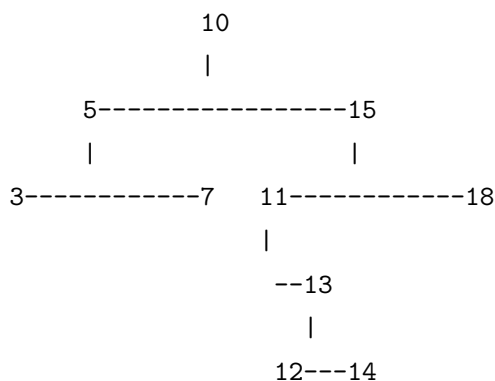
detachMin 负责传回将继任者找到并且无痕地取出来，由于继任者要求是右子树的最小者，一路往左就行。找到继任者后，要将继任者从树上剥离下来，但继任者可能也有子辈，于是将其子辈接回断的那根枝，即：继任者子辈和继任者父辈相接，别忘了先要弄个指针盯着这个继任者（否则就弄丢了），该指针值也作为参数传出去，被 remove 函数中的变量接收。

现在 remove 函数拿到了剥离下来的继任者，只需要做善后工作：把删除对象的左枝右枝告诉继任者，把指向删除对象的指针改成指向继任者，然后把删除对象 delete 即可。

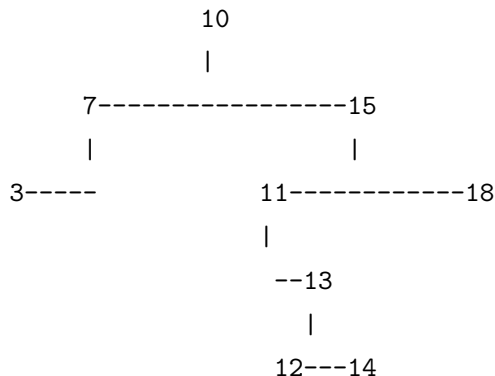
2 测试程序的设计思路

先测试正常的情况（即用户的使用不会触发抛出异常警告），为了验证 remove 正确操作，需要验证这些情况：1 删除对象不存在 2 删除对象恰好是叶节点 3 删除对象的继任者恰好是叶节点 4 删除对象的继任者还有后继 5 对一个空树进行删除

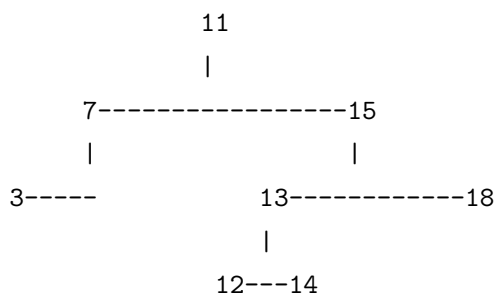
1. 我首先 insert 了一些更加多的元素，使得树变得更加复杂，最终，形成的 BST 树如下：



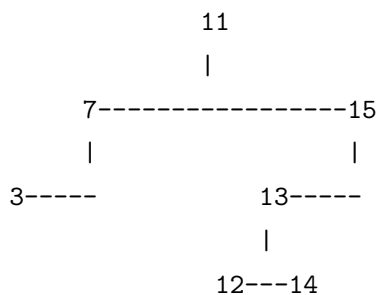
2. 接着，进行 remove 操作，首先测试普通的 remove(5)，对应的是：“接替删除对象的那个节点”恰好是叶子的情况，理想结果是成为树：



- 接着，测试 remove 一个不存在的数据 6，希望的结果是什么都不发生。
- 然后，测试 remove 根节点 10，预期的结果是，由右子树的最小者（11）来接替成为根节点，然后 11 的子辈被 11 的父辈接管，即：



- 再测试被删除对象恰好就是叶节点的情况，remove(18):



- 最后测试对一棵空树进行 remove，是否会出问题。理想情况是什么都不做，打印该树，仍然输出 “empty tree”

3 测试的结果

在工作空间下使用命令 `make run`，输出将会打印到 terminal 中，如下

Initial Tree:

3
5
7
10
11
12
13
14
15
18

Minimum element: 3

Maximum element: 18

Contains 7? Yes

Contains 20? No

Tree after removing 5:

3

7

10

11

12

13

14

15

18

Tree after removing 6:

3

7

10

11

12

13

14

15

18

Tree after removing 10:

3

7

11

12

13

14

15

18

Tree after removing 18:

3

7

11

12

13

14

15

Tree after making empty:

Empty tree

Is tree empty? Yes

Empty Tree after removing 7:

Empty tree

Copied Tree (bst3):

1

2

3

Assigned Tree (bst4):

1

2

3

Moved Tree (bst5):

1

2

3

Move Assigned Tree (bst6):

1

2

3

我用 `valgrind -leak-check=full ./test_BST` 进行测试，发现没有发生内存泄露：ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

4 异常测试报告

将最后一行代码解注释，即测试对一棵空树进行 `bst7.findMax()`；测试，将会触发 `throw UnderflowException{ }`