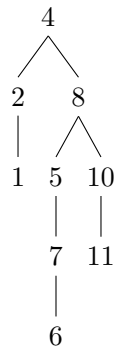


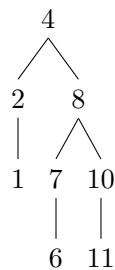
1 remove 函数的实现思路

我们使用一颗具体的二叉树来说明我们是如何实现 `remove()` 函数的。



按照提示，我们先构造了一个 `detachMin()` 函数用以查找以 `t` 为根的子树中的最小节点，返回这个节点，并从原子树中删除这个节点。显然，当要删除的节点具有两个子树时，通过这个函数返回的右子树最小节点将代替被删除节点。具体而言，我们如果需要删除以 8 为根的子树中的最小节点，`detachMin()` 函数会返回 5，并将 7 接到 8 的左节点处。

此时二叉树变成



在理解了 `detachMin()` 的功能后，我们可以实现 `remove()` 的功能了，具体实现如下：

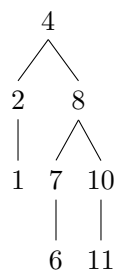
```

1  BinaryNode *detachMin (BinaryNode *&t)
2  {
3  if (t == nullptr) return nullptr;
4  if (t->left == nullptr)
5  { BinaryNode *minNode = t;
6    t = t->right;
7    return minNode;
8  }
9  return detachMin(t->left);
10 }
11 void remove (const Comparable &x, BinaryNode *&t)
12 {
13 if (t == nullptr) return;
14 if (x > t->element) remove(x, t->right);
15 else if (x < t->element) remove(x, t->left);
16 else
17 {
18   if (t->left == nullptr)

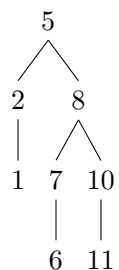
```

```
19     {
20         BinaryNode *oldNode = t;
21         t = t->right;
22         delete oldNode;
23     }
24     else if (t->right == nullptr)
25     {
26         BinaryNode *oldNode = t;
27         t = t->left;
28         delete oldNode;
29     }
30     else
31     {
32         BinaryNode *minNode = detachMin(t->right);
33         minNode->left = t->left;
34         minNode->right = t->right;
35         delete t;
36         t = minNode;
37     }
38 }
39 }
```

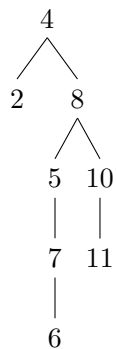
例如，当我们对前文的二叉树运行 `remove(4)` 时，`detachMin()` 函数先将二叉树化为



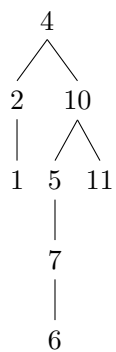
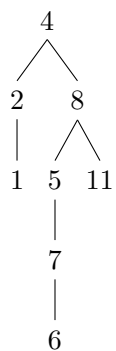
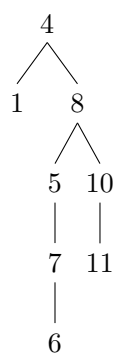
然后剩余的代码用 5 替代了根节点 4. 从而二叉树最后变成



这仍然是一个 BST. 如果我们最初选择删除叶子 1, 则二叉树变为



类似的，在最初分别选择删除 2（只有左子节点的节点）、10（只有右子节点的节点）8（有左右子节点的节点）后得到的二叉树分别为



2 测试程序

刚才的图示用代码表示如下

```
1 void testBinarySearchTree() {  
2     BinarySearchTree<int> bst;
```

```
3
4     bst.insert(4);
5     bst.insert(2);
6     bst.insert(8);
7     bst.insert(5);
8     bst.insert(1);
9     bst.insert(7);
10    bst.insert(6);
11    bst.insert(10);
12    bst.insert(11);
13    bst.printTree();
14    BinarySearchTree<int> bst_1 = bst;
15    BinarySearchTree<int> bst_2 = bst;
16    BinarySearchTree<int> bst_3 = bst;
17    BinarySearchTree<int> bst_4 = bst;
18    BinarySearchTree<int> bst_5 = bst;
19    BinarySearchTree<int> bst_6;
20
21    std::cout << "Delete a node with no child : " << std::endl;
22    bst_1.remove(1);
23    bst_1.printTree();
24
25    std::cout << "Delete a node with a left child : " << std::endl;
26    bst_2.remove(2);
27    bst_2.printTree();
28
29    std::cout << "Delete a node with a right child : " << std::endl;
30    bst_3.remove(10);
31    bst_3.printTree();
32
33    std::cout << "Delete a node with both right and left child : "<< std::endl;
34    bst_4.remove(8);
35    bst_4.printTree();
36    std::cout << "Delete a node twice : "<< std::endl;
37    bst_4.remove(8);
38    bst_4.printTree();
39
40    std::cout << "Delete a nonexisting node : "<< std::endl;
41    bst.remove(111);
42    bst.printTree();
43
44    std::cout << "Delete a root node : "<< std::endl;
45    bst_5.remove(4);
46    bst_5.printTree();
```

```
47
48     std::cout << "Delete a node of an empty tree : " << std::endl;
49     std::cout << "Before deletion : " << std::endl;
50     bst_6.printTree();
51     std::cout << "After deletion : " << std::endl;
52     bst_6.remove(1);
53     bst_6.printTree();
54 }
```

可以看到，我们还额外测试了对空树 `remove()` 的结果。

3 测试的结果

编译后，结果如下：

```
1  1
2  2
3  4
4  5
5  6
6  7
7  8
8  10
9  11
10 Delete a node with no child :
11  2
12  4
13  5
14  6
15  7
16  8
17  10
18  11
19 Delete a node with a left child :
20  1
21  4
22  5
23  6
24  7
25  8
26  10
27  11
28 Delete a node with a right child :
29  1
30  2
```

```
31 4
32 5
33 6
34 7
35 8
36 11
37 Delete a node with both right and left child :
38 1
39 2
40 4
41 5
42 6
43 7
44 10
45 11
46 Delete a node twice :
47 1
48 2
49 4
50 5
51 6
52 7
53 10
54 11
55 Delete a nonexisting node :
56 1
57 2
58 4
59 5
60 6
61 7
62 8
63 10
64 11
65 Delete a root node :
66 1
67 2
68 5
69 6
70 7
71 8
72 10
73 11
74 Delete a node of an empty tree :
```

75 Before deletion :

76 Empty tree

77 After deletion :

78 Empty tree

符合预期结果.