BST_remove 的改善

汉铮陈

2025 年 1 月 26 日

1 remove 函数的完善

这与部分主要分为三个主要内容：辅助函数 detachMin(BinaryNode* &t) 的构造、remove(const Comparable &x, BinaryNode * &t) 函数的完善、树的简单可视化打印 drawTree(BinaryNode* t ,int space , bool isLeft) 的实现。以下将上述函分别简写为 detachMin() 、remove() 、drawTree()

1.1 detchMin() 函数的构造

在这里构造辅助函数 detachMin() 的主要作用就是查找以 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。

该函数实现的基本原理与 findMin() 函数类似，只需要需要利用递归的方法找到右子树的最小节点，并构造一个临时节点记住该节点的值以及指针，最后把最小节点删掉即可。具体代码如下：

```
/**
 * @brief 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
 * @param t 当前节点指针
 * @return 以t 为根的子树中的最小节点
 */
BinaryNode* detachMin(BinaryNode *&t)
{
    if(!t) return nullptr;

    if(!t ->left)//判断是否为叶子节点
    {
        BinaryNode* minNode = t;
        t = t->right;
        return minNode;
    }
    else
    {
        return detachMin(t -> left);
    }
}
```

Listing 1: detachMin() 函数

而 findMin() 函数的代码如下：

```
BinaryNode *findMin(BinaryNode *t) const {
    /// 从一个空节点开始查找，返回空指针
```

```
    if (t == nullptr)
    {
        return nullptr;
    }
    /// 向左无路了，当前节点就是最小元素
    if (t->left == nullptr)
    {
        return t;
    }
    /// 否则继续向左查找
    return findMin(t->left);
}
```

Listing 2: findMin() 函数

1.2 remove() 函数的完善

要通过移动节点的方式实现 remove 操作总体分为四步：1. 找到要删除的节点 cur 和它的父节点 par; 2. 找到 cur 的右子树中的最小值节点 min , 并将其断开; 3. 将 min 的左子树和右子树分别链接到 cur 的左子树和右子树；4. 将 par 的指针指向 min，然后删除 cur。具体代码如下：

```
void remove(const Comparable &x, BinaryNode * &t)
{
    if(!t) return;//如果t为空节点

    BinaryNode* par = nullptr;
    BinaryNode* cur = t;
    while(cur && cur -> element != x )//找到值为x的节点以及其parent节点，注意条件顺序
    {
        par =cur;
        if(cur -> element < x )
        {
            cur = cur->right;
        }else if(cur -> element > x)
        {
            cur = cur->left;
        }
    }

    if(!cur) return;//x没有被找到

    if(!(cur ->left) || !(cur -> right))//要删除的节点至多只有一个child节点
    {
        if(!par)//如果删除根节点
        {
            t = (cur -> left)? cur->left : cur->right;//修改根节点
        }
        if(par -> left == cur)//t是par的左节点
        {
            par -> left = (cur -> left)? cur->left : cur->right;
        }else
```

```
        {
            par -> right = (cur -> left)? cur->left : cur->right;
        }
        delete cur;
    }else//有两个child节点
    {
        BinaryNode* min = detachMin(cur->right);

        min -> left = cur -> left;// 将当前节点的左子树链接到后继节点
        min -> right = cur -> right;// 将当前节点的右子树链接到后继节点
        if(!par)//如果删除根节点
        {
            t = min;//修改根节点
        }
        else if(par -> left == t)//t是par的左节点
        {
            par -> left = min;// 父节点指向后继节点
        }else//t是右节点
        {
            par -> right = min;// 父节点指向后继节点
        }
        delete cur;
    }
}
```

Listing 3: remove() 函数

1.3 树的可视化实现

在这里构造 drawTree() 函数主要是为了，便于直观的观察和跟踪 remove() 函数操作后的结果，这里打印的树的节点是纵向排列的。具体代码如下：

```
void drawTree(BinaryNode* node, int space , bool isLeft) {
    if (!node) return;

    space += 10; // 增加空间以便于显示

    // 先打印右子树
    drawTree(node->right, space , false);

    // 打印当前节点
    std::cout << std::endl;
    for (int i = 10; i < space; i++) {
        std::cout << ' '; // 输出空格
    }

    if (isLeft) {
        std::cout << " \\"; // 左子树的枝
    } else if (space > 10) {
        std::cout << " / "; // 右子树的枝
    }
```

```
20
21      std::cout << " " << node->element << "\n"; // 打印节点值
22
23      // 打印左子树
24      drawTree(node->left, space , true);
25  }
```

Listing 4: draw() 函数

2 测试程序的设计

这部分主要设计的是 test.cpp 部分，验证 remove() 函数的正确性。

首先构造一个 BST：

```
1   #include <iostream>
2   #include "BST.h"  // 假设 BinarySearchTree 类定义在这个头文件中
3   using namespace std;
4
5   int main() {
6       BinarySearchTree<int> bst;
7
8       // 测试插入功能
9       bst.insert(50);
10      bst.insert(30);
11      bst.insert(70);
12      bst.insert(20);
13      bst.insert(40);
14      bst.insert(60);
15      bst.insert(80);
16
17      std::cout << "before any deletions: ";
18      bst.drawTree();
19
20      // 删除叶子节点
21      cout << "\nDeleting leaf node (20): ";
22      bst.remove(20);
23      bst.drawTree();
24
25      // 删除只有一个子节点的节点
26      cout << "\nDeleting node with one child (30): ";
27      bst.remove(30);
28      bst.drawTree();
29
30      // 删除具有两个子节点的节点
31      cout << "\nDeleting node with two children (50): ";
32      bst.remove(50);
33      bst.drawTree();
34
35      cout<<endl;
36      // 测试清空树
37      bst.makeEmpty();
```

```
38      cout << "Tree after making empty:" << std::endl;
39      bst.printTree();
40
41      return 0;
42  }
```

Listing 5: testBinarySearchTree() 测试函数

构造树的输出结果如下:

```
before any deletions:
                    /  80

          /  70

                    \ 60

 50

                    /  40

          \ 30

                    \ 20
```

删除叶子节点后的树形状为:

```
Deleting leaf node (20):
                    /  80

          /  70

                    \ 60

 50

                    /  40

          \ 30
```

删除只有一个子节点的节点后，树的形状为:

```
Deleting node with one child (30):
                    /  80

          /  70
```

```
                    \ 60

 50

          \ 40
```

删除具有两个子节点的节点（这里是 root）后, 树的形状为:

```
Deleting node with two children (50):
                    /  80

          /  70

 60

          \ 40
```

最后清空树:

```
Tree after making empty:
Empty tree
```

至此测试程序设计结束，笔者输入的指令为”g++ -fsanitize=address -fno-omit-frame-pointer -g -std=c++17 test_BST.cpp -o test” 利用 AddressSanitizer 检查是否存在内存泄漏。如果不存在，则没有任何显示。

欢迎批评指正