BST remove 函数修改

1 简介

本报告介绍了二叉搜索树项目中 remove 函数的修改实现，并包含详细的测试和分析。主要目标是提升删除操作的效率，避免递归调用，并防止节点内容复制。在删除特定节点时考虑了各种情况，确保了树结构的完整性。

2 remove 函数的实现

在此项目中，remove 函数进行了如下优化：

1. 非递归实现：原始的 remove 函数通过递归方式寻找并删除目标节点。修改后的 remove 采用非递归实现，避免了递归调用带来的栈开销，提升了效率。 2. 避免节点内容复制：在删除具有两个子节点的节点时，我们采用一个 detachMin 辅助函数来分离右子树中的最小节点，以替换被删除节点。通过指针交换的方式直接替换，而不是复制内容，这样可以有效地减少内存开销。 3. 节点分离函数 detachMin：用于从给定子树中分离出最小节点。它将最小节点从树中删除并返回，用于替换被删除节点，实现了树结构的维护。

2.1 修改后的 remove 函数代码

以下是修改后的 remove 函数代码：

```
BinaryNode* detachMin(BinaryNode*& t)
{
if (t == nullptr)
{
    return nullptr;
}
BinaryNode* parent = nullptr;
BinaryNode* current = t;

while (current->left != nullptr)
{
    parent = current;
    current = current->left;
}

if (parent != nullptr)
```

```
{
    parent->left = current->right;
}
else
{
    t = current->right;
}

current->left = nullptr;
current->right = nullptr;
return current;
}

void remove( const Comparable & x, BinaryNode * & t )
{
    BinaryNode* parent = nullptr;
    BinaryNode* current = t;
    while (current != nullptr && current->element != x)
    {
        parent = current;
        if (x < current->element)
        {
            current = current->left;
        }
        else
        {
            current = current->right;
        }
    }
    if( current == nullptr )
    {
        return;    // Item not found; do nothing
    }
    if( current->left != nullptr && current->right != nullptr ) // Two children
    {
        BinaryNode* m = detachMin( current->right );
        m->left = current->left;
        m->right = current->right;
        if (parent == nullptr)
        {
            t = m;
        }
```

```
            else if (parent->left == current)
            {
                parent->left = m;
            }
            else
            {
                parent->right = m;
            }
        delete current;
        }
        else
        {
            if (parent == nullptr)
            {
                t = ( current->left != nullptr ) ? current->left : current->right;
            }
            else if (parent->left == current)
            {
                parent->left = ( current->left != nullptr ) ? current->left : curre
            }
            else
            {
                parent->right = ( current->left != nullptr ) ? current->left : curr
            }

            delete current;
        }
    }
```

3 测试结果与分析

我们设计了几个测试用例，来验证 `remove` 函数在不同情况下的表现。测试结果如下：

1. 删除叶子节点：测试删除叶子节点（如节点3）。删除后树的结构保持正确，且该节点不可再被查找。 2. 删除仅有一个子节点的节点：测试删除仅有一个子节点的节点（如节点5），并验证其唯一子节点正确连接到父节点。 3. 删除有两个子节点的节点：测试删除有两个子节点的节点（如根节点10）。删除后，右子树的最小节点（节点12）被用来替代原节点，且树的结构保持了二叉搜索树的特性。 4. 删除不存在的节点：测试删除一个不存在的节点（如100），确保删除操作安全退出且不会影响树的结构。

3.1 测试输出示例

以下是测试输出的部分示例：

```
插入元素：10，5，15，3，7，12，18
当前树的内容：
3 5 7 10 12 15 18

测试删除叶子节点
删除元素 3
删除后树的内容：
5 7 10 12 15 18
检查删除后查找元素 3：未找到

测试删除只有一个子节点的节点
删除元素 5
删除后树的内容：
7 10 12 15 18
检查删除后查找元素 5：未找到

测试删除有两个子节点的节点
删除元素 10
删除后树的内容：
7 12 15 18
检查删除后查找元素 10：未找到
```

3.2 结果分析

测试结果表明，修改后的 remove 函数在所有情况下均表现正确。以下是各测试用例的结果分析：

- 叶子节点删除：删除后，树的结构未受影响，查找该节点时返回未找到。
- 仅有一个子节点的节点删除：子节点正确接到父节点，删除操作后符合预期。
- 有两个子节点的节点删除：右子树的最小节点替代被删除节点，实现了无内容复制的结构调整。
- 不存在节点的删除：删除操作安全退出，没有影响树的结构。

4 总结

本次修改优化了 remove 函数，使其在删除节点时避免递归调用和内容复制，提升了效率。测试结果表明，remove 函数在所有情况中均表现正确，树的结构在多种操作下均保持了二叉搜索树的性质，验证了修改的有效性。