作业报告

自动化（控制）韩箫；3230100653

2024.11.1

1 相关代码呈现与分析

添加的 remove 函数以及相关补充函数的代码如下：

1.1 find_parent 函数

```
    BinaryNode *find_parent(const Comparable &x, BinaryNode *&root) const //寻亲函数，按值查找版
    {
        BinaryNode *hunter = root;//hunter用来定位父节点
        if( root == nullptr )
            return nullptr;//空子树情形
        else if( hunter == nullptr){
            std::cerr<<"No such a child"<<endl;//所要找的子树中没有该子节点，出错
            return nullptr;
        }
        else{
            if(x < hunter->element)
            {
                if(hunter->left->element == x)
                    return hunter;
                else
                    find_parent(x, hunter->left);
            }
            else{
                if(hunter->right->element == x)
                    return hunter;
                else
                    find_parent(x, hunter->right);
            }

        }
    }
```

如果不是删除根节点，那么需要找到父节点，在删除所需 remove 的节点后，可以将其子节点连接回树上。因为二叉查找树实际与链表相似，我分析之后觉得如果要规避递归，使用指针操作的话，寻亲的操作应该是不可避免的。所以添加了 find_parent 函数，集成了一下这个功能。

其中，如果对一个节点查找其父节点，预期中这个节点应该是存在于树中的，如果找不到这个节点，应该是发生了某些错误导致该节点丢失，所以在函数中加入了一些错误处理，会输出错误信息。

另外，作业要求通过修改指针和节点替换的方式，实现删除，避免递归删除，避免节点内容复制。但按照课上所讲，

遍历操作一般是由递归完成，而且也没有要求在相关的补充函数中不能使用递归操作，所以在这里使用；但是在删除操作中，是指针操作。所以应当是符合作业要求的。
最开始写的寻亲函数版本是按节点查找的，即传入所要寻亲的节点的指针，但在后续操作中，发现不如传入该节点的元素值更加方便，所以修改了函数的参数。

1.2 find 函数

```
    BinaryNode *find(const Comparable &x, BinaryNode * & t) const//查找函数，用于返回节点
    {
         if( t == nullptr )
         {
            std::cout<<"Not Found"<<endl;
            return nullptr;
         }

        else if( x < t->element )
            return find( x, t->left );
        else if( x > t->element )
            return find( x, t->right );
        else
            return t;
    }
```

查找函数与寻亲函数的逻辑是相似的，之前想了一下能否用寻亲函数替代。

- 如果所要查找的节点不是根节点，那么可以找到其父节点之后，再往下一层找到。
- 但如果是根节点，无父节点，这种操作不可行。

所以查找函数和寻亲函数一定程度上不可互相替代，都不可或缺。又因为两者功能相似，都是查找节点，所以在实现时都使用递推遍历的操作。

1.3 detachMin 函数

```
    BinaryNode *detachMin(BinaryNode *&t)//
    {       if(t == nullptr)
            return nullptr;
        else{
            BinaryNode * minNode = findMin(t);
            BinaryNode * parent = find_parent(minNode->element, root);
            parent->left = nullptr;//删除该节点，因为是最小节点，所以一定在左边
            return minNode;
        }
    }
```

detachMin 函数的作用是查找以 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。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。
这个函数的本质是将右子树最小节点与原树分离，即 detach，变为孤立节点用于后续操作，所以无需进行 delete 对内存分配进行操作。
在 remove 中，只有在将被删除的节点含有两个子节点的情况下，会调用 detachMin 找其右子树的 minNode, 而

minNode 一定为叶子。考虑子树的根本身是最小节点的情况，即右子树的最小节点恰为其根本身，如果 `BinaryNode * parent = find_parent(minNode->element, root);`中 root 换为 t，则寻亲函数失效，无法找到父节点，所以此处参数必须使用类中的 root 节点指针，从树的根部开始寻找。

1.4 remove 函数

```
    void remove( const Comparable & x, BinaryNode * & root )
    {
        if (!(contains(x, root)))//不包含该节点
            return;//没找到，不操作
        else{
            BinaryNode *toRemove = find(x, root);
            if (toRemove == root)//防止对根找父节点出错，单独考虑
            {
                BinaryNode * right_Min_child = detachMin(toRemove->right);
                right_Min_child->left = root->left;
                right_Min_child->right = root->right;
                root = right_Min_child;//删除根节点需要更新根节点信息，否则print操作无法实现
                delete toRemove;
                return;
            }
            else if((toRemove->left == nullptr)&&(toRemove->right == nullptr))//叶子
            {
                BinaryNode * parent = find_parent(x, root);
                if(x < parent->element)
                    parent->left = nullptr;
                else
                    parent->right = nullptr;
                delete toRemove;
                return;
            }
            else if(toRemove->left == nullptr)//只有右子节点
            {
                BinaryNode * parent = find_parent(x, root);
                if(x < parent->element)
                    parent->left = toRemove->right;
                else
                    parent->right = toRemove->right;
                delete toRemove;
                return;
            }
            else if(toRemove->right == nullptr)//只有左子节点
            {
                BinaryNode * parent = find_parent(x, root);
                if(x < parent->element)
                    parent->left = toRemove->left;
                else
                    parent->right = toRemove->left;
                delete toRemove;
                return;
```

```
45                  }
46              else//有两个子节点
47              {   BinaryNode * parent = find_parent(x, root);
48                  BinaryNode * right_Min_child = detachMin(toRemove->right);
49                  right_Min_child->left = toRemove->left;
50                  right_Min_child->right = toRemove->right;
51                  if(x < parent->element)
52                      parent->left = right_Min_child;
53                  else
54                      parent->right = right_Min_child;
55
56                  delete toRemove;
57                  return;
58              }
59              }
60      }
```

remove 函数就对将被删除节点的六种情况，进行分类讨论，分别为

1. 树不包含该节点
2. 该节点恰为树根
3. 该节点恰为叶子
4. 该节点只有右子节点
5. 该节点只有左子节点
6. 该节点有两个子节点

其中，第二种情况对根节点进行删除，`root = right_Min_child;`是必需的。如果没有这句语句，直接对 root 进行 delete 操作，由于这个函数中为左值引用参数，会直接修改 BinarySearchTree 类中的 root 信息。若 root 被清空，因为后续的内部 printTree 函数需要传入的参数是 root，会对 print 函数造成影响，如图 **??** 所示。

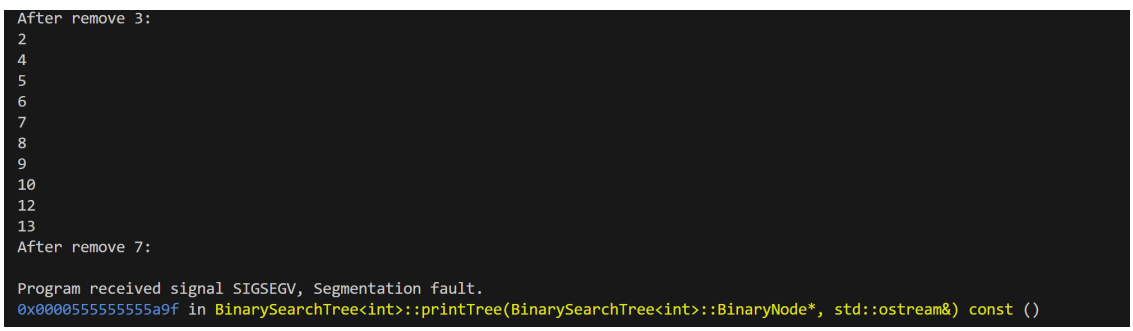

图 1: Segmentation fault

当 root 被删除后，再次调用 printTree 函数时，会出现段错误。所以需要及时对 root 进行更新。

另外，对于后四种情况，都需要知道将被删除的节点，是其父节点的左子树还是右子树，所以需要调用寻亲函数，找到父节点，利用 toRemove 的节点值与其父节点值进行比较，再依据各种情况进行相应的赋 nullptr 或者指针替换操作，如下所示。

```
        if(x < parent->element)
        parent->left = ...
    else
        parent->right = ...
```

2 测试函数

省略插入函数建立树的部分，得到的树应为

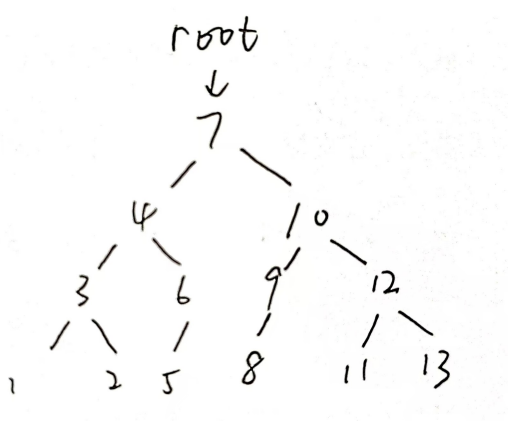

图 2: BinarySearchTree

只展示了删除部分的测试函数。完整详见 main.cpp 文件。

```
int main()
{
BinarySearchTree<int> t;

t.insert...

std::cout<<"The tree is:"<<endl;
t.printTree();

t.remove(1);//删除叶子节点
std::cout<<"After remove 1:"<<endl;
t.printTree();

t.remove(3);//只有右子节点
std::cout<<"After remove 3:"<<endl;
t.printTree();

t.remove(7);//删除根节点
std::cout<<"After remove 7:"<<endl;
t.printTree();

t.makeEmpty();
std::cout<<"After makeEmpty:"<<endl;
t.printTree();
```

```
    t.insert...

    std::cout<<"The tree is:"<<endl;
    t.printTree();
    t.remove(9);//只有左子节点
    std::cout<<"After remove 9:"<<endl;
    t.printTree();

    t.remove(10);//有两个子节点
    std::cout<<"After remove 10:"<<endl;
    t.printTree();

    t.remove(3);//有两个子节点，且右子树的最小节点恰为叶子
    std::cout<<"After remove 3:"<<endl;
    t.printTree();

    t.insert...

    std::cout<<"The tree is:"<<endl;
    t.printTree();

    t.remove(15);//移除不存在节点
    std::cout<<"After remove 15:"<<endl;
    t.printTree();
    }
```

即分别对上述六种情况逐一进行测试，查看是否会出现错误。其中对于有两个节点的情况，还要考虑右子树的最小节点恰为叶子，验证这种情况下 detachMin 函数能否正常工作。

3 测试结果

```
summer_hare@localhost:/mnt/d/浙大本科学习/大二上/数据结构与算法分析/My Project/BST$ ./test
The tree is:
1
2
3
4
5
6
7
8
9
10
11
12
13
After remove 1:
2
```

```
3
4
5
6
7
8
9
10
11
12
13
After remove 3:
2
4
5
6
7
8
9
10
11
12
13
After remove 7:
2
4
5
6
8
9
10
11
12
13
After makeEmpty:
Empty tree
The tree is:
1
2
3
4
5
6
7
```

```
8
9
10
11
12
13
After remove 9:
1
2
3
4
5
6
7
8
10
11
12
13
After remove 10:
1
2
3
4
5
6
7
8
11
12
13
After remove 3:
1
2
4
5
6
7
8
11
12
13
The tree is:
1
```

```
2
3
4
5
6
7
8
9
10
11
12
13
After remove 15:
1
2
3
4
5
6
7
8
9
10
11
12
13
```

从测试结果来看，删除操作正常，符合理论预期。

使用 valgrind 检查内存泄漏，结果如下：

```
==41042== HEAP SUMMARY:
==41042==     in use at exit: 0 bytes in 0 blocks
==41042==   total heap usage: 227 allocs, 227 frees, 21,159 bytes allocated
==41042==
==41042== All heap blocks were freed -- no leaks are possible
==41042==
==41042== For lists of detected and suppressed errors, rerun with: -s
==41042== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

无内存泄漏。