1 测试程序的设计思路和预期输出

我的测试程序分为 10 个部分。

1.1 copy

拷贝测试验证了如下内容：拷贝构造函数、赋值语句、自我赋值、= 运算符的重载、深度拷贝。我创建了 1 个原始列表 l1 和 3 个用于验证如上内容的列表 l2, l3, l4。l2 验证了 = 运算符的有效重载，l3 验证了拷贝构造函数，l4 则验证了连续赋值和自我赋值是否处理得当。在将它们全部输出完毕后，我对 l2, l3 分别进行了修改，然后再次将它们全部输出来验证深度拷贝。

1.2 push_back

我创建了一个空列表，随后调用 push_back() 方法依次插入了 1, 2, 3 三个元素，最后输出了整个列表的内容。按照预期，程序应当输出 1 2 3。

1.3 push_front

我创建了一个空列表，随后调用 push_front() 方法依次插入了 1, 2, 3 三个元素，最后输出了整个列表的内容。按照预期，程序应当输出 3 2 1。

1.4 pop_back

我创建了一个列表并初始化为 {1, 2, 3}，之后调用 pop_back 方法，最后输出了整个列表的内容。按照预期，程序应当输出 1 2。

1.5 pop_front

我创建了一个列表并初始化为 {1, 2, 3}，之后调用 pop_front 方法，最后输出了整个列表的内容。按照预期，程序应当输出 2 3。

1.6 iterator

对迭代器的测试包括前置/后置 ++ 运算和前置/后置–运算。我创建了一个列表并初始化为 {1, 2, 3, 4, 5}。随后，利用迭代器定位了尾部元素 5，并按顺序出了后置/前置-、后置/前置 ++ 后迭代器所指的元素。随后，我连续调用了三次前置–和两次前置 ++。按照预期，程序应当输出 5 3 3 5 2 4。

1.7 insert

我创建了一个列表，并初始化为 {1, 3}，之后调用迭代器定位了 3，并调用 insert 方法在它之前插入了 2，最后输出了整个列表的内容和返回的迭代器所指的元素。按照预期，程序应当输出 1 2 3 \n2。

1.8 erase_single

我创建了一个列表并初始化为 {1, 2, 3}，之后调用迭代器定位了 2，并调用 erase 方法删除了它，最后输出了整个列表的内容和返回的迭代器所指的元素。按照预期，程序应当输出 1 3 \n3。

1.9 erase_range

我创建了一个列表并初始化为 {1, 2, 3, 4, 5}，之后用迭代器和自增操作将区间起点和中点分别定位在了 2，4。随后，调用了 erase 方法删除了这个区间内的元素。最后输出了整个列表的内容和返回的迭代器所指的元素。由于区间是左闭右开的，因此程序按照预期应当输出 1 4 5 \n4。

1.10 size and empty

我创建了一个空链表并测试了 size 和 empty 方法的返回值。随后，分别调用了 push_back、push_front、pop_front、pop_back、erase(single)、erase(range) 所有可能的操作并检查每次操作后的 size 和 empty。最后，我还检查了通过 initializer_list 方法初始化的链表的 size 和 empty。

2 测试的结果

程序按照预期完成了所有的操作，所有的输出都与预期相同。

我用 valgrind 进行测试，没有发生内存泄露。

3 对于区间删除的性能优化

在下发的代码中，提示可以优化区间删除的实现。先前的实现方法是通过单点删除的方法依次删去 [head, tail) 的所有元素，然而，这其中有许多不必要的指针修改。实际上，除了区间起点和终点处及附近，其他位置的指针修改都是不必要的，毕竟它们终究会被删除。根据这个思路，我重构了区间删除的实现。具体地，直接将头的 next 指向尾部，尾的 prev 指针指向头部，中间删除时不再修改任何指针。注意左闭右开以及更新 size。