面向对象程序设计基础作业十三 设计文档

1. 模型部分
   1. 功能简述

本程序实现了一个学生学号与成绩管理系统，并使用双向链表储存数据，便于双向遍历访问数据。此管理系统支持用户添加任意数量（不超过系统内存）的学号、成绩信息（学号限制为long long范围的整数，成绩限制为unsigned int范围的非负整数），删除或访问第一个特定学号/全部特定分数的档案，或输出全部的档案信息。

本程序中包括的double\_list类是一个类模板，可以存储任意类型的数据。

本程序采用工厂模式方法进行设计，由factory类负责申请内存，student\_system类不自行申请内存，而是调用factory的接口get\_node。注意到factory类也是一个模板，可以用于各类数据的申请和临时存储。

说明：本次作业中的factory.h和double\_list.h在头文件中就实现了其函数接口，这是因为这两个类都写成了模板，不能在.cpp中实现，故没有做到定义与实现分离。

* 1. 数据结构

本程序包含student\_system类，可以实例化为成绩管理系统。此类调用double\_list类（即双向链表）作为数据存储结构。

Double\_list类提供基本的双向链表创建、添加节点、删除节点等功能，而且可扩展性好。和普通双向链表不同的是，本类中的双向链表不使用单一的头指针表示链表，而是使用“哨兵”机制：即，本类有一个成员变量sentinel，此变量为一个虚拟的节点，其后继为真实的表头，其前驱为真实的表尾。遍历时，如访问到哨兵，就认为是达到了尽头。这样做的好处是，在添加/删除节点时不需要考虑是否处于头/尾位置，大大减少了代码工作量、增加了可读性。

* 1. 算法

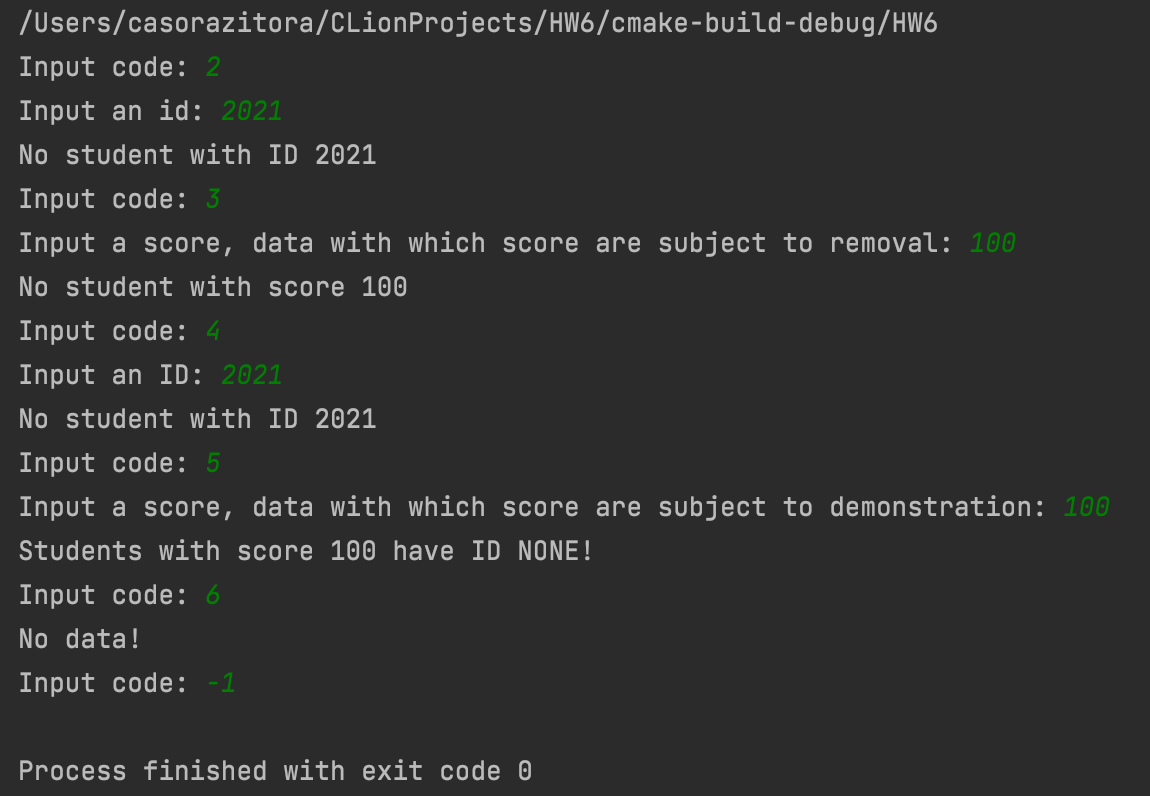
本程序主要使用模拟法。

当用户希望添加节点时，本程序会从表头（即sentinel.next）开始向后搜索，直到搜索到第一个比待加入节点学号大的节点、或学号相同但分数更高的节点、或sentinel为止，然后将新节点插入于此节点之前。插入的过程是：待插入节点的后继指针指向插入位置；前驱指针指向插入位置的前驱；待插入节点前驱的后继指针指向待插入节点；待插入节点后继的前驱指针指向待插入节点。由于引入了sentinel机制，无须判断是否处于头指针位置/链表是否为空。

当用户希望删除某特定学号/成绩的节点时，本程序会从表头开始向后搜索，直到搜索到足够数量的符合要求的节点为止。删除节点的过程是：待删除节点前驱的后继指针指向待删除节点的后继；待删除节点后继的前驱指针指向待删除节点的前驱；释放待删除节点。

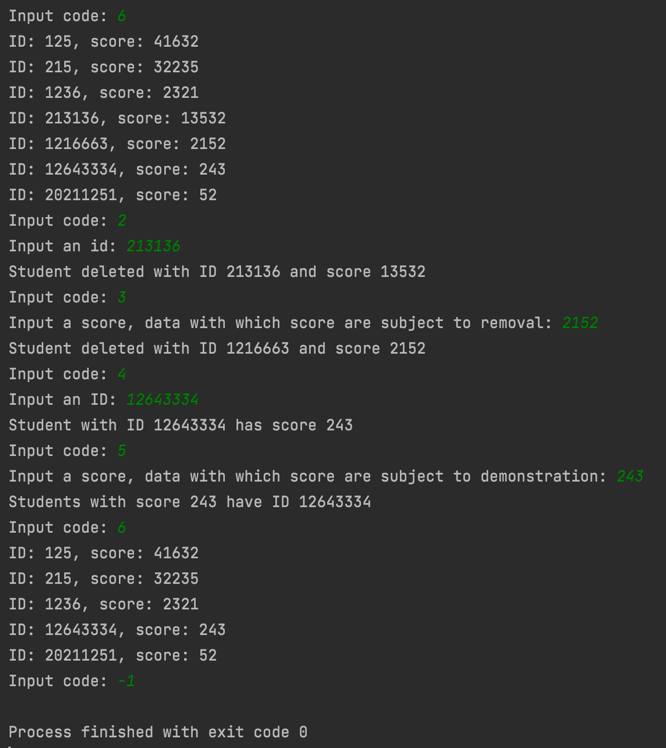
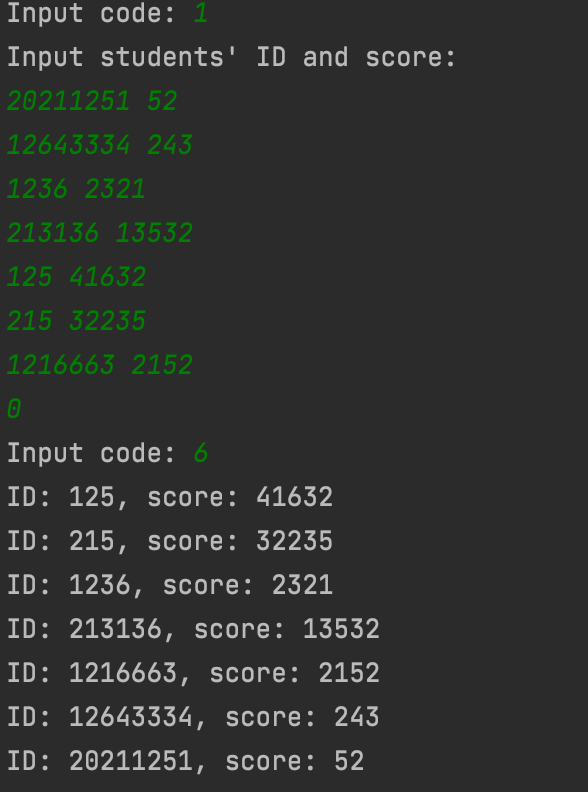
当程序结束、需要释放所有空间时，本程序使用了double\_list类的析构函数以实现自动回收内存。具体来说，此析构函数会逐一访问每个节点，记录下其后继，并释放它们的空间。由于这是一个析构函数，会在工厂类factory和产品类student\_system销毁时自动调用，删除这两个类的临时表单中的所有结点。

1. 验证部分

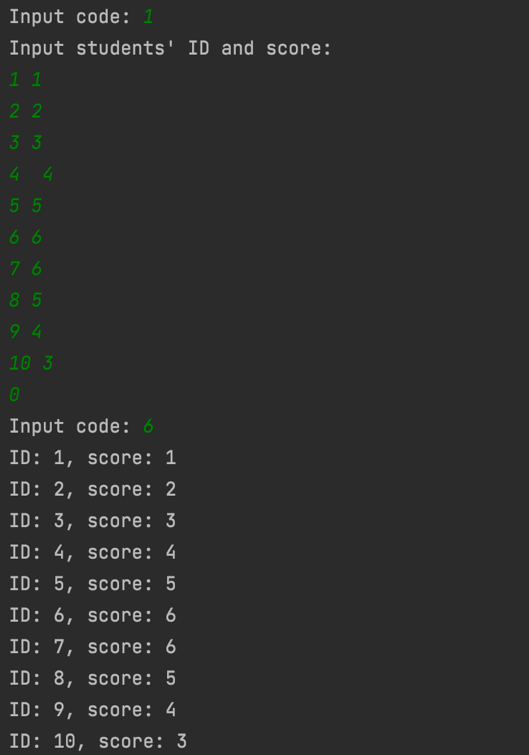
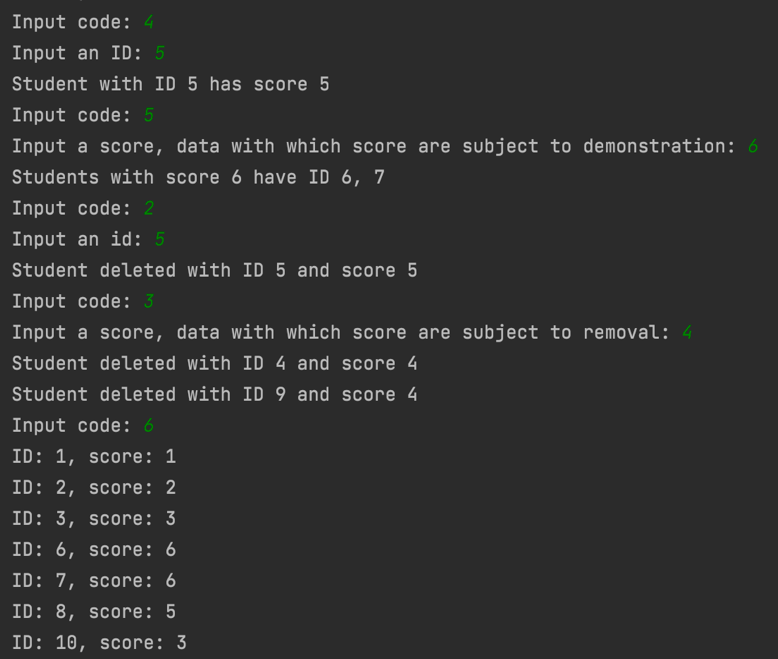
本程序的验证使用了五组不同的数据，代表不同的情形。在下述每个情形处会具体说明。

* 1. 列表是空表的情形

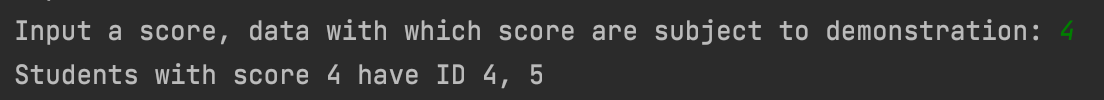
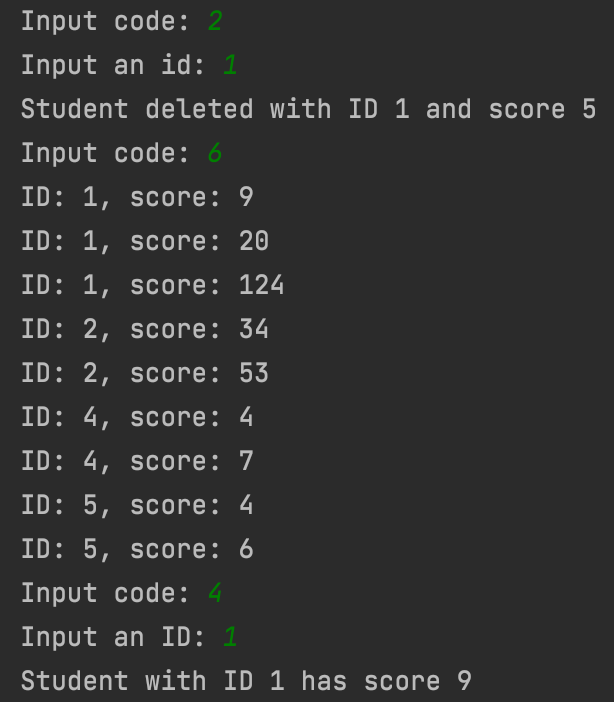
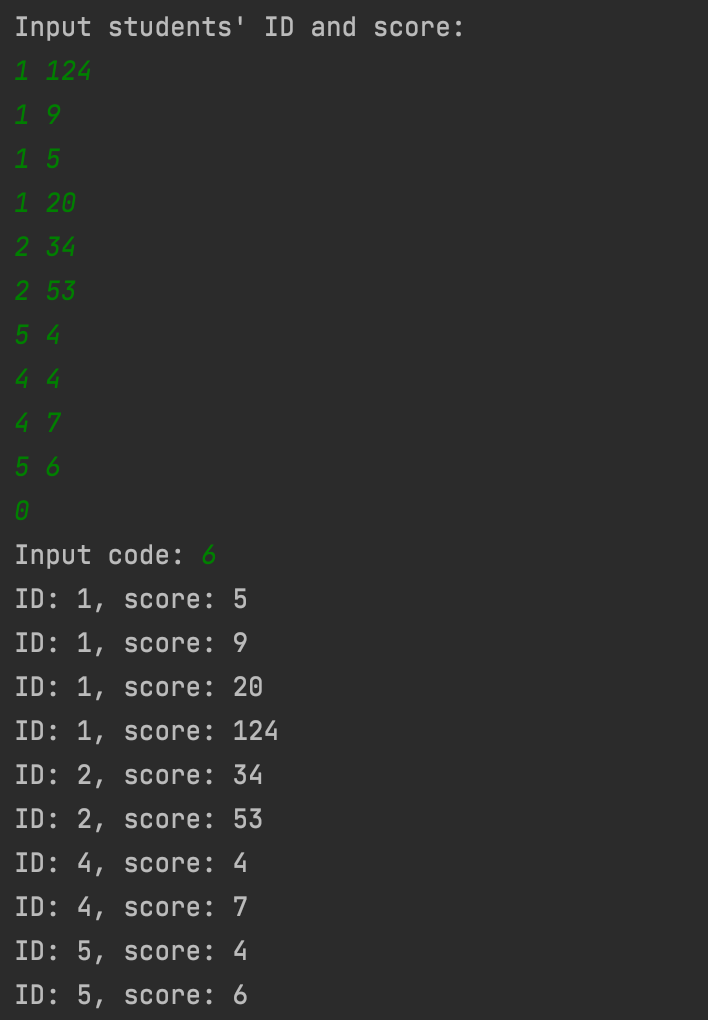
开始执行后，不为此程序读入任何数据，直接进行各项功能的测试。结果表明，每一个功能中都正确输出了“无数据”的结果，没有报错/内存异常。

* 1. 列表中没有重复分数/学号的情形

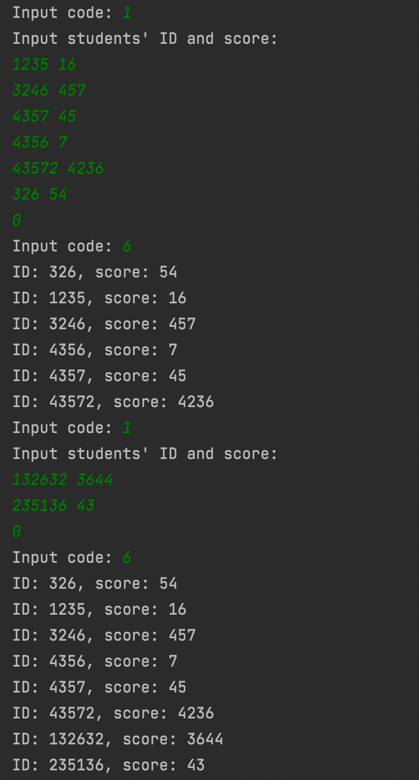
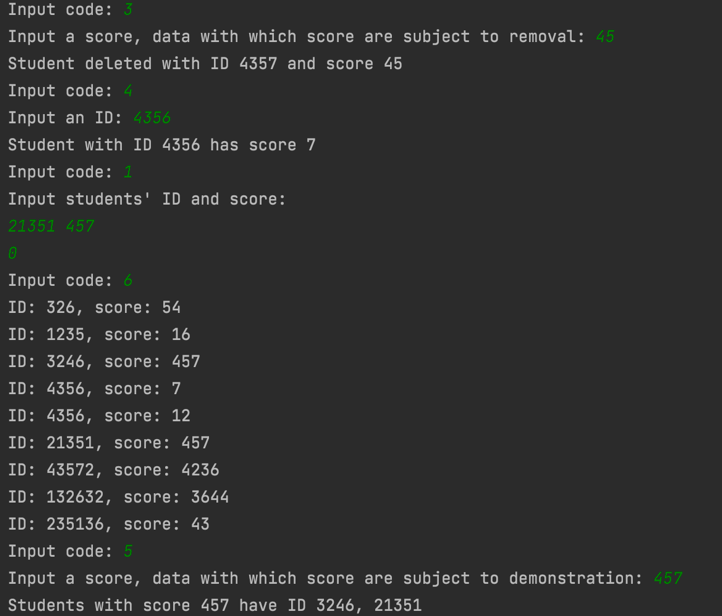
尽管功能1输入时顺序是完全打乱的，但使用功能6可以看出，在内存中的各个节点都已经按照学号升序排列。随后，使用2-5各个功能均能正确删除/查询所要求的数据。本例中，共删除了2次数据，查询了2次数据。考虑到列表中没有重复分数、也没有重复学号，输入规模为7，最终状态仅剩5个数据，这是正确的。

* 1. 列表中有重复分数的情形

使用功能1输入时，有四组数据（3-10，4-9，5-8，6-7）具有相同的分数。可以看出，程序仍然按照学号升序构造了对应的链表。在使用功能5查询时，输入分数6，程序给出了全部的两个具有分数6的学号，并使用逗号分隔。使用功能3删除时，输入分数4，程序删除了全部的两个具有分数4的数据。加上使用功能5删除的一个数据，最终正确地剩余了7个数据点。

* 1. 列表中有重复学号的情形

使用功能1添加数据时，输入了若干组分数不同、学号相同的数据。使用功能6显示，链表确实是按照分数升序排列的。使用功能2进行删除，也只会删除此序列里第一个此学号的数据。使用功能4进行查询，只会显示此序列里第一个此学号的数据。而使用功能5（查询所有某成绩的学号）可以正常输出所有有相同成绩的学号。

* 1. 列表中多次添加新数据的情形

此处测试了连续两次调用/相隔若干操作后再次调用功能1的场景。可以从功能6的结果看到，多次调用添加数据功能后，链表内的元素仍然按照上面提到的顺序排列。同时，其余几项功能也正确运行。

* 1. 性能测试

考虑到本程序（下称新程序）和本学期第六次小作业的程序（下称旧程序）有相同的功能，但采用了工厂类管理内存，我们希望测试这一变化是否有性能上的提升。

采用其他方式生成N个数据（N的取值见下表），分别对应学号1至N，分数均为1。（这是为了调用一次命令3就可以删除全部数据，更方便测试。）先使用指令1，将这n个数据全部输入，再使用指令3，全部删除。旧程序会直接释放这些内存，而新程序会交由factory类暂时保管。之后，再次用指令1将这些数据全部输入。旧程序需要重新分配内存，新程序则可以从factory类获取先前暂时保管的实例，不需要另外申请内存。

使用计时器，可以定量地分析性能差异。下表是本程序在MacOS上运行的测试结果。可以看出，使用工厂模式设计的程序在回收时有少量优势，在再次输入时有明显优势。

|  |  |  |  |
| --- | --- | --- | --- |
| N的取值 | 步骤 | 旧程序用时（ms） | 新程序用时（ms） |
| 10000 | 删除 | 11.461 | 10.963 |
| 再次输入 | 143.69 | 24.503 |
| 50000 | 删除 | 48.822 | 56.117 |
| 再次输入 | 2604.43 | 155.068 |
| 200000 | 删除 | 286.079 | 112.981 |
| 再次输入 | 13223.8 | 351.755 |