

2024 秋/2025 春计算机学院“数据结构课程设计”题目及要求

【课程设计任务】

“数据结构课程设计/数据结构与算法综合实习”（以下统称数据结构课程设计）是一门必修的实践课程，要求每个学生必须在规定时间里到实验室 机房上机完成。

“数据结构课程设计”题目共分两组，每个同学可任选一组完成。注意题目的备注，团队 合作的题目如果是 2 个人合作完成，每个人需要明确各自的具体分工，且分别提交此题报告列出合作题目两个人的具体分工，并详细介绍自己实现哪些功能、以及如何实现。

第一次上机前每位同学需完成选题，然后按工程化的基本流程分别完成如下任务：1) 阅读题目要求，熟悉问题；2) 对设计问题进行需求分析；3) 系统设计，明确程序的模块结构；设计数据结构（逻辑结构及其物理结构），设计主要子问题的求解算法；4) 程序实现；5) 程序测试；6) 程序优化；7) 设计总结，撰写课程设计报告。

成果提交：将程序源代码/工程文件、可独立运行的可执行程序、简要操作手册及“数据 结构课程设计”实习报告电子版（模板 <http://pan.baidu.com/s/1o8lHP6E>）打包，文件命名格式为“专业班级-学号-姓名”，如：19*231-学号-***。并将设计报告打印为纸质版（A4 双面打印），然后以班为单位在指定时间将电子版和纸质版集体提交到指导老师。

【课程设计总体要求】

1 坚守学术诚信

鼓励有个人特色的创新，可酌情予以加分。严禁对程序与报告的抄袭行为（包括来自网络资源及其他同学的代码和文档），后续会对代码和报告进行查重，一经发现抄袭，课程设计成绩计 0 分，以考试抄袭舞弊行为处理。

2 程序规范

程序遵从一般性规范：

- (1) 源码依据模块组织到不同.h 与.cpp 文件中，不要将全部程序放到一个源文件中。
- (2) 变量和函数名称尽量基于描述性命名，遵循见名知义的原则。
- (3) 函数头有统一注释，说明功能，输入输出与条件等。
- (4) 函数内部关键处理步骤处加上注释予以说明。

3 报告规范，内容完善

按照格式规范与内容要求撰写课程设计报告，报告主要内容应至少涵盖如下方面：（请参考数据结构课程设计模板 <http://pan.baidu.com/s/1o8lHP6E>，且仅粘贴关键代码并附上注释，不要把所有代码都贴到实习报告里，纸质报告请双面打印，一份报告包含全部题目）

- 一、问题描述
- 二、需求分析
- 三、程序总体设计(含模块结构图)
- 四、数据结构和算法详细设计
- 五、程序实现 (C++语言程序实现的简要说明，如开发环境、支持包、函数原型与功能及调用关系；全部源程序以电子版提供，报告中只能作为附录内容之一)
- 六、程序测试及结果分析
- 七、复杂度分析
- 八、总结、特色与不足

主要参考文献

4 检查与验收

课程设计最终成绩由平时成绩、演示程序并回答问题、课程设计报告三个部分综合评定。如果未经老师验收程序并回答提问、或者不交实习报告都将作为“不及格”处理。

在设计课内，全体同学需给指导老师演示程序，讲解程序，回答老师提问，验收或报告完成情况。结课后提交课程设计报告。

【课程设计题目】

本次课程设计题目分为两组，每人任选一组完成。

未经许可不要轻易更换组号和题目。

第一组：飞机票管理系统，Trie 树和后缀树的应用，交通咨询系统设计，简单搜索引擎系统

第二组：高效带权无向简单图与交通路线规划系统（团队完成）

题目 1：飞机票管理系统（个人完成，30 分）

【问题描述】

综合运用线性表、队列、图等数据结构来设计飞机票管理系统，要求实现航班信息管理、航班动态管理、票务管理和票务查询等功能。实现以下五个核心模块：

1. 航班信息管理：这部分涉及用数据结构存储和操作航班信息，如航班号、航空公司、起飞降落时间、经停地点和可售票数等。使用线性表来存储航班信息，支持增删改查操作。

2. 航班动态管理：当航班状态变化时，如延误或取消，需要及时更新并通知乘客。当两城市间无直飞航班时，能够推荐最合适的转机方案，考虑价格、时间等因素。

3. 票务管理：

- 客户购票：支持实时购票与预约抢票服务。

- 退票功能：允许乘客取消预订，释放座位资源。

预约抢票功能需要记录预约顺序，可以利用优先队列实现公平的分配机制。

4. 票务查询：查询功能需要快速响应用户的输入，如查询特定航班或城市间的航班信息。

这涉及到高效搜索结构的使用。

5. 查询结果排序：查询结果需要按照特定标准（如时间、价格等）进行排序，需要掌握排序算法，如快速排序等。

【基本要求】

1、实现以上五个核心模块的要求。

2、建议参考实际航空公司的信息构建测试数据，至少包含 20 个城市、300 条航线的信息，以覆盖各种可能的查询场景，可使用文件输入。

【扩展要求】

1、当航班状态变化如延误或取消时，需要及时更新并向乘客推荐替代航班。设计一个有效的方法来追踪航班状态，并用合适的数据结构（如图）表示航班间的关联，以便快速找到替代航班。

2、购票和退票功能处理并发请求。

3、设计可视化图形界面。

题目 2：Top-K问题求解（模拟热搜统计）（个人完成，20 分）

【问题描述】

1、基于给定的 QT/MFC 聊天室模板 (QT 模版链接：<https://pan.baidu.com/s/1fGnX-IWuwdnM8t9glyR8Xg?pwd=5sfcc>；MFC 模版链接：<https://pan.baidu.com/s/1XsmK4NMh2Gd3fNaox3Dhw?pwd=qjah>)，实现客户端定时向服务器端传输数据的功能(初始时先传输1万个随机数[范围0-1024]作为历史数据，此后默认每隔30s向服务器传输1万个随机数，这意味着服务器端接受到的数据是动态变化的，随机数生成请使用模板里提供的CreateRandNums类)；

2、步骤1中请对比不压缩数据直接传输(直接转为二进制模拟网络数据传输)和使用Huffman树进行数据压缩后再进行数据传输的开销(网络传输的byte总量)。提示：传输Huffman编码后的二进制数据时需先传输编解码字典，格式自定，否则无法译码；

3、在服务器端基于接受到的动态数据，建立最大堆寻找出现次数最多的20个整数并动态显示(30s-60s更新一次，请根据你的算法效率来决定最小更新间隔)；

4、请尝试对比其他能够从动态数据中找到出现次数最多的20个整数的算法，例如最朴素的循环遍历，并与基于堆的算法进行对比(对比次数和时间)，动态显示对比结果。需求4中如能够对比多个不同的算法可加分。

【基本要求】

1. 深入理解堆，并能够灵活运用。
2. 针对Top-K问题寻找并实现其他数据结构和算法，并与基于堆的算法对比分析算法的时间复杂度。

题目 3：交通咨询系统设计（个人完成，40 分）

【设计目的】

熟练掌握迪杰斯特拉算法和费洛伊德算法，能够利用它们解决最短路径问题。

掌握图的深度，广度遍历算法。

掌握快速排序算法。

【基本要求】

设计一个交通咨询系统，通过读取全国城市距离图 (<http://pan.baidu.com/s/1jIauHSE>，请在程序运行时动态加载到内存，可将 excel 转成 csv 方便读取)， 实现：

- 1、请验证全国其他省会城市（不包括港澳和两个宝岛台北和海口）到武汉中间不超过 2 个省（省会城市）是否成立？（正是因为武汉处于全国的中心位置，此次疫情才传播的如此广）；
- 2、允许用户查询从任一个城市到另一个城市之间的最短路径（两种算法均要实现，界面上可自行选择）以及所有不重复的可行路径（可限制最多经过 10 个节点），并利用快速排序对所有路径方案依据总长度进行排序输出（输出到文件），每一条结果均需包含路径信息及总长度，试比较排序后的结果与迪杰斯特拉算法和费洛伊德算法输出的结果；
- 3、假设在求解 2 个城市间最短路径时需要绕过某个特定的城市（用户输入或者选择，例如武汉），请问应该如何实现？
- 4、不基于功能 2 遍历的结果如何直接求解两个城市间的前第 K 短的路径，例如，武汉到北京之间第 3 短的路径。

题目 4：简单搜索引擎系统（团队完成，10 分）（自行组队，每队不超过 2 人，需有明确分工，实习报告中需明确注明每人的分工）

【设计目的】

掌握倒排索引相关算法。

掌握字符串的相关处理方法。

掌握文件的操作。

学习使用开源代码库。

学习团队分工合作以及 svn/git 代码版本控制软件。

【基本要求】

给定 n 个文本文件（检查时我会提供测试数据，编程时请自行收集，需实现导入某一个文件夹下所有文本文件的功能。同时也鼓励大家直接从网页上抓取数据(例如从微博网抓取网页到本地，该功能即为网页爬虫，可利用任何开源代码，建议使用 Python，处理网络数据很方便，请自行网上查找，不要求看懂内部实现，会用即可，如实现该功能有加分，参考 <https://github.com/dataabc/weiboSpider>，<https://github.com/liinnux/awesome-crawler-cn> 等），具体需求如下：

- 1、依次读取 n 个文本文件（或者自行爬取的网页数据）并利用分词函数建立倒排索引（详查看字符串章节 ppt，利用开源代码 <https://github.com/yanyiwu/cppjieba> 或者 <http://www.oschina.net/p/freeictclas>，亦可自行网上查找其他解决方案，不需要自行实现和看懂内部实现，只需要会用）；
- 2、自行设计查找算法，支持从倒排索引中检索给定的字符串，请提供用户输入查找字符串的页面，相当于百度首页的输入框（不要直接使用 Hashtable，对于海量数据，直接使用哈希表是不够的）；
- 3、自行设计排序算法，即需要考虑当查询字符串出现在多个文档中时，哪个文档在输出后的结果中排第一 or 第二 or...，可参考 pagerank 算法（基于超链接的网页权重计算方法，可查找相关开源代码并调用）以及常见文本相似度计算方法（基于文本相似度的文档权重计算方法，<https://zhuanlan.zhihu.com/p/88938220>），也可自行设计任何可用、有一定意义的算法，简单/复杂均可；
- 4、自行设计搜索结果展示界面，要求能将关键字出现的位置附近的上下文显示出来，请参考百度的搜索结果展示页面，注意，不能在检索后对所有检索结果用字符串模式匹配函数进行关键字查找（即不要进行事后匹配，因为返回的结果是海量的）！
- 5、思考一下，如果数据量继续增大应该如何进行算法优化？

测试数据：<http://www.nlpir.org/wordpress/download/tc-corpus-answer.rar>

题目 5：高效带权无向简单图与交通路线规划系统（团队完成，自行组队，每队不超过 2 人，需有明确分工，实习报告中需明确注明每人的分工）

介绍

图是用边成对连接的一组顶点。图是广泛、有用、强大的数学抽象，可以描述从生物学 和高能物理学到社会科学和经济学等领域的复杂关系和相互作用系统。目前已知数百种图算 法，数以千计的实际应用，是计算机科学和离散数学引人入胜的一个分支。元素之间的成对 连接在大量计算应用程序中起着至关重要的作用。例如在人工智能和大数据领域，知识图谱 使用图数据库，深度学习使用图神经网络。

为了使用图解决实际应用问题，我们需要：

1. 用于表示图的底层数据结构。底层数据结构的选择会对运行时间、内存使用情况、

实现各种图算法的难度等方面产生深远的影响。

2. 图的应用程序编程接口（Application Programming Interface, API）。这些 API 是我们提供给图客户端（Client）程序开发人员可用的方法列表，包括方法签名（每个函数接受的参数）及其行为。客户（希望使用我们的图数据结构的人）可以使用我们提供的任何函数来实现他们自己的算法而无需访问源码或理解内部工作机制的细节。API 定义客户端程序开发人员必须如何思考。我们提供的方法会对客户实施特定算法的难易程度产生重大影响。

我们的图 API 约定，将图的每个顶点与一个整数一一对应。可以通过维护一个映射来告诉我们为每个原始顶点关键字分配的整数。如图 1 所示。这样做允许我们定义自己的图 API 来专门处理整数结点，而不是引入泛型类型，从而简化底层内部数据结构，同时支持各种外部应用。

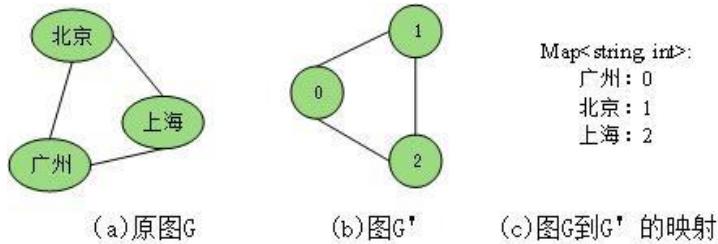


图1 图G 的整数顶点图G'以及顶点关键字整数映射

本课题在软件工程的基本指导原则下探索和练习一种高效的带权无向简单图（Weighted Undirected Simple, WUSGraph）数据结构及其 API 以及开发它的客户端应用程序系统，分为三部分。第一部分是实现一种高效的带权无向简单图的底层数据结构 WUSGraph 以及 WUSGraph API，第二部分是测试和应用 WUSGraph API 开发客户端程序 WUSGraphClient 以及 WUSGraphClient API。第三部分是测试和应用 WUSGraphClient API 开发图的应用系统 WUSGraph Application。

在这个课题，你将深入体会双向链表的优势、散列表技术的高效、C++ STL `unordered_map` 的实现原理、“伙伴”指针的优势、父指针表示法的强大描述能力、映射与无向图的应用；体会到底层数据结构的选择如何影响系统渐进运行时间和代码复杂性；体会到软件工程技术对系统软件开发的支持；体会到抽象与分离原则带来的好处；体会到面向对象编程中封装、继承、多态、重载、抽象以及访问权限等技术提高编程效率的优势。

带权无向简单图项目的初始类图见图 2。在后面将自底向上具体论述每个类。

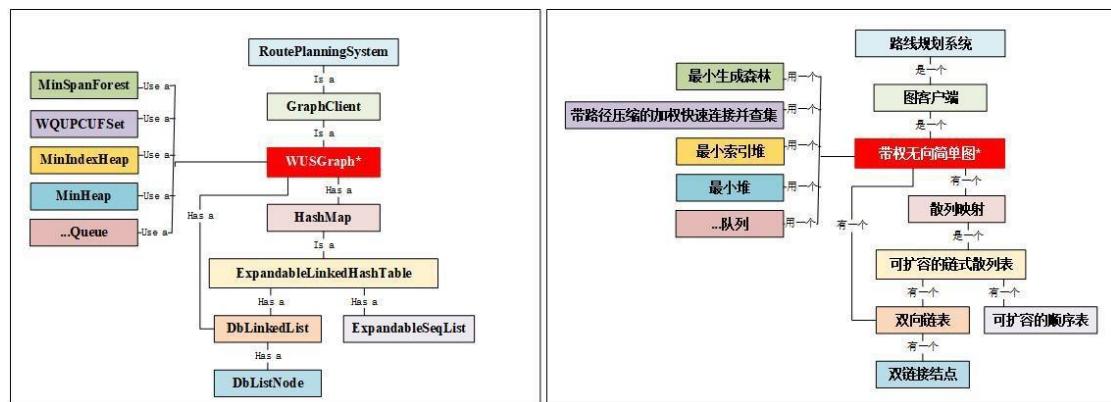


图2 WUSGraph 项目的初始类图参考

第一部分 高效的带权无向简单图应用程序接口

模块说明

K 表示数据元素关键字项的类模板，用于唯一标识数据元素。

V 表示数据元素除关键字项外，其余所有数据项的类模板，关键字项之外的数据整体被定义为 V 类型。

E 表示数据元素的类模板，定义为 std::pair<K, V>，其中 pair 是 STL 提供的标准模板类，用于将关键字和其对应的数据组合成一个整体。

类的说明

应确保类具有良好的封装性、灵活性和高效性，同时对外接口友好，符合面向对象编程 的设计原则。

1、成员变量：每个类的成员变量应根据功能需求自行设计。成员变量应使用 private 或 protected 可见性，确保数据封装性，防止外部直接访问和修改。

2、方法设计说明：为类的用户提供必要的公共接口，确保对外部用户暴露的接口简洁且功能明确。使用 private 和 protected 可见性隐藏敏感数据或仅供内部使用的方法，避免外部用户干扰类的内部实现。通过合理的接口设计，确保类的创建者与使用者之间良好的隔离，提升模块化和代码的可维护性。

3、封装要求：图类的内部数据结构不得直接暴露给外部应用。所有对图数据的访问和修改必须通过类提供的接口进行，避免直接操作内部数据结构。

4、构造函数与析构函数：根据每个类的功能需求自行设计合适的构造函数与析构函数。为了节省篇幅，文内不详细讨论构造函数与析构函数的设计，但需确保其满足类的初始化与清理需求。

5、辅助方法：根据实际需要设计有效的辅助方法，帮助实现类的功能模块化。辅助方法应保持灵活性和高效性，并根据使用场景设置合适的访问权限。

6、迭代器支持：如果需要支持遍历功能，应设计适配类的迭代器，并提供迭代器方法，例如 iterator Begin() 和 iterator End()。迭代器设计应符合类的功能需求，并能够高效地访问和操作数据。

7、容器限制：本项目不使用 C++ STL 提供的容器类。

合作说明

1、任务划分与分工：建议采用自底向上的方法，将每层数据结构和 API 开发任务进行合理划分，确保工作量和开发难度在各层之间尽可能均衡分布。在动手开发之前，合作双方需提交一份分工文档，明确详细的任务分配情况，包括每位成员负责的模块开发与测试任务。

2、界面设计约定：虽然界面设计属于系统功能的一部分，但它并非本课题的主要内容。为避免偏离核心目标，界面设计的工作量按照以下规则计入整体任务：文本界面设计的工作量视为相当于 2 个功能模块的工作量；图形界面设计的工作量视为相当于 4 个功能模块的工作量。

3、合作要求：工作任务分配合理。责任明确。项目管理与进度控制得当。团队成员能有效沟通，共同解决问题。

课题评分说明

功能的正确性、数据结构与算法的效率、面向对象设计质量以及内存管理（避免出现内存泄漏）是本项目评分的重要依据。

任务 1.1. 可扩容的数组列表 **ExpandableArrayList**

可扩容的数组列表用于动态存储元素。在需要时自动调整底层数组的大小。容量扩展对 用户透明，无需手动管理底层存储。支持常数时间 O(1)的随机访问。同时为了保持高效的访问性能，仅在数组的尾部插入和删除元素。课题中所有需要使用动态数组的场景，均要求统一采用 **ExpandableArrayList** 类型。

表1 可扩容数组列表 API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	<code>ExpandableArrayList(int initialSize)</code>	创建一个具有初始容量 initialSize 的空数组。如果没有给出 initialSize，设置为默认值 defaltSize= 8。	O(1)	
2	<code>void Push_back(const E&e)</code>	在数组的尾部插入元素 e。当列表的容量不足时，分配一个更大的数组，并将现有元素复制到新数组中。调整容量时，请确保倍数增加容量，而不是常数（例如乘以 2，而不是加 100 或其他）。你不需要考虑缩小容量。	平均 O(1)，最坏 O(n)	
3	<code>bool Pop_back(E&e)</code>	如果表空，返回 false。否则，删除表尾元素并释放所占空间，由参数 e 返回被删元素，返回 true；否则。	O(1)	
4	<code>E& operator[](int i)</code>	如果 i 合法返回第 i 个元素的引用，否则抛出 <code>out_of_range</code> 异常。例如： <code>ExpandableArrayList<int> A(10);</code> <code>for (int i=0; i<10; i++) A[i]=i;</code> <code>for (int i=0; i<10; i++)</code> <code> std::cout<<' '<<A[i];</code>	O(1)	
5	<code>bool IsEmpty()const</code>	判断是否为空数组：是则返回 true，否则返回 false。	O(1)	
6	<code>int getCapacity()const</code>	返回数组的容量。	O(1)	
7	<code>int getSize()const</code>	返回数组内实际元素的个数	O(1)	
8	<code>void reSize(int newCapacity)</code>	擴大数组容量为新的容量 newCapacity，并复制元素。	O(V)	
9	<code>void Clear()</code>	移除所有元素并销毁当前数组，同时重新分配一个容量为 defaultSize 的数组，即重置容器为初始状态。	O(1)	
10	<code>~ExpandableArrayList()</code>	移除所有元素，并释放所有存储空间。	O(1)	

任务 1.2. 双链结点 **DbListNode**

略。

任务 1.3. 实现双向循环链表 **DbLinkedList**

根据课题需求，需自行设计表 2 中的参数和返回值，确保与实际功能需求相匹配。方法设计应合理，尽量避免冗余，返回值应清晰指示操作结果或提供所需数据。

方法的性能要求：1、所有涉及链表扫描的操作（例如，查找、插入、删除操作）必须 在一次完整的扫描过程中完成（避免多次遍历同一部分链表），即操作的时间复杂度应为 $O(|V|)$ ，其中 $|V|$ 代表链表中当前的元素数目；2、空间要求：不得使用额外的辅助数组或类似的外部存储结构（如临时数组、栈等），操作应基于链表本身的指针结构完成。

表 2 双向链表 API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	DbLinkedList()	初始化双向链表	$O(1)$	
2	bool isEmpty()const	判断是否为空表：是则返回 true，否则返回 false。	$O(1)$	
3	DbListNode * Find(const E &e)const	查找第一个匹配指定值 e 的节点。如果存在，返回节点地址；否则返回 NULL。	$O(V)$	
4	void Insert(const E&e)	添加一个新的元素 e 到链表的表尾。	$O(V)$	
5	int getSize()const	返回当前元素数量	$O(1)$	
6	DbListNode*Remo ve(const E&e)	删除指定值 e 的元素并释放所占的空间，返回被删结点的地址。如果值不存在，返回 NULL。	$O(V)$	
7	void Clear()	清空所有元素并释放所占空间，恢复初始状态。	$O(V)$	
8	~DbLinkedList()	移除所有元素，并释放所有所占空间。	$O(V)$	

任务 1.4. 实现可调整容量的链式散列表：

ResizableChainedHashTable

设计一个可调整容量的链式散列表，采用链地址法作为冲突解决策略。桶数组采用任务 1.1 提供的可扩容数组列表（ExpandableArrayList）实现，每个链表（桶）使用任务 1.3 提供的双向循环链表（DbLinkedList）实现。散列表的表长应为素数，以减少冲突的可能性（在实现 reHash() 时，可以通过提前生成素数表来优化性能，从而快速找到接近目标值的素数）。

设计一个简易且随机性较高的哈希函数，保证键值的均匀分布。哈希函数先根据输入的 key，计算哈希码，结合当前哈希表的容量，将哈希码映射为哈希地址值，即桶号（链表索引），哈希地址值必须落在 [0..当前表长-1] 范围内。哈希函数应尽量将不同的键均匀映射到各个桶中，减少冲突；在处理相似键时，具备较高的随机性，减小冲突的可能性。哈希码的计算可以使用经典哈希方法（如乘法散列或模运算），也可以结合 C++ 标准库中的 std::hash<K>，利用其对 C++ 基本类型和标准库定义类型的特例化实现。对于自定义类型 K，需要程序员提供 std::hash<K> 的特例化实现，以便计算 K 的哈希码。因此，散列表默认使用 std::hash<K> 作为哈希码函数，但可以扩展自定义的哈希码方法。为增加灵活性，设计支持用户自定义的哈希码函数：通过在类模板中增加一个哈希码类模板参数，使用户可以传入自己的哈希码函数。支持用户自定义的哈希码函数，通过模板参数 Hasher 实现。散列表的模板参数设计如下：

K: 键的类型。

V: 值的类型。

Hasher: 哈希码函数类型，默认为 std::hash<K> 类型。

请记住，C++ 遵循一个重要约定：如果两个对象通过 operator== 或类似的比较函数（如 equals(x, y)）返回 true，那么它们的哈希值也必须相等。如果违反这一约定，散列表将无法正常工作，可能导致元素无法正确查找或插入。在本项目中，调用程序提供的两个对象如果被判定为相等（即 operator== 或 equals() 返回 true），那么它们应被视为表示相同的元素，其哈希值也必须一致。

为了保证散列表支持 O(1) 的查找效率，哈希表必须能够进行 **动态扩容**：当装载因子高于某个预设常数时，哈希表需要扩大，否则哈希冲突会增多，多个键值可能被映射到同一个桶，导致查找效率将从 O(1) 退化为 O(|V|)。扩容应按照 **倍数增长** 的策略进行，例如将当前容量扩大至两倍，否则扩容频繁且代价较高，会影响性能。

此外，为了确保散列表的迭代操作能在 O(|V|) 时间内完成，哈希表必须能够进行 **动态缩容**：当装载因子低于某个预设常数且当前容量大于初始容量时，哈希表需要缩小。否则，如果我们向图中添加了大量顶点（导致哈希表扩容），然后删除了大部分顶点，散列表的迭代操作将会变得非常慢。缩容同样应按照 **倍数减少** 的策略进行，例如将当前容量缩小至一半。

因此，散列表应采用 **双向倍数调整大小** 的机制，即根据装载因子的变化，动态地执行倍数扩容和倍数缩容，使得 minLoadFactor ≤ 当前装载因子 ≤ maxLoadFactor，以保持高效的查找、迭代性能和空间利用率。

表3 可扩容的链式冲突解决策略散列表API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	ResizableChainedHashTable()	初始化空散列表。创建一个初始大小为 initialSize 的散列表。如果没有指定 initialSize 和 maxLoadFactor，则使用默认值 表长 defaultSize = 13、最大装载因子 maxLoadFactor= 0.75 和最小装载因子 minLoadFactor= 0.25。	O(1)	
2	ResizableChainedHashTable(int initialSize)			
3	ResizableChainedHashTable(int initialSize, double maxLoadFactor, double minLoadFactor)			
4	int Hash(const E& e) const	哈希函数。根据输入的值 e 和哈希码函数，计算并返回哈希值 H(key)，确保范围为[0..	O(1)	

		表长-1]。		
5	DbLinkedList* findBucket (const E& e) const	根据输入的 e 调用 Hash()计算哈希值（桶号），并返回存储 e 的链表指针（即对应的桶）。	O(1)	
6	DbListNode* find (const E&e) const	查找 e 是否存在于散列表中。若存在，返回其对应的结点地址；否则，返回 nullptr。返回值类型自行设计。	O(1)	
7	int count (const E& e) const;	如果值等于 e 的元素找到返回 1，否则返回 0。	O(1)	
8	void Insert (const E&e)	插入元素 e：如果 e 已存在，则忽略；若不存在，则插入到对应的桶中。如果当前装载因子超过 maxLoadFactor，则自动扩充桶数量并重新散列元素到新的桶中。	O(1)	
9	bool Remove (const E&e)	如果散列表中已存在 e，则将元素 e 从散列表中删除，返回 true，否则返回 false。如果当前装载因子低于 minLoadFactor，则自动缩减桶数量并重新散列元素到新的桶中。	O(1)	
10	void reHash (int newcapacity)	自动调整桶的数量，确保满足条件： minLoadFactor ≤ 当前装载因子 ≤ maxLoadFactor，并重新计算所有现有元素的哈希值，将它们分配到新的桶中。新容量应为当前容量的两倍（扩容）或一半（缩容）并选择最近的素数。	O(V)	
11	bool IsEmpty ()const	判断是否为空表：是则返回 true，否则返回 false。	O(1)	
12	int getCapacity ()const	返回哈希表中当前的桶数量。	O(1)	
13	Int getLoadFactor ()const	返回哈希表的当前的装载因子。	O(1)	
14	int getMaxLoadFactor ()const	返回哈希表的最大装载因子。	O(1)	
15	int getMinLoadFactor ()const	返回哈希表的最小装载因子。	O(1)	
16	getHashFunction ()const	返回散列函数。请自行设计返回值类型。	O(1)	
17	int getBucketSize (int i)const	返回第 i 个桶中存储的元素数量。	O(1)	
18	void Clear ()	清空散列表并释放所有存储空间，但保留初始大小 defaultSize 的散列表结构。	O(V)	
19	~ResizableChainedHashTable ()	销毁散列表，并释放分配的所有存储空间。	O(V)	

任务 1.5. 散列映射 **HashMap** 及其两个内置迭代器

HashMap::iterator 和 **HashMap::const_iterator**

键是关键字的简称。在键值对(key, value)中，键是用于唯一标识对应值 (value) 的标志。值 (Value) 是与键一一对应的数据，表示键关联的具体内容或信息。在这种配对关系

中，键的类型通常记为 K，值的类型记为 V。

映射 (Map) 是由 (key, value) 键值对组成的集合。它的每个数据元素都是一个键值对 (key, value)，其中键用于唯一标识和索引值。映射的核心特点是：

1. **唯一性**: 在映射中，每个键 key 只能出现一次，即映射中的键是唯一的。
2. **键索引访问**: 映射中所有操作均通过键来查找或操作元素，值的访问依赖于键的索引。

任务 1.5.1 散列映射 HashMap

使用任务 1.4 提供的可调整容量的链式散列表 (ResizableChainedHashTable) 实现散列映射 HashMap。

因为散列映射中元素的键值必须保证唯一性和哈希一致性，禁止直接修改映射中元素的关键字段。为了确保这一点，元素类型应被定义为 `std::pair<const Key, Value>`，这样用户无法直接修改元素的关键字段，只能通过删除并重新插入的方式来更新映射中元素的关键字段。映射的值 (value) 可以修改，但用户可能需要只读访问键值对，因此需要两种迭代器。

可参考：

1、LeetCode 706：不使用任何内置的哈希表设计一个哈希映射。

2、C++ STL 容器 `unordered_map`: http://www.cplusplus.com/reference/unordered_map/

表 4 散列映射 API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	<code>HashMap()</code>	初始化映射。最初有 initialSize 个桶。	$O(1)$	
2	<code>HashMap(int initialSize)</code>			
3	<code>HashMap(int initialSize, double MaxLoadFactor, MinLoadFactor)</code>			
4	<code>V& getValue(const K& key)</code>	返回 Map 中指定键 key 映射的值的引用（可以修改），如果 Map 中不存在 key，抛出 <code>out_of_range</code> 异常。	$O(1)$	
5	<code>const V& getValue(const K& key) const</code>	返回 Map 中指定键 key 映射的值的引用（不可修改），如果 Map 中不存在 key，抛出 <code>out_of_range</code> 异常。	$O(1)$	
6	<code>E& operator[](const K &key)</code>	如果 key 不存在，则插入一个新的键值对 <code><key, V()></code> 。否则返回 Map 中指定键 key 映射的值的引用（即可修改）。 例如： <code>ResizableChainedHashTable<string, int> A;</code> <code>A["张三"] = 20;</code>	$O(1)$	

		std::cout<<'<<A["张三"];		
7	HashMap::iterator Find(const K&k)	搜索容器中键为 k 的元素，如果找到，则返回指向该元素的普通迭代器；否则返回指向 HashMap::end（即容器末尾之后位置）的普通迭代器。允许修改元素的值。	O(1)	
8	HashMap::const_iterator Find(const K&k) const	搜索容器中键为 k 的元素，如果找到，则返回指向该元素的只读迭代器；否则返回指向 HashMap::end（即容器末尾之后位置）的只读迭代器。不允许修改元素的 value 字段。	O(1)	
9	void Insert(const K&k, const V&v);	插入或更新键值对：如果 Map 中已存在 key，则更新相应的值为 value；如果 key 不存在，则插入一个新的键值对到对应的桶中。如果字典存储的键值对数量超过当前容量 * maxLoadFactor 时，调用 reHash()扩充桶数量并重新散列键值对到新的桶中。	O(1)	
10	bool Remove(const K& key, V &returnValue)	删除与指定 key 关联的键值对：如果 key 存在，删除与 key 关联的键值对，并通过参数 returnValue 返回其对应的值，返回 true。如果 key 不存在，返回 false。你需要考虑缩容。	O(1)	
11	bool Remove(const K& key, const V& val);	如果 key 存在，且键 key 当前映射到指定值 val 时，则删除与 key 关联的键值对并返回 true。如果 key 不存在，返回 false。你需要考虑缩容。	O(1)	
12	HashMap::iterator begin()const	返回指向第一个非空桶的第一个元素的普通迭代器	O(1)	
13	HashMap::iterator end()const	返回指向最后一个非空桶的最后一个元素后的普通迭代器（即最后一个非空桶的"尾后"位置）。	O(V)	
14	HashMap::const_iterator cbegin()const	返回指向第一个非空桶的第一个元素的只读迭代器	O(1)	
15	HashMap::const_iterator cend()const	返回一个指向最后一个非空桶的最后一个元素后的只读迭代器（即最后一个非空桶的"尾后"位置）	O(1)	
16	-HashMap()	销毁所有元素，并释放分配的所有内存	O(V)	

		有存储空间。		
--	--	--------	--	--

此外，HashMap 类继承自 ResizedChainedHashTable，并拥有以下方法：

int Hash(const K& key) const 计算指定键的哈希值。
 DbLinkedList* findBucket(const K& key) const 查找与指定键对应的桶。
 void reHash(int) 根据新的容量重新分配桶并调整哈希表。
 bool IsEmpty() const 检查哈希表是否为空。
 int getCapacity() const 返回哈希表的当前容量（桶的数量）。
 int getLoadFactor() const 返回哈希表的当前装载因子。
 int getMaxLoadFactor() const 返回哈希表允许的最大装载因子。
 int getMinLoadFactor() const 返回哈希表允许的最小装载因子。
 getHashFunction() const 返回当前使用的哈希函数。
 int getBucketSize(int i) const 返回指定桶的大小。
 void clear() 清空哈希表。

注意事项：

ResizedChainedHashTable 类的大多数方法的输入和输出参数是针对键值对 (key-value) 元素进行处理，而继承的子类 HashMap 的方法则支持按键 (key) 对元素进行操作。这导致类型匹配的问题。为解决此问题，可以采用以下两种设计方案：

方法一：仿照 STL 使用类模板的嵌套技术

通过类模板的嵌套技术，让 HashMap 和 ResizedChainedHashTable 共享相同的模板参数，能够灵活地支持键和键值对的类型转换。这种方法类型安全，利用模板机制自动进行类型匹配。在子类中无须重写父类方法，只需适配接口调用即可。但是，需要理解模板嵌套的复杂性，增加类的设计和阅读成本。

方法二：提供从节点值中提取键的方法作为参数

在 ResizedChainedHashTable 的方法中增加一个参数，允许调用者提供提取键的函数。在子类 HashMap 中调用时，可以通过 lambda 表达式将元素处理为键。这种方法灵活性高，可以根据需要动态调整键的提取逻辑。可以适配更多的使用场景。但是需要额外的函数定义和传递，可能增加运行时的函数调用开销。

建议根据项目的复杂程度和扩展需求选择适合的方法。

任务 1.5.2 HashMap::iterator 类

HashMap::iterator 是用于遍历散列映射中的元素的迭代器类，提供常见的操作如移动、解引用和相等性比较。此外，允许修改迭代器指向的键值对元素<key,value>的 value 字段。

表5 散列映射普通迭代器类的API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	HashMap::iterator()	初始化迭代器，定位到第一个非空桶的第一个元素。	O(1)	
2	HashMap::iterator operator++(int)	后置++操作符：将迭代器移动到下一个节点	O(1)	
3	const K& operator*() const	返回当前节点存储的数据	O(1)	
4	bool operator==(const	判断两个迭代器是否指向不	O(1)	

	HashMap::Iterator& other)const	同的节点		
--	--------------------------------	------	--	--

任务 1.5.3 HashMap::const_iterator 类

HashMap::const_iterator 是用于遍历散列映射中键值对的只读迭代器类，提供以下常见操作如：移动、解引用和相等性比较。与 HashMap::iterator 不同，HashMap::const_iterator 不允许修改其指向的元素的值。你可能认为普通迭代器应该是够用的，但实际工程中提供 const_iterator 仍然有重要的设计考量和好处：如果 HashMap 是 const 容器，因为普通迭代器无法用于 const 容器，所以不能在 const 容器上进行遍历。所以需要提供 const_iterator 类型迭代器以确保 HashMap 在各种 const 场景下都能正常工作。

表6 散列映射只读迭代器类API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	HashMap::const_iterator()	初始化迭代器，定位到第一个非空桶的第一个元素。	O(1)	
2	HashMap::const_iterator operator++(int)	后置++操作符：将迭代器移动到下一个节点	O(1)	
3	const K& operator*() const	返回当前节点存储的数据	O(1)	
4	bool operator==(const HashMap::const_Iterator& other)const	判断两个迭代器是否指向相同的节点	O(1)	

任务 1.6. *高效的带权无向简单图

为了提高速度，带权无向简单图 $G=(V, E, W)$ (其中 V 代表顶点集合， E 代表边集合， W 代表权值集合)的数据结构使用两个数据结构来实现：双向邻接链表 **DbLinkedList** 和散列映射 **HashMap**。请按照下表要求的时间复杂度来实现图的具体操作，其中 $|V|$ 表示图 G 的顶点数， $|E|$ 表示图 G 的边数， d 是顶点的度数，假设 $|E|>|V|$ 。图的顶点数据类型通过模板参数实现，可使用任意类型（用 **Vertex** 表示顶点关键字类型模板），边的权值类型同样通过模板参数实现（用 **Weight** 表示边权值关键字类型模板）。为了确保功能的正确性，假设顶点数据类型和边权值类型均为可比较类型。

图的内部结构约定：如图 3 所示，每个顶点与一个整数一一对应，即顶点处理为整数顶点，被编号的顶点与“标签”无关，并在整个图内部结构顶点是整数，边是整数顶点偶对，与顶点“标签”无关。将邻接点使用整数编号而不是直接存储顶点关键字（**Vertex**）的方式，会使查询和存储更加高效。在输入输出时，可以通过双向映射表（例如 $\text{HashMap}<\text{Vertex}, \text{int}>$ 和 $\text{HashMap}<\text{int}, \text{Vertex}>$ ）实现顶点关键字（**Vertex**）与整数编号之间的映射。

Neighbors 类是一个专门设计的类，用于使方法 **WUGraph.getNeighbors()** 能够同时返回两个数组。**Neighbors** 类定义如下：

```
template<typename Vertex, typename Weight> class Neighbors
{
    ExpandableArrayList<Vertex> neighborArray;
    ExpandableArrayList<Weight> weightArray;
}
```

给定一个输入顶点，`getNeighbors()`方法返回一个 `Neighbors` 对象。`neighborArray` 是与输入顶点相连的所有顶点列表（由应用程序提供的类型，而不是图的内部顶点整数表示）。`weightArray` 列出每条边的权重。每次调用 `getNeighbors()` 时，都应构造并返回一个新的 `Neighbors` 对象。

表7 高效的带权无向简单图API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	<code>WUSGraph()</code>	初始化一个空图	$O(V)$	
2	<code>int vertexCount()const</code>	返回图中的顶点数	$O(1)$	
3	<code>int edgeCount()const</code>	返回图中的边数	$O(1)$	
4	<code>getVertices()const</code>	返回图的所有顶点。 返回值类型自行设计 。注意不要返回顶点的整数编号（一直是隐藏的），应返回应用程序在调用 <code>addVertex()</code> 时提供的顶点关键字（ <code>Vertex</code> ）。	$O(V)$ 如果可能，请考虑优化为 $O(1)$ 。	
5	<code>void addVertex(Vertex)</code>	添加顶点	$O(1)$	
6	<code>void removeVertex(Vertex)</code>	删除顶点	$O(d)^*$	
7	<code>bool isVertex(Vertex)const</code>	判断该顶点是否在图中	$O(1)^*$	
8	<code>int Degree(Vertex)const</code>	返回顶点的度	$O(1)$	
9	<code>Neighbors</code> <code>getNeighbors(Vertex)const</code>	输入一个顶点，返回一个 <code>Neighbors</code> 对象。	$O(d)^*$	
10	<code>void addEdge(Vertex, Vertex, Weight)</code>	添加边	$O(1)$	
11	<code>void removeEdge(Vertex, Vertex)</code>	删除边	$O(1)^*$	
12	<code>bool isEdge(Vertex, Vertex)const</code>	判断是否为图中的边	$O(1)^*$	
13	<code>Weight</code> <code>getWeight(Vertex, Vertex)const</code>	求某边的权值	$O(1)$	
14	<code>getEdges()const</code>	返回图的所有边。 返回值的类型自行设计 。确保返回的边信息与添加边时所使用的顶点关键字（ <code>Vertex</code> ）和权重（ <code>Weight</code> ）一致，而不是图的内部实现细节（如顶点的整数编号）。	$O(V + E)$ 如果可能，请考虑优化为 $O(1)$ 。可根据具体的数据结构设计。例如，如果图中的边信息已经缓存并能在常数时间内返回，可优化为 $O(1)$ 。	
15	<code>~WUSGraph()</code>	销毁图，并释放分配的所有存储空间。	$O(V + e)$	

*标注操作的性能得到显著提升。

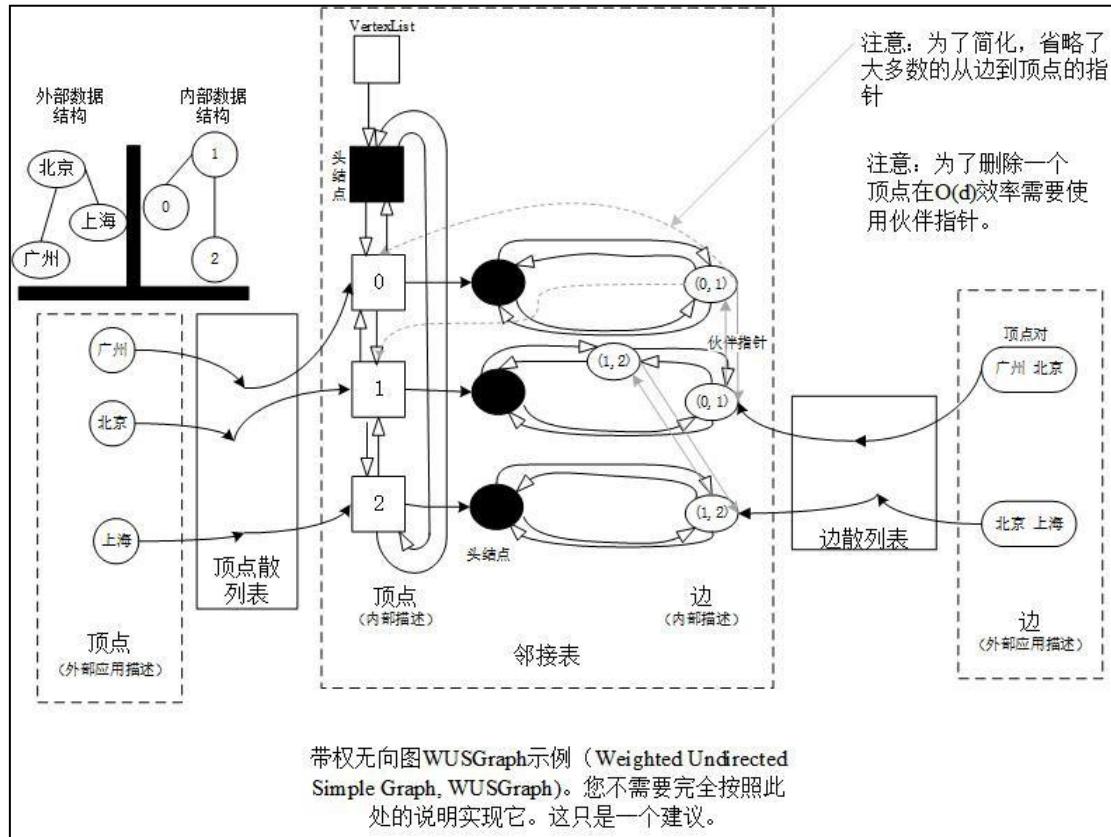


图3 *带权无向图WUSGraph 数据结构参考解决方案

参考解决方案: 为了实现表 7 中描述的高效操作方法, 我们采取以下设计思想 (参考图 3) :

(1) 内部顶点数据结构应始终是隐式的, 不能暴露给外部应用程序。为了实现外部应用 程序与内部顶点数据结构之间的高效映射, 最好的方法是采用散列映射 (HashMap)。具体地, 外部应用程序提供的每个顶点应通过散列映射到内部顶点结构, 同时, 内部顶点结构也 可以通过散列映射快速映射回外部应用程序的顶点表示。此外, 这个散列表还可以在 $O(1)$ 的时间内 支持 isVertex() 操作。实现时使用任务 1.5 中的散列映射 HashMap。

(2) 为了使 getVertices() 在 $O(|V|)$ (甚至 $O(1)$) 的时间内运行, 你需要维护一个顶点 列表。同时, 为了使 removeVertex() 的时间复杂度为 $O(d)$ (其中 d 为顶点的度数), 顶点 列表应当是一个双向链表, 这样可以在删除顶点时, 高效地进行删除。具体实现使用任务 1.3 中提供的双向链表 DbLinkedList。getVertices() 应返回调用程序在调用 addVertex() 时 提供的对象, 而不是无向加权图的内部顶点数据结构, 因为内部结构应始终对外部程序隐藏。 因此, 每个内部顶点表示应包含对调用程序使用的顶点对象的引用 (见参考附图中的虚线箭头)。 另一种实现 getVertices() 的方法是遍历散列表。但是, 只有当散列表能够双向调整大小, 这种方法的时间复杂度才能达到 $O(|V|)$ 。

(3) 为了确保 getNeighbors() 操作的时间复杂度为 $O(d)$ (即返回顶点的邻居边列表时), 你需要为每个顶点维护一个邻接边表。为了让 removeEdge() 的时间复杂度为 $O(1)$, 每个邻 接边表应当是一个双向链表。为使 removeVertex() 时间复杂度为 $O(d)$, 需要有从边到顶点的 指针。使用任务 1.2 中的双向链表 DbLinkedList。

(4) 在无向图中, 边 (u, v) 必须出现在两个链表中 (u 顶点和 v 顶点的邻接边链表), 除非 $u == v$ 。若要删除顶点 u , 则必须删除 u 顶点的邻居顶点的邻接边链表中所有与 u 相关

联的边。为了使 `removeVertex()` 操作的时间复杂度为 $O(d)$, 不能简单地遍历所有邻居顶点的这些邻接边链表, 可以尝试用下列方法来实现 $O(d)$ 的运行时间: 既然边 (u, v) 在图中出现在两个链表中, 则可以用两个结点来表示 (u, v) , 设分别在 u 表和 v 表中。这些结点每一个可称为“半边”, 互为“伙伴”。每一个半边节点具有指向前后节点的指针(用于链接邻接表), 还需要包含指向其伙伴的“伙伴指针”。这样, 当删除顶点 u 时, 可以遍历 u 的邻接边链表, 使用伙伴指针在 u 的邻居的邻接边链表中删除对应的伙伴边, 实现 $O(1)$ 的时间复杂度!

(5) 为在 **$O(1)$** 时间内支持 `removeEdge()`、`isEdge()` 和 `getWeight()`, 需要第二个散列映射, 用于将无序顶点对(由调用程序提供的两个顶点)映射到内部边的数据结构。使用任务

1.5 的散列映射 `HashMap`。使用半边表示法, 可以通过一个半边引用找到另一个半边。

需要设计边散列表的键的类型以及针对该键类型的哈希码函数以及 `operator==` 方法, 以确保顶点对 (u, v) 和 (v, u) 被视为相等的键, 具有相同的哈希值。

尽管可以将顶点和边放在同一个散列表中, 但这容易导致混乱。因此, 使用两个独立的散列表有助于调试和减少错误。

为了让 `removeVertex()` 的时间复杂度为 $O(d)$, 在从顶点的邻接边链表上删除边的同时, 也需要从边散列表中删除边, 且从顶点散列映射表中删除顶点, 同时更新顶点的度数。因此, 每条边或半边应包含对其关联顶点的引用。

(6) 为使 `vertexCount()`、`edgeCount()` 和 `Degree()` 操作的时间复杂度为 $O(1)$, 需要在内部数据结构中维护顶点数、边数以及每个顶点的度数, 并且在进行每次操作时, 都需要动态更新这些计数器。

`WUGraph` 的任何内部字段或类都不应公开, 所有这些仅用于图的内部表示。`Neighbors` 类是公共的, 因为它不属于图的内部表示。

第 2 部分 测试和应用带权无向图 API 开发带权无向图客户端及 API

高效带权无向图客户端 `WUSGraphClient` 类继承高效带权无向简单图 `WUSGraph` 类, 有插入顶点、插入边等公共方法, 此外, 具有表 8 中方法。第二部分实现的类、方法要满足下面封装要求: (表 8、9 和 10 中的方法) 图的外部应用不接触内部数据结构, 只可通过 `WUSGraph` API 访问图内部数据结构。

第二部分使用 `Vertex` 和 `Weight` 模板分别表示支持顶点关键字、边权值可以是任何类型(假设它们是可比较的)。`Path`、`Tree`、`Forest` 分别代表路径、树、森林类型。`Path`、`Tree`、`Forest` 采用父指针表示法。由于要满足封装要求, `Path`、`Tree`、`Forest` 不能接触图内部数据结构。可以再次使用 `HashMap`, 将外部应用顶点映射到唯一外部应用整数编号, 再利用数组的按下标索引和映射的按关键字索引实现。

自行设计表 8 中 `DFS()` 和 `BFS()` 函数形参的参数、`Steiner()` 和 `LongestPath()` 的函数参数, 实现 `Steiner()`、`LongestPath()` 方法前分析算法时空性能填入表 8。`Eula()` 是选做内容。

表 8 带权无向图客户端 API 以及性能(时间 空间)要求

序号	函数	函数功能	时间复杂度要求	空间复杂度要求	完成情况
1	<code>void CreateGraphFromFile(string</code>	从给定文件路径 <code>filepath</code> 读取文件, 构造图 <code>g</code> 。	$O(V + E)$	$O(V + E)$	

	filepath, WUSGraph& g)				
2	int MaxDegree(const WUSGraph&)	返回 g 顶点的最大度数。	O(V)	O(1)	
3	void Print(const WUSGraph& g)	输出 g 顶点集和边集。	O(V + E)	O(1)	
4	void DFS(WUSGraph& g, void (*visit) (Vertex))	深度优先遍历 g, 访问函数是 (*visit)()。	O(V + E)	O(V)	
5	void BFS(WUSGraph& g, , void (*visit) (Vertex))	广度优先遍历 g, 访问函数是 (*visit)()。	O(V + E)	O(V)	
6	void Kruskal(const WUSGraph& g, Forest& msf)	优化的Kruskal 方法求解g 最小生成森林 msf。	排序时 O(E log V)* 排序后 O(E α(V))	O(V + E)	
7	void Dijkstra(const WUSGraph&g, const Vertex &s,Forest& spf)	优化的Dijkstra 方法求解g 源点 s 的最短路径森林 spf。	O(E log V)*	O(V)	
8	void Prim(const Grah&g, const Vertex &s, Forest& msf)	优化的 Prim 方法求解 g 最小生成森林 msf。	O(E log V)	O(V)*	
9	void LongestPath(const WUSGraph& g, const Vertex &s, Forest& lpf)	求 g 源点 s 到其它各顶点的最长路径森林 lpt。	O(?)	O(?)	
10	void CreateSubgraph(const WUSGraph& G, const ExpandableArrayList<Vertex> & vertexlist, WUSGraph&g)	创建图 G 的子图, 由参数 g 返回。该子图 g 包含原图的顶点集合 vertexlist。			
11	void Steiner(const WUSGraph&g, const ExpandableArrayList< Vertex> & vertexlset, Forest& stf)	给定 g 的顶点子集 vertexset, 求解对应的斯捷纳森林stf。	O(?)	O(?)	
12	(选做 1) Eula()	判断存在后求解 g 欧拉回路。	O(E)	O(V)	

任务 2. 1. CreateGraphFromFile()

从文件以块读取方式读入顶点和边数据, 通过调用 WUSGraph API 将顶点和边信息存入到高效带权无向图对象的内、外部数据结构。

从外存读取文件到内存时, 以块 (block) 为单位, 借助磁盘缓冲区实现内外存数据交换。根据计算机原理可知, 从磁盘读写 1B 与读写 1KB 几乎一样快。为高效读取文件, 每一次读取应该尽可能读取更多的数据。因此采用块读取, 而不是一个字符一个字符、一个字 符串一个字符串、一行文本一行文本本地读取。

任务 2. 2. MaxDegree()返回图顶点的最大度数

任务 2. 3. Print()输出图所有顶点和所有边

任务 2. 4. 使用递归和队列实现图的遍历

任务 2. 4. 1. 选择并实现一种合适本课题的队列

任务 2. 4. 2. 深度优先遍历算法 DFS()

任务 2. 4. 3. 广度优先遍历算法 BFS()

任务 2. 5. 基于数据结构尽可能高效实现 Kruskal()求最小生成森林

任务 2. 5. 1. 实现最小堆类 MinHeap

堆是用于优先级功能的非常重要的数据结构。它在优先级排序方面的表现有助于我们有效地实现基于找到最小值、最大值的算法。

任务 2. 5. 2. 实现森林类 Forest

使用任务 1.5 中提供的 HashMap 存储顶点到其前驱和边权值的映射。

表9 森林类API 以及性能要求

序号	函数	函数功能	时间复杂度要求	空间复杂度要求	完成情况
1	构造 Forest()	构造森林	O(1)	O(V)	
2	void insertEdge(Vertex,Verte x,Weight)	插入一条边到森林	O(1)	O(1)	
3	void Print()	输出森林	O(V)	O(1)	

任务 2.5.3. 带路径压缩的加权快速连接并查集类 **WQUPCUFSet**

带路径压缩的加权快速连接并查集（Weighted Quick Union with Path Compression UFSets, **WQUPCUFSet**）并、查操作的性能要求 $O(\alpha(|V|))$ ，非常接近常数时间。 $|V|$ 是元素个数。

任务 2.5.4. **Kruskal()**求最小生成森林

使用任务 2.4.1-2.4.3 中的 **MinHeap**、**WQUPCUFSet**、**MinSpanForest** 实现 **Kruskal()**算法求出最小生成森林。你的实现应该在 $O(|E|\log|E|)$ 时间中运行：排序时 $O(|E|\log|E|)$ 、排序后 $O(|E|\alpha(|V|))$ ，其中 $|V|$ 是 G 的顶点数， $|E|$ 是 G 的边数。

在列出 G 中所有的边时，你不能通过对每对顶点调用 **isEdge()**来构建此列表，因为这样做需要 $O(|V|^2)$ 时间。你需要使用多次调用 **getNeighbors()**来获取完整的边列表。**封装要求：**图类的内部数据结构不接触应用（包括 **Kruskal**、**Prim**、**Dijkstra** 等所有图算法）。

使用改进的并查集求最小生成树的边时，需要一种方法将顶点关键字映射为唯一的整数或者顶点结点。如果两个不是它们各自的集合的根的顶点合并，或者如果顶点与其自身合并，会导致灾难性失败。如果你添加简单的错误检查，可能会为你节省大量调试时间。

任务 2.6. 基于数据结构尽可能高效实现 **Dijkstra()**得到最短路径树

任务 2.6.1. *实现最小索引堆 **MinIndexHeap**

在堆中发生交换时直接交换数组中两个元素，数组元素位置发生改变会导致元素与位置 关联信息丢失，以至于很难找到元素做修改。为此，使用两个数组，一个是存放数据的数组（数据不移动），一个是存放各个元素当前的位置信息的索引数组。当堆调整发生交换时，交换的是索引数组对应两个元素的位置，而不是交换数据数组的两个元素。这主要有两个好处：
1.减小交换操作的消耗，尤其是对于元素交换需要很多资源的对象来说，比如大字符串。
2.数据数组的元素不发生移动，可以在原位置找到元素，即便这个元素的索引已经修改。

E 表示堆中数据元素的模板。 $|V|$ 表示堆中当前元素的数目。

表 10 最小索引堆 API 以及性能要求

序号	函数	函数功能	时间复杂度要求	完成情况
1	MinIndexHeap(E arr[], int n)	大小为 n 的数组最小堆化	$O(V)$	
2	bool Insert(const E &x)	数组插入 x 后最小堆化	$O(\log V)$	
3	bool removeMin(E &x)	数组删除堆顶元素后最小堆化	$O(\log V)$	
4	bool Modify(int i,const E& e)	数组第 i 个元素值修改为 e 后，最小堆化	$O(\log V)^*$	
5	bool getMin(E &e)	得到最小项,通过参数 e 返回。	$O(1)$	

任务 2.6.2. 实现 **Dijkstra** 算法得到最短路径树

使用任务 2.5.1 的最小索引堆存放当前各顶点距离源点 s 的最短距离值。在 u 顶点找到

最短路径后，调用 `getNeighbours()` 获取 u 邻居。修正邻居的到达源点 s 的最短距离时，修改最小索引堆中相应的值和堆重新调整。你的 Dijkstra 算法应该在 $O(|V| \log|V|) + |E| \log|V|$ 时间中运行。

任务 2.7. 基于数据结构尽可能高效实现 **Prim()** 求解最小生成森林

使用任务 2.5.1 的 $|V|$ 个分量的最小索引堆存放当前 MSF 与各顶点的最小距离，第 i 个分量是当前最小生成森林顶点集 $V(MSF)$ 到第 v_i 个顶点的最小距离值。使用任务 2.4.2 的最小生成森林类。在 u 顶点加入到最小生成森林后，调用 `getNeighbours()` 获取 u 的邻居。修正邻居的到当前最小生成森林顶点集 $V(MSF)$ 的最短距离，修改最小索引堆中相应的值和堆重新调整。你的实现应该在 $O(|E|\log|V|)$ 时间中运行，空间开销应该是 $O(|V|)$ 。

任务 2.8. 求最长路径树

目标是寻找从起始顶点到其他每个顶点的最长路径树，要求路径为简单路径（不包含环）。注意：不能直接将权值取负后使用 Dijkstra 算法。因为 Dijkstra 算法假定所有边的权值为非负，并基于贪心策略逐步扩展最短路径。如果将权值取负，Dijkstra 的核心假设会被破坏，导致算法可能无法正确运行，甚至进入死循环。常用的方法有边松弛法（类似于 Bellman-Ford 算法）。查找相关资料，理解并选择适用于最长路径的算法。基于现有的图数据结构，设计辅助数据结构，以支持算法的高效实现求解最长路径。

任务 2.9. 求子图

给定一个带权图 $G(V, E, W)$ 和一个顶点关键字集合的子集 V' ($V' \subseteq V$)，返回图 G 的子图 g 。初始 g 是空图。将 G 中顶点在 V' 中，表示该顶点属于子图，将顶点加入 g ，然后在 G 中遍历与该顶点相连的边，如果目标顶点也在 V' 中，添加边到子图。

任务 2.10. 求解斯坦纳森林

给定一个带权图 $G(V, E, W)$ 和一个结点集合的子集 V' ($V' \subseteq V$)。斯坦纳树问题的目的是找到一棵包含这个子集 V' 中的所有结点且权重之和最小的子树。与最小生成树问题不同，斯坦纳树问题不需要包含图中的所有结点 V ，而只需关注子集 V' 。由于斯坦纳树问题是 NP 难的，我们通常采用启发式算法（例如修剪树法、迭代改进法）、近似算法（例如基于最小生成树的近似方法）或动态规划。查找相关资料，理解并选择适用于斯坦纳森林的算法。基于现有的图数据结构，设计辅助数据结构，以支持算法的高效实现求解斯坦纳森林。

任务 2.11. (选做 1) 求解欧拉回路

欧拉回路是指图中所有边均只被遍历一次的环路，每个结点可以被访问多次。查找相关

资料，理解并选择适用于欧拉回路的算法。基于现有的图数据结构，设计辅助数据结构，以支持算法的高效实现求解欧拉回路。

第 3 部分 路线规划系统类 RoutePlanningSystem

设有 n 个城市，每个城市都和相邻的若干个城市有直接的道路相通（每条道路有对应的距离），请设计城市路线规划系统 RoutePlanningSystem 类，并设计合理系统运行界面。

RoutePlanningSystem 类继承 WUSGraphClient 类。第三部分实现的类、方法要满足封装要求：图的外部应用（表 11 的方法）不接触内部数据结构，只可通过 WUSGraph API 和 WUSGraphClient API 访问图内部数据结构。表 11 中系统功能的函数名、参数、返回值自行设计。完成任务 3.16-3.18 后分析算法的时间性能，填入表 11。任务 3.20-3.21 是选做内容。部分任务后面有进一步说明。

先使用小规模数据 <ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-1.txt>、[usa-10.txt](ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-10.txt)、[usa-100short.txt](ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-100short.txt)、[usa-100long.txt](ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-100long.txt) 测试系统功能的正确（请使用 ftp 软件下载。数据格式说明参看：<https://www.cs.princeton.edu/courses/archive/spring04/cos226/assignments/map.html>），再使用 <ftp://ftp.cs.princeton.edu/pub/cs226/map/usa.txt> 大规模数据测试系统功能的效率。

表 11 交通路线规划系统功能与性能要求

序号	系统功能	时间复杂度要求	完成情况
任务 3.1	初始化城市路线规划系统（一个空城市交通库）	$O(1)$	
任务 3.2	城市的增/删/改和交通路线的增/删/改	$O(1)/O(d)/O(1)/O(1)/O(1)/O(1)$	
任务 3.3	从文件读数据建立城市交通库	$O(V + E)$	
任务 3.4	输出城市数/所有城市/相邻城市间的道路数（不重复）/所有道路（不重复）	$O(1)/O(V)/O(1)/O(E)$	
任务 3.5	计算图的稀疏程度	$O(V)$	
任务 3.6	计算地图的连通分量个数	$O(V + E)$	
任务 3.7	每一个连通分量是否有环且输出环路	$O(V + E)$	
任务 3.8	判断当前城市交通库中是否有给定城市/给定道路/输出某城市相邻的城市数/求某条道路的距离值	$O(1)$	
任务 3.9	输出某城市的所有邻接城市	$O(d)$	
任务 3.10	输出从给定顶点出发可以到达的所有顶点	$O(V + E)$	
任务 3.11	求相邻的城市数最多的城市	$O(V)$	
任务 3.12	输入任意两个城市，找出它们之间的最短简单路径和最短距离	$O(E \log V)$	
任务 3.13	从给定城市 s 出发，以与 s 的距离最小的城市为优先，请选出一系列道路，能够连接所有城市。输出这些道路的城市、距离。输出总距离。	$O(E \log V)$	
任务 3.14	从给定城市 s 出发，以与所选择的城市集合的距离最小的城市为优先，请选出一系列道路，能够连接所有城市，并且总距离最小。输出这些道路的城市、距离。输出总距离。	$O(E \log V)$	
任务 3.15	以最短的道路为优先，请选出一系列道路，能够连接所有城市，并且总距离最小。输出这些道路的城市、距离。输出总距离。	$O(E \log V)$	
任务 3.16	用户一键知晓周围所有城市：输入城市交通库的任意一个城市，系统	$O(?)$	

	就会显示周围 R 公里范围内的所有城市以及数目		
任务 3.17	用户输入感兴趣的一组无重复的城市后, 请选出一系列道路, 能够连接所有输入城市, 并且总距离最短。输出这些道路的城市、距离。输出总距离。	$O(?)$	
任务 3.18	输入任意两个城市, 找出它们之间的最长简单路径和最长距离。	$O(?)$	
任务 3.19	系统菜单界面		
任务 3.20	(选做 2) 判断是否存在每条边只使用一次的环路并且输出	$O(E)$	
任务 3.21	(选做 3) 系统图形界面		

任务 3.2. 城市和交通路线的增/删/改

城市的增加 $O(1)$

城市的删除 $O(d)$

城市的修改 $O(1)$

相邻两城市间道路的增加 $O(1)$

相邻两城市间道路的删除 $O(1)$

相邻两城市间道路距离的修改 $O(1)$

任务 3.3. 从文件读数据建立城市交通库

读取城市地图文件 (例如 <ftp://ftp.cs.princeton.edu/pub/cs226/map/usa.txt>)。该文件描述了一个地图, 该地图有 87575 个十字路口和 121961 条道路。该图非常稀疏, 平均度为 2.8。这个文件先罗列出地图的顶点和边的数量, 然后罗列出所有顶点的 x、y 坐标 (索引号后是顶点坐标, 可以假设所有的 x 和 y 坐标都是 0 到 10000 之间的整数), 每条道路对应的距离可以由坐标计算。文件然后罗列出边信息 (即顶点对), 最后罗列出测试任务 3.5.5 的源顶点和汇顶点。在系统启动时读取文件动态加载到内存构建图。

文件中顶点的索引号请理解为城市名, 不作为内部结构顶点编号, 也不作为外部应用结构顶点的编号。图的内部结点编号和外部应用结点编号均与文件中顶点的索引号无关。

任务 3.4. 输出当前的城市交通库信息

输出城市数 $O(1)$

输出所有城市 $O(|V|)$

输出相邻城市间的道路数 (不重复) $O(1)$

输出所有道路 (不重复) $O(|E|)$

任务 3.5. 计算图的稀疏程度 Sparseness()

无向简单图 G 的稀疏程度 $\text{Sparseness} = \text{顶点的平均度数}/(\text{顶点数}-1)$, $0 \leq \text{Sparseness} \leq 1$ 。输出 Sparseness。

任务 3.8. 地图查找

判断某城市是否在当前城市交通库中 $O(1)$

判断当前库中是否有给定道路 $O(1)$

输出某城市相邻的城市数 $O(1)$

求某条道路的距离值 $O(1)$

任务 3.12. 输入任意两个城市，找出它们之间的最短简单路径和最短距离

先使用小规模数据 <ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-1.txt>、[usa-10.txt](ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-10.txt)、[usa-100short.txt](ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-100short.txt)、[usa-100long.txt](ftp://ftp.cs.princeton.edu/pub/cs226/map/usa-100long.txt) 测试系统功能正确与否，再使用 <ftp://ftp.cs.princeton.edu/pub/cs226/map/usa.txt> 大规模数据测试系统功能的效率。

结束！