

**2021年春季学期  
计算学部《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 | 肖泽强 |
| 学号 | 1191000814 |
| 班号 | 1903001 |
| 电子邮件 | [3543226403@qq.com](mailto:3543226403@qq.com) |
| 手机号码 | 16673479299 |

**目录**

[1 实验目标概述 1](#_Toc72249750)

[2 实验环境配置 1](#_Toc72249751)

[3 实验过程 1](#_Toc72249752)

[3.1 Poetic Walks 1](#_Toc72249753)

[3.1.1 Get the code and prepare Git repository 1](#_Toc72249754)

[3.1.2 Problem 1: Test Graph <String> 1](#_Toc72249755)

[3.1.3 Problem 2: Implement Graph <String> 1](#_Toc72249756)

[3.1.3.1 Implement ConcreteEdgesGraph 2](#_Toc72249757)

[3.1.3.2 Implement ConcreteVerticesGraph 2](#_Toc72249758)

[3.1.4 Problem 3: Implement generic Graph<L> 2](#_Toc72249759)

[3.1.4.1 Make the implementations generic 2](#_Toc72249760)

[3.1.4.2 Implement Graph.empty() 2](#_Toc72249761)

[3.1.5 Problem 4: Poetic walks 2](#_Toc72249762)

[3.1.5.1 Test GraphPoet 2](#_Toc72249763)

[3.1.5.2 Implement GraphPoet 2](#_Toc72249764)

[3.1.5.3 Graph poetry slam 2](#_Toc72249765)

[3.1.6 使用Eclemma检查测试的代码覆盖度 2](#_Toc72249766)

[3.1.7 Before you’re done 2](#_Toc72249767)

[3.2 Re-implement the Social Network in Lab1 2](#_Toc72249768)

[3.2.1 FriendshipGraph类 2](#_Toc72249769)

[3.2.2 Person类 3](#_Toc72249770)

[3.2.3 客户端main() 3](#_Toc72249771)

[3.2.4 测试用例 3](#_Toc72249772)

[3.2.5 提交至Git仓库 3](#_Toc72249773)

[4 实验进度记录 3](#_Toc72249774)

[5 实验过程中遇到的困难与解决途径 3](#_Toc72249775)

[6 实验过程中收获的经验、教训、感想 4](#_Toc72249776)

[6.1 实验过程中收获的经验和教训 4](#_Toc72249777)

[6.2 针对以下方面的感受 4](#_Toc72249778)

# 实验目标概述

本次实验训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象

编程（OOP）技术实现ADT。具体来说：

1. 针对给定的应用问题，从问题描述中识别所需的 ADT；
2. 设计ADT规约（pre-condition、post-condition）并评估规约的质量；
3. 根据ADT的规约设计测试用例；
4. ADT的泛型化；
5. 根据规约设计 ADT 的多种不同的实现；针对每种实现，设计其表示 （representation）、表示不变性（rep invariant）、抽象过程（abstraction function）
6. 使用 OOP 实现 ADT，并判定表示不变性是否违反、各实现是否存在表 示泄露（rep exposure）；
7. 测试ADT的实现并评估测试的覆盖度；
8. 使用ADT及其实现，为应用问题开发程序；
9. 在测试代码中，能够写出 testing strategy 并据此设计测试用例。

# 实验环境配置

系统：windows10，JDK8

IDE：IntelliJ IDEA 2021

版本管理：TortoiseGit

代码托管：Github

在这里给出你的GitHub Lab2仓库的URL地址（Lab2-学号）。

https://github.com/ComputerScienceHIT/HIT-Lab2-1191000814.git

# 实验过程

请仔细对照实验手册，针对三个问题中的每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## Poetic Walks

在Java集合框架提供了许多有用的数据结构与对象的集合工作：列表，地图，队列，组，等等。它不提供图形数据结构。让我们实现图形，然后使用它们来创造无与伦比的辉煌的永恒艺术。我们将实现的类型是Graph<L>，带有标记顶点的可变加权有 向图的抽象数据类型。

可变图:可以在图中添加和删除顶点和边

有向边:边从源顶点到目标顶点

加权边:边与正整数权重相关联

标记的顶点:顶点由某种不可变类型的标签区分，例如它们可能有String名称或IntegerID

Graph<L>是类似于or的泛型类型。在 的规范中，是列表中元素类型的占位符。In ,和是键和值类型的占位符。只有稍后客户端才会通过构造这些占位符或为这些占位符选择特定类型，例如a,List<E>,Map<K,V>,List<E>,Map<K, V>,ListMap<K,V>,List<Clown>,Map<Car,Integer>

泛型类型的规范是根据占位符编写的。例如，规范Map<K,V>说 thatK必须是一个不可变类型：如果你想在a中创建一个键Map，它不应该是一个可变对象，因为Map可能无法正常工作。

在我们的 规范中Graph<L>，我们对类型的不变性提出了同样的要求L。Graph尝试使用可变顶点标签的客户违反了前提条件。他们不能期望正确的行为。

对于这个问题集，我们将实施Graph两次，用两个不同的代表，练习选择抽象函数和代表不变量并防止代表暴露。一个库提供一个类型的多个实现有很多很好的理由（例如，满足具有不同性能要求的客户的ArrayList和LinkedList实现List），我们遵循该模型。

### Get the code and prepare Git repository

如何从GitHub获取该任务的代码、在本地创建git仓库、使用git管理本地开发。

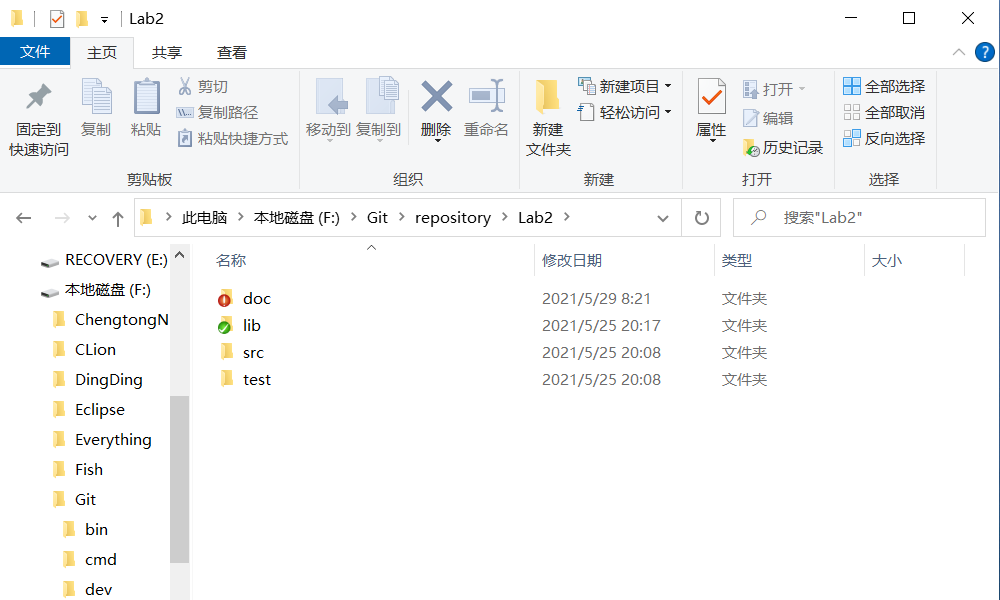
获取代码:



本地git仓库:



管理git版本信息:



### Problem 1: Test Graph <String>

设计、记录和实施Graph<String>.

(1)现在，我们将只测试（然后实现）带有String顶点标签的图。稍后，我们将扩展到其他类型的标签。为了适应在接口的多个实现上运行我们的测试Graph，这里设置：静态 Graph.empty()方法的测试策略和测试在GraphStaticTest.java. 由于该方法是静态的，因此只有一种实现，我们只需要运行一次这些测试。我们提供了这些测试。您可以随意更改或添加它们，但您可以将它们保留原样以解决此问题。

(2)编写测试策略和.cs文件中所有实例方法的测试GraphInstanceTest.java。在这些测试中，您必须使用该emptyInstance()方法来获取新的空图，而不是Graph.empty()!

(3)Graph­InstanceTest是一个抽象类。抽象类和子类化有其用途，但通常应避免使用。不像GraphStaticTest，它像我们编写的任何其他 JUnit 测试类一样工作，GraphInstanceTest是不同的，因为你不能直接运行它。它有一个空白等待填充：emptyInstance()提供空Graph对象的方法。在接下来的问题中，我们将看到两个子类的GraphInstanceTest通过返回不同类型的空图的空白填补。

(4)测试GraphInstanceTest必须是Graph规范的合法客户。任何特定于您的实现的测试都将在下一个问题的子类中进行。

### Problem 2: Implement Graph <String>

现在我们将实现带String标签的加权有向图-两次,对于在此问题集中编写的所有类：

(1)记录抽象函数和表示不变量。

(2)连同 rep 不变式，记录该类型如何防止 rep 暴露。

(3)实施checkRep以检查表示不变量。

(4)实现toString与该抽象值的有用人类可读表示。

(5)将来，您编写的每个不可变类型都将覆盖equals并hashCode实现第15 类关于等式中Object所述的契约，但这不是此问题集所必需的。

(6)您所有的类都必须有明确的文档规范。这意味着每个方法都有一个 Javadoc 注释，除非您使用@Override：

(7)当在接口中规定的方法，然后通过一个具体的类实现-例如，add(..)在ConcreteEdgesGraph中指定Graph-在@Override没有Javadoc注释注释指示该方法具有相同的规格。除非具体类需要加强规范，否则不要再写（DRY）。

(8)您可以选择ConcreteEdgesGraph或ConcreteVerticesGraph先写。您的两个实现之间不应该有任何依赖或代码共享。

#### Implement ConcreteEdgesGraph

(1).属性:

private final Set<L> vertices = new HashSet<>(); 所有组成图的顶点

private final List<Edge<L>> edges = new ArrayList<>(); 所有组成图中的边

边类Edge:

private final L source; 边起点

private final L target; 边终点

private final int weight; 边长

Edge类重写了equals和hashoCode方法,仅仅用name作为equals的依据

(2).方法checkRap():edge中每个成员的起点和终点都在vertices中,且边长度weight都大于0

(3).方法public List<Vertex<L>> getVertices():用防御性赋值返回属性vertices

(4).方法public int getWeight(L source, L target):返回从source到target之间的距离,不存在则返回-1

(5). @Override public boolean add(L vertex):在图中添加一个顶点对象,返回是否添加成功

(6). @Override public int set(L source, L target, int weight):设置两点之间的距离,并返回这个距离,设置失败返回-1

(7). @Override public boolean remove(L vertex):在图中删除该顶点,如无删除成功则返回true

(8). @Override public Set<L> vertices() :返回顶点组成的set集合

(9). @Override public Map<L, Integer> sources(L target):返回能到达指定点的点与距离的映射

(10) [.@Override](mailto:%20.@Override) public Map<L, Integer> targets(L source) 所有能由指定点达到的目标点集合与对应长度组成的映射

(11).该类也重写了hashCode与equals方法,用于比较两个图是否相等

#### Implement ConcreteVerticesGraph

(1).属性:private final List<Vertex<L>> vertices

由所有Vertex类对象的顶点组成的数组

顶点类Vertex:

private final L name; 顶点的名称

private final Map<L, Integer> map = new HashMap<>(); 顶点与其他顶点距离的映射

Vertex类重写了equals和hashoCode方法,仅仅用name作为equals的依据

(2).方法checkRap():Vertex类的name不为空字符串,vertices中的每一个对象的map属性中的keySet必须在vertices之中,且value都大于0

(3).方法public List<Vertex<L>> getVertices():用防御性赋值返回属性vertices

(4).方法public int getWeight(L source, L target):返回从source到target之间的距离,不存在则返回-1

(5). @Override public boolean add(L vertex):在图中添加一个顶点对象,返回是否添加成功

(6). @Override public int set(L source, L target, int weight):设置两点之间的距离,并返回这个距离,设置失败返回-1

(7). @Override public boolean remove(L vertex):在图中删除该顶点,如无删除成功则返回true

(8). @Override public Set<L> vertices() :返回顶点组成的set集合

(9). @Override public Map<L, Integer> sources(L target):返回能到达指定点的点与距离的映射

(10) [.@Override](mailto:%20.@Override) public Map<L, Integer> targets(L source) 所有能由指定点达到的目标点集合与对应长度组成的映射

(11).该类也重写了hashCode与equals方法,用于比较两个图是否相等

### Problem 3: Implement generic Graph<L>

(1).静态工厂:

public static <L> Graph<L> empty() {

return new ConcreteEdgesGraph<>();

}

(2)另外有六种抽象方法:add, remove, set, vertices, sources, targets

#### Make the implementations generic

根据已经写好的类型为String的类,将String全部改为泛型L即可

#### Implement Graph.empty()

public static <L> Graph<L> empty() {  
 return new ConcreteEdgesGraph<>();  
}

### Problem 4: Poetic walks

#### Test GraphPoet

通过测试并用Assert.assertEquals来检测答案是否正确:

public class GraphPoetTest{

@Test

public void testPoem() throws IOException{

File file1 = new File("src\\Lab2\\test\\P1\\poet\\test1.txt");

GraphPoet poet1 = new GraphPoet(file1);

String poem1 = poet1.poem("Seek to explore new and exciting synergies!");

String actualPoem1 = "Seek to explore strange new life and exciting synergies!";

Assert.assertEquals(poem1, actualPoem1);

File file2 = new File("src\\Lab2\\test\\P1\\poet\\test2.txt");

GraphPoet poet2 = new GraphPoet(file2);

String poem2 = poet2.poem("Test the system.");

String actualPoem2 = "Test of the system.";

System.out.println(poem2);

Assert.assertEquals(poem2, actualPoem2);

}

}

#### Implement GraphPoet

1.属性:

private final Graph<String> graph = Graph.empty(); 词亲和图

private final Set<String> words = new HashSet<>(); 图中的单词集合(区分大小写)

2.方法:

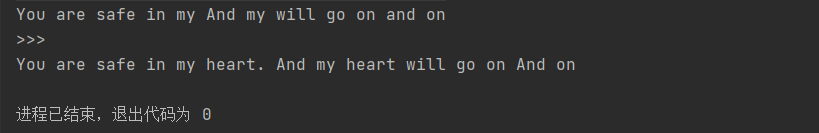
(1).构造方法: public GraphPoet(File corpus) throws IOExceptionHashSet<>();

根据所给的文本文件corpus,将文件中每一个区分大小写的单词都放入词库words中,再根据单词顺序,每两个相邻的单词(不区分大小写)之间都有一条边,边的权重为这两个单词相邻的次数,从而构造出一个ConcreteVerticesGraph类表示的有向图

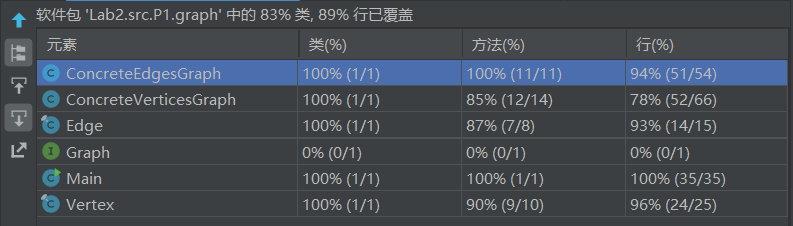
(2).poem:根据给定的一个另一个输入字符串，GraphPoet通过尝试在输入的每对相邻词之间插入一个桥词来生成一首诗。输入词“w1”和“w2”之间的桥接词将是这样一些“b”,使得w1 -> b -> w2是从 w1 到 w2 的权重最大的路径。如果没有这样的路径，则不插入桥字。在输出诗中，输入词保持原来的大小写，而桥词是小写的。诗中每个单词之间都有一个空格。

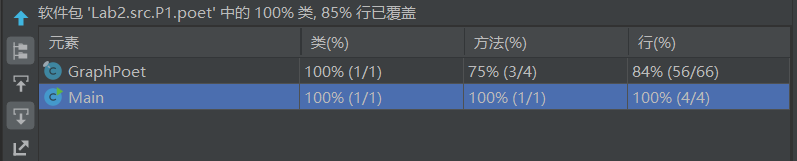
实现:将区分大小的词库和不区分大小写的词库分别放入两个列表中,遍历给出的字符串,在每两个单词之间按上述规则插入桥接词,生成一个全小写的新句子。然后遍历新句子每一个单词,如果是桥接词,则不用大写,如果不是桥接词,从words中找有无这个词的大写形式,如有则替换,最后得到的就是要返回的poem诗句

#### Graph poetry slam



### 使用Eclemma检查测试的代码覆盖度





### Before you’re done

请按照<http://web.mit.edu/6.031/www/sp17/psets/ps2/#before_youre_done>的说明，检查你的程序。

如何通过Git提交当前版本到GitHub上你的Lab2仓库。

在这里给出你的项目的目录结构树状示意图。

## Re-implement the Social Network in Lab1

回顾 Lab1 实验手册中的 3.2 节 Social Network，你针对所提供的客户端代码实现了 FriendshipGraph 类和 Person 类。 在本次实验中，请基于你在 3.1 节 Poetic Walks 中定义的 Graph<L>及其两种实现，重新实现 Lab1 中 3.3 节的 FriendshipGraph 类。 注 1：可以忽略你在 Lab1 中实现的代码，无需其基础上实现本次作业； 注 2：在本节 FriendshipGraph 中，图中的节点仍需为 Person 类型。故你的新 FriendshipGraph 类要利用 3.1 节已经实现的 ConcreteEdgesGraph<L> 或 ConcreteVerticesGraph<L>，L 替换为 Person。根据 Lab1 的要求， FriendshipGraph 中应提供 addVertex()、addEdge()和 getDistance()三 个方法:针对 addVertex() 和 addEdge() ，你需要尽可能复用 ConcreteEdgesGraph<L>或 ConcreteVerticesGraph<L>中已经实现的 add() 和 set()方法，而不是从 0 开始写代码实现或者把你的 Lab1 相关代码直接复制 过来；针对 getDistance()方法，请基于你所选定的 ConcreteEdgesGraph<L> 或 ConcreteVerticesGraph<L>的 rep 来实现，而不能修改其 rep。 注 3：不变动 Lab1 的 3.3 节给出的客户端代码（例如 main()中的代码），即 同样的客户端代码仍可运行。重新执行你在 Lab1 里所写的 JUnit 测试用例，测 试你在本实验里新实现的 FriendshipGraph 类仍然表现正常。

### FriendshipGraph类

1.属性:

ConcreteVerticesGraph<Person> graph = new ConcreteVerticesGraph<>();

2.方法:

(1). public Set<Person> vertices() 返回图中点组成的集合,使用了defensive copies

(2). public void addVertex(Person p) 向图中增加一个点

(3). public void addEdge(Person p1, Person p2) 向图中增加一条边,

(4). public int getWeight(Person pSource, Person pTarget) 返回源点到终点的直接距离

(5). public int getDistance(Person pBegin, Person pEnd) 返回两个对象之间的最短距离,使用了Dijkstra算法

(6). private int minDistance(Set<Person> pSet, Person p) 选出一个点集合中离一个点最近的距离,私有方法,被getDistance调用

### Person类

public class Person{

String name;

//有参构造函数

public Person(String name){

this.name = name;

}

//重写了equals函数,当name相等时对象相等

@Override

public boolean equals(Object o) {

if(this == o) return true;

if(o == null || getClass() != o.getClass()) return false;

Person person = (Person)o;

return Objects.equals(name,person.name);

}

@Override

public int hashCode() {

return Objects.hash(name);

}

}

### 客户端main()

// 模拟客户端同实验Lab1的代码  
public class Main{  
 public static void main(String[] args){  
 FriendshipGraph graph = new FriendshipGraph();  
 Person rachel = new Person("Rachel");  
 Person ross = new Person("Ross");  
 Person ben = new Person("Ben");  
 Person kramer = new Person("Kramer");  
 graph.addVertex(rachel);  
 graph.addVertex(ross);  
 graph.addVertex(ben);  
 graph.addVertex(kramer);  
 graph.addEdge(rachel, ross);  
 graph.addEdge(ross, rachel);  
 graph.addEdge(ross, ben);  
 graph.addEdge(ben, ross);  
 System.out.println(graph.getDistance(rachel, ross)); //should print 1  
 System.out.println(graph.getDistance(rachel, ben)); //should print 2  
 System.out.println(graph.getDistance(rachel, rachel)); //should print 0  
 System.out.println(graph.getDistance(rachel, kramer)); //should print -1  
 }  
}

### 测试用例

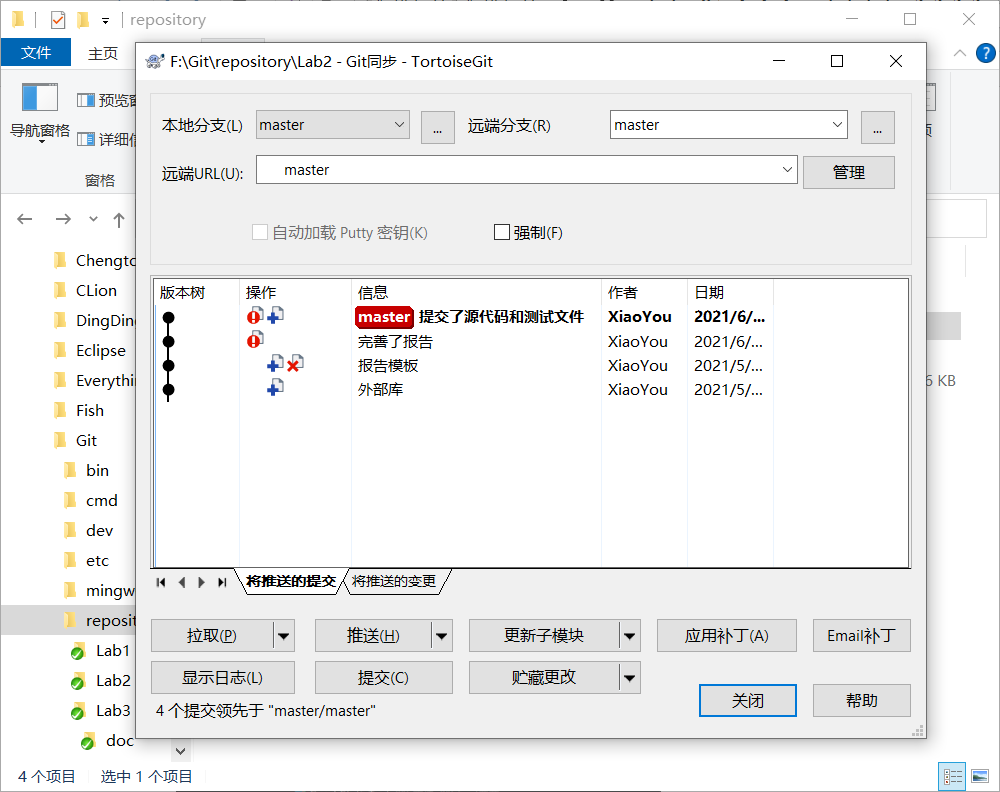
通过测试并用Assert.assertEquals来检测答案是否正确:

public class FriendshipGraphTest {  
 FriendshipGraph graph = new FriendshipGraph();  
 Set<Person> vertices = new HashSet<>();  
  
 Person rachel = new Person("Rachel");  
 Person ross = new Person("Ross");  
 Person ben = new Person("Ben");  
 Person kramer = new Person("Kramer");  
  
 @Test  
 // 测试addVertex和vertices函数  
 public void testAddVertex(){  
 graph.addVertex(rachel);  
 graph.addVertex(ross);  
 vertices.add(rachel);  
 vertices.add(ross);  
  
 Assert.assertEquals(graph.vertices(), vertices);  
 }  
  
 @Test  
 // 测试addEdge和getWeight函数  
 public void testEdge(){  
 graph.addVertex(rachel);  
 graph.addVertex(ross);  
 graph.addVertex(ben);  
 graph.addEdge(rachel, ross);  
  
 Assert.assertEquals(graph.getWeight(rachel, ross), 1);  
 Assert.assertEquals(graph.getWeight(rachel, rachel), 0);  
 Assert.assertEquals(graph.getWeight(rachel, ben), -1);  
 Assert.assertEquals(graph.getWeight(rachel, kramer), -1);  
 }  
  
 @Test  
 /\*测试getDistance函数\*/  
 public void testGetDistance(){  
 graph.addVertex(rachel);  
 graph.addVertex(ross);  
 graph.addVertex(ben);  
 graph.addVertex(kramer);  
  
 graph.addEdge(rachel, ross);  
 graph.addEdge(ross, ben);  
 graph.addEdge(rachel, ben);  
 graph.addEdge(ben, kramer);  
 graph.addEdge(kramer, rachel);  
 assertEquals(graph.getDistance(rachel, ross), 1);  
 assertEquals(graph.getDistance(rachel, ben), 1);  
 assertEquals(graph.getDistance(rachel, rachel), 0);  
 assertEquals(graph.getDistance(rachel, kramer), 2);  
 }  
}

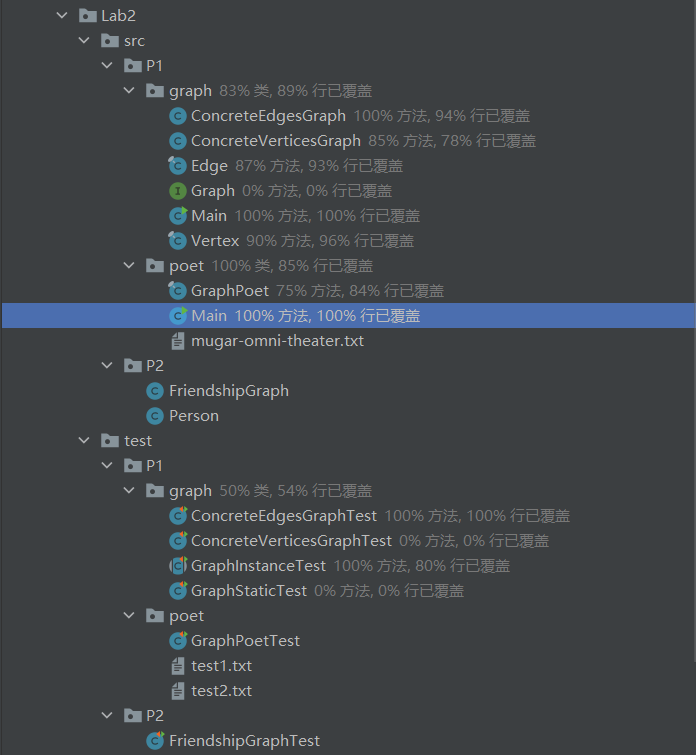
### 提交至Git仓库

如何通过Git提交当前版本到GitHub上你的Lab3仓库。

用TortoiseGit将本地仓库推送至远程仓库:



在这里给出你的项目的目录结构树状示意图。



# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 5.29 | 9:50 | 基本完成了Graph<String>的两个子类的设计,(不包括泛型) | 按计划完成 |
| 5.29 | 11:00 | 基本完成了非泛型的测试 | 按计划完成 |
| 5.29 | 11:30 | 基本完成了泛型的设计和测试 | 按计划完成 |
| 6.1 | 15:30 | 完成了P2及其Test测试 | 按计划完成 |
| 6.3 | 16:15 | 基本完成了GraphPoem类 | 按计划完成 |
| 6.5 | 11:30 | 完成了GraphPoemTest类 | 按计划完成 |
| 6.13 | 8:45 | 基本全部完成,准备提交 | 按计划完成 |
| 6.13 | 20:00 | 修改了FriendshipGraph的一个bug | 按计划完成 |
| 6.13 | 20:10 | 再一次push到远程 |  |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 不知道两个已经写好的测试testAssertions Enabled和graphStaticTest  是要测试什么函数的 | testAssertion不需要改,只是为了给出一个测试的示例,graphstaticTest用来测试类中的静态方法 |
| List接口中的empty函数不会写 | 在两个实现类中随便找一个的构造方法实现即可 |
| emptyInstance不会写,是否要将抽象方法变为实例方法,又是否要两个类都重写该方法? | 重新建两个GraphInstanceTest的子类,分别用返回Vertices和Edge的图重写emptyInstance方法 |
| testAdd出现空指针异常 | ConcreteVerticesGraph的map变量未初始化(new)  先把增强for循环换成普通for循环,然后方便调试,查看具体是哪里出现了空指针异常 |
| for(Vertex<Person> v: graph.vertices())的数据类型不匹配 | vertices方法只是添加this.name,而不是this, getVertices方法才是添加Vertex对象 |
| FriendshipTest的addEdge函数总是会在checkRap的时候抛出异常 | checkRap检查一个点的map对应所有终点都必须在vertices里,使用了List.contain方法,而Vertex类的equals和hashcode方法考虑了成员map,导致contain方法返回false,checkRap不能通过。Vertex类的equals和hashcode方法应该只考虑了成员name |
| poem slam不会弄,想不出有趣的东西 | 随便整了首英文歌词,虽然没什么新意,但还可以符合要求 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

更深入地理解了ADT的构造和原理, 学会了AF,RI和checkRap函数等新知识并将其运用到写的代码上,加深了对上课学过知识的印象

## 针对以下方面的感受

1. 面向ADT的编程和直接面向应用场景编程，你体会到二者有何差异？

ADT编程复用性高,可以重复使用,且程序模块化,清晰易懂提高了编程效率。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

使用泛型更加抽象化,能适应的范围更广,但是代码的适应性相对较低,不能直接拿来使用,需要经过一些改进,但是也提高了复用性。

1. 在给出ADT的规约后就开始编写测试用例，优势是什么？你是否能够适应这种测试方式？

优势是明确了内容和条件,编写程序时思路能更加清晰,但是我不太喜欢这种测试方式,因为给出的规约很多时候都不是最优的,我喜欢自己设计属性和方法。

1. P1设计的ADT在多个应用场景下使用，这种复用带来什么好处？

降低成本和开发时间; 经过充分的测试, 可靠安全bug少; 标准化,在不同程序中保持一致。

1. P3要求你从0开始设计ADT并使用它们完成一个具体应用，你是否已适应从具体应用场景到ADT的“抽象映射”？相比起P1给出了ADT非常明确的rep和方法、ADT之间的逻辑关系，P3要求你自主设计这些内容，你的感受如何？

适应了,我更偏好于P3的自主设计,给了我更多发挥的空间。

1. 为ADT撰写specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，这些工作的意义是什么？你是否愿意在以后编程中坚持这么做？

防止泄露, 保证每一步的正确性, 意义是保证信息的安全和保密性, 我愿意这么做。

1. 关于本实验的工作量、难度、deadline。

工作量较大, 难度一般, deadline较轻松,没有明显的时间压力。

1. 《软件构造》课程进展到目前，你对该课程有何体会和建议？

讲的太快了! 太快了! 孩子一直在掉队的边缘徘徊