SAN DIEGO STATE UNIVERSITY

CS310: Program 2

Maps and Trees

Due Date:	2018-NOV-04	Late Work:	Up to 72 hours
Points:	100	Penalty:	−10% per 24

1 Overview

Maps associates unique keys with their corresponding values, and they provide an essential function to software developers, for they allow one to store and quickly retrieve entries from within the data structure. Access to this tool simplifies several computational problems, (e.g., calculating word and bigram frequency, looking up a record from a corresponding identification number).

In this assignment, students shall construct two concrete data structures implementing a common Map interface. The project exposes students to a classical data structure – the binary search tree – while simultaneously exploring Java generics and the *Comparable* design pattern.

Student paired partners shall each implement the instructor-provided Map interface by creating two concrete classes which each satisfy the ADT's requirements. The first method incorporates *composition* and simply adapts a `java.util` compliant `TreeMap` to meet the interface specification.

Next, students shall implement the required `IMap` methods directly in a new concrete class consisting almost entirely of their own code. In this solution, students may not use the `java.util` data structures or algorithms (except where noted as a return value) and must demonstrate low-level Java knowledge in their software.

2 Requirements

All submitted programs require pair-programming; no single person teams permitted. For assistance acquiring a partner, submit an email to the class instructor. Individual submissions may not receive a grade.

Students must implement the algorithms and data structures identified in this assignment directly from scratch (except where noted), and they may not use applications or assignments developed for a different course.

Instructor Provided Files

All students must download the instructor provided interface file from the class website. This file defines the methods required by the student created

data structures. An error in this file may result in the final submission failing to compile.

File	Package	Description
IMap.java	edu.sdsu.cs.datastructures	A simple map interface

2.1 Coding Style

In general, all submitted software must:

- Adhere to a consistent naming convention and follow best practices for variable naming.
- **None** of the classes developed for this assignment shall include **any** *public* functions or methods unspecified in the instructor provided interface files.
- Use multiple, small private, or protected, internal methods to help clean the code.
- Keep method size to a minimum.
- Remain free of commented out blocks of code.
- Use consistent formatting [1] with spaces, not tabs.
- Line indentation shall be 4 spaces.
- The maximum column width is 80 characters.

Should students wish to include method comments for private internal methods, they may do so freely, but it may not be required with a meaningful method name.

2.2 Student Generated Files

Students must create the following files, named precisely as indicated, to receive credit.

[1] most IDEs provide a method for automatically reformatting code.

File	Package	Description
BalancedMap.java	edu.sdsu.cs.datastructures	Created through composition with `java.util.TreeMap`
UnbalancedMap.java	edu.sdsu.cs.datastructures	Entirely student code

All student files must include a comment for the file indicating each partner's name. No additional class files shall appear in the project. Each of the identified project classes may, however, include private inner classes (e.g., `IteratorHelper`, `MapEntry`).

Students shall not modify the instructor provided files, for the grading procedure uses the instructor's versions, and any student changes may result in a compilation failure.

2.3 Concrete Map Classes

Both maps created for this project shall implement the instructor provided `IMap.java` interface (see Section 6 for reference). The student provided data structures must meet the performance requirements, both for correctness as well as computational complexity.

Both student created classes must include two public constructors: default and copy. The default constructor takes no parameters and initializes the data structure to an empty state. The second constructor, known as a copy constructor, takes an `IMap` as its single parameter and initializes the new data structure with each of the entries in the parameter object.

The use of the `equals()` method attached to every object remains undesirable, for it will not yield correct performance. All items stored in the data structures designed for this assignment exclusively contain `Comparable` items. Consequently, students shall use the `compareTo()` method defined therein when testing for equivalence. This behavior remains *essential*. See section 5 for an example of its use.

Iterable Object

Several of the method signatures provided in the interface file indicate an `Iterable` return type. Classes adhering to this type must provide an `Iterator` object upon request. To complete this requirement, students may use a `java.util.LinkedList`, for it implements the `Iterable` interface.

2.3.1 Balanced Map

The code produced for this class simply *adapts* a `java.util.TreeMap` to the Instructor provided interface file. This new data structure must use composition with a `java.util.TreeMap`. Many of the methods created in this class will simply consist of method calls to the internal `TreeMap`.

2.3.2 Unbalanced Map

This implementation requires each student team produce the data structure code directly from scratch. Students shall produce an unbalanced binary search tree. This class will include the private, internal `Node` class the data structure as well as any helper classes necessary.

The binary search trees created for this assignment must adhere to the binary search tree convention of placing small items in the left-subtree and larger items in the right-subtree.

Students may use either the in-order predecessor or successor when deleting nodes with two children from the tree.

As with the balanced map, students may use a `java.util.LinkedList` to fulfill the `Iterable` requirements.

Creation of this class will require students to create a private, inner-class representing the map's node objects.

Students may find by finding a way to create a common entry set privately simplifies producing the key set and values.

3 Delivery

Students must perform an electronic submission of the Java source code produced for this project through the supplied class accounts on the Edoras university server. This process requires students create the necessary folder structure on the remote server, so they will likely use secure shell and secure copy to accomplish this task.

The grading script will attempt to collect the completed work by recursively copying the following directory: `~/submission/program2`

Students should simply recursively copy their project's source folder – complete with its directory structure – to this folder for proper credit. Students shall not submit a copy of the target directory or any of the .class files generated during the build cycle.

4 Grading

In general, all submissions must meet every requirement in this specification. Incorrectly named files, placed in the wrong package, will cause the grading script to fail. This will also result in a **substantial** penalty.

Students only receive credit for this assignment if their names appear in the source code's initial comment. This applies to every student-generated file. Unnamed students receive **zero** points on that portion of the assignment.

Requirement	Points
2.1 Coding Style	5
2.2 Student Files	5
2.3.1 Balanced Map	30
2.3.2 Unbalanced Map	60

5 Other Notes

The instructor and teaching assistants remain available to assist students with questions that arise during the two data structure's development and design. Due to high demand, however, students may not receive attention. Start early and communicate frequently with one's partner to mitigate this situation.

Neither the instructor nor the TAs will run the grading script for students prior to delivery and the official grading time. Neither the instructor nor the TAs will test the students submissions prior to delivery.

Maven Settings

Students creating this project through the IDEA compiler should use the following settings to simplify setup:

Project Type:	Maven
Project SDK:	Java 1.8
Archetype:	org.apache.maven.archetypes:maven-apache-quickstart
GroupId:	edu.sdsu.cs
ArtifactId:	program2

For this project, students should make certain to enable Auto-Import of Maven dependencies if requested.

Comparable

The `IMap` interface uses generic objects of type Comparable. When inserting them into the `UnbalancedMap`, students must use the `.compareTo()` method on the key to establish its proper position. The following code, assumed to exist within the internal `MapNode` class, illustrates how to perform a comparison using this interface.

```
public int compareTo(MapNode otherNode) {
  return ((Comparable<K>)key).compareTo((K)otherNode.key);
}
```

Of note, the comparison returns either zero (when equal) or a non-zero number where the sign indicates the relation between the two objects (typically -1 and +1).

Secure Copy

Prior to the due date, teams shall submit their software to the required folder in their edoras class accounts. Doing so on a Linux or MacOS system remains fairly simple through use of the SCP program. An example follows:

```
scp -r ./src csscXXXX@edoras.sdsu.edu:~/submission/progam2
```

6 Interface (Reference)

```
package edu.sdsu.cs.datastructures;

/**
 * The interface for a basic Map/Dictionary Abstract Data Type.
    This structure
 * associates keys and values.
 *
 * @param <K> The object type to use as the Key
 * @param <V> The generic object type to use as Values
 */
public interface IMap<K extends Comparable<K>, V> {

    /**
     * Indicates if the map contains the object identified by the
        key inside.
     *
     * @param key The object to compare against
     * @return true if the parameter object appears in the structure
     */
    boolean contains(K key);

    /**
     * Adds the given key/value pair to the dictionary.
     *
     * @param key
     * @param value
     * @return false if the dictionary is full, or if the key is a
        duplicate.
     * Returns true if addition succeeded.
     */
    boolean add(K key, V value);

    /**
     * Deletes the key/value pair identified by the key parameter.
     *
     * @param key
     * @return The previous value associated with the deleted key or
        null if not
     * present.
     */
    V delete(K key);
```

```
/**
 * Retreives, but does not remove, the value associated with the
    provided
 * key.
 *
 * @param key The key to identify within the map.
 * @return The value associated with the indicated key.
 */
V getValue(K key);

/**
 * Returns a key in the map associated with the provided value.
 *
 * @param value The value to find within the map.
 * @return The first key found associated with the indicated
    value.
 */
K getKey(V value);

/**
 * Returns all keys associated with the indicated value
    contained within the
 * map.
 *
 * @param value The value to locate within the map.
 * @return An iterable object containing all keys associated
    with the
 * provided value.
 */
Iterable<K> getKeys(V value);

/**
 * Indicates the count of key/value entries stored inside the
    map.
 *
 * @return A non-negative number representing the number of
    entries.
 */
int size();

/**
 * Indicates if the dictionary contains any items.
 *
 * @return true if the dictionary is empty, false otherwise.
 */
```

```
    boolean isEmpty();

    /***
     * Returns the map to an empty state ready to accept new entries.
     */
    void clear();

    /**
     * Provides an Iterable object of the keys in the dictionary.
     * <p>
     * The keys provided by this method must appear in their
         natural, ascending,
     * order.
     *
     * @return An iterable set of keys.
     */
    Iterable<K> keyset();

    /**
     * Provides an Iterable object of the keys in the dictionary.
     * <p>
     * The values provided by this method must appear in an order
         matching the
     * keyset() method. This object may include duplicates if the
         data structure
     * includes duplicate values.
     *
     * @return An iterable object of all the dictionary's values.
     */
    Iterable<V> values();

}
```