

p1 Implement and Test an ADT

[Read First](#) | [Overview](#) | [Files](#) | [Requirements](#) | [Instructions](#) | [Submit](#)

Read First

- [Student Conduct](https://canvas.wisc.edu/courses/202692/pages/academic-conduct) (<https://canvas.wisc.edu/courses/202692/pages/academic-conduct>)
- TAs will be available for [TA Lab Consulting](https://canvas.wisc.edu/courses/202692/pages/teaching-assistants) (<https://canvas.wisc.edu/courses/202692/pages/teaching-assistants>)
- [Programming Assignment: Requirements](https://canvas.wisc.edu/courses/202692/pages/programming-assignment-requirements) (<https://canvas.wisc.edu/courses/202692/pages/programming-assignment-requirements>)
- [Learning Outcomes p1](https://canvas.wisc.edu/courses/202692/pages/Learning%20Outcomes%20p1?titleize=0) (<https://canvas.wisc.edu/courses/202692/pages/Learning%20Outcomes%20p1?titleize=0>)




Overview


Implement, test, and compare alternate implementations of an abstract data type (ADT). The ADT is generic and defines operation for storing and retrieving (key, value) data pairs, where each key is Comparable and unique and has an associated value.

Files

Links are to files that can be downloaded individually or together in one file: [p1.zip](https://canvas.wisc.edu/courses/202692/files/13055945/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055945/download?wrap=1>) (contains all of these files and a couple of others)

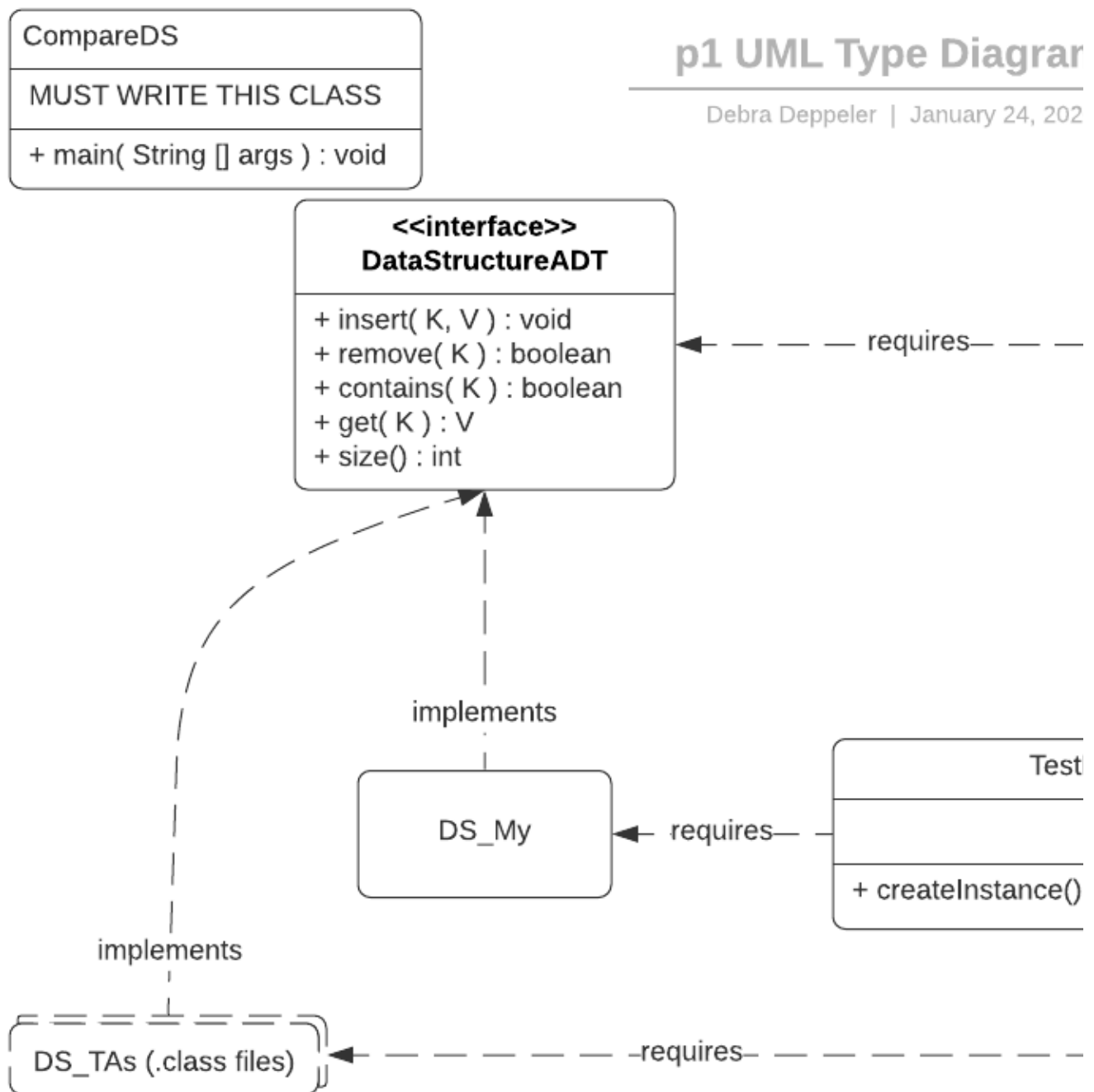
Given:

- [interface DataStructureADT](https://canvas.wisc.edu/courses/202692/files/13055948/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055948/download?wrap=1>)  (<https://canvas.wisc.edu/courses/202692/files/13055948/download?wrap=1>) - a generic interface for storing key,value pairs (DO NOT EDIT or SUBMIT)
- [abstract class DataStructureADTTest](https://canvas.wisc.edu/courses/202692/files/13055949/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055949/download?wrap=1>)  (<https://canvas.wisc.edu/courses/202692/files/13055949/download?wrap=1>) - super class for other test class (Must EDIT and SUBMIT)
- [class DS_My](https://canvas.wisc.edu/courses/202692/files/13055950/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055950/download?wrap=1>)  (<https://canvas.wisc.edu/courses/202692/files/13055950/download?wrap=1>) - class that will implement DataStructureADT (Must EDIT and SUBMIT)

- [class TestDS_My](https://canvas.wisc.edu/courses/202692/files/13055947/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055947/download?wrap=1>) - a subclass of  (<https://canvas.wisc.edu/courses/202692/files/13055947/download?wrap=1>) - a subclass of DataStructureADT that will test your DS_My (No need to edit)
- [classes.zip](https://canvas.wisc.edu/courses/202692/files/13055952/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055952/download?wrap=1>) - a compressed file containing several **DS_ta.class** files that each must be tested. DS_ta.class files are implementations of the interface created by the TAs and instructor for the spring 2020 version of CS400. (Do not DELETE, EDIT, or SUBMIT this file)
- [p1.zip](https://canvas.wisc.edu/courses/202692/files/13055945/download?wrap=1) (<https://canvas.wisc.edu/courses/202692/files/13055945/download?wrap=1>) - a compressed file contain all of the above files. (If downloaded, it must be unzipped and installed as instructed below)
- **Class Diagram** (below) - a Unified Modeling Language (UML) style diagram provided to help you interpret and understand the relationships between various types (interfaces, abstract classes, and classes) of this project.

The diagram shows that **DataStructureADTTest** is a *super-abstract-class*, and that there are multiple **Test_DS** classes that sub-class DataStructureADTTest and implement the abstract method *createInstance()*. This means that you must define *test* methods in the *super-type* and then define the ***createInstance()*** method for each sub-type. Note: It is because *createInstance* method is *abstract* (not defined) that the super-class is also *abstract*. Compare provided code with this diagram to help you understand how the design is reflected in a UML diagram.

Note: The arrow is from the sub-class to its super-class because it is the sub-class that knows its super-class.



Requirements

1. Must implement interface **DataStructureADT** in class **DS_My** without using any built-in Java collection types. Use arrays or your own custom linked structures, not `ArrayList`, `LinkedList` or other built-in types.
2. Must leave the existing JUnit test methods and define new tests in **DataStructureADTTest**. (**DataStructureADTTest** is the abstract JUnit *super-class* that has a few example JUnit tests and

where you will add more tests.)

3. Must define the sub-class type **TestDS_My** and run this test class to test your implementation of interface **DataStructureADT**.

4. Must **add new JUnit tests** to **DataStructureADTTest** super-type.

Add tests to ensure that your implementation works correctly. Rerun **TestDS_My** after each test add or edit.

You may use Java's built-in types **ArrayList** and **LinkedList** in the tests you write but not in your ADT implementation.

5. Must **define new sub-class test types** for each provided TA implementation:

(Example: Must define **TestDS_Sapan** to test **DS_Sapan**, and so on. Hint: copy, paste, edit your **TestDS_My**)

For each data structure implementation class provided in **classes.zip**, you must write a test class. In this case, we will be giving you many implementations to test so that means many test classes are required. However, we wish for the tests run for each test class to be the same. This is why we have designed this project to use inheritance. By inheriting tests from a super-type, we reduce the amount of code that needs to be written and still ensure that all implementations are tested with all tests.

6. Must **run JUnit tests for all DS_types**, and save screenshot(s) of test results

7. Must write a new **class CompareDS** that compares the performance of your implementation **DS_My** against any of one of the TA implementations that works (passes all of your tests).

CompareDS must:

1. must be a public class named **CompareDS** in a file named **CompareDS.java**
2. must contain a **public static void main(String [] args)** method that does not require user input to run
3. must output the names of the classes being compared
4. must produce some output that describes your performance test using **DS_My** for a very large number of inserts and retrievals (large is 1,000,000,000 or more) and do the same comparison for a TA class being compared.
5. Output must display a brief description of the work done for each trial, run multiple trials for each data structure and data set size. At a minimum, compare the time it takes to do the work for N items where the test is run and reported for both implementation for values of N equal to 100, 1000, 10000, 100000.

YOU DO NOT HAVE TO MATCH THIS EXAMPLE OUTPUT, just include the same or similar information:

```
CompareDS.main Compares work time for: DS_My and DS_ta-name-here
Description: <description of what work you do for each trial>

DS_My is doing work for 100000 values
It took 28131728 ns to process 100000 items
DS_TA is doing work for 100000 values
It took 1831728 ns to process 100000 items
```

Tip: You can use these Java functions to get the current time in nano seconds and then compute the elapsed time between a saved start time and a new end time.

```
long startTime = System.nanoTime();

// work code to be timed for a single run

long endTime = System.nanoTime();
```

You may notice a wide variation between runs. Multiple attempts will give you best idea about how long it takes on average, but you do not have to store or compute average run times for comparison.

Note: You are given TA implementations of the ADT in the form of Java byte-code (class) files. You will need to extract the *classes.zip* file into a classes folder and edit Eclipse project's build configuration to mark that folder as an "external folder" before you can use them in your code. See details below.

Instructions

CAUTION: the class files were compiled using Java v11, they will not compile if you are still using Java 8.

Using the JUnit framework means that you can focus on what tests to define and how to define them. When you run JUnit tests in Eclipse you get a graphic display showing green for passed tests and red for tests that failed. Here are the steps to create an Eclipse project that can run JUnit 5 tests. Different IDEs will likely have similar, but different instructions.

The green highlights indicate instructions that students like to skip. Please read and follow these instructions.

1. Get provided code to compile and run

1. Create a New Java Project: **File -> New -> Java Project**
 1. **Do not use separate src and bin folders.**
 2. **Disable module-info.java creation.**
 3. **For even more details on creating projects in Eclipse: [p1 Getting Started In Eclipse](https://canvas.wisc.edu/courses/202692/pages/p1-getting-started-in-eclipse) (<https://canvas.wisc.edu/courses/202692/pages/p1-getting-started-in-eclipse>)**
2. Add a JUnit Test class: **File -> New -> JUnit Test Case**
 1. remove the package name from the wizard dialog
 2. select the "**New JUnit Jupiter Test**" option, then Finish
 3. select "**Add JUnit 5 library to Build Path**", then Finish
 4. We will not need this TestClass, so it can be deleted from the project.

3. Copy the assignment starter files to your project folder ([p1.zip](#) (<https://canvas.wisc.edu/courses/202692/files/13055945/download?wrap=1>) contains all files)
 1. download [p1.zip](#) (<https://canvas.wisc.edu/courses/202692/files/13055945/download?wrap=1>) to your Eclipse project folder
 2. unzip p1.zip (you should have java files, a Makefile, and classes.zip file)
 1. If the files were placed in a sub-folder **p1**, move all files up one directory to the project folder.
 3. unzip **classes.zip** (if you do not have a classes folder with **DS_*.class** files already present)
4. Get provided code to compile.
 1. Add unimplemented methods to DS_My
 2. Add *stub* (fake) return values as needed for each unimplemented method to compile.
 3. Resolve any other compiler problems before continuing.
5. Run the **TestDS_My** test class
 1. Double-click to open the **TestDS_My.java** file in the Eclipse editor.
 2. **Run -> Run As -> JUnit Test**
 3. If the above steps do not work, try
 1. right-click on your project
 2. choose **Refresh**
 3. Repeat

Note: An alternative way is to download the JUnit 5 library and place a copy of the **junit-platform-console-standalone-1.5.2.jar** in your project folder. This step should not be necessary with Java v11. This file is also included in the p1.zip file. If you wish to try this option, you will likely need to add this file to the build path:

1. Right-click on the **jar** file
2. Select **Build Path**
3. Select **Add to Build Path**

2. Define required Tests in the DataStructureADTTest class

Now, that your project is setup, you are ready to add tests in the **DataStructureADTTest** class. We have some required tests to help you get started. Complete the **DataStructureADTTest** class that has been started for you. This class includes comments to help guide your work at the start. But, it is not complete. There is one complete test and you must define tests as described here and you must define additional tests to detect any problems. Recall: sequences that should throw exceptions must also be tested.

ProTip: Try running and check your tests after each test edit and update.

These tests have been provided to help you understand the test pattern:

1. **test01_insert_one()** - insert one key value pair into the data structure and then confirm that `size()` is 1.
Tip: use a call to **`fail("message string")`** or **`assertEquals(leftsidevalue,rightsidevalue)`** to get it count as a fail. If no fail occurs, it counts as a pass.
2. **test02_insert_remove_one_size_0()** - insert one key,value pair and remove it, then confirm size is 0.
3. **test03_duplicate_exception_thrown** - insert a few key,value pairs such that one of them has the same key as an earlier one. Confirm that a `RuntimeException` is thrown.
4. **test04_remove_returns_false_when_key_not_present** - insert some key,value pairs, then try removing a key that was not inserted. Confirm that the return value is false.

These tests must be written by you:

1. **test05_insert_remove_one** - inserts one item and fails if unable to remove it
2. **test06_insert_many_size** - inserts many items and fails if size is not correct
3. **test07_duplicate_values** - inserts two pairs with different keys, but same values; fails if second doesn't insert
4. TODO: add more tests to ensure that you can detect implementations that fail in some other way(s).

Tip: consider different sequences of inserts and removes. Check results of method like: `size()`, `get()`, `contains()`, `remove()`. Can you figure out if those methods work or not?

Run your tests on the **DS_My** class by running the **TestDS_My** JUnit test class. Open the file and click the run button. Assuming that you have not yet completed your version, we expect most tests to fail.

3. Complete **DS_My** (your class implementation)

Implement your own data structure in the class **DS_My**. Be sure that **DS_My** implements `DataStructureADT` and does not have any *public* or *package* level fields or inner classes. You should also keep you methods private, except for the ones that implement methods from the interface `DataStructureADT`. As you implement each method in the class, run your **TestDS_My** class. You should be able see more and more tests passing as your implementation is completed. This process may also help you to identify new things to test for.

WARNING: Your **DS_My** may not use any Java Collection classes as internal data members. You must implement using your own arrays, linked lists, or other. Your **DS_My** implementation may use `String` for key type and `String` for value type.

4. Add classes as an *external class folder*

You will need to do more configuration, to add TA implementation classes as external folder.

Your project should have a **classes** folder that will contain the TA implementations that must also be tested. You must add this folder as an "*External Class Folder*". Don't worry if the files are GONE. Eclipse deletes class files that do not have a corresponding source file until you place them in a folder that is designated as an *external class folder*.

Just keep unzipping and re-copying them into your project's classes folder as needed until you get the project configuration correct with classes as an external folder.

1. Right-click the "**classes**" folder within your project folder
2. Select **Build-Path -> Configure Build Path**
3. Select **Libraries** tab
4. Select "**Add External Class Folder**"
5. Find the **classes** folder in the project workspace
6. **Apply and Close**
7. Now, you can extract the files from classes.zip into the classes folder and they will not be deleted by Eclipse.

5. Create TestDS_* classes for each sub-class of DataStructureADTTest

To test each TA class, you must make a new sub-class of the **DataStructureADTTest** class for each Test class.

1. Right-click the **TestDS_My** class
2. Click **Ctrl-C** to copy
3. Click **Ctrl-V** to paste the copy in the same folder
4. Set the name as **TestDS_TaName** (where TaName is replaced with the particular TA implementation to be tested)
5. Edit the source code in the newly created class so that it constructs an instance of the particular **DS_** type
6. Save and repeat for all other DS implementations

6. Try running your tests on the TA implementation classes

1. Open and run each **TestDS_** class independently to test that test class.
2. If the compiler complains that it can not find the various DS_TaName types, you may need to unzip the class files into the classes folder again. See or repeat Step 4 for this folder.

7. Save a Screenshot of your test results

After you complete your tests, make the additional Test classes, and complete your implementation in **DS_My** , **run All Tests** and take a screen shot of the results.

Edit the Run Configuration to run all tests at once.

1. Run -> Run Configurations
2. Select "Run All Tests" from the configuration dialog
3. Name the configuration "Run All Tests"
4. Apply and run the tests
5. Take a screenshot of your test results (they must be your results) and name it *screenshot.png*.
Make sure that you include all your test results in the screenshot(s) (pass / fail for each test method in the suite) and the overall results (number of passed and failed tests).

8. Add additional tests to DataStructureADTTest

This is an individual project and each student must figure out their own tests for this assignment. These ideas will help you get started for things to test on your data structure.

Additional Test Ideas

- test that a key can be re-added if the key was inserted and then removed (this should not be a duplicate)
- check that it can store at least 1,000 items, should be able remove all of them too
- think of your own additional tests to ensure that you can detect problems

It's your development process that will win the day!

Use [Incremental Development](#)

(<https://canvas.wisc.edu/courses/202692/pages/Incremental%20Development?titleize=0>) and try [Test Driven Development](#) (<http://agiledata.org/essays/tdd.html>) strategies right from the start.

Finally, don't forget to Submit your latest work (daily or more frequently):

[p1 Testing, Debugging and Performance](#)

(<https://canvas.wisc.edu/courses/202692/assignments/833515>)

