Project 2
Heap

Out: Tuesday, February 26th
In: Friday, March 15th 11:59pm
"Booyah!"
-Ron Stoppable

1 Installing, Handing In, Demos, and Docs

1. To install, type `cs0160_install <project_name>` into a shell. The script will create the appropriate directories and deposit stencil files into them.
2. To compile your code, type `make` in your project directory. To run your code and launch the visualizer, type `make run` in the same directory. Make sure that your `Makefile` is in the same directory as your code.
3. To run tests, run `make run_tests` from your project directory.
4. To hand in your project, go to the directory you wish to hand in, and type `cs0160_handin <project_name>` into a shell
5. To run a demo of this project, type `cs0160_runDemo <project_name>` into a shell.
6. The NDS4 (youll be using some data structures from this package) documentation can be found here: `http://cs.brown.edu/courses/cs016/static/files/docs/nds4/index.html`. This is also linked off of the class website.
7. Remember to not include any identifying information in your hand in.

2 Using Eclipse

If you would like to use Eclipse, you may certainly do so. However, please make sure that you are using Eclipse Luna, not Eclipse Photon. You can launch Eclipse Luna by running `eclipse &`. In order to set up your project and make Eclipse work with the support code, you'll need to do the following:

- Select `File` → `New` → `Java Project`
  - Enter "<project_name>", lowercased, for example "heap", for the project name.
  - Un-check the box saying "Use default location" and in the "Location" box, enter `/gpfs/main/home/<your-login>/course/cs0160/<project_name>`
  - Click "next"
  - Under the "libraries" tab choose "Add External JARs..."

  - Select `/course/cs0160/lib/cs0160.jar`
  - Select `/course/cs0160/lib/nds4/nds4.jar`
  - Select `/course/cs0160/lib/junit-4.12.jar`
  - Select `/course/cs0160/lib/hamcrest-core-1.3.jar`
  - Click "Finish"
  - If it isn't already made for you, use `File` → `New` → `Source` Folder to create a new source folder in your new project named "src".
  - Use `File` → `New` → `Package` to create a new package in your new src folder. This package should have the same name as the project, for example "heap", and move all of the stencil java files (including the Makefile) into this package. Ignore any errors.

- Right-click on App.java and select `Run As` → `Java Application`. Now you can run your program by pressing the green play button at the top of your screen and selecting "Java application" if prompted

- To run the tests in Eclipse, you can right-click on `TestRunner.java` and click `Run As` → `Java Application`.

- Alternatively, if you want to run a test file, you can right-click on that file and select `Run As` → `Junit test`

- To configure your Eclipse projects to run over FastX or SSH, follow these setup steps.

  - Right click on the package icon next to the project name. Go to properities.
  - Go to Run/Debug Settings, select the main window App and click Edit.
  - Go to the arguments tab and, and enter `-Dprism.order=sw` in the VM arguments block
  - Hit Apply and OK
  - You should be all set to work on this project remotely with Eclipse. Make sure to do this for each new project.

2.1 Working Locally

In order to do heap on your own computer, you will have to copy the .jar files listed above from the department filesystem into your local directory. You will also need to transfer the stencil code. Follow the instructions below to transfer everything!

- If you haven't set up ssh yet, follow the instructions here: **PC** or **OSX**.

- In your personal terminal, type `sftp <your login>@ssh.cs.brown.edu` and enter your password that you normally use for ssh.

- Using the `cd` command, navigate into your cs0160 folder. Note: You cannot run the install script using sftp, so make sure to install using ssh.
- Using the `get` command, you can transfer your file to your personal computer. The command is `get -r heap <path to destination on personal computer (i.e Desktop)>`.
- Then cd into the cs0160 course lib folder using `cd /course/cs0160/lib`
- You should be able to find the .jar files using `ls`! The `nds4.jar` file is located in the `nds4` directory within `lib`.
- Using the get command again, transfer the .jar files. Note: For individual files, do not include `-r` after the get command, so for example: `get cs0160.jar Desktop`.
- Once all the .jar files and stencil code are on your personal computer, follow the eclipse setup instructions above!
- Note: If you have trouble setting up, feel free to ask a sunlab consultant for help!
- To handin, use the `put` command to transfer your project folder back onto the department filesystems. Make sure to check that the transfer was successful and everything works, and then run the handin script through ssh.

3 Introduction

3.1 Silly Premise

Shego and Dr. Drakken have teamed together to ruin Kim Possible and Ron Stoppable's prom. Thankfully, someone told Kim about their evil plan, but now she has to balance stopping Shego and Drakken, doing schoolwork, getting ready for prom, and so much more. Ron wants to help her, but he needs your help implementing a heap that will allow Kim to store her tasks in a prioritized way so she can do it all.

3.2 Before You Begin

Make sure to *thoroughly* read the handout **before** you start coding.

3.3 Task Overview

In this assignment you will implement a heap, which is an implementation of a priority queue. The underlying data structure of your heap will be a binary tree.

3.4 Purpose

The purpose of this assignment is to help you:

- Understand a link-based binary tree and its use in implementing a heap-based priority queue.
- Learn about using comparators.
- Introduce JUnit testing to test the functionality of your project implementation.
- Increase your familiarity with throwing exceptions and using try/catch blocks.
- Practice coding efficiently to meet specific runtime requirements.

4 Your Task

We have provided stencil code for the following classes: `MyLinkedHeapTree<E>`, `MyHeap<K,V>`, `MyHeapEntry<K,V>`, `MyHeapTest`, and `MyLinkedHeapTreeTest`. Your job will be to fill in all of these classes, which are described below:

1. `MyLinkedHeapTree<E>`: This class extends an existing binary tree implementation (`LinkedBinaryTree<E>`) to ensure left-completeness. Recall that a heap must be implemented with a *left-complete* binary tree, and since the existing `LinkedBinaryTree<E>` is not guaranteed to be left-complete, you must fill in this class to add left-complete functionality.

    - This class is an implementation of the `CompleteBinaryTree<E>` interface (from NDS4 library), and extends the NDS4 `LinkedBinaryTree<E>` data structure.
    - Since much of the binary tree functionality of this class is inherited from `LinkedBinaryTree<E>`, you should read the documentation for this class carefully in order to understand how the class you are extending operates.

    Note: The generic `E` represents the type of the elements that the tree is capable of holding. Due to the fact that `E` is generic, think carefully about where you want much of the heap functionality to lie.

2. `MyHeap<K,V>`: This class represents the heap itself, relying on an underlying left-complete binary tree (your `MyLinkedHeapTree<E>`) to hold its data.

    - This class is an implementation of the interface `AdaptablePriorityQueue<K,V>`, from the NDS4 package.

  - The difference between normal and adaptable priority queues is that adaptable priority queues have the ability to replace the key or value of an entry *after* it has been added, and additionally have the option of removing from the priority queue an item that is not at the top of the heap.

    Hint: In lecture we talked about a normal priority queue, so think about the special cases that come with altering nodes in the middle of the queue.

- `K` and `V` are generics that represent the types of the keys and values for the key/value pair of each element in the heap. For more information on Generics, see the **Java Concepts** section.

3. `MyHeapEntry<K,V>`: This class represents an element that will be stored in the heap. Elements of our heap will have mutable key/value pairs. (See the **Design Considerations** section for information on the difference between a `MyHeapEntry` and a `Position`)

   - Your implementation will require several additional accessors and mutators that will allow you to perform all of the actions associated with an adaptable PQ.
   - This class will implement the `Entry<K,V>` interface

4. `MyHeapTest` and `MyLinkedHeapTreeTest`: These classes will be used to unit test your MyHeap and MyLinkedHeapTree implementations. We provide stencil code for how tests should be formatted, but you are expected to write a plethora of your own comprehensive tests. Carefully read through the **Testing** section to see how this is done!

5 Design Considerations

When working through the design process, here are a number of things that you should consider:

- **Positions vs. Entries**: A `Position` is a tree node. It is a container for an `Entry`: once created, a `Position` stays in the same place in the tree until it is deleted. However, `Entry`s can be moved among `Positions` freely. Notice that the NDS4 `Position<E>` interface contains a generic, `E`, which is the type of the *element* that a `Position` contains. `Positions` are tied down in a single location on the heap because it's too much work to re-link parent and child nodes every time you want to up-heap or down-heap. Instead, the `Entry`s contained *within* the `Positions` move, swapping themselves between `Positions` when called for heap-order restoration.
- **Entry Key/Value Pairs**: `Entry`s contain a key/value pair (`K` and `V`) and are the data-containing *elements* of your heap. For example, the entries used in the visualizer consist of an integer key between 1 and 99 and a string value of three characters.

- **Extra MyHeap Methods**: In this project, we ask you to fill in the stencil class `MyHeapEntry<K,V>`. Think about the `replaceKey(...)` and `replaceValue(...)` methods of your `MyHeap` class. It doesn't make sense to simply instantiate a new `MyHeapEntry<K,V>` every time you need to alter the contents of an existing one. Perhaps some mutator methods would be helpful? More substantially, you need to think about the `Entry`'s relationship with the `Position` holding it. Why does an `Entry` need to know its location in the heap? What should happen when an `Entry`'s key is changed?

- **"First" and "Last" Nodes**: One of your major tasks in this project is to extend the NDS4 class `LinkedBinaryTree<E>` so that it represents a left-complete binary tree—your `MyLinkedHeapTree` class. Doing so means ensuring that the tree always knows where its "last" node is (the node to be moved to a different position within the tree in case of a `remove()` call), and where the next node to be inserted should go. Note that the stringent running time requirements of the methods in `MyHeap<K,V>` depend on the `MyLinkedHeapTree`'s `add(...)` and `remove()` methods being implemented efficiently. This means that the methods you implement in the `MyLinkedHeapTree<E>` must run in O(1) time! More on this in the next bullet...

- **O(1) "Node-Tracking" Algorithms:** We're not explicitly going to tell you how to implement your "node-tracking" in `MyLinkedHeapTree<E>`. We covered one possible traversal-based option in class, but this solution is worst case $O(\log n)$. It's up to you to find a better solution. Any solution that is worst-case $O(1)$ is acceptable! There are several algorithms that run in worst-case $O(1)$ time but they will require some thinking. All the classes of the NDS4 package are at your disposal, and anything that will make the `add(...)` and `remove()` methods of `MyLinkedHeapTree` run in O(1) time and with reasonable space usage is acceptable. Feel free to do a sanity check with a TA. You will develop one solution as a section exercise (using a `Deque`, a data structure in NDS4) which is the recommended approach.

6 Java Concepts

6.1 Comparators

You will be using a comparator to compare key values. Because your keys are generic (you do not know what type they'll be), you of course cannot just use normal Java comparison symbols ($>, <, ==$) to compare keys. A comparator is any class that implements Java's `Comparator` interface and is capable of comparing objects of a particular type. For example, the heap visualizer in the support code uses an `IntegerComparator` to compare integers. A comparator for objects of generic type `K` (the same type as your keys) must be passed into the constructor of `MyHeap.java`. You can use this comparator to compare your keys. For more details on how comparators work, refer to the Javadocs, which can be found here: `http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html`.

6.2 Generics

Java Generics allow a type to be replaced with some placeholder value - think of it as a variable for a class type. They are especially useful when implementing data structures as it allows the instantiator of the data structure to define which data type they would like it to store.
You've seen generics before when instantiating ArrayLists. The class `ArrayList<E>` is defined with generic type `E`. In the past, you've probably created ArrayLists that store Integers (e.g. `new ArrayList<Integer>`). Similar to `E`, the `K` and `V` in `MyHeapEntry<K, V>` are placeholders for the key's type and the value's type, respectively. Note that the letters `K` and `V` are a matter of convention - any letter could be used in practice.

7 Exceptions

You will need to throw a number of exceptions for this project, each of which is well-defined in the stencil code, and below.

- Every method that requires an exception is specified in the stencil code, so you do not need to come up with any other times when these exceptions should be thrown.
- You **must** identify (within a given method) the conditions that will cause a problem which necessitates throwing an exception, and write code to check whether these conditions exist.

Here is the list of the exceptions that you will have to handle:

- `IllegalStateException`: An exception from the `java.lang` package intended to be thrown when a method has been invoked at an illegal or inappropriate time. In this project, we are using it to signal that a priority queue cannot change the comparator it is using unless it is currently empty. This exception is thrown in `MyHeap.setComparator(...)`.
- `EmptyPriorityQueueException`: An exception from NDS4 that indicates that a priority queue cannot fulfill the requested operation because it is empty. This exception is thrown in `MyHeap.min()` and `MyHeap.removeMin()`—as you would expect, since you can't look at/remove elements from an empty heap.
- `InvalidKeyException`: An exception from NDS4 indicating that the key which is being handled has been determined to be invalid. The `Comparator` in use is the only object that can determine a key's validity. The `Comparator` has a `compare(...)` method which can compare two keys. You can use this method to test whether a key is valid: if you pass an invalid key as an argument to your `Comparator`, it will throw a `ClassCastException`. Thus, you can use the `Comparator` to compare a key to itself; if the `Comparator` throws an exception, you should catch it, and throw your own exception— the `InvalidKeyException`. Don't forget that a `null`

key is also invalid. `MyHeap.insert()` is the only method with a signature explicity throwing this exception, but you may find that it makes sense to also throw it from `MyHeap.replaceKey(...)`.

- `InvalidEntryException`: An exception from NDS4 that is thrown when the `Entry<K,V>` that is being handled is invalid. You'll find this in the code, but you don't need to worry about it. We've taken care of throwing it in our implementation of the `checkAndConvertEntry()` method.

- `EmptyTreeException`: An exception from NDS4 indicating that the tree cannot fulfill the requested operation because it is empty. Note that this is very similar to the `EmptyPriorityQueueException` which was described above. In fact, you should never see *this* exception thrown from your heap, because `EmptyPriorityQueueException` should "intercept" the same condition first. However, you still need to throw this exception from your `MyLinkedHeapTree.remove()` method. This may seem silly, but it is good coding practice to make your code extensible; that is, you know that in this project you are using your `MyLinkedHeapTree` underneath your `MyHeap` class, and thus will catch this condition with an `EmptyPriorityQueueException`, but including *this* exception will ensure that your `CompleteBinaryTree<E>` implementation is usable in other contexts in which you may not necessarily catch an empty priority queue condition elsewhere in the code.

- `IllegalArgumentException`: A Java exception that is typically thrown when bad input is given to a function or method. This should be thrown if the setComparator() method is passed in a null comparator. While you may never do this, it's good coding practice to make your code bulletproof against all possible things that people could do with it.

8 Testing

Introducing... JUnit testing! JUnit is a commonly used testing framework for *unit tests*, which focuses on testing individual bits and pieces of a program (for example, a single method) rather than the program as a whole. This will be extremely helpful in pinpointing the exact source of any bugs/funky functionality you may encounter. For this project and all future projects, you will be required to write your own unit tests for all of the methods you have implemented. Don't worry- we've provided ample stencil code so you can focus on writing your tests instead of setting up JUnit.

In your heap directory, you will see two files named `MyHeapTest.java` and `MyLinkedHeapTreeTest.java`. These contain stencils with JUnit setup code and example tests. You should use these stencils as a guide to writing many more test functions for `MyHeap` and `MyLinkedHeapTree` in their respective test files. For comprehensive instructions on writing JUnit tests, see the guide on our website:

http://cs.brown.edu/courses/cs016/static/files/docs/javaunittesting.pdf.

- Your testing should be comprehensive- i.e. it should ensure that all methods you have filled in or written from scratch function properly. You should write a **separate** JUnit test for each method you implement. Remember to include the `@Test` annotation above each of your test methods, and to initialize a new, fresh `MyHeap/MyLinkedHeapTree` object for each test.
- Your testing should also ensure that your implementation handles all possible edge cases and throws exceptions as expected. You will need to write a separate JUnit test for each exception to test that it has been thrown properly. Remember to add the `(expected = <some exception here>.class)` annotation next to `@Test` for these tests. See the stencil code in `MyLinkedHeapTreeTest` for an example of how this is done. Please clearly indicate which edge case the test handles in its header comment.
- For every test you write, please include in its header comment a detailed description of the test's purpose, which edge case the test handles (if any), and anything else notable about the test.

To run all of your tests in Eclipse, right-click on the TestRunner.java file in the file tree on the left side of the screen and choose "Run As → Java Application." To run just one of your test files, right-click on that file and choose "Run As → JUnit Test." You will see your tests run in a new tab on the left side of the screen. To run your tests from the terminal, type `make run_tests` from your heap directory.

9 Requirements

1. Fill in the three classes `MyLinkedHeapTree<E>`, `MyHeap<K,V>`, and `MyHeapEntry<K,V>`. The methods you need to fill in for each class are specified in the stencil code.
2. Fill in `MyHeapTest` and `MyLinkedHeapTreeTest` with your own comprehensive tests. Refer to the **Testing** section above for more details on how to do this.
3. The methods you write for this project **must** adhere to strict runtime requirements– this will be a significant portion of your grade. These requirements are listed in the method comments in the stencil code. Make sure that you note any assumptions that you make in the comments above the method.
4. You must throw appropriate exceptions when necessary, see the **Exceptions** section above for more details.
5. A detailed README file that is explained below.

9.1 README

The README for this project has some specific points that you need to address. Make a text file entitled "README.txt" which should be saved in the same directory as your project. Each of the questions should be answered in about 1-4 lines.

- Discuss the design choices you made in your `MyHeapEntry<K,V>` and `MyHeap<K,V>` implementations. If you defined your own methods, explain why you chose to add them.
- Describe the method you used to keep track of where to add and remove nodes in the tree.
- Describe the running time of the `MyLinkedHeapTree<E>` methods.
- If there is anything very notable about your JUnit testing that you want to point out (that wasn't explicitly included in the test file's header comment) you should include it here. [Note: a good rule of thumb is that your comprehensive list of tests should cover each line of code that you have written.]
- Include anything else particularly notable about your implementation.
- If you note any bugs in your code, we will be more lenient with respect to those bugs in the grading of your project.
- Whether you plan to hand in this project again in the near future (allowing us to start grading earlier).

In addition to the above points, please see the "README Guide" on the Docs page of the website for an explanation of what we are looking for in a CS16 project README.

10 What to Hand In

1. Code for the classes `MyHeap`, `MyHeapEntry`, `MyLinkedHeapTree`, `MyHeapTest`, `MyLinkedHeapTreeTest`, and an unchanged `App`.
2. A README.txt.

Note: You must now hand in from `~/course/cs0160/heap/`, but if you use Eclipse and your code is in a lower subdirectory, that is fine since the handin script searches subdirectories. Be sure to verify that all of your files are listed in the confirmation email sent by running the handin script.