Implementing a heap

1. Introduction

A5 involves implementing a heap —either a min-heap or a max-heap— with the added functionality of being able to change a priority. You will see an example of a common phenomenon: *When functionality is added, a data structure may have to be changed or a new one added to keep it all efficient*.

The result of this assignment is used in A6, which is to implement an algorithm to find a shortest path in a graph.

Keep track of how much time you spent on A5; we will ask for it upon submission.

One semester, a similar assignment took a mean and median of 4.2 and 4.0 hours. Do it right, and you can get by with writing less than 60 lines of code. We suggest getting A5 done *well* before the due date.

So that A5 doesn't take *too* long, we give you a complete JUnit testing class. If your code passes the test cases, your code *should* be correct. However, the possibility does exist that you did something entirely weird that is incorrect but passes the test cases, and we may dock you points for that.

There are LOTS of test cases, trying to catch all the weird things people have written. Using white-box testing, we could look at your code and determine good test cases. But we don't see your code until you submit it! So we have to rely on what we saw in previous semesters in which implementing a heap was required.

You may lose many points for not adhering to the execution-time bounds given in the specifications of the methods. For example, if an operation should be done in expected constant time and your code searches an array in linear fashion, that will cost points. Finally, points may be deducted for not following all the instructions.

Collaboration policy and academic integrity

You may do this assignment with one other person. Both members of the group should get on the CMS and do what is required to form a group well before the assignment due date. Both must do something to form the group: one proposes, the other accepts.

You should work together. For example, it is a violation of academic integrity for two people to write and test separate required methods and both people get credit for both of them. That is not working together.

With the exception of your CMS-registered group partner, you may not look at anyone else's code, from this semester or earlier ones, in any form, or show your code to anyone else (except the course staff), in any form.

Getting help

If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders.

2. Installing the release code

The zip file release.zip contains an Eclipse project with two .java files. They belong in package `heap`.

1. File `Heap.java`. The methods you must write are marked with //TODO comments, giving the order in which method bodies should be written and tested and giving information on testing. Do not remove the //TODO comments. You should *never* begin writing a method until the preceding ones pass all the tests for them.
2. File `HeapTest.java`, which has testing procedures to completely test the methods you write.

These two files go in package `heap`. Start a new project, called perhaps `a5`, create the package, and copy the two files into the package directory. You will have to get JUnit5 on the build path for `HeapTest.java` to compile.

3. What to do for this assignment

Your job is to implement and test the methods in class Heap that are marked "//TODO". Write them and test them in the order given! Do not move on to another method until the one(s) you are working on are correct. The methods are: `ensureSpace`, `insert`, `swap`, `bubbleUp`, `peek`, `bubbleDown`, `poll`, and `changePriority`. Below, in the suggested timetable, we give suggestions on how to proceed, including what to read when.

4. Suggested timetable

The methods form natural groups to program together:

0. Thu. 8 Apr. Study the following sections carefully. Scan the rest of this note so you have an idea what's in it.
   7. *The basic class invariant*.
   8. *A new field, map*.
   9. *Is it a min-heap or a max-heap*?
   10. *Using an array for the heap*.

1. Fri. 9 Apr. Write and test methods `insert` and `ensureSpace`. Before beginning to test, read sections 11 and 12, on understanding the testing class and debugging.

2. Sat. 10 Apr. Methods `swap` and `bubbleUp`.

3. Sun. 12 Apr. Methods `peek`, `bubbleDown`, and `poll`.

4. Tue. 13 Apr. Method change`Priority`.

5. Dos and don'ts

1. **Check pinned Piazza post *Assignment A5* regularly, perhaps every other day**. We post necessary changes there. You are responsible for them.
2. **Don't declare any more fields**. We declared all necessary fields in class `Heap` and wrote a class invariant for it.
3. **Implement the methods marked with "//TODO" *so that they have the specified time bounds***. Points will be deducted if you violate the comments given in the bodies of the "//TODO" methods. Do not remove the "// TODO" comments, and please place your code *after* them.
4. **Add other methods if you want**. Make them private. Points will be deducted if methods you add do not have appropriate javadoc specifications. *We* did not see the need to add methods.
5. **Do not remove assert statements** for preconditions that we *may* have placed in some methods.
6. **About map.put**. Suppose a key value pair (`k`, `v`) is already in `map` and you want to change the value to `v1`. You do NOT have to remove the old pair (`k`, `v`) before doing that. Here is a sentence from the API spec, version 8, for HashMap.put —It helps to read the specs!!!!

   ```
   /** Associate value with key in this map. ...
     * If the map previously contained a mapping for key, the old value is replaced. */
   public V put(K key, V value)
   ```

   Points may be deducted if you remove a pair (`k`, `p1`) in order to then insert a pair (`k`, `p2`).
7. **Swap and bubble only when the class invariant is true**. Methods `swap`, `bubbleUp`, and `bubbleDown` rely on fields `size`, `b`, and `map`. *Never* swap or bubble-up or bubble-down unless: the class invariant is true except that the one value in the heap may be out of place.
8. **Elements b[size..] are unknown**. Do not use their values. (Of course, if you are inserting an element it will go in b[size]). For example, do not attempt to test that some element `b[k]` is in the tree by using `b[k] != null`. The class invariant says *nothing* about `b[size..]`, and you cannot assume that all those elements are `null`. Referencing them will cause a *big* deduction in your grade, even if your code passes all the test cases.

6. What/How to submit

1. Remove all `println` statements from class `Heap`; 3 points will be deducted if your code outputs anything.

2. Tell us how much time you spent on this assignment by replacing the value `-1` that is currently assigned to static field `timeSpent` (about line 22).

3. Near the top of `Heap.java`, in the comment, put your name(s) and netid(s) and write a few lines about what you thought about this assignment A5. 2 points will be deducted if you don't do this.

4. Submit (only) file `Heap.java` on the CMS.

7. The basic class invariant

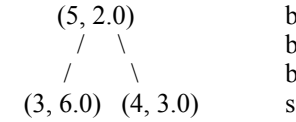
Study points 1..4 of the class invariant, beginning on about line 24 of `Heap.java`. These are essentially what you have seen already in lecture, but with both a value and its priority. Note that elements of array `b` are objects of class `Pair` —a `Pair` object contains a value and its priority.

8. A new field, map

Suppose we want to update the priority of a value `v1` in the heap to `p1`. To do that, we have to find `v1` in the heap, and this means traversing array segment `b[0..size-1]`. The average and worst case time for this is O(`size`) —i.e. linear in the size of the heap. Too expensive.

Therefore, we introduce `HashMap map`. A `HashMap` has (`key`, `value`) pairs; give it a `key` and it gives back the `value`. Moreover, `HashMap` is implemented using a technique called hashing. Using hashing, the expected or average time to find a `key` is constant! The worst-case time is linear in the size of the `HashMap`. You will learn about hashing later in the course. It's an important, invaluable data structure.

We look at an example. Below is a min-heap with 3 values. Each node contains a value and a priority (a double). To the right, we indicate what is in each element of array `b`, and we also show field size.



b[0] contains (5, 2.0)
b[1] contains (3, 6.0)
b[2] contains (4, 3.0)
size contains 3

Look carefully at points 1..4 of the invariant and check that they are true, given the fields as above.

Now, field `map` contain pairs (value, index in array `b`). To satisfy points 5..6 of the class invariant, `map` contains these pairs:

(5, 0) because value 5 is in node `b`[0] of the heap
(3, 1) because value 3 is in node `b`[1] of the heap
(4, 2) because value 4 is in node `b`[2] of the heap

Therefore, the index in array `b` of a value `v` is the value returned by the call `map.get(v)`.

Don't begin programming until you fully understand this.

9. Is it a min-heap or a max-heap?

Having a class implement either a min-heap or a max-heap might seem to complicate the class a lot. It can, if one is not careful.

Where does it matter what kind of heap it is? In bubbling up or down, and perhaps when changing the priority of a node. In bubbling up, for example, the priorities of a node and its parent have to be compared, and what is done depends on what kind of a heap it is.

We eliminate the need to check all over the place which kind of heap is being used by declaring two `compareTo` functions. The spec for one of them appears below. These functions determine what kind of heap it is —you don't have to deal with this issue at all!

```
/** If a value with priority p1 should be above a value with priority p2 in the heap, return 1.
 *  If priority p1 and priority p2 are the same, return 0.
 *  If a value with priority p1 should be below a value with priority p2 in the heap, return -1. */
public int compareTo(double p1, double p2)
```

Suppose in bubble-up, node `k` has priority `pk` and its parent `p` has priority `pp`, Then, if `compareTo(pk, pp)` returns 1, bubbling should continue, otherwise it should stop. Neat!

Points will be deducted if you use field `isMinHeap` instead of these functions anywhere in the methods you are writing.

[Take note of what we did here and use the same technique in similar situations when you are programming. If some test has to be made in lots of different places, write a function that will perform that test.]

10. Using an array for the heap

Generally, one uses an `ArrayList` for the heap so the heap can be any size. Instead, we use an array, for two reasons: (1) you get to see how `ArrayList` can change the size of its backing array when the backing array is too small. For this purpose, you will write procedure `ensureSpace`. Read its specification and make sure you implement that specification exactly. (2) The use of array notation makes the code a lot easier to read.

11. Understanding class HeapTest.java

Class `HeapTest.java` has over 30 testing procedures, with @Test before them. So many are needed to catch the kinds of errors people make in implementing class `Heap`. Class `HeapTest` also contains about 15 other methods that are used in testing. We outline how many of these testing methods work, by looking at one of them.

Look at procedure `test20Swap`, lines 305..316. The first statement,

```
Heap<Integer> mh= minHeapify(new Integer[] {5, 7, 8, 9});
```

stores in `mh` a pointer to a `Heap` object that contains this heap and map:

```
      (5, 5.0)              map entry: (5, 0)
       /     \              map entry: (7, 1)
  (7, 7.0)  (8, 8.0)        map entry: (8, 2)
     /                      map entry: (9, 3)
(9, 9.0)
```

Following that are four calls on `mh.swap`. Finally, there is this call:

```
check(new Integer[]{9, 8, 5, 7}, new double[]{9.0, 8.0, 5.0, 7.0}, mh);
```

The last argument is the heap, `mh`. The first two arguments indicate what the heap should be at this point —it should be the tree shown below, and you can also determine what the `map` entries are:

```
      (9, 9.0)              map entry: (9, 0)
       /     \              map entry: (8, 1)
  (8, 8.0)  (5, 5.0)        map entry: (5, 2)
     /                      map entry: (7, 3)
(7, 7.0)
```

Procedure `check` tests that the heap is correct, that `map` is correct, and that the heap and `map` have the same size.

Almost all the testing procedures work like this one. We summarize:

(1) Call some method to construct a heap, giving it either
an integer array that contains the ints, to be used both as values and priorities
or an integer array of values and a double array of corresponding priorities.

(2) Execute some call that changes the heap in some way.

(3) Call some method check to see that changes were made correctly.

What if running the JUnit testing class shows a red line in the JUnit pane and indicates the error is in a call like this?:

```
check(new Integer[]{9, 8, 5, 7}, new double[]{9.0, 8.0, 5.0, 7.0}, mh);
```

Do what we did above: Draw the heap and map, and then look at your code to see why it didn't change the heap/map properly. You can also put in println statements in your methods class Heap to help debug. Remove them when you are done debugging.

12. Debugging suggestions

If there is an error in your code, say in method `insert`, start with the picture as described in section 8. *A new field, map*, and hand-execute a call, like `heap.insert(9, -1.0)`.

When it is time to execute a call on `bubbleUp`, step into it, executing the method body step by step drawing the frame for the call to help you.

When it is time to execute a call on `swap`, step into it, executing the method body step by step.

If you still can't find the error, use println statements. Print enough information so you can see what is in array `b[0..size-1]` and `HashMap map`.

Eclipse does have a very good debugging tool, which we have not had time to teach you. It allows you to set "breakpoints" —places where execution will stop when reached. Then, at those places you can have Eclipse execute one statement at a time, showing what is in each variable. If you are familiar with such tools, try them out in Eclipse. We have no documentation on it to show you.