TreapMap Testing Report

1 Summary of Peer Review Process

1.1 Summary of Peer Testing Methodology

I implemented a variety of White and Black Box Tests to determine the correctness of each implementation of TreapMap. Note that this testing suite which can be found in `PeerTreapMapTest.java`, is specifically designed to go through the interface and should pass for any correct implementation of `Treap`. The specific tests are outlined below.

White Box Testing

Because the TreapMap is supposed to handle generic types, I tested the implementation on a few different Key/Value pairs that cover all cases, including a custom comparable key. Each of the below tests were conducted on these generic types.

- **insert()-** To test regular insertion, I inserted a unique set of keys into the TreapMap and checked that the BST property was satisfied (that is, everything is iterated in sorted order). If this failed, it was difficult to pinpoint the exact issue since it could be the case that the entire treap does not do rotations correctly to maintain the BST property or that the iterator does not do the in order traversal correctly. I also checked lookup(), ensuring that everything that had been inserted already could be correctly looked up in the map. More diagnostics on how I pinpointed the issue if this failed can be found in the black box testing section.

- **remove()-** To test regular removal, I removed each of the previously inserted entries from the insert() test and asserted that they were (1) found correctly, (2) correctly bubbled to the bottom of the treap, (3) and removed corectly so as to maintain the BST and Heap properties of the resulting treap. Similarly, I checked that the BST property and lookup() functionality were preserved on the updated treap.

- **split()-** To test regular splits, I split the treap across 4 different exhaustive cases and ensured that each of the subtreaps satisfied the BST property. These include: splitting on lower boundary, upper boundary, in range of values, and a specific value in the treap. I also checked whether the subtreaps were disjoint and that every key/value pair in the original treap was accounted for (i.e. was in the correct subtreap).

- **join()-** To test regular joins, I joined the subtreaps split by split() testing and also had a separate join test to help isolate any issues. Once again, I ensured the BST property was preserved and every key/value pair was accounted for.

- **Edge Cases-** Certain edge cases such as duplicate key insertion, null inputs for all functions, concurrent modification, joining a treap not of the implementing class, etc. were tested in this portion.

- **Iterator-** The iterator is mainly tested when checking that the treap maintains the BST property after every operation. I also tested a few edge cases separately to ensure it could handle calling next() too many times and modifying the treap while iterating.
- **lookup() -** This was mainly tested throughout the entire testing procedure when checking for BST property preservation. I did test special edge cases for this separately such as looking up null keys and keys that aren't present in the treap.

Black Box Testing

I used black box testing as more of a sanity check to ensure that everything I placed in the treap was actually in there. That said, there were some specific things I looked for

- I did a quick visual check of the priorities in the human readable version to make sure that the Heap property was satisfied based on priorities.
- I also did a quick visual check to ensure the BST property was satisfied

1.2 Learnings from Writing Test Code and Testing Others' Solutions

While devising a strategy to test my peers' solutions, I came up with many edge cases that were not themselves handled in my original submission. While I did handle most basic edges in my first submission, I was able to think of the trickier ones once I specifically sat down to formulate a robust testing strategy. Some of these cases that were only realized during the peer testing phase include checking for concurrent modification during iteration, joining a treap not of the implementing class, and specifically ensuring that keys equal to the "split key" end up in the right subtreap.

Furthermore, I learned a lot regarding interpretation and invalid input handling once I started testing peer solutions. For example, some teams had different interpretations how `null` input should be handled. My original implementation threw `IllegalArgumentException`s, but some of my peers' implementations simply returned without altering any of the structures. These varying interpretations gave me more to think about and highlighted some potential issues that I had not previously considered. In some cases, I even had to clarify that certain interpretations are valid.

1.3 Learnings from Received Peer Reviews

While I had already found many unhandled cases by testing my peers' submissions, there were still some good points brought up in my peer reviews. Some notable cases that were pointed out include a `NullPointerException` upon `lookup()` on an empty treap and interpretation on dealing with invalid input. A more detailed discussion of the specific edge cases and foundational issues found in the peer reviews can be found in section 2.

2 Peer Review Rankings

1. **Team 23:** Team 23 gave me the best feedback in terms of problems found and potential solutions. They captured cases such as `NullPointerException`s on empty treaps or missing

items and was the only one to clearly point out where all my invalid input mis-handling occurred. While there weren't too many issues in my submission, I still found team 23's review to be extremely helpful to know exactly what my submission handled correctly. Each case was spelled out succinctly with a result that indicated how my solution performed. I also appreciated this review's clear discussion of testing multiple different generic key/value pairs. Because this is a large part of the correctness, I found it re-assuring to know what exactly my solution could handle. This greatly reduced my debugging process for my revised solution and narrowed down potential problems area significantly. The testing methodology for Team 23 was also very robust in general and covered many of the same cases I tested for.

2. **Team 1:** Team 1 provided me with a lot of the same and interesting cases that Team 23 provided: `NullPointerException`s on empty treaps (didn't mention missing items) and some mention of incorrectly throwing exceptions when the function should simply return. However, one major flaw in their testing procedure (as described in their summary) was an over-emphasis of the generics type `<Integer, Integer>` and negligence of other key/value pair types. While the feedback provided and problems found were indeed correct, their review did not provide any meaningful insight into whether my solution correctly handles different generics. Apart from this, however, this review was helpful in validating the issues that had already been mentioned by other reviews.

3. **Team 33:** Team 33 provided me with a lot of the same and interesting cases that Team 23 and Team 1 provided: `NullPointerException`s on empty treaps and missing items and some mention of incorrectly throwing exceptions when the function should simply return. However, this review had a similar issue to that of Team 1 since there was no mention of tests on multiple different generic key/value pairs. Furthermore, I found their description of testing methodology to be somewhat vague since they simply mentioned overarching goals of certain tests. This review also mentioned that duplicate insertion failed in their testing suite, but after review, thorough testing, and consideration of other peer reviews, I came to the conclusion that my original submission correctly handled duplicate insertion. Overall, though I still found the review to be useful in validating the issues that had already been mentioned by other reviews.

4. **Team 12:** Team 12's review provided me with some good feedback on the topic of input validation, but was largely un-informative due to a limited description of testing methodology. There was a very vague mention of ”type checking” in the testing methodology which may imply that different generic key/value pair types were tested, but no detailed discussion of this was provided. Furthemore, no specifics were provided on cases that my solution handled correctly and the summary of testing methodology gave me very little insight into what was actually tested. This would be useful to know so I could narrow down problem areas and write tests that may not have been tested by my peer. In addition to this, there was a comment about joining treaps where one is not necessarily less than the other. However, it is stated in the handout and on Piazza that this can be assumed. Overall, though I still found the review to be useful in validating the issues that had already been mentioned by other reviews.

3 Revised Solution

Here, I outline a list of revisions and the insights that led to them

- **`lookup()` on an Empty Treap/Missing Item** - This was a major bug that resulted in `NullPointerExceptions` and was pointed out by all 4 teams that peer reviewed my submission. Previously, I was naively returning the value of the result of my helper method `find()`, which had the potential of being `null` in the case of an empty treap or search for an item not present. I fixed this wiith a simple check for a `null` result from `find()` and returned the correct value accordingly.
- **Invalid Input** - This was a minor bug that was a product of differing interpretations of the interface specifications. Team 23 pointed out that I was incorrectly throwing `IllegalArgumentExceptions` in certain cases where the interface specified to just return without doing anything. Some of these instances incude insertin `null` keys, removing `null`, etc. I fixed this by removing the exceptions.