# Project 6 Report

Zhaosong Zhu | CS 314H

## Assignment Goals

In the Treap project, the objectives are to implement a *TreapMap* based on the interface *Treap*, conduct peer-testing among the class, and perform a comprehensive testing on my implementations. Through my thinking on the properties of the binary search tree and a heap, I am to develop a relatively comprehensive understanding of how to implement a data structure that takes in a Key and a Value and contains various operations. My personal goals include writing out an efficient data structure that searches, deletes, adds, splits, and joins between one or more *TreapMap* quickly, reviewing the properties of both binary search tree and heap, getting familiar with rotations, and finishing bonus Karma to understand better this data structure I implemented.

## Solution Design

The design of the *Treap* project involves three aspects: *Node*, which keeps the key, value, and a randomly created priority value; *insert, remove, and lookup,* which rely on the binary search tree property at first and then restore the heap property if necessary; *split and join*, which separates a Treap into two or vice versa in reliance on the *remove* and *insert* method. To see if our TreapMap is working correctly, I dedicate a lot of time to design a test harness to test the program thoroughly.

### A. High-level details on implemented classes and methods

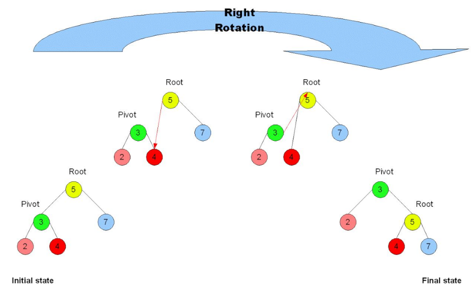
1. ***Heap Property Preservation***

   As specified above, the Treap data structure is an integration of Heap and Binary Search Tree(BST). Thus, whenever an element is found or inserted by the BST property, we then need to ensure that the Heap property is preserved.

   Two methods *leftRot(node)* and *rightRot(node)* are implemented. In these methods, a current node is passed in, and the root of the rotated tree is returned. *Figure 1* gives a graphical illustration of right rotation on the node passed in. An important question is how should we deal with the right child of the left subtree, in this case, the red node. Assuming the BST property is preserved, the right node can be put in the left part of the right subtree. This is because the red node is larger than the green node, which is why it goes to right after rotation, and smaller than the yellow node, which is why it goes to the left of the yellow node. In *figure 2*, the left rotation is performed similarly, in that the red node goes into the right part of the left subtree since it is smaller than the yellow node but larger than the green node. So, BST property is preserved during rotation.

Since this *Treap* is a maximum treap, where the maximum is on the top, the right rotation is performed whenever the current node's priority is smaller than its left child(insertion), or the priority of the right child is larger than the left child(removal). Similarly, the left rotation is performed when the current node's priority is smaller than its right child(insertion), or the priority of the right child is smaller than the left child(removal).

The justification behind is that under insertion we want to rotate the current node down if its priority is smaller than the children, and under removal, the goal is to try to rotate the current node down without breaking the BST and heap property.



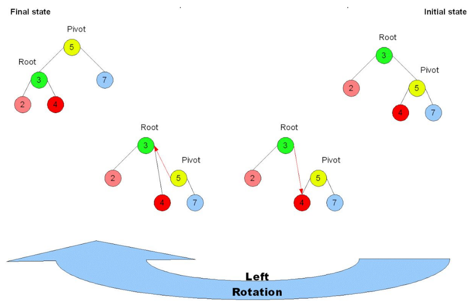


Figure 1 Figure 2

2. ***Node design***

In my *TreapMap* class implementation, a node class is provided inside. The node class provides two kinds of functionalities.

The node stores value: according to the generic type key K and value V, a *K key* and *V value* are provided to store the user's input. Besides, using the MAX_PRIORITY given by the interface, an integer of priority value will be generated in the range of [0, MAX_PRIORITY). I didn't allow the priority to be MAX_PRIORITY because, in the split, or testing, I might insert a node with MAX_PRIORITY so that it can stand to the top.

The node also provides two references: Node *left* and Node *right*. These two references are *null* in the beginning without initialization. With these two references, we can build up the entire tree without using arrays.

This class provides a constructor which takes in a K and a V. Besides, another private method is provided to set the priority value, in the case of split and join.

3. ***Split and Join method design***

As specified in the assignment instruction, the split method heavily relies on insert and the join method relies on remove.

When splitting an item, the program will invoke the internal insert method while setting the newly inserted item with MAX_PRIORITY to ensure that the node inserted will be the new root. During the insertion, the key will be the parameter passed in while the value is not a big deal. Since all nodes with keys larger than or equal to the parameter need to

be on the right side of the Treap, a lookup is used to keep the value. Right before the Treap is divided by left and right subtree, I'll check if the possibly lost item should be inserted to left or right subtree. The two sub - Treaps will then be returned.
Joining two treaps follows the similar idea. Under the assumption that all keys in T1 are smaller than those in T2, the program will create another treap, who has a random root with T1 being the left subtree and T2 being the right subtree. Since the key can be other than an integer, I set the new root with *key = root.key* and *value = root.value*. Although the key and value are equal to those of the root of the left subtree, during removal, this situation will be handled with a rotation, which will successfully rotate the node down.

## B. Abstractions

1. ***Recursion***

   Recursion is frequently used throughout the project. The most obvious examples will be *lookup, insert, remove*, each of which has their internal method that can take not only key and value but also a node in the parameter. In doing so, I can recursively call on codes to check if the left or right subpart of the current node is valid and points the result back to the corresponding subparts.

   Besides the three "backbone" methods above, recursion is also used in iterator traversal and finding the balance factor. Since the iterator uses inorder traversal, the recursive method will try to go to the leftmost node by calling itself with *currentNode.left*. Then, as the stack pops out all the calls, a recursive call on whether the right side is null is used to add all nodes in an ordered fashion into an ArrayList. In the balance factor method, it finds minimum and maximum depth with two recursive methods too – the minimum method recur on the right subtree if the left is null and vice versa, with the edge case of null node, whose height =0; the maximum method will just travel down the tree as much as possible.

2. ***Tree traversal***

   Throughout the project, three types of traversals are used: BST traversal, inorder traversal, and preorder traversal.

   In *lookup, insert, remove*, we find the location of an item using BST traversal, which relies on the property that items smaller than the current node go left and larger go right. Starting from the root, a comparison is made between the key of the current node and the targeted key. If the targeted key is smaller than the current key, we go left and vice versa.

   Inorder traversal is used for the iterator. This is because inorder traversal satisfies the requirement that the first item added to a list will be the smallest while the last item is the biggest. As explained in the recursion section, inorder traversal tries to go to the leftmost node at the beginning. Through this process, the 1st node added will be the leftmost node, and as the recursive call stack pops out nodes from the bottom part of the tree to top, we then add the parent node, then the right node. This process continues until the entire tree is covered.

I used preorder traversal for the toString method, which prints out a human-readable treap with hierarchy tabs. As defined by the interface, we need to use preorder traversal, which first prints out the root node, and keep doing so in left subtree then the right subtree. In doing so, the everything of the left part of the tree is printed out first and then the right part of the entire tree.

3. ***Linked list to connect the nodes***
   To build up the entire tree, references are used to connect each node. As specified in the Node design, each node contains two references – *left* and *right*. On those references, another node can be created and so can we on the newly created node. As this process goes on, the entire tree is connected by references. The advantage is that when removing an item, we will only need to change the references instead of manipulating data in a List. However, the downside is that we will not be able to access an item by indexing, which somewhat slows down the retrieving process compared to a List.

# Project Discussions

## A. *TreapMap* Designs, Assumptions, Strength, and Weakness

As I write up codes for *Treap* class, one important consideration, as mentioned in the instructions, is to perform the search in *O(log n)* time.

- *Assumptions*

1. ***Randomized Priority Value***
   The first assumption is that the priority value generated is randomized when generated by *Math.Random()*. This assumption is very important because first, randomization keeps a treap balanced, and second, it also prevents a huge amount of rotation due to extremely large value.

   Assume the random generator gives the same priority value all the time. In this case, if we keep inserting keys that are smaller than the last key inserted, a slender node chain will be formed instead of a somewhat balanced tree. This is because, without the difference of priority value, no rotation can happen during insertion and thus no change in the tree structure.

   Another possible case will be that each time a newly inserted node is given a priority value larger than any one of them in the current Treap. In this case, O(h) times of rotations need to happen so that this node can be rotated to the top of the treap. Although rotations by themselves are O (1), with randomization the rotation will not happen as often.

2. ***$T_1$. join($T_2$)***

   The assumption made for the two treaps $T_1$ and $T_2$ are that all keys in $T_1$ are smaller than those in $T_2$. This premise is the key to a correct output – a Treap that contains all key and values. In this case, the deletion of the root will result in a rotation of node from either left or right subtree. Knowing that BST property is preserved when looking at both subtrees, we will be able to move a root node from either side without breaking BST and heap property. However, if this assumption is not the case, then although all keys and values are probably still preserved, the new treap will not be good to use.

- *Class design highlights*

1. ***Internal insert and remove method***

   *BSTInsert* and *BSTRemove* are two helper methods for *insert* and *remove* method.

   The BSTInsert method takes in three parameters: node, key, value. Inside this recursive method, the base case is when the current node is null, which implies either the root or a reference is empty, and we can insert the value here. Other than this case, three more possibilities happen during the search:

   a. Current key > targeted key – in this case, we will continue finding the key recursively on the left subpart of the node; in case the current node's priority value is smaller than that of its left child, a right rotation happens on the current node;
   b. Current key = targeted key – in this case, we substitute the currently stored value with targeted value;
   c. Current key < target key – in this case, we will continue finding the key recursively on the right subpart of the node; in case the current node's priority value is smaller than that of its right child, a left rotation happens on the current node;

   Since the rotation check if happened right after the recursive method, each time when an item is returned by order of the stack, a rotation will be performed when necessary. Thus, all nodes will be rotated if necessary in a bottom-up order.

   The BSTRemove method takes in two parameters: node and key. Inside this recursive method, there are as well multiple cases considered when finding and removing the node:

   a. Current key > targeted key – the remove method begins to check the left child of the current node recursively
   b. Current key < targeted key – the remove method begins to check the right child of the current node recursively
   c. Current key = targeted key

i. If the current node only has one child, just delete the node and connect the only child back to its reference
ii. Left.priorityValue < Right.priorityValue – rotate the right node up as the current node and continue to remove from the right child of the current node, which is the targeted node
iii. Left.priorityValue > Right.priorityValue – rotate the left node up as the current node and continue to remove from the left child of the current node, which is the targeted node

Finally, this method returns the modified tree's root, as the passed in value if the root of the tree.

2. **Efficiency for *lookup, insert, remove, split, join***

   As specified in the assignment guidelines, all the operations should take O(h) expected, which is O(log *n*) if the expected height is Θ(log n). Below is a justification of the time bound for each method.

   a. Lookup
      Essentially, a lookup operation in Treap is the same as in a regular BST. Whenever a search performed, it takes O(1) to decide which child should we go to. Since each time the search is split in half, in total it costs O(log *n*) expected time to find the value of an element.
   b. Insert
      Insertion takes O(log n) time as well. Since the first part of the insertion take focuses on finding a spot/ the key inside the tree, which is the same as lookup process, this part takes O(log n) time. Having found the element, the program will start to rotate nodes, which is a method that only changes references. This part takes constant time. Thus in total insertion takes O(log n) time.
   c. Remove
      Similar to insert, it takes O(log n) time for the algorithm to locate the element. As soon as the element is located, a rotation may happen as outlined above. Thus, the remove method may still be called for multiple times. However, since all of them takes O(log n) time, the sum of them is still O(log n), ignoring the constant.
   d. Split and Join
      Since split and join only performs remove or insert after linking some nodes to others, an O(1) procedure and O(log n) procedure should take O(log n) time to accomplish everything.

- ## *Weakness and possible improvements*

  My implementation of insert, lookup, and remove relies heavily on recursion. As we know, recursion is hard to trace and find errors, so we need to have faith in its correctness. However, I did make a small mistake when implementing the remove class, which leads to null pointer error and it took me some time to trace the logic of the method. Besides, $T_1$. *Join*($T_2$) assumes that all keys in T1 must be smaller than T2. Although this assumption is crucial, it will be good for us to have a quick algorithm that can check for any inconsistency and inform the user.

Finally, the rotations I perform didn't check if each node connected are legal to rotate. This means I assume the BST property is already established before then. However, in any case, that the BST property is broken, the rotations may just blindly continue and mess up the entire treap.

## B. Edge cases

This is a list of edge cases that I found during testing phase:

**LookUp**

1. Null parameter
2. Find Item - empty treap
3. Find Item - Multiple Add in

**Insert**

1. Null K/V or K+V

**Remove**

1. Null K - give error or maintain the same treap
2. Can I still find the item?
3. BST property? - rotation
4. Behavior on empty treap

**toString**

1. Empty Treap

**Iterator**

1. Empty Treap
2. K matches with add-ins
3. # of Items
4. Check per-order property?/ BST

**Split**

1. Null parameter
2. Behavior on empty treap - should return two empty treaps
3. Normal Split testing
4. Give parameter the min/ max value of input
5. value < min
6. value > max

**Join**

1. Null parameter
2. Is everything in a after join?
3. a.join(b) - a empty
4. a.join(b) - b empty
5. if change tree b, will it affect a? - remove or insert

## D. Karma & Personal Reflections

1. Karma discussion
    a. Balance statistics

    As specified in the assignment, the balance statistics is a ratio, namely $\frac{treap\ height}{minimum\ path\ length}$.
    Two recursive methods, *minimumDepth* and *maxDepth,* are used to determine each one of the factor. The minimum method recurs on the right subtree if the left is null and vice versa, with the edge case being encountering a null node, whose height =0; the maximum method will calculate a left and right height at the current node. To obtain the maximum, we will compare the length of the path on left and right the side. Whichever is longer, we will return that one.

    However, there is an obvious edge case – what is the balance statistics for a treap when its empty? Since 0/0 is not allowed, we will manually check if the bottom is 0. Since this indicates that the entire treap must be empty, we will return 1.0 instead to show that the treap is balanced in that it's just empty.

    To see how well the treap balance itself with the random heap property, I performed ten tests adding different number of K, V pair and see the balance factor. Table 1 shows the results:

| # of items added | Balance statistics |
|---|---|
| 10 | 1 |
| 10 | 2 |
| 10 | 1 |
| 50 | 1 |
| 50 | 2 |
| 100 | 2 |
| 100 | 3 |
| 500 | 3 |
| 500 | 3 |

    b. Problem diagnoses

    Based on my experience of peer testing and testing my own program, I find out certain errors given may help us diagnose errors. Below are heuristics that I discovered during testing and can be used with my testing harness to help diagnose.

A. Integrated test
    1. If a NullPointerException is thrown during any assert for split - consider checking if any value should be in the tree is lost
    2. If errors happens after the normal split test, check your algorithm to see if you have handled the special cases listed in the comments
    3. If a NullPointerException or any other error given during join two normal Treaps, please test your remove method; also ensure that a Treap with T1 being left and T2 being right with a random node is constructed correctly.
    4. If error happens when testing with either one empty Treap, check your join method to see if you have handled these two cases → just do a check at the beginning and change references to root

B. Lookup and insert
    1. You should have seen three System.err messages in the console, which signal that you have handled the null parameter correctly

C. Remove
    1. If an error happens when removing elements, consider if all elements are successfully added? Also ensure that your remove method has actually shifted the node to the bottom and set to null?
    2. If an System.err message is not printed, you might consider handle *remove(null)* case; null should not modify any elements in the Treap

D. toString and iterator
    1. If the toString is incorrect, check if you preorder traversal implement is correct; you might have mistaken the order of taking left and right node
    2. If the iterator is not giving correct elements, check if the iterator will be refreshed when a new one is created on a modified Treap.

Besides the heuristics above, I have also added some meaningful System.err print messages throughout the test. The user can use those printouts as tips to diagnose the program.

2. What I learned from this project and future improvements

For the one thing, this project allows us to apply the concepts of Treap(aka BST + Heap). By implementing the entire Treap, I can develop a relatively deep understanding of both the BST property and Heap property. For example, when inserting the key "5", I will start from the tree node to determine whether I should go left or right. As soon as the key is, I will need to perform rotations to ensure the heap property. In doing so, I was able to think about how the rotation should happen with left or right rotation.

The peer testing process has also helped me greatly. For example, when thinking about testing cases for others, I was able to fix my bug that during split sometimes an element on the right accidentally goes to left. Many thanks to the feedbacks from other groups, I was also able to fix some other corner cases, including the inconsistence of throwing NullPointerException with null as a parameter. I wish to do more peer testing in the future!

# Testing

## A. Testing Methodology

The JUnit testing for this project includes one integration test and three smaller unit test: *Lookup + Insert, Lookup + Remove, toString + Iterator*. Detailed explanations are shown below:

### **<u>Integration Test:</u>**

This test intends to test the interactions between multiple with *split* and *join* method.

As specified below in the table, I have performed tests in eight different aspects.

Starting from line 45, the test's focus is on if correct results can be returned after split and join are performed on traps with different conditions:

- Split an empty treap;
    - EXPECTED: return two empty treaps
- Split a normal treap with a given key;
    - EXPECTED: find all K < key on the first treap and everything else in the second treap
- Split a treap using max inserted value;
    - EXPECTED: find only the $key_{max}$ on the second treap and everything else in the first one
- Split a treap using key> max as inserted value;
    - EXPECTED: find everything in the first treap
- Split a treap using min/ key< min as inserted value;
    - EXPECTED: *contains* and *isPrefix* return *FALSE*
- Join two treaps a.join(b) – is all K,V are in a;
    - EXPECTED: *YES*
- Join two treaps a.join(b) - if modify treap b, will it affect K,V in a?
    - EXPECTED: NO
- Join a treap with an empty treap;
    - EXPECTED: find all element in Treap a

| Testing item | Aspect covered |
|---|---|
| Split a normal treap with a given key; | Split () |
| Split an empty treap; | Split () edge cases |
| Split a treap using max inserted value | |

| | |
|---|---|
| Split a treap using key> max as inserted value | |
| Split a treap using min/ key< min as inserted value | |
| Join two treaps a.join(b) – is all K,V are in a | Join () |
| Join two treaps a.join(b) - if modify treap b, will it affect K,V in a? | Join () edge cases |
| Join a treap with an empty treap | |

**Implementation:**
I used JUnit 4 for running all our unit tests. For the most part, I used the assertEquals and assertArrayEquals methods to check if the actual values matched with the expected values. For loops are also used when iterate through all the K-V sets I want to assert. Following is a list of the things I tested and their line numbers:

| Testing item | Code lines |
|---|---|
| Split a normal treap with a given key; | 45-71 |
| Split an empty treap; | 38-45 |
| Split a treap using max inserted value | 74-95 |
| Split a treap using key> max as inserted value | 147-164 |
| Split a treap using min/ key< min as inserted value | 97-145 |
| Join two treaps a.join(b) – is all K,V are in a | 183-186 |
| Join two treaps a.join(b) - if modify treap b, will it affect K,V in a? | 201-208 |
| Join a treap with an empty treap | 190-199 |

**Negative tests:**
I used a couple of assert(false, a==b) statements to make sure that the test harness detected that I were passing in the wrong inputs and that it wasn't just passing all the cases.

| Testing item | Code lines |
|---|---|
| Split an empty treap; | 38-45 |

| | |
|---|---|
| Split a treap using max inserted value | 74-95 |
| Split a treap using key> max as inserted value | 147-164 |
| Split a treap using min/ key< min as inserted value | 97-145 |
| Join two treaps a.join(b) - if modify treap b, will it affect K,V in a? | 201-208 |
| Join a treap with an empty treap | 190-199 |

## Lookup + Insert Test:

This test intends to test the insert method and lookup method at the same time.

Starting from line 214, the test's focus is on if correct results can be returned after a K-V is inserted and various edge cases with lookup and insert:

- Look up in an empty treap;
  - EXPECTED: return null for all key
- Insert with null parameters;
  - EXPECTED: throw NullPointerException
- Retrieve a value after insertion with K-V;
  - EXPECTED: return the correct value
- Retrieve a value after multiple insertions with same K;
  - EXPECTED: return the final inserted value
- Look up with null parameter;
  - EXPECTED: return null

The negative edge cases I tested include:

   a. Null parameter
   b. Insert multiple time with the same K but different V

Lookup is the only case where I should not throw a NullPointerException with a null parameter.

## Remove Test:

This test intends to test the remove method.

Starting from line 253, the test's focus is on if correct results can be returned after a K-V is inserted and various edge cases with lookup and insert:

- Remove in an empty treap;
  - EXPECTED: return null for all key
- Does it return correct Values when removing keys;

    - EXPECTED: YES
- Find the same set of keys after removing;
    - EXPECTED: NO
- Null parameter;
    - EXPECTED: throw NullPointerException

The negative edge cases I tested include:

a. Null parameter
b. Catch a NullPointerException and see if I can still find added items
c. Remove in an empty treap

## <u>ToString and Iterator Test:</u>

This test intends to test the toString and Iterator.

Starting from line 311, the test's focus is on if the toString method can print correctly with tabs and if the iterator can use iterator through all elements in ascending order:

- ToString on empty treap;
    - EXPECTED: return empty string
- Obtain an iterator from an empty treap;
    - EXPECTED: iterator.hasNext() will be false
- ToString format;
    - EXPECTED: [priority] <key, value>\n
- Iterator from a non-empty treap;
    - EXPECTED: should give back all keys in ascending order

The negative edge cases I tested include:

a. ToString on empty treap;
b. Obtain an iterator from an empty treap;

## B. Black Box Testing:

While I was coding the assignment, I inputted random values into the TreapMap and tried to call on the methods. Black-box testing was also very helpful in dissecting a bigger problem into smaller parts. For example, when my join method was not working, I instead went back to give some values to the remove method to see if it performs well under different inputs. This helps me turn my attention to remove, which is the backbone of join method.

| |
|---|
| Find Item - 1st add in |
| Find Item - Multiple Add in |
| Insert a value and find it |
| Remove a value and try to find it again |

| |
|---|
| Print out toString result – on split/ joined treaps as well |
| Print out items from iterator |

In addition to calling the methods, I used System.Out.Print statements in several places for debugging the code.

## C. White box testing:

As explained above, I created one integration test and three smaller unit test: *Lookup + Insert, Lookup + Remove, toString + Iterator.*

## D. Testing Exceptions:

A NullPointerException is thrown when a null parameter is passed in for insert, remove, join, and split.

When testing the null parameter for a different method, I will catch the exception and print out a specific message showing that a correct exception is thrown. By looking at the console, I can see if the program has handled the exception thrown correctly.

## E. Strengths and Weaknesses in our Testing Methodology

**Strengths:**

Testing the code increased my confidence in its correctness. Since I used JUnit, I was able to automate the testing process. Instead of having the program print diagnostic statements and examining those statements to make inferences about the correctness of our program, I was able to have the JUnit test method check for correctness automatically. Having an automatic checking mechanism shortened the time required for testing individual cases considerably.

**Weaknesses:**

I came up with a large number of test cases and tried to test as many cases as I could think of, but couldn’t code all of them due to time constraints. However, the peer testing process helps open my mind to creating a more comprehensive testing harness.