Program 6 - TreapMap

1 Overview and Goals

1.1 Overview

The intent of this project is to implement the logic and interface for the TreapMap, a map based on the foundational concept of a randomized Balanced Binary Tree with the Heap property. This assignment allowed me to explore recursive alorgithms in great detail, explore the concept of generics in Java, and make key design decisions on data storage and computation.

Goals

My goals for this project included implementing a robust set of algorithms that efficiently maintain the BST and heap properties of the treap for all supported operations and functioned as a map on all possible key/value pair types. Additionally, I aimed to write a robust testing suite that would catch any and all errors in my implementation and narrow down the problem areas. I also focused some of my time on implementing extra functionalities such as `balanceFactor()` and `meld()`.

2 Description

2.1 Solution Design

The essential modules in this project is `TreapMap.java` which houses the `TreapMap`, `TreapNode`, and `TreapMapIterator` classes. As is implied from their names, the `TreapMap` class implements all operation functionality logic and maintains the internal structure of the treap, the `TreapNode` class stores and maintains the internal structure of eahc individual node that comrpises the treap, while the `TreapMapIterator` class implements the logic to efficiently iterate over the treap in an in order fashion. I outline overarching design and data structures for each of the components below.

2.1.1 `TreapMap`

The crux of this project is efficiently modifying and storing the treap structure while maintaining correctness of the Binary Search Tree and Max-Heap Properties. By doing so, the treap is balanced on average due to the randomness introduced in the form of priorities. Because the underlying structure of the treap is that of a Binary Search Tree, I chose to use the canonical representation of `TreapNode` objects to allow efficient traversal, rotations, etc. The `TreapMap` class stores a reference to the current root of the treap which is updated according to the various operations performed.

An important component of this module is the use of generics. Because `TreapMap` should behave as a standard map, any key/value pairs types should be allowed (with the added restriction that the key must be Comparable). Thus, type checking was enforced throughout the implementation and special consideration was taken to ensure correctness of item comparison (use of `compareTo()`).

2.1.2 `TreapNode`

As mentioned above, the `TreapNode` class implements the data representation for nodes in the treap. Each `TreapNode` has references to its two children, and stores its `key`, `value`, and randomized `priority`. Because each node is represented by an object with pointers to its children, rotations can be performed in $O(1)$ time.

Generics are also an important consideration here where the key and value are specified as generic types `K` and `V`.

2.1.3 `TreapMapIterator`

The `TreapMapIterator` class implements iterator functionaity for a regular BST using stacks to achieve an $O(1)$ amortized iteration. A more thorough explanation of this can be found in section 3.4. Once again, generics are emphasized to ensure that the iterator traverses over type `K` of all keys in the treap.

2.2 Assumptions

Below are some of the assumptions that are made in the implementation of my solution

- **Disjoint and Strict Key Magnitude Treaps in `join()`** - It is assumed that the two treaps provided to `join()` are disjoint and that one treap has keys that are strictly less than the other.
- **TreapMap Instantiation Without Generics** - It is assumed that the user does not instantiate a `TreapMap` without the use of generics. This would allow insertion and storage of different key/value types within the same treap, resulting in `ClassCastExceptions`. I do handle these cases, by type checking the input.
- **Reasonable Input Size** - It is assumed that the user does not enter extreme input sizes. As discussed in section 3.1, this could potentially result in `OutOfMemoryErrors`, `StackOverflowError`, etc.

3 Discussion

3.1 Scope and Quality of Solution

While my implementation of the `TreapMap` covers many edge cases and invalid inputs, it still has some notable limitations

- **Memory Limits** - An important limitation of my implementation is the reliance on memory to (a) store the nodes and (b) perform and handle recursive calls for operations. Thus, large inputs are not ideal for the vast amount of memory storing Node objects takes. Furthermore the upper bound for static storage is greatly reduced due to the dynamic space needed for operations on large treaps which may involve large recursive calls, and potential OutOfMemoryErrors or overflow of the recursive stack.

- **Guarantees on Balance Statistics** - Unlike Red Black Trees or AVL Trees, Treaps do not have a rigid set of rules that guarantee tree height on the order of $O(log(n))$. Because the implementation of the treap is heavily dependent on randomization, it may be the case that the tree degenerates to a linked list. While this is highly unlikely for sufficiently large input, the probability is non-zero. This results in inefficient operations and a loss of the Balanced Binary Search Tree property.
- `difference()` - the `difference()` function is not implemented, so the implementation of the `TreapMap` is considered incomplete.

There are tangible and finite cases under which my implementation will fail as noted by the above limitations. That being said, my solution is designed to minimize the time and space complexity of required operations in an effort to limit the resources required to maintain and modify the provided data. Generics are also used in an effort to make the implementation of `TreapMap` more universally viable. Key design choices regarding choice of data structure, algorithmic approaches, generics, and exception handling are essential to my solution.

3.2 `TreapMap`

Here, I describe each operation in detail, including discussion of over-arching algorithm and data structures used.

3.2.1 Tree Rotations

A major component of treap operations such as `insert()` and `remove()` is tree rotations. Here, I describe the overarching logic of a left rotation (analagous logic can be applied to right rotations). Consider my implementation of `leftRotate()` which takes in `TreapNode pivot` to rotate left around and returns the new root `TreapNode`.

1. A new `TreapNode newRoot` is assigned to the right child of `pivot`. This intuitively makes sense since a left rotation around `pivot` would make its right child the new root of the treap. This step simply initializes the new root to point at the correct node, setting up the next few pointer switches.
2. The right child of `pivot` is assigned to the left child of `newRoot`. This completes the child transfer step that must occur to ensure `newRoot` will have only 2 children. The BST property is maintained here.
3. The left child of `newRoot` is assigned to `pivot`. Once again this makes intuitive sense from the left rotation concept.
4. `newRoot` is returned.

As can be seen, these rotations maintain the BST property of the treap, while providing an efficient method to correct for the max heap property. Since the only operations involved are pointer re-allocations, we can conclude that the tree rotation can be performed in $O(1)$ time.

3.2.2 `insert()`

I took a recursive approach to the `insert()` function and therefore wrote a recursive method `insertRec()` that was called from the driver function `insert()`. The recursive algorithm is described in detail below

- **Parameters:** The following parameters are passed to the recursive function and completely describe a unique recursive state
  - `TreapNode toInsert` that is the `TreapNode` to be inserted into the treap. This does not change over recursive calls.
  - `root` is the root of the current subtreap we are located at.
- **Return Type:** The recursive implementation returns a `TreapNode` that is the new root of subtreap at that position.
- **Base Case:** If `root` is `null`, we return `toInsert` as the new root of the subtreap at the leaf position (A special consideration base case is made for the case where `toInsert` is `null`. In this case, nothing is to be added to the treap, so the original root is retuned).
- **Recursive Step on the Way Down:** This is a standard BST insertion step where we determine whether to move right or left at the current state based on how `toInsert` compares with the current `root`. We set the respective child of the current `root` to be the result of `insert()`on the subtreap rooted at that child. Once we have trickled all the way down a unique path on the tree, this completes the step of finding the leaf position to insert our node.
- **Recursive Step on the Call Stack Back Up:** Here, the insertion algorithm differs from the standard BST insertion. Because our inserted node may have a high priority and we want to maintain the max-heap property to decrease the balance factor of our treap, we must perform rotations on the way up to correctly place `toInsert` while maintaining both the BST and max-heap properties. We do this by checking which child (if any) of the current root has a higher priority than itself (note that both children can never have higher priorities than the current root since we start off with a legal treap). The current root is then updated to the result of the respective rotation around the old root and is returned.

In essence, the `insert()`algorithm I implemented travels down and back up a given unique path, to first find the leaf location of insertion in accordance with the BST properety, then performs rotations on the way back up in accordance with the max heap property. Therefore, the time complexity of this solution is $O(h)$ where h is the height of the treap. It can also be shown that the time complexity is $O(log(n))$ in the average case due to randomness where n is the number of nodes in the treap.

3.2.3 `remove()`

Similar to the `insert()`, I took a recursive approach to the `remove()` function and therefore wrote a recursive method `removeRec()` that was called from the driver function `remove()`. The recursive algorithm is described in detail below

- **Parameters:** The following parameters are passed to the recursive function and completely describe a unique recursive state
  - `TreapNode K key` that is the key to be removed.
  - `root` is the root of the current subtreap we are located at.
- **Return Type:** The recursive implementation returns a `TreapNode` that is the new root of subtreap at that position.
- **Base Case:** If `root` is `null`, we return `null` as the new root of the subtreap since the treap was originally empty, and nothing can be removed.
- **Recursive Step to Find the Node to Remove:** This is a standard BST removal step where we determine whether to move right or left at the current state based on how `key` compares with the key of the current `root`. We set the respective child of the current `root` to be the result of `insert()`on the subtreap rooted at that child. This step ends once we reach a node with key equal to the key that must be removed (if the key is not present in the treap, this step may not end until we reach a leaf, in which case null is returned).
- **Recursive Step to Remove:** Here, the removal algorithm differs from the standard BST removal. We consider 3 exhaustive cases that outline the possible types of node the be removed
  1. **Node to be Removed has no children:** In this case, we can simply return `null` to be the new root of this subtreap. This essentially snips off the node to be removed.
  2. **Node to be Removed has One Child:** In this case, we return the child to be the new root of this subtreap. This essentially splices out the node to be removed.
  3. **Node to Be Removed has Two Children:** In this case, we must check which child has the higher priority and assign the new root of this subtreap to be the result of the appropriate rotation (making the higher priority child the new root). We then recurse again on the tree to remove the specified key since it has now be rotated down.

In essence, the `remove()`algorithm I implemented travels down the treap once, and separates the recursive steps in 2 to first find the node to be removed, then perform the removal via appropriate rotations to maintain the BST and heap properties. Therefore, the time complexity of this solution is $O(h)$ where h is the height of the treap. It can also be shown that the time complexity is $O(log(n))$ in the average case due to randomness where n is the number of nodes in the treap.

3.2.4 `lookup()`

The `lookup()` function is implemented recursively using a standard BST search algorithm that travels down a unique path on the treap and returns the found node. The recursive details of this are very similar to the basic BST search outlined as a part of the `remove()` discussion, so they will be ommitted here. Therefore, the time complexity of this solution is $O(h)$ where h is the height of the treap. It can also be shown that the time complexity is $O(log(n))$ in the average case due to randomness where n is the number of nodes in the treap.

3.2.5 `split()`

The `split()` algorithm follows a basic concept to split a treap into two halves where one has keys less than a specified key, and the other has keys greater than the same specified key. A new node with `MAX_PRIORITY` and specified key is inserted into the treap. By the max heap property of the treap enforced during insertion, this node will necessarily become the new roof the treap. Furthermoree, due to the BST property that the `insert()` enforces, the entire left subtreap must have keys less than the new root, while the entire right subtreap must have keys greater than the key of the new root. Because this operation just performs one insertion and multiple constant time operations such as instantiating the new node, the time complexity of this solution is $O(h)$ where h is the height of the treap. It can also be shown that the time complexity is $O(log(n))$ in the average case due to randomness where n is the number of nodes in the treap.

A key design decision made here included specifying the new "max node" 's value to be `null` since this is the only universal instance of type `V`.

3.2.6 `join()`

The `join()` algorithm follows a basic concept to join two treaps with the necessary assumptions. A new node with key and value of `null` is explicitly made the root by setting its children to the roots of the treaps to be joined. Here the children are assigned sides based on how their keys compare to the root. Note that here, we only need to check the comparison of the roots of each subtreap since we are guaranteed by assumption that all the keys in one of the treaps is larger than all those in the other. The arbitrary `null` node is now removed from the treap (directly calling the recursive remove function so that `null` key removal is acceptable). By the rules enforced during `remove()`, the BST and heap properties will be maintained in the resulting treap with all nodes. Because this operation just performs one removal and multiple constant time operations such as instantiating the new node, is $O(h)$ where h is the height of the new joined treap. It can also be shown that the time complexity is $O(log(n+m))$ in the average case due to randomness where n and m are the sizes of the two treaps.

3.2.7 Generics

As mentioned previously, generics are a big part of the final implementation of `TreapMap`. To make the solution universal to all key/value pair types, I make sure to use universal comparisons using `compareTo` on keys and check for any type mismatches.

3.2.8 Invalid Input

Invalid input is handled very similarly for all operations. `null` inputs are handled using either an `IllegalArgumentException` or simply returning without doing anything. Parameters of the incorrect types result in an `IllegalArgumentException` (this is a special case where the `TreapMap` is instantiated without generics).

3.2.9 Space Complexity

The main storage of data for this implementation is the `TreapNode` instances and their respective pointers. This scales linearly with the number of nodes present in the treap, so the space complexity is $O(n)$ where n is the number of nodes in the treap.

3.3 `TreapNode`

As mentioned in section 2.1.2, the `TreapNode` class houses the representation of data nodes that (along with the respective pointers) comprise the treap. An important design decision I made here had to do with choosing the best way to represent connections between nodes so as to maximize efficiency. The two main options for the overarching representation were a pointer based connection or an array based representation where the children for a given node in position i can be found in positions $2i + 1$ and $2i + 2$. While it is true that the array based representation has a slight edge in terms of space complexity since no explicit pointers need to be stored, it loses out greatly in terms of time complexity of operations. To see why this is true, consider the rotation operation that is required for both the `insert()` and `remove()` functions. This can be performed in $O(1)$ time with a pointer based representation while the array based representation requires $O(n)$ time asymptotically due to the shifting and manipulation of nodes in the array. Due to this, I decided to go with the pointer based representation.

3.3.1 Data Fields

The data fields each `TreapNode` stores is listed below

- `TreapNode left` - The left child of this `TreapNode`. `null` if it has no left child.
- `TreapNode right` - The right child of this `TreapNode`. `null` if it has no right child.
- `K key`- The key associated with this `TreapNode`
- `V` The value associated with this `TreapNode`
- `int priority` - The priority of this `TreapNode`

A key design design made here has to do with choosing which pointers each `TreapNode` should store. In some initial versions of my solution, I considered storing a pointer to `parent` and an additional `enum CHILD_TYPE` (along with `left` and `right`) that indiciated whether this `TreapNode` was a left or right child. This was mainly due to my naive implementation of the `TreapMapIterator` where the successor was found iteratively at each call of `next()`. This posed significant challenges since it not only added extra space (still $O(1)$ for pointers asymptotically but in the short run, it does take up more memory), but would also require updates of all of these fields after all operations and rotations. A more thorough description of this initial solution can be found in section 3.4.

In the end, however, a more efficient implementation of the iterator allowed me to maintain correctness with pointers only to `left` and `right`.

3.3.2 Generics

As mentioned in section 1.2.1, generics were also a significant component of the `TreapNode` class since the node structure is the most basic representation of the data storage. Since all types must be accomodated for, the `TreapNode` class makes use of generics `<K,V>`. Most of the type checking, however, occured in `TreapMap`, so it is guaranteed that the correct types are used in `TreapNode`.

3.4 `TreapMapIterator`

3.4.1 Algorithm

I went through a few different algorithms to implement the iterator logic before deciding on the best one. My initial solution was to pre-process an in-order traversal and return the items from the pre-processed cache. This, however, ends up being a simple wrapper over another iterator, so it is not a viable solution. The next solution I considered included storing the current node and recursively find its successor in `next()` and `hasNext()`. This resulted in an $O(logn)$ time complexity on average per operation where n is the number of nodes inthe treap, and so was not a viable solution for the iterator operations which should do operations in $O(1)$ time.

The algorithm I finally decided on that satisfied the $O(1)$ operation time and was not a simple wrapper over another iterator made use of a global stack to essentially do a discrete in order traversal. The steps of the algorithm are outlined below.

- Initial Push: `pushLeft()` is called on the root node
- `pushLeft()`: This function iteratively pushes the entire left ancestry of the passed node into the global stack. This ensures the furthest left child of the subtreap is visited before any others.
- `next()`: This function pops off the topmost node of the global stack and saves it as `ret` since this is the node to return. `pushLeft()` is called on the right child of `ret`. This preserves the in order fashion on the right subtreap and maintains the "left-curr-right" ordering.
- `hasNext()`: This function returns a boolean indicating whether there is a next element to iterate based on whether the stack is empty or not.

The above algorithm essentially simulates the recursive in order traversal by working bottom up and left to right on any given subtreap. This algorithm has an $O(1)$ amortized time complexity per operation. To easily see this, note that each node is pushed and popped from the stack exactly once. The `push()` and `pop()` operations are $O(1)$, therefore there are a total of $2n$ constant time operations (where n is the number of nodees in the treap). This results in an $O(n)$ complexity for the complete traversal and an $\frac{O(n)}{n} = O(1)$ amortized complexity per operation.

3.4.2 Concurrent Modification

One key design decision made here had to do with tracking concurrent modification of the treap while iterating. I decided to keep a counter called `modCount` in the `TreapMap` instance that would increment everytime an operation is attempted. My `Iterator` instance stores this specific `TreapMap`

instance and stores the initial state of the modifier count. Then, the current `modCount` of the `TreapMap` instance is checked at the beginning of every call to `next()` or `hasNext()`. If it is different from the original, then the treap was modified, and a `ConcurrentModificationException` should be thrown.

3.5 Issues/Debugging

I encountered some issues with design decisions and debugging over the course of this project

- **Debugging Recursive Functions:** Recursive functions are especially challenging to debug since it's hard to follow traces and minor bugs can have drastic impacts on the result. Some specific issues I had included minor mistakes in the handling of case 3 in `remove()`and `StackOverFlowError`s with `meld()`.
- **Generics:** I faced some initial difficulty in trying to understand how I should use Java Generics. This included doubts over the design of the `TreapNode` class (whether or not to use generics there) and universal implementations of all functions. As I read more of the specifications and instructions, I was able to decide on my current design.

3.6 Edge Cases

This featured a multitude of edge cases that had the potential to break my program

- `null` **Input -** All five operations could potentially be passed `null` input as one or more of their parameters. To handle this, I either chose to return without performing any operation or throw an `IllegalArgumentException` depending on my interpretation of the interface specifications.
- **Concurrent Modification -** The intended behavior of any `Iterator` is to recognize concurrent modification while iterating. To handle this I kept track of the number of modifications to the `TreapMap` and threw a `ConcurrentModificationException` if the counter had increased during iteration.
- `Treap` **of the Incorrect Type in** `join()` **and** `meld()` **-** It is possible that another instance of `Treap` that is not `TreapMap` is passed into `join()` and `meld()`. To handle this case, I check for the type of the passed instance and only perform the operation if it is of type `TreapMap`.
- **Duplicate Insertion -** It is possible that a duplicate key is inserted into the `TreapMap`. To handle this, I remove the previous entry associated with this key, then insert the new one.
- **Instantiation of** `TreapMap` **Without Generics -** In Java, standard data structures can be instantiated without generics and can then hold objects of many different types. However, this is not possible in the `Treap` since insertion requires comparisons between existing elements and the element to be inserted. This would result in a `ClassCastException`. To handle this, I perform a check in all functions to ensure the proper types are passed in and throw exceptions accordingly.

- **Operations on Empty Treaps -** It is possible for operations such as `remove()`, `split()`, `lookup()`, etc. to be called on empty treaps. Here, one must ensure that `NullPointerException` is not thrown by handling this case specially.

3.7 Interesting Results

I tested the balance factor on 3 different sizes and averaged them. The results are outlined in the table below.

Number of Elements	10	50	100
Average Balance Factor	1.385	1.827	1.891

I expected the larger treaps to have a lower balance factor, but it turned out to be the opposite. One reason for this might have to do with the sampling size of random numbers, but it is certainly an interesting and peculiar result!

3.8 Karma

3.8.1 `meld()`

I implemented the `meld()` function recursively and abstracted it to perform meld on any two treaps. The logic is described below

- **Parameters:** The following parameters are passed to the recursive function and completely describe a unique recursive state
  - `Treap<K, V> f` that is the first treap to be melded
  - `Treap<K, V> f` that is the second treap to be melded
- **Return Type:** The recursive implementation returns a `TreapNode` that is the new root of subtreap at that position.
- **Base Case:** If either of the treaps are empty, we return the root of the other treap since a meld on an treap and an empty treap is itself.
- **Recursive Step:** I split both treaps across the key of the root of the first treap (this is just an arbitrary choice. It works for any choice). The two resulting left subtreaps are recursively melded together and the two resulting right subtreaps are recursively melded together. Because we performed a split, we can be certain that the meld of the left subtreaps is strictly smaller than the meld of the right subtreaps (strictly smaller in the sense of key magnitudes). The two results are then joined together and the root of the resulting treap is returned.

This function was very fun to implement! The time complexity of the implementation can be analyzed as $O(nlog\frac{n}{m})$ where $n \geq m$ and n, m are the number of nodes in the two treaps to be melded. I'm not entirely sure how to show this, but given more time, I would certainly think about this more!

The implementation of `meld()` forced me to rethink the design of my functions since I needed to be able to join two arbitrary treaps (not necessarily one treap with this instance) and similarly meld two arbitrary treaps. This led to design changes and abstractions of these functions.

3.8.2 Balance Statistics

To find the balance statistics of a given treap, I wrote two functions. `height()` recursively found the height of a treap by adding 1 to the maximum of the heights of the two subtreaps. `balanceFactor()` computes the number of nodes in the treap by doing a simple traversal and maintaining a counter, then computes the minimum subtreap height which is minHeight $= \lceil log_2(n+1) \rceil$ (Given more time, I would prove this rigorously). The balance factor is then computed by performing $BF = \frac{height}{minHeight}$.

4 Testing

This assignment relied on proper testing to ensure that each component/module was functioning properly. An important component of testing included repetition on different combinations of key/value pair types. To do this, I tested my program on `<Integer, Integer>`, `<String, Character>`, and `<ExtraKey, String>` (`ExtraKey` is a custom comparable type that uses `double` comparison) In order to perform thorough testing on my implementation, I wrote a testing suite using JUnit and performed visual inspection on program output.

4.1 Black Box Testing

Black box testing using the `toString()` was an effective tool to ensure that everything was working at a high level. By visually confirming the BST and Max-Heap properties, priorities, key/value pairs, etc. of the treap after each operation, I gained valuable insight into whether I was moving in the right direction.

4.1.1 Black Box Testing on `remove()`, `lookup()`

While implementing my solution, I continuously tested my work by running a few basic tests in a separate file. When implementing `remove()` and `lookup()` specifically, I took advantage of the return signature and printed the result of a given removal or lookup. This was performed alongside the usual inspection of the `toString()` human readable representation and was of immense help as an extra verification for the algorithm's correctness.

4.1.2 Black Box Testing on `insert()`, `split()`, `join()`

Here, the usual visual tests on the unmodified and modified treaps were used to test program correctness.

4.1.3 Black Box Testing on Inavlid Input

I tested a number of different cases that were designed to assess my program under invalid input. The notable tests are listed below

- `null` input in one or more parameters of all functions
- Paramater of wrong type (when `TreapMap` is instantiated withot generics)
- `Treap`of wrong implementing class for `join()` and `meld()`

I mainly checked to see how my program handled invalid input, why it crashed if it did, and ensured the proper exception was being thrown after a fix.

4.1.4 Issues Found During Black Box Testing

- In particular, black box testing for word searching helped me find and fix bugs in my recursive functions. For example, in an initial implementation of the `split()` function, a key equal to the "split key" ended up in the left subtreap rather than the right one. While performing the black box tests, I was able to notice the irregularity in the printed treap and fix it immediately.
- Most edge cases listed in section 4.1.3 required revisions. In particular, `null` input required checks and fixes in every function.

4.2 White Box Testing

White box testing allowed me to be much more thorough in my testing and examine cases that were difficult to create otherwise. While Black Box testing gave me a good idea of how the implementations was faring as a whole, I used White box testing to ensure the underlying logic implementations were correct. Using JUnit, I implemented my own unit tests to individually test different functionalities. I will outline our process for testing those functionalities and some more specific cases here.

Note that the testing suite discussed here can be found in `TreapMapTest.java` , `AdditionalTreapMapTest.java`, and `TreapNodeTest()` and does NOT go through the interface in some instances. This testing suite is written specific to my implementation and probes the internal structure of my treap, nodes, etc. thus it will not pass for a given correct implementation of `Treap`. The testing suite written for peer testing `PeerTreapMapTest.java` works through the interface and will pass for any correct implementation of `Treap`. My discussion below regards the testing suite specific to my implementation. For a thorough discussion of the peer test suite, look into `testingReport.pdf`.

4.2.1 `TreapNode` White Box Testing

Testing on `TreapNode` instances were done using different key/value pair types to ensure robust coverage of cases.

- `priorityTest()` - This test asserted that all `TreapNode` instances were instantiated with a random priority in the range $[0, \texttt{MAX\_PRIORITY})$. The purpose of this test was to ensure correctness of the random priorities.
- `keyValueTest()` - This test asserted that the stored `key` and `value` for each `TreapNode` was the same as intended. The purpose of this test was to ensure correctness of the key/value data storage
- `toStringTest()` - This test asserted that the returned string representation of the `TreapNode` instance was as intended (determined by the randomized priority and inserted key/value). The purpose of this test was to ensure correctness of the `toString()` functionality.

4.2.2 `TreapMap` White Box Testing

To test `TreapMap` I decided to take two main approaches: Randomized and Deterministic. Randomized tests entailed populating a `HashMap` with random unique `key/value` pairs and inserting these into the `Treap`for performance testing. The main advantage of randomized testing over deterministic testing was the possiblity for large scale testing. Thus, randomized testing was used for "regular cases" to ensure the intended functionality was working at a basic level. Deterministic testing on the other hand, made use of manually crafted test cases aimed at exploiting extreme edge cases.

My main strategy for testing was to perform operations individually, then check at each step whether different properties were maintained. I wrote a specific helper function to check this which is described below alongside the specific tests. Note that different verisons of these functions were made to account for different generic types, but the discussion will only describe the underlying logic that each version possesses.

Helper Method: `checkProperties()`
The `checkProperties()` helper method checks that a given treap satisifes the following properties. Note that this function directly probes the internal structure of the treap, given the root node, and traverses through it step by step rather than using an iterator wrapper. This adds an extra layer of verification.

- **BST Property** - While traversing through the `TreapMap` in an inorder fashion it adds the result to a `List`. The list is then checked to be sorted by `key`.
- **Max Heap Property**- While traversing through the `TreapMap` in an in order fashion it ensures that the priority of the current `TreapNode` is larger than both of its children.
- **Duplicates** - While traversing through the `TreapMap` in an in order fashion, keeping track of the visited `key`s it ensures that all keys are distinct.
- **Null keys and/or values** - While traversing through the `TreapMap` in an in order fashion, it ensures none of the keys or values of the nodes are `null`.
- **toString() Correctness** - Check that the string representation of the treap is correct by doing a separate pre-order traversal

Note that `checkProperties()` abstracts and delgates some of these tasks to other helper functions to perform these checks.

Tests

- `IRTest()` - This test first inserts everything from the randomized hashmap into a treap individually. Two new hashmaps are introduced to keep track of the contained and not-contained items. After each insertion, the properties of the treap are checked, and `lookup()` is checked to ensure everything in `contained` could be found correctly. The test then removes all elements incrementally and performs similar checks on properties and `lookup()`. The purpose

of this test was to ensure correctness of regular insertion and removal (and by extension tree rotations and lookup) on varying levels of input size.

- `SplitJoinTest()` - Broadly, this test splits a given treap around an arbitrary key, and joins the two subtreaps back together. There are 4 variations of this test that covers the 4 cases of splitting.
  - `SplitJoinBelowRange()` - Here, the split key is smaller than smallest key of treap. In this case, we expect the entire treap to now become the right subtreap, and the left subtreap to be null. This is asserted in the test.
  - `SplitJoinInRange()` - Here, the split key is within the range of the treap keys, but is not equal to one of the treap keys. In this case, we expect there to be non-null subtreaps. Helper methods are used to assert that all keys in left subtreap are less than split key and all keys in right subtreap are greater than split key.
  - `SplitJoinOnDot()` - Here, the split key is equal to one of the treap keys. In this case, we expect there to be non-null subtreaps AND that the original entry with same key as split key ends up in the right subtreap. Helper methods are used to assert that all keys in left subtreap are less than split key and all keys in right subtreap are greater than or equal to split key.
  - `SplitJoinAboveRange()` - Here, the split key is larger than the largest key of treap. In this case, we expect the entire treap to now become the left subtreap, and the right subtreap to be null. This is asserted in the test.
- `joinTest()` - Because the success of the `SplitJoinTest()` is dependent on both `split()` and `join()` working correctly, it is hard to pinpoint the issue if that test fails. This test independently ensures that `join()` works by randomizing another set of unique key/value pairs that have keys that are all larger than the original set. Two separate treaps are made and are joined. The resulting treap is tested for correct properties.
- `EdgeCaseTest()` - This test checks all the edge cases listed below
  - `remove()`and `lookup()`on Empty Treap - Should return `null` and avoid `NullPointerException`
  - Splits on Empty Treap - Should do nothing and avoid `NullPointerException`
  - Duplicate Insertion - No exceptions should be thrown, old entry should be replaced with new one, all properties should still hold.
  - `lookup()`on `null` and/or Keys not Present in Treap - Should return `null` regardless
  - `insert()`and `remove()null` - Should do nothing and return `null` respectively
  - `join()` on `BogusTreap` - `BogusTreap` is a class that implements the `Treap`interface, but has no legit implementations. An attempt to `join()` a `TreapMap` with a `BogusTreap` should do nothing.
- `iteratorTest()` - This test asserts that the iterator iterates in sorted order, covers all the items in the treap, `NoSuchElementException` is thrown when one tries to call `next()` too

many times, and `ConcurrentModificationException`is thrown when concurrent modification is attempted. I decided not to test the internal structure of the stack used in the iterator because this would end up being redundant with the source code, and I would not be able to truly test its correctness. In any case, the test for sorted order and coverage of all elements is sufficient to guarantee correctness.

- `meldTest()` - This test asserts that the meld function on any arbitray treaps works as intended by verifying the properties, and checking to ensure all elements fromt he original treaps are accounted for.

4.2.3 Issues Found During White Box Testing

White box testing proved to be very useful in pinpointing bugs and specific failures of logic.

- One of the perfect examples of this has to do with the `lookup()` functionality on my custom key `ExtraKey`. In the `EdgeCaseTest()` I assert that `lookup(new ExtraKey(), value)`. That is, I assert that a newly formed instance of `ExtraKey` with the same values as the one in the treap should be indentified as equal to the one in the treap. The expected behavior is for this to reutrn true since the two objects have the same contents. My implementation, however, failed on this case. Further inspection reveals that this occurred because I was using the `.equals()` function in my source code. However, custom keys may not necessarily override the `.equals()` function and may thus leave it compare memory addresses rather than contents. I then fixed this to `compareTo() == 0` for a more universal check.
- In an initial version of my solution, `lookup()` was throwing `NullPointerException` when tried on `null` and keys not present in the treap. This allowed me to pinpoint the issue to a failure to check for the `null` case.