Design Documentation

MySymbolTable:

- The underlying data structure is based upon the *HashMap* found in Java's API.
- I imported it and used it to satisfy the Big-O requirements for the class.
    - For searching, inserting, and deleting, a hashmap averages O(1) complexity, matching the requirements for MySymbolTable.
- Since a hashmap assigns a value to a unique key and is able to perform lookup and other operations on this value, I used HashMap because MySymbolTable requires that the SymbolType stores and performs lookup for the RecordType.
- Here is a link to the structure:
    - http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

MySequence:

- The underlying data structure is based upon the *TreeSet* found in Java's API.
- I imported this structure and used it to satisfy most of the Big-O requirements for the MySequence class.
    - For accessing, searching, deleting, and inserting, TreeSet (which itself is based upon a red-black tree) is O(logN) complexity.

    - I acknowledge that it may not be the most efficient structure in terms of average Big-O for the countNoSmallerThan() method, but, in colloquial terms, it "gets the job done." See more details on this method at the end of this page.
    - However, TreeSet should meet the O(N) requirement for the toStringAscendingOrder() method, seeing that I used a for-each loop to traverse the tree in that method.
- I also used TreeSet because it sorts values into a sequence and does not store duplicates, which are two major requirements for the MySequence class.
- Here is a link to the structure:
    - http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

MyPriorityQueue:

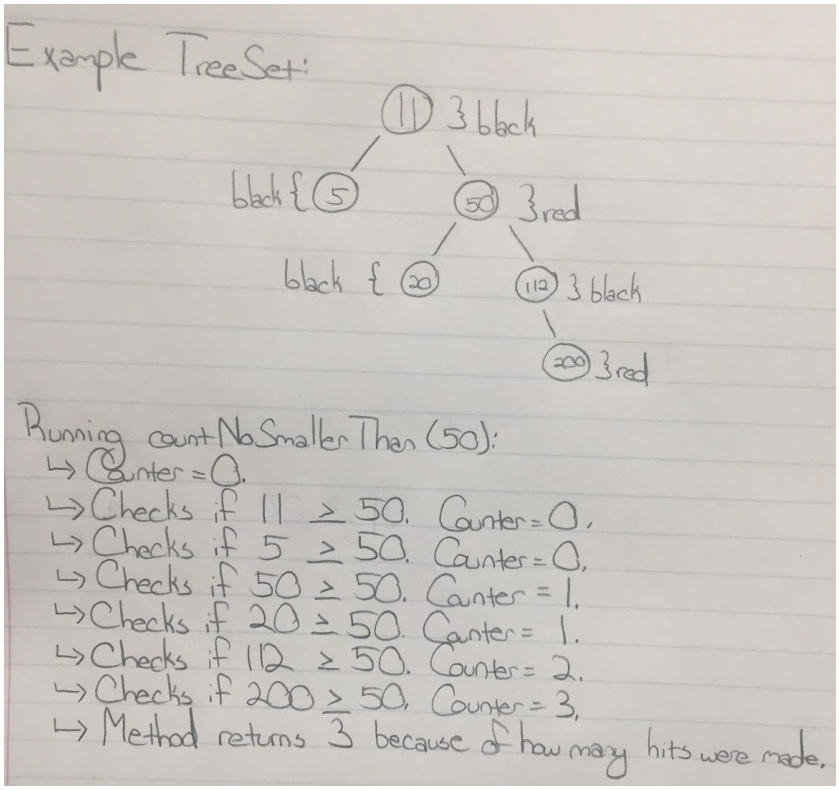
- The underlying data structure is based upon a dynamic array.
- I acknowledge that implementing a priority queue using a binary heap is the most effective/efficient way of implementing such a class, but, for the purposes of creating a priority queue easily and less complicatedly, I chose to use a dynamic array.
- It utilizes a private, internal Pair class in order to avoid casting and type errors from the dynamic array. Pair groups the data of an item and its priority together. This internal class also extends Comparable by implementing the compareto() method.
- No outside code was used to implement this class.

countNoSmallerThan():

- This method initializes a counter variable at 0. Then it uses a for-each loop to go through each item stored in the TreeSet, comparing each to the input item *v*. Using the

compareTo() method, if the current item in the tree greater than or equal to item *v*, then the count increases. The final count of items greater than or equal to item *v* is returned by the method.

- Unfortunately, by using one for-loop and conditional statements, this method is O(N) and not O(logN).
- However, by using compareTo(), this method utilizes Comparable.
- Here is a written example of how this algorithm functions:



- In the above image, I wrote out a red-black tree (since Java's implementation of TreeSet is based upon this kind of tree) and labeled the nodes as red or black accordingly. Based on such a tree, the number *fifty* is passed as an argument into the countNoSmallerThan() method. It compares fifty to each node in the tree (by using the compareTo() method).

The counter remains the same number for “misses” and increments by *one* for “hits.” By the end of the method, the number recorded by the counter variable is returned, symbolizing how many numbers are greater than or equal to the argument passed. In the image, the method returns *three* because three numbers in the TreeSet are greater than or equal to fifty.