1 Introduction

In this project, we’ll be implementing three of the miscellaneous data structures and algorithms: Hash Tables, Heaps, and Radix Sort. Each has been stripped down to a few minimal methods, and I’m writing testcases to test all of them. (Right now, the test suite is pretty thin - but expect it to get better. I’m releasing the spec before all of the tests are done so that all of you can get started writing your solutions.)

As with Project 5, most of your grade (60%, this time) will come from the testcases; 40% will come from a manual inspection of your code. The manual inspection will first of all examine commenting, style, etc. However, it will also include points for the various required features. For instance, if you use some other sort algorithm when I ask you to do Radix Sort or Heap Sort, you’ll lose points!

As with Project 5, you’re not allowed to turn in a modified version of the main class - your code must compile with my standard class.

1.1 Automatic testing with my main()

You must provide three classes: HashTable, MaxHeap, and RadixSort. I have provided a main() method, which tests each of these classes. main() will report the outcome of the testcases, and give a total at the end.

In order for main() to compile, you will need to implement all three classes - and include all of the required public methods. (Two also require some public data.) Each class will be detailed below.

While you are implementing features, I encourage you to write stub methods for everything, so that you can compile the code and test it. Don’t wait until you’ve finished all of the code to start testing it!

The files that I am providing for this program are here: http://www.cs.arizona.edu/classes/cs345/spring16/projects/proj06_files/

2 The HashTable Class

The HashTable class models a single hash table. This hash table models (key,value) pairs; each key is a String, and each value is an Object. (Expect that my testcases may insert values with many different types into your hash table.)
Since this hash table uses (key,value) pairs, your `insert()` method must take two parameters (the key, and the value), and `search()` will return the proper value for a key (or `null`, if the key is not found).

The size of the table is fixed when the table is created (you don’t have to implement resizing).

2.1 Implementing the Linked List
You must use the “linked list” method of conflict resolution. How you implement that is up to you.

If you want, you may use Java’s generic `LinkedList` class; that’s what I tried first. However, Java’s generics are a huge kludge in some ways, and making an array of a generic container is actually kind of hard. (It can be done, but it’s a pain.)

As an alternative, I’ve provided the `RussList` class. It’s a trivial linked list - and it’s not a generic class. But it works OK - and it has fields for both the key and the value.

2.2 Hashing the Key
I have provided a hash function, which you must use. The method is `Proj06Main.genHash(String)`, and it hashes a `String` into an `int` using the SHA-256 algorithm. (I encourage you to go see the implementation.)

Of course, it returns a non-negative integer (basically a 31-bit value). So you’ll need to chop it down to the right size using modulo.

2.3 Required Method: `HashTable(int)`
The constructor must allocate the hash table. If you have any other variables, initialize those as well - however, I don’t expect that you will. The parameter tells you the size of the table that you should allocate.

Throw an `IllegalArgumentException` if the parameter is less than 2.

2.4 Required Method: `void insert(String, Object)`
This method inserts the given (key,value) pair into the hash table. Of course, you should only use the key (not the value) in the hash function.

This method should not check for duplicate keys in the hash table; if the user inserts the same key twice, that’s fine.

Throw an `IllegalArgumentException` if the value is `null`.

2.5 Required Method: `Object search(String)`
This method searches the hash table for the given key. If it finds the key, the method returns the value associated with that key. If it does not, then it returns `null`.  

2
Remember: the \texttt{insert()} method allows you to insert duplicates, and \texttt{search()} can’t tell the difference between entries with the same key. \texttt{search()} should just return the first match it finds (if any) - and not worry if there might be duplicate keys as well.

2.6 Required Method: \texttt{debugDump()}

I have no requirements on this method, except that it must exist. \texttt{main()} will call this if it detects a testcase failure using this object. I encourage you to \texttt{println()} useful debug data.

3 The RadixSort Class

The \texttt{RadixSort} class models a run of the Radix Sort algorithm. It sorts an array of \texttt{ints} in place.

3.1 The public Array

The class \texttt{RadixSort} must have the following instance variable:

\begin{verbatim}
    public int[] values;
\end{verbatim}

Don’t change the name or the type - and don’t make it \texttt{private} - because the testcase code in \texttt{main()} directly accesses this array.

Since \texttt{main()} directly accesses this data field, the class is not required to have a constructor, getters, or setters.

3.2 Required Method: \texttt{int \texttt{getBucket}(int \texttt{value}, \texttt{int \texttt{chunk}})}

Our Radix Sort sorts integers, and it breaks each one down into 4-bit chunks. Chunk 0 is the least significant chunk, and chunk 7 is the most.

\texttt{getBucket()} takes an \texttt{int} as input, along with a chunk number; it must return the 4 bits of that chunk, shifted down so that the value is in the range 0-15 (inclusive).

3.3 Required Method: \texttt{void \texttt{doRadixSort\_onePass}(\texttt{int \texttt{chunk}})}

This method will do one pass of Radix Sort - that is, it must do Counting Sort, using the selected chunk as the key to sort the rest of the values.

3.4 Required Method: \texttt{doRadixSort()}

This method performs the entire Radix Sort. It must call \texttt{doRadixSort\_onePass()} to perform each of the passes through the data.
4 The MaxHeap Class

The MaxHeap class models a single max heap. This heap stores ints.

4.1 public Data

Like RadixSort, this class has public data fields - in this case, two of them:

    public int[] data;
    public int count;

4.2 Required Method: maxHeapify(int)

This method runs the maxHeapify() algorithm, starting at a given index and recursing down through the tree. When the method returns, the value that was previously at the index will have been shoved down to its proper location. (Of course, one of the possibilities is that it doesn’t have to move at all.)

    If the index is not valid, throw an IllegalArgumentException.

4.3 Required Method: buildMaxHeap()

This method runs the buildMaxHeap() algorithm, as described in the slides. It must call maxHeapify() repeatedly.

4.4 Required Method: int removeMax()

This method removes the maximum value from the heap, and then returns it. Before it returns, it fixes up the heap, as shown in the slides.

4.5 Required Method: heapSort()

This method performs Heap Sort. It must call buildMaxHeap() and removeMax().

    If count is not equal to data.length when the method begins, throw
    IllegalArgumentException. When the method returns, count must be zero.

4.6 Required Method: debugDump()

I have no requirements on this method, except that it must exist. main() will call this if it detects a testcase failure using this object. I encourage you to println() useful debug data.

5 Compilation Requirements

Your code must compile on Lectura, using javac. If you write your code in an IDE (like Eclipse), you may have added some package statements which will make your code not compile on Lectura - if so, make sure to remove them before
you turn in your code! Code that does not compile will not be graded; if you have to turn in a replacement version to fix a compile issue, it will be treated as if it was late by one day (or one additional day, as the case may be).

6 Comments and Style

You must comment your code. Make sure that your comments are clear; they need to express both what you are trying to do, and how you plan to do it. Clear comments will make it easier for us to grade your code, and to give you partial credit if/when we find bugs.

Use good Java programming style, including good use of whitespace, consistent indentation, and meaningful variable names. Follow the Java variable naming conventions (start with lowercase, and camel case after that).

Each file should include a header comment, which includes:

- Java class name
- Our class and assignment name (CSc 345 Spring 16 - Project 6)
- Your name
- A description of the basics of the class

7 Turning in Your Program

You must turn in at least the following files:

```
HashTable.java
RadixSort.java
MaxHeap.java
```

You may turn in additional files if you want - but any copy of `Proj06Main.java` or `RussList.java` that you turn in will be ignored.

Turn in the files using the assignment name

```
cs345_s16_proj6
```

7.1 Using turnin

Hopefully, you remember how to use turnin from previous CS courses. However, here are a few reminders:

- turnin can only be run from lectura.cs.arizona.edu; upload your files to lectura using SCP (or a thumb drive in the labs).
- To turn in files, use the command:
turnin <assignment_name> <file1> <file2> ... 

- To confirm that your files have been turned in correctly, check with the command:

  turnin -ls <assignment_name>

- You may turn in as many times as you want; turn in one file at a time, or all of them together. You may also turn in the same file multiple times; turnin will only keep the most recent version.