Assignment 7
CSc 210 Fall 2017
Exercises Due October 16th, 8:00 pm MST
Programs Due October 21st, 8:00 pm MST (EXTENDED)

Introduction

In class, we have been discussing how to use classes such as ArrayLists and HashMaps. For this assignment, you will be implementing a hash table class in Java.

NOTE: THIS ASSIGNMENT HAS TWO DUE DATES.
There are two parts (and two due dates!) to this assignment, exercises done via CodeStepByStep and programs to be turned in via github classroom. The exercises will be due Monday night while the actual programs will be due Thursday night.

You will turn in your code via github classroom. Here is the link to receive your repository: https://classroom.github.com/a/1LROIJXW

For each Java program assigned, you will turn in a .java file for that program. We should be able to run "javac *file*.java" and then "java *file*" on lectura and your program should compile and run successfully.

Your Java programs should adhere to our style guidelines we will give to you. Here is the link: https://www2.cs.arizona.edu/classes/cs210/fall17/StyleGuidelines.pdf

Specification Part I: CodeStepByStep (15 points)

Complete the following exercises on CodeStepByStep by Monday, October 16th, at 8:00 pm.

- Java->recursion-->mystery1
- Java->recursion-->recursionMystery1
- Java->recursion-->recursionMystery1X
- Java->recursion-->repeatString
- Java->recursion-->mirrorSequence

Specification Part IIa: Hash Tables (35 points)

A hash table is a data structure that offers very quick operations such as inserting data and searching for data. You have used a form of a hash table when using the HashMap class in Java. However, you will now create your own hash map implementation.

First, you should read up on what a hash table is. Wikipedia is a good place to start: https://en.wikipedia.org/wiki/Hash_table

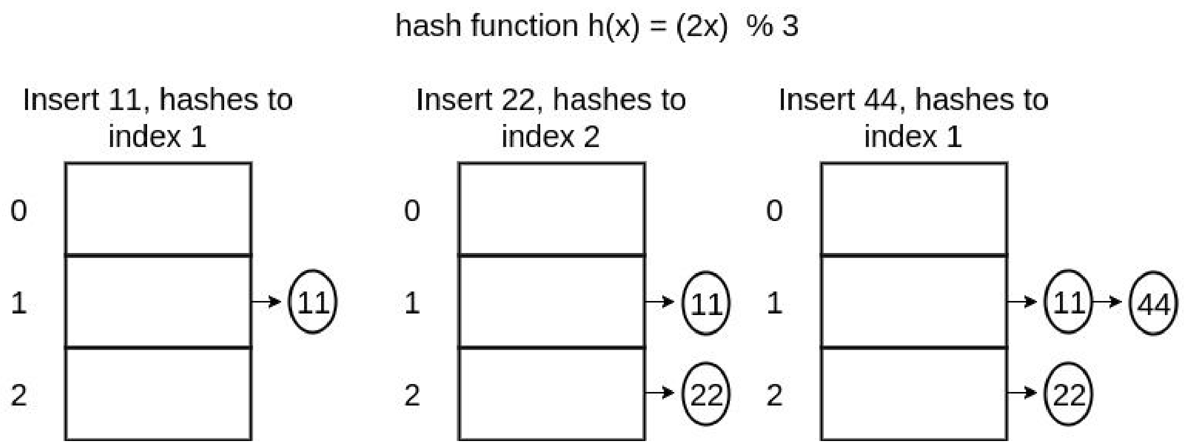
At its heart a hash table (hash map) is an array together with a *hash function* that takes a key and returns an index into that array. We usually call the elements in the array *buckets*. A critical part of writing a good hash map implementation is writing a good (and appropriate) hash function. It must spread out values evenly based on different input values and needs to always produce the same value if the same element is given to it.

For this assignment, you will be tasked with

1. Implementing a HashMap Java class.
2. Writing a hash function for this hash map that hashes Strings.
3. Using this hash map implementation to solve a problem.

A perfect hash function is one that will produce a unique value for every unique key given to it. Writing a perfect hash function is almost impossible. Therefore we do not expect you to do this. Because your hash functions will not be perfect, they will probably produce collisions! Collisions are when two different elements given to the hash function produce the same input. That means some insertion values will have the same resulting index in your hash table. To handle this, we will make every bucket in the array a linked list. This is sometimes called *chained hashing*.

Below is an example hash table, hash function, and inserting three values into it. Notice how we get a collision and insert the new value into the linked list at that index.


hash function h(x) = (2x) % 3
Insert 11, hashes to index 1
Insert 22, hashes to index 2
Insert 44, hashes to index 1
0
1
2
11
22
44

You will write a class called MyHashMap that maps Strings to doubles. It will implement a hash map that uses chaining to handle collisions of elements. You will implement the following public methods:

- A method void *insert(k, v)* that adds the double v to the table at the index produced by hashing k, if it is not already there. If there is already an entry with key k, then the value of the entry is replaced with v.
- A method void *remove(k)* that removes the double from the table at the index of the string k, if it's there. If there is no element with key k, then the method returns having done nothing. It does not report an error.
- A method double *getValue*(k) that returns the double from the table that is associated with the key k if such an element exists. It should return 0.0 if no value is stored in the table for key k. (Note: It would better to return something that indicates the key wasn't found, but since we're returning a double we have limited choices, so we'll return 0.0)
- A method boolean *contains(k)* that returns true if there is a value in the table associated with key k, and false otherwise.
- A method int *size()* that returns the number of values currently in the table.
- A method int capacity() that returns the number of buckets (size of the array) being used by the structure. (Note size() returns how many things are currently being stored and capacity returns how many buckets there are to store things.)
- A method void *printTable()* that prints out the elements of your table. The purpose of this is so we can look at the output and determine how the values are spread throughout your hash map. The output for this method is specified below.
- Two constructors. One with no parameters that creates a table with an internal array of default size 100. One that takes an integer n as a parameter and creates an array of size n to be used by the structure.

In addition you will need some private methods including the following. (Of course you can include as many private methods as you think you need. You are also free to choose whatever fields you think this class will need.)

- A method to compute the hash code of a string. (Note: You must come up with a way to do this, feel free to reference lecture material.)
- A method to resize the table if necessary (see below).

If the number of elements in the hash table ever equals the size of the table, you should double its size (what must you do about pre-existing values?).

Your printTable() method should have the following format:

1. For each entry in the underlying table, print out on a line the index number.
2. For each line, print out the linked list at that index with ' -> ' connecting the nodes. If the linked list does not exist, print out null.

For the example table used earlier, we would have:

```
Index 0: null
Index 1: 11 -> 44
Index 2: 22
```

The output of this method does not need to be exact, we are just using it to look at the spread of values throughout your hash table.

You must turn in at least on file, MyHashTable.java, but you are certainly encouraged to turn in more (especially since you need a linked list). You are not allowed to use any built in java classes within your HashTable class or linked list implementation other than String, arrays, and possibly Double (though you don't really need that last one). You must implement the data structure yourself.

We will be creating a test program that will use your hash map to test if it is working correctly. It is not required, but to test your code is working you can re-write the program HappyScore.java. However, you will now use your HashMap implementation instead of Java's. Refer to the previous spec to refresh on what this program must do.

Specification Part IIb: QuickSort (25 points)

Most of you have probably heard of (and maybe even implemented!) an algorithm called quicksort. Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting. Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

You are to implement a java class called QuickSort.java. It should implement a static method:

public static int[] sort(int[] toSort);

Which will be given an integer array. You will sort this array using quicksort and return the newly sorted array from the method. Note the array you are returning is the one that was passed to you. In other words, don't create a new array. Instead sort the array sent to you as a parameter and then return that array.

NOTE ONE OF THE BENEFITS OF THE QUICK SORT ALGORITHM IS THAT IT USES CONSTANT SPACE. YOUR CODE FOR SORT SHOULD NOT EVERY CREATE A NEW ARRAY. INSTEAD YOU NEED TO SORT THE EXISTING ARRAY I*N PLACE*.

Hints:

1. Since you're sorting the array in place, you will probably want to create a helper function that does the actual recursion so it can include start and end indexes.
2. It doesn't matter which value you pick for the pivot. It is common to pick the first element in the array.
3. To do the partition operation you can run an index from left to right to left as long as the values are not larger than the pivot and another from right to left while the values are not smaller than the pivot. If both these indexes stop before they cross each other, exchange their values and repeat. When the indexes cross each other you know the right half has values larger or equal to the pivot and the left has values less than or equal to the pivot. Exchange the value of the pivot so it appears in the middle.
4. Lastly, it IS cheating to look up a Java implementation of quicksort, it is NOT cheating to look up more information(in pseudo code) on quicksort if you are having trouble understanding it.

You should also write a main function that reads input from standard in into an array and sorts it, and then prints it out You will first read in an integer n that defines how many integers you will read in. You will then read in that many integers into an array. This main function should check for errors in input: These would include no size or a bad size entered (Note, a size of 0 will not be counted as an error. If a size of 0 is entered then the array is empty). Also if the size n is not followed by at least n integers, you should print an error message. You do not need to check that the input doesn't have more items than the n integers. Once you've read the n integers go on to the rest of the program. Once you've read the array, call YOUR sort method to sort it, and then print out the elements of the sorted array, one per line.

Turn in one file called QuickSort.java with the content described above.

Miscellaneous

In order to receive a grade, your repo must only contain your source java files in the root directory of your repo.

This assignment will be submitted through github classroom. Make sure all of your code you would like to submit is in your repository when the due date arrives.

Note: Your output must match what is defined here in the spec. We will give you a small selection of test cases so you can make sure you have the right format. Do not print extraneous output, you may lose a lot of points (This includes prompts when reading in input!). **These test cases can be found in your repo.**

Remember, do not cheat! Refer to the syllabus and first lecture for more information.