

Homework 4 (due by 4 pm on Nov 20, 2014)

Objectives:

- Have a deeper understanding about how HashTable works by implementing one from scratch
 - Get a chance to think about what makes a good hash function
-

In homework assignments 1 and 3, you implemented MyArray and SortedLinkedList to store words parsed from a text file.

This time, your goal is to search a word very quickly. So, your plan is to implement a new Data Structure class

public class MyHashTable **implements** MyHTInterface

MyHashTable must have the following **properties and capabilities**.

- It implements MyHTInterface that is provided to you.
- It uses an array as its underline data structure and each element of the array is DataItem node (private static nested class). Each DataItem has String as its data value and int value to keep track of the frequency of its data value.
- As you learned from the lecture, it does not care about the order because our focus is a faster search.
- It uses **linear probing** as its collision resolution mechanism.
- It has the default capacity of 10 and the default load factor of 0.5. Along with a constructor with no parameters, it should also allow the client to pass an initial capacity value as its constructor's parameter. If the initial capacity is less than or equal to 0, it should complain with an exception.
- Its hash function should be implemented based on the algorithm shown in the lecture 11. For example, the hash value of "cats" is: $(3*27^3 + 1*27^2 + 20*27^1 + 19*27^0) = 60,337 \% \text{arrayLength}$.
- However, it recognizes the importance of making the hash function more efficient. For that purpose, it should apply **Horner's method** (http://en.wikipedia.org/wiki/Horner_scheme) to its hash function as follows.

$$\text{var4} * n^4 + \text{var3} * n^3 + \text{var2} * n^2 + \text{var1} * n^1 + \text{var0} * n^0$$

can be rewritten as follows using Horner's method

$$(((\text{var4} * n + \text{var3}) * n + \text{var2}) * n + \text{var1}) * n + \text{var0}$$

- *In case the number of items in comparison to the array length hits the load factor, **it should rehash** the current hashtable so that the load factor stays within 0.5 at any point.*
- As it rehashes, it should print how many items are now being rehashed and what the new length of the table is to users. (Check the expected results, testing your program with HTTest.java file!)
- Also, **when rehashing**, increase the underline array by twice bigger than the current length. *However, if the new length is not a prime number, then you should use the next prime number that is larger than the new length.* For example, if your current array length is 2 and you need to rehash, 4 is twice bigger than the current array length but you need to use 5 instead of 4 because 4 is not a prime number and 5 is the next prime number.
- Unlike Map or Set interface of Java Collections Framework, it has a few useful (?) methods that can be helpful for a better understanding of Hashtable such as getting the number of collisions and getting the hash value of any string.

Just like homework assignments 1 and 3, you are not allowed to use any of the Collections Framework and, to make it simpler, your clients know that they need to convert all of the words in a file to lowercase while inserting. Thus, you do NOT need to worry about converting strings to lowercases in your implementation.

A word is a sequence of letters [a..zA..Z] that does not include digits [0..9] and the underscore character. Also, your implementation should not make an assumption about the length of words.

Here are examples of non-words: abc123, a12bc, 1234, ab_c, abc_

Testing

There will be one test case provided to you, HTTest.java. **Your output should match the output in the expected result comment section in the HTTest.java file.** Also, test your program with other text files.

Be careful!

- When `display()` method is called, `DataItem` should show its `String` value and its frequency as pair, “[Value, Frequency]”, and the empty space is represented with “**” and any space that used to have an item but deleted later should be represented as “#DEL#”
- It is advised to spend some time to think about how to count the number of collisions. The definition of collision you are to rely on is: “Two different keys map to **the same hash value.**”

There is `MyHTInterface` interface where you can find about the methods to be implemented and their descriptions. Read carefully!

Deliverables:

- A few sheets of paper that have your initial code as well as your comments (Submit this in class that is on the due date.)
 - Remember to focus on your mistakes and be more detailed on what you learned as you write comments on papers!
- Your source code. (***Submit your `MyHashTable.java` file on Blackboard by the due. Do not zip it!***) ***Test extensively!***

Grading:

Your homework will be graded first by compiling and testing it. After that, we will read your code to determine appropriate methods/classes are used. In addition, we will judge the overall style and modularity of your code. Points will be deducted for poor design decisions, uncommented and unreadable code. Your comments on your paper should be able to demonstrate your proper understanding of the code.

- Working code: 60 points
- Coding style (refer to the guideline): 20 points
- Your paper code's comments: 20 points

Late submission will not be accepted and, if you have multiple versions of your code, make sure to submit the correct version. Only the version that is submitted before the due will be graded. When in doubt, ask questions. Do not proceed with your own interpretations!