Version: ~~April 20 2020 4:44 PM~~ April 25 12:20 PM

CS 2223 D20 Term. Homework 3

This homework covers material that extends back to HW2. Based on the performance of the midterm, I think this review is worthwhile.

Homework Instructions

- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples as I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/d20/#policies.
- Due Date for this assignment is 6PM Thursday April 30th. Homeworks received after 6PM will receive zero credit.
- Submit your assignments electronically using the canvas site for CS2223. Submit your homework under "HW3". You must submit a single ZIP file that contains all of your code as well as the written answers to the assignment.
- All of your Java classes must be defined in a packager **`USERID.hw3`** where USERID is your CCC user id (i.e., your email address without the @wpi.edu).

Please copy these classes into your Homework area

- `algs.hw3.AVL` → `USERID.hw3.AVL`
- `algs.hw3.BST` → `USERID.hw3.BST`
- `algs.hw3.Heap` → `USERID.hw3.Heap`
- `algs.hw3.TaleOfTwoCitiesExtractor` → `USERID.hw3.TaleOfTwoCitiesExtractor`
- `algs.hw3.Question1` → `USERID.hw3.Question1`
- `algs.hw3.Question2` → `USERID.hw3.Question2`
- `algs.hw3.Question3` → `USERID.hw3.Question3`

You will refer to the existing `InstrumentedSeparateChainingHashST` class. Note you could also copy your hw2 `TaleOfTwoCitiesExtractor` which will work as is for HW3

Version: ~~April 20 2020 4:44 PM~~ April 25 12:20 PM

Q1. HeapSort Empirical Evaluation (20 pts)

Algorithm 2.7 in Sedgewick, Heapsort, shows how to use a heap to sort a comparable array. The code is provided for you in **`algs.hw3.Heap`**. The first step is to construct a heap from a `Comparable[]` array. This takes place in the `constructHeap(a)` method.

```
// construct heap from the raw array of which we know nothing.
int n = a.length;
for (int k = n/2; k >= 1; k--) {
  sink(a, k, n);
}
```

If you look at this code with an eye towards its performance, it sure looks like the **`for`** loop will execute n/2 times (which means its performance is linearly dependent on the size of the array). You also know that the **`sink`** method behavior (in the worst case) is directly proportional to the log of the number of elements in the heap. Thus, at first glance, it looks like this behavior will be proportional to ~ (N*log N)/2.

It turns out that you can mathematically prove that the performance constructHeap is in direct proportion to N alone. Your task is to instead count the number of comparisons and exchanges, and validate the proposition (page 323) that it will, in fact, require fewer than 2N compares and fewer than N exchanges to construct a heap from N items.

Copy `algs.hw3.Heap` into `USERID.hw3` and modify it to record empirical results. For the domain of data, use uniformly computed random numbers from 0 to 1 (such as you can generate from `StdRandom.uniform()`), and generate a table of results (showing N, # of exchanges, and #comparisons) for N=16, 32, 64, … 512. For each size N, run T=100 trials, and record the maximum number of exchanges and comparisons you witnessed **solely during the constructHeap construction**.

Do your empirical results support the proposition? Explain why or why not.

Copy the `Question1` class to `USERID.hw3` and modify it so its output should look something like this:

```
N       MaxComp       MaxExch
16      21            11          NOTE YOUR NUMBERS WILL BE DIFFERENT THAN THESE
32      xxx           yyy
64      xxx           yyy
128     xxx           yyy
256     xxx           yyy
512     xxx           yyy
```

Version: ~~April 20 2020 4:44 PM~~ April 25 12:20 PM

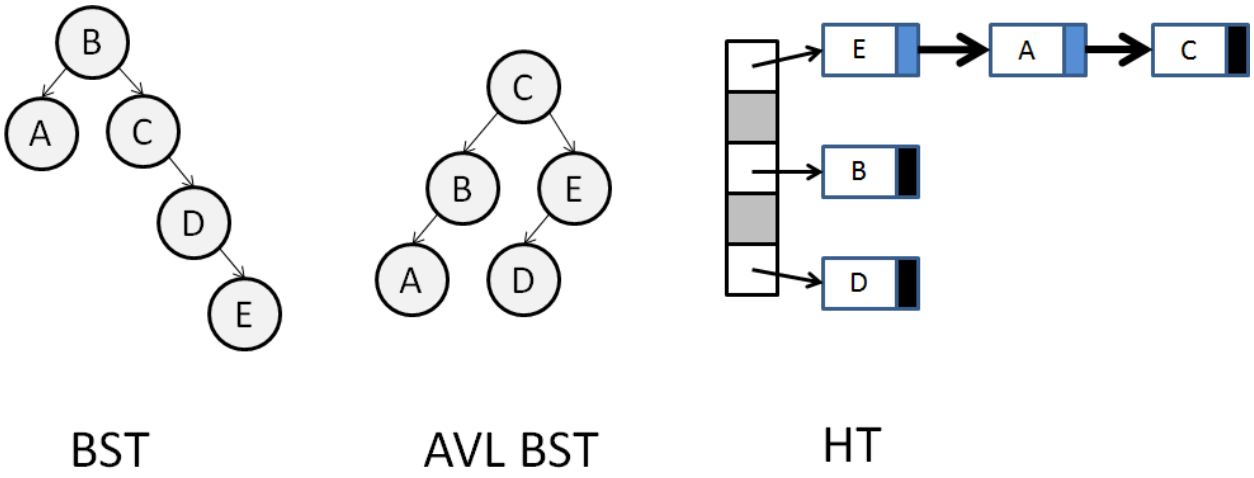
Q2. Empirical Evaluation of Symbol Table structures (50 pts)

This question explores the different structures that result from using binary search trees and separate chaining hash table to support the Symbol Table API, which allows you to associate a (key, value) pair for future retrieval.

Once again, using the Tale Of Two Cities data set, modify this program to produce the following table. This question is based on the number of comparisons needed to locate each key in the symbol table.

- For Binary Search Trees, the depth of a node reflects the distance from the root, and this depth is one value smaller than the total number of comparisons needed to locate that node in the tree
- The same is true of AVL trees
- For Separate Chaining Hash Symbol Tables, each of the buckets stores its size, and each of these will contribute to the overall values.

For example, If there were five (key, value) pairs stored in the symbol table implemented as above as follows (note that only the keys are shown).



In the above structures, the number of comparisons to find each of the five keys (note that hashing doesn't count as a comparison) is:

key	BST	AVL	HT
A	2	3	2
B	1	2	1
C	2	1	3
D	3	3	1
E	4	2	1
Avg.	2.4	2.2	1.6

Your task is to complete this experiment using the Tale Of Two Cities data set to record the count of each word in a symbol table. You will have to use put (key, value) properly. That is, for the first

occurrence of a given word, w, you would `put(w, 1)`. Use the existing API to determine when a word already exists in the symbol. For each subsequent occurrence of a given word, just increment the value associated with w by calling `put()` with a count that is one greater.

To be clear, the depth of a node in a binary search tree is the number of edges it takes to get to that node from the root. Thus the root of the binary search tree has a depth of zero. With regards to the number of comparisons which you need to output below, this is a reflection of how many comparisons it takes to determine whether a key value exists within the AVL, BST or Hash Symbol Table. The number of comparisons for a key value in the BST and AVL tree is 1 greater than the depth of the node. Why? Because you need to compare just to see if you have even found the key you are looking for, even when looking for the key associated with the root of a tree.

For the Hashtable, you don't count the cost of hashing a key to its individual bucket. You only count the number of comparisons to find these keys. Note this part of the question is the trickiest part of this homework. Come to office hours if you have questions.

When you are done, your output should look like this:

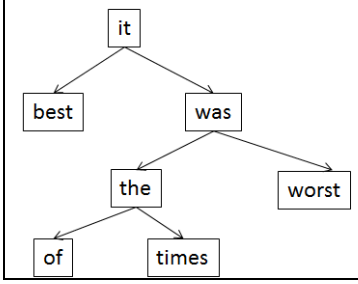
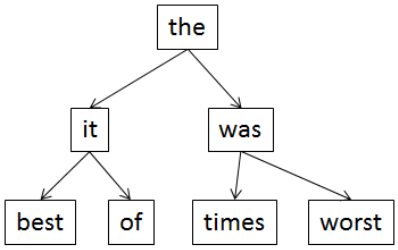
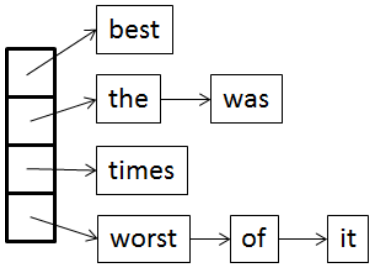
```
There are 10650 unique words.
The Height of the BST is 29
The Height of the AVL is 15

N      1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,  15,  16,  17,  18,  19  …
#BST   1,   2,   4,   8,  15,  30,  57,  98, 159, 252, 383, 549, 740, 886, 997,1085,1053,1024, 955, …
#AVL   1,   2,   4,   8,  16,  32,  64, 128, 256, 512,1020,1944,2904,2583,1066, 110
#HT 2035,1977,1820,1549,1204, 867, 557, 331, 166,  92,  40,  11,   1

AVG. BST NumberOfComparisons:16.509671361502347
AVG. AVL NumberOfComparisons:12.716338028169014
AVG. HT NumberOfComparisons:3.6068544600938965
```

Tasks:

- Complete implementation of collect() in AVL
- Complete implementation of height() and height(Node) in BST
- Complete implementation of collect() in BST
- Complete implementation of Question2

Consider inserting the keys `{ "it", "was", "the", "best", "of", "times", "it", "was","the", "worst", "of", "times" }` into a BST, an AVL and the Hast Table. The results appear on the next page. Note that the structures represent a symbol table, so the BST, AVL and Hash Table all associate the respective frequencies of each word in the structures (not shown below). The following words appear twice ("it", "was", "the", "of", "times") and the words "best" and "worst" appear just once. So these keys would have either 2 or 1 associated with them.

Version: ~~April 20 2020 4:44 PM~~ April 25 12:20 PM

BST TREE	AVL TREE	HASH TABLE (M=4)
		

With these structures, here are the counts of comparisons for locating each key.

key	BST	AVL	HT
Best	2	3	1
It	1	2	3
Of	4	3	2
The	3	1	1
Times	4	3	1
Was	2	2	2
Worst	3	3	1
Avg.	2.714286	2.428571	1.571429

Output would look like this:

```
There are 7 unique words.
The Height of the BST is 3
The Height of the AVL is 2
N      1,   2,   3,   4,   5,   …
#BST   1,   2,   2,   2,
#AVL   1,   2,   4,
#HT    4,   2,   1,

AVG. BST NumberOfComparisons:2.7142857142857144
AVG. AVL NumberOfComparisons:2.4285714285714284
AVG. ST NumberOfComparisons:1.5714285714285714
```

As a further hint, please review the code example “CountingObject” I’ve made this available in day 18 and the “outputDepthInfo” method in BST as found in day 17. You will need to pull the latest version of the Algorithms D2020 repository to get this code.

Version: ~~April 20 2020 4:44 PM~~ April 25 12:20 PM

Q3. Binary Search Trees (30 pts)

Using Binary Search Trees to build a Symbol Table that counts the number of occurrences of unique words in *The Tale Of Two Cities* by Charles Dickens. Copy the TaleOfTwoCitiesExtractor that you had used for HW2 into your USERID.hw3 package. Q2 Copy **`algs.hw3.Question3`** into your `USERID.hw3` package and modify it for this problem.

Q3.1 (10 points) Complete the method that returns the Key whose Value is highest.

```
public String mostFrequent() { … }
```

You can add any number of helper methods as part of this assignment.

Q3.2 (15 points) Complete the method that prints in ascending order the number of words that appear only once in the Binary Search Tree (i.e., their count is 1) **and returns the total number of keys sthat were printed.**

```
public int printUnique() { … }
```

You can add any number of helper methods as part of this assignment. Note that the output will likely exceed your console history, so you will only see the last

Q3.3 (5 points) `Question4` should construct a BST from the *Tale of Two Cities* and output the 10 most common words (together with their corresponding appearance counts). It does this by repeatedly deleting the key whose count frequency is highest in the BST until all values are printed. The output should look like:

```
Top ten most frequent words

the    7983
and    4935
of     3999
to     3460
a      2908
in     2579
his    2005
it     2003
i      1913
that   1889

Number of words that appear once
aa
aaabusiness
aamatter
aback
abandon
…
youths
youties
youunder
5158 unique words.
```

Version: ~~April 20 2020 4:44 PM~~ April 25 12:20 PM

Change Log

1. ANY changes will appear here
2. Binary Search Tree has a height of 29.
3. Clarified difference between Depth and #of Comparisons