COMP2402B (Fall 2015)

Assignment 4 of 5 — Due: November 23rd, 2015 at noon.

Caution

You will be uploading your solutions to cuLearn but this assignment <u>does</u> have several written components.

Only the latest submission received before the deadline will be used to determine your mark on this assignment.

You must adhere to the following rules (as your submission may be subjected to automatic marking system):

- ***Download** the compressed file **"COMP2402-F15-A4-(Start Skeleton).zip"** from cuLearn.*
- ***Retain the directory structure** (i.e., if you find a file in subfolder "comp2402a4", you must not remove it).*
- ***Retain the package directives** (i.e., if you see "package comp2402a4;" in a file, you must not remove it).*
- ***Do not rename any of the methods already present** in the files provided.*
- ***Do not change the visibility of any of the methods provided** (e.g., do not change private to public).*
- ***Do not change the main method of any class**; on occasion the main method provided will use command line arguments and/or open input and output files – you must not change this behaviour).*

*Please also note that **your code may be marked for efficiency as well as correctness** – this is accomplished by placing a hard limit on the amount of time your code wil be permitted for execution. If you select/apply your data structures correctly, your code will easily execute within the time limit, but **if your choice or usage of a data structure is incorrect**, your code may be **judged to be too slow** and **it may receive a grade of zero**.*

*You are expected to **demonstrate good programming practices at all times** (e.g., choosing appropriate variable names, provide comments in your code, etc.) and **your code may be penalized if it is poorly written**.*

Part I - Written Question

1. For this question you will be analyzing specific properties of a skiplist when the coin used in the "coin-flips" to generate the skiplist is NOT a fair coin.

   Your analysis should be similar to that found in Chapter 4.4 of the textbook. Your analysis is expected to first provide the expression associated with each of the following expected values, and then a proof of correctness for each of these expressions.

   If the probability of the unfair coin flip resulting in a value of "Heads" is set to p where $0 < p < 1$ (so that the probability of "Tails" is $1 - p$), then what is:

   a. the expected number of promotions that would be received by an arbitrary node when adding it to the skiplist?

   b. the expected height of a skiplist that contains n elements?

   c. the expected number of nodes (including the sentinel) in a skiplist of n elements?

Part II - Implementation Questions

2. For this question you will be reading a file that contains an arithmetic expression that has been written in "postfix" notation. Most of the arithmetic expressions you have encountered in academia so far have been written in "infix" notation, where binary operators like +, –, *, etc., are written between the operands (and often bracketed to indicate operation precedence). In postfix notation, the operators are written after the operands.

   As a clarifying example, the "infix" expression:

   (1 + 2) * 3

   would be equivalent to the "postfix" expression:

   1 2 + 3 *

   Every line in the file that your program reads will contain either a single operand (which is guaranteed to be an integer, but may be positive, negative or zero) or a single operator from the set {+, –, *} (i.e., integer addition, subtraction, or multiplication, respectively). Your program must read these values and use them to construct a binary tree corresponding to the expression (where the integer operands will appear as leaf nodes and the operators will be non-leaf nodes). Your program is then expected to print out the tree as an infix expression, using brackets to indicate the precedence, and provide the evaluated result of the expression as well.

   Your program must recognize an invalid expression (e.g., "10 + +") and throw an appropriate exception.

   Your solution is permitted to use a stack/queue/deque to create the tree, but you must <u>TRAVERSE THE TREE TO EVALUATE THE RESULT</u> (i.e., you cannot use a stack to evaluate the expression).

3. For this question you will revisit your solution to question 2 but this time your program is expected to also provide support for the unary negation operator. This means that the appearance of a – symbol might indicate a negation operation applied to the term in the expression or a subtraction operation applied to the two previous terms in the expression. You are to assume that binary subtraction has a higher operator precedence than unary negation. To clarify, the file:

   10
   –
   20
   +

   should correspond to the infix expression "(-10) + 20" while

   10
   20
   –
   –

   should correspond to the infix expression "-(10 - 20)".

   Your program must recognize an invalid expression (e.g., "10 + +") and throw an appropriate exception.

   Your solution is permitted to use a stack/queue/deque to create the tree, but you must <u>TRAVERSE THE TREE TO EVALUATE THE RESULT</u> (i.e., you cannot use a stack to evaluate the expression).

Part III - Written and Implementation Question

4. We have seen two hash table implementations in class that yield expected constant time $find()$ and $delete()$ functions. In this problem, you will implement a different hash table that has guaranteed constant time $find()$ and $delete()$ functions; unfortunately this entails a more complicated $add()$ function, that may not always run in constant time.

   The **cuckoo hash** uses two hash functions, $h_1(y)$ and $h_2(y)$, and two hash tables (arrays), T_1 and T_2, to store the data. Every element y in the hash table is either located in $T_1[\ h_1(y)\ ]$ or in $T_2[\ h_2(y)\ ]$. The $find()$, $delete()$, and $add()$ functions in the cuckoo hash are defined as follows:

   - $find(y)$ simply checks if y is located in $T_1[\ h_1(y)\ ]$ or in $T_2[\ h_2(y)]$
     (n.b., this will obviously run in constant time, because only two locations must be checked)

   - $remove(y)$ calls $find(y)$ to determine if y is in the hash table and, if so, removes it
     (n.b., again, since this is basically just a find operation, this will be constant time as well)

   - $add(y)$ will always add y to $T_1[\ h_1(y)\ ]$, but this process is complex if this is already occupied...

   When a function call to $add(y)$ determines that y should be added at $T_1[\ h_1(y)\ ]$, if that location is already occupied by some other element z (i.e., $h_1(y) = h_1(z)$) it might be necessary to "eject" the element z from its currently location (i.e., $T_1[\ h_1(z)\ ]$) = $T_1[\ h_1(y)\ ]$)) and insert it into its correct position in T_2 (i.e., $T_2[\ h_2(z)\ ]$), in order to make room for y in $T_1[\ h_1(y)]$. In turn, $T_2[h_2(z)]$ might already be occupied as well, so that element may need to be "ejected" (into T_1) as well.

   It is hypothetically possible for this process to repeat indefinitely - an ejection to make room in T_1 might require an ejection in T_2, which itself may require a different ejection in T_1, etc. This is unfortunate but it only becomes a real problem when the process "cycles" (e.g., y gets ejected to make room for x, z gets ejected to make room for y, and x needs to be ejected to make room for z). To address this your $add(y)$ function must be able to detect the occurrence of a cycle.

   The easiest way to detect the occurrence of a cycle is to count the number of times an $add()$ operation needs to "eject" something - since there are only n unique items in the cuckoo hash, if the number of ejections every exceeds n then the only possible explanation is a cycle. If you encounter a cycle, you will need to "rehash" the entire hash table - this entails choosing two new hash functions (i.e., choosing two new random odd values z1 and z2) and rehashing every element using these new hash functions.

   To reduce the likelihood of cycles, it is necessary for the capacity (i.e., $T_1.length + T_2.length$) of the cuckoo hash to always be twice the number of elements n. In other words, if the number of elements n ever exceeds $\frac{T_1.length + T_2.length}{2}$, then you must resize the hash table by choosing the smallest integer dimension $d \geq 1$ that will result in a table with a capacity that is greater than $3n$. Please also note that we are operationally defining dimension d (for this assignment) such that $T_1.length = T_2.length = 2^d$.

   Please note that you are expected to use the binary multiplicative hash function (Section 5.1.1 of the required textbook) for all your hash functions in your CuckooHashTable implementation, and **whenever you $resize()$ or $rehash()$ your table you should visit everything in T_1, in the order that it is stored in the array, before visiting everything in T_2. Failure to follow the latter instruction may result in a mark of zero, as it will no longer be possible to test the behaviour of your program against a model solution.**

   Please also note that your CuckooHashTable implementation must use a constructor that takes the "random" numbers as arguments (i.e., this constructor doesn't need to randomly generate values for z1 and z2). This constraint will allow us to test your programs quickly, with values known to collide. Failure to follow the latter instruction may result in a mark of zero.

A useful interactive visualization of the cuckoo hash table can be found online at: http://www.lkozma.net/cuckoo_hashing_visualization/.

In addition to the implementation of the cuckoo hash table, for this question, you are expected to conduct some experiments to compare the efficiency of your implementation with Java's HashTable class (http://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html) and the LinearHashTable class from the ods package(http://opendatastructures.org/ods-java.tgz). Your comparisons should consider the time needed to create a large hash table from an empty hash table and the time needed for finding items in a large hash table. To clarify, the experiments should be as follows:

Compute the average (over at least 30 trials) amount of time required to start from an empty hash table and add 1,000,000 randomly generated integers. Compute this average for each of the different hash tables (i.e., the Java HashTable, the ODS LinearHashTable, and your implementation of the cuckoo hash).

Starting from an empty hash table, add the numbers from [0, 100000) - this will result in a hashtable with n = 100,000. Compute the average (over at least 30 trials) amount of time required to find 2,000,000 random values from [0, 100000) (n.b., for this experiment, every value you attempt to find should be in the hash table somewhere, since you already added all the numbers from 0 to 100,000). Compute this average for each of the different hash tables (i.e., the Java HashTable, the ODS LinearHashTable, and your implementation of the cuckoo hash table).

Starting again from the hash table that contains all of the numbers from [0, 100000), compute the average (over at least 30 trials) amount of time required to find 2,000,000 random values when **exactly 50% of the values to be found are guaranteed to be in the hash table** (i.e., from [0, 100000)) and **exactly 50% of the values are guaranteed to NOT be in the hash table** (e.g., from [100000, 200000)). Compute this average for each of the different hash tables (i.e., the Java HashTable, the ODS LinearHashTable, and your implementation of the cuckoo hash table). Your comparison should be submitted in a PDF file called HashTableComparisons.pdf in your submitted zip file.