Assignment 4

COMP 352: Data Structure and Algorithms (Fall 2018)
Due: December 2 (Sunday) at 11:59 PM

---

*Important Note: No deadline extension will be granted and no late assignments will be accepted **beyond that due date** (not even with a penalty!)*

Part 1: Written Theory Questions (40 Marks)

Question 1.

(a) Consider a binary tree of size n with the keys {0,1,2,…,n-1}. For n=4, how many binary search trees are possible? How many of these are also AVL trees?

(b) For n=10, draw a valid AVL tree with (a) minimum height and (b) maximum height. Note the value of each height as well.

(c) For n=10, draw a valid AVL tree where the removal of key 6 requires the greatest possible rebalancing operations (or rotations). How many operations is this?

Question 2. Consider the following sorting algorithm. The algorithm takes an array of unsorted comparable objects, places them in an AVL tree, and removes them from the tree in sorted order. (a) What is the time-complexity to add the elements to the tree? (b) What is the most efficient way to remove the elements from the tree in sorted order? (c) What is the time complexity of removing the elements? (d) What is the total time complexity? (e) What is the total memory complexity?

Question 3. We want to maintain a helper variable in our AVL tree that maintains the smallest key that has been added to the tree so far. In this case, we can run `firstEntry()` in constant time. (a) Describe how to update `put(k,v)` and `remove(k)` to add this functionality. (b) Does this help reduce the time complexity of the sorting algorithm in the previous question? Why or why not?

Question 4. In the textbook, AVL trees store the height of each node's subtree. A more common implementation is to have each node store a *balance factor*, which is defined as the height of its left subtree minus the height of its right subtree. Thus, the balance factor of a node is always equal to –1, 0, or 1, except during an insertion or removal, when it may become temporarily equal to –2 or +2.

Write an efficient algorithm for computing the height of a given AVL tree that stores a balance factor at each node. Your algorithm should run in time `O(log n)` on an AVL tree of size `n`. In the pseudocode, use the following terminology: `T.left`, `T.right`, and `T.parent` indicate the left child, right child, and parent of a node `T` and `T.balance` indicates its balance factor (-1, 0, or 1). For example, if `T` is the root we have `T.parent=null` and if `T` is a leaf we have `T.left` and `T.right` equal to `null`. The input is the root of the AVL tree. Justify correctness of the algorithm and provide a brief justification of the runtime.

Question 5. Assume a hash table utilizes an array of 13 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: *h(k)=k mod 13*.

(a) Draw the contents of the table after inserting elements with the following keys: {32, 147, 265, 195, 207, 180, 21, 16, 189, 202, 91, 94, 162, 75, 37, 77, 81, 48}

(b) What is the maximum number of collisions caused by the above insertions?

Question 6. To reduce the maximum number of collisions in the hash table described in Question 5 above, someone proposed the use of a larger array of 15 elements (that is roughly 15% bigger) and of course modifying the hash function to: *h(k)=k mod 15*. The idea was to reduce the *load factor* and hence the number of collisions. Does this proposal hold any validity to it?

- If yes, indicate why such modifications would actually reduce the number of collisions.
- If no, indicate clearly the reasons you believe/think that such proposal is senseless.

Question 7. Assume an *open addressing* hash table implementation, where the size of the array $N = 19$, and that *double hashing* is performed for collision handling. The second hash function is defined as: $d(k) = q - k \bmod q$, where k is the key being inserted in the table and the prime number $q = 11$. Use simple modular operation ($k \bmod N$) for the first hash function.

(a) Show the content of the table after performing the following operations, in order: `put(42), put(19), put(48), put(20), put(72), put(18), put(48), put(27), put(9)`

(b) What is the size of the longest cluster caused by the above insertions?

(c) What is the number of occurred collisions as a result of the above operations?

(d) What is the current value of the table's *load factor*?

Question 8. Assume the utilization of *linear probing* instead of *double hashing* for the above implementation given in Question 7. Still, the size of the array $N = 19$, and that simple modular operation ($\boldsymbol{k \bmod N}$) is used for the hash function. Additionally, you must use a special *AVAILABLE* object to enhance the complexity of the operations performed on the table.

(a) Show the contents of the table after performing the following operations, in order:
`put(29), put(53), put(14), put(95), remove(53), remove(29), put(32), put(19), remove(14), put(30), put(12), put(72)`

(b) What is the size of the longest cluster caused by the above insertions? Using Big-O notation, indicate the complexity of the above operations.

(c) What is the number of occurred collisions as a result of the above operations?

Part 2: Programming Question (60 Marks)

This assignment will explore the performance of various types of hash tables and the influence of load factor (the number of elements added to the table vs. the size of the table) on their performance.

Step 1: Hash Code Generation. Create a map element class for `<Key, Value>` pairs. This will be a non-generic implementation. Specifically, have `Keys` of `Integer` type, while `Values` can be anything you want. Ensure your implementation has the following two methods (in addition to standard methods you will need):

* Write a constructor that generates a new pair with a *random* `Integer` key
* Write a method: `public int hashCode()`. This should implement one of the hashing methods discussed in class (e.g., polynomial accumulation, cyclic shifts, etc.). The output of `hashCode()` is `int`; your hash does not have to use the full capacity of `int` but should be at least 16 bits.

Step 2: Compression and Hash Tables. Create an abstract hash table with a fixed capacity (passed in as an `int` at construction time). Create a set of children classes that are concrete hash tables that implement the following collision resolution mechanisms: (1) separate chaining (the bucket object can be any suitable data structure), (2) linear probing, and (3) quadratic probing. Any function that is the same across these three implementations should be placed in the abstract parent class. To compress the output of `hashCode()` to an index in the table, mod the output by the capacity of the table. The following methods should be supported:

size():	Returns the number of entries in M.
isEmpty():	Returns a boolean indicating whether M is empty.
get(k):	Returns the value v associated with key k, if such an entry exists; otherwise returns null.
put(k, v):	If M does not have an entry with key equal to k, then adds entry (k,v) to M and returns null; else, replaces with v the existing value of the entry with key equal to k and returns the old value.
remove(k):	Removes from M the entry with key equal to k, and returns its value; if M has no such entry, then returns null.

Step 3: Instrumentation. Instrument your hash table with the following data gathering methods. Each invocation of `put(k,v)` should print out:

(1) the size of the table,
(2) the number of elements in the table after the method,

(3) the number of keys that resulted in a collision (you do not have to decrement this amount after a remove),
(4) the number of probing attempts before adding the element (for linear/quadratic probing only) and the number of items in the bucket (for separate chaining).

Additionally, each invocation of `get(k)`, `put(k,v)`, and `remove(k)` should print the time used to run the method. If any `put(k,v)` takes an excessive amount of time, handle this with a suitable exception.

Step 4: Validate. Construct a hash table with each of the three collision resolution mechanisms (separate chaining, linear probing and quadratic probing), each with capacity of 100 (note: you should never use a non-prime in practice but do this for the purposes of this experiment). For each table, generate 50 random `<key,value>` pairs and run `put(k,v)` on each pair. Next run `get(k)` on each of the 50 keys. Then run `remove(k)` on the first 25 keys, followed by `get(k)` on each of the 50 keys. Ensure your table functions correctly for all values.

Step 5: Experiment and Interpret. Construct a new hash table of each of the three types, each with capacity 100. Now generate 150 `<key,value>` pairs. Write a short document to:

1) Describe (by inspection or graphing) how the time to run `put(k,v)` increases as you approach (for example) 50 values, 75 values, 95 values, 100 values, and 150 values; and provide reasons for your observations.
2) If your `put(k,v)` method takes an excessive amount of time, describe why this is happening and why it happens at the value it happens at.
3) Rerun the experiment for quadratic probing with a table that has capacity of 101 (which is a prime number). Does this make a difference for any of the experiments? If so, why?

Step 6: Dynamic Resizing. For one of the hash collision resolution mechanisms (either one is fine), implement dynamic resizing. For dynamic resizing, the capacity of the table should start at 128 and double every time `size()` reaches one half of the table's `capacity`. That is, once 64 out of 128 elements have been added, the capacity will increase to 256. Benchmark the cost of adding 10,000 elements to the table, the cost of an additional `put(k,v)` on a table of this size, and the cost of a `get(k)`.

Deliverables:

- A written specification of each class you implemented, including the abstract parent hash table. Include any information about design decisions.
- A written report with the trial run from Step 4, answers to Step 5, and a description of times for Step 6.
- Well-formatted and documented Java source code and the corresponding class files.

Submission Notes

- For the programming component, you must make sure that you upload the assignment to the correct directory (Programming) of Assignment 4 using EAS. Assignments uploaded to the wrong directory will be discarded and no resubmission will be allowed.
- Submit all your answers to written questions in PDF. Trees can be hand-drawn but all answers should be typed. Please be concise and brief (less than 1⁄4 of a page for each question) in your answers.
- The programming part must be done in groups of 2 students. You have to indicate the 2 members of your group in your submission by listing their student IDs and full names. Only one submission per group is allowed.
- For the Java programs, you must submit the source files together with the compiled executables. The solution for programming part must be zipped together into one .zip or .tar.gz file and submitted via EAS. You may upload at most one file to EAS.