Department of Computer Science and Software Engineering Concordia University

COMP 352: Data Structure and Algorithms
Summer 2020

Assignment # 3

Due date and time: **June 12th 11:59 PM**

Part 1: Written Questions (50 marks):

Question 1

a) Given a tree T, where *n* is the number of nodes of *T*.
Give an algorithm for computing the depths of all the nodes of a tree T. What is the complexity of your algorithm in terms of Big-O?

b) We say that a node in a binary search tree is full if it has both a left and a right child.
Write an algorithm called *Count-Full-Nodes(t)* that takes a binary search tree rooted at `node t`, and returns the number of full nodes in the tree. What is the complexity of your solution?

Question 2

a) Draw the min-heap that results from the bottom-up heap construction algorithm on the following list of values:
20, 12, 35, 19, 7, 10, 15, 24, 16, 39, 5, 19, 11, 3, 27.

Starting from the bottom layer, use the values from left to right as specified above. Show immediate steps and the final tree representing the min-heap. Afterwards perform the operation `removeMin` 6 times and show the resulting min-heap after each step.

b) Create again a min-heap using the list of values from the above part (a) of this question but this time you have to insert these values step by step using the order from left to right (i.e. insert 20, then insert 12, then 35, etc.) as shown in the above question. Show the tree after each step and the final tree representing the min-heap.

Question 3

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*.

i) 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.

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

Question 4

To reduce the maximum number of collisions in the hash table described in Question 3 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 is 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 5
Assume an *open addressing* hash table implementation, where the size of the array is $N = 19$, 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 is = 7. Use simple modular operation ($k \bmod N$) for the first hash function.

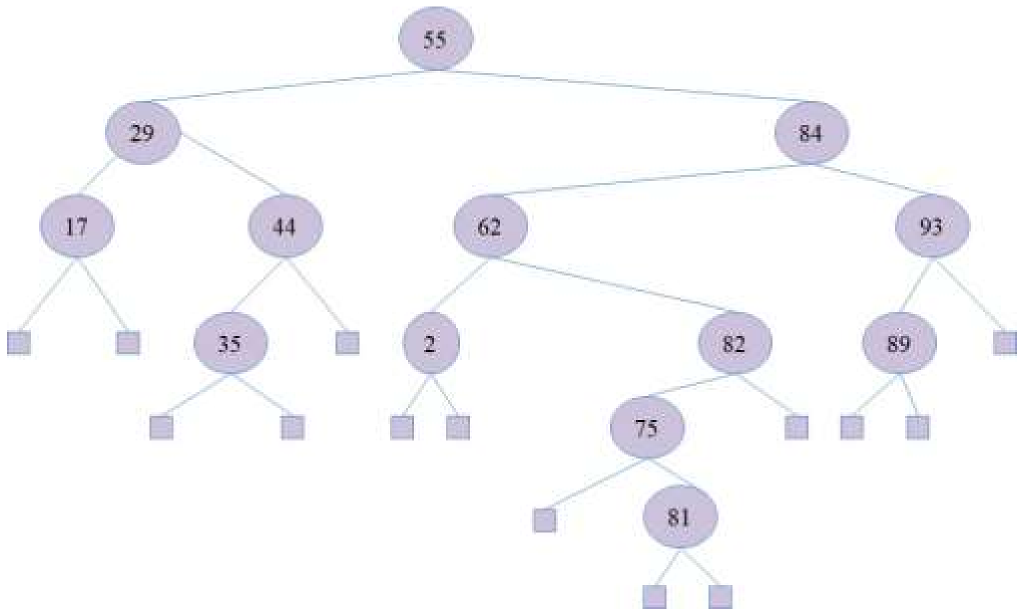
i) Show the content of the table after performing the following operations, in order:
put(25), put(12), put(42), put(31), put(35), put(39), remove(31), put(48), remove(25), put(18), put(29), put(29), put(35).

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

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

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

Question 6 Given the following tree, which is assumed to be an AVL tree:



i) Are there any errors with the tree as shown? If so, indicate what the error(s) are, correct these error(s) [you should attempt applying the smallest possible number of changes to correct the tree], show the corrected AVL tree, then proceed to the following questions (Questions ii to iv) and start with the tree that you have just corrected. If no errors are there in the above tree, indicate why the tree is correctly an AVL tree, then proceed to the following questions (Questions ii to iv) and continue working on the tree as shown above.

ii) Show the AVL tree after ***put(74)*** operation is performed. Give the complexity of this operation in terms of Big-O notation.

iii) Show the AVL tree after ***remove(62)*** is performed. Give the complexity of this operation in terms of Big-O notation.

iv) Show the AVL tree after ***remove(93)*** is performed. Show the progress of your work step-by-step. Give the complexity of this operation in terms of Big-O notation.

Part 2: Programming Question (50 marks):

CARFAX, is a commercial service that supplies vehicle history reports to individuals and businesses on used cars and light trucks for the American and Canadian consumers. It keeps records of all Vehicle Identification Number (VIN). Where each VIN is unique and consists of alphanumeric characters (e.g. 4DUSR6IQ8LN209176). The composition of Vehicle Identification Number is unique for each vehicle with maximum length is restricted to 17 alphanumeric characters. CARFAX has some lists that are local for cities and areas, where *n* counts a few hundred properties. Others are at the provincial/ state level, that is *n* counts tens of thousands or more, or even at country levels, that is *n* counts millions or more. Furthermore, it is important to have access to the vehicle history. Such a historical record for a Vehicle Identification Number should be kept in reverse chronological order (i.e. from most recent)

CARFAX, needs your help to design a Customized "Vehicle history Report listing" data structure called CVR. Keys of CVR entries are vehicle identification numbers, that are strings composed of 10-17 alphanumeric characters, and one can retrieve the key of a CVR or access a single element by its key. Furthermore, similar to sequences, given a vehicle identification number in a CVR one can access its predecessor or successor (if it exists).

CVR adapts to its usage and keeps the balance between memory and runtime requirements. For instance, if a CVR contains only a small number of entries (e.g., few hundreds), it might use less memory overhead but slower (sorting) algorithms. On the other hand, if the number of contained entries is large (greater than 1000 or even in the range of tens of thousands of elements or more), it might have a higher memory requirement but faster (sorting) algorithms. CVR might be almost constant in size or might grow and/or shrink dynamically. Ideally, operations applicable to a single CVR entry should be between O (1) and O (log n) but never worse than O(n). Operations applicable to a complete CVR should not exceed $O(n^2)$.

You are asked to **design** and **implement** CVR, a customized data structure which automatically adapts to the dynamic content that it operates on. In other words, it accepts the size (total number of Vehicle Identification Number) as a parameter and uses an appropriate (set of) data structure(s) from the one(s) studied in class in order to perform the operations below efficiently[1].

The CVR must implement the following methods:

- **setThreshold(Threshold)**: where $100 \leq \text{Threshold} \leq \sim 900{,}000$ is an integer number that defines when a listing should be implemented with a data structure such as a Tree, Hash Table, AVL tree, binary tree, if its size is greater than or equal to value of Threshold. Otherwise it is implemented as a Sequence.
- **setKeyLength(Length)**: where $10 \leq \text{Length} \leq 17$ is an integer number that defines the fixed string length of keys.
- **generate(n)**: randomly generates a sequence containing n new non-existing keys of alphanumeric characters.
- **allKeys()**: return all keys as a **sorted sequence (lexicographic order)**
- **add(key,value[2])**: add an entry for the given key and value
- **remove(key)**: remove the entry for the given key
- **getValues(key)**: return the values of the given key
- **nextKey(key)**: return the key for the successor of key.
- **prevKey(key)**: return the key for the predecessor of key
- **prevAccids(key)**: returns a sequence (sorted in reverse chronological order) of accidents(previously) registered with the given key (dates).

[1] The lower the memory and runtime requirements of the ADT and its operations, the better will be your grades.
[2] Value here could be any feature of the vehicle's report.

1. Write the pseudo code for each of the methods above.
2. Write the Java code that implements the methods above.
3. Discuss how both the time and space complexity change for each of the methods above if the underlying structure of your CVR is an array or a linked list?

You have to submit the following deliverables:

a) A detailed report about your design decisions and specification of your CVR ADT including a rationale and comments about assumptions and semantics.
b) Well-formatted and documented Java source code and the corresponding class files with the implemented algorithms.
c) Demonstrate the functionality of your CVR by documenting at least 10 different but representative data sets. These examples should demonstrate all cases of your CVR ADT functionality (e.**g., all operations of your ADT for different sizes).**

The written part must be done individually (no groups are permitted). The programming part can be done in groups of two students (maximum!).
For the written questions, submit all your answers in PDF (or text formats only). Exceptionally, images of the assignment, can be hand-drawn and scanned; however, you must make sure that they are clear and very readable. Please be concise and brief (less than ¼ of a page for each question) in your answers.
For the Java programs, you must submit the source files together with the compiled files. The solutions to all the questions should be zipped together into one .zip or .tar.gz file. You must upload at most one file (even if working in a team; please read below). In specific, here is what you need to do:

1) Create **one** zip file, containing the necessary files (.java, .doc, .html, etc.). Please name your file following this convention:
If the work is done by 1 student: Your file should be called *a#_studentID*, where # is the number of the assignment *studentID* is your student ID number.
If the work is done by 2 students: The zip file should be called *a#_studentID1_studentID2*, where # is the number of the assignment, and *studentID1* and *studentID2* are the ID numbers of each student.
2) If working in a group, only one of the team members can submit the programming part. Do not upload 2 copies.

Very Important: Again, the assignment must be submitted in the right folder of the assignments. **Assignments uploaded to an incorrect folder will not be marked and result in a zero mark. No resubmissions will be allowed.**

⇨ Additionally, for the programming part of the assignment, an **online demo maybe** required (please refer to the course outline for full details). In such case, the markers will inform you about the demo times and how it will take place online. **If an online demo is required, please notice that failing to demo your assignment will result in zero mark regardless of your submission.** If working in a team, both members of the team must be present during the online demo time. More details on these online demos will be provided to you around the due date of the assignment.