COMP 352: Data Structure and Algorithms
Fall 2019

Department of Computer Science and Software Engineering
Concordia University

<u>Combined Assignment #3 and #4</u>

<u>Due date and time:</u> Saturday November 30th 10:00 AM (morning!)

<u>Important Note:</u>
Since demos must take place during the period of Saturday, November 30 and Tuesday December 3, no deadline extension will be granted and no late assignments will be accepted (not even with a penalty!)

Part 1: Written Questions (50 marks):

Question 1

Draw a (single) tree T, such that

- Each internal node of T stores a single character;
- A preorder traversal of T yields: E K D M J G I A C F H B L;
- A postorder traversal of T yields: D J I G A M K F L B H C E.

Question 2

Given an array *A* of *n* integers, write an algorithm that finds the smallest *x* values in the array, where *x* <=*n*. However, the first constraints are given: You are not allowed to use heaps; the array A should not be modified, and you can only use a maximum auxiliary space of O(n).

Question 3

Draw the min-heap that results from running the bottom-up heap construction algorithm on the following list of values:

42 25 49 18 57 105 112 35 38 64 53 20 25 19 16.

Starting from the bottom layer, use the values from left to right as specified above. Show intermediate steps and the final tree representing the min-heap. Afterwards perform the operation removeMin() 3 times and show the resulting min-heap after each step.

Question 4

Create a max-heap using the list of values from Question 3 above but this time you have to insert these values one by one using the order from left to right (i.e insert 42, then 25, then 49, etc.). Show the tree after each step and the final tree representing the max-heap.

Question 5

Given 2 trees *T1* and *T2*, where T1 has more nodes than T2, and where each of the nodes in both trees holds an integer value. Further, the values in any of the trees are assumed to be unique (no duplicate values within the same tree).

i) Give an algorithm that determines whether or not T2 represents a subtree of T1.

ii) What is the time and space complexity of your algorithm in i)?

iii) Will your algorithm still work if duplicate values are permitted in these trees?
   a. If your algorithm still works when duplicate values are allowed, explain clearly why this change could not have affected your solution.
   b. If your algorithm will fail if duplicate values are allowed, provide an example of the failure and propose some modification to fix this failure.

Question 6

Assume the utilization of *linear probing* for hash-tables. To enhance the complexity of the operations performed on the table, a special *AVAILABLE* object is used. Assuming that all keys are positive integers, the following two techniques were suggested in order to enhance complexity:

i) In case an entry is removed, instead of marking its location as AVAILABLE, indicate the key as the negative value of the removed key (i.e. if the removed key was 16, indicate the key as -16). Searching for an entry with the removed key would then terminate once a negative value of the key is found (instead of continuing to search if AVAILABLE is used).

ii) Instead of using AVAILABLE, find a key in the table that should have been placed in the location of the removed entry, then place that key (place the entire entry of course) in that location (instead of setting the location as AVAILABLE). The motive is to find the key faster since it is now in its hashed location. This would also avoid the dependence on the AVAILABLE object.

Will either of these proposals enhance the *search* time complexity? Additionally, will any of these approaches result in misbehaviors (in terms of functionalities, or even performance)? If so, explain clearly through illustrative examples.

Question 7

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 8

To reduce the maximum number of collisions in the hash table described in Question 7 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 9

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 mod 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*?

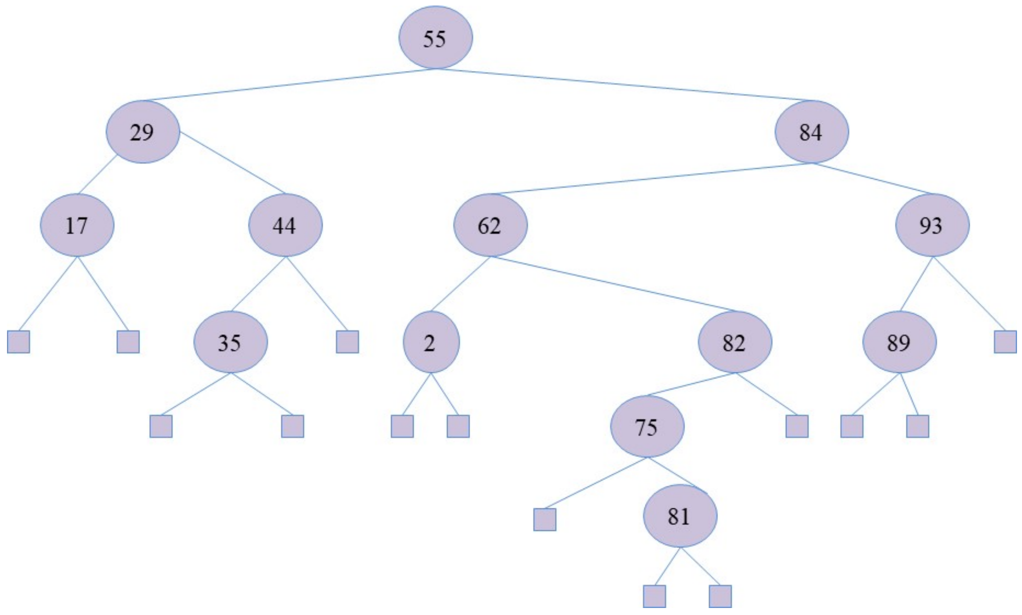
Question 10

Show the steps that a radix sort takes when sorting the following array of integer keys:

832 91 411 172 243 573 326 292 682 489 96

Question 11

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.

<u>Part 2: Programming Question (50 marks):</u>

The General Automobile Registration Office (GARO) keeps records of car registrations. It operates on multiple lists of *n* car registrations, where each registered car is identified by its unique license plate that consists of alphanumeric characters (e.g. R4G5OO54TE). The composition of license plates can be different for provinces and countries but their maximum length is restricted to 12 alphanumeric characters. Some of the lists are local for cities and areas, where *n* counts a few hundred properties. Others are at the provincial 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 history cars of that have been registered with the same license plate. Such a historical record for a license plate should be kept in reverse chronological order.

GARO needs your help to design a smart "automobile registration listing" data structure called SmartAR. Keys of SmartAR entries are strings composed of 6-12 alphanumeric characters, and one can retrieve the key of a SmartAR or access a single element by its key. Furthermore, similar to sequences, given a SmartAR element one can access its predecessor or successor (if it exists). SmartAR adapts to its usage and keeps the balance between memory and runtime requirements. For instance, if a SmartAR 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. SmartAR might be almost constant in size or might grow and/or shrink dynamically. Ideally, operations applicable to a single SmartAR entry should be between O(1) and O(log n) but never worse than O(n). Operations applicable to a complete SmartAR should not exceed O(n^2).

You are asked to **design** <u>and</u> **implement** SmartAR, a smart ADT, which automatically adapts to the dynamic content that it operates on. In other words, it accepts the size (total number of *n* car registrations identified by their key, i.e., license plate) as a parameter and uses an appropriate (set of) ADTs from the ones studied in class in order to perform the operations below efficiently[1].

The SmartAR must implement the following methods:

- **setThreshold(Threshold)**: where 100 ≤ Threshold ≤ ~500,000 is an integer number that defines how a listing should be implemented. If the listing size is larger than or equal to that threshold value, then you need to use a data type such as a Tree, a Hash Table, an AVL tree, or a Binary Tree. If the size is smaller than this threshold, then it needs to be implemented as a Sequence.

- **setKeyLength(Length)**: where 6 ≤ Length ≤ 12 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).**

[1] The lower the memory and runtime requirements of the ADT and its operations, the better your mark will be.

- **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.
- **previousCars(key)**: returns a sequence (sorted in reverse chronological order) of cars (previously) registered with the given key (license plate).

1. Write the Java code that implements the methods above.
2. Discuss how both the time and space complexity change for each of the methods above if the underlying structure of your SmartAR 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 SmartAR 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 SmartAR by documenting at least 10 different but representative data sets. These examples should demonstrate all cases of your SmartAR ADT functionality (e.g**., all operations of your ADT for different sizes).** You have to additionally test your implementation with benchmark files containing a sequence of keys (one key per line without values) that is posted along with this assignment.

<u>**Important Notes**</u>

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). Your assignment <u>must</u> be typed; <u>however, you are allowed to provide hand-drawn images if you wish</u>. Please be concise and brief (less than ¼ of a page for each question) in your answers. Submit the assignment under Theory Assignment 2 directory in EAS or the correct Dropbox/folder in Moodle (depending on your section).

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 and

[2] Value here could be any feature of the car or its owner.

submitted via EAS under Programming 1 directory or under the correct Dropbox/folder in Moodle. 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. Depending on your section, you will either upload to EAS or to Moodle (your instructor will indicate which one to use). **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, a demo is required (please refer to the courser outline for full details). The marker will inform you about the demo times. **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 demo.