Department of Computer Science and Software Engineering
Concordia University

COMP 352: Data Structure and Algorithms
Fall 2014

<u>Combined Assignments 3 & 4</u>

<u>Due date and time:</u> Sunday November 23rd 11:59:59 PM

<u>Important Note:</u> Since solutions of these assignments will be discussed during the tutorials of the last week of classes (week of November 24), no late assignments will be accepted (not even with a penalty!)

Part 1: Written Questions (60 marks):

<u>Question 1</u>

Draw a (single) binary tree T, such that

- Each internal node of T stores a single character;
- A preorder traversal of T yields JLFEYLYKTSEUUSV;
- An inorder traversal of T yields EFYLYLKJUEUSSTV;
- A postorder traversal of T yields EYFYKLLUUESSVTJ.

<u>Question 2</u>

Assume that the binary tree from Question 1 above is stored in an array-list as a complete binary tree as discussed in class and in §7.3.5 of the textbook. Specify the contents of such an array-list for this tree.

<u>Question 3</u>

Draw the min-heap that results from running the bottom-up heap construction algorithm (from Section §8.3.6 of the textbook and as described in class) on the following list of values: 24 13 30 2 27 88 94 17 19 53 28 9 10 6 48. 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.

<u>Question 4</u>

Create again a min-heap using the list of values from Question 3 above but this time you have to insert these values step by step using the order from left to right as shown in Question 3. Show the tree after each step and the final tree representing the min-heap.

<u>Question 5</u>

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 (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 now in its hashed location. This would also avoid the dependence on the AVAILABLE object.

Will either of these proposal have an advantage of the achieved complexity? You should analyze both time-complexity and space-complexity. Additionally, will any of these approaches result in misbehaviors (in terms of functionalities)? If so, explain clearly through illustrative examples.

Question 6

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 7

To reduce the maximum number of collisions in the hash table described in Question 6 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 8

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 is = 11. 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(38), put(15), put(43), put(22), put(71), put(8), put(28), put(37), put(19).

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 9

Assume the utilization of *linear probing* instead of *double hashing* for the above implementation given in Question 8. Still, the size of the array $N = 19$, and that simple modular operation (***k* mod 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.

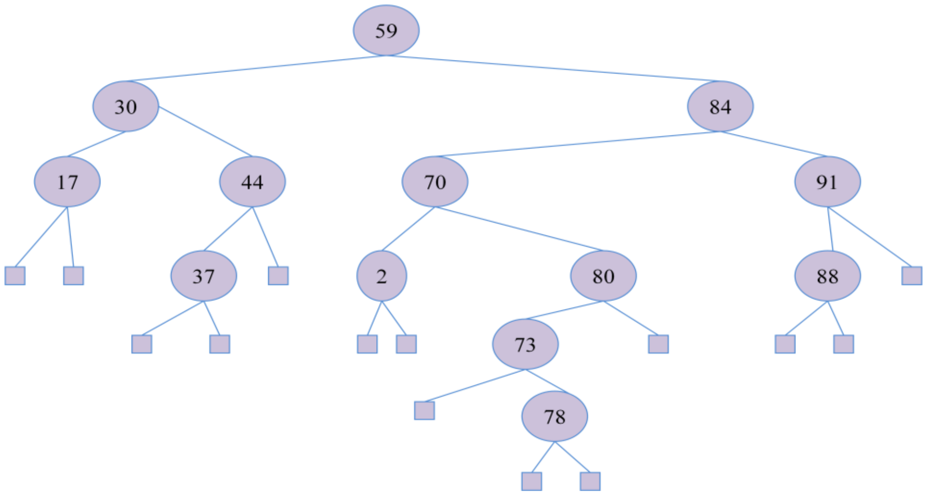
i) Show the contents of the table after performing the following operations, in order: **put(29), put(53), put(14), put(95), remove(53), remove(29), put(32), put(19), remove(14), put(30), put(12), put(72).**

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

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

Question 10

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), 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(70)*** is performed. Give the complexity of this operation in terms of Big-O notation.

iv) Show the AVL tree after ***remove(91)*** 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 (40 marks):

In class, we discussed min-heaps. In a min-heap the element of the heap with the smallest key is the root of a binary tree. A max-heap has as root always the element with the biggest key and the relationship between the keys of a node and its parent is less than or equal (≤).

Your task is to develop your own binary tree ADT and your own ***flex-heap*** ADT. The flex-heap ADT must implement a min- as well as a max-heap. Apparently, instead of defining a removeMin or removeMax operation you will only provide a remove operation in your flex-heap. It must be implemented using your binary tree ADT. You have to implement these two ADTs in Java. The flex-heap ADT has to additionally support the dynamic switch from a min- to a max-heap and vice versa:

- **remove()** removes and returns the element with the smallest or biggest key value depending on the heap status (min-heap vs. max-heap) and repairs the flex-heap afterwards accordingly.
- **toggleHeap()** transforms a min- to a max-heap or vice versa.
- **switchMinHeap()** transforms a max- to a min-heap.
- **switchMaxHeap()** transforms a min- to a max-heap.

Binary trees must be implemented with an extendable array-list similar to what we discussed in class and in §7.3.5 of the textbook. You are not allowed to implement trees with lists. Further, you are not allowed to use any array-list, queue, vector, (binary) tree, or heap interfaces already available in Java. Your toggleHeap, switchMinHeap, and switchMaxHeap operations must run in O(n) time. All other flex-heap operations must be either in O(1) or O(log n). You may safely assume for the binary tree and flex-heap ADT that keys are of type integer and values are of type character. So, the use of generics is not required.

<u>You have to submit the following deliverables:</u>

a) Specification of the binary tree and flex-heap ADTs including comments about assumptions and semantics (especially about the 3 added flex-heap operations).
b) Pseudo code of your implementation of the binary tree and flex-heap ADTs. Keep in mind that Java code will not be considered as pseudo code. Your pseudo code must be on a higher and more abstract level.
c) Well-formatted and documented Java source code and the corresponding executable jar file with at least 20 different but representative examples demonstrating the functionality of your implemented ADTs. These examples should demonstrate all cases of your ADT functionality (e.g., all operations of your ADTs, sufficient sizes of flex-heaps, sufficient number of down and up-heap, toggleHeap, switchMinHeap, and switchMaxHeap operations). You must have separate tests for each ADT but the majority of tests should cover flex-heaps because they are implemented using binary trees.

All submission must be made via EAS. Submit all your answers to written questions in PDF (no scans of handwriting) or text formats only (PDF is preferable). 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 executables. All files must be zipped together into one .zip or .tar.gz file. You may upload at most one file to EAS for each of the two parts of the assignment.

For the programming component, only one submission is to be made per group (≤ 2 students). In general, you must make sure that you upload the assignment to the correct directory of <u>Assignment 3</u> using EAS. Assignments uploaded to the wrong directory will be discarded and no re-submission will be allowed.