Computer Science CSC263H
St. George Campus

January 17, 2019
University of Toronto

Homework Assignment #2

Due: January 31, 2019, by 5:30 pm

- You must submit your assignment as a PDF file, named **a2.pdf**, of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  https://markus.teach.cs.toronto.edu/csc263-2019-01

  To work with one or two partners, you and your partner(s) must form a group on MarkUs.
- The **a2.pdf** PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTeX typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can use other typesetting systems if you prefer, but handwritten documents are not accepted.
- If this assignment is submitted by a group of two or three students, the **a2.pdf** PDF file that you submit should contain for each assignment question:

  1. The name(s) of the student(s) who *wrote* the solution to this question, and
  2. The name(s) of the student(s) who *read* this solution to verify its clarity and correctness.

- By virtue of submitting this assignment you (and your partners, if you have any) acknowledge that you are aware of the homework collaboration policy that is stated in the csc263 course web page: http://www.cs.toronto.edu/~sam/teaching/263/#HomeworkCollaboration.
- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.
- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.
- Your **a2.pdf** submission should be no more than 4 pages long in a 10pt font.

Question 1. (1 marks)

In this question, H denotes a *binomial max heap*, n is the number of items in H, x is (a pointer to the node of) an item inside H, and k is a number (key).

a. Describe a *simple* algorithm to *increase* the key of a given item x in a binomial max heap H to become k. Your algorithm should not change anything if $k \leq x.key$. The worst-case running-time of your algorithm must be $O(\log n)$. Give a high-level description of your algorithm in clear English.

b. Using part (a), describe a *simple* algorithm to *delete* a given item x from a binomial max heap H. The worst-case running-time of your algorithm must be $O(\log n)$. Give a high-level description of your algorithm in clear English.

Question 2. (1 marks)

Design a data structure called *SuperHeap* that supports the following operations:

- Insert(k): inserts the key k into the SuperHeap,
- ExtractMax(): removes a max key from the SuperHeap,
- ExtractMin(): removes a min key from the SuperHeap,
- Merge(D,D'): merges SuperHeaps D and D' into one SuperHeap.

The worst-case running-time of each operation in your data structure must be $O(\log n)$ where n is the total number of items. *Your data structure must be based on a data structure that we discussed in this course.*

1. Explain the main high-level idea of your solution in clear English.

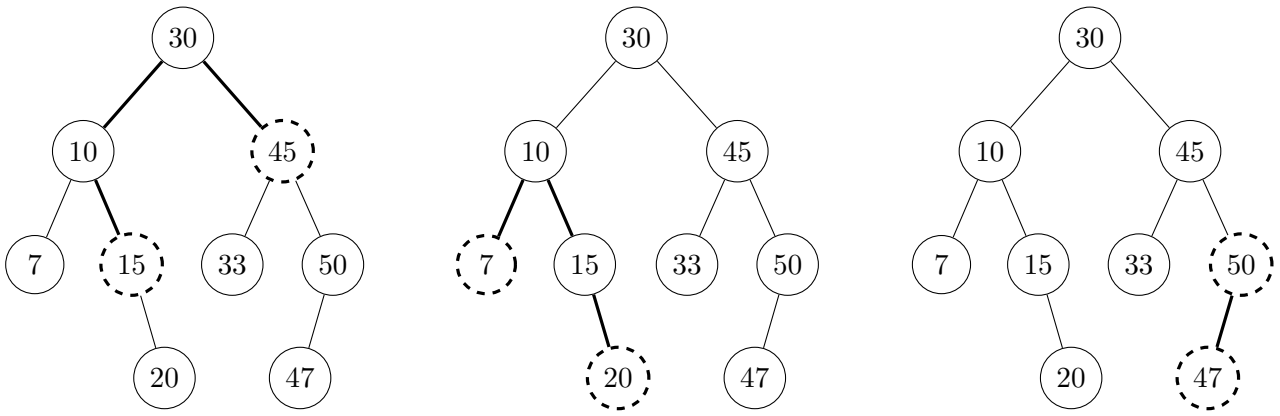
   As part of this explanation, (1) state what is the underlying data structure that you are using in your solution, and (2) explain clearly all the information that you are *adding* to this underlying data structure. Specifically, what additional information are you storing in each node?

2. Give a high-level description in clear English of how to implement each operation of the SuperHeap.

HINT: Use binomial heaps and your solution to Question 1.

Question 3. (1 marks)

Two nodes in a Binary Search Tree (BST) are said to be $t-away$, if *the length of the path* between the two nodes is at most t. A *path* between two nodes u, v in a tree, is a sequence of edges connecting a sequence of *adjacent* nodes in the tree, where the starting node in the sequence is u and the ending node is v; *the length of a path* is the number of edges in that path. Two distinct nodes u, v are said to *adjacent* if either u is the parent of v or v is the parent of u. For example, the figure below shows the path between 15 and 45 (length 3), the path between 7 and 20 (length 3), and the path between 47 and 50 (length 1) in a BST.



To derive an algorithm to check whether two nodes of a BST are $t-away$, solve the three subquestions outlined below.

Henceforth assume that *root* is not NIL and the BST rooted at *root* does not have duplicate keys. Morever, each node u of the BST has the following fields: $key(u)$, containing the key of the node, $lchild(u)$ and $rchild(u)$, containing pointers to u's left and right children respectively; note that node u does ***not*** have a pointer to its parent. For a key k in the BST, let $node(k)$ be the BST node with key k.

Give ***pseudocode*** and a ***concise and clear*** English description of your algorithm for each of the following subquestions. Also give a brief explanation of why your algorithm achieves the worst-case time complexity specified in that subquestion (where h is the height of the BST rooted at *root*).

a. Give an efficient algorithm for the following procedure.

PATHLENGTHFROMROOT($root, k$): Given the *root* of a BST and a key k, return the length of the path between *root* and $node(k)$. Assume that the key k is in the BST.

For example, if *root* is the root of the BST in **Figure 1**, then PATHLENGTHFROMROOT(*root*, 15) should return 2, and PATHLENGTHFROMROOT(*root*, 47) should return 3.

The worst-case time complexity of your algorithm should be $O(h)$.

b. Given the *root* of a BST and two distinct keys k, m present in the BST, define the *FCP of k and m* in the BST rooted at *root*, to be the root of the subtree that is furthest away from *root* which contains both k and m. In other words, the FCP of k and m is a node *parent* such that: (a) the subtree rooted at *parent* has both the keys k and m in it, and (b) the length of the path between *root* and *parent* is the maximum among all such *parent*s. Give an efficient algorithm for the following procedure.

FCP($root, k, m$): Given the *root* of a BST and two distinct keys k and m, return the FCP of k and m in the BST rooted at *root*. Assume that both k and m are present in the BST.

For example, if *root* is the root of the BST in **Figure 1**, then FCP(*root*, 15, 45) should return the node with key 30, FCP(*root*, 7, 20) should return the node with key 10, and FCP(*root*, 50, 47) should return the node with key 50.

The worst-case time complexity of your algorithm should be $O(h)$.

c. Give an efficient algorithm for the following procedure.

ISTAWAY($root, k, m, t$): Given the *root* of a BST, two distinct keys k and m, and a non-negative integer t, return ***true*** if the length of the path between $node(k)$ and $node(m)$ is *at most* t, and ***false*** otherwise. Assume that k and m are present in the BST.

For example, if *root* is the root of the BST in **Figure 1**, then ISTAWAY(*root*, 15, 45, 3) should return ***true***, and ISTAWAY(*root*, 7, 20, 2) should return ***false***.

The worst-case time complexity of your algorithm should be $O(h)$.

HINT: Use the procedures from Parts **a** and **b**.

[***The questions below will not be corrected/graded. They are given here as interesting problems that use material that you learned in class.***]

Question 4. (0 marks)

This question is about the cost of successively inserting k elements into a binomial heap of size n.

a. Prove that a binomial heap with n elements has exactly $n - \alpha(n)$ edges, where $\alpha(n)$ is the number of 1's in the binary representation of n.

b. Consider the worst-case total cost of successively inserting k new elements into a binomial heap H of size $|H| = n$. In this question, we measure the worst-case cost of inserting a new element into H as the maximum number of pairwise comparisons between the keys of the binomial heap that is required to do this insertion. It is clear that for $k = 1$ (i.e., inserting one element) the worst-case cost is $O(\log n)$. Show that when $k > \log n$, the *average* cost of an insertion, i.e., the worst-case total cost of the k successive insertions divided by k, is bounded above by constant.

Hint: Note that the cost of each one of the k consecutive insertions varies — some can be expensive, other are cheaper. Relate the cost of each insertion, i.e., the number of key comparisons that it requires, with the number of extra edges that it forms in H. Then use part (a).

Question 5. (0 marks)

A Binary Search Tree (BST) T is an *AVL tree* if, for every node v of T, the heights of the left and right subtrees of v differ by at most 1, i.e., if $|\textsc{Height}(\mathit{lchild}(v)) - \textsc{Height}(\mathit{rchild}(v))| \leq 1$ (where the height of an *empty* subtree is defined to be -1).

Give a linear-time algorithm that determines if a Binary Search Tree (BST) is an AVL tree.

The algorithm's input is a pointer u to the root of a BST T where each node v has the following fields: an integer *key*, and pointers *parent*, *lchild* and *rchild* to the parent, the left and right children of v in T (any unused pointer is set to Nil). The algorithm's output should be True if T is an AVL tree, and False otherwise.

The worst-case running time of your algorithm ***must be*** $\Theta(n)$ where n is the number of nodes in T.

Describe your algorithm by giving its ***pseudo-code***.

Question 6. (0 marks)

In the following, B_1 and B_2 are two binary search trees such that every key in B_1 is smaller than every key in B_2.

Describe an algorithm that, given pointers b_1 and b_2 to the roots of B_1 and B_2, merges B_1 and B_2 into a single binary search tree T. Your algorithm should satisfy the following two properties:

1. Its worst–case running time is $O(min\{h_1, h_2\})$, where h_1 and h_2 are the heights of B_1 and B_2.
2. The height of the merged tree T is at most $max\{h_1, h_2\} + 1$.

Note that the heights h_1 and h_2 are ***not*** given to the algorithm (in other words, the algorithm does not "know" the heights of B_1 and B_2). Note also that B_1, B_2 and T are ***not*** required to be balanced.

Describe your algorithm, and justify its correctness and worst-case running time, in ***clear and concise*** English.

Hint: First derive an algorithm that runs in $O(max\{h_1, h_2\})$ time, and then optimize it.