Foundations of Computer Science (Semester 2) – 2015

Assessed Exercise Sheet 4 – 10% of Continuous Assessment Mark

Deadline : 11pm Sunday 15th February, via Canvas

Question 1 (22 marks)

In the lecture notes (section 2.2) we saw an *O(n)* procedure `append(l1,l2)` that returns the combined list made up of list l1 followed by list l2. If we had an improved data structure for lists that had pointers to both the first and last items in the list (like in a queue as discussed in lecture notes section 2.4), it would be possible create a much more efficient procedure `append2(l1,l2)` to combine lists. State in words (not pseudocode) a sequence of operations that would make `append2(l1,l2)` work in constant time.

Write a procedure `bst2list(bst)` in pseudocode (not words) that converts a binary search tree into a linked list of nodes in ascending order. You can call any of the standard primitive operators for lists and trees, and also your new procedure `append2(l1,l2)`. [Hint: Consider how this task relates to the procedure `printInOrder(t)` from the lecture notes section 6. 10.]

What is the total time complexity of this algorithm?

Question 2 (20 marks)

In the lecture notes (section 6.8), a procedure `isbst(t)` is defined that returns `true` if `t` is a binary search tree and `false` if it is not.

Explain why that algorithm is not efficient.

Derive an expression for the number of comparisons *C(h)* this algorithm requires for a full binary search tree of height *h*? [Hint: It is easiest to write *C(h)* as the sum of the number of comparisons at each level, and then sum that series using the result from the lectures that the number of nodes *n* in a full tree of height *h* is its size $s(h) = 2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$. To get the individual terms in the sum, think how many nodes are there at each level *i* in the tree, and how many nodes further down the tree each of them is compared to.]

Consider how *C(h)* varies for large *h* and comment on the complexity of this algorithm in terms of the size of the tree.

Question 3 (20 marks)

Using the `bst2list(bst)` procedure from Question 1, and any of the standard primitive list and tree operators, write an improved procedure `isbst2(t)` that performs the same task as `isbst(t)` but more efficiently. [Hint: You may find it most straightforward to make `isbst2(t)` a non-recursive procedure that calls a separate recursive procedure.]

What is the complexity of your improved procedure?

Question 4 (20 marks)

Explain what is meant by a *tree rotation* and why it might be useful to perform one. [Hint: This is the kind of "bookwork" question you can expect in the exam. The answers are in the lecture notes, but it is worth practicing writing clear and concise answers from memory.]

Draw the binary search tree that results from inserting the items [20, 30, 35, 10, 40, 25] in that order into an initially empty tree, and explain how a tree rotation can usefully be applied to that tree.

Question 5 (18 marks)

Write a recursive procedure `isHeap(t)` that returns `true` if the binary tree `t` is a heap tree, and `false` if it is not. You can call any of the standard primitive binary tree procedures `isEmpty(t)`, `root(t)`, `left(t)` and `right(t)`, and also a procedure `complete(t)` that returns `true` if `t` is complete, and `false` if it is not.

What can be said about the overall complexity of your algorithm?