# Data Structures and Algorithms – 2018

## Assignment 2 – 25% of Continuous Assessment Mark

## Deadline : 5pm Monday 12th February, via Canvas

**Question 1 (22 marks)**

It is sometimes useful to treat collections of items as *sets* upon which standard set theory operations (like membership, subset, intersection, and union) can be applied. A convenient representation of a *set* is as a list in which repeated items are not allowed and the order of the items does not matter.

Suppose you have sets `S1` and `S2` represented as linked-lists, and access to the standard primitive list operators `first`, `rest` and `isEmpty`. Write a recursive procedure `member(x,S1)` that returns `true` if item `x` is in set `S1`, and `false` if it is not.

Provide an argument that leads to the average-case and worst-case time complexities of your `member(x,S1)` algorithm in *Big O* notation.

Now write a recursive procedure `subset(S1,S2)` that returns `true` if set `S1` is a subset of set `S2`, and `false` if it is not. It is only allowed to call the standard primitive list operators `first`, `rest` and `isEmpty` and your `member` procedure.

Finally, write a recursive procedure `union(S1,S2)` that returns the union of sets `S1` and `S2` represented as linked-lists. It is only allowed to call the standard primitive list operators `makelist`, `first`, `rest` and `isEmpty` and your `member` procedure.

**Question 2 (14 marks)**

Draw the binary search tree that results from inserting the items [19, 30, 36, 10, 40, 25, 33] in that order into an initially empty tree.

Show how the tree rotation approach in the Lecture Notes (Section 7.10) can be used to balance that tree.

Draw tree that results from deleting the item 30 from your balanced tree using the `delete` algorithm in the Lecture Notes (Section 7.7).

**Question 3 (16 marks)**

In the Lecture Notes (Section 7.8), a simple procedure `isbst(t)` was defined that returns `true` if `t` is a binary search tree and `false` if it is not. Explain why that algorithm is not efficient, and illustrate your explanation with a simple example.

Derive an expression for the number of value 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 there are at each level *i* in the tree, and how many nodes further down the tree each of them is compared to.]

Deduce the time complexity of this algorithm in terms of the size *n* of the tree. [Hint: Derive approximate expressions for the size *n* and complexity *C* as functions of large *h*, and hence obtain an approximate expression for the complexity *C* as functions of large *n*.]

**Question 4 (16 marks)**

Suppose you already have an *O(n)* procedure `tree2list(t)` that takes a binary tree `t` and returns a linked list of node values in such a way that they will be in ascending order if `t` is a binary search tree (e.g., using an approach similar to that described in the Lecture Notes Section 7.9). Use that procedure, and any of the standard primitive list and tree operators, to write a 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 are the average-case and worst-case time complexities of your improved procedure?

What scope is there to develop even more efficient procedures?

**Question 5 (12 marks)**

The 15 numbers {1, 2, 3, …, 15} can be stored in a perfectly balanced binary search tree of height 3. Draw that tree.

Suppose the determination of "bigger, equal or smaller" is counted as one "comparison". For each of the above 15 numbers, state how many comparisons are required to determine whether that number is in the search tree.

Compute the total number of comparisons *C* required to search for all 15 numbers, and hence find the average number of comparisons *A* required to search for one number.

**Question 6 (20 marks)**

Derive expressions for *C(h)* and *A(h)* which generalize the total and average number of comparisons computed in Question 5 to the case of a full binary search tree of any height *h*.

[Hint: This question is quite challenging! If your mathematics is strong, it is possible to write *C(h)* as the sum of the number of comparisons at each level, and then simplify that using the usual algebraic tricks for summing series. Or, you may find it easier to proceed in the same way as in Lecture Notes Section 6.5 where an expression was derived for the tree size *s(h)*, i.e. the total number of nodes in a full binary tree of height *h*. First think about how *C(h+1)* is related to *C(h)* by adding in the extra row *h*+1, then think about how the height *h*+1 tree's *C(h+1)* is related to its two sub-tree's *C(h)*, and finally combine the two equations you get for *C(h+1)* to give an equation for *C(h)*. You can use the expression for *s(h)* from the Lecture Notes or Question 3. Whichever approach you use, show all the steps involved.]

What can you deduce about the average number of comparisons required, and hence the average time complexity of the search, for large trees?