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.

1. Replace the empty list pointer associated with the last item in the first list by the pointer to the first item in the second list.
2. Create a new pair of pointers for the combined list made up of the pointer to the first item in the first list and the pointer to the last item of the second list.
3. Delete the pointers to the first and last items of the two original lists.

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.]

```
bst2list(bst) {
   if( isEmpty(bst) )
      return EmptyList
   else return append2(bst2list(left(bst)),
                  makeList(root(bst),bst2list(right(bst))))
}
```

What is the total time complexity of this algorithm?

The procedure `bst2list` is called twice for each node in the tree. The procedures `makeList` and `append2` both have constant time complexity. So the overall time complexity is *O(n)*, where *n* is the number nodes n in the tree.

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.

It is not efficient because many redundant checks are made, i.e. many conditions are checked more than once. For example, once the algorithm has already checked that the left sub-tree is a binary search tree, it is sufficient to check the biggest element in that sub-tree is smaller than the root node, whereas the algorithm actually checks that all the nodes in the sub-tree are smaller.

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.]

At each level *i* there are 2^i nodes, and each of those are compared to all the other nodes in the sub-tree of size *h-i* that it is the root of, i.e. $s(h-i)-1 = 2^{h-i-1}-2$ of them. So the total number of comparisons is

$$C(h) = \sum_{i=0}^{h} 2^i.(2^{h-i+1} - 2)$$

$$C(h) = 2^{h+1}.\sum_{i=0}^{h} 1 - 2.\sum_{i=0}^{h} 2^i$$

$$C(h) = 2^{h+1}.(h+1) - 2.(2^{h+1} - 1)$$

$$C(h) = (h-1).2^{h+1} + 2$$

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

For large *h* we have $C(h) \sim 2^{h+1}.h \sim n \log_2 n$, so the complexity of the algorithm is $O(n \log n)$, where *n* is the number of nodes in 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.]

If `t` is a binary search tree, `bst2list(t)` will produce a list of nodes in ascending order. If `t` is not a binary search tree, `bst2list(t)` will produce a list that is not in ascending order. So, we can check whether a tree `t` is a BST by first using `bst2list(t)` to convert it into a list and then using a separate recursive procedure to check whether the order is ascending:

```
isbst2(t) {
   if( isEmpty(t) )
      return true
   return checkascend(bst2list(t))
}
```

```
checkascend(tlist) {
   if( isEmpty(rest(tlist)) )
      return true
   elseif( first(tlist) > first(rest(tlist)) )
      return false
   else return checkascend(rest(tlist))
}
```

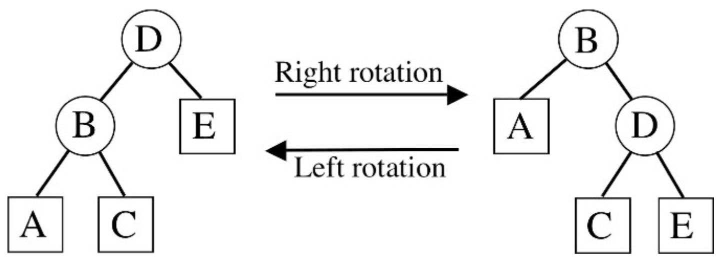
What is the complexity of your improved procedure?

Converting the tree to a list is *O(n)*, and checking whether the order is ascending is also *O(n)*, so the procedure as a whole is *O(n),* where *n* is the number of nodes in the tree.

Question 4 (20 marks)

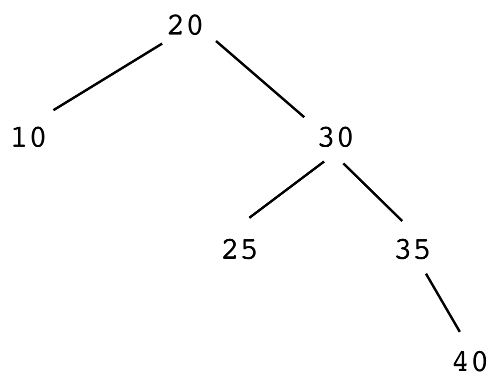
Explain what is meant by a *tree rotation* and why it might be useful to perform one. [Hint: This is the kind of "bookwork" question you can expect in the exam. The answers are in the lecture notes, but it is worth practicing writing clear and concise answers from memory.]

A tree rotation is an operation that one can perform on a binary search tree to render it more balanced, without losing its required node value ordering. It will be useful because a more balanced tree will have reduced height, and that makes various useful algorithms more efficient on it. The classic form of tree rotation is:

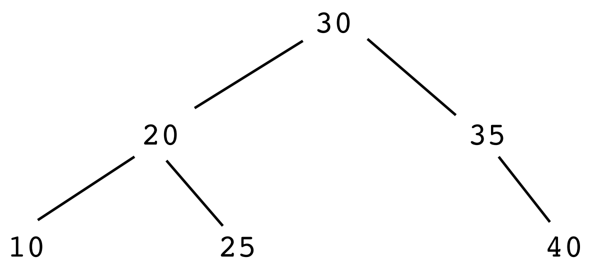


where B and D are nodes and A, C and E are any binary search sub-trees (any of which may be empty). Typically, a sequence of such tree rotations will be required to balance an arbitrary tree.

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.



To balance that tree we need a left rotation, so in this case the above general form has A = 10, B = 20, C = 25, D = 30, E = [35, 40], and we end up with:



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.

```
isHeap(t) {
   if( isEmpty(t) )
      return true
   if( not complete(t) )
      return false
   if( not isEmpty(left(t)) )
      if( root(t) < root(left(t)) )
         return false
   if( not isEmpty(right(t)) )
      if( root(t) < root(right(t)) )
         return false
   return ( isHeap(left(t)) and isHeap(right(t)) )
}
```

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

Since we are not told the complexity of the procedure `complete(t)`, it is impossible to say what the overall complexity of the algorithm is.