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.

```
member(x,S1) {
  if ( isEmpty(S1) )
     return false
  elseif ( x == first(S1) )
     return true
  else return member(x,rest(S1))
}
```

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

If set S1 contains n items including x, the recursive `member(x,S1)` procedure will on average need to check n/2 items, and in the worst case check all n items, before returning true. If S1 does not include x, the procedure will have to check all n items before returning false. So there are O(n) checks in any case, meaning that the average-case and worst-case time complexities are both O(n).

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.

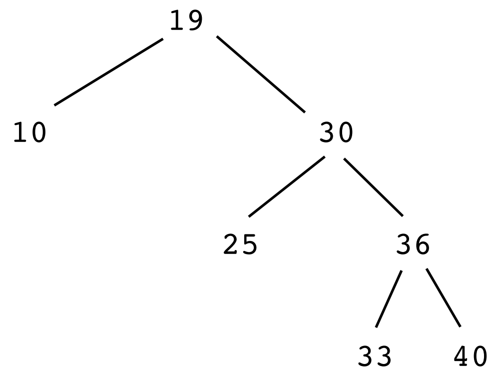
```
subset(S1,S2) {
  if ( isEmpty(S1) )
     return true
  elseif ( !member(first(S1),S2) )
     return false
  else return subset(rest(S1),S2)
}
```

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.

```
union(S1,S2) {
   if ( isEmpty(S1) )
      return S2
   elseif ( member(first(S1),S2) )
      return union(rest(S1),S2)
   else return MakeList(first(S1),union(rest(S1),S2))
}
```

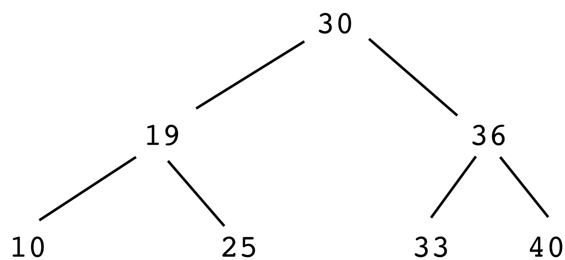
Question 2 (14 marks)

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.

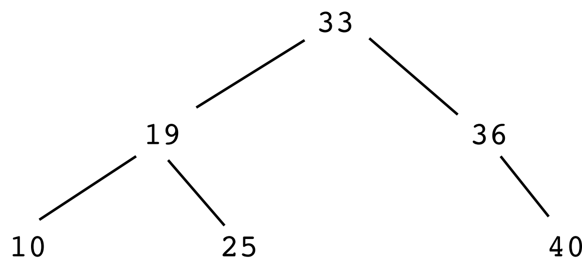


Show how the tree rotation approach in the Lecture Notes (Section 7.10) can be used to balance that tree.

To balance that tree we need a left rotation, so in this case the above general form from the Lecture Notes has A = 10, B = 19, C = 25, D = 30, E = [36, 33, 40], and we end up with:



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.

It is not efficient because many redundant checks or comparisons 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 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.]

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

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

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

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

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

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.]

For large h we have $n = s(h) \sim 2^{h+1}$ and $h \sim \log_2 n$, so $C(h) \sim 2^{h+1}.h \sim n \log_2 n$, and the time complexity of the algorithm is proportional to $C(h)$, so it is $O(n \log n)$.

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.]

The procedure `tree2list(t)` will produce a list that is in ascending order if, and only if, `t` is a BST. So, we can check whether a tree `t` is a BST by first using `tree2list(t)` to convert it into a list and then using a separate recursive procedure to check whether the list order is ascending:

```
isbst2(t) {
   if( isEmpty(t) )
      return true
   else return checkascend(tree2list(t))
}
checkascend(tlist) {
   if( isEmpty(rest(tlist)) )
      return true
   elseif( first(tlist) > first(rest(tlist)) )
      return false
   else return checkascend(rest(tlist))
}
```

What are the average-case and worst-case time complexities of your improved procedure?

Converting the tree to a list is always $O(n)$, and checking whether the order is ascending is also always $O(n)$, where n is the number of nodes in the tree. So the average and worst case time complexities are both $O(n)$.

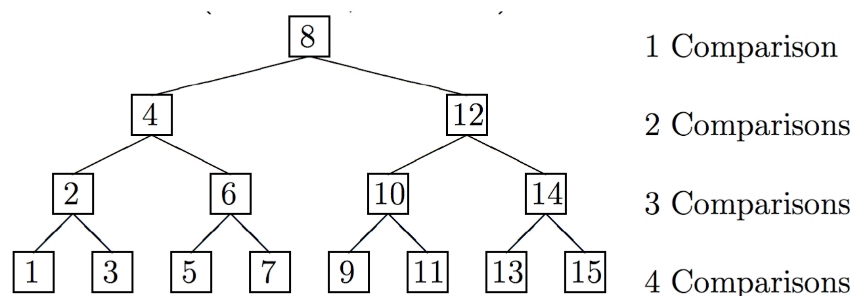
What scope is there to develop even more efficient procedures?

Ultimately, the position of all n node values needs to be checked, so doing better than the $O(n)$ complexity here will be impossible. However, it is likely to be quicker to traverse the tree in the same order as `tree2list(t)` and check immediately for ascending values instead of outputting to a list first.

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.

$$C = (1 \times 1) + (2 \times 2) + (4 \times 3) + (8 \times 4) = 49$$

$$A = 49/15 \approx 3.267$$

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.]

Algebraic summation of series solution:

Level i contains 2^i nodes each requiring (i+1) comparisons, so that level requires $2^i.(i+1)$ comparisons, Summing that over all levels gives the total number of comparisons:

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

Summing this series algebraically can be done in various ways. Probably the simplest is to write out the series, then subtract from it from that series multiplied by 2, then use the known series for the number of nodes in the tree $s(h) = 2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$, and finally simplify the resulting expression:

$$C(h) = 2^0.1 + 2^1.2 + 2^2.3 + 2^3.4 + \ldots + 2^h.(h+1)$$

$$2.C(h) = \quad 2^1.1 + 2^2.2 + 2^3.3 + \ldots + 2^h.h + \ 2^{h+1}.(h+1)$$

$$-C(h) = \ 2^0 \ + \ 2^1 \ + \ 2^2 \ + \ 2^3 \ + \ldots + \ 2^h \ - \ 2^{h+1}.(h+1)$$

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

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

Two expression for $C(h+1)$ solution:

Level i contains $2^i.(i+1)$ comparisons as above, so level h+1 contains $2^{h+1}.(h+2)$ comparisons. Adding that to the $C(h)$ comparisons of the height h tree gives:

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

The two height h sub-trees within the height $h+1$ tree would each have $C(h)$ comparisons if

the height h+1 tree's root node were not there. But that root node is there, so each of the $s(h+1)$ nodes in the whole tree has one more comparison due to that root node, so:

$$C(h+1) = 2.C(h) + s(h+1).1$$

Then subtracting one expression for $C(h+1)$ from the other gives:

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

From the lecture notes or Question 3, or a similar proof, we have:

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

so substituting that into the expression for $C(h)$ gives

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

which, not surprisingly, is the same expression obtained by the other approach.

Compute the average:

Simply divide $C(h)$ by the total number of nodes $s(h)$ to give the average

$$A(h) = \frac{C(h)}{s(h)} = \frac{h.2^{h+1} + 1}{2^{h+1} - 1}$$

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

For large trees $A(h) \sim h \sim \log_2 s(h)$, so the searching has linear average time complexity in terms of the height of the tree, and logarithmic average time complexity in terms of the size of the tree.