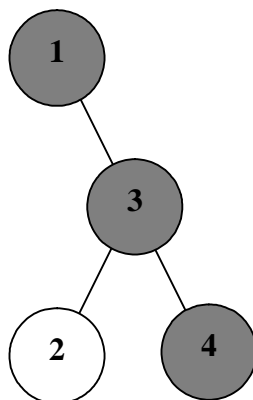


CSE3358 Problem Set 7 Solution

Problem 1: A remarkable property of BST (10 points)

Professor R. E. Mark Cable thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A , the keys to the left of the search path; B , the keys on the search path; and C , the keys to the right of the search path. Professor R. E. Mark Cable claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counter example to the professor's claim.

ANSWER: Here's a smallest counter example:



If we perform $\text{SEARCH}(\text{root}[T], 4)$, the search path will end up in a leaf and will consist of the gray nodes above. Obviously, 2 is to the left of the search path, but it is not \leq to all keys on the search path, namely $2 > 1$.

Problem 2: Height and average depth (30 points)

Consider a randomly built binary search tree, i.e. distinct keys are inserted into an initially empty binary search tree in random order. It is a fact that a randomly built binary search tree on n nodes has an expected height $\Theta(\log n)$. In class we argued this fact by drawing a similarity to the recursive tree of Randomized Quicksort (see your notes).

A common mistake is to claim the above by showing that the expected average depth in the binary search tree is $\Theta(\log n)$, where the average depth is given by $\frac{\sum_{i=1}^n \text{depth}(x_i)}{n}$, because that's easier to prove.

This problem is designed to expose this common mistake.

Let $P(n) = \sum_{i=1}^n \text{depth}(x_i)$ for a binary search tree on n nodes. Therefore, the average depth in the binary search tree is $\frac{P(n)}{n}$.

(a) (5 points) Show that $P(n)$ obeys the following recurrence for a **complete** binary search tree:

$$P(n) = 2P(n/2) + \Theta(n)$$

and conclude that the average depth in a complete binary search tree is $\Theta(\log n)$ by solving for $P(n)$.

ANSWER: Let L and R be the left and right subtrees of the root respectively. Consider a tree having k nodes in L and, therefore, $n - 1 - k$ nodes in R .

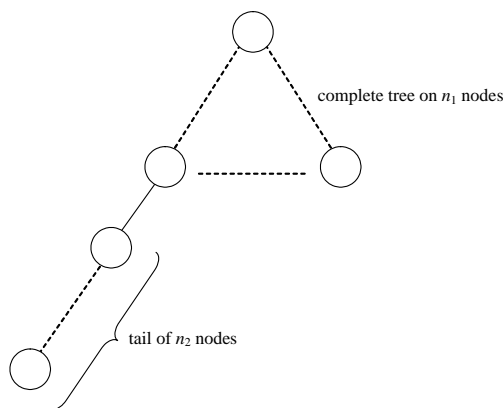
$$P(n) = \sum_{i=1}^n \text{depth}(x_i) = \sum_{x_i \in L} \text{depth}(x_i) + \sum_{x_i \in R} \text{depth}(x_i)$$

The root contributes 1 to the depth of every node in L and R . Therefore,

$$P(n) = P(k) + P(n - 1 - k) + n - 1$$

If the tree is complete then $k = \lfloor n/2 \rfloor$ and hence $n - 1 - k = \lfloor n/2 \rfloor$. Therefore, $P(n) = 2P(\lfloor n/2 \rfloor) + n - 1$. If we are interested in the asymptotic behavior of $P(n)$ we can express $P(n)$ as the recurrence $P(n) = 2P(n/2) + \Theta(n)$. By the Master theorem, $P(n) = \Theta(n \log n)$. Therefore, the average depth in the binary search tree will be $\frac{P(n)}{n} = \Theta(\log n)$.

Consider the following binary search tree consisting of a complete binary tree with a long tail.



(b) (5 points) What is the height of the above binary search tree in Θ notation?

ANSWER: Since the first part is a complete binary tree on n_1 nodes, the height of that tree is $\Theta(\log n_1)$ (we can use the same argument as the one for a binary heap). The tail of the tree adds a $\Theta(n_2)$ to the height of the complete tree. Therefore, the height of this binary search tree is $\Theta(\log n_1 + n_2)$.

(c) (5 points) Show that $P(n = n_1 + n_2) = \Theta(n_1 \log n_1 + n_2 \log n_1 + n_2^2)$ for the above binary search tree and conclude that the average depth in the above binary search tree is $\Theta(\log n_1 + \frac{n_2^2}{n_1 + n_2})$.

Using part (a), $P(n) = \Theta(n_1 \log n_1) + \sum_{x \in \text{tail}} \text{depth}(x)$. The i^{th} node on the tail has depth $\Theta(\log n_1) + i$. Therefore, $P(n) = \Theta(n_1 \log n_1) + \Theta(n_2 \log n_1) + \sum_{i=1}^{n_2} i = \Theta(n_1 \log n_1 + n_2 \log n_1 + n_2^2)$. Therefore, the average depth in the binary search tree is $\frac{P(n)}{n} = \Theta(\log n_1 + \frac{n_2^2}{n_1 + n_2})$.

(d) (10 points) Find a relation between n_1 and n_2 to make the average depth in the above binary search tree $\Theta(\log n)$ but its height $\neq \Theta(\log n)$. What is that height in Θ notation?

ANSWER: Note that if $n_2 > n_1$, then asymptotically, $n = \Theta(n_2)$ and $\frac{n_2^2}{n_1 + n_2} = \Theta(n_2) = \Theta(n)$. Therefore, the average depth will be $\Theta(n) \neq \Theta(\log n)$. Hence, we conclude that we must have $n_2 < n_1$ and as a result $n = \Theta(n_1)$ and we can work with n_1 instead of n asymptotically.

The question is now how big can n_2 be without violating that the average depth is $\Theta(\log n_1)$? Well, let's set $\Theta(\frac{n_2^2}{n_1+n_2})$ to be $\Theta(\log n_1)$. Since $n_1 > n_2$, $\Theta(\frac{n_2^2}{n_1+n_2}) = \Theta(\frac{n_2^2}{n_1})$. Therefore, we can safely set

$$n_2 = \Theta(\sqrt{n_1 \log n_1})$$

and we get that the average depth is $\Theta(\log n_1) = \Theta(\log n)$.

Let's see what this will do to the height.

$h = \Theta(\log n_1 + n_2) = \Theta(\log n_1 + \sqrt{n_1 \log n_1}) = \Theta(\sqrt{n_1 \log n_1}) = \Theta(\sqrt{n \log n}) > \Theta(\log n)$ (with abuse of notation for $>$).

As a side note, this is the best we can achieve. In other words, given that the average depth is $\Theta(\log n)$, the height of the tree must be $O(\sqrt{n \log n})$. To show this fact note that the sum of depths of nodes on a path of length l is $\Theta(l^2)$. Therefore, if the height of the tree is more than $\Theta(\sqrt{n \log n})$, the nodes on the longest path alone contribute more than $\Theta(n \log n)$ to $P(n)$. Hence, the average depth will be more than $\Theta(\log n)$.

(e) (5 points) How can you intuitively explain the fact that a bound on the average depth in a tree does not imply the same bound on the height of the tree?

ANSWER: Simply because the height of a tree is equal to the maximum depth, which can be more than the average. Of course, as mentioned above in the side note, the average depth does impose an upper bound on the height, but does not exclude the fact that the height can be much more than the average depth. Again, the reason is because the height is nothing but the maximum depth.

Here's an example from every day life not related to trees. We have 10 students in a class and the average grade is 7/100. Then this imposes an upper bound on any grade. In fact you can claim that no one got 71/100 as a grade; otherwise, the average will be $\frac{\dots+71+\dots}{10} > 7$ (well... assuming no negative grades). But it is possible to find a student with a grade 70/100. As a trivial example, one student got 70/100 and the remaining students all got 0/100. The average is 7. The maximum grade is 70.

Problem 3: Tree walks (30 points)

(a) (10 points) Write recursive pseudocodes similar to the INORDER-TREE-WALK we discussed in class for PREORDER-TREE-WALK and POSTORDER-TREE-WALK (see book page 254 for definitions).

ANSWER:

```

PREORDER-TREE-WALK( $x$ )
  if  $x \neq NIL$ 
    then print  $key[x]$ 
       PREORDER-TREE-WALK( $left[x]$ )
       PREORDER-TREE-WALK( $right[x]$ )

```

```

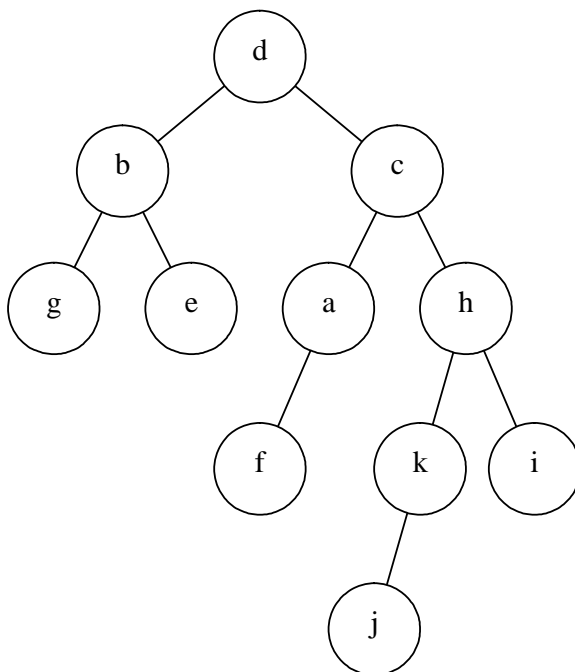
POSTORDER-TREE-WALK( $x$ )
  if  $x \neq NIL$ 
    then POSTORDER-TREE-WALK( $left[x]$ )
         POSTORDER-TREE-WALK( $right[x]$ )
         print  $key[x]$ 

```

(b) (10 points) Assume the INORDER-TREE-WALK outputs $g, b, e, d, f, a, c, j, k, h, i$ and the POSTORDER-TREE-WALK outputs $g, e, b, f, a, j, k, i, h, c, d$. Construct the tree and determine the output of the PREORDER-TREE-WALK.

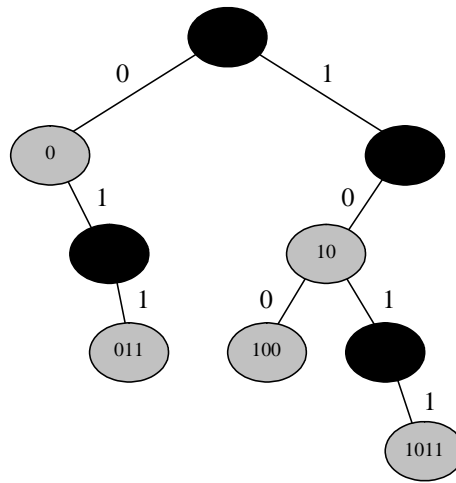
ANSWER: The post order tree walk on a tree rooted at node x outputs x last. Therefore, d is the root of the tree. Looking at the inorder tree walk output, we see that g, b, e must be in the left subtree of d and f, a, c, j, k, h, i must be in the right subtree of d .

Now we apply the same reasoning recursively on the left and right subtrees. Among g, b, e , the post order tree walk outputs b last; therefore, b is the root. From the inorder tree walk, we see that g must be on its left and e must be on its right. We continue in this way...



The PREORDER-TREE-WALK will output: $d, b, g, e, c, a, f, h, k, j, i$.

(c) (10 points) Consider a set S of distinct bit strings, i.e. string over the alphabet $\{0, 1\}$. A special tree, called radix tree, can represent such a set S . Here's the radix tree for $S = \{1011, 10, 011, 100, 0\}$.



Given a set of distinct binary strings whose length sum is n , show how you can build its radix tree in $\Theta(n)$ time and then use it to sort S lexicographically (dictionary order) in $\Theta(n)$ time. For example, the output of the sort for the radix tree above should be: 0, 011, 10, 100, 1011.

Hint: use one of the tree walks.

ANSWER: First, we build a radix tree for the set of strings S by inserting the strings into an originally empty radix tree. Insertion takes $\Theta(n)$ time since the insertion of each string takes a time proportional to its length (traversing a path through the tree whose length is the length of the string), and the sum of all the string lengths is n . At the end of each insertion, we color the node white and store the corresponding string as its key.

Second, we perform a modified preorder walk of the tree that only prints the keys for white nodes. This will print the strings in lexicographic order because:

- Any node's string is a prefix of all its descendants' strings, hence belongs before them in the sorted order
- A node's left descendants belong before its right descendants because the corresponding strings are identical up to that parent node, and in the next position the left tree's strings have 0 while the right tree's strings have 1.

Problem 4: Binary search tree and Min-Heap (20 points)

(a) (10 points) Design an algorithm that converts a binary search tree on n elements into a Min-Heap containing the same elements in $\Theta(n)$ time.

ANSWER: One possible way of performing this task is to traverse the tree using a tree walk algorithm (inorder, preorder, or postorder) and output the keys of the nodes to an array. Then Perform BUILD-HEAP on the array. Both the tree walk and BUILD-HEAP run in $\Theta(n)$ time. In particular if the INORDER-TREE-WALK is used, the array will be sorted and there is no need to run BUILD-HEAP (min-HEAP). Here's a modified INORDER-TREE-WALKS that outputs the keys to an array.

```
count ← 0
INORDER-TREE-WALK(root[T])

INORDER-TREE-WALK(x)
  if x ≠ NIL
    then INORDER-TREE-WALK(left[x])
           count ← count + 1
           A[count] ← key[x]
           INORDER-TREE-WALK(right[x])
```

PS: this assumes that a key is the only information stored at a node. If this is not true, we need to copy the whole object into the array, not just the key.

(b) (10 points) Very Challenging (EXTRA CREDIT): Do the same thing *inplace*, i.e. convert the tree into a Min-Heap in $\Theta(n)$ time without using any extra storage. We adopt the standard representation of a tree with the three *left*, *right*, and *parent* pointers.

ANSWER: The idea is the following. First, convert the binary search tree into a sorted linked list. One way of doing this is by starting at the minimum element, and repeatedly call SUCCESSOR, and link the element to its successor. Since the repeated SUCCESSOR calls will walk the tree in the same way the INORDER-TREE-WALK does, the total time for these calls will be $\Theta(n)$ (later we will prove this more formally maybe in another exercise).

The question is which pointer should we use to link an element to its successor. Messing up with the pointers might cause future successor calls to fail doing the correct thing. As an observation, the left or right pointers of a node x are never used after SUCCESSOR(x) is called. Therefore, either the left or the right pointer can serve to link x to its successor. Here's the code for converting the binary search tree into a sorted linked list using the left pointer as a "next" pointer.

```

x ← MINIMUM(root[T])
root[T] ← x
while x ≠ NIL
    do y ← SUCCESSOR(x)
       left[x] ← y
       x ← y

```

Now we can transform the sorted linked list into a min-Heap by setting the left, right, and parent pointers appropriately. The idea is that each node will have the earliest available two nodes as its children because we can fill the heap level by level from left to right without having to worry about the order of elements (it is a sorted list now).

Here's the code which takes linear time because it just traverses the list once:

```

x ← root[T]
parent[x] ← NIL
y ← left[x]
while x ≠ NIL
    do ▷ z is the node to consider as x in the next iteration
       z ← left[x]
       ▷ y is first node without a parent
       left[x] ← y
       if y ≠ NIL
           then parent[y] ← x
                ▷ find next node without a parent
                y ← left[y]
       right[x] ← y
       if y ≠ NIL
           then parent[y] ← x
                ▷ find next node without a parent
                y ← left[y]
       ▷ x has both left and right defined
       x ← z

```

(c) (10 points) Can you do the reverse process, i.e. convert a Min-Heap containing n elements into a binary search tree on the same elements in $\Theta(n)$ time? How? or Why not?

ANSWER: It is generally not possible to transform a min-Heap into a binary search tree on the same elements in linear time. Had this been possible, we would be able to sort in linear time, which is not possible due to the $\Omega(n \log n)$ lower bound on sorting.

Here's how we would sort an array A in linear time:

```
 $H \leftarrow \text{BUILD-HEAP}(A)$   
 $T \leftarrow \text{convert}(H)$   
 $\text{INORDER-TREE-WALK}(\text{root}[T])$ 
```

Since all these operations take $\Theta(n)$ time, we have a $\Theta(n)$ time general sorting algorithm. Knowing that this is impossible, the assumption that conversion takes linear time must be wrong.