Problem 1 (CLRS 12.1-3). (3 points) Give a non-recursive algorithm that performs an in-order traversal of a binary tree.

Solution:

(a) An easy solution uses a stack as an auxiliary data structure.

\[
\text{def stack-inorder(node):}
\]
\[
\quad \text{stack = []}
\]
\[
\quad // Proceed as long as we have a node
\quad // in our hands or on the stack.
\]
\[
\quad \text{while node != NULL or stack.size() > 0:}
\]
\[
\quad \quad \text{if node != NULL:}
\quad \quad \quad // We have a node in our hands;
\quad \quad \quad // save it for later and go left.
\quad \quad \quad \text{stack.push(node)}
\quad \quad \quad \text{node = node.left}
\quad \quad \text{else:}
\quad \quad \quad // node is NULL, so cannot go left anymore;
\quad \quad \quad // restore and process last seen node, go right.
\quad \quad \quad \text{node = stack.pop()}
\quad \quad \quad \text{visit(node)}
\quad \quad \quad \text{node = node.right}
\]

(b) A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality. Getting familiar with threaded binary trees is a good start.

\[
\text{def morris-inorder(node):}
\]
\[
\quad \text{while node != NULL:}
\]
\[
\quad \quad \text{if node.left == NULL:}
\quad \quad \quad // If there is no left child, in-order traversal
\quad \quad \quad // is just visiting the current node and going right.
\quad \quad \quad \text{visit(node)}
\quad \quad \quad \text{node = node.right}
\quad \quad \text{else:}
\quad \quad \quad // Find predecessor. The check pred.right != node
\quad \quad \quad // is to protect against cycles in case we already
\quad \quad \quad // threaded that node.
\quad \quad \quad \text{pred = node.left;}
\quad \quad \quad \text{while pred.right != NULL and pred.right != node:}
\quad \quad \quad \quad \text{pred = pred.right}
\]
// Predecessor is a rightmost child and cannot itself
// have a right child. If it does - it's a thread.
if pred.right == NULL:
    // Not threaded. Thread, so that after traversing
    // the left subtree we can go to node.right and
    // jump straight to the successor.
    pred.right = node
    node = node.left
else:
    // Already threaded. This means we are returning
    // from the left subtree. Unthread, then proceed
    // as usual.
    pred.right = NULL
    visit(node)
    node = node.right

Problem 2 (CLRS 12.3-1). (1 point) Give a recursive version of the Tree-Insert procedure.

Solution:

def recursive-insert(curr, prev, node):
    if curr == NULL:
        node.parent = prev
        if node.key < prev.key:
            prev.left = node
        else:
            prev.right = node
        return

    if node.key < curr.key:
        recursive-insert(curr.left, curr, node)
    else:
        recursive-insert(curr.right, curr, node)

Problem 3 (CLRS 12.3). (4 points) In this problem, we prove that the average depth of a node
in a randomly built binary search tree with n nodes is O(lg n). Although this result is weaker
than that of Theorem 12.4 in CLRS, the technique we shall use reveals a surprising similarity
between the building of a binary search tree and the execution of Randomized-Quicksort from
Section 7.3.

We define the total path length $P(T)$ of a binary tree $T$ as the sum, over all nodes $x$ in $T$, of
the depth of node $x$, which we denote by $d(x, T)$.

Solution:
(a) Argue that the average depth of a node in $T$ is

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.

By definition of $P(T)$ and dividing by $n$:

$$P(T) = \sum_{x \in T} d(x, T)$$

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

(b) Let $T_L$ and $T_R$ denote the left and right subtrees of tree $T$, respectively. Argue that if $T$ has $n$ nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

Going from $T_L$ or $T_R$ to $T$ increases the depth of all nodes by 1, so we can express $P(T)$ as follows:

$$P(T) = \sum_{x \in T_L} (d(x, T_L) + 1) + \sum_{x \in T_R} (d(x, T_R) + 1)$$

$$= \sum_{x \in T_L} d(x, T_L) + \sum_{x \in T_R} d(x, T_R) + \sum_{x \in T_L} 1 + \sum_{x \in T_R} 1$$

$$= \sum_{x \in T_L} d(x, T_L) + |T_L| + \sum_{x \in T_R} d(x, T_R) + |T_R|$$

$$= P(T_L) + P(T_R) + |T_L| + |T_R|.$$

Every node in $T$ except root is in $T_L$ or $T_R$, so:

$$|T_L| + |T_R| = |T| - 1$$

$$= n - 1,$$

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

(c) Let $P(n)$ denote the average total path length of a randomly built binary search tree with $n$ nodes. Show that

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left( P(i) + P(n - i - 1) + n - 1 \right).$$

Let us define $P(n, k)$ to be the total path length of a binary tree with $n$ nodes total and $k$ nodes in the left subtree. By (b):

$$P(n, k) = P(k) + P(n - k - 1) + n - 1.$$
The number of nodes in the left subtree is the number of elements smaller than the root. In a randomly built tree the root is equally likely to be any of the $n$ elements, and the number of elements smaller than the root element is equally likely to be any of $\{0, 1, ..., n-1\}$. Thus, the average of $P(n)$ can be expressed as:

$$P(n) = \frac{1}{n} \sum_{k=0}^{n-1} P(n, k)$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} (P(k) + P(n-k-1) + n-1).$$

(d) Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n).$$

Splitting $(n-1)$ into a separate sum:

$$P(n) = \frac{1}{n} \sum_{k=0}^{n-1} (P(k) + P(n-k-1) + n-1)$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} (P(k) + P(n-k-1)) + \frac{1}{n} \sum_{k=0}^{n-1} (n-1)$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} (P(k) + P(n-k-1)) + \Theta(n).$$

Using the fact that every term in the sum occurs twice:

$$\sum_{k=0}^{n-1} (P(k) + P(n-k-1)) = P(0) + P(n-1)$$

$$+ P(1) + P(n-2)$$

$$+ ...$$

$$+ P(n-2) + P(1)$$

$$+ P(n-1) + P(0),$$

and dropping $P(0) = 0$ gives:

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n).$$

(e) Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3 in CLRS, conclude that $P(n) = O(n \lg n)$. 

4
We use steps (c), (d), and (e) from Problem 7-3 in CLRS. To make the recurrence for \( P(n) \) obtained in the previous step more similar to recurrence (7.6) we observe that \( P(1) = 0 \) and rewrite it as:

\[
P(n) = \frac{2}{n} \sum_{k=2}^{n-1} P(k) + \Theta(n).
\]

We now proceed with proving inequality (7.7):

\[
\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2.
\]

Following the hint provided in 7-3(d):

\[
\sum_{k=2}^{n-1} k \lg k = \sum_{k=2}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k.
\]

We can obtain an upper bound by replacing \( \lg k \) in each sum with maximum values of \( k \) (and adding one for convenience):

\[
\sum_{k=2}^{n-1} k \lg k < \lg n \sum_{k=2}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k
\]

\[
= \lg n \sum_{k=2}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k
\]

\[
= \lg n \sum_{k=2}^{\lceil n/2 \rceil - 1} k - \sum_{k=2}^{\lceil n/2 \rceil - 1} k
\]

\[
\leq \lg n \frac{(n-1)n}{2} - \frac{(n/2 - 1)n/2}{2}
\]

\[
= \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 - \frac{1}{2} n \lg n + \frac{1}{4} n
\]

\[
\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \quad \text{for } n \geq 2.
\]

Following the hint in 7-3(e), we will now prove by substitution that \( P(n) = O(n \lg n) \).
We assume $P(k) \leq ck \log k$ for $1 < k < n$ and proceed with showing that $P(n) \leq cn \log n$:

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n)$$

$$\leq \frac{2}{n} \sum_{k=1}^{n-1} c k \log k + \Theta(n)$$

$$= \frac{2c}{n} \sum_{k=1}^{n-1} k \log k + \Theta(n)$$

$$\leq \frac{2c}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n)$$

$$= cn \log n - \left( \frac{c}{4} n - \Theta(n) \right)$$

$$\leq cn \log n,$$

as long as:

$$\frac{c}{4} n - \Theta(n) \geq 0$$

$$\frac{c}{4} n \geq \Theta(n),$$

and we can always find $c$ large enough so that $\frac{c}{4} n$ dominates $\Theta(n)$. Thus, $P(n) = O(n \log n)$.

(f) At each recursive invocation of quicksort, we choose a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

In quicksort:

- All elements processed after picking a pivot are compared to the pivot.
- Elements in the left and right partitions are never compared.

When inserting into a binary search tree:

- All elements inserted after inserting the root are compared to the root.
- Elements inserted into the left subtree are never compared to any elements in the right subtree, and the other way around.

Thus, in a quicksort implementation that chooses the pivots in the same order in which the elements are inserted into a binary search tree, the comparisons are going to be the same as those made when inserting the elements into the tree.
**Problem 4 (CLRS 13.1-5).** (1 point) Show that the longest simple path from a node $x$ in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node $x$ to a descendant leaf.

**Solution:** By property 5, every path contains $bh(x)$ black nodes. Thus, the shortest path has length at least $bh(x)$. By definition of height, the longest path has length $h(x)$. By property 4, $bh(x) \geq h(x)/2$ or $h(x) \leq 2bh(x)$ – the longest path has length at most twice that of the shortest path.

**Problem 5 (CLRS 13.1-6).** (1 point) What is the largest possible number of internal nodes in a red-black tree with black-height $k$? What is the smallest possible number?

**Solution:** For the largest possible number of nodes, consider a single-node red-black tree (by property 2, the node is black). Keeping black height $k$ constant rules out adding black nodes. We can, however, add up to a full level of red nodes without increasing $k$. After doing so, property 4 guarantees that all nodes on the next level are black. Once again, keeping $k$ constant rules out adding black nodes, but allows up to a full level of red nodes. This applies to all subsequent levels, resulting in a perfect red-black tree with alternating black and red levels (ending with a red level), height $h = 2k$ and number of nodes $n = 2^{2k} - 1$.

For the smallest possible number of nodes, by lemma 13.1 in CLRS, the red-black tree with black-height $k$ contains at least $2^k - 1$ internal nodes.

**Problem 6 (CLRS 13.1-7).** (1 point) Describe a red-black tree on $n$ keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

**Solution:** For the largest red/black ratio we need to increase the number of red nodes, while decreasing or keeping constant the number of black nodes. We employ the same reasoning as in problem 5: increasing the number of red nodes in a single-node tree is only possible up to a full level, after which property 4 forces us to fill the next level with black nodes; at this point we again cannot decrease the number of black nodes, but can add up to a full level of red nodes. Repeating this procedure, the largest number of red nodes will be observed in a perfect red-black tree with alternating black and red levels, ending with a red level. In this case, $n_{\text{red}} = 2n_{\text{black}}$ and the red/black ratio is 2.

The smallest possible ratio (0) is observed in a tree with black nodes only.