Assignment 6

Benjamin Jakubowski

July 7, 2016

1. Non-recursive algorithms for inorder BST traversal

a. Solution using a stack

The idea for the in-order BST traversal algorithm using a stack can be summarized simply:

- Go as far left as you can
- Once you've gone as far left as you can, you've found the (next) smallest element. Visit (print) it, then move right

The algorithm is presented in full below:

```
function INORDER_TRAVERSAL_WITH_STACK(T)
    x = T.root
    nodes_seen = new Stack()
    while (len(nodes_seen) > 0) or (x is not Null) do
        if (x is not Null) then
            nodes_seen.push(x)
            x = x.left
        else
            x = nodes.seen.pop()
            print x
            x = x.right
```

b. Solution without stack, using threading

Note this algorithm assumes that threads are not already present in the tree (i.e. the threads from leaves to successors are constructed and removed in the course of the algorithm). The algorithm is explained via comments.

```
function INORDER_TRAVERSAL_WITHOUT_STACK(T)
    x = T.root
    while (x is not Null) do
```

```
    if (x.left is Null) then                          ▷ Then x's successor is its right child
        print x
        x = x.right
    else                                  ▷ Check if x's predecessor is already threaded to x
        tmp = x.left                      ▷ x's predecessor is right-most leaf of left subtree
        while (tmp.right is not Null) and (tmp.right is not x) do:
            tmp=tmp.right
        if tmp.right is Null then         ▷ Then no thread from predecessor to x
            tmp.right = x                                                    ▷ Add thread
            x = x.left                                  ▷ Move from x into its left subtree
        else                          ▷ Broke out of while loop with tmp.right == x
            tmp.right = Null          ▷ Thus we've visited x's entire left subtree.
            print x                                  ▷ Visit (print) x after nulling thread.
            x = x.right                                        ▷ Move into x's right subtree
```

2. Recursive version of `Tree-Insert`

```
function RECURSIVE_INSERT(T, z, x = T.root)
    T is tree, z is node to insert, x is starting node for recursive call
    Note we use default arguments to handle the initial call
    if z is Null then
        return                                                        ▷ Handles edge case
    else if x is Null then                                        ▷ Then T.root is Null
        T.root = z
    else if z.key < x.key then
        if x.left is Null then                          ▷ Then insert z as left child of x
            x.left = z
            z.parent = x
        else                                                          ▷ Then recurse
            recursive_insert(T, z, x.left)
    else                                                      ▷ Then z.key ≥ x.key
        if x.right is Null then                       ▷ Then insert z as right child of x
            x.right = z
            z.parent = x
        else                                                          ▷ Then recurse
            recursive_insert(T, z, x.right)
```

3. Average depth of node in BST

Our objective in this problem is to show that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$.
First, 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 $d(x, T)$.

A. Average depth of a node in T

The total path length is

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

Thus, dividing by $\frac{1}{n}$ yields the desired result

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

B. Recursive definition of $P(T)$

Let T_L and T_R be the left and right subtrees of tree T, respectively. We aim to show if T has n nodes, then

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

First, note

$$d(x, T_L) = d(x, T) - 1 \qquad \text{for all } x \in T_L$$
$$d(x, T_R) = d(x, T) - 1 \qquad \text{for all } x \in T_R$$

Thus,

$$\begin{aligned}
P(T) &= \sum_{x \in T} d(x, T) \\
&= \sum_{x \in T_L} d(x, T) + \sum_{x \in T_R} d(x, T) + d(x_{root}, T) \\
&= \sum_{x \in T_L} [d(x, T_L) + 1] + \sum_{x \in T_R} [d(x, T_R) + 1] + 0 \\
&= \sum_{x \in T_L} d(x, T_L) + \sum_{x \in T_R} d(x, T_R) + \underbrace{(n-1)}_{\text{since } n-1 \text{ nodes in } T_R + T_L} \\
&= P(T_L) + P(T_R) + n - 1
\end{aligned}$$

C. Expression for $P(n)$

Now let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. We aim to show that

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

We use induction on n to do so. First, note that $P(0)$ is trivially 0. Thus, for a base case, we show this expression holds for $P(1)$ (noting in advance a randomly build a tree with $n = 1$ nodes always has a total path length of 0):

$$\begin{aligned} P(n=1) &= \frac{1}{1}\sum_{i=0}^{1-1}\left(P(i) + P(1-i-1) + 1 - 1\right) \\ &= \left(P(0) + P(1-0-1) + 1 - 1\right) \\ &= 0 \end{aligned}$$

Next, assume this expression is true for all $k < n$. Now consider a tree with n nodes. Let the n keys in this tree be ordered as

$$k_0, k_1, \cdots, k_{n-1}$$

Then, under the assumption the n keys are inserted into the BST in permuted order, with all permutations equiprobable, the probability the root node (i.e. first key inserted) has key k_j is $\frac{1}{n}$ for all $j \in \{0, 1, \cdots, n-1\}$.
Next, note (given k_j is the root)

- T_L has j nodes
- T_R has $n - j - 1$ nodes

Thus, (given k_j is the root), by our inductive hypotheses the average total path for this tree is $P(j) + P(n-j-1) + n - 1$.
Finally, to find the average total path length for randomly built n node BST's, we simply take the expectation over j, the index of the root key in our ordered set:

$$\begin{aligned} P(n) &= \sum_{j=0}^{n-1} P(\text{Insert key } k_j \text{ first})\left[P(j) + P(n-j-1) + n - 1\right] \\ &= \sum_{j=0}^{n-1} \frac{1}{n}\left[P(j) + P(n-j-1) + n - 1\right] \\ &= \frac{1}{n}\sum_{j=0}^{n-1}\left[P(j) + P(n-j-1) + n - 1\right] \end{aligned}$$

D. Rewriting $P(n)$

We proceed from the expression derived in part (c)

$$
\begin{aligned}
P(n) &= \frac{1}{n}\left[\sum_{j=0}^{n-1} P(j) + \sum_{j=0}^{n-1} P(n-j-1)\right] + \frac{1}{n}n(n-1) \\
&= \frac{1}{n}\left[P(0) + P(1) + \cdots + P(n-1) + P(n-1) + P(n-2) + \cdots + P(0)\right] + \underbrace{(n-1)}_{\Theta(n)} \\
&= \frac{2}{n}\underbrace{\sum_{j=0}^{n-1} P(j)}_{\text{Recall } P(0)=0} + \Theta(n) \\
&= \frac{2}{n}\sum_{j=1}^{n-1} P(j) + \Theta(n)
\end{aligned}
$$

E. SHOWING $P(n) = O(n \lg n)$

First, following CLRS problem 7-3, we show

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$$

$$
\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &= \sum_{k=2}^{n-1} k \lg k \qquad \text{Since } \lg 1 = 0 \\
&\leq \sum_{k=2}^{n} k \lg k \\
&\leq \int_{k=2}^{n} k \lg k \; \mathrm{d}k \qquad \text{Since } k \lg k > 0 \text{ and monotone} \\
&= \left[\frac{1}{2}k^2 \lg k - \frac{1}{4 \ln 2}k^2\right]\Bigg|_{k=2}^{k=n} \\
&\leq \left[\frac{1}{2}k^2 \lg k - \frac{1}{8}k^2\right]\Bigg|_{k=2}^{k=n} \\
&= \left[\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right] - \left[\frac{1}{2}2^2 \lg 2 - \frac{1}{8}2^2\right] \\
&= \left[\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right] - \left[2 - \frac{1}{2}\right] \\
&\leq \left[\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right]
\end{aligned}
$$

Next, note for $k = 1, P(k = 1) = 0 \leq 0 = c \cdot 1 \cdot \lg 1$ for any $c > 0$. Now assume for all $k < n$, there exists some $c > 0$ such that

$$P(k) \leq c \cdot k \lg k$$

Then

$$\begin{aligned} P(n) &= \frac{2}{n} \sum_{j=1}^{n-1} P(j) + \Theta(n) \\ &= \frac{2}{n} \sum_{j=2}^{n-1} P(j) + \Theta(n) \qquad \text{Since } P(1) = 0 \\ &\leq \frac{2}{n} \sum_{j=2}^{n-1} c_j \cdot k \lg k + \Theta(n) \end{aligned}$$

Now let $c_{\max} = \max\{c_2, \cdots, c_{n-1}\}$. Then

$$\begin{aligned} P(n) &\leq \frac{2}{n} \sum_{j=2}^{n-1} c_{\max} \cdot k \lg k + \Theta(n) \\ &\leq \frac{2c_{\max}}{n} \left[\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right] + \Theta(n) \\ &= c_{\max} \cdot n \lg n - \frac{c_{\max}}{4} n + \Theta(n) \end{aligned}$$

Next, let's unpack $\Theta(n)$ (using the implied upper bound $k \cdot n$ for all $n > n_0$).

$$\begin{aligned} P(n) &\leq c_{\max} \cdot n \lg n - \frac{c_{\max}}{4} n + k \cdot n \qquad \forall\ n > n_0 \\ &= c_{\max} \cdot n \lg n + \left(k - \frac{c_{\max}}{4} \right) n \end{aligned}$$

Now, note the inequality assumed in the inductive hypothesis

$$P(k) \leq c \cdot k \lg k \leq c_{\max} \cdot k \lg k$$

also holds for any $c' > c_{\max}$. In particular, it holds for $c' \geq \max\{c_{\max}, 4k\}$. But then note

$$\left(k - \frac{c'}{4} \right) n \leq 0$$

This yields the final inequality

$$\begin{aligned} P(n) &= c_{\max} \cdot n \lg n + \left(k - \frac{c_{\max}}{4} \right) n \\ &\leq c' \cdot n \lg n + \left(k - \frac{c'}{4} \right) n \\ &\leq c' \cdot n \lg n \qquad \forall\ n > n_0 \end{aligned}$$

Thus, $P(n) = O(n \lg n)$.

f. An implementation of quicksort with same comparisons as BST

Consider a random array of n distinct integer elements.

$$A = [a_1, \cdots, a_n]$$

Consider constructing a BST from A. Note the root of the tree is the first element inserted in the BST. It is also compared to all other elements in A- thus it is the first pivot in our desired quicksort algorithm.
Similarly, given an internal node x

- $x.left$ is
  - The pivot for the first recursive quicksort call (i.e. the call for all elements in the range `lo` to `hi` that are less than x).
  - The first element in this range inserted into the BST
- $x.right$ is
  - The pivot for the second recursive quicksort call (i.e. the call for all elements in the range `lo` to `hi` that are greater than x).
  - The second element in this range inserted into the BST.

Thus, we can implement quicksort with the same comparisons as BST fairly simply. All we need to do is change how we select the pivot. Recall in vanilla quicksort, the function `partition(A, lo, hi)` picks `pivot = A[hi]`. Instead, we pick the element in the range $A[lo, hi]$ inclusive that had the lowest index in the original ordering of array A.
We do so by implementing the following function, which calls a helper function `original_index` which gets the original index of `A[i]`:

```
function PUT_PIVOT_AT_HI(A, lo, hi)
    original_index_of_pivot = original_index(A[lo])
    current_index = lo
    for i in lo+1, ··· hi do
        if original_index(A[i]) < original_index_of_pivot then
            original_index_of_pivot = original_index(A[i])
            current_index = i
    swap(A[i], A[hi])
```

Note this version of the algorithm takes more space, but ensures we make the same comparisons.

4. Bound on longest simple path in red-black tree

Let x be a node in a red-black tree. Then, by red-black tree property 5 , the shortest simple path from x to a descendent leaf also has $bh(x)$ black nodes and at least 0 red nodes. Thus the length of the shortest path is

$$P_{shortest} \geq bh(x)$$

Next consider the longest simple path from x to a descendent leaf. The length of this path is simply the height of x, namely $h(x)$. Note the number of black nodes on the path is $bh(x)$. Thus, the number of red nodes on this path is

$$rh(x) = h(x) - bh(x)$$

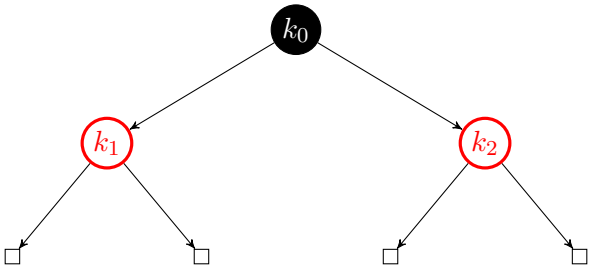
Now note $rh(x) \leq bh(x)$. This follows by red-black tree property 4 (since every red node in the path is immediately followed by a black node). Thus,

$$h(x) = rh(x) + bh(x) \leq bh(x) + bh(x) = 2bh(x) \leq 2P_{shortest}$$

Hence, the length of the longest simple path $h(x)$ is at most twice that of the shortest simple path.

5. Largest and smallest number of internal nodes

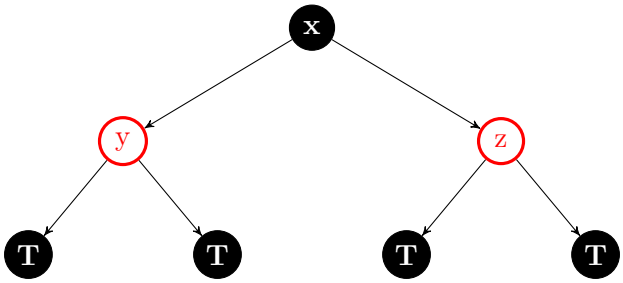
First, by CLRS page 309, a tree rooted at node x with black-height $bh(x) = k$ has at least $2^k - 1$ internal nodes.
Now for an upper bound on the number of internal nodes. To start, consider a single node tree with key k_0. Note this single node tree has black height 1, and adding two red nodes with keys k_1 and k_2 such that $k_1 < k_0 < k_2$ does not change the black height.



However, adding any additional nodes increases the black height (since the parents of the $T.nil$ leaf nodes are now all red, so adding additional nodes necessarily requires recoloring and rotating, which increases the black height).
Thus, when the black height is 1, we can have no more than 3 internal nodes. This suggests the bound on $n(T_k)$, the number of internal nodes in a tree T with black-height k, of

$$n(T_k) \leq 2^{2k} - 1$$

Now we prove this bound by induction. Assume all trees with black-height $k' < k$ have at most $2^{2k'} - 1$ internal nodes. Now consider a red black tree with black-height k. Note we can construct the tree black-height k with a maximum number of internal nodes such a tree as follows:



where T are subtrees with black-height $k-1$ and at most $2^{2(k-1)}-1$ internal nodes. Then the number of internal nodes in this tree is

$$\begin{aligned} n(T_k) &\leq 4 \cdot (2^{2(k-1)} - 1) + 3 \\ &\leq 2^2 \cdot 2^{2k-2} - 4 + 3 \\ &\leq \cdot 2^{2k-2+2} - 1 \\ &\leq \cdot 2^{2k} - 1 \end{aligned}$$

Thus $n(T_k) \leq 2^{2k} - 1$.

6. Largest and smallest possible ratio of red to black internal nodes

Per the hint, we chose to pick a convenient n. Specifically, let's take some n such that

$$n = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

for some $k \geq 1$ where $k \mod 2 = 1$.
Then it is possible to construct a complete binary tree. This tree satisfies the 5 red-black properties if we color all its nodes black, and have 0 red nodes. Hence the minimum ratio of red to black nodes for this special case is 0 red : n black, which is clearly the minimum ratio.

Now we consider the maximum case: then, the maximum number of red nodes in our tree is obtained by alternating the color of each level, so nodes x with $d(x,T) \mod 2 = 1$ are red, and all others are black.
Then we have

$$n_{black} = \sum_{i=0,2,\cdots,k-1} 2^k$$

and

$$n_{red} = \sum_{i=1,3,\cdots,k} 2^k$$

But then we clearly have twice as many (internal) red nodes as black nodes, so the ratio of internal red to black nodes is 2 red: 1 black.