CSC263: Problem Set 3

Angus Fung,[1] Syed Kamran,[1] Michelle Leung[1]

[1]Division of Engineering Science, University of Toronto,

Question 1

(a) Let P_k be the probability that the *Binary Search Tree*, which we will hereafter abbreviate as BST, has height k after insertion of a permutation chosen uniformly at random from the set $\{1, ..., 2^k - 1\}$. For P_1, which is the probability that the BST will have height 1, is simply 1, or in other words, will happen 100% of the time. That is because choosing a random number from the set $\{1\}$ will always result in a tree of height 1. For $k = 2$, the probability space is $\{1, 2, 3\}$ which yields $3! = 6$ permutations. Enumerating said permutations, it is obvious to see that only the permutations $\{2, 1, 3\}$ and $\{2, 3, 1\}$ result in height 2 under the stipulations of BST insertion. Furthermore, it can be seen that in any probability space P_k as defined in the problem set, the median must be inserted as the root node in order to have minimum height. Finally, $P_2 = 2/6 = 1/3$, which is the number of permutations yielding height $k = 2$ over the total number of possible permutations.

(b) For the case $k = 3$, $\{4, 2, 6, 1, 3, 5, 7\}$ and $\{4, 6, 2, 1, 3, 5, 7\}$ both result in a height of $k = 3$, the minimal height. Note that there exist more permutations that result in a height of 3, but they require that the root node being 4 (the median) with its sub-trees being 2 and 6. 2 and 6 are the medians of the partitioned sets. Since if we start if any other root node, we are restricted by the positions to which additional nodes can be inserted. It is intuitive that inserting 4 as the root node is most optimal, as its left sub-tree will contain nodes smaller than 4, and the right sub-tree will contain nodes larger than 4. Since 4 is middle element of the list $\{1, ..., 7\}$, then this will potentially partition the tree into two evenly balanced sub-trees.

(c) For a given P_k, for $k \geq 2$, P_k is the probability of inserting the *correct* root node multiplied by P_{k-1} squared. P_{k-1} is the probability of the sub-trees being of minimal height for which there are two. As the recurrence is carried out, this will apply aptly for the sub-trees, which can be seen as a BST in its own right. There is only one possible choice for the root node, and as stated earlier, the root node must be the center or middle element in the set. This element can always be found because $2^k - 1 \mod 2 = 1$: it is always odd for $k \geq 1$. Since all $2^k - 1$ nodes must be inserted, the minimum height k must be a *complete* tree. In this sense, choosing any other root node would skew the balance (where one sub-tree has more nodes than the other) and would necessarily have a height greater than k. The probability of choosing the root node is $\frac{1}{2^k-1}$. Since the probability of each sub-tree being of minimal height $k-1$ is independent of each other, we can simply multiple them together. Therefore, $P_k = (P_{k-1})^2 \cdot \frac{1}{2^k-1}$.

(d) Prove $P_k \geq 2^{k+2-2^{k+1}}$ for $k \geq 1$, where $P_k = \frac{P_{k-1}^2}{2^k-1}$ for $k \geq 2$, using mathematical induction.

Proof. This is true for k=1:

Since the aforementioned formula derived applies only for $k \geq 2$, we can use the result of $P_1 = 1$ from part 1a).

$$\begin{aligned} P_1 &= 1 \\ &\geq 2^{3-2^3} \\ &= 2^{-1} \\ &= \frac{1}{2} \end{aligned}$$

Assume true for $k = n$ (induction hypothesis):

$$\frac{P_{n-1}^2}{2^n - 1} \geq 2^{n+2-2^{n+1}}$$

Then,

$$\begin{aligned}
P_{n+1} &= \frac{P_n^2}{2^{n+1}-1} \\
&= \frac{1}{2^{n+1}-1}\left(\frac{P_{n-1}}{2^{n-1}}\right)^2 \\
&\geq \frac{1}{2^{n+1}-1} 2^{n+2-2^{n-1}} 2^{n+2-2^{n-1}} \text{ (induction hypothesis)} \\
&= \frac{1}{2^{n+1}-1} 2^{2(n+2)-2(2^{n+1})} \\
&\geq \frac{1}{2^{n+1}} 2^{2(n+2)-2(2^{n+1})} \\
&= 2^{2(n+2)-2(2^{n+1})-(n+1)} \\
&= 2^{(n+1)+2-2^{(n+1)+1}} \text{ as required.}
\end{aligned}$$

Since this is true for $k = n+1$, it must be true for all $k \geq 1$ by mathematical induction.

■

Question 2

(a) We note that two key properties of a Binary Search Tree is that there exists two distinguished subtrees, denoted left and right, and that the key in each node is smaller than all the keys stored in the right subtree and greater than all the keys stored in the left subtree.

```
def Rank(D,k):
    # Returns the number of key-value pairs in D have have a
    # key greater than or equal to k

    if D.root == null:
        return 0

    elif D.root.key == k:
        return D.root.rank

    elif D.root.key > k:
        if D.left == null:
            return D.root.rank
        else:
            return Rank(D.left, k)

    else:
        if D.right == null:
            return D.root.rank - 1
        else:
            return Rank(D.right, k)
```

Since the above algorithm is starts at the root node and compares the key with k, then checks the left and right subtrees by recursion, the algorithm runs through each level of the tree once until it has found all the keys greater than k, thus the algorithm is $O(h)$. Furthermore, in the worst case scenario where k is some key greater than all other keys in the tree, the algorithm will check one node on every level of the tree once until it gets to the maximum key with rank 1, but since the right child of that node is null, the value returned will be 0 but the algorithim will still have run in $O(h)$ time.

(b) Any *Insert* algorithm must run through the list from the minimum key at rank n up the node below the key which was just inserted to add one to the rank to correctly update the rank attributes of all the nodes below the inserted node. In the worst case scenario, the node being inserted is the new maximum key, hence its rank will be 1, so the algorithm must run through every node of the original tree and add one to the rank attribute in order to update the ranks of all the nodes below it. Since the size of the original list is n, the dictionary must take $\Omega(n)$ in the worst case.

(c) Let's begin by considering an arbitrary binary search tree, of size n, and height h. We additionally define 4 potential cases that our algorithm will encounter, two trivial and two slightly involved. Let's consider our first trivial case of the root of the tree being null, in which case we return 0 and our algorithm is correct. The second trivial case we consider is the possibility that the root is equal to the key we are searching for, in which case since we know that the right side of the tree contains all numbers higher than the root, we return the size of the right sub-tree incremented by 1.

We continue to tackle our two slightly involved cases, the first being that the key of the root is greater than the key we are searching for. In this case since we know that the key we are searching for may exist in the left sub-tree of the root, and that the right sub-tree contains all keys larger than the left sub-tree, we return the size of the right sub-tree incremented by 1 as well as a recursive call to the Rank function with the left sub-tree as the new root.

Finally, we consider our last case where the key we are in search of is greater than the root. In this case we simply return a recursive call to the Rank function with the right sub-tree as the new root.

```
def Rank(D,k):
    # Returns the number of key-value pairs in D have have a
    # key greater than or equal to k

    if D.root is null:
        return 0;

    elif D.root.key == k:
        return D.right.size + 1

    elif D.root.key > k:
        return D.right.size + Rank(D.left, k) + 1

    else:
        return Rank(D.right, k)
```