a1

Tony and Andrew

January 2024

1. **Solution:**

1.

Let H(n) be the average height of a BST with keys from $1, ..., n$ inserted in uniformly random order.

Assume that a non-empty, single node with 0 subtrees has a height 1.

To find the average height of the resultant tree when 1,2,3,4, and 5 are inserted, in other words H(5), we must find H(1) to H(4) first. This is because we can find H(5) from the average height of smaller trees.

The height of a BST is the max height between each of their sub-trees plus 1. When considering the average height of a tree, with n number of nodes, we can break down a tree into one root node with left and right subtrees. By the implementation in class, the left subtree must have elements strictly less than the root node while the right subtree must have elements strictly greater than the root node. So the elements in the left and right subtree are determined by what is the root node. Since we may assume that the probability of any element from 1 to 5 may be the root node is equal, we can evaluate cases for when each number is the root node.
Another premise we must establish is that the average height of a tree with a root node k such that $1 <= k <= n$ can be determined by finding the average height of the greater sub tree and adding 1 (in many, but NOT all cases).

IMPORTANT NOTE: This only works in some cases because it assumes that, in all cases, when calculating the average size of the binary search tree the height of the larger tree with more nodes is greater than or equal to the smaller one. So when the average is calculated, only the average of the greater subtree is considered. Again, this is only possible if the minimum height of the subtree with more nodes is greater than or equal to the maximum height of the subtree with fewer nodes.

In this case, to find H(5), we can first find the separate average heights of the tree given that the root node is 1,2,3,4 or 5: conditional expectations (Chapter 1.2.2 http://jeffe.cs.illinois.edu/teaching/algorithms/notes/01-random.pdf). Then, we can find the expectation of these conditional expectations. In simpler terms, we are basically taking the sum of the average heights when the root node is 1,2,3,4,and 5, then dividing it by 5 to get the average.

The general formula for this idea is as follows. NOTE: this is not actually a general formula for all cases, only the cases of n where the IMPORTANT NOTE above holds in all cases for each node and the subsequent subtrees possible for that node.

$$H(n) = \frac{1}{n}\sum_{i=1}^{n}(1 + \max\left(H(i-1) + H(n-i)\right))$$

Now, lets calculate the necessary values for finding H(5). Note, again, that the height of the subtrees.
Let $rootis(< number)$ represent root is $(number)$.

It is clear that H(0) = 0. Since there are no nodes we can assume the average height of such a tree is 0.

It is also obvious that H(1) = 1, since a tree with one node can only have a height of 1.

For H(2), there are two cases. Given that the root node is 1 (which has a probability of 1/2), then H(2) is 1 + H(1) = 2. Given that the root note is 2, the same result occurs. So H(2) = P(root node is 1) * (H(2) — root node is 1) + P(root node is 2) * (H(2) — root node is 2) = (1/2)*2 + (1/2)*2 = 2. In short, H(2)=2 due to the semi-general formula from earlier which is supported by conditional probability.

For H(3), there are three cases: the root node is 1,2, or 3. The probability that the root node is an single one of those listed is 1/3. The possible subtree pairings, with reference to the number of nodes in each subtree, are as follows: 0 and 2, 1 and 1. Denoting L as the subtree with more (or the same) number of nodes, and M as the subtree with fewer (or the same) number of nodes, it is easy to see in these cases that the minimum height of L is always greater than or equal to the maximum height of M. Therefore, as stated previously, we find the conditional H(3) for a particular root node by adding 1 to the larger of the average heights of the two subtrees. Here are the computations for each condition:

Given root is 1:

$$H(3) = 1 + max(H(1-1) + H(3-1)) = 3$$

Given root is 2:

$$H(3) = 1 + max(H(2-1) + H(3-2)) = 2$$

Given root is 3:

$$H(3) = 1 + max(H(3-1) + H(3-3)) = 3$$

Total for H(3)

$$H(3) = \frac{1}{3}((H(3)|rootis1) + (H(3)|rootis2) + (H(3)|rootis3))$$
$$H(3) = \frac{1}{3}(3 + 2 + 3)$$
$$H(3) = 8/3$$

For H(4), there are four cases: the root node is 1,2, 3, 4. The probability that the root node is an single one of those listed is 1/4. The possible subtree pairings, with reference to the number of nodes in each subtree, are as follows: 0 and 3, 1 and 2. Denoting L as the subtree with more nodes, and M as the subtree with fewer nodes, it is easy to see in these cases that the minimum height of L is always greater than or equal to the maximum height of M. We can thus follow the same calculations as before, and apply the semi-generalized formula for H(n).

Total for H(4)

$$H(4) = \frac{1}{4}((H(4)|rootis1) + (H(4)|rootis2) + (H(4)|rootis3) + (H(4)|rootis4))$$
$$H(4) = \frac{1}{4}\sum_{i=1}^{4}(1 + \max(H(i-1) + H(4-i)))$$
$$H(4) \approx 3.333$$

For H(5), there are five cases: the root node is 1,2, 3, 4, 5. The probability that the root node is an single one of those listed is 1/5. The possible subtree pairings, with reference to the number of nodes in each subtree, are as follows: 0 and 4, 1 and 3, 2 and 2. Denoting L as the subtree with greater (or equal) number of nodes, and M as the subtree with fewer (or equal) number of nodes, the minimum height of L is always greater than or equal to the maximum height of M. It is a little less clear here than for the previous cases though, so let us be more thorough.

A binary tree with no nodes has a maximum height of 0. This is less than the minimum height of 3 for a BST with 4 nodes (consider a "complete tree"). When calculating the average height for a five-node BST when the subtrees follow the above split, it is clear that you consider only the average height of the subtree with four nodes, and add one to it. The maximum height for a BST with one node is 1, while the minimum height for a BST with 3 nodes is 2. For two and two, the maximum and minimum heights are the same: 2. So it adheres to the important note from earlier and thus can follow the property noted for the semi-generalized formula for H(n).

Total for H(5)

$$H(5) = \frac{1}{5}((H(5)|rootis1) + (H(5)|rootis2) + (H(5)|rootis3) + (H(5)|rootis4) + (H(5)|rootis5))$$

$$H(5) = \frac{1}{5}\sum_{i=1}^{5}(1 + \max\left(H(i-1) + H(5-i)\right))$$

$$H(5) = 3.8$$

Therefore, $H(5) = 3.8$.

2.

a)

```
1 def ExtractSecondLargest(Q):
2    size = len(Q)
3    if size < 2:
4        print("Please input an array of size greater than 2")
5        return None
6    elif size == 2:
7        second_largest = Q[1]
8        Q.pop()
9        return second_largest
10   else:
11       SL_value = Q[1]
12       SL_index = 1
13       if Q[1] < Q[2]:
14           SL_value = Q[2]
15           SL_index = 2
16       Q[SL_index], Q[size-1] = Q[size-1], Q[SL_index]
```

```
17        Q.pop() # removal from array end is O(1), as state in class
18        if SL_index <= size - 2:
              # the if statement here prevents a potential "index out
              # of bounds error" that may occur if Q was of size 3,
              # and its second largest value was in index 2.
19            MaxHeapify(Q, SL_index)
20        return SL_value
```

b)

Explanation for why the second largest element is returned:

Consider an input max heap array of size n.

- Case 1: $n < 2$. Then there is no second largest element. As expected, the pseudocode returns None.
- Case 2: $n = 2$. By the max heap property and the fact that all the elements in the heap are distinct, the child of the root must have a smaller element than the root. Therefore, the second largest element is $Q[1]$, and as expected, its value is returned.
- Case 3: $n > 2$. There are 2 or more distinct elements; therefore there is a second largest element. It can't be the value of the root (the node with depth 0), as by the max heap property, it's the largest. It also can't be the value of a node with depth ≥ 2, because such a node has at least 2 ancestors, and those ancestors have distinct values greater than the node's. Thus, it is the value of one of the two nodes with depth 1 (which are at either index 1 or 2). More specifically, it must be the one with the larger value. (If the smaller one had the second largest value, then there would exist two nodes with distinct, greater values, which is a contradiction). As expected, the code returns the larger value between the ones stored at indices 1 and 2.

Explanation for why the max-heap property is maintained:

Consider an input max heap array Q of size n.

- Case 1: $n < 2$. No alterations are made, and the max-heap property is maintained.
- Case 2: $n = 2$. Only a leaf node is removed (the left child of the root), so the max-heap property is maintained.
- Case 2: $n > 2$. After the "swap" and deletion is made on lines 16-17 and before `MaxHeapify` is called on line 19, the root of Q (let's call this node R) still has a value greater than all other values in the tree (since only values in its subtrees were rearranged or deleted). Consider the subtree of R that did not contain the second largest element in its root. The only possible change that could've occurred to this

subtree is the "migration" of one of its leaf nodes to the other subtree. Therefore, it is still a max heap. Now consider the subtree of R that did contain the second largest element in its root (let's call the root of this subtree S). Both subtrees of S are still max-heaps, since the only possible change that could've occurred to them is the "migration" of a leaf node to S. In the code, "`SL_index`" stores the index of S, and as the preconditions are satisfied, calling `MaxHeapify`(Q, SL_index) results in the subtree starting at S to become a max heap.

As both subtrees of R are max-heaps, and R contains the greatest value in the tree, Q remains a max-heap after the code finishes executing.

Correctness Proof for `MaxHeapify`(Q, i)

*** We aren't sure if this is necessary. Our reasoning for including this is that although we covered `MaxHeapify` in class, we did not explicitly go over the proof for it. Disregard this part if it is not needed ***

—

Precondition: the subtrees of the node stored at index i are max-heaps.

Postcondition: the tree with root at i is a max-heap.

This is specifically a proof on the *recursive* implementation of `MaxHeapify`. We prove this by induction on n, the number of nodes stored in the tree rooted at i.

- **Base case:** Let $n = 1$. Consider a tree with one node and the precondition satisfied. The code does not change anything, and by default a tree with one node is a max heap.
- **Inductive step:** Let $n \geq 2$, and assume for all trees less than size n that satisfy the precondition, calling `MaxHeapify` on the root of the tree results in the postcondition being satisfied. Consider a tree of size n rooted at i, with the preconditions satisfied. We refer to the root as R. If R is greater than both the root values of its subtrees, then the tree rooted at R is a max-heap, and nothing more needs to be done, the postcondition is satisfied.

  Otherwise, if R has a value less than one of the roots of its subtrees, it swaps with the largest of the two. Let's call this new root R', and the root (with the smaller value) of the other subtree S. The subtree rooted at S is a max-heap (by the precondition); since the value of R' is greater than the value of S, it is also greater than all the other values in the subtree. The value of R' is also greater than the values stored in its old subtrees and R. After `MaxHeapify` is called on R, as the subtree rooted at R has fewer than n nodes and the precondition is satisfied, the tree rooted at R becomes a max-heap.

  As the value of R' is greater than all other nodes in the tree and both its subtrees are max-heaps, the tree rooted at R' is a max-heap and the postcondition is satisfied.

c)

Determining an upper bound on the worst case runtime

Consider an arbitrary max-heap array of height h and number of nodes n (so n is between 2^{h-1} and $2^h - 1$), where $h \geq 3$. The size of the array is greater than 2, and all steps before `MaxHeapify` (steps 11-18) and the return statement on line 20 take constant time.

From class, we know that `MaxHeapify` takes $\mathcal{O}(\log n)$ time. We include the runtime justification just in case, but if it is not necessary, please skip to the end.

***

We first define the runtime of `MaxHeapify` in terms of the height h of the root node it is called on. We define the upper bound of the worst-case runtime of `MaxHeapify` $T(h)$ as:

$$T(h) = \begin{cases} 1 & \text{if } h = 1, \\ 1 + T(h-1) & \text{if } h > 1. \end{cases}$$

This is because the recursive call is on a subtree, with height at most $h-1$, and otherwise the rest of the steps take constant time. We see that for $h \geq 3$,

$$\begin{aligned} T(h) &= 1 + T(h-1) \\ &= 1 + (1 + T(h-2)) \\ &\vdots \\ &= h - 1 + T(1) \\ &= h \end{aligned}$$

So $T(h) \in \mathcal{O}(h)$. *This can be proved rigorously through induction.

Since the relationship between height and number of nodes for a max-heap can be described by the following:

$$h = \lfloor \log_2(n) \rfloor + 1,$$

therefore $T(h) \in \mathcal{O}(\log n)$. ***

As `MaxHeapify` takes at most logarithmic time with respect to n, and the the rest of `ExtractSecondLargest` takes constant time, the worst case total runtime is thus $\in \mathcal{O}(\log n)$.