UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 1, Version 1
CSC263H1F

October 14 2016, 10:10-11:00am (**50 min.**)
Examination Aids: No aids allowed

Name:

Student Number:

Please read the following guidelines carefully!

- Please write your name on the front **and back** of the exam.
- This examination has **4** questions. There are a total of **11 pages, DOUBLE-SIDED**.
- Answer questions clearly and completely. Give complete justifications for all answers unless explicitly asked not to. You may use any claim/result from class, unless you are being asked to prove that claim/result, or explicitly told not to.

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

Good luck!

1. Consider the following algorithm, which checks whether two arrays are equal.

```
def array_equal(array1, array2):
  i = 0
  while i < array1.length and i < array2.length:
    if array1[i] != array2[i]:
      return False
    i = i + 1

  # Check whether we've reached the end of both arrays
  if i != array1.length or i != array2.length:
    return False
  else:
    return True
```

(a) [**2 marks**] Let m be the length of `array1` and n be the length of `array2`. Prove that the worst-case running time of this algorithm is $\Omega(\min(m, n))$. (Note the Big-**Omega**.)

Solution

Input family: the two arrays have equal corresponding elements (e.g., both arrays only contains 1's).

In this case, the loop continues until `i` reaches the end of one array, for a total of $\min(m, n)$ iterations. The loop body takes constant time, so the running time for this input family is $\Omega(\min(m, n))$.

[Comment: in fact the running time for this input family is $\Theta(\min(m, n))$. But since we're only proving a lower bound on the worst-case running time, we only need a lower bound on the running time for this input family.]

(b) **[4 marks]** Consider the following input distribution: each item in `array1` and `array2` is independently chosen uniformly at random from the range 1 to 263, inclusive, and both input lists have the same length n. Suppose we only count the number of comparisons `array1[i] != array2[i]` made in the loop.

Find the exact expected number of comparisons made by this algorithm in terms of n.

Hint: use the following formula. For all $r < 1$ and $n \in \mathbb{N}$:

$$\sum_{t=1}^{n} t r^{t-1}(1-r) = \frac{1}{1-r} - r^n \left(n + \frac{1}{1-r}\right).$$

Solution

[Note: this is quite similar to a question on Assignment 1.]

For each $t < n$, the probability that exactly t comparisons are made is $\left(\frac{1}{263}\right)^{t-1} \cdot \frac{262}{263}$ (the first term coming from having the first $t-1$ corresponding elements equal, and the second for stopping at the t-th comparison).

The probability that n comparisons occur is $\left(\frac{1}{263}\right)^{n-1}$. (Some students missed the "x not in A" case on Assignment 1, but did better here.)

So the expected number of comparisons made is

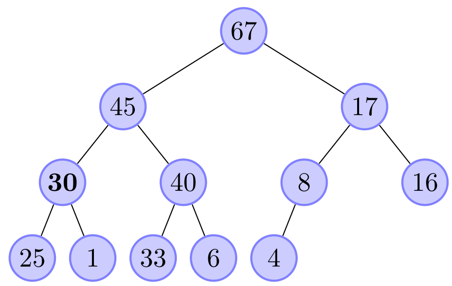
$$\begin{aligned}
&\sum_{t=1}^{n-1} t \left(\frac{1}{263}\right)^{t-1} \cdot \frac{262}{263} + n \left(\frac{1}{263}\right)^{n-1} \\
&= \frac{1}{1-\frac{1}{263}} - \left(\frac{1}{263}\right)^{n-1} \left(n - 1 + \frac{1}{1-\frac{1}{263}}\right) + n \left(\frac{1}{263}\right)^{n-1} \\
&= \frac{263}{262} - \left(\frac{1}{263}\right)^{n-1} \left(n + \frac{1}{262}\right) + n \left(\frac{1}{263}\right)^{n-1} \\
&= \frac{263}{262} - \frac{1}{262} \left(\frac{1}{263}\right)^{n-1}
\end{aligned}$$

2. Assume for this question that we're using heaps to *only* store priorities (and not corresponding data). We will refer to these priorities simply as "values" for this question.

Suppose we want to implement the following heap operation:

- FINDKTHLARGEST($heap, k$): return the k-th largest value in the heap. Do not modify the contents of the heap.

For example, calling FINDKTHLARGEST on the following heap with $k = 5$ would return 30.



(a) [**2 marks**] Consider the following algorithm, which performs k EXTRACTMAX operations, saving the results in a 1-indexed array.

```
def FindKthLargest(heap, k):
    items = new array of length k
    for i = 1 to k:
        items[i] = ExtractMax(heap)
    for i = 1 to k:
        Insert(heap, items[i])
    return items[k]
```

why need to insert here

Let n be the size of the heap, and assume that $1 \leq k \leq n$. Prove that the worst-case running time of this algorithm is $\mathcal{O}(k \cdot \log n)$.

Solution

There are k EXTRACTMAX operations and k INSERT operations. Each is done on a heap of size at most n, so each one's running time is $\mathcal{O}(\log n)$.

So the total cost of the $2k$ operations is $\mathcal{O}(k \cdot \log n)$.

[Comment: it's not enough to show that one of the loops run in $\mathcal{O}(\log n)$ time; both loops must be taken into consideration.]

(b) **[2 marks]** Prove that the k-th largest element in the heap can have an array index of $2^k - 1$, *and* this is the maximum possible index.

Solution

The k-th largest element in the heap can't be at depth greater than k (since it can have at most $k-1$ ancestors, which would have bigger priorities). The largest index of an element at depth k in a heap is $2^k - 1$.

The k-largest element could be at this index if the heap has the property that for each heap node, the priorities of its right descendants are $<$ the priorities of its left descendants.

(c) **[3 marks]** Assume the input heap contains more than 2^k items. Part (b) suggests a different implementation of FindKthLargest, using `heap[i]` to directly access the array storing the heap elements (in constant time).

```
def FindKthLargest(heap, k):
    temp_heap = new, empty heap
    for i from 1 to 2^k - 1:
        Insert(temp_heap, heap[i])
    for i from 1 to k - 1:
        ExtractMax(temp_heap)
    return ExtractMax(temp_heap)
```

Give a tight upper bound on the worst-case running time of this algorithm. You should justify that your upper bound is correct, but you do *not* need to prove that it is tight (i.e., don't prove a matching lower bound).

Solution

The Insert happens $2^k - 1$ times, and each one is put into a heap of size at most $2^k - 1$. The total cost there is at most $\mathcal{O}((2^k - 1)\log(2^k - 1)) = \mathcal{O}(2^k k)$.

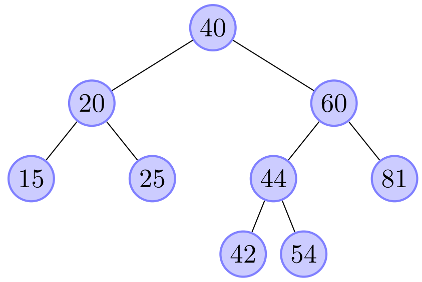
Then k ExtractMax operations happen, each one on a heap of size at most $2^k - 1$, for a cost of $\mathcal{O}(k^2)$.

So the worst-case is upper bounded by $\mathcal{O}(2^k k + k^2) = \mathcal{O}(2^k k)$.

Better. (The above solution was awarded full marks, but can be improved.) Because the items Inserted are already in heap order, no bubble-up swaps will occur. You can think of this as simply copying the top k levels of the heap over to a new heap. So the running time of the Insert loop is actually $\mathcal{O}(2^k)$, leading to an overall running time of $\mathcal{O}(2^k)$.

3. Here are some questions about binary search trees and AVL trees.

(a) [**2 marks**] Consider the following AVL tree:



Suppose we insert a value chosen uniformly at random between 1 and 100 inclusive, using the AVL Tree INSERT algorithm from lecture. Calculate the expected number of **rotations** that will occur for this insertion. You do not need to simplify nor add fractions in your final answer.

You may make the following two assumptions in your work:

- If the chosen value is already in the AVL Tree, no rotations occur.
- At most one node will have an imbalance fixed. That is, the possible number of rotations performed is always between 0 and 2. counting double rotation as 2

Solution

One nice observation is an imbalance only occurs when inserting into the subtree rooted at 44.

- If an item is inserted under 42, 1 right rotation occurs (around 60).
- If an item is inserted under 54, 1 left rotation occurs (around 44), then 1 right rotation occurs (around 60).

So the expected number of rotations is

$$1 \cdot \frac{1+1}{100} + 2 \cdot \frac{9+5}{100}.$$

(b) [**3 marks**] Write an algorithm to return the *second-largest* key in a binary search tree, or `null` if the BST has size < 2. You may assume the BST has no duplicates, and that you have access to a `FindMax` operation on BSTs that runs in $\mathcal{O}(h)$ time, where h is the height of the BST. Your algorithm must run in $\mathcal{O}(h)$ time in the worst case.

No justification or runtime analysis is required for this question; however, you will lose marks if your solution is hard to understand, so please write comments to clarify your work if necessary.

Solution

The key idea is to always try to recurse to the right, stopping only if the right subtree's size is < 2.

```
def SecondLargest(D):
  if D is empty or D is a single node:
    return null
  else:
    second_largest = SecondLargest(D.right)
    if second_largest is null:
      if D.right is empty:       # The root is the largest
        return FindMax(D.left)
      else:
        return D.root.key        # The root is the second-largest
    else:
      return second_largest      # The second-largest was on the right
```

[Comment: a common solution used FindMax to find and delete the largest key to make it easier to find the second-largest. As long as the largest key was re-inserted, this solution is perfectly acceptable.]

(c) [**3 marks**] Suppose we augment binary search trees so that each node stores the size of the subtree rooted at that node (i.e., 1 plus the number of its descendants).

Show how to use this to support the following operation:

- NumInRange(D, a, b): return the number of keys in D that are $\geq a$ and $\leq b$, **assuming** that $a \leq b$.

Your algorithm should do something better than always visit every node in the BST, and in particular make use of both the BST property and the `size` attribute.

In addition to the pseudocode, briefly justify why your solution is correct. You do not need to analyse the running time of your algorithm.

Solution

The key idea is to use the BST property to decide whether to recurse on the left, the right, or both. The `size` attribute helps skip over "middle" subtrees; for example, when a is less than the root and b is greater than the right child, the entire left subtree of the right child can be counted at once.

[Comments: Even though the code appears long, the cases are mostly symmetric by swapping `left` and `right`. This approach can be simplified if we allow access to a `parent` attribute.]

```
def NumInRange(D, a, b):
  if D is empty:
    return 0
  else if b < D.root.key:  # only recurse on left
    return NumInRange(D.left, a, b)
  else if a > D.root.key:  # only recurse on right
    return NumInRange(D.right, a, b)
  else:                   # will need to recurse on both
    # a <= D.root.key and b >= D.root.key, so count the root
    count = 1
    if D.left is not empty:
      if a <= D.left.root.key:
        count += NumInRange(D.left.left, a, b) + D.left.right.size
      else:
        count += NumInRange(D.left, a, b)
    if D.right is not empty:
      if b >= D.right.root.key:
        count += NumInRange(D.right.right, a, b) + D.right.left.size
      else:
        count += NumInRange(D.right, a, b)
    return count
```

how about left.right

4. Let n be a positive integer. Suppose we have an empty binary search tree, and insert n distinct numbers into it, in some order.

   If we use the naïve INSERT algorithm for BSTs, the maximum height of a BST we could get is n. Recall that there are 2^{n-1} permutations of the n distinct numbers that result in a BST of height n when the items are inserted according to this permutation. You may use this fact, without proof, in this question.

   Let C_n be the number of permutations of the n distinct numbers that result in getting a BST of height exactly $n-1$. Note that C_1 and C_2 are both 0.

   (a) **[1 mark]** Determine the value of C_3.

Solution

We can think of this as looking for orders of $\{1, 2, 3\}$ that yield a BST of height 2. There's two of them: $[2, 1, 3]$ and $[2, 3, 1]$.

   (b) **[3 marks]** Find a formula relating C_n and C_{n-1} that is valid for all $n \geq 4$.
   Hint: a permutation that results in getting a BST of height $n-1$ has exactly four possibilities for its first number.

Solution

There are two kinds of orders that are counted by C_n:

- Orders that start with the smallest/largest value.
- Orders that start with the second smallest/second largest value.

From the first group, the remaining $n-1$ values must be inserted in an order that results in a tree of height $n-2$; there are C_{n-1} such orders.

From the second group, the remaining items are divided into two groups: the single element that is smaller/greater than the starting value (when it is the second smallest or second largest, respectively), and the remaining $n-2$ items. These $n-2$ items must be inserted in an order that results in a BST of height $n-2$. There are 2^{n-3} such permutations, and the single element can be inserted into each order in $n-1$ different spots, for a total of

$$C_n = 2C_{n-1} + 2 \cdot 2^{n-3} \cdot (n-1).$$

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

Name:

	Q1	Q2	Q3	Q4	Total
Grade					
Out Of	6	7	8	4	25