Homework 03: Solutions

Due date: Monday, 7/23 at 11:59 pm

Instructions:

Submit a typed or neatly handwritten scan of your responses on Canvas in PDF format.

Note: you will need to submit a separate PDF per each section.

1. Big-O Notation

Let $f(n) = 10^6 \cdot n^2 + 10^8 \cdot n^{1.5} + n^3 + 10^{10} \cdot n^{2.99}$. Show that $f(n) = O(n^3)$ by finding a constant c and an integer n_0 and applying the Big-O definition. **Solution:**

We will show that for every $n \geq 1$, $f(n) \leq 4 \cdot 10^{10} \cdot n^3$. By applying the Big-$O$ definition with $n_0 = 1$ and $c = 4 \cdot 10^{10}$, this will imply that $f(n) = O(n^3)$. For every $n \geq 1$, we have

$$\begin{aligned} f(n) &= 10^6 \cdot n^2 + 10^8 \cdot n^{1.5} + n^3 + 10^{10} \cdot n^{2.99} \\ &\leq 10^6 \cdot n^3 + 10^8 \cdot n^3 + n^3 + 10^{10} n^3 \\ &\leq 4 \cdot 10^{10} \cdot n^3, \end{aligned}$$

which concludes the proof.

2. Solving Recurrences

(a) For the following recurrence relations, state which case of the Master Theorem applies and give the Big-Θ runtime bound.

(i)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7 \cdot T(n/2) + n^2 & \text{otherwise} \end{cases}$$

Solution:

Since $\log_2 7 > 2$, Case 3 of the Master Theorem implies that $T(n) = \Theta(n^{\log_2 7})$.

(ii)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 4 \cdot T(n/2) + n^2 & \text{otherwise} \end{cases}$$

Solution:

Since $\log_2 4 = 2$, Case 2 of the Master Theorem implies that $T(n) = \Theta(n^2 \log n)$.

(iii)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot T(n/2) + \sqrt{n} & \text{otherwise} \end{cases}$$

Solution:

Since $\log_2 2 > 1/2$, Case 3 of the Master Theorem implies that $T(n) = \Theta(n)$.

(iv)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 4 \cdot T(n/2) + n^3 & \text{otherwise} \end{cases}$$

Solution:

Since $\log_2 4 < 3$, Case 1 of the Master Theorem implies that $T(n) = \Theta(n^3)$.

(v)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3 \cdot T(n/2) + n & \text{otherwise} \end{cases}$$

Solution:

Since $\log_2 3 > 1$, Case 3 of the Master Theorem implies that $T(n) = \Theta(n^{\log_2 3})$.

(b) Consider the recurrence

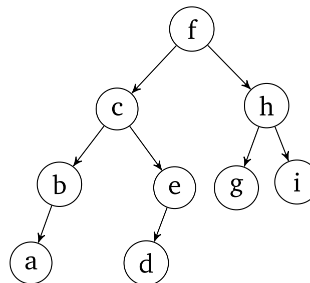
$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2 \cdot T(\sqrt{n}) + \log n & \text{otherwise} \end{cases}$$

Solve the above recurrence using the tree method. First unroll two levels of the tree and draw this unrolling as a tree. Finish your analysis by completing the missing entries of this table.

Calculations for level of base case: We want $n^{\frac{1}{2^i}} = 2$, where i denotes the level of the base case. In other words, $2^i = \log n$. Therefore, $i = \log \log n$.

3. Binary Search Trees

(a) Consider the following Binary Search tree:



Write down the

(i) inorder traversal **Solution:**

a,b,c,d,e,f,g,h,i

(ii) preorder traversal **Solution:**

# of nodes at level i	2^i
input size at level i	$n^{\frac{1}{2^i}}$
work per node at level i	$\log\left(n^{\frac{1}{2^i}}\right) = \frac{\log n}{2^i}$
total work at level i	$2^i \cdot \frac{\log n}{2^i} = \log n$
level of base case	$\log \log n$
number of nodes in the base case level	$2^{\log \log n} = \log n$
expression for recursive work	$T(n) = 2^i T\left(n^{\frac{1}{2^i}}\right) + \sum_{j=0}^{i-1} 2^j \log n^{\frac{1}{2^j}}$
expression for non-recursive work	$T(n) = \log n + \sum_{i=0}^{\log \log n - 1} \log n$
closed form for total work	$\log n \cdot (\log \log n + 1)$
simplest Big-Θ for the total work	$\Theta(\log n \cdot \log \log n)$

f,c,b,a,e,d,h,g,i

(iii) postorder traversal **Solution:**

a,b,d,e,c,g,i,h,f

Do you notice an interesting property of the in order traversal? What is it? **Solution:**

Inorder traversal outputs elements in sorted order.

(b) Let a binary search tree be defined by the following class:

```
public class IntTree {
    private IntTreeNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntTreeNode {
        public int data;
        public IntTreeNode left;
```

```
            public IntTreeNode right;

            // constructors omitted for clarity
        }
    }
```

In class, we saw how to search for an element in a binary search tree. This question will demonstrate that binary search trees are more powerful. Modify the definition of IntTreeNode class to also compute the k'th smallest element in the tree. (k, the input, is a number in $\{1, 2, \cdots, n\}$, where n is the number of nodes in the tree). Also, discuss the algorithm that computes the k'th smallest element. **Solution:**

The modified class stores the number of nodes in the left and right sub-trees.

```
private class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;
        public int sizeleft;
        public int sizeright;

                // constructors omitted for clarity
}
```

We shall now outline the algorithm. Given k and the root node, with the *sizeleft* and *sizeright* values, we check whether the k'th smallest element is the root or in the left sub-tree or right sub-tree. It is the root when $k = \text{sizeleft} + 1$. If it is in the left sub-tree (when $k \leq \text{sizeleft}$), we recurse to find the k'th smallest element in the left sub-tree. Otherwise, we recurse to find the $k - \text{sizeleft} - 1$'th smallest element in the right sub-tree. This completes the description of the algorithm. Clearly, every execution of the algorithm traverses a path from the root to a leaf, and hence the runtime is $O(h)$.

4. AVL Tree Implementation

Write pseudocode for the AVL tree methods Balance, RotateLeft, and RotateRight. Assume that the rest of the data structure is implemented as in the Java code here `https://courses.cs.washington.edu/courses/cse373/18su/files/homework/AVLTree.java`.

You may use the skeleton of Balance from that code as a guide. Note you are not required to write your solution in Java, pseudocode is sufficient.

Solution:

```
private Node balance(Node t) {
        if (t == null) {
            return t;
        }

        if (height(t.left) - height(t.right) > 1 ) { // left tree too tall

            if (height(t.left.right) > height(t.left.left) ) { // left-right case
                t.left = rotateLeft(t.left); //Turn into left-left case
            }

            t = rotateRight(t); // Handle left-left case
        } else if (height(t.right) - height(t.left) > 1 ) { // right tree too tall
```

```
            if (height(t.right.left) > height(t.right.right)) { // right-left case
                t.right = rotateRight(t.right); // Turn into right-right case
            }

            t = rotateLeft(t); // Handle right-right case
        }
        t.height = 1 + Math.max(height(t.right), height(t.left));
        return t;
    }

    private Node rotateLeft(Node t) {
     /*
        t                x
       / \              / \
      a   x   ===>     t   c
         / \          / \
        b   c        a   b
     */
        Node x = t.right;
        Node b = x.left;
        x.left = t;
        t.right = b;
        t.height = 1 + Math.max(height(t.left),height(b));
        x.height = 1 + Math.max(height(t), height(x.right));
        return x;
    }

    private Node rotateRight(Node t) {
        Node x = t.left;
        Node b = x.right;
        x.right = t;
        t.left = b;
        t.height = 1 + Math.max(height(t.right), height(b));
        x.height = 1 + Math.max(height(x.left), height(t));
        return x;
    }
```

5. Hashing

Let the capacity of the hash table be 10 and the hash function be $h(x) = x$. Insert elements

$$42, 102, 12, 33, 25, 14, 62$$

to a hash table

(a) that uses linear probing **Solution:**

0	1	2	3	4	5	6	7	8	9
		42	102	12	33	25	14	62	

The total number of collisions is 15.

(b) that uses quadratic probing **Solution:**

0	1	2	3	4	5	6	7	8	9
	62	42	102	33	25	12		14	

The total number of collisions is 9.

Write down the total number of collisions and the hash table after all insertions in both cases. Why does the secondary clustering in quadratic probing less problematic than the primary clustering in linear probing? **Solution:**

If a hashed value is anywhere in a primary cluster, we need to probe over the entire rest of the cluster. In quadratic probing, we only hit a cluster if the hashed value lands exactly on the start of the cluster.