CSE 332: Data Abstractions

Section 3: BSTs, Recurrences, and Amortized Analysis Solutions

0. Interview Question: Binary Search Trees

Write pseudo-code to perform an in-order traversal in a binary search tree without using recursion.

Solution:

This algorithm is implemented as the BST Iterator in P2. Check it out!

1. Big-Oh Bounds for Recurrences

For each of the following, find a Big-Oh bound for the provided recurrence.

(a) $T(n) = \begin{cases} 1 & \texttt{if } n = 1 \\ 8T(n/2) + 4n^2 & \texttt{otherwise} \end{cases}$

Solution:

Note that $a = 8$, $b = 2$, and $c = 2$. Since $\log_2(8) = 3 > 2$, we have $T(n) \in \Theta(n^{\log_2(8)}) = \Theta(n^3)$ by Master Theorem.

(b) $T(n) = \begin{cases} 1 & \texttt{if } n = 1 \\ 7T(n/2) + 18n^2 & \texttt{otherwise} \end{cases}$

Solution:

Note that $a = 7$, $b = 2$, and $c = 2$. Since $\log_2(7) = 3 > 2$, we have $T(n) \in \Theta(n^{\log_2(7)})$ by Master Theorem.

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

Solution:

Note that $a = 1$, $b = 2$, and $c = 0$. Since $\log_2(1) = 0 = 2$, we have $T(n) \in \Theta(\lg(n))$ by Master Theorem.

2. Recurrences and Closed Forms

For the following code snippet, find a recurrence for the worst case runtime of the function, and then find a closed form for the recurrence.
Consider the function g:

```
g(n) {
   if (n == 1) {
      return 1000;
   }
   if (g(n/3) > 5) {
      for (int i = 0; i < n; i++) {
         System.out.println("Yay!");
      }
      return 5 * g(n/3);
   }
   else {
      for (int i = 0; i < n * n; i++) {
         System.out.println("Yay!");
      }
      return 4 * g(n/3);
   }
}
```

- Find a recurrence for $g(n)$.

Solution:

$$T(n) = \begin{cases} c_0 & \texttt{if } n = 1 \\ 2T(n/3) + c_1 n & \texttt{otherwise} \end{cases}$$

- Find a closed form for $g(n)$.

Solution:

The recursion tree has height $\log_3(n)$. Level i has work $\left(\frac{c_1 n 2^i}{3^i}\right)$.

So, putting it together, we have:

$$\sum_{i=0}^{\log_3(n)-1} \left(\frac{c_1 n 2^i}{3^i}\right) + 2^{\log_3(n)} c_0 = c_1 n \sum_{i=0}^{\log_3(n)-1} \left(\frac{2}{3}\right)^i + n^{\log_3(2)} c_0 \quad = c_1 n \left(\frac{1 - \left(\frac{2}{3}\right)^{\log_3(n)}}{1 - \frac{2}{3}}\right) + n^{\log_3(2)} c_0$$

$$= 3c_1 n \left(1 - \left(\frac{2}{3}\right)^{\log_3(n)}\right) + n^{\log_3(2)} c_0$$

$$= 3c_1 n \left(1 - \frac{n^{\log_3(2)}}{n}\right) + n^{\log_3(2)} c_0$$

3. MULTI-pop

Consider augmenting the `Stack` ADT with an extra operation:

> `multipop(k)`: Pops up to k elements from the `Stack` and returns the number of elements it popped

What is the amortized cost of a series of `multipop`'s on a `Stack` assuming `push` and `pop` are both $\mathcal{O}(1)$?

Solution:

Consider an *empty* `Stack`. If we run various operations (`multipop`, `pop`, and `push`) on the `Stack` until it is once again empty, we see the following: Note that `multipop(k)` takes ck time. If over the course of running the operations, we `push` n items, then each item is associated with *at most* one `multipop` or `pop`. It follows that the largest number of time the `multipop`s can take in aggregate is n. Note that the *smallest possible number of operations to amortize over* is $n+1$ (n `push`es and 1 `multipop`). So, the worst amortized cost of a series of `push`es, `pop`s, and `multipop`s is $\dfrac{2n}{n+1} = \mathcal{O}(1)$.