CSE 332: Data Structures and Parallelism

BSTs, Recurrences, and Amortized Analysis 3 Solutions

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!

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.

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:

$$T(n) = \begin{cases} c_0 & \texttt{if } n \le 1 \\ 2T(n/3) + c_1 n + c_2 & \texttt{otherwise} \end{cases}$$

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

  Solution:

  The recursion tree has height $\log_3(n)$. Level i has work $\left(\frac{c_1 n 2^i}{3^i} + 2^i c_2\right)$.

  So, putting it together, we have:

$$\begin{aligned} \sum_{i=0}^{\log_3(n)-1} \left(\frac{c_1 n 2^i}{3^i} + 2^i c_2\right) + 2^{\log_3(n)} c_0 &= c_1 n \sum_{i=0}^{\log_3(n)-1} \left(\frac{2}{3}\right)^i + \sum_{i=0}^{\log_3(n)-1} 2^i c_2 + n^{\log_3(2)} c_0 \\ &= c_1 n \left(\frac{1 - \left(\frac{2}{3}\right)^{\log_3(n)}}{1 - \frac{2}{3}}\right) + c_2 \left(\frac{1 - 2^{\log_3(n)}}{1 - 2}\right) + n^{\log_3(2)} c_0 \end{aligned}$$

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 $\mathcal{O}(1)$?

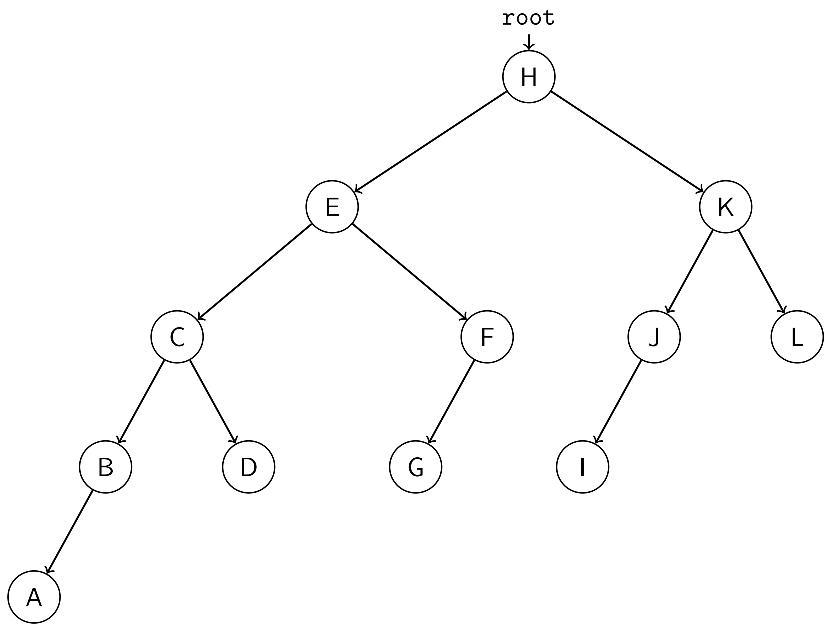
Solution:

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 `multipops` can take in aggregate is n. Note that the *smallest possible number* *of operations to amortize over* is $n+1$ (n `pushes` and 1 `multipop`). So, the worst amortized cost of a series of `pushes`, `pops`, and `multipops` is $\dfrac{2n}{n+1} = \mathcal{O}(1)$.

MinVL Trees

Draw an AVL tree of height 4 that contains the minimum possible number of nodes.

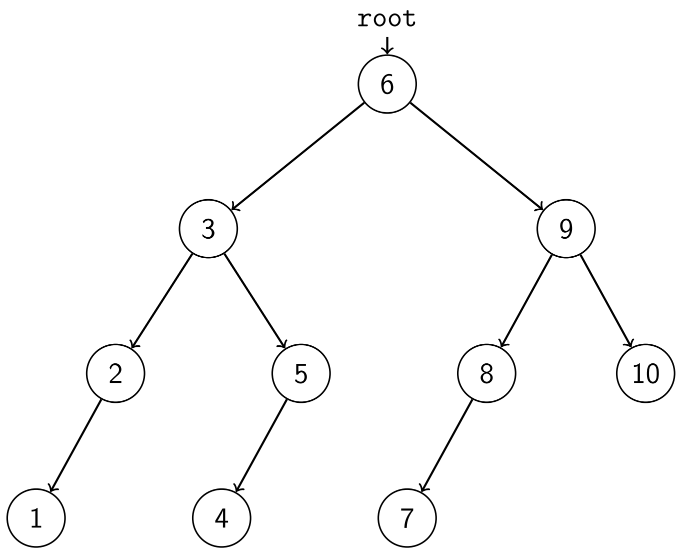
Solution:



AVL Trees

Insert 6, 5, 4, 3, 2, 1, 10, 9, 8, 7 into an initially empty AVL Tree.

Solution:



AVL Trees

Given a binary search tree, describe how you could convert it into an AVL tree with worst-case time $\mathcal{O}(n \lg(n))$. What is the best case runtime of your algorithm?

Solution:

Since we already have a BST, we can do an in-order traversal on the tree to get a sorted array of nodes. We could now simply insert all of these nodes back into an AVL tree using rotations which would give us an $\mathcal{O}(n \lg(n))$ runtime.

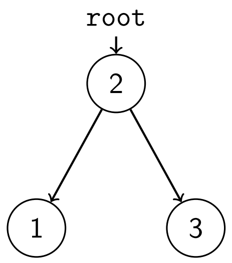
Another option is to do a binary search on this sorted array of nodes. Make a queue and insert arr[n/2] as the root. Then take it out and insert (root, left) and (root, right). We can continue to remove from the queue, updating the node, then adding it's children to the queue to be updated as well. If we continue this pattern, each insert will be $\mathcal{O}(1)$ giving us a $\mathcal{O}(n)$ worst-case runtime.

HeapVL Trees

Is there an AVL Tree that isn't a heap? Is there a heap that isn't an AVL tree? Is there a binary search tree that is neither? Is there a binary search tree that is both?

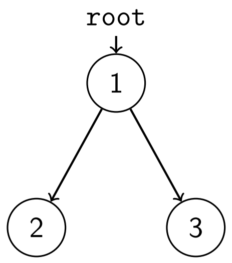
Solution:

Is there an AVL Tree that isn't a heap? Yes. Consider the following:



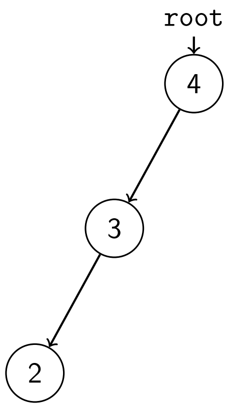
This tree meets the AVL properties - left children are smaller, right children are larger, and the tree is balanced. But it does not meet following property of a max-heap - all children must be larger.

Is there a heap that isn't an AVL tree? Yes. Consider the following:



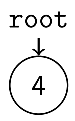
This meets the properties of a heap, but is not in the correct format for an AVL tree since the left child should be smaller than the root value.

Is there a binary search tree that is neither? Yes. Consider the following:



This BST is unbalanced, so it is not an AVL tree. It also contains gaps across the 2nd level, so it cannot be a heap since it doesn't meet the structure property.

Is there a binary search tree that is both? Yes. Consider the following:

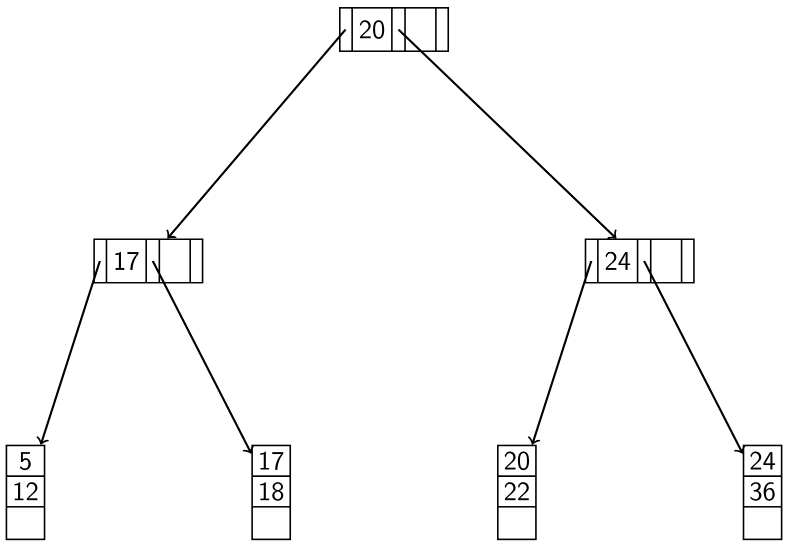


This is both a valid heap and a valid AVL tree!

B-Trees

(a) Insert the following into an empty B-Tree with $M = 3$ and $L = 3$: $12, 24, 36, 17, 18, 5, 22, 20$.

Solution:



(b) Delete $17, 12, 22, 5, 36$

Solution:

18
20
24

(c) Given the following parameters for a B-Tree with $M = 11$ and $L = 8$

- Key Size = 10 bytes
- Pointer Size = 2 bytes
- Data Size = 16 bytes per record (includes the key)

Assuming that M and L were chosen appropriately, what is the likely page size on the machine where this implementation will be deployed? Give a numeric answer and a short justification based on two equations using the parameter values above.

Solution:

We use the following two equations to find M and L to fit as best as possible in the page size, where:

- 1 page on disk is p bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

$M = \left\lfloor \frac{p+k}{t+k} \right\rfloor$ and $L = \left\lfloor \frac{p}{v} \right\rfloor$

Plugging in the given values, we get:

$M = \left\lfloor \frac{p+10}{2+10} \right\rfloor$ and $L = \left\lfloor \frac{p}{16} \right\rfloor$

And solving for p gives us an answer of 128 bytes.