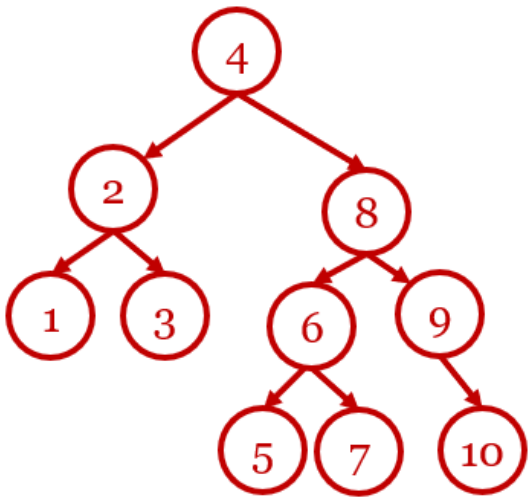
Homework 4 sample solution

Due 10/03/2024

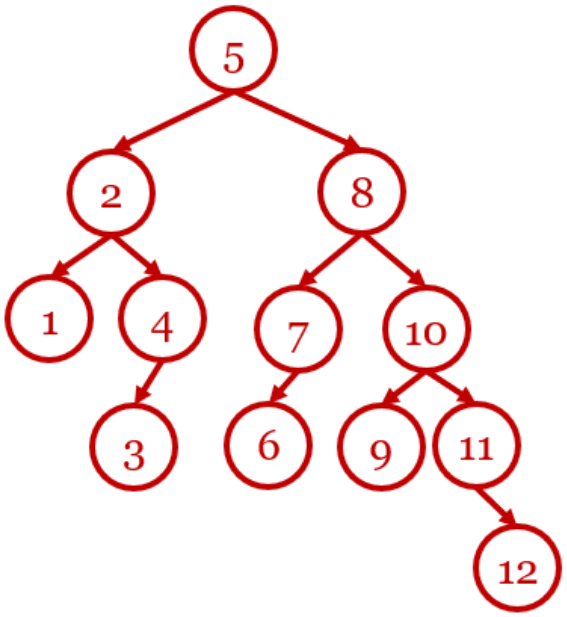
October 4, 2024

1. Sketch the contents of an AVL tree after adding all of the values 1–10 in sorted order to an empty AVL tree.



2. Sketch an AVL tree that contains values 1–n (where n is the number of nodes in your tree) in which removing one node results in *exactly* three rotations: right, then left, then left again. Clearly indicate which node in your tree has this property.

   Smallest possible tree (delete node 1):



The subtrees rooted at 7 and 10 could be as large as complete trees, as long as they have the same height.

It's also possible to build the tree so that it performs 3 single rotations instead of a double rotation followed by a single rotation, though this tree will be larger.

3. List all coefficients c such that the hash function $h(x) = cx \bmod capacity$ can potentially return any index when $capacity = 24$. Only list coefficients in the range 0 to 23 (inclusive).

   The coefficients relatively prime to 24 are 1, 5, 7, 11, 13, 17, 19, and 23. All of these coefficients will produce every index equally often.

4. Draw the contents of a hash table with capacity 20 using open addressing with linear probing and the hash function $h(x) = 7x + 5 \bmod 20$ after inserting the values 1, 5, 11, 18, 3, and 8 into an empty hash table.

5	8	11				3					18	1							

5. Describe an efficient implementation of a Set ADT that combines the advantages of a hash table and a balanced binary search tree. In particular, the search function for your data structure should have $O(\lg n)$ worst case complexity and $\Theta(1)$ expected complexity, and it should not be limited to values in a small range. Your description should include:

   (a) how you are storing data in memory,

   (b) how you search for elements, and

(c) how you insert new elements into the data structure.

Hint: your description should relate to existing data structures. Do not reproduce pseudocode for any of the operations for any structure discussed in class. Instead, describe how you would modify these operations.

There is more than one way to approach this problem. One solution would be to implement a hash table with separate chaining, except that each node is the root of a balanced BST. Search and insert as in a hash table with separate chaining, except that you should perform a BBST search or insertion on collision. If there are $\Theta(1)$ hash collisions, each BST will have $O(1)$ elements and will take $\Theta(1)$ time to search. In the worst case, every element is in the same BBST, which can be searched in $O(\lg n)$.

Another solution is to store both a balanced BST and hash table (using either separate chaining or open addressing). To insert a value, hash it, and search at that index in the hash table. If that index is occupied, see if you can insert it as the root node of the BBST. Alternate between scanning the next position in the hash table and testing the next level down in the BBST. (Note: if using separate chaining, it's important to insert at the end of the list rather than the beginning so that the number of steps required to find something in the hash table doesn't increase over time.) Searching is similar, alternating between searching the hash table and the BBST. The search continues until you locate the element or the BBST search fails (the hash table will never insert later than the BBST). If the hash function exhibits $\Theta(1)$ collisions, you will test an expected $\Theta(1)$ hash table entries and $\Theta(1)$ BBST nodes before finding the element, but in the worst case, this may take $O(\lg n)$ to scan the BBST.