CS2040: Data Structures and Algorithms

Tutorial Problems for Week 9: BST and bBST

For: 17 October 2024, Tutorial 7

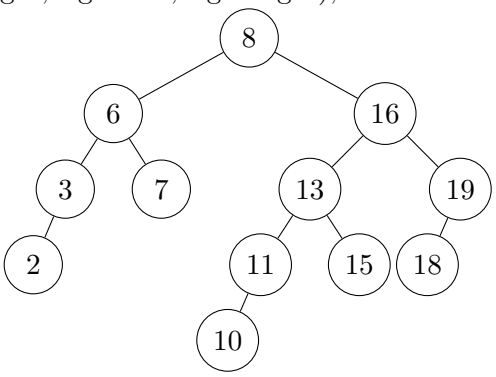
Solution: Secret! Shhhh... This is the solutions sheet.

Problem 1. True or False?

Decide whether each of the following statements is true or false. Provide a short justification for your choice.

Problem 1.a. (AY19/20 Sem 1 Final Exam) Given any AVL tree of height 4, deleting any vertex in the tree will not result in more than 1 rebalancing operation (not rotation but rebalancing operations!).

Solution: False. For the binary search tree below (found in the lecture notes), deleting vertex 7 triggers two rebalancing operations. **Important:** Note that a *rebalancing operation* is not the same as a *rotation*: a rebalancing operation is one of the four rebalancing cases mentioned in the lecture notes (left-left, left-right, right-left, right-right), and can consist of more than one rotation.



Problem 1.b. The minimum number of vertices in an AVL tree of height 5 is 21.

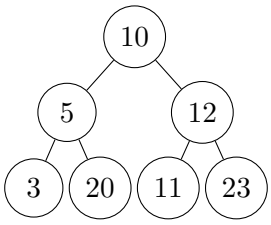
Solution: False. The minimum is 20. The minimum number of vertices n_h in an AVL tree of height h is given by the recurrence

$$n_0 = 1$$
$$n_1 = 2$$
$$n_i = n_{i-1} + n_{i-2} + 1$$

Using the recurrence, we have $n_0 = 1, n_1 = 2, n_2 = 4, n_3 = 7, n_4 = 12, n_5 = 20$

Problem 1.c. In a tree, if for every vertex x that is not a leaf, $x.left.key < x.key$ if x has a left child and $x.key < x.right.key$ if x has a right child, the tree is a BST.

Solution: False. Consider the following binary tree. Note that the above statement holds for every node in the binary tree, but the binary tree is not a BST. For a binary tree to be a BST, you need all vertices y in the left subtree of x to have $y.key < x.key$ and not just the left child y of x having $y.key < x.key$. Similar argument for vertices in the right subtree of x and the right child of x.



Problem 2. In-Order Traversal

The in-order traversal visits nodes in a BST in sorted order. The following implementation of in-order traversal that prints the nodes in a balanced BST in sorted order has been proposed.

Algorithm 1 In-Order Traversal

```
1: procedure InOrderTraversal(T)                    ▷ bBST T is supplied as input
2:     currentNodeValue ← T.findMin()
3:     while currentNodeValue ≠ −1 do
4:         output currentNodeValue
5:         currentNodeValue ← T.successor(currentNodeValue)
6:     end while                                    ▷ successor returns -1 if there is no successor
7: end procedure
```

Answer the following questions assuming that T is a balanced BST.

Problem 2.a. What is the running time of Algorithm 1?

Solution: $O(n \log n)$

Problem 2.b. Propose modifications to the *successor* function such that Algorithm 1 runs in $O(n)$ time.

Solution: Notice that in a standard in-order traversal, we perform something that is similar to Algorithm 1 – we go one node to its successor, then from the next node, we go to its successor again, and so on, until we visit all nodes in the BST. However, the difference is that in a standard in-order traversal, we move directly from one node to its successor, instead of locating its successor starting from the root node.
Hence, we can modify Algorithm 1 such that it works in a way similar to the standard in-order traversal. In particular, for the successor function, we can get it to accept a reference to the node that we want to find the successor of, so that we can begin searching for the successor from the node itself, rather than from the root node.

Problem 3. Rank and Select

A node x has rank k in a BST if there are $k-1$ nodes that are smaller than x in the BST. The *rank* operation finds the rank of a node in a BST.

Problem 3.a. Describe an algorithm that finds the rank of a given node in the BST in $O(h)$ time, where h is the height of the BST.

Solution: Augment every node in the BST with the size of its subtree. Then, we can compute the rank of a given node using the following algorithm.

Algorithm 2 BST Rank

```
1: function RANK(node, v)
2:     if node.key = v then
3:         return node.left.size + 1
4:     else if node.key > v then
5:         return RANK(node.left, v)
6:     else
7:         return node.left.size + 1 + RANK(node.right, v)
8:     end if
9: end function
```

The *select* operation returns the node with rank k in the BST.

Problem 3.b. Describe an algorithm that finds the node with rank k in the BST in $O(h)$ time, where h is the height of the BST.

Solution: Once again, we augment every node in the BST with the size of its subtree. Then, we can find the node with rank k using the following algorithm.

Algorithm 3 BST Select

```
1: function SELECT(node, k)
2:     q ← node.left.size
3:     if q + 1 = k then
4:         return node.key
5:     else if q + 1 > k then
6:         return SELECT(node.left, k)
7:     else
8:         return SELECT(node.right, k − q − 1)
9:     end if
10: end function
```

Problem 4. Transmission System

There are n servers in a line, numbered from 1 to n. A server can either be enabled or disabled, only enabled servers will send data. Server i has been configured in a way such that it can only send data directly to server $i+1$, $1 \leq i < n$. For $i < j$, server i can send data indirectly to server

j, as long as both servers are enabled, and all servers between them (if any) are all enabled as well. Initially, all servers are enabled.

You need to support q of the following three types of operations:

- *Enable*(i): Enable the ith server. If the ith server is already enabled, nothing happens.
- *Disable*(i): Disable the ith server. If the ith server is already disabled, nothing happens.
- *Send*(i, j): Return *true* if it is possible to send data from server i to j, $false$ otherwise.

Describe the most efficient algorithm you can think of for each of the three types of operations. What is the running time of your algorithm for each of the three types of operations?

Solution 1: We can use a bBST to maintain the servers that are currently disabled. When a server is disabled, we insert it into the bBST, and when a server is enabled, we remove it from the bBST. To check if we are able to send a message from server i to server j, we need to check that i is not in the bBST, and either i has no successor in the bBST, or the successor of i in the bBST is greater than j.

Here, we note that the successor of x, or *successor(x)*, refers to the node in the bBST containing the smallest key that is greater or equal to x. This definition of successor is similar to the `higherEntry` method in the Java `TreeMap`. A simple way to implement this is to first do a search to check for existence, then do an insertion and call the usual successor, and then doing a deletion. However, there is way to do this directly without any insertions or deletions by modifying the search operation, and is left as an exercise.

Algorithm 4 Solution to Problem 4

```
1: T ← bBST, initially empty
2: procedure ENABLE(i)
3:     T.delete(i)
4: end procedure
5: procedure DISABLE(i)
6:     T.insert(i)
7: end procedure
8: function SEND(i, j)
9:     if i is in T or T.successor(i) ≤ j then
10:        return false
11:    else
12:        return true
13:    end if
14: end function
```

Solution 2: Another even simpler way of implementing the 3 operations is as follows. Maintain a bBST that contains the servers which are currently enabled. When a server is enabled, insert it into the bBST if it is not already in the bBST. Similarly when a server is disabled, remove it from the bBST if it is not already removed. To check if we can send a message from server i to

server j, first check if both i and j exist in the bBST. If both exist, get rank(i) and rank(j). If rank(j)-rank(i) = j-i then i can send a message to j since every server in between must be enabled. Otherwise i cannot send a message to j.

Problem 5. Lowest Common Ancestor (CS2040S AY19/20 Sem 1, Quiz 2)

The Lowest Common Ancestor (LCA) of two nodes a and b in a BST is the node furthest from the root that is an ancestor of both a and b. For example, given the Binary Tree shown in Fig 1, the LCA for 2 and 6 is 5. The LCA of 12 and 2 is 10. Note that a node is an ancestor of itself. That means the LCA of 5 and 7 is 5. Similarly, the LCA of 18 and 14 is 18.

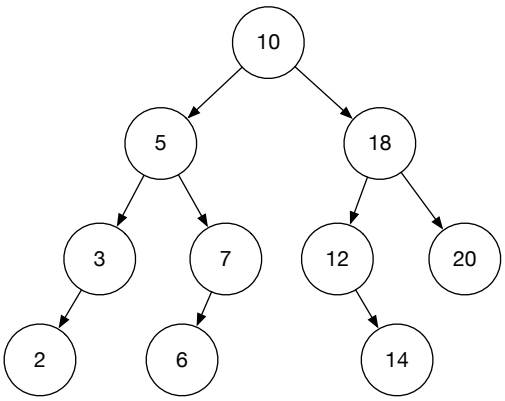


Figure 1: An example BST.

Consider a **Balanced Binary Search Tree** (bBST) T containing n nodes with *unique* keys, where **nodes do *not* have parent pointers** (i.e. given a node, you *cannot* find its parent in $O(1)$ time).

Describe the most efficient algorithm you can think of to find the LCA of two given nodes in the bBST. What is the running time of your algorithm?

Solution: Here, we can exploit the property of a Binary Search Tree (BST). At any node, there are essentially four cases to consider:

- If either a or b is the same value as the current node, we can stop and we have found our LCA.
- If a and b are both smaller than the value at the current node, we move to the left child of the current node and it becomes the new current node.
- If a and b are both greater than the value at the current node, we move to the right child of the current node and it becomes the new current node.
- If a is smaller than the value at the current node, and b is greater than the value at the current node, then the current node is the LCA. This is also true if we swap the positions of a and b.

The algorithm takes $O(\log n)$ time and $O(1)$ space, since the BST is now balanced.