CS2040C Semester 1 2018/2019

Data Structures and Algorithms

Tutorial 08 - Binary Search Trees, AVL Trees

For Week 10 (Week Starting 22 October 2018)

Document was last modified on: October 18, 2018

1 Introduction and Objective

This tutorial marks the end of the discussion on data structures in CS2040C. We will reinforce the concepts of Binary Search Tree (BST) and the importance of having a balanced BST. Adelson-Velskii Landis (AVL) Tree is only one of the many possible ways to balance a BST.

2 Tutorial 08 Questions

Q1). Revise AVL Tree on VisuAlgo `https://visualgo.net/en/bst` and think about the following questions listed in the 'e-Lecture Mode'. Tutor will go through the solutions to these questions in class.

1. When removing a vertex **B** which has 2 children, why is it that replacing **B** with its successor **C** is always a valid strategy? `https://visualgo.net/en/bst?slide=10-4` (With reference to the lecture notes, the successor in this case will always be the minimum item of the right subtree.)

2. Can we replace **B** with its predecessor **A** instead? Why or why not? `https://visualgo.net/en/bst?slide=10-4` (With reference to the lecture notes, the predecessor in this case will always be the maximum element of the left subtree.)

3. Is there only 1 tree rotation case for any Insert(v) operation on an AVL tree? `https://visualgo.net/en/bst?slide=14-11`

Q1.1 and Q1.2) We claim that vertex C, which is the successor of vertex B that has two children, must only have at most one child (which is an easier removal case).

Vertex B has two children, so B must have a right child. Let's name it R. Successor of B must be

the minimum vertex of subtree rooted at R.
Remember that the minimum element of a subtree in BST has no left child (it may have right child). Thus, C, the successor of B has at most one child.
Before removal, we have X (may be empty) $< A < B < C < Z$ (may be empty) in the BST. Replacing B with its successor C then deleting the old and duplicate C will maintain the BST properties of all vertices involved. Similarly replacing B with its predecessor A will also achieve the same result. We just need to be consistent.
Q1.3) There is no other case.
Insert(v) may indeed cause more than one vertex along the insertion path to have its height increase by one unit.
However, as soon as we fix the lowest vertex x that is out-of-balance by a bit (+2 or -2), the subsequent one (or double) rotation(s) will make the height of vertex x decreases by one again. Thus, any other vertices above x along the insertion path will no longer be out-of-balance anymore.

Q2). Draw a valid AVL Tree and nominate a vertex to be deleted such that if that vertex is deleted:
a). No rotation happens
b). Exactly one of the four rotation cases happens
c). Exactly two of the four rotation cases happens (you can**not** use the sample given in VisuAlgo which is `https://visualgo.net/en/bst?mode=AVL&create=8,6,16,3,7,13,19,2,11,15,18,10`, delete vertex 7; think of your own test case)

This part is also open ended and there are many possible answers. The tutor will help students construct the answer based on the idea of minimum-sized AVL Tree.

Extra BST Operations

Q3). There are two important BST operations: Rank and Select that are not included in VisuAlgo yet but useful for PS4. Please discuss on how to implement these two operations efficiently.

1. `Rank(v)`: Given a key v, determine what is its rank (1-based index) in the sorted order of the BST elements. That is, Rank(FindMin()) = 1 and Rank(FindMax()) = N. If v does not exist, we can report -1.

2. `Select(k)`: Given a rank k, $1 \leq k \leq N$, determine the key v that has that rank k in the BST. Or in another word, find the k-th smallest element in the BST. That is, Select(1) = FindMin() and Select(N) = FindMax().

In pseudo code with some edge cases missing. For eg, what if v for rank does not exist in the tree?

```
int rank(node, v) { // assume that v exists in the BST and size attribute is there
       if (node.key == v) return node.left.size + 1; // this is discussed in Lec04
  else if (node.key <  v) return rank(node.left, v); // v must be on the left
  else                    return node.left.size+1 // v is > node's left and the node
                                 + rank(node.right, v); // and plus this rank
}

// select is very similar to rank and similar with QuickSelect from tut01... :O
int select(node, k) { // assume size attribute is there
  int q = node.left.size;
       if (q+1 == k) return node.key; // this node has rank k
  else if (q+1 >  k) return select(node.left, k); // rank k is in the left subtree
  else               return select(node.right, k-q-1); // do you understand why?
}
```