MATH 182: Homework #6

Professor Wotao Yin

Eric Chuu

UID: 604406828

Assignment: 3.2.3, 3.2.6, 3.3.2, 3.3.4, 3.3.13

February 27, 2017

Question 3.2.3

Give five orderings of the keys `A X C S E R H` that when inserted into an initially empty BST, produce the best-case tree.

Solution

The best-case tree for 7 inputs is one that fully populated on each level. In this case, we would want the tree to be rooted at `H`, so `A C E` are in the left sub-tree and `R S X` are in the right sub-tree. The respective sub-trees would then be rooted in the middle elements, `C, S`. We can achieve this tree with the following five orderings of the keys:

[H C S A E R X]

[H S C A E R X]

[H S R X C A E]

[H C A E S R X]

[H C A S E R X]

□

Question 3.2.6

Add to `BST` method `height()` that computes the height of the tree. Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like `size()` that adds a field to each node in the tree (and takes linear space and constant time per query).

Solution

The code for both implementations is given below. The recursive routine is called `height1()` and the non-recursive routine is called `height2()`. Note that `height2()` relies on adding a field, `h`, to every node, which is equal to the height of the sub-tree rooted at that node. In order to accommodate this feature, we add an extra line in the `put()` method, which is also shown below with the unchanged portion of the code truncated. This allows the `height2()` method achieve constant time per query. □

```
// both height() methods return the hieght of tree rooted at input node

// recursive implementation of height()
private static int height1(Node node) {
   if (node == null) {
      // implementation counts ahead, so decrement at leaf
      return -1;
   }

   return 1 + Math.max(height1(node.left), height1(node.right));
}

// non-recursive implementation of height()
private static int height2(Node node) {
   if (node == null) {
      return 0;
   }
   return node.h;
}

// We keep the original implementation of put, with an added line
private Node put(Node node, Key key, Value val) {
   if (node == null) {
      return new Node(key, val, 1);
   }

   /** implementation stays unchanged
         ...
         ...
         ...
    **/

   node.n = size(node.left) + size(node.right) + 1; // udpate size

   // update new height field
   node.h = Math.max(height2(node.left), height2(node.right)) + 1;

   return node;
}
```

Question 3.3.4
Prove that the height of a 2-3 tree with n keys is between $\lfloor \log_3 n \rfloor \approx 0.63 \lg n$ (for a tree that is all 3-nodes) and $\lfloor \lg n \rfloor$ (for a tree that is all 2-nodes).

Solution
For a fixed height h, a 2-3 tree with only 2-nodes gives us a lower bound on the number of total keys in the 2-3 tree. In particular, a 2-3 tree with height h with only 2-nodes has $2^{h+1} - 1$ keys. Thus, for a general 2-3 tree, we have the following lower bound.

$$\begin{aligned} & 2^{h+1} - 1 \leq n \\ & \Rightarrow 2^h \leq 2^{h+1} - 1 \leq n \\ & \Rightarrow \log_2(2^h) \leq \log_2(n) \\ & h \leq \log_2(n) \end{aligned}$$

Since h must be a non-negative integer, we can conclude that

$$h \leq \lfloor \log_2(n) \rfloor \tag{1}$$

where n is the number of keys in the 2-3 tree. Similarly, if we consider the case in which the 2-3 tree has only 3-nodes, then we can find an upper bound on the number of total nodes in the 2-3 tree of height h. In the case of only 3-nodes, the maximum number of keys is given by $3^{h+1} - 1$, so we have the following inequality

$$\begin{aligned} & n \leq 3^{h+1} - 1 \\ & \Rightarrow n < 3^{h+1} \\ & \Rightarrow \log_3(n) < h + 1 \\ & \Rightarrow \lfloor \log_3(n) \rfloor < h + 1 \\ & \Rightarrow \lfloor \log_3(n) \rfloor \leq h \end{aligned}$$

Combining this with inequality (1) above gives us

$$\lfloor \log_3(n) \rfloor \leq h \leq \lfloor \log_2(n) \rfloor$$

which is exactly the inequality we wanted to show. □

Question 3.3.13
True or False: If you insert keys in increasing order into a red-black BST, the tree height is monotonically increasing.

Solution
The statement is true. This can be seen if we consider the operations used when inserting keys in increasing order into the red-black BST. Since each key inserted into the tree is the largest key seen so far, it will be initially added as the right-most child. From here, there is either a rotation performed or no rotation performed. In the case of no rotation, adding the the key will either leave the height unchanged or increase the height by 1. In the case of a rotation, the tree is adjusted so that it leans left. This results in the height staying unchanged, as it only alters the orientation of the tree. In both cases, the height either remains the same or increases. Thus, for keys inserted in increasing order into a red-black BST, the tree height is monotonically increasing. □