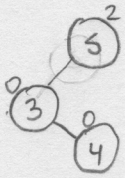
Algorithms HW #4 Problem 1 Jose Nino 10/07/2014.

a) The extra piece of information that I would store in each node would be the number of nodes on the left side of the node. So for example the singleton node with key x would have a value of 0

A tree that looks like the following



would have values 2 for key 5, 0 for 3, 0 for 4.
and so on. lets call this value leftNodes.

b) having the above additional piece of information enables us to calculate the rank for any particular node by navigating from the particular node all the way to the root, so in worst case it would be an

O(depth) operation.

the algorithm works as follows.

1. Set your rank to leftNodes.
2. look at your parent if there is parent, else go to step 6.
3. If parent's key is larger than your key dont add anything to rank
4. If parent's key is smaller than your key add its leftNodes + 1 to the rank.
5. Move to parent, go to step 2.
6. return rank.

this key is the key of the node we just came from.

this algorithm would return the correct rank of node x because as it traverse up the tree it only added leftNodes values for keys that were smaller than the previous node which in turn is smaller than the original key.

Now for running time, It is clear that steps 1, 3, 4, 5, 6 run in O(1) time. step 2, the traversal, moves from the starting node all the way up to the root. Therefore in the worst case we are looking up the rank of

a node in the bottom of the tree in which case steps 3 or 4 (both constant time) would be executed the depth of the tree times.

c) In order to maintain this information we need to update nodes on every insert. The following algorithm solves this problem. Note that we are looking at this problem post insertion, ie we don't take into account the time it takes to traverse from the root to the insertion point.

Update leftNodes algorithm.

1. Insert node x and set its leftNodes to 0
2. look at your parent, if there is a parent, else go to step 6.
3. If the key is larger than your key add 1 to the parent's leftNode value.
4. If the key is smaller than your key dont add anything to the parent's leftNode value
5. Move to parent, go to step 2
6. Done updating

this algorithm successfully updates the leftNode value for every affected Node as it traverses up to the root. It only updates parents with larger keys because this means that a node was added on the left side of their tree.

For the algorithm's running time, it is clear that all steps run at O(1) amount of time. However we will need to repeat steps 3 or 4 a total of the nodes visited. Now, in the worst case, where a node is inserted at the bottom level of the tree, we will have to traverse from parent to parent all the way up to the root, which means we will perform steps 3 or 4 a total of the depth of the tree times so this means the algorithm runs in O(depth) time.

Problem 2 P

a) lets describe all numbers in the universe as 3-bit numbers. And lets have $h_1(x), h_2(x), h_3(x), h_4(x)$ be represented by the following table.

	000	001	010	011	100	101	110	111
h_1 100	0	0	0	0	1	0	1	1
h_2 101	0	1	0	1	1	1	1	0
h_3 110	0	0	1	1	1	0	0	0
h_4 111	0	1	1	0	1	1	0	1

the mappings are computed by the cross product of the function's vector and the 3 bit representation of the integers in U.

Now need to prove that our family $H = \{h_1, h_2, h_3, h_4\}$ is a universal hash function.

Notice that for any given $x, y \in U$ at least two of the hash functions map them to different values. So for instance for 1 (001 in 3-bits) and 7 (111 in 3-bit) functions h_3, h_4 map them both to 0, 1 respectively, causing collisions. However h_1 and h_2 map them to different values. $1 \to 0 / 7 \to 1$ and $1 \to 1 / 7 \to 0$ respectively.

this means that for all $x, y \in U$ $x \neq y$ the probability that they will collide if we picked an h from H at random is 1/2. By definition an Universal Hash family is that in which $\forall x, y$ $x \neq y$. for H with U.

$$\Pr_{h \sim H}[h(x) = h(y)] \leq \frac{1}{M}$$

In our case $M = 2$. So the above hash family is Universal by definition

b) we know from class that the expectation for the collision of any given x and y is $1/M$ where M is the size of the hash table.

So the expected number of pairs that collide is the sum of the expectation that any given pair will collide over all possible pairs. That is:

$$E(\text{Total Collisions}) = \sum_{(x,y)} E(C_{xy})$$ C_{xy} is the expectation that x and y will collide.

And we know that all possible pairs x, y is the size of the set S that we want to hash chose 2. $|S| = m$

So $E(\text{total Collisions}) = \binom{m}{2} \cdot \overbrace{E(C_{xy})}$ which is $\frac{1}{M}$ $M = |\text{Hash table}| = m$

$$= \binom{m}{2} \cdot \frac{1}{m}$$

$$= \frac{m(m-1)}{2} \cdot \frac{1}{m}$$

$$= \frac{m-1}{2}$$

So the expected number of pairs x, y that will collide (total number of collisions) is at most $\frac{m-1}{2}$.

c) In part b we proved that the total expected number of collisions is bounded by $\frac{m-1}{2}$. By this we can state that there are no more than $\frac{m-1}{2}$ expected collisions in any single bin.

Moreover, we can state that if there are x elements in a bin there are $\binom{x}{2}$ distinct pairs that collide in that bin.

From markov's inequality we can say.

$$P(X > hE[X]) < \frac{1}{h}.$$

So lets say that our random variable is the number of collisions in a single bin, which we know it has an expectation bounded by $\frac{m-1}{2}$. so

$$Pr\left(X > 4\left(\frac{m-1}{2}\right)\right) < \frac{1}{4}. \qquad \text{and } \frac{4(m-1)}{2} = 2m-2.$$

Now lets say that a bin contains an arbitrary number of elements $1+2\sqrt{m}$. That means that that bin contains $\binom{1+2\sqrt{m}}{2}$ collisions which simplifies to $\sqrt{m}+2m$ collisions. Which is greater than $2m-2$. So this means that for $1+2\sqrt{m}$ elements the number of collisions is greater than four times the number of expected collisions, which by markov's inequality happens with probability $\frac{1}{4}$. this means that by the number of collisions we can see that the probability of having more than or $1+2\sqrt{m}$ elements in a bin is less than 1/4. So it must be that with probability of at least 3/4 the single bin has more than $\binom{1+2\sqrt{m}}{2}$ collisions and thus no more than $1+2\sqrt{m}$ elements in it.

Problem 3a

Claim: a height-balanced tree of height k has at least F_k nodes where F_k is the k^{th} fibonacci Number.

Proof// Base case for height 0 there is 1 node, the singleton node, and $F_0 = 1$

assume the claim is true for all height balance trees up to height k.

let T be a height balanced tree of height $k+1$. Therefore based on the definition given in the problem since the height of T is $k+1$, then at least one of T's children has height k and neither has height greater than k. Thus either both T.left and T.right have height k or one has height k and the other has height $k-1$. Thus by our induction hypothesis the number of nodes is at least the number of nodes in a tree with height k (F_k) plus the number of nodes in a tree with height $k+1$ (F_{k-1}) plus 1 (the new root). That is the number of nodes at a tree of height $k+1$ is at least $F_k + F_{k-1} + 1$ which is at least F_{k+1} ∎

Now we can use the above claim to prove that in any height balance tree with n nodes, the height at the root is $O(\log n)$.

We know that for the n^{th} fibonacci number it is true that

$$F_h \geq \left(\frac{1+\sqrt{5}}{2}\right)^h$$ where h is the height of the tree.

Based on the proof above number of nodes n for a tree of height h is at least F_h so $n \geq F_h$ so:

$$n \geq \left(\frac{1+\sqrt{5}}{2}\right)^h$$ now let $\left(\frac{1+\sqrt{5}}{2}\right) = Q$ a constant Q

$$n \geq Q^h$$

$$\log_Q(n) \geq \log_Q(Q^h)$$

$$\log_Q(n) \geq h$$

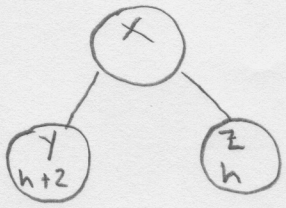
$$O(\log n) = \frac{\log n}{\log Q} \geq h$$

So for a height balanced tree with n nodes the height of the tree, h, is $O(\log n)$.

b) The problem lets us assume the following. Bf denotes Balance factor

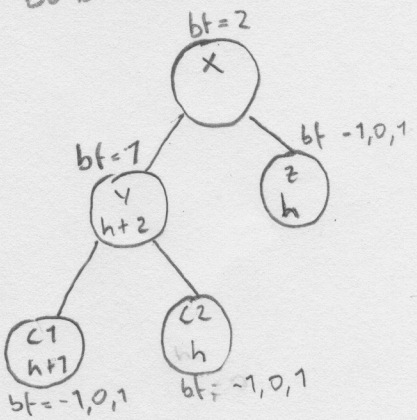
Suppose balance factor of x is 2 so the height of x.left is more than the height of x.right.

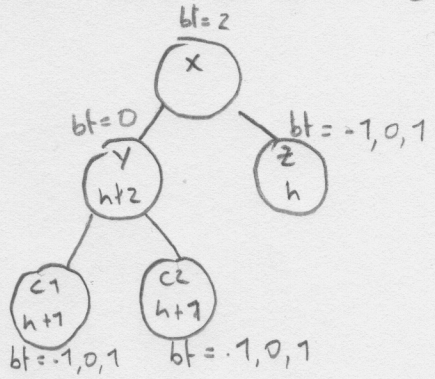
From this we have



Moreover y and z are balanced AND The balance factor of Y is either 1 or 0. So we have

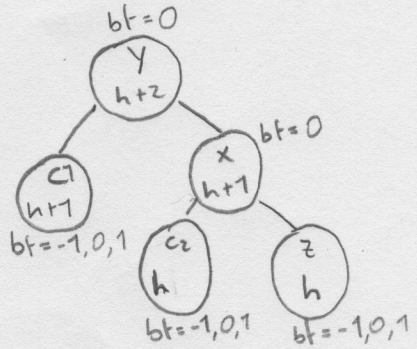
balance factor of Y = 1 or balance factor of Y is 0



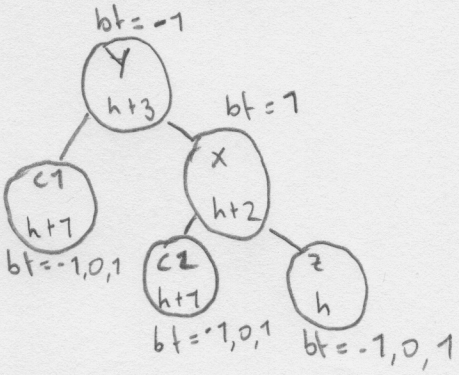


In any of the two cases we can perfor a rotation to the right by making Y the new root, x its right child, C1 its left child, z stays x's right child and C2 becomes x left child. Note that the only height that changes is X's.

Original balance factor of Y is 1 after rotation

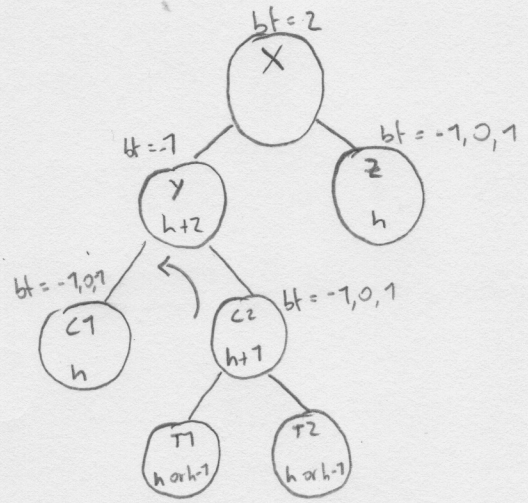


Original balance factor of Y is 0 after rotation

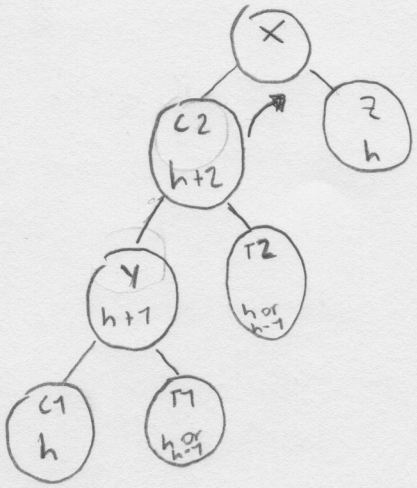


in any of the two cases the right rotation eliminated the ubalance at node x and maintained every other balance factor at -1, 0, or 1. Moreover, the rotation kept the ordering property of the binary search tree as the key in X and all its children's keys are greater than Y's key and the key in C2 and all its chindren's keys are less than X's key.

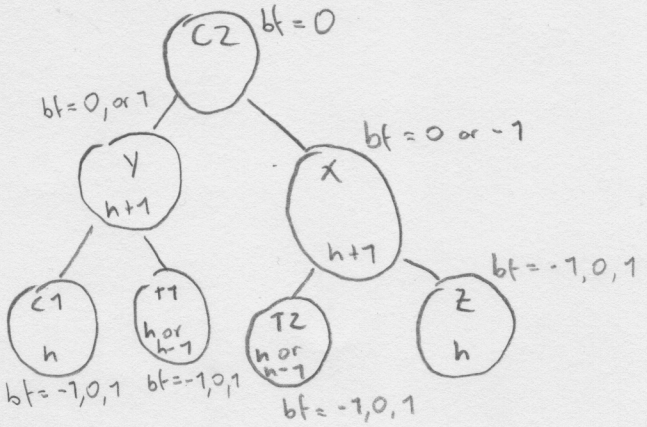
c) This problem represents the last case at which the balance factor of Y is -1.



if we do a left rotation on Y and C2 we can get to one of the cases in part b.



Now we can do a right rotation like in part b because the node C2 is leaning left (note that C2 itself might have a balance factor of 2 momentarily in this step, Note that the rotation kept BST rules as C2's key is greater than Y's key all all of Y children's key. And the key at T1 and all of T1's children's keys are greater than Y's key.



So after rotating left every node in the tree rooted at C2 will have a balance factor of 0, -1, or 1. Here the ordering property is maintained as explained in part b.

d) To insert a new node we traverse from the root of the tree to the place where it needs to be inserted. From part a we know that the height of the tree is at most O(log n) so we might need to in the worst case descend O(log n) levels to insert a node.
After insertion we have to go back up to the root updating heights. Before insertion the tree was height balanced, so the balance factor for any given node was 0, 1, or -1. Moreover, only nodes on the path from the root to the insertion point change in height by 1. So when going up to the root it is possible that for any and all given nodes in the path to have gained a balance factor of 2 or -2. In this case we will need to rotate accordingly, but these rotations are all constant time.

Altogether after inserting a new node the number of nodes we need to update height for is O(log n) and thus the number of rotations needed is O(2 log n) so going back up the tree costs at most O(log n). Therefore if going down the tree to insert cost O(log n) and updating heights and rebalancing cost O(log n), then the entire insert operation costs O(log n).

collab with
Emily Wagner
Corby Rosset