KØBENHAVNS UNIVERSITET

Assignment 4

Alex (zhc522), Adit (hjg708), Usama (mhw630), Nikolaj (bxz911)

AD 2023

10. april 2023

Task 1

Consider the node types used in CLRS chapters 12 and 13. We will add an extra field $x.size$, which denotes the size of the subtree rooted in x. The size of a subtree is the number of nodes in that subtree. In order to maintain this field a few things need to be changed.

Using the size field described above, give pseudocode for a procedure **Get-kth-Key**(x, k), which returns the kth smallest key in the binary search tree (BST) rooted in x. If k is not positive or bigger than the number of elements in the tree, your function should return **Nil**.

Algorithm 1 GET-KTH-KEY(x, k)

1: **if** $k \leq 0$ or $k \geq x.size + 1$ **then**
2:     **return** NIL
3: **if** $k > x.left.size + 1$ **then**
4:     **return** GET-KTH-KEY$(x.left, k)$
5: **if** $k < x.left.size$ **then**
6:     **return** GET-KTH-KEY$(x.right, k - x.left.size - 1)$
7: **if** $k = x.left.size + 1$ **then**
8:     **return** x

Task 2

Prove the correctness of your algorithm in task 1.
Hint: You may do this using induction over the tree size.

For us to proof the correctness of the algorithm for Get-kth-Key(x,k), we are using induction proof.

Base case:
For our base case we use the condition where we have a tree with only one node, which is the root. In this base case $x.size$ would be 1, and k can only be 1, or else it would return NIL. The algorithm would correctly return the kth key in the tree, as it is the only node of the tree.

Induction hypothesis: Assume that the algorithm works for all Binary Search Trees of size less to $x.zise$, which is the size of the tree rooted at x.

Our base case $k = 1$ and $x.size = 1$ would enter the if-statement in line 7. Here $x.size$ would be 1, $x.right.size$ would 0, which is the same of saying $1 - 0 == 1$. The if statement is true, so we would get x in return.

Inductive step:
Consider a Binary Search Tree with the size of $x.size + 1$ size rooted in x. In this step we let k be an integer such that $1 \leq k \leq x.size + 1$.
Case 1:
If $k \leq 0$ or $k \geq x.size + 1$ in line 1, the algorithm returns Nil.

Case 2:
If $k < x.left.size$ in line 5, then the kth smallest key is in the left sub tree of x. The algorithm will run recursively on the left sub tree, each recursion will use a smaller sub tree. In the end the algorithm will correctly return the kth smallest key. This fulfills our induction hypothesis of the algorithm working for all BST of size less to $x.size$.

Case 3:
If $k = x.left.size + 1$ in line 7, then the kth smallest key is the root key x. This is because all key in the left sub tree of x are smaller than the root key, and all keys in the right sub tree of x are greater that the root key.

Case 4:
If $k > x.left.size + 1$ in line 3, then the kth smallest key is in the right sub tree of x. The algorithm will run recursively on the right sub tree, each recursion will use a smaller sub tree. In the end the algorithm will return the $(k - x.left.size - 1)$th smallest key in the right sub tree. This fulfills our induction hypothesis of the algorithm working for all BST of size less to $x.size$.

We have therefore proven through induction that the algorithm will work for all Binary Search Trees of the size $x.size + 1$, as long as the following statement holds: $1 \leq k \leq x.size + 1$.

Task 3

Modify the pseudocode of Left-Rotate (CLRS, Fig. 13.3) such that it correctly updates the size field. You do not have to include the entire pseudocode – just state the necessary changes in a readable way.

Algorithm 2 GET-KTH-KEY(x, k)

LEFT-ROTATE(T, x)
//We update y.size to the old x.size, since we swap which is the
//parent and which is the child node, so the size stays the same
$y.size \leftarrow x.size$
$x.size \leftarrow x.left.size + x.right.size + 2$

We update y.size to the old x.size, since when we swap, we swap x and y's position, so the new y.size is the old x.size. We then update x.size to its 2 childrens sizes + 2. +2 as when you call x.size it does not include x itself.

Task 4

Modify the pseudocode of RB-Insert and RB-Insert-Fixup (CLRS, p. 315-316) such that the size field is updated. You can assume that the rotate functions update the size field correctly (this is addressed in Task 3 for Left-Rotate). You do not have to include the entire pseudocode – just state the necessary changes in a readable way.

Algorithm 3 increaseNodeSize(T,z)

//Between lines 16-17 in RB-Insert(T,z)
$x \leftarrow z$
while x != T.root **do**
    $x \leftarrow x.parent$ //Go up 1 step
    $x.size \leftarrow x.size + 1$ //Update the old size

Before RB-Insert-Fixup is called, we update the size of each parentnode from the inserted spot to the root. We do not change RB-Insert-Fixup as task 4 already says the rotate updates the size correctly.

Task 5

What is the worst-case running time of RB-Insert now? What are the worst-case running times of the three operations discussed in Section 2?

We've understood this question as what is the worst-case running times of the three operations when updated to changing x.size, as that is what the rest of the assignment has been about. n is the amount of nodes in the tree.

The runtime of RB-Insert was $O(lg\, n)$, it is still $O(lg\, n)$ after the change as the while loop takes $O(lg\, n)$, this is because it needs to traverse from the bottom of the tree to the root.

Deleting a node from a binary search tree normally takes $O(lg\, n)$, but we have to update the size after the deletion, this takes $O(lg\, n)$ too, so it is still the same.

Get-kth-Key traverses the tree by making a decision of going left or right and then calling itself recursively. This means that it makes worst-case $O(lg\, n)$ decisions, where n is the the number of nodes in the tree.

Task 6

Consider a sequence of n insertions/deletions and m queries (that is calls to **Get-kth-Key**) intertwined arbitrarily. What is the worst-case running time of handling such a sequence with your algorithm?

Handling such sequence would take the slowest running time operation out of the sequence. As all three operations take $O(lg\, n)$ time, the worst-case running time for handling a sequence of n insertions/deletions and m queries with the algorithm for **Get-kth-Key** is $O(n \cdot h + m \cdot h)$, where h is $2\, log\,(n)$, when you talk about a RB-tree.