Assignment 4 — AD

Schmidt, Victor Alexander, `rqc908`
Ibsen, Helga Rykov, `mcv462`
Larsen, Christian Vadstrup, `ljq919`

Monday 08:00, March 13th

Task 1

Algorithm 1 Get-kth-Key(x,k)

```
if k < 0 or k > x.num_elements then
    return Nil
else
    if k = x.left.num_elements then
        return x
    else if k > x.left.num_elements then
        Get-kth-Key(x.right, k- x.left.num_elements)
    else
        Get-kth-Key(x.left, k)
    end if
end if
```

An alternative implementation of `Get-kth-Key()` contains a helper-method `Search-kth-Key` that recursively searches through the tree and returns the kth smallest key in the binary search tree (BST) rooted in x.

Algorithm 2 Get-kth-Key(x,k)

```
counter = 0
if x == Nil then
    return x
end if
if k > x.size OR k < 1 then
    return Nil
else
    return Search-kth-Key(x,k)
end if
```

Algorithm 3 Search-kth-Key(x,k)

```
if x ≠ Nil then
    left = Search-kth-Key(x.left,k)
    counter = counter + 1
    if counter = k then
        return x.key
    end if
    right = Search-kth-Key(x.right,k)
    if left ≠ Nil then
        return left
    end if
    if right ≠ Nil then
        return right
    end if
    return Nil
else
    return Nil
end if
```

Task 2

Let $P(n)$ be the proposition that **Algorithm 2** returns the kth key in the in-order traversal of a binary search tree rooted at x, where the size of the tree is n and n is greater than or equal to k.

Base case: When $n = 0$ or $k = 0$, the algorithm returns Nil, which is correct.

Induction hypothesis: We assume that $P(n)$ is true for all values of k less than or equal to n, where n is greater than or equal to 1.

Inductive step: Let T be a binary search tree rooted at x with $n + 1$ nodes, and assume that k is a valid index in the inorder traversal of T, where k is between 1 and $n + 1$ inclusive.

We need to consider two cases:

Case 1: If $k <= x.left.size$, then the kth key in T is in the left subtree of x. By the induction hypothesis, $P(x.left.size)$ is true, so `Get-kth-Key(x.left, k)` will return the kth key in the inorder traversal of the left subtree. Since counter is incremented each time a node is visited, the value of counter after visiting the left subtree will be $x.left.size$. If counter equals k, then $x.key$ is the kth key in T, and the algorithm correctly returns $x.key$. Otherwise, we need to continue the search in the right subtree.

Case 2: If $k > x.left.size$, then the kth key in T is in the right subtree of x. By the induction hypothesis, $P(x.right.size)$ is true, so `Get-kth-Key(x.right, k - x.left.size - 1)` will return the $(k - x.left.size - 1)$th key in the inorder traversal of the right subtree. Since counter is incremented each time a node is visited, the value of counter after visiting the right subtree will be $x.left.size+1$ (for visiting x) + $x.right.size$. If counter equals k, then $x.key$ is the kth key in T, and the algorithm correctly returns $x.key$. Otherwise, the algorithm will return Nil if the kth key is not found in T.

Therefore, the algorithm is correct and $P(n + 1)$ is true, assuming that $P(n)$ is true for all values of k less than or equal to n.

Task 3

We have modified the pseudocode of Left-Rotate (CLRS, Fig. 13.3) such that it correctly updates the size field by adding two extra lines of code at the end of Left-Rotate.

Algorithm 4 LEFT-ROTATE(T, x)

```
...
x.p = y
x.size = x.left.size + x.right.size + 1 // updates x.size after left-rotate
y.size = y.left.size + y.right.size + 1 // updates y.size after left-rotate
```

Task 4

We do not need to change the pseudocode of RB-Insert-Fixup with respect to the size field because the size is being updated by Left-Rotate and Right-Rotate.

As far as RB-Insert is concerned, in order to update the size field we have added 2 lines of code: 1) after line 4, where x.size is being incremented by 1, and 2) immediately before the call of RB-Insert-Fixup in line 17.

Algorithm 5 RB-Insert(T, z)

```
1: ...
2: while x ≠ T.nil do
3:     y = x
4:     x.size = x.size + 1     // updates x.size
5: end while
6: ...
7: z.color = RED
8: z.size = 1                  // updates z.size
9: RB-Insert-Fixup(T,z)
```

Task 5

The worse-case runtime of `RB-Insert` can be calculated by evaluating the added lines:

Line 4 in **Algorithm 5**: a constant operation within a while-loop. Constant operations were already conducted in `RB-Insert`, so the worst-case runtime will remain the same.

Line 8 in **Algorithm 5**: is a constant operation run once in a `RB-Insert` call. This does not change the worst case run time (other constant operations have been conducted).

This means that the worst case runtime of `RB-Insert` remains the same, which in the book is $T(n) = O(\lg n)$ for an Red-Black-Tree insertion operation.

Now, to answer the worst case running time of the other two operations mentioned before:

Deletion previously took $O(\lg n)$ time, and should now recursively update the size of each parent node to be one less than it was before. This means worst-case that for each layer in the red-black tree, one size-change operation should be done. As we know red-black trees are halved at each step up a layer (as all paths must contain the same amount of black nodes), we can upper bound the running time to $O(\lg n)$ operations.

Using the same argumentation, we can say that searching for the kth smallest element has a worst-case runtime of $O(\lg n)$, as it runs at maximum once per layer in the tree, calling at max one child node recusively.

In summary:

1. Inserting a key: still takes $O(\lg n)$ time.
2. Deleting a key: still takes $O(\lg n)$ time.
3. Querying the kth smallest key in the data structure: takes $O(\lg n)$ time.

Task 6

In case with a BST where the nodes are inserted in sorted order (i.e., the tree is a linked list), if we want to search for the kth key, we need to traverse the entire tree to reach the kth node, which takes $\mathcal{O}(n)$ time. Similarly, if we want to insert or delete a node at the end of the linked list, we need to traverse the entire list to reach the last node, which also takes $\mathcal{O}(n)$ time. Therefore, the worst-case running time of handling a sequence of n insertions/deletions and m queries is $\mathcal{O}(n^2 + m \cdot n)$. If number of queries equals the number of nodes in the tree $m = n$, we have the worst complexity $\mathcal{O}(n^2 + n^2) = \mathcal{O}(n^2)$.

However, in case BST is balanced, the worst-case running time of handling a sequence of n insertions/deletions and m queries with the given algorithm is $\mathcal{O}(n \cdot logn + m \cdot logn)$, where n is the size of the binary search tree.

During each insertion or deletion operation, we traverse the tree to find the appropriate location for the new node or the node to be deleted, which takes $\mathcal{O}(logn)$ time in the worst case. Therefore, n insertions/deletions take $\mathcal{O}(n \cdot logn)$ time in the worst case.

For each query operation, we traverse the tree in search of the kth key using the Search-kth-Key subroutine, which takes $\mathcal{O}(logn)$ time in the worst case. Therefore, m queries take $\mathcal{O}(m \cdot logn)$ time in the worst case.

Thus, the worst-case running time of handling a sequence of n insertions/deletions and m queries with **Algorithm 2** is $\mathcal{O}(n \cdot logn + m \cdot logn)$, which dominates the time complexity of the algorithm.