Exercise 1: Binary Search Trees I

Consider the following binary search tree.

1. Give all sequences of \texttt{insert(key)} operations that generate the tree.

2. Draw the tree after the following sequence of operations: \texttt{insert(6)}, \texttt{insert(5)}, \texttt{remove(3)}.

Sample Solution

1. (i) \texttt{insert(8)}, \texttt{insert(3)}, \texttt{insert(12)}, \texttt{insert(10)}
   (ii) \texttt{insert(8)}, \texttt{insert(12)}, \texttt{insert(3)}, \texttt{insert(10)}
   (iii) \texttt{insert(8)}, \texttt{insert(12)}, \texttt{insert(10)}, \texttt{insert(3)}

2. After \texttt{insert(6)} and \texttt{insert(5)}:

After \texttt{remove(3)}:
Exercise 2: Binary Search Trees II

(a) Describe a function that takes a binary search tree $B$ and a key $x$ as input and generates the following output:

- If there is an element $v$ in $B$ with $v.key = x$, return $v$.
- Otherwise, return the pair $(u, w)$ where $u$ is the tree element with the next smaller key and $w$ is the element with the next larger key. It should be $u = None$ if $x$ is smaller than any key in the tree and $w = None$ if $x$ is larger than any key in the tree.

For your description you can use pseudo code or a sufficiently detailed description in English. Analyze the runtime of your function.

(b) Describe a function which returns the depth of a binary search tree and analyze the runtime.

(c) Describe a function that for a given binary search tree with $n$ nodes and a given $k \leq n$ returns a list with the $k$ smallest keys from the tree. Analyze the runtime.

Sample Solution

(a) Algorithm 1 return-closest($x$)

```
v ← find(x)
if v ≠ None then
    return v
else
    insert(x)
    (p, s) ← (pred(x), succ(x))
    delete(x)
    return (p, s)
```

All subprocedures that we call (find, insert, pred, succ) are known from the lecture and take $O(d)$ with $d$ being the depth of the tree. So the overall runtime is $O(d)$.

(b) We can do a recursive traversal of the tree where we keep track of the current recursion depth. Then a call of depth($r$) on the root $r$ of the BST returns its depth.

Algorithm 2 depth($v$)

```
if v = None then
    return -1  ▷ depth of a childless node must be 0, hence we define the depth of None as -1
else return max(depth(v.left)+1, depth(v.right)+1)
```

The runtime corresponds to the runtime of the traversal of the whole tree which is $O(n)$ as we have just one recursive call for each node and each recursive call costs $O(1)$ (c.f., pre-, in-, post-order traversal algorithms given in the lecture).

As an alternative solution, we can run a BFS which takes $O(n)$. If $v$ is the node visited last by the BFS, do
**Algorithm 3** `traverse-up(v)`

```python
    d ← 0
    while v.parent ≠ None do
        d ← d + 1
        v ← v.parent
    return d
```

This takes $O(d)$ where $d$ is the depth of the tree. Since $d \leq n$ the overall runtime is $O(n+d) = O(n)$.

(c) Initialize an empty list $K$. We roughly do the following. Make an in-order traversal of the tree and each time visiting a node, add it to $K$. Stop if $|K| \geq k$. The following pseudocode formalizes this.

**Algorithm 4** `inorder_variant(node)`  

 Diablo! Assume list $K$ is given globally, initially empty.

```python
    if node ≠ None then
        inorder_variant(node.left)
        if |K| ≥ k then
            return
        K.append(node.key)
        inorder_variant(node.right)
```

The runtime is $O(d+k)$ where $d$ is the depth of the tree. We prove this in the following.

Let $K$ be the set of $k$ nodes representing the $k$ smallest keys in the BST. Obviously, the in-order traversal must visit all nodes in $K$ once. In accordance with the lecture a call of `inorder_variant(root)` adds all keys in ascending order to $K$.

Let $A$ be the set of nodes in the BST which are not in $K$ but in which a recursive call will be made. Since the recursion is aborted (with the `return` statement) after reporting $k$ nodes, the set $A$ contains exactly the nodes which are ancestors of a node in $K$, but are not in $K$ themselves. Since the runtime of a single recursive call (neglecting subcalls) is (1) the total runtime is $O(|A| + |K|)$.

By definition we have $|K| = k$, so it remains to determine the size of $A$. We claim that all nodes in $A$ are on a path from the root to a leaf, that is, $|A| \leq d$. This is the case if there do not exist two nodes in $A$ so that neither is an ancestor of the other.

For a contradiction, suppose that two such nodes $u, v$ exist so that neither $u$ is ancestor of $v$ nor vice versa. Assume (without loss of generality) that `key(u) ≤ key(v)`. That means $u$ is in the left and $v$ is in the right subtree of some common ancestor $a$ of $u$ and $v$.

By definition $v$ has a node $w ∈ K$ in its subtree. Since $v$ is in the right subtree and $u$ is in the left subtree of $a$, we have `key(u) ≥ key(v)` and $w$ has a higher in-order-position. But then we would have $u ∈ K$ as well, a contradiction to $u ∈ A$. 