More Examples and Applications on AVL Tree

CSCI2100 Tutorial 10

Jianwen Zhao

Department of Computer Science and Engineering
The Chinese University of Hong Kong

Adapted from the slides of Junhao Gan and Tony Gong
Recall in lectures we studied the **AVL tree**, which is one type of self-balancing binary search tree. The aim was to store a set of integers $S$ supporting the following operations:

- **A predecessor query**: given an integer $q$, find its predecessor in $S$;
- **Insertion**: add a new integer to $S$; and
- **Deletion**: remove an integer from $S$.

We want all of these operations to run in $O(\log n)$ (where $n$ is the number of integers in $S$) in the worst case. If we were to attempt to accomplish this using a BST, we must ensure it is balanced after every operation, and the AVL tree presents one method of doing so.
Rebalancing

We know that a tree is balanced as long as the height of its subtrees differ by at most 1, and that insertion and deletion can only cause a 2-level imbalance (where the heights differ by 2).

In lectures we explored the Left-Left and Left-Right cases in detail, so here we will look at Right-Right-Right and Right-Left-Left:
Right-Right

Similar to Left-Left, fix by a rotation:

Note that $x = h$ or $h + 1$, and the ordering from left to right of $A, a, B, b, C$ is preserved after rotation.
Right-Left

Similar to Left-Right, fix by a **double rotation**:

Note that $x$ and $y$ must be $h$ or $h - 1$. Furthermore at least one of them must be $h$. 
Insertion

Let's see these in action with some concrete examples involving insertion and deletion. Suppose we start with an empty tree and add 10, 20 and 30. Inserting 30 yields:

We first traverse from root-to-leaf and add a node with key 30. The height of the subtrees along this path are now invalidated, so we traverse back up to the root and recalculate at each node. When we get to node 10, we find that we have an imbalance, in this case of type Right-Right.
Insertion

We fix via a rotation according to the Right-Right case:

One should check that at the end the tree is balanced, and satisfies the binary search tree property!
Insertion

Suppose we then add 25, 40 and 50. Upon inserting 50, we will find the need for another Right-Right rebalancing:
Insertion

To get a Right-Left case, let’s add 35, 33, 37, 60 and 38 (in this exact order). Upon inserting 38 we will find:

![Diagram showing the insertion process]
Let’s look at an example of a deletion as well: suppose that some sequence of operations later we arrive at the following and want to delete the key 20:

After identifying and deleting the appropriate node (more on this later), we again need to traverse back up to the root and rebalance. Here we run into a Right-Left case.
... and we are actually still not done because there is another Right-Right rebalance we need to do:

Remember that at most one rebalance is needed on insert; but deletion may require more than one!
Problem

Let $S$ be a dynamic set of integers, and $n = |S|$. Describe a data structure to support the following operations on $S$ with the required performance guarantees:

- **Insert** a new element to $S$ in $O(\log n)$ time.
- **Delete** an element from $S$ in $O(\log n)$ time.
- **Report** the $k$ smallest elements of $S$ in $O(k)$ time, for any $k$ satisfying $1 \leq k \leq n$.

Your data structure must consume $O(n)$ space at all times.
Solution

We maintain a binary search tree $T$ over $S$, which consumes $O(n)$ space at all times.

- For insertion, insert a new element into $T$, rebalance $T$ and identify the smallest element $e$ in $T$ ($O(\log n)$ time).

- For deletion, remove an element from $T$, rebalance $T$ and identify the smallest element $e$ in $T$ ($O(\log n)$ time).

- For reporting the $k$ smallest elements, start from $e$ and repeat the following until we find the $k$ elements:
  - Starting from current node, find the next node which stores the successor $s$ of the current node, and report $s$. 
Example

Suppose that $k = 5$ and after a sequence of insertions and deletions, we have the following BST:
Example

Then we can report the 5 smallest elements as follows:

```
30
15
32
40
20
10
35
73
60
36
3
```

CSCI2100, The Chinese University of Hong Kong

More Examples and Applications on AVL Tree
**Lemma:** During the process of traversing the $k$ smallest elements, any node can be visited at most twice.

**Proof:** In our algorithm, we traverse the $k$ elements in a bottom-up manner. Therefore, there are only two cases for visiting a node $u$:

- **Case 1:** $u$ has a right child. After visiting $u$, we have to visit its child node $v$, and then go back to visit $u$ again. Hence, $u$ was visited twice.
- **Case 2:** $u$ does not have a right child. After visiting $u$, we directly go up to visit the next node and never come back to $u$ again. Hence, $u$ is only visited once.
Cost analysis

Since visiting a node once takes $O(1)$ time, and based on the lemma we know that a node can be visited at most twice.

Hence, the time for reporting $k$ smallest elements is bounded by $2 \cdot k \cdot O(1) = O(k)$. 
In tutorial 9, we introduced range count problem and solved it by augmenting a balanced BST by storing the number of nodes in the subtree of $u$ in each node $u$. For example:
When we perform a insertion or deletion, we need to rebalance the BST as well as updating the counter values of some nodes.

Suppose that we insert 20 into the following BST:

The node with key 12 becomes imbalanced, and the counter values of the nodes along the path we traversed when inserting 20 are now invalidated.
We first rebalance the BST via a rotation (Right-Right case), and then recalculate the counter values of the aforementioned nodes.
Recall that when performing insertion or deletion, we essentially descend a root-to-leaf path, only the counter values of the nodes on this path need to be updated.

Therefore, we can update the counter values of these nodes in the bottom-up order, which follows the same idea with the updating of the subtree height values of these nodes.
Problem

Given an array \( A \) with \( n \) integers. Describe an algorithm to produce a new array \( B \), such that \( B[i] \) is the number of smaller elements to the right of \( A[i] \). Where, \( i = 1, 2, 3, \ldots, n \).

Example

\( A: \) 29 12 47 40 71 15 3 41 18 11 92 68

To the right of 41(\( A[8] \)), there are 2 smaller elements, then \( B[8] = 2 \);
To the right of 71(\( A[5] \)), there are 6 smaller elements, then \( B[5] = 6 \);

\[ \ldots \]

Hence, the output array \( B \) should be:

\( B: \) 5 2 6 4 6 2 0 2 1 0 1 0
Before we step into the solution, let’s first recall the definition of Lowest Common Ancestor introduced in tutorial 9.

- **Lowest Common Ancestor**: Let $t$ be the root. The lowest common ancestor of nodes $v_1$ and $v_2$ is the lowest node that is on both of the paths $P(t, v_1)$ and $P(t, v_2)$.

For example, the lowest common ancestor of node with key 3 and node with key 15 is the node with key 12.
Solution

We maintain a balanced BST $T$ with counter, at the beginning, $T$ is empty. For $i = (n - 1)$ downto 1 (since $B[n] = 0$), do the following:

- Find the smallest element in $T$ constructed on the elements from $A[i + 1]$ to $A[n]$, denoted by $s$.
- Find the predecessor of $A[i]$ in $T$, denoted by $p$. If $p$ does not exist, set $B[i] = 0$, otherwise, continue.
- Let $u$ be the lowest common ancestor of $s$ and $p$, and $w_1, w_2$ be the left and right child of $u$ respectively.
- Initialize $c = 1$.
- Increase $c$ by the number of nodes on $P(w_1, s)$ and $P(w_2, p)$ whose keys are in $[s, p]$.
- Walk along the path $P(w_1, s)$ and for each node $v$ being visited, increase $c$ by the counter of $v$'s right child.
- Walk along the path $P(w_2, p)$ and for each node $v$ being visited, increase $c$ by the counter of $v$'s left child.

CSCI2100, The Chinese University of Hong Kong
Example

\[ A: \quad 29 \, 12 \, 47 \, 40 \, 71 \, 15 \, 3 \, 41 \, 18 \, 11 \, 92 \, 68 \quad n = 12 \]

Suppose that after a sequence of operations, \( i = 5 \), then our BST \( T \) constructed on the elements from \( A[6] \) to \( A[12] \) should be:

Since \( A[i] = 71 \), identify \( s, p, u, w_1 \) and \( w_2 \), as indicated above.
• Initialize $c = 1$.

• $c+$ = the number of nodes on $P(w_1, s)$ and $P(w_2, p)$ whose keys are in $[s, p]$.

• $c+$ = the counter of right child for each node on $P(w_1, s)$

• $c+$ = the counter of left child for each node on $P(w_2, p)$

Example

Then, we should insert $A[i] = A[5] = 71$ into $T$ and make sure we still have a balanced BST and also update the counter values of nodes, this is for the convenience of processing $(i - 1)$-th element of $A$.

Now, we have finished the computation of $B[i]$. Following the above process, we can finally produce the counting array $B$. 

\[
\begin{align*}
\text{cnt} & = 7 \\
18 & \quad \text{cnt} = 3 \\
11 & \quad \text{cnt} = 3 \\
3 & \quad \text{cnt} = 1 \\
15 & \quad \text{cnt} = 1 \\
41 & \quad \text{cnt} = 1 \\
92 & \quad \text{cnt} = 1 \\
\text{cnt} & = 3 \\
18 & \quad \text{cnt} = 3 \\
11 & \quad \text{cnt} = 1 \\
3 & \quad \text{cnt} = 1 \\
15 & \quad \text{cnt} = 1 \\
41 & \quad \text{cnt} = 1 \\
92 & \quad \text{cnt} = 1 \\
71 & \quad \text{cnt} = 1 \\
\end{align*}
\]
Cost Analysis

- For each element in $A[i]$, we perform a range count and an insertion, which takes at most $O(\log n)$ time.
- We have $n$ elements in total.

Hence, the whole algorithm terminates in $O(n \log n)$ time.