Binary Search Tree (Part 2 – The AVL-tree)

Yufei Tao

ITEE
University of Queensland

We have already learned a static version of the BST. In this lecture, we will make the structure dynamic, namely, allowing it to support updates (i.e., insertions and deletions). The dynamic version we will learn is called the AVL-tree.

Recall:

Binary Search Tree (BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

- T has n nodes.
- Each node u in T stores a distinct integer in S, which is called the key of u.
- For every internal u, it holds that:
  - The key of u is larger than all the keys in the left subtree of u.
  - The key of u is smaller than all the keys in the right subtree of u.

Recall:

Balanced Binary Tree

A binary tree T is balanced if the following holds on every internal node u of T:

- The height of the left subtree of u differs from that of the right subtree of u by at most 1.

If u violates the above requirement, we say that u is imbalanced.

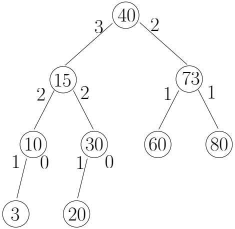
AVL-Tree

An AVL-tree on a set S of n integers is a balanced binary search tree T where the following holds on every internal node u

- u stores the heights of its left and right subtrees.

Example

An AVL-tree on $S = \{3, 10, 15, 20, 30, 40, 60, 73, 80\}$



For example, the numbers 3 and 2 near node 40 indicate that its left subtree has height 3, and its right subtree has height 2.

- By storing the subtree heights, an internal node knows whether it has become imbalanced.
- The left subtree height of an internal node can be obtained in $O(1)$ time from its left child (how?). Similarly for the right.

Next, we will explain how to perform updates. The most crucial step is to remedy a node u when it becomes imbalanced.

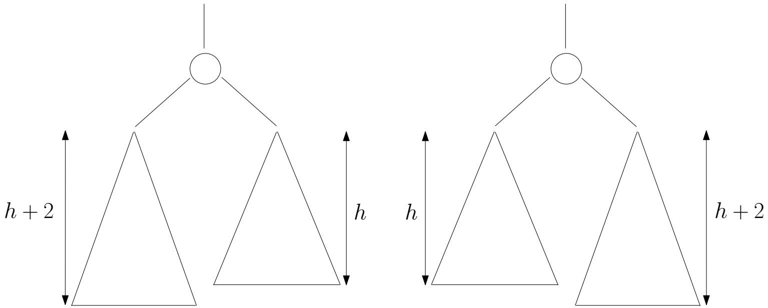
It suffices to consider a scenario called 2-level imbalance. In this situation, two conditions apply:

- There is a difference of 2 in the heights of the left and right subtrees of u.
- All the proper descendants of u are balanced.

Before delving into the insertion and deletion algorithms, we will first explain how to rebalance u in the above situation.

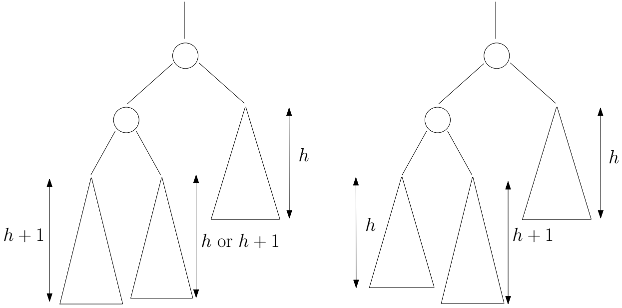
2-Level Imbalance

There are two cases:



Due to symmetry, it suffices to explain only the left case, which can be further divided into a left-left and a left-right case, as shown next.

2-Level Imbalance

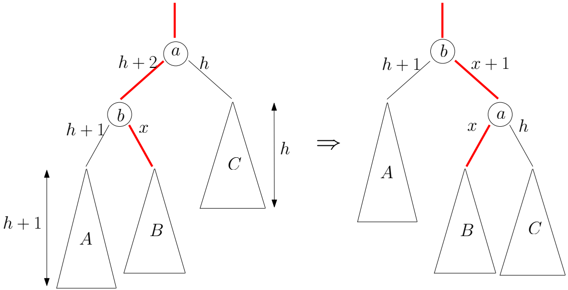


Left-left

Left-Right

Rebalancing Left-Left

By a rotation:

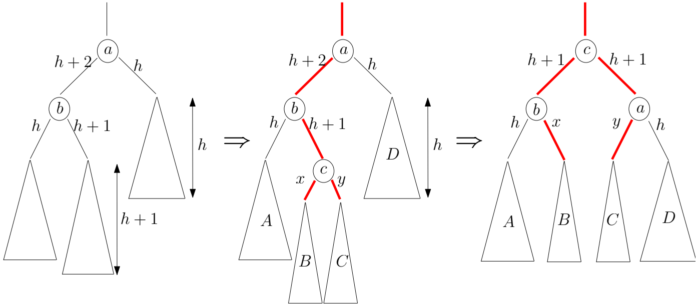


Only 3 pointers to change (the red ones). The cost is $O(1)$.

Recall $x = h$ or $h + 1$.

Rebalancing Left-Right

By a double rotation:



Only 5 pointers to change (see above). Hence, the cost is $O(1)$.

Note that x and y must be h or $h - 1$. Furthermore, at least one of them must be h (why?).

We are now to explain the insertion algorithm

Insertion

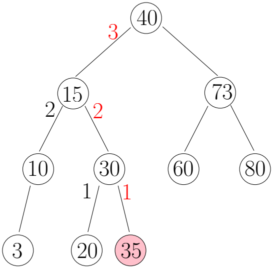
Suppose that we need to insert a new integer e. First create a new leaf z storing the key e. This can be done by descending a root-to-leaf path:

1. Set $u \leftarrow$ the root of T
2. If u is a leaf node:
   - 2.1 If the key of $u > e$, make z the left child of u, and done.
   - 2.2 Otherwise, make z the right child of u, and done.
3. Otherwise (i.e., u is an internal node):
   - 2.1 If the key of $u > e$, set u to the left child.
   - 2.2 Otherwise, set u to the right child.
4. Repeat from Line 2.

Finally, update the subtree height values on the nodes of the root-to-z path in the bottom-up order. The total cost is proportional to the height of T, i.e., $O(\log n)$.

Example

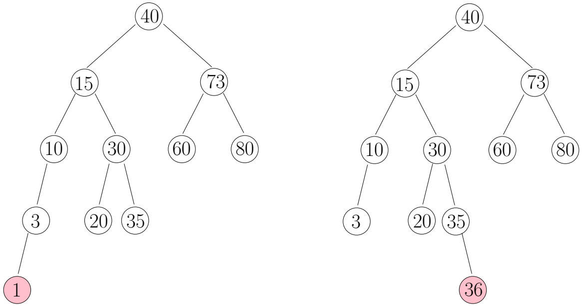
Inserting 35:



The red height values are modified by this insertion.

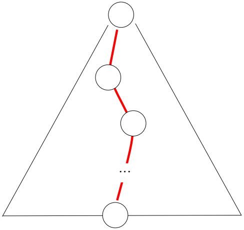
Example

An insertion may cause the tree to become imbalanced!



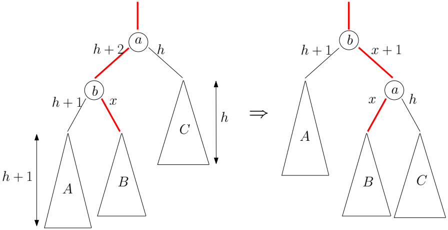
In the left tree, nodes 10 and 40 become imbalanced, whereas in the right, node 40 is now imbalanced.

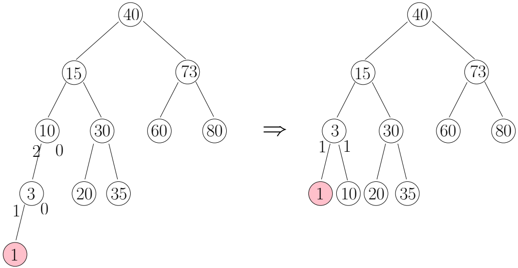
Imbalance in an Insertion



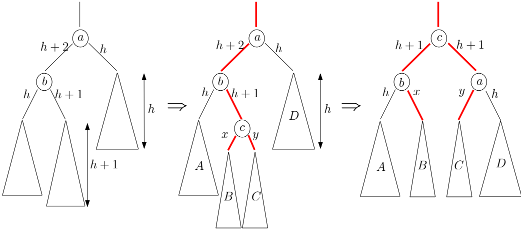
Only the nodes along the path from the insertion path (from the root to the newly added leaf) can become imbalanced. It suffices remedy only the lowest imbalanced node.

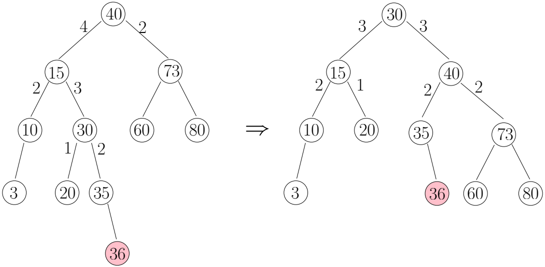
Left-Left Example



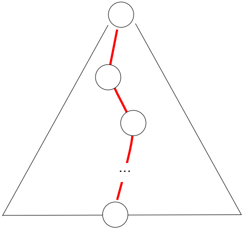


Left-Right Example





Insertion Time



It will be left as an exercise for you to prove:

- Only 2-level imbalance can occur in an insertion.
- Once we have remedied the lowest imbalanced node, all the nodes in the tree will become balanced again.

The total insertion time is therefore $O(\log n)$.

We now proceed to explain the deletion algorithm.

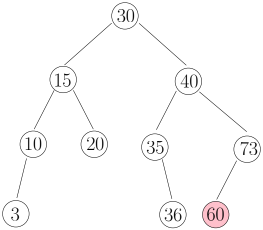
Deletion

Suppose that we want to delete an integer e. First, find the node u whose key equals e in $O(\log n)$ time (through a predecessor query).

Case 1: If u is a leaf node, simply remove it from (the AVL-tree) T.

Case 1 Example

Remove 60:



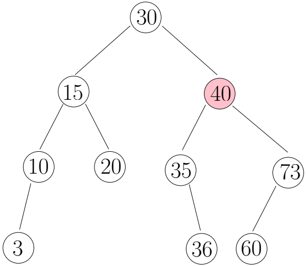
Deletion

Now suppose that node u (containing the integer e to be deleted) is not a leaf node. We proceed as follows:

- **Case 2:** If u has a right subtree:
  - Find the node v storing the successor s of e.
  - Set the key of u to s
  - **Case 2.1:** If v is a leaf node, then remove it from T.
  - **Case 2.2:** Otherwise, it must hold that v has a right child w, which is a leaf (why?), but not left child. Set the key of v to that of w, and remove w from T.
- **Case 3:** If u has no right subtree:
  - It must hold that u has a left child v, which is a leaf. (why?) Set the key of u to that of v, and remove v from T.

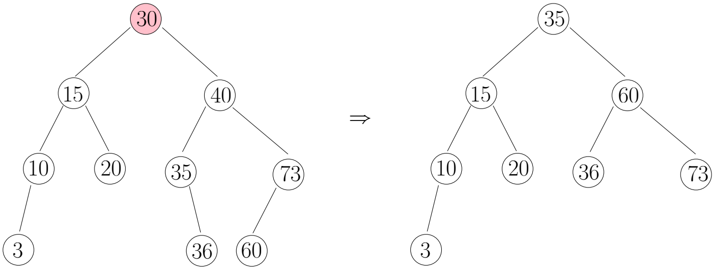
Case 2.1 Example

Delete 40:

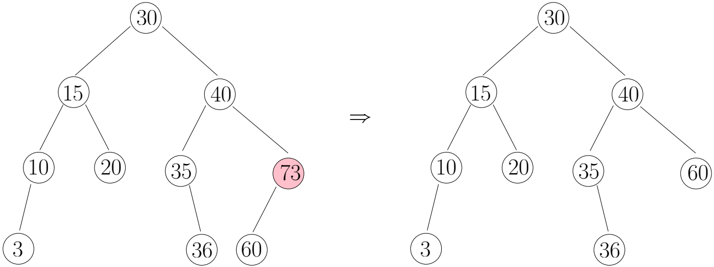


Case 2.2 Example

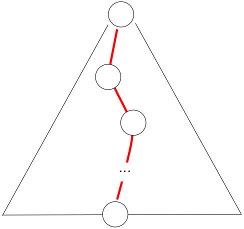
Delete 30:



Case 3 Example



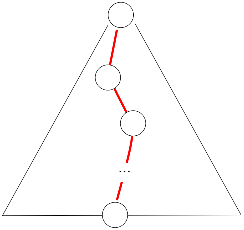
Deletion



In all the above cases, we have essentially descended a root-to-leaf path (call it the deletion path), and removed a leaf node. We can now update the subtree height values for the nodes on this path in the bottom-up order.

The cost so far is $O(\log n)$. Recall that the successor of an integer can be found in $O(\log n)$ time.

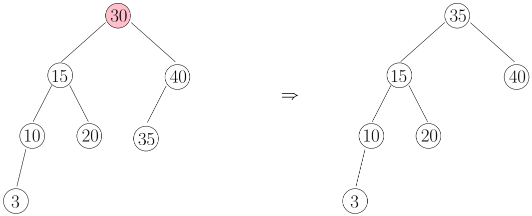
Imbalance in a Deletion



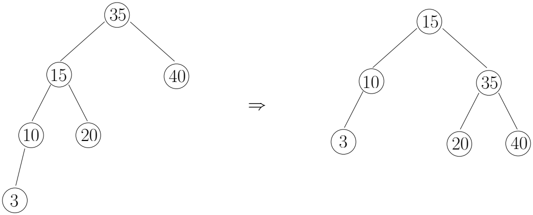
- Only the nodes along the deletion path may become imbalanced.
- We fix each of them in the bottom-up order in exactly the same way as described for insertions, namely, using either a rotation or double rotation.
    - Unlike an insertion, we may need to remedy more than one imbalanced node.

Left-Left Example 1

Delete 30:

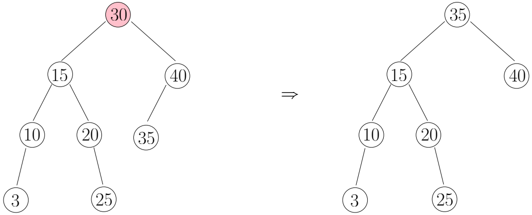


Node 35 becomes imbalanced – a left-left case handle by a rotation:

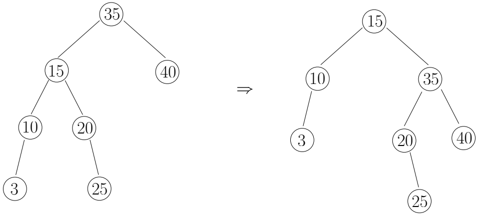


Left-Left Example 2

Delete 30:

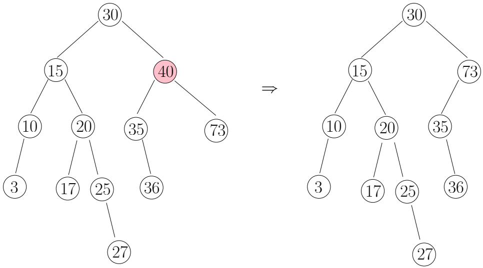


Node 35 becomes imbalanced – also a left-left case handle by a rotation:



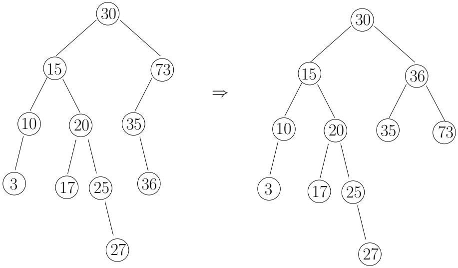
Left-Right Example

Delete 40:



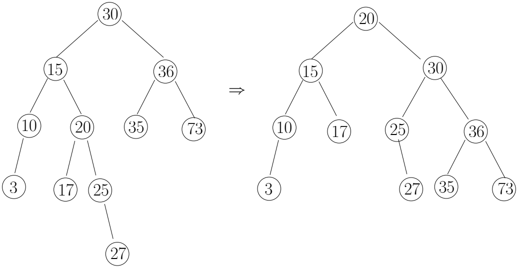
Node 73 becomes imbalanced – a left-right case handled by a double rotation. See the next slide.

Left-Right Example



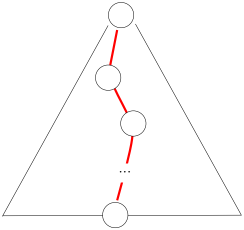
Node 30 is still imbalanced – a left-right case handled by a double rotation. See next.

Left-Right Example



Final tree after the deletion. Note that this deletion required fixing 2 imbalanced nodes.

Deletion Time



It will be left as an exercise for you to prove that

- Only 2-level imbalance can occur in an insertion.

Since we spend $O(1)$ time fixing each imbalanced nodes, the total deletion time is $O(\log n)$.

We now conclude our discussion on the AVL-tree, which provides the following guarantees:

- $O(n)$ space consumption.
- $O(\log n)$ time per predecessor query (hence, also per dictionary lookup).
- $O(\log n)$ time per insertion
- $O(\log n)$ time per deletion.

All the above complexities hold in the worst case.