Algorithms (COMP3600/6466)
Data Structures: AVL Trees

Pascal Bercher

(working in the Intelligent Systems Cluster)

School of Computing
The Australian National University

Wednesday, 30.8.2023

Australian National University

Introduction

Motivation

Recap that we want to do (at least) the following operations efficiently:

- access, i.e., search
- insertion
- deletion
- min/max

Which runtime did we have for binary search trees? All were $O(h)$.

What about heaps? Most are $O(log(n))$, max is $O(1)$, search is $O(n)$.

So, *can* we even do better? *Depends on what we want to do!*

- Many max (resp. min) operations? $\rightarrow$ use heaps.
- Many searches? $\rightarrow$ use AVL trees!

We want to improve, but how?

How are we going to do better?

- Binary search tree: No guarantee on height!
- Heap: Has to be a complete tree (too restrictive)
- Now: A balanced tree! Still only logarithmic height, but does not have to be complete.

Recall that completeness meant that only in the last layer of the tree (on its right) nodes may be missing.

History Lesson

Why is it called AVL tree?

- It was invented by **A**del'soon-**Vel'skii** and **L**andis (in 1962)
- This was the first self-balancing search tree.

What means self-balancing?

- The tree makes sure that levels (heights) between siblings are "not too different" thus ensuring $O(log(n))$ height.
- For AVL trees: height between *left* and *right* child of each node differ by at most 1.
- (Later we will see a different self-balancing rule!)

Basics

AVL Trees

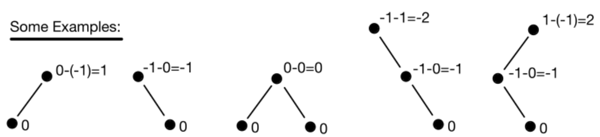
Recap on binary search trees: For all $x \in N$ holds:

- For all x' in the tree rooted in $x.left$ holds $x'.key < x.key$
- For all x' in the tree rooted in $x.right$ holds $x.key < x'.key$

An AVL tree is a binary search tree with the following properties:

- Each node x maintains its *balance factor* $bf(x)$.
  $bf(x) = height(leftSubtree(x)) - height(rightSubtree(x))$
  (Note that wikipedia uses an inverted definition)

  Conventions:
  - Height of a tree with one node: 0
  - Height of a tree with zero nodes: -1 (empty tree)
- For each node x in the tree must hold: $bf(x) \in \{-1, 0, 1\}$



AVL Trees

Recap on binary search trees: For all $x \in N$ holds:

- For all x' in the tree rooted in $x.left$ holds $x'.key < x.key$
- For all x' in the tree rooted in $x.right$ holds $x.key < x'.key$

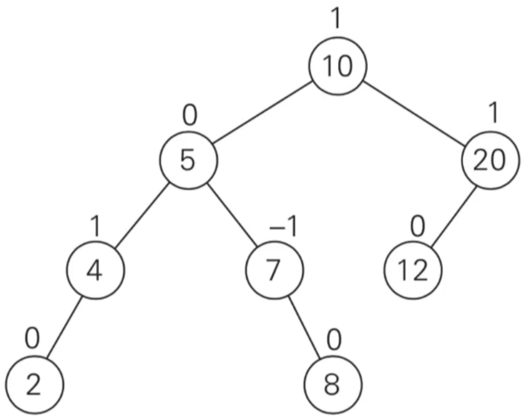
An AVL tree is a binary search tree with the following properties:

- Each node x maintains its *balance factor* $bf(x)$.
  $bf(x) = height(leftSubtree(x)) - height(rightSubtree(x))$
  (Note that wikipedia uses an inverted definition)

  Conventions:
  - Height of a tree with one node: 0
  - Height of a tree with zero nodes: -1 (empty tree)
- For each node x in the tree must hold: $bf(x) \in \{-1, 0, 1\}$
- Hint: Realize/remember that
  - $bf(x) = 0$ means: same height (trivial!)
  - $bf(x) < 0$ means: missing levels on the left
  - $bf(x) > 0$ means: missing levels on the right

Example

The numbers indicate the balance factors.



On the Height of AVL Trees

How to prove that the height h of an AVL tree with n nodes is always in $O(log(n))$?

- We show this for the worst case, i.e., for the deepest AVL tree.
- Such a tree has an imbalance everywhere! (Because then we can use all the missing nodes to construct a very deep path thus increasing the height.)
- Thus we know and define:
  - Let N_h be the minimum number of nodes for an AVL tree with height h.
  - Then, $N_h = 1 + N_{h-1} + N_{h-2}$
- Now compute N_h in relation to h!

On the Height of AVL Trees, cont'd

$$
\begin{aligned}
N_h &= 1 + N_{h-1} + N_{h-2} \\
&\geq 2 \cdot N_{h-2} \quad \text{// so we know: } N_h \geq 2 \cdot N_{h-2} \\
&= 2 \cdot 2 \cdot N_{h-4} \\
&= 2 \cdot 2 \cdot 2 \cdot N_{h-6} \\
&= 2^i \cdot N_{h-2 \cdot i} \\
&= \ldots \\
&= 2^{\frac{h}{2}} \cdot N_{h-2 \cdot \frac{h}{2}} = 2^{\frac{h}{2}} \cdot N_0 = 2^{\frac{h}{2}}
\end{aligned}
$$

Thus we get:

$$log(N_h) \geq \frac{h}{2} \quad \text{and thus:} \quad 2 \cdot log(N_h) \geq h$$

Recall that N_h is n, the number of nodes (we called it N_h to enforce the worst-case property of having imbalances to get a maximal h).

Rotations

Rotations

How to achieve the AVL property?

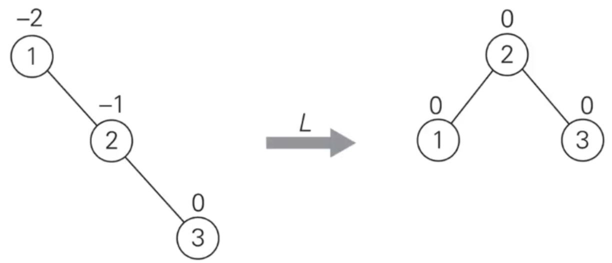
- AVL trees rely on a transformation operation called *rotation* to re-balance the tree.
- There are four rotations in total:
  - 2 “classes”, and
  - 2 types per class
- In the following slides, we always rotate some node “x”. This is always the node that is *unbalanced*, i.e., the one with a balance factor $= +/-2$.

L-rotation

L-rotation on node x:

Rotate the edge connecting x (here: 1) and its right child.

Example:

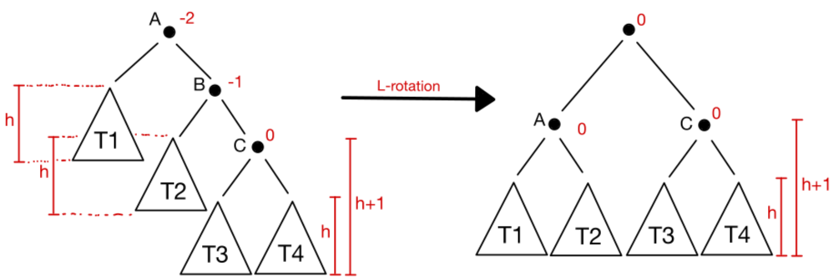


L-rotation

L-rotation on node x:

Rotate the edge connecting x (here: 1) and its right child.

Example:

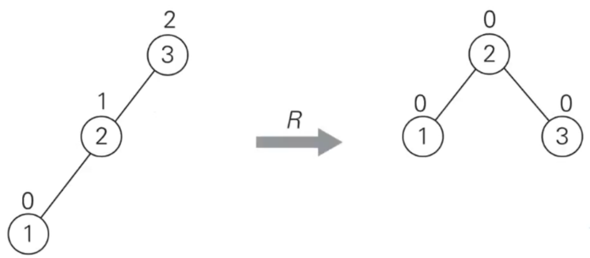


R-rotation

R-rotation on node x:

Rotate the edge connecting x and its left child.

Example:

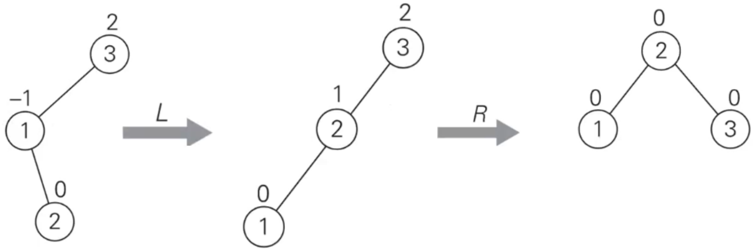


LR-rotation

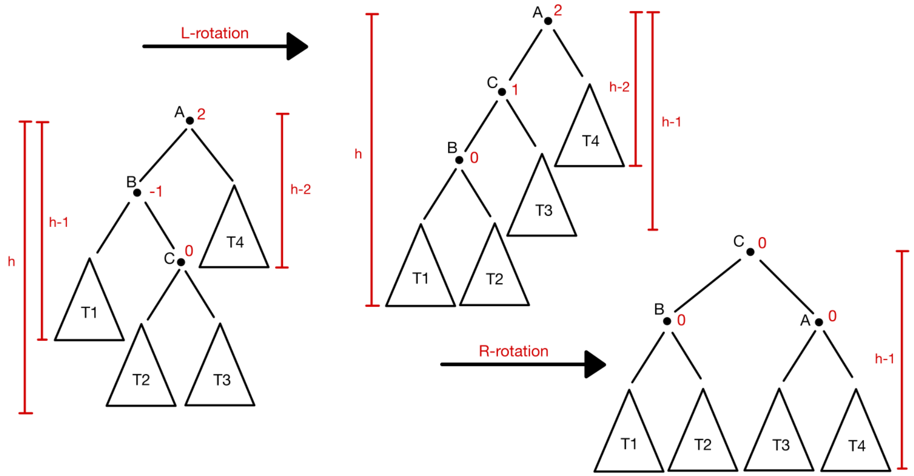
LR-rotation on node x: (that's a double-rotation)

L-rotation on the root of x's left subtree followed by R-rotation on x.

Example:



LR-rotation, (Generic) Example

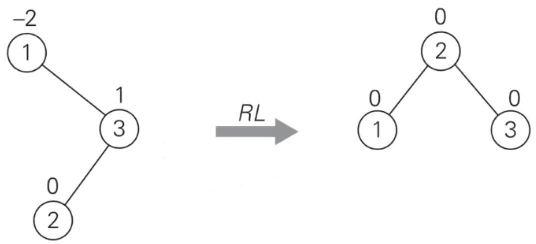


RL-rotation

RL-rotation on node x: (that's the other double-rotation)

R-rotation on the root of x's right subtree followed by L-rotation on x.

Example:



Rotations, Summary

How to decide when to do which rotation?

We just have four cases:

- $bf(x) = -2$
  - $bf(x.right) = -1$: L-rotation
  - $bf(x.right) = 1$: RL-rotation
- $bf(x) = 2$
  - $bf(x.left) = 1$: R-rotation
  - $bf(x.left) = -1$: LR-rotation

Why do we need rotations?

To re-establish AVL properties after we modified the tree (with insertion/deletes, see next slides).

Insertion

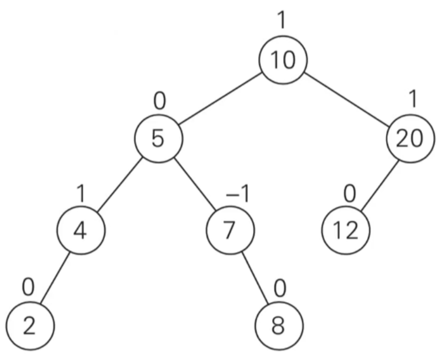
Algorithm

How to insert? Perform binary search tree insertion and then rebalance whenever necessary. Thus, to insert node x into tree T do:

- Insert x as if T were a usual binary search tree.
- Let z be the parent of the newly inserted node.
- While z is not the root node do:
  - Update balance factor of z.
  - If z violates the AVL property, rotate!
  - Set z as $z.parent$.

Example

Insert 9, then insert 15 into the following tree:



Let z be the parent of the newly inserted node. While z is not the root node do:

- Update balance factor of z.
- If z violates the AVL property, rotate!
- Set z as $z.parent$.

Runtime

- A single insertion requires only one single rotation (assuming that we insert into an AVL tree) and is thus a constant-time operation.
- However, we also need to find the right position to insert the new key into! This takes $O(log(n))$

Deletion

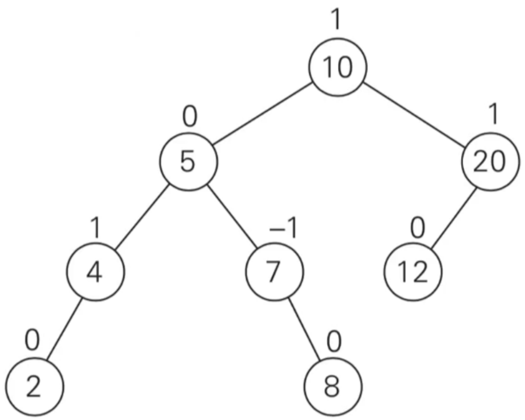
Algorithm

How to delete?

- Perform binary search tree deletion! (See yesterday's lecture; the complicated case is when the deleted node has two children.)
- Update balance factors.
- Perform rebalancing along the respective path whenever required, starting at the bottom.

Example

Delete 7, then delete 12 from the following tree:



Runtime

- We first need to find the key to delete, which costs $O(log(n))$.
- Then we have to perform (potentially multiple) rotations along the respective path, which gives another $O(log(n))$.
- Thus, in total, we have $O(log(n))$.

Summary

Summary

Today we covered **AVL trees**.

They are improved *binary search trees*, which are self-balancing.

Operations considered:

- Rotations to achieve balanced tree $O(1)$
- Insertion $O(log(n))$
- Deletion $O(log(n))$
- What about search? $O(log(n))$
- What about min/max? $O(log(n))$