Switch to a new session

This session has ended, but it looks like you still have a few items to complete. Join the next session to finish the course - your progress from this session will transfer with you.

Join new session

AVL, Red-Black and Splay Trees

In the last two modules, we study AVL trees and Splay trees. There is another data structure that is often used as a balanced binary search tree called Red-Black tree. The main reasons we don't cover them are that they are very similar to AVL trees in terms of the balancing idea (both use rotations to balance the tree), and the algorithms for insertion and deletion in the Red-Black tree are considerably more complex (especially deletion).

In theory, AVL trees should have faster lookups because of more rigid balance and better guarantees on the depth - $1.44\log(n)$ for AVL and $2\log(n)$ for RB. However, the insertions should be faster in the RB trees, because $O(1)$ amortized insertion time can be proved for them, while AVL requires $O(\log(n))$ rotations per insertion. In terms of memory consumption, RB trees require 2 or 3 pointers per node (left and right child and optionally parent) and a bit for color (red or black). AVL trees require 2 pointers and a field for the current balance per node.

Both Red-Black trees and AVL trees are often used in practice. However, Red-Black trees are used in the implementation of C++ STL set and map, and also Java TreeSet and TreeMap.

In practice, the performance of AVL and RB trees is very similar and the comparison depends on the implementation used. Here are two benchmarks 1 and 2 comparing different AVL implementations to the standard C++ STL library RB tree implementation. Both show superior performance of AVL trees in most tests, but the difference is not very significant. In terms of memory footprint, if RB tree stores the parent pointer, it automatically uses more memory per node (3 pointers + color bit), and if it doesn't store parent pointer, it works slower, and also the bit for node color still uses space of another pointer due to memory alignment reasons. So, AVL tree uses either the same memory or is a little more memory efficient.

Speaking about Splay trees, we describe them for the following reasons. They are relatively easy to implement. They adapt dynamically to data. They are mergeable - allow split and merge operations. AVL and RB trees can be made mergeable, but the corresponding algorithms are very complex. Splay trees also require just two pointers per node and no additional information.