Splay trees outline

- Splay tree concept
- Splaying operation: double-rotations
- Splay insertion
- Splay search
- Splay deletion
- Running time of splay tree operations

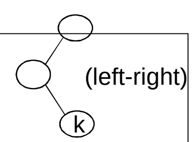
Splay trees

- A splay tree is a balanced BST built using an amortization approach –
  - that is, do more work after every insertion or search to save subsequent search time.
- "More work":
  - What: Bring the accessed (i.e., inserted or searched) node to the root.
  - Why: It pays off if the accessed node is accessed again soon.
  - How: Double-rotate – called "splaying."

Splaying operation

- Splay(k): move a node k to the root through double rotation operations.
  - Single rotation is not used in the splay tree.
- As a result, halve the depth of most nodes (excluding the root) encountered on the path from the root to the accessed node.
- Splaying *tends to* -- not always -- keep the splay tree more balanced than the BST.

Splay double-rotations

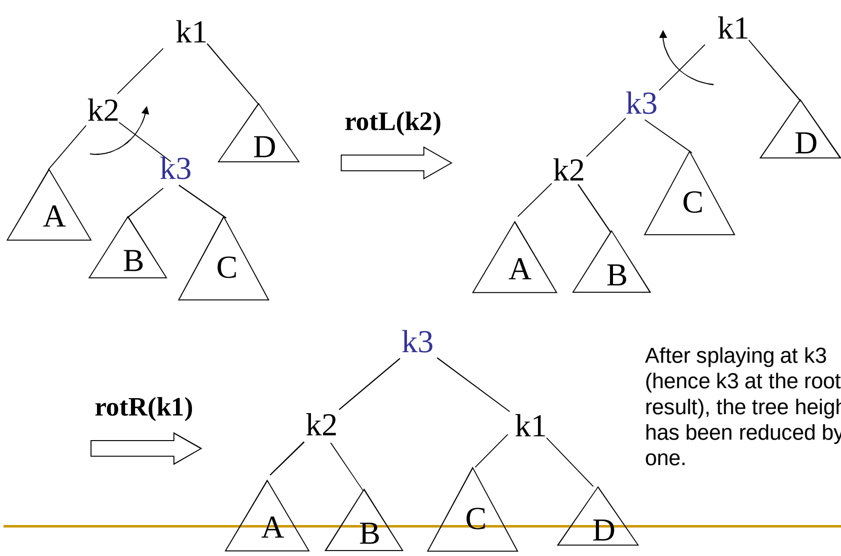


- Two cases for splay(k):
  - Zig-zag: left-right or right-left
    - The order of rotations is bottom-up.
    - It reduces the tree height by one.
    - This case is the same as the AVL tree.
  - Zig-zig: left–left or right–right
    - The order of rotations is top-down.
    - It tends to reduce the distances to most nodes encountered (except the root) to about half.
    - This case is different from the AVL tree, and brings the positive effect of splaying.

Summary: rotations in AVL and splay

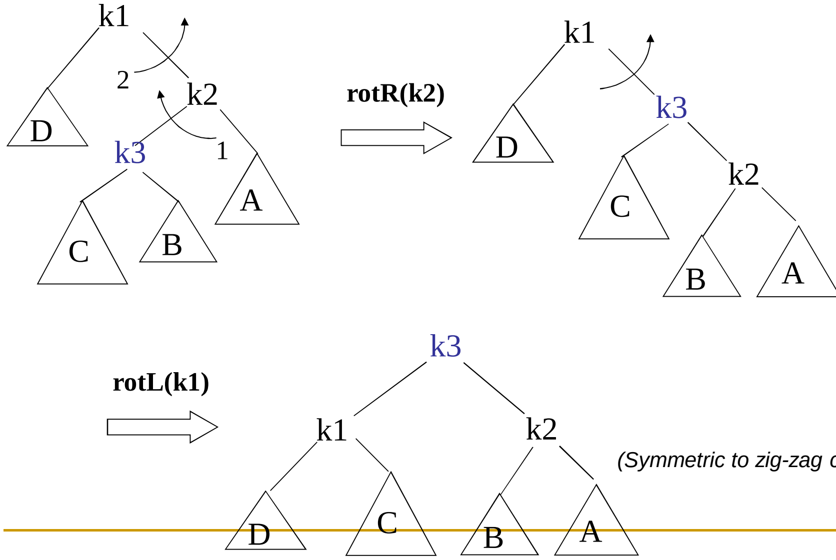
Tree type / Case	AVL tree	Splay tree
zig-zig	single rotation	top-down double rotation
zig-zag	bottom-up double rotation	bottom-up double rotation

Splay(k3): zig-zag case I



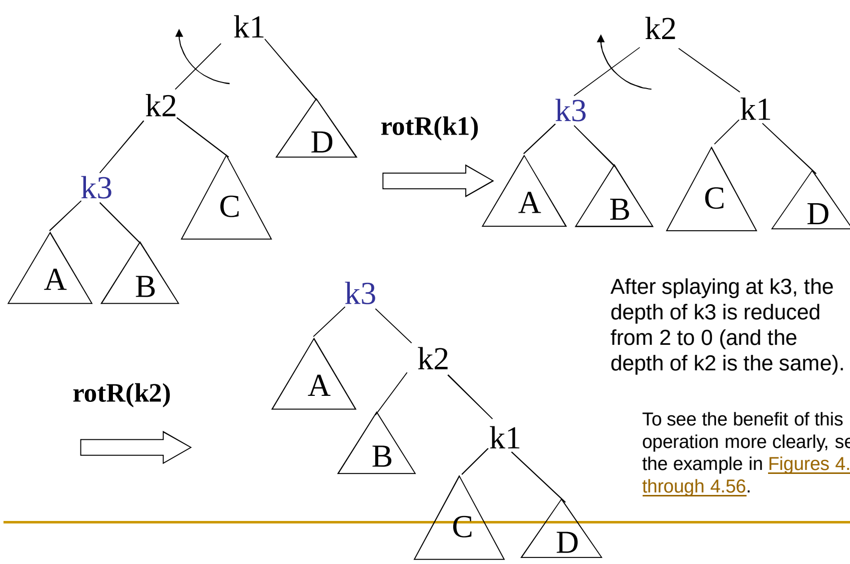
After splaying at k3 (hence k3 at the root as a result), the tree height has been reduced by one.

Splay(k3): zig-zag case II



(Symmetric to zig-zag case I)

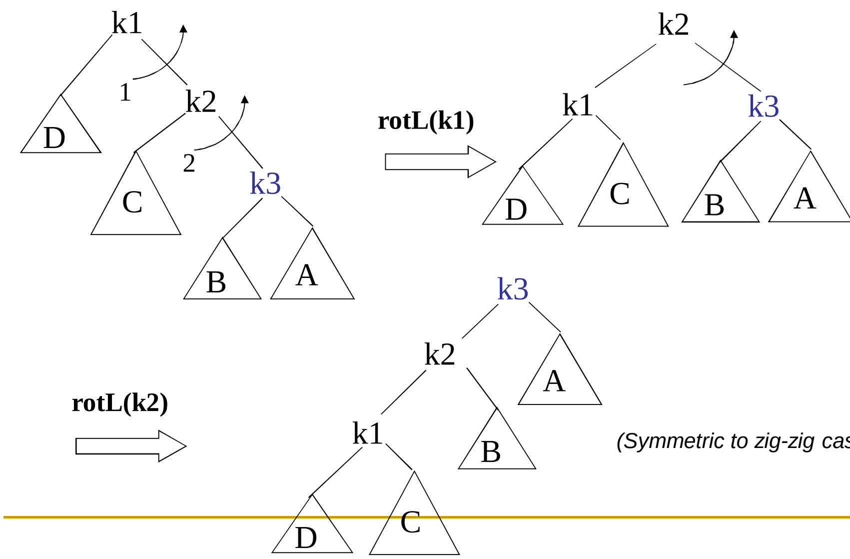
Splay(k3): zig-zig case I



After splaying at k3, the depth of k3 is reduced from 2 to 0 (and the depth of k2 is the same).

To see the benefit of this operation more clearly, see the example in Figures 4.48 through 4.56.

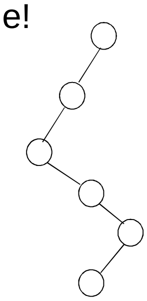
Splay(k3): zig-zig case II



(Symmetric to zig-zig case I)

What if the access path length is an odd number?

- One rotation must be a single rotation.
- Do it either at the bottom or at the top.
  - The resulting tree structures may be different. Both are fine!



Let's do it at the bottom in this class.

Splay insertion algorithm

insert(node) {

1. Insert the new node.
2. Let *n* be the length of the path traversed for the insertion.

   Then,

   a. If $n = 2m$ (i.e., even number)
      then perform *m* double rotations.

   b. If $n = 2m+1$ (i.e., odd number)
      then perform *one* single rotation followed by
      m double rotations.

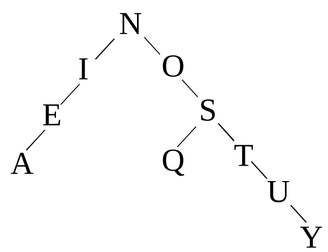
}

Exercise: splay tree insertion

- Construct a splay tree by inserting B, Y, U, A, G in sequence.
- Q. *What if the node already exists?*
- A. *Splay from the existing node.*
- (See the answer key.)

Exercise: splay tree search

- Search the tree below with the following keys in sequence: E, A, S, Y, Q, U, T, I, O, and N.



See the answer key.

- *Q. What if a node is not found?*
- *A. Splay from the last node visited.*

Splay deletion: remove(node)

- 1. Splay-search the node. (That is, find the node and rotate it up to the root.)
- 2. Find the largest node in the left subtree of the root (or find the smallest node in the right subtree) and rotate it up to replace the root.

Splay deletion: remove(node) – side note

- Q. What is the point of bringing the node to the root after finding it, only to remove it?
  - The rationale for splaying is that a node accessed now is likely to be accessed again soon.
  - But, in the case of remove(node), the node is removed (so never accessed again) after all the work done to splay it up to the root.
- A. Splaying tends to shorten the access path to every node encountered while finding the node.
  - Recall the demo of Figures 4.47 to 4.56 in the textbook and see how the depth of each node is reduced at each splaying, especially in the beginning with a bunch of zig-zig cases. Of course, this depth reduction does not happen all the time, but overall we see the trend.

Top-down splay tree

- Bottom-up splay tree walks the access path twice – once for searching and once for splaying.
- Top-down splay tree does the splaying while walking down the access path.
  - Section 12.1

Top-down splay tree -- side note

- The splay tree has been shown to be faster than the AVL tree in benchmark experiments performed using "real" applications.
  - All balanced BSTs we are studying are main-memory data structures and, thus, the run-time is very sensitive to the code implementation. Splay tree operations can be coded more efficiently (I mean, the top-down splay tree) than the AVL tree operations. This could be one reason for the benchmark results being in favor of the splay tree.

Running time of splay tree operations

- A single operation: O(logN) average, O(N) worst. But, the worst case occurs less frequently than the BST.
- Searching for M nodes: O(M·logN) for both average and worst.
- Inserting N nodes: O(N·logN) for both average and worst.
- So, the amortized running time of a single operation is O(logN).
- **Note**. Compare with the BST case, where searching for M nodes and inserting N nodes take O(M·N) and $O(N^2)$, respectively.

Running time of splay tree operations (cont.)

- Analyzing the running time of splay operations involves amortized algorithm analysis. This analysis appears in Section 11.5, but is beyond our scope here.
  - What's in Section 11.5?
    - Consider the *potential function* Φ of a splay tree T as: $\Phi(T) = \sum_{x \in T} R(x)$, where the rank of a node x, R(x), is defined as logS(x) where S(x) is the number of descendants of a node x (including x itself) in T. Then, the amortized running time to splay a tree at node x is at most $3(R(T)–R(x))+1 = 3(\log N – R(x))+1 = O(\log N)$, where R(T) is the rank of the root of T.
      - Proof of this claim appears in page 533.