

Chapter 22

Splay Trees

Introduction

- Splay trees support all the operations of binary trees.
- But they do not guarantee $O(\log N)$ worst-case performance.
- Instead, its bounds are amortized, meaning that although individual operations can be expensive, any sequence of operations is guaranteed to behave as if each operation in the sequence exhibited logarithmic behavior.

Self-adjusting and Amortized Analysis

- Limitations of balanced search trees:
 - Balanced search trees require storing an extra piece of information per node.
 - They are complicated to implement, and as a result insertions and deletions are expensive and potentially error-prone.
 - We do not win when easy inputs occur.
- Balanced search trees have a second feature that seems improvable:
 - Their worst-case, average-case, and best-case performance are essentially identical.
 - It would be nice if the second access to the same piece of data was cheaper than the first.
 - *The 90-10 rule* – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items.
 - It would be nice to get easy wins for the 90% case.

Amortized Time Bounds

- Given the above information, what can we do?
- Since the time to access in a binary tree is proportional to the depth of the accessed node, we can attempt to restructure the tree by moving frequently accessed items toward the root.
- Even if there are intervening operations, we would expect the node to remain close to the root and thus be quickly found.
- We could call this strategy the *rotate-to-root strategy*.
- An application of the rotate-to-root strategy to node 3 is shown in Figure 1.

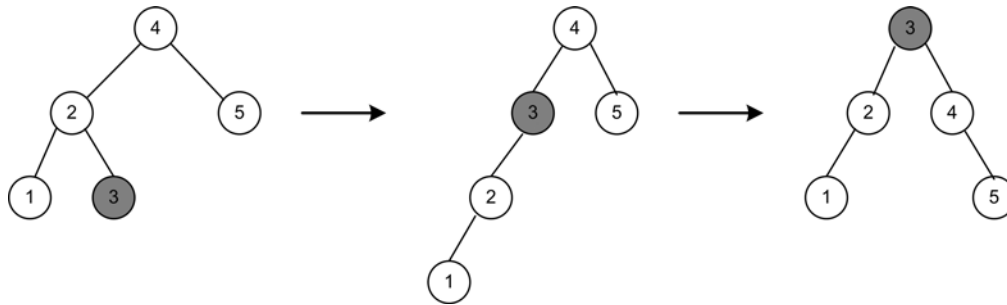


Figure 1 Rotate-to-Root

- When nodes are rotated to the root, other nodes go to lower levels in the tree.
- If we do not have the 90-10 rule, it is possible for a long sequence of bad accesses to occur.

The Basic Bottom-Up Splay Tree

- A technique called *splaying* can be used so a logarithmic amortized bound can be achieved.
- We use rotations such as we've seen before.
- The *zig* case.
 - Let X be a non-root node on the access path on which we are rotating.
 - If the parent of X is the root of the tree, we merely rotate X and the root as shown in Figure 2.

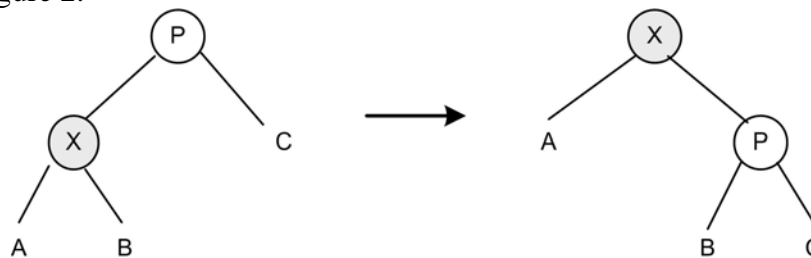


Figure 2 Zig Case

- This is the same as a normal single rotation.
- The *zig-zag* case.
 - In this case, X and both a parent P and a grandparent G . X is a right child and P is a left child (or vice versa).
 - This is the same as a double rotation.
 - This is shown in Figure 3.

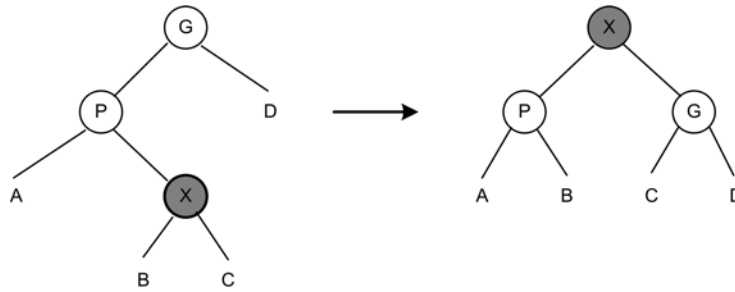


Figure 3 Zig-zag Case

- The *zig-zig* case.
 - This is a different rotation from those we have previously seen.
 - Here X and P are either both left children or both right children.
 - The transformation is shown in Figure 4.
 - This is different from the rotate-to-root. Rotate-to-root rotates between X and P and then between X and G . The zig-zig splay rotates between P and G and X and P .

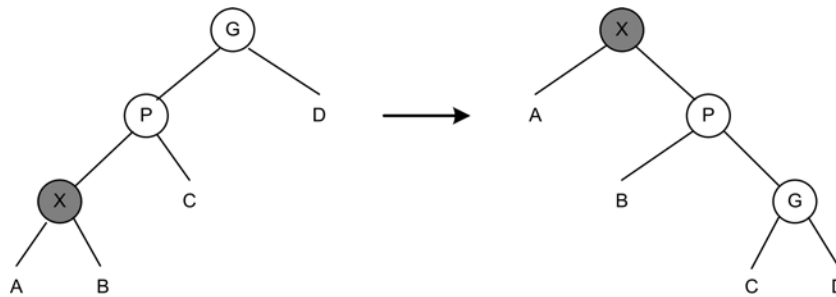


Figure 4 Zig-zig Case

- Given the rotations, consider the example in Figure 5, where we are splaying c .

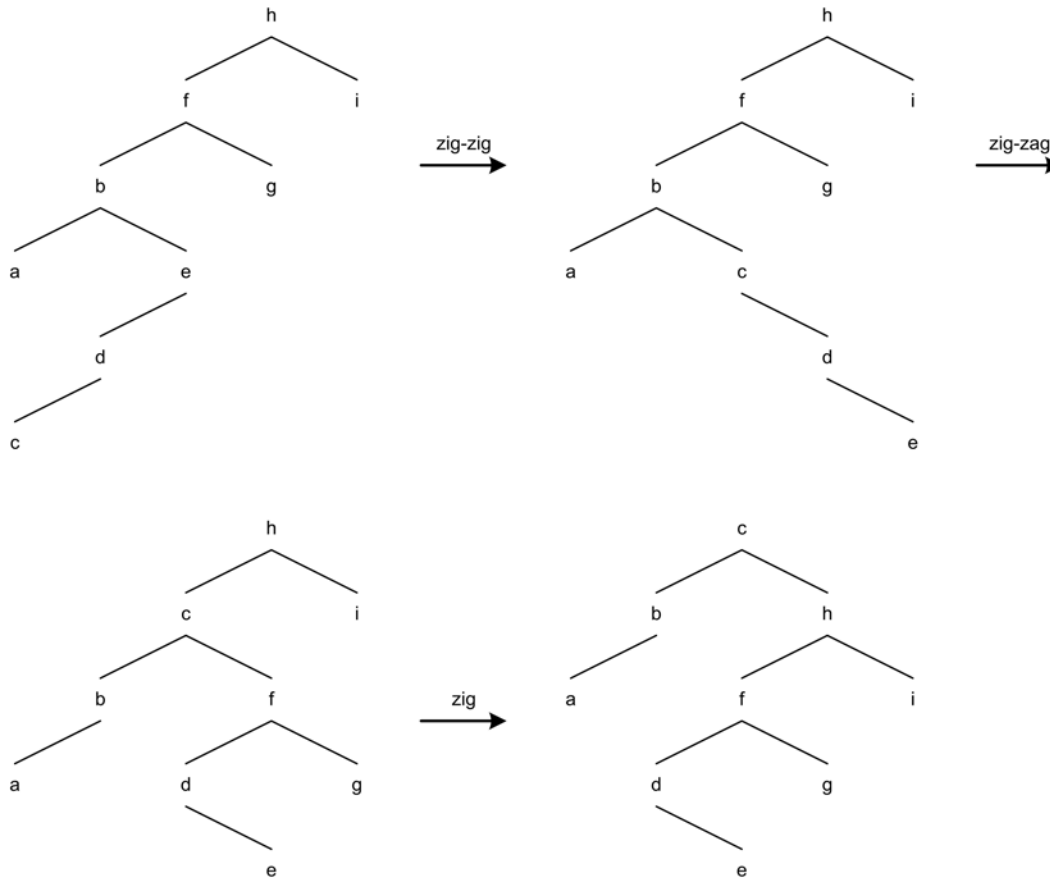


Figure 5 Example of Splay Tree

Basic Splay Tree Operations

- Insertion – when an item is inserted, a splay is performed.
 - As a result, the newly inserted item becomes the root of the tree.
- Find – the last node accessed during the search is splayed
 - If the search is successful, then the node that is found is splayed and becomes the new root.
 - If the search is unsuccessful, the last node accessed prior to reaching the NULL pointer is splayed and becomes the new root.
- FindMin and FindMax – perform a splay after the access.
- DeleteMin and DeleteMax – these are important priority queue operations.
 - DeleteMin
 - Perform a FindMin.
 - This brings the minimum to the root, and by the binary search tree property, there is no left child.
 - Use the right child as the new root and delete the node containing the minimum.
 - DeleteMax

- Perform a FindMax.
 - Set the root to the post-splay root's left child and delete the post-splay root.
- Remove
 - Access the node to be deleted bringing it to the root.
 - Delete the root leaving two subtrees L (left) and R (right).
 - Find the largest element in L using a DeleteMax operation, thus the root of L will have no right child.
 - Make R the right child of L's root.
 - As an example, see Figure 6.

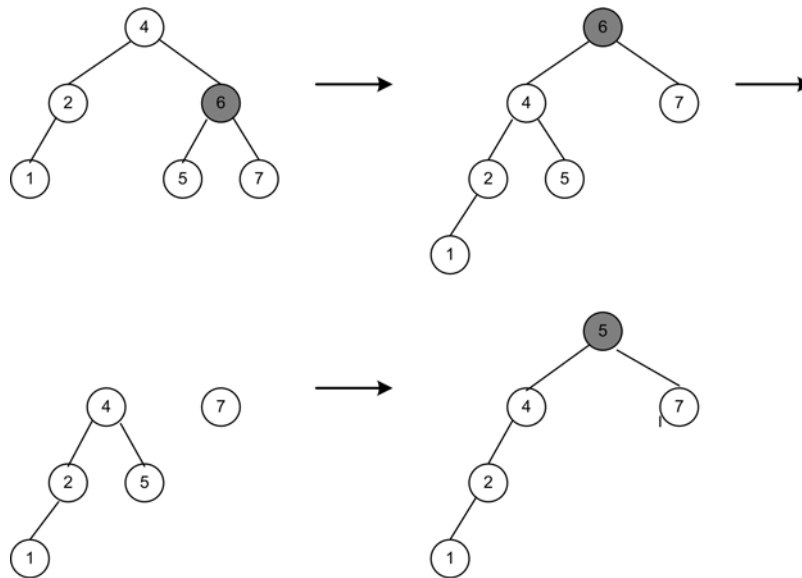


Figure 6 Remove from Splay Tree

Top-Down Splay Trees

- Bottom-up splay trees are hard to implement.
- We look at top-down splay trees that maintain the logarithmic amortized bound.
 - This is the method recommended by the inventors of splay trees.
- Basic idea – as we descend the tree in our search for some node X, we must take the nodes that are on the access path, and move them and their subtrees out of the way. We must also perform some tree rotations to guarantee the amortized time bound.
- At any point in the middle of a splay, we have:
 - The current node X that is the root of its subtree.
 - Tree L that stores nodes less than X.
 - Tree R that stores nodes larger than X.
- Initially, X is the root of T, and L and R are empty.
- As we descend the tree two levels at a time, we encounter a pair of nodes.
 - Depending on whether these nodes are smaller than X or larger than X, they are placed in L or R along with subtrees that are not on the access path to X.

- When we finally reach X, we can then attach L and R to the bottom of the middle tree, and as a result X will have been moved to the root.
- We now just have to show how the nodes are placed in the different tree. This is shown below:

- Zig rotation – Figure 7

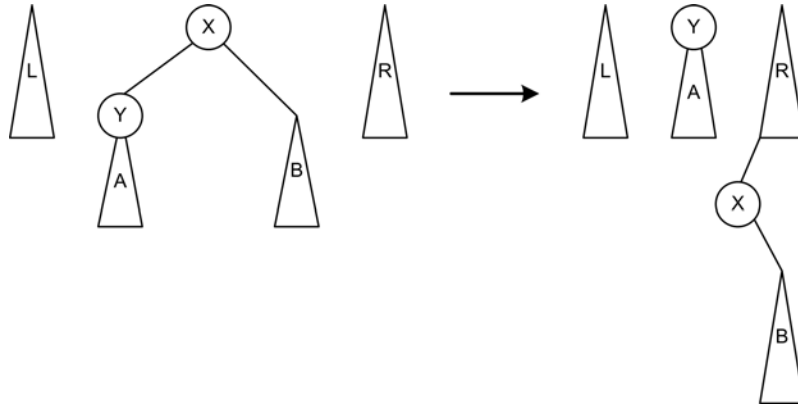


Figure 7 Zig

- Zig-Zig – Figure 8

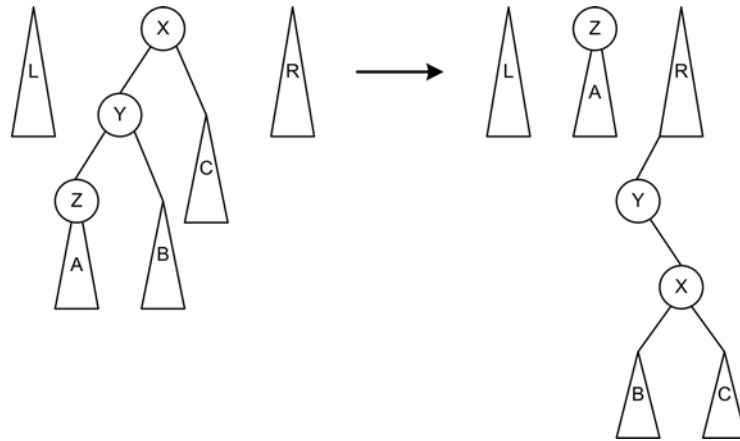


Figure 8 Zig-Zig

- Zig-Zag – Figure 9

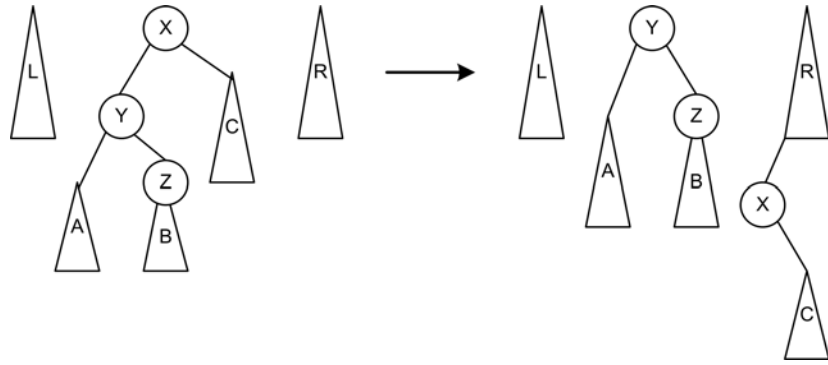


Figure 9 Zig-Zag

- Notice that the zig-zag is simplified.
 - Making Y the root simplifies the coding because the action for the zig-zag becomes identical to the zig case and would be advantageous, as testing for a host of cases is time-consuming.
 - The disadvantage is that a descent of only one level results in more iterations in the splaying procedure.
- The only thing we have to do now is reassemble the splay tree. This is shown in Figure 10

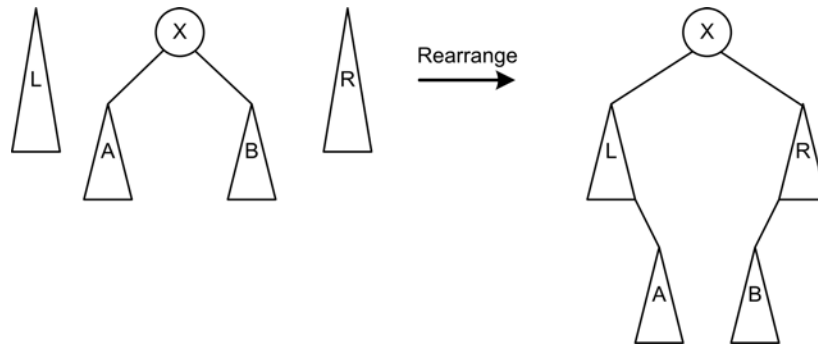


Figure 10 Reassembling

- See Figure 11 for an example of these rotations when accessing 19.

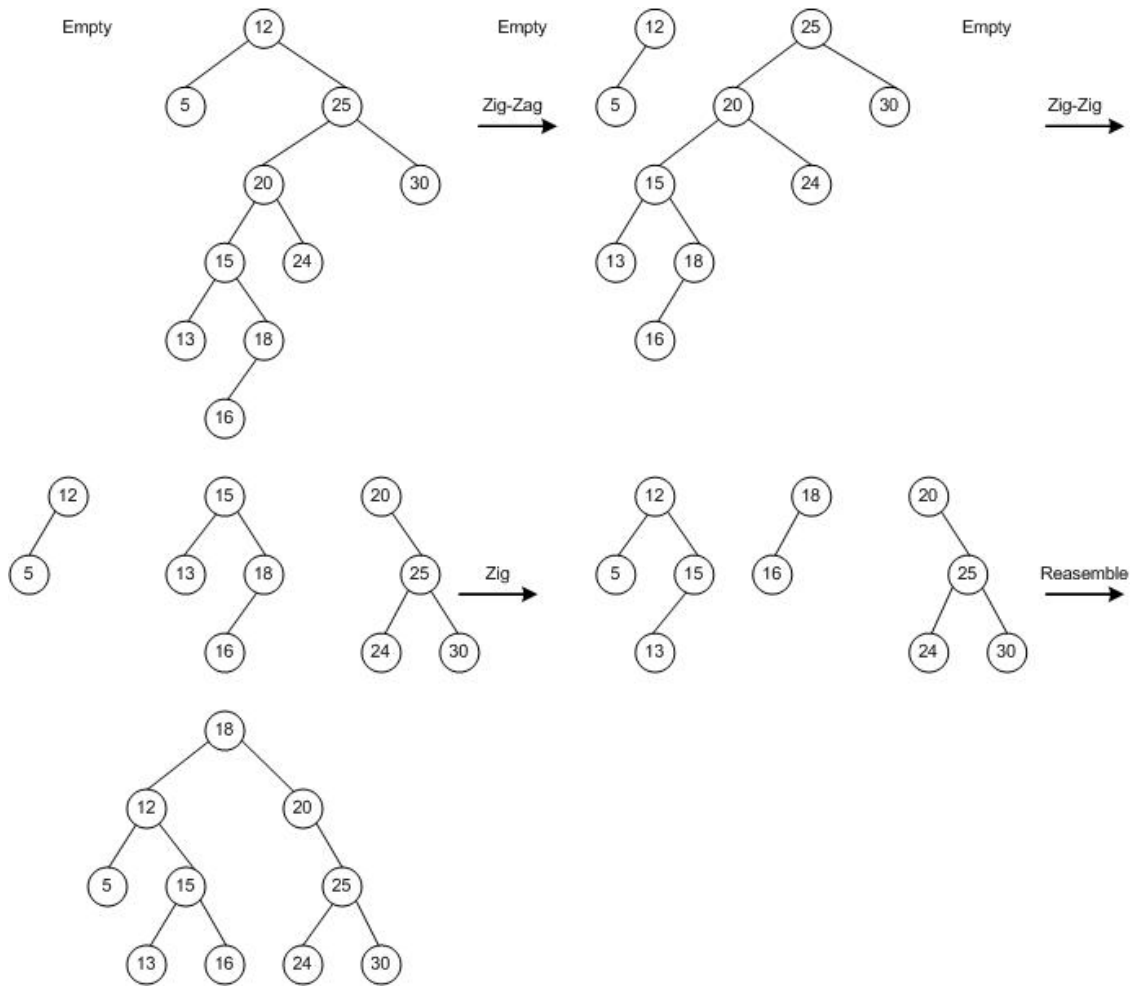


Figure 11 Example 1 of Splaying

- Another example when finding c is shown in Figure 12.

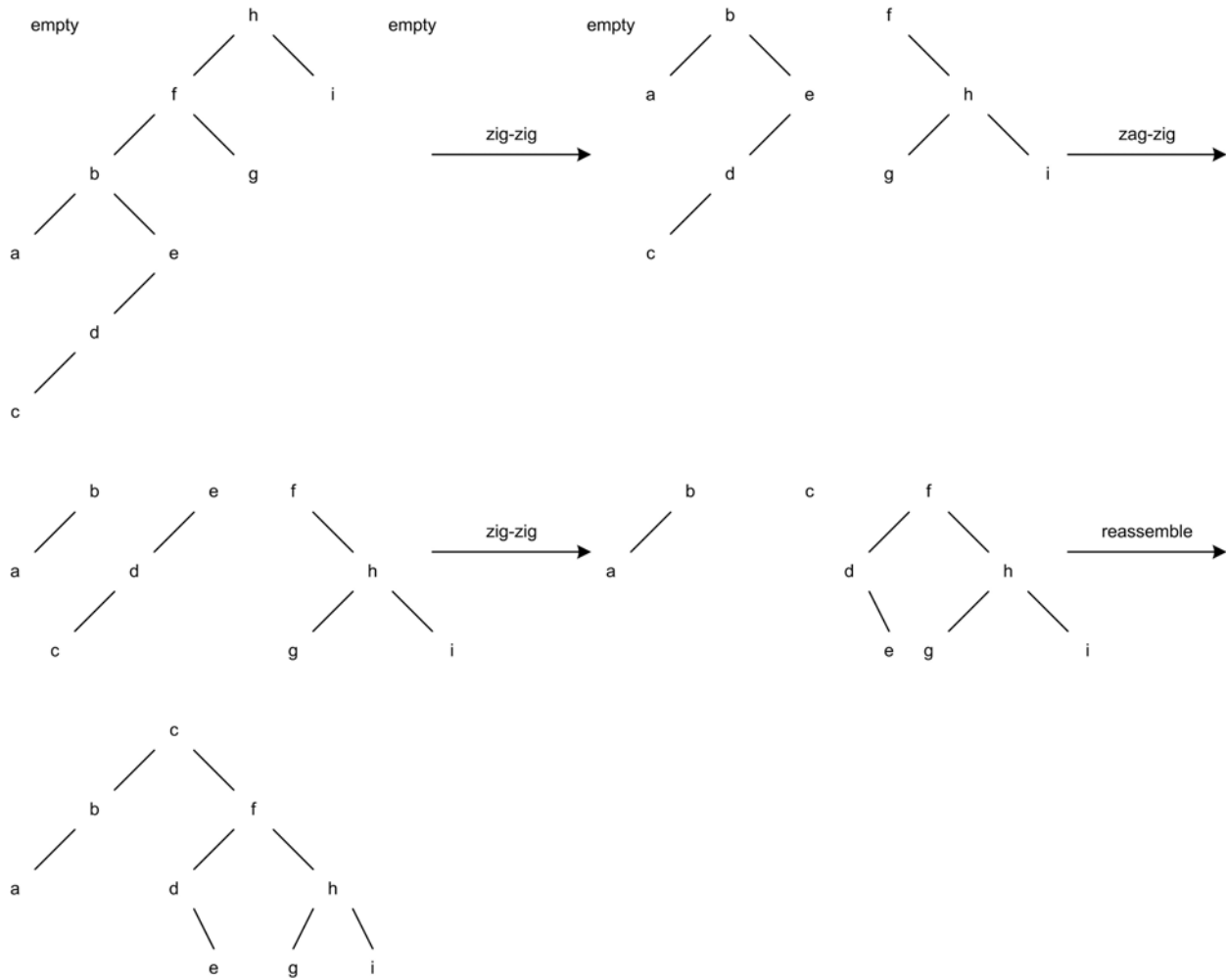


Figure 12 Example 2 of Splaying