Splay Tree Data Structure

261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

Background

AVL trees are binary search trees with logarithmic height

- This ensures all operations are $O(\ln(n))$
- tree is always balanced after an insert or delete

An alternative to maintaining a height logarithmic with respect to the number of nodes, an alternative idea is to make use of an old maxim:

> Data that has been recently accessed is more likely to be accessed again in the near future.

Self Adjusting Trees

- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
- Self-adjusting trees get reorganized over time as nodes are accessed
  - Tree adjusts after insert, delete, or find

Background

Accessed nodes could be rotated or *splayed* to the root of the tree:

- Accessed nodes are splayed to the root during the count/find operation
- Inserted nodes are inserted normally and then splayed
- The parent of a removed node is splayed to the root

Invented in 1985 by Daniel Dominic Sleator and Robert Endre Tarjan

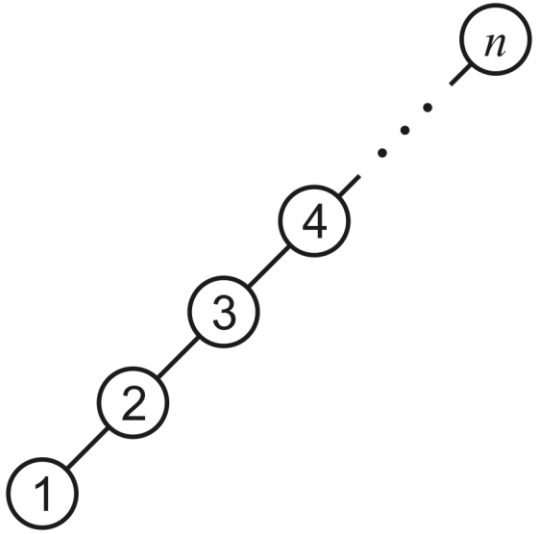
Splay Trees

- Splay trees are tree structures that:
  - Are not perfectly balanced all the time
  - Data most recently accessed is near the root.
  - Improve accessing speed (average case) for some data that are more frequently accessed than the others
- The procedure:
  - After node X is accessed, perform “splaying” operations to bring X to the root of the tree.
  - Do this in a way that leaves the tree more balanced as a whole

Insertion at the Root

Immediately, inserting at the root makes it clear that we will still have access times that are $O(n)$:

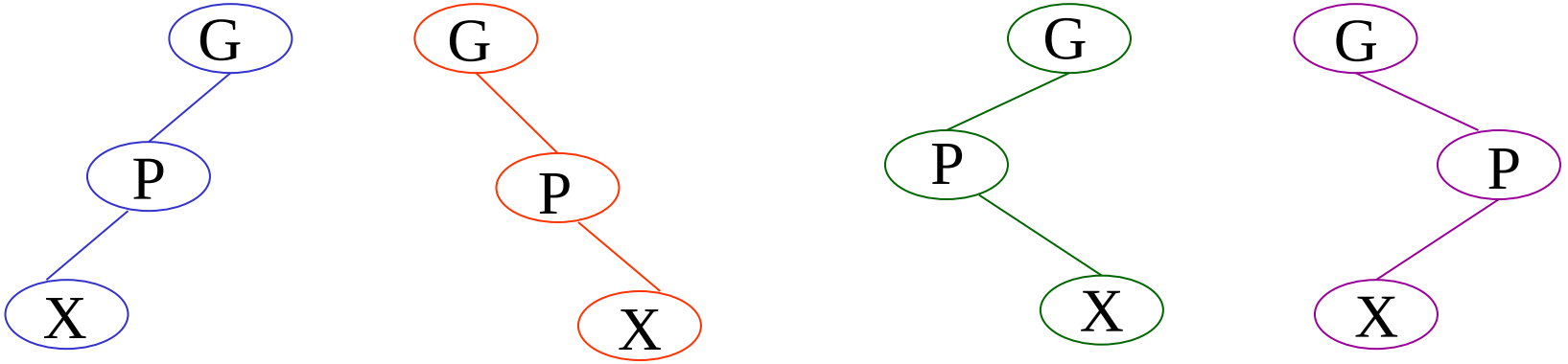
- Insert the values n, n-1, n-2, …, 4, 3, 2, 1, in that order



- Now, an access to 1 requires that a linked list be traversed

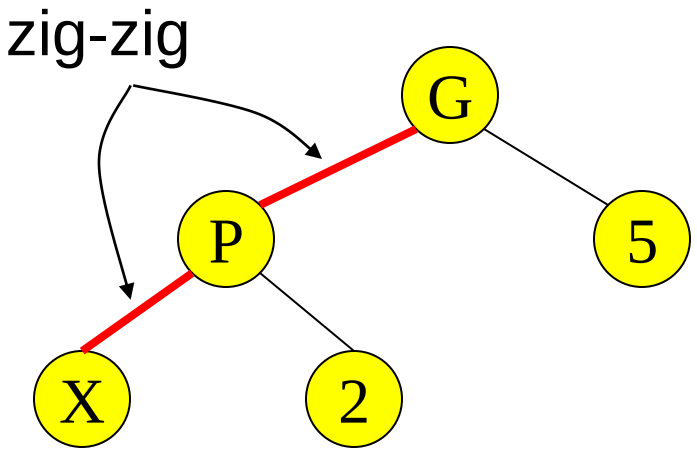
Splay Tree Terminology

- Let X be a non-root node with ≥ 2 ancestors.
- P is its parent node.
- G is its grandparent node.

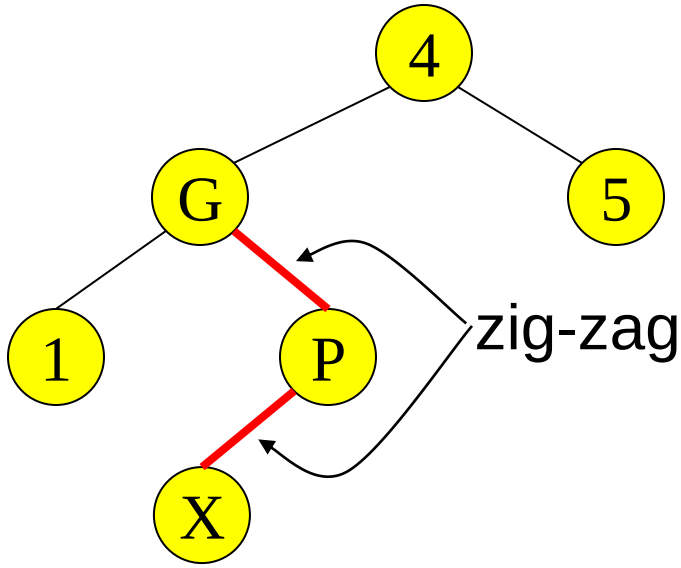


Zig-Zig and Zig-Zag

Parent and grandparent
in same direction.

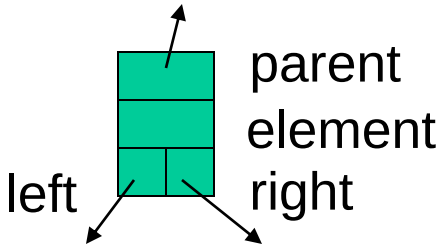


Parent and grandparent
in different directions.



Splay Tree Operations

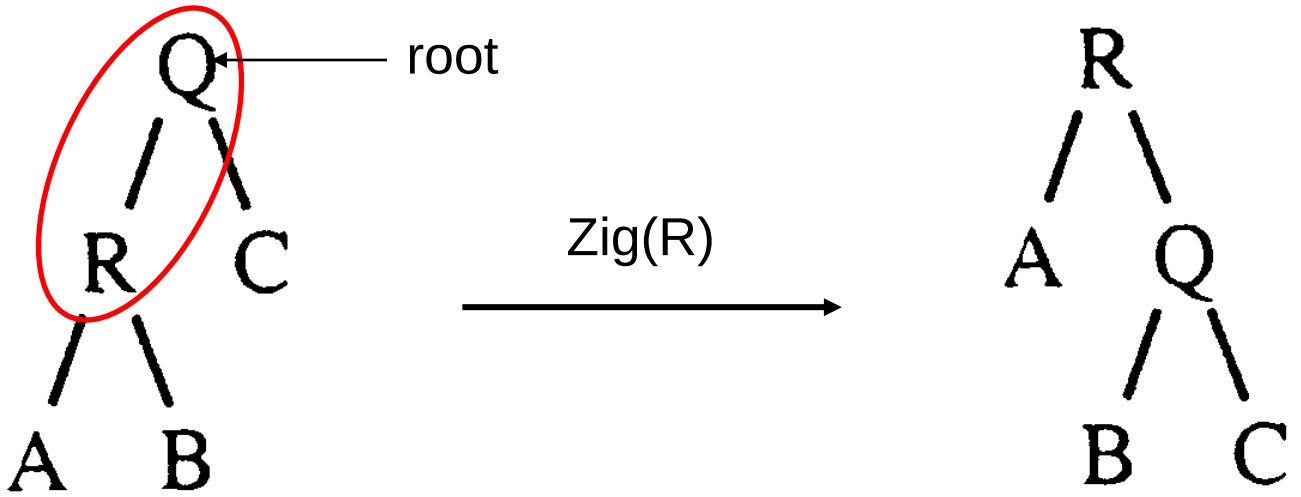
1. Helpful if nodes contain a parent pointer.



2. When X is accessed, apply one of six rotation routines.
   - Single Rotations (X has a P (the root) but no G)
     ZigFromLeft(X), ZigFromRight(X)
   - Double Rotations (X has both a P and a G)
     ZigZigFromLeft(X), ZigZigFromRight(X)
     ZigZagFromLeft(X), ZigZagFromRight(X)

Zig at depth 1 (root is the parent)

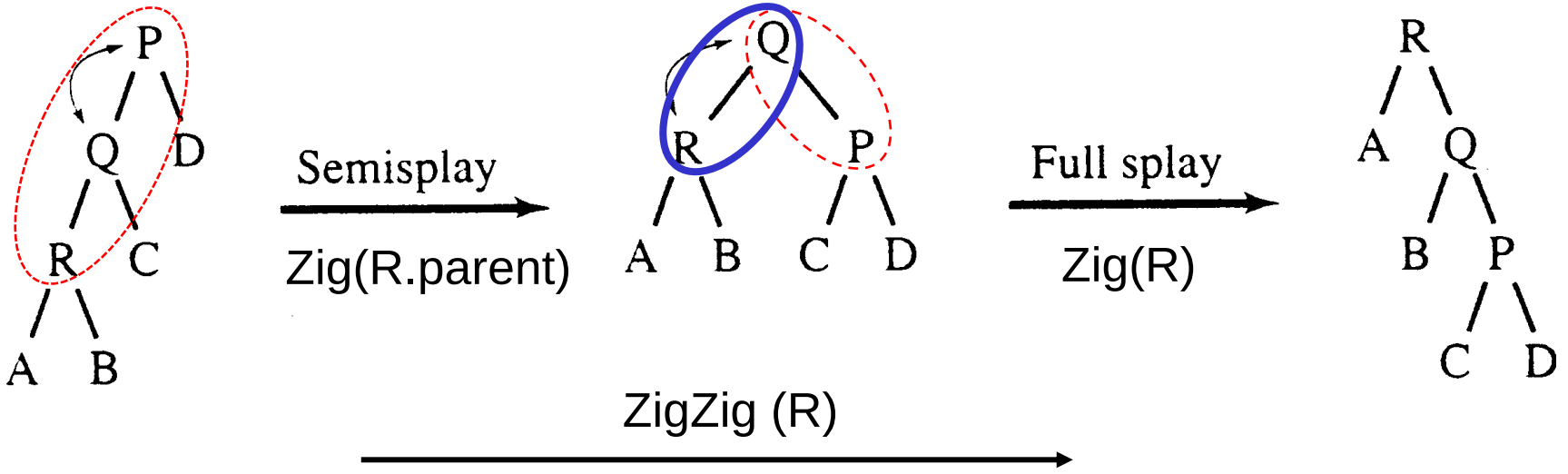
- Suppose R is now accessed using Find
- By splaying R, R should be moved at the top



- Zig operation is similar to “single rotation” except the input is the child node
- ZigFromLeft moves R up to the top

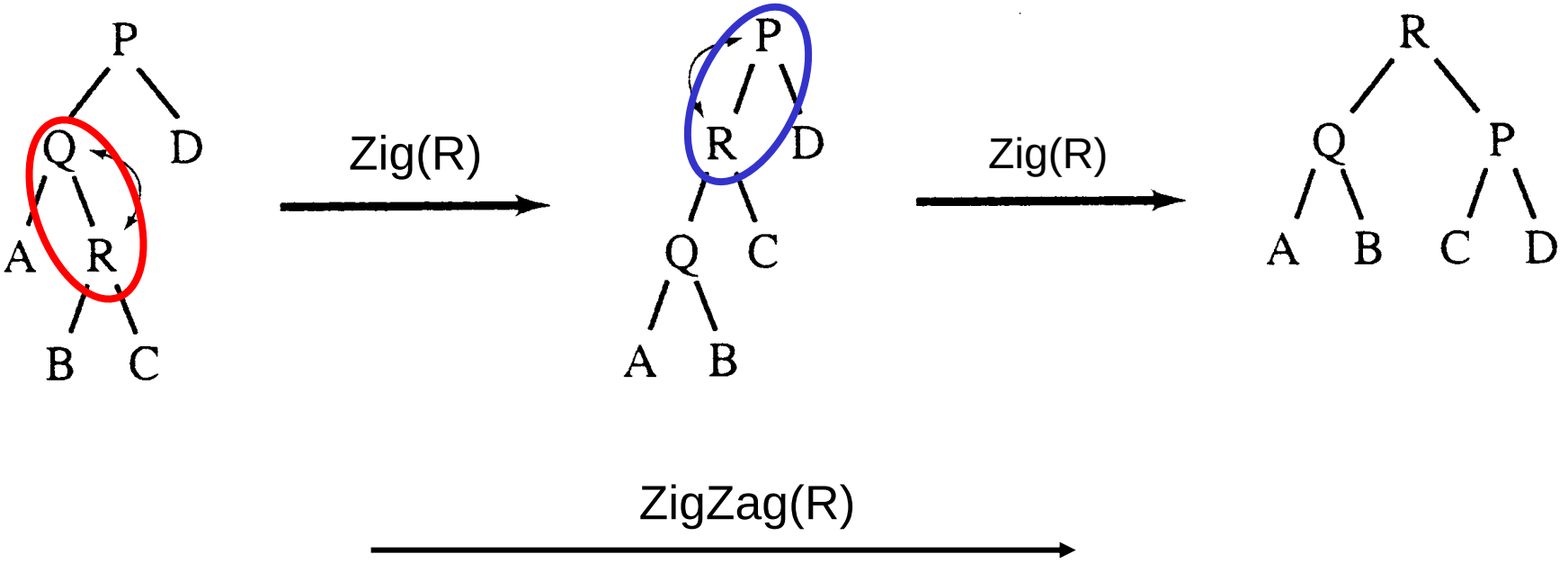
Zig-Zig operation (At depth 2+)

- "Zig-Zig" is similar to DoubleRotationFromOuterNode except:
  - the input is the child
  - the parent-grandparent switch roles first before switching with the child



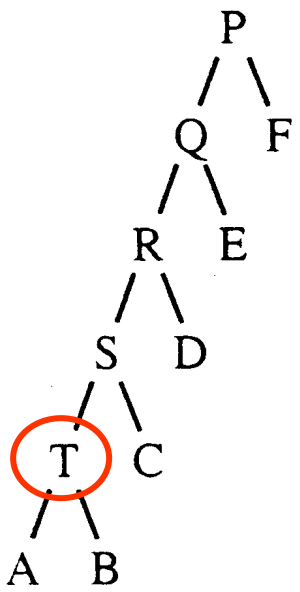
Zig-Zag operation (At depth 2+)

- "Zig-Zag" is the same as DoubleRotationFromInnerNode
- R is the inner relative to the grandparent (P)
- Child ->Parent; Child -> Grandparent
- Input is the child (R)

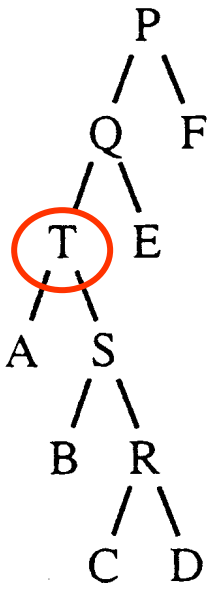


Autobalance (Decreasing Depth)

Tree Height 5



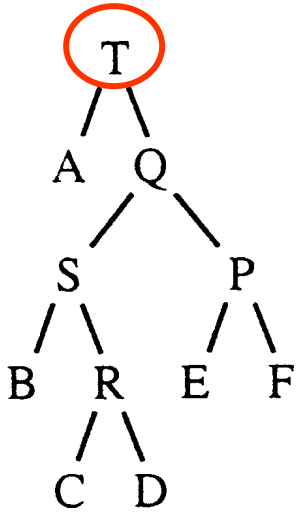
(a)



(b)

Find(T)

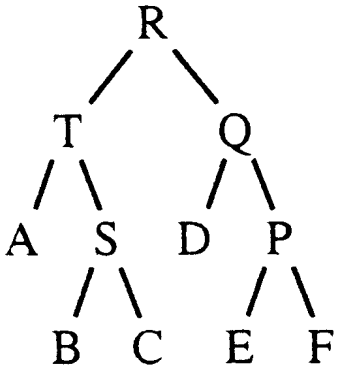
Tree Height 4



(c)

Find(R)

Tree Height 3



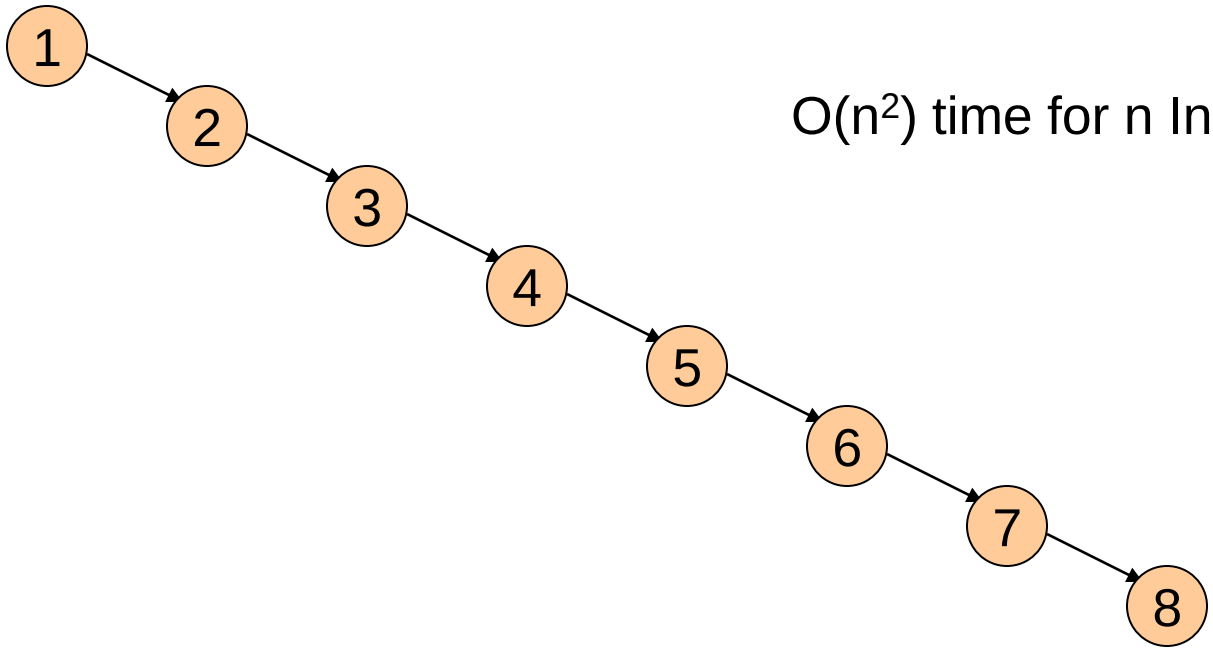
(d)

Splay Tree: Insert and Delete

- Insert x
  - Insert x as normal then splay x to root.
- Delete x
  - Splay x to root and remove it. (note: the node does not have to be a leaf or single child node like in BST delete.) Two trees remain, right subtree and left subtree.
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.
  - Alternatively…

Example: Insert

- Inserting in order 1,2,3,…,8
- Without self-adjustment

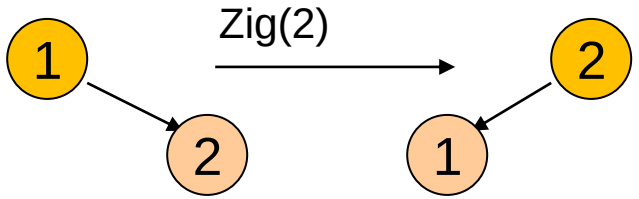


$O(n^2)$ time for n Insert

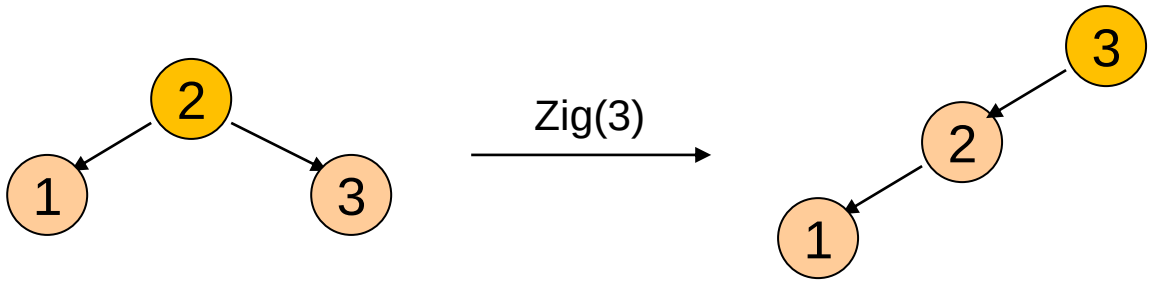
Insert with self-adjustment feature

Insert(1)

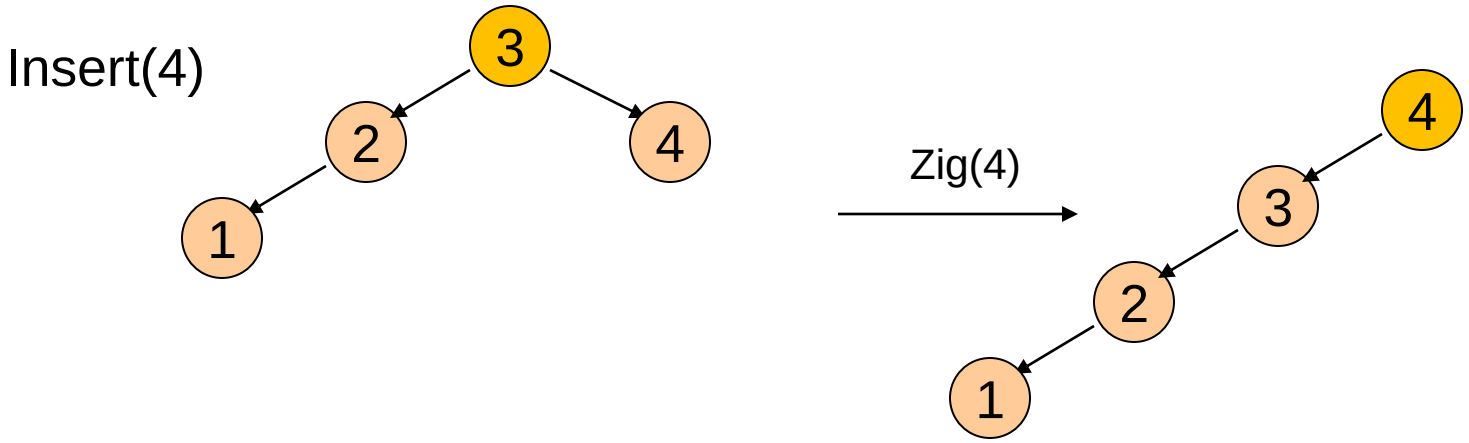
Insert(2)



Insert(3)



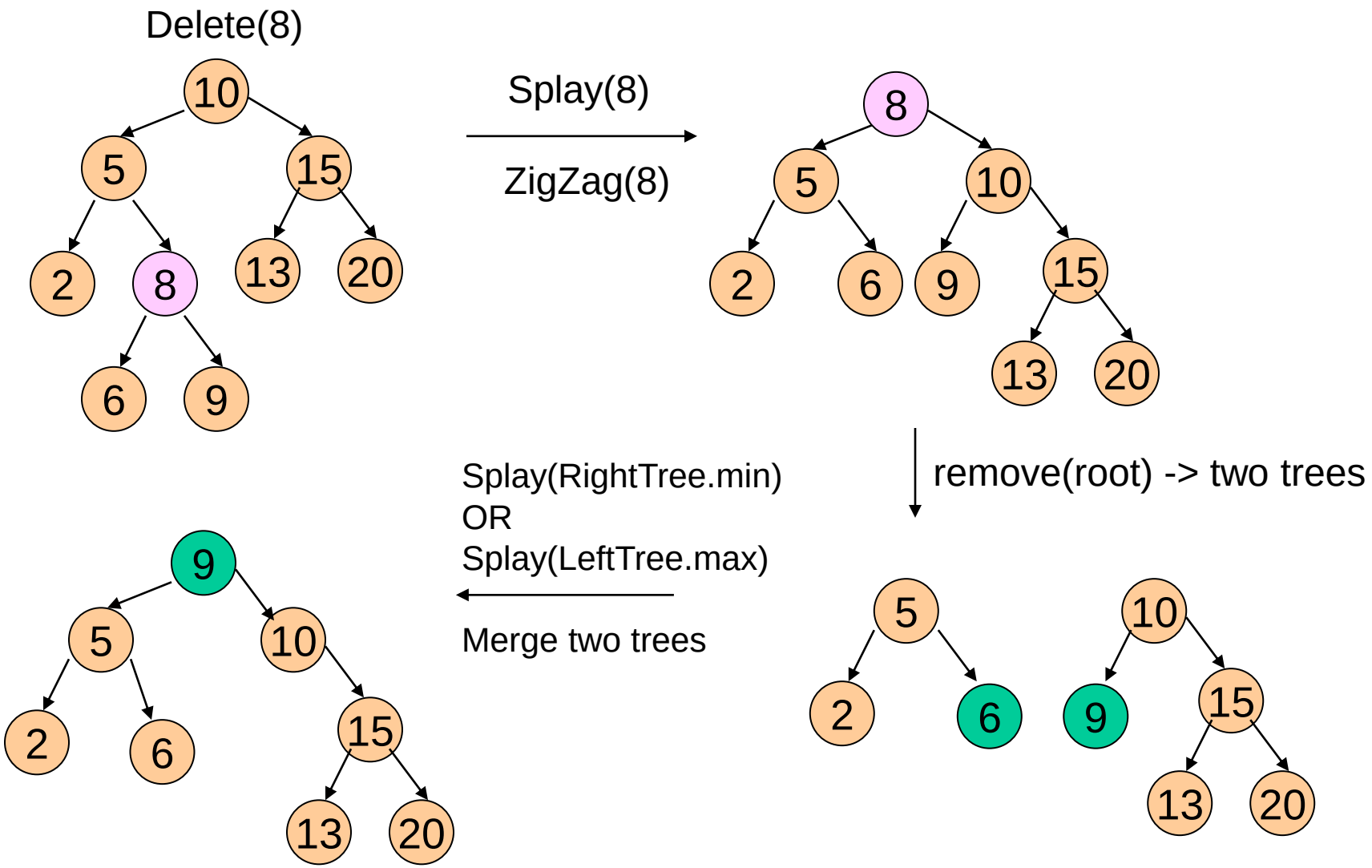
Insert with self-adjustment feature



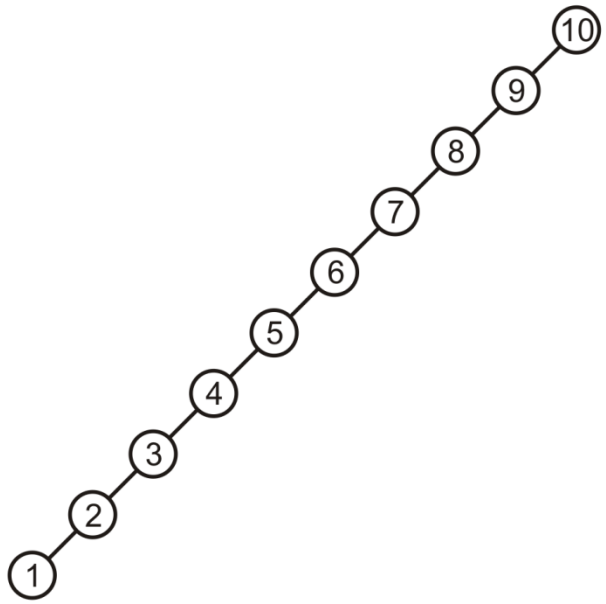
Each Insert takes O(1) time therefore O(n) time for n Insert!!

What could be the insert runtime, if the tree was not a splay tree?

Delete with self-adjustment feature

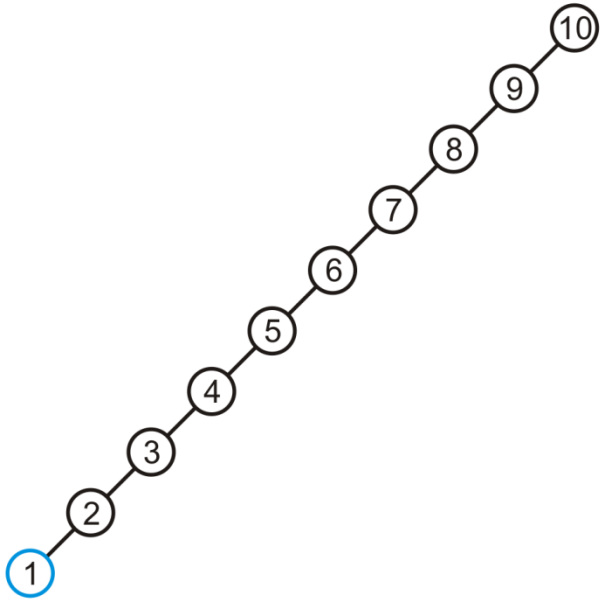


Let’s start with an ordinary tree. It becomes obvious that inserting 1 through 10, without any splaying, in that order, will produce the degenerate tree

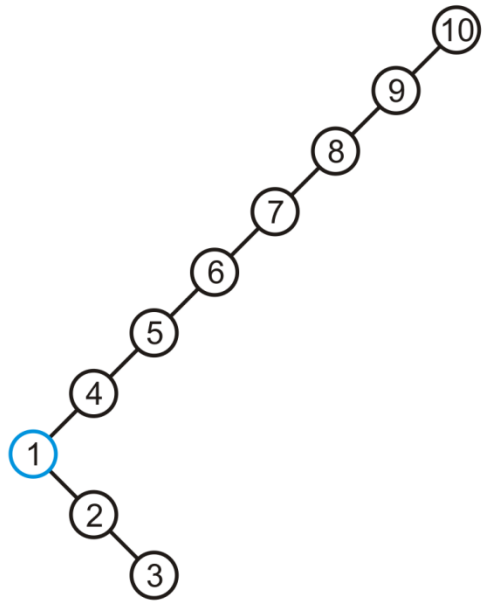


We will repeatedly access the deepest node in the tree

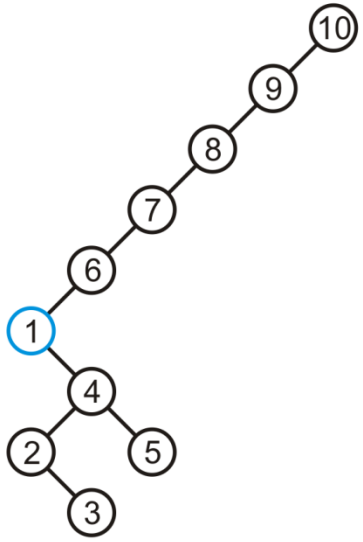
- With each operation, this node will be splayed to the root
- We begin with a zig-zig rotation



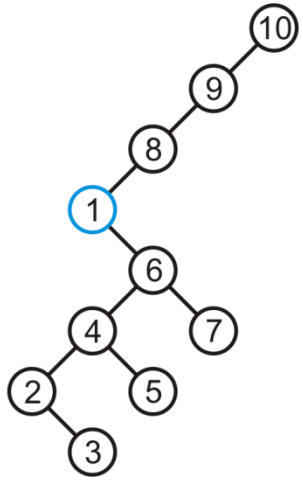
This is followed by another zig-zig operation...



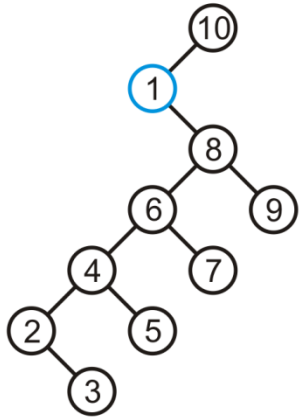
...and another



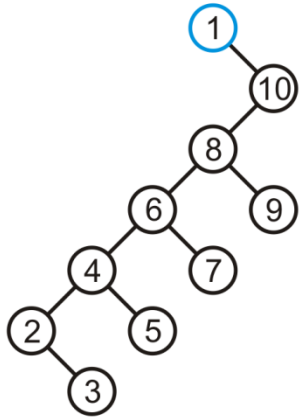
...and another



At this point, this requires a single zig operation to bring 1 to the root

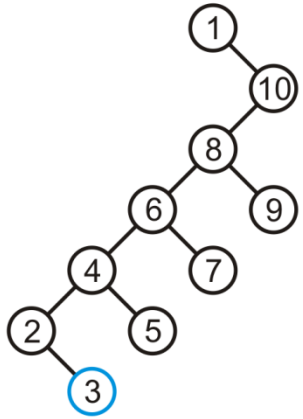


The height of this tree is now 6 and no longer 9



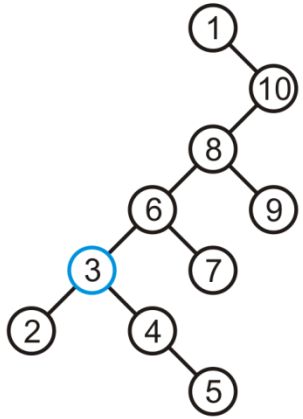
The deepest node is now 3:

– This node must be splayed to the root beginning with a zig-zag operation

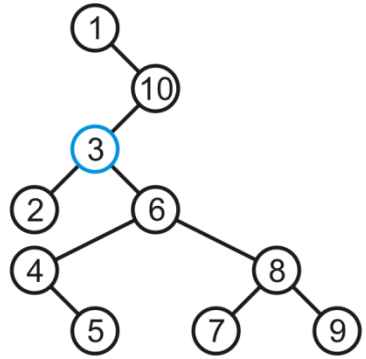


The node 3 is rotated up

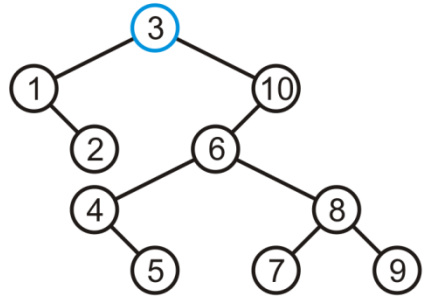
– Next we require a zig-zig operation



Finally, to bring 3 to the root, we need a zig-zag operation

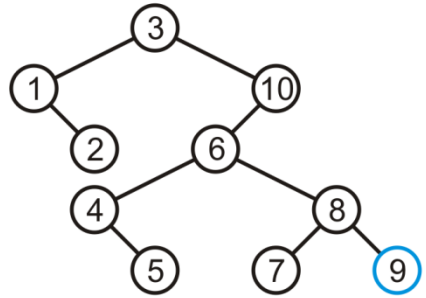


The height of this tree is only 4



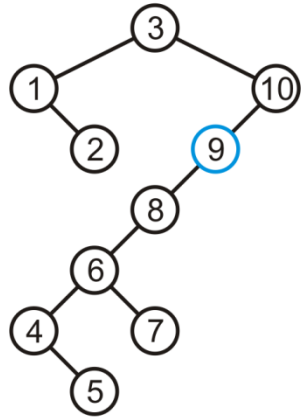
Of the three deepest nodes, 9 requires a zig-zig operation, so will access it next

– The zig-zig operation will push 6 and its left sub-tree down

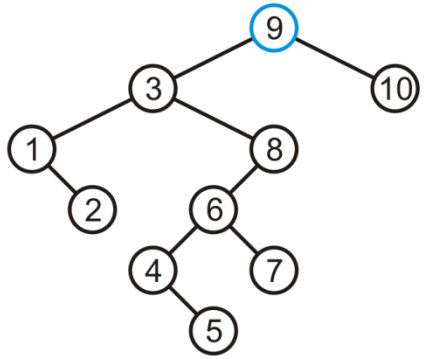


This is closer to a linked list; however, we're not finished

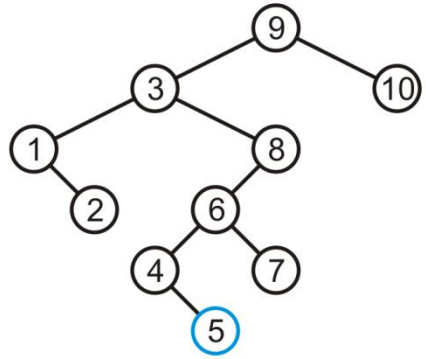
- A zig-zag operation will move 9 to the root



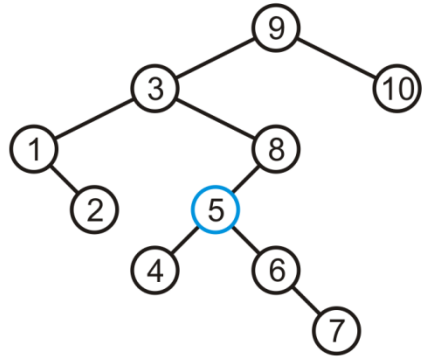
In this case, the height of the tree is now greater: 5



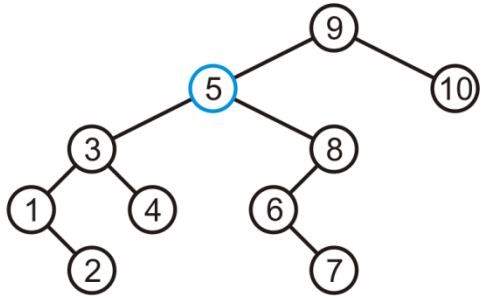
Accessing the deepest node, 5, we must begin with a zig-zag operation



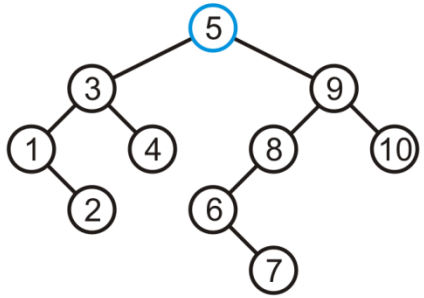
Next, we require a zig-zag operation to move 5 to the location of 3



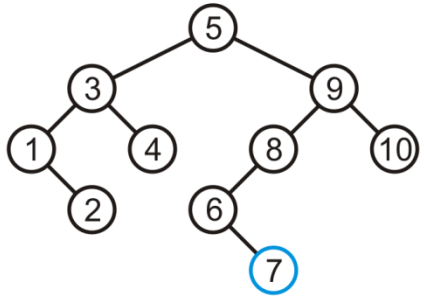
Finally, we require a single zig operation to move 5 to the root



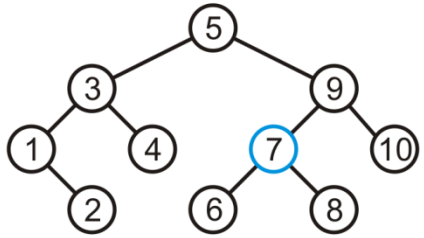
The height of the tree is 4; however, 7 of the nodes form a perfect tree at the root



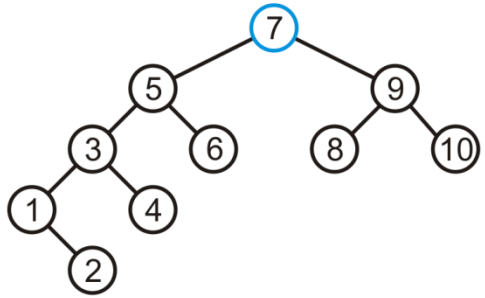
Accessing 7 will require two zig-zag operations



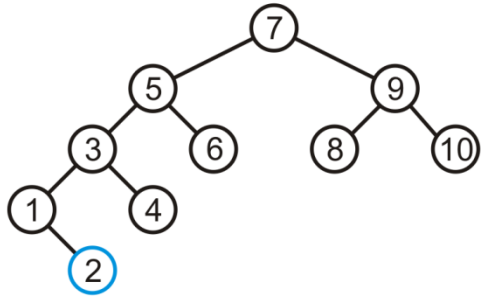
The first zig-zag moves it to depth 2



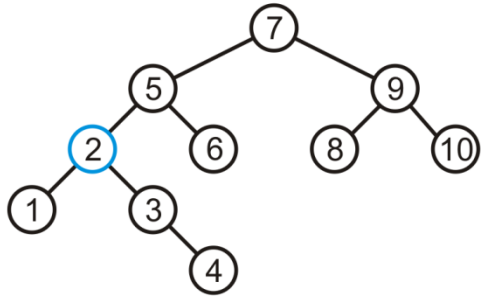
7 is promoted to the root through a zig-zag operation



Finally, accessing 2, we first require a zig-zag operation

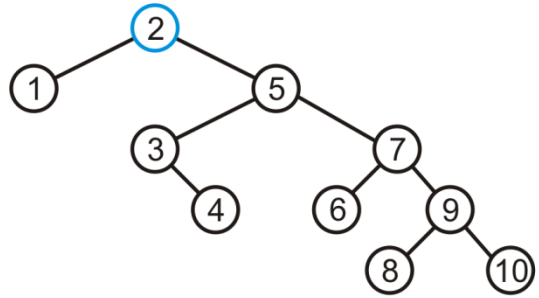


This now requires a zig-zig operation to promote 2 to the root



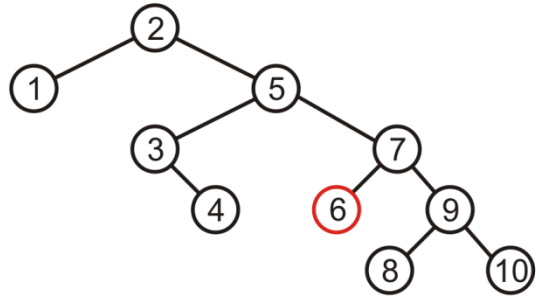
In this case, with 2 at the root, 3-10 must be in the right sub-tree

– The right sub-tree happens to be AVL balanced

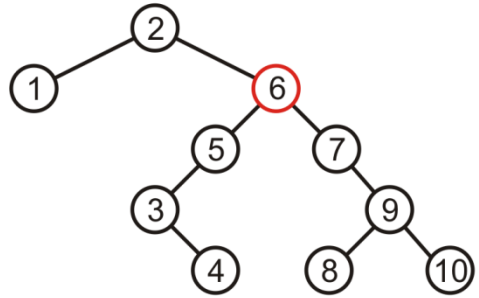


To remove a node, for example, 6, splay it to the root

– First we require a zig-zag operation

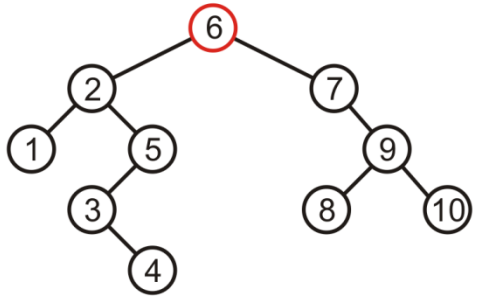


At this point, we need a zig operation to move 6 to the root

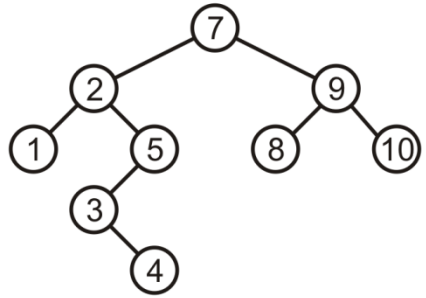


We will now copy the minimum element from the right sub-tree

- In this case, the node with 7 has a single sub-tree, we will simply move it up



Thus, we have removed 6 and the resulting tree is, again, reasonably balanced



Analysis of Splay trees

- Splay trees tend to be balanced
  - M operations takes time O(M log N) for M $\geq$ N operations on N items. (proof is difficult)
  - Amortized O(log n) time.
- Splay trees have good "locality" properties
  - Recently accessed items are near the root of the tree.
  - Items near an accessed one are pulled toward the root.

Comparisons

Advantages:

- The amortized run times are similar to that of AVL trees
- The implementation is easier
- No additional information (height) is required

Disadvantages:

- The tree will change with read-only operations