Data Structures and Algorithms

LECTURE 31

Splay Trees

- Cons of AVL trees:
  - AVL trees require extra storage/complexity for height fields
  - Deletion is not easy to implement
- Solution: Splay trees invented by Daniel Sleator and Robert Tarjan in 1985

Splay Trees

- Balanced BST
  - Insert/search always rotates node to the root
- Amortized time for all operations is O(log n)
- Worst case time is O(n)
- Simpler in terms of implementation of AVL
- Structure of choice in industry

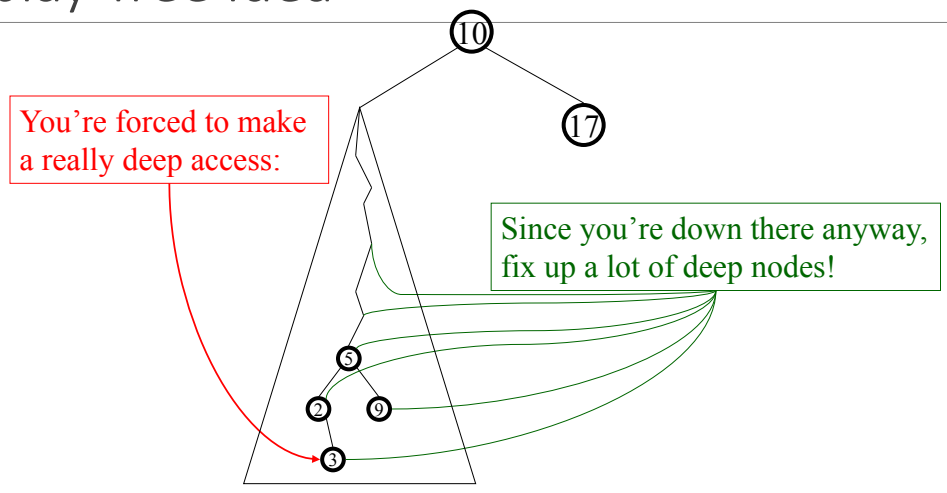
Splay Trees - Running Time

- Can be shown that any k consecutive operations starting from an empty tree take at most O(k log n)
  - All splay tree operations run in amortized O(log n) time
- O(n) operations can occur, but splaying makes them infrequent

Splay Trees

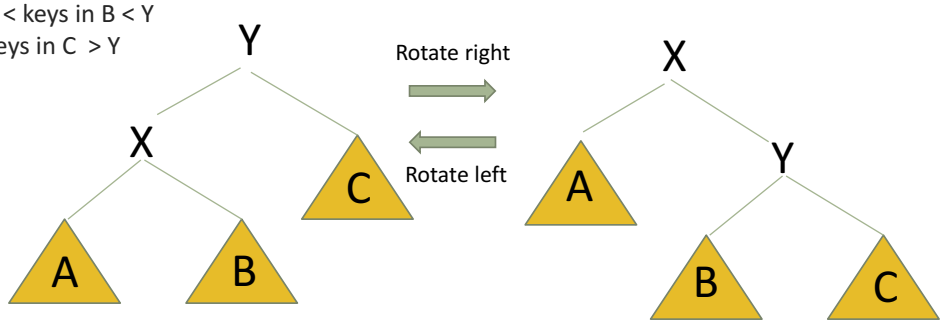
- Splay trees are not kept perfectly balanced
- Keeping a tree perfectly balanced is hard work
- This is why they don't guarantee operations in O(logn)

Splay Tree Idea



Splay Trees - Rotation

- In order to keep splay tree balanced rotation is necessary
  - ✓ keys in A < X
  - ✓ X < keys in B < Y
  - ✓ keys in C > Y

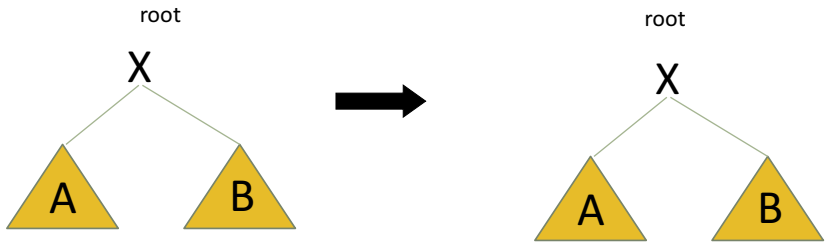


Splaying Cases

- Node being accessed (e.g. X) is:
  - Root
  - Child of the root (either left or right child)
  - Has both a parent (P) and a grandparent (G)
    - ✓ X is the left child of a right child (or X is the right child of a left child)
    - ✓ X is the left child of a left child (or X is the right child of a right child)

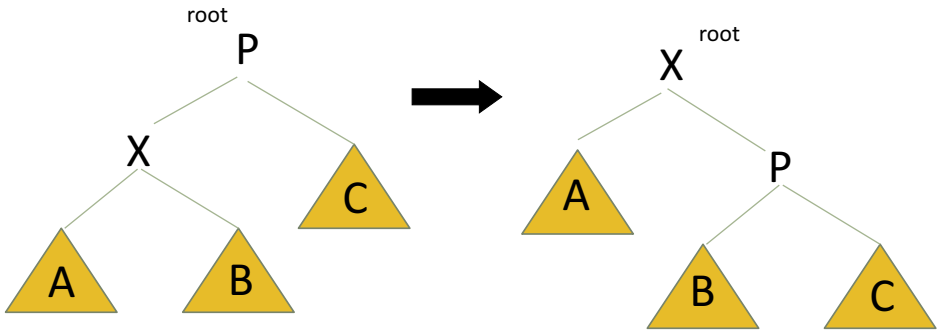
Case 1: root

- Do nothing!



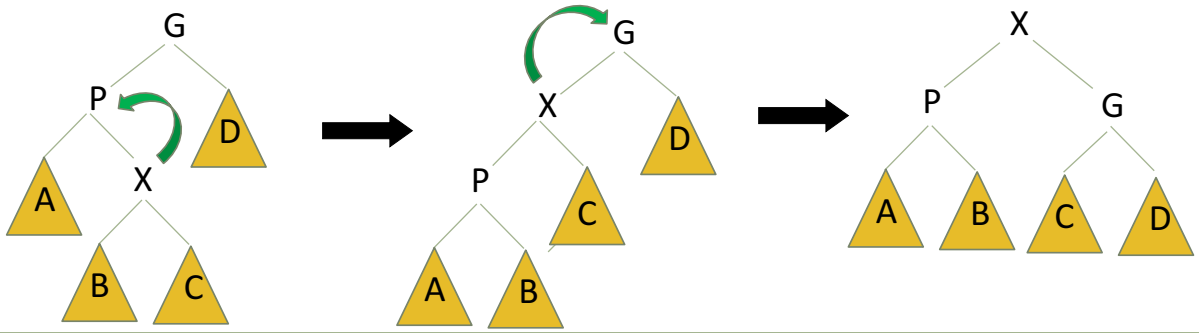
Case 2: child of the root "Zig"

- In the case X is just one step away from the root, we single rotate
  - NOTE: Zig = AVL single rotation



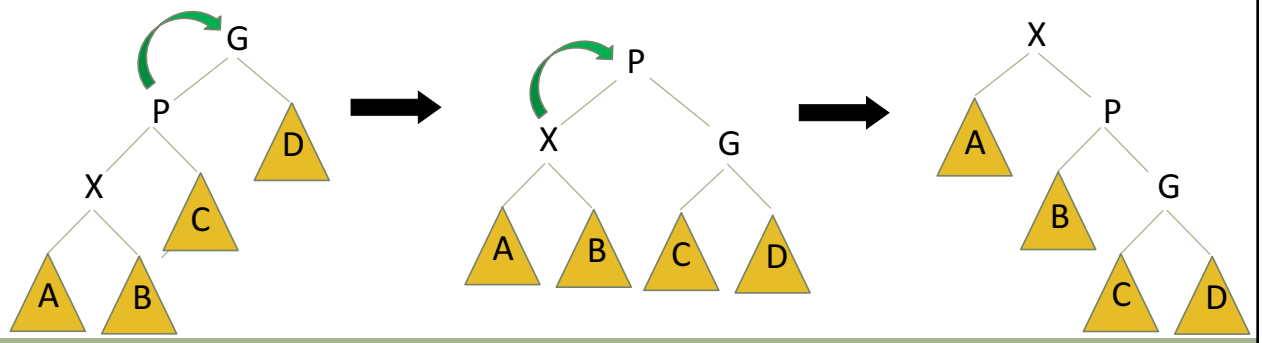
Case 3: X has P and G "Zig-Zag"

- X is the left child of a right child (or X is the right child of a left child)
  - In this case X gets rotate twice
  - Zig-Zag = AVL double rotation

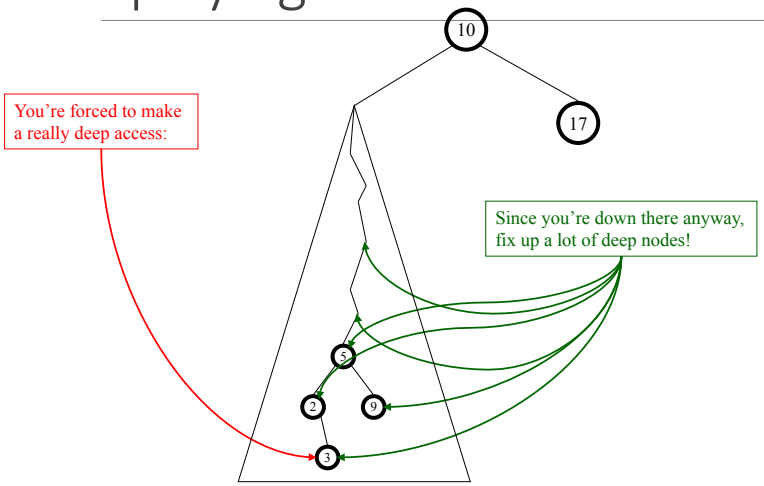


Case 3: X has P and G "Zig-Zig"

- X is the left child of a left child (or X is the right child of a right child)
  - Before starting to rotate X, go up one level (to the parent of X) and first rotate the parent then X

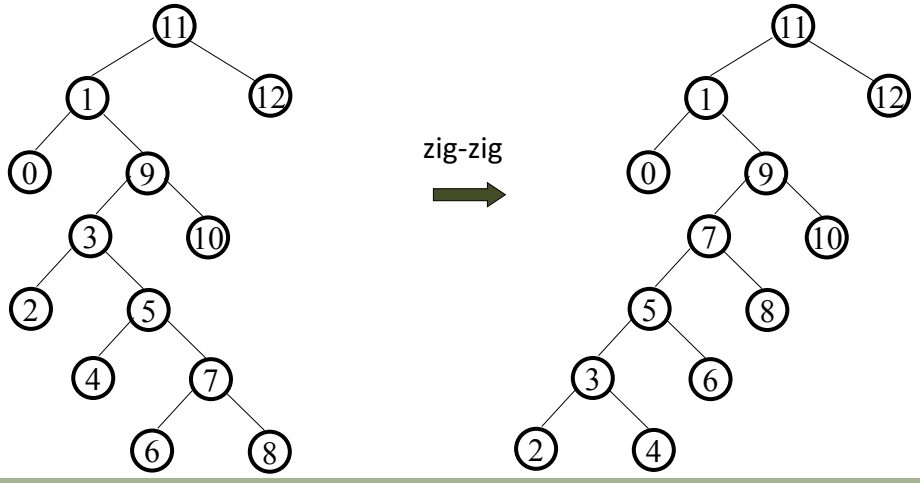


Splaying Idea

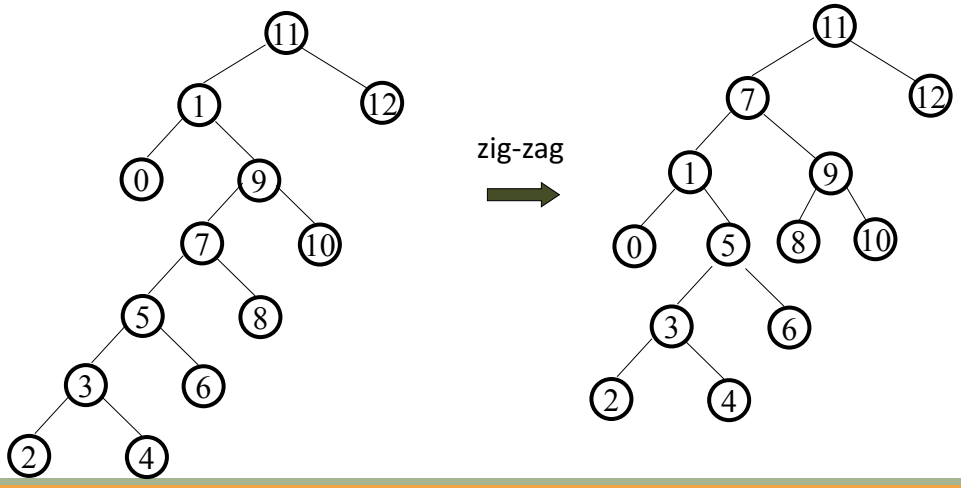


- Repeat "zig", "zig-zag", or "zig-zig" until you get to the root or you get to the child of the root

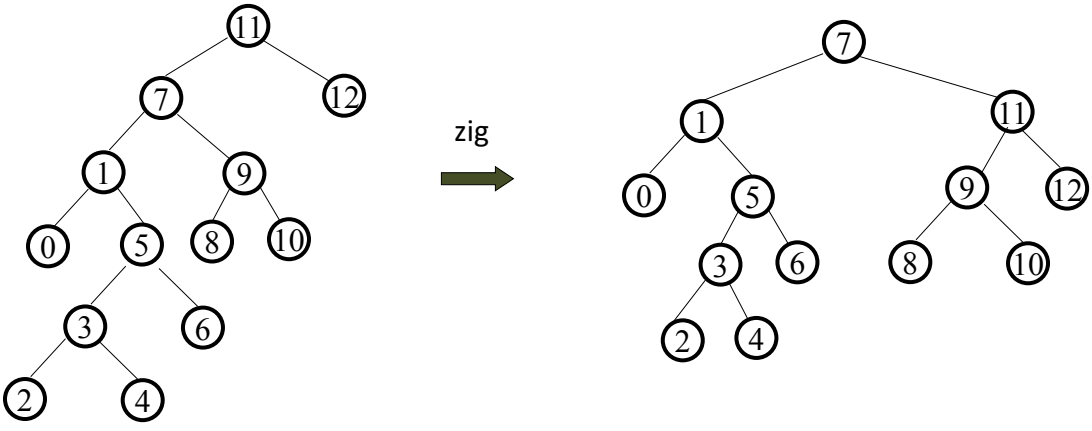
Splaying Example – search 7 and splaying 7



Splaying Example – search 7 and splaying 7



Splaying Example – search 7 and splaying 7



Why Splaying Helps

- If a node n on the access path is at depth d before the splay, it's at about depth $d/2$ after the splay
  - Exceptions are the root, the child of the root, and the node splayed
- Overall, nodes which are below nodes on the access path tend to move closer to the root
- Splaying gets amortized O(log n) performance
  - Maybe not now, but soon, and for the rest of the operations

Splay Trees - Operations

- **Search** for node with key *k*
  - Like BST search
  - Let X be the node where the search ends (succeed or not)
  - Take node X and splay up to the root of the tree using a sequence of rotations
- Why X becomes the new root?
  - If we access the key *k* over and over again, by moving X up it insurance fast search for *k* later

Splay Trees - Operations

- **Insert X**
  - Like BST insert
  - Splay X to the root

Splay Trees - Operations

- **Delete X**
- There are two ways to perform this operation:
  - Delete X method 1
  - Delete X method 2

Splay Trees - Operations

- **Delete X method 1 (**NOTE**:** The node splayed is the parent of the deleted node)
  - Like BST delete
  - If X has no left child, delete X and make the right child of X the child of the parent of X, then splay the parent X (if X has no right child, do the reverse)
  - If X has two children, swap the data of X with its successor, and then delete the successor node and splay its parent node

Splay Trees - Operations

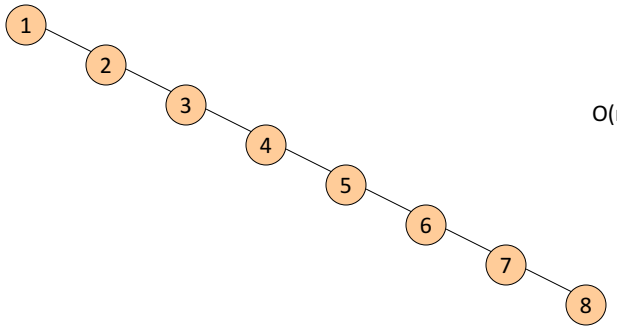
- **Delete X method 2**
  - Splay X to root and remove it (note: the node does not have to be a leaf or single child node like in BST delete)
  - Two trees remain, right subtree and left subtree
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree

Splay Trees - Operations

- **Find max and min**
  - Like BST
  - Once find max or min splay it to the root

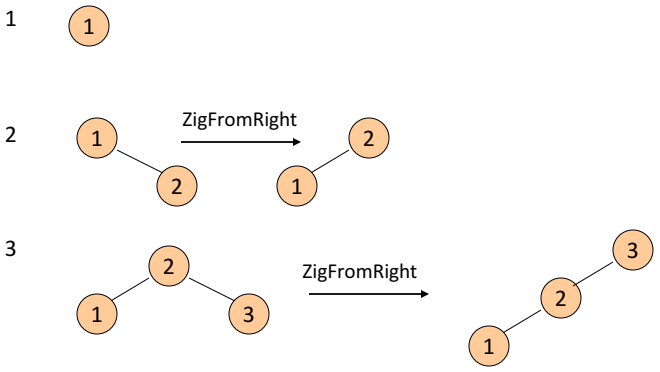
Example Insert

- Inserting in order 1,2,3,…,8 (without splaying)

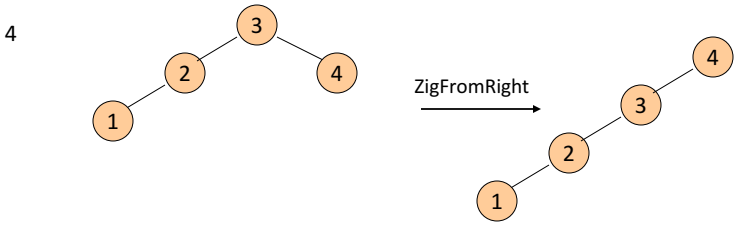


$O(n^2)$ time for n Insert

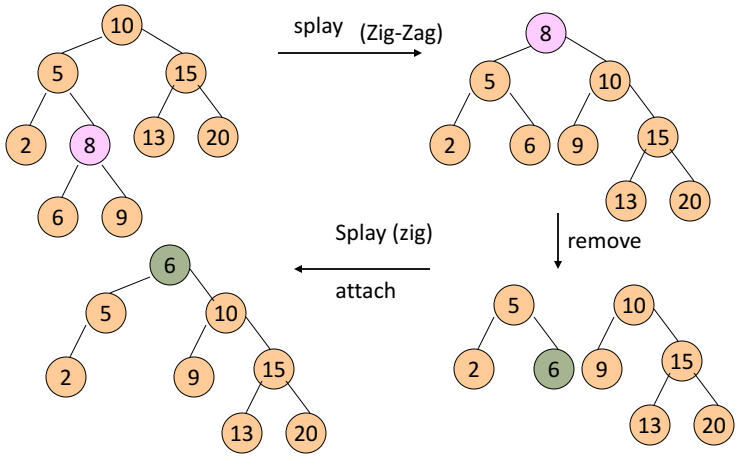
Example Insert with splaying



Example Insert with splaying



Example Deletion



Analysis of Splay Trees

- Splay trees tend to be balanced
  - 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