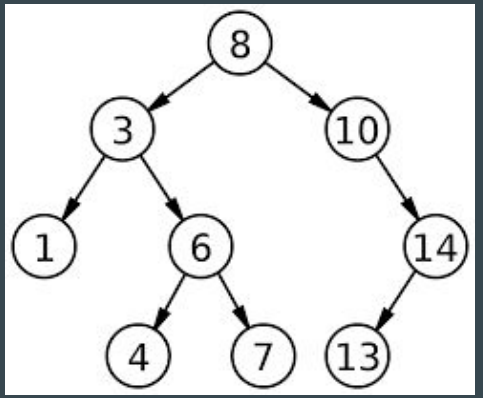


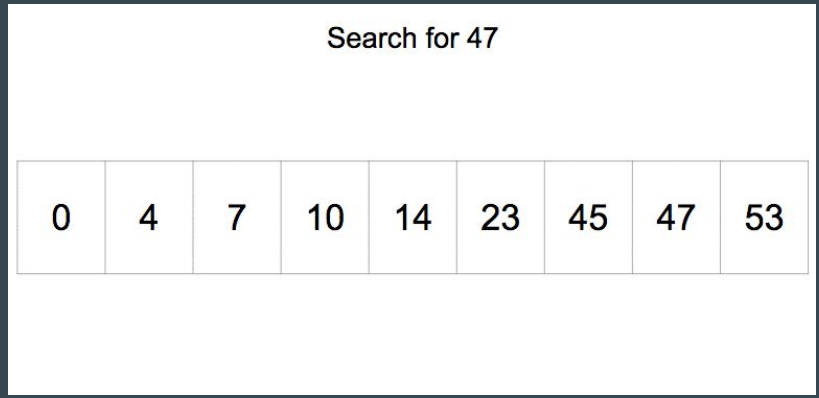
# Splay Trees

Dean Carey & Jeffrey Huang

# Binary Search Trees

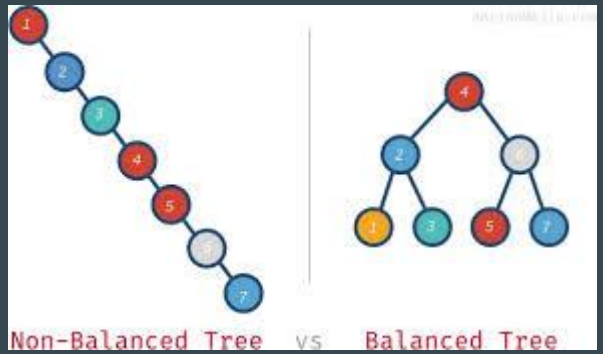
- Memory structure which organizes data in a tree-like structure
- Consists of "root" and "leaf" data points, or nodes, that branch out in two directions (left and right)
- Capable of three basic functions: search, insert, & delete
- These functions are carried out by the Binary Search function
    - Binary search works by halving the number of elements to look through and hones in on the desired value



# Balanced vs. Unbalanced Trees

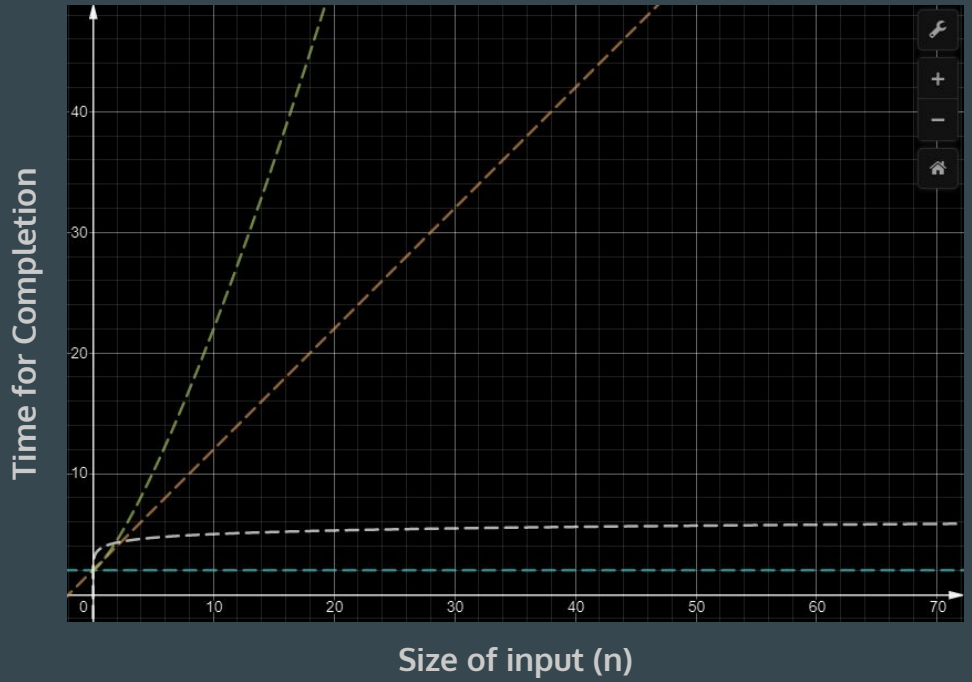
- Balanced Trees - has the smallest possible height, and is the most efficient configuration of the tree.
  - Much faster to reach leaves from root of tree
  - Effective for static databases (e.g. program dictionaries)
    - AVL and Red-Black Trees self-balance their structures for more efficient searches
- Unbalanced Trees - Generally inefficient, no set structure determinants
  - Unorganized; more steps on average to reach leaves from root of tree
    - Linked Lists (O(n) time, worst case)
    - Splay Trees (Special case)



# Time Complexity and Big O Notation

- Linear Time O(n)
- Constant Time O(1)
- Quadratic Time $O(n^2)$

- Gives insight as to how a function scales in time as input grows larger
- All splay tree operations run in O(log n) amortized (average) time with a worst case of O(n) time




Orange - Linear

Blue - Constant

White - Logarithmic

Green - Quadratic

# Splay Trees

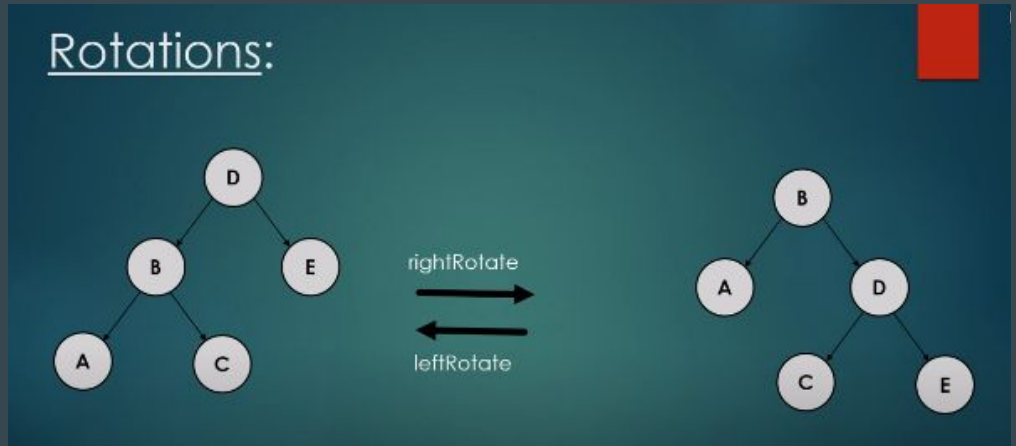
- Variation of binary search tree that rotates the tree based on recent access to an element/node
- Whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root node of the tree (act of **splaying**)
- In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus searched for quicker (**locality**)

**The goal of the splay tree is not to make each individual operation fast, like a balanced tree does, but to make the sequence of operations faster when dealing with skewed data.**

# Splaying & Tree Rotations

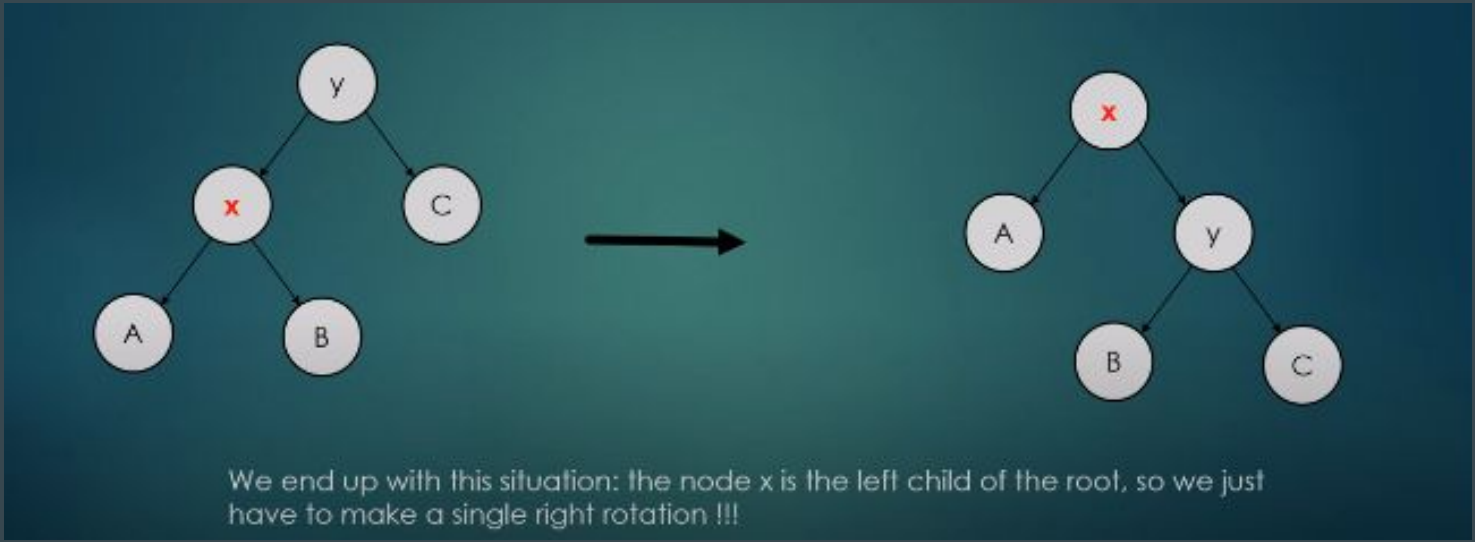
When splaying, the tree looks at three possibilities:

1. The node's parent is the root
2. The node is the left child of a right child (or the right child of a left child)
3. The node is the left child of a left child (or the right child of a right child)



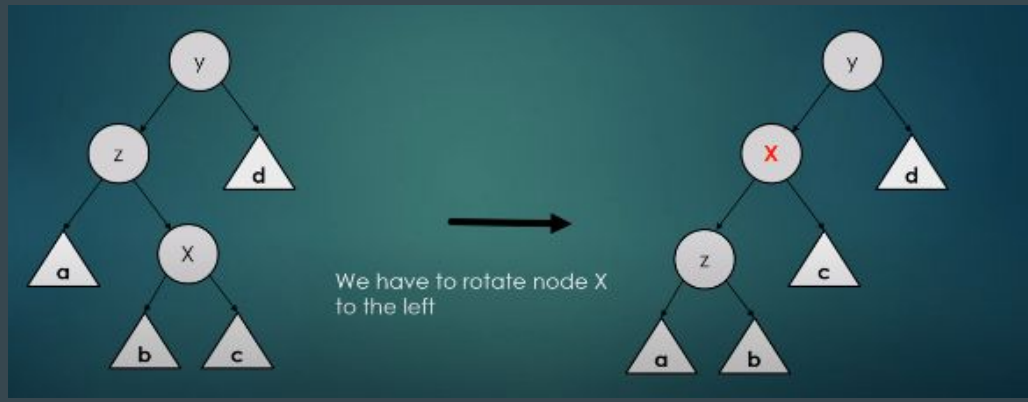
# 1. Zig Rotation

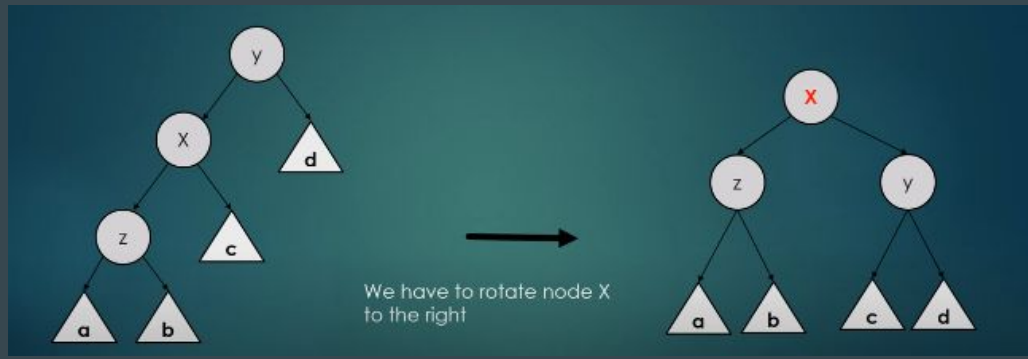
- Splaying node whose parent is the root
- Singular rotation needed to bring splayed node to root
    - Left rotation for a right child, right rotation for a left child



# 2. Zig-Zag Rotation

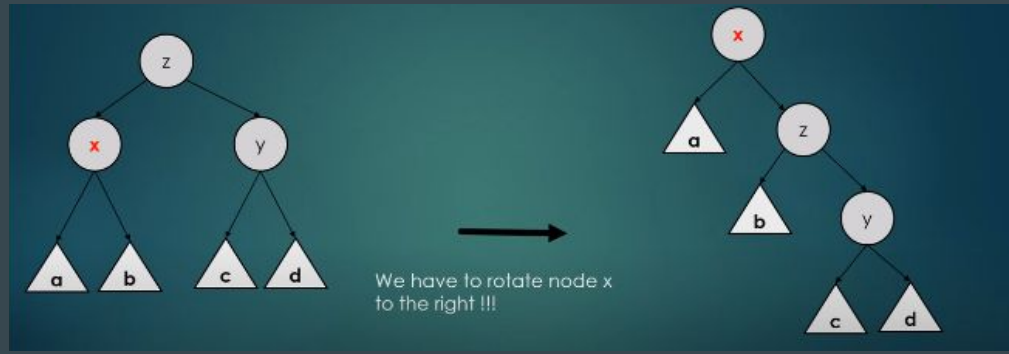
- Splaying a node that is a right child of a left child (or a left child of a right child)
- Left rotation, followed by right rotation to bring splayed node to root
    - (Or right rotation followed by left rotation for left child of a right child)



# 3. Zig-Zig Rotation

- Splaying a node that is a left child of a left child (or right child of a right child)
- Two consecutive right (or left) rotations to bring splayed node to root

# Fundamental Splay Tree Operations

- Search: Binary search down the tree to locate the node. Then, perform splay on that node to bring it to the top of the tree.
- Insert:  Find appropriate location at the bottom of the tree using binary search. Then perform splay on that node.
- Delete: Since deleting a node isn't as obvious as to which node to splay, the process is left up to the implementer. One typical method is to first splay the node to be deleted, making it the root node, then to delete it. We are left with two separate trees that are then joined together using the join operation.

# Join / Split Operations

- Join: Two binary trees must exist, and the elements in tree S < tree T
  1. Splay the largest element in S
  2. Set the right child of the root of S to be the root of T

- Split: Results in two separate trees, one with all elements greater than the given node, one with all elements less than **or equal to** that node.
  1. Splay the given node to the root of the tree
  2. Take the right child of the root and make it its own tree

# Live Demo

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

# Importance and Use

- The most popular data structure in the industry
- Fast access to elements accessed recently
- Splay trees are used when you expect **locality** from your searches. The splay operation moves the located element to the root, but it also keeps the previous root near the top.
- MRU Applications (Most recently used)
- Best for information that is modified frequently (non-static)
- **Cache-Algorithms**
  - Cached data is information from a website or app that is stored on your device to make the browsing process faster. Cached data saves on loading time, though it occupies space
- Network Router
  - A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times.

# Resources

https://www.cs.cornell.edu/courses/cs3110/2011sp/Recitations/rec25-splay/splay.htm

https://www.youtube.com/watch?v=IBY4NtxmGg8&ab_channel=GlobalSoftwareSupport

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

https://brilliant.org/wiki/splay-tree/

https://www.geeksforgeeks.org/splay-tree-set-1-insert/

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

# Thanks for listening! Any Questions?