5 - Self Balancing Trees

Joseph Afework
CS 241

Dept. of Computer Science
California Polytechnic State University, Pomona, CA

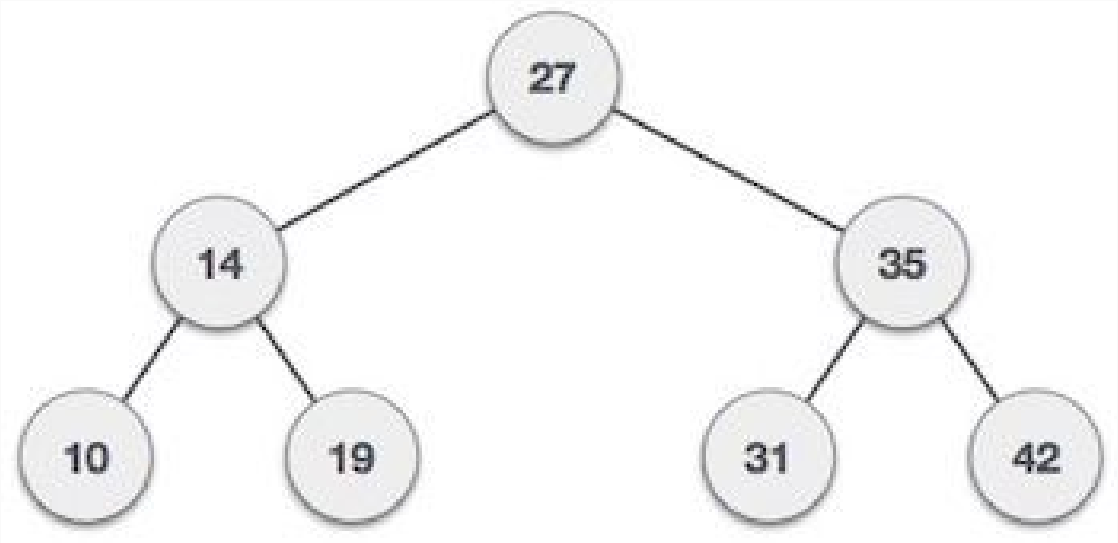
Agenda

- Tree Recap
- Observations
- Self Balancing Trees
- Examples

Reading Assignment

- Read Chapter 27 - Balanced Search Trees
  - Chapter 27 (Read about: AVL, Red-Black Trees, B-Trees)

Trees



Tree Operations

Binary Tree	Average Case	Worst Case
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Search	O(log(n))	O(n)

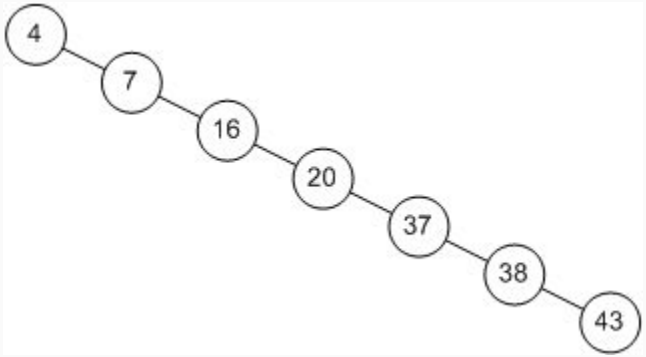
Tree Operations Contd.

Binary Tree	Average Case	Worst Case
Insert	O(log(n))	**O(n)**
Delete	O(log(n))	**O(n)**
Search	O(log(n))	**O(n)**

Remember

Worst Case for a Tree:

- resembles a linked lists…. But still a valid binary tree



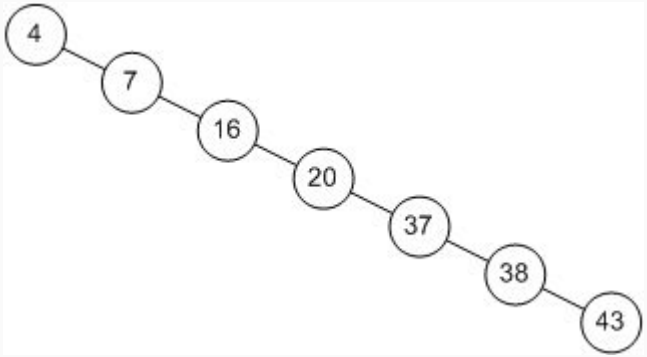
Observations Contd.

1) Is there a way we can reorder the data to be more performant?
    - Does our data lose meaning after reordering?
    - Still want to have a tree structure….
    - We don’t want to use a different ADT

2) Can we create a rule to follow when we need to re-order data?

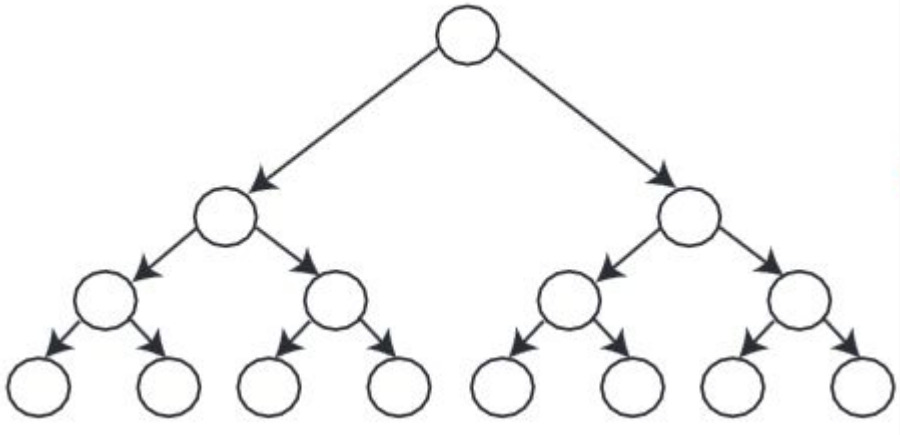
ICE 5.1 Reorder the tree

Instructions: Reorder the data in the tree to yield a better search runtime...



Note

- n = the height of a tree
- The runtime is between $\log_2(n)$ and n



$\log_2(n)$ vs $\log(n)$

- **Derivation/Absolute Runtime has detail:** $\log_2(n)$
- **Big O notation doesn't care about constant terms:** $\log(n)$
  - $\log_2$ has a base of 2
  - Log has a base of 10
  - **Remember change of base:**

$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$$

$\ln(n) = \log_2(n)$

$\ln(n) = \log_{10}(n) / \log_{10}(2) =$ **$\log(n) \times 1/C$**

C is a constant.. O(n) drops constants… = log(n)

Observations

- Minimize the height of the tree
  - Reduces the worst case to average case
- Goal is to preserve the same data
- Goal is to preserve the **relative hierarchy (Binary Search Tree Properties)**

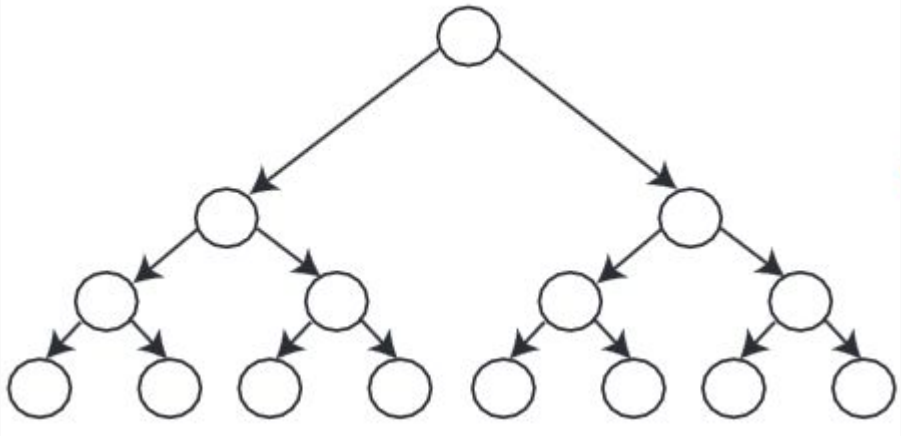
- Move from: **O(n) -> O (log(n))**

Self Balancing Trees

- **Self Balancing Tree:** Type of trees that try to reduce the height of the tree for performance gains
  - The smaller the height, the less exaggerated the runtime (n) vs log(n).

- The tree is rebalanced at specific times/operations to reduce the runtime overhead of rebalancing
- Rebalancing does add an overhead to execution, but it’s worth it.

Balanced Tree



SBT Contd.

- **Max height <= $log_2 n$**

- Balanced BSTs are not always so precisely balanced...
  - expensive to keep a tree at minimum height at all times...
  - most algorithms keep the height within a constant range - **balance factor**

Tree Operations Revisited

Binary Tree	Average Case	Worst Case
Insert	O(log(n))	**O(n)**
Delete	O(log(n))	**O(n)**
Search	O(log(n))	**O(n)**

Balanced Tree Operations

Binary Tree	Average Case	Worst Case
Insert	O(log(n))	**~ O(log(n))**
Delete	O(log(n))	**~ O(log(n))**
Search	O(log(n))	**~ O(log(n))**

Self Balancing Trees

- AA tree
- **AVL tree**
- **Red-black tree**
- **B-tree**
- Many more….

Applications

1) Improve the runtime performance of a BST…

2) Useful in preserving ordered relationships nodes - implementing a queue/priority queue

Learning Outcomes

- Understand the tree data structure and its related terminologies.

References

http://www.purplemath.com/modules/logrules5.htm

https://en.wikipedia.org/wiki/Binary_search_tree

https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree