

CSE 332 Winter 2024

Lecture 9: AVL Trees and B-Trees

Nathan Brunelle

<http://www.cs.uw.edu/332>




Dictionary (Map) ADT

- Contents:
 - Sets of key+value pairs
 - Keys must be comparable
- Operations:
 - `insert(key, value)`
 - Adds the (key,value) pair into the dictionary
 - If the key already has a value, overwrite the old value
 - Consequence: Keys cannot be repeated
 - `find(key)`
 - Returns the value associated with the given key
 - `delete(key)`
 - Remove the key (and its associated value)

Dictionary Data Structures

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

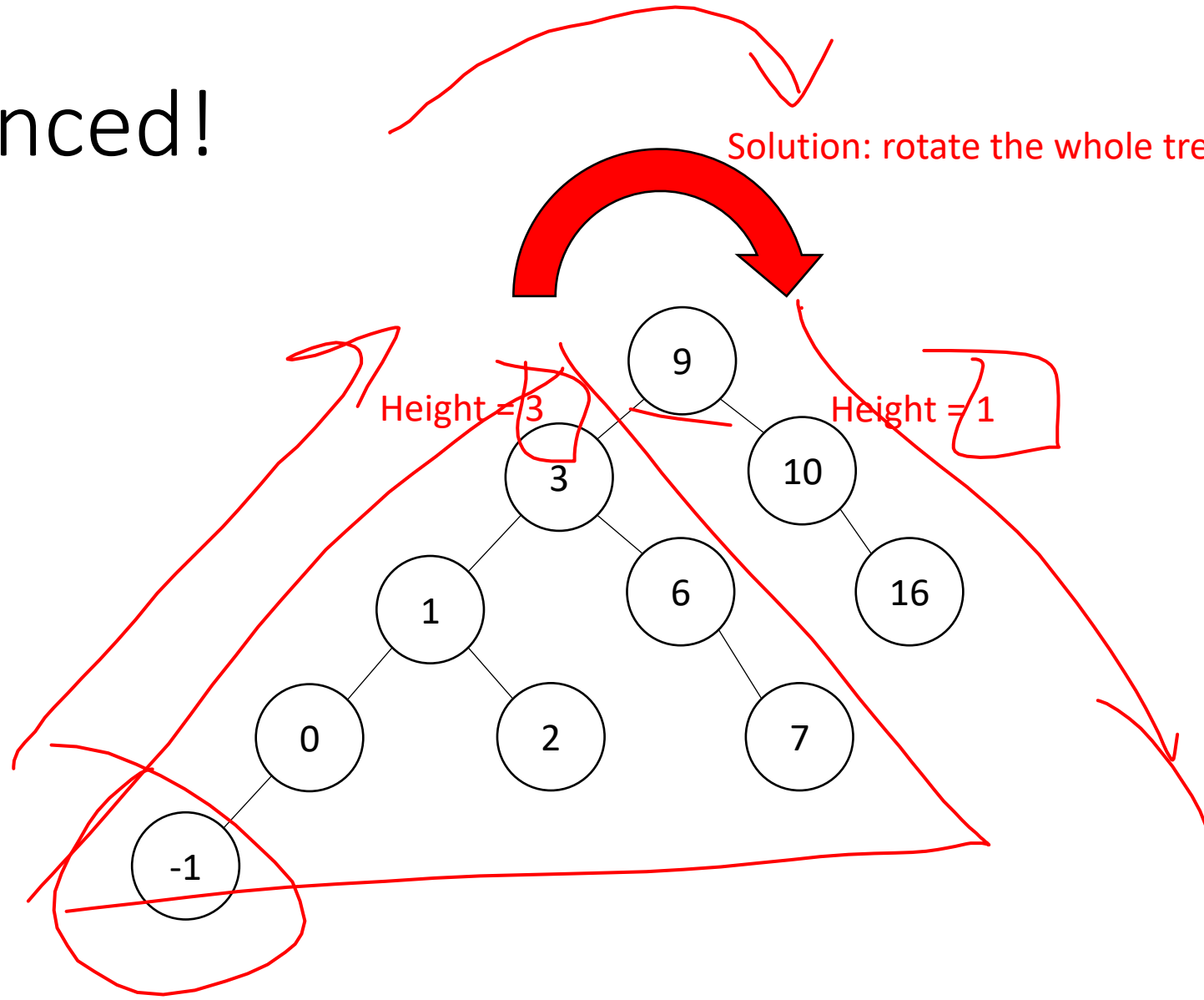
AVL Tree

- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one. 
 - height of left subtree and height of right subtree off by at most 1
 - Not too weak (ensures trees are short)
 - Not too strong (works for any number of nodes)
- Idea of AVL Tree: 
 - When you insert/delete nodes, if tree is "out of balance" then modify the tree
 - Modification = "rotation" 

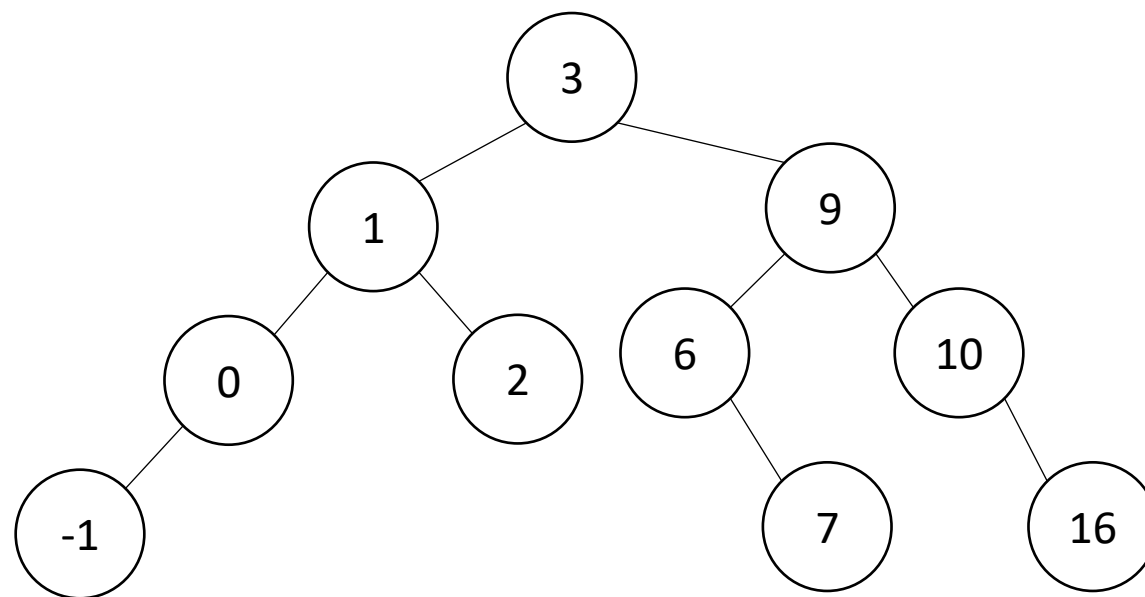
Not Balanced!

Solution: rotate the whole tree to the right

$\log_2 n$

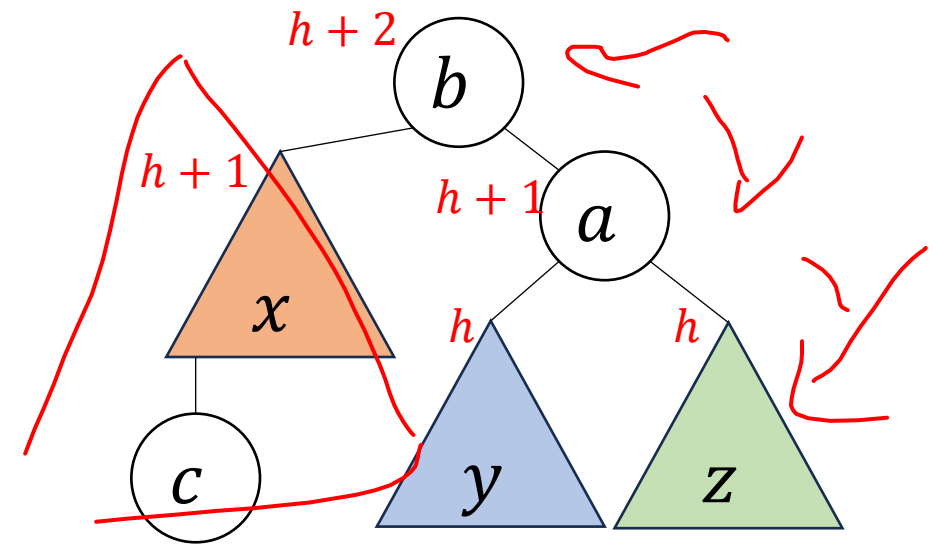
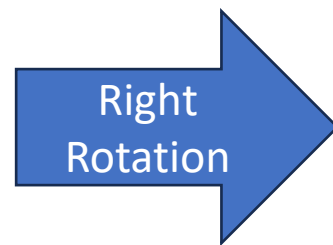
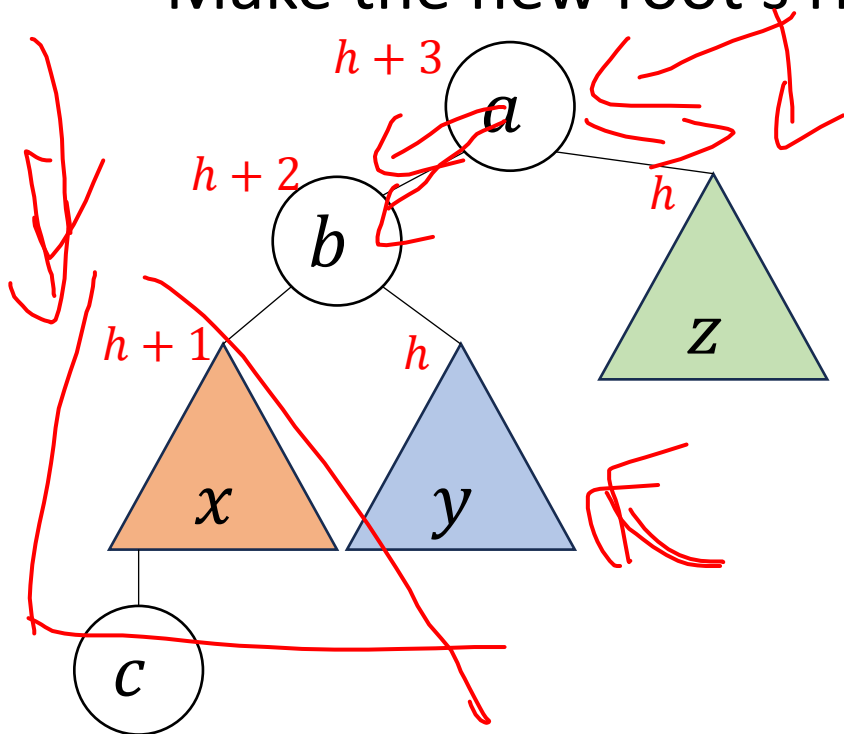


Balanced!

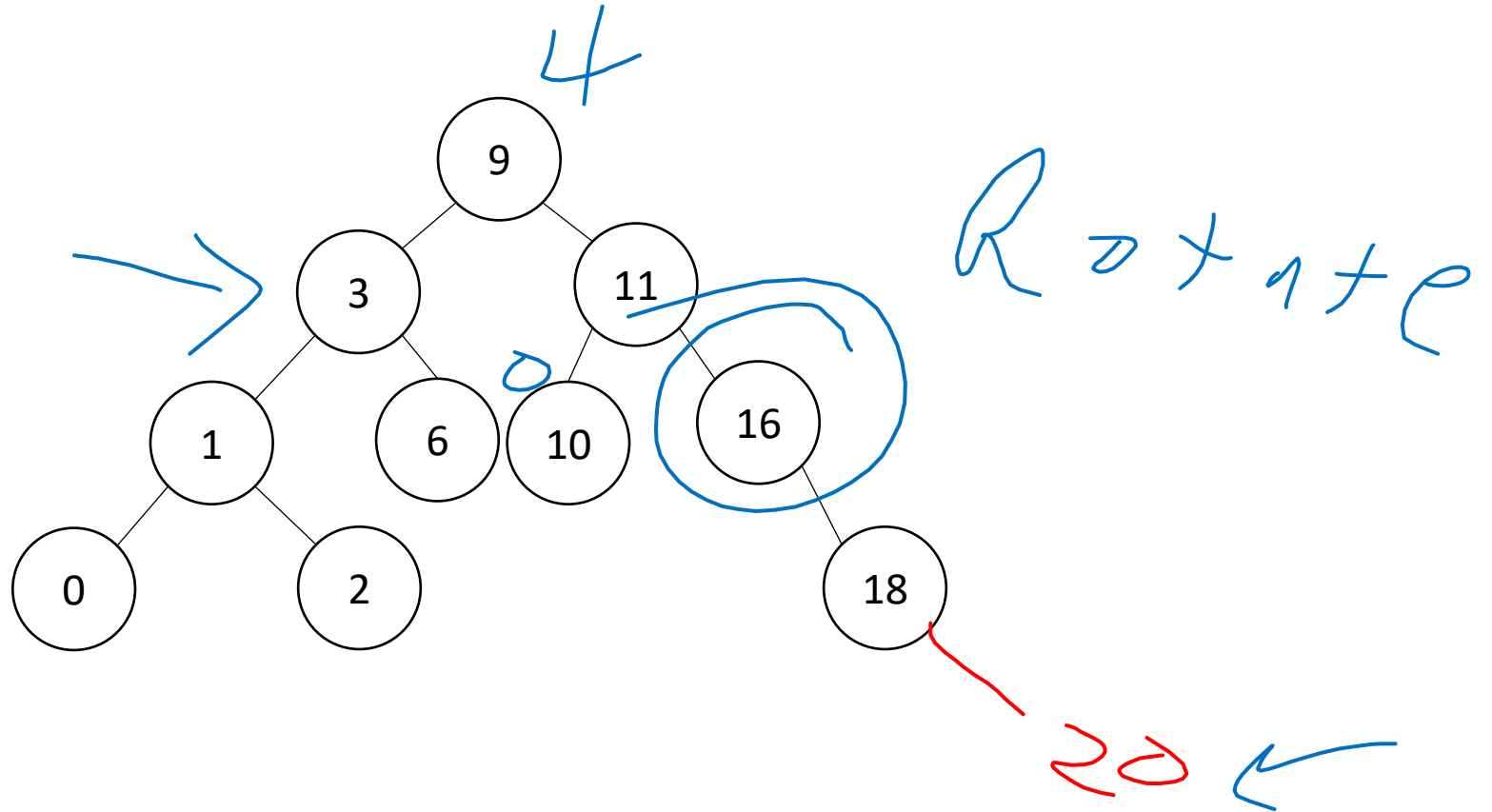
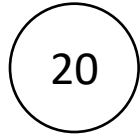


Right Rotation

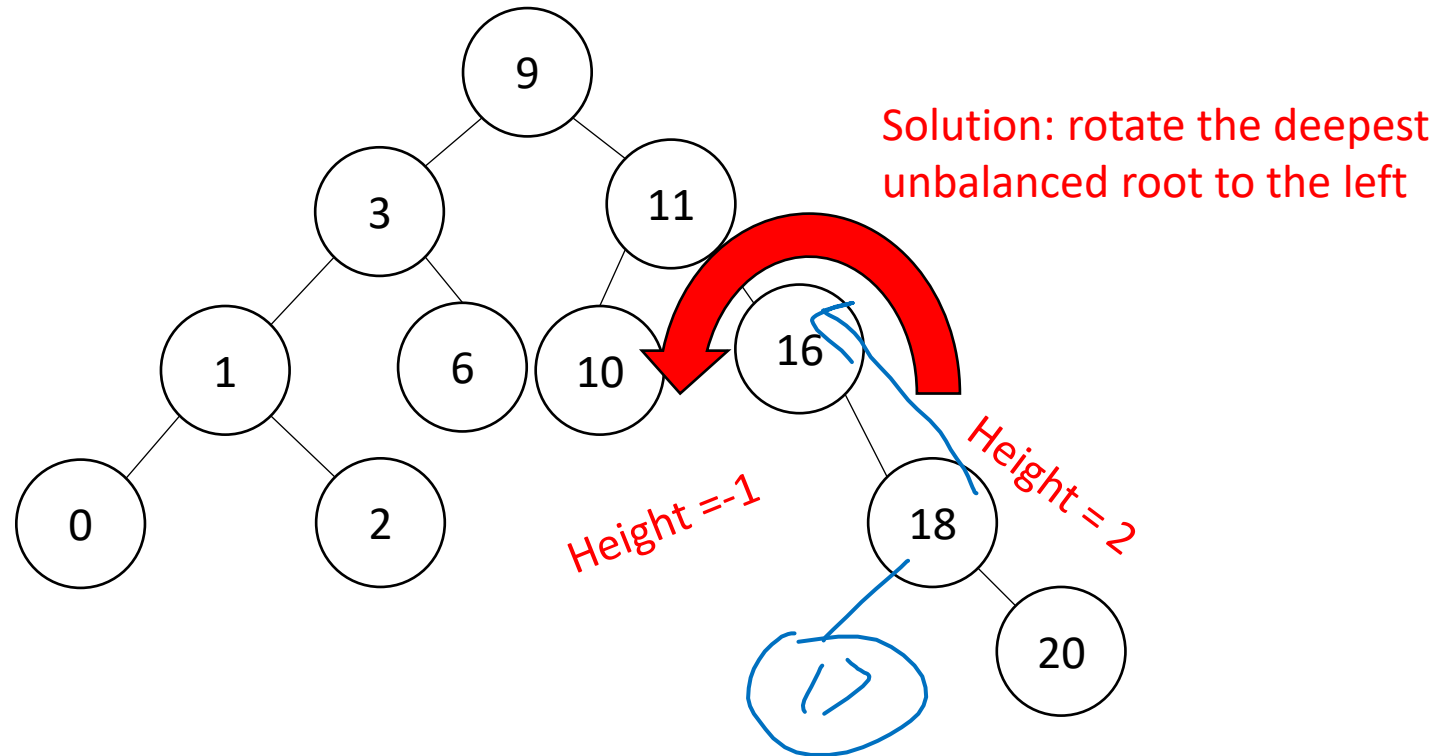
- Make the left child the new root
- Make the old root the right child of the new
- Make the new root's right subtree the old root's left subtree



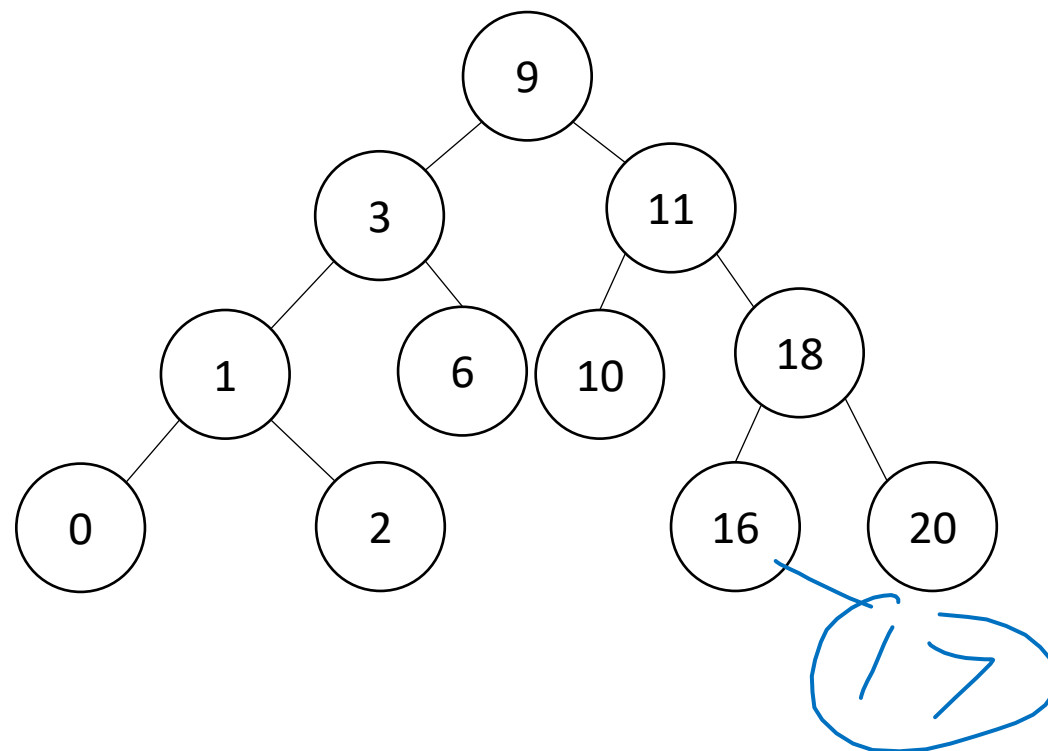
Insert Example



Not Balanced!

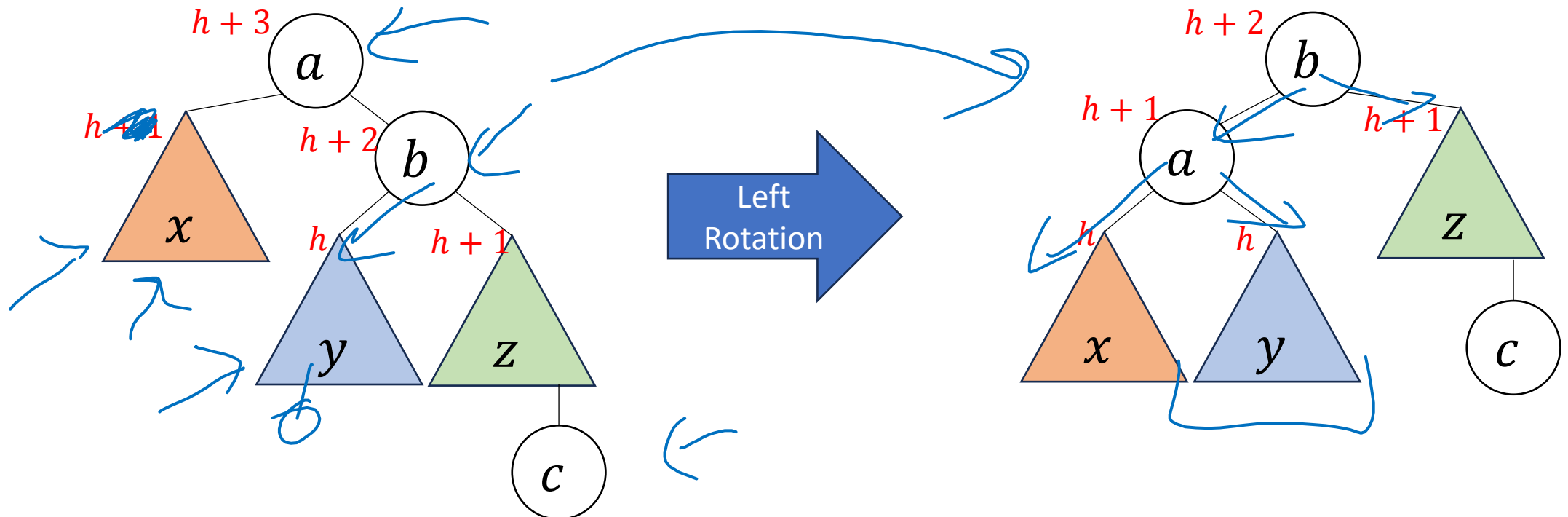


Balanced!



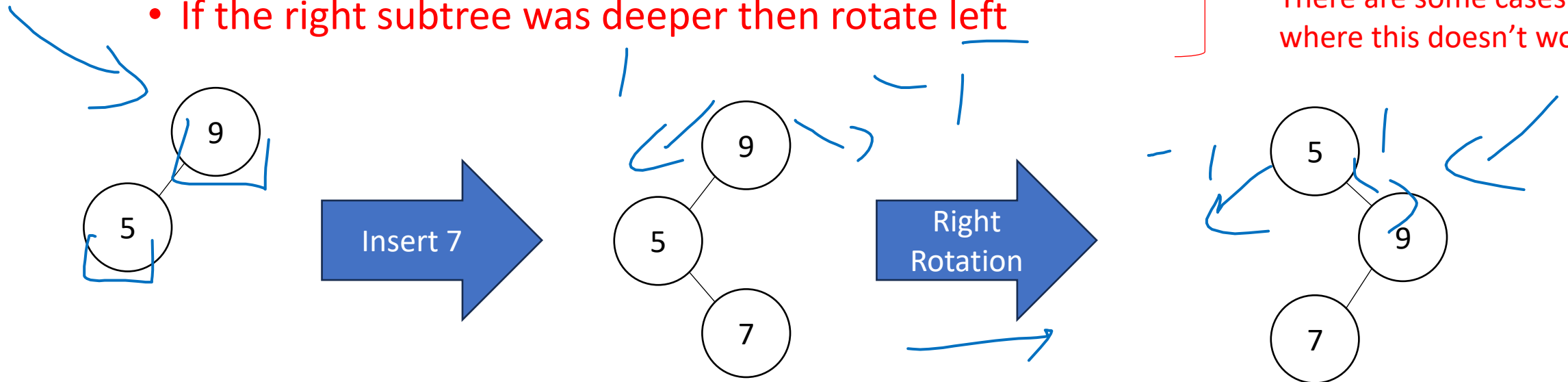
Left Rotation

- Make the right child the new root
- Make the old root the left child of the new
- Make the new root's left subtree the old root's right subtree

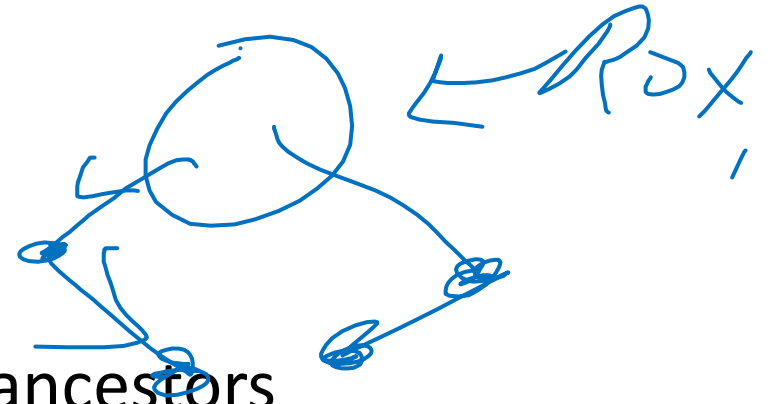


Insertion Story So Far

- After insertion, update the heights of the node's ancestors
- Check for unbalance
- If unbalanced then at the deepest unbalanced root:
 - If the left subtree was deeper then rotate right
 - If the right subtree was deeper then rotate left



Insertion Story So Far

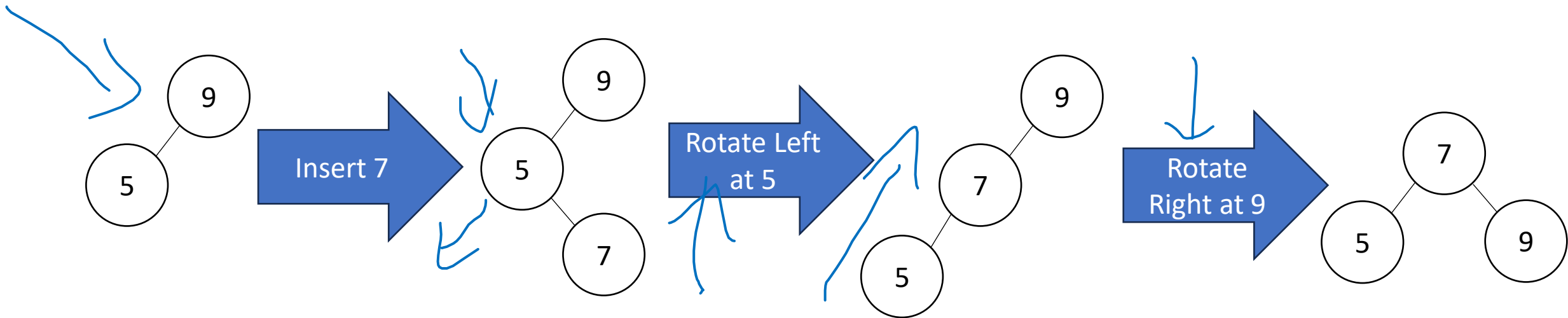


- After insertion, update the heights of the node's ancestors
- Check for unbalance
- If unbalanced then at the deepest unbalanced root:
 - Case LL: If we inserted in the **left** subtree of the **left** child then rotate right
 - Case RR: If we inserted in the **right** subtree of the **right** child then rotate left
 - Case LR: If we inserted into the **right** subtree of the **left** child then ???
 - Case RL: If we inserted into the **left** subtree of the **right** child then ???

Cases LR and RL require 2 rotations!

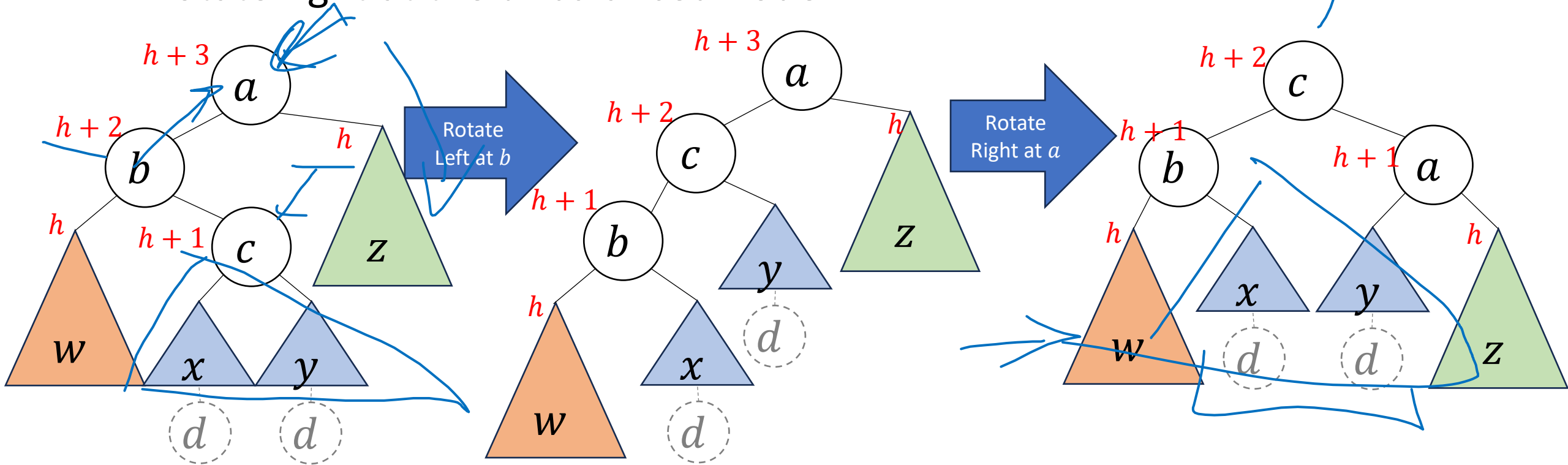
Case LR

- From deepest unbalanced root:
 - Rotate left at the left child
 - Rotate right at the root



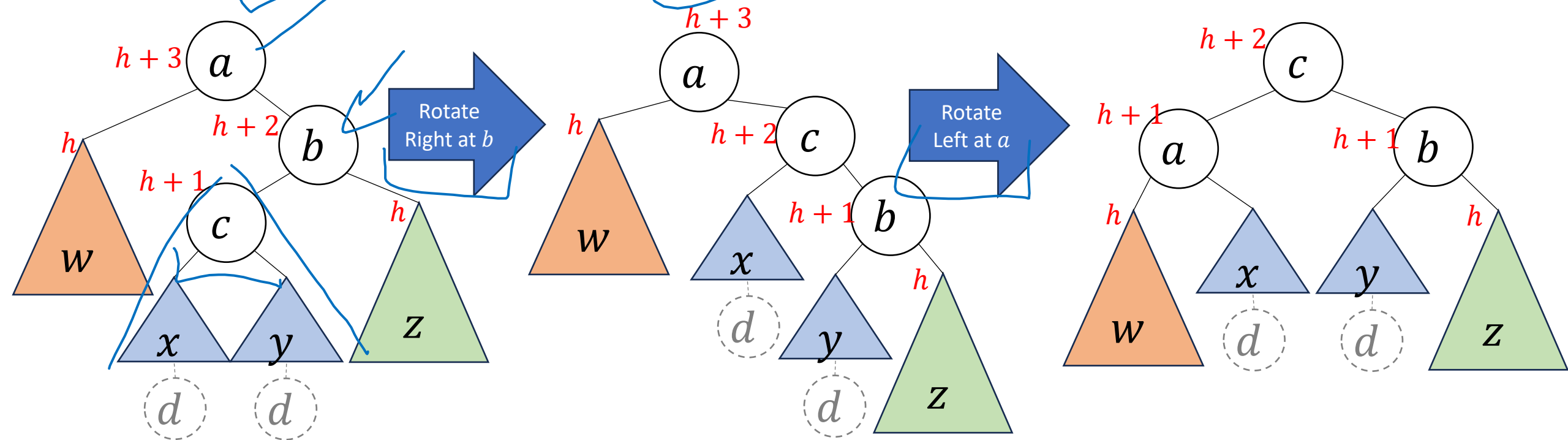
Case LR in General

- Imbalance caused by inserting in the left child's right subtree
- Rotate left at the left child
- Rotate right at the unbalanced node



Case RL in General

- Imbalance caused by inserting in the right child's left subtree
- Rotate right at the right child
- Rotate left at the unbalanced node



Insert Summary

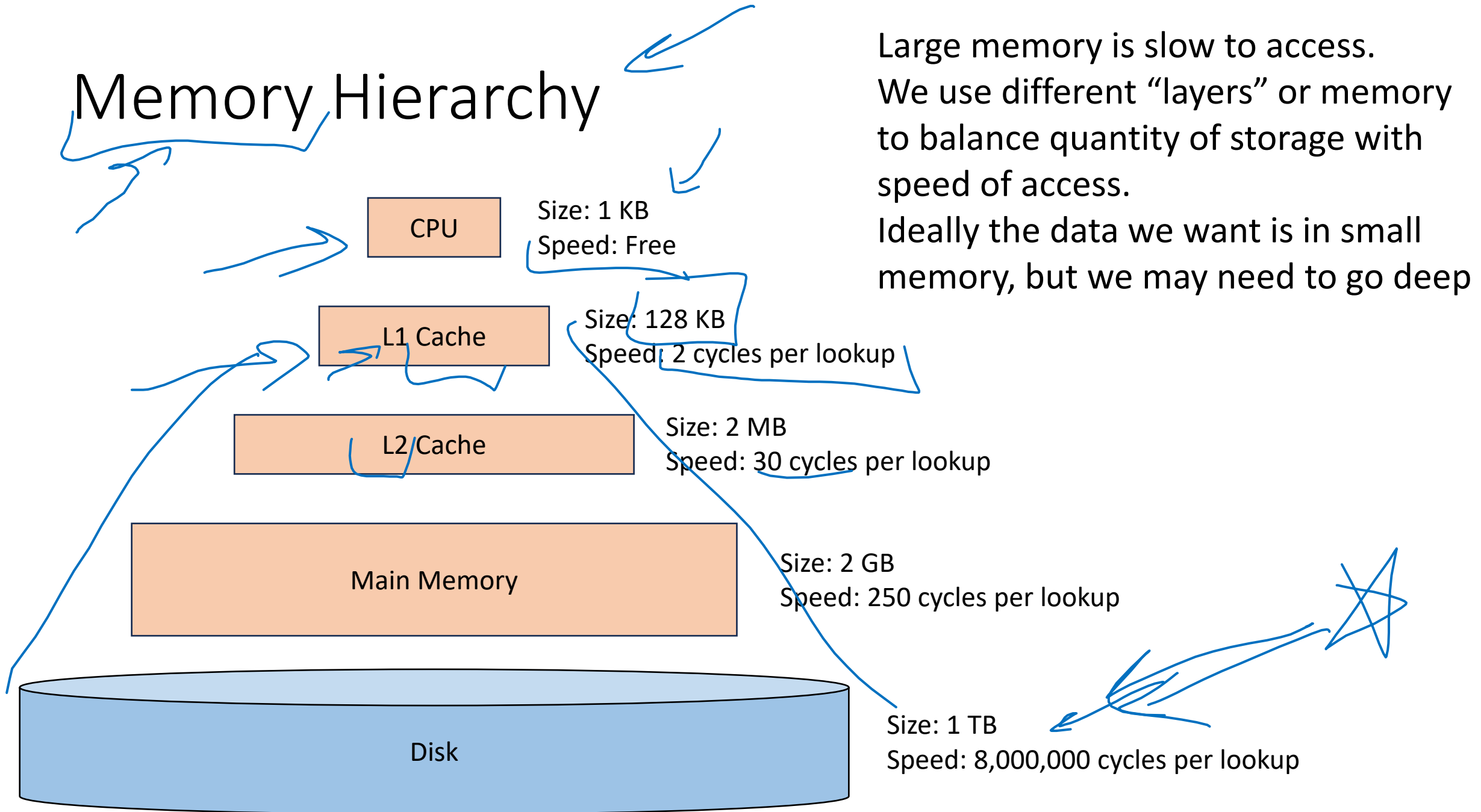
- After a BST insertion, update the heights of the node's ancestors
- From leaf to root, check if each node is unbalanced
- If a node is unbalanced then at the deepest unbalanced node:
 - Case LL: If we inserted in the **left** subtree of the **left** child then: rotate right
 - Case RR: If we inserted in the **right** subtree of the **right** child then: rotate left
 - Case LR: If we inserted into the **right** subtree of the **left** child then: rotate left at the left child and then rotate right at the root
 - Case RL: If we inserted into the **left** subtree of the **right** child then: rotate right at the right child and then rotate left at the root
- Done after either reaching the root or applying **one** of the above cases

Delete Summary



- Tldr: same cases, reverse direction of rotation, may need to repeat with ancestors
- After a BST deletion, update the heights of the node's ancestors
- From leaf to root, check if each node is unbalanced
- If a node is unbalanced then at the deepest unbalanced node:
 - Case LL: If we deleted in the **left** subtree of the **left** child then: **rotate left**
 - Case RR: If we deleted in the **right** subtree of the **right** child then: **rotate right**
 - Case LR: If we deleted into the **right** subtree of the **left** child then: **rotate right** at the left child and then **rotate left** at the root
 - Case RL: If we deleted into the **left** subtree of the **right** child then: **rotate left** at the right child and then **rotate right** at the root
- **Continue checking until reach the root**

Memory Hierarchy

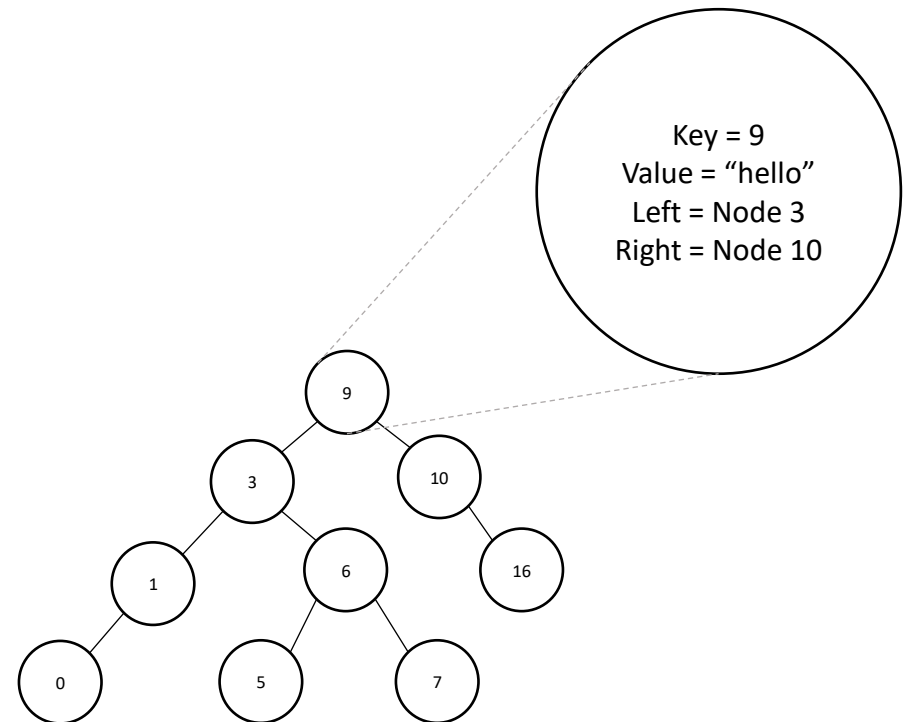


B Trees Motivation

- Memory Locality
 - Observation: in practice, when you read from memory you're likely to soon thereafter read from nearby memory
 - When memory is "fetched", it's collected in blocks at a time
 - Works well for arrays (they're contiguous in memory)
 - May not be helpful for linked lists, BSTs, etc. (pointers could go wherever)
- Solution: Have a BST-like data structure which can take advantage of locality

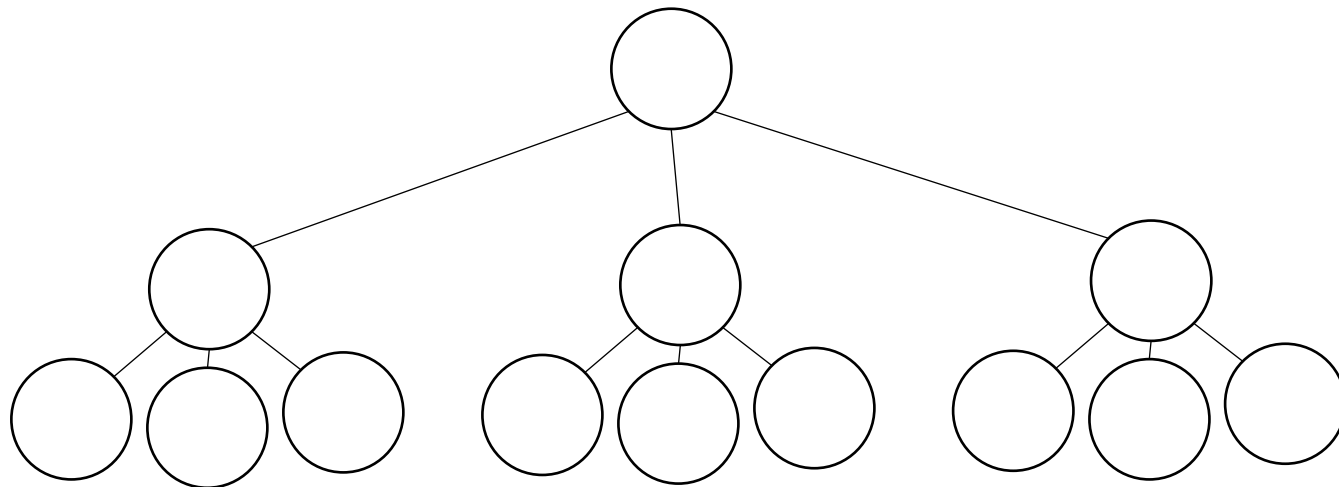
First Idea

- BST nodes have a lot of information inside them
- We don't need that information for "intermediate" nodes
- Solution: Delay loading anything except keys as long as possible



Second Idea

- Nodes may not be close to each other in memory
- In the worst case, each step in a traversal could go deep in memory
- Solution: Increase branching factor of tree load blocks of keys at a time
 - M-ary tree: each node has at most M children
 - Choose M to snugly fit in a block



B Trees (aka B+ Trees)

- Two types of nodes:

- Internal Nodes

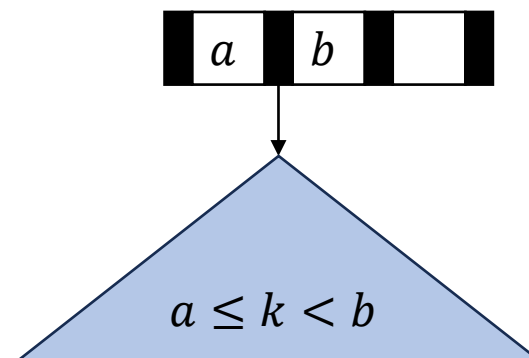
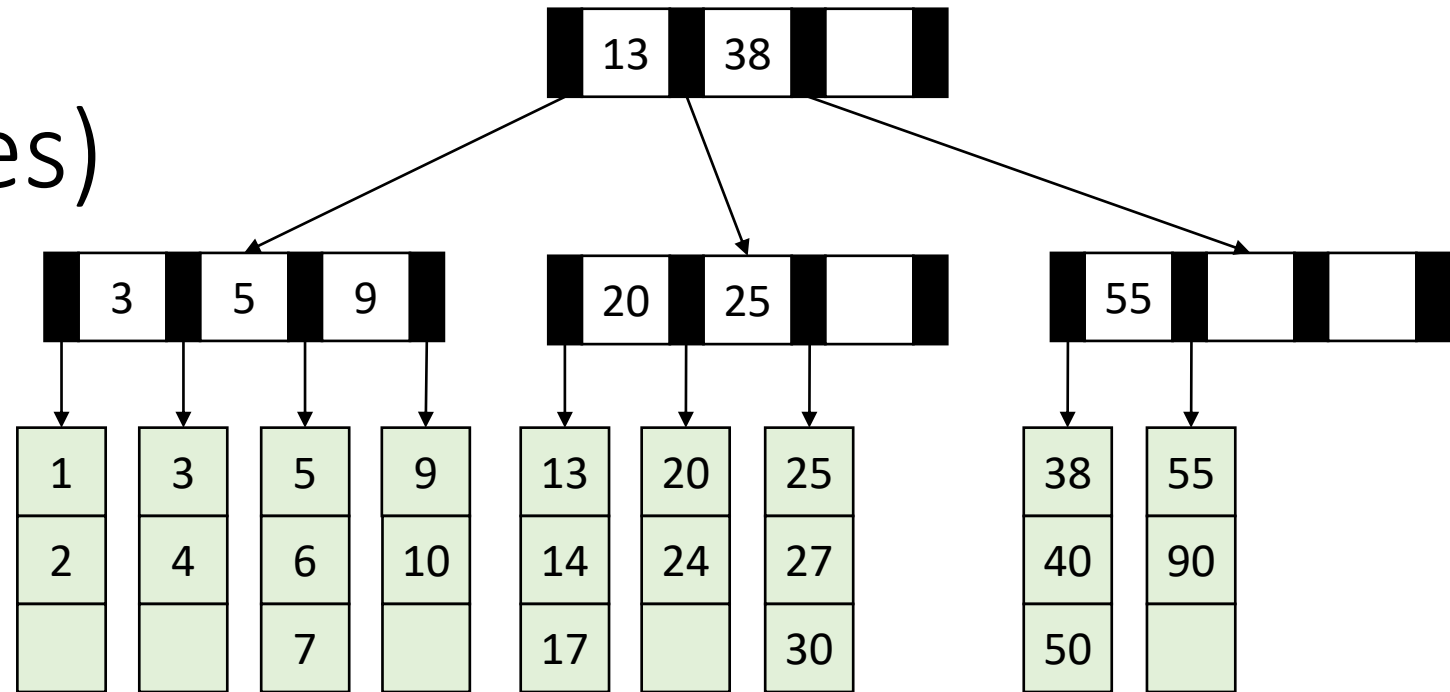
- Sorted array of $M - 1$ keys
 - Has M children
 - No other data!

- Leaf Nodes

- Sorted array of L key-value pairs

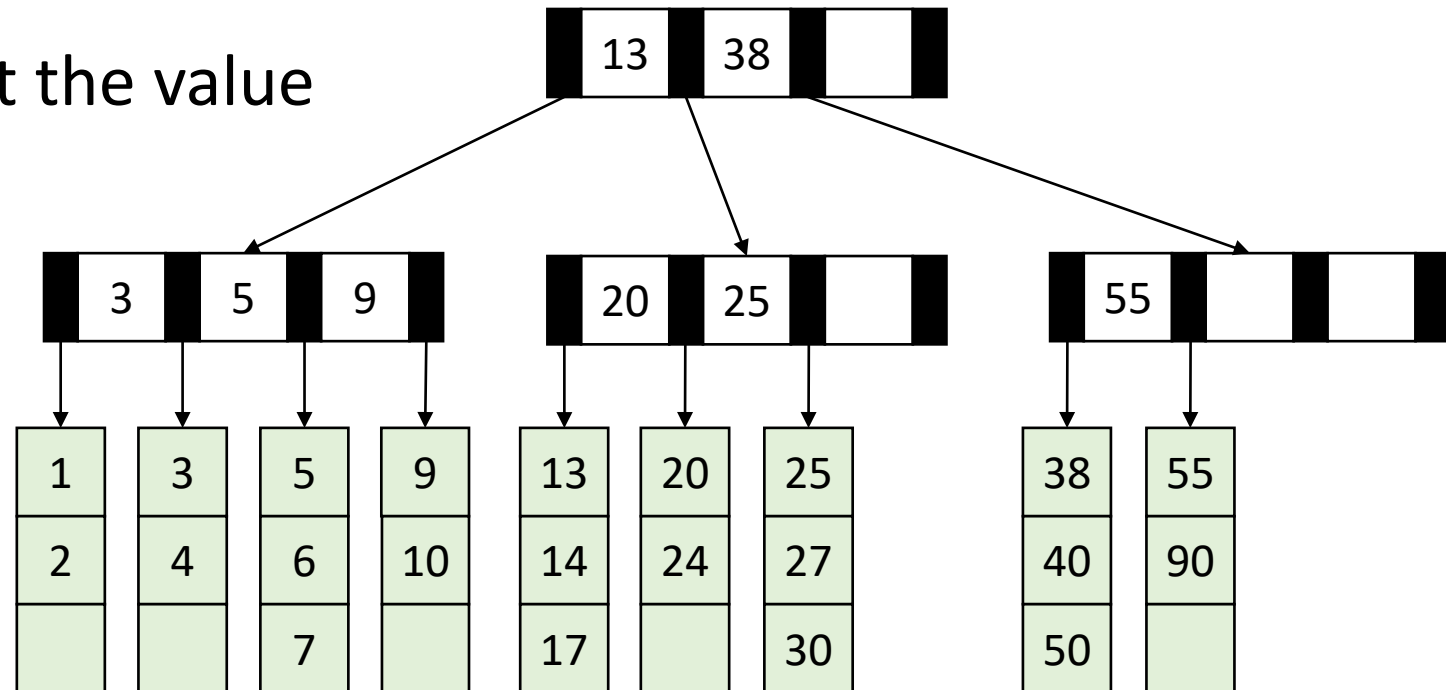
- Subtree between values a and b must contain only keys that are $\geq a$ and $< b$

- If a is missing use $-\infty$
 - If b is missing use ∞



Find

- Start at the root node
- Binary search to identify correct subtree
- Repeat until you reach a leaf node
- Binary search the leaf to get the value



B Tree Structure Requirements

- Root:
 - If the tree has $\leq L$ items then root is a leaf node
 - Otherwise it is an internal node
- Internal Nodes:
 - Must have at least $\lceil \frac{M}{2} \rceil$ children (at least have full)
- Leaf Nodes:
 - Must have at least Must have at least $\lceil \frac{L}{2} \rceil$ items (at least have full)
 - All leaves are at the same depth

Insertion Summary

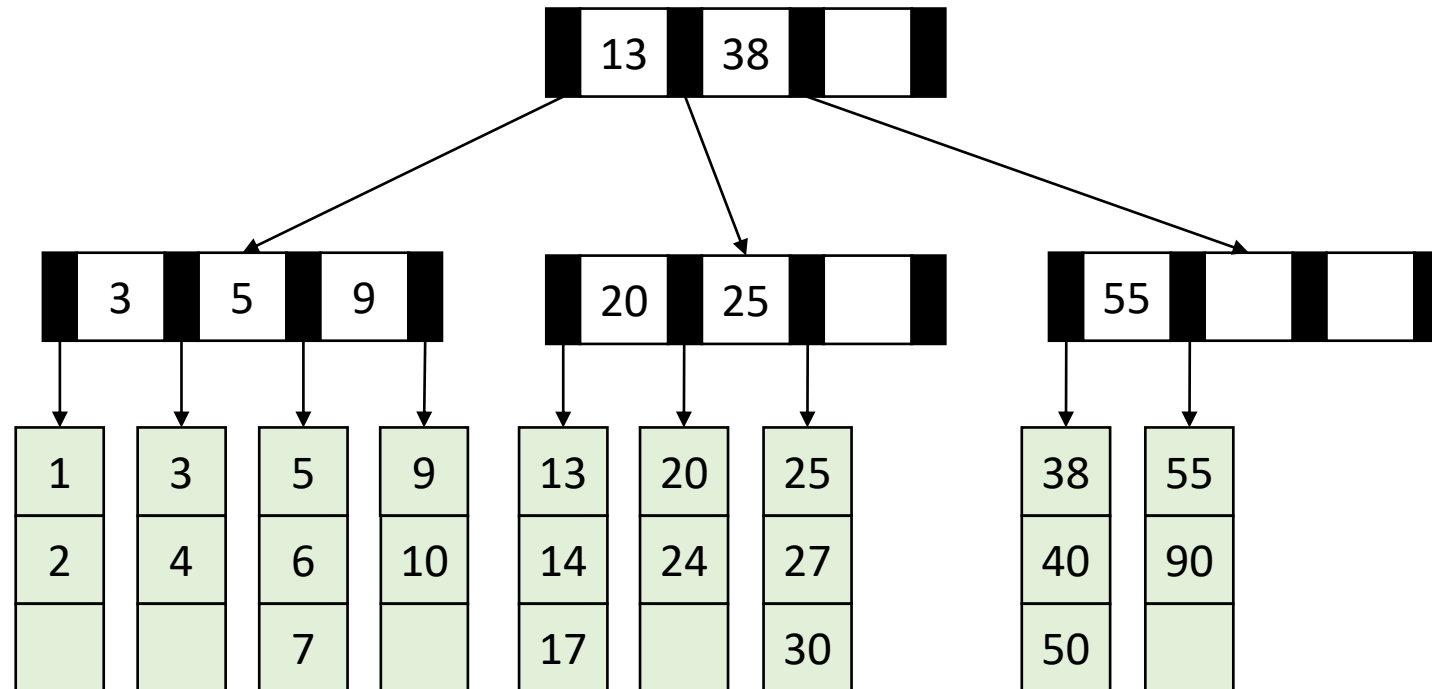
- Binary search to find which leaf should contain the new item
- If there's room, add it to the leaf array (maintaining sorted order)
- If there's not room, **split**
 - Make a new leaf node, move the larger half of the items to it
 - If there's room in the parent internal node, add new leaf to it (with new key bound value)
 - If there's not room in the parent internal node, **split** that!
 - Make a new internal node and have it point to the half the leaves (with correct key bound values)
 - If there's room in the parent internal node, add this internal node to it
 - If there's not room, repeat this process until there is!

Insertion TLDR

- Find where the item goes by repeated binary search
- If there's room, just add it
- If there's not room, split things until there is

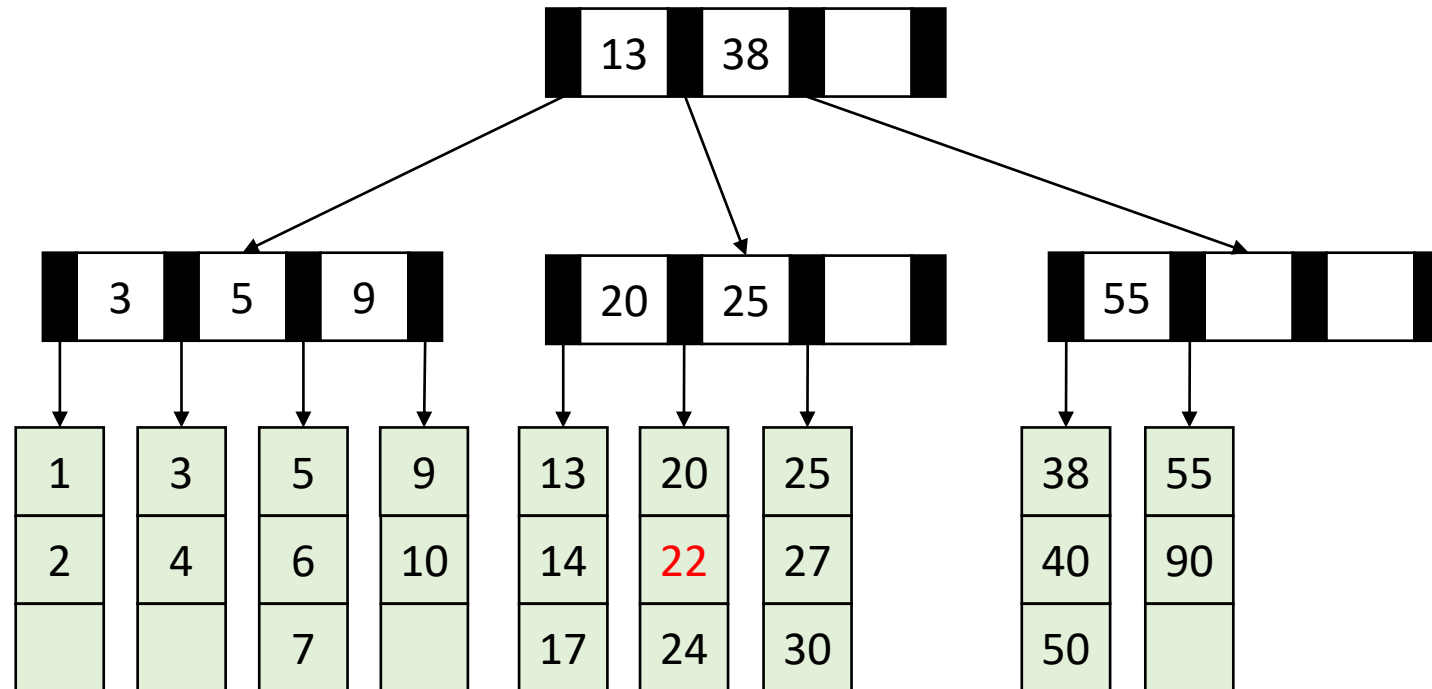
Insert Example

Insert 22



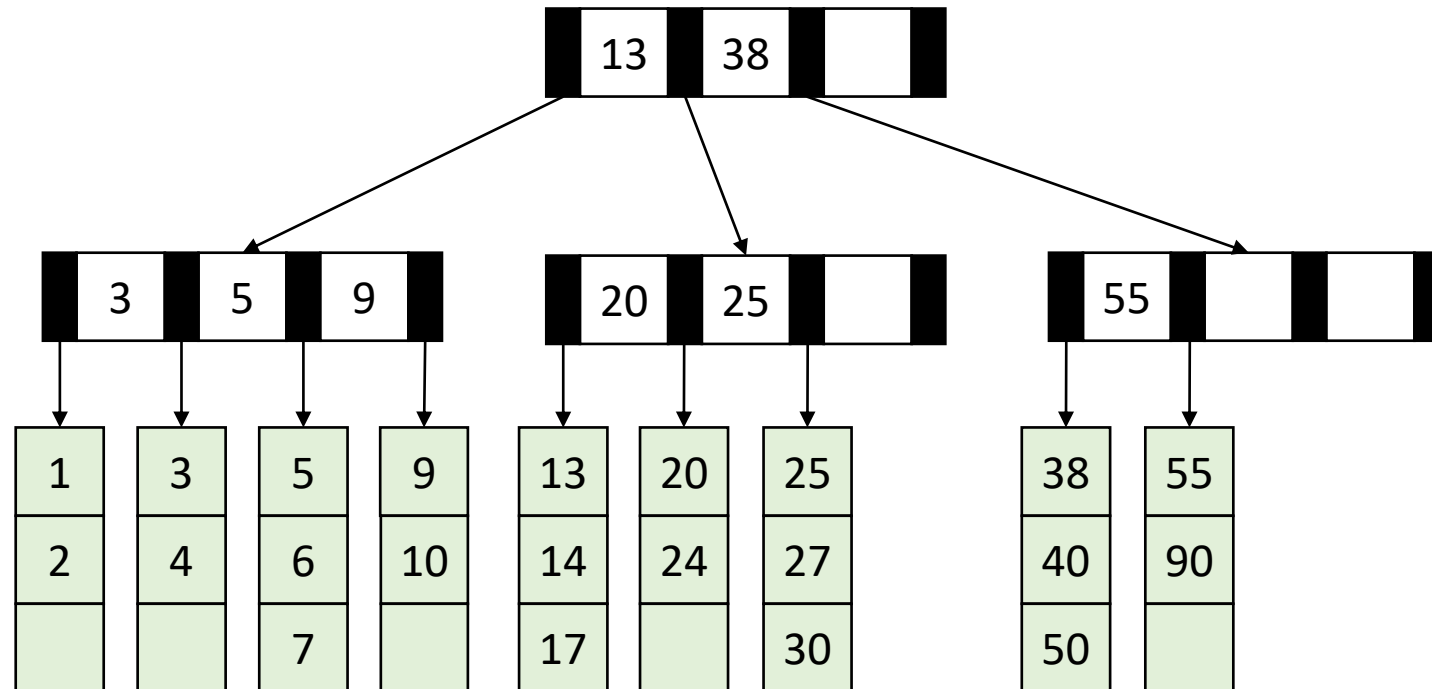
Insert Example

Insert 22



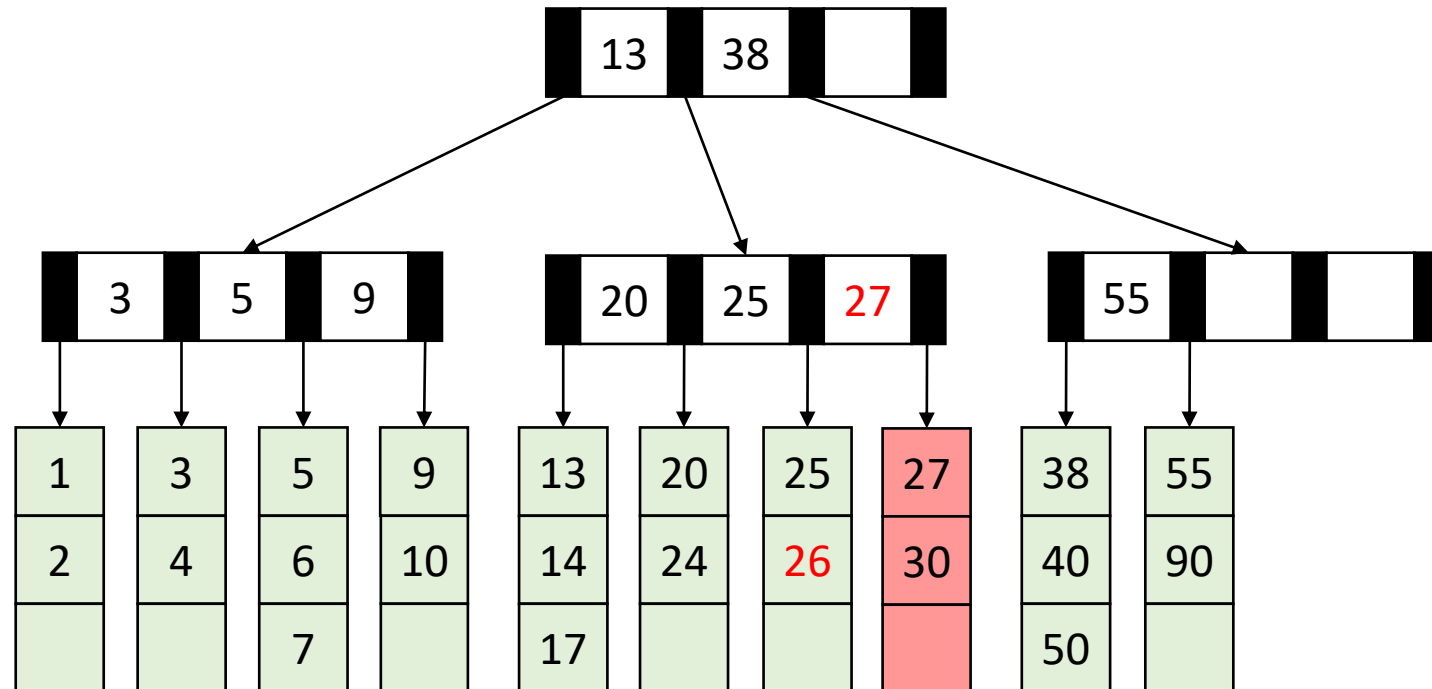
Insert Example

Insert 26



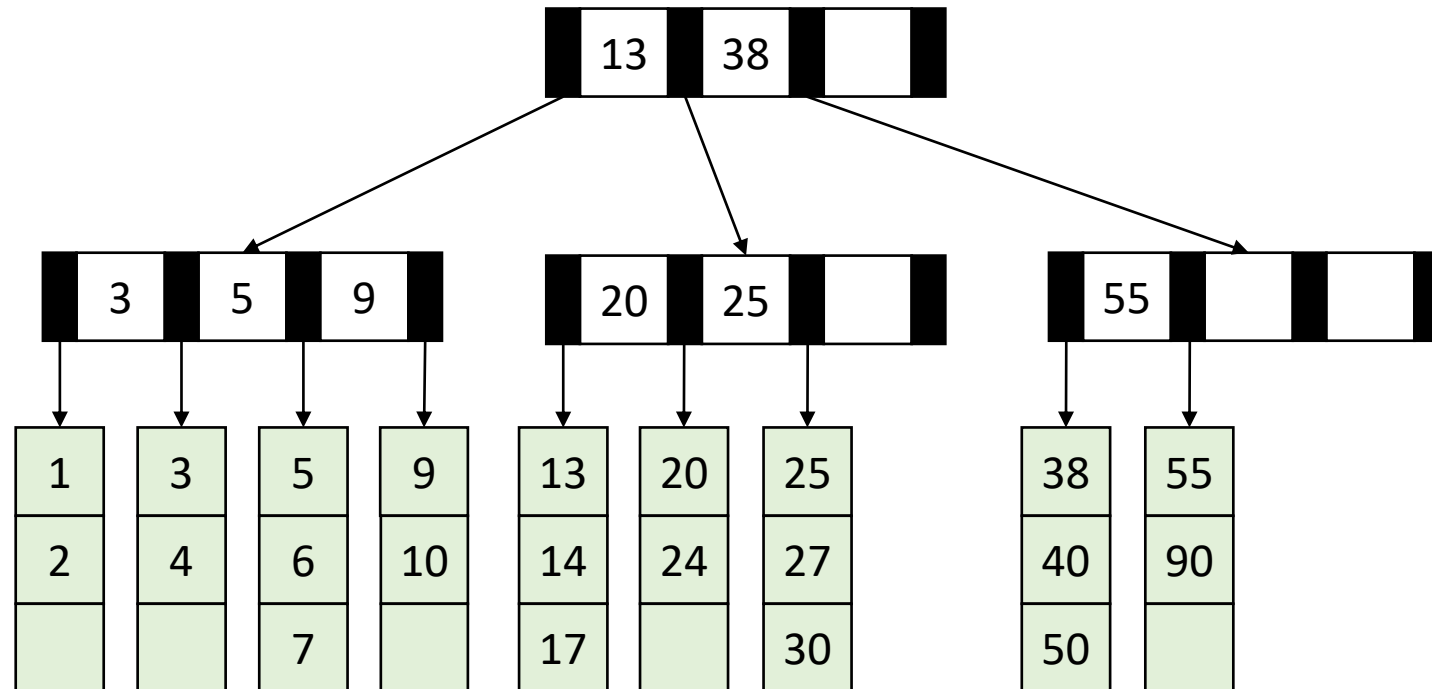
Insert Example

Insert 26



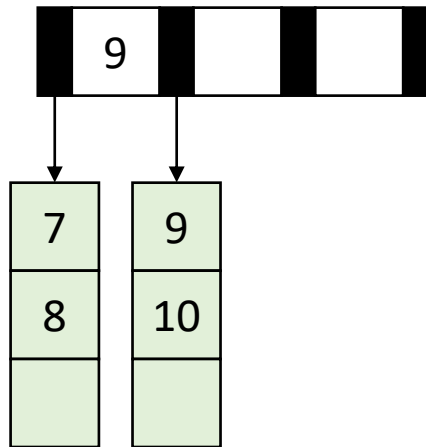
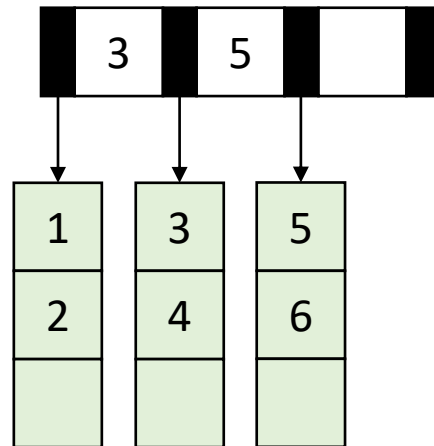
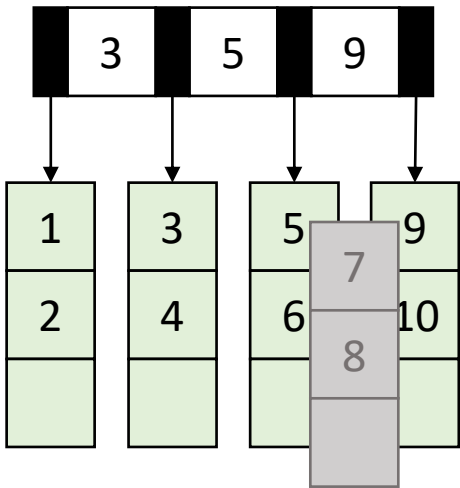
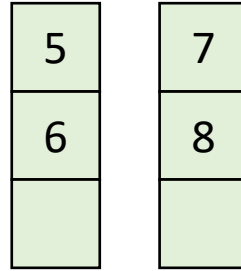
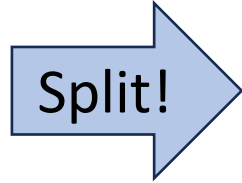
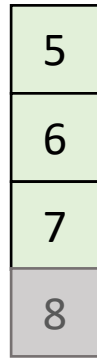
Insert Example

Insert 8



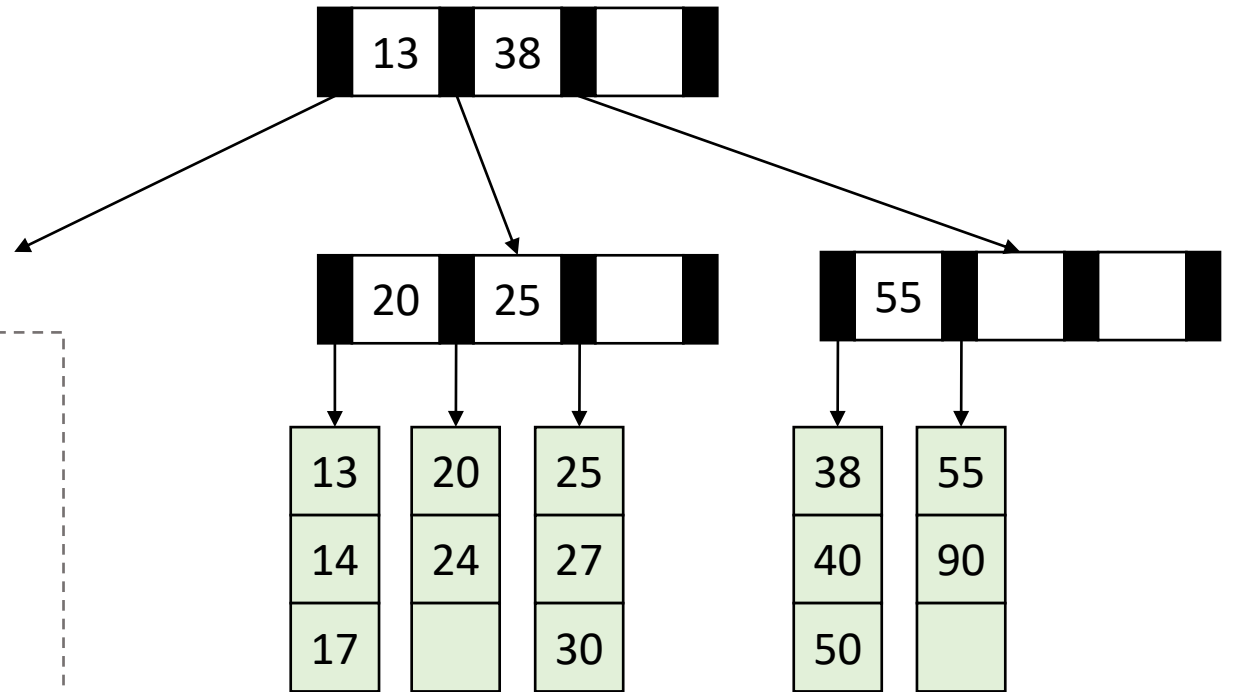
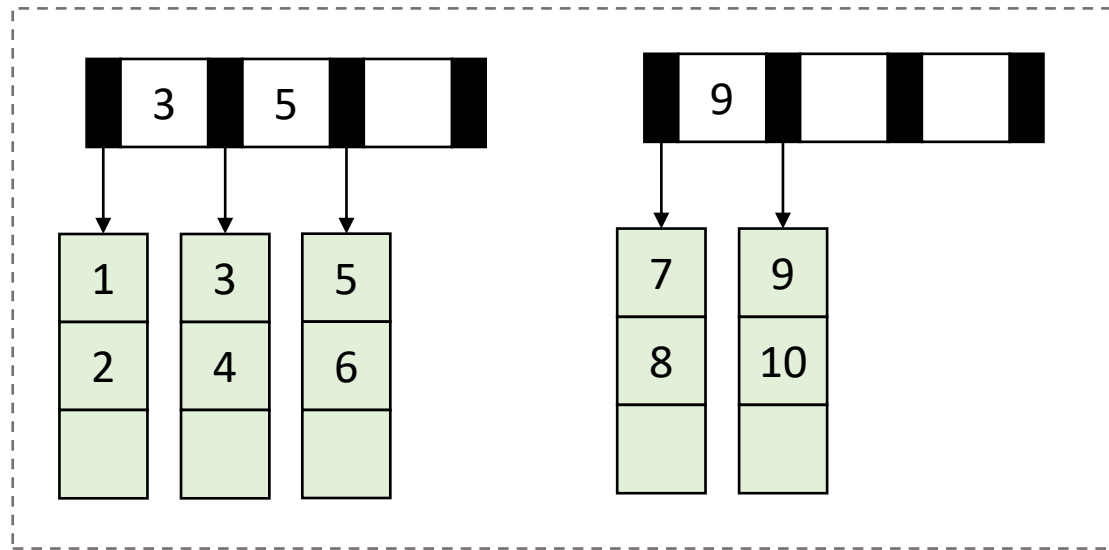
Insert Example

Insert 8



Insert Example

Insert 8



Insert Example

Insert 8

