Lecture 19

DICTIONARIES, HASHING &

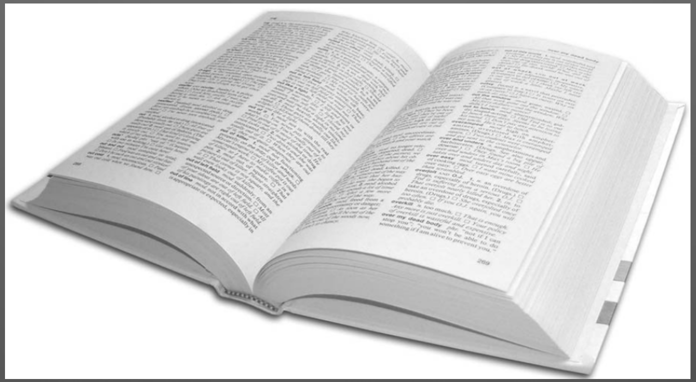
IMPROVED BALANCED SEARCH TREES

Outline

- Dictionary ADT and its Implementation
- Hash Functions
- Balanced Search Trees
- AVL Trees
- 2-3 Trees
- Red-Black Trees

What is a Dictionary?

- A dictionary is an associative container that stores elements in a mapped fashion. Each element has a key value and a mapped value.



- Example: a natural language dictionary: keys are words, value are the entries (part of speech, definitions, etc).

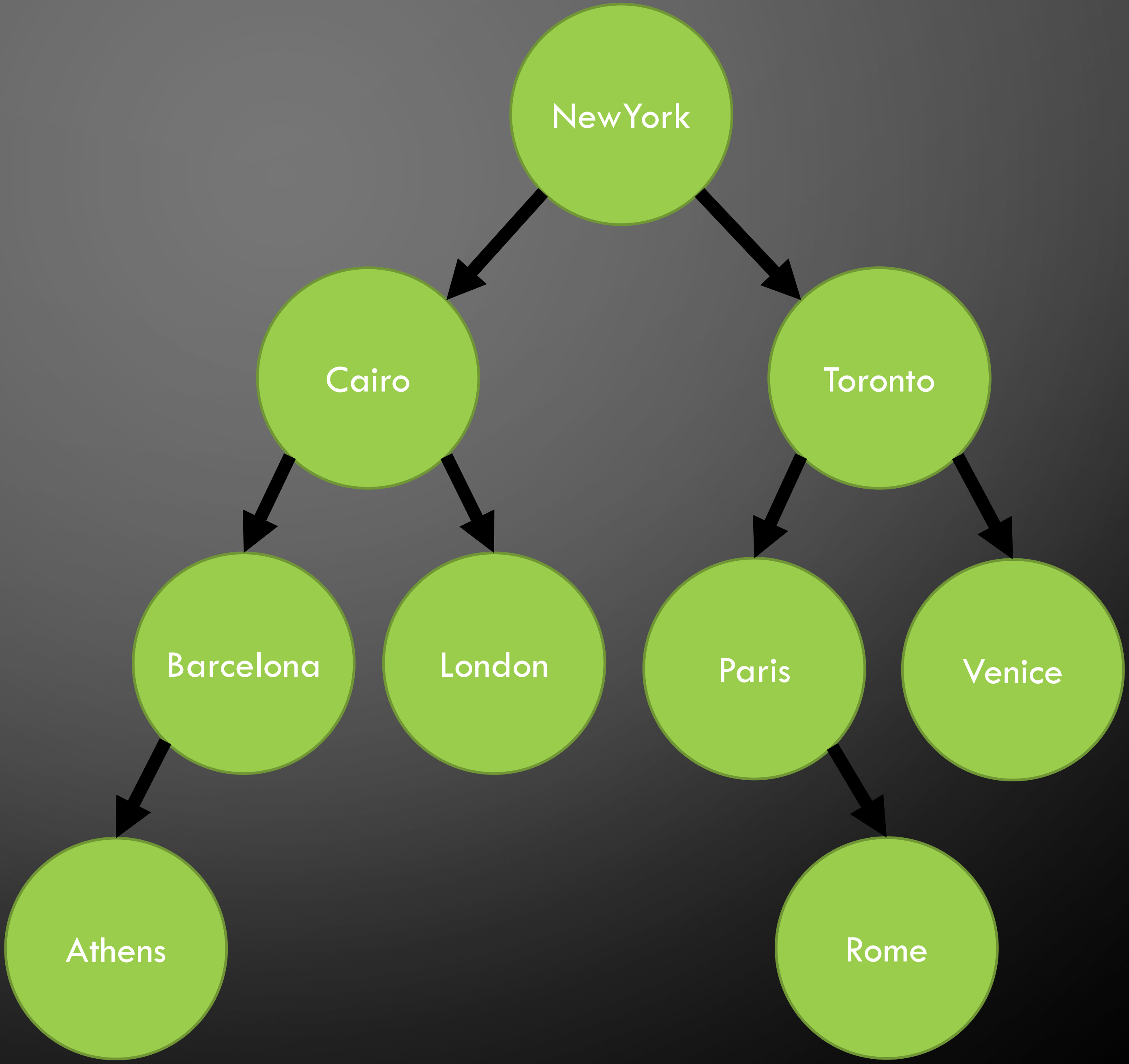
Dictionary ADT

- isEmpty()
- getNumberOfItems()
- add(searchKey, newItem)
- remove(searchKey)
- clear()
- getItem(searchKey)
- contains(searchKey)

Implementations

- List
- Sorted List
- Binary Search Tree

String Matching is Inelegant

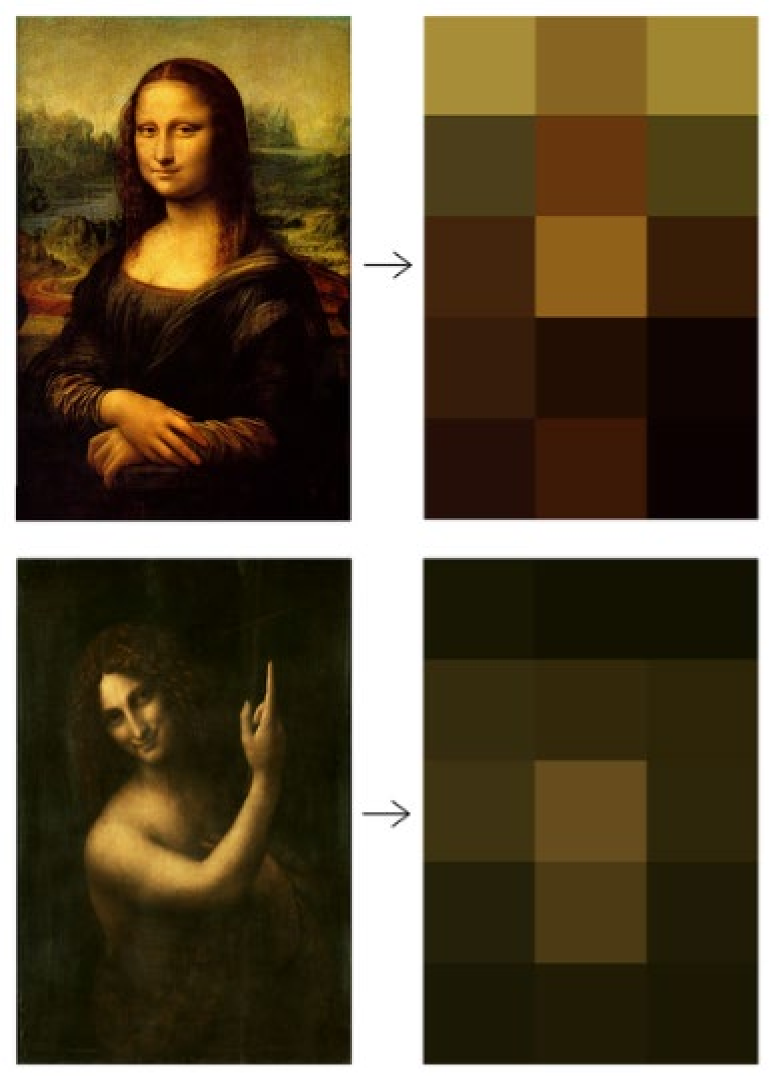


Duplicated Keys

- NewYork (NY)
- Pittsburgh (PA)
- Pittsburg (KS)
- Arlington (TX)
- Salem (OR)
- Arlington (VA)
- Salem (MA)

What is a hash function?

- A hash function is any function that can be used to map data of arbitrary size onto data of a fixed size (smaller size).



Example

- Cryptography and Passwords (example: SHA)
- Error correction (example: CRC)
- Identification and verification (example: MD5)

Goal is to improve efficiency of access

Retrieve Using a Hash Function

- retrieve(in key:keyType, out item:itemType): bool
  - Node loc = hash(key)
  - if(loc.key != key)
    - return false
  - else
    - item = loc.item
    - return true endif
  - endif

Insert Using a Hash Function

- insert(in key:keyType, in item:itemType)
  - Node loc = hash(key)
  - loc.item = item

Collision

- Since the result of hash function is smaller than the original data size (can be integer or string), two keys can produce the same **result.**
- A hash that has no collision is called perfect

Example

0	→ 6
1	
2	→ 2
3	
4	
5	

- Hash function: $h(k) = k \bmod m$

Insert(2): $2 \bmod 6 = 2$

Insert(6): $6 \bmod 6 = 0$

Insert(8): $8 \bmod 6 = 2$ **collision**

Addressing Collisions: 1. Open addressing

- In open addressing, we move on to another slot. If that one is full, we move to another, ….
- This is called probing. We probe for an empty slot. (note this probe sequence must be repeatable)
- Several probing exist:
  - Linear probing: linearly probe for the next spot
  - Quadratic probing: we look for the i^2th slot in ith operation

0	6
1	
2	2
3	8
4	
5	

- Hash function: h(k) = k mod m

Insert(2): 2 mod 6 = 2

Insert(6): 6 mod 6 = 0

Insert(8): 8 mod 6 = 2 **collision**

Probing

- Linear Probing:
  - index = h(key)
  - while array[index] is full
    - index = index + 1 mod array.size
  - endwhile
- Quadratic Probing:
  - index = h(key)
  - probe = 1
  - while array[index] is full
    - index = h(key) + probe*probe mod array.size
    - probe += 1
  - endwhile

How to Determine if an Index is Full?

- Reserve an item value that indicates empty.
- Each array entry is a struct with item and empty fields
- Array is an array of pointers, with NULL indicating empty.

Addressing Collisions: 2. Chaining

- Make the hash table an array of linked lists

Example

0 →	6	
1		
2 →	2 →	8
3		
4		
5		

- Hash function: h(k) = k mod m

Insert(2): 2 mod 6 = 2

Insert(6): 6 mod 6 = 0

Insert(8): 8 mod 6 = 2 **collision**

Advantages/Disadvantages of Hashing

- Advantages: (good hash function, not close to full)
  - Insert is O(1)
  - Retrieve is O(1)
  - Delete is O(1)
- Disadvantages:
  - Traversals in order by key is (very) slow
  - Selection in a range of keys is (very) slow

Well-known Hash Functions

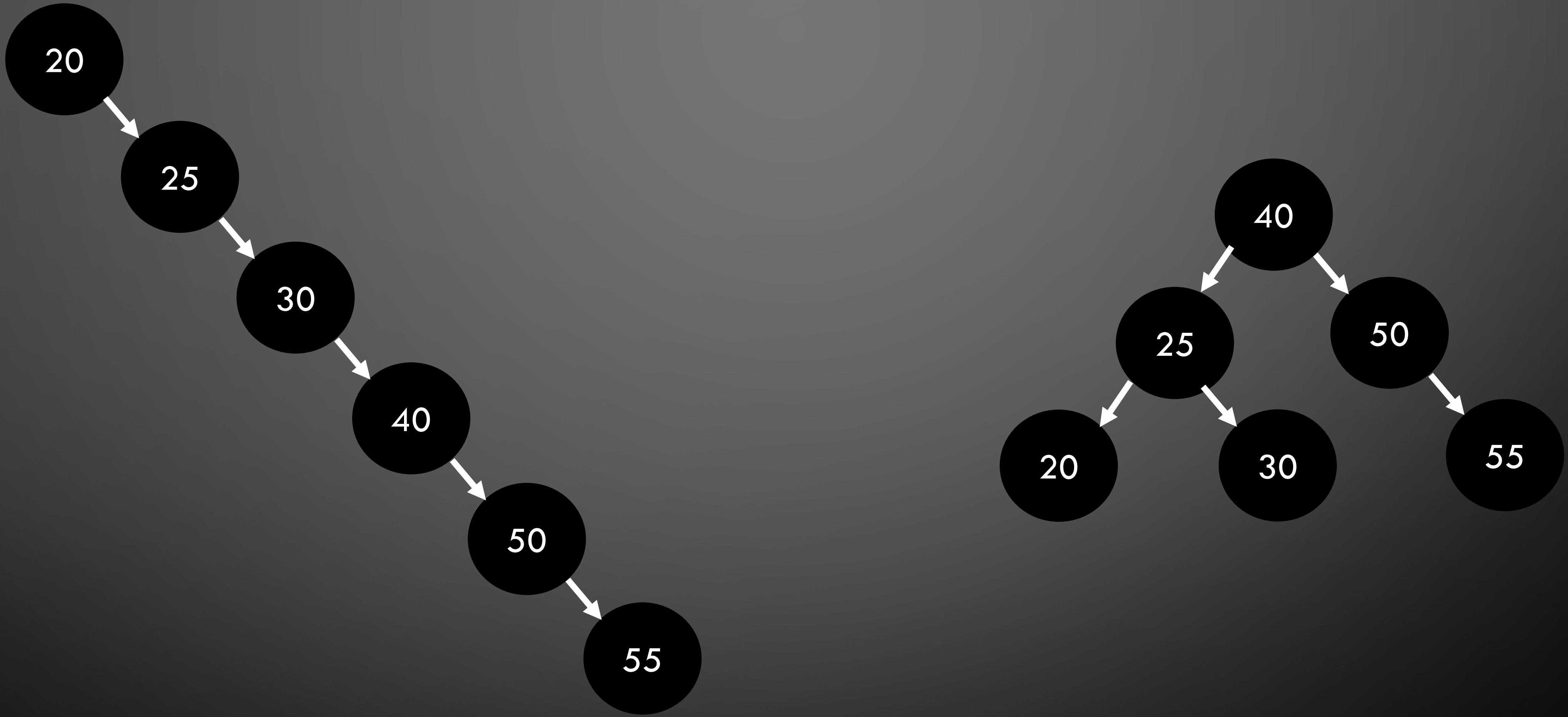
- RSHash
- JSHash
- ELFHash
- DEKHash
- MurmurHash

BST

- The basic operations are:
  - Insert
  - Delete
  - Search
- How the organization of a tree impacts the time to find a key?
- Questions:
  - what is the number of comparisons in the best case?
  - what is the number of comparisons in the worst case?
  - what is the number of comparisons in the average case?

BST

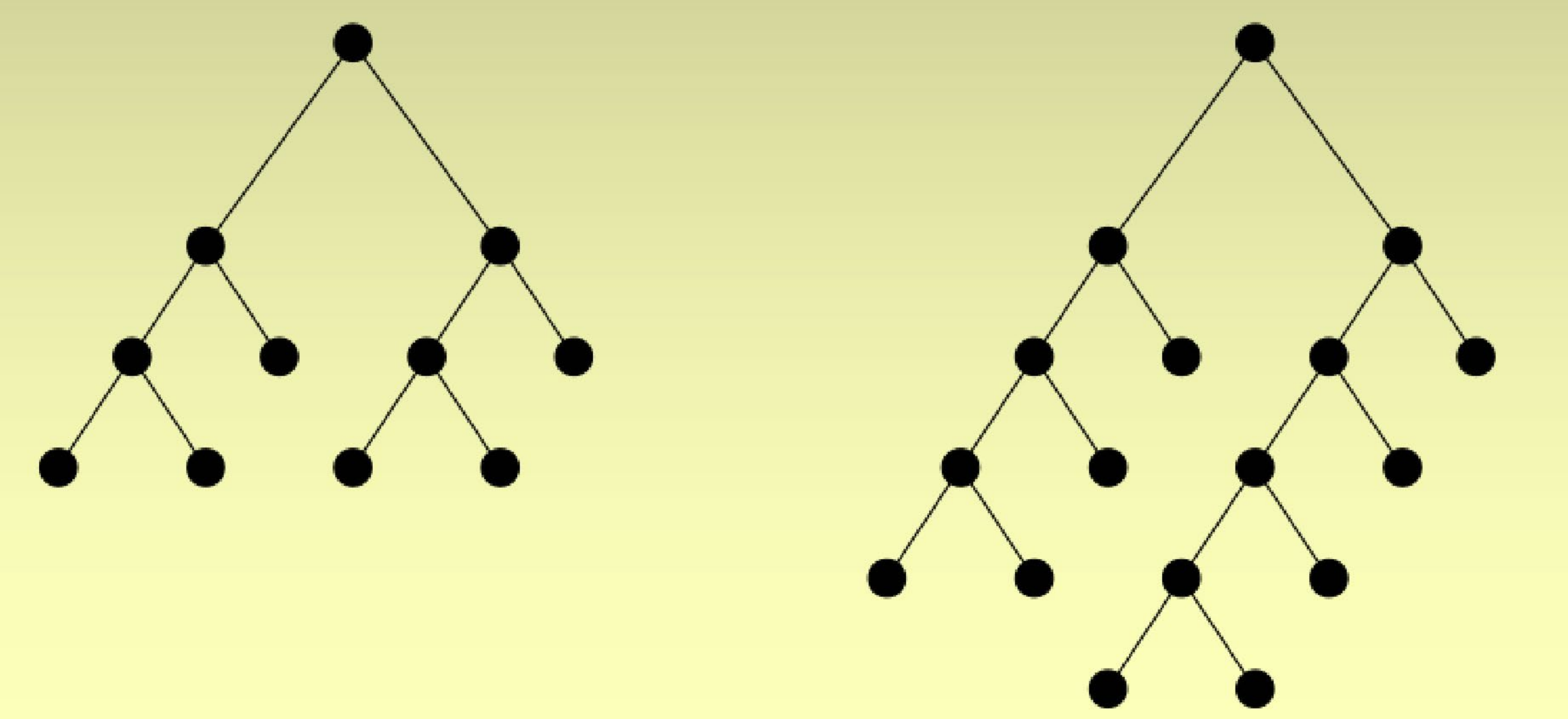
Example



The average complexity (number of comparisons) when searching a BST is best when the tree is balanced.

How can we force the BST to be balanced?

- Recall, a tree of height h is balanced if it is full down to level h-1; and the depth of a tree was the number of nodes from the root to a leaf.



Basic approach to making a balanced binary tree

1. Insert/Delete a node
2. Restore the balance of the tree

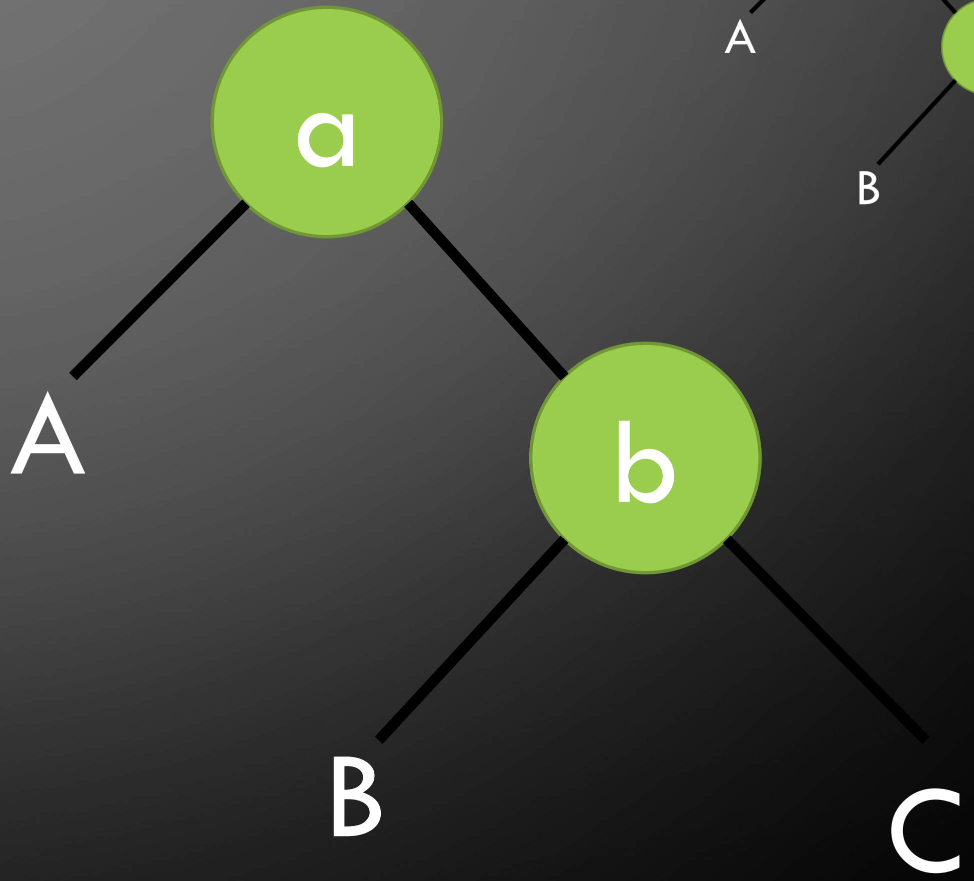
The primary tool used to restore balance is called a rotation

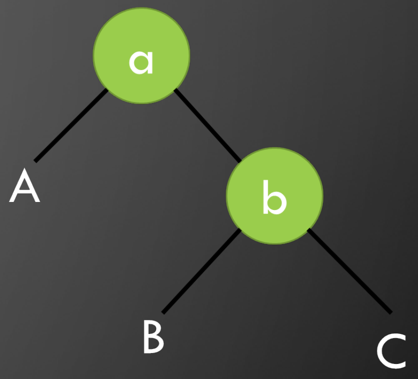
There are left and right rotations

The rotation should not violate the binary tree property

Left Rotation

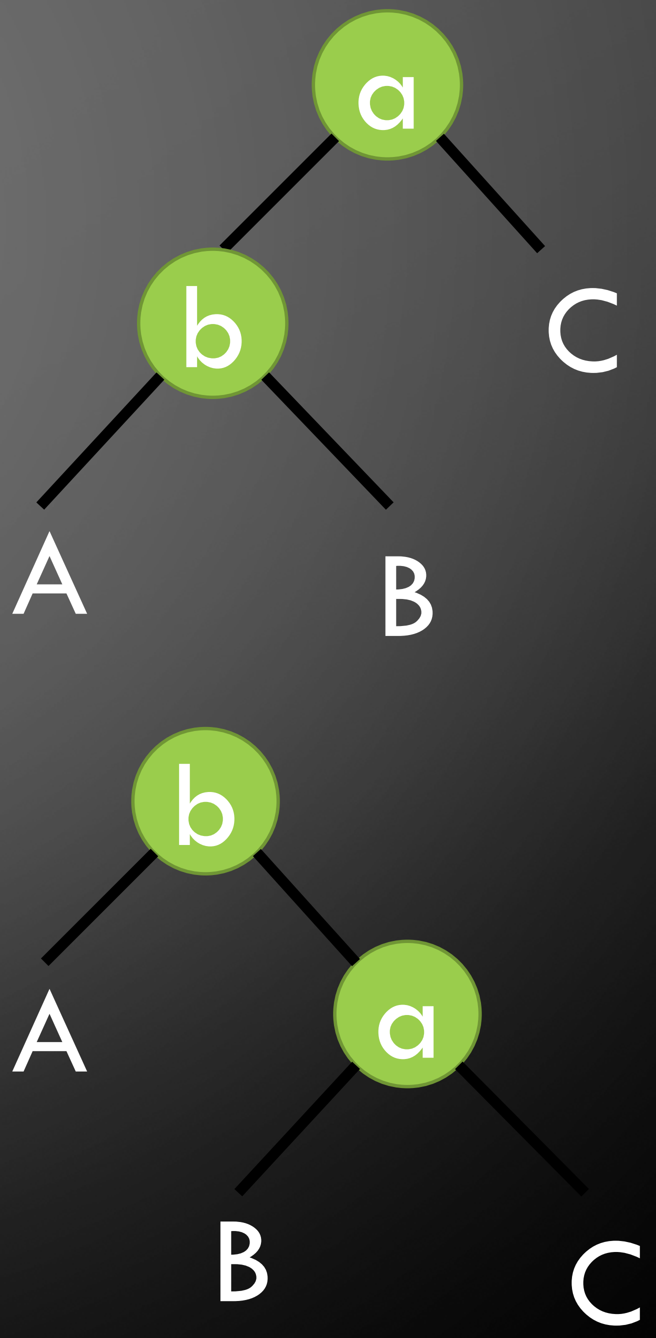
Let A, B, and C be subtrees and a, b nodes in the following tree



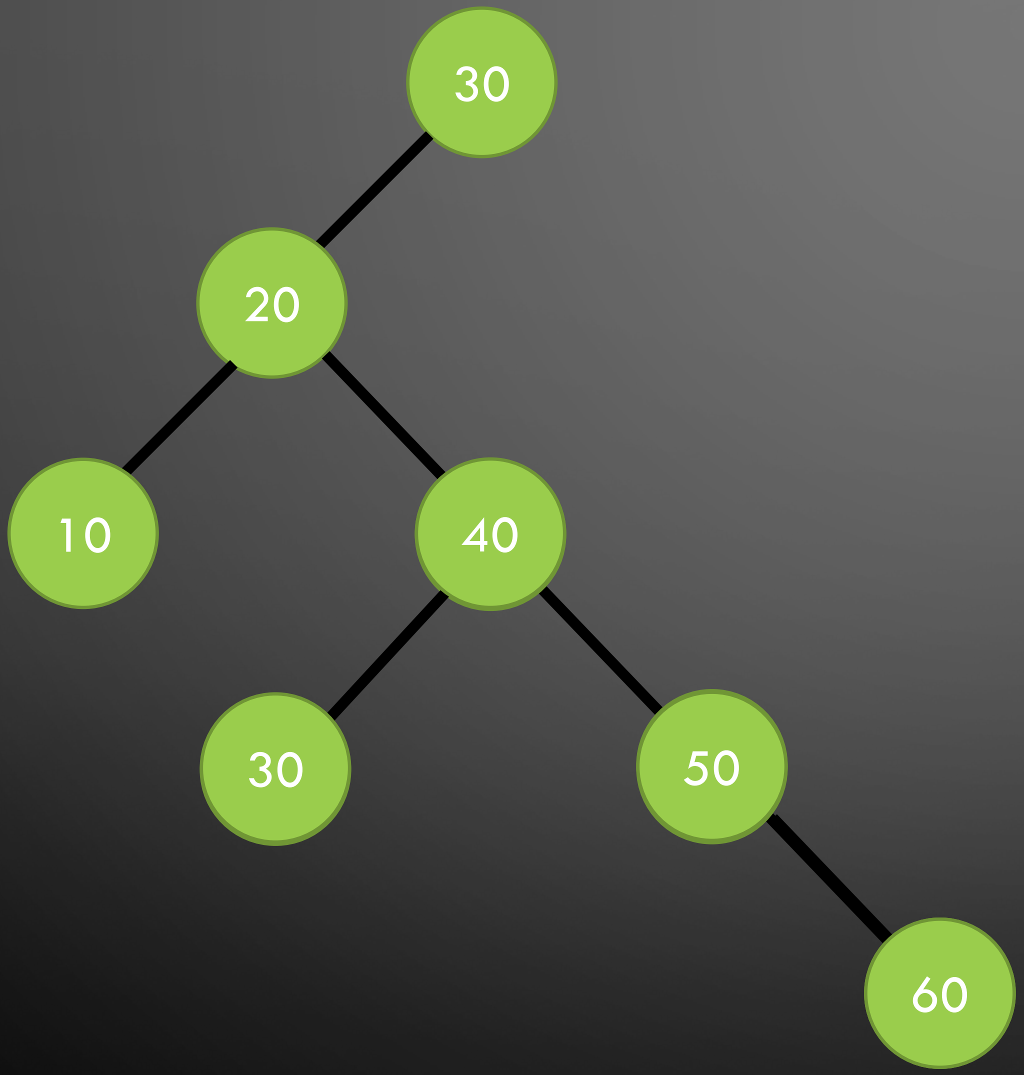


Rotate Right

```
// rotate a tree rooted at node
rotateRight(in node:TreeNode)
x = node
y = node->left_child
// if x is a left child
If x = x->parent->left_child
   x->parent->left_child=y
else // x is a right child
   x->parent->right_child=y
x->left_child = y->right_child
y->right_child = x
node = y
```

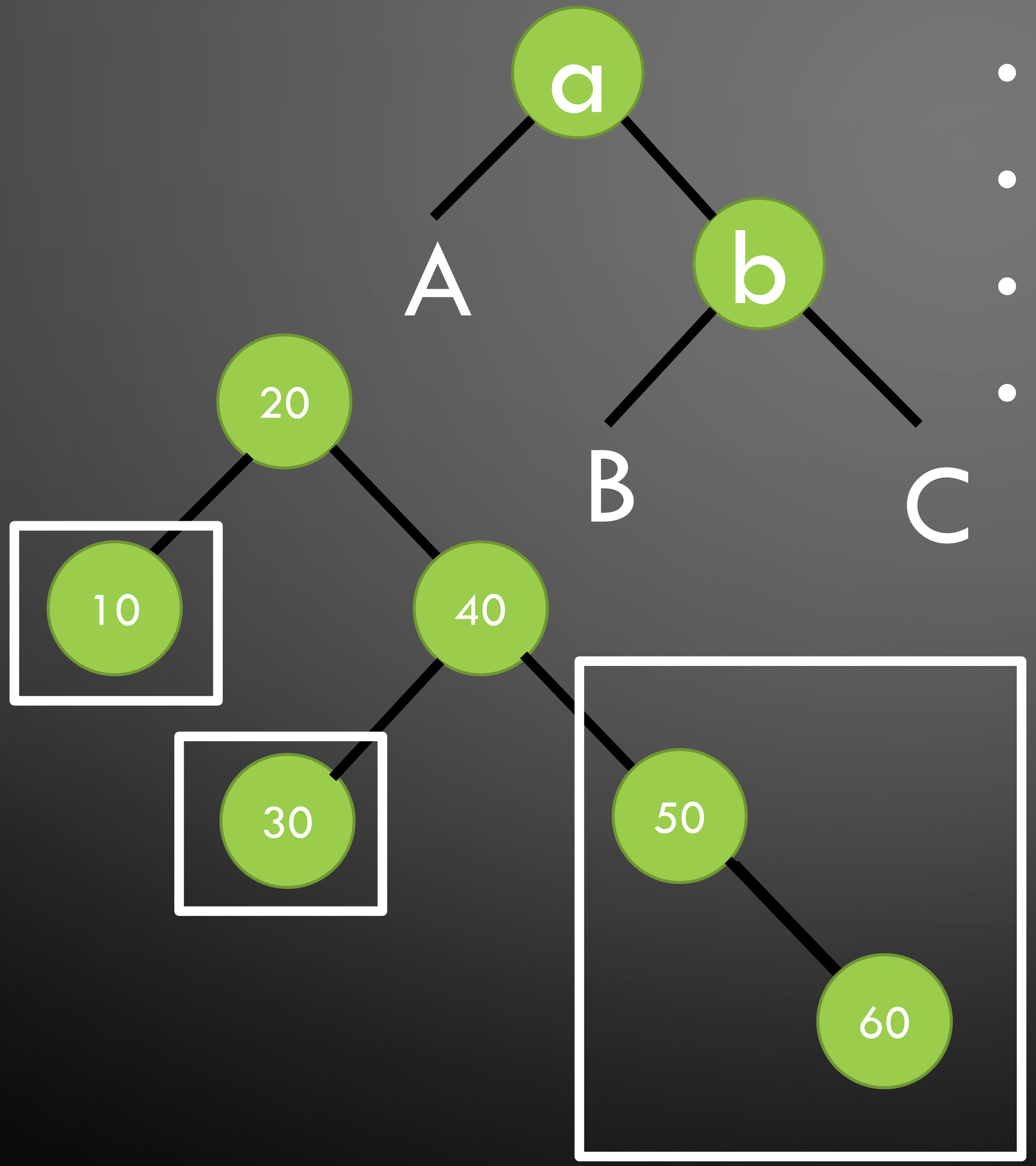


Keep the tree balanced – AVL Trees

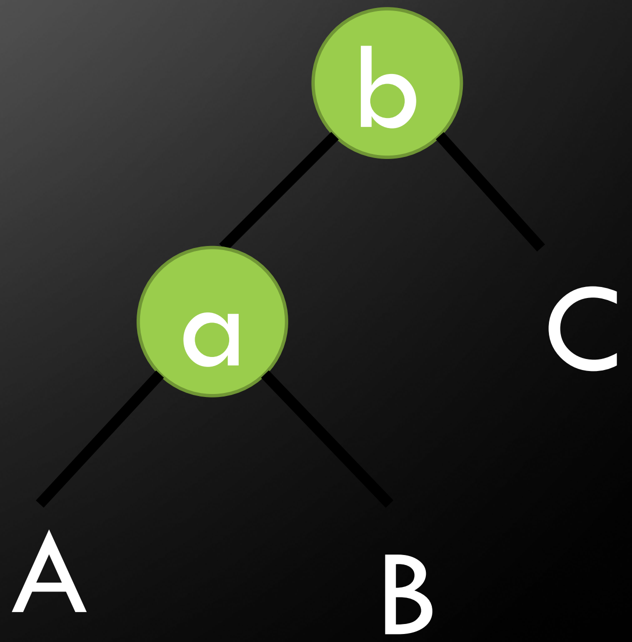


- Add 30
- Add 20
- Add 10
  - Tree is unbalanced
- Rotate Right
- Add 40
- Add 50
  - Tree is unbalanced
- Rotate subtree Left
- Add 60
  - Tree is unbalanced
- Where do I rotate?

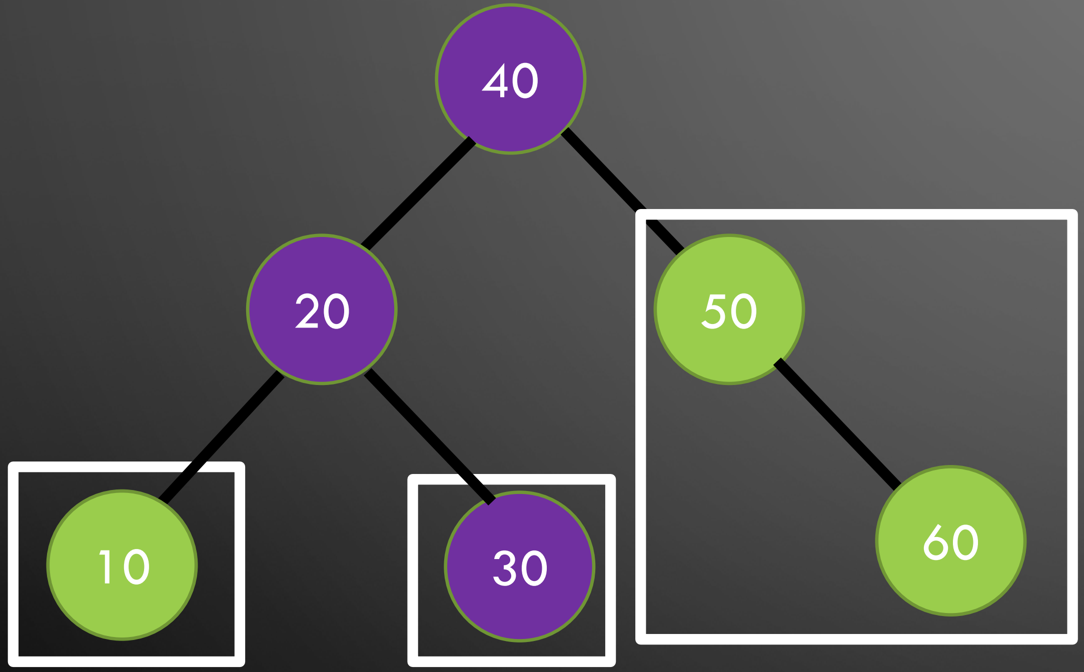
Keep the tree balanced – AVL Trees



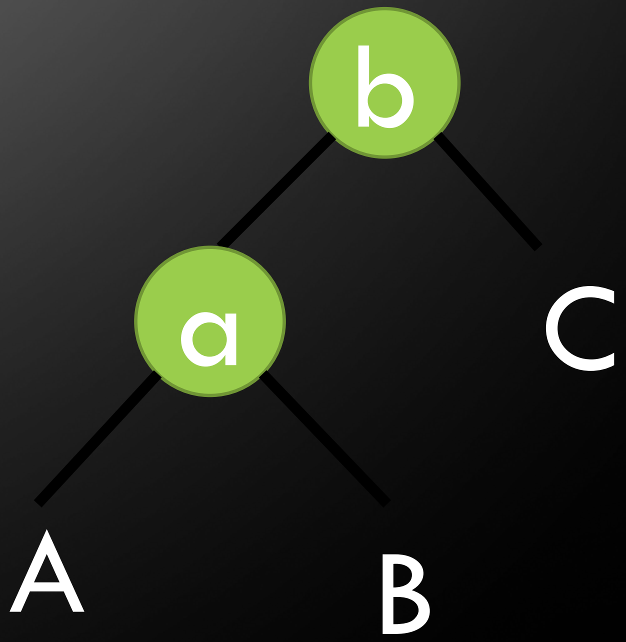
- Height of A is h
- Height of B is h+1
- Height of C is h+2
- Rotate left at root:



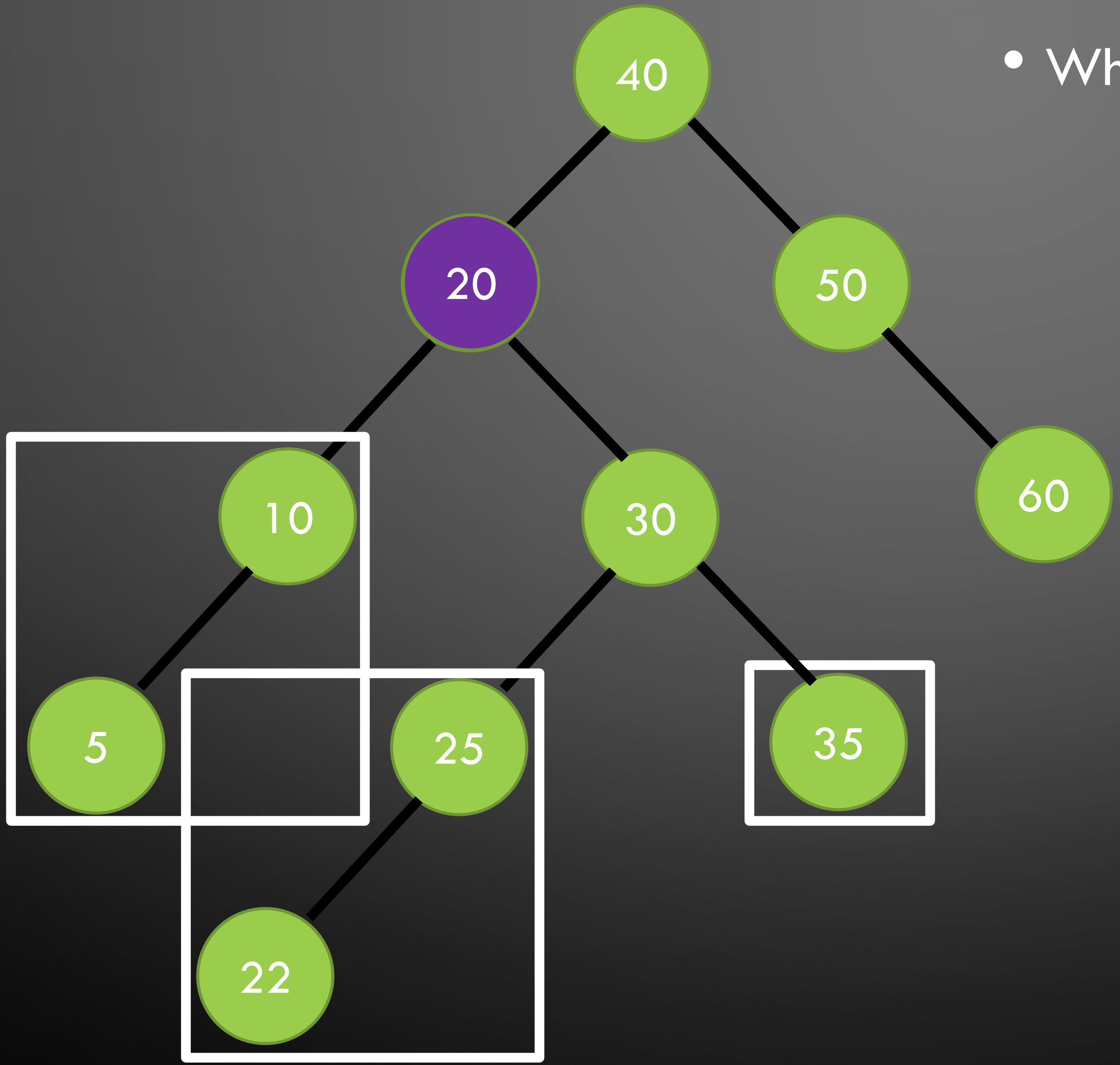
Keep the tree balanced – AVL Trees



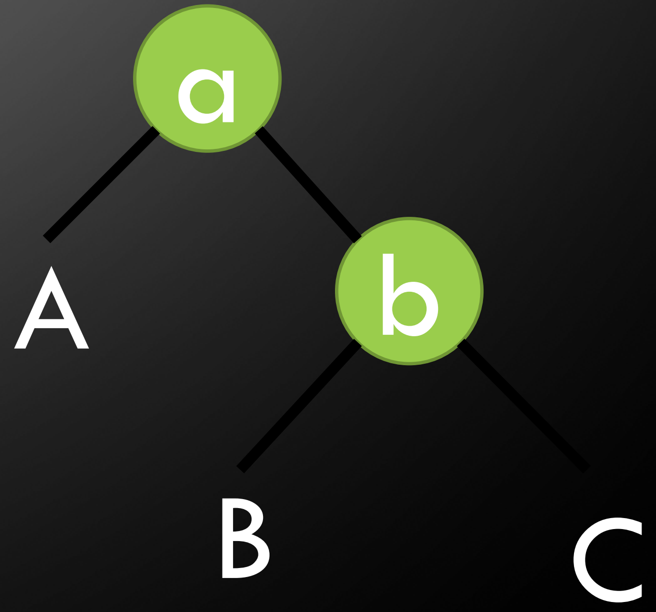
- New Height of A is h+1
- Height of B is h+1
- Height of C is h+1



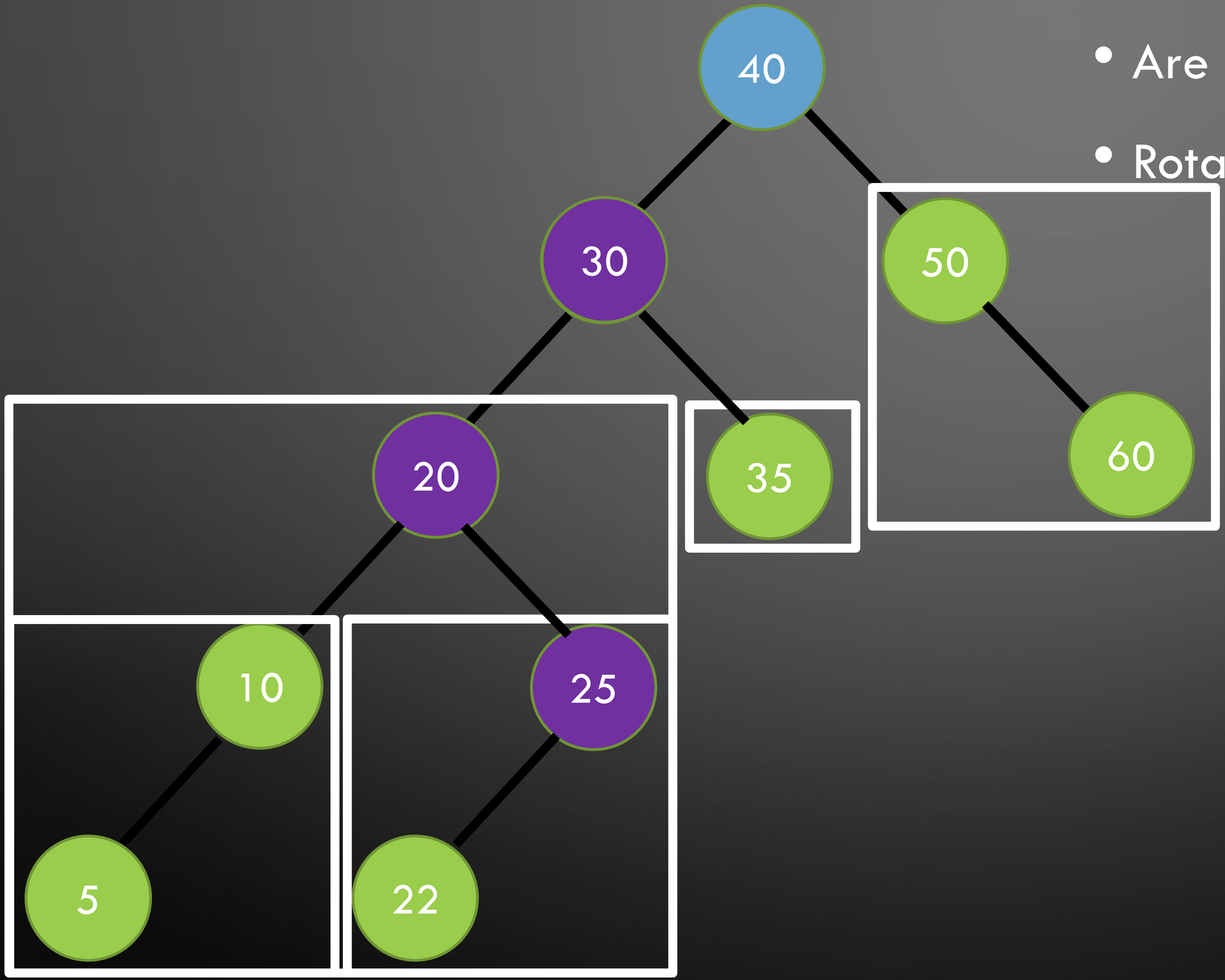
Keep the tree balanced – AVL Trees



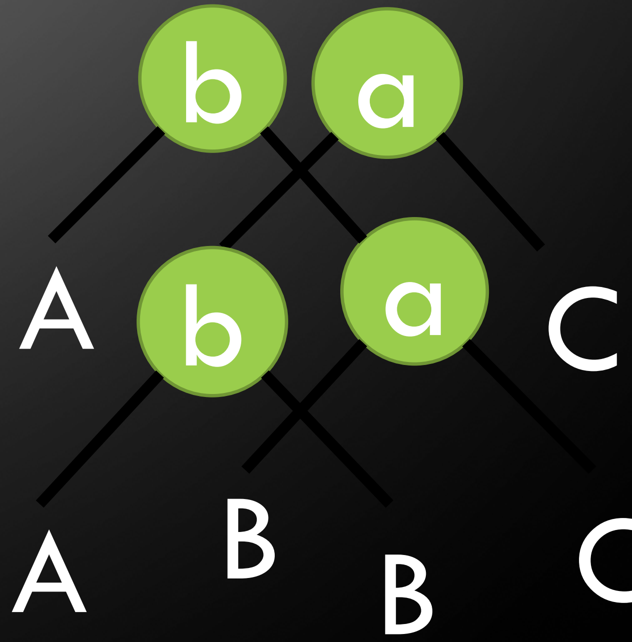
- Which direction do you rotate
- Where do you rotate?



Keep the tree balanced – AVL Trees

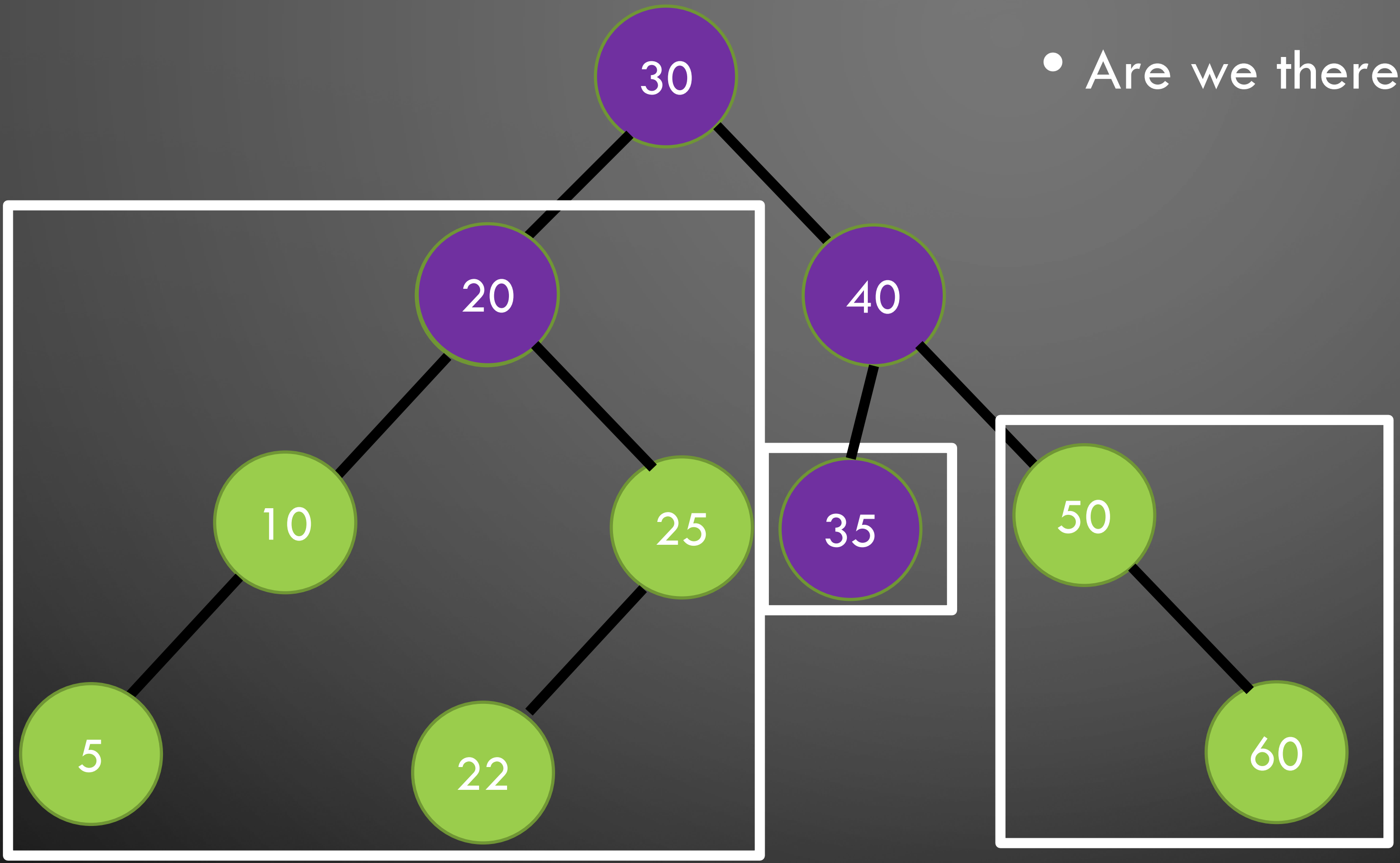


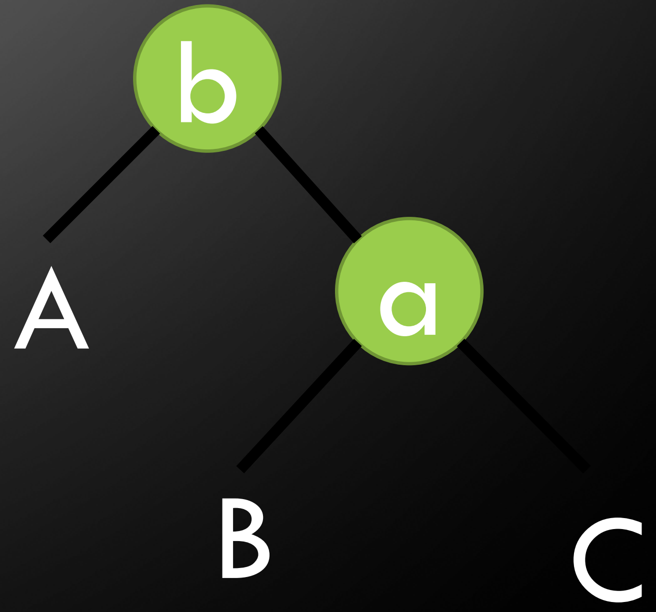
- After Left Rotation
- Are we there yet?
- Rotate Right



Keep the tree balanced – AVL Trees

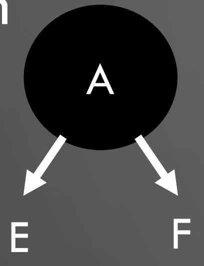
- After Right Rotation
- Are we there yet?

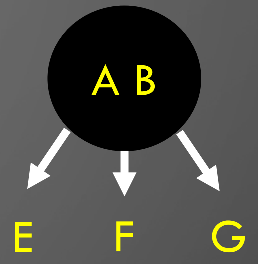




2-3 Trees

- 2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children





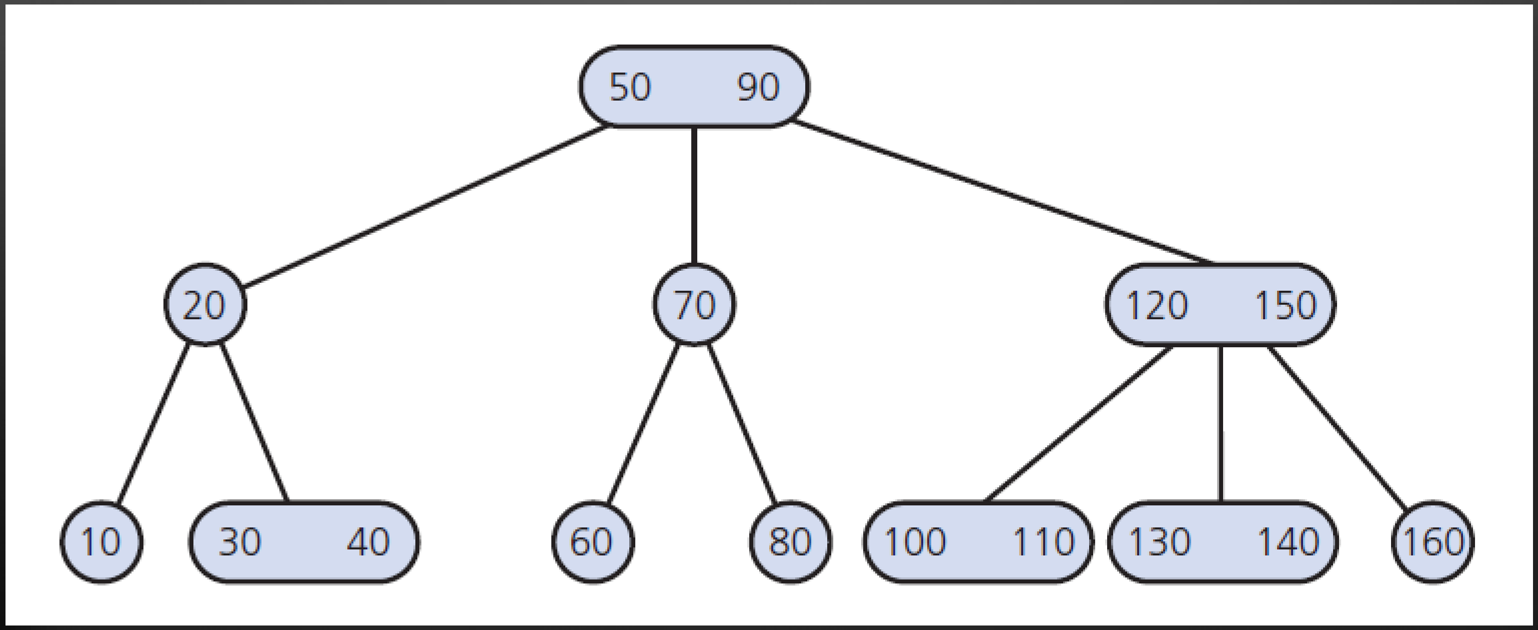
E is items < A

F is items > A

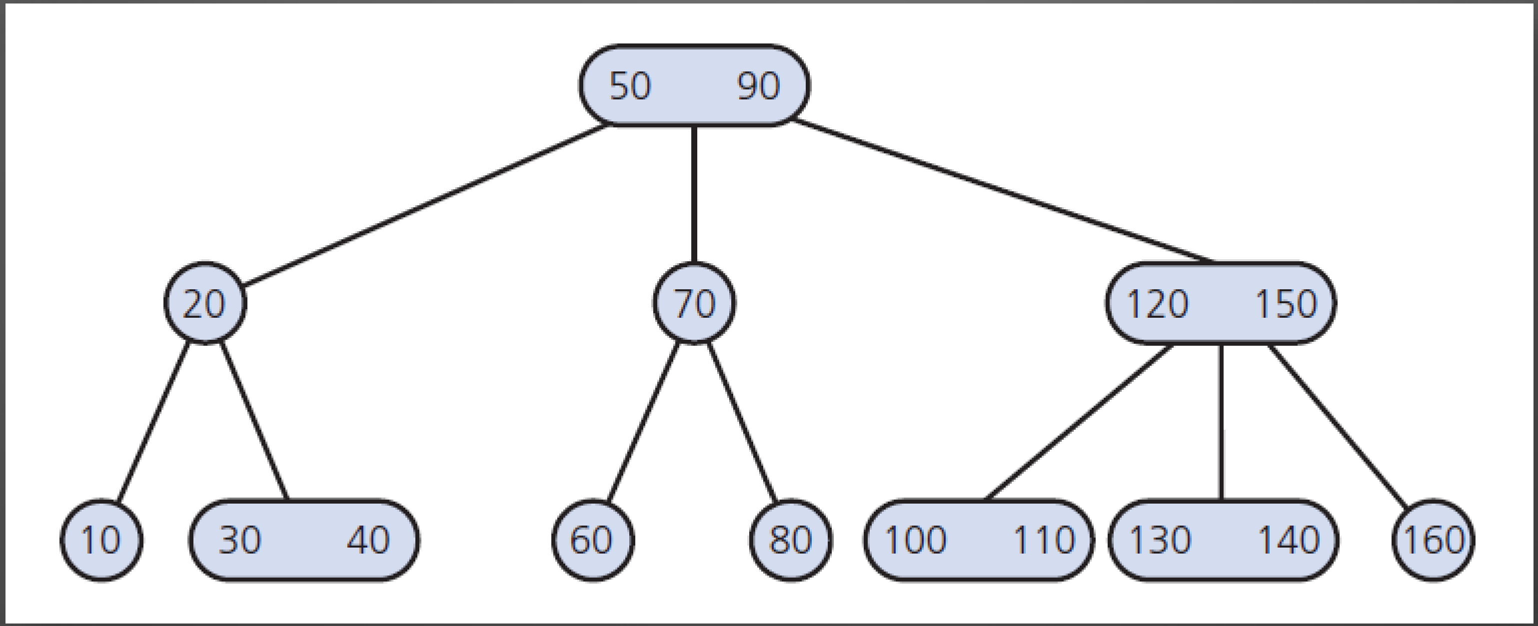
E is items < A

F is items > A and < B

G is items > B



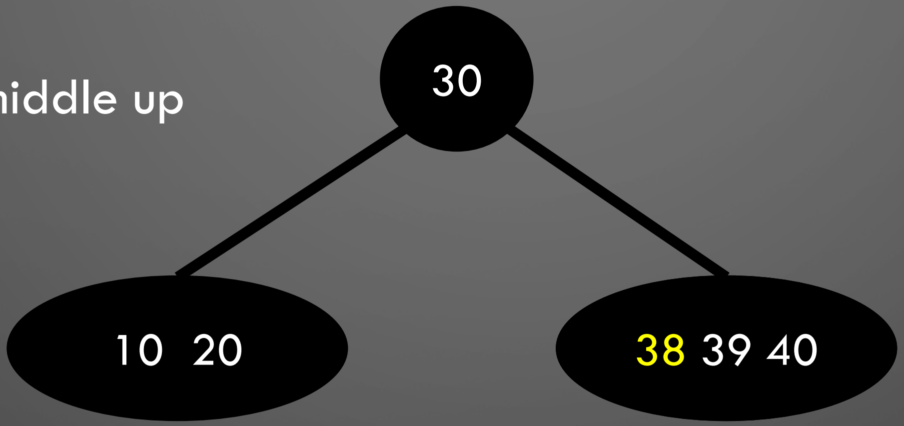
Searching a 2-3 Tree



- Compare search complexity of a 2-3 and shortest binary search tree
  - Complexity is?
- A binary search tree with *n* nodes cannot be shorter than $\log_2(n + 1)$
- A 2-3 tree with n nodes cannot be taller than $\log_2(n + 1)$
- Node in a 2-3 tree has at most two data items
- Searching 2-3 tree is O(log *n*)

Adding data into a 2-3 Tree

- Find the location as you would in a BST
- Add 38
  - Doesn't fit…
  - Split and move middle up

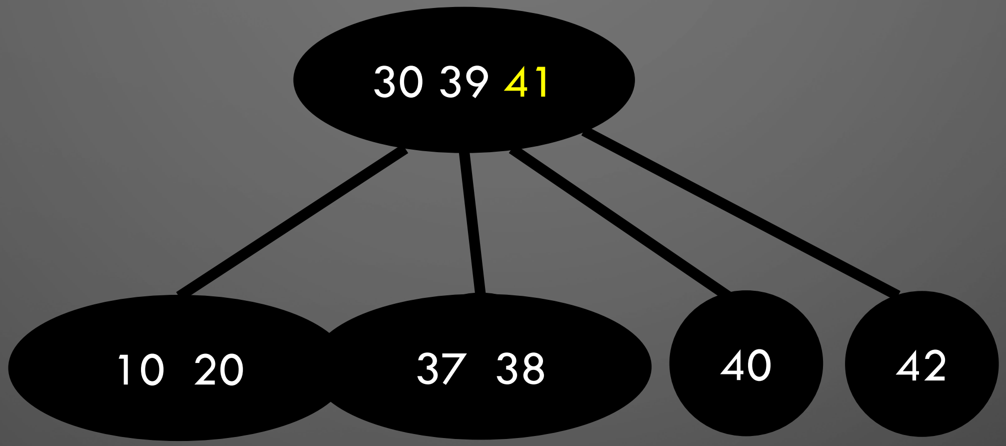


Adding data into a 2-3 Tree

- Find the location as you would in a BST
- Add 38
  - Doesn't fit…
  - Split and move middle up
- Add 37
- Add 42
- Add 41
  - Doesn't fit…
  - Split and move the middle up…

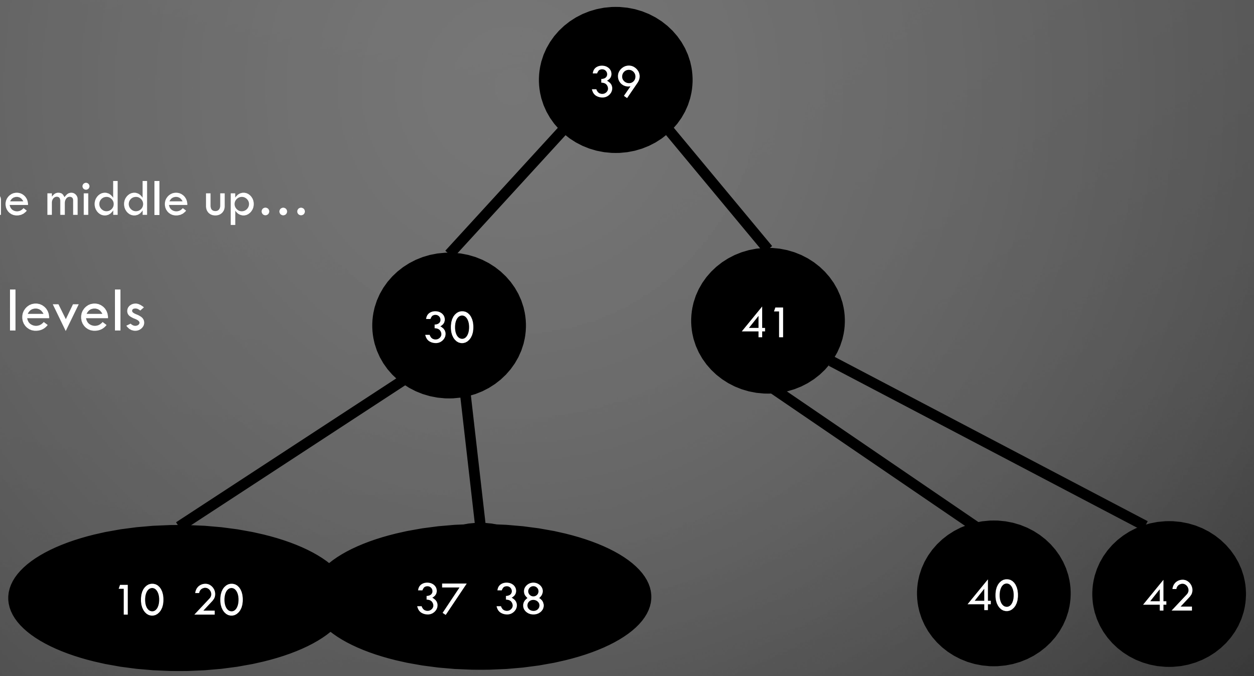
Adding data into a 2-3 Tree

- Add 41
  - Doesn't fit…
  - Split and move the middle up…



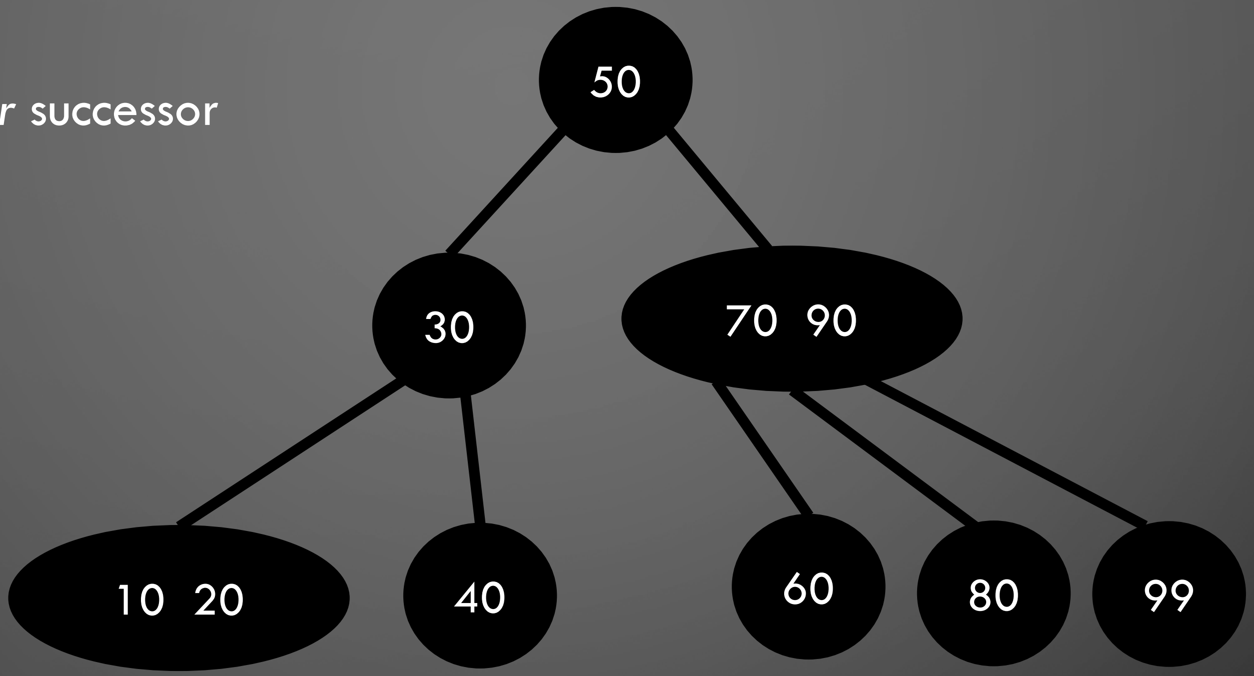
Adding data into a 2-3 Tree

- Add 41
  - Doesn't fit…
  - Split and move the middle up…
- Split root into two levels



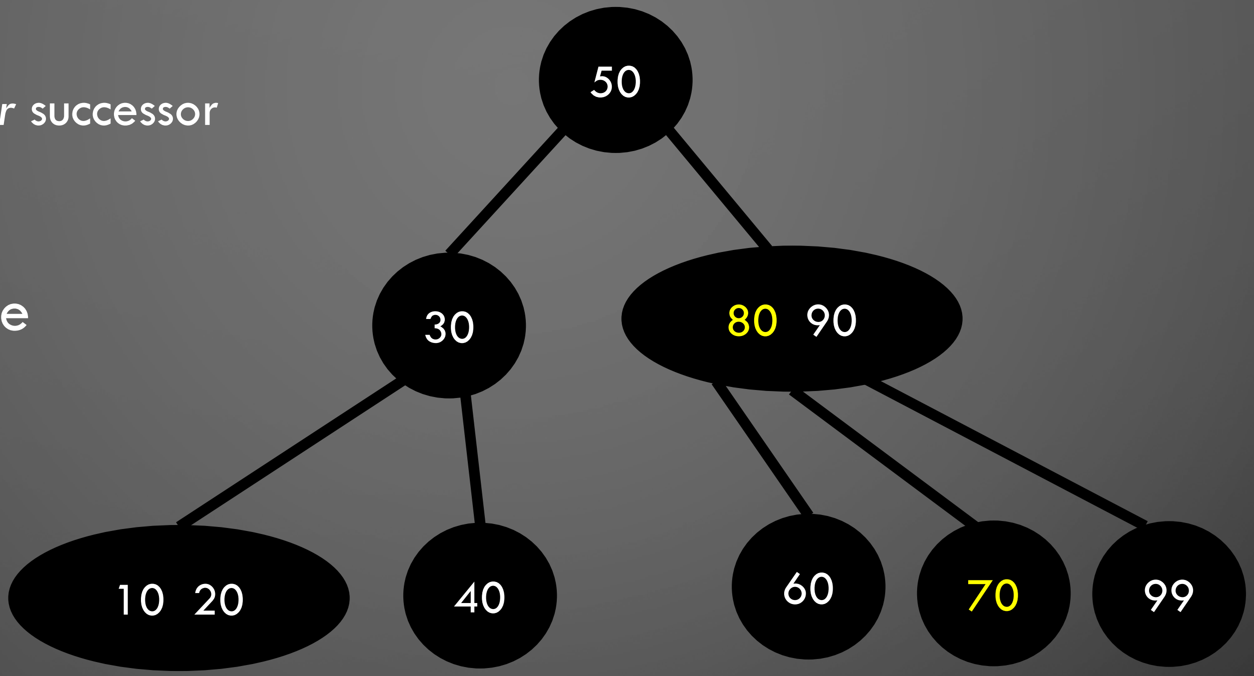
Removing data

- Remove 70
  - Swap with *inorder* successor



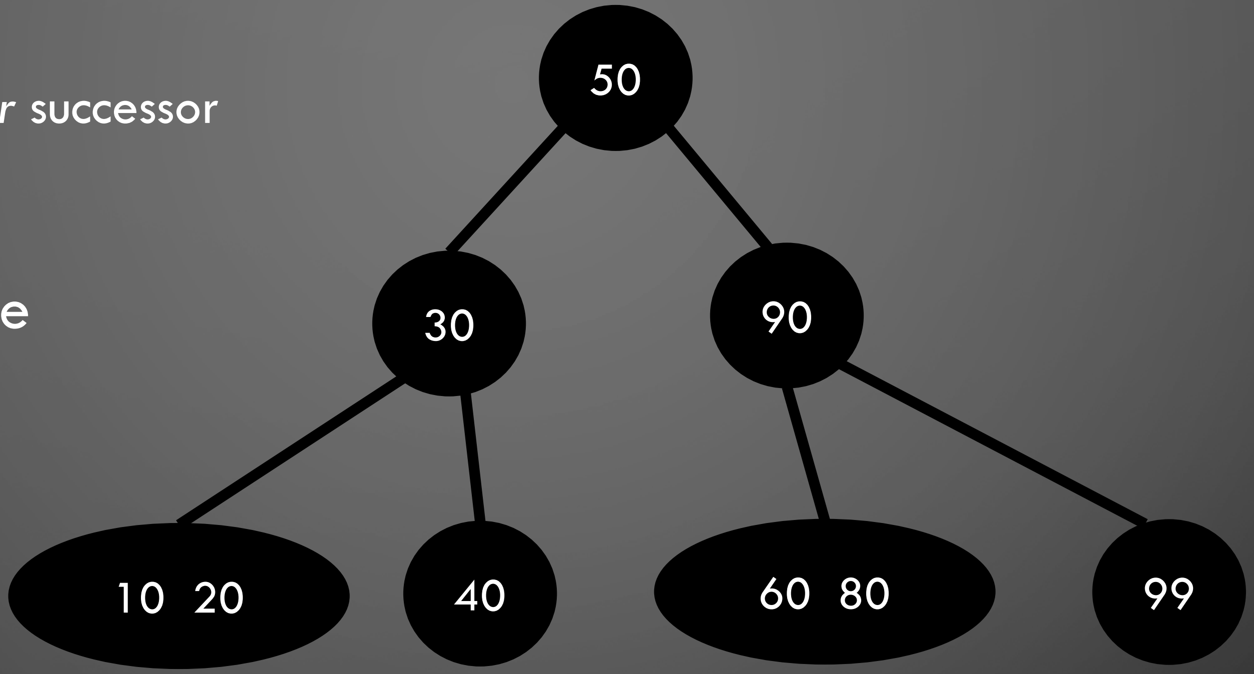
Removing data

- Remove 70
  - Swap with *inorder* successor
  - Remove new 70
- Now we have some bookkeeping
- Move 80 down



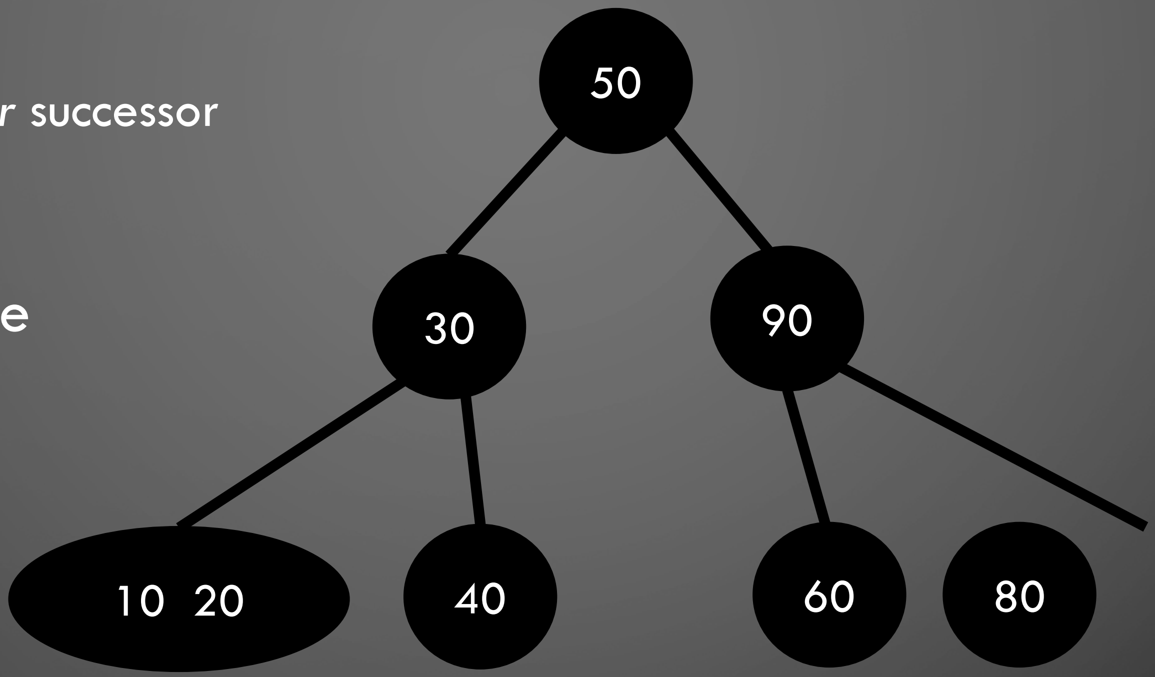
Removing data

- Remove 70
  - Swap with *inorder* successor
  - Remove new 70
- Now we have some bookkeeping
- Move 80 down
- Remove 99
- Sort of like rotate right



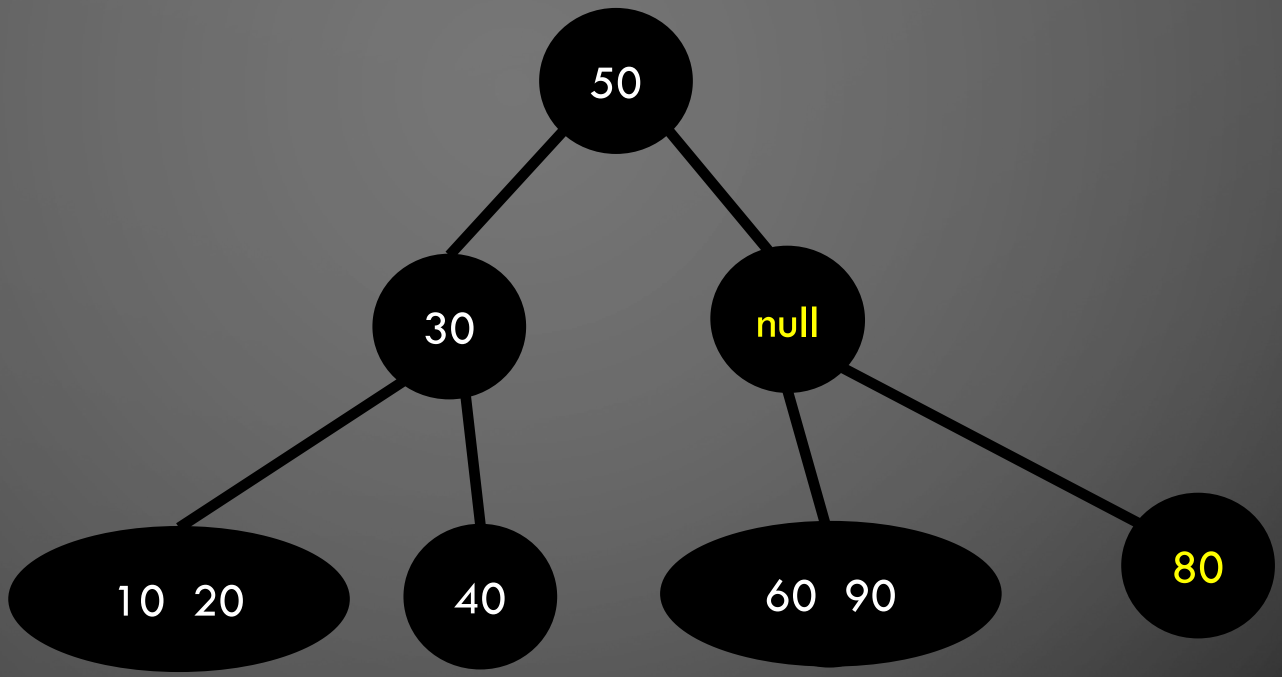
Removing data

- Remove 70
  - Swap with *inorder* successor
  - Remove new 70
- Now we have some bookkeeping
- Move 80 down
- Remove 99
- Sort of like rotate right
- Delete 80



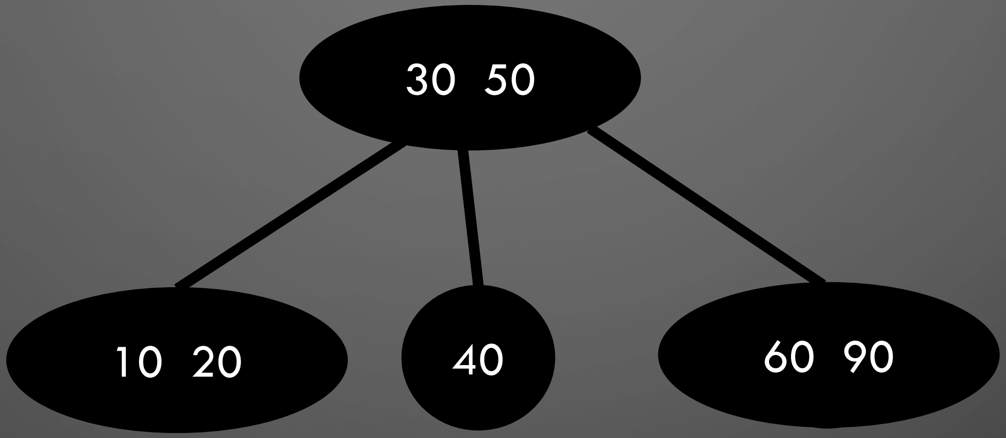
Removing data

- Delete 80
- Swap 80 and 90
- Remove 80
- Move 90 down



Removing data

- Delete 80
- Swap 80 and 90
- Remove 80
- Move 90 down
- Move 50 down
  - Adopt empty node's child



See Fig. 19-23 in the textbook for a summary of all the possible situations during removing an item

Red-Black Trees

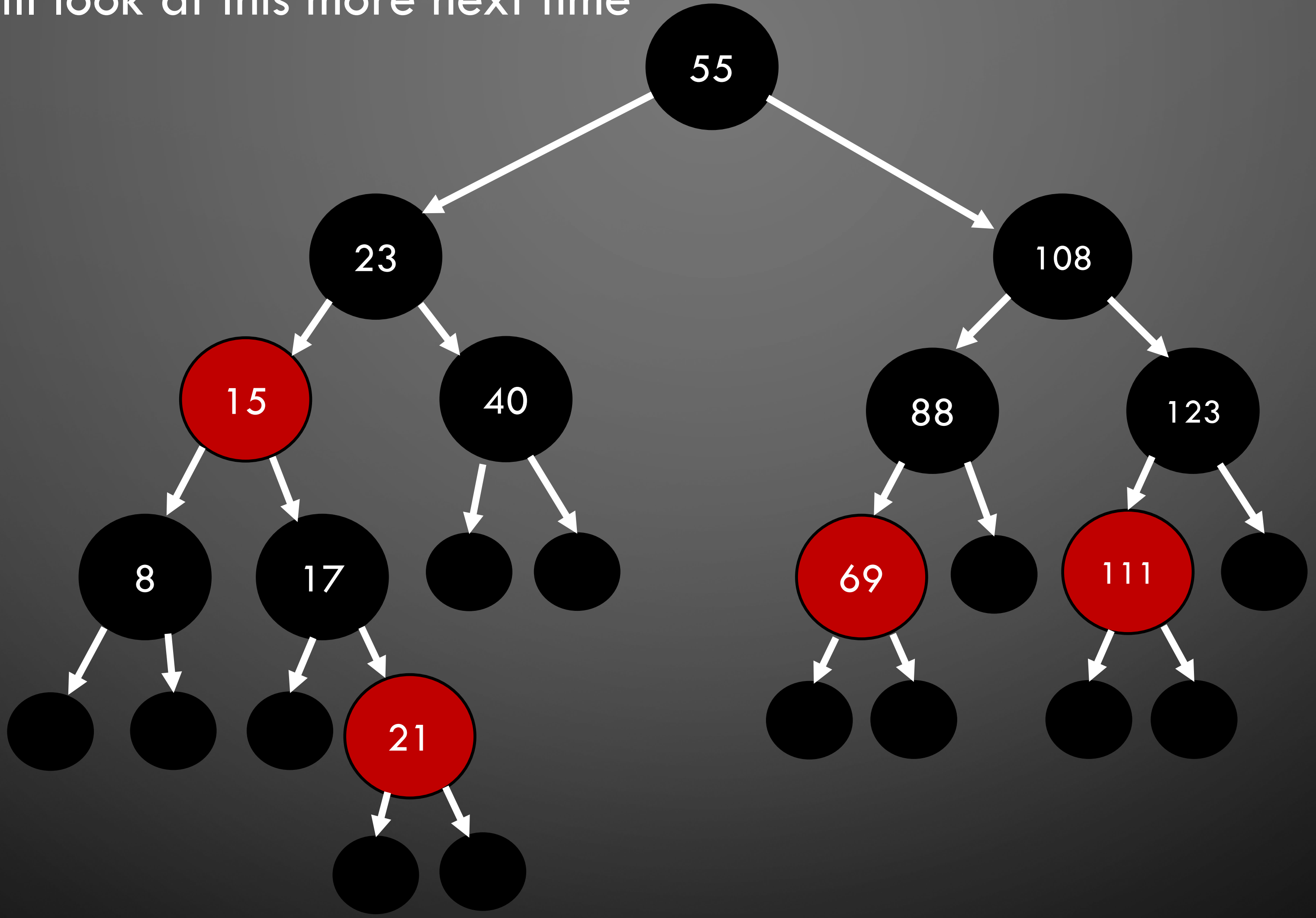
- Every node has a color either red or black.
- Root of tree and leaf are always black.
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.

<u>Consequence</u>

- The maximum depth of the red-black tree T with n nodes is at most twice the minimum depth.
- $\text{Depth}(T) \leq 2 \log_2(n+1)$

Red-Black Trees example

- We will look at this more next time



Assignment/Homework

- Reading pp. 614-625
- ICE 9 due on Friday.
- Homework 7 due on Friday.
- Homework 8 is released.
- ICE 10 is released.