Week 7 - Friday

COMP 2100

Last time

- What did we talk about last time?
- 2-3 trees and red-black tree practice

Questions?

Project 2

Infix to Postfix Converter

AVL Trees

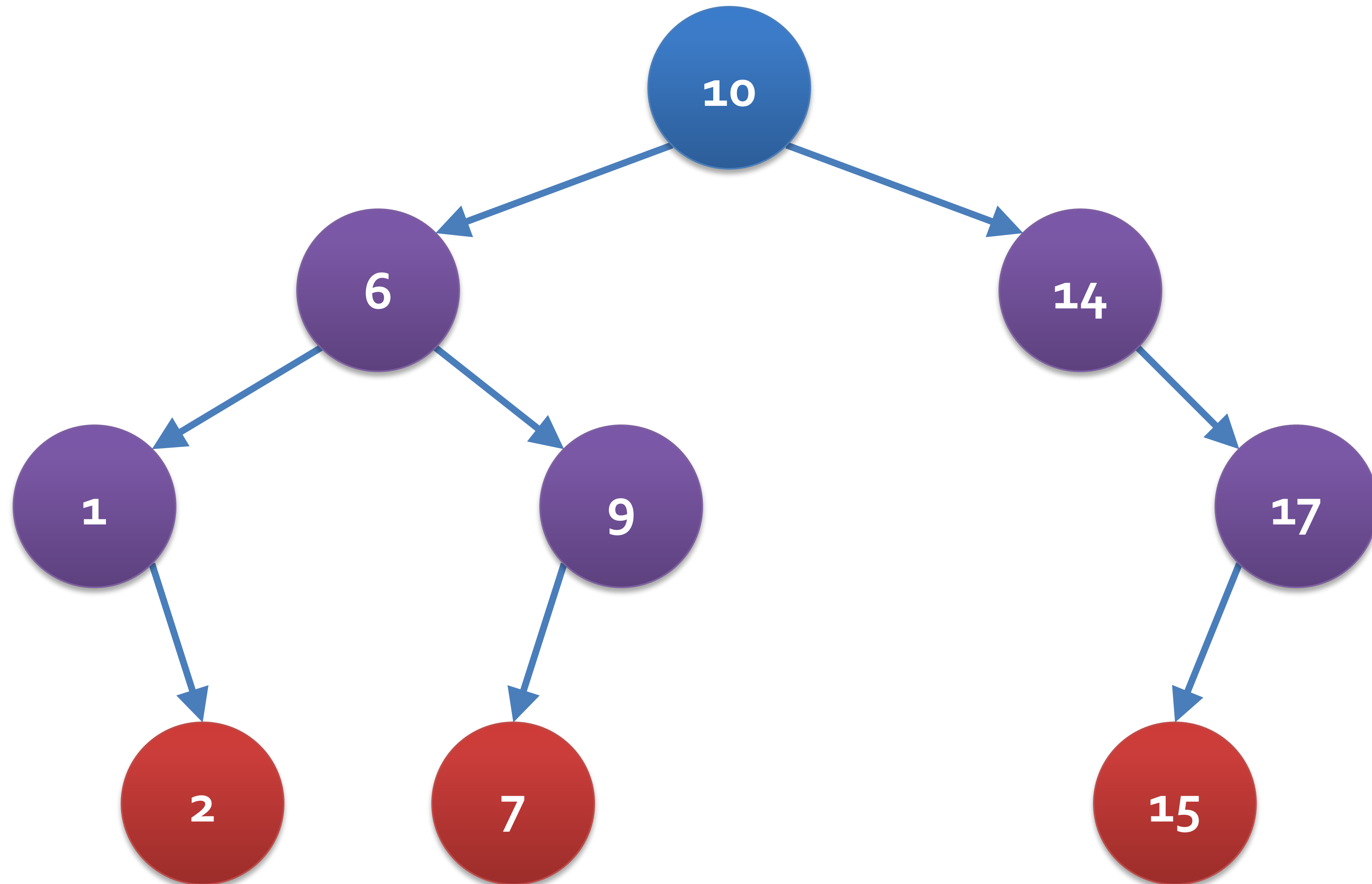
AVL trees

- Another approach to balancing is the **AVL tree**
  - Named for its inventors G.M. Adelson-Velskii and E.M. Landis
  - Invented in 1962
- An AVL tree is better balanced than a red-black tree, but it takes more work to keep such a good balance
  - It's faster for finds (because of the better balance)
  - It's slower for inserts and deletes
  - Like a red-black tree, all operations are $\Theta(\log \boldsymbol{n})$

AVL definition

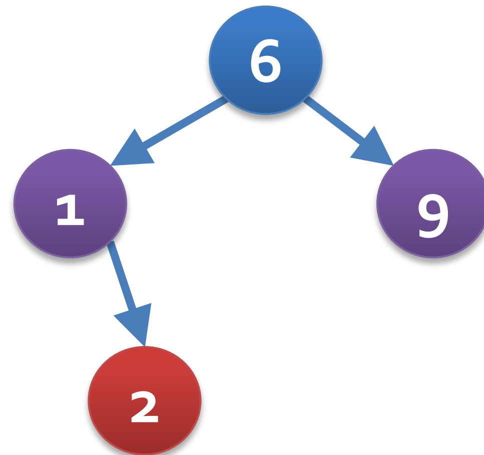
- An AVL tree is a binary search tree where
  - The left and right subtrees of the root have heights that differ by at most one
  - The left and right subtrees are also AVL trees

AVL tree?



Balance factor

- The balance factor of a tree (or subtree) is the height of its right subtree minus the height of its left
- In an AVL tree, the balance factor of **every** subtree is -1, 0, or +1



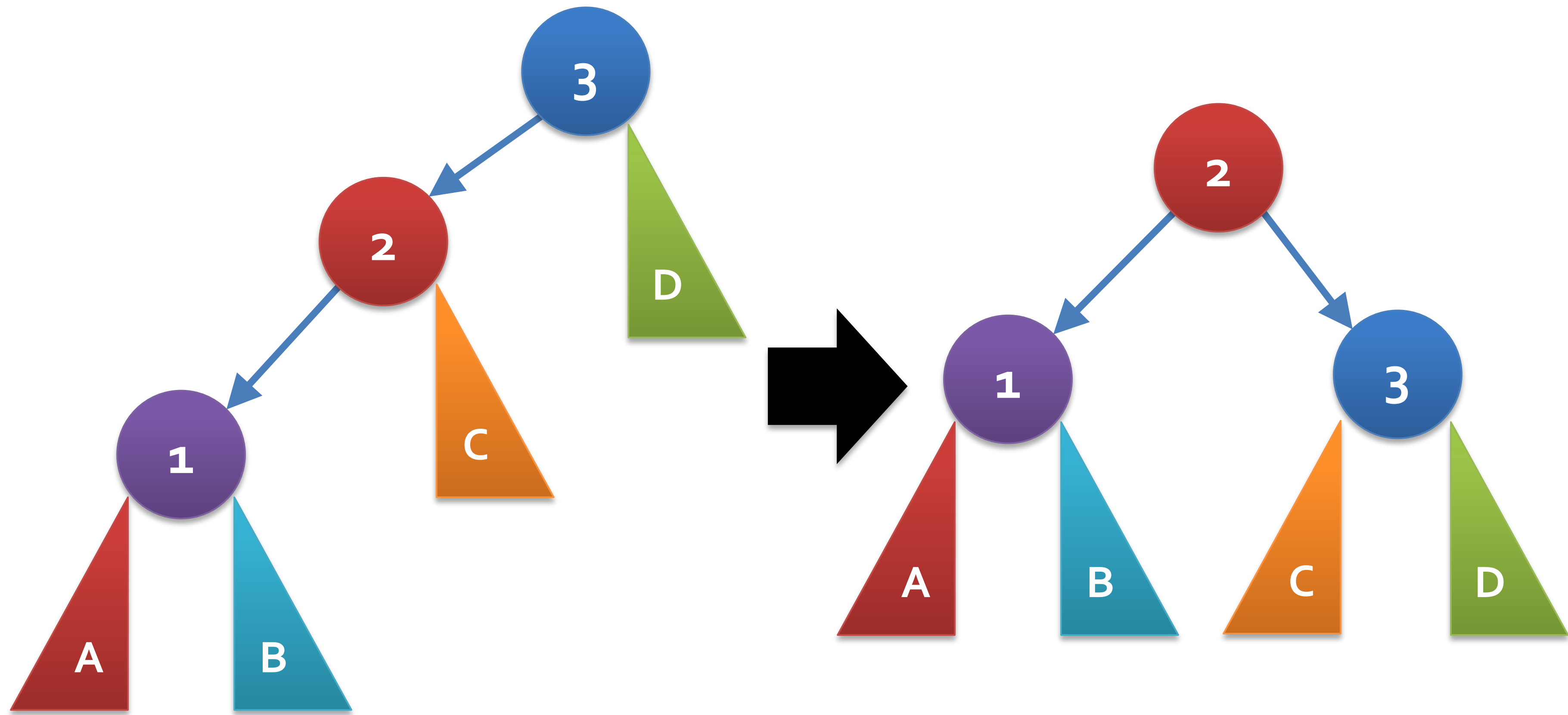
- What's the balance factor of this tree?

How do we build an AVL tree?

- Carefully.
- Every time we do an add, we have to make sure that the tree is still balanced
- There are 4 cases

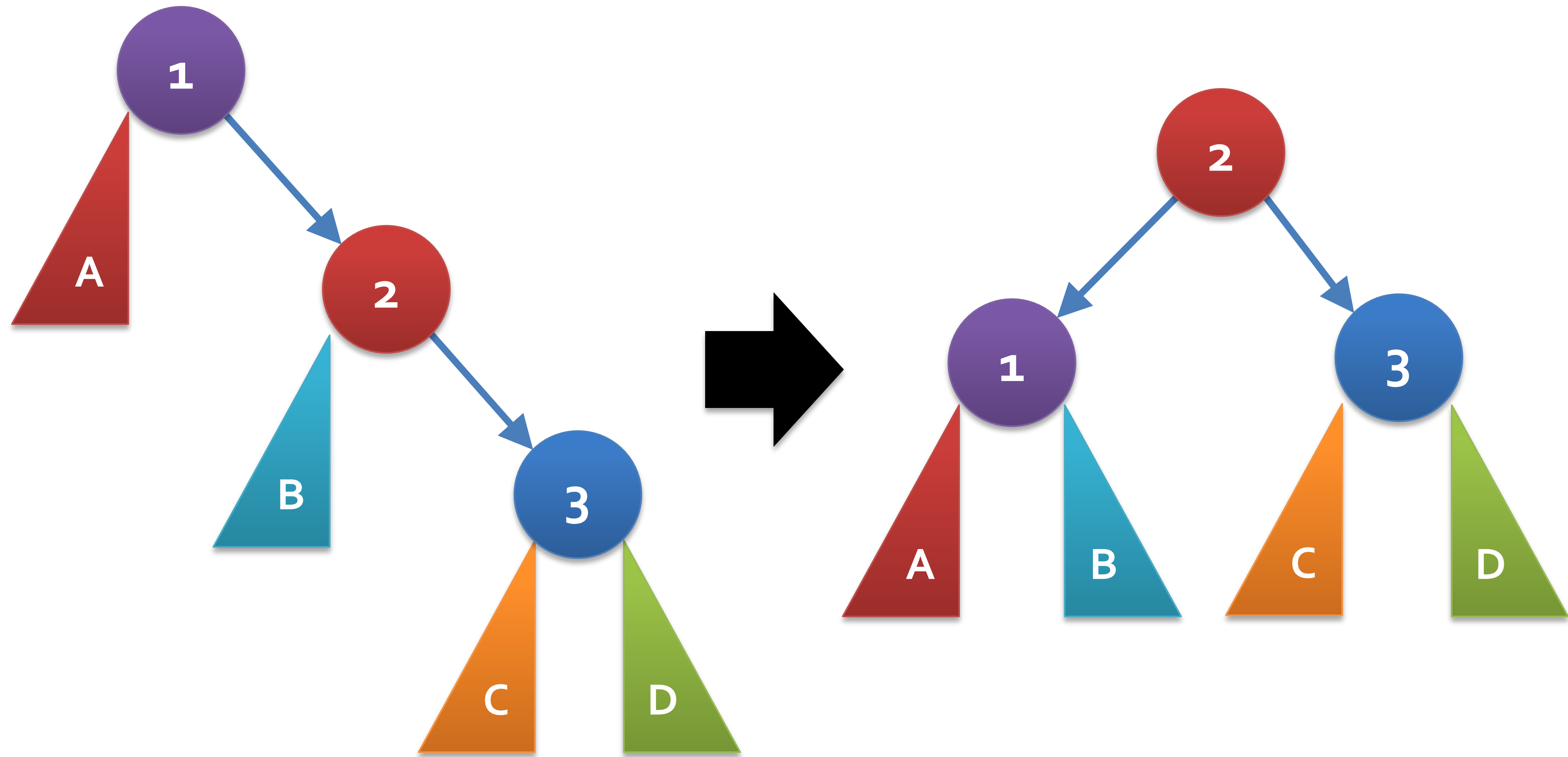
1. Left Left
2. Right Right
3. Left Right
4. Right Left

Left Left



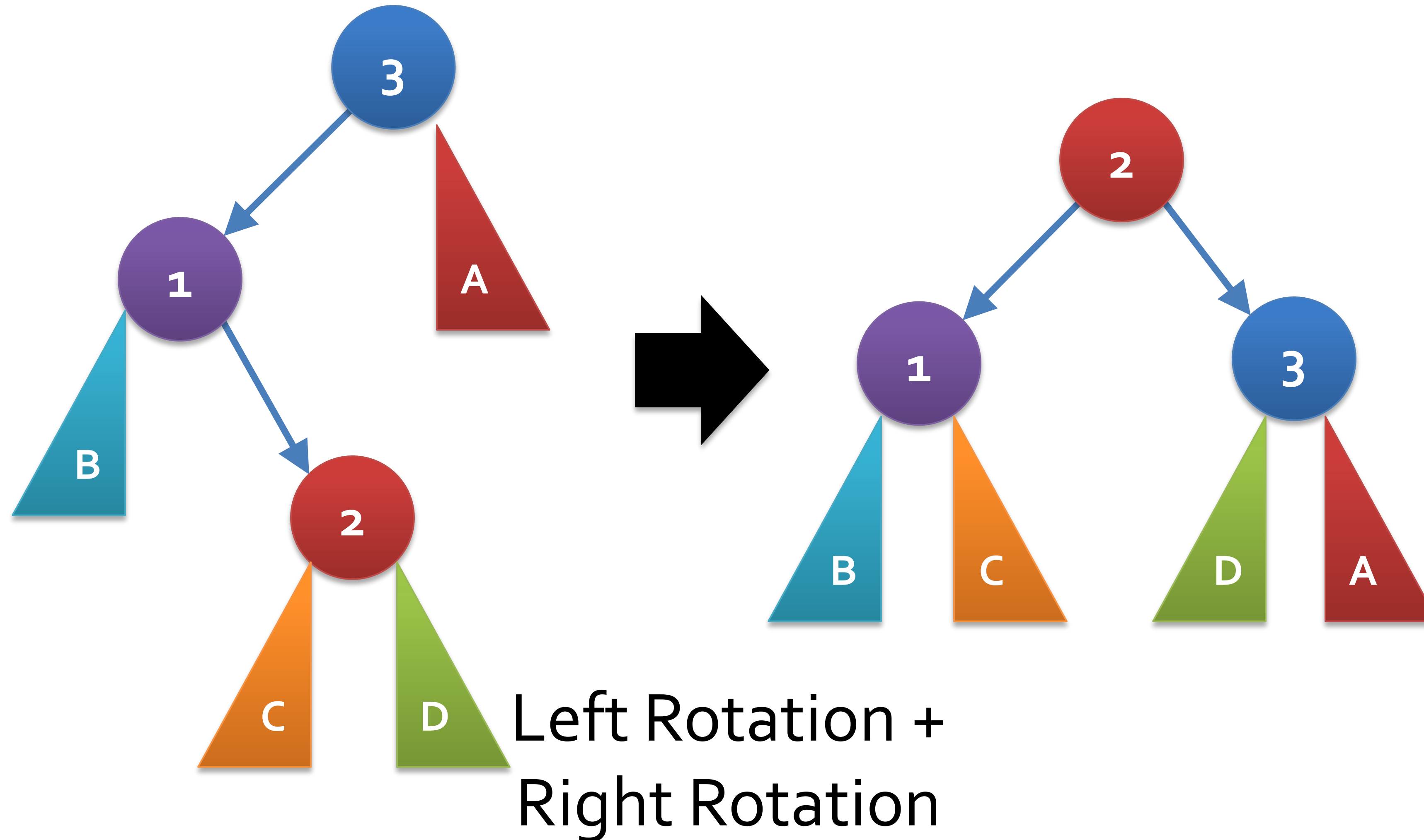
Right Rotation

Right Right

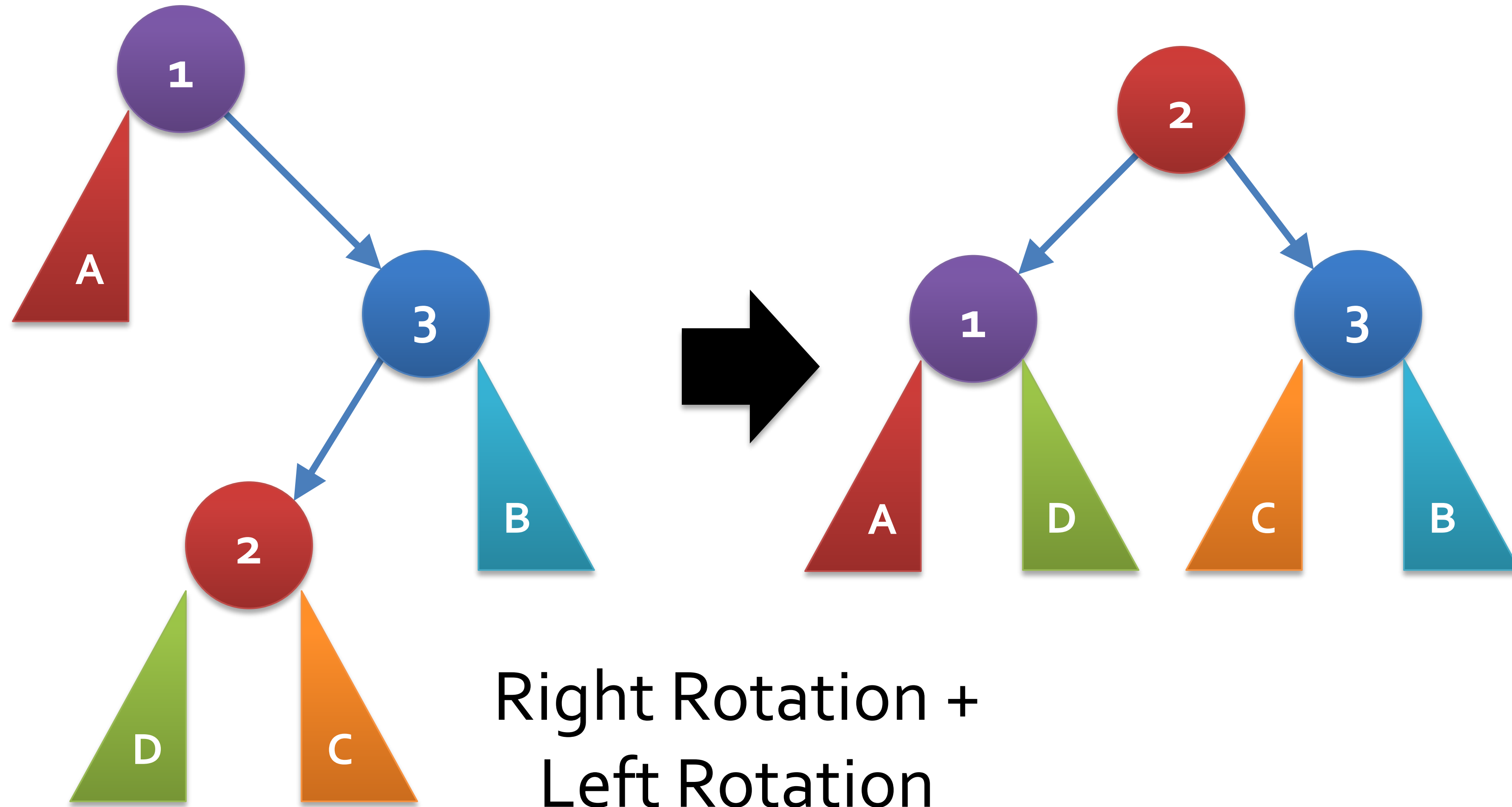


Left Rotation

Left Right



Right Left



Let's make an AVL tree!

- Let's just add these numbers in order:

  1, 2, 3, 4, 5, 6, 7, 8, 9, 10

- What about these?

  7, 24, 92, 32, 2, 57, 67, 84, 66, 75

Balancing a Tree by Construction

How to make a balanced tree

- What if we knew ahead of time which keys we were going to put into a tree?
- How could we make sure the tree is balanced?
- Answer:
  - Sort the keys
  - Recursively add the keys and values such that the tree stays balanced

Balanced insertion

- Write a recursive method that adds a sorted array of data such that the tree stays balanced
- Assume that an **`put(int key, Object value)`** method exists
  - It adds according to the normal BST insertion
- Use the usual convention that **`start`** is the beginning of a range and **`end`** is the location after the last legal element in the range

```
public void balance(int[] keys, Object[]
  values, int start, int end)
```

DSW algorithm

- Takes a potentially unbalanced tree and turns it into a balanced tree
- Step 1
  - Turn the tree into a degenerate tree (backbone or vine) by right rotating any nodes with left children
- Step 2
  - Turn the degenerate tree into a balanced tree by doing sequences of left rotations starting at the root
- The analysis is not obvious, but it takes **O**($\boldsymbol{n}$) total rotations

Which method is best?

- How much time does it take to insert ***n*** items with:
  - Red-black or AVL tree
  - Balanced insertion method
  - Unbalanced insertion + DSW rebalance
- How much space does it take to insert ***n*** items with:
  - Red-black or AVL tree
  - Balanced insertion method
  - Unbalanced insertion + DSW rebalance

Hash Tables

Recall: Symbol table ADT

- We can define a symbol table ADT with a few essential operations:
  - put(Key key, Value value)
    - Put the key-value pair into the table
  - get(Key key):
    - Retrieve the value associated with key
  - delete(Key key)
    - Remove the value associated with key
  - contains(Key key)
    - See if the table contains a key
  - isEmpty()
  - size()
- It's also useful to be able to iterate over all keys

Unordered symbol table

- We have been talking a lot about trees and other ways to keep *ordered* symbol tables
- Ordered symbol tables are great, but we may not always need that ordering
- Keeping an unordered symbol table might allow us to improve our running time

Hash tables: motivation

- Balanced binary search trees give us:
  - $\Theta(\log \boldsymbol{n})$ time to find a key
  - $\Theta(\log \boldsymbol{n})$ time to do insertions and deletions

- Can we do better?
- What about:
  - $\Theta(1)$ time to find a key
  - $\Theta(1)$ to do an insertion or a deletion

Hash tables: theory

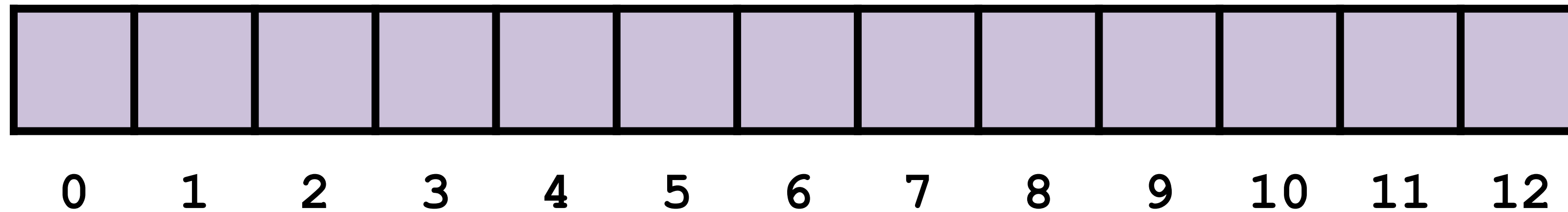
- We make a huge array, so big that we'll have more spaces in the array than we expect data values
- We use a **hashing function** that maps keys to indexes in the array
- Using the hashing function, we know where to put each key but also where to look for a particular key

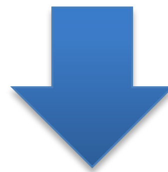
Hash table: example

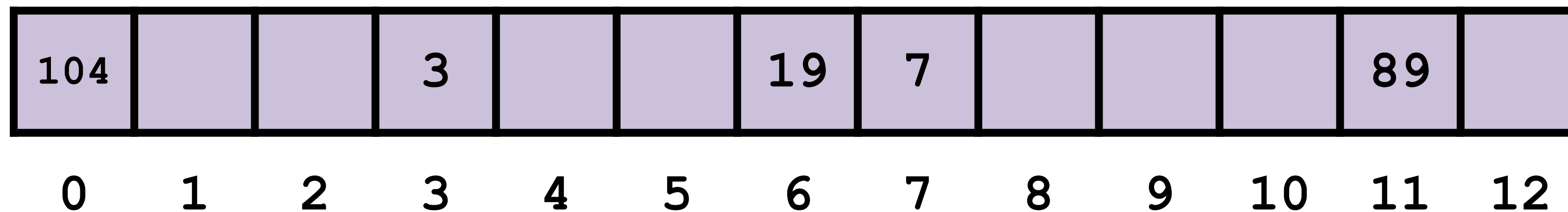
- Let's make a hash table to store integer keys
- Our hash table will be 13 elements long
- Our hashing function will be simply modding each value by 13

Hash table: example

- Insert these keys: 3, 19, 7, 104, 89







Hash table: example

- Find these keys:

104			3			19	7				89	
0	1	2	3	4	5	6	7	8	9	10	11	12

- 19
  - YES!
- 88
  - NO!
- 16
  - NO!

Hash table: issues

- We are using a hash table for a space/time tradeoff
- Lots of space means we can get down to $\Theta(1)$
- How much space do we need?
- How do we pick a good hashing function?
- What happens if two values **collide** (map to the same location)

Upcoming

Next time…

- Collisions
- Chaining implementation of hash tables
- No class on Monday or Tuesday

Reminders

- Finish Project 2
  - **Due tonight by midnight!**
- Keep reading Section 3.4
- Have a good break!