Lecture 09: Binary Search Trees

CSE 373: Data Structures and Algorithms

Warm Up

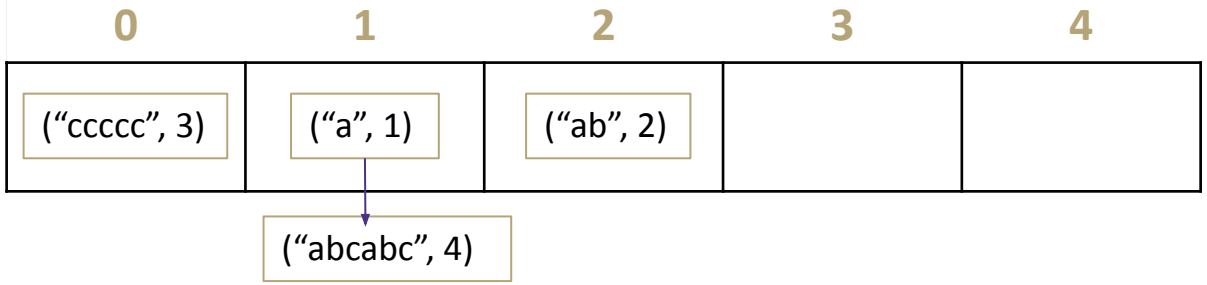
Slido Event #2357722
https://app.sli.do/event/bQQDgvkWCLE521e1Nurgfc

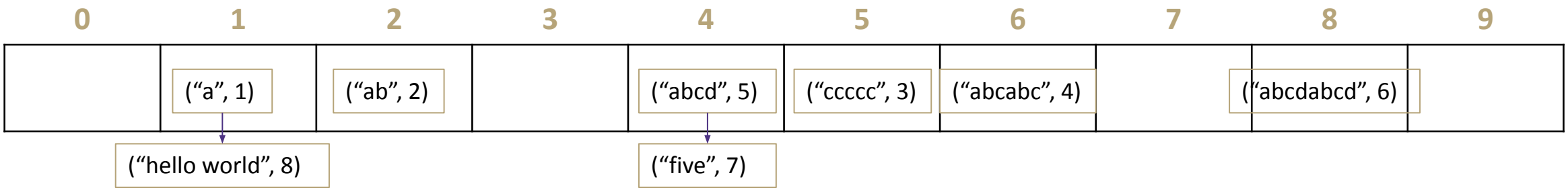
Consider a `StringDictionary` using separate chaining with an initial capacity of 5, but will resize **before** λ=1 by doubling the current capacity. Assume our buckets are implemented using a `LinkedList`. Use the following hash function:

```
public int hashCode(String input) {
   return input.length();
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

("a", 1) ("ab", 2) ("ccccc", 3) ("abcabc", 4) ("abcd", 5) ("abcdabcd", 6) ("five", 7) ("hello world", 8)





Announcements

- Project 2 out now, due Wednesday 4/26
- Project 1 turn in closes on Saturday
- Exercise 2 due on Monday at 11:59PM

Java's hashCode function

All Java Objects **must** include a hashCode function:

```
public int hashCode();
```

From official Oracle Java documentation:

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(`java.lang.Object`) method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

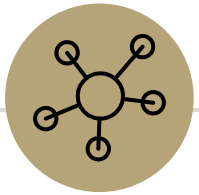
Java and Hash Functions

- Object class includes default functionality:
  - `int equals(Object other)`
  - `int hashCode()`

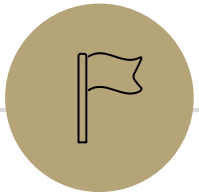
- If you want to implement your own hashCode you should:
  - Override BOTH `hashCode()` and `equals()`

- If a.equals(b) is true then a.hashCode() == b.hashCode() **MUST** also be true
  - This is how Java knows to replace the value associated with the key or to add a new key to the bucket

- That requirement is part of the Object interface
  - Other people's code will assume you've followed this rule.
- Java's HashMap (and HashSet) will assume you follow these rules and conventions for your custom objects if you want to use your custom objects as keys.



Questions?



Linear Probing

Quadratic Probing

Double Hashing

Summary

Handling Collisions

Solution 2: Open Addressing

Resolves collisions by choosing a different location to store a value if natural choice is already full.

Type 1: Linear Probing

If there is a collision, keep checking the next element until we find an open spot.

```
int findFinalLocation(Key s) {
    int naturalHash = this.hashCode(s);
    int index = naturalHash % array.length;
    while (index in use) {
        i++;
        index = (naturalHash + i) % array.length;
    }
    return index;
}
```

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions

`1, 5, 11, 7, 12, 17, 6, 25`

0	1	2	3	4	5	6	7	8	9
	1	11	12		5	6	7	17	25

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions

`38, 19, 8, 109, 10`

Problem:

- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering

When probing causes long chains of occupied slots within a hash table

Runtime

When is runtime good?

When we hit an empty slot

- (or an empty slot is a very short distance away)

When is runtime bad?

When we hit a "cluster"

Maximum Load Factor?

λ at most 1.0

When do we resize the array?

$\lambda \approx ½$ is a good rule of thumb

Can we do better?

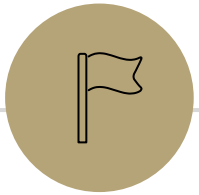
Clusters are caused by picking new space near the natural index

Solution 2: Open Addressing (still)

Type 2: Quadratic Probing

Instead of checking i past the original location, check i^2 from the original location

```
int findFinalLocation(Key s)
    int naturalHash = this.hashCode(s);
    int index = naturalHash % array.length;
    while (index in use) {
        i++;
        index = (naturalHash + i*i) % array.length;
    }
    return index;
```



Linear Probing

Quadratic Probing

Double Hashing

Summary

Quadratic Probing

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions

`89, 18, 49, 58, 79, 27`

0	1	2	3	4	5	6	7	8	9
49		58	79				27	18	89

(49 % 10 + 0 * 0) % 10 = 9
(49 % 10 + 1 * 1) % 10 = 0

(58 % 10 + 0 * 0) % 10 = 8
(58 % 10 + 1 * 1) % 10 = 9
(58 % 10 + 2 * 2) % 10 = 2

(79 % 10 + 0 * 0) % 10 = 9
(79 % 10 + 1 * 1) % 10 = 0
(79 % 10 + 2 * 2) % 10 = 3

Now try to insert 9.

Uh-oh

Problems:

If $\lambda \geq$ ½ we might never find an empty spot
    Infinite loop!
Can still get clusters

Quadratic Probing

There were empty spots. What Gives?

Quadratic probing is not guaranteed to check every possible spot in the hash table

The following is true:

> If the table size is a prime number *p*, then the first *p*/2 probes check distinct indices.

Notice we have to assume p is prime to get that guarantee

Secondary Clustering

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions
19, 39, 29, 9

0	1	2	3	4	5	6	7	8	9
39			29					9	19

Secondary Clustering

When using quadratic probing, you sometimes need to probe the same sequence of table cells, not necessarily next to one another

Probing

h(k) = the natural hash

h'(k, i) = resulting hash after probing

i = iteration of the probe

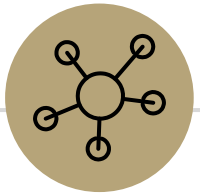
T = table size

Linear Probing:

$h'(k, i) = (h(k) + i) \% T$

Quadratic Probing

$h'(k, i) = (h(k) + i^2) \% T$



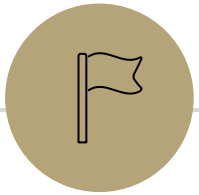
Questions?

Topics Covered:

- Writing good hash functions
- Open addressing to resolve collisions:
  - Linear probing
  - Quadratic probing

Linear Probing

Quadratic Probing



Double Hashing

Summary

Double Hashing

Probing causes us to check the same indices over and over– can we check different ones instead?

Use a second hash function!

h'(k, i) = (h(k) + i * g(k)) % T <- Most effective if g(k) returns value relatively prime to table size

```
int findFinalLocation(Key s)
     int naturalHash = this.getHash(s);
     int index = natrualHash % TableSize;
   while (index in use) {
      i++;
      index = (naturalHash + i*jumpHash(s)) % TableSize;
   }
   return index;
```

Resizing: Open Addressing

How do we resize? Same as separate chaining

- Remake the table
- Evaluate the hash function over again
- Re-insert

When to resize?

- Depending on our load factor λ AND our probing strategy
  - If λ = 1, `put` with a new key fails for linear probing
  - If λ > 1/2, `put` with a new key **might** fail for quadratic probing, even with a prime `tableSize`
    - And it might fail earlier with a non-prime size
  - If λ = 1, `put` with a new key fails for double hashing
    - And it might fail earlier if the second hash isn't relatively prime with the `tableSize`

Summary

1. Pick a hash function to:
   - Avoid collisions
   - Uniformly distribute data
   - Reduce hash computational costs

2. Pick a collision strategy
   - Chaining
   - LinkedList
   - AVL Tree
   - Probing
   - Linear
   - Quadratic
   - Double Hashing

No clustering
Potentially more "compact" (λ can be higher)

Managing clustering can be tricky
Less compact (keep λ < ½)
Array lookups tend to be a constant factor faster than traversing pointers

Summary

Separate Chaining

- Easy to implement
- Running times $O(1+\lambda)$ in practice

Open Addressing

- Uses less memory (usually)
- Various schemes:
  - Linear Probing – easiest, but lots of clusters
  - Quadratic Probing – middle ground, but need to be more careful about λ
  - Double Hashing – need a whole new hash function, but low chance of clustering

Which one you use depends on your application and what you're worried about

Java's HashMap Implementation

- default array capacity is 16
  - *Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high - Javadocs*
- resizes at load factor 0.75
  - *As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost - Javadocs*
- uses separate-chaining collision resolution
  - Initially uses LinkedLists as the buckets
  - After 8 collisions across the table at the next resize the buckets will be created as balanced trees to reduce runtime of possible worst case scenario - Javarevisited

Other Hashing Applications

We use it for hash tables but there are lots of uses! Hashing is a really good way of taking arbitrary data and creating a succinct and unique summary of data.

Caching

- You've downloaded a large video file, You want to know if a new version is available, Rather than re-downloading the entire file, compare your file's hash value with the server's hash value.

File Verification / Error Checking

- Compare the hash of a file instead of the file itself
- Find similar substrings in a large collection of strings – detecting plagiarism

Cryptography

Hashing also "hides" the data by translating it, this can be used for security

- For password verification: Storing passwords in plaintext is insecure. So your passwords are stored as a hash
- Digital signatures

Fingerprinting

git hashes ("identification")

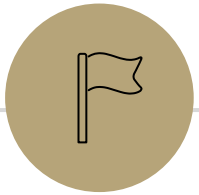
- That crazy number that is attached to each of your commits
- SHA-1 hash incorporates the contents of your change, the name of the files and the lines of the files you changes

Ad Tracking

- Track who has seen an ad if they saw it on a different device (if they saw it on their phone don't want to show it on their laptop)
- https://panopticlick.eff.org will show you what is being hashed about you

YouTube Content ID

- Do two files contain the same thing? Copyright infringement
- Change the files a bit!



Binary Search Trees

Binary Trees

A **tree** is a collection of nodes
- Each node has at most 1 parent and anywhere from 0 to 2 children
- pretty similar to node based structures we've seen before (linked-lists)

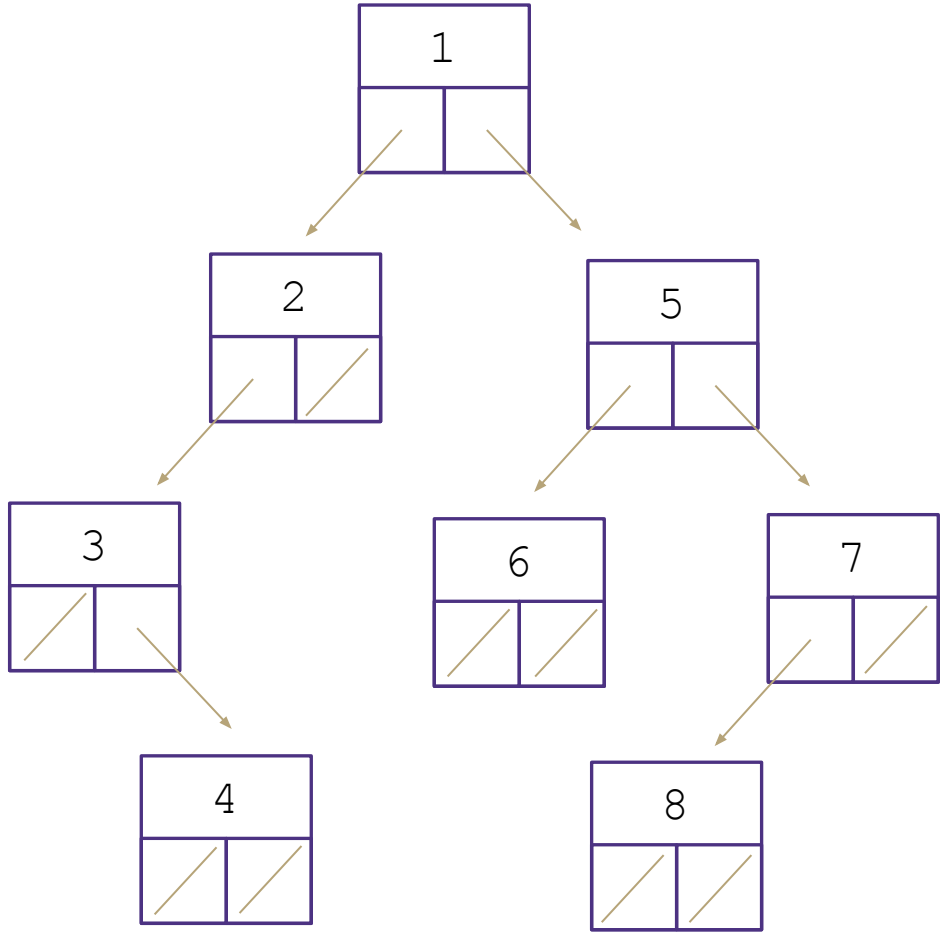
```
public class Node<K> {
    K data;
    Node<K> left;
    Node<K> right;
}
```

Root node: the single node with no parent, "top" of the tree. Often called the 'overallRoot'

Leaf node: a node with no children

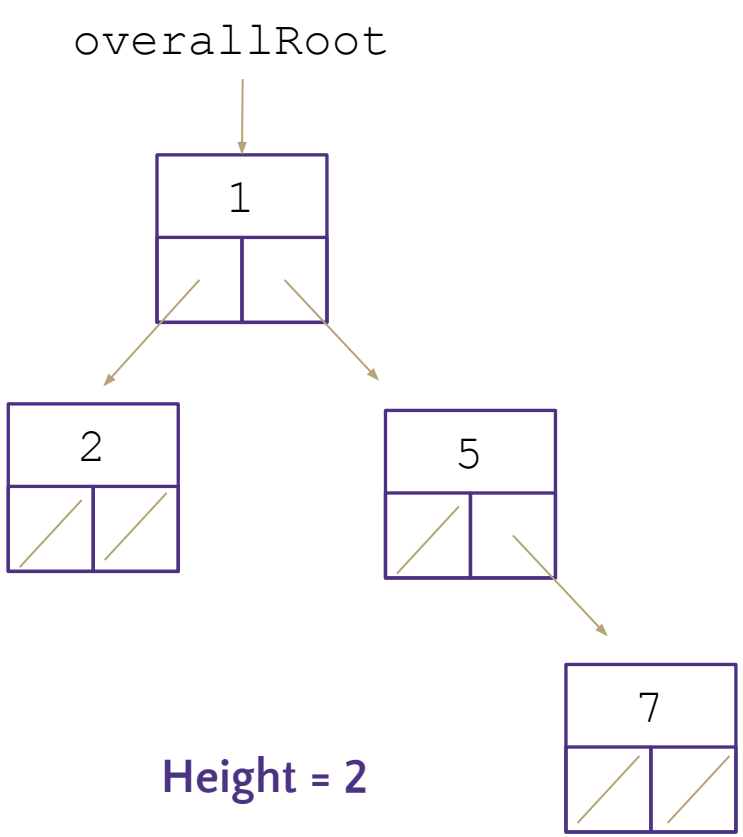
Subtree: a node and all it descendants

Height: the number of edges contained in the longest path from root node to some leaf node

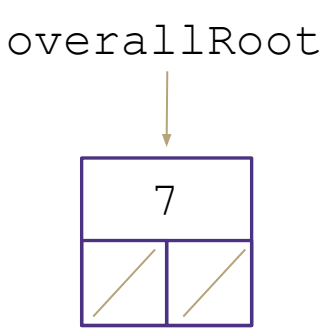


Tree Height

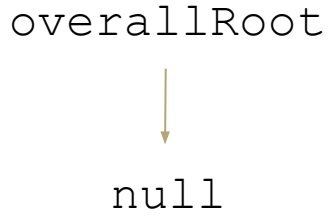
What is the height (the number of edges contained in the longest path from root node to some leaf node) of the following binary trees?



Height = 2



Height = 0



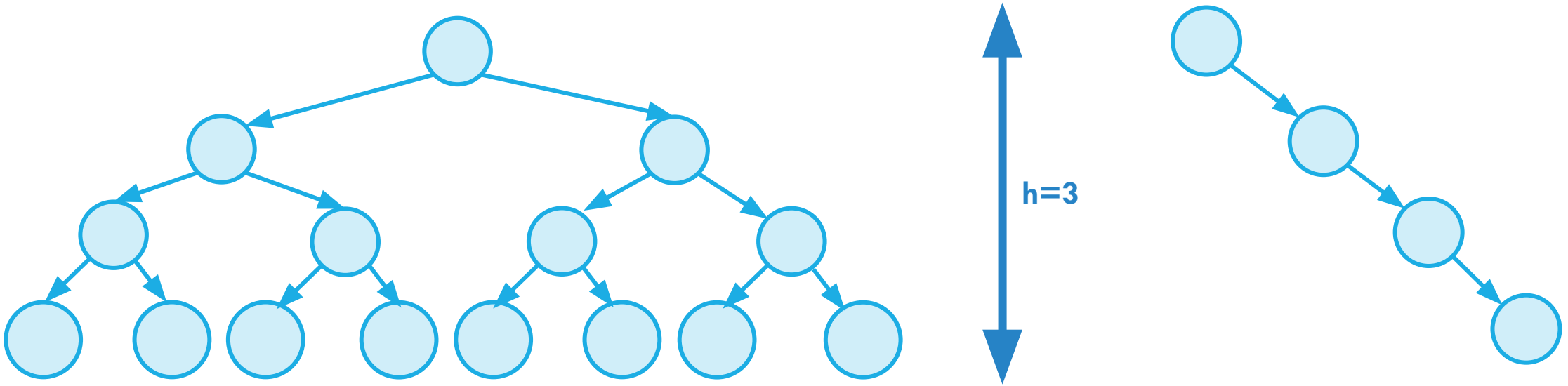
Height = -1 or NA

Other Useful Binary Tree Numbers

For a binary tree of height h:

Max number of leaves: 2^h
Max number of nodes: $2^{(h+1)} - 1$

Max number of leaves: 1
Max number of nodes: $h + 1$



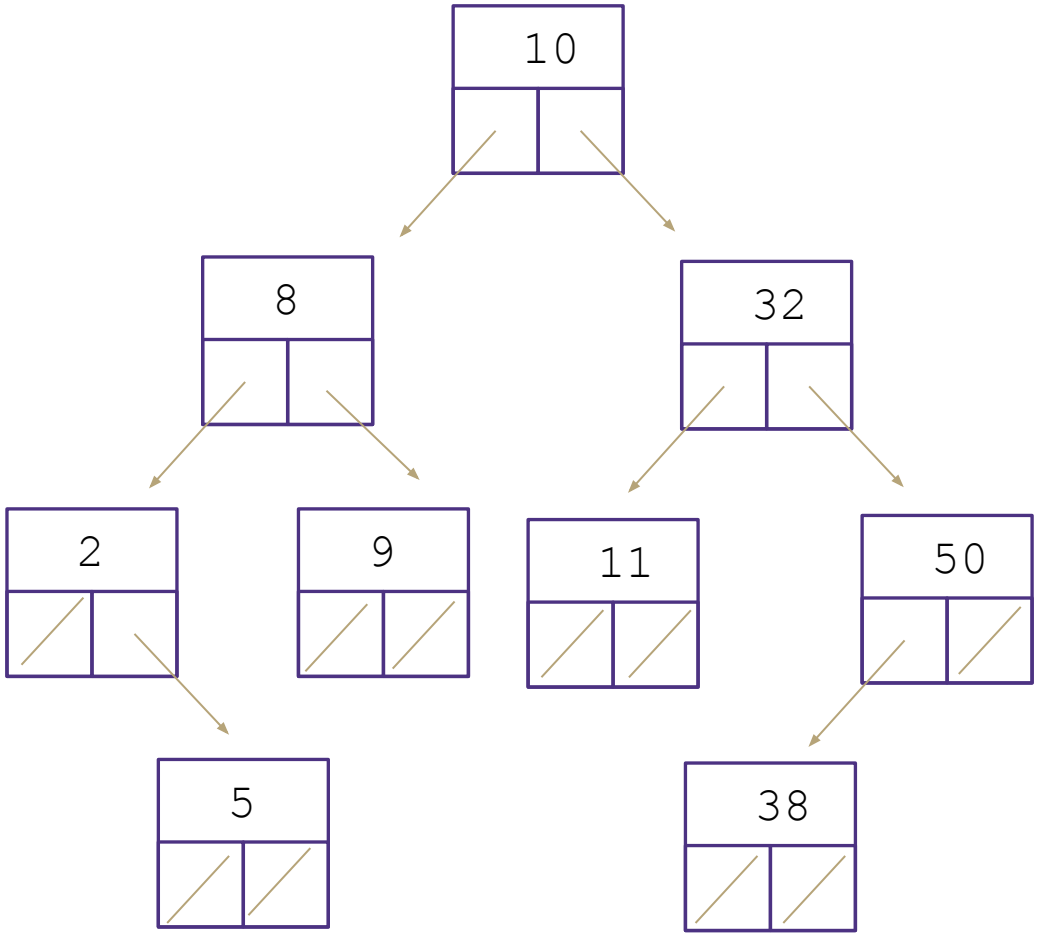
Binary Search Tree (BST)

Invariants (A.K.A. rules for your data structure)

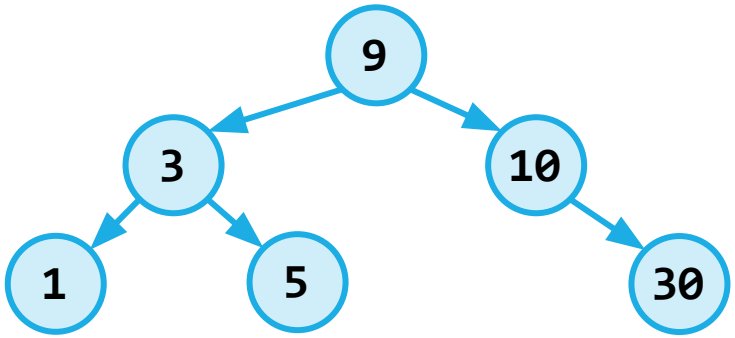
- Things that are always true. If they're always true, you can assume them so that you can write simpler and more efficient code.
- You can also check invariants at the ends/beginnings of your methods to ensure that your state is valid and that everything is working

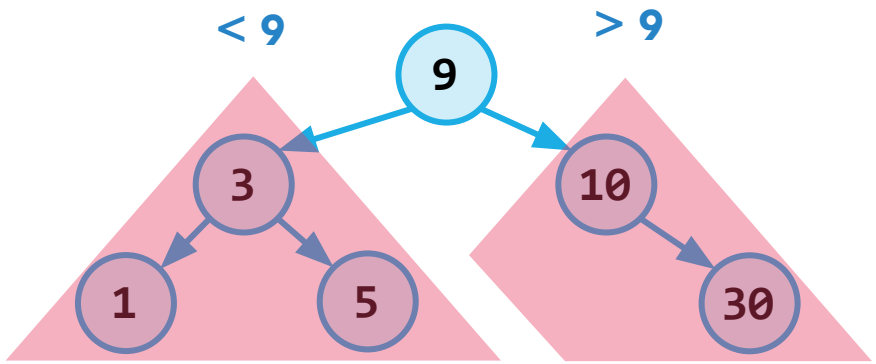
Binary Search Tree invariants:

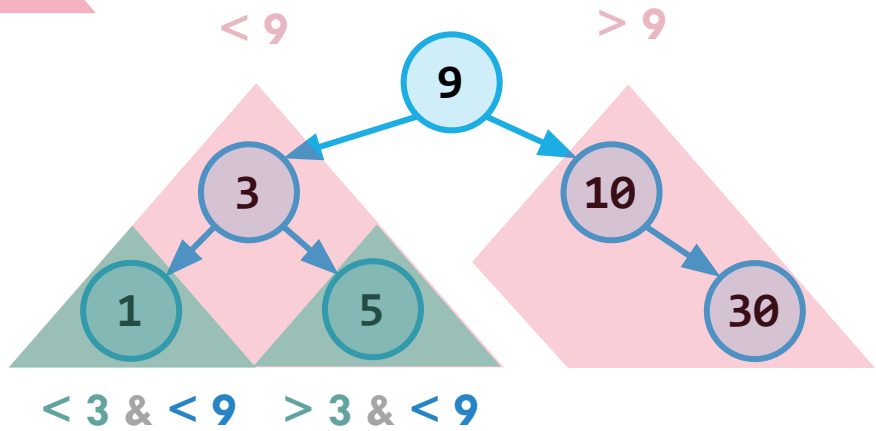
- For every node with key *k*:
  - The left subtree has only keys smaller than *k*
  - The right subtree has only keys greater than *k*



BST Ordering Applies *Recursively*





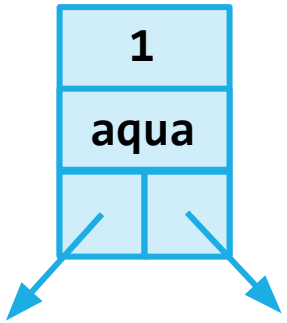


Aside Anything Can Be a Map

Want to make a tree implement the Map ADT?

- No problem – just add a value field to the nodes, so each node represents a key/value pair.

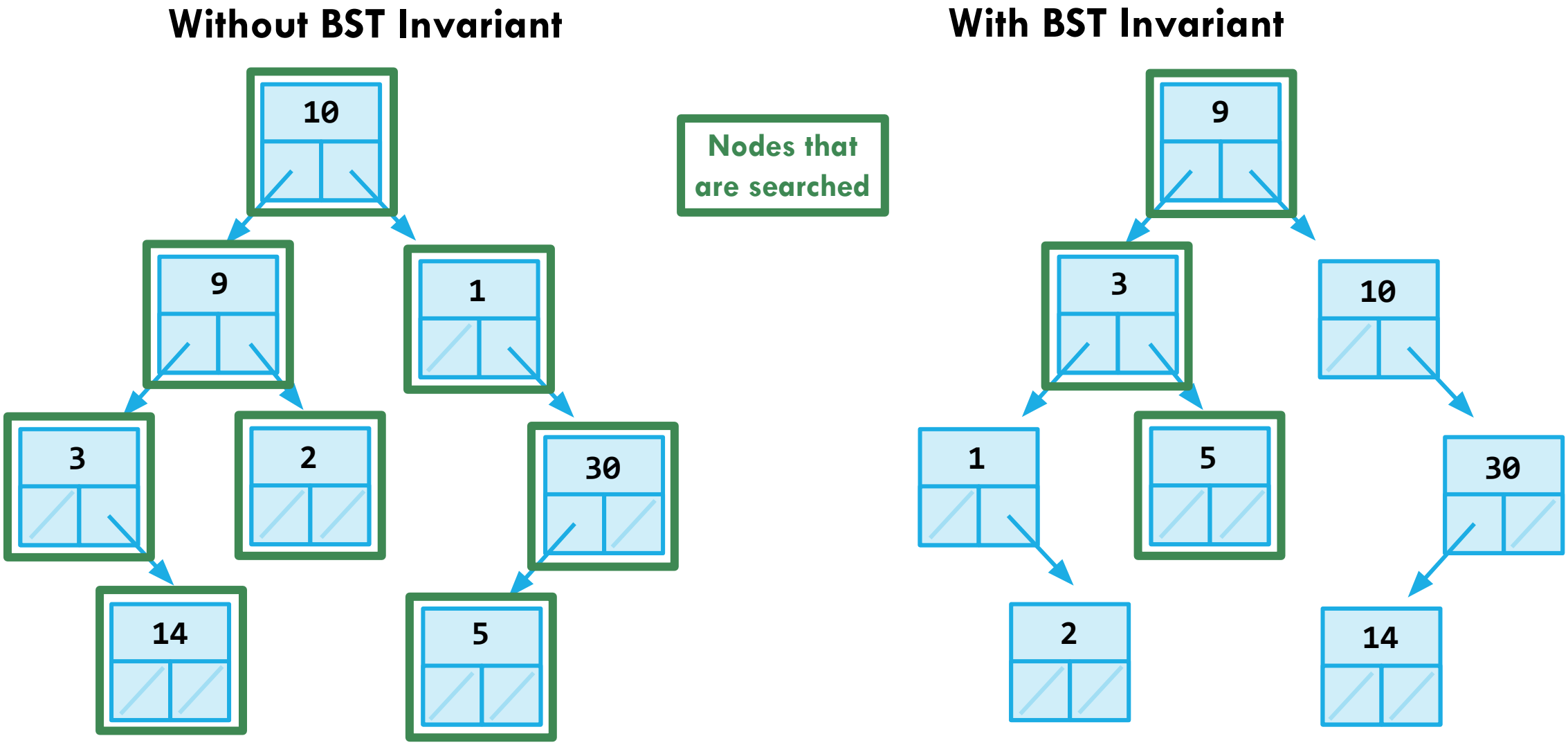
```
public class Node<K, V> {
    K key;
    V value;
    Node<K, V> left;
    Node<K, V> right;
}
```



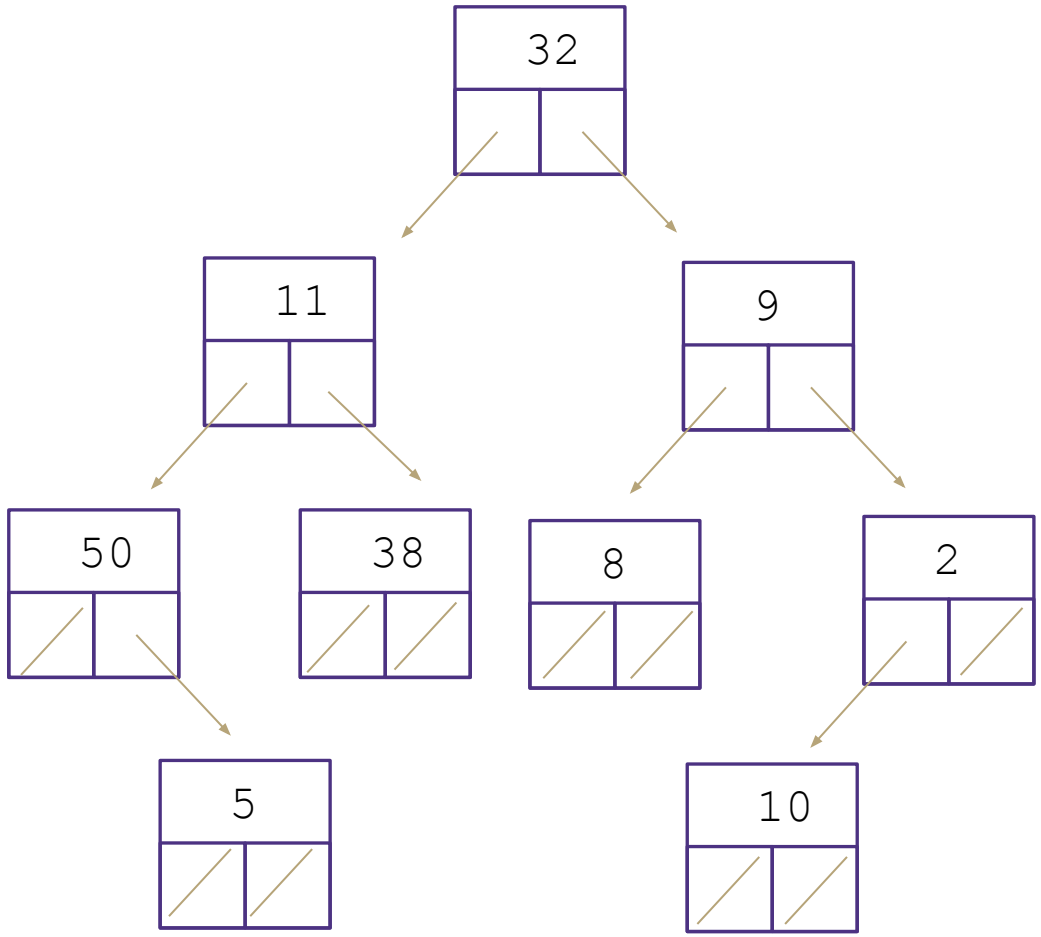
For simplicity, we'll just talk about the keys

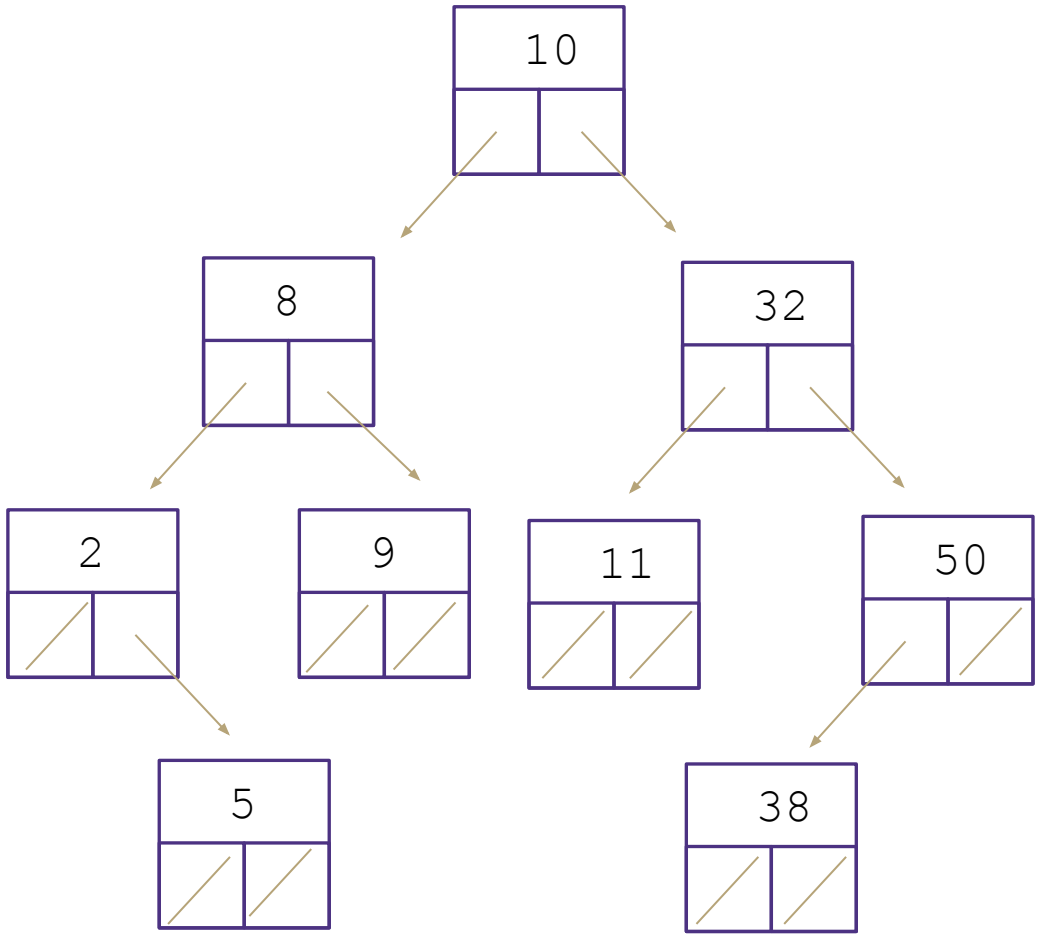
- Interactions between nodes are based off of keys (e.g. BST sorts by keys)
- In other words, keys determine where the nodes go

Binary Tree vs. BST: containsKey(5)



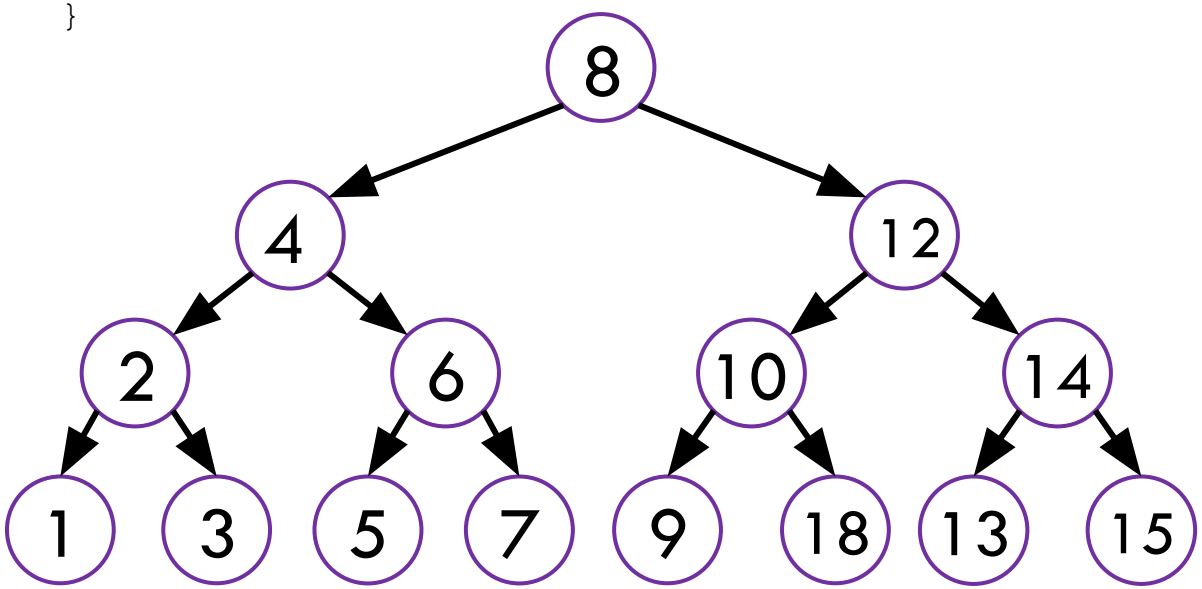
Binary Trees vs Binary Search Trees: containsKey(2)





Binary Trees vs Binary Search Trees: containsKey(2)

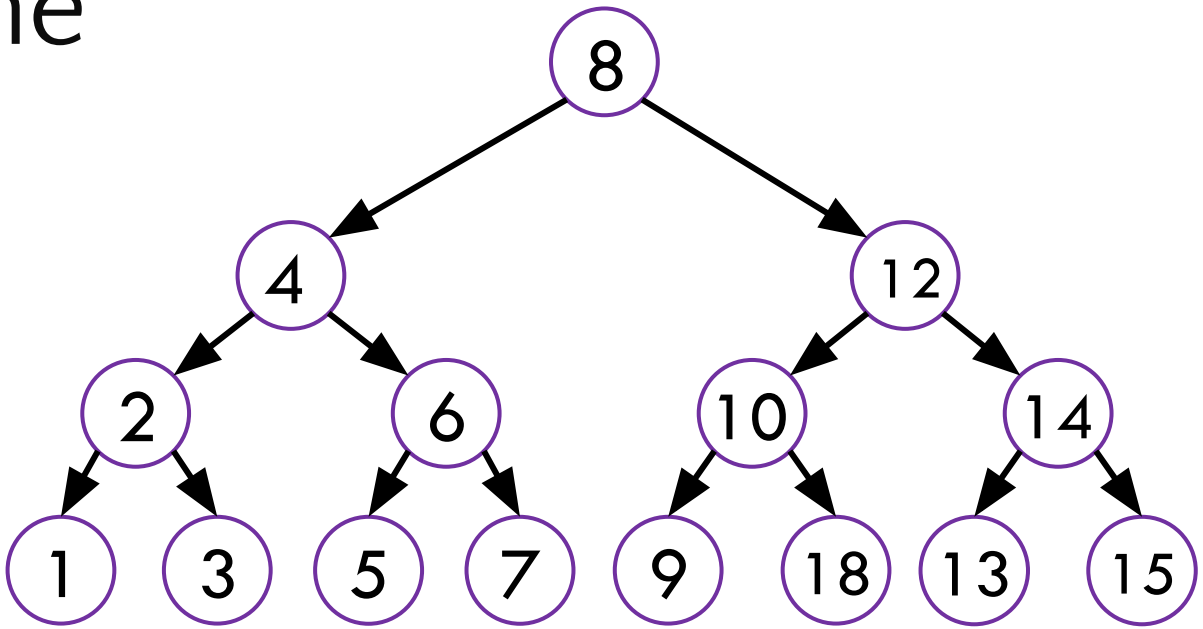
```
public boolean containsKeyBT(node, key) {
   if (node == null) {
      return false;
   } else if (node.key == key) {
      return true;
   } else {
      return containsKeyBT(node.left) ||
                     containsKeyBT(node.right);
   }
}
```




```
public boolean containsKeyBST(node, key) {
   if (node == null) {
      return false;
   } else if (node.key == key) {
      return true;
   } else {
      if (key <= node.key) {
         return containsKeyBST(node.left);
      } else {
         return containsKeyBST(node.right);
      }
   }
}
```

BST containsKey runtime

```
public boolean containsKeyBST(node, key) {
   if (node == null) {
      return false;
   } else if (node.key == key) {
      return true;
   } else {
      if (key <= node.key) {
         return containsKeyBST(node.left);
      } else {
         return containsKeyBST(node.right);
      }
   }
}
```



For the tree on the right, **what are some possible interesting cases (best/worst/other?) that could come up?** Consider what values of key could affect the runtime

- **best**: `containsKey(8)`, runtime will be O(1) since it will end immediately
- **worst**: `containsKey(-1)` since it has to traverse all the way down (other values will work for this)

`containsKey()` is a recursive method → recurrences!

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 \text{ if } n > 1 \\ 3 \quad \text{otherwise} \end{cases}$$

* if tree is balanced we eliminate half the nodes to search at each level ie n/2

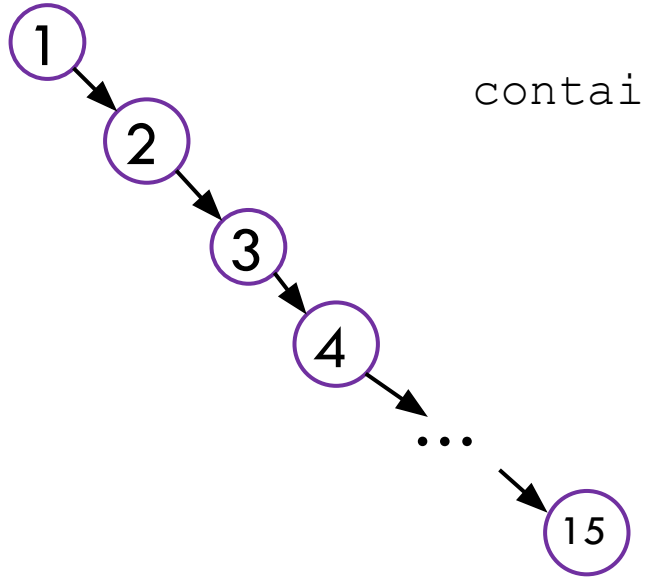
Is it possible to do worse than O(log n) 😈

We only considered changing the key parameter for that one particular BST in our last thought exercise, but what about if we consider the different possible arrangements of the BST as well?

Let's try to come up with a valid BST with the numbers 1 through 15 (same as previous tree) and key combination that result in a worse runtime for containsKey.

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \qquad\qquad \text{otherwise} \end{cases}$$

$$\mathrm{T(n)} = \Theta(n)$$



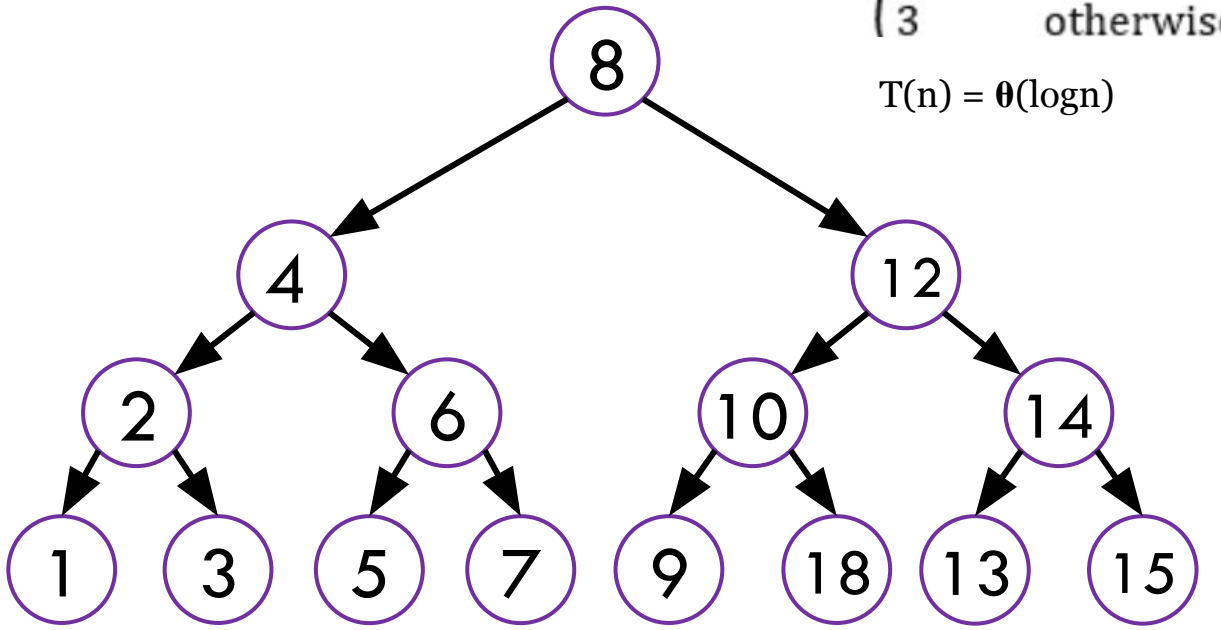
BST different states

Two different extreme states our BST could be in (there's in-between, but it's easiest to focus on the extremes as a starting point). Try containsKey(15) to see what the difference is.

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 \text{ if } n > 1 \\ 3 \quad \text{otherwise} \end{cases}$$

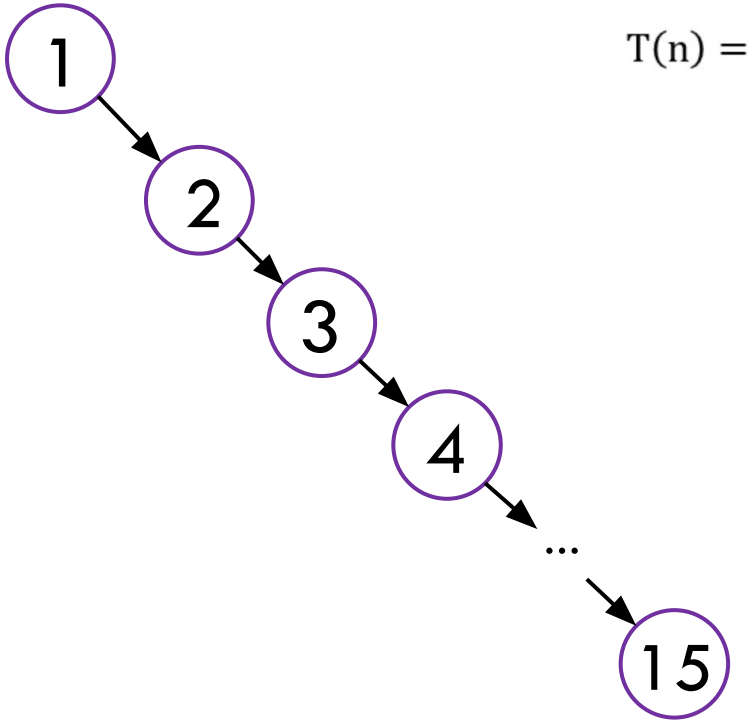
$T(n) = \theta(\log n)$

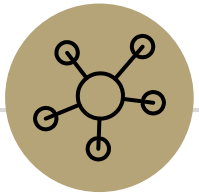


Degenerate – for every node, all of its descendants are in the right subtree.

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \quad \text{otherwise} \end{cases}$$

$T(n) = \Theta(n)$





Questions?

So far:

- Binary Trees, definitions
- Binary Search Tree, invariants
- Best/Worst case runtimes for BTs and BSTs
  - where the key is located
  - how the tree is structured

How are we going to make this simpler/more efficient? Let's enforce some invariants!

Observation: What was important was actually the height of the tree.

- **Height:** number of edges on the longest path from the root to a leaf.

That's the number of recursive calls we're going to make

- And each recursive call does a constant number of operations.

The BST invariant makes it easy to know where to find a `key`

But it doesn't force the tree to be short.

Let's add an invariant that forces the height to be short!

Invariants

Why not just make the invariant "keep the height of the tree at most O(log *n*)"?

The invariant needs to be easy to maintain.

Every method we write needs to ensure it doesn't break it.
Can we keep that invariant true without making a bunch of other methods slow?

It's not obvious…

Writing invariants is more art than science

- Learning that art is beyond the scope of the course
- But we'll talk a bit about how you might have come up with a good invariant (so our ideas are motivated)

When writing invariants, we usually start by asking "can we maintain this" then ask "is it strong enough to make our code as efficient as we want?"

Avoiding $\Theta(n)$ Behavior

Take 1 minute to consider this question and then discuss with those around you!

- Here are some invariants you might try.
  Can you maintain them? If not what can go wrong?
- Do you think they are strong enough to make containsKey efficient?
- **Try to come up with BSTs that show these rules aren't useful / too strict.**

Root Balanced: The root must have the same number of nodes in its left and right subtrees

Recursively Balanced: Every node must have the same number of nodes in its left and right subtrees.

Root Height Balanced: The left and right subtrees of the root must have the same height.

Avoiding $\Theta(n)$ Behavior

Root Balanced: The root must have the same number of nodes in its left and right subtrees

too weak

Recursively Balanced: Every node must have the same number of nodes in its left and right subtrees.

too strong

Root Height Balanced: The left and right subtrees of the root must have the same height.

too weak

Takeaways

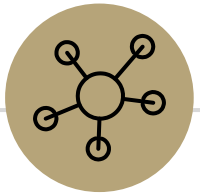
- Need requirements everywhere, not just at root
- Forcing things to be exactly equal is too difficult to maintain.

Invariant Lessons

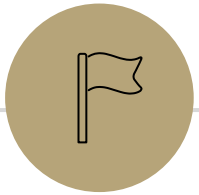
- Need requirements everywhere, not just at root
- Forcing things to be exactly equal is too difficult to maintain.

Roadmap

- Binary Trees
- Binary Search Trees, invariants
  - runtimes
- **AVL Trees, invariants**



Questions?



That’s all!

Avoiding the Degenerate Tree

An AVL tree is a binary search tree that also meets the following invariant

AVL invariant: For every node, the height of its left subtree and right subtree differ by at most 1.

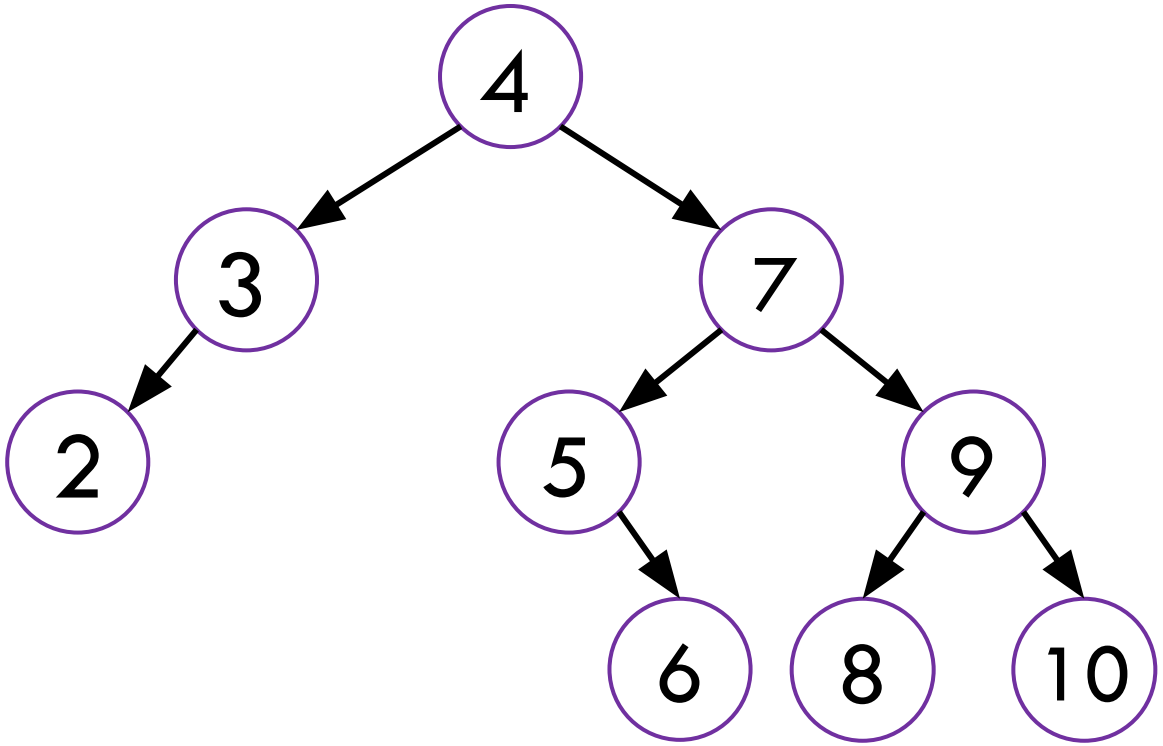
This will avoid the $\Theta(n)$ worst-case behavior! As long as

1. Our tree will have height $O(\log n)$.
2. We're able to maintain this property when inserting/deleting

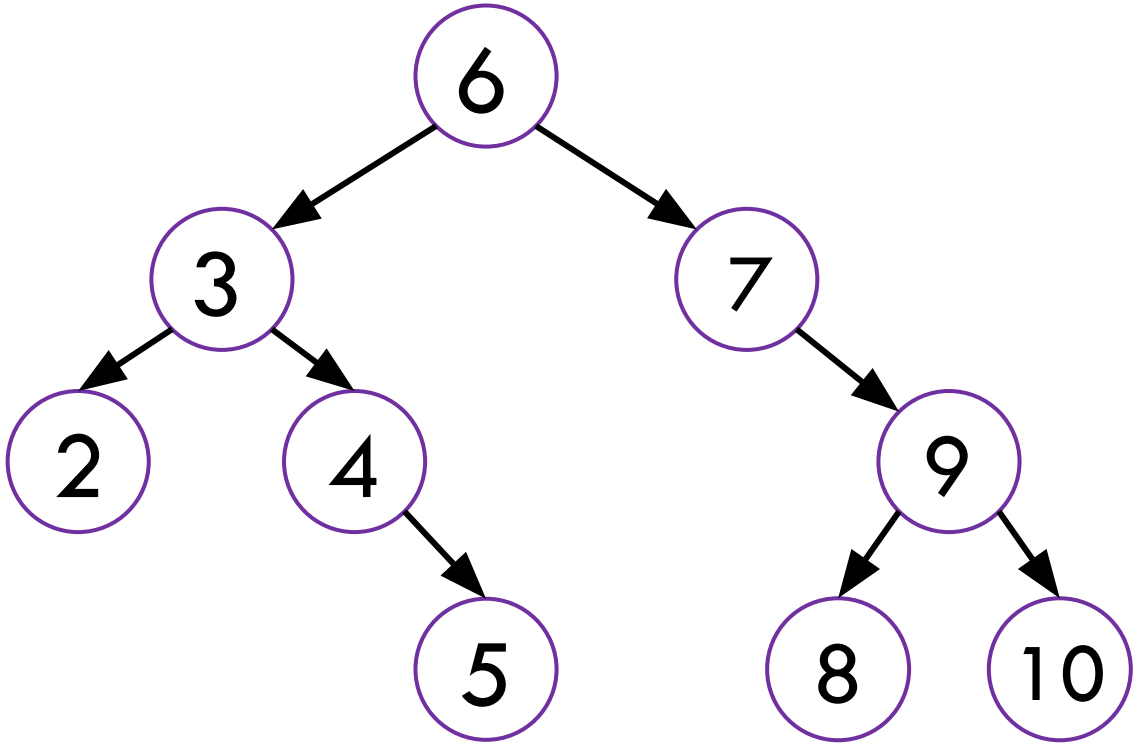
Practice w AVL invariants

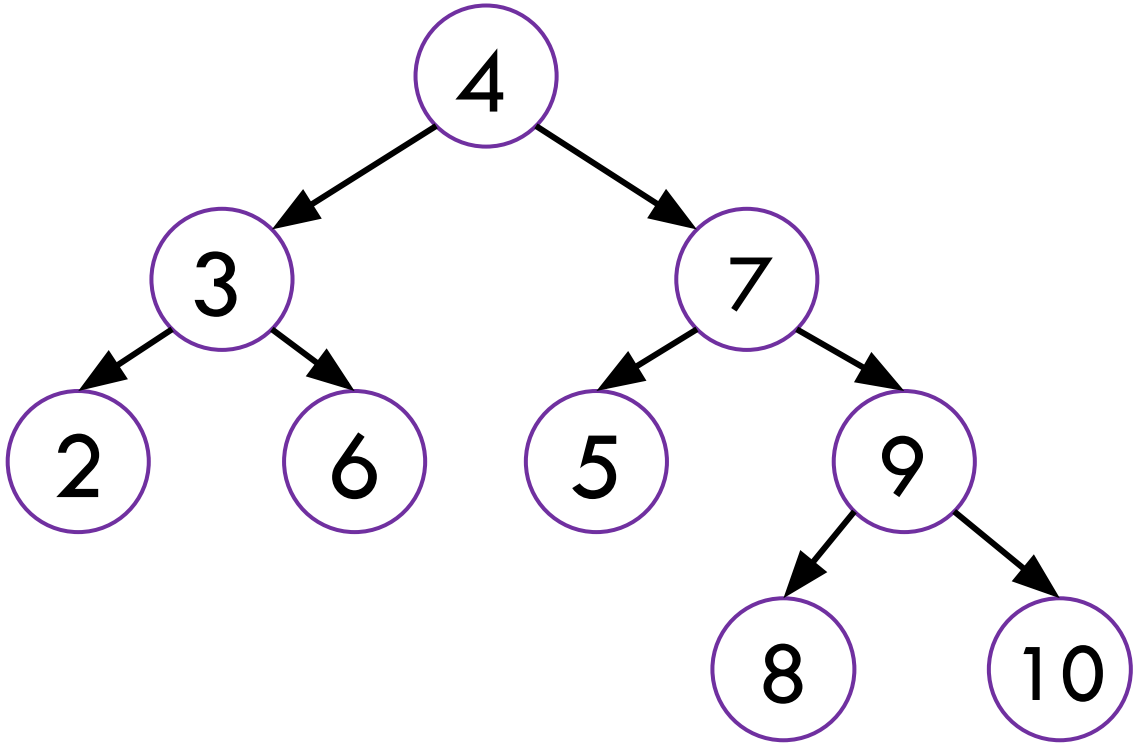
AVL invariant: For every node, the height of its left subtree and right subtree differ by at most 1.

Is this a valid AVL tree?



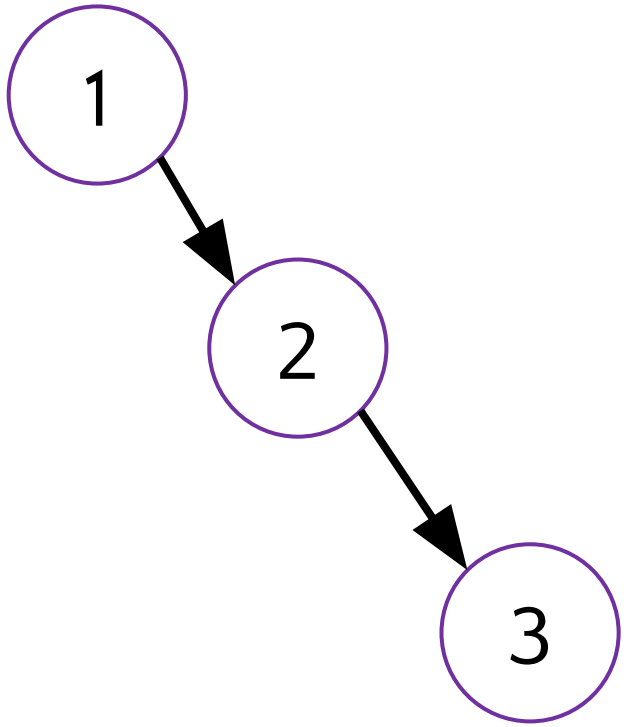
Are These AVL Trees?

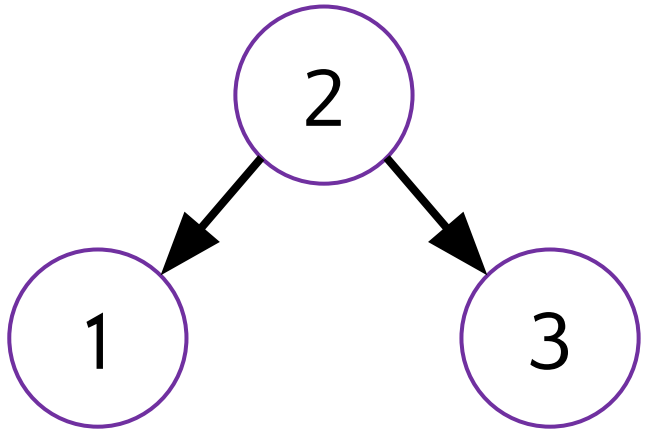




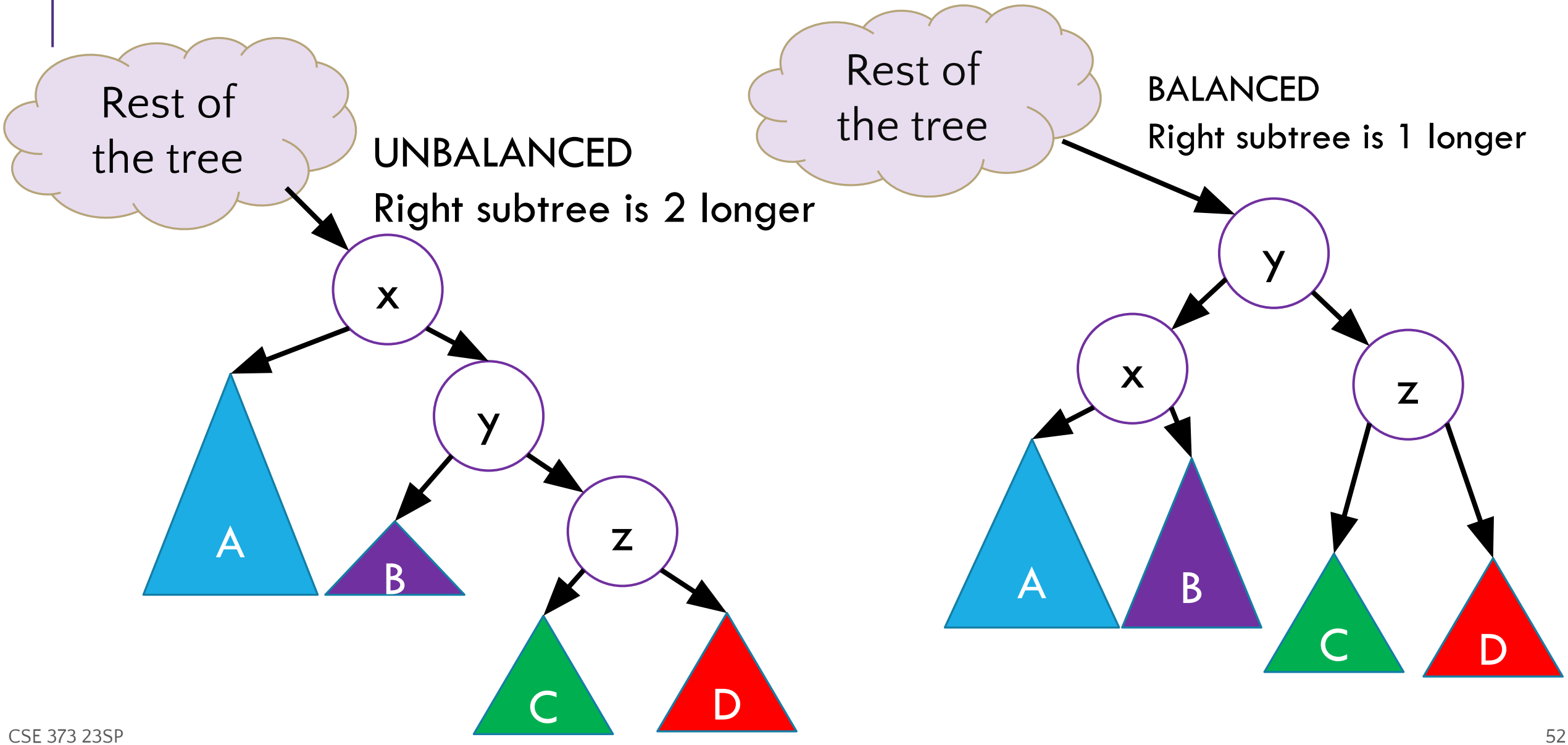
Insertion

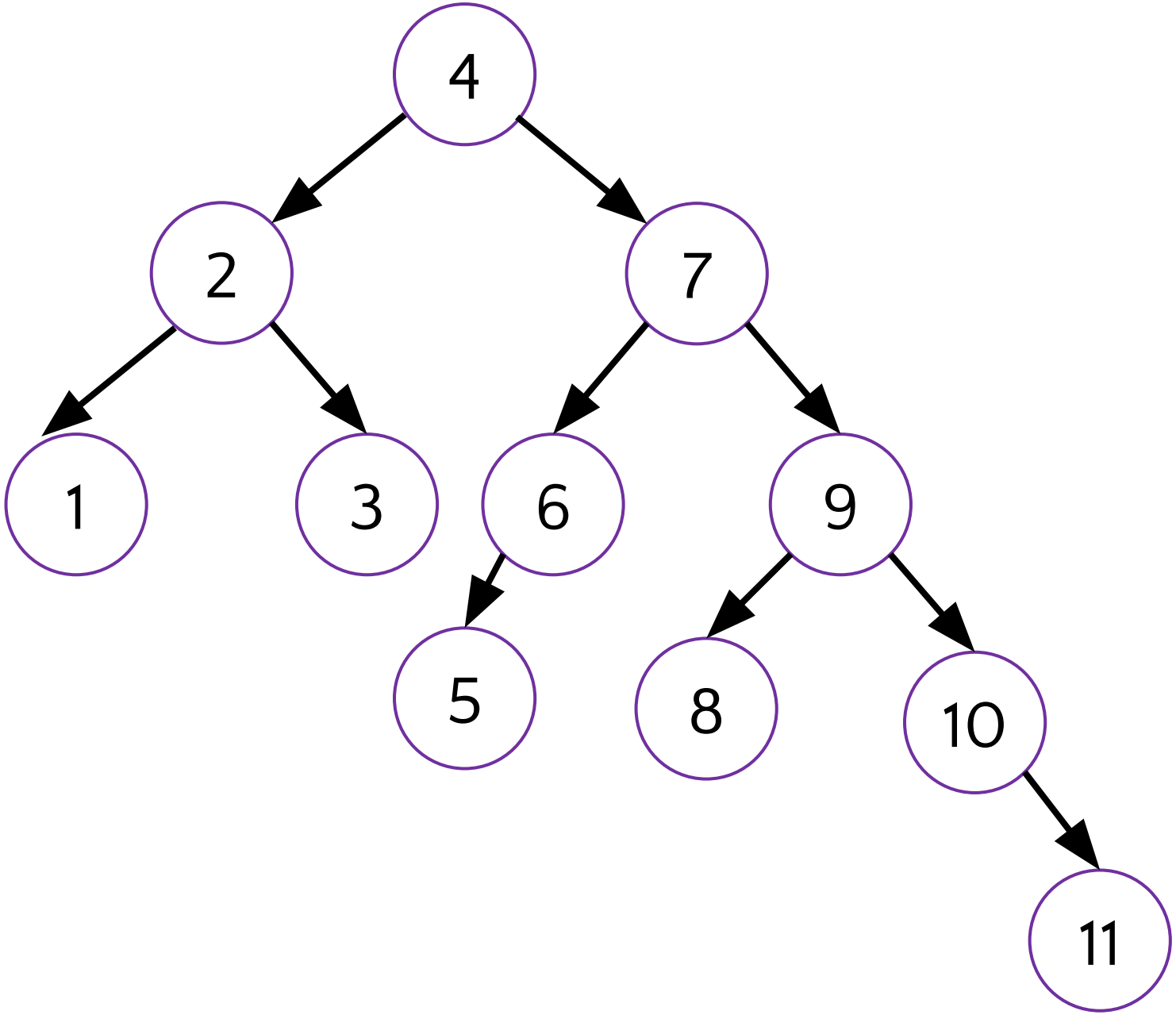
What happens if when we do an insertion, we break the AVL condition?

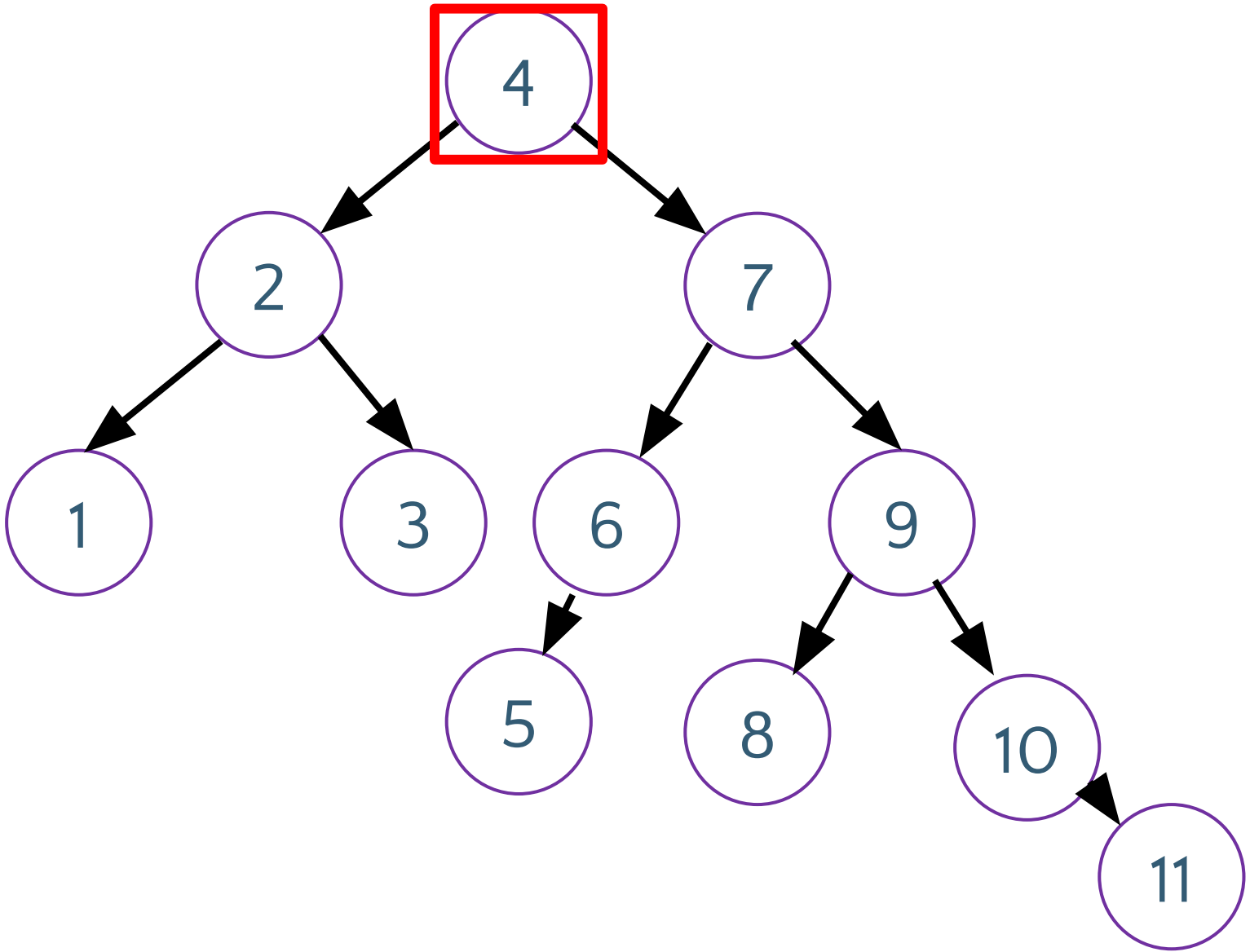


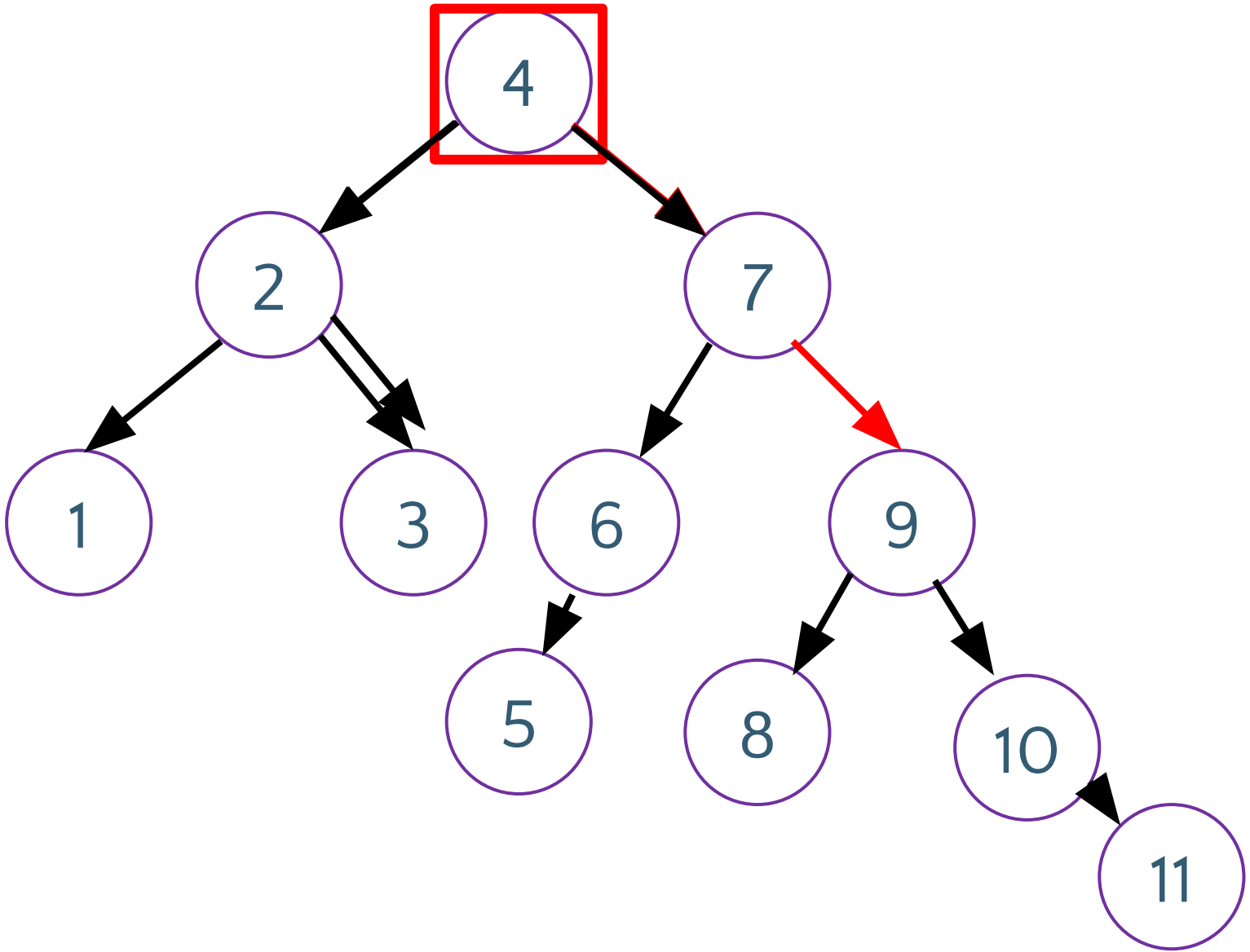


Left Rotation



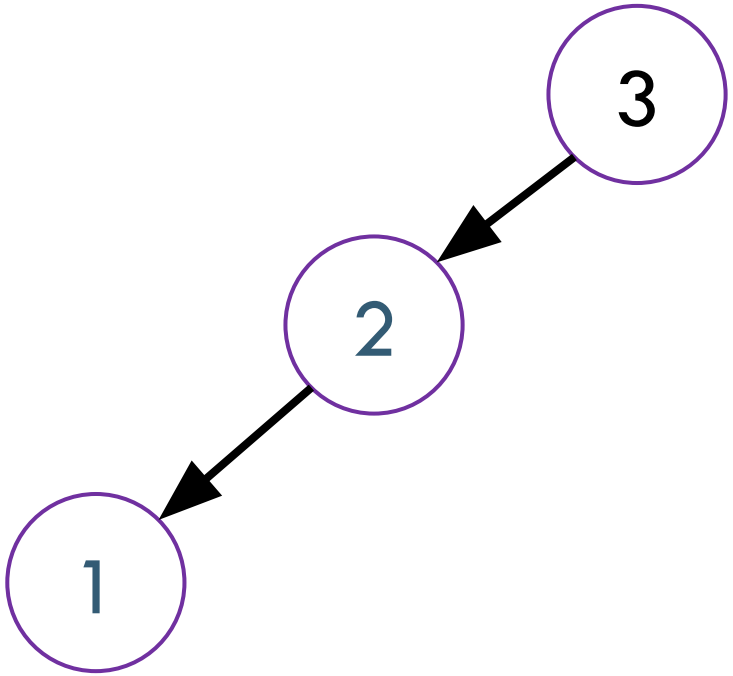

4
2
7
1
3
6
9
5
8
10
11

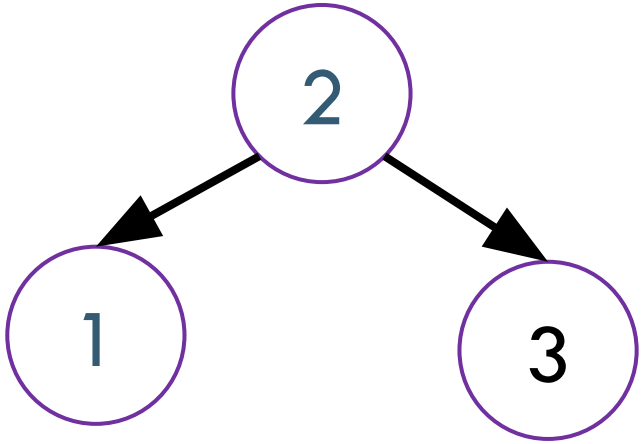

4
2
7
1
3
6
9
5
8
10
11


4
2
7
1
3
6
9
5
8
10
11

Meme break (it's from some marvel movie that I haven't watched -- you're not alone if you don't get this reference)

Right rotation

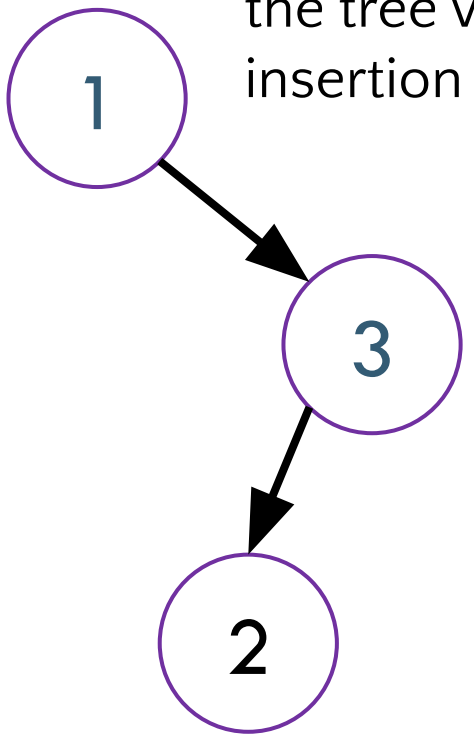


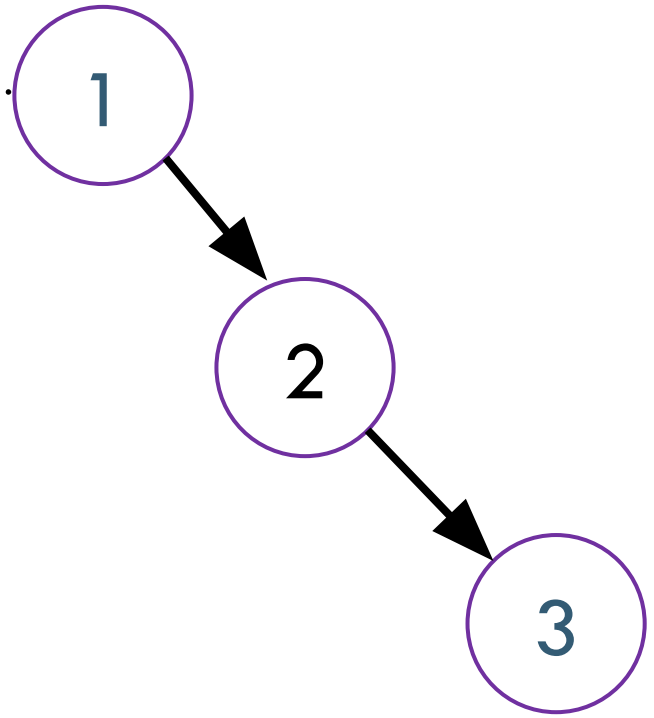


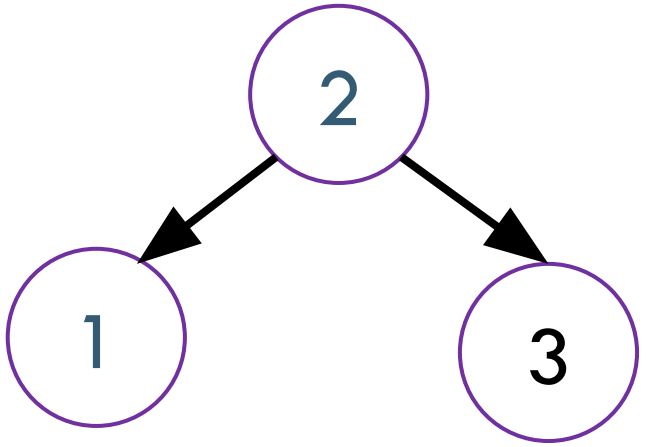
Just like a left rotation, just reflected.

It Gets More Complicated

There's a "kink" in the tree where the insertion happened.





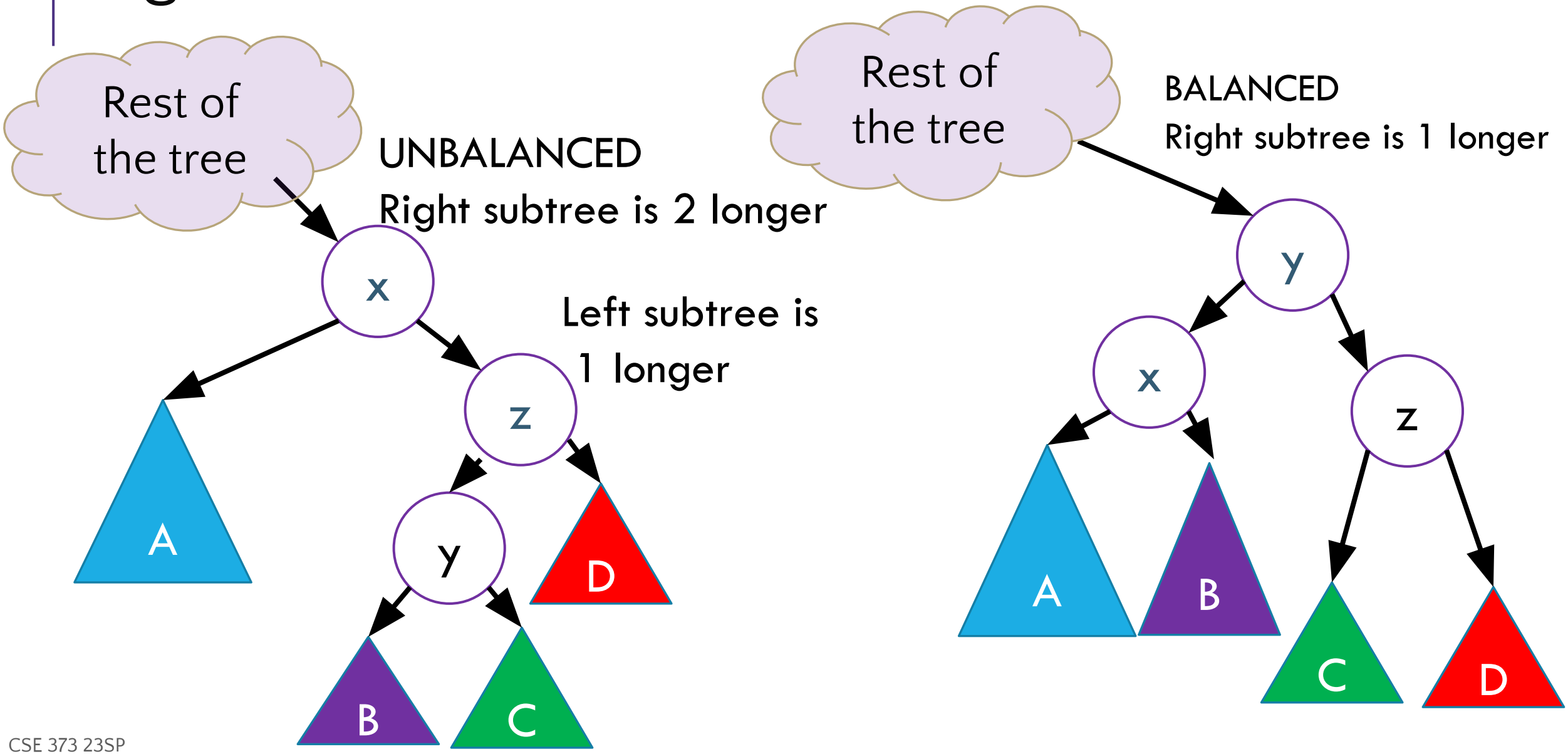


Can't do a left rotation

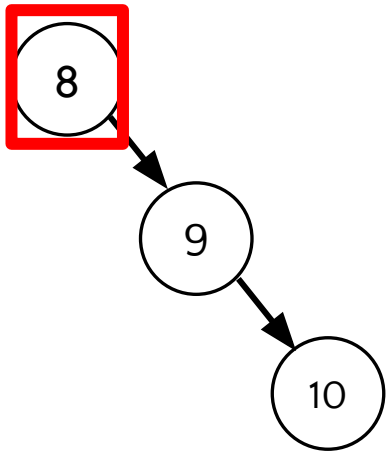
Do a "right" rotation around 3 first.

Now do a left rotation.

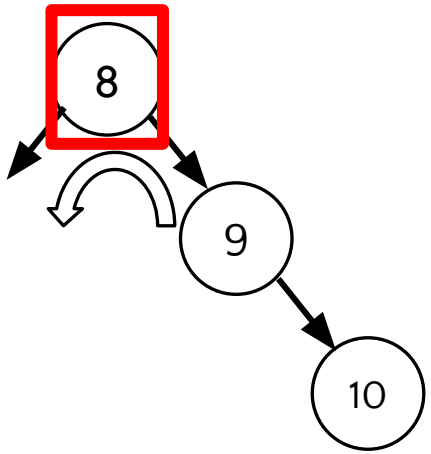
Right Left Rotation



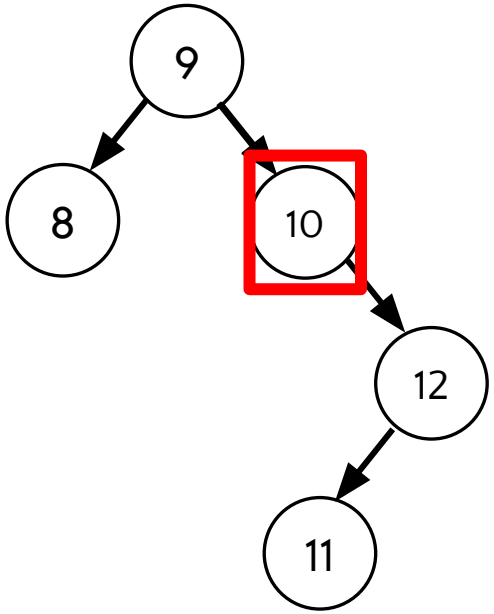
AVL Example: 8,9,10,12,11



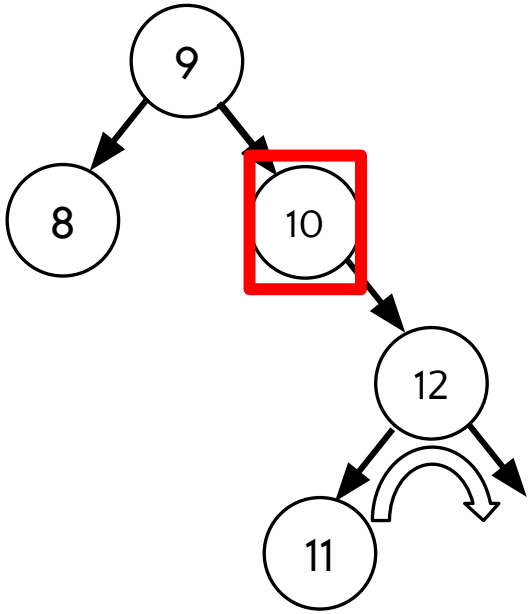
AVL Example: 8,9,10,12,11



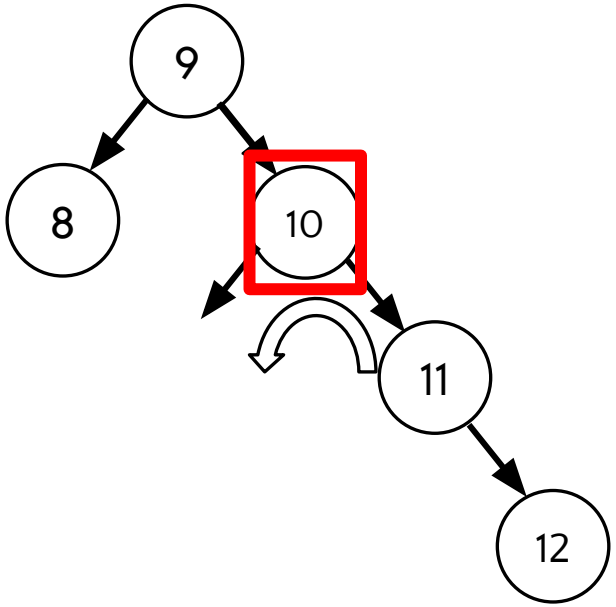
AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



How Long Does Rebalancing Take?

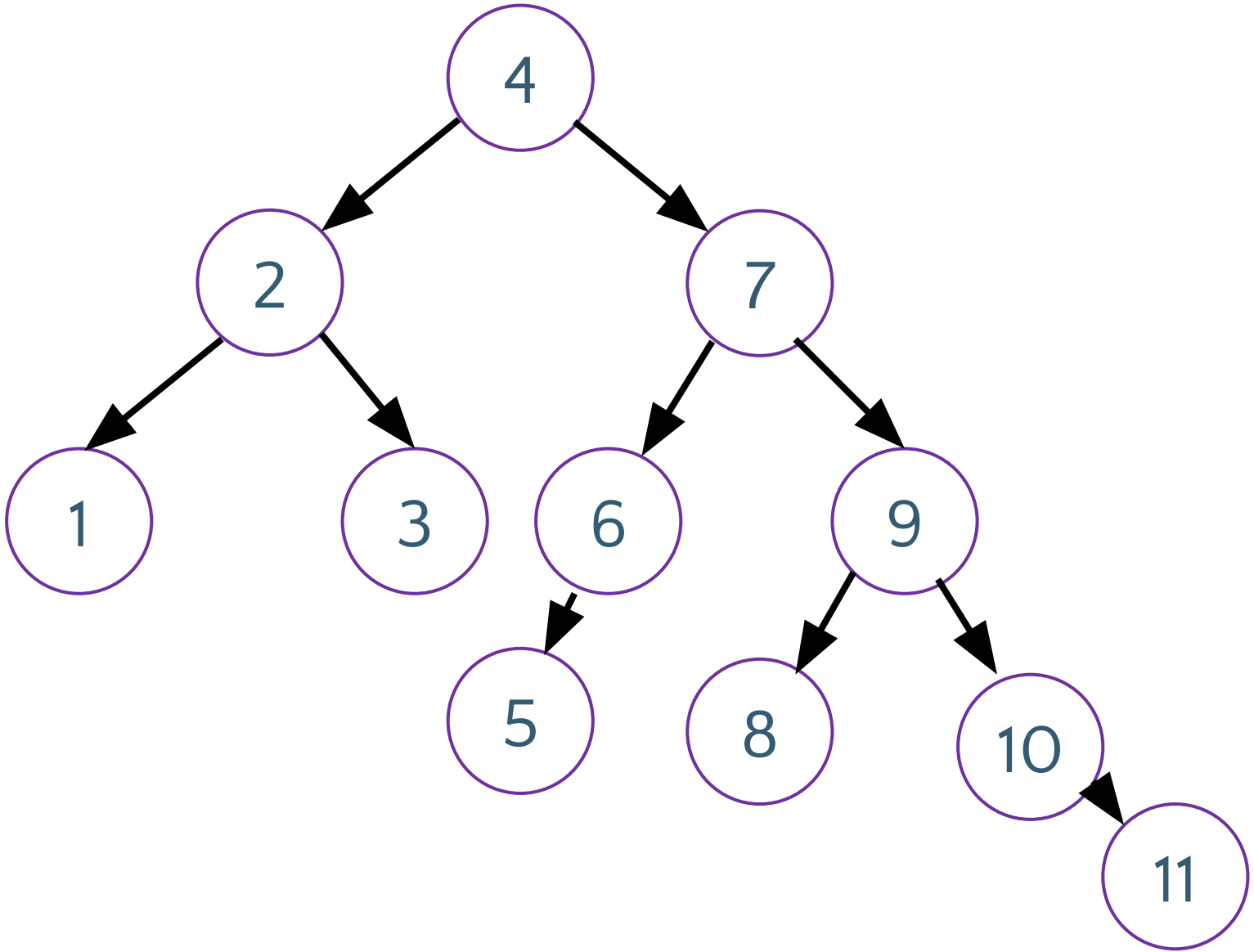
Assume we store in each node the height of its subtree.

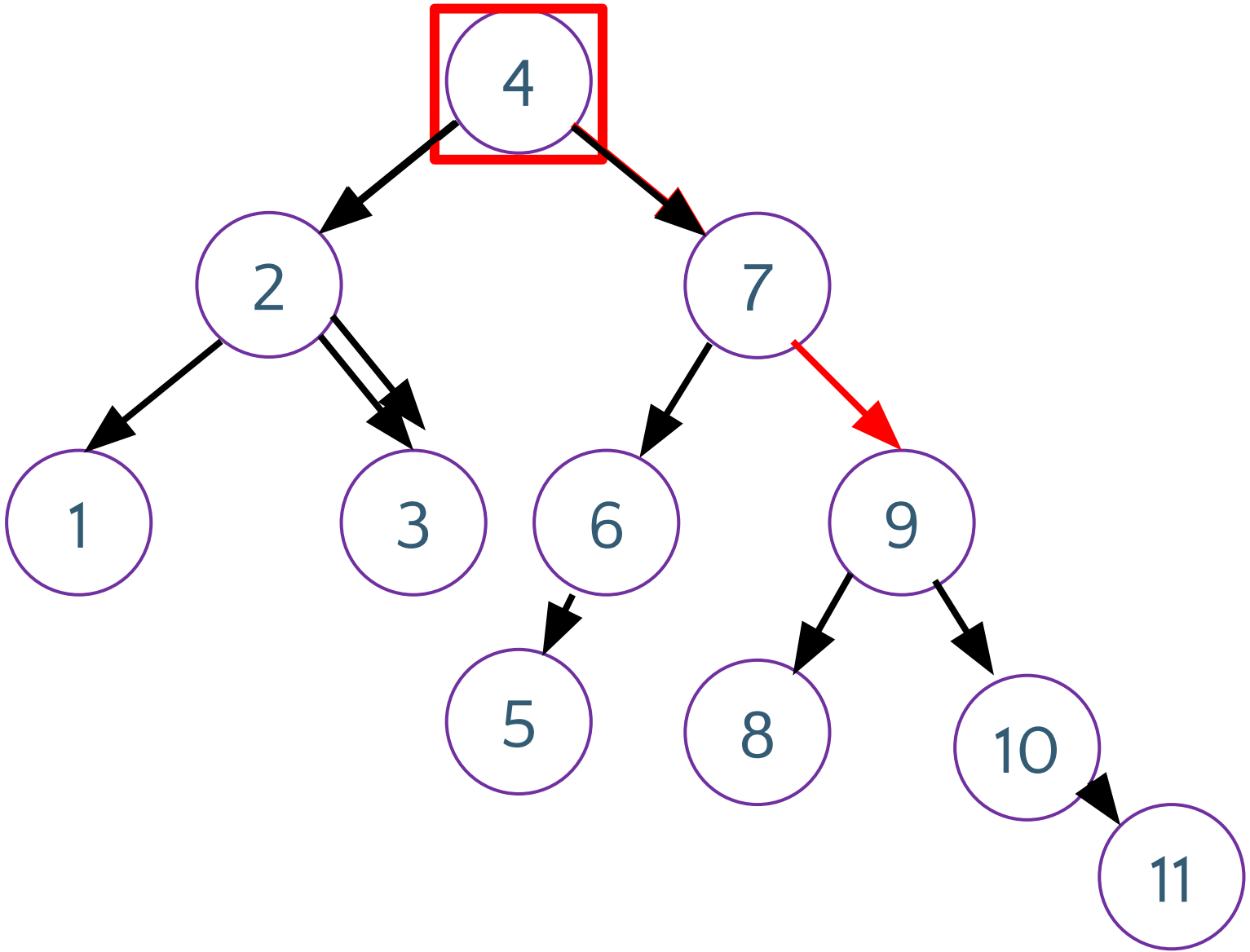
How do we find an unbalanced node?

How many rotations might we have to do?

How Long Does Rebalancing Take?

- Assume we store in each node the height of its subtree.
- How do we find an unbalanced node?
  - Just go back up the tree from where we inserted.
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node.
  - A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion
  - log(n) time to traverse to a leaf of the tree
  - log(n) time to find the imbalanced node
  - constant time to do the rotation(s)
  - **Theta(log(n)) time for put** (the worst case for all interesting + common AVL methods (get/containsKey/put is logarithmic time)


4
2
7
1
3
6
9
5
8
10
11


4
2
7
1
3
6
9
5
8
10
11

Deletion

There is a similar set of rotations that will always let you rebalance an AVL tree after a deletion

The textbook (or Wikipedia) can tell you more.

We won't test you on deletions but here's a high-level summary about them:

- Deletion is similar to insertion
- It takes $\Theta(\log n)$ time on a dictionary with n elements
- We won't ask you to perform a deletion