Data Structures

Map Implementation with BSTs

COMP128 Data Structures

Binary Search Trees

- One of the tree applications is binary search trees.
- This presentation illustrates how another data type, a map, or dictionary is implemented with binary search trees.

The Map Data Type

```
public void put(key for new item, new item)
```

- The insertion method for a dictionary has two parameters.

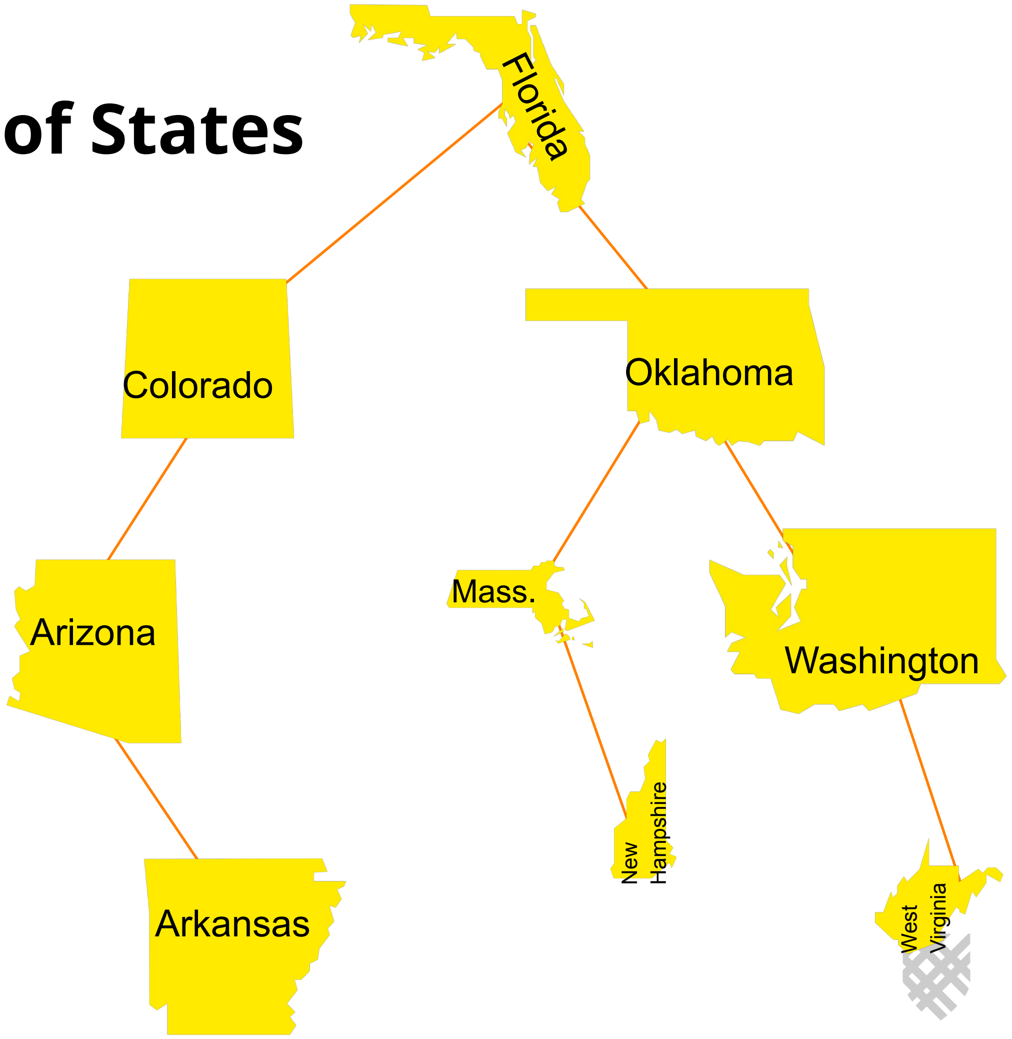
```
public Object get("Washington")
```

- When you want to retrieve an item, you specify the key, and the retrieval method returns the item.

We'll look at how a binary tree can be used as the internal storage mechanism for the dictionary.

A Binary Search Tree of States

The data in the dictionary will be stored in a binary tree, with each node containing an **item** and a **key**.

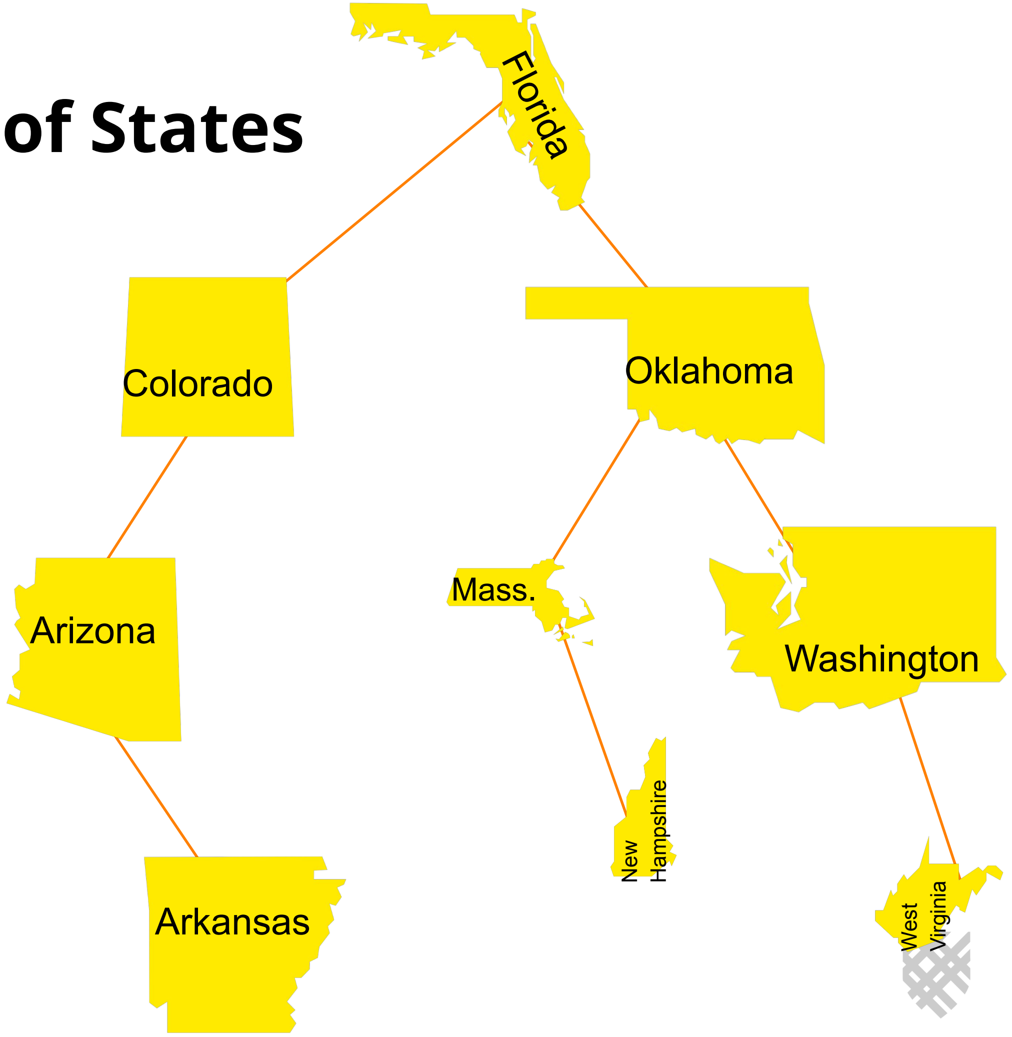


A Binary Search Tree of States

Storage rules:

Every key to the **left** of a node is alphabetically **before** the key of the node.

Every key to the **right** of a node is alphabetically **after** the key of the node.



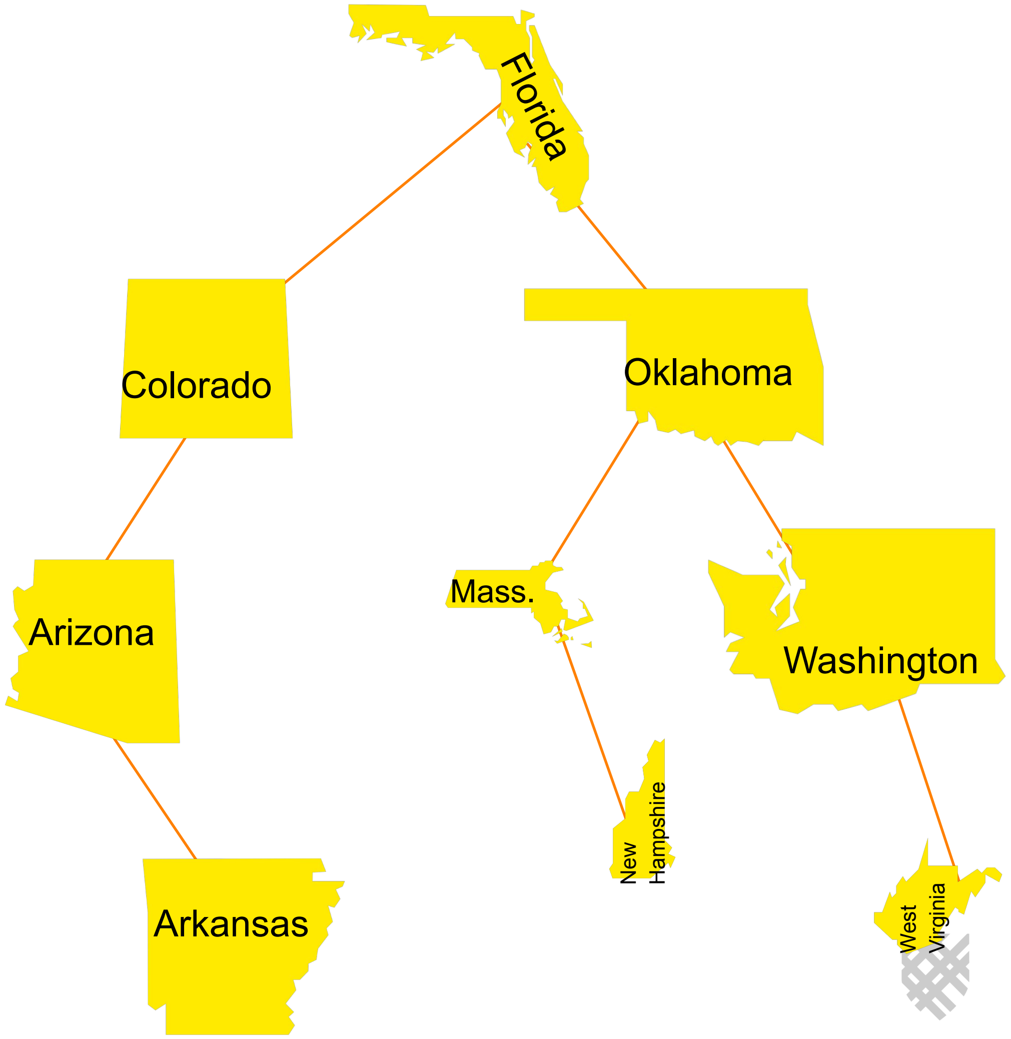
Retrieving Data

Start at the root.

If the current node has the key, then stop and retrieve the data.

If the current node's key is too **large**, move **left** and repeat 1-3.

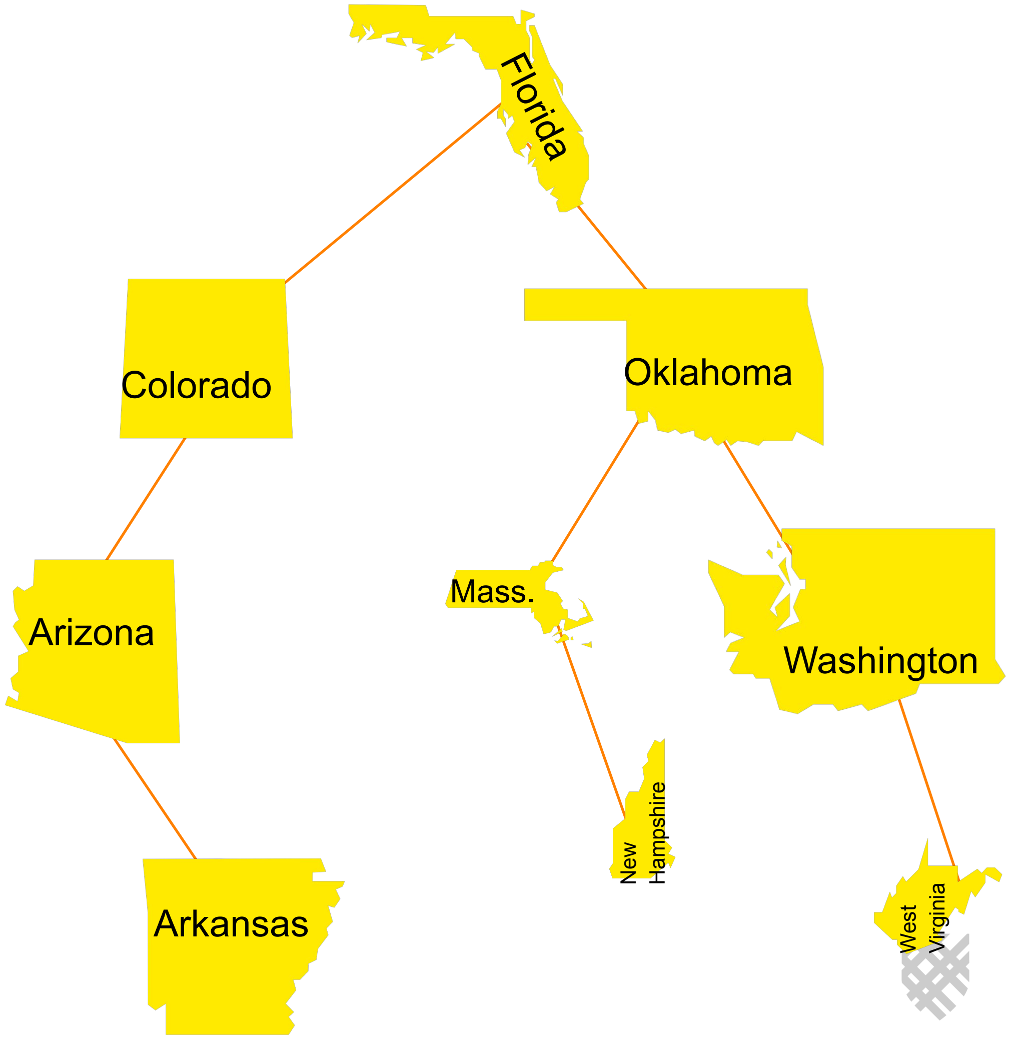
If the current node's key is too **small**, move **right** and repeat 1-3.



Adding a New Item

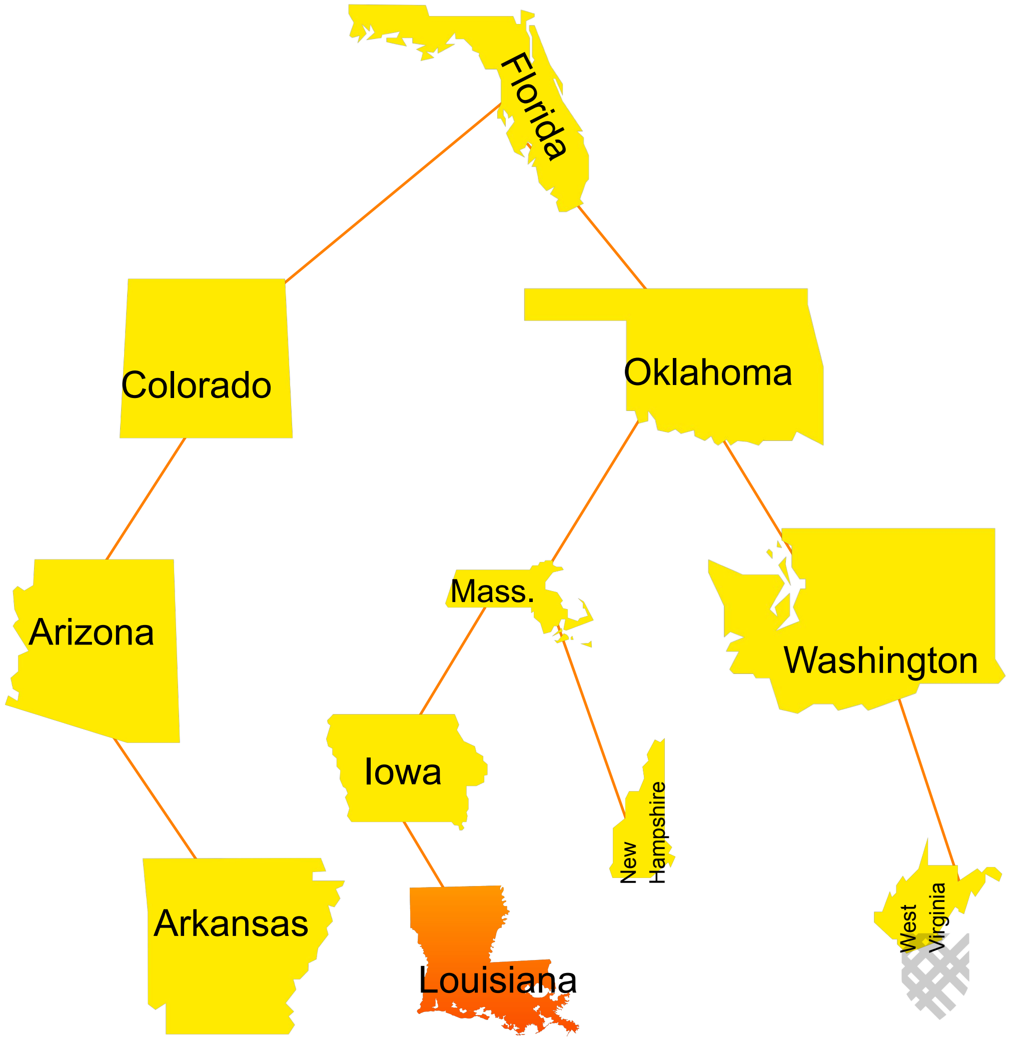
Pretend that you are trying to find the key, but stop when there is no node to move to.

Add the new node at the spot where you would have moved to if there had been a node.



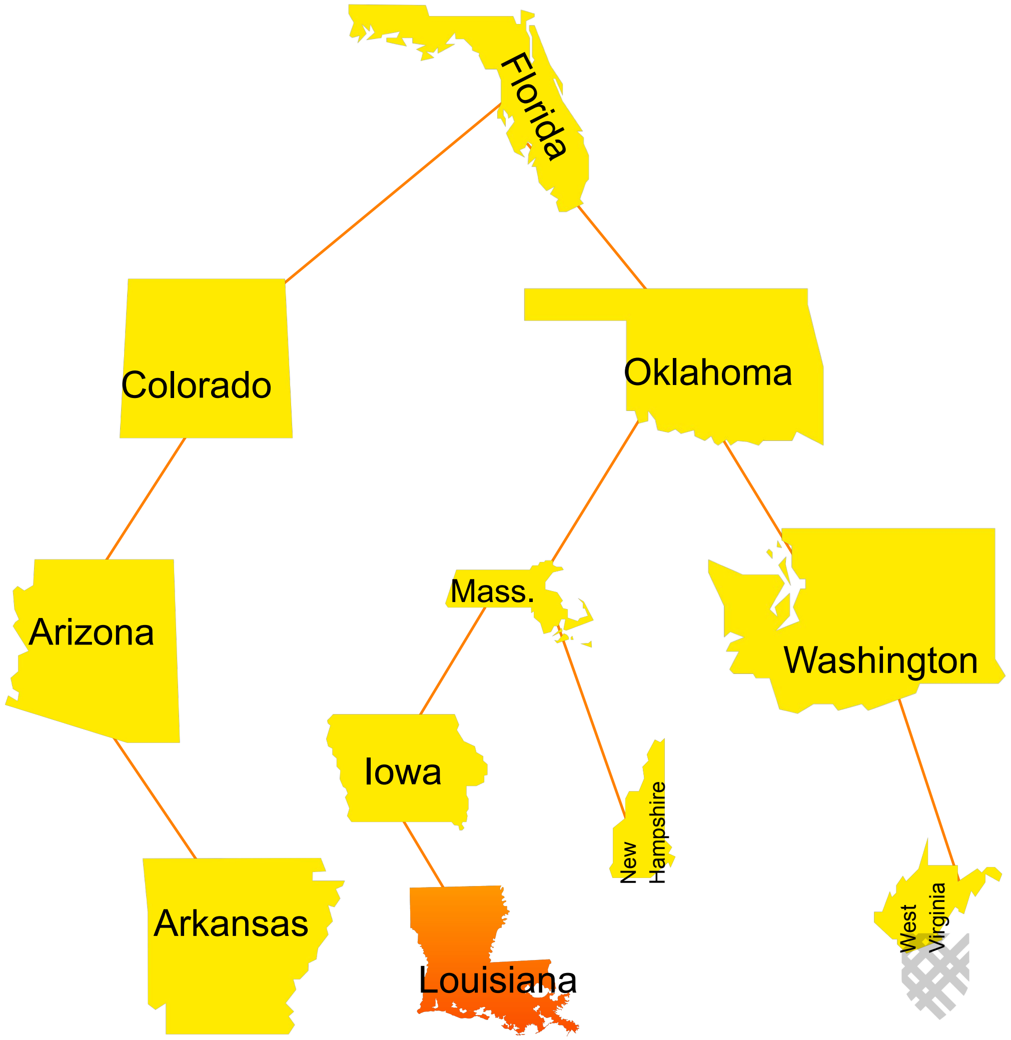
Removing an Item

- Find the item.
- If necessary, swap the item with one that is easier to remove.
- Remove the item.



Removing “Florida”

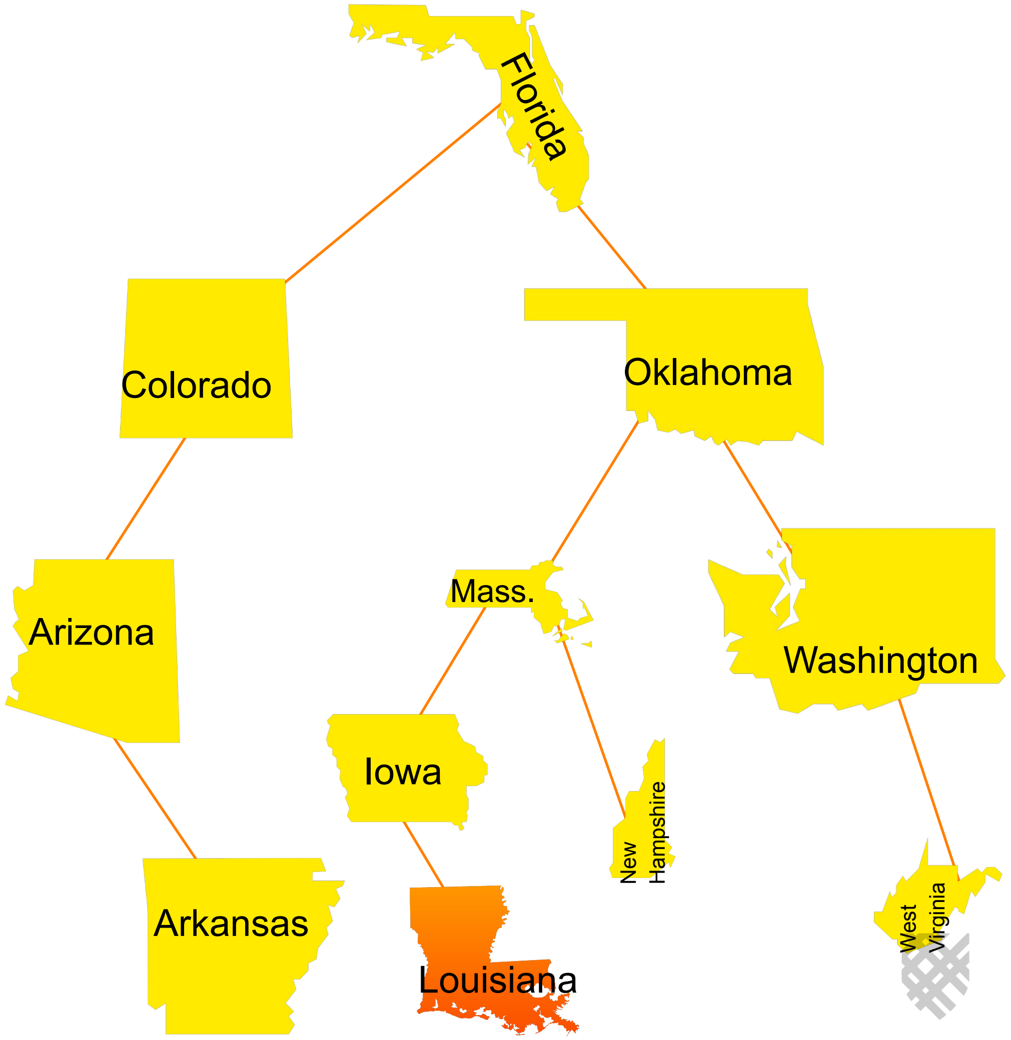
- Find the item.



Removing “Florida”

- Find the item.

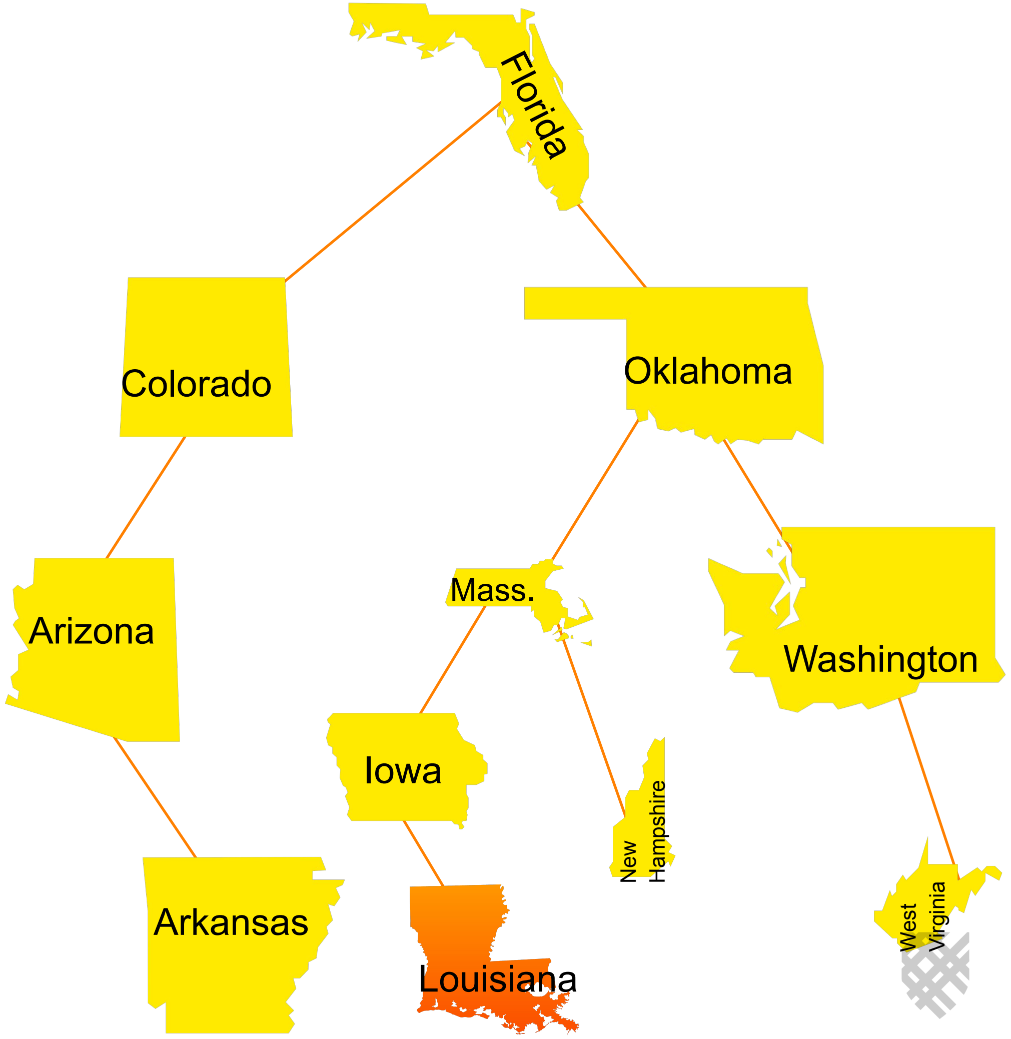
Florida cannot be removed at the moment…



Removing “Florida”

- Find the item.
- If necessary, swap the item with one that is easier to remove.

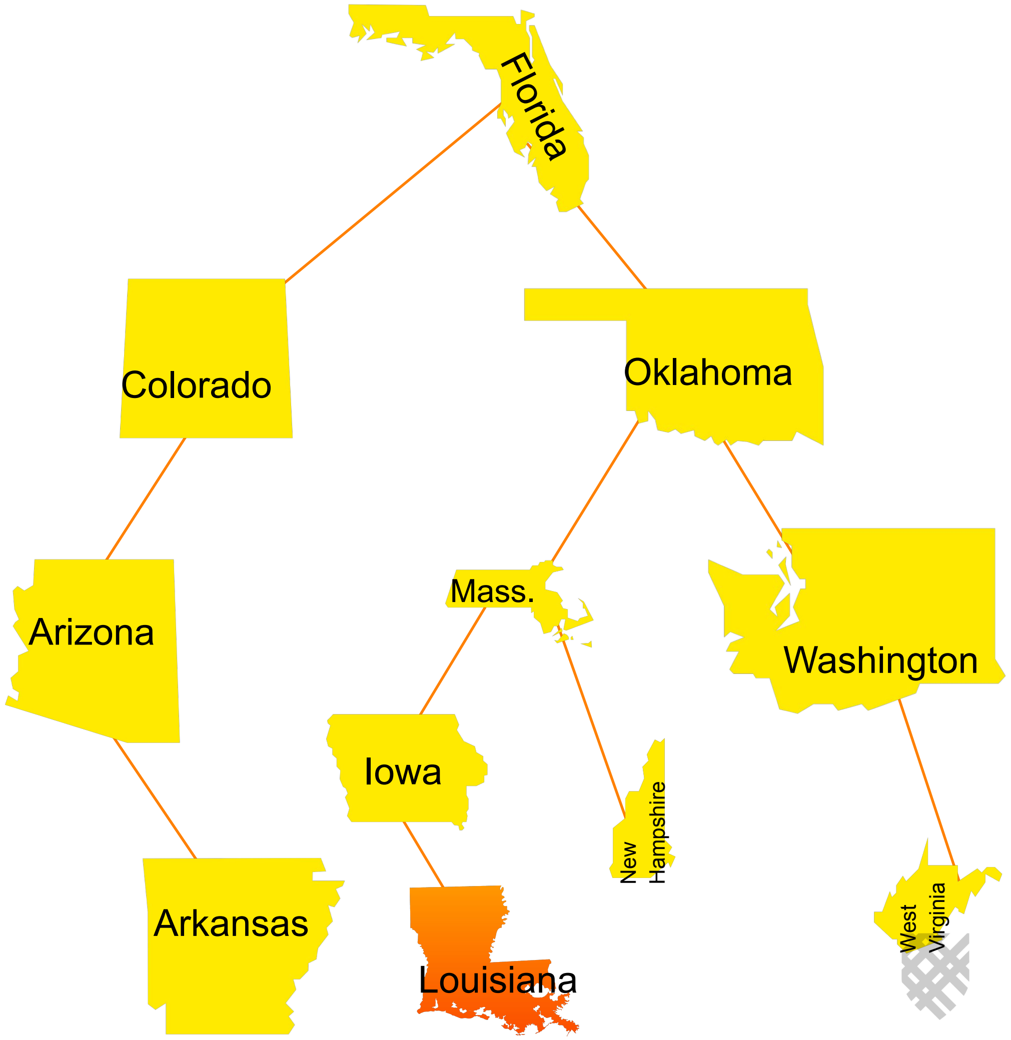
The problem of breaking the tree happens because Florida has 2 children.



Removing “Florida”

- Find the item.
- If necessary, swap the item with one that is easier to remove.

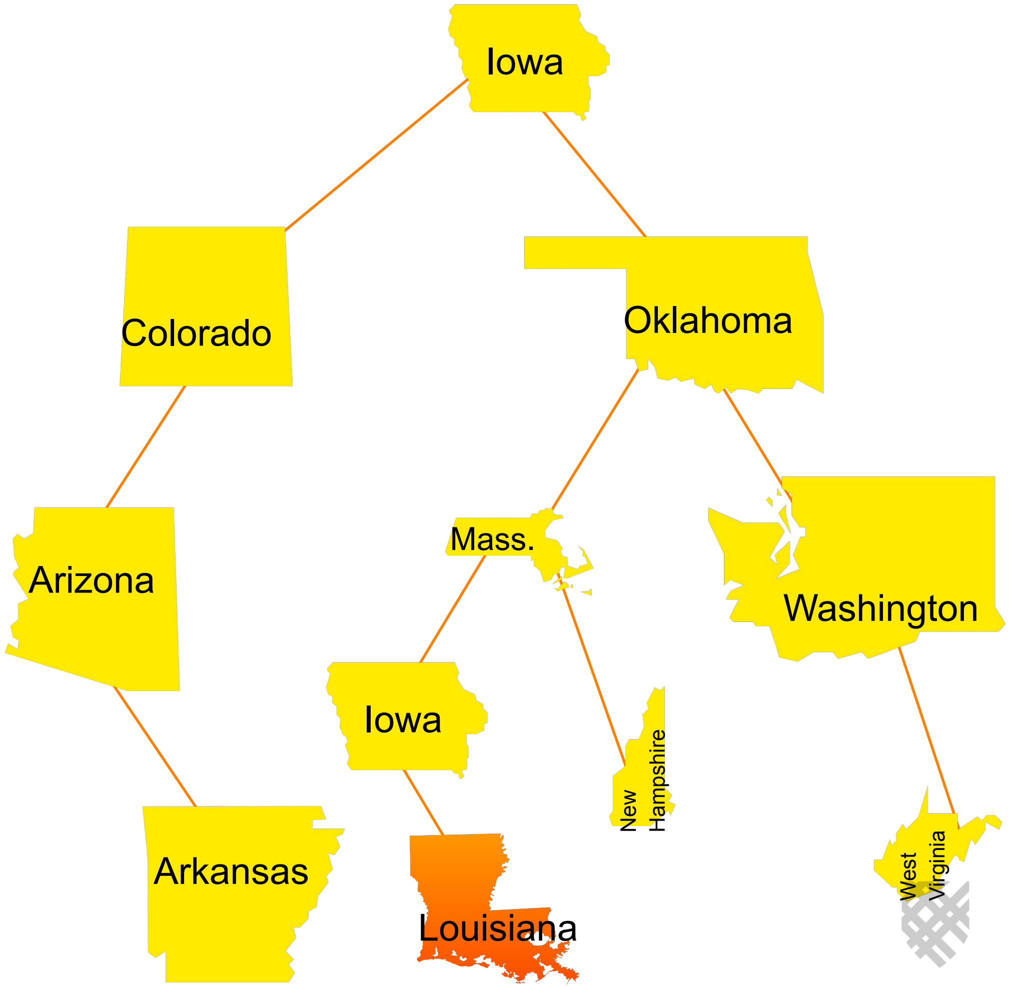
For the rearranging, take the smallest item in the right subtree…



Removing “Florida”

- Find the item.
- If necessary, swap the item with one that is easier to remove.

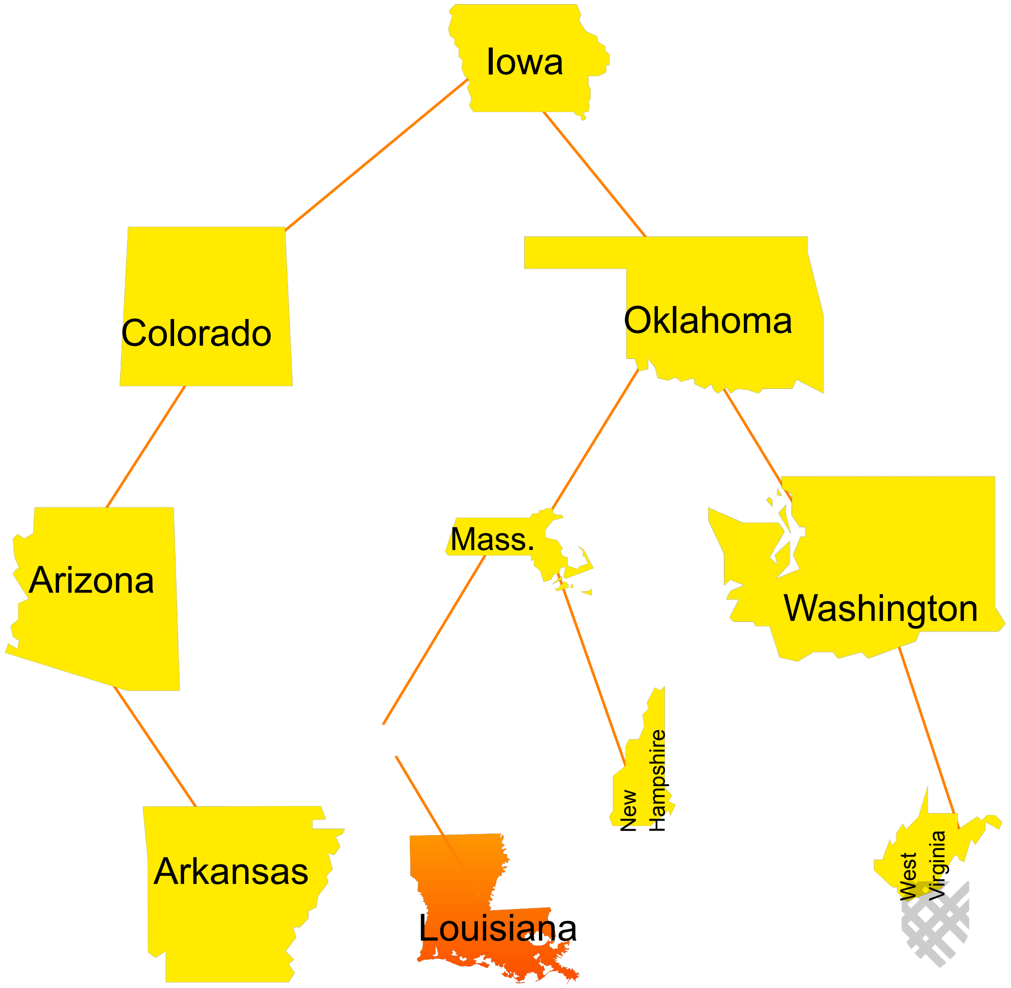
…copy that smallest item onto the item that we’re removing…



Removing "Florida"

- Find the item.
- If necessary, swap the item with one that is easier to remove.

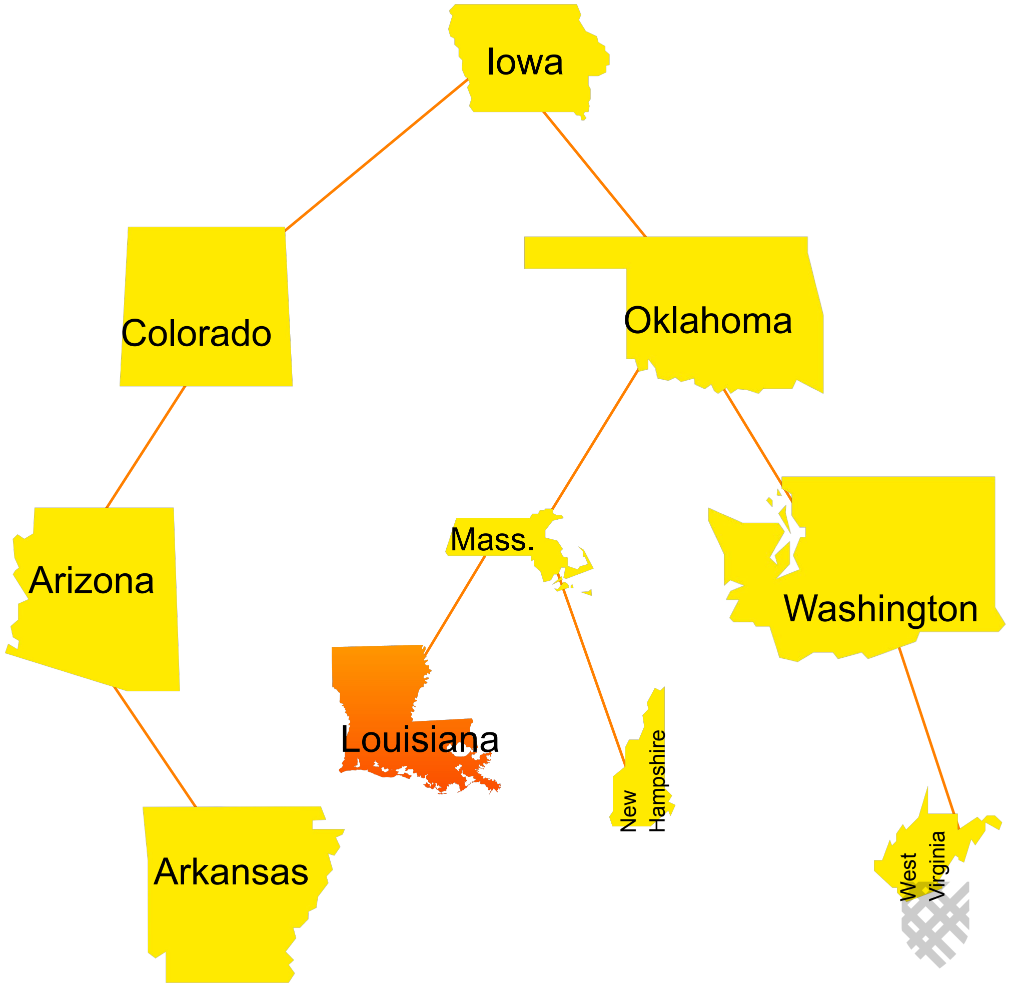
... and then remove the extra copy of the item we copied...



Removing “Florida”

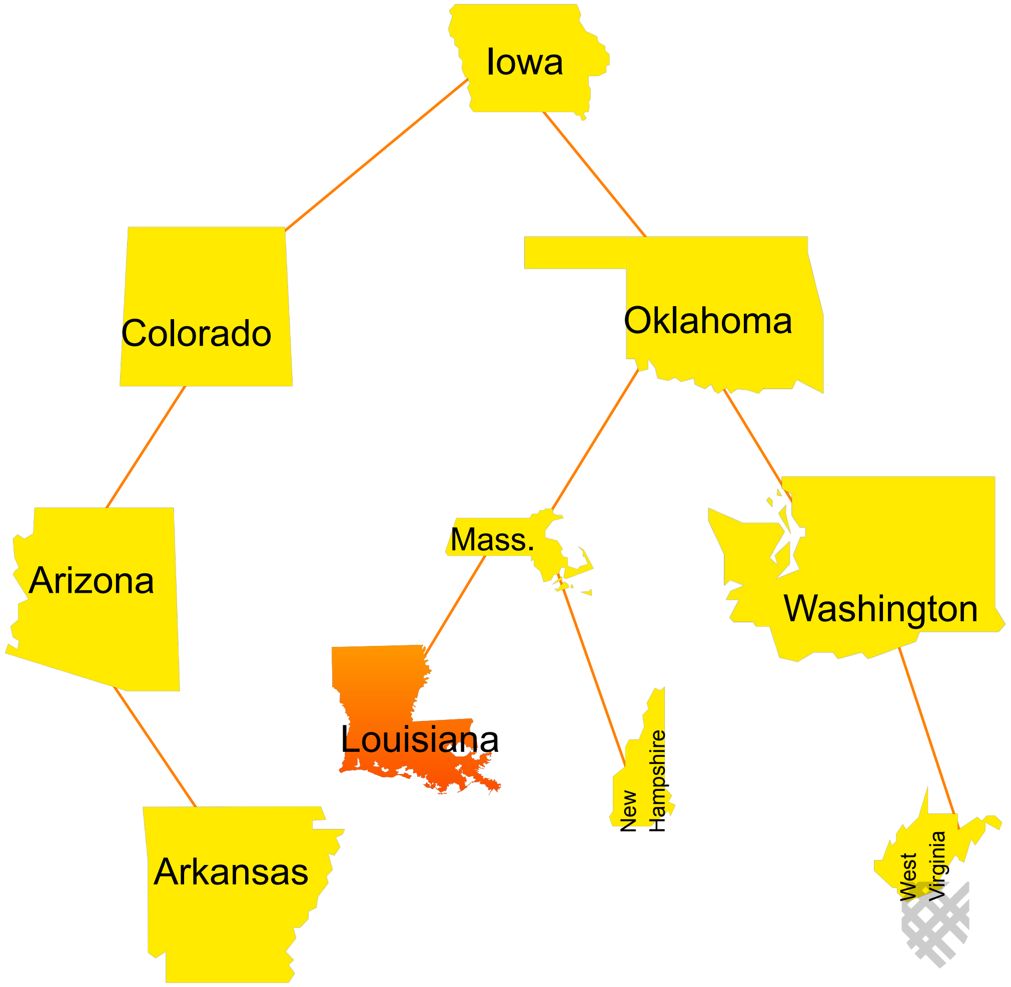
- Find the item.
- If necessary, swap the item with one that is easier to remove.

… and reconnect the tree



Removing “Florida”

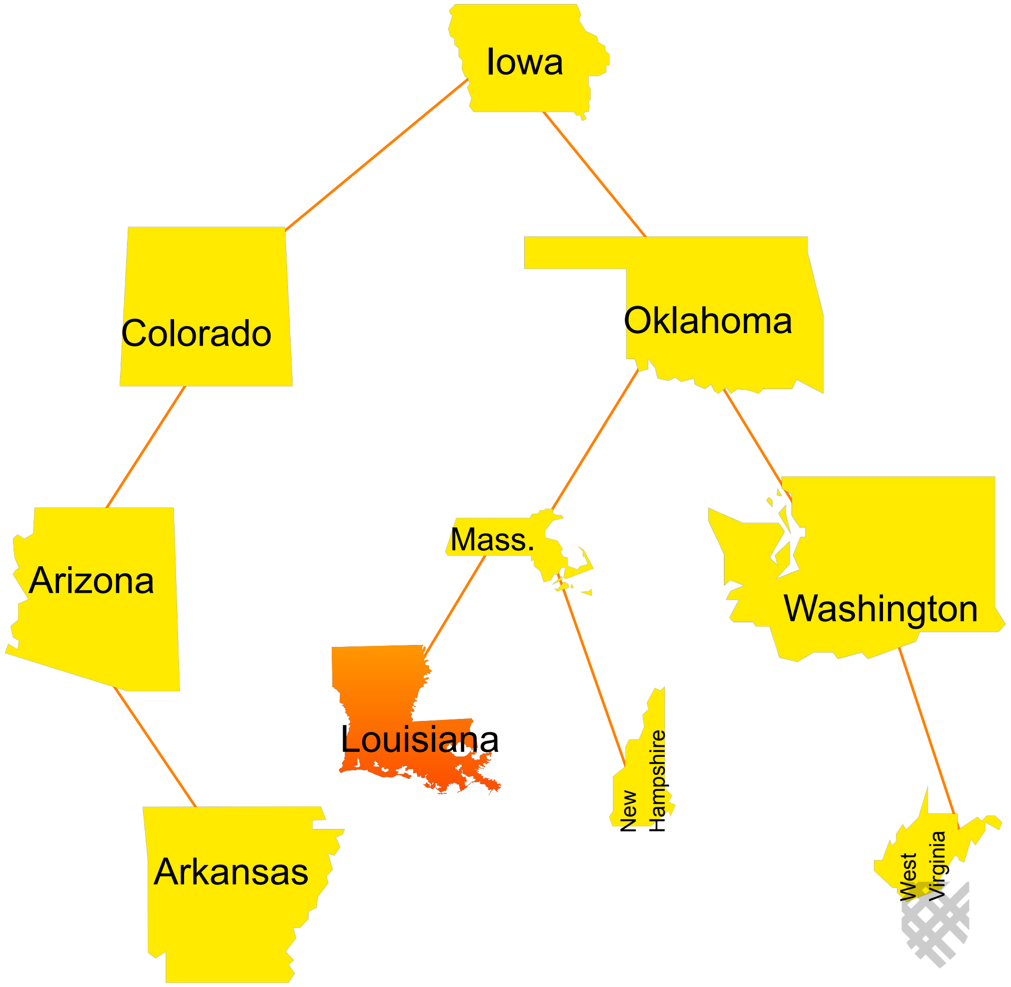
Why did I choose the smallest item in the right subtree?



Removing “Florida”

Why did I choose the smallest item in the right subtree?

Because every key must be smaller than the keys in its right subtree.



Removing an Item with a Given Key

- Find the item.
- If the item has a right child, rearrange the tree:
    - Find smallest item in the right subtree;
    - Copy that smallest item onto the one that you want to remove;
    - Remove the extra copy of the smallest item (making sure that you keep the tree connected).
- Else just remove the item.

Summary

- Binary search trees are a good implementation of data types such as sets, bags, and maps, where you have a notion of sorted order of items.
- In Java, the sorted BST implementation of a map or dictionary is the TreeMap
- The BST is implemented as a balanced red-black tree
- Useful when traversing in sorted order (treesort) is needed regularly

Recursion on Trees

There are often two patterns on how you can write recursive methods to process trees:

- Accumulating information about a tree
  - Counting total number of leaves, height
- Searching a tree node with certain property
  - Delete, add, search, findClosest