CSC212

Data Structure

- Section FG

Lecture 16

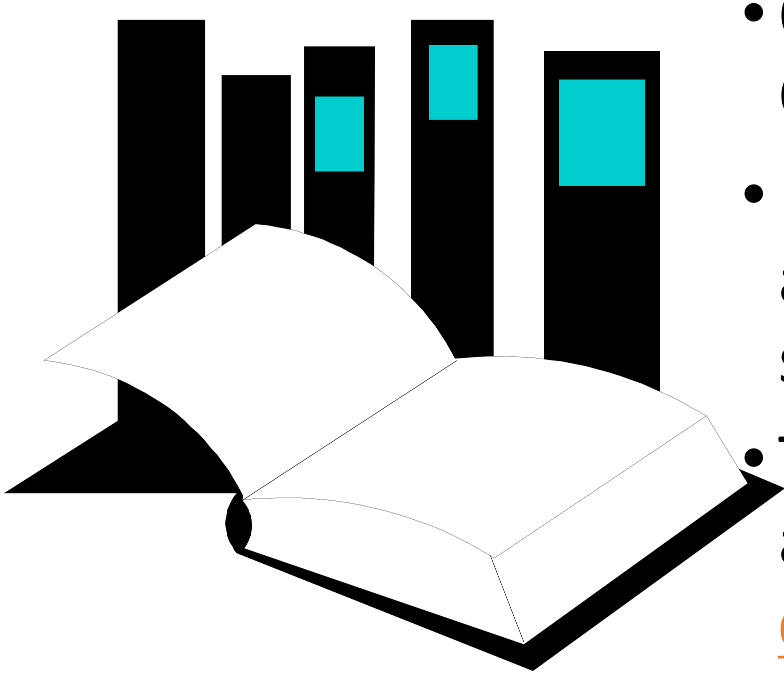
Binary Search Trees

Instructor: Feng HU

Department of Computer Science

City College of New York

Binary Search Trees



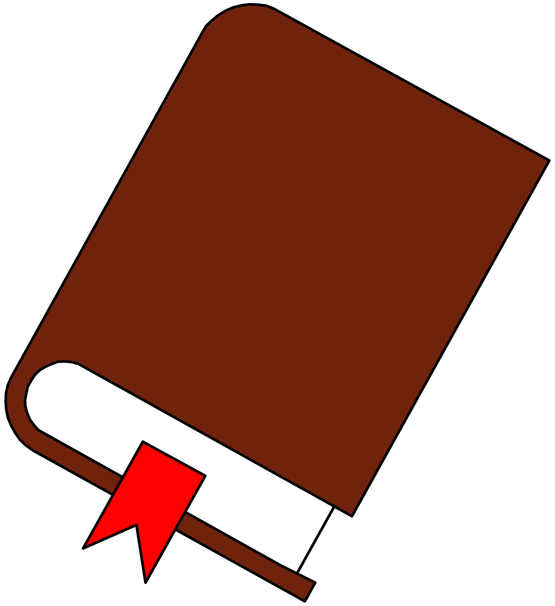
- One of the tree applications in Chapter 10 is **binary search trees**.
- In Chapter 10, binary search trees are used to implement bags and sets.
- This presentation illustrates how another data type called a **dictionary** is implemented with binary search trees.

Binary Search Tree Definition

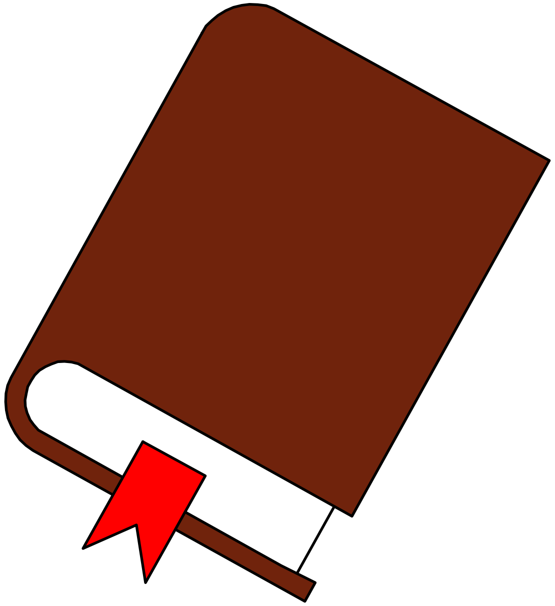
- In a binary search tree, the entries of the nodes can be compared with a strict weak ordering. Two rules are followed for every node n:
  - The entry in node n is NEVER less than an entry in its left subtree
  - The entry in the node n is less than every entry in its right subtree.

The Dictionary Data Type

- A dictionary is a collection of items, similar to a bag.
- But unlike a bag, each item has a string attached to it, called the item's <u>key</u>.



The Dictionary Data Type



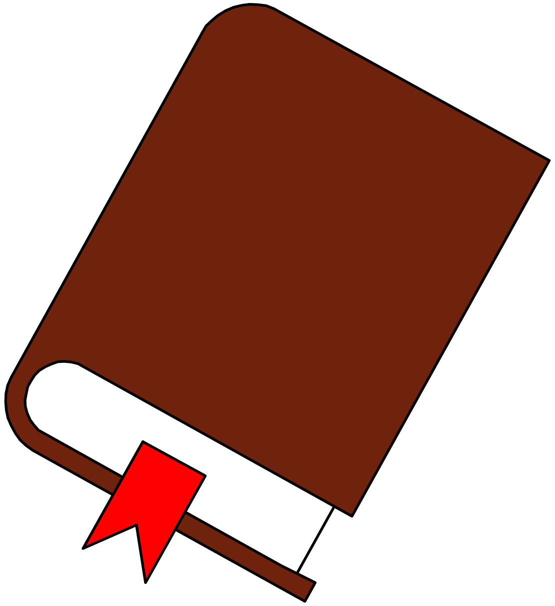
- A dictionary is a collection of **items**, similar to a bag.
- But unlike a bag, each item has a string attached to it, called the item's **key**.

Example:

The **items** I am storing are records containing data about a state.

The Dictionary Data Type

- A dictionary is a collection of **items**, similar to a bag.
- But unlike a bag, each item has a string attached to it, called the item's **key**.



Example:

The **key** for each record is the name of the state.

The Dictionary Data Type

void Dictionary::insert(<u>The key for the new item</u>, <u>The new item</u>);

- The insertion procedure for a dictionary has two parameters.

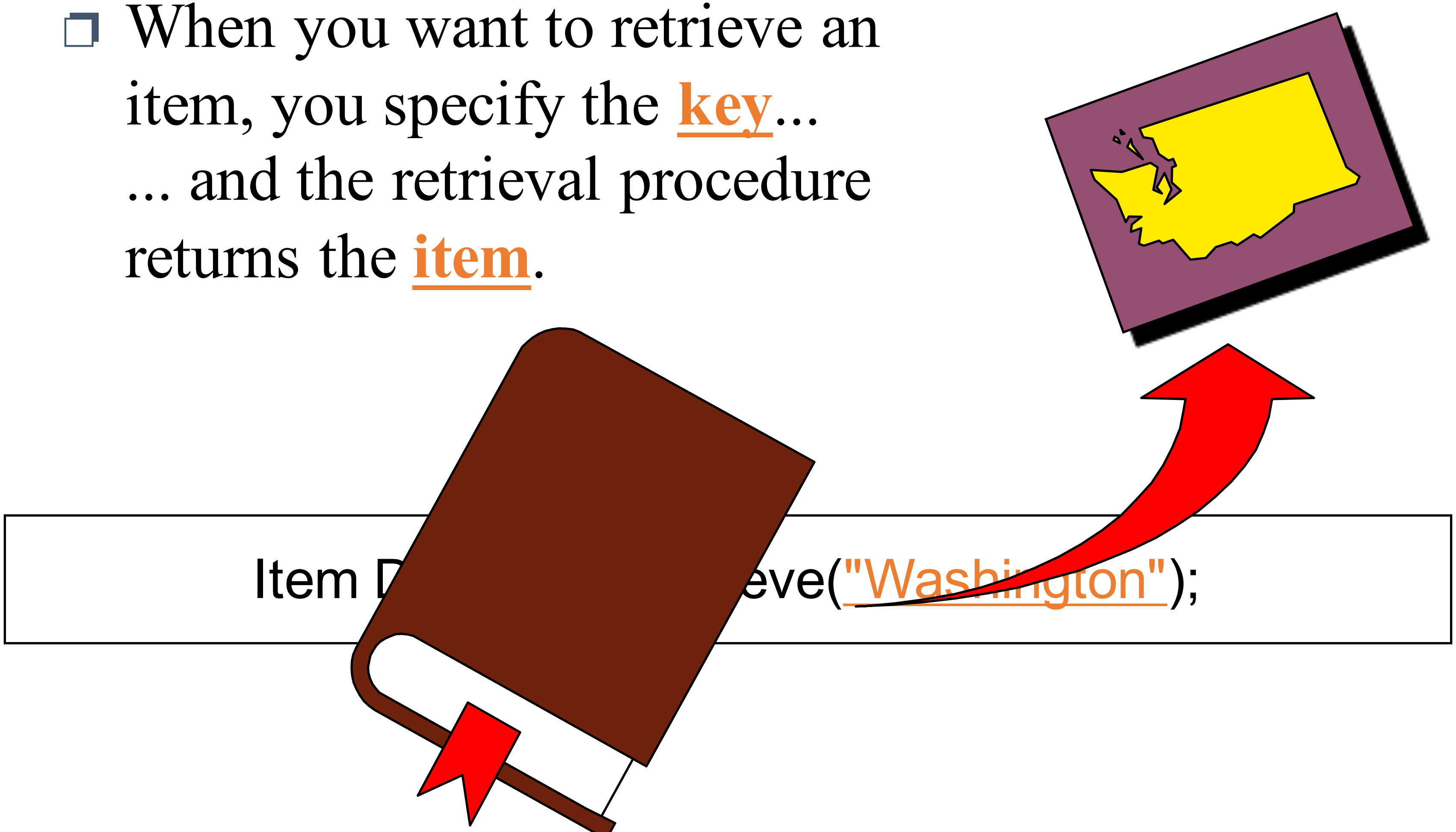
The Dictionary Data Type

- When you want to retrieve an item, you specify the **key**...

```
Item Dictionary::retrieve("Washington");
```

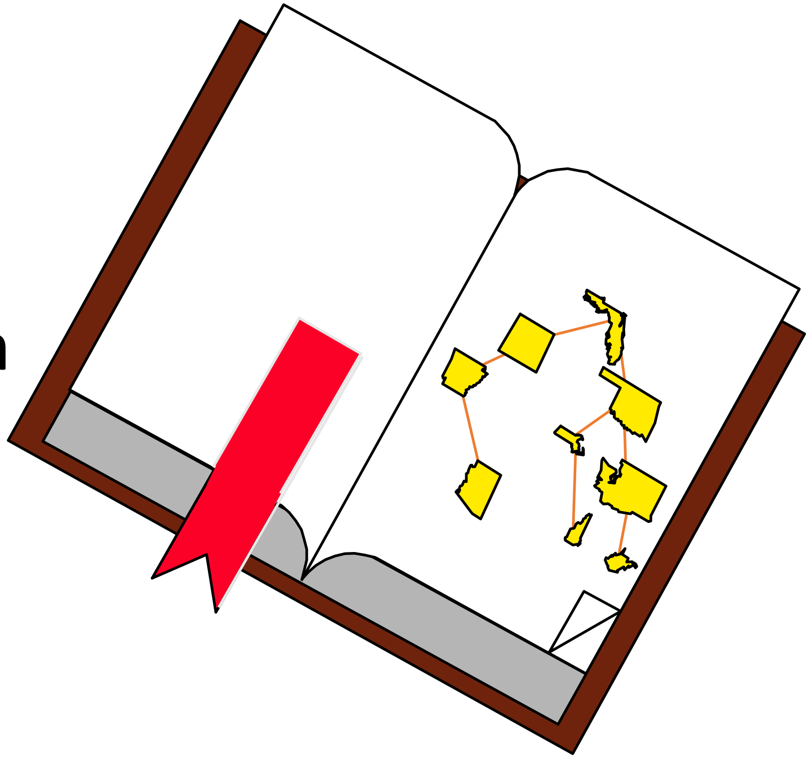
The Dictionary Data Type

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



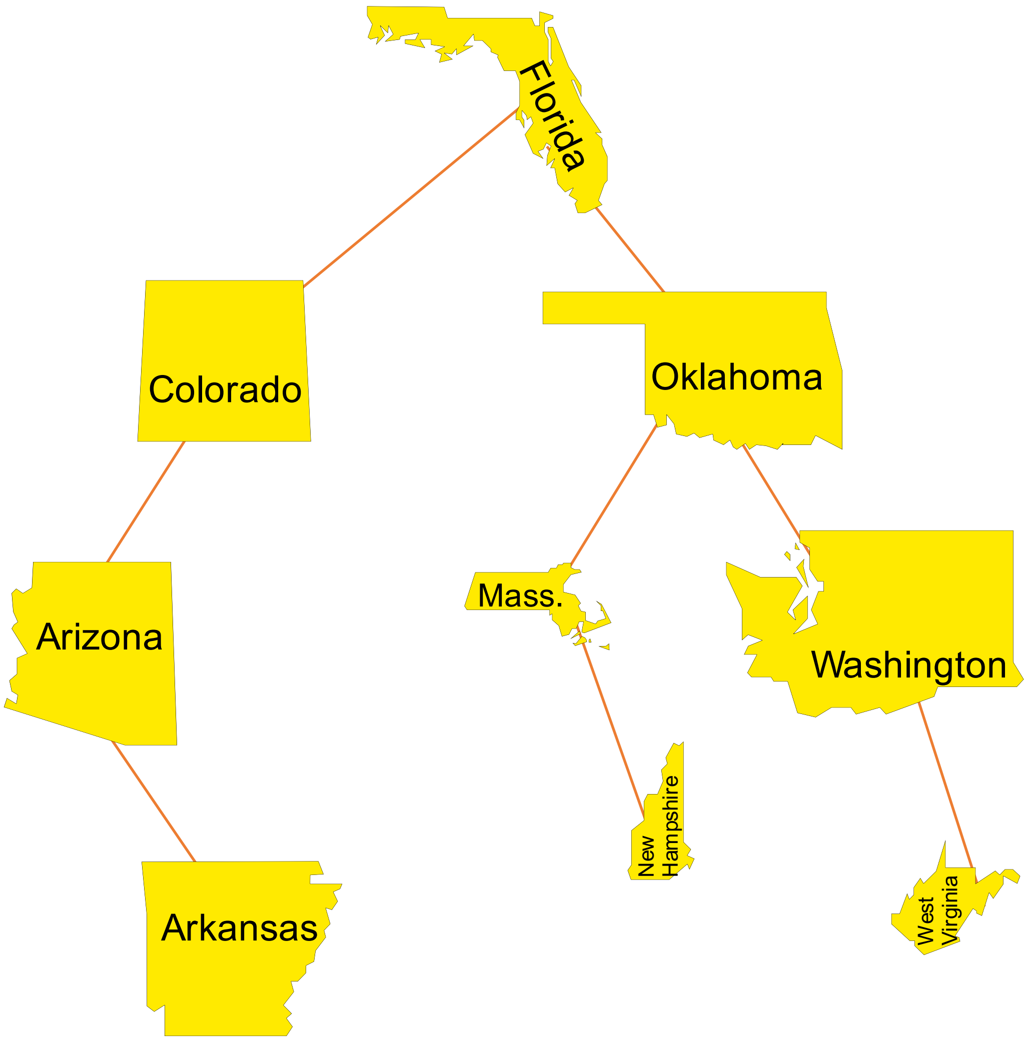
The Dictionary Data Type

- 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

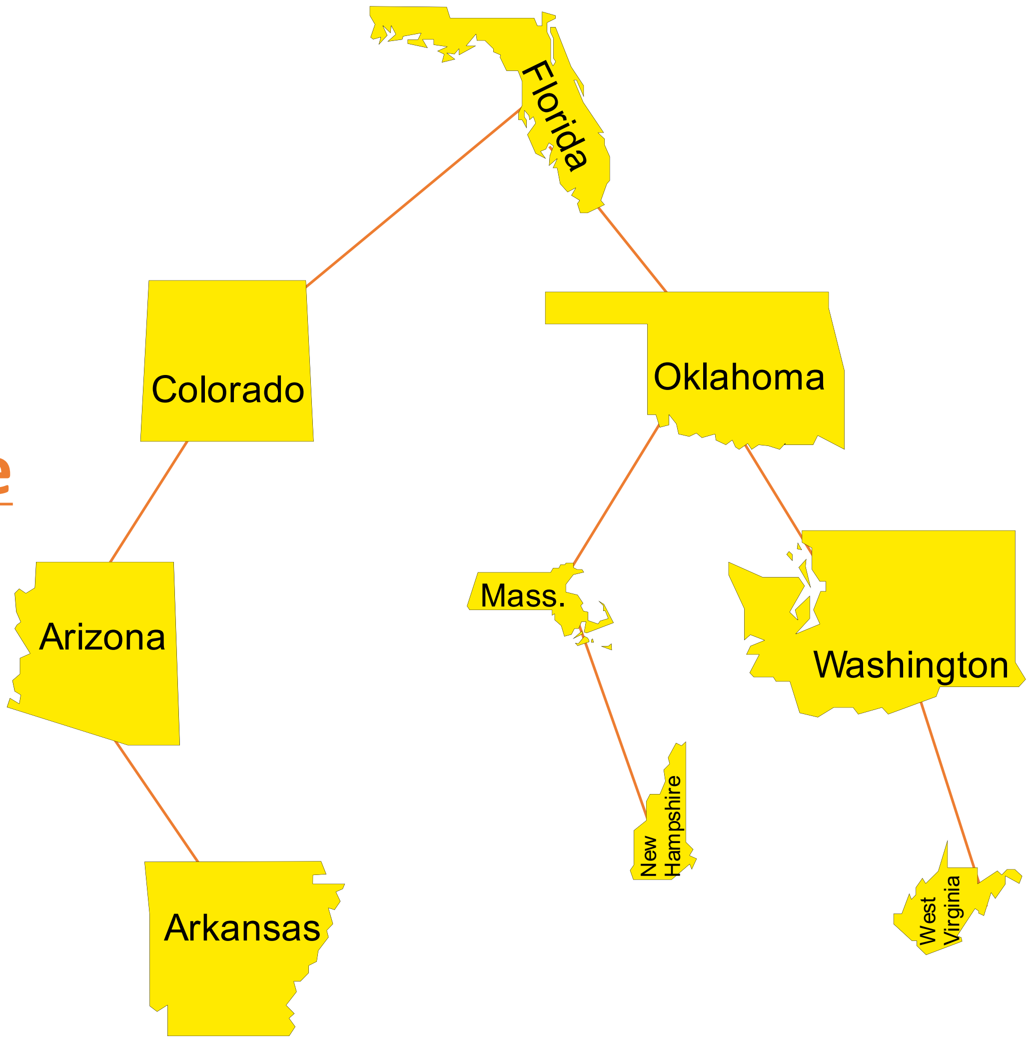
The data in the dictionary will be stored in a binary tree, with each node containing an item and a key.



A Binary Search Tree of States

Storage rules:

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

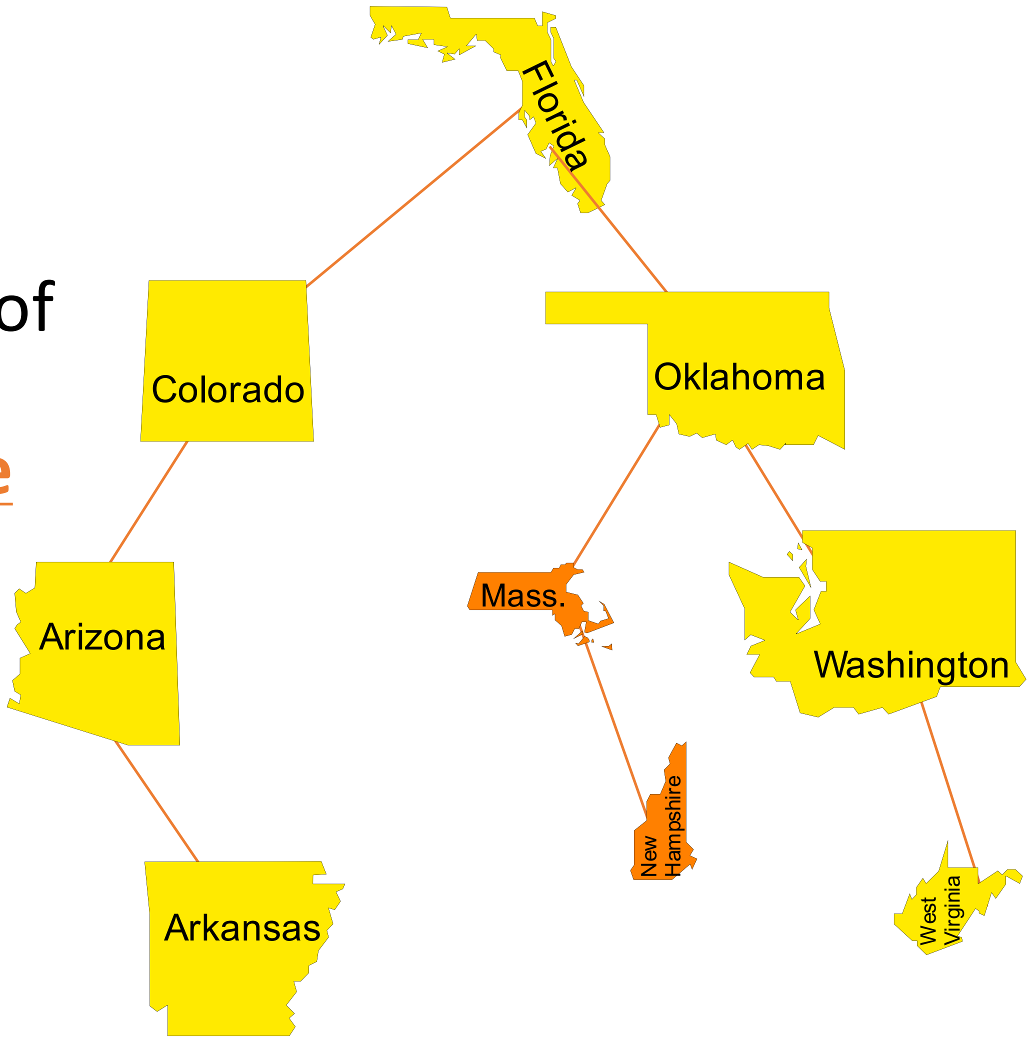


A Binary Search Tree of States

Storage rules:

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

Example:
' Massachusetts' and
' New Hampshire'
are alphabetically before 'Oklahoma'

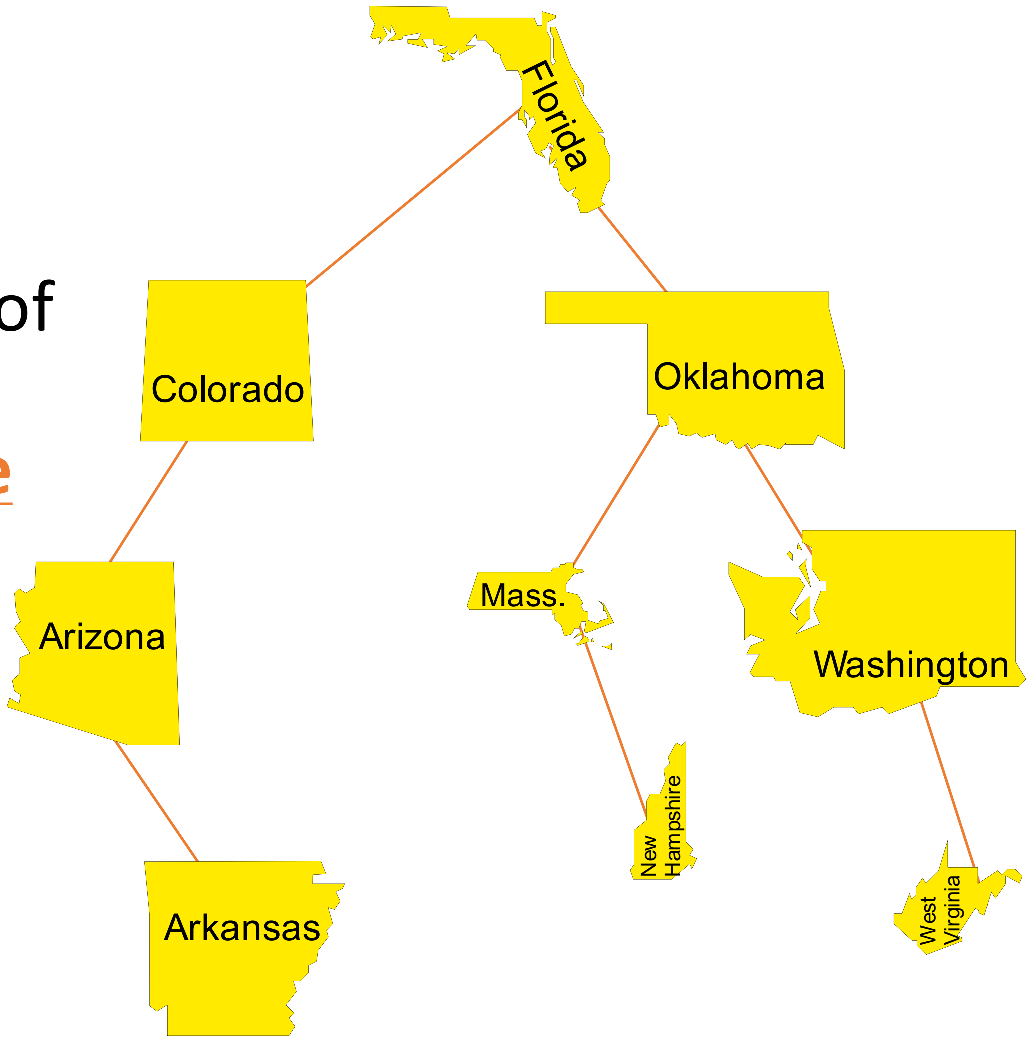


A Binary Search Tree of States

Storage rules:

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

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

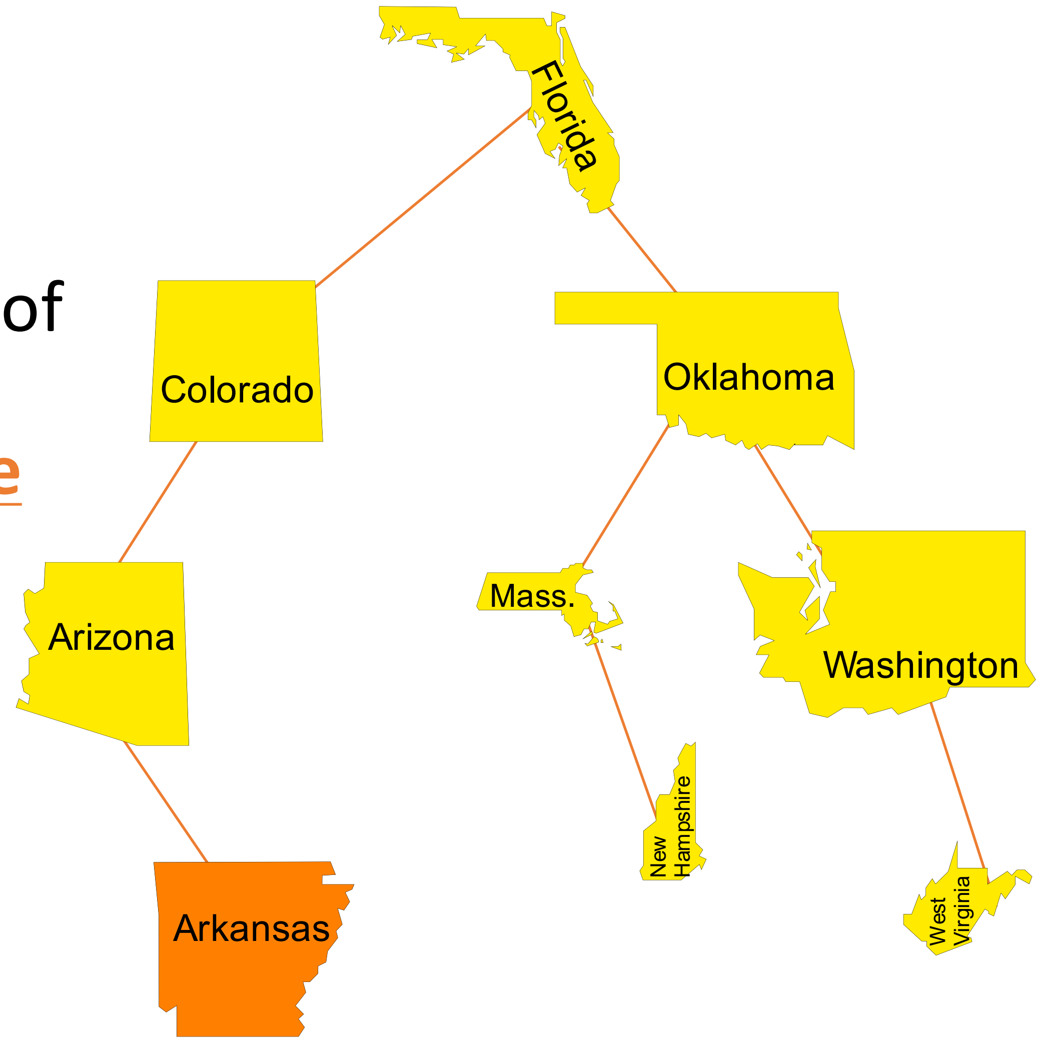


A Binary Search Tree of States

Storage rules:

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

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



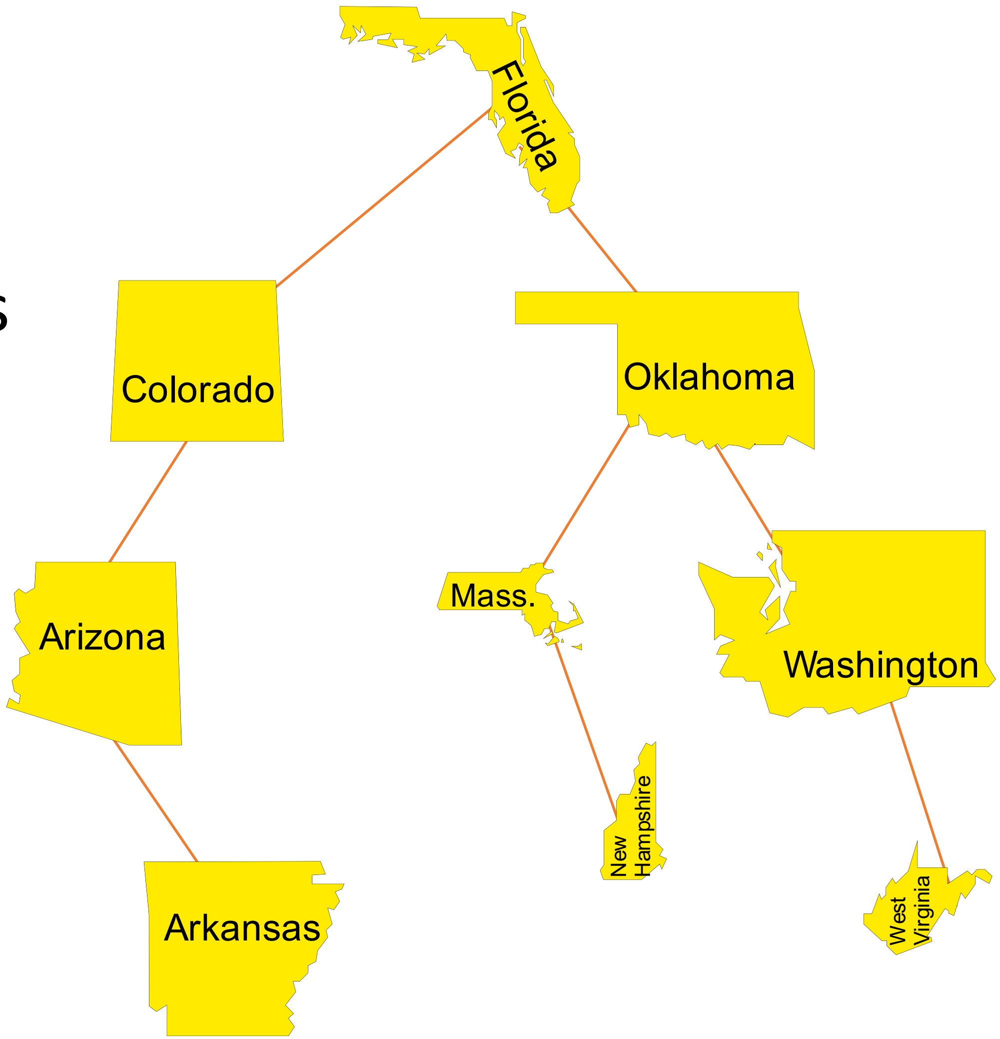
Retrieving Data

Start at the root.

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

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

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



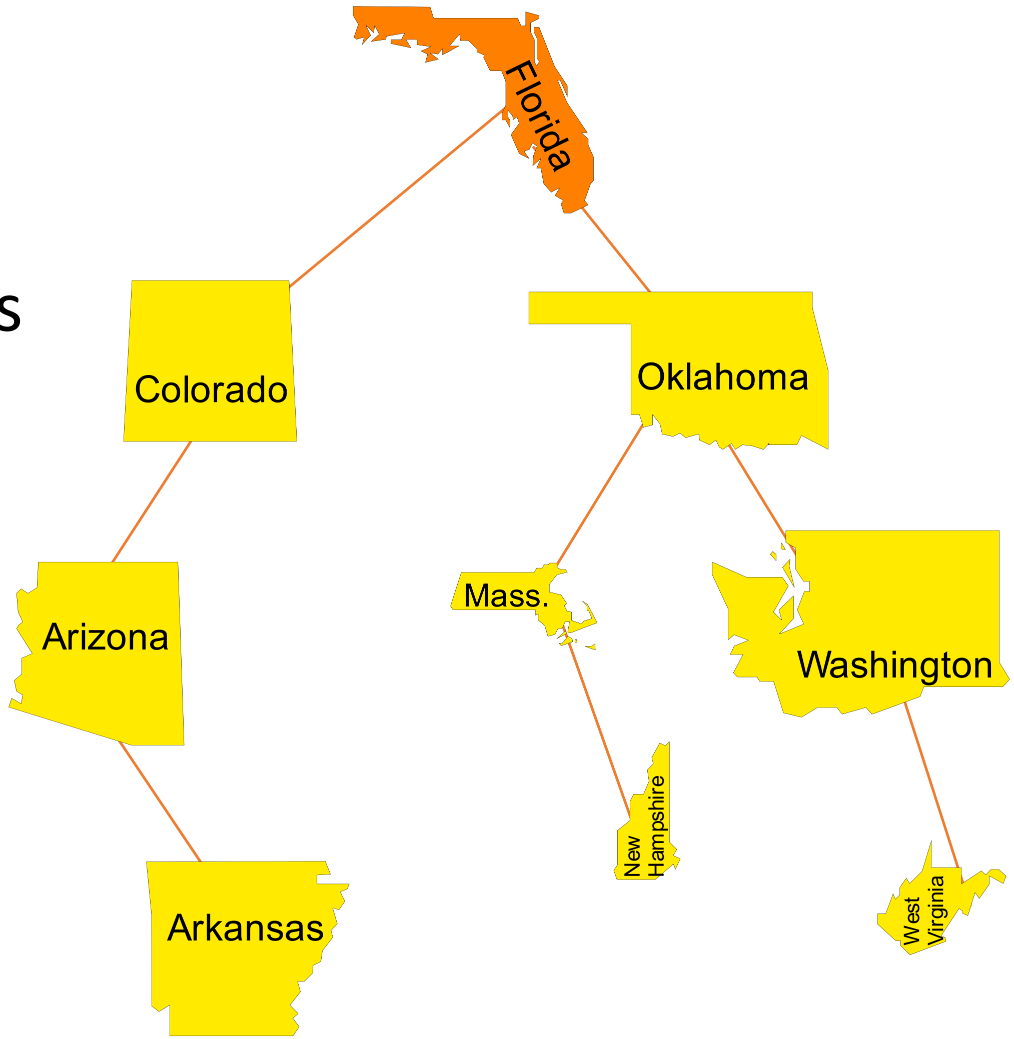
Retrieve ' New Hampshire'

Start at the root.

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

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

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



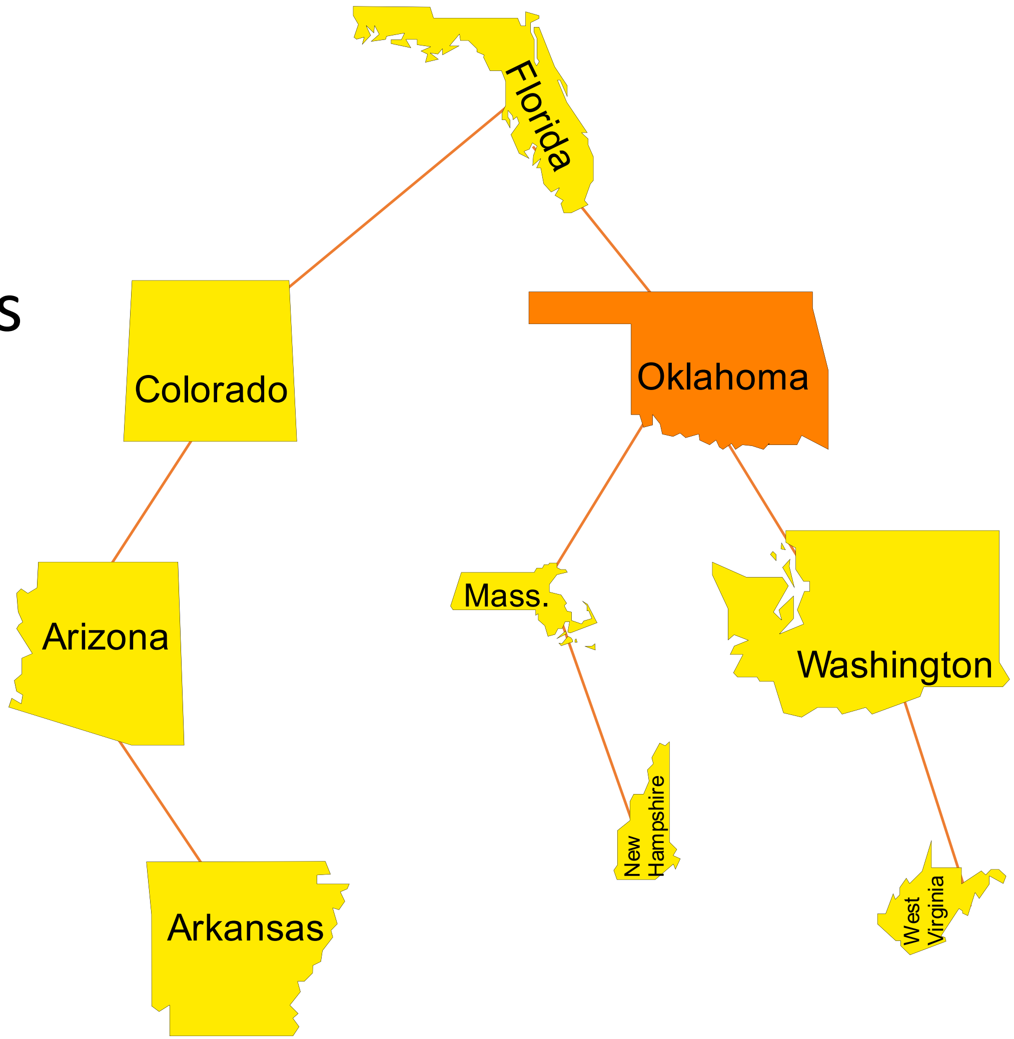
Retrieve 'New Hampshire'

Start at the root.

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

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

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



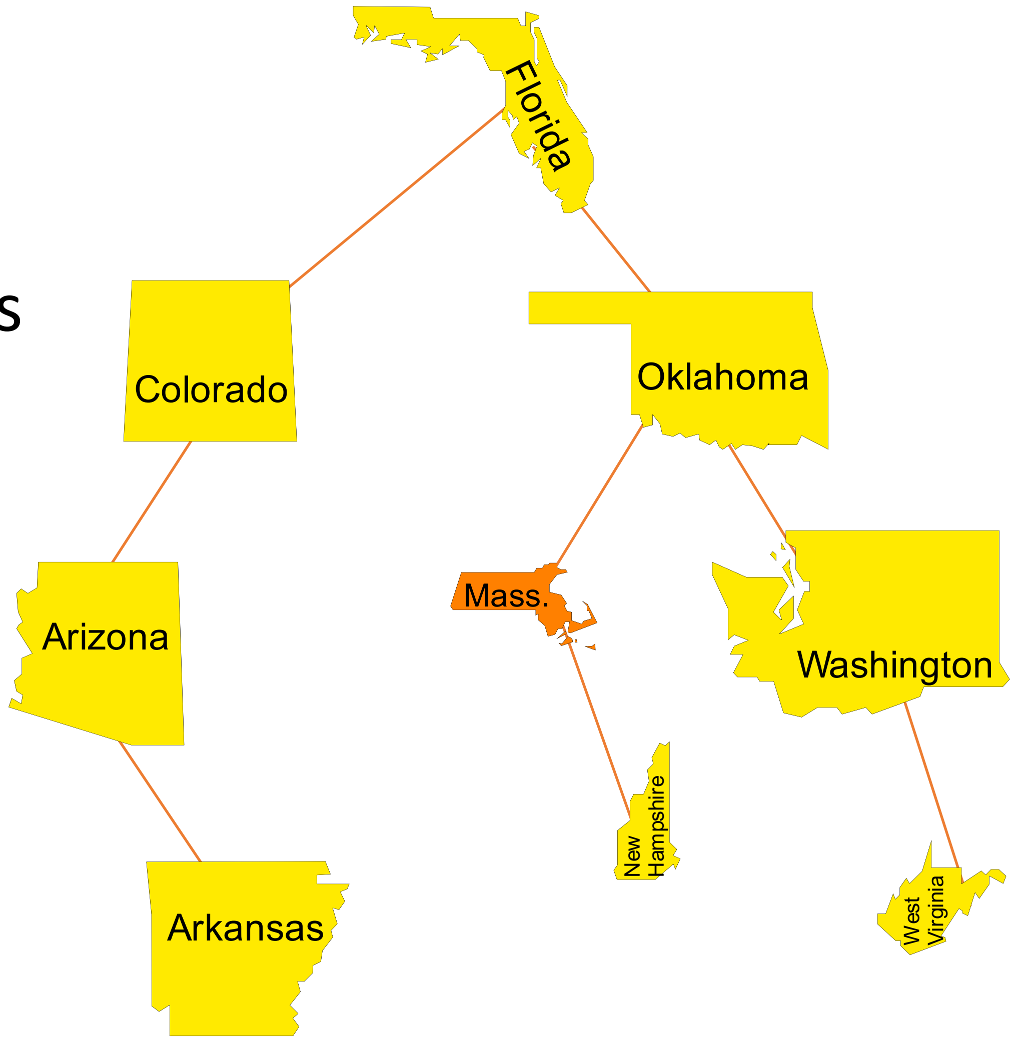
Retrieve 'New Hampshire'

Start at the root.

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

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

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



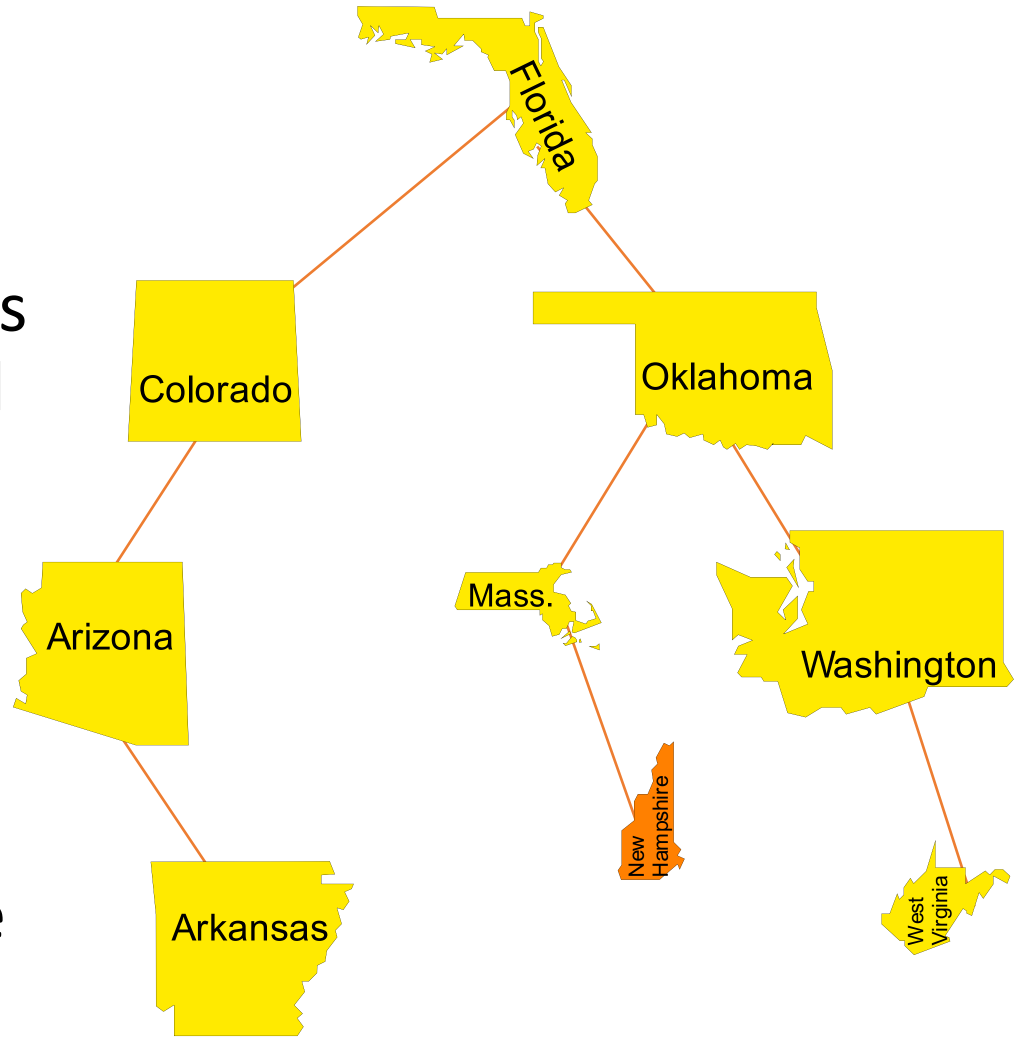
Retrieve 'New Hampshire'

Start at the root.

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

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

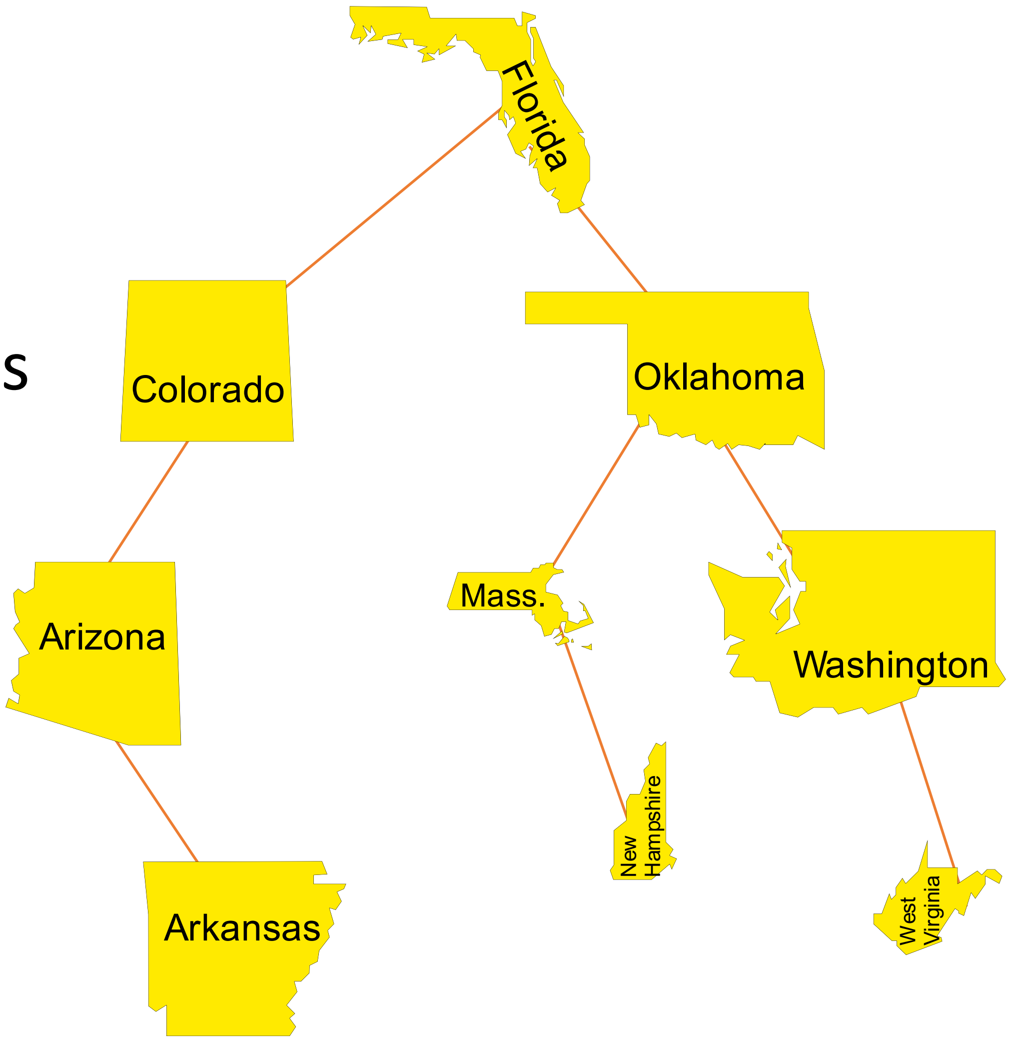
❸ If the current node's key is too **small**, move **right** and repeat 1-3.



Adding a New Item with a Given Key

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

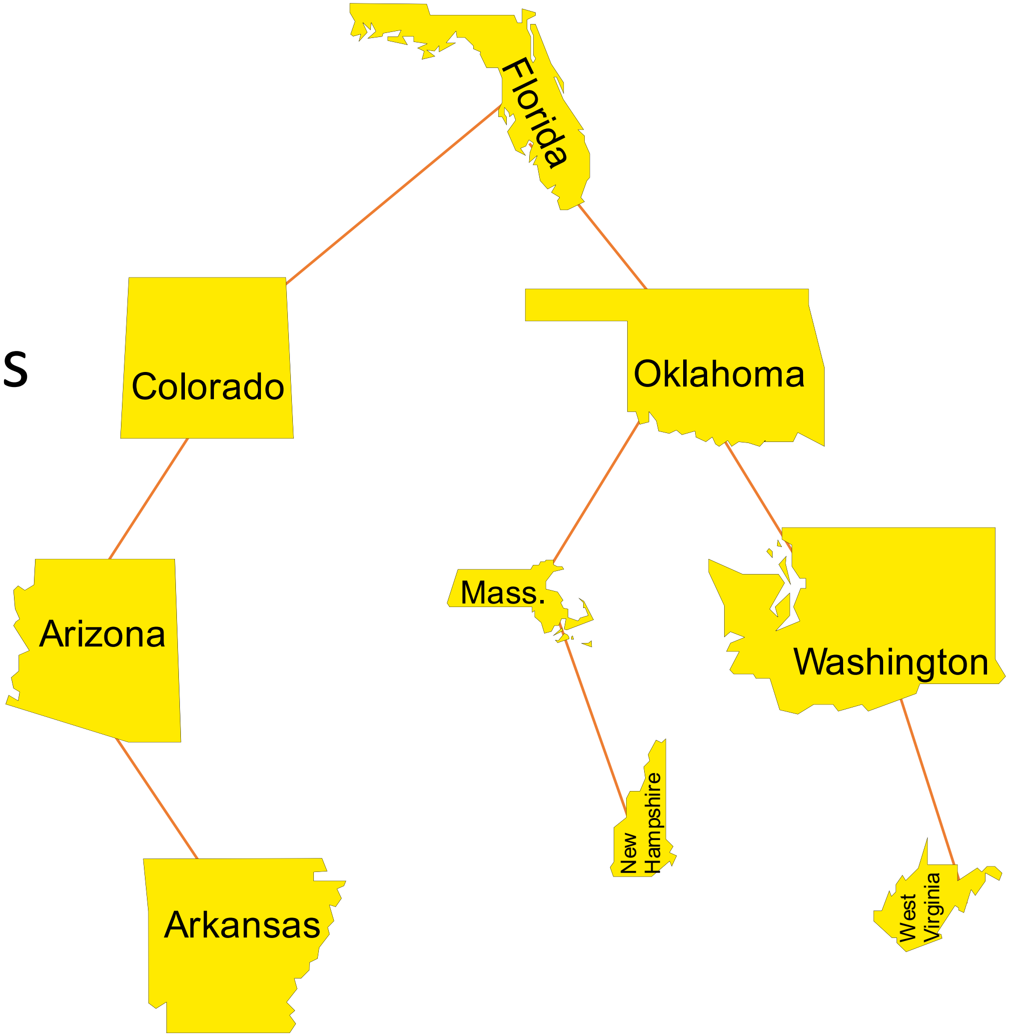
❷ Add the new node at the spot where you would have moved to if there had been a node.



Adding

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

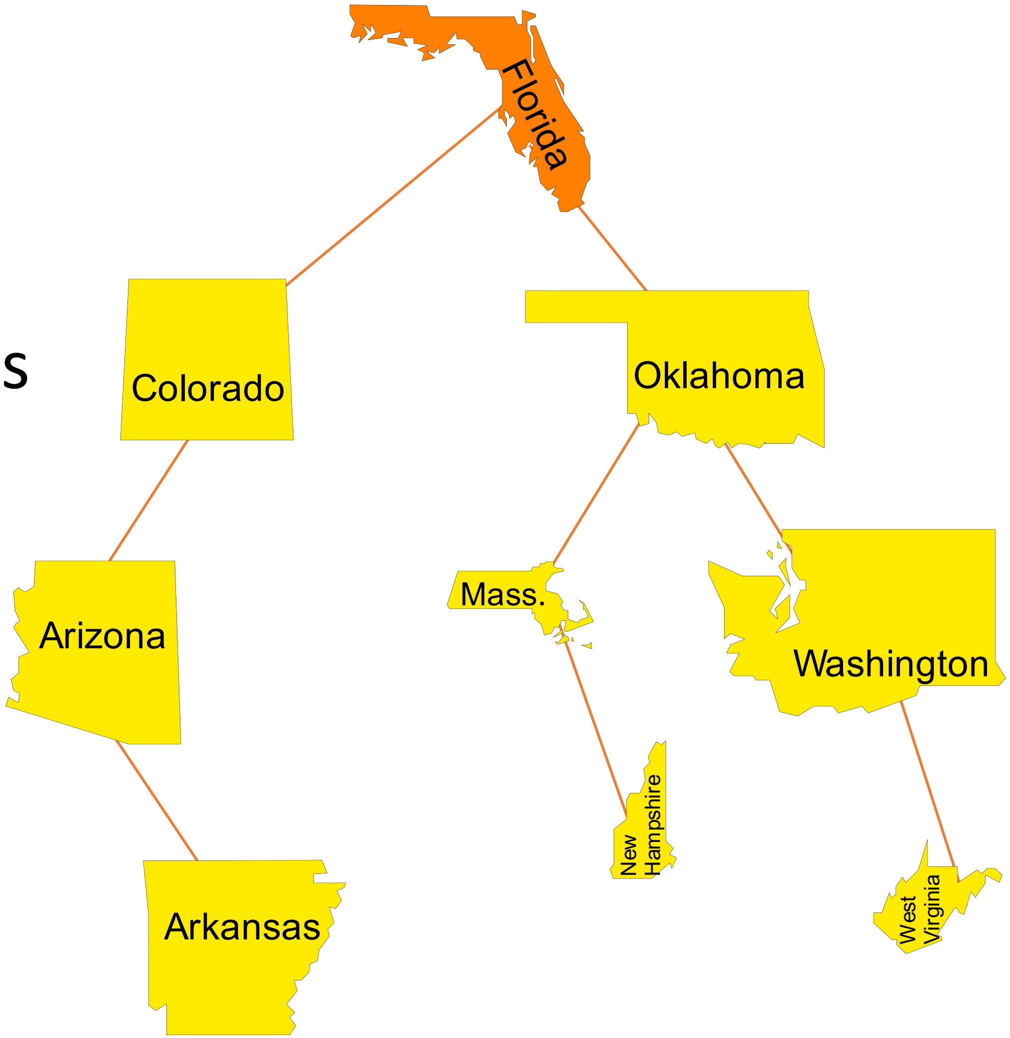
❷ Add the new node at the spot where you would have moved to if there had been a node.



Adding

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

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



Adding

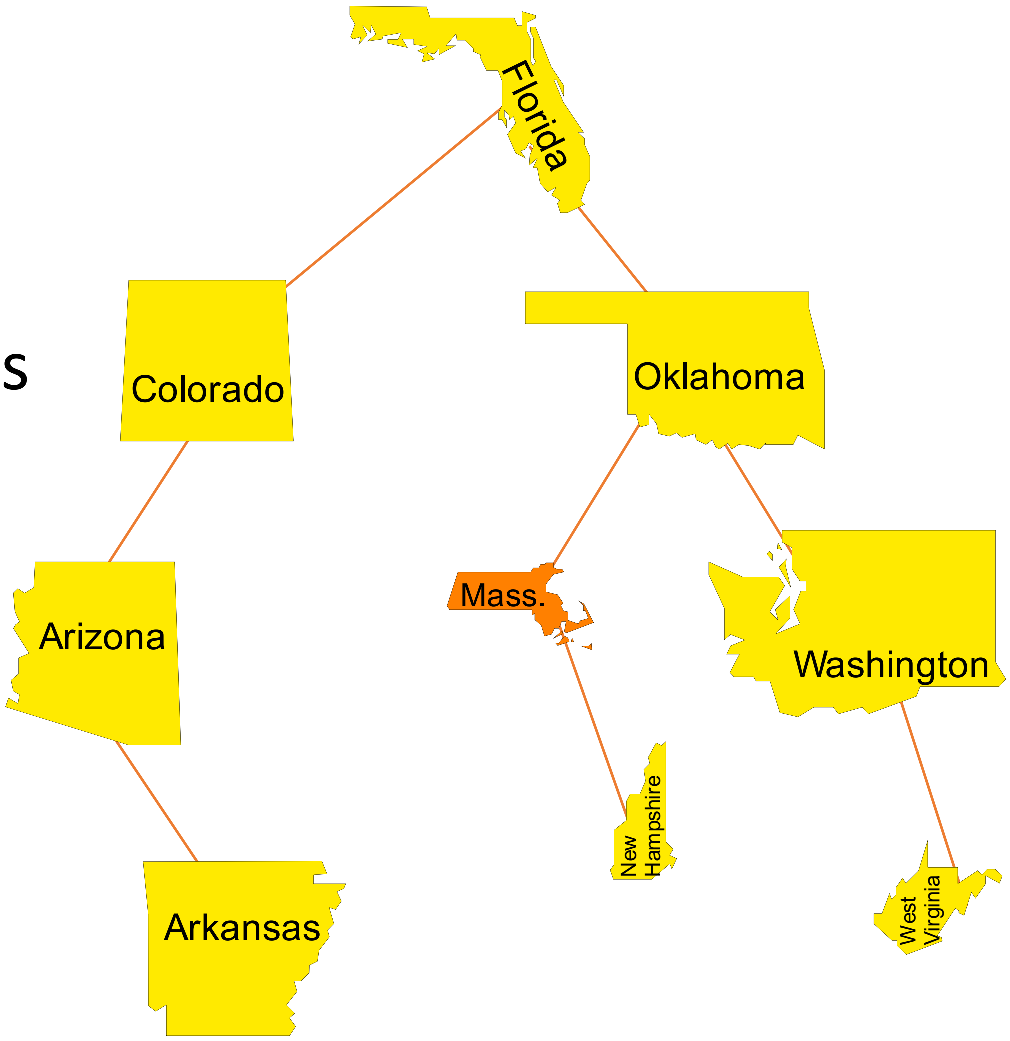
❶ Pretend that you are trying to find the key, but stop when there is no node to move to.

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

Adding

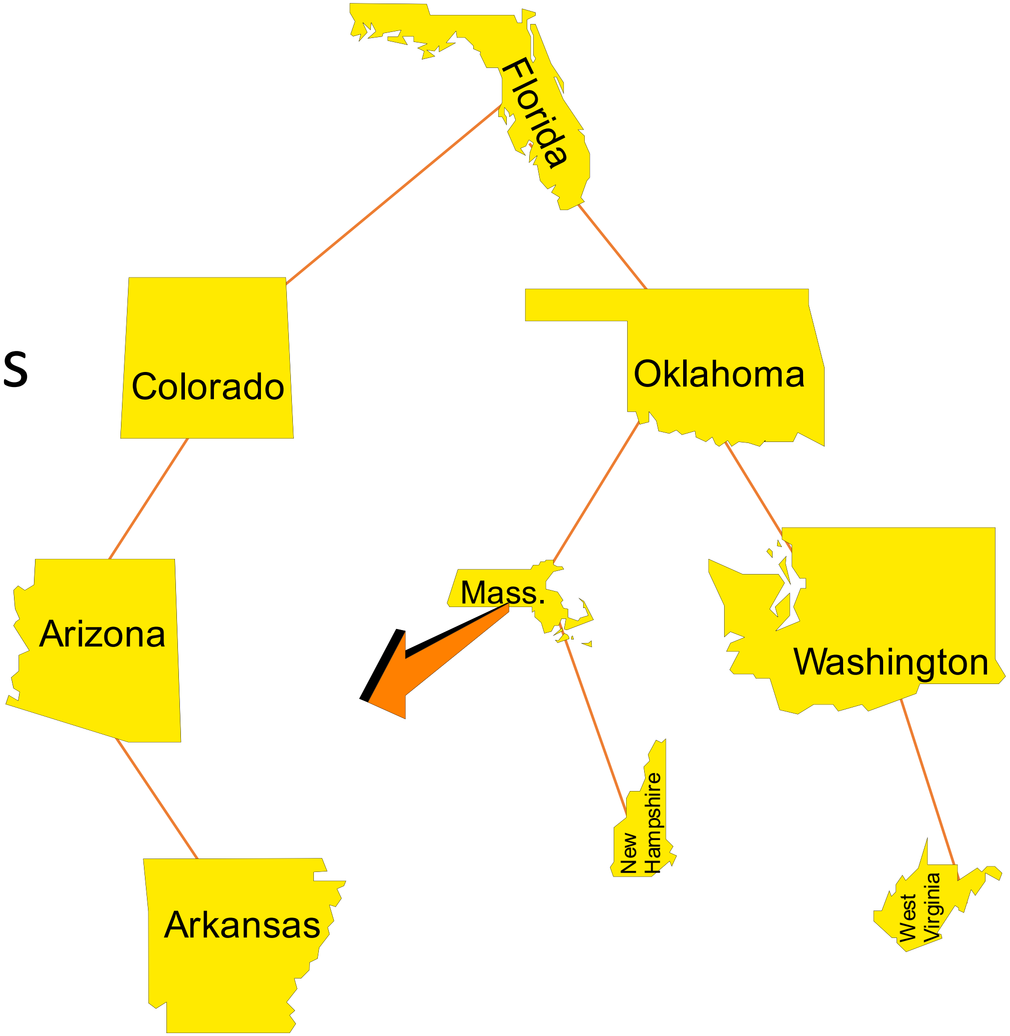
❶ Pretend that you are trying to find the key, but stop when there is no node to move to.

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



Adding

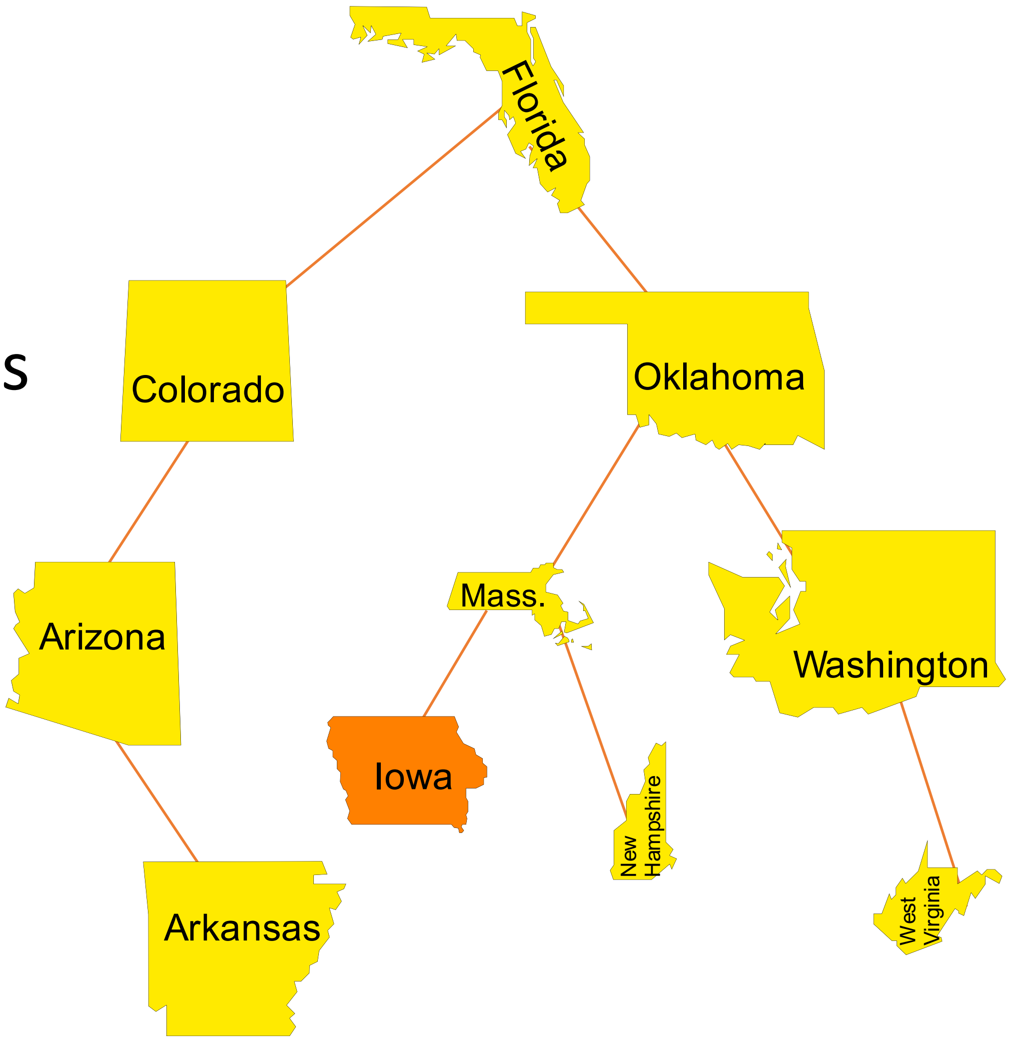
1. Pretend that you are trying to find the key, but stop when there is no node to move to.
2. Add the new node at the spot where you would have moved to if there had been a node.



Adding

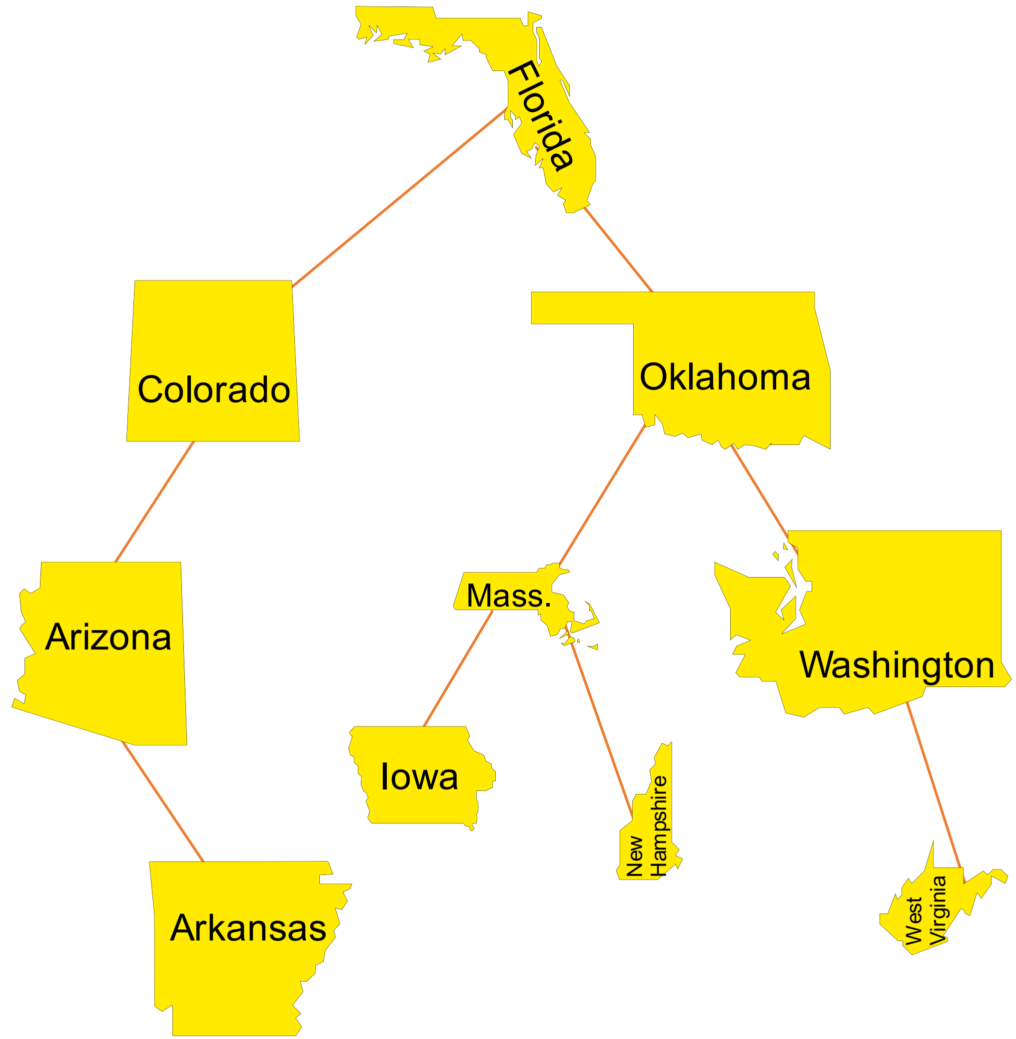
➊ Pretend that you are trying to find the key, but stop when there is no node to move to.

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



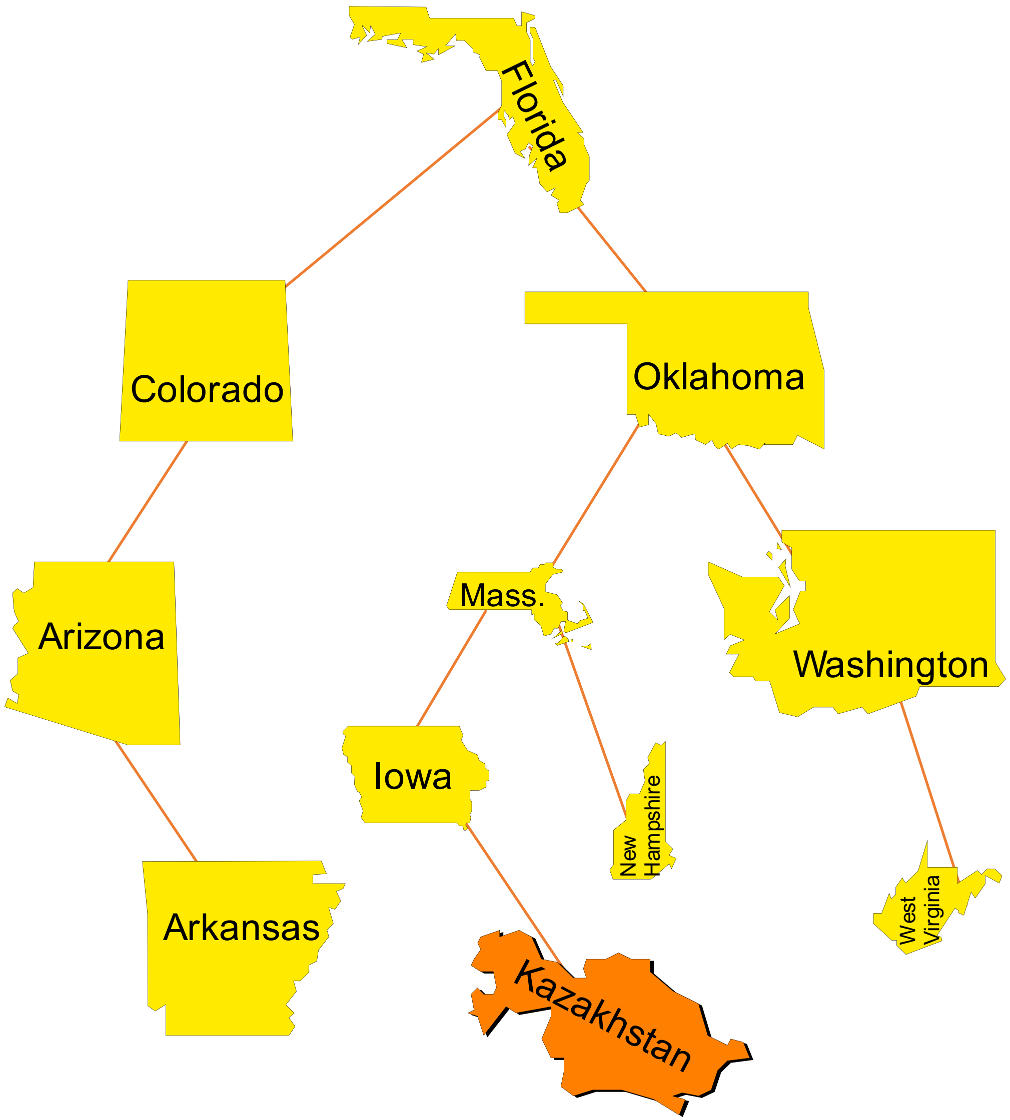
Adding

Where would you add this state?



Adding

Kazakhstan is the new right child of Iowa?

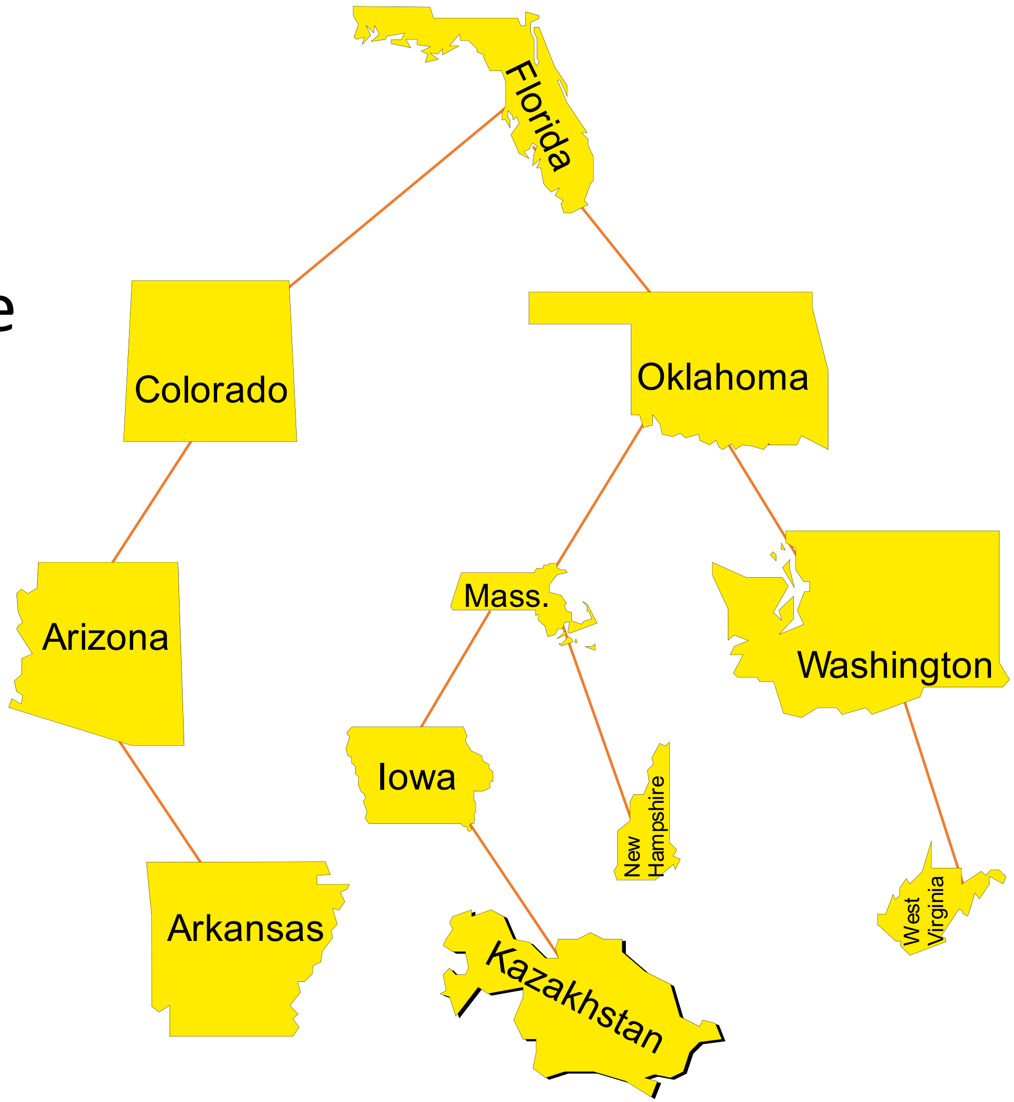


Removing an Item with a Given Key

❶ Find the item.

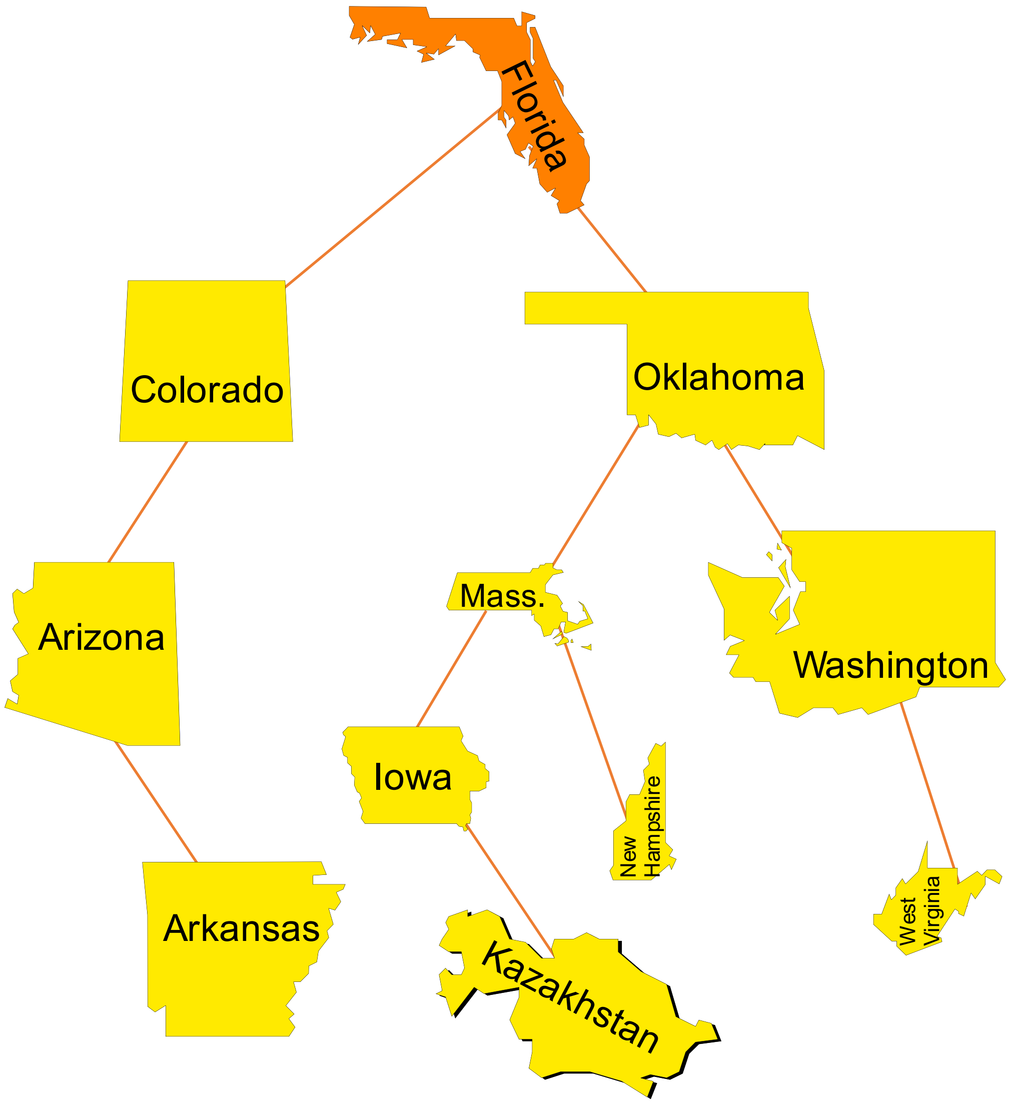
❷ If necessary, swap the item with one that is easier to remove.

❸ Remove the item.

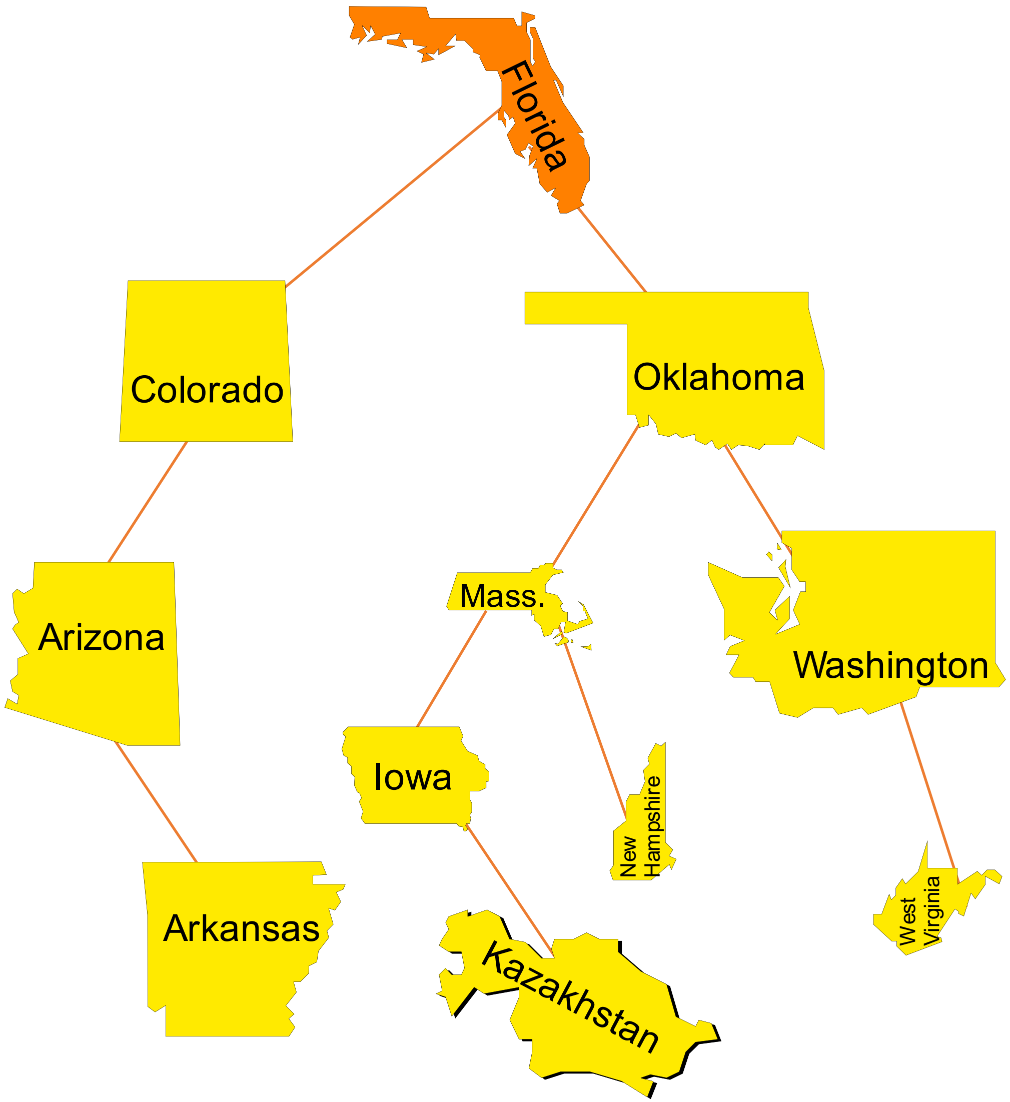


Removing 'Florida'

❶ **Find** the item.

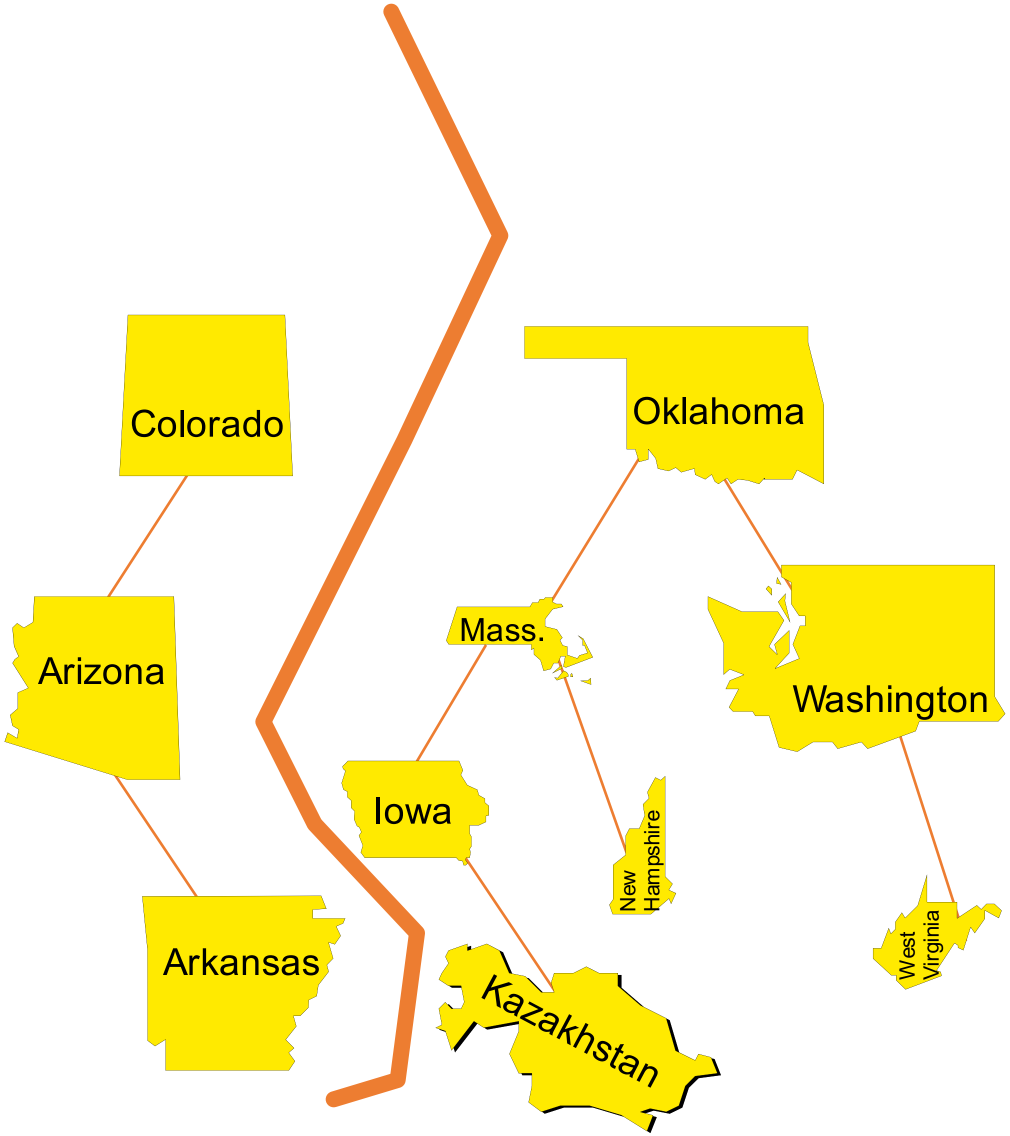


Removing 'Florida'



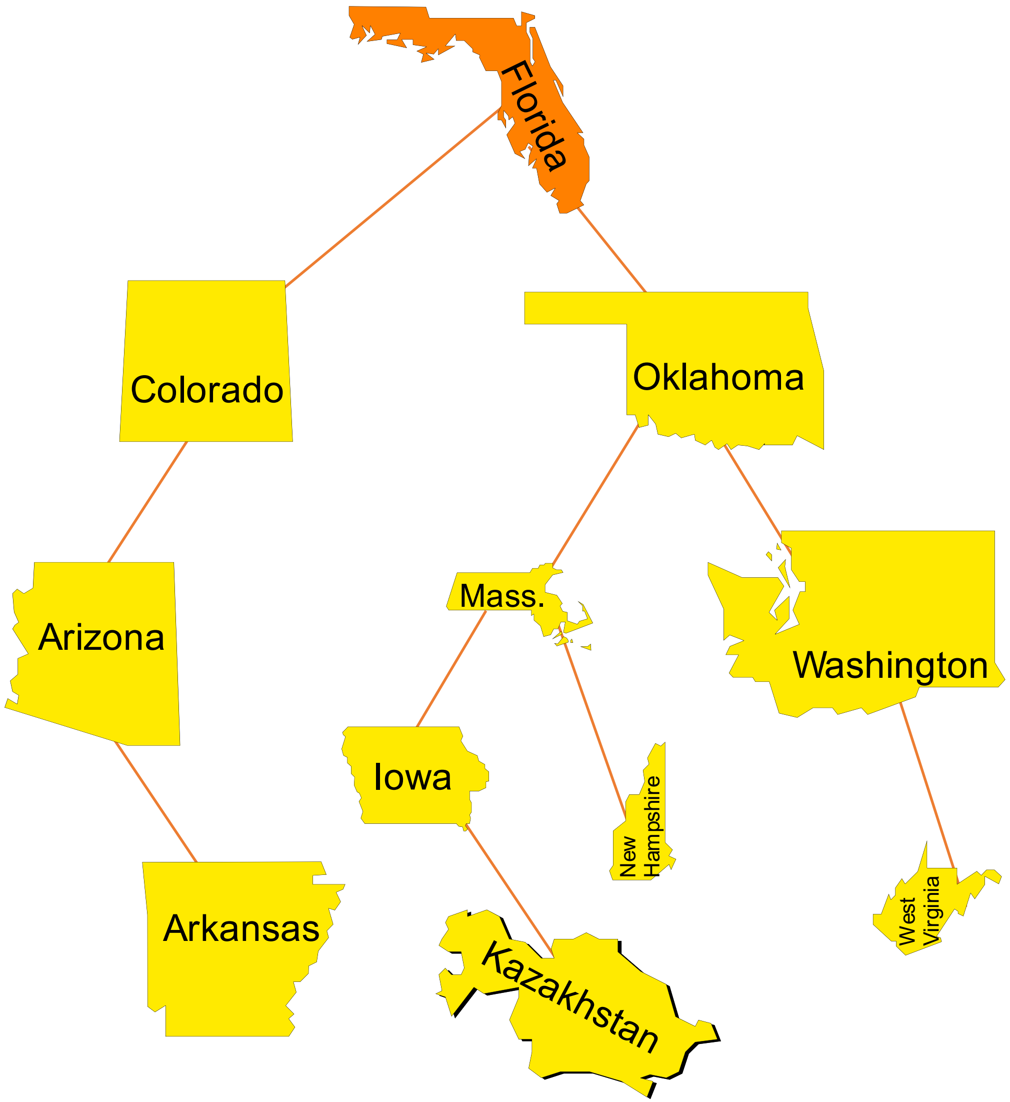
Florida cannot be removed at the moment...

Removing 'Florida'



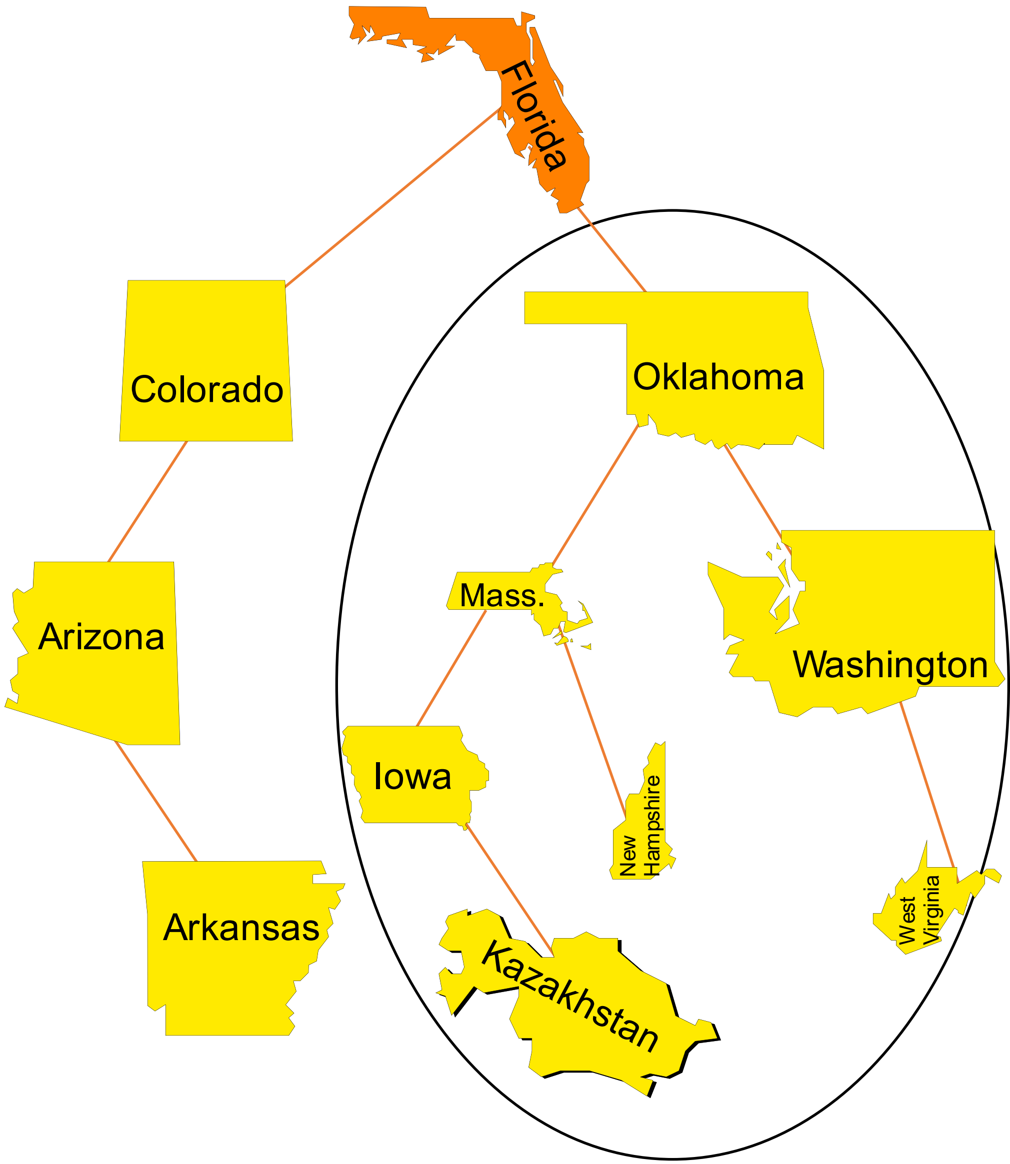
... because removing Florida would break the tree into two pieces.

Removing 'Florida'



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

Removing 'Florida'

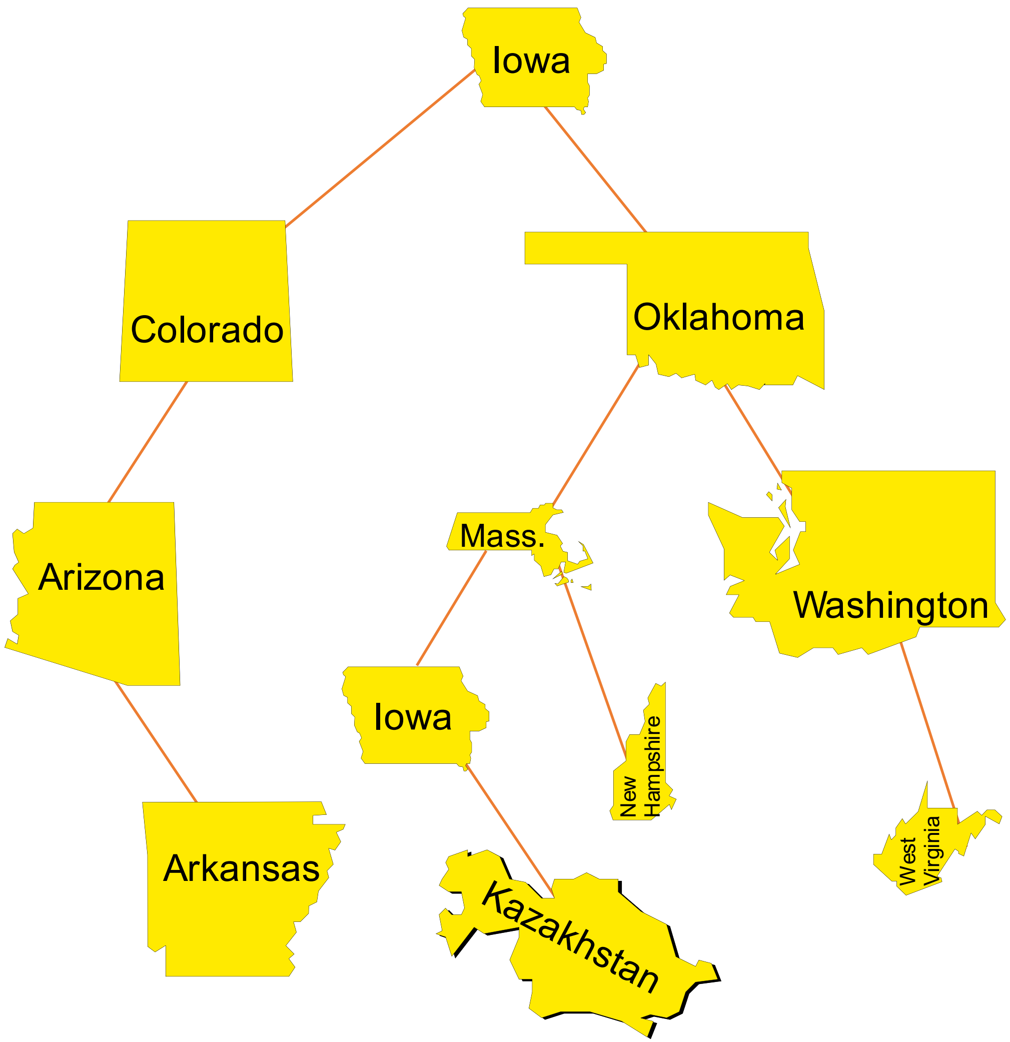


For the rearranging, take the **smallest** item in the right subtree...

Work for multi-set?

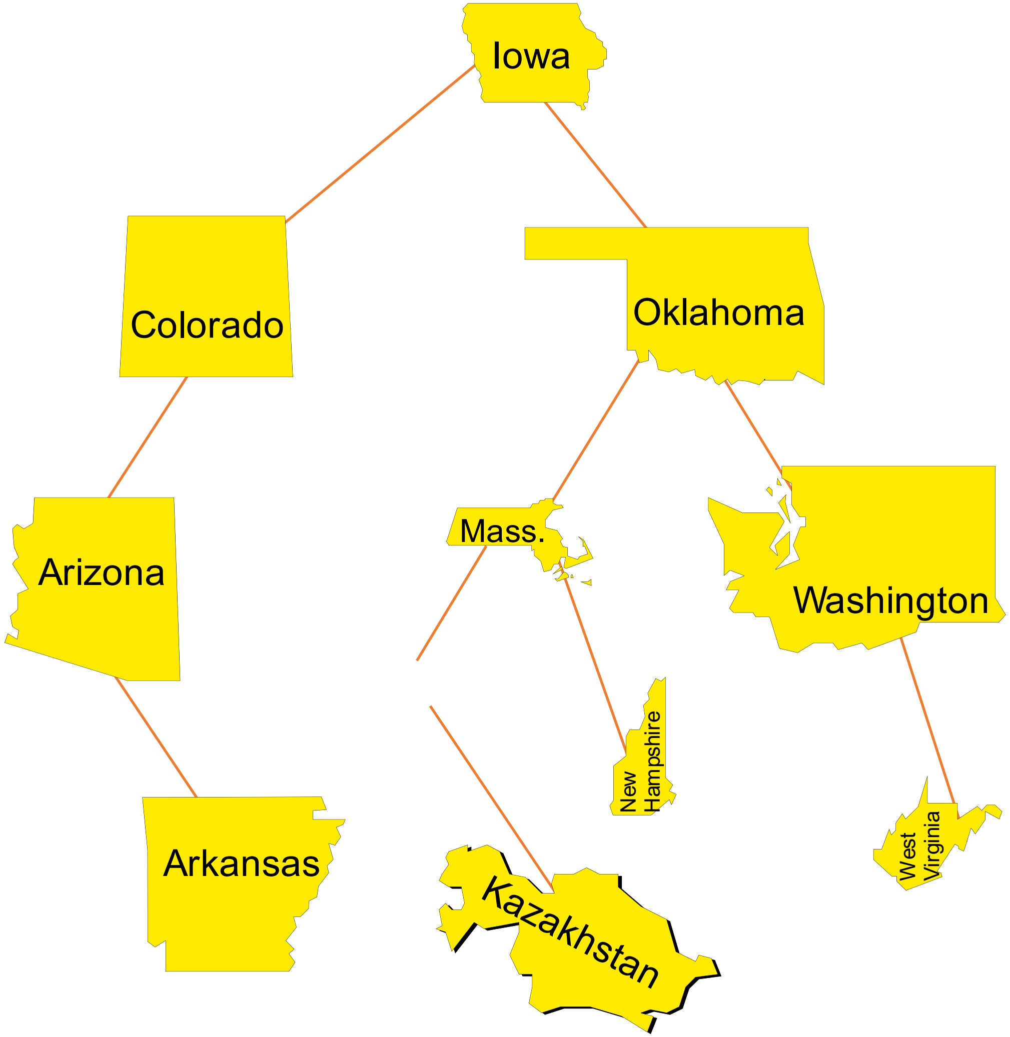
Removing 'Florida'

...**copy** that smallest item onto the item that we're removing...

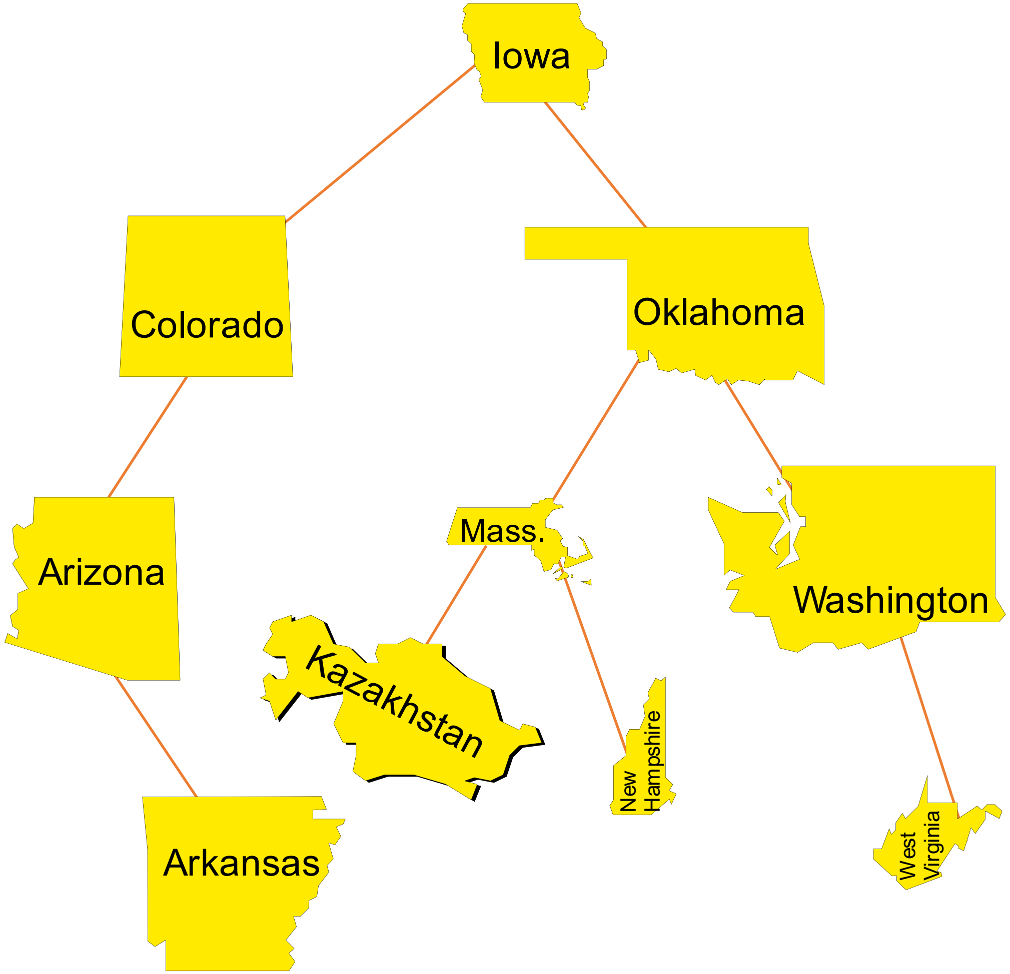


Removing 'Florida'

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

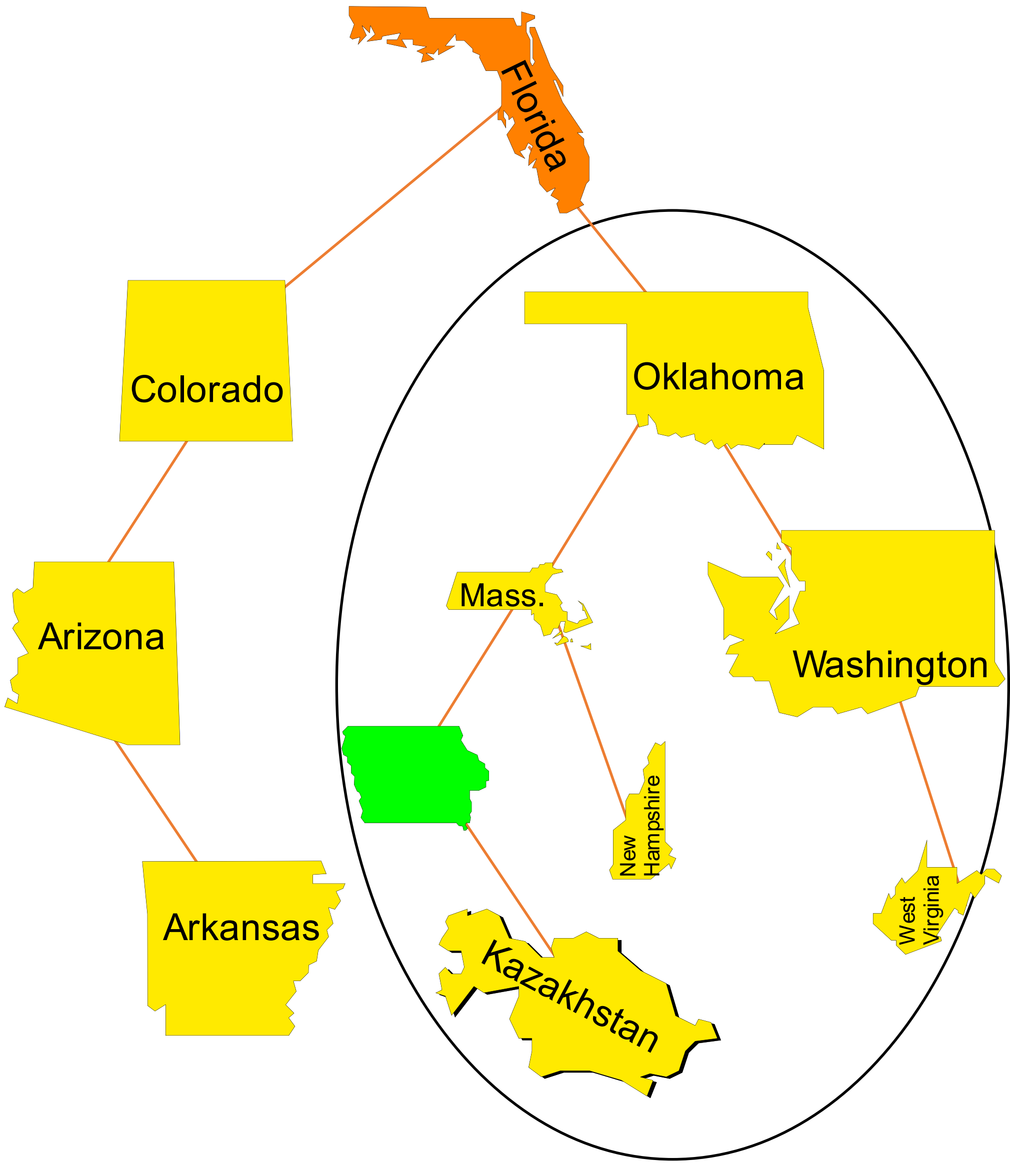


Removing 'Florida'



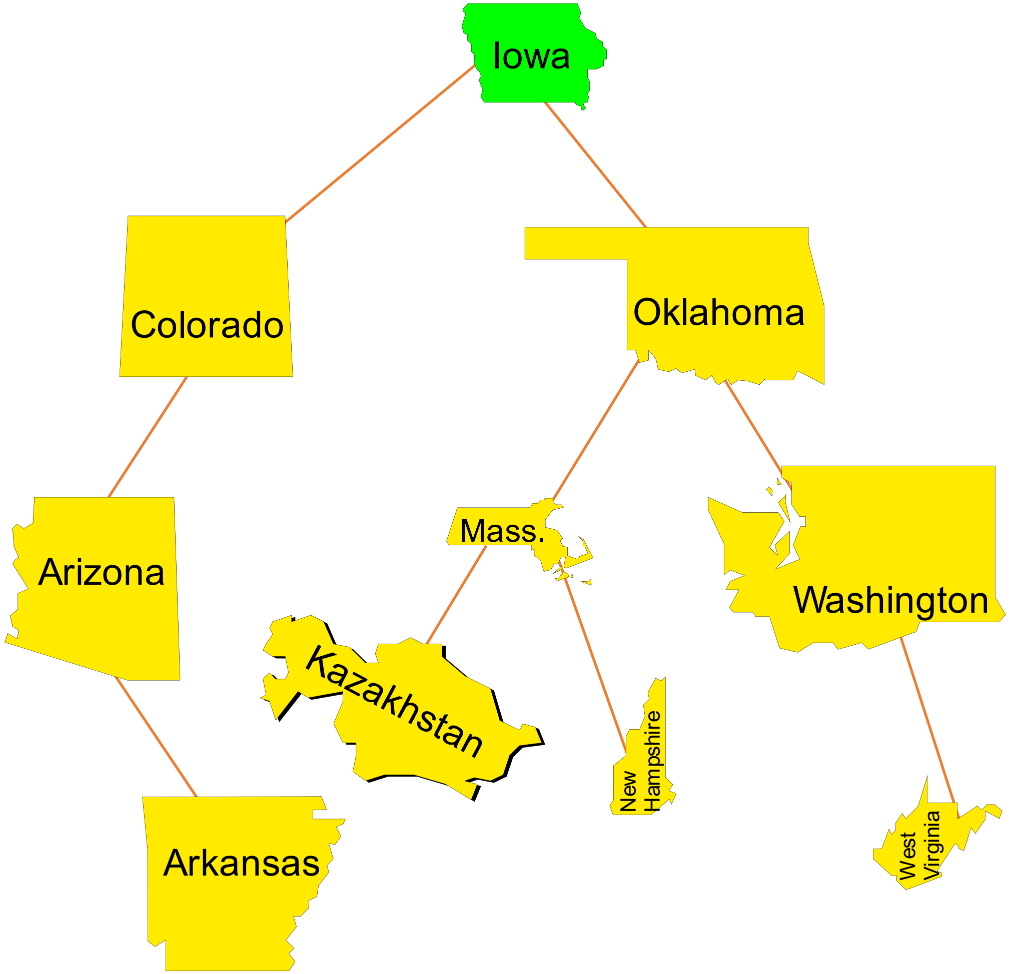
... and reconnect the tree

Removing 'Florida'



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

Removing 'Florida'



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

Removing an Item with a Given Key

❶ Find the item.

❷ 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.

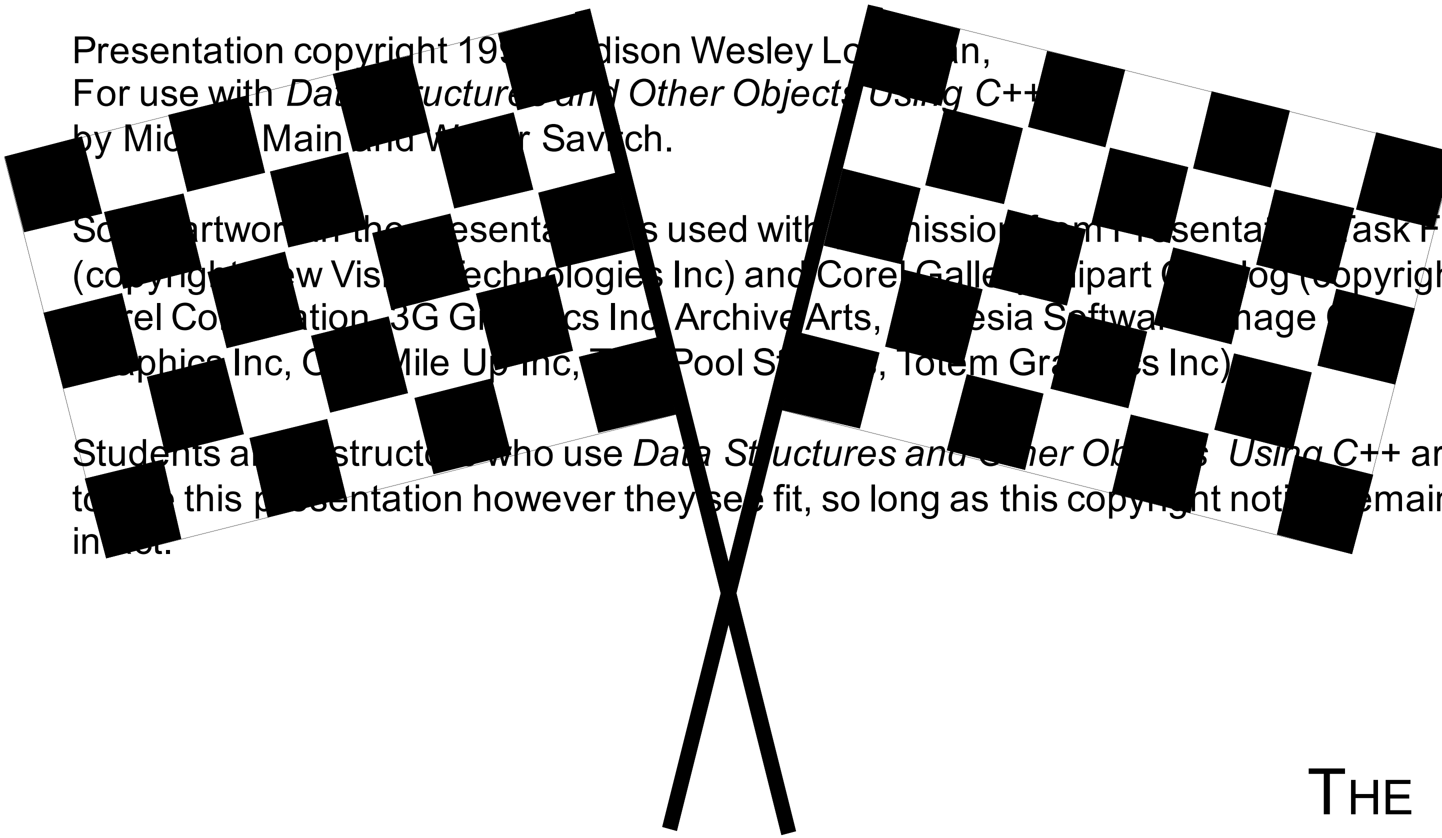
Summary

- Binary search trees are a good implementation of data types such as sets, bags, and dictionaries.
- Searching for an item is generally quick since you move from the root to the item, without looking at many other items.
- Adding and deleting items is also quick.
- But as you'll see later, it is possible for the quickness to fail in some cases -- can you see why?

Assignment

- Read Section 10.5
- Assignment 6 – Bag class with a BST
  - Memeber functions
    - void insert(const Item& entry);
    - size_type count (const Item& target);
  - Non-member functions
    - viod bst_remove_all(binary_tree_node<Item>*& root const Item& target);
    - void bst_remove_max(binary_tree_node<Item>*& root, Item& removed);

Deadline: Monday, November 28, 2016



THE END