Graph Data Structure

Session-26

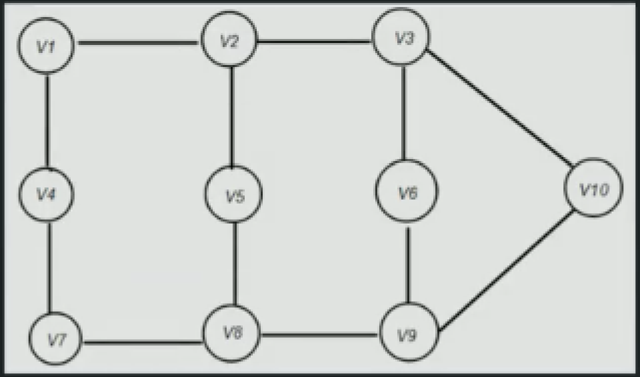
Dr. Subhra Rani Patra
SCOPE, VIT Chennai

What we will learn ?

- ✓What / Why of graph
- ✓Lots of Terminologies
- ✓Types of Graphs
- ✓Graph Representation
- ✓Graph Traversal Techniques – BFS, DFS.
- ✓Single Source Shortest Path - BFS, Dijkstra,
- ✓All Pair Shortest Path - BFS, Dijkstra,
- ✓Minimum Spanning Tree – Prims, Kruskal.
- ✓What / Why / Pros & Cons / Practical uses / Comparison.

What is Graph:

- ✓Graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.
- ✓How does a typical graph look like ?

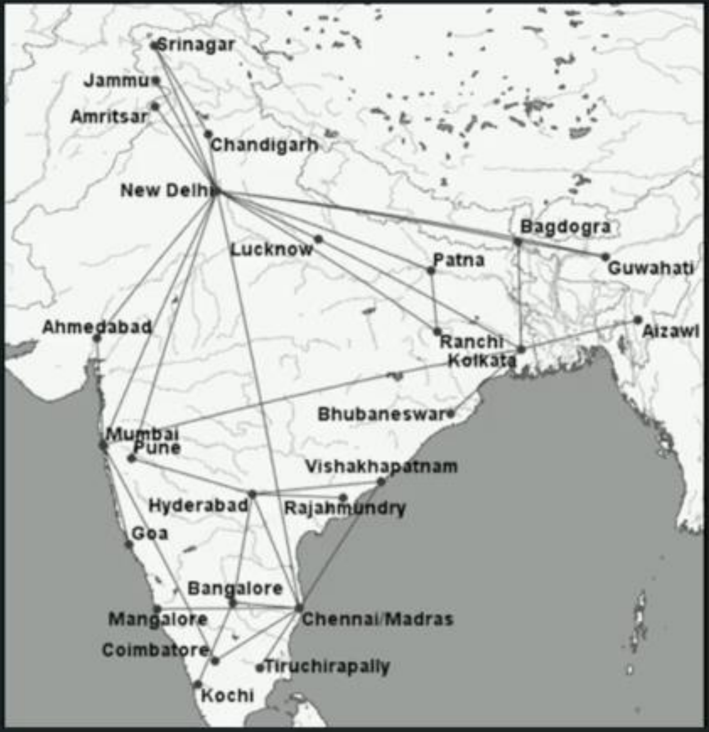


V = {v1, v2, v3, v4, v5, v6, v7, v8, v9, v10}

E = {v1v2, v2v3, v1v4, v4v7, v7v8, v2v5, v5v8, v3v6, v6v9, v8v9, v3v10, v9v10}

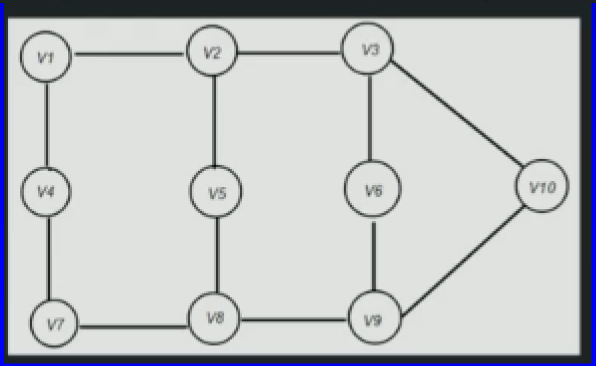
Why should we learn Graph ?

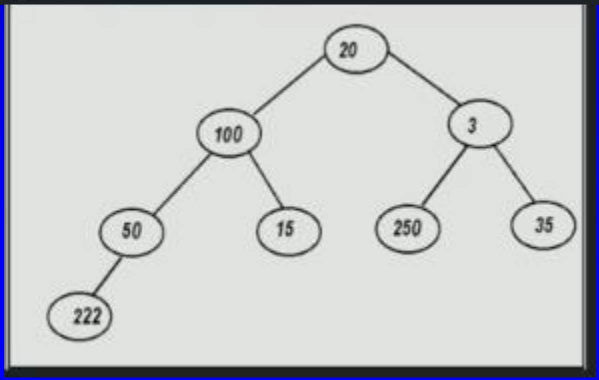
✓ Shortest path between cities



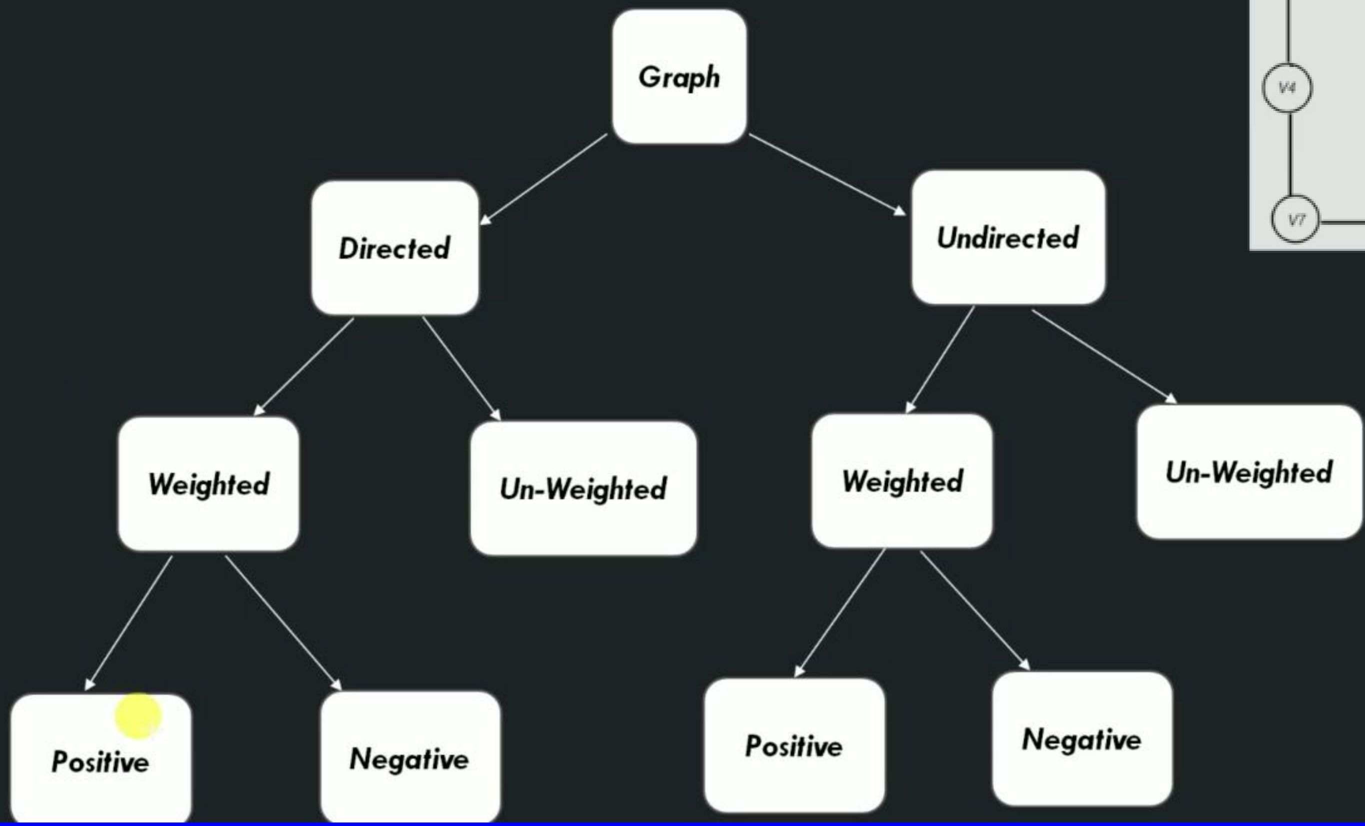
Some Terminologies:

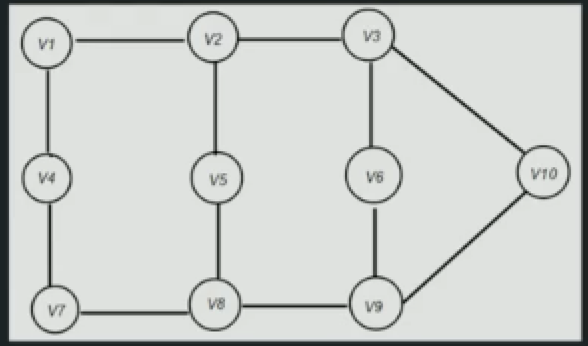
- ✓**Vertices**: Vertices are the nodes of the graph
- ✓**Edges**: Edges are the arcs that connect pairs of vertices
- ✓**UnWeighted Graph:** A graph not having a weight associated with any edge
- ✓**Weighted Graph:** A graph having a weight associated with each edge
- ✓**Undirected Graph:** It is a graph that is a set of vertices connected by edges, where the edges don't have a direction associated with them.
- ✓**Directed Graph:** It is a graph that is a set of vertices connected by edges, where the edges have a direction associated with them.
- ✓**Cyclic Graph:** A cyclic graph is a graph having atleast one loop.
- ✓**Acyclic Graph:** An Acyclic graph is a graph having no loop.
- ✓**Tree:** Tree is a Special case of Directed Acyclic Graph (DAG).





Types of Graph (Pictorial Representation):

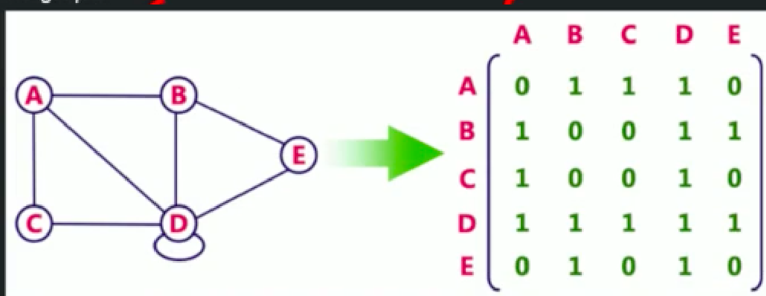


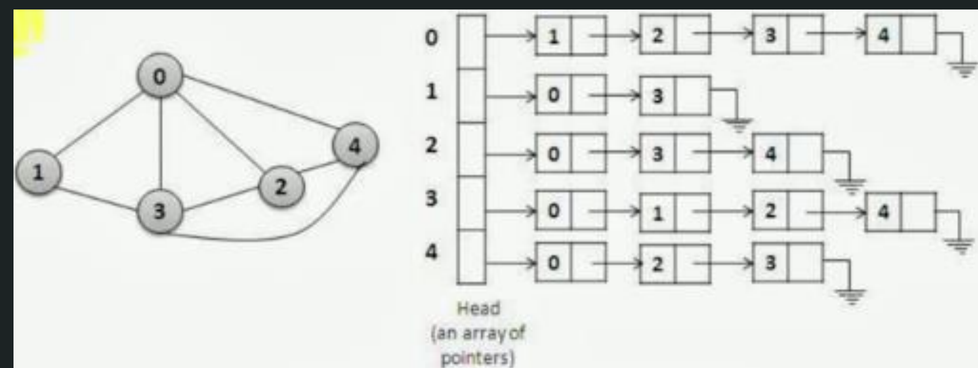


How is Graph represented:

✓ **Adjacency Matrix:** *In graph theory, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.*

✓ **Adjacency List:** *In graph theory, an adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in the graph.*





What is 'Graph Traversal':

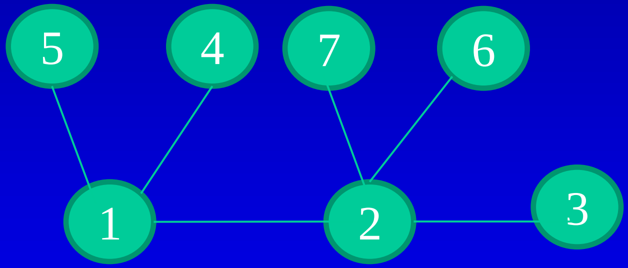
- ✓ Graph traversal refers to the process of visiting each vertex in a graph.

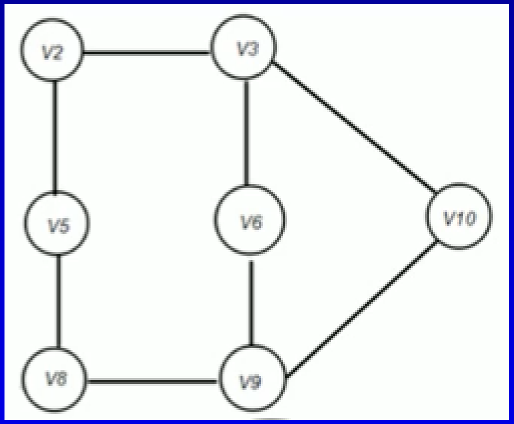
Types of 'Graph Traversal' :

- ✓ Breadth First Search (BFS)
- ✓ Depth First Search (DFS)

What is 'Breadth First Search' (BFS) ?

✓BFS is an algorithm for traversing Graph data structures. It starts at some arbitrary node of a graph and explores the neighbor nodes (which are at current level) first, before moving to the next level neighbors.





1. Visiting a vertex
2. Exploring a vertex

Algorithm: Breadth First Search (BFS):

BFS(G):

while all the vertices are not explored, do:

enqueue (any vertex)

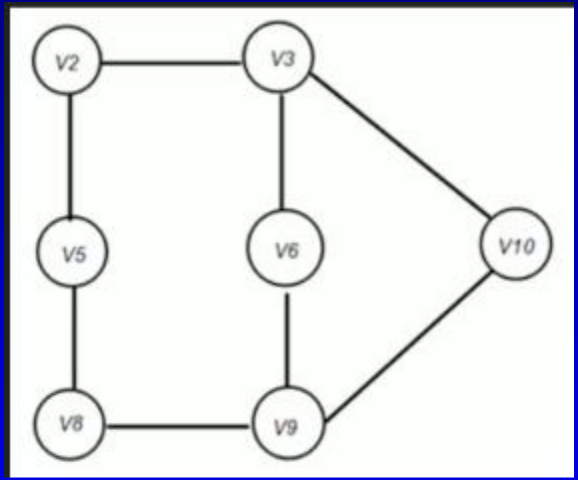
while Q is not empty

p = Dequeue()

if p is unvisited

print ‘p’ and mark ‘p’ as visited

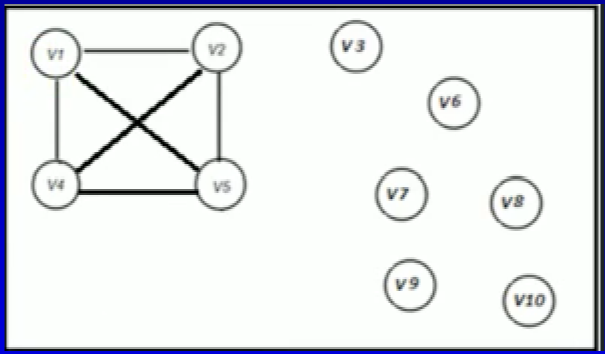
enqueue(all adjacent unvisited vertices of ‘p’)



V2,v3,v5,v6,v10,v8,v9

Handling one Special Scenario of BFS:

✓ **Disconnected Graph:**



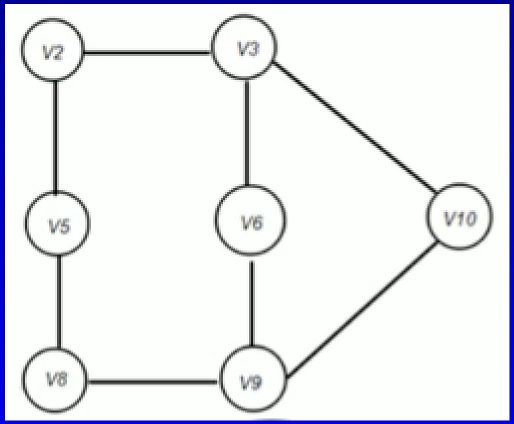
Time Complexity - Breadth First Search (BFS):

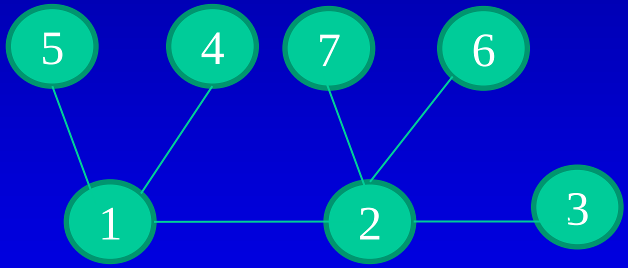
```
BFS(G):
  while all the vertices are not explored, do: ---------------------------------------- O(V)
    enqueue (any vertex) ---------------------------------------------------------- O(1)
    while Q is not empty  ------------------------------------ O(V)                  -------- O(E)
      p = Dequeue() ----------------------------- O(1)
      if  p is unvisited ------------------------ O(1)    ----- O(Adj Vertex)   ------- O(E)
        print 'p' and mark 'p' as visited ------- O(1)
        enqueue(all adjacent unvisited vertices of 'p') ------- O(Adj Vertex)
```

Time Complexity – $O(V + E)$

What is Depth First Search (DFS):

✓Depth-first search (DFS) is an algorithm for traversing Graph data structures. It starts selecting some arbitrary node and explores as far as possible along each edge before backtracking.

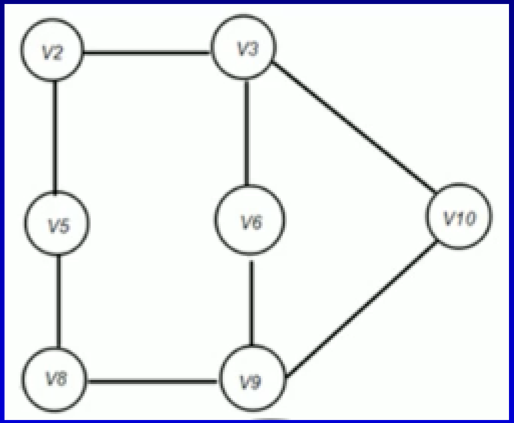




1. Visiting a vertex
2. Exploring a vertex

Algorithm – Depth First Search (DFS):

```
DFS()
 while all the vertices are not explored, do:
       push (any vertex)
      while Stack is not empty
            p = pop()
            if  p is unvisited
                 print 'p' and mark 'p' as visited
                 push(all unvisited adjacent vertices of 'p' )
```



V2,v5,v8,v9,v10,v3,v6

Time Complexity – Depth First Search (DFS):

DFS(G)

while all the vertices are not explored, do: -- O(V)

push (any vertex) -- O(1)

while Stack is not empty -- O(V)

p = pop() -- O(1)

if p is unvisited -- O(1)

print 'p' and mark 'p' as visited -- O(1)

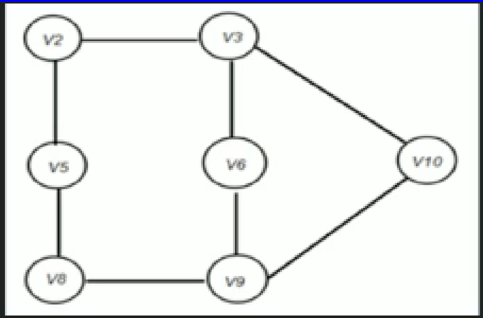
push(all adjacent vertices of 'p' which are not visited) ---------------- O(AdjVertex)

-------- O(AdjVertex)

----- O(E)

Time Complexity – O(E + V)

Space Complexity – O(E + V)

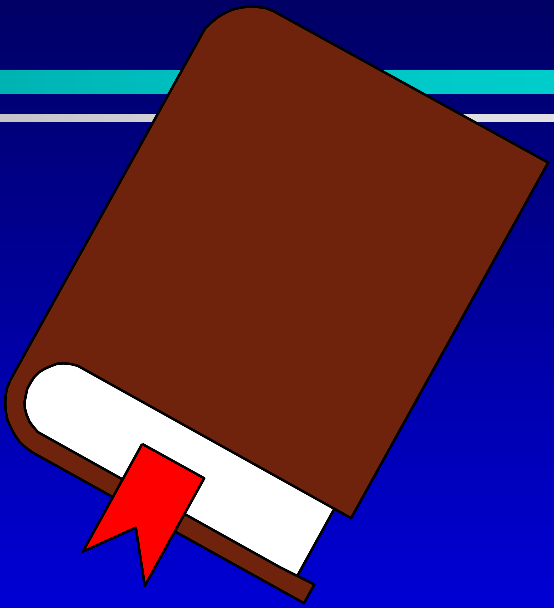


DFS vs BFS:

	DFS	BFS
How it works internally	*It goes in 'depth' first*	*It goes in 'breadth' first*
Internally uses which DS	*Stack*	*Queue*
Time Complexity	$O(E + V)$	$O(E + V)$
Space Complexity	$O(E + V)$	$O(E + V)$
When to use which ?	*If we already know that the target vertex is buried very deep.*	*If we know that target vertex is close to starting point.*

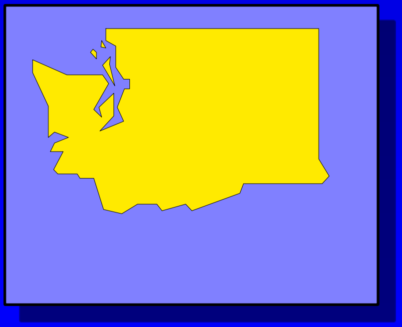
Dictionary

The Dictionary Data Type

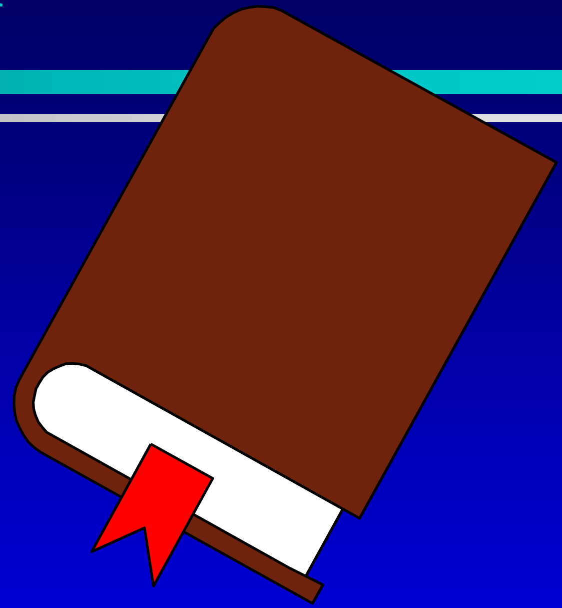


- A dictionary is a collection of **items**, similar to a bag.
- But unlike a bag, each item has a string attached to it, called the item's **key**.

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.

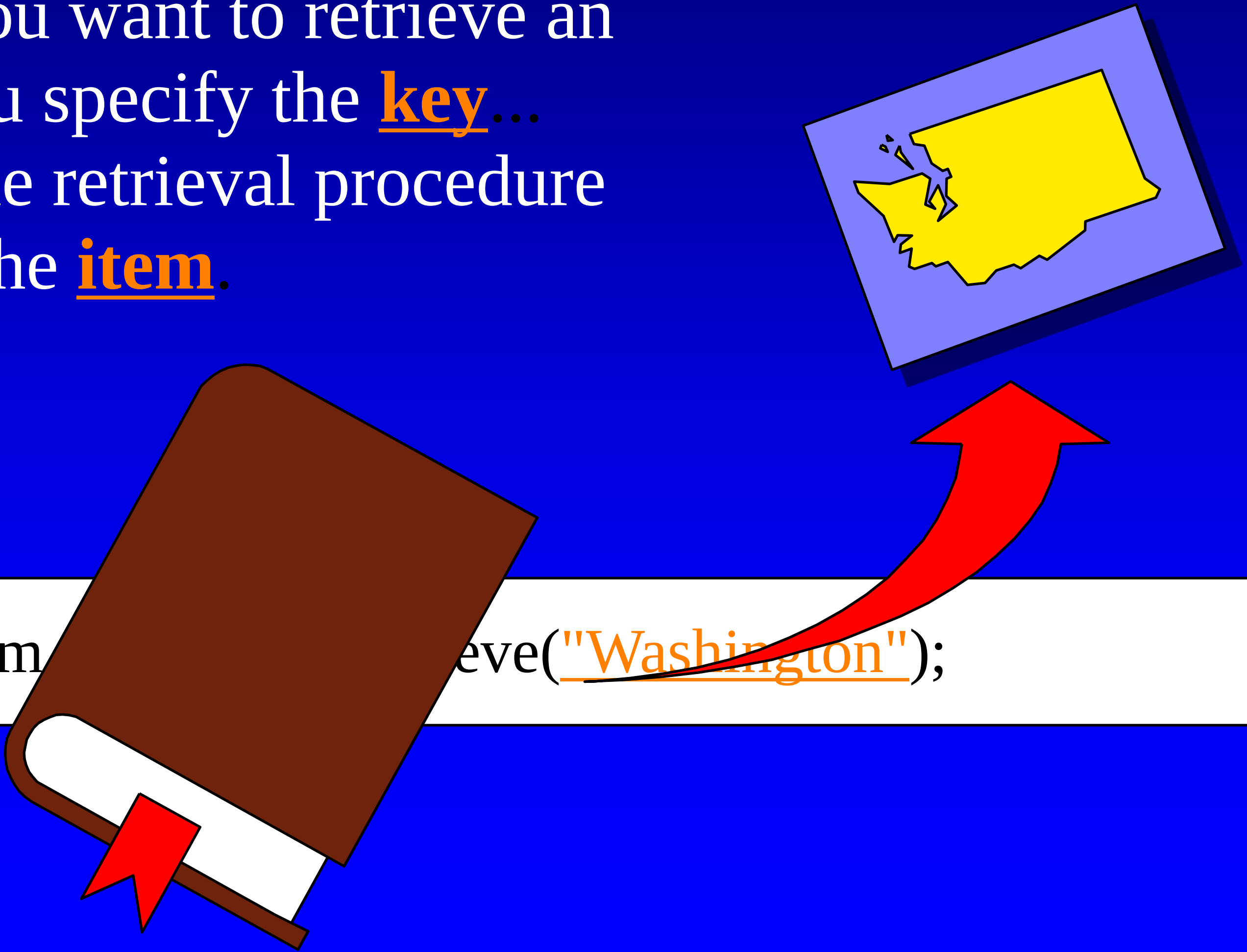
The Dictionary Data Type

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

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

The Dictionary Data Type

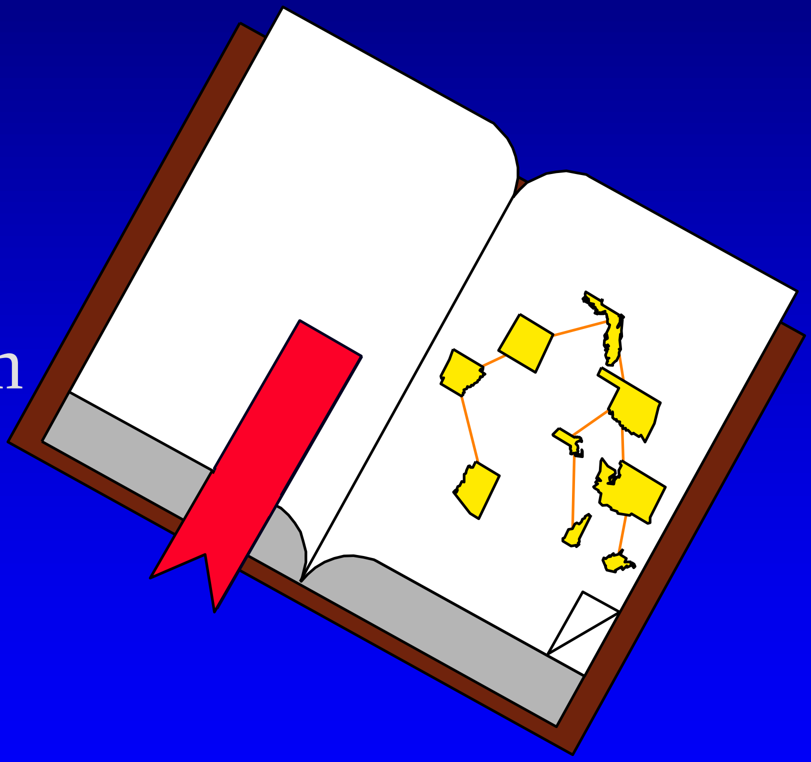
- When you want to retrieve an item, you specify the **key**...
  ... and the retrieval procedure returns the **item**.



Item ... eve("Washington");

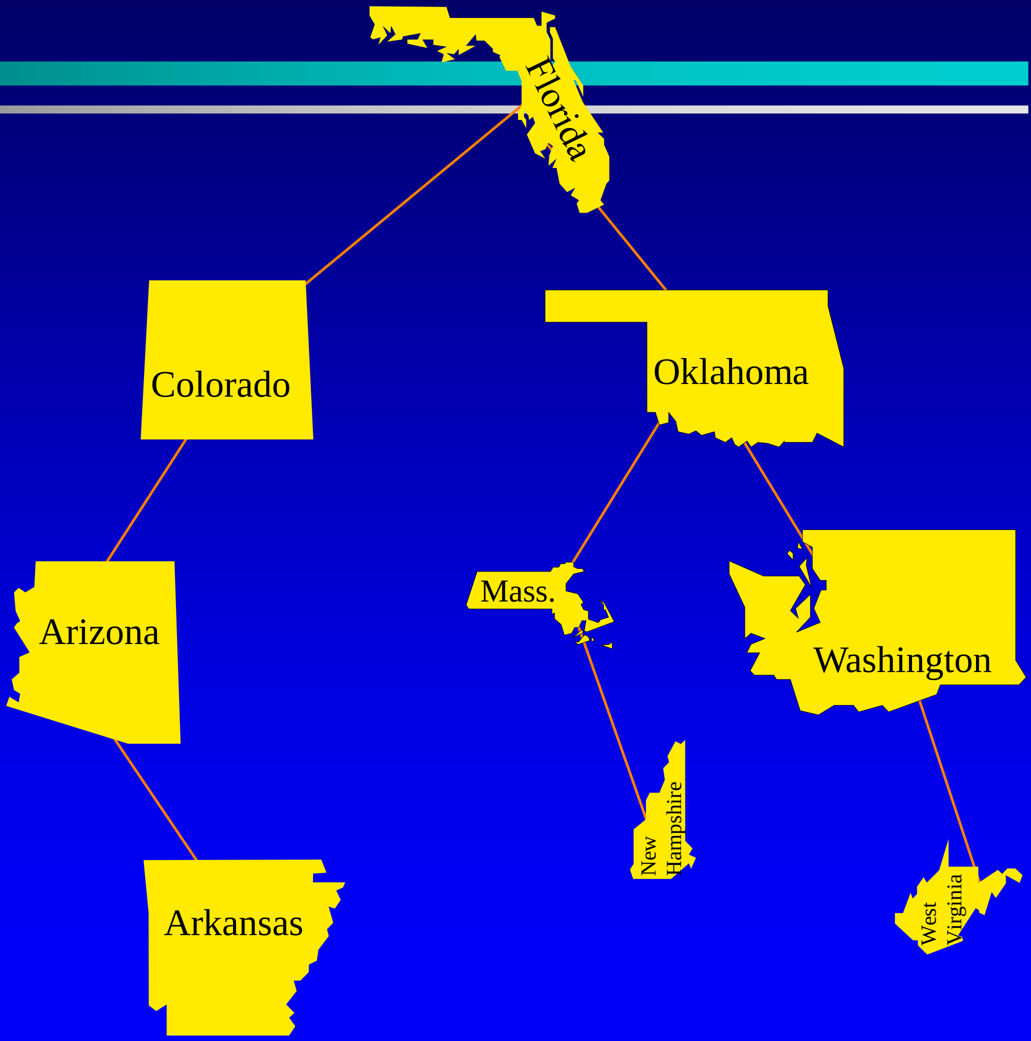
The Dictionary Data Type

- 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**.



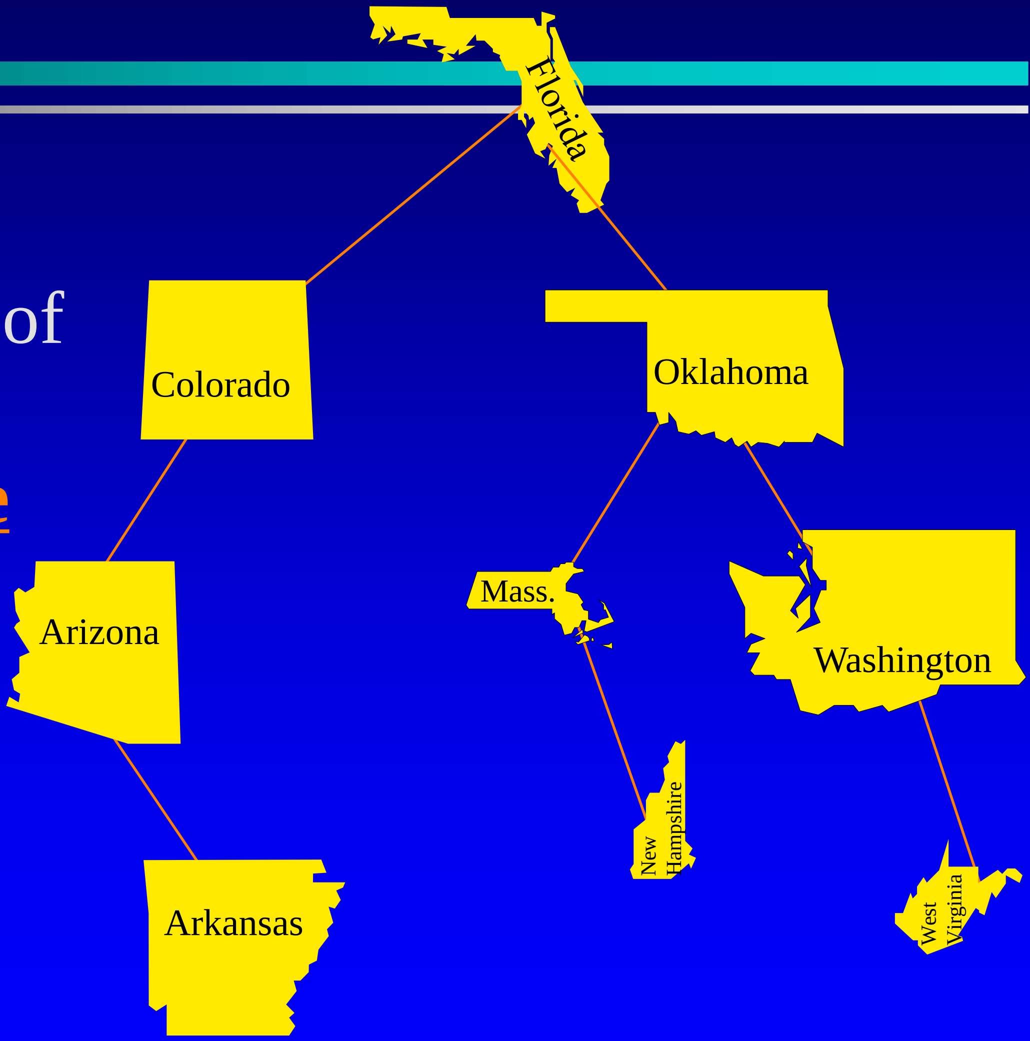
A Binary Search Tree of States

Storage rules:

- Every key to the <u>**left**</u> of a node is alphabetically <u>**before**</u> the key of the node.

Example:

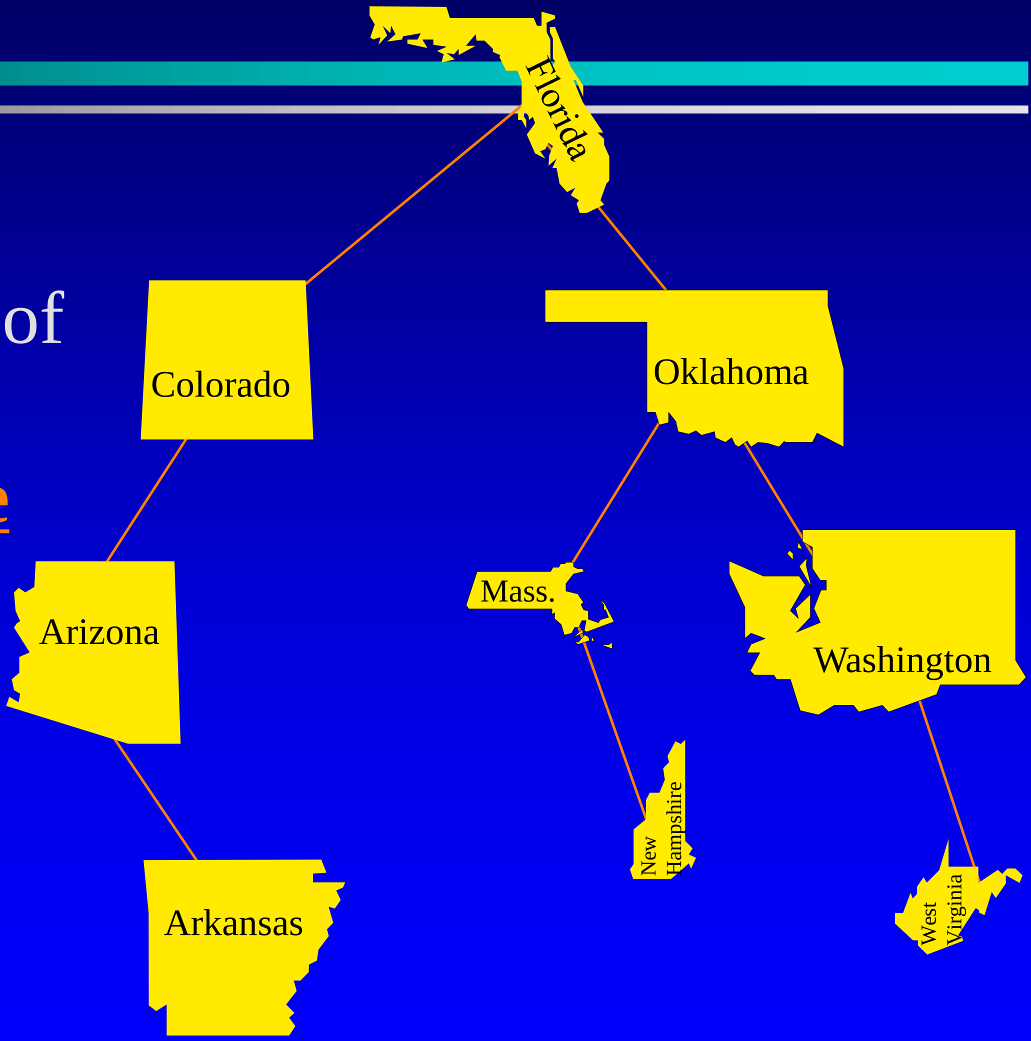
"Massachusetts" and " New Hampshire" are alphabetically before "Oklahoma"



A Binary Search Tree of States

Storage rules:

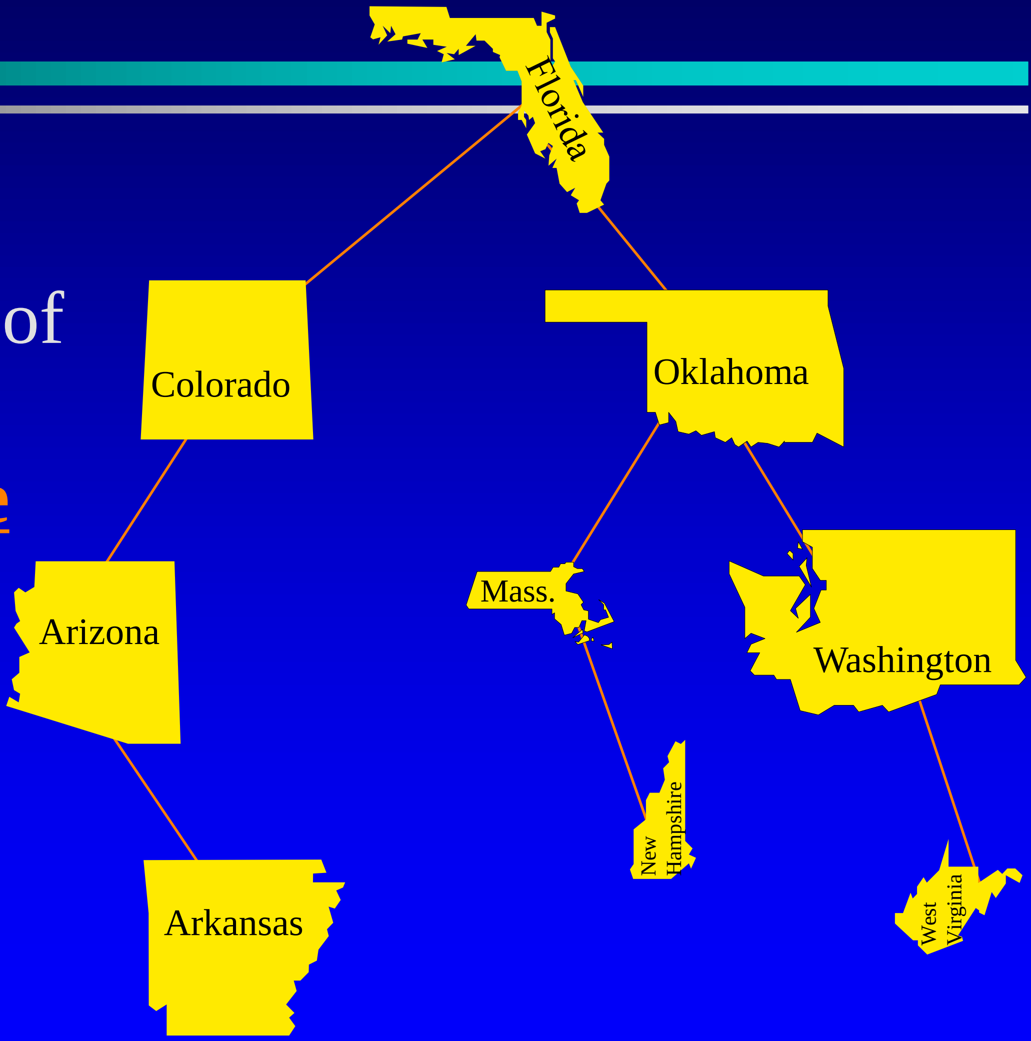
- Every key to the **left** of a node is alphabetically **before** the key of the node.
- Every key to the **right** of a node is alphabetically **after** the key of the node.



A Binary Search Tree of States

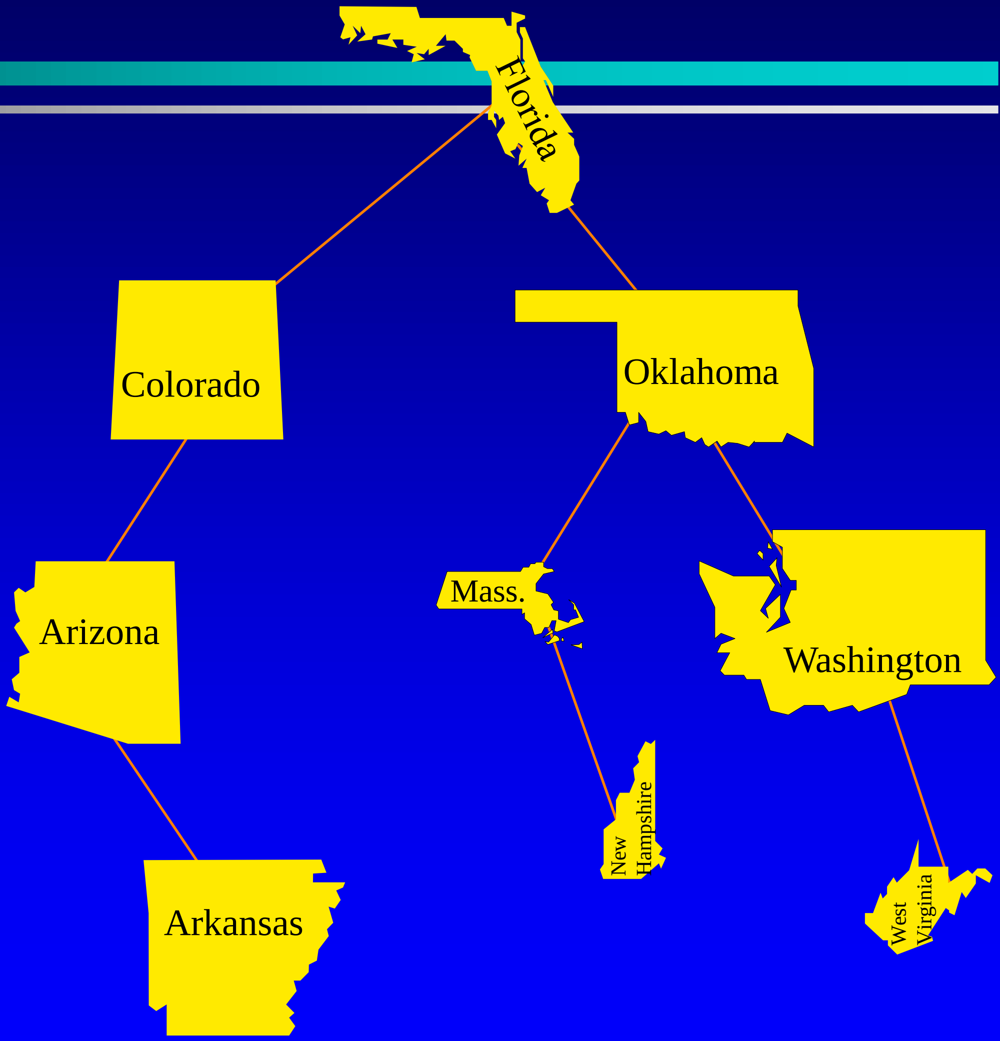
Storage rules:

- 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.



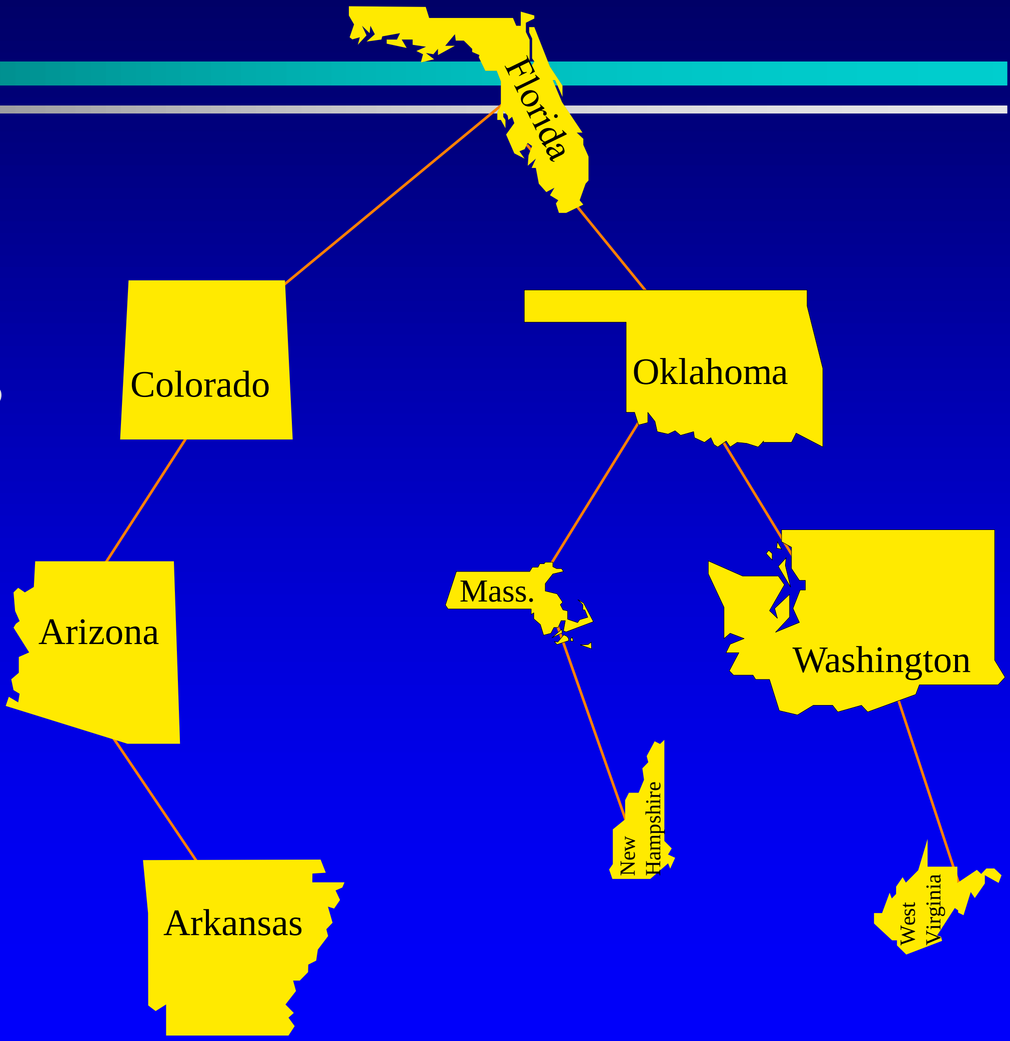
Retrieve " New Hampshire"

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 with a Given Key

- 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.



Adding

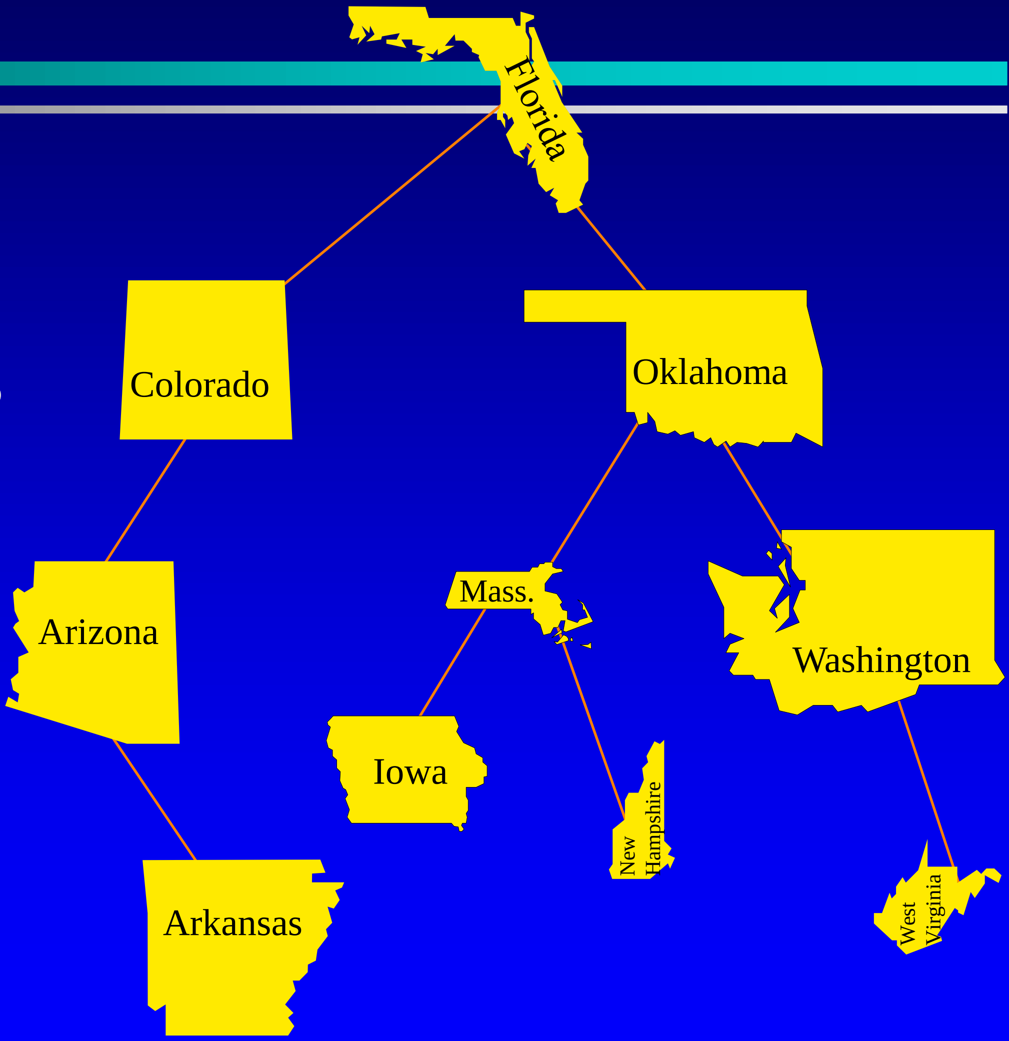
- Pretend that you are trying to find the key, but stop when there is no node to move to.
- Add the new node at the spot where you would have moved to if there had been a node.

Adding

- 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.

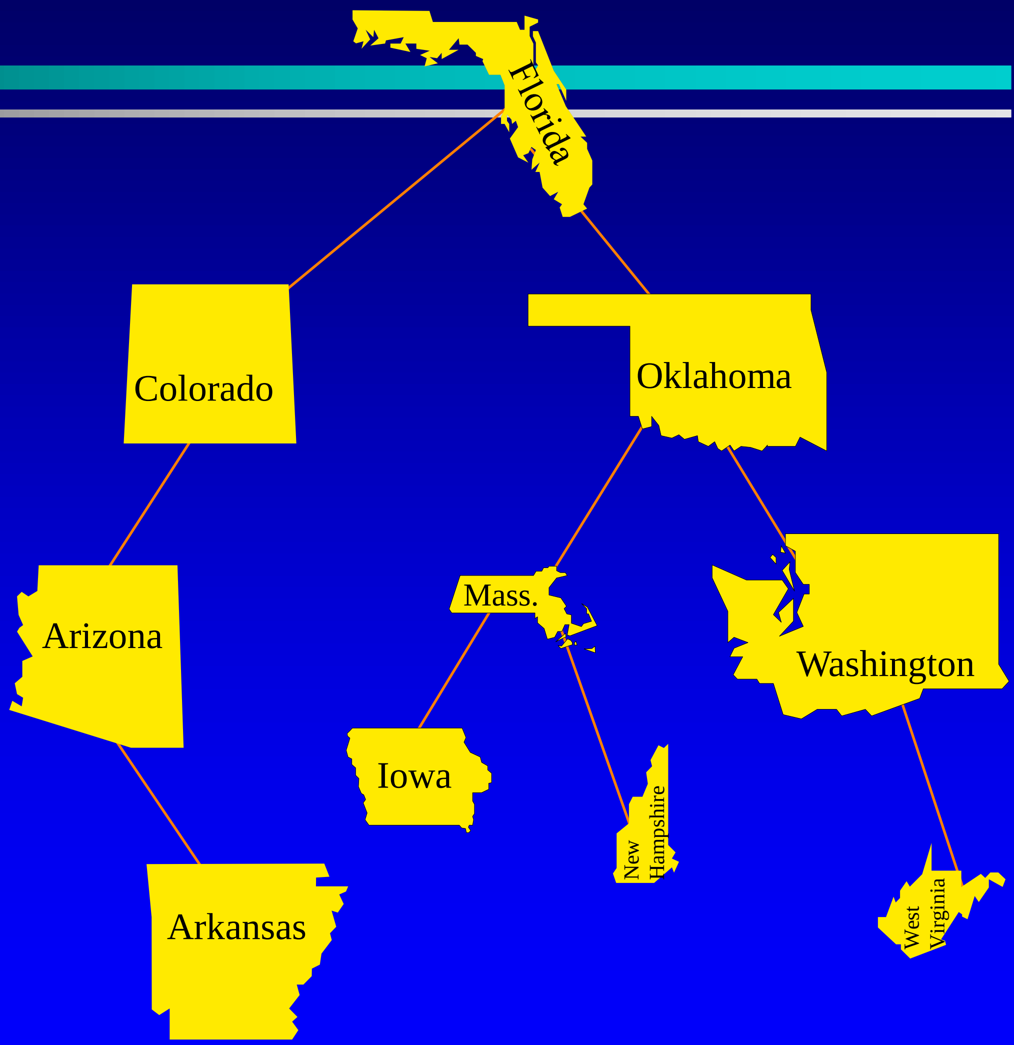
Adding

- 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.



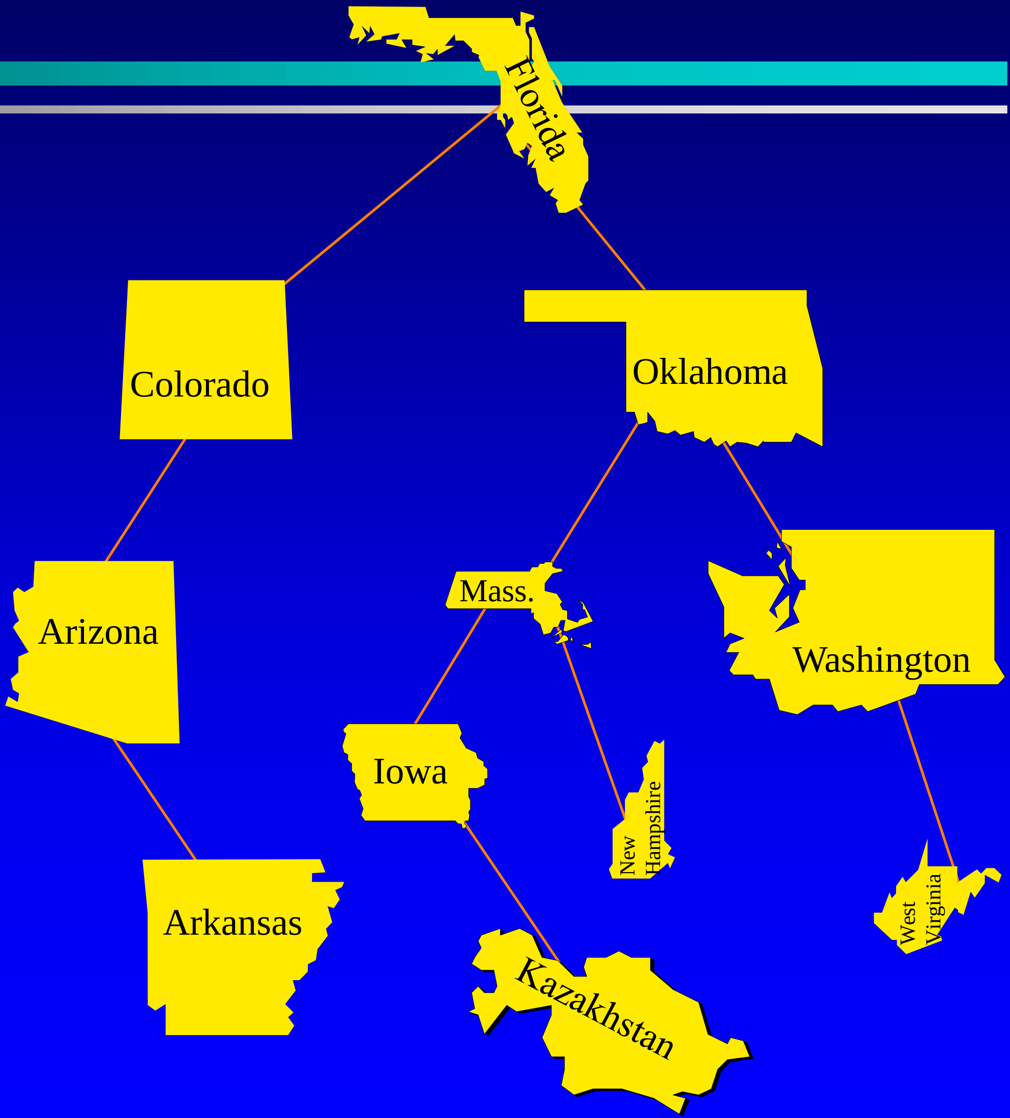
Adding

Where would you add this state?



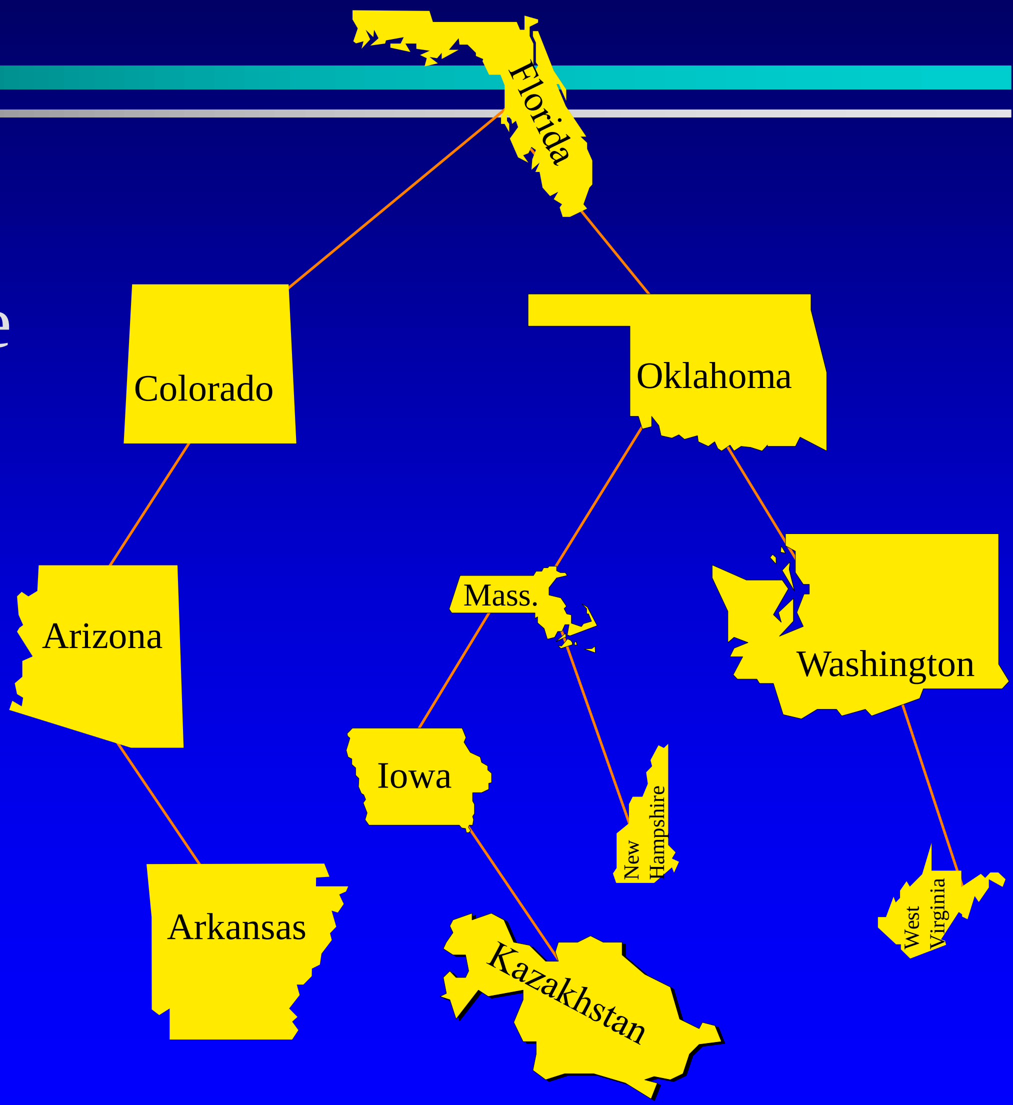
Adding

Kazakhstan is the new right child of Iowa?



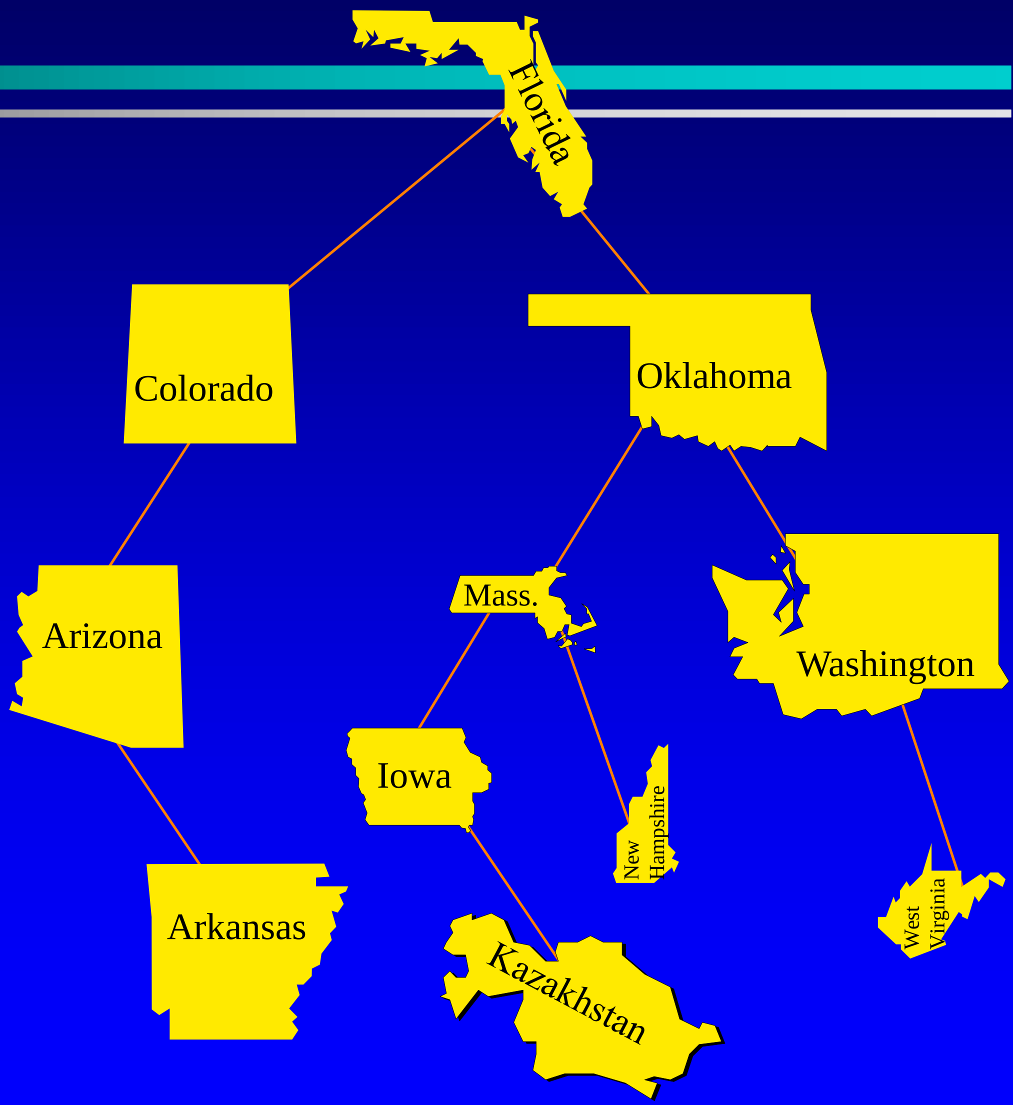
Removing an Item with a Given Key

- 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"

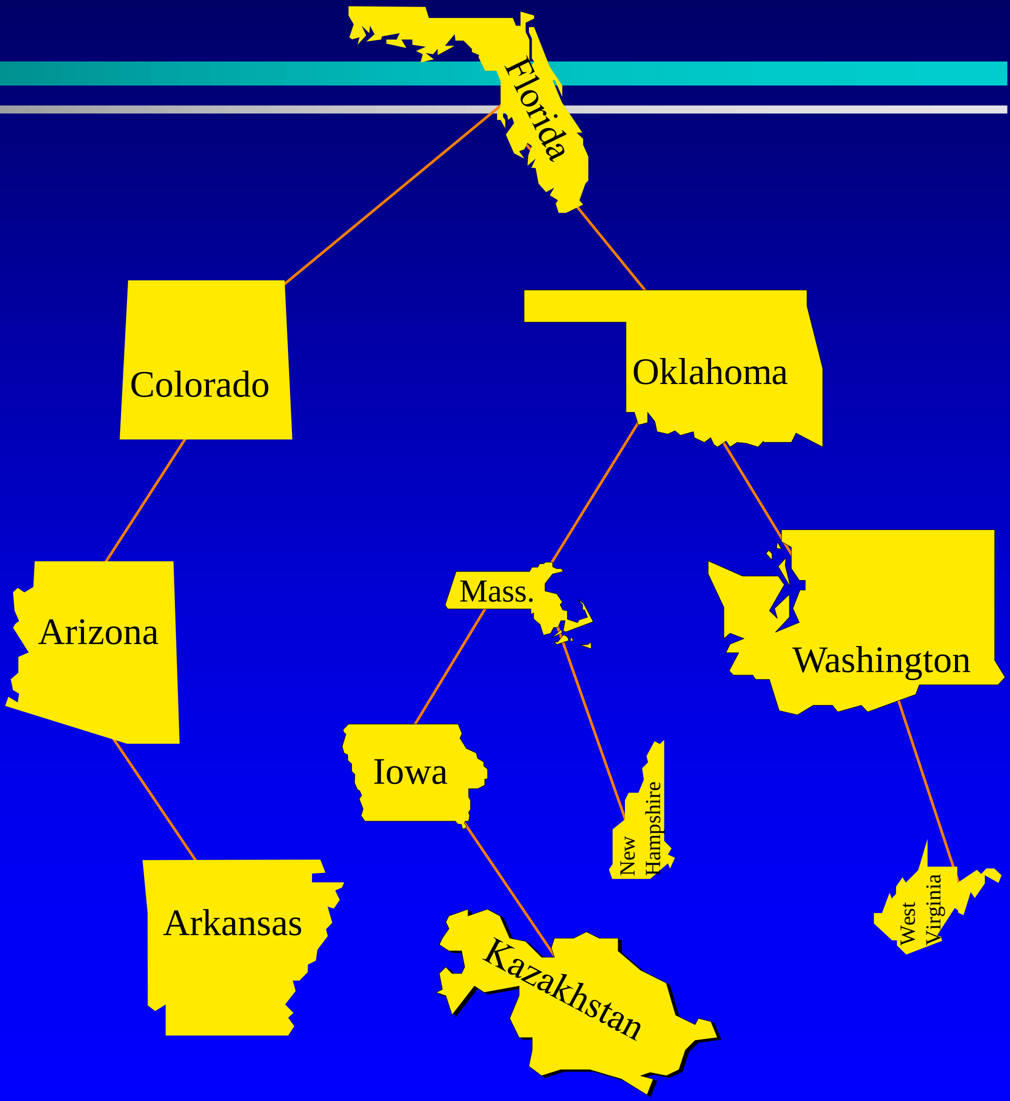
Florida cannot be removed at the moment...

Removing "Florida"

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

Removing "Florida"

- If necessary, do some rearranging.



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

Removing "Florida"

- If necessary, do some rearranging.

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

Removing "Florida"

- If necessary, do some rearranging.

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

Removing "Florida"

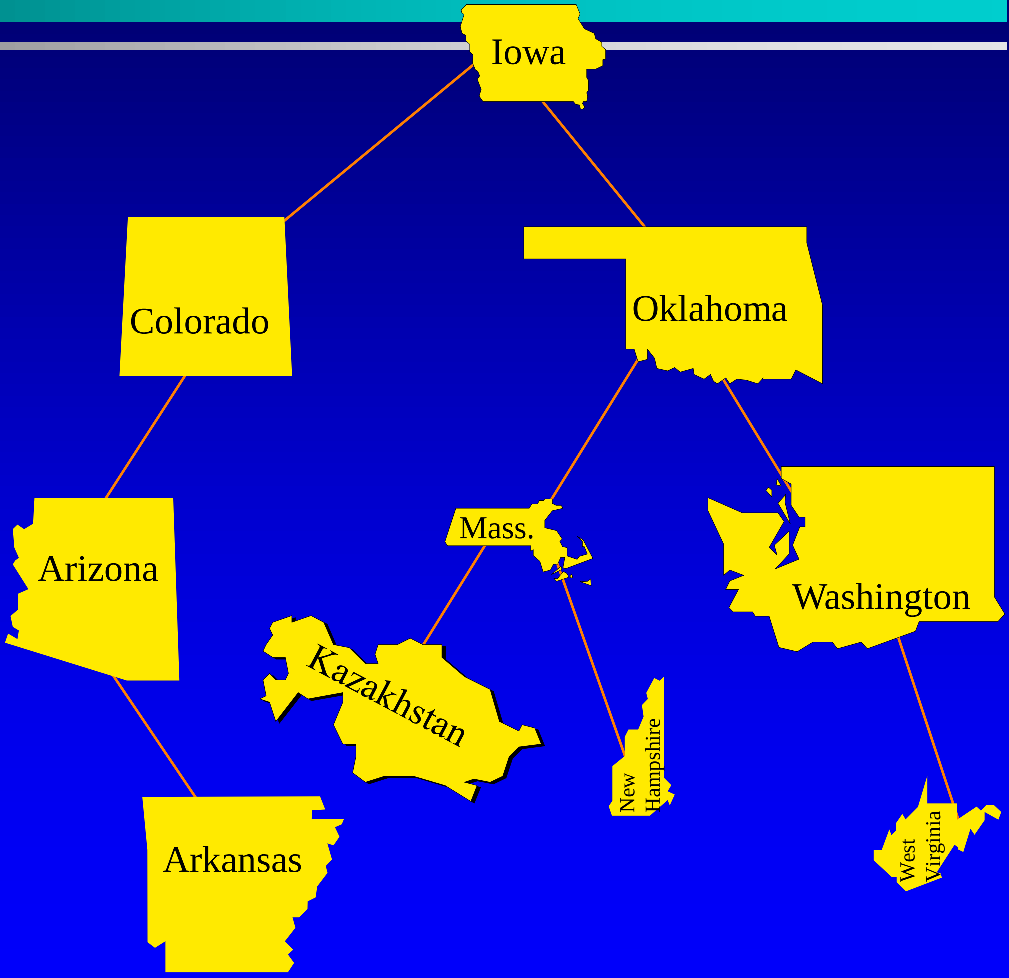
- If necessary, do some rearranging.

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

Removing "Florida"

- If necessary, do some rearranging.

... 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

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.

Thank you