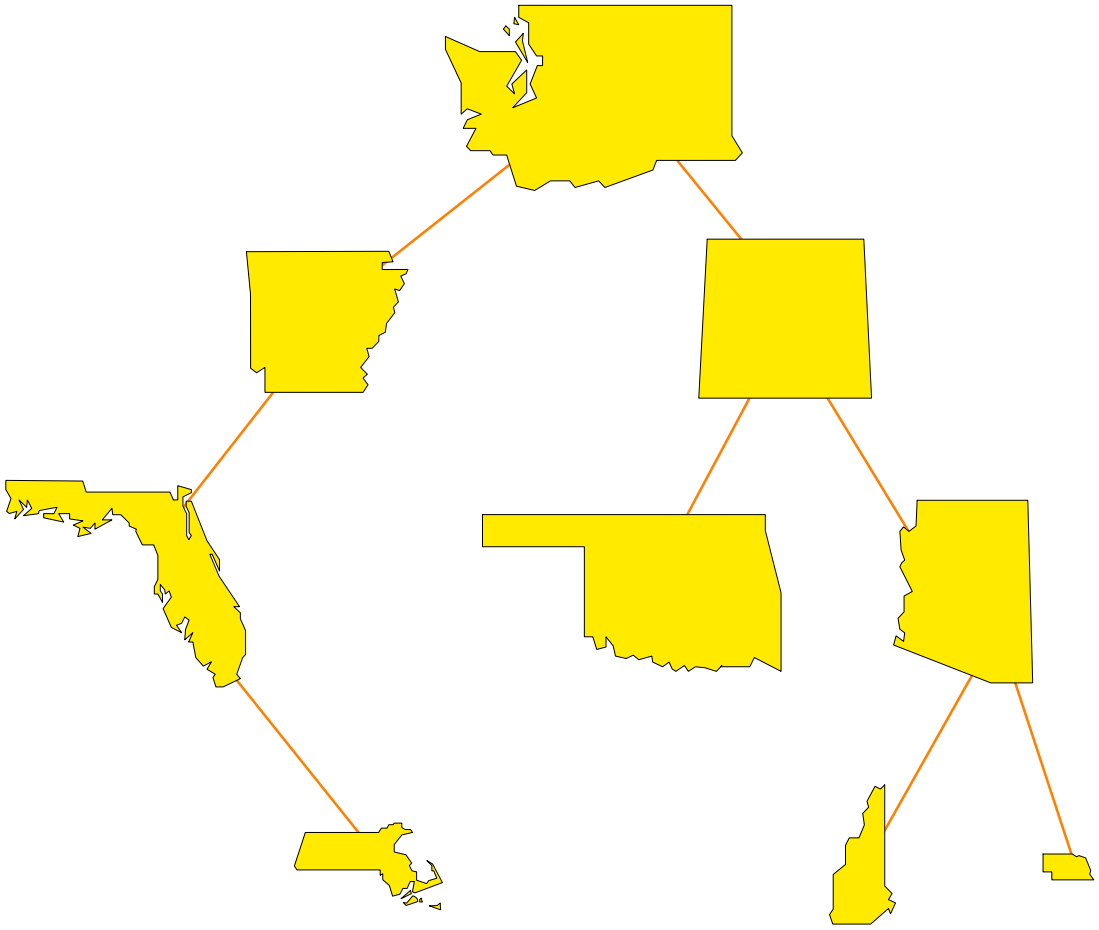
Trees

Binary Trees

- A binary tree has **nodes**, similar to nodes in a linked list structure.
- **Data** of one sort or another may be stored at each node.
- But it is the **connections** between the nodes which characterize a binary tree.

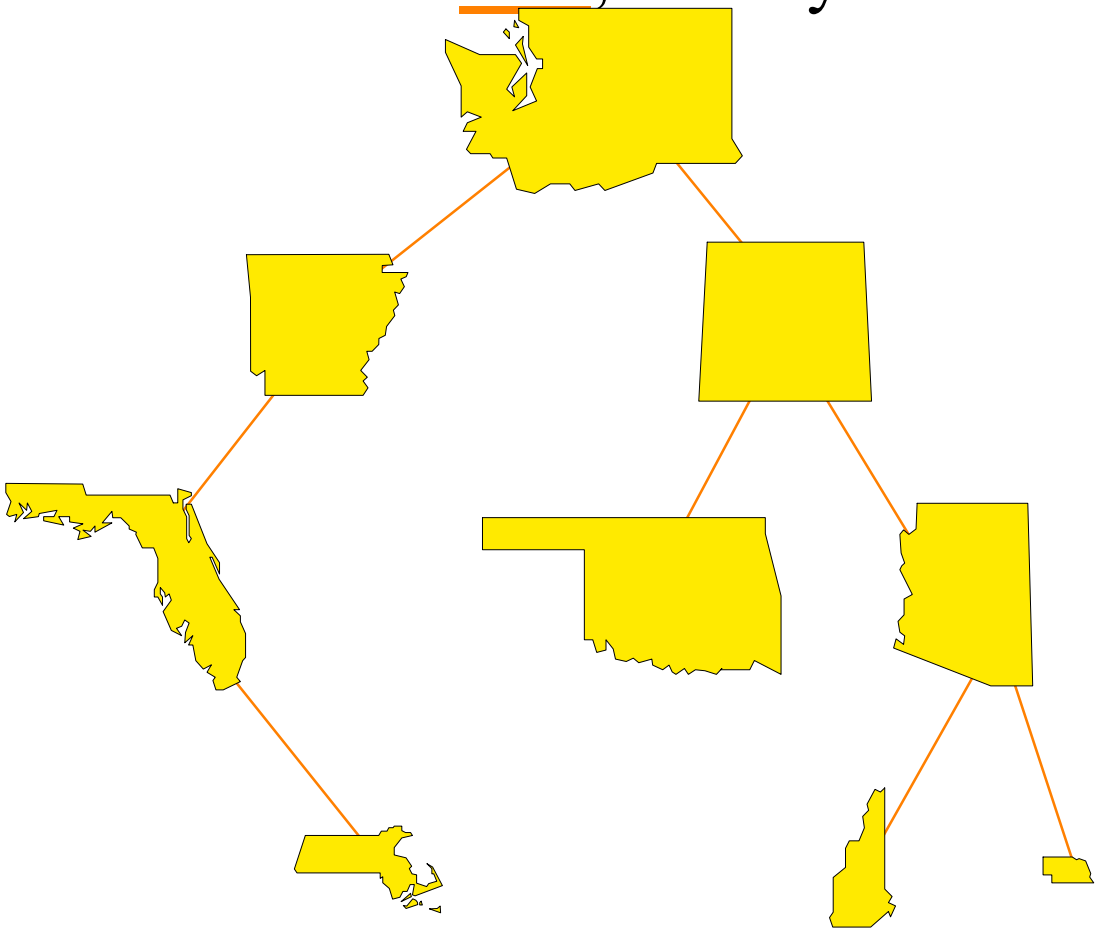
A Binary Tree of States

In this example, the data contained at each node is one of the 50 states.



A Binary Tree of States

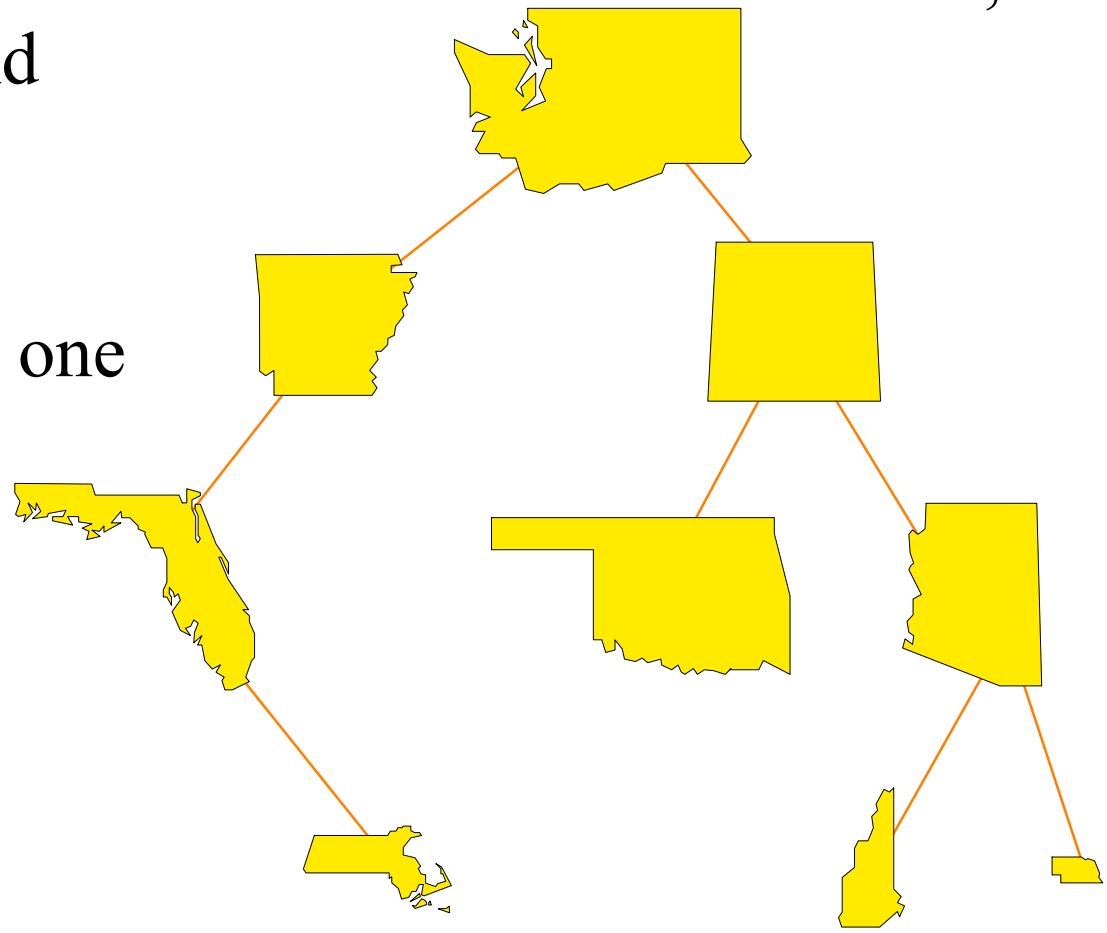
Each tree has a special node called its **root**, usually drawn at the top.



A Binary Tree of States

Each node is permitted to have two links to other nodes, called the **left child** and the **right child**.

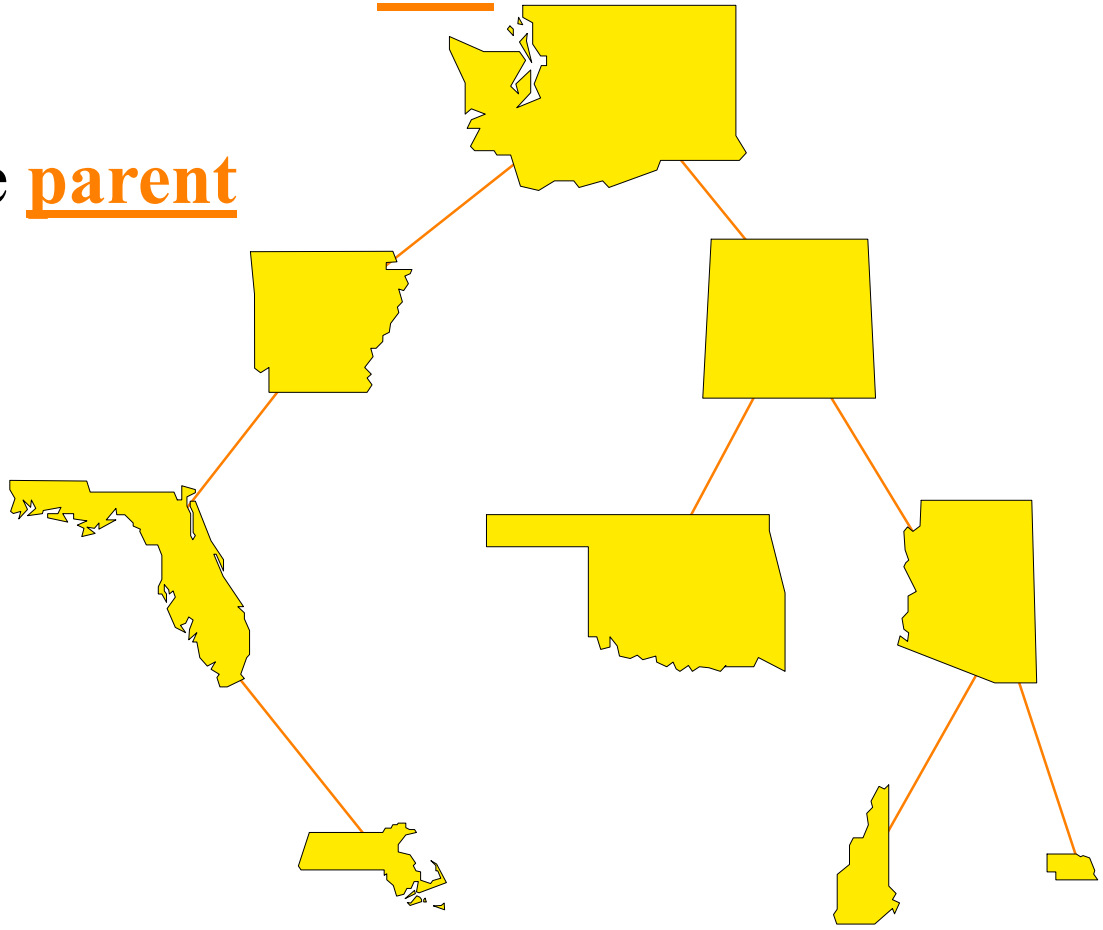
Some nodes have only one child.



A Binary Tree of States

A node with no children is called a **leaf**.

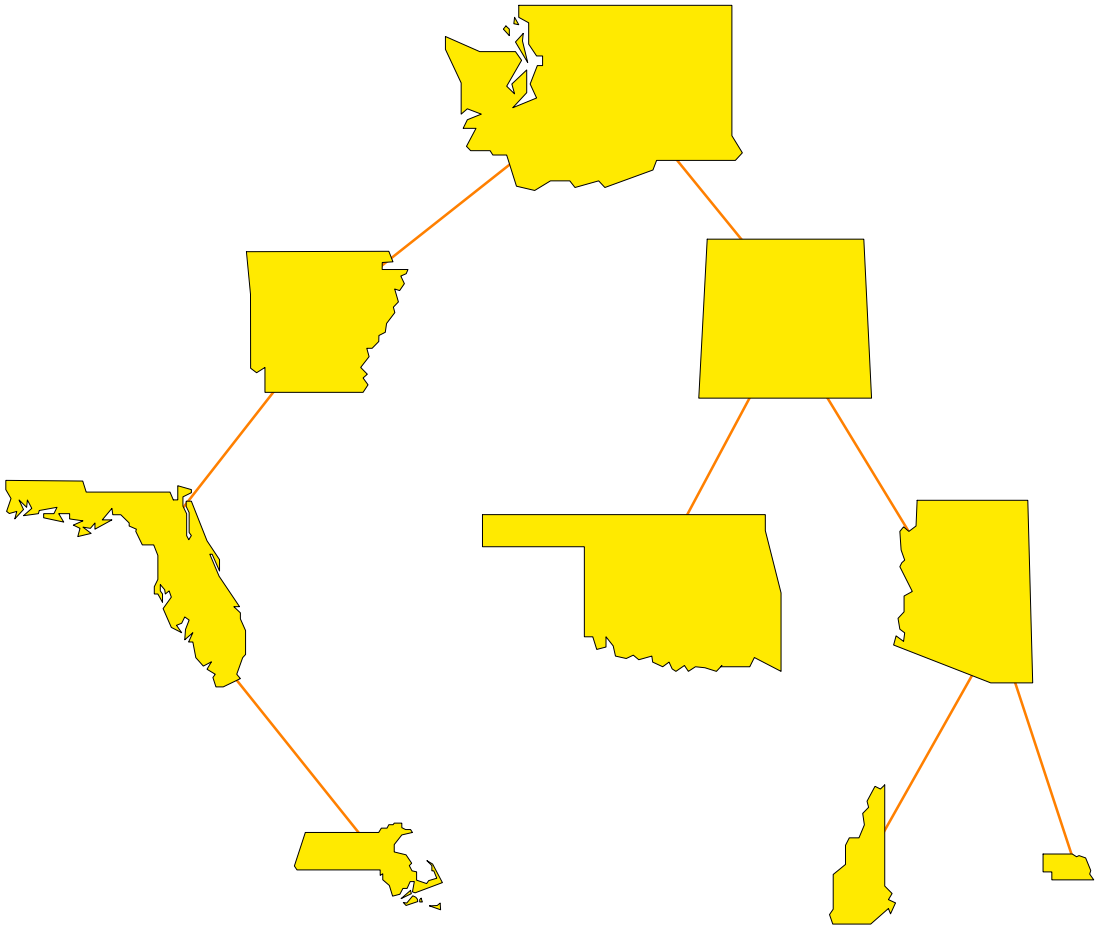
Each node is called the **parent** of its children.



A Binary Tree of States

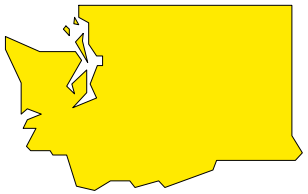
Two rules about parents:

- The root has no parent.
- Every other node has exactly one parent.



Complete Binary Trees

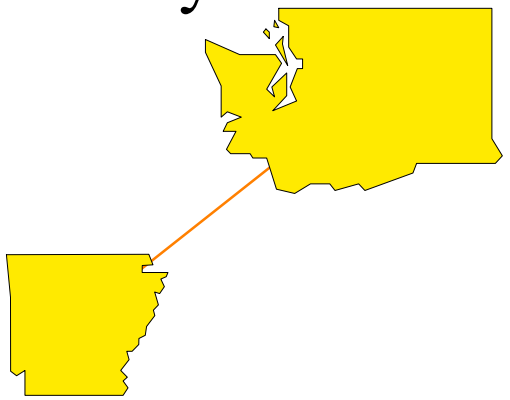
A complete binary tree is a special kind of binary tree which will be useful to us.



When a complete binary tree is built, its first node must be the root.

Complete Binary Trees

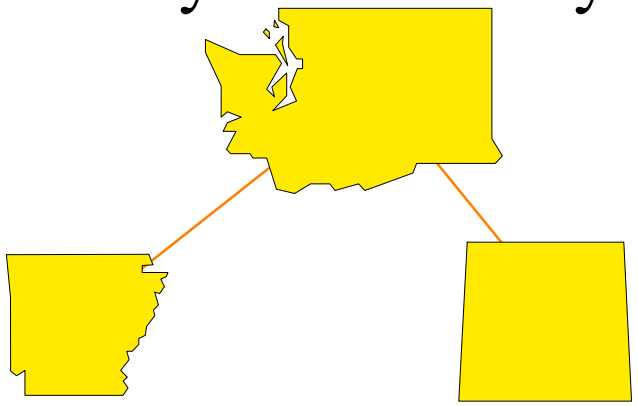
The second node of a complete binary tree is always the left child of the root...



Complete Binary Trees

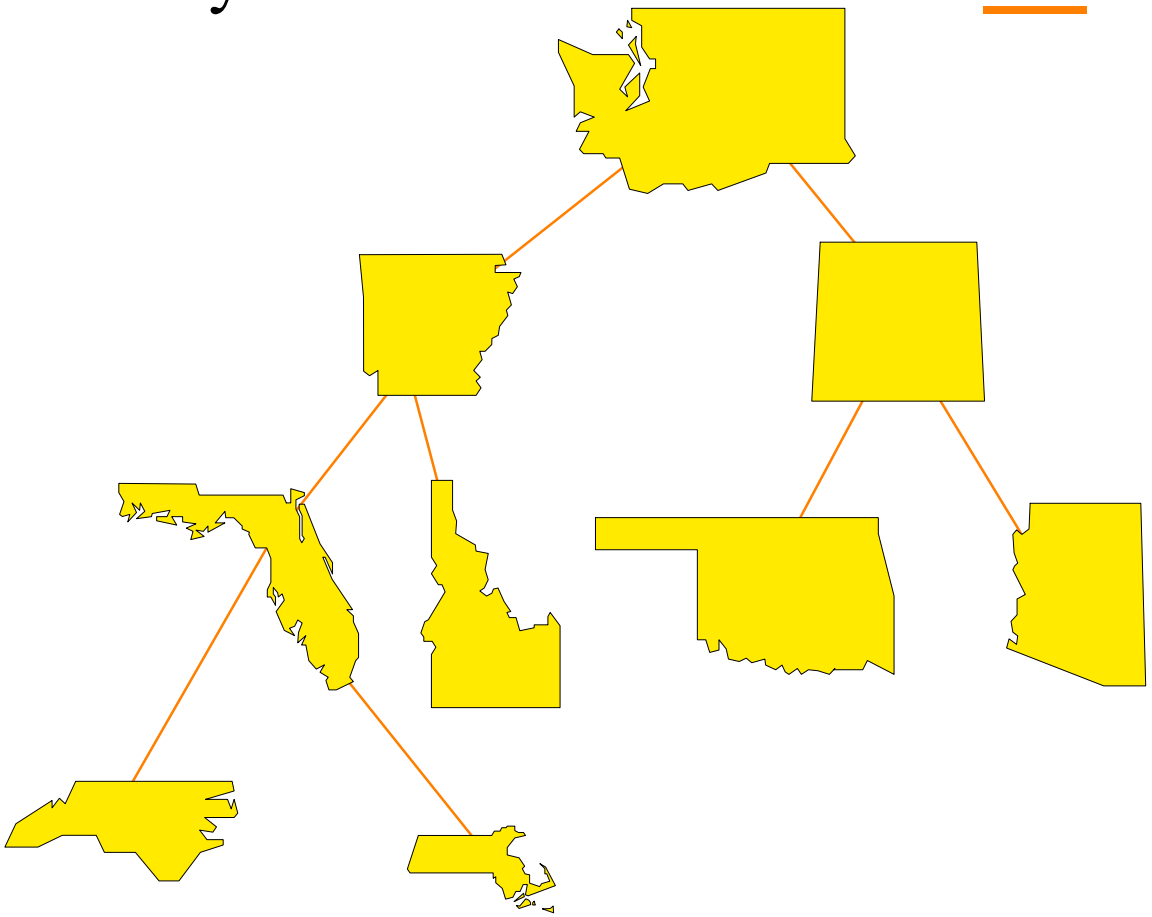
The second node of a complete binary tree is always the left child of the root...

... and the third node is always the right child of the root.

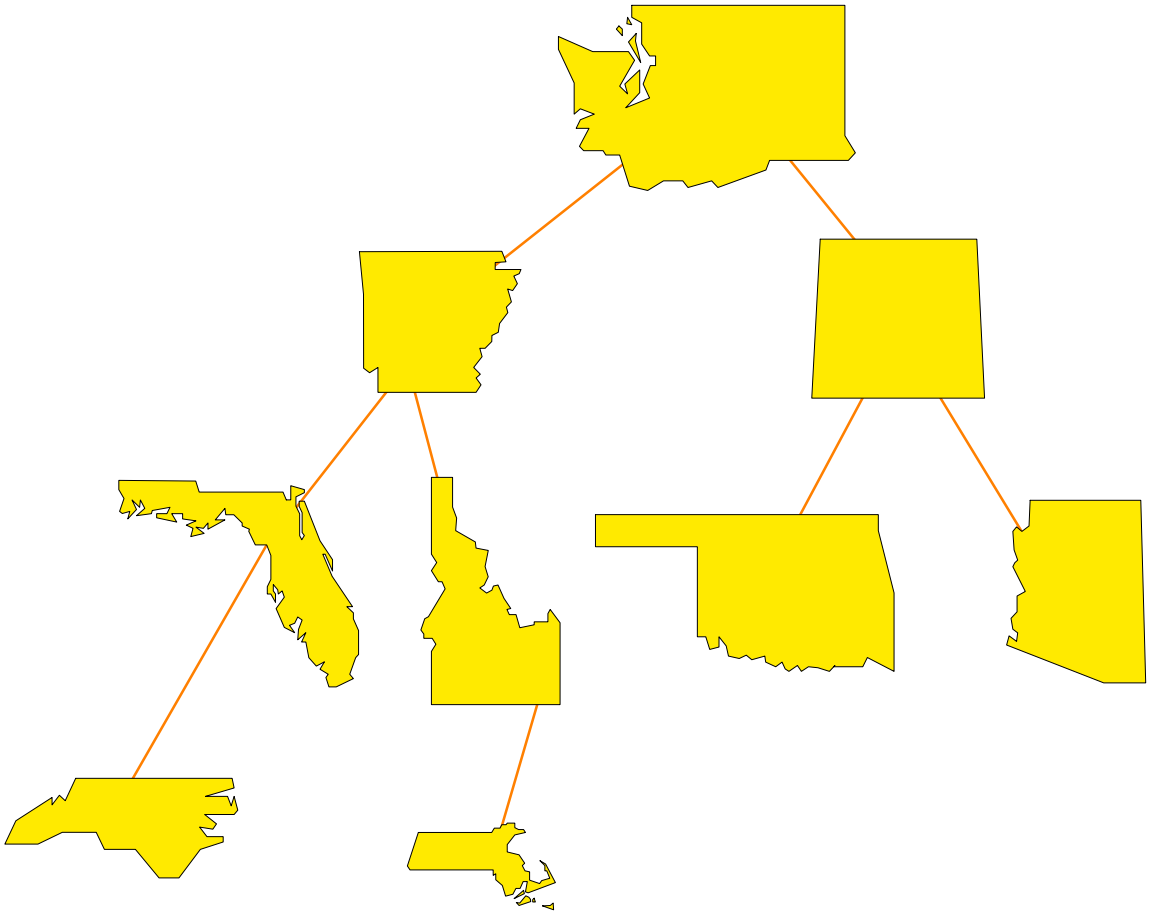


Complete Binary Trees

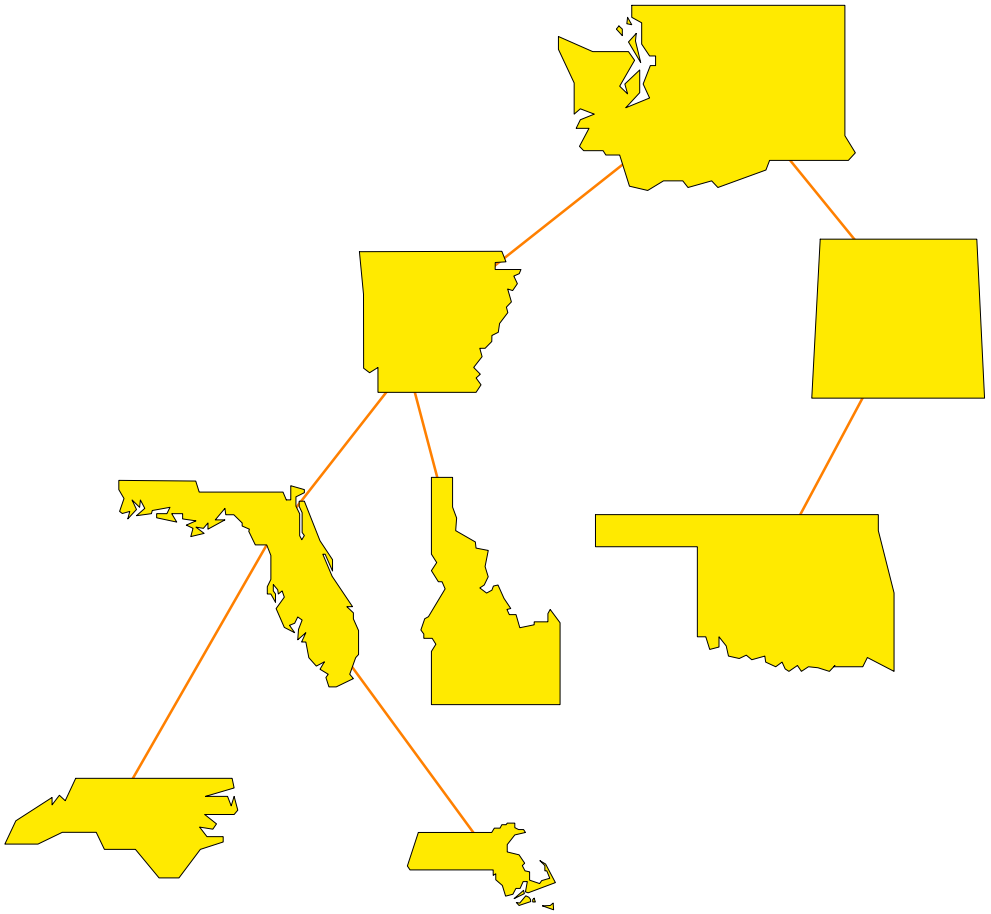
The next nodes must always fill the next level from **left to right**.



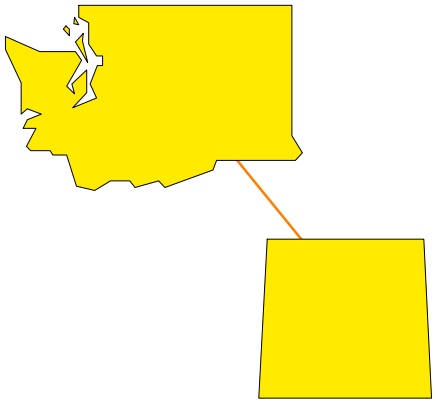
Is This Complete?



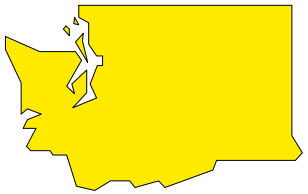
Is This Complete?



Is This Complete?

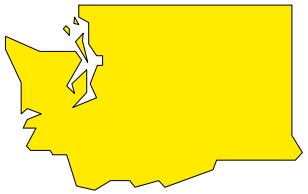


Is This Complete?



Full Binary Trees

A full binary tree is a special kind of complete binary tree

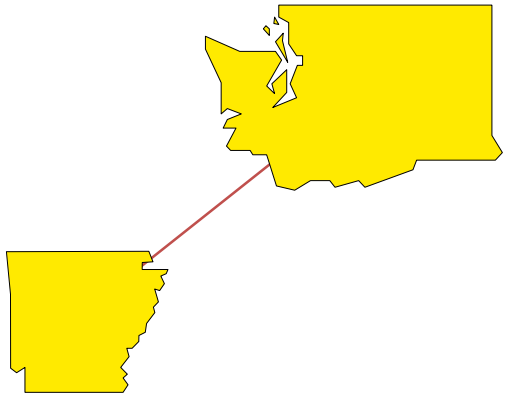


FULL

When a full binary tree is built, its first node must be the root.

Full Binary Trees

The second node of a full binary tree is always the left child of the root...

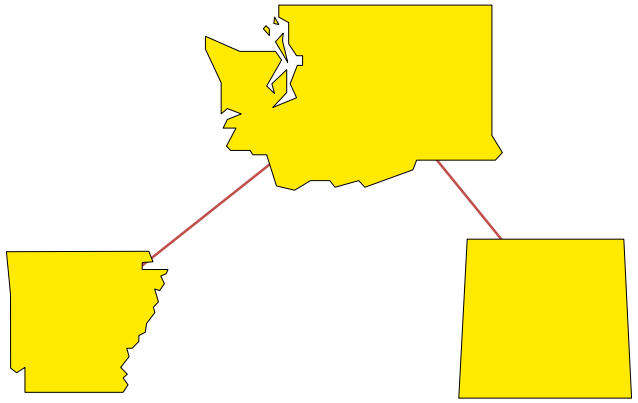


not FULL yet

Full Binary Trees

The second node of a full binary tree is always the left child of the root...

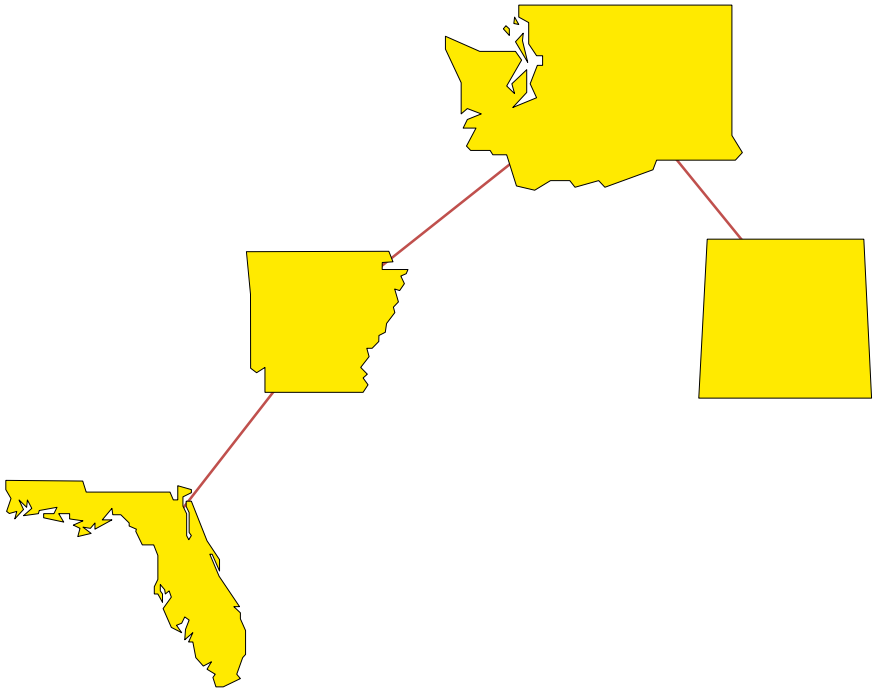
... and you MUST have the third node which always the right child of the root.



FULL

Full Binary Trees

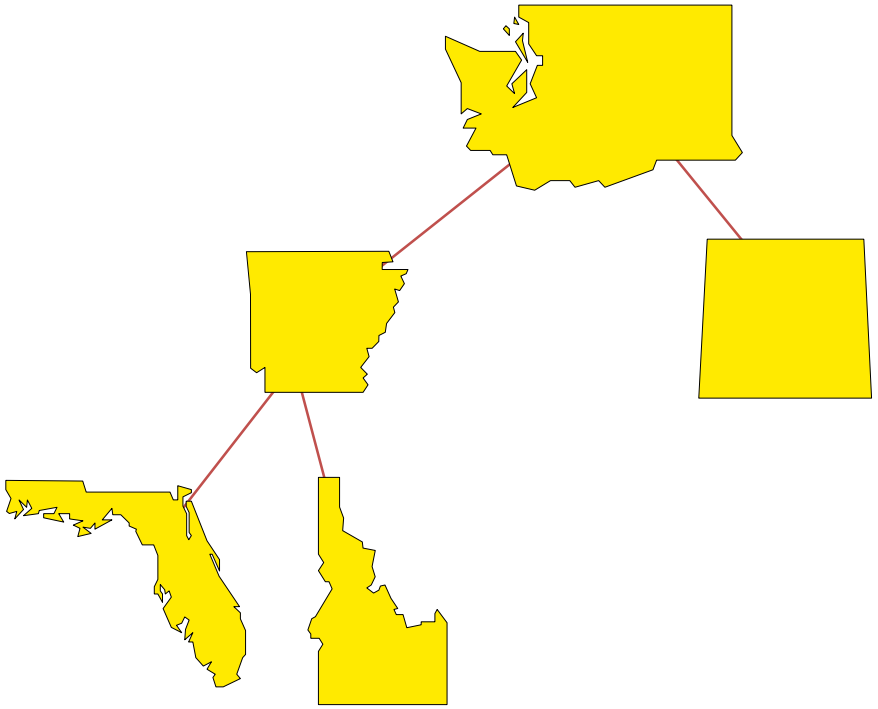
The next nodes must always fill the next level from **left to right**.



not FULL yet

Full Binary Trees

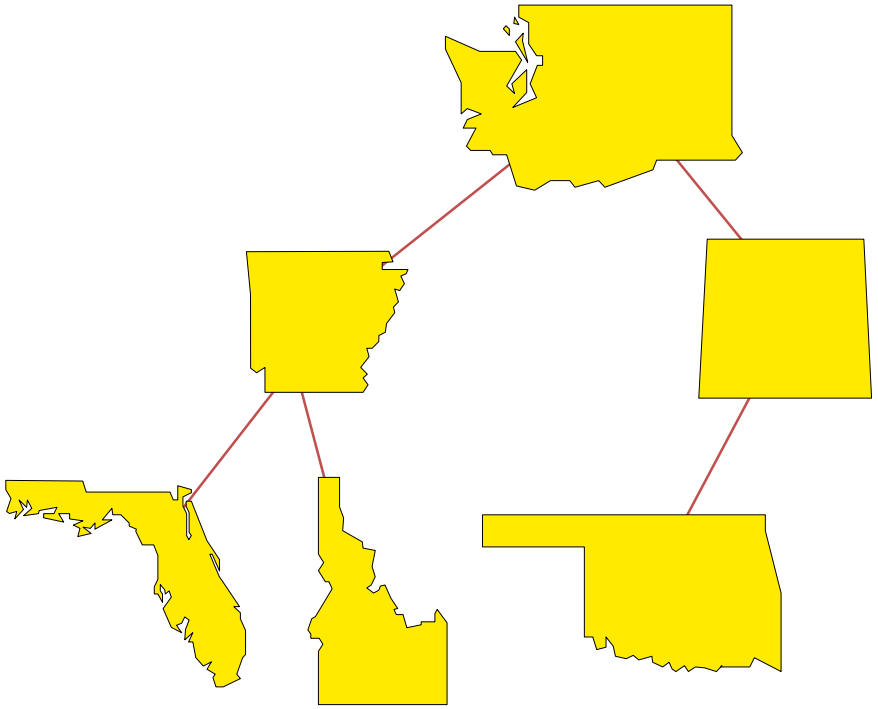
The next nodes must always fill the next level from **left to right**.



not FULL yet

Full Binary Trees

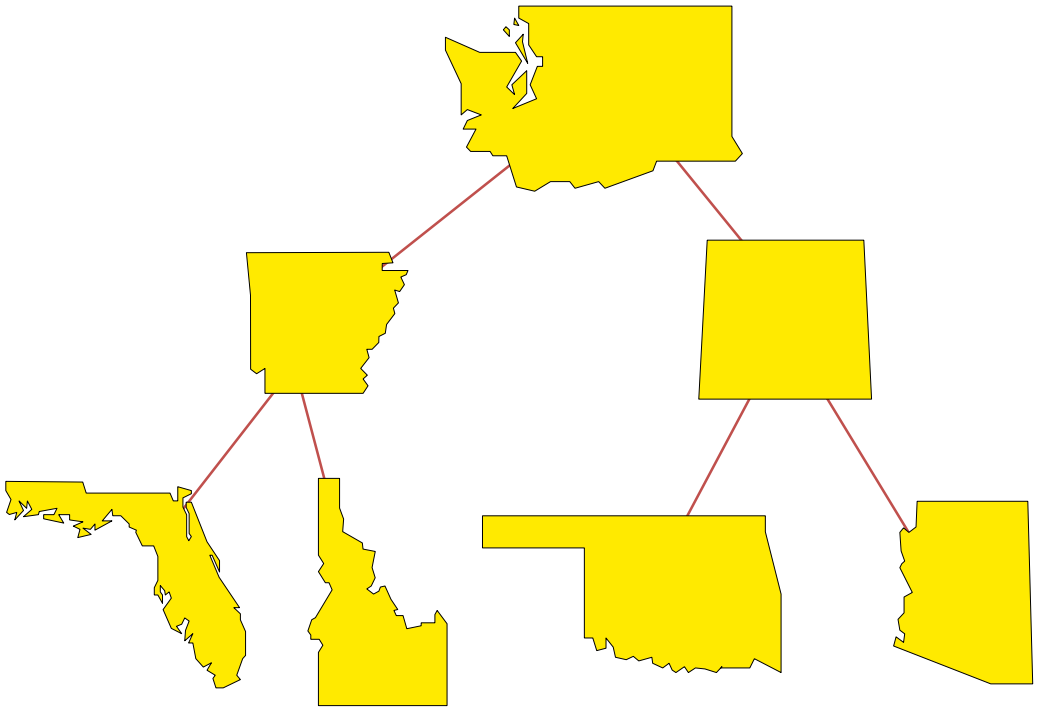
The next nodes must always fill the next level from **left to right**.



not FULL yet

Full Binary Trees

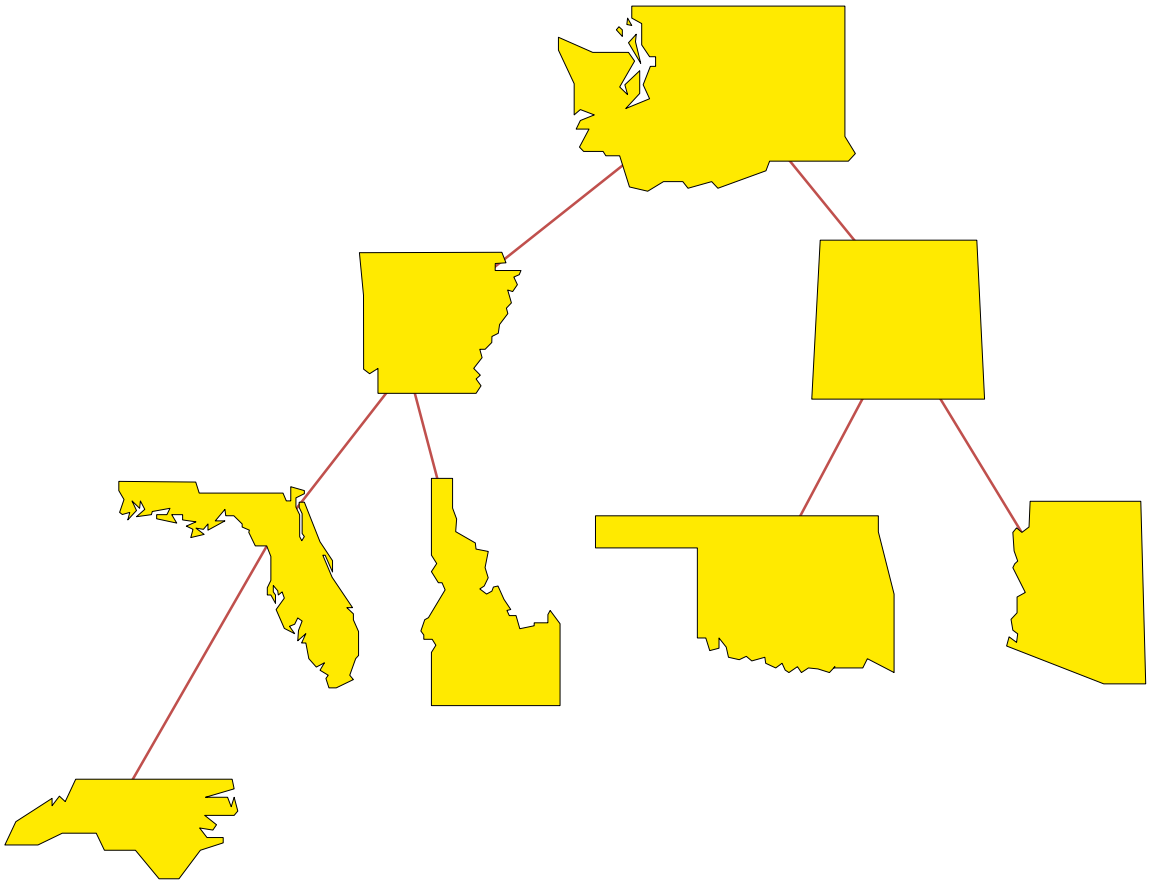
The next nodes must always fill the next level from **left to right**...until every leaf has the same depth (2)



FULL!

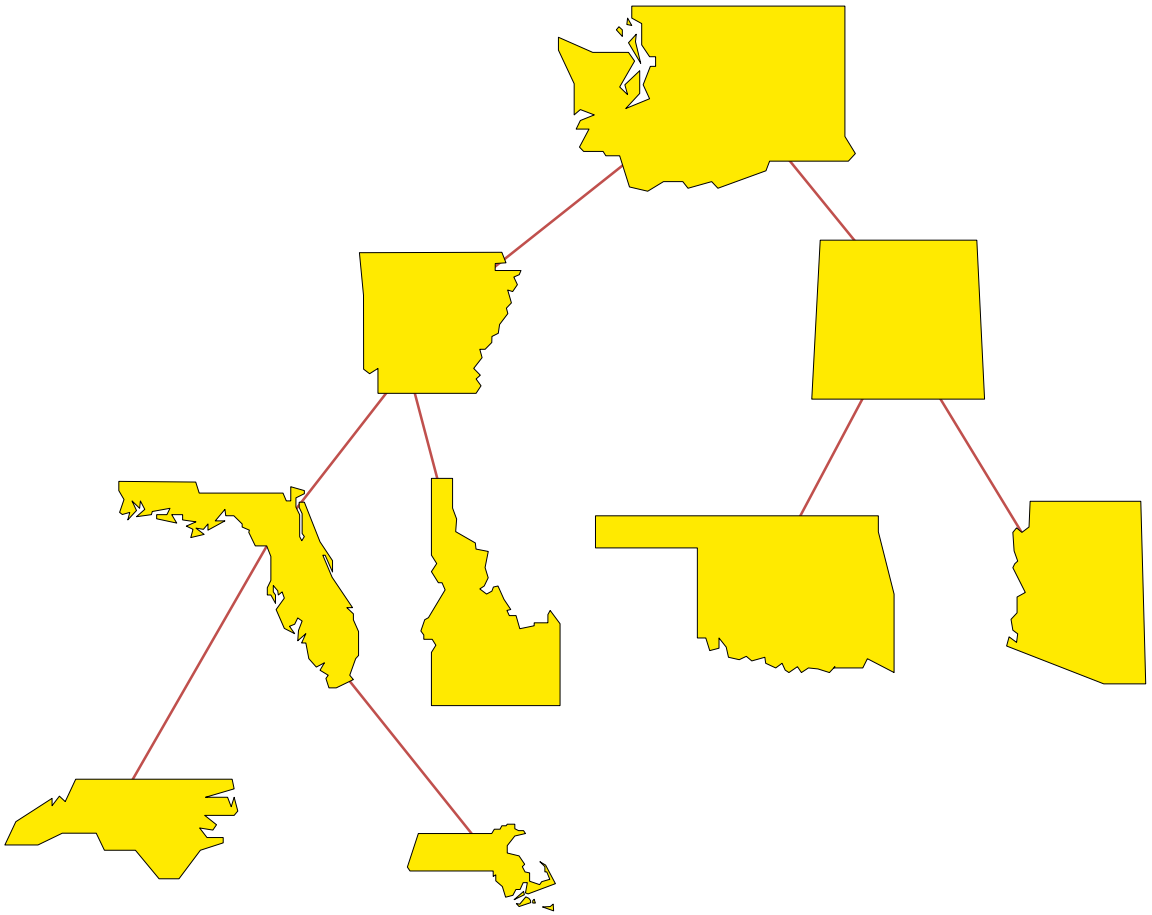
Full Binary Trees

The next nodes must always fill the next level from **left to right**.

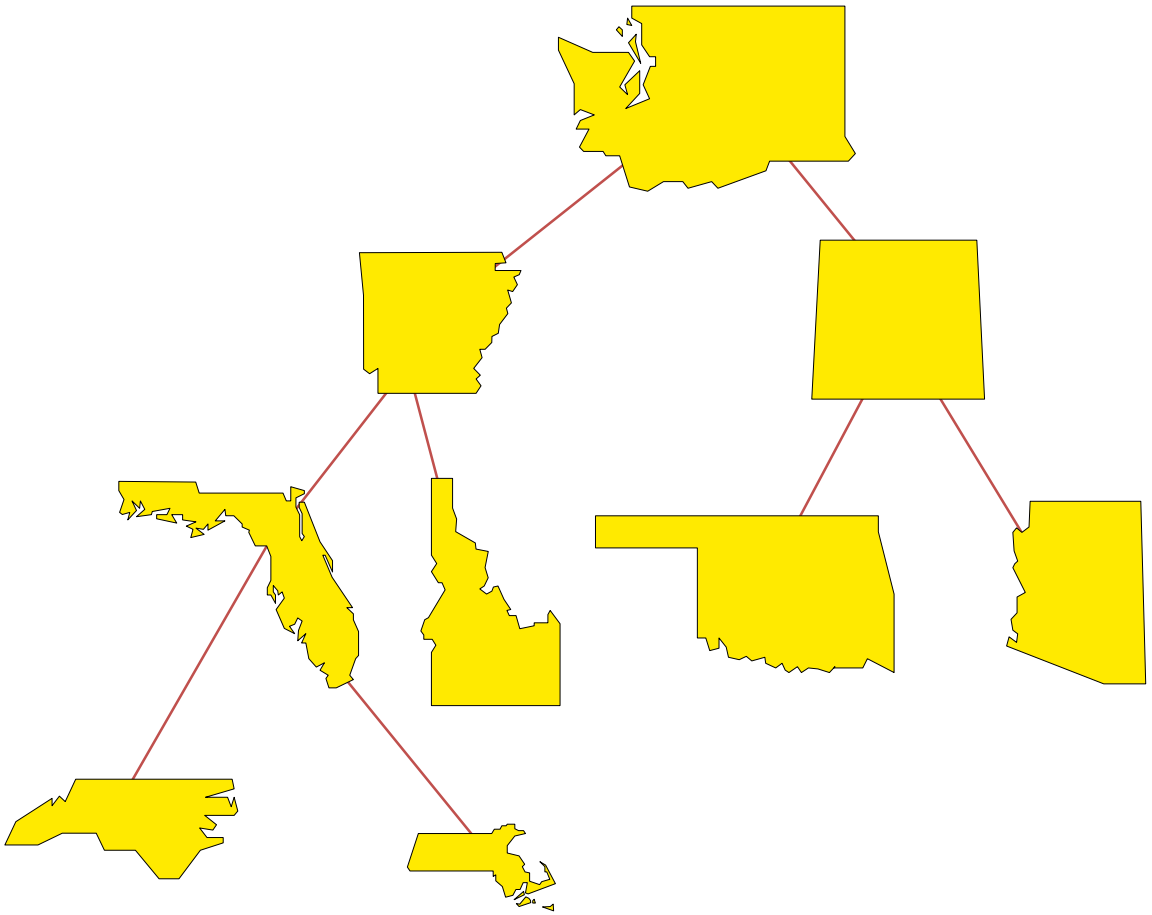


Full Binary Trees

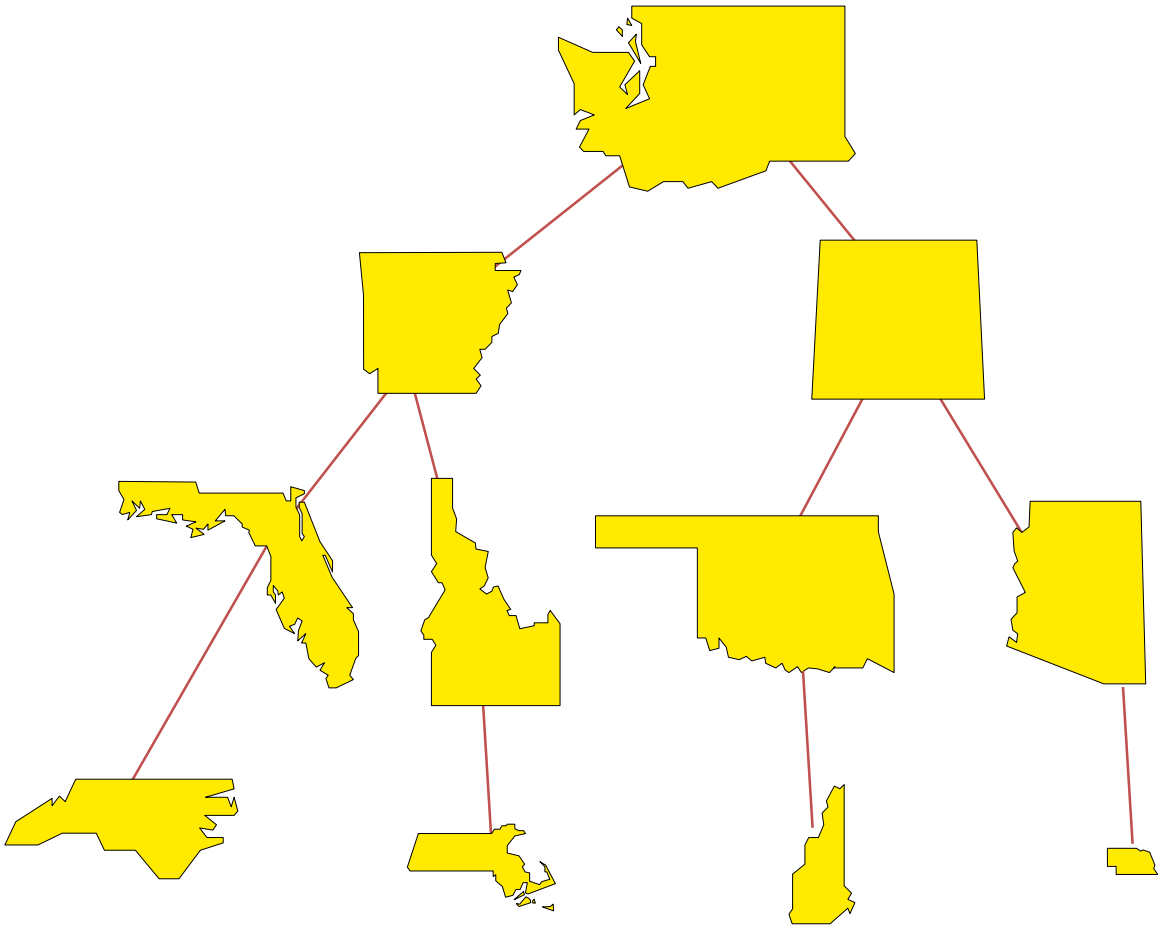
The next nodes must always fill the next level from **left to right**.



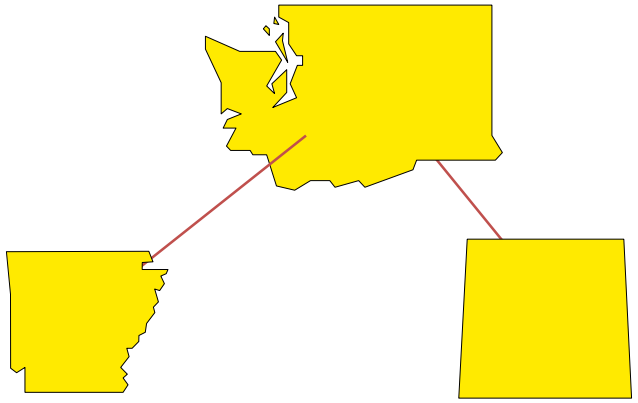
Is This Full?



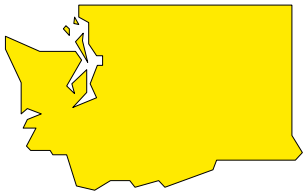
Is This Full?



Is This Full?

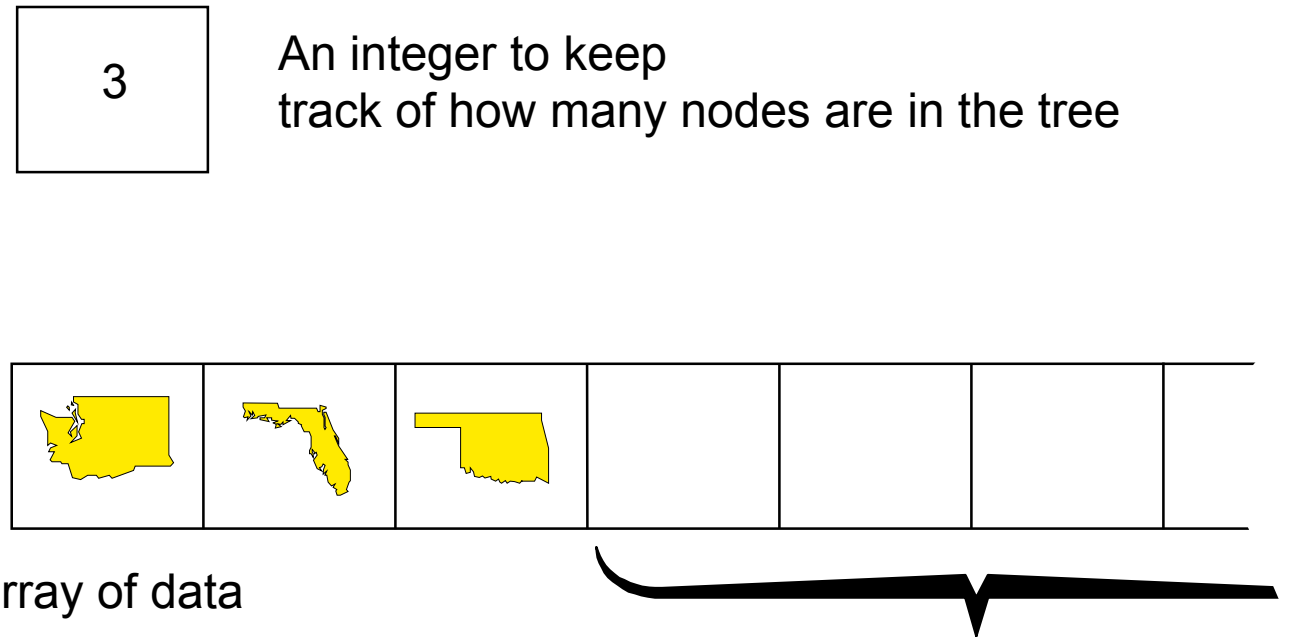


Is This Full?

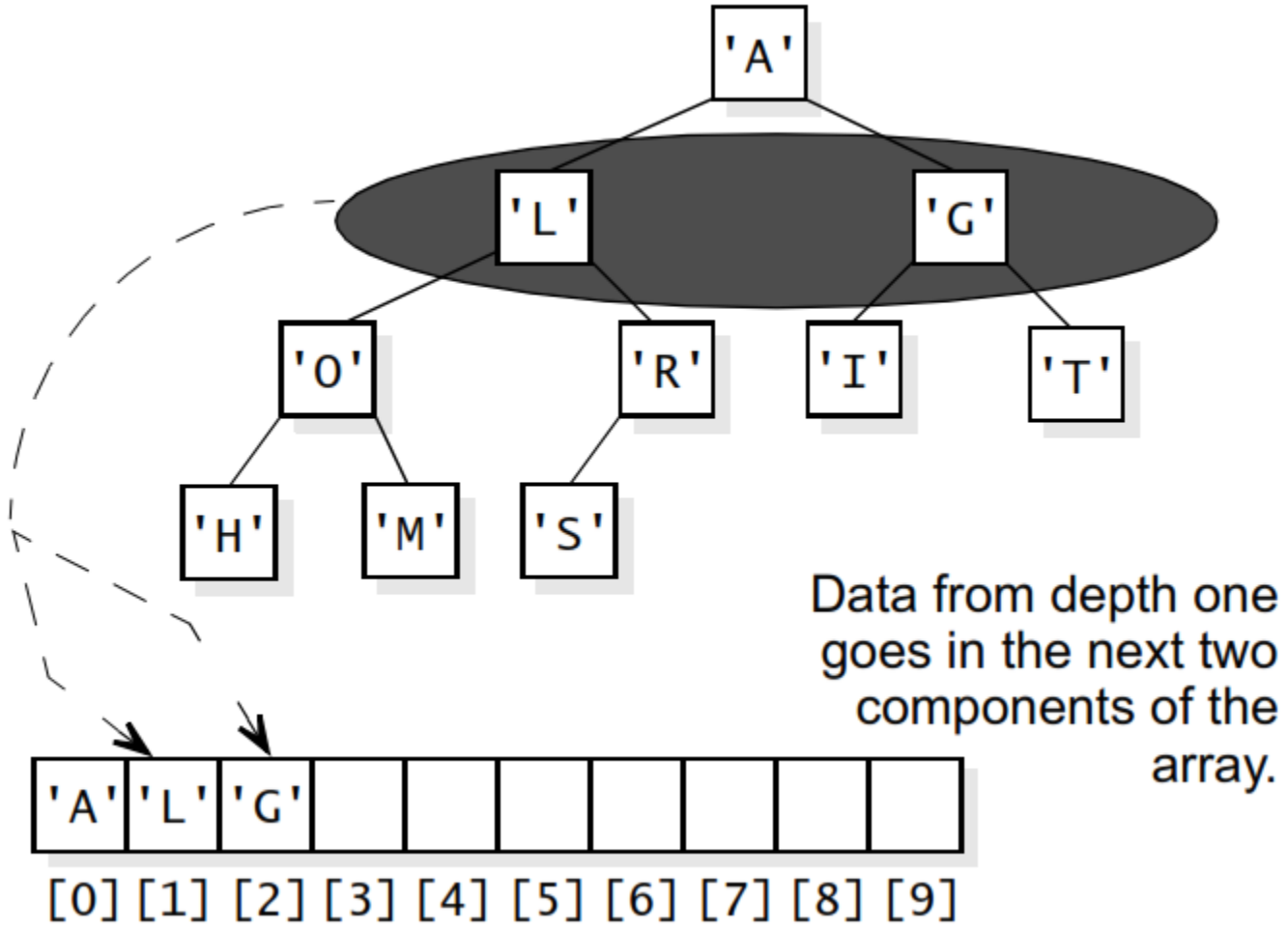


Array implementation of a Complete Binary Tree

- We will store the date from the nodes in a partially-filled array.



Array implementation of a Complete Binary Tree



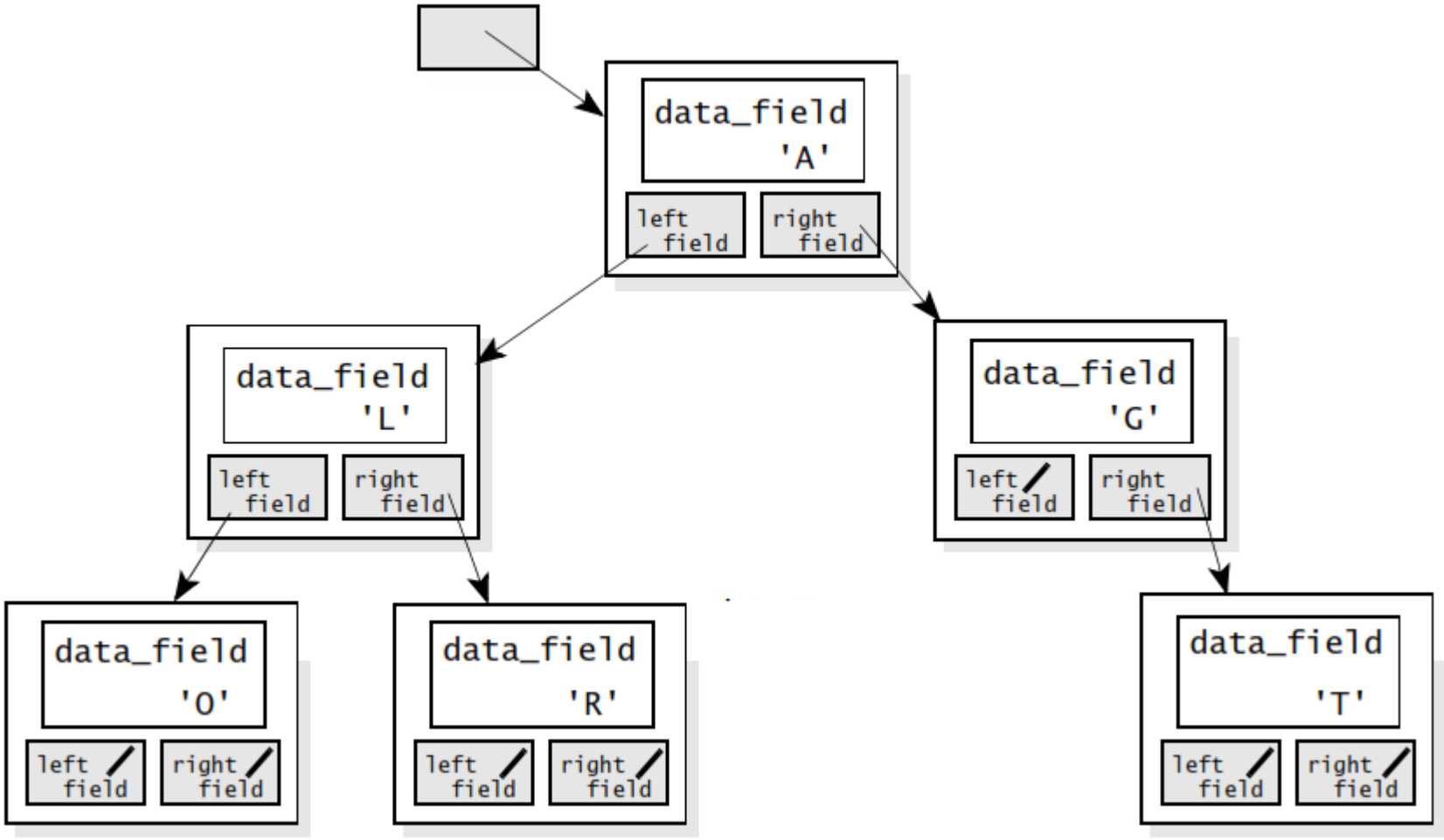
Array implementation of a Complete Binary Tree

- Root is at [0]
- Parent of node in [i] is at [(i-1)/2]
- Children (if exist) of node [i] is at [2i+1] and [2i+2]
- Total node number
  - $2^0+2^1+2^2+\ldots+2^{d-1}+r$, $r <= 2^d$, d is the depth

In a complete binary tree with 10,000 nodes,
suppose that a node has its value stored in location 4999.

Where is the value stored for this node's parent?
Where are the values stored for its left child and right child?

Binary Tree implementation with a Class for Nodes



Binary Tree Nodes

- Each node of a binary tree is stored in an object of a new *binary_tree_node* class
- Each node contains data as well as pointers to its children
- An entire tree is represented as a pointer to the root node

binary_tree_node Class

```
template <class Item>
class binary_tree_node
{
private:
   Item data_field;
   binary_tree_node *left_field;
   binary_tree_node *right_field;
```

```
public:

    // TYPEDEF
    typedef Item value_type;

    // CONSTRUCTOR
    binary_tree_node(
        const Item& init_data = Item( ),
        binary_tree_node* init_left = NULL,
        binary_tree_node* init_right = NULL
    )
    {
        data_field = init_data;
        left_field = init_left;
        right_field = init_right;
    }
```

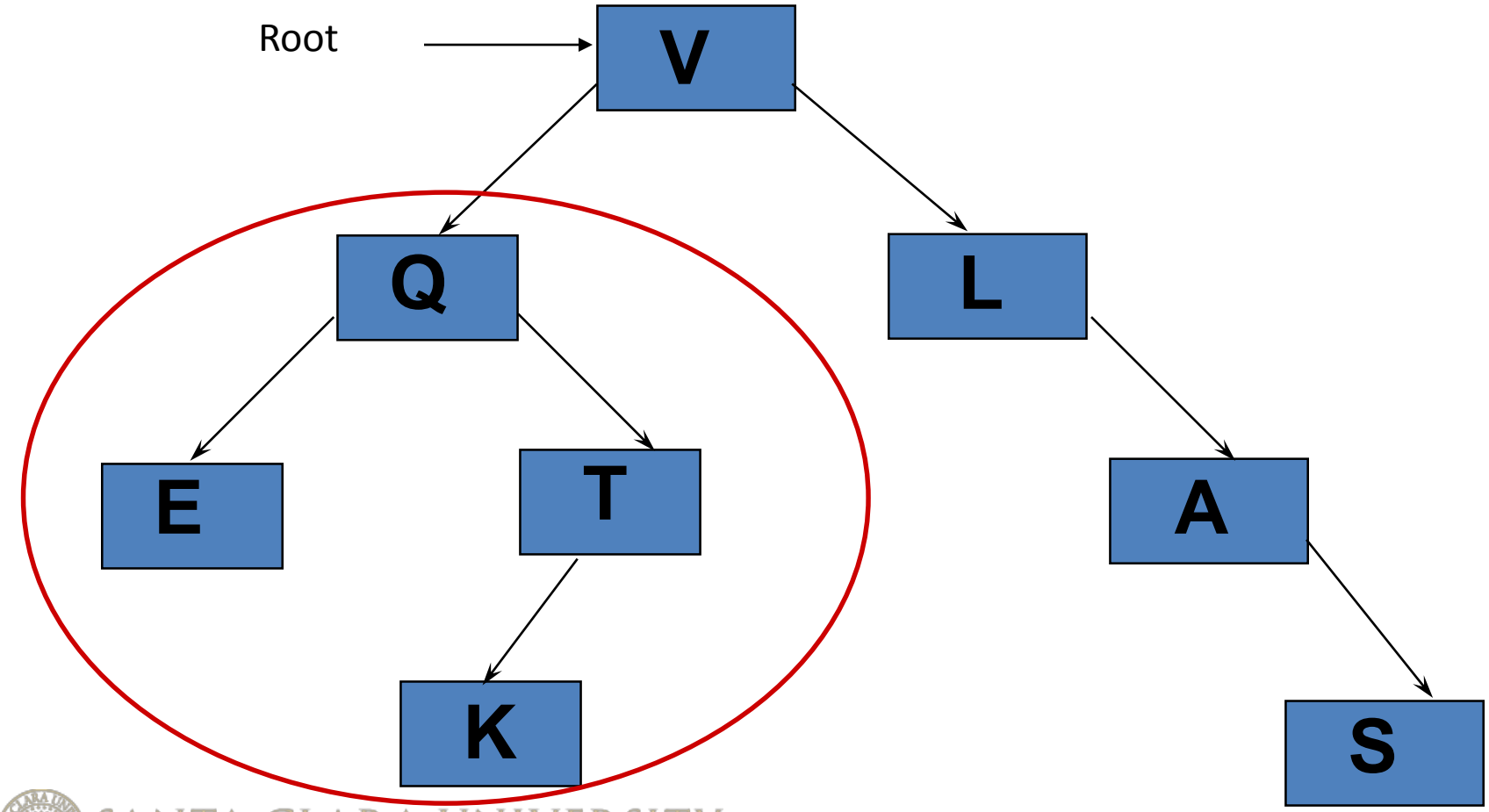
```
// MODIFICATION MEMBER FUNCTIONS
Item& data( ) { return data_field; }
binary_tree_node* left( ) { return left_field; }
binary_tree_node* right( ) { return right_field; }
void set_data(const Item& new_data) { data_field = new_data; }
void set_left(binary_tree_node* new_left) { left_field = new_left; }
void set_right(binary_tree_node* new_right)
                                        { right_field = new_right; }

// CONST MEMBER FUNCTIONS
const Item& data( ) const { return data_field; }
const binary_tree_node* left( ) const { return left_field; }
const binary_tree_node* right( ) const { return right_field; }
bool is_leaf( ) const
    { return (left_field == NULL) && (right_field == NULL); }
```

Creating and Manipulating Trees

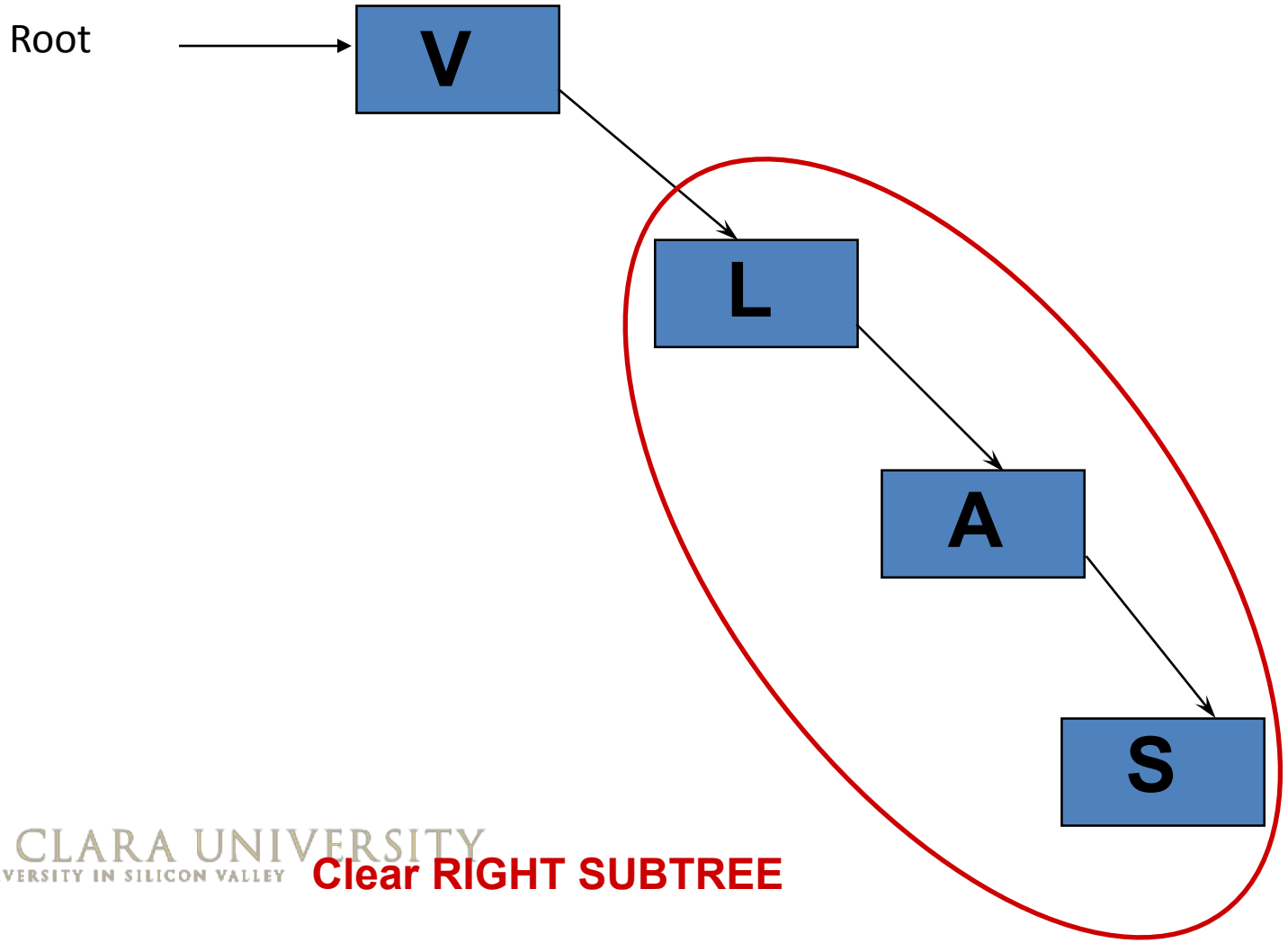
- Consider two functions
  - Clearing a tree
    - ✓Return nodes of a tree to the heap
  - Copying a tree
- The Implementation is easier than it seems
  - if we use recursive thinking

Clearing a Tree



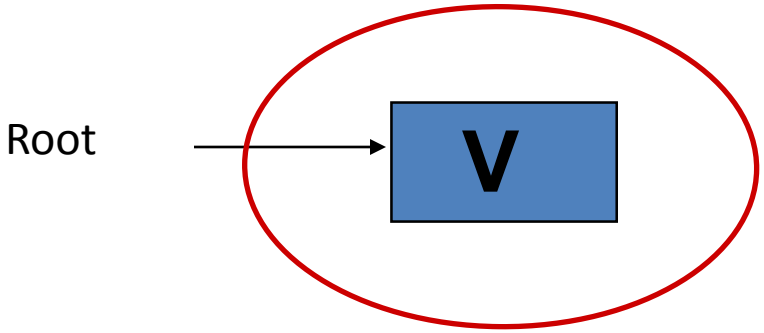
Clear LEFT SUBTREE

Clearing a Tree



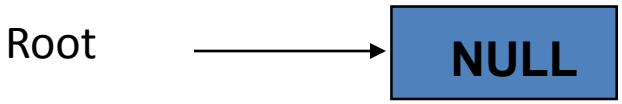
Clear RIGHT SUBTREE

Clearing a Tree



Return root node to the heap

Clearing a Tree



Set the root pointer to NULL

Clear a Tree

- key: recursive thinking

```
template <class Item>
void tree_clear(binary_tree_node<Item>*& root_ptr)
// Library facilities used: cstdlib
{
    if (root_ptr != NULL)
    {
        tree_clear( root_ptr->left( ) ); // clear left subtree
        tree_clear( root_ptr->right( ) ); // clear right subtree
        delete root_ptr;   // return root node to the heap
        root_ptr = NULL; // set root pointer to the null
    }
}
```

Copy a Tree

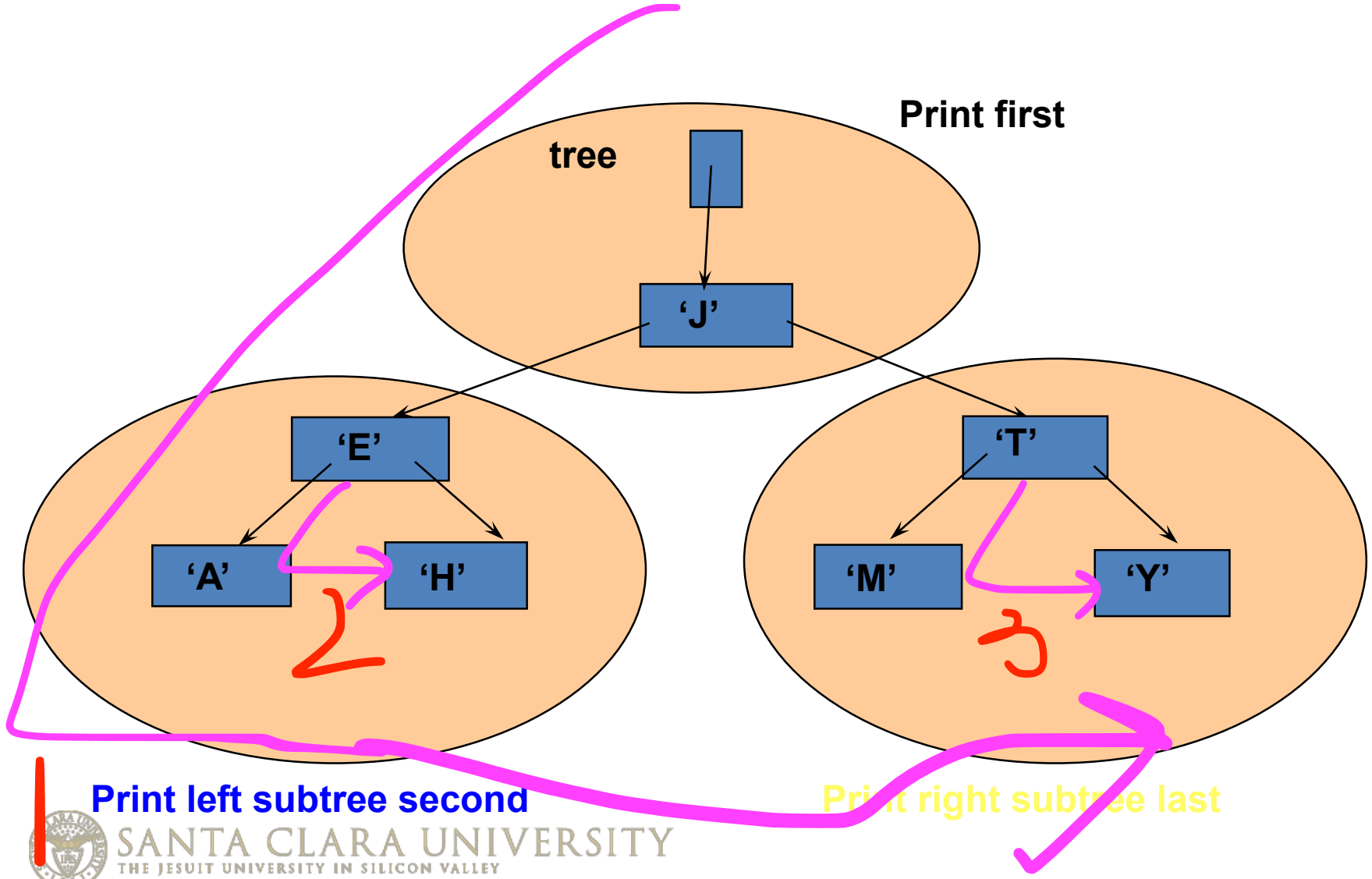
```
template <class Item>
binary_tree_node<Item>* tree_copy
                    (const binary_tree_node<Item>* root_ptr)
// Library facilities used: cstdlib
{
    binary_tree_node<Item> *l_ptr;
    binary_tree_node<Item> *r_ptr;

    if (root_ptr == NULL)
        return NULL;
    else
    {
        // copy the left sub_tree
        l_ptr = tree_copy( root_ptr->left( ) );
        // copy the right sub_tree
        r_ptr = tree_copy( root_ptr->right( ) );
        return new binary_tree_node<Item>
                            (root_ptr->data( ), l_ptr, r_ptr);
    }  // copy the root node and set the root pointer
}
```

Binary Tree Traversals

- pre-order traversal
  - root (left sub_tree) (right sub_tree)
- in-order traversal
  - (left sub_tree) root (right sub_tree)
- post-order traversal
  - (left sub_tree) (right sub_tree) root
- backward in-order traversal
  - (right sub_tree) root (left sub_tree)

Preorder Traversal: J E A H T M Y

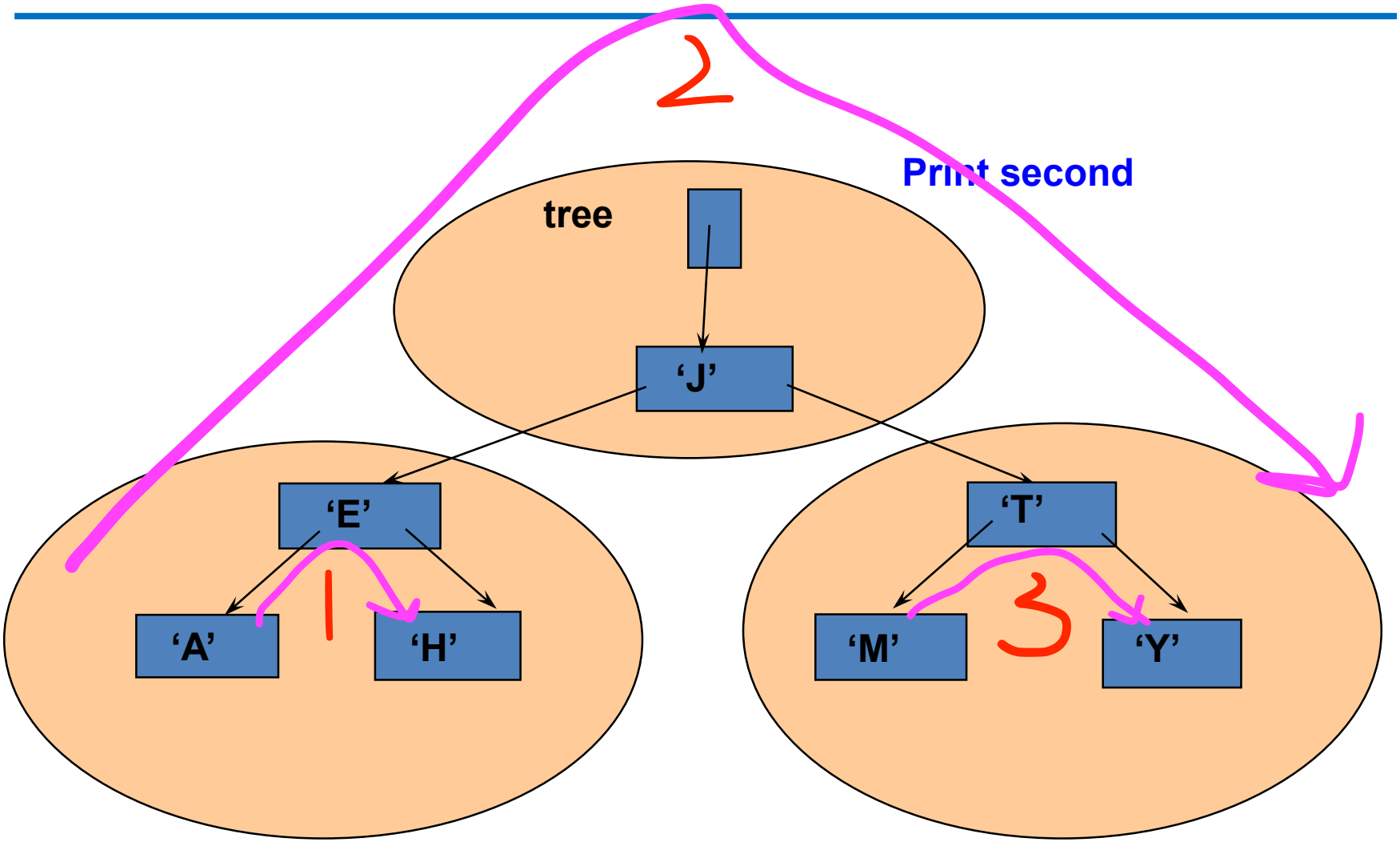


Preorder Traversal

- Example: print the contents of each node

```
template <class Item>
void preorder_print(const binary_tree_node<Item>* node_ptr)
// Library facilities used: cstdlib, iostream
{
    if (node_ptr != NULL)
    {
        std::cout <<  node_ptr->data( ) << std::endl;
        preorder_print(node_ptr->left( ));
        preorder_print(node_ptr->right( ));
    }
}
```

Inorder Traversal: A E H J M T Y



Print left subtree first

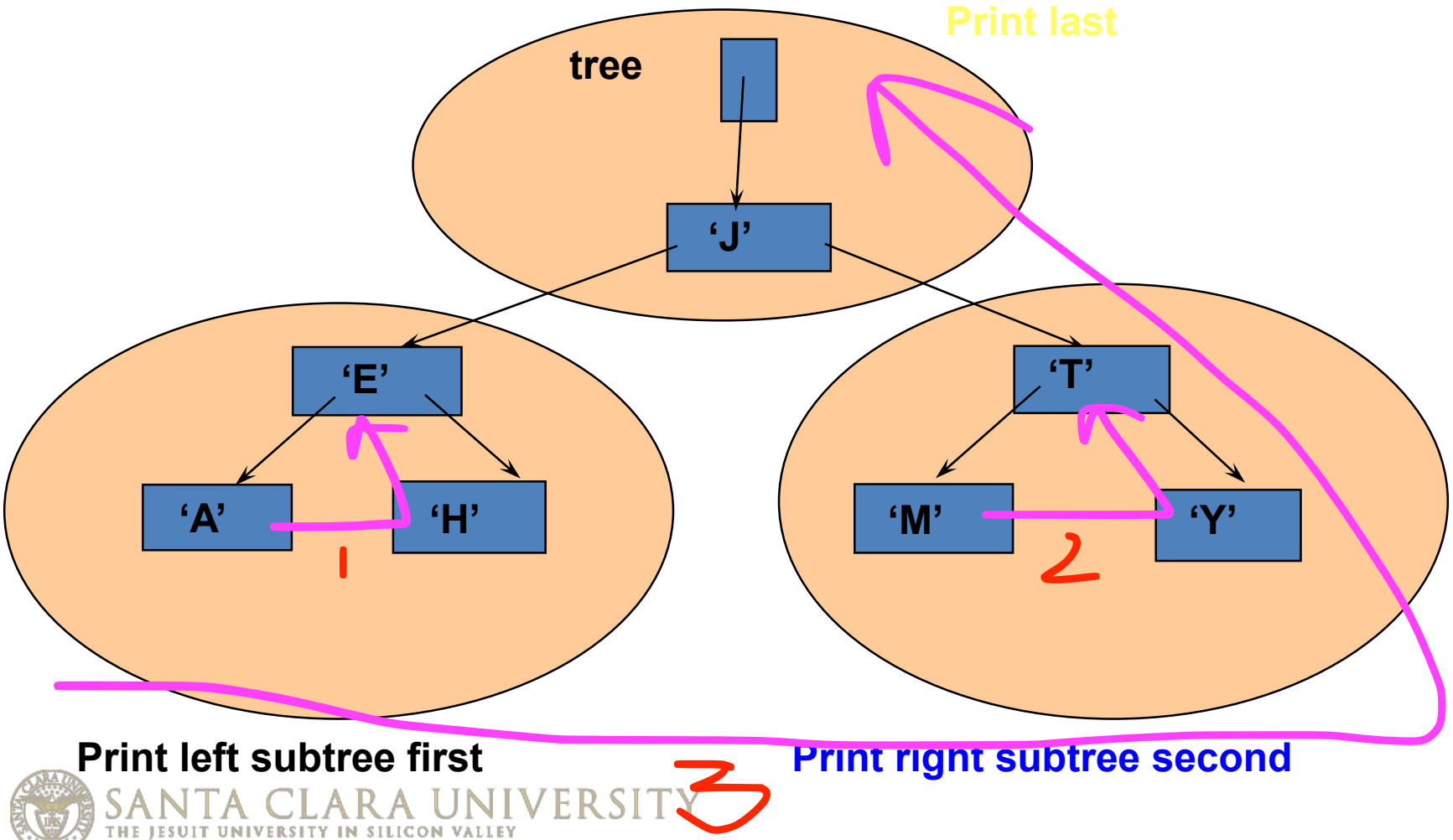
Print right subtree last

Inorder Traversal

- Example: print the contents of each node

```
template <class Item>
void inorder_print(const binary_tree_node<Item>* node_ptr)
// Library facilities used: cstdlib, iostream
{
    if (node_ptr != NULL)
    {
        inorder_print(node_ptr->left( ));
        std::cout <<  node_ptr->data( ) << std::endl;
        inorder_print(node_ptr->right( ));
    }
}
```

Postorder Traversal: A H E M Y T J

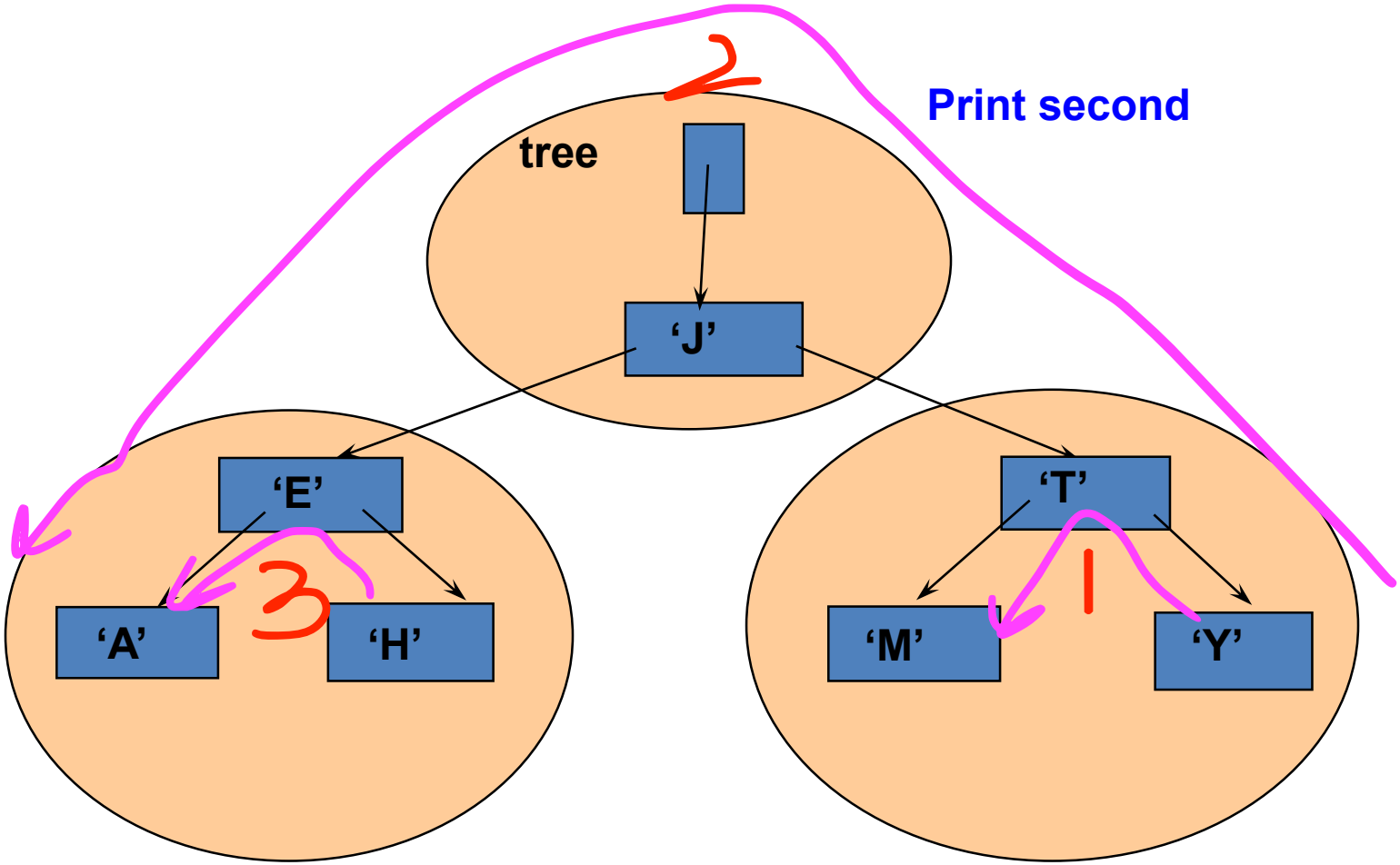


Postorder Traversal

- Example: print the contents of each node

```
template <class Item>
void postorder_print(const binary_tree_node<Item>* node_ptr)
// Library facilities used: cstdlib, iostream
{
    if (node_ptr != NULL)
    {
        postorder_print(node_ptr->left( ));
        postorder_print(node_ptr->right( ));
        std::cout <<  node_ptr->data( ) << std::endl;
    }
}
```

Backward Inorder Traversal: Y T M J H E A



A Useful Backward Inorder Traversal

- Indent each letter according its depth

```
template <class Item, class SizeType>
void print(binary_tree_node<Item>* node_ptr, SizeType depth)
// Library facilities used: iomanip, iostream, stdlib
{
    if (node_ptr != NULL)
    {
        print(node_ptr->right( ), depth+1);
        std::cout << std::setw(4*depth) << "";
        std::cout << node_ptr->data( ) << std::endl;
        print(node_ptr->left( ),  depth+1);
    }
}
```

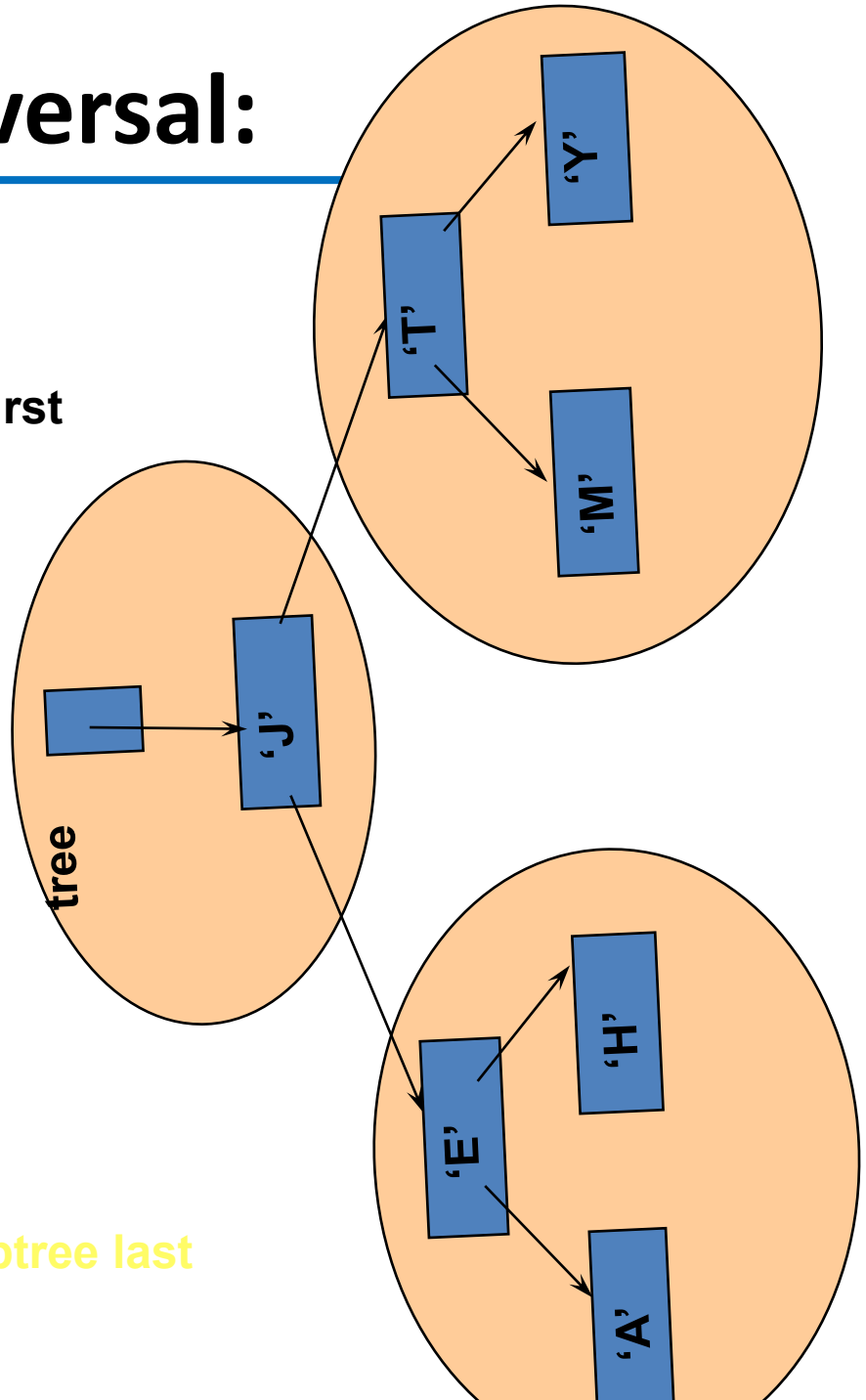
Backward Inorder Traversal:

Y T M J H E A

Print right subtree first

Print second

Print left subtree last



A Challenging Question:

- For the traversals we have seen, the "processing" was simply printing the values of the node
- But we'd like to do any kind of processing
  - We can replace "cout" with some other form of "processing"
- But how about 1000 kinds of processing?
  - Can template be helpful?

A parameter can be a function

- write one function capable of doing anything
- A parameter to a function may be a function. Such a parameter is declared by
  - the name of the function's return type (or void),
  - then the name of the parameter (i.e. the function),
  - and finally a pair of parentheses ( ).
  - Inside ( ) is a list of parameter types of that parameter function
- Example
  - int sum (void f (int&, double), int i,...);

Preorder Traversal – print only

- Example: print the contents of each node

```
template <class Item>
void preorder_print(const binary_tree_node<Item>* node_ptr)
// Library facilities used: cstdlib, iostream
{
    if (node_ptr != NULL)
    {
        std::cout <<  node_ptr->data( ) << std::endl;
        preorder_print(node_ptr->left( ));
        preorder_print(node_ptr->right( ));
    }
}
```

Preorder Traversal – general form

- A template function for tree traversals

```
template <class Item>
void preorder(void f(Item&), binary_tree_node<Item>* node_ptr)
// Library facilities used: cstdlib
{
    if (node_ptr != NULL)
    {
        // node_ptr->data() return reference !
        f( node_ptr->data( ) );
        preorder(f, node_ptr->left( ));
        preorder(f, node_ptr->right( ));
    }
}
```

Preorder Traversal – how to use

- Define a real function before calling

```
void printout(int & it)
   // Library facilities used: iostream
{
   std::cout <<  it << std::endl;
}
```

- Can you print out all the node of a tree pointed by root ?

```
binary_tree_node<int> *root;
....
preorder(printout, root);
```

Preorder Traversal – another functions

- Can define other functions...

```
void assign_default(int& it)
   // Library facilities used: iostream
{
      it  = 0;
}
```

- You can assign a default value to all the node of a tree pointed by root:

```
binary_tree_node<int> *root;
....
preorder(assign_default, root);
```

Preorder Traversal – a more general form

- An extremely general implementation

```
template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr)
// Note: BTNode may be a binary_tree_node or a const
// binary tree node.
// Process is the type of a function f that may be
// called with a single Item argument (using the Item
// type from the node), as determined by the actual f
// in the following.
    // Library facilities used: cstdlib
{
    if (node_ptr != NULL)
    {
        f( node_ptr->data( ) );
        preorder(f, node_ptr->left( ));
        preorder(f, node_ptr->right( ));
    }
}
```

Functions as Parameters

- We can define a template function *X* with functions as parameters – which are called *function parameters*
- A function parameter can be simply written as *Process f* (where Process is a template), and the forms and number of parameters for *f* are determined by the actual call of *f* inside the template function *X*
- The function argument for *f* when calling the template function *X* cannot be a template function, it must be instantiated in advance or right in the function call

Summary

- Tree, Binary Tree, Complete Binary Tree
  - child, parent, sibling, root, leaf, ancestor,...
- Array Representation for Complete Binary Tree
  - Difficult if not complete binary tree
- A Class of binary_tree_node
  - each node with two link fields
- Tree Traversals
  - recursive thinking makes things much easier
- A general Tree Traversal
  - A Function as a parameter of another function

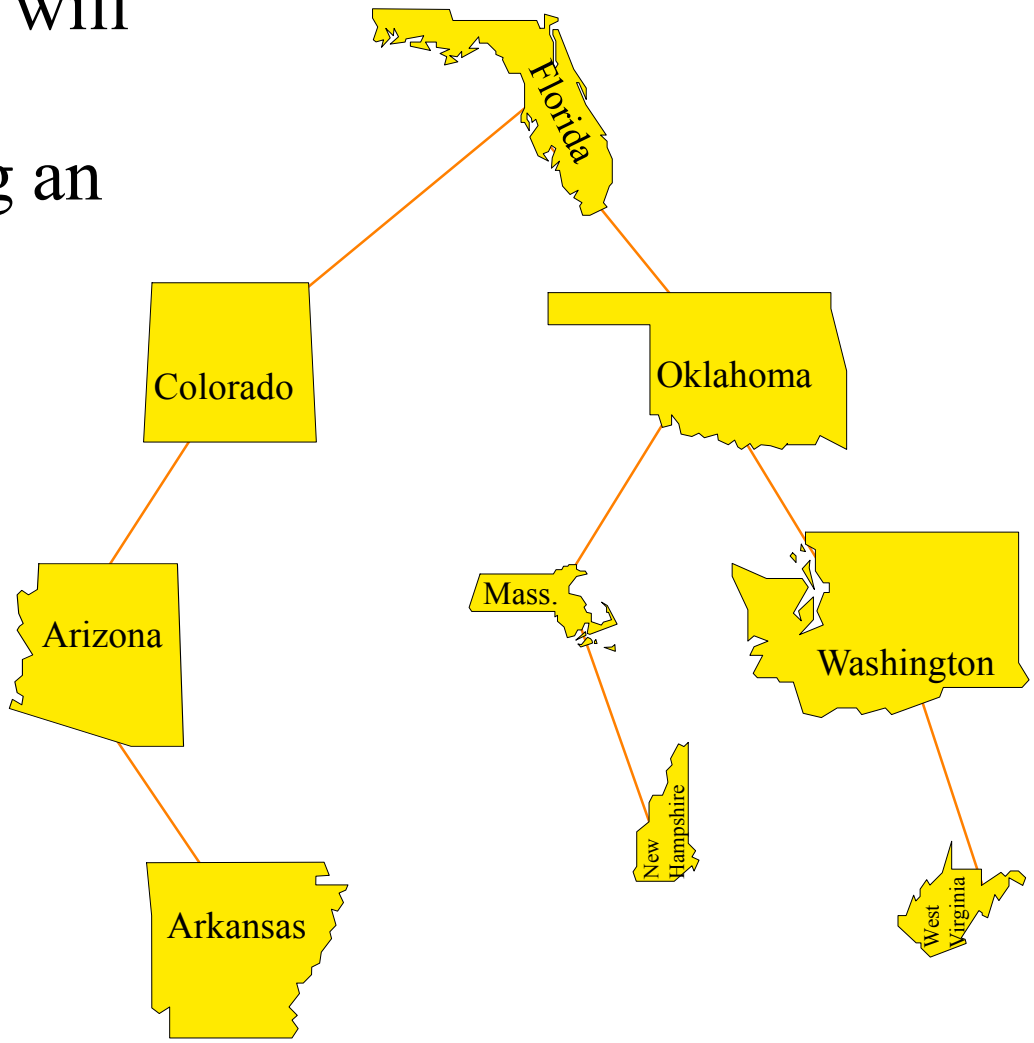
BINARY SEARCH TREES

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.

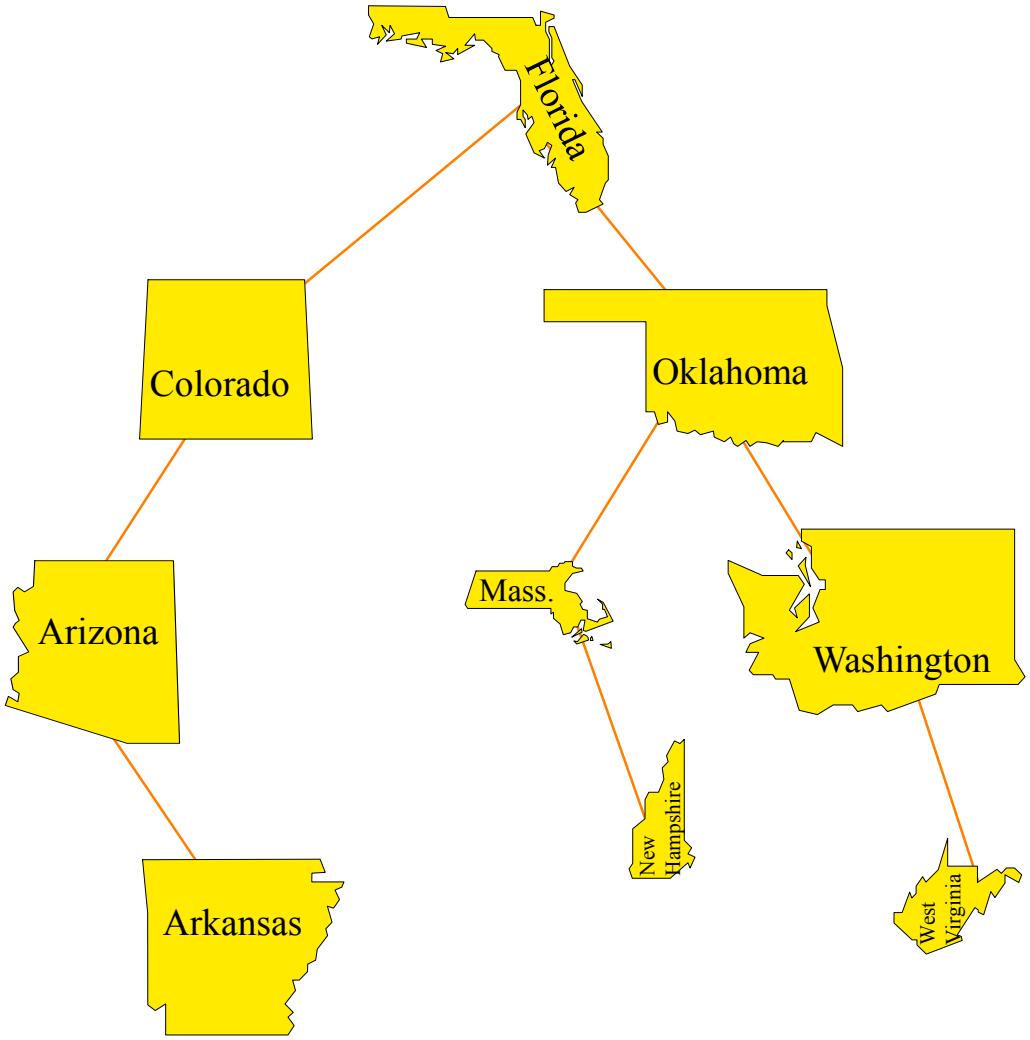
A Binary Search Tree of States

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

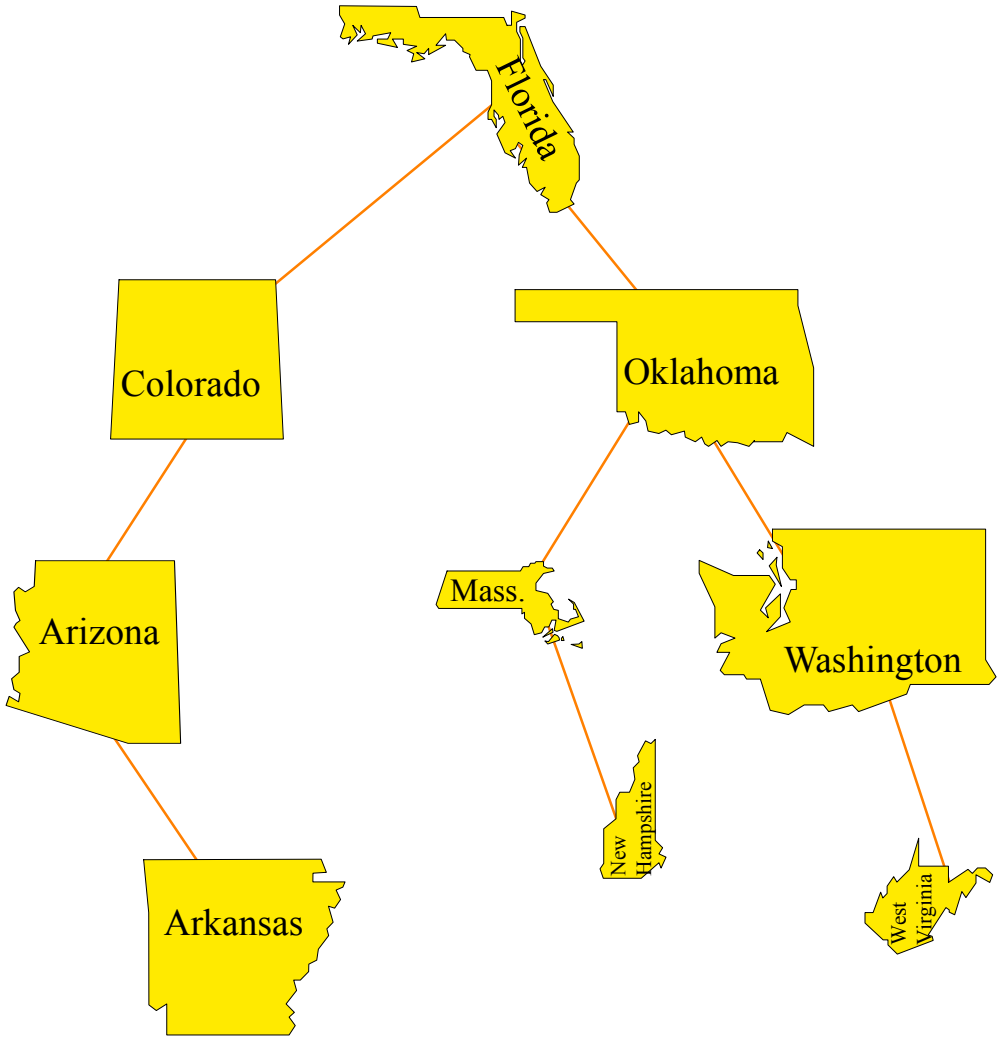
- Storage rules:
  - Every key to the **left** of a node is alphabetically **before** the key of the node.
  - Every key to the **right** of a node is alphabetically **after** the key of the node.



Retrieving Data

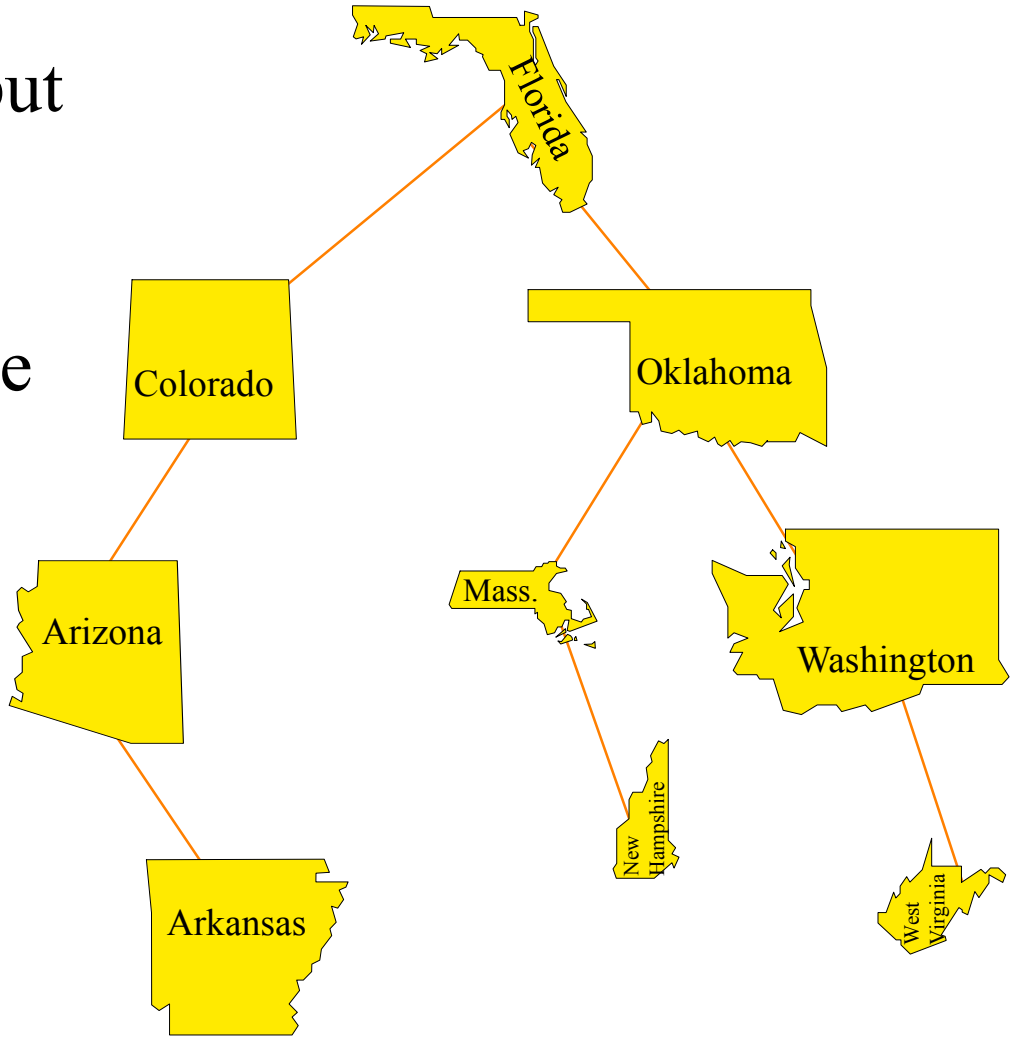
Start at the root.

- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too **large**, move **left**.
- If the current node's key is too **small**, move **right**.



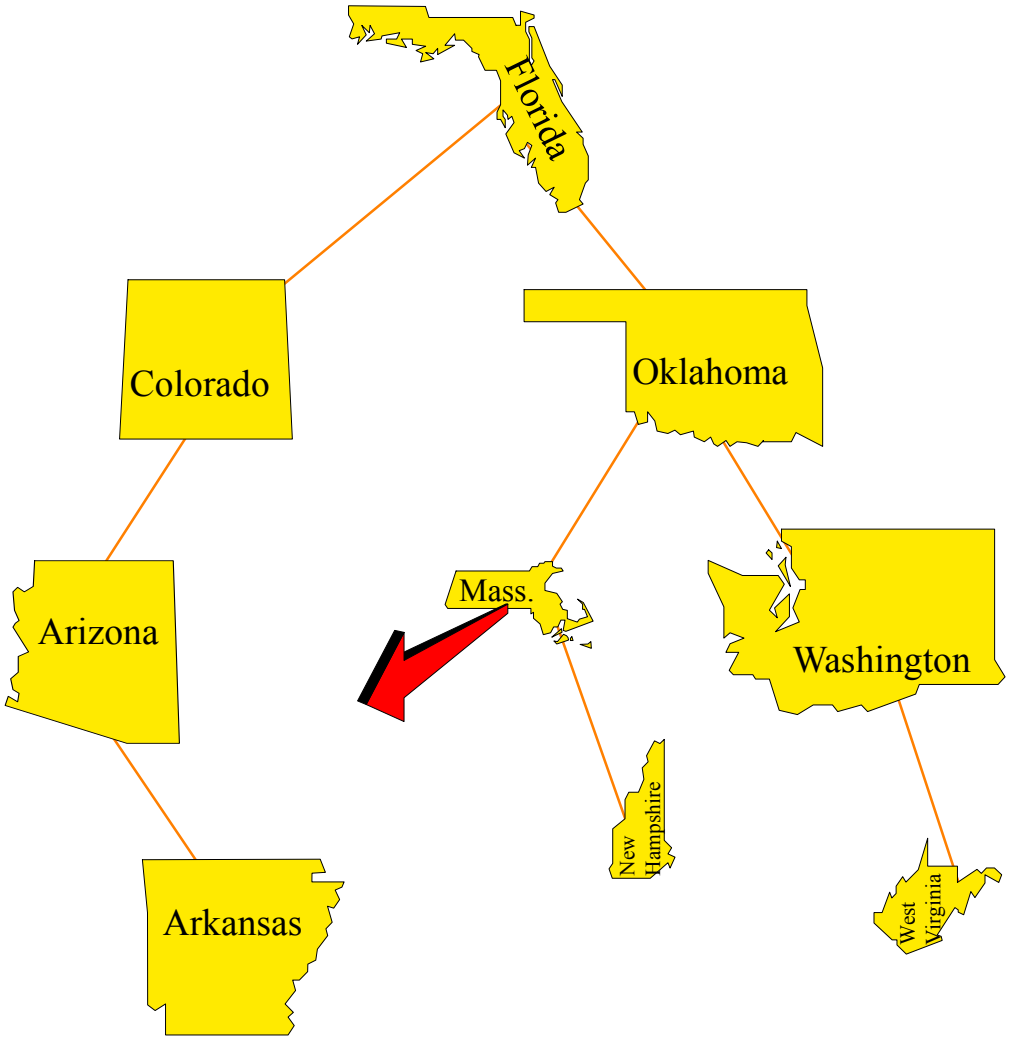
Adding a New Item with a Given Key

- 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.



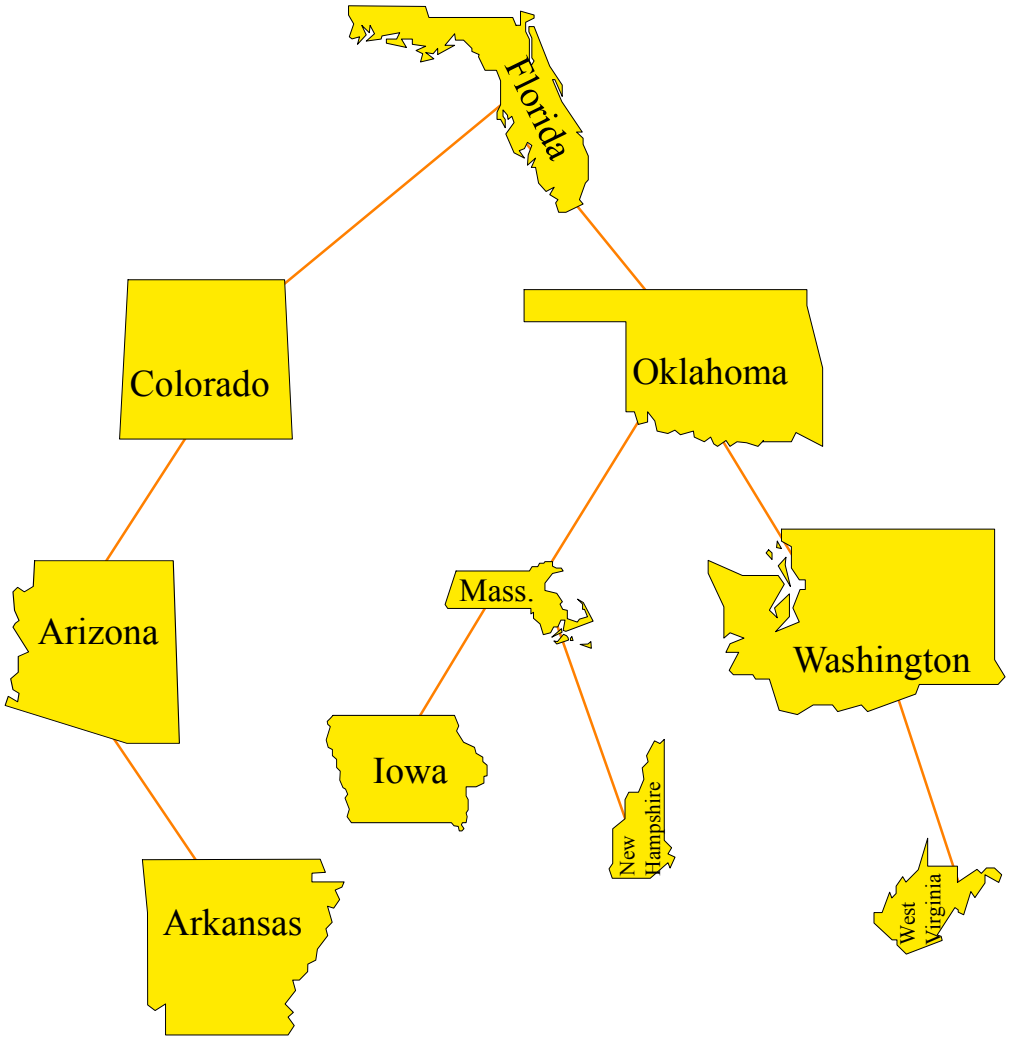
Adding

- 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.



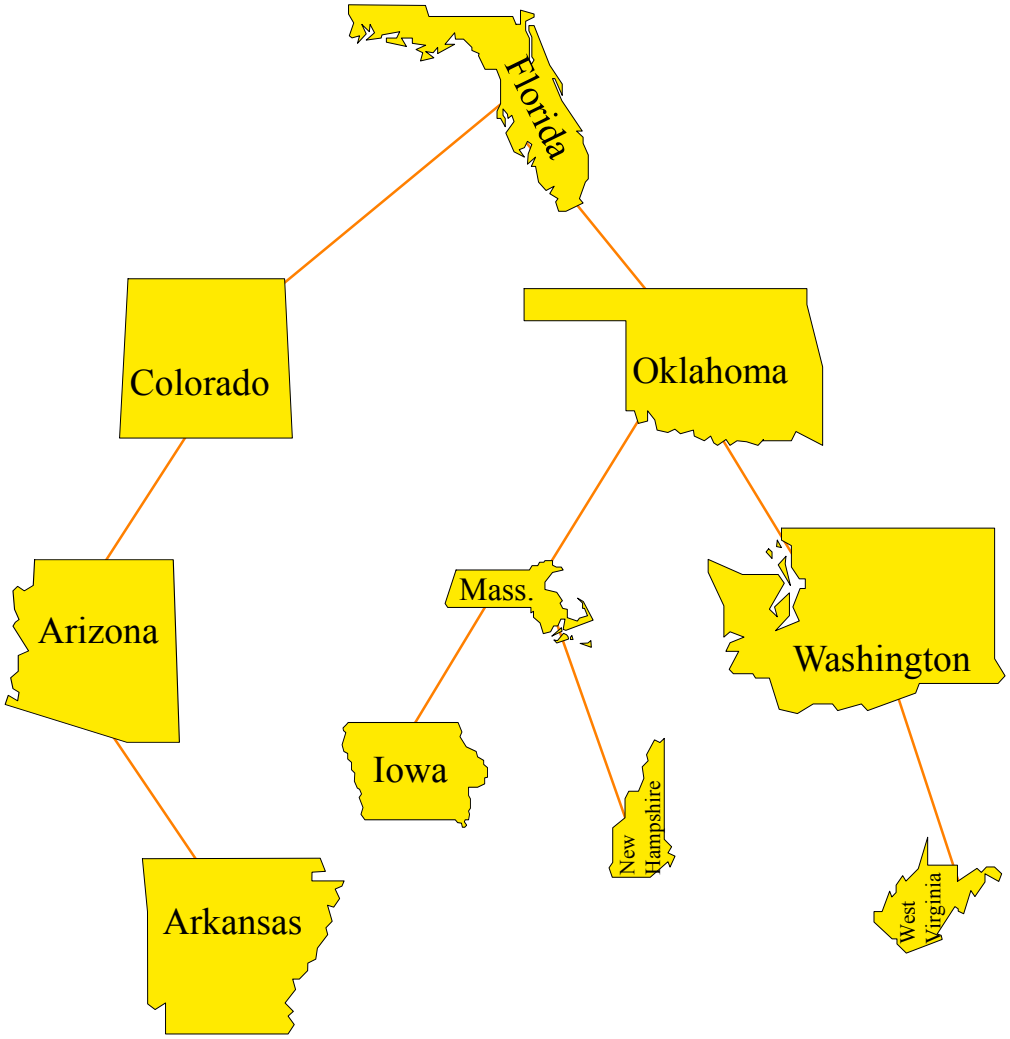
Adding

- 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.



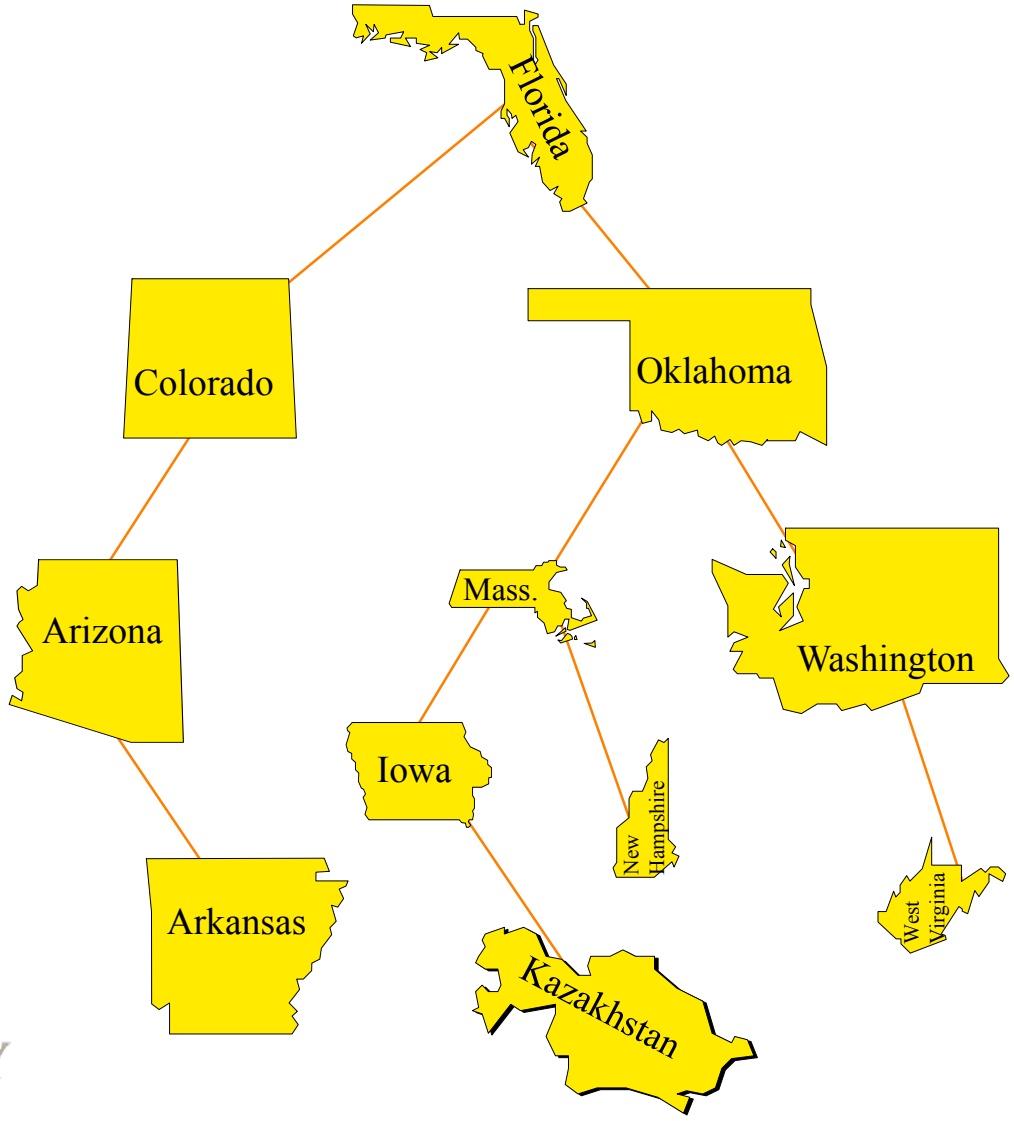
Adding

Where would you add this state?



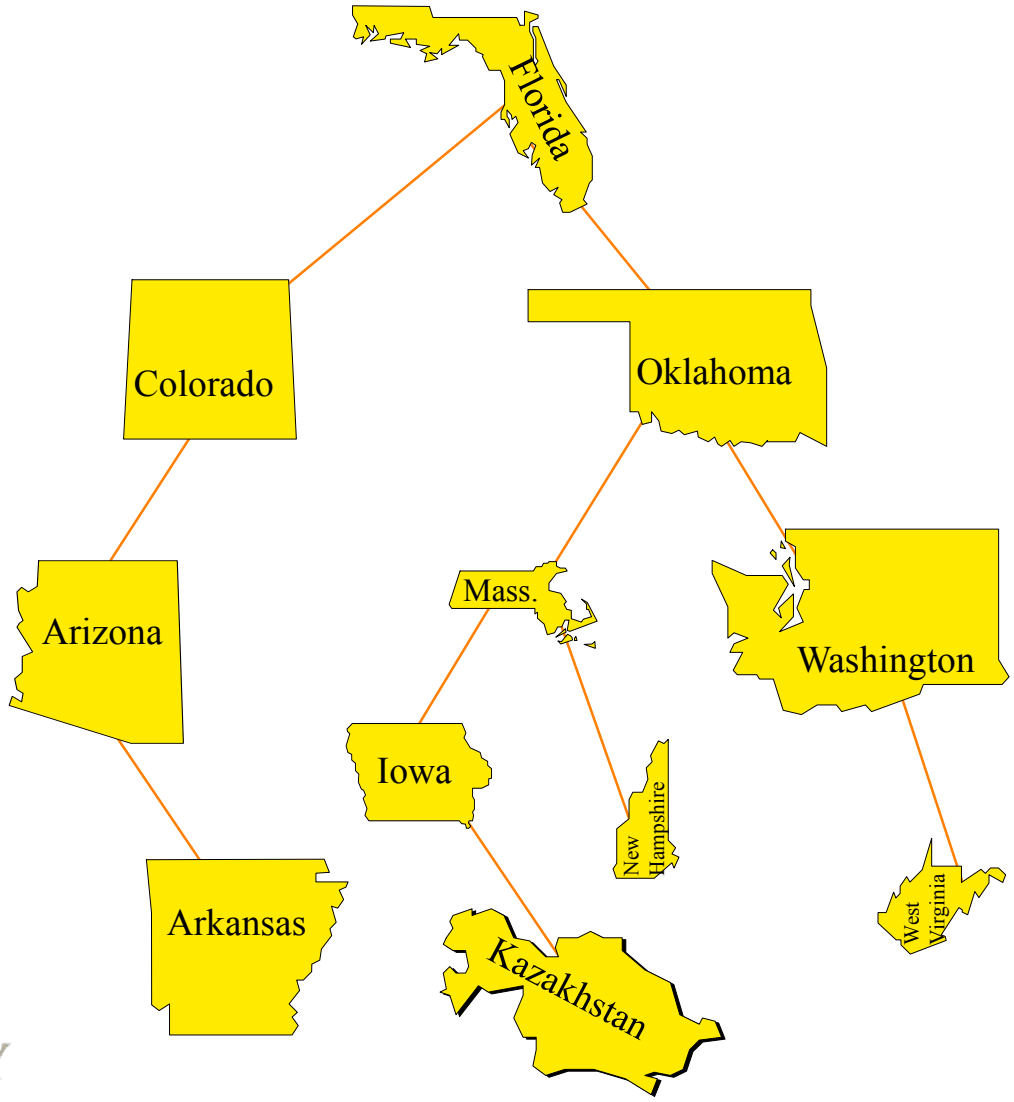
Adding

Kazakhstan is the new right child of Iowa?



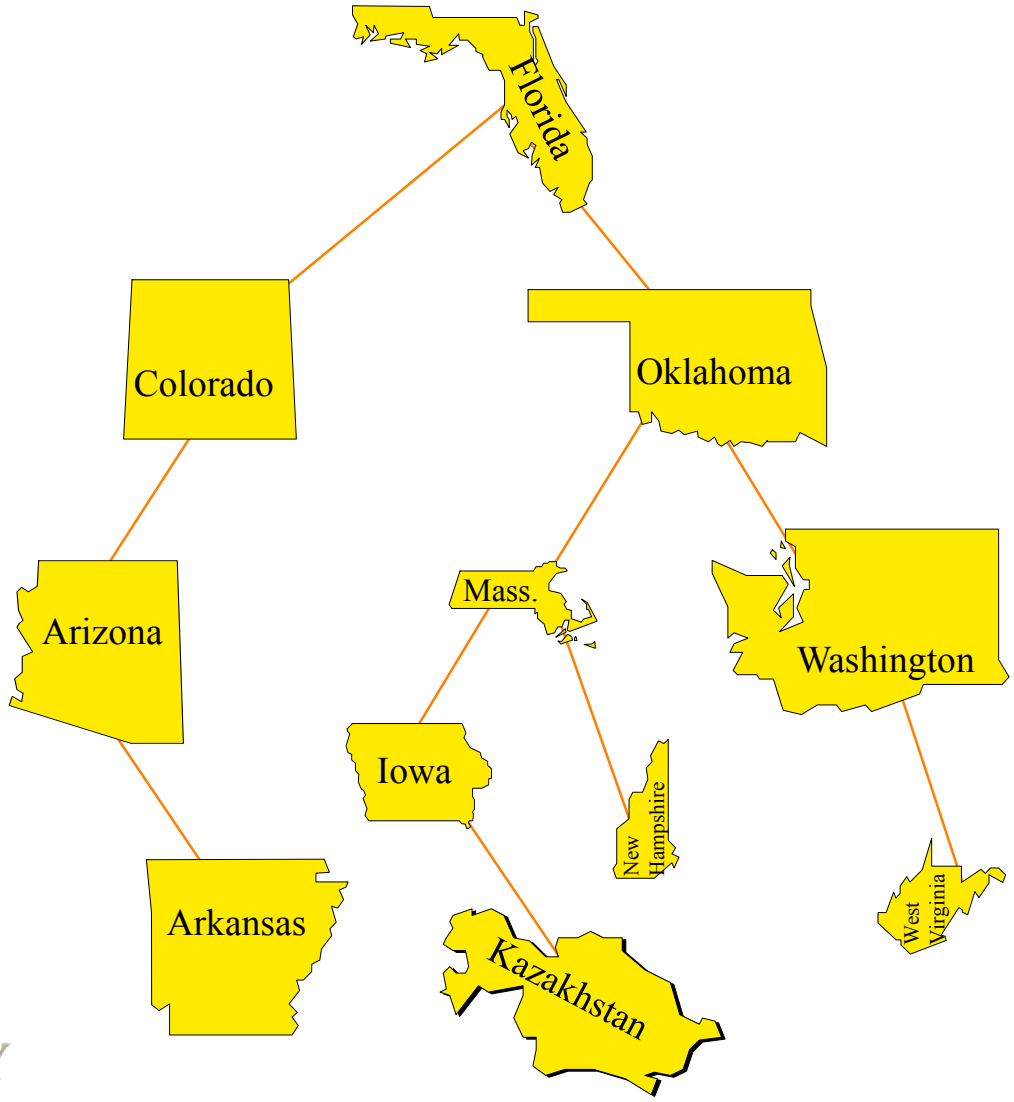
Removing an Item with a Given Key

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



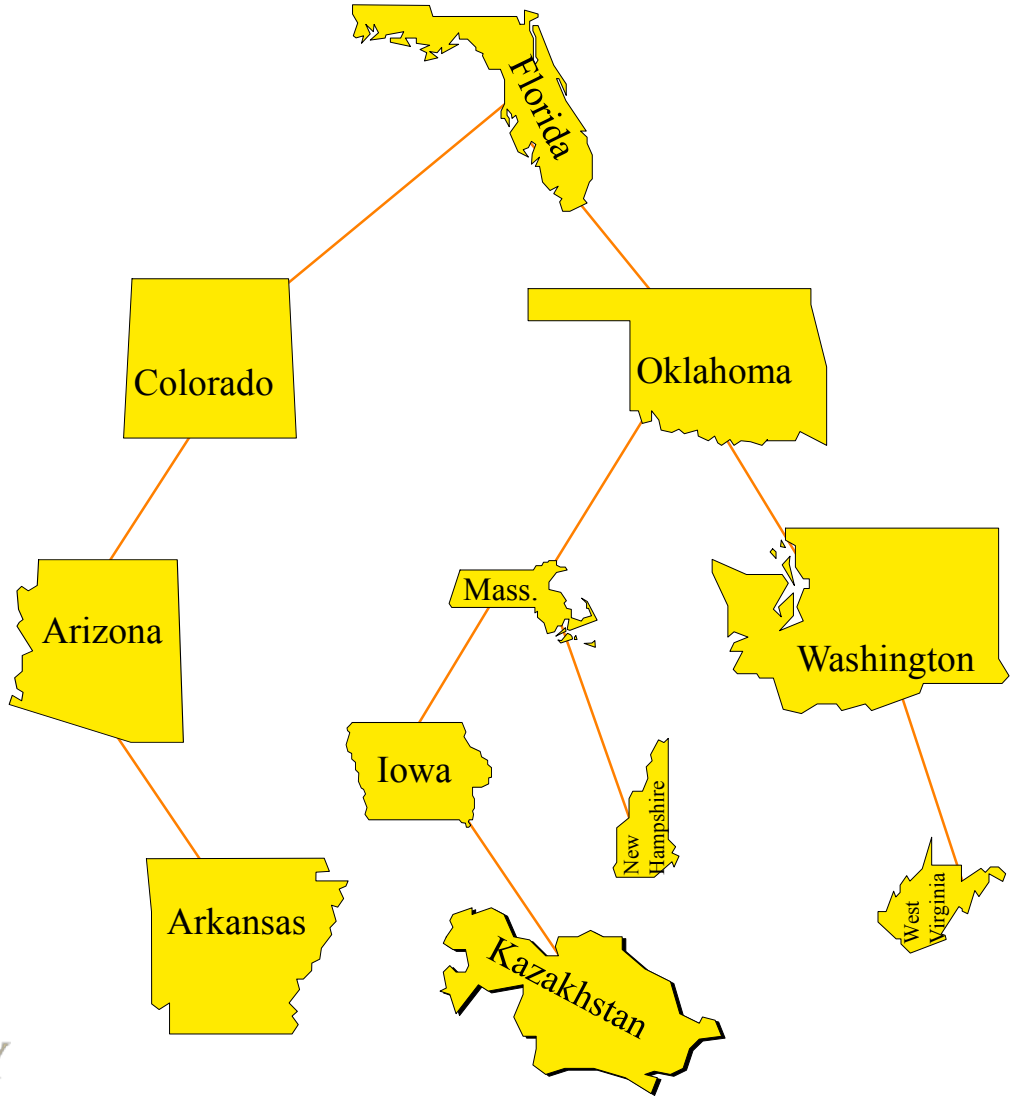
Removing "Florida"

- **Find** the item.



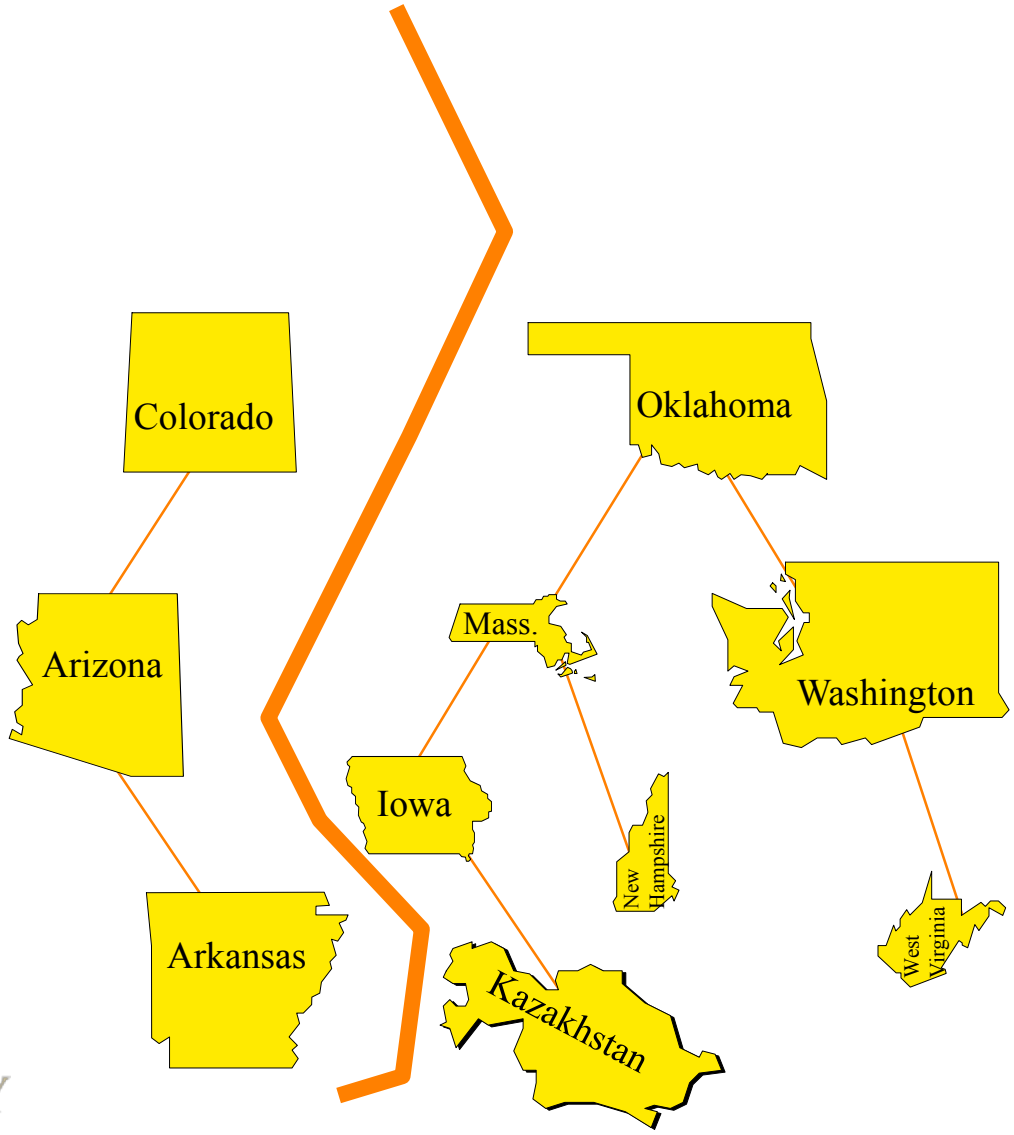
Removing "Florida"

Florida cannot be removed at the moment...



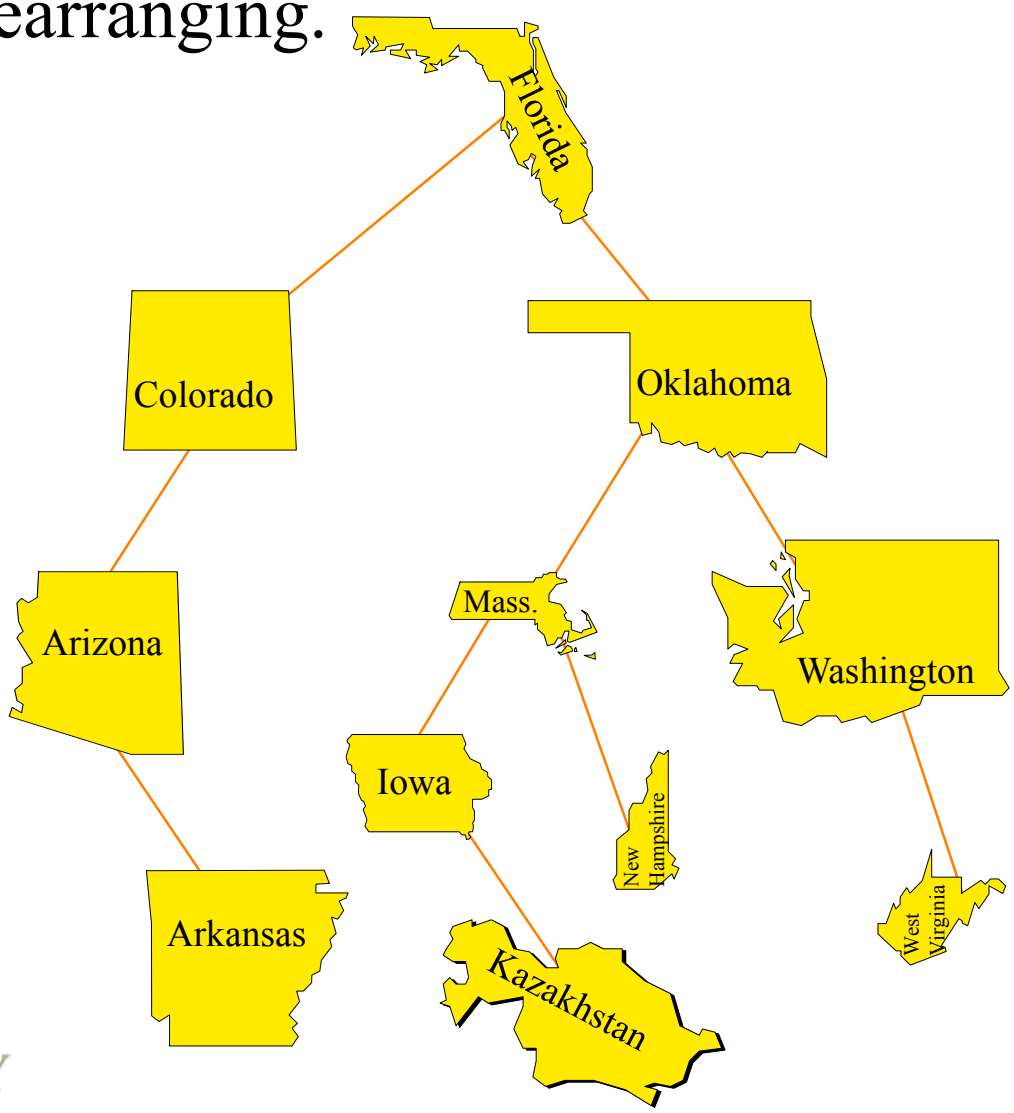
Removing "Florida"

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



Removing "Florida"

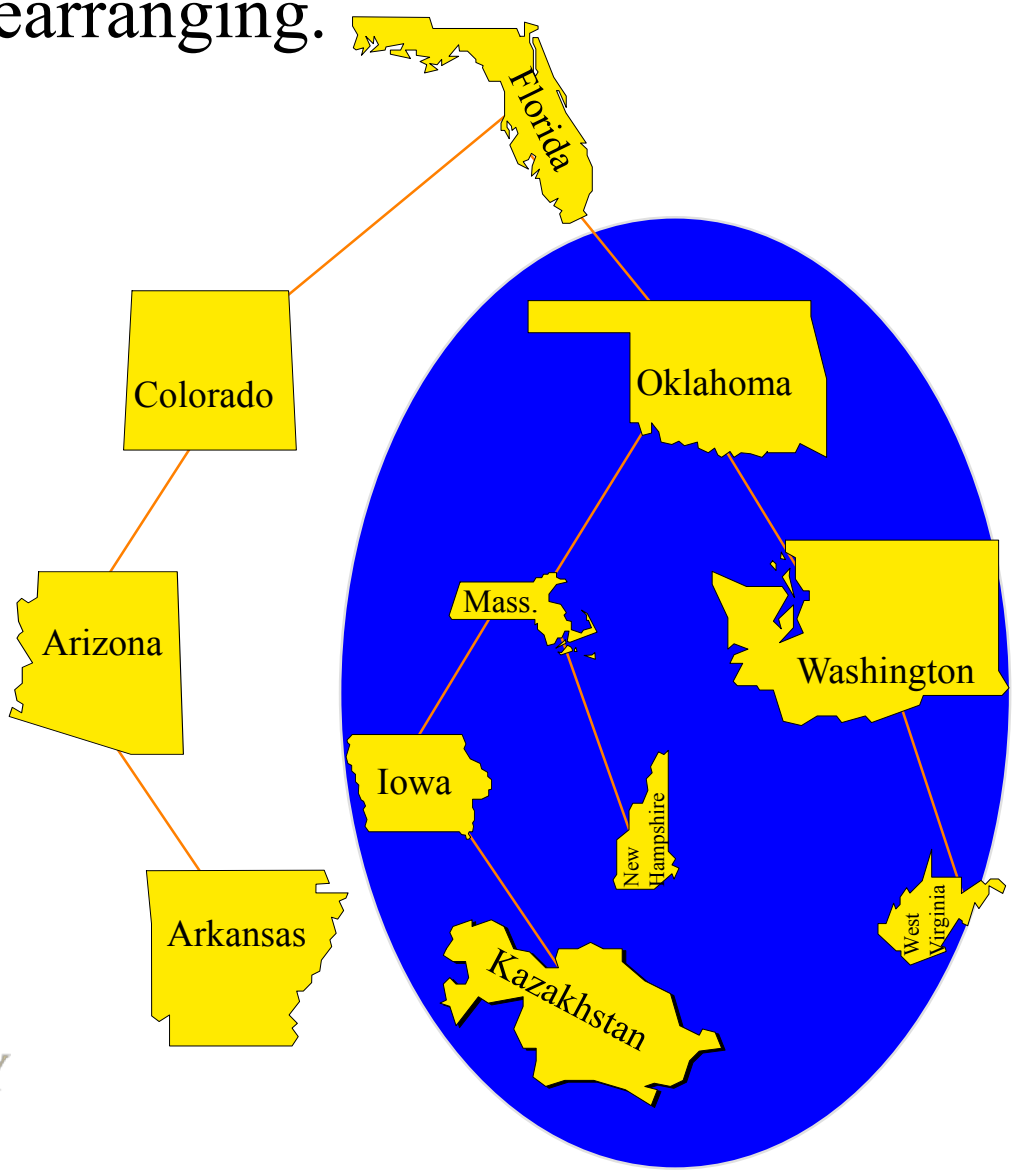
- If necessary, do some rearranging.



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

Removing "Florida"

- If necessary, do some rearranging.

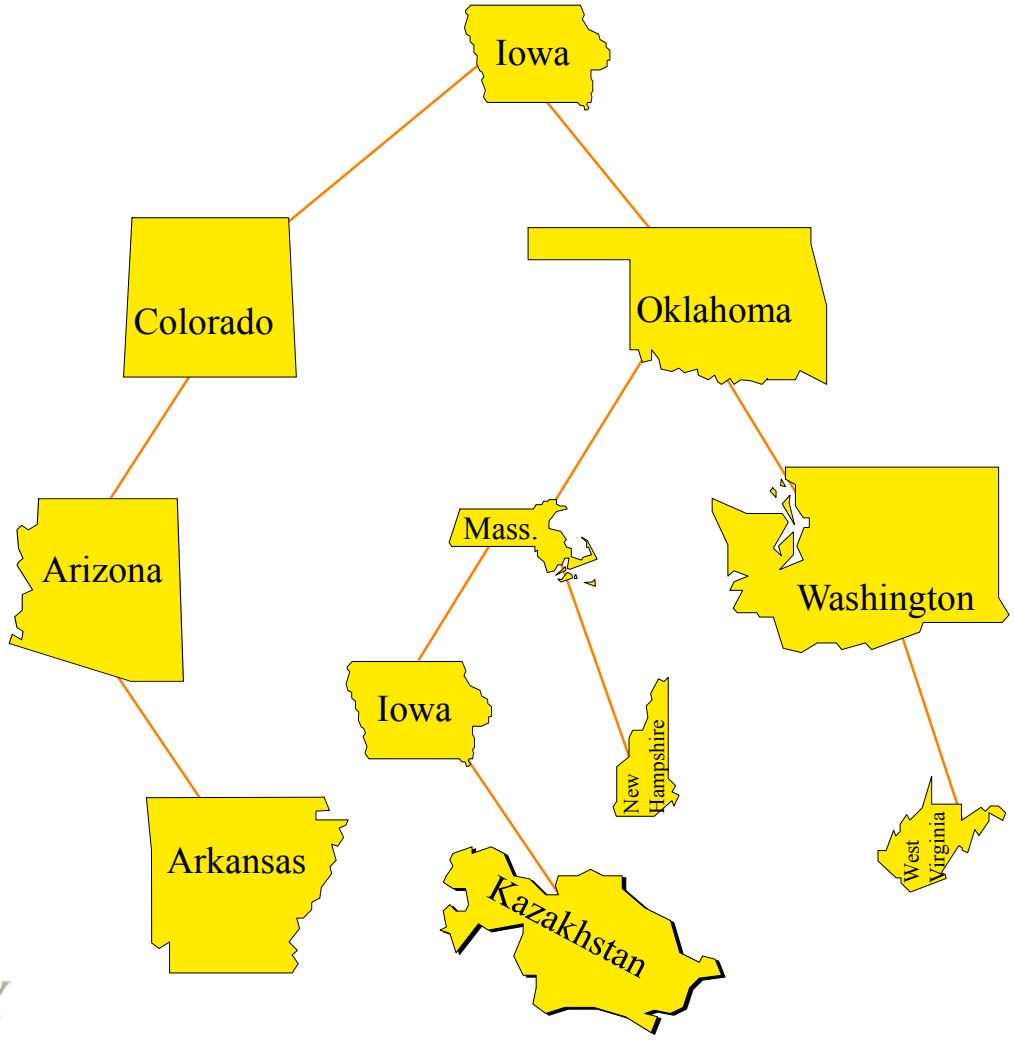


For the rearranging, take the **smallest** item in the right subtree...

Removing "Florida"

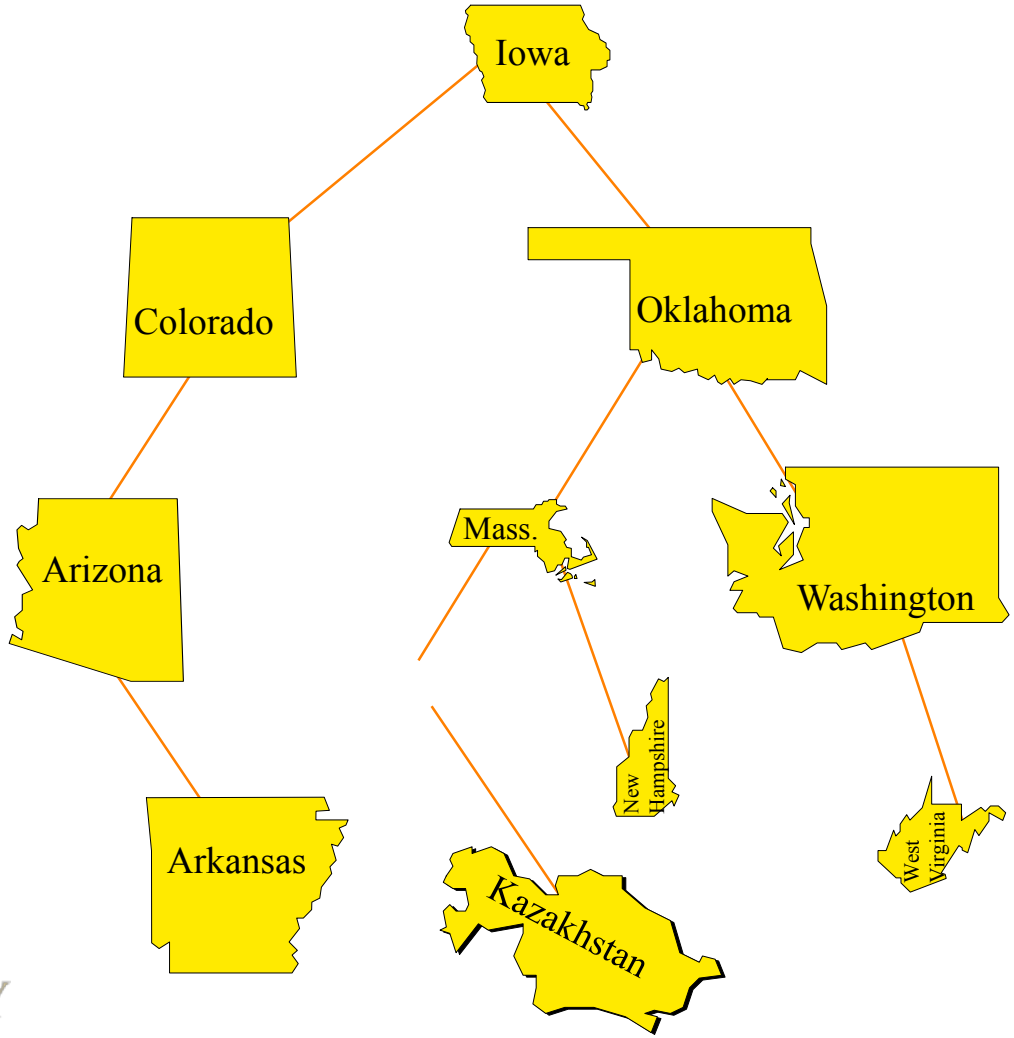
- If necessary, do some rearranging.

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



Removing "Florida"

- If necessary, do some rearranging.

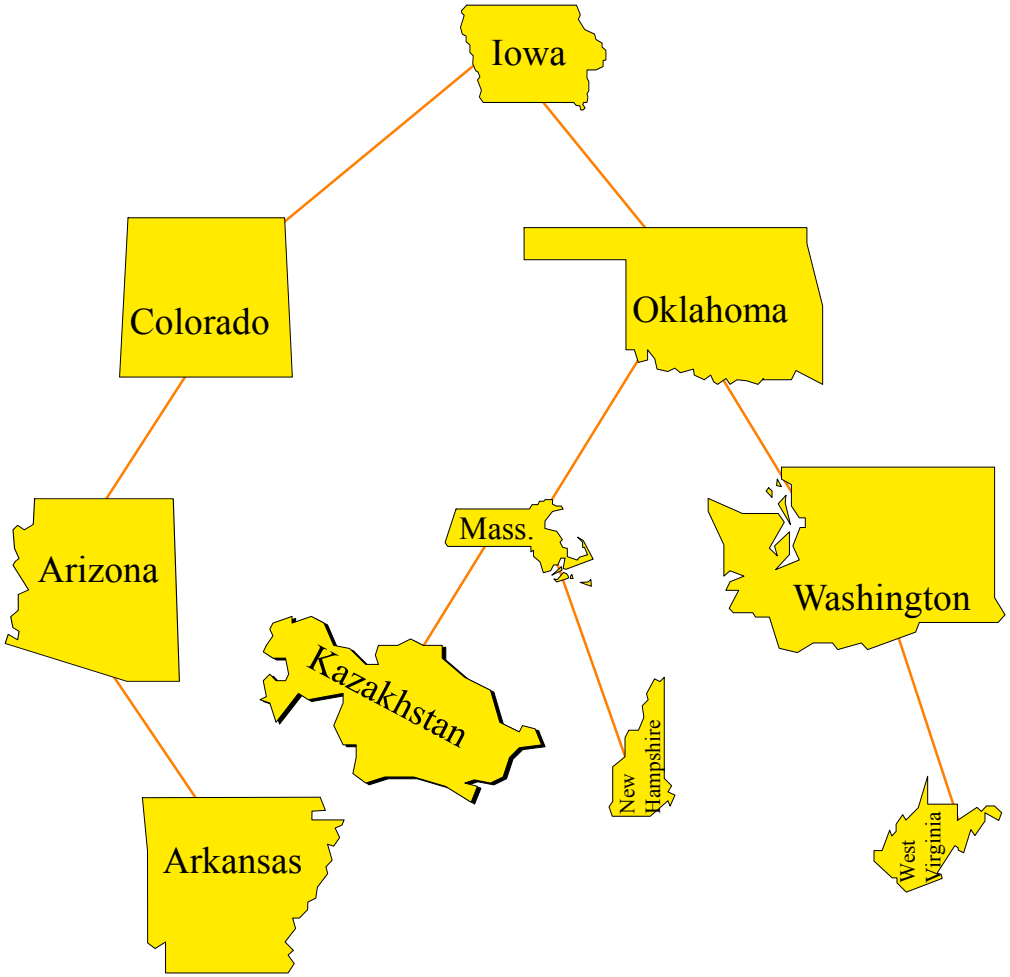


... and then remove the extra copy of the item we copied...

Removing "Florida"

- If necessary, do some rearranging.

... and reconnect the tree



Removing 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.