DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

BS (AI)

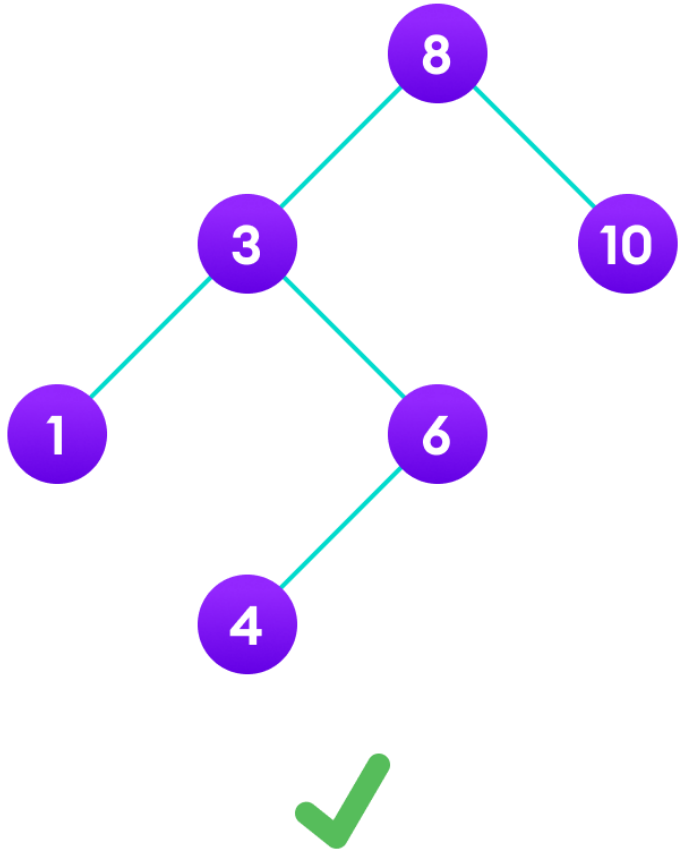
Binary Search Tree

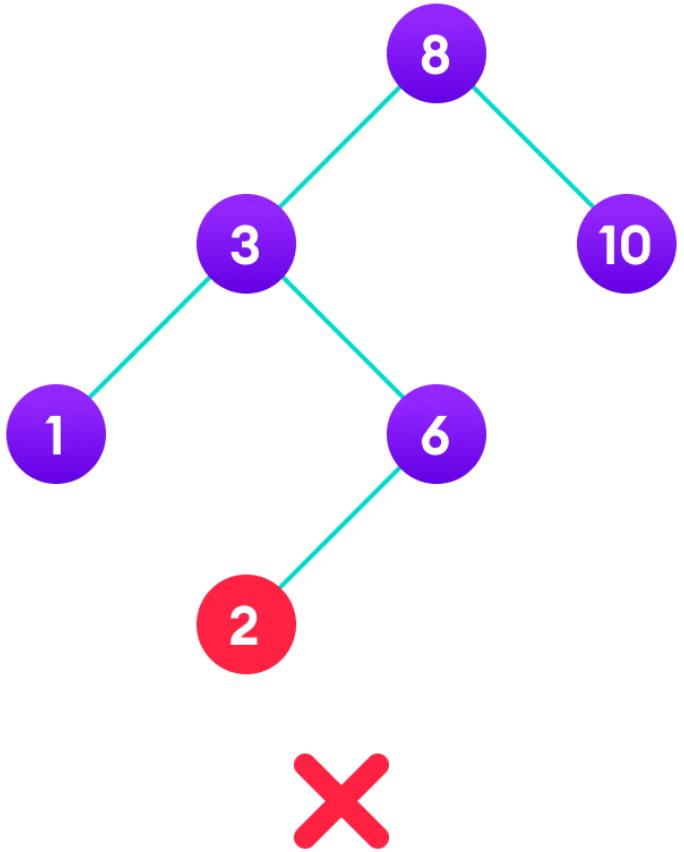
Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in $O(\log(n))$ time.

The properties that separate a binary search tree from a regular binary tree is

- All nodes of left subtree are less than the root node
- All nodes of right subtree are more than the root node
- Both subtrees of each node are also BSTs i.e. they have the above two properties





The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

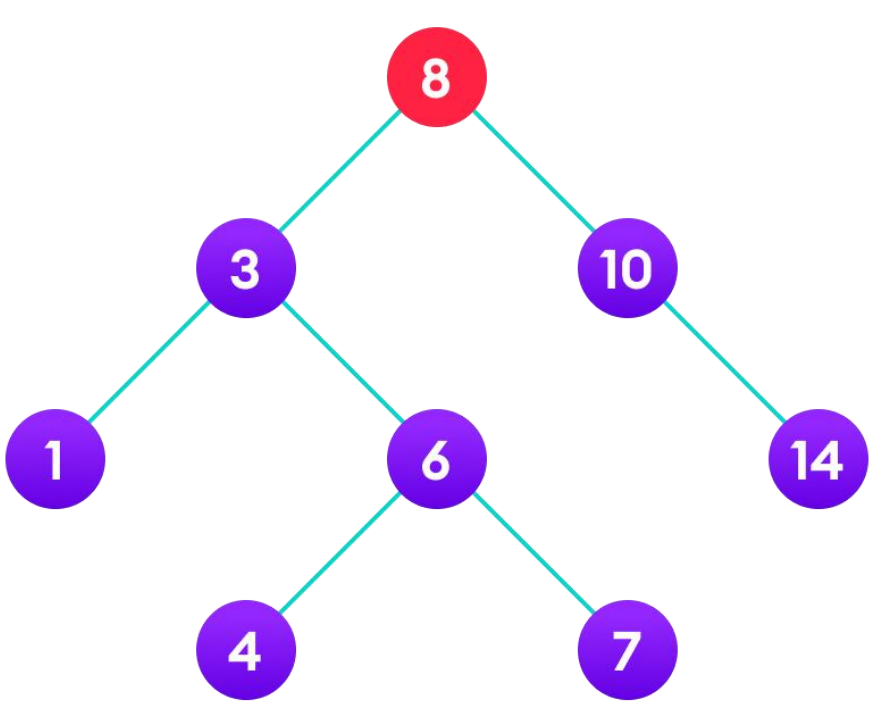
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

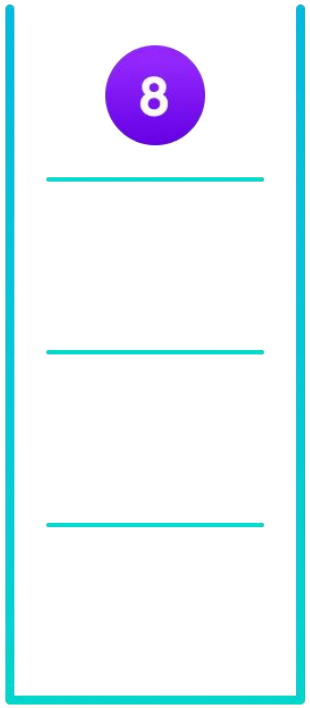
Search Operation

Algorithm

```
If root == NULL
        return NULL;
If number == root->data
        return root->data;
If number < root->data
        return search(root->left)
If number > root->data
        return search(root->right)
```

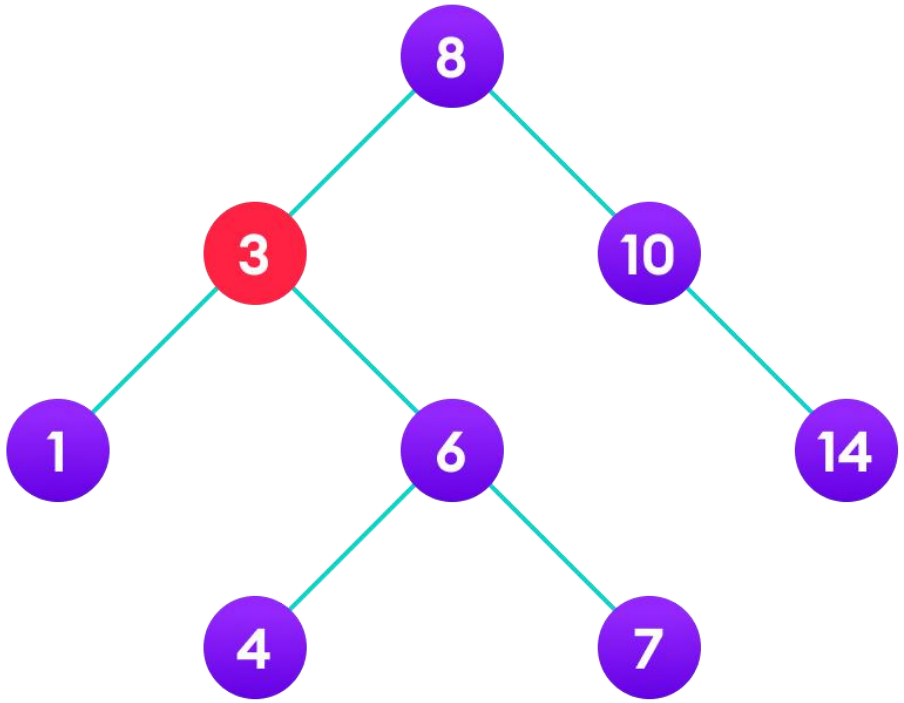
Search Operation

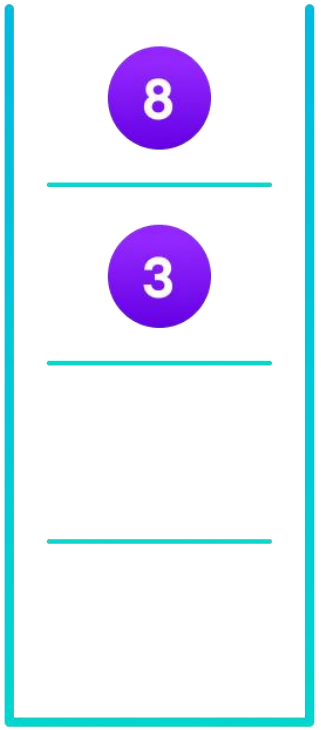




Search for 4

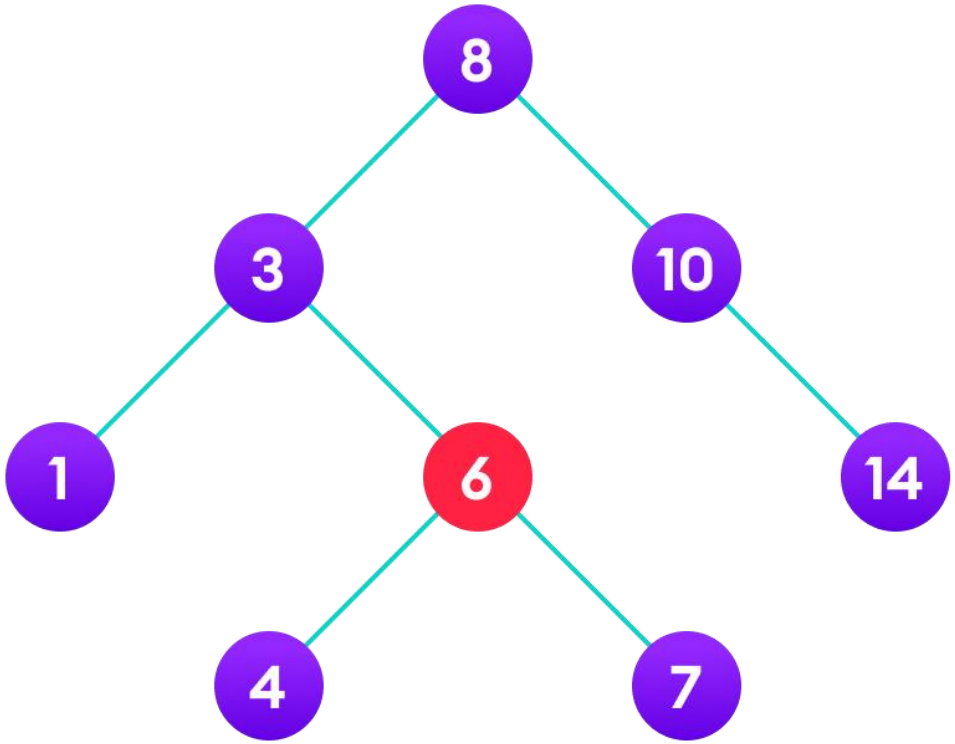
Search Operation

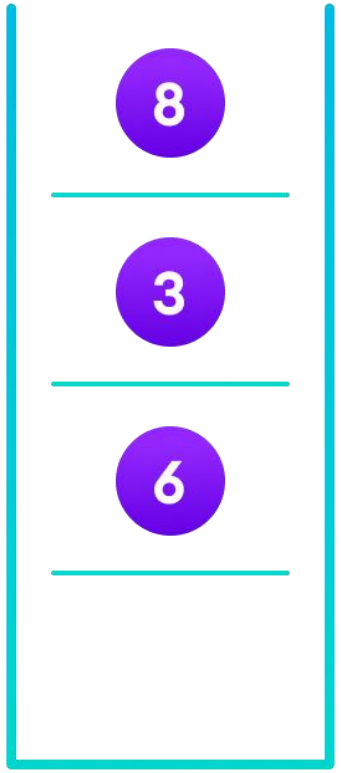




Search for 4

Search Operation





Search for 4

Insert Operation

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

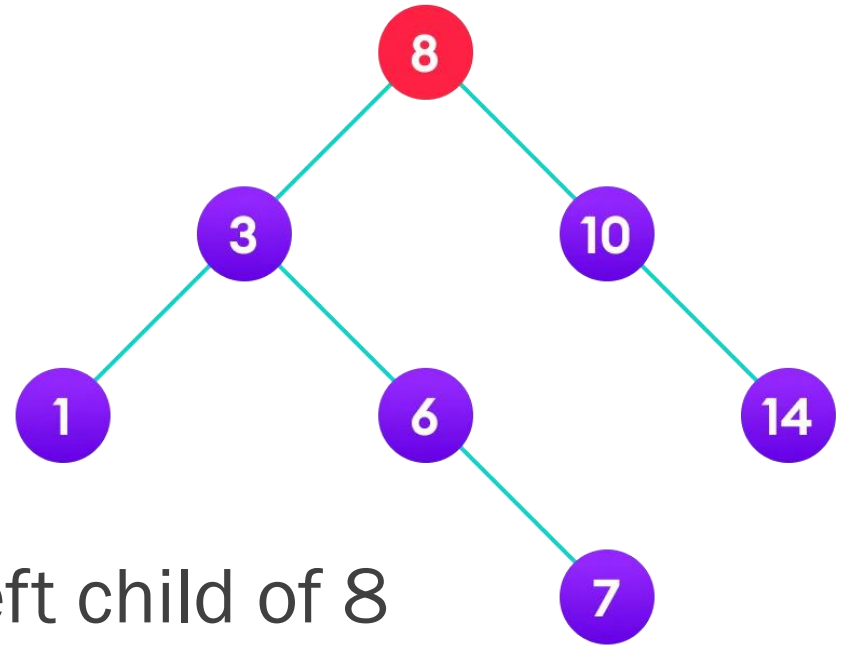
Insert Operation

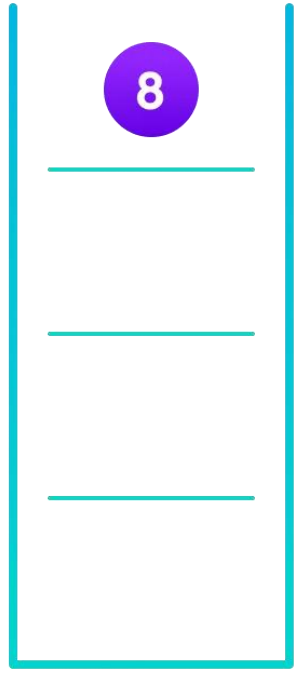
Algorithm

```
If node == NULL
        return createNode(data)
if (data < node->data)
        node->left = insert(node->left, data);
else if (data > node->data)
        node->right = insert(node->right, data);
return node;
```

Insert Operation: Visualization

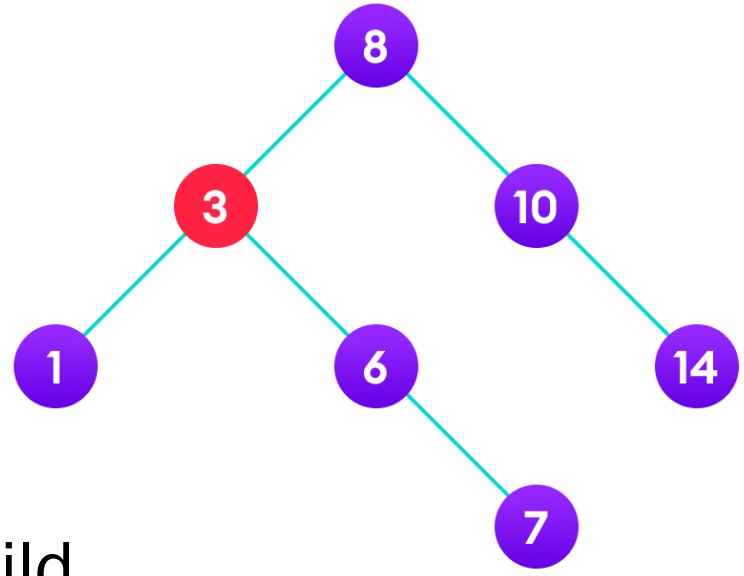
Insert 4

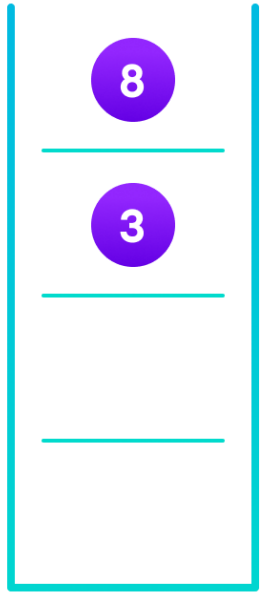




4<8 so, transverse through the left child of 8

Insert Operation: Visualization

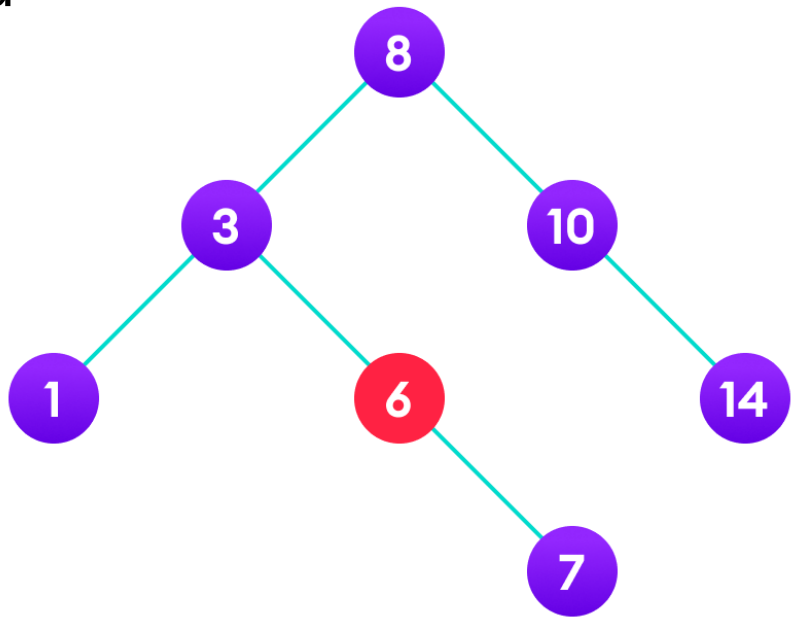


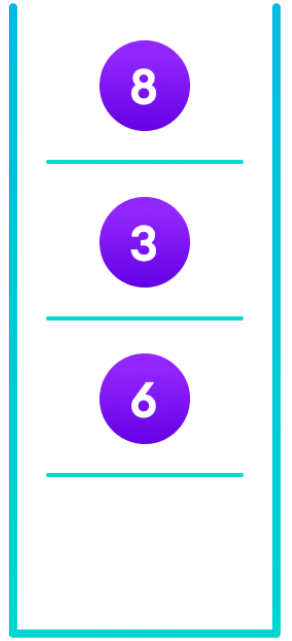


4>3 so, transverse through the right child of 8

Insert Operation: Visualization

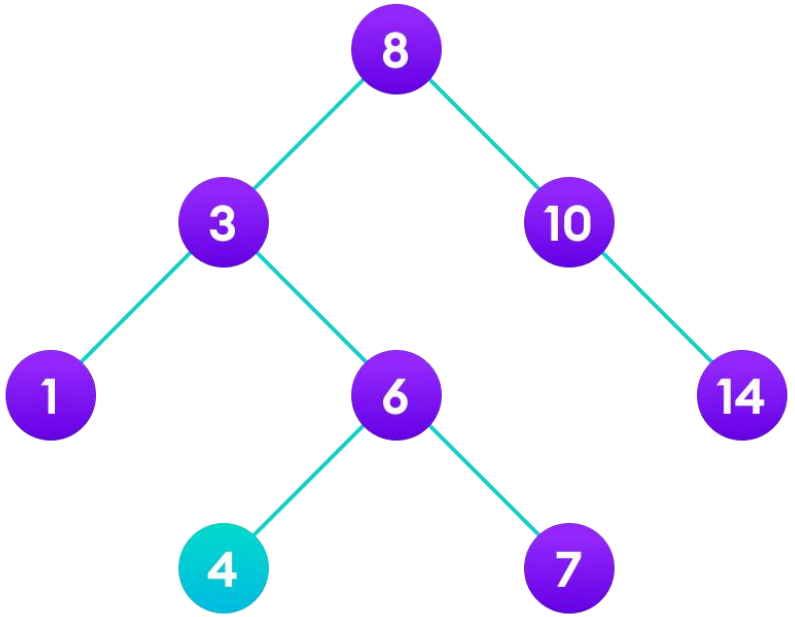
4<6 so, transverse through the left child of 6

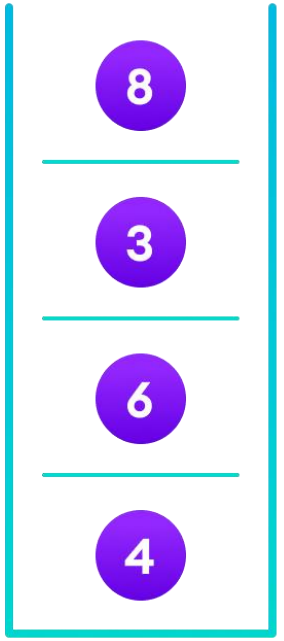




Insert Operation: Visualization

Insert 4 as a left child of 6



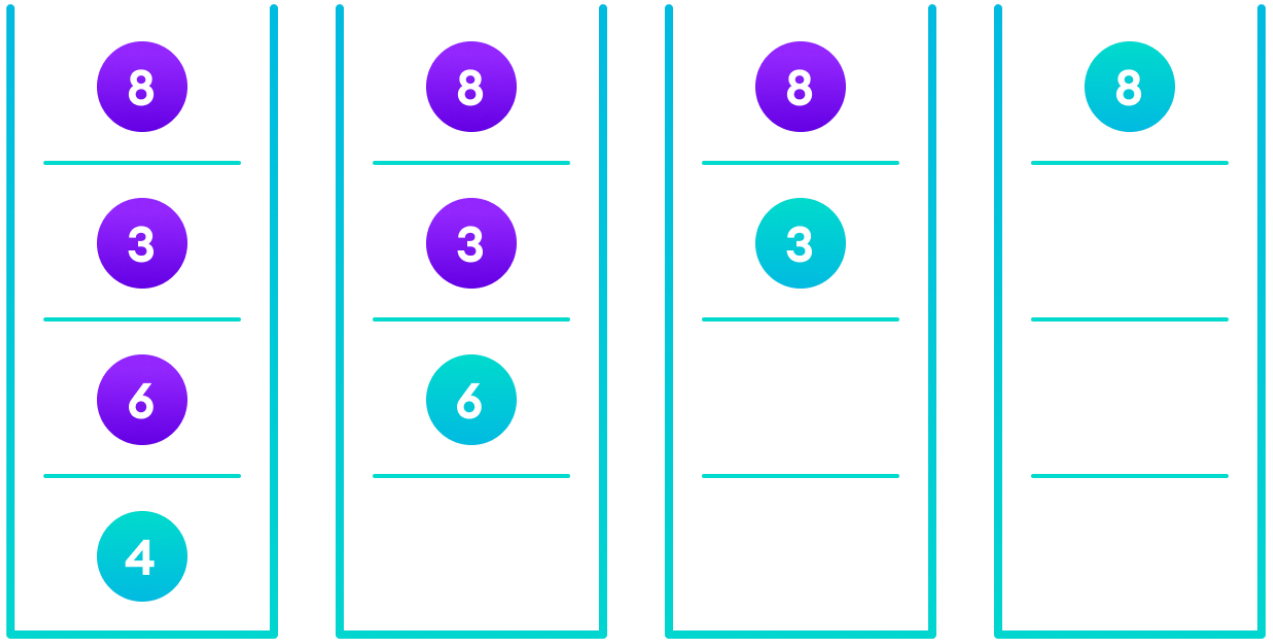


Insert Operation: Visualization

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree.
This is where the return node; at the end comes in handy. In the case of NULL, the newly created node is returned and attached to the parent node, otherwise the same node is returned
without any change as we go up until we return to the root.
This makes sure that as we move back up the tree, the other node connections aren't changed.

Insert Operation: Visualization

Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.



Deletion Operation

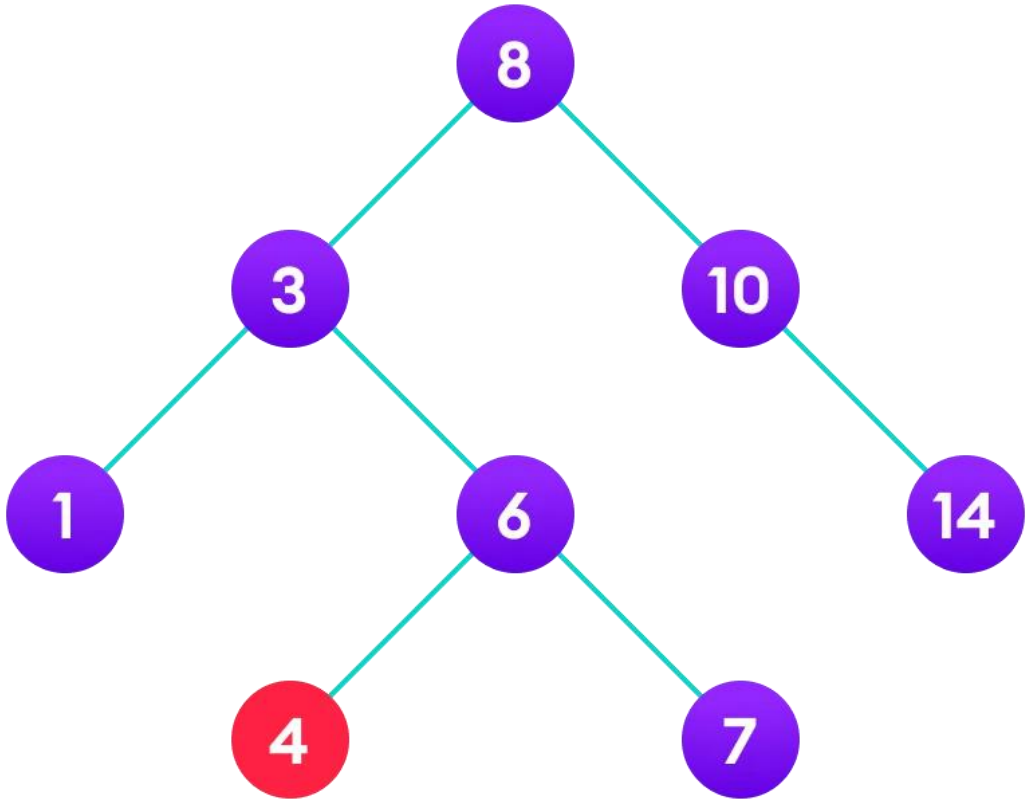
There are three cases for deleting a node from a binary search tree.

Case I: Delete the Leaf Node

Deletion Operation: Case 1

Case I

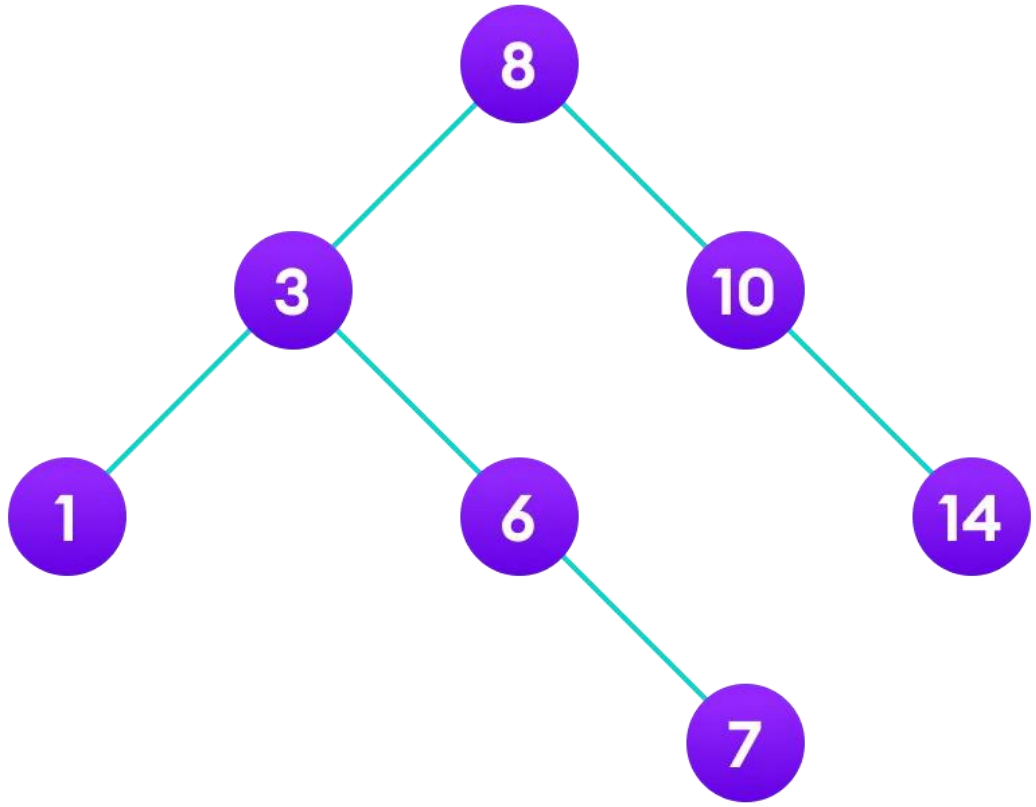
In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



4 is to be deleted

Deletion Operation: Case 1

Delete the node



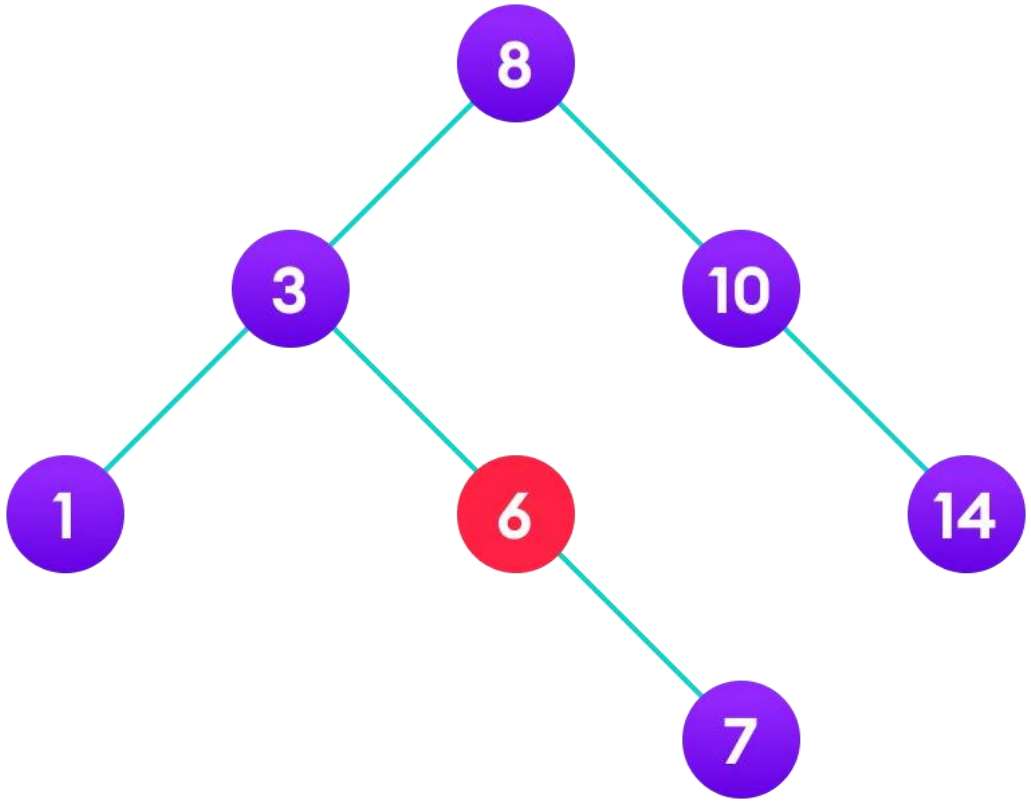
Deletion Operation: Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

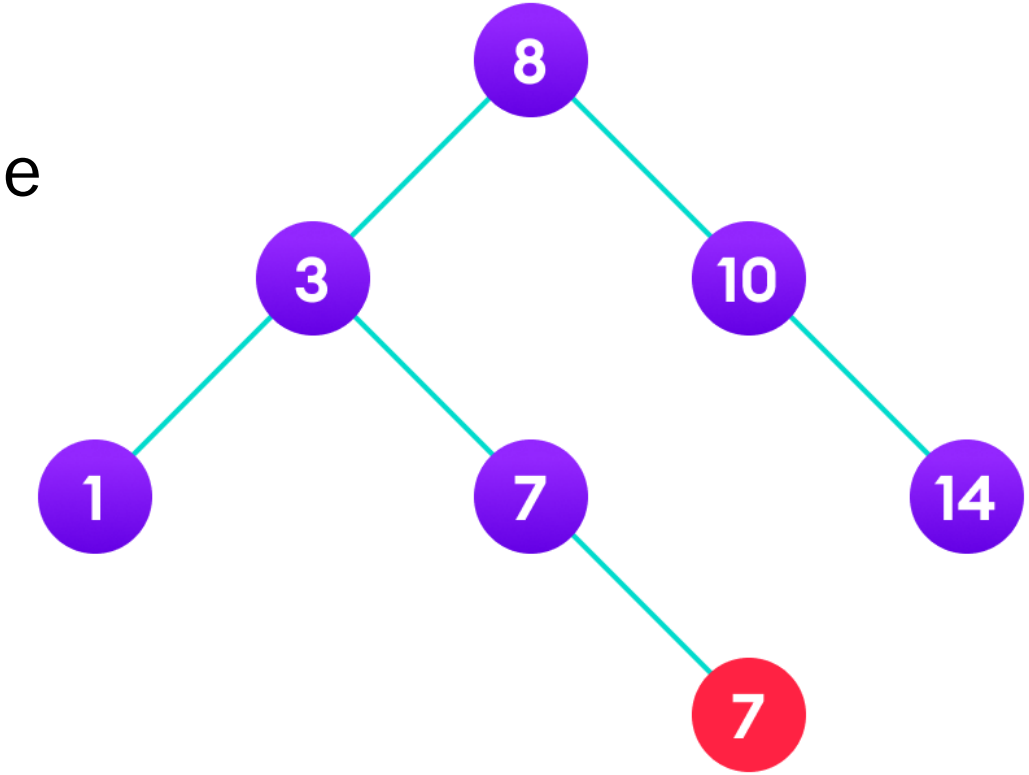
Deletion Operation: Case II

6 is to be deleted



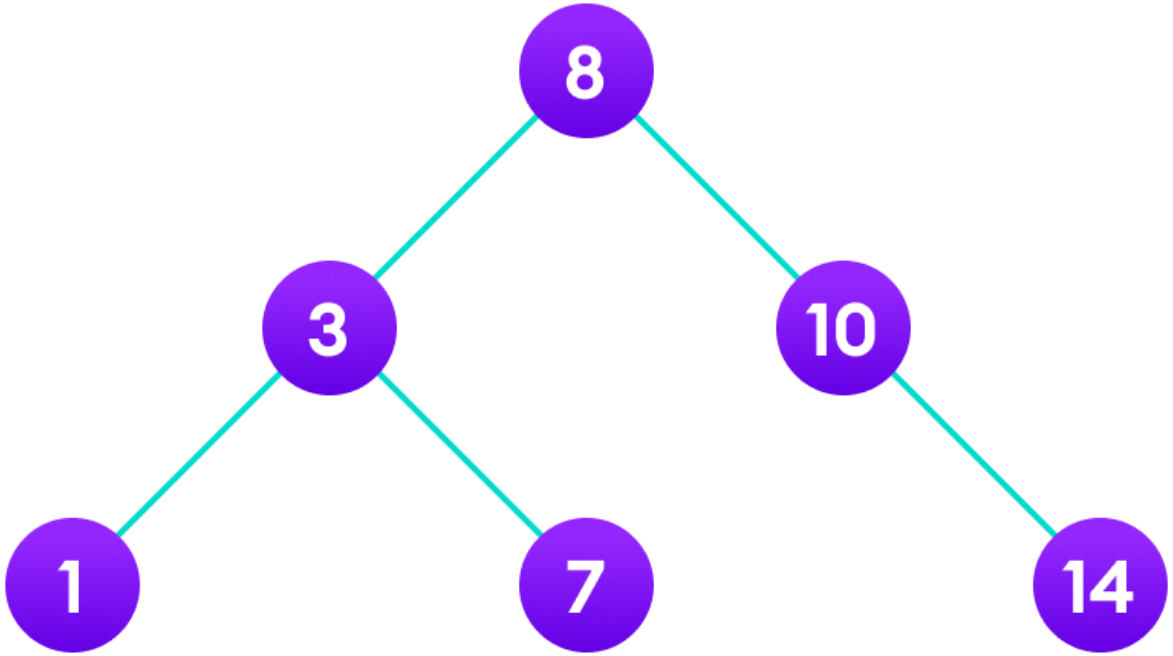
Deletion Operation: Case II

copy the value of its child to the node and delete the child



Deletion Operation: Case II

Final tree



Deletion Operation: Case III

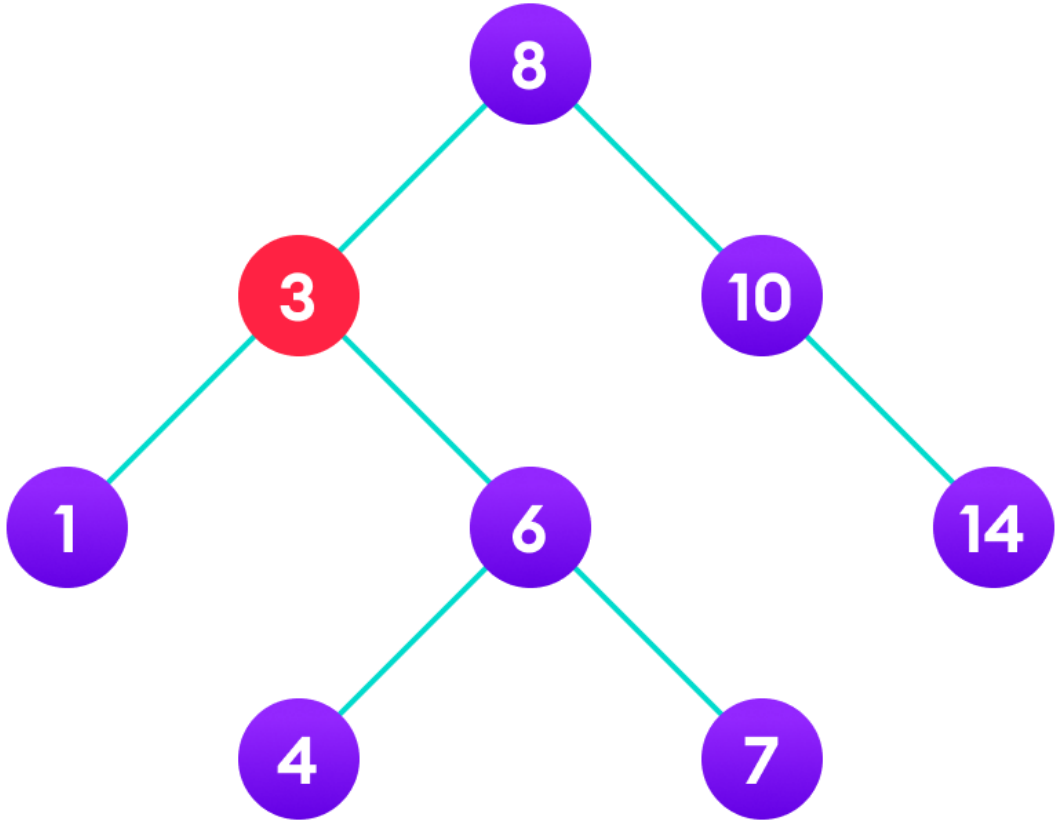
Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

Get the inorder successor of that node.
Replace the node with the inorder successor.
Remove the inorder successor from its original position.

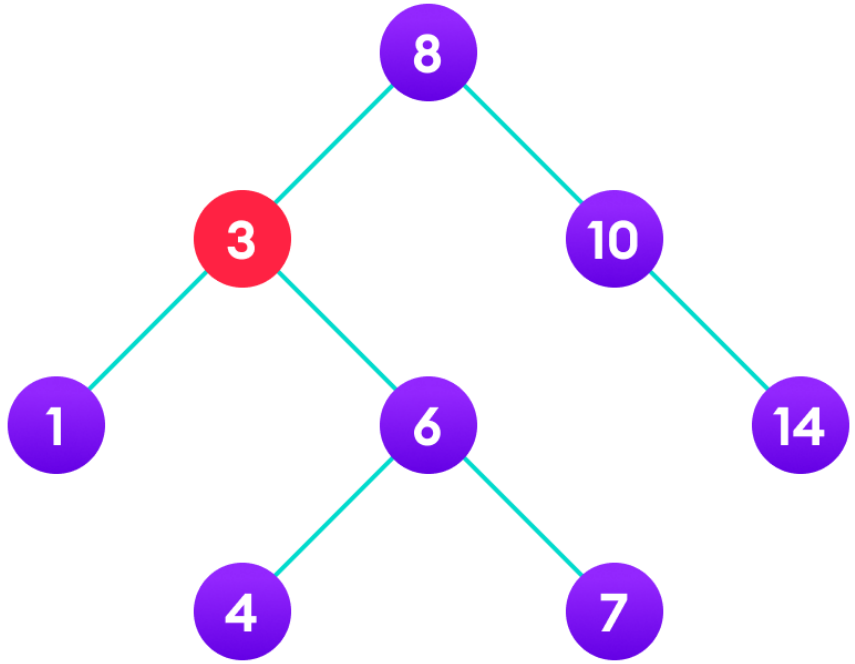
Deletion Operation: Case III

3 is to be deleted



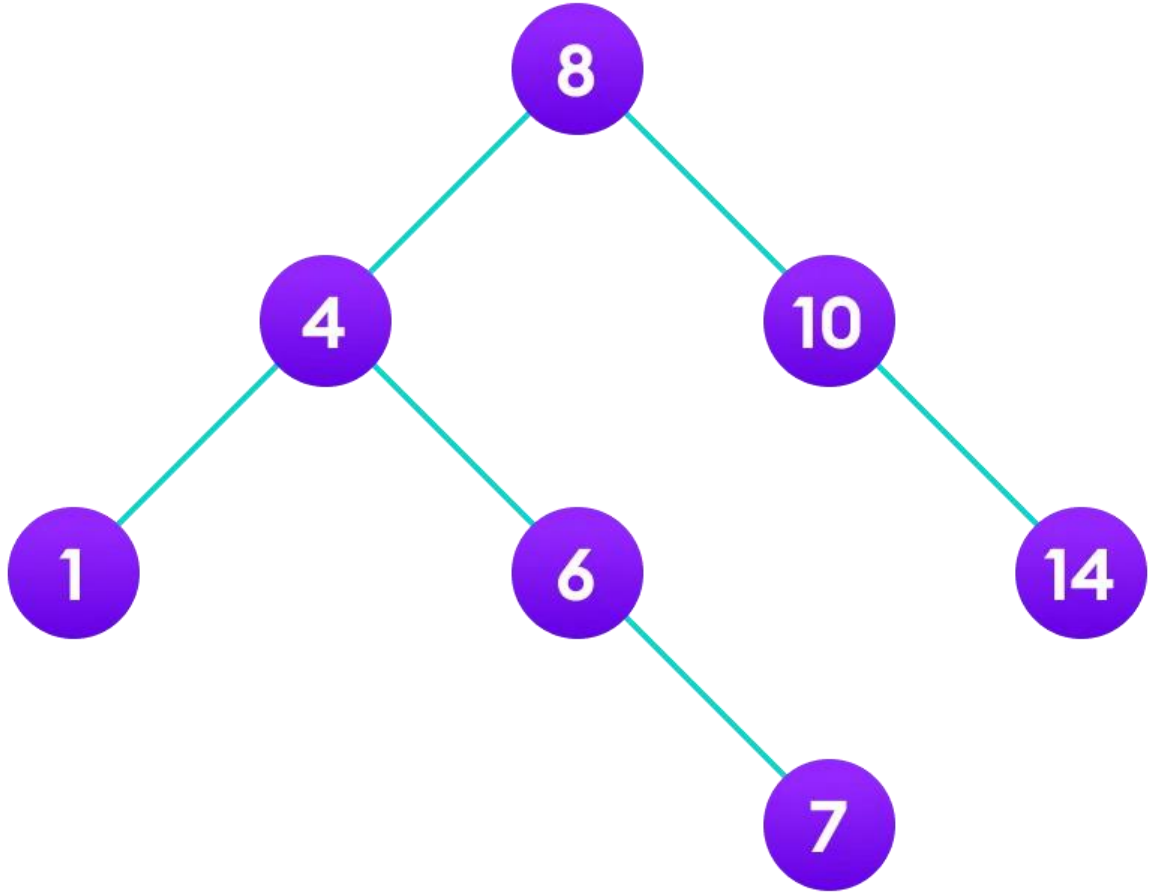
Deletion Operation: Case III

Copy the value of the inorder successor (4) to the node



Deletion Operation: Case III

Delete the inorder successor



Binary Search Tree: Code

Binary Search Tree: Code

```
struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
struct node *temp = new node;
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}
```

Binary Search Tree: Code

```
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);                                    // Inorder Traversal

    // Traverse root
    cout << root->key << " -> ";

    // Traverse right
    inorder(root->right);
  }
}
```

Binary Search Tree: Code

```
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL)
      return newNode(key);
  // Traverse to the right place and insert the node
  if (key < node->key)
      node->left = insert(node->left, key);
  else
      node->right = insert(node->right, key);
  return node;
}
```

Insert a node

Binary Search Tree: Code

```
// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}
```

Binary Search Tree: Code

```
// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
```

Binary Search Tree: Code

```
else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }
```

Binary Search Tree: Code

```
// If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}
```

Binary Search Tree: Code

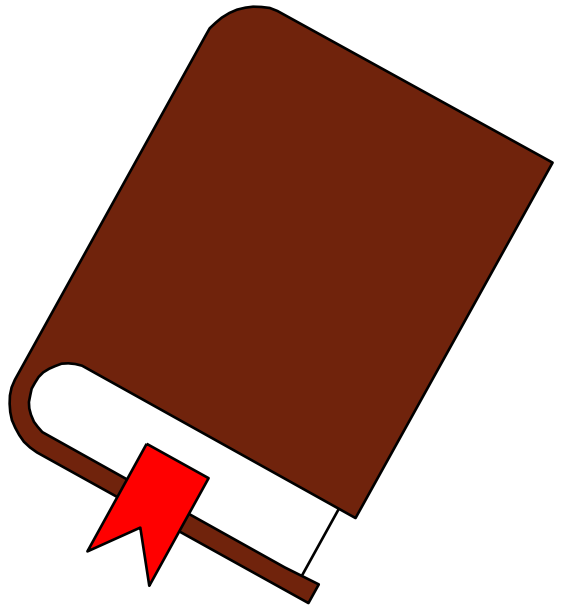
```
// Driver code

int main() {

  struct node *root = NULL;

  root = insert(root, 8);

  root = insert(root, 3);

  root = insert(root, 1);

  root = insert(root, 6);

  root = insert(root, 7);

  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);

  cout << "Inorder traversal: ";
  inorder(root);

  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
}
```

Application of BST

The Dictionary Data Type

A dictionary is a collection of items,

Each item has a string attached to it, called the item's key.



The Dictionary Data Type

void dictionary::insert(The key for the new item, The new item);

The insertion procedure for a dictionary has two parameters.

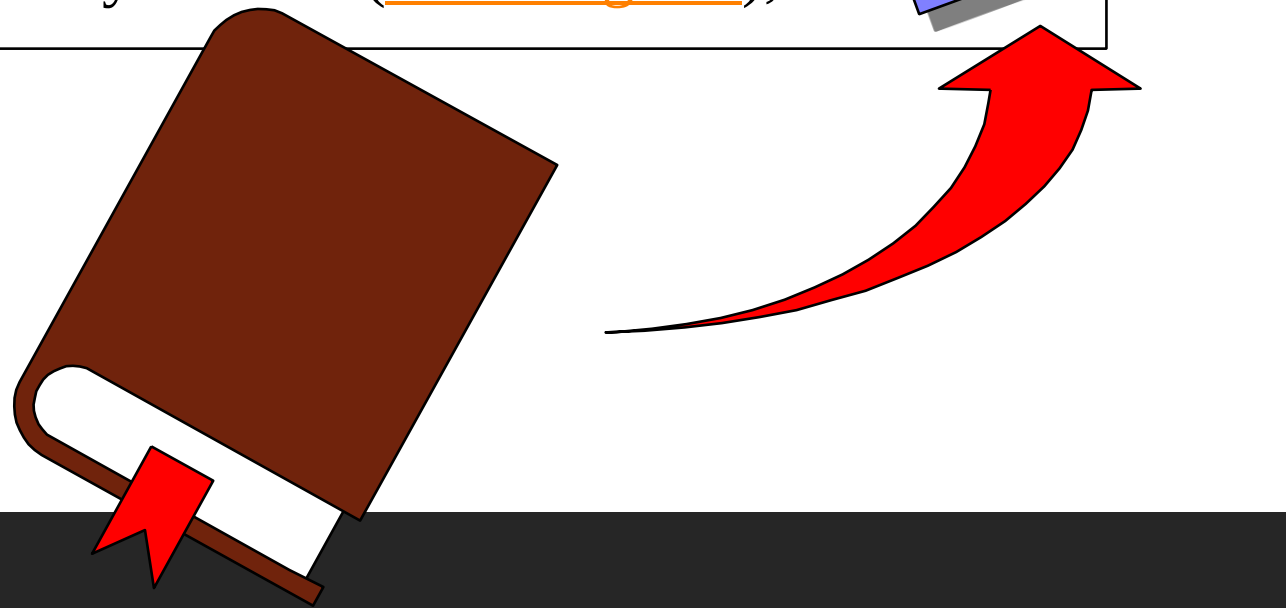
The Dictionary Data Type

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

The Dictionary Data Type

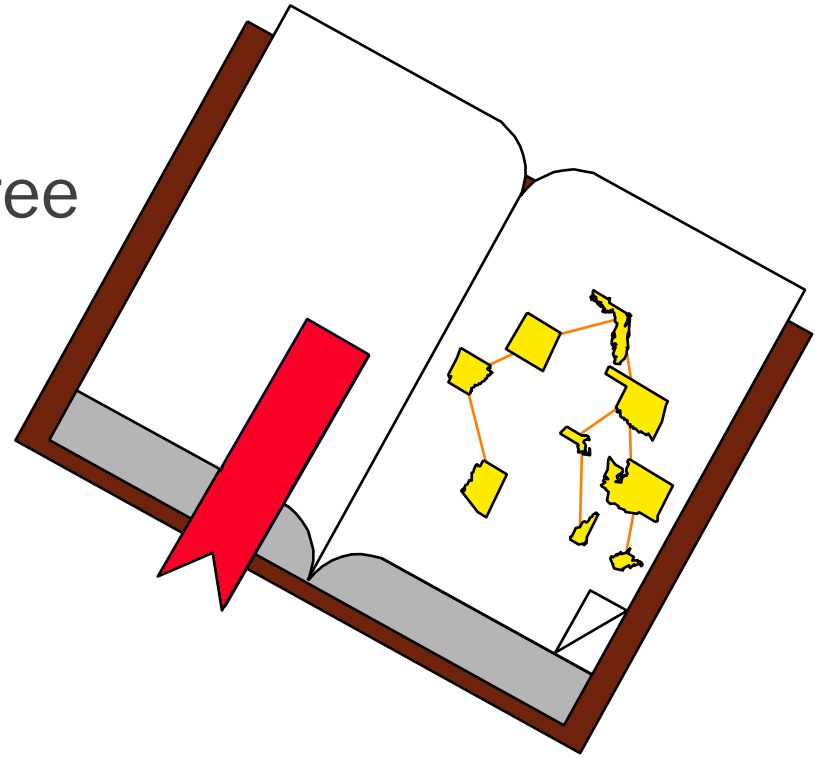
- When you want to retrieve an item, you specify the **key**...
  ... and the retrieval procedure returns the **item**.

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



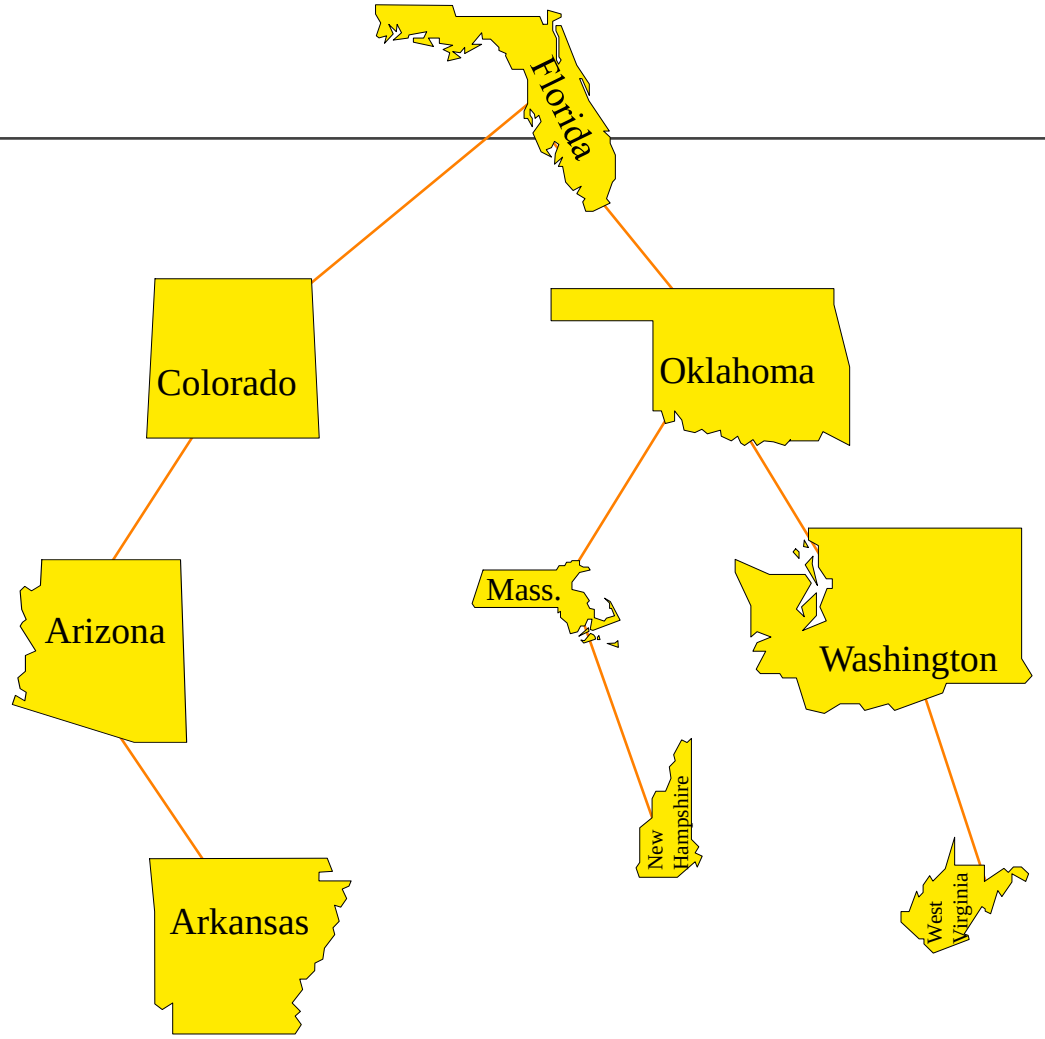
The Dictionary Data Type

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

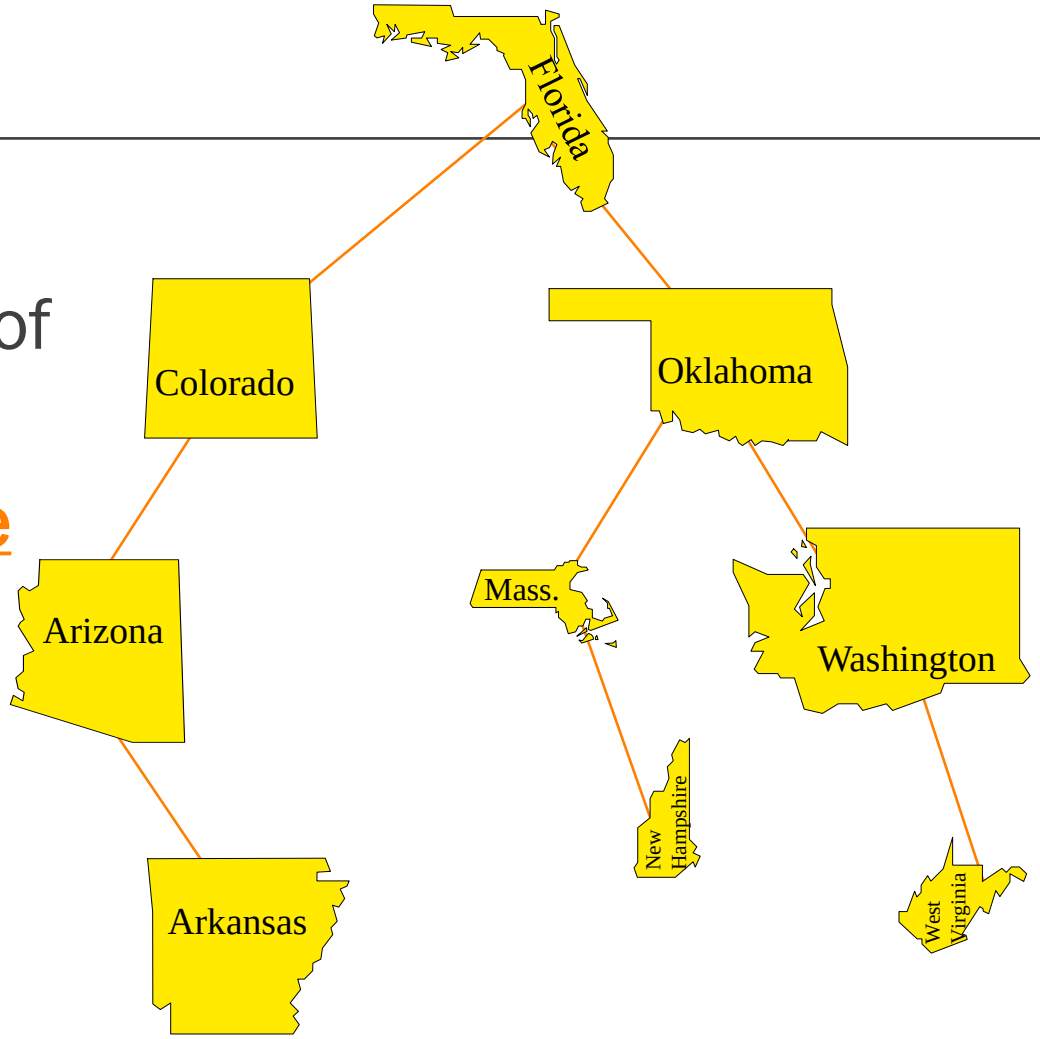
The data in the dictionary will be stored in a binary tree, with each node containing an item and a key.



A Binary Search Tree of States

Storage rules:

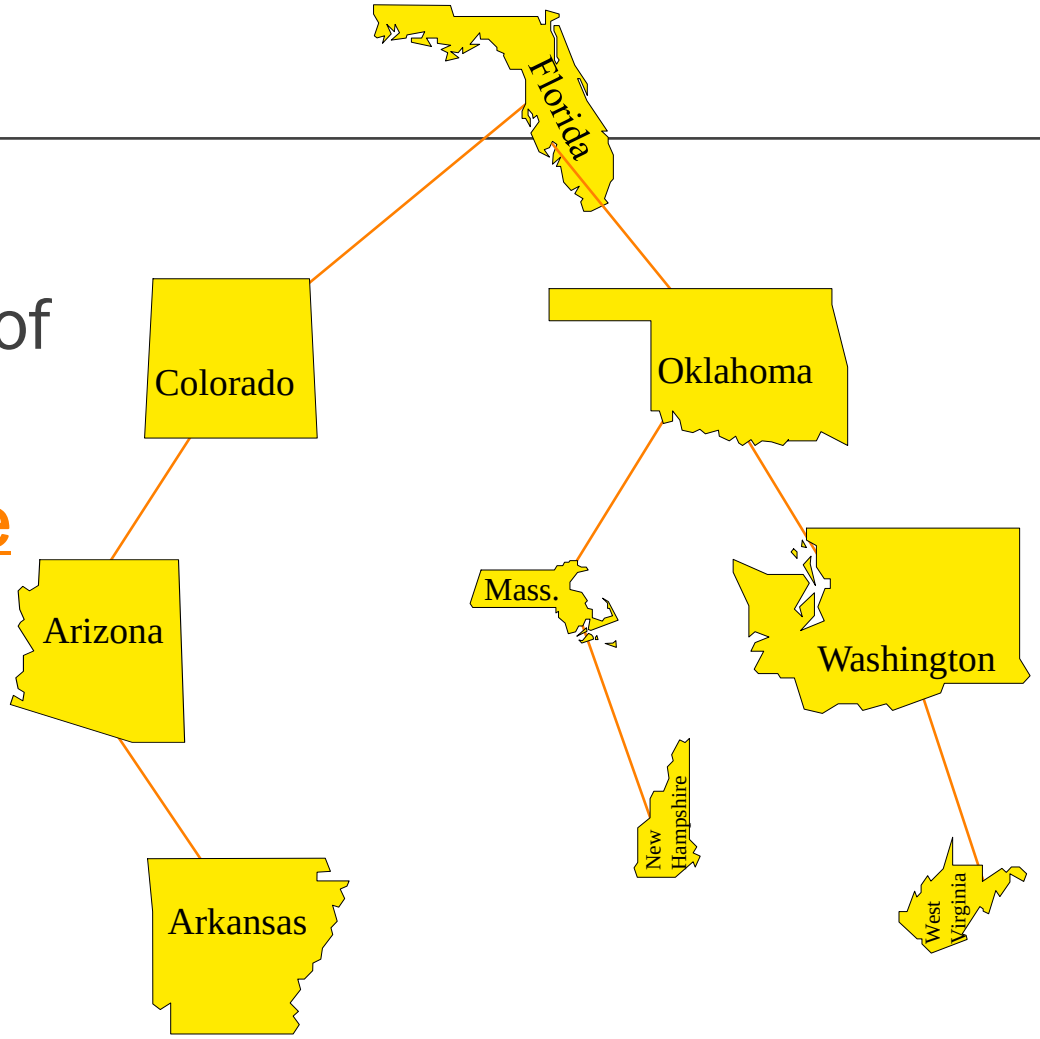
- Every key to the left of a node is alphabetically before the key of the node.



A Binary Search Tree of States

Storage rules:

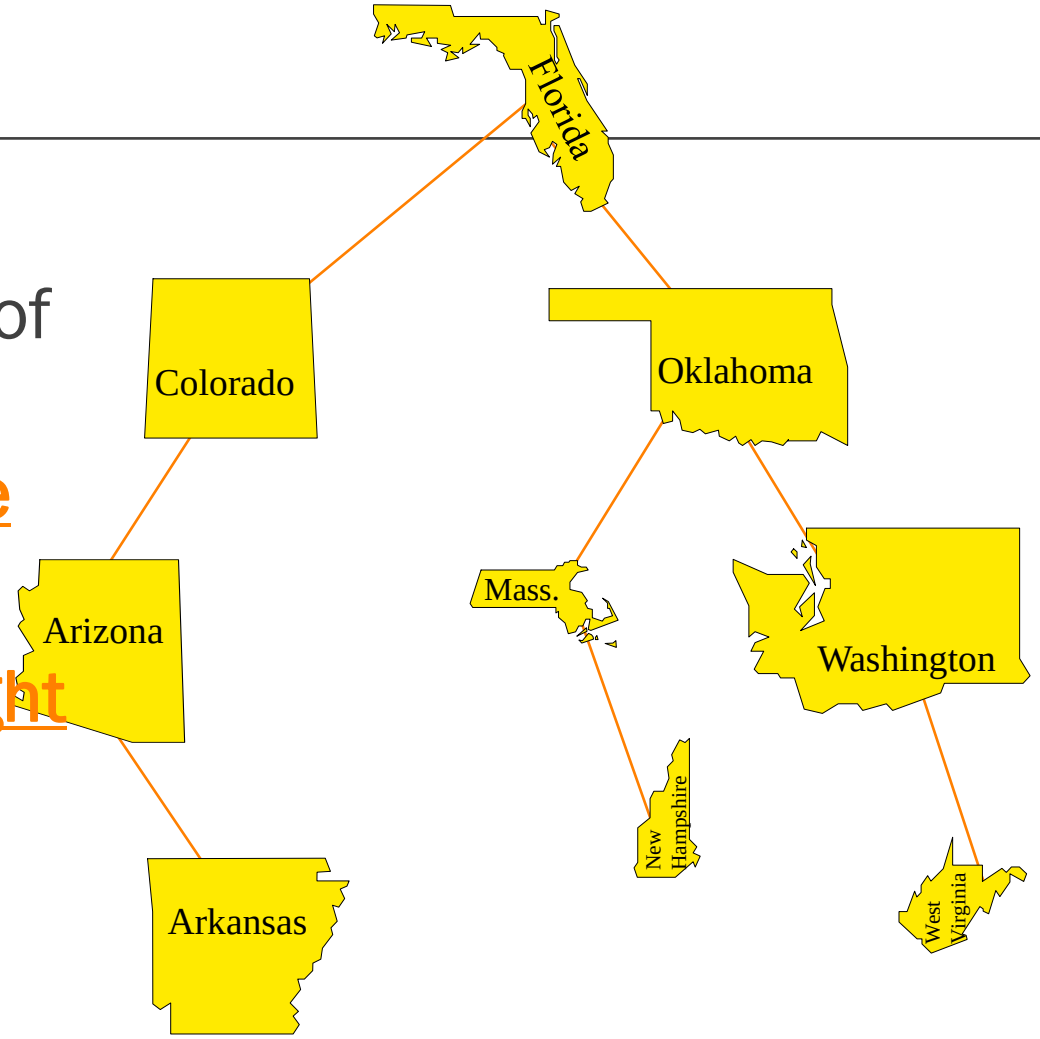
- Every key to the left of a node is alphabetically before the key of the node.



A Binary Search Tree of States

Storage rules:

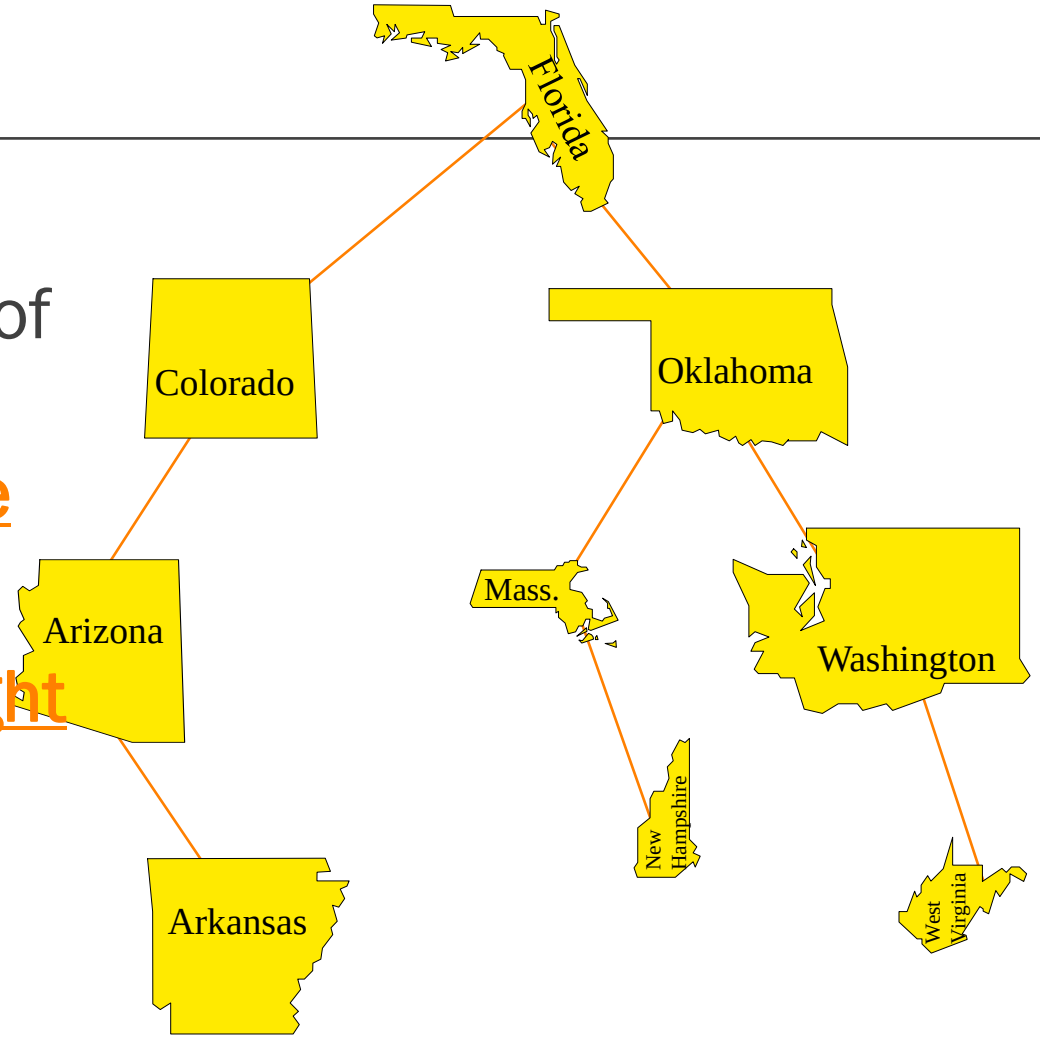
- Every key to the left of a node is alphabetically before the key of the node.
- Every key to the right of a node is alphabetically after the key of the node.



A Binary Search Tree of States

Storage rules:

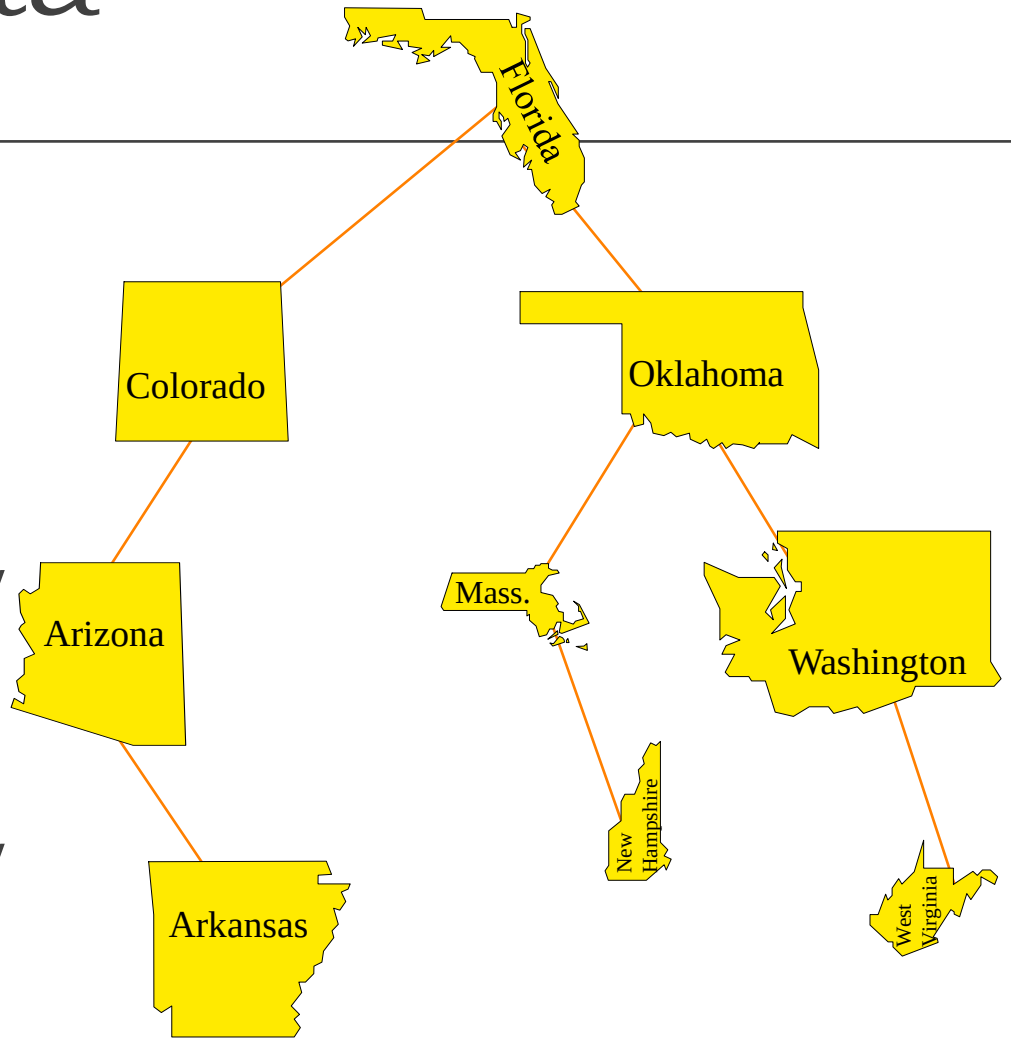
- Every key to the left of a node is alphabetically before the key of the node.
- Every key to the right of a node is alphabetically after the key of the node.



Retrieving Data

Start at the root.

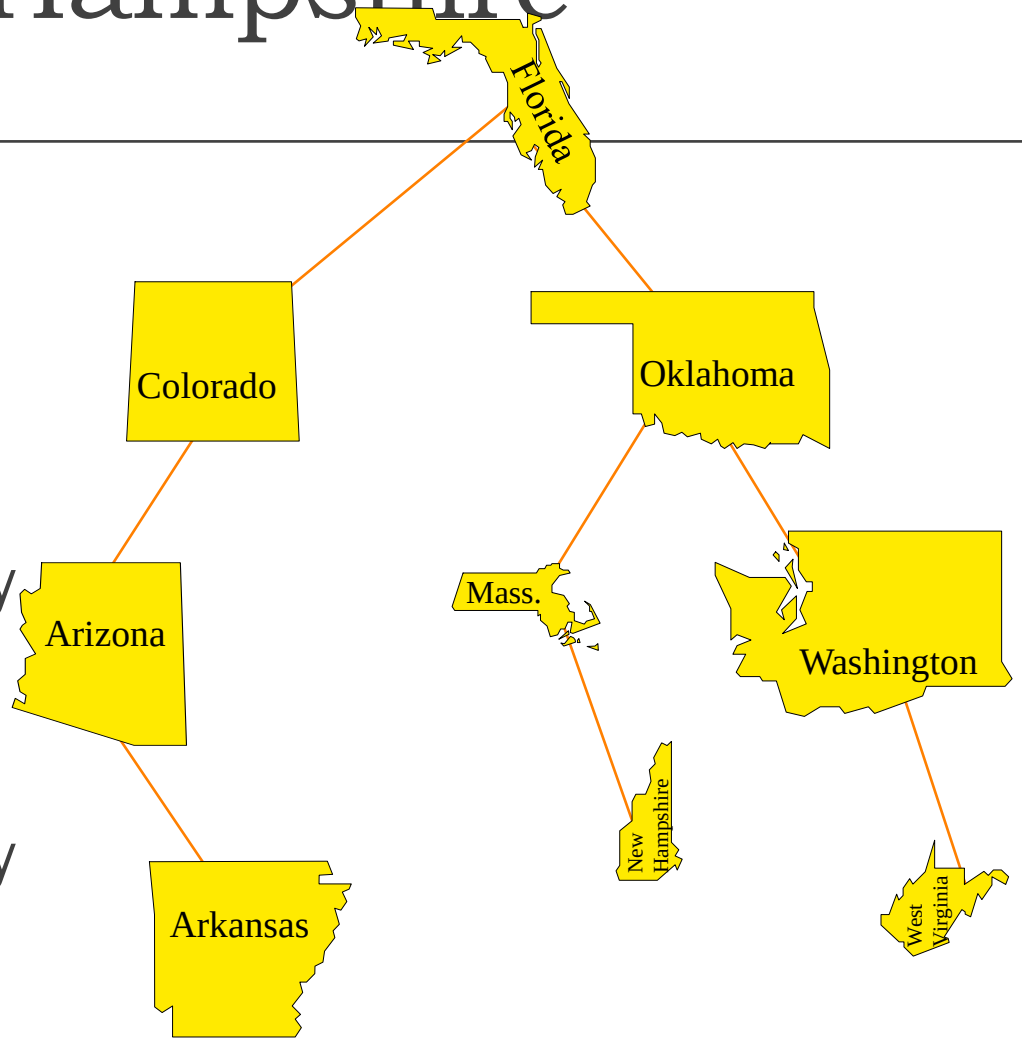
- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too large, move left and repeat 1-3.
- If the current node's key is too small, move right and repeat 1-3.



Retrieve " New Hampshire"

Start at the root.

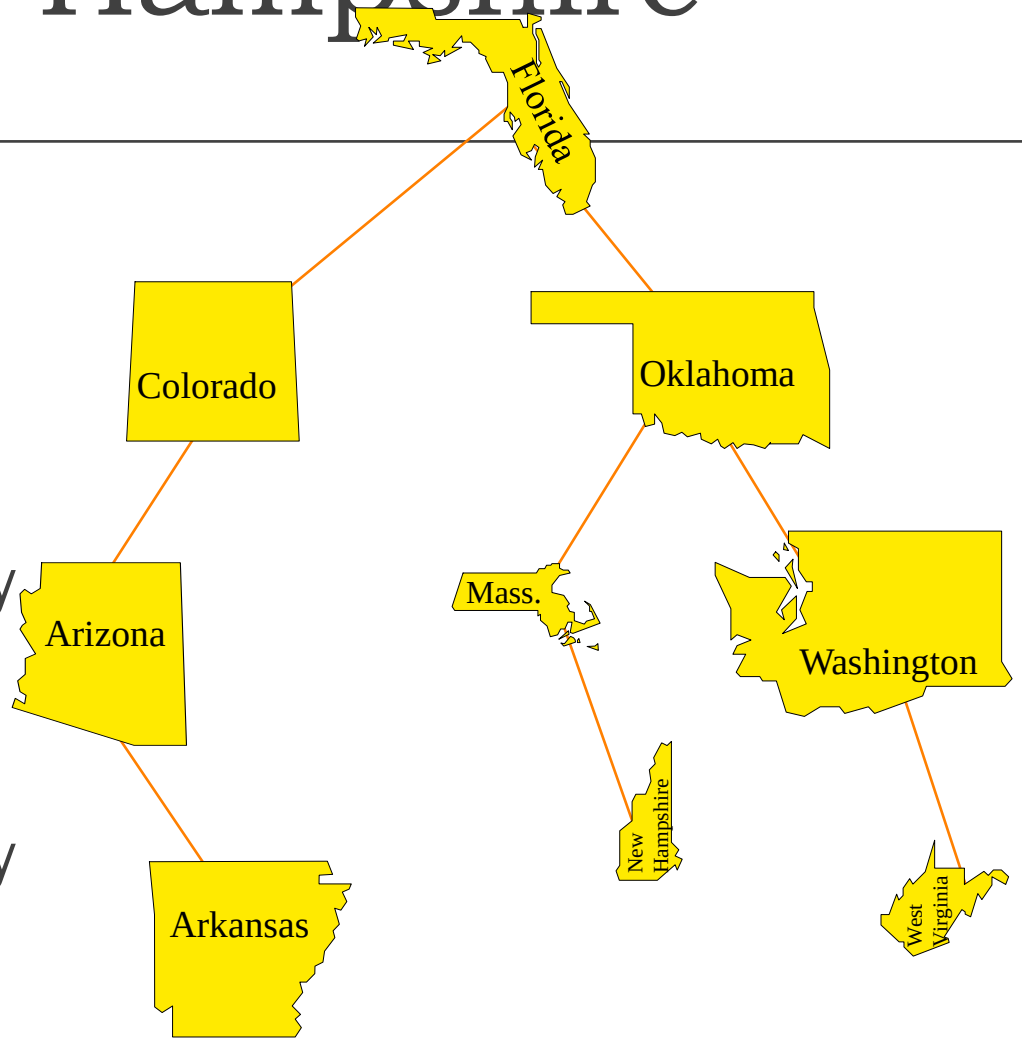
- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too large, move left and repeat 1-3.
- If the current node's key is too small, move right and repeat 1-3.



Retrieve "New Hampshire"

Start at the root.

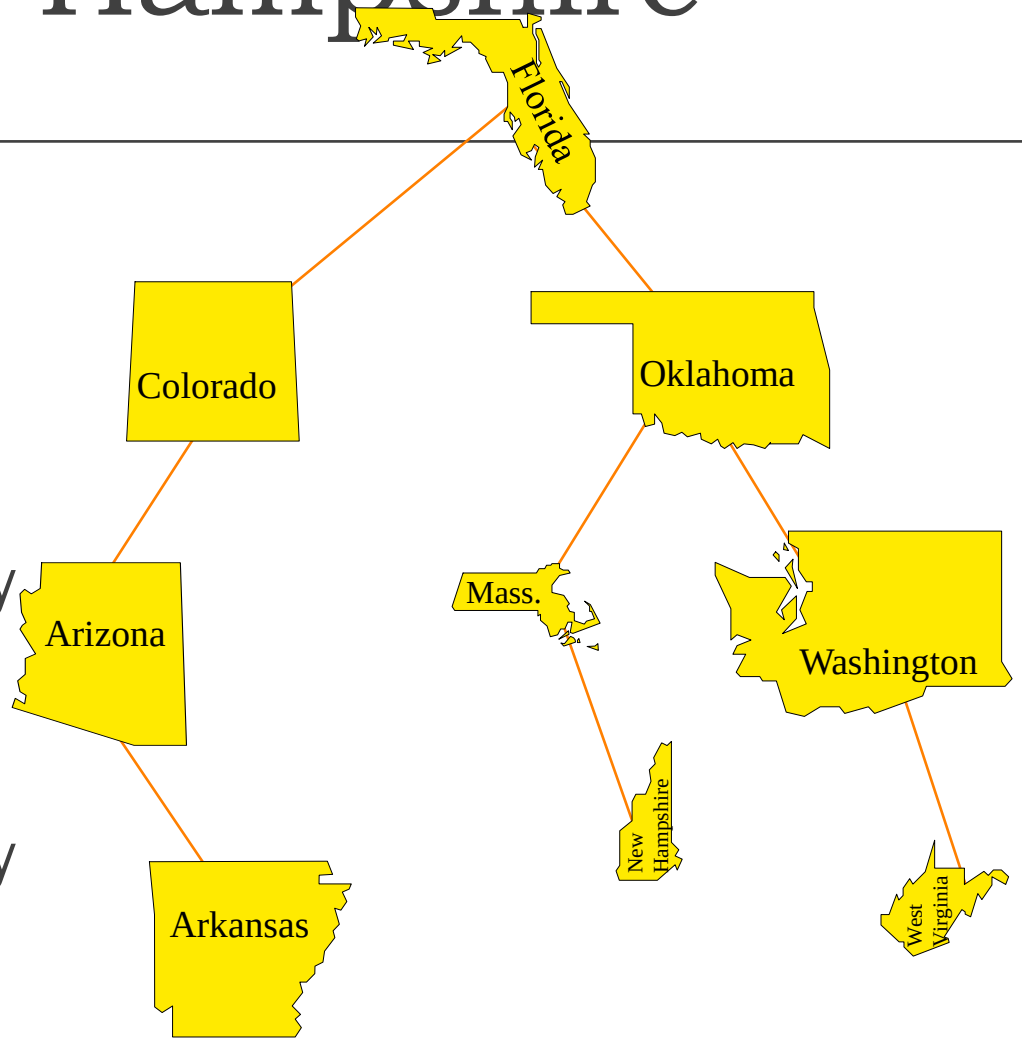
- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too large, move left and repeat 1-3.
- If the current node's key is too small, move right and repeat 1-3.



Retrieve "New Hampshire"

Start at the root.

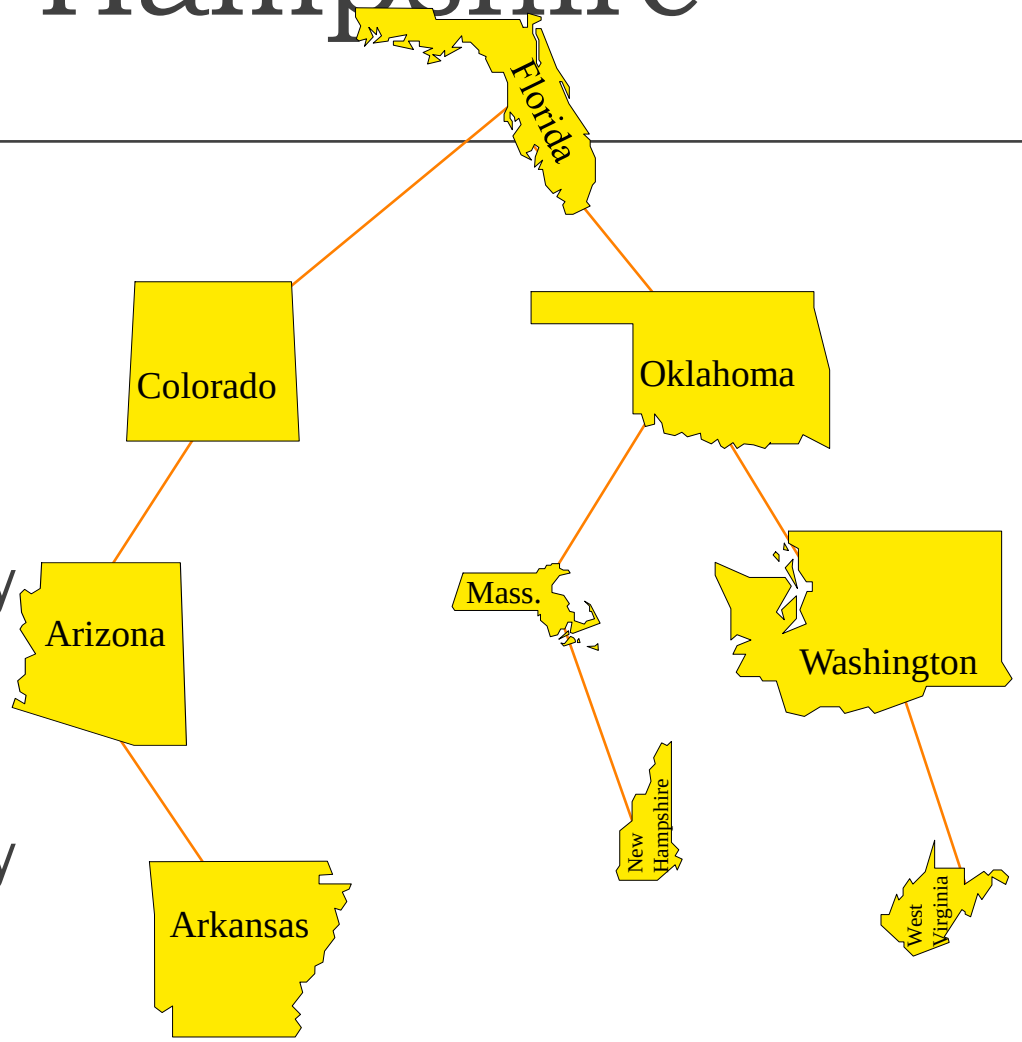
- If the current node has the key, then stop and retrieve the data.
- If the current node's key is too large, move left and repeat 1-3.
- If the current node's key is too small, move right and repeat 1-3.



Retrieve "New Hampshire"

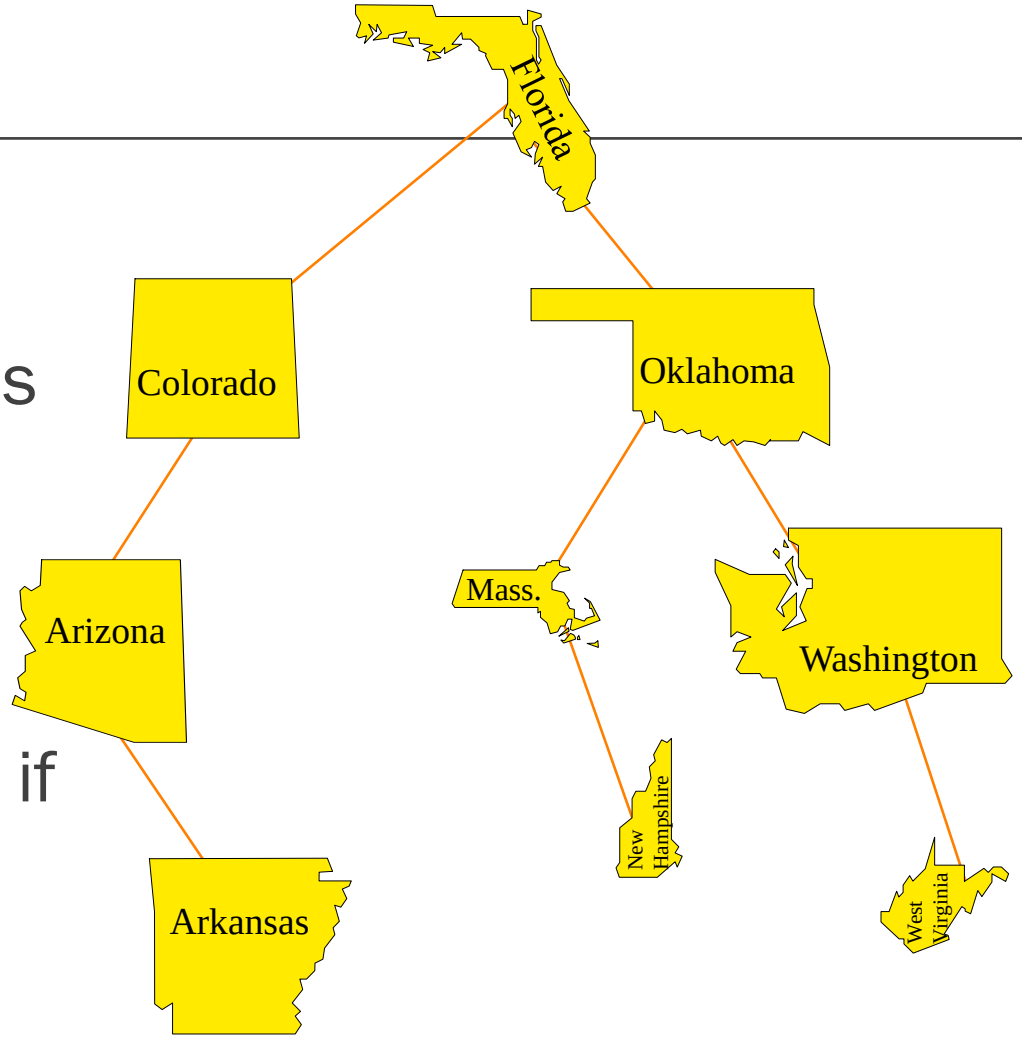
Start at the root.

- 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.



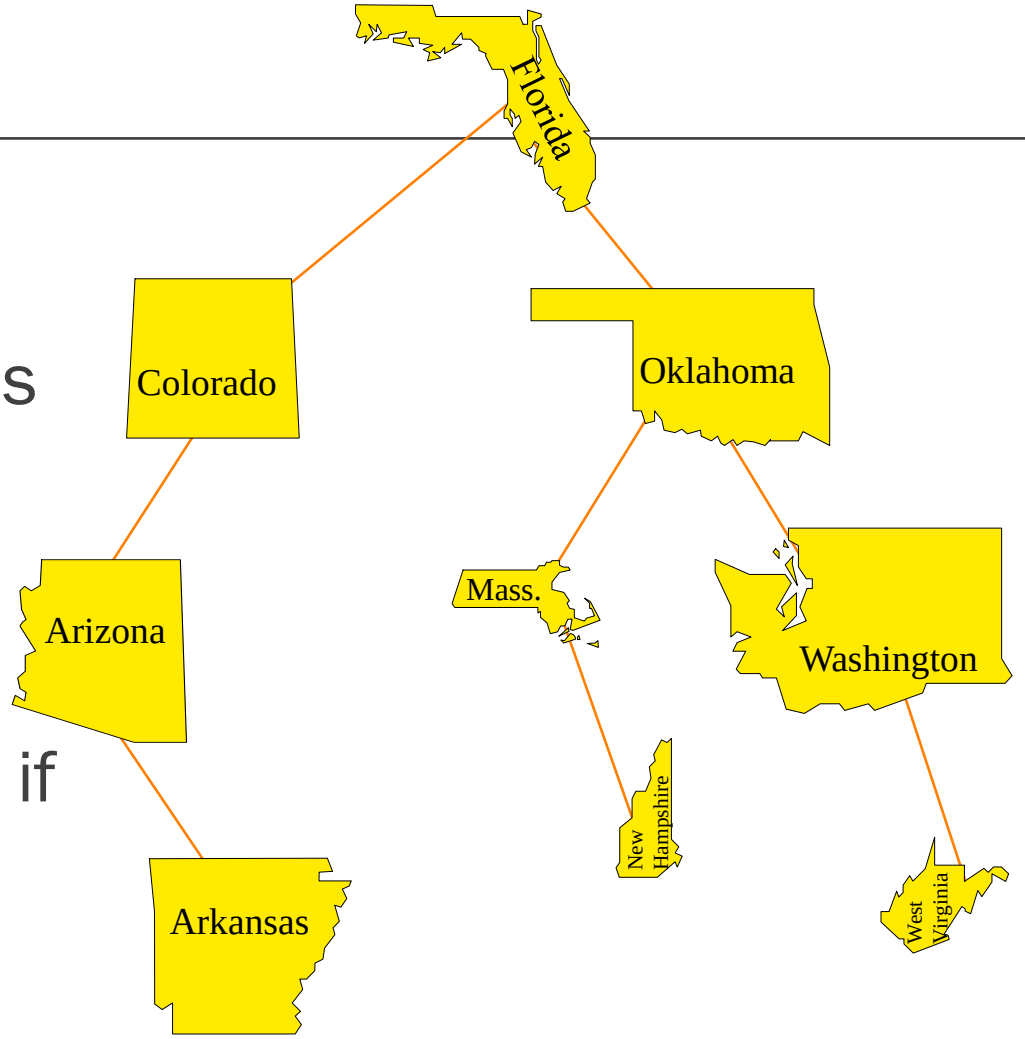
Adding a New Item with a Given Key

- 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.



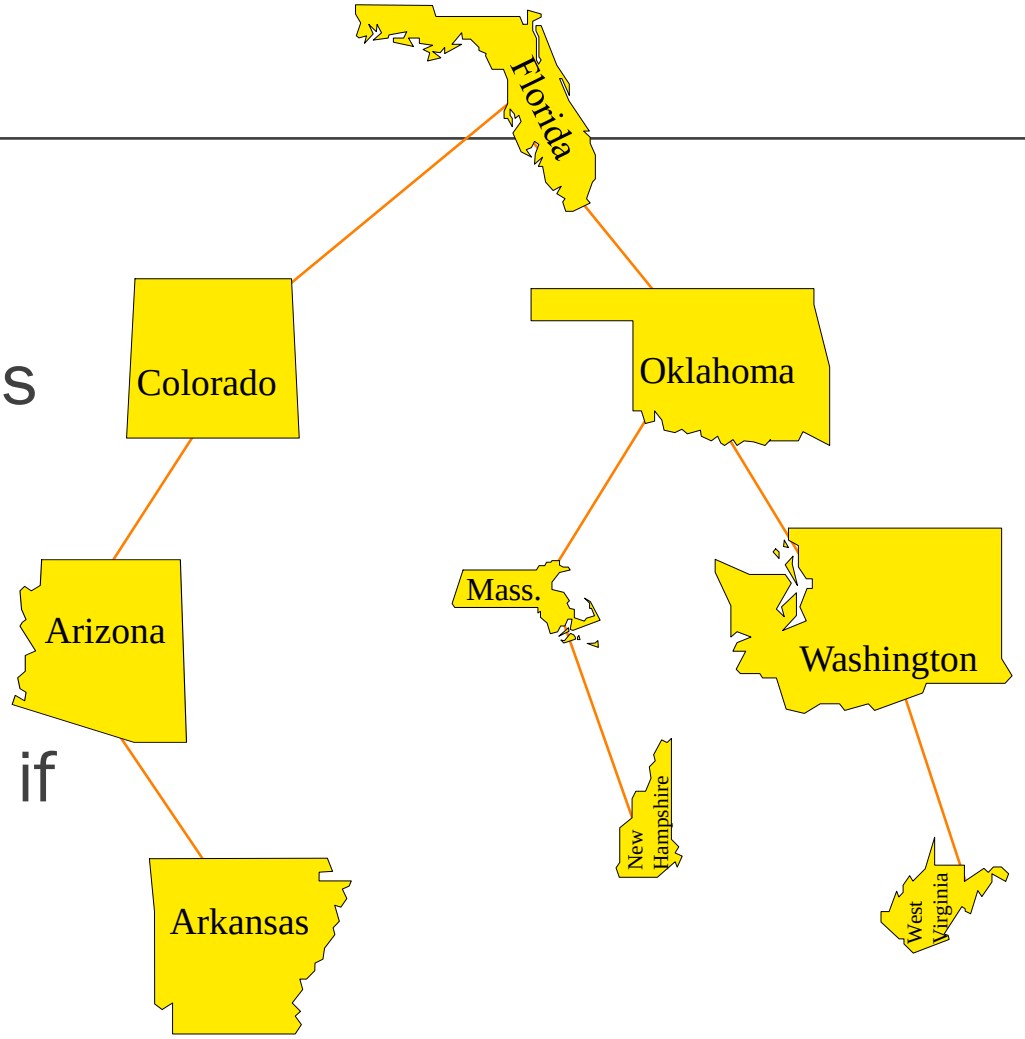
Adding 

- 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.



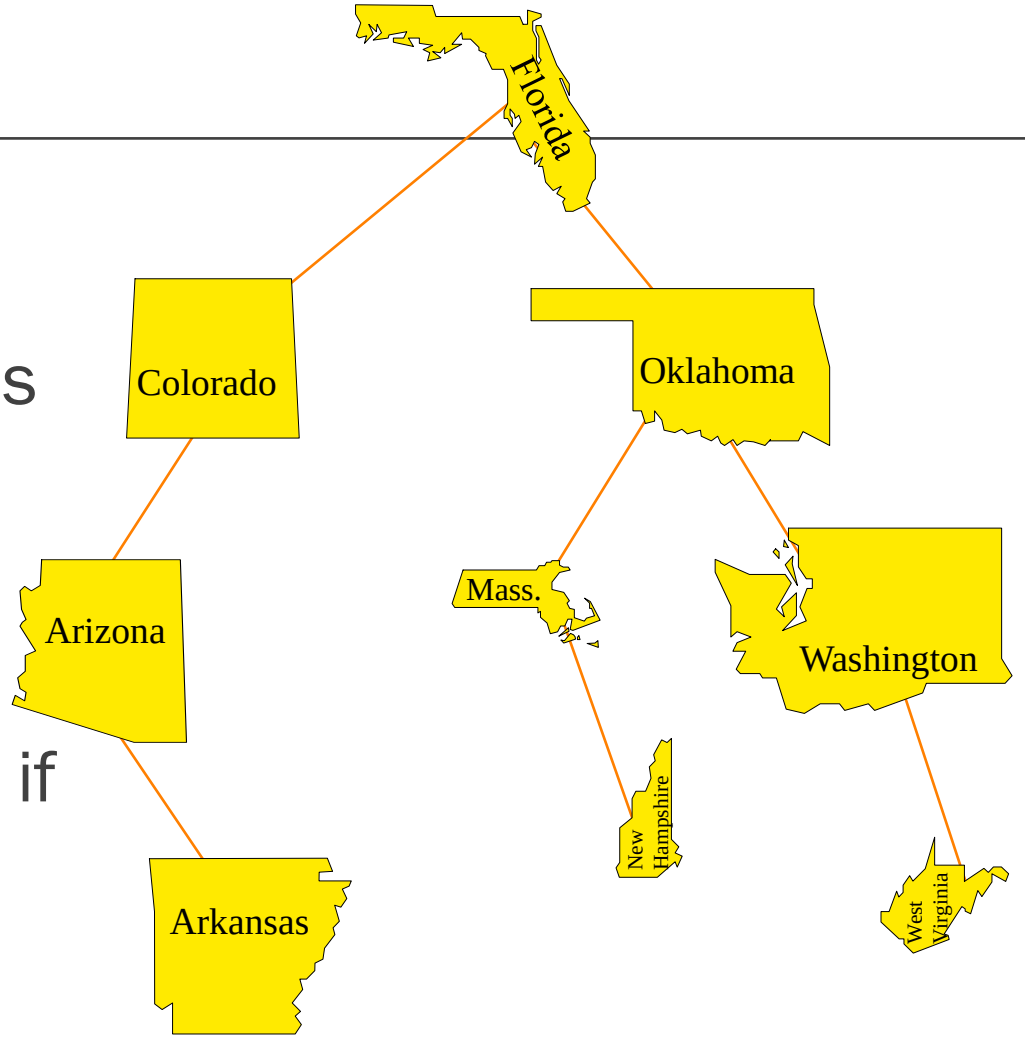
Adding 

- 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.



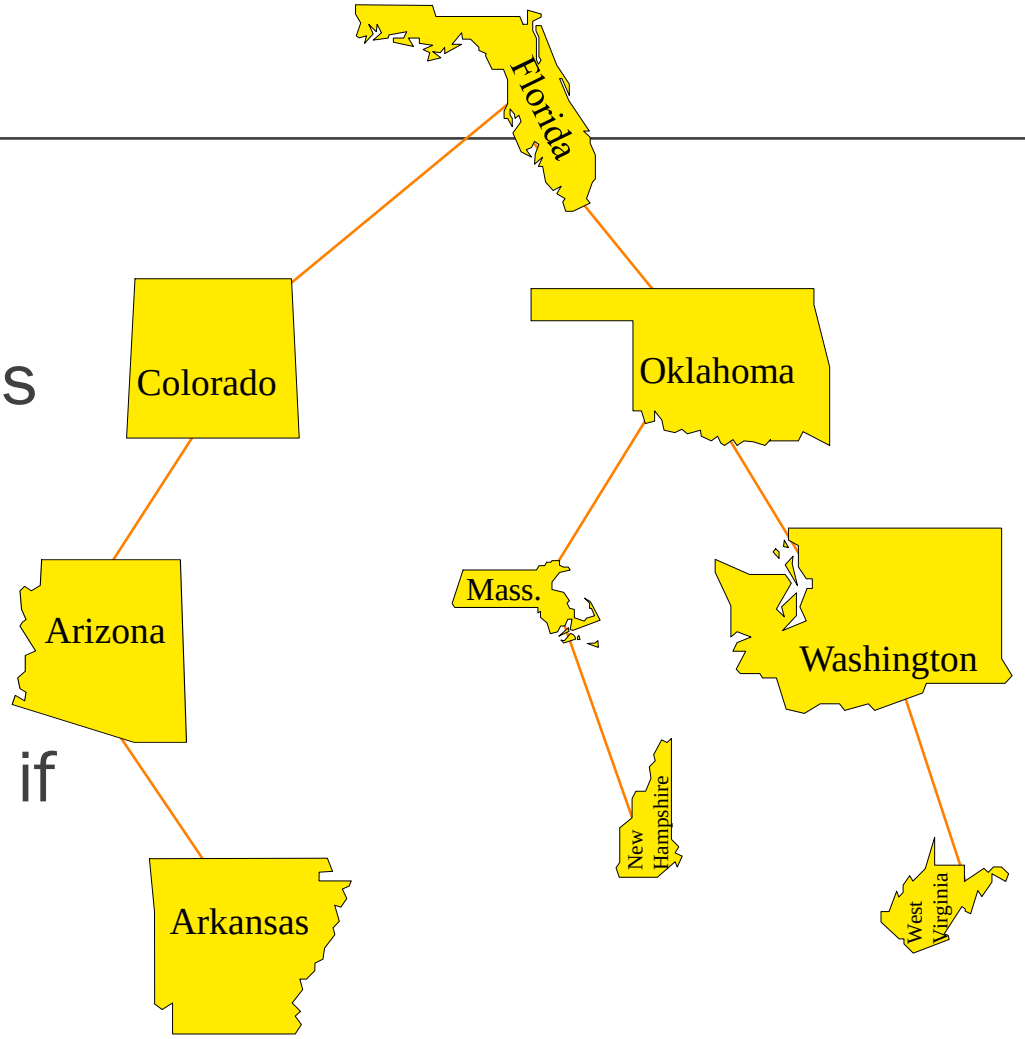
Adding 

- 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.



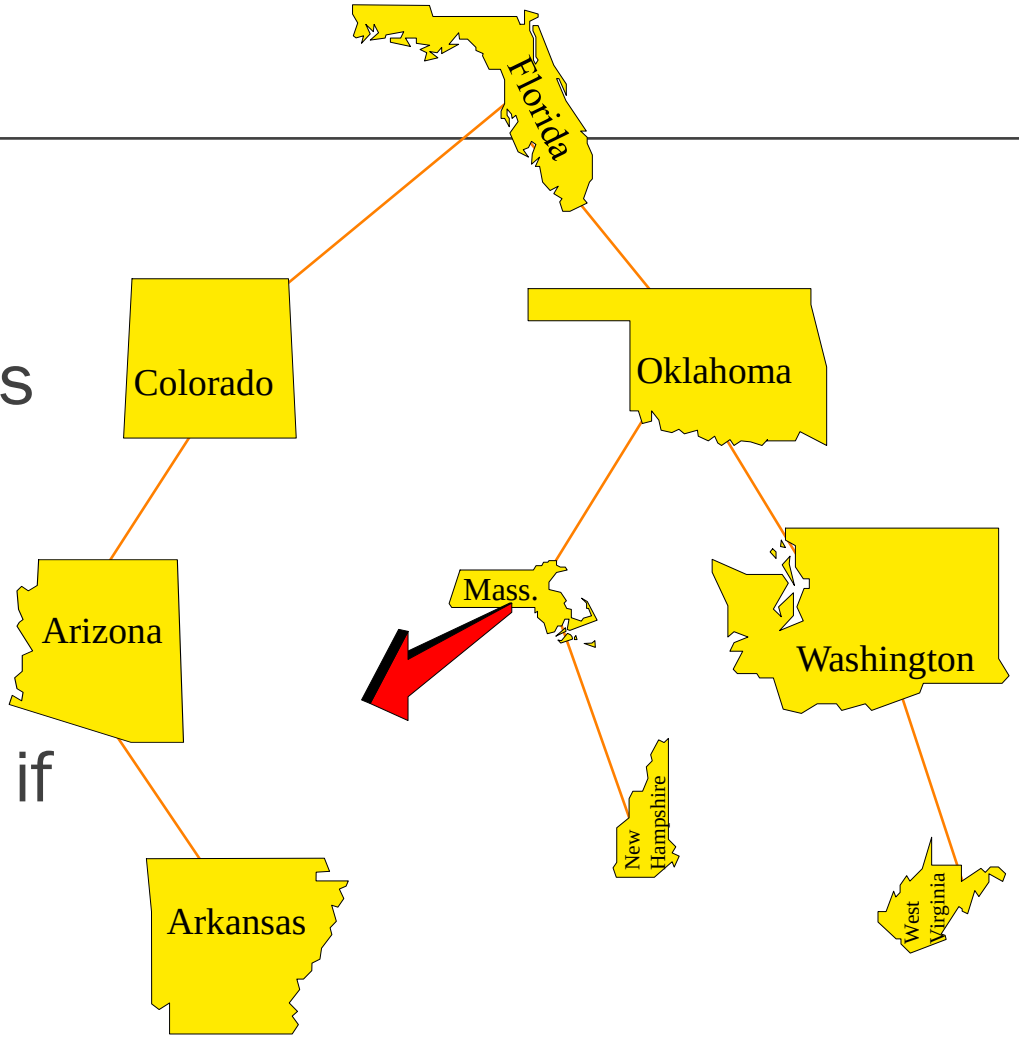
Adding 

- 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.



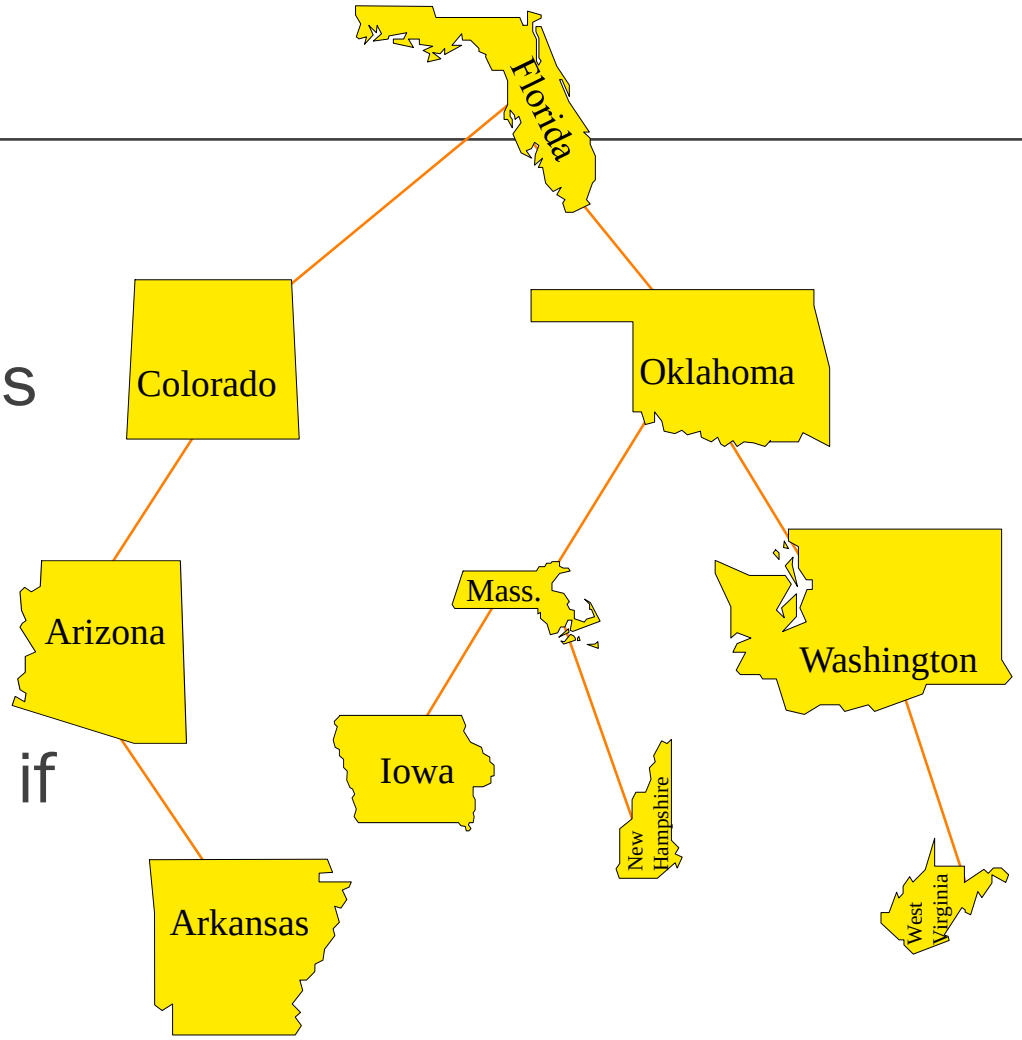
Adding 

- 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.



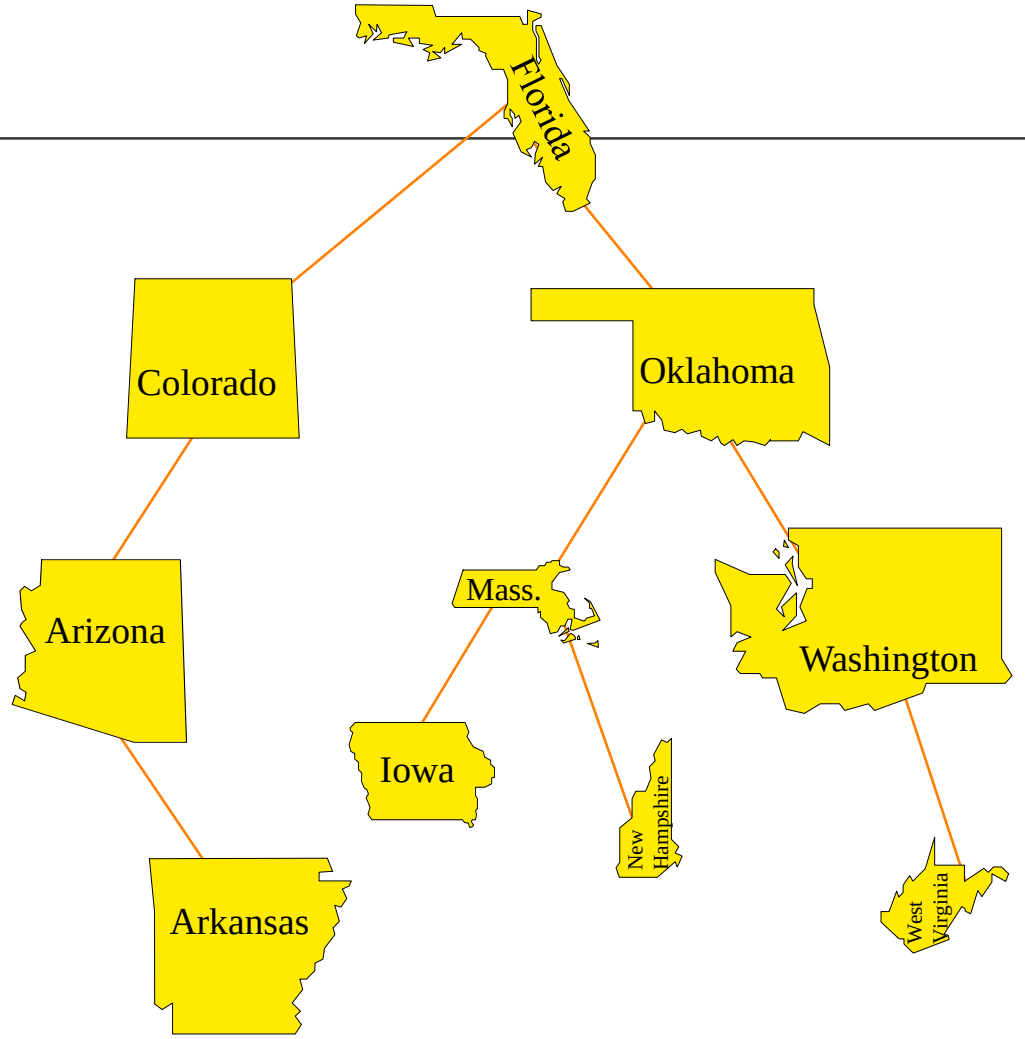
Adding

- 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.



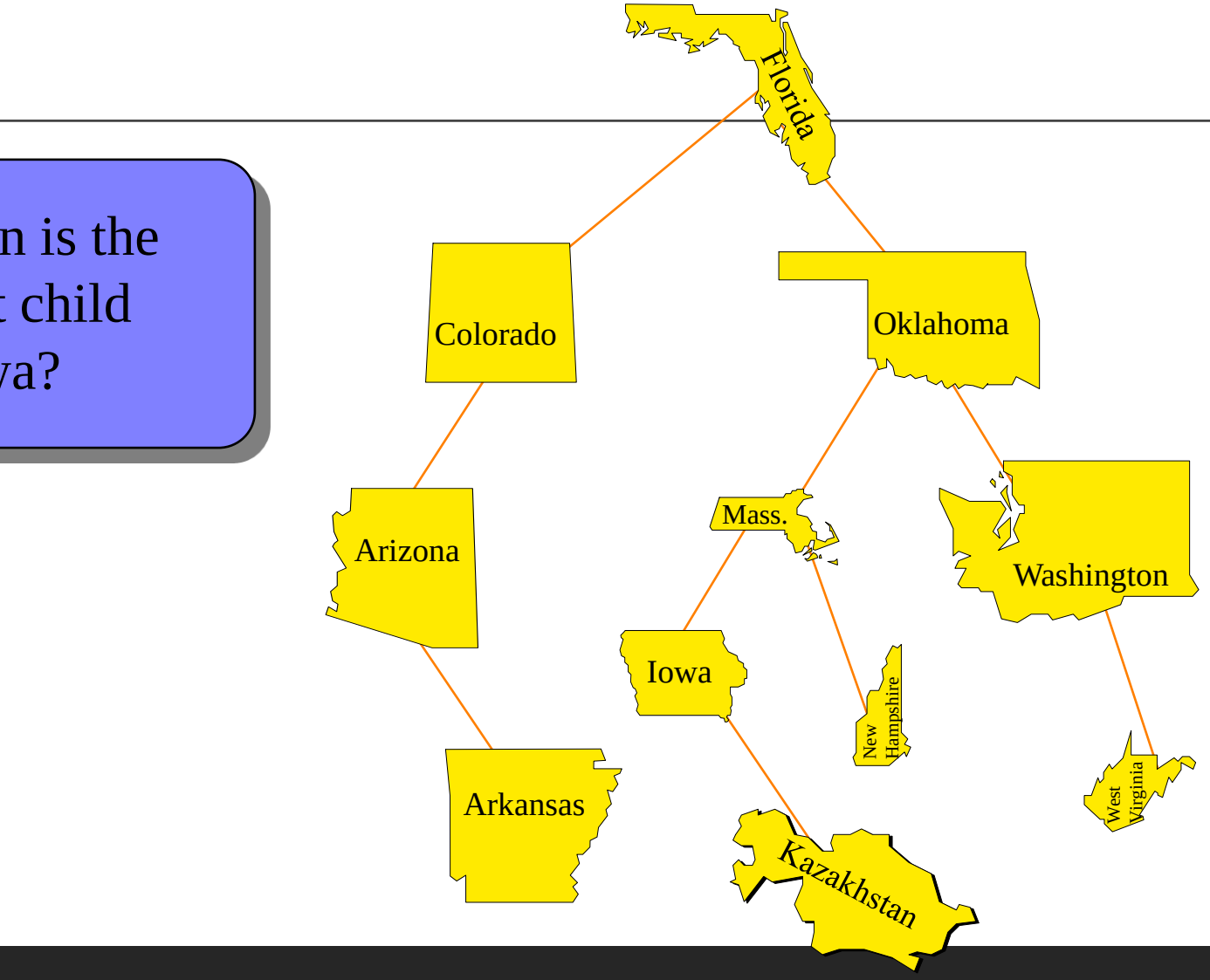
Adding

Where would you add this state?



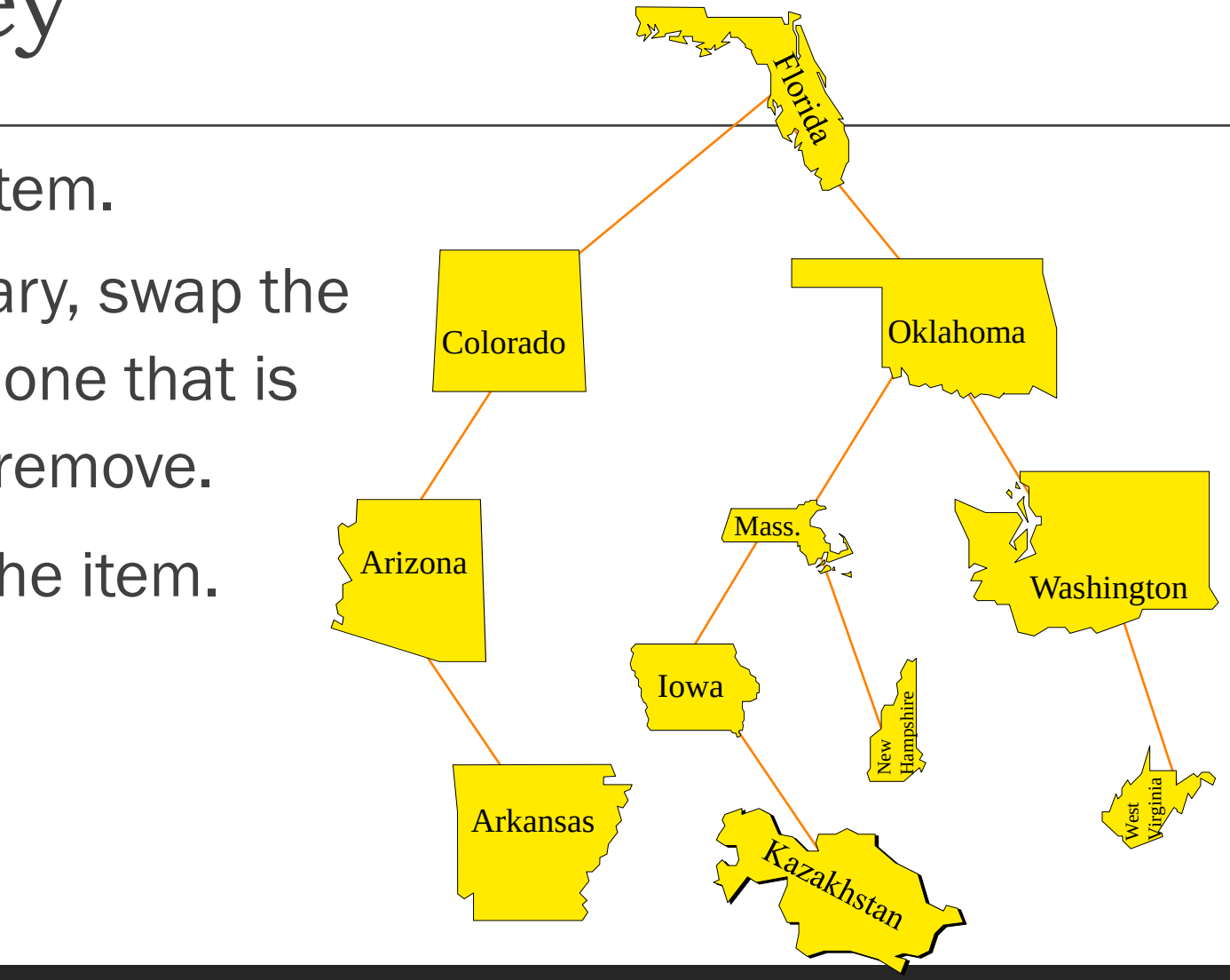
Adding

Kazakhstan is the new right child of Iowa?



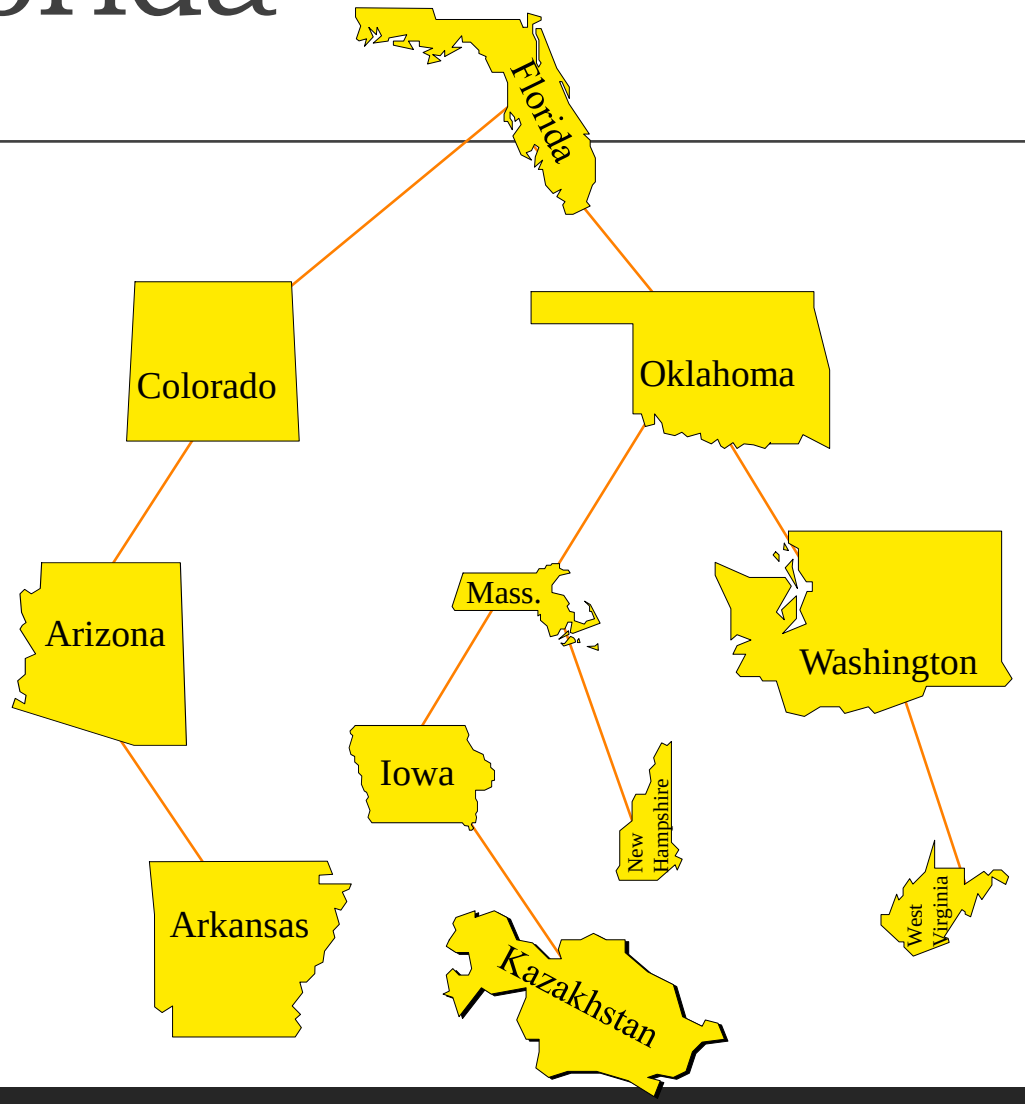
Removing an Item with a Given Key

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



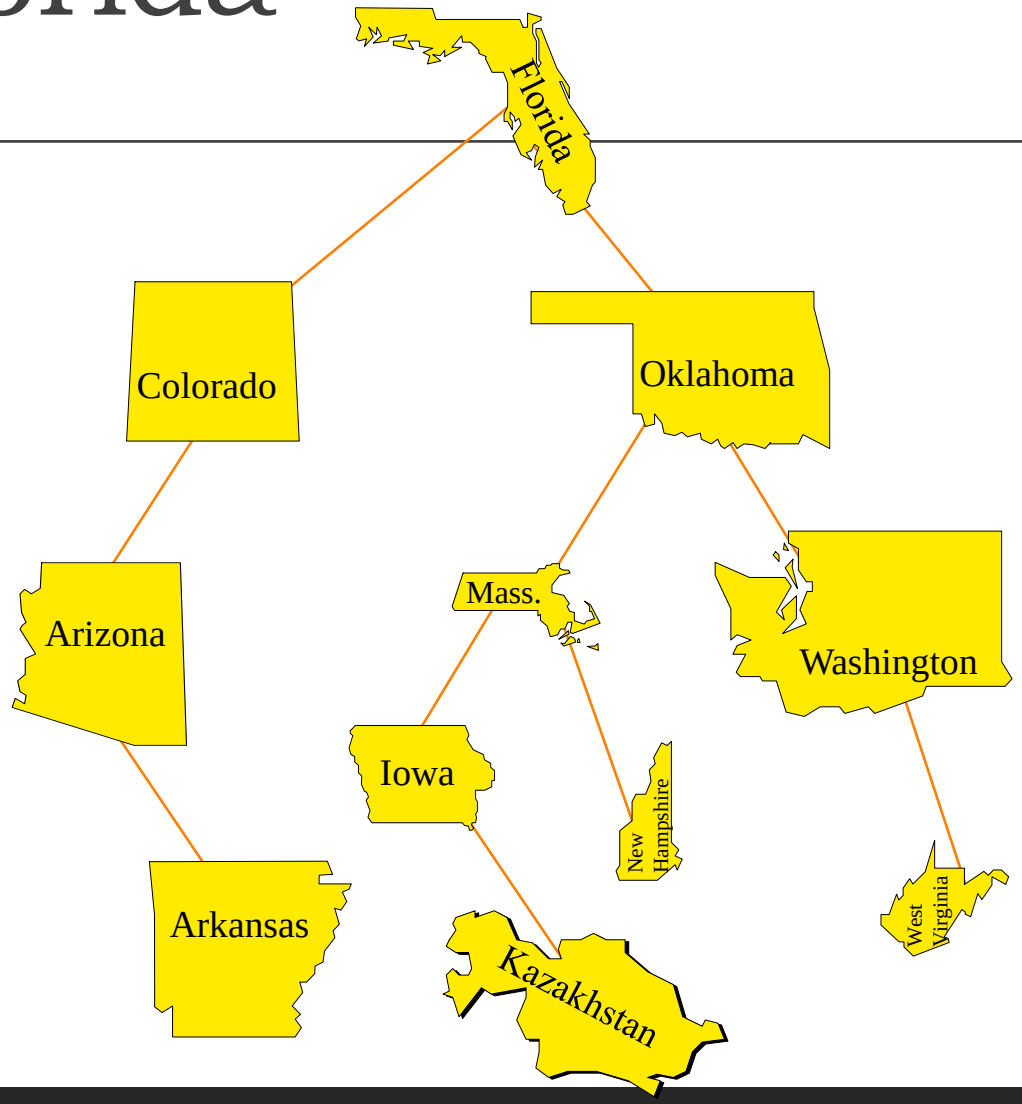
Removing "Florida"

- Find the item.



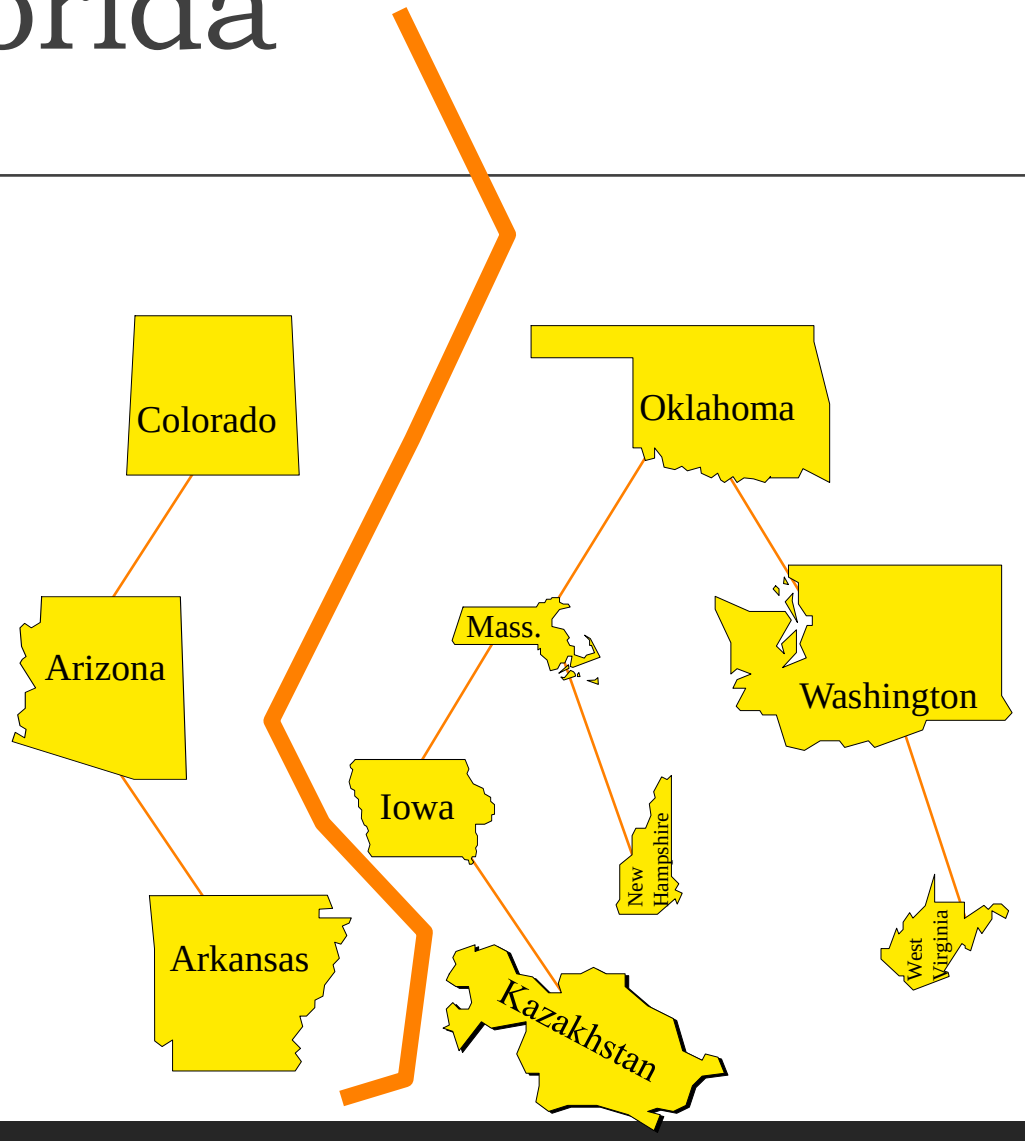
Removing "Florida"

Florida cannot be removed at the moment...

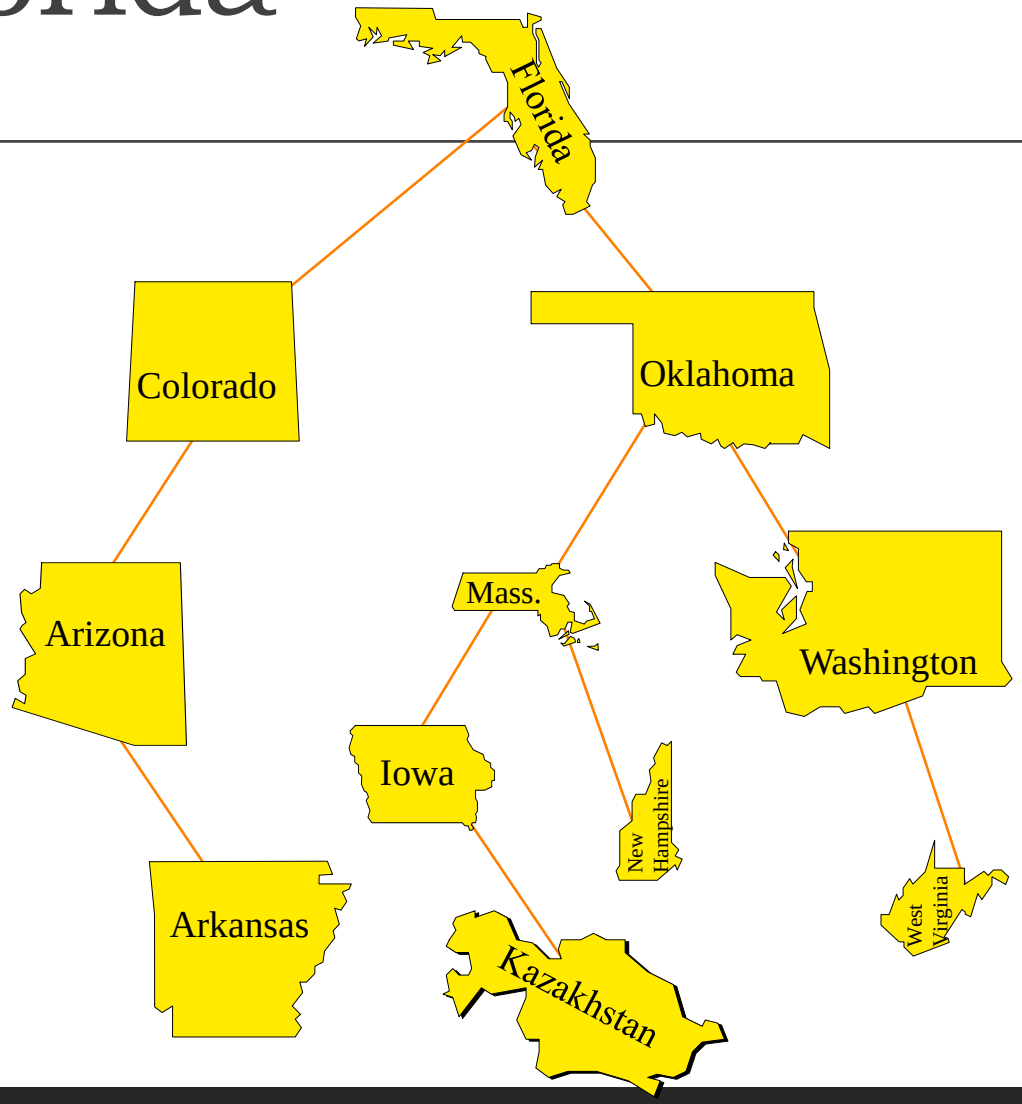


Removing "Florida"

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



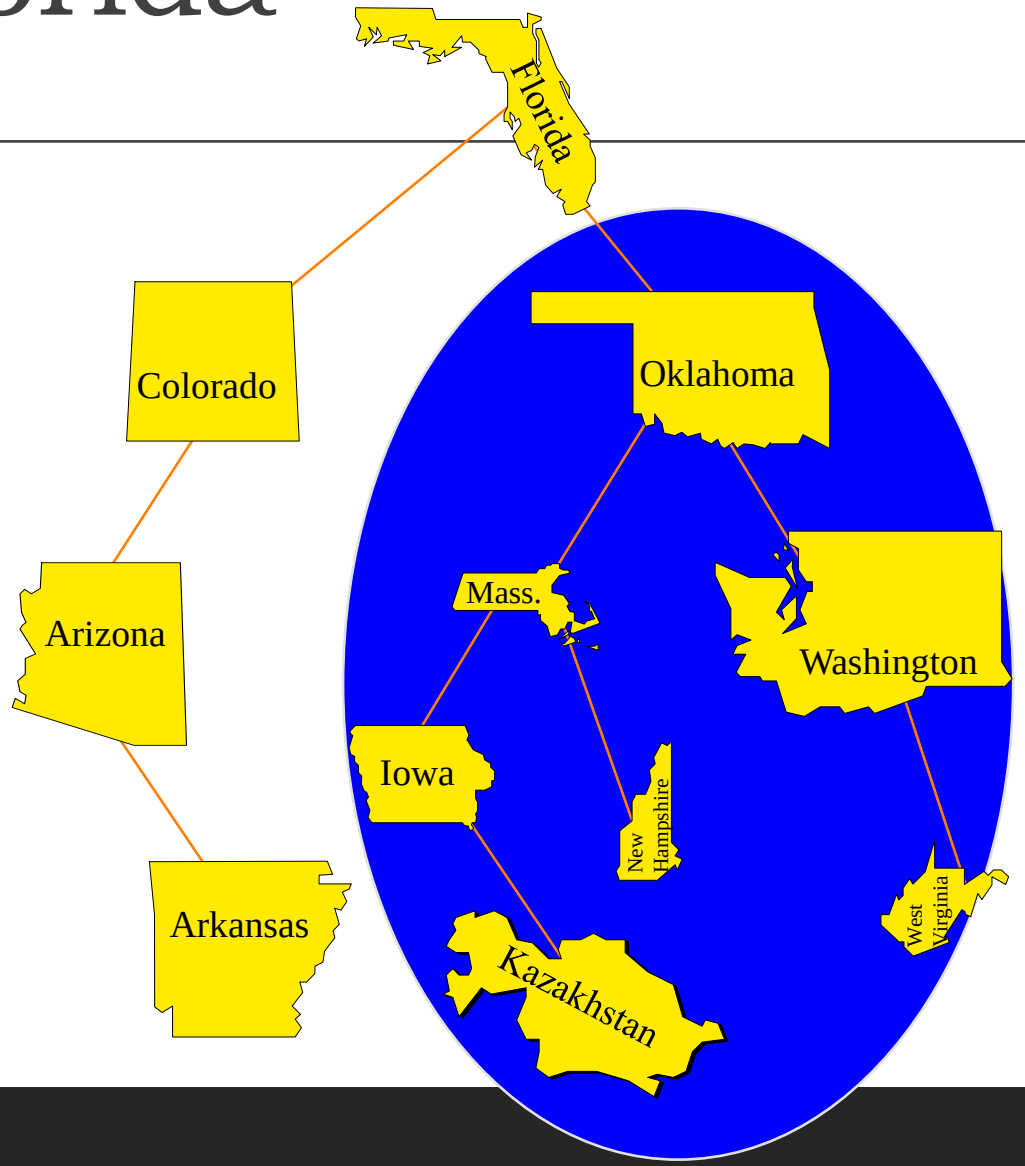
Removing "Florida"



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

Removing "Florida"

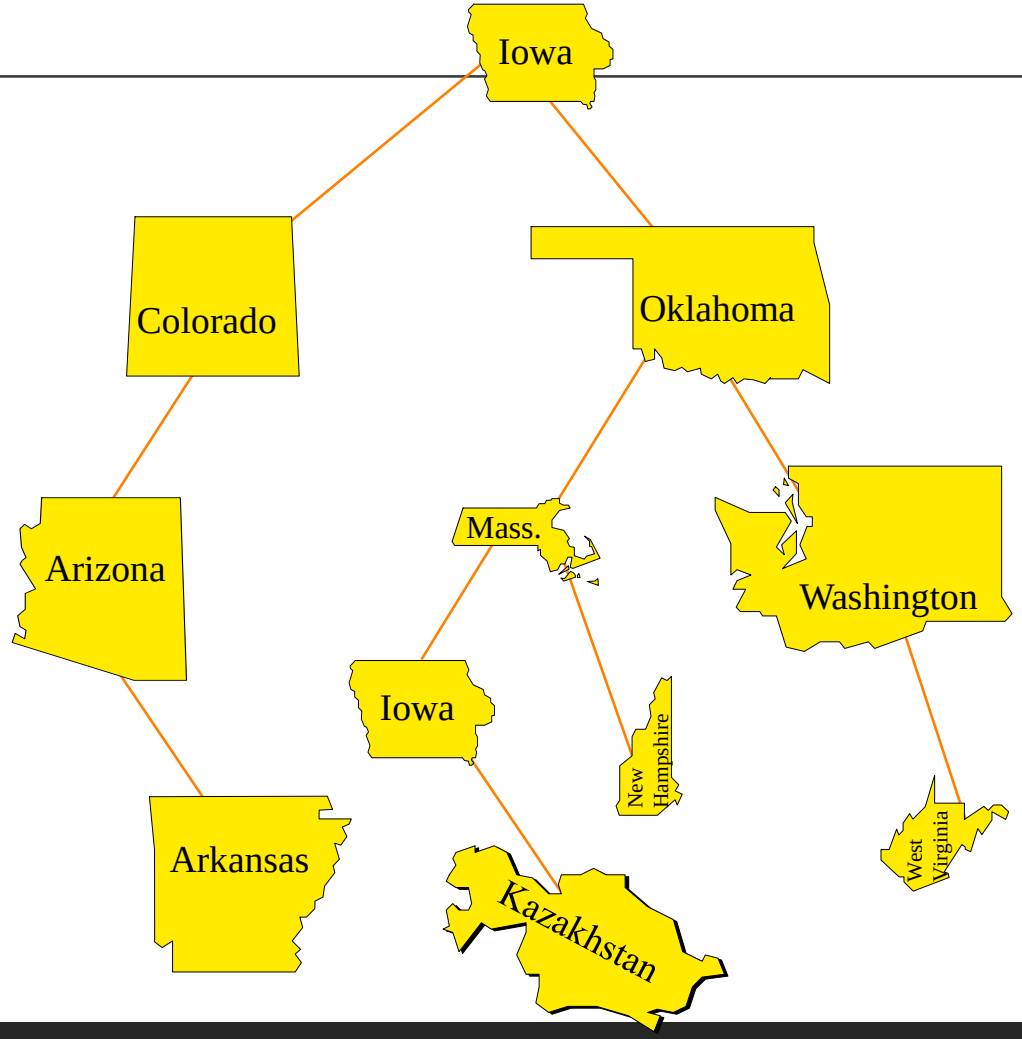
For the rearranging, take the **smallest** item in the right subtree...



Removing "Florida"

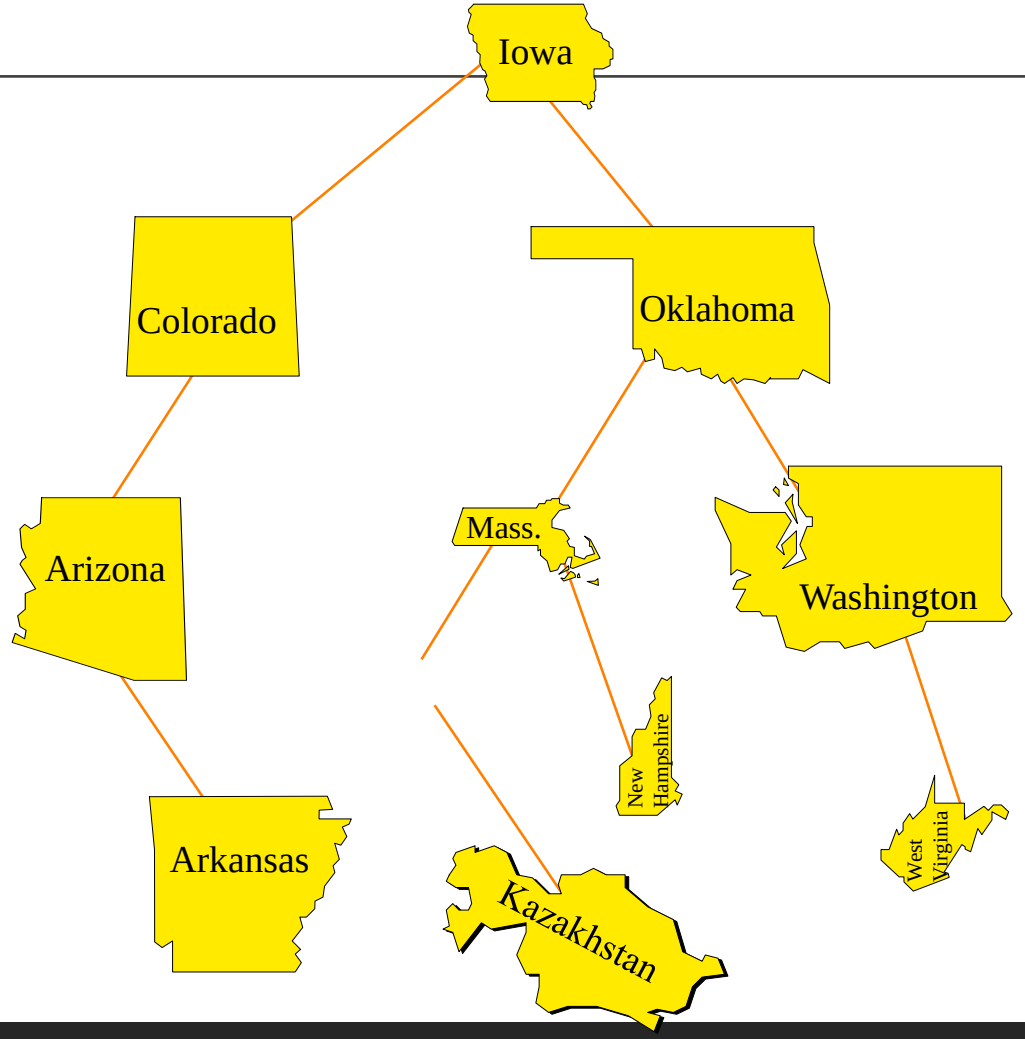
-

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



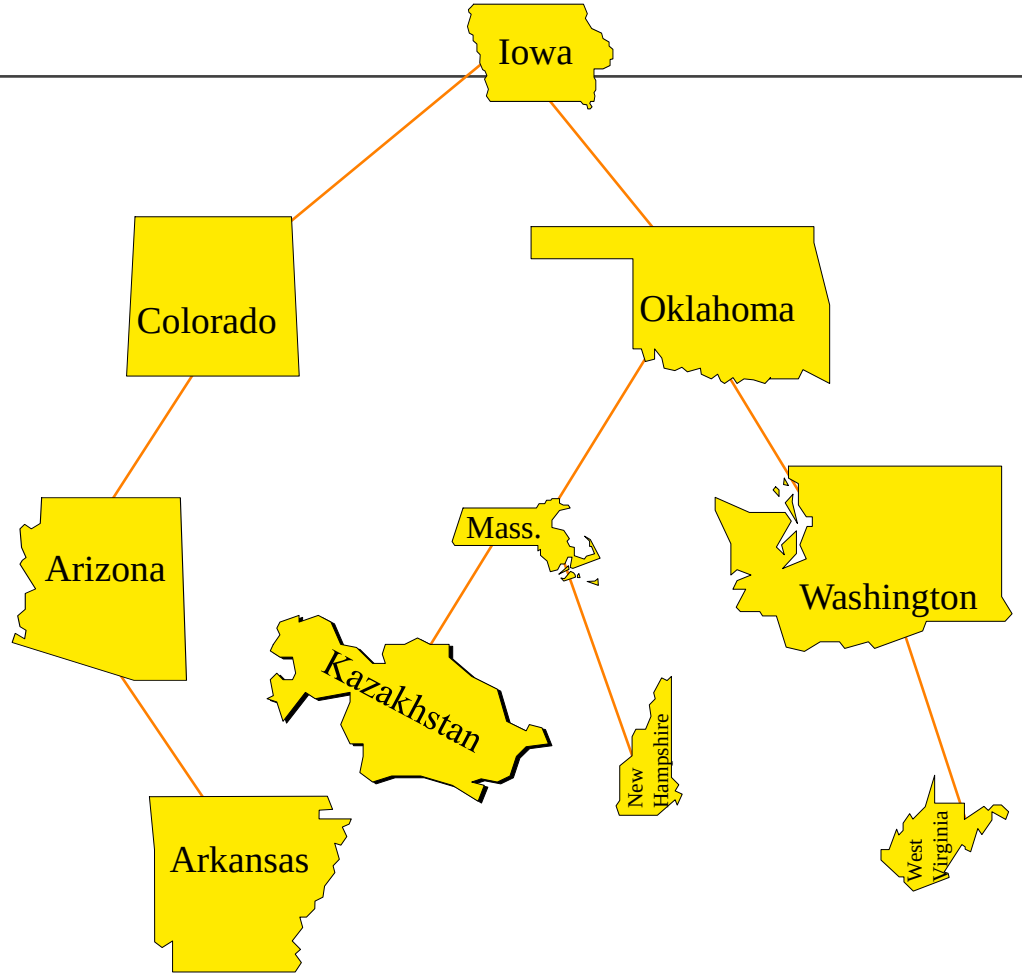
Removing "Florida"

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

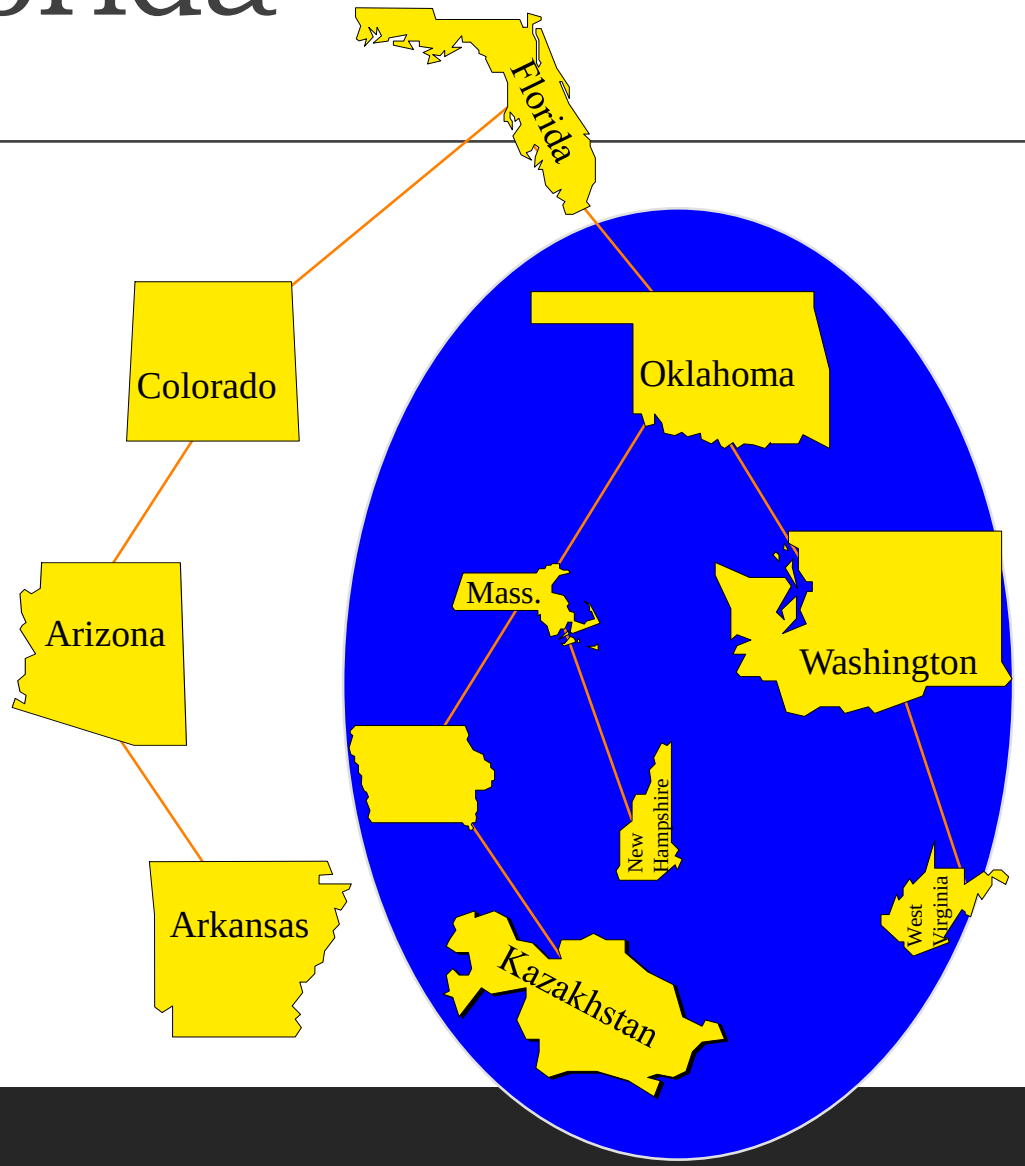


Removing "Florida"

... and reconnect the tree

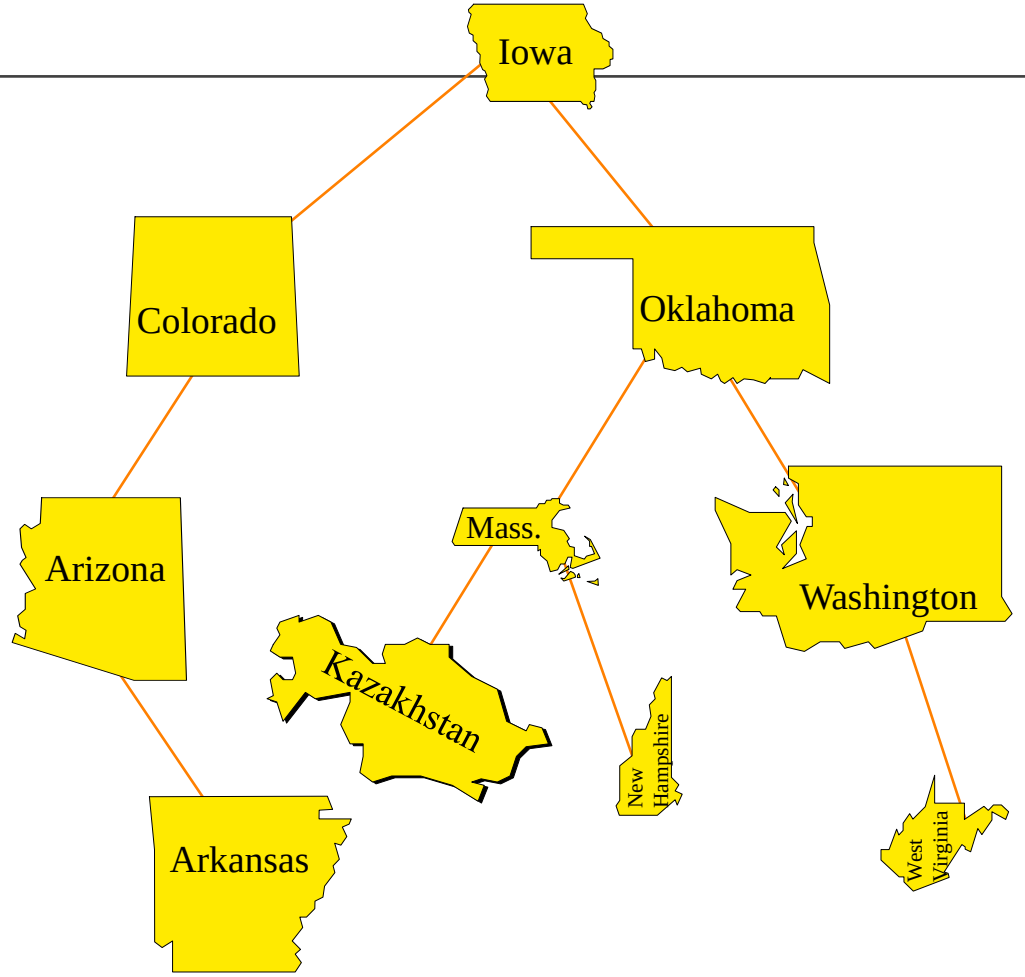


Removing "Florida"



*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

- 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.

Credits and Acknowledgements

https://www.gatevidyalay.com

https://www.programiz.com/