CSC212

Data Structure

COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

Lecture 13

Trees and Tree Traversals

Instructor: Prof. George Wolberg
Department of Computer Science
City College of New York

Motivation

- Linear structures
  - arrays
  - dynamic arrays
  - linked lists
- Nonlinear Structures
  - trees - Hierarchical Structures
  - Graphs
- Why???

Application: Mailing Addresses

George Wolberg, CS Dept, CCNY, New York, NY 10031, USA

6 billion = 6,000,000,000 people in the world

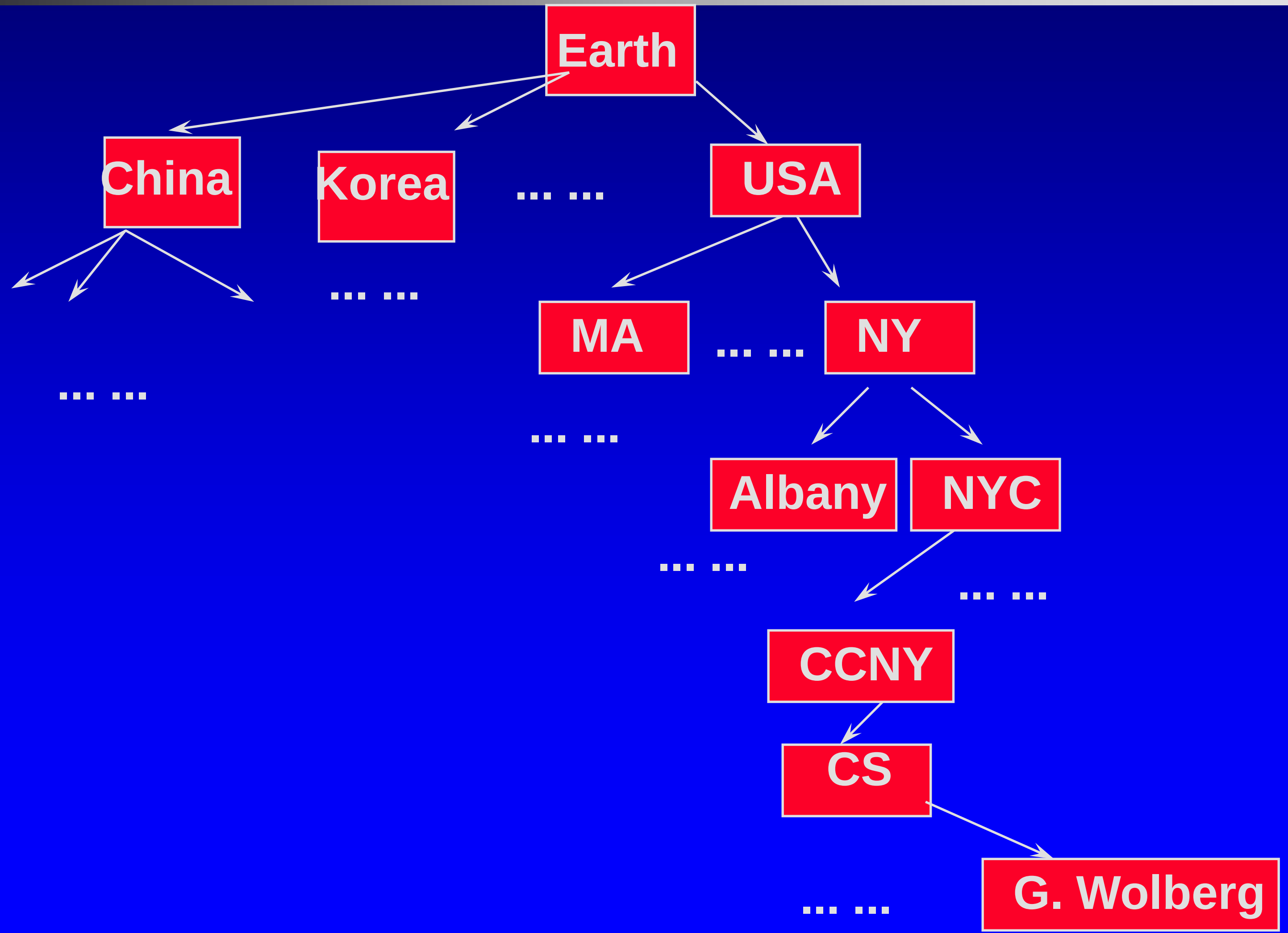
What kind of structure is the best for a postman to locate me?

Array ?

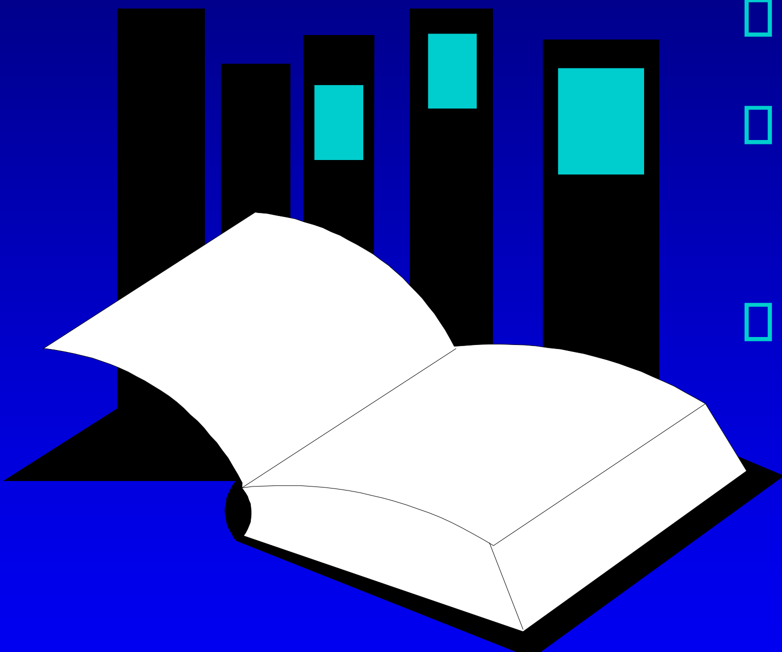
Linked list ?

Tree ?

A Tree for all the mailing addresses



Trees and Binary Trees



- Chapter 10 introduces **trees**.
- This presentation illustrates basic terminology for binary trees
- and focuses on
  - **Complete Binary Trees:** the simplest kind of trees
  - **Binary Tree Traversals:** any kind of binary trees

Data Structures and Other Objects Using C++

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.

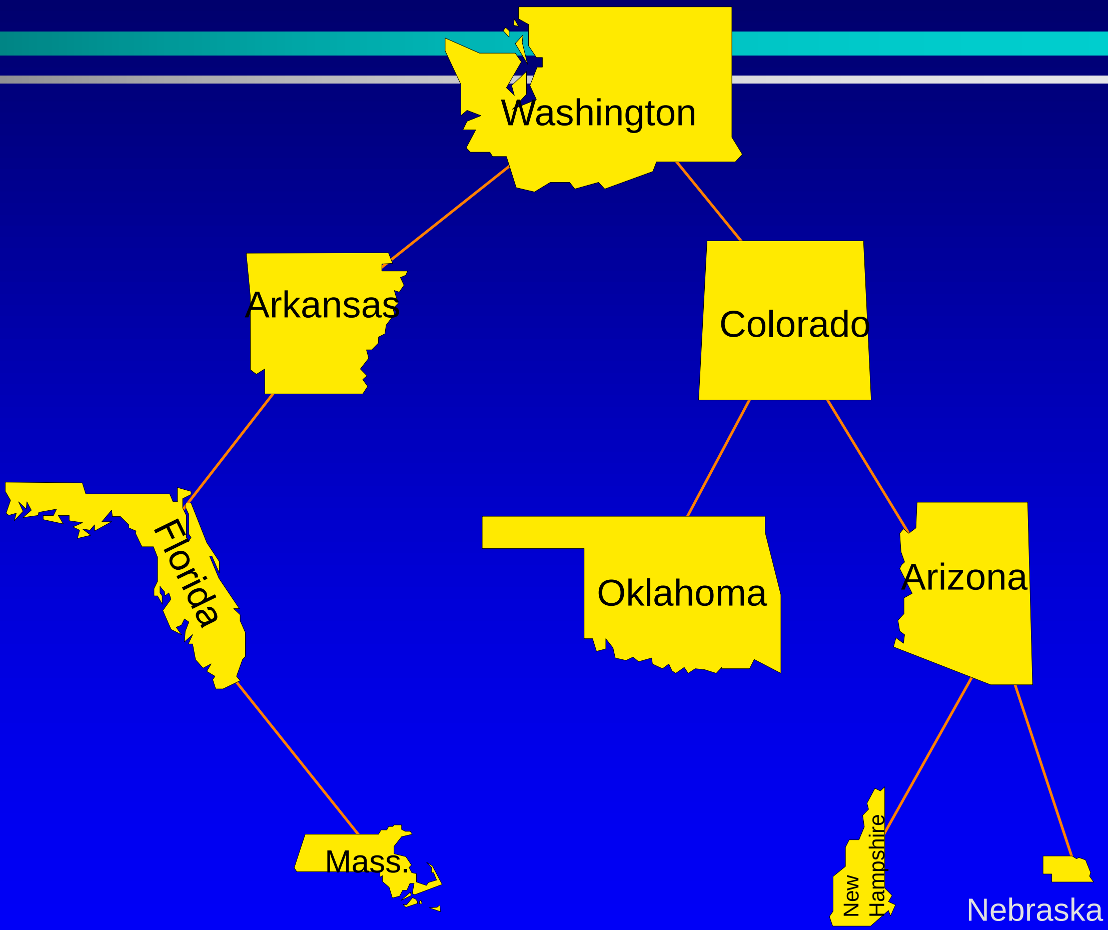
Binary Trees

- 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.

An example can illustrate how the connections work

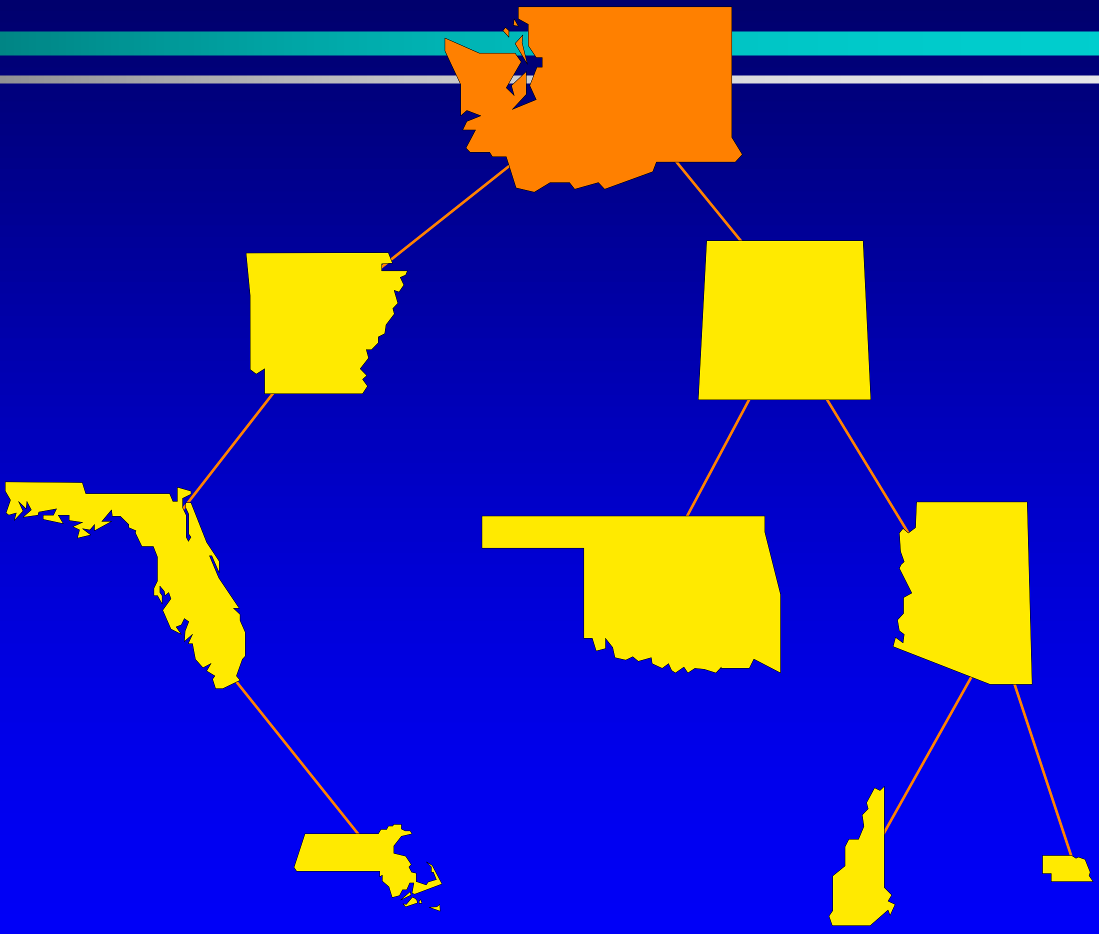
A Binary Tree of States

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.

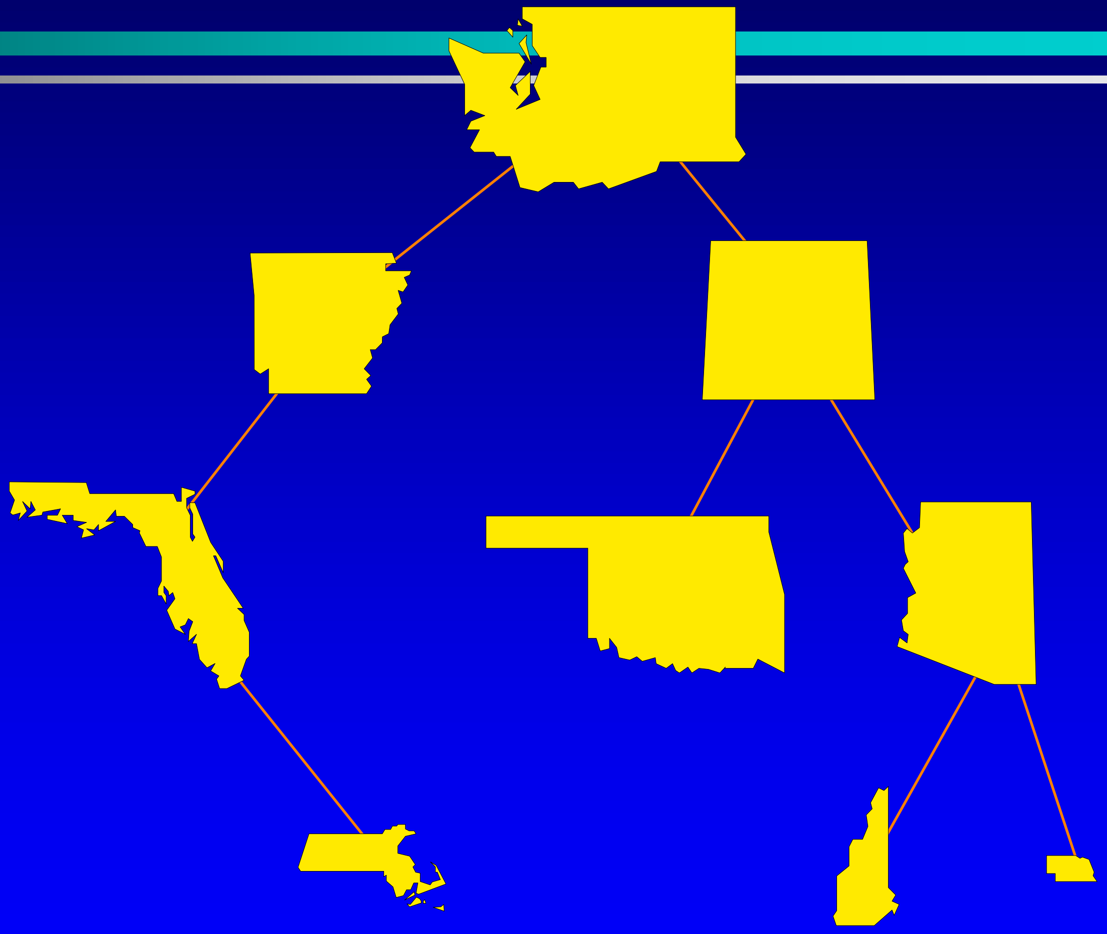


A Binary Tree of States

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

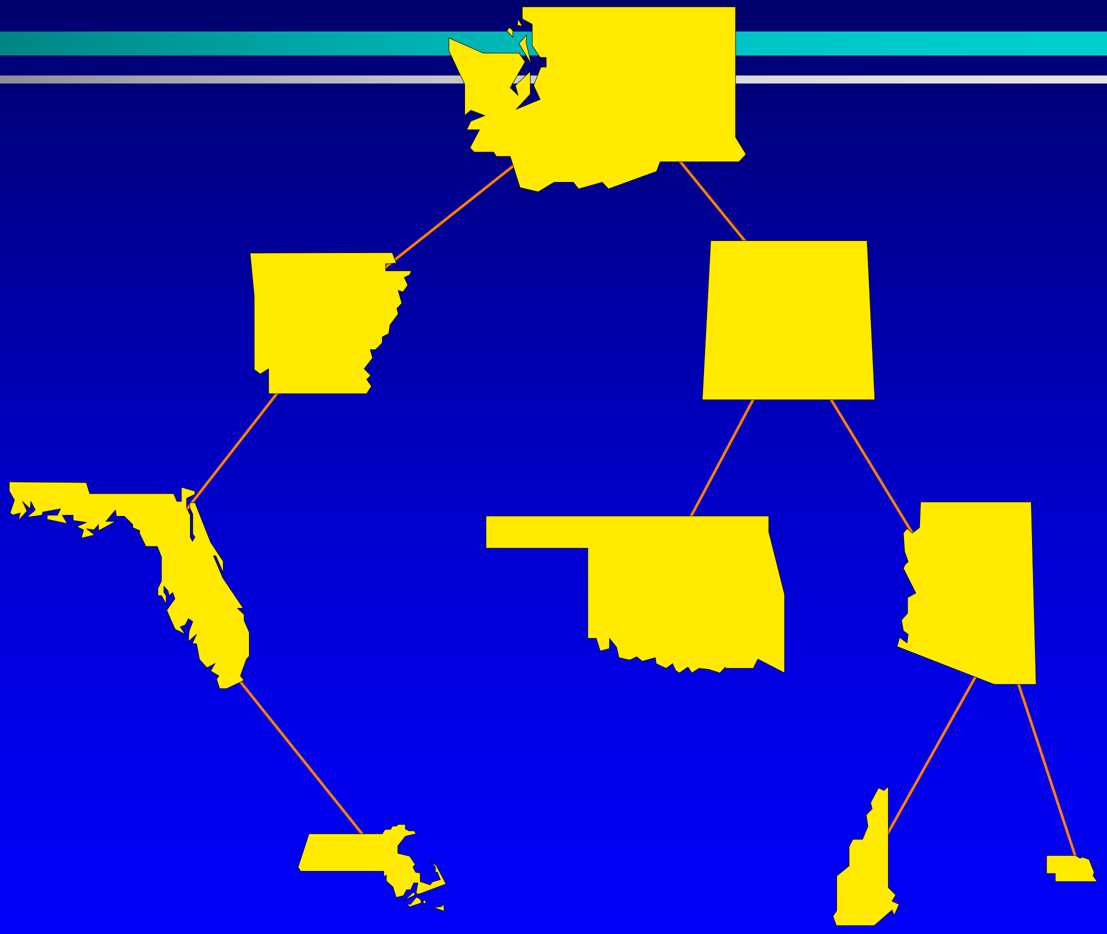
A Binary Tree of States

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



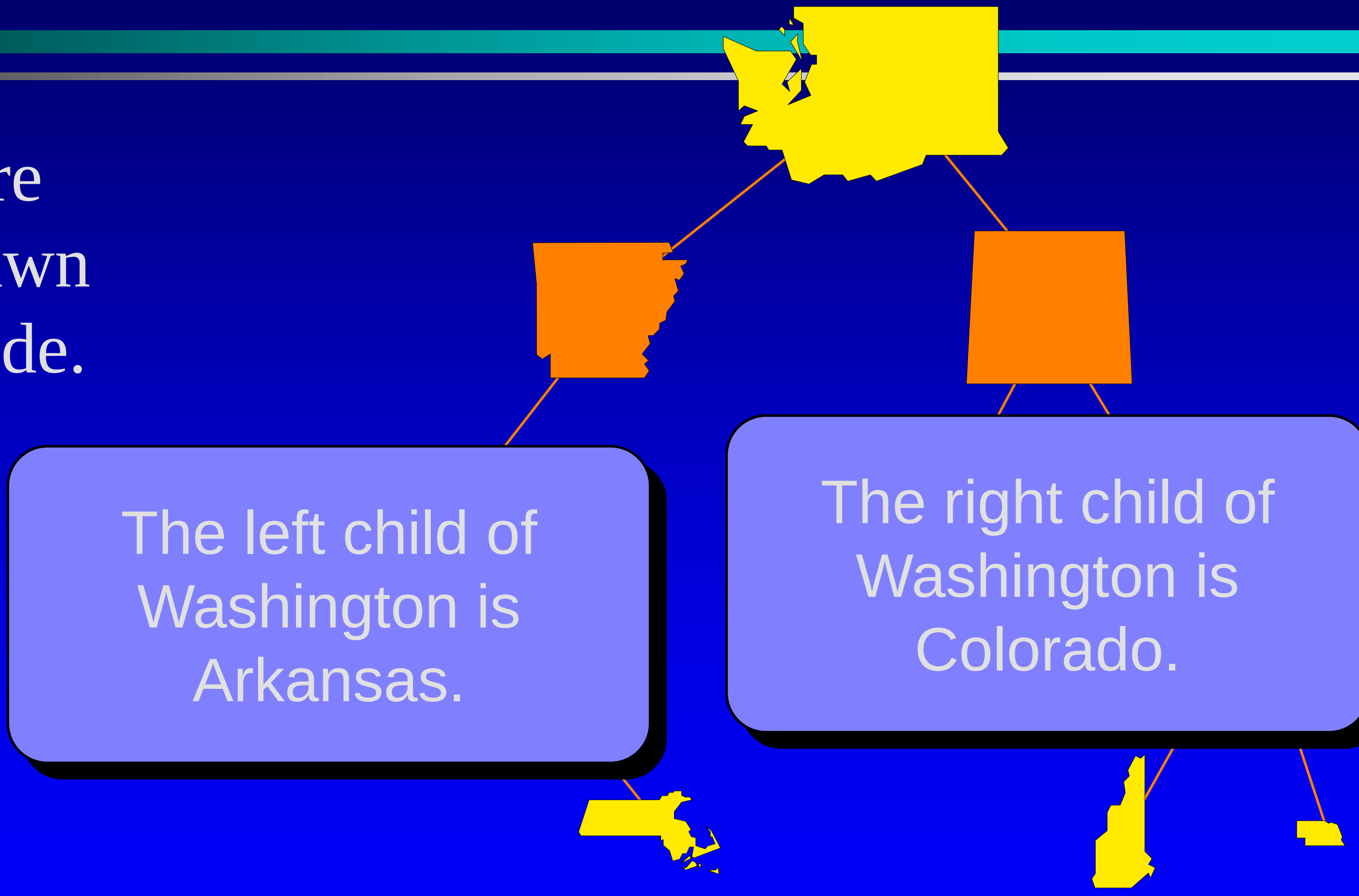
A Binary Tree of States

Each node is permitted to have $\boxed{\text{two links}}$ to other nodes, called the **left child** and the **right child**.



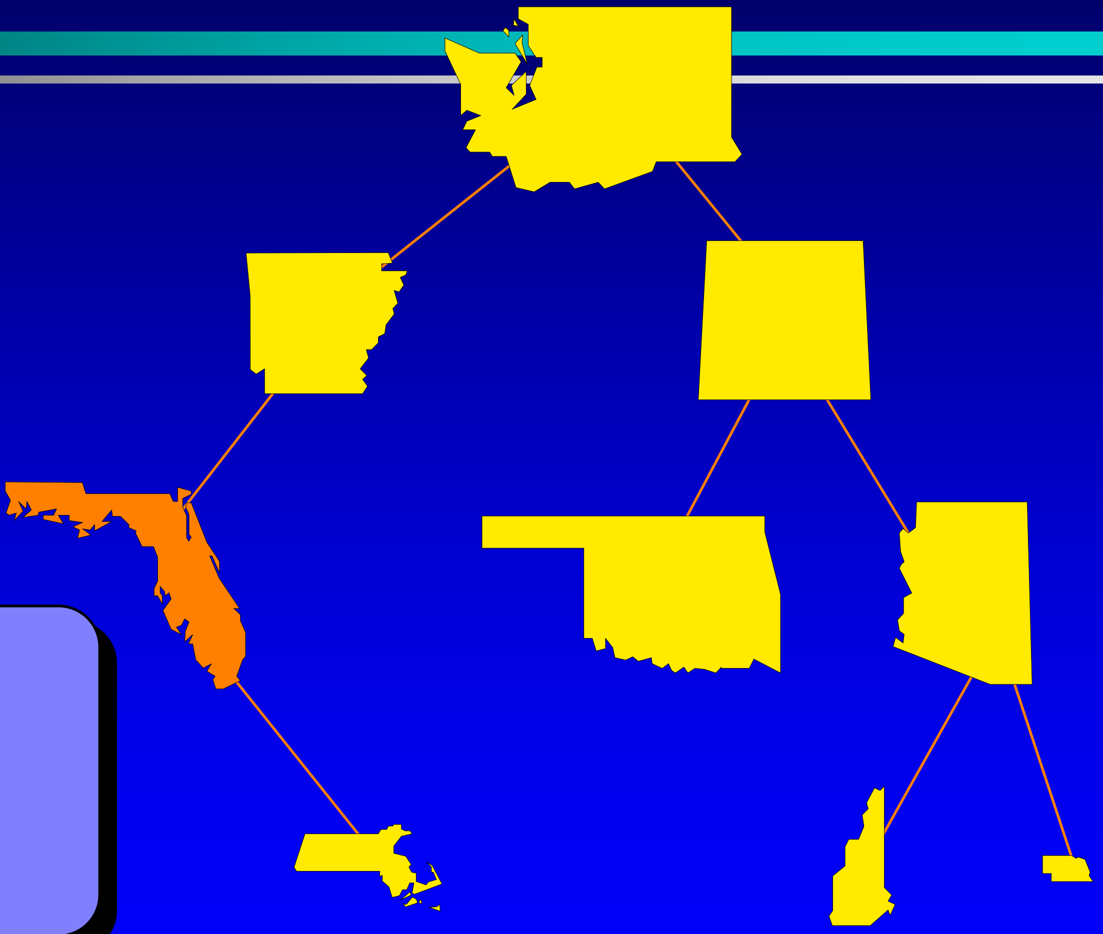
A Binary Tree of States

Children are usually drawn below a node.



A Binary Tree of States

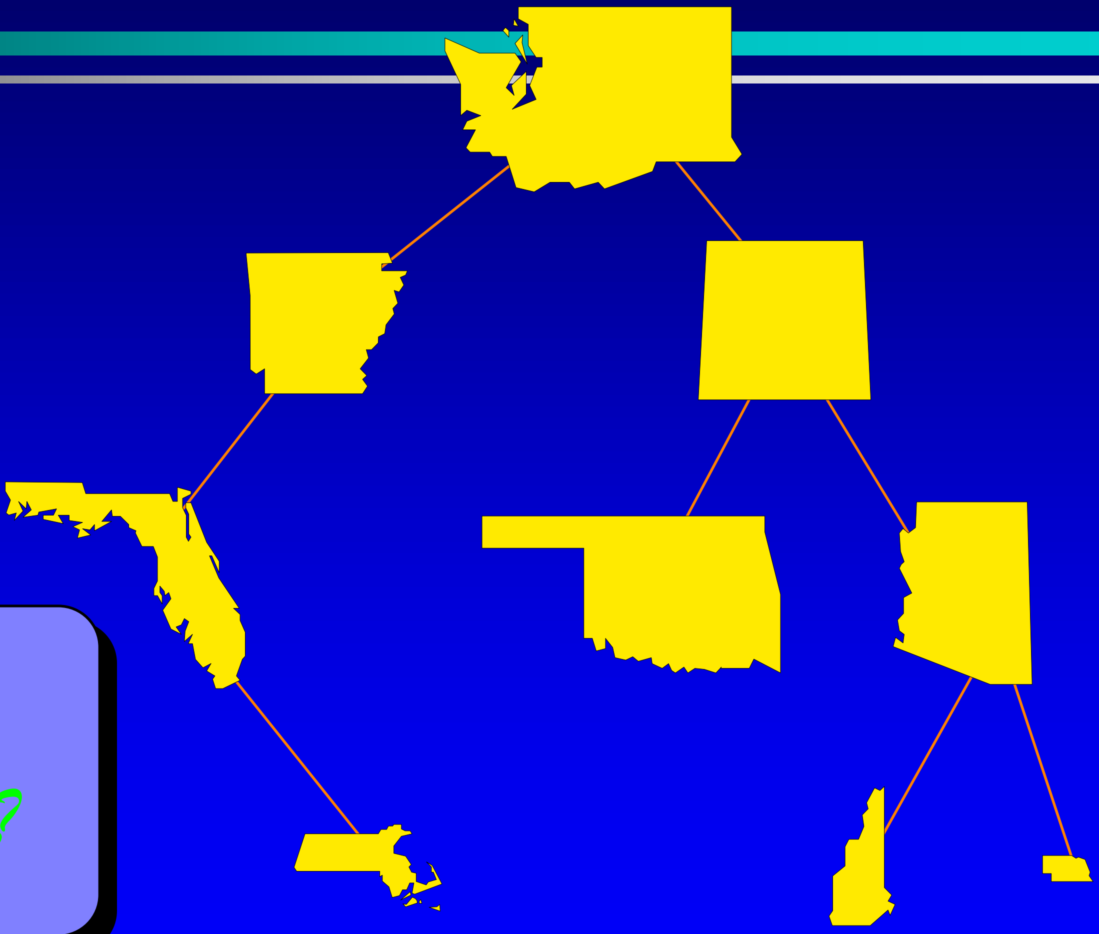
Some nodes have only one child.



Arkansas has a left child, but no right child.

A Quiz

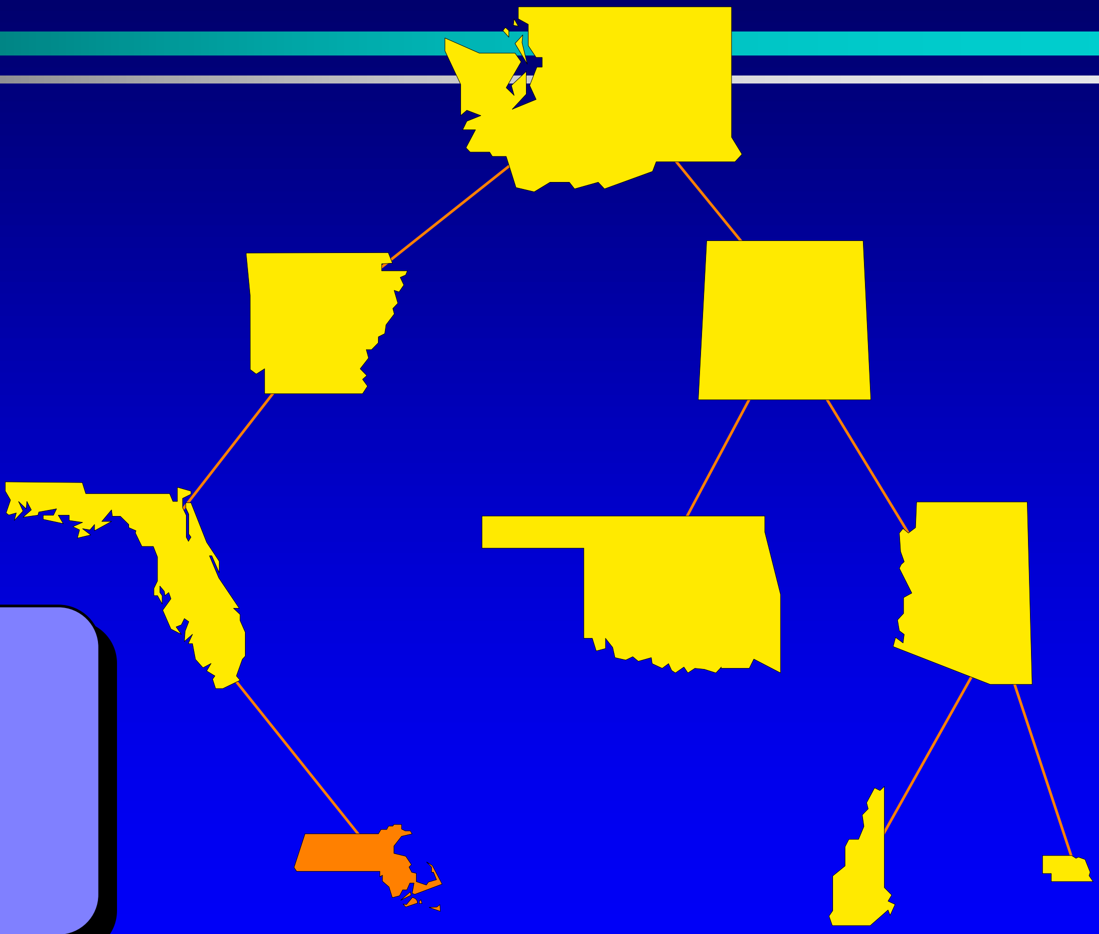
Some nodes have only one child.



Which node has only a right child?

A Quiz

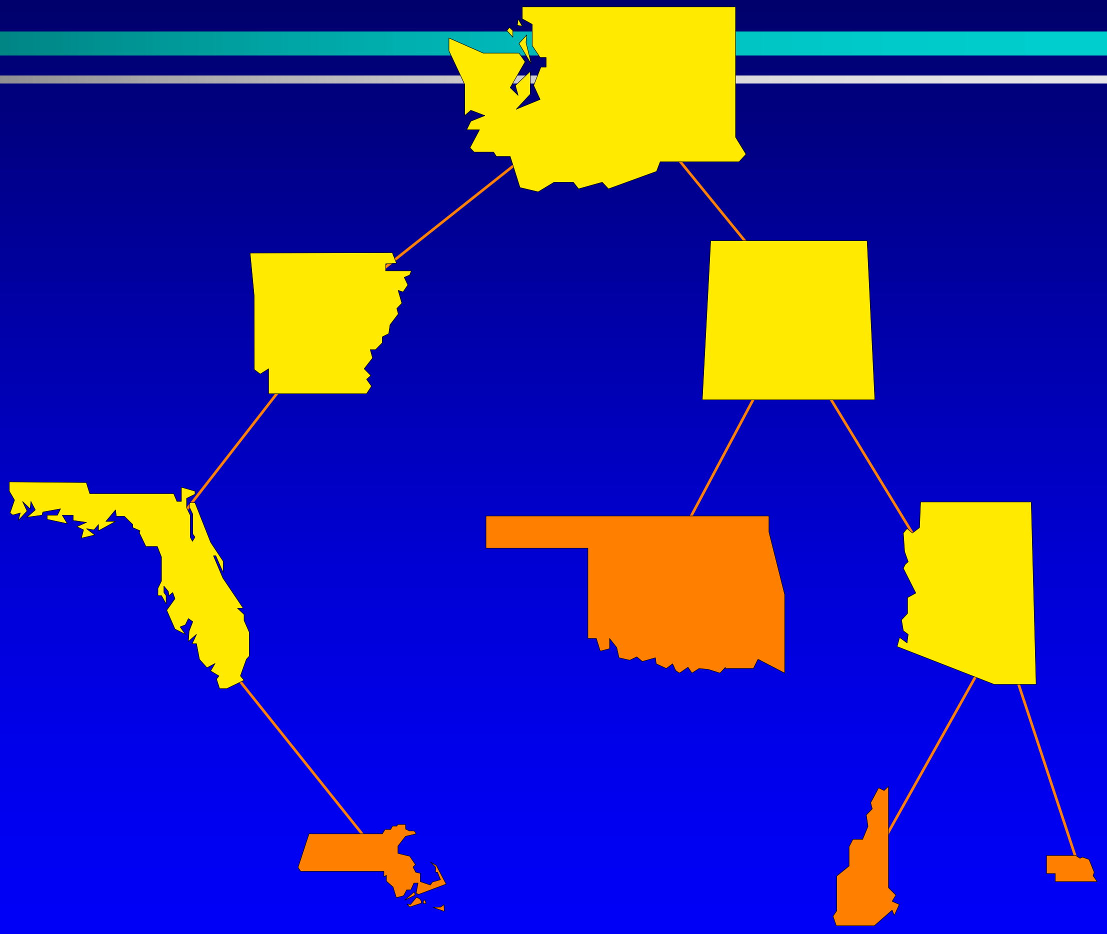
Some nodes have only one child.



Florida has
only a right child.

A Binary Tree of States

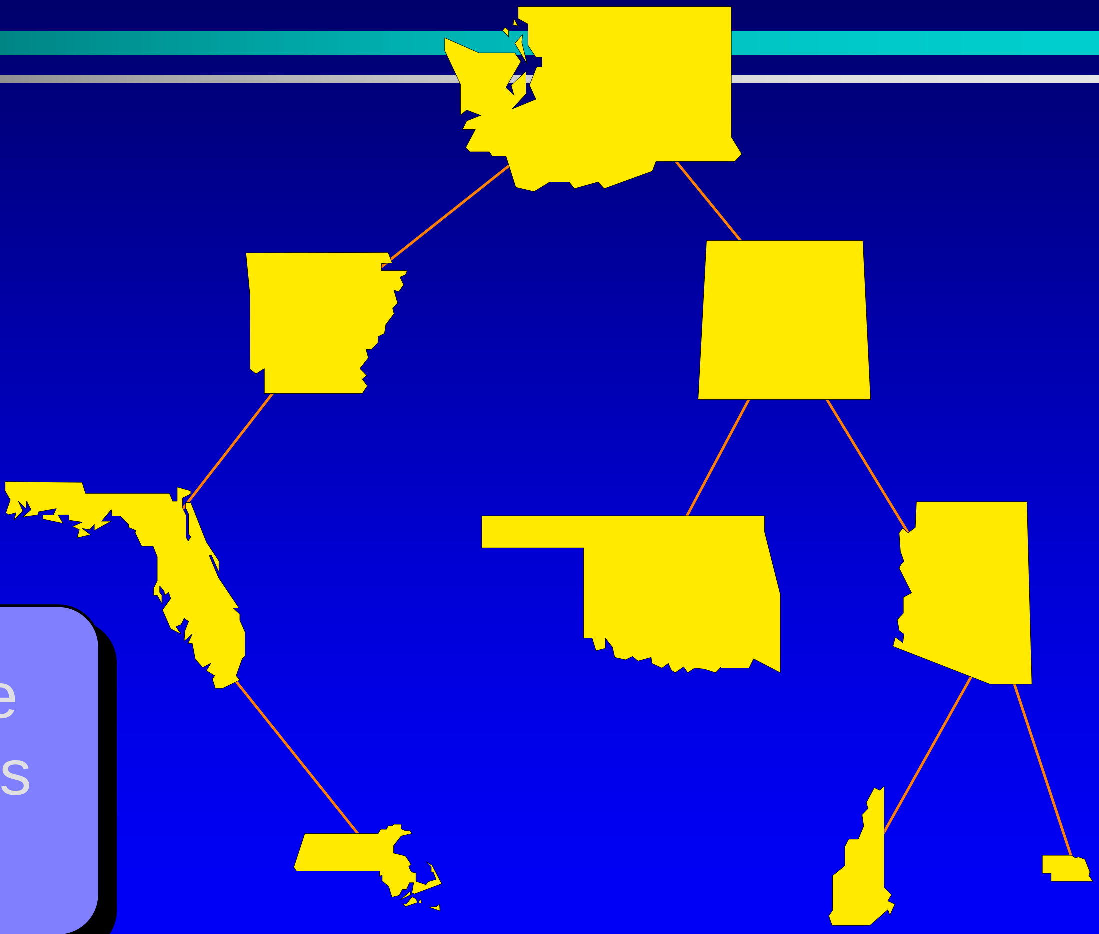
A node with no children is called a **leaf**.



A Binary Tree of States

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

Washington is the parent of Arkansas and Colorado.



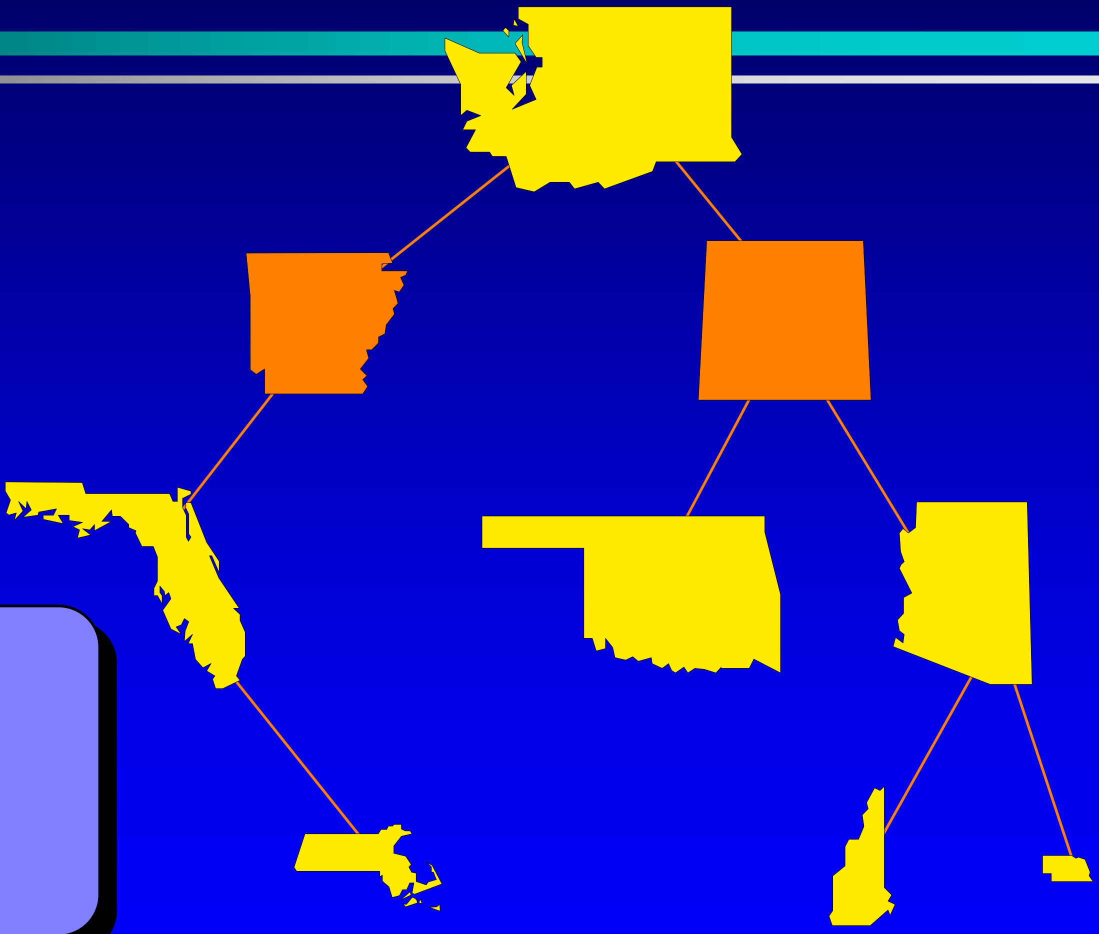
A Binary Tree of States

Two rules about parents:

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

A Binary Tree of States

Two nodes with the same parent are called siblings.



Arkansas
and Colorado
are siblings.

Complete Binary Trees

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

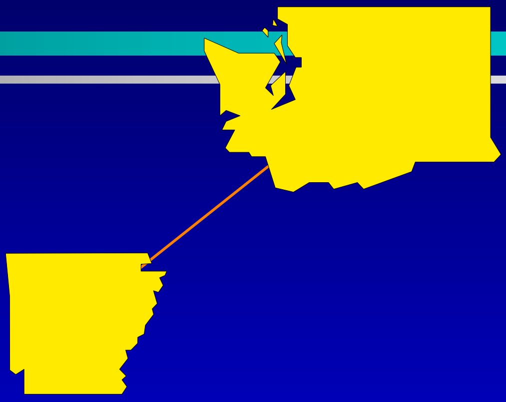
Complete Binary Trees

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

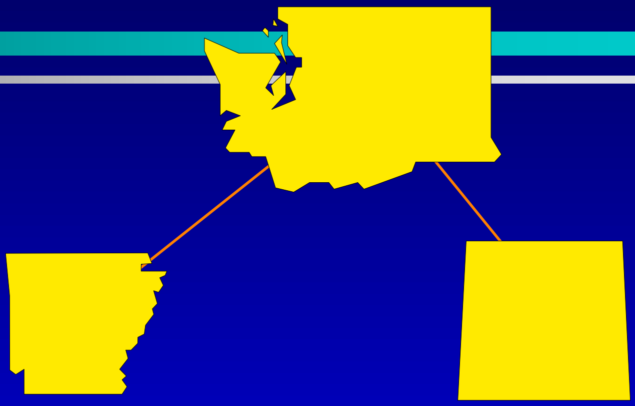
When a complete binary tree is built, its first node must be the root.

Complete Binary Trees

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



Complete Binary Trees

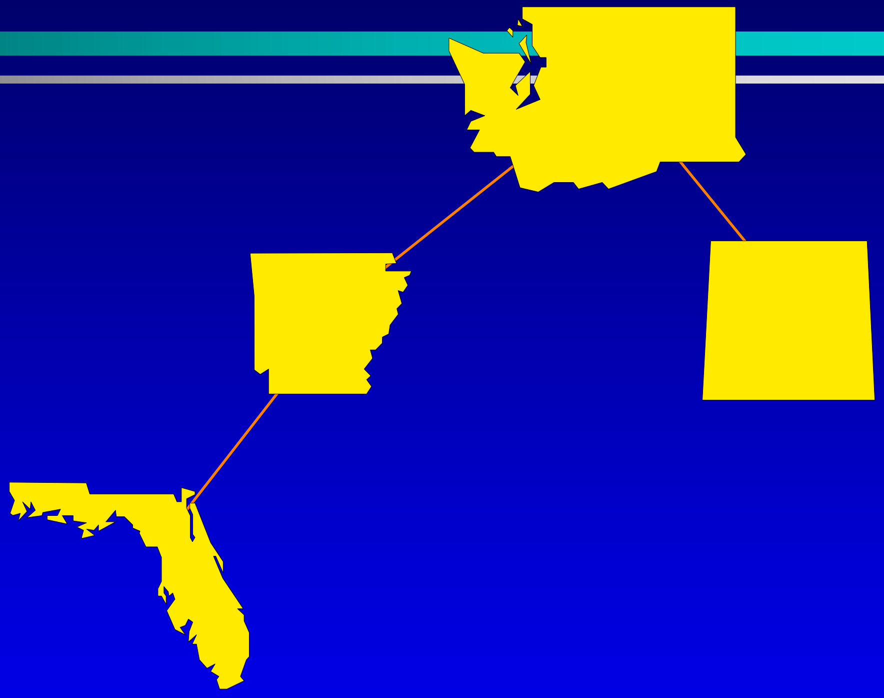


The second node of a complete binary tree is always the left child of the root...

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

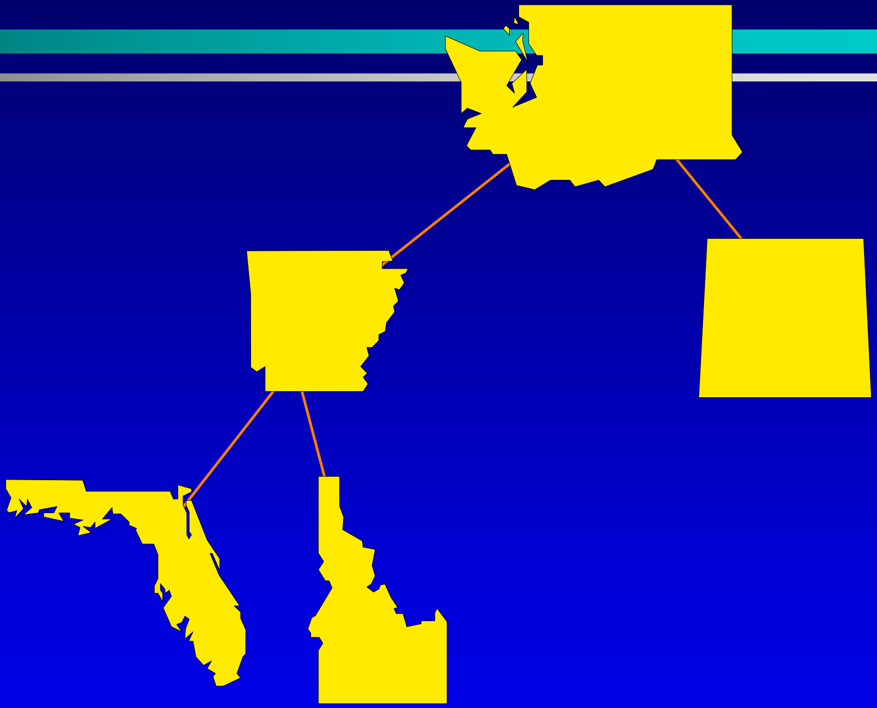
Complete Binary Trees

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



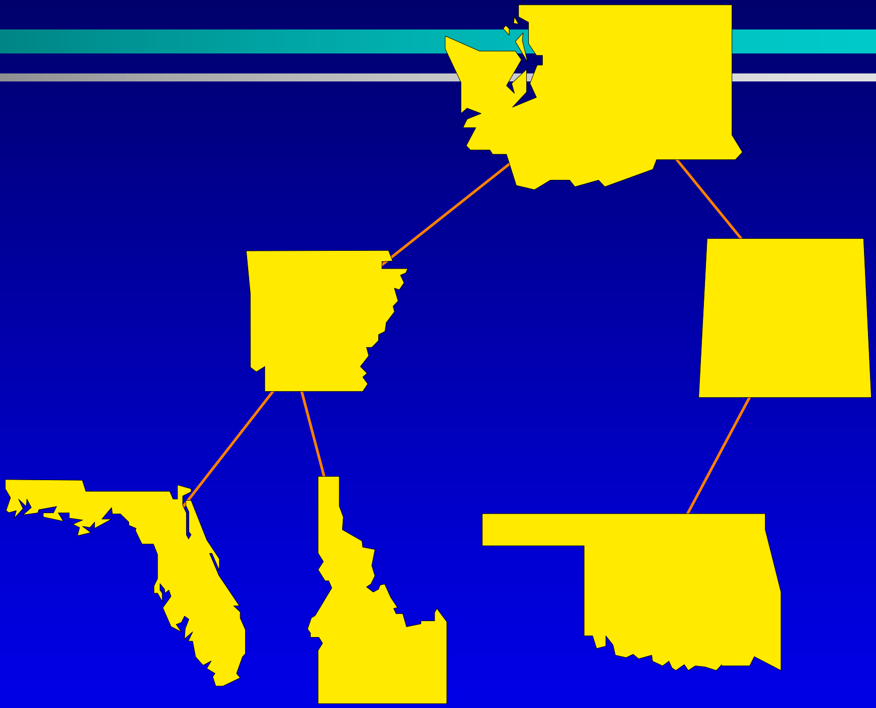
Complete Binary Trees

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



Complete Binary Trees

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

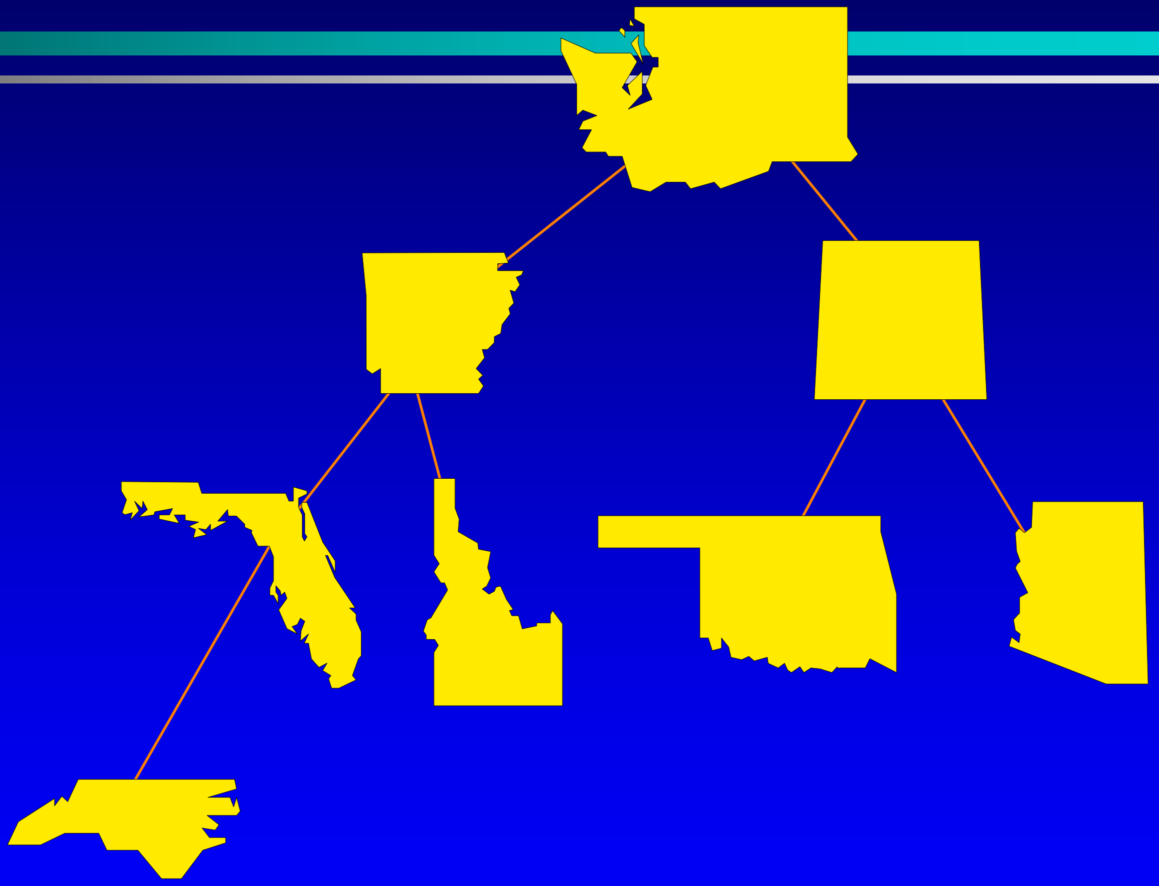


Complete Binary Trees

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

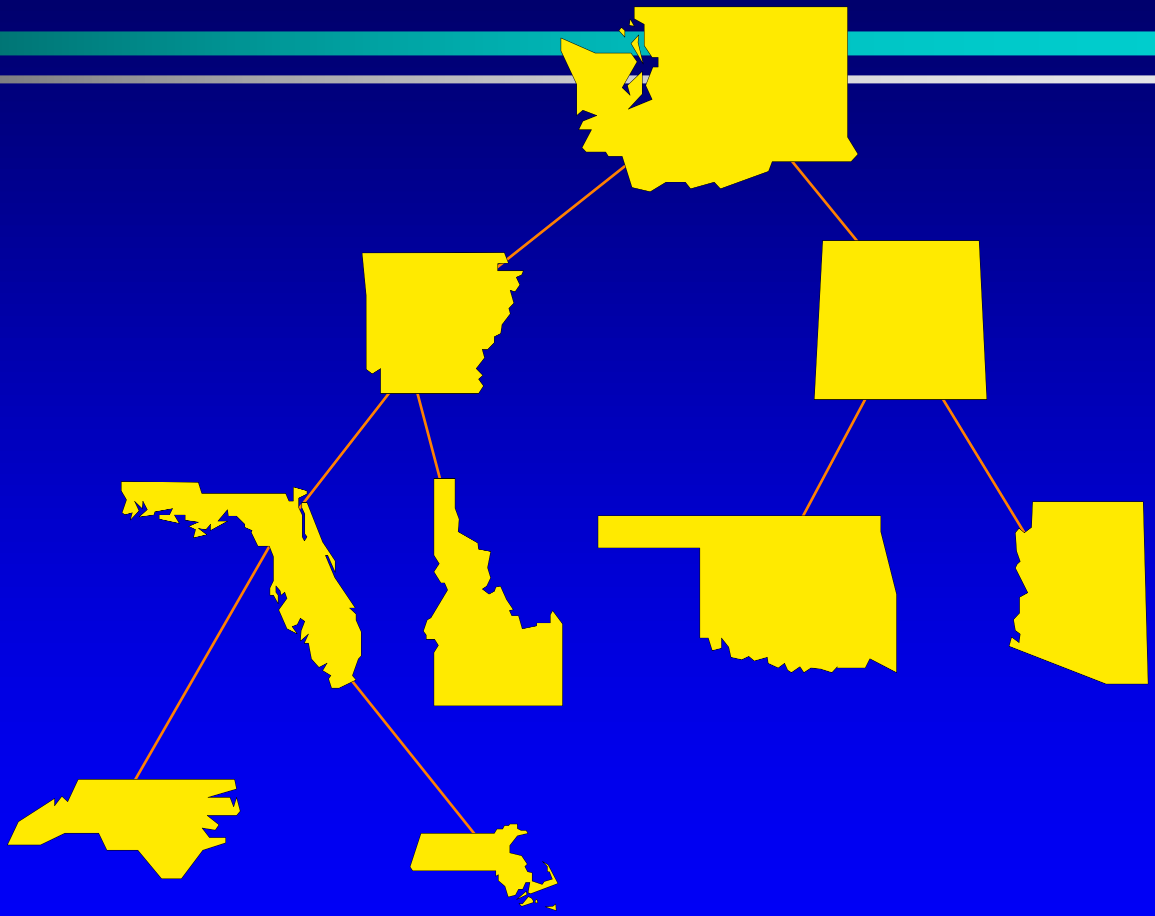
Complete Binary Trees

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

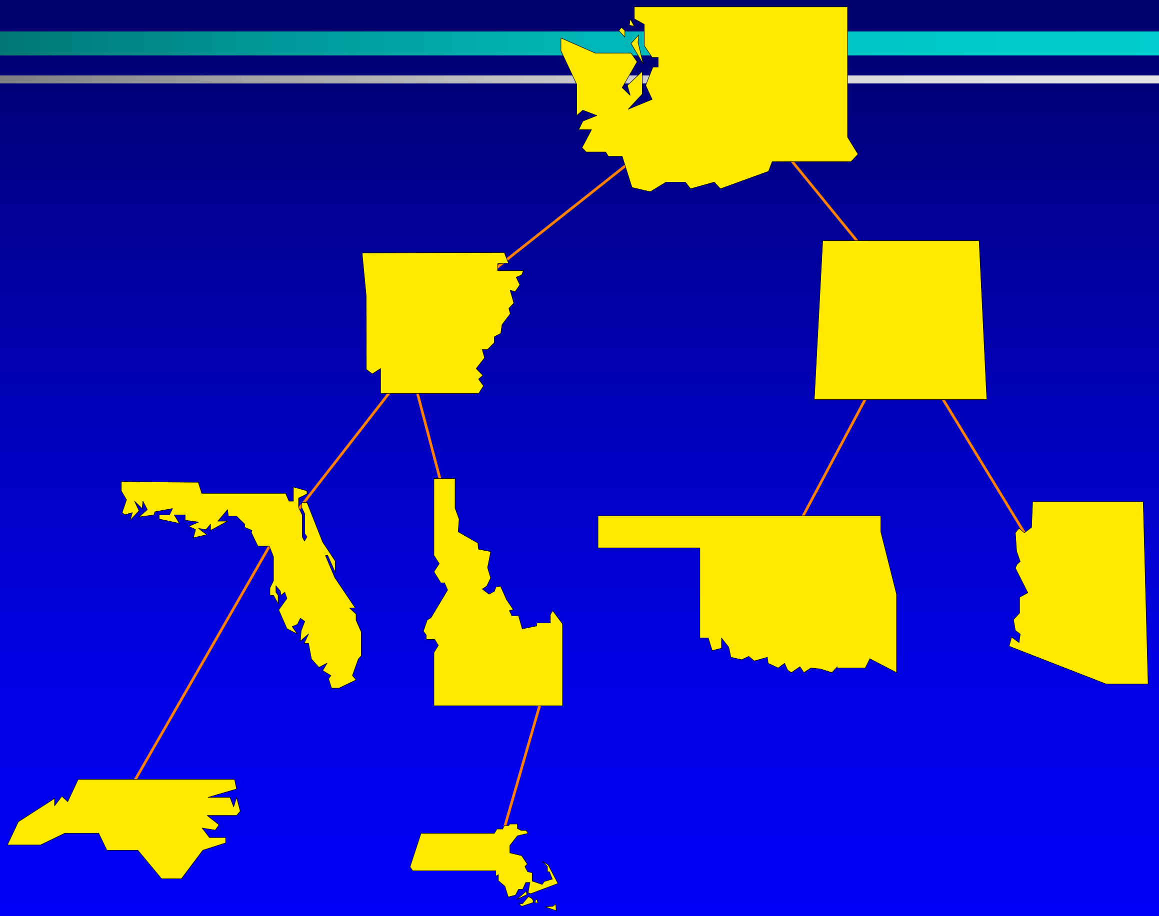


Complete Binary Trees

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

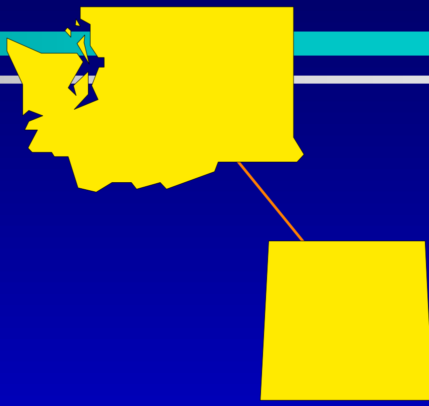


Is This Complete?



Is This Complete?

Is This Complete?



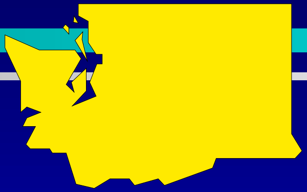
Is This Complete?

Is This Complete?

Yes!

- It is called the empty tree, and it has no nodes, not even a root.

Full Binary Trees



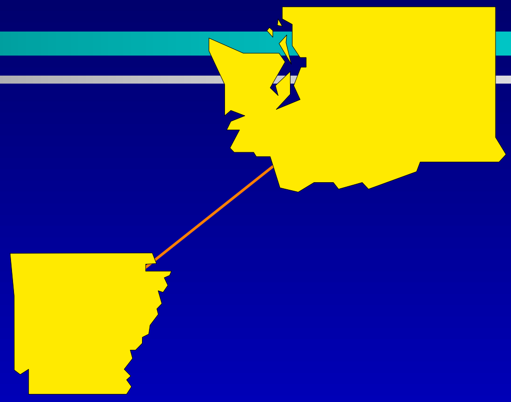
A full binary tree is a special kind of complete binary tree

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...

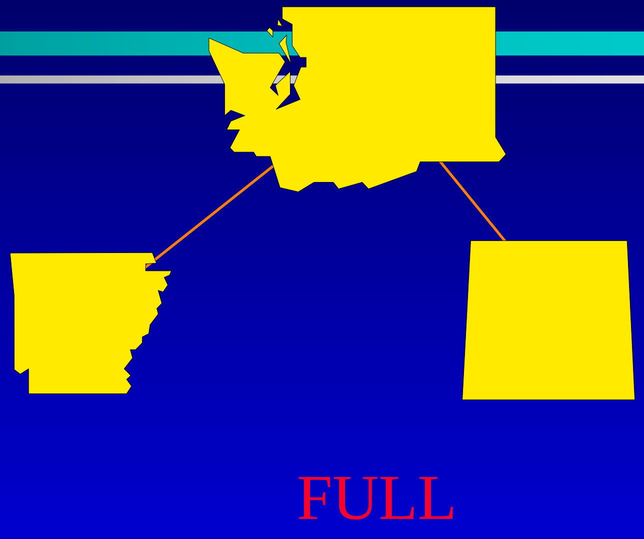


not FULL yet

Full Binary Trees

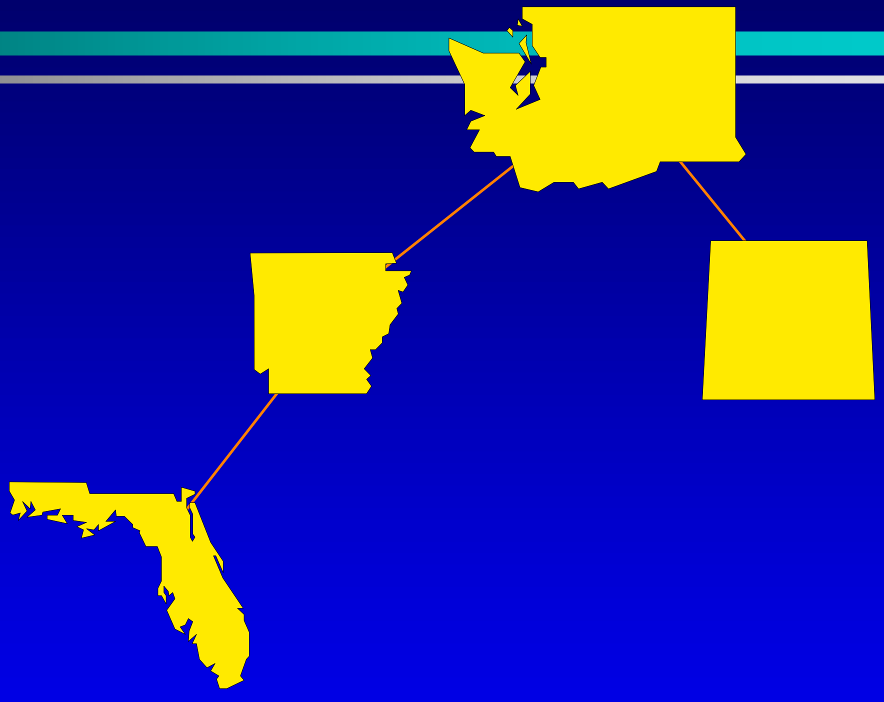
The second node of a full binary tree is always the left child of the root...

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



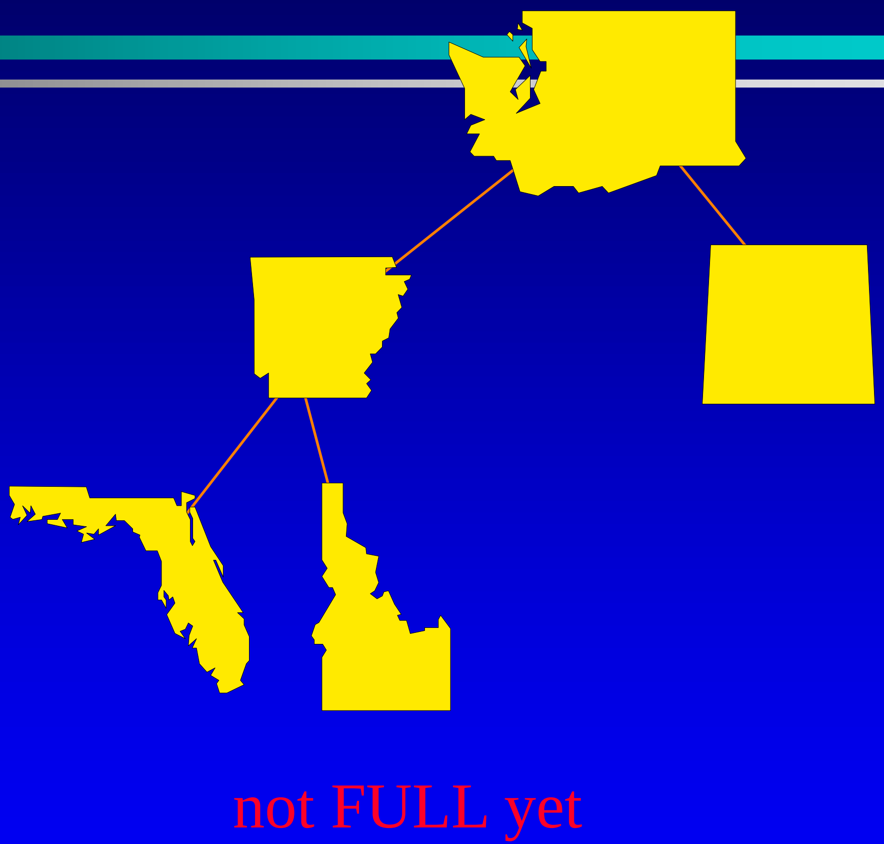
Full Binary Trees

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



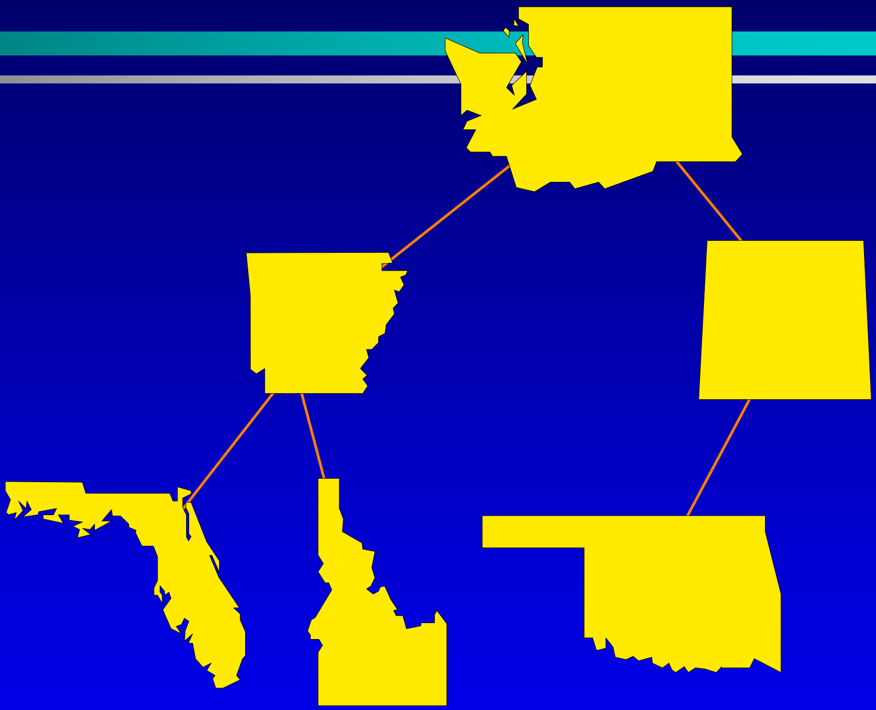
Full Binary Trees

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



Full Binary Trees

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



not FULL yet

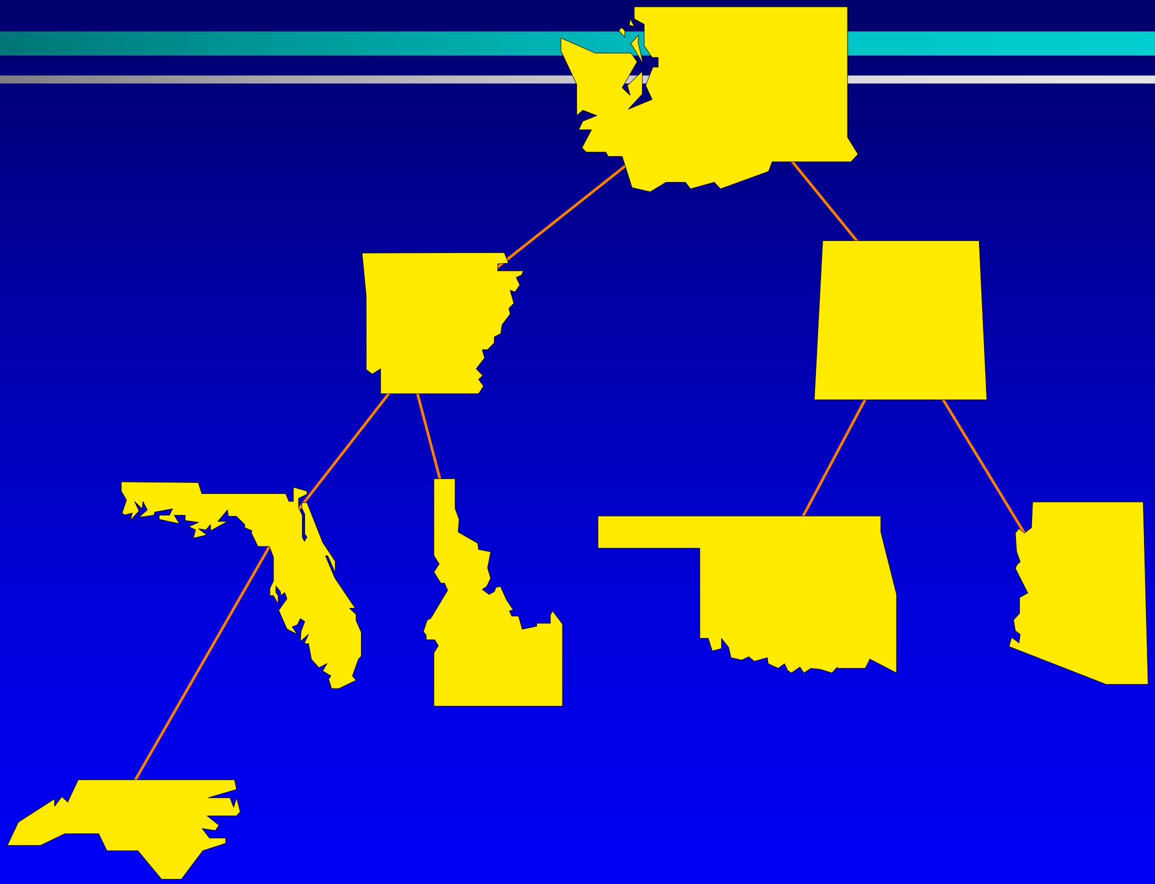
Full Binary Trees

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

FULL!

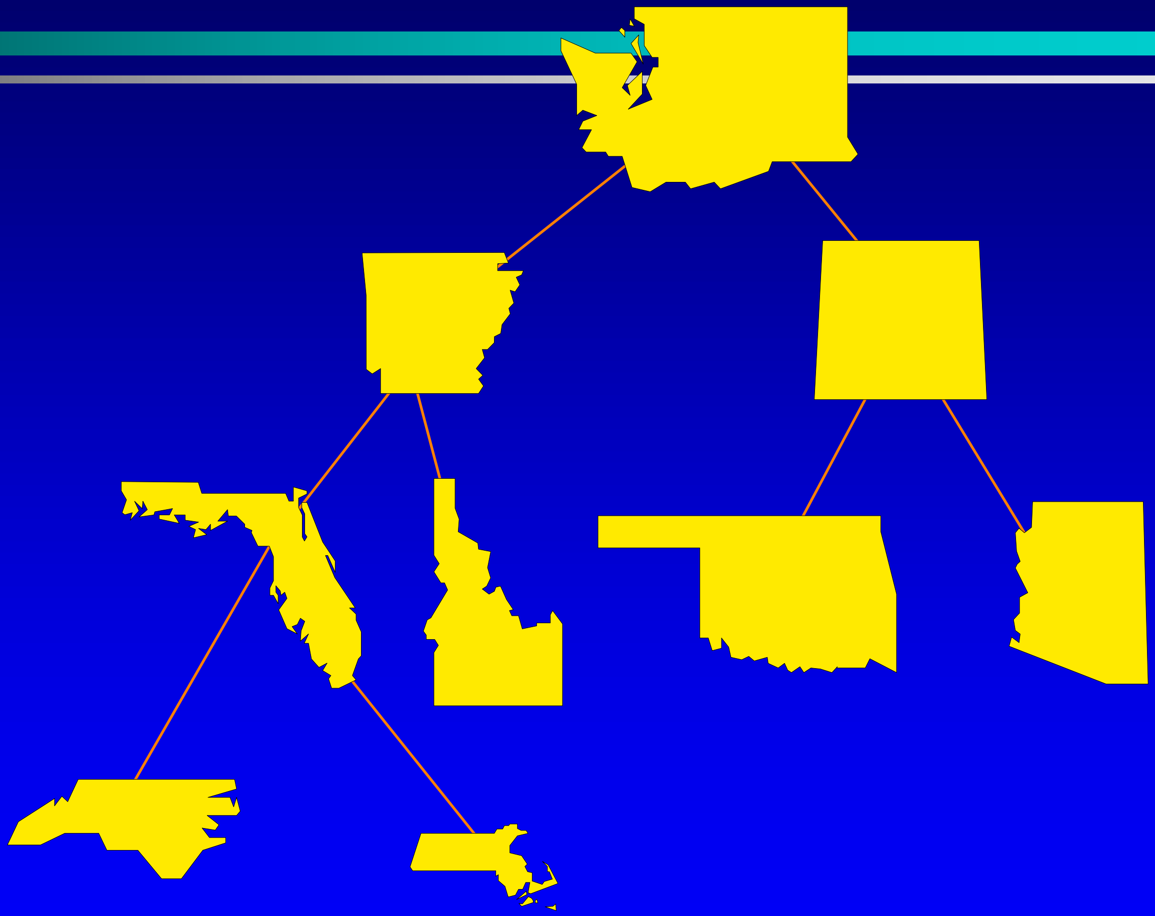
Full Binary Trees

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

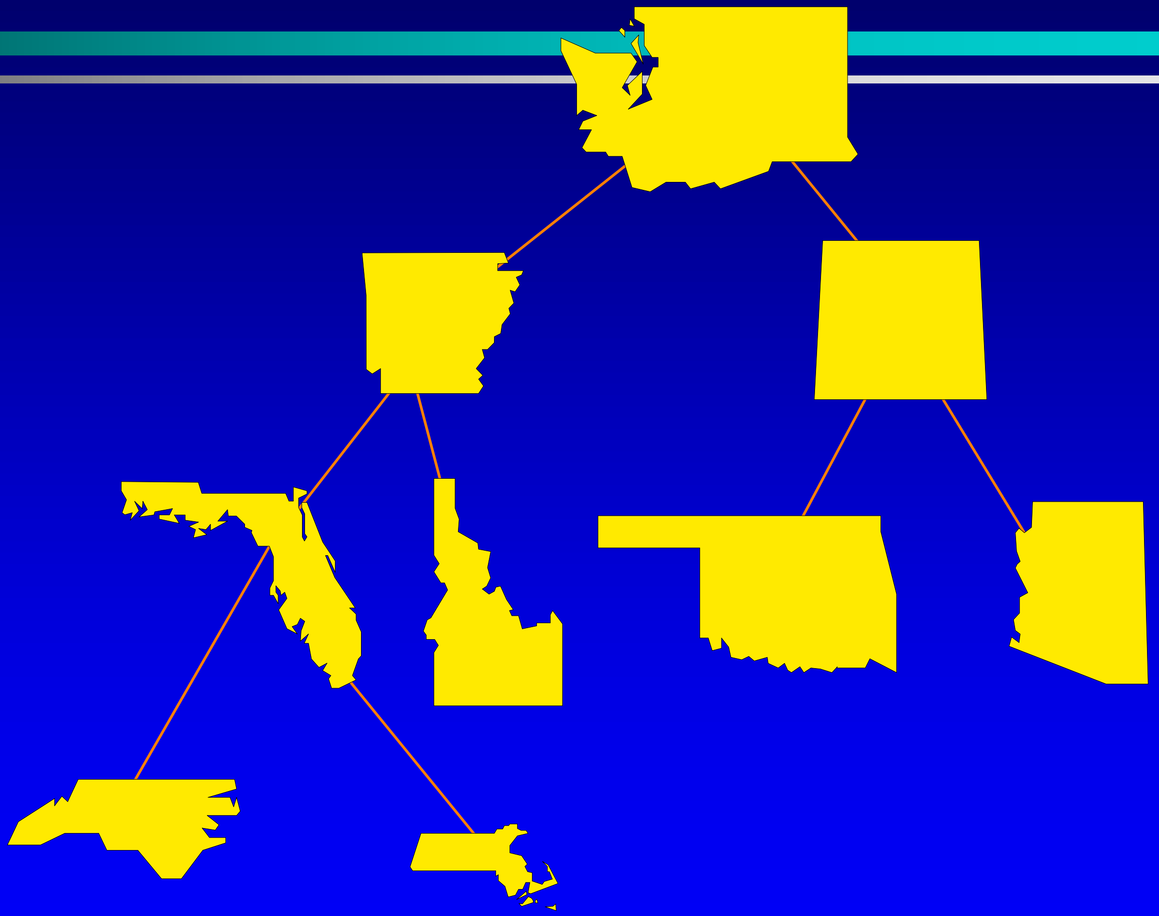


Full Binary Trees

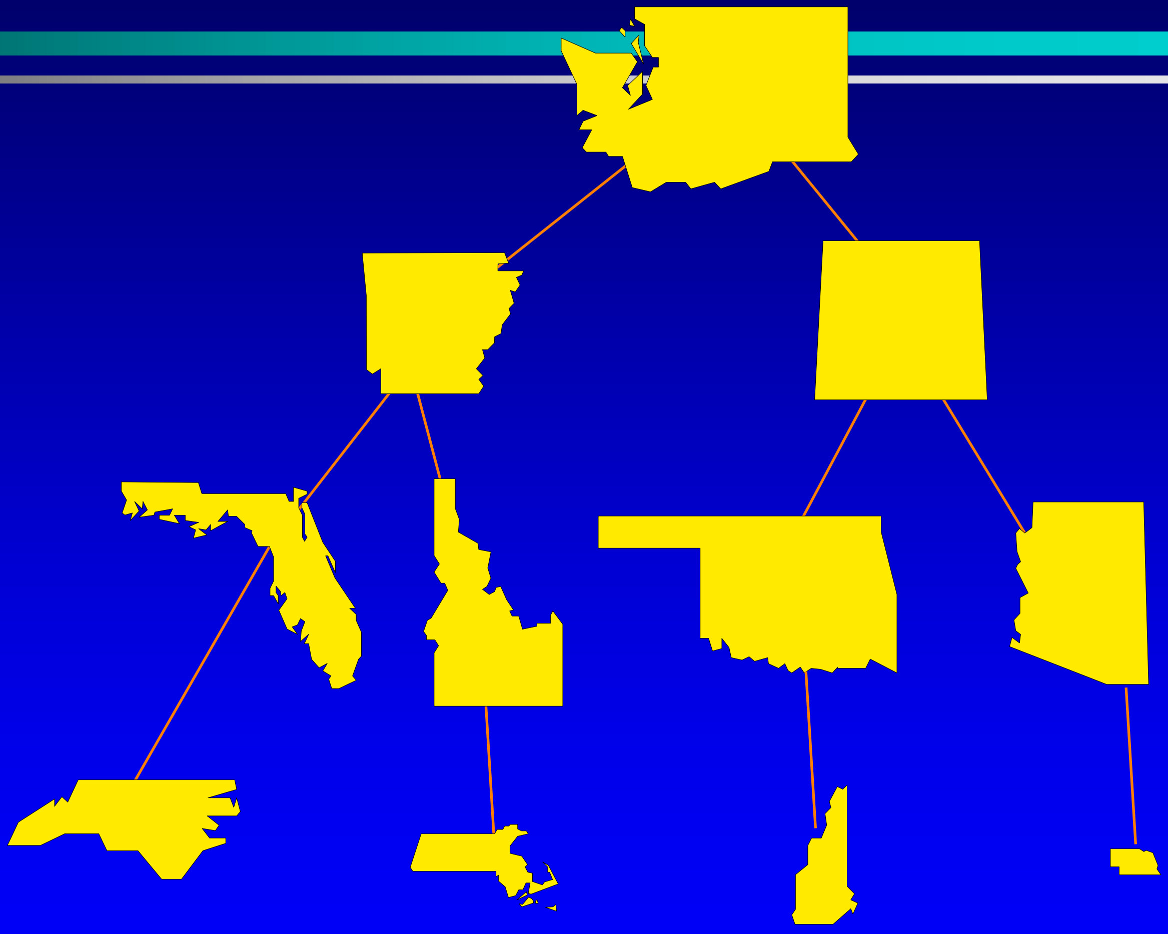
The next nodes must always fill the next level from **left to right**.



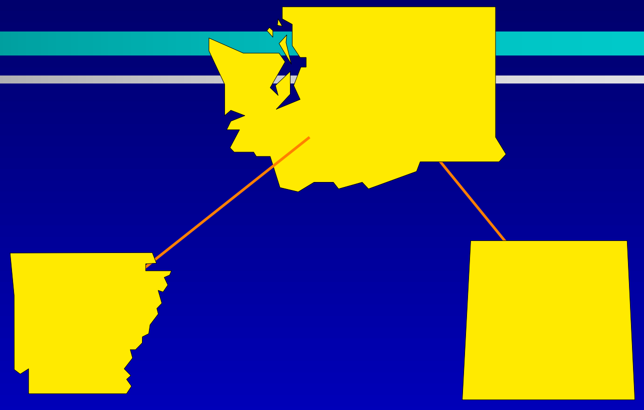
Is This Full?



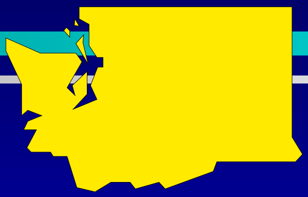
Is This Full?



Is This Full?



Is This Full?



Is This Full?

Yes!

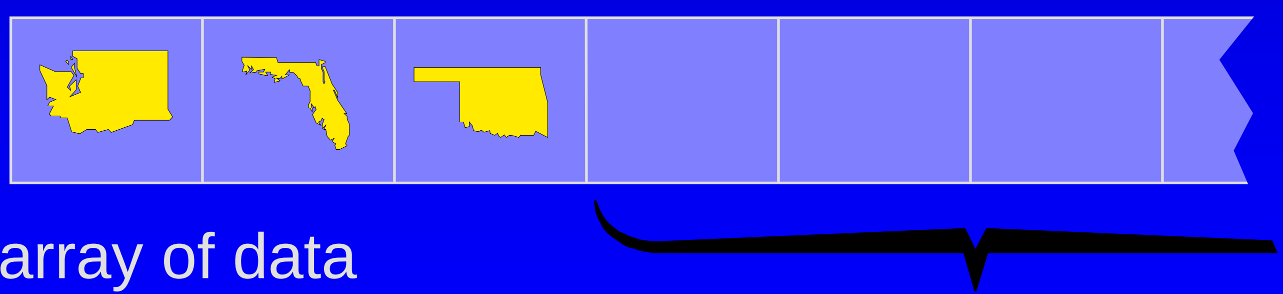
- It is called the empty tree, and it has no nodes, not even a root.

Implementing a Complete Binary Tree

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

3

An integer to keep track of how many nodes are in the tree



An array of data

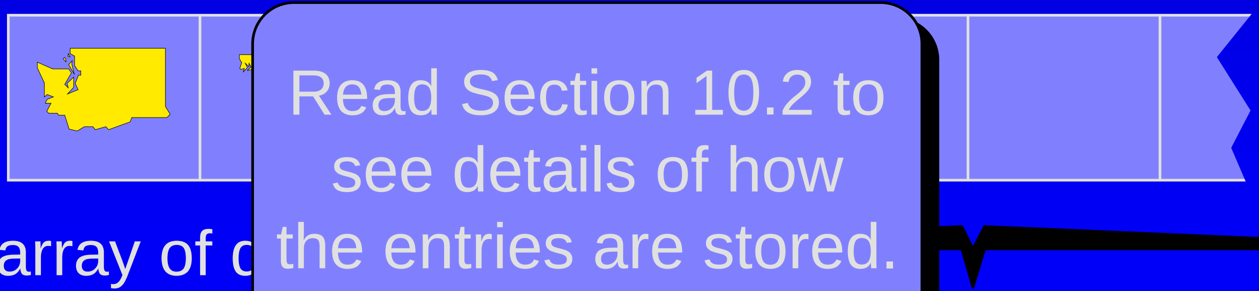
We don't care what's in this part of the array.

Implementing a Complete Binary Tree Using an Array

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

3

An integer to keep track of how many nodes are in the tree



Read Section 10.2 to see details of how the entries are stored.

An array of

care what's in this part of the array.

Implementing a Complete Binary Tree Using an Array

- Root is at component [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

Binary Tree Summary

- Binary trees contain nodes.
- Each node may have a left child and a right child.
- If you start from any node and move upward, you will eventually reach the root.
- Every node except the root has one parent. The root has no parent.
- Complete binary trees require the nodes to fill in each level from left-to-right before starting the next level.

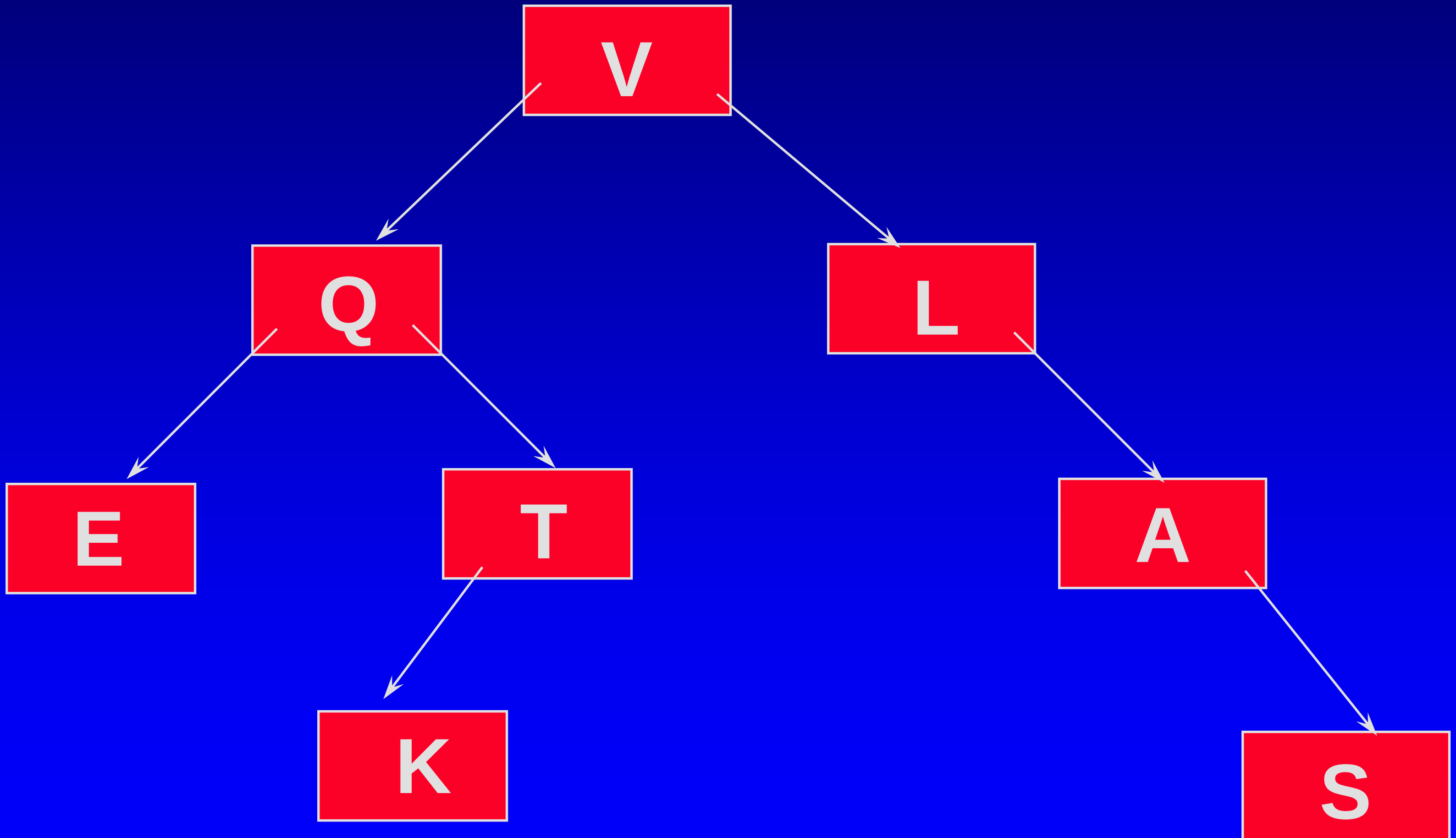
Binary Tree Basics

A binary tree is a structure in which:

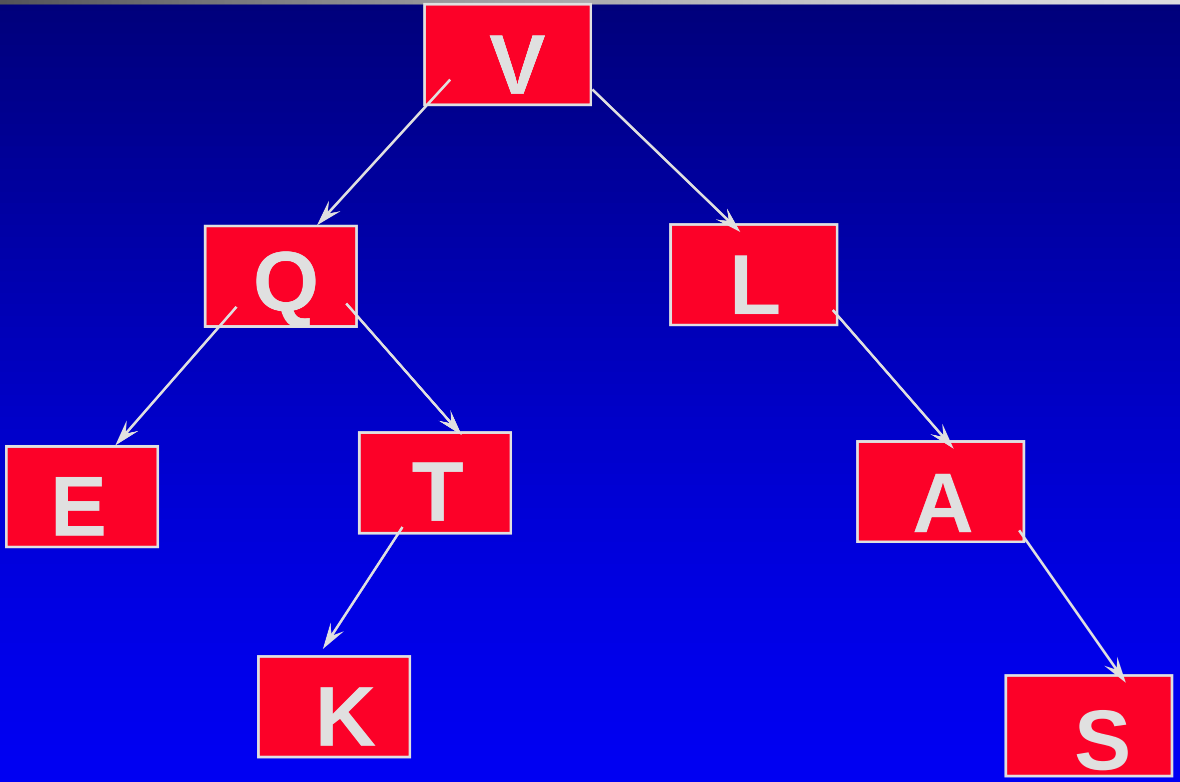
Each node can have at most two children, and in which a unique path exists from the root to every other node.

The two children of a node are called the **left child** and the **right child,** if they exist.

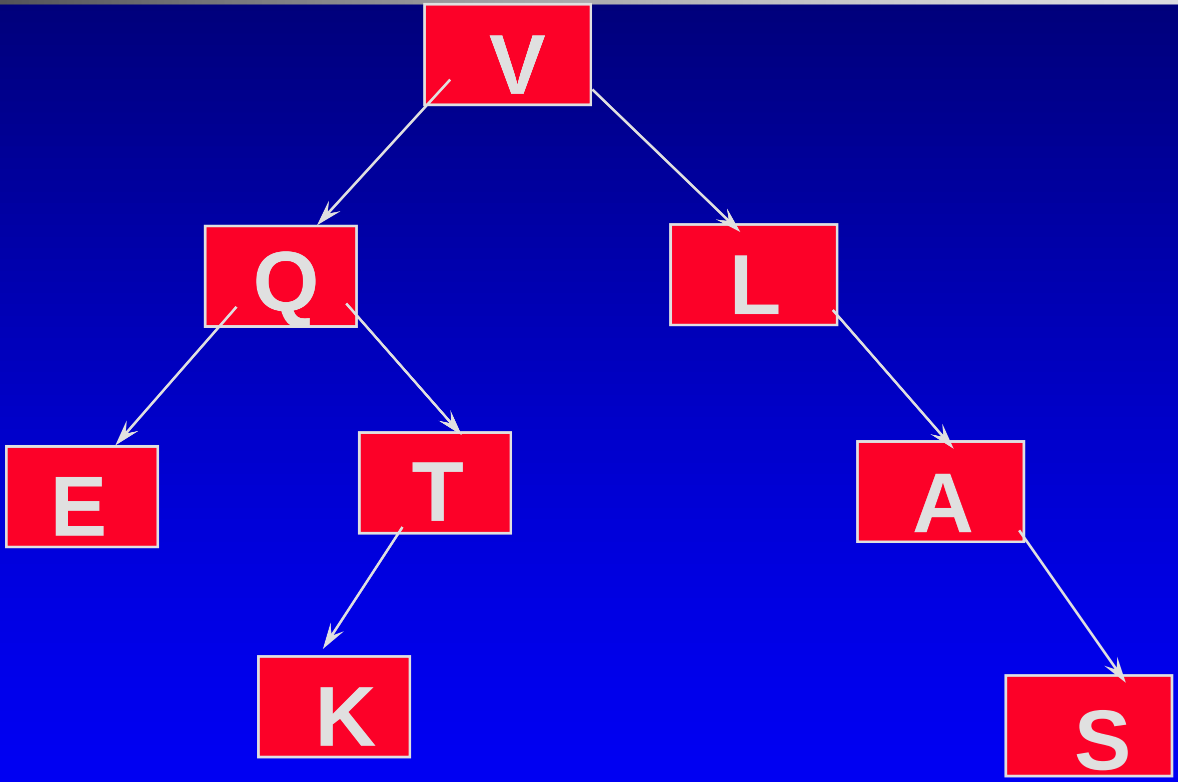
A Binary Tree Exercise



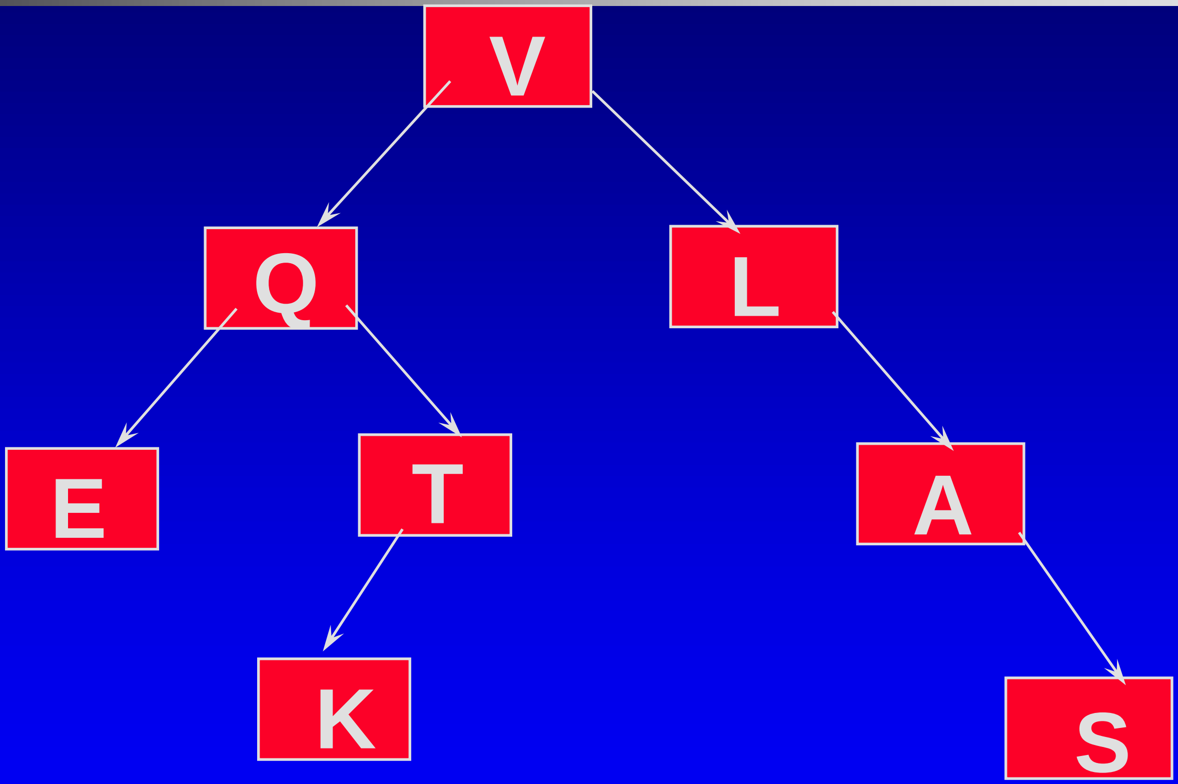
How many leaf nodes?



How many descendants of Q?

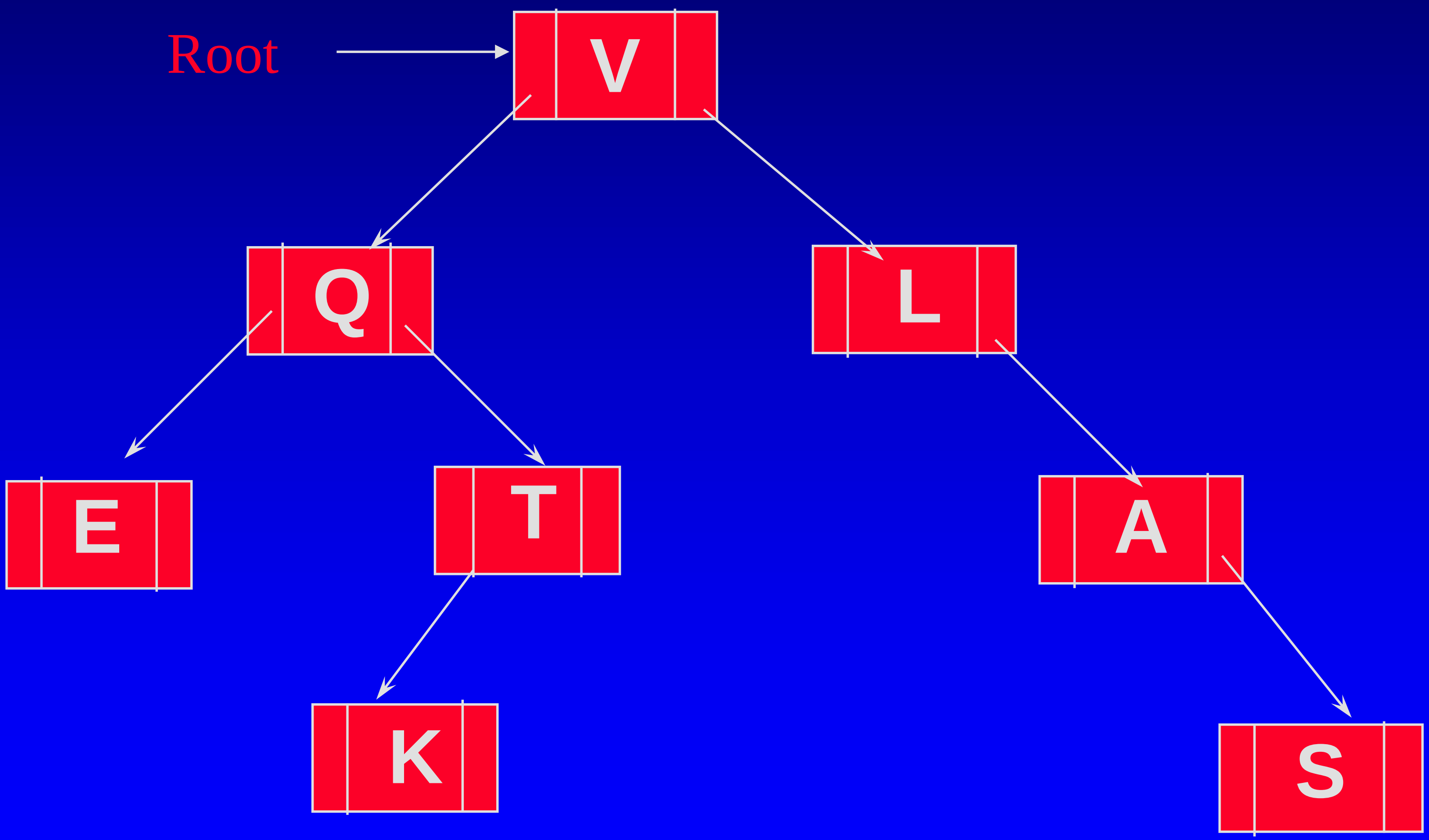


How many ancestors of K?



Question: How to implement a general binary tree ?

Implementing a Binary Tree 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 that we are going to define
- Each node contains data as well as pointers to its children (nodes)
- An entire tree is represented as a pointer to the root node

binary_tree_node Class

bintree

- variables
- functions

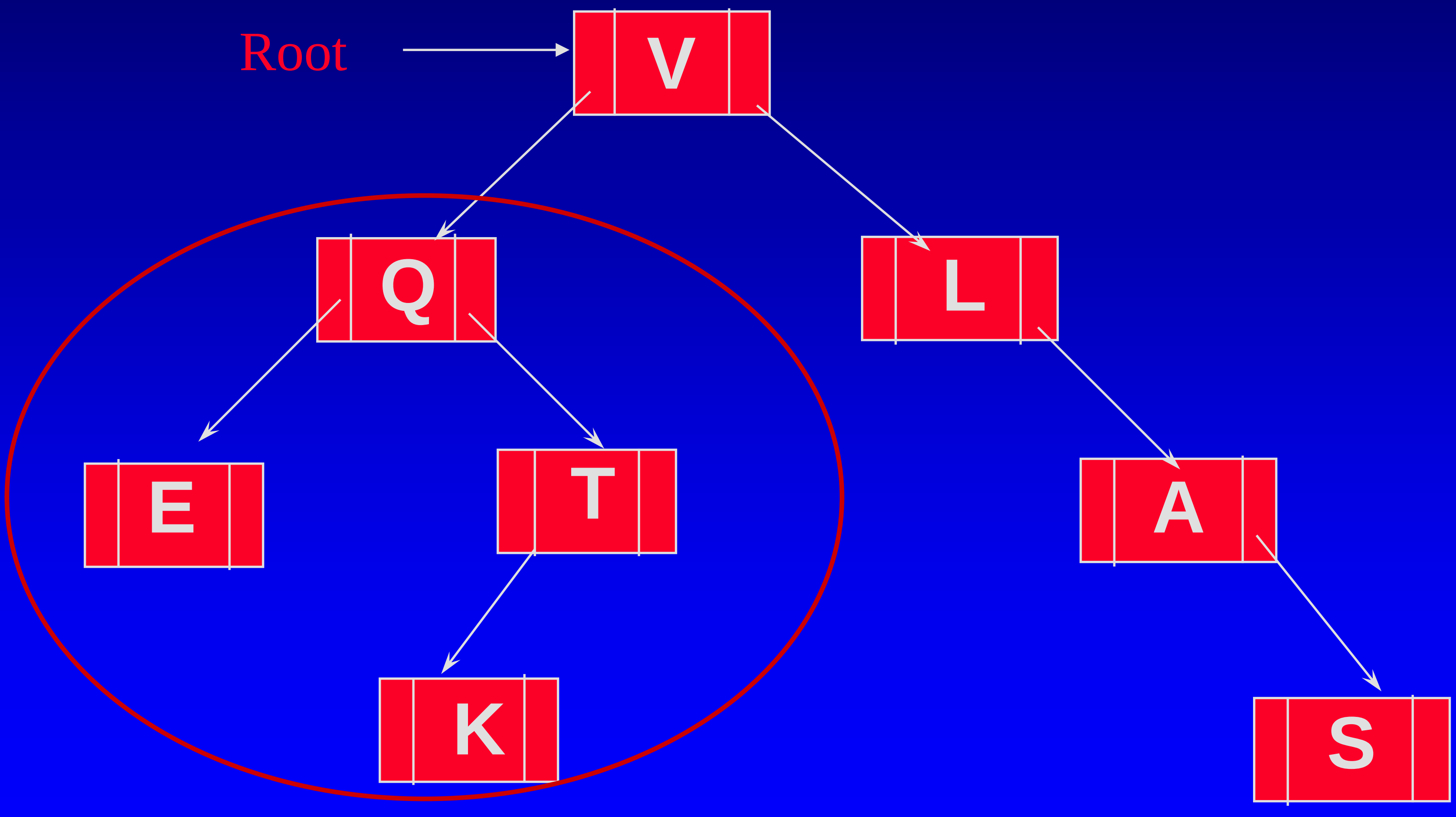
```
template <class Item>
  class binary_tree_node
  {
  public:
          ......
  private:
          Item data_field;
          binary_tree_node *left_field;
          binary_tree_node *right_field;
  };
```

//retrievals
data
left
right
//set
set_data
set_left
set_right
//boolean
is_leaf

Creating and Manipulating Trees

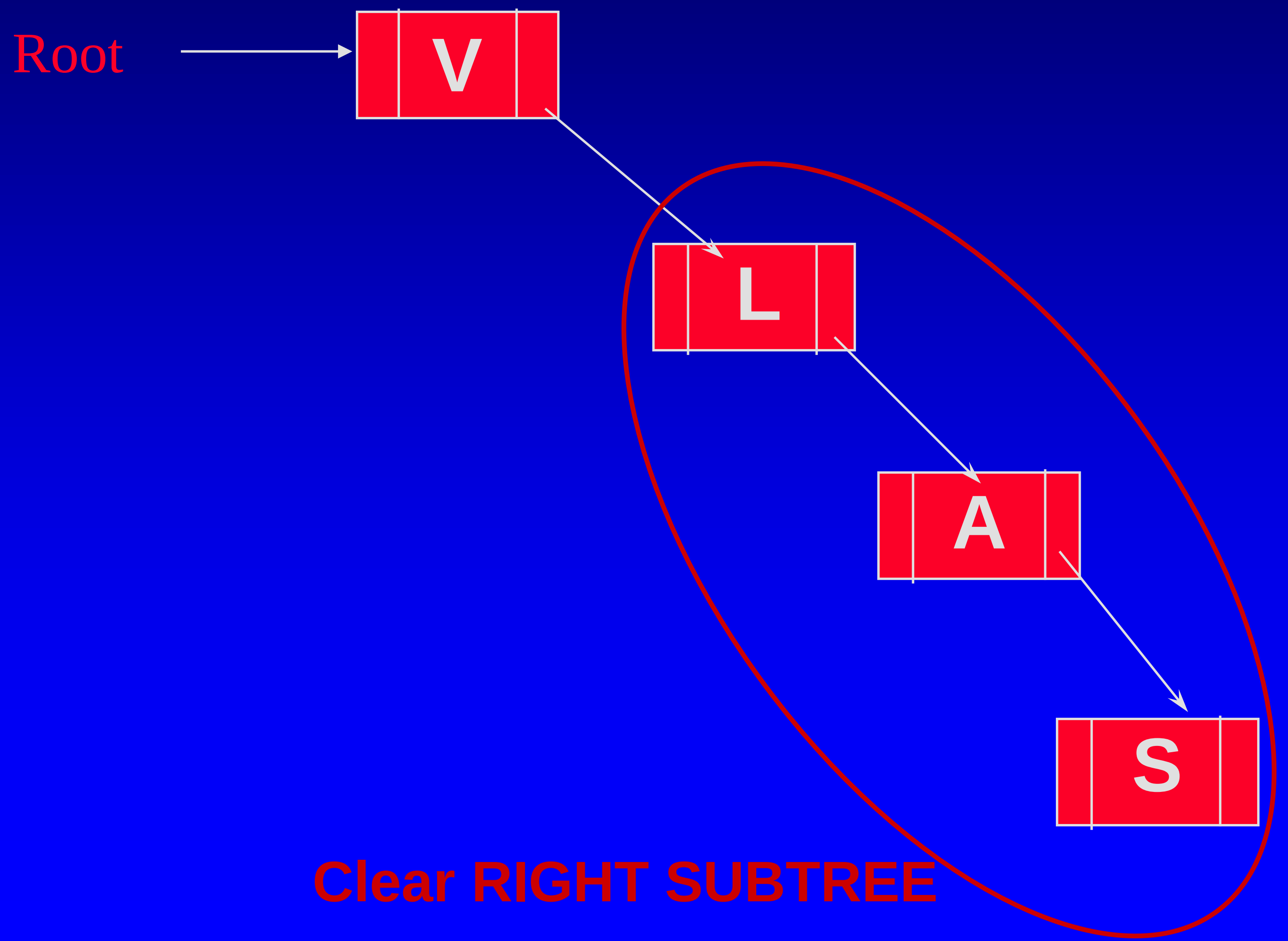
- Consider only two functions
  - Clearing a tree
    - Return nodes of a tree to the heap
  - Copying a tree
- The Implementation is easier than it seems
  - if we use recursive thinking

Clearing a Tree



Clear LEFT SUBTREE

Clearing a Tree



Clear RIGHT SUBTREE

Clearing a Tree

Return root node to the heap

Clearing a Tree

Root → NULL

Set the root pointer to NULL

Clear a Tree

bintree

- 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 sub_tree
               tree_clear( root_ptr->right( ) ); // clear right sub_tree
               delete root_ptr;   // return root node to the heap
               root_ptr = NULL; // set root pointer to the null
           }
  }
```

Copy a Tree

bintree

- Can you implement the copy? (p 467)

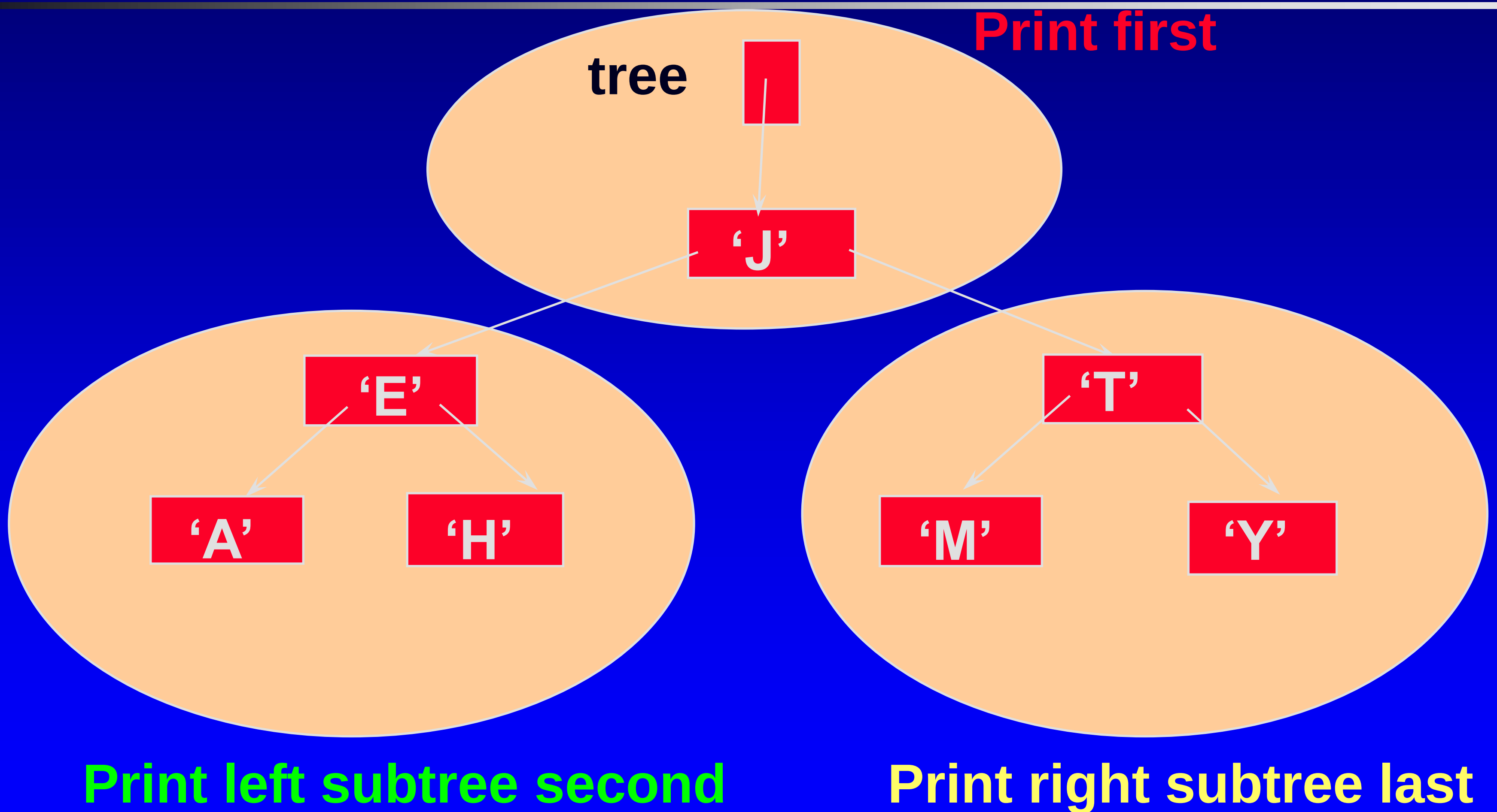
```
template <class Item>
  binary_tree_node<Item>* tree_copy(const binary_tree_node<Item>* root_ptr)
  // Library facilities used: cstdlib
  {
          binary_tree_node<Item> *l_ptr;
          binary_tree_node<Item> *r_ptr;

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

Binary Tree Traversals

bintree

- 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

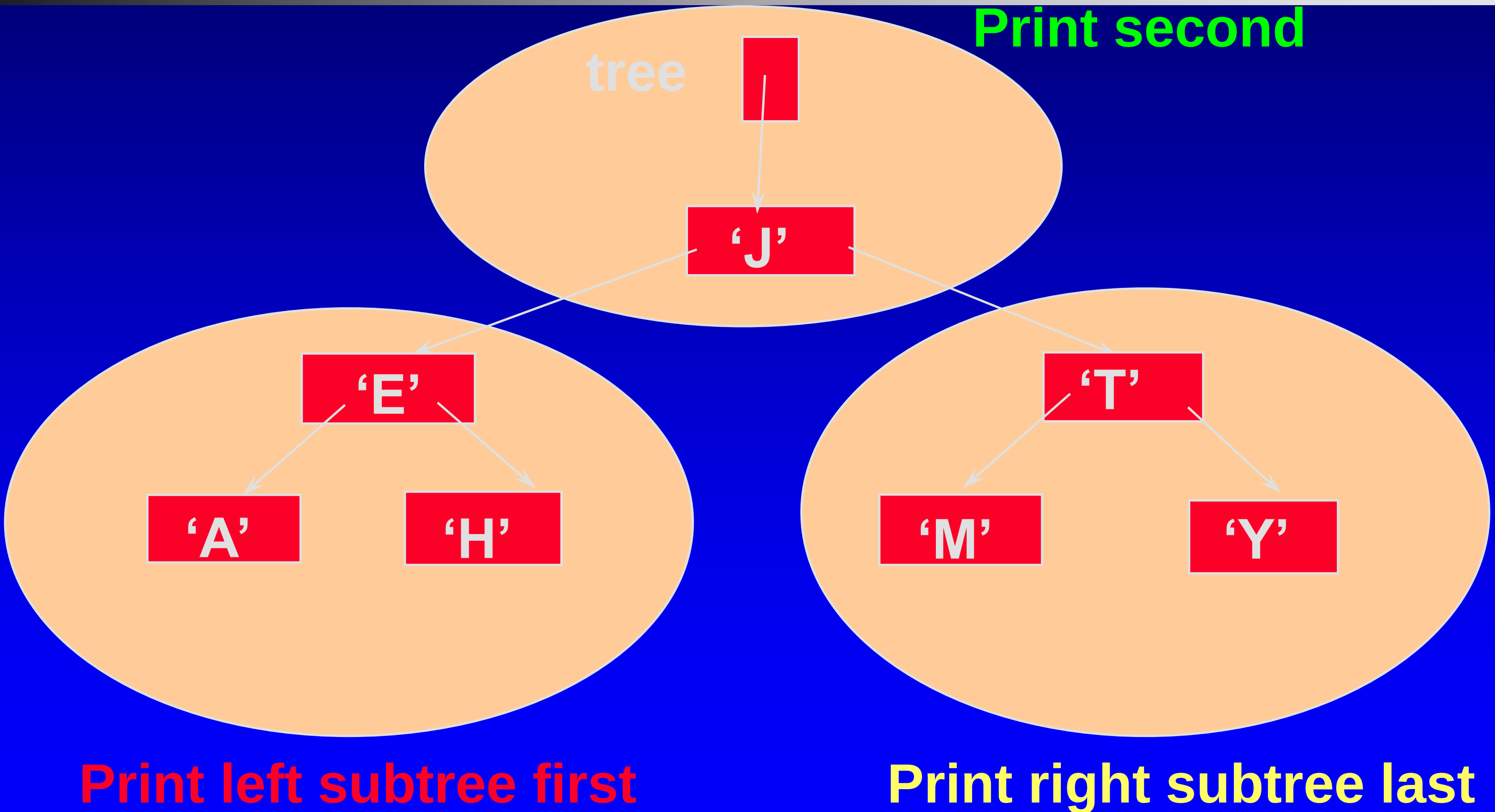


Preorder Traversal

- 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

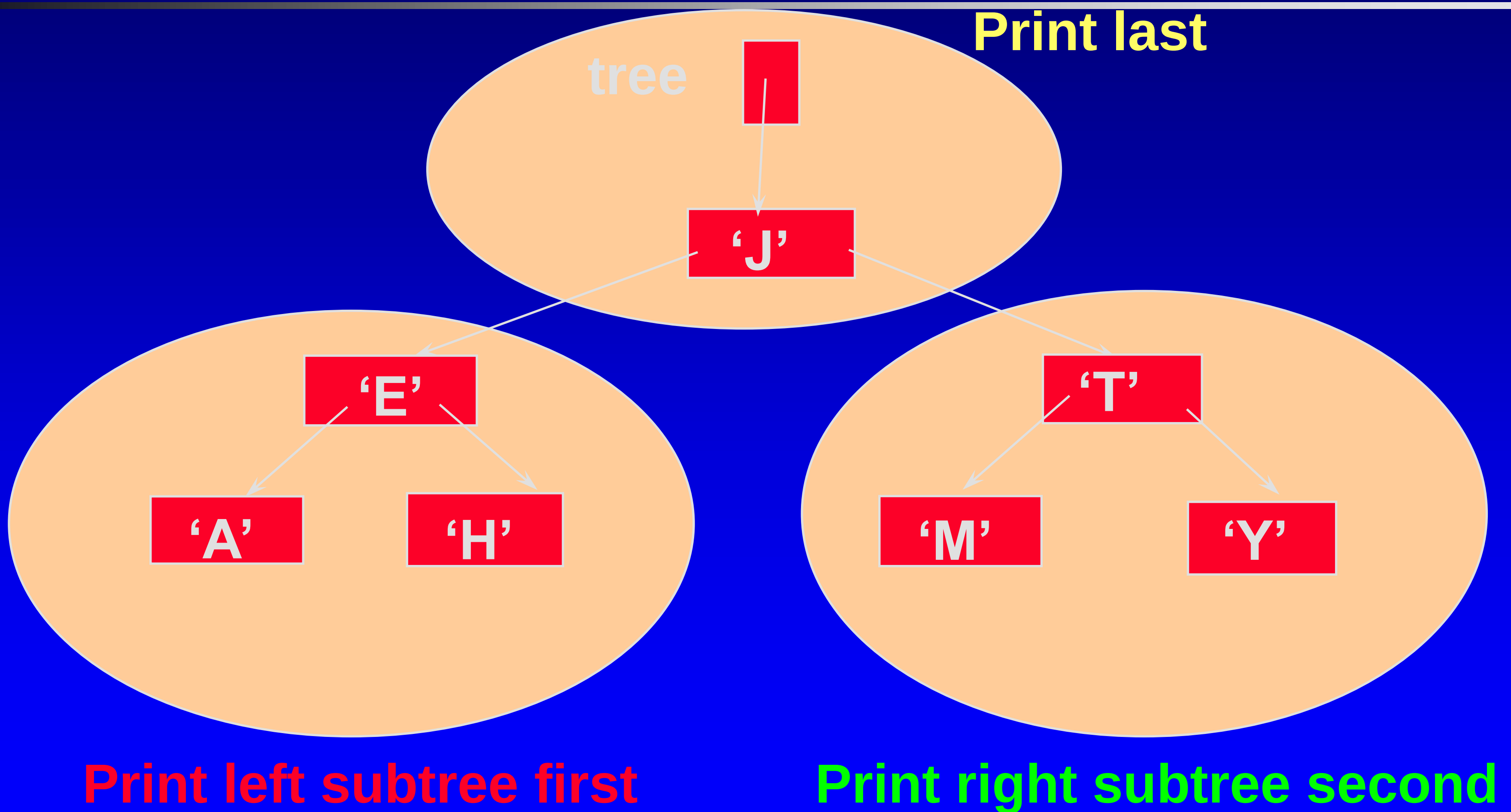


Inorder Traversal

- 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



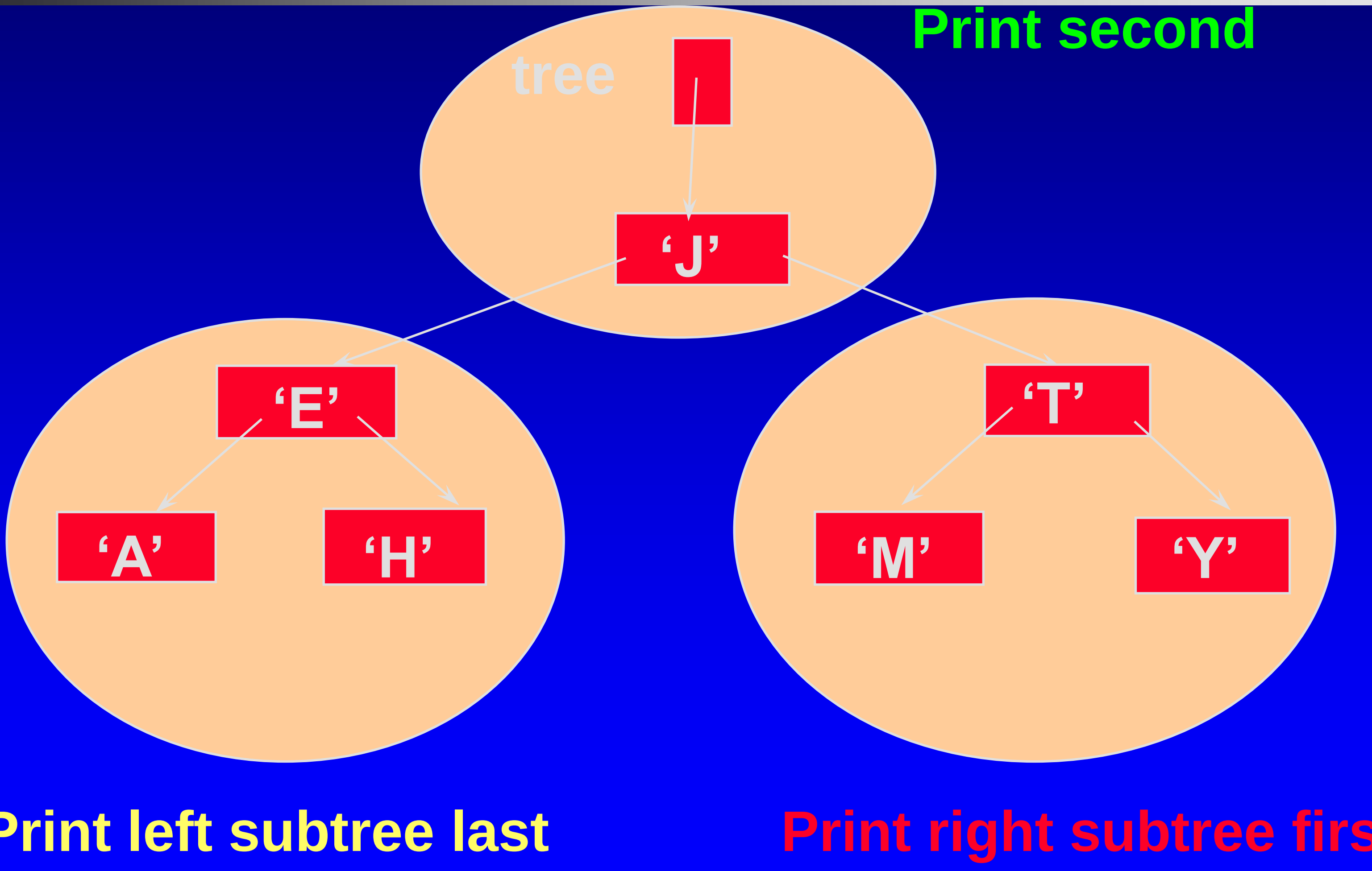
Postorder Traversal

- 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



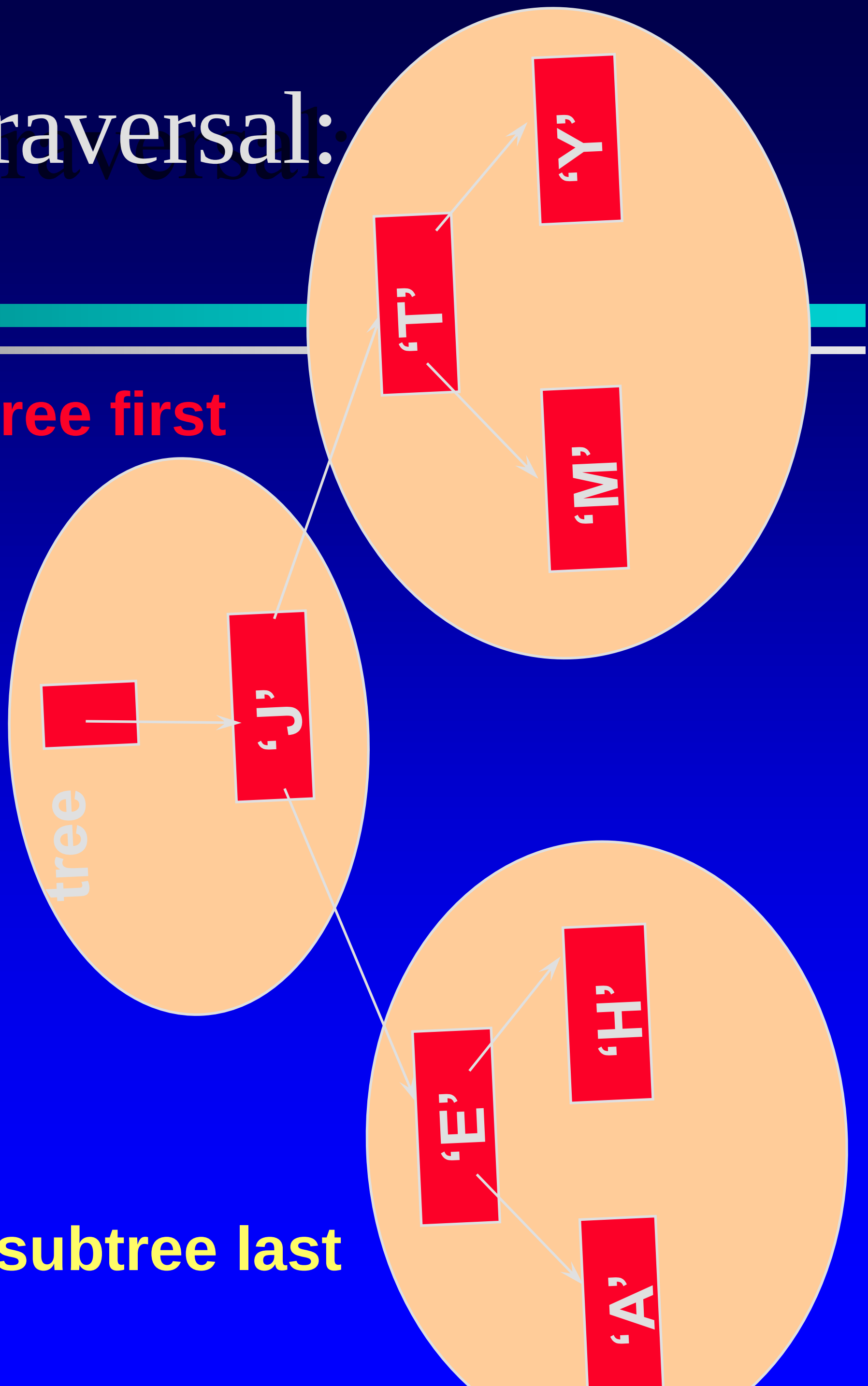
Backward Inorder Traversal:

Y T M J H E A

Print right subtree first

Print second

Print left subtree last



A Useful Backward Inorder Traversal

bintree

- Indent each number 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) << ""; // Indent 4*depth spaces.
           std::cout << node_ptr->data( ) << std::endl;
           print(node_ptr->left( ),  depth+1);
       }
   }
```

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?
  - Can template be helpful?
- Solution::::::::> (pages 501 – 507)

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)
       {
           f( node_ptr->data( ) ); // node_ptr->data() return reference !
           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);
```

Yes!!!

Preorder Traversal – another functions

- Can define other functions...

```
void assign_default(int& it)
   // Library facilities used: iostream
 {
            it  = 0;
 }  // unfortunately template does not work here for function parameters

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 – how to use

- Can the function-arguments be template?

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

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

binary_tree_node<string> *root;
....
preorder(print_out, root);
```

X ! print_out should have real types

Preorder Traversal – how to use

- The function-arguments may be template if...

```
template <class Item>
   void printout(Item& 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<string> *root;
....
preorder(print_out<string>, root);
```

But you may do the instantiation like this

Preorder Traversal
– a more general form

bintree

- An extremely general implementation (p 505)

```
   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( ));
      }
   }
```

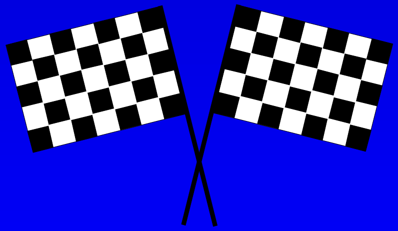
Functions as Parameters

- 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 real function argument for *f* when calling the 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

Copyright from slide 2 – slide 49:



THE END