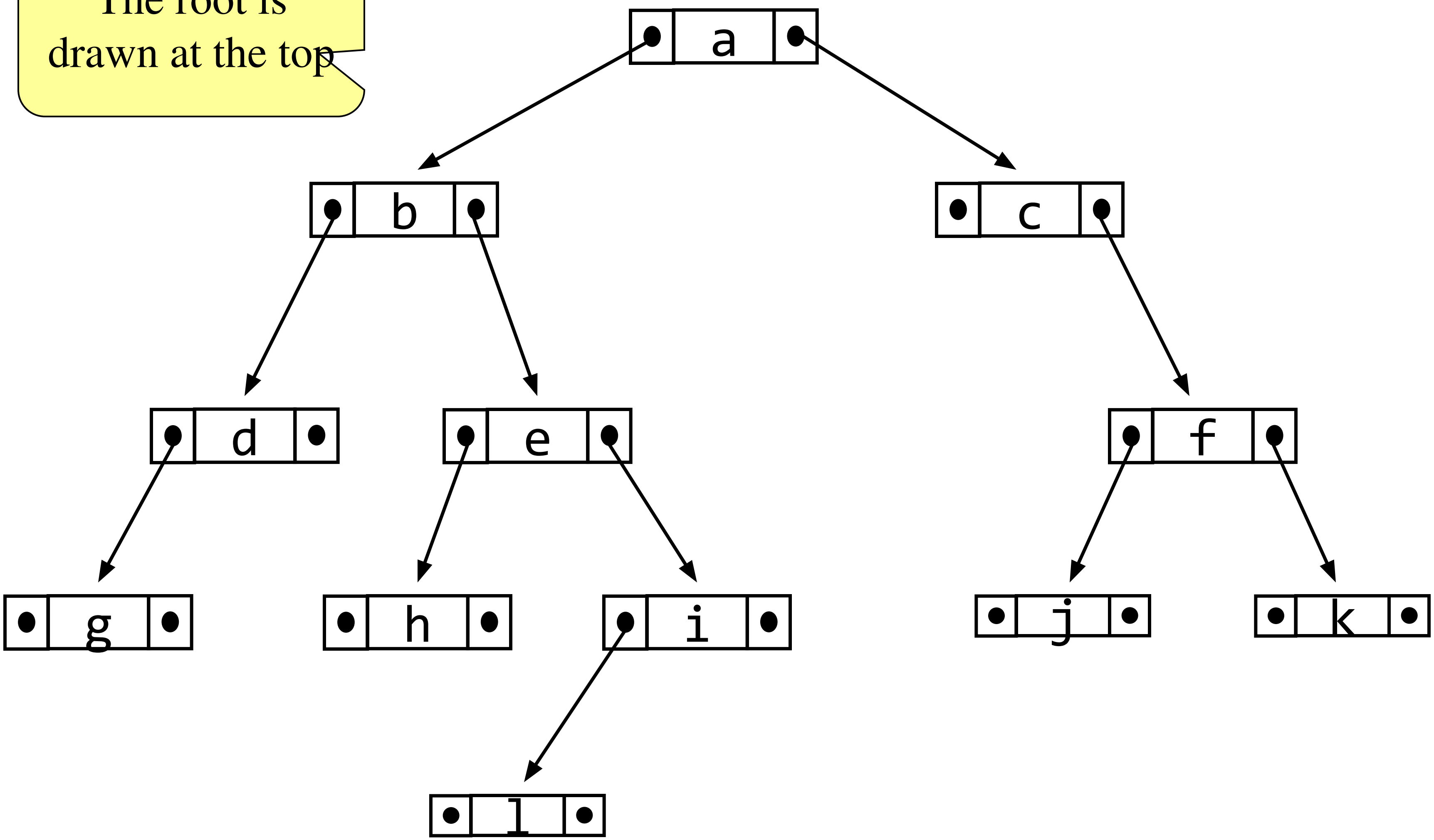
Binary Trees

Parts of a binary tree

- A binary tree is composed of zero or more nodes
  - In Java, a reference to a binary tree may be `null`
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be `null`), and
  - A reference or pointer to a right child (may be `null`)
- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a *unique* path
- A node with no left child and no right child is called a leaf
  - In some binary trees, only the leaves contain a value

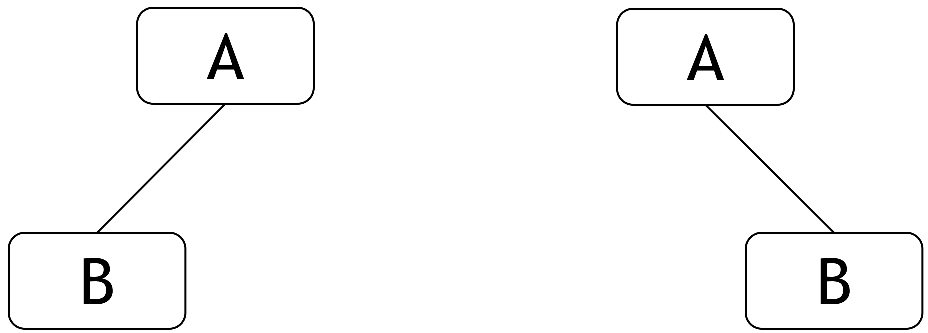
Picture of a binary tree

The root is drawn at the top



Left ≠ Right

- The following two binary trees are *different:*

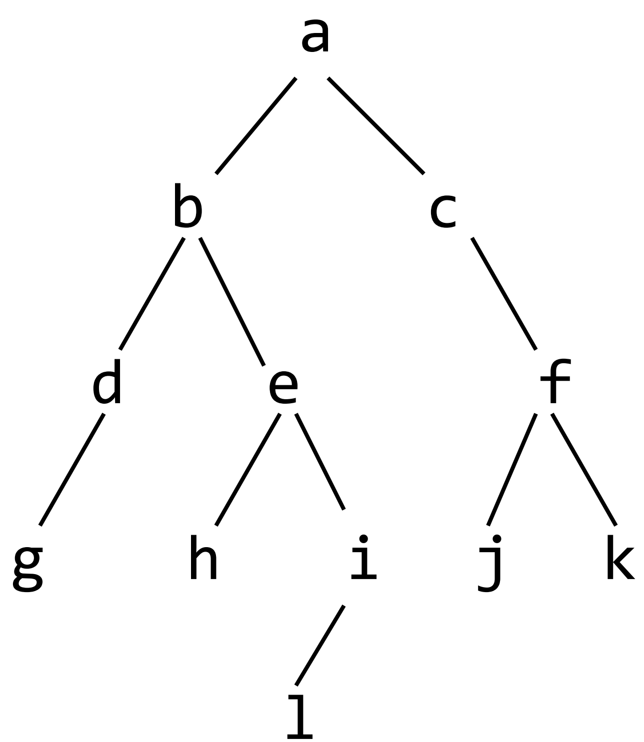


- In the first binary tree, node A has a left child but no right child; in the second, node A has a right child but no left child
- Put another way: Left and right are *not* relative terms

More terminology

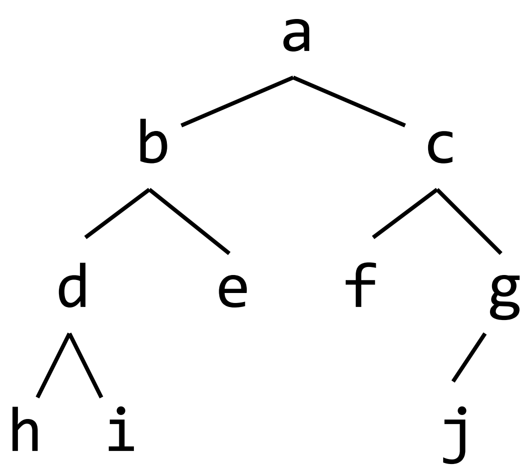
- Node A is the parent of node B if node B is a child of A
- Node A is an ancestor of node B if A is a parent of B, or if some child of A is an ancestor of B
  - In less formal terms, A is an ancestor of B if B is a child of A, or a child of a child of A, or a child of a child of a child of A, etc.
- Node B is a descendant of A if A is an ancestor of B
- Nodes A and B are siblings if they have the same parent

Size and depth

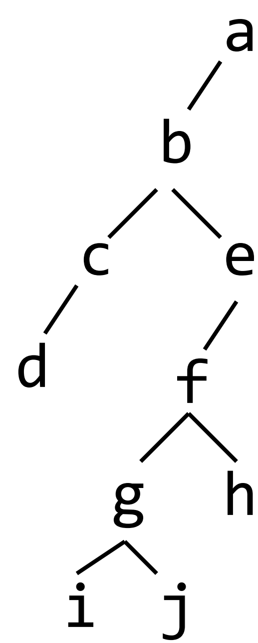


- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero
  - e is at depth 2
- The depth of a binary tree is the depth of its deepest node
  - This tree has depth 4

Balance



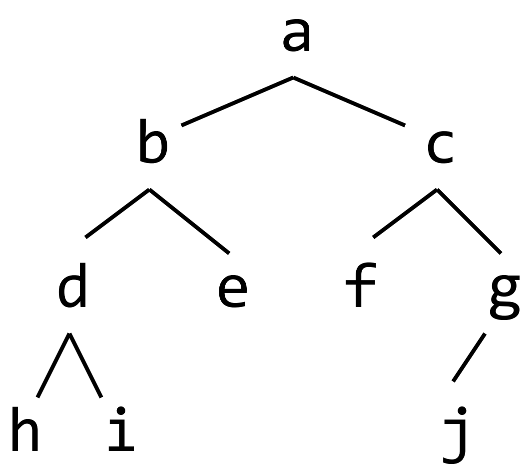
A balanced binary tree



An unbalanced binary tree

- In most applications, a reasonably balanced binary tree is desirable

Definitions of “balanced”



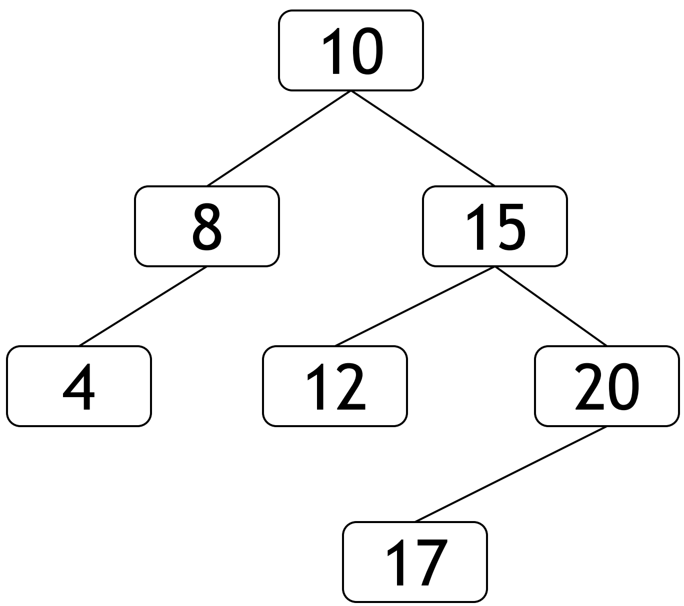
A balanced binary tree

Define the height of a node as the largest distance from that node to a leaf

1. A binary tree is balanced if every level above the lowest is “full” (contains 2^n nodes)
2. A binary tree is balanced if the height of each node differs by at most 1 from the heights of its sibling

Sorted binary trees

- A binary tree is sorted if every node in the tree is larger than (or equal to) its left descendants, and smaller than (or equal to) its right descendants
- Equal nodes can go either on the left or the right (but it has to be consistent)



BinarySearchTree class

```
public class BinarySearchTree {
    BinaryTreeNode root;
}
public class BinaryTreeNode {
    int val;
    BinaryTreeNode leftChild=null;
    BinaryTreeNode rightChild=null;
    BinaryTreeNode(int val, BinaryTreeNode leftChild,
BinaryTreeNode rightChild){
     this.val = val;
     this.leftChild = leftChild;
     this.rightChild = rightChild;
    }

}
```

BinarySearchTree class

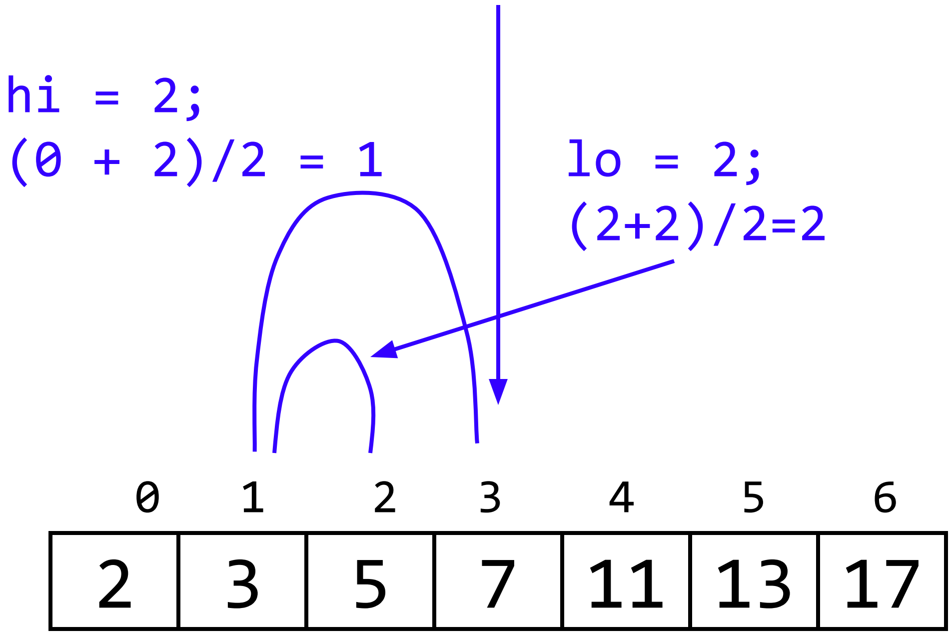
```
public class BinarySearchTree {
    BinaryTreeNode root;
    boolean search(int key){
     BinaryTreeNode tmp = root;
     while(tmp!=null){
         if (tmp.val==key)
             return true;
         if (key<tmp.val)
             tmp = tmp.leftChild;
         else // key>tmp.val
             tmp = tmp.rightChild;
     }
     return false;
    }
}
```

Binary search in a sorted array

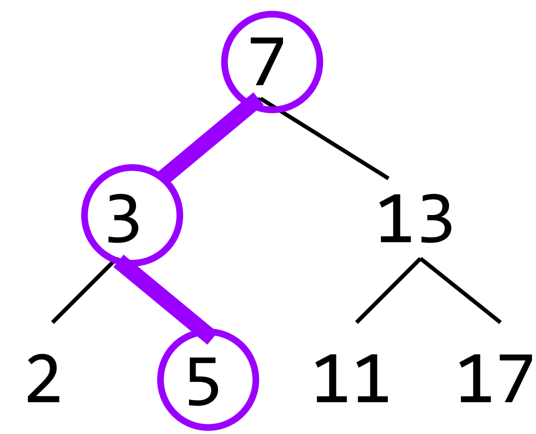
- Look at array location `(lo + hi)/2`

Searching for 5:

`(0+6)/2 = 3`



Using a binary search tree



Tree traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root
- (inorder, preorder, postorder)

class BinaryTree

-
- A constructor for a binary tree should have three parameters, corresponding to the three fields
- An “empty” binary tree is just a value of null
  - Therefore, we can’t have an `isEmpty()` method (why not?)

Preorder traversal

- In preorder, the root is visited *first*

```
public class BinaryTreeNode {
    int val;
    BinaryTreeNode leftChild=null;
    BinaryTreeNode rightChild=null;
    // root, left, right
    void preOrderTraverse(){
      System.out.println(val);
      if (leftChild!=null)
          leftChild.preOrderTraverse();
      if (rightChild!=null)
          rightChild.preOrderTraverse();
    }
}
public class MyBinaryTree {
    BinaryTreeNode root;
    void preOrderTraverse(){
      if (root!=null)
          root.preOrderTraverse();
    }
}
```

Inorder traversal

- In inorder, the root is visited *in the middle*

```
public class MyBinaryTree {
    BinaryTreeNode root;
    void inOrderTraverse(){
     if (root!=null)
         root.inOrderTraverse();
    }
}
public class BinaryTreeNode {
    int val;
    BinaryTreeNode leftChild=null;
    BinaryTreeNode rightChild=null;
    // left, root, right
    void inOrderTraverse(){
     if (leftChild!=null)
         leftChild.inOrderTraverse();
     System.out.println(val);
     if (rightChild!=null)
         rightChild.inOrderTraverse();
    }
}
```

Postorder traversal

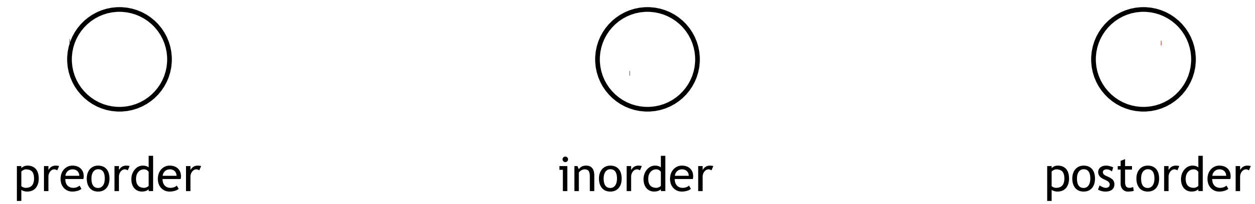
- In postorder, the root is visited *last*

```
public class MyBinaryTree {
    BinaryTreeNode root;
    void postOrderTraverse(){
     if (root!=null)
         root.postOrderTraverse();
    }
}
public class BinaryTreeNode {
    int val;
    BinaryTreeNode leftChild=null;
    BinaryTreeNode rightChild=null;
    // left, root, right
    void postOrderTraverse(){
     if (leftChild!=null)
         leftChild.inOrderTraverse();
     if (rightChild!=null)
         rightChild.inOrderTraverse();
     System.out.println(val);
    }
}
```

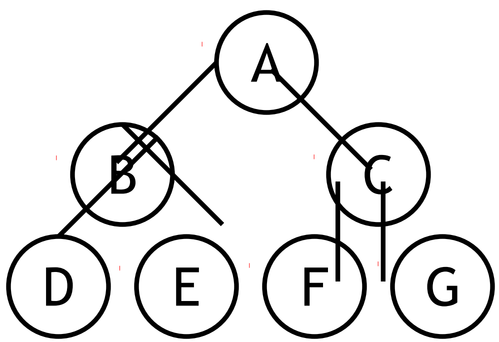
Tree traversals using "flags"

http://tinyurl.com/y7aa6w7j

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:

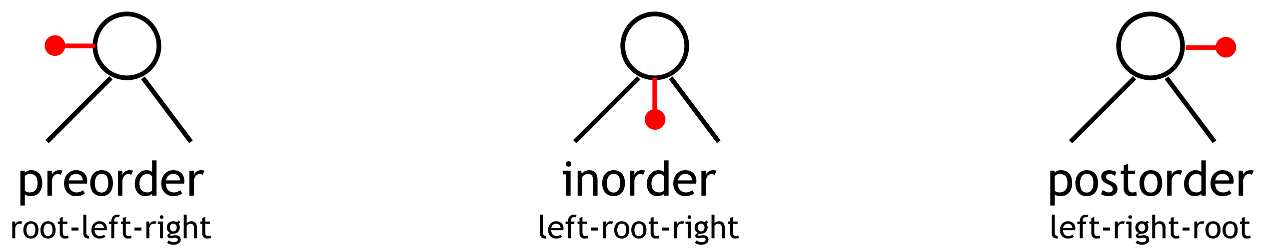


- To traverse the tree, collect the flags:

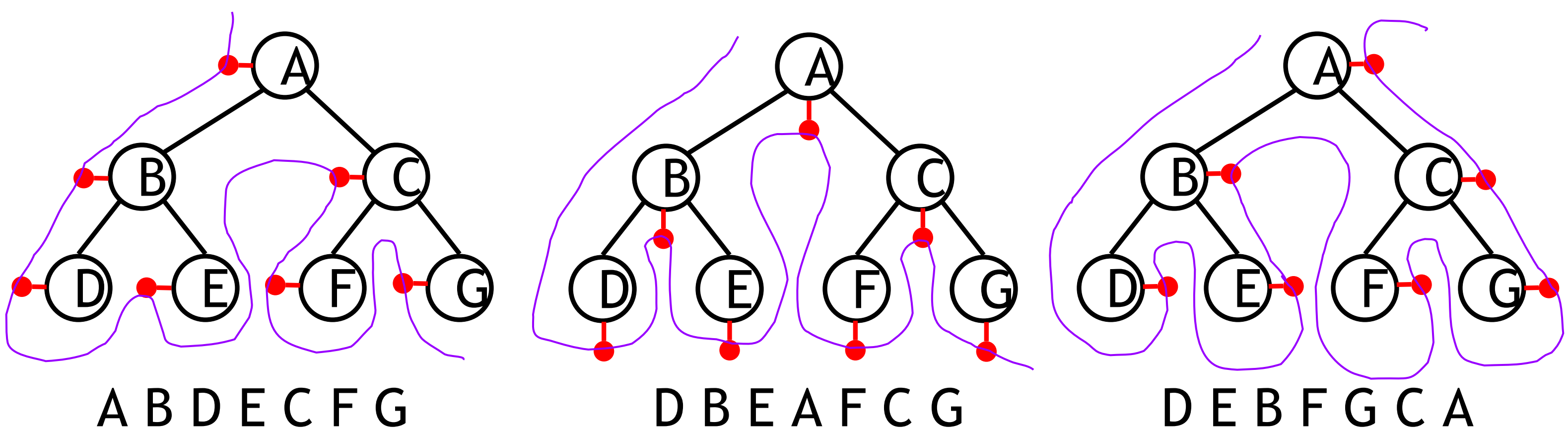


Tree traversals using “flags”

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



- To traverse the tree, collect the flags:



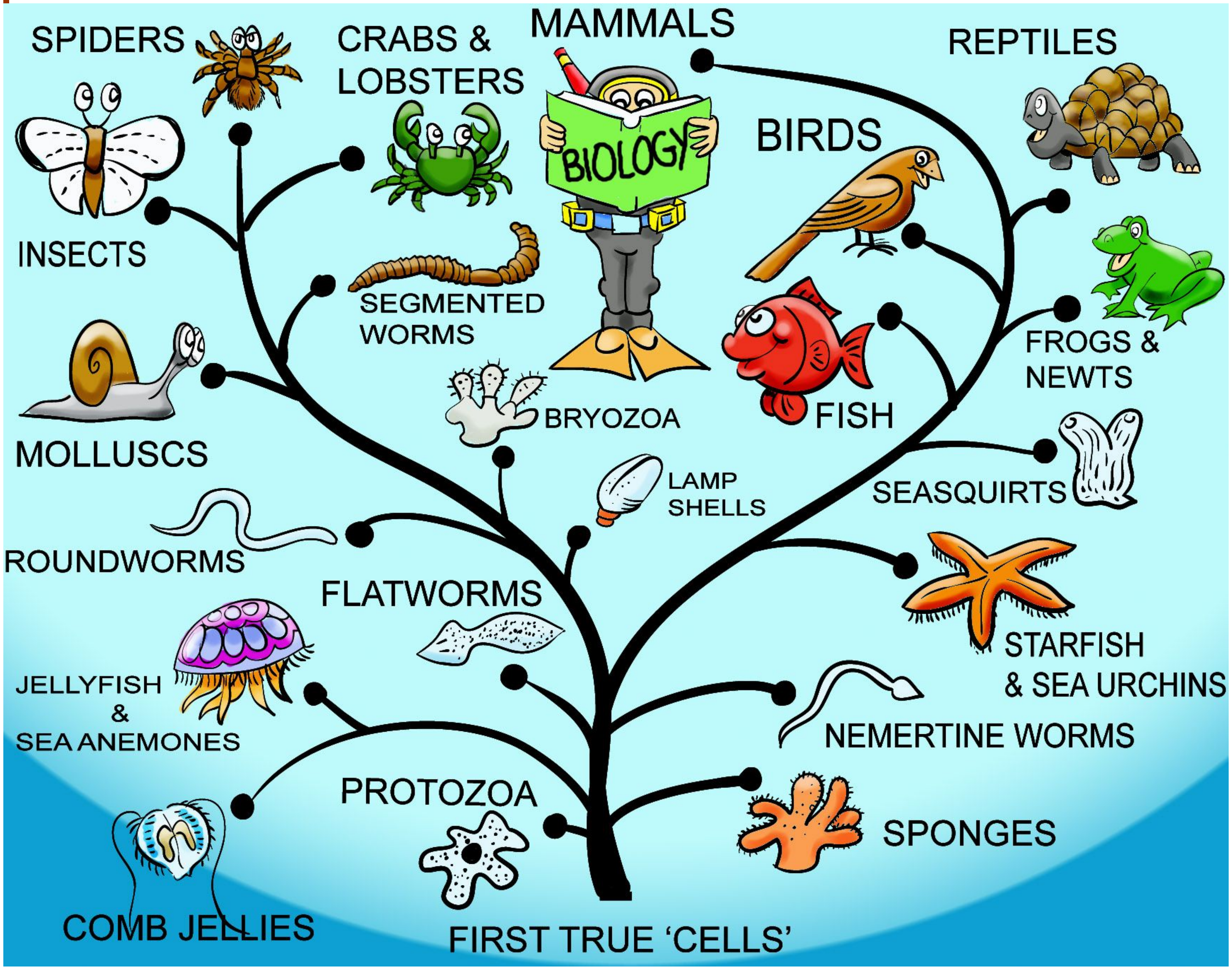
Copying a binary tree

```
public class MyBinaryTree {
    BinaryTreeNode root=null;
    MyBinaryTree copyTree(){
       MyBinaryTree newTree = new MyBinaryTree();
       if (root!=null)
             newTree.root = root.copyNode();
       return newTree;
    }
}
public class BinaryTreeNode {
    int val;
    BinaryTreeNode leftChild=null;
    BinaryTreeNode rightChild=null;
    BinaryTreeNode(int val, BinaryTreeNode leftChild, BinaryTreeNode rightChild){
       this.val = val;
       this.leftChild = leftChild;
       this.rightChild = rightChild;
    }
    BinaryTreeNode copyNode(){
       BinaryTreeNode newLeft=null;
       BinaryTreeNode newRight=null;
       if (leftChild!=null)
             newLeft = leftChild.copyNode();
       if (rightChild!=null)
             newRight = rightChild.copyNode();
       BinaryTreeNode newNode = BinaryTreeNode(val, newLeft, newRight);
       return newNode;
    }
}
```

Other traversals

- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
- Reverse preorder: root, right subtree, left subtree
- Reverse inorder: right subtree, root, left subtree
- Reverse postorder: right subtree, left subtree, root

The End



http://www.cienciasatlantico.blogsek.es/2013/05/07/evolution-and-the-tree-of-life/evolutionary-tree-original/