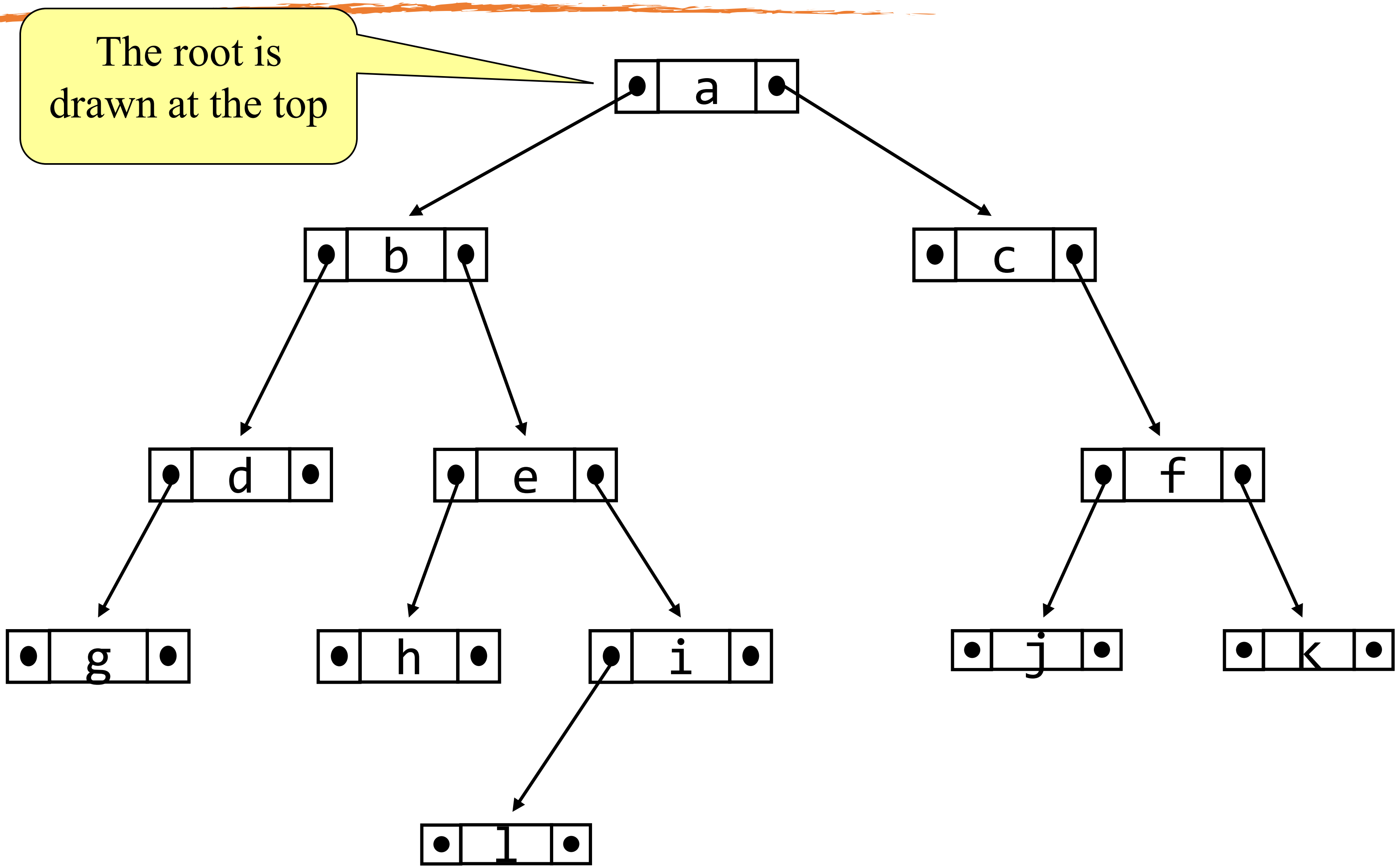
Lecture 05

Binary Trees

Parts of a binary tree

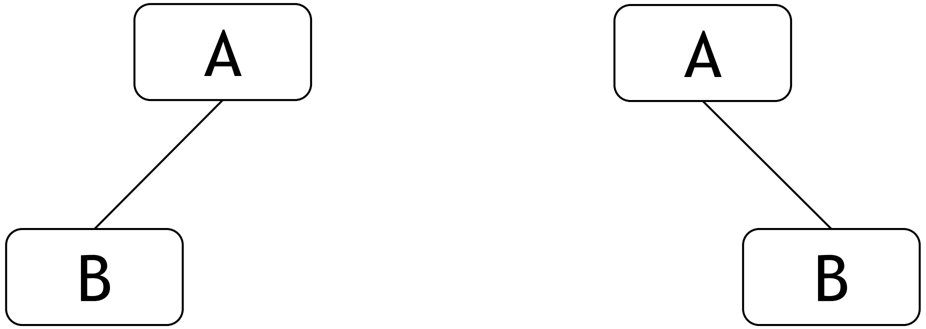
- A binary tree is composed of zero or more nodes
  - In Java, a reference to a binary tree may be `null`
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be `null`), and
  - A reference or pointer to a right child (may be `null`)
- A binary tree 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



Left ≠ Right

- The following two binary trees are *different:*

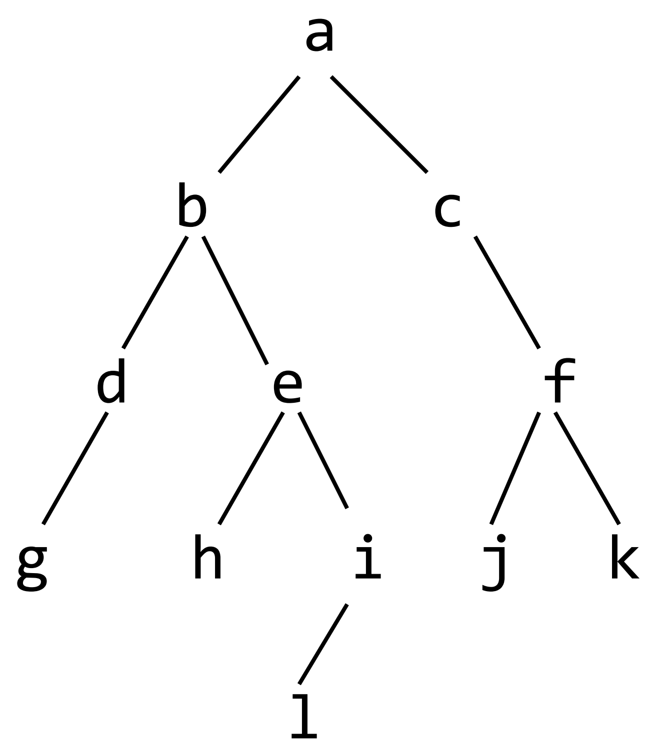


- In the first binary tree, node A has a left child but no right child; in the second, node A has a right child but no left child
- Put another way: Left and right are *not* relative terms

More terminology

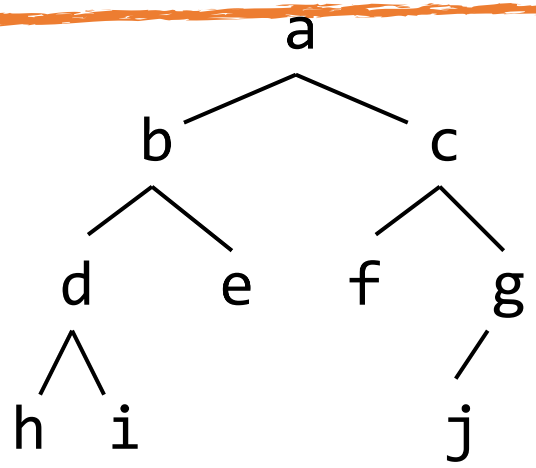
- Node A is the parent of node B if node B is a child of A
- Node A is an ancestor of node B if A is a parent of B, or if some child of A is an ancestor of B
  - In less formal terms, A is an ancestor of B if B is a child of A, or a child of a child of A, or a child of a child of a child of A, etc.
- 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

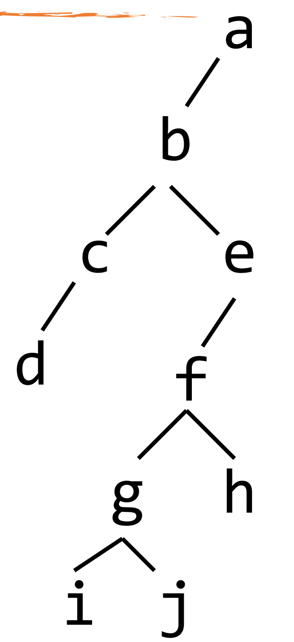


- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero
  - e is at depth 2
- The depth of a binary tree is the depth of its deepest node
  - This tree has depth 4

Balance



A balanced binary tree

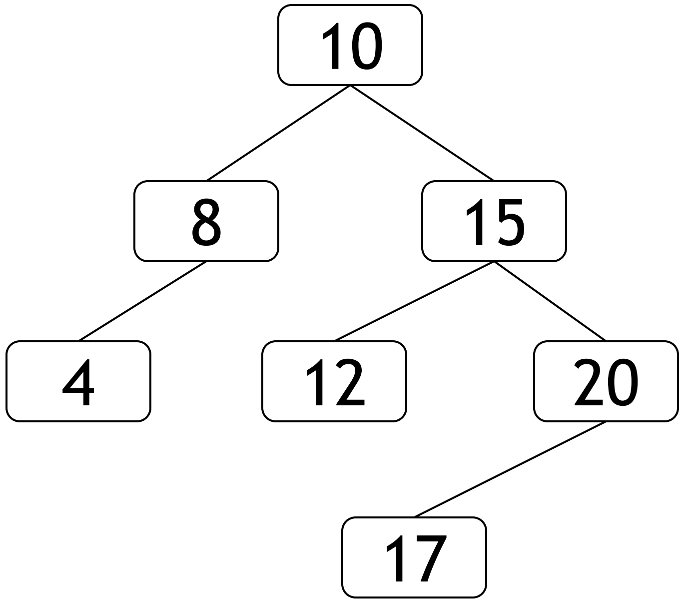


An unbalanced binary tree

- A binary tree is balanced if every level above the lowest is “full” (contains 2^n nodes)
- In most applications, a reasonably balanced binary tree is desirable

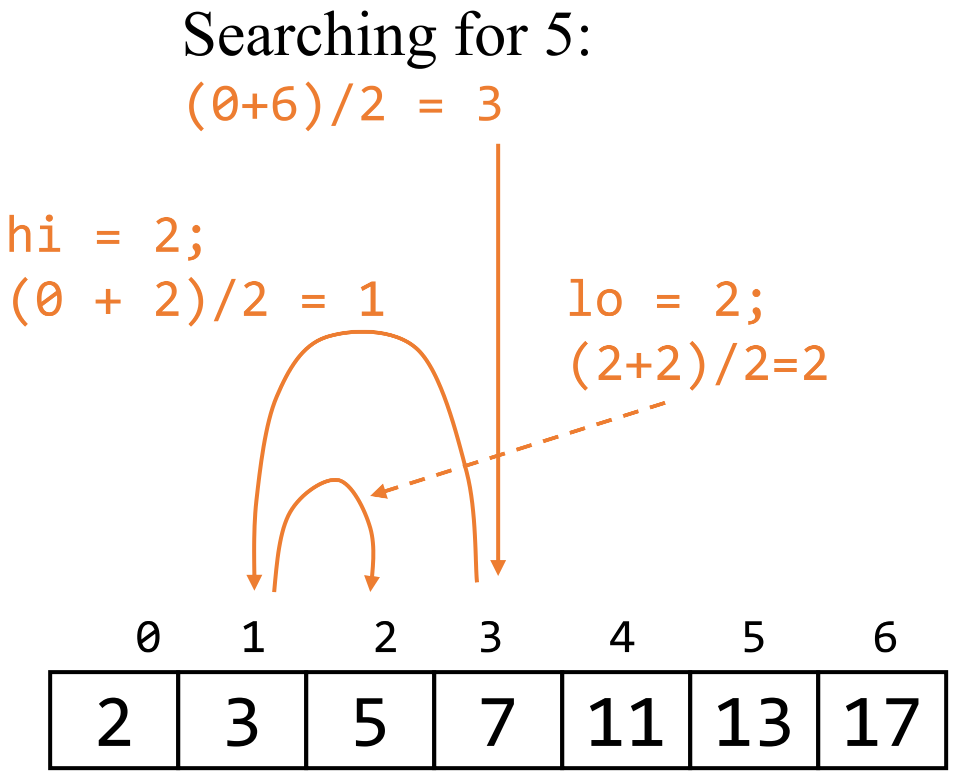
Sorted binary trees

- 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)

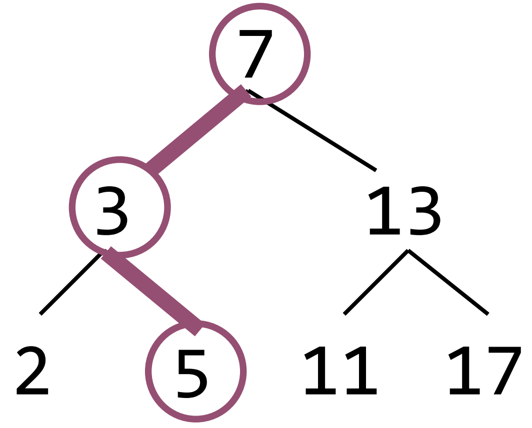


Binary search in a sorted array

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



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

class BinaryTree

- ```
  class BinaryTree<V> {
      V value;
      BinaryTree<V> leftChild;
      BinaryTree<V> rightChild;

      // Assorted methods…
  }
  ```
- 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*
- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorderPrint(BinaryTree bt) {
     if (bt == null) return;
     System.out.println(bt.value);
     preorderPrint(bt.leftChild);
     preorderPrint(bt.rightChild);
}
```

Inorder traversal

- In inorder, the root is visited *in the middle*
- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorderPrint(BinaryTree bt) {
     if (bt == null) return;
     inorderPrint(bt.leftChild);
     System.out.println(bt.value);
     inorderPrint(bt.rightChild);
}
```

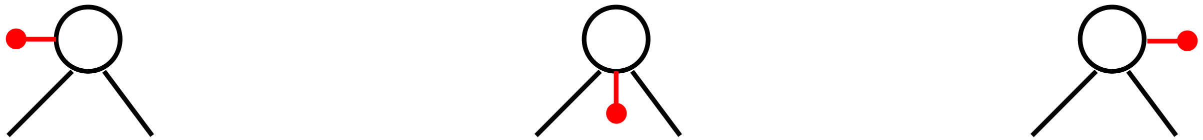
Postorder traversal

- In postorder, the root is visited *last*
- Here's a postorder traversal to print out all the elements in the binary tree:

```
public void postorderPrint(BinaryTree bt) {
     if (bt == null) return;
     postorderPrint(bt.leftChild);
     postorderPrint(bt.rightChild);
     System.out.println(bt.value);
}
```

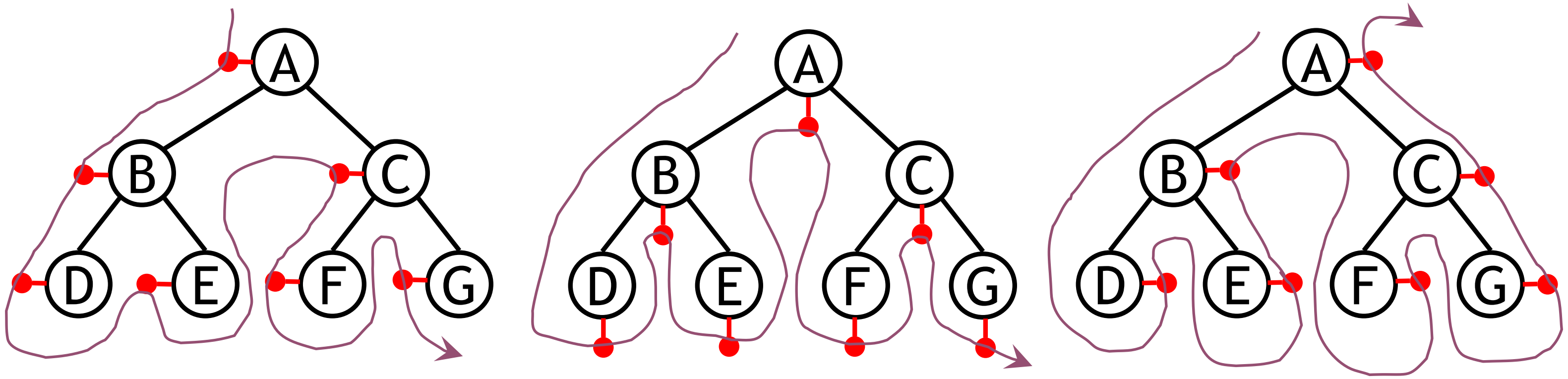
Tree traversals using “flags”

- 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:



preorder inorder postorder

- To traverse the tree, collect the flags:



A B D E C F G D B E A F C G D E B F G C A

Copying a binary tree

- In postorder, the root is visited *last*
- Here's a postorder traversal to make a complete copy of a given binary tree:

```
public static BinaryTree copyTree(BinaryTree bt) {
     if (bt == null) return null;
     BinaryTree left = copyTree(bt.leftChild);
     BinaryTree right = copyTree(bt.rightChild);
     return new BinaryTree(bt.value, left, right);
}
```

Copying a binary tree

- Many programs use copy constructors—a constructor that constructs a new object that is a copy of the given object
- Here's a copy constructor for nonempty binary trees:

```
public BinaryTree(BinaryTree bt) {
    value = bt.value;
    if (bt.leftChild != null) {
        leftChild = new BinaryTree(bt.leftChild);
    }
    if (bt.rightChild != null) {
        rightChild = new BinaryTree(bt.rightChild);
    };
}
```

- Notice the tests to avoid `nullPointerException`s!

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