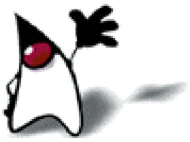
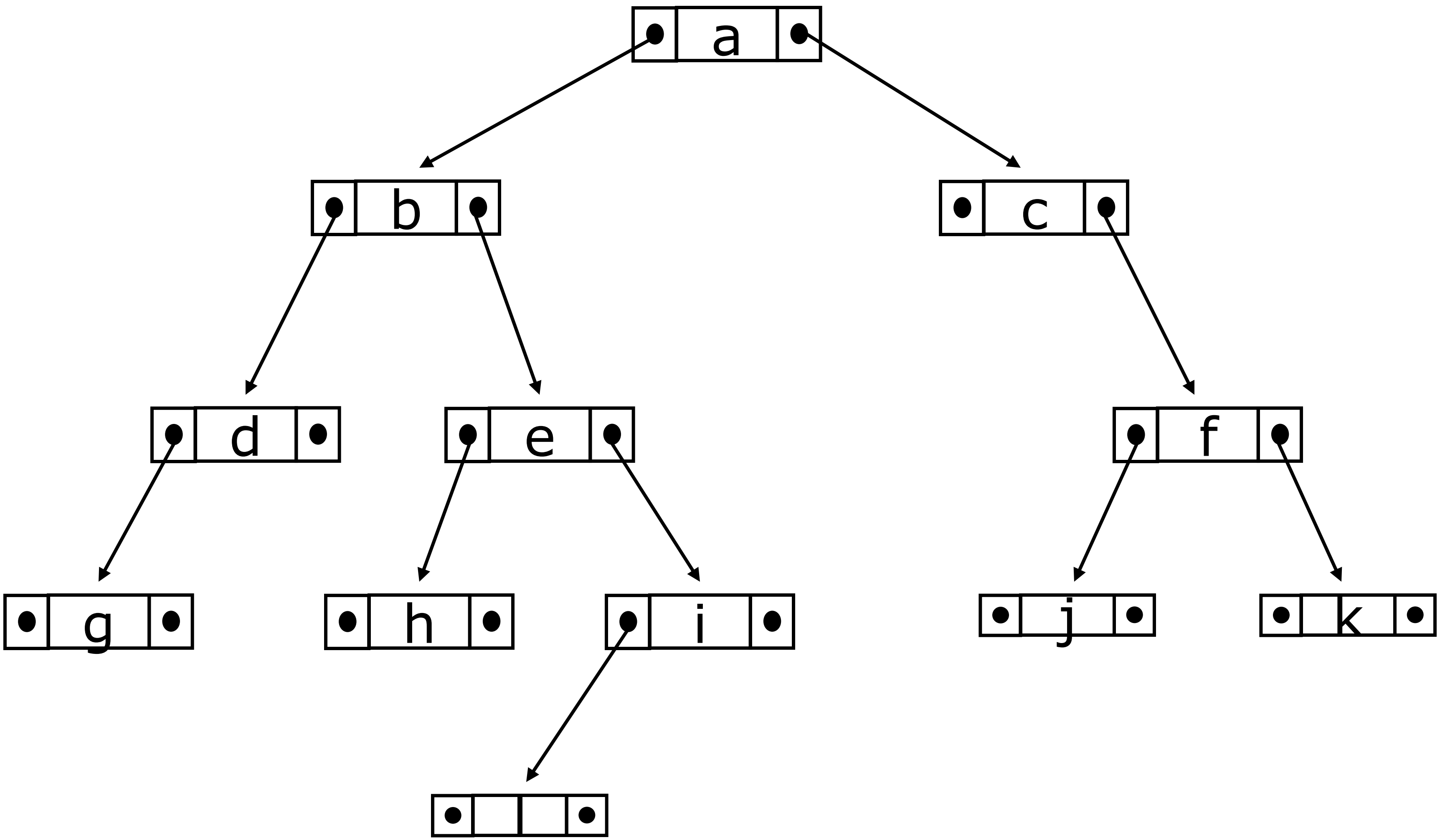
Binary Trees



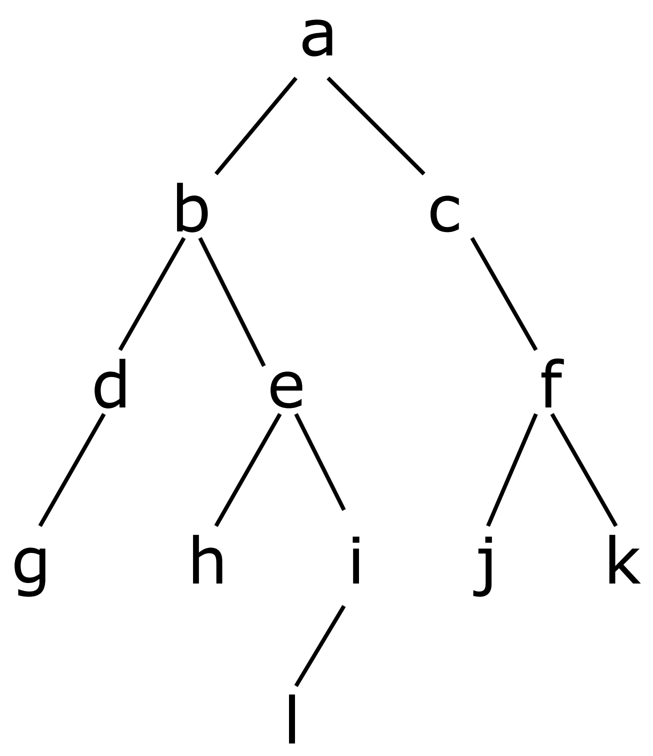
Parts of a binary tree

- A binary tree is composed of zero or more nodes
- 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 neither a left child nor a right child is called a leaf
  - In some binary trees, only the leaves contain a value

Picture of a binary tree

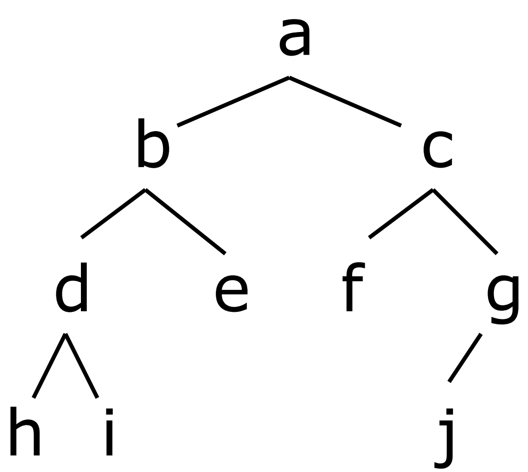


Size and depth

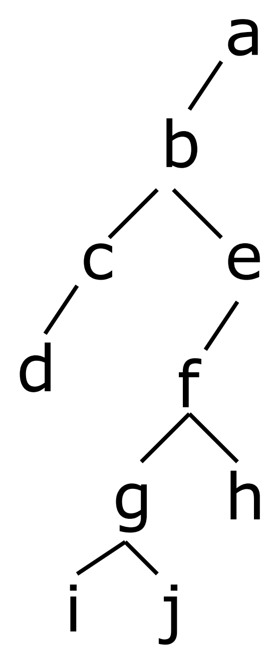


- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero
  - e is at depth 2
- The depth of a binary tree is the depth of its deepest node
  - This tree has depth 4

Balance



A balanced binary tree

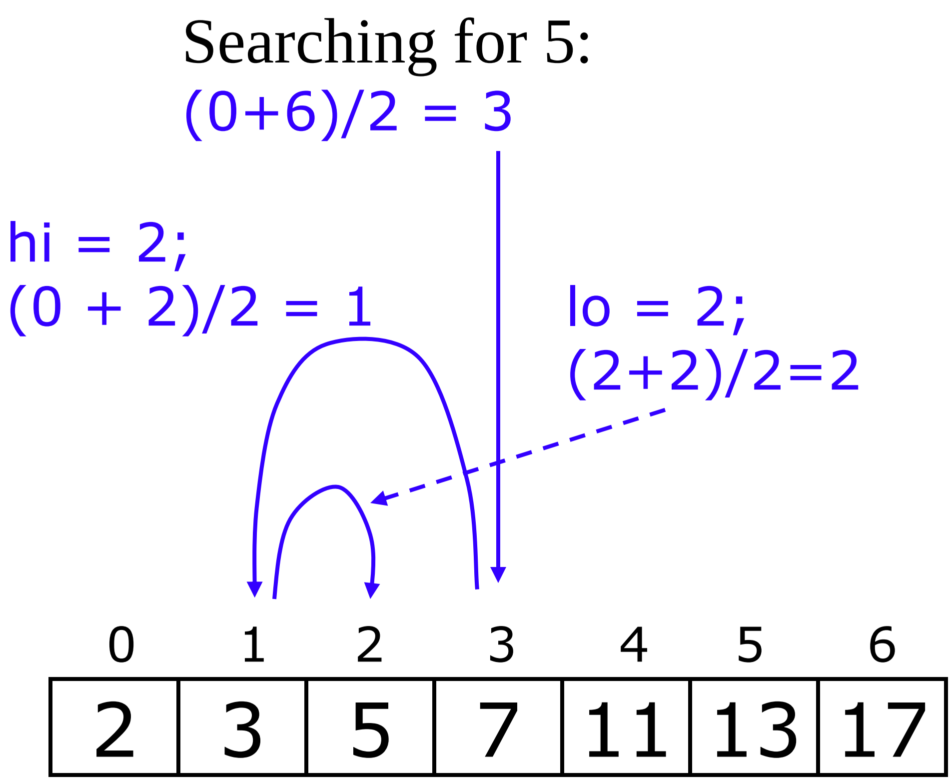


An unbalanced binary tree

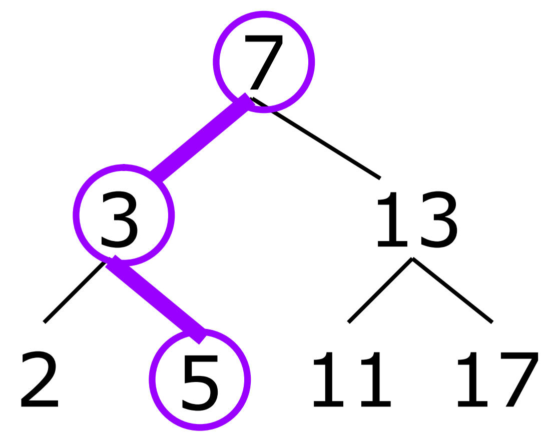
- A binary tree is balanced if every level above the lowest is “full” (contains 2^n nodes)
- In most applications, a reasonably balanced binary tree is desirable

Binary search in an 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

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);
}
```

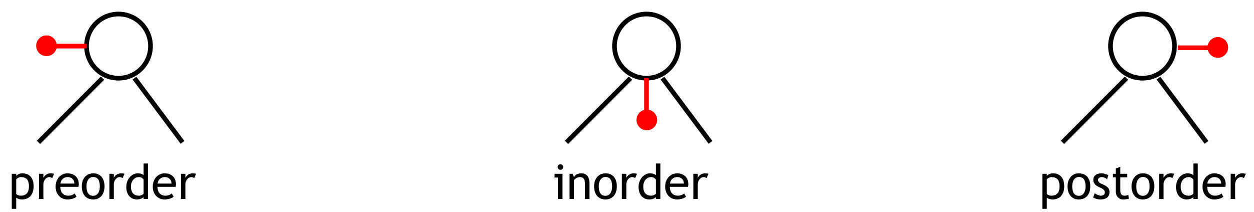
Postorder traversal

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

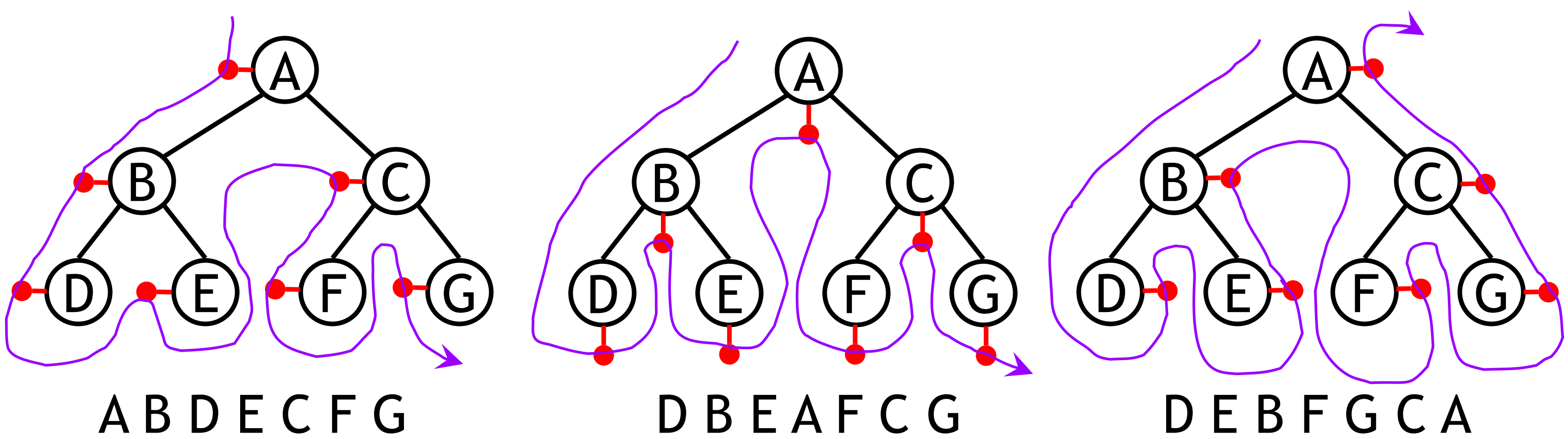
```
public void postorderPrint(BinaryTree bt) {
    if (bt == null) return;
    postorderPrint(bt.leftChild);
    postorderPrint(bt.rightChild);
    System.out.println(bt.value);
}
```

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:



- To traverse the tree, collect the flags:



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 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);
}
```

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