Binary Tree (Cont.)

`remove()` + Tree Traversals

Tree Traversals

- breath-first-search (BFS)
- depth-first-search (DFS)

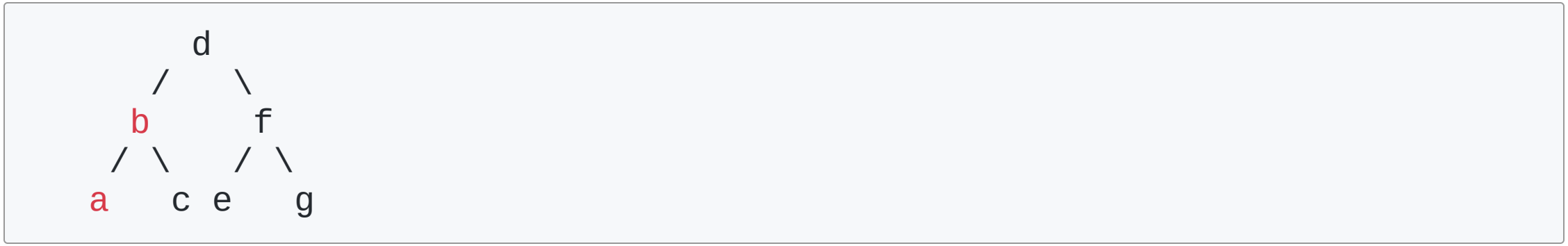
depth-first-search (DFS)

- in-order
- pre-order
- post-order

In-order traversal

Algorithm in-order

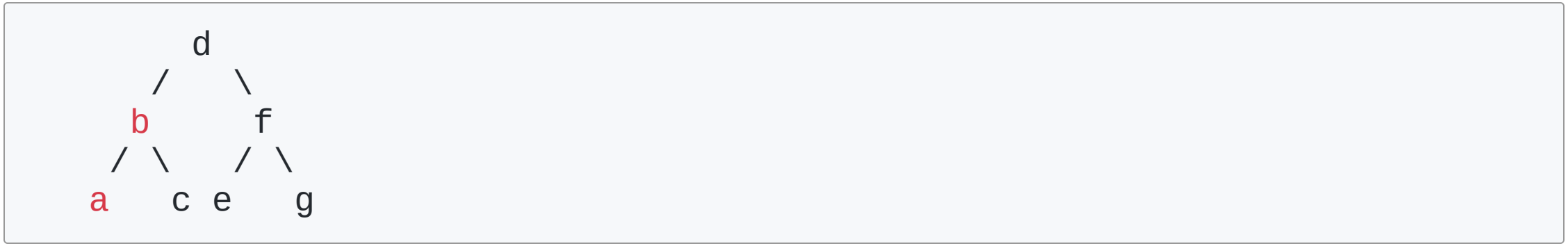
1. Traverse the left subtree, i.e., call in-order (left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call in-order (right-subtree)



In-order traversal (Cont.)

Algorithm in-order

1. Traverse the left subtree, i.e., call in-order (left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call in-order (right-subtree)

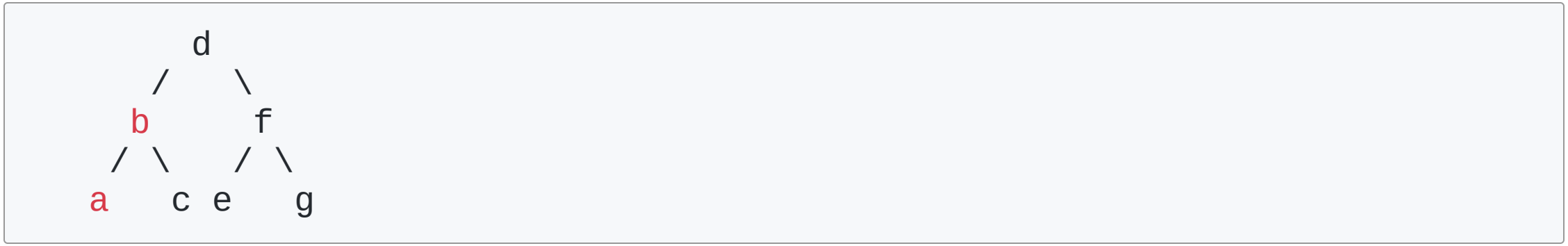



```
>> [a, b, c, d, e, f, g]
```

Pre-order traversal

Algorithm pre-order(tree)

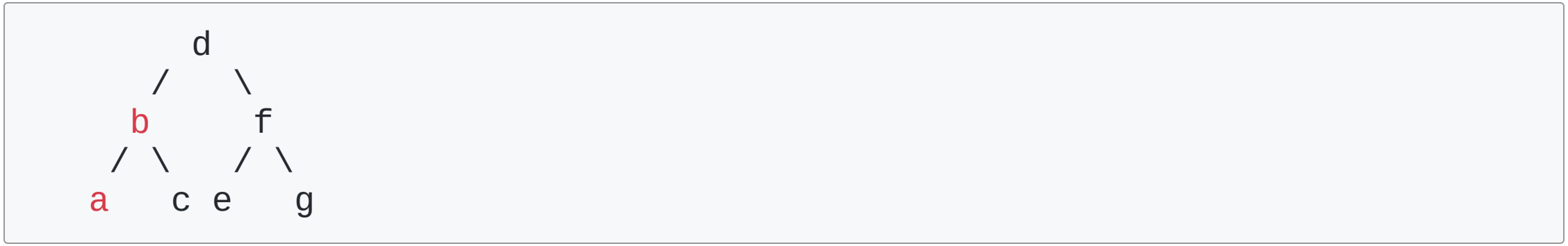
1. Visit the root.
2. Traverse the left subtree, i.e., call pre-order (left-subtree)
3. Traverse the right subtree, i.e., call pre-order (right-subtree)



Pre-order traversal (Cont.)

Algorithm pre-order(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call pre-order (left-subtree)
3. Traverse the right subtree, i.e., call pre-order (right-subtree)

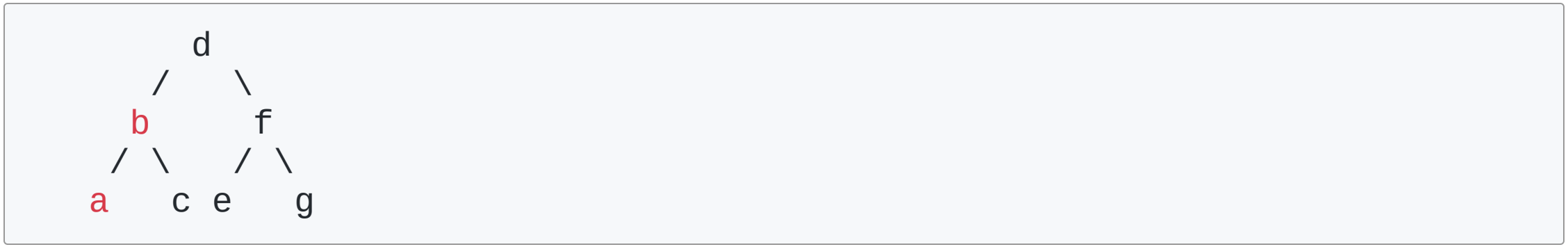



```
>> [d, b, a, c, f, e, g]
```

Post-order traversal

Algorithm post-order(tree)

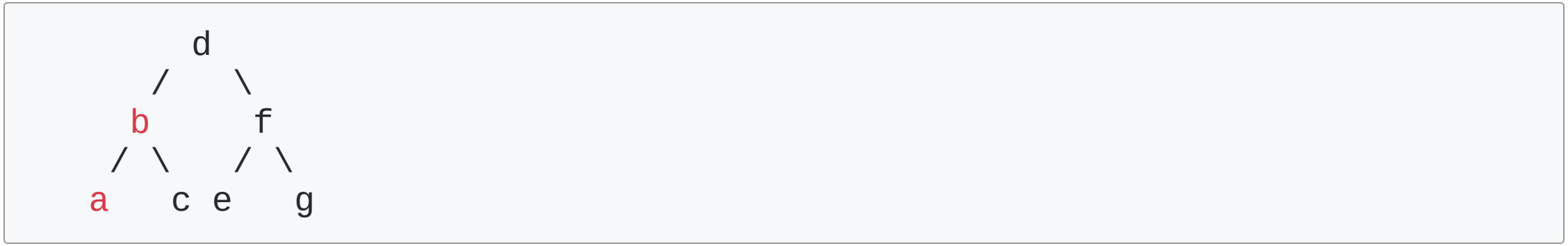
1. Traverse the left subtree, i.e., call post-order (left-subtree)
2. Traverse the right subtree, i.e., call post-order (right-subtree)
3. Visit the root.



Post-order traversal (Cont.)

Algorithm post-order(tree)

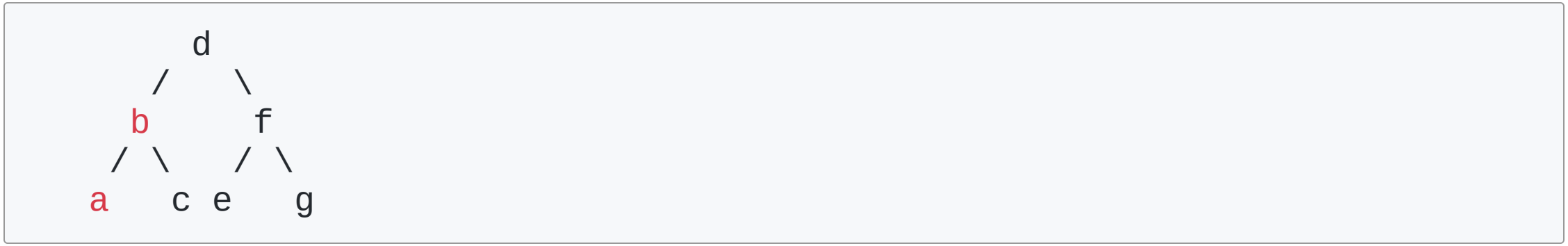
1. Traverse the left subtree, i.e., call post-order (left-subtree)
2. Traverse the right subtree, i.e., call post-order (right-subtree)
3. Visit the root.




```
>> [a, c, b, e, g, f, d]
```

breath-first-search (BFS)

Traverse tree one level at a time!




```
>> [d, b, f, a, c, e, g]
```

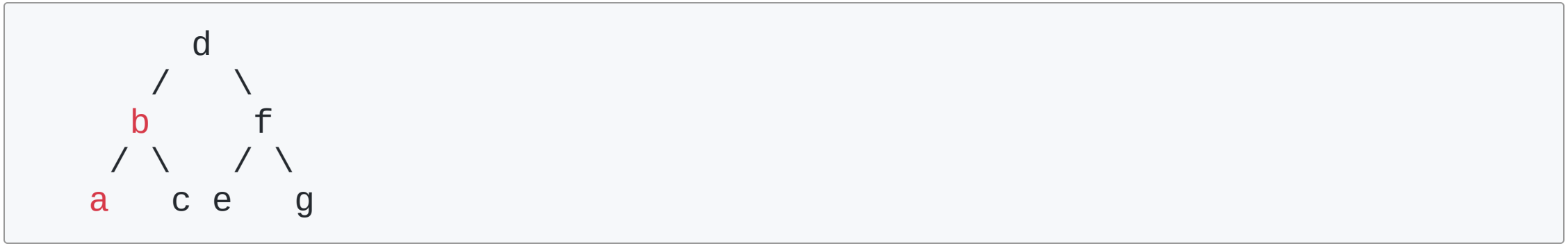
breath-first-search (BFS) (Cont.)

Queue is the key!

```
public void bfsTraversal(Node root) {
  Queue<Node> q = new LinkedList<Node>();
  if (root == null) return;
  q.add(root);
  while (!q.isEmpty()) {
    Node n = (Node) q.remove();
    System.out.println(n.data);
    if (n.left != null)
      q.add(n.left);
    if (n.right != null)
      q.add(n.right);
  }
}
```

Warm-up exercise #1

Print nodes are `k` level from the node



`printKDistant(root, 0) --> [d]`

`printKDistant(root, 1) --> [b, f]`

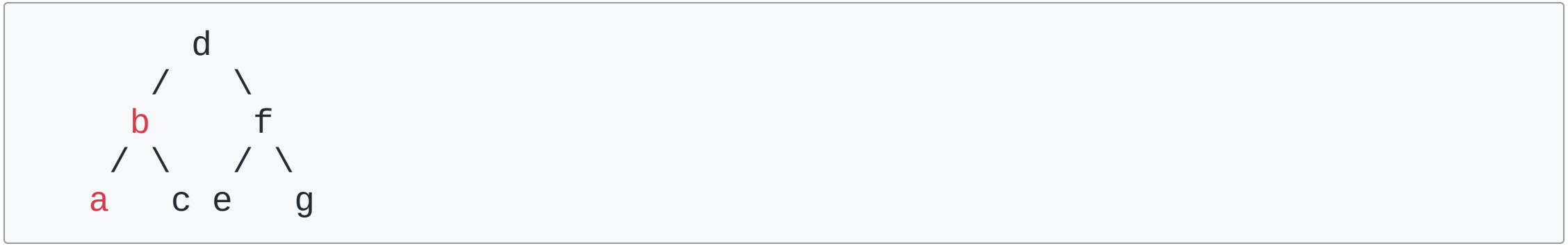
`printKDistant(root, 2) --> [a, c, e, g]`

Solution #1

```
void printKDistant(Node node, int k) {
    if (node == null)
        return;
    if (k == 0) {
        System.out.println(node.data);
        return;
    } else {
        printKDistant(node.left, k - 1);
        printKDistant(node.right, k - 1);
    }
}
```

Warm-up exercise #2

`nthSmallestElement(Node<E> node, int n)`



`nthSmallestElement(root, 0) --> a`

`nthSmallestElement(root, 6) --> g`

`nthSmallestElement(root, 2) --> c`

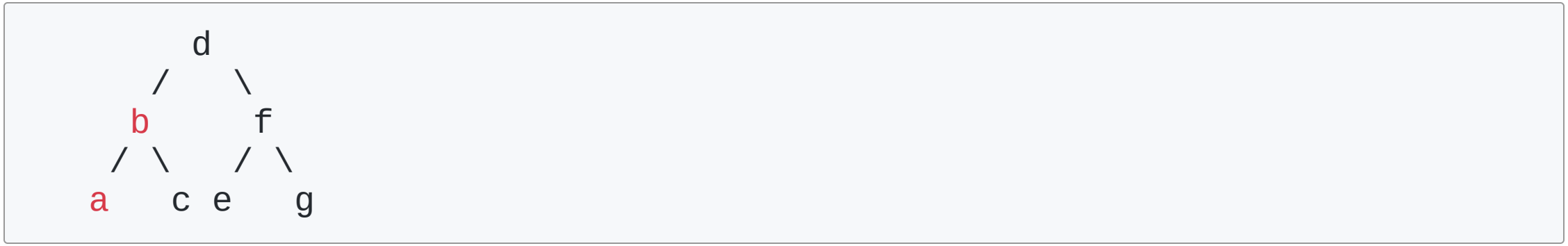
Solution #2

```
public ArrayList<E> inOrder(Node<E> root, ArrayList<E> list) {
    if (root == null) return arr;
    inOrder(root.left, list);
    list.add(root.data);
    inOrder(root.right, list);
    return list;
  }

public E nthSmallestElement(Node<E> root, int k) {
  ArrayList<E> result = inOrder(root, new ArrayList<E>());
  return result.get(k);
}
```

Delete a node from BST

Exercise: Helper method to get smallest node in the subtree




```
>> a
```

Solution

```
T minValue(Node<T> root) {
    T currentMin = root.data;
    while (root.left != null) {
        currentMin = root.left.data;
        root = root.left;
    }

    return currentMin;
}
```

We can use this method to find the replacement for the current node in the right subtree while deleting a node.

Delete node recursively

- Deletes a node holding the value from the subtree
- Returns the root of updated subtree

```
public Node<T> deleteRecursive(Node<T> node, T value) {

// If Node is null return the node
// Traverse left if value is less than current node's value
// Traverse right if value is greater than current node's value
// If we find the target
//    - if left child is null then return the right child as the new root
//    - if right child is null then return the left child as the new root
//    - otherwise,
//        - find the next successor of the node in the right subtree
//        - or largest successor of the node in the left subtree
//      as the replacement of right of left child of the node
//      Then call deleteRecursive to remove duplicate
}
```

Delete (Cont.)

```
public Node<T> deleteRecursive(Node<T> node, T value) {
    /* base case: If the tree is empty */
    if (node == null)
        return node;

    int compareVal = value.compareTo(node.value);

    /* otherwise, recur down the tree */
    if (compareVal < 0) {
        node.left = deleteRecursive(node.left, value);
    } else if (compareVal > 0) {
        node.right = deleteRecursive(node.right, value);
    }

    // if key is same as root's key, then This is the node
    // to be deleted
    else {
        // decrement the size, we just found the element
        size--;

        // node with only one child or no child
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        }

        // node with two children: get the in-order successor (smallest
        // in the right subtree)
        node.data = minValue(node.right);

        // delete the in-order successor, we do not want duplicates stored in tree
        node.right = deleteRecursive(node.right, node.data);
    }

    return node;
}
```

Lab exercise

Creating an in-order traversal iterator on BST