Binary Tree (Cont.)

`remove()` + Tree Traversals

Warm-up exercise

Print nodes are ` k ` level from the node

```
        d
       / \
      b   f
     / \ / \
    a  c e  g
```
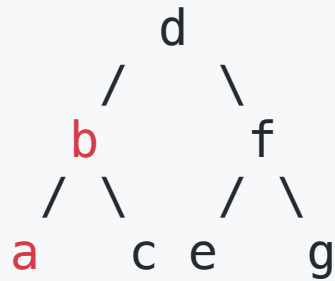
```
printKDistant(root, 1) --> [b, f]
printKDistant(root, 2) --> [a, c, e, g]
```

Solution

```java
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);
    }
}
```

Tree Traversals

- breath-first-search (BFS)

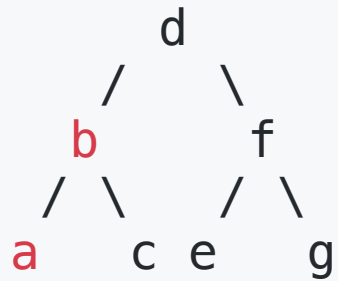- depth-first-search (DFS)

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)

```
        d
       / \
      b       f
     / \     / \
    a    c e    g
```

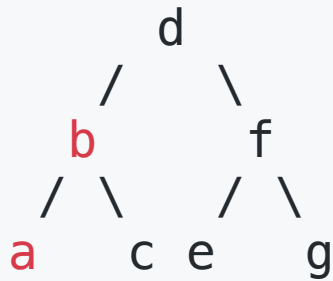# In-order traversal (Cont.)

Algorithm in-order

  1. Traverse the left subtree, i.e., call in-order (left-subtree)

  2. Visit the root.

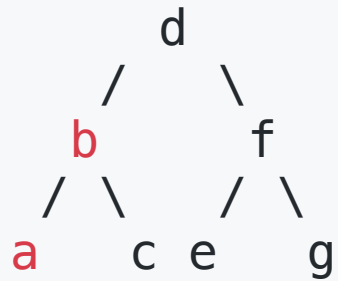 3. Traverse the right subtree, i.e., call in-order (right-subtree)

```
        d
       / \
      b   f
     / \ / \
    a  c e  g
```

```
>> [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)

```
        d
      /   \
    b       f
   / \     / \
  a   c   e   g
```

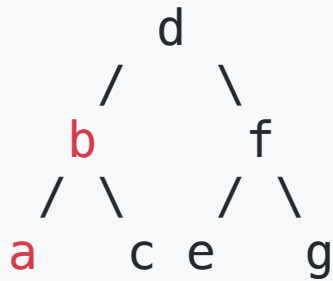# Pre-order traversal (Cont.)

Algorithm pre-order(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call pre-order (left-subtree)

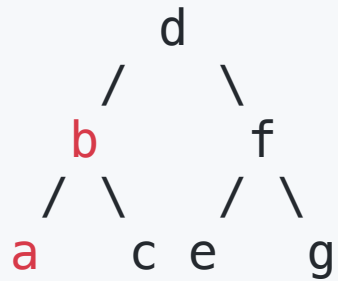3. Traverse the right subtree, i.e., call pre-order (right-subtree)

```
        d
       / \
      b   f
     / \ / \
    a  c e  g
```

```
>> [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.

```
        d
       / \
      b     f
     / \   / \
    a   c e   g
```

Post-order traversal (Cont.)

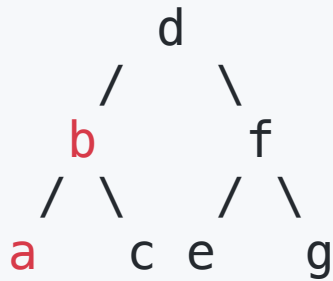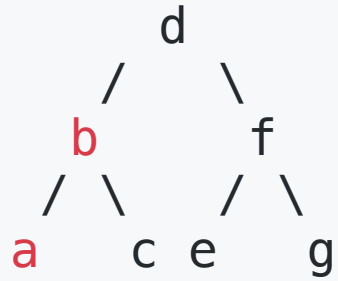Algorithm post-order(tree)

   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.

```
        d
       /   \
      b       f
     / \     / \
    a    c e    g
```

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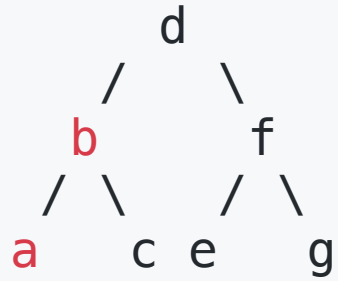
```
>> [d, b, f, a, c, e, g]
```

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

Queue is the key!

```java
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);
  }
}
```

Delete a node from BST

Exercise: Helper method to get smallest node in the subtree

```
        d
      /   \
     b      f
    / \    / \
   a   c  e   g
```

`>> a`

Solution

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

    return currentMin;
}
```

Delete node recursively

```java
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.value = minValue(node.right);

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

    return node;
}
```