

# CS 201

# Data Structures

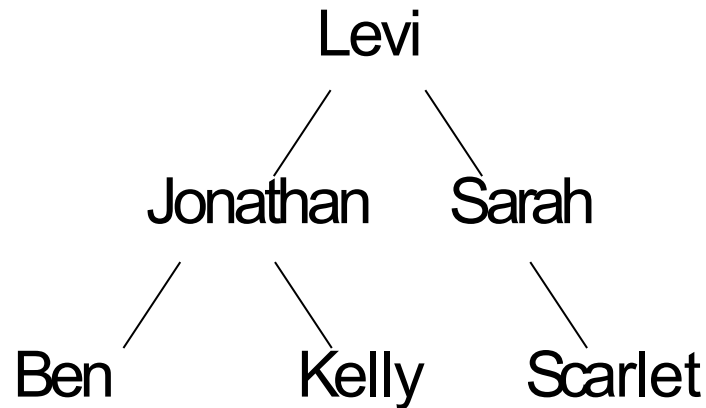
## Tree Traversals

# Tree Traversal Methods

# Tree Traversals

- In linear structures, there are only a few basic ways to traverse (visit) the elements of the data structure
  - Start at one end and visit each element
  - Start at the other end and visit each element
- How do we traverse binary trees?
  - (At least) four reasonable mechanisms
- We imagine that we want to do some work at each node
  - We call that work *processing* the node

# Tree Traversals



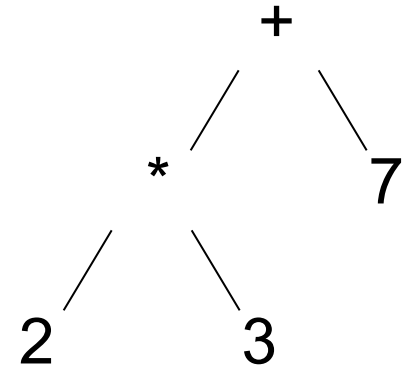
In-order: Ben, Jonathan, Kelly, Levi, Sarah, Scarlet

Pre-order: Levi, Jonathan, Ben, Kelly, Sarah, Scarlet

Post-order: Ben, Kelly, Jonathan, Scarlet, Sarah, Levi,

Level-order: Levi, Jonathan, Sarah, Ben, Kelly, Scarlet

# Tree Traversals



- Pre-order

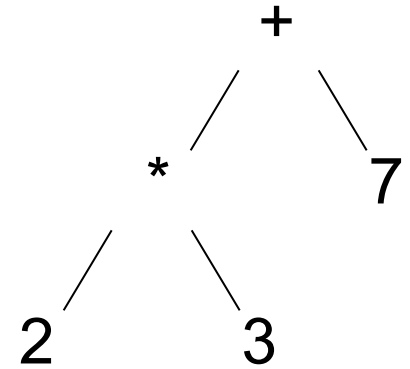
- Each node is processed before any children.  
Process node, process left subtree, then process right subtree. (node, left, right)
  - +\*237

- In-order

- Each node is processed after all nodes in left subtree are processed and before any nodes in right subtree. (left, node, right)
  - 2\*3+7

(“pseudocode”)

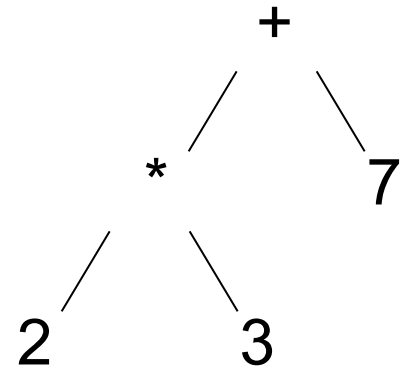
# Tree Traversals



- In-order
  - Each node is processed after all nodes in left subtree are processed and before any nodes in right subtree. (left, node, right)
    - $2*3+7$
- Aside: If processing means *printing*, we could also print a "(" before we process a subtree and a ")" after we process the subtree (skip leaves)
  - $((2 * 3) + 7)$

("pseudocode")

# Tree Traversals



- Post-order

- Each node is processed after its children. Process all nodes in left subtree, then all nodes in right subtree, then node itself. (left, right, node)

- $23*7+$

- Post-order = PostScript order = RPN

- Level-order (not obviously recursive!)

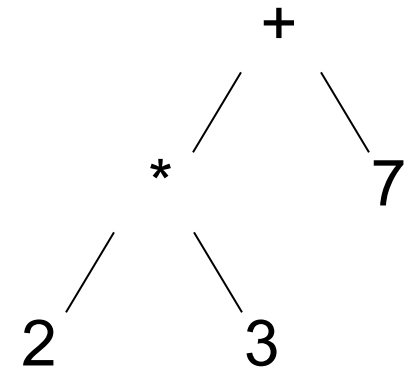
- Nodes at level  $i$  are processed before nodes at level  $i+1$ . (process nodes left to right on each level)

- $+*723$

(“pseudocode”)

# Tree Traversals

```
public void pre-order(BinaryTree t) {  
    if(t.isEmpty()) return;  
    process(t); // some method  
    preOrder(t.left());  
    preOrder(t.right());  
}
```

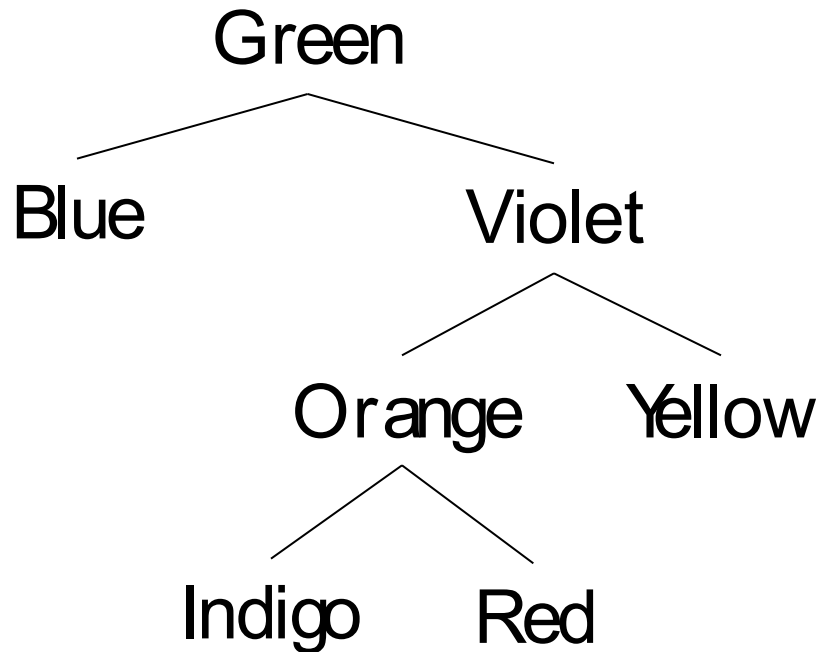


For in-order and post-order: just move  
`process(t)`!

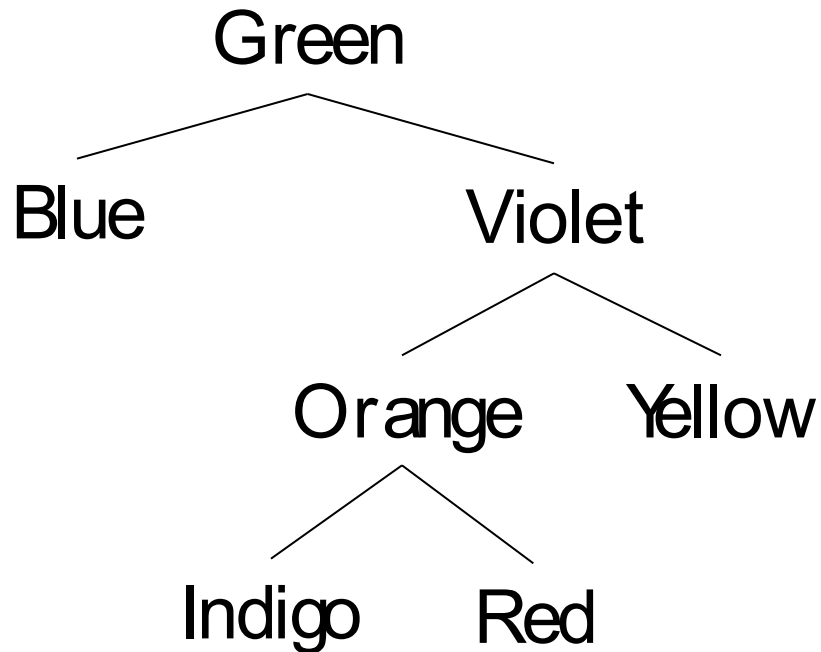
But what about level-order???



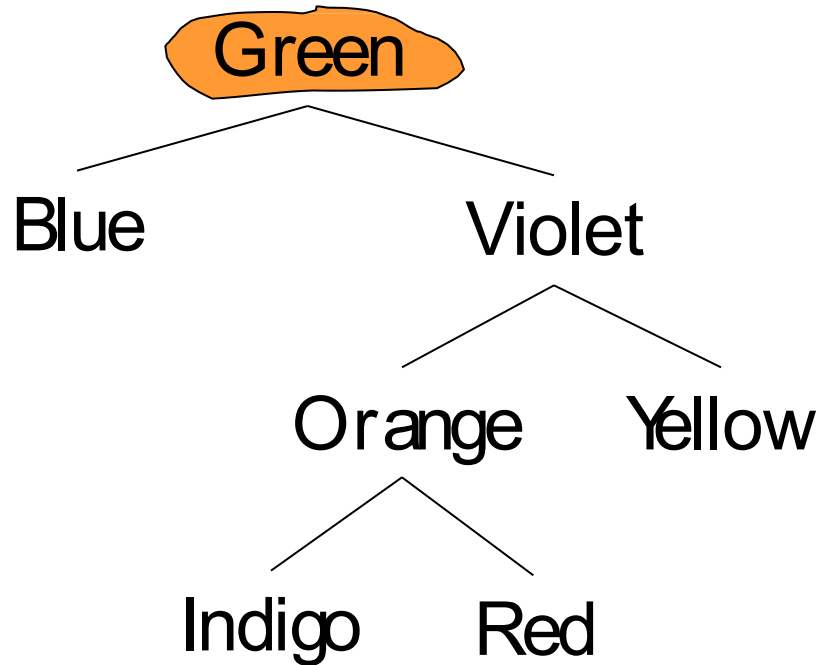
# Level-Order Traversal



# Level-Order Traversal

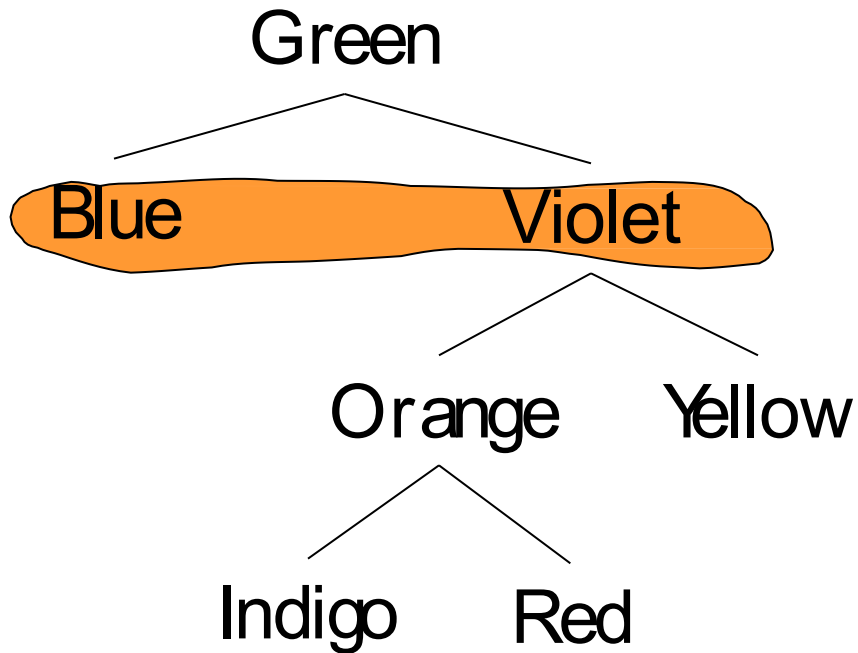


# Level-Order Traversal



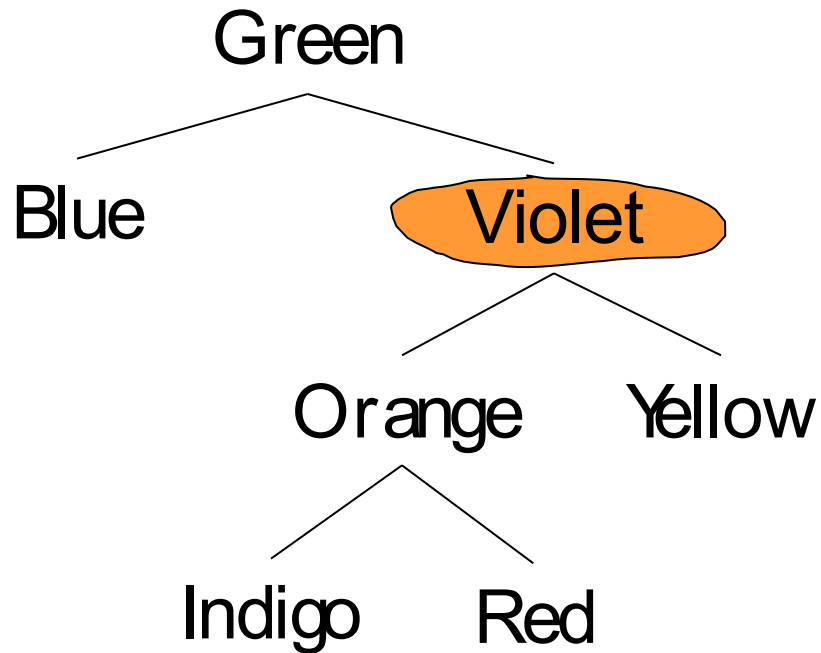
G

# Level-Order Traversal



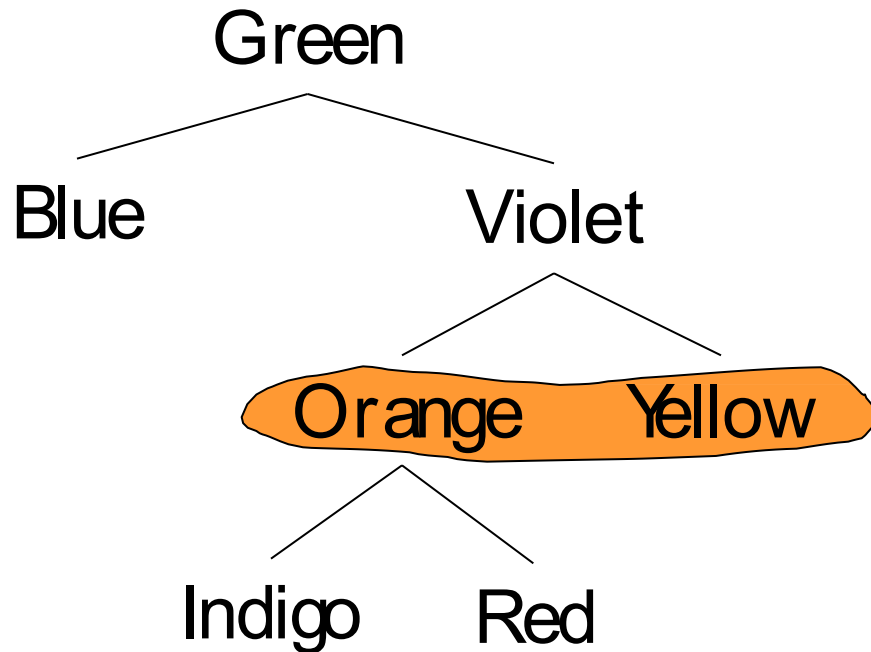
G

# Level-Order Traversal



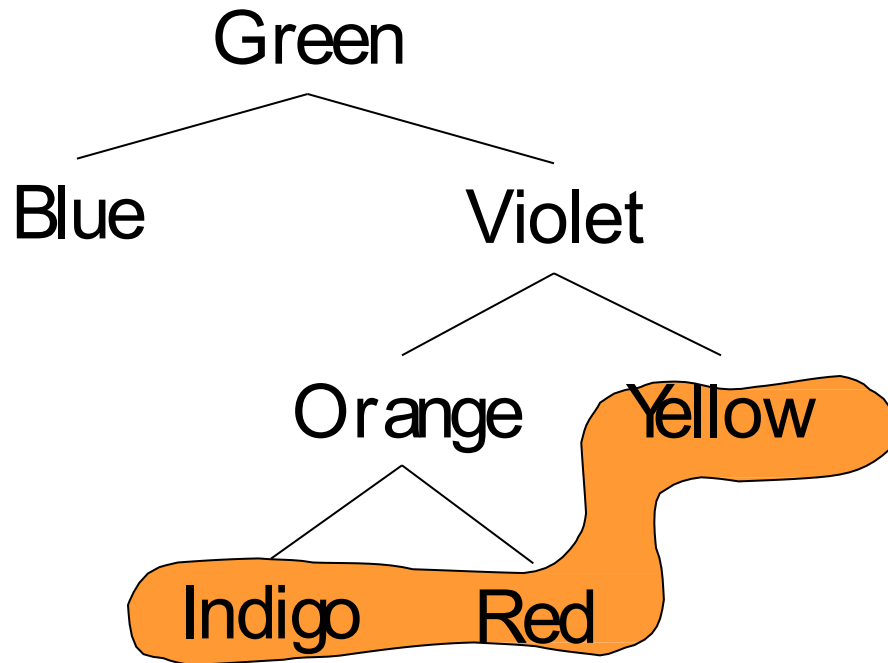
G B

# Level-Order Traversal



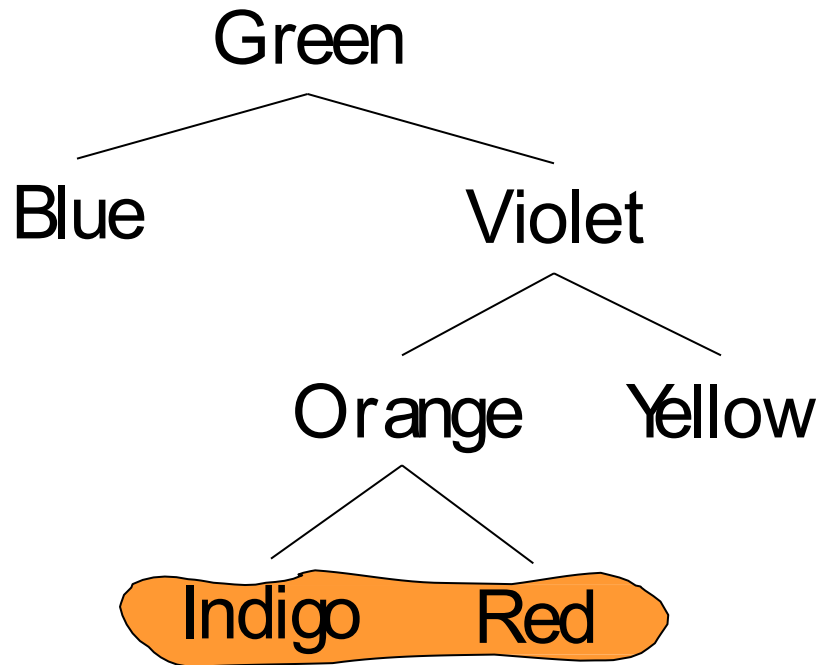
G B V

# Level-Order Traversal



G B V O

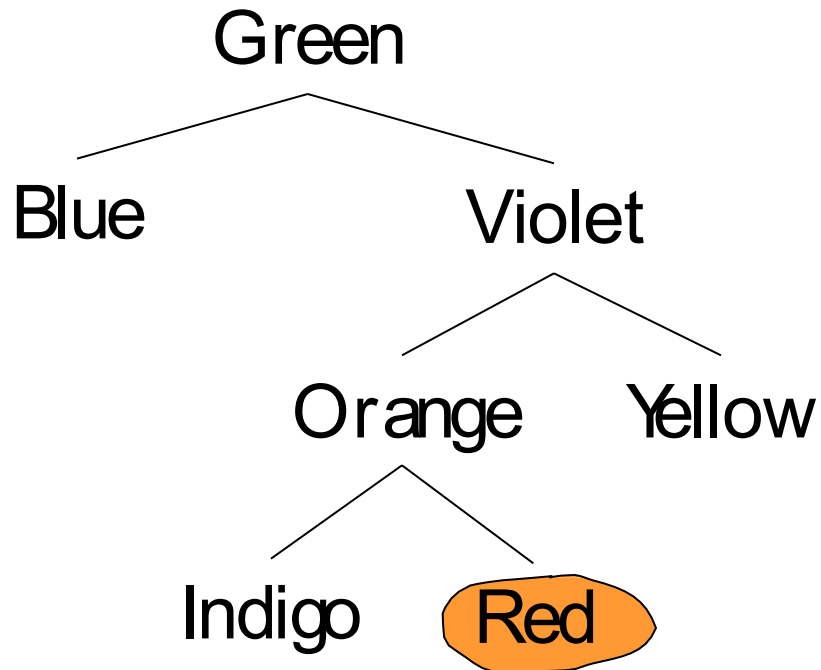
# Level-Order Traversal



GBVOY

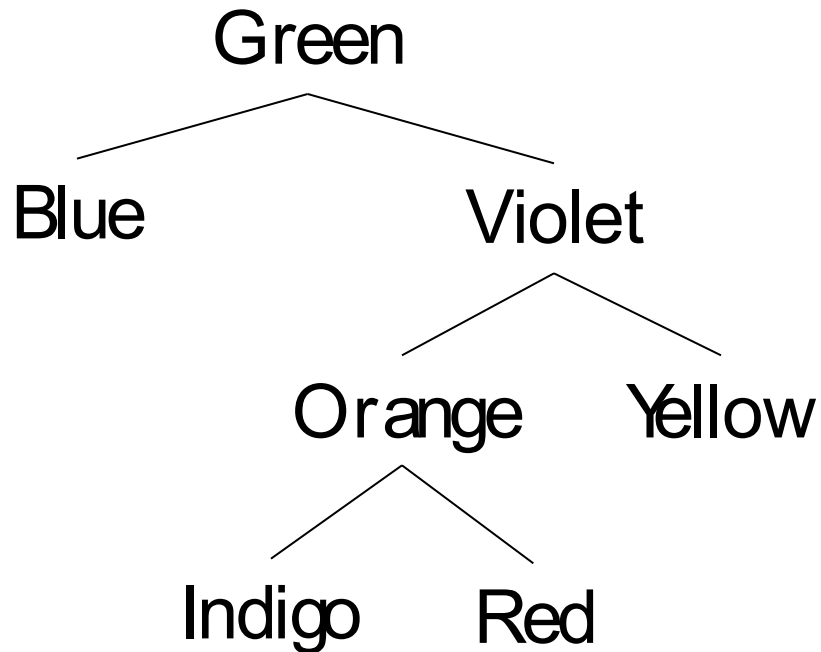


# Level-Order Traversal



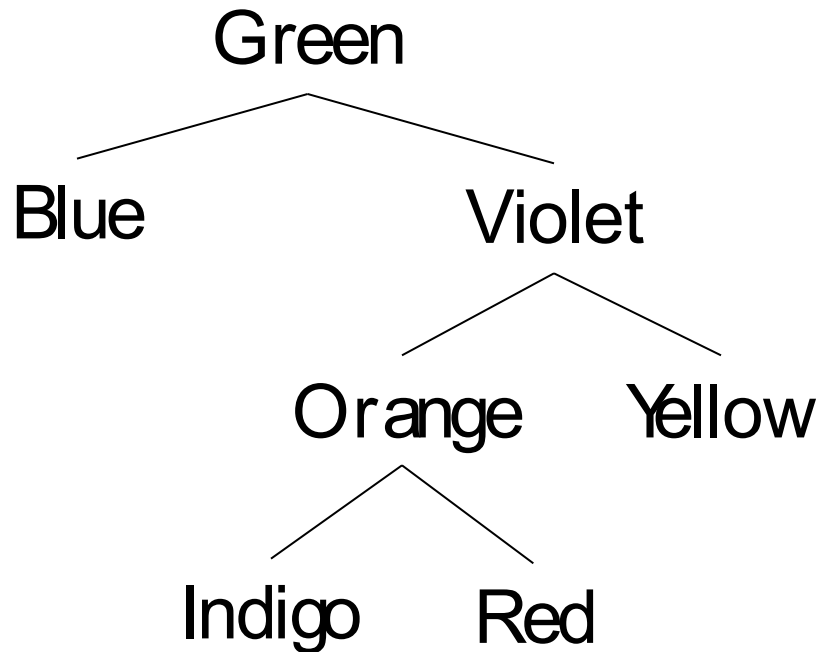
G B V O Y I

# Level-Order Traversal

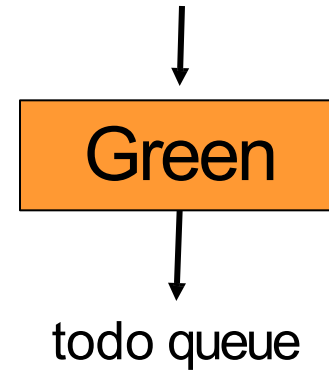
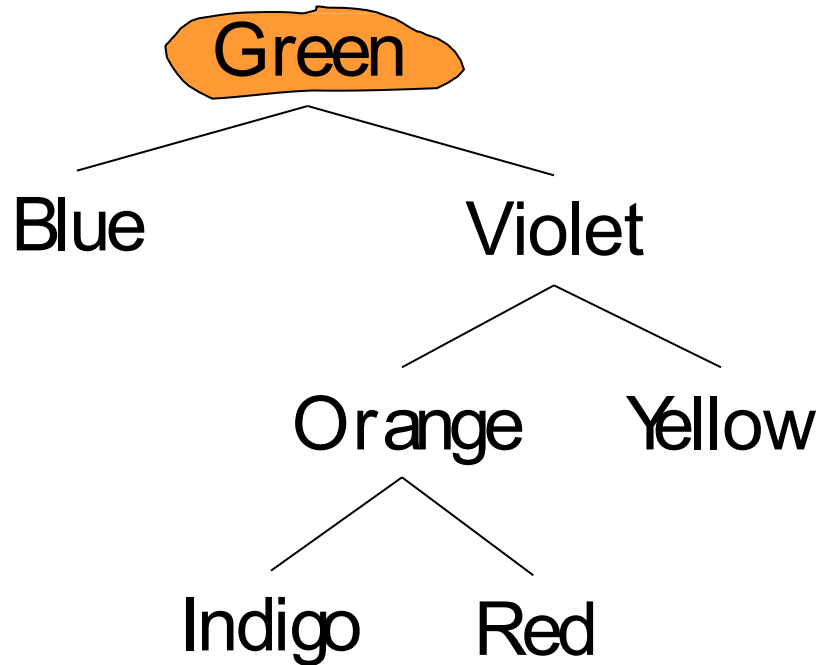


GBVOYIR

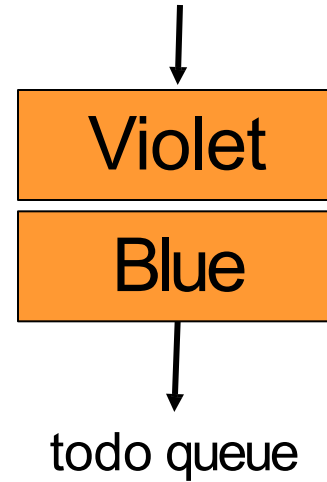
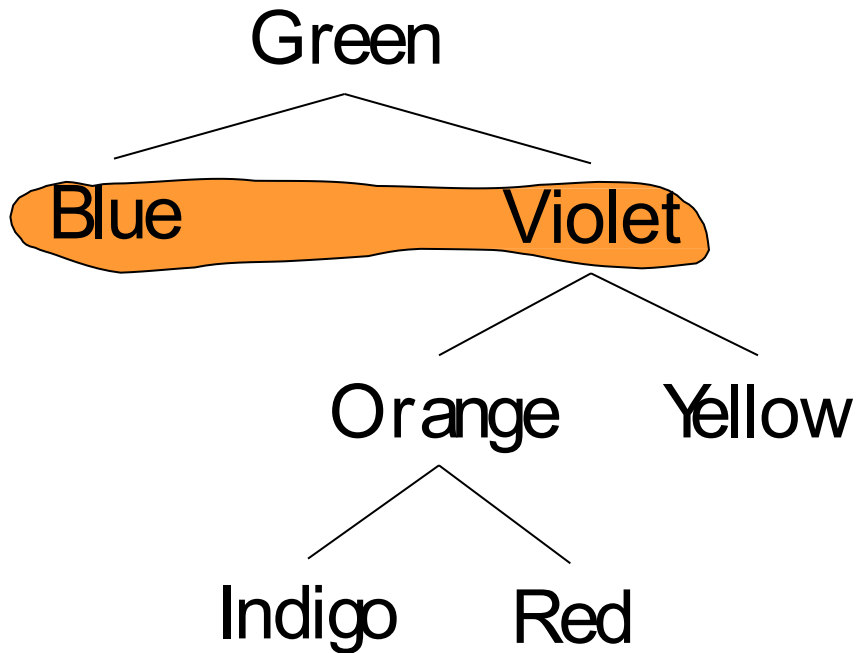
# Level-Order Traversal



# Level-Order Traversal

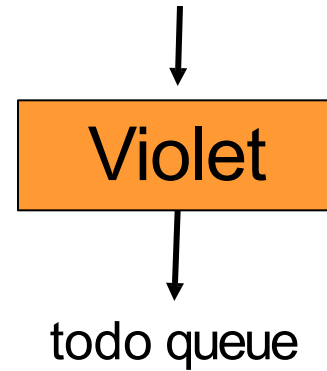
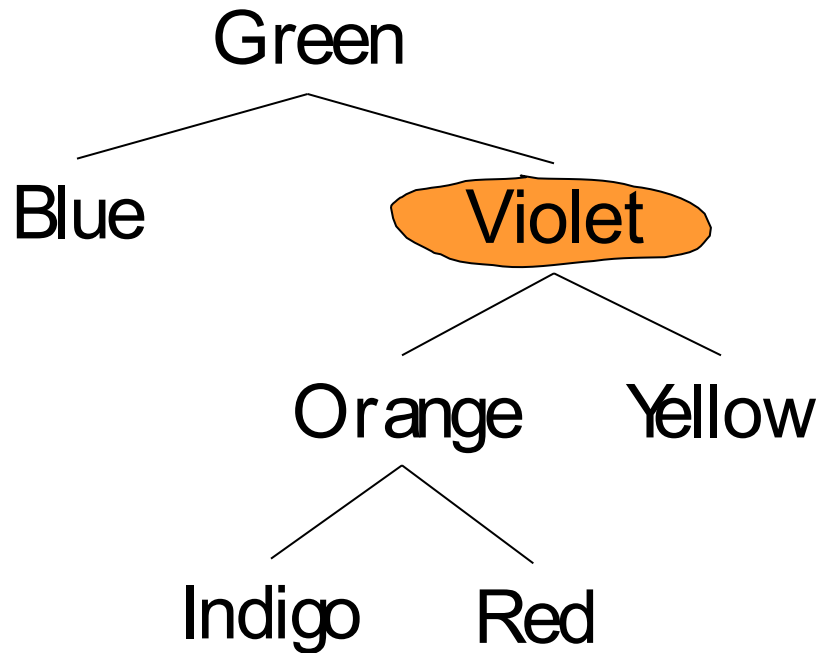


# Level-Order Traversal



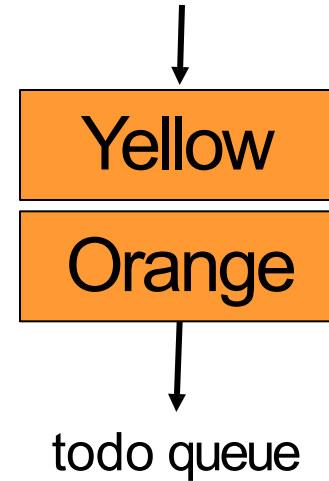
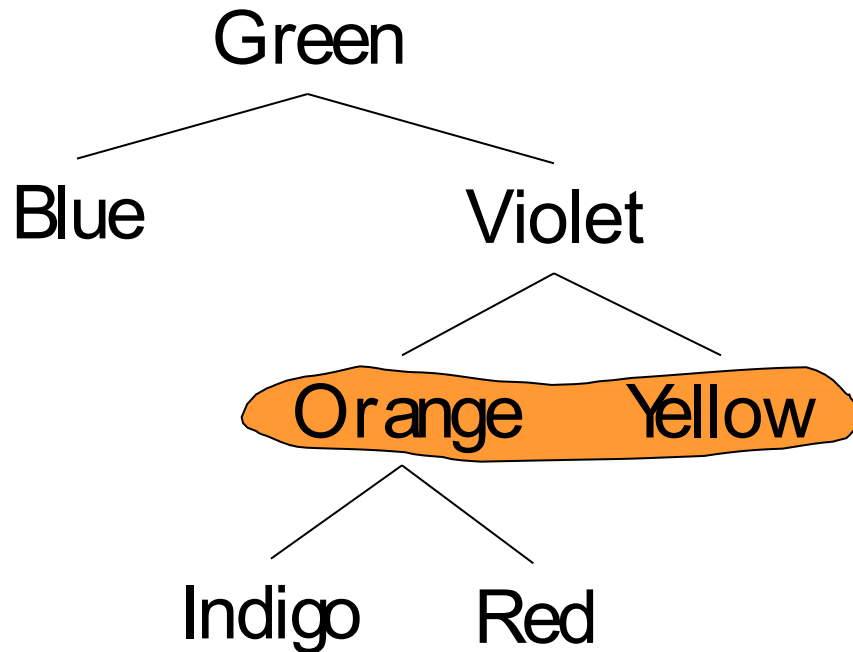
G

# Level-Order Traversal



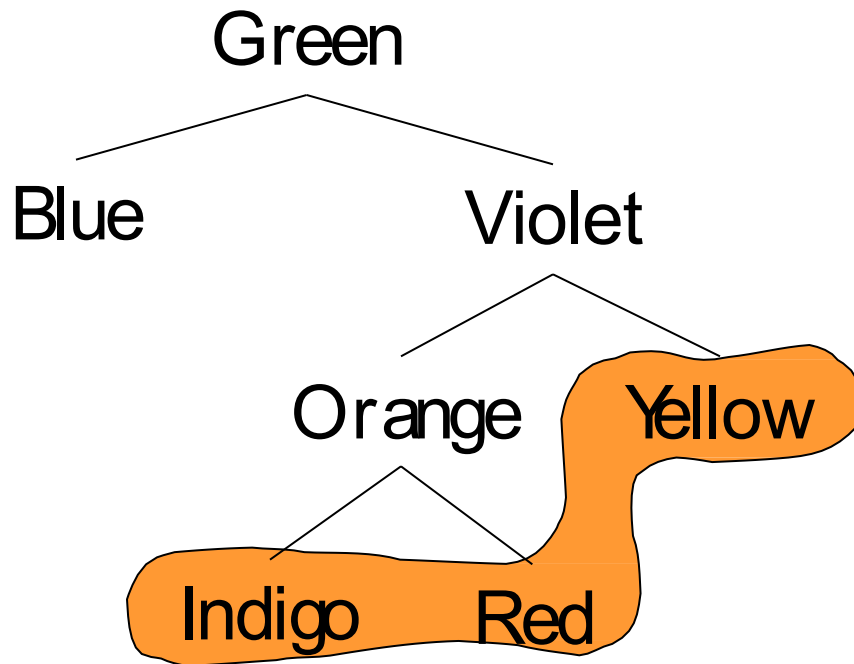
G B

# Level-Order Traversal



G B V

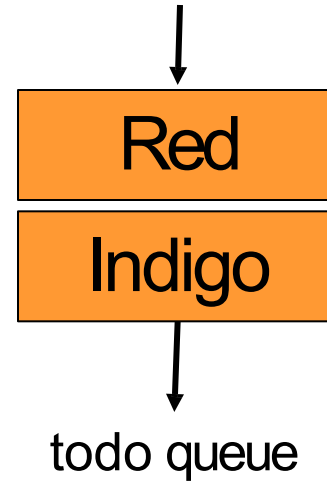
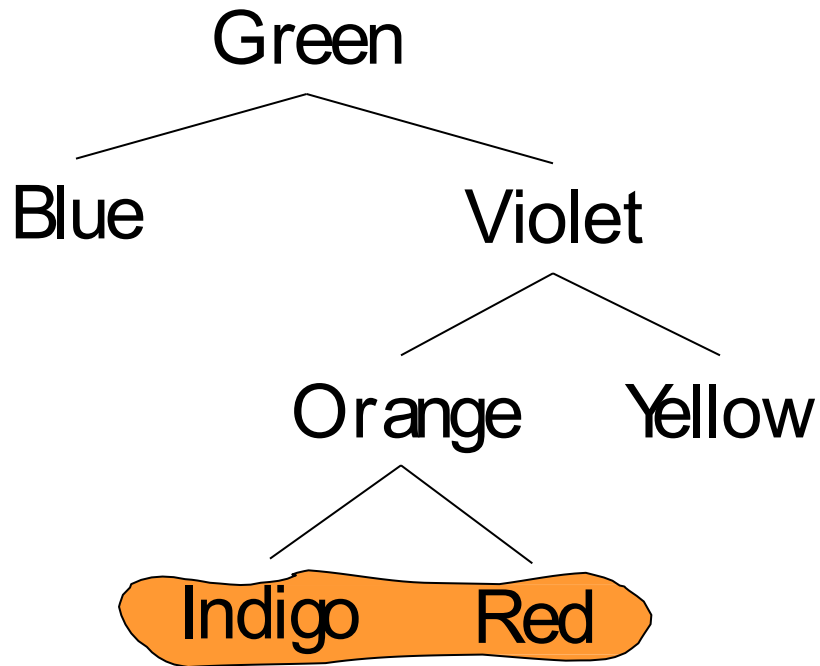
# Level-Order Traversal



GBVO

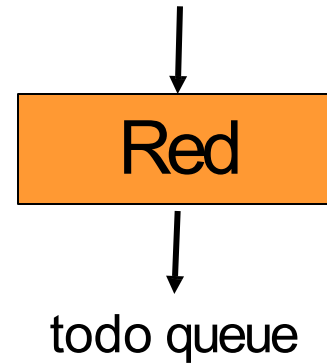
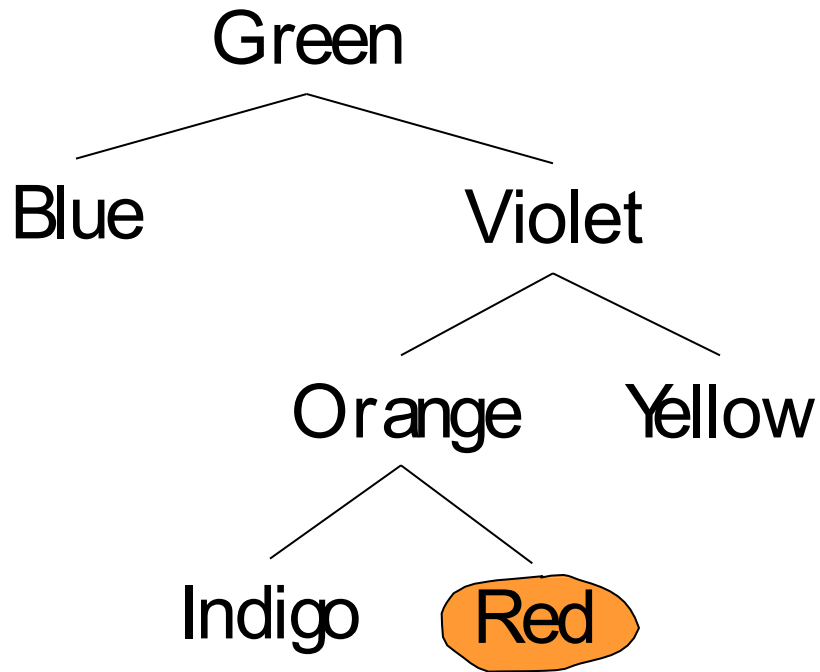


# Level-Order Traversal



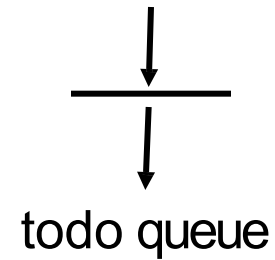
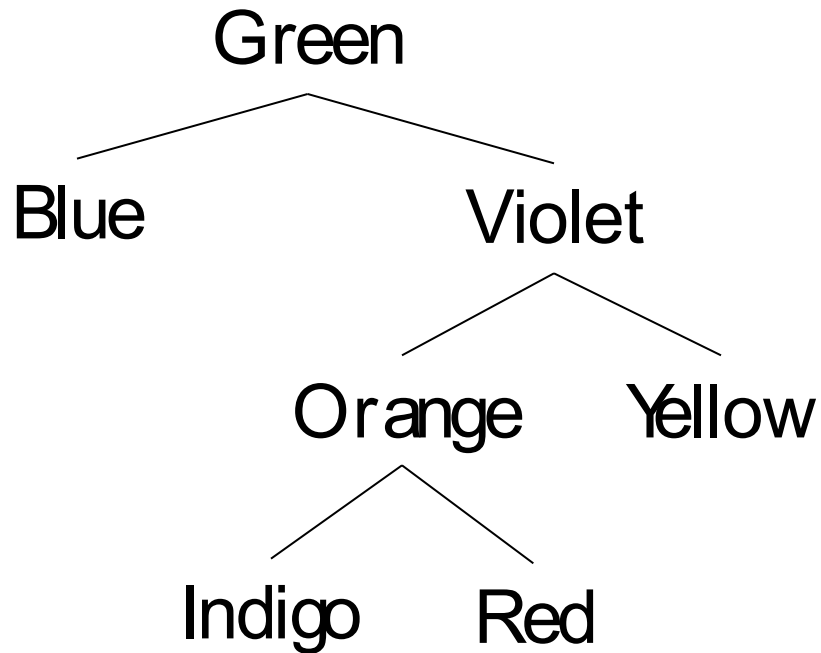
GBVOY

# Level-Order Traversal



G B V O Y I

# Level-Order Traversal



GBVOYIR

# Level-Order Tree Traversal

```
public static <E> void levelOrder(BinaryTree<E> t) {
    if (t.isEmpty()) return;

    // The queue holds nodes for in-order processing
    Queue<BinaryTree<E>> q = new QueueList<BinaryTree<E>>();
    q.enqueue(t); // put root of tree in queue

    while(!q.isEmpty()) {
        BinaryTree<E> next = q.dequeue();
        process(next);
        if(!next.left().isEmpty() ) q.enqueue( next.left() );
        if(!next.right().isEmpty() ) q.enqueue(next.right());
    }
}
```

# Pre-Order Tree Traversal

```
public static <E> void preOrder(BinaryTree<E> t) {
    if (t.isEmpty()) return;

    // The stack holds nodes for in-order processing
    Stack<BinaryTree<E>> st = new StackList<BinaryTree<E>>();
    st.push(t); // put root of tree in stack

    while(!st.isEmpty()) {
        BinaryTree<E> next = st.pop();
        process(next);
        if(!next.right().isEmpty() ) st.push(next.right());
        if(!next.left().isEmpty() ) st.push( next.left() );
    }
}
```

# Summary & Observations

We've seen 4 reasonable traversal methods for trees

They can be efficiently implemented using

- A queue to guide a level-order traversal, or
- A stack to guide a pre-order traversal
  - By storing different information on the stack, we can turn our pre-order traversal into either a post-order or an in-order traversal.