CSCI 136
Data Structures &
Advanced Programming

Lecture 21

Spring 2018

Profs Bill & Jon

Administrative Details

- Lab 7 posted
  - Two towers
  - Use iterators to solve a challenging problem
  - Bitwise operations help

Last Time

- Trees!
  - General Idea and Uses
  - Terminology
  - Some examples
    - Expression trees

Today

- The `structure5` `BinaryTree` class
  - implementation details
- Some quick proofs and theory
- Traversing trees

Branching Out: Trees

- A tree is a data structure where elements can have multiple successors (called children)
- But still only one predecessor (called parent)

Tree Features

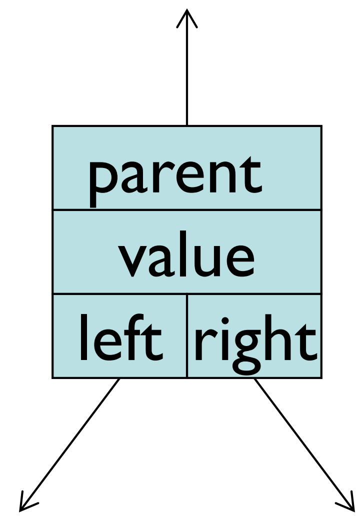
- Trees express hierarchical relationships
  - Directed: root to leaf
- Root at the top
- Leaf at the bottom
- Interior nodes in middle
- Parent, children, ancestors, descendants, siblings
- Degree (of node): number of children of node
- Degree (of tree): maximum degree (across all nodes)
- Depth of node: number of *edges* from root to node
- Height: maximum depth (across all nodes)

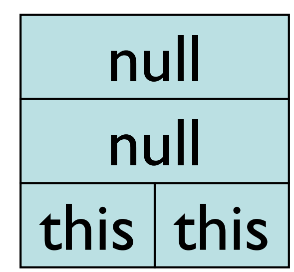
Introducing Binary Trees

- Degree of each node <= 2
- Recursively defined. A tree can either be:
  - Empty
  - Root with left and right subtrees
- Binary Tree: No "inner" node class like SLL; single `BinaryTree` class does it all
- (Not part of the `structure` inheritance hierarchy)

Implementing structure5 BinaryTree

- BinaryTree<E> class
  - Instance variables
    - BinaryTree: parent, left, right
    - E: value
- left and right are *never* null
  - If no child, they point to an "empty" tree
    - Empty tree T has value null, parent null, left = right = T
  - Only empty tree nodes have null value

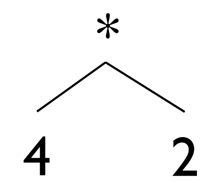


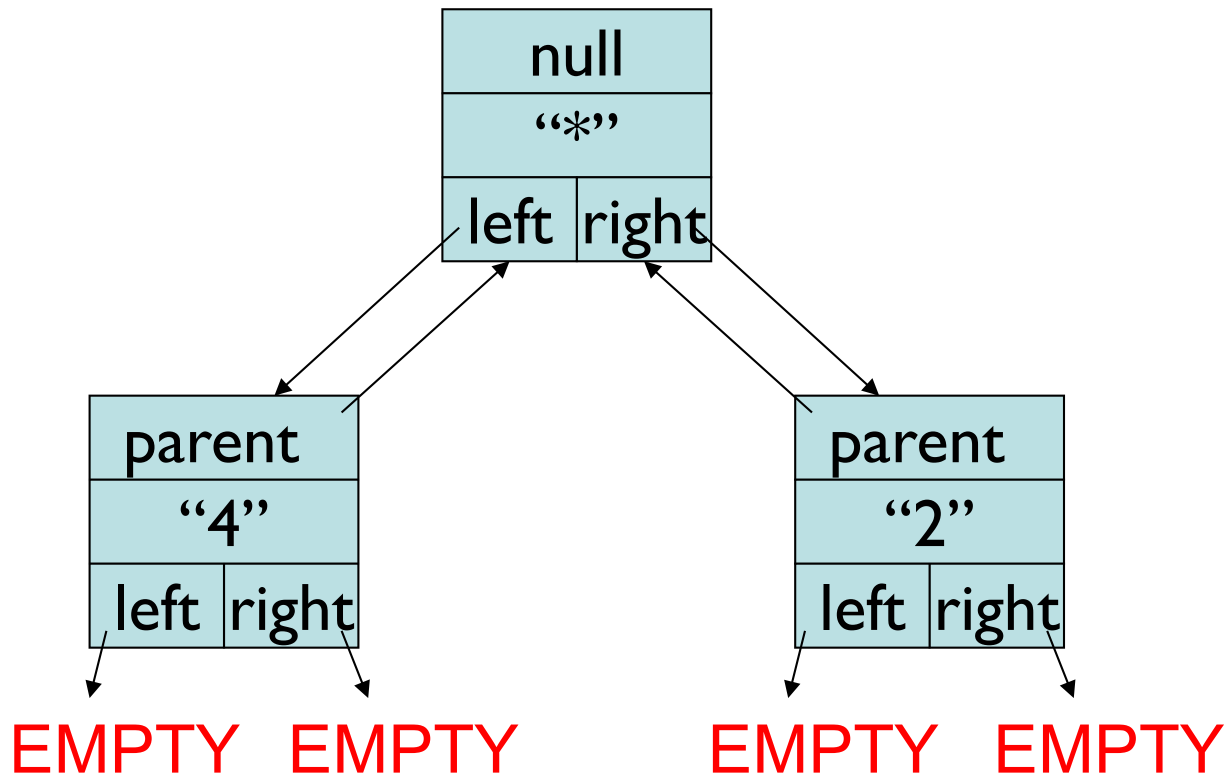


EMPTY BT

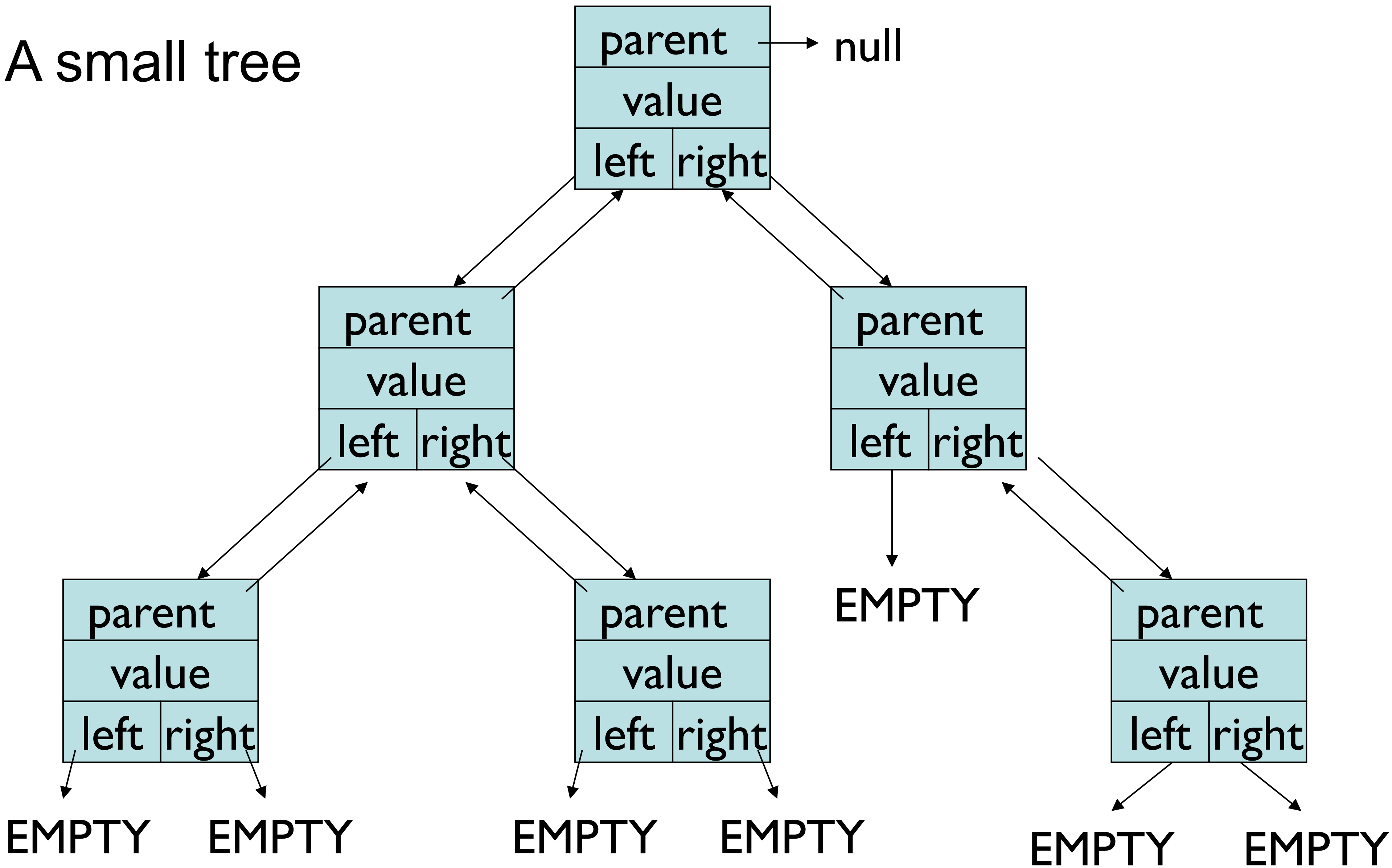
Implementing BinaryTree

- BinaryTree class
  - Instance variables
    - BT parent, BT left, BT right, E value





A small tree



EMPTY != null!

Implementing BinaryTree

- Many (!) methods: See BinaryTree javadoc page
- All "left" methods have equivalent "right" methods
  - public BinaryTree()
    - // generates an empty node (EMPTY)
    - // parent and value are null, left=right=this
  - public BinaryTree(E value)
    - // generates a tree with a non-null value and two empty (EMPTY) subtrees
  - public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right)
    - // returns a tree with a non-null value and two subtrees
  - public void setLeft(BinaryTree<E> newLeft)
    - // sets left subtree to newLeft
    - // re-parents newLeft by calling newLeft.setParent(this)
  - protected void setParent(BinaryTree<E> newParent)
    - // sets parent subtree to newParent
    - // called from setLeft and setRight to keep all "links" consistent

Implementing BinaryTree

- Methods:
  - public BinaryTree<E> left()
    - // returns left subtree
  - public BinaryTree<E> parent()
    - // post: returns reference to parent node, or null
  - public boolean isLeftChild()
    - // returns true if this is a left child of parent
  - public E value()
    - // returns value associated with this node
  - public void setValue(E value)
    - // sets the value associated with this node
  - public int size()
    - // returns number of (non-empty) nodes in tree
  - public int height()
    - // returns height of tree rooted at this node
  - But where's "remove" or "add"?!?!

BT Questions/Proofs

- Prove
  - The number of nodes at depth n is at most 2^n.
  - The number of nodes in tree of height n is at most $2^{(n+1)}-1$.
  - A tree with n nodes has exactly n-1 edges

BT Questions/Proofs

Prove: Number of nodes at depth $d \geq 0$ is at most 2^d.

Idea: Induction on depth d of nodes of tree

Base case: d= 0: 1 node. $1 = 2^0$ ✓

Induction Hyp.: For some $d \geq 0$, there are at most 2^d nodes at depth d.

Induction Step: Consider depth $d+1$. It has at most 2 nodes for every node at depth d.

Therefore it has at most $2*2^d = 2^{d+1}$ nodes ✓

BT Questions/Proofs

Prove that any tree of $n \geq 1$ nodes has $n-1$ edges

Idea: Induction on number of nodes

Base case: $n = 1$. There are no edges ✓

Induction Hyp: Assume that, for some $n \geq 1$, every tree of n nodes has exactly $n-1$ edges.

Induction Step: Let T have $n+1$ nodes. Show it has exactly n edges.

- Remove a leaf v (and its single edge) from T
- Now T has n nodes, so it has n-1 edges
- Now add v (and its single edge) back, giving n+1 nodes and n edges.

Representing Knowledge

- Trees can be used to represent knowledge
  - Example: InfiniteQuestions game
- We often call these trees decision trees
  - Leaf: object
  - Internal node: question to distinguish objects
- Move down decision tree until we reach a leaf node
- Check to see if the leaf is correct
  - If not, add another question, make new and old objects children
- Let's play....

Building Decision Trees

- Gather/obtain data
- Analyze data
  - Make greedy choices: Find good questions that divide data into halves (or as close as possible)
- Construct tree with shortest height
- In general this is a *hard* problem!
- Example

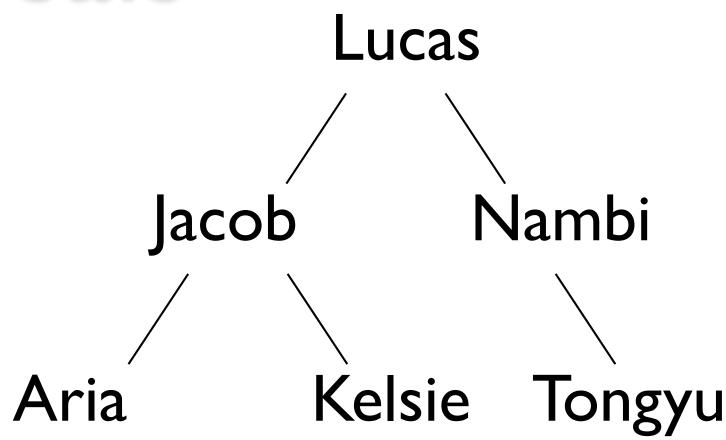
Representing Arbitrary Trees

- What if nodes can have many children?
  - Example: Game trees
- Replace left/right node references with a list of children (Vector, SLL, etc)
  - Allows getting "i^{th}" child
- Should provide method for getting degree of a node
- Degree 0   Empty list   No children   Leaf
- We will use this idea in the Lexicon Lab

Tree Traversals

- In linear structures, there are only a few basic ways to traverse 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

Tree Traversals



In-order: "left, node, right"

Aria, Jacob, Kelsie, Lucas, Nambi, Tongyu

Pre-order: "node, left, right"

Lucas, Jacob, Aria, Kelsie, Nambi, Tongyu

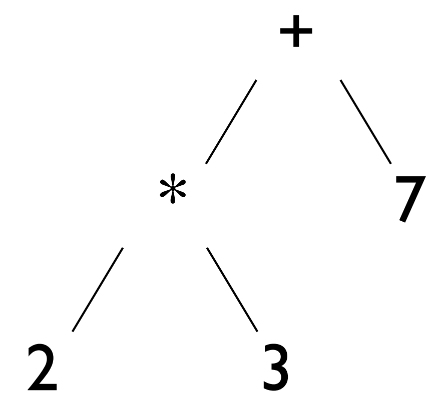
Post-order: "left, right, node"

Aria, Kelsie, Jacob, Tongyu, Nambi, Lucas,

Level-order: visit all nodes at depth i before depth i+1

Lucas, Jacob, Nambi, Aria, Kelsie, Tongyu

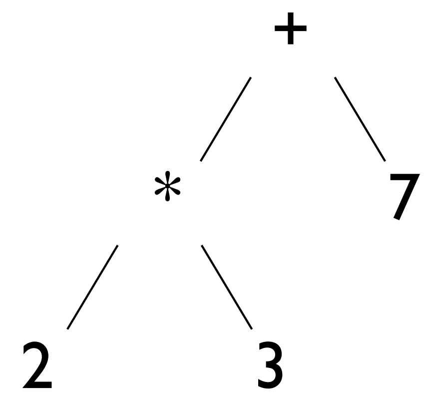
Tree Traversals



- Pre-order
  - Each node is visited before any children. Visit node, then each node in left subtree, then each node in right subtree. (node, left, right)
    - +*237
- In-order
  - Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree. (left, node, right)
    - 2*3+7

("pseudocode")

Tree Traversals

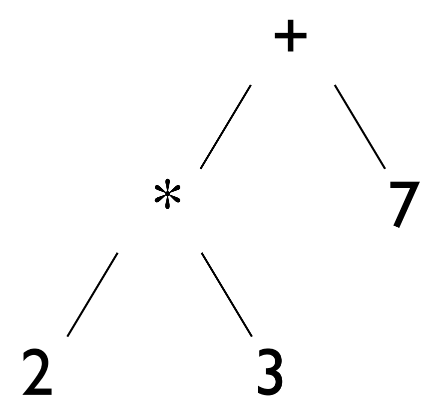


- Post-order
  - Each node is visited after its children are visited. Visit all nodes in left subtree, then all nodes in right subtree, then node itself. (left, right, node)
    - 23*7+
- Level-order (not obviously recursive!)
  - All nodes of level i are visited before nodes of level i+1. (visit nodes left to right on each level)
    - +*723

("pseudocode")

Tree Traversals

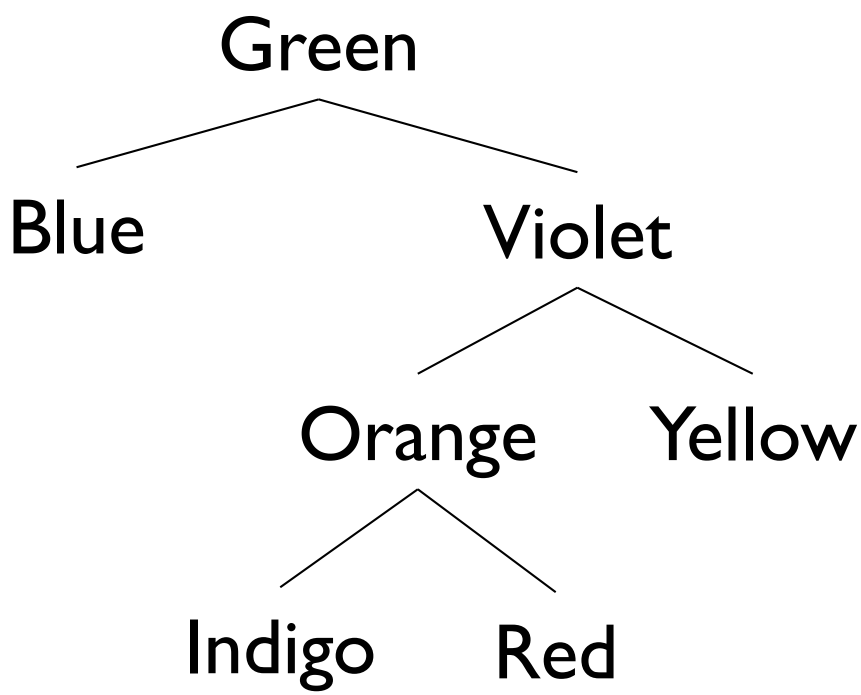
```
public void preOrder(BinaryTree t) {
      if(t.isEmpty()) return;
      touch(t); // some method
      preOrder(t.left());
      preOrder(t.right());
}
```



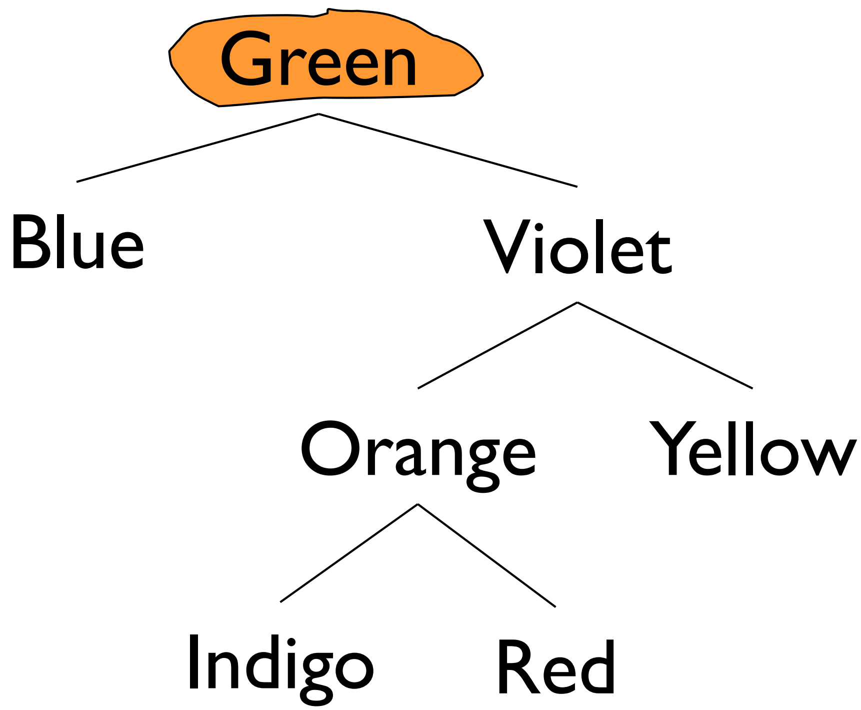
For in-order and post-order: just move touch(t)!

But what about level-order???

Level-Order Traversal

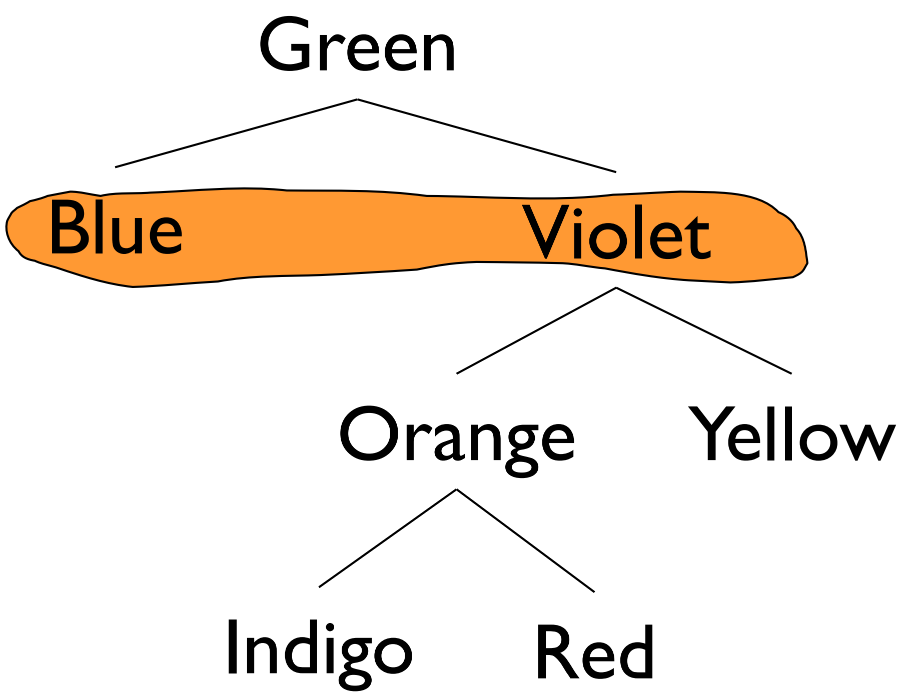


Level-Order Traversal



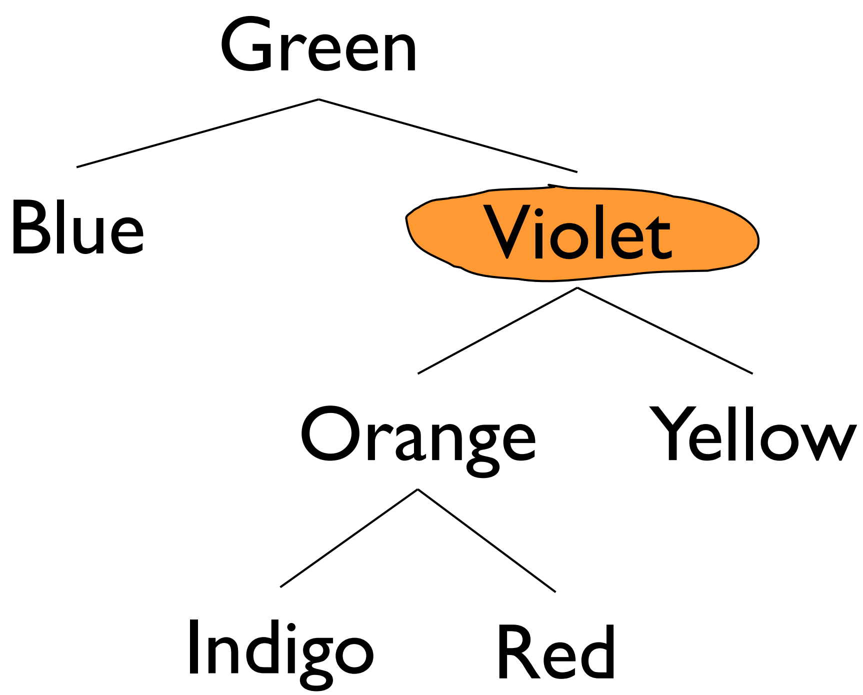
G

Level-Order Traversal



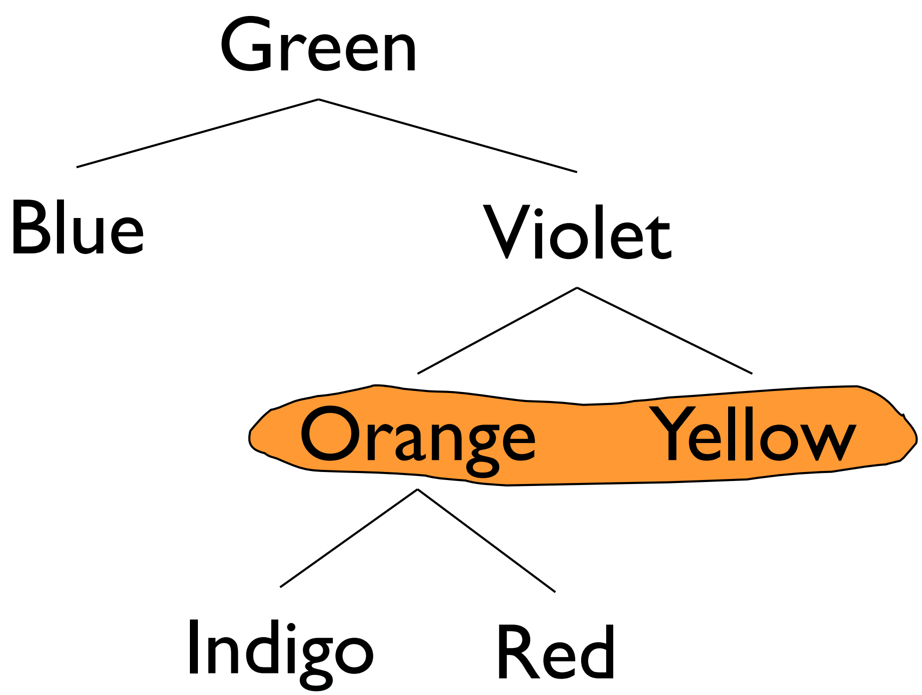
G

Level-Order Traversal



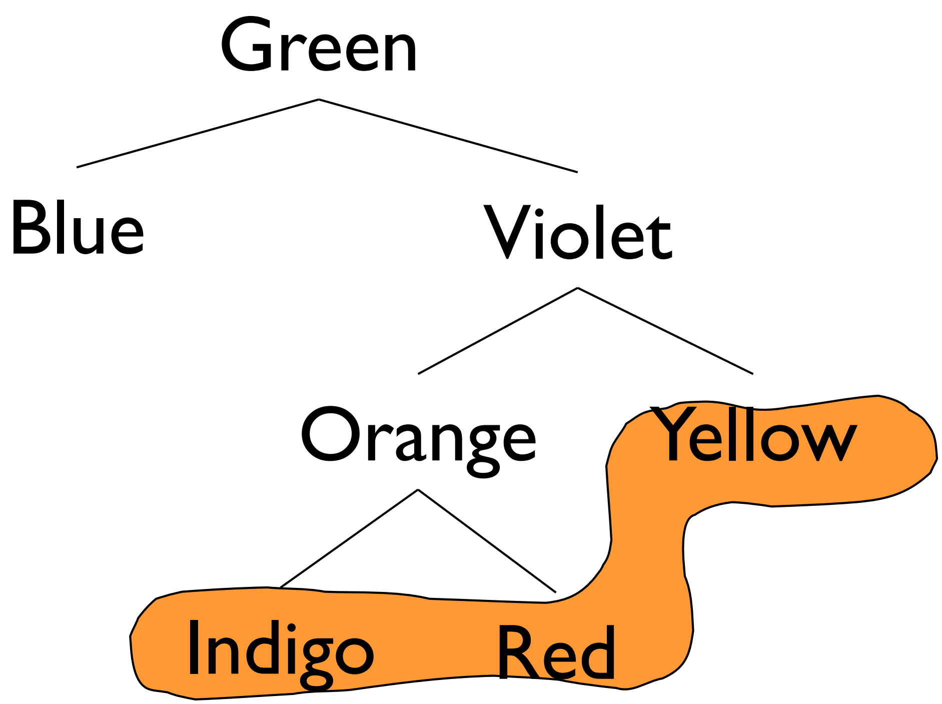
G B

Level-Order Traversal



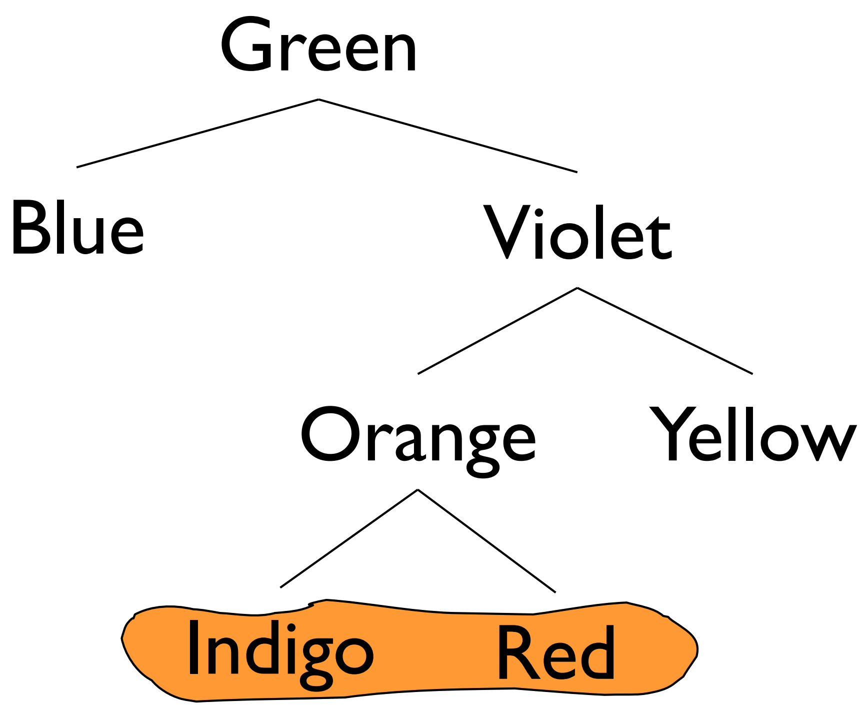
G B V

Level-Order Traversal



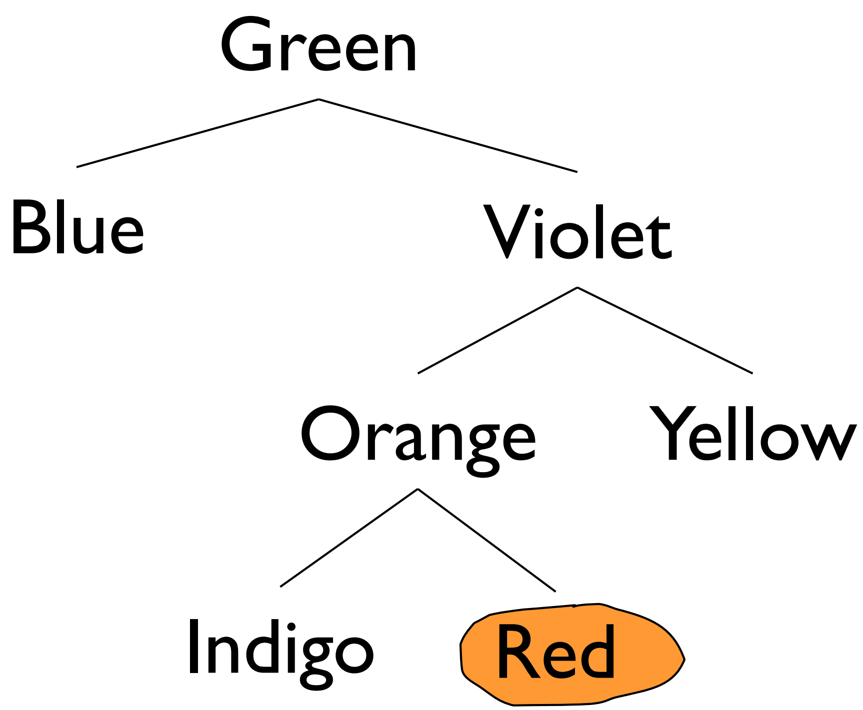
G B V O

Level-Order Traversal



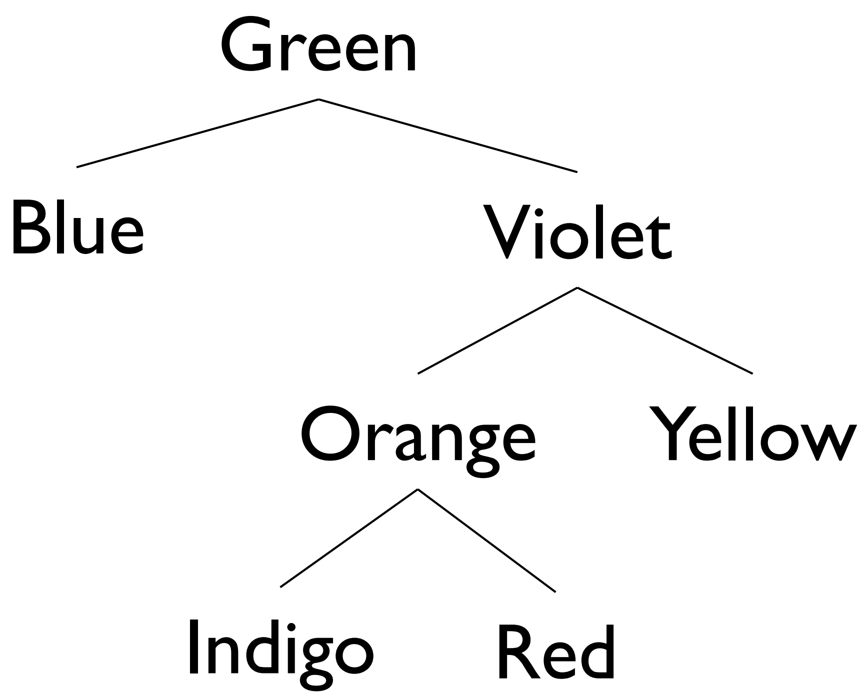
G B V O Y

Level-Order Traversal



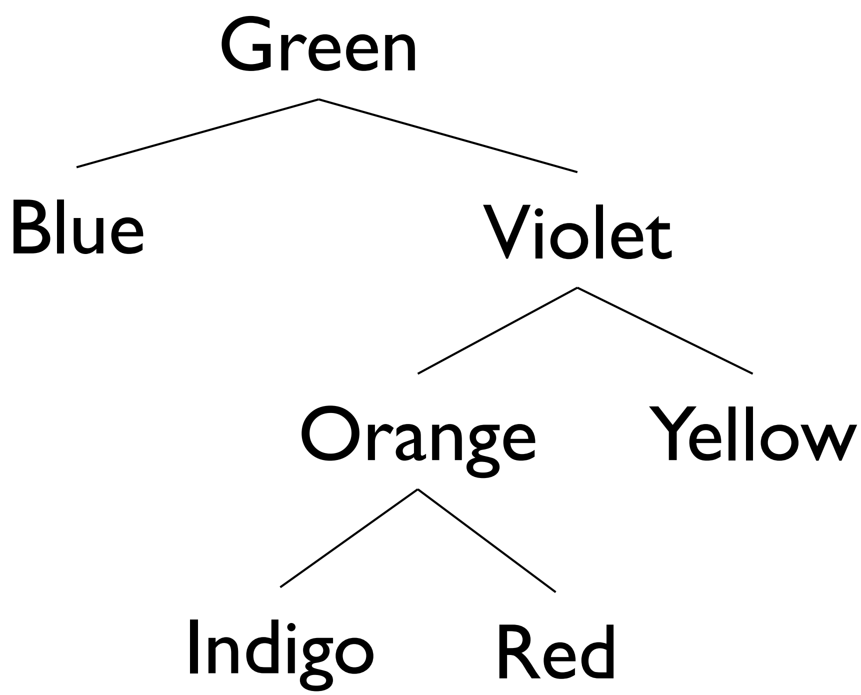
G B V O Y I

Level-Order Traversal

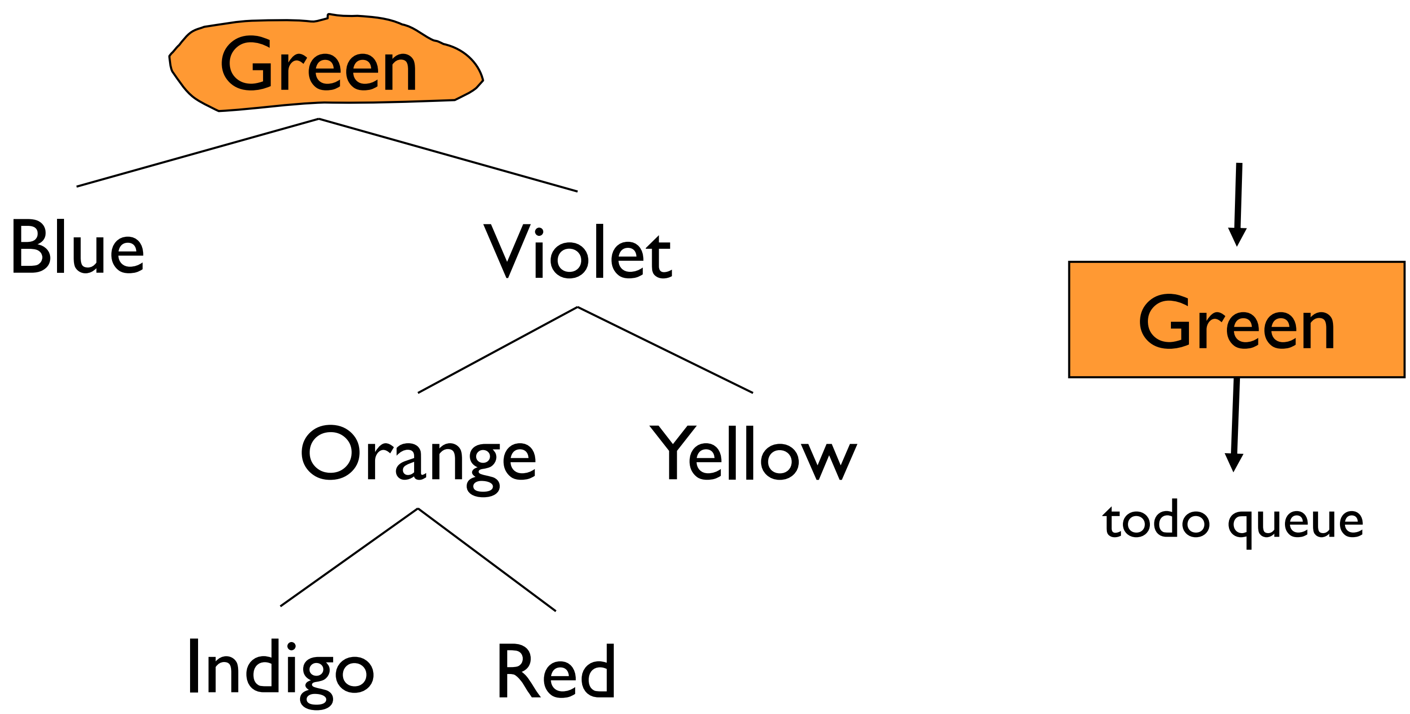


G B V O Y I R

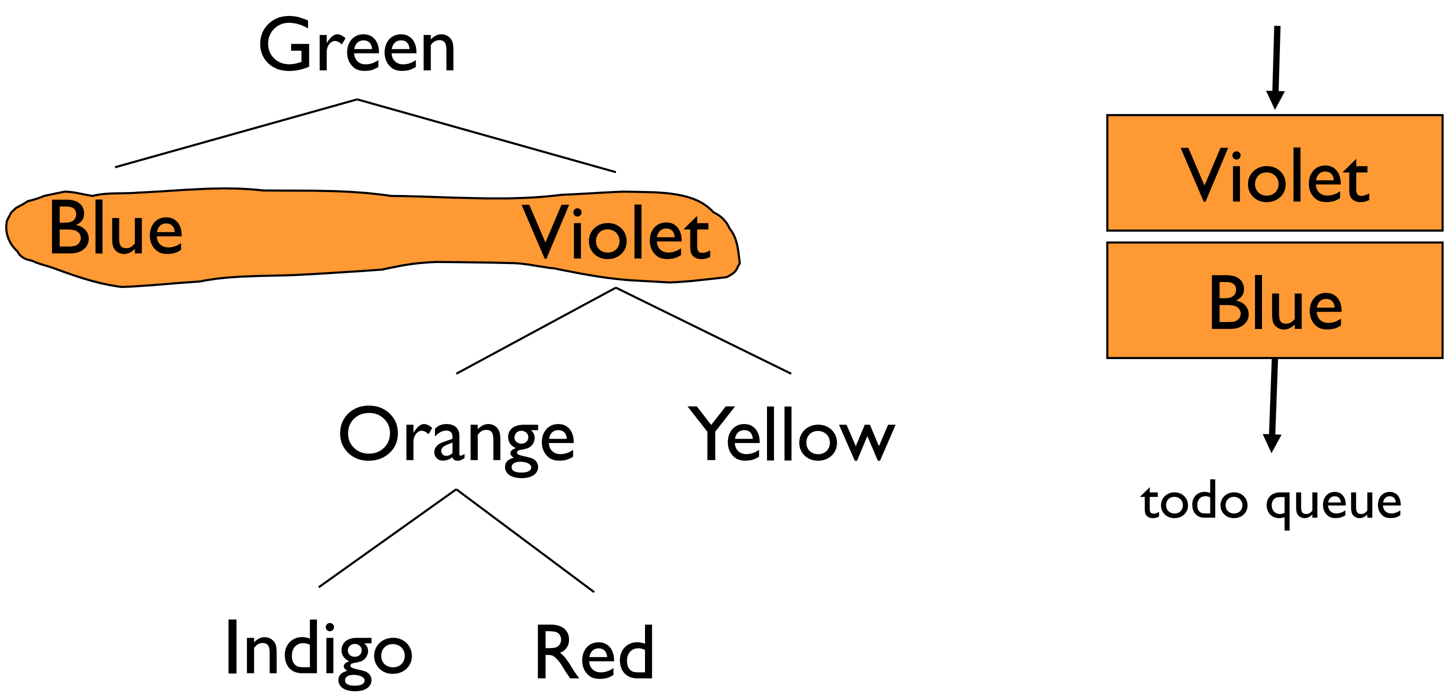
Level-Order Traversal



Level-Order Traversal

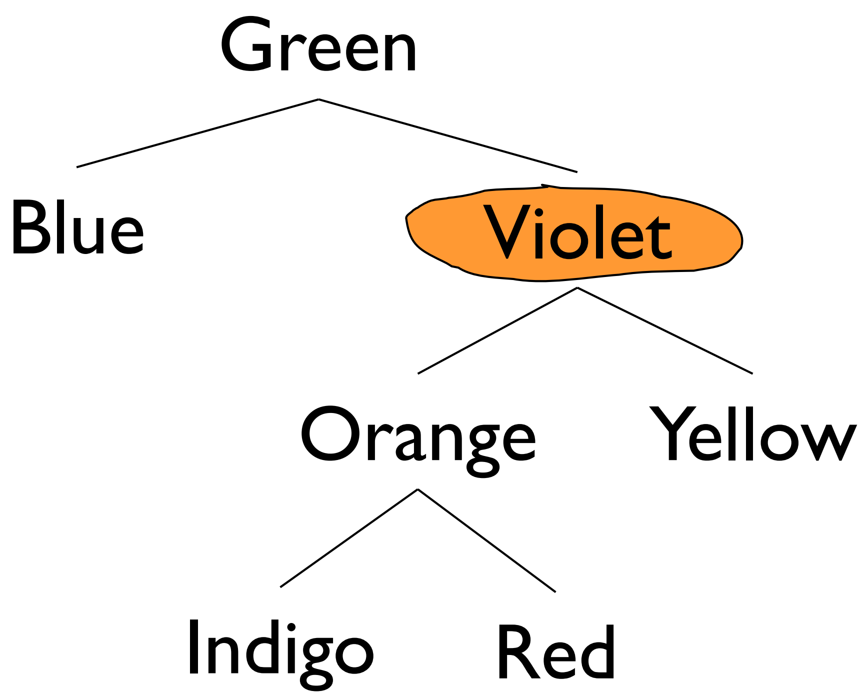


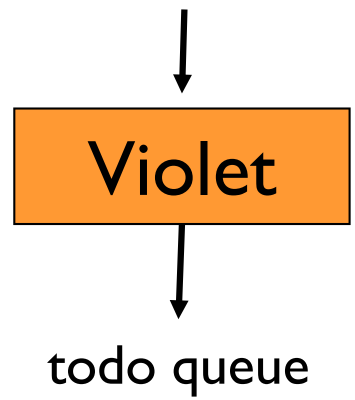
Level-Order Traversal



G

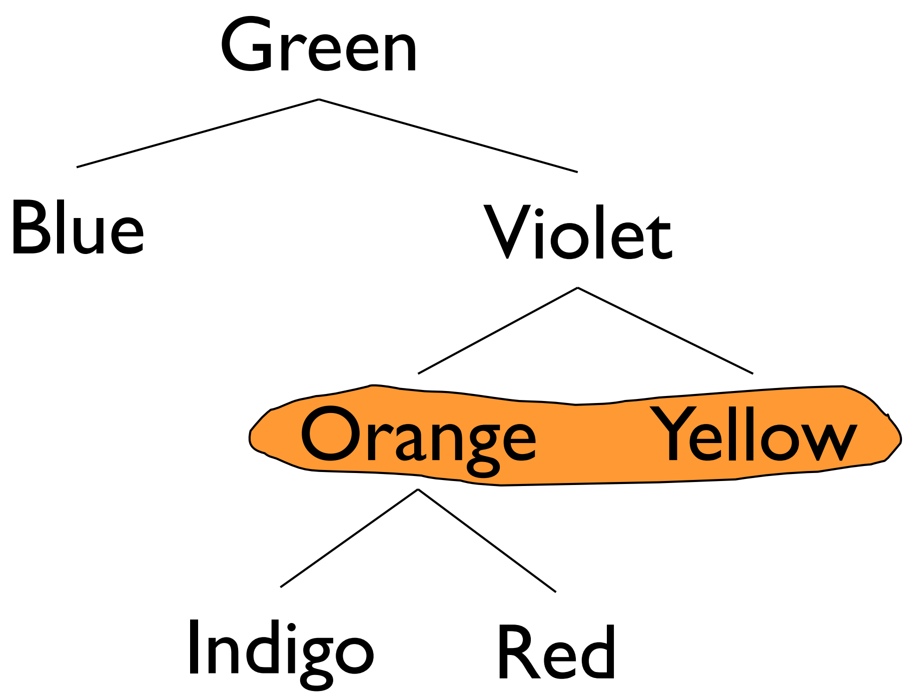
Level-Order Traversal

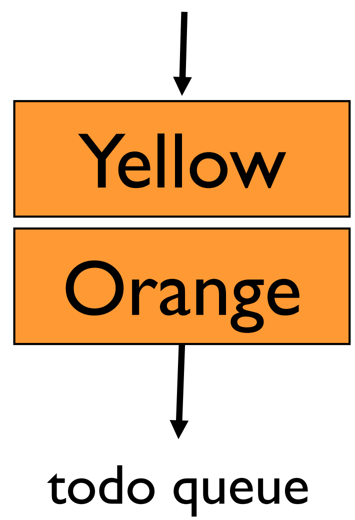




G B

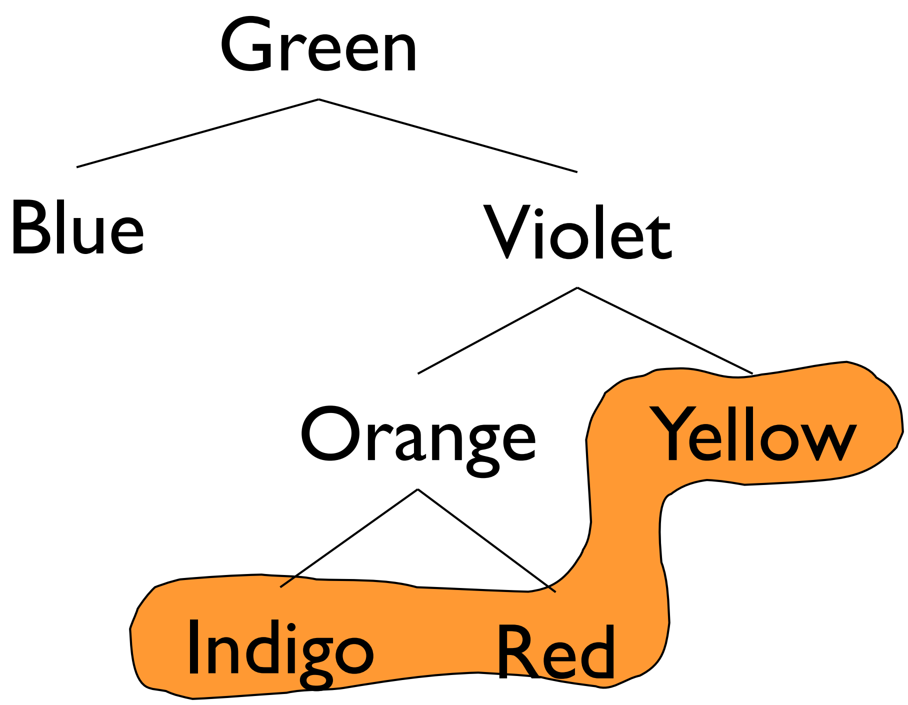
Level-Order Traversal

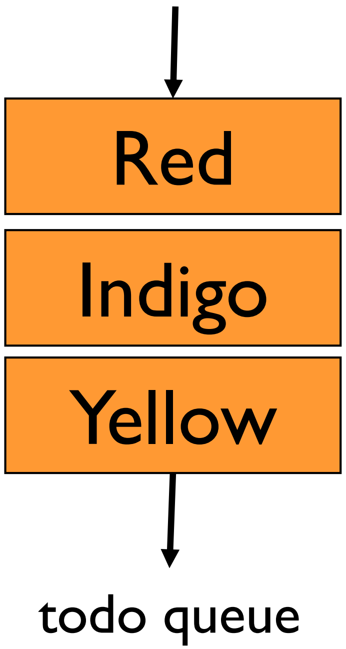




G B V

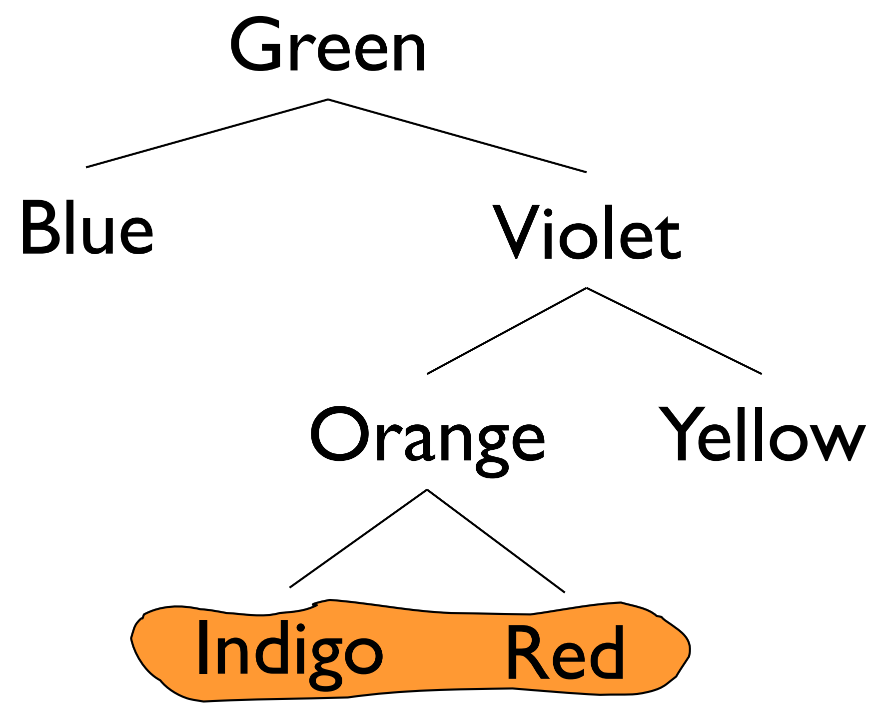
Level-Order Traversal

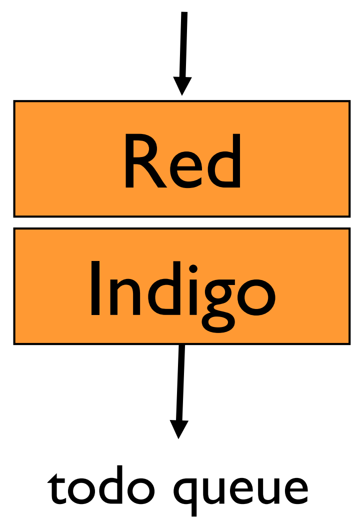




G B V O

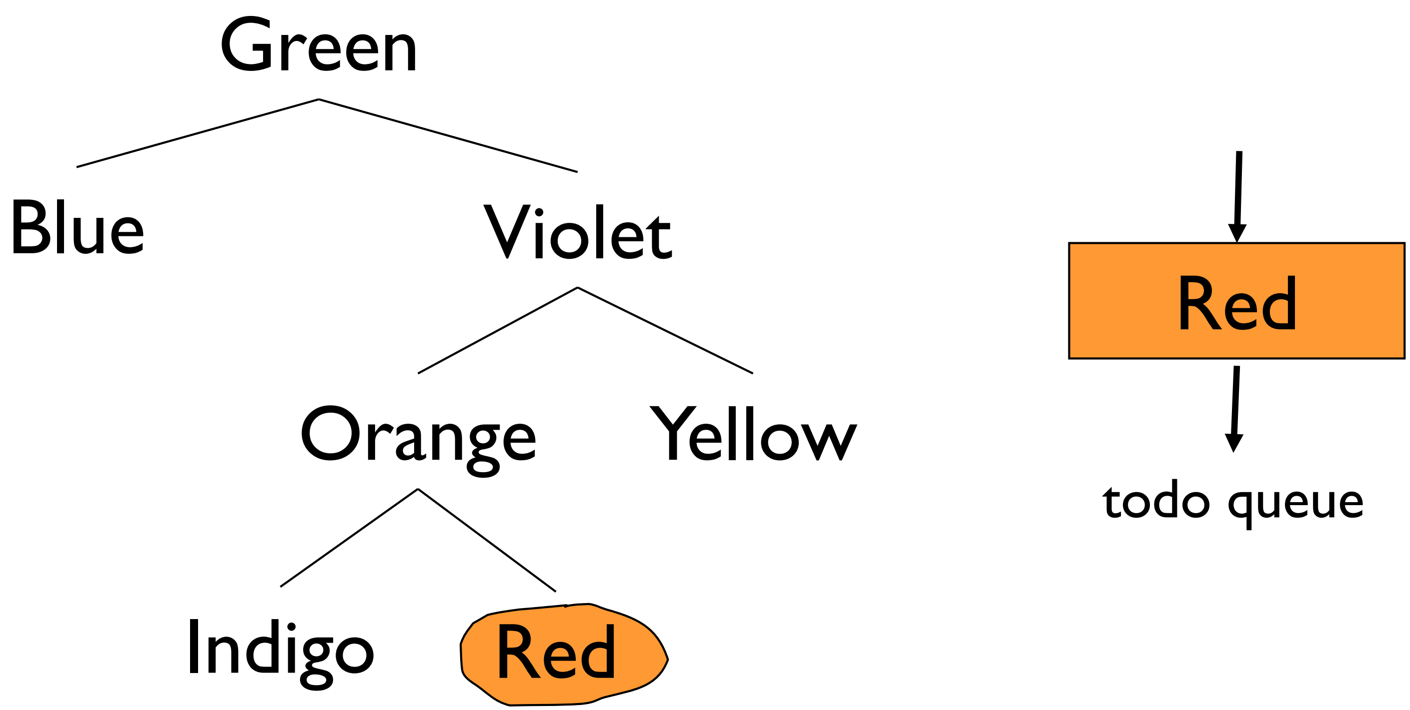
Level-Order Traversal





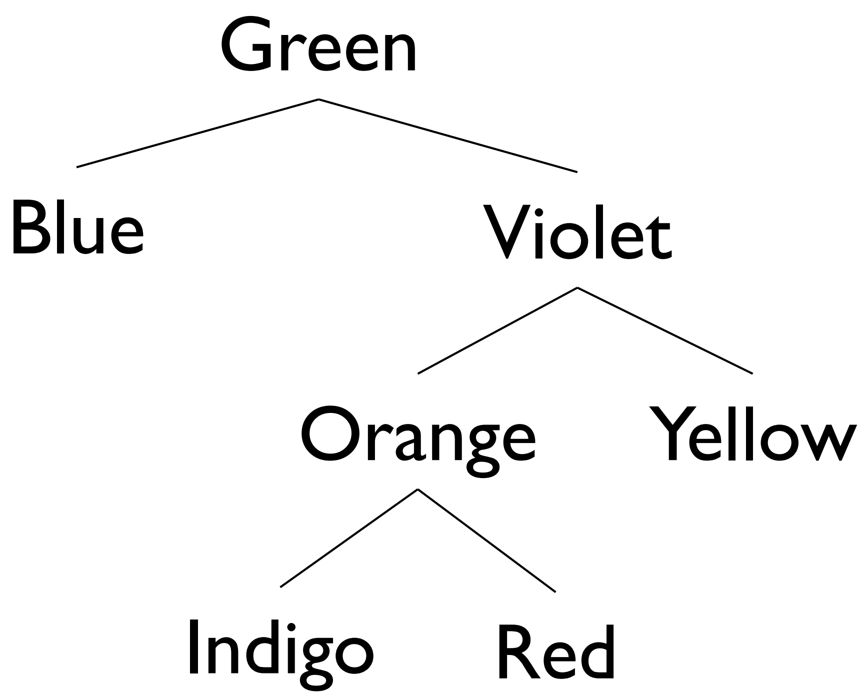
G B V O Y

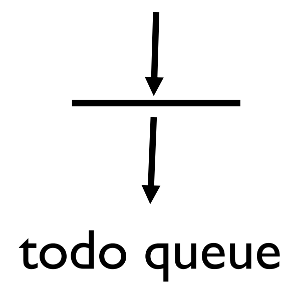
Level-Order Traversal



G B V O Y I

Level-Order Traversal





G B V O Y I R

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();
      touch(next);
      if(!next.left().isEmpty())  q.enqueue( next.left() );
      if(!next.right().isEmpty()) q.enqueue(next.right());
   }
}
```