CSCI 136
Data Structures & Advanced Programming

Lecture 19

Fall 2019

Instructor: B&S

Last Time

- Trees
  - Expression Trees
    - Recursive evaluation
  - Implementation
  - Recursion/Induction on Trees

Today

- Applications: Decision Trees
- Trees with more than 2 children
  - Representations
- Traversing Binary Trees
  - As methods taking a BinaryTree parameter
  - With Iterators

Representing Knowledge

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

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

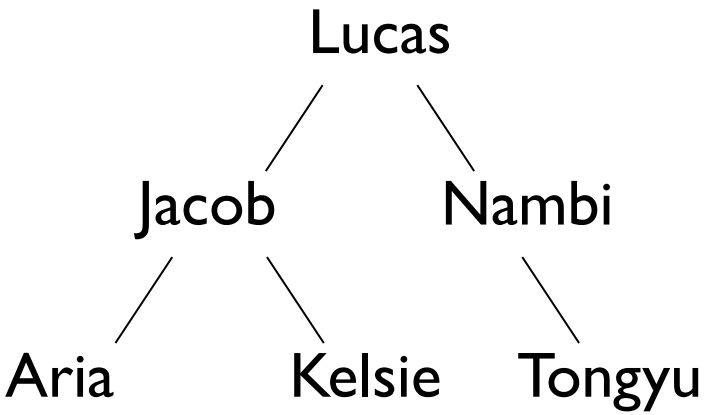
Representing Arbitrary Trees

- 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

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



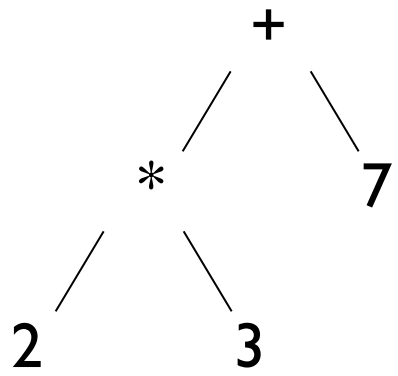
In-order: Aria, Jacob, Kelsie, Lucas, Nambi, Tongyu

Pre-order: Lucas, Jacob, Aria, Kelsie, Nambi, Tongyu

Post-order: Aria, Kelsie, Jacob, Tongyu, Nambi, Lucas,

Level-order: Lucas, Jacob, Nambi, Aria, Kelsie, Tongyu

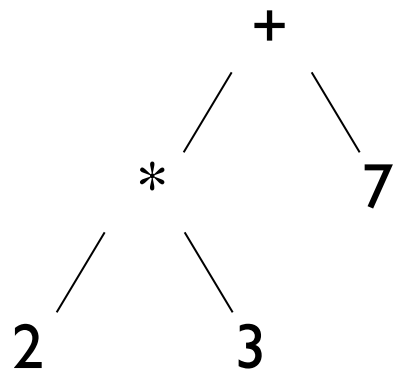
Tree Traversals



- 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

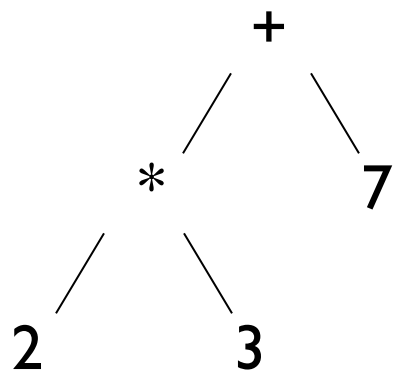


- Post-order
  - Each node is visited after its children are visited. Visit all nodes in left subtree, then all nodes in right subtree, then node itself. (left, right, node)
    - 23*7+
- Level-order (not obviously recursive!)
  - All nodes of level i are visited before nodes of level i+1. (visit nodes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

("pseudocode")

Tree Traversals

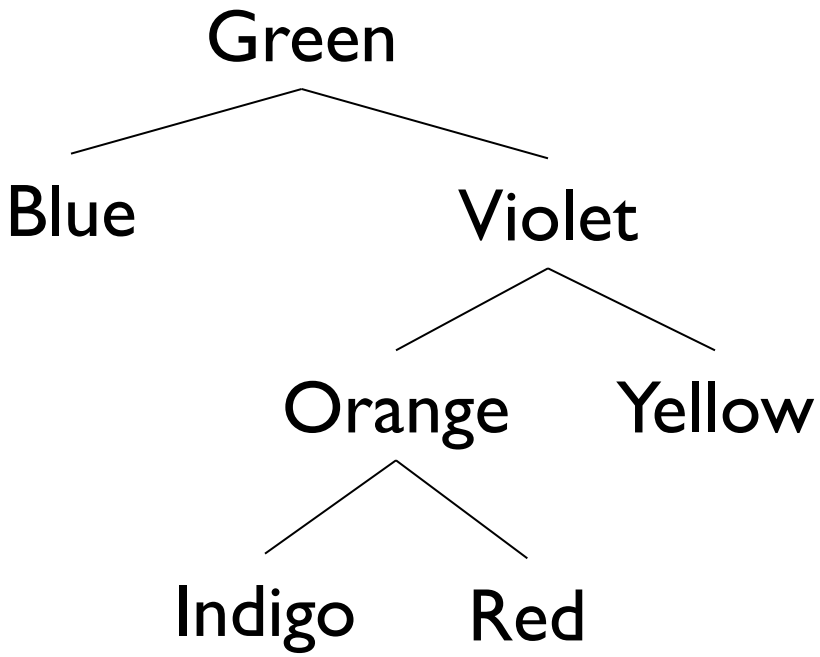
```
public void pre-order(BinaryTree t) {
      if(t.isEmpty()) return;
      touch(t); // some method
      preOrder(t.left());
      preOrder(t.right());
}
```



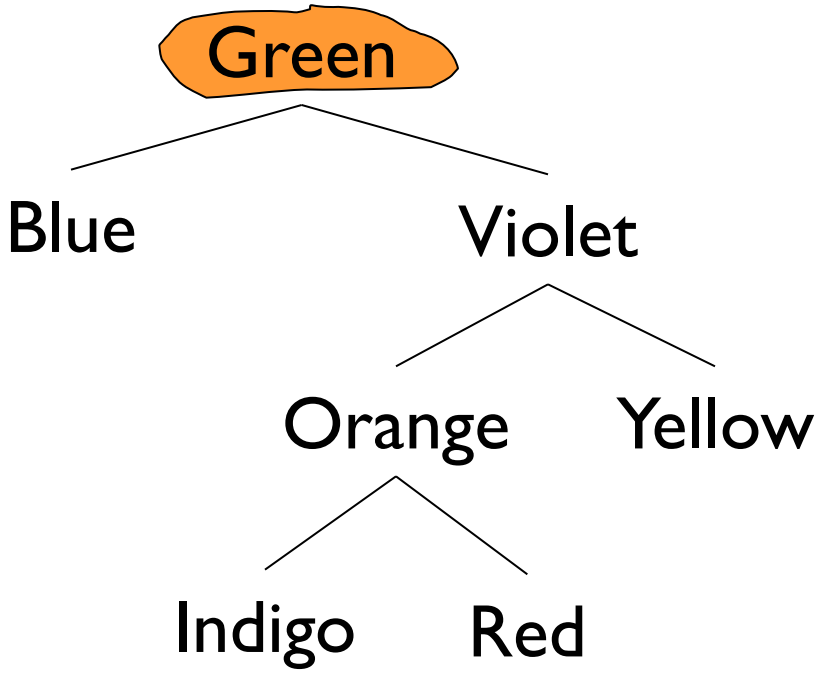
For in-order and post-order: just move touch(t)!

But what about level-order???

Level-Order Traversal

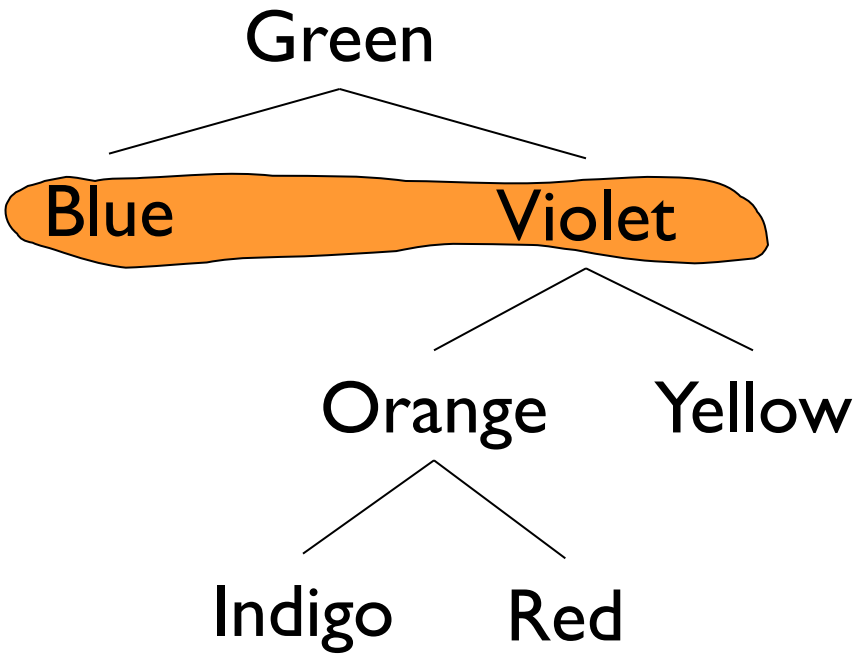


Level-Order Traversal



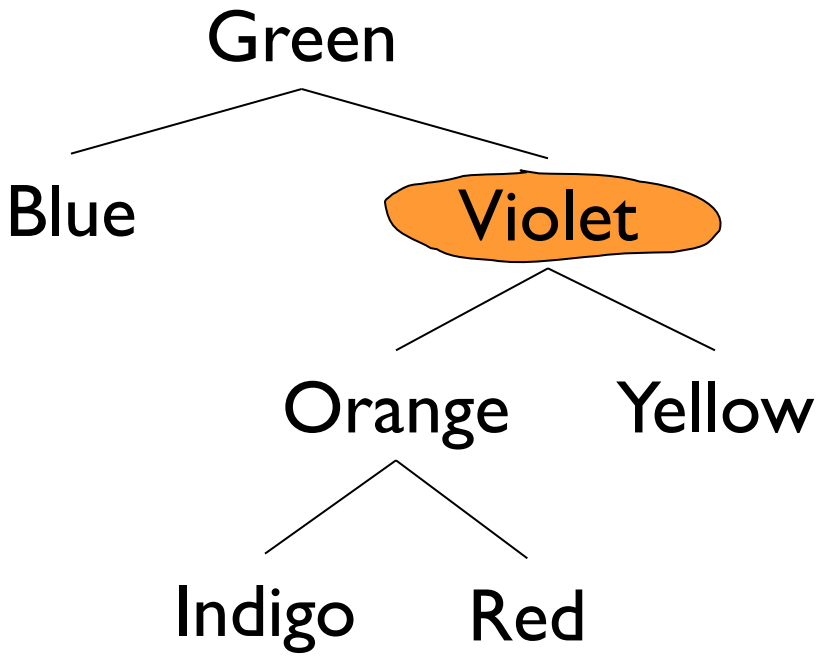
G

Level-Order Traversal



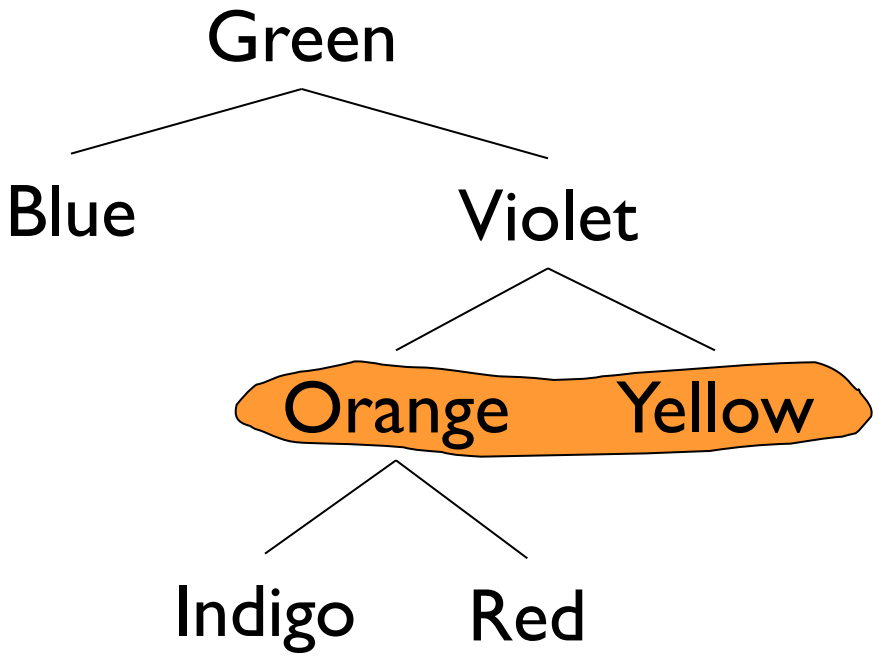
G

Level-Order Traversal



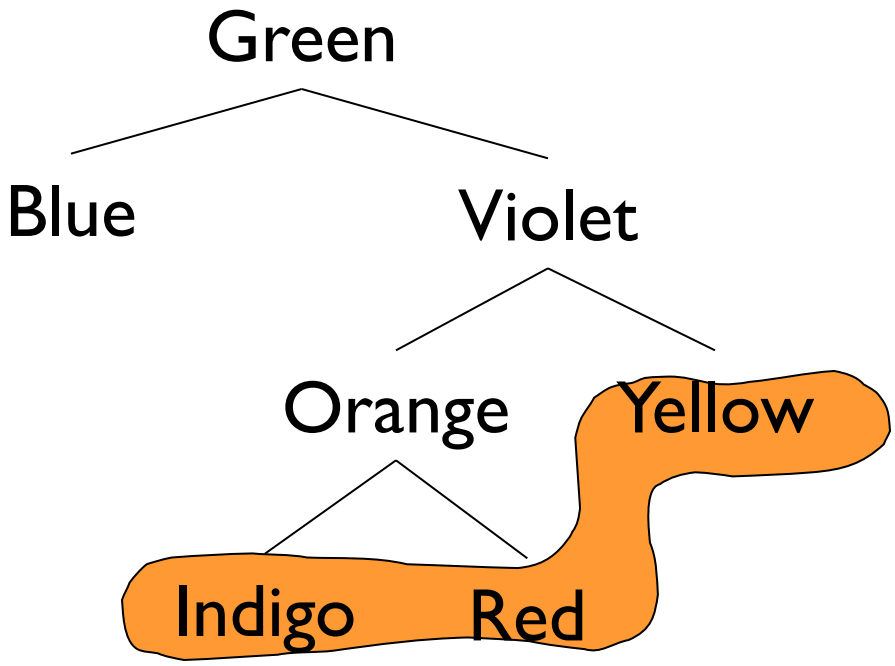
G B

Level-Order Traversal



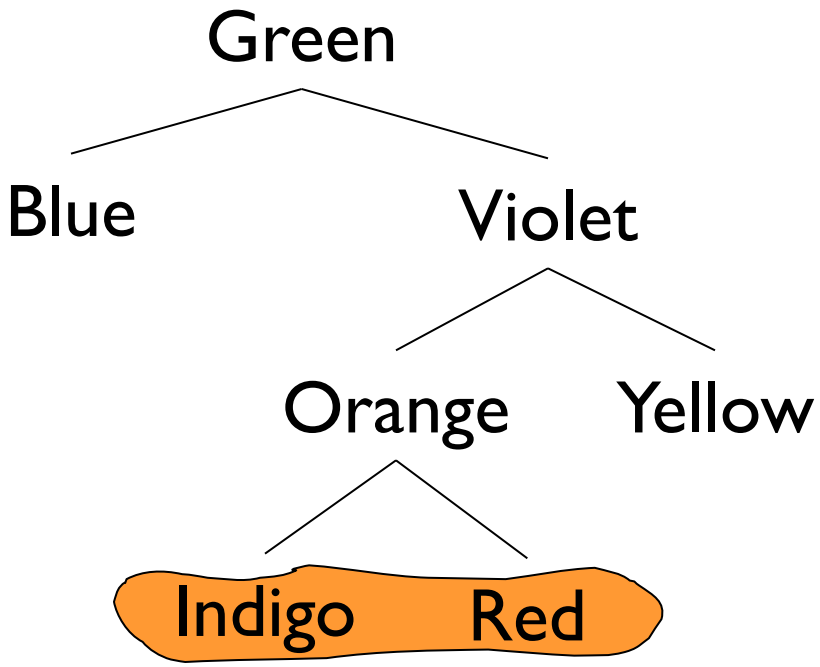
G B V

Level-Order Traversal



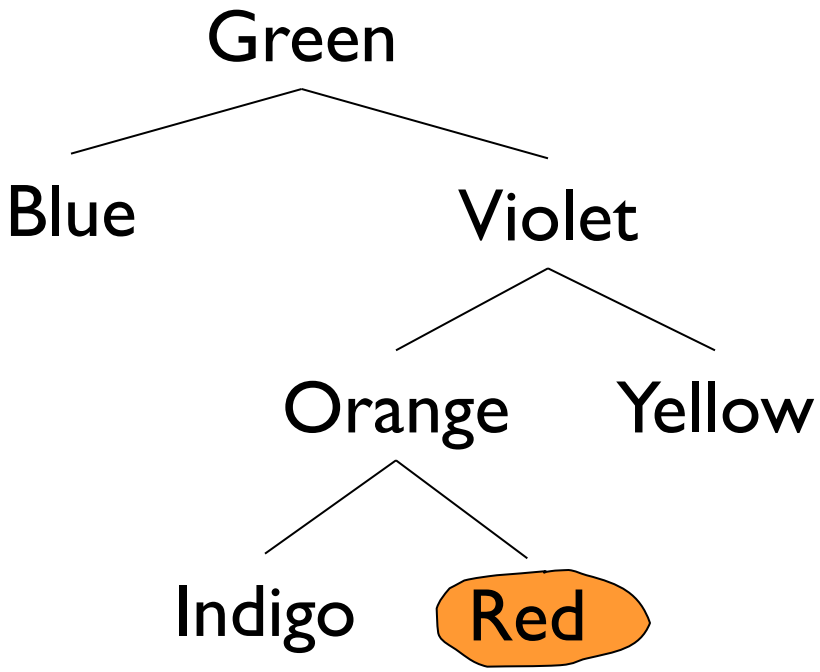
G B V O

Level-Order Traversal



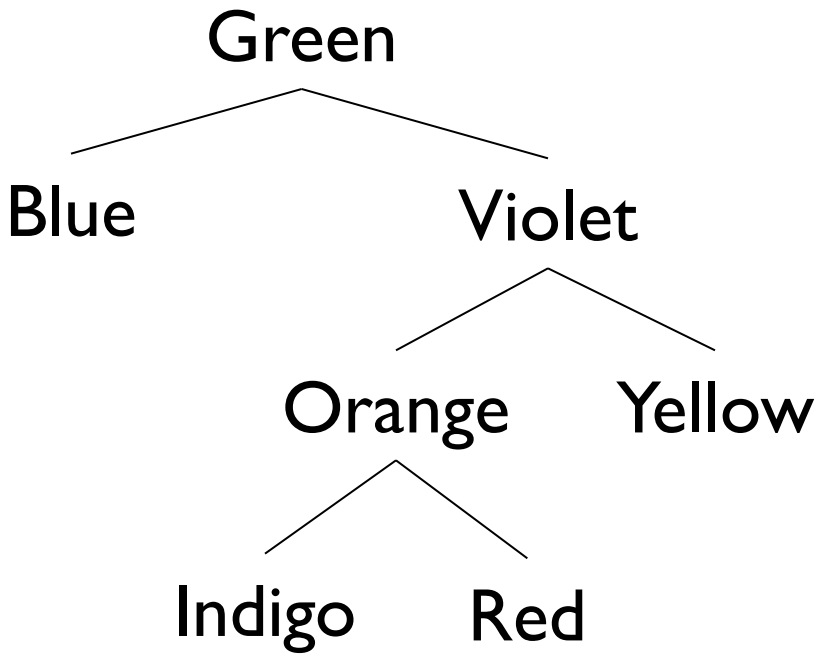
G B V O Y

Level-Order Traversal



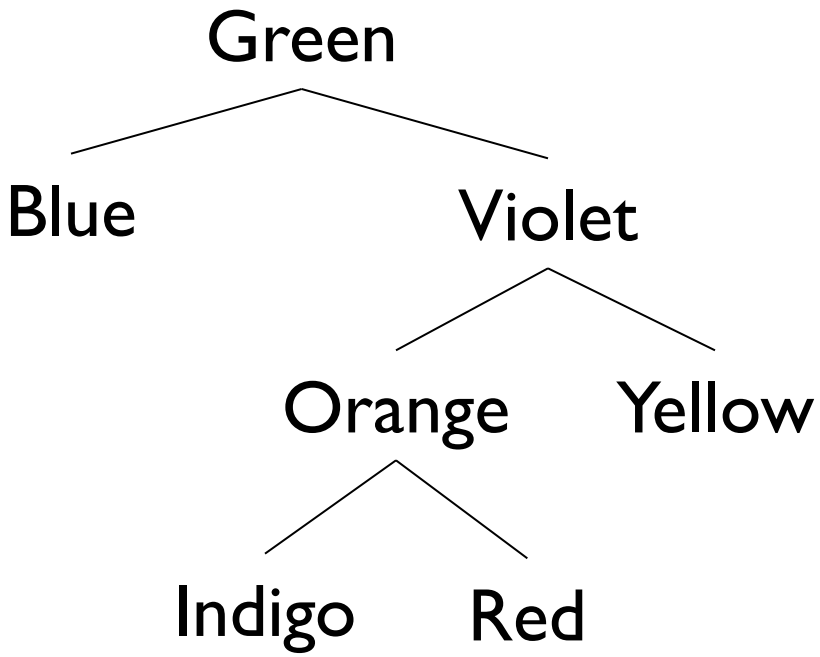
G B V O Y I

Level-Order Traversal

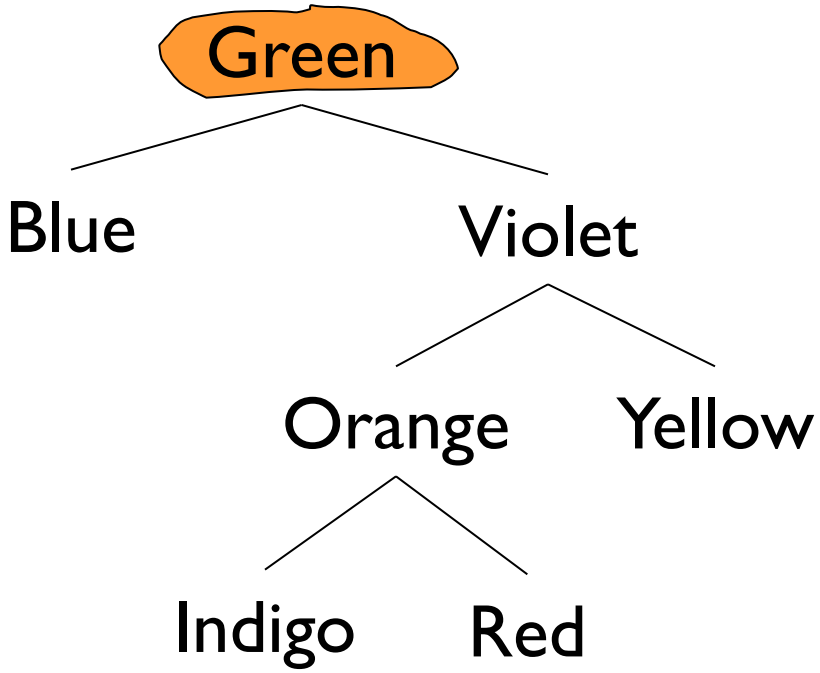


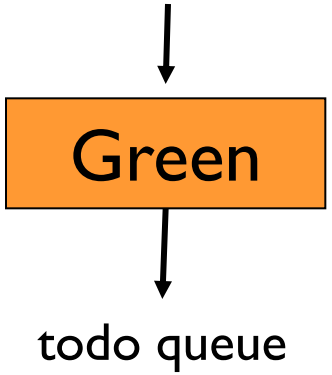
G B V O Y I R

Level-Order Traversal

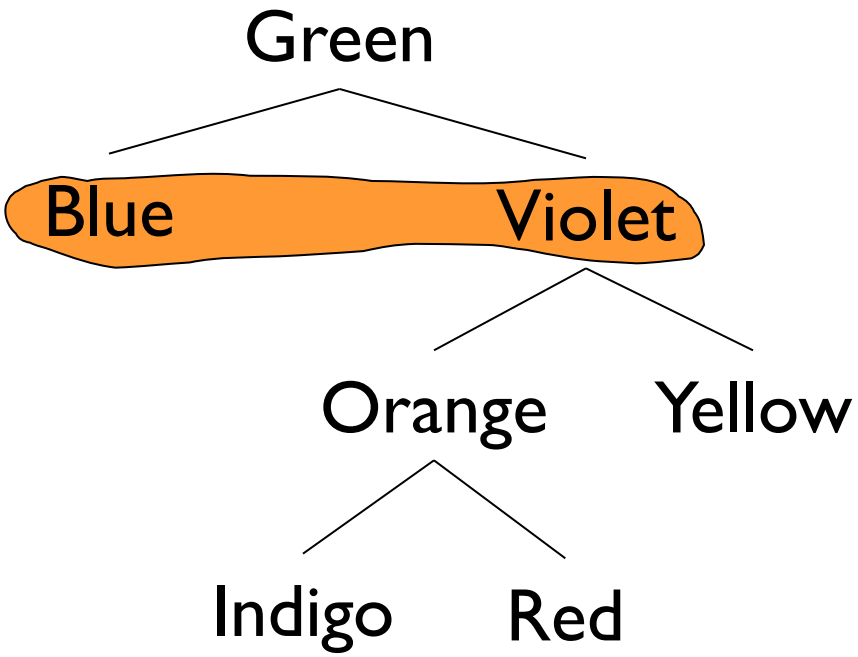


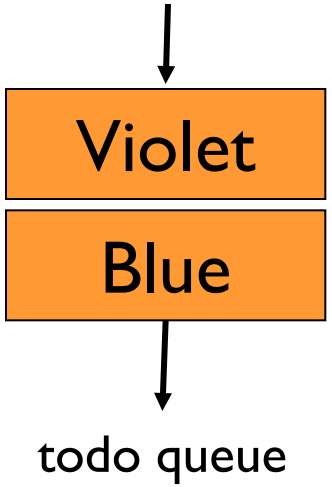
Level-Order Traversal





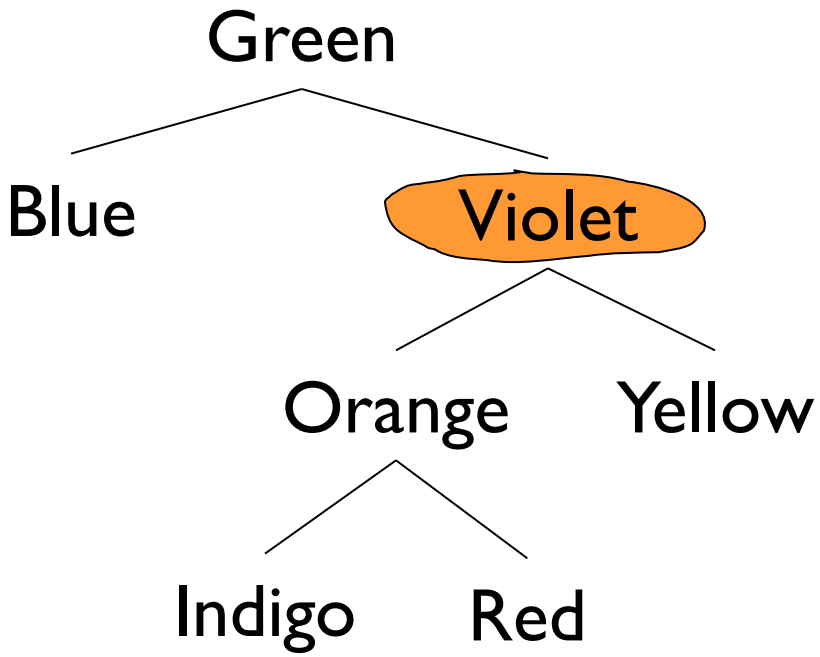
Level-Order Traversal

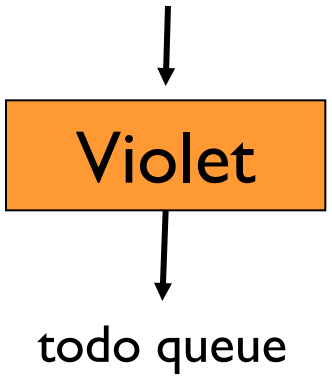




G

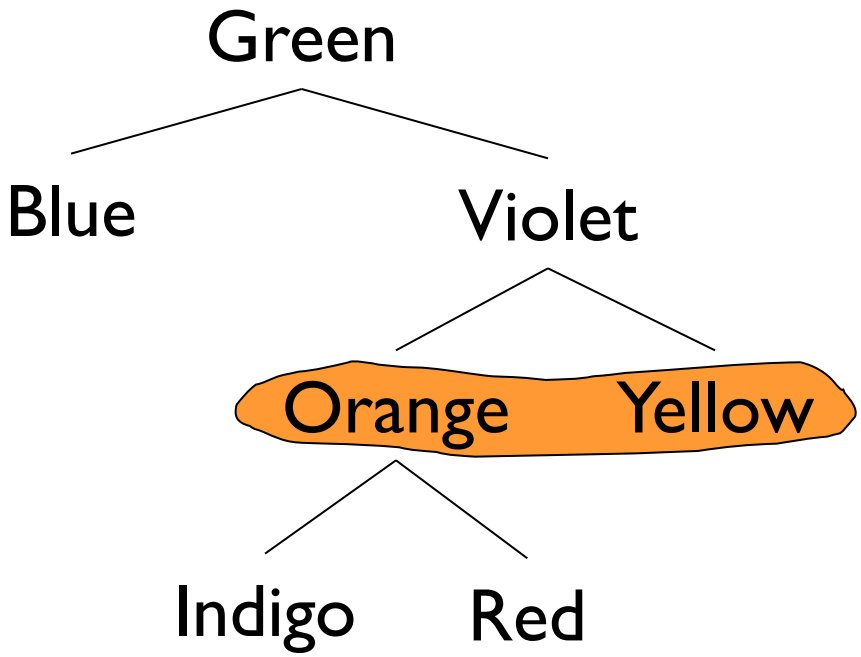
Level-Order Traversal

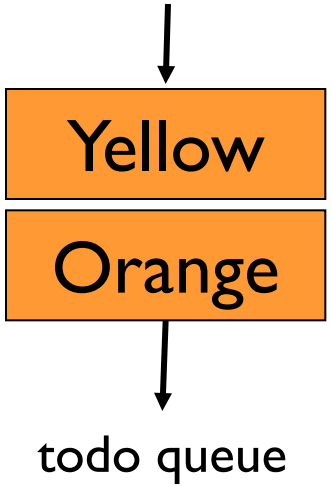




G B

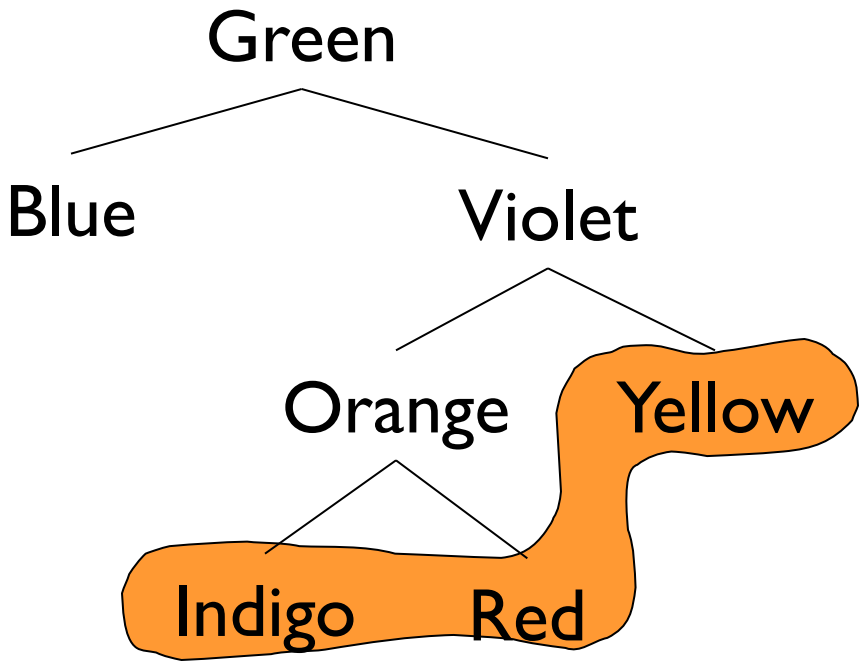
Level-Order Traversal

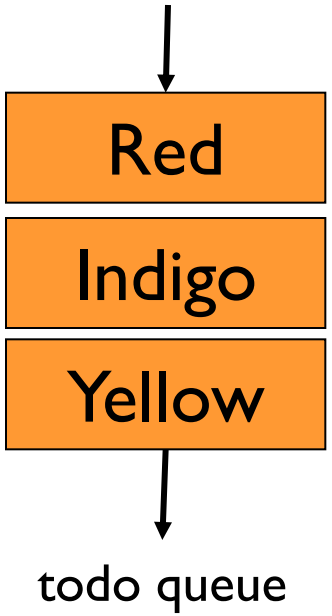




G B V

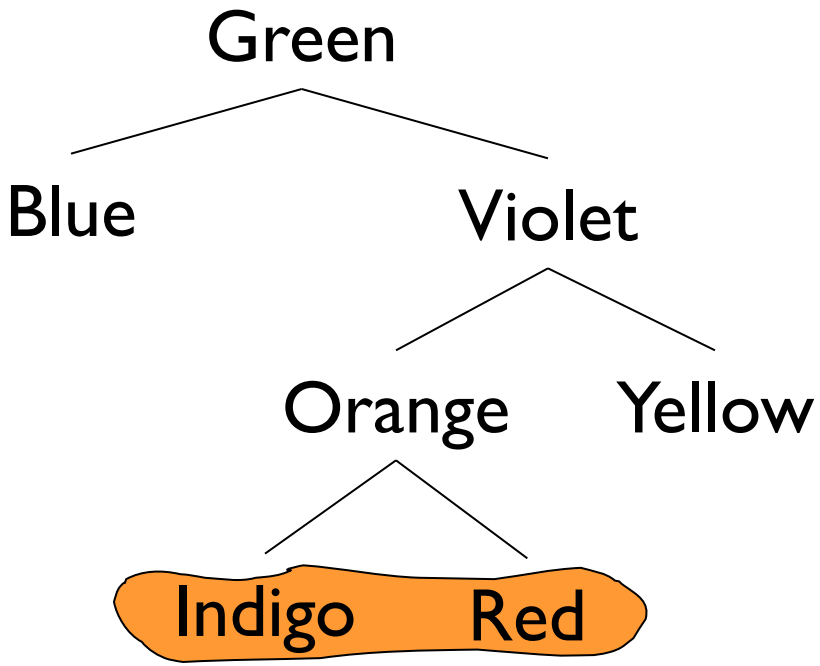
Level-Order Traversal

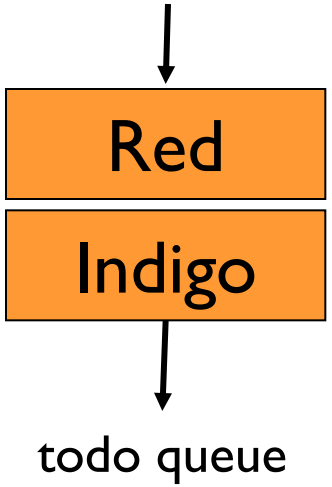




G B V O

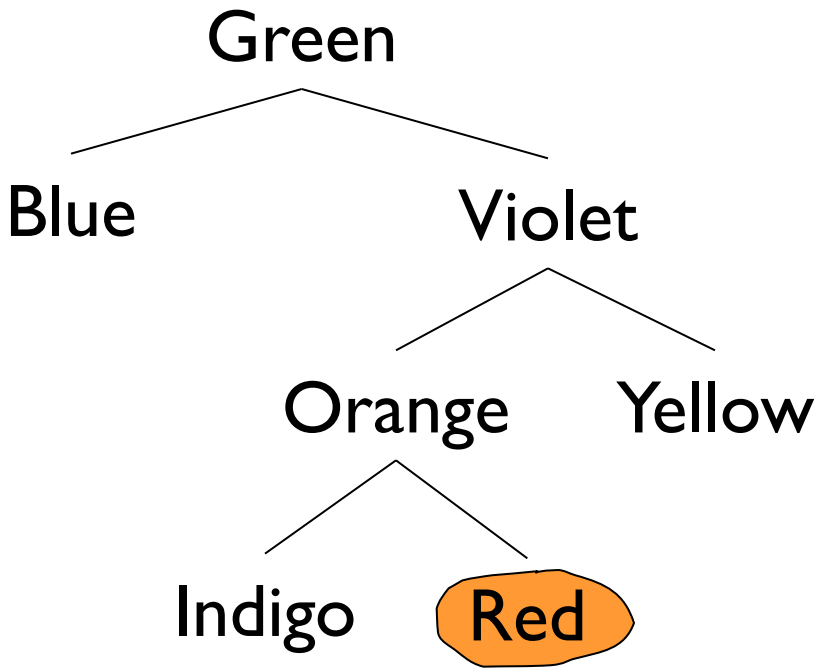
Level-Order Traversal

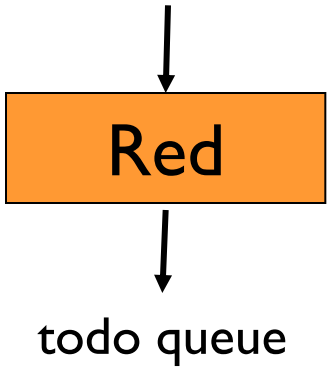




G B V O Y

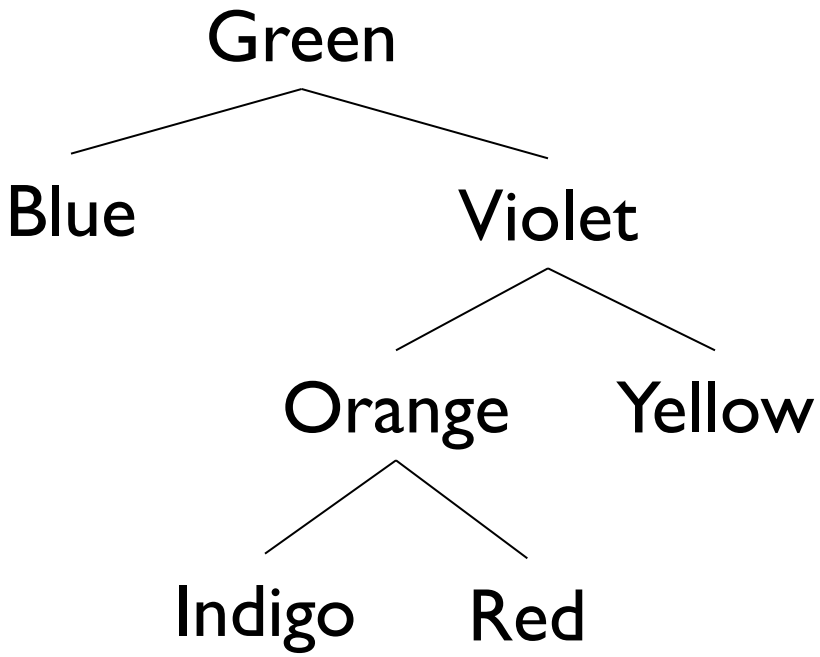
Level-Order Traversal

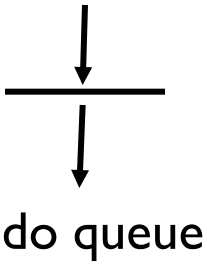




G B V O Y I

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());
   }
}
```

Iterators

- Provide iterators that implement the different tree traversal algorithms

- Methods provided by BinaryTree class:
  - preorderIterator()
  - inorderIterator()
  - postorderIterator()
  - levelorderIterator()

Implementing the Iterators

- Basic idea
  - Should return elements in same order as corresponding traversal method shown
  - Recursive methods don't convert as easily: must phrase in terms of next() and hasNext()
  - So, let's start with levelOrder!

Level-Order Iterator

```
public BTLevelorderIterator(BinaryTree<E> root)
   {
       todo = new QueueList<BinaryTree<E>>();
       this.root = root; // needed for reset
       reset();
   }

public void reset()
    {
        todo.clear();
        // empty queue, add root
        if (!root.isEmpty()) todo.enqueue(root);
    }
```

Level-Order Iterator

```
public boolean hasNext() {
        return !todo.isEmpty();
}

public E next() {
        BinaryTree<E> current = todo.dequeue();
        E result = current.value();
        if (!current.left().isEmpty())
            todo.enqueue(current.left());
        if (!current.right().isEmpty())
            todo.enqueue(current.right());
        return result;
}
```

Pre-Order Iterator

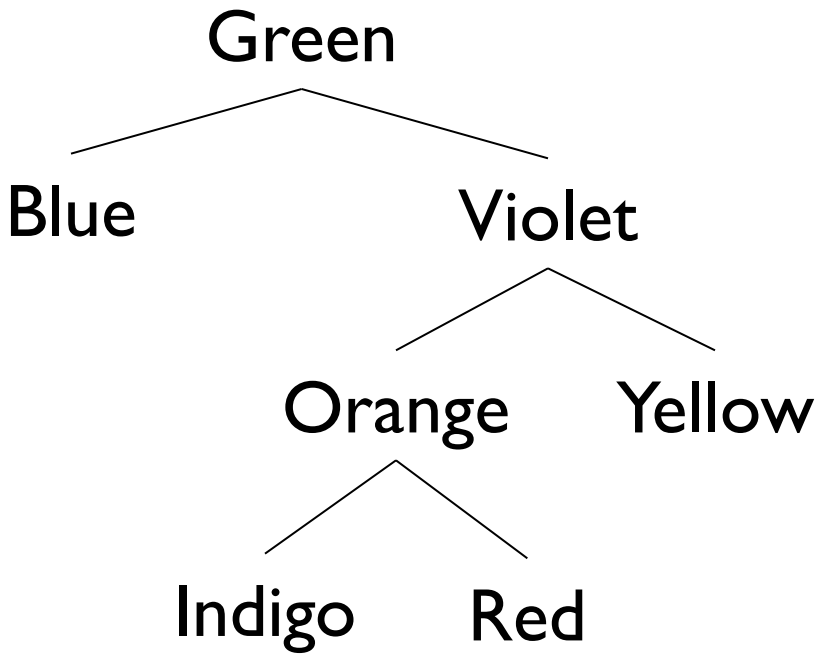
- Basic idea
  - Should return elements in same order as processed by pre-order traversal method
  - Must phrase in terms of next() and hasNext()
  - We "simulate recursion" with stack
    - The stack holds "partially processed" nodes

Pre-Order Iterator

- Outline: node - left tree – right tree
  1. Constructor: Push root onto todo stack
  2. On call to next():
     - Pop node from stack
     - Push right and then left nodes of popped node onto stack
     - Return node's value
  3. On call to hasNext():
     - return !stack.isEmpty()

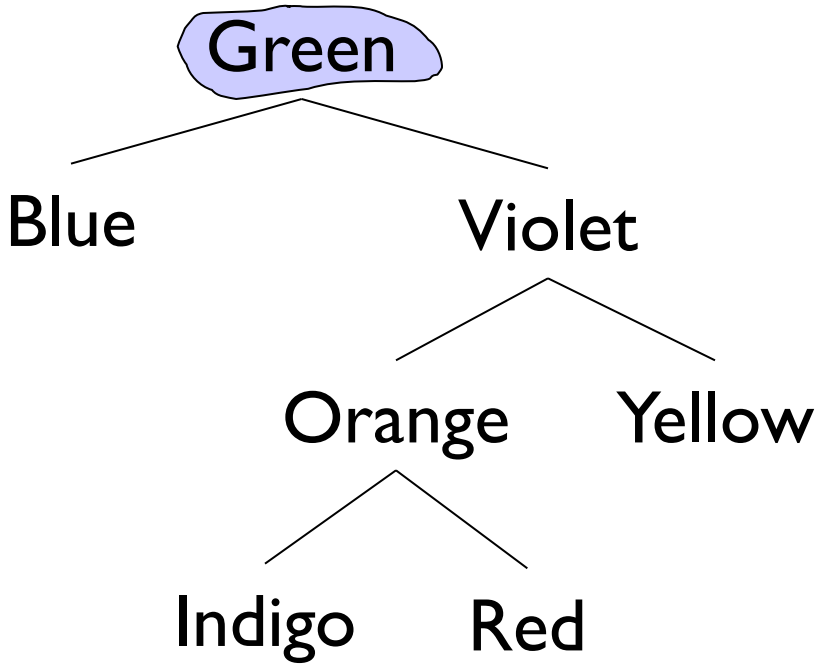
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



Pre-Order Iterator

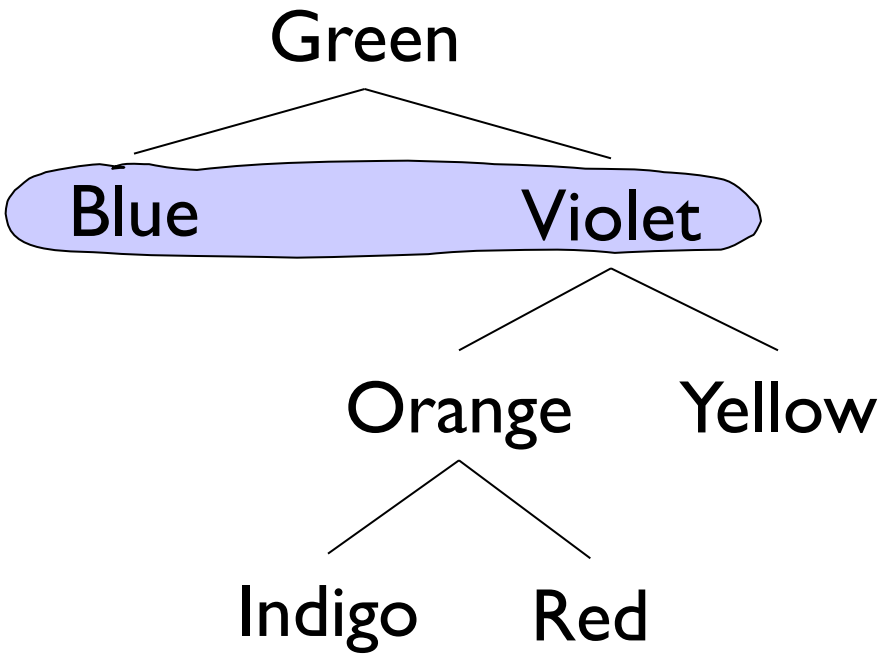
Visit node, then each node in left subtree, then each node in right subtree.

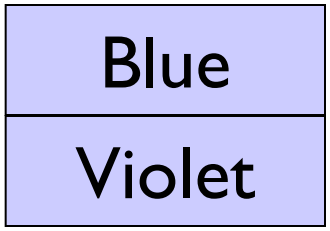


todo stack

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



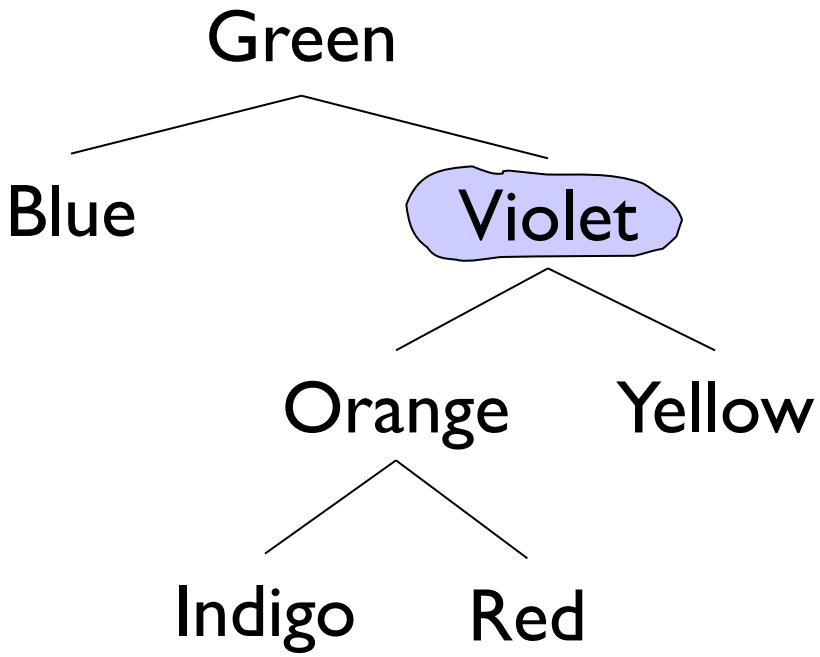


todo stack

G

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

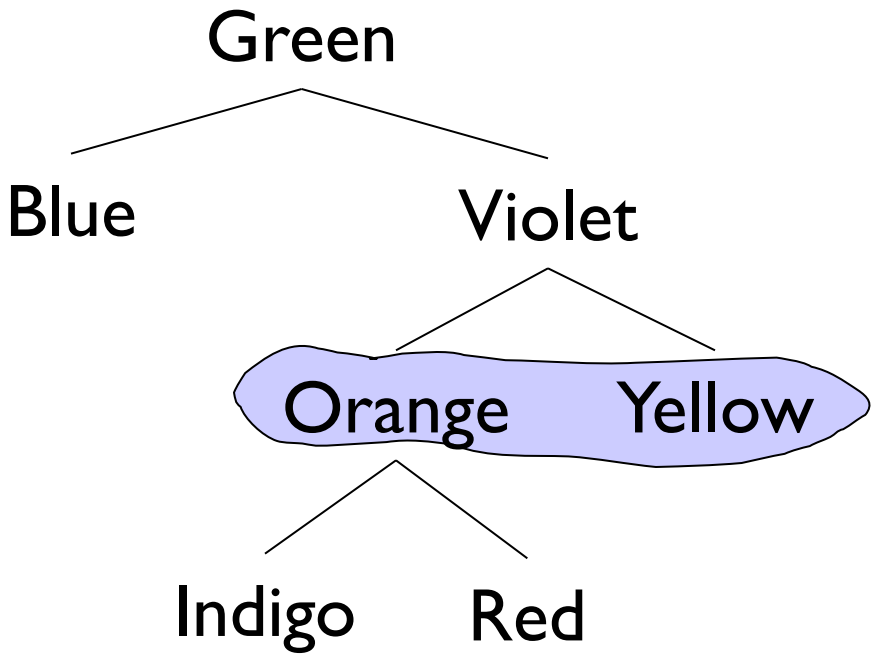


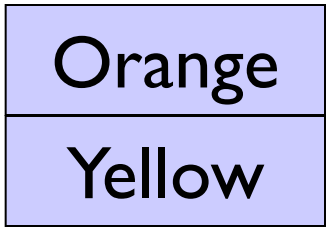
todo stack

G B

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



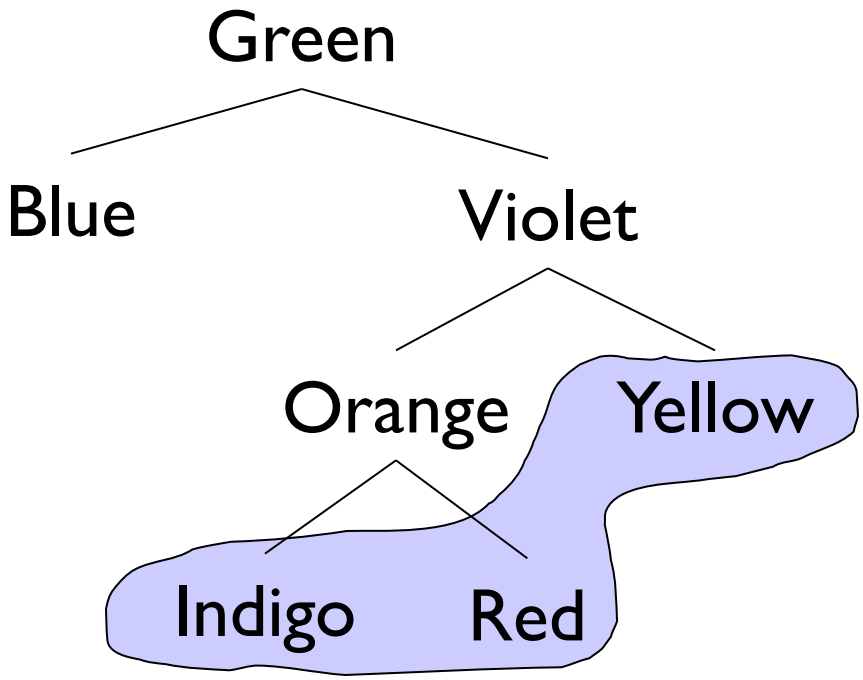


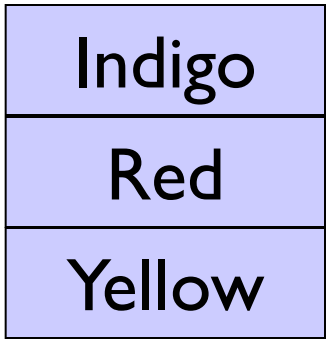
todo stack

G B V

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



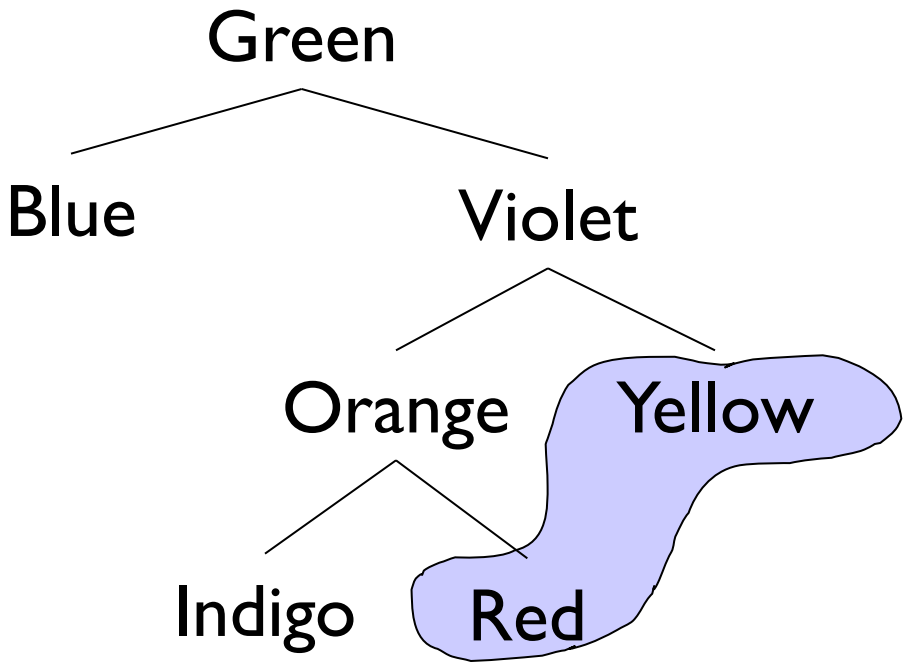


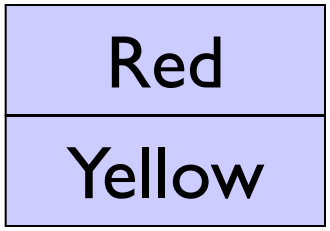
todo stack

G B V O

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



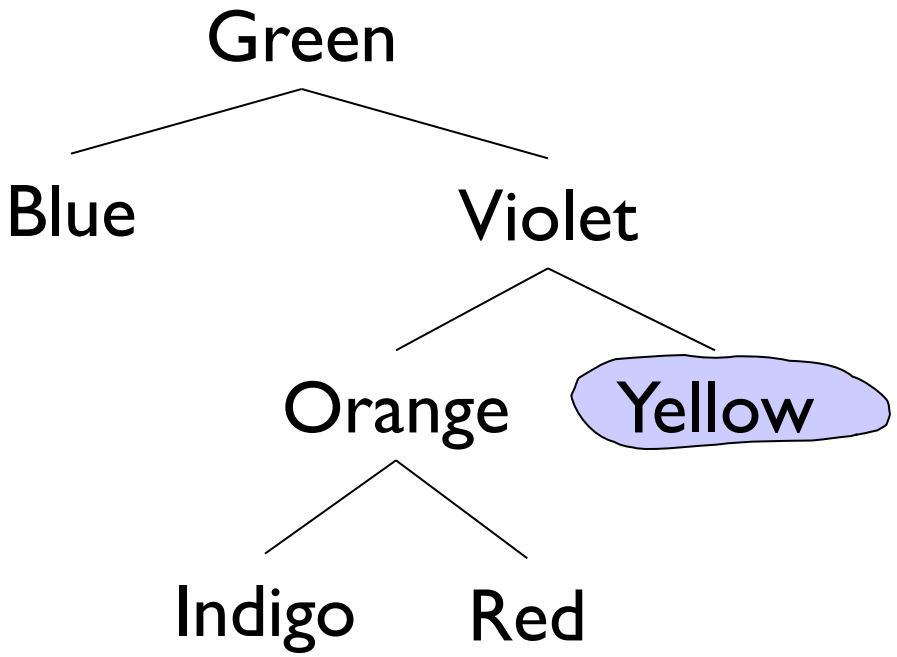


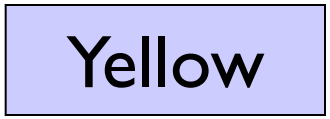
todo stack

G B V O I

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



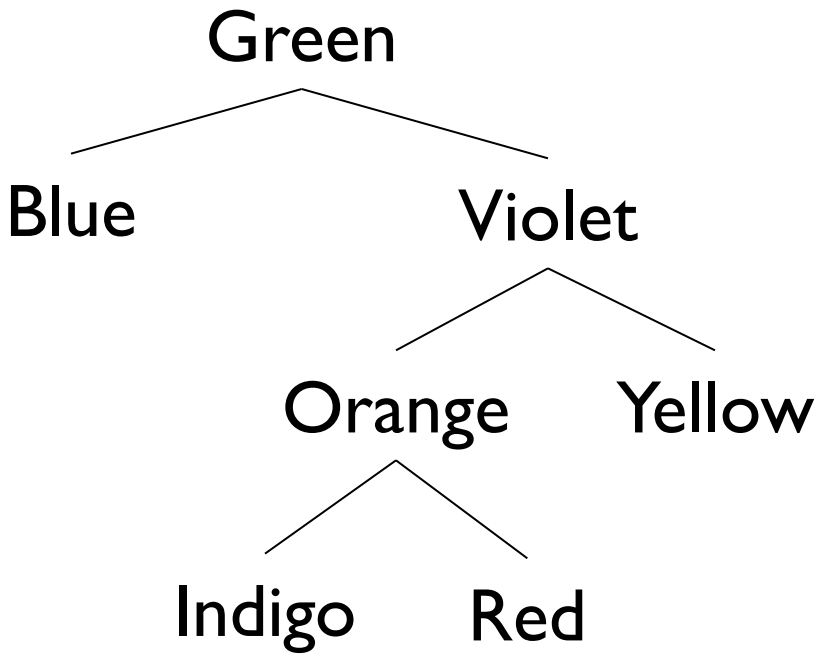


todo stack

G B V O I R

Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.



todo stack

G B V O I R Y

Pre-Order Iterator

```
public BTPreorderIterator(BinaryTree<E> root)
    {
        todo = new StackList<BinaryTree<E>>();
        this.root = root;
        reset();
    }

public void reset()
    {
        todo.clear(); // stack is empty; push on root
        if ((!root.isEmpty()) todo.push(root);
    }
```

Pre-Order Iterator

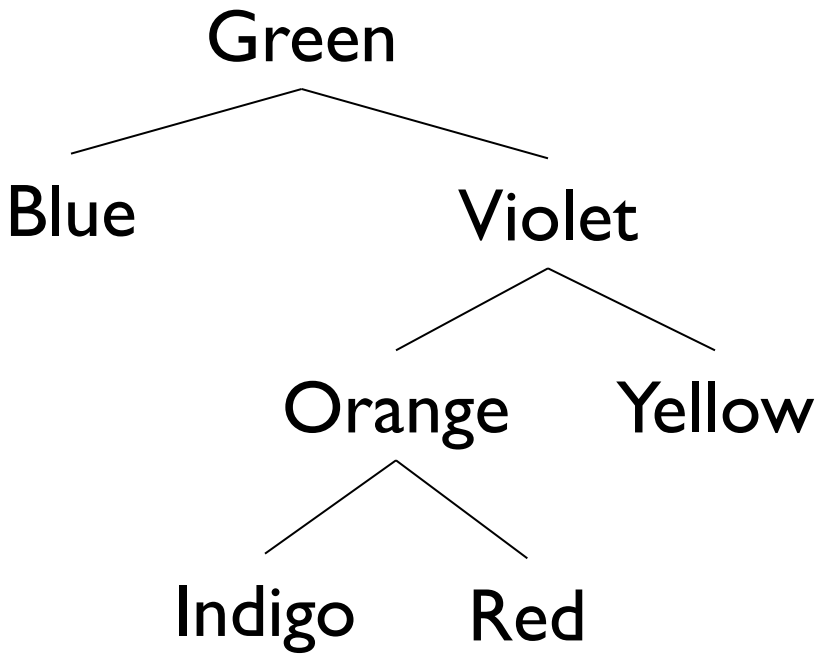
```
public boolean hasNext() {
        return !todo.isEmpty();
}

public E next() {
     BinaryTree<E> old = todo.pop();
     E result = old.value();

     if (!old.right().isEmpty())
           todo.push(old.right());
     if (!old.left().isEmpty())
           todo.push(old.left());
       return result;
   }
```

Tree Traversal Practice Problems

- Prove that levelOrder() is correct: that is, that it touches the nodes of the tree in the correct order (Hint: induction by level)
- Prove that levelOrder() takes O(n) time, where n is the size of the tree
- Prove that the PreOrder (LevelOrder) Iterator visits the nodes in the same order as the PreOrder (LevelOrder) traversal method

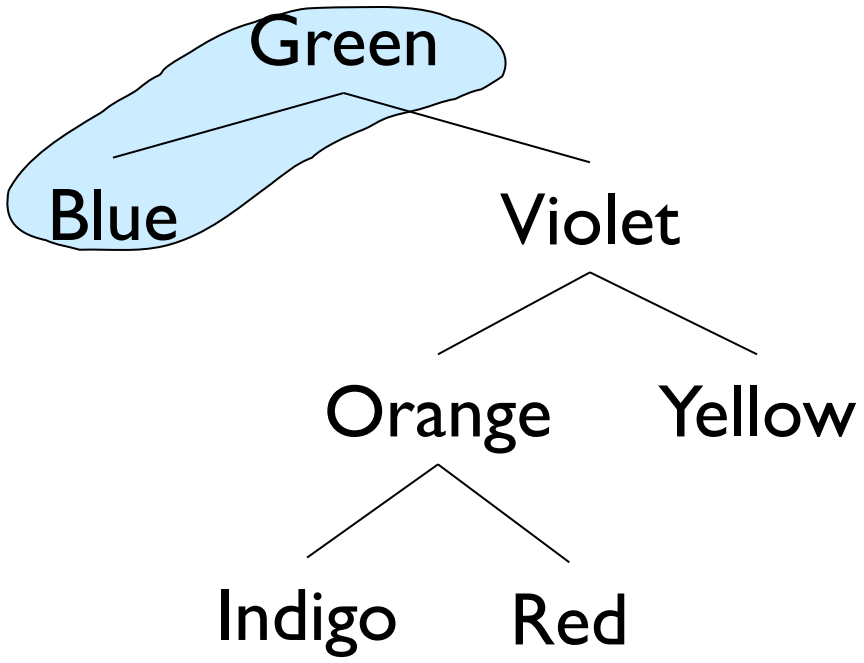
In-Order Iterator

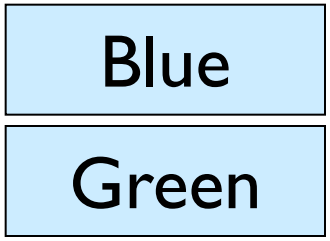
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.



In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

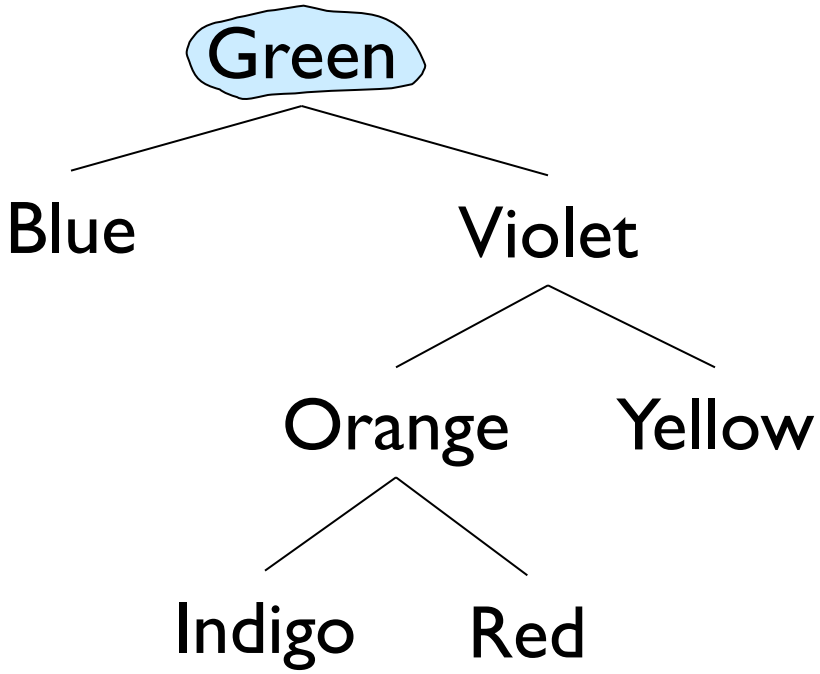




todo stack

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

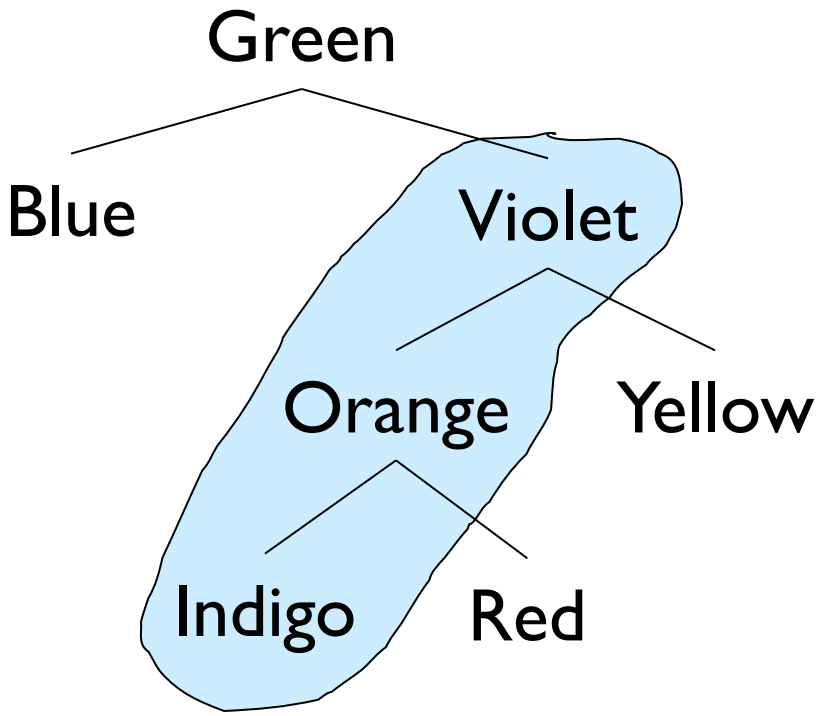


todo stack

B

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

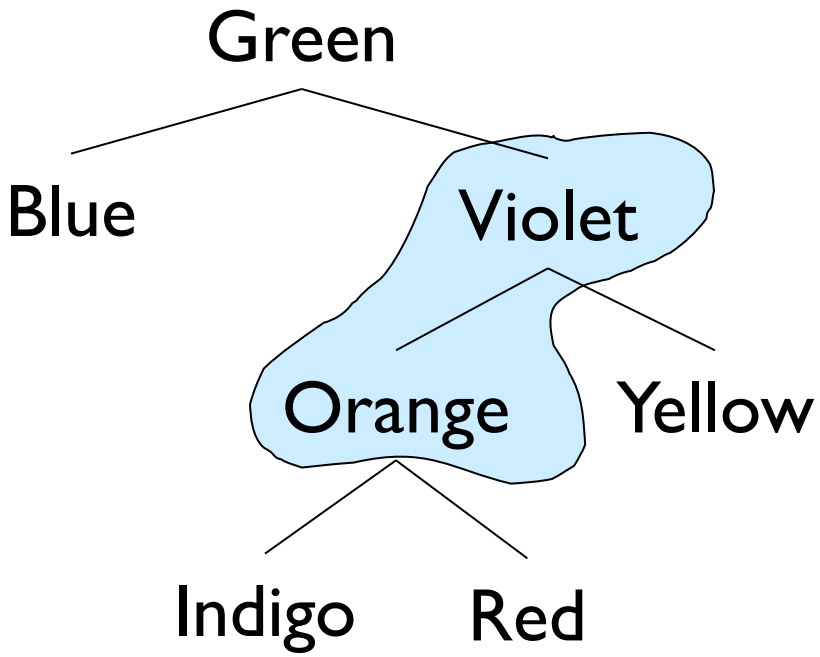


todo stack

B G

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

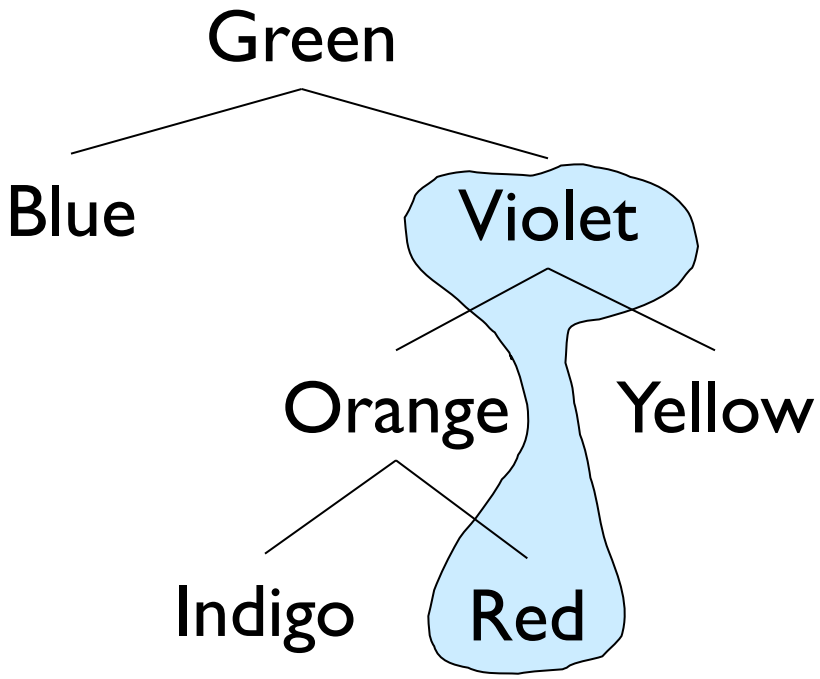


todo stack

B G I

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

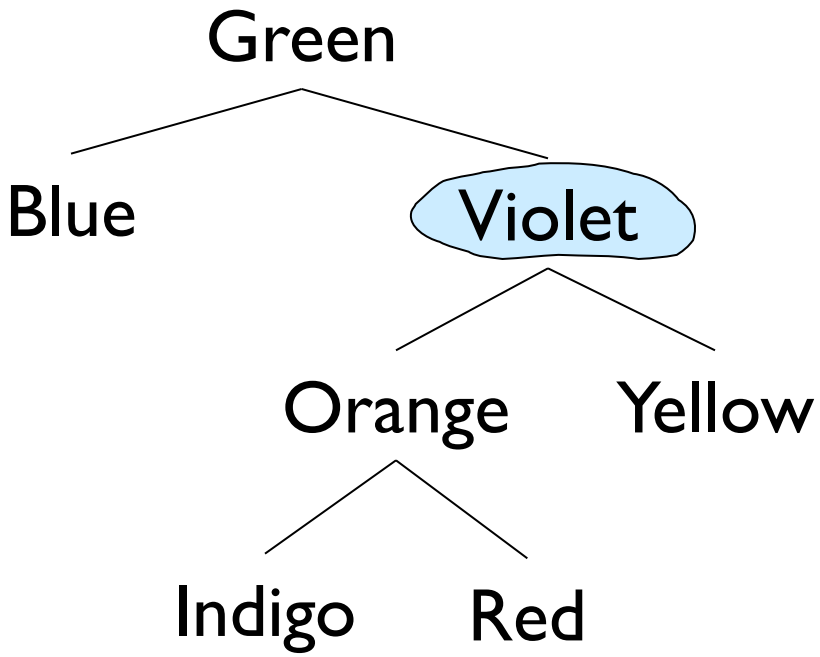


todo stack

B G I O

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

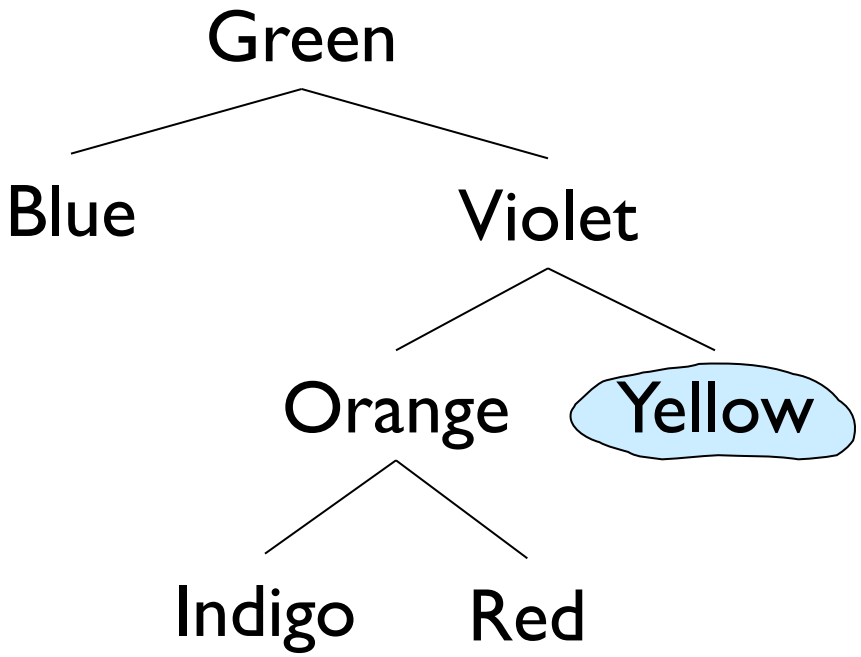


todo stack

B G I O R

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

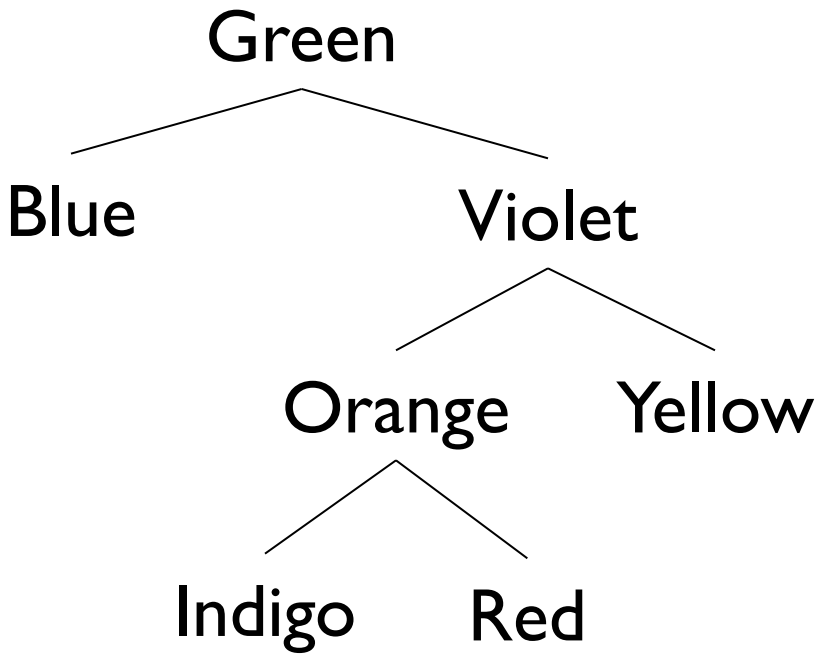


todo stack

B G I O R V

In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.



todo stack

B G I O R V Y

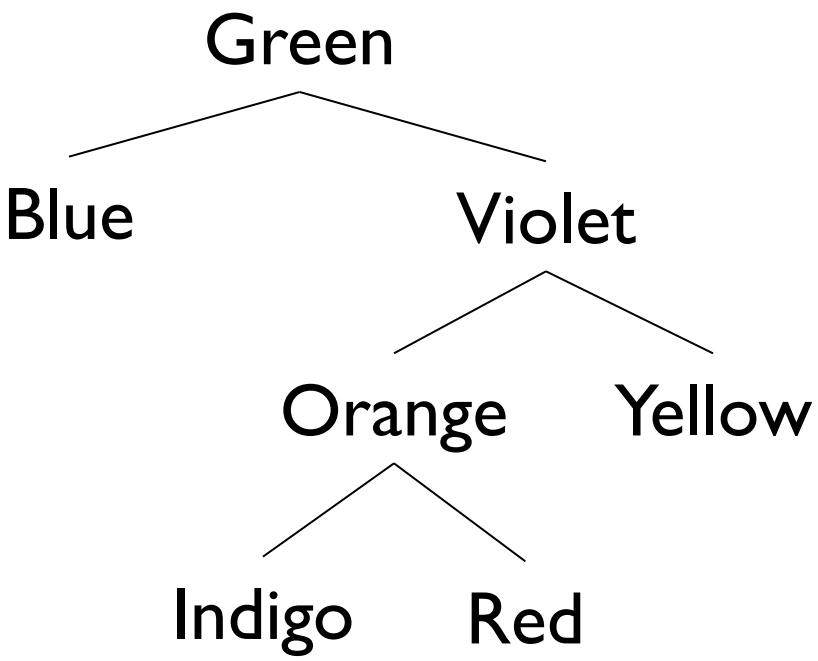
In-Order Iterator

- Outline: left - node - right
  1. Push left children (as far as possible) onto stack
  2. On call to next():
     - Pop node from stack
     - Push right child and follow left children as far as possible
     - Return node's value
  3. On call to hasNext():
     - return !stack.isEmpty()

Post-Order Iterator

- Left as an exercise…

Alternative Tree Representations



- Total # "slots" = 4n
  - Since each BinaryTree maintains a reference to left, right, parent, value
- 2-4x more overhead than vector, SLL, array, …
- But trees capture successor and predecessor relationships that other data structures don't…

Array-Based Binary Trees

- Encode structure of tree in array indexes
  - Put root at index 0
- Where are children of node i?
  - Children of node i are at 2i+1 and 2i+2
  - Look at example
- Where is parent of node j?
  - Parent of node j is at (j-1)/2

ArrayTree Tradeoffs

- Why are ArrayTrees good?
  - Save space for links
  - No need for additional memory allocated/garbage collected
  - Works well for full or complete trees
    - Complete: All levels except last are full and all gaps are at right
    - "A *complete* binary tree of height h is a full binary tree with 0 or more of the rightmost leaves of level h removed"
- Why bad?
  - Could waste a lot of space
  - Tree of height of n requires 2^{n+1}-1 array slots even if only O(n) elements