CSCI 136
Data Structures &
Advanced Programming

Lecture 20

Fall 2018

Instructor: Bills

Administrative Details

- Lab 7 is available online
  - No partners this week
  - Review before lab; come to lab with design doc
  - We'll give an overview shortly

Last Time

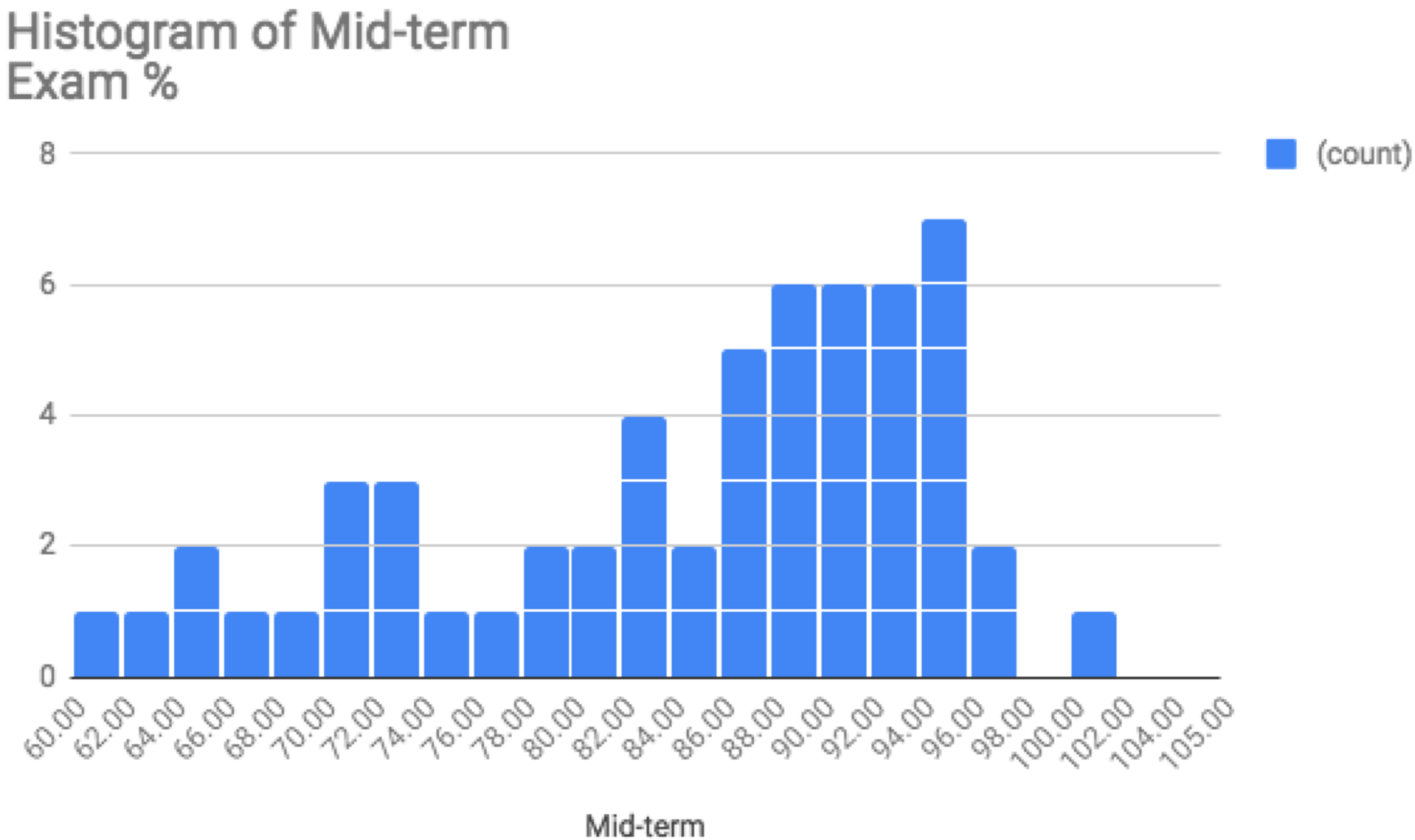
- Recursion/Induction on Trees
- Applications: Decision Trees
- Trees with more than 2 children
  - Representations
- Traversing Binary Trees
  - As methods taking a BinaryTree parameter

Today

- Binary Trees Traversals
  - As methods taking a BinaryTree parameter
    - Lever Order Traversal
  - With Iterators
- Big Trees
- Lab 7 Discussion
- Storing Trees in Arrays

Mid-Term Results

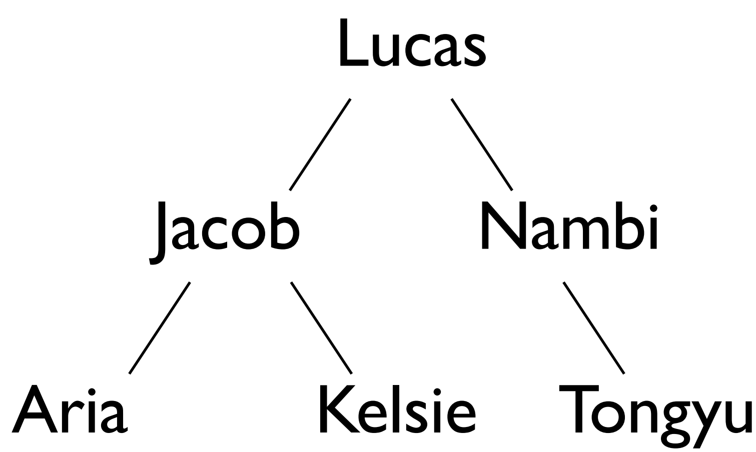
- Average grade: 84.7%



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



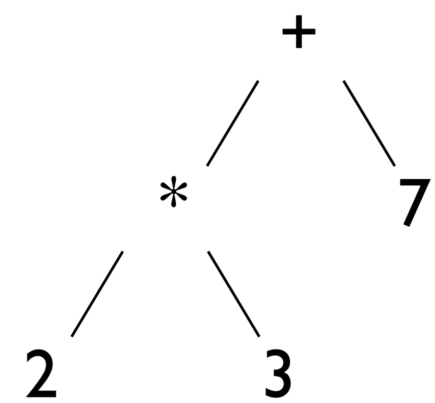
In-order: Aria, Jacob, Kelsie, Lucas, Nambi, Tongyu

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

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

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

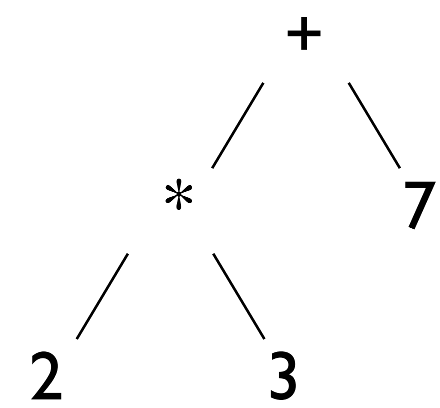
Tree Traversals



- 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

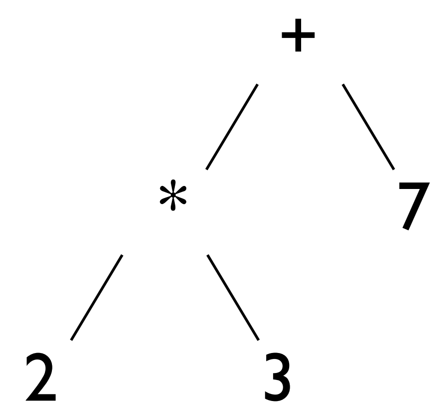


- Post-order
  - Each node is visited after its children are visited. Visit all nodes in left subtree, then all nodes in right subtree, then node itself. (left, right, node)
    - 23*7+
- Level-order (not obviously recursive!)
  - All nodes of level i are visited before nodes of level i+1. (visit nodes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

("pseudocode")

Tree Traversals

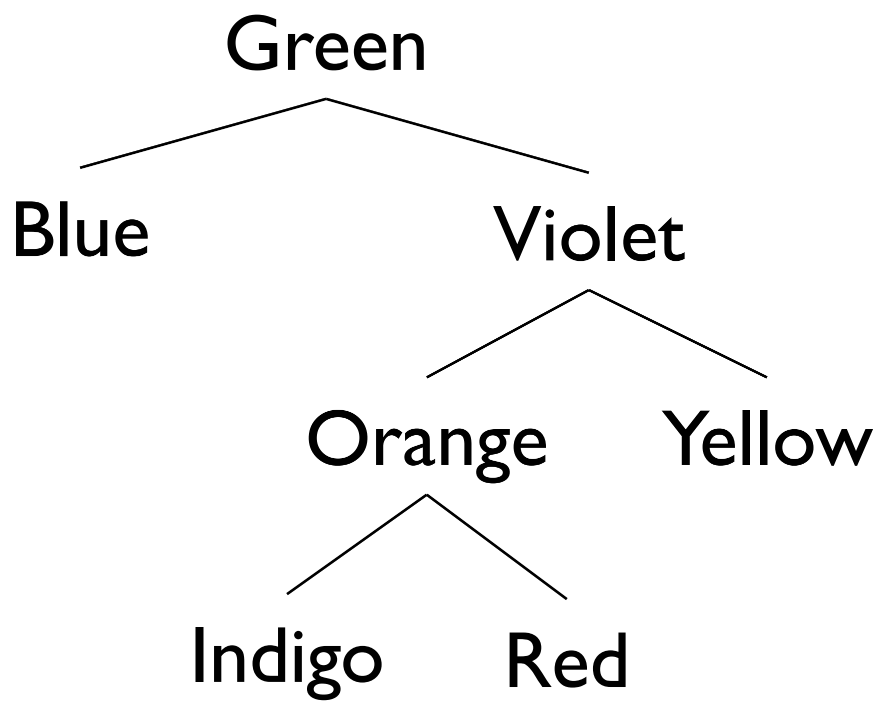
```
public void pre-order(BinaryTree t) {
      if(t.isEmpty()) return;
      touch(t); // some method
      preOrder(t.left());
      preOrder(t.right());
}
```



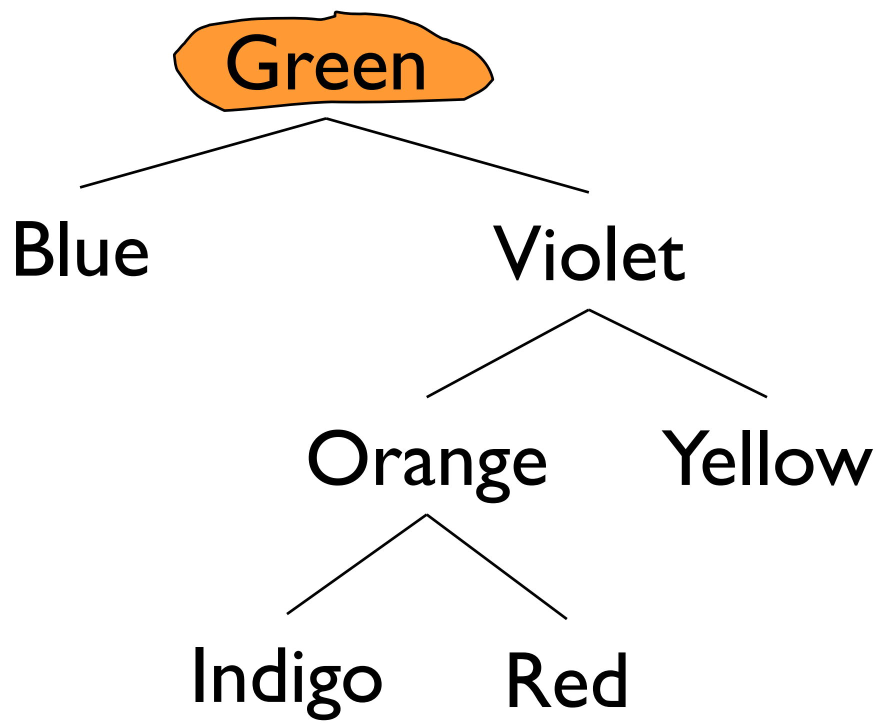
For in-order and post-order: just move touch(t)!

But what about level-order???

Level-Order Traversal

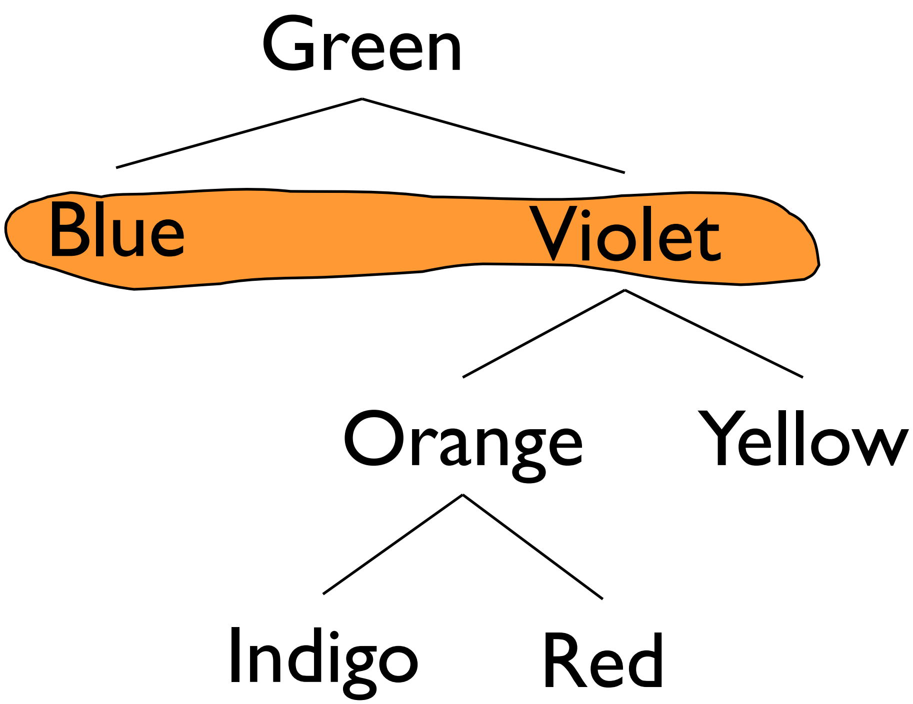


Level-Order Traversal



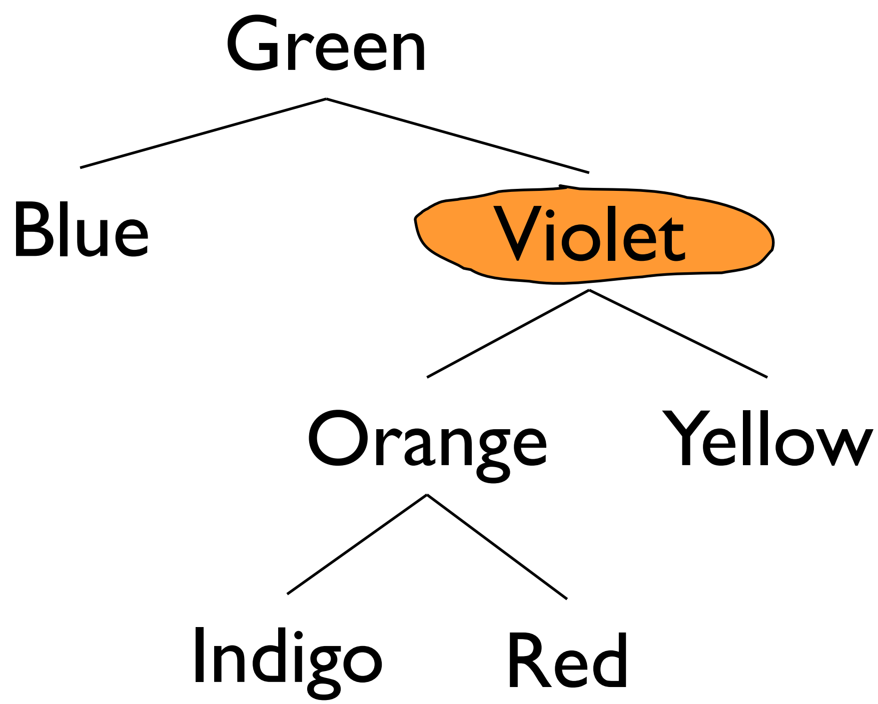
G

Level-Order Traversal



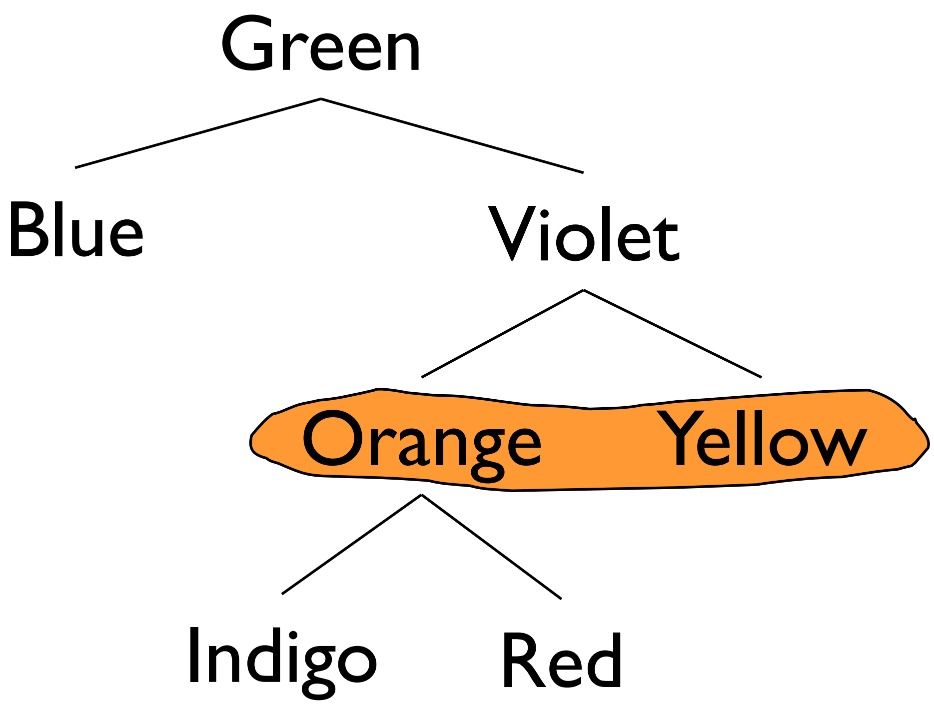
G

Level-Order Traversal



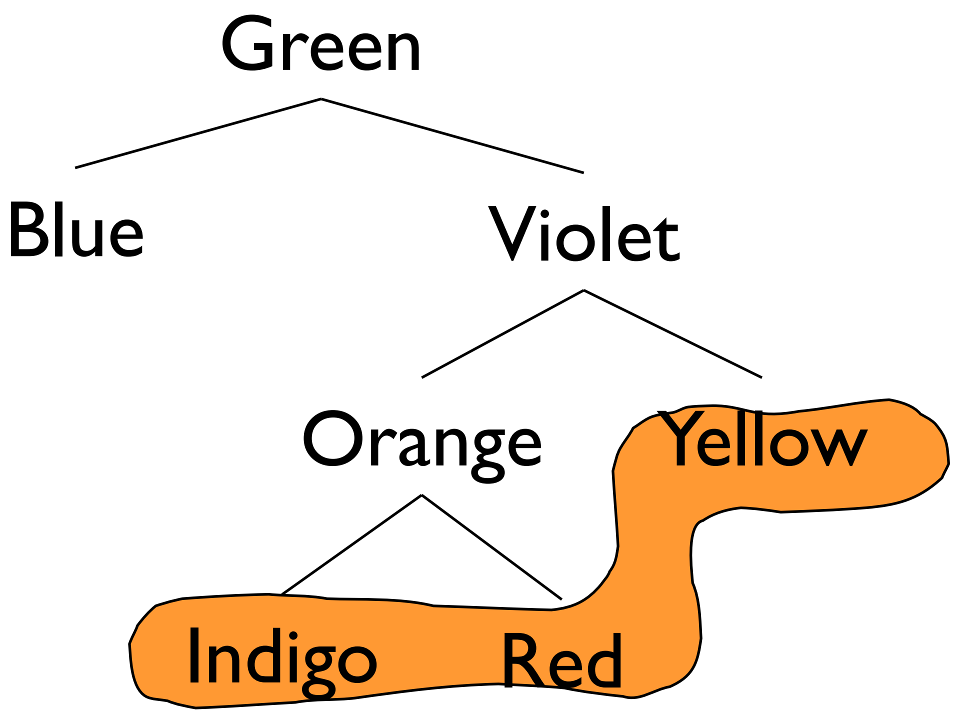
G B

Level-Order Traversal



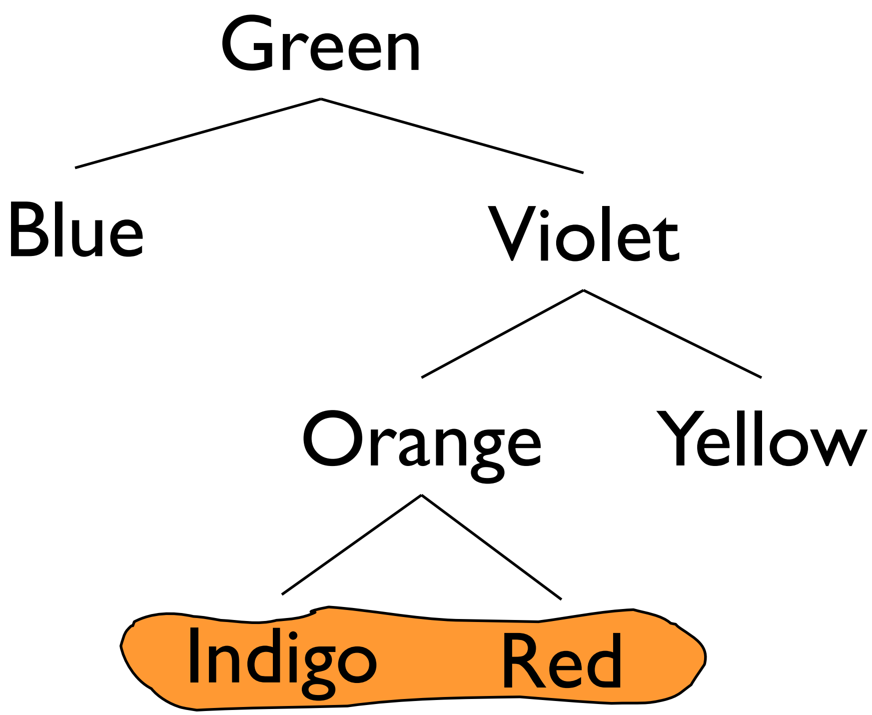
G B V

Level-Order Traversal



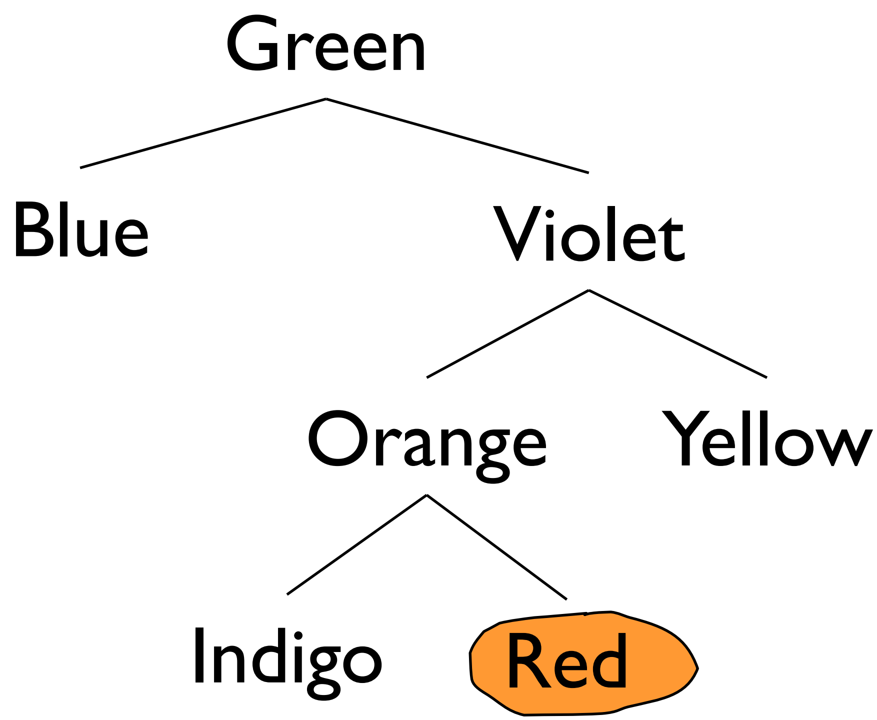
G B V O

Level-Order Traversal



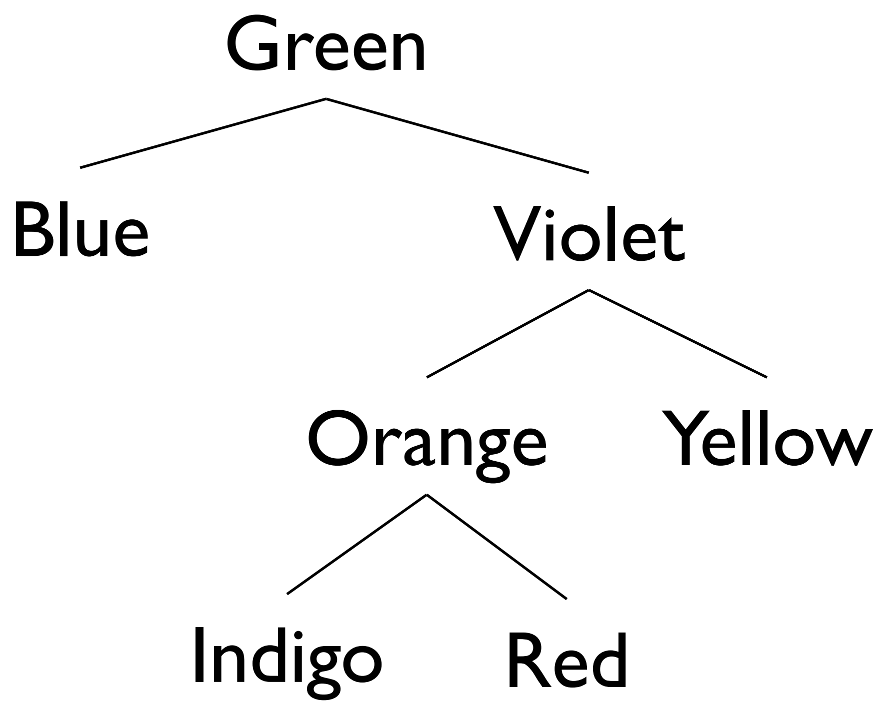
G B V O Y

Level-Order Traversal



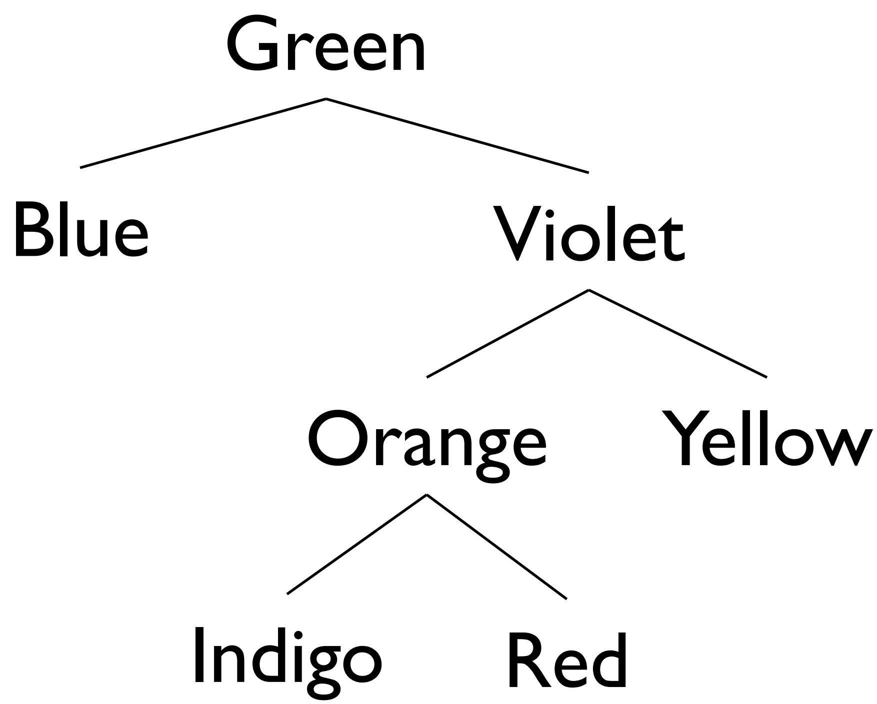
G B V O Y I

Level-Order Traversal

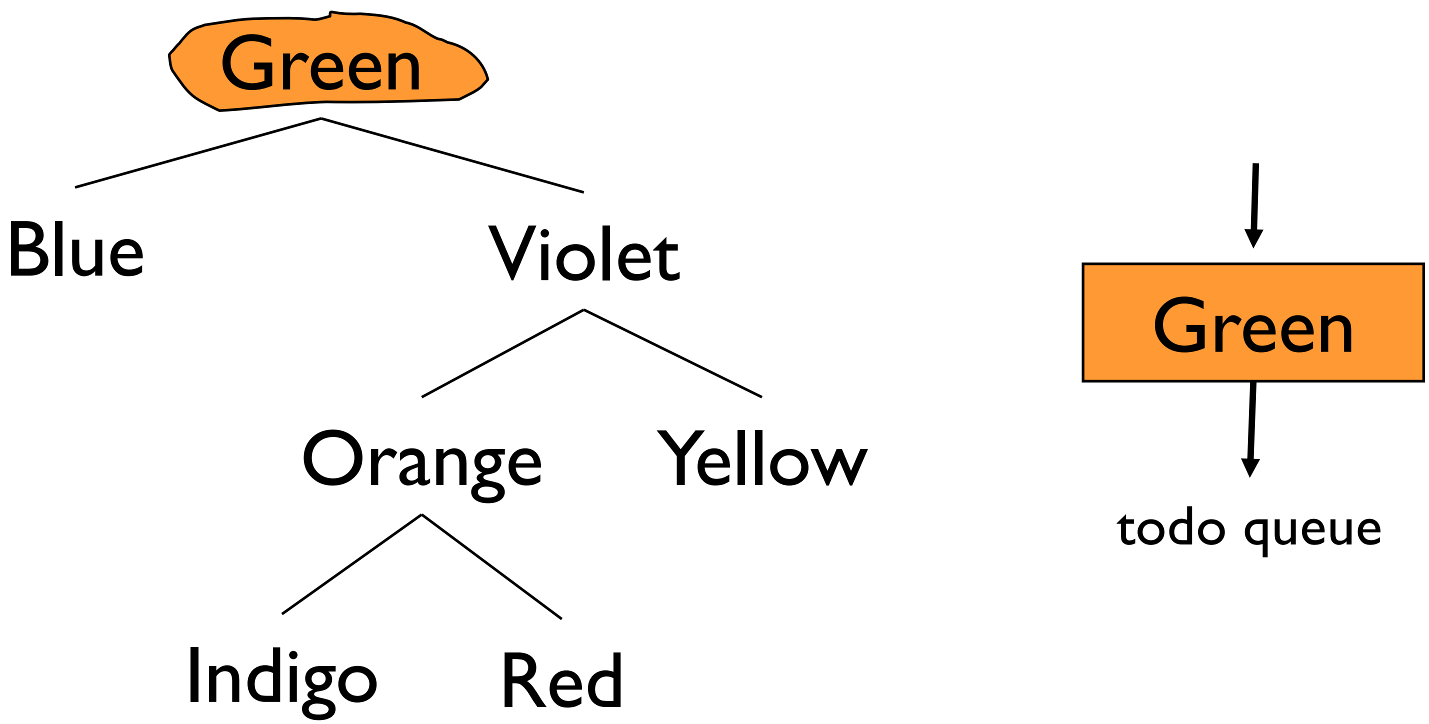


G B V O Y I R

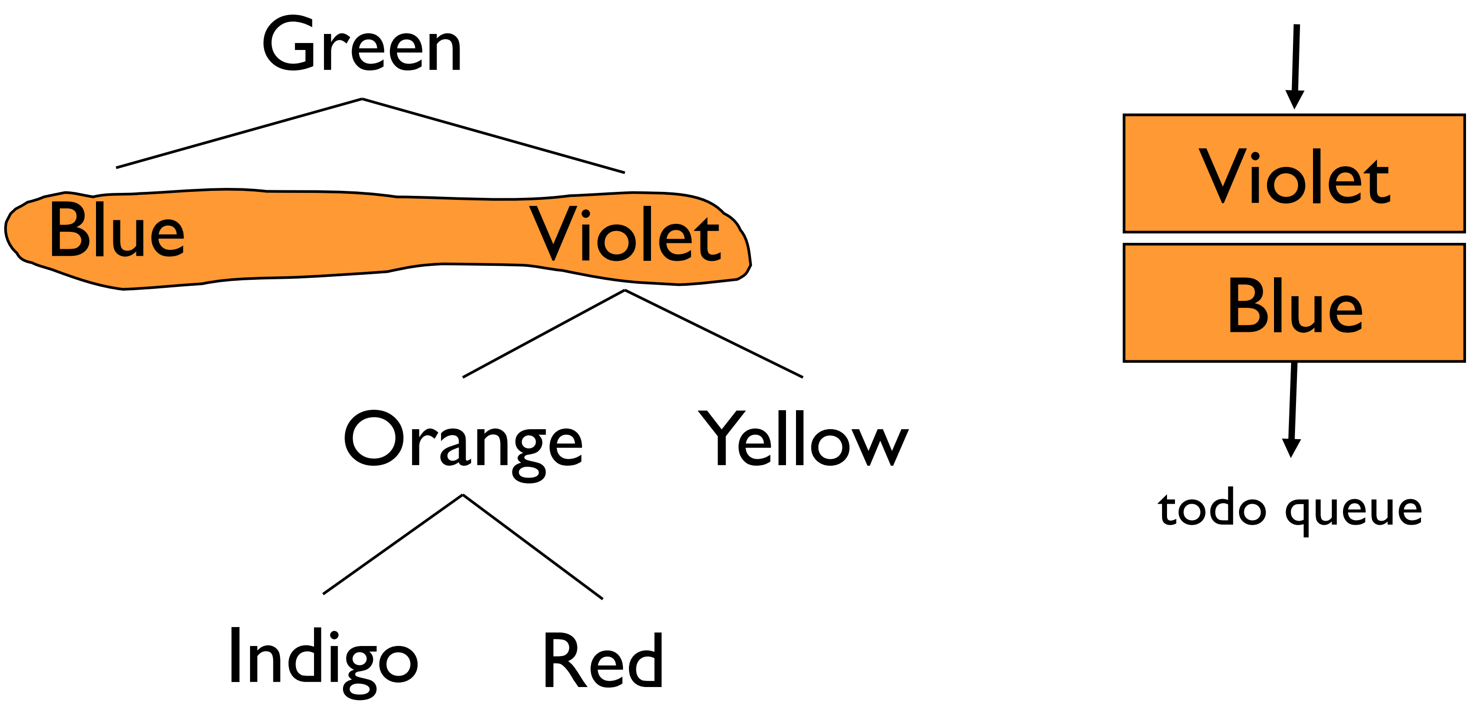
Level-Order Traversal



Level-Order Traversal

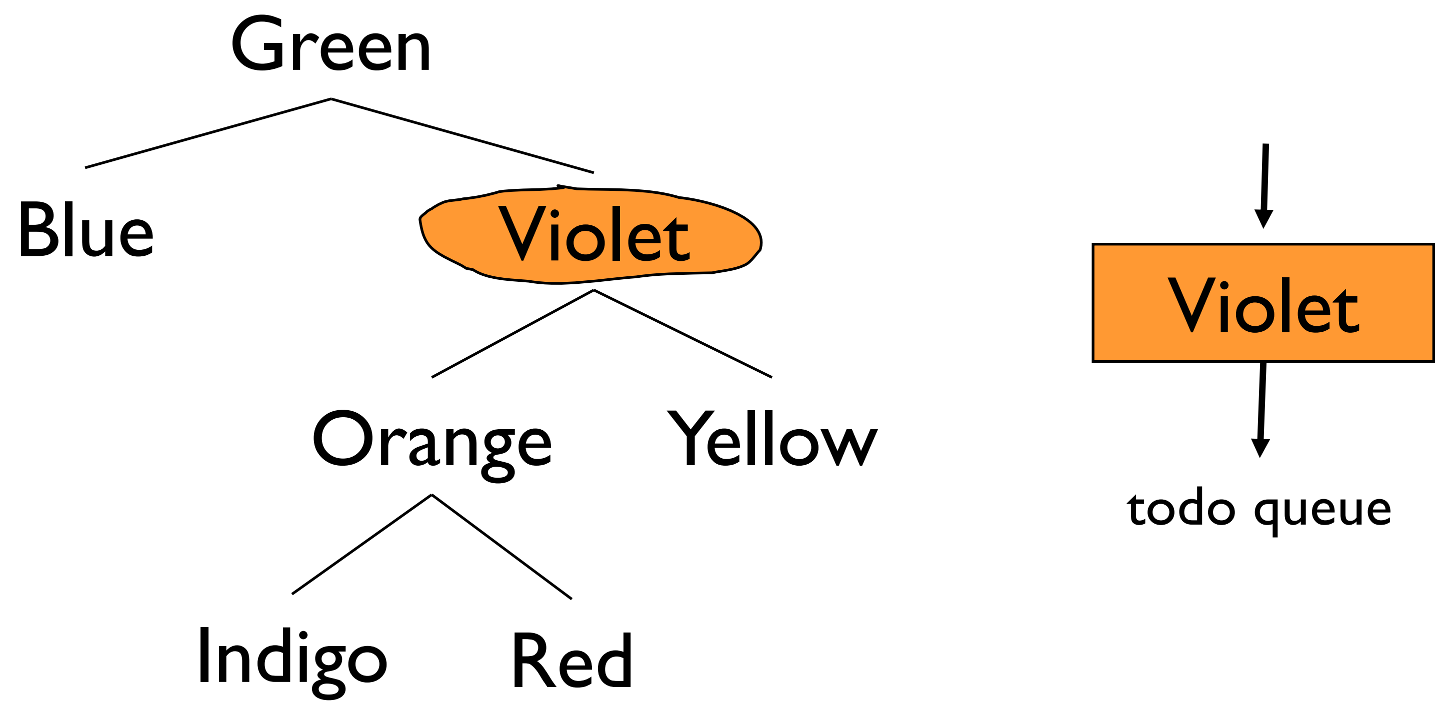


Level-Order Traversal



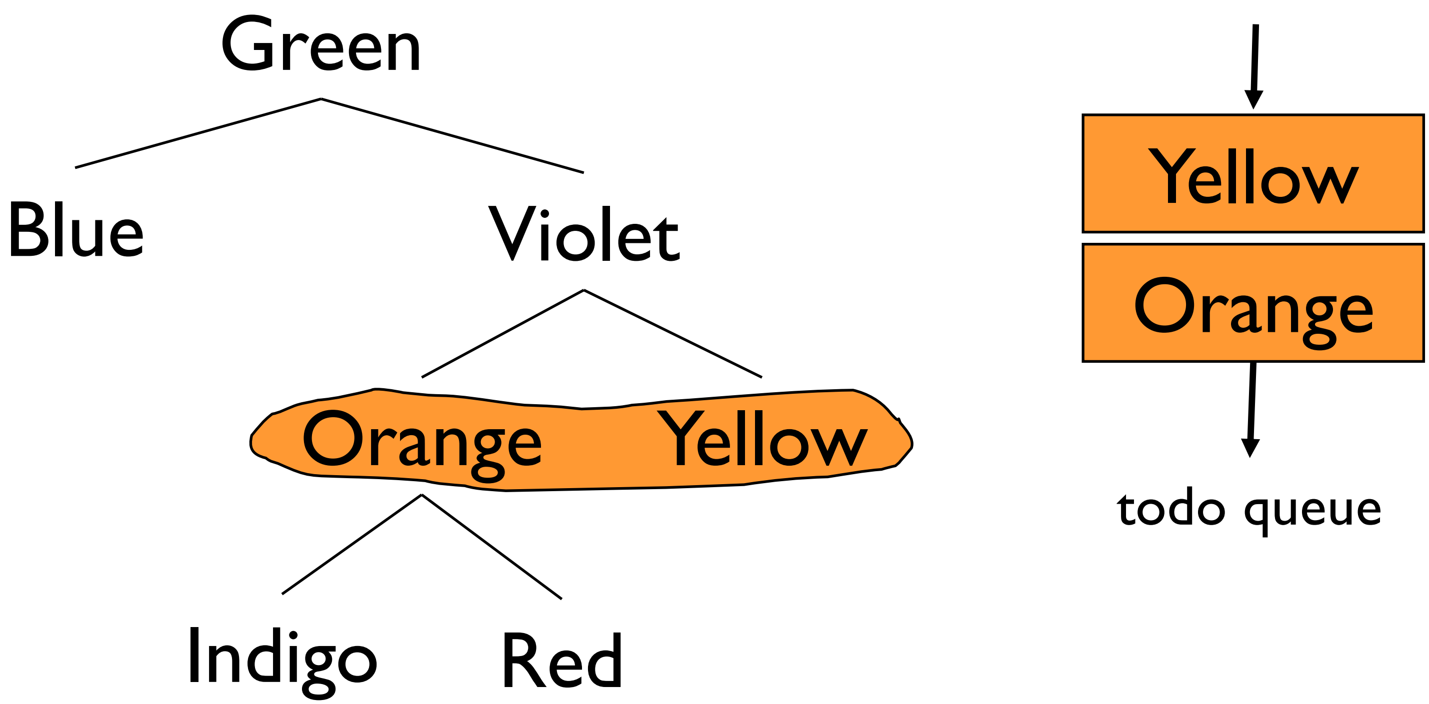
G

Level-Order Traversal



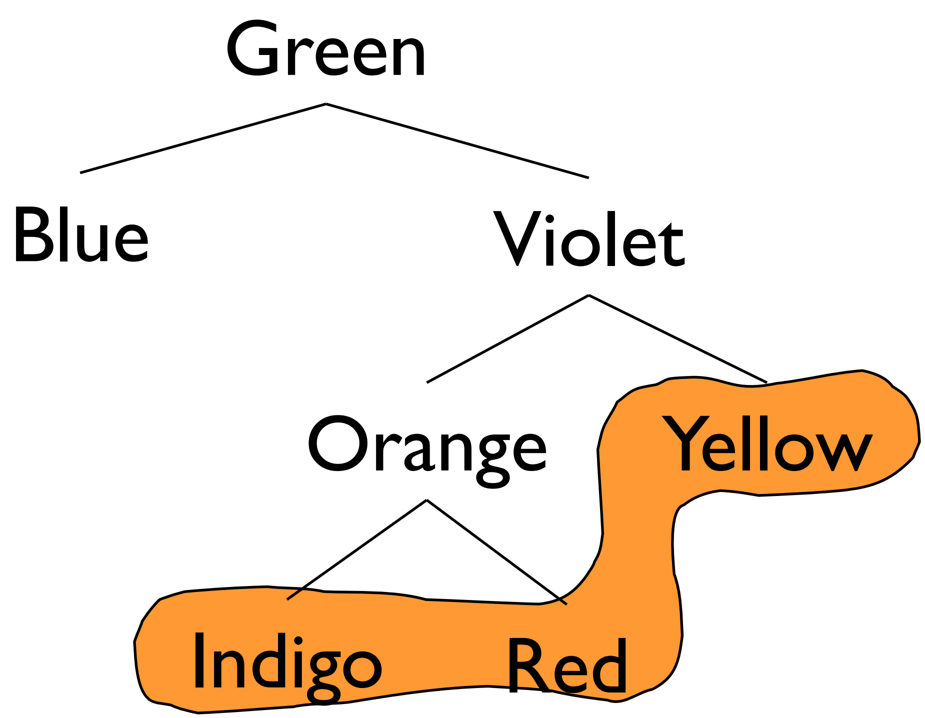
G B

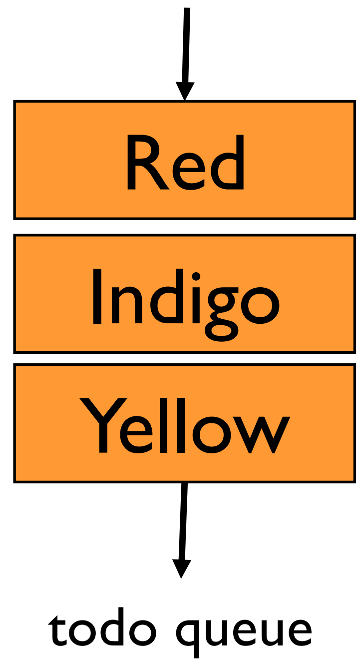
Level-Order Traversal



G B V

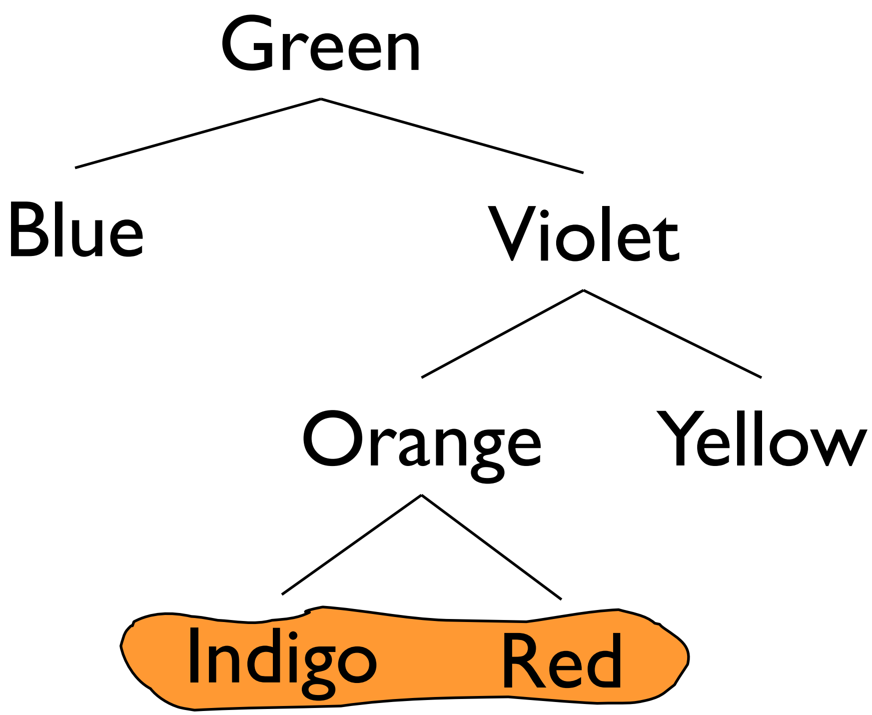
Level-Order Traversal

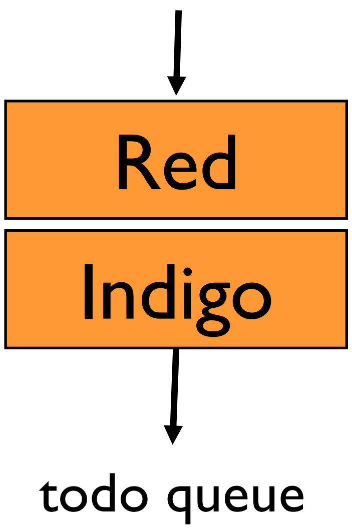




G B V O

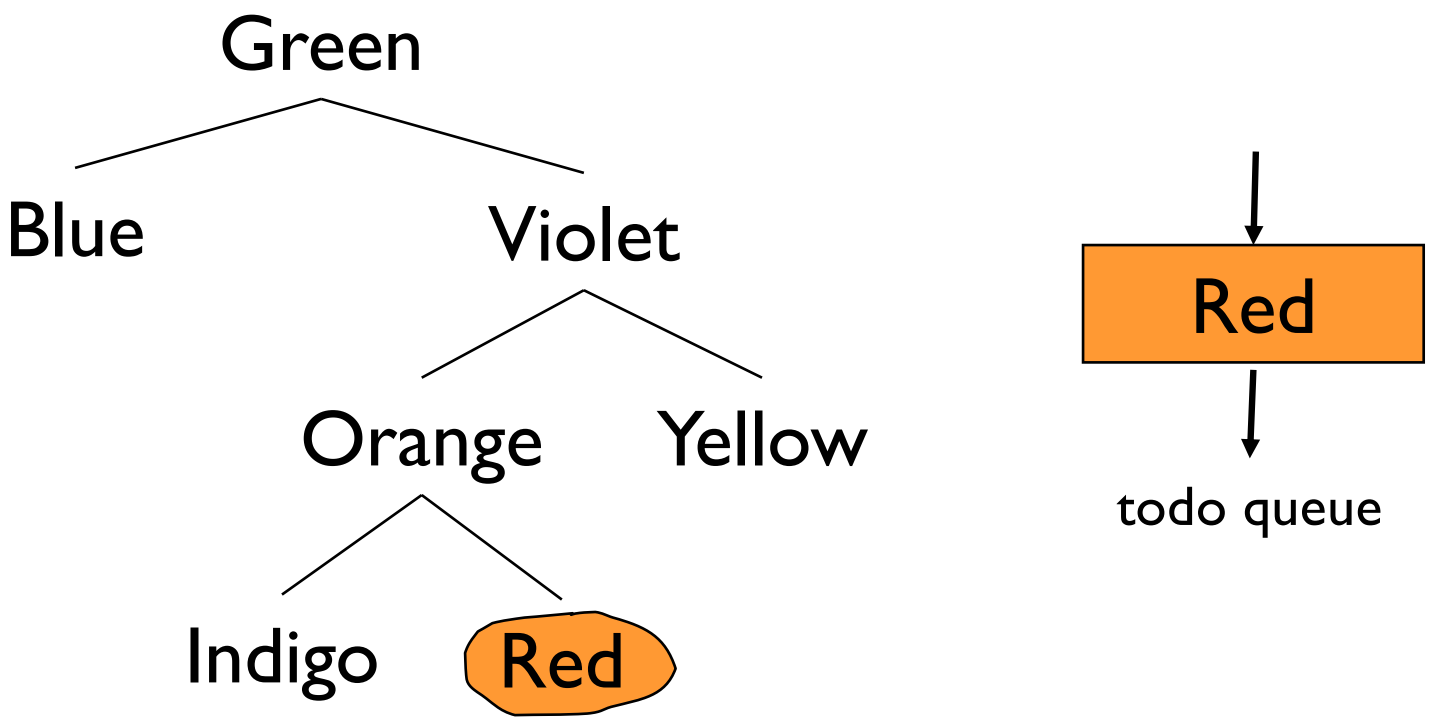
Level-Order Traversal





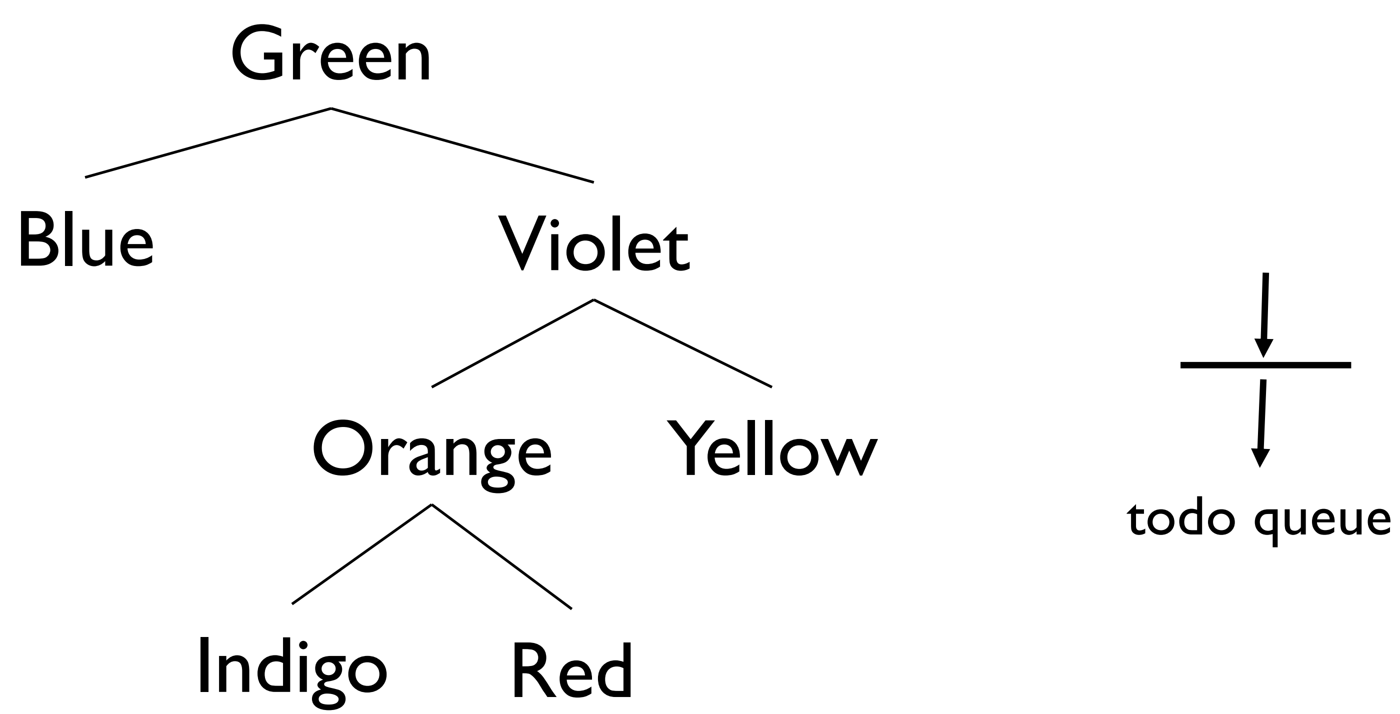
G B V O Y

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()
  - iterator() : calls inorderIterator()

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

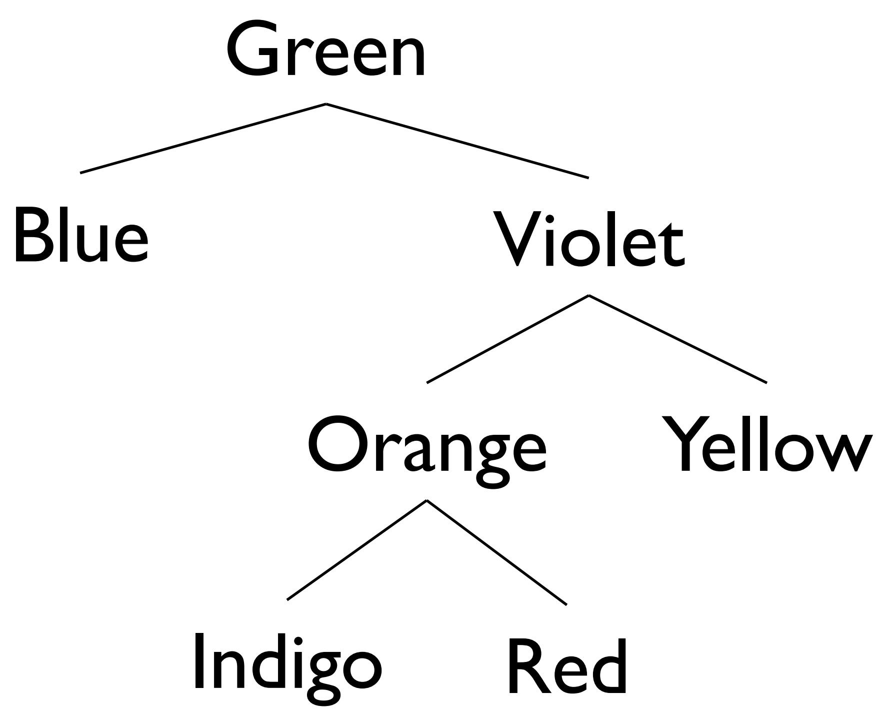
- Basic idea
  - Should return elements in same order as processed by pre-order traversal method
  - Must phrase in terms of next() and hasNext()
  - We “simulate recursion” with stack
    - The stack holds “partially processed” nodes

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()

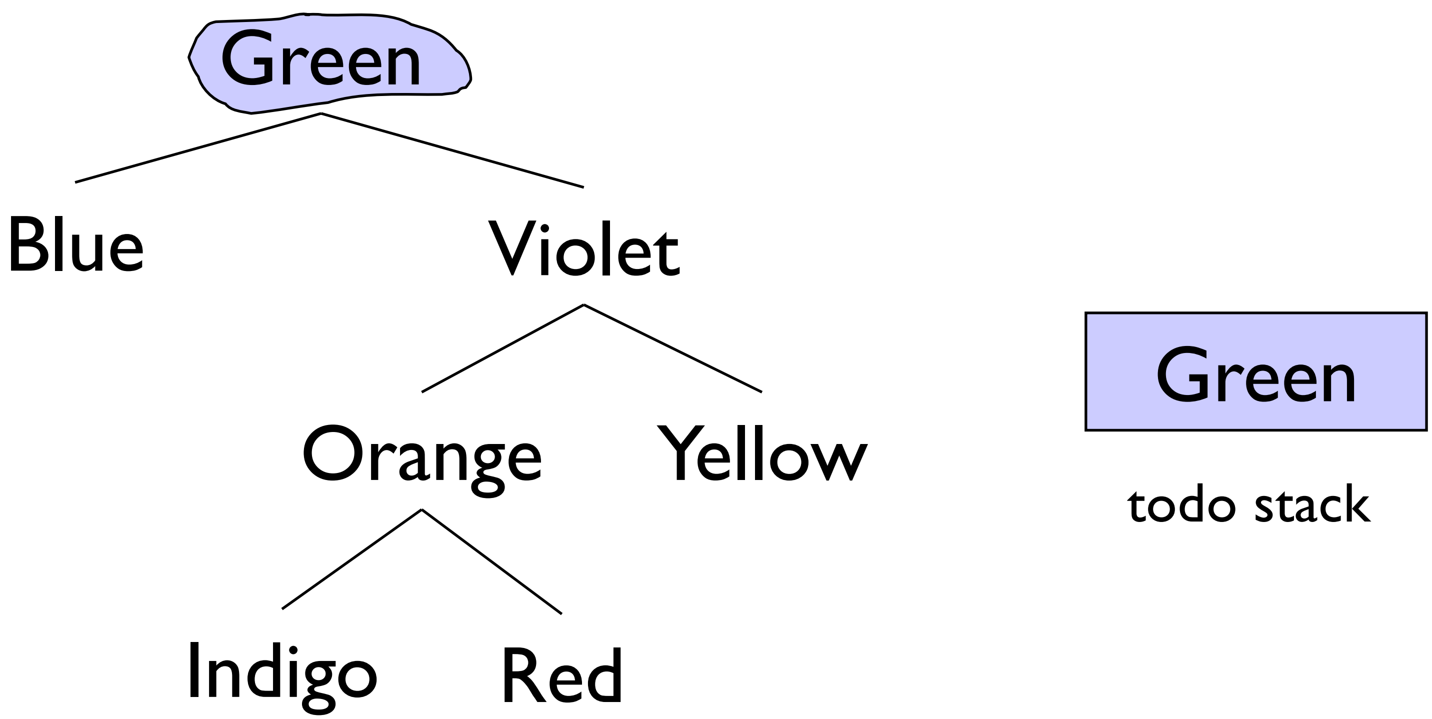
Pre-Order Iterator

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



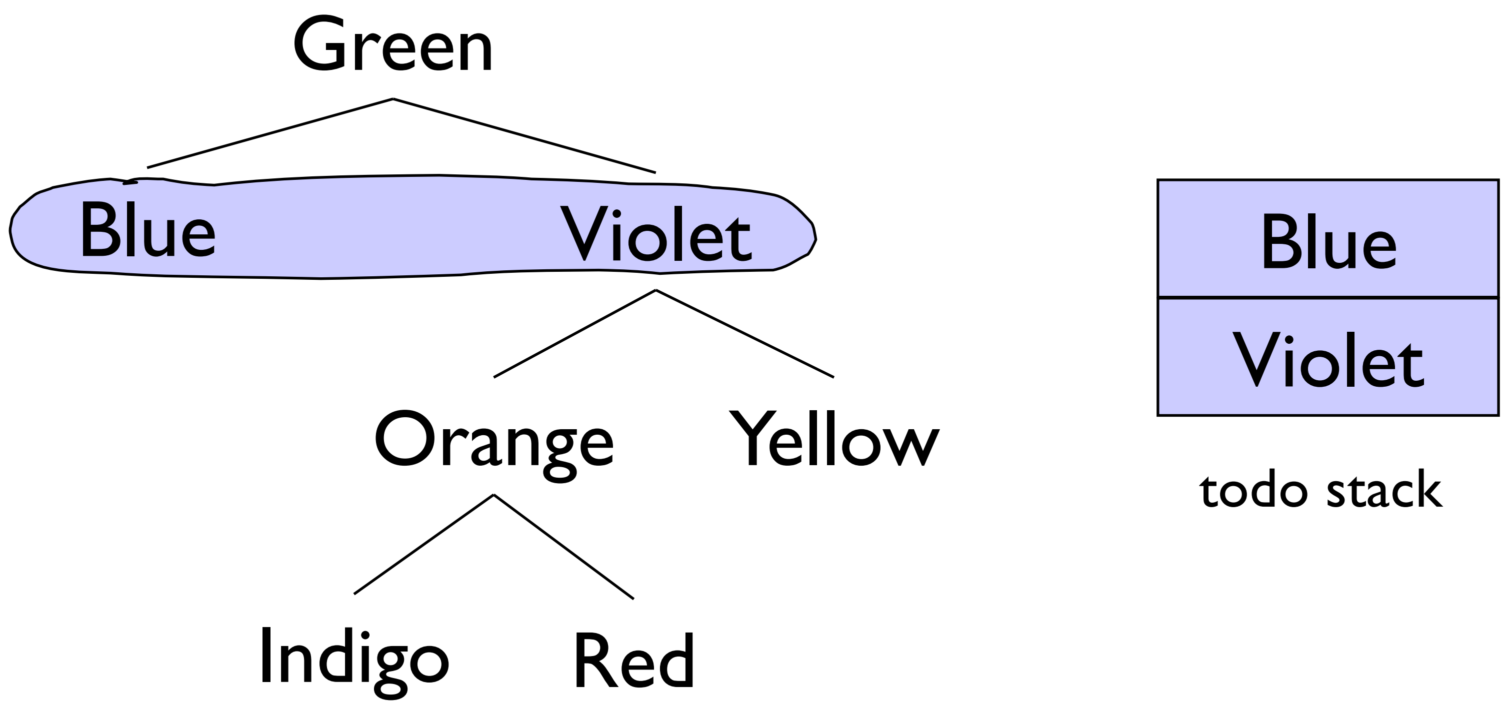
Pre-Order Iterator

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



Pre-Order Iterator

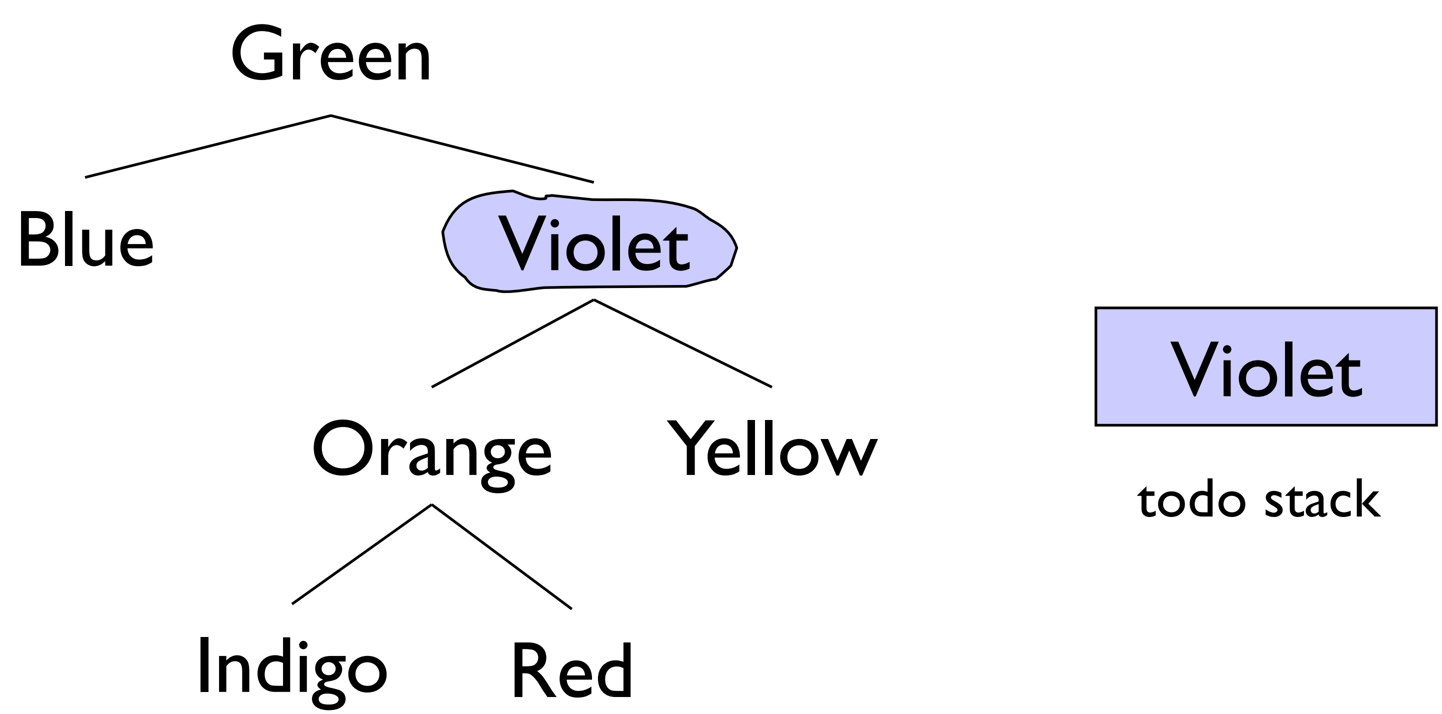
Visit node, then each node in left subtree, then each node in right subtree.



G

Pre-Order Iterator

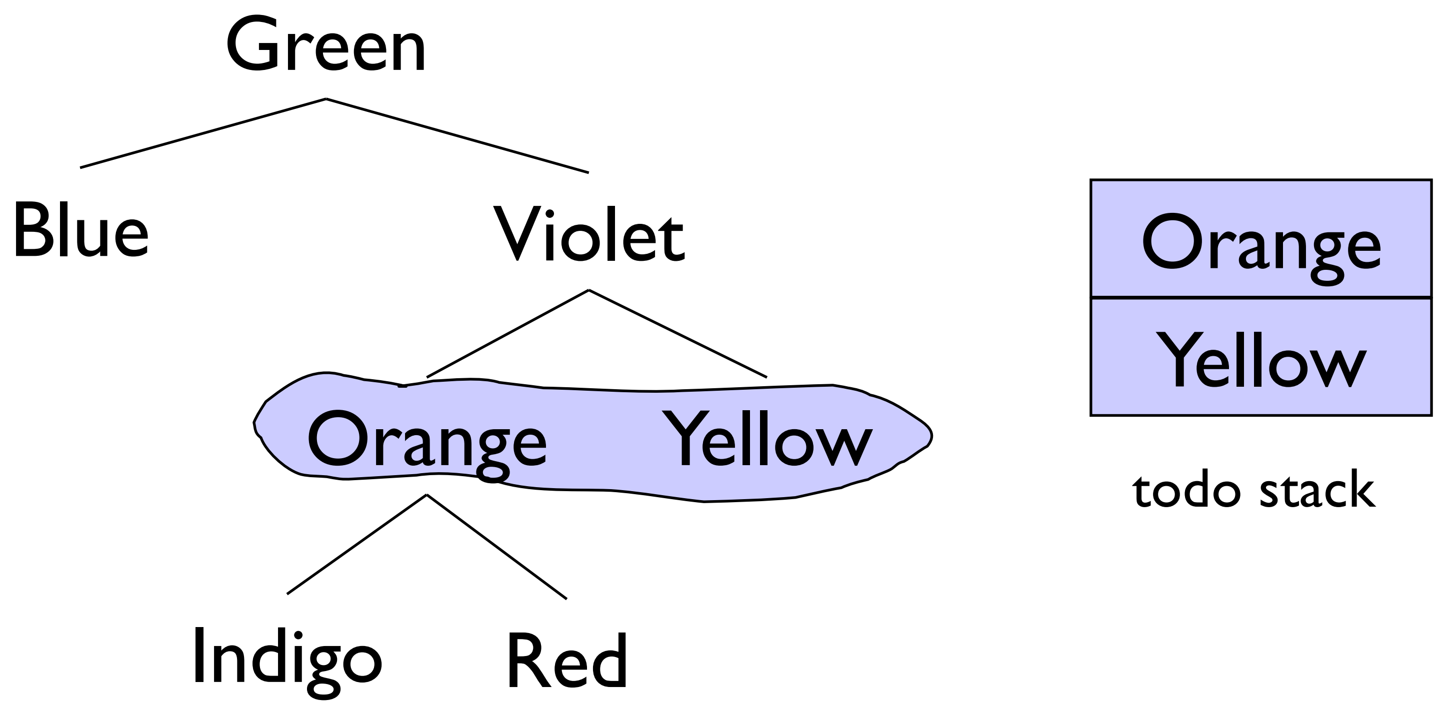
Visit node, then each node in left subtree, then each node in right subtree.



G B

Pre-Order Iterator

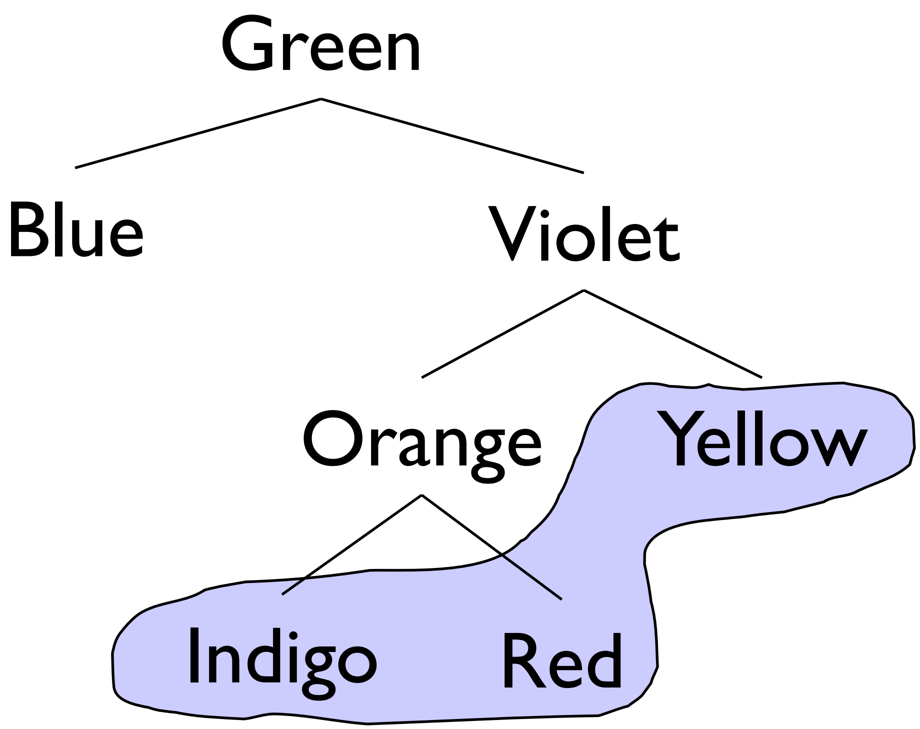
Visit node, then each node in left subtree, then each node in right subtree.

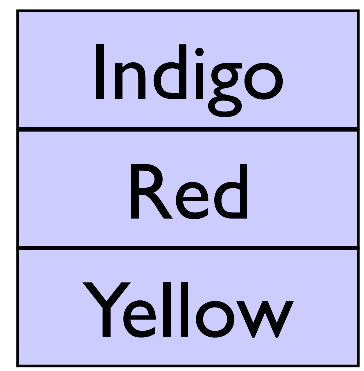


G B V

Pre-Order Iterator

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



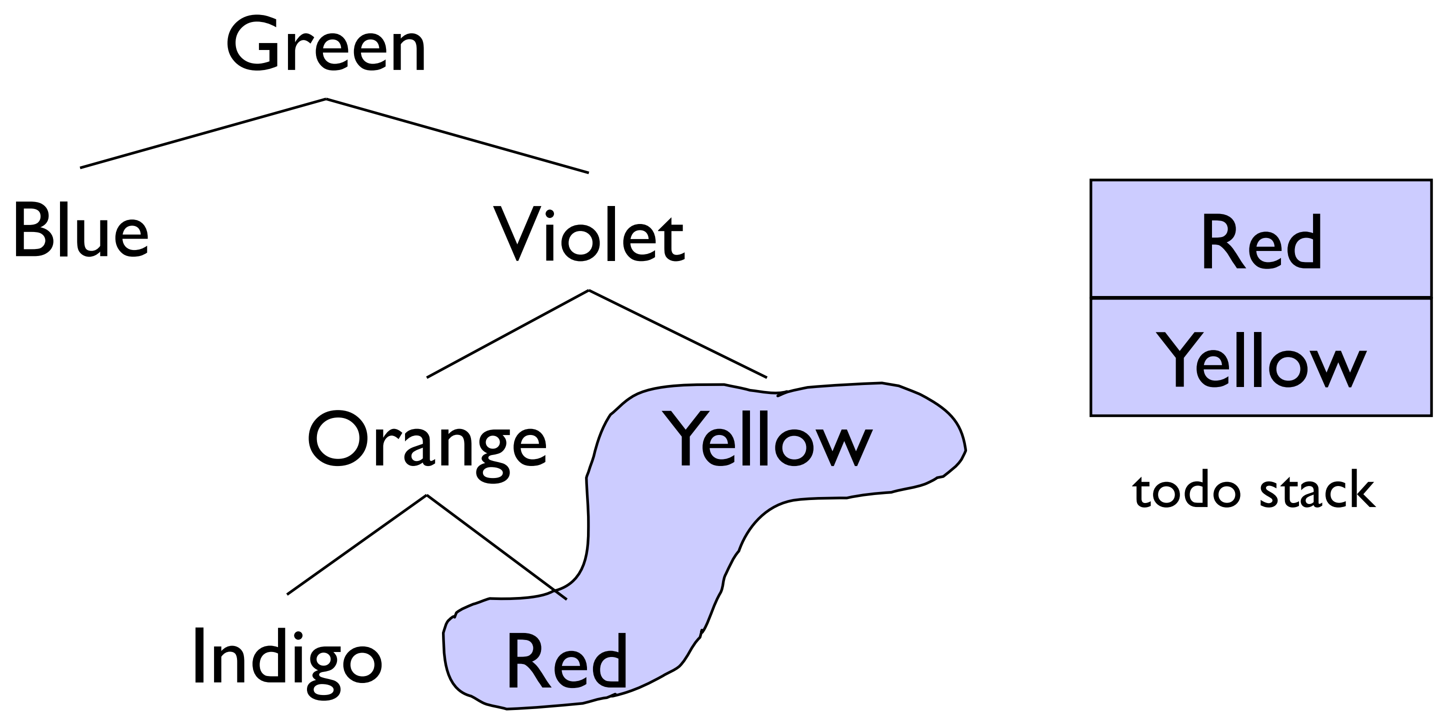


todo stack

G B V O

Pre-Order Iterator

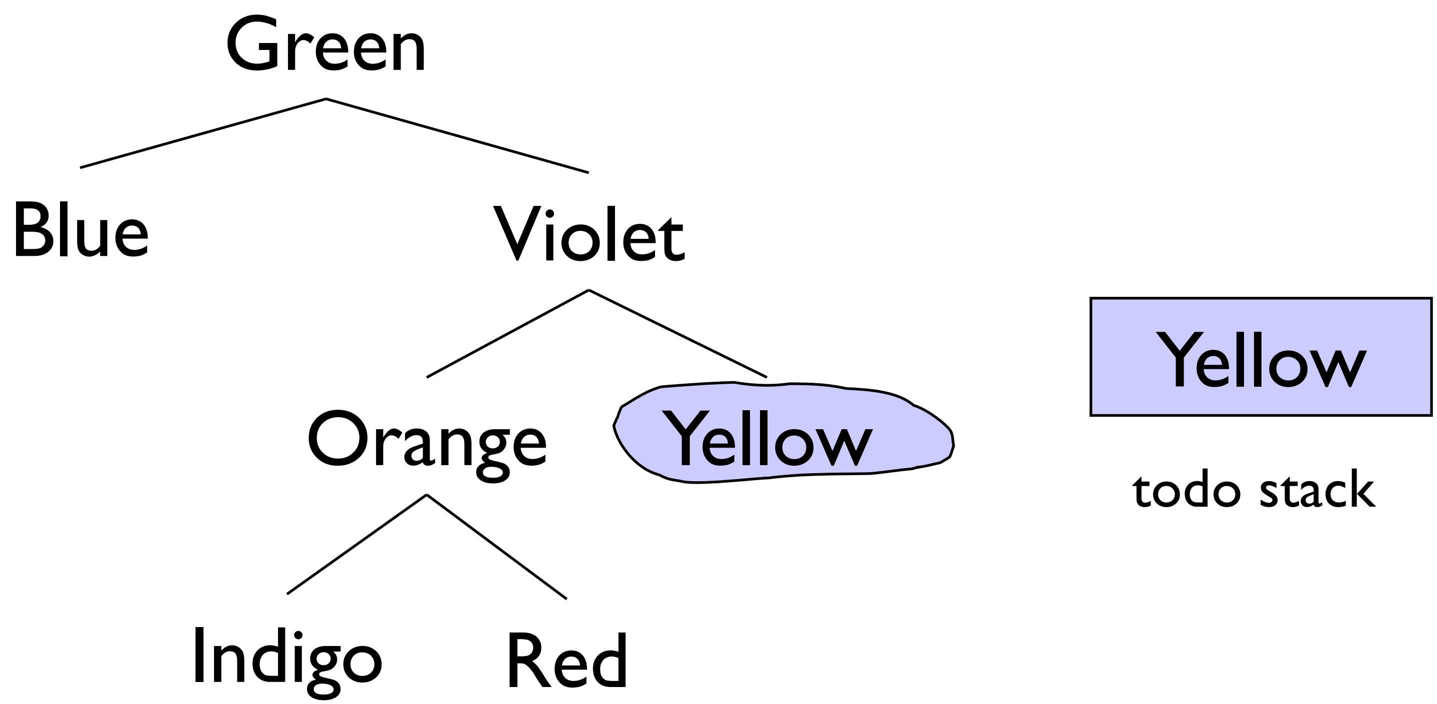
Visit node, then each node in left subtree, then each node in right subtree.



G B V O I

Pre-Order Iterator

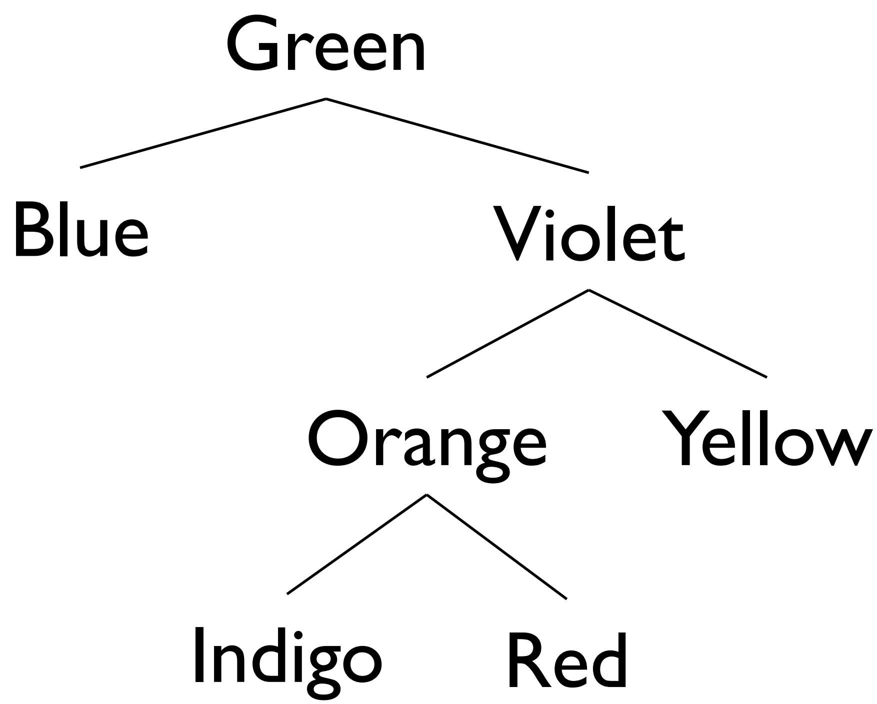
Visit node, then each node in left subtree, then each node in right subtree.



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

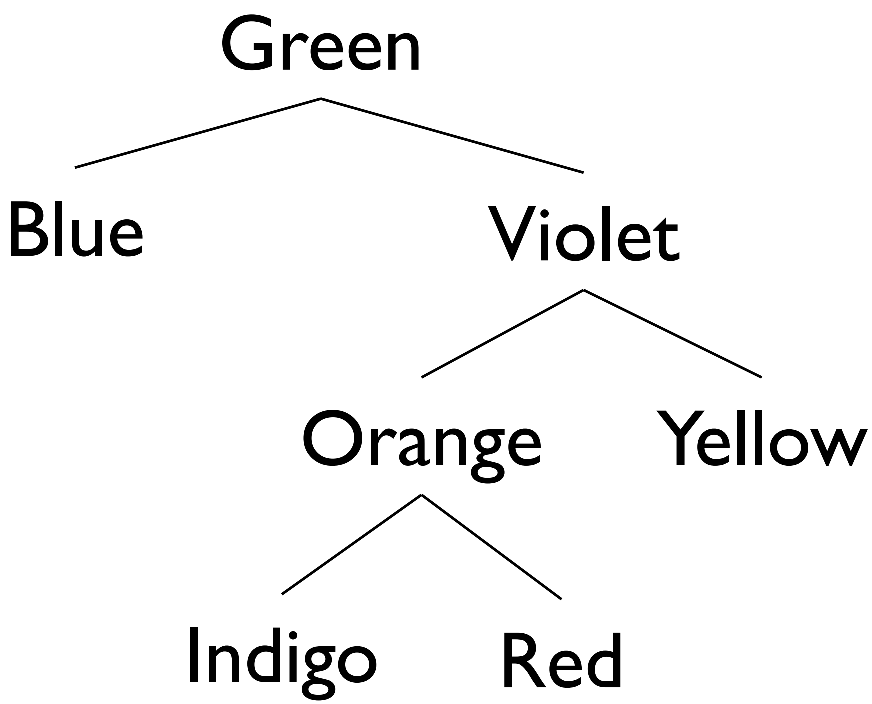
```
public boolean hasNext() {
       return !todo.isEmpty();
}

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

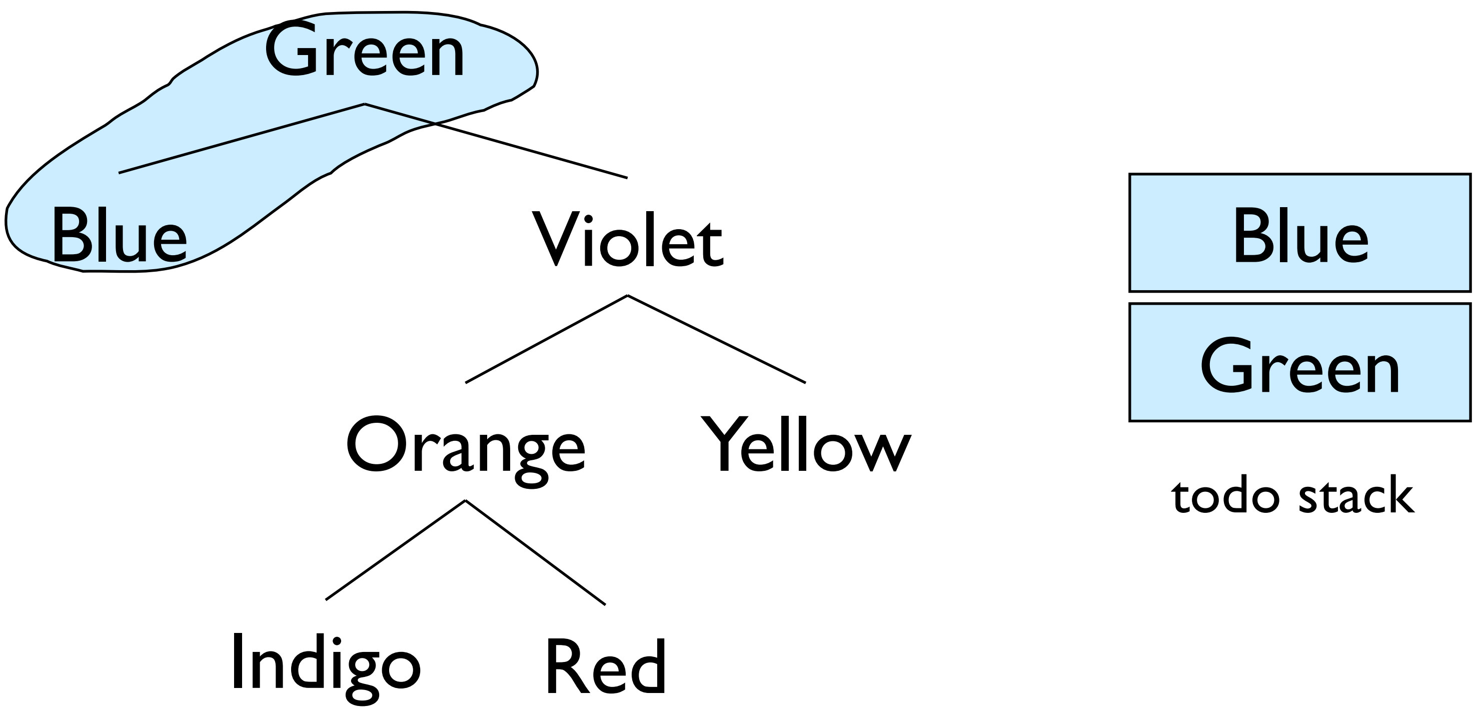
In-Order Iterator

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



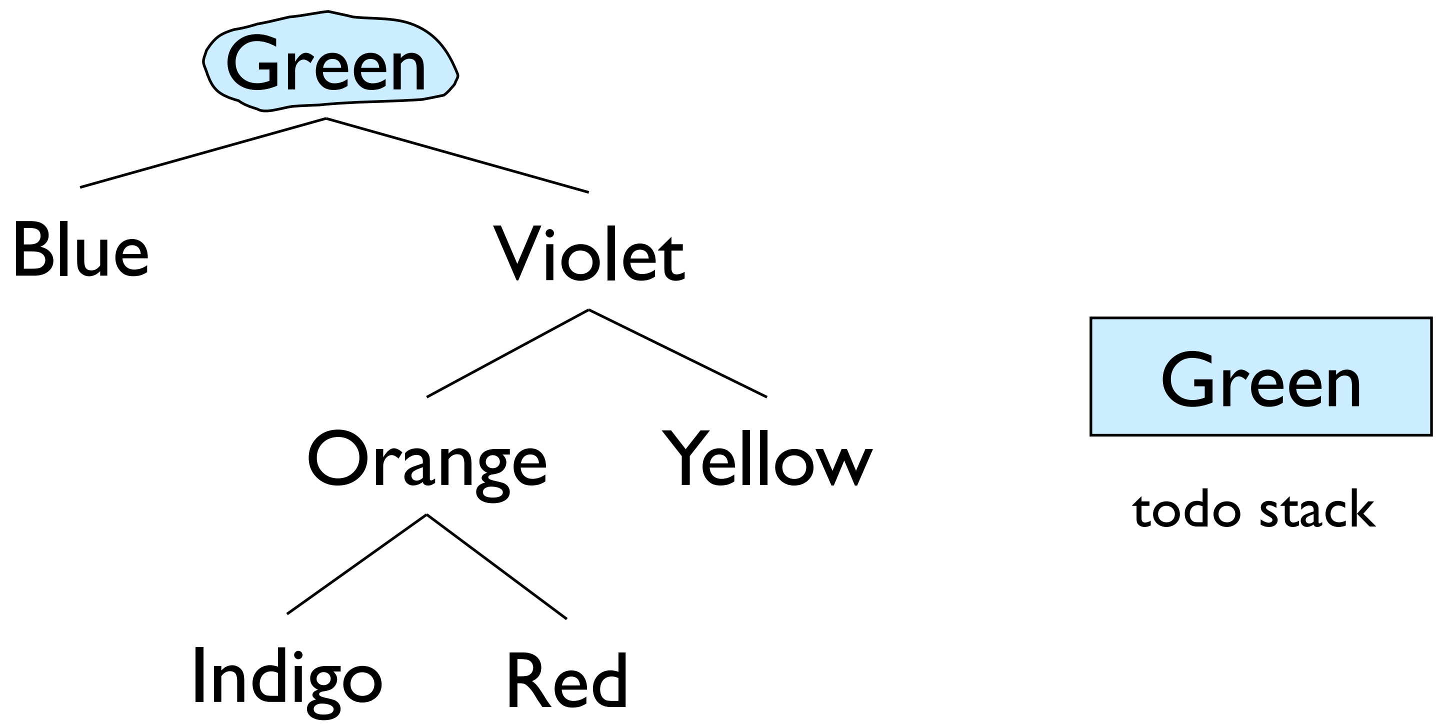
In-Order Iterator

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



In-Order Iterator

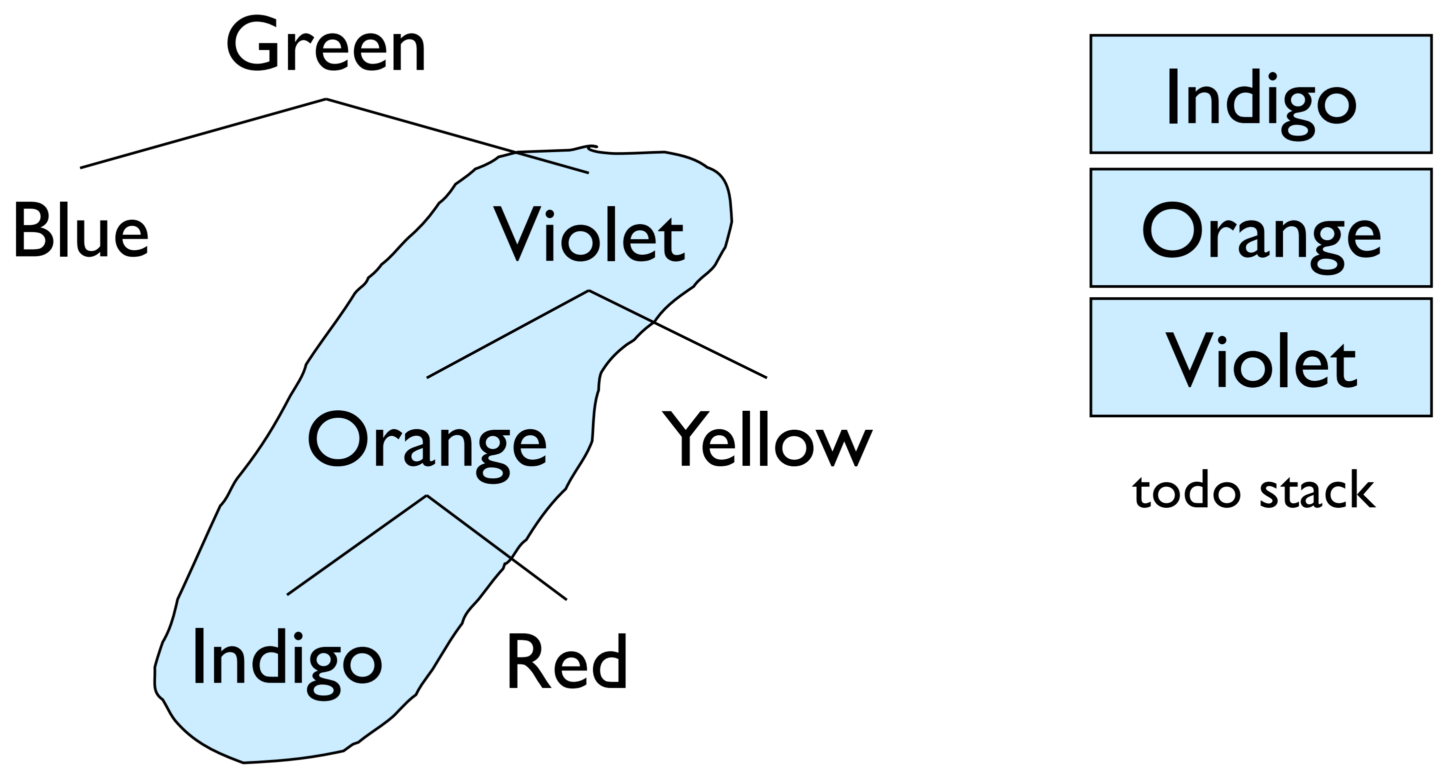
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.



B

In-Order Iterator

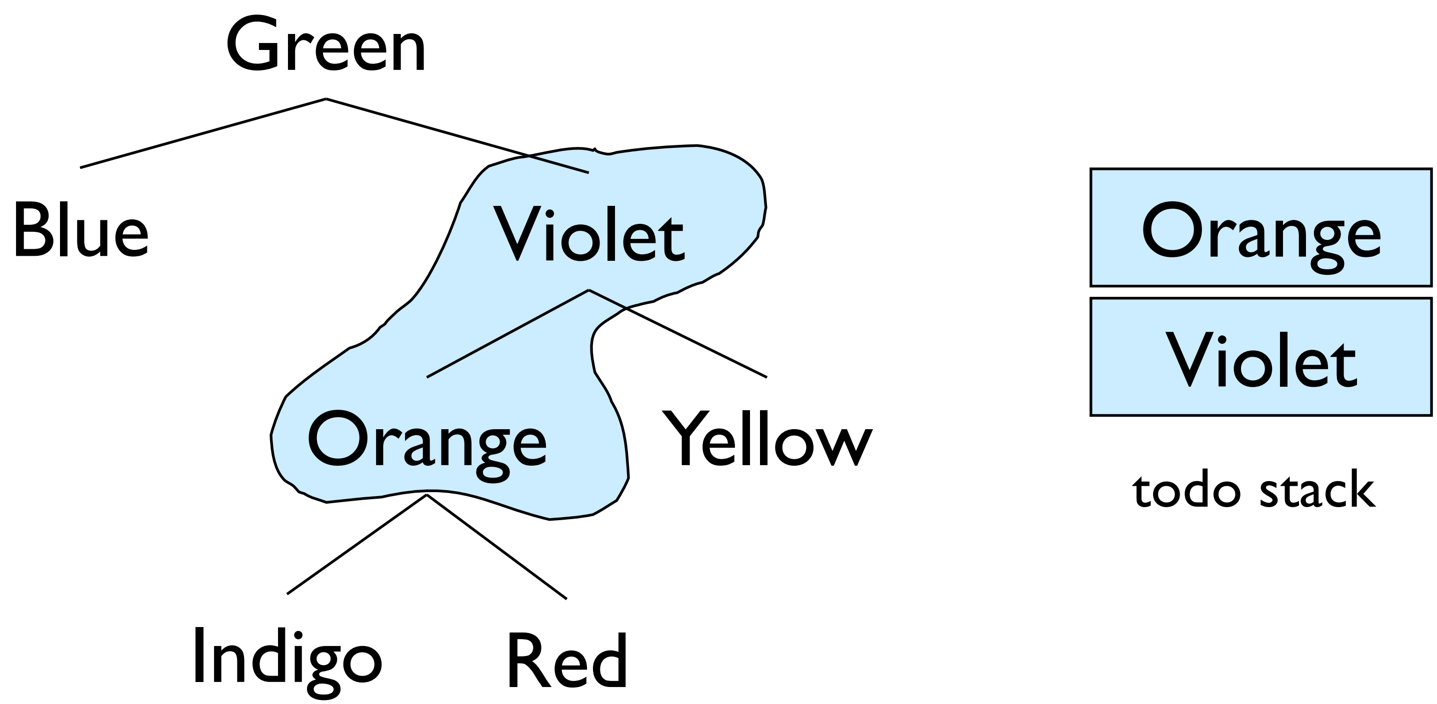
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.



B G

In-Order Iterator

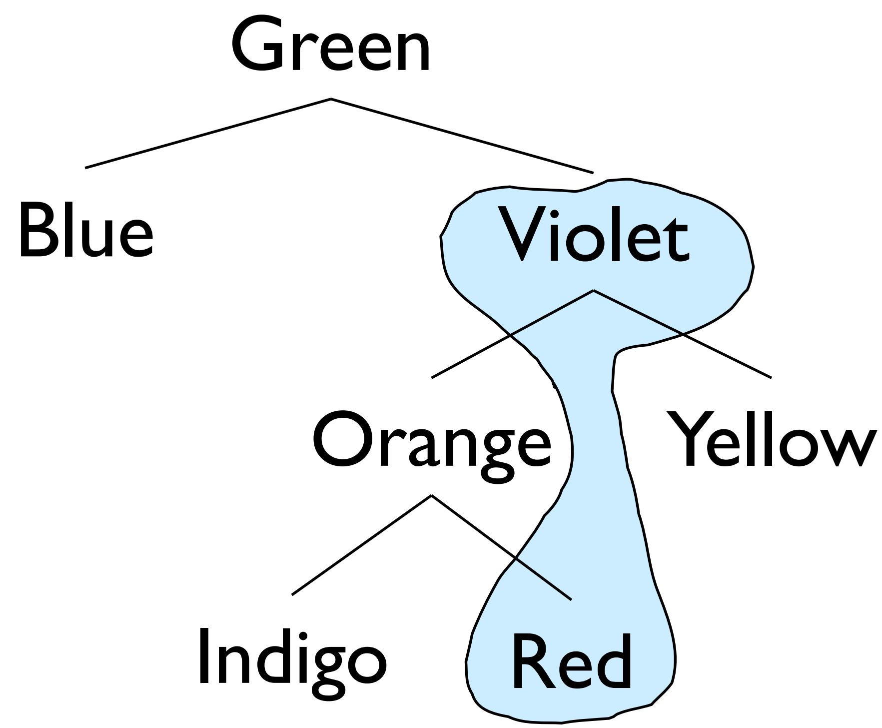
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

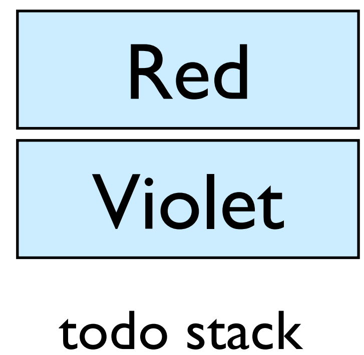


B G I

In-Order Iterator

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

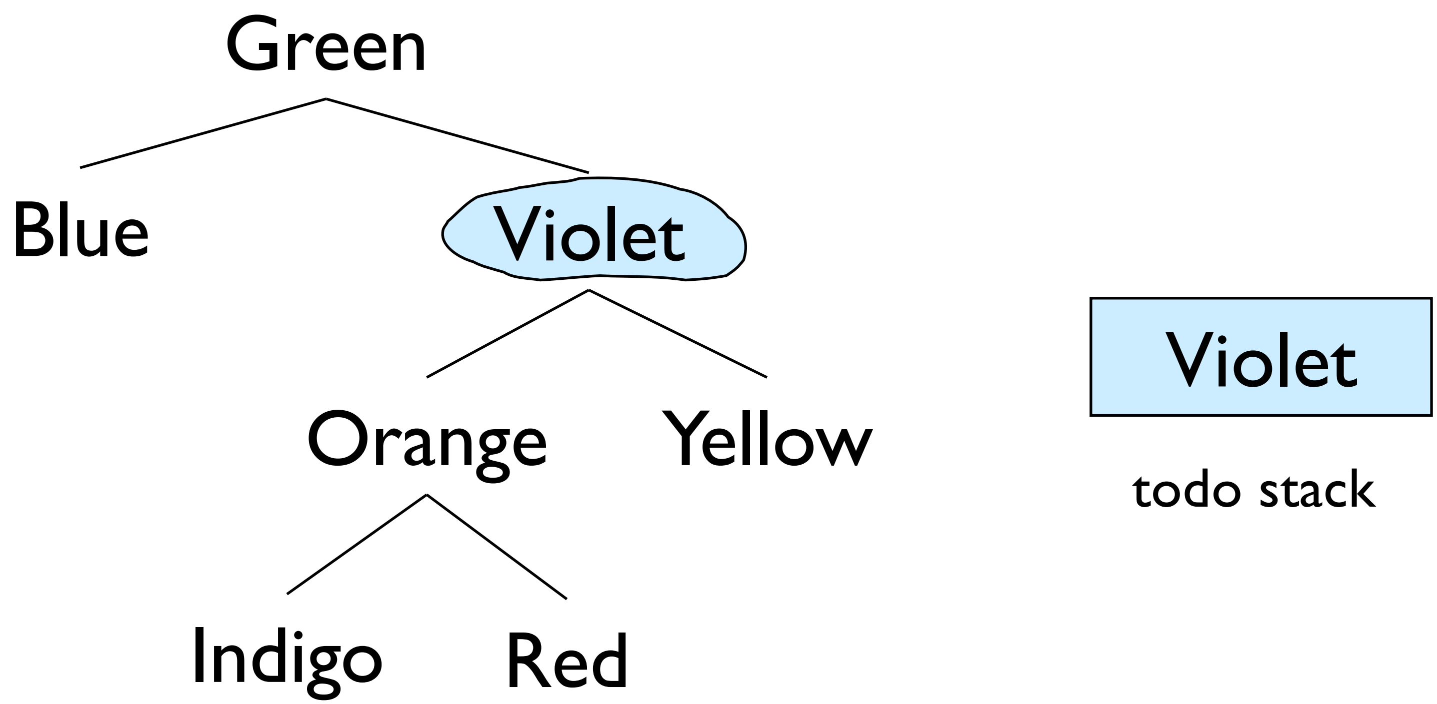




B G I O

In-Order Iterator

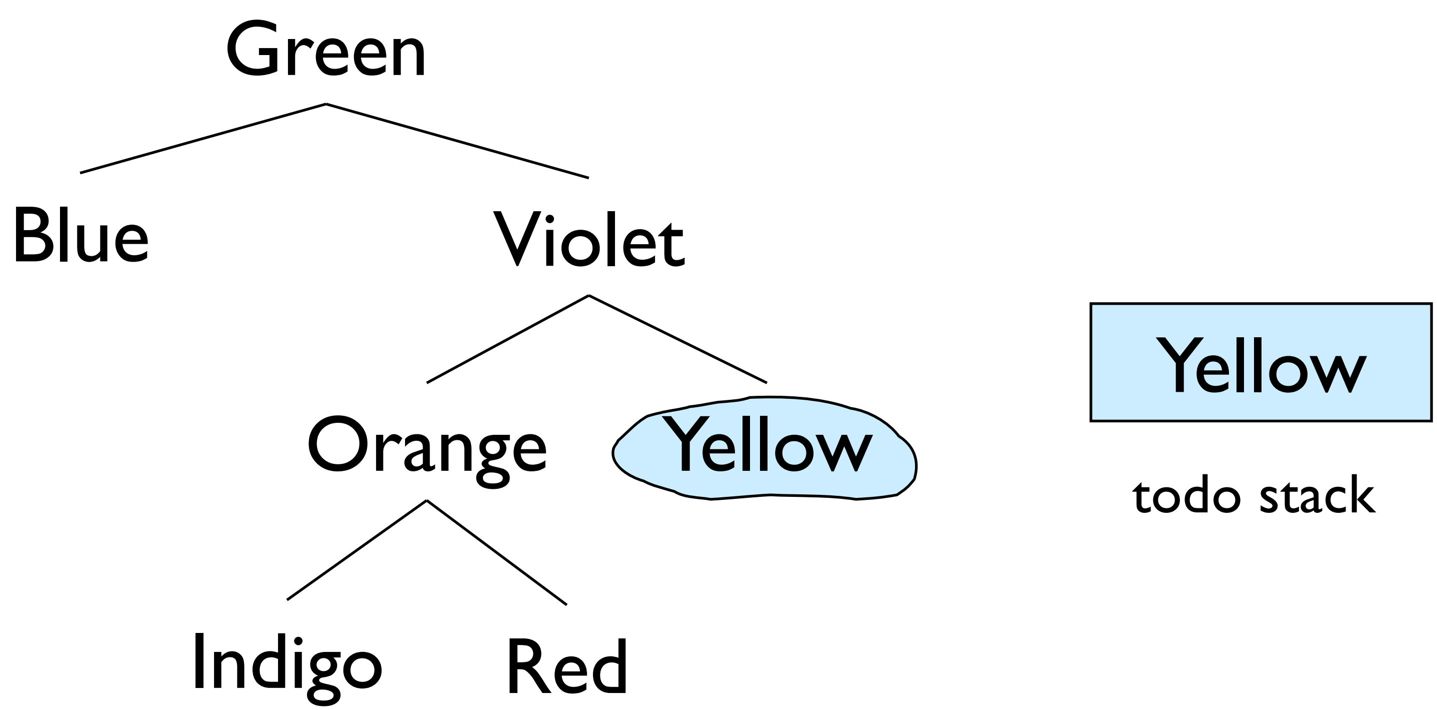
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.



B G I O R

In-Order Iterator

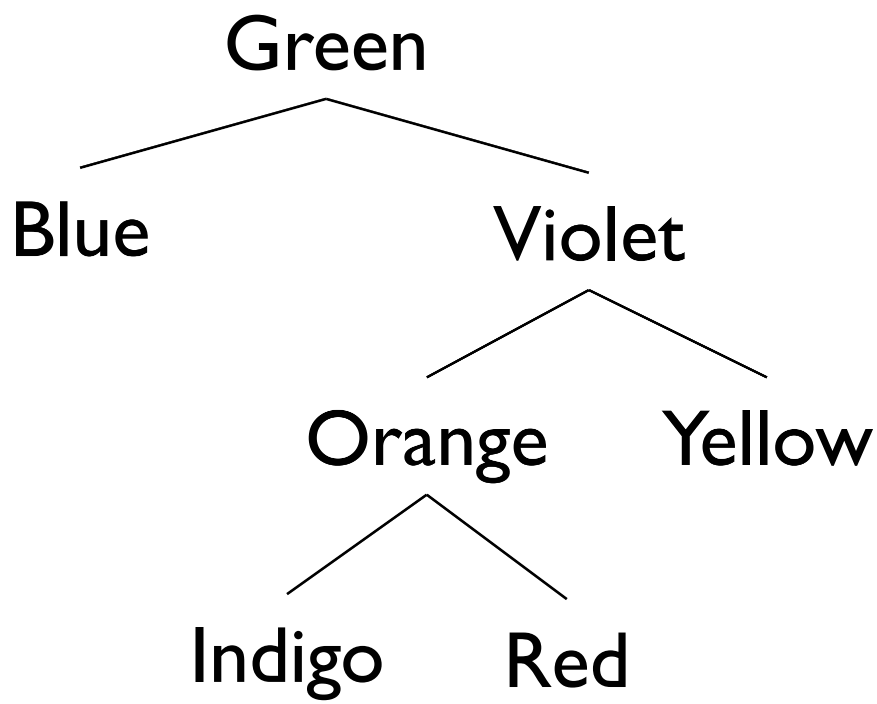
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.



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

```
public BTPostorderIterator(BinaryTree<E> root) {
      todo = new StackList<BinaryTree<E>>();
      this.root = root;
      reset();
}
public void reset() {
      todo.clear();
      BinaryTree<E> current = root;
      while (!current.isEmpty()) {
           todo.push(current);
           if (!current.left().isEmpty())
               current = current.left();
           else
               current = current.right();
       } // Top of stack is now left-most unvisited leaf
    }
```

Post-Order Iterator

```
public E next() {
        BinaryTree<E> current = todo.pop();
        E result = current.value();
        if (!todo.isEmpty()) {
            BinaryTree<E> parent = todo.get();
            if (current == parent.left()) {
                current = parent.right();
                while (!current.isEmpty()) {
                    todo.push(current);
                    if (!current.left().isEmpty())
                         current = current.left();
                    else current = current.right();
                }
            }
        }
        return result;
}
```

Tree Traversals

In summary:

- In-order: “left, node, right”
- Pre-order: “node, left, right”
- Post-order: “left, right, node”

Stack

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

Queue

Traversals & Searching

- We can use traversals for searching trees
- How might we search a tree for a value?
  - Breadth-First: Explore nodes near the root before nodes far away (level order traversal)
    - Nearest gas station
  - Depth-First: Explore nodes deep in the tree first (post-order traversal)
    - Solution to a maze

Loose Ends – Really Big Trees!

- In some situations, the tree we need might be too big or expensive to build completely
  - Or parts of it might not be needed
- Example: Game Trees
  - Chess: you wouldn't build the entire tree, you would grow portions of it as needed (with some combination of depth/breadth first searching)

Lab 7: Representing Numbers

- Humans usually think of numbers in base 10
- But even though we write `int x = 23;` the computer stores `x` as a sequence of `1`s and `0`s
- Recall Lab 3:

```
public static String printInBinary(int n) {
       if (n <= 1)
            return "" + n%2;

       return printInBinary(n/2)+n%2;
}
```

- 00000000 00000000 00000000 00010111

Bitwise Operations

- We can use *bitwise* operations to manipulate the 1s and 0s in the binary representation
  - Bitwise 'and': &
  - Bitwise 'or': |
- Also useful: bit shifts
  - Bit shift left: <<
  - Bit shift right: >>

& and |

- Given two integers `a` and `b`, the bitwise *or* expression `a | b` returns an integer s.t.
  - At each bit position, the result has a `1` if that bit position had a `1` in EITHER `a` OR `b` (or both)
  - `3 | 6 = ?`
- Given two integers `a` and `b`, the bitwise *and* expression `a & b` returns an integer s.t.
  - At each bit position, the result has a `1` if that bit position had a `1` in BOTH `a` AND `b`
  - `3 & 6 = ?`

>> and <<

- Given two integers a and `i`, the expression `(a << i)` returns (a * 2^{i})
  - Why? It shifts all bits left by `i` positions
  - `1 << 4 = ?`
- Given two integers a and `i`, the expression `(a >> i)` returns (a / 2^{i})
  - Why? It shifts all bits right by `i` positions
  - `1 >> 4 = ?`
  - `97 >> 3 = ?`   `(97 = 1100001)`
- Be careful about shifting left and "overflow"!!!

Revisiting printInBinary(int n)

- How would we rewrite a recursive `printInBinary` using bit shifts and bitwise operations?

```
public static String printInBinary(int n) {
    if (n <= 1) {
        return "" + n;
    return printInBinary(n >> 1) + (n & 1);
}
```

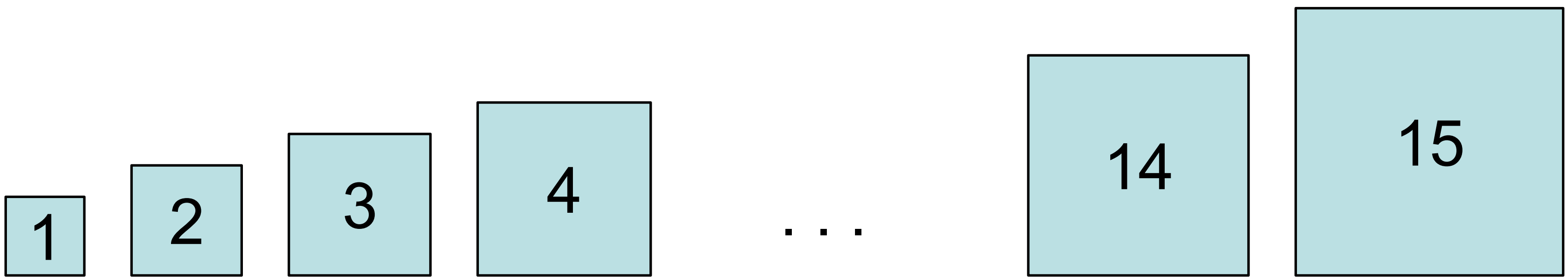
Revisiting printInBinary(int n)

- How would we write an iterative `printInBinary` using bit shifts and bitwise operations?

```
public static String printInBinary(int n,
                                   int width) {
    String result = "";
    for(int i = 0; i < width; i++)
        if ((n & (1<<i)) == 0)
            result = 0 + result;
        else
            result = 1 + result;
    return result;
}
```

Lab 8: Two Towers

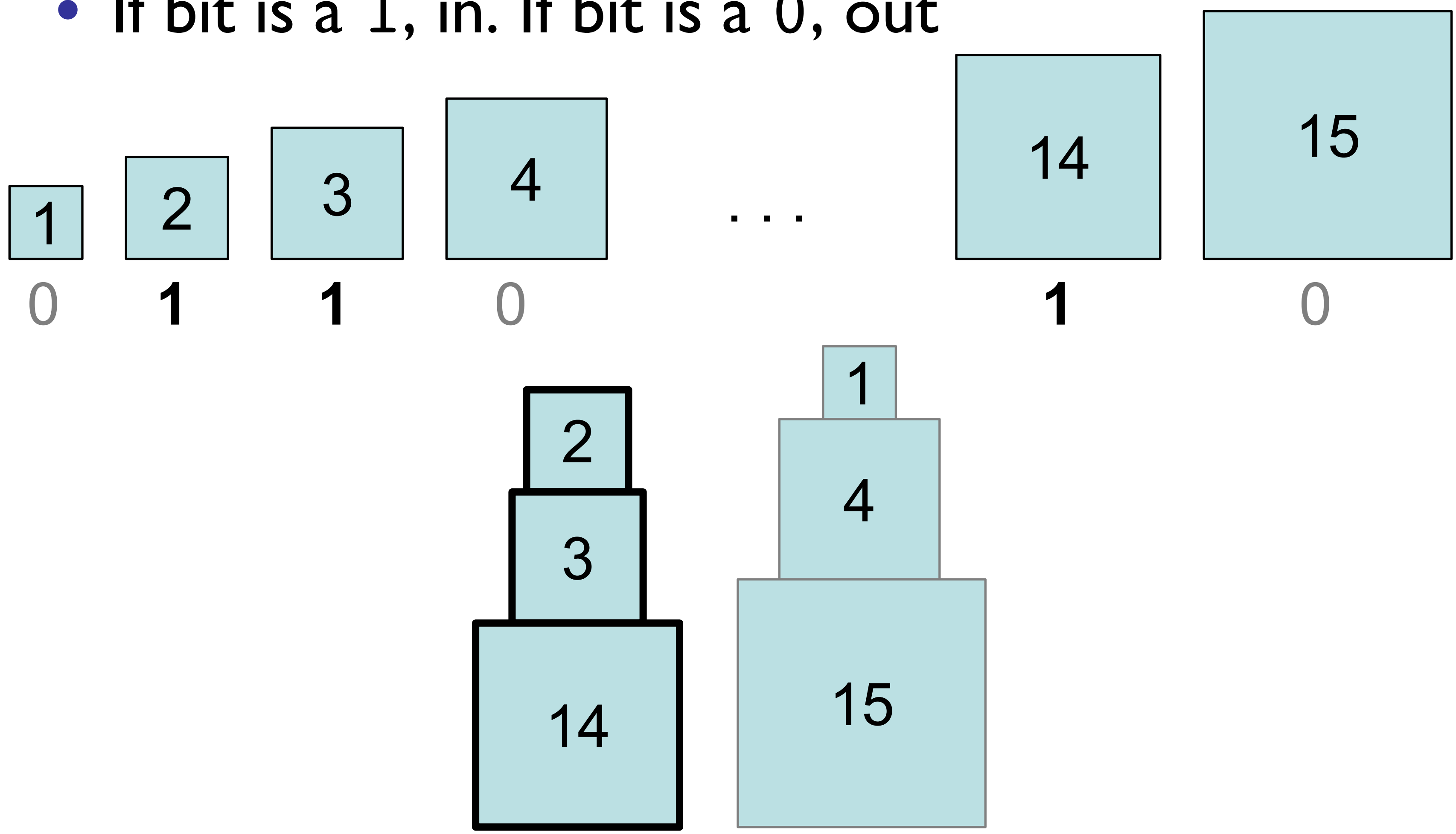
- Goal: given a set of blocks, iterate through all possible subsets to find the *best* set



- "Best" set produces the most balanced towers
- Strategy: create an iterator that uses the bits in a binary number to represent subsets

Lab 8: Two Towers

- A block can either be in the set or out
  - If bit is a `1`, in. If bit is a `0`, out



Questions?

- We will write a "SubsetIterator" to enumerate all possible subsets of a Vector<E>
- We will use SubsetIterator to solve this problem
- Can also be used to solve other problems
  - Identify all Subsequences of a String that are words
    - You just need a dictionary of legal words
    - Coming soon!