CSCI 136
Data Structures &
Advanced Programming

Traversing Trees using Iterators

Designing Tree Iterators

- Goal: design iterators to dispense items in the same order that the different tree traversal algorithms visit nodes.

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

Implementing the Iterators

- Iterators should dispense values in same order as corresponding the traversal method
- Challenge: We must phrase algorithm steps in terms of `next()` and `hasNext()`
  - Recursive methods don't convert as easily, so, let's start with the most "straightforward" traversal order: level-order!

Level-Order Iterator

- Should return elements in same order as processed by level-order traversal method
  - Visit all nodes at depth *i* before visiting any node at depth *i+1*
- Must phrase in terms of `next()` and `hasNext()`
  - Basic Idea: We "capture" our traversal in a queue
    - The queue holds "to be visited" nodes

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

- Should return elements in same order as processed by pre-order traversal method:
  - Visit node, then left subtree, then right subtree
- Must phrase in terms of `next()` and `hasNext()`

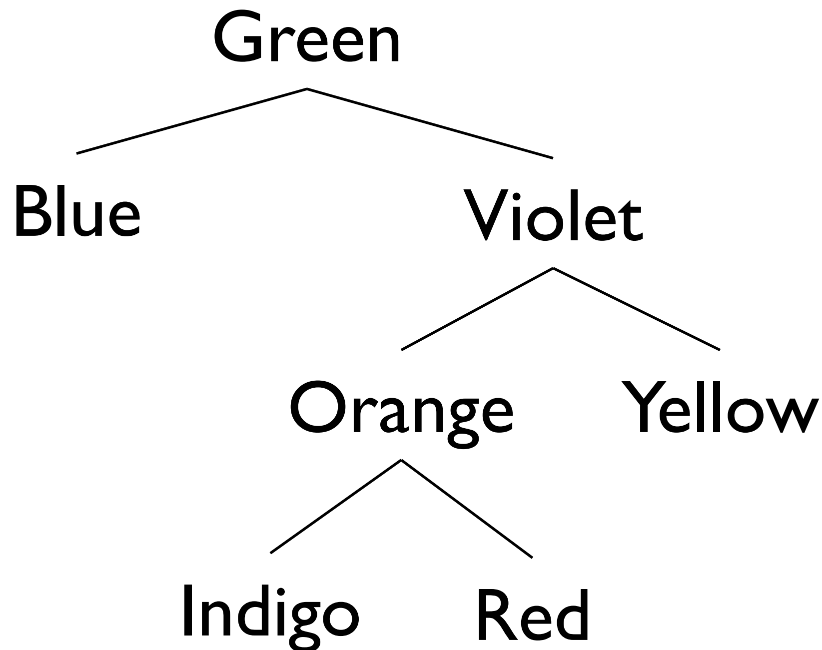
- Basic idea: We “simulate recursion” with stack
  - The stack holds “partially processed” nodes

Pre-Order Iterator

- Order: node -> left subtree -> right subtree
  1. Constructor: Push root onto TODO stack
  2. On call to `next()`:
     - Pop node from TODO stack
     - Push right and then left nodes of popped node onto TODO stack
     - Return popped node's value
  3. On call to `hasNext()`:
     - return `!stack.isEmpty()`

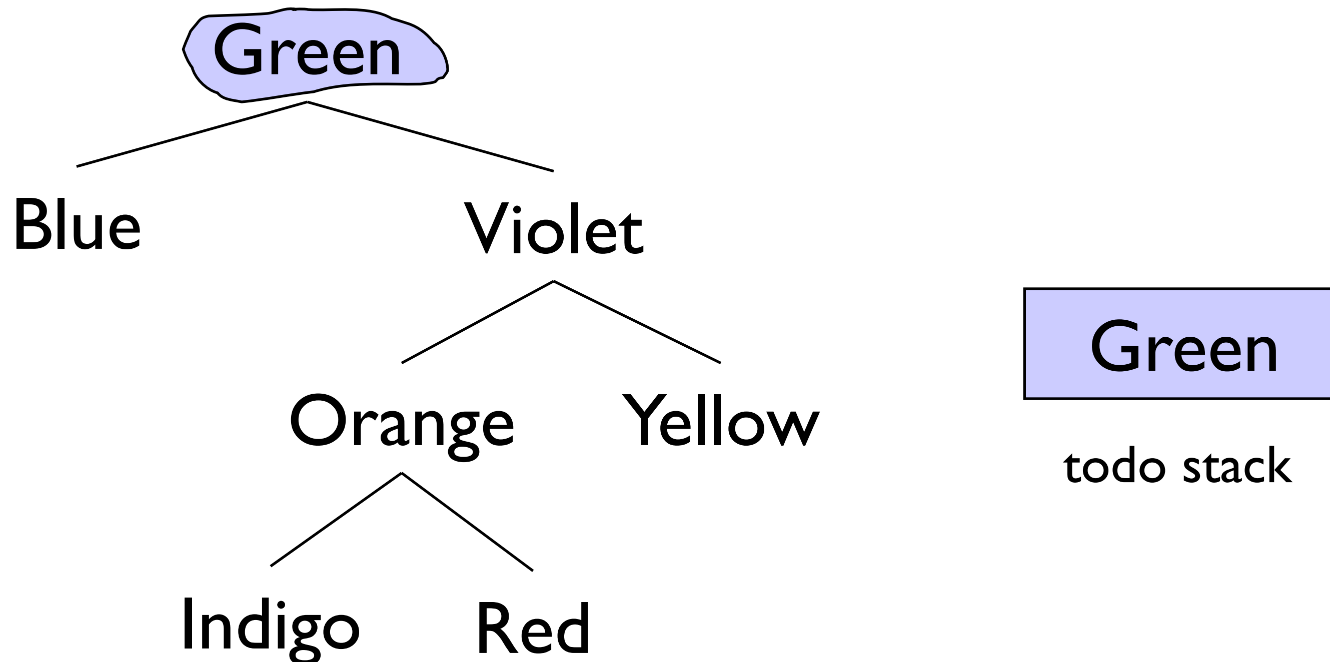
Pre-Order Iterator

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



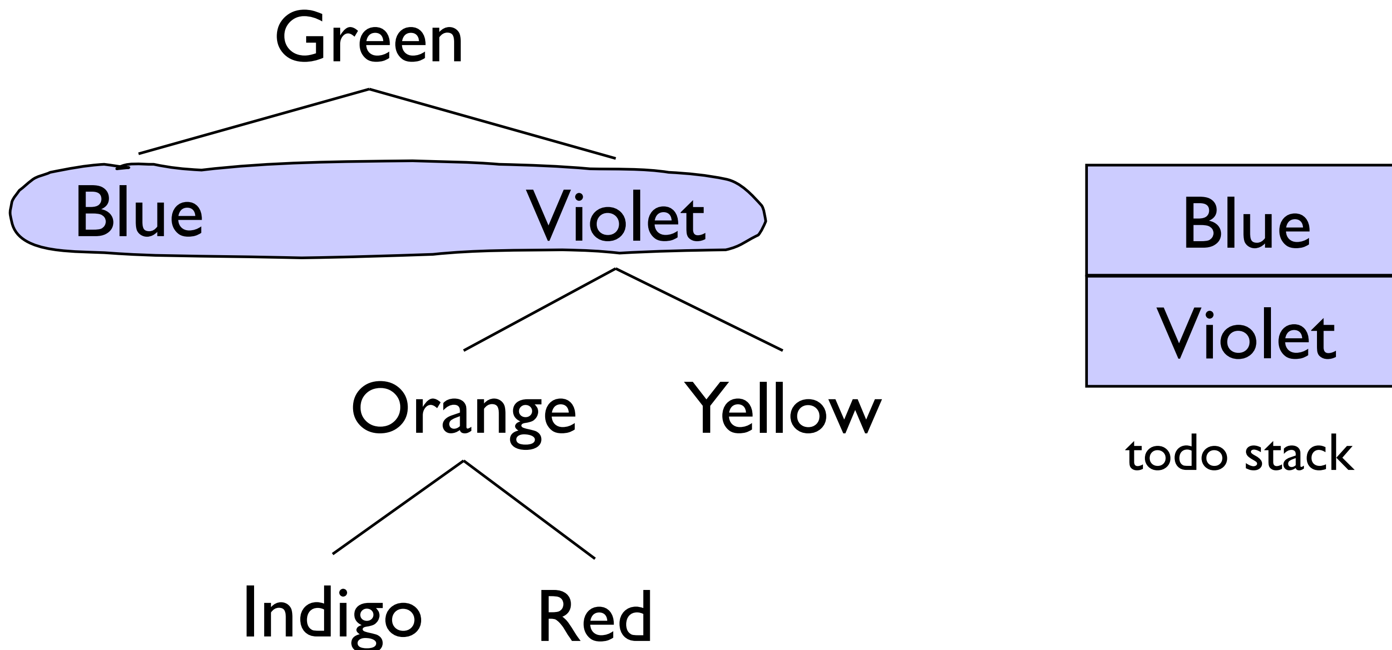
Pre-Order Iterator

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



Pre-Order Iterator

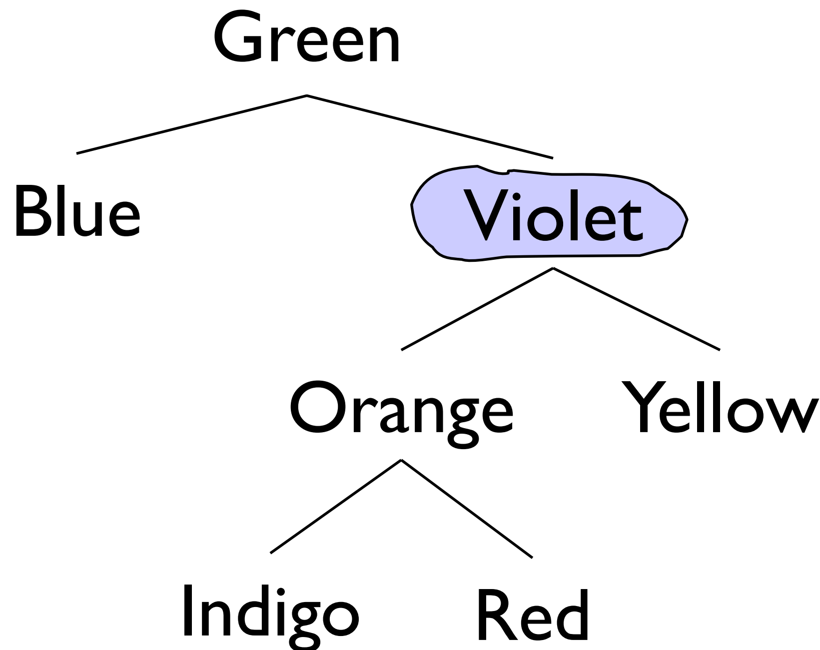
Visit node, then each node in left subtree, then each node in right subtree.



G

Pre-Order Iterator

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

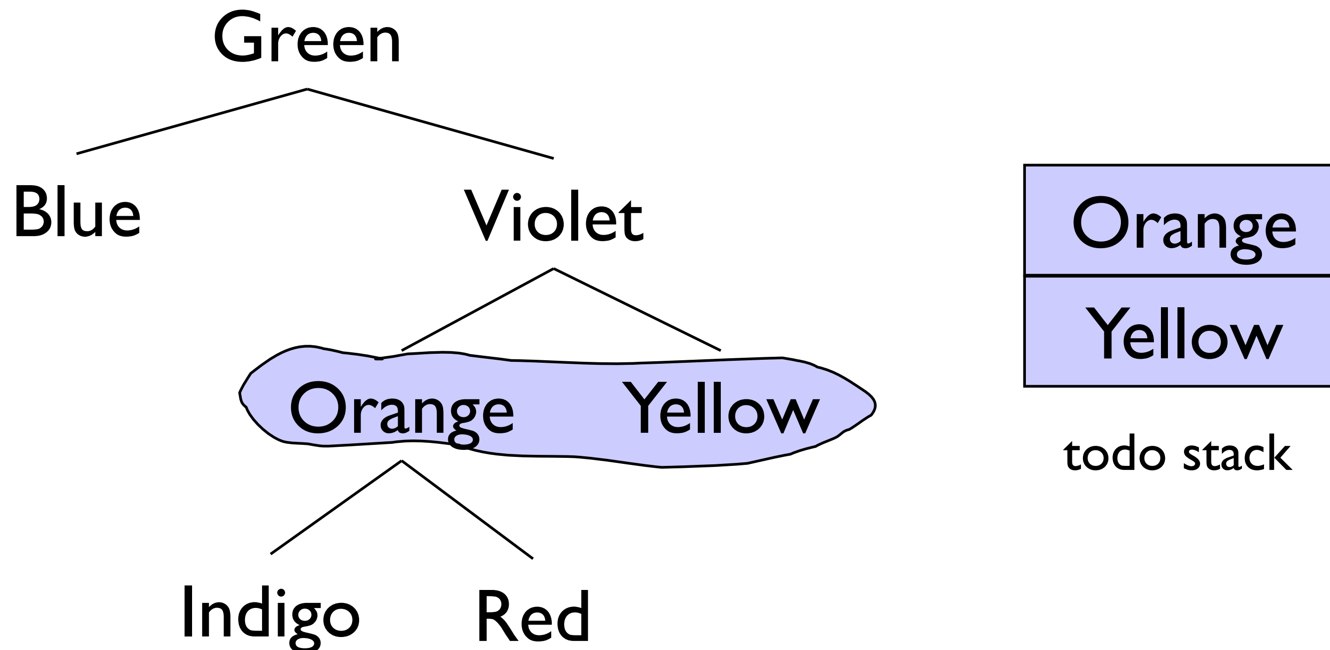


todo stack

G B

Pre-Order Iterator

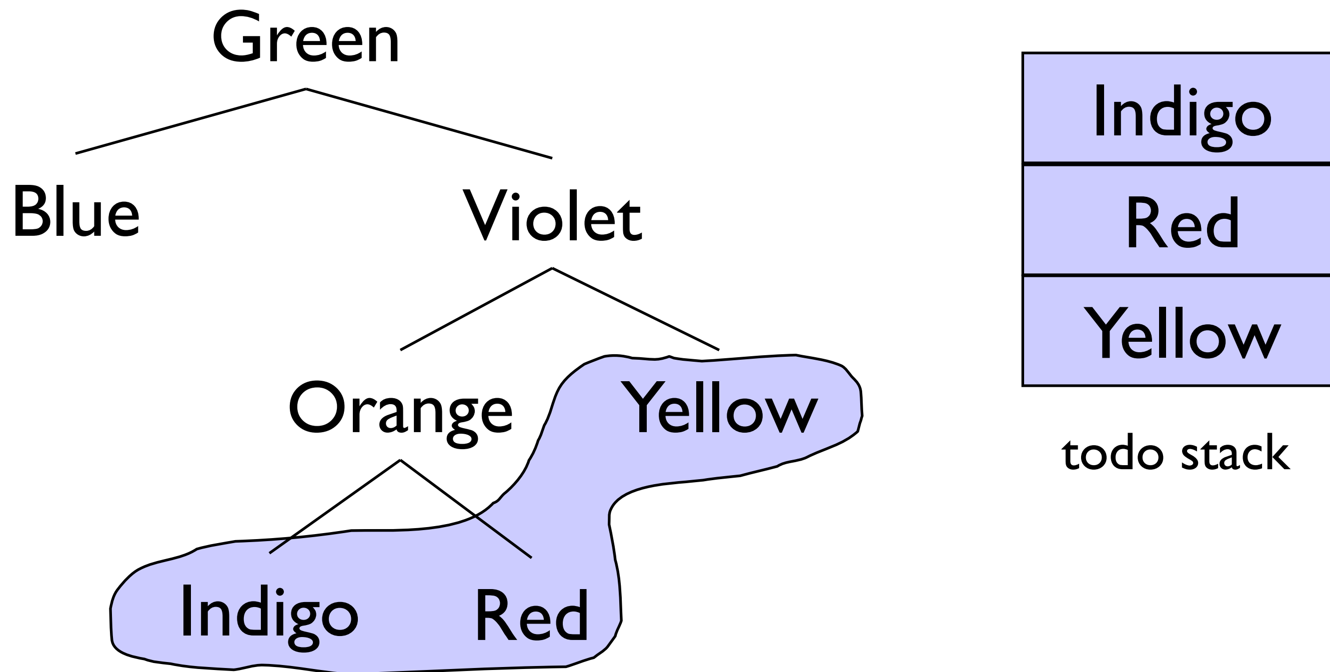
Visit node, then each node in left subtree, then each node in right subtree.



G B V

Pre-Order Iterator

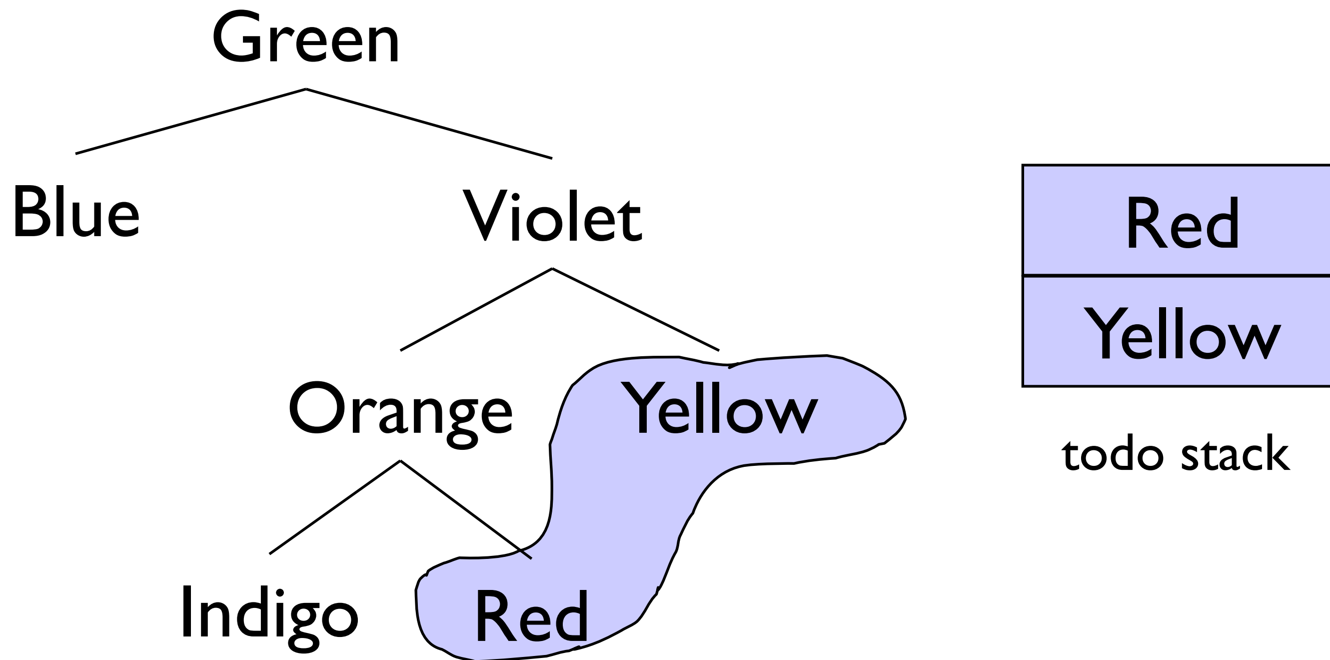
Visit node, then each node in left subtree, then each node in right subtree.



G B V O

Pre-Order Iterator

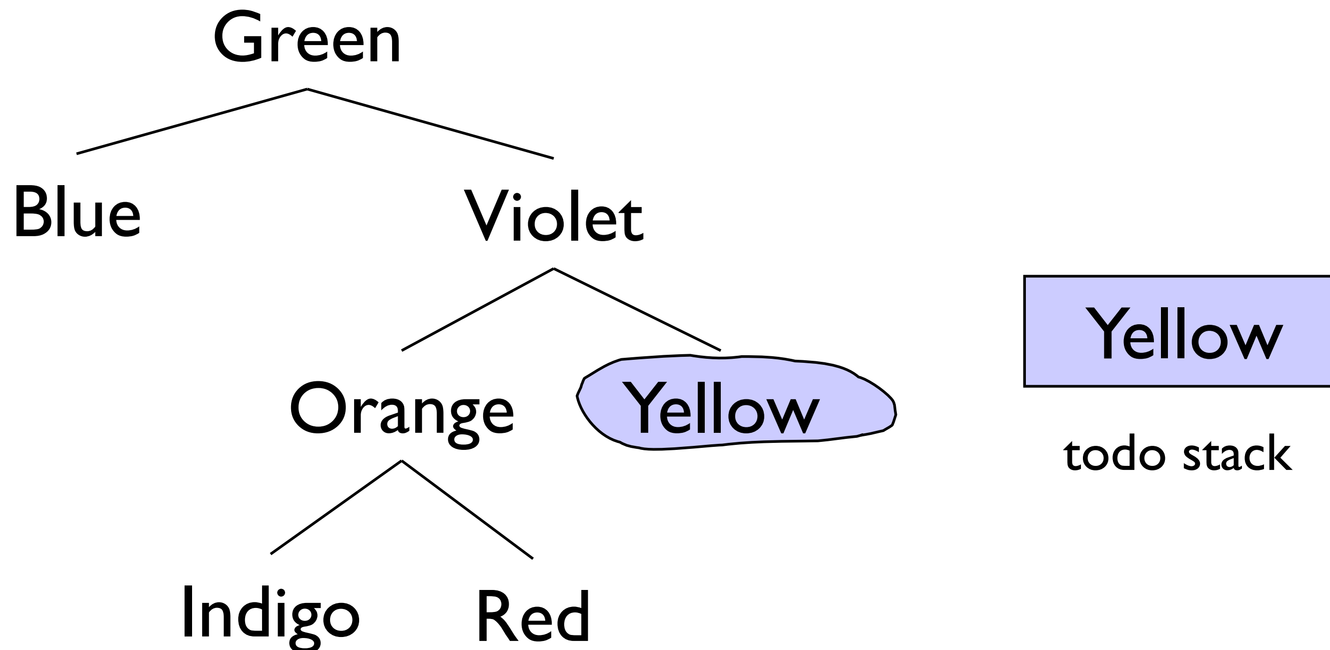
Visit node, then each node in left subtree, then each node in right subtree.



G B V O I

Pre-Order Iterator

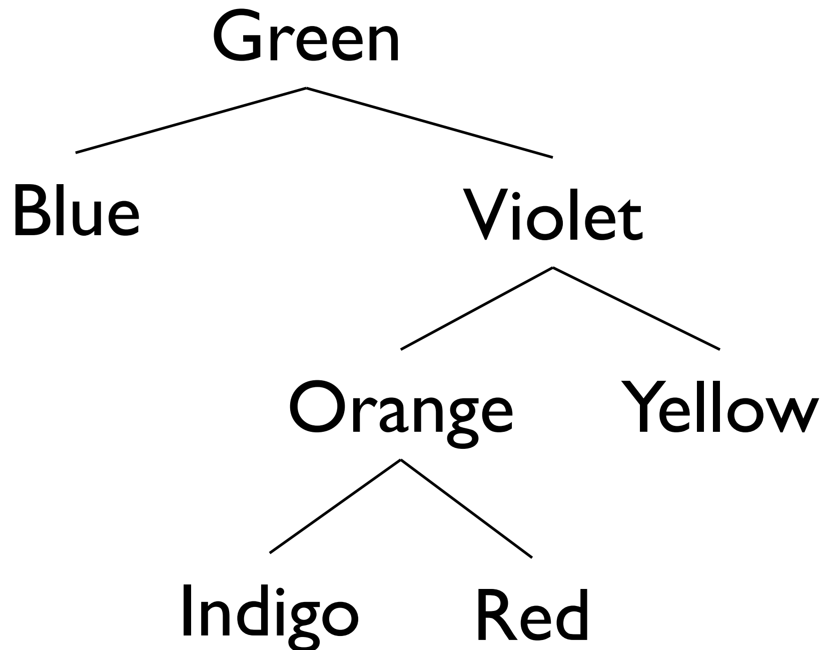
Visit node, then each node in left subtree, then each node in right subtree.



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 now empty; push root on TODO stack
        if ((!root.isEmpty())
                todo.push(root);
}
```

Pre-Order Iterator

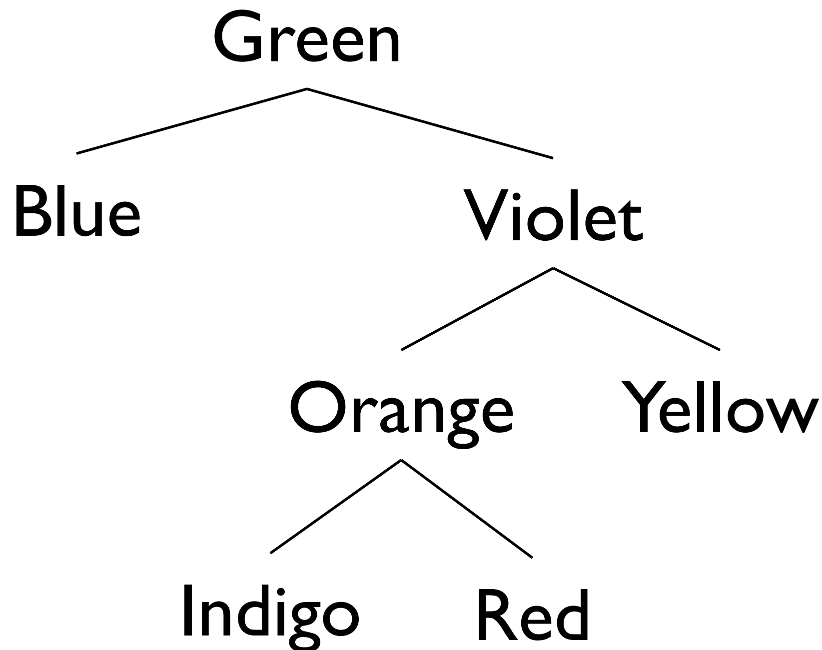
```
public boolean hasNext() {
       return !todo.isEmpty();
}

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 (or LevelOrder) Iterator visits the nodes in the same order as the PreOrder (or LevelOrder) traversal method

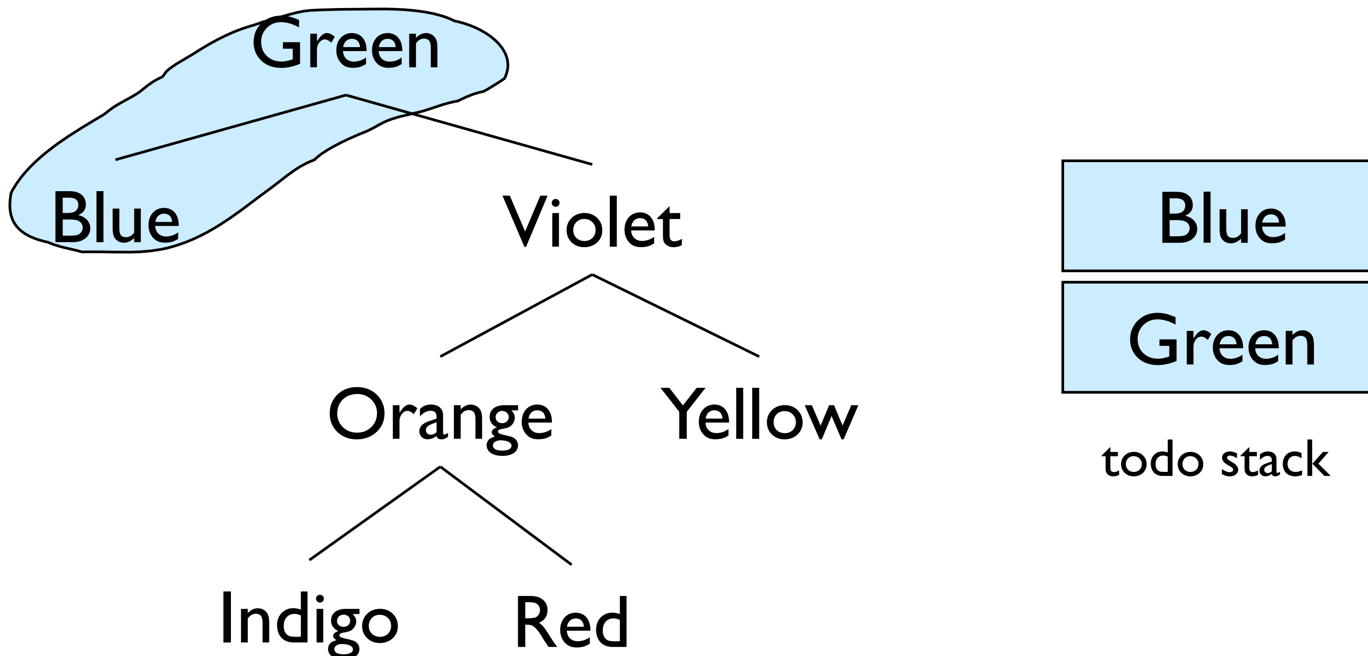
In-Order Iterator

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



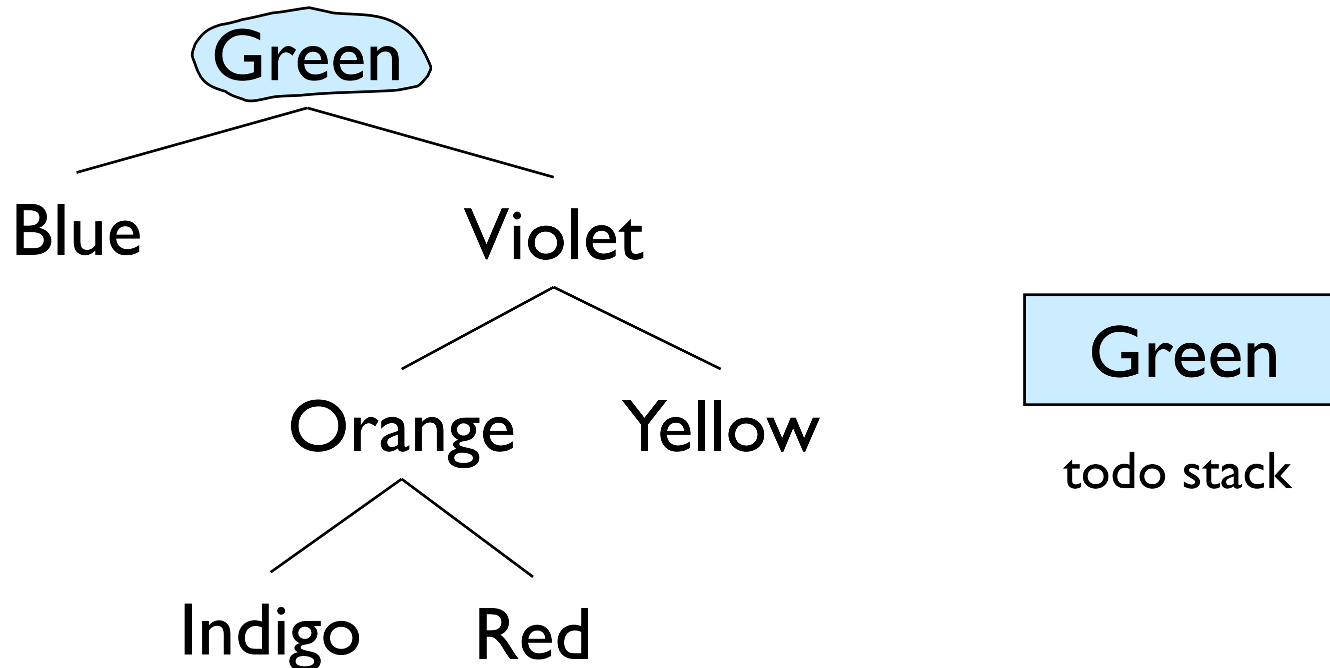
In-Order Iterator

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



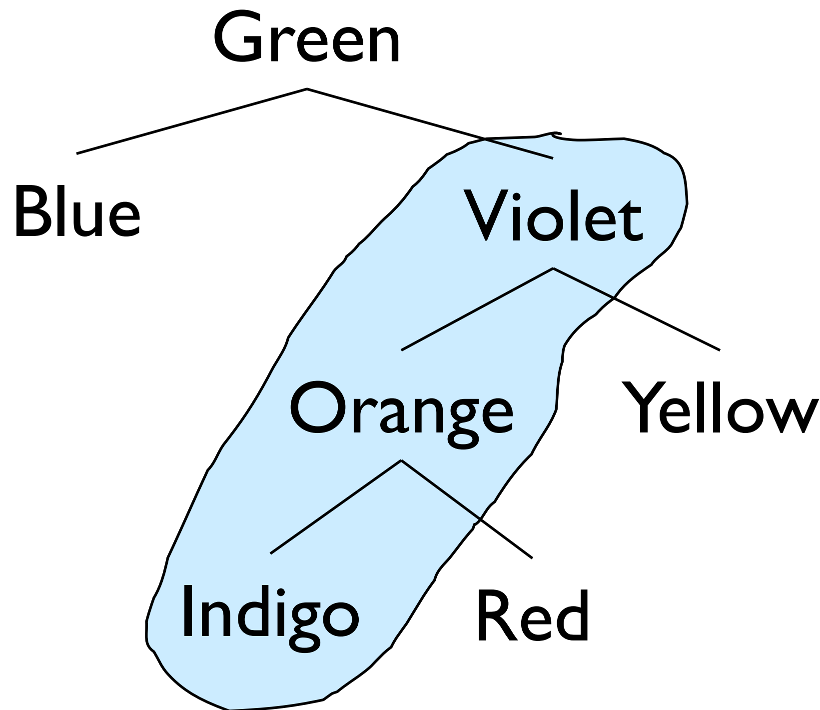
In-Order Iterator

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



In-Order Iterator

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

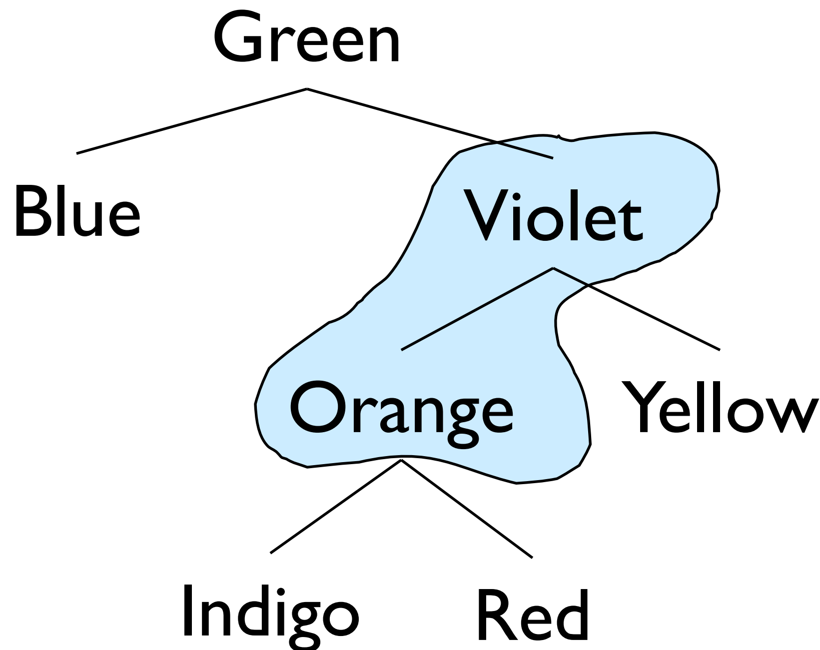


todo stack

B G

In-Order Iterator

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

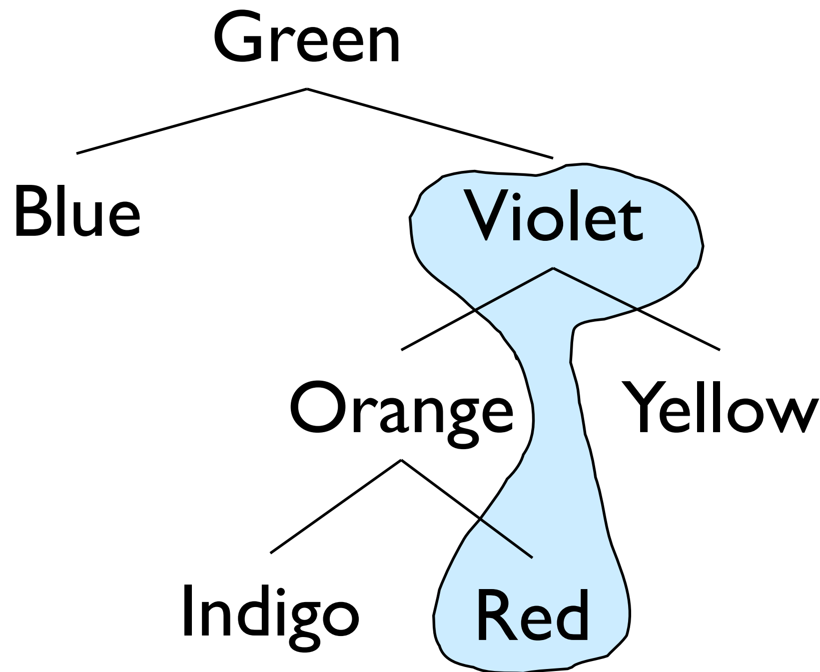


todo stack

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.

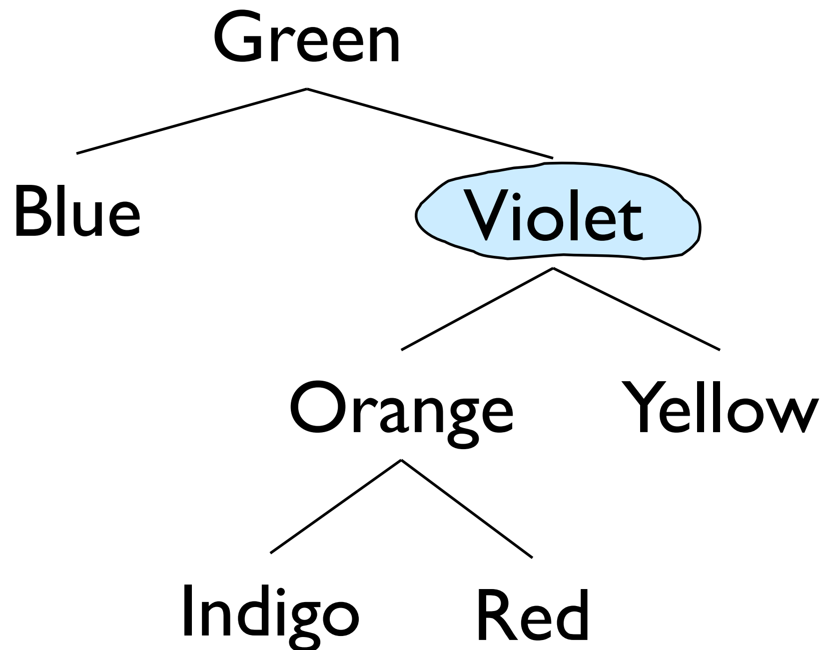


todo stack

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.

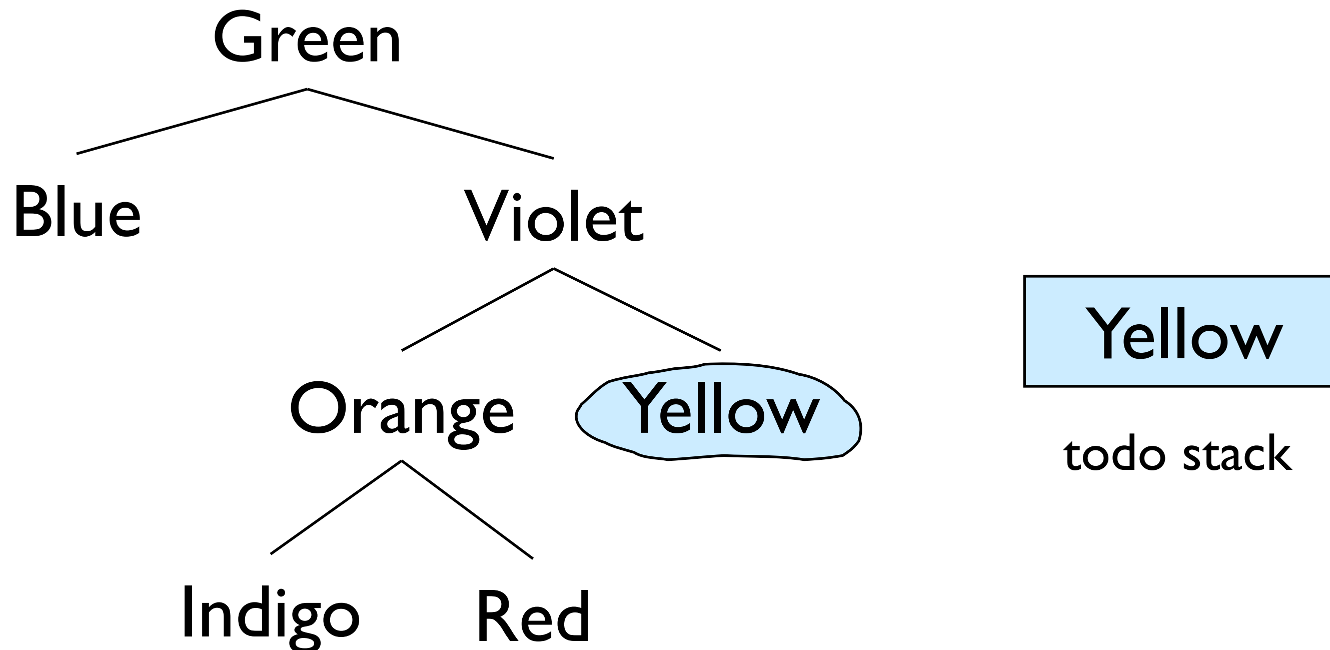


todo stack

B G I O R

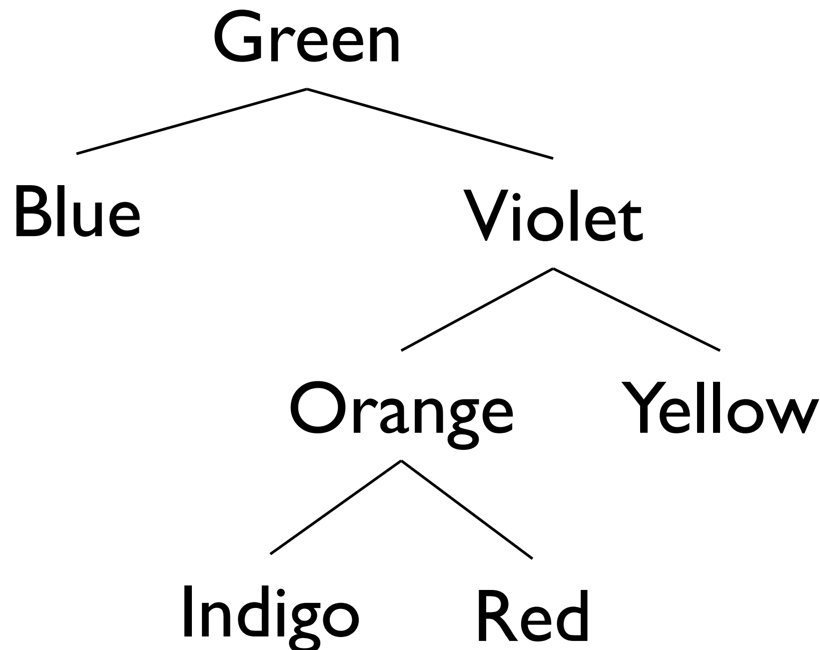
In-Order Iterator

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



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

- Should return elements in same order as processed by in-order traversal method:
  - Traverse left subtree, then node, then right subtree
- Must phrase in terms of `next()` and `hasNext()`
- Basic idea: We again "simulate recursion" with stack

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()`

In-Order Iterator

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

public void reset() {
       todo.clear();
       // stack is empty.  Push on nodes from root along
       // longest “left-only” path
       BinaryTree<E> current = root;
       while (!current.isEmpty()) {
           todo.push(current);
           current = current.left();
       }
}
```

In-Order Iterator

```
public E next() {
       BinaryTree<E> old = todo.pop();
       E result = old.value();
       // we know this node has no unvisited left children;
       // if this node has a right child,
       //   we push right child and longest “left-only” path
       // else
       //   top element of stack is next node to be visited
       if (!old.right().isEmpty()) {
           BinaryTree<E> current = old.right();
           do {
               todo.push(current);
               current = current.left();
           } while (!current.isEmpty());
       }
       return result;
   }
```

Post-Order Iterator

- Outline: left -> right -> node
  1. Push path to leftmost leaf onto stack
  2. On call to `next()`:
     - Pop node from stack
     - Determine whether it was the left or right node of its parent
       - If left child, push parent's right child and the entire path to leftmost leaf parent's right subtree
     - Return node's value
  3. On call to `hasNext()`:
     - return `!stack.isEmpty()`

Post-Order Iterator

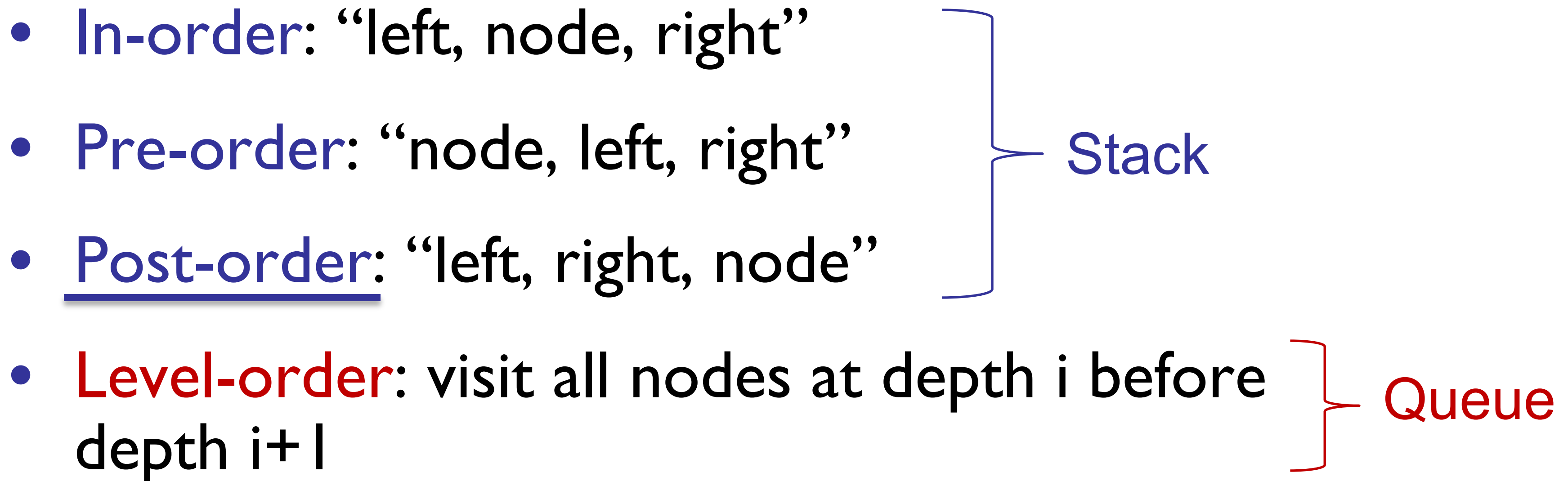
```
public BTPostorderIterator(BinaryTree<E> root) {
        todo = new StackList<BinaryTree<E>>();
        this.root = root;
        reset();
}
public void reset() {
        todo.clear();
        BinaryTree<E> current = root;
        while (!current.isEmpty()) {
              todo.push(current); // current now ‘below’ children
              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:



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)
  - Depth-First: Search until leaves are reached
    - (post-order traversal; but halt when solution found)
- Which is better?
  - Depends on the situation!
    - Does the tree structure represent a concept, e.g., distance or relationship between items?
    - Is the tree "sparse" or "dense"?

Final Thoughts

- Iterators continue to provide a useful service: common structure to enumerate the contents of a data structures
- We have defined four iterators that let us traverse the nodes of a tree in a variety of principled ways
- The best iterator for the task at hand will depend on our problem and our goals. So think critically!