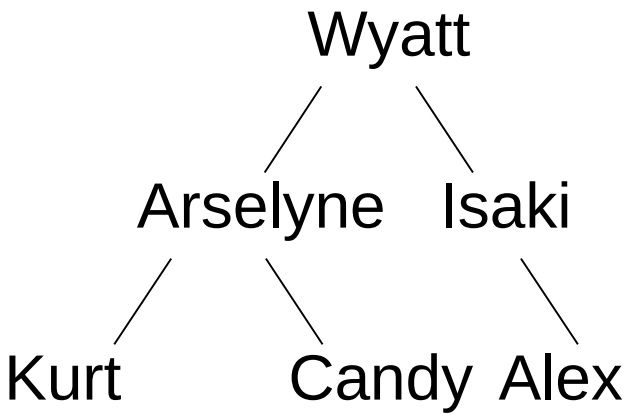
# [TAP:KQNBP] Traversals

- Which of the following is a correct pairing of a traversal algorithm and the resulting order in which nodes are visited?
  - A. Pre-order: Kurt, Candy, Arselyne, Alex, Isaki, Wyatt
  - B. In-order: Kurt, Arselyne, Candy, Wyatt, Isaki, Alex
  - C. Post-order: Wyatt, Arselyne, Isaki, Kurt, Candy, Alex
  - D. Level-order: Wyatt, Arselyne, Kurt, Candy, Isaki, Alex
  - E. Whatever



# Administrative Details

- CS Colloquium?!?!
  - Meets (almost) every Friday at 2:30pm
  - Guest speaker presents their research
  - Next Friday (4/20) we will have an information session instead of a normal speaker
    - Discussion of courses offered next semester
    - Advising about majoring in CS
      - We can sign major declaration sheets there
  - Food!

# Today's Outline

- Binary Tree
  - Iterators
    - Level-order
    - Pre-order
    - In-order
    - Post-order

# Iterators

- Let’s implement iterators for the tree traversal algorithms:
  - PreorderIterator()
  - InorderIterator()
  - PostorderIterator()
  - LevelorderIterator()

# Implementing the Iterators

- Basic idea
  - Should return elements in the same order as corresponding traversal method

# Today's Outline

- Binary Tree
  - Iterators
    - Level-order
    - Pre-order
    - In-order
    - Post-order

# Level-Order Tree Traversal

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

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

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

# Level-Order Iterator

```
public class BTLevelorderIterator<E> extends AbstractIterator<E>{
    Queue<BinaryTree<E>> q;
    BinaryTree<E> root;
  public BTLevelorderIterator(BinaryTree<E> root) {
      q = new QueueList<BinaryTree<E>>();
      this.root = root;
    if(!root.isEmpty())
      q.enqueue(root);

  }
  public void reset() {
      q.clear();
    if(!root.isEmpty())
      q.enqueue(root);

  }
```

# Level-Order Iterator

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

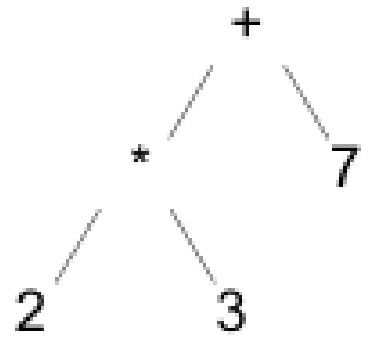
# Today's Outline

- Binary Tree
  - Iterators
    - Level-order
    - Pre-order
    - In-order
    - Post-order

# Tree Traversals

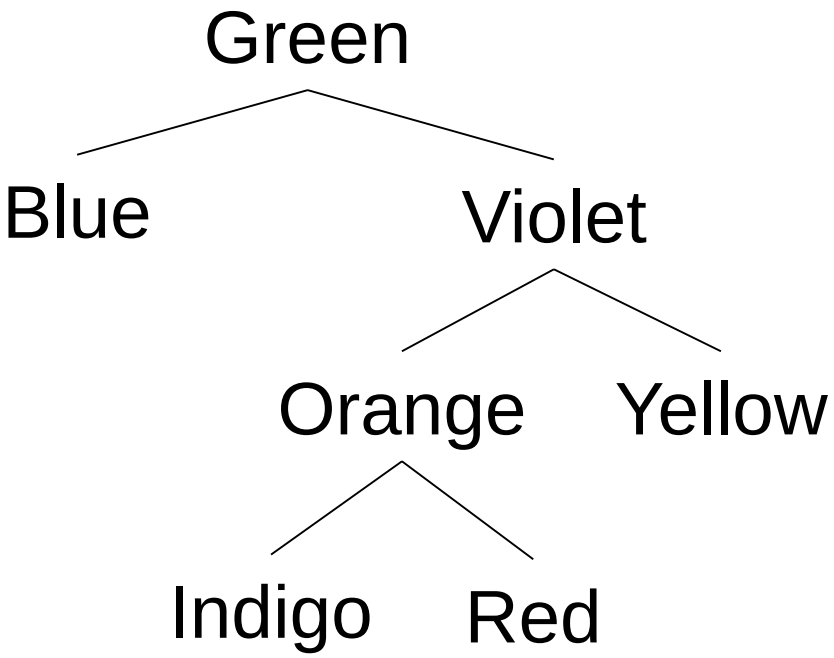
postOrder
inOrder

```
public void preOrder(BinaryTree t) {
    if (t.isEmpty())
        return;

    doSomething(t);
    preOrder(t.left());
    preOrder(t.right());

}
```



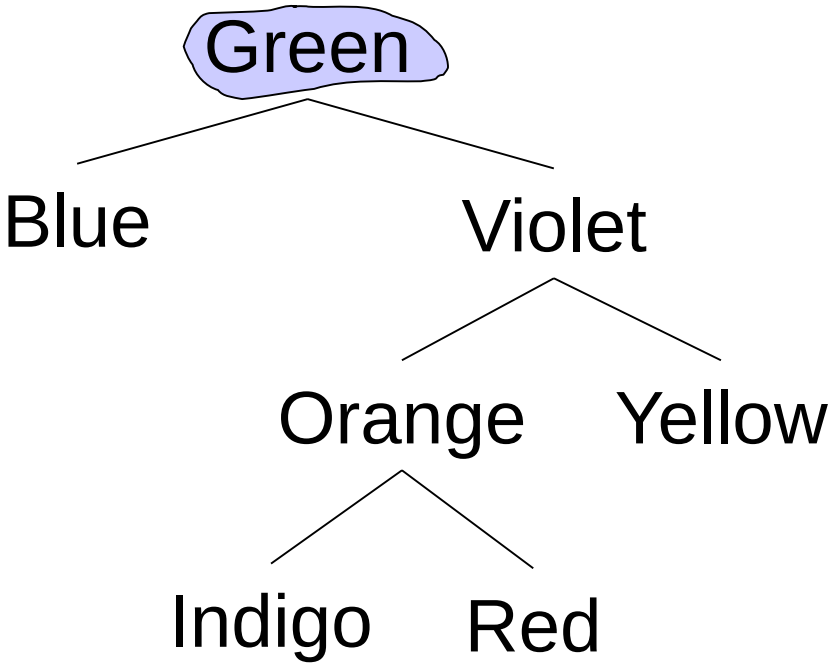
# Pre-Order Iterator

Order: node, left, right



# Pre-Order Iterator

Order: node, left, right

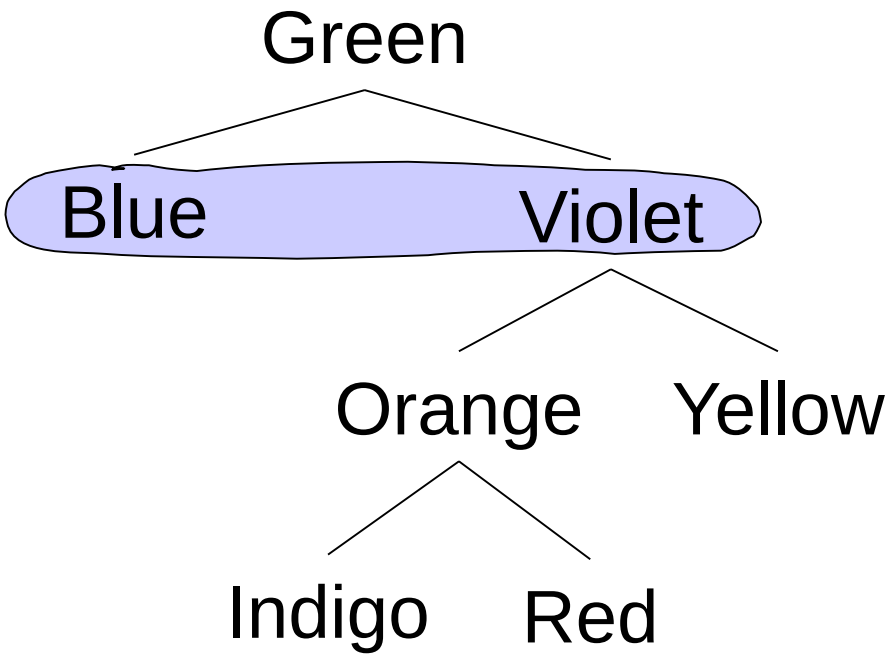


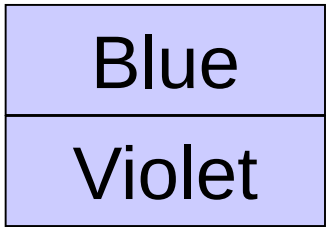

Green

todo stack

# Pre-Order Iterator

Order: node, left, right



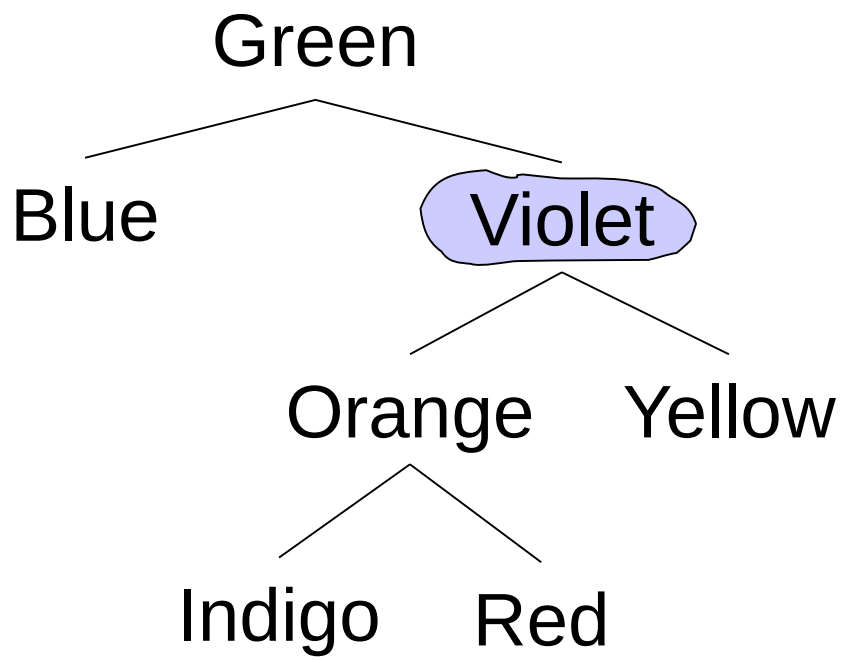



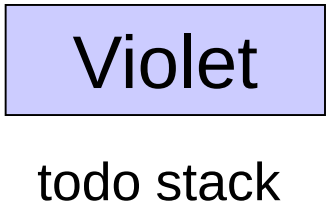

todo stack

G

# Pre-Order Iterator

Order: node, left, right



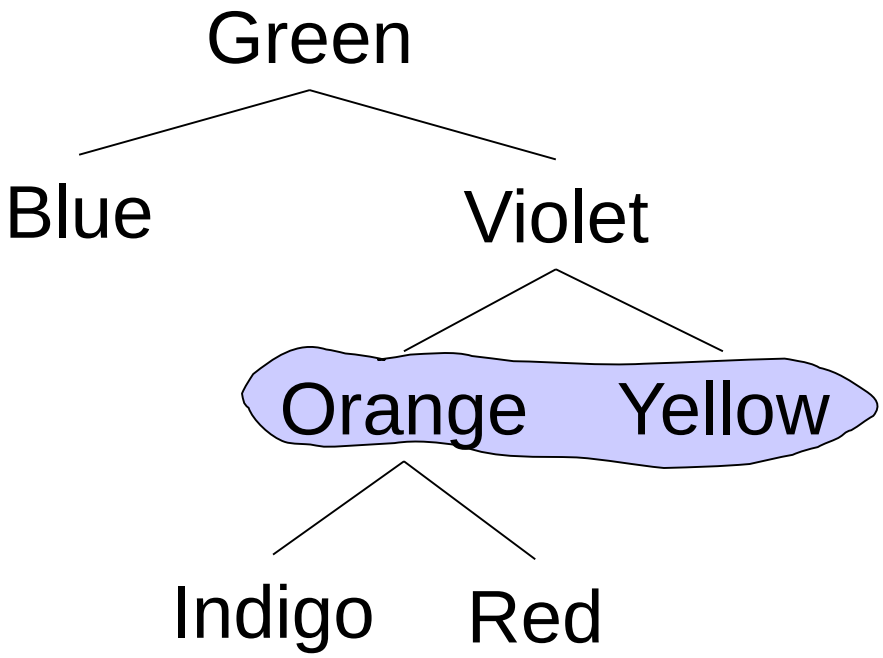



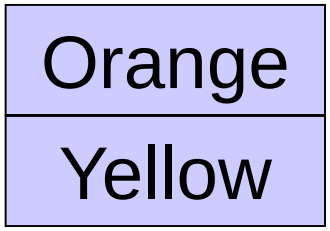

todo stack

G B

# Pre-Order Iterator

Order: node, left, right



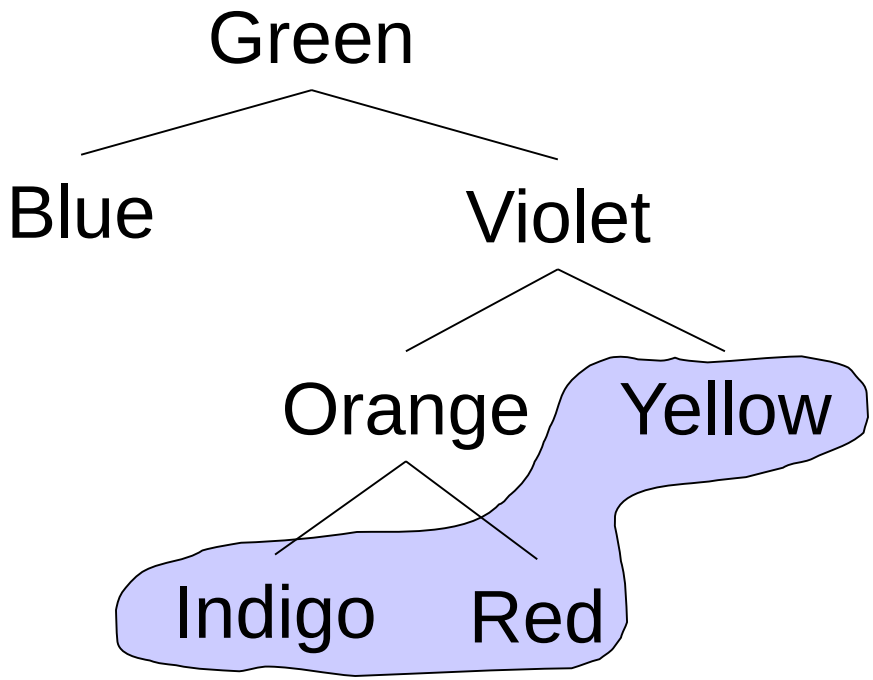



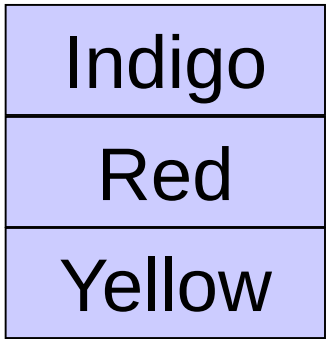

todo stack

G B V

# Pre-Order Iterator

Order: node, left, right



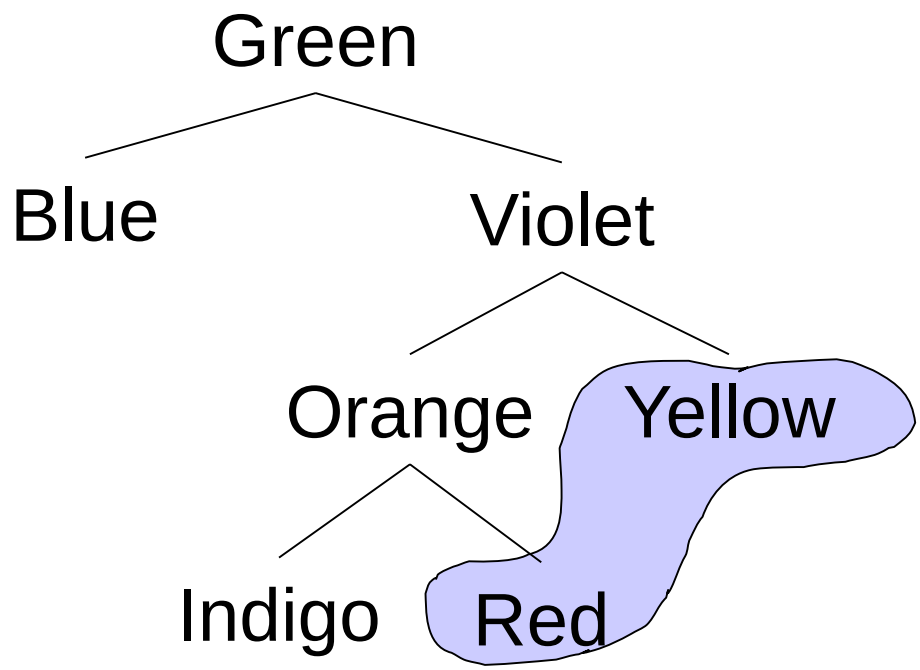



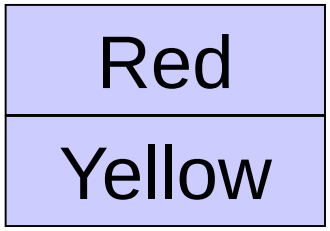

todo stack

G B V O

# Pre-Order Iterator

Order: node, left, right



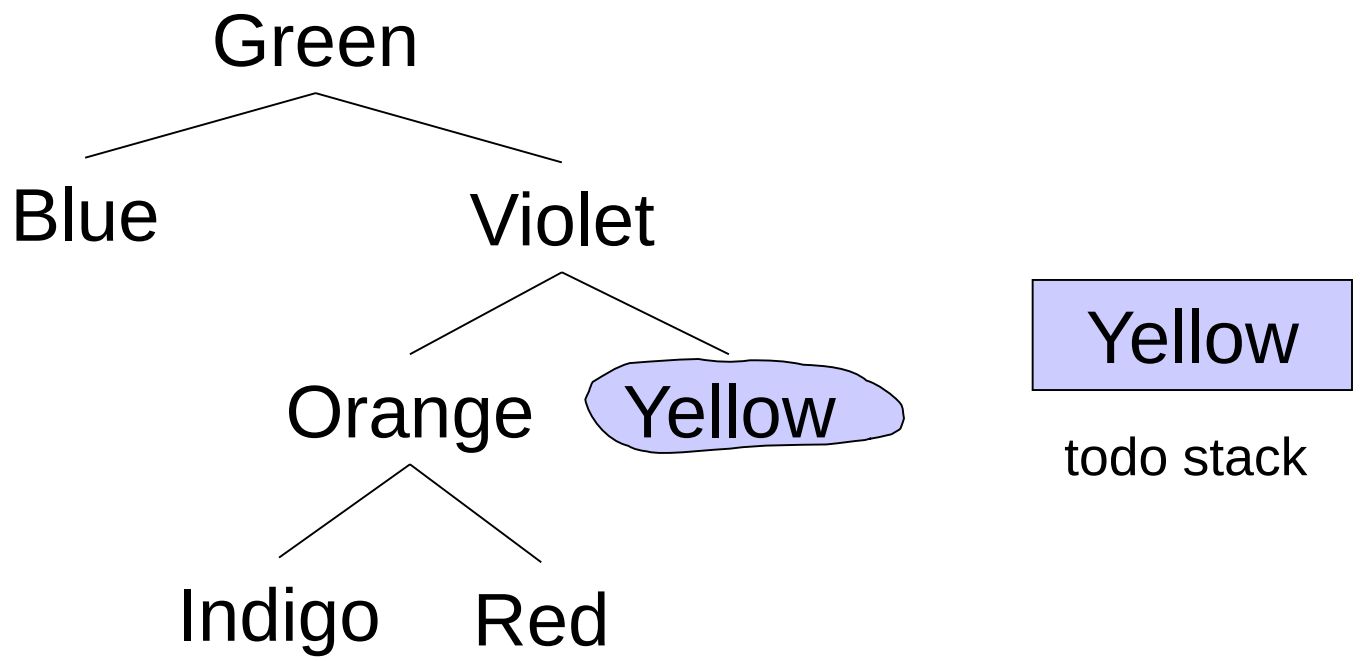




todo stack

G B V O I

# Pre-Order Iterator

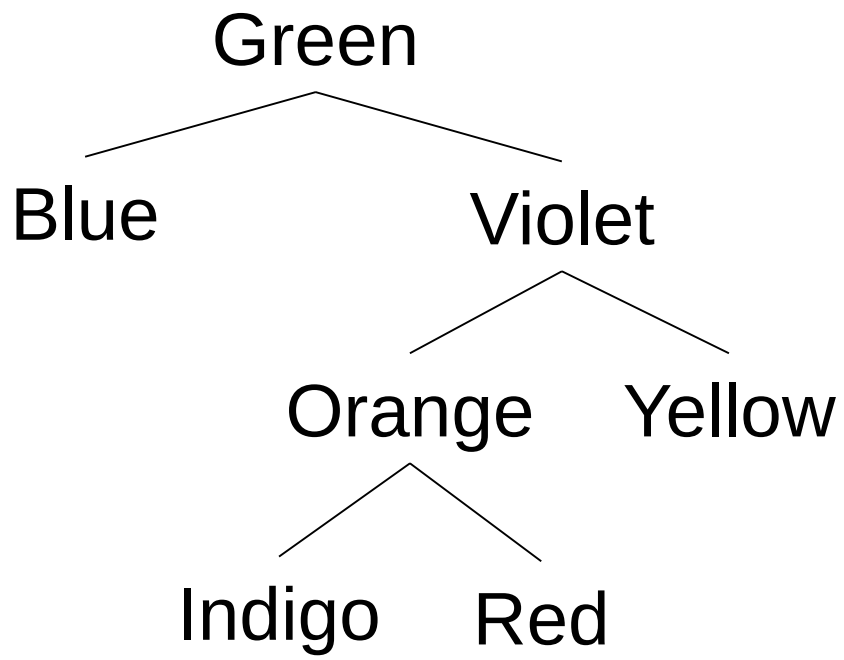
Order: node, left, right




G B V O I R

# Pre-Order Iterator

Order: node, left, right




todo stack

G B V O I R Y

# Pre-Order Iterator

```
public class BTPreorderIterator<E> extends AbstractIterator<E>{
    Stack<BinaryTree<E>> s;
    BinaryTree<E> root;
    public BTPreorderIterator(BinaryTree<E> root) {
        s = new StackList<BinaryTree<E>>();
        this.root = root;
        if(!root.isEmpty())
            s.push(root);
    }
     public void reset() {
            s.clear();
            if(!root.isEmpty())
                s.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;
  }
```

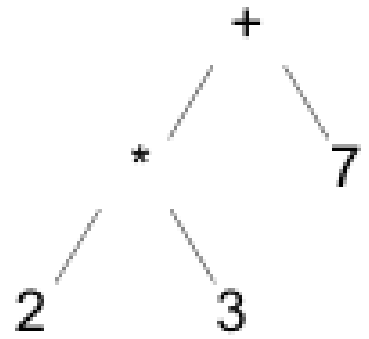
# Today's Outline

- Binary Tree
  - Iterators
    - Level-order
    - Pre-order
    - In-order
    - Post-order

# Tree Traversals

postOrder
inOrder

```
public void preOrder(BinaryTree t) {
    if (t.isEmpty())
        return;

    doSomething(t);
    preOrder(t.left());
    preOrder(t.right());

}
```

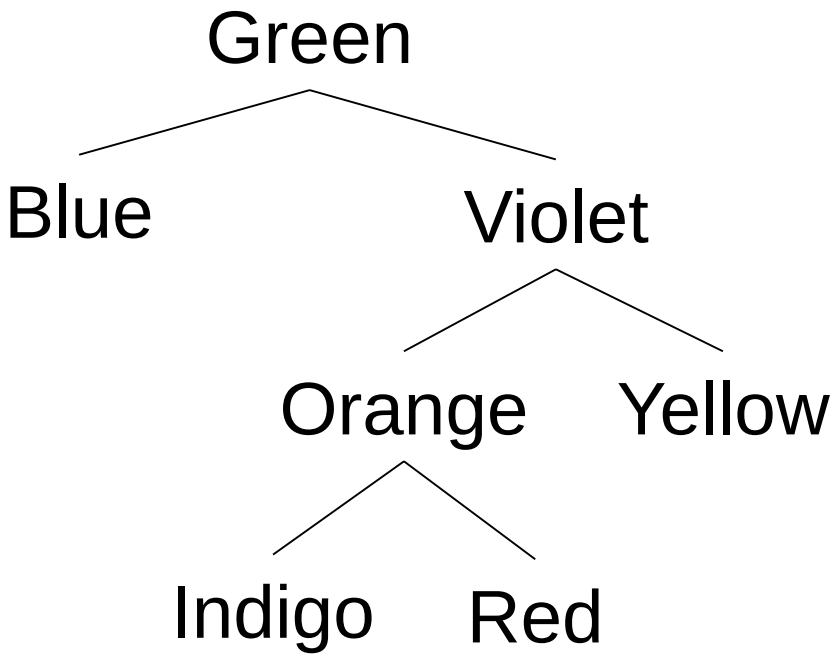


# In-Order Iterator

- Order: 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()

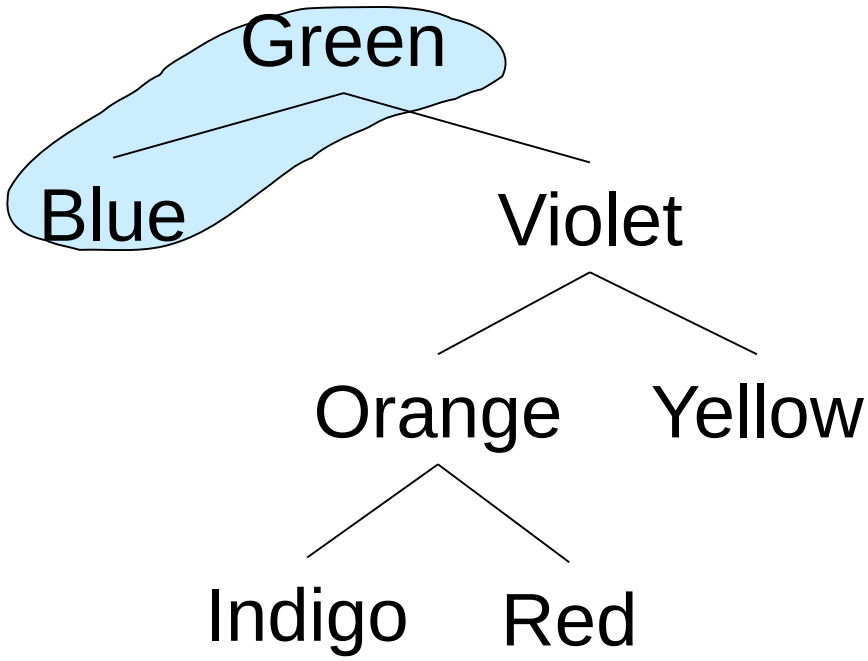
# In-Order Iterator

Order: left, node, right



# In-Order Iterator

Order: left, node, right

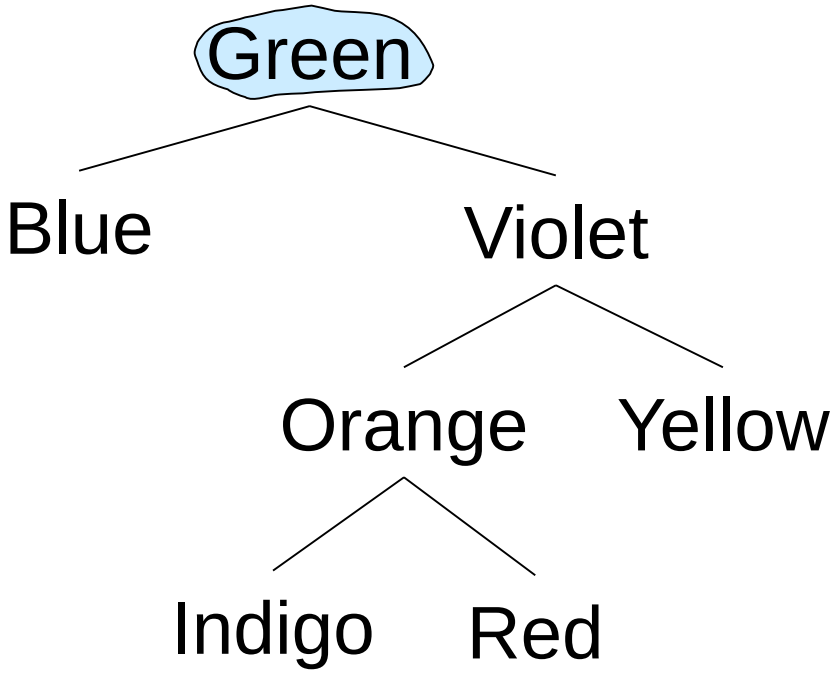




todo stack

# In-Order Iterator

Order: left, node, right



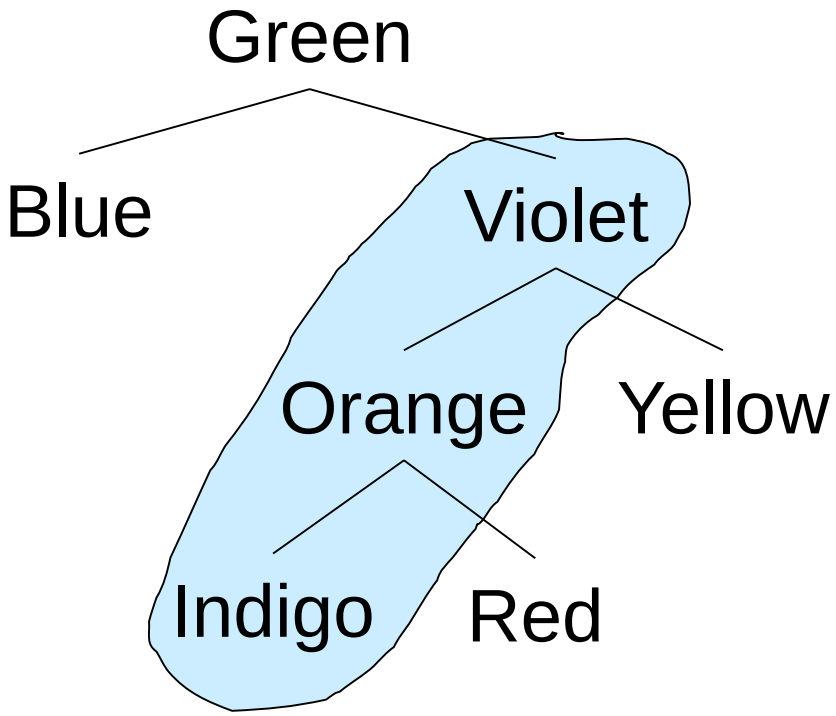

Green

todo stack

B

# In-Order Iterator

Order: left, node, right



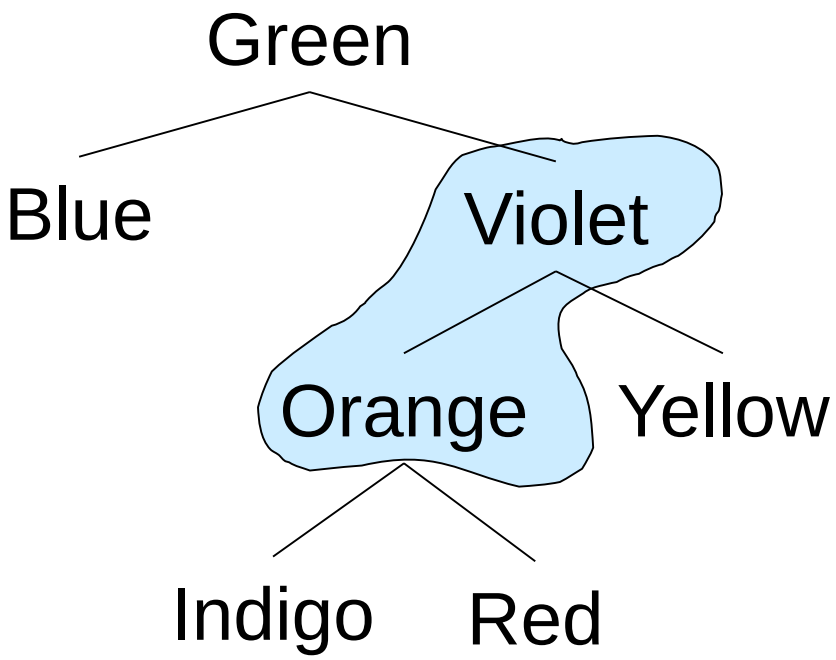


todo stack

B G

# In-Order Iterator

Order: left, node, right



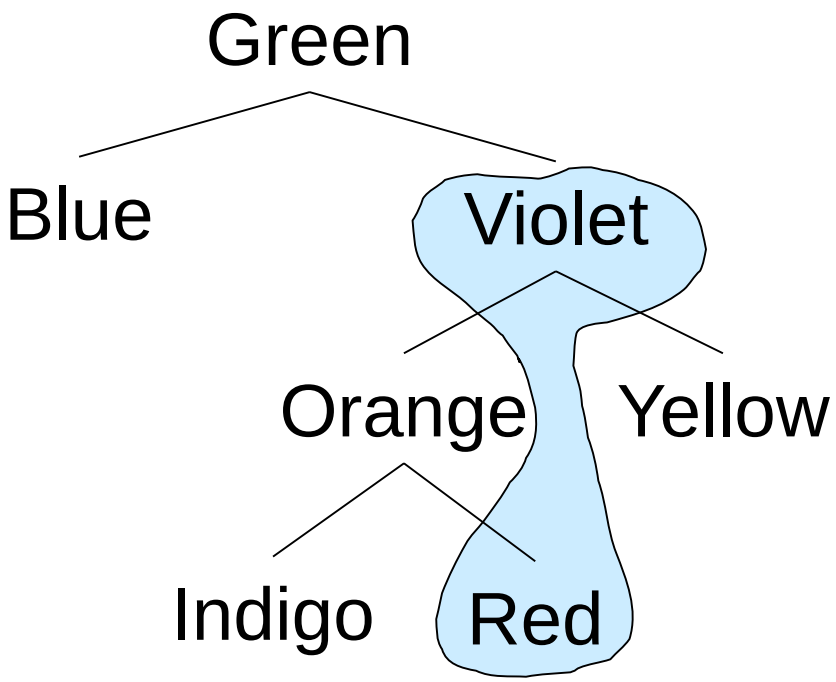


todo stack

B G I

# In-Order Iterator

Order: left, node, right



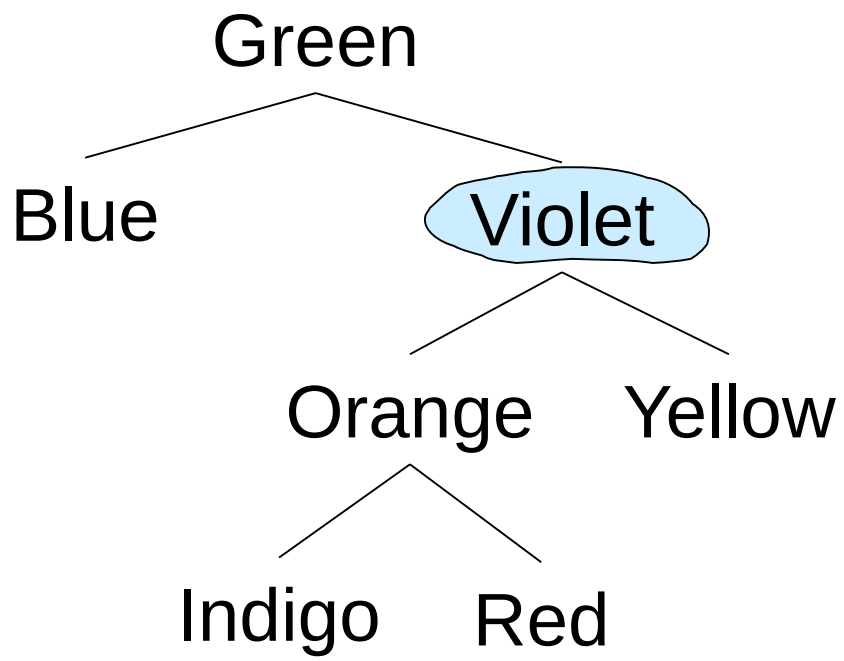


todo stack

B G I O

# In-Order Iterator

Order: left, node, right



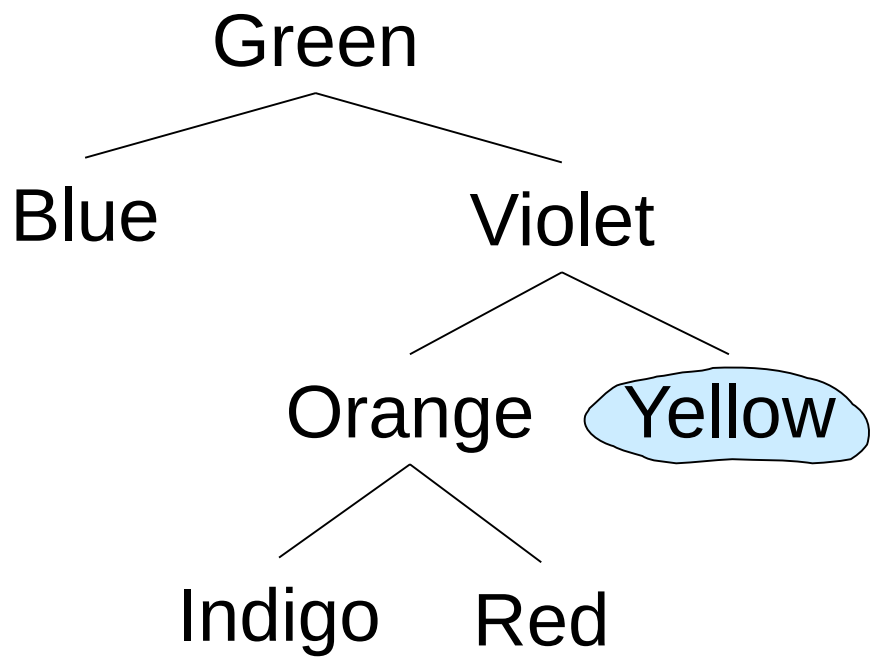

Violet

todo stack

B G I O R

# In-Order Iterator

Order: left, node, right



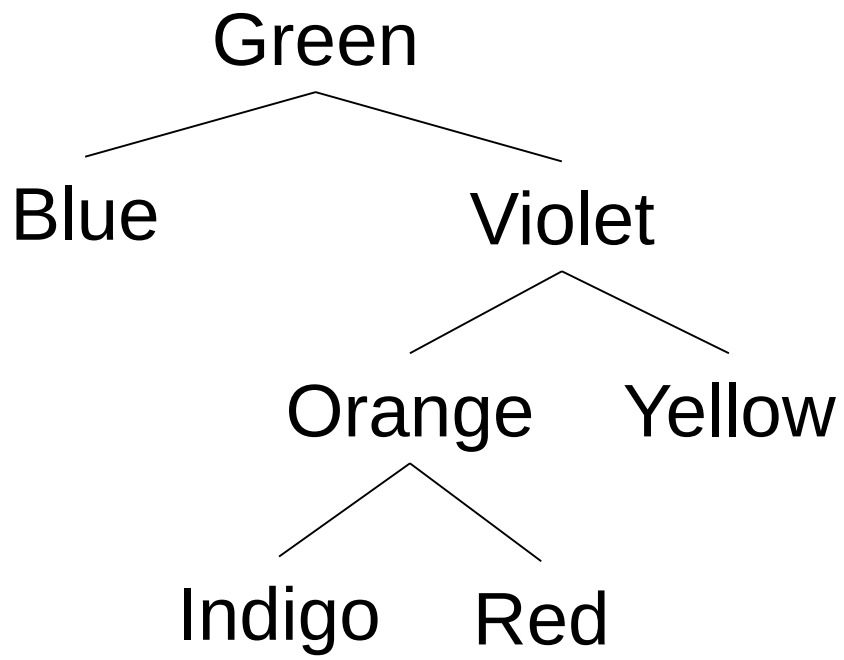

Yellow

todo stack

B G I O R V

# In-Order Iterator

Order: left, node, right




todo stack

B G I O R V Y

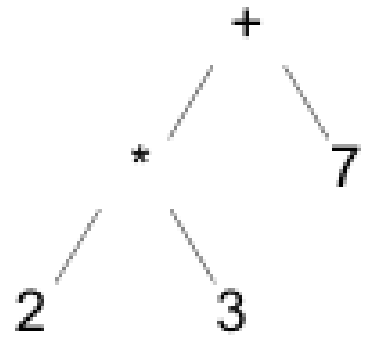
# Today's Outline

- Binary Tree
  - Iterators
    - Level-order
    - Pre-order
    - In-order
    - Post-order

# Tree Traversals

postOrder
inOrder

```
public void preOrder(BinaryTree t) {
    if (t.isEmpty())
        return;

    doSomething(t);
    preOrder(t.left());
    preOrder(t.right());

}
```



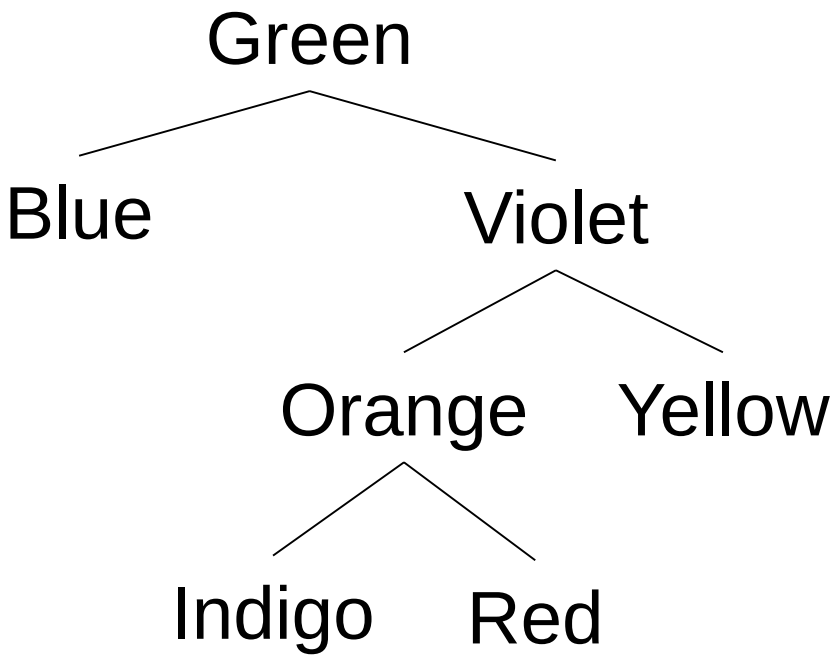
# Post-Order Iterator

- Left as an exercise!

# Today's Outline

- Binary Tree
  - Iterators
    - Level-order
    - Pre-order
    - In-order
    - Post-order

# 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
- Where is parent of node j?
  - Parent of node j is at (j-1)/2