# CSCI 136
# Data Structures &
# Advanced Programming

Jeannie Albrecht
Lecture 24
April 18, 2014

## Administrative Details

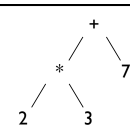
- Lab 8 – due Monday
  - Any questions?

## Last Time

- Wrapped up decision trees
- Discussed tree traversal
  - Looked closely at code for pre-order

```
public E next() {
        BinaryTree top = todo.pop();
        E result = top.value();
        if(!top.right().isEmpty()) {
            todo.push(top.right());
        }
        if(!top.left().isEmpty()) {
            todo.push(top.left());
        }
        return result;
}
```

## Pre-order




- Pre-order: +*237
  - Each node is visited before any children. Visit node, then each node in left subtree, then each node in right subtree.

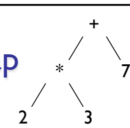
```
public void preOrder(BT root) {
   if (root.isEmpty()) return;
   process(root.value());
   preOrder(root.left());
   preOrder(root.right());
}
```

- In real code, we need to keep track of our own stack!

## Today's Outline

- Finish discussing tree iterators
  - In-order, level-order, post-order
- Wrap up chapter 12 (Binary Trees) and start chapter 13 (Priority Queues)
- Briefly discuss Huffman codes

## Tree Traversal Recap



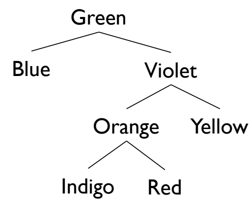

- Pre-order: +*237
  - Each node is visited before any children. Visit node, then each node in left subtree, then each node in right subtree.
- In-order: 2*3+7
  - Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
- Post-order: 23*7+
  - Each node is visited after its children are visited. Visit all nodes in left subtree, then all nodes in right subtree, then node itself.
- Level-order: +*723
  - All nodes of level i are visited before nodes of level i+1.

# InOrder Iterator

- Outline: left - node - right
  1. Push left children (as far as possible) onto todo 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()

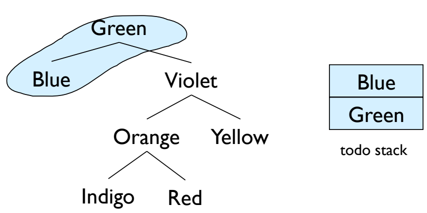
# InOrder Iterator

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



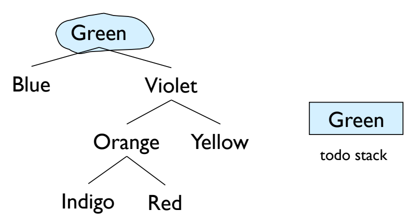

# InOrder Iterator

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




# InOrder Iterator

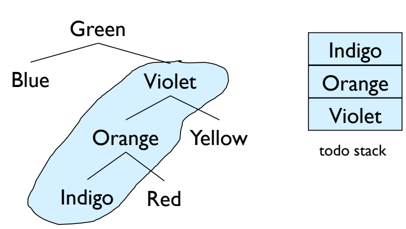
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B

# InOrder Iterator

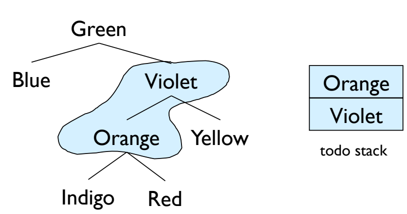
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B G

# InOrder Iterator

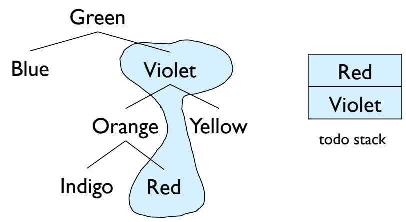
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B G I

## InOrder Iterator

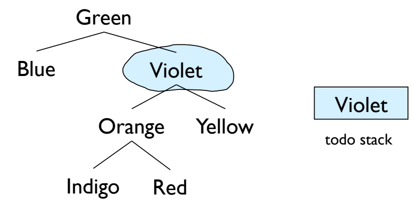
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B G I O

## InOrder Iterator

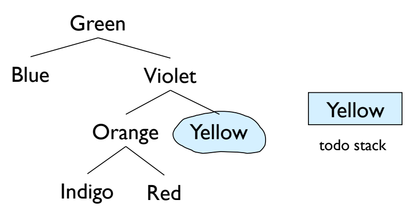
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B G I O R

## InOrder Iterator

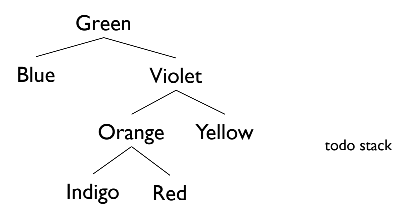
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B G I O R V

## InOrder Iterator

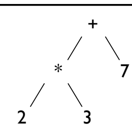
Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.




B G I O R V Y

Code?

## Level-order

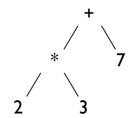



- Let's take a closer look at LevelOrder…
- Level-order: +*723
  - All nodes of level i are visited before nodes of level i+1.

## LevelOrder Iterator

- Do we want to use a stack??
  - No! Use a queue instead.
- Outline:
  1. Enqueue root
  2. On call to next():
     - Dequeue node
     - Enqueue left and right child
     - Return node
  3. On hasNext()
     - Return !queue.isEmpty()

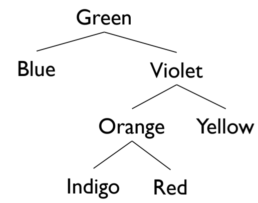
## Level-order



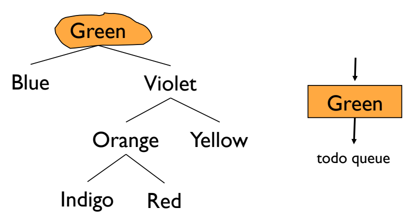

- Level-order: +*723
  - All nodes of level i are visited before nodes of level i+1.

```
public void levelOrder(BT root) {
   Queue q = new QueueList();
   q.enqueue(root);
   while(!q.isEmpty()) {
      BT tree = (BT) q.dequeue();
      if (!tree.isEmpty()) {
     process(tree.value());
     q.enqueue(tree.left());
     q.enqueue(tree.right());
      }
   }
}
```

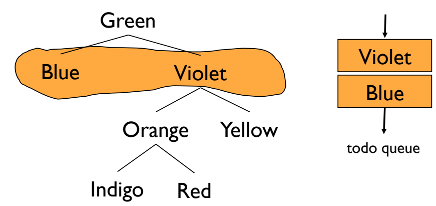
## LevelOrder Iterator




## LevelOrder Iterator

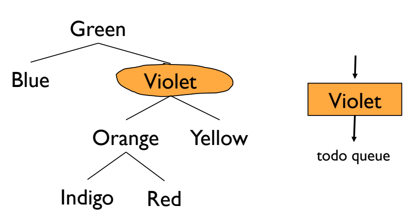



## LevelOrder Iterator



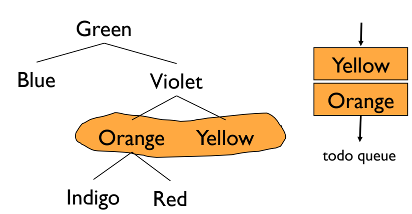

G

## LevelOrder Iterator



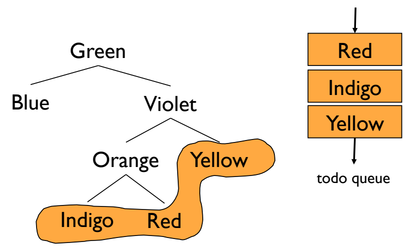

G B

## LevelOrder Iterator



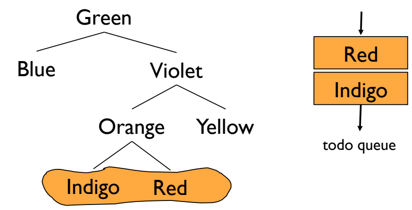

G B V

## LevelOrder Iterator



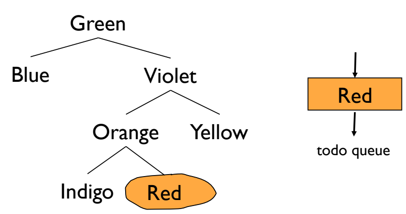

G B V O

## LevelOrder Iterator



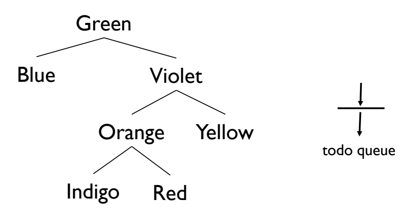

G B V O Y

## LevelOrder Iterator




G B V O Y I

## LevelOrder Iterator




G B V O Y I R

## PostOrder Iterator

- Left as an exercise…

## Moving on…

- Note:
  - Code for PostOrder is similar to PreOrder with minor differences
  - Please see Bailey for details (preferably before your midterm!)

## An Aside: Tree Search Strategies

- Two main approaches
  - Breadth-first search (BFS)
    - Search across tree before searching down to another level
    - Level-order traversal
  - Depth-first search (DFS)
    - Search down tree (to leaf) before search across tree
    - Pre-order traversal
  - DFS is more efficient if solution is "far away" from root (i.e., many edges between root and solution)

## Next up: Huffman Codes

- Normally, 1 character = 8 bits (1 byte)
  - Allows for $2^8$ = 256 different characters
- 'A' = 01000001, 'B' = 01000010
- Space to store "AN ANTARCTIC PENGUIN"
  - 20 characters -> 20*8 bits = 160 bits
- Is there a better way?
  - Only 11 symbols are used (ANTRCIPEGU_)
  - Only need 4 bits per symbol (since $2^4$>11)!
    - 20*4 = 80 bits instead of 160!
  - Can we still do better??

## Huffman Codes

- General idea
  - Use less bits for most common letters
  - AN ANTARCTIC PENGUIN
  - Compute letter frequencies

```
A: 3        N: 4
T: 2        R: 1
C: 2        I: 2
P: 1        E: 1
G: 1        U: 1
_: 2
```

  - Build tree by recursively creating trees of smallest weighted components

## How Many Bits?

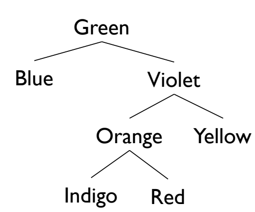
```
A:  100 x 3     N:  101 x 4
T:  001 x 2     R: 0000 x 1
C:  010 x 2     I:  011 x 2
P: 0001 x 1     E: 1100 x 1
G: 1101 x 1     U: 1110 x 1
_: 1111 x 2
```

- So total number of bits = 67
- Note: There may be multiple possible Huffman trees
  - All trees should use same total number of bits

## Other Compression Techniques

- Examine larger pieces of data for patterns
  - AAAAA BBBBBBBBB CC AAAAAAA
  - (5,A) (9,B) (2,C) (7,A)
- Lempel-Ziv-Welch (LZW)
  - Huffman code for longer substrings
    - ABCABCABC
      - 0-255: ASCII characters
      - 256: AB
      - 257: ABC

## Alternative Tree Representations




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

## Using Arrays to Store Trees

- Encode structure of tree in array indexes
- 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
  - Height of n requires $2^{n+1}$-1 array slots even if only O(n) elements