CSCI 136
Data Structures &
Advanced Programming

Jeannie Albrecht
Lecture 23
April 16, 2014

Administrative Details

- Lab 8 is today
  - Can work with a partner again
  - We'll briefly go over design in lab
  - Faculty meeting at 4 today
- Looking ahead:
  - Lab 9 – Darwin, due 5/7 (2 weeks)
  - Wed 4/30: Midterm 2 (during lab again)
- One (or two) more labs after that (last one is probably optional)
- Office hours on Thursday: 2ish – 3:30ish

Last Time

- Looked at ways to prove tree properties using induction
- Started discussing decision trees

BT Questions/Proofs

- (A) Prove that number of nodes at level n <= 2^n.
- (B) Prove that number of nodes in tree of height n is <= $2^{(n+1)} - 1$.
  - Base case n=0: Tree of height = 0 only contains root. Thus only 1 node when height=0.
    - $2^{(0+1)} - 1 = 1$. Base case holds.
  - IH: Assume true for all k<n.
    - That is, the number of nodes in tree of height k is <= $2^{(k+1)} - 1$
  - IS: Suppose k=n – 1. (We will show it holds for k=n.)
    - By our IH, we know that the number of nodes is <= $2^{(n)} - 1$.
    - By (A), we also know that the number of nodes at level n <= 2^n.
    - So at height n, the number of nodes in tree is at most (<=) $2^{(n)} + 2^{(n)} - 1 = 2 \times 2^{(n)} - 1 = 2^{(n+1)} - 1$.

Today's Outline

- Continue discussing decision trees
- Learn about tree traversal
  - In-order, pre-order, post-order, level-order
  - Learn how to implement tree iterators

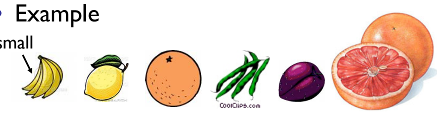
Recap: Representing Knowledge

- Trees can be used to represent knowledge
  - Example: InfiniteQuestions game
- We often call these trees **decision trees**
  - Leaf: object
  - Internal node: question to distinguish objects
- Move down decision tree until we reach a leaf node
- Check to see if the leaf is correct
  - If not, add another question, make new and old objects children

Building Decision Trees

- Gather/obtain data
- Run correlation analysis
  - Make greedy choices: Find good questions that divide data into halves (or as close as possible)
- Construct tree with shortest height
- Example

small



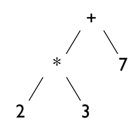
large and yellow

Moving on…

Tree Traversals

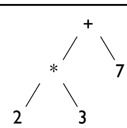
- In linear structures, there are only a few logical (useful) ways to traverse the data structure
  - Start at one end and visit each element
  - Start at the other end and visit each element
- How do we traverse binary trees?
  - (At least) four potential mechanisms

Tree Traversals



- In-order: 2*3+7
- Pre-order: +*237
- Post-order: 23*7+ (look familiar?)
- Level-order: +*723

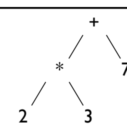
Tree Traversals



- 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

(Look at "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 recursive!)
  - All nodes of level i are visited before nodes of level i+1. (visit nodes left to right on each level)
    - +*723

(Look at "pseudocode")

Iterators

- We need to provide iterators that implement the different tree traversal algorithms
- Methods provided by BT class:
  - preorderIterator()
  - inorderIterator()
  - postorderIterator()
  - levelorderIterator()

PreOrder Iterator

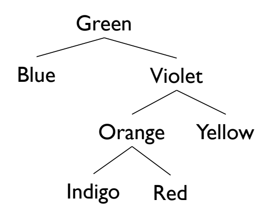
- Basic idea
  - Should return elements in same order as processed by pre-order traversal method
  - Recursive method won't work for iteration, must phrase in terms of next() and hasNext()
  - But we "simulate recursion" with stack
    - Maintain list of subtrees left to traverse
    - Todo stack: Roots of trees left to process
  - Stack is *frontier*: nodes left to traverse

PreOrder Iterator

- Outline: node - left - right
  1. 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()

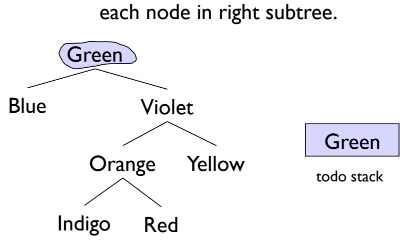
PreOrder Iterator

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



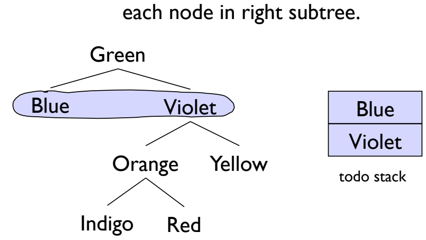
PreOrder Iterator

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



PreOrder Iterator

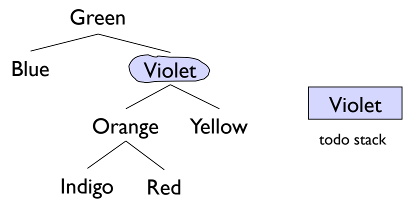
Visit node, then each node in left subtree, then each node in right subtree.



G

PreOrder Iterator

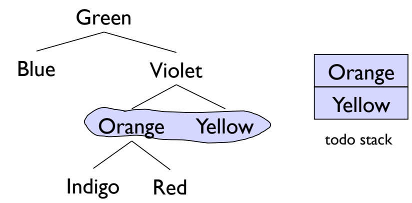
Visit node, then each node in left subtree, then each node in right subtree.



G B

PreOrder Iterator

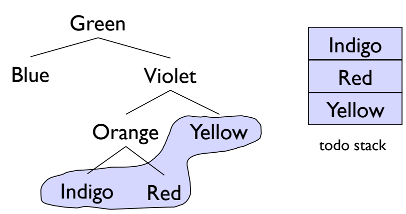
Visit node, then each node in left subtree, then each node in right subtree.



G B V

PreOrder Iterator

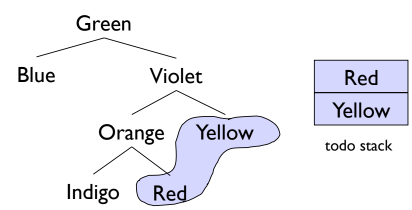
Visit node, then each node in left subtree, then each node in right subtree.



G B V O

PreOrder Iterator

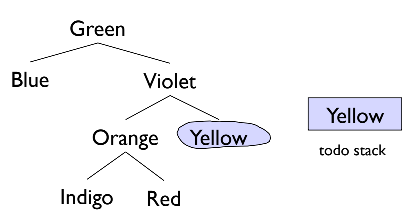
Visit node, then each node in left subtree, then each node in right subtree.



G B V O I

PreOrder Iterator

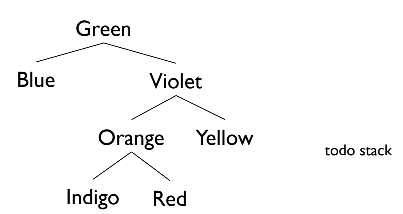
Visit node, then each node in left subtree, then each node in right subtree.



G B V O I R

PreOrder Iterator

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



G B V O I R Y

Now let's look at the code…