Announcements

1. HW6 posted. Have you started
   A. No, not at all
   B. I’ve read it
   C. I’ve started coding
   D. I’ve made significant progress
   E. I’m done!

2. HW7 coming later today
Today’s topics

- Tree traversals
- BST and their implementations
Pre-order traversal

```java
preorder(node) {
    if (node != null)
    {
        visit this node
        preorder(node.left)
        preorder(node.right)
    }
}
```

A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more
Post-order traversal

```java
postorder(node) {
    if (node != null){
        postorder(node.left)
        postorder(node.right)
        visit this node
    }
}
```

A. D B E A F C G  
B. A B D E C F G  
C. A B C D E F G  
D. D E B F G C A  
E. Other/none/more
In-order traversal

\[ \text{inorder}(\text{node}) \{ \]
if (node != null) {
    inorder(node.left)
    visit this node
    inorder(node.right)
}
\}

A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more
Level-order traversal

AKA Breadth-first search
Level-order traversal

- Also commonly called **Breadth-First Search** or **BFS**
- As opposed to something like pre-order or post-order, which are **Depth-First Search (DFS)** algorithms
Level-order challenges

- How do we know where to go next?
- 1, 2, 3—but there is no edge from 2 to 3!
- While we’re still at 1, we must “remember” to come back to 3 after we’re done with 2
  - At 2, remember to come back to 4 and 5 after 3
  - At 8, remember to come back to 16 and 17 after 15
- Etc …
How to “remember”

- When we visit a node, add references (pointers) to its children to a queue
- To know which node to visit next, remove next element from the queue
Tracing the queue in BFS

Which shows the state of the queue at the end of “visiting” node #10? ("head" of the queue, i.e. the next element that will be removed, is the leftmost)

A. 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11
B. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
C. 20, 21
D. 1, 2, 5
E. Other/none/more
BFS code

BFS (node)
{
    Queue<Bnode> q = new Queue<Bnode>();
    q.add(node);
    while (!q.isEmpty()) {
        current = q.remove();
        System.out.println(current.getData());
        A. q.add(current.getLeftChild());
        B. q.add(current.getRightChild());
        C. q.add(q.remove());
        D. q.sort();
        E. Other/none/more
    }
}
Binary Search Trees
Which of the following is/are a binary search tree?

A.  
B.  
C.  
D.  
E. More than one of these
Binary Search Tree (BST)

RULE:
- For **every** node:
  - Everything in its left subtree is less than it
  - Everything in its right subtree is greater than it
Binary Search Tree (BST)

FACT:
- The “in-order” traversal method, when applied to a BST, gives sorted order

FACT:
- Easy to find things: just recursively check if you should go left or right based on > or <
The BSTNode Class

1 /**
2 * A binary search tree is composed of <tt>BSTNode</tt>s.
3 * The element stored must be comparable with other
4 * elements in the BST.
5 */
6 public class BSTNode<E extends Comparable<? super E>> {
7     protected BSTNode<E> parent;
8     protected BSTNode<E> leftChild;
9     protected BSTNode<E> rightChild;
10     protected E element;

Note: the data fields are protected instead of private. Why?

Plus two constructors:
- one that takes a data element to store in the node
- one that takes a data element and references to become the
  node’s left and right subtrees
**BST Contains: Let’s write it!**

// Return true if toFind is in the BST rooted at root, 
// false otherwise

boolean contains( BSTNode root, E toFind ) {

    You will need two base cases. What are they? How can you check for them?
BST Contains: Let’s write it!

// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {

    Base case 1: Element is not in the tree
    Write a line of code that will check to see if we are in this base case
    And return the appropriate value
BST Contains: Let’s write it!

// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;

    Base case 2: Element is found
    We will roll this in with our recursive step
BST Contains: Let’s write it!

// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( _______________________________ )
        return contains( _____________, toFind );
    else if ( _______________________________ )
        return contains( _____________, toFind );
    else
        return _____________;

Recursive step and base case 2: How do you know which way to go? Fill in the blanks above. Hint: use compareTo.
// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return _______1_______________________;
    else if (toFind.compareTo( root.getElement() ) > 0 )
        return ________2______________________;
    else
        return ________3______________________;
}

What goes in blank 1?
A. false
B. true
C. contains( root.getLeftChild(), toFind )
D. contains( root.getRightChild(), toFind )
E. contains( root, toFind )
// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return contains( root.getLeftChild(), toFind );
    else if (toFind.compareTo( root.getElement() ) > 0 )
        return contains( root.getRightChild(), toFind );
    else
        return true;
}

What goes in blank 2?
A. false
B. true
C. contains( root.getLeftChild(), toFind )
D. contains( root.getRightChild(), toFind )
E. contains( root, toFind )
// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return contains( root.getLeftChild(), toFind );
    else if ( toFind.compareTo( root.getElement() ) > 0 )
        return contains( root.getRightChild(), toFind );
    else
        return ________3______________________;
}

What goes in blank 3?
A. false
B. true
C. contains( root.getLeftChild(), toFind )
D. contains( root.getRightChild(), toFind )
E. contains( root, toFind )
BST Contains: Let’s write it!

// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return contains( root.getLeftChild(), toFind );
    else if (toFind.compareTo( root.getElement() ) > 0 )
        return contains( root.getRightChild(), toFind );
    else
        return false;
contains() – a recursive method

contains( node, target ) =
\begin{cases} 
\text{false, if node is null} & \text{base case – failure} \\
\text{true, if node’s element is equal to the target} & \text{base case – success} \\
\text{contains( node’s left child, target), if target < node’s element} & \text{recursive case – left} \\
\text{contains( node’s right child, target), if target > node’s element} & \text{recursive case – right} 
\end{cases}

where:
node – the root of the subtree to search (initially the root of the tree)
target – the element for which we are searching

Call and return paths for successful and unsuccessful contains() calls
BST Add: Slightly different from your reading

```java
void add( E toAdd ) {
    if ( this.root == null )
        this.root = new BSTNode( toAdd );
    else
        addToExistingTree( root, toAdd );
}

void addToExistingTree( BSTNode root, E toAdd )
{
    ...
}
```

void addToExistingTree( BSTNode root, E toAdd )
{
    int value = toAdd.compareTo( root.getElement() );
    if ( value < 0 ) {
        if ( value.getLeftChild() == null ) {
            BSTNode n = new BSTNode( toAdd );
            root.setLeftChild( n );
            n.setParent( root );
        }
        else {
            addToExistingTree( root.getLeftChild(), toAdd );
        }
    }
    else if ( value > 0 ) {
        // Repeat for other side
    }
}
You should also be familiar with BST delete. We might ask you questions about it or ask you to write some part of it on the exam (not the whole thing from scratch, though). See your reading.
Is it bigger than a breadbox?

- no
  - Do you eat it with eggs?
    - no
      - a mouse
    - yes
      - Spam
- yes
  - Is it worth a lot of money?
    - no
      - a bag of trash
    - yes
      - Does it know Java?
        - no
          - a house
        - yes
          - a computer scientist
Is it bigger than a breadbox?

- No: Do you eat it with eggs?
  - No: a mouse
  - Yes: Spam

- Yes: Is it worth a lot of money?
  - No: a bag of trash
  - Yes: Does it know Java?
    - No: a house
    - Yes: a computer scientist

Let's say I want to add "an airplane" and the question "Does it fly" to the tree as the left child of Does it know Java? Draw the new tree. Which nodes do you need references to?