

Introduction to Object-Oriented Programming

Binary Search Trees

Christopher Simpkins

`chris.simpkins@gatech.edu`

Trees are everywhere.



¹Source:http://commons.wikimedia.org/wiki/File:Winnersh_Meadows_Trees.jpg

They're in our web browsers.

The screenshot shows a web browser window with the address bar displaying `www.cc.gatech.edu/~simpkins/teaching/gatech/cs1331/syllabus/cs1331-fall2013.html`. The page title is "CS 1331 Introduction to Object Oriented Programming". The main heading on the page is "CS 1331 Introduction to Object Oriented Programming" followed by "Fall 2013 Syllabus for Sections A1-A4, B1-B4, and GR". Below the heading, the text states "Last updated on 2013-12-03 at 20:10." and provides information about the syllabus, including a link to a general syllabus PDF and course policies. The browser's developer tools are open at the bottom, showing the DOM tree with the `h1` element selected. The DOM tree shows a document structure with a single `h1` element. The console, inspector, debugger, style editor, profiler, and network panels are also visible.

CS 1331 Introduction to Object Oriented Programming

GT | GT Login

CS 1331 Introduction to Object ...

CS 1331 (237 unread)

www.cc.gatech.edu/~simpkins/teaching/gatech/cs1331/syllabus/cs1331-fall2013.html

Google

Most Visited

Getting Started

Work

Personal

Scala

Diving

CS 1331 Introduction to Object Oriented Programming

Fall 2013 Syllabus for Sections A1-A4, B1-B4, and GR

Last updated on 2013-12-03 at 20:10.

This open-access part of the syllabus contains schedule and general information for my sections. All grade-related information are on T-Square.

There is also a general syllabus here: [cs1331-syllabus.pdf](#).

Please read and be sure you fully understand the [cs1331-course-policies.html](#).

Please read and be sure you fully understand the [cs1331-course-policies.html](#).

I'm serious! Please read and be sure you fully understand the [cs1331-course-policies.html](#) !!!

Console

Inspector

Debugger

Style Editor

Profiler

Network

html

body

div#cs-1331-introduction-to-object-or

h1.title

```
<!--?xml version="1.0" encoding="utf-8" ?-->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head> ... </head>
  <body>
    <div id="cs-1331-introduction-to-object-oriented-programming" class="document"> ... </div>
  </body>
</html>
```

Rules

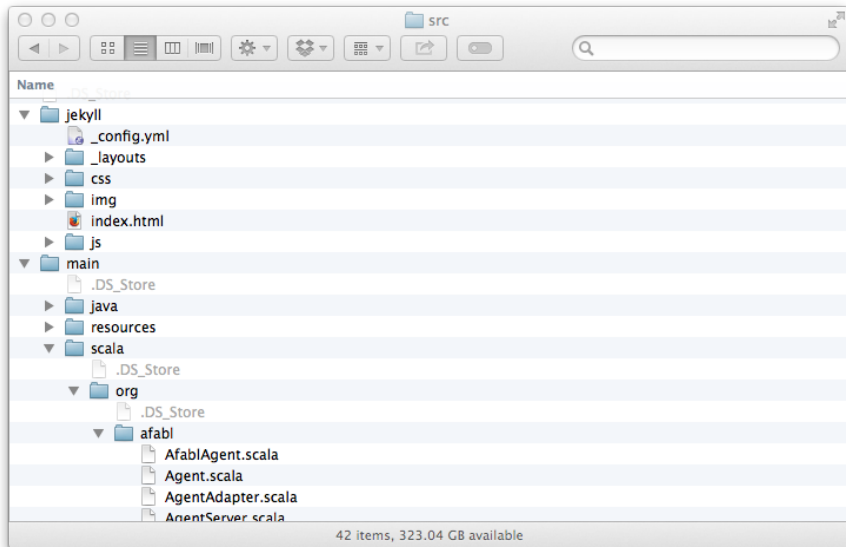
Computed

Fonts

Box

```
element {
} inline
```

They're in our file systems.



They're even in pop culture.



²Source:

<http://userserve-ak.last.fm/serve/500/44019065/Neon+Trees.png>

But they're not in Kansas.



3

³Source:

http://en.wikipedia.org/wiki/File:Wabaunsee_County_View.JPG



Binary Tree Nodes

The nodes of a binary tree have

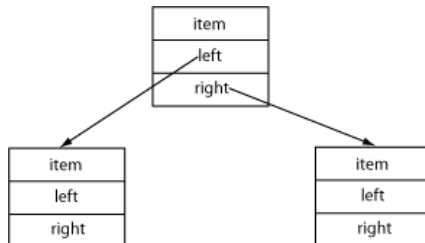
- a data item,
- a link to a left node, and
- a link to a right node.

```
private class Node<E> {  
    E item;  
    Node<E> left;  
    Node<E> right;  
  
    Node(E item, Node<E> left, Node<E> right) {  
        this.item = item;  
        this.left = left;  
        this.right = right;  
    }  
}
```

Just as in the other linked data structures we've studied, binary trees are recursive.

Binary Tree Structure

- Every tree has a distinguished *root node* with no parent.
 - All other nodes have exactly one parent.
- Nodes which have no children are called *leaf nodes*.
- Nodes which have children are called *interior nodes*.
- Every node has 0, 1, or 2 children.
- Every node can be reached by a unique path from the root node.



Binary Search Trees

A binary search tree (BST) encodes the binary search algorithm into its structure. The BST property: for any node,

- all the elements in the node's left subtree are less than the node's data item, and
- all the elements in the node's right subtree are equal to or greater than the node's data item.

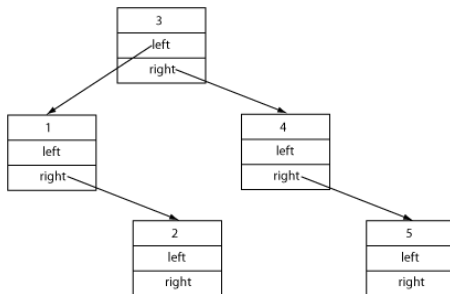
A BST is distinguished by this property, but it's ADT is just like the others we've seen: add elements, find element's in the tree, and iterate over the elements in a tree.

Maintaining The BST Property

To add a new value to binary tree and maintain the BST property, we

- insert new nodes for data items into the left subtree of a node if the new item is less than the node's item, or
- the right subtree otherwise.

Every new item creates a leaf node, which can later become an interior node after additional items have been added. Here's the structure of a BST after adding the sequence of numbers 3, 4, 1, 5, 2:



Adding Elements to a BST

```
public class BinarySearchTree<E extends Comparable<? super E>>
    implements Iterable<E> {

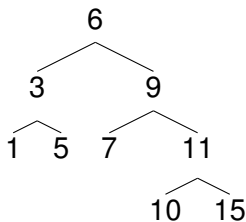
    protected class Node<E> { ... }

    protected Node<E> root;

    public void add(E item) {
        root = insert(item, root);
    }
    protected Node<E> insert(E newItem, Node<E> node) {
        if (node == null) {
            return new Node<E>(newItem, null, null);
        } else if (newItem.compareTo(node.item) < 0) {
            node.left = insert(newItem, node.left);
            return node;
        } else {
            node.right = insert(newItem, node.right);
            return node;
        }
    }
}
```

Exercise: Insertion Locations

Given the following tree that conforms to the binary search tree property:



Where would 2, 4, 8, and 16 be inserted in the tree?

Traversing a Binary Tree

There are three primary ways to traverse a binary tree:

Pre-order:

- Process node's item.
- Process left subtree.
- Process right subtree.

In-order:

- Process left subtree.
- Process node's item.
- Process right subtree.

Post-order:

- Process left subtree.
- Process right subtree.
- Process node's item.

Simple In-Order Traversal

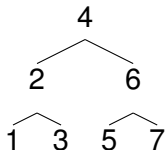
Traversal code follows the recursive structure of the tree:

```
public void printInOrder() {  
    printInOrder(root);  
}  
  
private void printInOrder(Node<E> node) {  
    if (node != null) {  
        printInOrder(node.left);  
        System.out.print(node.item + " ");  
        printInOrder(node.right);  
    }  
}
```

The code above prints the elements in ascending order. Let's add a `printDescending()` method to [BinarySearchTree.java](#).

Exercise: Traversal Orders

Given the following tree:



If we processed each element by printing it, in what order would the elements be printed

- For a pre-order traversal:
- For an in-order traversal:
- For a post-order traversal:

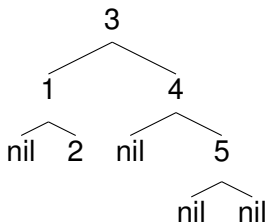
The Path to an Item

To find a path to an item in a BST:

- set the path to the empty list
- set the root node as the currentNode
- until we find the node containing the item or exhaust the BST:
 - if currentNode contains the item, add it to the path and return it
 - else if query item is less than the item in currentNode, add currentNode to path and set the left child as the new currentNode
 - else add add currentNode to path and set the right child as the new currentNode
- if the item wasn't found, set the path to the empty list

Path Examples

Adding the elements [3, 4, 1, 5, 2] to a BST would result in the following structure:



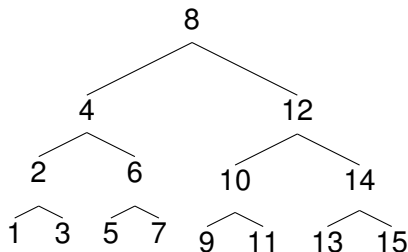
and the paths to each element in the tree would be:

- Path to 1: [3, 1]
- Path to 2: [3, 1, 2]
- Path to 3: [3]
- Path to 4: [3, 4]
- Path to 5: [3, 4, 5]

See `public List<E> path(E queryItem)` in [BinarySearchTree.java](#) for the code.

Exercise: Paths

Given the following tree:



- What's the path to 1?
- What's the path to 11?

Recursively Building a Result: `inOrderList()`

We can use the recursive accumulator idiom to collect the elements of the tree in an in-order traversal:

```
public List<E> toList() {  
    return inOrderList(root, new ArrayList<E>());  
}  
  
private List<E> inOrderList(Node<E> node, List<E> accum) {  
    if (null == node) {  
        return accum;  
    } else {  
        inOrderList(node.left, accum);  
        accum.add(node.item);  
        inOrderList(node.right, accum);  
    }  
    return accum;  
}
```

Again, the code follows the recursive structure of the tree.

Imperative traversal: `inOrderImperative()`

Contrast the previous code for getting an in-order list of BST elements with an imperative version:

```
public List<E> inOrderImperative() {
    Node<E> curNode = root;
    Stack<Node<E>> fringe = new LinkedStack<>();
    List<E> accum = new ArrayList<E>();
    while ((curNode != null) || !fringe.isEmpty()) {
        while (curNode != null) {
            fringe.push(curNode);
            curNode = curNode.left;
        }
        curNode = fringe.pop();
        accum.add(curNode.item);
        curNode = curNode.right;
    }
    return accum;
}
```

We need extra bookkeeping variables to keep track of where we are in the tree so we can back-track. See [BinarySearchTree.java](#) for comments explaining the algorithm.

Iterators

Iterators can free clients from having to implement traversal algorithms. We can even plug our data structures into Java's for-each loop by implementing `java.lang.Iterable`:

```
public interface Iterable<T> {  
    java.util.Iterator<T> iterator();  
}
```

As a reminder, `java.util.Iterator`:

```
public interface Iterator<E> {  
    boolean hasNext();  
  
    E next();  
  
    void remove();  
}
```

Stateful In-Order Tree Traversal

In the traversal examples we saw earlier the traversal order was effected by the method call stack. A stateful iterator is much more challenging because:

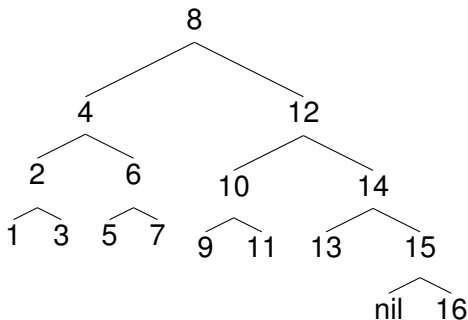
- The iterator must remember where it is in the tree
- The iterator must be able to back-track to parent nodes after processing child branches

The essential implementation idea is to use a stack to store nodes for back-tracking. Traditionally (at least in AI), this “to-do list” stack is called the *fringe*.

Let's look at [BinarySearchTree.java](#) again to see how we implement a stateful in-order iterator.

Analysis of BSTs

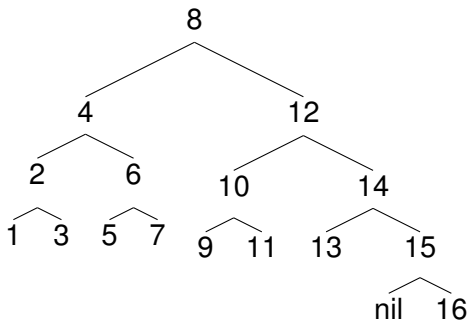
What is the Big-O of finding an element in a BST?



- Proportional to height of tree
- Height of tree is proportional to $\log n$
- 16-element tree has height 4

Analysis of BSTs

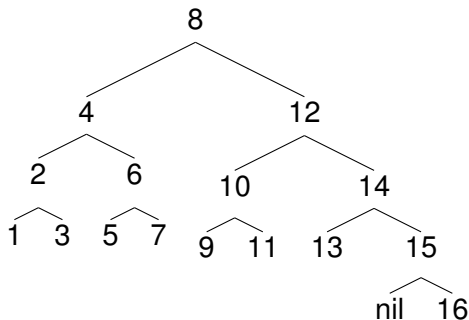
What is the Big-O of finding an element in a BST?



- Proportional to height of tree
- Height of tree is proportional to $\log n$
- 16-element tree has height 4

Analysis of BSTs

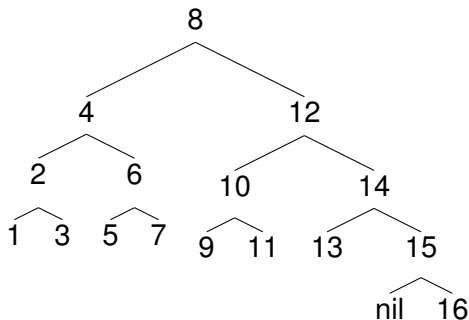
What is the Big-O of finding an element in a BST?



- Proportional to height of tree
- Height of tree is proportional to $\log n$
- 16-element tree has height 4

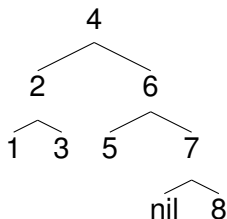
Analysis of BSTs

What is the Big-O of finding an element in a BST?



- Proportional to height of tree
- Height of tree is proportional to $\log n$
- 16-element tree has height 4

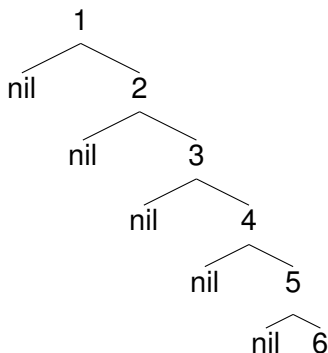
Height of BST Proportional to $\log n$



8-element tree has height 3

End of course cliff hanger ...

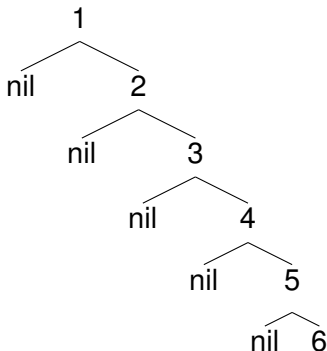
What if we add elements to a BST in order?



- We end up with a linked list!
- In your algorithms and data structures course you'll learn how to maintain a balanced binary search tree.

End of course cliff hanger ...

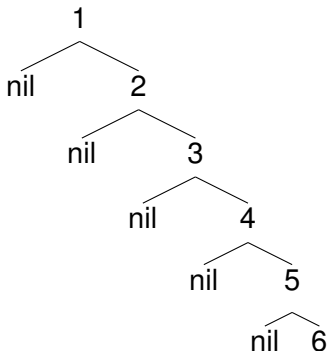
What if we add elements to a BST in order?



- We end up with a linked list!
- In your algorithms and data structures course you'll learn how to maintain a balanced binary search tree.

End of course cliff hanger ...

What if we add elements to a BST in order?



- We end up with a linked list!
- In your algorithms and data structures course you'll learn how to maintain a balanced binary search tree.