## CSE 143 Java

**Trees**



## Overview

- Topics
  - Trees: Definitions and terminology
  - Binary trees
  - Tree traversals
  - Binary search trees
  - Applications of BSTs



## Trees

- Most of the structures we've looked at so far are linear
  - Arrays
  - Linked lists
- There are many examples of structures that are not linear, e.g. hierarchical structures
  - Organization charts
  - Book contents (chapters, sections, paragraphs)
  - Class inheritance diagrams
- *Trees* can be used to represent hierarchical structures



## Looking Ahead To An Old Goal

- Finding algorithms and data structures for fast searching
  - A key goal
  - Sorted arrays are faster than unsorted arrays, for searching
    - Can use binary search algorithm
    - Not so easy to keep the array in order
  - LinkedLists were faster than arrays (or ArrayLists), for insertion and removal operations
    - The extra flexibility of the "next" pointers avoided the cost of sliding
    - But... LinkedLists are hard to search, even if sorted
- Is there an analogue of LinkedLists for sorted collections??
- The answer will be...Yes: a particular type of *tree*!

## Tree Definitions

- A *tree* is a collection of *nodes* connected by *edges*
- A *node* contains
  - Data (e.g. an Object)
  - References (edges) to two or more *subtrees* or *children*
- Trees are hierarchical
  - A node is said to be the *parent* of its *children* (subtrees)
  - There is a single unique *root* node that has no parent
  - Nodes with no children are called *leaf nodes*
  - A tree with no nodes is said to be *empty*

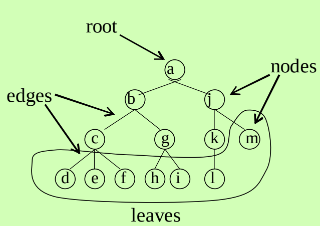


## Drawing Trees

- For whatever reason, computer sciences trees are normally drawn upside down: root at the top



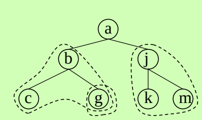
## Tree Terminology






## Subtrees

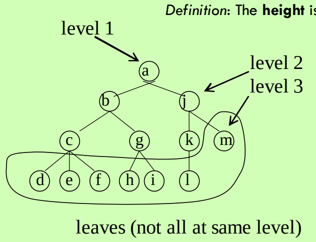
- A *subtree* in a tree is any node in the tree together with all of its descendants (its children, and their children, recursively)




- Note: note every sub*set* is a sub*tree*!

## Level and Height

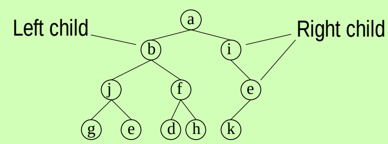
*Definition*: The root has **level** 1
Children have level 1 greater than their parent
*Definition*: The **height** is the highest level of a tree.






## Binary Trees

- A *binary tree* is a tree each of whose nodes has no more than two children
  - The two children are called the *left child* and *right child*
  - The subtrees belonging to those children are called the *left subtree* and the *right subtree*






## Binary Tree Implementation

- A node for a binary tree holds the item and references to its subtrees

```
public class BTNode {
    public Object item;          // data item in this node
    public BTNode left;          // left subtree, or null if none
    public BTNode right;         // right subtree, or null if none
    public BTNode(Object item, BTNode left, BTNode right) { ... }
}
```

- The whole tree can be represented just by a pointer to the root node, or null if the tree is empty

```
public class BinTree {
    private BTNode root;         // root of tree, or null if empty
    public BinTree( ) { this.root = null; }
    ...
}
```



## Tree Algorithms

- The definition of a tree is naturally recursive:
  - A tree is either null,
    or data + left (sub-)tree + right (sub-)tree
  - Base case(s)?
  - Recursive case(s)?
- Given a recursively defined data structure, recursion is often a very natural technique for algorithms on that data structure
  - Don't fight it!

## A Typical Tree Algorithm: size( )

```
public class BinTree {
    …
    /** Return the number of items in this tree */
    public int size( ) {
        return subtreeSize(root);
    }
    // Return the number of nodes in the (sub-)tree with root n
    private int subtreeSize(BTNode n) {
        if (n == null) {
            return 0;
        } else {
            return 1 + subtreeSize(n.left) + subtreeSize(n.right);
        }
    }
```



## Tree Traversal

- Functions like subtreeSize systematically "visit" each node in a tree
  - This is called a *traversal*
  - We also used this word in connection with lists
- Traversal is a common pattern in many algorithms
  - The processing done during the "visit" varies with the algorithm
- What order should nodes be visited in?
  - Many are possible
  - Three have been singled out as particularly useful for binary trees: *preorder*, *postorder*, and *inorder*



## Traversals

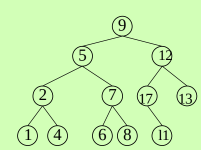
- **Preorder** traversal:
  - "Visit" the (current) node first
    i.e., do what ever processing is to be done
  - Then, (recursively) do preorder traversal on its children, left to right
- **Postorder** traversal:
  - First, (recursively) do postorder traversals of children, left to right
  - Visit the node itself last
- **Inorder** traversal:
  - (Recursively) do inorder traversal of left child
  - Then visit the (current) node
  - Then (recursively) do inorder traversal of right child

  Footnote: pre- and postorder make sense for all trees; inorder only for binary trees



## Example of Tree Traversal

In what order are the nodes visited, if we start the process at the root?



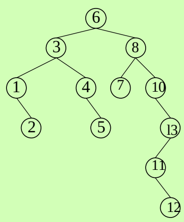

Preorder:

Inorder:

Postorder:

## More Practice

What about this tree?




Preorder:

Inorder:

Postorder:



## New Algorithm: Contains

- Return whether or not a value is an item in the tree

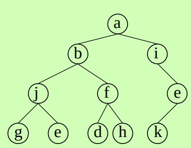
```
public class BinTree {
    ...
    /** Return whether elem is in tree */
    public boolean contains(Object elem) {
        return subtreeContains(root, elem);
    }
    // Return whether elem is in (sub-)tree with root n
    private boolean subtreeContains(BTNode n, Object elem) {
        if (n == null) {
            return false;
        } else if (n.item.equals(elem)) {
            return true;
        } else {
            return subtreeContains(n.left, elem) || subtreeContains(n.right, elem);
        }
    }
```



## Test

contains(d)

contains(c)






## Cost of Contains

- Work done at each node:
- Number of nodes visited:
- Total cost:
- Can we do better?
  - Why was binary search so much better than linear search?
  - Can we apply the same idea to trees?

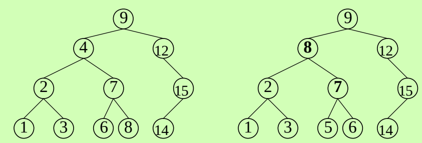
## Binary Search Trees

- Idea: order the nodes in the tree so that, given that a node contains a value *v*,
  - All nodes in its left subtree contain values <= *v*
  - All nodes in its right subtree contain values >= *v*
- A binary tree with these properties is called a *binary search tree* (BST)



## Examples(?)

- Are these are binary search trees? Why or why not?






## Implementing a Set with a BST

- Can exploit properties of BSTs to have fast, divide-and-conquer implementations of Set's add and contains operations
  - TreeSet!
- A TreeSet can be represented by a pointer to the root node of a binary search tree, or null of no elements yet

```
public class SimpleTreeSet implements Set {
    private BTNode root;        // root node, or null if none
    public SimpleTreeSet( ) { this.root = null; }
    // size as for BinTree
    …
}
```



## Contains

- Original contains had to search both subtrees
  - Like linear search
- With BSTs, can only search one subtree!
  - All small elements to the left, all large elements to the right
  - Search either left or right subtree, based on comparison between elem and value at root of tree
  - Like binary search

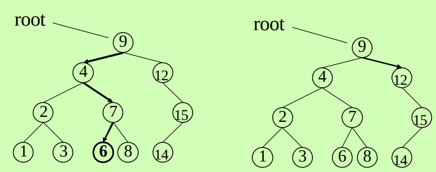
## Code for contains (in TreeSet)

```
/** Return whether elem is in set */
public boolean contains(Object elem) {
    return subtreeContains(root, (Comparable)elem);
}
// Return whether elem is in (sub-)tree with root n
private boolean subtreeContains(BTNode n, Comparable elem) {
    if (n == null) {
        return false;
    } else {
        int comp = elem.compareTo(n.item);
        if (comp == 0) { return true; }                // found it!
        else if (comp < 0) { return subtreeContains(n.left, elem); }       // search left
        else /* comp > 0 */ { return subtreeContains(n. right, elem); }   // search right
    }
}
```



## Examples

contains(6) contains(10)






## Cost of Contains

- Work done at each node:
- Number of nodes visited (depth of recursion):
- Total cost:



## Add

- Must preserve BST invariant: insert new element in correct place in BST
- Two base cases
  - Tree is empty: create new node which becomes the root of the tree
  - If node contains the value, found it; suppress duplicate add
- Recursive case
  - Compare value to current node's value
  - If value < current node's value, add to left subtree recursively
  - Otherwise, add to right subtree recursively

## Example

- Add 8, 10, 5, 1, 7, 11 to an initially empty BST, in that order:



## Example (2)

- What if we change the order in which the numbers are added?
- Add 1, 5, 7, 8, 10, 11 to a BST, in that order (following the algorithm):



## Code for add (in TreeSet)

```
/** Ensure that elem is in the set.  Return true if elem was added, false otherwise. */
public boolean add(Object elem) {
    try {
        BTNode newRoot = addToSubtree(root, (Comparable)elem);   // add elem to tree
        root = newRoot;                  // update root to point to new root node
        return true;                     // return true (tree changed)
    } catch (DuplicateAdded e) {
        // detected a duplicate addition
        return false;                    // return false (tree unchanged)
    }
}
/** Add elem to tree rooted at n.  Return (possibly new) tree containing elem, or throw
DuplicateAdded if elem already was in tree */
private BTNode addToSubtree(BTNode n, Comparable elem) throws DuplicateAdded {
    … }
```



## Code for addToSubtree

```
/** Add elem to tree rooted at n.  Return (possibly new) tree containing elem, or throw
DuplicateAdded if elem already was in tree */
private BTNode addToSubtree(BTNode n, Comparable elem) throws DuplicateAdded {
    if (n == null) { return new BTNode(elem, null, null); }     // adding to empty tree
    int comp = elem.compareTo(n.item);
    if (comp == 0) { throw new DuplicateAdded( ); }             // elem already in tree
    if (comp < 0) {                                             // add to left subtree
        BTNode newSubtree = addToSubtree(n.left, elem);
        n.left = newSubtree;                                    // update left subtree
    } else /* comp > 0 */ {                                     // add to right subtree
        BTNode newSubtree = addToSubtree(n.right, elem);
        n.right = newSubtree;                                   // update right subtree
    }
    return n;    // this tree has been modified to contain elem
}
```

## Cost of add

- Cost at each node:
- How many recursive calls?
  - Proportional to height of tree
  - Best case?
  - Worst case?



## A Challenge: iterator

- How to return an iterator that traverses the sorted set in order?
  - Need to iterate through the items in the BST, from smallest to largest
- Problem: how to keep track of position in tree where iteration is currently suspended
  - Need to be able to implement next( ), which advances to the correct next node in the tree
- Solution: keep track of a path from the root to the current node
  - Still some tricky code to find the correct next node in the tree



## Another Challenge: remove

- Algorithm: find the node containing the element value being removed, and remove that node from the tree
- Removing a leaf node is easy: replace with an empty tree
- Removing a node with only one non-empty subtree is easy: replace with that subtree
- How to remove a node that has two non-empty subtrees?
  - Need to pick a new element to be the new root node, and adjust at least one of the subtrees
  - E.g., remove the largest element of the left subtree (will be one of the easy cases described above), make that the new root



## Analysis of Binary Search Tree

- Cost of operations is proportional to height of tree
- Best case: tree is *balanced*
  - Depth of all leaf nodes is roughly the same
  - Height of a balanced tree with $n$ nodes is ~$\log_2 n$
- If tree is unbalanced, height can be as bad as the number of nodes in the tree
  - Tree becomes just a linear list

## Summary

- A binary search tree is a good general implementation of a set, if the elements can be ordered
  - Both contains and add benefit from divide-and-conquer strategy
  - No sliding needed for add
  - Good properties depend on the tree being roughly balanced
- Open issues (or, why take a data structures course?)
  - How are other operations implemented (e.g. iterator, remove)?
  - Can you keep the tree balanced as items are added and removed?