CSCI 241

Scott Wehrwein

Trees: Traversals and Thinking Recursively

Goals

Be able to implement a simple tree class and basic operations such as **`search`** and **`size`**.

Know how to execute on paper and implement pre-order, in-order, and post-order tree traversals.

Implementing Trees

```
/** A binary tree */
public class Tree {
   int value;
   Tree left;
   Tree right;
}
```

```
/** A general tree */
public class Tree {
   int value;
   List<Tree> children;
}
```

This is a *recursive* definition.

Thinking about trees recursively

A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

Thinking about trees recursively

A binary tree is

```
/** A binary tree */
public class Tree {
   int value;
   Tree left;
   Tree right;
}
```

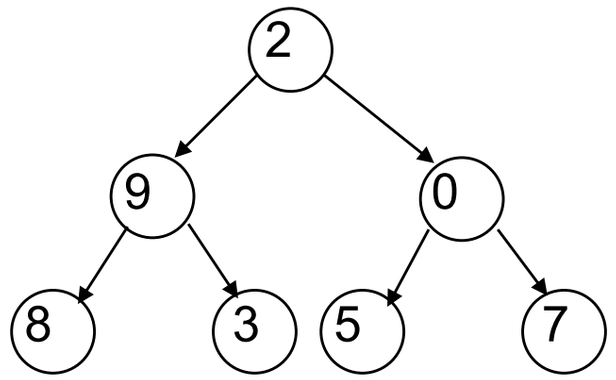
- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

Thinking about trees recursively

A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

```
/** A binary tree */
public class Tree {
   int value;
   Tree left;
   Tree right;
}
```

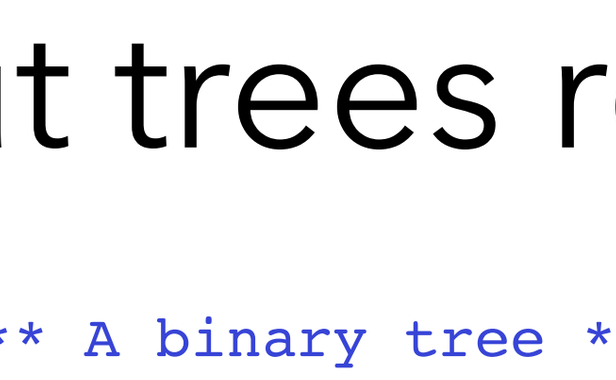


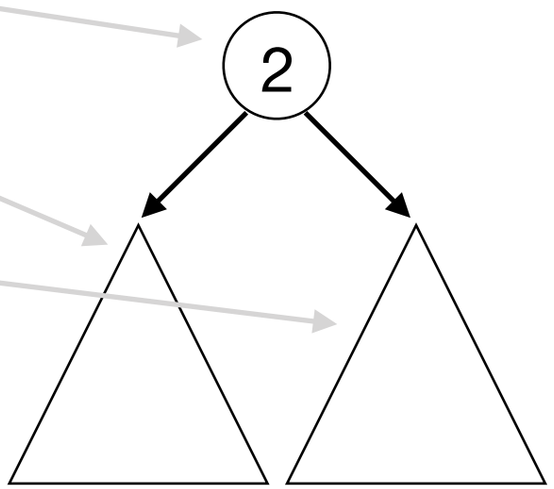
Thinking about trees recursively

A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

```
/** A binary tree */
public class Tree {
   int value;
   Tree left;
   Tree right;
}
```





Operations on trees

often follow naturally from the *definition* of the tree:

A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

Find v in a binary tree:

(base case - not found!)

(base case - is this v?)

(recursive call - is v in left?)

(recursive call - is v in right?)

Finding a value in a tree

code follows naturally from the definition of the tree:

A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

Find v in a binary tree:

```
boolean findVal(Tree t, int v):
```

(base case - not found!)

```
    if t == null:
        return false
```

(base case - is this v?)

```
if t.value == v: return true
```

(recursive call - is v in left?)

```
return findVal(t.left)
    || findVal(t.right)
```

(recursive call - is v in right?)

Traversing a Tree

Print (or "visit") every node in a tree:

A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree

Print all nodes in a binary tree:

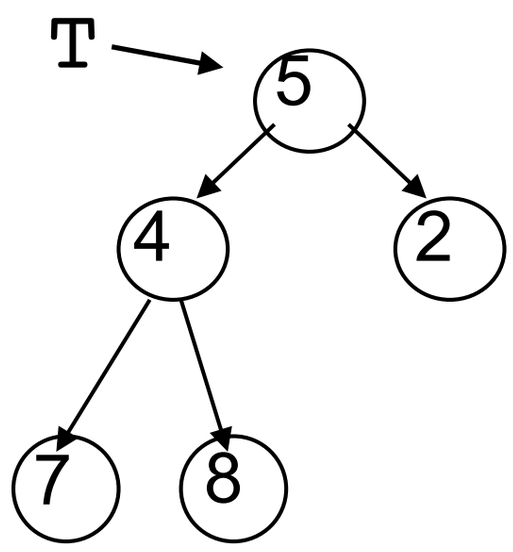
```
void printTree(Tree t):
    (base case - nothing to print)
    if t == null:
        return

    (print this node's value)
    System.out.println(t.value)

    (recursive call - print left subtree)
    printTree(t.left)
    (recursive call - print right subtree)
    printTree(t.right)
```

Traversing a Tree

Print (or otherwise process) every node in a tree:



Example: `printTree(T)`

Print all nodes in a binary tree:

```
void printTree(Tree t):
```

(base case - nothing to print)

```
if t == null:
   return
```

(print this node's value)

```
System.out.println(t.value)
```

(recursive call - print left subtree)

```
printTree(t.left)
```

(recursive call - print right subtree

```
printTree(t.right)
```

Tree Traversals

"Walking" over the whole tree is called a tree traversal

This is done often enough that there are standard names.

Previous example was a pre-order traversal:

1. Process root
2. Process left subtree
3. Process right subtree

Other common traversals:

in-order traversal:

1. Process left subtree
2. Process root
3. Process right subtree

post-order traversal:

1. Process left subtree
2. Process right subtree
3. Process root