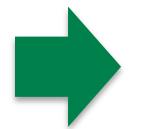


CS 10: Problem solving via Object Oriented Programming

Hierarchies

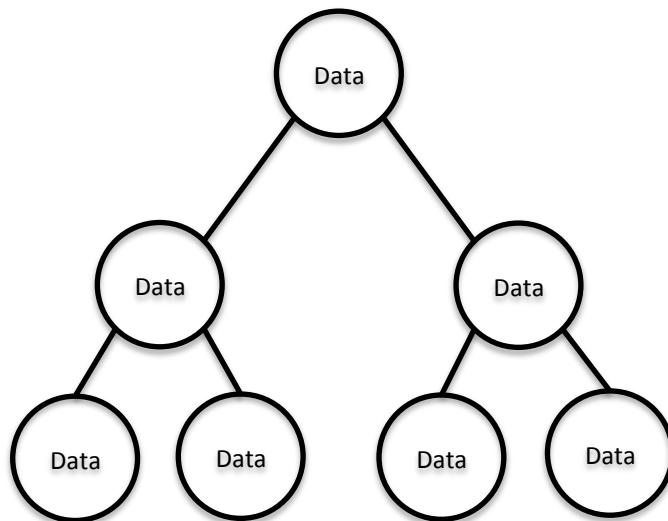
Agenda



1. General-purpose binary trees
2. Accumulators
3. Tree traversal
4. Expression trees

We can represent hierarchical data using a data structure called a tree

Tree data structure

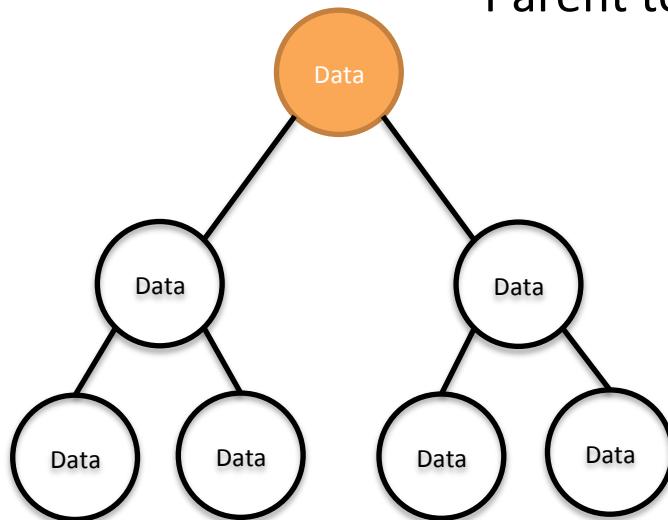


Meant for hierarchical data where there is a relationship between the data each node holds

We can represent hierarchical data using a data structure called a tree

Tree data structure

- Root node
- Parent to two children (called left and right)

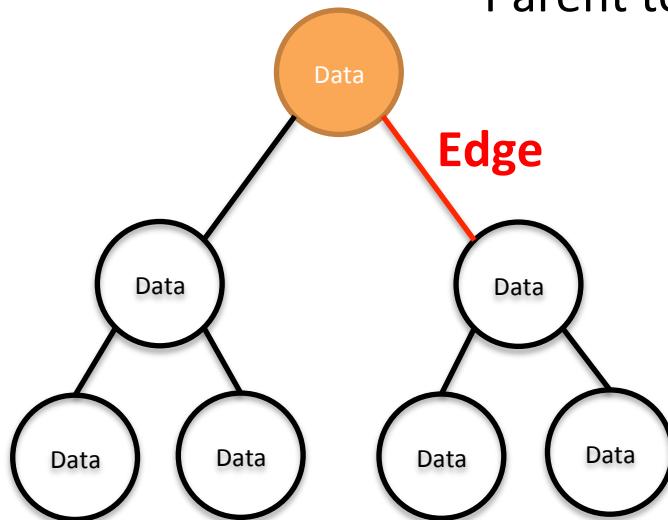


Meant for hierarchical data where there is a relationship between the data each node holds

We can represent hierarchical data using a data structure called a tree

Tree data structure

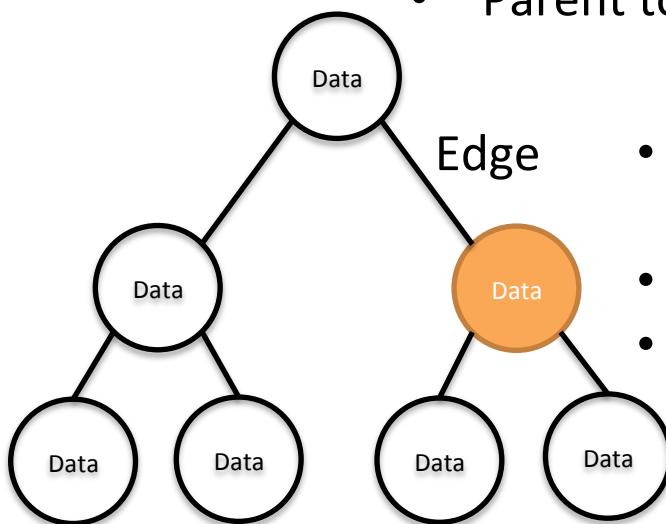
- Root node
- Parent to two children (called left and right)



Meant for hierarchical data where there is a relationship between the data each node holds

We can represent hierarchical data using a data structure called a tree

Tree data structure

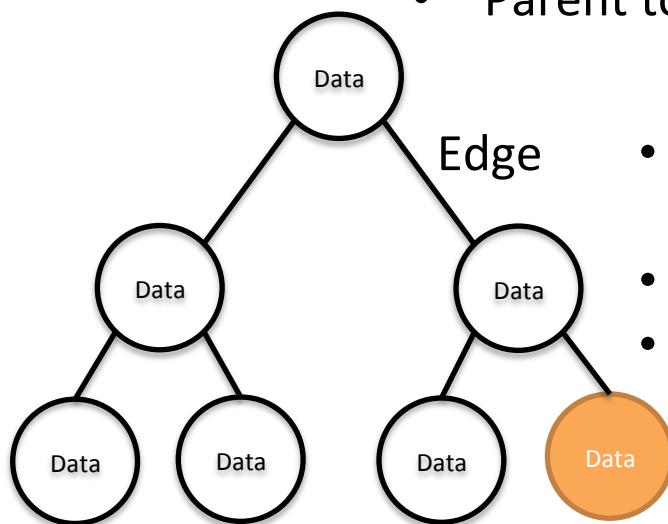


- Root node
- Parent to two children (called left and right)
- Child node of root (children have exactly one parent)
- Parent node to children below
- Interior node (nodes also called vertices)

Meant for hierarchical data where there is a relationship between the data each node holds

We can represent hierarchical data using a data structure called a tree

Tree data structure

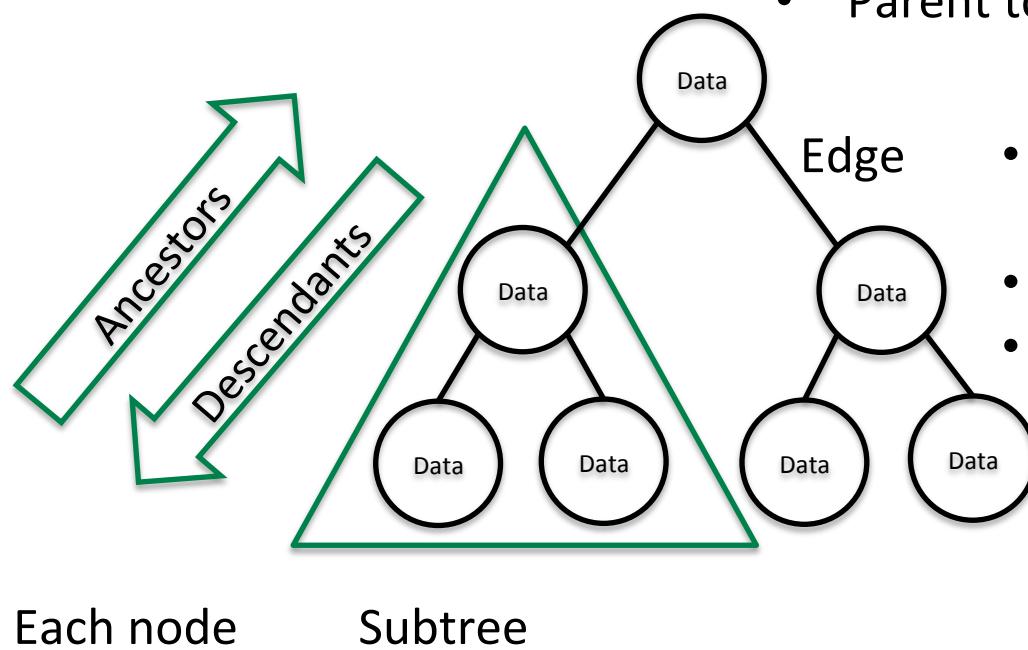


- Root node
- Parent to two children (called left and right)
- Child node of root (children have exactly one parent)
- Parent node to children below
- Interior node (nodes also called vertices)
- Leaf (or external) node
- Right child of parent node

Meant for hierarchical data where there is a relationship between the data each node holds

We can represent hierarchical data using a data structure called a tree

Tree data structure



- Root node
- Parent to two children (called left and right)
- Child node of root (children have exactly one parent)
- Parent node to children below
- Interior node (nodes also called vertices)
- Leaf (or external) node
- Right child of parent node

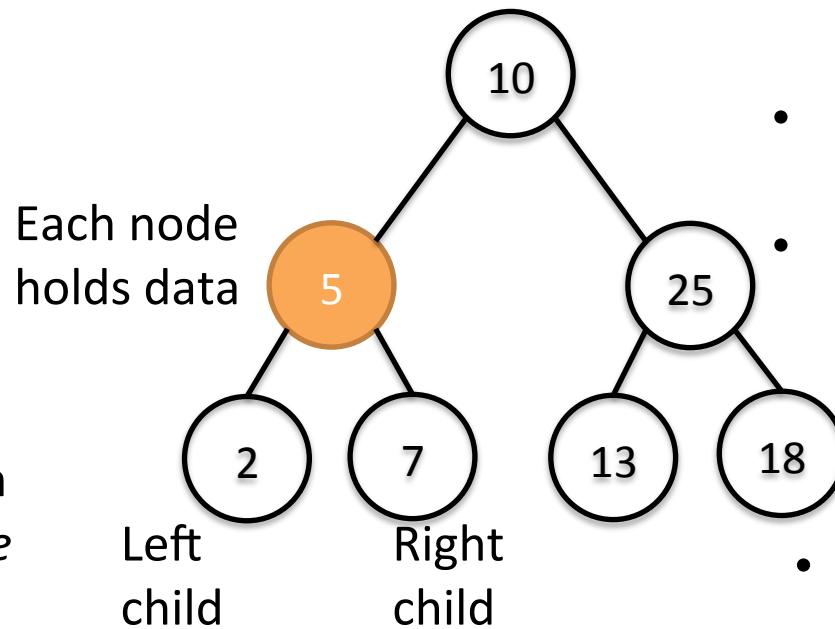
Meant for hierarchical data where there is a relationship between the data each node holds

Each node can be thought of as the root of a subtree

In a Binary Tree, each node has data plus 0, 1, or 2 children

Binary Tree data structure

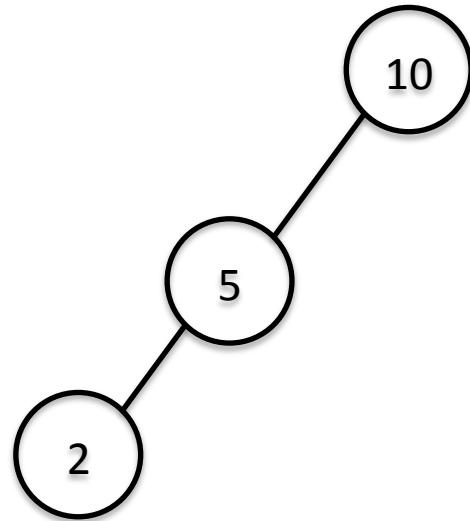
0, 1, or 2
children in
BinaryTree



- An *interior* node has at least one non-null child
- It could have two non-null children
- Leaf nodes have left and right children too, they are both just null
- We will commonly talk about them, however, as having no children

A Binary Tree does not need to be balanced

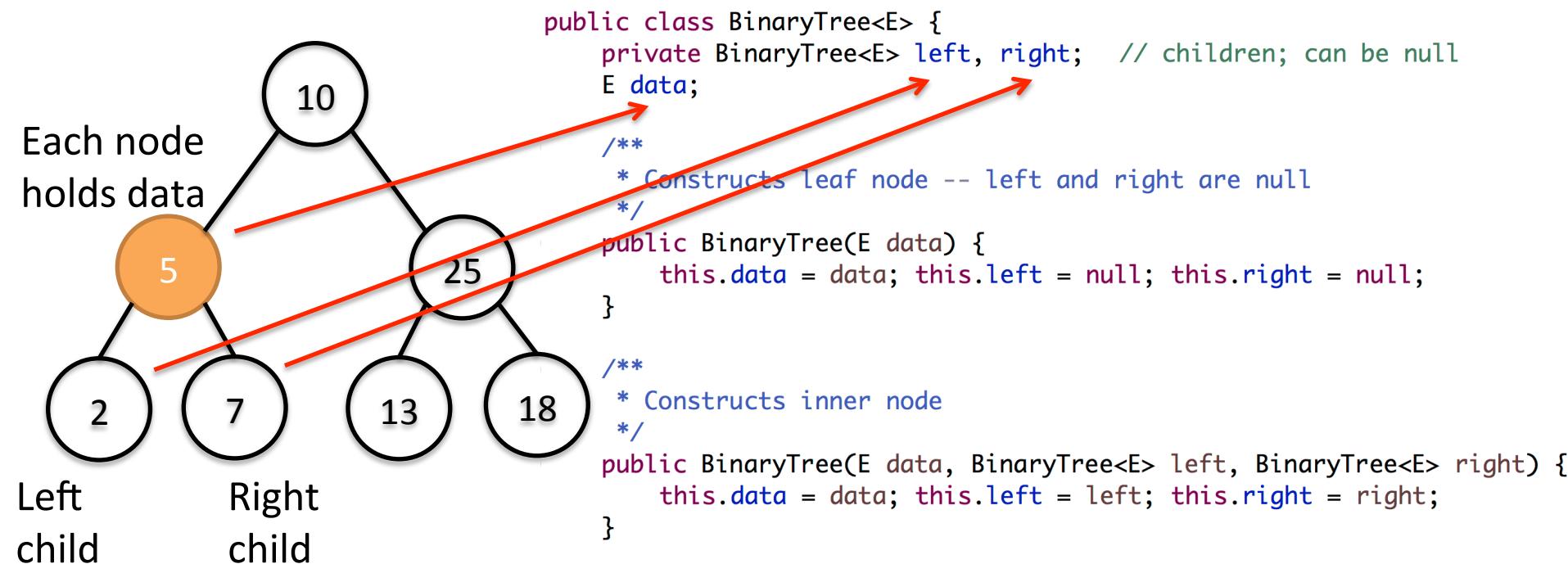
Binary Tree data structure



- This is a valid Binary Tree, each node has 0, 1, (or 2) children
- For now we make no guarantees a tree is balanced
- Later we will look at ways to ensure balance
- Balance will allow us to make stronger statements about run time performance

Each node in a tree can be thought of as the head of it's own subtree

BinaryTree.java

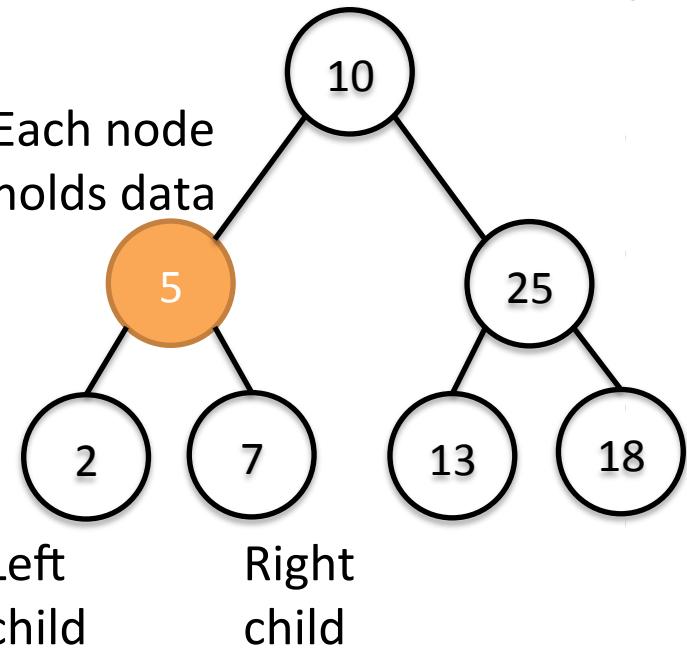


- Define a Tree with data element of generic type E plus left and right children
- Children are (sub) Trees themselves, so their type is BinaryTree
- No need to define a Tree Class and separate TreeNode Class
- Because of this structure, most Tree code is recursive

Each node in a tree can be thought of as the head of it's own subtree

BinaryTree.java

Each node holds data



```
public class BinaryTree<E> {  
    private BinaryTree<E> left, right; // children; can be null  
    E data;  
  
    /**  
     * Constructs leaf node -- left and right are null  
     */  
    public BinaryTree(E data) {  
        this.data = data; this.left = null; this.right = null;  
    }  
  
    /**  
     * Constructs inner node  
     */  
    public BinaryTree(E data, BinaryTree<E> left, BinaryTree<E> right) {  
        this.data = data; this.left = left; this.right = right;  
    }  
}
```

Two constructors

- One for leaf node
- One for interior node

Building a Binary Tree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right;
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

Create root node

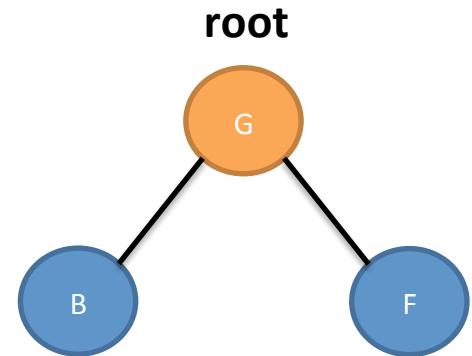


Building a Binary Tree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right;
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

Set left and right children

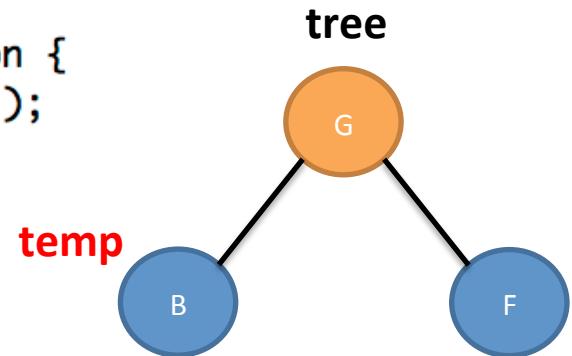


Building a Binary Tree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right;
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

Make temp node and traverse
down to left child



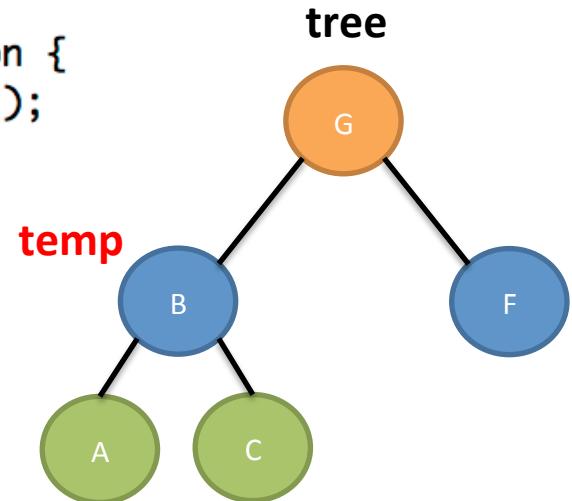
- What would happen if didn't create *temp = root.left*, but instead set *root = root.left*
- Would lose pointer to *root* node (*root* would be garbage collected)

Building a Binary Tree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right;
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

Set left and right children

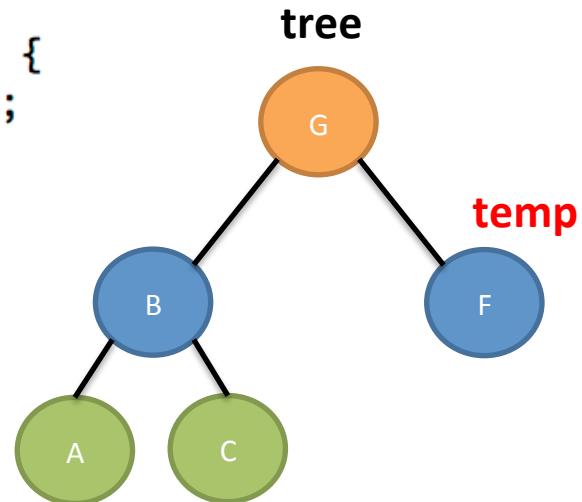


Building a Binary Tree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right; ↓
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

Move temp to root's right child

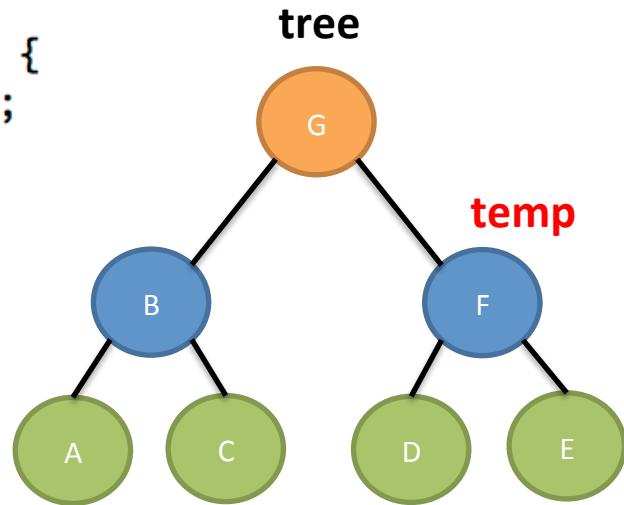


Building a Binary Tree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right;
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

Add children

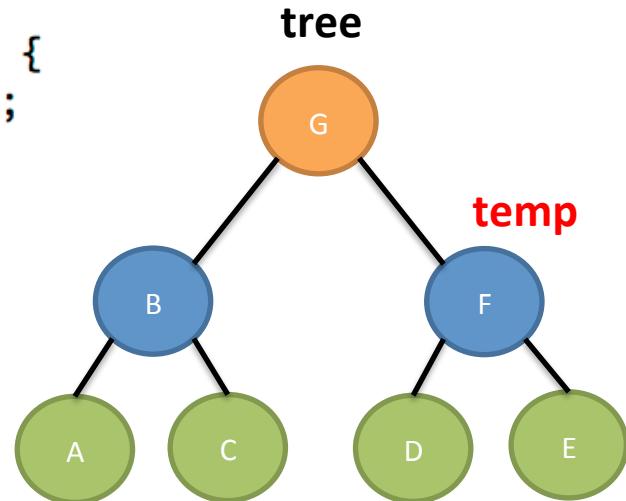
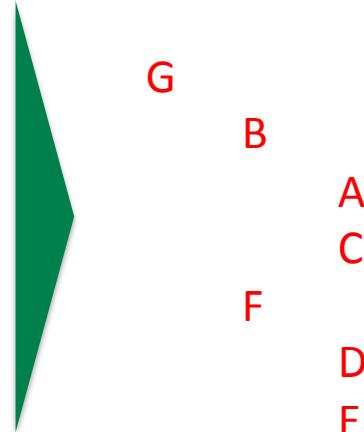


Building a BinaryTree

BinaryTree.java

```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String> temp = root.left;
    temp.left = new BinaryTree<String>("A");
    temp.right = new BinaryTree<String>("C");
    temp = root.right;
    temp.left = new BinaryTree<String>("D");
    temp.right = new BinaryTree<String>("E");
    System.out.println(root);
```

- Print tree from root
- Implicitly calls *toString()*
- Will define in a few slides
- Note: Nodes are not *required* to be in alphabetical order in this tree (check F and E)



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

size() returns the number of nodes in the (sub) tree

```
75  /**
76   * Number of nodes (inner and leaf) in tree
77   */
78  public int size() {
79      int num = 1;
80      if (hasLeft()) num += left.size();    One to account for this node
81      if (hasRight()) num += right.size();  hasLeft() and hasRight() return
82      return num;                         true if node has those children
83  }                                         Only make recursive call if node
84
```

Ask each child to return its size and add to *num*

Return size of this subtree

If leaf node, will return 1

Recursion will then “bubble up” until it gets back to the original node on which *size()* was called

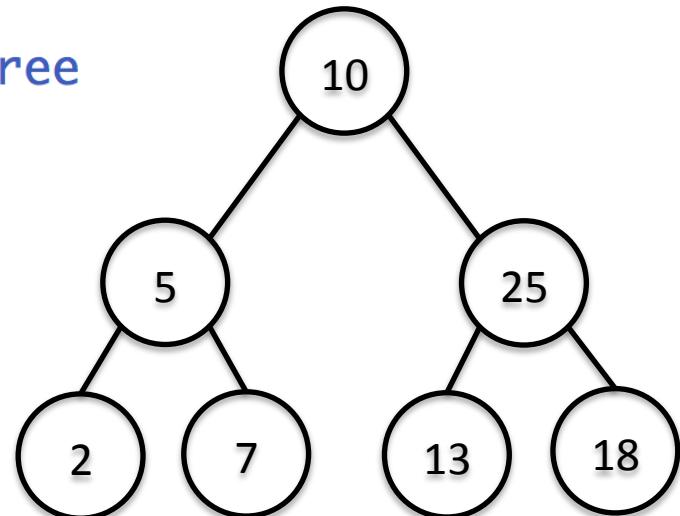
In that node *num* will then have the size of the entire subtree

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

Call `size()` on root node

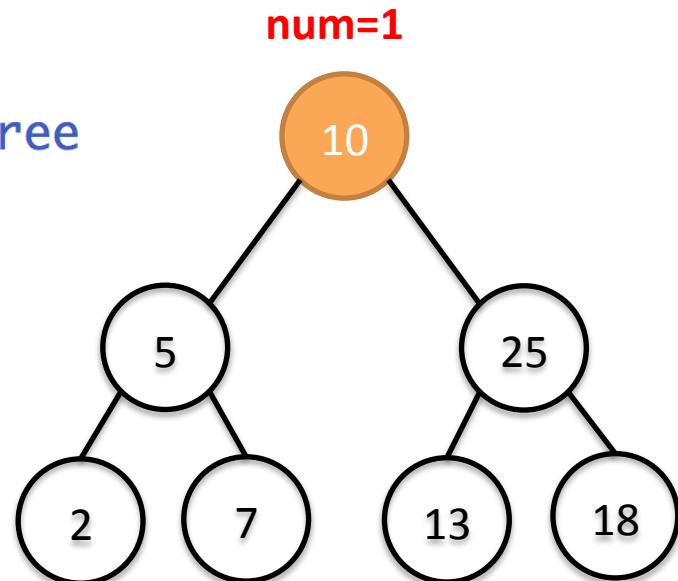
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

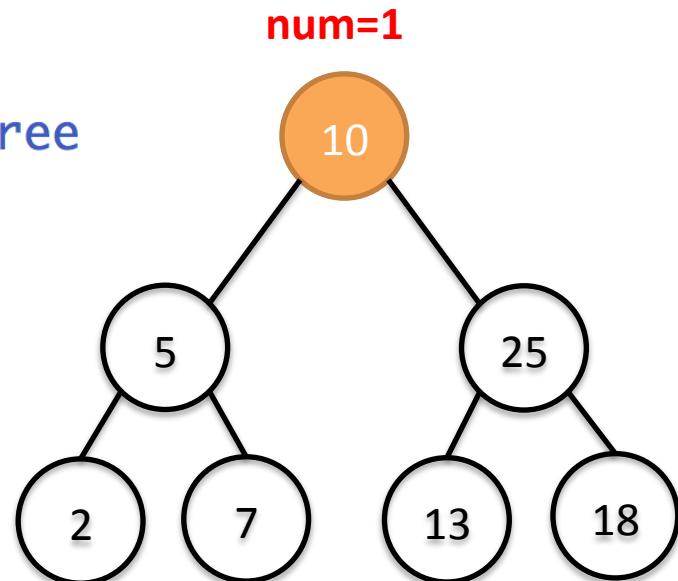
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    10 → int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    10 if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```

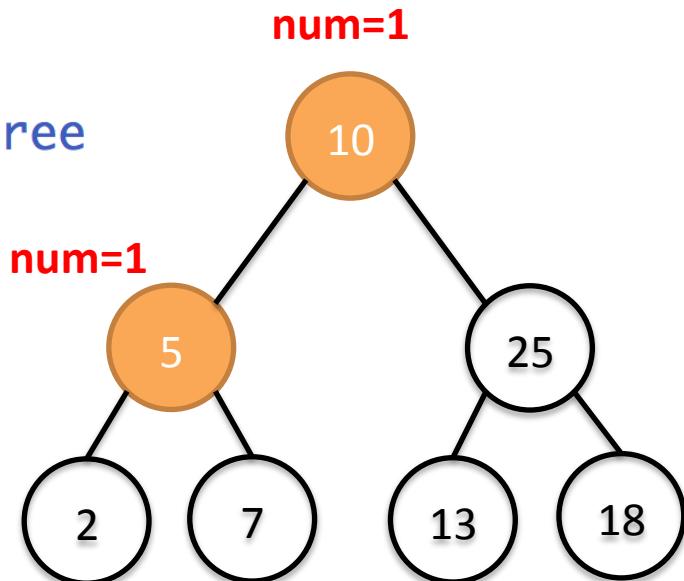


- Has left child
- Make recursive call on left child

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

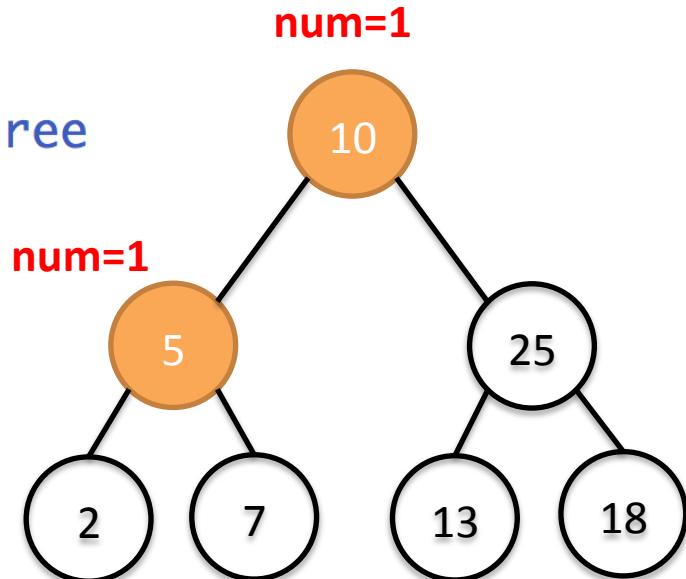
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```

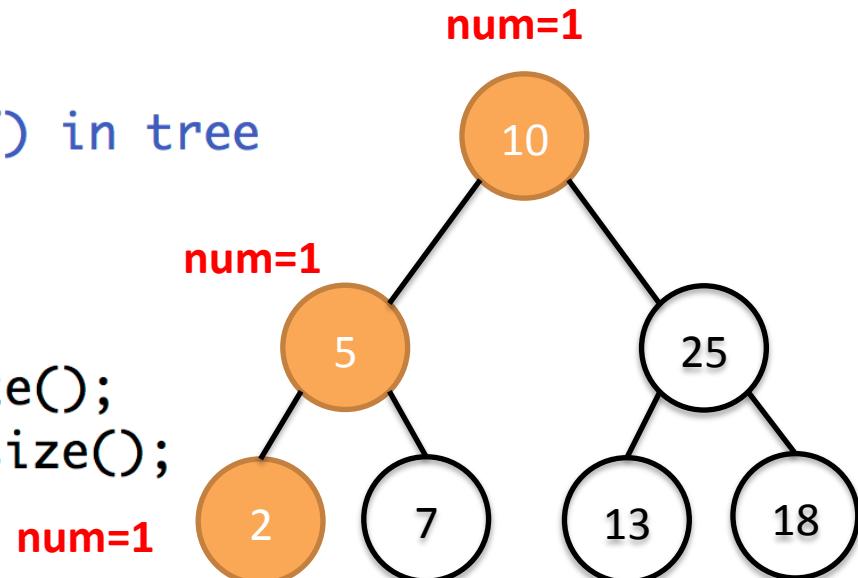


- Has left child
- Make recursive call on left child

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

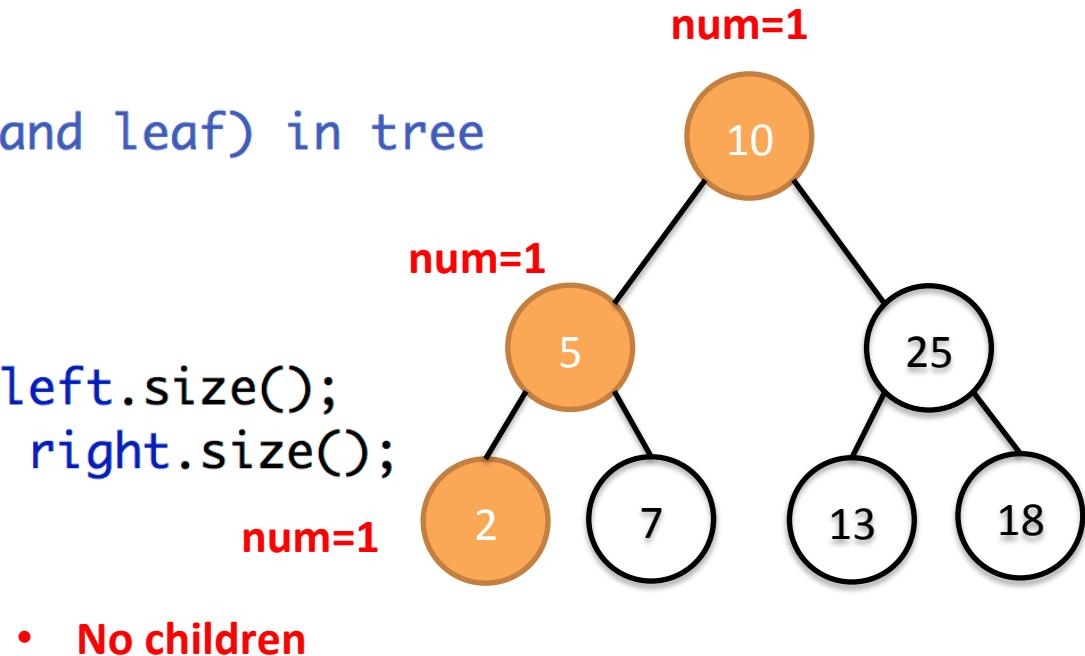
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    2 if (hasLeft()) num += left.size();  
    5 if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

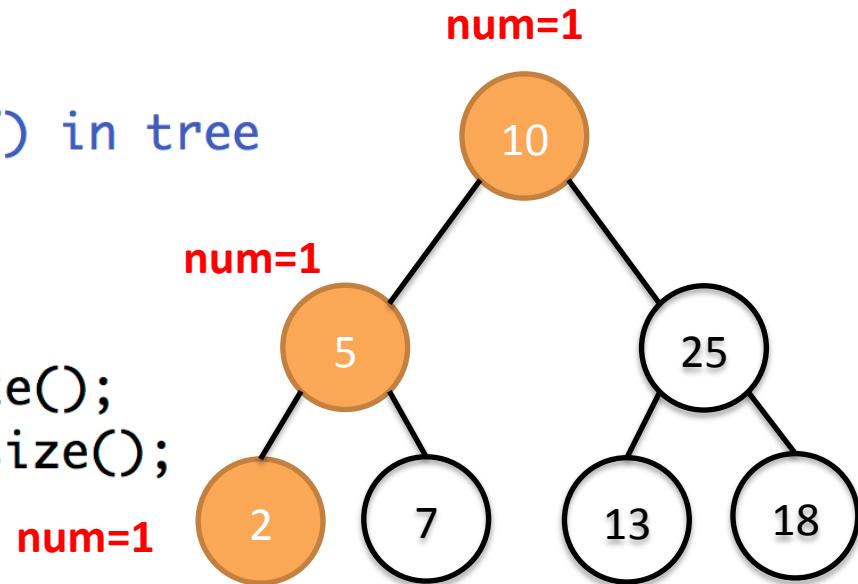
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    2 if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    5 → if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    2 → return num;  
}
```

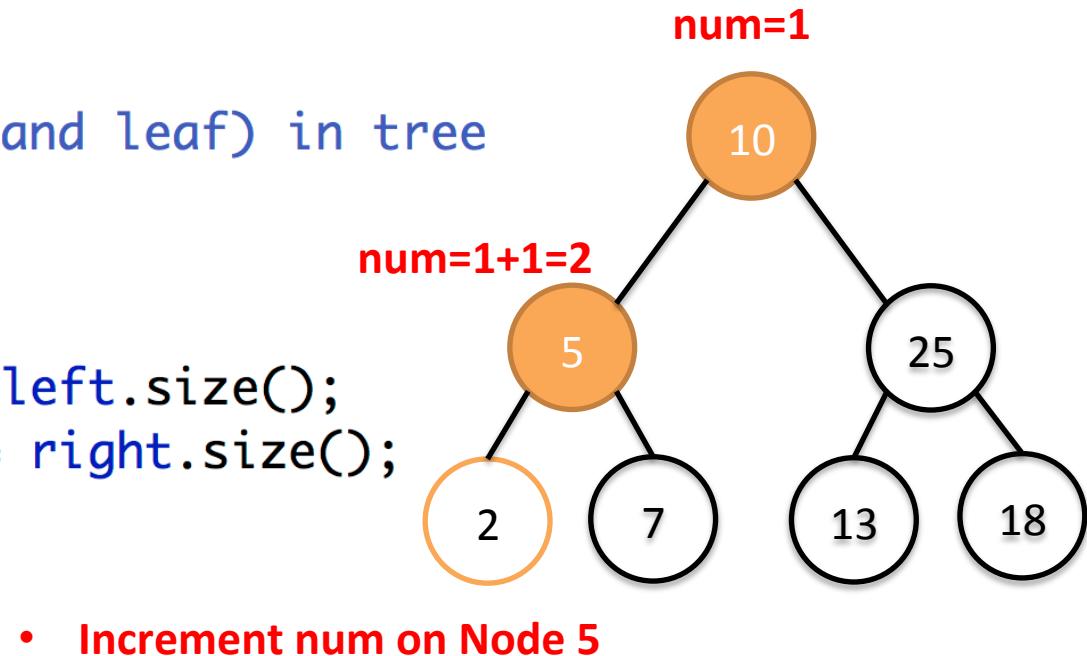


- No children
- Return 1 back to node 5

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

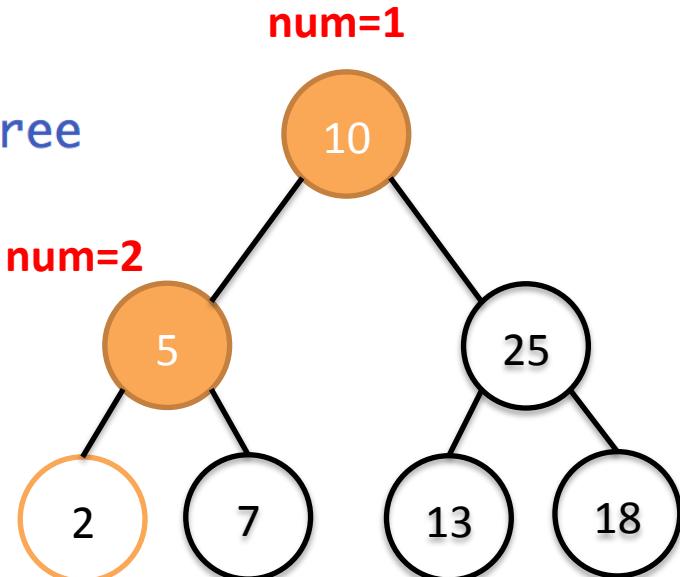
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    10 if (hasLeft()) num += left.size();  
    5 if (hasRight()) num += right.size();  
    return num;  
}
```

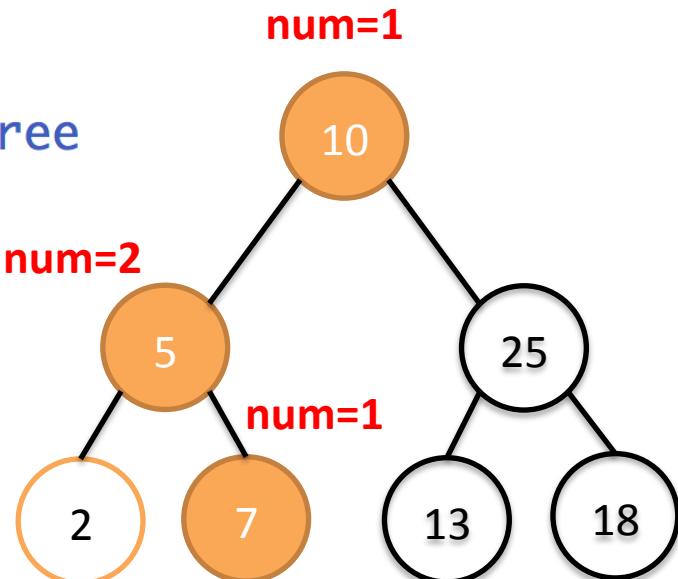


- Has right child
- Make recursive call on right child

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

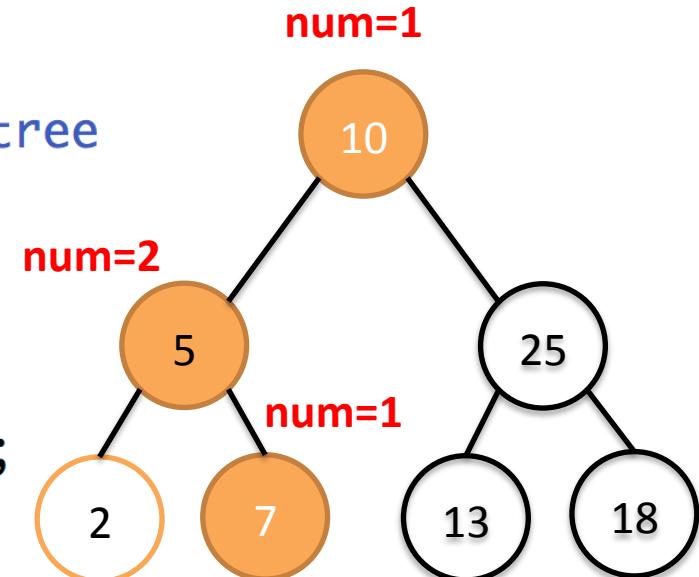
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```

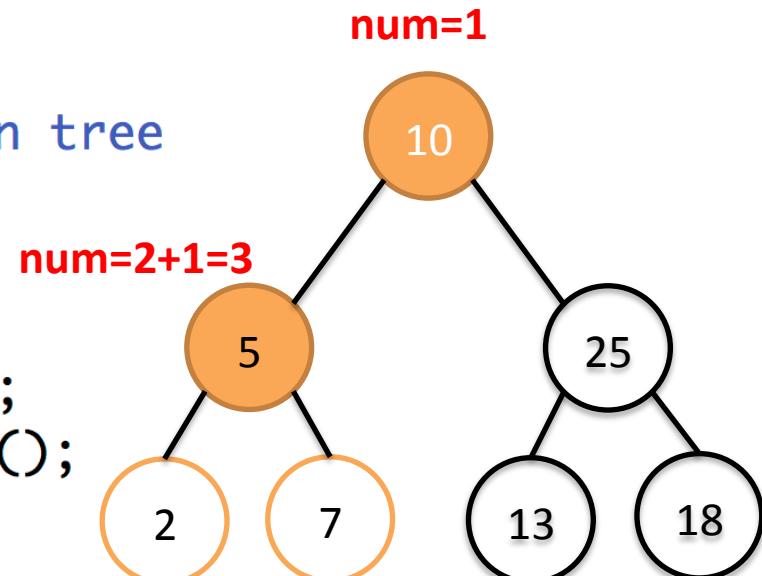


- No children
- Return 1 back to node 5

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

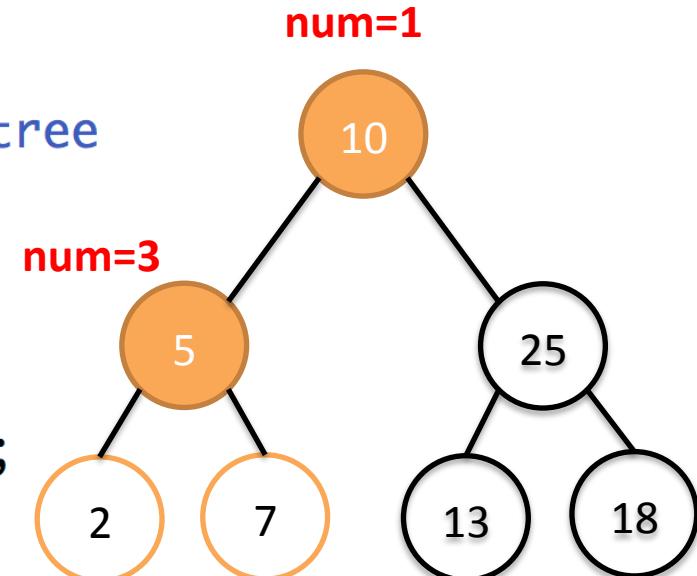
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    10 if (hasLeft()) num += left.size();  
    5 if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    10 if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    5 return num;  
}
```

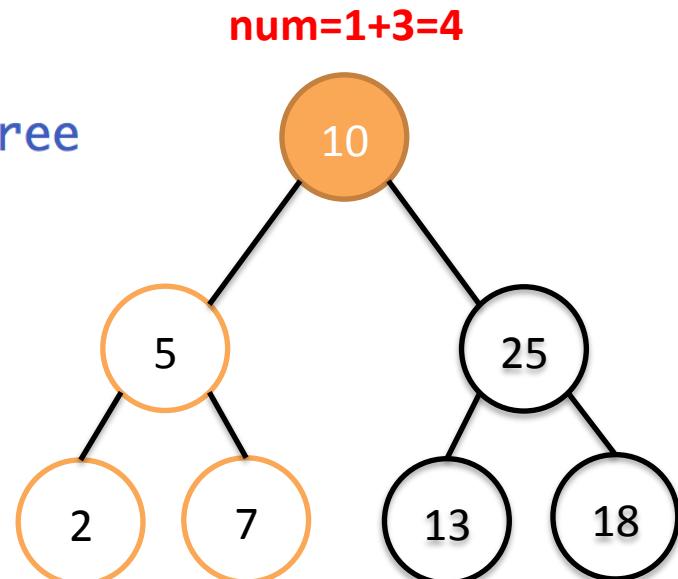


- Node 5 is done
- Return 3 to root

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    10 if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```

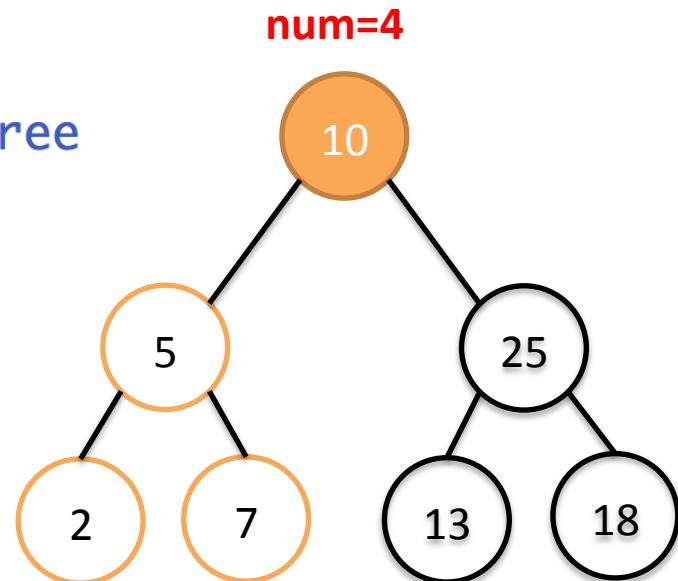


- Increment num on root

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
10    if (hasRight()) num += right.size();  
    return num;  
}
```

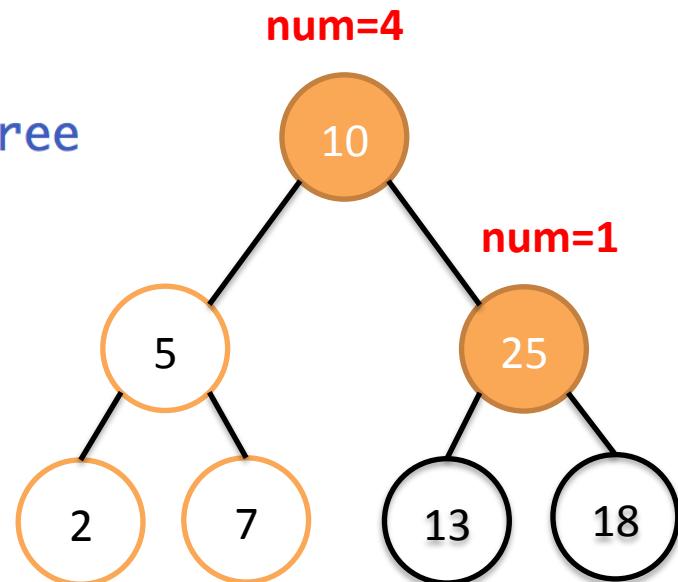


- Has right child
- Make recursive call on right child

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

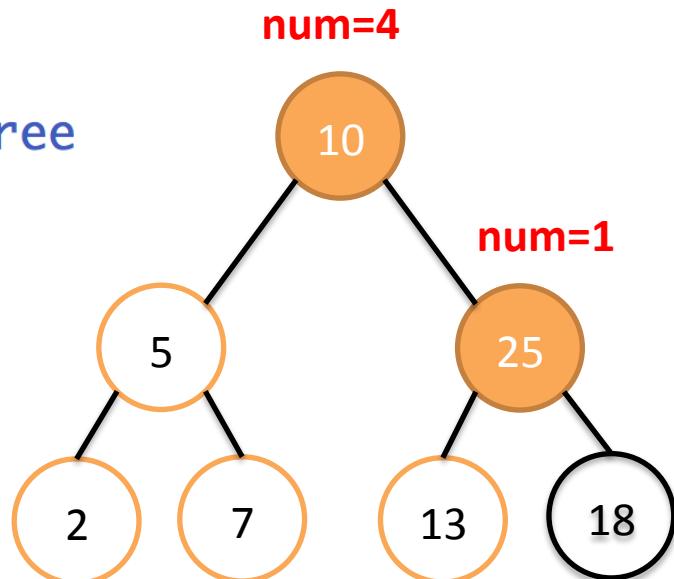
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    25 int num = 1;  
    if (hasLeft()) num += left.size();  
    10 if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```

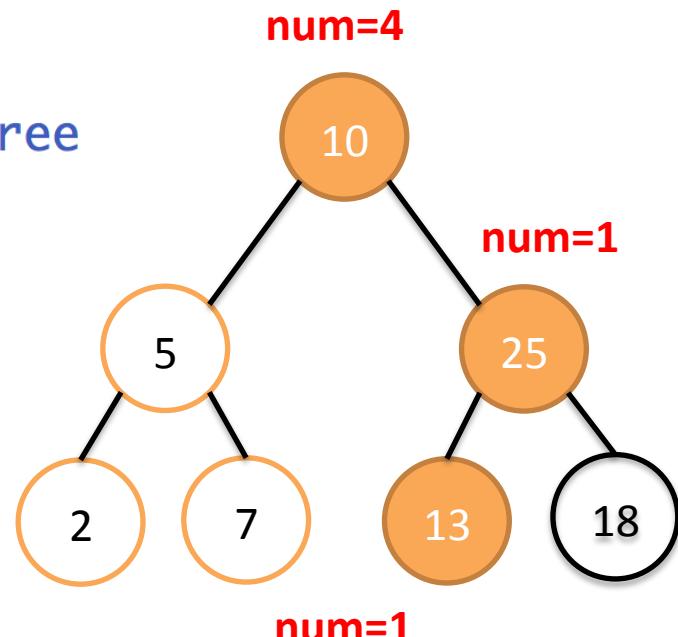


- Has left child
- Make recursive call on left child

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

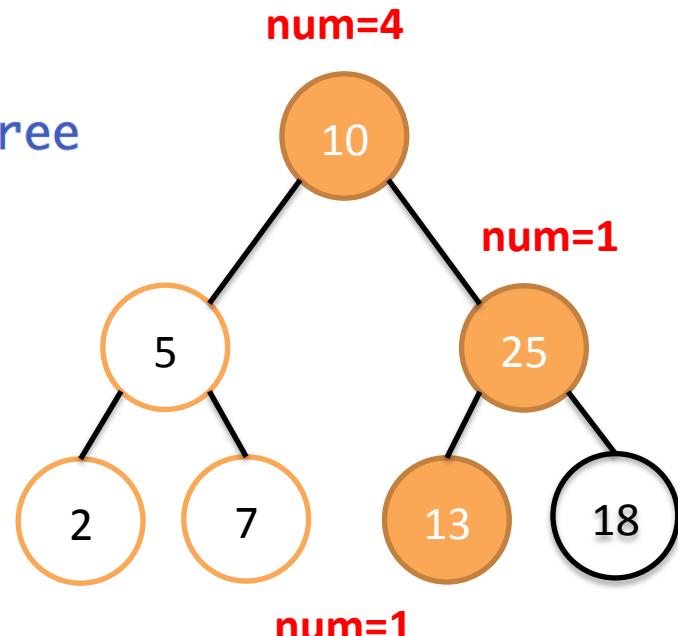
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```

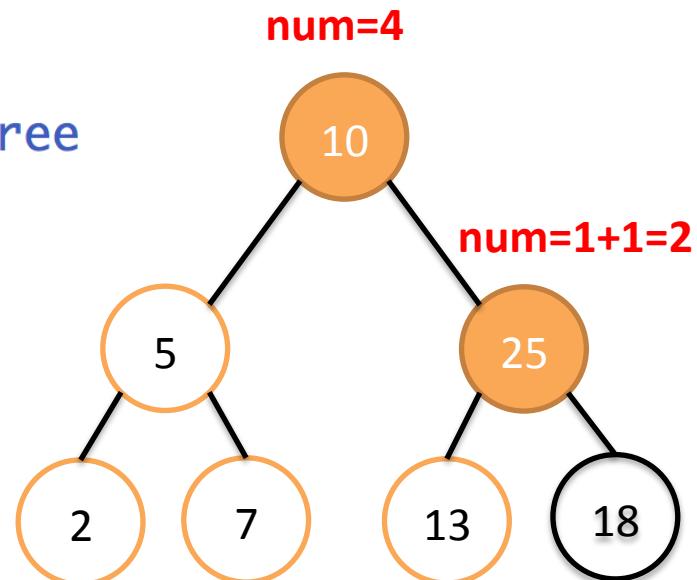


- No children
- Return 1 back to Node 25

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

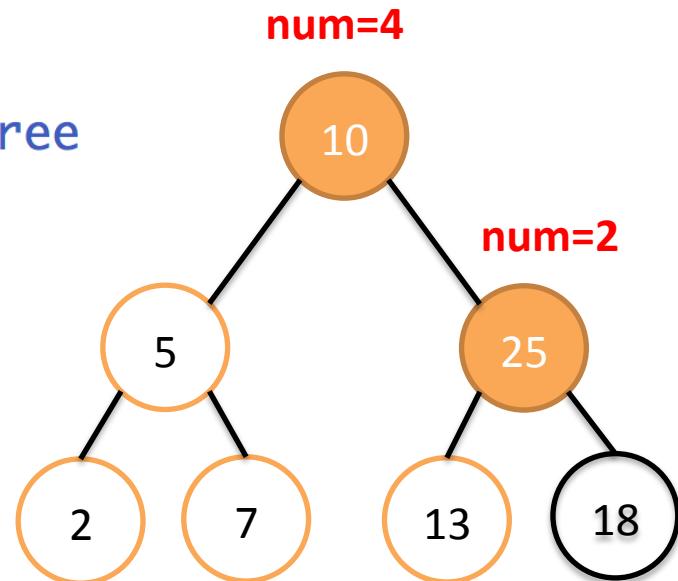
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



- Increment num on Node 25

Use recursion to calculate tree size from any given node = size of both children +1

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
25    if (hasRight()) num += right.size();  
    return num;  
}
```

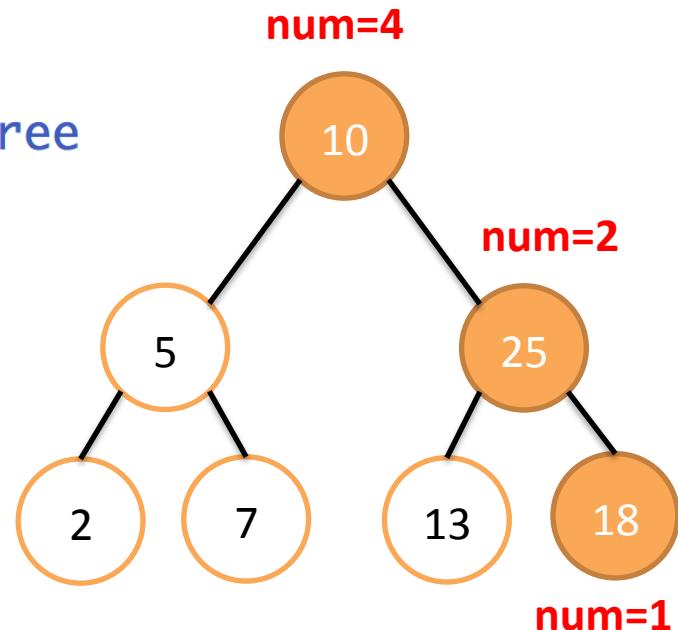


- Has right child
- Make recursive call on right child

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

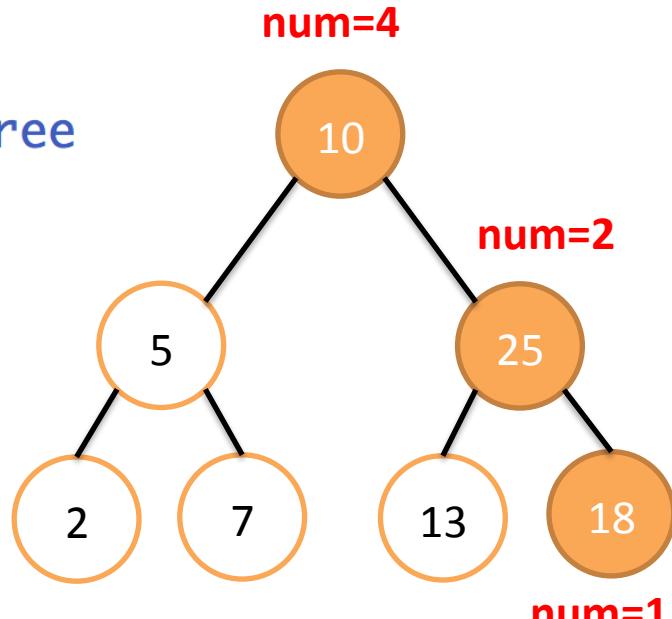
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    18 → int num = 1;  
    if (hasLeft()) num += left.size();  
    25 → if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



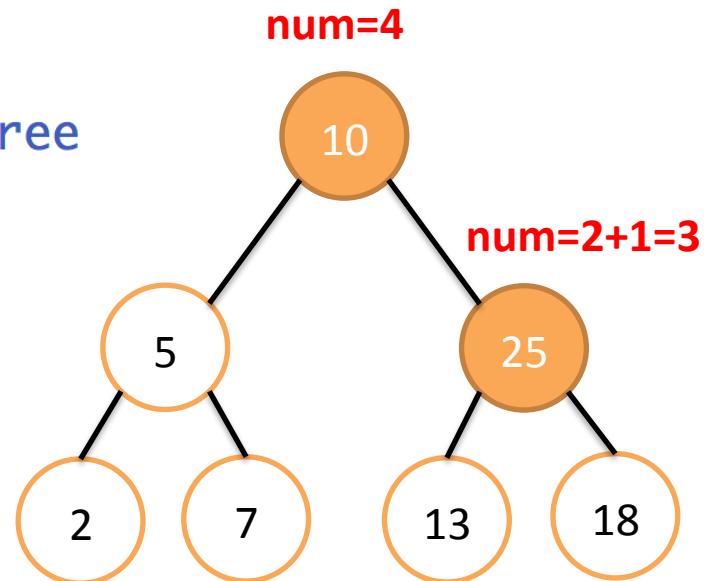
25
18
18
25
10
5
2
7
13
18
num=4
num=2
num=1

- No children
- Return 1 to Node 25

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
25    if (hasRight()) num += right.size();  
    return num;  
}
```

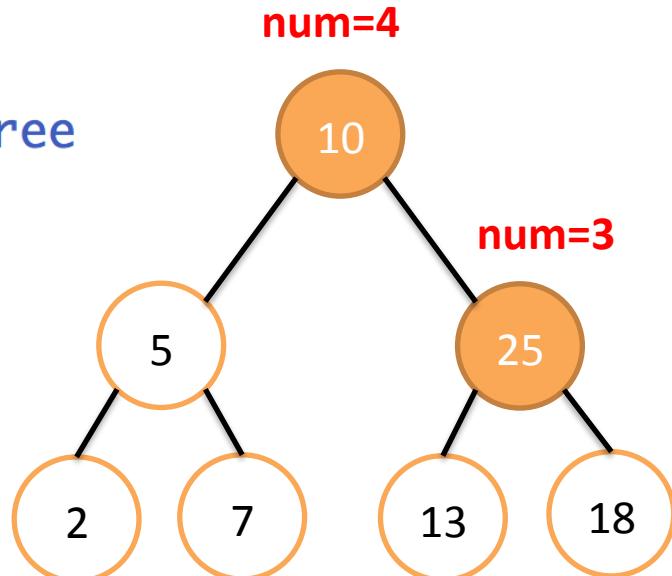


- Increment num on Node 25

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



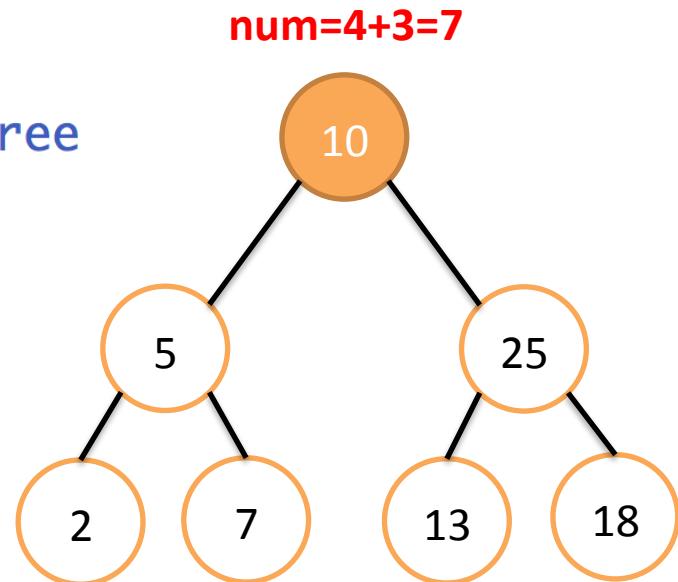
10 → 25

- Node 25 is done
- Return 3 back to root

Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

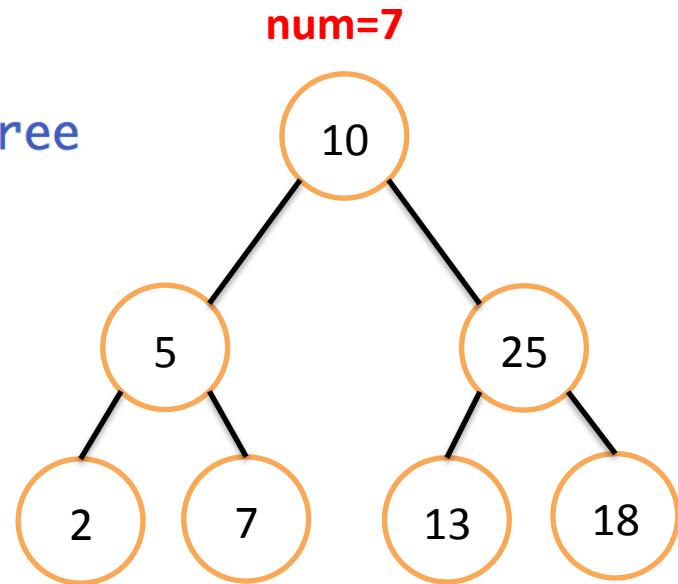
```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
10    if (hasRight()) num += right.size();  
    return num;  
}
```



Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```
/**  
 * Number of nodes (inner and leaf) in tree  
 */  
public int size() {  
    int num = 1;  
    if (hasLeft()) num += left.size();  
    if (hasRight()) num += right.size();  
    return num;  
}
```



Done!
Return 7

height() uses a similar recursive strategy to calculate the longest path to a leaf

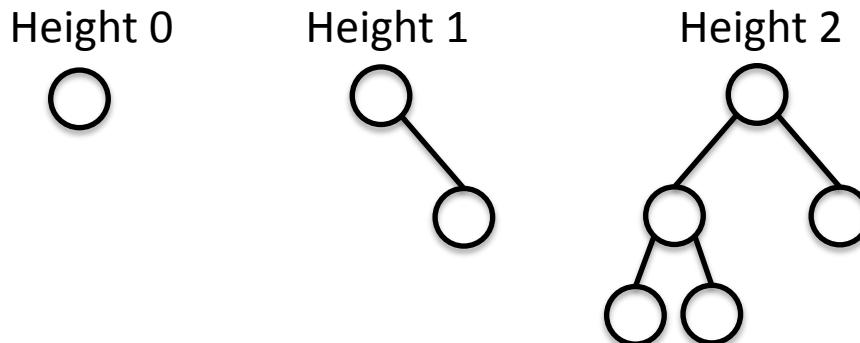
BinaryTree.java

```
...
86     * Longest length to a leaf node from here
87     */
88     public int height() {
89         if (isLeaf()) return 0;
90         int h = 0;
91         if (hasLeft()) h = Math.max(h, left.height());
92         if (hasRight()) h = Math.max(h, right.height());
93         return h+1;
94     }
95 }
```

- Height is the number of edges on the longest path from root to leaf
- By convention, a tree with one node (a leaf by definition) has height 0
- Recursively compute the height on the left and right child
- Keep the max

// inner: one higher than highest child

- Add one for this node
- This node isn't a leaf because if it was it would have returned zero in line 89



equalsTree() uses recursion to see if two trees have same data and structure

BinaryTree.java

```
96  /**
97  * Same structure and data?
98 */
99 public boolean equalsTree(BinaryTree<E> t2) {
100    if (hasLeft() != t2.hasLeft() || hasRight() != t2.hasRight()) return false;
101    if (!data.equals(t2.data)) return false;
102    if (hasLeft() && !left.equalsTree(t2.left)) return false;
103    if (hasRight() && !right.equalsTree(t2.right)) return false;
104    return true;
105 }
```

To see if two trees are equal, can we just check if `tree1 == tree2`?

No, that would only check to see if they are at the same memory address

Instead we traverse the tree, comparing node by node with the tree passed in as a parameter

Right way to compare objects is the *equals()* method

Trees are equal if same shape and same data

First check if same number number of children

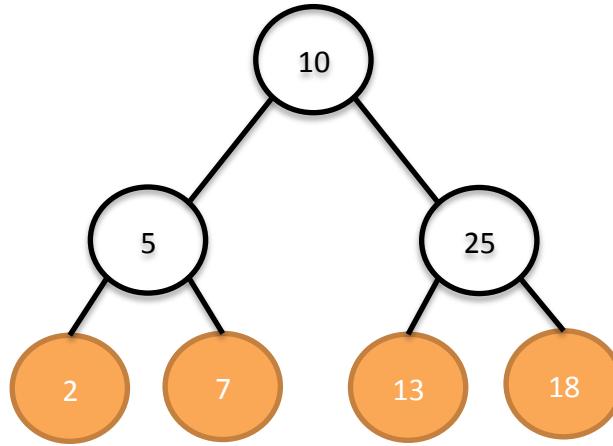
Next compare data is the same in each node

Finally, ask each child to compare itself

Agenda

1. General-purpose binary trees
2. Accumulators
3. Tree traversal
4. Expression trees

Accumulators are commonly used with trees for efficient operations



The fringe of a tree is the list of leaves in order from left to right

Here the fringe is [2, 7, 13, 18]

An efficient way to compute the fringe is to traverse the Tree and use an accumulator (course web page talks about an inefficient solution)

An accumulator keeps track of a variable during recursion

fringe() uses an accumulator pattern to get the leaves in order

BinaryTree.java

```
110  public ArrayList<E> fringe() {  
111      ArrayList<E> f = new ArrayList<E>();  
112      addToFringe(f); ←  
113      return f;  
114  }  
115  
116  /**  
117   * Helper for fringe, adding fringe data to the list  
118   */  
119  private void addToFringe(ArrayList<E> fringe) {  
120      if (isLeaf()) {  
121          fringe.add(data); ←  
122      }  
123      else {  
124          if (hasLeft()) left.addToFringe(fringe);  
125          if (hasRight()) right.addToFringe(fringe);  
126      }  
127  }  
  
NOTE: addFringe() does not have a return value, it doesn't need one!
```

fringe() method creates a variable *f* that will be used to *accumulate* results of tree traversal

Here we create a new *ArrayList* *f* as the accumulator, then pass it to a helper function that does recursion

Helper function uses accumulator during recursion

Node data added to fringe if leaf

Descend recursively

fringe() uses an accumulator pattern to get the leaves in order

BinaryTree.java

```
110  public ArrayList<E> fringe() {  
111      ArrayList<E> f = new ArrayList<E>();  
112      addToFringe(f);  
113      return f;  
114  }  
115  
116  /**  
117   * Helper for fringe, adding fringe data to the list as it goes  
118   */  
119  private void addToFringe(ArrayList<E> fringe) {  
120      if (isLeaf()) {  
121          fringe.add(data);  
122      }  
123      else {  
124          if (hasLeft()) left.addToFringe(fringe);  
125          if (hasRight()) right.addToFringe(fringe);  
126      }  
127  }
```

- Why use a helper method here?
- Why not just recursively call *fringe()*?
- Because we'd *new* an *ArrayList* at each recursive call
- Here we create a *new ArrayList* in *fringe()* and pass it to *addToFringe()*
- *addToFringe* updates *ArrayList*
- More notes on course web page

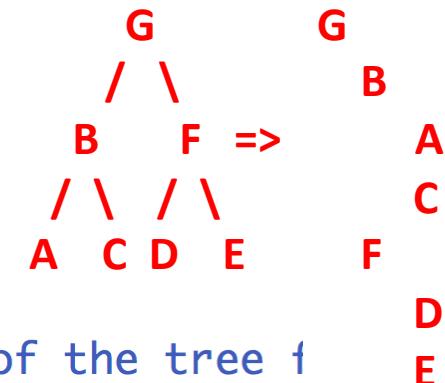
Similarly, *toString()* uses an accumulator to create a String representation of the tree

BinaryTree.java

```
129  * @param indent  
130  * Returns a string representation of the tree level  
131  */  
132  public String toString() {  
133      return toStringHelper("");  
134  }  
135  
136  *  
137  * Recursively constructs a String representation of the tree by  
138  * starting with the given indentation and indenting further going  
139  */  
140  public String toStringHelper(String indent) {  
141      String res = indent + data + "\n";  
142      if (hasLeft()) res += left.toStringHelper(indent+" ");  
143      if (hasRight()) res += right.toStringHelper(indent+" ");  
144      return res;  
145 }
```

Idea: keep an accumulator of how many spaces to indent

toString() called by Java if object is in `println`
Want to print Tree indented by level



Note: *toString()* doesn't take a parameter
How can we keep an accumulator?

Use a helper method!

Similarly, `toString()` uses an accumulator to create a String representation of the tree

BinaryTree.java

```
129  /**
130   * Returns a string representation of the tree
131  */
132  public String toString() {
133      return toStringHelper("");
134  }
135
136 /**
137  * Recursively constructs a String representation of the tree
138  * starting with the given indentation and indenting further going
139  */ Add indent spaces and data from this node to String
140  public String toStringHelper(String indent) { Helper function does recursion using indent variable
141      String res = indent + data + "\n";
142      if (hasLeft()) res += left.toStringHelper(indent+" ");
143      if (hasRight()) res += right.toStringHelper(indent+" ");
144      return res;
145 }
```

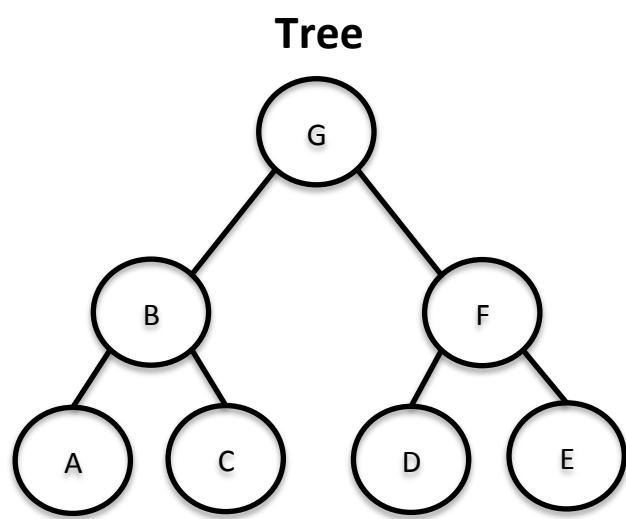
NOTE: "\n" means new line

***toString()* passes empty indent accumulator String to helper function**

***indent* will be the number of spaces before element so that String output looks like a tree (e.g., first level not indented, second level indented 2 spaces)**

Adds 2 extra spaces to indent every time go down a level in tree

Similarly, `toString()` uses an accumulator to create a String representation of the tree



Output of `System.out.println(tree)`

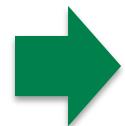
G
B
A
C
F
D
E

Each level in tree printed two spaces indented from parent level in tree

Each time `toString()` descended a level, it added two spaces to indent

Agenda

1. General-purpose binary trees
2. Accumulators
3. Tree traversal
4. Expression trees



There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

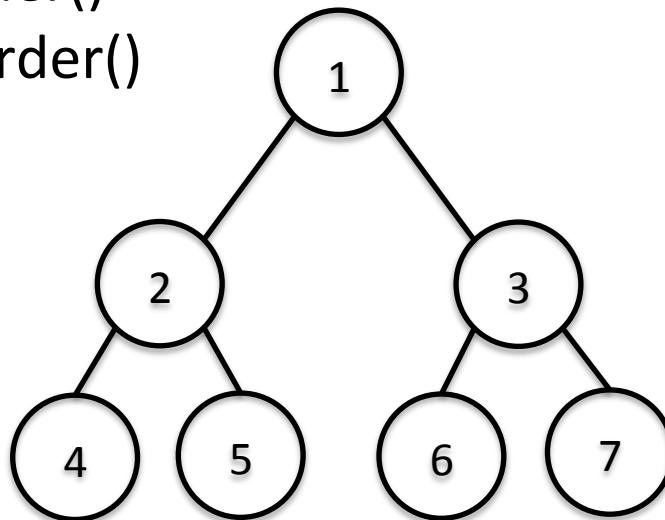
left.preorder()

right.preorder()

“do something”

means “handle
this node”, might
print it, might do
something else

Sometimes
referred to as
“visiting” the
node”



Examples:

File directory structure

Table of contents in book

toString()

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

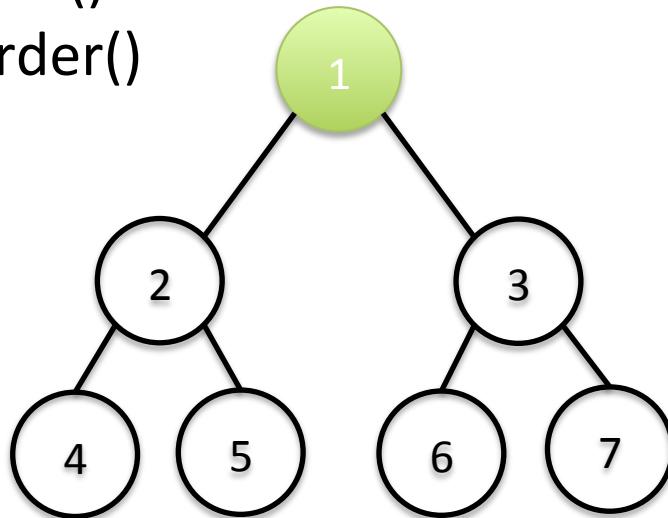
right.preorder()

Examples:

File directory structure

Table of contents in book

`toString()`



Visited

1

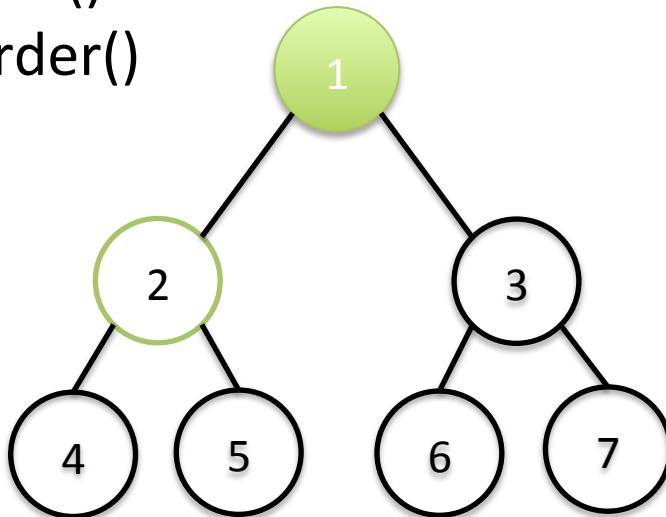
There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

right.preorder()



Visited

1

Examples:

File directory structure

Table of contents in book

`toString()`

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

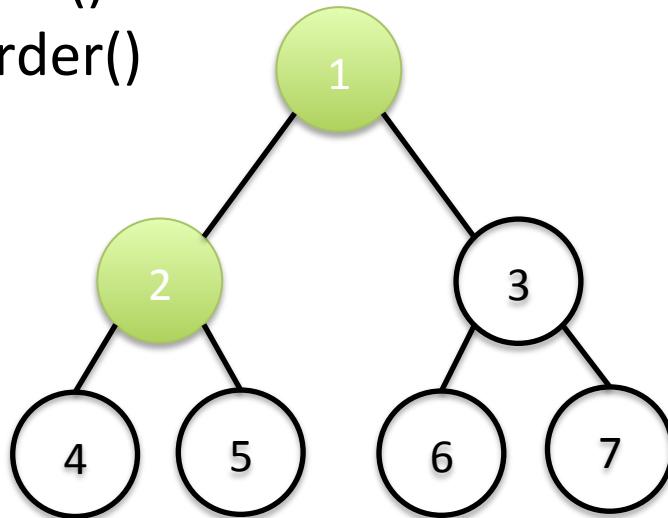
right.preorder()

Examples:

File directory structure

Table of contents in book

toString()



Visited

1, 2

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

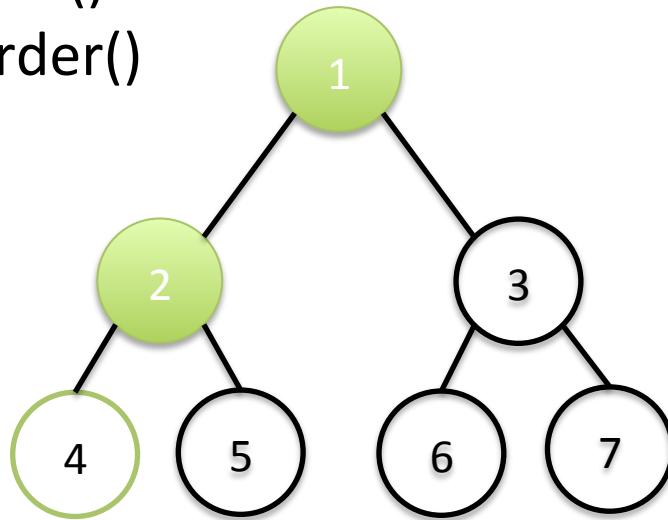
right.preorder()

Examples:

File directory structure

Table of contents in book

`toString()`



Visited

1, 2

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

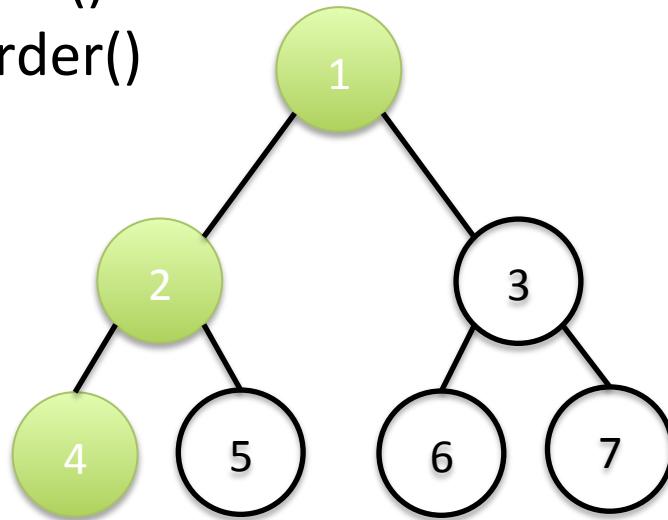
right.preorder()

Examples:

File directory structure

Table of contents in book

`toString()`



Visited

1, 2, 4

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

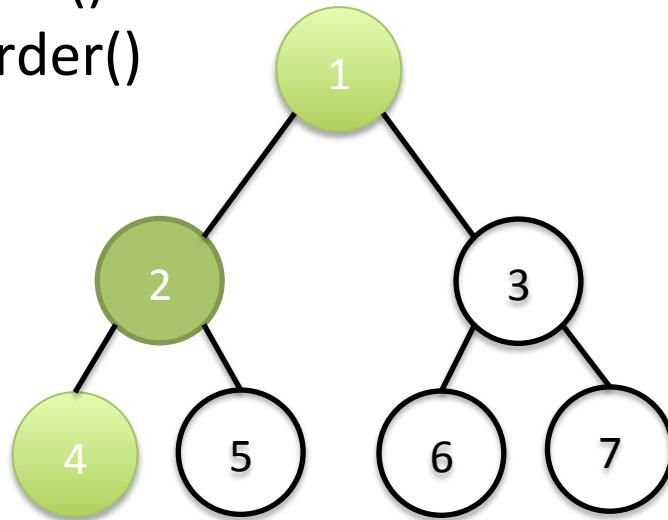
right.preorder()

Examples:

File directory structure

Table of contents in book

toString()



Visited

1, 2, 4

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

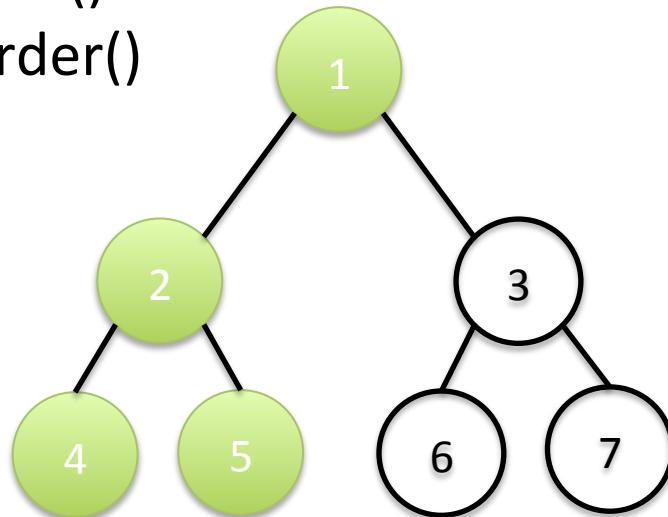
right.preorder()

Examples:

File directory structure

Table of contents in book

toString()



Visited

1, 2, 4, 5

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

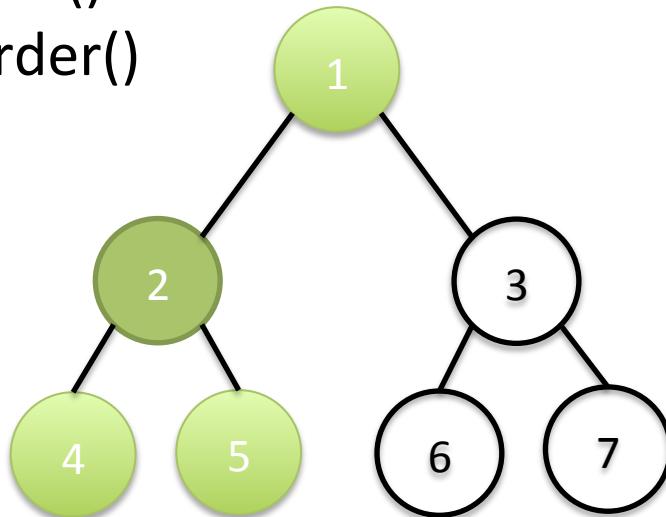
right.preorder()

Examples:

File directory structure

Table of contents in book

toString()



Visited

1, 2, 4, 5

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

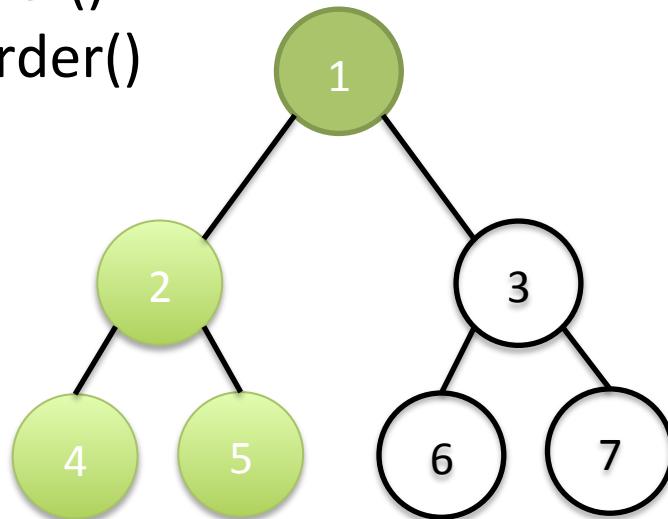
right.preorder()

Examples:

File directory structure

Table of contents in book

toString()



Visited

1, 2, 4, 5

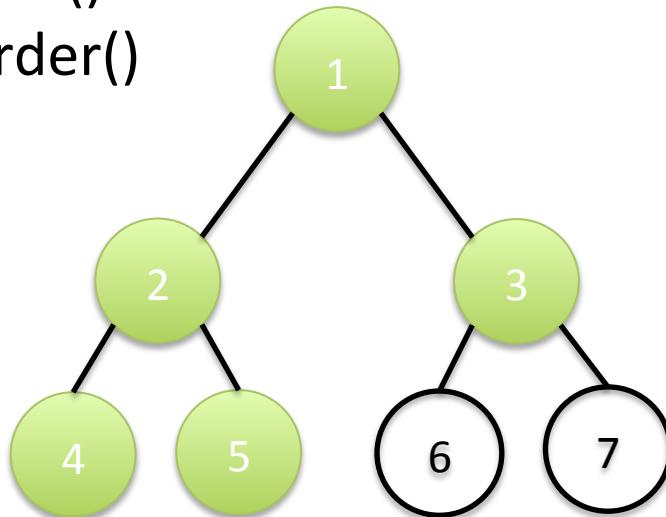
There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

right.preorder()



Examples:

File directory structure

Table of contents in book

`toString()`

Visited

1, 2, 4, 5, 3

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

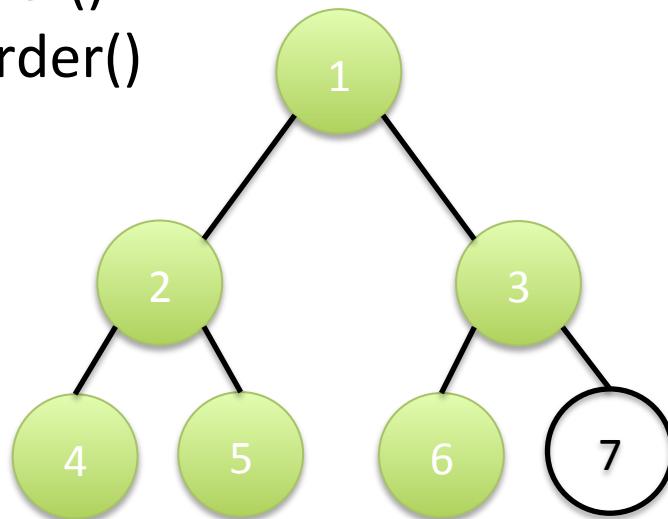
right.preorder()

Examples:

File directory structure

Table of contents in book

`toString()`



Visited

1, 2, 4, 5, 3, 6

There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

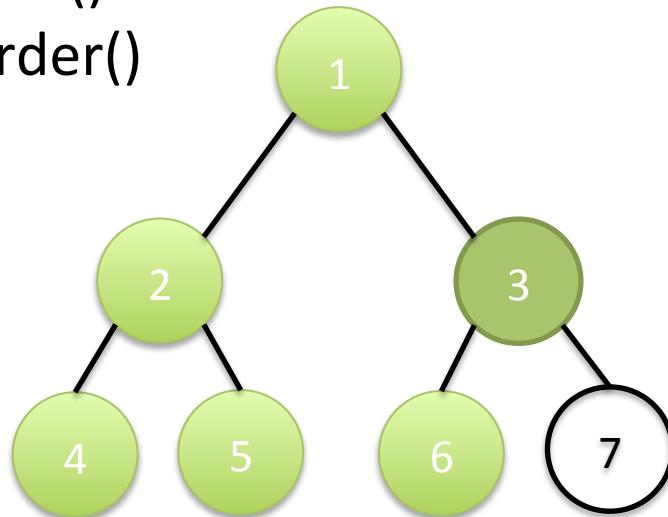
right.preorder()

Examples:

File directory structure

Table of contents in book

`toString()`



Visited

1, 2, 4, 5, 3, 6

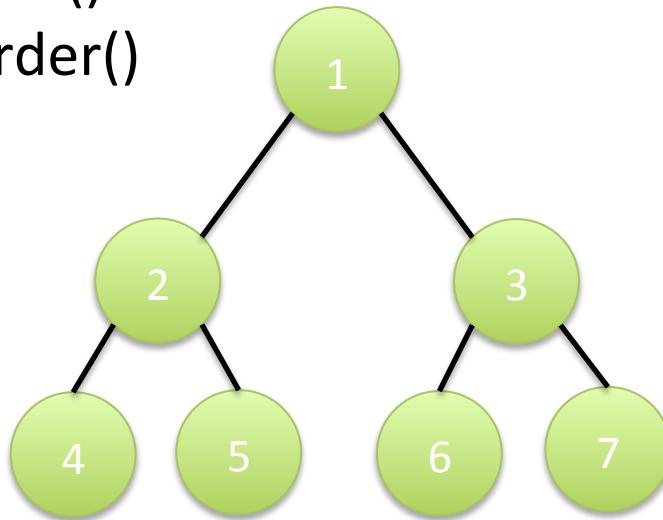
There are different ways to traverse a tree, depending on what needs to be done

preorder()

do something

left.preorder()

right.preorder()



Examples:

File directory structure

Table of contents in book

`toString()`

Visited

1, 2, 4, 5, 3, 6, 7

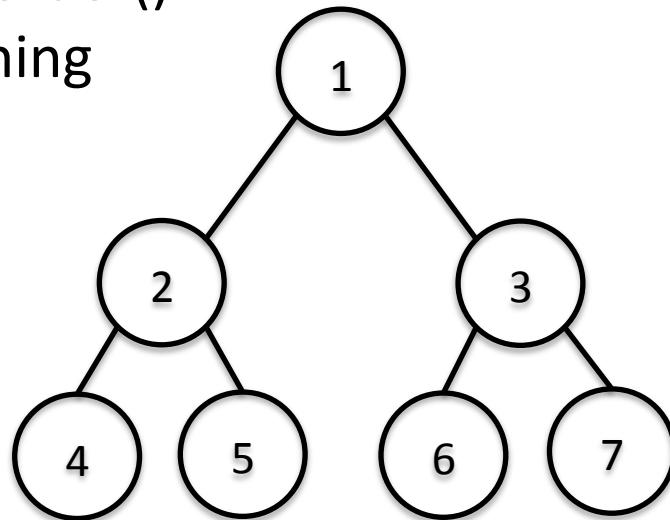
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

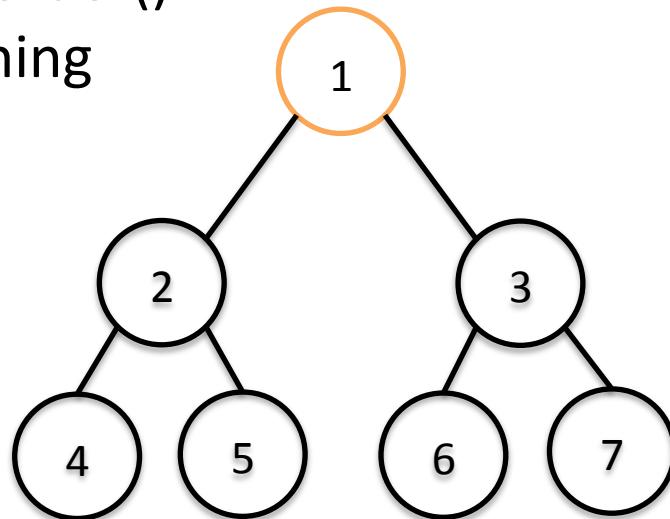
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

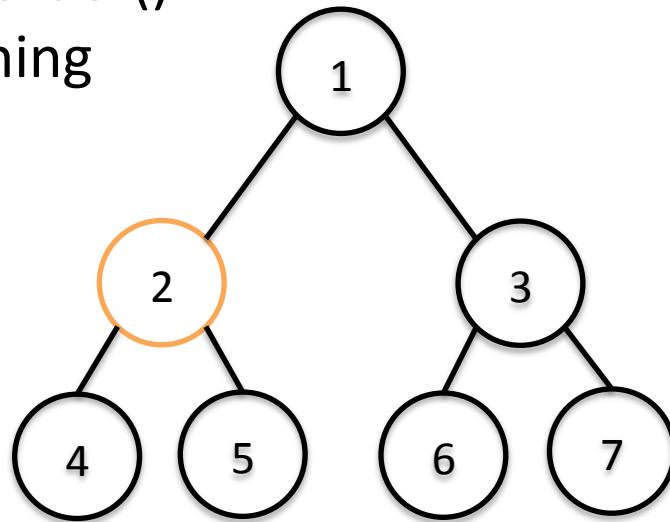
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

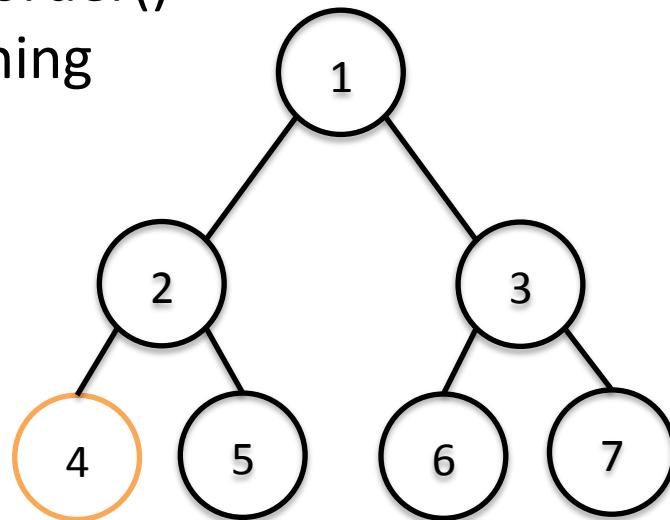
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

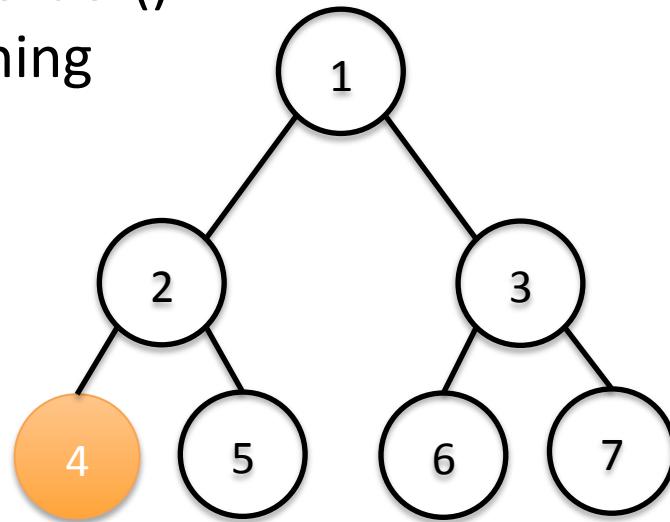
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4

Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

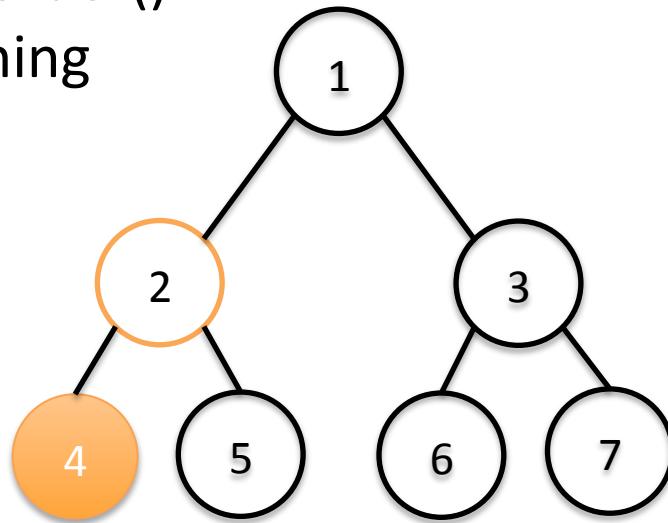
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4

Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

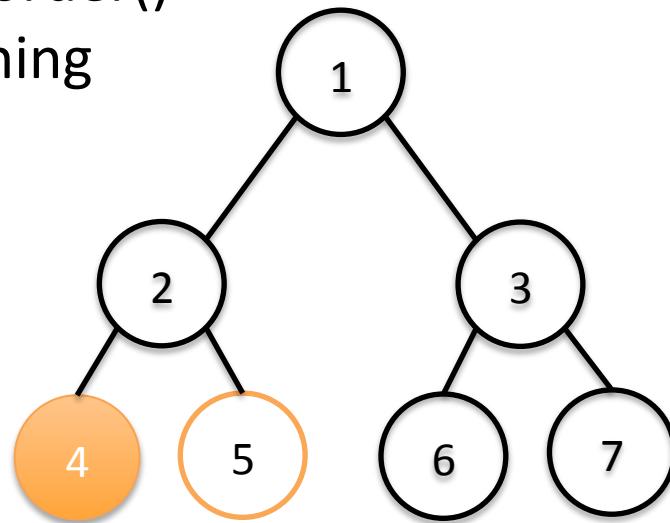
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4

Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

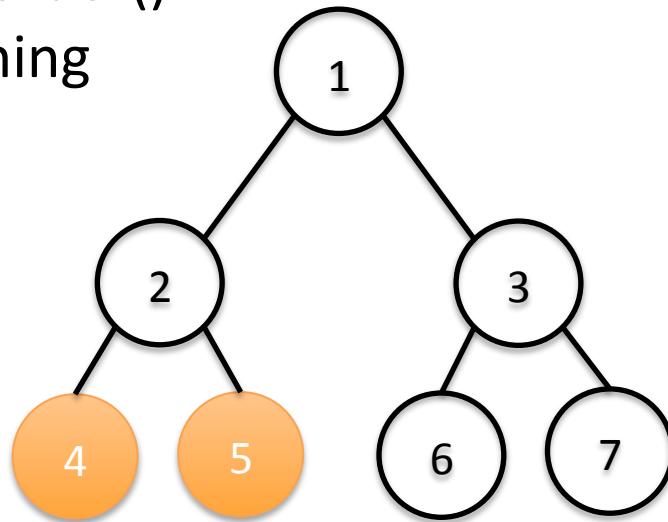
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5

Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

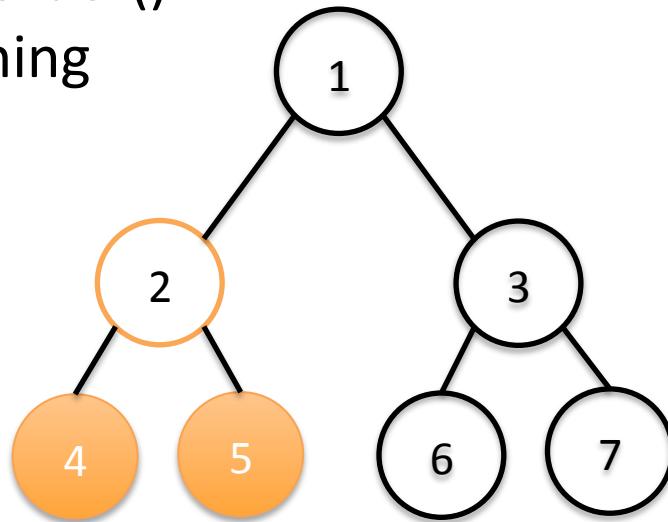
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5

Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

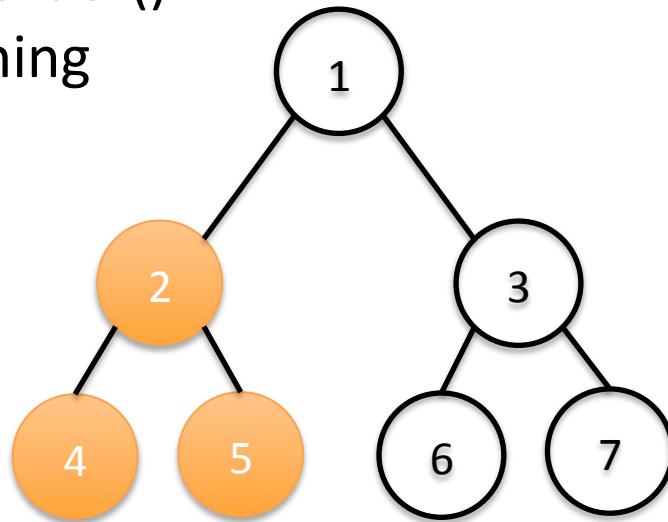
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5, 2

Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

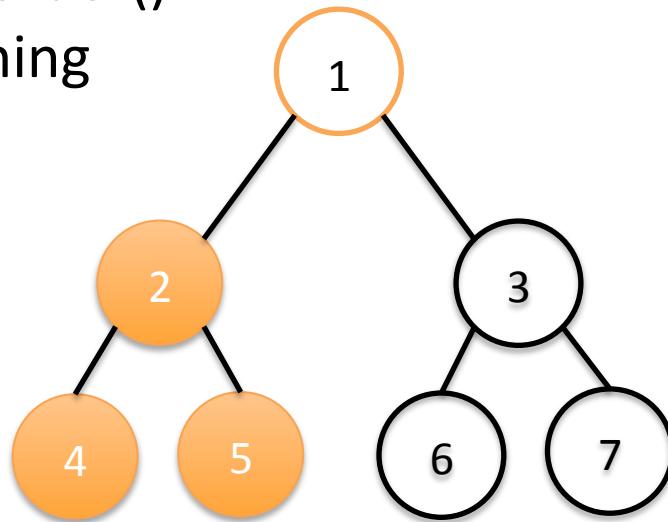
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5, 2

Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

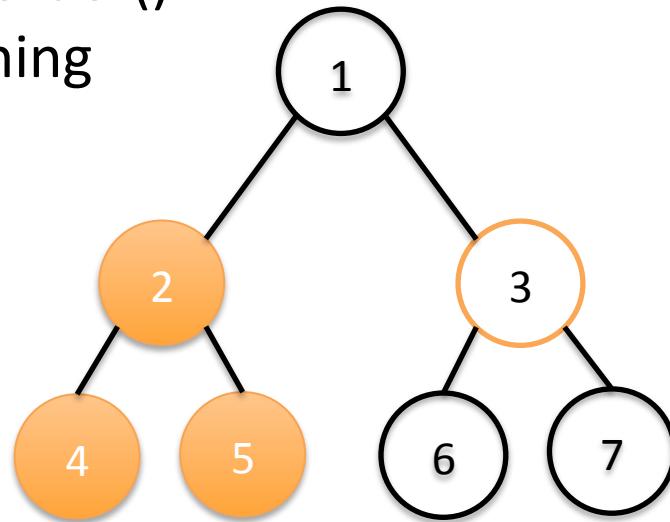
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5, 2

Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

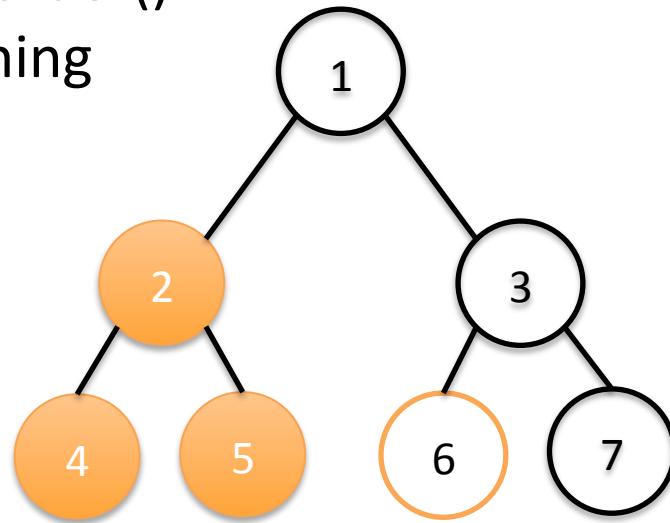
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5, 2

Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

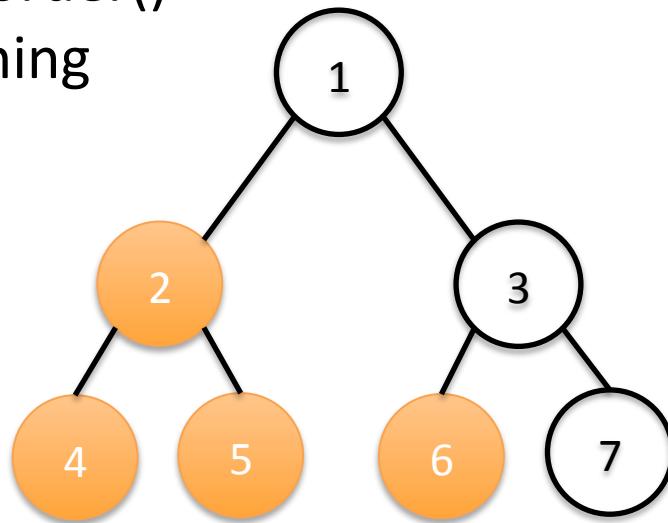
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

Visited

4, 5, 2, 6

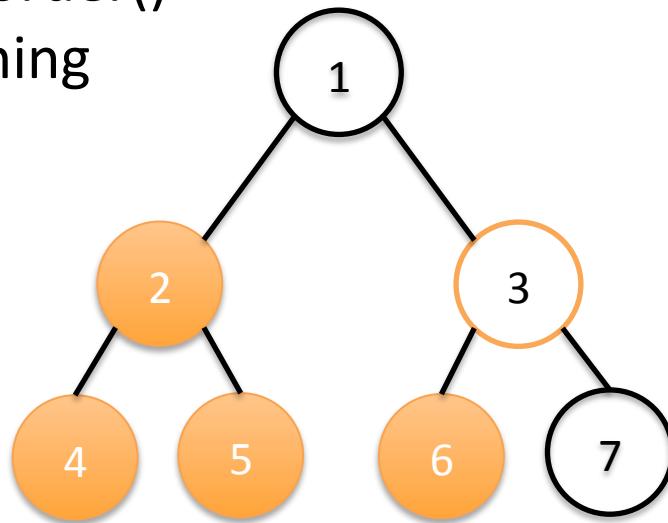
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5, 2, 6

Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

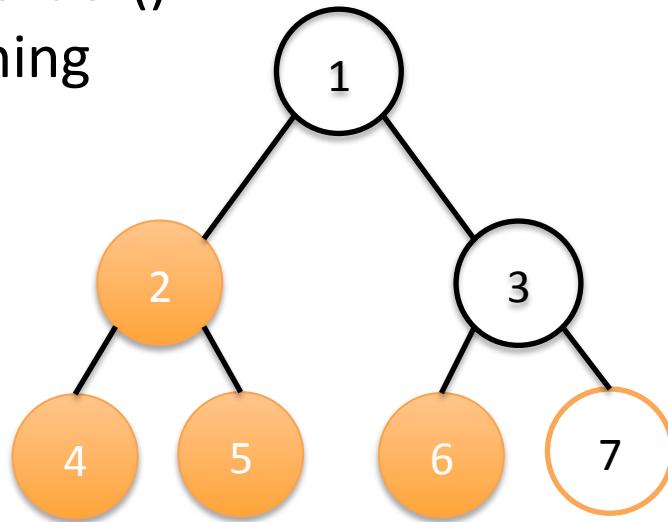
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Visited

4, 5, 2, 6

Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

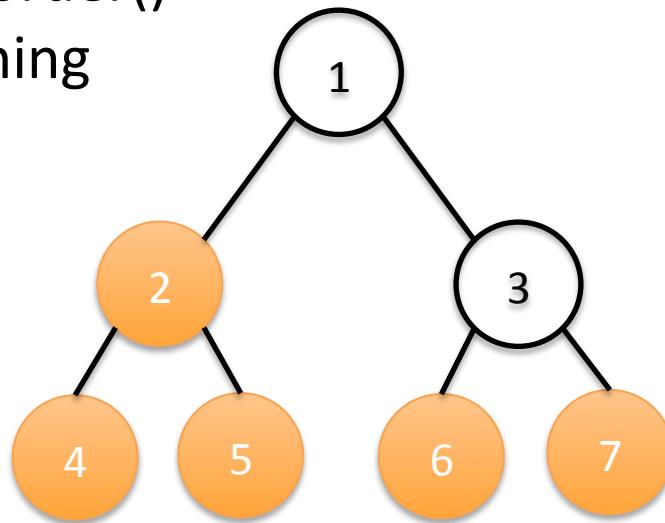
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

Visited

4, 5, 2, 6, 7

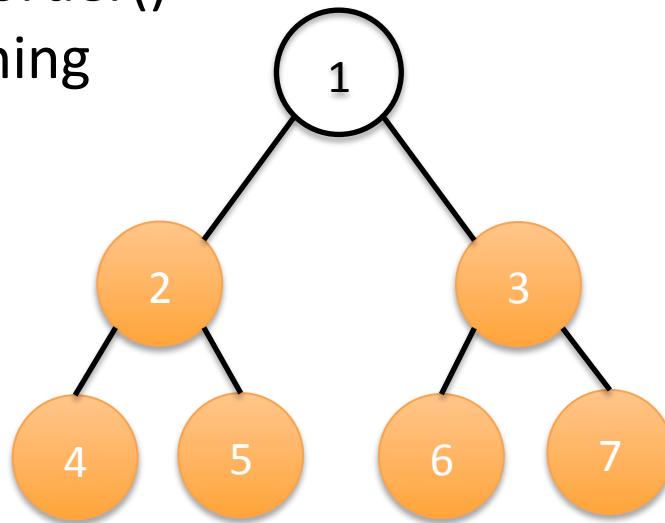
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not
sure how many bytes in
each directory until you
search all children)

Visited

4, 5, 2, 6, 7, 3

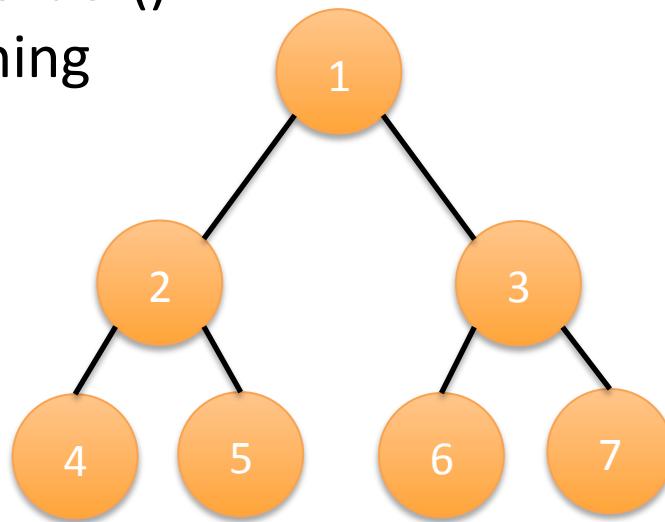
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

do something



Example:

Compute disk space (not sure how many bytes in each directory until you search all children)

Visited

4, 5, 2, 6, 7, 3, 1

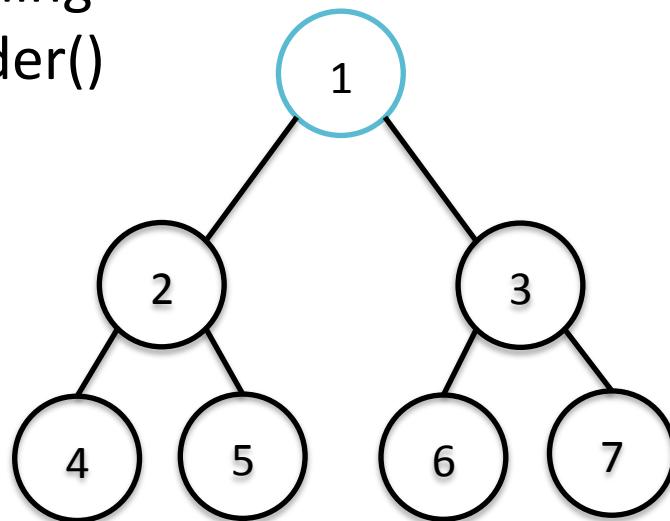
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



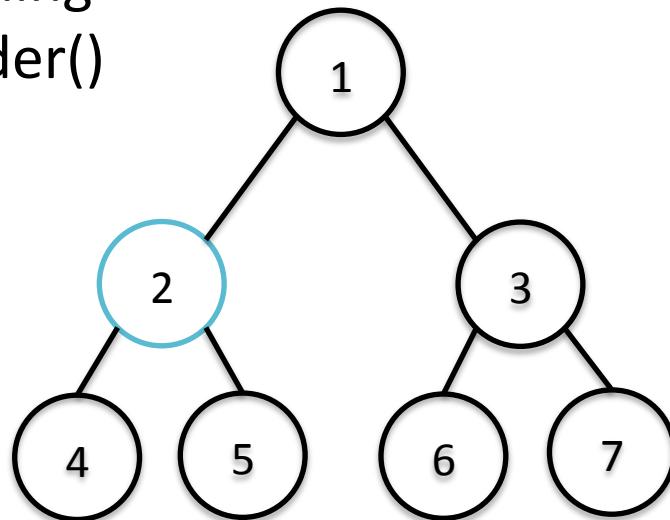
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



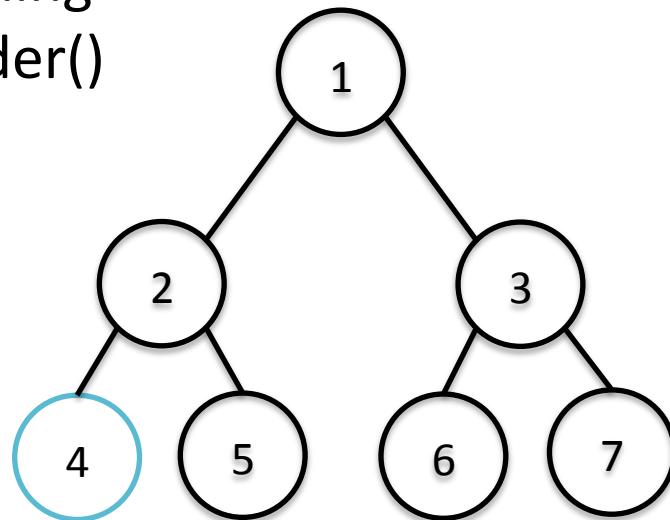
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



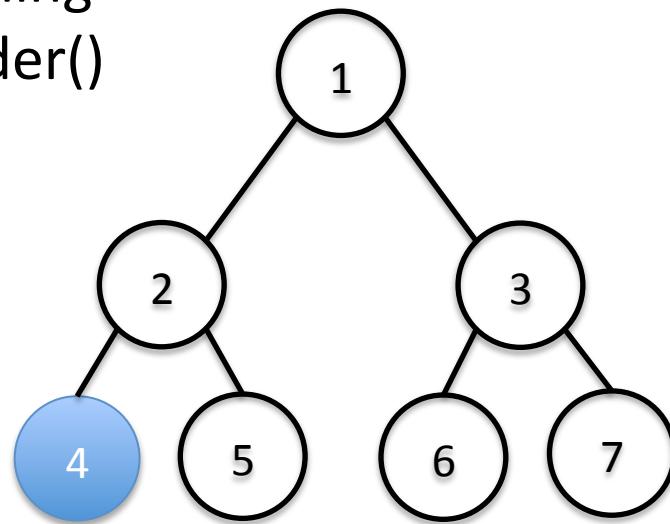
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4

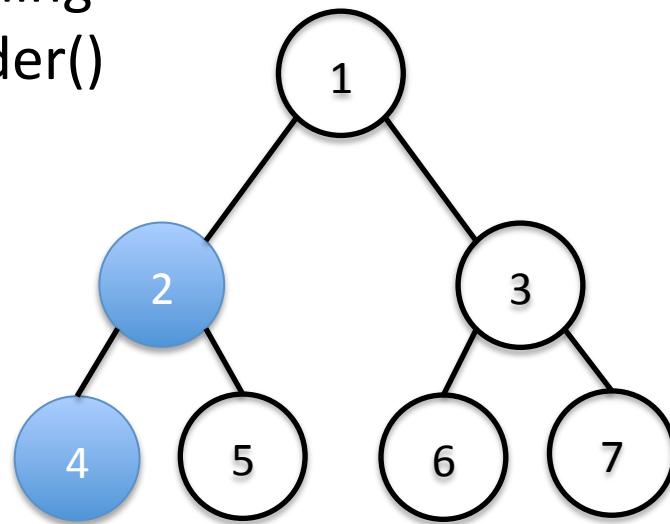
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2

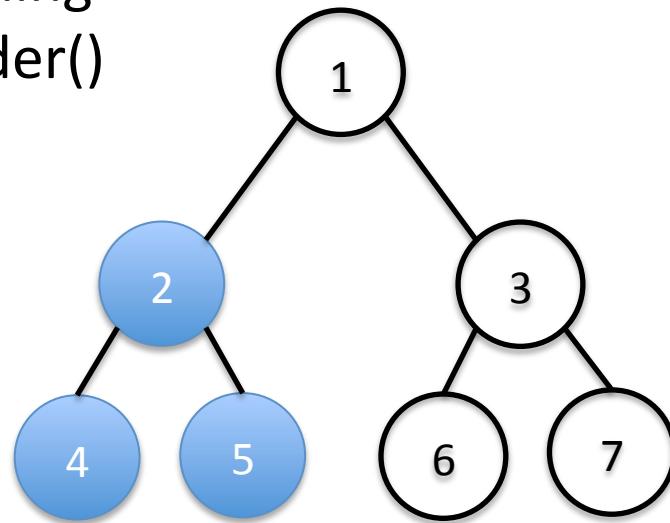
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5

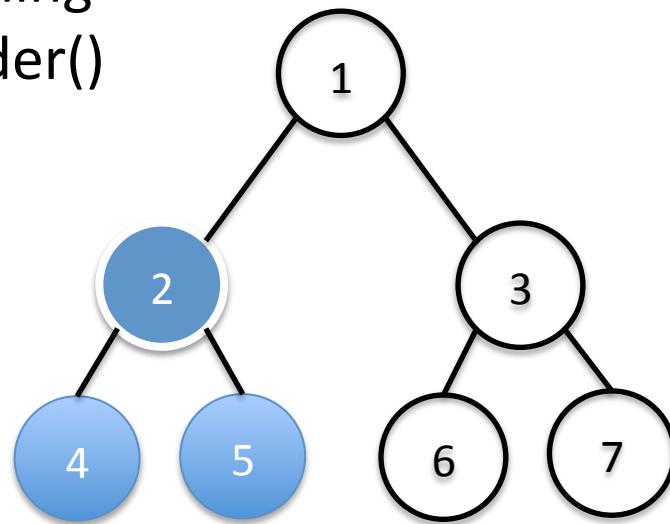
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5

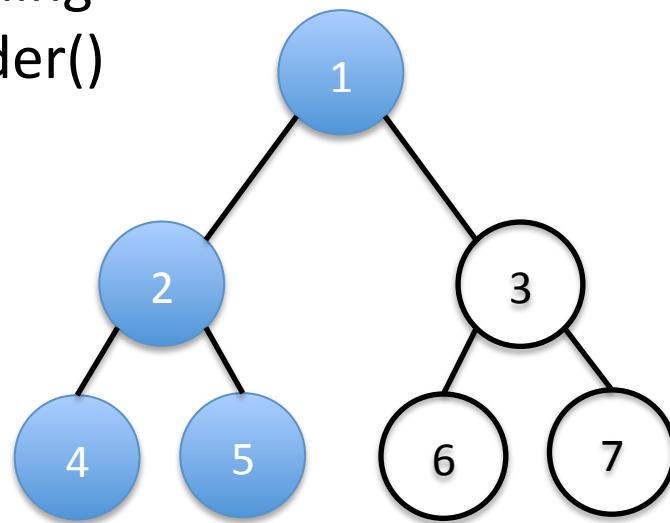
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1

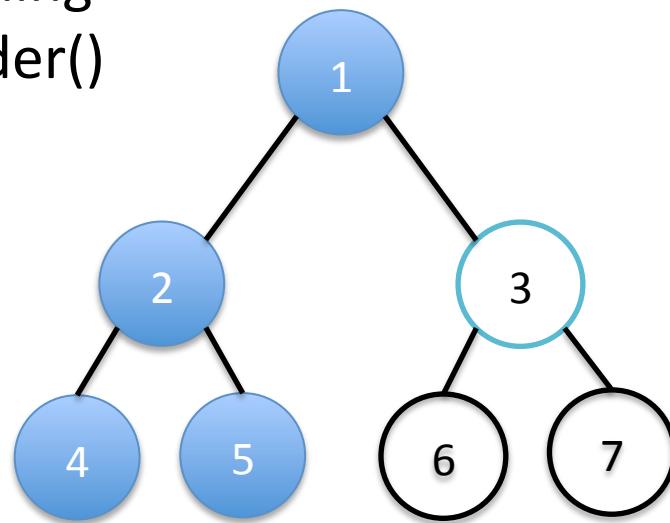
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1

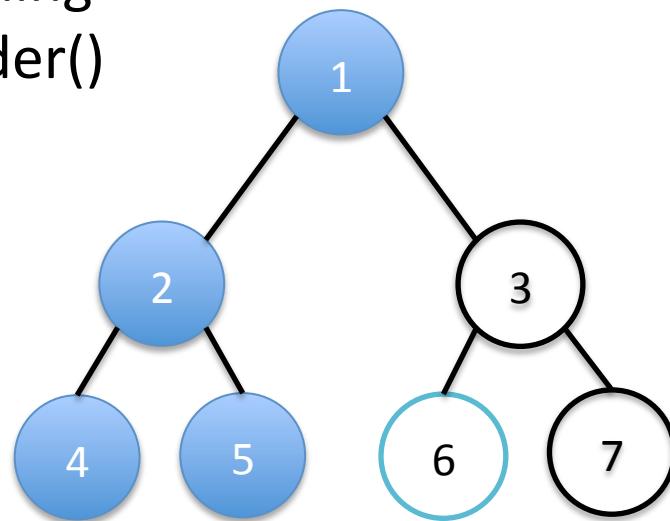
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1

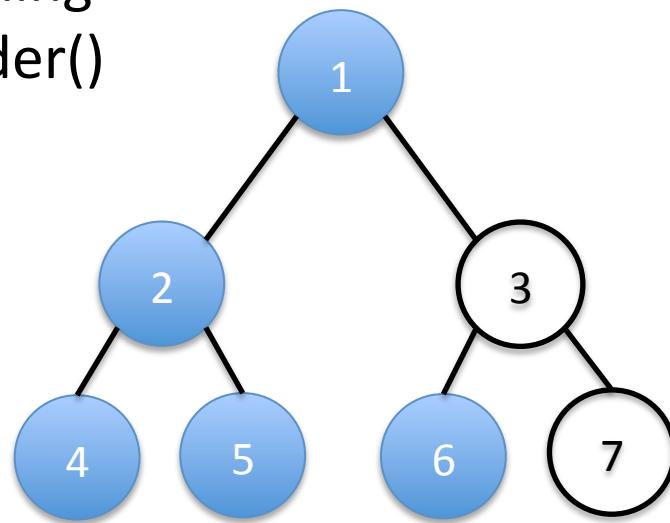
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1, 6

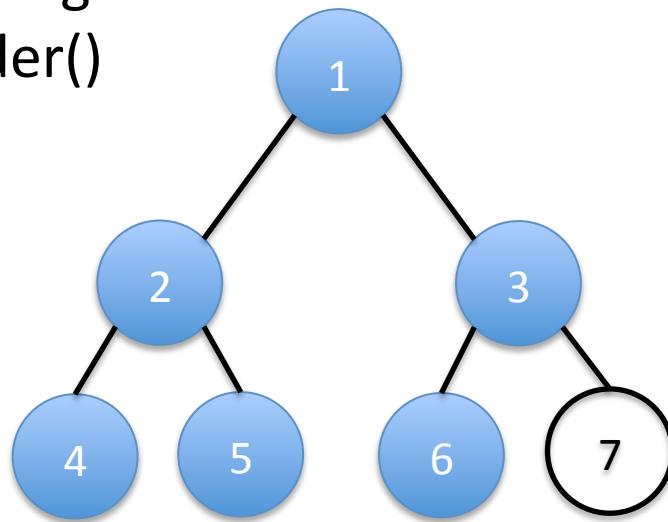
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
do something
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1, 6, 3

There are different ways to traverse a tree, depending on what needs to be done

inorder()

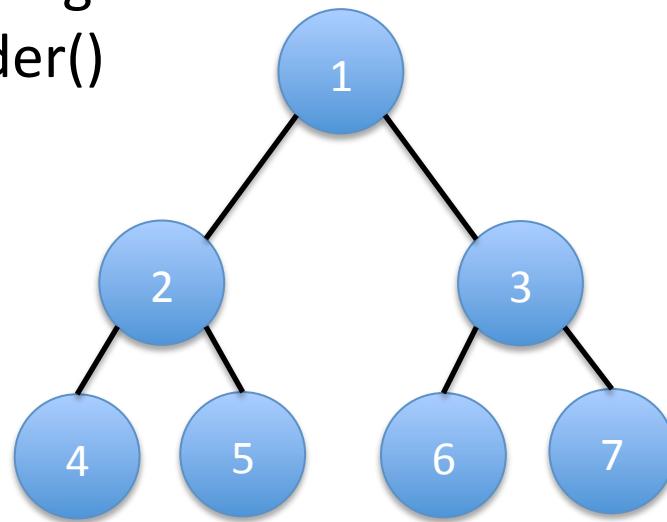
left.inorder()

do something

right.inorder()

Example:

Drawing a tree

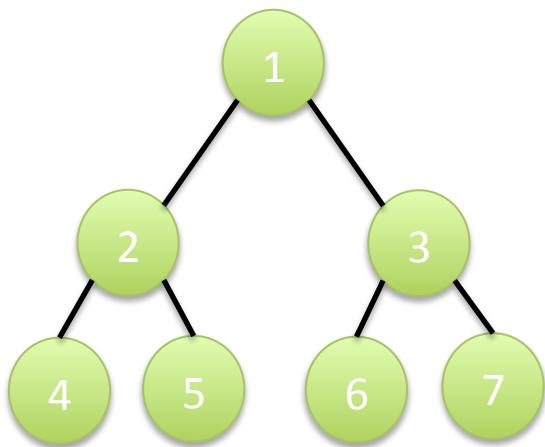


Visited

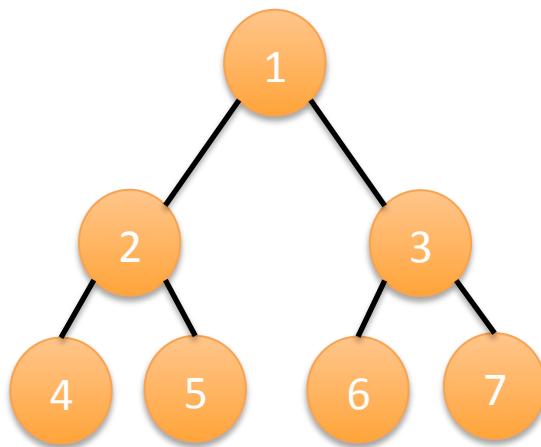
4, 2, 5, 1, 6, 3, 7

Summary

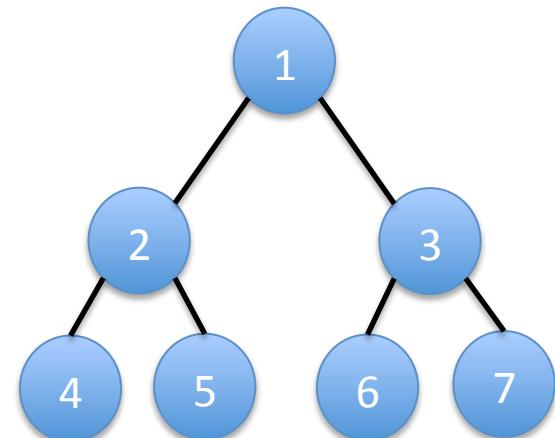
Preorder



Postorder



Inorder



Visited

1, 2, 4, 5, 3, 6, 7

Book chapters
toString()

Visited

4, 5, 2, 6, 7, 3, 1

Calculate disk space

Visited

4, 2, 5, 1, 6, 3, 7

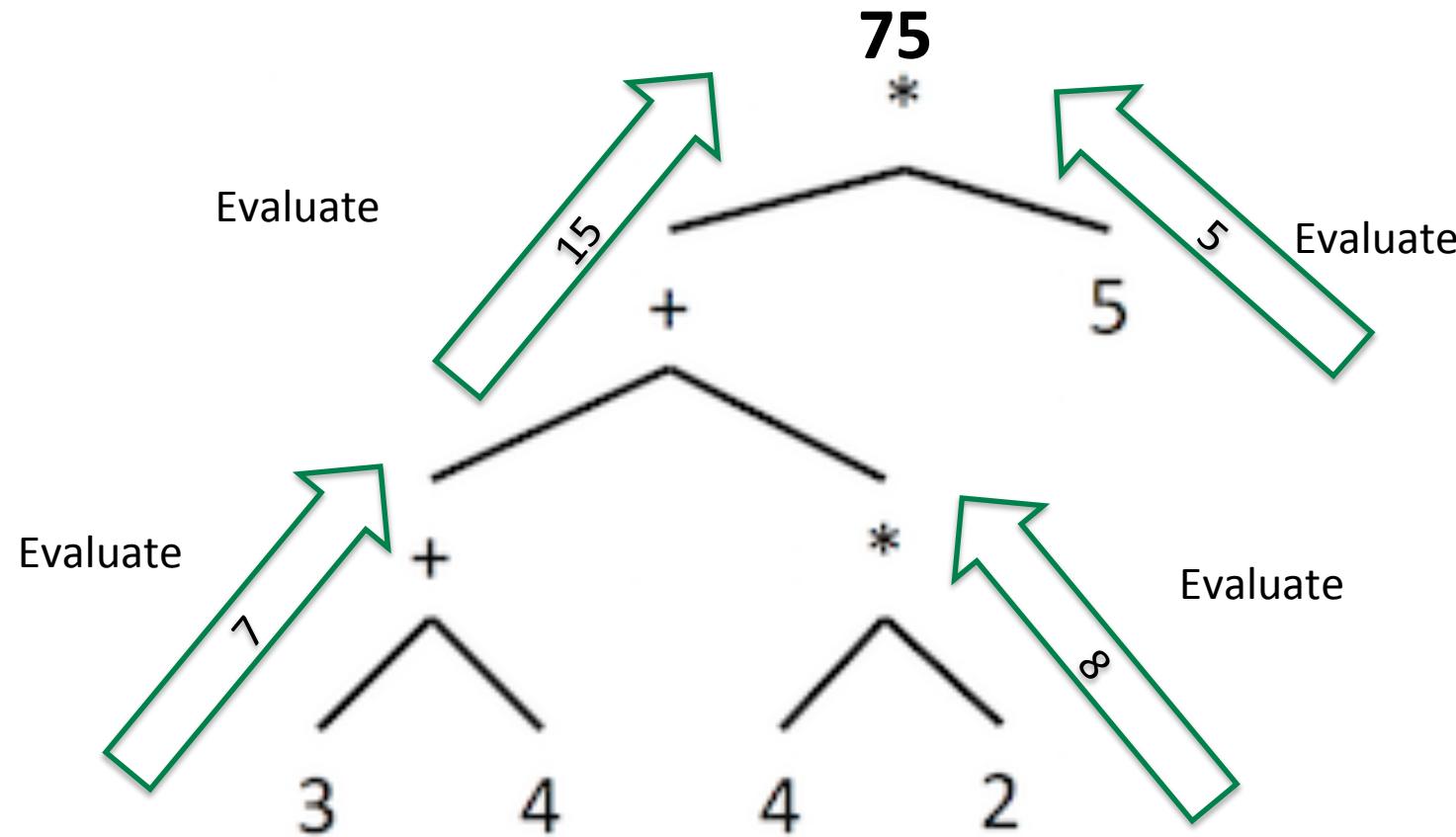
Drawing a tree
(left to right)

Agenda

1. General-purpose binary trees
2. Accumulators
3. Tree traversal
4. Expression trees

Expression trees ask their children to evaluate themselves

Expression tree for $((3+4) + (4*2)) * 5$



Expression Tree builds Expression nodes and evaluates them

Expression.java

- Interface, specifying that nodes must provide evaluate method

Number.java

- Implements Expression interface
- Holds double
- Returns that value on *evaluate()*

Addition.java

- Implements Expression interface
- Holds left and right Expression
- Returns left + right on *evaluate()*

Multiplication

- Implements Expression interface
- Holds left and right Expression
- Returns left * right on *evaluate()*

Expression Tree builds Expression nodes and evaluates them

ExpressionTest.java

- Creates Number and evaluates
- Creates $3 + 4$ calling new Addition and passing two Numbers
- Creates test by calling helper function below (for readability)
 - *c()* creates new Number
 - *plus (e1,e2)* creates new Addition
 - *times(e1,e2)* creates new Multiplication

