CS 10: Problem solving via Object Oriented Programming

Hierarchies – Binary trees

Main goals

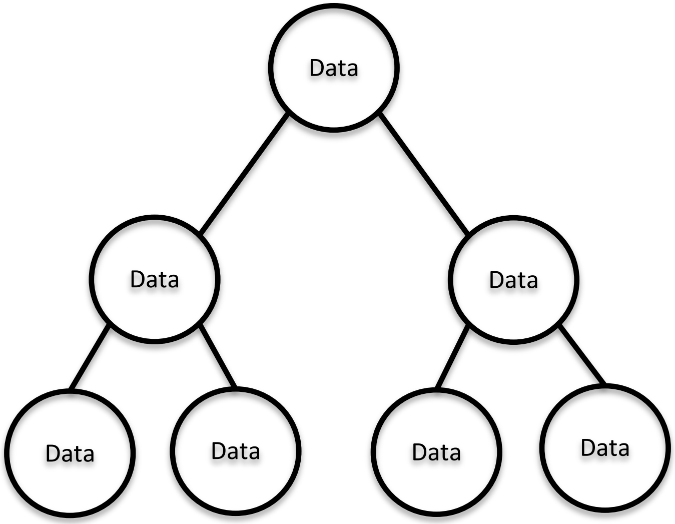
- Implement hierarchical data representation: binary trees
  - Implement methods using recursion
  - Implement methods using accumulators
  - Identify order of traversal

Agenda

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

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

Tree data structure

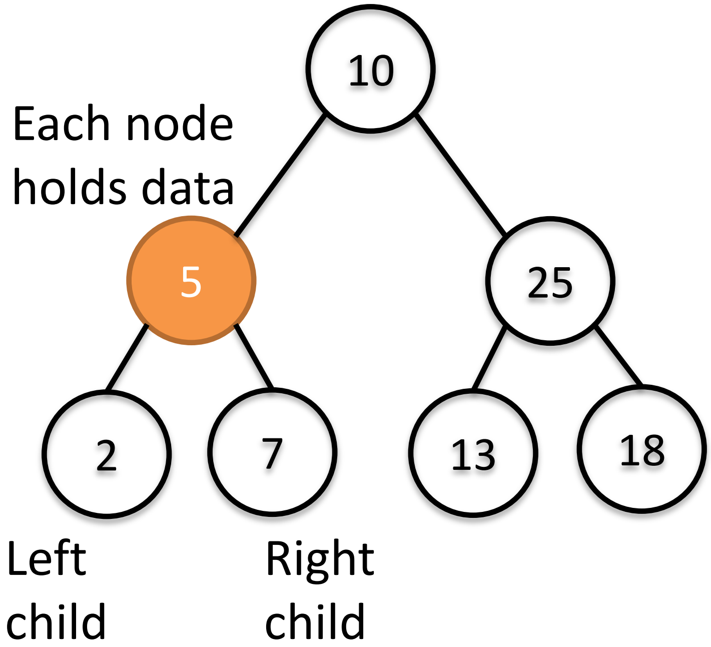


Difference with singly linked list?

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

Each node in a tree can be thought of as the head of its own subtree

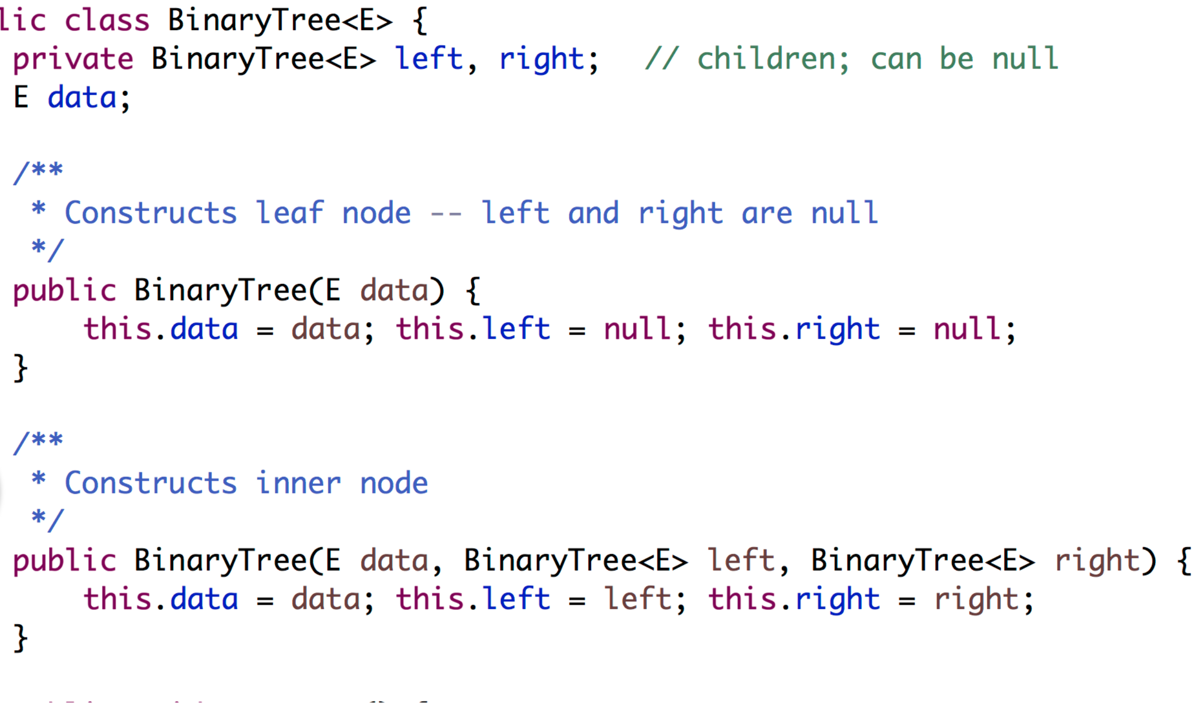
BinaryTree.java



Each node holds data

Left child

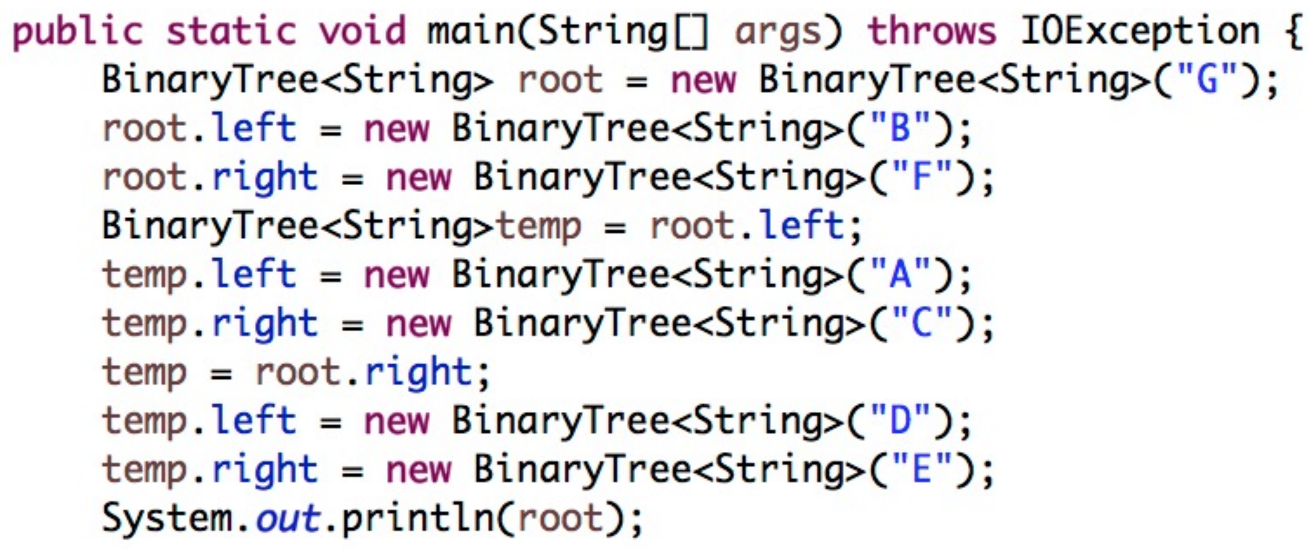
Right child



```
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;
    }
```

Building a BinaryTree

BinaryTree.java

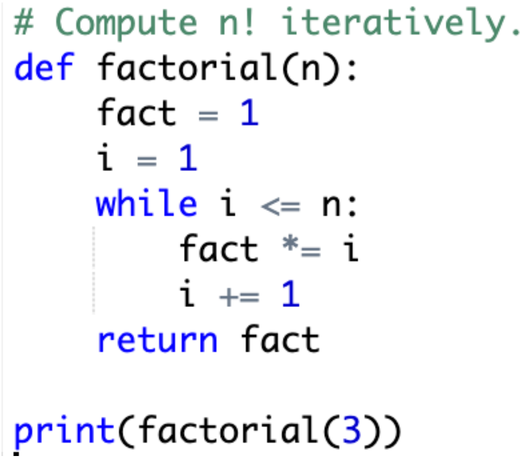


```java
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");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

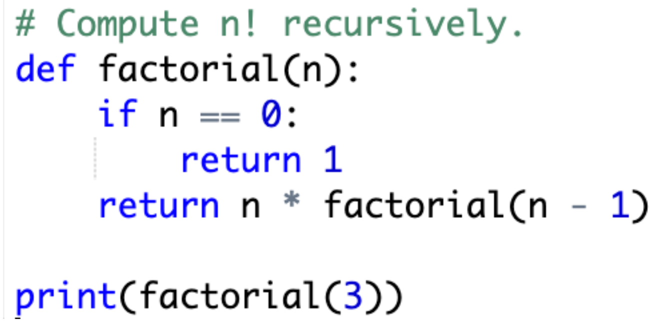
Draw the corresponding tree

Recursion: short review

- n! (n factorial)
- Iterative formulation
  - $n! = 1$, if $n = 0$, and
  - $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$, if $n > 0$
- Recursive formulation
  - $n! = 1$, if $n = 0$, and
  - $n! = n \times (n-1)!$, if $n > 0$



```
# Compute n! iteratively.
def factorial(n):
    fact = 1
    i = 1
    while i <= n:
        fact *= i
        i += 1
    return fact

print(factorial(3))
```



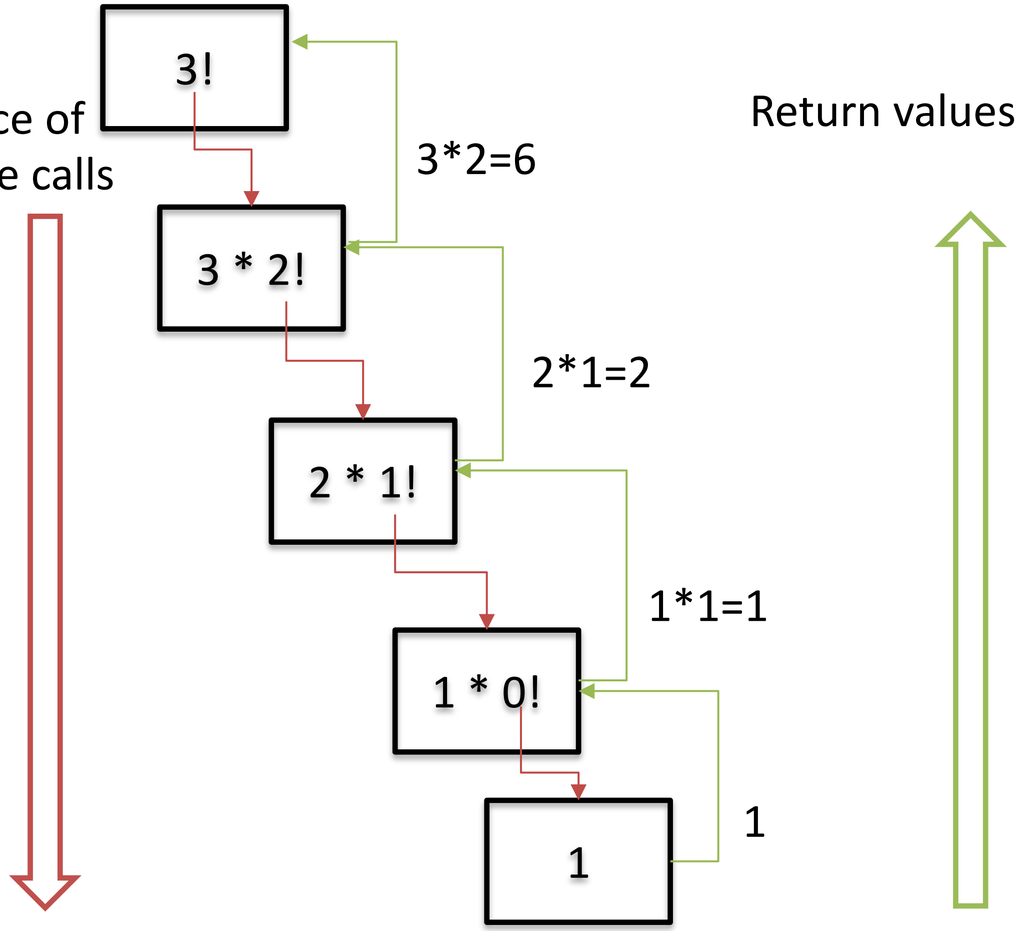
```
# Compute n! recursively.
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(3))
```

http://projectpython.net/chapter12/

Recursion: short review

- Call stack

factorial	n= 3 (return to): main	
factorial	n= 2 (return to): factorial	
factorial	n = 1 (return to): factorial	
factorial	n = 0 (return to): factorial	return value: 1



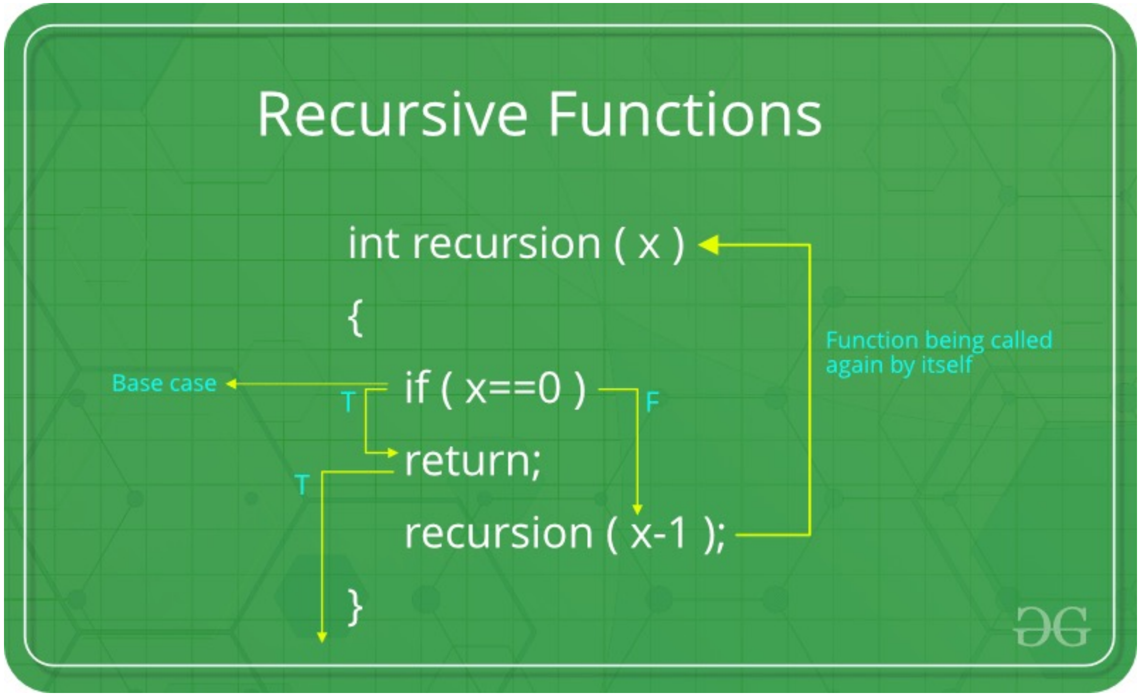
Sequence of recursive calls

Return values

http://projectpython.net/chapter12/

Recursion: short review

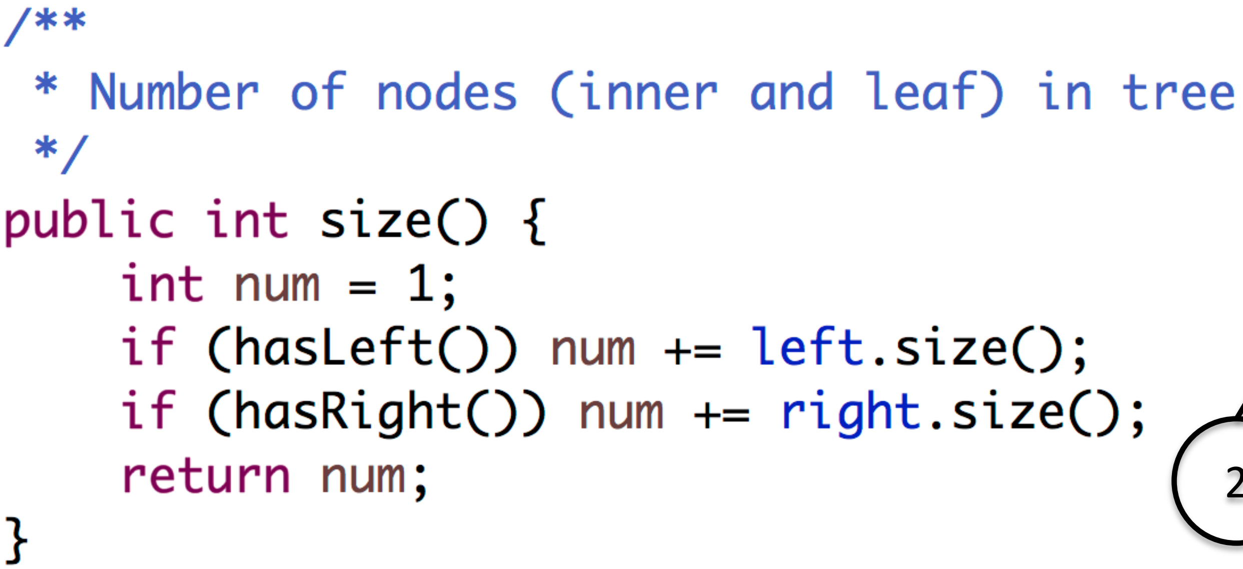
- General view: need to define
  - Base case
  - Recursive case



```
int recursion ( x )
{
   if ( x==0 )
   return;
   recursion ( x-1 );
}
```

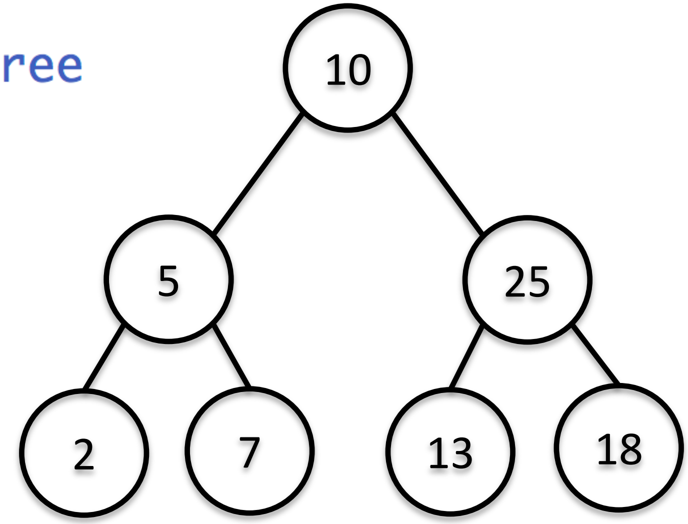
Use recursion to calculate tree size from any given node = size of both children +1

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;
}
```

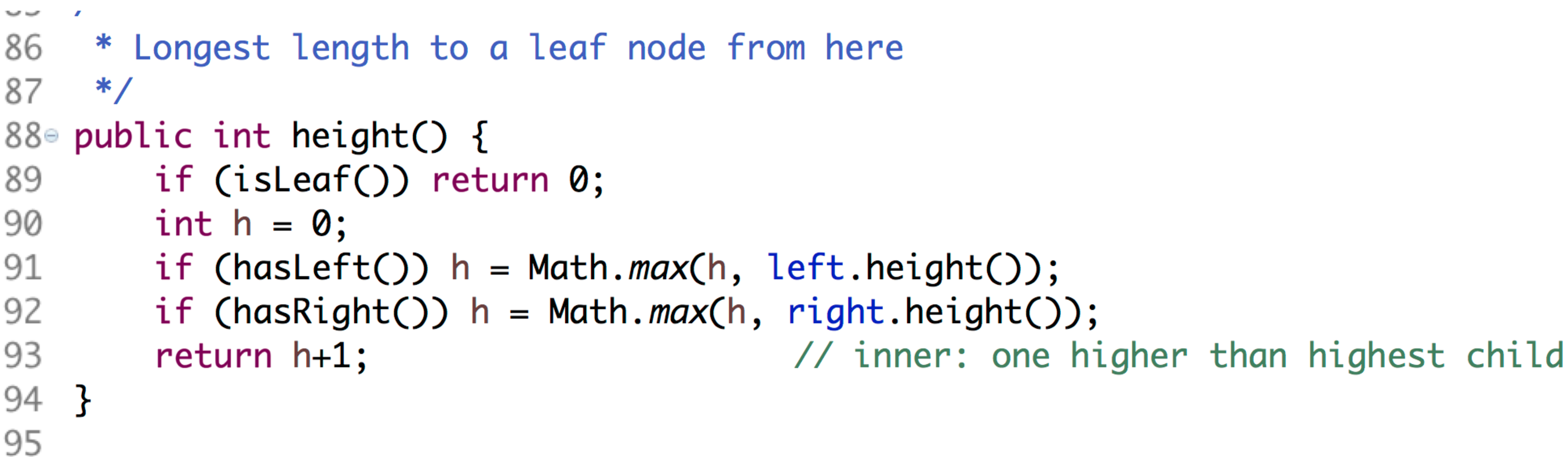
Call *size()* on root node



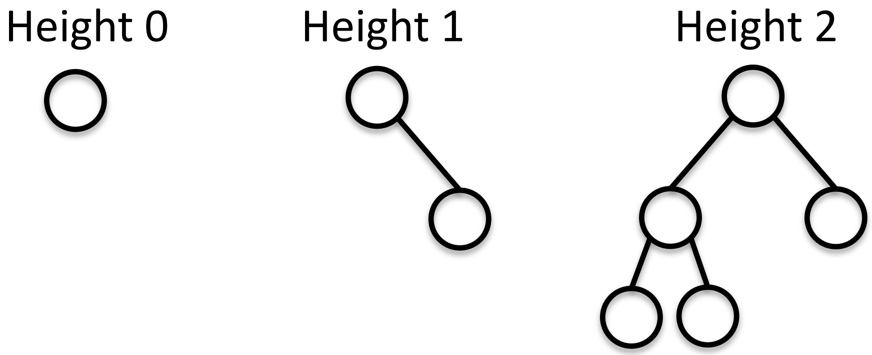
On paper example on the tree

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

BinaryTree.java

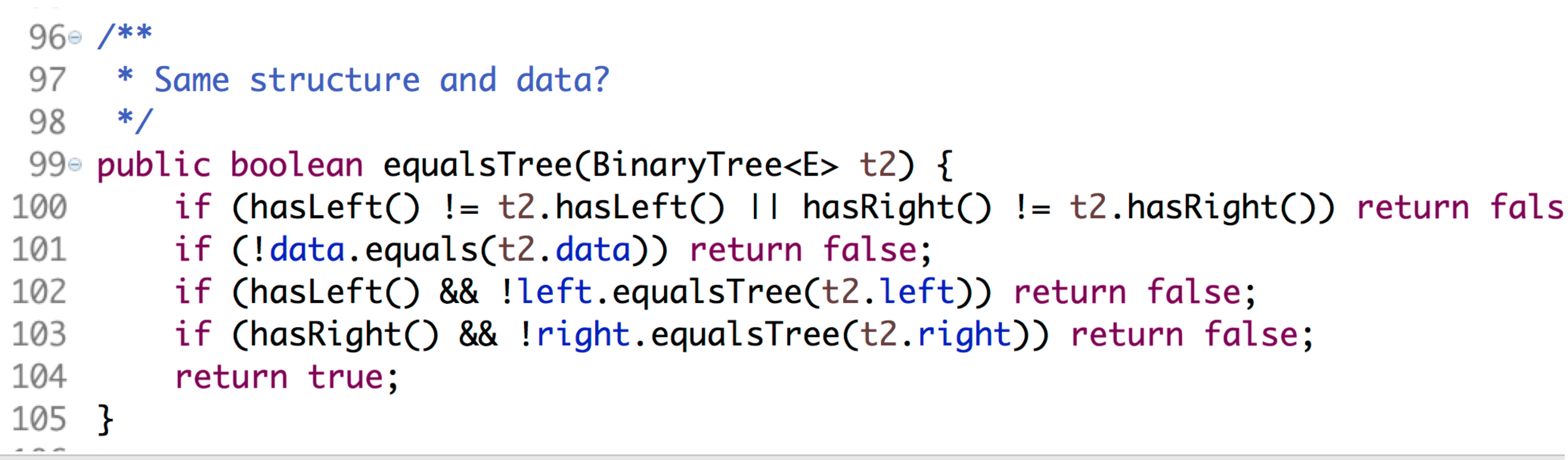


```java
 * Longest length to a leaf node from here
 */
public int height() {
    if (isLeaf()) return 0;
    int h = 0;
    if (hasLeft()) h = Math.max(h, left.height());
    if (hasRight()) h = Math.max(h, right.height());
    return h+1;                          // inner: one higher than highest child
}
```



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

BinaryTree.java



```java
/**
 * Same structure and data?
 */
public boolean equalsTree(BinaryTree<E> t2) {
    if (hasLeft() != t2.hasLeft() || hasRight() != t2.hasRight()) return fals
    if (!data.equals(t2.data)) return false;
    if (hasLeft() && !left.equalsTree(t2.left)) return false;
    if (hasRight() && !right.equalsTree(t2.right)) return false;
    return true;
}
```

Trees are equal if same shape and same data

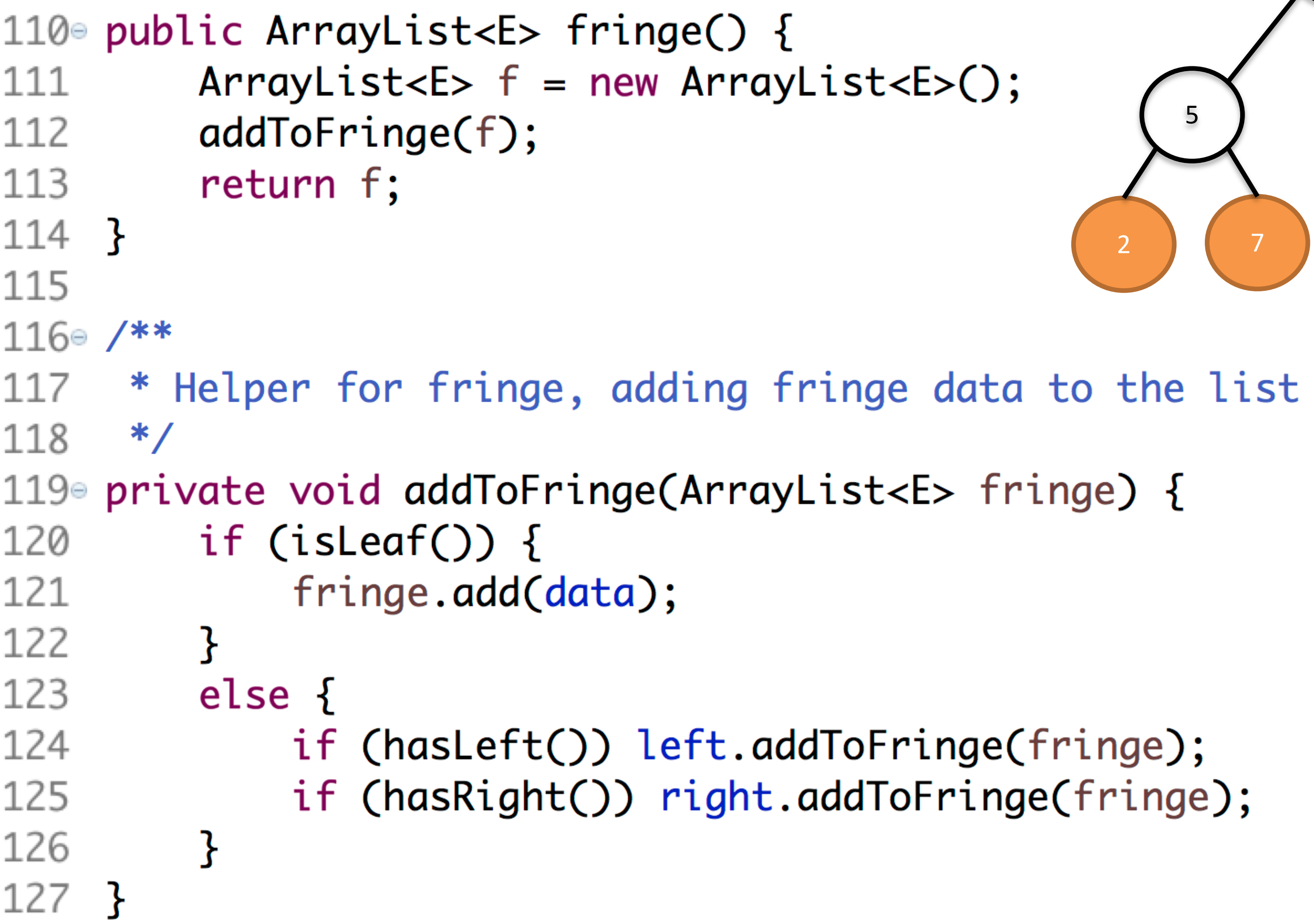
Would tree1 == tree2 work?

Agenda

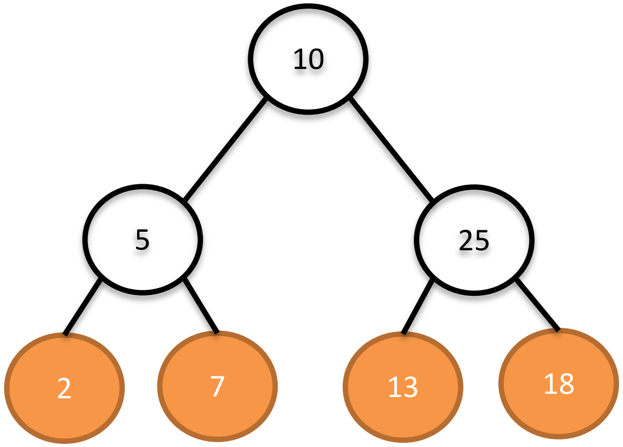
1. General-purpose binary trees
2. Accumulators
3. Tree traversal

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

BinaryTree.java



```
public ArrayList<E> fringe() {
    ArrayList<E> f = new ArrayList<E>();
    addToFringe(f);
    return f;
}

/**
 * Helper for fringe, adding fringe data to the list
 */
private void addToFringe(ArrayList<E> fringe) {
    if (isLeaf()) {
        fringe.add(data);
    }
    else {
        if (hasLeft()) left.addToFringe(fringe);
        if (hasRight()) right.addToFringe(fringe);
    }
}
```



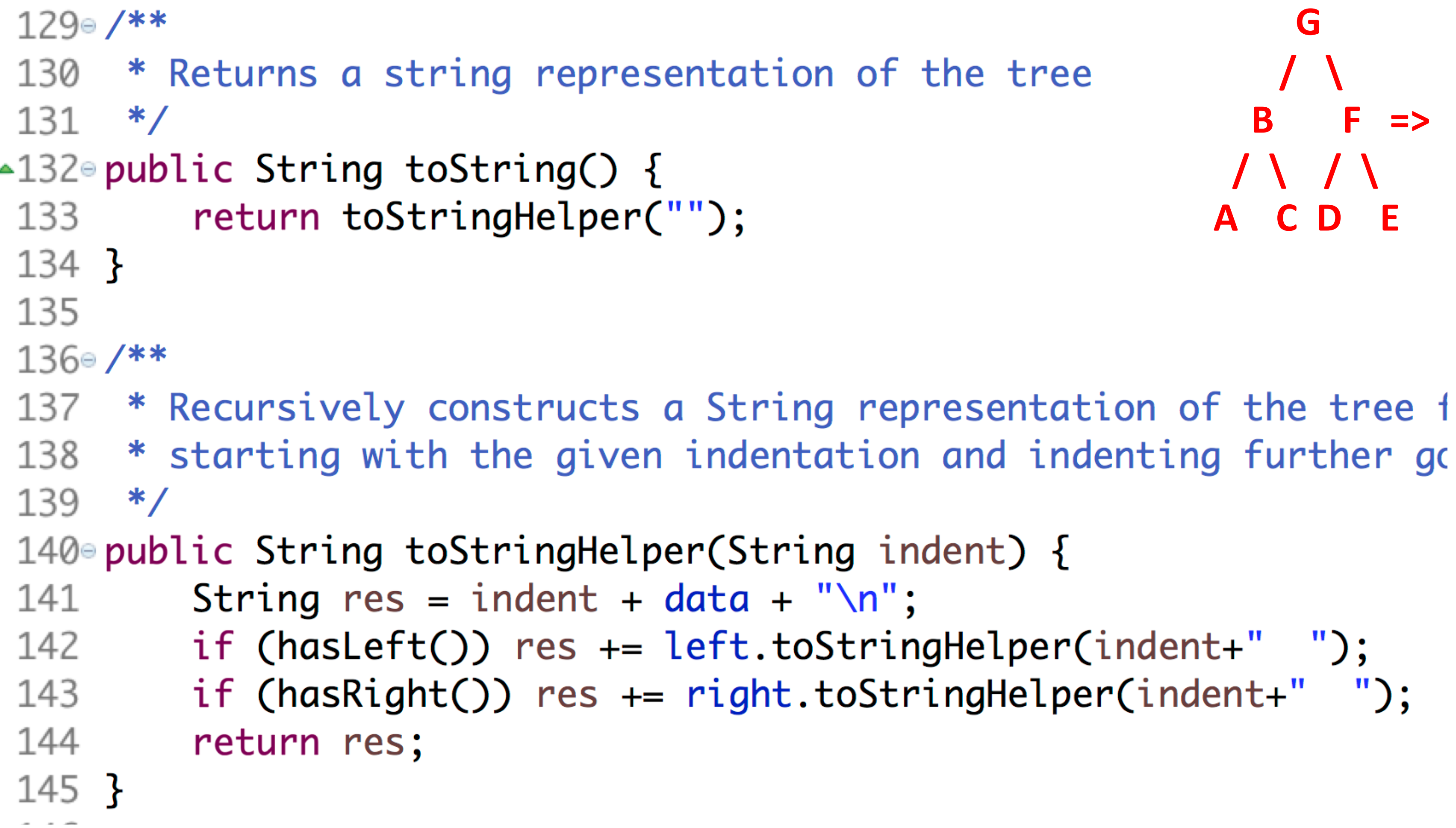
The *fringe* of a tree is the list of *leaves* in order from left to right [2, 7, 13, 18]

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

BinaryTree.java

Want to print Tree indented by level

```
      G           G
     / \            B
   B     F  =>        A
  / \   / \           C
 A   C D   E        F
                      D
                      E
```



```java
/**
 * Returns a string representation of the tree
 */
public String toString() {
    return toStringHelper("");
}

/**
 * Recursively constructs a String representation of the tree f
 * starting with the given indentation and indenting further g
 */
public String toStringHelper(String indent) {
    String res = indent + data + "\n";
    if (hasLeft()) res += left.toStringHelper(indent+"  ");
    if (hasRight()) res += right.toStringHelper(indent+"  ");
    return res;
}
```

Agenda

1. General-purpose binary trees

2. Accumulators

➡ 3. Tree traversal

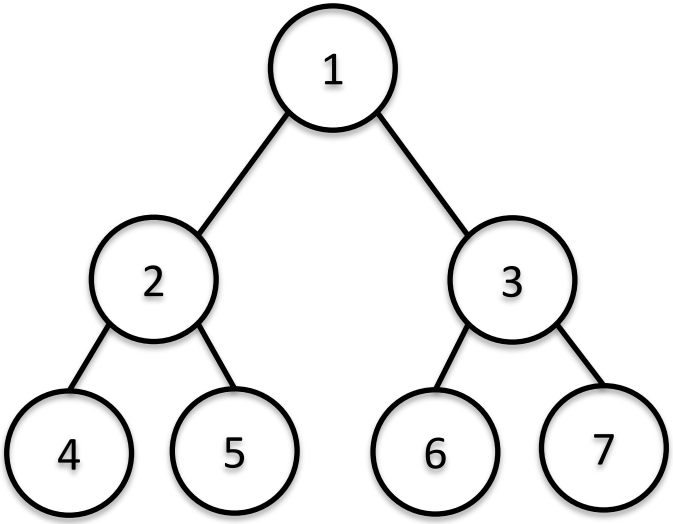
There are different ways to traverse a tree, depending on what needs to be done

```
preorder()
    visit
    left.preorder()
    right.preorder()
```

```
postorder()
    left.postorder()
    right.postorder()
    visit
```

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

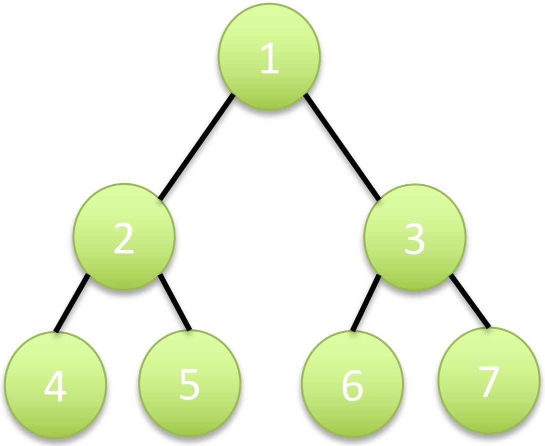
“visit” means “handle this node”, might print it, might do something else



On paper example on the tree

Summary: order in which nodes are visited depends on the type of traversal

Preorder



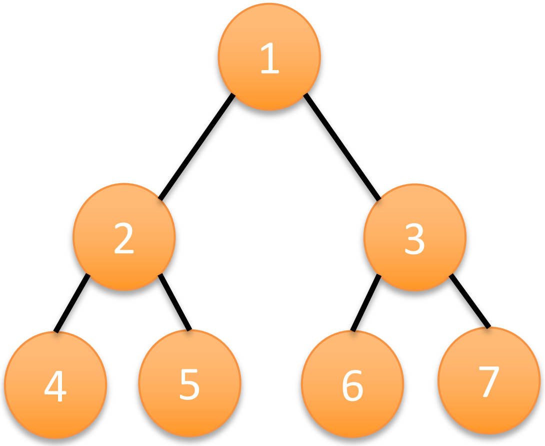
Visited

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

Book chapters

toString()

Postorder

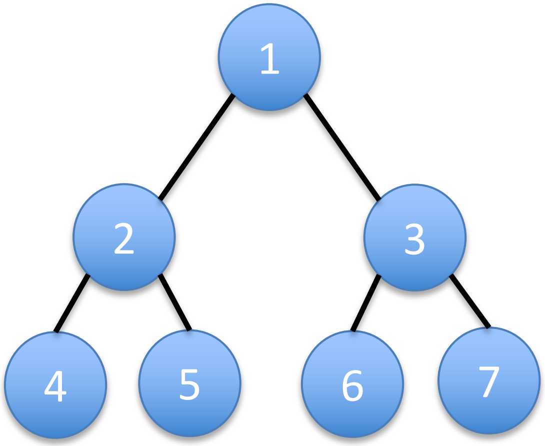


Visited

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

Calculate disk space

Inorder



Visited

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

Drawing a tree

(left to right)

Summary

- BinaryTree implementation
- Recursive strategy to visit the tree
- Accumulator+helper method to efficiently perform operations and store partial results
- Different traversal order for different operations

Next

- Use of binary tree for binary search

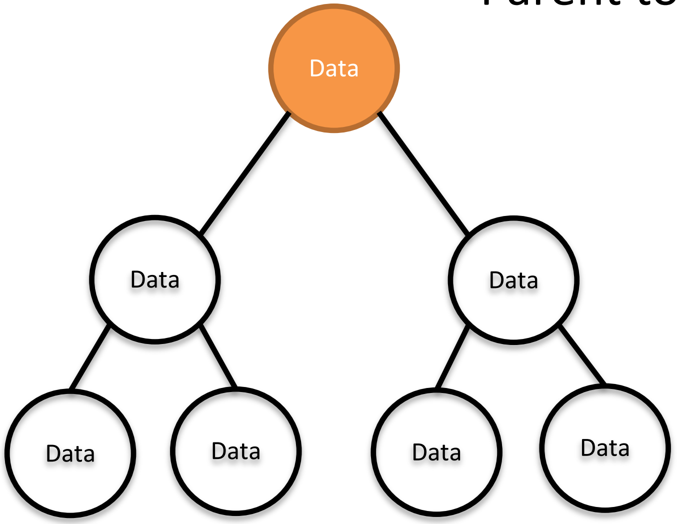
Additional Resources

TREE DATA STRUCTURE

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)

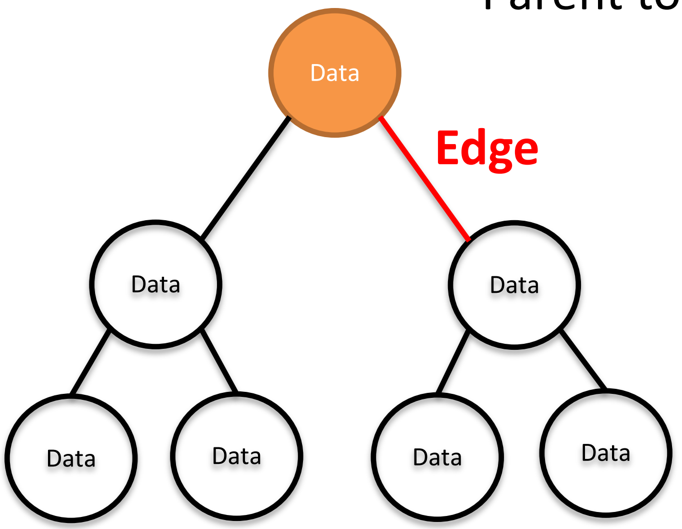


Meant for hierarchical data where there is a relationship between the data each node holds

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)

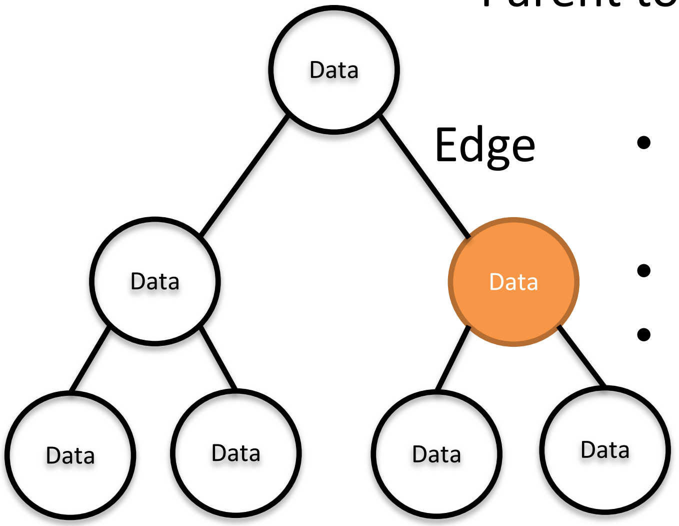


Meant for hierarchical data where there is a relationship between the data each node holds

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)



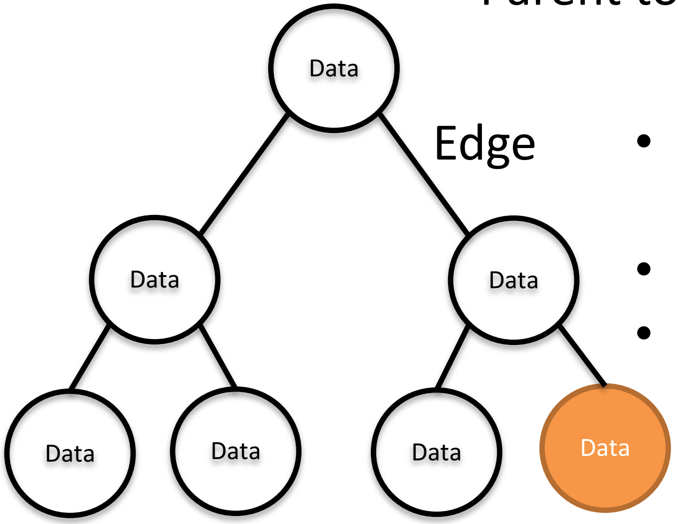
- Child node of root (children have exactly one parent)
- Parent node to children below
- Interior node (nodes also called vertices)

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)

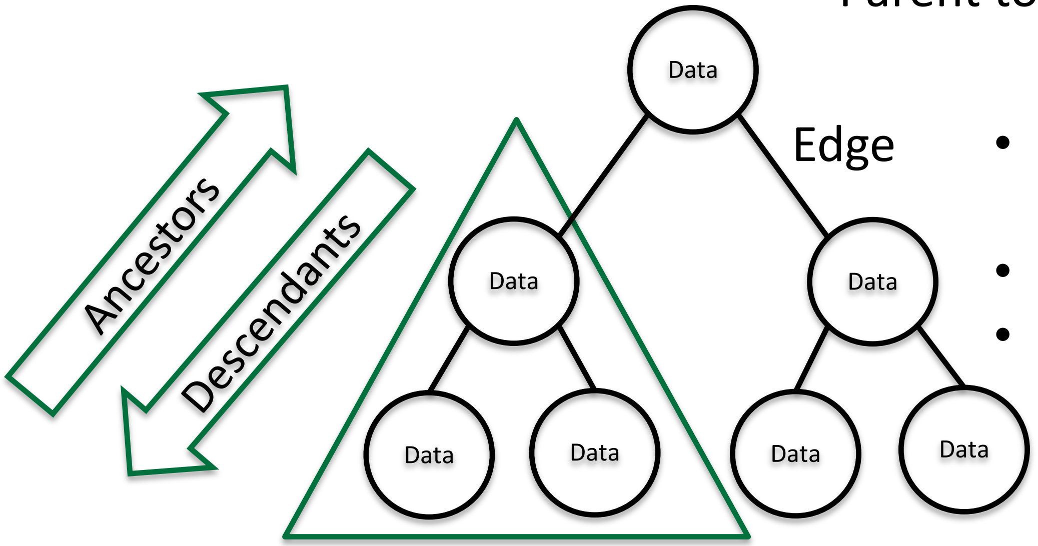
- Leaf (or external) node
- Right child of parent node

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

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

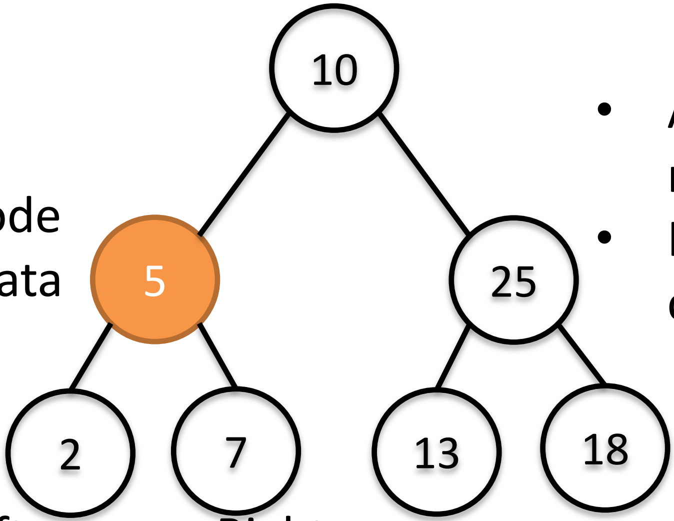
Subtree

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

BINARY TREE DATA STRUCTURE

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

Binary Tree data structure



Each node holds data

0, 1, or 2 children in *BinaryTree*

Left child

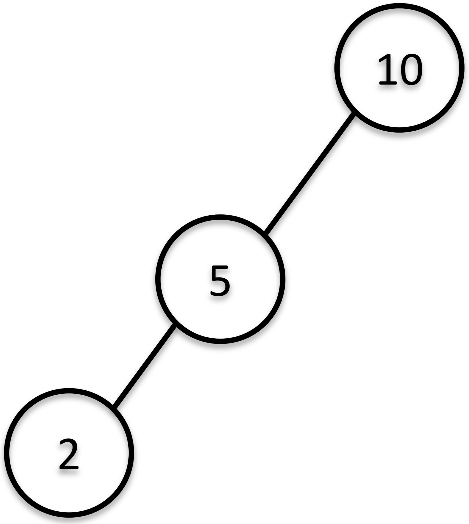
Right child

- 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



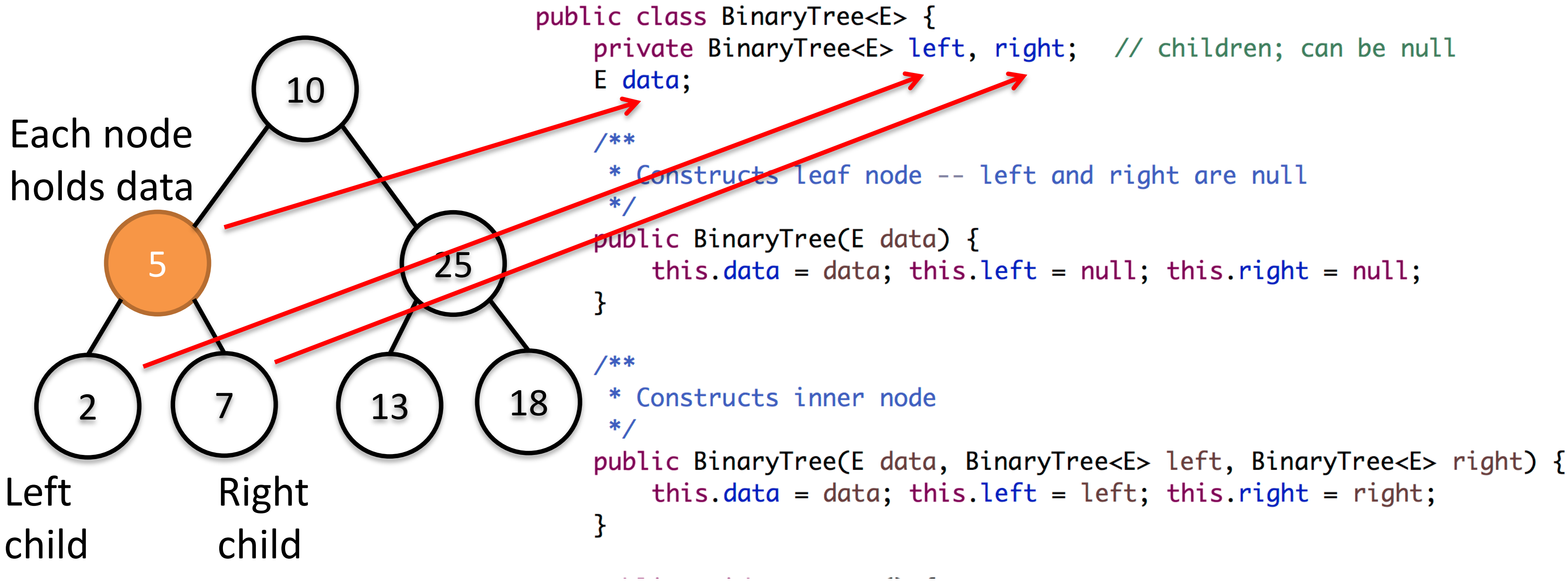
- This is a valid Binary Tree, each node has 0, 1, (or 2) children
- For now we make no guarantees a tree is balanced
- Later we will look at ways to ensure balance
- Balance will allow us to make stronger statements about run time performance

BinaryTree.java

ANNOTATED SLIDES

Each node in a tree can be thought of as the head of its own subtree

BinaryTree.java

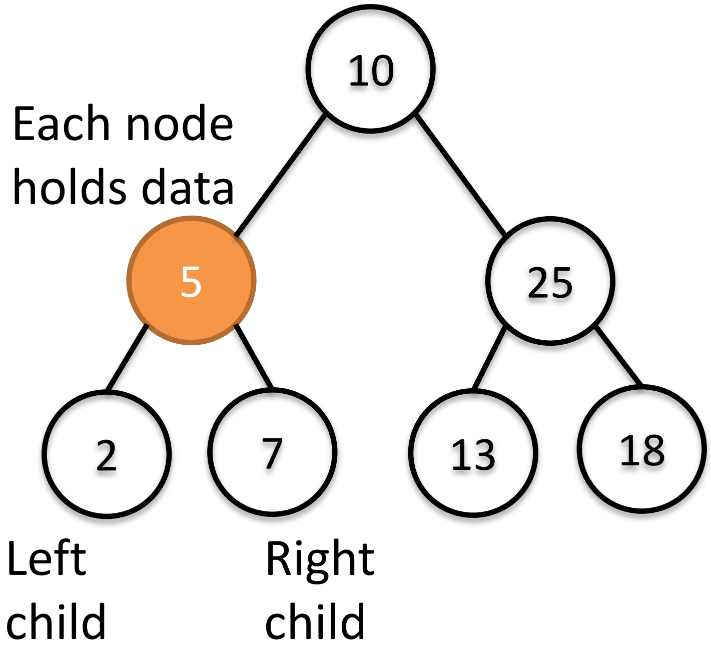


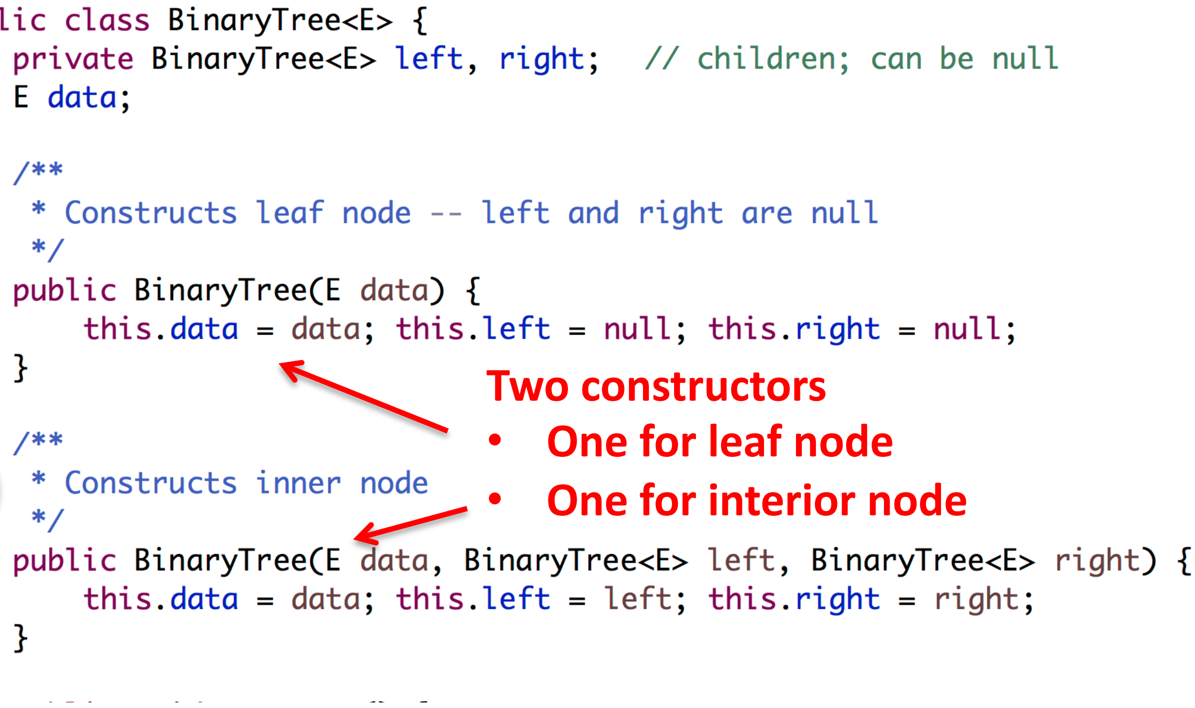
```
public class BinaryTree<E> {
    private BinaryTree<E> left, right;   // children; can be null
    E data;

    /**
     * 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;
    }
```

- **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 its own subtree

BinaryTree.java





```
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**

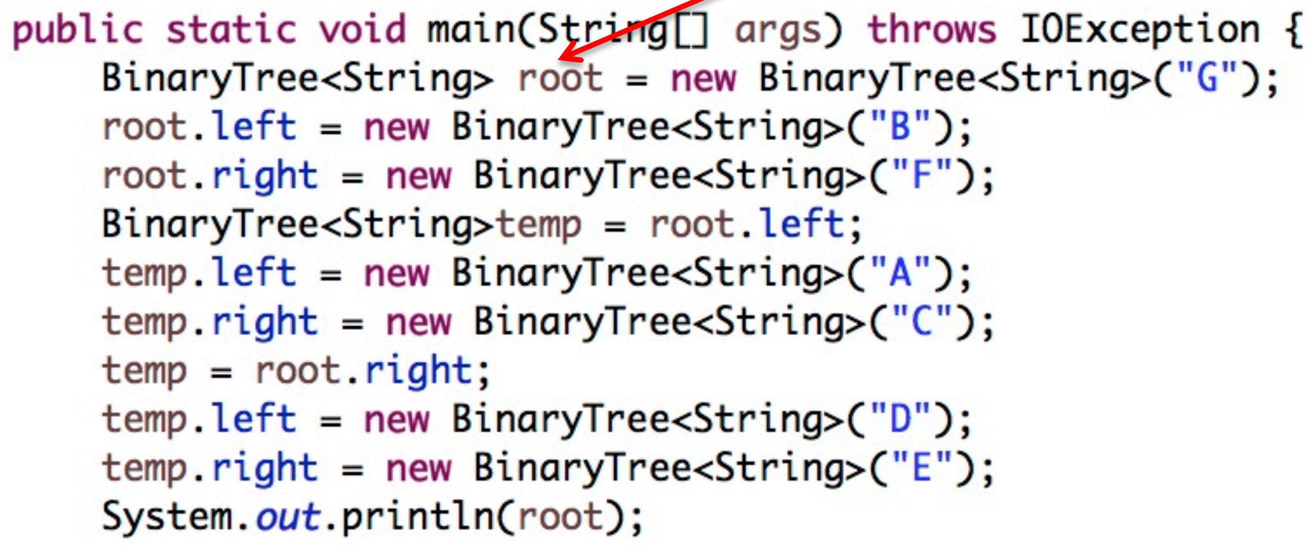
BinaryTree.java – main example

ANNOTATED SLIDES

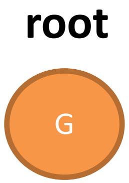
Building a BinaryTree

BinaryTree.java

Create root node



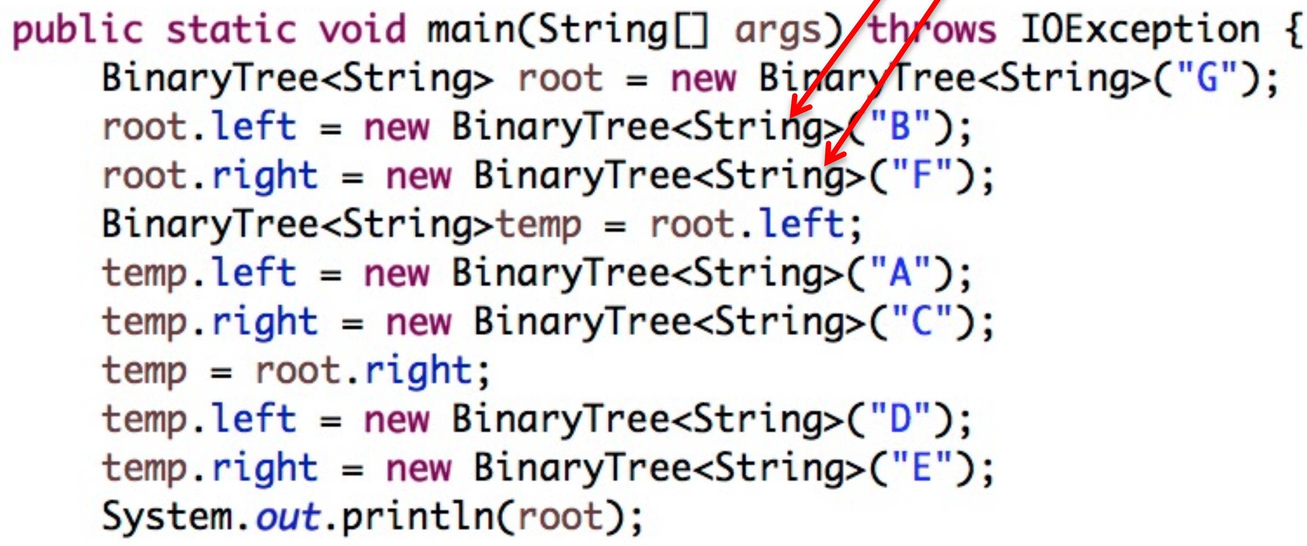
```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
 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



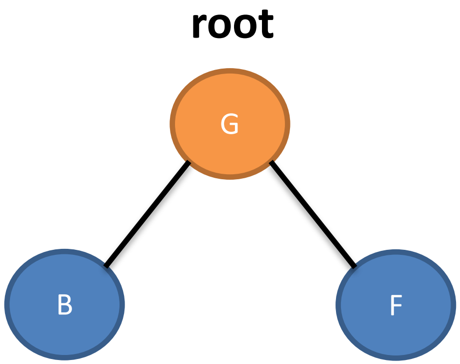
Building a BinaryTree

BinaryTree.java

Set left and right children



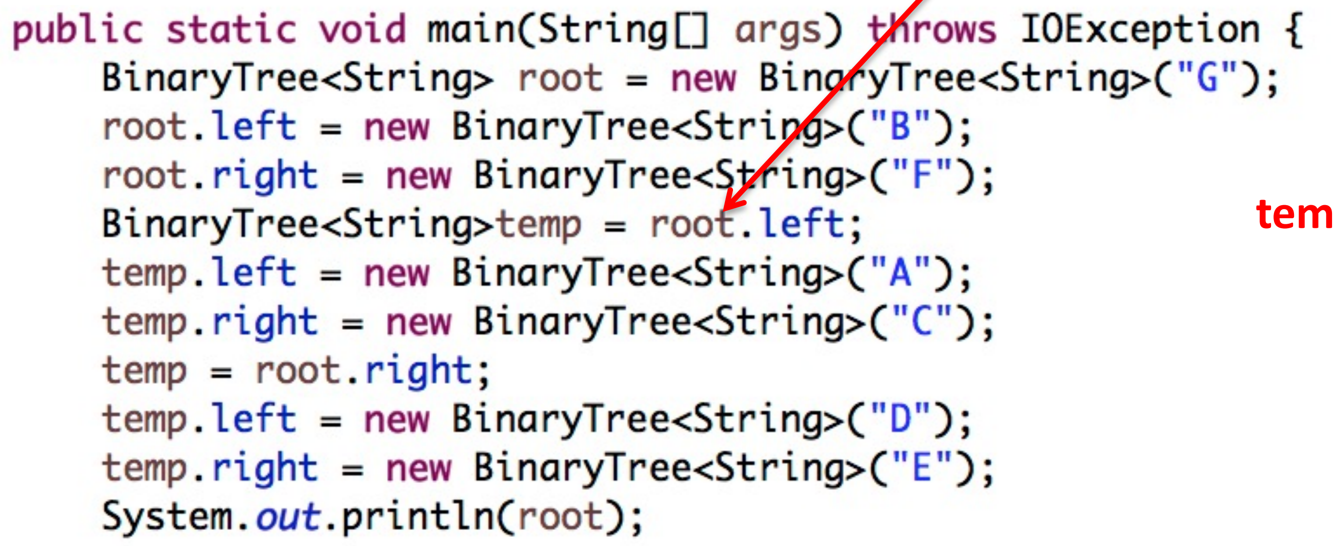
```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
 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



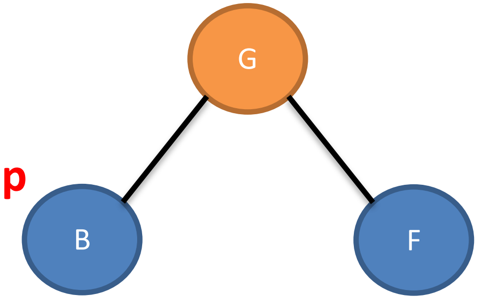
Building a BinaryTree

BinaryTree.java

Make temp node and traverse down to left child



```
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);
```

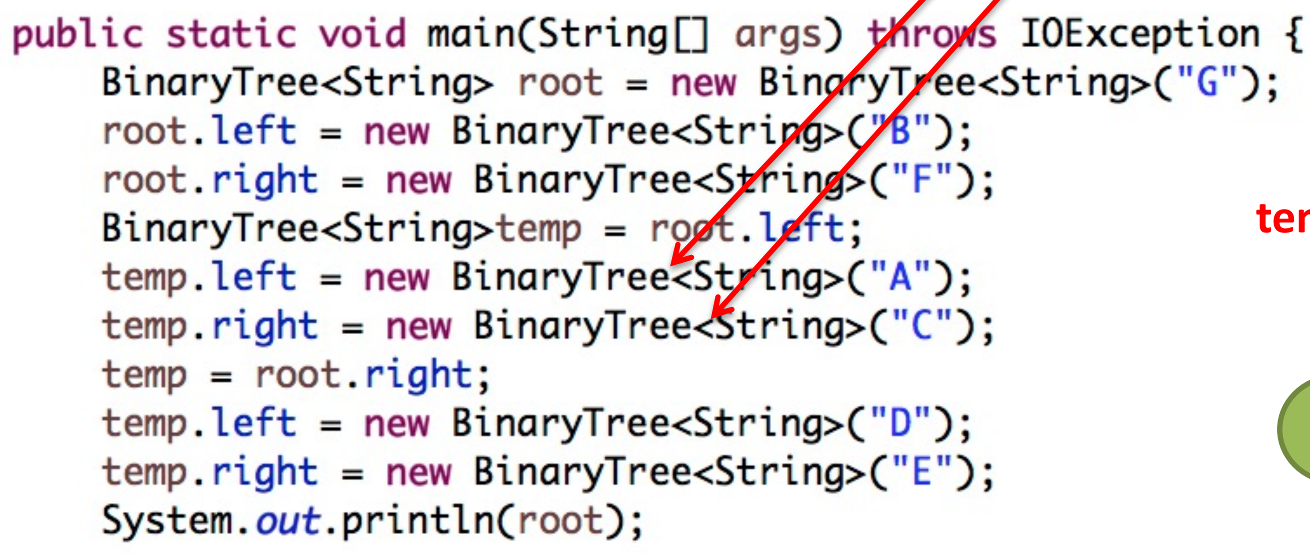


- What would happen if didn't create *temp = root.left*, but instead set *root = root.left*
- Would lose pointer to *root* node (*root* would be garbage collected)

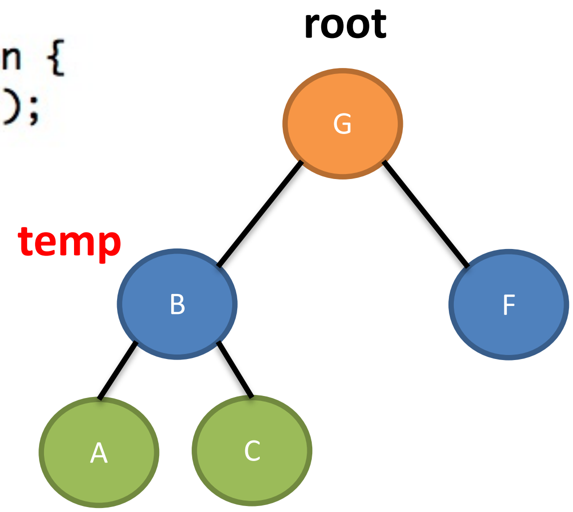
Building a BinaryTree

BinaryTree.java

Set left and right children



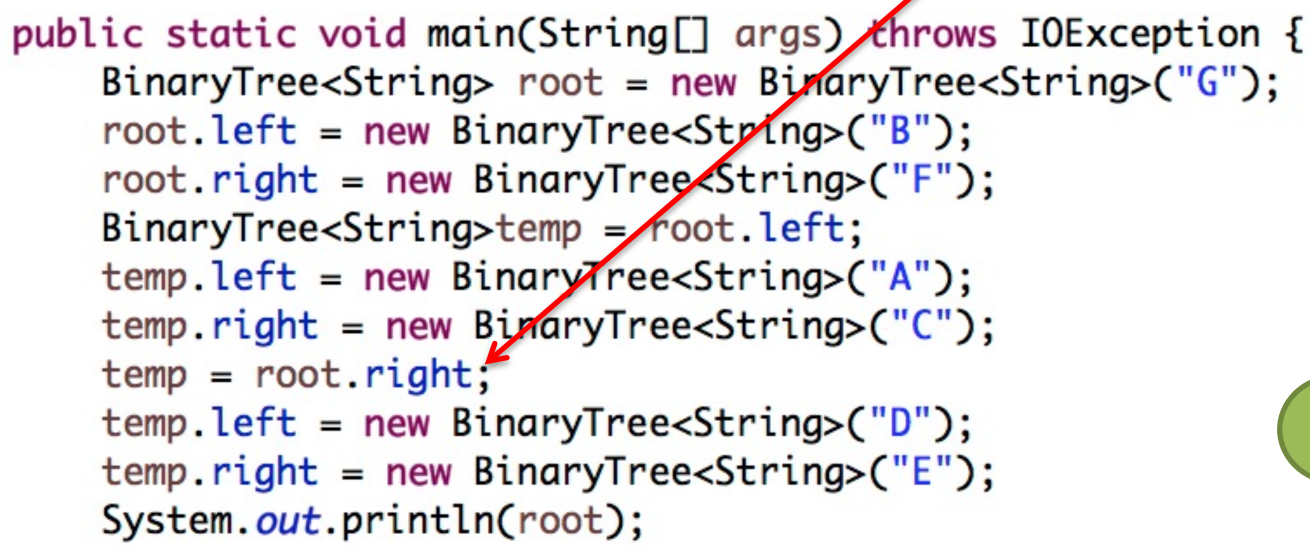
```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
 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



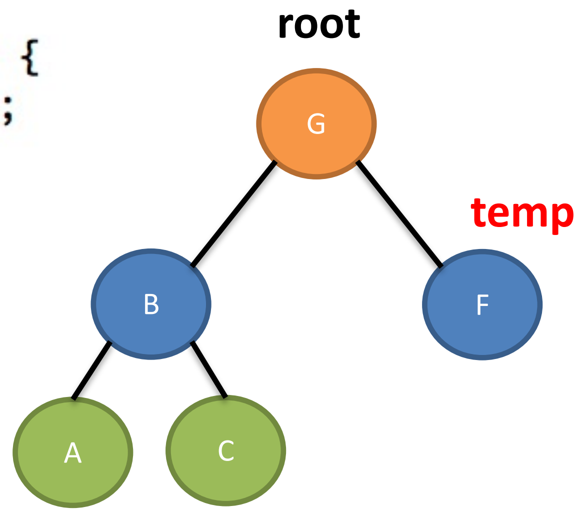
Building a BinaryTree

BinaryTree.java

Move temp to root's right child



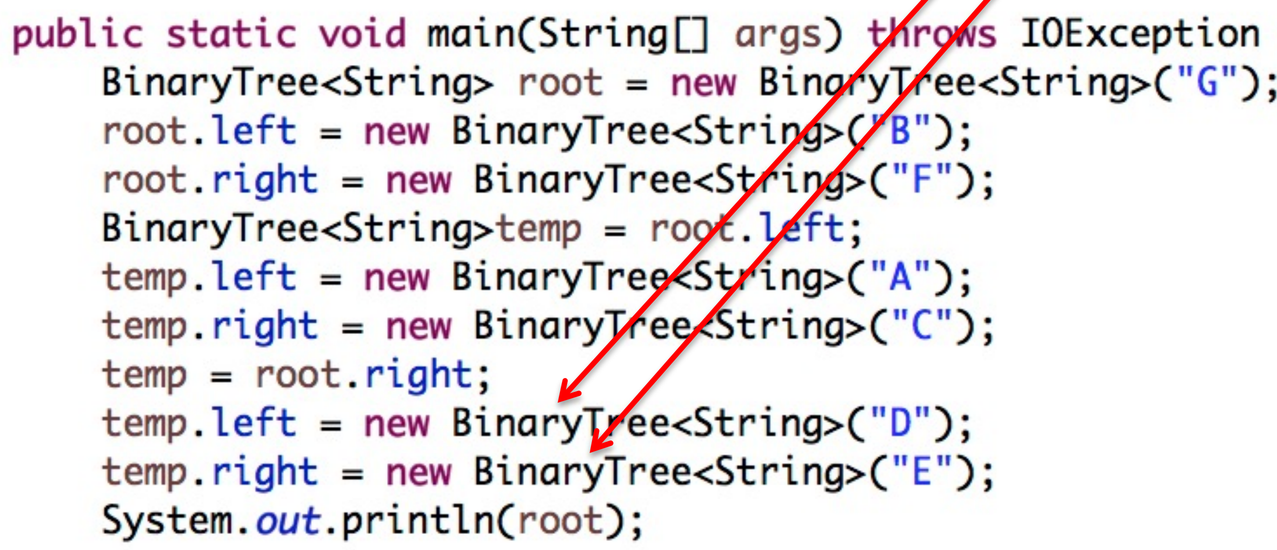
```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
 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



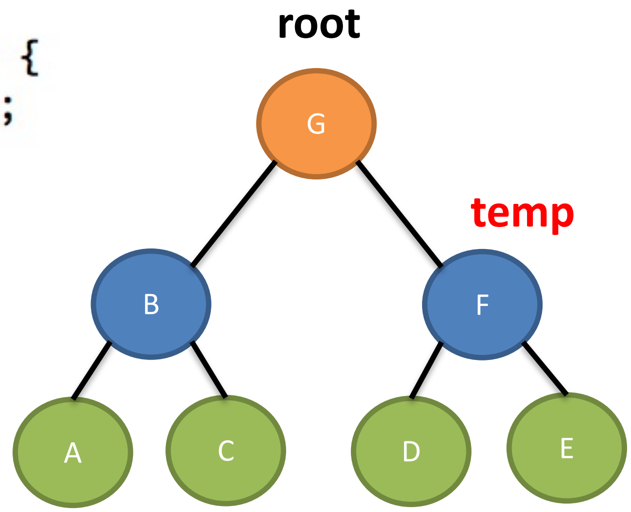
Building a BinaryTree

BinaryTree.java

Add children

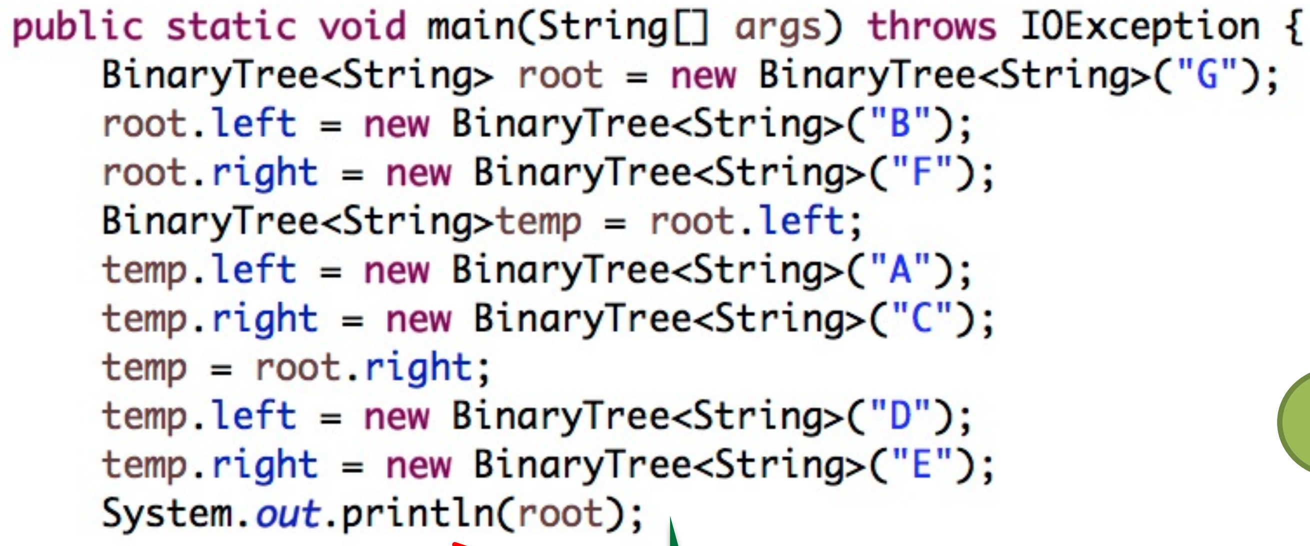


```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
 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

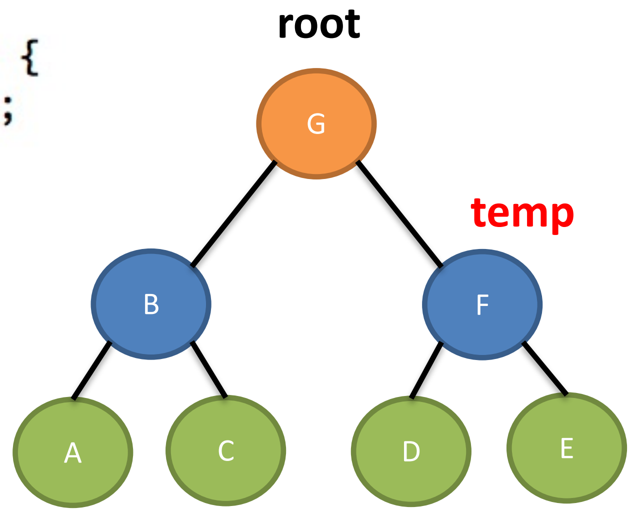


Building a BinaryTree

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)**

```
G
    B
        A
        C
    F
        D
        E
```

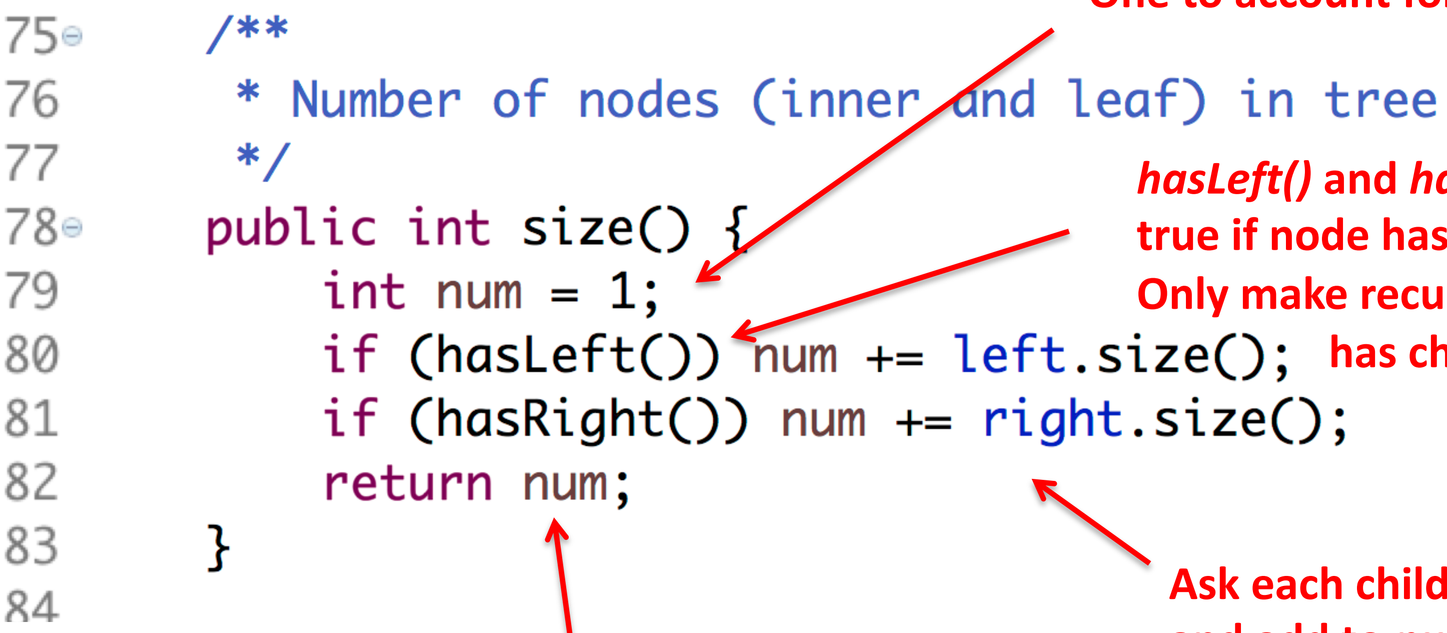
BinaryTree.java – size

ANNOTATED SLIDES

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**



```
/**
 * 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;
}
```

One to account for this node

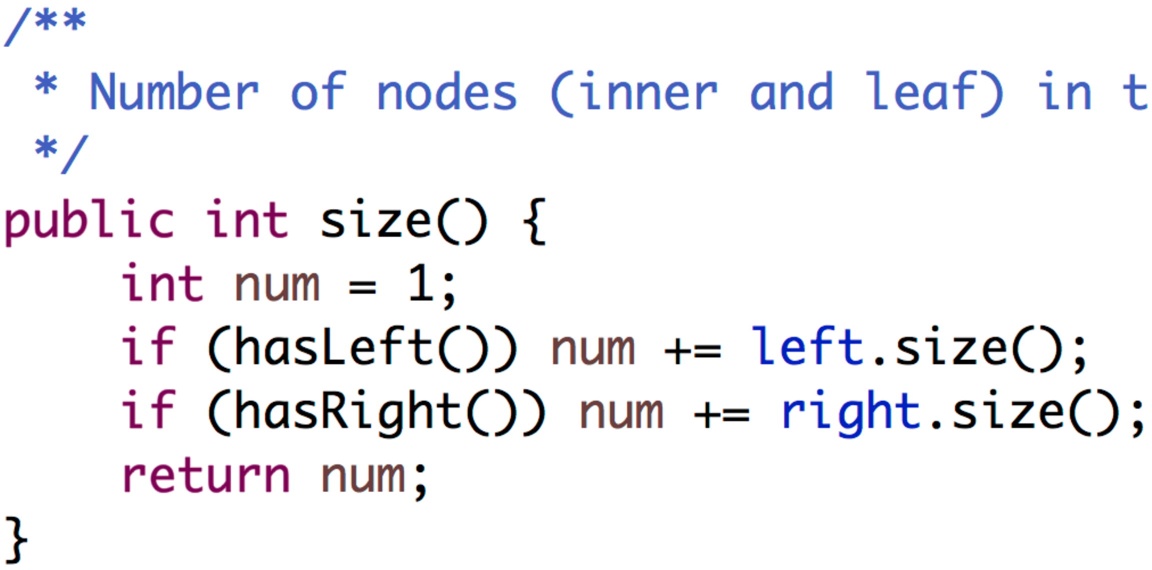
***hasLeft()* and *hasRight()* return true if node has those children**
Only make recursive call if node has child

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

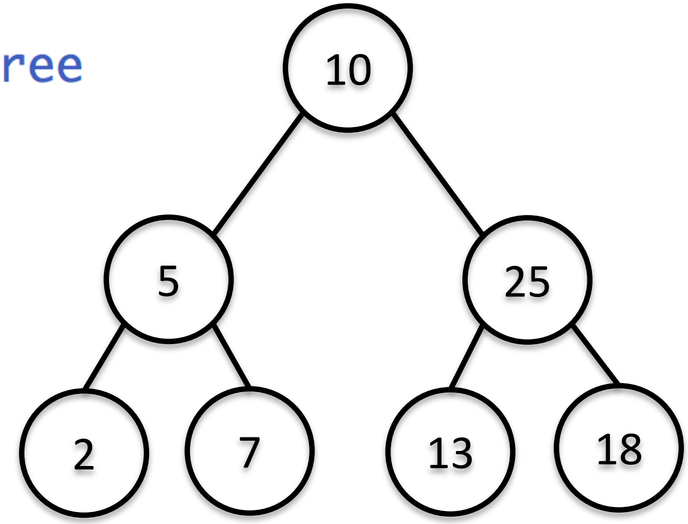
Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java



```
/**
 * 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;
}
```

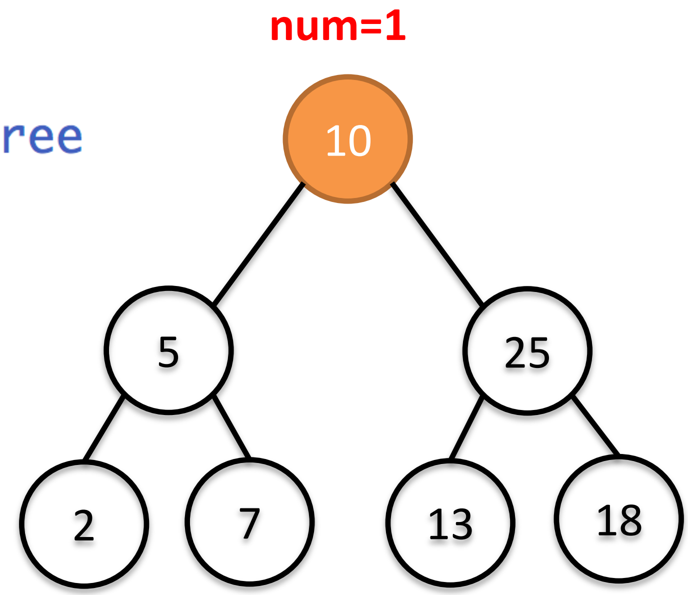
Call *size()* on root node



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;
}
```

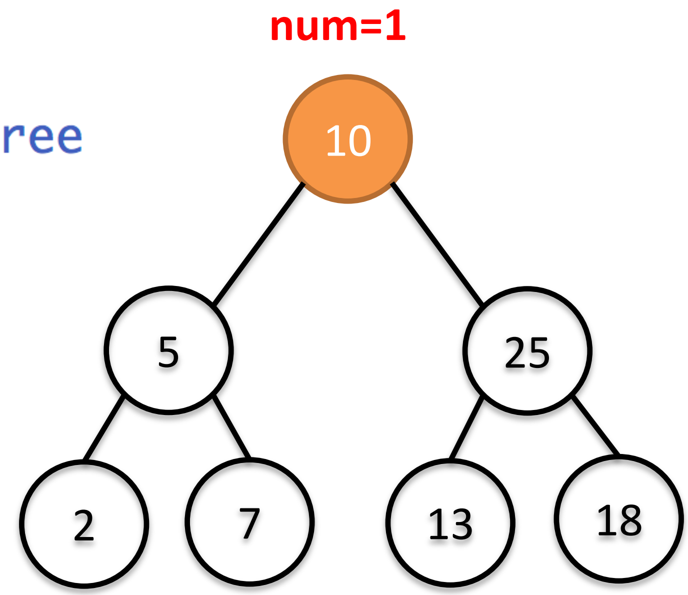


Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

10

```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;
}
```

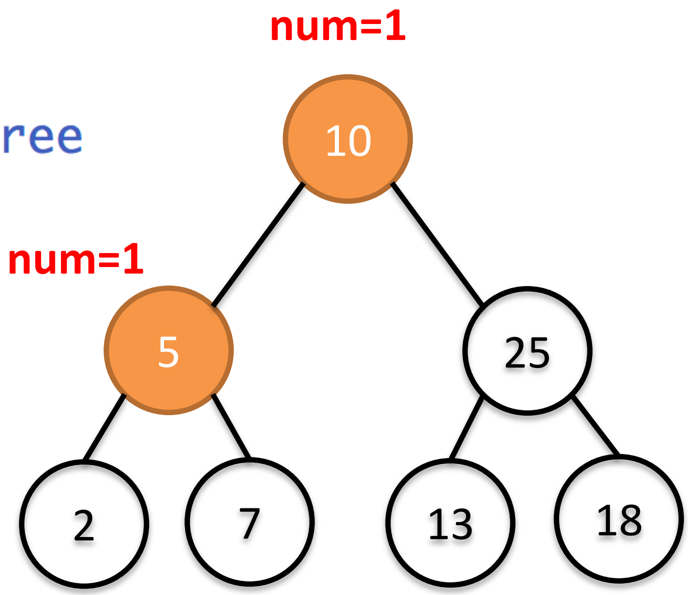


- Has left child
- Make recursive call on left child

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

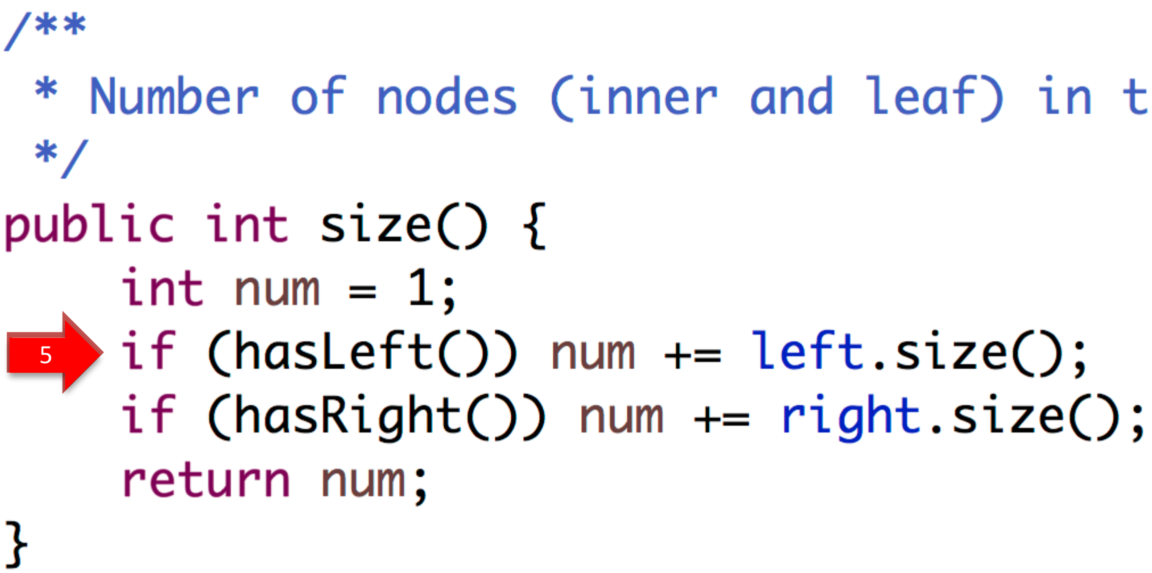
BinaryTree.java

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

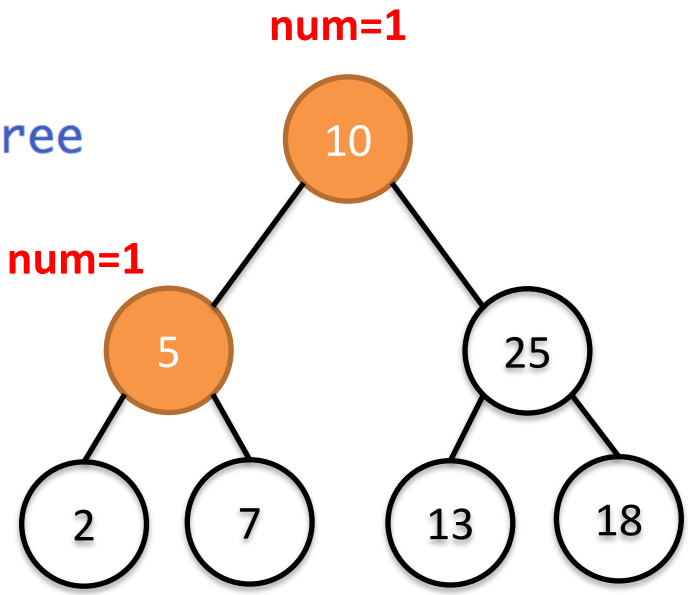


Use recursion to calculate tree size from any given node = size of both children +1

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;
}
```

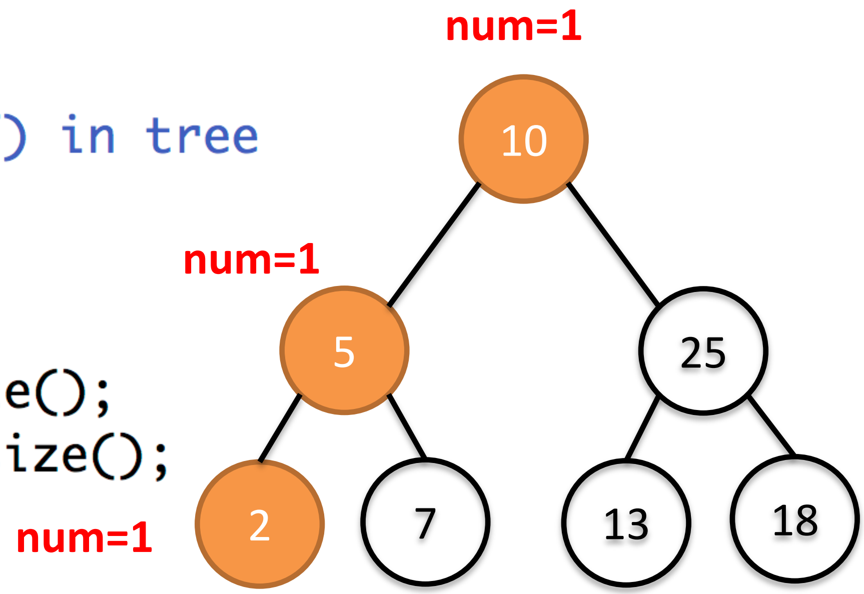


- Has left child
- Make recursive call on left child

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

BinaryTree.java

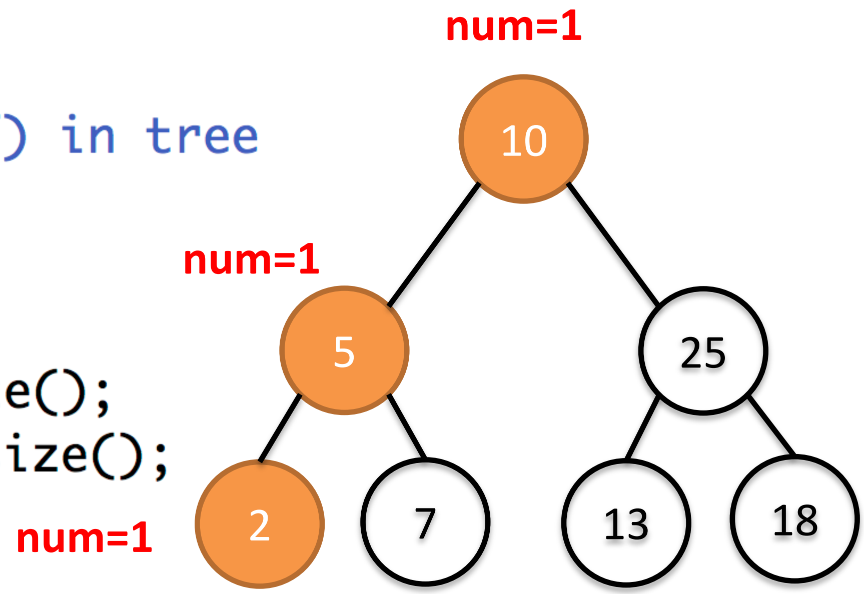
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
2   int num = 1;
5   if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



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;
}
```

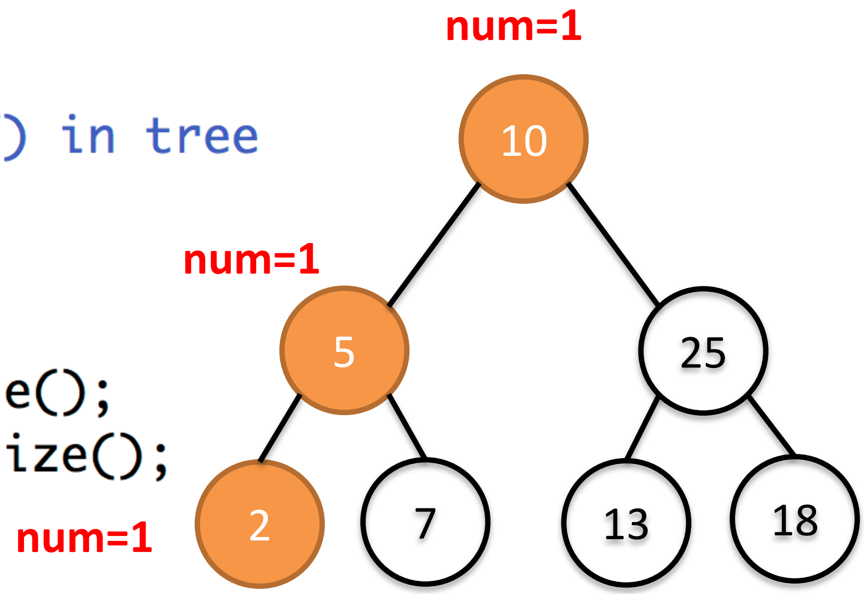


- No children

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;
}
```

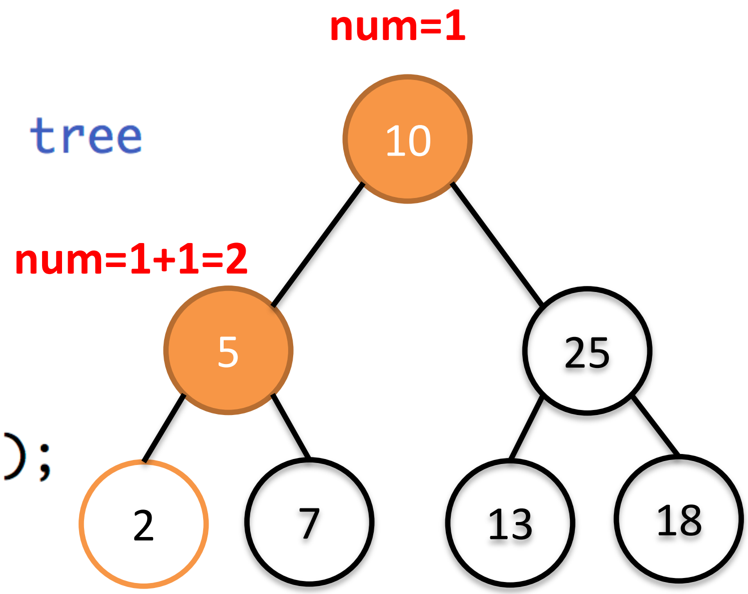


- No children
- Return 1 back to node 5

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

BinaryTree.java

```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;
}
```

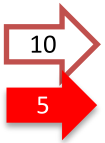


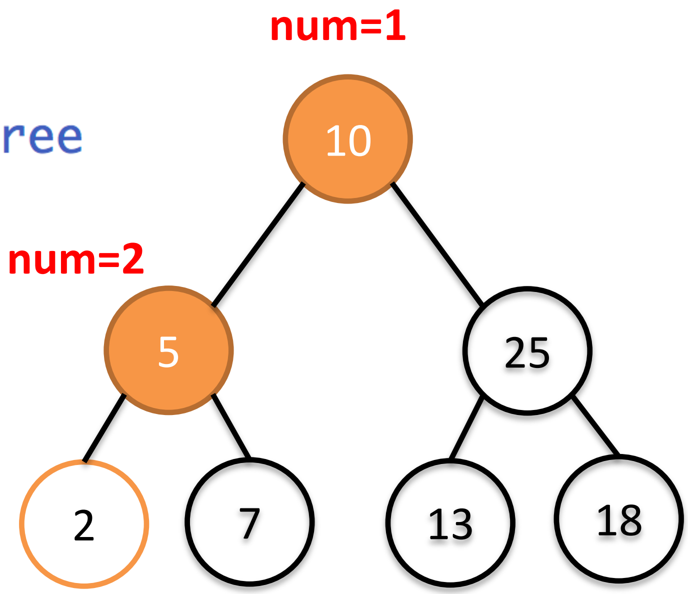
- Increment num on Node 5

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;
}
```

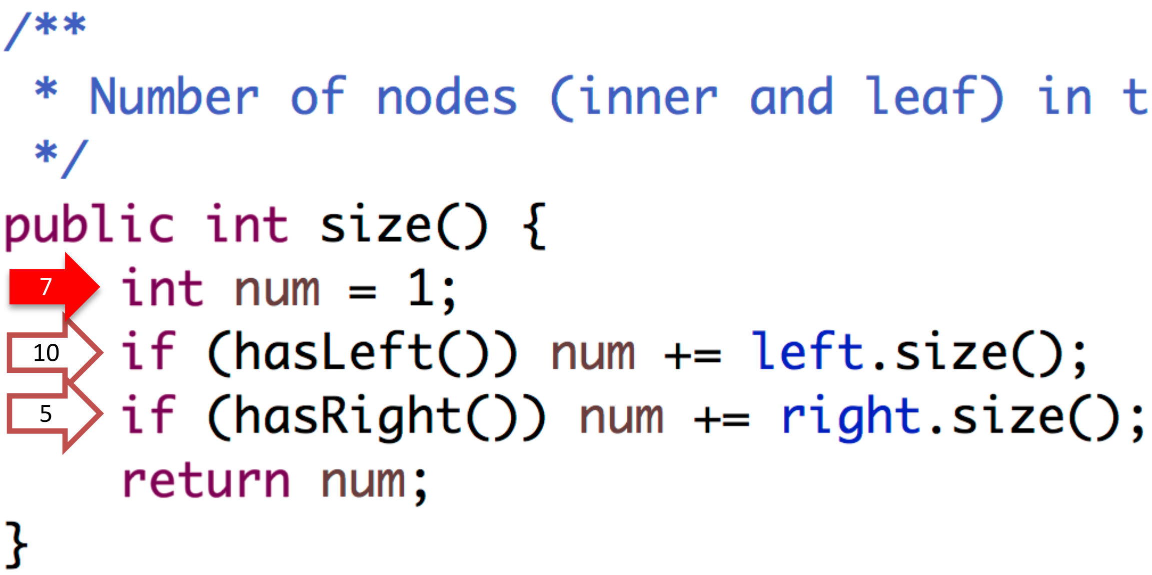




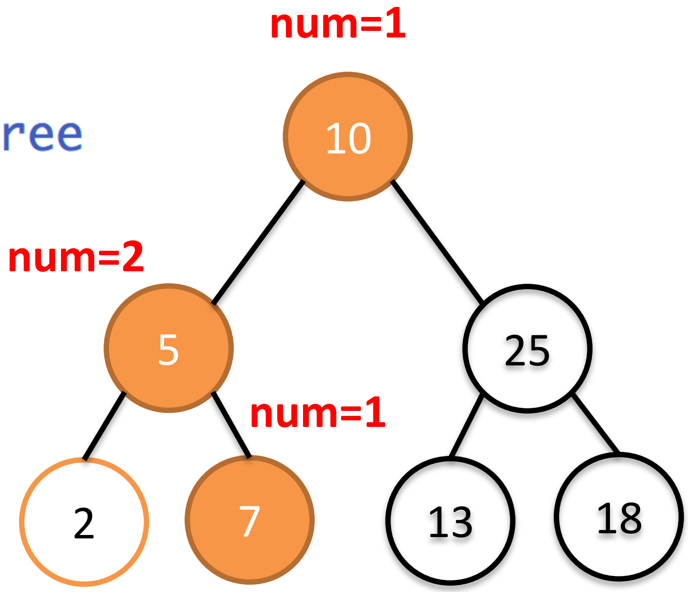
- Has right child
- Make recursive call on right child

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

BinaryTree.java



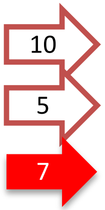
```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```

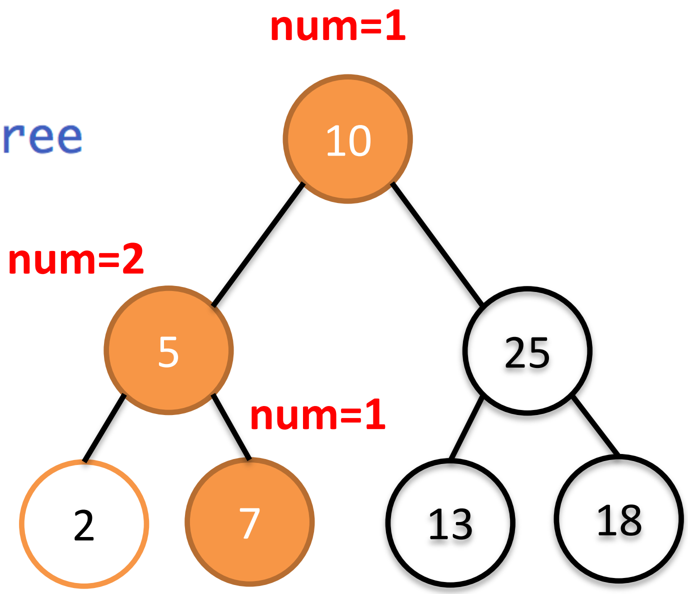


Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```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;
}
```

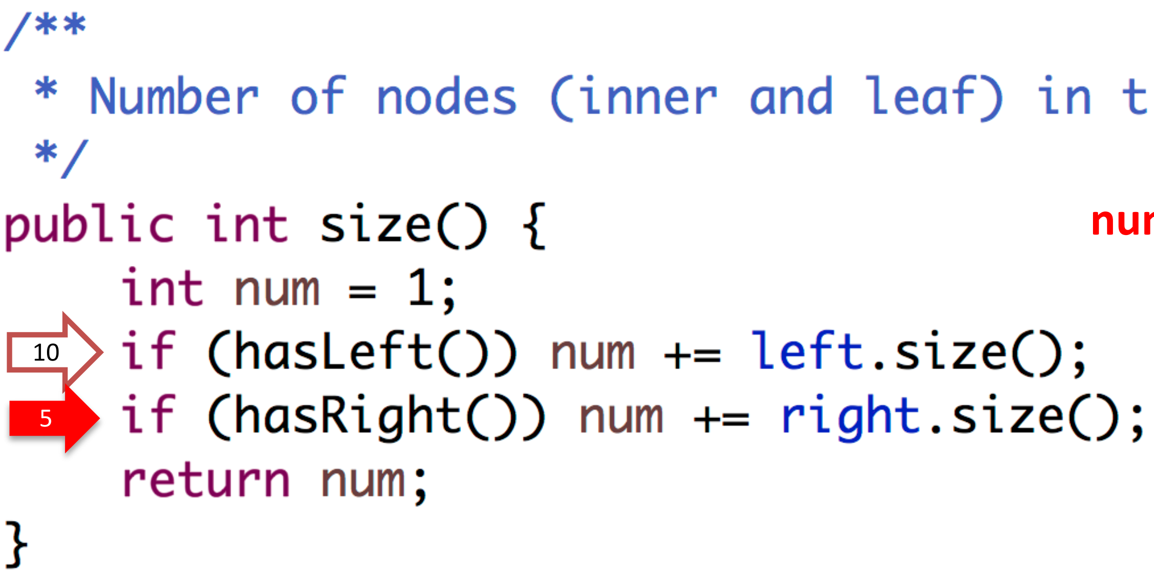




- **No children**
- **Return 1 back to node 5**

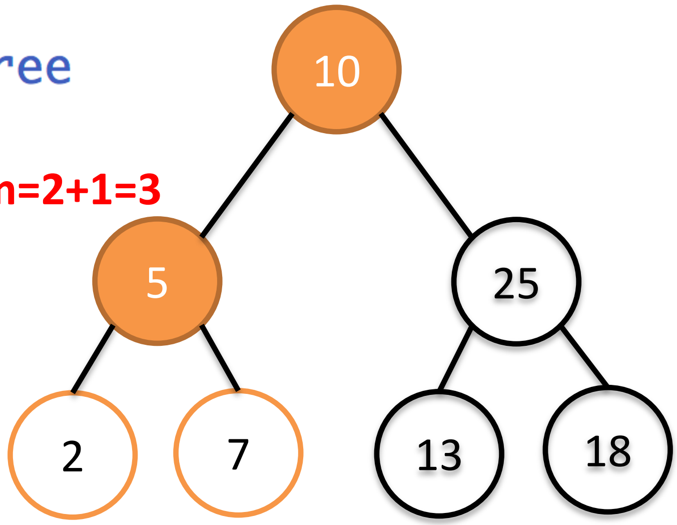
Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java



```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;
}
```

num=1



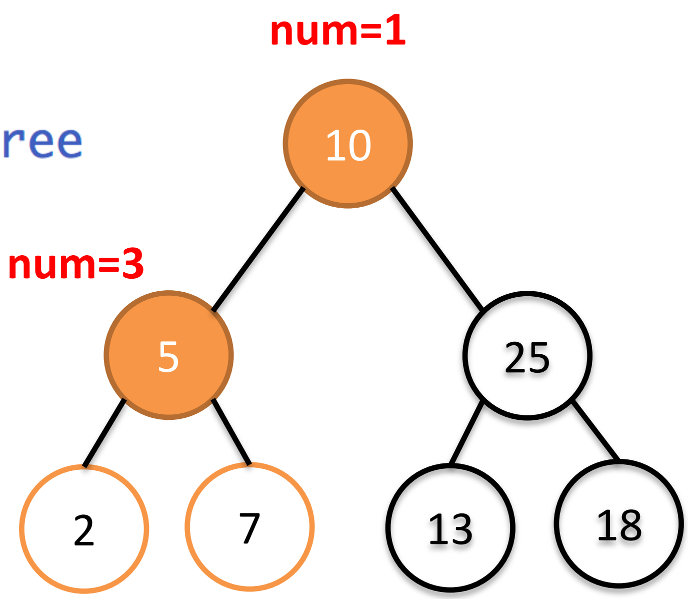
num=2+1=3

Increment num on Node 5

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

BinaryTree.java

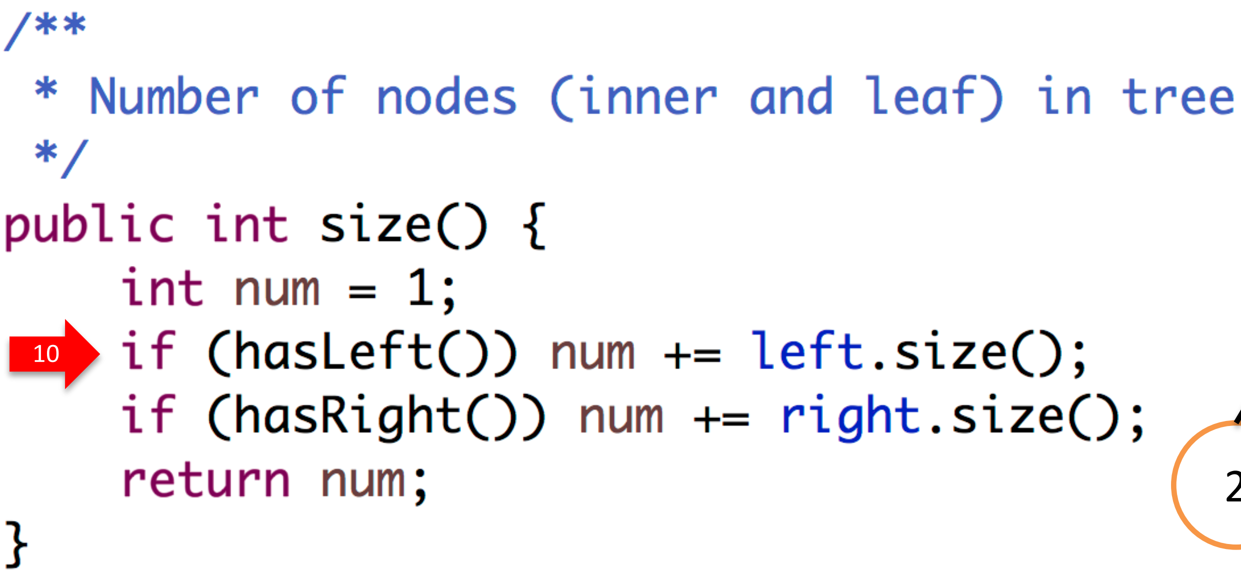
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
        int num = 1;
10 ⇨    if (hasLeft()) num += left.size();
        if (hasRight()) num += right.size();
5 ➡     return num;
}
```



- **Node 5 is done**
- **Return 3 to root**

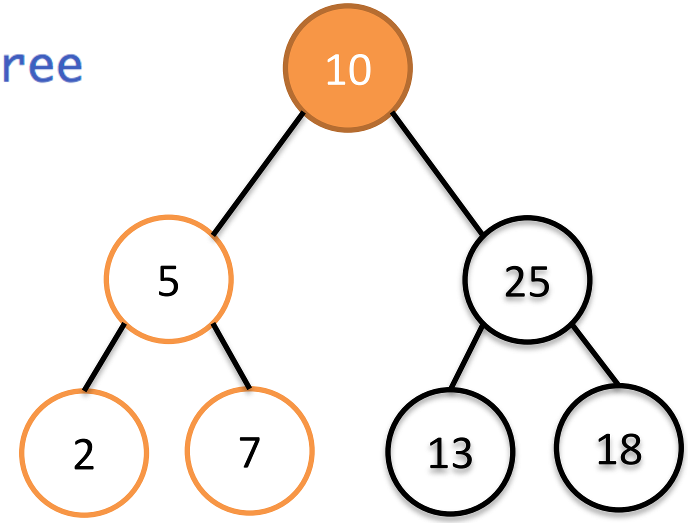
Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java



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

num=1+3=4



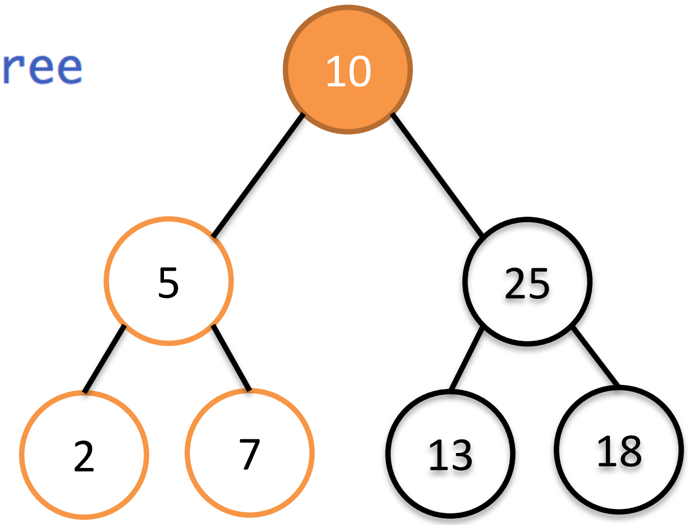
- **Increment num on root**

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;
}
```

num=4

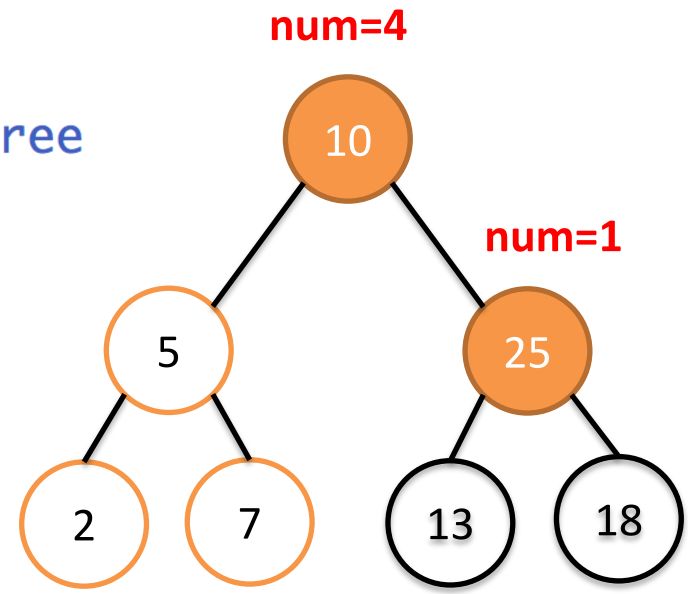


- Has right child
- Make recursive call on right child

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

BinaryTree.java

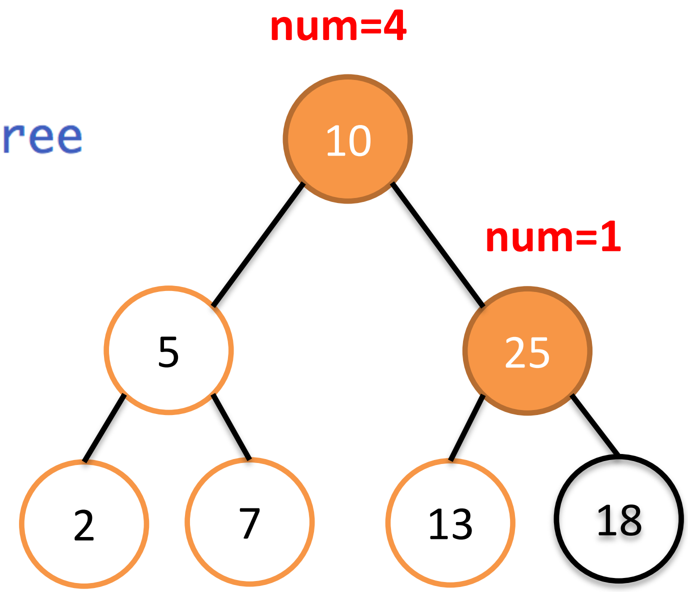
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
25 ➡ int num = 1;
     if (hasLeft()) num += left.size();
10 ⇨ if (hasRight()) num += right.size();
     return num;
}
```



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;
25      if (hasLeft()) num += left.size();
10      if (hasRight()) num += right.size();
        return num;
}
```

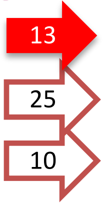


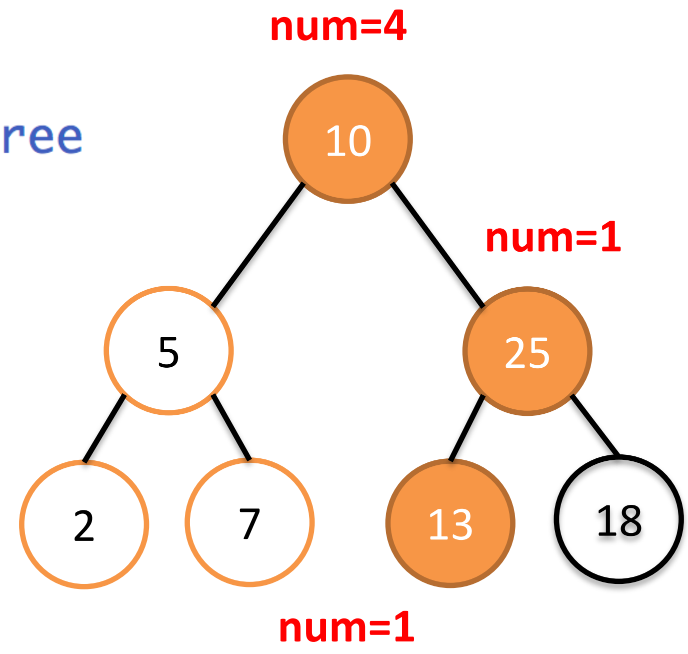
- **Has left child**
- **Make recursive call on left child**

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

BinaryTree.java

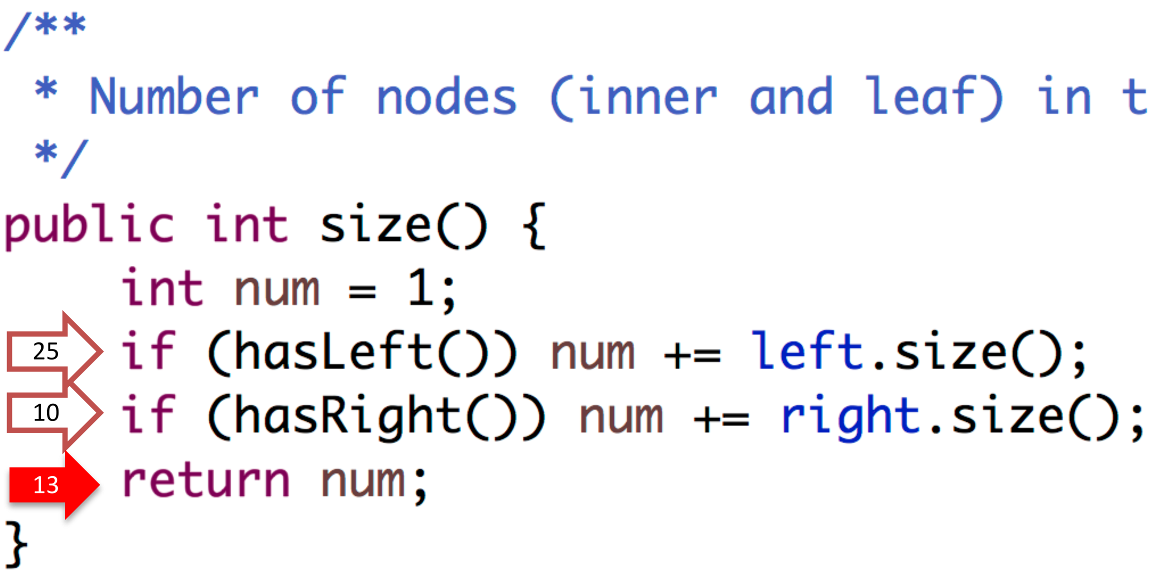
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



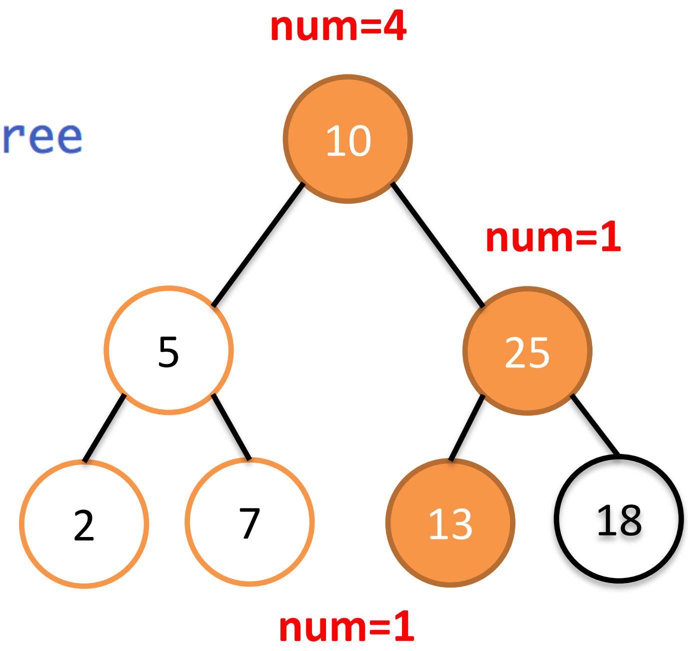


Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java



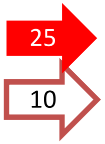
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



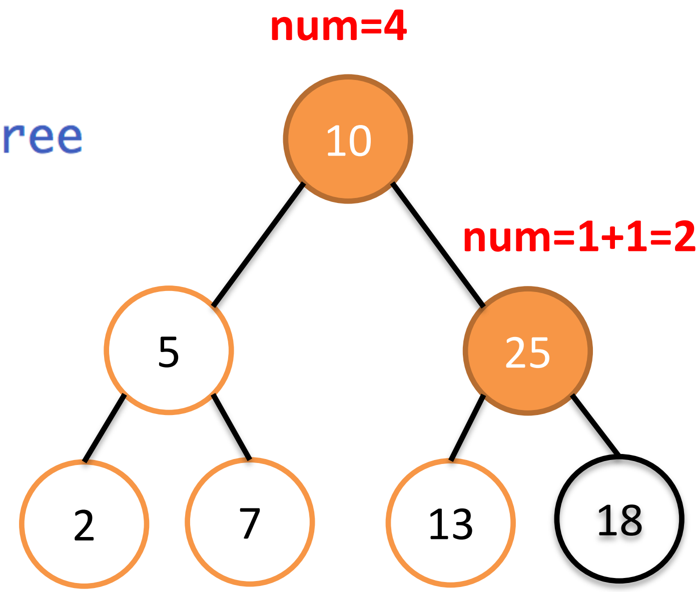
- **No children**
- **Return 1 back to Node 25**

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;
}
```



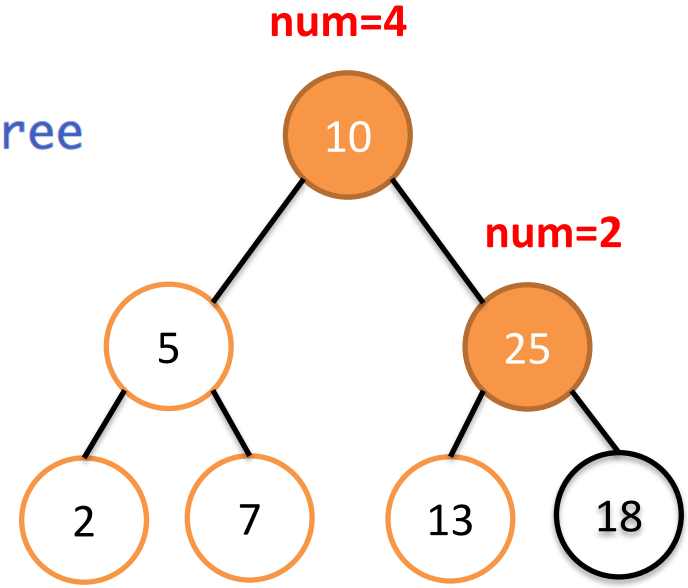
num=4

num=1+1=2

- 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();
    if (hasRight()) num += right.size();
    return num;
}
```

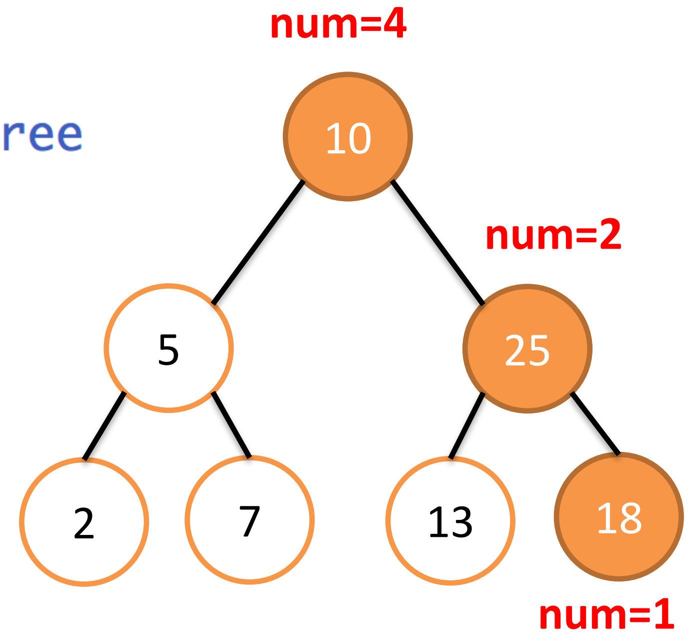


- **Has right child**
- **Make recursive call on right child**

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

BinaryTree.java

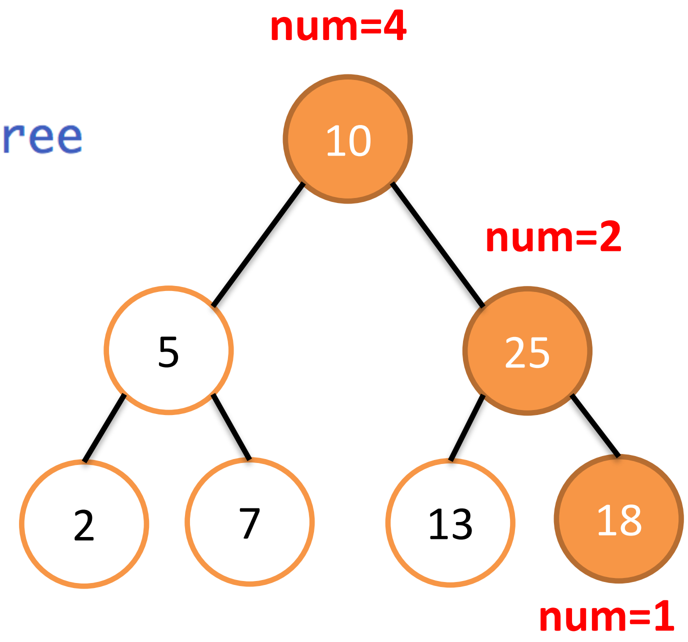
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
18  int num = 1;
    if (hasLeft()) num += left.size();
25  if (hasRight()) num += right.size();
    return num;
}
```



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

BinaryTree.java

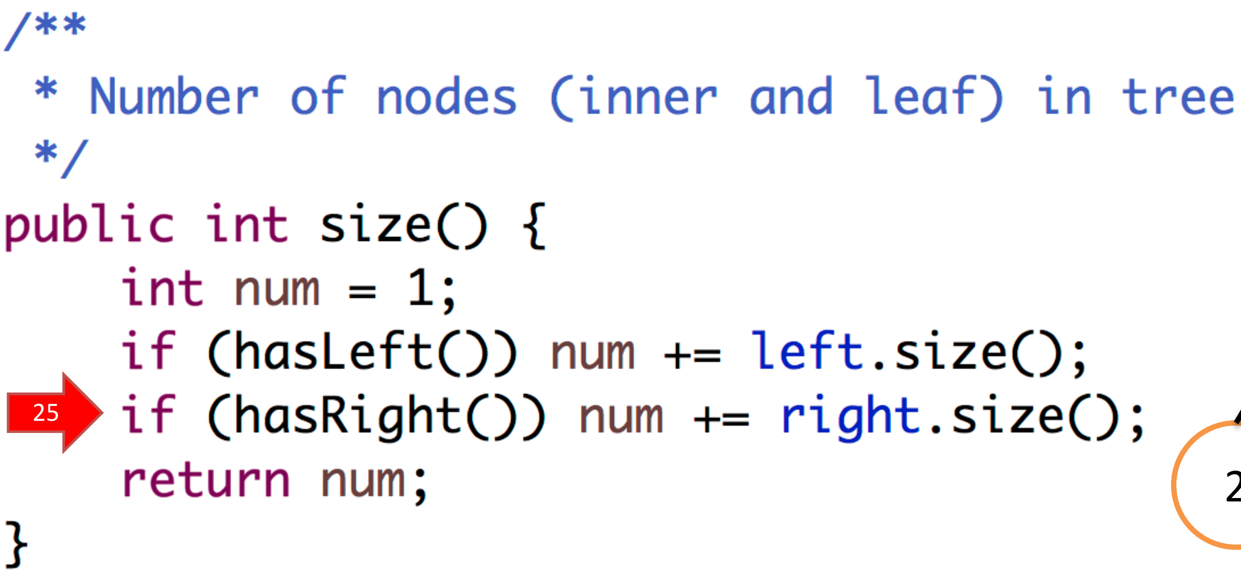
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
25  if (hasRight()) num += right.size();
18  return num;
}
```



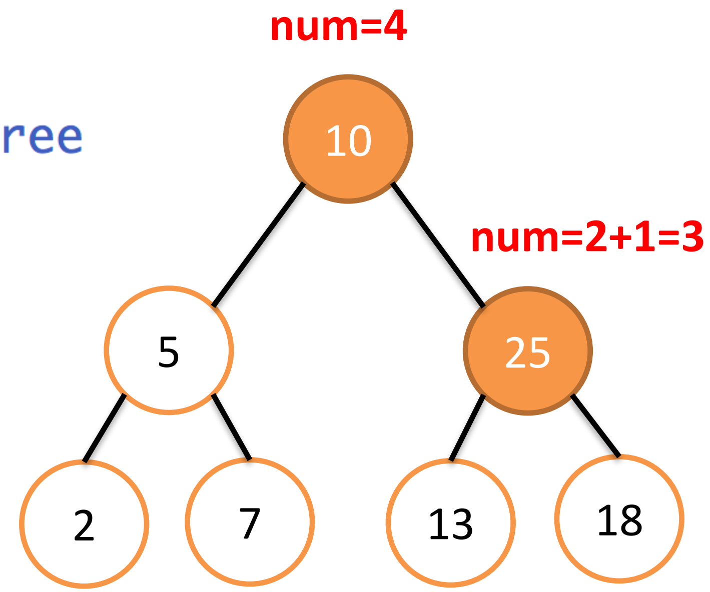
- **No children**
- **Return 1 to Node 25**

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

BinaryTree.java



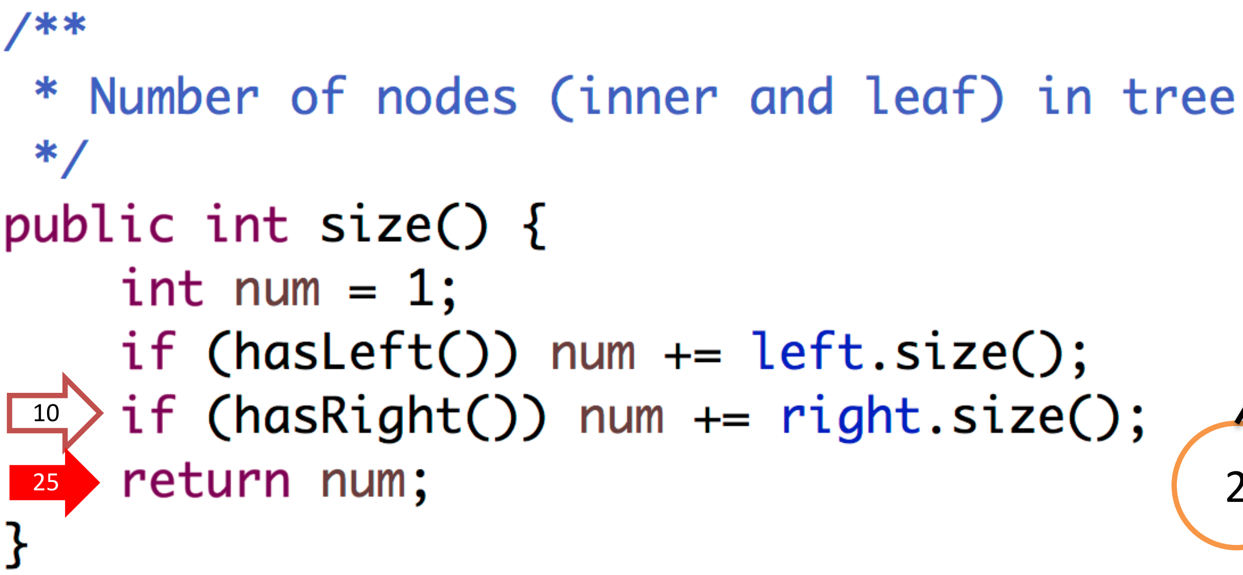
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



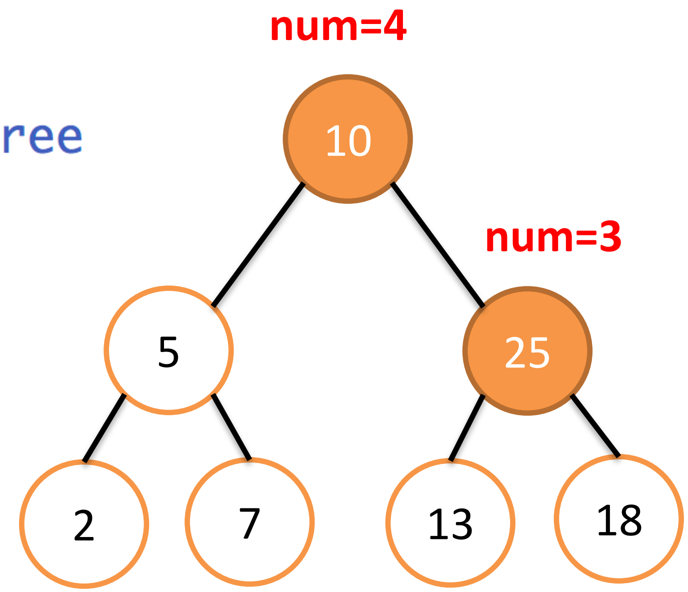
- **Increment num on Node 25**

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

BinaryTree.java



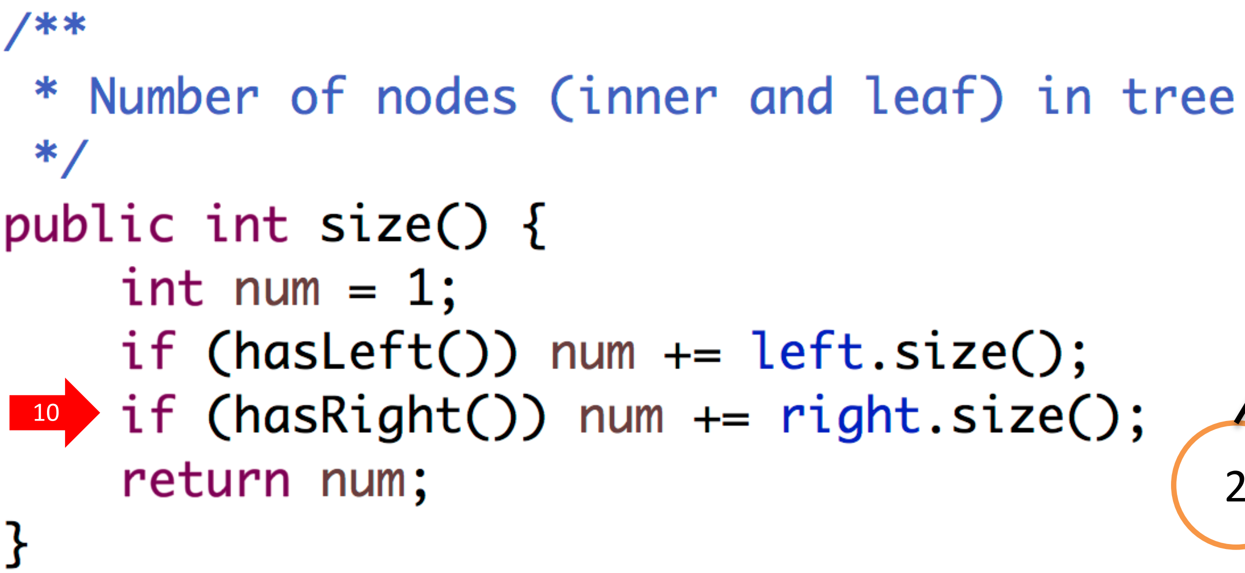
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



- **Node 25 is done**
- **Return 3 back to root**

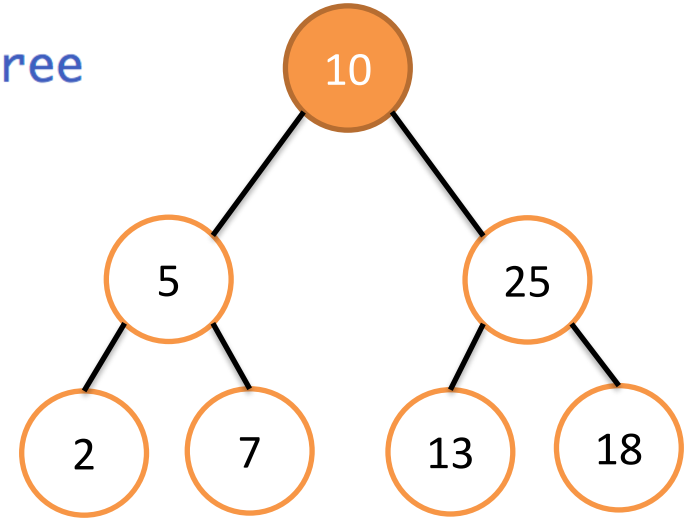
Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java



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

num=4+3=7



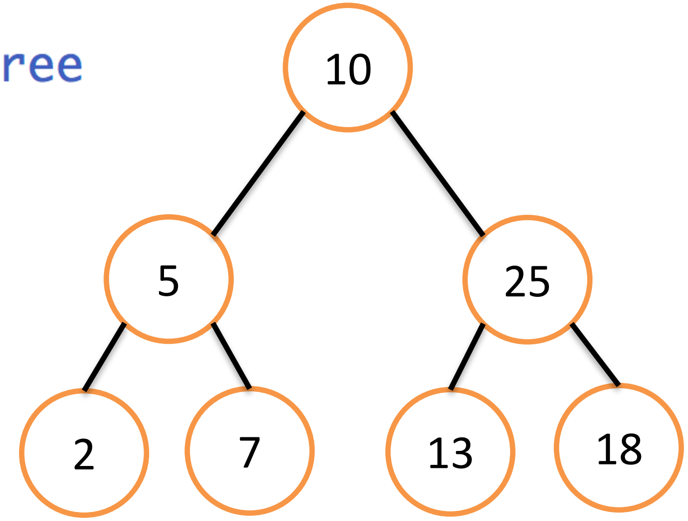
Increment num on root

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();
    if (hasRight()) num += right.size();
10 ➡ return num;
}
```

num=7

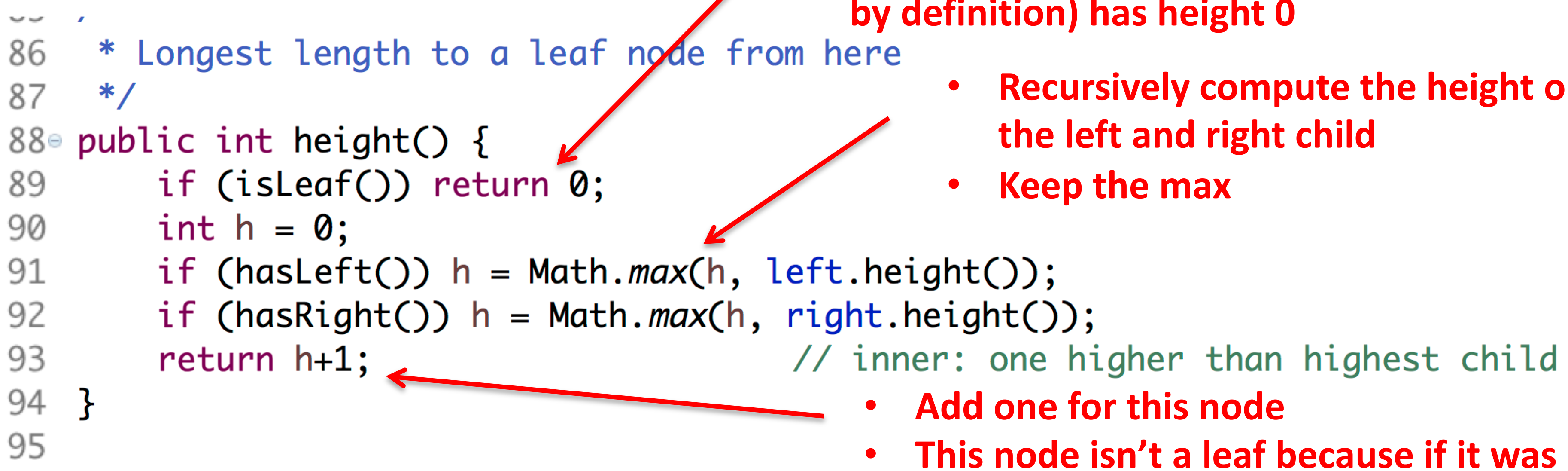


Done!
Return 7

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

BinaryTree.java

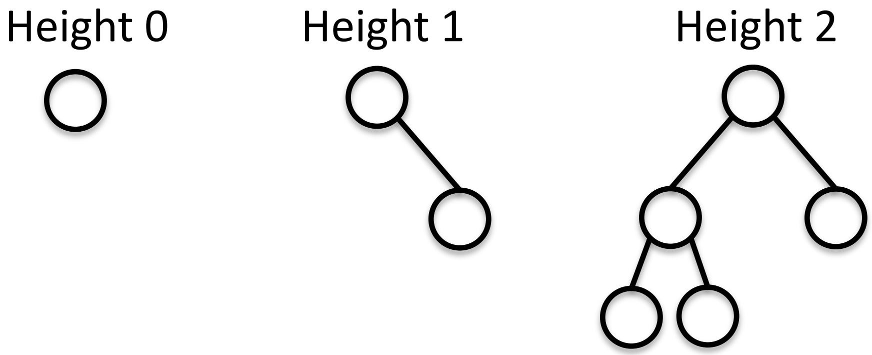
- **Height is the number of edges on the *longest* path from root to leaf**
- **By convention, a tree with one node (a leaf by definition) has height 0**



```
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;                          // inner: one higher than highest child
94 }
95
```

- **Recursively compute the height on the left and right child**
- **Keep the max**

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



BinaryTree.java – equalsTree

ANNOTATED SLIDES

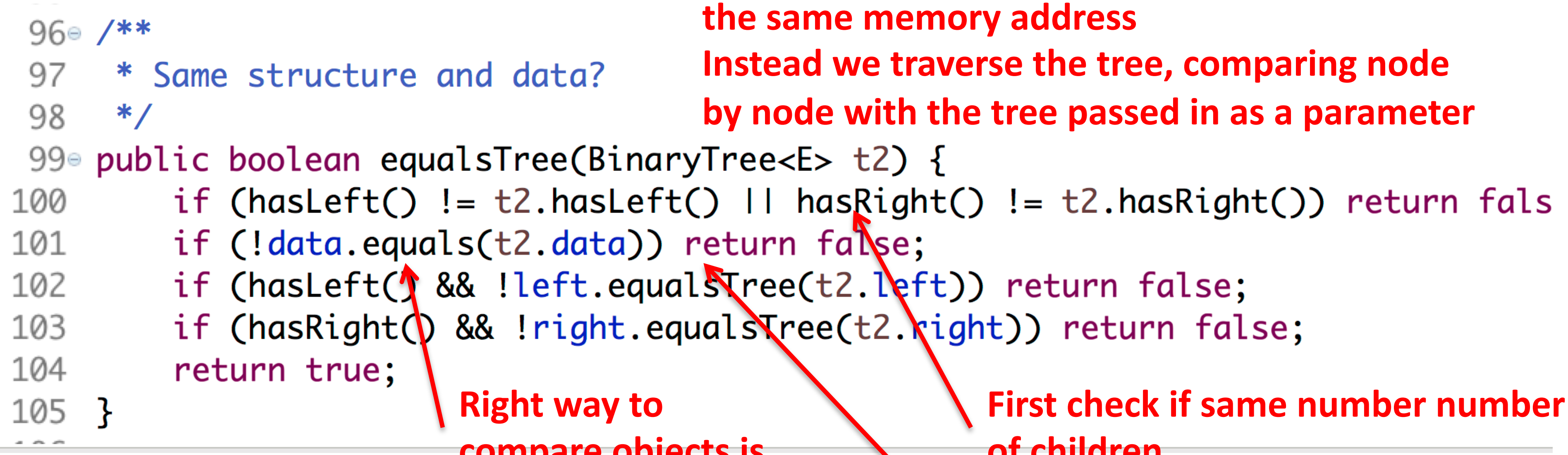
equalsTree() uses recursion to see if two trees have same data and structure

BinaryTree.java

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



```
/**
 * Same structure and data?
 */
public boolean equalsTree(BinaryTree<E> t2) {
    if (hasLeft() != t2.hasLeft() || hasRight() != t2.hasRight()) return fals
    if (!data.equals(t2.data)) return false;
    if (hasLeft() && !left.equalsTree(t2.left)) return false;
    if (hasRight() && !right.equalsTree(t2.right)) return false;
    return true;
}
```

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

First check if same number number of children

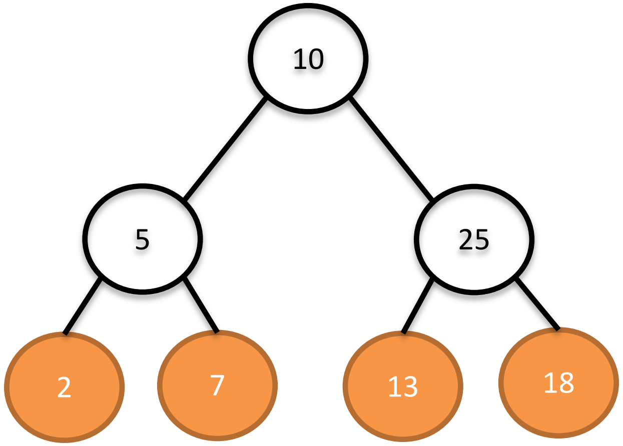
Next compare data is the same in each node

Trees are equal if same shape and same data

Finally, ask each child to compare itself

ACCUMULATORS PATTERN

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)

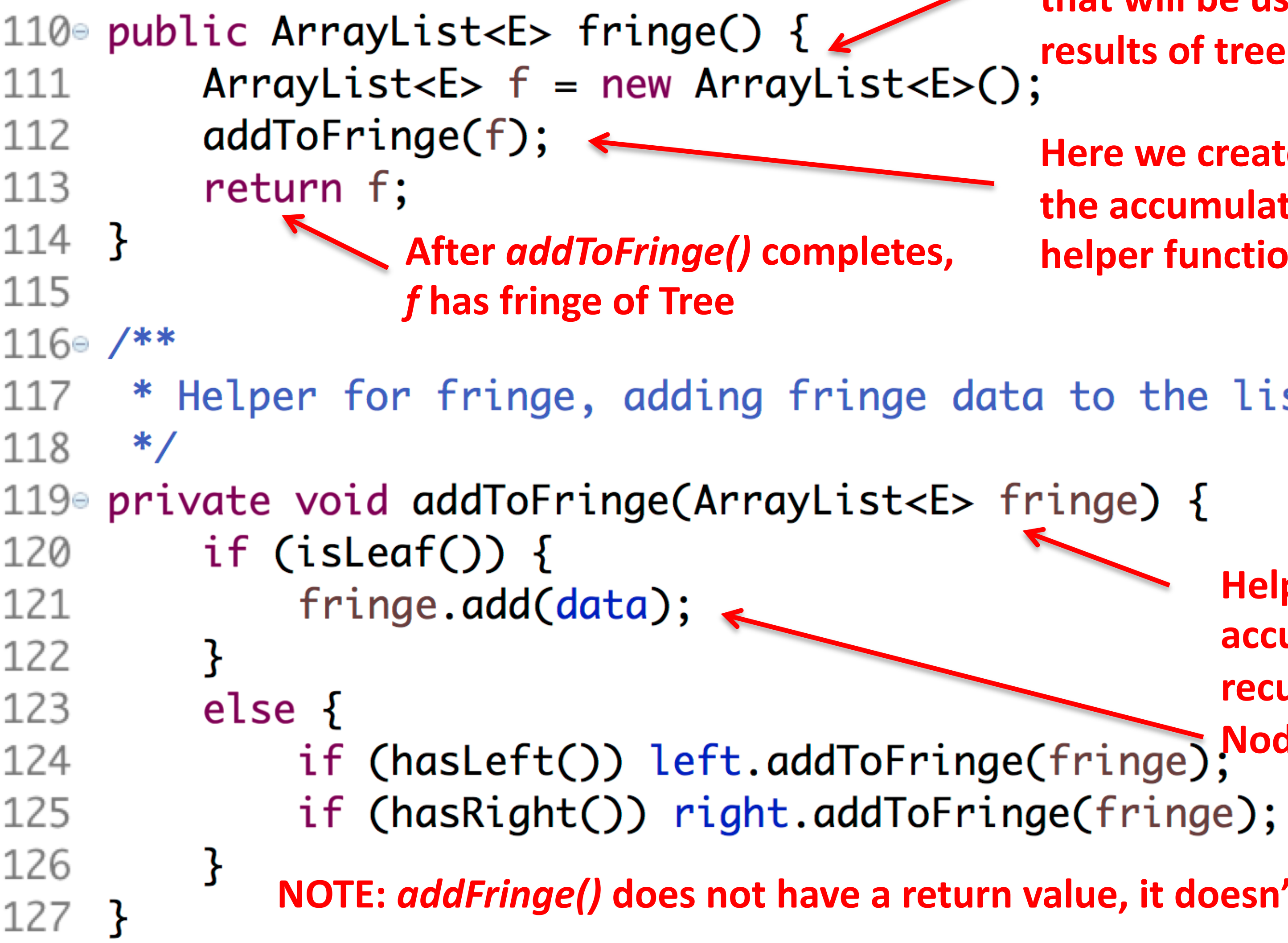
An accumulator keeps track of a variable during recursion

BinaryTree.java – fringe

ANNOTATED SLIDES

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

BinaryTree.java



```java
public ArrayList<E> fringe() {
    ArrayList<E> f = new ArrayList<E>();
    addToFringe(f);
    return f;
}

/**
 * Helper for fringe, adding fringe data to the list
 */
private void addToFringe(ArrayList<E> fringe) {
    if (isLeaf()) {
        fringe.add(data);
    }
    else {
        if (hasLeft()) left.addToFringe(fringe);
        if (hasRight()) right.addToFringe(fringe);
    }
}
```

***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

After *addToFringe()* completes, *f* has fringe of Tree

Helper function uses accumulator during recursion

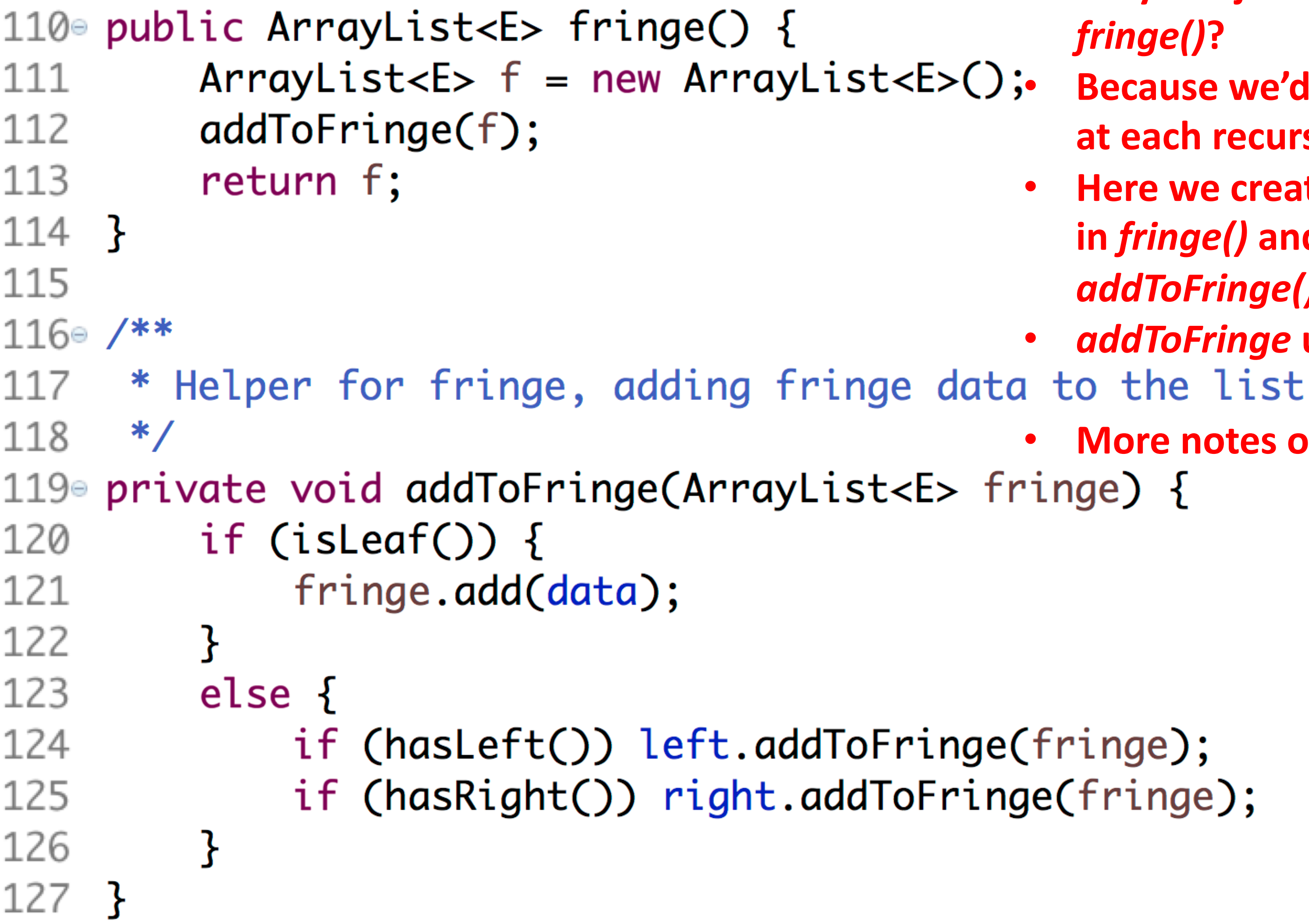
Node data added to fringe if leaf

Descend recursively

NOTE: *addFringe()* does not have a return value, it doesn't need one!

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

BinaryTree.java



```
public ArrayList<E> fringe() {
    ArrayList<E> f = new ArrayList<E>();
    addToFringe(f);
    return f;
}

/**
 * Helper for fringe, adding fringe data to the list
 */
private void addToFringe(ArrayList<E> fringe) {
    if (isLeaf()) {
        fringe.add(data);
    }
    else {
        if (hasLeft()) left.addToFringe(fringe);
        if (hasRight()) right.addToFringe(fringe);
    }
}
```

- **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 as it goes**
- **More notes on course web page**

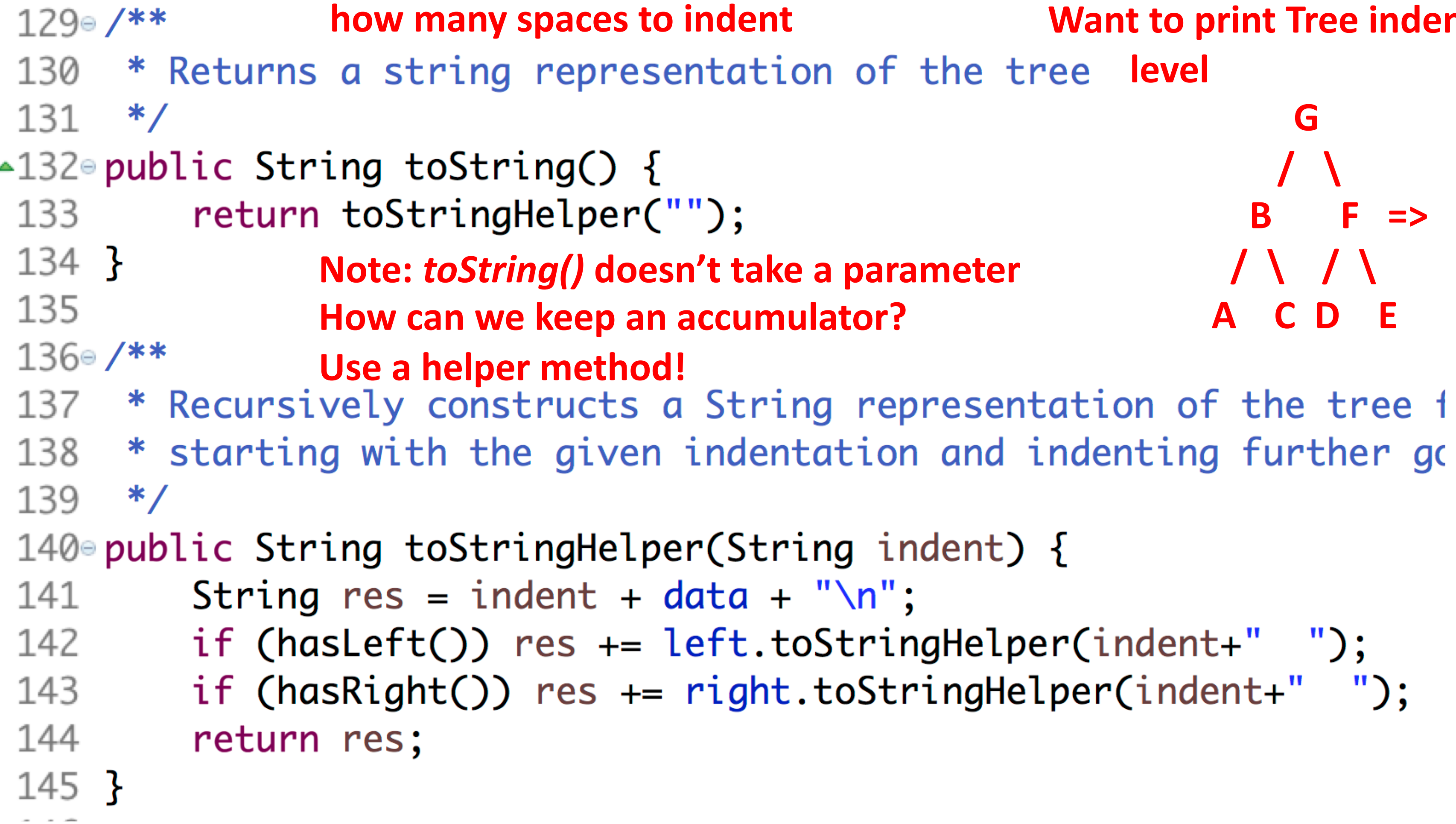
BinaryTree.java – toString

ANNOTATED SLIDES

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

BinaryTree.java

Idea: keep an accumulator of how many spaces to indent



```
/**
 * Returns a string representation of the tree
 */
public String toString() {
    return toStringHelper("");
}

/**
 * Recursively constructs a String representation of the tree f
 * starting with the given indentation and indenting further go
 */
public String toStringHelper(String indent) {
    String res = indent + data + "\n";
    if (hasLeft()) res += left.toStringHelper(indent+"  ");
    if (hasRight()) res += right.toStringHelper(indent+"  ");
    return res;
}
```

Note: *toString()* doesn't take a parameter

How can we keep an accumulator?

Use a helper method!

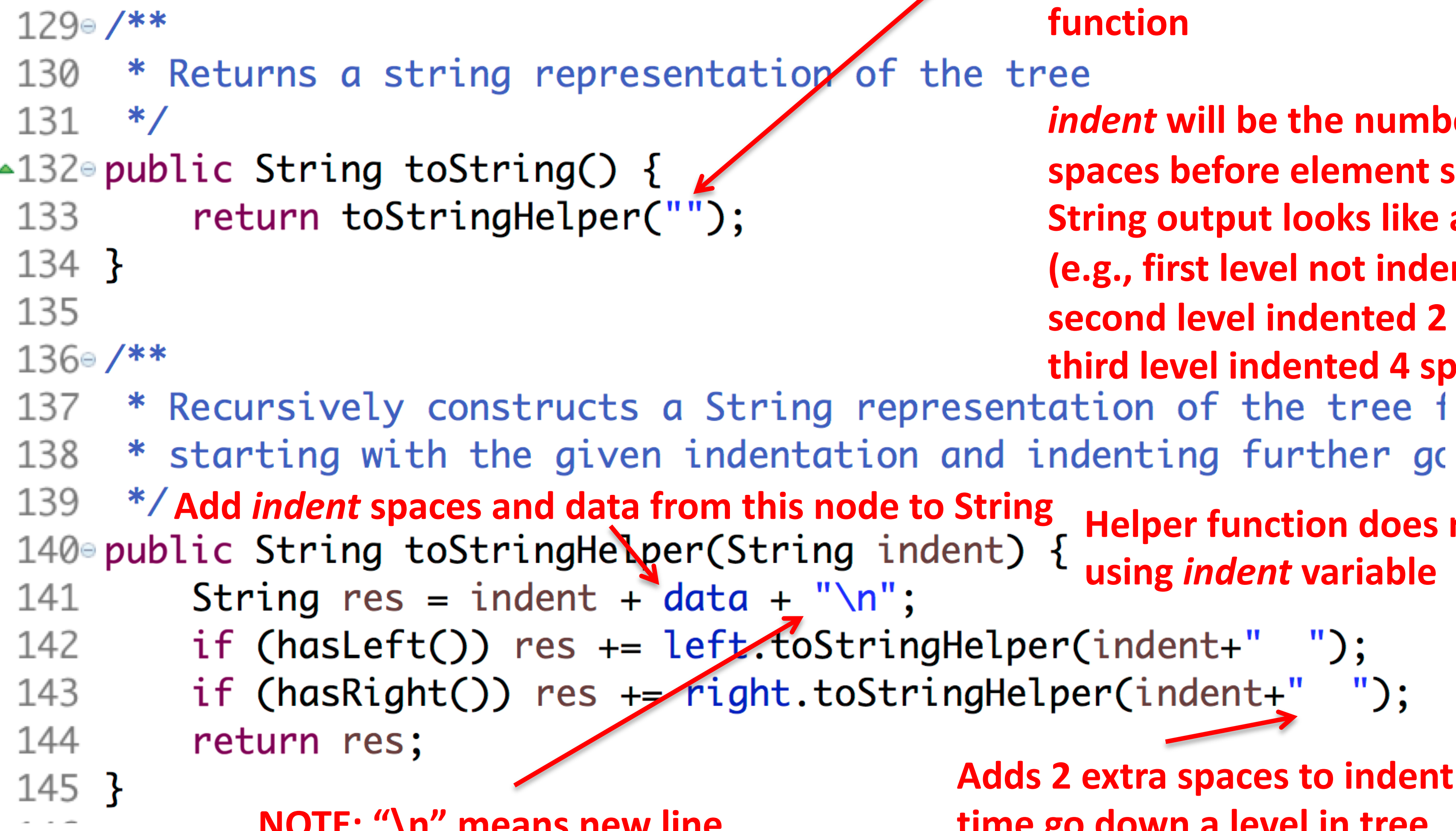
***toString()* called by Java if object is in println statemen**

Want to print Tree indented by level

```
        G              G
       / \               B
      B   F   =>           A
     / \ / \               C
    A  C D  E            F
                           D
                           E
```

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

BinaryTree.java



```java
/**
 * Returns a string representation of the tree
 */
public String toString() {
    return toStringHelper("");
}

/**
 * Recursively constructs a String representation of the tree f
 * starting with the given indentation and indenting further gc
 */
public String toStringHelper(String indent) {
    String res = indent + data + "\n";
    if (hasLeft()) res += left.toStringHelper(indent+"  ");
    if (hasRight()) res += right.toStringHelper(indent+"  ");
    return res;
}
```

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, third level indented 4 spaces…)

Add *indent* spaces and data from this node to String

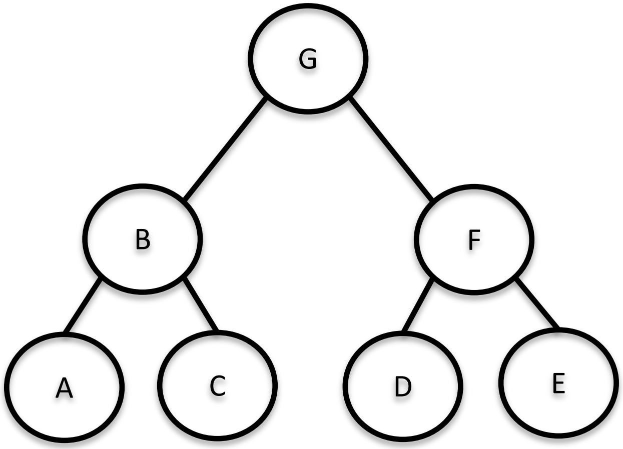
Helper function does recursion using *indent* variable

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

NOTE: “\n” means new line

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

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*

preorder

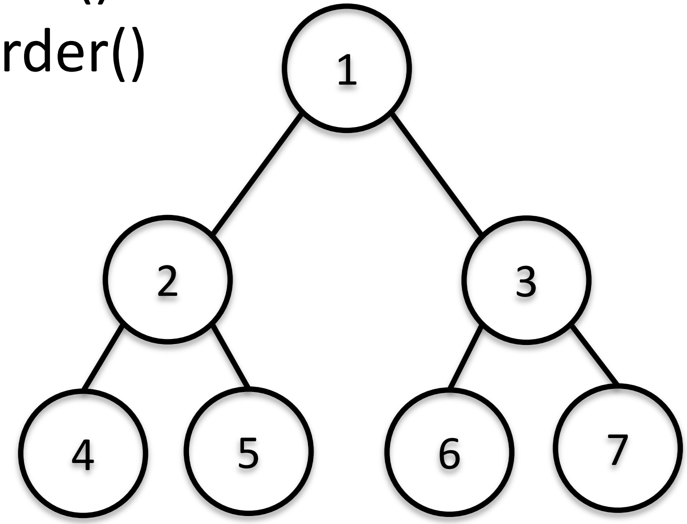
DIFFERENT TREE TRAVERSALS

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

preorder()

```
    visit
    left.preorder()
    right.preorder()
```

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



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()

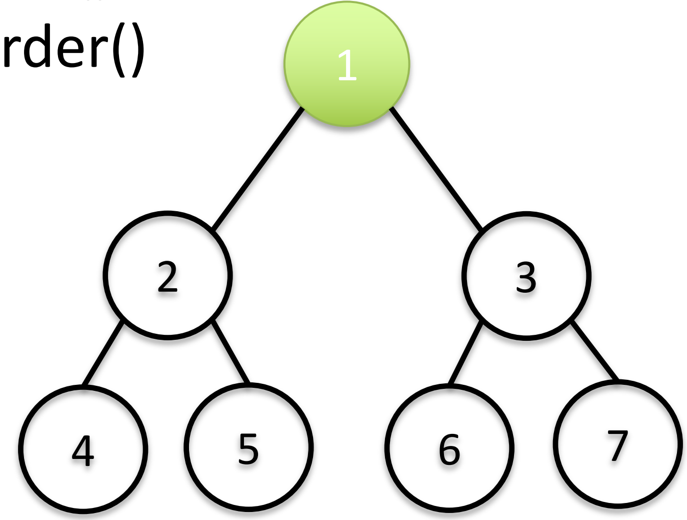
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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

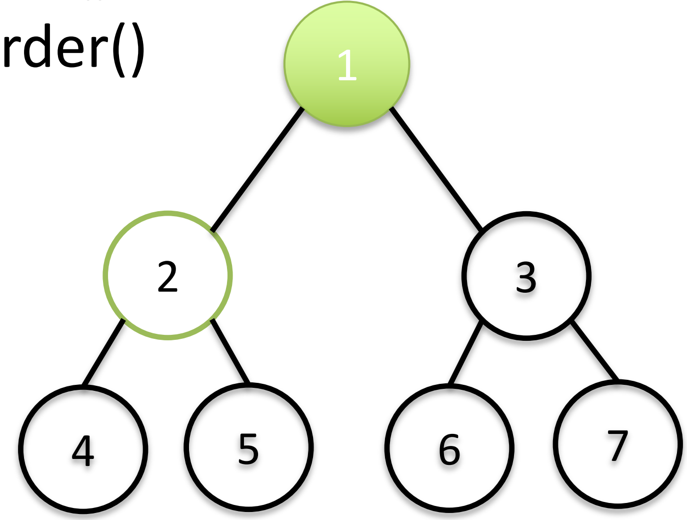
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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

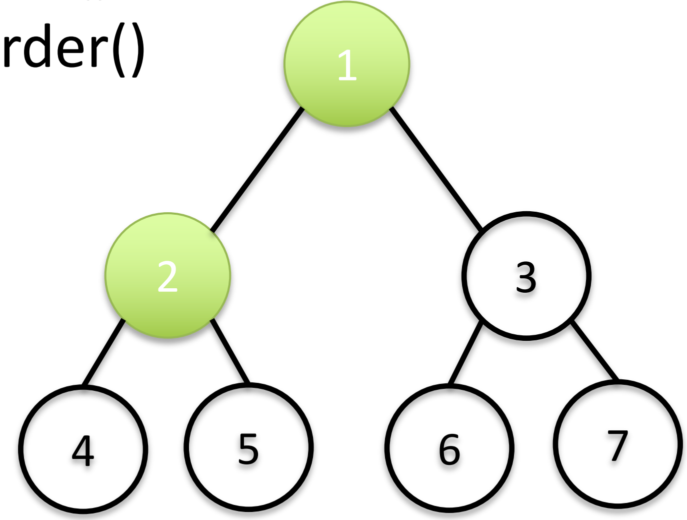
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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

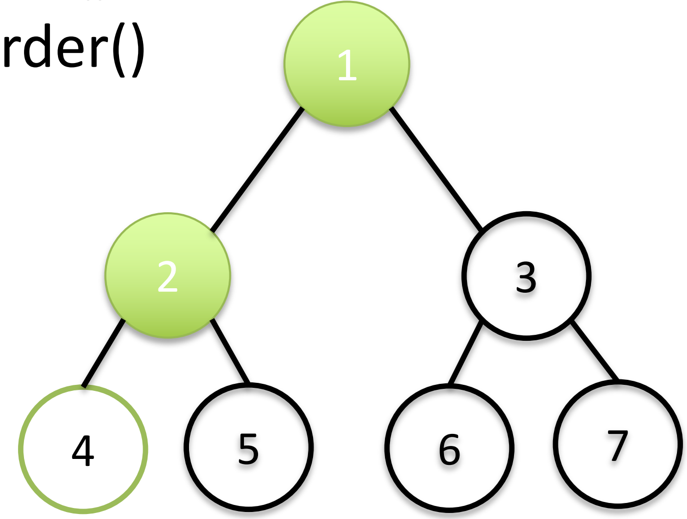
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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()

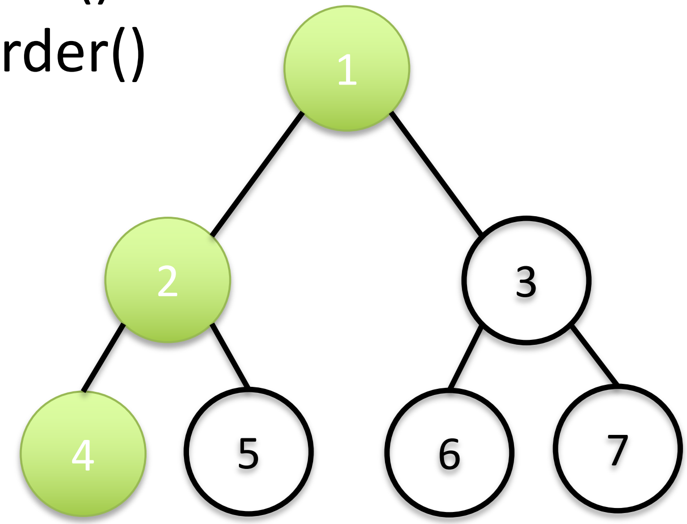
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

Examples:

File directory structure

Table of contents in book

toString()

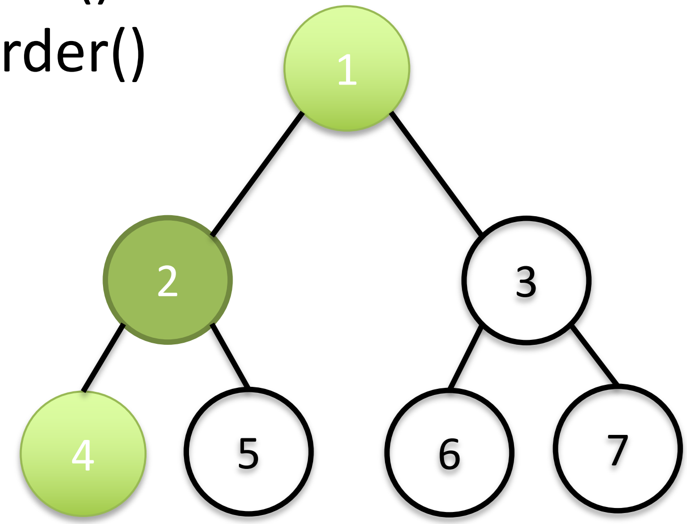


Visited

1, 2, 4

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

```
preorder()
    visit
    left.preorder()
    right.preorder()
```



Visited

1, 2, 4

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

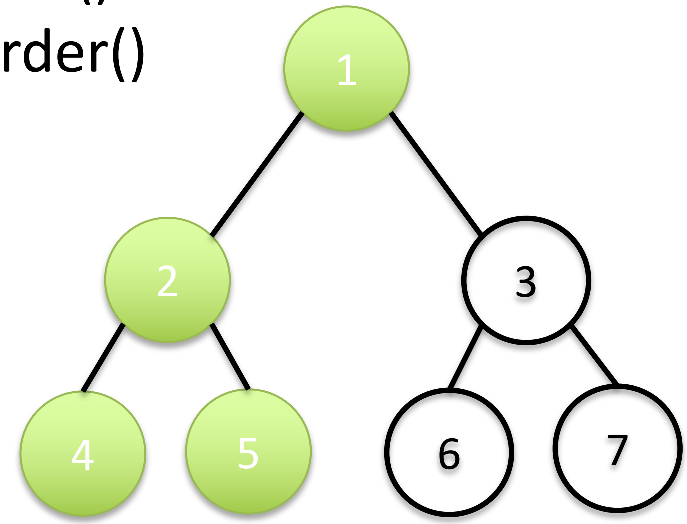
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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

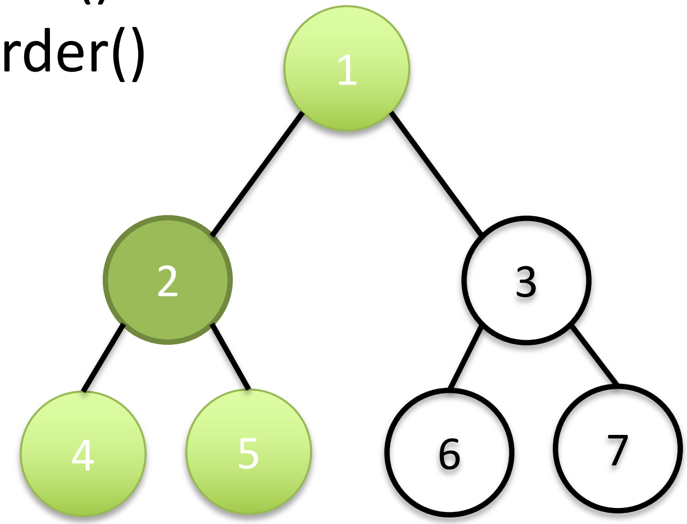
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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

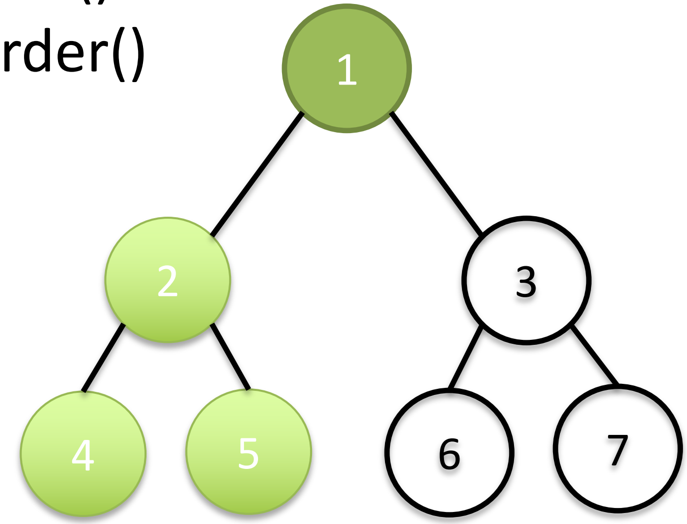
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

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

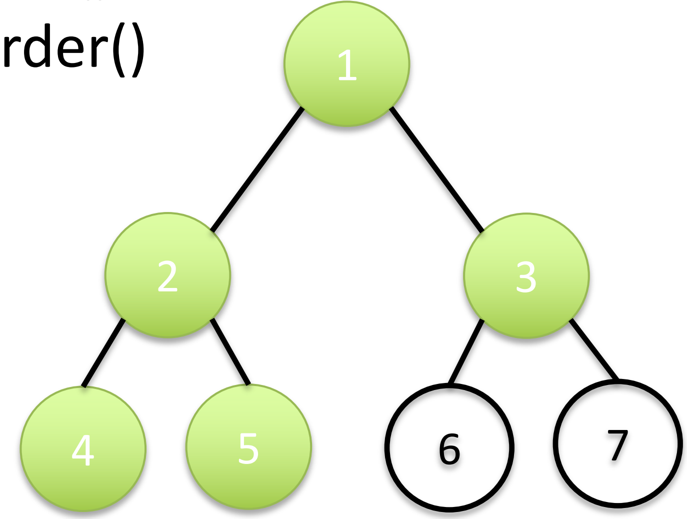
```
preorder()
    visit
    left.preorder()
    right.preorder()
```

Examples:

File directory structure

Table of contents in book

toString()



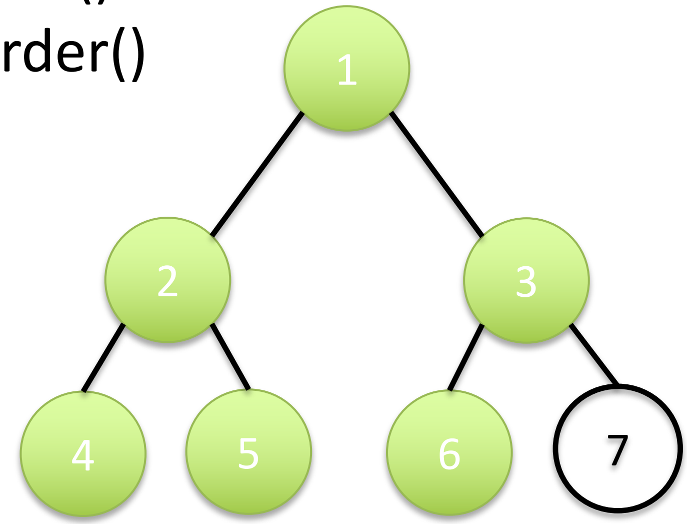
Visited

1, 2, 4, 5, 3

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

preorder()

```
    visit
    left.preorder()
    right.preorder()
```



Visited

1, 2, 4, 5, 3, 6

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()

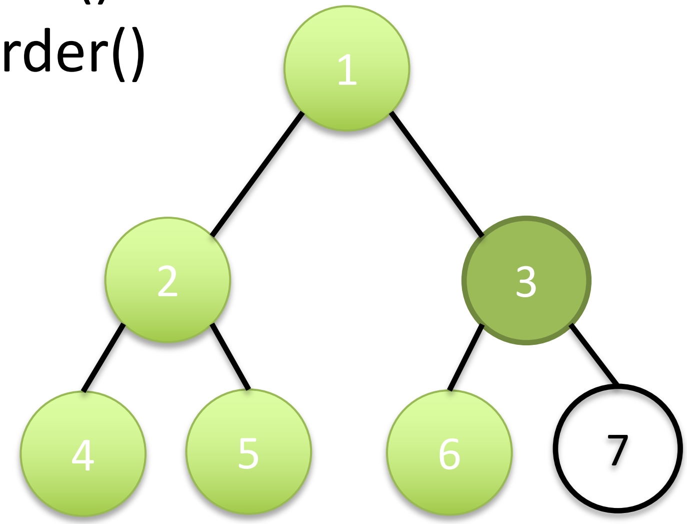
```
    visit
    left.preorder()
    right.preorder()
```

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()

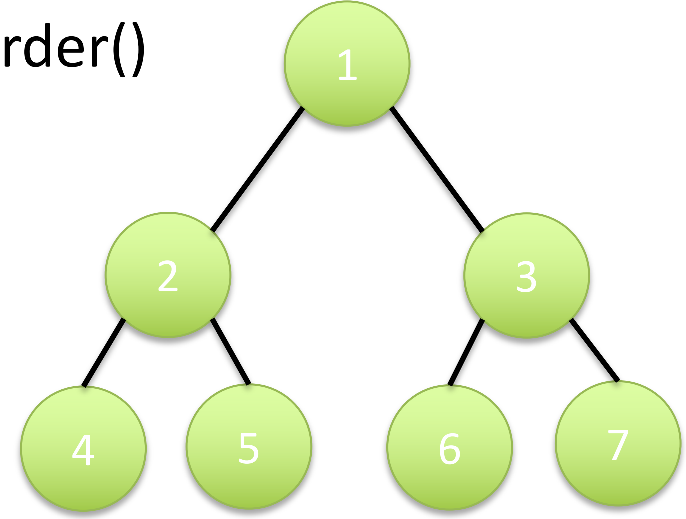
```
    visit
    left.preorder()
    right.preorder()
```

Examples:

File directory structure

Table of contents in book

toString()



Visited

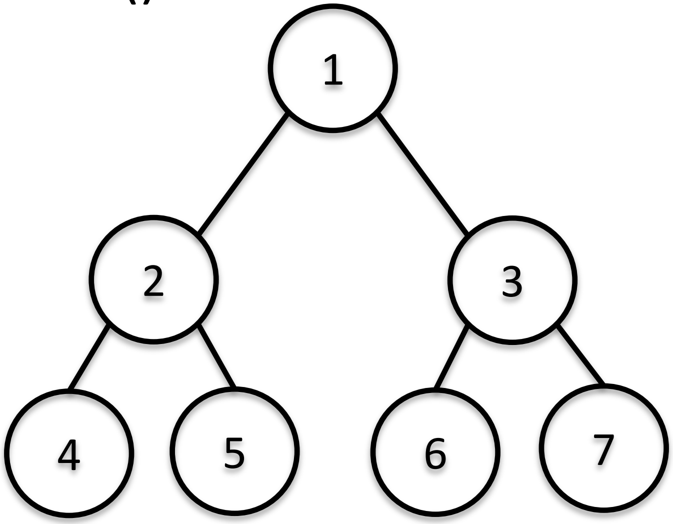
1, 2, 4, 5, 3, 6, 7

postorder

DIFFERENT TREE TRAVERSALS

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

```
postorder()
    left.postorder()
    right.postorder()
    visit
```

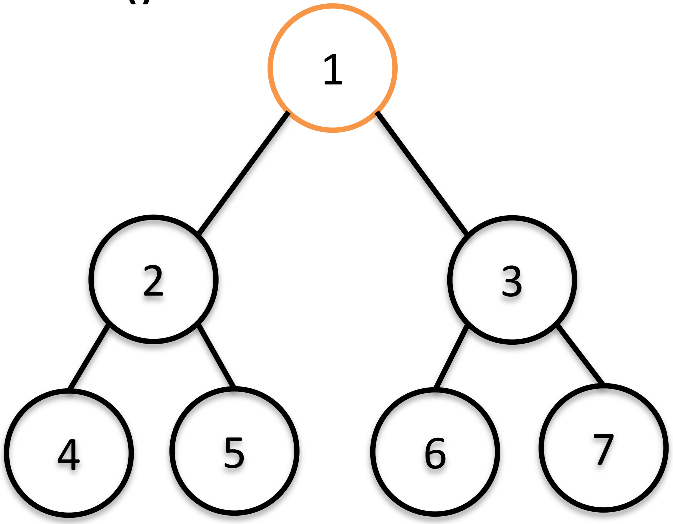


Example:

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()
    visit
```

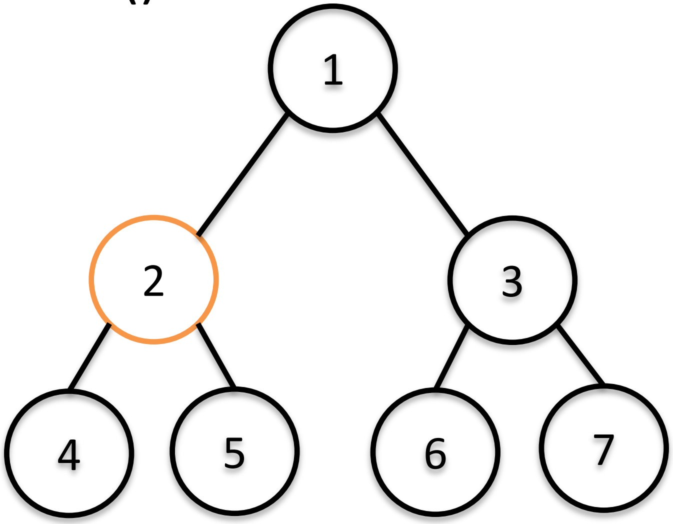


Example:

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()
    visit
```



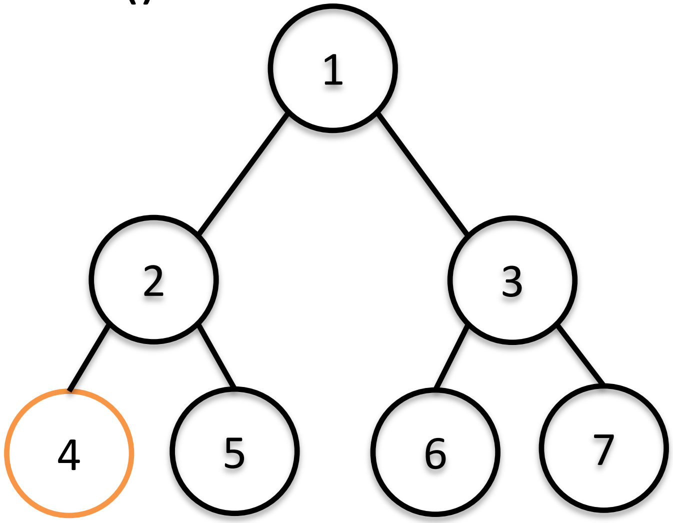
Example:

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()

```
postorder()
    left.postorder()
    right.postorder()
    visit
```



Example:

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

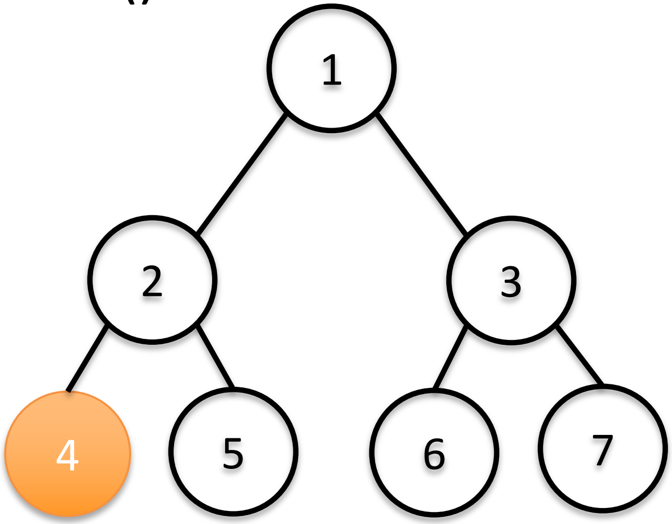
There are different ways to traverse a tree, depending on what needs to be done

postorder()

```
        left.postorder()
        right.postorder()
        visit
```

Example:

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

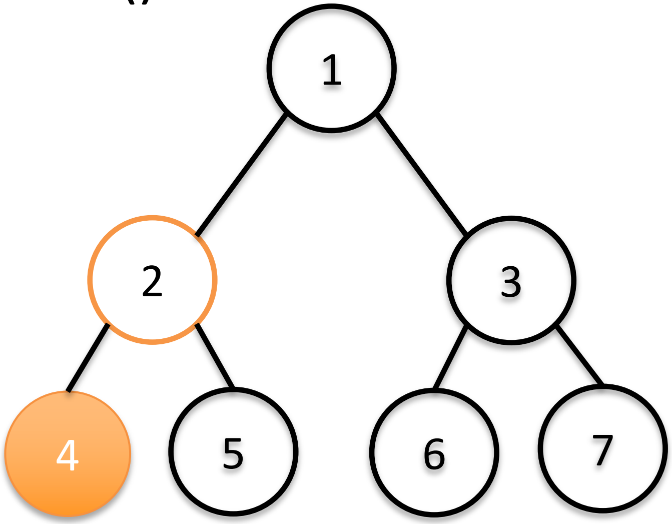


Visited

4

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

```
postorder()
    left.postorder()
    right.postorder()
    visit
```



Visited

4

Example:

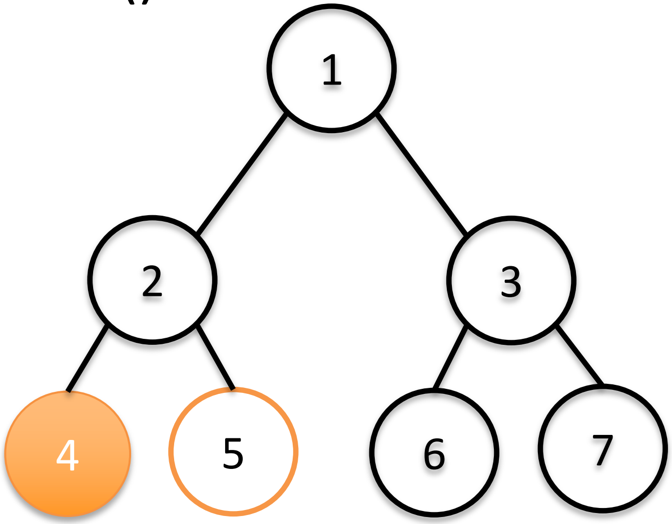
Compute disk space (not sure how many bytes in each directory until you search all children)

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

```
postorder()
        left.postorder()
        right.postorder()
        visit
```

Example:

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



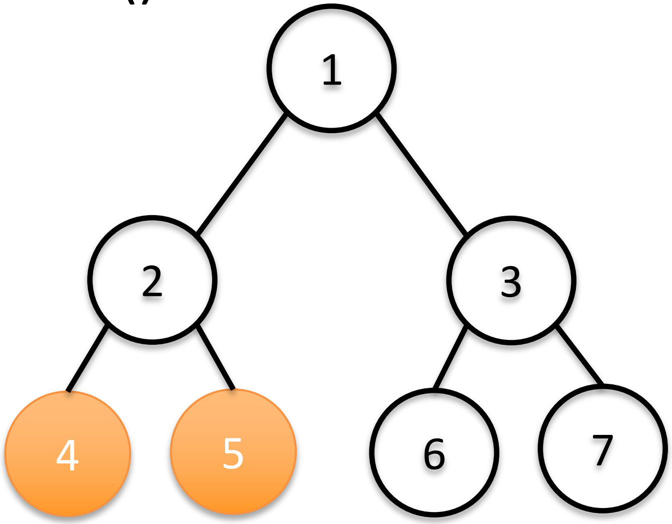
Visited

4

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

postorder()

```
postorder()
    left.postorder()
    right.postorder()
    visit
```



Visited

4, 5

Example:

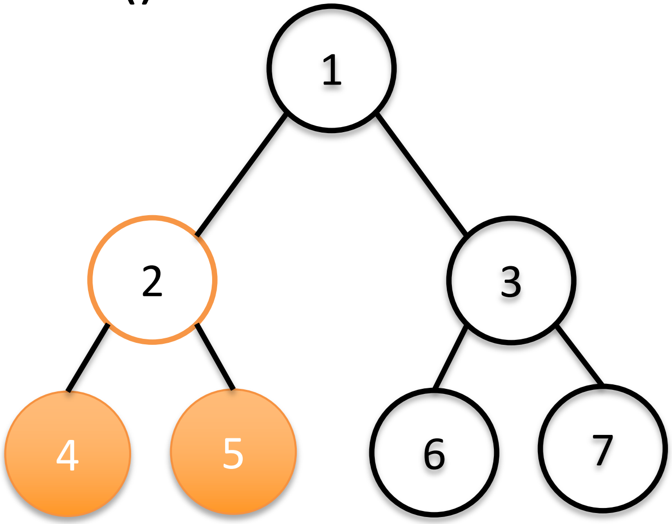
Compute disk space (not sure how many bytes in each directory until you search all children)

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

```
postorder()
    left.postorder()
    right.postorder()
    visit
```

Example:

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



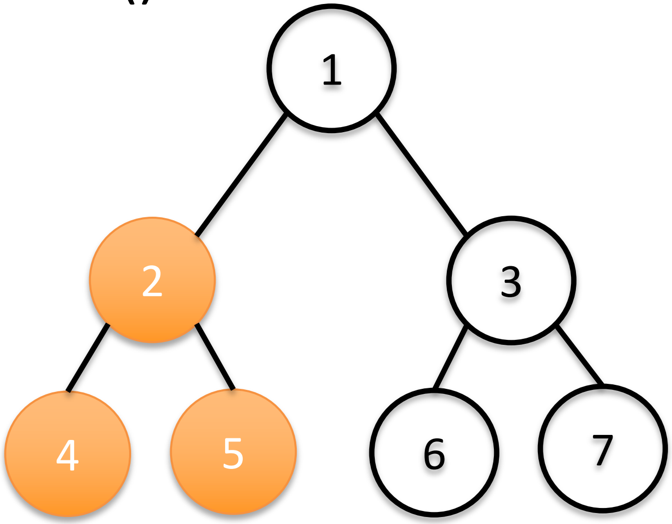
Visited

4, 5

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

postorder()

```
postorder()
    left.postorder()
    right.postorder()
    visit
```



Visited

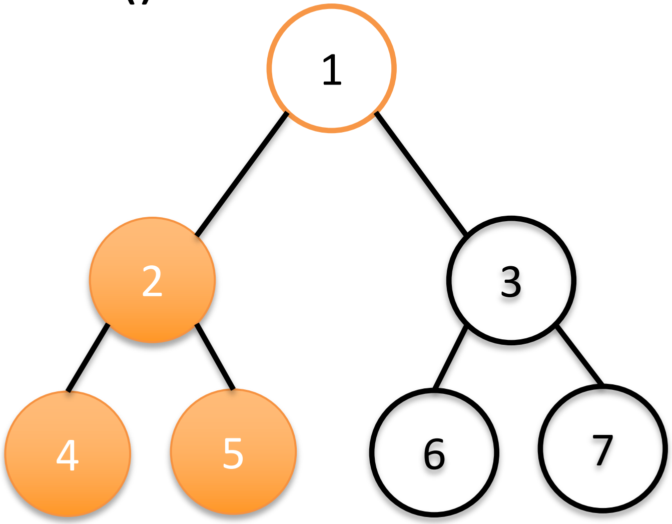
4, 5, 2

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()
        visit
```



Visited

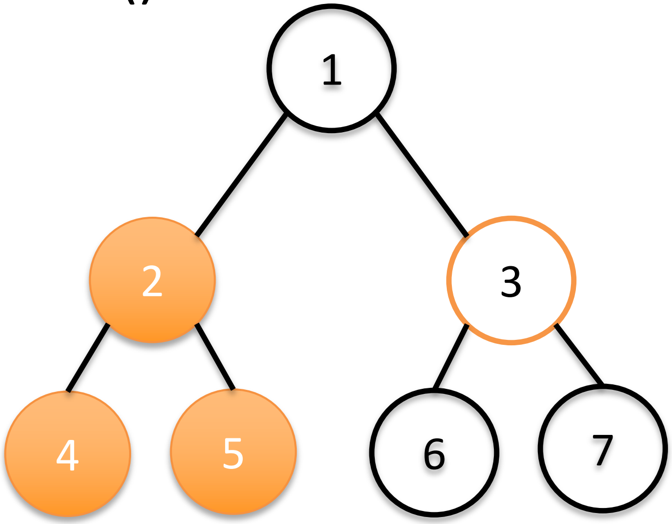
4, 5, 2

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()
    visit
```



Example:

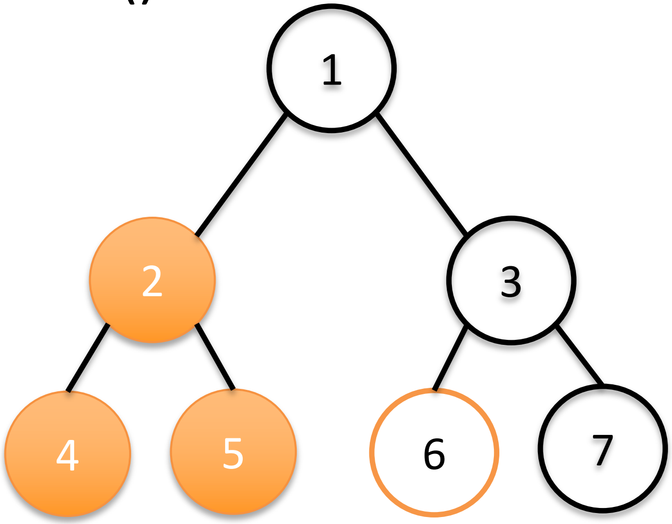
Compute disk space (not sure how many bytes in each directory until you search all children)

Visited

4, 5, 2

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

```
postorder()
        left.postorder()
        right.postorder()
        visit
```



Visited

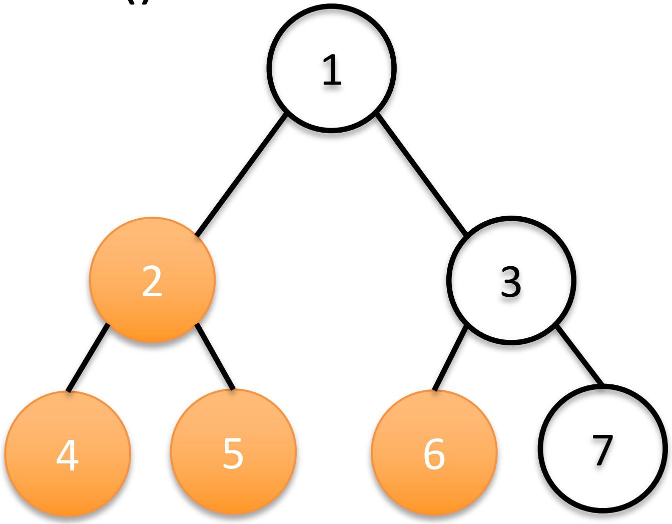
4, 5, 2

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()
    visit
```



Visited

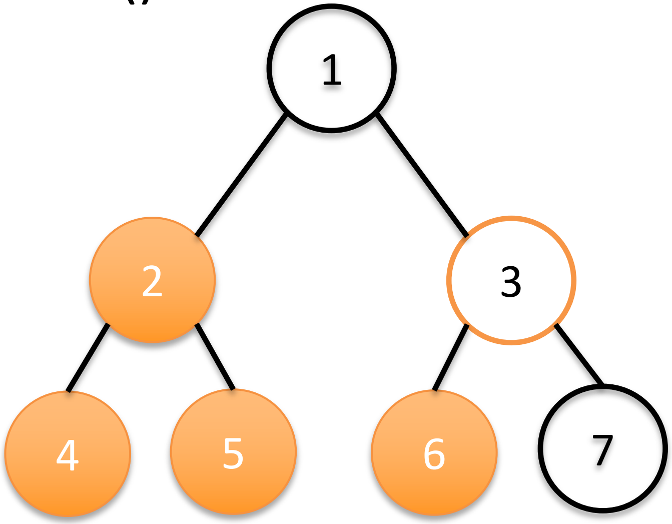
4, 5, 2, 6

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()
    visit
```



Visited

4, 5, 2, 6

Example:

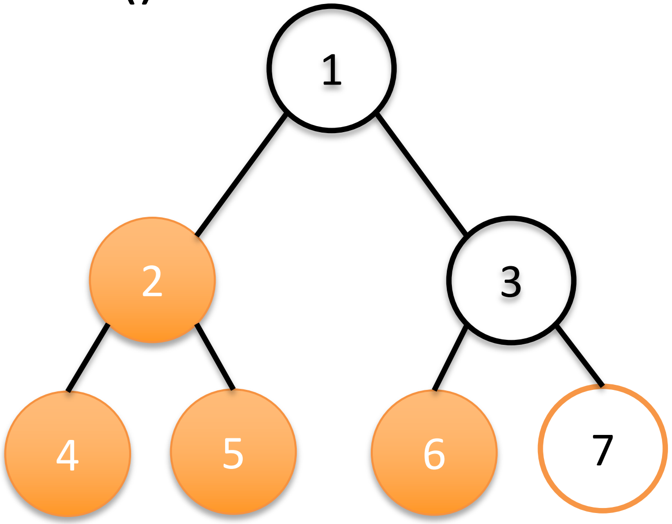
Compute disk space (not sure how many bytes in each directory until you search all children)

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

```
postorder()
    left.postorder()
    right.postorder()
    visit
```

Example:

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



Visited

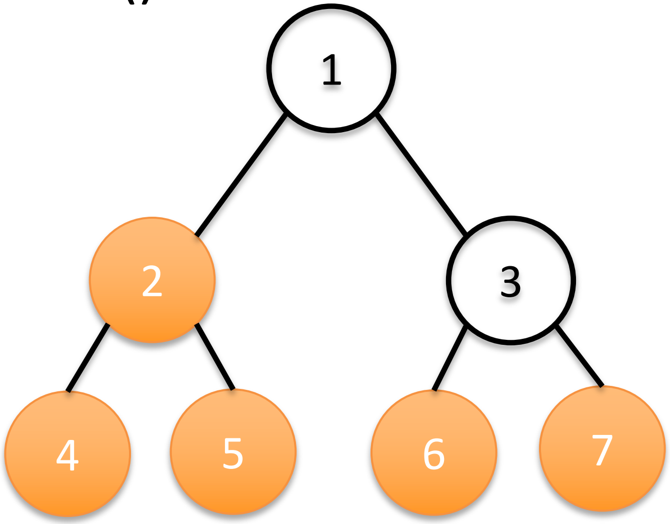
4, 5, 2, 6

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

```
postorder()
        left.postorder()
        right.postorder()
        visit
```

Example:

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

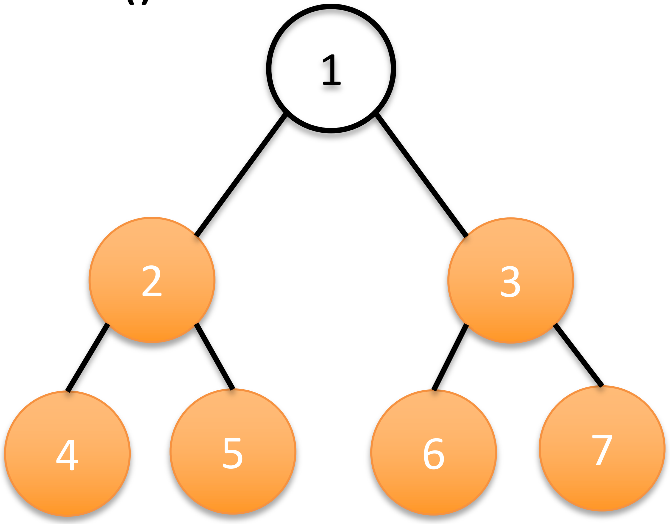


Visited

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()
    visit
```



Visited

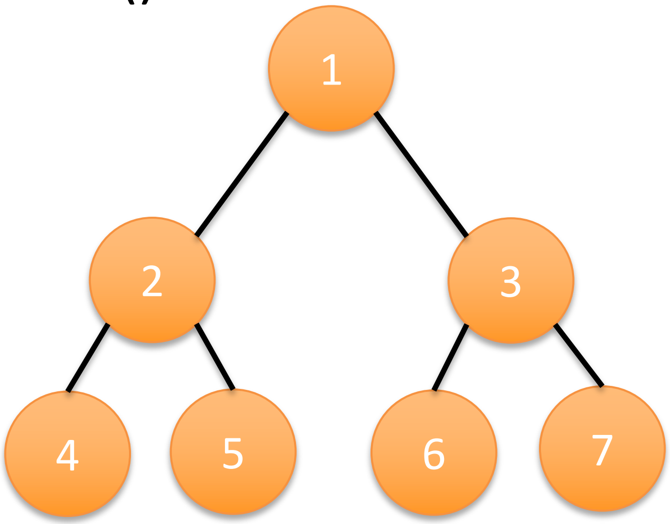
4, 5, 2, 6, 7, 3

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()
    visit
```



Visited

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

Example:

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

inorder

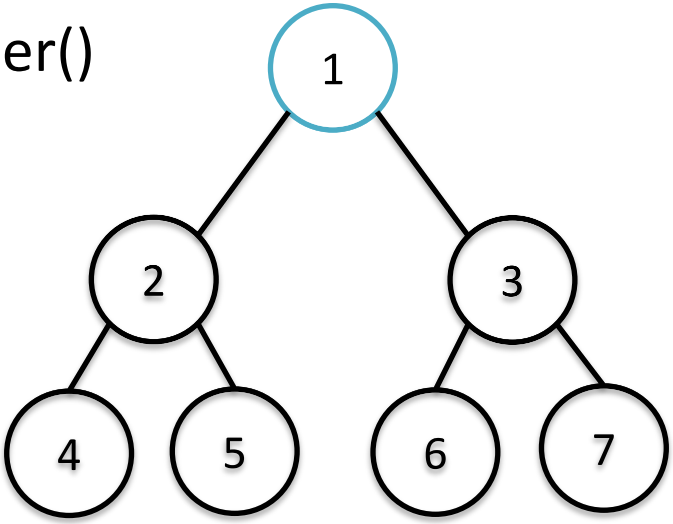
DIFFERENT TREE TRAVERSALS

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

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



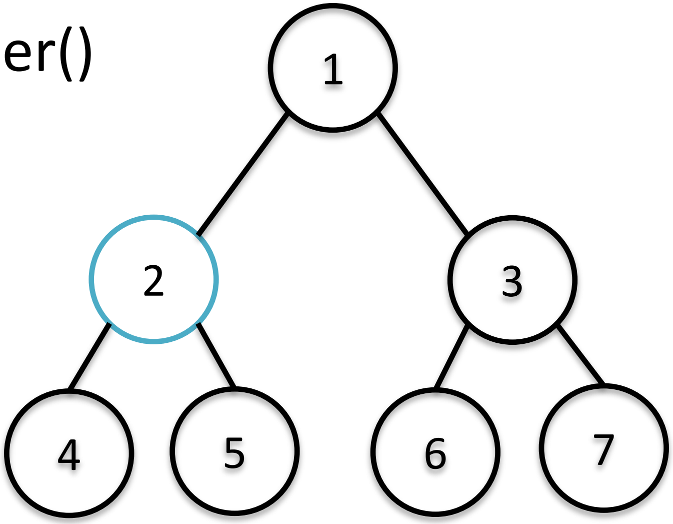
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



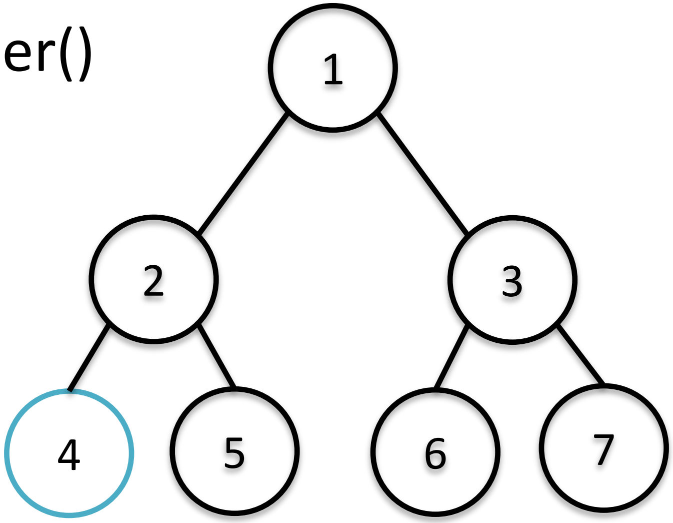
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree

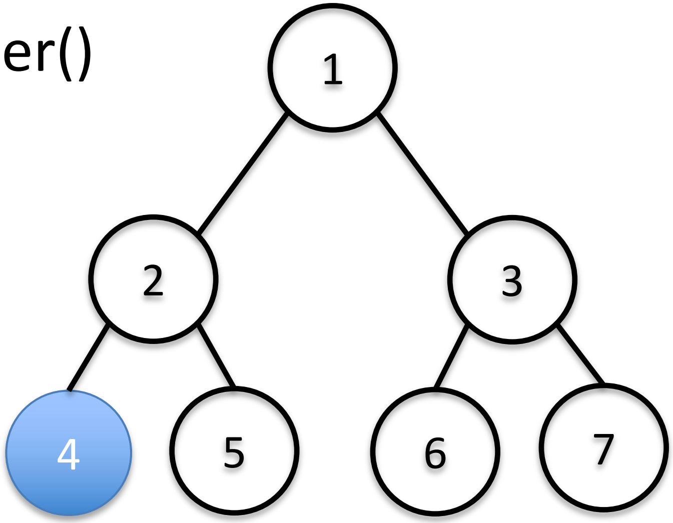


There are different ways to traverse a tree, depending on what needs to be done

```
inorder()
        left.inorder()
        visit
        right.inorder()
```

Example:

Drawing a tree



Visited

4

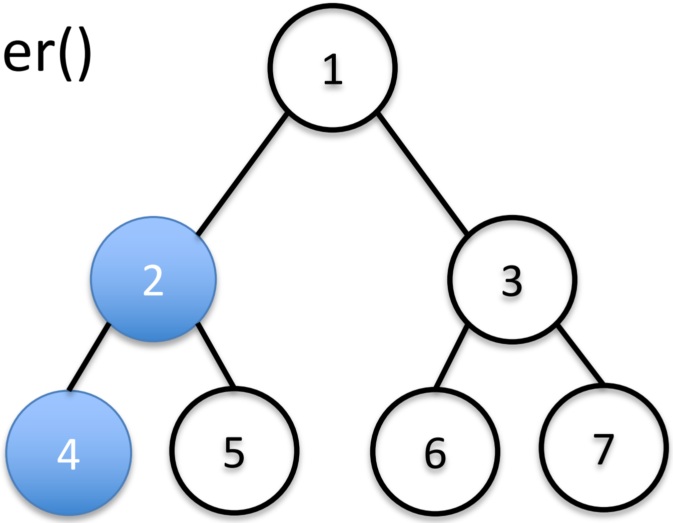
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
        left.inorder()
        visit
        right.inorder()
```

Example:

Drawing a tree



Visited

4, 2

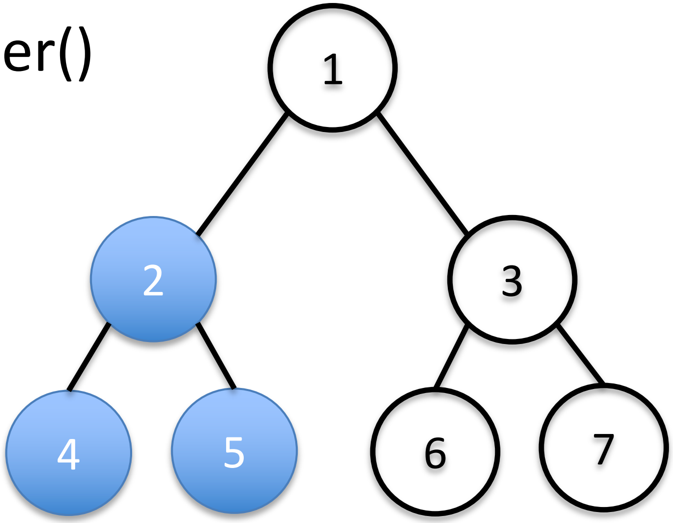
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

4, 2, 5

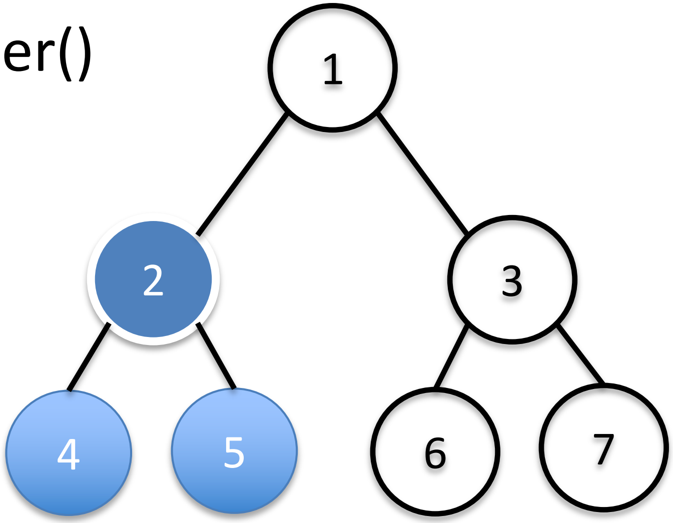
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

4, 2, 5

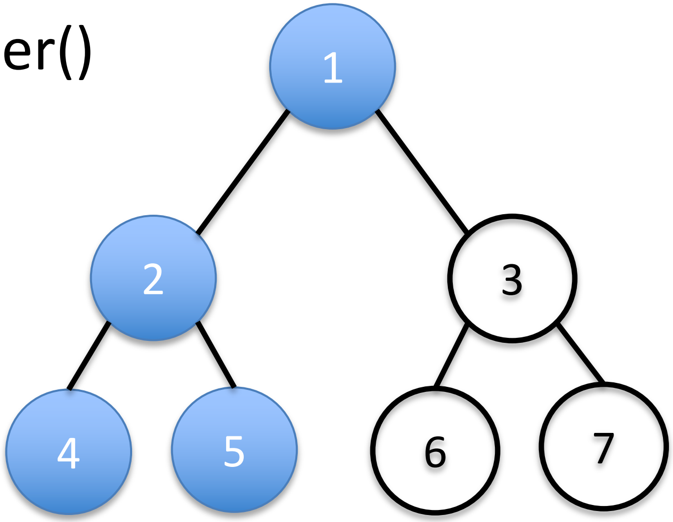
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

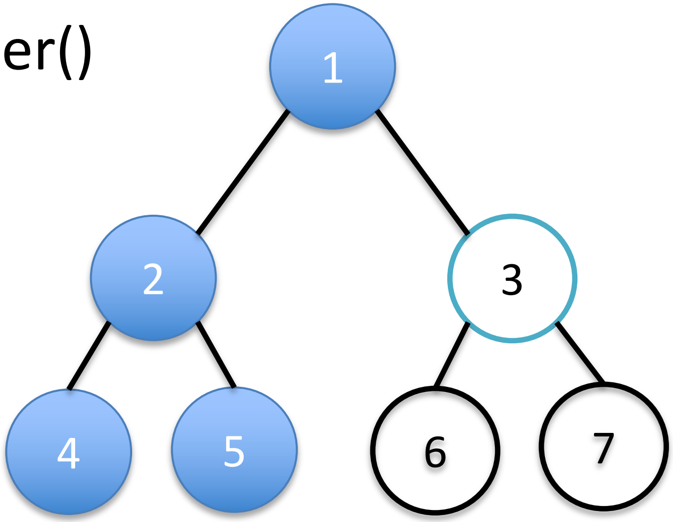
4, 2, 5, 1

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

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

4, 2, 5, 1

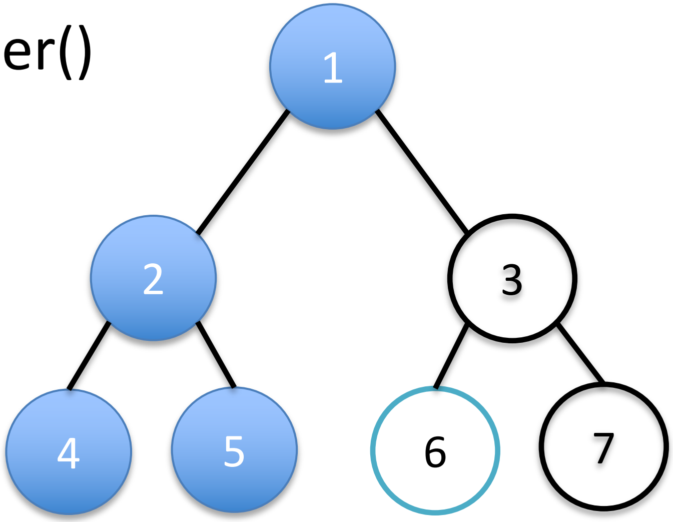
There are different ways to traverse a tree, depending on what needs to be done

inorder()

```
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

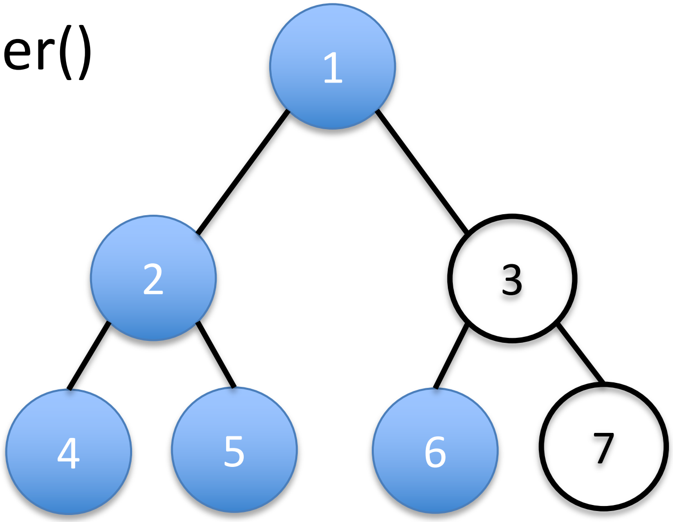
4, 2, 5, 1

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

```
inorder()
        left.inorder()
        visit
        right.inorder()
```

Example:

Drawing a tree



Visited

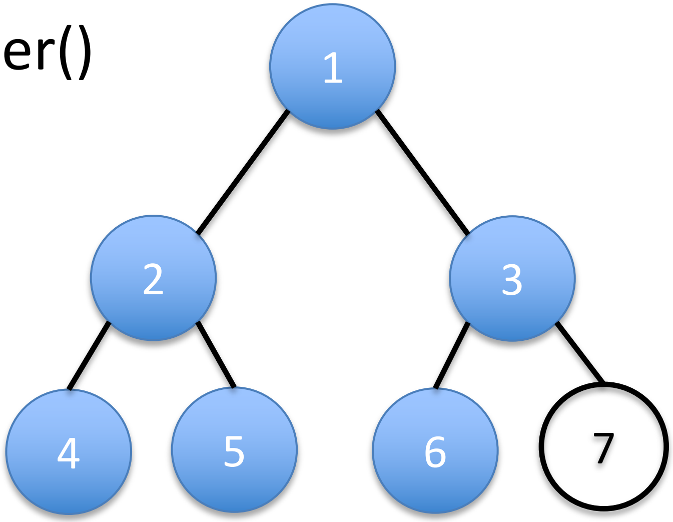
4, 2, 5, 1, 6

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

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

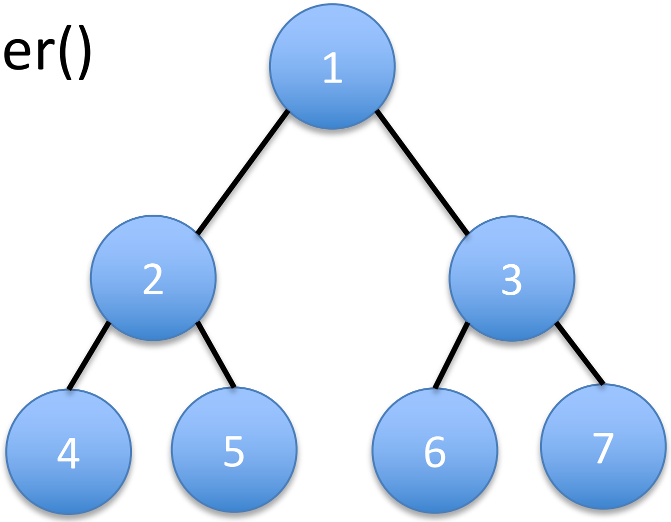
4, 2, 5, 1, 6, 3

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

```
inorder()
    left.inorder()
    visit
    right.inorder()
```

Example:

Drawing a tree



Visited

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