CS 10:
Problem solving via Object Oriented Programming

Hierarchies 1: Binary Trees

Agenda

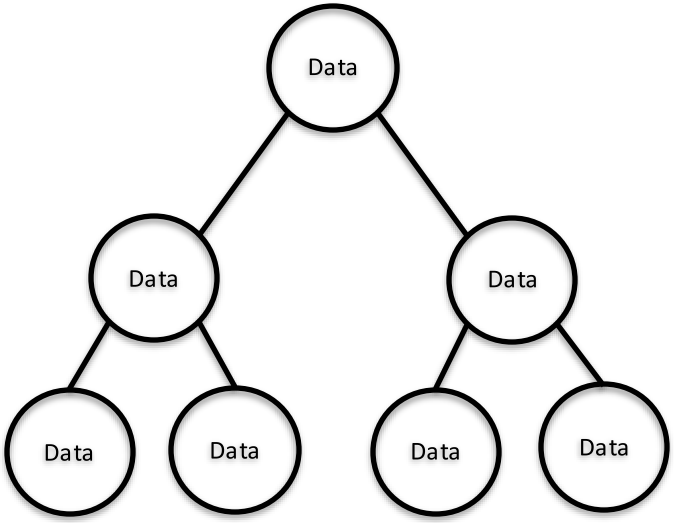
1. General-purpose binary trees
2. Accumulators
3. Tree traversal

Key points:

1. **Trees are useful for hierarchical data**
2. **Binary trees have 0, 1, or 2 children at each node**
3. **Not all trees are binary (PS-2 isn't)**
4. **Trees may not be balanced**
5. **Trees lead to beautiful recursive code (so beautiful it brings a tear to my eye!)**

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

Tree data structure



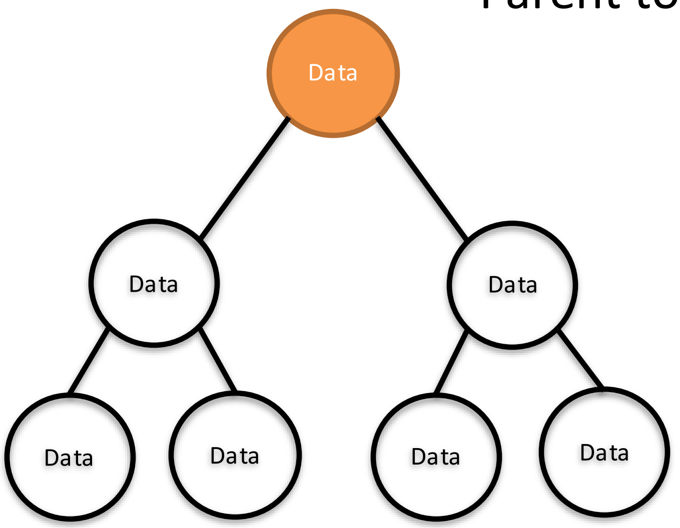
Meant for hierarchical data where there is a relationship between the data each node holds

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)



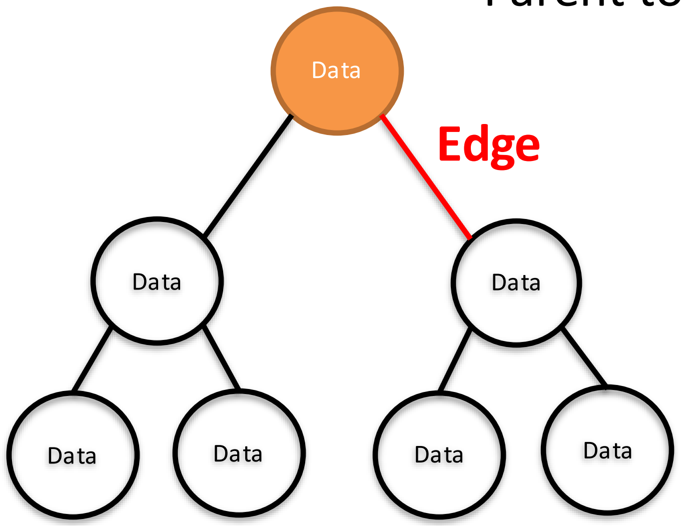
Meant for hierarchical data where there is a relationship between the data each node holds

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)



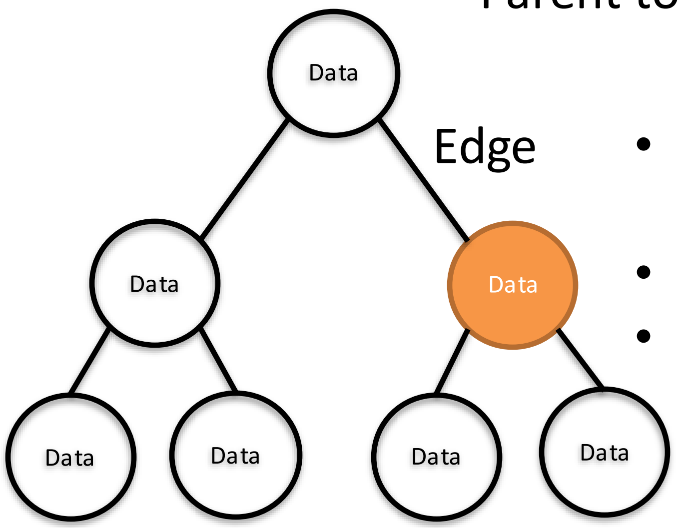
Meant for hierarchical data where there is a relationship between the data each node holds

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

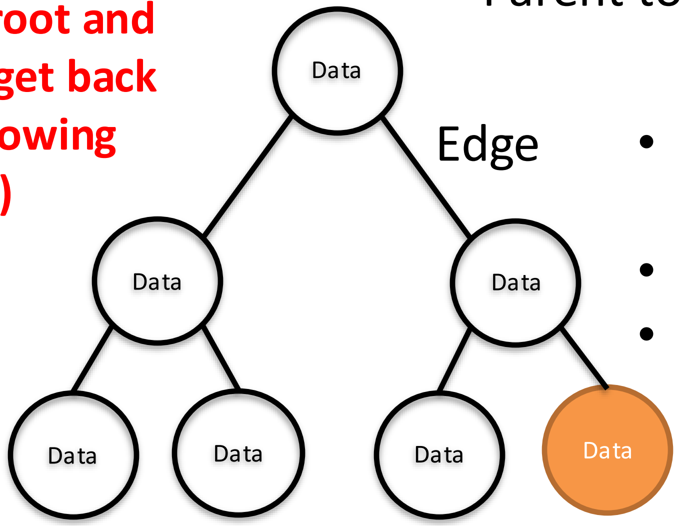
We can represent hierarchical data using a data structure called a tree

Tree data structure

Root node

- Parent to two children (called left and right)

Trees have one root and no cycles (can't get back to parent by following edge to children)



Edge

- 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

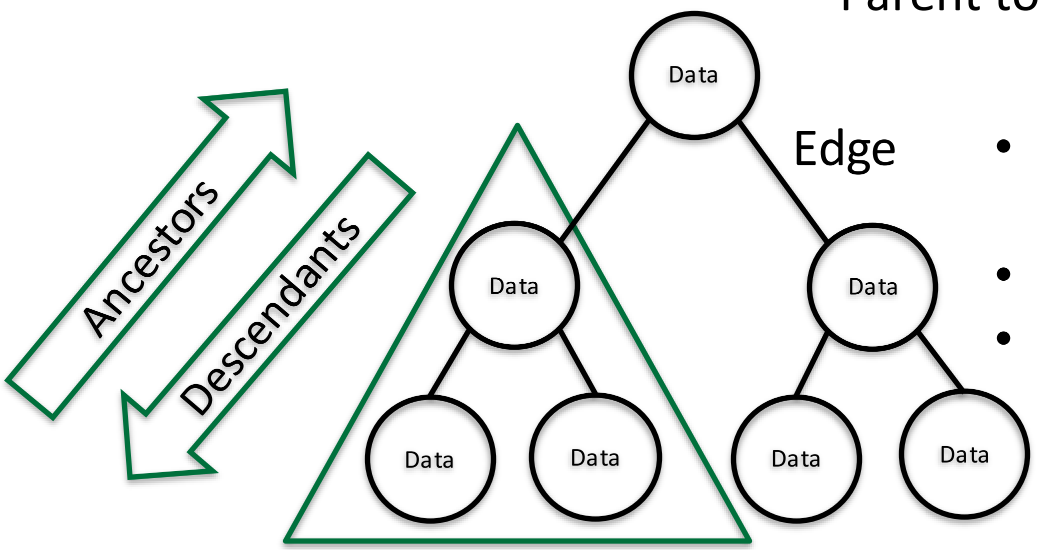
Meant for hierarchical data where there is a relationship between the data each node holds

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

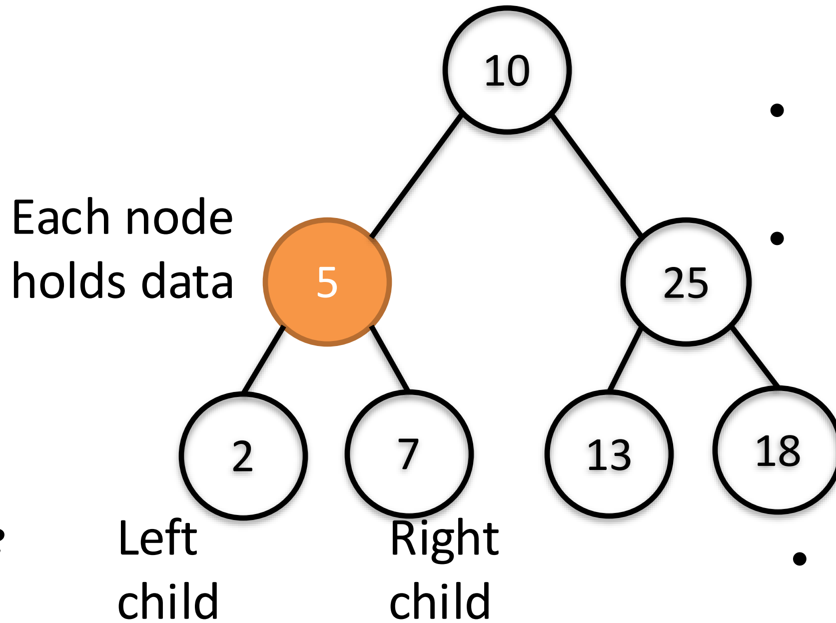
Each node can be thought of as the root of a subtree

Subtree

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

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

Binary Tree data structure



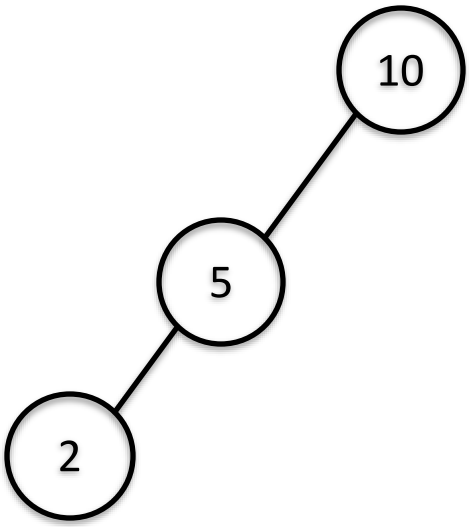
0, 1, or 2 children in *BinaryTree*

- 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

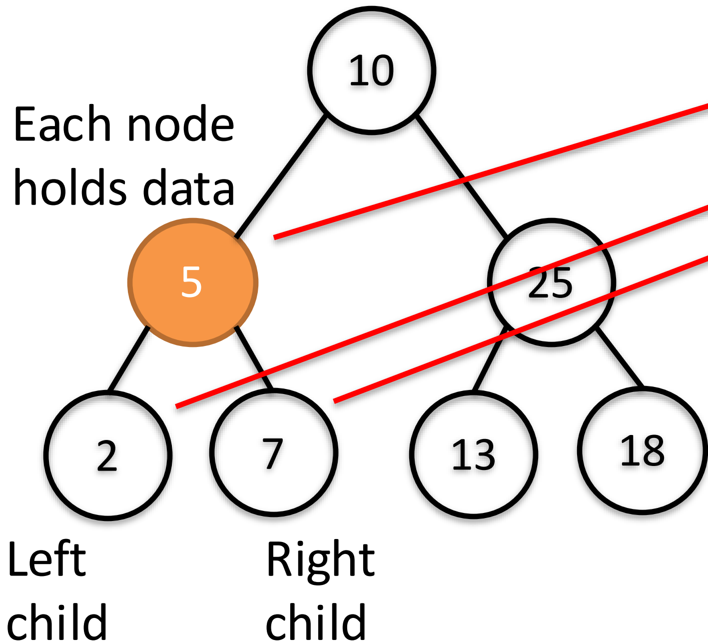
Binary Tree data structure



- 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 root of its own subtree

BinaryTree.java

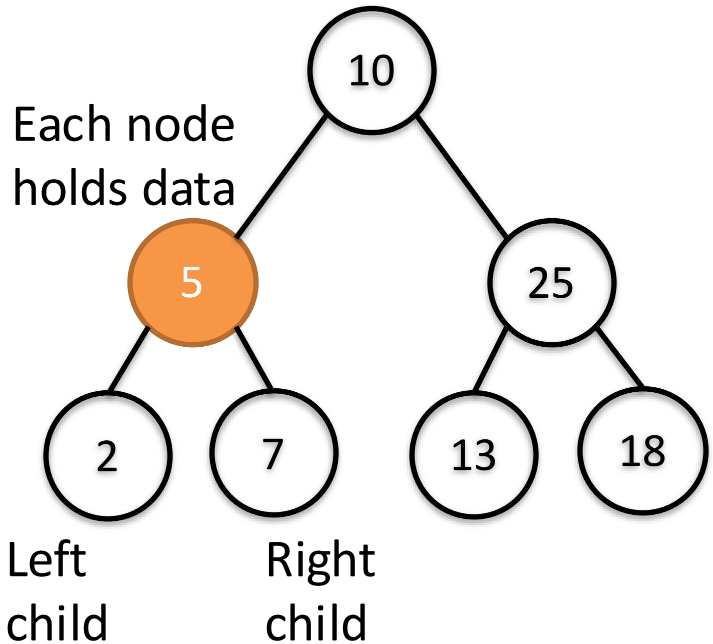



```java
public class BinaryTree<E> {
    private BinaryTree<E> left, right;   // children; can be null
    E data;

    /**
     * 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 root of its own subtree

BinaryTree.java




```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**

Building a BinaryTree

BinaryTree.java

Create root node

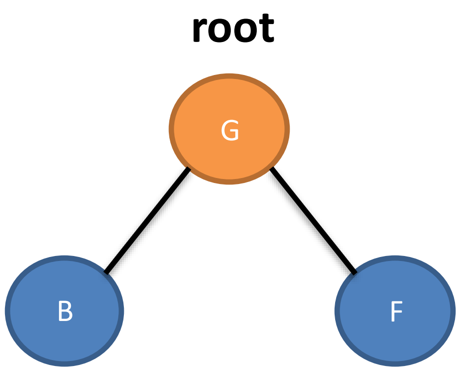
```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
 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

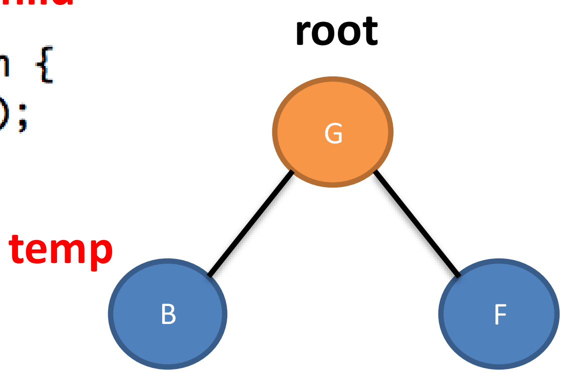


Building a BinaryTree

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);
```



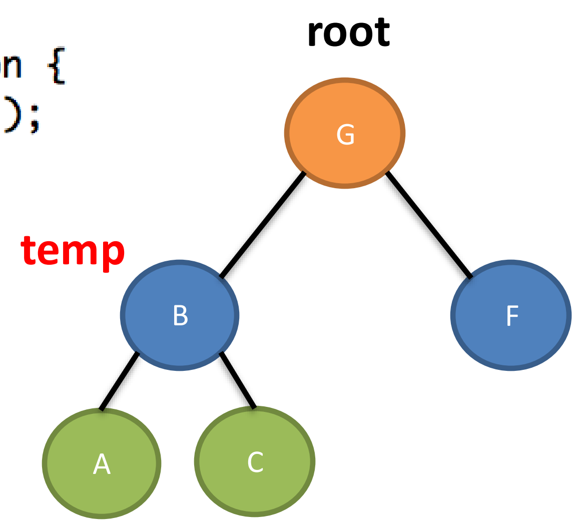
- **What would happen if didn't create *temp = root.left*, but instead set *root = root.left***
- **Would loose pointer to *root* node (*root* would be garbage collected)**

Building a BinaryTree

BinaryTree.java

Set left and right children

```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);
```

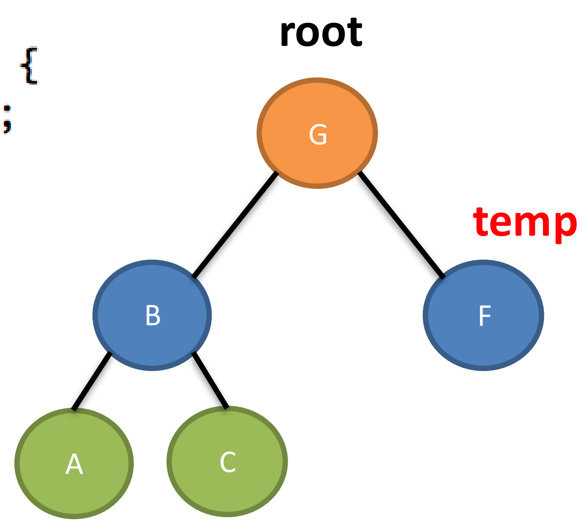


Building a BinaryTree

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);
```

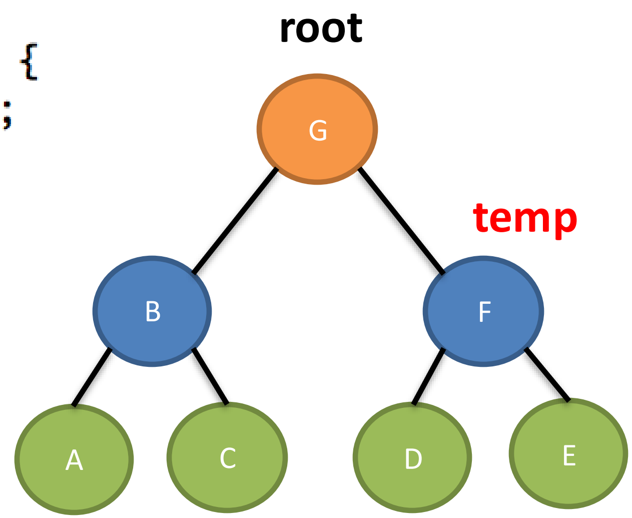


Building a BinaryTree

BinaryTree.java

Add children

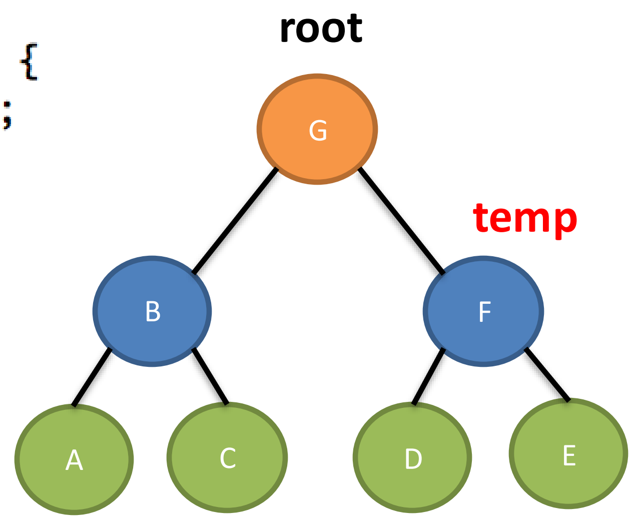
```
public static void main(String[] args) throws IOException {
    BinaryTree<String> root = new BinaryTree<String>("G");
    root.left = new BinaryTree<String>("B");
    root.right = new BinaryTree<String>("F");
    BinaryTree<String>temp = root.left;
    temp.left = new BinaryTree<String>("A");
    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 has three useful helper methods: hasLeft, hasRight, isleaf

```
public boolean hasLeft() { return left != null; }

public boolean hasRight() { return right != null; }

public boolean isLeaf() { return left == null && right == null; }
```

True if left child not null, indicates has a left child

True if right child not null, indicates has a right child

True if left and right children are null, indicates no children (leaf)

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**

```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;
}
```

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***

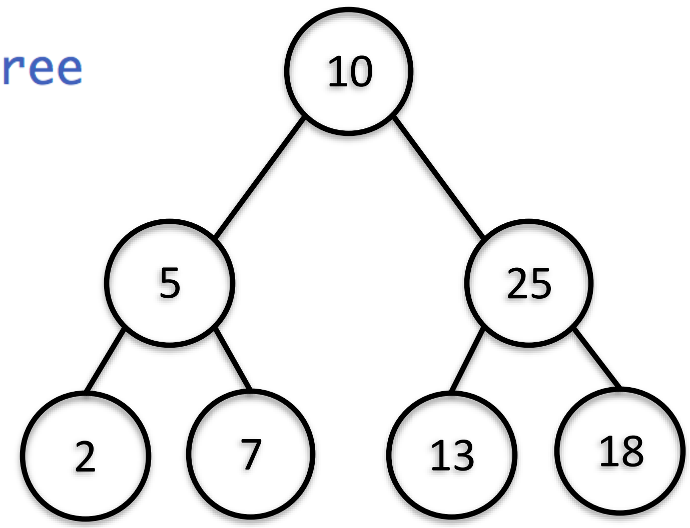
Return size of this subtree
If leaf node, will return 1
Recursion will then “bubble up” until it gets back to the original node on which ***size()*** **was called**
In that node ***num*** **will then have the size of the entire subtree**

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;
}
```

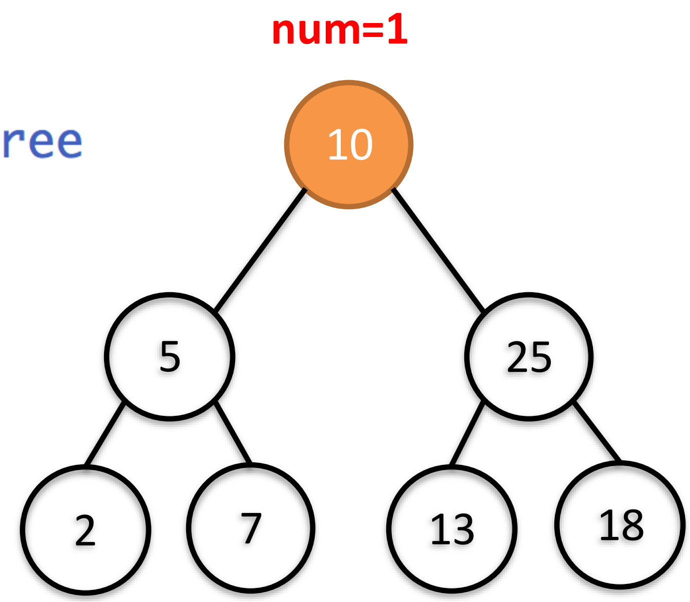
Call *size()* on root node



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

BinaryTree.java

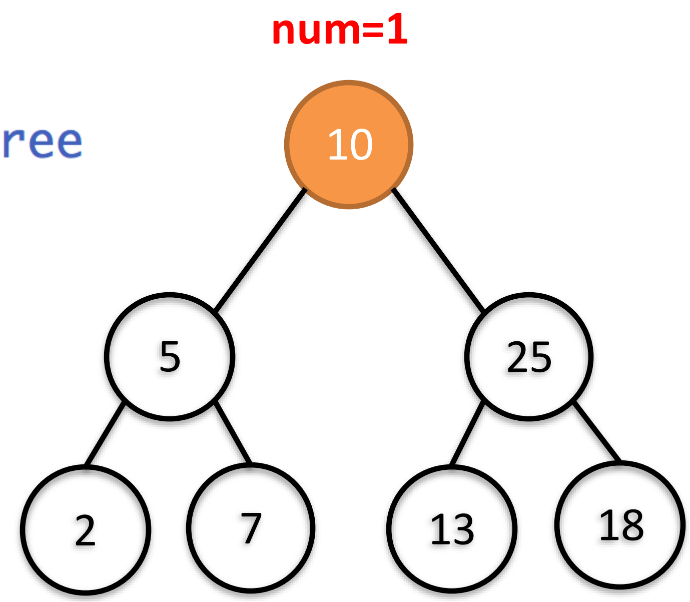
```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



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;
}
```

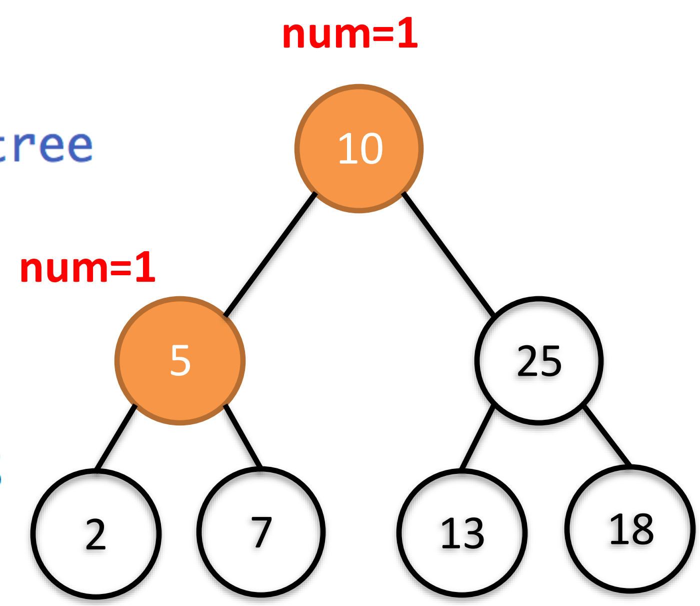


- **Has left child**
- **Make recursive call on left child**

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

BinaryTree.java

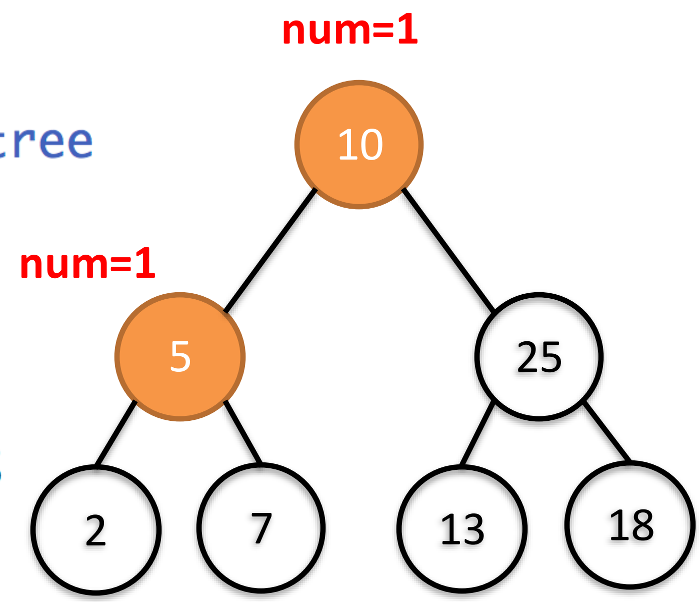
```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



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;
}
```

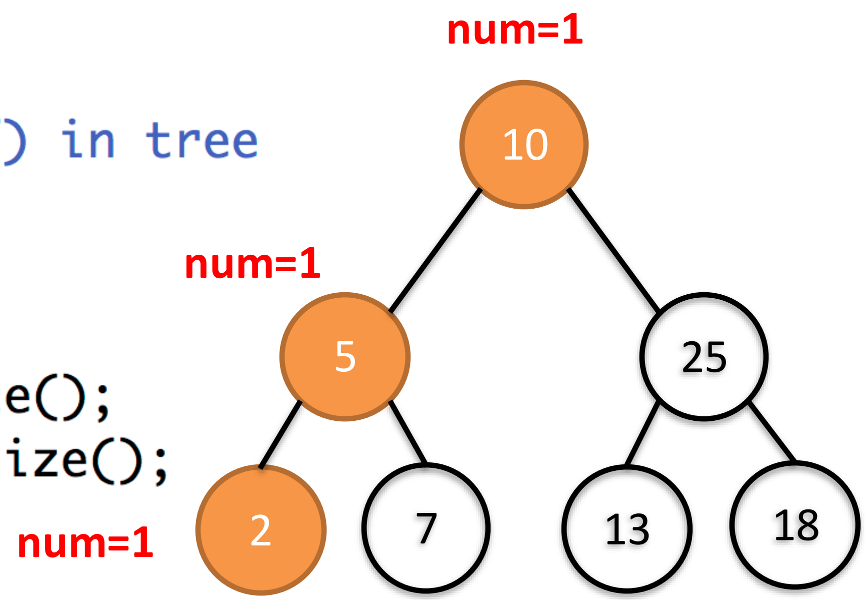


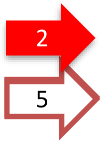
- **Has left child**
- **Make recursive call on left child**

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;
}
```

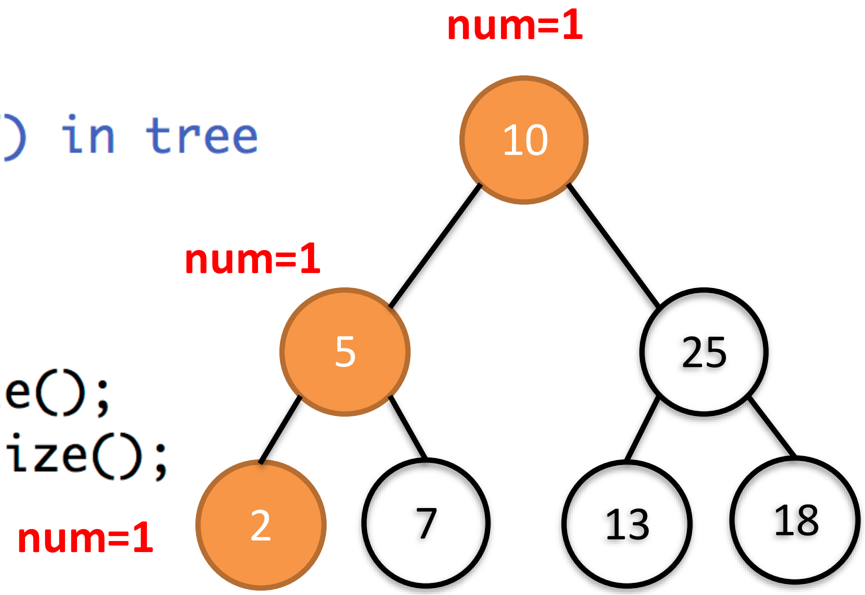




Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

```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;
}
```

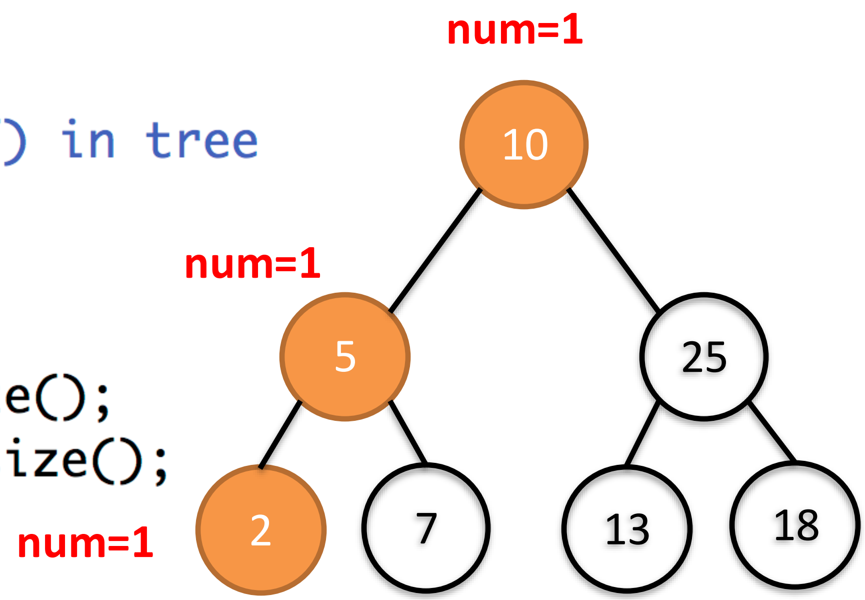


- **No children**

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

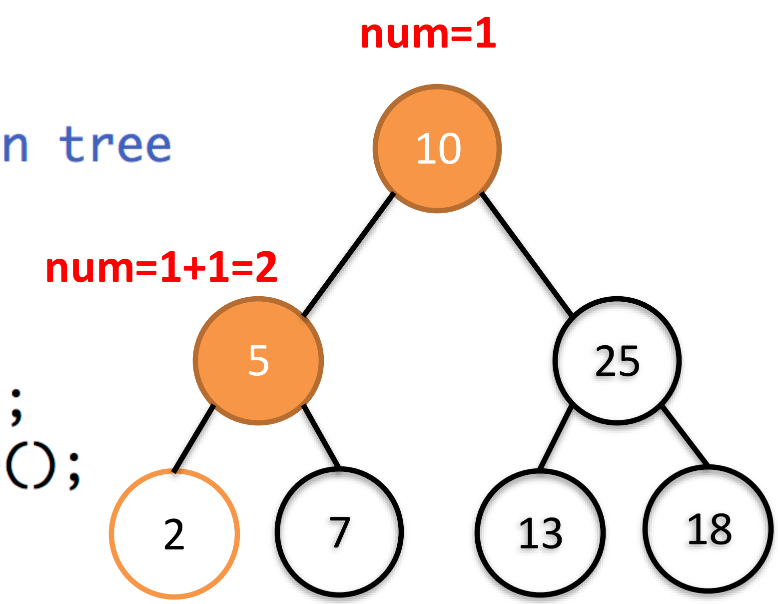


- No children
- Return 1 back to node 5

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

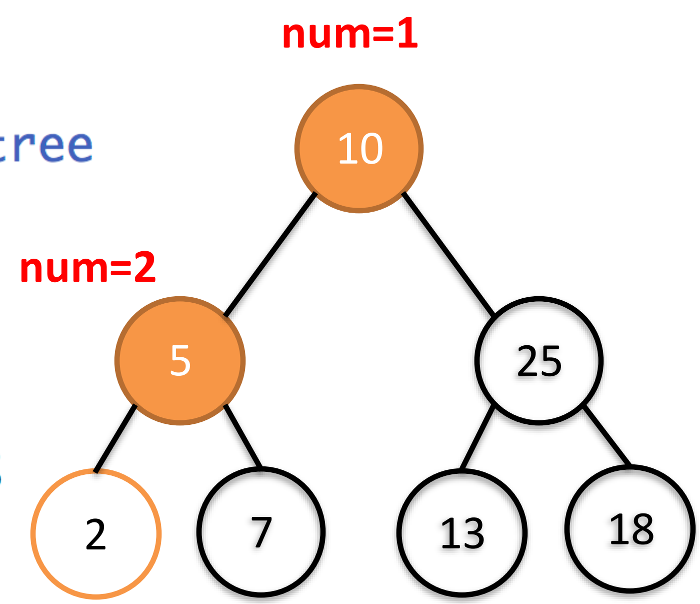


- **Increment num on Node 5**

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

BinaryTree.java

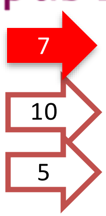
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



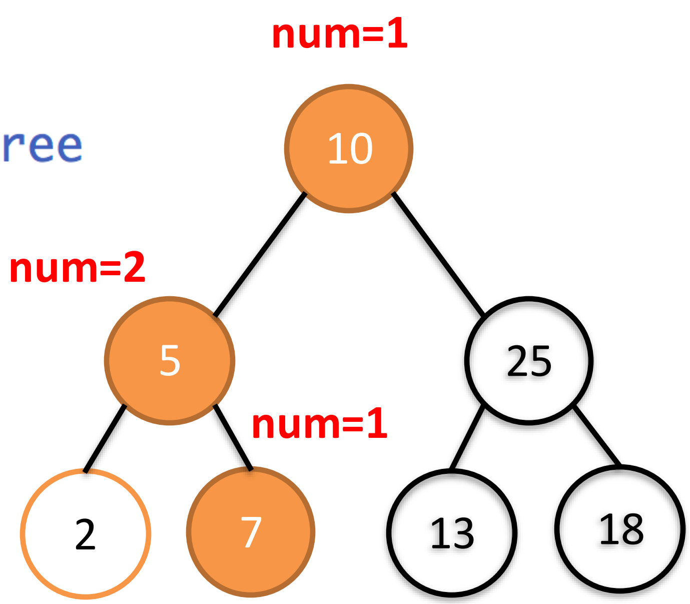
- Has right child
- Make recursive call on right child

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

BinaryTree.java



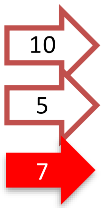

```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```

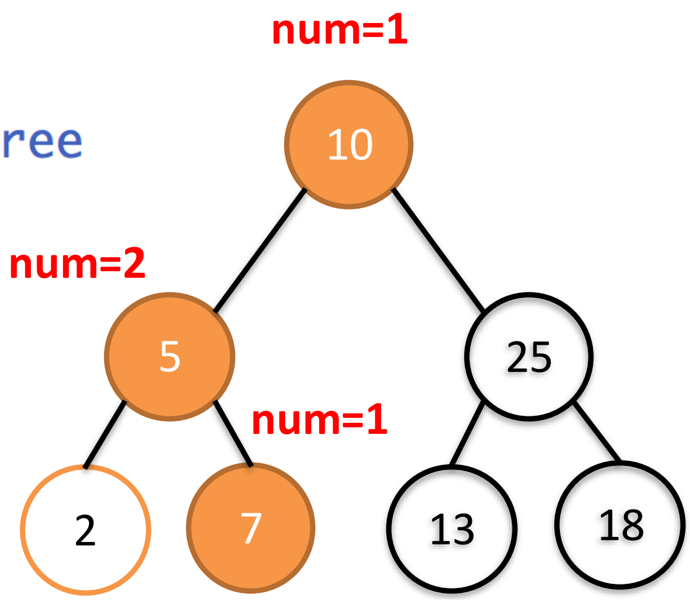


Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

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



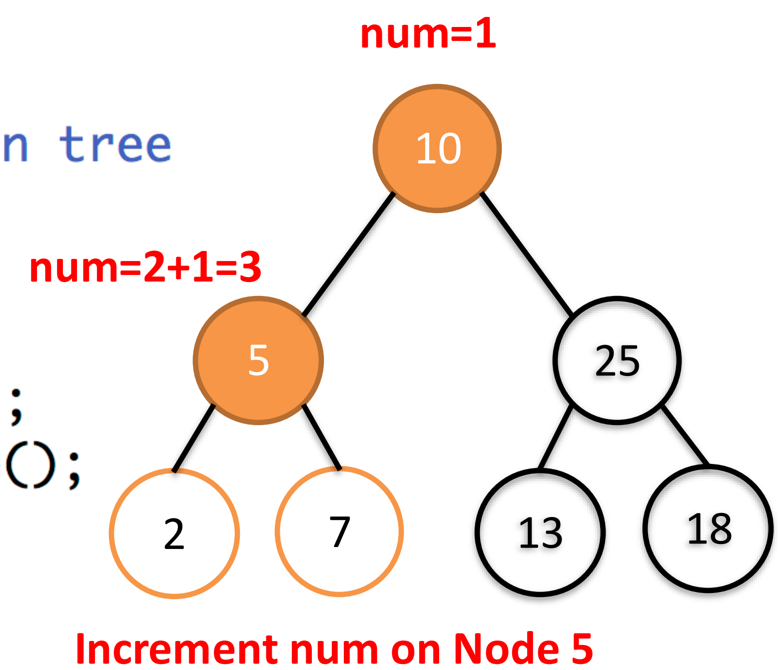


- **No children**
- **Return 1 back to node 5**

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

BinaryTree.java

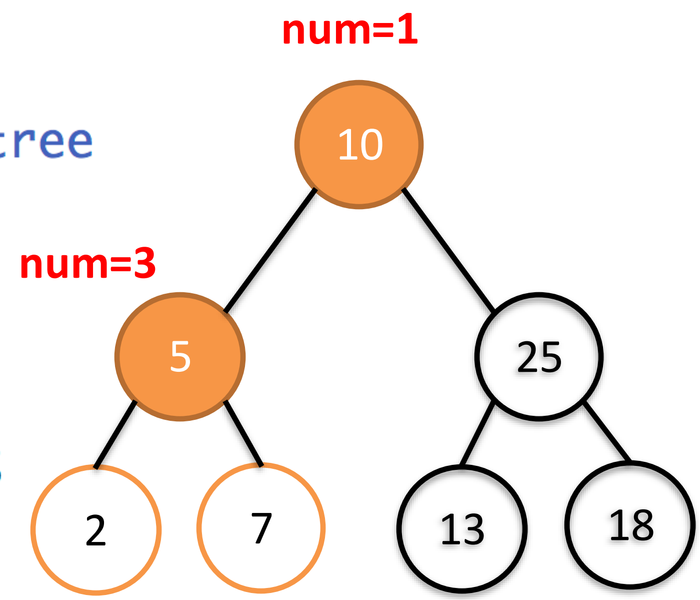
```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



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;
}
```



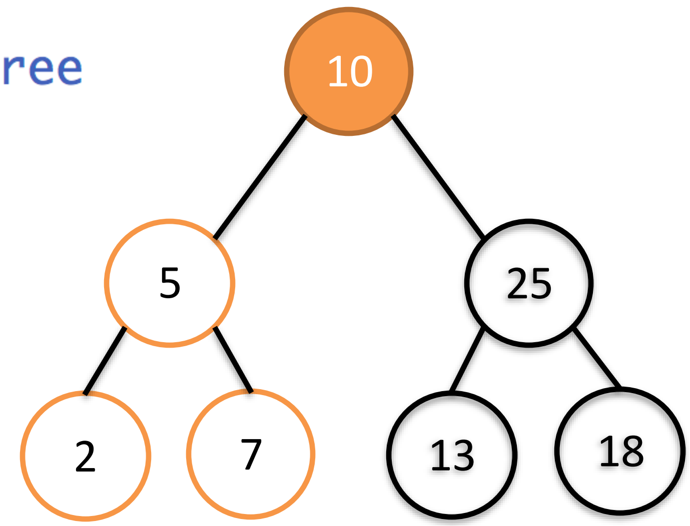
- **Node 5 is done**
- **Return 3 to root**

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

BinaryTree.java

```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

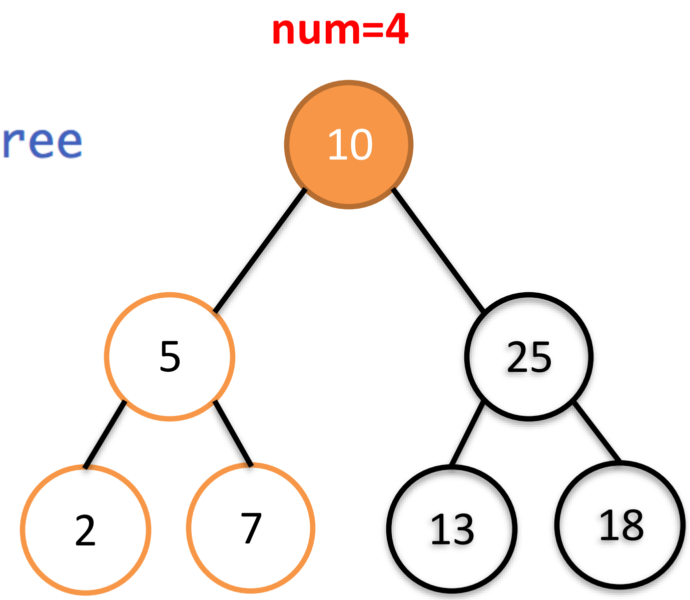


- **Increment num on root**

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;
}
```

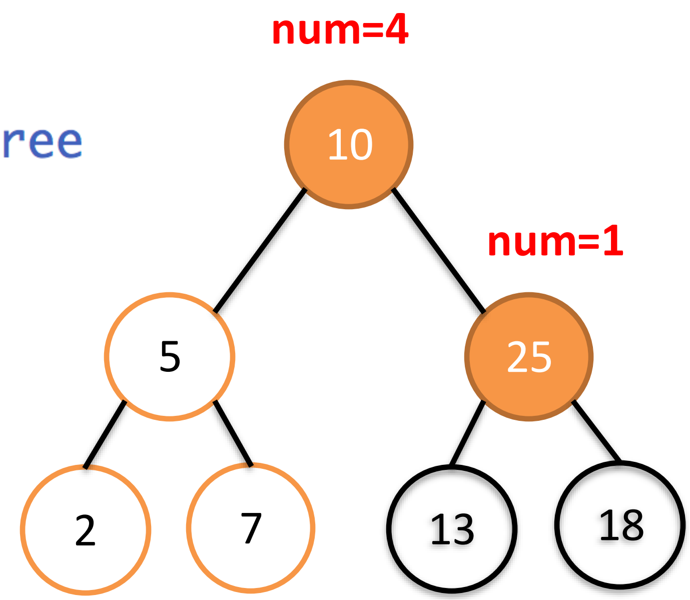


- **Has right child**
- **Make recursive call on right child**

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

BinaryTree.java

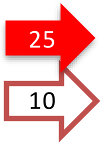
```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```

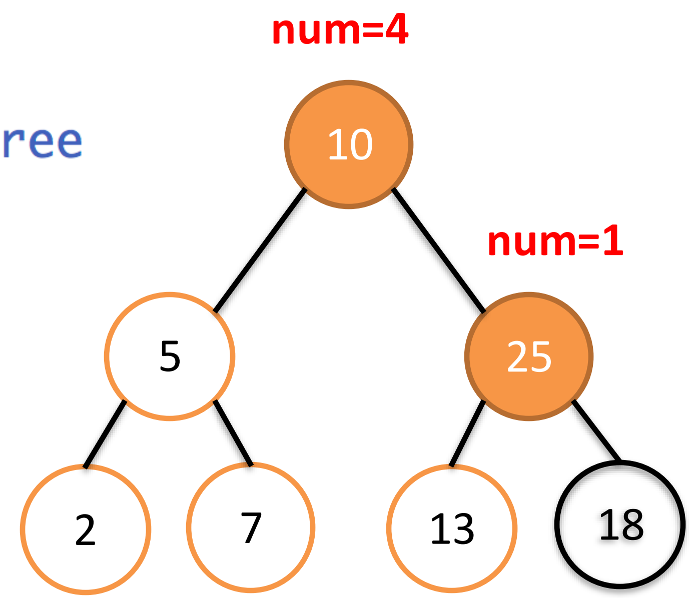


Use recursion to calculate tree size from any given node = size of both children +1

BinaryTree.java

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



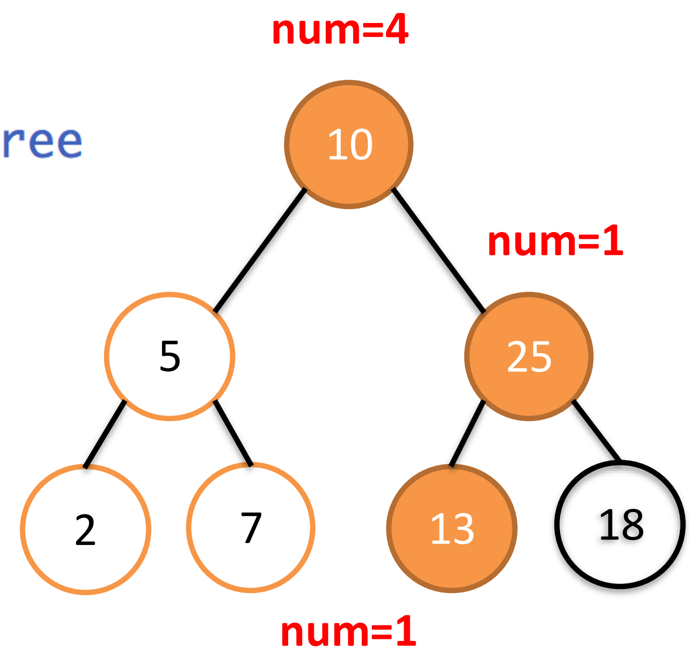


- **Has left child**
- **Make recursive call on left child**

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

BinaryTree.java

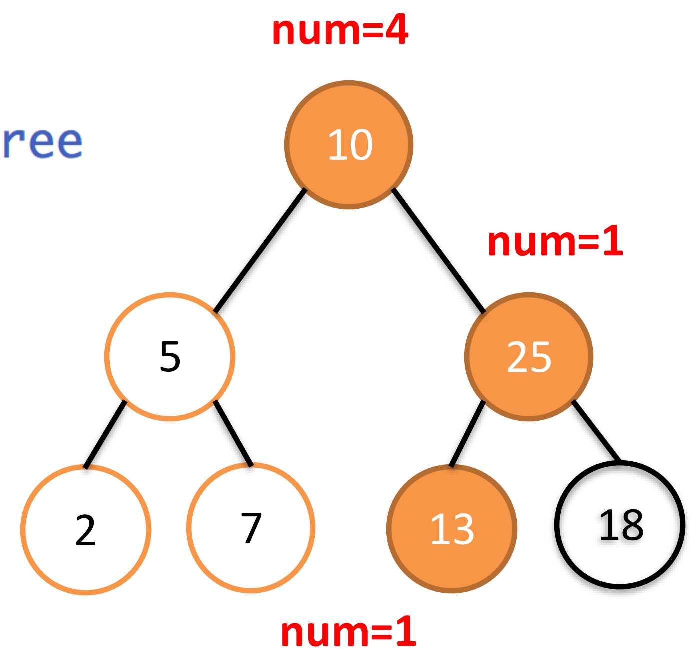
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
13  int num = 1;
25  if (hasLeft()) num += left.size();
10  if (hasRight()) num += right.size();
    return num;
}
```



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

BinaryTree.java

```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;
}
```

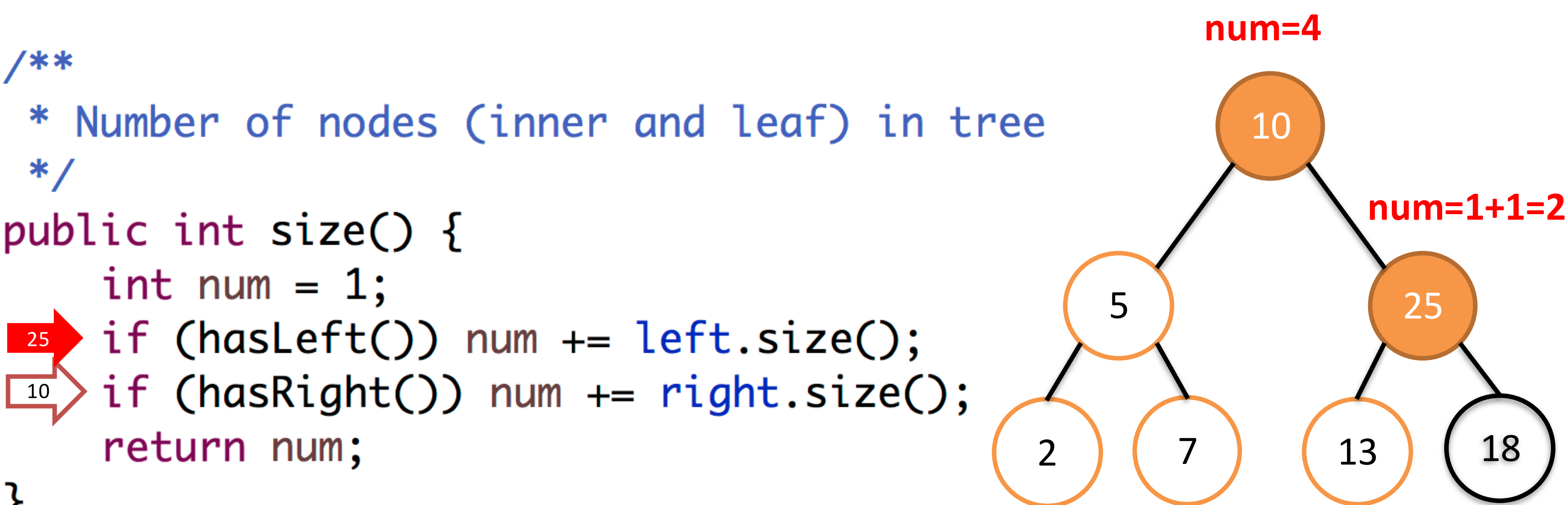


- **No children**
- **Return 1 back to Node 25**

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

BinaryTree.java

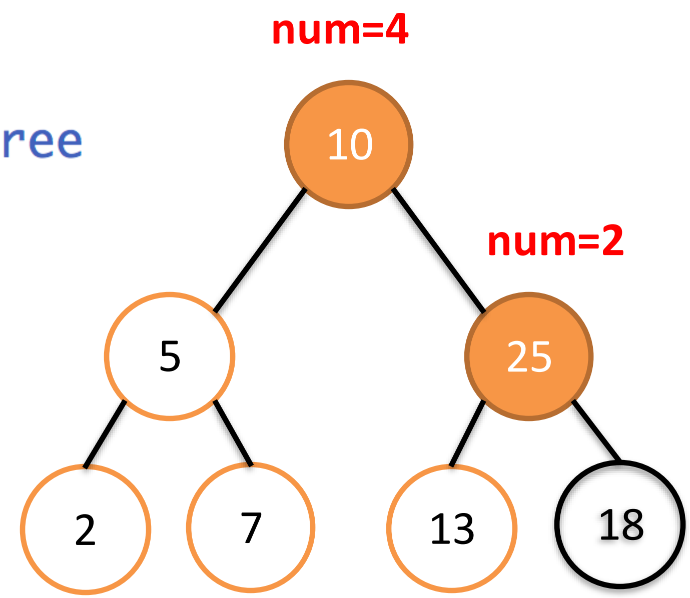
```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```



- **Increment num on Node 25**

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

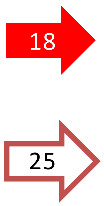
```
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
25 ➔ if (hasRight()) num += right.size();
    return num;
}
```



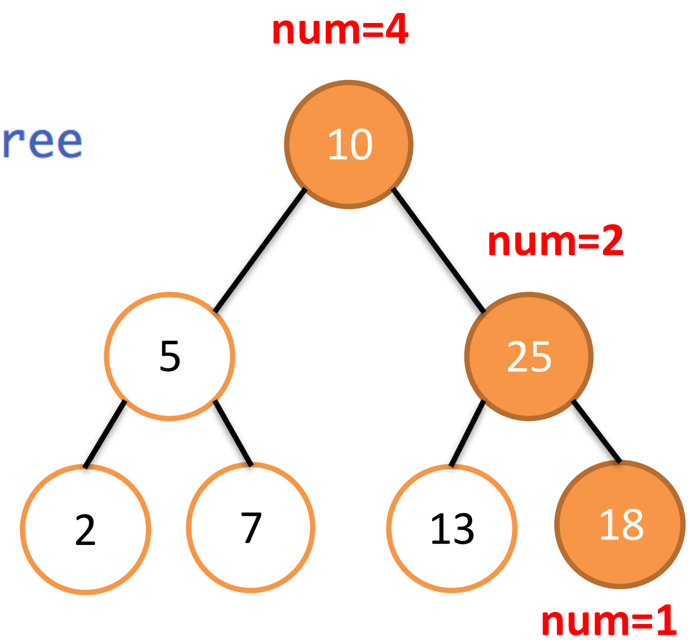
- **Has right child**
- **Make recursive call on right child**

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

BinaryTree.java



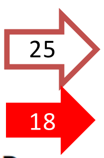

```java
/**
 * Number of nodes (inner and leaf) in tree
 */
public int size() {
    int num = 1;
    if (hasLeft()) num += left.size();
    if (hasRight()) num += right.size();
    return num;
}
```

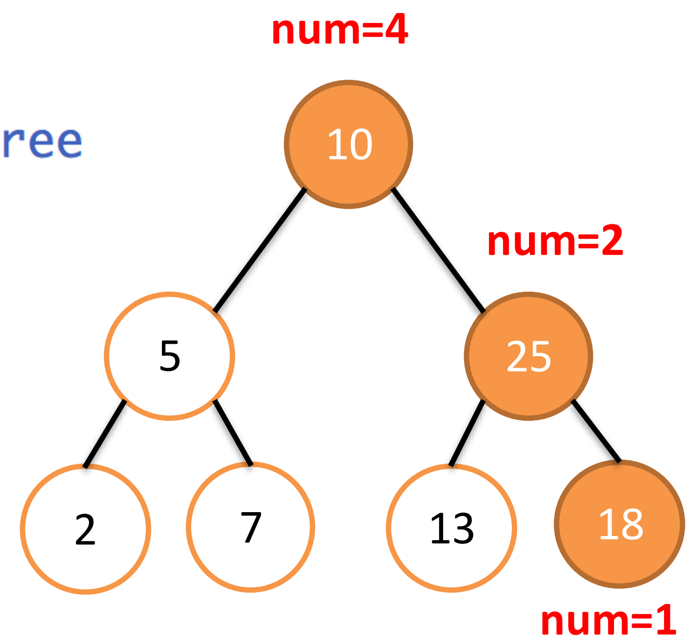


Use recursion to calculate tree size from any given node = size of both children +1

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;
}
```



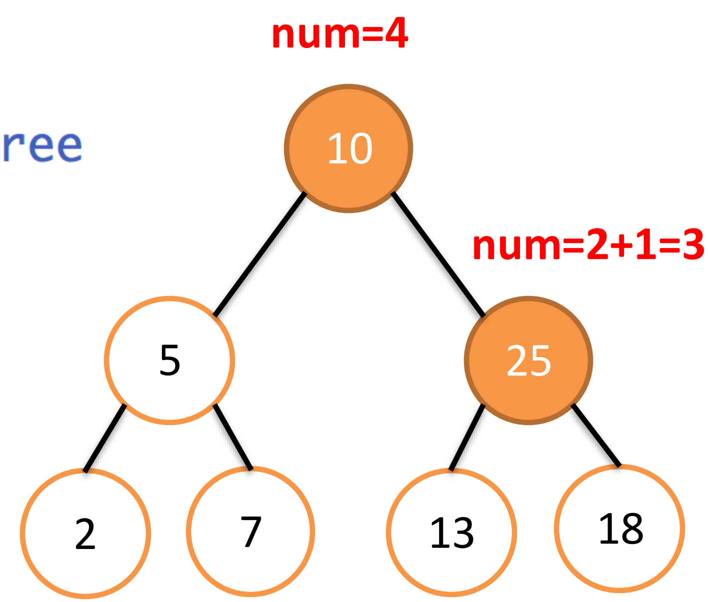


- **No children**
- **Return 1 to Node 25**

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;
}
```

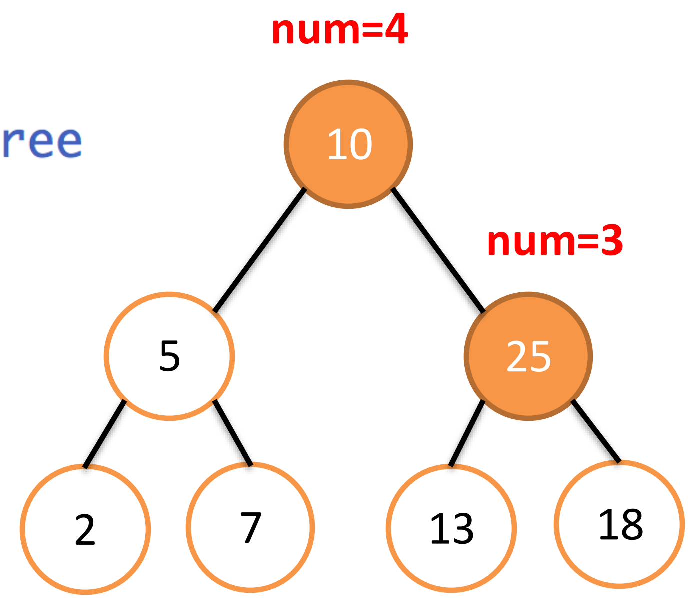


- **Increment num on Node 25**

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



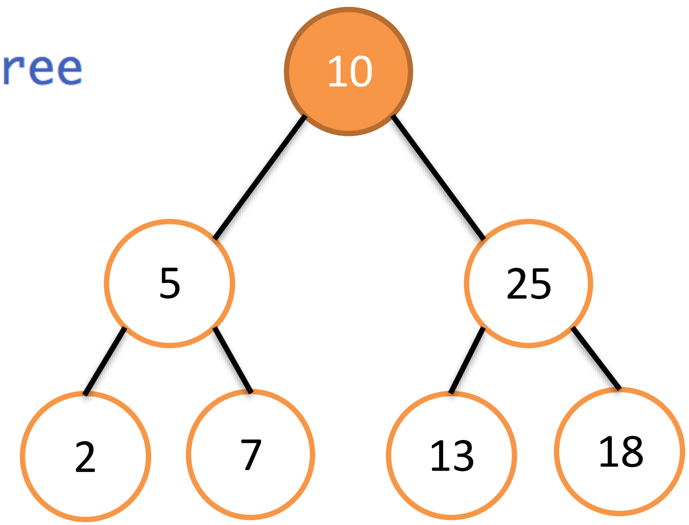
- **Node 25 is done**
- **Return 3 back to root**

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

BinaryTree.java

```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

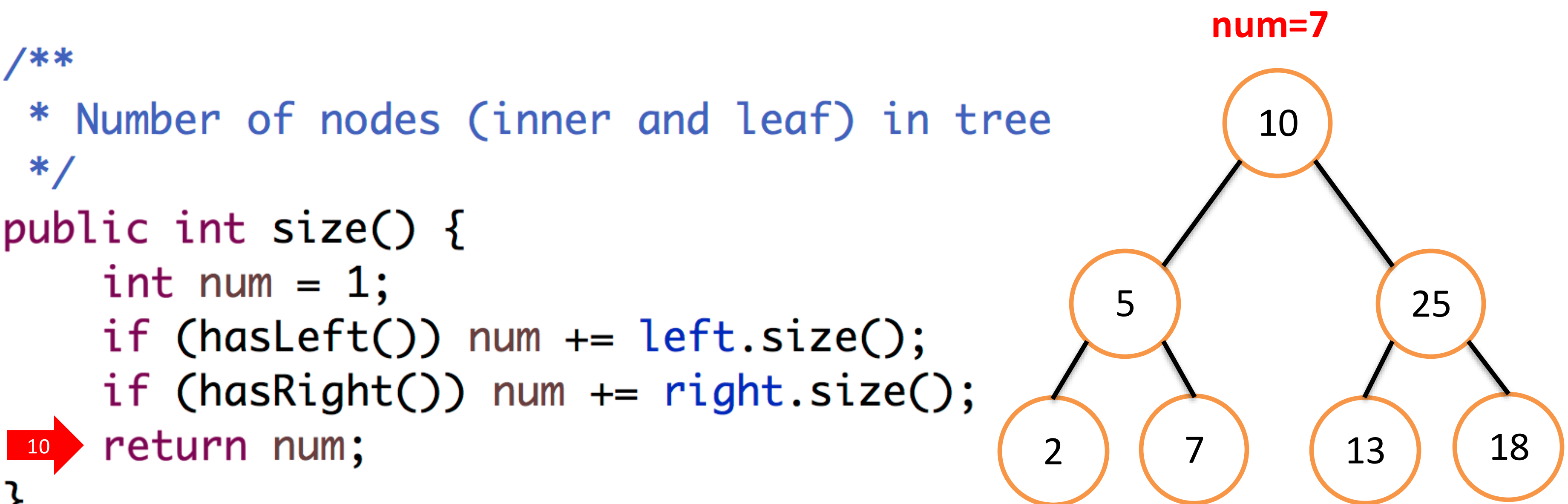


Increment num on root

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();
    return num;
}
```



Done!
Return 7

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

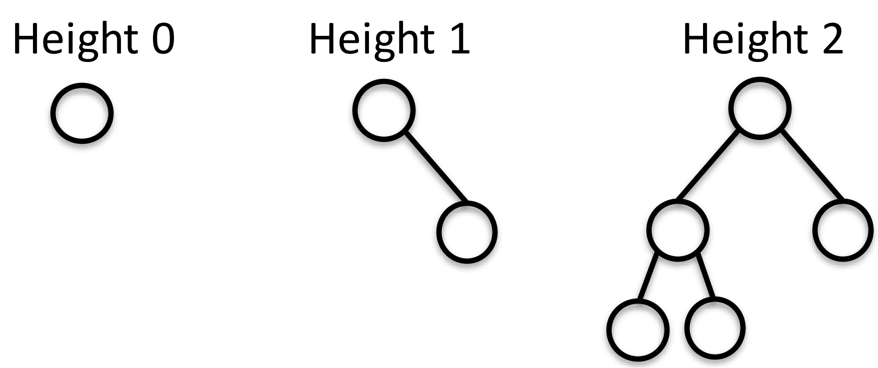
BinaryTree.java

- **Height is the number of <u>edges</u> on the *<u>longest</u>* 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**



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

```
/**
 * Same structure and data
 * @param t2 compare with this tree
 * @return true if this tree and t2 have the have structure and data in each node, else false
*/
public boolean equals(BinaryTree<E> t2) {
   if (hasLeft() != t2.hasLeft() || hasRight() != t2.hasRight()) return false;
   if (!data.equals(t2.data)) return false;
   if (hasLeft() && !left.equals(t2.left)) return false;
   if (hasRight() && !right.equals(t2.right)) return false;
   return true;
}
```

First check if same number number of children

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

```
/**
 * Same structure and data
 * @param t2 compare with this tree
 * @return true if this tree and t2 have the have structure and data in each node, else false
*/
public boolean equals(BinaryTree<E> t2) {
   if (hasLeft() != t2.hasLeft() || hasRight() != t2.hasRight()) return false;
   if (!data.equals(t2.data)) return false;
   if (hasLeft() && !left.equals(t2.left)) return false;
   if (hasRight() && !right.equals(t2.right)) return false;
   return true;
}
```

Next compare data is the same in each node

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

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

```
/**
 * Same structure and data
 * @param t2 compare with this tree
 * @return true if this tree and t2 have the have structure and data in each node, else false
*/
public boolean equals(BinaryTree<E> t2) {
   if (hasLeft() != t2.hasLeft() || hasRight() != t2.hasRight()) return false;
   if (!data.equals(t2.data)) return false;
   if (hasLeft() && !left.equals(t2.left)) return false;
   if (hasRight() && !right.equals(t2.right)) return false;
   return true;
}
```

Finally, ask each child to compare itself

Trees are equal if same shape and same data at all nodes

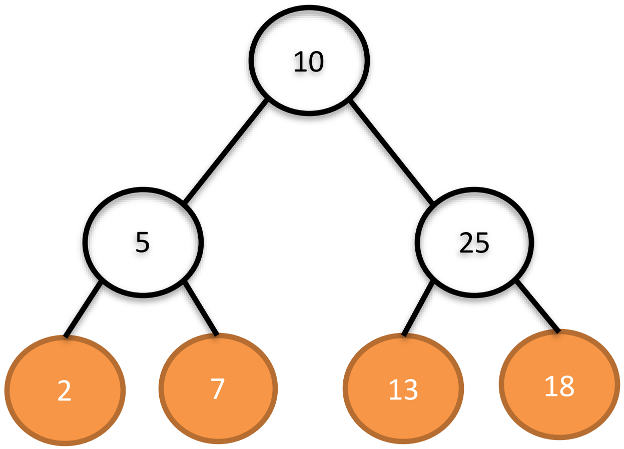
Agenda

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

Key points:

1. **Accumulators are a way to "build up" a value as a tree is traversed**
2. **Accumulators allow efficient code**

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

```
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**

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

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

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 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;
}
```

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

Note: the *BinaryTree.java* linked from the course web page prints in a slightly more sophisticated way

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 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;
}
```

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

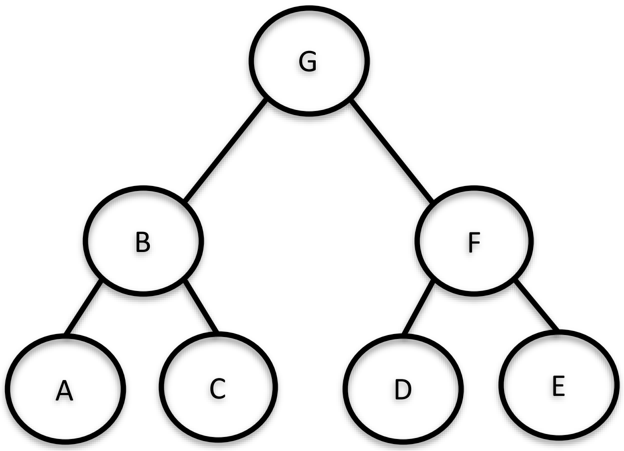
NOTE: "\n" means new line

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

Remember, toString returns a String, it doesn't print!

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*

Agenda

1. General-purpose binary trees

2. Accumulators

3. Tree traversal

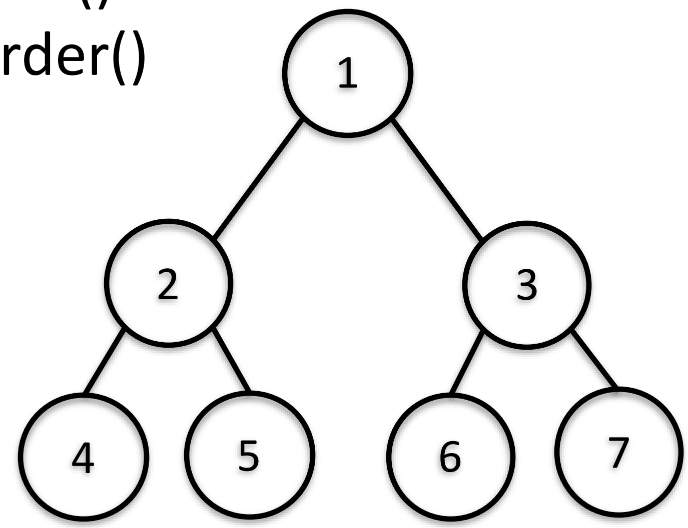
Key points:
Trees are commonly traversed in three ways
1. **Pre-order**
2. **Post-order**
3. **In-order**

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

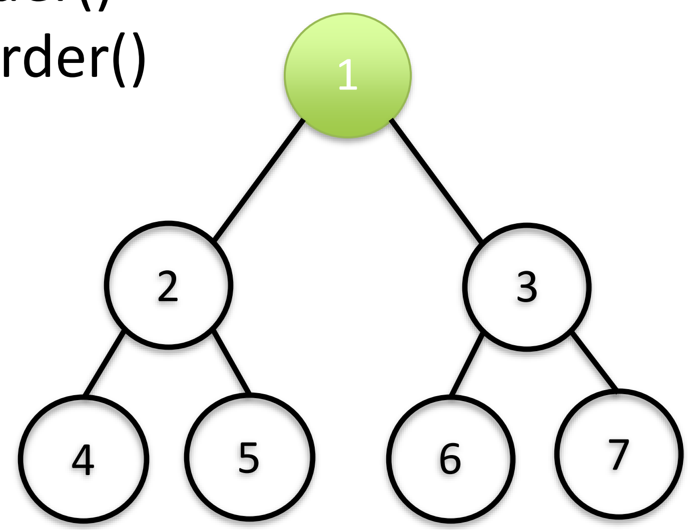
Table of contents in book

toString()

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

preorder()

visit
left.preorder()
right.preorder()



Visited
1

Examples:

File directory structure
Table of contents in book
toString()

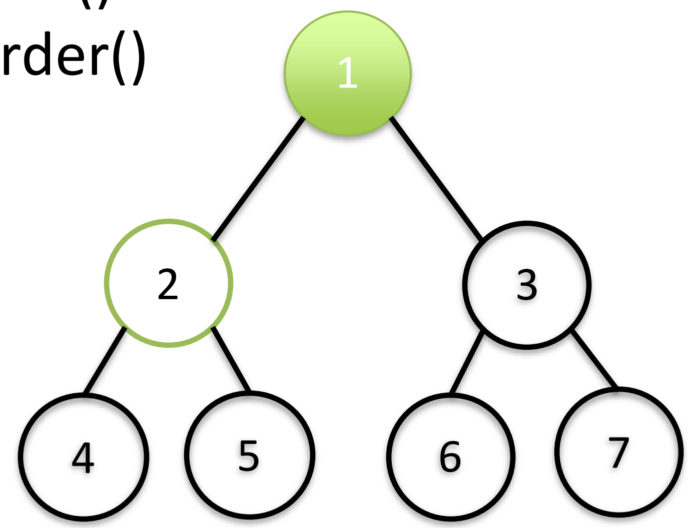
There are different ways to traverse a tree, depending on what needs to be done

preorder()

visit
left.preorder()
right.preorder()

Examples:

File directory structure
Table of contents in book
toString()

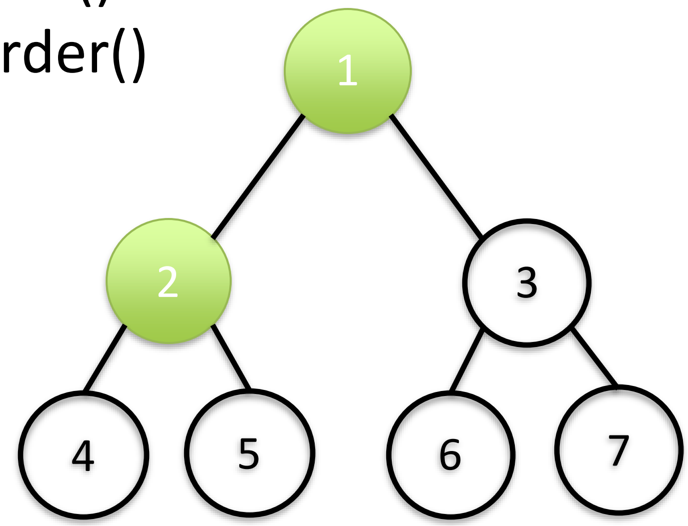


Visited
1

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

preorder()

visit
left.preorder()
right.preorder()



Visited
1, 2

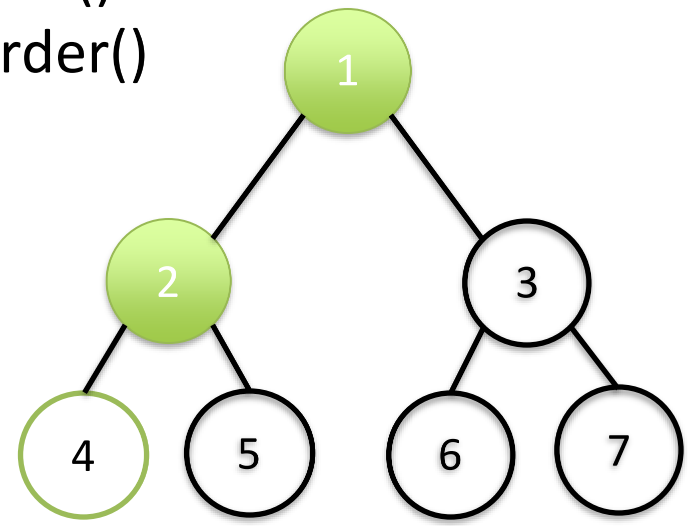
Examples:

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()



Visited

1, 2

Examples:

File directory structure

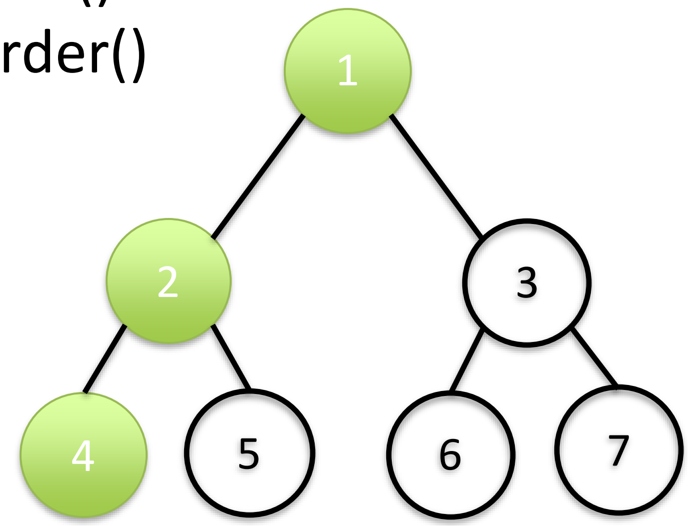
Table of contents in book

toString()

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

preorder()

visit
left.preorder()
right.preorder()



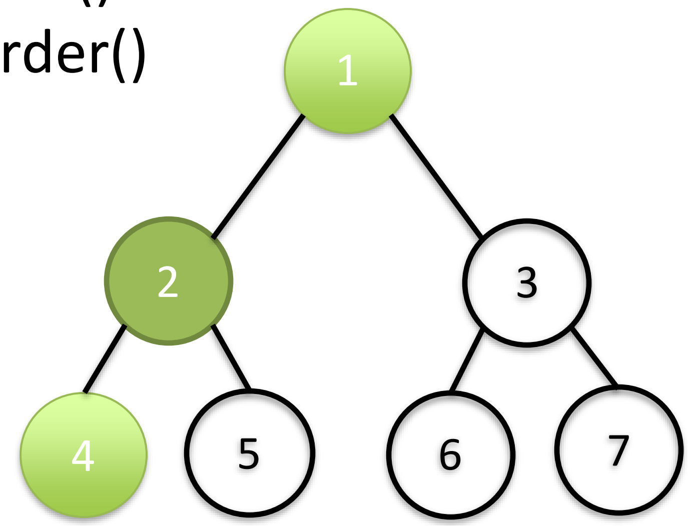
Visited
1, 2, 4

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()



Visited
1, 2, 4

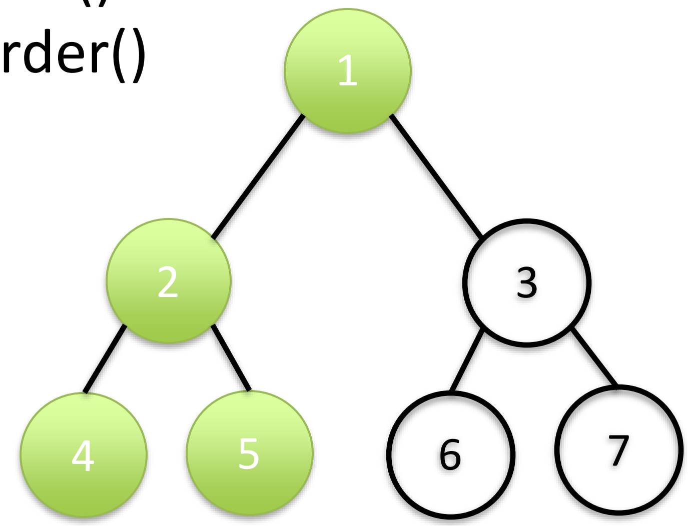
Examples:

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()



Visited

1, 2, 4, 5

Examples:

File directory structure

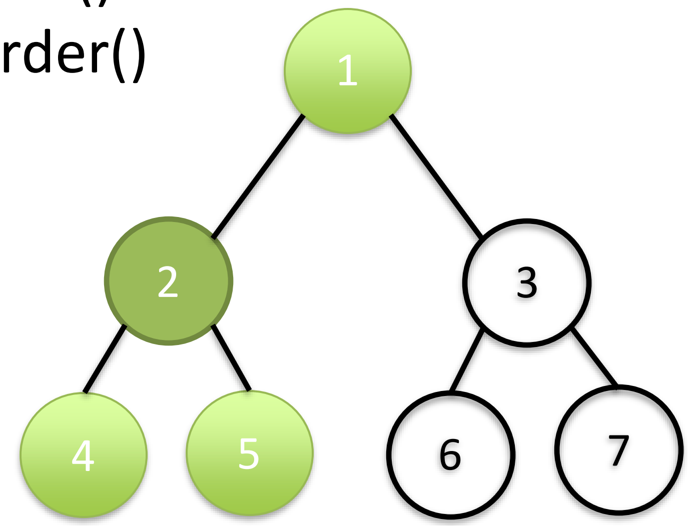
Table of contents in book

toString()

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

preorder()

visit
left.preorder()
right.preorder()



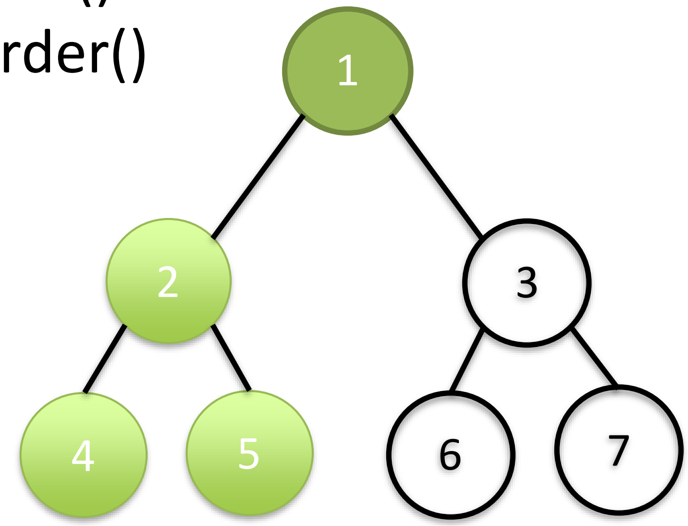
Visited
1, 2, 4, 5

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()



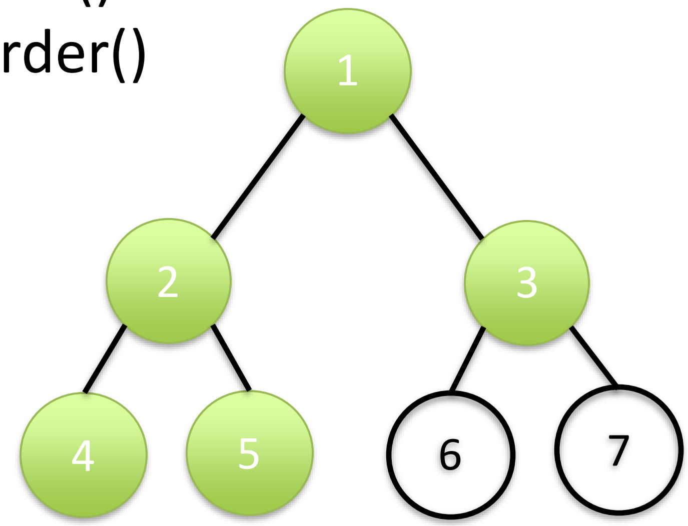
Visited
1, 2, 4, 5

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()



Visited

1, 2, 4, 5, 3

Examples:

File directory structure

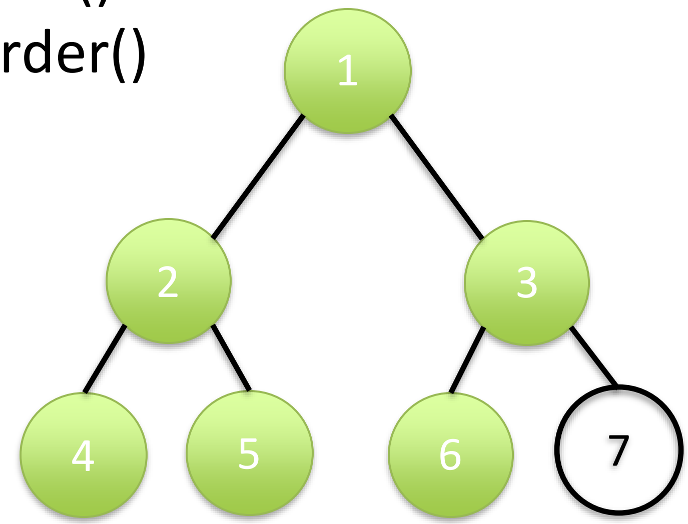
Table of contents in book

toString()

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

preorder()

visit
left.preorder()
right.preorder()



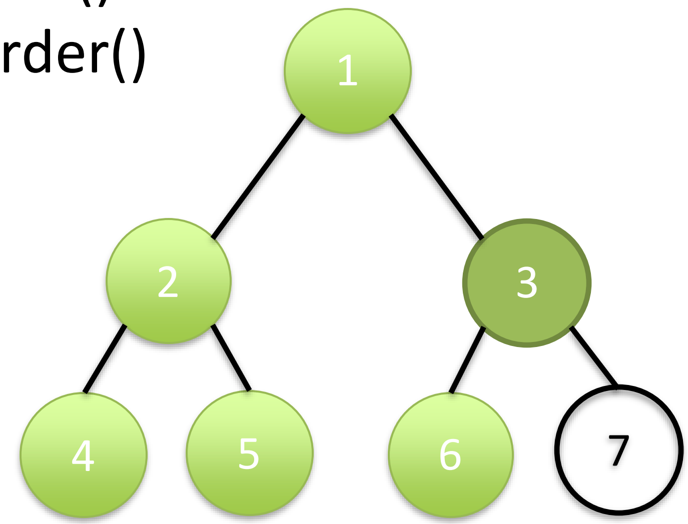
Visited
1, 2, 4, 5, 3, 6

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()



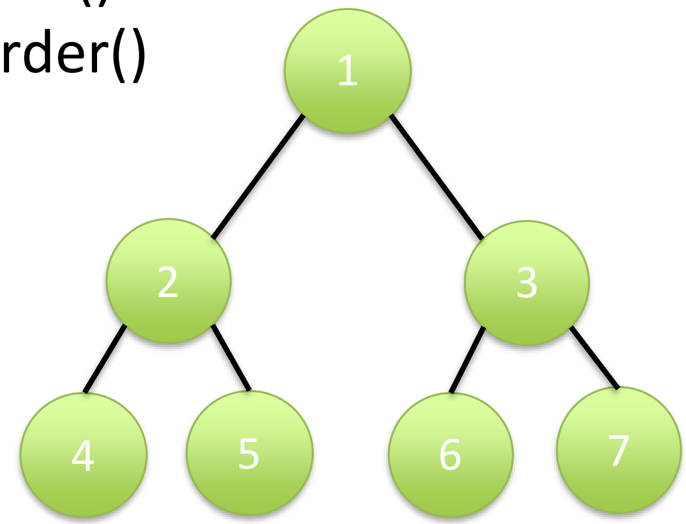
Visited
1, 2, 4, 5, 3, 6

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()



Visited

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

Examples:

File directory structure

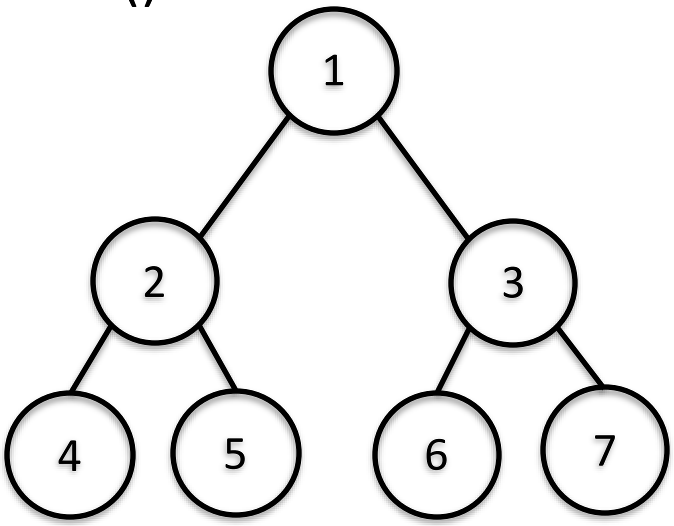
Table of contents in book

toString()

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

postorder()

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



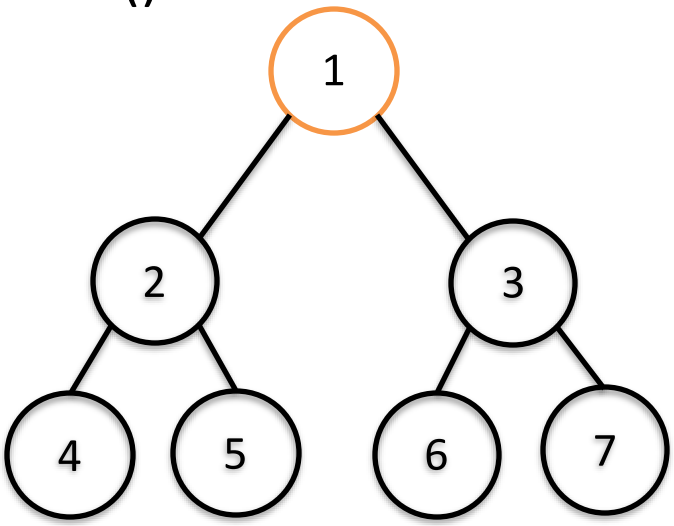
Example:

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

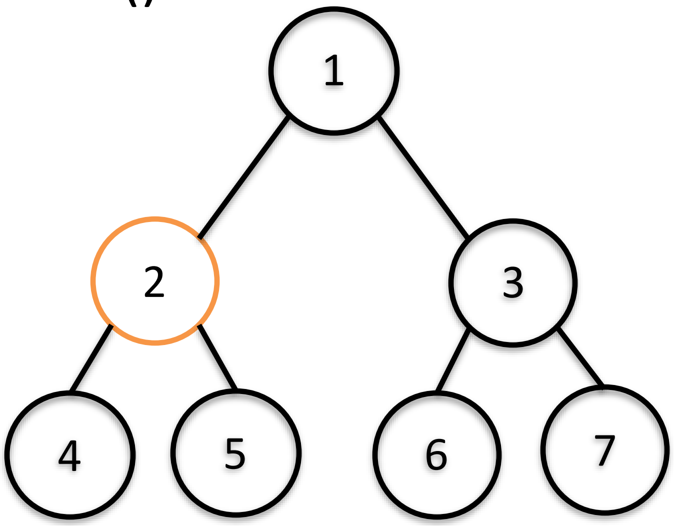


Example:

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

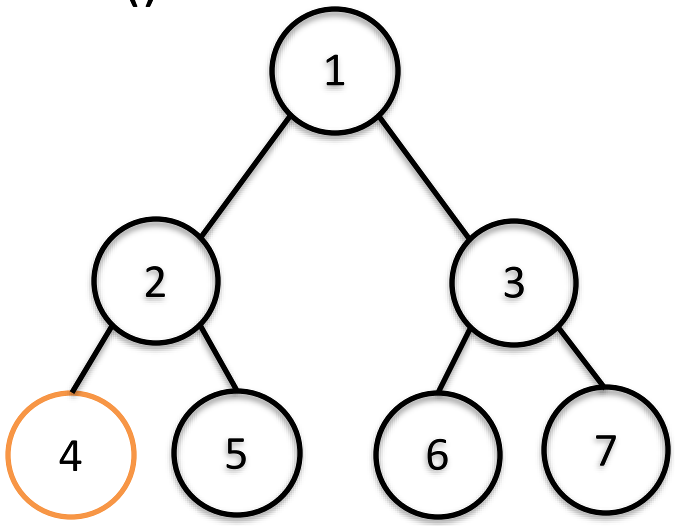


Example:
Compute disk space (not sure how many bytes in each directory until you search all children)

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

postorder()

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



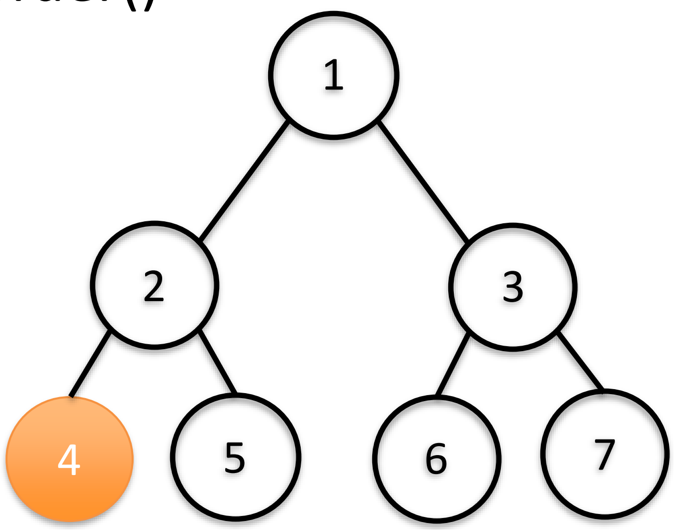
Example:

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



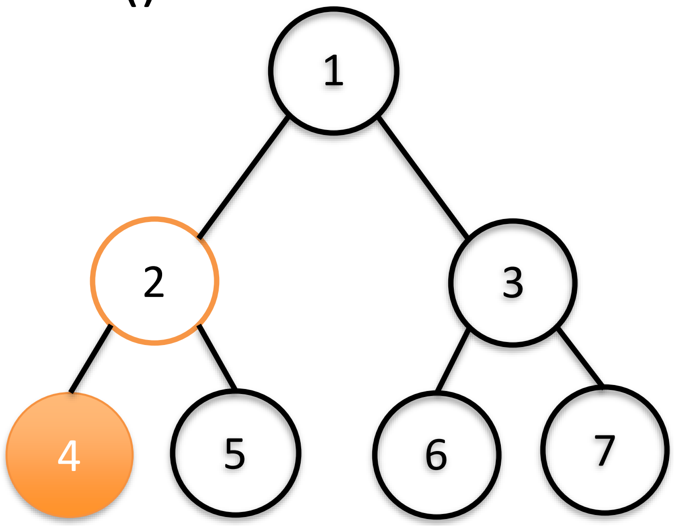
Visited
4

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



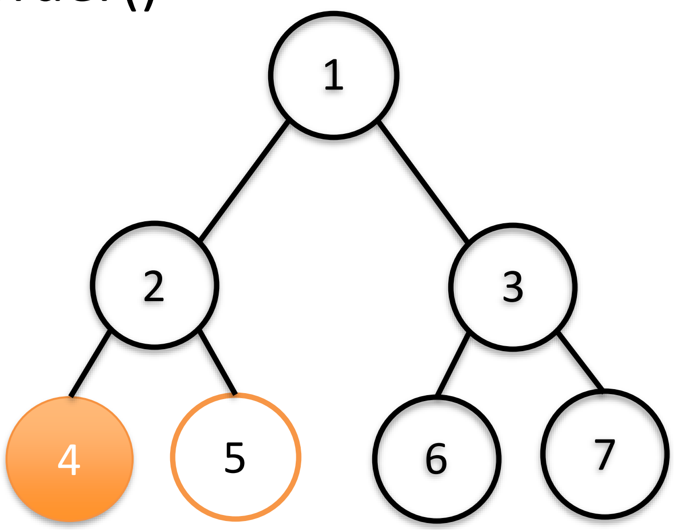
Visited
4

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



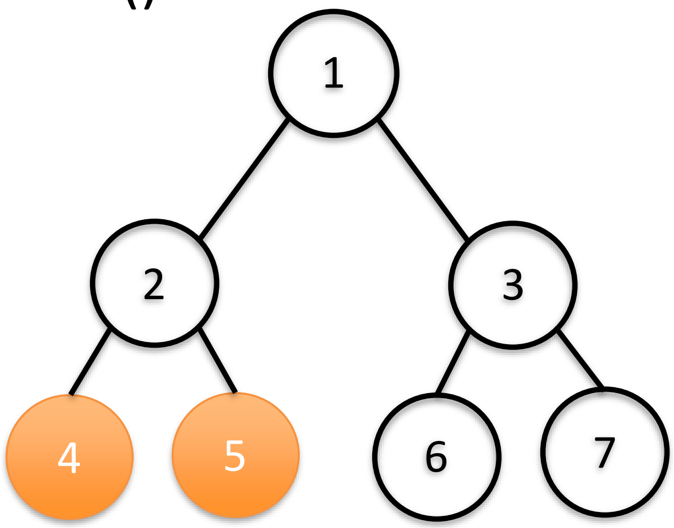
Visited
4

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



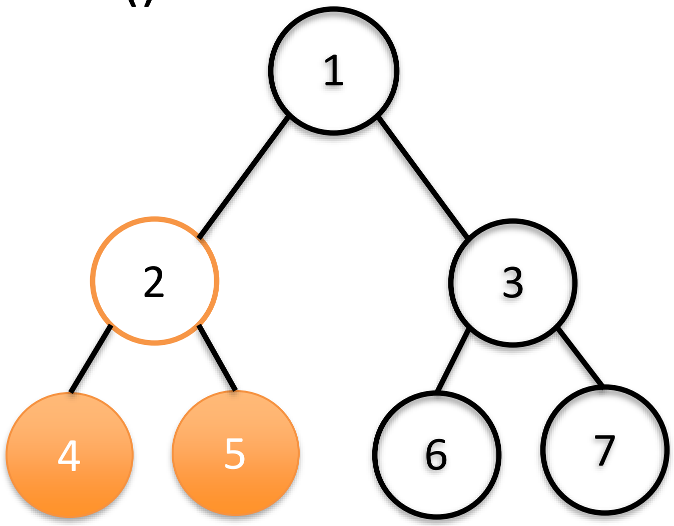
Visited
4, 5

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

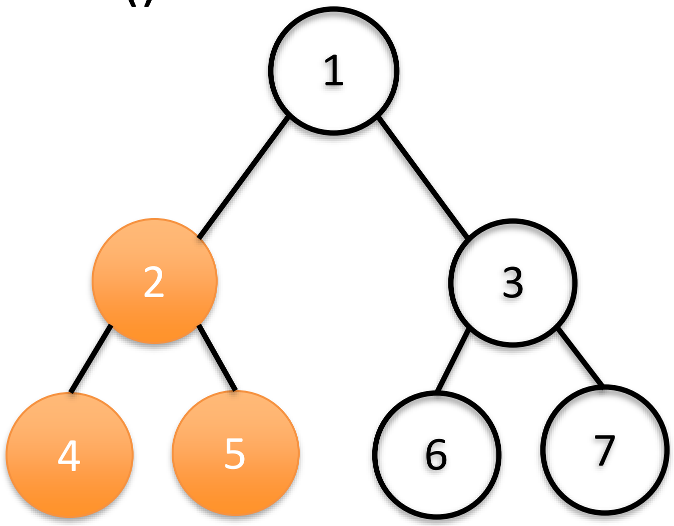


Visited
4, 5

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



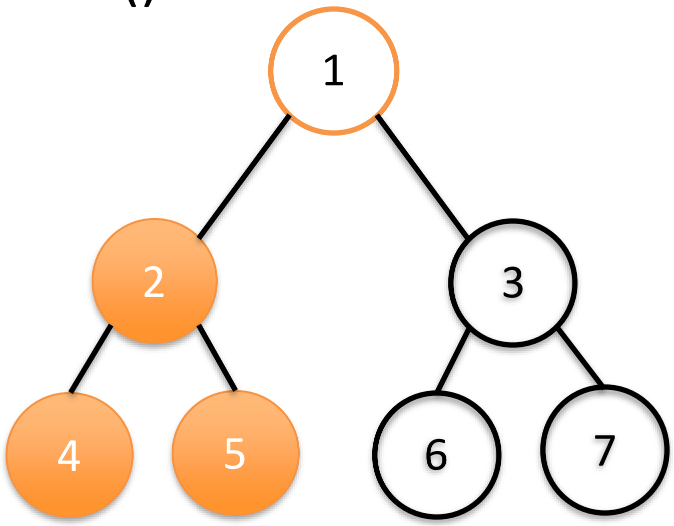
Visited
4, 5, 2

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)

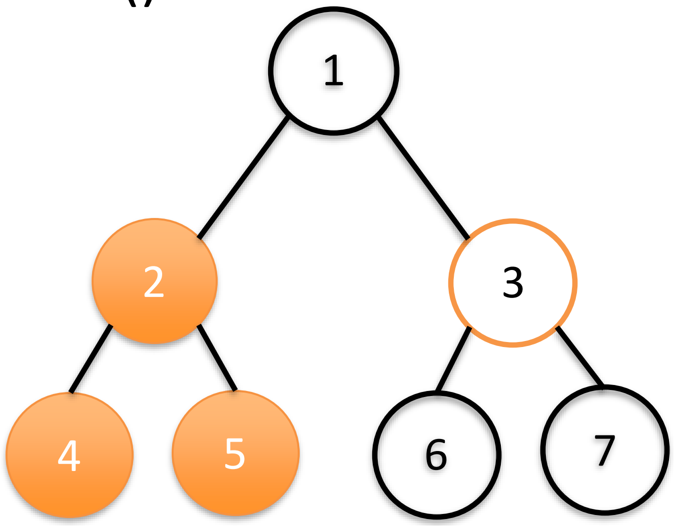
There are different ways to traverse a tree, depending on what needs to be done

postorder()

left.postorder()

right.postorder()

visit



Visited

4, 5, 2

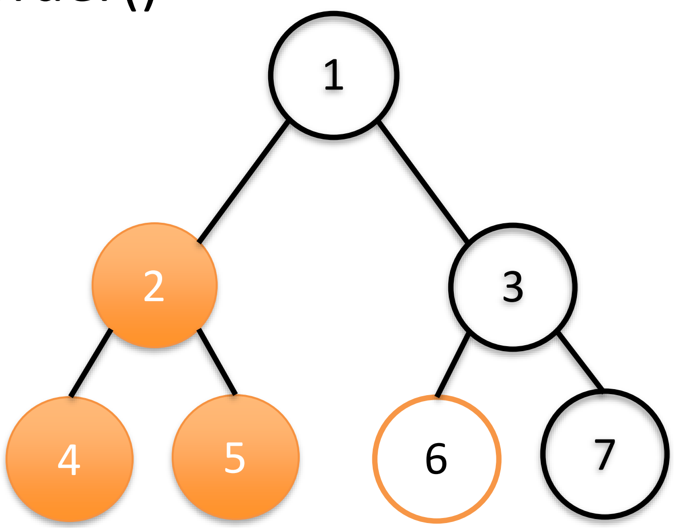
Example:

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



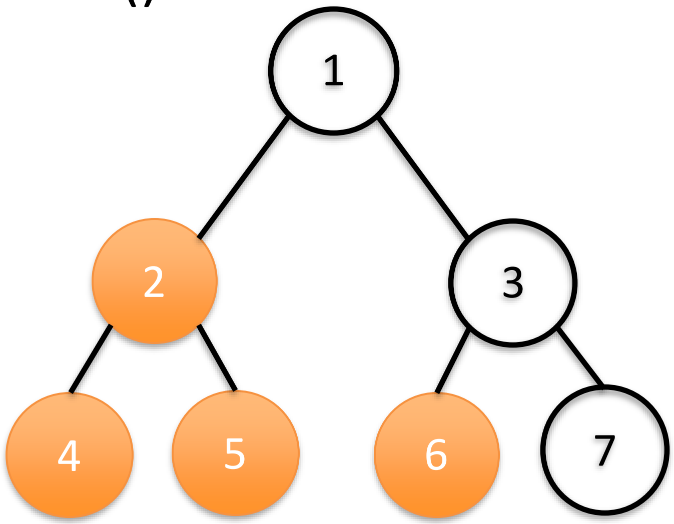
Visited
4, 5, 2

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



Visited

4, 5, 2, 6

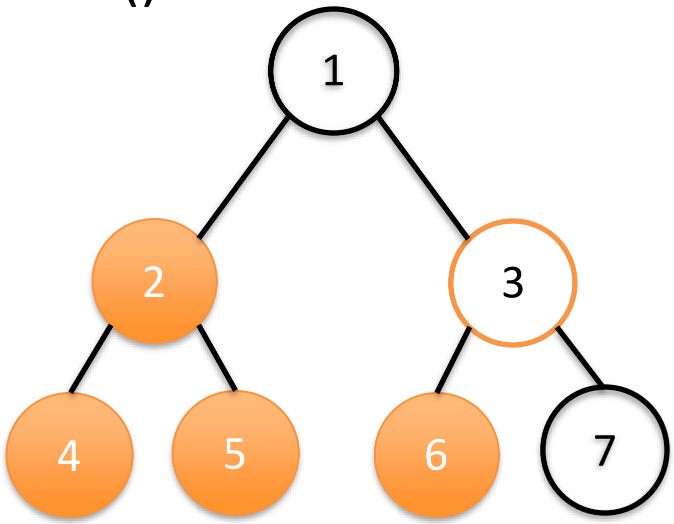
Example:

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



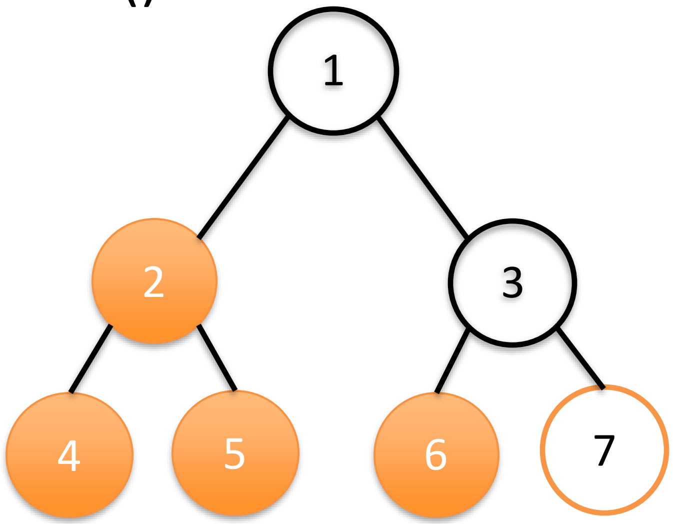
Visited
4, 5, 2, 6

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



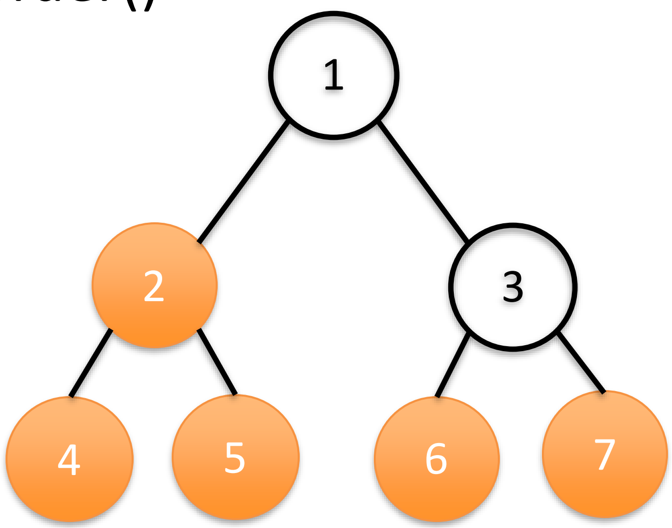
Visited
4, 5, 2, 6

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, 7

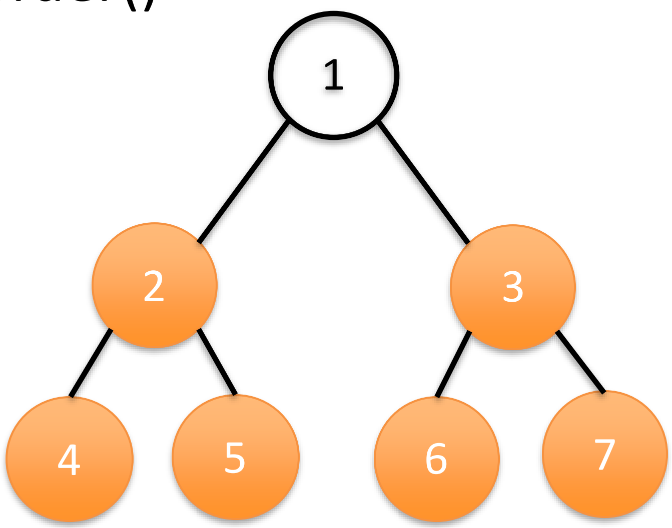
Example:

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



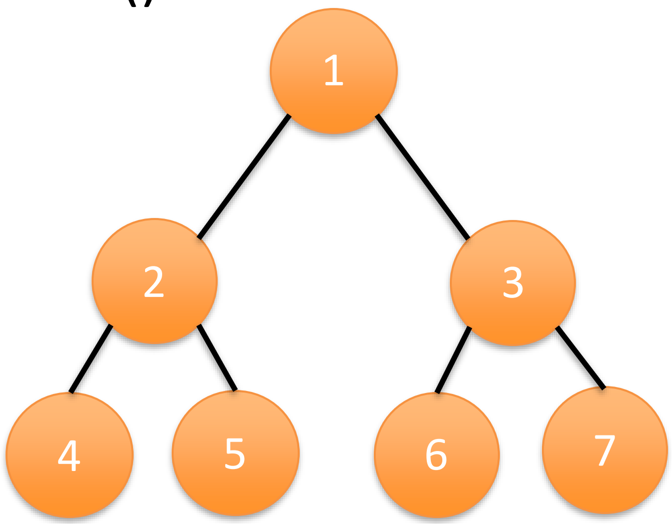
Visited
4, 5, 2, 6, 7, 3

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)

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

inorder()

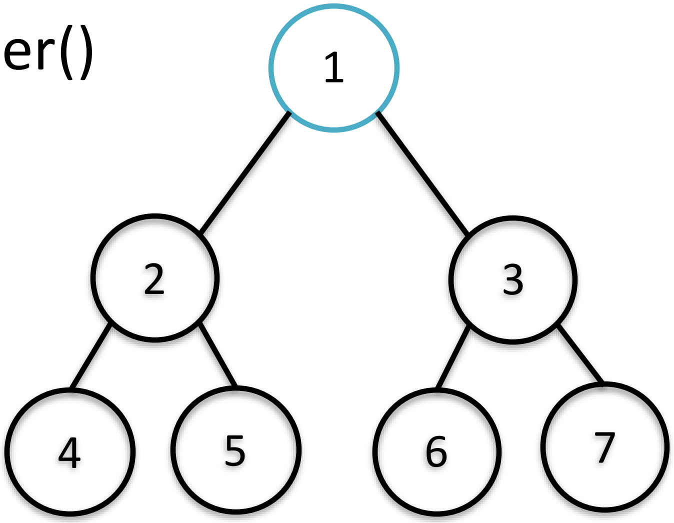
left.inorder()

visit

right.inorder()

Example:

Drawing a tree



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

inorder()

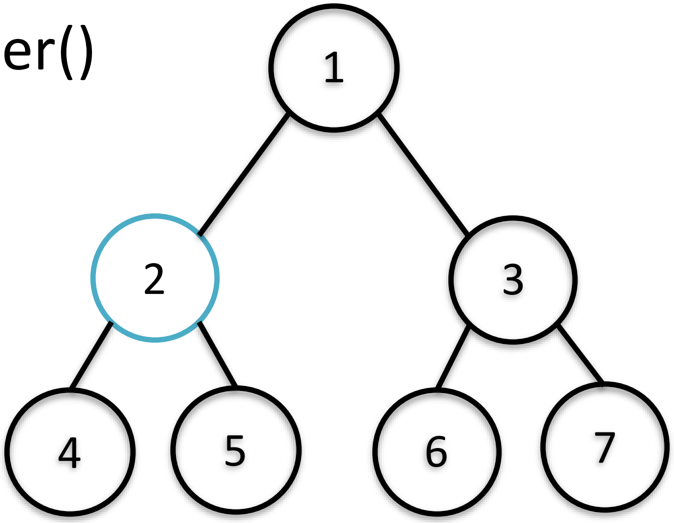
left.inorder()

visit

right.inorder()

Example:

Drawing a tree



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

inorder()

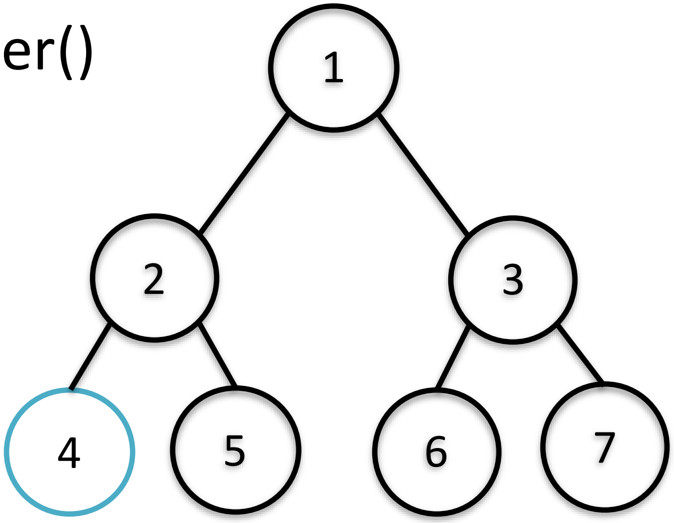
left.inorder()

visit

right.inorder()

Example:

Drawing a tree



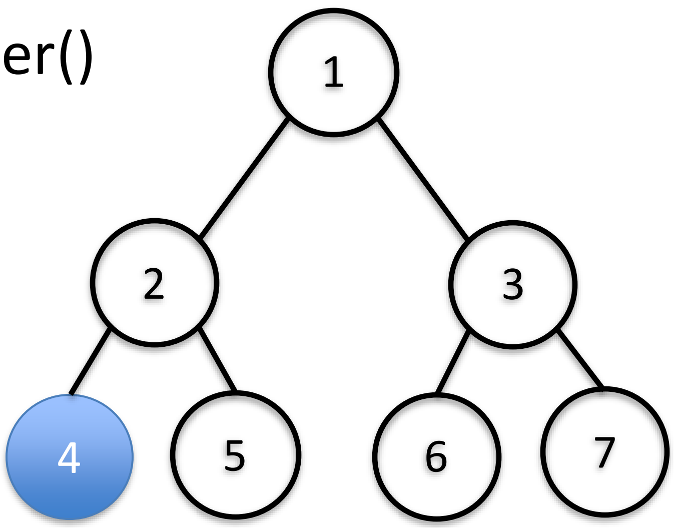
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
visit
right.inorder()

Example:

Drawing a tree



Visited

4

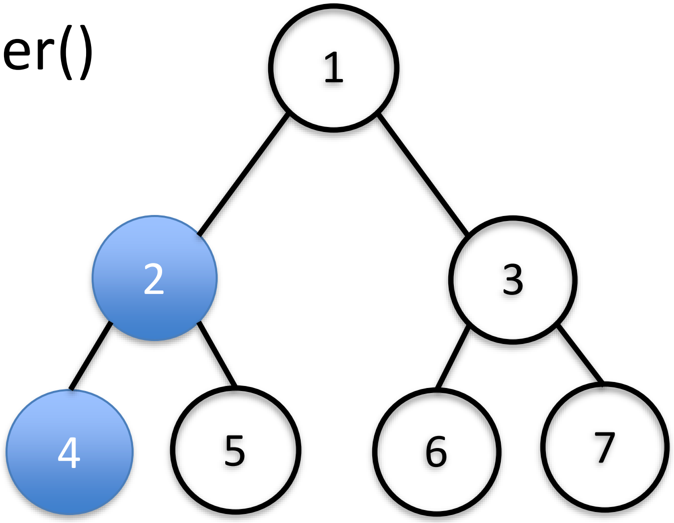
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
visit
right.inorder()

Example:

Drawing a tree



Visited

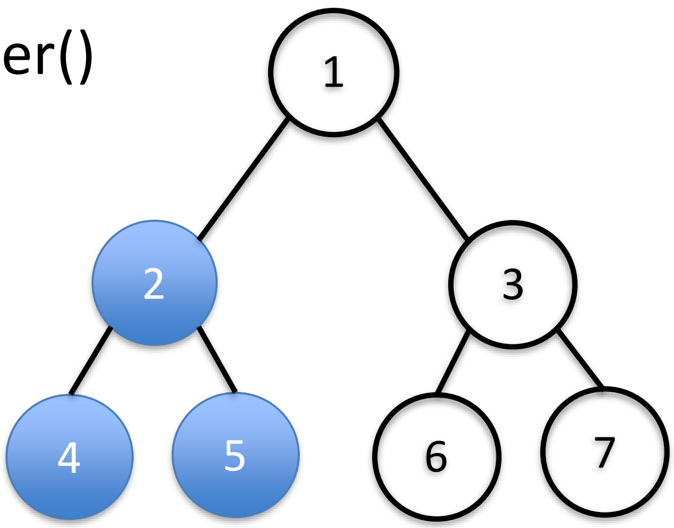
4, 2

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

inorder()

left.inorder()
visit
right.inorder()

Example:
Drawing a tree



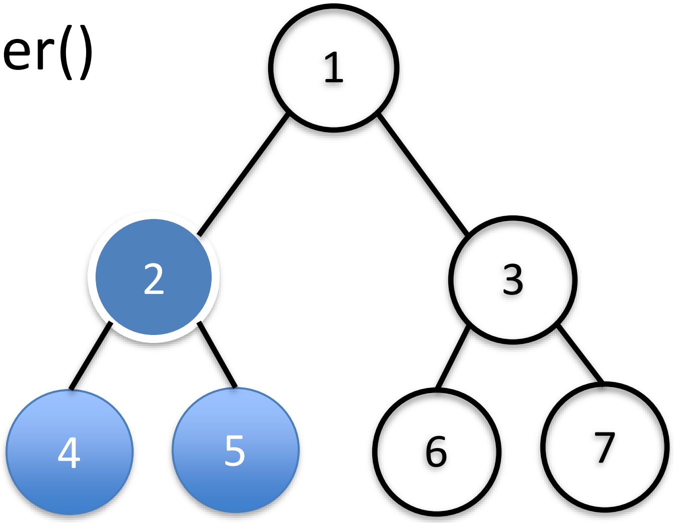
Visited
4, 2, 5

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

inorder()

left.inorder()
visit
right.inorder()

Example:
Drawing a tree



Visited
4, 2, 5

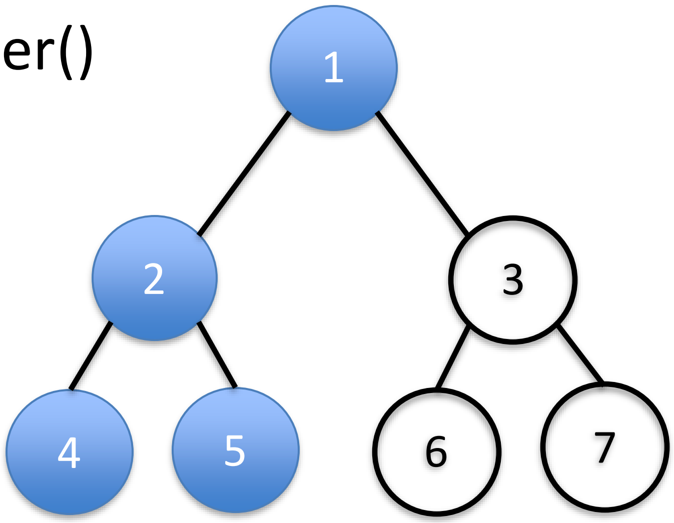
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
visit
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1

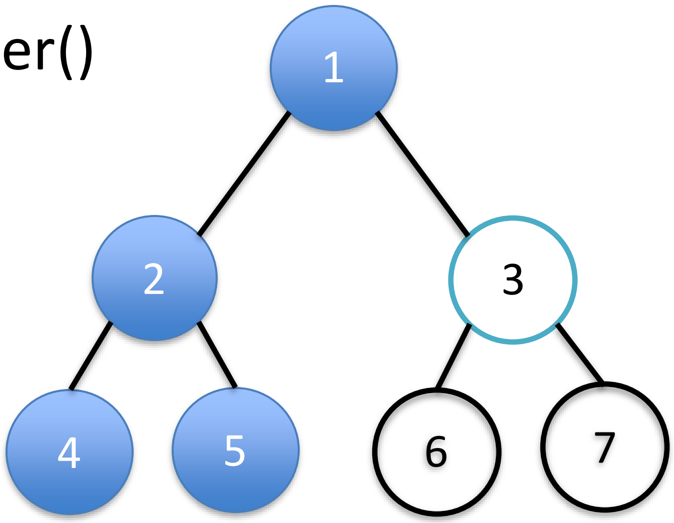
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
visit
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1

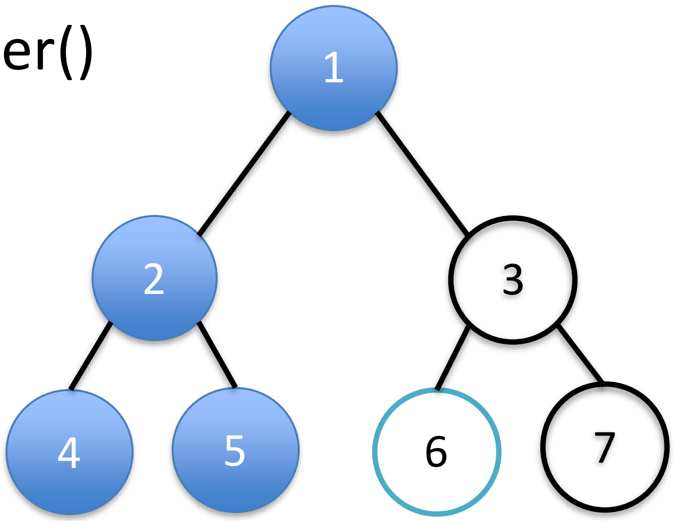
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
visit
right.inorder()

Example:

Drawing a tree



Visited

4, 2, 5, 1

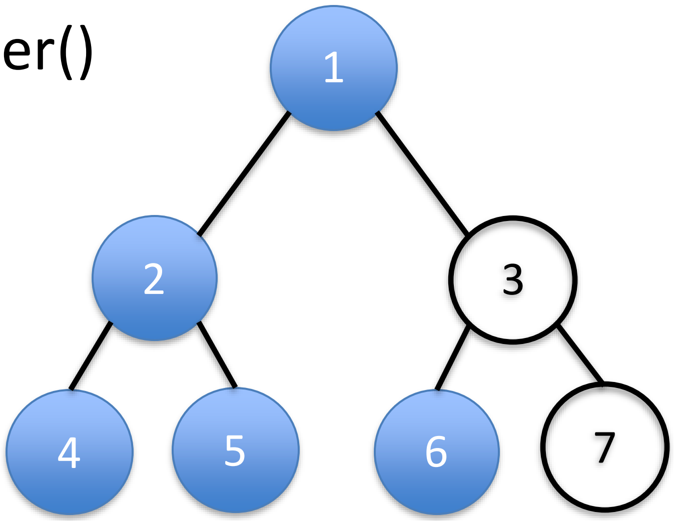
There are different ways to traverse a tree, depending on what needs to be done

inorder()

left.inorder()
visit
right.inorder()

Example:

Drawing a tree



Visited

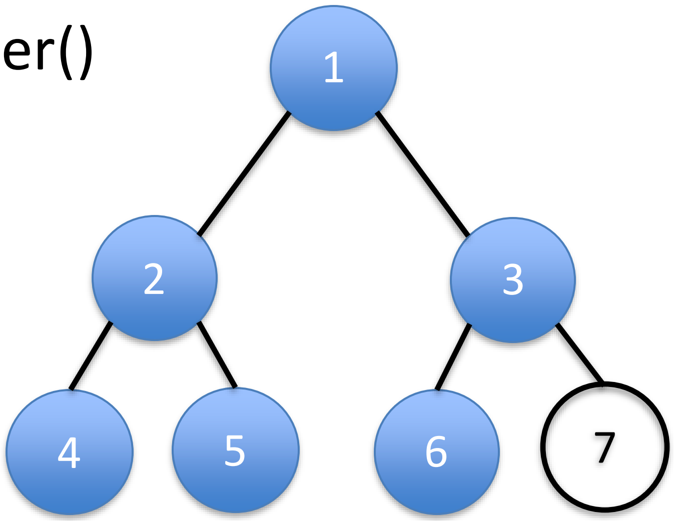
4, 2, 5, 1, 6

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

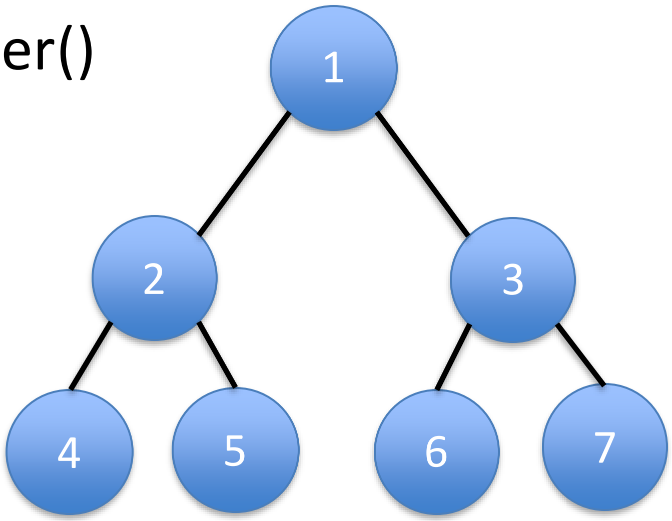


Visited
4, 2, 5, 1, 6, 3

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

inorder()

left.inorder()
visit
right.inorder()



Visited

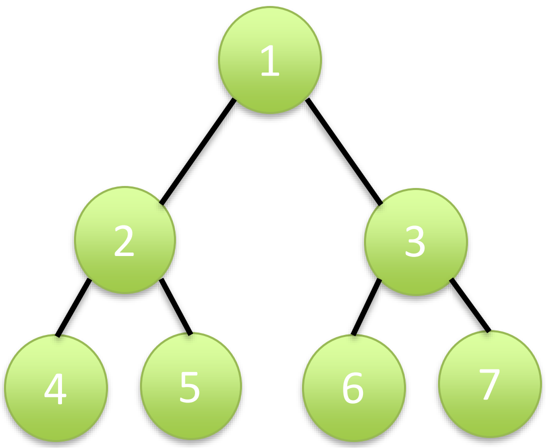
4, 2, 5, 1, 6, 3, 7

Example:

Drawing a tree

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

Preorder



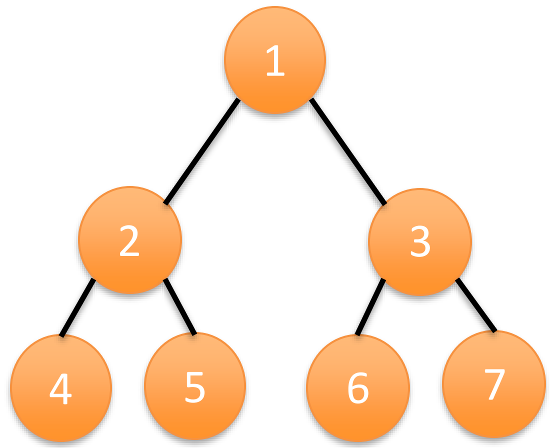
Visited

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

Book chapters

toString()

Postorder

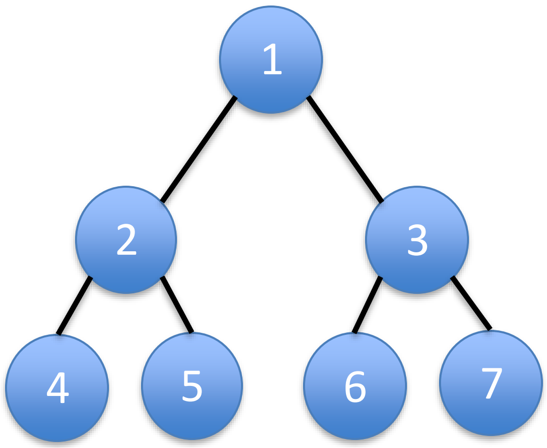


Visited

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)

Key points

1. Trees are useful for hierarchical data
2. Binary trees have 0, 1, or 2 children at each node
3. Not all trees are binary (PS-2 isn’t)
4. Trees may not be “balanced”
5. Trees lead to beautiful recursive code (so beautiful it brings a tear to my eye!)
6. Accumulators are a way to “build up” a value as a tree is traversed
7. Accumulators allow efficient code
8. Trees are commonly traversed in three ways
    1. Pre-order
    2. Post-order
    3. In-order