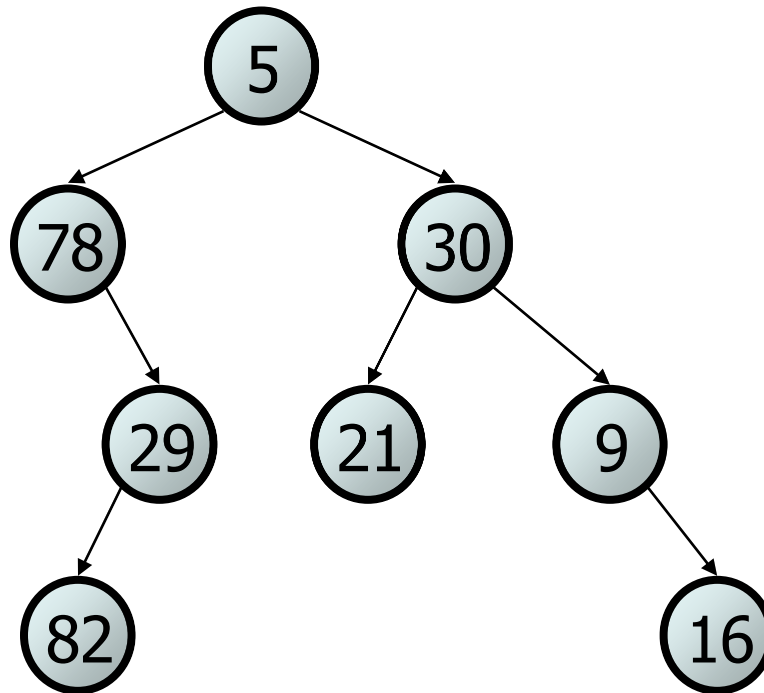
# CSE 143
# Lecture 15

## Binary Trees

slides created by Alyssa Harding
http://www.cs.washington.edu/143/

# Binary trees

- Another data structure, shaped like an upside down tree:

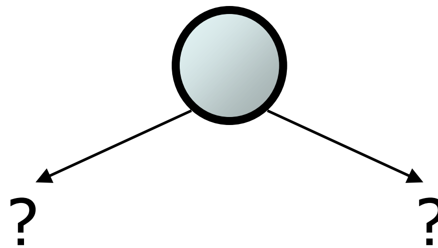


# Definition

- A binary tree is either
  (a) an empty tree or

See?  No tree!

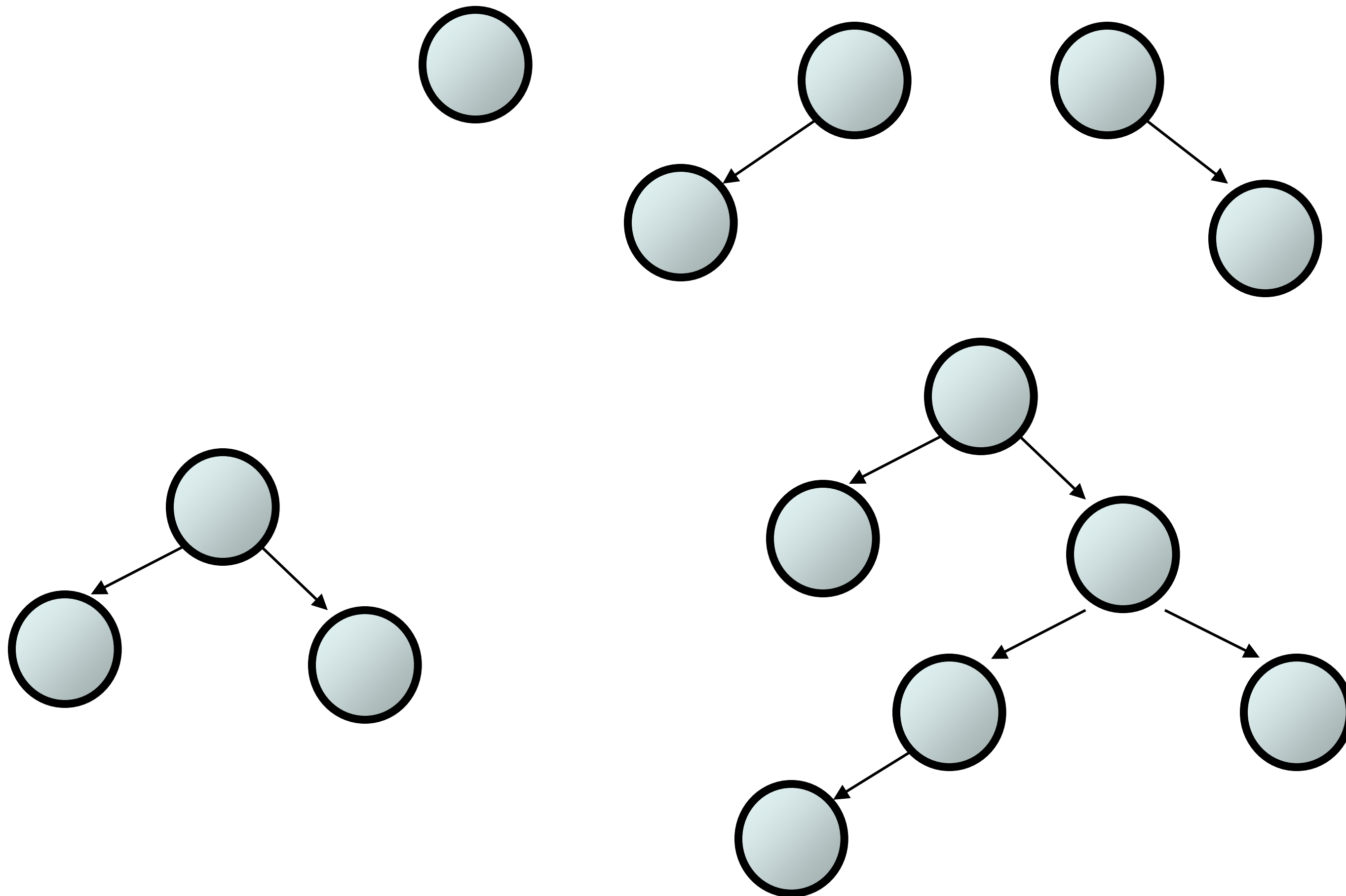
  (b) a root node with a left subtree and a right subtree




- This definition is *recursive*

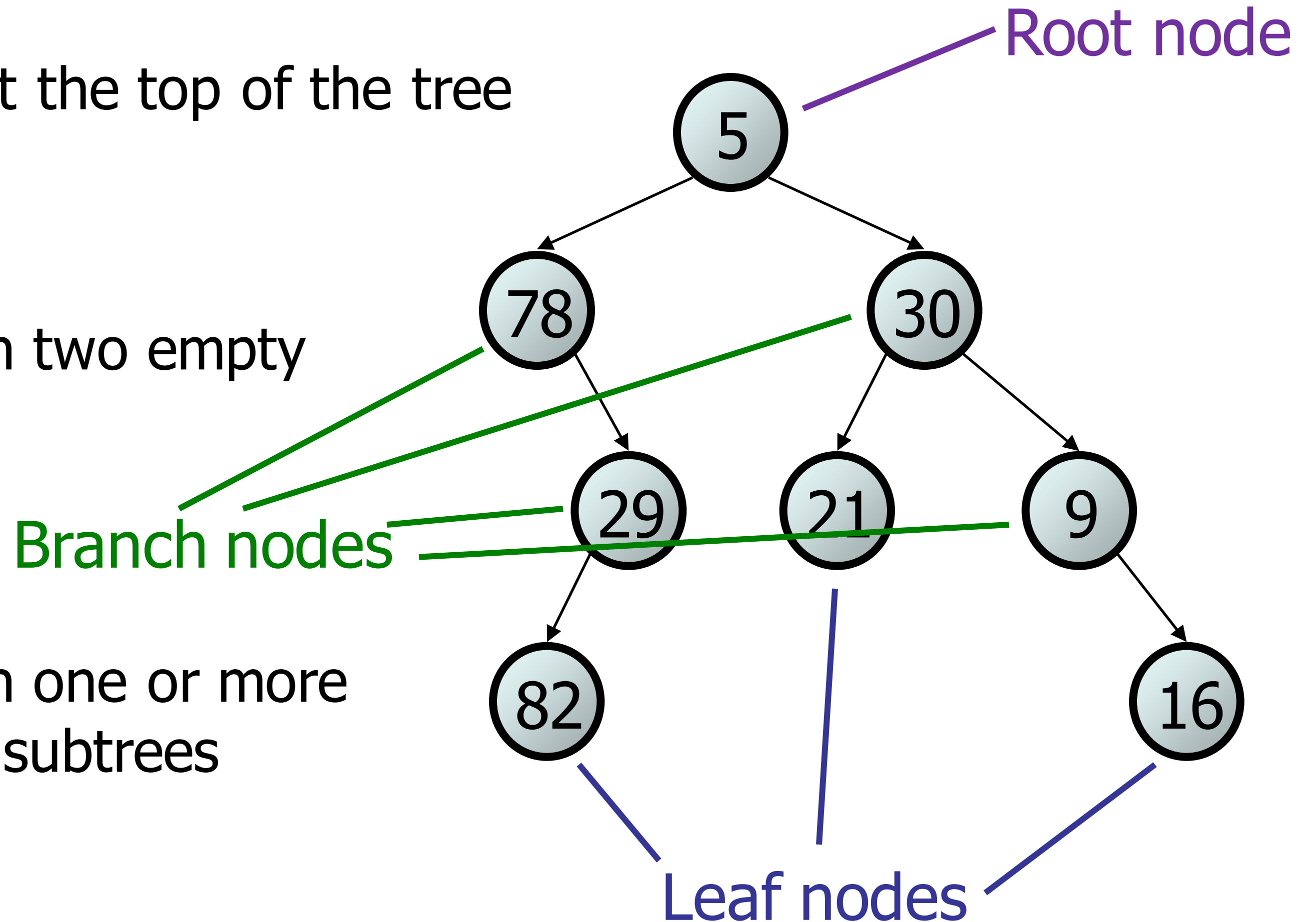
# Definition

- The recursive definition lets us build any shape tree:



# Terminology

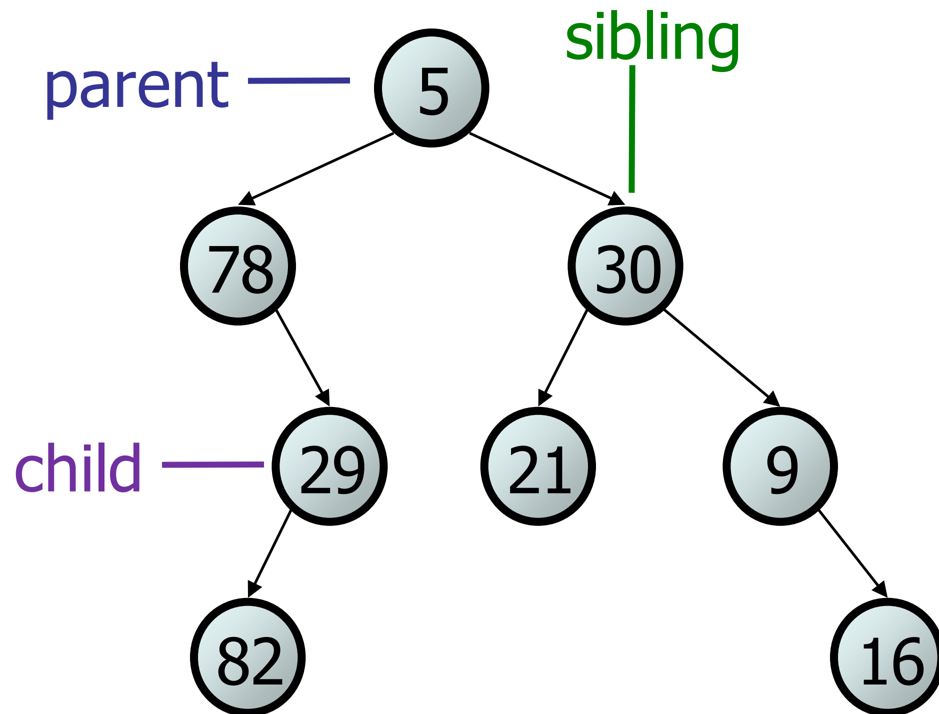
- Root
  - The node at the top of the tree
- Leaf
  - A node with two empty subtrees
- Branch
  - A node with one or more non-empty subtrees



# Terminology

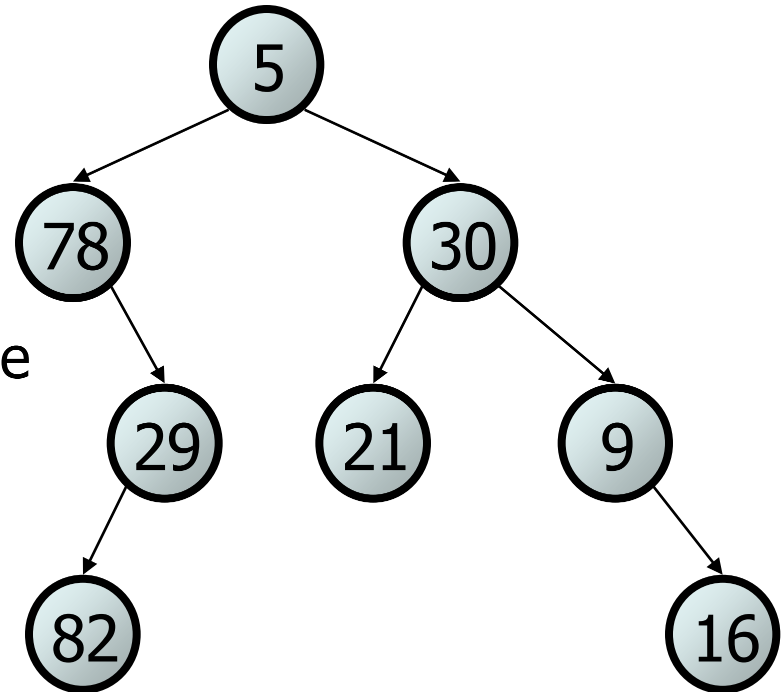
- Child
  - Any node our node refers to
- Parent
  - The node that refers to our node
- Sibling
  - Another child of the parent of our node

Looking at our 78 node:



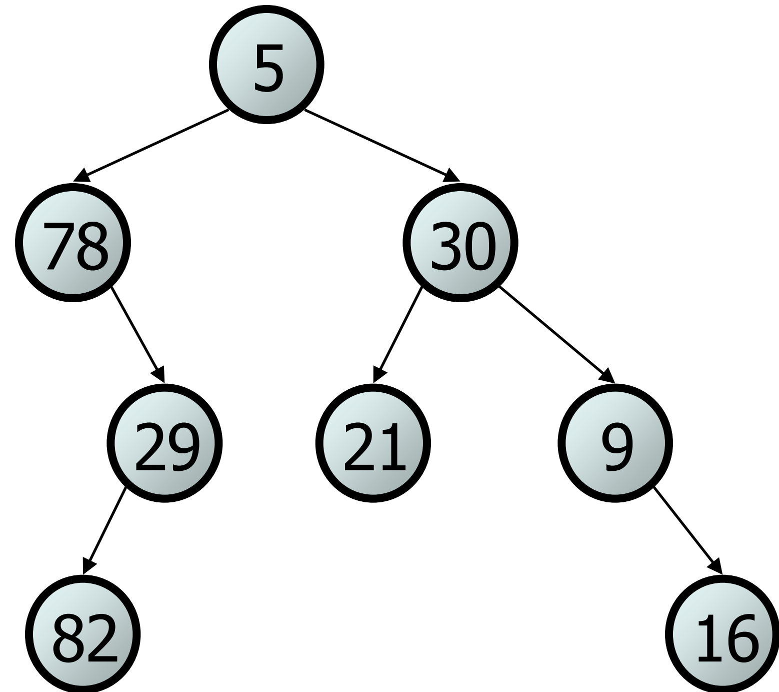
# Terminology

- Ancestor
  - A parent of a parent of…our node
  - 5 is an ancestor of 82
- Descendent
  - A child of a child of…our node
  - 16 is a descendent of 30



# Terminology

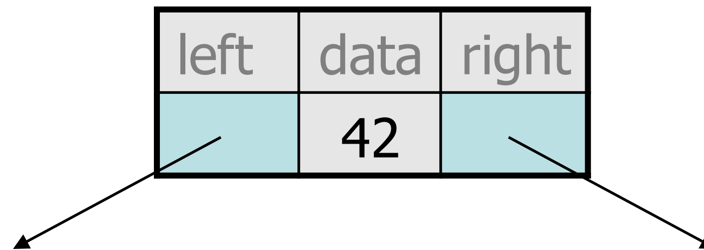
- Depth of a node
  - Length of the path from the root to the node
  - Depth of the 29 node is 2
- Height
  - Length of longest path from the root to a node
  - Height is 3



# `IntTreeNode`

- So how do we make these trees?

- We need building blocks
  - For our **`LinkedIntList`**, we had **`IntNode`**s
  - For our **`IntTree`**, we have **`IntTreeNode`**s



# IntTreeNode

- Our new building block has two pointers

```
public class IntTreeNode {
    public int data;
    public IntTreeNode left;
    public IntTreeNode right;

    public IntTreeNode(int data) {
      this(data, null, null);
    }

    public IntTreeNode(int data, IntTreeNode left,
                       IntTreeNode right) {
      this.data = data;
      this.left = left;
      this.right = right;
    }
}
```

# IntTree

- We encapsulate the building blocks in a class:

```
public class IntTree {
    private IntTreeNode overallRoot;


    ...


}
```

The client never sees the nodes,
And we have keep track of the root
of our entire tree

# `IntTree`

- We have code that will build a random tree of a given height
- We have code that prints the structure of the tree
- We can use JGrasp to view the tree

# Traversals

- Great, but what if we want to print out one line of output?

- It's not like a list where we know what order to print in
  - We need to print the root node's data
  - We need to print the left subtree
  - We need to print the right subtree

- We get different **traversal order** from choosing different orders to process the tree

# Traversals

- Preorder:
  root, left, right

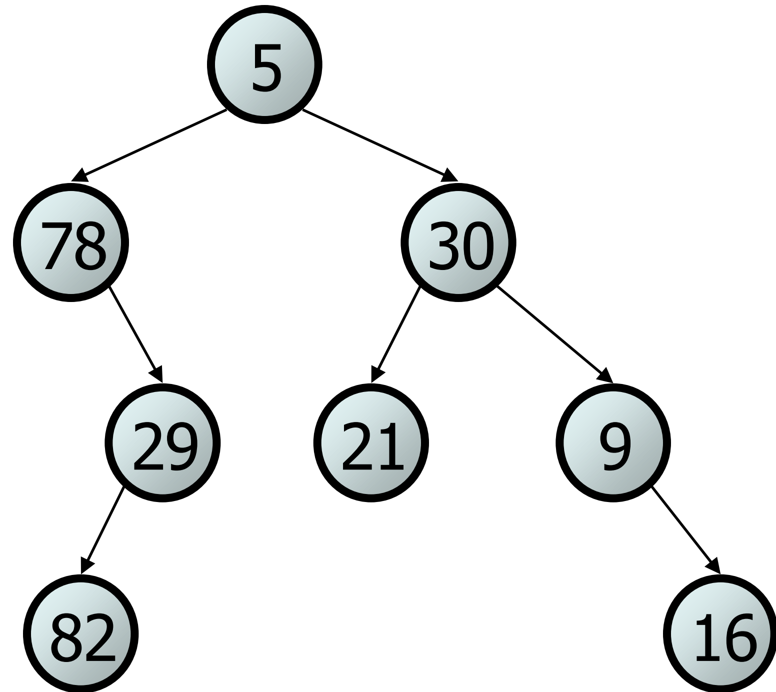
  5 78 29 82 30 21 9 16

- Inorder:
  left, root, right

  78 82 29 5 21 30 9 16

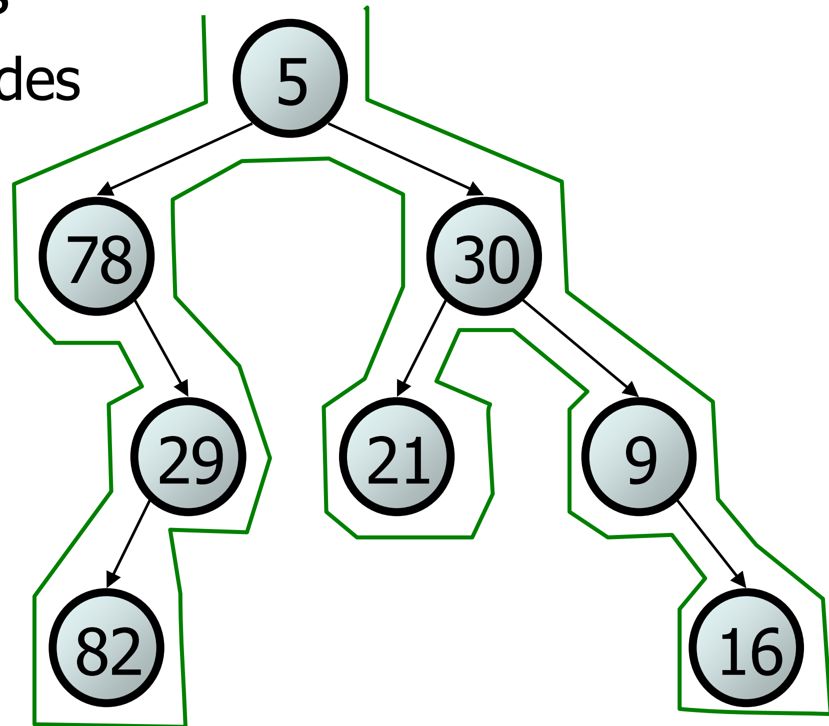
- Postorder:
  left, right, root

  82 29 78 21 16 9 30 5



# Traversals

- Sailboat method:
  A visual way to do traversals
  - Trace a path around the nodes
  - Write down the data of the node when you pass…
    - On its left, for a **preorder traversal**
    - Under it, for an **inorder traversal**
    - On its right, for a **postorder traversal**



# Example: `printPreorder`

- Now we want a method to print the preorder traversal:

```
public void printPreorder() {
    ...
}
```

We need to know which node we're examining

# Example: `printPreorder`

- We make a private helper method to look at one specific node:

```
public void printPreorder() {
    System.out.println("Preorder:");
    printPreorder(overallRoot);
    System.out.println();
}

private void printPreorder(IntTreeNode root) {
    ...
}
```

The public method also starts the whole process by calling the private method with the **`overallRoot`**

# Example: `printPreorder`

- What is our base case? A null node is an empty tree!

```
private void printPreorder(IntTreeNode root) {
    if ( root == null ) {
        // do nothing?
    } else {
        ...
    }
}
```

Instead of having an empty if statement,
invert the test!

# Example: `printPreorder`

- What is our recursive case?
  Since it's preorder, we first want to print the root's data:

```
private void printPreorder(IntTreeNode root) {
   if ( root != null ) {
      System.out.print(root.data + " ");
   }
}
```

# Example: `printPreorder`

- We also want to print a preorder traversal of the left subtree. If only we had a method...

```
private void printPreorder(IntTreeNode root) {
   if ( root != null ) {
      System.out.print(root.data + " ");
      printPreorder(root.left);
   }
}
```

# Example: `printPreorder`

- The last part is the right subtree:

```
private void printPreorder(IntTreeNode root) {
  if ( root != null ) {
    System.out.print(root.data + " ");
    printPreorder(root.left);
    printPreorder(root.right);
  }
}
```

# Example: `printPreorder`

- It's amazingly short code, just like we've seen before with recursion
- When we call our recursive method, it prints the entire subtree
- The code for **`printInorder`** and **`printPostorder`** are very similar
  - Remember, the difference was in the order in which we processed the root, the left subtree, and the right subtree

# Example: `printInorder`

- For an inorder traversal, we process the root in the middle:

```
private void printInorder(IntTreeNode root) {
  if ( root != null ) {
     printPreorder(root.left);
     System.out.print(root.data + " ");
     printPreorder(root.right);
  }
}
```

# Example: printPostorder

- For a postorder traversal, we process the root last:

```
private void printPostorder(IntTreeNode root) {
  if ( root != null ) {
     printPreorder(root.left);
     printPreorder(root.right);
     System.out.print(root.data + " ");
  }
}
```