Without telling you what a binary tree is, here are some examples (that I will draw on the board):

The dots/circles are called **nodes**, or **vertices** (singular: **one vertex**). The lines (that connect 2 nodes) are called **edges**, or sometimes **arcs**. A tree has a designated **root** node, typically drawn at the “top” of the tree.
We can define a binary tree recursively, as follows:
A binary tree is either

- an empty tree
- a root node $r$, connected by an edge to up to two non-empty trees

Every node except the root has exactly one parent node (above it).
Every node $v$ has a unique path from the root to $v$.
A tree has no cycles (i.e., loops).
The number of edges in a tree is one less than the number of nodes.
Trees come with a lot of special vocabulary that you should get familiar with!

- **a leaf node** is a node with no non-empty children.
- **an internal node** is any node that is not a leaf node.
- **a (simple) path** is a sequence of consecutive edges.
- **a cycle** is a sequence of consecutive edges that starts and ends at the same vertex (a tree is acyclic, that is, it has no cycles.)
- **the length of a path** is the number of edges in that path.
  - fact: the number of nodes in a path = 1 + the number of edges in the path.
- **the depth, or level,** of a node $v$ is the length of the path from the root to $v$.
- **the height of a tree** is the length of the longest path from the root to any node.
  - by this definition, the height of the single-node tree is 0.
  - some people define height differently, so that height is one more than this.
- **the degree of a node** is the number of edges incident to it.
- **ancestor, descendant, sibling:** are exactly what you expect...
- **a forest** is a collection of trees.
- **a directed tree** is one where the edges are arrows from one node to the other.
  - in directed trees, a node has both in-degree and out-degree.
A binary tree stores data in its nodes, like a linked list. Some tree operations:

- `size()` - return the number of nodes in the tree
- `isEmpty()` - return whether this tree has any nodes
- `isLeaf()` - return whether this tree is a single leaf node
- `height()` - return the height of the tree
- `leafCount()` - return the number of leaves in the tree
- `nodeCount()` - return the number of nodes in the tree
- `mirrorImage()` - return a tree that is the mirror image of this tree
- `traverse()`/`iterate()` - go through the nodes of the tree in a specific order
- many, many others
Just like the recursive LL implementation we’ll have 3 separate public classes:

/* This abstract class represents a Binary Tree.
   It defines the methods that every binary tree must have,
   but doesn’t implement them. We will have different
   implementations depending on whether our tree is empty, or
   is a “ConsTree”, i.e. a root with left & right subtrees. */
public abstract class BinaryTree<T> {
   // method headers for Binary Tree methods go here
}

public class EmptyTree<T> extends BinaryTree<T> {
   // method implementations for empty trees go here
}

public class ConsTree<T> extends BinaryTree<T> {
   T               data    // the data of the root node
   BinaryTree<T>   left    // the left subtree
   BinaryTree<T>   right   // the right subtree
   // method implementations for non-empty trees go here
}
Recursive Binary Tree Implementation

To add a method to our BinaryTree class, we must do three things:
- add the method as an abstract method header in the BinaryTree class
- implement the method in the EmptyTree class
- implement the method in the ConsTree class

For example, let’s start with the isEmpty() method.

We first add the method header to the BinaryTree class, to declare that any class that implements BinaryTree must have the isEmpty method.

```java
public abstract class BinaryTree<T> {
    public abstract boolean isEmpty();
}
```

In the EmptyTree class, we implement the method.

```java
public class EmptyTree<T> {
    public boolean isEmpty() { return true; } // always.
}
```

In the ConsTree class, we implement the method.

```java
public class ConsTree<T> {
    public boolean isEmpty() { return false; } // always.
}
```
Recursive Binary Tree Implementation

Now let's try the size() method that returns the number of nodes in the tree:

We add the method header to the abstract LinkedList class:

```java
public abstract class BinaryTree<T> {
    public abstract int size();
}
```

We implement the method in the Empty class.

```java
public class EmptyTree<T> {
    public int size() { return 0; } // "base case"
}
```

We implement size in the Cons class. What is the size of this cons list?

```java
public class ConsTree<T> {
    T data
    BinaryTree<T> left
    BinaryTree<T> right

    public int size() {
        return 1 + left.size() + right.size()
    }
}
Now let’s try the `height()` method, which is the length of the longest root-leaf path in the tree.

```java
public abstract class BinaryTree<T> {
    public abstract int height();
}
```

```java
public class EmptyTree<T> {
    public int height() { return -1; }
}
```

```java
public class ConsTree<T> {
    T data
    BinaryTree<T> left
    BinaryTree<T> right

    public int size() {
        return 1 + Math.max( left.height(), right.height() )
    }
}
```
Binary Tree Traversals

There are many ways to traverse the tree such that each node is “visited” exactly once, in some order.

The three steps for common tree traversals:
1. visit the current node
2. recursively traverse the left subtree
3. recursively traverse the right subtree

Different orderings of these three steps leads to three different traversals!
pre-order traversal: 1-2-3
  (traverse the root node FIRST (pre), then do the recursive calls.)
post-order traversal: 2-3-1
  (do the recursive calls first, then traverse the root node (POST).)
in-order traversal: 2-1-3
  (traverse the root node IN between the two recursive calls.)
A couple of other important traversals are as follows:

A breadth-first search (bfs) is a level-order traversal, where you visit the nodes in the tree level-by-level from the root down to the leaves.

A depth-first search (dfs) starts at the root, visits nodes along one path until it hits a leaf, then backs up as little as possible before going down a different path to a leaf.

(Hopefully this reminds you of backtracking, also, it is the same as preorder.)