A tree is a non-linear hierarchical structure.
Tree is comprised of a set of nodes in which elements are stored and edges connect one node to another.

- A node can have only one parent, but may have multiple children.
- Nodes that have the same parent are siblings.
- The root is the only node which has no parent.
- A node that has no children is a leaf node.

**Tree Terminology**

- A node that is not the root and has at least one child is an internal node.
- A subtree is a tree structure that makes up part of another tree.
- We can follow a path through a tree from parent to child, starting at the root.
- A node is an ancestor of another node if it is above it on the path from the root.
- Nodes that can be reached by following a path from a particular node are the descendants of that node.
- The level of a node is the length of the path from the root to the node.
- The path length is determined by counting the number of edges that must be followed to get from the root to the node.
- The height (or depth) of a tree is the length of the longest path from the root to a leaf.

**Tree Classifications**

- We classify trees by the maximum number of children any node in the tree may have.
- This is sometimes referred to as the order of the tree.
- General trees have no limit to the number of children a node may have.
- A tree that limits each node to no more than $n$ children is referred to as an $n$-ary tree.
- Trees in which nodes may have at most two children are called binary trees.

**Balanced, Complete and Full**

- A tree is balanced if all of the leaves of the tree are on the same level or at least within one level of each other.
- A balanced binary tree with $n$ nodes has a height of $O\left(\log(n)\right)$.
- A balanced $n$-ary tree with $m$ nodes will have a height of $O\left(\log(m)\right)$.
- An $n$-ary tree is full if all leaves of the tree are at the same height and every non-leaf node has exactly $n$ children.
- A tree is complete if it is full, or full to the next-to-last level with all leaves at the bottom level on the left side of the tree.
Tree Traversals

- Traversing a tree (visiting all nodes in a sequence) is generally more interesting than traversing a linear structure.
- A particular type of traversal simply dictates the order in which the elements of a collection are assessed.

Traversals:

- Preorder: Visit the root in between the traversals of the left and right subtrees.
  - Traverse (left)
  - Visit Node
  - Traverse (right)

- Inorder: Nodes are visited before any subtrees are visited.
  - Visit Node
  - Traverse (left)
  - Traverse (right)

- Postorder: Visit the root node after the traversals of the left and right subtrees.
  - Traverse (left)
  - Traverse (right)
  - Visit Node

Level-Order Traversal

- Visit the nodes on each level, left to right, top to bottom starting at the root.

Strategies for Implementing Trees

- Array-based implementations are the less obvious choice, but sometimes useful.
- Computed Links in an Array:
  - Where can you find D’s children?
  - How about D’s parent?
Stored Links in an Array

- Array positions are allocated on a first-come, first-served basis.
- Each element of the array is an object that stores a reference to the tree element and the array index of each child.

Linked Nodes

- Each tree node can be defined using a separate class—similar to ListNode or linked lists.
- Nodes contain a reference to the data stored in the node, and references for each of the possible children of the node.
  - Binary tree: 2 references required—left and right children.
  - n-ary tree: n references required—one for each possible child.
- Trees organized this way lend themselves to recursive processing for many operations.

```
16.4 – javafoundations.BinaryTree

package javafoundations;
import java.util.Iterator;
public interface BinaryTree<T> extends Iterable<T> {
    // Returns the element stored in the root of the tree.
    public T getRootElement();
    // Returns the left subtree of the root.
    public BinaryTree<T> getLeft();
    // Returns the right subtree of the root.
    public BinaryTree<T> getRight();
    // Returns true if the binary tree contains an element that matches the specified element and false otherwise.
    public boolean contains (T target);
    // Performs a level-order traversal on the binary tree.
    public Iterator<T> levelorder();
}
```
A possible set of operations for a binary tree is shown in the BinaryTree interface.

- BinaryTree has no methods to add a particular element or to remove a particular element from the tree.
- Refined versions of binary tree (such as binary search trees) will define those methods based on specific characteristics.
- BinaryTree is still useful in certain situations.

```java
package javafoundations;

public class BTNode<T> {
    protected T element;
    protected BTNode<T> left, right;

    // Creates a new tree node with the specified data.
    public BTNode (T element) {
        this.element = element;
        left = right = null;
    }

    // Returns the element stored in this node.
    public T getElement() {
        return element;
    }

    // Sets the element stored in this node.
    public void setElement (T element) {
        this.element = element;
    }

    // Returns the left subtree of this node.
    public BTNode<T> getLeft() {
        return left;
    }

    // Sets the left child of this node.
    public void setLeft (BTNode<T> left) {
        this.left = left;
    }

    // Returns the right subtree of this node.
    public BTNode<T> getRight() {
        return right;
    }

    // Sets the right child of this node.
    public void setRight (BTNode<T> right) {
        this.right = right;
    }
}
```
16.4 – javafoundations.BTNode

```java
public BTNode<T> find (T target)
{
    BTNode<T> result = null;
    if (element.equals(target))
        result = this;
    else
    {
        if (left != null)
            result = left.find(target);
        if (result == null && right != null)
            result = right.find(target);
    }
    return result;
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public void inorder (ArrayIterator<T> iter)
{
    if (left != null)
        left.inorder (iter);
    iter.add (element);
    if (right != null)
        right.inorder (iter);
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public LinkedBinaryTree (T element)
{
    root = new BTNode<T>(element);
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public class LinkedBinaryTree<T> implements BinaryTree<T>
{
    protected BTNode<T> root;

    public LinkedBinaryTree()
    {
        root = null;
    }

    public LinkedBinaryTree (T element)
    {
        root = new BTNode<T>(element);
    }
```
16.4 – javafoundations.LinkedBinaryTree

```java
public LinkedBinaryTree(T element, LinkedBinaryTree<T> left, LinkedBinaryTree<T> right)
{
    this.root = new BTNode<T>(element);
    this.root.setLeft(left.root);
    this.root.setRight(right.root);
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public LinkedBinaryTree<T> getLeft()
{
    if (root == null)
        throw new EmptyCollectionException("Get left failed. Tree is empty.");
    LinkedBinaryTree<T> result = new LinkedBinaryTree<T>();
    result.root = this.root.getLeft();
    return result;
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public T getRootElement()
{
    if (root == null)
        throw new EmptyCollectionException("Get root failed. Tree is empty");
    return root.getElement();
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public int size()
{
    int result = 0;
    if (root != null)
        result = root.count();
    return result;
}
```

16.4 – javafoundations.LinkedBinaryTree

```java
public Iterator<T> inorder()
{
    ArrayIterator<T> iter = new ArrayIterator<T>();
    if (root != null)
        root.inorder(iter);
    return iter;
}
```
This is AN iterator, not THE iterator

```java
public Iterator<T> levelorder()
{
    LinkedQueue<BTNode<T>> queue = new LinkedQueue<BTNode<T>>();
    ArrayIterator<T> iter = new ArrayIterator<T>();
    if (root != null)
    {
        queue.enqueue(root);
        while (!queue.isEmpty())
        {
            BTNode<T> current = queue.dequeue();
            iter.add (current.getElement());
            if (current.getLeft() != null)
                queue.enqueue(current.getLeft());
            if (current.getRight() != null)
                queue.enqueue(current.getRight());
        }
    }
    return iter;
}
```

You also need the iterator() method!

```java
public Iterator<T> iterator()
{
    return inorder();
}
```

### IMPORTANT
To test this file you need to create a driver OUTSIDE javafoundations. You cannot do it by adding a main() method in LinkedBinaryTree.java

Is this clear?

---

16.5 – Decision Trees

- A decision tree is a tree whose nodes represent decision points, and whose children represent the options available.
- The leaves of a decision tree represent the possible conclusions that might be drawn based on the answers.
- Decision trees are sometimes used in expert systems – software that attempts to represent the knowledge of an expert in a particular field.
- A simple decision tree, with yes/no questions, can be modeled by a binary tree.
- Expertise examples:
  - a doctor
  - a car mechanic
  - accountant
  - PC help desk??

"Pause for a moment so you can let this information sink in."
The left child represents the answer "No"
The right child represents the answer "Yes"