```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Stack;

/**
 * This class implements a binary tree by using an array.
 *
 * @author Wesley Febrian - CS 111C, Instructor: Jessica Masters
 * @version 2.0
 */
public class ArrayBinaryTree<T> implements BinaryTreeInterface<T>,
        java.io.Serializable {
    private T theData[];
    private int height; // height of tree
    private int size; // number of locations in array for a full tree of this
                      // height

    public ArrayBinaryTree() {
        // IMPLEMENT DEFAULT CONSTRUCTOR
        // SUGGESTION: DIRECTLY INITIALIZE THE THREE INSTANCE DATA VARIABLES
        theData = (T[]) new Object[0];
        height = 0;
        size = 0;
    }

    public ArrayBinaryTree(T rootData) {
        // IMPLEMENT THE CONSTRUCTOR FOR A ONE-NODE TREE
        // SUGGESTION: DIRECTLY INITIALIZE THE THREE INSTANCE DATA VARIABLES
        setTree(rootData);
    }

    public ArrayBinaryTree(T rootData, ArrayBinaryTree<T> leftTree,
            ArrayBinaryTree<T> rightTree) {
        // IMPLEMENT THE CONSTRUCTOR THAT TAKES A NEW ROOT AND LEFT AND RIGHT
        // SUBTREE
        // SUGGESTION: INVOKE THE PRIVATE SETTREE METHODS BELOW
        privateSetTree(rootData, (ArrayBinaryTree<T>) leftTree, (ArrayBinaryTree<T>) rightTree);
    }

    public void setTree(T rootData) {
        // SET THE TREE TO BE A NEW ONE-NODE TREE
        // SUGGESTION: INVOKE THE PRIVATE SETTREE METHOD BELOW
        theData = (T[]) new Object [1];
        height = 1;
        size = 1;
        theData[0] = rootData;
    }

    public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
            BinaryTreeInterface<T> rightTree) {
        // SET THE TREE TO BE A NEW TREE WITH THE SPECIFIED ROOT AND LEFT AND
        // RIGHT SUBTREES
        // SUGGESTION: INVOKE THE PRIVATE SETTREE METHOD BELOW
        privateSetTree(rootData, (ArrayBinaryTree<T>) leftTree, (ArrayBinaryTree<T>) rightTree);
    }

    /*
     * a helper method that can be used to set up a tree from existing subtrees
     */
    private void privateSetTree(T rootData, ArrayBinaryTree<T> leftTree,
            ArrayBinaryTree<T> rightTree) {

        // SUGGESTION:  DETERMINE WHAT THE NEW HEIGHT AND SIZE OF THE TREE SHOULD BE
        //              INITIALIZE THE ARRAY AND THE ROOT
        //              INVOKE THE PRIVATE SETLEFT/SETRIGHT METHODS BELOW

        if(leftTree == null && rightTree == null){
            setTree(rootData);
        }

        //(rootData, !=null, !=null)
        if(leftTree != null && rightTree != null){
            ArrayBinaryTree<T> lTree = (ArrayBinaryTree<T>) leftTree;
            ArrayBinaryTree<T> rTree = (ArrayBinaryTree<T>) rightTree;
```

```java
        height = Math.max(lTree.getHeight(), rTree.getHeight()) + 1;

        theData = (T[]) new Object[getSizeFromHeight(height)];

        setRootData(rootData);
        size = getSizeFromHeight(height);

        setLeftSubtree((ArrayBinaryTree<T>) leftTree);
        setRightSubtree((ArrayBinaryTree<T>) rightTree);
    }

    //(rootData, null, !=null)
    if(leftTree == null && rightTree != null){
        ArrayBinaryTree<T> rTree = (ArrayBinaryTree<T>) rightTree;

        height = rTree.getHeight() + 1;

        theData = (T[]) new Object[getSizeFromHeight(height)];

        setRootData(rootData);
        size = getSizeFromHeight(height);

        setRightSubtree((ArrayBinaryTree<T>) rightTree);
    }

    //(rootData, !null, null)
    if(leftTree != null && rightTree == null){
        ArrayBinaryTree<T> lTree = (ArrayBinaryTree<T>) leftTree;

        height = lTree.getHeight() + 1;

        theData = (T[]) new Object[getSizeFromHeight(height)];

        setRootData(rootData);
        size = getSizeFromHeight(height);

        setLeftSubtree((ArrayBinaryTree<T>) leftTree);
    }
}

/*
 * Copies the data values from the given subtree into the leftsubtree.
 * Precondition: The array theData is large enough to hold the new values.
 */
private void setLeftSubtree(ArrayBinaryTree<T> subTree) {
    // THIS IS THE PLACE WHERE YOU NOW DIRECTLY ACCESS THE DATA IN theData ARRAY
    // COPY THE DATA FROM THE SUBTREE ARRAY INTO theData ARRAY
    // I RECOMMEND TRACING OUT ON PAPER HOW THE INDICES OF THE SUBTREE ARRAY MAP TO
    //      THE INDICES IN theData

    int subTreeIndex = 0;
    int nodesInRow = 1;
    int firstIndex = 0;
    int lastIndex = firstIndex + nodesInRow - 1;

    for (int i = 1; i <= subTree.height; i++) {
        firstIndex = 2 * firstIndex + 1;
        lastIndex = firstIndex + nodesInRow - 1;
        int currentIndex = firstIndex;

        for (int j = firstIndex; j <= lastIndex; j++) {
            theData[currentIndex] = subTree.theData[subTreeIndex];
            currentIndex++;
            subTreeIndex++;
        }

        nodesInRow = 2 * nodesInRow;
    }
}

/*
 * Copies the data values from the given subtree into the rightsubtree.
 * Precondition: The array theData is large enough to hold the new values.
 */
private void setRightSubtree(ArrayBinaryTree<T> subTree) {
```

```java
        // THIS IS THE PLACE WHERE YOU NOW DIRECTLY ACCESS THE DATA IN theData ARRAY
        // COPY THE DATA FROM THE SUBTREE ARRAY INTO theData ARRAY
        // I RECOMMEND TRACING OUT ON PAPER HOW THE INDICES OF THE SUBTREE ARRAY MAP TO
        // THE INDICES IN theData
        int subTreeIndex = 0;
        int nodesInRow = 1;
        int firstIndex = 0;
        int lastIndex = 0;

        for (int i = 1; i <= subTree.height; i++) {
            lastIndex = 2 * lastIndex + 2;
            firstIndex = lastIndex - nodesInRow + 1;
            int currentIndex = firstIndex;

            for (int j = firstIndex; j <= lastIndex; j++) {
                theData[currentIndex] = subTree.theData[subTreeIndex];
                currentIndex++;
                subTreeIndex++;
            }

            nodesInRow = 2 * nodesInRow;
        }
    }

    /*
     * Finds the size of the array necessary to fit a tree of height h.
     */
    private int getSizeFromHeight(int h) {
        // YOU MIGHT FIND THIS METHOD HELPFUL
        // IT CALCULATES THE SIZE OF THE ARRAY NEEDED TO ACCOMODATE A TREE OF
        // HEIGHT H
        //maxSize is 2^h-1
        return (int) Math.round(Math.pow(2.0, (double) h) - 1.0);
    }

    public T getRootData() {
        // RETURNS THE ROOT OF THE TREE
        // BE SURE TO ACCOUNT FOR EMPTY TREES
        if(isEmpty()){
            return null;
        }else{
            return theData[0];
        }
    }

    public boolean isEmpty() {
        // RETURNS TRUE IF THE TREE IS EMPTY
        return (height == 0 && size == 0);
    }

    public void clear() {
        // EMPTIES THE TREE
        for (int i = 0; i < theData.length; i++){
                theData[i] = null;
            }
        height = 0;
        size = 0;
    }

    protected void setRootData(T rootData) {
        // SETS THE ROOT OF THE TREE TO A NEW VALUE
        theData[0] = rootData;
    }

    public int getHeight() {
        // GETS THE HEIGHT OF THE TREE

        // NOTE: IF YOU ARE KEEPING TRACK OF THE HEIGHT OF THE TREE EVERYTIME YOU SET
        // THE TREE, THEN THIS IS JUST A REGULAR GETTER. ANOTHER OPTION IS JUST TO
        // FIND THE HEIGHT WHEN THIS METHOD IS INVOKED BASED ON THE ACTUAL DATA IN
        // THE TREE AT THAT MOMENT. IF YOU DO THIS, I SUGGEST ADDING A PRIVATE METHOD THAT
        // HELPS FIND THE HEIGHT USING RECURSION

        return height;
    }
```

```java
public int getNumberOfNodes() {
    // RETURNS THE NUMBER OF NODES IN THE TREE
    // REMEMBER THAT NOT ALL SPOTS OF THE ARRAY WILL NECESSARILY BE FILLED
    int counter = 0;

    for(int i = 0; i < theData.length; i++){
        if(theData[i] != null){
            counter++;
        }
    }
    return counter;

}

/*
 * The following operations allow one to move in the tree and test to see
 * whether a child exists. These methods have already been implemented.
 */
private boolean hasLeftChild(int i) {
    return nodeExists((2 * i + 1));
}

private int leftChild(int i) {
    return 2 * i + 1;
}

private boolean hasRightChild(int i) {
    return nodeExists((2 * i + 2));
}

private int rightChild(int i) {
    return 2 * i + 2;
}

private boolean nodeExists(int i) {
    return (i >= 0 && i < size) && (theData[i] != null);
}

private int parent(int i) {
    return (i - 1) / 2;
}
private T getData(int i) {
    T result = null;

    if (nodeExists(i))
        result = theData[i];
    return result;
}
/* display the contents of the array */
public void display() {
    for (int i = 0; i < size; i++) {
        if (nodeExists(i))
            System.out.println("index: " + i + " has " + getData(i));
    }
}

public Iterator<T> getInorderIterator() {
    return new InorderIterator();
}

private class InorderIterator implements Iterator<T> {
    private Stack<Integer> nodeStack;
    private Integer currentNode;

    public InorderIterator() {
        nodeStack = new Stack<Integer>();
        currentNode = 0;
    }

    public boolean hasNext() {
        return !nodeStack.isEmpty() || nodeExists(currentNode);
    }

    public T next() {
```

```
        Integer nextNode = -1;

        // find leftmost node with no left child
        while (nodeExists(currentNode)) {
            nodeStack.push(currentNode);
            currentNode = leftChild(currentNode);
        }

        // get leftmost node, then move to its right subtree
        if (!nodeStack.isEmpty()) {
            nextNode = nodeStack.pop();
            assert nodeExists(nextNode); // since nodeStack was not empty
                                         // before the pop
            currentNode = rightChild(nextNode); // right subchild
        } else
            throw new NoSuchElementException();

        return theData[nextNode];
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
} // end InorderIterator

public Iterator<T> getPreorderIterator() {
    return new PreorderIterator();
}

private class PreorderIterator implements Iterator<T> {
    // EXTRA CREDIT
    // IMPLEMENT THE PREORDER ITERATOR

    private Stack<Integer> nodeStack;

    public PreorderIterator() {
        nodeStack = new Stack<Integer>();
        if(!isEmpty()){
            nodeStack.push(0);
        }
    }

    public boolean hasNext() {
        return !nodeStack.isEmpty();
    }

    public T next() {
        T result = null;
        if (nodeStack.isEmpty()) {
            throw new NoSuchElementException();
        } else {
            Integer top = nodeStack.pop();
            result = theData[top];

            // Push the children on the stack. Right then left.
            if (hasRightChild(top)) // has right child
                nodeStack.push(rightChild(top));
            if (hasLeftChild(top)) // has left child
                nodeStack.push(leftChild(top));
        }

        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
} // end PreorderIterator

public Iterator<T> getPostorderIterator() {
    return new PostorderIterator();
}

private class PostorderIterator implements Iterator<T> {
    private Stack<PostOrderNode> nodeStack;
```

```java
        public PostorderIterator() {
            nodeStack = new Stack<PostOrderNode>();
            if (!isEmpty())
                nodeStack.push(new PostOrderNode(0, PostOrderState.LEFT));
        }

        public boolean hasNext() {
            return !nodeStack.isEmpty();
        }

        public T next() {
            T result = null;
            if (nodeStack.isEmpty()) {
                throw new NoSuchElementException();
            } else {
                PostOrderNode top = nodeStack.pop();
                PostOrderState state = top.state;

                while (state != PostOrderState.TOP) {
                    if (state == PostOrderState.LEFT) {
                        top.state = PostOrderState.RIGHT;
                        nodeStack.push(top);

                        if (hasLeftChild(top.node)) // hasLeftChild
                            nodeStack.push(new PostOrderNode(
                                    leftChild(top.node), PostOrderState.LEFT));
                    } else {
                        assert state == PostOrderState.RIGHT;
                        top.state = PostOrderState.TOP;
                        nodeStack.push(top);

                        if (hasRightChild(top.node)) // hasRightChild
                            nodeStack.push(new PostOrderNode(
                                    rightChild(top.node), PostOrderState.LEFT));
                    }
                    top = nodeStack.pop();
                    state = top.state;
                }
                result = theData[top.node];
            }

            return result;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    } // end PostorderIterator

    private enum PostOrderState {
        TOP, LEFT, RIGHT
    };

    private class PostOrderNode {
        public Integer node;
        public PostOrderState state;

        PostOrderNode(Integer theNode, PostOrderState theState) {
            node = theNode;
            state = theState;
        }
    }

    public Iterator<T> getLevelOrderIterator() {
        throw new UnsupportedOperationException();
    }

}
```