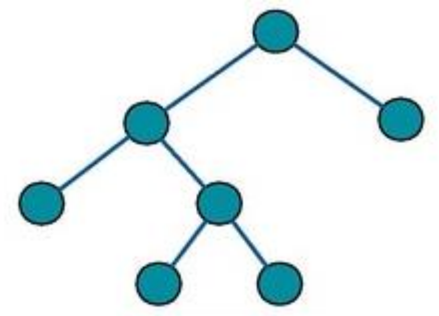
Forest Play Ground - Binary Trees

This problem is worth 15 points.

Please understand, according to Wikipedia, Tree terminology is not well-standardized and varies in the literature. You should read the descriptions carefully in this problem to ensure that you are solving the given problem.

- A rooted binary tree has a root node and every node has at most two children.
- A full binary tree (sometimes referred to as a proper or plane binary tree) is a tree in which every node in the tree has either 0 or 2 children.
- In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes at the last level h. A complete binary tree can be efficiently represented using an array.


A full binary tree

Methods for storing binary trees

Arrays

Binary trees can also be stored in breadth-first order as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, Assuming the root has index zero, if a node has an index i, its children are found at indices $2i+1$ (for the left child) and $2i+2$ (for the right), while its parent (if any) is found at index $\left\lfloor \frac{i-1}{2} \right\rfloor$. (In this case, $\lfloor x \rfloor$ is the largest (Closes to positive infinity) Integer value smaller than or equal to x. For example, $\lfloor 2.9 \rfloor = 2$.

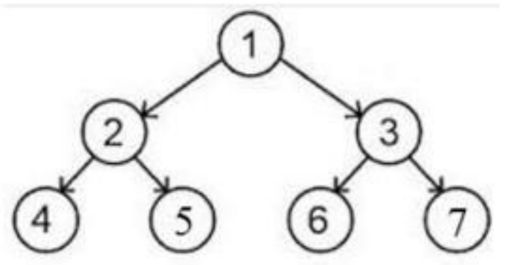


Figure 1

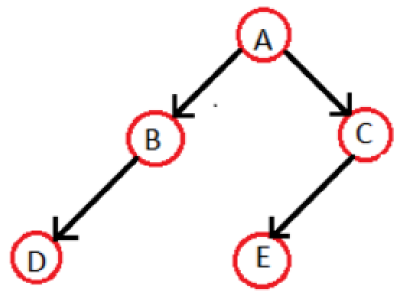


Figure 2

For example, the rooted binary tree in Figure 1 and Figure 2 would be represented by the flowing lines of code:

```
String[] figure1 = {"1", "2", "3" ,"4", "5", "6", "7"};

String[] figure2 = {"A", "B", "C" ,"D", null, "E", null};
```

Note: figure 1 is full and complete while figure 2 is no full and not complete.

In this problem you are to complete the `ForestPlayGround` class which implements the functionality of a rooted binary tree. You may implement this class in any manner, but the rooted binary tree and the algorithms described in this problem assume an array as the under lining data structure used to store the rooted binary tree. The nine methods are:

The `getNumNodes()` method returns the number of non null nodes in the rooted binary tree.

The following table show sample results of the `getNumNodes` method.

The following code	Returns

`String[] figure1 = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(figure1);`
 	
`t.getNumNodes()`
	8

`String[] figure2 = {"A", "B", "C", "D", null, "E"};`
`ForestPlayGround t2 = new ForestPlayGround(figure2);`
 	
`T2.getNumNodes()`
	5

The `getNumLeafs()` returns the number of leaf nodes in the rooted binary tree. A node is a leaf node if the node has no children. That is, the node at index i is a leaf node if the nodes at both index $2i+1$ and index $2i+2$ do not exist or are null.

The following table show sample results of the `getNumLeafs` method.

The following code	Returns

`String[]figure1 = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(figure1);`
 	
`t. getNumLeafs();`
	4

`String[] figure2 = {"A", "B", "C", "D", null, "E"};`
`ForestPlayGround t2 = new ForestPlayGround(figure2);`
 	
`T2.getNumLeafs()`
	2

The `getLeftChild(int p)` method returns the value contained in the node that is the left child of the node at index p. The left child of the node at index p is the node at index $2p+1$. If no node exists, return null.

The following tables show sample results of the `getLeftChild` method.
You may assume the node at index p is not `null`.

The following code	Returns

`String[] tree = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(tree);`
 	
`t. getLeftChild(4);`
	`null`

The `getRightChild(int p)` method returns the value contained in the node that is the right child of the node at index p. The right child of the node at index p is the node at index $2p+2$. If no node exists, return null.

The following table show sample results of the `getRightChild` method.
You may assume the node at index p is not `null`.

The following code	Returns

`String[] tree = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(tree);`
 	
`t. getRightChild(2);`
	“6”

The `getParent(int p)` method returns the value contain in the node that is the parent of the node at index p. Use the information given in `getLeftChild` and `getRightChild` methods to determine the index of the parent node.

The following table show sample results of the `getParent` method.
You may assume the node at index *p* is not `null`.

The following code	Returns

`String[] tree = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(tree);`
 	
`t.getParent(5);`
	"2"

The `getAncestors(int p)` method returns a `List<String>` of the values contained all of the nodes that are ancestors of the node at index p. Node A is an ancestor of node B if A is a parent of B, or if some child of A is an ancestor of B. In less formal terms, A is an ancestor of B if B is a child of A, or a child of a child of A, or a child of a child of a child of A, etc.

The following table show sample results of the `getAncestors` method.
You may assume the node at index *p* is not `null`.

The following code	Returns

`String[] tree = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(tree);`
 	
`List<String> ancestors = t.getAncestors(6);`	

`ancestors.size()`	2
`ancestors.contains("2")`	`true`
`ancestors.contains("0")`	`true`

The `getDescendants(int p)` method returns a `List<String>` of the values contained all of the nodes that are descendants of the node at index p. Node B is a descendant of A if A is an ancestor of B. If the node at index p has no descendant, return an empty list.

The following table show sample results of the `getDescendants` method.
You may assume the node at index *p* is not `null`.

The following code	Returns

`String[] tree = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`ForestPlayGround t = new ForestPlayGround(tree);`
 	
`List<String> descendants = t.getDescendants(1);`	

`descendants.size()`	3
`descendants.contains("3")`	`true`
`descendants.contains("4")`	`true`
`descendants.contains("7")`	`true`

The `isFull()` method returns true if the `String[]` represents a Full binary tree. Recall that a full binary tree is a binary tree in which every node in the tree has either 0 or 2 children. If the tree is empty, return true.

The following table show sample results of the `isFull` method.

The following code	Returns
`String[] tree = {"0", "1", "2", "3", "4", "5", "6", "7"};` `ForestPlayGround t = new ForestPlayGround(tree);`	
`t.isFull()`	`false`
`String[] emptyTree = {};` `ForestPlayGround emptyT = new ForestPlayGround(emptyTree);`	
`emptyT.isFull()`	`true`

The `isComplete()` method returns true if the `String[]` represents a Complete binary tree. Recall, in a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. If the tree is empty, return true.

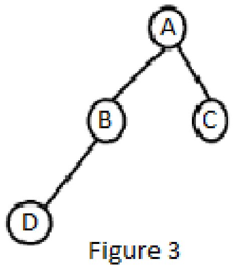


Figure 3

For example, the rooted binary tree in Figure 3 represented by the flowing code is a complete binary tree.

- `String[] figure3 = {“A”, “B”, “C” ,”D” };`

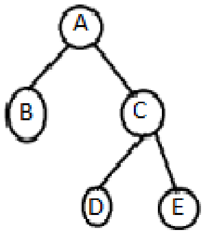


Figure 4

Again, the rooted binary tree in Figure 4 represented by the flowing code is a NOT complete binary tree.

- `String[] figure4 = {“A”, “B”, “C”, null, null, ”D” , “E”};`

The following table show sample results of the `isComplete` method.

The following code	Returns
`String[] figure3 = {"A", "B", "C", "D"};` `t = new ForestPlayGround(figure3);`	
`t.isComplete()`	`true`
`String[] figure4 = {"A", "B", "C", null, null, "D", "E"};` `t = new ForestPlayGround(figure4);`	
`t.isComplete()`	`false`
`String[] emptyTree = {};` `ForestPlayGround emptyT = new ForestPlayGround(emptyTree);`	
`emptyT.isComplete()`	`true`

The `getLowestCommonAncestor(String child1, String child2)` the lowest common ancestor (LCA) of two nodes `v` and `w` in a tree T is the lowest node that has both `v` and `w` as descendants, where we define each node to be a descendant of itself (so if v has a direct connection from w, w is the lowest common ancestor).

The LCA of `v` and `w` in T is the shared ancestor of `v` and `w` that is located farthest from the root (In this case, the node with the largest index in the array).

The following table show sample results of the `getLowestCommonAncestor` method.
You may assume `child1` and `child2` are valid nodes in the tree, `getLowestCommonAncestor` will always return a node in the tree.

The following code	Returns

`String[] figure3 = {"0", "1", "2", "3", "4", "5", "6", "7"};`
`t = new ForestPlayGround(figure3);`	

`t.getLowestCommonAncestor("5", "6")`	`"2"`
`t.getLowestCommonAncestor("5", "2")`	`"2"`
`t.getLowestCommonAncestor("7", "2")`	`"0"`

Extra points are awarded for completing:

- `getLeftChild` and `gerRightChild` methods
- `getAncestor` and `getDescendants` methods
- `isFull` and `isComplete` methods
- `getAncestor, getDescendants, isFull` and `isComplete` methods
- all methods in the `ForestPlayGround` class.