Forest Play Ground - Binary Trees

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.

This problem will require you to implement several methods on a rooted binary tree. For this problem, a rooted binary tree is a data structure that has a root node and every node has at most two children.

Methods for storing binary trees

Arrays

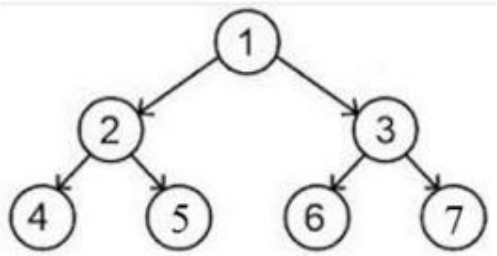


Figure 1

Binary trees can also be stored in breadth-first order as an implicit data structure in arrays. 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: $\lfloor 2.9 \rfloor = 2$. Example of the rooted binary tree in Figure 1 and Figure 2 stored as arrays is represented by the flowing lines of code:

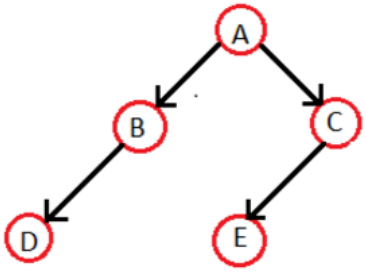


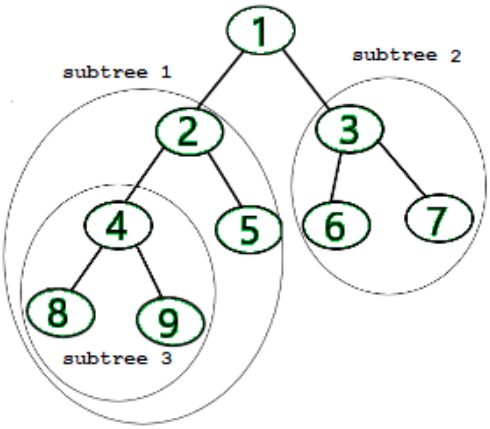
Figure 2

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

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

In figure 2, the value `null` is used to represent the absence of a node.

Consider the following definitions: In a binary tree, a child node is a node that has a parent node and is connected to it by a branch. Each node in a binary tree has at most two child nodes, which are called the left child and the right child. In Figure 1, 2 is the left child of the root node 1, and 3 is the right child of the root node 1. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. (A descendant is a child of a node, or a child of a child of …). The left subtree of a node in a binary tree consists of a node's left child and all of this node's descendants.
Consider the diagram to the right. The left subtree of the root (top node containing a 1) is circled and label as subtree 1. The right subtree of the root is circled and labeled as subtree 2. The left subtree of the node containing the value 2 is circled and labeled as subtree 3. In addition, the right subtree of the node containing a 4 is the single node 9.



Note: All valid values in this problem will be non-negative (greater than or equal to 0), and all negative integers represent `null` (the absence of a node).

Note: this is question 4 from the 1996 AP CS AB test, the first question I ever graded.

In this problem you are to complete five methods in 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 five methods are:

- `int[] getValuesInLeftSubtree(int p)`
- `int[] getValuesInRightSubtree(int p)`
- `boolean valsLess(int p)`
- `boolean valsGreater(int p)`
- `boolean isBST()`

The `getValuesInLeftSubtree(int p)` method returns an `int[]` containing all values (values, not indexes), in any order, in the left subtree of the node at index `p`. The number of elements in the left subtree equals the length of the `int[]` being returned.

Recall negative values in the tree represent the absence of a node and should NOT be included.

The following table shows the results of the `getValuesInLeftSubtree` method called on the root of the given Tree.

The following code	Returns

`int[] tree1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`
`ForestPlayGround t = new ForestPlayGround(tree1);`
 	
`int[] result = t.getValuesInLeftSubtree(0);`
 	
`Arrays.sort(result);  // I'll sort the values for you ☺`	

`result.length`	7
`result[0]`	1
`result[1]`	3
`result[2]`	4
`result[3]`	7
`result[4]`	8
`result[5]`	9
`result[6]`	10

The following table shows the results of the `getValuesInLeftSubtree` method called on a node that is NOT the root of the Tree.

The following code	Returns

`int[] tree1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`
`ForestPlayGround t = new ForestPlayGround(tree1);`
 	
`int[] result = t.getValuesInLeftSubtree(2);`
`Arrays.sort(result); // I'll sort the values for you ☺`	

`result.length`	3
`result[0]`	5
`result[1]`	11
`result[2]`	12

The following table shows the results of the `getValuesInLeftSubtree` method called on the root of the Tree with 'missing' values.

The following code	Returns

`int[] tree1 = { 10, 5, -3, 4, 7, -1, -1, 2};`
`ForestPlayGround t = new ForestPlayGround(tree1);`
 	
`int[] result = t.getValuesInLeftSubtree(2);`
`Arrays.sort(result); // I'll sort the values for you ☺`	

`result.length`	4
`result[0]`	2
`result[1]`	4
`result[2]`	5
`result[3]`	7

The `getValuesInRightSubtree(int p)` method returns an `int[]` containing all values (values, not indexes), in any order, in the right subtree of the node at index `p`. The number of elements in the right subtree equals the length of the `int[]` being returned.

Recall negative values in the tree represent the absence of a node and should NOT be included.

The following table shows the results of the `getValuesInRightSubtree` method called on the root of the given Tree.

`int[] tree1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`
`ForestPlayGround t = new ForestPlayGround(tree1);`
 	
`result = t.getValuesInRightSubtree(0);`
`Arrays.sort(result);  // I'll sort the values for you ☺`	

`result.length`	5
`result[0]`	2
`result[1]`	5
`result[2]`	6
`result[3]`	11
`result[4]`	12

The following table shows the results of the `getValuesInRightSubtree` method called on a node that is NOT the root of the Tree.

The following code	Returns

`int[] tree1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`
`ForestPlayGround t = new ForestPlayGround(tree1);`
 	
`int[] result = t.getValuesInRightSubtree(2);`
`Arrays.sort(result); // I'll sort the values for you ☺`	

`result.length`	1
`result[0]`	6

The following table shows the results of the `getValuesInRightSubtree` method called on the root of the Tree with 'missing' values.

The following code	Returns

`int[] tree1 = { 10, 5, -3, 4, 7, -1, -1, 2};`
`ForestPlayGround t = new ForestPlayGround(tree1);`
 	
`int[] result = t.getValuesInLeftSubtree(2);`
`Arrays.sort(result); // I'll sort the values for you ☺`	

`result.length`	`0`

The `valsLess(int p)` method returns `true` only if all values in the left subtree of node at index `p` are less than `myTree[p]`, otherwise, return `false`. If the left subtree is empty, return `true`.

The following table shows the results of the `valsLess` method called on the root of the Tree.

The following code	Returns

`int[] figure2 = {50, 22, 70, 15, 32, 66, 90,`
`5, 20, 33, 14, 55, 63, 68, 95};`
`ForestPlayGround t2 = new ForestPlayGround(figure2);`	

`t2.valsLess(1)`	`true`
`t2.valsLess(2)`	`true`
`t2.valsLess(10)`	`true`
`t2.valsLess(0)`	`false`
`t2.valsLess(4)`	`false`

The `valsGreater(int p)` method returns `true` only if all values in the right subtree of node at index `p` are greater than `myTree[p]`, otherwise, return `false`. If the right subtree is empty, return `true`.

The following table shows the results of the `valsGreater` method called on the root of the Tree.

The following code	Returns

`int[] figure2 = {50, 22, 70, 15, 32, 66, 90,`
`5, 20, 33, 14, 55, 63, 68, 95};`
`ForestPlayGround t2 = new ForestPlayGround(figure2);`	

`t2.valsGreater(0)`	`true`
`t2.valsGreater(1)`	`false`
`t2.valsGreater(8)`	`true`
`t2.valsGreater(5)`	`false`
`t2.valsGreater(4)`	`false`

The `isBst()` method returns `true` if the binary tree is a ***binary search tree*** that contains no duplicate values; otherwise `isBST` should return `false`.

Note that a binary tree T is a ***binary search tree*** that contains no duplicate values if and only if

1. T is empty

   or

2. all of the following are true
   - All values stored in the left subtree of the root of T are less than the value stored at the root of T.
   - All values stored in the right subtree of the root of T are greater than the value stored at the root of T.
   - T's left subtree is a binary search tree that contains no duplicate values.
   - T's right subtree is a binary search tree that contains no duplicate values.

The following table shows the results of the `isBST` method called on the root of the Tree.

The following code	Returns

`int[] figure2 = {50, 22, 70, 15, 32, 66, 90, 5, 20, 33, 14, 55, 63, 68, 95};`
 	
`ForestPlayGround t2 = new ForestPlayGround(figure2);`	

`t2.isBST()`	`false`

In the above example, `T2` is not a ***binary search tree*** that contains no duplicate values since the following are all `false`:

- `t2.valsLess(4)`
- `t2.valsGreater(1)`

The following table shows the results of the `isBST` method called on the root of a different Tree.

`int[]figure3 = {50, 25, 75, 10, 35, 60, 90, 1, 20, 27, 40, 55};`
 	
`ForestPlayGround t3 = new ForestPlayGround(figure3);`	

`t3.isBST()`	`true`