1. **import** java.util.ArrayList;
3. **public** **class** MaxDistance {
5. //Represent a node of binary tree
6. **public** **static** **class** Node{
7. **int** data;
8. Node left;
9. Node right;
11. **public** Node(**int** data){
12. //Assign data to the new node, set left and right children to null
13. **this**.data = data;
14. **this**.left = **null**;
15. **this**.right = **null**;
16. }
17. }
19. //Represent the root of binary tree
20. **public** Node root;
22. **int**[] treeArray;
23. **int** index = 0;
25. **public** MaxDistance(){
26. root = **null**;
27. }
29. //calculateSize() will calculate size of tree
30. **public** **int** calculateSize(Node node)
31. {
32. **int** size = 0;
33. **if** (node == **null**)
34. **return** 0;
35. **else** {
36. size = calculateSize (node.left) + calculateSize (node.right) + 1;
37. **return** size;
38. }
39. }
41. //convertBTtoArray() will convert binary tree to its array representation
42. **public** **void** convertBTtoArray(Node node) {
43. //Check whether tree is empty
44. **if**(root == **null**){
45. System.out.println("Tree is empty");
46. **return**;
47. }
48. **else** {
49. **if**(node.left != **null**)
50. convertBTtoArray(node.left);
51. //Adds nodes of binary tree to treeArray
52. treeArray[index] = node.data;
53. index++;
54. **if**(node.right != **null**)
55. convertBTtoArray(node.right);
56. }
57. }
59. //getDistance() will find distance between root and a specific node
60. **public** **int** getDistance(Node temp, **int** n1) {
61. **if** (temp != **null**) {
62. **int** x = 0;
63. **if** ((temp.data == n1) || (x = getDistance(temp.left, n1)) > 0
64. || (x = getDistance(temp.right, n1)) > 0) {
65. //x will store the count of number of edges between temp and node n1
66. **return** x + 1;
67. }
68. **return** 0;
69. }
70. **return** 0;
71. }
73. //lowestCommonAncestor() will find out the lowest common ancestor for nodes node1 and node2
74. **public** Node lowestCommonAncestor(Node temp, **int** node1, **int** node2) {
75. **if** (temp != **null**) {
76. //If root is equal to either of node node1 or node2, return root
77. **if** (temp.data == node1 || temp.data == node2) {
78. **return** temp;
79. }
81. //Traverse through left and right subtree
82. Node left = lowestCommonAncestor(temp.left, node1, node2);
83. Node right = lowestCommonAncestor(temp.right, node1, node2);
85. //If node temp has one node(node1 or node2) as left child and one node(node1 or node2) as right child
86. //Then, return node temp  as lowest common ancestor
87. **if** (left != **null** && right != **null**) {
88. **return** temp;
89. }
91. //If nodes node1 and node2 are in left subtree
92. **if** (left != **null**) {
93. **return** left;
94. }
95. //If nodes node1 and node2 are in right subtree
96. **if** (right != **null**) {
97. **return** right;
98. }
99. }
100. **return** **null**;
101. }
103. //findDistance() will find distance between two given nodes
104. **public** **int** findDistance(**int** node1, **int** node2) {
105. //Calculates distance of first node from root
106. **int** d1 = getDistance(root, node1) - 1;
107. //Calculates distance of second node from root
108. **int** d2 = getDistance(root, node2) - 1;
110. //Calculates lowest common ancestor of both the nodes
111. Node ancestor = lowestCommonAncestor(root, node1, node2);
113. //If lowest common ancestor is other than root then, subtract 2 \* (distance of root to ancestor)
114. **int** d3 = getDistance(root, ancestor.data) - 1;
115. **return** (d1 + d2) - 2 \* d3;
116. }
118. //nodesAtMaxDistance() will display the nodes which are at maximum distance
119. **public** **void** nodesAtMaxDistance(Node node) {
120. **int** maxDistance = 0, distance = 0;
121. ArrayList<Integer> arr = **new** ArrayList<>();
123. //Initialize treeArray
124. **int** treeSize = calculateSize(node);
125. treeArray = **new** **int**[treeSize];
127. //Convert binary tree to its array representation
128. convertBTtoArray(node);
130. //Calculates distance between all the nodes present in binary tree and stores maximum distance in variable maxDistance
131. **for**(**int** i = 0; i < treeArray.length; i++) {
132. **for**(**int** j = i; j < treeArray.length; j++) {
133. distance = findDistance(treeArray[i], treeArray[j]);
134. //If distance is greater than maxDistance then, maxDistance will hold the value of distance
135. **if**(distance > maxDistance) {
136. maxDistance = distance;
137. arr.clear();
138. //Add nodes at position i and j to treeArray
139. arr.add(treeArray[i]);
140. arr.add(treeArray[j]);
141. }
142. //If more than one pair of nodes are at maxDistance then, add all pairs to treeArray
143. **else** **if**(distance == maxDistance) {
144. arr.add(treeArray[i]);
145. arr.add(treeArray[j]);
146. }
147. }
148. }
149. //Display all pair of nodes which are at maximum distance
150. System.out.println("Nodes which are at maximum distance: ");
151. **for**(**int** i = 0; i < arr.size(); i = i + 2) {
152. System.out.println("( " + arr.get(i) + "," + arr.get(i+1) + " )");
153. }
154. }
156. **public** **static** **void** main(String[] args) {
158. MaxDistance bt = **new** MaxDistance();
159. //Add nodes to the binary tree
160. bt.root = **new** Node(1);
161. bt.root.left = **new** Node(2);
162. bt.root.right = **new** Node(3);
163. bt.root.left.left = **new** Node(4);
164. bt.root.left.right = **new** Node(5);
165. bt.root.right.left = **new** Node(6);
166. bt.root.right.right = **new** Node(7);
167. bt.root.right.right.right = **new** Node(8);
168. bt.root.right.right.right.left = **new** Node(9);
170. //Finds out all the pair of nodes which are at maximum distance
171. bt.nodesAtMaxDistance(bt.root);
172. }
173. }

