```java
package com.bo.structure;

import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class BinaryTree {

    class Node{
        int value;
        Node left;
        Node right;
        Node(int v){
            this.value = v;
            left = null;
            right = null;
        }
    }

    private Node root;

    public BinaryTree(int[] arr){
        for(int i:arr){
            insert(i);
        }
    }

    public void insert(int i){
        root = insert(root, i);
    }

    public Node insert(Node node, int i){
        if(node == null){
            node = new Node(i);
        }else if(i <= node.value){
            node.left = insert(node.left, i);
        }else{
            node.right = insert(node.right, i);
        }

        return node;
    }

    public void visit(Node p){
        System.out.println(p.value);
    }

    public void pre(Node p){
        if(p != null){
            visit(p);
            pre(p.left);
            pre(p.right);
```

```
53.             }
54.         }
55.
56.         public void ppp(){
57.             pre(root);
58.         }
59.
60.         public void preOrder(Node p){
61.             Stack<Node> stack = new Stack<Node>();
62.             if(p != null){
63.                 stack.push(p);
64.                 while(!stack.isEmpty()){
65.                     Node t = stack.pop();
66.                     visit(t);
67.                     if(p.right != null){
68.                         stack.push(p.right);
69.                     }
70.                     if(p.left != null){
71.                         stack.push(p.left);
72.                     }
73.                 }
74.             }
75.         }
76.
77.         //first visit then push into stack
78.         //much more easier to understand
79.         public void preOrder2(Node p){
80.             Stack<Node> stack = new Stack<Node>();
81.             Node node = p;
82.             while(node != null || !stack.isEmpty()){
83.                 while(node != null){
84.                     visit(node);
85.                     stack.push(node);
86.                     node = node.left;
87.                 }
88.                 if(!stack.isEmpty()){
89.                     node = stack.pop();
90.                     node = node.right;
91.                 }
92.             }
93.
94.         }
95.
96.         public void inOrder(Node p){
97.             Stack<Node> stack = new Stack<Node>();
98.             while(p != null){
99.                 while(p != null){
100.                    if(p.right != null)
101.                        stack.push(p.right);
102.                    stack.push(p);
103.                    p = p.left;
104.                }
105.                p = stack.pop();
106.                while(!stack.isEmpty() && p.right == null){
```

```
107.                 visit(p);
108.                 p = stack.pop();
109.             }
110.             visit(p);
111.             if(!stack.isEmpty())
112.                 p = stack.pop();
113.             else
114.                 p = null;
115. 
116.         }
117.     }
118. 
119.     public void inOrder2(Node p){
120.         Stack<Node> stack = new Stack<Node>();
121.         Node node = p;
122.         while(node != null || !stack.isEmpty()){
123.             while(node != null){
124.                 stack.push(node);
125.                 node = node.left;
126.             }
127.             if(!stack.isEmpty()){
128.                 node = stack.pop();
129.                 visit(node);
130.                 node = node.right;
131. 
132.             }
133.         }
134.     }
135. 
136.     public void IOS(){
137.         inOrder(root);
138.     }
139. 
140.     public void postOrder(Node p){
141.         Node q = p;
142.         Stack<Node> stack = new Stack<Node>();
143.         while(p != null){
144.             //stack in left tree
145.             for(; p.left != null; p = p.left){
146.                 stack.push(p);
147.             }
148.             //current node have no right children or right children have been visi
    t
149.             while(p != null && (p.right == null || p.right == q)){
150.                 visit(p);
151.                 q = p;
152.                 if(stack.isEmpty())
153.                     return;
154.                 p = stack.pop();
155.             }
156.             stack.push(p);
157.             p = p.right;
158.         }
159.     }
```

```
160.
161.     //double stack
162.     public void postOrder2(Node p){
163.         Stack<Node> leftstack = new Stack<Node>();
164.         Stack<Node> rightstack = new Stack<Node>();
165.         Node node = p, right;
166.         do{
167.             while(node != null){
168.                 right = node.right;
169.                 leftstack.push(node);
170.                 rightstack.push(right);
171.                 node = node.left;
172.             }
173.             node = leftstack.pop();
174.             right = rightstack.pop();
175.             if(right == null){
176.                 visit(node);
177.             }else{
178.                 leftstack.push(node);
179.                 rightstack.push(null);
180.             }
181.             node = right;
182.
183.         }while(!leftstack.isEmpty() || !rightstack.isEmpty());
184.     }
185.
186.     //single stack
187.     public void postOrder3(Node p){
188.         Stack<Node> stack = new Stack<Node>();
189.         Node node = p, prev = p;
190.         while (node != null || stack.size() > 0) {
191.             while (node != null) {
192.                 stack.push(node);
193.                 node = node.left;
194.             }
195.             if (stack.size() > 0) {
196.                 Node temp = stack.peek().right;
197.                 if (temp == null || temp == prev) {
198.                     node = stack.pop();
199.                     visit(node);
200.                     prev = node;
201.                     node = null;
202.                 } else {
203.                     node = temp;
204.                 }
205.             }
206.
207.         }
208.     }
209.
210.     //double stack 2
211.     public void postOrder4(Node p){
212.         Stack<Node> stack = new Stack<Node>();
213.         Stack<Node> temp = new Stack<Node>();
```

```
214.         Node node = p;
215.         while (node != null || stack.size() > 0) {
216.             while (node != null) {
217.                 temp.push(node);
218.                 stack.push(node);
219.                 node = node.right;
220.             }
221.             if (stack.size() > 0) {
222.                 node = stack.pop();
223.                 node = node.left;
224.             }
225.         }
226.         while (temp.size() > 0) {//把插入序列都插入到了temp。
227.             node = temp.pop();
228.             visit(node);
229.         }
230.     }
231.
232.     public void reverseLeftAndRight(Node p){
233.         if(p == null)
234.             return;
235.         if(null == p.left && null == p.right)
236.             return;
237.         Node temp = p.left;
238.         p.left = p.right;
239.         p.right = temp;
240.         reverseLeftAndRight(p.left);
241.         reverseLeftAndRight(p.right);
242.     }
243.
244.     //use Queue
245.     public void reverseAgain(Node p){
246.         if(p == null)
247.             return;
248.         if(null == p.left && null == p.right)
249.             return;
250.         Queue<Node> queue = new LinkedList<Node>();
251.         queue.offer(p);
252.         Node temp;
253.         Node q = p;
254.         while(!queue.isEmpty()){
255.             if(null != q.left){
256.                 queue.offer(q.left);
257.             }
258.             if(null != q.right){
259.                 queue.offer(q.right);
260.             }
261.             temp = q.left;
262.             q.left = q.right;
263.             q.right = temp;
264.             q = queue.poll();
265.         }
266.
267.     }
```

```
268.
269.     public void reverse(){
270.         reverseAgain(root);
271.     }
272.
273.     public void POS(){
274.         postOrder2(root);
275.     }
276.
277.      public void morris_inorder(Node root) {
278.             while(root != null) {
279.                 if(root.left != null) {
280.                     Node temp = root.left;
281.                     while(temp.right != null && temp.right != root) {
282.                         temp = temp.right;
283.                     }
284.                     if(temp.right == null) {
285.                         temp.right = root;
286.                         root = root.left;
287.                     } else {
288.                         System.out.print(root.value + " ");
289.                         temp.right = null;
290.                         root = root.right;
291.                     }
292.                 } else {
293.                     System.out.print(root.value + " ");
294.                     root = root.right;
295.                 }
296.             }
297.
298.     }
299.
300.     public static void main(String... args){
301.         int[] arr = {15,6,23,4,7,71,5,50};
302.         BinaryTree b = new BinaryTree(arr);
303.         b.reverse();
304.         b.ppp();
305.
306.     }
307. }
```

```
1.  package com.bo.offer;
2.
3.  import java.util.Arrays;
4.  import java.util.LinkedList;
5.  import java.util.Queue;
6.
7.  public class Tree {
8.
9.      /**
10.      * 包含子树
11.      */
12.     public static boolean IsSubTree(Node first, Node second){
```

```
13.         boolean result = false;
14.         if (first != null && second != null) {
15.             if (first.val == second.val) {
16.                 result = DoesContains(first, second);
17.             }
18.             if(!result)
19.                 result = IsSubTree(first.left, second);
20.             if(!result)
21.                 result = IsSubTree(first.right, second);
22.         }
23.
24.         return result;
25.     }
26.
27.     public static boolean DoesContains(Node first, Node second){
28.         if (second == null) {
29.             return true;
30.         }
31.         if (first == null) {
32.             return false;
33.         }
34.         if (first.val != second.val) {
35.             return false;
36.         }
37.         return DoesContains(first.left, second.left) && DoesContains(first.right,
    second.right);
38.     }
39.
40.     /**
41.      * 镜像
42.      */
43.     public static void Mirror(Node root){
44.         if (root == null) {
45.             return;
46.         }
47.         if (root.left ==null && root.right == null) {
48.             return;
49.         }
50.
51.         Node temp = root.right;
52.         root.right = root.left;
53.         root.left = temp;
54.         if (root.left != null) {
55.             Mirror(root.left);
56.         }
57.         if (root.right != null) {
58.             Mirror(root.right);
59.         }
60.     }
61.
62.     /**
63.      * 层次遍历
64.      */
65.     public static void LevelPrint(Node root){
```

```
66.         if(root == null)
67.             return;
68.         Queue<Node> queue = new LinkedList<>();
69.         queue.offer(root);
70.         while(!queue.isEmpty()){
71.             Node node = queue.poll();
72.             System.out.print(node.val + " ");
73.             if (node.left != null) {
74.                 queue.offer(node.left);
75.             }
76.             if(node.right != null){
77.                 queue.offer(node.right);
78.             }
79.         }
80.     }
81.
82.     /**
83.      * 判断输入序列是不是某二叉树的后续遍历
84.      */
85.     public static boolean VerifySequenceOfBST(int[] sequence){
86.         if (sequence.length < 1) {
87.             return false;
88.         }
89.
90.         int root = sequence[sequence.length-1];
91.         int i = 0;
92.         for (;i < sequence.length - 1; i++) {
93.             if(sequence[i] > root)
94.                 break;
95.         }
96.         int j=i;
97.         for (j = i; j < sequence.length-1; j++) {
98.             if(sequence[j] < root)
99.                 return false;
100.         }
101.         //这里注意使用Arrays.copyOfRange的时候开始下标和结束下标 实际取得元素是 se
    quence[start....end-1]
102.         boolean left = true;
103.         if (i> 0) {
104.             left = VerifySequenceOfBST(Arrays.copyOfRange(sequence, 0, i));
105.         }
106.         boolean right = true;
107.         if (i < sequence.length -1) {
108.             right = VerifySequenceOfBST(Arrays.copyOfRange(sequence, i, sequence.l
    ength-i));
109.         }
110.
111.         return (left && right);
112.     }
113.
114.     /**
115.      * 二叉树中和为某一值得路径
116.      */
117.     public static void FindPath(Node node, int expect){
```

```
118.         if (node == null) {
119.             return;
120.         }
121.         LinkedList<Integer> path = new LinkedList<Integer>();
122.         int currentsum = 0;
123.         FindPath(node, expect, path, currentsum);
124.     }
125. 
126.     public static void FindPath(Node root, int expect, LinkedList<Integer> path, i
    nt currentsum){
127.         currentsum += root.val;
128.         path.add(root.val);
129. 
130.         boolean isleaf = root.left == null && root.right == null;
131.         if (currentsum == expect && isleaf) {
132.             //输出list中的路径
133.             for(int i:path)
134.                 System.out.print(i+" ");
135.             System.out.println();
136.         }
137.         if (root.left != null) {
138.             FindPath(root.left, expect, path, currentsum);
139.         }
140.         if(root.right != null){
141.             FindPath(root.right, expect, path, currentsum);
142.         }
143. 
144.         path.removeLast();
145.     }
146. 
147.     public static void main(String[] args) {
148.         int[] first = {8,8,7,9,2,-1,-1,-1,-1,4,7};
149. //      int[] second = {8,9,2};
150.         Node n_first = LevelConstruct(first);
151. //      Node n_second = LevelConstruct(second);
152. //
153. //      System.out.println(IsSubTree(n_first, n_second));
154. //      LevelPrint(n_first);
155. //      int[] sequence = {5,7,6,9,11,10,8};
156. //      System.out.println(VerifySequenceOfBST(sequence));
157. 
158.         int[] bst = {10,5,12,4,7};
159.         Node root = LevelConstruct(bst);
160.         FindPath(root, 22);
161.     }
162. 
163.     //层次构建子树
164.     public static Node LevelConstruct(int[] data){
165.         Node[] nodes = new Node[data.length];;
166.         if (data.length > 0) {
167.             for (int i = 0; i < nodes.length; i++) {
168.                 if(data[i] == -1){
169.                     nodes[i] = null;
170.                 }else
```

```
                    nodes[i] = new Node(data[i]);
                }
            }
            for (int i= (data.length-2)/2; i>=0;i--) {
                Node node = nodes[i];
                node.left = nodes[2*i+1];
                node.right = nodes[2*i+2];
            }
            return nodes[0];
        }

        //构建二叉搜索树
        public static Node BinarySearchTree(int[] data){
            if (data.length < 1) {
                return null;
            }
            Node root = new Node(data[0]);
            for (int i = 1; i < data.length; i++) {
                Node node = new Node(data[i]);
                Node move = root;
                while(move != null){
                    if (move.val < node.val) {
                        move = move.right;
                    }else
                        move = move.left;
                }

            }
        }
    }

    class Node{
        int val;
        Node left;
        Node right;

        public Node(){}

        public Node(int val){
            this.val = val;
            this.left = null;
            this.right = null;
        }
    }
```