```
1
2  /*
3   *    An implementation of a generic Binary Search Tree.
4   */
5
6  import java.util.Scanner;
7
8  public class BST<T extends Comparable <T>> implements BSTInterface<T>
9  {
10     private Node root = null;   // The root of the binary tree.
11
12     public BST()
13     {
14         root = null;
15     }
16
17     public void makeEmpty()
18     {
19         root = null;
20     }
21
22     public boolean isEmpty()
23     {
24         return root == null;
25     }
26
27
```

```
27
28
29      /*
30       *   equals() − return true if two BST’s have the same
31       *              structure and data values, else
32       *              return false.
33       */
34
35      public boolean equals(BST<T> b)
36      {
37          return requals(this.root, b.root);
38      }
39
40
41      private boolean requals(Node r, Node b)
42      {
43          if(r==null&&b==null){
44              return true;
45          }
46          if(r!=null&&b!=null){
47              return ((r.data.equals(b.data))&&requals(r.lchild,b.lchild)&&requals(r.rchi
    ld,b.rchild));
48          }
49          return false;
50      }
51
52
53
```

```
54
55      /*
56       *  insert() − insert a node with data value x
57       *             in the Binary Search Tree.
58       */
59
60      public void insert(T x)
61      {
62          this.root = rinsert(this.root, x);
63      }
64
65
66      /*
67       *   rinsert() − return a pointer to the root of a BST
68       *               with data item x inserted.  Do not
69       *               insert duplicate data items.
70       */
71
72      private Node rinsert(Node root, T x)
73      {
74          if (root == null)                          // Base Step − Empty tree
75              root =  new Node(x, null, null);
76          else if (x.compareTo(root.data) < 0)        // Smaller values go in
77              root.lchild =  rinsert(root.lchild, x); // the left subtree,
78          else if (x.compareTo(root.data) > 0)        // larger values go in
79              root.rchild =  rinsert(root.rchild, x); // the right subtree.
80
81          return root;
82
83      }
84
85
86      public void printTree()
87      {
88          rPrintTree(root,0);
89      }
90
91      /*
92       *    rPrintTree() − the usual quick recursive method to print a tree.
93       */
94
95      public void rPrintTree(Node r, int level)
96      {
97
98          if (r == null)          // Empty tree.
99              return;
100
101         rPrintTree(r.rchild, level + 1);   // Print the right subtree.
102
103         for (int i = 0; i < level; i++)
104             System.out.print("   ");
105
106         System.out.println(r.data.toString());
107
108         rPrintTree(r.lchild, level + 1);
109     }
110
111     public void preorder()
112     {
113         rpreorder(this.root);  //stub
114     }
115
116     /*
117      *    rpreorder() − a recursive routine to perform
118      *                  a preorder traversal of a BST.
119      *                  We will simply write the data items
120      *                  the order they are visited by the traversal.
121      */
122
```

```
123     private void rpreorder(Node r)
124     {
125         if(r==null){
126             return;
127         }
128         System.out.print(r.data+" ");
129         rpreorder(r.lchild);
130         rpreorder(r.rchild);
131     }
132 
133     public void postorder()
134     {
135         rpostorder(this.root);
136     }
137 
138     /*
139      *    rpostorder() − a recursive routine to perform
140      *                   a postorder traversal of a BST.
141      *                   We will simply write the data items
142      *                   the order they are visited by the traversal.
143      */
144 
145     private void rpostorder(Node r)
146     {
147         if(r==null){
148             return;
149         }
150         rpostorder(r.lchild);
151         rpostorder(r.rchild);
152         System.out.print(r.data+" ");
153     }
154 
155 
156     /*
157      *   Perform an inorder traversal of the tree.
158      */
159 
160     public void inorder()
161     {
162         rinorder(this.root);
163     }
164 
165     /*
166      *    rinorder() − a recursive routine to perform
167      *                 an inorder traversal of a BST.
168      *                 We will simply write the data items
169      *                 the order they are visited by the traversal.
170      */
171 
172     private void rinorder(Node r)
173     {
174         if(r==null){
175             return;
176         }
177         rinorder(r.lchild);
178         System.out.print(r.data+" ");
179         rinorder(r.rchild);
180     }
181 
182 
183     public T find(T x)
184     {
185         Node ptr=root;
186         while(ptr!=null){
187             if(ptr.data.equals(x)){
188                 return ptr.data;
189             }
190             else if(x.compareTo(ptr.data)>0){
191                 ptr=ptr.rchild;
```

```
192             }
193             else{
194                 ptr=ptr.lchild;
195             }
196         }
197         return null;
198     }
199 
200 
201     public T findMax()
202     {
203         if(root==null){
204             return null;
205         }
206         Node ptr=root;
207         while(ptr.rchild!=null){
208             ptr=ptr.rchild;
209         }
210         return ptr.data;
211     }
212 
213 
214     public T findMin()
215     {
216         if(root==null){
217             return null;
218         }
219         Node ptr=root;
220         while(ptr.lchild!=null){
221             ptr=ptr.lchild;
222         }
223         return ptr.data;
224     }
225 
226     public void removeMin()
227     {
228         T x = findMin();
229 
230         if (x == null)
231            return;
232 
233         remove(x);
234     }
235 
236     public void removeMax()
237     {
238         T x = findMax();
239 
240         if (x == null)
241             return;
242 
243         remove(x);
244     }
245 
246 
247 
```

```
248     public void remove(T x)
249     {
250         Node ptr = root;
251         Node follow = ptr;
252         boolean done = false;
253 
254         while (!done) {
255             if (ptr == null)
256                 return;
257             else if (x.compareTo(ptr.data) == 0) {
258                 done = true;
259             }
260             else if (x.compareTo(ptr.data) < 0) {
261                 follow = ptr;
262                 ptr = ptr.lchild;
263             }
264             else {
265                 follow = ptr;
266                 ptr = ptr.rchild;
267             }
268         }
269 
270         // Handle the case where the node to delete is a leaf.
271 
272         if (ptr.lchild == null && ptr.rchild == null) {
273             if (ptr == root)
274                 root = null;
275             else if (follow.lchild == ptr)
276                 follow.lchild = null;
277             else
278                 follow.rchild = null;
279             return;
280         }
281 
282         // Handle the case where the node to delete has a left subtree
283         // but no right subtree.
284 
285         if (ptr.rchild == null) {
286             if (ptr == root)
287                 root = ptr.lchild;
288             else if (follow.lchild == ptr)
289                 follow.lchild = ptr.lchild;
290             else if (follow.rchild == ptr)
291                 follow.rchild = ptr.lchild;
292             return;
293         }
294 
295         // Handle the case where the node to delete has a right subtree
296         // but no left subtree.
297 
298         if (ptr.lchild == null) {
299             if (ptr == root)
300                 root = ptr.rchild;
301             else if (follow.lchild == ptr)
302                 follow.lchild = ptr.rchild;
303             else if (follow.rchild == ptr)
304                 follow.rchild = ptr.rchild;
305             return;
306         }
307 
308 
```

```
309         // Handle the case where the node to delete has both subtrees.
310         // In this case, we swap the data in the node with the
311         // smallest data value in its right subtree, then we delete this
312         // "smallest data" node.
313
314         Node smallPtr = ptr.rchild;
315
316         // Locate the smallest data value in the right subtree.
317
318         follow = ptr;
319
320         while (smallPtr.lchild != null) {
321             follow = smallPtr;
322             smallPtr = smallPtr.lchild;
323         }
324
325         T temp = ptr.data;           // Swap the values.
326         ptr.data = smallPtr.data;
327         smallPtr.data = temp;
328
329         // Now delete the smallPtr node.
330
331         if (ptr.rchild == smallPtr) {
332             ptr.rchild = smallPtr.rchild;
333         }
334         else {
335             follow.lchild = smallPtr.rchild;
336         }
337
338
339
340
341     }
342
```

```
343
344
345
346       //
347       //  inner Node class
348       //
349
350       private class Node
351       {
352           T data;
353           Node lchild;
354           Node rchild;
355
356
357           public Node()
358           {
359               this.data = null;
360               this.lchild = null;
361               this.rchild = null;
362           }
363
364
365           public Node(T data, Node lchild, Node rchild)
366           {
367               this.data = data;
368               this.lchild = lchild;
369               this.rchild = rchild;
370           }
371       }
372
373
374       public static void main(String[] args)
375       {
376           Scanner keyb = new Scanner(System.in);
377
378           BST<Integer> t1 = new BST<Integer>();  // Create Binary Search Trees
379           BST<Integer> t2 = new BST<Integer>();
380           BST<Integer> t3 = new BST<Integer>();
381           BST<Integer> t4 = new BST<Integer>();
382           BST<Integer> t5 = new BST<Integer>();
383           BST<Integer> t6 = new BST<Integer>();
384           BST<Integer> t7 = new BST<Integer>();
385           BST<Integer> t8 = new BST<Integer>();
386           BST<Integer> t9 = new BST<Integer>();
387           BST<Integer> t10 = new BST<Integer>();
388           BST<Integer> t11 = new BST<Integer>();
389           BST<Integer> t12 = new BST<Integer>();
390           BST<Integer> t13 = new BST<Integer>();
391
392
393           t1.insert(40);   // Two identical trees are created to test the equals()
394           t1.insert(60);   // method.  To test trees that are not equal, simply
395           t1.insert(20);   // comment out one of the Insertions − or change one of
396           t1.insert(new Integer(50));   // the data values.
397           t1.insert(80);
398           t1.insert(70);
399           t1.insert(new Integer(10));
400           t1.insert(48);
401           t1.insert(41);
402           t1.insert(46);
403           t1.insert(47);
404           t1.insert(44);
405           t1.insert(5);
406           t1.insert(15);
407
408
409           t2.insert(40);
410           t2.insert(60);
411           t2.insert(20);
```

```
412         t2.insert(50);
413         t2.insert(80);
414         t2.insert(70);
415         t2.insert(10);
416         t2.insert(48);
417         t2.insert(41);
418         t2.insert(46);
419         t2.insert(47);
420         t2.insert(44);
421         t2.insert(5);
422         t2.insert(15);
423
424         System.out.println(" tree t1 ----------------");
425         t1.printTree();
426         System.out.println(" tree t2 ----------------");
427         t2.printTree();
428         System.out.println("------------------------");
429
430         System.out.println("t1 and t2 should be equal.");
431
432
433         if (t1.equals(t2))
434             System.out.println("Tree t1 equals tree t2\n");
435         else
436             System.out.println("Tree t1 doesn't equal tree t2\n");
437
438         System.out.println("--------------------");
439
440         t4.insert(40);   // Two identical trees are created to test the equals()
441         t4.insert(60);   // method.  To test trees that are not equal, simply
442         t4.insert(20);   // comment out one of the Insertions − or change one of
443         t4.insert(new Integer(50));   // the data values.
444         t4.insert(80);
445         t4.insert(90);
446         t4.insert(new Integer(10));
447         t4.insert(48);
448         t4.insert(41);
449         t4.insert(46);
450         t4.insert(47);
451         t4.insert(44);
452         t4.insert(5);
453         t4.insert(15);
454
455
456         t5.insert(40);
457         t5.insert(60);
458         t5.insert(20);
459         t5.insert(50);
460         t5.insert(80);
461         t5.insert(70);
462         t5.insert(10);
463         t5.insert(48);
464         t5.insert(41);
465         t5.insert(46);
466         t5.insert(47);
467         t5.insert(44);
468         t5.insert(5);
469         t5.insert(15);
470
471
472         System.out.println(" tree t4 ----------------");
473         t4.printTree();
474         System.out.println(" tree t5 ----------------");
475         t5.printTree();
476         System.out.println("------------------------");
477
478
479         System.out.println("t4 and t5 should not be equal.");
480
```

```
481         if (t5.equals(t4))
482             System.out.println("Tree t4 equals tree t5\n");
483         else
484             System.out.println("Tree t4 doesn't equal tree t5\n");
485 
486         System.out.println(" tree t1 ----------------");
487         t1.printTree();
488         System.out.println(" tree t3 ----------------");
489         t3.printTree();
490 
491         System.out.println("---------------------");
492         System.out.println("t1 and t3 should not be equal.");
493         if (t1.equals(t3))
494             System.out.println("Tree t1 equals tree t3\n");
495         else
496             System.out.println("Tree t1 doesn't equal tree t3\n");
497 
498         System.out.println(" tree t9 ----------------");
499         t9.printTree();
500         System.out.println(" tree t3 ----------------");
501         t3.printTree();
502 
503         System.out.println("t9 and t3 should be equal.");
504         if (t9.equals(t3))
505             System.out.println("Tree t9 equals tree t3\n");
506         else
507             System.out.println("Tree t9 doesn't equal tree t3\n");
508 
509         System.out.println("---------------------");
510         t10.insert(20);
511         t10.insert(10);
512         t10.insert(2);
513         t10.insert(40);
514 
515         t11.insert(20);
516         t11.insert(10);
517         t11.insert(2);
518 
519         System.out.println(" tree t10 ----------------");
520         t10.printTree();
521         System.out.println(" tree t11 ----------------");
522         t11.printTree();
523         System.out.println("-------------------------");
524 
525         System.out.println("---------------------");
526         System.out.println("t10 and t11 should not be equal.");
527         if (t10.equals(t11))
528             System.out.println("Tree t10 equals tree t11\n");
529         else
530             System.out.println("Tree t10 doesn't equal tree t11\n");
531 
532         System.out.println("---------------------");
533         t12.insert(20);
534         t12.insert(10);
535         t12.insert(40);
536         t12.insert(50);
537 
538         t13.insert(20);
539         t13.insert(10);
540         t13.insert(2);
541         t13.insert(40);
542         t13.insert(50);
543 
544         System.out.println(" tree t12 ----------------");
545         t12.printTree();
546         System.out.println(" tree t13 ----------------");
547         t13.printTree();
548         System.out.println("-------------------------");
549 
```

```
550         System.out.println("----------------------");
551         System.out.println("t12 and t13 should not be equal.\n");
552         if (t10.equals(t11))
553             System.out.println("Tree t12 equals tree t13");
554         else
555             System.out.println("Tree t12 doesn't equal tree t13\n");
556 
557         System.out.println("\n\nTesting findMin()\n");
558         System.out.println("The minimum value in t1 is " + t1.findMin());
559         System.out.println("The minimum value in t10 is " + t10.findMin());
560         System.out.println("The minimum value in t8 is " + t8.findMin());
561 
562 
563         System.out.println("\n\nTesting findMax()\n");
564         System.out.println("The maximum value in t1 is " + t1.findMax());
565         System.out.println("The maximum value in t10 is " + t10.findMax());
566         System.out.println("The maximum value in t8 is " + t8.findMax());
567 
568 
569         System.out.println("\n\nTesting inorder traversal\n\n");
570         System.out.println("\nAn inorder traversal of t1 is:");
571         t1.inorder();
572         System.out.println("\nAn inorder traversal of t10 is:");
573         t10.inorder();
574         System.out.println("\nAn inorder traversal of t11 is:");
575         t11.inorder();
576 
577         System.out.println("\n\nTesting preorder traversal\n\n");
578         System.out.println("\nA preorder traversal of t1 is:");
579         t1.preorder();
580         System.out.println("\nA preorder traversal of t10 is:");
581         t10.preorder();
582         System.out.println("\nA preorder traversal of t11 is:");
583         t11.preorder();
584 
585         System.out.println("\n\nTesting postorder traversal\n\n");
586         System.out.println("\nA postorder traversal of t1 is:");
587         t1.postorder();
588         System.out.println("\nA postorder traversal of t10 is:");
589         t10.postorder();
590         System.out.println("\nA postorder traversal of t11 is:");
591         t11.postorder();
592 
593         System.out.println("\n\nTest find()\n");
594 
595         Integer  n;
596         do {
597             System.out.print("Enter a value to search for in t1 (-1 to quit): ");
598             n = keyb.nextInt();
599             Integer value = t1.find(n);
600             if (value == null)
601                 System.out.println(n.toString() + " is not in t1");
602             else
603                 System.out.println(value.toString() + " is in t1");
604         } while (n != -1);
605 
606     }
607 
608 }
```