PROGRAMMING PROJECT 5

Posting ID: 5439-820

1 CONTENTS

2 PLEDGE

I pledge that this submission is entirely my own work. I have not attempted to find other solutions to the same problem, and I have not looked at or shown my code or write-up to any person other than the grader, tutors, or instructor. I understand that I may discuss ideas with others but I may not share code. I understand that, should I accidentally violate any of these conditions I must inform the grader and instructor immediately before submitting my work. I understand that if I am unable to explain aspects of my code to a grader when I am asked, then it will be considered cheating, and I will receive a grade of EX (failure due to a breach of academic integrity) in the course.

Signed: [Armando Minor] [02-21-16]

3 REFLECTION

Beginning my project one of the first things I did was understand the desired outcome. I took into consideration what would be the best approach for myself tackling this project. My best approach was to take the whole project and divide it into pieces to make the workload easier. Each method needed to complete the project was done individually. This approach helped me compile the desired outcome with less effort and less errors, although I did have a few. I researched a lot of my errors since many were new to me and I wasn't sure how to fix them. Once fixing them I saved the information on how to fix the errors going forward. I ran a few tests

during and after my project was completed to ensure the desired result was achieved. One of my biggest challenges was to get the desired outcome for the methods. Making each one was working correctly and efficiently. The time taken on this assignment exceeded those done earlier in the semester.

4 SOURCE CODE

```
1 import java.util.*;
2 import java.util.Iterator;
3 /**
4  * LinkedBinaryTree implements the BinaryTreeADT interface
5  *
6  * @author Lewis and Chase
7  * @version 4.0
8  */
9 public class LinkedBinaryTree<T> implements BinaryTreeADT<T>, I
     terable<T>
10   {
11       protected BinaryTreeNode<T> root;
12       protected int modCount;
13
14       /**
15        * Creates an empty binary tree.
16        */
17       public LinkedBinaryTree()
18       {
19           root = null;
20       }
21
22       /**
23        * Creates a binary tree with the specified element as
     its root.
24        *
25        * @param element the element that will become the root
      of the binary tree
26        */
27       public LinkedBinaryTree(T element)
28       {
29           root = new BinaryTreeNode<T>(element);
30       }
31
32       /**
33        * Creates a binary tree with the specified element as
     its root and the
34        * given trees as its left child and right child
35        *
36        * @param element the element that will become the root
```

```
     of the binary tree
37       * @param left the left subtree of this tree
38       * @param right the right subtree of this tree
39       */
40      public LinkedBinaryTree(T element, LinkedBinaryTree<T>
    left,
41                              LinkedBinaryTree<T> right)
42      {
43          root = new BinaryTreeNode<T>(element);
44          root.setLeft(left.root);
45          root.setRight(right.root);
46      }
47
48      /**
49       * Returns a reference to the element at the root
50       *
51       * @return a reference to the specified target
52       * @throws EmptyCollectionException if the tree is empt
    y
53       */
54      @Override
55      public T getRootElement() throws EmptyCollectionExcepti
    on
56      {
57          return root.getElement();
58      }
59
60      /**
61       * Returns a reference to the node at the root
62       *
63       * @return a reference to the specified node
64       * @throws EmptyCollectionException if the tree is empt
    y
65       */
66      protected BinaryTreeNode<T> getRootNode() throws EmptyC
    ollectionException
67      {
68          return root;
69      }
70
71      /**
72       * Returns the left subtree of the root of this tree.
73       *
74       * @return a link to the left subtree fo the tree
75       */
76      public LinkedBinaryTree<T> getLeft()
77      {
```

```
78          if (root == null)
79              throw new EmptyCollectionException
80                      ("Get left operation " + "failed. The t
    ree is empty.");
81          LinkedBinaryTree<T> result = new LinkedBinaryTree<T
    >();
82          result.root = root.getLeft();
83          return result;
84      }
85
86      /**
87       * Returns the right subtree of the root of this tree.
88       *
89       * @return a link to the right subtree of the tree
90       */
91      public LinkedBinaryTree<T> getRight()
92      {
93          if (root == null)
94              throw new EmptyCollectionException
95                      ("Get left operation " + "failed. The t
    ree is empty.");
96          LinkedBinaryTree<T> result = new LinkedBinaryTree<T
    >();
97          result.root = root.getRight();
98          return result;
99      }
100
101     /**
102      * Returns true if this binary tree is empty and false
    otherwise.
103      *
104      * @return true if this binary tree is empty, false oth
    erwise
105      */
106     @Override
107     public boolean isEmpty()
108     {
109         return (root == null);
110     }
111
112     /**
113      * Returns the integer size of this tree.
114      *
115      * @return the integer size of the tree
116      */
117     @Override
118     public int size()
```

```
119     {
120         int result = 0;
121         if (root != null)
122             result = ;
123         return result;
124     }
125 
126     /**
127      * Returns the height of this tree.
128      *
129      * @return the height of the tree
130      */
131     public int getHeight()
132     {
133         return height(root);
134     }
135 
136     /**
137      * Returns the height of the specified node.
138      *
139      * @param node the node from which to calculate the hei
    ght
140      * @return the height of the tree
141      */
142     private int height(BinaryTreeNode<T> node)
143     {
144         if (node == null) {
145             return 0;
146         } else {
147             int leftHeight = height(node.getLeft());
148             int rightHeight = height(node.getRight());
149             return (1 + Math.max(leftHeight, rightHeight));

150         }
151     }
152 
153     /**
154      * Returns true if this tree contains an element that m
    atches the
155      * specified target element and false otherwise.
156      *
157      * @param targetElement the element being sought in thi
    s tree
158      * @return true if the element in is this tree, false o
    therwise
159      */
160     @Override
```

```
161     public boolean contains(T targetElement)
162     {
163         boolean result = false;
164         try {
165             T element = this.find(targetElement);
166             if (element != null){
167                 result = true;
168             }
169         } catch (ElementNotFoundException exception){
170             result = false;
171         } catch (EmptyCollectionException ece) {
172             result = false;
173         }
174         return result;
175     }
176 
177     /**
178      * Returns a reference to the specified target element
    if it is
179      * found in this binary tree.  Throws a ElementNotFound
    Exception if
180      * the specified target element is not found in the bin
    ary tree.
181      *
182      * @param targetElement the element being sought in thi
    s tree
183      * @return a reference to the specified target
184      * @throws ElementNotFoundException if the element is n
    ot in the tree
185      */
186     public T find(T targetElement) throws ElementNotFoundEx
    ception
187     {
188         BinaryTreeNode<T> current = findNode(targetElement,
     root);
189 
190         if (current == null)
191             throw new ElementNotFoundException("LinkedBinar
    yTree");
192 
193         return (current.getElement());
194     }
195 
196     /**
197      * Returns a reference to the specified target element
    if it is
198      * found in this binary tree.
```

```
199      *
200      * @param targetElement the element being sought in thi
     s tree
201      * @param next the element to begin searching from
202      */
203     private BinaryTreeNode<T> findNode(T targetElement,
204                                        BinaryTreeNode<T> ne
     xt)
205     {
206         if (next == null)
207             return null;
208 
209         if (next.getElement().equals(targetElement))
210             return next;
211 
212         BinaryTreeNode<T> temp = findNode(targetElement, ne
     xt.getLeft());
213 
214         if (temp == null)
215             temp = findNode(targetElement, next.getRight())
     ;
216 
217         return temp;
218     }
219 
220     /**
221      * Returns a string representation of this binary tree
     showing
222      * the nodes in an inorder fashion.
223      *
224      * @return a string representation of this binary tree
225      */
226     @Override
227     public String toString()
228     {
229         if (root == null){
230             return "empty tree.";
231         }
232         return root.toString();
233     }
234 
235     /**
236      * Returns an iterator over the elements in this tree u
     sing the
237      * iteratorInOrder method
238      *
239      * @return an in order iterator over this binary tree
```

```
240      */
241     @Override
242     public Iterator<T> iterator()
243     {
244         return iteratorInOrder();
245     }
246 
247     /**
248      * Performs an inorder traversal on this binary tree by
     calling an
249      * overloaded, recursive inorder method that starts wit
    h
250      * the root.
251      *
252      * @return an in order iterator over this binary tree
253      */
254     @Override
255     public Iterator<T> iteratorInOrder()
256     {
257         ArrayUnorderedList<T> tempList = new ArrayUnordered
    List<T>();
258         inOrder(root, tempList);
259 
260         return new TreeIterator(tempList.iterator());
261     }
262 
263     /**
264      * Performs a recursive inorder traversal.
265      *
266      * @param node the node to be used as the root for this
     traversal
267      * @param tempList the temporary list for use in this t
    raversal
268      */
269     protected void inOrder(BinaryTreeNode<T> node,
270                            ArrayUnorderedList<T> tempList)

271     {
272         if (node != null)
273         {
274             inOrder(node.getLeft(), tempList);
275             tempList.addToRear(node.getElement());
276             inOrder(node.getRight(), tempList);
277         }
278     }
279 
280     /**
```

```
281      * Performs an preorder traversal on this binary tree b
     y calling
282      * an overloaded, recursive preorder method that starts
      with
283      * the root.
284      *
285      * @return a pre order iterator over this tree
286      */
287     @Override
288     public Iterator<T> iteratorPreOrder()
289     {
290         //Not required.
291         return null;
292     }
293 
294     /**
295      * Performs a recursive preorder traversal.
296      *
297      * @param node the node to be used as the root for this
      traversal
298      * @param tempList the temporary list for use in this t
     raversal
299      */
300     protected void preOrder(BinaryTreeNode<T> node,
301                             ArrayUnorderedList<T> tempList)

302     {
303         //Not required.
304     }
305 
306     /**
307      * Performs an postorder traversal on this binary tree
     by calling
308      * an overloaded, recursive postorder method that start
     s
309      * with the root.
310      *
311      * @return a post order iterator over this tree
312      */
313     @Override
314     public Iterator<T> iteratorPostOrder()
315     {
316         //Not required.
317         return null;
318     }
319 
320     /**
```

```
321      * Performs a recursive postorder traversal.
322      *
323      * @param node the node to be used as the root for this
     traversal
324      * @param tempList the temporary list for use in this t
    raversal
325      */
326     protected void postOrder(BinaryTreeNode<T> node,
327                              ArrayUnorderedList<T> tempList
    )
328     {
329         //Not required.
330     }
331
332     /**
333      * Performs a levelorder traversal on this binary tree,
     using a
334      * templist.
335      *
336      * @return a levelorder iterator over this binary tree
337      */
338     @Override
339     public Iterator<T> iteratorLevelOrder()
340     {
341         ArrayUnorderedList<BinaryTreeNode<T>> nodes =
342                 new ArrayUnorderedList<>();
343         ArrayUnorderedList<T> tempList = new ArrayUnordered
    List<>();
344         BinaryTreeNode<T> current;
345
346         nodes.addToRear(root);
347
348         while (!nodes.isEmpty())
349         {
350             current = nodes.removeFirst();
351
352             if (current != null)
353             {
354                 tempList.addToRear(current.getElement());
355                 if (current.getLeft() != null)
356                     nodes.addToRear(current.getLeft());
357                 if (current.getRight() != null)
358                     nodes.addToRear(current.getRight());
359             }
360             else
361                 tempList.addToRear(null);
362         }
```

```
363
364         return new TreeIterator(tempList.iterator());
365     }
366
367     /**
368      * Inner class to represent an iterator over the elemen
    ts of this tree
369      */
370     private class TreeIterator implements Iterator<T>
371     {
372         private int expectedModCount;
373         private Iterator<T> iter;
374
375         /**
376          * Sets up this iterator using the specified iterat
    or.
377          *
378          * @param iter the list iterator created by a tree
    traversal
379          */
380         public TreeIterator(Iterator<T> iter)
381         {
382             this.iter = iter;
383             expectedModCount = modCount;
384         }
385
386         /**
387          * Returns true if this iterator has at least one m
    ore element
388          * to deliver in the iteration.
389          *
390          * @return  true if this iterator has at least one
    more element to deliver
391          *          in the iteration
392          * @throws  ConcurrentModificationException if the
    collection has changed
393          *          while the iterator is in use
394          */
395         @Override
396         public boolean hasNext() throws ConcurrentModificat
    ionException
397         {
398             if (!(modCount == expectedModCount))
399                 throw new ConcurrentModificationException()
    ;
400
401             return (iter.hasNext());
```

```
402         }
403
404         /**
405          * Returns the next element in the iteration. If th
    ere are no
406          * more elements in this iteration, a NoSuchElement
    Exception is
407          * thrown.
408          *
409          * @return the next element in the iteration
410          * @throws NoSuchElementException if the iterator i
    s empty
411          */
412         @Override
413         public T next() throws NoSuchElementException
414         {
415             if (hasNext())
416                 return (iter.next());
417             else
418                 throw new NoSuchElementException();
419         }
420
421         /**
422          * The remove operation is not supported.
423          *
424          * @throws UnsupportedOperationException if the rem
    ove operation is called
425          */
426         @Override
427         public void remove()
428         {
429             throw new UnsupportedOperationException();
430         }
431     }
432 }
433
434
435
436
437
438
439
440 /**
441  * LinkedBinarySearchTree implements the BinarySearchTreeAD
    T interface
442  * with links.
443  *
```

```
444   * @author Lewis and Chase
445   * @version 4.0
446   */
447  public class LinkedBinarySearchTree<T> extends LinkedBinary
     Tree<T>
448          implements BinarySearchTreeADT<T> {
449      /**
450       * Creates an empty binary search tree.
451       */
452      public LinkedBinarySearchTree() {
453          super();
454      }
455
456      /**
457       * Creates a binary search with the specified element a
     s its root.
458       *
459       * @param element the element that will be the root of
     the new binary
460       *                search tree
461       */
462      public LinkedBinarySearchTree(T element) {
463          super(element);
464
465          if (!(element instanceof Comparable))
466              throw new NonComparableElementException("Linked
     BinarySearchTree");
467      }
468
469      /**
470       * Adds the specified object to the binary search tree
     in the
471       * appropriate position according to its natural order.
       Note that
472       * equal elements are added to the right.
473       *
474       * @param element the element to be added to the binary
      search tree
475       */
476      @Override
477      public void addElement(T element) {
478          if (!(element instanceof Comparable))
479              throw new NonComparableElementException("Linked
     BinarySearchTree");
480
481          Comparable<T> comparableElement = (Comparable<T>) e
     lement;
```

```
482
483         if (isEmpty())
484             root = new BinaryTreeNode<T>(element);
485         else {
486             if (comparableElement.compareTo(root.getElement
    ()) < 0) {
487                 if (root.getLeft() == null)
488                     this.getRootNode().setLeft(new BinaryTr
    eeNode<T>(element));
489                 else
490                     addElement(element, root.getLeft());
491             } else {
492                 if (root.getRight() == null)
493                     this.getRootNode().setRight(new BinaryT
    reeNode<T>(element));
494                 else
495                     addElement(element, root.getRight());
496             }
497         }
498         modCount++;
499     }
500
501     /**
502      * Adds the specified object to the binary search tree
    in the
503      * appropriate position according to its natural order.
      Note that
504      * equal elements are added to the right.
505      *
506      * @param element the element to be added to the binary
     search tree
507      */
508     private void addElement(T element, BinaryTreeNode<T> no
    de) {
509         Comparable<T> comparableElement = (Comparable<T>) e
    lement;
510
511         if (comparableElement.compareTo(node.getElement())
    < 0) {
512             if (node.getLeft() == null)
513                 node.setLeft(new BinaryTreeNode<T>(element)
    );
514             else
515                 addElement(element, node.getLeft());
516         } else {
517             if (node.getRight() == null)
518                 node.setRight(new BinaryTreeNode<T>(element
```

```
    ));
519             else
520                 addElement(element, node.getRight());
521         }
522     }
523
524
525     /**
526      * Removes the first element that matches the specified
     target
527      * element from the binary search tree and returns a re
    ference to
528      * it.  Throws a ElementNotFoundException if the specif
    ied target
529      * element is not found in the binary search tree.
530      *
531      * @param targetElement the element being sought in the
     binary search tree
532      * @throws ElementNotFoundException if the target eleme
    nt is not found
533      */
534     @Override
535     public T removeElement(T targetElement)
536             throws ElementNotFoundException {
537         T result = null;
538
539         if (isEmpty())
540             throw new ElementNotFoundException("LinkedBinar
    ySearchTree");
541         else {
542             BinaryTreeNode<T> parent = null;
543             if (((Comparable<T>) targetElement).equals(root
    .element)) {
544                 result = root.element;
545                 BinaryTreeNode<T> temp = replacement(root);

546                 if (temp == null)
547                     root = null;
548                 else {
549                     root.element = temp.element;
550                     root.setRight(temp.right);
551                     root.setLeft(temp.left);
552                 }
553
554                 modCount--;
555             } else {
556                 parent = root;
```

```
557                 if (((Comparable) targetElement).compareTo(
    root.element) < 0)
558                     result = removeElement(targetElement, r
    oot.getLeft(), parent);
559                 else
560                     result = removeElement(targetElement, r
    oot.getRight(), parent);
561             }
562         }
563 
564         return result;
565     }
566 
567     /**
568      * Removes the first element that matches the specified
     target
569      * element from the binary search tree and returns a re
    ference to
570      * it.  Throws a ElementNotFoundException if the specif
    ied target
571      * element is not found in the binary search tree.
572      *
573      * @param targetElement the element being sought in the
     binary search tree
574      * @param node          the node from which to search
575      * @param parent        the parent of the node from whi
    ch to search
576      * @throws ElementNotFoundException if the target eleme
    nt is not found
577      */
578     private T removeElement(T targetElement, BinaryTreeNode
    <T> node, BinaryTreeNode<T> parent)
579             throws ElementNotFoundException {
580         T result = null;
581 
582         if (node == null)
583             throw new ElementNotFoundException("LinkedBinar
    ySearchTree");
584         else {
585             if (((Comparable<T>) targetElement).equals(node
    .element)) {
586                 result = node.element;
587                 BinaryTreeNode<T> temp = replacement(node);

588                 if (parent.right == node)
589                     parent.right = temp;
590                 else
```

```
591                     parent.left = temp;
592
593                 modCount--;
594             } else {
595                 parent = node;
596                 if (((Comparable) targetElement).compareTo(
    node.element) < 0)
597                     result = removeElement(targetElement, n
    ode.getLeft(), parent);
598                 else
599                     result = removeElement(targetElement, n
    ode.getRight(), parent);
600             }
601         }
602
603         return result;
604     }
605
606     /**
607      * Returns a reference to a node that will replace the
    one
608      * specified for removal.  In the case where the remove
    d node has
609      * two children, the inorder successor is used as its r
    eplacement.
610      *
611      * @param node the node to be removed
612      * @return a reference to the replacing node
613      */
614     private BinaryTreeNode<T> replacement(BinaryTreeNode<T>
     node) {
615         BinaryTreeNode<T> result = null;
616
617         if ((node.left == null) && (node.right == null))
618             result = null;
619
620         else if ((node.left != null) && (node.right == null
    ))
621             result = node.left;
622
623         else if ((node.left == null) && (node.right != null
    ))
624             result = node.right;
625
626         else {
627             BinaryTreeNode<T> current = node.right;
628             BinaryTreeNode<T> parent = node;
```

```
629
630             while (current.left != null) {
631                 parent = current;
632                 current = current.left;
633             }
634
635             current.left = node.left;
636             if (node.right != current) {
637                 parent.left = current.right;
638                 current.right = node.right;
639             }
640
641             result = current;
642         }
643
644         return result;
645     }
646
647     /**
648      * Removes elements that match the specified target ele
    ment from
649      * the binary search tree. Throws a ElementNotFoundExce
    ption if
650      * the specified target element is not found in this tr
    ee.
651      *
652      * @param targetElement the element being sought in the
     binary search tree
653      * @throws ElementNotFoundException if the target eleme
    nt is not found
654      */
655     @Override
656     public void removeAllOccurrences(T targetElement)
657             throws ElementNotFoundException {
658         removeElement(targetElement);
659
660         try {
661             while (contains((T) targetElement))
662                 removeElement(targetElement);
663         } catch (Exception ElementNotFoundException) {
664         }
665     }
666
667     /**
668      * Removes the node with the least value from the binar
    y search
669      * tree and returns a reference to its element.  Throws
```

```
 an
670      * EmptyCollectionException if this tree is empty.
671      *
672      * @return a reference to the node with the least value

673      * @throws EmptyCollectionException if the tree is empt
    y
674      */
675     @Override
676     public T removeMin() throws EmptyCollectionException {

677         T result = null;
678 
679         if (isEmpty())
680             throw new EmptyCollectionException("LinkedBinar
    ySearchTree");
681         else {
682             if (root.left == null) {
683                 result = root.element;
684                 root = root.right;
685             } else {
686                 BinaryTreeNode<T> parent = root;
687                 BinaryTreeNode<T> current = root.left;
688                 while (current.left != null) {
689                     parent = current;
690                     current = current.left;
691                 }
692                 result = current.element;
693                 parent.left = current.right;
694             }
695 
696             modCount--;
697         }
698 
699         return result;
700     }
701 
702     /**
703      * Removes the node with the highest value from the bin
    ary
704      * search tree and returns a reference to its element.
     Throws an
705      * EmptyCollectionException if this tree is empty.
706      *
707      * @return a reference to the node with the highest val
    ue
708      * @throws EmptyCollectionException if the tree is empt
```

```
y
709          */
710         @Override
711         public T removeMax() throws EmptyCollectionException {
712             T result = null;
713 
714             if (isEmpty())
715                 throw new EmptyCollectionException("binary tree
    ");
716             else {
717                 if (root.right == null) {
718                     result = root.element;
719                     root = root.left;
720                 } //if
721                 else {
722                     BinaryTreeNode<T> parent = root;
723                     BinaryTreeNode<T> current = root.right;
724 
725                     while (current.right != null) {
726                         parent = current;
727                         current = current.right;
728                     } //while
729 
730                     result = current.element;
731                     parent.right = current.left;
732                 } //else
733 
734                 count--;
735             } //else
736 
737             return result;
738         }
739 
740         /**
741          * Returns the element with the least value in the bina
    ry search
742          * tree. It does not remove the node from the binary se
    arch tree.
743          * Throws an EmptyCollectionException if this tree is e
    mpty.
744          *
745          * @return the element with the least value
746          * @throws EmptyCollectionException if the tree is empt
    y
747          */
748         @Override
```

```
749     public T findMin() throws EmptyCollectionException {
750         T result = null;
751 
752         if (isEmpty())
753             throw new EmptyCollectionException("binary tree
    ");
754         else {
755             BinaryTreeNode<T> current = root;
756 
757             while (current.left != null)
758                 current = current.left;
759 
760             result = current.element;
761         } //else
762 
763         return result;
764     }
765 
766     /**
767      * Returns the element with the highest value in the bi
    nary
768      * search tree.  It does not remove the node from the b
    inary
769      * search tree.  Throws an EmptyCollectionException if
    this
770      * tree is empty.
771      *
772      * @return the element with the highest value
773      * @throws EmptyCollectionException if the tree is empt
    y
774      */
775     @Override
776     public T findMax() throws EmptyCollectionException {
777         T result = null;
778 
779         if (isEmpty())
780             throw new EmptyCollectionException("binary tree
    ");
781         else {
782             BinaryTreeNode<T> current = root;
783 
784             while (current.right != null)
785                 current = current.right;
786 
787             result = current.element;
788         } //else
789 
```

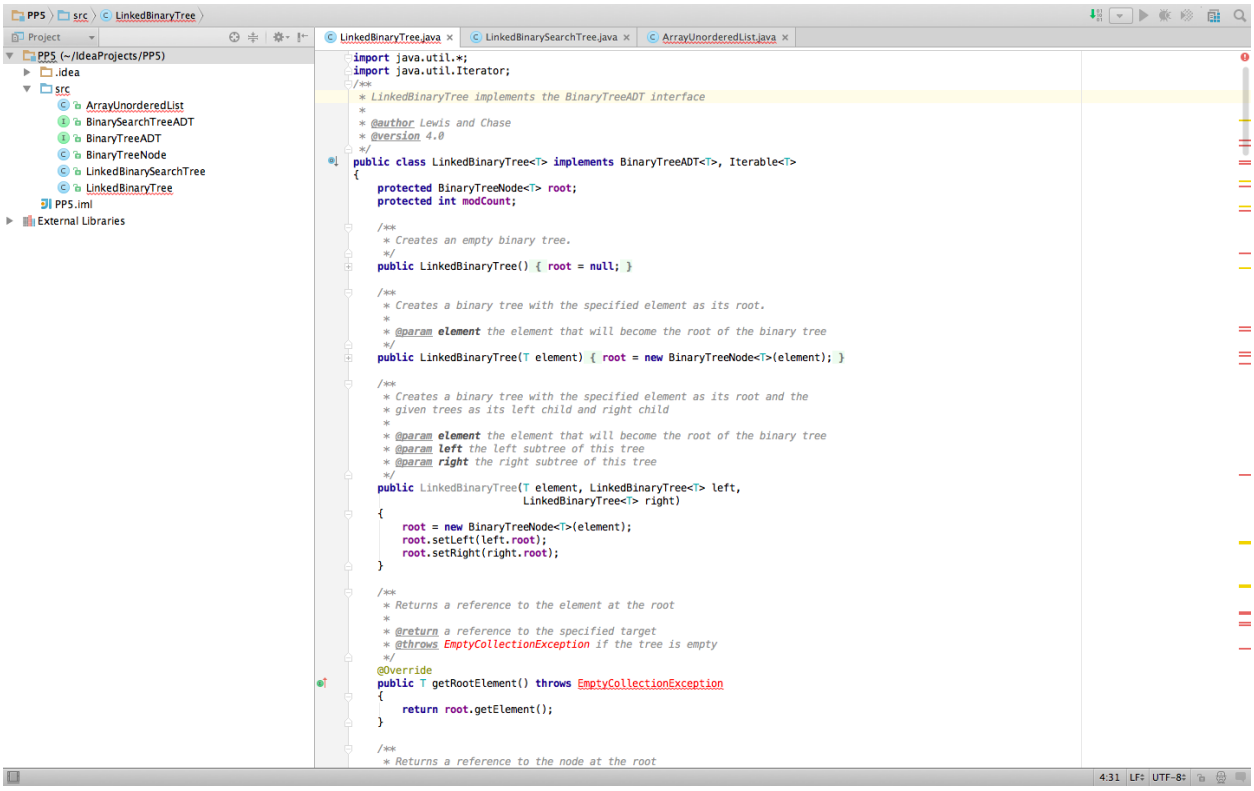
```
790             return result;
791         }
792 
793         /**
794          * Returns a reference to the specified target element
    if it is
795          * found in the binary tree.  Throws a NoSuchElementExc
    eption if
796          * the specified target element is not found in this tr
    ee.
797          *
798          * @param targetElement the element being sough in the
    binary tree
799          * @throws ElementNotFoundException if the target eleme
    nt is not found
800          */
801         //@Override
802         public T find(T targetElement) throws ElementNotFoundEx
    ception {
803             BinaryTreeNode<T> current = root;
804             BinaryTreeNode<T> temp = current;
805 
806             if (!(current.element.equals(targetElement)) && (cu
    rrent.left != null) && (((Comparable) current.element).comp
    areTo(targetElement) > 0))
807                 current = findNode(targetElement, current.left)
    ;
808 
809             else if (!(current.element.equals(targetElement)) &
    & (current.right != null))
810                 current = findNode(targetElement, current.right
    );
811 
812             if (!(current.element.equals(targetElement)))
813                 throw new ElementNotFoundException("binarytree"
    );
814 
815             return current.element;
816         }
817 
818         //  Returns a reference to target if found
819 
820         private BinaryTreeNode<T> findNode(T targetElement, Bin
    aryTreeNode<T> next) {
821             BinaryTreeNode<T> current = next;
822             if (!(next.element.equals(targetElement)) && (next.
    left != null) && (((Comparable) next.element).compareTo(tar
```

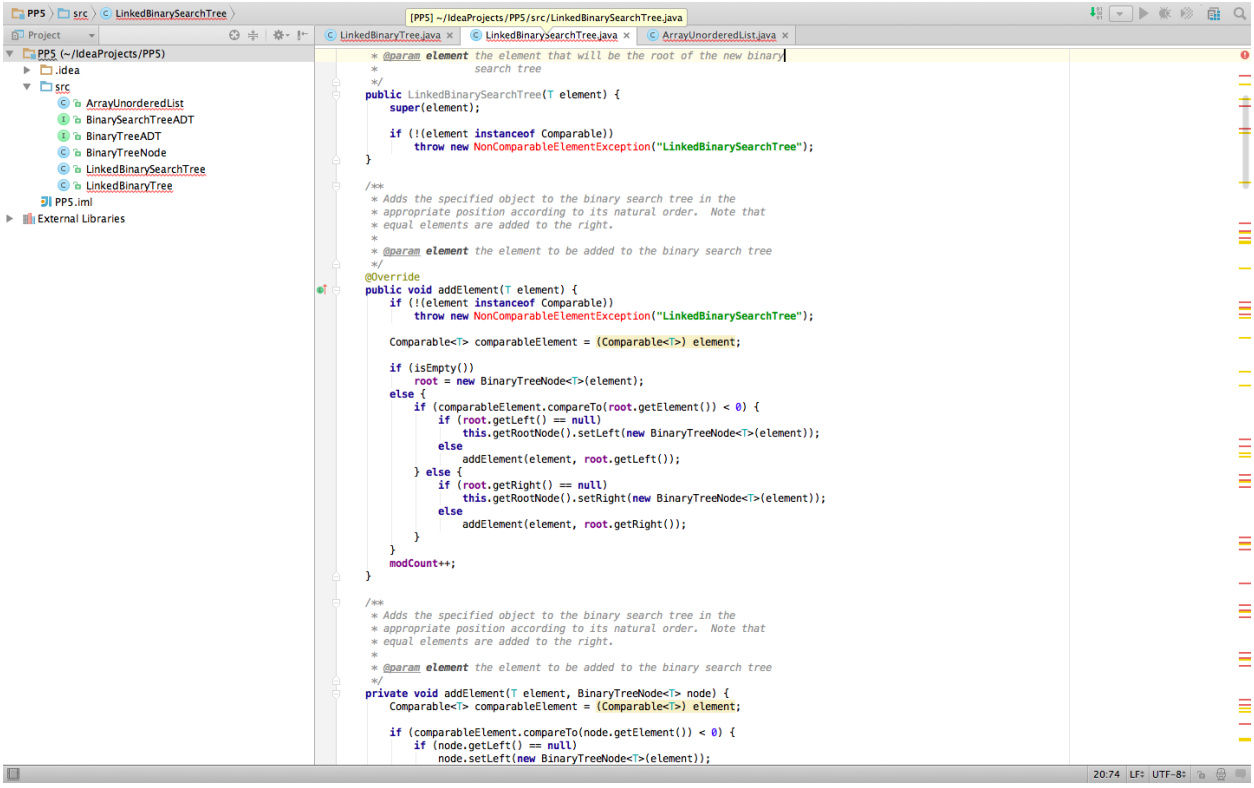
```
    getElement) > 0))
823             next = findNode(targetElement, next.left);
824         else if (!(next.element.equals(targetElement)) && (
    next.right != null))
825             next = findNode(targetElement, next.right);
826
827         return next;
828
829     }
830 }
831
832
833
834
835
836
837
838
839
840
841
842
843
844 /**
845  * ArrayUnorderedList represents an array implementation of
     an unordered list.
846  *
847  * @author Lewis and Chase
848  * @version 4.0
849  */
850 public class ArrayUnorderedList<T> extends ArrayList<T>
851         implements UnorderedListADT<T>
852 {
853     /**
854      * Creates an empty list using the default capacity.
855      */
856     public ArrayUnorderedList()
857     {
858         super();
859     }
860
861     /**
862      * Creates an empty list using the specified capacity.
863      *
864      * @param initialCapacity the intial size of the list
865      */
866     public ArrayUnorderedList(int initialCapacity)
```

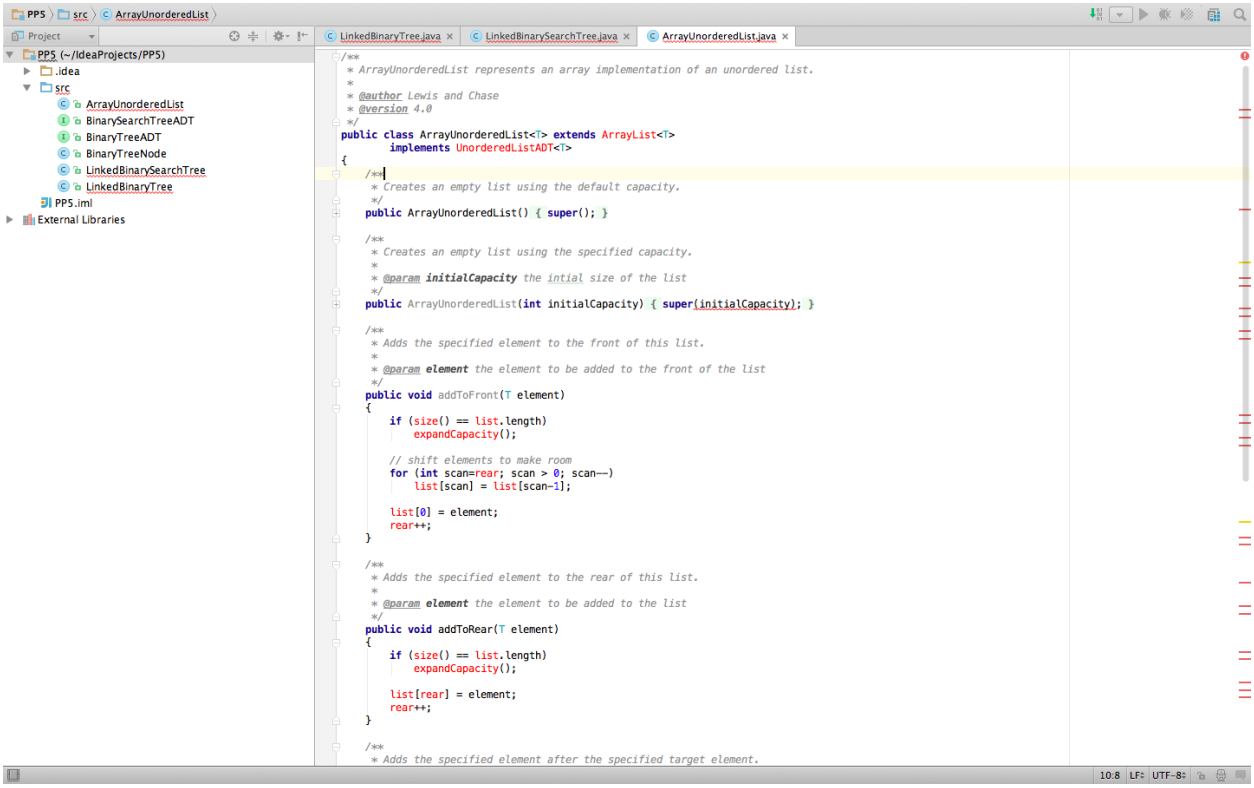
```
867     {
868         super(initialCapacity);
869     }
870 
871     /**
872      * Adds the specified element to the front of this list
    .
873      *
874      * @param element the element to be added to the front
    of the list
875      */
876     public void addToFront(T element)
877     {
878         if (size() == list.length)
879             expandCapacity();
880 
881         // shift elements to make room
882         for (int scan=rear; scan > 0; scan--)
883             list[scan] = list[scan-1];
884 
885         list[0] = element;
886         rear++;
887     }
888 
889     /**
890      * Adds the specified element to the rear of this list.

891      *
892      * @param element the element to be added to the list
893      */
894     public void addToRear(T element)
895     {
896         if (size() == list.length)
897             expandCapacity();
898 
899         list[rear] = element;
900         rear++;
901     }
902 
903     /**
904      * Adds the specified element after the specified targe
    t element.
905      * Throws an ElementNotFoundException if the target is
    not found.
906      *
907      * @param element the element to be added after the tar
    get element
```

```
908         * @param target  the target that the element is to be
    added after
909         */
910        public void addAfter(T element, T target)
911        {
912            if (size() == list.length)
913                expandCapacity();
914
915            int scan = 0;
916
917            // find the insertion point
918            while (scan < rear && !target.equals(list[scan]))
919                scan++;
920
921            if (scan == rear)
922                throw new ElementNotFoundException("UnorderedLi
    st");
923
924            scan++;
925
926            // shift elements up one
927            for (int shift=rear; shift > scan; shift--)
928                list[shift] = list[shift-1];
929
930            // insert element
931            list[scan] = element;
932            rear++;
933            modCount++;
934        }
    }
```

5 } COMPILATION OUTPUT


PP5 src LinkedBinaryTree
Project
LinkedBinaryTree.java
LinkedBinarySearchTree.java
ArrayUnorderedList.java
PP5 (~/IdeaProjects/PP5)
.idea
src
ArrayUnorderedList
BinarySearchTreeADT
BinaryTreeADT
BinaryTreeNode
LinkedBinarySearchTree
LinkedBinaryTree
PP5.iml
External Libraries
import java.util.*;
import java.util.Iterator;
/**
* LinkedBinaryTree implements the BinaryTreeADT interface
*
* @author Lewis and Chase
* @version 4.0
*/
public class LinkedBinaryTree<T> implements BinaryTreeADT<T>, Iterable<T>
{
protected BinaryTreeNode<T> root;
protected int modCount;
/**
* Creates an empty binary tree.
*/
public LinkedBinaryTree() { root = null; }
/**
* Creates a binary tree with the specified element as its root.
*
* @param element the element that will become the root of the binary tree
*/
public LinkedBinaryTree(T element) { root = new BinaryTreeNode<T>(element); }
/**
* Creates a binary tree with the specified element as its root and the
* given trees as its left child and right child
*
* @param element the element that will become the root of the binary tree
* @param left the left subtree of this tree
* @param right the right subtree of this tree
*/
public LinkedBinaryTree(T element, LinkedBinaryTree<T> left,
LinkedBinaryTree<T> right)
{
root = new BinaryTreeNode<T>(element);
root.setLeft(left.root);
root.setRight(right.root);
}
/**
* Returns a reference to the element at the root
*
* @return a reference to the specified target
* @throws EmptyCollectionException if the tree is empty
*/
@Override
public T getRootElement() throws EmptyCollectionException
{
return root.getElement();
}
/**
* Returns a reference to the node at the root
4:31 LF: UTF-8:


PP5 src LinkedBinarySearchTree
[PP5] ~/IdeaProjects/PP5/src/LinkedBinarySearchTree.java
Project
LinkedBinaryTree.java
LinkedBinarySearchTree.java
ArrayUnorderedList.java
PP5 (~/IdeaProjects/PP5)
.idea
src
ArrayUnorderedList
BinarySearchTreeADT
BinaryTreeADT
BinaryTreeNode
LinkedBinarySearchTree
LinkedBinaryTree
PP5.iml
External Libraries
* @param element the element that will be the root of the new binary
* search tree
*/
public LinkedBinarySearchTree(T element) {
super(element);
if (!(element instanceof Comparable))
throw new NonComparableElementException("LinkedBinarySearchTree");
}
/**
* Adds the specified object to the binary search tree in the
* appropriate position according to its natural order. Note that
* equal elements are added to the right.
*
* @param element the element to be added to the binary search tree
*/
@Override
public void addElement(T element) {
if (!(element instanceof Comparable))
throw new NonComparableElementException("LinkedBinarySearchTree");
Comparable<T> comparableElement = (Comparable<T>) element;
if (isEmpty())
root = new BinaryTreeNode<T>(element);
else {
if (comparableElement.compareTo(root.getElement()) < 0) {
if (root.getLeft() == null)
this.getRootNode().setLeft(new BinaryTreeNode<T>(element));
else
addElement(element, root.getLeft());
} else {
if (root.getRight() == null)
this.getRootNode().setRight(new BinaryTreeNode<T>(element));
else
addElement(element, root.getRight());
}
}
modCount++;
}
/**
* Adds the specified object to the binary search tree in the
* appropriate position according to its natural order. Note that
* equal elements are added to the right.
*
* @param element the element to be added to the binary search tree
*/
private void addElement(T element, BinaryTreeNode<T> node) {
Comparable<T> comparableElement = (Comparable<T>) element;
if (comparableElement.compareTo(node.getElement()) < 0) {
if (node.getLeft() == null)
node.setLeft(new BinaryTreeNode<T>(element));
20:74 LF: UTF-8:



6 TESTING STRATEGY

My testing strategy for this assignments was to make sure each method was invoked properly. With each call one has to be careful and ensure each step is processed properly. Without these precautions I would have been unable to complete the project successfully. The biggest challenges were to invoke these methods properly. With the helper methods the correct structure has to be in place to make the correct calls to the methods. Once the methods are place correctly one looks for any compile errors to ensure the quality of the project. Once I finished the correct code I detailed the code with comments to enhance readability.