10/26/15 01:55:39 /home/15504319/DSA120/DSAAssignment/connorLib/BinarySearchTree.java

```
  1  /*****************************************************************************
  2  *    FILE: BinarySearchTree.java
  3  *    AUTHOR: Connor Beardsmore - 15504319
  4  *    UNIT: DSA120 Assignment S2- 2015
  5  *    PURPOSE: Stores a Binary Tree in Memory with Generic key and value
  6  *    LAST MOD: 22/10/15
  7  *  REQUIRES: java.util
  8  *****************************************************************************/
  9  package connorLib;
 10  import java.util.*;
 11
 12  public class BinarySearchTree<K extends Comparable<K>, V>
 13  {
 14  //---------------------------------------------------------------------------
 15      //PRIVATE INNER CLASS TreeNode
 16      //COMPRESSED FOR SIMPLICITY
 17
 18      public class TreeNode<K extends Comparable<K>, V>
 19      {
 20          private K key;
 21          private V value;
 22          private TreeNode<K,V> leftChild;
 23          private TreeNode<K,V> rightChild;
 24
 25          public TreeNode(K inKey, V inValue)
 26          {
 27              if ( inKey == null )
 28              {
 29                  throw new IllegalArgumentException("Key cannot be null");
 30              }
 31              key = inKey;
 32              value = inValue;
 33              leftChild = null;
 34              rightChild = null;
 35          }
 36          public K getKey()
 37          {
 38              return key;
 39          }
 40          public V getValue()
 41          {
 42              return value;
 43          }
 44          public TreeNode<K,V> getLeft()
 45          {
 46              return leftChild;
 47          }
 48          public void setLeft( TreeNode<K,V> newLeft )
 49          {
 50              leftChild = newLeft;
 51          }
 52          public TreeNode<K,V> getRight()
 53          {
 54              return rightChild;
 55          }
 56          public void setRight( TreeNode<K,V> newRight )
 57          {
 58              rightChild = newRight;
 59          }
 60          public String toStrings()
 61          {
 62              return ("Key: " + key + " Value: " + value);
 63          }

 64      }
 65  //---------------------------------------------------------------------------
 66      //CLASSFIELDS
 67      private TreeNode<K,V> root;
 68  //---------------------------------------------------------------------------
 69      //DEFAULT Constructor
 70
 71      public BinarySearchTree()
 72      {
 73          root = null;
 74      }
 75  //---------------------------------------------------------------------------
 76      //ACCESSOR find
 77      //IMPORT: key (String)
 78      //EXPORT: value (Object)
 79      //PURPOSE: Wrapper Method. Kickstarts Recursive Find
 80
 81      public V find(K key)
 82      {
 83          return findRec( key, root );
 84      }
```

```
85
86      //ACCESSOR findRec
87      //IMPORT: key (String), currNode (TreeNode)
88      //EXPORT: value (Object)
89      //PURPOSE: Recursively Traverses Tree to Find Specific Node
90
91      private V findRec(K key, TreeNode<K,V> currNode)
92      {
93          V value = null;
94
95          //Element doesn't exist in list
96          //Throwing an exception limits flexibility, avoid it here
97          if ( currNode == null )
98          {
99              value = null;
100         }
101         //Base Case: Element Found. Return it
102         else if ( key.equals(currNode.getKey()) )
103         {
104             value = currNode.getValue();
105         }
106         //Follow Left Child Recursively
107         else if ( key.compareTo(currNode.getKey()) < 0 )
108         {
109             value = findRec( key, currNode.getLeft() );
110         }
111         //Follow Right Child Recursively
112         else
113         {
114             value = findRec( key, currNode.getRight() );
115         }
116         return value;
117     }
118 //---------------------------------------------------------------------------
119     //MUTATOR insert
120     //IMPORT: key (String), value (Object)
121     //EXPORT: status (int). 0 = success. -1 = failure
122     //PURPOSE: Wrapper method, kickstarts recursive insert
123
124     public int insert(K key, V value)
125     {
126         //Adapted for usability with having a tree with linked list nodes
127         //Allows for checking if a specific key exists, and doing something
128         //based on this. Using exception for flow is very poor coding,
129         //couldn't determine a workaround and thus used a kind of error return
130         //instead.
131         int status = 0;
132         try
133         {
134             root = insertRec( key, value, root );
135         }
136         catch(IllegalStateException e)
137         {
138             status = -1;
139         }
140         return status;
141     }
142
143     //MUTATOR insertRec
144     //IMPORT: key (String), value (Object), currNode (TreeNode)
145     //EXPORT: updateNode (TreeNode)
146     //PURPOSE: Recursively inserts New Node Into Tree At Bottom Level
147
148     //THIS IS INEFFICIENT. COULD DO EASIER ITERATIVELY
149     //HAVE KEPT SINCE IT WAS IN THE LECTURE SLIDES
150
151     private TreeNode<K,V> insertRec(K key, V value, TreeNode<K,V> currNode)
152     {
153         TreeNode<K,V> updateNode = currNode;
154
155         //Create New Node At Bottom Level
156         if ( currNode == null )
157         {
158             updateNode = new TreeNode<K,V>( key, value );
159         }
160
161         //Key Already Exists in Tree
162         else if ( key.equals( currNode.getKey() ) )
163         {
164                 throw new IllegalStateException("Key Already Exists in Tree");
165         }
166         //Remake parent links. Pretty much unrequired if using iterative method
167         else if ( key.compareTo( currNode.getKey() ) < 0 )
168         {
169             currNode.setLeft( insertRec( key, value, currNode.getLeft() ) );
170         }
171
```

```
172         else
173         {
174             currNode.setRight( insertRec( key, value, currNode.getRight() ) );
175         }
176 
177         return updateNode;
178     }
179 
180 //--------------------------------------------------------------------------
181     //MUTATOR delete
182     //IMPORT: key (String)
183     //PURPOSE: Wrapper method, kickstarts recursive insert
184 
185     public void delete(K key)
186     {
187         root = deleteRec( key, root );
188     }
189 
190     //MUTATOR deleteRec
191     //IMPORT: key (String), currNode (TreeNode)
192     //EXPORT: updateNode (TreeNode)
193     //PURPOSE: Recursively Deletes A Given Node From The Tree
194 
195     private TreeNode<K,V> deleteRec( K key, TreeNode<K,V> currNode )
196     {
197         TreeNode<K,V> updateNode = currNode;
198 
199         //Can't Delete If Element Doesn't Exist In Tree
200         if ( currNode == null )
201             throw new NoSuchElementException("Element not in tree. Cannot Delete");
202 
203         //Base Case. Found The Node To Delete
204         else if ( key.equals( currNode.getKey() ) )
205             updateNode = deleteNode( key, currNode );
206 
207         //Recurse Left
208         else if ( key.compareTo( currNode.getKey() ) < 0 )
209             currNode.setLeft( deleteRec( key, currNode.getLeft() ) );
210 
211         //Recurse Right
212         else
213             currNode.setRight( deleteRec( key, currNode.getRight() ) );
214 
215         return updateNode;
216     }
217 //--------------------------------------------------------------------------
218     //MUTATOR deleteNode
219     //IMPORT: key (String), deNode (TreeNode)
220     //EXPORT: updateNode (TreeNode)
221     //PURPOSE: RDeletes Given Node From Tree, Fixes Required Links
222 
223     private TreeNode<K,V> deleteNode( K key, TreeNode<K,V> delNode )
224     {
225         TreeNode<K,V> updateNode = null;
226 
227         //No Children - Simply Delete
228         if ( ( delNode.getLeft() == null ) && ( delNode.getRight() == null ) )
229             updateNode = null;
230 
231         //Left Child - Adopt Orphan
232         else if ( ( delNode.getLeft() != null ) && ( delNode.getRight() == null ) )
233             updateNode = delNode.getLeft();
234 
235         //Right Child - Adopt Orphan
236         else if ( ( delNode.getLeft() == null ) && ( delNode.getRight() != null ) )
237             updateNode = delNode.getRight();
238 
239         //Two Children
240         else
241         {
242             //Sort Out The Successor
243             updateNode = promoteSucc( delNode.getRight() );
244 
245             //No Cycles
246             if ( updateNode != delNode.getRight() )
247             {
248                 //Update Right
249                 updateNode.setRight( delNode.getRight() );
250             }
251 
252             //Update Left
253             updateNode.setLeft( delNode.getLeft() );
254         }
255 
256         return updateNode;
257     }
258 //--------------------------------------------------------------------------
```

```
259      //MUTATOR promoteSucc
260      //IMPORT: currNode (TreeNode)
261      //EXPORT: successor (TreeNode)
262      //PURPOSE: Finds Successor To Promote In Node Deletion
263
264      private TreeNode<K,V> promoteSucc( TreeNode<K,V> currNode )
265      {
266          TreeNode<K,V> successor = currNode;
267
268          if ( currNode.getLeft() != null )
269          {
270              successor = promoteSucc( currNode.getLeft() );
271
272              if ( successor == currNode.getLeft() )
273              {
274                  currNode.setLeft( successor.getRight() );
275              }
276          }
277
278          return successor;
279      }
280  //---------------------------------------------------------------------------
281      //ACCESSOR calcHeight
282      //EXPORT: height (int)
283      //PURPOSE: Wrapper Method, kickstarts height recursive height calculation
284
285      public int calcHeight()
286      {
287          return heightRec( root );
288      }
289
290      //heightRec
291      //IMPORT: currNode (TreeNode)
292      //EXPORT height
293      //PURPOSE: Recursively calculate height of binary tree
294
295      private int heightRec( TreeNode<K,V> currNode )
296      {
297          int height, leftHt, rightHt;
298
299          //Base Case - no more along this branch
300          if ( currNode == null )
301              height = -1;
302          else
303          {
304              //Calc left and right subheights from here
305              leftHt = heightRec( currNode.getLeft() );
306              rightHt = heightRec( currNode.getRight() );
307
308              //Get highest of the two branches
309              if ( leftHt > rightHt )
310              {
311                  height = leftHt + 1;
312              }
313              else
314              {
315                  height = rightHt + 1;
316              }
317          }
318
319          return height;
320      }
321  //---------------------------------------------------------------------------
322      //traverse
323      //IMPORT: traverseType (int)
324      //PURPOSE: Traverse Tree And Output Data in pre/in/post order
325
326      public void traverse( int traverseType )
327      {
328          switch ( traverseType )
329          {
330              case 1: System.out.print("\nPreOrder Traversal: ");
331                      preOrder(root);
332                      break;
333              case 2: System.out.print("\nInOrder Traversal: ");
334                      inOrder(root);
335                      break;
336              case 3: System.out.print("\nPostOrder Traversal: ");
337                      postOrder(root);
338                      break;
339          }
340          System.out.println();
341          System.out.println();
342      }
343
344      //preOrder
345      //IMPORT: localRoot (TreeNode)
```

```
346      //PURPOSE: Recursively Prints PreOrder of Binary Tree
347
348      private void preOrder( TreeNode<K,V> localRoot )
349      {
350          if ( localRoot != null )
351          {
352              System.out.print( localRoot.value + " ");
353              preOrder( localRoot.getLeft() );
354              preOrder( localRoot.getRight() );
355          }
356      }
357
358      //inOrder
359      //IMPORT: localRoot (TreeNode)
360      //PURPOSE: Recursively Prints InOrder of Binary Tree
361
362      private void inOrder( TreeNode<K,V> localRoot )
363      {
364          if ( localRoot != null )
365          {
366              inOrder( localRoot.getLeft() );
367              System.out.print( localRoot.value + " " );
368              inOrder( localRoot.getRight() );
369          }
370      }
371
372      //postOrder
373      //IMPORT: localRoot (TreeNode)
374      //PURPOSE: Recursively Prints PostOrder of Binary Tree
375
376      private void postOrder( TreeNode<K,V> localRoot )
377      {
378          if ( localRoot != null )
379          {
380              postOrder( localRoot.getLeft() );
381              postOrder( localRoot.getRight() );
382              System.out.print( localRoot.value + " " );
383          }
384      }
385  //-------------------------------------------------------------------------
386      //printTree
387      //PURPOSE: Wrapper method to recursively call printSubTree
388
389      public void printTree()
390      {
391          printSubTree(root, "");
392      }
393
394      //printSubTree
395      //IMPORT: root (TreeNode), indent (String)
396      //PURPOSE: Recursively prints Binary Tree in readable format
397
398      private void printSubTree(TreeNode<K,V> root, String indent)
399      {
400          if(root != null)
401          {
402              if(root.getLeft() != null)
403              {
404                  printSubTree(root.getLeft(), indent + "      ");
405                  System.out.println(indent + "     /");
406              }
407
408               System.out.println(indent + root.toStrings());
409
410              if(root.getRight() != null)
411              {
412                  System.out.println(indent + "     \\");
413                  printSubTree(root.getRight(), indent + "      ");
414              }
415            }
416         }
417
418  //-------------------------------------------------------------------------
419  }
```