

```

1  /* Binary Search Tree Data Type
2  *
3  * by Rosanna Heise & AUSCI 235
4  * Dec 1, 2023 at 2:17:34 p.m.
5  *
6  * Contains the following public methods:
7  * default constructor
8  * insert(int, String) --> void
9  * adds a node to the binary search tree, maintaining order
10 * Complexity: O(log n) if tree is balanced
11 * preOrderTraversal() --> String
12 * gives a pre-order traversal of the tree
13 * Complexity: O(n)
14 * postOrderTraversal() --> String
15 * gives a post-order traversal of the tree
16 * Complexity: O(n)
17 * inOrderTraversal() --> String
18 * gives an in-order traversal of the tree
19 * Complexity: O(n)
20 * toString() --> String
21 * gives an in-order traversal of the tree
22 * Complexity: O(n)
23 * findNameOf(int) --> String
24 * finds the name matching the idNum; "Not Found" if idNum is not in tree
25 * Complexity: O(log n) if tree is balanced
26 * findIdNumOf(String) --> int
27 * find the id number matching the name; -1 if name is not in the tree
28 * Complexity: O(n)
29 * sumMinMax() --> int
30 * finds the minimum and maximum values in a binary tree and retruns the sum of
31 * of the two.
32 *
33 */
34
35 public class BSTree {
36     //Data =====
37     private Node root;
38     private int size;
39
40     //Constructor =====
41     public BSTree(){ //start the tree empty to keep control of order
42         root = null;
43         size = 0;
44     }
45
46     //Getter =====
47     public int getSize(){
48         return size;
49     }
50
51     //Put stuff into tree =====
52     /**
53      * Places a new node into the tree, maintaining Binary Search Order.
54      * idNums are assumed unique, so if an idNum repeats, an error
55      * message is printed and the node is NOT added.
56      * Complexity: O(log n) for a balanced tree.
57      *
58      * @param idNum the unique identification number of the person to
59      *             be inserted
60      * @param name the person's name
61      */
62     public void insert(int idNum, String name){
63         if (root == null){
64             root = new Node(idNum, name);
65         }
66         else{
67             insert(idNum, name, location: root);
68         }
69
70         size = size + 1;
71     }
72
73     /**
74      * Recursive worker to place item into the Binary Search Tree, as long
75      * as location starts at root. Note: if location does not start at root
76      * the item will be placed into the subtree starting at location and
77      * part of the tree may be missed yielding an unexpected result.
78      *
79      * @param idNum the unique identification number of the person to
80      *             be inserted
81      * @param name the person's name
82      * @param location the current location of the tree being checked
83      */
84     private void insert(int idNum, String name, Node location){
85         if (idNum < location.getIdNum()){ //Go left
86             if (location.getLeftChild() == null){
87                 location.setLeftChild(new Node(idNum, name));
88             }
89             else{
90                 insert(idNum, name, location: location.getLeftChild());
91             }
92         }
93         else if (idNum > location.getIdNum()){ //Go right
94             if (location.getRightChild() == null){ //Here is where the node goes
95                 location.setRightChild(new Node(idNum, name));
96             }
97             else{
98                 insert(idNum, name, location: location.getRightChild());
99             }
100         }
101         else{ //Same idNums
102             System.out.println("ERROR same id numbers is not allowed " +
103                               idNum + name);
104             size = size - 1;
105         }
106     }
107
108     // PreOrder Traversal =====
109     /**
110      * Gives a String of the pre-order traversal of the tree.
111      * Complexity: O(n)
112      *
113      * @return string which is pre-order traversal of the tree or "null" if
114      *         the tree is empty
115      */
116     public String preOrderTraversal(){
117         if (root == null){
118             return "null";
119         }
120         return preOrderTraversal(location: root);
121     }
122
123     /**
124      * Recursive worker to create the String for the pre-order traversal of
125      * the subtree starting from location. Note: if the subtree is empty it
126      * returns the empty String.
127      *
128      * @param location current location in the tree
129      * @return String which is pre-order traversal of the subtree at the
130      *         given location
131      */
132     private String preOrderTraversal(Node location){
133         if (location == null){ //finished that branch
134             return "";
135         }
136         return location.toString() +

```

```

137         preOrderTraversal(location: location.getLeftChild()) + //go left
138         preOrderTraversal(location: location.getRightChild()); //go right
139     } //preOrderTraversal recursive worker
140
141     // PostOrder Traversal =====
142     /**
143      * Gives a String of the post-order traversal of the tree.
144      * Complexity: O(n)
145      *
146      * @return string which is post-order traversal of the tree or "null" if
147      *         the tree is empty
148      */
149     public String postOrderTraversal(){
150         if (root == null){ //initial tree empty
151             return "null";
152         }
153         return postOrderTraversal(location: root);
154     }
155
156     /**
157      * Recursive worker to create the String for the post-order traversal of
158      * the subtree starting from location. Note: if the subtree is empty it
159      * returns the empty String.
160      *
161      * @param location current location in the tree
162      * @return String which is post-order traversal of the subtree at the
163      *         given location
164      */
165     private String postOrderTraversal(Node location){
166         if (location == null){ //finished that branch
167             return "";
168         }
169         return postOrderTraversal(location: location.getLeftChild()) + //go left
170                postOrderTraversal(location: location.getRightChild()) + //go right
171                location.toString(); //visit
172     } //postOrderTraversal recursive worker
173
174     // InOrder Traversal =====
175     /**
176      * Gives a String of the in-order traversal of the tree.
177      * Complexity: O(n)
178      *
179      * @return string which is in-order traversal of the tree or "null" if
180      *         the tree is empty
181      */
182     public String inOrderTraversal(){
183         if (root == null){ //initial tree empty
184             return "null";
185         }
186         return inOrderTraversal(location: root);
187     }
188
189     /**
190      * Recursive worker to create the String for the in-order traversal of
191      * the subtree starting from location. Note: if the subtree is empty it
192      * returns the empty String.
193      *
194      * @param location current location in the tree
195      * @return String which is in-order traversal of the subtree at the
196      *         given location
197      */
198     private String inOrderTraversal(Node location){
199         if (location == null){ //finished that branch
200             return "";
201         }
202         return inOrderTraversal(location: location.getLeftChild()) + //go left
203                location.toString() + //visit
204                inOrderTraversal(location: location.getRightChild()); //go right
205     } //inOrderTraversal recursive worker
206
207     // toString() =====
208     /**
209      * Returns a String representation of the tree, using an in-order
210      * traversal.
211      * Complexity: O(n)
212      *
213      * @return in-order String of the tree
214      */
215     @Override
216     public String toString(){
217         return inOrderTraversal();
218     }
219
220     // Given idNum, find matching name =====
221     /**
222      * Given an id number, will return the name associated with that id
223      * number. Note: takes efficient single path through the tree, since tree
224      * is sorted by id number.
225      * Complexity: O(log n) for a balanced tree
226      *
227      * @param idNum the id number to look for
228      * @return the name associated with idNum; "NOT FOUND" if idNum
229      *         is not in the tree
230      */
231     public String findNameOf(int idNum){
232         return findNameOf(idNum, location: root);
233     }
234
235     /**
236      * Recursive worker to find the id number in the tree and return the
237      * corresponding name. Note: this will traverse a single branch of the
238      * subtree starting at location.
239      *
240      * @param idNum the id number to look for
241      * @param location the current location in the tree
242      * @return the name associated with idNum; "NOT FOUND" if idNum
243      *         is not in the subtree
244      */
245     private String findNameOf(int idNum, Node location){
246         if (location == null){ //End of that branch, so item is not in the tree
247             return "NOT FOUND";
248         }
249         else if (location.getIdNum() == idNum){ //Found it!!
250             return location.getName();
251         }
252         else{ //Go either left or right depending on the id number
253             if (idNum > location.getIdNum()){
254                 return findNameOf(idNum, location: location.getRightChild());
255             }
256             else{
257                 return findNameOf(idNum, location: location.getLeftChild());
258             }
259         } //else outer
260     } //findNameOf recursive worker
261
262     // Given name, find matching idNum =====
263     /**
264      * Returns the id number associated with the given name.
265      * Since the tree is not sorted by names, this algorithm may search
266      * through the entire tree though it does stop early if the item is
267      * found.
268      * Complexity: O(n)
269      *
270      * @param name the name to search for
271      * @return the id number corresponding to the name or -1 if not found
272      */
273     public int findIdNumOf(String name){

```

```

273 public int findIdNumOf(String name, Node location)
274 {
275     return findIdNumOf(name, location: root);
276 }
277
278 /**
279  * Recursive worker to find the given name in the subtree starting at
280  * location and return the corresponding id number.
281  *
282  * @param name the name to search for
283  * @param location the current location
284  * @return the id number corresponding to the name or -1 if not found
285  */
286 private int findIdNumOf(String name, Node location){
287     if (location == null){ //Reached end of a path, not there
288         return -1;
289     }
290     else if (name.equals(location.getName())){ //Found it!
291         return location.getIdNum();
292     }
293     else{
294         //Go left
295         int answerToLeft;
296         answerToLeft = findIdNumOf(name, location: location.getLeftChild());
297         //Only if we didn't find it on the left do we go right
298         //This helps with complexity because I do not continue the
299         //search unless I have to
300         if (answerToLeft == -1){
301             return findIdNumOf(name, location: location.getRightChild());
302         }
303         else{
304             return answerToLeft;
305         }
306     }
307 } //findIdNumOf recursive worker
308
309 // Any code above here will not be evaluated for the lab exam grade
310 //=====
311 //=====
312 //=====
313 //=====
314 //=====
315 //=====
316 //=====
317 // Put your code here for sumMinMax
318
319 /**
320  * This method finds the minimum and maximum value in a sub tree, then
321  * returns the sum of both those values combined.
322  * If the size is 0 it will return -1 to signal such
323  * if the size is one, it will return double the IdNum value of the tree.
324  *
325  * @return The sum of the min and max values in the binary search tree.
326  */
327 public int sumMinMax(){
328     //if the size of the tree is 0 then it returns -1 to signal this case
329     if(size == 0){
330         return -1;
331     }
332     //if the size of the tree is only one it returns the double the number, as it
333     //returns the min and max value, when size is one, the root is both the min
334     //and max value
335     if(size == 1){
336         return 2 * root.getIdNum();
337     }
338     // Find minimum and maximum
339     return sumMin(currentLoc: root) + sumMax(currentLoc: root);
340 } //sumMinMax
341
342 /**
343  * This method is a helper method to sumMinMax, it returns the sumMin, or the
344  * min value in the sub tree.
345  * @param currentLoc the current location the helper is at in the tree.
346  * @return the minimum value of the binary search tree.
347  */
348 private int sumMin(Node currentLoc) {
349     //as long as the left node is not null it goes left as far as it can,
350     //to find the min value
351     return currentLoc.getLeftChild() == null ? currentLoc.getIdNum() :
352         sumMin(currentLoc: currentLoc.getLeftChild());
353 } //sumMin
354
355 /**
356  * This method is a helper method to sumMinMax, it returns the sumMax, more
357  * precisely the Max value in a sub tree.
358  * @param currentLoc the current location of the helper in the tree.
359  * @return the max value capable of being found in a binary search tree.
360  */
361 private int sumMax(Node currentLoc) {
362     //gets right child as long as right is not null, it continues to go right
363     //as far as it can to find the max value
364     return currentLoc.getRightChild() == null ? currentLoc.getIdNum() :
365         sumMax(currentLoc: currentLoc.getRightChild());
366 } //sumMax
367
368 } //class

```