~\OneDrive - VietNam National University - HCM INTERNATIONAL UNIVERSITY\Desktop\DSA\DSA LAB NEW\Lab 7 Hash Tables\ITITSB22029_DoMinhDuy_Lab7\3\TreeApp.java

```java
import java.io.*;
import java.util.*;

// TreeApp.java
// demonstrates binary tree
public class TreeApp {
   public static void main(String[] args) throws IOException {
      int value;
      Tree theTree = new Tree();

      theTree.insert(50, 1.5);
      theTree.insert(25, 1.2);
      theTree.insert(75, 1.7);
      theTree.insert(12, 1.5);
      theTree.insert(37, 1.2);
      theTree.insert(43, 1.7);
      theTree.insert(30, 1.5);
      theTree.insert(33, 1.2);
      theTree.insert(87, 1.7);
      theTree.insert(93, 1.5);
      theTree.insert(97, 1.5);

      while (true) {
         System.out.print("\nEnter first letter of show and there are some special cases, ");
         System.out.print(
               "insert(i), find(f), delete(d), traverse(t), clear(c), random(r), min(m), max(x), save(a), or quit(q): ");

         int choice = getChar();
         switch (choice) {
            case 's':
               System.out.print("horizontal or vertical (1 or 2)? ");
               value = getInt();
               if (value == 1) {
                  System.out.println();
                  showTree(0, theTree.root);
               } else
                  theTree.displayTree();
               break;
            case 'i':
               System.out.print("Enter value to insert: ");
               value = getInt();
               theTree.insert(value, value + 0.9);
               System.out.println("Comparisons = " + theTree.comps);
               break;
            case 'f':
               System.out.print("Enter value to find: ");
               value = getInt();
               Node found = theTree.find(value);
               if (found != null) {
```

```
50                   System.out.print("Found: ");
51                   found.displayNode();
52                   System.out.print("\n");
53                } else {
54                   System.out.print("Could not find ");
55                   System.out.println(value);
56                }
57                System.out.println("Comparisons = " + theTree.comps);
58                break;
59             case 'd':
60                System.out.print("Enter value to delete: ");
61                value = getInt();
62                boolean didDelete = theTree.delete(value);
63                if (didDelete)
64                   System.out.print("Deleted " + value + '\n');
65                else {
66                   System.out.print("Could not delete ");
67                   System.out.println(value);
68                }
69                System.out.println("Comparisons = " + theTree.comps);
70                break;
71             case 't':
72                System.out.print("Enter type 1, 2 or 3: ");
73                value = getInt();
74                theTree.traverse(value);
75                break;
76             case 'c':
77                theTree.clearTree();
78                System.out.println("Tree cleared.");
79                break;
80             case 'r':
81                System.out.print("Enter the number of random items to insert: ");
82                value = getInt();
83                insertRandomItems(theTree, value);
84                break;
85             case 'm':
86                Node minNode = theTree.findMin();
87                System.out.print("Minimal item: ");
88                minNode.displayNode();
89                System.out.println();
90                break;
91             case 'x':
92                Node maxNode = theTree.findMax();
93                System.out.print("Maximal item: ");
94                maxNode.displayNode();
95                System.out.println();
96                break;
97             case 'a':
98                int[] items = saveItemsInArray(theTree.root);
99                System.out.println("Items saved to array.");
100               theTree.clearTree();
101               System.out.println("Tree cleared.");
102               reinsertionFromArray(theTree, items);
103               break;
```

```
            case 'q':
               return;
            default:
               System.out.print("Invalid entry\n");
         }
      }
   }

   public static String getString() throws IOException {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
   }

   public static char getChar() throws IOException {
      String s = getString();
      return s.charAt(0);
   }

   public static int getInt() throws IOException {
      String s = getString();
      return Integer.parseInt(s);
   }

   public static void showTree(int n, Node t) {
      tab(n);
      if (t == null)
         System.out.println("*");
      else {
         n = n + 3;
         System.out.println(t.iData);
         if (t.leftChild == null && t.rightChild == null)
            return;
         showTree(n, t.leftChild);
         showTree(n, t.rightChild);
      }
   }

   public static void tab(int n) {
      for (int i = 0; i < n; i++)
         System.out.print(" ");
   }

   public static void insertRandomItems(Tree tree, int n) {
      Random rand = new Random();
      for (int i = 0; i < n; i++) {
         int value = rand.nextInt(100) + 1; // Random values between 1 and 100
         tree.insert(value, value + 0.9);
      }
   }

   public static int[] saveItemsInArray(Node root) {
      List<Integer> itemsList = new ArrayList<>();
```

```
      inOrderSave(root, itemsList);
      return itemsList.stream().mapToInt(i -> i).toArray();
   }

   private static void inOrderSave(Node node, List<Integer> itemsList) {
      if (node != null) {
         inOrderSave(node.leftChild, itemsList);
         itemsList.add(node.iData);
         inOrderSave(node.rightChild, itemsList);
      }
   }

   public static void reinsertionFromArray(Tree tree, int[] items) {
      for (int item : items) {
         tree.insert(item, item + 0.9);
      }
   }

   public static Node node(int data, Node l, Node r) {
      Node a = new Node();
      a.iData = data;
      a.leftChild = l;
      a.rightChild = r;
      return a;
   }
}

// Node.java
class Node {
   public int iData; // data item (key)
   public double dData; // data item
   public Node leftChild; // this node's left child
   public Node rightChild; // this node's right child

   public void displayNode() {
      System.out.print('{');
      System.out.print(iData);
      System.out.print(", ");
      System.out.print(dData);
      System.out.print("} ");
   }
}

// Tree.java
class Tree {
   int comps = 0;
   Node root; // first node of tree

   public Tree() {
      root = null;
   }

   public Node find(int key) {
      comps = 0;
```

```
212         Node current = root;
213         while (current.iData != key) {
214            comps++;
215            if (key < current.iData)
216               current = current.leftChild;
217            else
218               current = current.rightChild;
219            if (current == null)
220               return null;
221         }
222         comps++; // final comparison when key is found
223         return current;
224      }
225
226      public void insert(int id, double dd) {
227         comps = 0;
228         Node newNode = new Node();
229         newNode.iData = id;
230         newNode.dData = dd;
231         if (root == null)
232            root = newNode;
233         else {
234            Node current = root;
235            Node parent;
236            while (true) {
237               parent = current;
238               comps++;
239               if (id < current.iData) {
240                  current = current.leftChild;
241                  if (current == null) {
242                     parent.leftChild = newNode;
243                     return;
244                  }
245               } else {
246                  current = current.rightChild;
247                  if (current == null) {
248                     parent.rightChild = newNode;
249                     return;
250                  }
251               }
252            }
253         }
254      }
255
256      public boolean delete(int key) {
257         comps = 0;
258         Node current = root;
259         Node parent = root;
260         boolean isLeftChild = true;
261
262         while (current.iData != key) {
263            comps++;
264            parent = current;
265            if (key < current.iData) {
```

```
266             isLeftChild = true;
267             current = current.leftChild;
268          } else {
269             isLeftChild = false;
270             current = current.rightChild;
271          }
272          if (current == null)
273             return false;
274       }
275 
276       if (current.leftChild == null && current.rightChild == null) {
277          if (current == root)
278             root = null;
279          else if (isLeftChild)
280             parent.leftChild = null;
281          else
282             parent.rightChild = null;
283       } else if (current.rightChild == null) {
284          if (current == root)
285             root = current.leftChild;
286          else if (isLeftChild)
287             parent.leftChild = current.leftChild;
288          else
289             parent.rightChild = current.leftChild;
290       } else if (current.leftChild == null) {
291          if (current == root)
292             root = current.rightChild;
293          else if (isLeftChild)
294             parent.leftChild = current.rightChild;
295          else
296             parent.rightChild = current.rightChild;
297       } else {
298          Node successor = getSuccessor(current);
299          if (current == root)
300             root = successor;
301          else if (isLeftChild)
302             parent.leftChild = successor;
303          else
304             parent.rightChild = successor;
305          successor.leftChild = current.leftChild;
306       }
307       return true;
308    }
309 
310    public Node getSuccessor(Node delNode) {
311       Node successorParent = delNode;
312       Node successor = delNode;
313       Node current = delNode.rightChild;
314       while (current != null) {
315          successorParent = successor;
316          successor = current;
317          current = current.leftChild;
318       }
319 
```

```
320         if (successor != delNode.rightChild) {
321            successorParent.leftChild = successor.rightChild;
322            successor.rightChild = delNode.rightChild;
323         }
324         return successor;
325      }
326
327      public void traverse(int type) {
328         switch (type) {
329            case 1:
330               System.out.print("In-order traversal: ");
331               inOrder(root);
332               break;
333            case 2:
334               System.out.print("Pre-order traversal: ");
335               preOrder(root);
336               break;
337            case 3:
338               System.out.print("Post-order traversal: ");
339               postOrder(root);
340               break;
341            default:
342               System.out.println("Invalid type");
343         }
344         System.out.println();
345      }
346
347      public void inOrder(Node node) {
348         if (node != null) {
349            inOrder(node.leftChild);
350            node.displayNode();
351            inOrder(node.rightChild);
352         }
353      }
354
355      public void preOrder(Node node) {
356         if (node != null) {
357            node.displayNode();
358            preOrder(node.leftChild);
359            preOrder(node.rightChild);
360         }
361      }
362
363      public void postOrder(Node node) {
364         if (node != null) {
365            postOrder(node.leftChild);
366            postOrder(node.rightChild);
367            node.displayNode();
368         }
369      }
370
371      public void displayTree() {
372         System.out.println("\nTree (in-order): ");
373         inOrder(root);
```

```
    }

    public Node findMin() {
        Node current = root;
        while (current.leftChild != null)
            current = current.leftChild;
        return current;
    }

    public Node findMax() {
        Node current = root;
        while (current.rightChild != null)
            current = current.rightChild;
        return current;
    }

    public void clearTree() {
        root = null;
    }
}

// How the Tree Changes:
// In-order Traversal will "flatten" the tree by visiting the nodes in
// increasing order, which is especially useful for a binary search tree because
// it gives you the sorted order of the values.
// Pre-order Traversal starts with the root, which makes it useful when you want
// to process the root before its children (for instance, when copying or saving
// the tree structure).
// Post-order Traversal works from the leaves to the root, making it useful for
// deletion or when you need to process nodes in a bottom-up manner.
// In all of these traversals, the tree structure itself doesn't change (i.e.,
// it remains the same throughout the traversal). Traversal merely determines
// the order in which the nodes are visited or processed, which can be useful in
// different algorithms or operations on the tree.
```