Tunbutr, Bryant

CSCI 220
Data Structures I

Lab Project #4

SEARCH TREE

Due Date
12/3/2013
Date Turned In
12/3/2013

Student Name: _____ Bryant Tunbutr _____ Project Number: _____4_____

Project Name: _____SearchTree_____ Eclipse Version: ___Kepler_____

Files: AvlNode.java, AvlTree.java, AvlDriverClass.java

Project specification

This software is intended to store airport data including airport code, city, and check in time using an AVL Tree Structure

The program is complete and includes the extra credit ability to draw a tree/debug, although it is drawn on its side, it is accurate

I think I learned many lessons:

Do a huge project in bite size chunks, i.e. just get the file to upload, then get it to display

Write many comments to learn how things work

Try to do things to make it checking faster, i.e. I used toUppercase for my functions so I did not have to type with capitals.

Printout of program input/output

OUTPUT
Author: Bryant Tunbutr
Select an option:
0. Debug. This displays the airport code at each node,
but rotated sideways
1. Search for an airport
2. Insert a new airport
3. Delete an airport
4. List all airports
5. Exit

INPUT
0

OUTPUT
```
|     |     |-------SFO
|     |     |     |-------SEA
|     |-------ORD
|     |     |     |-------MSY
|     |     |-------MIA
|-------LAX
|     |     |     |-------LAS
|     |     |-------JFK
|     |-------DFW
|     |     |-------BOS
```

INPUT
3

OUTPUT
Enter the to be deleted airport code

INPUT
lax

OUTPUT
Removal time is 0 milliseconds

INPUT
3

OUTPUT
Enter the to be deleted airport code

INPUT
LAX

OUTPUT
LAX not found
Removal time is 4 milliseconds

INPUT
1

OUTPUT
Enter the airport code to be searched for

INPUT
MIA

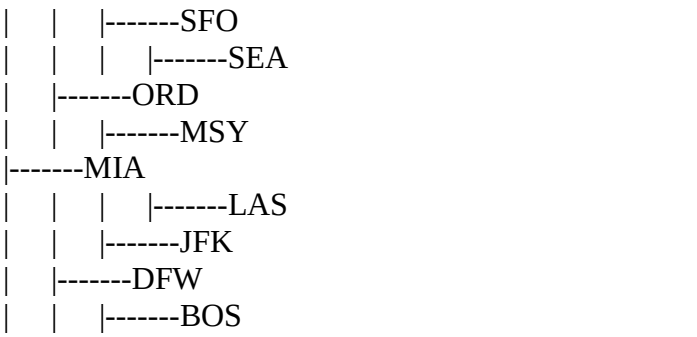
OUTPUT
MIA Miami 90

Search time is 0 milliseconds

INPUT
LAX

OUTPUT
LAX not found
Removal time is 4 milliseconds

INPUT
0

OUTPUT


INPUT
2

OUTPUT
Enter the airport code, city, check in time

INPUT
ONT ONTARIO 850

OUTPUT
Insertion time is 2 milliseconds

INPUT
4

OUTPUT
BOS Boston 120
DFW Denver 90
JFK New_York 150
LAS Las_Vegas 90
MIA Miami 90
MSY New_Orlean 60
ONT ONTARIO 850
ORD Chicago 120
SEA Seatle 90
SFO San_Francisco 120

INPUT
3

OUTPUT
Enter the to be deleted airport code

INPUT
ORD

OUTPUT
Removal time is 1 milliseconds

INPUT
0

OUTPUT

```
|     |     |-------SFO
|     |-------SEA
|     |     |     |-------ONT
|     |     |-------MSY
|-------MIA
|     |     |     |-------LAS
|     |     |-------JFK
|     |-------DFW
|     |     |-------BOS
```

AvlDriverClass.java

```
/*  Java Class: AvlDriverClass
    Author: Bryant Tunbutr
    Class: Data Structures
    Date: 12/3/13
    Description:

    I certify that the code below is my own work.

   Exception(s): N/A

 */

// to get user input
import java.util.Scanner;

// this is the driver class that runs everything
public class AvlDriverClass {

   public static void main(String[] args) {

      System.out.println("Author:  Bryant Tunbutr");

      AvlTree avlTree = new AvlTree();

      // to get file information
      avlTree.openFile();
      avlTree.readFile();

      boolean exitBool = false;

      // to make user entry continue until they enter exit value
      while (!exitBool) {

         // display user options

         System.out.print("Select an option:" + "\n"
               + "0. Debug. This displays the airport code at each node,"
               + "\n" + "but rotated sideways" + "\n"
               + "1. Search for an airport" + "\n"
               + "2. Insert a new airport " + "\n"
               + "3. Delete an airport" + "\n" + "4. List all airports "
               + "\n" + "5. Exit " + "\n");

         // use user input
         Scanner userInput = new Scanner(System.in);
         char inputChar = userInput.next().charAt(0);
```

```
// switch statements based on user commands
switch (inputChar) {

case '0':

   // debug/print starting from root, using recursion
   avlTree.debugAkaDrawTree(avlTree.rootNode, 1);

   break;

case '1':

   // instructions
   System.out.println("Enter the airport code to be searched for ");

   // make user input upper case
   String airportCodeString = userInput.next().toUpperCase();

   // user input to run find method
   avlTree.find(airportCodeString);

   // print results
   System.out.println("Search time is "
         + avlTree.getSearchTimeInt() + " milliseconds");

   break;

case '2':

   // instructions
   System.out
         .println("Enter the airport code, city, check in time");

   // store user input
   // run method, get info by using .next() which allows user to
   // enter on one line

   String airportEntryCodeString = userInput.next().toUpperCase();
   String airportCityString = userInput.next();
   String airportCheckInString = userInput.next();

   // display info
   System.out.println("Insertion time is "
         + avlTree.getInsertTimeInt() + " milliseconds");

   // run addNode method
   avlTree.addNode(airportEntryCodeString, airportCityString,
         airportCheckInString);

   break;

case '3':

   // instructions
   System.out.println("Enter the to be deleted airport code ");
```

```
            // user input to run remove method
            String airportCodeDeleteString = userInput.next().toUpperCase();

            avlTree.remove(airportCodeDeleteString);

            // display info
            System.out.println("Removal time is "
                  + avlTree.getRemoveTimeInt() + " milliseconds");

            break;

         case '4':

            // run print method
            avlTree.inOrderToPrint(avlTree.rootNode);

            break;

         case '5':

            // change boolean which causes loop to exit & program to close

            exitBool = true;
            break;
         }
      }

   }
}
```

AvlNode.java

```java
/*  Java Class: AvlNode
    Author: Bryant Tunbutr
    Class: Data Structures
    Date: 12/3/13
    Description:

    I certify that the code below is my own work.

   Exception(s): N/A

 */

public class AvlNode {

   public AvlNode leftChildNode;
   public AvlNode rightChildNode;
   public AvlNode parentNode;

   public String keyString;
   public String cityString;
   public String checkInTimeString;

   public int balanceInt;

   public AvlNode(String key, String city, String checkInTime) {
      leftChildNode = rightChildNode = parentNode = null;

      balanceInt = 0;

      keyString = key;

      cityString = city;
      checkInTimeString = checkInTime;
   }

   public String toString() {
      return keyString + " " + cityString + " " + checkInTimeString + "\n";
   }

}
```

AvlTree.java

```java
/*  Java Class: AvlTree
    Author: Bryant Tunbutr
    Class: Data Structures
    Date: 12/3/13
    Description:

    I certify that the code below is my own work.

   Exception(s): N/A

 */

import java.util.Scanner;
import java.io.*;

public class AvlTree {

   protected AvlNode rootNode;

   // add new node with string elements
   // key string is airport code
   public void addNode(String keyString, String cityString,
         String checkInTimeString) {

      // create new node using input
      AvlNode n = new AvlNode(keyString, cityString, checkInTimeString);

      // recursively insert node
      insertAVL(this.rootNode, n);
   }

   // variable to track insertion time
   // works by counting number of method calls
   private int insertTimeInt = 0;

   // compare node to be inserted with current nodes
   // using recursion
   public void insertAVL(AvlNode currentlyComparedNode,
         AvlNode toBeInsertedNode) {

      // used to track nodes visited
      setInsertTimeInt(0);

      // if currently compared node is null, then insert toBeInsertedNode
      // if null root node, toBeInsertedNode becomes the root node
      if (currentlyComparedNode == null) {
         this.rootNode = toBeInsertedNode;

      } else {

         // if compared node is smaller, go to left child node
```

```
      // use compareTo method to compare strings lexicographically
      if (toBeInsertedNode.keyString
            .compareTo(currentlyComparedNode.keyString) < 0) {

         // if encounter null node, insert it here
         if (currentlyComparedNode.leftChildNode == null) {
            currentlyComparedNode.leftChildNode = toBeInsertedNode;
            toBeInsertedNode.parentNode = currentlyComparedNode;

            // after node insertion, check the balance recursively
            rebalance(currentlyComparedNode);

         } else {
            insertAVL(currentlyComparedNode.leftChildNode,
                  toBeInsertedNode);

            // set time run time of insertion
            setInsertTimeInt(getInsertTimeInt() + 1);
         }
         // if node to be inserted is larger than current node
         // go to right child
      } else if (toBeInsertedNode.keyString
            .compareTo(currentlyComparedNode.keyString) > 0) {

         // if encounter null node, insert it here
         if (currentlyComparedNode.rightChildNode == null) {
            currentlyComparedNode.rightChildNode = toBeInsertedNode;
            toBeInsertedNode.parentNode = currentlyComparedNode;

            // after node insertion, check the balance recursively
            rebalance(currentlyComparedNode);

         } else {
            insertAVL(currentlyComparedNode.rightChildNode,
                  toBeInsertedNode);
            // set time run time of insertion
            setInsertTimeInt(getInsertTimeInt() + 1);
         }
      } else {
         // if comparison does not yield a smaller or bigger node
         // it already exists
         // nothing is done
      }
   }
}

// check balance using recursion to ensure the AVL +/-1 requirement
// at all levels

// if out of balance call rotation methods until balance occurs
public void rebalance(AvlNode currentNode) {

   // store balance for use for other methods
   setBalance(currentNode);
```

```
      int balanceInt = currentNode.balanceInt;

      // check the balance

      // balance value requires rotation
      // balance of -2 means the left side is too "heavy"
      if (balanceInt == -2) {

         // so right rotation occurs

         // left subtree is left heavy so single right rotation
         if (height(currentNode.leftChildNode.leftChildNode) >=
height(currentNode.leftChildNode.rightChildNode)) {
            currentNode = rotateRight(currentNode);

            // left subtree is right heavy so double right rotation
         } else {
            currentNode = doubleRotateLeftRight(currentNode);
         }

         // balance of 2 means the right "heavy"
         // so left rotation occurs
      } else if (balanceInt == 2) {

         // right subtree is right heavy so single left rotation
         if (height(currentNode.rightChildNode.rightChildNode) >=
height(currentNode.rightChildNode.leftChildNode)) {
            currentNode = rotateLeft(currentNode);

            // right subtree is left heavy so double left rotation
         } else {
            currentNode = doubleRotateRightLeft(currentNode);
         }
      }

      // run method until root is reached
      if (currentNode.parentNode != null) {
         rebalance(currentNode.parentNode);
      } else {
         this.rootNode = currentNode;
      }
   }

   // remove node from tree
   public void remove(String airportToBeDeletedString) {

      // first make sure node exists
      removeAVL(this.rootNode, airportToBeDeletedString);
   }

   // tracks removal time by number of method calls
   private int removeTimeInt = 0;

   // search for node
   // if found, run method to remove
```

```
// if not found, send message to user
public void removeAVL(AvlNode currentNode, String keyToBeRemovedNode) {
   setRemoveTimeInt(0);

   // if tree is empty
   if (currentNode == null) {
      System.out.println(keyToBeRemovedNode + " not found");
      return;

   } else {
      // if current node is larger than what you are searching for
      // go to left child
      if (currentNode.keyString.compareTo(keyToBeRemovedNode) > 0) {
         removeAVL(currentNode.leftChildNode, keyToBeRemovedNode);

         // increment every time method is called to track number of
         // nodes visited
         setRemoveTimeInt(getRemoveTimeInt() + 1);

         // if current node is smaller than desired node, go to right
         // child
      } else if (currentNode.keyString.compareTo(keyToBeRemovedNode) < 0) {

         removeAVL(currentNode.rightChildNode, keyToBeRemovedNode);
         setRemoveTimeInt(getRemoveTimeInt() + 1);

         // if it is equal success! and can run removal method
      } else if (currentNode.keyString.equals(keyToBeRemovedNode)) {
         removeFoundNode(currentNode);
      }
   }
}

// remove node from tree
// check for balance after removal
public void removeFoundNode(AvlNode toBeRemovedNode) {

   AvlNode tempNode;

   // if there is at least one child, remove node directly
   if (toBeRemovedNode.leftChildNode == null
         || toBeRemovedNode.rightChildNode == null) {

      // if node to be removed doesn't have a parent
      // it is the root node
      // delete by setting to null
      if (toBeRemovedNode.parentNode == null) {
         this.rootNode = null;
         toBeRemovedNode = null;
         return;
      }
      tempNode = toBeRemovedNode;

   } else {
      // if there are 2 children, need to assign a successor
```

```
            tempNode = getSuccessor(toBeRemovedNode);

            // replace key, city, and check in time of node
            toBeRemovedNode.keyString = tempNode.keyString;
            toBeRemovedNode.cityString = tempNode.cityString;
            toBeRemovedNode.checkInTimeString = tempNode.checkInTimeString;
        }

        // if there are no children

        AvlNode pNode;

        // left easy case
        if (tempNode.leftChildNode != null) {
            pNode = tempNode.leftChildNode;

            // right easy case
        } else {
            pNode = tempNode.rightChildNode;
        }

        // otherwise it is at a node with internal children

        if (pNode != null) {
            pNode.parentNode = tempNode.parentNode;
        }

        if (tempNode.parentNode == null) {
            this.rootNode = pNode;

        } else {

            // if remove is the left child of parent, update parent's left node
            if (tempNode == tempNode.parentNode.leftChildNode) {
                tempNode.parentNode.leftChildNode = pNode;

                // if remove is the right child of parent, update parent's right
                // node
            } else {
                tempNode.parentNode.rightChildNode = pNode;
            }

            // re balance recursively until root node
            rebalance(tempNode.parentNode);
        }
        // delete tempNode
        tempNode = null;
    }

    // left rotation
    public AvlNode rotateLeft(AvlNode rotationNode) {

        AvlNode rotatedTreeRoot = rotationNode.rightChildNode;
        rotatedTreeRoot.parentNode = rotationNode.parentNode;
```

```
    rotationNode.rightChildNode = rotatedTreeRoot.leftChildNode;

    if (rotationNode.rightChildNode != null) {
        rotationNode.rightChildNode.parentNode = rotationNode;
    }

    rotatedTreeRoot.leftChildNode = rotationNode;
    rotationNode.parentNode = rotatedTreeRoot;

    if (rotatedTreeRoot.parentNode != null) {
        if (rotatedTreeRoot.parentNode.rightChildNode == rotationNode) {
            rotatedTreeRoot.parentNode.rightChildNode = rotatedTreeRoot;
        } else if (rotatedTreeRoot.parentNode.leftChildNode == rotationNode) {
            rotatedTreeRoot.parentNode.leftChildNode = rotatedTreeRoot;
        }
    }

    // track balance integers
    setBalance(rotationNode);
    setBalance(rotatedTreeRoot);

    return rotatedTreeRoot;
}

// right rotation
public AvlNode rotateRight(AvlNode rotationNode) {

    AvlNode rotatedRoot = rotationNode.leftChildNode;
    rotatedRoot.parentNode = rotationNode.parentNode;

    rotationNode.leftChildNode = rotatedRoot.rightChildNode;

    if (rotationNode.leftChildNode != null) {
        rotationNode.leftChildNode.parentNode = rotationNode;
    }

    rotatedRoot.rightChildNode = rotationNode;
    rotationNode.parentNode = rotatedRoot;

    if (rotatedRoot.parentNode != null) {
        if (rotatedRoot.parentNode.rightChildNode == rotationNode) {
            rotatedRoot.parentNode.rightChildNode = rotatedRoot;
        } else if (rotatedRoot.parentNode.leftChildNode == rotationNode) {
            rotatedRoot.parentNode.leftChildNode = rotatedRoot;
        }
    }

    setBalance(rotationNode);
    setBalance(rotatedRoot);

    return rotatedRoot;
}

// double right rotation of left child node
public AvlNode doubleRotateLeftRight(AvlNode rotationNode) {
```

```
    rotationNode.leftChildNode = rotateLeft(rotationNode.leftChildNode);

    // return the doubly rotated root
    return rotateRight(rotationNode);
}

// double left rotation of right child node
public AvlNode doubleRotateRightLeft(AvlNode rotationNode) {
    rotationNode.rightChildNode = rotateRight(rotationNode.rightChildNode);

    // return the doubly rotated root
    return rotateLeft(rotationNode);
}

// find successor
public AvlNode getSuccessor(AvlNode toBeRemovedNode) {

    // nested loop to determine successor
    // if node to be removed has a right child
    if (toBeRemovedNode.rightChildNode != null) {

        // use right child as successor
        AvlNode rightNode = toBeRemovedNode.rightChildNode;

        // while the right node has a left child
        while (rightNode.leftChildNode != null) {

            // use the left child node
            rightNode = rightNode.leftChildNode;
        }

        // return node
        return rightNode;

        // if the node to be removed does not have have a right child
    } else {
        AvlNode parentNode = toBeRemovedNode.parentNode;

        // while there is a parent node & the node to be removed is equal
        // to the parent's right child node
        while (parentNode != null
              && toBeRemovedNode == parentNode.rightChildNode) {

            // the parent of the parent of the node to be removed will
            // succeed?
            toBeRemovedNode = parentNode;
            parentNode = toBeRemovedNode.parentNode;
        }
        return parentNode;
    }
}

// height calculation
private int height(AvlNode currentNode) {
```

```
   // node does not exist
   if (currentNode == null) {
      return -1;
   }

   // if no children than height is 0
   if (currentNode.leftChildNode == null
         && currentNode.rightChildNode == null) {
      return 0;

      // if only one child than height is child height + 1
      // because of AVL property of height differences w/in 1
   } else if (currentNode.leftChildNode == null) {
      return 1 + height(currentNode.rightChildNode);
   } else if (currentNode.rightChildNode == null) {
      return 1 + height(currentNode.leftChildNode);

      // helps keep overall tree balanced
   } else {
      return 1 + maximum(height(currentNode.leftChildNode),
            height(currentNode.rightChildNode));
   }
}

// integer that helps keep overall tree height balanced
private int maximum(int a, int b) {
   if (a >= b) {
      return a;
   } else {
      return b;
   }
}

private void setBalance(AvlNode currentNode) {
   currentNode.balanceInt = height(currentNode.rightChildNode)
         - height(currentNode.leftChildNode);
}

// uses in order traversal then prints
public void inOrderToPrint(AvlNode currentNode) {

   if (currentNode != null) {

      inOrderToPrint(currentNode.leftChildNode);

      System.out.print(currentNode.keyString + " "
            + currentNode.cityString + " "
            + currentNode.checkInTimeString + "\n");

      inOrderToPrint(currentNode.rightChildNode);
   }
}

public void find(String searchingForKeyString) {
```

```
   // find node then delete
   findAVL(this.rootNode, searchingForKeyString);
}

// to track number of nodes visited
private int searchTimeInt = 0;

// recursive search
public void findAVL(AvlNode currentNode, String keySearchedString) {
   setSearchTimeInt(0);

   // if end up with no match key node not present
   if (currentNode == null) {
      System.out.println(keySearchedString + " not found");
      return;

   } else {

      // if key is larger search left child
      if (currentNode.keyString.compareTo(keySearchedString) > 0) {

         findAVL(currentNode.leftChildNode, keySearchedString);
         setSearchTimeInt(getSearchTimeInt() + 1);

         // otherwise search right child
      } else if (currentNode.keyString.compareTo(keySearchedString) < 0) {

         findAVL(currentNode.rightChildNode, keySearchedString);
         setSearchTimeInt(getSearchTimeInt() + 1);

         // if equals there is a match
      } else if (currentNode.keyString.equals(keySearchedString)) {
         System.out.println(currentNode);
      }
   }
}

// use scanner
private Scanner file;

// open file
public void openFile() {

   try {
      file = new Scanner(new File("p4a.txt"));

      // catch if file not present
   } catch (Exception e) {
      System.out.println("Could not find file");
   }

}

// read file contents
public void readFile() {
```

```
   // while file has something in it
   while (file.hasNext()) {

      // store each item as separate strings
      String inputAirportCodeString = file.next();
      String inputCityString = file.next();
      String inputCheckInTimeString = file.next();

      // add these 3 strings to each node by calling method
      addNode(inputAirportCodeString, inputCityString,
            inputCheckInTimeString);
   }

}

// close file
public void closeFile() {
   file.close();
}

// draw tree but only with airport codes
// use recursion
public static void debugAkaDrawTree(AvlNode rootNode, int levelInt) {

   // if root node is null exit
   if (rootNode == null)
      return;

   // this works from printing the right child first
   debugAkaDrawTree(rootNode.rightChildNode, levelInt + 1);

   // then increasing the level
   // for every level there is an increase in lines and tabs
   if (levelInt != 0) {
      for (int i = 0; i < levelInt - 1; i++)
         System.out.print("|\t");

      // if there is no right child print the root node
      // print out only the node's airport code aka keyString
      System.out.println("|-------" + rootNode.keyString);

      // otherwise print the left child
   } else

      // print out only the node's airport code aka keyString
      System.out.println("|-------" + rootNode.keyString);

   // run method again and increase level
   debugAkaDrawTree(rootNode.leftChildNode, levelInt + 1);
}

// setters and getters for shared variables

public int getSearchTimeInt() {
```

```
        return searchTimeInt;
    }

    public void setSearchTimeInt(int searchTimeInt) {
        this.searchTimeInt = searchTimeInt;
    }

    public int getInsertTimeInt() {
        return insertTimeInt;
    }

    public void setInsertTimeInt(int insertTimeInt) {
        this.insertTimeInt = insertTimeInt;
    }

    public int getRemoveTimeInt() {
        return removeTimeInt;
    }

    public void setRemoveTimeInt(int removeTimeInt) {
        this.removeTimeInt = removeTimeInt;
    }

}
```