Project 5: (Java) The implementation of 23 trees insertion.

****
Language: Java
Project points:12 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):

12/12 on time: 10/18/2020 Sunday before midnight
+1 early submission: 10/14/2020 Wednesday before midnight
-1 for 1 day late: 10/19/2020 Monday before midnight
-2 for 2 days late: 10/20/2020 Tuesday before midnight
-12/12 : after 10/20/2020 Tuesday after midnight
-6/12: does not pass compilation
0/12: program produces no output
0/12: did not submit hard copy.

*** Follow "Project Submission Requirement" to submit your project.

Include in your hard copy:
a) cover page
b) draw an illustration of the final 23 tree
c) source code
d) treeFile
e) deBugFile

****

I. Input (args [0]): A text file contains a list of integer data items, not in any particular format

****

II. Outputs:

****

a) deBugFile (args [1]) : for all debugging prints
b) treeFile (args [2]): for printing the final 23tree.

****

III. Data structure:

****

- A 23_trees class
  - A treeNode class {
    - key1 (int)
    - key2 (int)
    - child1 (treeNode)
    - child2 (treeNode)
    - child3 (treeNode)
    - father (treeNode)

    Methods:
    - constructor (…) // with given parameters
    - printNode (Tnode, outFile)
      in the format as below:
      if Tnode is a leaf-node, print Tnode 's

(key1, key2, null, null, null, father's key1)
if Tnode is not a leaf, and child3 is not null, then print Tnode's
(key1, key2, child1's key1, child2's key1, child3's key1, father's key1);
if Tnode is not a leaf, and child3 is null, then print Tnode's
(key1, key2, child1's key1, child2's key1, null, father's key1);
}

- Root (treeNode)

Methods:
- constructor (…) // may not need it.
- initialTree (inFile, deBugFile) // get the first two data items to build the initial 2 nodes 23-tree
// see algorithm below
- swap (data1, data2) // swap the two)
- preOrder (…) // see algorithm below
- bool isLeaf (node) // returns true if all three children are null
- bool isRoot (node) // returns true if node's father is null, false otherwise
- treeInsert (…) // see algorithm below
- findSpot (…) // see algorithm below
- updateFather (fatherNode) // update fatherNode 's key1 and key2
- (int) findMinSubtree (…) // find the left most leaf of a given subtree, and return the leaf's key1
// see algorithm below
- makeNewRoot (…) // this is the case when spot is the Root, on your own, as taught in class

****
IV. Main (….)
****
Step 0: inFile ← args[0]
deBugFile ← args[1]
treeFile ← args[2]
Step 1: initialTree (inFile, deBugFile)
Step 2: data ← read one data item from inFile
Step 3: Spot ← findSpot (Root, data)
If Spot == null write "data is in the database, no need to insert" to treeFile
Repeat step 2
Else printNode (Spot, deBugFile) // with caption saying it is Spot
Step 4: newNode ← get a treeNode (data, -1, null, null, null, null)
Step 5: treeInsert (Spot, newNode)
Step 6: preOrder (deBugFile) // if printing is too much, then, call preorder every 3 insertions.
Step 7: repeat step 2 to step 6 until inFile is empty
Step 8: preOrder (treeFile)
Step 9: close all files

****
V. initialTree (inFile, deBugFile)
****

Step 1: Root ← get a treeNode (-1, -1, null, null, null, null)
Step 2: data1 ← read one data item from inFile
    data2 ← read one data item from inFile
    if data2 < data1
        swap (data1, data2)
Step 3: newNode1 ← get a treeNode (data1, -1, null, null, null, Root)
Step 4: newNode2 ← get a treeNode (data2, -1, null, null, null, Root)
Step 5: Root.child1 ← newNode1
    Root.child2← newNode2
    Root.key1 ← data2
Step 6: printNode (Root, deBugFile) // write a caption indicating this is the Root node.

****
VI. (treeNode) findSpot (Spot, data) // a recursive function
****

Step 1: if Spot is a leaf node // this is a boundary case, but should not occur.
        Write to deBugFile "You are at leaf level, you are too far down the tree, "
        and return null

    else if Spot's child1 is a leaf node
        return Spot

Step 2: if Spot is not a leaf node

    Case 1: if data == Spot's key1 or data == Spot's key2 // data is already exist
            return NULL

    Case 2: if (data < Spot's key1)
            return find Spot (Spot's child1, data)

    Case 3: if (Spot's key2 == -1 or data < Spot's key2
            return find Spot (Spot T's child2, data)

    Case 4: if (SPOT's key2 != -1 and data >= Spot's key2)
            return find Spot (Spot's child3, data)

****
VII. (int) findMinSubtree (node) // a recursive function
****

Step 1: if node is null
        return -1

    if node is a leaf
        return node's key1
    else
        return findMinSubtree (node's child1)

****
VI. UpdateFather (fatherNode) // a recursive method
****

```
Step 1: if fatherNode is null
            return
Step 2: fatherNode.key1 ← findMinSubtree (fatherNode.child2)
Step 3: fatherNode.key2 ← findMinSubtree (fatherNode.child3)
Step 4: UpdateFather (fatherNode's father)
```

****
V. treeInsert (Spot, newNode)
****

```
Case 1: If Spot has two children
      Step 1.1 : arrange the three nodes (Spot's child1, Spot's child2,  and newNode)
            in ascending order with respect to their key1 values

      Step 1.2 : Spot.child1 ← smallest of the three nodes
                 Spot.child2 ← middle  of the three nodes
                 Spot.child3 ← largest  of the three nodes

      Step 1.3: Spot.key1 ← findMinSubtree (Spot.child2)
                Spot.key2 ← findMinSubtree (Spot.child3)

      Step 1.4: if Spot is Spot's father's child2 or Spot's father's child3
                      UpdateFather (Spot.father)

Case 2:  If Spot has three children
      Step 2.1:  arrange the four nodes (Spot's child1, Spot's child2, Spot's child3, and newNode)
             in ascending order with respect to their key1 values

      Step 2.2: divide the 4 nodes into 2 groups, A and B

      Step 2.3 : Sibling ← get a treeNode(-1, -1, null, null, null, Spot.father)

      Step 2.4:  Spot.child1 ← smaller node of A
                 Spot.child2 ← larger node of  A
                 Spot.child3 ← null
                 Sibling.child1 ← smaller node of B
                 Sibling.child2 ← largest  node of B
                 Sibling.child3 ← null  // a redundant assignment

      Step 2.5:  Spot.key1 ← findMinSubtree (Spot.child2)
                 Spot.key2 ← -1
                 Sibling.key1 ← findMinSubtree (Sibling.child2)
                 Sibling.key2 ← -1
```

```
Step 2.6: if Spot is Spot.father's child2 or child3
                UpdateFather (Spot.father)

Step 2.7: if Sibling is Sibling.father's child2 or child3
              UpdateFather (Sibling.father)

 Step 2.8: if Spot is Root
                makeNewRoot (Spot, Sibling)
            else
                treeInsert(Spot.father, Sibling)
                // see if this works recursively,
                // if not, you may need to write a new method with a similar code here
              // to insert an internal node.
```