Binary Search Tree (BST)

1. Problem Description

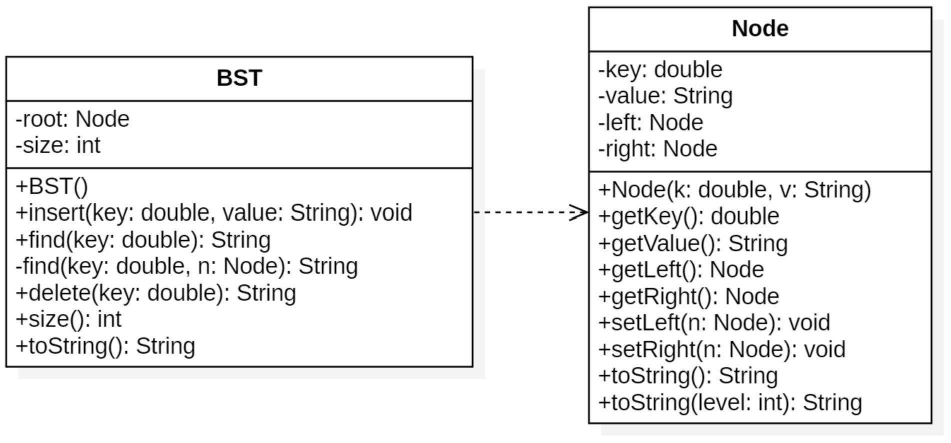
You are to develop a simple Binary Search Tree ADT and run it against a test program. Avoid the temptation of finding code online. I am aware of all the available solutions and will be looking closely at your code. You do *not* need to balance your tree.

- **Basic Level (for a maximum grade of 85%):** Develop the BST as a class that uses `double`s as keys and `String`s as values.
- **Advanced Level (for a maximum grade of 100%):** Develop the BST as a *generic* class that uses *any* objects as keys and values.

Remember that a BST is a proper Binary Tree with the following property:

Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***. We have ***key***(***u***) < ***key***(***v***) ≤ ***key***(***w***)

- Basic Level:
  The complete design of the BST class (non-generic) is shown below. You will need to create the inner class Node.



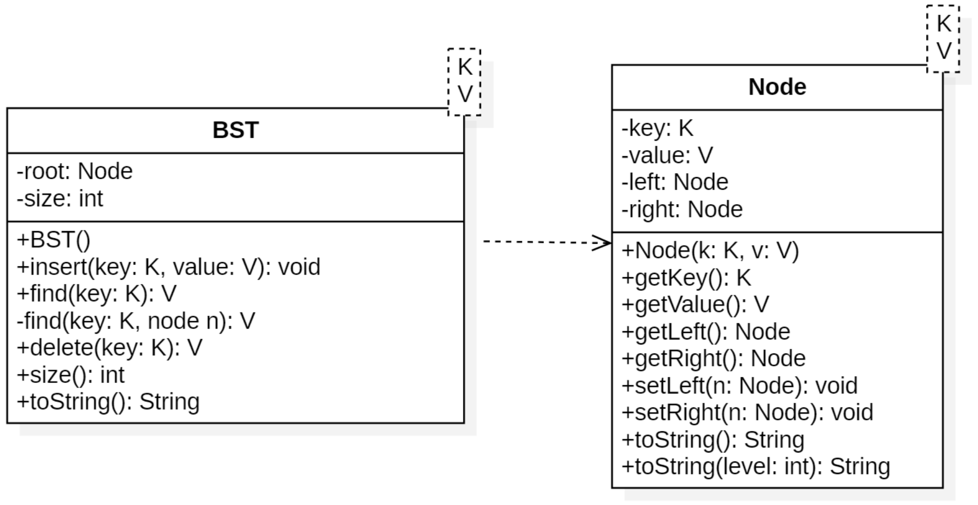
  - BST(): Construct an empty BST.
  - void insert(double key, String value): Insert the element (key, value) into the BST into the proper position.
  - String find(double key): Find the first value with the matching key. Return `null` if the key is not found. You may want to define a private helper method find(double, Node) to help with the recursive solution.
  - String delete(double key): Remove the first element with the matching key and return the value. Return `null` if the key is not found.
  - int size(): Return the number of elements in the BST.

  - String toString(): Convert the BST to a hierarchical, indented String, as shown in the output below. Notice that I have also included a String toString(int level) in the Node class to help implement this hierarchy.

- Advanced Level:
  The complete design of the generic BST class is shown below. You will need to create the inner class Node. Refer to the Basic Level description above for a definition of the methods.

  When you define your generic version of BST, use the following syntax. It ensures that the type of Key used to instantiate the class is `Comparable`. With that knowledge, you can safely call the `compareTo` method on the key.

```
public class BST<K extends Comparable<K>, V>
```



With either solution, when your implementation is complete, run it with the test driver *BSTTest.java* (provided). Review the results to make sure that your BST ADT is working correctly. The expected output is included in the file *BSTExpectedOutput.txt* and portions are shown below. Your output must include your name.

2. Notes

- You must use the test driver code found with this assignment: BSTTest.java. You should not modify BSTTest.java except to add your name.
- Turn in only your source files: BST.java and BSTTest.java. If you have created other classes as part of the solution, you need to turn those in as well.
- Make sure your class is not in a package (that is, it is in the *default* package).
- We normally must be careful when comparing double values. However, for the purposes of this assignment, you can assume that `d1 == d2` will work for matching double keys.
- For the generic solution (Advanced level), you can assume that keys are `Comparable`.
- You will need to download the file `words.txt` and put it at the Project level within Eclipse.

3. Required Main Class

BSTTest, provided

4. Required Input

Not applicable

5. Required Output

Your output should look like the following. The complete output is too long to include here, so I have removed selected lines (shown as …). The complete output is available in *BSTExpectedOutput.txt* for comparison purposes. The Large Tree is random so your tree will not match mine.

```
BST Test Program – Your Name

1) Building a Tree
========================

  Initially: null

  Insert (3.0, A):
   (k=3.00, v=A)
      null
      null
   …
  Insert (4.0, E)
   (k=3.00, v=A)
      (k=2.00, v=C)
         null
         null
      (k=5.00, v=B)
         (k=4.00, v=E)
            null
            null
         (k=6.00, v=D)
            null
            null

2) Finding Elements
========================

  b1.find(2.0): C
   …

3) Deleting Elements
========================
   …
  Delete(1.0) Left child, leaf: A
     (k=6.00, v=E)
      (k=3.00, v=H)
         (k=2.00, v=F)
            null
            null
         (k=5.00, v=C)
            (k=4.00, v=G)
               null
               null
            null
      (k=9.00, v=I)
         (k=7.00, v=J)
            null
            (k=8.00, v=B)
               null
               null
         (k=10.00, v=D)
            null
            (k=11.00, v=K)
               null
               null
   …
  Delete(5.0) Has left child, internal: C
     (k=6.00, v=E)
      (k=3.00, v=H)
         (k=2.00, v=F)
```

```
            null
            null
         (k=4.00, v=G)
            null
            null
      (k=9.00, v=I)
         (k=7.00, v=J)
            null
            null
         (k=10.00, v=D)
            null
            (k=11.00, v=K)
               null
               null
   …
  Delete(6.0) Has two children, internal: E
     (k=7.00, v=J)
      (k=3.00, v=H)
         (k=2.00, v=F)
            null
            null
         (k=4.00, v=G)
            null
            null
      (k=9.00, v=I)
         null
         (k=11.00, v=K)
            null
            null

4) A Large Tree
========================

  First 10: Size is now 10
   …

  Remaining Elements: Size is now 100000
```