CSCI3901–Assignment2-Problem2

Name: Pratheep Kumar Manoharan

Banner Id: B00837436

Files: SearchTree.java, MainUI.java, Node.java

Objective:

To implement a data structure that has the structure of an unbalanced binary search tree, but that is designed to make searches for frequently accessed items happen quickly.

Assumptions:

1. String compareTo method is used to compare the strings to identify which is bigger.
2. find method will return the depth of the element before rearranging based on the search counter value. Since it is the actual depth that find method traversed to find the element.
3. reset method will reset the searchCounters on each method call and it will not check whether the find method is called at least once or the method is called continuously twice or more.

Technical Code Flow:

Used the MainUI.java class file shared on the assignment page as the main class.

Node Class:

1. Node object has the attributes of each element of the binary search tree.
2. Node object has left node, right node, parent node, searchCounter, depth and value of each element of the binary search tree as the attributes.
3. Node class has setter and getter methods for all the attributes.
4. Node constructor method is used in the addNode method of the searchTree class.

SearchTree Class:

1. SearchTree object will have the complete details of the binary search tree.
2. rootNode and size are the important attributes of the class.

3. The *constructor* is used to initialize the attributes.

4. *printMethod* is used to print the messages to the user.

5. *add* method

    a. Gets the key from the main program (user) as the input parameter.
    b. Basic validation is done on the key.
    c. find method is called to check whether the key is already available in the tree.
    d. If the key is already available in the tree, then the error message is printed and the add method is returned.
    e. addNode method is called if the key is valid.
    f. rootNode, key, and depth are passed as parameters.
    g. All exceptions are caught, and an error message is printed to the user.

6. *addNode* method

    a. The method is used to do the recursive operations while adding a node at the bottom.
    b. Gets the currentNode, key and depth as input parameters.
    c. currentNode is validated and in case the currentNode is null then the node is set as rootNode. In the same code block, size of the tree is incremented, depth is incremented, and depth of the node is set into the object.
    d. If currentNode is not null, then the key is compared with the value of currentNode to check whether to add the node on the left side or right side.
    e. After comparing the key, the same method will be called with left node or right node, key, and incremented depth as parameters.
    f. The same set of operations from step c to e are executed till we find the last node to add the value by a recursive call to the same method.
    g. On each recursive call, the depth gets incremented and the currentNode descends in the tree.
    h. The method returns true if the element is added.

7. *find* method

    a. Gets the key as the input parameter.
    b. Basic validation is done on the key.
    c. Return 0 if the tree is empty.
    d. Loop the nodes one by one until the condition fails.
    e. If the loop is exited because of the currentNode value is null or currentNode is null, then return 0.
    f. If the loop is exited because the key is found in the tree, then return the depth of the node.
    g. For each search of the node, the search counter value of the node is incremented.

h. If the search counter of the parent node is less than the current node, then the nodes are rearranged by calling the rearrange method.
i. All exceptions are caught, and an error message is printed to the user.

8. *rearrange* method

    a. Gets the previousNode and currentNode as inputs.
    b. rootNode check is done to check whether the rootNode is to be changed.
    c. The values of the nodes are compared to find the shift is on the left side or the right side.
    d. Values of the parent of the previous node and current node are compared to check whether to add the current node on the left side or right side of the parent of the previous node.
    e. If the currentNode has any child nodes, then they are assigned to the previous node accordingly.
    f. To set the depth of all the affected nodes, the depthCalculator method is called.

9. *depthCalculator* method

    a. The method checks whether the node is null.
    b. The method is used to do the recursive operations to set the depth of all nodes.
    c. depth is incremented in each execution.
    d. depth is set into the currentNode.
    e. The same set of operations from step a to d are done for all the nodes from rootNode by a recursive call to the same method.

10. *reset* method

    a. As a first step, the method checks whether the tree is empty.
    b. If the tree has nodes, then the resetAll method is called with the rootNode as the parameter.
    c. If the tree is empty, then a message is printed to the user.
    d. All exceptions are caught, and an error message is printed to the user.

11. *resetAll* method

    a. The method is used to do the recursive operations for resetting the search counters of all nodes.
    b. Gets the currentNode as the input parameter.
    c. currentNode is validated to null check.
    d. The search counter of the currentNode is set to ZERO.
    e. The search counter of all nodes is set to ZERO by a recursive call to the same method.

12. *printTree* method

    a. At first, the method checks whether the tree is empty.
    b. If the tree is empty, then the method returns null.
    c. If the tree has nodes, then calls the formString method with the rootNode as the parameter.
    d. The value returned from the formString method is returned to the user.
    e. All exceptions are caught, and an error message is printed to users.

13. *formString* method

    a. The method is used to do prepare the output string in sorted order.
    b. Gets the currentNode as the parameter.
    c. currentNode is validated to null check.
    d. The print string of this execution is prepared.
    e. The left side nodes have lesser values compared to parent node and the right-side nodes, so the value of the nodes is added at the end of the return string while building the return string. This is done to prepare the string in sorted order.
    f. The right-side nodes have larger values compared to parent node and the left side nodes, so the value of the nodes is added at the front of the return string while building the return string. This is done to prepare the string in sorted order.
    g. The string is prepared by doing the steps b, c, d, e and f for all the nodes by a recursive call to the same method.
    h. The value is returned to the printTree method in the sorted order.

Test Scenarios:

1. All the test cases provided in the assignment are tested in the local server and the bluenose server and the results are as expected.
2. All exceptions scenarios are tested with the catch blocks.

Assignment take away:

1. I have a clear picture of how the binary search tree works internally.
2. Covered all the test cases.
3. I learned how recursion works.