```java
/* Binary Search Tree Data Type
 *
 * by Rosanna Heise & AUSCI 235
 * Dec 1, 2023 at 2:17:34 p.m.
 *
 * Contains the following public methods:
 * default constructor
 * insert(int, String) --> void
 *    adds a node to the binary search tree, maintaining order
 *    Complexity: O(log n) if tree is balanced
 * preOrderTraversal() --> String
 *    gives a pre-order traversal of the tree
 *    Complexity: O(n)
 * postOrderTraversal() --> String
 *    gives a post-order traversal of the tree
 *    Complexity: O(n)
 * inOrderTraversal() --> String
 *    gives an in-order traversal of the tree
 *    Complexity: O(n)
 * toString() --> String
 *    gives an in-order traversal of the tree
 *    Complexity: O(n)
 * findNameOf(int) --> String
 *    finds the name matching the idNum; "Not Found" if idNum is not in tree
 *    Complexity: O(log n) if tree is balanced
 * findIdNumOf(String) --> int
 *    find the id number matching the name; -1 if name is not in the tree
 *    Complexity: O(n)
 * sumMinMax() -- > int
 * finds the minimum and maximum values in a binary tree and retruns the sum of
 * of the two.
 *
 */

public class BSTree {
    //Data ============================================================
    private Node root;
    private int size;

    //Constructor =====================================================
    public BSTree(){ //start the tree empty to keep control of order
        root = null;
        size = 0;
    }

    //Getter ==========================================================
    public int getSize(){
        return size;
    }

    //Put stuff into tree =============================================
    /**
     * Places a new node into the tree, maintaining Binary Search Order.
     * idNums are assumed unique, so if an idNum repeats, an error
     * message is printed and the node is NOT added.
     * Complexity: O(log n) for a balanced tree.
     *
     * @param idNum the unique identification number of the person to
     *              be inserted
     * @param name the person's name
     */
    public void insert(int idNum, String name){
        if (root == null){
            root = new Node(idNum, name);
        }
        else{
            insert(idNum, name, location: root);
        }

        size = size + 1;
    }

    /**
     * Recursive worker to place item into the Binary Search Tree, as long
     * as location starts at root.  Note:  if location does not start at root
     * the item will be placed into the subtree starting at location and
     * part of the tree may be missed yielding an unexpected result.
     *
     * @param idNum the unique identification number of the person to
     *              be inserted
     * @param name the person's name
     * @param location the current location of the tree being checked
     */
    private void insert(int idNum, String name, Node location){
        if (idNum < location.getIdNum()){//Go left
            if (location.getLeftChild() == null){
                location.setLeftChild(new Node(idNum, name));
            }
            else{
                insert(idNum, name, location: location.getLeftChild());
            }
        }
        else if (idNum > location.getIdNum()){//Go right
            if (location.getRightChild() == null){//Here is where the node goes
                location.setRightChild(new Node(idNum, name));
            }
            else{
                insert(idNum, name, location: location.getRightChild());
            }
        }
        else{ //Same idNums
            System.out.println("ERROR same id numbers is not allowed " +
                    idNum + name);
            size = size - 1;
        }
    }

    // PreOrder Traversal ==============================================
    /**
     * Gives a String of the pre-order traversal of the tree.
     * Complexity: O(n)
     *
     * @return string which is pre-order traversal of the tree or "null" if
     *         the tree is empty
     */
    public String preOrderTraversal(){
        if (root == null){
            return "null";
        }
        return preOrderTraversal(location: root);
    }

    /**
     * Recursive worker to create the String for the pre-order traversal of
     * the subtree starting from location.  Note: if the subtree is empty it
     * returns the empty String.
     *
     * @param location current location in the tree
     * @return String which is pre-order traversal of the subtree at the
     *         given location
     */
    private String preOrderTraversal(Node location){
        if (location == null){ //finished that branch
            return "";
        }
        return location.toString() +                        //visit
```

```
                preOrderTraversal(location: location.getLeftChild()) +   //go left
                preOrderTraversal(location: location.getRightChild());   //go right
    }//preOrderTraversal recursive worker

    // PostOrder Traversal ================================================
    /**
     * Gives a String of the post-order traversal of the tree.
     * Complexity: O(n)
     *
     * @return string which is post-order traversal of the tree or "null" if
     *         the tree is empty
     */
    public String postOrderTraversal(){
        if (root == null){ //initial tree empty
            return "null";
        }
        return postOrderTraversal(location: root);
    }

    /**
     * Recursive worker to create the String for the post-order traversal of
     * the subtree starting from location.  Note: if the subtree is empty it
     * returns the empty String.
     *
     * @param location current location in the tree
     * @return String which is post-order traversal of the subtree at the
     *         given location
     */
    private String postOrderTraversal(Node location){
        if (location == null){ //finished that branch
            return "";
        }
        return postOrderTraversal(location: location.getLeftChild()) +    //go left
                postOrderTraversal(location: location.getRightChild()) +   //go right
                location.toString();                                       //visit
    }//postOrderTraversal recursive worker

    // InOrderTraversal ===================================================
    /**
     * Gives a String of the in-order traversal of the tree.
     * Complexity: O(n)
     *
     * @return string which is in-order traversal of the tree or "null" if
     *         the tree is empty
     */
    public String inOrderTraversal(){
        if (root == null){ //initial tree empty
            return "null";
        }
        return inOrderTraversal(location: root);
    }

    /**
     * Recursive worker to create the String for the in-order traversal of
     * the subtree starting from location.  Note: if the subtree is empty it
     * returns the empty String.
     *
     * @param location current location in the tree
     * @return String which is in-order traversal of the subtree at the
     *         given location
     */
    private String inOrderTraversal(Node location){
        if (location == null){ //finished that branch
            return "";
        }
        return inOrderTraversal(location: location.getLeftChild()) +      //go left
                location.toString() +                                     //visit
                inOrderTraversal(location: location.getRightChild());     //go right
    }//inOrderTraversal recursive worker

    // toString() =========================================================
    /**
     * Returns a String representation of the tree, using an in-order
     * traversal.
     * Complexity: O(n)
     *
     * @return in-order String of the tree
     */
    @Override
    public String toString(){
        return inOrderTraversal();
    }

    // Given idNum, find matching name ====================================
    /**
     * Given an id number, will return the name associated with that id
     * number. Note:  takes efficient single path through the tree, since tree
     * is sorted by id number.
     * Complexity: O(log n) for a balanced tree
     *
     * @param idNum the id number to look for
     * @return the name associated with idNum; "NOT FOUND" if idNum
     *         is not in the tree
     */
    public String findNameOf(int idNum){
        return findNameOf(idNum, location: root);
    }

    /**
     * Recursive worker to find the id number in the tree and return the
     * corresponding name.  Note: this will traverse a single branch of the
     * subtree starting at location.
     *
     * @param idNum the id number to look for
     * @param location the current location in the tree
     * @return the name associated with idNum; "NOT FOUND" if idNum
     *         is not in the subtree
     */
    private String findNameOf(int idNum, Node location){
        if (location == null){ //End of that branch, so item is not in the tree
            return "NOT FOUND";
        }
        else if (location.getIdNum() == idNum){ //Found it!!
            return location.getName();
        }
        else{ //Go either left or right depending on the id number
            if (idNum > location.getIdNum()){
                return findNameOf(idNum, location: location.getRightChild());
            }
            else{
                return findNameOf(idNum, location: location.getLeftChild());
            }
        }//else outer
    }//findNameOf recursive worker

    // Given name, find matching idNum ====================================
    /**
     * Returns the id number associated with the given name.
     * Since the tree is not sorted by names, this algorithm may search
     * through the entire tree though it does stop early if the item is
     * found.
     * Complexity: O(n)
     *
     * @param name the name to search for
     * @return the id number corresponding to the name or -1 if not found
     */
    public int findIdNumOf(String name){
```

```
    public int findIdNumOf(String name){
        return findIdNumOf(name, location: root);
    }

    /**
     * Recursive worker to find the given name in the subtree starting at
     * location and return the corresponding id number.
     *
     * @param name the name to search for
     * @param location the current location
     * @return the id number corresponding to the name or -1 if not found
     */
    private int findIdNumOf(String name, Node location){
        if (location == null){ //Reached end of a path, not there
            return -1;
        }
        else if (name.equals(anObject: location.getName())){ //Found it!
            return location.getIdNum();
        }
        else{
            //Go left
            int answerToLeft;
            answerToLeft = findIdNumOf(name, location: location.getLeftChild());

            //Only if we didn't find it on the left do we go right
            //This helps with complexity because I do not continue the
            //search unless I have to
            if (answerToLeft == -1){
                return findIdNumOf(name, location: location.getRightChild());
            }
            else{
                return answerToLeft;
            }
        }//else outer
    }//findIdNumOf recursive worker


    // Any code above here will not be evaluated for the lab exam grade
    //=========================================================================
    //=========================================================================

    //=========================================================================
    //=========================================================================
    //=========================================================================
    //=========================================================================
    // Put your code here for sumMinMax
/**
 * This method finds the minimum and maximum value in a sub tree, then
 * returns the sum of both those values combined.
 * If the size is 0 it will return -1 to signal such
 * if the size is one, it will return double the IdNum value of the tree.
 *
 * @return The sum of the min and max values in the binary search tree.
 */
public int sumMinMax(){
    //if the size of the tree is 0 then it returns -1 to signal this case
    if(size == 0){
        return -1;
    }//if
    //if the size of the tree is only one it returns the double the number, as it
    //returns the min and max value, when size is one, the root is both the mind
    //and max value
    if(size == 1){
        return 2 * root.getIdNum();
    }//if

    // Find minimum and maximum
    return sumMin(currentLoc: root) + sumMax(currentLoc: root);
}//sumMinMax


 /**
  * This method is a helper method to sumMinMax, it returns the sumMin, or the
  * min value in the sub tree.
  * @param currentLoc the current location the helper is at in the tree.
  * @return the minimum value of the binary search tree.
  */
private int sumMin(Node currentLoc) {
    //as long as the left node is not null it goes left as far as it can,
    //to find the min value
    return currentLoc.getLeftChild() == null ? currentLoc.getIdNum() :
            sumMin(currentLoc: currentLoc.getLeftChild());
}//sumMin

/**
 * This method is a helper method to sumMinMax, it returns the sumMax, more
 * precisely the Max value in a sub tree.
 * @param currentLoc the current location of the helper in the tree.
 * @return  the max value capable of being found in a binary search tree.
 */
private int sumMax(Node currentLoc) {
    //gets right child as long as right is not null, it contnues to go right
    // as far as it can to find the maxValue
    return currentLoc.getRightChild() == null ? currentLoc.getIdNum() :
            sumMax(currentLoc: currentLoc.getRightChild());
}//sumMax




}//class
```