Binary Decision Trees.
Project 7. Deadline: Saturday, 11:59pm.
100 points

In this project you will write three classes: BSTree, BSTNode and Person.
BSTNode is an inner class of BSTree.

Please comment all your classes and methods and add a brief description of what each method does. Remember that nothing is obvious.
Provide brief descriptions of what each class and method does in the form of a header comments. Utilize inline comments when necessary to explain logic.

You have a chance to gain more points in this homework. Read below.

What to submit:

1. Person.java
2. BSTree.java

- If HW2 is improved, then submit your files to HW2_Regrade
- If you implemented extra credit problem, then submit it to HW7_extraCredit folder.

Program requirements and structure

Your tree should store Person-objects into a binary search tree. PLEASE make sure that your methods have the required parameters and also make sure that they return the required values. Please follow instructions carefully.

The back end of this implementation is up to your design. However you must have the specified classes:

- BSTree.java, which will implement all the binary tree functionality.
- BSTNode (tree node) which will create nodes for your tree
- Person.java, this class will create persons that will go inside the node.

This will be a binary search tree formed with nodes that contain person-objects.

Important: At least three methods must be done recursively. We will check that. Other than that, you have total freedom.

Required Methods:
(You can find them in the provided interfaces as well)

Class Person	
public Person (String name, int key);	**name** must be a non-empty String The **key** must be an integer number between 1 and 200 (inclusive) Make sure that the parameters are in the order specified.
public void setName(String name);	Sets the non-empty name of the person.
public String getName( );	Returns the name of the person
public int getKey( );	Returns the int value of a person's key

Class BSTNode	
public BSTNode(BSTNode lChild, Person p, BSTNode rChild);	Person should be a non-null object. Throw NullPointerException for a null Person object. Please refer to the Person constructor. lChild and rChild are BSTNodes. They may or not be null depending on your code implementation. Make sure that the parameters are in the order specified.
public Person getPerson( );	Returns a Person object from inside a given node
public BSTNode getLChild( );	Returns the left child of a given node
public BSTNode getRChild( );	Returns the right child of a given node
public void setLChild(BSTNode	Sets newLNode as the left child of a node

newLChild);	
public void setRChild(BSTNode newRChild);	Sets newRNode as the right child of a node

Class BSTree	

public BSTNode getRoot() {
return root;
}
	Public getter for root (or whatever you named the root)

public void insert(String name, int key);	This method will take the two arguments and create a Person-object. If the String **name** is empty or if the **key** is not an int number in the range, the method will throw an IllegalArgumentException. Here is where you will validate the information that the Person-object constructor method will receive. If this method detects any errors then the constructor for the Person-object and the constructor for the TNode object shouldn't be called. Assume no “full” duplicates (No same key **and** string). We will not test your tree using duplicate Persons. Use a right child to insert a duplicate key.
public boolean findPerson(int key, string name);	This method looks for a person with the key given by the parameter. If the person is found the method will return "true". If the person is not found it will return false. The string name will be used in case that the key is duplicated. This will be the way to find the person. If the String name is empty or if the key is not an int number in the range, the method will throw an IllegalArgumentException

public Person [] printToArray(BSTNode root);	Prints the tree , using INORDER traversal. One Person per line. Returns an array of values in order. If the tree is empty, throw a nullpointerexception
public Person delete(int key, string name);	Deletes a person with a given key and name. If the String name is empty, if the key is not an int number in the range or if it is not found, the method will throw an IllegalArgumentException. Use any deletion method. (lazy is OK)
public int FindDepth(BSTNode root);	Returns the depth of the node root. -1 for null.
public int leafCount( );	Returns the number of nodes that are leaves in the tree.
public int levelCount(int level);	Return the number of nodes at a given level in the tree. Level 0 is the level of the root node, level 1 are the (up to 2) children of the root, and so on. Return -1 if no such level exists.

Part 2. (Optional) HW2 OR HW4 Improvements

This part is optional. If you are unsatisfied with your score or think you could get more points by completing extra credit problems you can do so.

You can submit either HW2 or HW4, but not BOTH

You have a chance to get some points back by improving your code for HW2. The points you can get back are:

1. **Compilation**. Your code **must** compile. No excuses are allowed.
2. **MasterTest** points. You can submit your code and read the submission report messages to see how many tests you have pasted.
3. **StudentTest** points. Make sure your DDL12 passes all of your own tests.
4. **BrokenCode** points. You can submit your code and read the submission report messages to see how many tests you have failed.
5. **10orMore** tests. If you did not have enough tests in your tester there is a change to make it better.
6. **Iterator test points**. Make sure you test *every* method in your iterator implementation.

You will have to submit it in a separate assignment on Vocareum called HW2_Regrade.

By a popular demand: You have a chance to get your points back for HW4.

The points you can get back are:

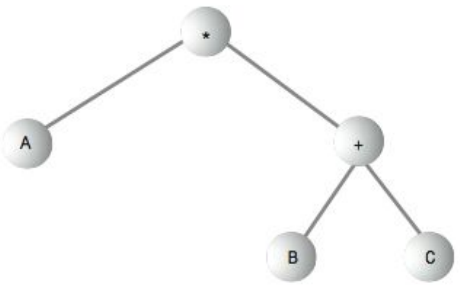
1. **5 rooms**. We will create 5 new rooms.
2. **TesterFile**

You will have to submit it in a separate assignment on Vocareum called HW4_Regrade.

Part 3. Extra Credit Problem: (15 points) Postfix binary tree.

Preambule (it is lengthy but easy):

A binary tree (not a binary search tree) can be used to represent an algebraic expression that involves the binary arithmetic operators +, –, /, and *. The root node holds an operator, and the other nodes hold either a variable name (like A, B, or C), or another operator. Each subtree is a valid algebraic expression.




```
Infix: A*(B+C)
Prefix: *A+BC
Postfix: ABC+*
```

For example, the binary tree shown above represents the algebraic expression A*(B+C). This is called *infix* notation(left, value, right); it's the notation normally used in algebra. Traversing the tree inorder will generate the correct inorder sequence A*B+C, but you'll need to insert the parentheses yourself.

Traversing the tree shown above using preorder would generate the expression

*A+BC

This is called *prefix* notation (value, left, right). One of the nice things about it is that parentheses are never required; the expression is unambiguous without them. Starting on the left, each operator is applied to the next two things in the expression. For the first

operator, *, these two things are A and +BC. For the second operator, +, the two things are B and C, so this last expression is B+C in inorder notation. Inserting that into the original expression *A+BC (preorder) gives us A*(B+C) in inorder. By using different traversals of the tree, we can transform one form of the algebraic expression into another.

For the tree above, visiting the nodes with a postorder traversal would generate the expression

ABC+*

This is called *postfix* notation (left, right, value). It means “apply the last operator in the expression, *, to the first and second things.” The first thing is A, and the second thing is BC+.

BC+ means “apply the last operator in the expression, +, to the first and second things.” The first thing is B and the second thing is C, so this gives us (B+C) in infix. Inserting this in the original expression ABC+* (postfix) gives us A*(B+C) postfix.

What needs to be implemented.

You can construct a tree like above using a postfix expression as input (I will use a text file as a command line argument). Here are the steps when we encounter an operand:

1. Make a tree with one node that holds the operand.
2. Push this tree onto the stack.

Here are the steps when we encounter an operator:

1. Pop two operand trees B and C off the stack.
2. Create a new tree A with the operator in its root.
3. Attach B as the right child of A.
4. Attach C as the left child of A.
5. Push the resulting tree back on the stack.

When you’re done evaluating the postfix string, you pop the one remaining item off the stack. Somewhat amazingly, this item is a complete tree depicting the algebraic expression. You can see the prefix and infix representations of the original postfix (and recover the postfix expression) by traversing the tree.

When the tree is generated, traverse in order and output each visited node. The output sequence is your answer (everything is one line).

Notes:

1. One expression per line/per file
2. Only +,-,*,/ operations
3. Use small letters like a,b,c for operands
4. You can use built-in Java stack
5. I would prefer you to use your own implementation, but not required
6. No, assume the intelligent user

Submit: Postfix.java