**1**



**package** Com.TSL.UtilitiesForBstSizeMethodWithQueue;

/\*\*

\* **@author** YINGJIN CUI

\* **@version** 1.0

\* since 2020-05

\*

\* GA1Driver.java: the driver program for MyIntBSTTree.java

\*/

**public** **class** GA1Driver{

/\*\*

\* main represents the entry point of this program, which adds integers to a binary search tree with nodes containing

\* integers and prints information about the tree.

\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args){

MyIntBSTTree tree = **new** MyIntBSTTree();

tree.add(8);

tree.add(30);

tree.add(28);

tree.add(12);

tree.add(1);

tree.add(6);

System.***out***.println("Number of elements: "+ tree.size());

tree.printInOrder();

}

}

**package** Com.TSL.UtilitiesForBstSizeMethodWithQueue;

/\*\*

\* **@author** Yingjin Cui

\* version 1.0

\* since 2020-05-24 16:55

\*

\* Student name: Tom Lever

\* Completion date: 07/02/21

\*

\* MyIntBSTTree represents the structure of a binary search tree whose nodes contain integers.

\*/

**import** java.util.\*;

**public** **class** MyIntBSTTree{

**private** Node root;

/\*\*

\* MyIntBSTTree() is the zero-parameter constructor for MyIntBSTTree, which sets this

\* tree's reference to a root node to null.

\*/

**public** MyIntBSTTree() {

root=**null**;

}

**public** **int** size() {

// \*\*\* Student task #1 \*\*\*

/\*

Requirements:

- Implement this method with a queue.

- This method returns the number of elements in the tree

\*\*\* Enter your code below \*\*\*

\*/

**if** (root == **null**) {

**return** 0;

}

**int** count = 0;

Queue<Node> queue = **new** LinkedList<Node>();

Node thePresentNode;

queue.add(root);

**while** (!queue.isEmpty()) {

thePresentNode = queue.remove();

count++;

**if** (thePresentNode.getLeft() != **null**) {

queue.add(thePresentNode.getLeft());

}

**if** (thePresentNode.getRight() != **null**) {

queue.add(thePresentNode.getRight());

}

}

**return** count;

}

/\*\*

\* add adds provided data to this tree.

\* **@param** data

\*/

**public** **void** add(**int** data) {

root = addHelper(root, data);

}

/\*\*

\* addHelper is a recursive method. If a provided node variable contains a null reference, addHelper sets the

\* variable to reference a node with the provided data. Else, if the provided integer data is less than the data of

\* the provided node, then addHelper passes to itself the reference to the provided node's left child and the

\* provided data. Else, addHelper passes to itself the reference to the provided node's right child and the provided

\* data. addHelper returns the node variable with an updated reference.

\*

\* **@param** node

\* **@param** data

\* **@return**

\*/

**private** Node addHelper(Node node, **int** data) {//add node helper

**if** (node == **null**){

node = **new** Node(data);

}**else** **if** (data <= node.getData()){

node.setLeft(addHelper(node.getLeft(), data));

}**else**{

node.setRight(addHelper(node.getRight(), data)); //System.out.println(data);

}

**return** node;

}

/\*\*

\* enqueueInOrderTheIntegerOf is a recursive method. If a provided reference to a node is null, this method does

\* nothing. This method passes itself the reference to the left child node of the provided reference to a node and

\* a reference to the queue in which to enqueue integers in ascending order. After this method's descendant returns,

\* this method enqueues the integer in the node referenced by the provided reference to a node. Finally, this

\* method passes itself the reference to the right child node of the provided reference to a node and a reference

\* to the queue.

\*

\* **@param** theNode

\* **@param** theQueue

\*/

**private** **void** enqueueInOrderTheIntegerOf(Node theNode, Queue<Integer> theQueue) {

**if** (theNode == **null**) {

**return**;

}

enqueueInOrderTheIntegerOf(theNode.getLeft(), theQueue);

theQueue.add(theNode.getData());

enqueueInOrderTheIntegerOf(theNode.getRight(), theQueue);

}

**public** **void** printInOrder(){

// \*\*\* Student task #2 \*\*\*

/\*

Requirements:

- Print all elements in the tree in ascending order.

- For example, if the tree contains nodes with values 5, 2, 8,

then calling printInOrder() should print as follows:

[5, 2, 8]

- You may implement this method either recursively or iteratively.

\*\*\* Enter your code below \*\*\*

\*/

**if** (root == **null**) {

System.***out***.println("[]");

**return**;

}

Queue<Integer> theQueue = **new** LinkedList<Integer>();

enqueueInOrderTheIntegerOf(root, theQueue);

String theRepresentationOfAnArrayOfTheIntegersInThisBst = "[";

**int** thePresentInteger;

**while** (!theQueue.isEmpty()) {

thePresentInteger = theQueue.remove();

theRepresentationOfAnArrayOfTheIntegersInThisBst += Integer.*toString*(thePresentInteger);

**if** (!theQueue.isEmpty()) {

theRepresentationOfAnArrayOfTheIntegersInThisBst += ", ";

}

}

theRepresentationOfAnArrayOfTheIntegersInThisBst += "]";

System.***out***.println(theRepresentationOfAnArrayOfTheIntegersInThisBst);

}

}

**package** Com.TSL.UtilitiesForBstSizeMethodWithQueue;

/\*\*

\* **@author** Yingjing Cui

\* version 1.0

\* since 2020-05-24 16:24

\*

\* Node.java: Node class

\*/

**public** **class** Node {

**private** **int** data;

**private** Node left;

**private** Node right;

/\*\*

\* Node(int data) is the one-parameter constructor for Node, which sets the value of this Node's data to provided

\* data, sets this Node's reference to a left child node to null, and sets this Node's reference to a right child

\* node to null.

\*

\* **@param** data

\*/

**public** Node(**int** data) {

**this**.data =data;

left=right=**null**;

}

/\*\*

\* getData provides this Node's data.

\*

\* **@return**

\*/

**public** **int** getData() {

**return** data;

}

/\*\*

\* getLeft provides this Node's reference to its left child node.

\* **@return**

\*/

**public** Node getLeft(){

**return** left;

}

/\*\*

\* getRight provides this Node's reference to its right child node.

\* **@return**

\*/

**public** Node getRight(){

**return** right;

}

/\*\*

\* setLeft sets this Node's reference to its left child node to a provided reference to a node.

\* **@param** node

\*/

**public** **void** setLeft(Node node){

**this**.left = node;

}

/\*\*

\* setRight sets this Node's reference to its right child node to a provided reference to a node.

\* **@param** node

\*/

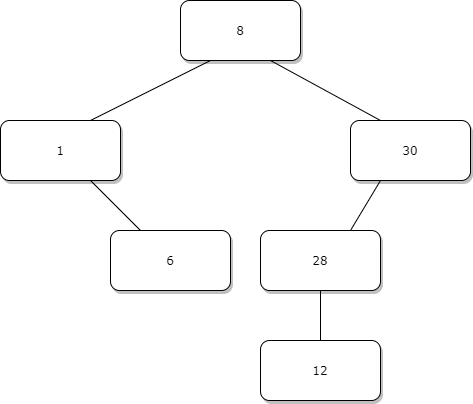
**public** **void** setRight(Node node){

**this**.right = node;

}

}

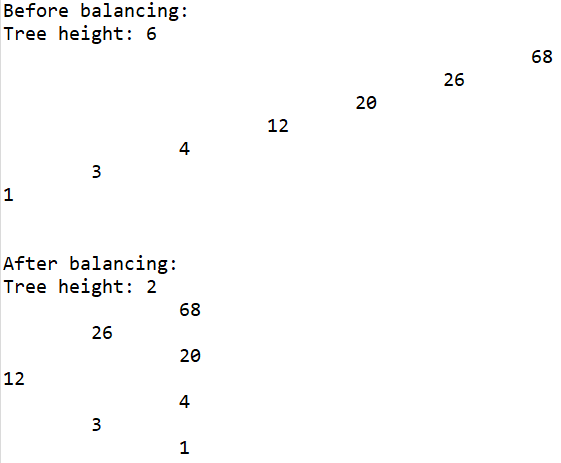
After adding integers to the binary search tree, the tree may be represented by the following diagram.



In the stack-based implementation of the tree’s size method, a count is initialized to zero. Since the tree is not empty, the count of zero is not provided as the size of the tree. An empty stack of references to nodes is created. The reference to the root node of the tree with integer 8 is pushed onto the stack. A variable thePresentNode for a reference to a present node is declared. The method loops while the stack is not empty. The reference on top of the stack (to node with 8) is stored in thePresentNode and removed from the stack (the stack becomes empty). The count is incremented (to 1). Since the reference to the left child node (with 1) of the node (with 8) referenced by thePresentNode is not null, the reference to this child is pushed onto the stack. Similarly, a reference to the right child node (with 30) is pushed onto the stack. At the end of this method’s first iteration through its loop, the count is 1 and the stack is [30, 1] from top to bottom. On the second iteration of the loop, this method pops 30 off the stack, increments count, and pushes 28 to the stack, resulting in count 2 and stack [28, 1]. On the third iteration of the loop, this method pops 28 off the stack, increments count, and pushes 12 to the stack, resulting in count 3 and stack [12, 1]. On the fourth iteration of the loop, this method pops 12 off the stack and increments count, resulting in count 4 and stack [1]. On the fifth iteration of the loop, this method pops 1 off the stack, increments count, and pushes 6 to the stack, resulting in count 5 and stack [6]. The method pops 6 off the stack and increments count, resulting in count 6 and an empty stack. Count 6 is returned.

In the stack-based implementation of the tree’s size method, generally, an active node is removed from the stack, the count is incremented, the node’s children are pushed to the stack from left to right, and the right child node, being the last node pushed to the stack, is the next node to become the active node. In the above queue-based implementation of the tree’s size method, generally, an active node is removed from the stack, the count is incremented, the node’s children are enqueued from left to right, and the left child node, being the first node enqueued, is the next node to become the active node.

**2**



**package** Com.TSL.UtilitiesForBalancingBinarySearchTree;

/\*\*

\* **@author** Yingjin Cui

\* version 1.0

\* since 2020-05-24

\*

\* GA2Driver.java: the driver program for MyIntBSTTree class

\*/

**public** **class** GA2Driver {

/\*\*

\* main is the entry point of this program, which adds nodes to a binary search tree, displays information about and

\* a representation of the tree, balances the tree, and provides information about and a representation of the tree.

\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args){

MyIntBSTTree tree=**new** MyIntBSTTree();

tree.add(1);

tree.add(3);

tree.add(4);

tree.add(12);

tree.add(20);

tree.add(26);

tree.add(68);

System.***out***.println("Before balancing:");

System.***out***.println("Tree height: "+ tree.height());

tree.display();

System.***out***.println("\n\nAfter balancing:");

tree = tree.balance();

System.***out***.println("Tree height: "+ tree.height());

tree.display();

}

}

**package** Com.TSL.UtilitiesForBalancingBinarySearchTree;

/\*\*

\* **@author** Yingjin Cui

\* version 1.0

\* since 2020-05-24 16:55

\*

\* Student name: Tom Lever

\* Completion date: 07/02/21

\*

\* MyIntBSTTree represents the structure of a binary search tree with nodes of integers.

\*/

**import** java.util.\*;

**public** **class** MyIntBSTTree{

**private** Node root;

/\*\*

\* MyIntBSTTree() is the zero-parameter constructor for MyIntBSTTree, which sets this tree's reference to a root

\* node to null.

\*/

**public** MyIntBSTTree(){

root=**null**;

}

/\*\*

\* getTheHeightOfTheTreeWithRootNode provides the height of a binary search tree with a provided root.

\*

\* **@param** theRootOfTheTree

\* **@return**

\*/

**private** **int** getTheHeightOfTheTreeWithRootNode(Node theRootOfTheTree) {

**if** (theRootOfTheTree == **null**) {

**return** -1;

}

**return**

1 + Math.*max*(

getTheHeightOfTheTreeWithRootNode(theRootOfTheTree.getLeft()),

getTheHeightOfTheTreeWithRootNode(theRootOfTheTree.getRight())

);

}

**public** **int** height() {

// \*\*\* Student task #1 \*\*\*

/\* Requirements:

The height of a binary tree is the largest number of edges in a path from the root node to a leaf node.

Essentially, it is the height of the root node. Note that if a tree has only one node, then that node

is at the same time the root node and the only leaf node, so the height of the tree is 0, similary,

the height of a tree with only two nodes is 1.

- Implement this method to return height of the tree

\*\*\* Enter your code below \*\*\*

\*/

**return** getTheHeightOfTheTreeWithRootNode(**this**.root);

}

/\*\*

\* enqueueInOrder is a recursive method. If a provided reference to a node is null, this method does nothing. This

\* method passes itself the reference to the left child node of the provided reference to a node and a reference to

\* the queue in which to enqueue nodes with integers in ascending order. After this method's descendant returns,

\* this method enqueues the node referenced by the provided reference to a node. Finally, this method passes itself

\* the reference to the right child node of the provided reference to a node and a reference to the queue.

\*

\* **@param** theNode

\* **@param** theQueue

\*/

**private** **void** enqueueInOrder(Node theNode, Queue<Node> theQueue) {

**if** (theNode == **null**) {

**return**;

}

enqueueInOrder(theNode.getLeft(), theQueue);

theQueue.add(theNode);

enqueueInOrder(theNode.getRight(), theQueue);

}

**public** Node[] inOrderArray() {

// \*\*\* Student task #2 \*\*\*

/\* Requirements:

This method get all elements in the tree and return them as sorted Node array

\*\*\* Enter your code below \*\*\*

\*/

**if** (root == **null**) {

**return** **new** Node[0];

}

Queue<Node> theQueue = **new** LinkedList<Node>();

enqueueInOrder(**this**.root, theQueue);

**return** theQueue.toArray(**new** Node[theQueue.size()]);

}

/\*\*

\* grow grows a binary tree with elements of a sorted array of nodes with integers between the first index of the

\* array and the last index of the array inclusive.

\*

\* **@param** theTreeToGrow

\* **@param** theSortedArrayOfNodes

\* **@param** theIndexOfTheFirstElement

\* **@param** theIndexOfTheLastElement

\*/

**private** **void** grow(

MyIntBSTTree theTreeToGrow,

Node[] theSortedArrayOfNodes,

**int** theIndexOfTheFirstElement,

**int** theIndexOfTheLastElement

) {

**if** (theIndexOfTheFirstElement == theIndexOfTheLastElement) {

theTreeToGrow.add(theSortedArrayOfNodes[theIndexOfTheFirstElement].getData());

}

**else** **if** ((theIndexOfTheFirstElement + 1) == theIndexOfTheLastElement) {

theTreeToGrow.add(theSortedArrayOfNodes[theIndexOfTheFirstElement].getData());

theTreeToGrow.add(theSortedArrayOfNodes[theIndexOfTheLastElement].getData());

}

**else** {

**int** theIndexOfTheMiddleElement = (theIndexOfTheFirstElement + theIndexOfTheLastElement) / 2;

theTreeToGrow.add(theSortedArrayOfNodes[theIndexOfTheMiddleElement].getData());

grow(theTreeToGrow, theSortedArrayOfNodes, theIndexOfTheFirstElement, theIndexOfTheMiddleElement - 1);

grow(theTreeToGrow, theSortedArrayOfNodes, theIndexOfTheMiddleElement + 1, theIndexOfTheLastElement);

}

}

**public** MyIntBSTTree balance() {

// \*\*\* Student task #3 \*\*\*

/\* Requirements:

This method rebuilds tree to minimize the level (height) of the tree.

To do so, you are going to rebuild a new tree from the ordered node elelemts array.

To minimize the height of the tree, for any node, you try to keep balanced numbers

of it's left and right subtrees. Please following the steps to achieve this goal:

1. select and add the middle element of the array,the middle element divides the

arry into two parts: part1-(before middle one) and part2-(after the middle one)

2. For part1 and part2, go to step 1. Repet the process until all elements are added.

For example, for an array {1,3,6,8,9,12,20}, add 8 to tree, the middle value 8 divides

the array into two parts: Part 1: {1,3,6} and Part 2: {9,12,20}, for part 1, add 3,

for part 2, add 12, repeat the process until all elements are added.

3. Return the newly builded tree.

\*\*\* Enter your code below \*\*\*

\*/

Node[] theSortedArrayOfNodes = inOrderArray();

MyIntBSTTree theBalancedBinarySearchTree = **new** MyIntBSTTree();

grow(theBalancedBinarySearchTree, theSortedArrayOfNodes, 0, theSortedArrayOfNodes.length - 1);

**return** theBalancedBinarySearchTree;

}

/\*\*

\* add adds provided data to this tree.

\* **@param** data

\*/

**public** **void** add(**int** data) {

root = addHelper(root, data);

}

/\*\*

\* addHelper is a recursive method. If a provided node variable contains a null reference, addHelper sets the

\* variable to reference a node with the provided data. Else, if the provided integer data is less than the data of

\* the provided node, then addHelper passes to itself the reference to the provided node's left child and the

\* provided data. Else, addHelper passes to itself the reference to the provided node's right child and the provided

\* data. addHelper returns the node variable with an updated reference.

\*

\* **@param** node

\* **@param** data

\* **@return**

\*/

**private** Node addHelper(Node node, **int** data) {//add node helper

**if** (node == **null**){

node = **new** Node(data);

}**else** **if** (data <= node.getData()){

node.setLeft(addHelper(node.getLeft(), data));

}**else**{

node.setRight(addHelper(node.getRight(), data));//System.out.println(data);

}

**return** node;

}

**public** **void** display(){

//print tree structure

displayHelper(root, 0);

}

/\*\*

\* displayHelper is a recursive method. If a provided reference to a node is null, this method does nothing. This

\* method passes to itself a reference to the right child node of the node at the provided reference and the index

\* of the level after a provided level. Each version of this method outputs a tab character to the standard output

\* stream for every level between the first level in the tree and this version's level inclusive. Each version of

\* this method outputs the provided node's integer. Each version of this method passes itself a reference to the

\* left child node of the node at the provided reference and the inex of the level after the provided level.

\*

\* **@param** t

\* **@param** level

\*/

**private** **void** displayHelper(Node t, **int** level){

**if**(t==**null**) **return** ;

displayHelper(t.getRight(), level + 1);

**for**(**int** k = 0; k < level; k++)

System.***out***.print("\t");

System.***out***.println(t.getData());

displayHelper(t.getLeft(), level + 1); //recurse left

}

/\*\*

\* size provides the number of nodes in this tree.

\*

\* **@return**

\*/

**public** **int** size(){

**return** sizeHelper(root);

}

/\*\*

\* sizeHelper is a recursive method. If a provided reference to a node is null, this method provides a number of

\* nodes of 0. Otherwise, this method provides the sum of 1, the output of itself when passed a reference to the

\* left child node of the node referenced by the provided reference, and the output of itself when passed a

\* reference to the right child node referenced by the provided reference.

\*

\* **@param** node

\* **@return**

\*/

**private** **int** sizeHelper(Node node){

**if**(node==**null**) **return** 0;

**else** **return** 1+sizeHelper(node.getLeft())+sizeHelper(node.getRight());

}

}

**package** Com.TSL.UtilitiesForBalancingBinarySearchTree;

/\*\*

\* **@author** Yingjing Cui

\* version 1.0

\* since 2020-05-24 16:24

\*

\* Node.java: Node class

\*/

**public** **class** Node{

**private** **int** data;

**private** Node left;

**private** Node right;

/\*\*

\* Node(int data) is the one-parameter constructor for Node, which sets the value of this Node's data to provided

\* data, sets this Node's reference to a left child node to null, and sets this Node's reference to a right child

\* node to null.

\*

\* **@param** data

\*/

**public** Node(**int** data) {

**this**.data =data;

left=right=**null**;

}

/\*\*

\* getData provides this Node's data.

\*

\* **@return**

\*/

**public** **int** getData() {

**return** data;

}

/\*\*

\* getLeft provides this Node's reference to its left child node.

\* **@return**

\*/

**public** Node getLeft(){

**return** left;

}

/\*\*

\* getRight provides this Node's reference to its right child node.

\* **@return**

\*/

**public** Node getRight(){

**return** right;

}

/\*\*

\* setLeft sets this Node's reference to its left child node to a provided reference to a node.

\* **@param** node

\*/

**public** **void** setLeft(Node node){

**this**.left = node;

}

/\*\*

\* setRight sets this Node's reference to its right child node to a provided reference to a node.

\* **@param** node

\*/

**public** **void** setRight(Node node){

**this**.right = node;

}

/\*\*

\* toString provides provided data as a string.

\*

\* **@param** data

\* **@return**

\*/

**public** String toString(**int** data) {

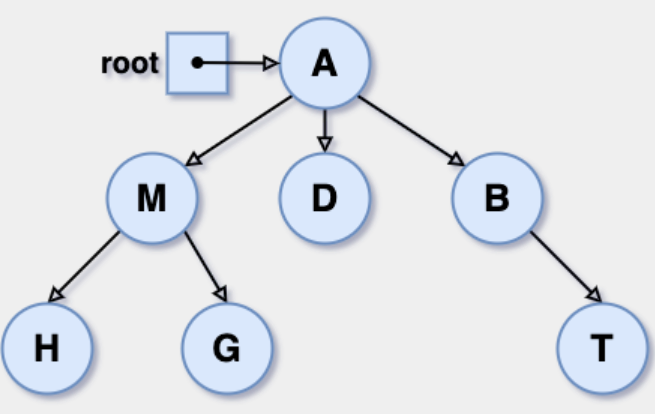
**return** Integer.*toString*(data);

}

}

**3**

Given the following binary tree and left-to-right in-layer movement, the breadth-first traversal is A, M, D, B, H, G, T.



Given the following binary search tree and an initial walk from M to D, the depth-first traversal and the pre-order traversal is M, D, A, G, T, Z.

Given the following binary search tree and an initial walk from M to D, the post-order traversal is A, G, D, Z, T, M.

