COMP09044

Algorithms and collections coursework

B00239486

Amending and experimenting with the BinarySearchTree class

The following project parts will each be split into three sections to give the report some structure, the first section on each of the parts will be “Intended changes” this will include the description of what is to be achieved in this part and then a list in which I will pre-plan the stages I will take to make these changes to the project specifications. The second section will be “Changes made” which will be split into two parts if needed which will be “creations” and “edits” this will include anything that is added as a whole to the code as in a whole method and also any edits I make to the existing code such as including the new methods where needed. The third section will be the “Testing” phase this will include a description of which tests can be performed on the code from the application window and any tests done manually.

**Part A – Extended Binary Search Trees**

Intended changes:

The first part of the course work is to amend the BinarySearchTree<E> class discussed in the lecture so that it represents an extended binary search tree. Whenever a new value is inserted into the tree the result is that an external node is replaced by an internal node containing that value and two external nodes (where the external nodes are the left and right children of the new internal node).

Also ensure that the Entry<E> class has a constructor that will allow you to create an internal node and another constructor that will allow you to create an external node.

Add a method to the BinarySearchTree<E> class to allow the elements to be displayed in the order they would be visited in a breadth-first traversal of the tree (it is easy to work out the tree structure from this). You can think about whether you want a method that displays the tree directly or whether to override the toString() method so that it returns a String representation of the tree that can be displayed. You may find it helpful if the display includes the external nodes. See pages 391-392 of the book for a description of an iterative implementation of a breadth-first traversal using a queue.

Add a method to the BinarySearchTree<E> class that returns the height of the tree (include the external nodes when calculating this).

Splitting this into pre-planned tasks to complete I intend to:

* Add the isExternal method
* Add the makeExternal method
* Add the makeInternal method
* Edit the current functions so that they use

new methods instead of searching for null.

* Create a constructor for external/internal creation.
* Create a method to display height of tree (including externals)
* Make a method that displays the tree in a Breadth-first traversal (i.e. 45 24 47 50)

Actual creations performed:

- Created a method in Entry<E> class for checking if the node passed in is external or not by comparing if the element, left child and right child of the node are all null if so it returns true as the node is external, if not it returns false signifying internal.

/\*\* returns true if this entry is an external node and false otherwise. if element, left and right child are all null is external. \*/

**public** **boolean** isExternal() {

**boolean** elementNull = (**this**.element == **null**);

**boolean** leftNull = (**this**.left == **null**);

**boolean** rightNull = (**this**.right == **null**);

**boolean** isExternal = (leftNull && rightNull && elementNull);

**return** isExternal;

}

- Created a new method again in Entry<E> class for making the passed in element an external node, it does this by setting the element, left child and right child of the passed element null. Used when deleting an internal node that has no internal nodes as children.

/\*\* converts this internal node to an external node and returns the element that the internal node contained (used when deleting an internal node that has no internal nodes as children) \*/

**public** E makeExternal() {

E value = **this**.element;

**this**.left = **null**;

**this**.right = **null**;

**this**.element = **null**;

**return** value;

}

- Created a third method in Entry<E> class for making the passed in element an internal node by making the passed in element equal element (**this**.element = element) and making the left and right children new external nodes. Used when inserting an element in to the tree.

/\*\* converts this external node to an internal node containing the given element and adds two new external nodes as the left and right children of the node (used when inserting an element in to the tree)

\*/

**public** **void** makeInternal (E element) {

**this**.element = element;

**this**.left = **new** Entry<E>(**null**, **this**);

**this**.right = **new** Entry<E>(**null**, **this**);

}

- Created a constructor method again in the Entry<E> class to give the choice of either makeExternal or makeInternal by choice of a input Boolean.

**public** Entry(E element, Entry<E> parent, **boolean** makeInternal) {

**this**.parent = parent;

**if** (makeInternal)

makeInternal(element);

**else**

makeExternal();

}

- Created a new method called height() it begins by checking if the height of the passed in tree is empty if so it just returns 0, if not it will then be passed to a second method findHeight() which will then find the height of the tree.

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

**int** Height;

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

Height = 0;

**else**

Height = findHeight(root);

System.*out*.println("Tree Height (including externals): " + Height);

**return** Height;

}

- Created the findHeight() method that the height method will pass into which will then find the height of the passed tree if it is not empty this includes the external nodes. If the entered node is external just return 0, if not find the max height of the tree from the longest branch + 1 for external nodes.

**public** **int** findHeight(Entry<E> entry) {

**if** (entry.isExternal())

**return** 0;

**else**

**return** 1 + Math.*max*(findHeight(entry.left),findHeight(entry.right));

}

- Created a breadth first traversal method to display the tree in an easily recognized way (i.e. 25 21 40 35 50). This method is heavily based of the supplied algorithm example on page 392 of the William J Collins book, I have included comments to the right of the code to show how it follows the provided algorithm.

William J Collins algorithm:

if (t is not empty)

{

queue.enqueue (t);

while (queue is not empty)

{

tree = queue.dequeue();

process trees root;

if (leftTree (tree) is not empty)

queue.enqueue (leftTree (tree));

if (rightTree (tree) is not empty)

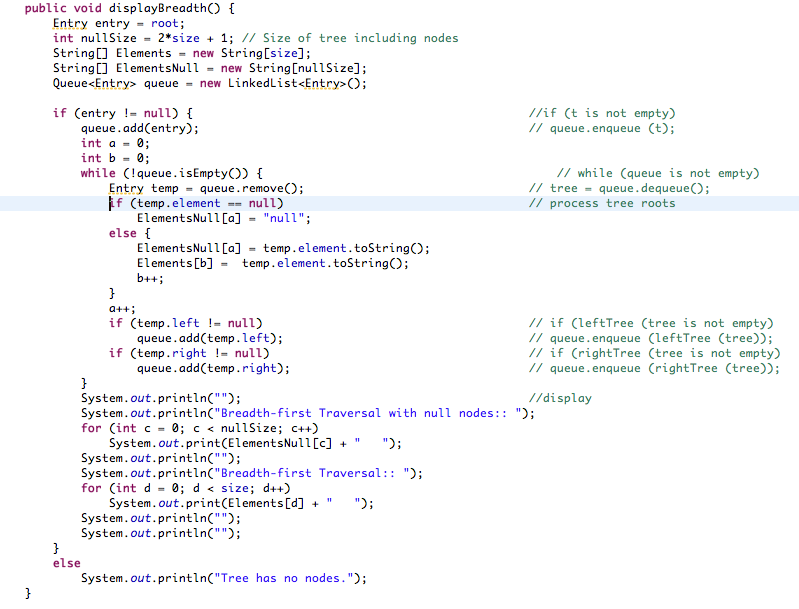
queue.enqueue (rightTree (tree));

} // while

} // if t not empty

} // breadthFirst traversal

(Code is too large and unformatted to display here, so a picture of said code has been included)



Actual edits performed:

- Added to the add method so that it would now incorporate the newly created methods, these changes include are, checking if a child is now external using the isExtrernal method and then if so changing it to an Internal node using makeInternal.

// Handle left

**if** (temp.left.isExternal()){ // if external

temp.left.makeInternal (element); // make internal

size++;

**return** **true**;

}

**else**

temp = temp.left;

// Handle right

**else** **if** (temp.right.isExternal()) { // if external

temp.right.makeInternal(element); // make internal

size++;

**return** **true**;

}

**else**

temp = temp.right;

- Added to the getEntry() method, changed it so that it would now check e against the isExternal method instead of just comparing to null.

**while** (!e.isExternal())

- Added to the deleteEntry() method, changed it so that during the child check phase it would now use isExternal instead of comparing to null for both children.

**if** (p.left.isExternal() && p.right.isExternal())

- Added to the successor method, changed it so it would use isExternal to return null when e is external, also adjusted it so that instead of testing if a child a is null it will now specifically check if a child’s element is null in both child checks.

**if** (e == **null** || e.isExternal())

**else** **if** (e.right.element != **null**)

**while** (p.left.element != **null**)

- Added to the TreeIterator class, changed it so now the while loop continues to check until the node is classed as external by the isExternal method instead of just for when it is null.

**while** (next.left.isExternal())

Testing section:

The main included with this part of the project includes a number of options the user can choose from to test the application (with the help of the “input” class form a previous module) here I shall run through said options and then give a description of what each predefined test shows.

Free test options:

* The first option available to the user is to add a predefined number to the tree (45) this is just to show that basic pre defined addition is possible
* The second option is to check for the number 45 within the tree, this is a basic test to show that the contains method is still operational
* Option three allows the user to quickly fill the tree with a set of numbers so that a more filled out tree can be quickly tested the numbers are 32, 25, 47, 42 and 50.
* Option four lets the user input their own number into the tree to allow dynamic testing of any number combinations
* Option five allows the user to remove any number form the tree, including numbers added by the user or the system, if a number does not exist in the tree it will simple let the user choose another menu option.
* Option six will display the currents trees height and size.
* Option 7 will display the tree in a breadth-first traversal form and will include a tree with null values and one without.

Pre-defined options, these options create their own trees as not to interfere with other tests:

* Option 8 will run a predefined test that will:
  + Create a new tree (tree1)
  + Display the current trees size, height and a breadth traversal
  + Add the numbers: 37, 40, 32, 23 and 6
  + Display the current trees size, height and a breadth traversal a second time
  + Tests if the tree contains a value that is in the tree (32, should return true)
  + Tests if the tree contains a value that is not in the tree (20, should return false)
  + Remove the number 37
  + Remove the number 6
  + Display the current trees size, height and a breadth traversal a third time
* Option 9 will run a larger predefined test that will:
  + Create a new tree (tree2)
  + Add the numbers: 31, 25, 47, 42, 50
  + Display the current trees size, height and a breadth traversal
  + Add the number: 37, 40, 32, 23 and 6
  + Display the current trees size, height and a breadth traversal a second time
  + Tests if the tree contains a value that is in the tree (25, should return true)
  + Tests if the tree contains a value that is not in the tree (43, should return false)
  + Remove the number 37
  + Remove the number 6
  + Display the current trees size, height and a breadth traversal a third time

Output for Test on tree1 (option 8):

0: quit

1: Add 45

2: Check for number 45

3: Add numbers: 32,25,47,42,50

4: Add own number.

5: Remove own number.

6: See size and height of tree

7: Display tree in breadthfirst

8: Run predefined test. (5 numbers, Creates its own tree)

9: Run predefined test. (10 numbers, Creates its own tree)

Input option: 8

# Display tree: #

- Tree has no nodes.

- Tree size: 0

- Tree Height (including externals): 0

# Add: 37, 40, 32, 23 and 6 then display #

Breadth-first Traversal with null nodes::

37 32 40 23 null null null 6 null null null

Breadth-first Traversal::

37 32 40 23 6

- Tree size: 5

- Tree Height (including externals): 4

# Does tree contain 32? #

- true

# Does tree contain 20? #

- false

# Remove 37 and 6 then display.#

Breadth-first Traversal with null nodes::

32 23 6 null null null null

Breadth-first Traversal::

32 23 6

- Tree size: 3

- Tree Height (including externals): 2

Input option:

END OF TEST

This test has therefore been passed as all information is correct this test showed that the tree could be displayed, height measured, items added, items removed, use of isExternal makeExternal and makeInternal and that features such as the contains method still worked.

Output for Test on tree2 (option 9):

0: quit

1: Add 45

2: Check for number 45

3: Add numbers: 32,25,47,42,50

4: Add own number.

5: Remove own number.

6: See size and height of tree

7: Display tree in breadthfirst

8: Run predefined test. (5 numbers, Creates its own tree)

9: Run predefined test. (10 numbers, Creates its own tree)

Input option: 9

# Display tree: #

-

Breadth-first Traversal with null nodes::

31 25 47 null null 42 50 null null null null

Breadth-first Traversal::

31 25 47 42 50

- Tree size: 5

- Tree Height (including externals): 3

# Add: 37, 40, 32, 23 and 6 then display #

Breadth-first Traversal with null nodes::

31 25 47 23 null 42 50 6 null 37 null null null null null 32 40 null null null null

Breadth-first Traversal::

31 25 47 23 42 50 6 37 32 40

- Tree size: 10

- Tree Height (including externals): 5

# Does tree contain 25? #

- true

# Does tree contain 43? #

- false

# Remove 37 and 6 then display. #

Breadth-first Traversal with null nodes::

31 25 47 23 null 42 50 null null 32 null null null null null null null

Breadth-first Traversal::

31 25 47 23 42 50 32 null

- Tree size: 8

- Tree Height (including externals): 4

Input option:

END OF TEST

This test has therefore been passed as all information is correct. This test backs up the test performed on tree 1 by running the same forms of test but on a larger scale.

**Part B – Insertion at the Root**

Intended changes:

Modify your solution to Part A to provide a constructor for the BinarySearchTree<E> that takes a boolean parameter which, if true, means that the add() method at the root and if false will insert in an external node. Do not provide method to change the insertion method once a tree has been created. This means that when a BinarySearchTree is created it will either always insert at the root or always in a leaf. You may also retain the default constructor which takes no parameter you should ensure that by default the tree will insert using the standard algorithm (not at the root). Note that, whether insertion is at the root or in an external (leaf) node, the tree should still be an extended binary search tree that does not allow duplicate values.

Splitting this into pre-planned tasks to complete I intend to:

* Create and initialise “addRoot”
* Create a second tree constructer that takes an input to allow choosing of tree at creation as specified in the coursework.
* Create a new add method to decide which insertion action to perform
* Make the old add method the “addLeaf” add option
* Create rotation methods
* Create the addRoot method

Actual creations performed:

- Added addRoot to the initialization section of BinarySearchTree and then initialized it within the BinarySearchTree and set it to false on default as specified in the coursework.

**private** **boolean** addRoot;

addRoot = **false**;

- Created a second BinarySearchTree constructor to change between adding at root or adding at leaf upon construction so it cannot be changed later as specified.

**public** BinarySearchTree(**boolean** addRoot) {

root = **null**;

size = 0;

**this**.addRoot = addRoot;

}

- Created a second add method to decide which insertion action to perform, this also watches for duplicate elements being input to protect both trees.

**public** **boolean** add(E element) {

**if** (contains(element))

**return** **false**;

**if** (addRoot) {

root = addRoot(root, element);

**return** **true**;

}

**else**

**return** addLeaf(element);

}

- Changed the original add method to now be the addLeaf add method, nothing within the addition code was changed merely its title

**private** **boolean** addLeaf(E element)

- Created a rotateLeft method based off of the rotate left function on page 433 of the William.j.Collins book and the notes provided, it is used to rearrange the tree for root additions.

**private** Entry<E> rotateLeft(Entry<E> p) {

Entry<E> r = p.right;

p.right = r.left;

r.left = p;

r.parent = **null**;

r.left.parent = r;

p.parent = r;

**return** r;

}

- Created an accompanying rotateRight method which is much the same as the rotateLeft method but with the values reversed.

**private** Entry<E> rotateRight(Entry<E> p) {

Entry<E> r = p.left;

p.left = r.right;

r.right = p;

r.parent = **null**;

r.right.parent = r;

p.parent = r;

**return** r;

}

- Added a method so that new elements can be added to the root, it does this by monitoring if elements inserted are higher or lower than the current elements and rotating the tree appropriately so that insert at the root will work.

**private** Entry<E> addRoot(Entry<E> root, E element) {

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

{

**if**(element == **null**)

**throw** **new** NullPointerException();

root = **new** Entry<E>(element, **null**, **true**);

size++;

**return** root;

}

**if**(root.isExternal()) {

root = **new** Entry<E>(element, **null**, **true**);

size++;

**return** root;

}

**else** {

**int** comp = ((Comparable) element).compareTo(root.element);

**if**(comp < 0){

root.left = addRoot(root.left, element);

root = rotateRight(root);

**return** root;

}

**else** {

root.right = addRoot(root.right, element);

root = rotateLeft(root);

**return** root;

}

}

}

Testing section:

Again this main allows the user to input options to play with the trees (with the help of the “input” class form a previous module) here I shall run through said options and then give a description of what each predefined test shows.

However in this test two separate trees are created one that will be an addRoot tree and one that will be an addLeaf tree, all options chose will effect both trees so that when displayed it is clear the differences between the two addition methods.

Free test options:

* The first option available to the user is to add a predefined number to both of the trees (45) this is just to show that basic pre defined addition is possible
* The second option is to check for the number 45 within both of the trees, this is a basic test to show that the contains method is still operational
* Option three allows the user to quickly fill both trees with a set of numbers so that a more filled out tree can be quickly tested the numbers are 31, 25, 47, 42 and 50.
* Option four lets the user input their own number into both trees to allow dynamic testing of any number combinations
* Option five allows the user to remove any number form both trees, including numbers added by the user or the system, if a number does not exist in the tree it will simple let the user choose another menu option.
* Option six will display the currents height and size for both of the trees.
* Option 7 will display a breadth-first traversal display for both trees and will include a tree display with null values and one without.

Pre-defined options, these options create their own trees as not to interfere with other tests:

* Option 8 wil run predefined test with adding to leaf tree (Proof of part A still working):
  + Create a new tree (treeLeaf)
  + Display the current trees size, height and a breadth traversal
  + Add the numbers: 37, 40, 32, 23 and 6
  + Display the current trees size, height and a breadth traversal a second time
  + Tests if the tree contains a value that is in the tree (32, should return true)
  + Tests if the tree contains a value that is not in the tree (20, should return false)
  + Remove the number 37
  + Remove the number 50
  + Display the current trees size, height and a breadth traversal a third time
* Option 9 will run the same predefined test as option 8 but on a addRoot tree:
  + Create a new tree (treeRoot)
  + Display the current trees size, height and a breadth traversal
  + Add the numbers: 37, 40, 32, 23 and 6
  + Display the current trees size, height and a breadth traversal a second time
  + Tests if the tree contains a value that is in the tree (32, should return true)
  + Tests if the tree contains a value that is not in the tree (20, should return false)
  + Remove the number 37
  + Remove the number 50
  + Display the current trees size, height and a breadth traversal a third time
* Option 10 will run a separate class (moved to lessen clutter) which will go through a step by step process of adding 5 numbers to each tree and displaying them both at the same time then removing two numbers.
  + Create a two new trees, treeLeaf and treeRoot
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and add 10 to both trees
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and add 37 to both trees
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and add 16 to both trees
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and add 50 to both trees
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and add 26 to both trees
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and remove 50 from both trees
  + Display height, size and breadth traversal of both trees.
  + Request to proceed and remove 37 from both trees
  + Display height, size and breadth traversal of both trees.
  + Return to normal menu

Output for Test on treeLeaf (option 8):

Add at leaf tree created

Add at root tree created

0: quit

1: Add 45 to both trees

2: Check for number 45 in both trees

3: Add numbers: 31,25,47,42,50 to both trees

4: Add own number to both trees.

5: Remove own number form both trees.

6: See size and height of both trees

7: Display both trees in breadthfirst

8: Run predefined test with adding to leaf tree (Proof af part A still working)

9: Run same predefined test with adding at root tree (Proof inserting at the root works)

10: Watch as step by step numbers are add to both trees and removed (5 numbers, creates seperate tree.)

Input option: 8

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Display tree: #

- Tree has no nodes.

- Tree size: 0

- Tree Height (including externals): 0

# Add: 37, 40, 32, 23 and 6 then display #

Breadth-first Traversal with null nodes::

37 32 40 23 null null 50 6 null null null null null

Breadth-first Traversal::

37 32 40 23 50 6

- Tree size: 6

- Tree Height (including externals): 4

# Does tree contain 32? #

- true

# Does tree contain 20? #

- false

# Remove 37 and 50 then display.#

Breadth-first Traversal with null nodes::

40 32 null 23 null 6 null null null

Breadth-first Traversal::

40 32 23 6

- Tree size: 4

- Tree Height (including externals): 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Input option:

END OF TEST

This test shows that all the original functions from Part A still work correctly with a normal add at root tree.

Output for Test on treeRoot (option 9):

Add at leaf tree created

Add at root tree created

0: quit

1: Add 45 to both trees

2: Check for number 45 in both trees

3: Add numbers: 31,25,47,42,50 to both trees

4: Add own number to both trees.

5: Remove own number form both trees.

6: See size and height of both trees

7: Display both trees in breadthfirst

8: Run predefined test with adding to leaf tree (Proof af part A still working)

9: Run same predefined test with adding at root tree (Proof inserting at the root works)

10: Watch as step by step numbers are add to both trees and removed (5 numbers, creates seperate tree.)

Input option: 9

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Display tree: #

- Tree has no nodes.

- Tree size: 0

- Tree Height (including externals): 0

# Add: 37, 40, 32, 23 and 6 then display #

Breadth-first Traversal with null nodes::

6 null 23 null 50 32 null null 40 37 null null null

Breadth-first Traversal::

6 23 50 32 40 37

- Tree size: 6

- Tree Height (including externals): 6

# Does tree contain 32? #

- true

# Does tree contain 20? #

- false

# Remove 37 and 50 then display.#

Breadth-first Traversal with null nodes::

6 null 23 null 32 null null null null

Breadth-first Traversal::

6 23 32 null

- Tree size: 4

- Tree Height (including externals): 3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Input option:

END OF TEST

This test shows that all the original functions from Part A will also work on the new addition type, addition at root.

Output for Test both treeRoot and treeLeaf (option 10):

Add at leaf tree created

Add at root tree created

0: quit

1: Add 45 to both trees

2: Check for number 45 in both trees

3: Add numbers: 31,25,47,42,50 to both trees

4: Add own number to both trees.

5: Remove own number form both trees.

6: See size and height of both trees

7: Display both trees in breadthfirst

8: Run predefined test with adding to leaf tree (Proof af part A still working)

9: Run same predefined test with adding at root tree (Proof inserting at the root works)

10: Watch as step by step numbers are add to both trees and removed (5 numbers, creates seperate tree.)

Input option: 10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 0

Tree Height (including externals): 0

Tree has no nodes.

\*\*TreeRoot\*\*

Tree size: 0

Tree Height (including externals): 0

Tree has no nodes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to ADD 10 to both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 1

Tree Height (including externals): 1

Breadth-first Traversal with null nodes::

10 null null

Breadth-first Traversal::

10

\*\*TreeRoot\*\*

Tree size: 1

Tree Height (including externals): 1

Breadth-first Traversal with null nodes::

10 null null

Breadth-first Traversal::

10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to ADD 37 to both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 2

Tree Height (including externals): 2

Breadth-first Traversal with null nodes::

10 null 37 null null

Breadth-first Traversal::

10 37

\*\*TreeRoot\*\*

Tree size: 2

Tree Height (including externals): 2

Breadth-first Traversal with null nodes::

37 10 null null null

Breadth-first Traversal::

37 10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to ADD 16 to both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 3

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

10 null 37 16 null null null

Breadth-first Traversal::

10 37 16

\*\*TreeRoot\*\*

Tree size: 3

Tree Height (including externals): 2

Breadth-first Traversal with null nodes::

16 10 37 null null null null

Breadth-first Traversal::

16 10 37

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to ADD 50 to both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 4

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

10 null 37 16 50 null null null null

Breadth-first Traversal::

10 37 16 50

\*\*TreeRoot\*\*

Tree size: 4

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

50 16 null 10 37 null null null null

Breadth-first Traversal::

50 16 10 37

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to ADD 26 to both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 5

Tree Height (including externals): 4

Breadth-first Traversal with null nodes::

10 null 37 16 50 null 26 null null null null

Breadth-first Traversal::

10 37 16 50 26

\*\*TreeRoot\*\*

Tree size: 5

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

26 16 50 10 null 37 null null null null null

Breadth-first Traversal::

26 16 50 10 37

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to REMOVE 50 from both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 4

Tree Height (including externals): 4

Breadth-first Traversal with null nodes::

10 null 37 16 null null 26 null null

Breadth-first Traversal::

10 37 16 26

\*\*TreeRoot\*\*

Tree size: 4

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

26 16 37 10 null null null null null

Breadth-first Traversal::

26 16 37 10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

About to REMOVE 37 from both trees.

Proceed? (1 yes, 0 no): 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*TreeLeaf\*\*

Tree size: 3

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

10 null 16 null 26 null null

Breadth-first Traversal::

10 16 26

\*\*TreeRoot\*\*

Tree size: 3

Tree Height (including externals): 3

Breadth-first Traversal with null nodes::

26 16 null 10 null null null

Breadth-first Traversal::

26 16 10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*RETURNING TO NORMAL MENU\*\*\*\*\*

0: quit

1: Add 45 to both trees

2: Check for number 45 in both trees

3: Add numbers: 31,25,47,42,50 to both trees

4: Add own number to both trees.

5: Remove own number form both trees.

6: See size and height of both trees

7: Display both trees in breadthfirst

8: Run predefined test with adding to leaf tree

9: Run same predefined test with adding at root tree

10: Watch as step by step numbers are add to both trees (5 numbers)

Input option:

END OF TEST

This test allows the user to see how both trees perform through additions and removals at the same time giving a better perspective on the trees.

**Part C – Serialization**

Intended changes:

For this final part you are asked to have your BinarySearchTree<E> class declare that it implements Serializable, so that the state of a tree can be serialized by passing a tree to the writeObject() method of an ObjectOutputStream, and deserialized by calling the readObject() method of an ObjectInputStream and assigning the Object returned to a BinarySearchTree variable.

Splitting this into pre-planned tasks to complete I intend to:

* Implement serialization into BinarySearchTree
* Use the serialization from main for testing

Actual creations performed:

- In BinarySearchTree the only action to be performed was to declare in BinarySearchTree and the Entry class implements Serializable.

**public** **class** BinarySearchTree<E> **extends** AbstractSet<E> **implements** Serializable

**private** **static** **class** Entry<E> **implements** Serializable

- In main a way for the tree to be creates then serialized an then de-serialized had to be devised, much of this came form the data class provided with the module

**try** {

FileOutputStream fos = **new** FileOutputStream ("data.ser");

ObjectOutputStream oos = **new** ObjectOutputStream (fos);

oos.writeObject(tree);

System.*out*.println("Tree Serialized");

oos.close();

}

**catch** (Exception e) {

e.printStackTrace();

}

**try** {

FileInputStream fos = **new** FileInputStream ("data.ser");

ObjectInputStream ois = **new** ObjectInputStream(fos);

BinarySearchTree tree2 = (BinarySearchTree) ois.readObject();

System.*out*.println("Tree Deserialized");

ois.close();

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

System.*out*.println("\*\*Tree Output: Deserialized\*\*");

tree2.displayBreadth();

}

**catch** (Exception e) {

e.printStackTrace();

}

Testing section:

For the testing, the user it initially asked to input five numbers to input into the addRoot tree this tree will then be displayed normal to show that it is in fact what the user input, this tree will then be serialized and will inform the user that it did in fact serialize using:

**try** {

FileOutputStream fos = **new** FileOutputStream ("data.ser");

ObjectOutputStream oos = **new** ObjectOutputStream (fos);

oos.writeObject(tree);

System.*out*.println("Tree Serialized");

oos.close();

}

**catch** (Exception e) {

e.printStackTrace();

}

This tree will then be de-serialized by the following code and displayed:

**try** {

FileInputStream fos = **new** FileInputStream ("data.ser");

ObjectInputStream ois = **new** ObjectInputStream(fos);

BinarySearchTree tree2 = (BinarySearchTree) ois.readObject();

System.*out*.println("Tree Deserialized");

ois.close();

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

System.*out*.println("\*\*Tree Output: Deserialized\*\*");

tree2.displayBreadth();

}

**catch** (Exception e) {

e.printStackTrace();

}

Output for serialization test.

Root tree created, please insert 5 number sinto tree:

Input first number: 40

Input second number: 57

Input third number: 23

Input fourth number: 13

Input fifth number: 7

\*\*Tree Output: Before\*\*

Breadth-first Traversal with null nodes::

7 null 13 null 23 null 57 40 null null null

Breadth-first Traversal::

7 13 23 57 40

Tree Serialized

Tree Deserialized

\*\*Tree Output: Deserialized\*\*

Breadth-first Traversal with null nodes::

7 null 13 null 23 null 57 40 null null null

Breadth-first Traversal::

7 13 23 57 40

**Critical Appraisal**

I found this module to be the most challenging of the year, in both test form and programming project. I severely miss allocated my time towards the project, I early on completed the first two section and only had to make minor improvements to them upon coming back to the project a few weeks later, however I underestimated the time I would need to complete section C and ended up having to scrap what I had and leave it unfulfilled.

I believe I struggled most with part A of the project as it took me a while to accompany myself with the class itself and the coding style, after this initial struggle part B fell quite easily in line with me only having to reread chapters of the book to understand its process a few times.

I believe this project has however grown my confidence and programming skills when I come to handling an unknown format, and has shown me that I need to more evenly distribute my time towards some things and properly evaluate how much time is needed per section.