|  |
| --- |
|  |
| Algorithms & Collections Report |
|  |
| By B00235610 |

**B00235610**

January 9, 2014

Authored by: B00235610

Algorithms & Collections Report

By B00235610

# Introduction

The following is a report split into 4 parts that is part of my coursework for the Algorithms and Collections module. The report covers the following areas.

* Extended Binary Search Trees
* Root insertion when adding new elements to the tree
* Insertion at external nodes versus at the root
* Serialization of a Binary Search Tree

The report is split into 4 parts with a final part at the end saved for references and a self-evaluation of my own work.

# Part 1 – Implementing an Extended Binary Search Tree

In this section we were to implement some extra features to a Binary Search Tree class provided from our lecturer which was taken from a book written by William J Collins. The class was also written by him as well. I will start of by writing about my initial updates I applied to the class to make it more readable and then go over the updates that were performed to apply the functionality requested.

**A general clean up**

I first went through a general clean-up of the code and added scope to areas where it was difficult to navigate due to lots of 1 line scope levels. Like this

if(true)

something;

else

something\_else;

I for one don’t like this style as when presented with a large block of these statements it because less readable and as I would be spending a lot of time with this class I thought it best to insert braces to suit my preferences. I also removed the useless scope comments which were rather insulting and replaced them with general comments that help stablish what a function is doing and how it is going about doing it. I also needed to add @Override annotations on some of the functions as well as they weren’t present. Once this was done I have finished the immediate updates.

**External and Internal nodes**

The first task was to extend the functionality of both the Entry<E> internal class and the **Binary Search Tree** class. Now the **Binary Search Tree** would become an **Extended Binary Tree** where new nodes are inserted at an external nodes position. Where internal nodes are nodes that contain a value and external being one of the many leafs of the tree which don’t hold a value but do represent the end of the tree and hold a reference to their parent.

To tell if an Entry<E> is external or internal I have provided the two functions:

public boolean isExternal()

public boolean isInternal()

Is external simply checks to see if the node stores a value if it does it returns false as it isn’t an external node. The ***isInternal()*** function just calls **the *isExternal()***function and reverses the value it returns and is used in areas to make code a little more readable.

The task was also to implement the following functions for the class Entry<E>

**public void makeInternal(E element)**

**public E makeExternal()**

These were simple to implement, ***makeInternal()*** simply sets the value of the node and ***makeExternal()*** simply makes everything null apart from the parent and returns the value it did hold back to the user. However this wasn’t all that was required. I also needed to make sure that nodes were inserted at an external node in the add function.

Now all I needed to do was to change the add function so it inserted at an external node. So instead of search for the first null reference instead we just keep searching until an external node is found so we change this.

**if**(comp < 0) {

**if**(temp.left != null) {

temp = temp.left;

} **else** {

temp.left = new Entry<E>(element);

insertionSuccess = **true**;

}

}

**To this**

**if**(comp < 0) {

**if**(!temp.left.isExternal()) {

temp = temp.left;

} **else** {

temp.left.makeInternal(element);

insertionSuccess = **true**;

}

}

Instead of checking for a null node we just search for the next external node. Once found we make it an internal node and add two external nodes to fill its left and right sub trees. However once this has been changed I had to change all functions that use a null reference as a terminator for a loop. So for instance in the contains() method the search now terminates when it finds an external node rather than a null node and this is now the case for the following functions.

***deleteEntry(), successor(), contains(), getEntry(), equals()***

Just as a note I also needed to implement an equal’s method for the entry class that will handle when external nodes are being compared as they won’t have a value to compare to.

**Displaying a tree**

For displaying the tree I have overridden the standard *toString()* method for the Binary Search Tree class. I have also overridden the toString() method for the Entry internal class which will either send back a string representation of the value it contains to the function or will return a hash if it is an external node.

For doing a breadth width traversal for the tree I used the following pseudo code which was taken from **Stack Overflow** where a link will be in the final section of the report.

Queue<TreeNode> queue = new LinkedList<BinaryTree.TreeNode>();

public void breadth(TreeNode root) {

if (root == null)

return;

queue.clear();

queue.add(root);

while(!queue.isEmpty()){

TreeNode node = queue.remove();

System.out.print(node.element + " ");

if(node.left != null) queue.add(node.left);

if(node.right != null) queue.add(node.right);

}

}

**Calculating the height of the tree**

Calculating the height of the tree was a fairly simple process that was done recursively based on the pseudo code provided as an answer to a question in a practice test during the module. I have listed the pseudo code below.

if t is empty

height(t) = -1

else

height(t) = 1 + max(height(leftTree(t)), height(rightTree(t)))

### Test results

##### Test case 1

The first test I did was just to confirm that all the methods I updated do indeed work and to ensure that the iterator class works as desired as well. The test starts by setting up the tree and by then adding random integers for a set number of insertions we also check for duplicates to make sure we don’t have duplicate values. Once then we add 10 and immediately remove it. Followed by a print of the details (height, contains 10, nodes) of the tree. We also include comparison test to see if it matches itself. The following test produces a result similar to this confirming my update works as expected.

In-order traversal of tree: 18, 21, 26, 30, 36,

Number of duplicates: 0

Binary Tree in BWT: 21, 18, 36, #, #, 30, #, 26, #, #, #,

Tree Height: 3

Contains 10: false

Matches: true

##### Test case 2

The second test I did was a comparison between the new tree I have just created and the tree before it was altered. I took the time between 10 insertions of both classes respectively then deleted all elements. The following test resulted in the following times which very much surprised me.

Insertion time for new tree: 42844 ns

Insertion time for old tree: 491103 ns

Deletion time for new tree: 45674 ns

Deletion time for old tree: 17381 ns

During my tests I found that an insertion time for an extended binary tree are faster than a regular binary tree however I found that it is slower for removing nodes than a regular binary search tree. My biggest guess for the insertion performance increase would be because our extended binary tree class has its root node pre allocated before we add elements.

# Part 2 – Insertion at the root

### Adding the new constructor

The first update that needed doing for this part of the assessment was to add a new boolean field called ***insertAtRoot*** and a new constructor that would set the variables value. The value would be false by default.

**public** BinarySearchTree(**boolean** insertAtRoot) {

// call default constructor

**this**();

// then set boolean value

**this**.insertAtRoot = insertAtRoot;

}

This was very simple the next few parts were the more tricky parts.

### Adding the modified functions provided

Insertion at the root is not what it first seems first the value is actually inserted at an external node that is then rotated to reach the root node. While both the rotation functions will work fine the recursive ***insertT()*** that was provided needed to be modified due to the fact we were using external nodes and not null nodes. We also had to deal with the fact that the function provided didn’t deal with the parent of the node that needs to be sorted. Down below I have left the function as it was before modification and after the new function which has been modified to fit the needs of the Extended Binary Search Tree.

**Function given**

**private** Node insertT(Node h, ITEM x) {

**if** (h == **null**) **return** **new** Node(x);

**if** (less(x.key(), h.item.key()) {h.l = insertT(h.l, x); h = rotR(h);}

**else** {h.r = insertT(h.r, x); h = rotL(h);}

**return** h;

}

**public** **void** insert(ITEM x) { root = insertT(root, x); }

**Altered function**

**private** Entry<E> insertT(Entry<E> e, E v) {

// if we have reached an external node we will insert the value

**if**(e.isExternal()) {

e.makeInternal(v);

} **else** {

// if we haven’t we will see what sub tree to check next

**if**(((Comparable<E>)v).compareTo(e.element) < 0) {

// rotate after the value is inserted to bring to the top

e.left = insertT(e.left, v);

e = rightRotationOn(e);

}

**if**(((Comparable<E>)v).compareTo(e.element) > 0) {

e.right = insertT(e.right, v);

e = leftRotationOn(e);

}

}

**return** e;

}

The first change you will notice that we now handle that we will in fact be inserting at the external node instead of a null reference. We also ensure that the new internal node is preceded with two external nodes for its left and right sub trees keeping the tree as a complete tree.

I have also ensured that the comparison matches the comparison operation that is found in the add function which is done by using the compareTo method defined by the comparable interface. Here is no difference. We don’t handle that case for if compareTo isn’t implemented as a value won’t be inserted into the tree is it hasn’t been implemented anyway.

### Extending the add function

Extending the add function was quite a simple task first we check to see if we should insert at the root and if so we just call the ***insertT()*** function providing the root of the tree as the starting node. However this doesn’t handle the case of duplicate values. To solve this we altered the function above just to return null if we tried to insert a duplicate value and the root will only be updated if the node returned isn’t null. As for the add function all we do is add a simple if statement and call the insertT() function. One problem that did come up was with the parent for each node. When applying a rotation the reference to the parent would be lost and it’s not really possible to sort this problem in the rotation function itself. As the parent you may assign may be rotated and replaced with a different parent after. To solve this I updated the rotate functions to also switch the parents during the rotation which seemed like the most logical thing to do. While I could of just called a recursive function to correct all the parents after the new value had been added to the tree this would have been inefficient as it would of resulted in another traversal of the tree.

**if**(insertAtRoot) {

**int** oldSize = size;

Entry<E> tempValue = insertT(root, element);

**if**(tempValue != **null**) {

root = tempValue;

root.parent = **null**;

}

**if**(oldSize != size) {

insertionSuccess = **true**;

}

### }

### Test results

#### Test 1

The first test I did was similar for the test I did for part one. I tested a few methods but this time I also print the insertion order so I can make sure values are inserted correctly with the last inserted value being the root and the second last being one of its children. Here are the results of my test.

Insertion Order: 3, 10, 8, 9, 8,

Binary Tree in BWT: 8, 3, 9, #, #, #, 10, #, #,

Tree Height: 2

Contains 5: false

Matches: true

It’s also worth mentioning that you will notice that duplicates are inserted into the tree but are ignored.

#### Test 2

The second test I did was again similar to the test for part one I just compared the insertion and deletion performance when it comes to times again against the Binary Search Tree that was provided by the lecturer. A similar result to last time however while insertions are quicker deletions are not.

Insertion time for new tree: 64267 ns

Insertion time for old tree: 494739 ns

Deletion time for new tree: 34356 ns

Deletion time for old tree: 16977 ns

# 3 – Comparing Tree Structures

### Comparisons needed for successful and unsuccessful searches

##### Expectation

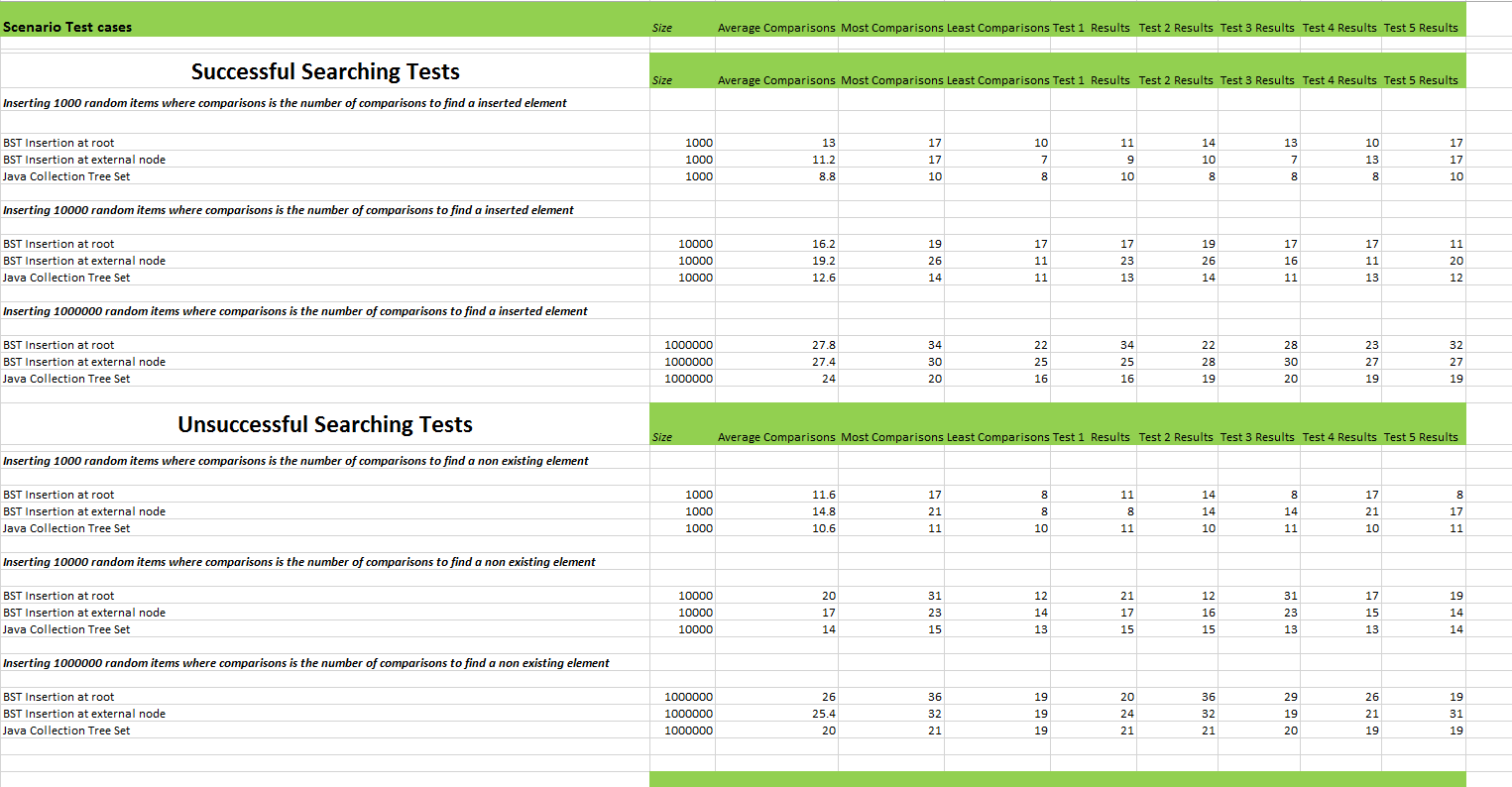
My initial expectation would be that the TreeSet would dominate the tests that required the least amount of comparisons to find a certain Item element and indeed to terminate an unsuccessful search. However I would expect that the insertion at the root to be more efficient than insertion at an external node as there is a chance that upon searching for the node that it is closer to the root as it was inserted recently. However I have no comment or idea on how this would affect unsuccessfully searches.

##### Tests

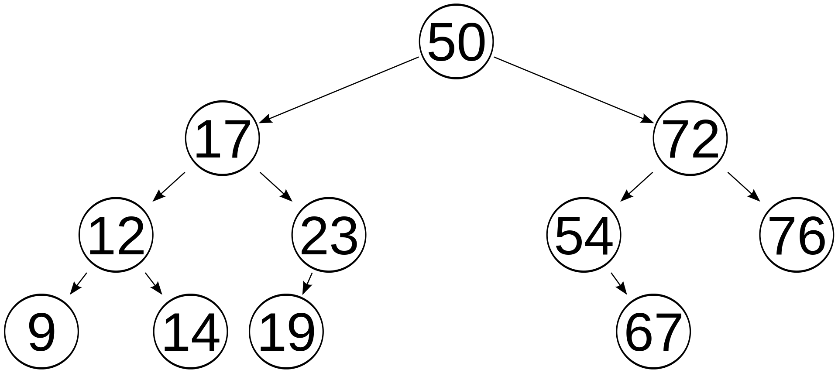
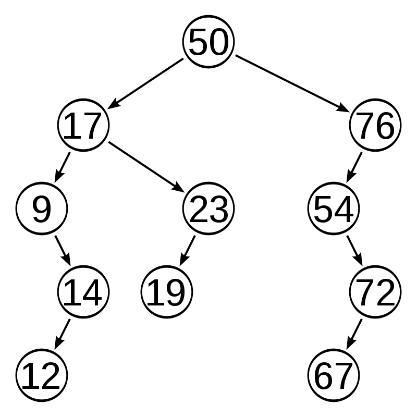
The tests that were done were simple I did 5 tests for successful and unsuccessful searches each. I then took the average number of comparisons as well as the highest number of comparisons as well as the lowest. I also did each of the 5 tests with different size. I did the tests with 3 sizes. Small was capped at 1000 elements, Medium was capped at 10000 elements and Large was capped at 1000000 elements. The following tests where then inputted into Microsoft Excel to make it easier to analyse the results. The excel file is also located next to this report in my submission.

##### Results

The following tests resulted in this excel spreadsheet.



As expected the red black tree beats the other two methods of insertion in two ways. First its average number of comparisons is better. This is no doubt due to the fact that red black trees are self-balancing meaning the height of both left sub trees and right sub trees are kept as equal as possible resulting in fewer comparisons to find a node and also to reach at the end of the tree. Due to the self-balancing act the number of comparisons is also more consistent compared to the other two methods. The other two insertion methods are not as efficient as they are unbalanced trees meaning the depth of the tree can be far larger than a self-balanced tree as shown in the example below.



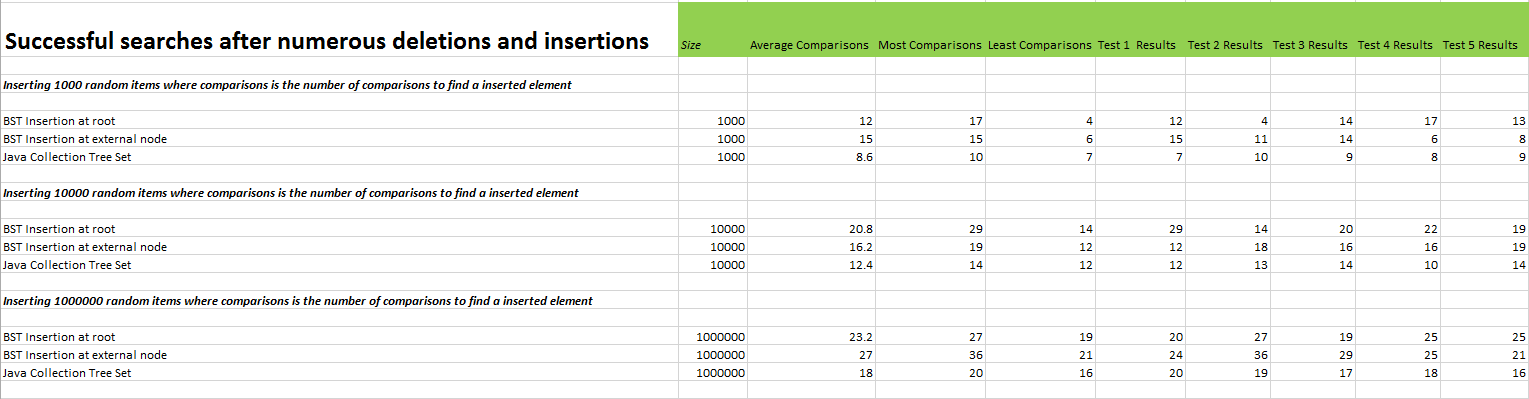
### Comparisons need for searches after numerous deletions and insertions

##### Expectation

Here I would expect again the red black tree to be even more victorious than before again due to the fact that the tree is self-balancing meaning searching for elements will be quicker as the tree is balanced after every deletion insuring that a search operation takes as little comparisons as possible.

##### Tests

Again like previously I did this test with various number of insertions. 1000, 10000 and 1000000. Also like previous I took 5 tests and got the best score, worst score and also averaged all the 5 tests to ensure I had a simplified view of the data.



##### Results

As expected the red black tree here dominates in every test. On closer inspection though we do find some interesting things happening which I did not expect. First insertion at the root and at external nodes performance is completely dependent on the random node picked to search for. If it’s a recently inserted item then insertion at the root wins as shown in the average number of comparisons in the test with 10000 elements. If the node picked is one closer to the root due to an unbalanced tree then insertion at external nodes wins. However because of this the number of comparisons required is sporadic at best whereas the red black tree delivers a more consistent number of comparisons needed to find an element. Over time this effect is worsened as well as after deleting and inserting new nodes the tree will become more unbalanced due to the fact they aren’t self-balancing which was proven with calls the getHeight function on each test where the left trees height would differ by a large amount. I would then try the roots sub trees and find the same result. This was not put in the excel document however I did inspect and notice that the trees become more unbalanced over time. Also a bigger tree results in a more unbalanced tree

at least the insertion at the root and external nodes trees are because the TreeSet class is a red black tree it self-balances itself making the tree as balanced as possible.

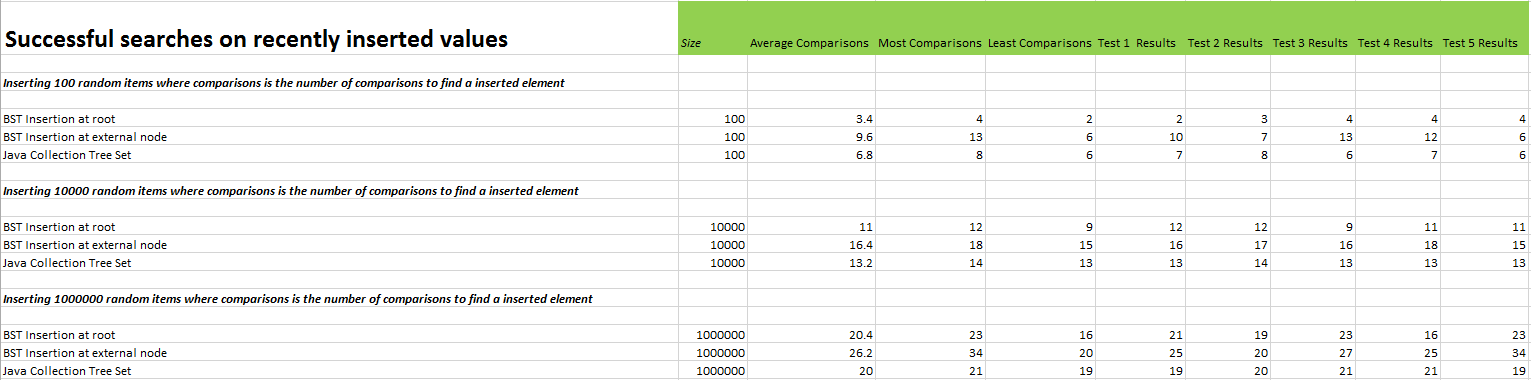
### Comparisons needed for recently inserted items

##### Expectation

My expectations is that here the insertion at root will result in the least number of comparisons where insert at external root will lead to the most number of comparisons where the red black tree will be just better as its self-balancing act will make it slightly better. I believe this to be true as insertion at the root will always result in the last 10% of items inserted being close to the root. Whereas the 10% of recently inserted items will be at the bottom of the tree for the insert at the external node tree.

##### Tests

I first tested the number of comparisons needed to find an item which was part of the last 10% of recently inserted items. I did 5 tests where I took the average comparisons the least and the most and can be viewed in the Excel Spreadsheet. I also did each of these tests with varying tree sizes, 100, 10000 and 1000000.

******

##### Results

My initial expectation was correct that insertion at the root was faster than the other tree types. However I didn’t expect this advantage to be lost when the tree size is dramatically increased to say 100000 elements. In my testing I found this to be because the tree was less balanced and when dealing with large numbers of nodes the last 10% of nodes can be very far from the root of the tree due to the rotations needed to bring the most recent node up to the root will also push the previous nodes down. Thus here the red black self-balancing tree is better here.

# Part 4 – Serialization of Extended Binary Search Trees

### Serialization method

The final part of the assessment was to make the tree serializable by implementing the serializable java interface.

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

The method I developed isn’t perfect as it doesn’t re construct the tree in the most efficient way as I do just re insert the values stored in the file by calling the add function. Also it wasn’t possible to save space due to a founding fact about Binary Search Trees which is that each value is unique so we can’t save space in the typical way of identifying patters in the data because each node is unique as no duplicate values can be inserted into the tree.

#### Writing process

So to implement a custom serialization I must first implement the private writeObject function which takes the stream that is writing the object as a parameter. I implemented the function as follows.

**private** **void** writeObject(ObjectOutputStream stream) **throws** IOException {

// write the insertion type & the size of the tree

stream.writeBoolean(insertAtRoot);

stream.writeInt(size);

// a breadth width traversal of the tree inserting values into the stream

Queue<Entry<E>> queue = **new** LinkedList<Entry<E>>();

queue.add(root);

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

Entry<E> node = queue.remove();

**if**(node.left != **null**) {

queue.add(node.left);

}

**if**(node.right != **null**) {

queue.add(node.right);

}

stream.writeObject(node.element);

}

}

I start of by writing the way we are currently inserting the nodes into the tree into the stream. Following that I then insert the current size of the tree so we know how many node values to read when we read from the stream. I use object packing I pack each node value as an Object and then cast it back to its generic value when we read the input stream. When it comes to packing the values into the stream I just do a breadth width traversal as it doesn’t require any extra methods and it allows us to reconstruct the tree perfectly.

#### Reading process

So to be able to read the object from an input stream I implement the other function that is part of the serializable interface, readObject() which takes the input stream as a parameter which is the stream that has loaded the file for us.

**private** **void** readObject(ObjectInputStream stream) **throws** IOException, ClassNotFoundException {

// First initialize the object as the constructor isnt called

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

insertAtRoot = **false**;

size = 0;

// then get the insertion type

**boolean** insertionType = stream.readBoolean();

// and the number of internal nodes

**int** sz = (stream.readInt());

// then iterate through reading each object

**for**(**int** i = 0; i < sz; i++) {

E v = (E)stream.readObject();

// if its an external node

**if**(v == **null**) {

// increment the size so this node doesnt count

sz++;

} **else** {

// else add this value to the tree

add(v);

}

}

// then set the setting for future insertions

insertAtRoot = insertionType;

}

Here we first do everything the standard constructor would do as it won’t be called if we create the object directly from file. Once that is done we get the insertion type that should be applied ***after*** all elements have been inserted into the Binary Search Tree. We then get the number of elements that we need to read and read them one by one. If null is returned it means the node was an external node and here we just increment the size variable as an indication to skip that value and say it doesn’t count. If it isn’t a null value we insert into the tree using the standard add function which we have implemented.

The writeObject function is even simpler. We simply write the Boolean on weather we should insert at the root and then write the size of the tree. Finally we perform a Breadth Width Traversal and insert every nodes value into the stream including external nodes. I have linked the source code for this function below.

**private** **void** writeObject(ObjectOutputStream stream) **throws** IOException {

// write the insertion type & the the size of the tree

stream.writeBoolean(insertAtRoot);

stream.writeInt(size);

// then perform a breadth width traversal of the tree inserting values into the stream

Queue<Entry<E>> queue = **new** LinkedList<Entry<E>>();

queue.add(root);

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

Entry<E> node = queue.remove();

**if**(node.left != **null**) {

queue.add(node.left);

}

**if**(node.right != **null**) {

queue.add(node.right);

}

stream.writeObject(node.element);

}

}

The following successfully serializes and de serializes the binary search tree class.

#### Tests

Here we are only saving and loading the object from a stream so I just did a single test. In this single test I simply inserted a random set of values and then printed the two trees created. The one that was initially created by inserted values and then saved to disk and also the tree that was created by de serializing the object that was saved to disk which should match the tree created previously.

Tree created at runtime: 6, 2, 9, 0, #, #, #, #, #,

Tree created from file: 6, 2, 9, 0, #, #, #, #, #,

Trees Equal? True

# Critical Appraisal

I did this report and the project individually and am fairly confident that both the report and the project were done well. While I cannot say the entire solution is bug free because there is bound to be some bugs somewhere I have found it bug free when it came to the tests I ran which tested large parts of the functionality.

# References

<http://stackoverflow.com/questions/5262308/how-do-implement-a-breadth-first-traversal>

<http://ceca.pku.edu.cn/file/2012111491622939.pdf>

http://en.wikipedia.org/wiki/Self-balancing\_binary\_search\_tree

<http://planetmath.org/extendedbinarytree>

http://en.wikipedia.org/wiki/Tree\_rotation