ICP-2027 ASSIGNMENT TWO

The Task

Run empirical studies to compute the average length of a path to a random node in both a Sorted Array and a Binary Search Tree. Build both structures by insertion of N random keys, into an initially empty structure, for N from 100 to 10,000. Plot the results with the horizontal axis representing tree size N and the vertical axis representing average number of comparisons.

Defining the data structures

Array:

An array is a data structure of fixed length, determined on compilation, containing a sequence of indexed components. Each index of the array is unique to a single component, allowing components to be accessed by their index. Access to components via their index ensures a time complexity of *0(1)* . Figure 1 is an example of an array containing integers.

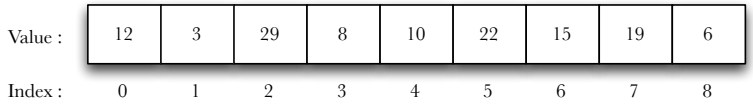

Figure 1. An array of length 9 holding integer values.

Sorted Array:

An array can be defined as a sorted array if it's components appear in the ascending order of the values associated with them. Figure 2 shows a sorted version of the array seen in Figure 1.

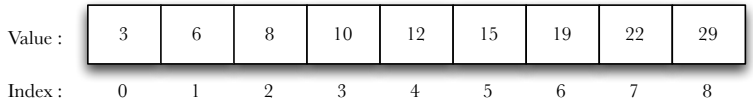

Figure 2. A sorted version of the array seen in Figure 1.

Tree:

A tree is a hierarchical collection of elements beginning with a unique root element. Each element within the tree can contain any number of links to child elements but must have only one parent. An element with no children is a leaf.

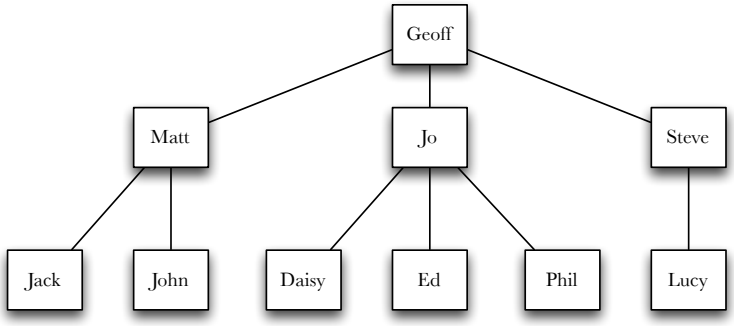

Figure 3. A family tree.

Binary Tree:

A binary tree is a tree who's elements contain links to two children only, a left child and a right child. The size of a binary tree is not fixed at compilation because there will remain empty spaces for links at every leaf element and some intermediate elements. Figure 4 shows a binary tree.

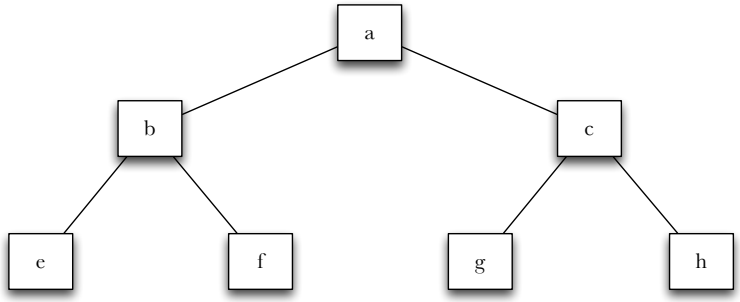

Figure 4. A binary tree.

Binary Search Tree:

A binary search tree is a binary tree where every element contained in an elements left sub tree will evaluate to lower than than the elements value and every element contained in it's right sub tree will evaluate to hight then the elements value. Figure 5 below shows a binary search tree.

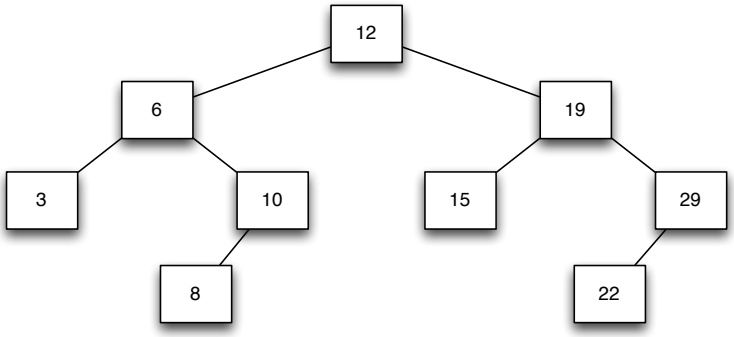

Figure 5. A binary search tree of the sorted array in Figure.2.

Balanced Binary Trees

A binary tree is well balanced if both sub trees of the root element are similar in size. Figure 5 above shows a well balanced binary search tree. It follows that an ill balanced binary tree will have uneven sub trees of the root element and in the worst case the root element will have only one sub tree with each child having only one child except the leaf element. Figure 7 shows an ill balanced version of the same tree while Figure 6 shows the most ill balanced tree possible.

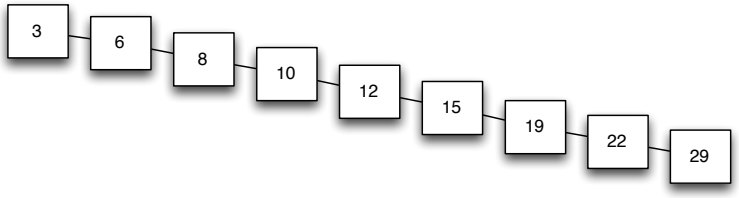

Figure 6. The worst case of an unbalanced binary search tree of the sorted array in Figure.2.

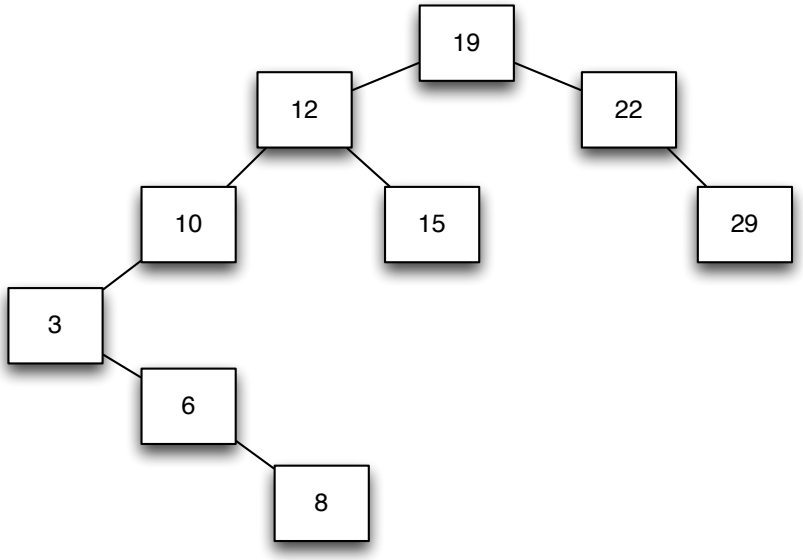

Figure 7. An unbalanced binary search tree of the sorted array in Figure.2.

Implementing the data structures

The implementations of both sorted array and binary search tree that have been chosen rely heavily on the Comparable interface provided by Java. The class implementing Comparable is called Node and contains two instance variables named value & comparisons. The value is the number to be stored in the data structure while comparisons is a count of the number of times the Comparable's compareTo() method is called. Figure 8 below is compareTo() from the Node class and demonstrates how Nodes will be compared.

```
@Override
public int compareTo(Object o) {
    Node n = (Node) o;
    if (value > n.getValue()) {
        comparisons++;
        return 1;
    } else if (value == n.getValue()) {
        return 0;
    } else {
        comparisons++;
        return -1;
    }
}
```

Figure 8. compareTo() method from Node class.

Sorted Array Implementation:

The sorted array is implemented by the class SortedArray with an ArrayList as provided by Java. The integrity of the sorted array is maintained by the method fillArray(). In fillArray() Nodes are appended to the array with a randomly generated value if a node with the same value is not already present. After a node has been appended then the array is sorted using the Java Collections method sort(Collection c). A local count variable is used to control the Node insertion loop and ensure the array is full. Figure 9 below is the code for fillArray().

```
private void fillArray() {
    Random gen = new Random();
    while (count < numNodes) {
        //Value to instantiate Node
        int input = gen.nextInt(numNodes);
        //Node to be inserted
        Node node = new Node(input);
        if (!array.contains(node)) {
            array.add(node);
            //Reset node depth after search above.
            node.setDepth(0);
            Collections.sort(array);
            count++;
        }
    }
}
```

Figure 9. fillArray() method from SortedArray class.

Binary Search Tree Implementation:

The binary search tree is implemented by the class BinarySearchTree with a combination of BST and BSTNode as provided by Watt Brown. The class BST ensures the integrity of the tree, as defined in the task description, and the method fillSearchTree() of BinarySearchTree presents random nodes to be added to the BST. Just as in SortedArray a count variable controls the iteration of the insertion loop. Figure 10 below shows fillSearchTree().

```
private void fillSearchTree() {
    //new random number generator
    Random gen = new Random();
    int count = 0;
    while (count < numNodes) {
        //Value to instantiate Node
        int input = gen.nextInt(numNodes);
        //Node to be inserted
        Node node = new Node(input);
        if (tree.insert(node)) {
            count++;
        }
    }
}
```

Figure 10. fillSearchTree() from BinarySearchTree class.

Collecting the data

Although comparisons are counted within the nodes of the data structure, data is collected through the class DataCollection. This class has a single public method called getDataAndWriteXML() detailed below in Figure 11.

```
public void getDataAndWriteXML() {
    gatherData();           //Data Collection
    writeXML();             //write XML
}
```

Figure 11. getDataAndWriteXML() from DataCollection.

The helper method writeXML() writes the data collected by gatherData() to disk for quick access when visualising. The average number of comparisons for the trials is calculated and collected by the method gatherData(), outlined below in Figure 12.

```
private void gatherData() {
    //Data Collection
    for (int i = 100; i < 10000; i += 5) {
        double avgCompBST = 0; //Average number of comparisons
        double avgCompSA = 0;
        //The data structures to be searched
        BinarySearchTree bst = new BinarySearchTree(i);
        SortedArray a = new SortedArray(i);
        int trialLength = 1000; //The trial
        for (int j = 0; j < trialLength; j++) {
            avgCompBST += bst.search(new Node(gen.nextInt(i))) + 1;
            avgCompSA += a.searchArray(new Node(gen.nextInt(i))) + 1;
        }
        avgCompBST = avgCompBST / trialLength; //Calculate average
        avgCompSA = avgCompSA / trialLength;
        //Add to data set
        binaryTreeDataSetA.add(new DataNode(i, avgCompBST));
        binaryTreeDataSetB.add(bst.getDataNode());
        sortedArrayDataSetA.add(new DataNode(i, avgCompSA));
        sortedArrayDataSetB.add(a.getDataNode());
    }
}
```

Figure 12. gatherData() from DataCollection.

This method runs the necessary trials at a sample rate of 5. For each sample it creates a binary search tree and a sorted array then repeats the trial 1000 times. During each trial a search of both data structures is performed which returns an integer of the number of comparisons required to locate the given node. 1 is added to the comparison count because the comparisons variable in Figure 8 is not mutated if the comparator evaluates to zero. At the end of the trials the average comparisons are calculated and are used to construct a new DataNode which is added to the list of DataNode to be written to disk. The call to getDataNode on the data structures is to retrieve the insertion data collected when the data structure was constructed. The collection of this data is achieved by adding a few lines of code the the fillArray() and fillSearchTree() methods detailed above. Figure 13 shows the changes to fillSearchTree(), the changes to fillArray are not shown but are identical to those detailed below.

```
private void fillSearchTree() {
    Random gen = new Random();
    double avgComps = 0;     //Average Number of comparisons to build tree
    int avgCompsCount = 0;
    int count = 0;
    while (count < numNodes) {
        int input = gen.nextInt(numNodes); //Value to instantiate Node
        Node node = new Node(input);       //Node to be inserted
        if (tree.insert(node)) {
            count++;
            avgComps += node.getDepth();
            avgCompsCount++;
        }
    }
    avgComps = avgComps / avgCompsCount;
    dataNode = new DataNode(numNodes, avgComps);
}
```

Figure 13. Modified fillSearchTree() of BinarySearchTree.

The two search methods of the data structures return the number of comparisons as apposed to the object being searched for. In the case of the binary search tree, a new Node is passed to the method search(Node n) and it calls the BST method search(Node n), being confident that the number generated is in the bounds of the structure, the comparisons value of the node passed in is returned. Figure 14 shows the method search(Node n) of BinarySearchTree.

```
public int search(Node n) {
    tree.search(n);
    return n.getComps();
}
```

Figure 14. Search(Node n) method of BinarySearchTree

The search method for SortedArray is modified version of the Java Collections indexedBinarySearch(List<? extends Comparable<? super T>> list, T key). The simple modification is a local variable comparisons, which is mutated during iterations then returned. Figure 15, below, is the indexedBinarySearch modification from SortedArray.

```
/**
 * Binary Search method for a given list & matching comparable.
 *
 * This is method as found in Collections, amended to return the number of
 * comparisons.
 *
 * @param <T>
 * @param list
 * @param key
 * @return
 */
private <T> int indexedBinarySearch(List<? extends Comparable<? super T>> list,
                                    T key) {
    //Min position in array
    int low = 0;
    //Max position in array
    int high = list.size() - 1;
    //Number of comparisons
    int comparisons = 0;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        Comparable<? super T> midVal = list.get(mid);
        int cmp = midVal.compareTo(key);

        if (cmp < 0) {
            comparisons++;
            low = mid + 1;
        } else if (cmp > 0) {
            comparisons++;
            high = mid - 1;
        } else {
            return comparisons;     // key found
        }
    }
    return comparisons;  // key not found
}
```

Figure 15. indexedBinarySearch(List<? extends Comparable<? super T>> list, T key) of SortedArray,

Visualising the data

The visualisation of the data collected is through the class DataVisualizer. There are various helper methods within the class to draw the graph's axis and labels but the bulk of the work is done by two methods: getMappedCoords(ArrayList<DataNode> list) and mapRange(double a1, double a2, double b1, double b2, double s). Figure 16 below shows the mapRange method which takes a value, s, in the range, a1 to a2, and translates it into the range, b1 to b2.

```
private static double mapRange(double a1, double a2, double b1, double b2, double s) {
    return b1 + ((s - a1) * (b2 - b1)) / (a2 - a1);
}
```

Figure 16. mapRange(double a1, double a2, double b1, double b2, double s) of the class DataVisualizer.

Figure 17, below, takes values from an ArrayList and converts each value into a pair of coordinates for plotting the data in the graph.

```
private Point[] getMappedCoords(ArrayList<DataNode> list) {
    Point[] coords = new Point[list.size()];
    for (int i = 0; i < list.size(); i++) {
        int y = (int) mapRange(0, upperScaleLimit, 0, (getHeight() - 200),
                               list.get(i).getNumCompare());
        int x = (int) mapRange(0, list.size(), 0, getWidth() - 200, i);
        coords[i] = new Point(x, y + 12);
    }
    return coords;
}
```

Figure 17. getMappedCoords(ArrayList<DataNode> list) of the class DataVisualizer.

Conclusion

Figures 18 and 19 show the program output once data has been collected.

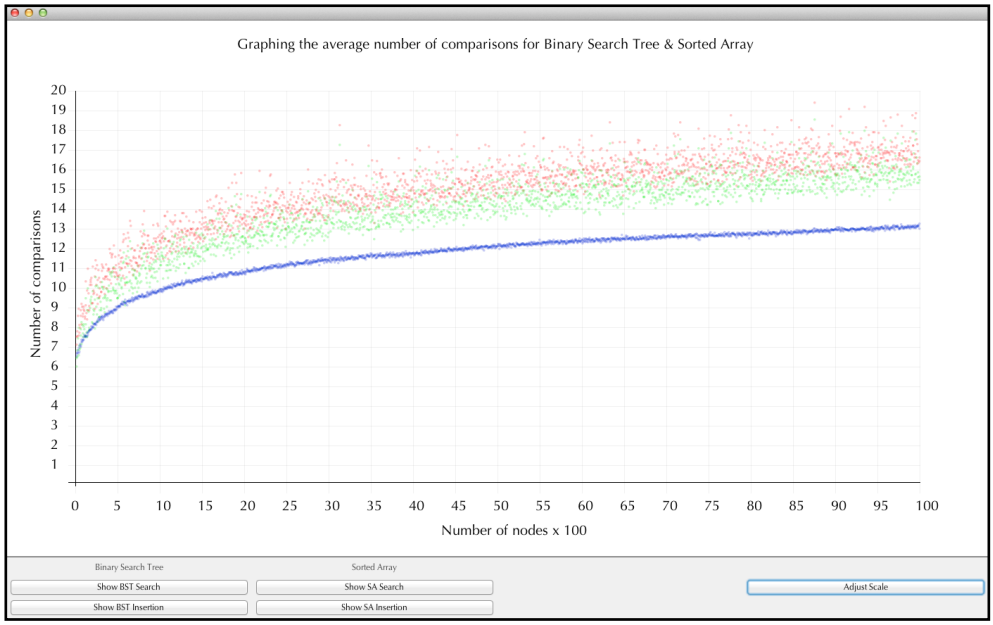

Figure 18. Final program output. Red dots represent number of comparisons for BST blue dot comparisons for SA and green dots are are the number of comparisons for insertion into BST.

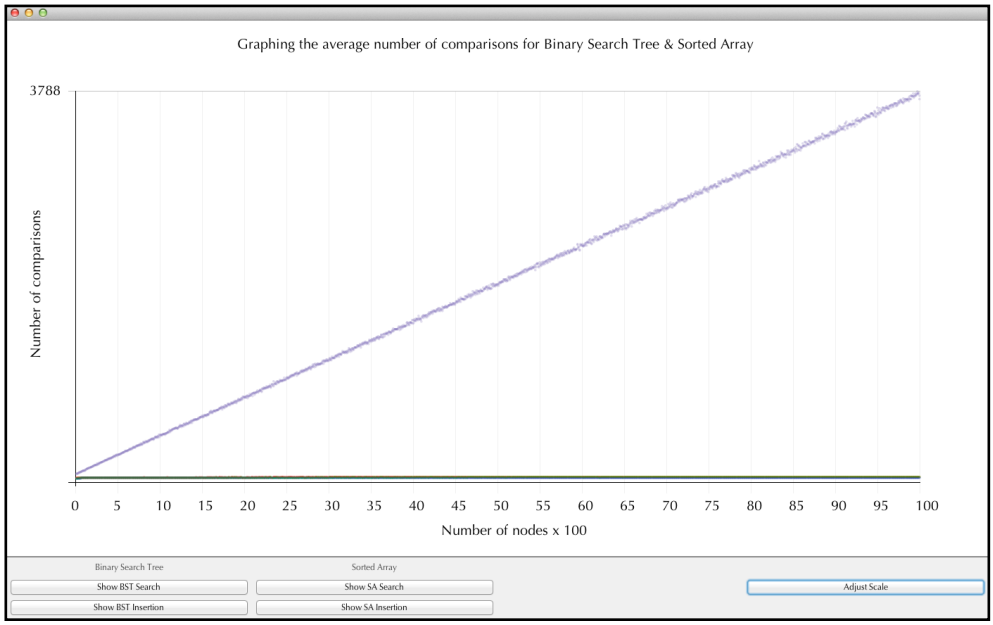

Figure 19. Final program output. The grey line is the same date as Figure 18 on a much larger scale. Purple dots are number of comparisons for insertion into SA.

From the data presented in Figure 18, I conclude that by building a binary search tree with randomly generated or shuffled objects it is possible to achieve a time complexity for searching very close to *O(log n)* without the heavy cost of maintaining the balance of a binary search tree. Figure 19 shows the linear cost of maintaing the sorting order of an array whilst adding components, it demonstrates how the time complexity of inserting objects into a sorted array will be the linear cost *O(n)*. The combination of both Figures 18 & 19 suggests that the cost of sorting data makes the binary search tree much more efficient at processing unsorted data and the sorted array much more efficient at processing sorted data.