1. **Perform a series of experiments with various input sizes to show the complexity of insertion into your AVL tree. Describe how you performed the experiments, justify your choices of input size, and visually represent your results. [5 marks]**

The experiments were performed as follows: Two arrays are created, one with ordered data and one with randomized data. sortedArray[] and randArr[]. (ln 236, ln 243) Alongside two maps used to store the input size and times called sortedMap and unsortedMap. (ln 249-250)

These two arrays were looped through in increments of 10 using two for loops, up until 10,000 was reached giving 1000 readings and therefore datapoints. The time for the last 10 insertions of each operation was noted in nanoseconds using high_resolution_clock and subsequently added into a respective map (sorted or unsorted) alongside the value of the input size, for which the time is respectively for.

```
for (int i = 10; i <= 10000; i += 10) {
    AVL* manager = new AVL(0);
    AVL* TREE = NULL;
    int j = 0;
    for (; j < i - 10; j++) {
        TREE = manager->insertNode(TREE, randArr[j]);
    }
    auto start = std::chrono::high_resolution_clock::now();
    for (; j < i; j++) {
        TREE = manager->insertNode(TREE, randArr[j]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    long timeDelta = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();
    unsortedMap.insert({ i, timeDelta });
}
```

Figure 1, The unsorted loop timing program

This map is then outputted to a CSV for easier graphing within excel. (sortedData.csv and UnsortedData.csv, ln 284-300) This also allows for an assessor to test and confirm similar graphed values easily.

More specifically the input sizes were:

Random numbers inserted in increments of 10, increasing until 10,000 which gave me 1000 datapoints. (IE: 10 random numbers inserted, 20 random numbers inserted, 30 random numbers inserted and so forth). (ln 252)

This was then repeated by using data of the same size that increased sequentially (1,2,3,4,5…10,000.) with the same input size (10 numbers, 20 numbers etc) increasing until 10,000. (ln 268)

The justification for the choices of input size was that a suitable experiment required a range of input sizes, varying from smaller numbers (10, 100 etc) to a reasonably large amount of data (1000, 10,000) to get a trend visible. In order to graph a representative curve (given we know insertion is O(log n)), and earlier testing with smaller numbers gave an indication but it wasn't immediately visible in comparisons less than 10,000. It was also necessary to be tested quicky so larger values that took substantially longer were impractical. It was settled on 10,000 as a reasonable figure as a result.

The sorted dataset was brought in as an experimental condition to examine if it had any meaningful impact on the performance for insertion given the difference in rotation behaviour. (IE: A sequentially sorted dataset will always be repeating the same rotation as the subsequent number will always be larger, relative to random insertion that may require numerous rotations, or possibly none depending on the number inserted.)

Some other factors worth mentioning are the reasons why data was measured in increments of 10 – this is since even whilst measuring in those increments the time difference was often negligible therefore if measured in increments of 1 up to 10,000 all that would occur would be making the graph increasingly crowded unnecessarily. This is similar to the reason the last 10 insertions were measured, relative to say only the last insertion (which was tried) was since it gave a larger window of timing – which in turn decreased the noise on the graph and made it easier to see a visible curve.

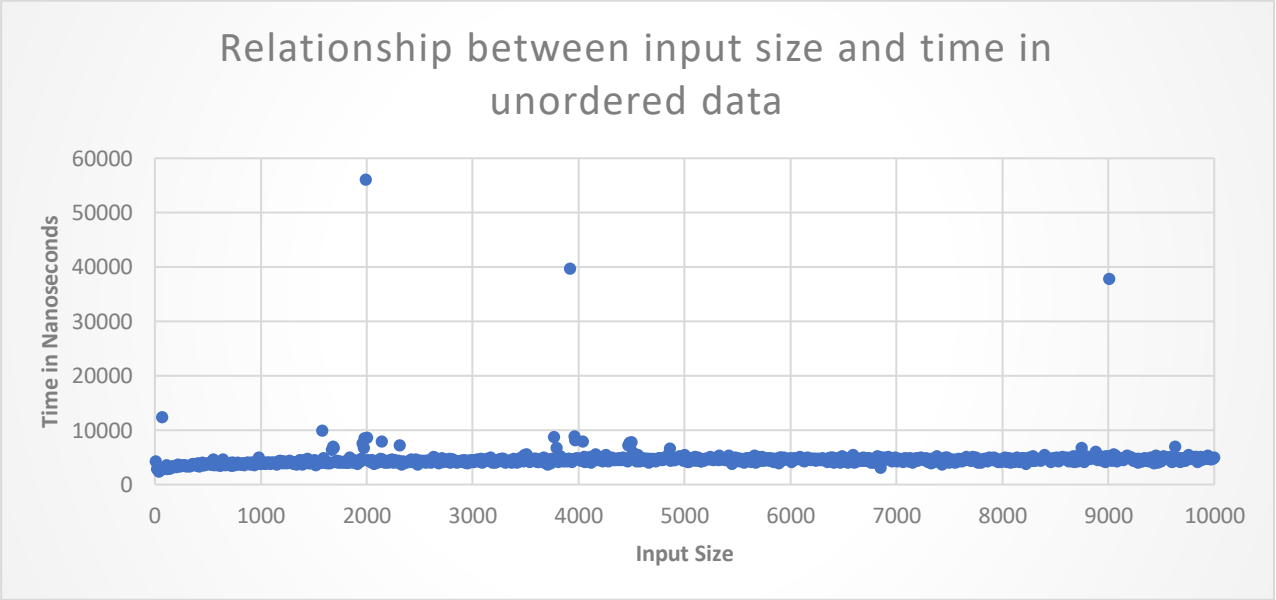

Figure 2

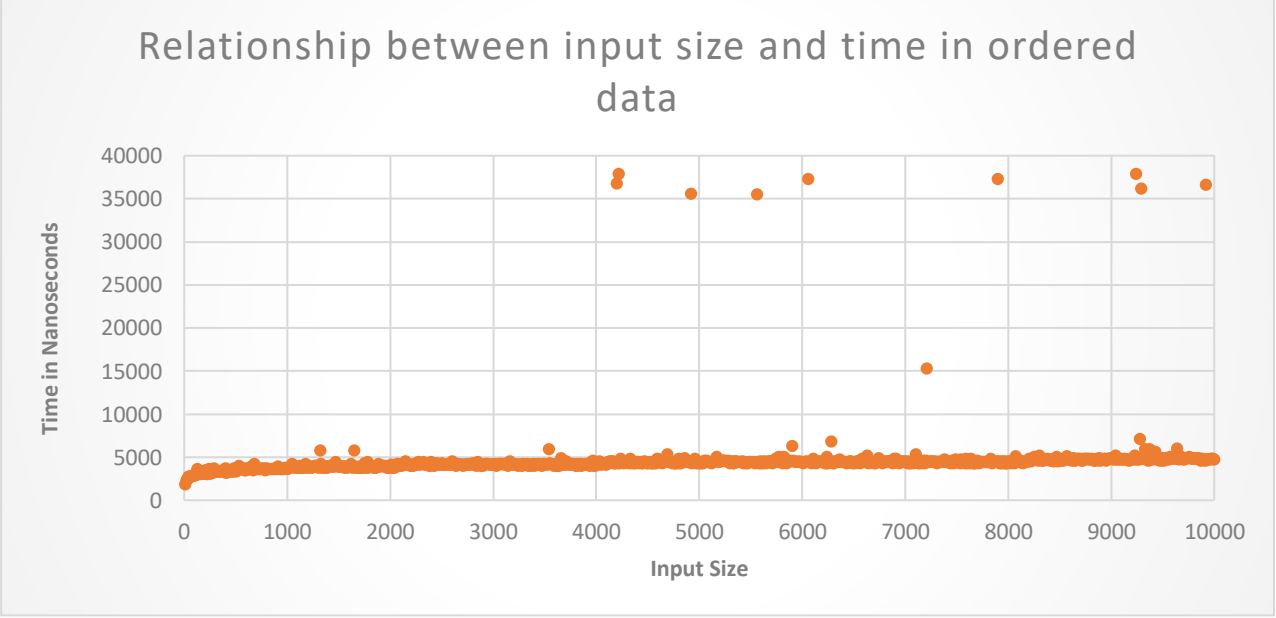

Figure 3

As we can see the outliers in figure 2 and increase the y axis size and as such it is difficult to visualize clearly, so if we zoom in on our y axis, we can see the curve better.

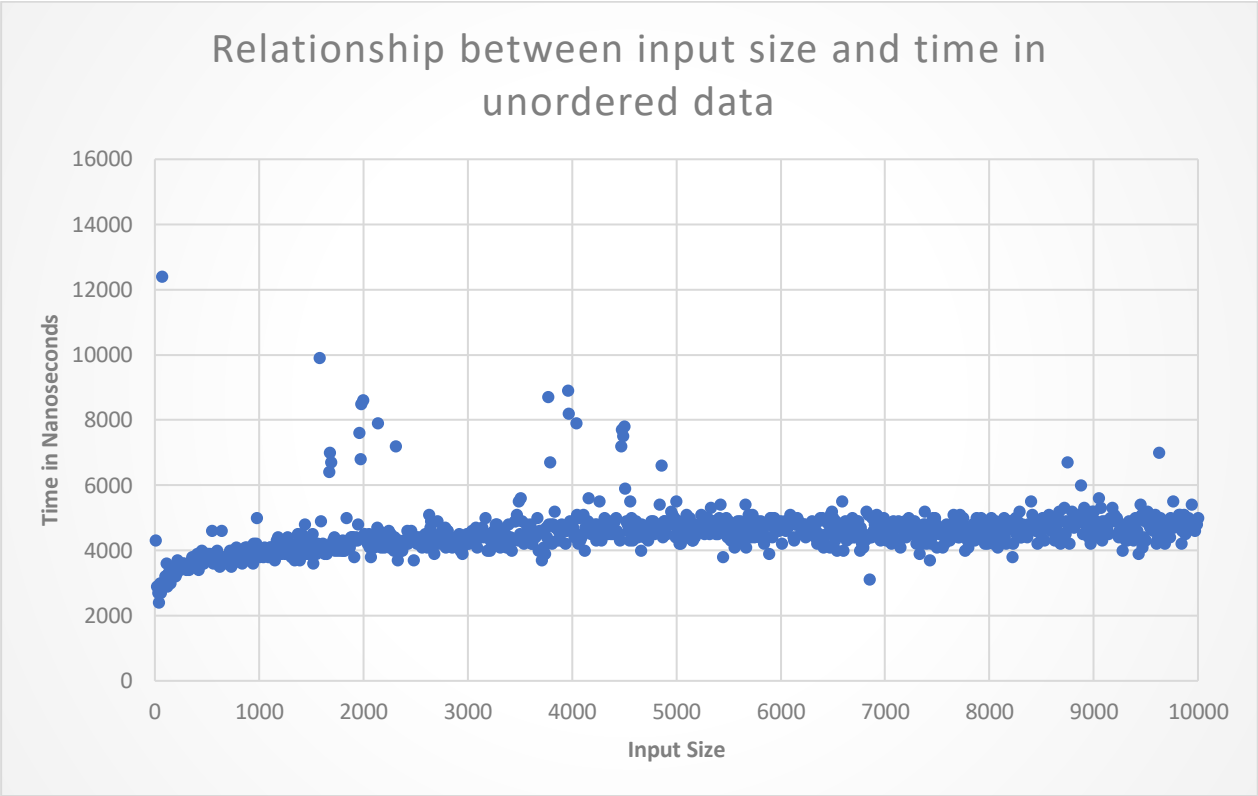

Figure 4

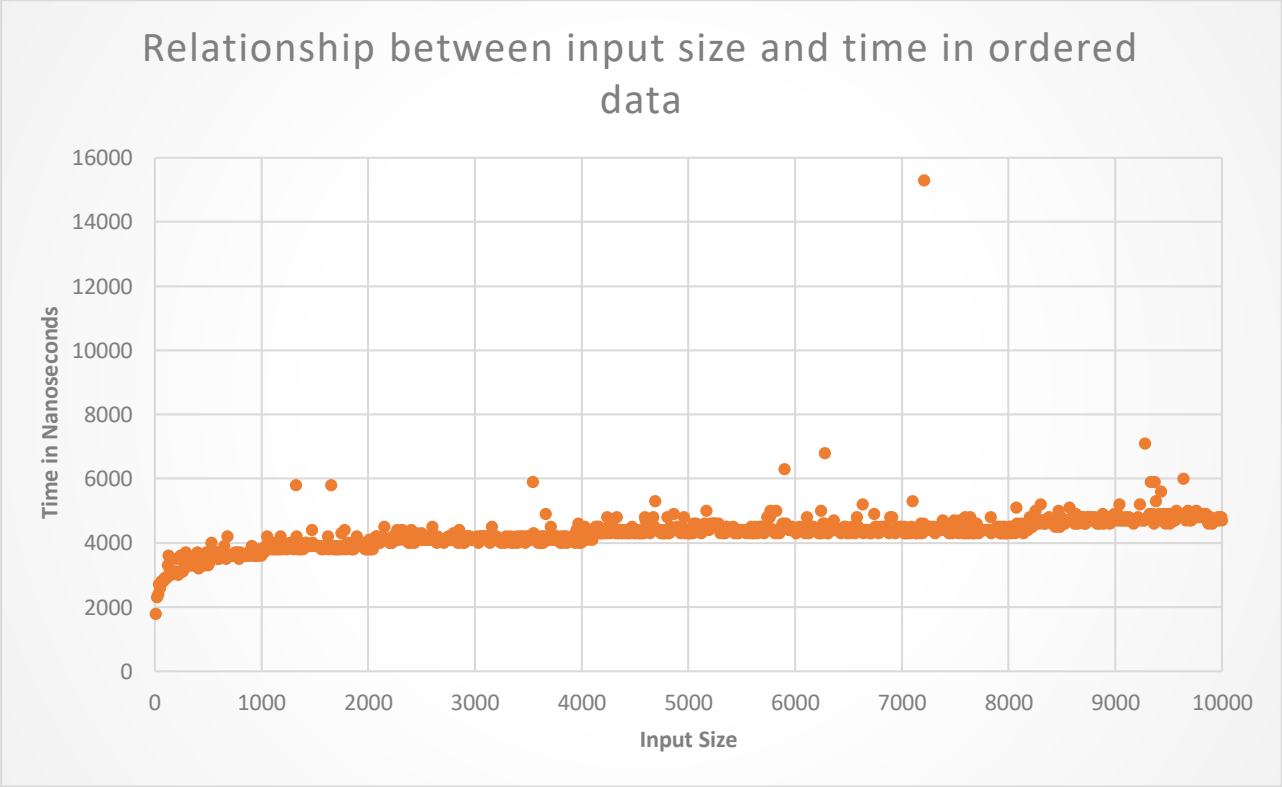

Figure 5

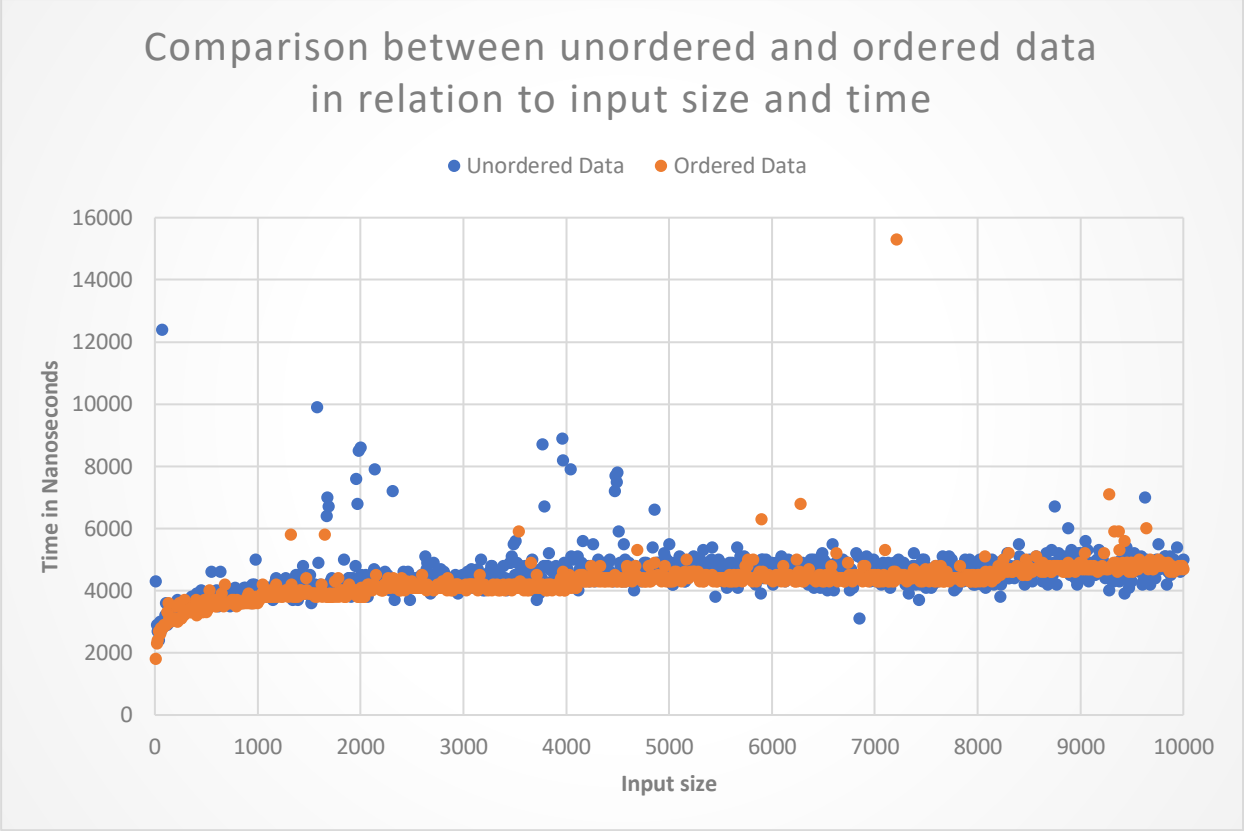
Comparison between unordered and ordered data in relation to input size and time
Unordered Data
Ordered Data
Time in Nanoseconds
16000
14000
12000
10000
8000
6000
4000
2000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input size

Figure 6

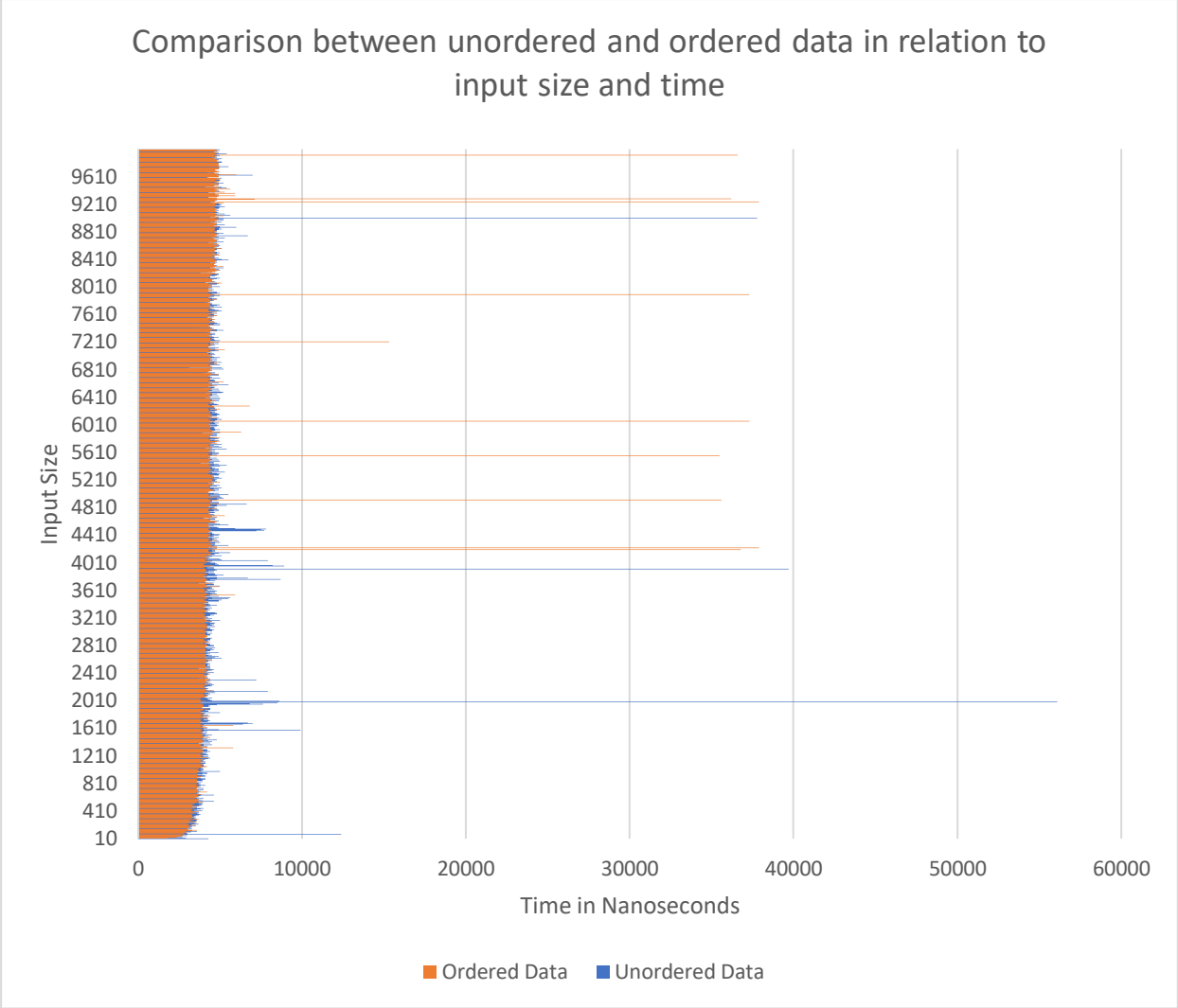

Figure 7

These graphs (Figures 2,3,4,5,6) demonstrate a logarithmic trend which is consistent with a graph of O(log N) which we would expect given that is the complexity of insertion. These also show that unsorted data is generally slower than sorted data when it comes to time taken, alongside figure 7, which we would expect given the potentially huge number of rotations that may need to be performed, relative to the consistent rotations in ordered data. The outliers can be explained easily as performance blips that could be due to any myriad of PC related operations occurring at the same time and do not detract from the clear and obvious trend.

1. [5] - 5 Exactly what was asked for.

2. The usage of what algorithm on an AVL tree can obtain all items in ascending order? Does this technique constitute an efficient sorting algorithm considering insertion into the structure and performance? Compare this approach to other sorting algorithms and weigh the benefits and drawbacks. [4 marks]

The usage of the inOrder algorithm obtains all items in a given tree in Ascending order. This algorithm first requires you to build a tree and first goes down the left branch, before returning to the root, and going down the right branch.

The complexity of this algorithm is O(n log n) when you consider that the insertion of all elements into a tree is O(n log n), and the inOrder function is O(n) so O(n log n) + O(n) which is just O(n log n). This means that even with the insertion operation the algorithm is still efficient relative to others.

When you consider quadratic sorting algorithms (Bubble sort, Insertion sort) which run at O(n^2) it performs far better. In theory it performs about as well as Quicksort, but Quicksort has a worse case of (O(n^2) which is a greater complexity than the worst case an AVL tree has, which means an AVL tree generally performs better. When compared to quicksort it is worth realizing that quicksort essentially corresponds to a BST where the root is the pivot and the left and right subtree represents the smaller and larger subarrays seen in Quicksort. This in turn means we get similar performance for quicksort and inOrder traversal of an AVL (which is just a balanced BST). An additional consideration however is that quicksort does not require allocation in the same way that the creation of an AVL tree does, which makes quicksort much more memory efficient. So overall it would be fair to characterize quicksort as still likely superior, but AVL is undeniably efficient compared to algorithms such as bubble or insertion sort.

2. [4] - 3 Almost! You are describing treeSort, which is what we wanted. But QS is much faster as the cost of balancing slows TS down [-1].

3. Describe an operation to find the largest element of an AVL tree. What would the complexity of such an operation be? [2 marks]

The operation to find the largest element of an AVL tree would be simple, due to the fact that the right child of a given node is always larger, you would traverse to the right, checking the next node until you came across a NULL value, at which point you would have fully traversed the tree and the element you were currently on would be the largest element. The worst-case potential complexity of such an operation would be O(log n) given we'd be checking one element per level of height. This could also be phrased as O(h) where h referred to the height of the tree.

3. [2] - 2

4. What are the advantages of an AVL tree over a BST? In what situations would you an AVL tree be preferred over a BST? In what situation would you not want to use an AVL tree? [3 marks]

First, one of the clear examples of a positive of an AVL tree over a BST is that in an AVL the worst case time complexity is O(log (n)) for operations (insertion, deletion, searching) whilst a BST has a worse case of complexity O(n) which is an obvious improvement for the AVL.

In most instances an AVL tree would be preferred over a BST, one such example being that in which you wish to search data. This would benefit from the usage of an AVL tree as the operation by virtue of the balancing will be substantially quicker than performing a search on a binary search tree given that a given branch could have a long series of nodes that are required to traverse (pathological linear tree) in a BST which would not occur in an AVL.

An instance in which you may wish to not use an AVL tree is in an instance in which you do not require to do searching and are inserting and deleting a lot of data, as the insertions and deletions of an AVL tree will mandate the calculation of balance and appropriate rotations which will increase the time taken for operations such that a BST will be faster. This is especially true with unsorted data where there will likely be multiple rotations performed per element inserted.

The AVL will also take up more slightly more memory when you consider it maintains either a height or a balance factor somewhere so if there's a memory constraint a binary search tree may be superior.

4. [3] - 3 really good.

5. What is the complexity of rotation operations in relation to the size of the tree? [1 mark]

The complexity of rotation operations would be O(log n), as it would be proportional to the tree's height, though it would be O(1) to perform the actual rotation operation when re-balancing was shown to be necessary. O(log n) + O(1) = O(log n).

5. [1] - 1 O(1) was in there somewhere, this is what we were looking for