Confronto Order Statistics

Alessandro Aldo Raoul Bonciani

Aprile 2024

Index

1 Introduction **2**
1.1 Description of the problem . . . 2
1.2 Test technical specs . . . 2

2 The data structures **2**
2.1 Sorted List . . . 2
2.2 BST . . . 2
2.3 ARN . . . 2

3 Order Statistics **3**
3.1 Order statistics in Sorted lists . . . 3
3.1.1 Pseudocodes . . . 3
3.2 Order statistics in BST . . . 4
3.2.1 Pseudocodes . . . 4
3.3 Order statistics in ARN . . . 4
3.3.1 Pseudocodes . . . 5

4 Expected results **5**

5 Tests and results **6**
5.1 Analysis of experimental results . . . 6
5.1.1 100 elements . . . 6
5.1.2 500 elements . . . 10
5.1.3 1000 elements . . . 14
5.1.4 1500 elements . . . 19

6 Test discussion and conclusions **23**

1 Introduction

1.1 Description of the problem

In this report we are going to compare the different implementations regarding dynamic order statistics of these data structures:

- BST without the size attribute
- Ordered List
- ARN

1.2 Test technical specs

The tests are all executed on the same machine, which runs on:

- **Processor:** Intel Core i7-7700HQ 2.80 GHz,
- **RAM:** 16.0 GB
- **Operating System:** Ubuntu 22.04.4 LTS

The code has been written and executed on Visual Studio Code (1.88.1) while this report has been written online using **OverLeaf**

2 The data structures

2.1 Sorted List

A sorted list is a data structure that maintains its elements in a particular order, typically either in ascending or descending order. This arrangement allows for efficient searching, insertion, and deletion operations. When new elements are added to a sorted list, they are inserted into the appropriate position to maintain the sorted order. Similarly, when elements are removed, the list adjusts to maintain the order.

2.2 BST

A **B**inary **S**earch **T**ree is a kind of binary tree in which each vertex can have up to two children. This structure adheres to the BST property, stipulating that every vertex in the left subtree of a given vertex must carry a value smaller than that of the given vertex, and every vertex in the right subtree must carry a value larger. To go through this data structure, we use the *inorderTreeWalk* method that works recursively. There are two versions for this method, one each for OS algorithm.

2.3 ARN

In comparison to those of the previous binary search trees, the nodes of red-black trees have additional attributes: color, useful during insertion to maintain the tree balanced, and *size*, representing the size of the subtree rooted at that node.

3 Order Statistics

Dynamic order statistics are a class of problems that require efficiently maintaining and querying a sequence of sorted elements. The OS-Select and OS-Rank algorithms are two operations of fundamental importance in this context:

- **OS Select(k)** allows finding the k-th smallest element in a sequence, returning its pointer.
- **OS Rank(x)** determines the position of an element x in the sorted sequence, returning the index.

3.1 Order statistics in Sorted lists

The OS Select algorithm for sorted lists has been implemented as follows: the list is traversed from the head until reaching the k-th element, which is returned by the function (if k exceeds the length of the list, a null value is returned).

For the OS Rank algorithm, a similar approach is taken: the list is traversed, comparing each node with the input node x provided to the function, while keeping track of how many nodes have been analyzed. Once the node x is found, the function outputs its position in the sorted sequence (or returns 0 if x does not belong to the list).

3.1.1 Pseudocodes

```
Input: An integer $i$
Output: Node at the $i$-th position in the list
OS_Select(i)
$j \leftarrow 1$
$tmpNode \leftarrow$ self.head
while $tmpNode \neq None \wedge j \neq i$ do
    $j \leftarrow j + 1$
    $tmpNode \leftarrow tmpNode.next$
end
return $tmpNode$
```

```
Input: A $node$
Output: The rank of the $node$ in the $List$
OS_Rank($node$)
$tmpNode \leftarrow$ self.head
$i \leftarrow 1$
while $tmpNode \neq None \wedge tmpNode.key \neq node.key$ do
    $tmpNode \leftarrow tmpNode.next$
    $i \leftarrow i + 1$
end
if $tmpNode$ è $None$ then
    $i \leftarrow 0$
end
return $i$
```

3.2 Order statistics in BST

Regarding the binary search tree, the operation is entirely analogous to that of sorted lists, with the only difference being that the tree is traversed not sequentially but through the inorderTreeWalk method, which operates recursively. There are two versions of this method, one for the OS algorithm, which differ in their termination criteria.

3.2.1 Pseudocodes

```
Input: An integer $i$
Output: Node at the $i$-th position in the BST
OS_Select(i)
self.osCounter ← 1
self.__inorderTreeWalkSelect(self.root, i)
self.osCounter ← 0
self.found ← False
$x$ ← self.rankedNode
self.rankedNode ← None
return $x$
```

```
Input: A node
Output: The rank of the node in the BST
OS_Rank(node)
self.osCounter ← 1
self.__inorderTreeWalkRank(self.root, node)
$i$ ← self.osCounter
self.osCounter ← 0
self.found ← False
if $i > self.size$ then
    $i \leftarrow 0$
end
return $i$
```

3.3 Order statistics in ARN

The *OS Select* algorithm starts from the root and compares the value of k with the *size* value of the left child: if they are equal, it means that the root is the k-th element, so the process ends; if the value of k is smaller, it means that the k-th element is in the left subtree, and the algorithm is recursively called on that subtree; if the value of k is greater, it means that the k-th element is in the right subtree, and the algorithm is called on that subtree, subtracting from k the *size* value of the left child and the root.

The *OS Rank* algorithm, instead, starts by calculating the size of the left subtree of the input node x. Then it traverses the tree up to the root: every time it encounters a node as the right child of its parent, it means that all nodes of the left subtree of the parent and the parent itself precede it, so in this case, a counter is incremented with the size of the left subtree of the parent plus 1; if instead the node is encountered as the left child of its parent, the counter is left unchanged. When the loop reaches the root of the tree, the value of the counter represents the position of the node x in the sorted sequence of nodes.

3.3.1 Pseudocodes

```
Input: Index $i$
Output: Node with rank $i$
$x \leftarrow root$;
found $\leftarrow$ False;
while $not\ found\ and\ x \neq None$ do
    $j \leftarrow (x.left.size$ se $x.left \neq$ None altrimenti $0) + 1$;
    if $j == i$ then
        found $\leftarrow$ True;
    end
    else if $i < j$ then
        $x \leftarrow x.left$;
    end
    else
        $x \leftarrow x.right$;
        $i \leftarrow i - j$;
    end
end
return $x$;
```

```
Input: Node $x$
Output: Rank of $x$
$i \leftarrow (x.left.size$ se $x.left \neq$ None altrimenti $0) + 1$;
$y \leftarrow x$;
while $y \neq self.root$ do
    if $y \neq y.p.right$ then
        $i \leftarrow i + (y.p.left.size$ se $y.p.left \neq$ None altrimenti $0) + 1$;
    end
    $y \leftarrow y.p$;
end
return $i$;
```

4 Expected results

Since the OS algorithms on ordered lists operate by traversing the elements of the list, we can expect a complexity of $O(n)$. The same reasoning can be applied regarding ABRs since the implementation logic is analogous. Finally, concerning ARNs, given that the *OS algorithms* traverse the tree vertically, we can expect a complexity of $O(h)$, which in balanced trees like red-black trees is equivalent to $O(lg(n))$.

It is also interesting to reflect on the different results obtained by requesting pointers or positioning indices of nodes at the top of the list compared to those at the bottom. In this regard, it is reasonable to assume that structures like ordered lists or unbalanced ABRs are faster when dealing with the first elements of the ordered sequence; conversely, balanced binary trees (including red-black trees, which are balanced by definition) should report similar performances regardless of the nodes/indices they have to work with.

5 Tests and results

We will analyze the behavior of the OS algorithms in cases where the data has been inserted sequentially or randomly (using the Python language's *random* library). We will conduct tests on arrays with:

- 100 elements
- 500 elements
- 1000 elements
- 1500 elements

To measure the execution time, we will use the `perf_counter()` function from the `time` library, which, when called before and after the execution of the algorithm, allows us to measure the execution times. In order to significantly speed up the execution times while still achieving satisfactory results, the performance measurement does not occur with every new element insertion, but only every $N_{\max}$ insertions, where $N_{\max}$ is an arbitrarily chosen interval.

In particular, the formula to calculate the execution times as a function of the number of elements is:

$$time_k = \frac{1}{N_k}\sum_{n=1}^{N_k}(end_n - start_n) + time_{k-1}$$

$$\text{where } time_0 = 0 \text{ and } k > 1$$

The first term represents the value returned by the `TimeCheck` methods, which is the average of the execution times over all elements inserted into the data structure up to step k (N_k).

The reason for choosing this approach is that in this way, it will be much easier to notice the asymptotic behavior of the algorithms' complexity in the graphs printed by the program.

5.1 Analysis of experimental results

5.1.1 100 elements

From these initial tests, it is possible to notice the linear behavior of the OS algorithms on ordered lists and binary search trees (BST), and the logarithmic behavior regarding red-black trees (ARN). It can also be noted how, even for relatively small data structures, the performance of the algorithms on binary search trees is significantly worse compared to cases of lists and red-black trees, which in this initial scenario report similar performances (especially regarding the OS Select function).

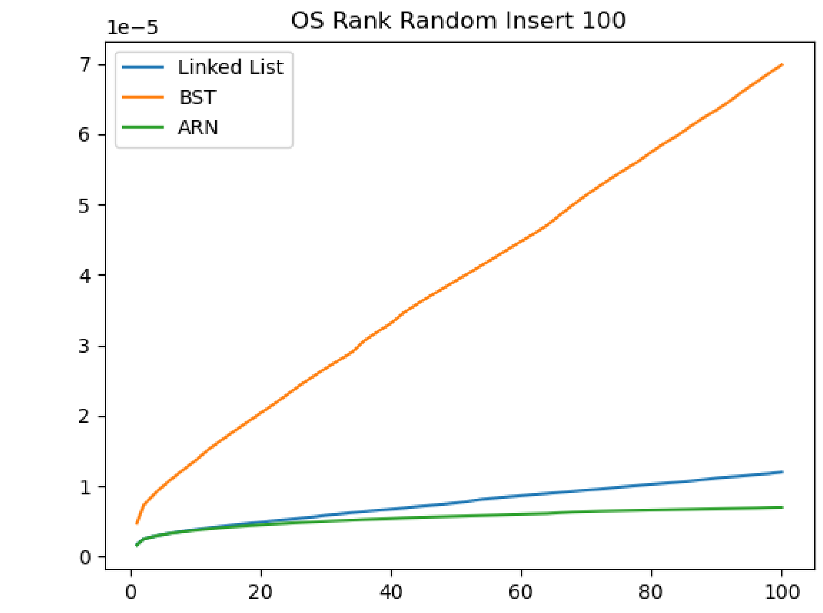


Figure 1: OS Rank Random Insert 100

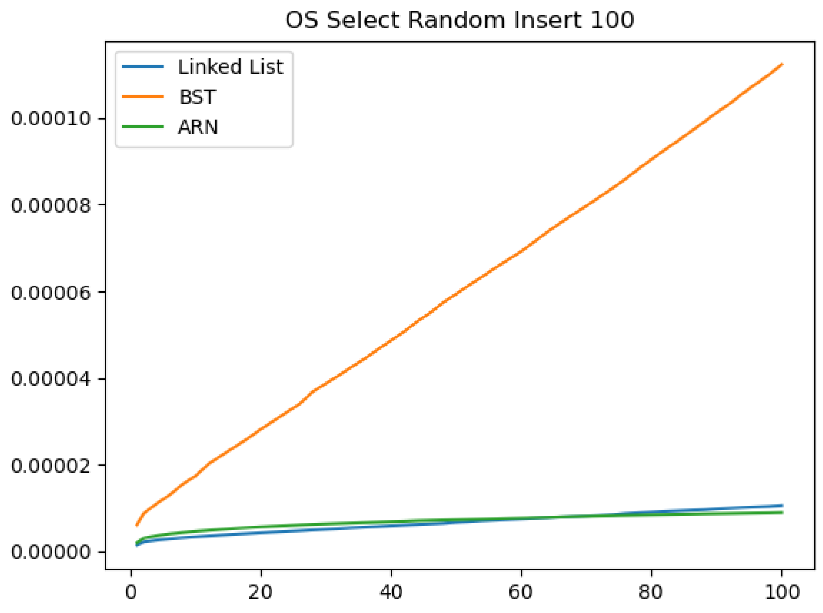


Figure 2: OS Select Random Insert 100

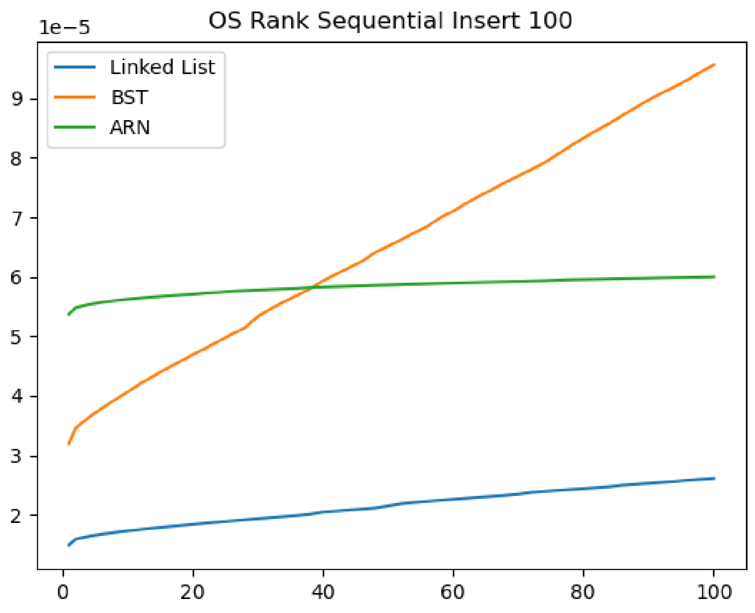


Figure 3: OS Rank Sequential Insert 100

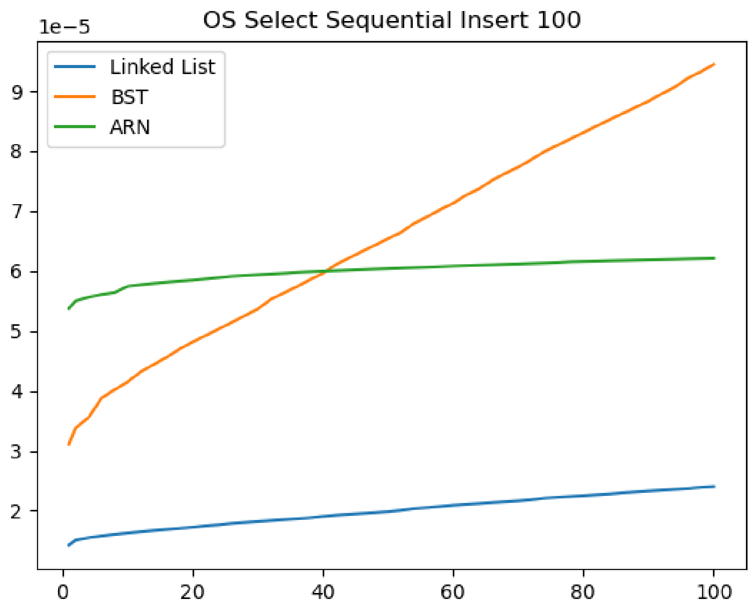
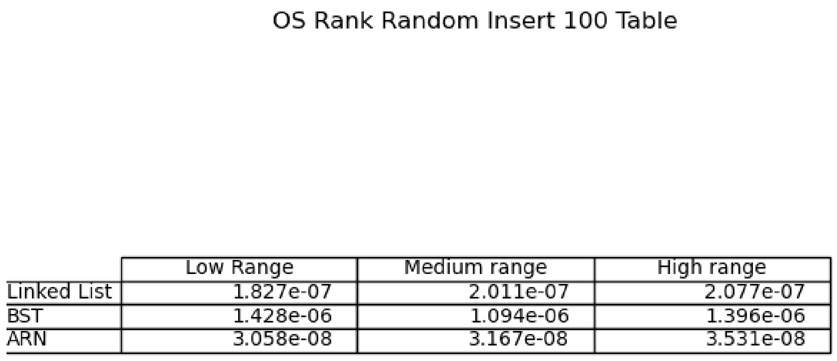
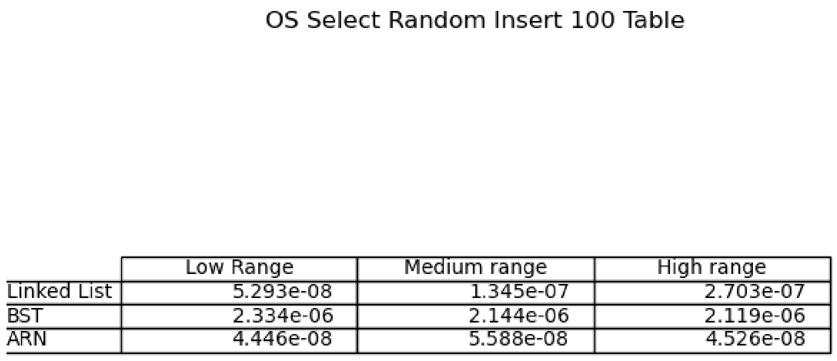


Figure 4: OS Select Sequential Insert 100


OS Rank Random Insert 100 Table

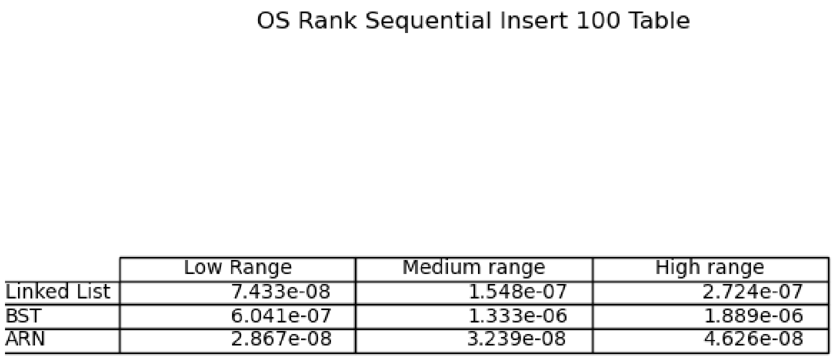
	Low Range	Medium range	High range
Linked List	1.827e-07	2.011e-07	2.077e-07
BST	1.428e-06	1.094e-06	1.396e-06
ARN	3.058e-08	3.167e-08	3.531e-08

Figure 5: OS Rank Random Insert 100 Table


OS Select Random Insert 100 Table

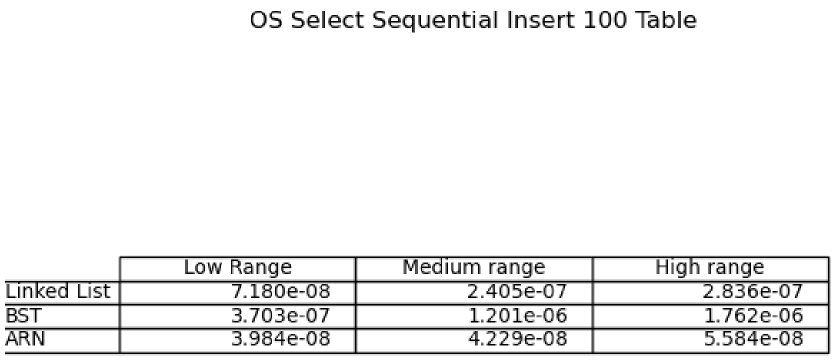
	Low Range	Medium range	High range
Linked List	5.293e-08	1.345e-07	2.703e-07
BST	2.334e-06	2.144e-06	2.119e-06
ARN	4.446e-08	5.588e-08	4.526e-08

Figure 6: OS Select Random Insert 100 Table


OS Rank Sequential Insert 100 Table

	Low Range	Medium range	High range
Linked List	7.433e-08	1.548e-07	2.724e-07
BST	6.041e-07	1.333e-06	1.889e-06
ARN	2.867e-08	3.239e-08	4.626e-08

Figure 7: OS Rank Sequential Insert 100 Table


OS Select Sequential Insert 100 Table

	Low Range	Medium range	High range
Linked List	7.180e-08	2.405e-07	2.836e-07
BST	3.703e-07	1.201e-06	1.762e-06
ARN	3.984e-08	4.229e-08	5.584e-08

Figure 8: OS Select Sequential Insert 100 Table

5.1.2 500 elements

In this case, we will choose $K = 100$. Figures 9 - 12 confirm the hypotheses formulated in the previous case regarding the linear behavior (for lists and ABR) and logarithmic behavior (for ARN). However, it can be observed that for larger data structures, the implementation on ARN is faster compared to that on ordered lists, while the gap with the approach that uses BST starts to widen increasingly.

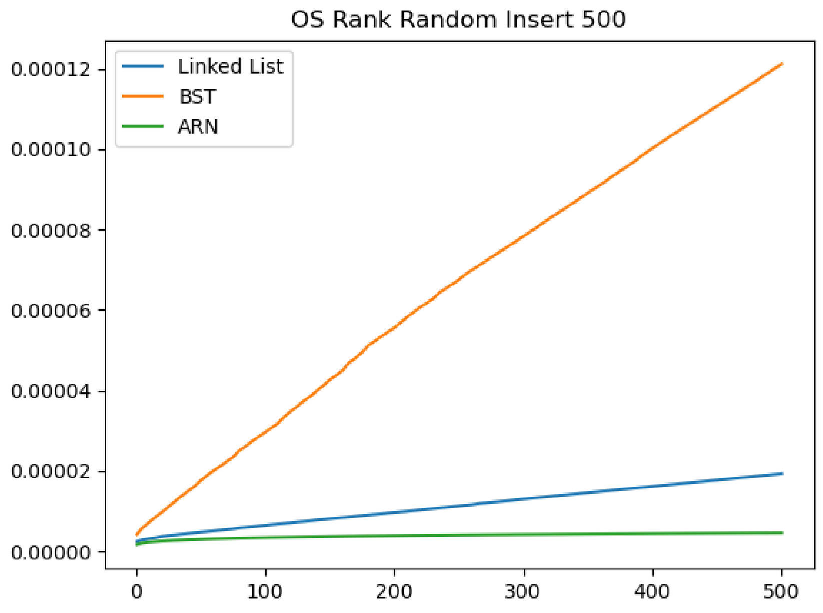


Figure 9: OS Rank Random Insert 500

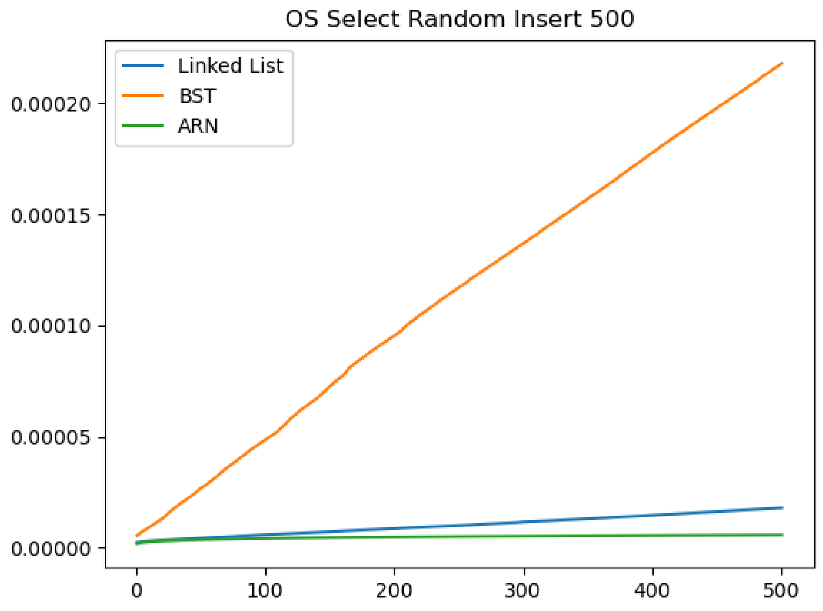


Figure 10: OS Select Random Insert 500

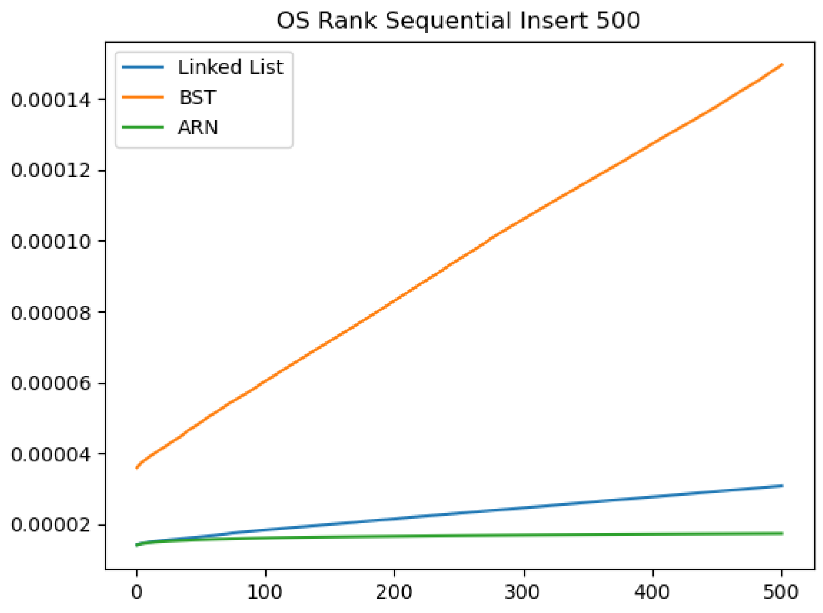


Figure 11: OS Rank Sequential Insert 500

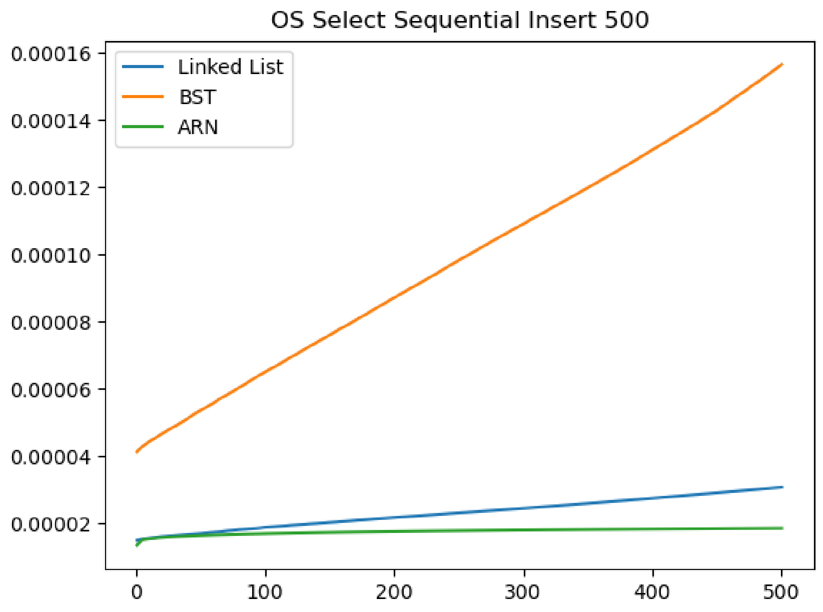
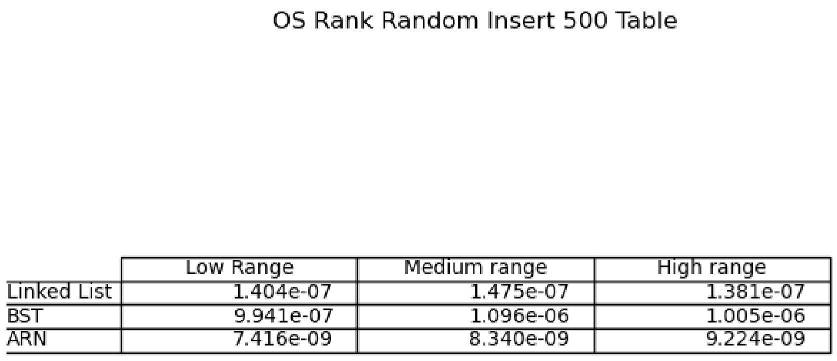


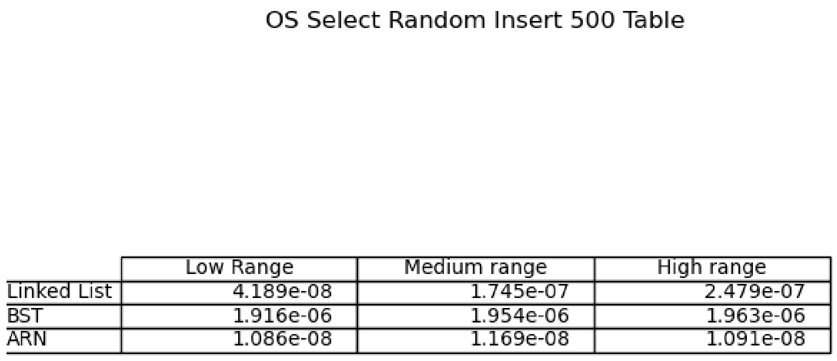
Figure 12: OS Select Sequential Insert 500



OS Rank Random Insert 500 Table

	Low Range	Medium range	High range
Linked List	1.404e-07	1.475e-07	1.381e-07
BST	9.941e-07	1.096e-06	1.005e-06
ARN	7.416e-09	8.340e-09	9.224e-09

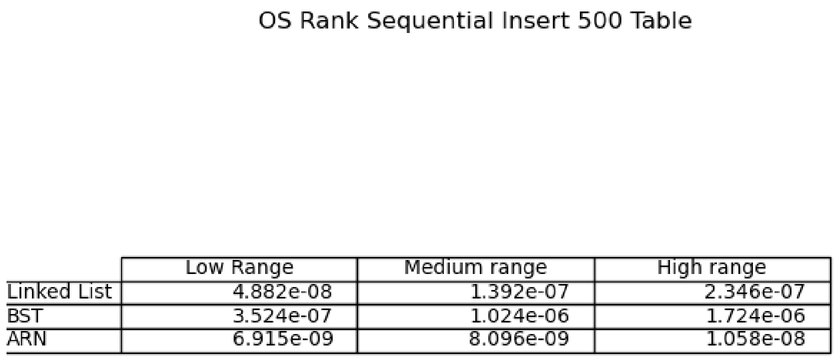
Figure 13: OS Rank Random Insert 500 Table



OS Select Random Insert 500 Table

	Low Range	Medium range	High range
Linked List	4.189e-08	1.745e-07	2.479e-07
BST	1.916e-06	1.954e-06	1.963e-06
ARN	1.086e-08	1.169e-08	1.091e-08

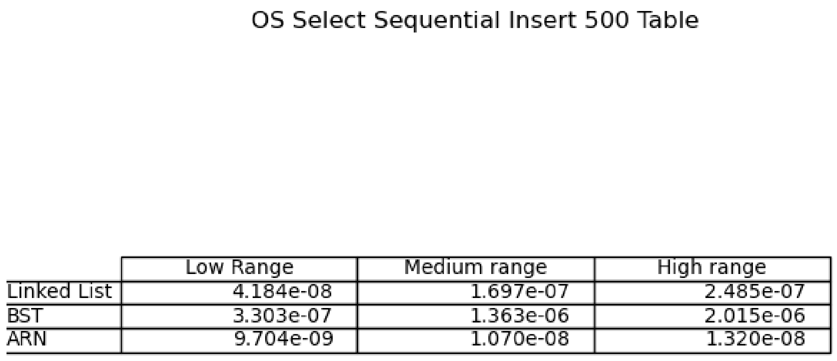
Figure 14: OS Select Random Insert 500 Table



OS Rank Sequential Insert 500 Table

	Low Range	Medium range	High range
Linked List	4.882e-08	1.392e-07	2.346e-07
BST	3.524e-07	1.024e-06	1.724e-06
ARN	6.915e-09	8.096e-09	1.058e-08

Figure 15: OS Rank Sequential Insert 500 Table



OS Select Sequential Insert 500 Table

	Low Range	Medium range	High range
Linked List	4.184e-08	1.697e-07	2.485e-07
BST	3.303e-07	1.363e-06	2.015e-06
ARN	9.704e-09	1.070e-08	1.320e-08

Figure 16: OS Select Sequential Insert 500 Table

5.1.3 1000 elements

In this case, we'll also choose $K = 100$. Having verified in previous cases the complexity of the different implementations of the OS algorithms, in this instance, we'll only comment on the graphs that allow us to compare them. By observing Figures 17-20, it can be noted that for large data structures, the OS algorithms perform much better in the case of ARNs with the 'size' attribute, while the gap in terms of execution time between the implementation on BST and those on other data structures is even more evident.

Now, let's analyze the average execution times for the three ranges of indices/nodes entering the OS Select and OS Rank functions, as observable in Figures 21-24. We immediately notice that the performances in the case of ARNs are similar in all three ranges, and this situation doesn't change even if the data insertion is randomized, since the insert-Fixup() function precisely maintains the tree balanced with each new insertion.

In the case of ordered lists, on the other hand, we notice how the algorithms are faster in the lower range and slower in the higher one; this is because the indices/nodes are searched starting from the first ones. Few differences, instead, in the case of random insertion, since, being the list ordered, the order of the nodes in the list is not influenced by their insertion order.

As for BSTs with sequential insertion, we can make a similar argument to the one made for lists, since the implementations of the OS algorithms on these data structures are based on the same underlying idea (we can also think that a completely unbalanced binary tree becomes a list). In the case of random insertion, however, we find ourselves with a definitely more balanced tree, and this is also reflected in the performances, which, as in the case of ARN, are now more balanced.

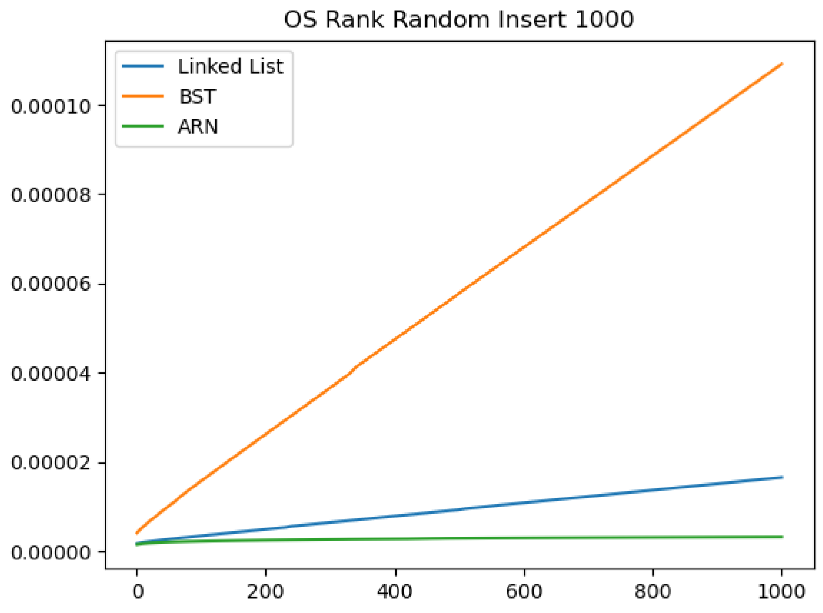


Figure 17: OS Rank Random Insert 1000

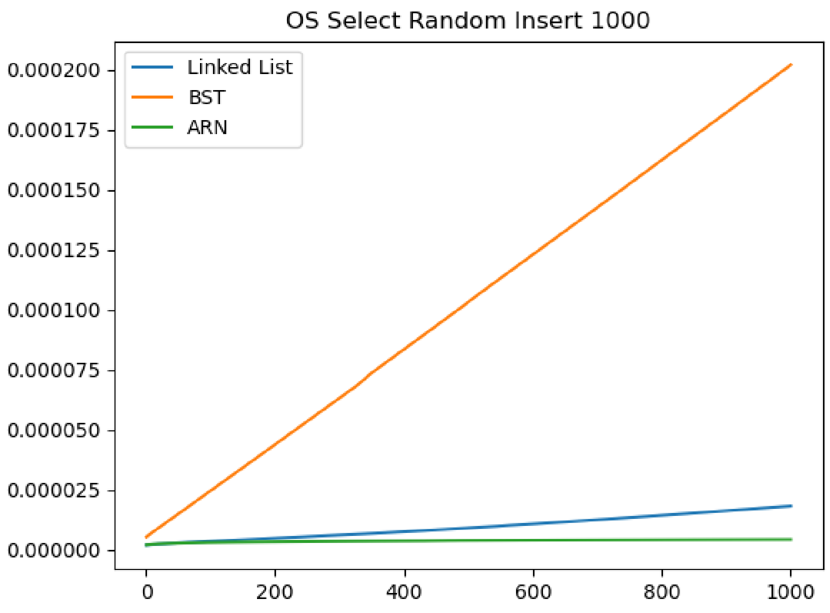


Figure 18: OS Select Random Insert 1000

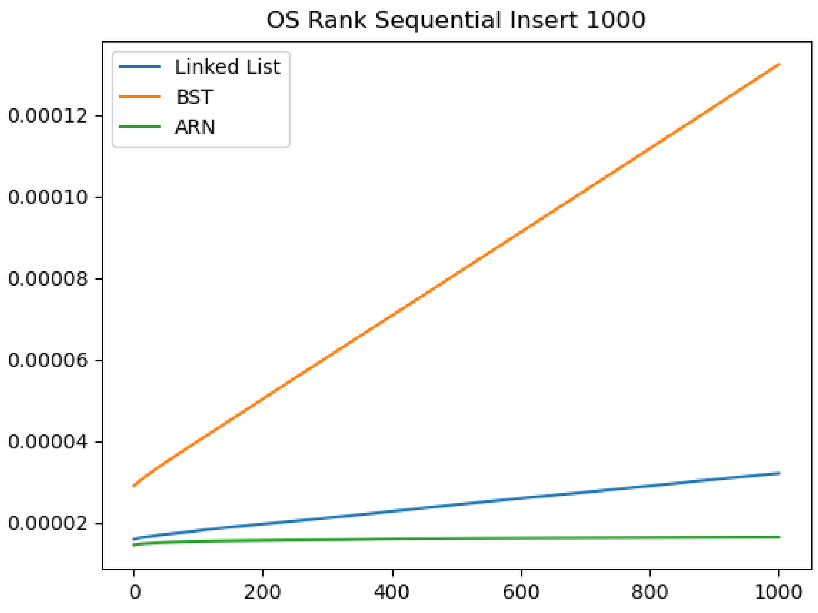


Figure 19: OS Rank Sequential Insert 1000

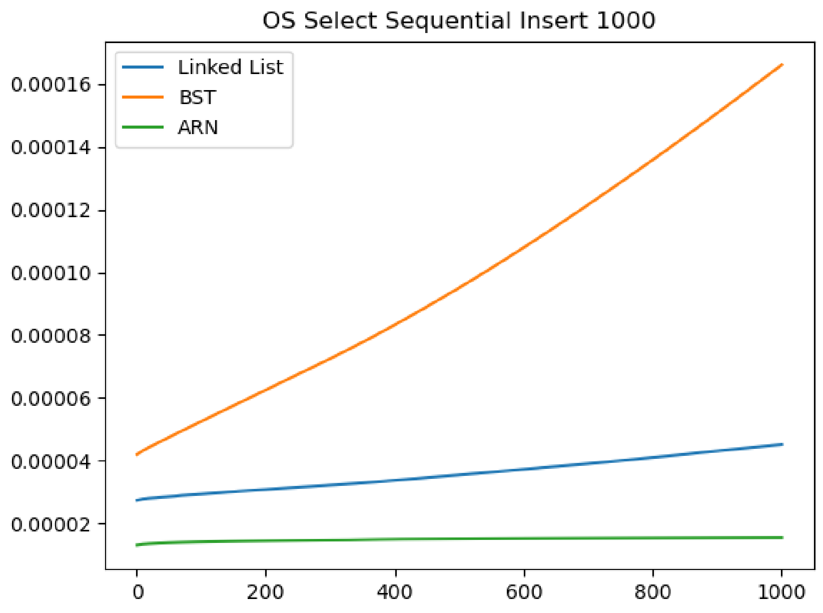
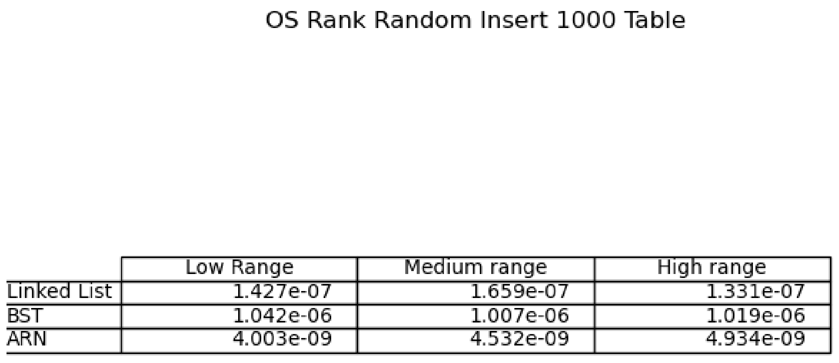


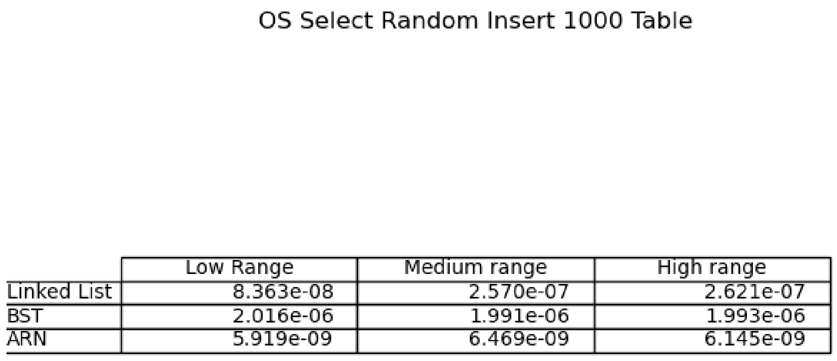
Figure 20: OS Select Sequential Insert 1000



OS Rank Random Insert 1000 Table

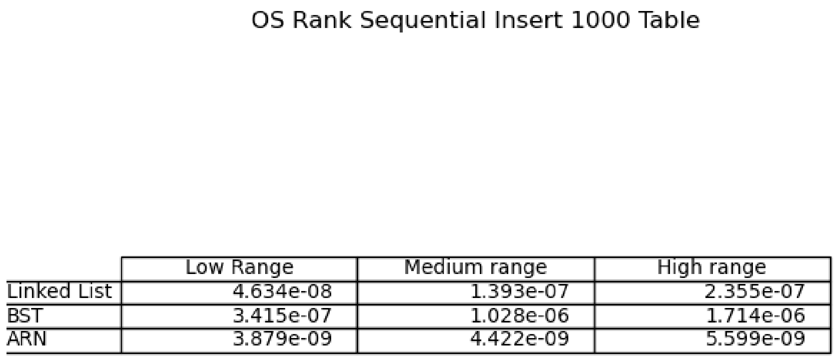
	Low Range	Medium range	High range
Linked List	1.427e-07	1.659e-07	1.331e-07
BST	1.042e-06	1.007e-06	1.019e-06
ARN	4.003e-09	4.532e-09	4.934e-09

Figure 21: OS Rank Random Insert 1000 Table


OS Select Random Insert 1000 Table

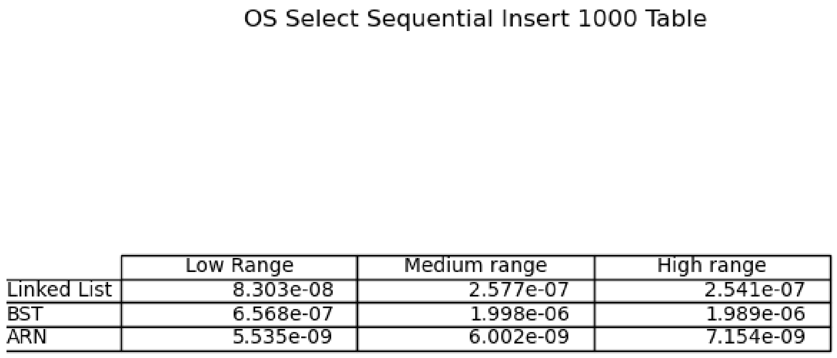
	Low Range	Medium range	High range
Linked List	8.363e-08	2.570e-07	2.621e-07
BST	2.016e-06	1.991e-06	1.993e-06
ARN	5.919e-09	6.469e-09	6.145e-09

Figure 22: OS Select Random Insert 1000 Table


OS Rank Sequential Insert 1000 Table

	Low Range	Medium range	High range
Linked List	4.634e-08	1.393e-07	2.355e-07
BST	3.415e-07	1.028e-06	1.714e-06
ARN	3.879e-09	4.422e-09	5.599e-09

Figure 23: OS Rank Sequential Insert 1000 Table


OS Select Sequential Insert 1000 Table

	Low Range	Medium range	High range
Linked List	8.303e-08	2.577e-07	2.541e-07
BST	6.568e-07	1.998e-06	1.989e-06
ARN	5.535e-09	6.002e-09	7.154e-09

Figure 24: OS Select Sequential Insert 1000 Table

5.1.4 1500 elements

For this final test, it was necessary to allocate more memory to VisualStudio to handle the high number of recursive calls.

In this case as well, we will only observe the graphs that compare the performances on different data structures (Figures 18 and 19), printed by choosing $K = 300$.

These last tests only confirm the observations made in previous cases, showing how the implementation on BST becomes less performant as the input sizes increase, while the implementation on ARN maintains satisfactory execution times even in the face of much larger data structures.

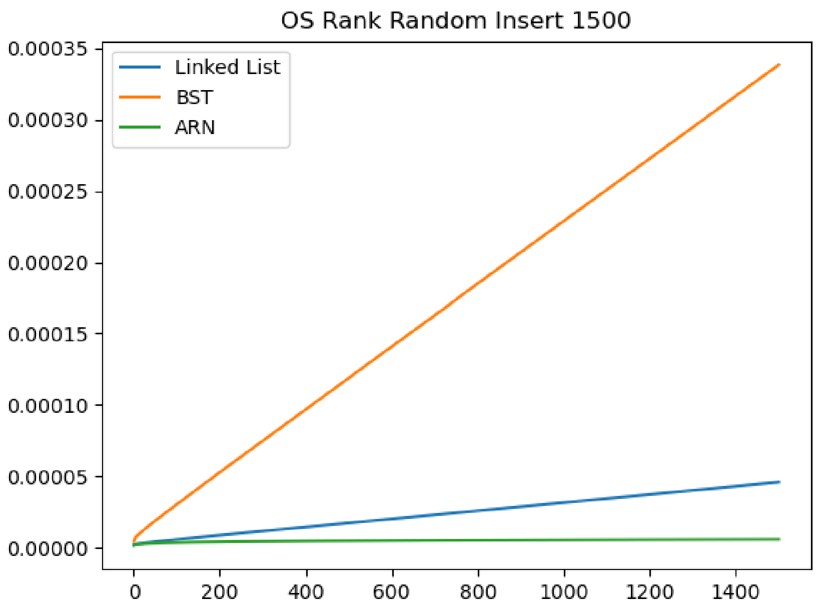


Figure 25: OS Rank Random Insert 1500

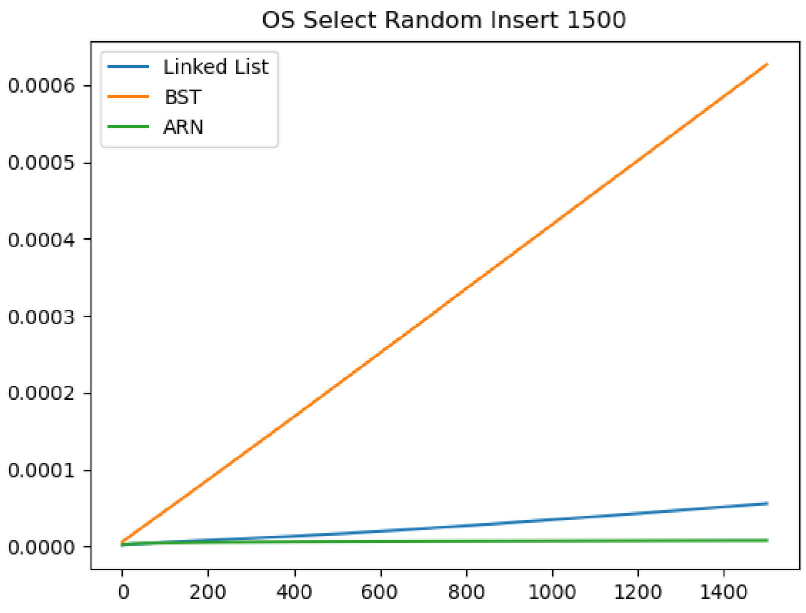


Figure 26: OS Select Random Insert 1500

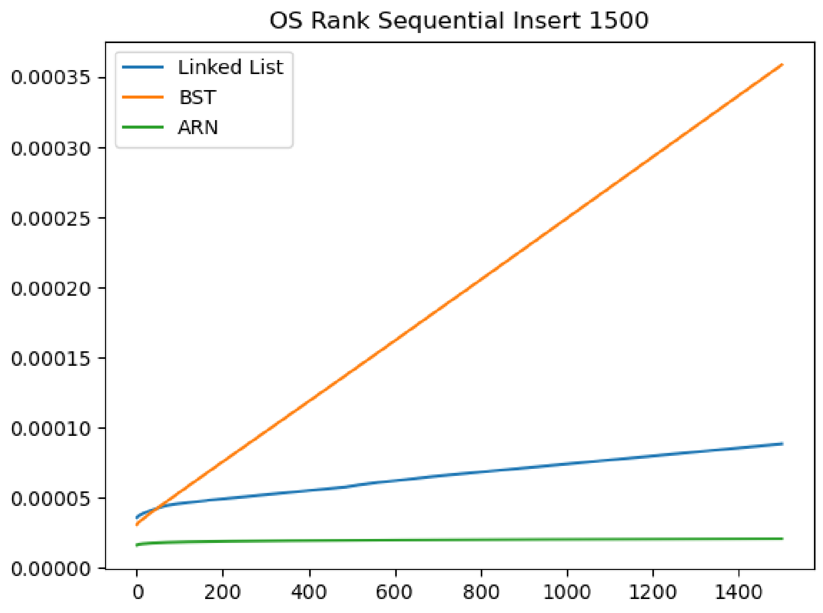


Figure 27: OS Rank Sequential Insert 1500

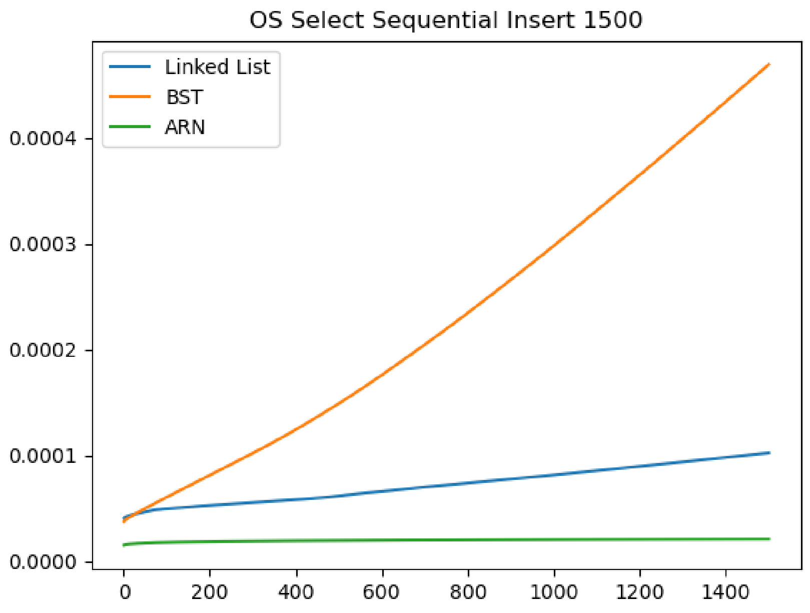
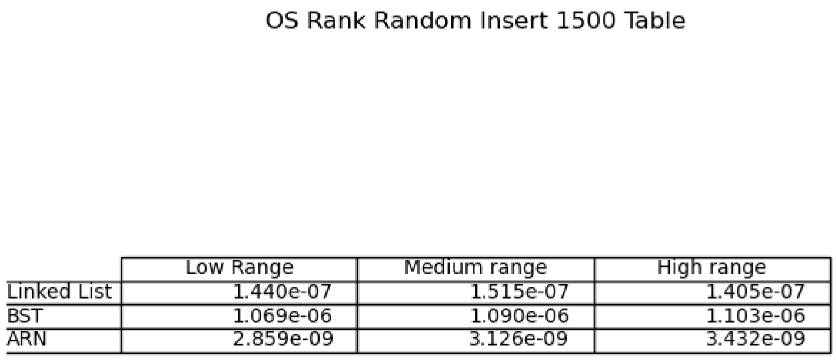


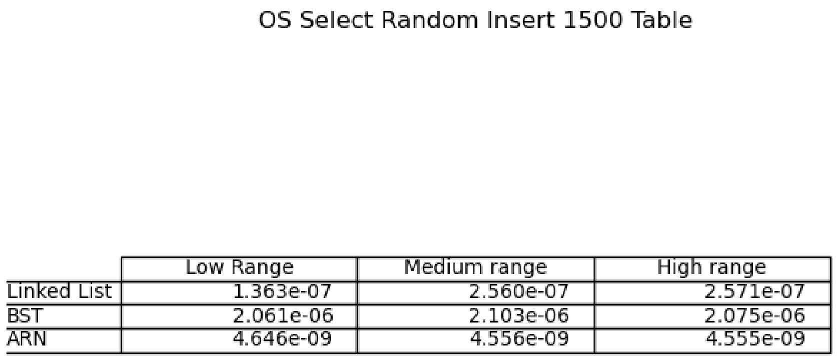
Figure 28: OS Select Sequential Insert 1500



OS Rank Random Insert 1500 Table

	Low Range	Medium range	High range
Linked List	1.440e-07	1.515e-07	1.405e-07
BST	1.069e-06	1.090e-06	1.103e-06
ARN	2.859e-09	3.126e-09	3.432e-09

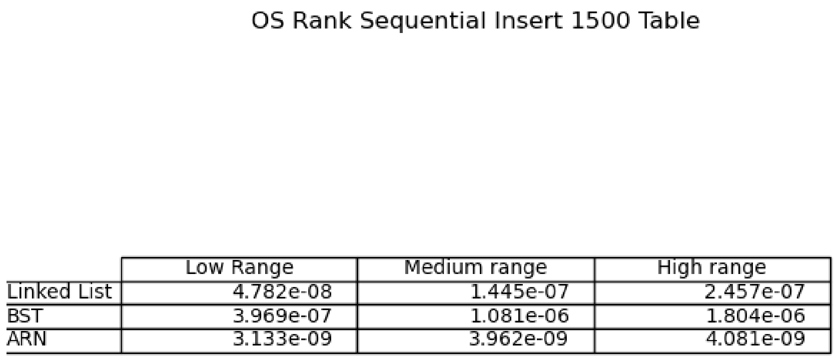
Figure 29: OS Rank Random Insert 1500 Table



OS Select Random Insert 1500 Table

	Low Range	Medium range	High range
Linked List	1.363e-07	2.560e-07	2.571e-07
BST	2.061e-06	2.103e-06	2.075e-06
ARN	4.646e-09	4.556e-09	4.555e-09

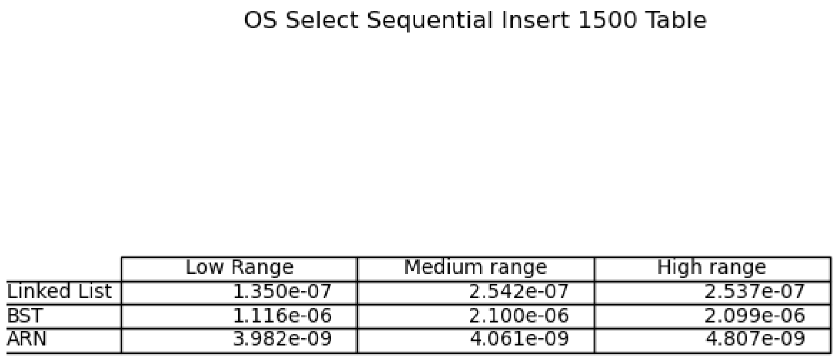
Figure 30: OS Select Random Insert 1500 Table



OS Rank Sequential Insert 1500 Table

	Low Range	Medium range	High range
Linked List	4.782e-08	1.445e-07	2.457e-07
BST	3.969e-07	1.081e-06	1.804e-06
ARN	3.133e-09	3.962e-09	4.081e-09

Figure 31: OS Rank Sequential Insert 1500 Table



OS Select Sequential Insert 1500 Table

	Low Range	Medium range	High range
Linked List	1.350e-07	2.542e-07	2.537e-07
BST	1.116e-06	2.100e-06	2.099e-06
ARN	3.982e-09	4.061e-09	4.807e-09

Figure 32: OS Select Sequential Insert 1500 Table

6 Test discussion and conclusions

The results of the tests conducted confirm the hypotheses made previously regarding the complexity of the OS algorithms in the different implementations: we are talking about

$O(n)$ for ordered lists and search trees and $O(lg(n))$ for ARNs, regardless of the type of insertion adopted.

The implementation for ARNs with the size attribute proves to be the most performant in general and reports similar performances regardless of the requested node/index.

Ordered lists, on the other hand, are a valid choice for data structures of small sizes, particularly fast when dealing with the first elements of the list.

Finally, the implementation for BSTs without the size attribute has proven to be the least effective, as the absence of this parameter makes it impossible to exploit the potential of the data structure and instead forces a much less elaborate approach similar to that adopted for lists. The performances on BSTs are much worse due to the time spent descending and ascending the tree through recursive calls, and even when the tree is more balanced, the improvement in terms of performance is relative.