UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Mandatory Assignment

INF-1101 - Data structures and algorithms
Department of Physics and technology
Christian Salomonsen

1 Introduction

This task involves implementing two sets using different data structures: a linked list implementation and a Binary Search Tree (BST) implementation [Wikipedia, 2023c]. We will then use the sets to create a spam filter, utilizing the union, difference, and intersection operations with pre-classified spam and non-spam emails. Finally, we will compare the two set implementations to evaluate their performance.

2 Technical Background

The focus of the assignment is to implement an ADT [Wikipedia, 2023a] for a set, which allows the user to interact with the program without knowing the internal data storage.

The pre-code provided for the assignment provides multiple C-source files for verifying that the output of the set implementation is correct. We use Make [Free Software Foundation, 2023b] to build the source code into an executable.

Provided with the specific implementations, there will also be provided a GNU Bash [Free Software Foundation, 2023a] script for quickly comparing the outputs of the spam-filter. A Python file for data wrangling and visualizing the results is also provided.

2.1 The set criteria

A set as mentioned in the introduction is a datatype, which contain unique ordered comparable elements. The set datatype is well suited with mathematical operations like union, intersection and difference, which each on their own produce a new set, which will be either a subset between multiple sets or the empty set, $\emptyset$.

Given two sets A and B, the operations mentioned can be further described as:

1. Union: A $\cup$ B produces a new set of all elements in A and B.
2. Intersection: A $\cap$ B produces a new set of only the elements that are both in A and B.
3. Difference: A - B produces a new set of elements in A that are not in B.

Additionally, we also need ways of adding elements to the sets, checking if an element is contained in a set, creation and destruction, etc. but these implementations are specific to the underlying datatype, which handles storage of elements and will be discussed in the next sections.

3 Design and implementation

The following section is divided into two parts: one focuses on the implementation of the set using the API provided by the linked list source code, and the other on creating a new implementation for a set using a binary search tree to store elements.

3.1 Set based on a linked list

In the current implementation, a doubly linked list is used as the backend for the set. As the elements are not arranged in any specific order, the linked list requires ordering methods. The linked list is a simple yet powerful data structure that allows for traversal in both directions. However, lookups can be slow for large sets of elements, while iteration is quick and does not require nesting in the inner structure of the data structure.

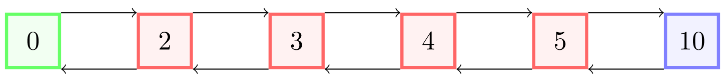

Figure 1: *Doubly linked list*

3.1.1 Adding a new element

When adding a new element to the set, two steps are performed. Firstly, the algorithm checks whether the element already exists in the list by iterating over each element until either the element is found or the end of the list is reached. Secondly, the element is added either to the start or end of the list. It is worth noting that the list could also be ordered by sorting it every time a new element is inserted. However, this approach is not always necessary.

Although there is an optimization that could be implemented where the algorithm only compares elements up until the first occurrence of another element greater than the element being searched for, this approach was not chosen due to the condition that the linked list always had to be sorted. Instead, a different optimization was chosen for the current implementation.

3.1.2 Iterating over the set

To iterate over the set based on the linked list, follow these steps:

1. Set the current element to the head of the list.
2. Check if the next pointer exists.
3. If it does, retrieve the element of the next pointer and set it as the current element.
4. Repeat steps 2-3 until there are no more elements.

Note that since the linked list is not ordered, and the implementation may not sort the list after each element is added, it is necessary to sort the list when creating a set iterator. This approach saves computation time, and is sufficient for this specific assignment since the set only needs to be ordered when generating the iterator. However, if the source code is applied in a different context, it may be necessary to make a different choice.

3.2 Set based on a binary search tree

The provided implementation of a binary search tree in this assignment does not use a balanced tree algorithm like the AVL tree [Wikipedia, 2023b]. Instead, it relies on simple comparison operations to place elements in the tree. This means that the structure can be very sensitive to the order of the input elements. For instance, if the elements are inserted in order, the tree would resemble a linked list. One way to avoid this issue is to ensure that the first element inserted into the tree is the median value of all the data, but this requires prior knowledge of the data. Another option would be to create a new tree for each element added and use the median value as the root node, but this would be less efficient.

In this implementation, each node stores a pointer to its two children (located below and to the left or right) and a pointer to its parent node. The tree starts with a root node, and elements are added to the tree by comparing them with existing nodes and placing them accordingly. The binary nature of the tree means that each node can store at most two elements. However, this is a simplification of a more general tree structure that can categorize input by multiple features rather than just a single value.

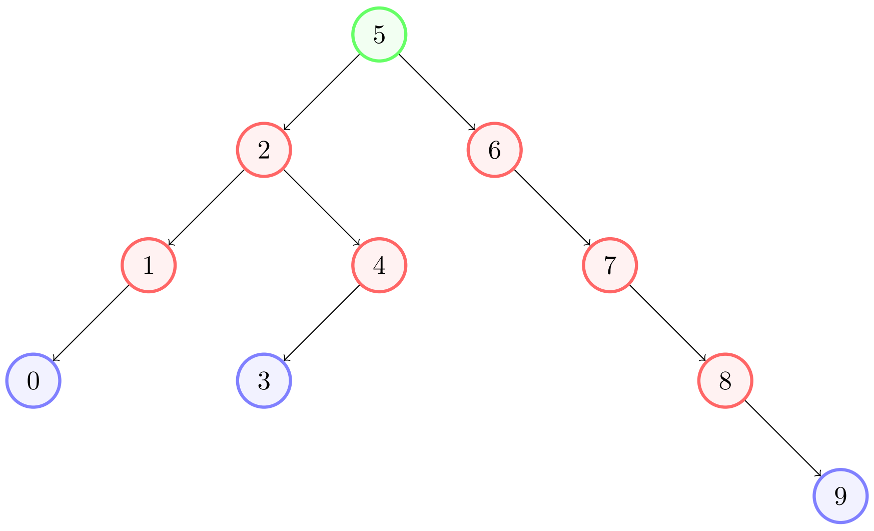

Figure 2: *Binary tree visualization*

3.2.1 Adding a new element

The binary search tree is a data structure that organizes data in a tree-like structure. Each node in the tree has at most two children and is structured in a way such that the left child is less than the parent node, and the right child is greater. Traversing the tree is done by moving left or right from the root node based on whether the element being searched for is less than or greater than the current node. To add an element, the algorithm compares it to the root node and moves left or right depending on the comparison result until it finds an empty spot to add the new node. The binary search tree is efficient for large datasets as it eliminates the need to compare with many elements of the other direction sub-tree. Additionally, the tree is ordered by default when new elements are added, so no additional sorting algorithms are required.

3.2.2 Iterating over the set

The process of iterating over a set that uses a BST can be more challenging than adding to it, requiring careful consideration of the structure. To traverse a binary search tree in an ordered sequence, the following steps can be taken:

1. Begin by setting the current node to be the smallest element (i.e., the leftmost node).
2. If the current node has a right node, move to the leftmost node of that right node.
3. If the current node is the right child of its parent, move up one step to the parent node.
4. Repeat step 3 until this condition is no longer true.

5. Set the current node to be its parent.
6. Repeat steps 2-5 until the largest (rightmost) node is reached.

By following this algorithm, the tree can be traversed in an ordered manner.

3.3 Set operation implementation

As previously explained, the set data structure supports the following operations: union, intersection, and difference. These operations are implemented identically in both `set.c` and `set_bst.c`. Here's how they work:

- Union: Create a copy of one of the sets and iterate over the other set. Add each element to the copy. Since sets do not allow duplicates, only unique values will be added.
- Intersection: Create an empty set and iterate over one of the input sets. For each element, check if it is also present in the other input set. If so, add the element to the new set.
- Difference: Create an empty set, iterate over the **first** set. If the element is not present in the second set, add it to the new set. This implementation is the negation of the intersection implementation.

By following these steps, the set operations can be efficiently performed on any implementation of the set data structure.

4 Evaluation

The following sections will discuss how the two versions of the set implementation, by using a doubly linked list versus a binary search tree as the back-end of the set.

Note that the results which will be presented in this section does not necessarily show the whole picture of how the structures compare. This is because of when for instance adding elements to the sets those values are drawn from a uniform distribution, hence duplicates are very likely to exist. We know that duplicate values are discarded, therefore at a particular horisontal axis value the cardinalities are likely different, hence the comparisons are not truly fair. On the contaty the cardinalities at a certain value will follow a normal distribution and will give a somewhat accurate depiction of the time taken.

4.1 Adding elements to the set

In this section, we evaluate the experimental results of timing the adding operation over the two set data structures. While the binary search tree (BST) is generally expected to outperform the linked list in terms of inserting new elements due to its ability to do fewer comparisons, there are some scenarios where the linked list can perform better.

For instance, if the size of the list is small, adding a new element to the linked list simply involves iterating over the few elements and adding the new element at an arbitrary position. In contrast, the binary search tree needs to traverse the tree structure to find the appropriate position to store the new element, incurring some overhead. However, the binary search tree will outperform the linked list as long as the values are not already ordered. When the tree is fed already ordered values, it effectively becomes a linked list, and the performance suffers.

When using the same optimization as in this assignment, which involves sorting the list only once the iterator is generated, the main bottleneck of the linked list implementation is checking whether the list already contains a new element. This requires iterating over each element, which, although fast in itself, adds up when performed for every new element added. Let us analyze this in more detail.

Suppose we have a linked list with n elements, denoted as $N = i_1, i_2, \ldots, i_n$, and we want to add k new elements $j_1, j_2, \ldots, j_k$ to the list. To insert the first new element, we must perform the following steps:

1. Iterate over all elements in the list to see if the value already exists, as we do not allow for duplicates.
2. Add the new element at the start of the list if and only if it is unique to the set.

In step 1, we perform n lookups. For the next iteration of adding, we need to look up $n+1$ elements, and for the next, $n+2$, and so on, until during the last iteration, we need to look up $n+(k-1)$ elements. Thus, to add all k elements, we perform a total of

$$m = \sum_{o=1}^{k} n + o = n + \frac{k(k-1)}{2} \tag{1}$$

lookups, where m is the total number of lookups needed to add k elements to the set N.

If we assume $N = \emptyset$, then the complexity of adding k elements is $m = \frac{k(k-1)}{2}$. In the limit as k approaches infinity, this complexity can be approximated by $m = \frac{k^2}{2}$, or just $c \cdot k^2$. Note, however, that this is the time complexity for adding k elements, not for adding a single new element, which has time complexity equal to the size of the superset D of $j_i \in K$ that are also in N. We can define $D = N \cup (N \cap K)$, where $|D| = |N| + |N \cap K| = d$. Thus, the time complexity of adding a single new element is d.

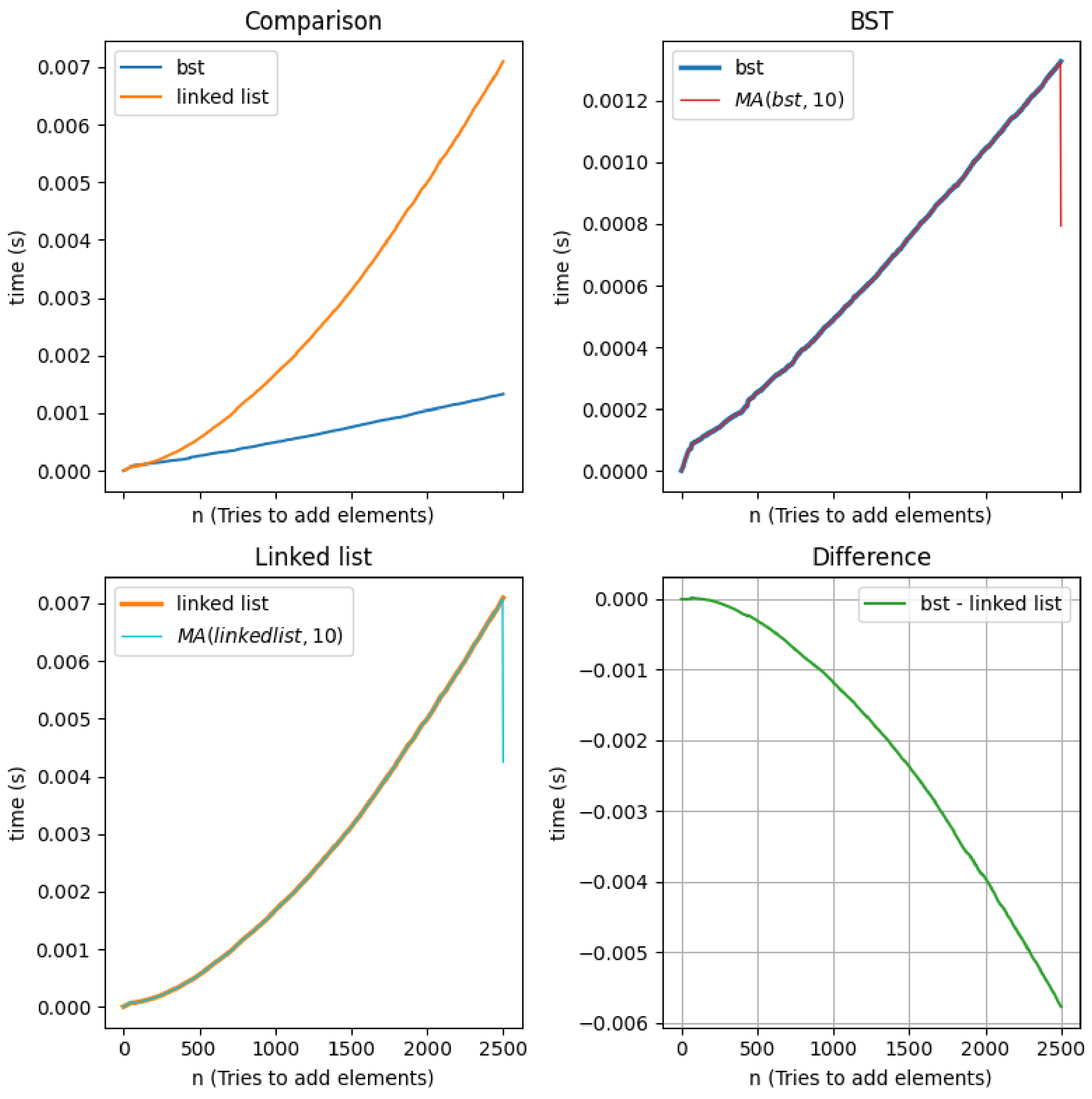

Figure 3: *Comparison in time taken to add elements to set abstract data type.*

Figure 3 displays the comparison between a BST and a linked list, by sequentially adding one and one element to the different sets.

As recently discussed the 'overhead' of the binary search tree is seen in the very start of the upper left plot of figure 3. Furthermore we see the time-complexity of adding multiple elements rise with what looks to be the square of the number of elements. On the contrary, the BST has the characteristics of following a linear time-complexity.

4.2 Union between sets

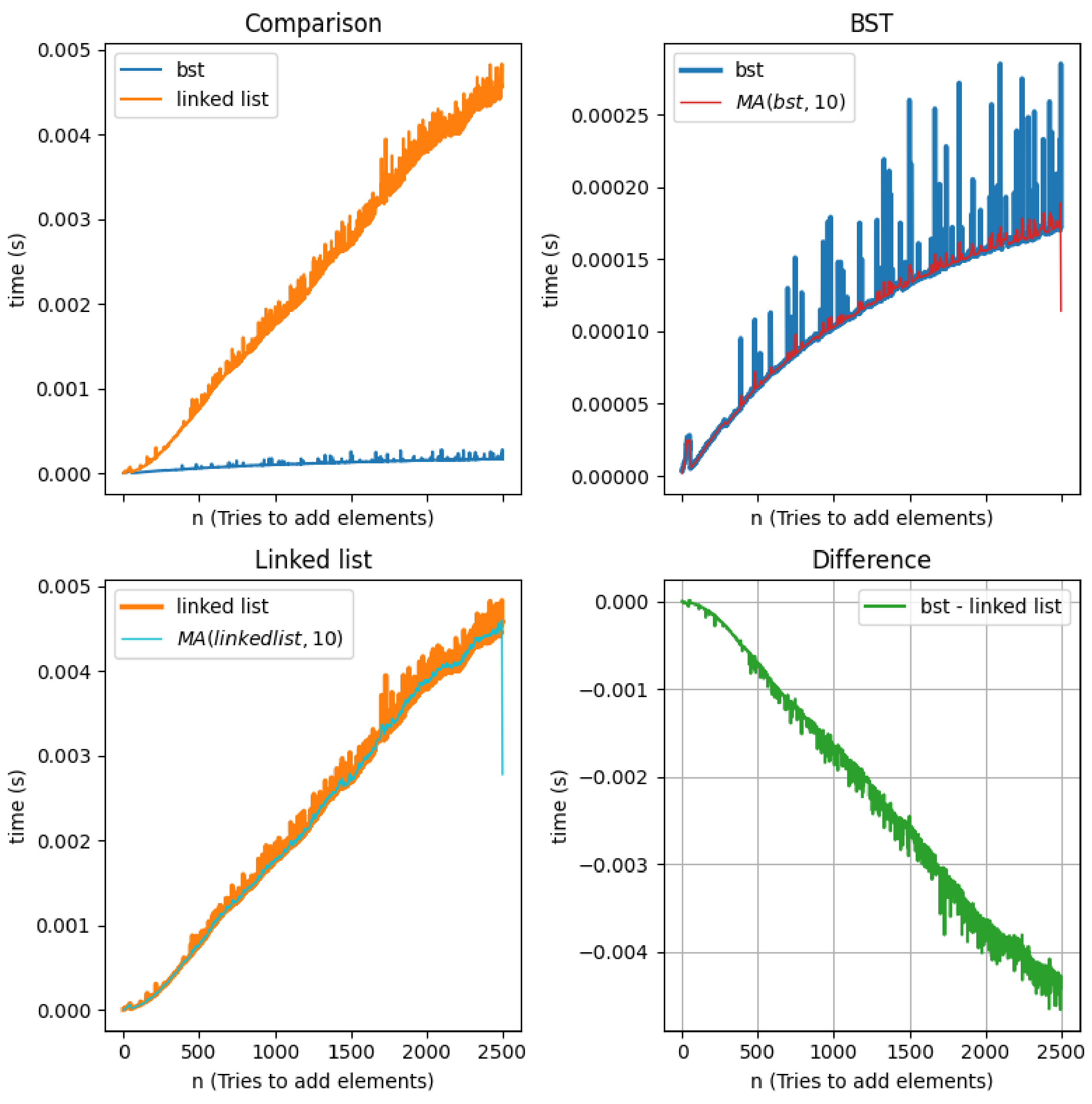

Figure 4: *Comparison of union between sets.*

The union operation between sets is the first set operation we will discuss. Figure 4 shows the performance comparison between two implementations: a binary search tree and a linked list.

The binary search tree outperforms the linked list, which is likely due to its faster element lookup times. Whenever an element is copied from one set to the other, the algorithm checks if the value already exists. This is where the binary search tree has the clear edge, being able to look up its elements much faster than the linked list. For a balanced binary search tree, the number of comparisons done will be at most the total depth of the tree.

We may compute the depth of the tree assuming we have a perfectly balanced tree. Because the tree grows with two new nodes per node of the previous depth, we find that for a tree with depth b we have 2^b nodes in total. Then we may approximate the depth of our tree given $n = 2500$ unique values to be,

$$\begin{aligned} n = 2^b &\iff \log_2 n = b, \quad n > 0 \\ b &= \log_2 2500 = \underline{11.2877} \end{aligned}$$

Therefore the worst case scenario for the perfectly balanced tree would require at most 12 comparisons.

However, we cannot assume the binary search tree in this implementation is balanced, and the worst case scenario is very likely worse than 12 comparisons per iteration. In fact, there is no guarantee that the binary search tree is not effectively a linked list, making the true worst case scenario in this implementation potentially $n = 2500$ comparisons.

Despite this, looking at the plot of the binary search tree in figure 4, we see a logarithmic function shape. Therefore, even though the worst case scenario may be 2500 comparisons, in this case, it is not.

4.2.1 Difference between sets

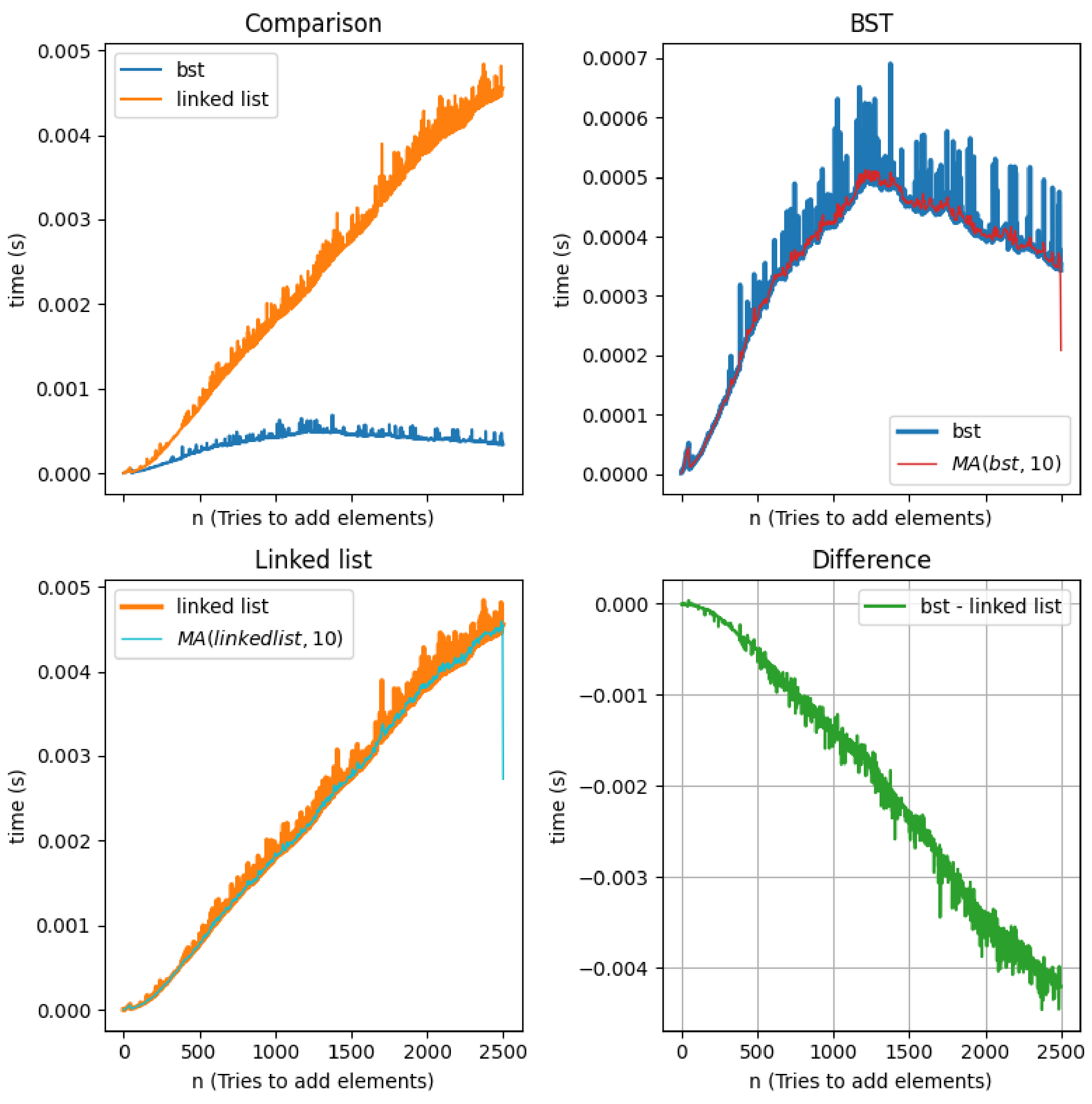

Figure 5: *Comparison of difference between sets.*

Figure 5 shows that the binary search tree consistently outperforms the linked list in set difference operations. This is likely due to the binary search tree's faster lookup times, which are required to check whether or not a value is contained in a set.

What's interesting about the figure is the dip in time taken by the binary search tree around the midpoint of the horizontal axis. Although the data is noisy, the moving average filter reveals a clear decreasing trend at this point. One possible explanation for this phenomenon is the way values are generated when benchmarking.

Values are generated as integers i such that $i \in \mathbb{N}$ and $i < n = 2500$. Both sets being compared receive values from the same interval, which means that at some point, we expect the proportion of values that are contained in both sets to be larger than the proportion of values contained in only one of the sets. This implies that the sets have become saturated with common values.

Now, recall that the set difference operation involves copying all unique values from one set to the other. This means that the algorithm only needs to add a new element to the new set if it is not already contained in the other set. Therefore, we only execute the line of code to add an element if the line of code that checks if the other set contains the value returns false.

If the value already exists in both sets, we save some time because we do not need to add it to the new set. This results in a dip in the curve for the binary search tree since it requires fewer operations in this case. However, we do not see this dip for the linked list because adding an element to the list is always a single operation, whereas checking if a value is contained requires n comparisons.

In conclusion, the dip in time taken by the binary search tree around the midpoint in the set difference operation is likely due to the sets becoming saturated with common values, which reduces the number of elements that need to be added to the new set.

4.2.2 Intersection between sets

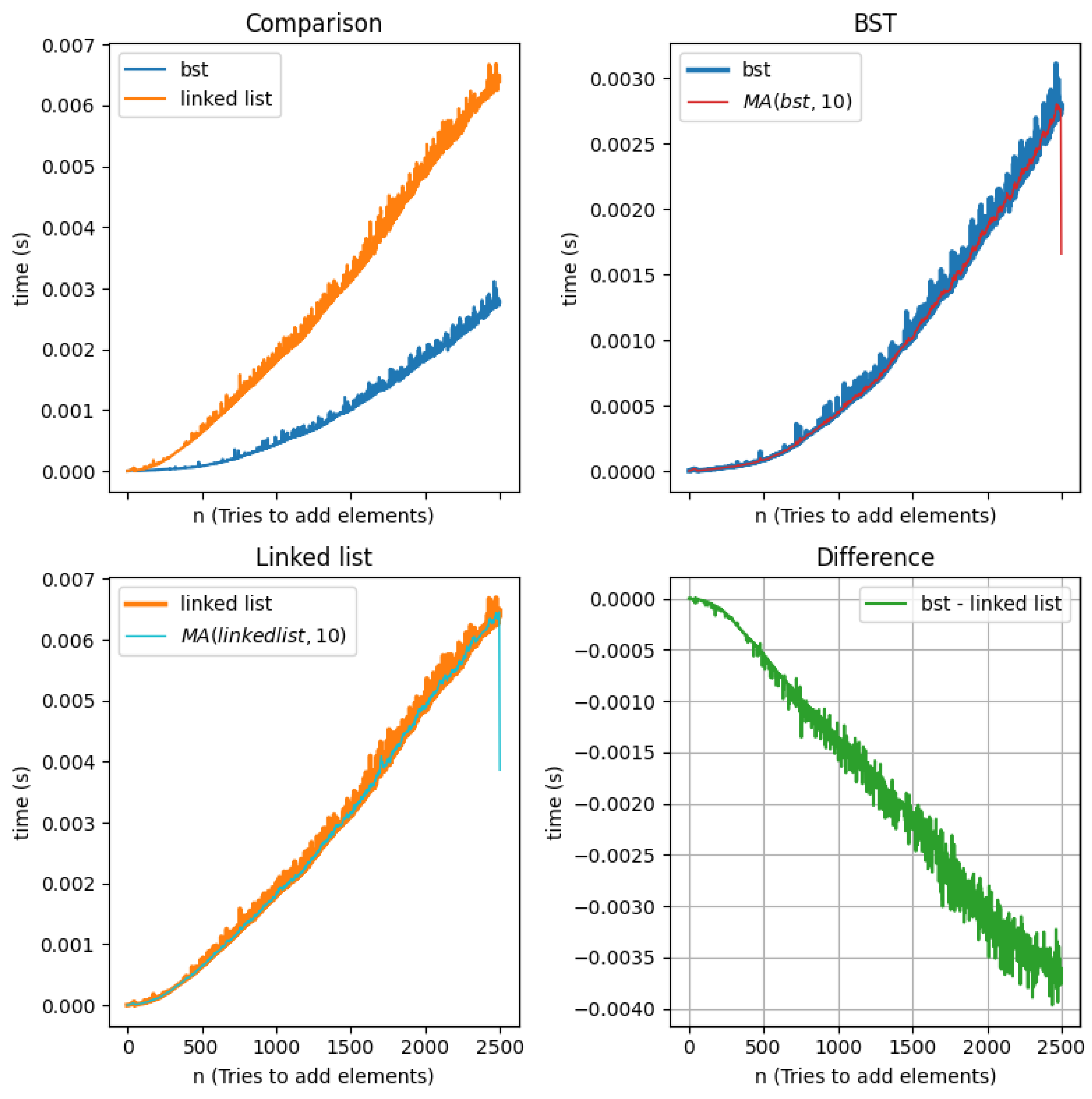

Figure 6: *Comparison of intersection between sets.*

The intersection comparison shown in figure 6 reveals that the linked list once again performs slower than the binary search tree and follows a similar curve to figures 4 and 5. Therefore, the linked list does not require further explanation in this section.

However, the binary search tree shows an increase in the time taken as the number of elements grows. This is due to the sets becoming saturated, as explained in the previous section. In this case, the consequence is the opposite of the dip observed in the difference comparison. As the number of values in both sets increases, the algorithm has to execute the add method more times, which explains why the curve follows a similar shape to a polynomial of degree 2.

5 Conclusion

Overall, the results presented in this assignment confirm the expected performance differences between the two data structures. The binary search tree consistently outperforms the linked list due to the faster lookup times for elements. The implementation of the binary search tree also benefits from its ability to maintain the data in a sorted order, which allows for efficient intersection and difference operations.

However, it has also been shown that the performance of the algorithms can be impacted by the nature of the data being used. In particular, the saturation of the sets can lead to some unexpected results, such as the dip in time for the binary search tree in the difference benchmark.

In terms of optimizations, there are some potential improvements that could be made to the current implementations. For example, the linked list implementation could be improved by using a hash table to speed up element lookup times. Similarly, the binary search tree implementation could be made more efficient by implementing rebalancing algorithms to prevent the tree from becoming unbalanced and slowing down operations.

Overall, the findings from this assignment demonstrate the importance of choosing the appropriate data structure for a given problem and the impact that data distribution can have on performance.

References

Free Software Foundation. (2023a). *GNU Bash.* `https://www.gnu.org/software/bash/`.

Free Software Foundation. (2023b). *GNU Make.* `https://www.gnu.org/software/make/`.

Wikipedia. (2023a). *Abstract data type.* `https://en.wikipedia.org/wiki/Abstract_data_type`.

Wikipedia. (2023b). *AVL tree.* `https://en.wikipedia.org/wiki/AVL_tree`.

Wikipedia. (2023c). *Binary search tree.* `https://en.wikipedia.org/wiki/Binary_search_tree`.