INF-1101 - fei003 - a1

This report details the implementation of a a set, more specifically an *ordered set*, which was used as the underlying data structure for a "spam filter" program. The results of the implementations are also tested and benchmarked.

1. Implementation requirements

Some implementation details were given for the ordered set. The supported operations are:

1. Adding an element to the set.
2. Getting the current size of the set.
3. Checking wheter a specific element is contained in the set.
4. Getting the union of the set and another set.
5. Getting the intersection of the set and another set.
6. Getting the relative component set of the set.
7. Iterating over the elements of the set, in sorted order.
8. No upper bound on the number of elements that may be inserted into the set.

1.1 Applications

The intention of the set ADT is to use it for an easy but naive implementation of a spamfilter to classify e-mails as spam or non-spam. The algorithm used by the spamfilter is simple, it starts out with a set of e-mails known to be spam/not to be spam.

An e-mail M is classified as spam if, and only if

$M \cap ((S1 \cap S2 \cap ... \cap Sn) - (N1 \cup N2 \cup ... \cup Sm)) \neq \varnothing$

where Sn is the spam words, Nm is the non-spam words. This states that if an e-mail is spam if it contains a word that occurs in *all* of the spam e-mails and *none* of the non-spam e-mails.

2. Theoretical background

[1] defines an abstract data type (ADT) as a data type whos operations are only accessible through an interface, and has its implementation hidden from the client (a program that uses an ADT).

What this inherently means, is that the same set of operations can have several different implementations, and that the client can safely switch betweeen implementations without breaking existing code (as long as the contract of the interface is held).

In order to evaluate the effectivenes, or rather, the complexity of an algorithm, this report will use a mathematical notation called *Big O*. Big O is used (in computer science) to classify an algorithms running time, or space requirements, as the input size increases. The mathematical definition will not be included in this report.

For implementing an ordered set as an ADT, an interface that described the operations with its return types was predefined in the precode (see source).

3. Implementation

When deciding on which way to implement the sorted set, two alternatives were considered. One that would have low development cost but high performance cost and vice versa. These are trade offs that often has to be considered in real life situations when developing software (i.e., development cost vs optimal performance), so this was a good exercise.

The sorted set implementation was done using the linked list implementation given in the precode. This is by far the slowest (in terms of run time), naive and reckless implementation (compared to other implementations). Very little effort went into optimizing the code. As long as the tests passed, it was fine. The positive part of this implementation was its low development cost (time), which was the prioritized factor for this assignment.

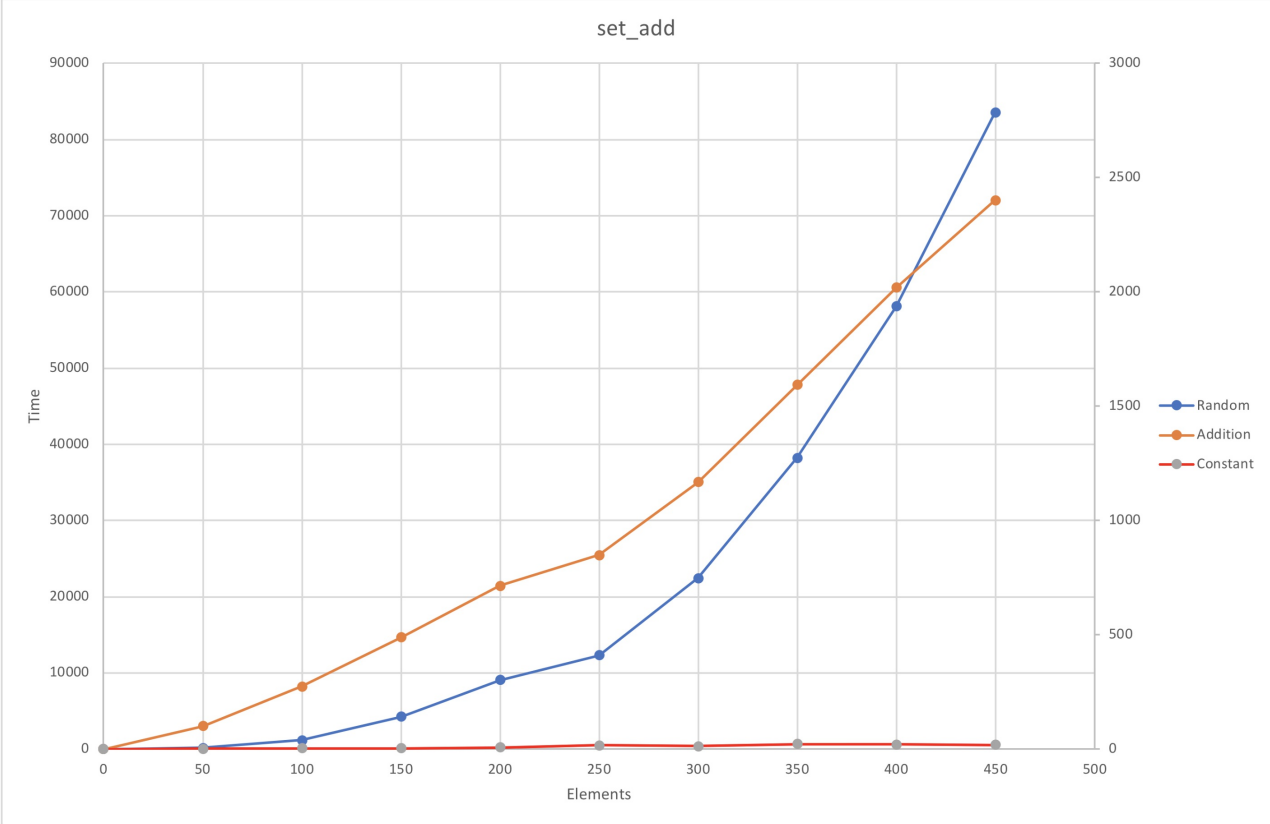
The caveat of using a linked list implementation for the set was definitely the sorting algorithm. This is due to the fact that the set had to be a *sorted* set, hence, the sort of the linked list had to be invoked on every add call.

4. Discussion

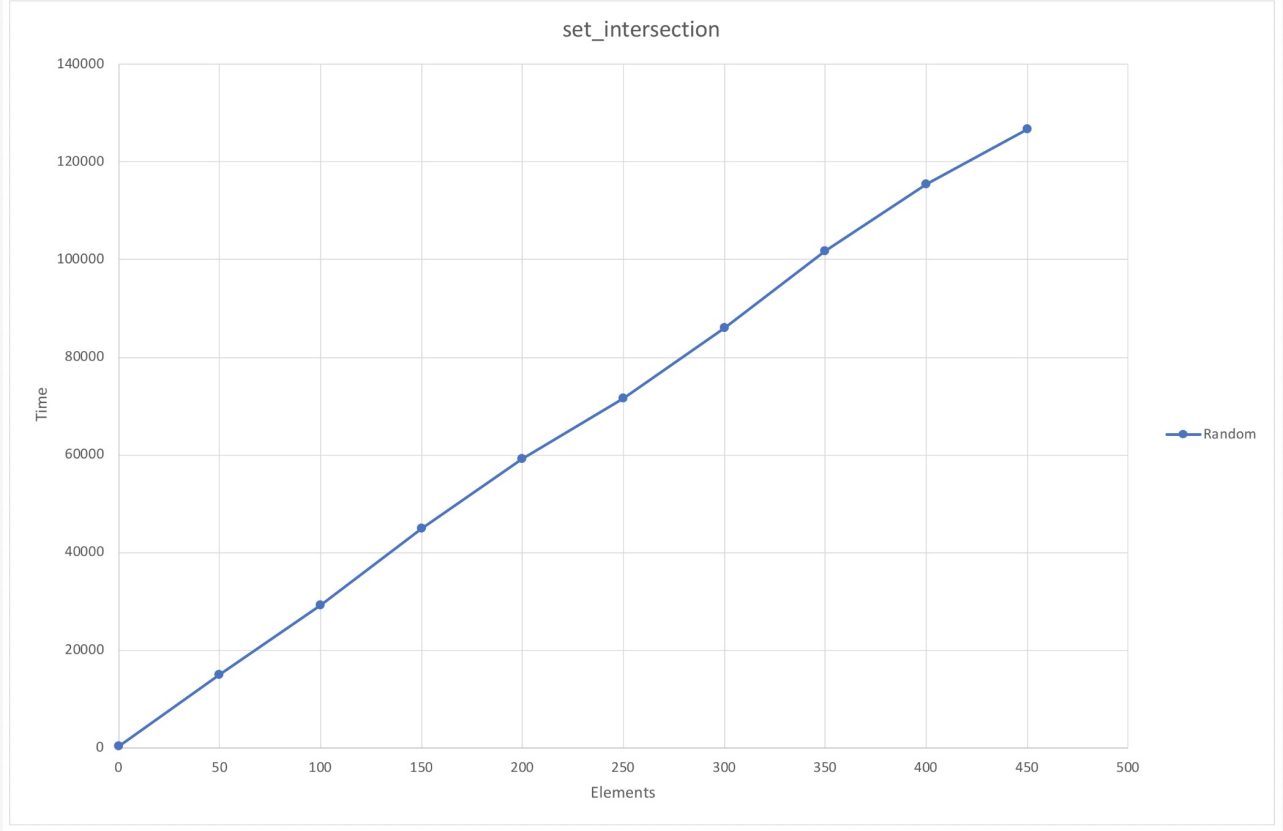
For benchmarking the `set_add` function, three sets of data with 450 elements were generated, one with random numbers, one with increasingly larger numbers and one with a constant value.

The figure below depicts three different types of complexity. The worst case was with random numbers (blue graph), which is O(n * lg n), second worst was the increasingly larger numbers (orange graph) which follows a more linear time complexity of O(n) and the third and the best case was when adding the constant value (red graph).

The best case scenario was due to the fact that the `set_add` function, which invoked a mergesort sorting algorithm, never had to sort anything. The worst case scenario was due to the `set_add` having to sort the set on every call.



The `set_intersection`, which has an identical time complexity as `set_difference` and `set_union`, had a time complexity of O(n). This is due to linked list `list_contains` having a time complexity of O(n).



5. Conclusion

Apart from a few, minor caveats, the code is working as intended. There are obvious improvements to be done in the implementation of the set, but poor design choices were taken due to some time constraints.

References

1 Robert Sedgewick, 1997. Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching: Fundamentals, Data Structures, Sorting, Searching. 3 Edition.