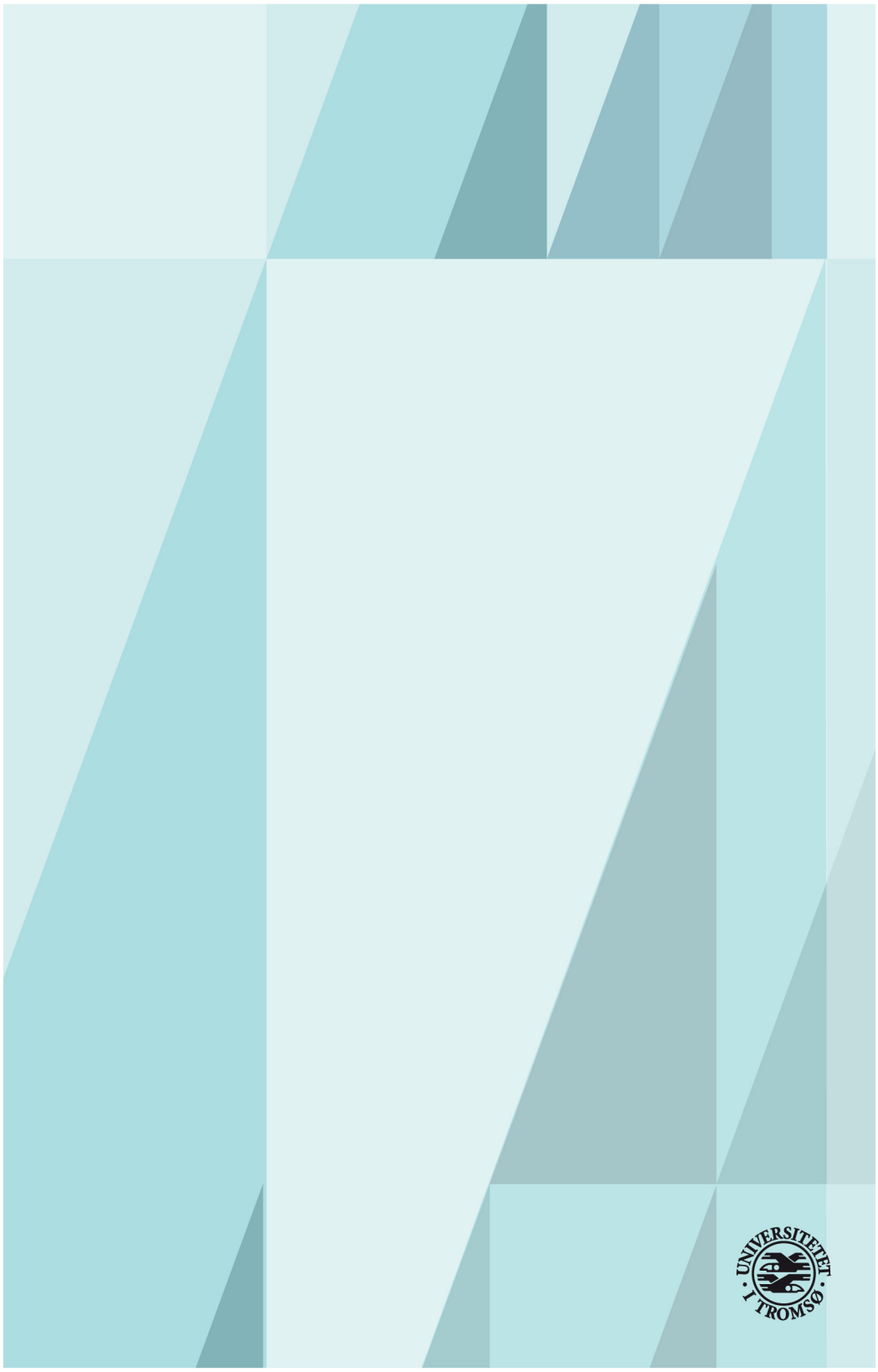
UiT

NORGES
ARKTISKE
UNIVERSITET

Ordered Set

2018

Fakultet for naturvitenskap og teknologi
INF 1101, HØST 2018
mbe101@post.uit.no



Innholdsfortegnelse

1 Introduction

Assignment number one is about the implementation of a Set *abstract data type* (ADT). Specifically, it's about implementing an ordered set. How the set is implemented is up to the student, but there should be no bounds to the size of the set, short of running out of memory.

1.1 Required set operations

- Adding an element to the set.
- Getting the current size of the set.
- Checking whether a specific element is contained in the set.
- Getting the union of the set and another set.
- Getting the intersection of the set and another set.
- Getting the set difference between the set and another set. (Also known as the relative complement set).
- Iterating over the elements of the set, in sorted order.

When the set is complete it should be used to implement a spamfilter. Attached to the assignment are thirteen emails. five mails, four sample emails and four spam emails where spam = S, sample emails = N and mails = M. The spamfilter implementation should determin if an email is spam based on the formula below.

$$\mathbf{M} \cap \mathbf{((S1} \cap \mathbf{S2} \cap \mathbf{...} \cap \mathbf{Sn)} - \mathbf{(N1} \cup \mathbf{N2} \cup \mathbf{...} \cup \mathbf{Nm))} \neq \varnothing$$

The output of the spamfilter should be formatted like this:

mail/mail1.txt: 0 spam word(s) -> Not spam

mail/mail2.txt: 0 spam word(s) -> Not spam

mail/mail3.txt: 1 spam word(s) -> SPAM

mail/mail4.txt: 2 spam word(s) -> SPAM

mail/mail5.txt: 0 spam word(s) -> Not spam

2 Theory

In this chapter we will discuss different types of implementations and sorting algorithms.

2.1 Implementation

There are many ways to implement the ADT Set and they may be semantically equivalent from another implementation, but vary in effectiveness. Two of these ways are List and Binary Tree. What they all have in common is that they act as a container which stores data in an organised manner.

2.1.1 List

Lists can be implemented in different ways, but the most common implementations are singly *linked list* (SLL) and *doubly linked lists* (DLL). In DLL each node keeps track of the previous and next node which lets you traverse the List in both directions (Illustration 1). Table 1 shows the time complexity of accessing, searching, inserting and deleting from a list.

	Best case				**Worst case**			
Linked list	**Access**	**Search**	**Insertion**	**Deletion**	**Access**	**Search**	**Insertion**	**Deletion**
Singly	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Doubly	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)

Table 1: Time complexity of List ref: http://bigocheatsheet.com/

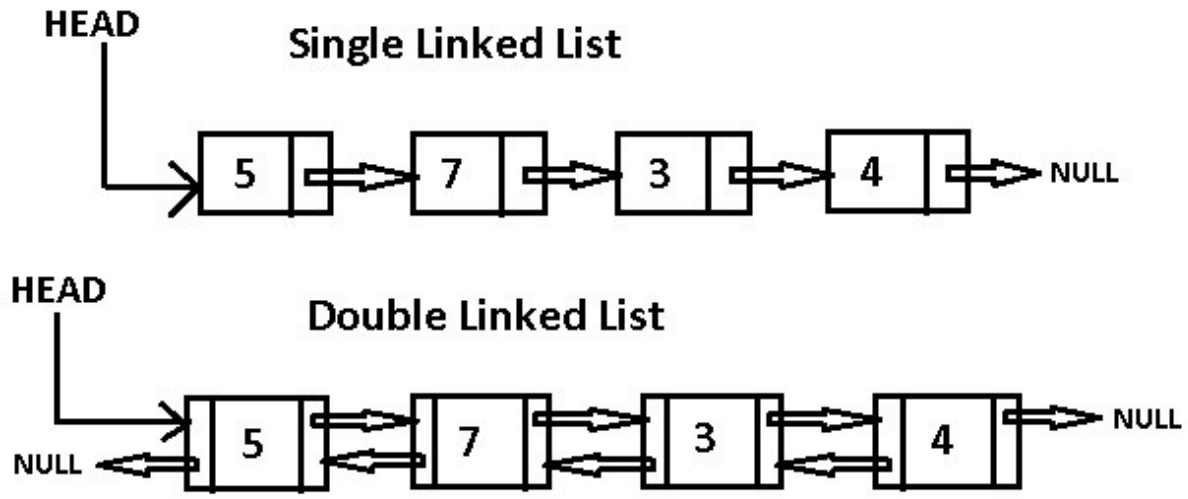


Illustration 1: Single linked list / Dobly linked list

2.1.2 Binary search tree

Like linked list, a *binary search tree (BST)* stores data in nodes. The first node in a binary tree is called the root node and each node has at most a reference to two children (Illustration 2). Typically, the value of the left child is always smaller than the parent node and vice versa for the right child. Table 1 shows the time complexity of accessing, searching, inserting and deleting from a binary tree.

	Best case				**Worst case**			
	Access	**Search**	**Insertion**	**Deletion**	**Access**	**Search**	**Insertion**	**Deletion**
BST	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)

Table 2: Time complexity of Binary Tree ref: http://bigocheatsheet.com/

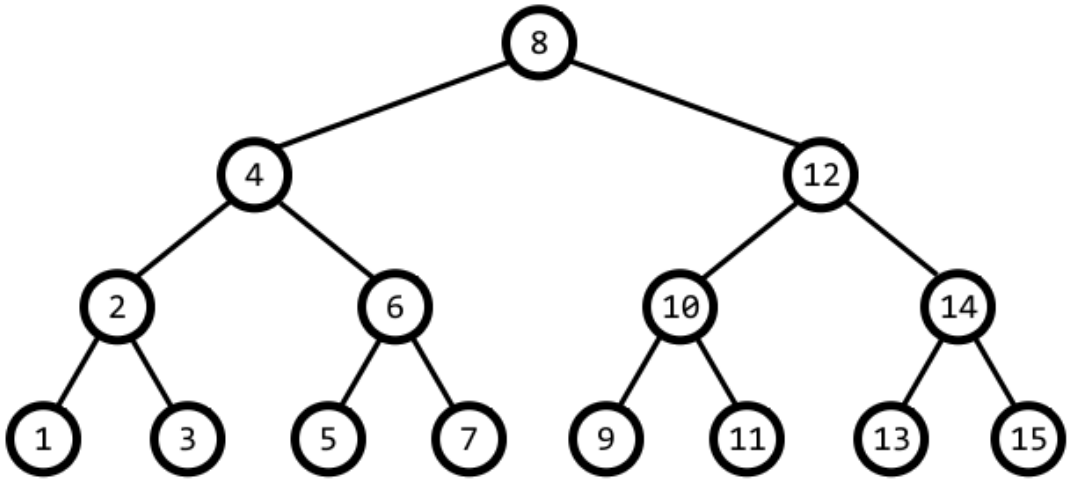


Illustration 2: Binary search tree

2.2 Sorting

A sorting algorithm, is an algorithm that takes data and sort elements in a certain order. The most commonly used orders are ascending or descending order. There are many different implementations of sorting algorithms. Figure 3 illustrate the time complexity of different sorting algorithms.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	0(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	0(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	0(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	0(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	0(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	0(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	0(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	0(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	0(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	0(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	0(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	0(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	0(n log(n))	O(n log(n))	O(n)

Table 3: Time complexity sorting algorithms ref: http://bigocheatsheet.com/

3 Results

I implemented the ADT Set as a DLL. The head of the DLL has a reference to both the head and tail of the list which means that the time complexity for adding both to the head and tail of the list is o(n). Further the node contains a reference to the next and the previous pointer which lets you traverse the list in two directions. The sorting algorithm used in the list_add function and by extension the set_add function is called Mergesort and by looking at the worst case scenario of this sort function (Table 3) we can quickly estimate that the time complexity is O(n log(n)). Considering this I expect that the results from the set_add function should follow a graph similar to (n log n) which means that the size of the set is in practice limited by the time complexity of this code aswell as the memory. Illustration 3 show the time complexity of the function set_intersection which operates in the same manner as set_union and set_difference. As you can see the time complexity of this cod eis O(n).

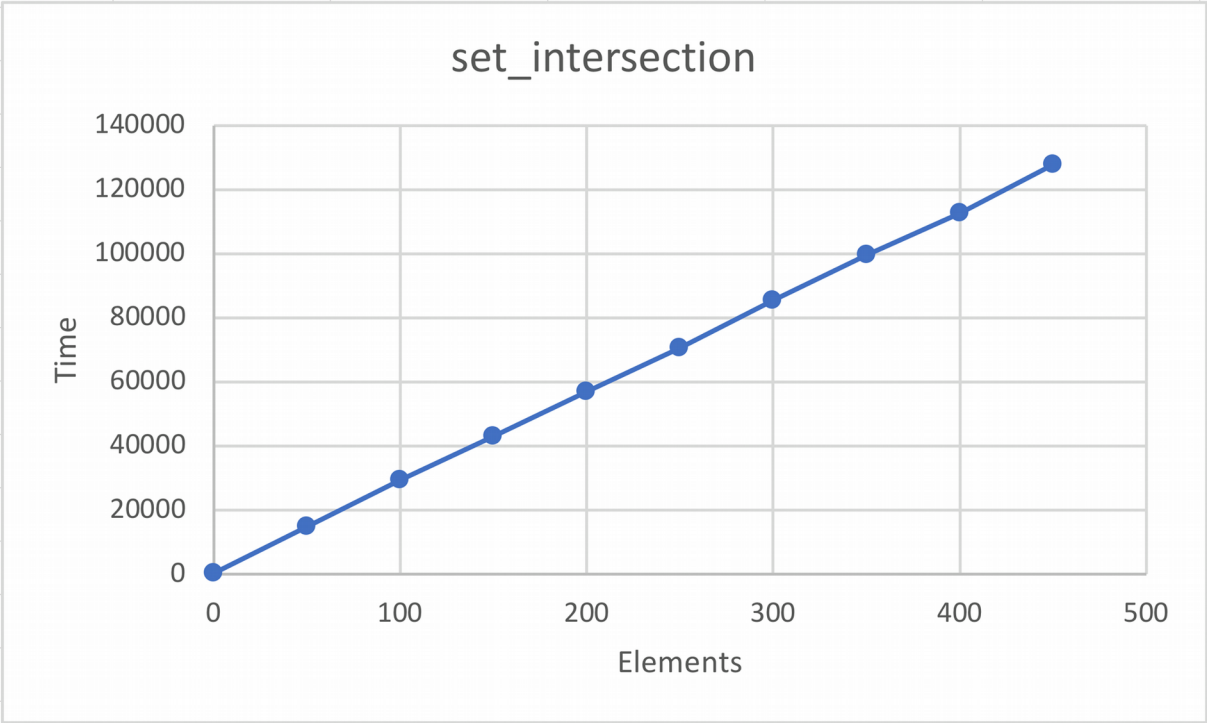


Illustration 3: Time complexity of set_union, set_intersect and set_difference

Illustration 4 shows the time complexity of set_add using three different lists of numbers. The reason for this is that the add function has a built in sorting function that operates differently based on the numbers in the set. The blue dataset is created using random numbers which, in our case was the worst case scenario and had a time complexity of O(n log(n)). The orange datasets add increasingly larger numbers which seems to follow a more linear time complexity O(n). In the end we tried to add a constant value to the set which has a time complexity close to O(1).

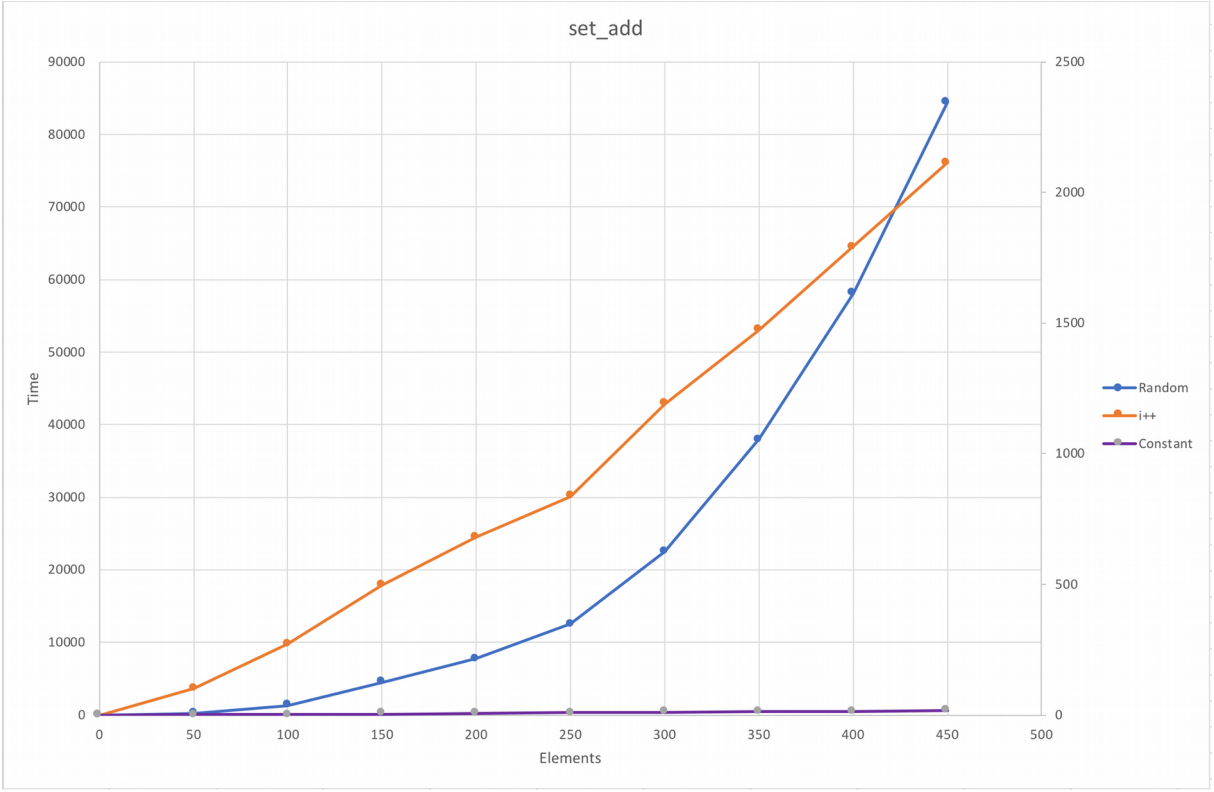


Illustration 4: Time complexity of set_add using random/increasingly larger and constant numbers

4 Conclusion

The implementation of assignment 1 works as expected. The time complexity of the code generated is O(n log(n)) where the weakest link is the sorting algorithm implemented in the list. The time complexity of this code can be changed using a better sorting algorithm, but the iterative length of the data sets can be reduced using a more complex container for the data. A container such as a list iterative length = N, but a tree could reduce this to $\log_2(n+1)/\log(2) - 1$. This could further be reduced by hashing, but i will not go into depth about this.