INF-1100: Data structures and algorithms
Mandatory Assignment: ADT

Eskil Bjørnbakk Heines

February 27th, 2023

1 Introduction

In this assignment I was tasked to make two separate implementations of an ADT (Abstract Data Structure) and implement an algorithm for filtering spam-mails. In the pre-code given with the assignment, there was already a linked list implementation which could be used in the implementation of the new ADT's. Additionally, I was tasked to do some experiments with the ADTs to see which one was the fastest doing operations on various sizes of data. The assignment can be broken down to these tasks:

- Implement set of doubly linked lists
- Implement set of arrays
- Implement algorithm for spam filtering
- Testing ADT implementation on different data sizes
- Evaluate the test

2 Technical Background

The pre-code is written in C, which is the language used for this assignment. It comes with a Makefile [1] which makes the compiling process of several programs easier. The pre-code comes with a lot of files with different purposes, some to be implemented and some only for support or testing of the implementation. The set and spam filter files are the ones which will be implemented. There is also an already implemented linked list which is open to be used in the implementation of the ADTs.

For checking the time complexity of the different ADTs, the GProf-program [2] for the terminal will be used. This program has the purpose of checking which operations are called upon the most in the program.

For every ADT it's usually a sorting algorithm going along with it, and which sorting algorithm that is chosen is up to the amount of data that's going to be sorted. The efficacy of sorting algorithms depends on two main parameters; time complexity and space complexity [3]. Space complexity is about the memory necessary to execute the program. Time complexity is not about the total time taken to do the sorting, but rather the number of times the instruction is executed. Both time and space complexity are calculated as the function of the input size (for the sheet n, meaning number of elements). Time complexity can be sorted into three different types;

- **Best time complexity**: Defines the input for which algorithm that takes the least amount of time
- **Average time complexity**: Takes all random inputs and calculates the computation time for all inputs and divides it by the total number of inputs
- **Worst time complexity**: Defines the input for which algorithm that takes the most amount of time

Under is a sheet viewing the different time and space complexity of different sorting methods that's relevant to this assignment's implementation:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Bubble sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Quick sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)

The complexity of the code is a combination of the number of operations run to complete the program, and the memory necessary to do this.

Here is also a sheet of the complexity of arrays and doubly linked list, the used ADTs in this assignment:

Data structure	Time complexity								Space complexity
	Average				Worst				Worst
	A	S	I	D	A	S	I	D	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Doubly Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)

A = Access, S = Search, I = Insertion, D = Deletion

Sheet from "Big-O Algorithm Complexity Cheat Sheet" [4]

3 Design and Implementation

In this assignment there are two main tasks; creating two different ADTs and implementing a working algorithm for filtering the spam mails in the given files. The new ADTs will be made like an extension of another ADT. Set will be made like a linked list with the functions of a set, and array set will be made like an array with the functions of a set. This makes the code easier to understand and implement, while also making it less complicated.

3.1 Set

The first implementation of the assignment is the set of doubly linked lists. The implementation represents doubly linked list as sets, sets being several elements in ordered setting with all elements being unique. This implementation uses a lot of the functions from the linked list that came with the pre-code while still being another form of an ADT. The set has a list in the structure that points to the list implementation, and just as its necessary to allocate memory for the set when created, the list that's linked to the set also has to have memory allocated. The functions for union, intersection and difference have all been implemented in an improved way, meaning that the implementation has been improved several times. As said in the acknowledgements, working with Kristian has given us both the opportunity to find the best implementation of the functions.

3.2 Array set

The second implementation of the assignment is the set of arrays. Just as the set is a doubly linked list represented as a set, the array set is an array represented as an array. As shared with the set, several of the functions are called upon from the set implementation, making it not only easier to understand but also easier to implement. The structure of the set has a double void pointer to an array, the first pointer pointing to the array and the second pointer pointing to the element in the array. This is important for the implementation to work. The logic behind the union, intersection and difference has a lot in common with the implementation of these function in set, only difference is the iteration through the ADT.

3.3 Spam Filter

In spam filter there should be implemented an algorithm for checking if a mail is spam or not. In the pre-code there are 3 files, named “mail”, “non spam” and “spam”, which should be used to check if the mails contain spam- or non-spam-words. If the mail contains a word that is in every one of the spam-mails, it is considered spam. If not, it’s obviously not a spam mail.

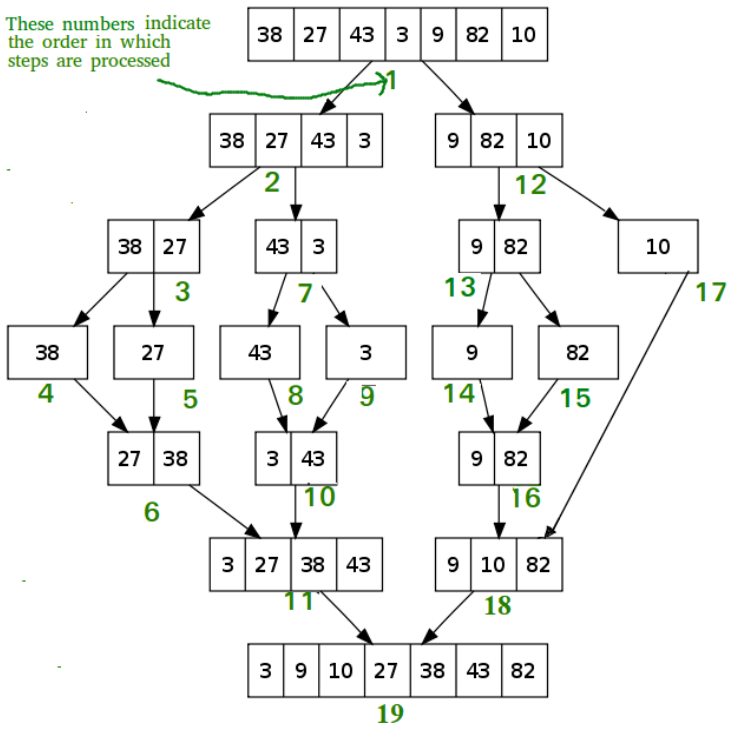
By tokenizing all the mails, both spam and non-spam, there will be made sets containing all the words in the mails. By using the union (given two sets a and b, union gives all the elements in either a or b), a non-spam list of absolutely all the words in all the non-spam mails can be made. Then by using intersection (given two sets a and b, intersection gives all the elements in both a and b), a spam list of all the words in all the spam-mails can be made. Be checking the difference (given two sets, difference gives the elements that’s in a and not in b) between these new lists, the words that’s considered spam-words can be added to another new list which will be checked with the mails for considering if their spam or not.

The implementation of spam filter is inspired by a code found on the internet [5] with some of the code being taken from there.

3.4 Sorting

Choosing the right sorting algorithm for the ADT for best complexity and best performance isn't always easy. For this assignment it’s not that difficult though. We know that the easiest to implement is the bubble sorting algorithm, but this is also not very efficient in sorting larger amounts of data, something that will show in the testing of the program with a bubble sorting algorithm. As in the doubly linked list, the sorting algorithm that’s been used in the implementation is merge sort, and here is why.

The merge sorting algorithm has the arguments of the start, end and middle of the given ADT. The start is always 0, the end is the size of the ADT, and middle is the size divided by two. Merge sorting algorithm will use this information to break down the given ADT to smaller and smaller parts until all parts contain only one element as shown below:



When the parts are only one element, they will piece by piece be merged again in the ordered order. As seen in the sheet of time complexity, merge sort has a Ω(n log(n)) for best, average and worst time complexity. This means that it will perform very well until the data gets to very big sizes, something that's not the case for this program.

This is the sorting algorithm chosen to be implemented in the array set ADT. It's also the same that's already implemented in the linked list, which is the one used for sorting the implemented set ADT. For array set, there is a bubble sort algorithm implemented for showing the difference between merge and bubble sort in larger data sizes.

4 Evaluation

With the different ADTs and the different sorting algorithms, the program will have different efficiency depending on which ADT and which sorting algorithm is used. The sheet under shows the total time taken to execute the program for union, intersection, difference and to add a given number of elements to the chosen ADT.

Here the total time taken to execute an operation is presented in the sheet. Showing both the set and array set ADT, with the merge and bubble sort algorithm. This is mainly to show the immense difference between them when working with larger data sizes. The bubble sort for 100.000 numbers is not in the sheet because it's still running.

Amount of elements	ADT (sorting method)	Union	Intersection	Difference	Add to set
100	Set (merge)	0,00014 sec	0,00005 sec	0,00009 sec	0,0001 sec
	Array set (merge)	0,0015 sec	0,000052 sec	0,00042 sec	0,0016 sec
	Array set (bubble)	0,0010 sec	0,000015 sec	0,00017 sec	0,0027 sec
1.000	Set (merge)	0,035 sec	0,0021 sec	0,0023 sec	0,044 sec
	Array set (merge)	0,0015 sec	0,000052 sec	0,00042 sec	0,058 sec
	Array set (bubble)	0,53 sec	0,00092 sec	0,075 sec	0,57 sec
10.000	Set (merge)	3,23 sec	0,13 sec	0,88 sec	3,35 sec
	Array set (merge)	4,08 sec	0,10 sec	1,10 sec	3,98 sec
	Array set (bubble)	552,64 sec	0,098 sec	69,56 sec	879,29 sec
100.000	Set (merge)	409,84 sec	24,58 sec	124,70 sec	417,66 sec
	Array set (merge)	500,87 sec	10,52 sec	132,46 sec	483,27 sec
	Array set (bubble)				

The sheet shows the total time taken, meaning it's not about the time complexity but more about how the program performs. Obviously the larger the data size, the longer the time it takes to execute. In the testing file the numbers that's used to check the different functions are even numbers and odd numbers, which is the reason intersection always takes only a few seconds since there are no numbers in intersection between even and odd numbers. This also shows us the side of the functions if there is nothing to be added to the new set, giving another dimension to the testing. There is no secret that the set with the merge sorting algorithm is the best performing algorithm, but array set implementation is not far behind.

From using the GProf-program the time complexity can be seen in a different light than only from the sheets. In both set and array set, the functions for checking if the given element is in the set, are the most

used function. This is logical because as said earlier, the set ADT only has unique elements which then requires the program to often check if an element is already in the set or not.

Acknowledgement (optional)

For this assignment I have discussed and worked a lot with Kristian Harvey Olsen (1st year student) and Gard Schive (1st year student) with Gard helping me with the spam filter implementation. Also, some of the code for spam filter is found online helping to understand and implement the algorithm. The code found online was not working 100% with my code, so it needed work to be working with the code.

References

[1] Makefile, GNU Make:

https://www.gnu.org/software/make/manual/make.html

[2] GProf, GNU gprof:

https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

[3] "Time Complexities of all Sorting Algorithms", GeeksforGeeks.com:

https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

[4] "Big-O Algorithm Complexity Cheat Sheet", Souravsengupta.com:

https://souravsengupta.com/cds2016/lectures/Complexity_Cheatsheet.pdf

[5] "spamfilter.c", GitHub.com:

https://github.com/HUEHUEATHAXME/src_spamfilter/blob/master/spamfilter.c