Exercise 1.1* [50 points]

This exercise is designed to help familiarize you with choosing the right data structure for the right problem.

For the following problems, you may assume implementations of the interface: Stack, Queue, Deque, USet, and SSet. That is, you can use the name of the Interface to refer to the data structure implementing it.

Each problem below involves reading a text file line by line. You have to propose an appropriate interface–Stack, Queue, Deque, USet, or SSet–which can be used to perform the specified operations on each line. Your proposed solution should be fast enough that even files containing a million lines can be processed efficiently.

For each solution, mention how your proposed data structure can be used to achieve the desired result.

* - adapted from Exercise 1.1 from the book.

(a) [5 points] Read the input one line at a time and then write the lines out in reverse order, so that the last input line is printed first, then the second last input line, and so on.

> **Solution:** The Interface that will be used in the proceeding scenario would be **Stack**. Stack works on the last in first out basis. Which means that the last line will be popped and printed out first, second last will be popped and printed after that and so on.

(b) [5 points] Read the first 50 lines of input and then write them out in reverse order. Read the next 50 lines and then write them out in reverse order. Do this until there are no more lines left to read, at which point any remaining lines should be output in reverse order.

In other words, your output will start with the 50th line, then the 49th, then the 48th, and so on down to the first line. This will be followed by the 100th line, followed by the 99th, and so on down to the 51st line. And so on.

You should never have to store more than 50 lines at any given time.

> **Solution:** The interface used for this part would be **Stack**. There would be one if condition catering along with the interface that would count the number of lines and when it reaches 50, it will start to pop and print the lines until it is emptied again.

(c) [5 points] Read the input one line at a time. At any point after reading the first 42 lines, if some line is blank (i.e., a string of length 0), then output the line that occurred 42 lines prior to that one. For example, if Line 242 is blank, then your program should output line 200. This program should be implemented so that it never stores more than 43 lines of the input at any given time.

> **Solution:** The interface that can used for this part would be **Queue**. The lines are to be counted and displayed in the same order as they have been read. There will be an if condition that would check if the length of the queue reaches 42 and still no blank line is read, it will start to dequeue the first element every time a new element is been enqueued.

(d) [5 points] Read the input one line at a time and write each line to the output if it is not a duplicate of some previous input line. Take special care so that a file with a lot of duplicate lines does not use more memory than what is required for the number of unique lines.

> **Solution:** The interface to be used in this scenario would be **Unordered Set**. The unordered set would add the line as its element along with its key but if the key repeats (which means that the line is repeated) it would not add the line to the set.

(e) [5 points] Read the input one line at a time and write each line to the output only if you have already read this line before. (The end result is that you remove the first occurrence of each line.) Take special care so that a file with a lot of duplicate lines does not use more memory than what is required for the number of unique lines.

> **Solution:** To implement this scenario we need to work with two interface, first being the **Unordered Set** and second being the **Queue**. How it would go along is that each line would be read and added to the U-set, but once the repeated line is read, U-set would not entertain it and it would be added to the Queue. The solution to your scenario would be the elements in the Queue rather than in the U-set.

(f) [10 points] Read the entire input one line at a time. Then output all lines sorted by length, with the shortest lines first. In the case where two lines have the same length, resolve their order using the usual sorted order. Duplicate lines should be printed only once.

> **Solution:** In order to deal with this scenario, **S-set** interface would be used. After reading each line, the line would be added as an element along with its length and a key value. As it is an S-set, the data inside the set would be sorted according to one its value (length). And as it is an S-set, the key values would be unique and no repeated line would be entertained.

(g) [5 points] Do the same as the previous question except that duplicate lines should be printed the same number of times that they appear in the input.

> **Solution:** The interface to be used in this scenario is **S-set**, and the same method as above. The only change that would be made is that another key would be added to it and it would operate on two keys. One assigned by the line and another by the time of its occurrence in the program. While it would be sorted according to the length.

(h) [5 points] Read the entire input one line at a time and then output the even numbered lines (starting with the first line, line 0) followed by the odd-numbered lines.

> **Solution:** This scenario would be dealt best with a **Dequeue** working in parallel. The lines are read one by one and enqueued once from the start and once from the end. which means that first will pushed from the front while second will enqueued from the end. In the end, even numbered lines will be in the start of the Dequeue and odd numbered lines will be in the end.

(i) [5 points] Read the entire input one line at a time and randomly permute the lines before outputting them. To be clear: You should not modify the contents of any line. Instead, the same collection of lines should be printed, but in a random order.

> **Solution:** In this scenario, **Unordered-set** can be the best possible interface. We may have to assume that we already know the keys for each line. We can randomly generate the keys and store them to avoid redundancy. And with each key being selected, a find function will ring a line that would be printed.

Exercise 1.2 [5 points]

A *Dyck word* is a sequence of +1's and -1's with the property that the sum of any prefix of the sequence is never negative. For example, +1,1,+1,1 is a Dyck word, but +1,1,1,+1 is not a Dyck word since the prefix $+1 \;\; 1 \;\; 1 < 0$. Describe any relationship between Dyck words and Stack push(x) and pop() operations.

Solution: Dyck Words are basically identitfied using the Stack operations pop and push and without these operation it be difficult to differentiate between the Dyck and Non-Dyck words. The functionality that is being implemented it that each element of the word is popped and pushed into a stack one by one. And after every addition the sum of the elements is calculated to find if the sum is non-negative after every step or not. If at any instance the sum appears to be negative then it can be concluded that the given word is not a Dyck Word.

Exercise 1.4 [10 points]

Suppose you have a Stack, s, that supports only the push(x) and pop() operations. Show how, using only a FIFO Queue, q, you can reverse the order of all elements in s.

Solution: Using a queue we can easily reverse the order of the elements in stack s through simple pop push operations. Each element starting from the last one in the stack is popped and pushed into the queue one by one. Once the stack is emptied, we can notice that the reverse order is obtained in the queue. Now using the dequeue functionality we can remove the elements one by one starting from the first one from the queue and push it back to stack to obtain a reverse order of s in-place of the previous order