Introduction

This project solves the problem of searching for anagrams, on a scale of files up to 50 million words. With the possibility of scaling.

The functionality includes the ability to read data from an input file, process it, find anagrams, and output it to the console. An anagram counting system is also implemented. The project includes a logging system for the main function's memory load.

Architectural solutions

This application processes the file line by line, through ***BufferedReader***, this architectural decision was made to unload memory and write a limited number of lines to the buffer.

A data structure such as ***Map*** with the implementation of ***LinkedHashMap*** was used. Words are grouped into a ***Map<String,List<String>>***, where the key is a word its characters sorted alphabetically (cat -> act).

All words with the same **key** are considered anagrams and are stored in the same list under that **key**.

Thank to this approach, all words that are anagrams of each other naturally map to the same **key** and are accumulated into a single list.

LinkedHashMap is used to preserve the insertion order of keys, ensuring stable output ordering.

To improve the quality of anagram search and to relieve memory, word filtering was implemented, which **normalized** words for further processing.

The output of anagrams was implemented through the String class method - **join,** for output groups of anagram in one line.

This application has a functionality for logging the state of memory, for a more correct presentation of the capabilities of this program. Logging was implemented through the object of the class ***Runtime.getRuntime*** - for working with the JVM

This was implemented in order to evaluate the feasibility of the program with big data.

Tools and technologies

Maven - build and dependency management
Docker application isolation and launch on any environment
CLI support - for more convenient switching between tested files (**file**, **resource**)

Scalability and limitations

Processing 10 million words, verified on the dictionary, the user device copes perfectly. **(~1.5 GB)**

This application is not suitable for processing 100 billion words, but it can be scaled up to BigData. It is possible to use **Apache Spark** to comfortably process **100 billion words.**

Apache Spark was not used for several reasons:

- Infrastructure setup is required.
- This solution works effectively within 10m words.
- Logic is easily transferred if necessary.

Conclusion

The task was successfully completed. This program is stable, tested, and easily scalable. Memory requirements were taken into account and verified.