# Initializations

* Initialize Flags **Interface** and **Testing** (Testing = “Y” indicates that you want to run test file, Testing = “N” runs the program on TREC Data and Interface=“Y” runs the interface to take input word to give its posting list). Variables **input file, stop file, output file, forward index file, inverted index file and main.qrels** are initialized, that are the files names (Eg: parser\_output.txt). The input files are different for TREC and Test file, these fields require the file directory path. For query processing, set the flag to “Y.”

# Main

* The main function manages the workflow of an information retrieval system. It begins by loading stopwords from the file supplied by stop\_file and initializing a TextParser\_Indexer instance with them. Then, based on the flags, it parses and indexes input files, saving the generated dictionaries. If query\_processing is not enabled, it parses the input file; otherwise, it parses multiple files and appends the suffixes "\_1" to "\_15" to the input\_file. In query processing mode, it creates paths for input/output files and, if relevancy is enabled, handles relevancy calculations, which include parsing topics from a specific file, calculating TF-IDF weights, comparing queries to documents, and computing relevancy metrics, which are then stored to a result file. Finally, it prints the complete program runtime, which includes tokenization and indexing. If the interface is enabled, it runs an interactive interface for querying.

# Text Parser and Indexer Class

This creates instances for Tokenizer, Word Dictionary, File Dictionary, and Indexer Class.

## Parse File

* This function uses regular expression to find and iterate over matches of the pattern "<DOC>...</DOC> to figure out the document in the file read. Then it loops iterates over each document in the file, gets the content between "<DOC>...</DOC> and calls function **parse\_document** to parse through each document.

## Parse Document

* Within the document content it finds the content between '<DOCNO>...</DOCNO>' and '<TEXT>...</TEXT>' and calls **tokenize** function to tokenizes the document content into lower case tokens and removes numeric, splits on non-alphanumeric and eliminates stop words. Then it uses **stem** function to stem the tokenized words to their root words
* Used **map\_word\_to\_id** and **map\_word\_id\_to\_doc** functions and saved both the file name with unique file ID and stemmed token/word with their unique ID in **file\_dict** and **word\_dict** dictionaries.
* Finally, we use the above dictionaries to create forward and inverted indexes using **index\_building** function and measure the time taken for indexing each document.

## Save Dictionary

* This function saves **file\_dict** and **word\_dict** dictionaries to output file, **forward index** dictionary to forward index file, **inverted index** dictionary to inverted index file, print indexing time, and print total size of indexes.

## Run Interface

* Loads the output file and inverted file and stores it in dictionaries. Takes user input and searches it in stop words file (if it is stop words throw an error), stems the word and gets its word ID from output file (if word does not exist throw error), searches stemmed word in inverted index file (if word ID does not exist throw error), and finally loads the posting list information from inverted index and prints it.

# Tokenizer Class

* The tokenize function first converts document to lowercase and uses regular expression to split it into stream of tokens. If tokens have numbers or stop words they are filtered away. It handles the non-alphanumeric split on punctuations too. This returns the final list of tokens after initial processing.

# Word Dictionary Class

* It initializes a mapping of document to word\_tokens, id-to-word\_token, and word\_token-to-id dictionaries for the class. Existing PorterStemmer function from nlkt to stem words to their roots.

## Stem

* This function Stems a given word using the Porter Stemmer.

## Map word to Word ID

* This function adds a new word/token to the dictionary and, if it doesn't already exist, gives it a unique numerical ID.

## Map Word ID to Document

* This function adds a word ID to the document's mapping, to keep the mapping information.

# File Dictionary Class

* It initializes a dictionary to map document IDs to internal document IDs, dictionary to map internal document IDs to the original document IDs and, a counter that keeps track of the current internal document ID.

## Map Document to Document ID

* This function adds a unique Document ID to each Document using a counter to update the IDs.

# Indexer Class

It initializes forward and inverted indexes.

## Index Building

* It uses stemmed tokens to get the frequency of the token in the document, It uses the word\_dict and file\_dict (file\_doc\_id) to get the unique IDs for the Words and Documents and construct inverted and forward index dictionaries.

# Query Processing

Initializes instances for the Tokenizer, Word Dictionary, File Dictionary, and Indexer classes.

## Calculate tf-idf weights.

* This function reads the forward index file, inverted index file, and output file to calculate TF-IDF weights for terms based on the provided parameters. It computes IDF for each term in the collection using the inverted index and then calculates TF-IDF weights for each term-document pair in the forward index. By leveraging these weights, the system generates TF-IDF representations for both queries and documents, facilitating efficient retrieval and comparison of relevant information.

## Parse topics file.

## This function parses the topics file, extracting document numbers, titles, descriptions, and narratives. It tokenizes, removes stop words, and stems the text, returning relevant information for each topic in a dictionary.

## Compare queries to documents.

## This function compares queries to documents using TF-IDF weights, calculating cosine similarity between query vectors and document vectors. It sorts documents based on cosine similarity scores for each query, writes the results to specified files, and returns a dictionary containing sorted similarities between queries and documents.

## Cosine similarity

## This function calculates cosine similarity between a query vector and document vectors by computing the dot product between them. It then calculates the magnitudes of both vectors and avoids division by zero, returning cosine similarity scores.

## Relevancy

## This function computes precision and recall for each topic and document by comparing the relevancy between MQRELs and computed relevancy. It reads the MQRELs file and relevancy file, then calculates precision and recall values, returning the results as a dictionary.

# Output

This program generates the “vsm\_output.txt” file and the “results.txt” file.

The output “vsm\_output.txt” is for the main query using just the title.

A screenshot of a computer

Description automatically generated

The results file has precision and recall for main query combinations.

A screenshot of a computer

Description automatically generated