Data Structures and Algorithms II (2024/25)
Lab: Building a search engine like Google

During this project you will **build a search engine like Google**. The user (red) must be able to write queries through the CLI. Then, the program must print the top 5 most relevant search results from the dataset that contain all the words from the query. For example:

```
******* recent searches ********
*  barcelona sagrada família   *
*  programming java            *
*  1992 olympic games          *
*******************************

Search: cute cats

(0) Cats are awesome!
---
Learn more about cats and how they live in the wild. They are really cute.
---
relevance score: 32


(1) Cat breeds
---
The following list of cat breeds includes only domestic
cat breeds and domestic and wild hybrids. It also includes the most cute ones.
The list includes established breeds recognized by v...
---
relevance score: 13


(2) Cats and Kittens for Adoption in Barcelona
---
If you are reading this, it's very likely that you are
looking to extend your family with a cute minion. We will help you make the
right decision when it comes to adopti...
---
relevance score: 8


[3 results]

-----------------------------
```

```
Select document: 2

ID
1453

TITLE
Cats and Kittens for Adoption in Barcelona

RELEVANCE SCORE
8

BODY
If you are reading this, it's very likely that
you are looking to extend your family.
We will help you make the right decision
when it comes to adopting a cat.

When you decide to share your life with a [cat](132),
you have to take into account many things,
one of them is that he will live in his
home and let you live with him.
He will decide when he will want to play,
eat, sleep... And he'll make you participate
in his decision, even when you're sleeping!

-----------------------------

******* recent searches ********
*  programming java            *
*  1992 olympic games          *
*  cute cats                   *
*******************************

Search: cute dogs

...
```

This lab is inspired by The anatomy of a large-scale hypertextual web search engine (Brin & Page, 1998). You are encouraged to read the paper to understand the more complex approach Google followed.

Plagiarism Disclaimer

- You **MUST NOT** look at anyone else's solutions, including previous year students.
- You **MUST** make your repository private.
- You **MAY** discuss the assignments with other students, but you may not look at or copy each others' code.
- You **MUST** commit frequently to the repository (at least 1 commit per exercise).
- You **MUST NOT** share your code with other students even if they ask you to.
- You **MUST** add a link next to any code you copy from external sources.
- You **MUST NOT** ask other students for their project or fragments of their code.

Definitions

- The **query** is a string consisting of keywords separated by spaces. E.g. \cute cats".
- The **document** is any searchable entity which has a textual title, textual content and can contain links (references) to other documents. For example, a website.
- The **search results** is an ordered collection of documents that contain the keywords of the query.
- The **search engine** is a program that searches through documents and returns the search results most relevant to the user query.

Task breakdown

***, must haves** (7/10 marks)
****, nice to haves** (3/10 marks)
*****, challenges** (+4 extra marks)

Before lab 1 you should be able to run C code, commit and push it to GitHub:

- Create a repository from the template using name \edaii-2025-pxoy-id" (where \pxoy" is your labs group, e.g. p102, and \id" is the id in the groups list) and type \Private"
- Share it with: @miquelvir, @hectorforido, @mcloses, @psantosUPF
- Make sure you can compile and run C code (\make r") and the tests (\make t")
- Finish reading this guide and start with lab 1 tasks

After lab 1 you should be able to load documents from the file system, print them to the CLI and allow the user to select and read one through the CLI:

- * Implement the links linked list
- * Initialize Document from file path
- * Implement the documents linked list and load all documents from the dataset
- * Print the documents to the CLI and allow the user to select and view one by index
- * Format your code with clang-format: \make f"
- *** Write at least 1 unit test for the document parsing functionality
- *** Write at least 3 unit tests for the documents linked list
- *** Write at least 3 unit tests for the links linked list
- *** Fix any memory leaks detected by Valgrind: \make v"

After lab 2 you should be able to write a query in the CLI (empty query should stop the program), find the first 5 documents that contain all the keywords from the query, and print them:

- * Implement the query linked list
- * Initialize Query from string
- * Linear search through the documents and print the results to CLI
- ** Show the last 3 queries using a queue
- ** Allow excluding keywords
- *** Write at least 3 unit tests for the query linked list
- *** Write at least 1 unit test for the linear search functionality
- *** Allow adding or conditions to the search query

- *** Fix any memory leaks detected by Valgrind: \make v"

After lab 3 you should be able to write a query and e f ciently print the frst 5 documents that contain the keywords to the CLI:

- * Implement a hashmap
- * Add all $(word, documentIds)$ to the reverse index hashmap
- * Use the reverse-index to e f ciently search through the documents
- ** Compare the search time with/out the reverse-index for each dataset
- ** Normalize words in the parser (uppercase, punctuation, etc.)
- *** Write at least 3 unit tests for the hashmap
- *** Write at least 1 unit test for the search functionality using hashmap
- *** Fix any memory leaks detected by Valgrind: \make v"
- *** Suggest and implement one improvement to lower the search runtime complexity
- *** Allow serializing/deserializing the reverse index from/to a fle
- *** Implement short and full barrels (match keywords in the title frst)
- *** Show document snippets of the text surrounding the matched word

After lab 4 you should be able to estimate document's relevance through the graph:

- * Implement a directed document graph
- * Get indegree of a document in the graph and print it as the relevance score
- ** Sort search results according to the relevance score (choose an adequate algorithm)
- *** Write at least 1 unit test for the graph
- *** Fix any memory leaks detected by Valgrind: \make v"
- *** Implement the page rank algorithm for computing the relevance score
- *** Precalculate and cache relevance scores
- *** Serialize/deserialize cache relevance scores from/to a fle
- (feel free to suggest other features to the TA/professor)

During lab 5 you will:

- Finish the project and ask any last minute questions.

Before 31/05/2025 at 23:59 you should deliver the project:

- Complete the code, tests, report and video. Remember to format all code: \make f".

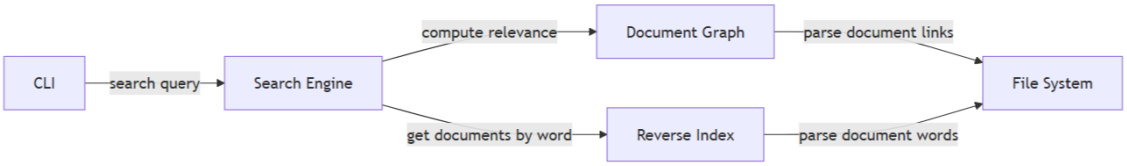
During lab 6 you will:

- Answer a test we will upload to Aula Global and be interviewed by your TA/professor.

Design overview

This project has 4 main components:

- **1. Parsing documents**. Parsing consists in reading the content of a document from a file character by character and storing the title, body and links to other documents in a struct. There are sample datasets of documents in the datasets folder.
- **2. Reverse-index**. Reverse-indexing consists in providing efficient access to all documents that contain a given word using a hashmap. For example: figuring out what documents contain the word \cat" without reading all documents.
- **3. Document graph**. The document graph is a directed graph, where each node is a document and each edge indicates a link from one document to another one. The indegree of a document (how many documents link to it) estimates the document's relevance score.
- **4. Search engine**. Finally, the search engine leverages the reverse-index (hashmap) to filter all documents that have all the words in the query. Then, it uses the relevance score from the document graph to return an ordered list of the most relevant documents.



To complete this project you will need to implement a linked list, a queue, a hash map and a directed graph. Additionally, you will need to implement algorithms for linear search and sorting. The following sections dive into each of these components, specify how you must implement them and provide hints.

1. Parsing documents

The template repository contains different datasets of documents inside the datasets folder. In the real world, this could be websites you crawled from the internet.

Each document complies with the following schema:

```
Title           =   *<TEXT; excluding LF>
DocumentId      =   INTEGER
LinkDestination =   DocumentId
LinkText        =   *<TEXT; excluding LBRACK and RBRACK>
Link            =   LBRACK
                    LinkText
                    RBRACK
                    LPAR
                    LinkDestination
                    RPAR
Body            =   *(
                        <TEXT; excluding LBRACK and RBRACK> |
                        Link
                    )
Document        =   DocumentId
                    LF
                    Title
                    LF
                    Body
                    EOF
```

TEXT is an alphanumerical character
LPAR is the left parenthesis \(" character
RPAR is the right parenthesis \)" character
LBRACK is the left bracket \[" character
RBRACK is the right bracket \]" character
LF is the newline character
EOF is the end of file character
INTEGER is one or more numeric characters without decimal point
* indicates repetition of the following schema element (0, 1 or more times)
| indicates an \or" between the type immediately to its left and right

(there is an explanation in the FAQ section at the end of the document to help you read these specifications)

Here is an example document from one of the datasets. And the following is a simpler but still valid example:

```
321
What is Python?
Python is a great language, learn today!
Python is an [interpreted](234), [object-oriented](276), high-level
[programming language](981) with dynamic semantics. Its [data structures](111),
combined with [dynamic typing](234) and [dynamic binding](321), make it very
attractive for rapid development, as well as for use as a [scripting]
(9872). See [comparisons between Python and C](99).
```

The documents should be represented in your program using a **Document structure** that contains: title (string), document id (integer), links (list of links), relevance (float) and body (string). The **Link structure** contains: id (integer). Then, you should create a parser which: (1) reads a document file given its file path, (2) parses the contents into a document structure, and (3) returns \NULL" if there are any errors:

```
Document* documentDesserialize(char* path)
```

The README has sample code for reading a line from a file. This is a hint showing how to parse the document and links:

```
char buffer[262144];
int bufferSize = 262144;
int bufferIdx = 0;
char ch;

// TODO parse id

// TODO parse title

// parse body
char linkBuffer[64];
int linkBufferSize = 64;
int linkBufferIdx = 0;
bool parsingLink = false;
Links *links = LinksInit();

bufferIdx = 0;
while ((ch = fgetc(f)) != EOF) {
  assert(bufferIdx < bufferSize);
  buffer[bufferIdx++] = ch;
  if (parsingLink) {
    if (ch == ')') { // end of link
```

```
        parsingLink = false;
        assert(linkBufferIdx < linkBufferSize);
        linkBuffer[linkBufferIdx++] = '\0';
        int linkId = atoi(linkBuffer);
        if (!LinksContains(links, linkId)) {
          LinksAppend(links, linkId);
        }
        linkBufferIdx = 0;
      } else if (ch != '(') { // skip first parenthesis of the link
        assert(linkBufferIdx < linkBufferSize);
        linkBuffer[linkBufferIdx++] = ch;
      }
    } else if (ch == ']') { // beginning of link id, e.g.: [my link text](123)
      parsingLink = true;
    }
  }
  assert(bufferIdx < bufferSize);
  buffer[bufferIdx++] = '\0';

  char *body = (char *)malloc(sizeof(char) * bufferIdx);
  strcpy(body, buffer);

  document->body = body;
  document->links = links;
```

2. Reverse index

The main question a search engine needs to answer is: what documents contain the keyword the user is searching for?

The most basic approach to searching is performing a **linear search**: given a query provided by the user, iterate through all the text of all the documents to find which documents contain the words. However, this approach is too slow when there are many documents or very lengthy documents.

```
# parameters
search_word, the word we want to find
all_documents, the list of all documents to search through
clean, a method which removes lowercases and accentuation marks

# query
found_documents = []
FOR document IN all_documents:
    FOR word IN document:
        IF clean(word) == search_word:
            found_documents.APPEND(document)
            BREAK
```

For efficient querying of the documents which contain any given word, we can implement a **hashmap mapping each word to a linked list of documents**. For every word, the dictionary contains a linked list of all documents that contain the word. This makes initialization slower, but querying faster.

```
# parameters
search_word, the word we want to find
all_documents, the list of all documents to search through
clean, a method which removes lowercases and accentuation marks

# initialization
reverse_index = {}
FOR document IN all_documents:
    FOR word IN document:
        clean_word = clean(word)
        IF NOT reverse_index.CONTAINS(clean_word):
            reverse_index.PUT(clean_word, [])
        word_docs = reverse_index.GET(clean_word)
        IF NOT word_docs.CONTAINS(document):
            word_docs.APPEND(document)
# query
found_documents = reverse_index.GET(clean(search_word))
```

For example:

```
Document 123:
    123
    How to code in Python?
    Python is a great language, learn today!
    To code in Python, go to python.org

Document 4:
    4
    How to code in Java?
    Java is a great language, learn today!
    To code in Java, go to java.org

Document 78:
    78
    How to code in C?
    C is a great language, learn today!
    To code in C, go to c.org

Reverse-index (hashmap):
    how -> 123, 4, 78
    to -> 123, 4, 78
    code -> 123, 4, 78
    in -> 123, 4, 78
    python -> 123
    java -> 4
    c -> 78
    is -> 123, 4, 78
    a -> 123, 4, 78
    great -> 123, 4, 78
    language -> 123, 4, 78
    learn -> 123, 4, 78
    today -> 123, 4, 78
    to -> 123, 4, 78
    code -> 123, 4, 78
    go -> 123, 4, 78
    python.org -> 123
    java.org -> 2
    c.org -> 3
```

You must divide the body in words. Then, for each of those words (key), add the document id to the list (value) of the map. Remember to handle the following edge cases:

- words which appear more than once in the same document

- words which are next to dots, commas, exclamation marks, etc., must be equivalent to those separated by spaces (**)
- words which are in uppercase must be equivalent to the lowercase alternatives (**)

Initializing the reverse index can be slow when documents are large or there are many documents, so you might want to implement serializing/de-serializing the reverse index to/from a file (***). You should use the following schema for persisting the reverse index in a file:

```
DocumentId          =   INTEGER
ReverseIndexLine    =   Word
                        *(SP DocumentId)
                        LF
ReverseIndex        =   *ReverseIndexLine
                        EOF
```

SP is the space \ " character

Another improvement for both efficiency and quality of search results is to create two reverse indexes: short reverse index and full reverse index (***). The short reverse index indexes words in the titles of articles. The full reverse index indexes words in the body. When searching use the short reverse index first.

3. Document graph

When searching for a word (e.g. \cat"), many documents (web pages) might contain that word. Thus, search engines need a way to sort results from most to least relevant. Most approaches rely on counting how many other documents link to any given document. The assumption is that if many documents link to one specifc document, it must be trustworthy, well-written and accurate (thus, relevant).

To easily compute how many documents link to any given document, we will **store documents in a directed graph**. For example:

```
Document 666:

    666
    The humans never went to the moon
    It was all fake and recorded on earth
    NASA faked going to the [moon](987),
    it was all an elaborate scheme to boost morale.

Document 987:

    987
    What is the moon?
    The Moon is a planetary-mass object or satellite planet
    In geophysical terms, the Moon is a planetary-mass.

Document 874:

    874
    What is the earth?
    Earth is the third planet from the Sun.
    Earth is the third planet from the Sun
    and the only astronomical object known to harbor life.
    The earth has a [moon](987): [the earth's moon](156).

Document 156:

    156
    What is the earth's moon?
    The Moon is Earth's only natural satellite
    The Moon is [Earth](874)'s only [natural satellite](987).
    The Moon makes [Earth](874) more livable, sets the rhythm of ocean tides,
    and keeps a record of our solar system's history.

Directed graph:

    666, 756, 874 -> 987
```

```
156 -> 874
874 -> 156
```

The **relevance score** of a document is the indegree of its node in the graph (the number of links pointing into a node). For example, \987" from the example above would have a relevance score of 3, while \666" would have a relevance score of 0:

```
float graphGetIndegree(DocumentGraph* graph, int documentId)
```

Then, use the relevance score to sort the results you print to the CLI (**):

```
DocumentsList *documentsListSortedDescending(DocumentsList *list);
```

You can compute a more **advanced relevance score** using the Page Rank algorithm (***):

```
- Graph G with n nodes (web pages / documents)
- Damping factor d (typically set to 0.85)
- Maximum number of iterations max_iterations
- Tolerance threshold tol

# initialize ranks uniformly
for node in G.nodes:
    current_rank[node] = 1/n
for iteration in range(max_iterations):
    new_rank = {}
    for node in G.nodes:
        # Initialize with (1 - d)/n, where n is the total number of nodes
        new_rank[node] = (1 - d) / n
    for node in G.nodes:
        for neighbor in G.neighbors(node):
            num_neighbors = len(G.neighbors(neighbor))
            # Update the rank of the neighbor
            new_rank[neighbor] += d * (current_rank[node] / num_neighbors)
    # Check for convergence
    max_diff = max(abs(new_rank[node] - current_rank[node]) for node in G.nodes)
    if max_diff < tol:
        break
    # Update the current rank with the newly calculated ranks
    current_rank = new_rank
```

To speed up the relevance score calculation, you might choose to pre-compute it for all documents during initialization (***) and even store/load them to/from a file (***).

4. Search engine

Before asking the user to write a query, you should print the most recent 3 searches using a queue (**).

Then, the user must be able to type a query in the CLI. A query consists of one or more space-separated keywords. For example: \cute white cat". The query must be split into the different keywords and the program must return the list of documents which contain all the keywords.

```
Keyword             =   *ALPHANUMERIC
QueryItem           =   Keyword
Query               =   <QueryItem *(*SP QueryItem); max 200 chars>
```

ALPHANUMERIC is a numeric or alpha character

You must create two structures. The Query must be a list of QueryItems. The QueryItem is a struct which contains a word. The DocumentList must be a list of documents. Then, you must implement two methods:

```
Query* queryInit(char* query);

DocumentsList* searchDocumentsWithQuery(
    ReverseIndex* index,
    Query* query
)
```

The output of the CLI must contain the title. Only the frst 150 characters of the body must be printed, adding an ellipsis (\...") at the end if it exceeds 150 characters. Only the top 5 results must be printed (or fewer if there's less than 5 matches). Then, the user must be able to select a document by index and view all of its content.

The results must be sorted in order of descending relevance (**).

More complex queries may contain exclusion keywords (**). For example: \cute white cat -sphynx -hairless". In this case, the program must return the list of documents which contain all the include keywords, but none of the exclude keywords.

```
IncludeQueryItem    =   Keyword
ExcludeQueryItem    =   DASH
                        Keyword
QueryItem           =   IncludeQueryItem |
                        ExcludeQueryItem
```

Even more complex queries may contain \or" conditions (***). For example: \cute white cat - sphynx (bobtail | longhair)". In this case, the program must return the list of documents which contain all the include keywords, none of the exclude keywords and at least one of the or-separated include keywords.

```
IncludeQueryItem    =   Keyword
ExcludeQueryItem    =   DASH
                        Keyword
OrQueryItem         =   LPAR
                        *SP
                        IncludeQueryItem
                        *SP
                        PIPE
                        *SP
                        IncludeQueryItem
                        *SP
                        RPAR
QueryItem           =   IncludeQueryItem |
                        ExcludeQueryItem |
                        OrQueryItem
```

PIPE is the pipe \|" character
DASH is the dash \−" character

Ideally, instead of printing the first characters of the body in the CLI, you should print the 75 characters before and 75 characters after the word that matched the query (***). If there are multiple words, choose the one which appears the earliest in the content. To do so efficiently, you might want to extend the reverse-index with the position of the word in the document. Also, you should distinguish between occurrences in the title and the body.

Report

You must write a report in \REPORT.md" which contains the following (and only the following) items:

- C4 component diagram showing the data structures (graphs, indexes, caches, ...), fles you read/write and their relation. Clearly distinguish components which are volatile (stored in memory) vs persistent (fles in disk) using two diferent colors.
- Table with 3 columns (description, Big-O, justifcation) with one row for each of the following (in this exact order):
  - Runtime complexity analysis of parsing a document into the struct (including adding the links to the list)
  - Runtime complexity analysis of parsing a query into the struct
  - Runtime complexity analysis of counting the neighbours in the graph
  - Runtime complexity analysis of counting the neighbours of a document in the graph
  - Runtime complexity analysis of fnding the documents that contain a keyword in the reverse-index
  - Runtime complexity analysis of fnding the documents that match all keywords in the query
  - Runtime complexity analysis of sorting the documents according to the relevance score
- Plot the search time with/without the reverse index for the diferent-sized datasets and discuss the results (2 sentences)
- Plot the initialization and search time for diferent hashmap slot count settings and discuss the results (2 sentences)
- Describe an improvement of the reverse index to improve search and initialization speed. Would it require the same, less or more memory overall? And what would the runtime complexity be (Big-O)? Would it take the same, less or more time to run?
- List any extra features you have implemented (*** and custom ones)
- A link to the video with the demo of your project ($<$ 180 seconds)

Use markdown to properly format your report in the repository.

FAQ

What editor should I use?

VSCode, but you are free to use whatever you want as long as the program runs from the CLI with \make r".

What OS should I use?

Linux or MacOS. Use the Windows Subsystem for Linux (WSL) to run Linux inside Windows. You could just use Windows, but the template repository won't work out of the box.

How much does each deliverable count?

There is a rubric available:

$Code = 0.5 * CodeGrade$

$Report = 0.063 * ReportGrade$

$Video = 0.063 * VideoGrade$

$Interview = 0.25 * InterviewGrade$

$Questionnaires = 0.125 * QuestionnairesGrade$

$Lab = Code + Report + Video + Interview + Questionnaires$

Many tasks mention I should write tests. What are unit tests?

Until now, you probably have written functions which solve specifc problems. For example, count the number of characters in a string, append an element to a linked list or fnd an element in a list. We will call these functions the **system under test (sut)**.

Every time you want to verify your function works as expected, you would call it from main and manually check that, given some inputs, it works as expected. This is called **manual testing**.

In contrast, we can write **automated tests**. Automated tests are functions which call the system under test and automatically verify that the result is the expected one. Automated tests allow developers to not have to re-test all functionalities after writing a new piece of code or refactoring it, as the tests run automatically in the computer.

When automated tests verify the behavior of a small unit or function in isolation, we call them **unit tests**. This is how a unit test looks in C:

```
void test_factorial(){
    // arrange
    int expectedResult = 24;

    // act
    int result = factorial(4);
```

```
    // assert
    assert(result == expectedResult);
}
```

We usually divide unit tests in 3 steps: \arrange" (setup the needed variables and data structures), \act" (call the system under test) and \assert" (verify the result is correct through asserts). The **assert function** errors out the program and stops execution when we pass a value different than 1 (true).

The template repository contains an example to help you get started writing unit tests. You can run the tests with the following command:

```
make t
```

Additionally, we can ask GitHub to automatically run unit tests through a workfow file. This has already been set up in the template repository so that you get a green checkmark when tests pass, or a red cross next to your commits when you break the tests. Building and automatically testing the code for each commit is called **Continuous Integration (CI)**.

What happens if one team member contributed less than the other ones?

If someone contributed signifcantly less than the other members when looking at the commit history, they may be individually awarded a 0 and fail the course.

What happens if I copy code I did not write?

If not clearly indicated with a code comment next to it, a fault may be noted in your academic expedient and you may fail the course and lose your right to attend a resit. The same happens if you don't commit often.

How should I read the format specifications in this document?

This notation is similar to BNF and is typically used in IETF RFCs. Consider the following example. It means the fle is composed of 0 or more lines, and then the \EOF" character. Each line starts with a word, and then 0 or more space separated document ids; each line ends with the newline character (\LF"). Every word is one or more text characters (alphanumeric), but cannot contain any spaces (\SP"). Finally, the document id is simply an integer.

```
Word =              *<TEXT; excluding SP>
DocumentId =        INTEGER
Line =              Word
                    *(SP DocumentId)
                    LF
File =              *Line
                    EOF
```