Informe de Laboratorio 6
Índice Invertido

Integrantes:

- Mayra Díaz Tramontana 201910147
- Maor Roizman Gheiler 201810323

Link del repositorio: **Github**

Estructura

El sistema de Recuperación de Información basado en Índice Invertido fue diseñado bajo la siguiente estructura de clases:

InvertedIndex:

Esta clase que se encarga del preprocesamiento y la creación del Índice Invertido.

- Maneja memoria secundaria para almacenar:
  - Tokens frecuentes ordenados alfabéticamente
  - Índice Invertido ordenado
  - Documentos procesados
- Permite añadir documentos a la colección
- Permite crear diferentes colecciones
- Permite saber la frecuencia de cada token, tanto la total como la frecuecia por libro.

OperatorHandler:

Esta clase es la encargada de interactuar con el InvertedIndex, de esta manera el usuario tiene únicamente las funciones necesarias para interactuar con el sistema (para ejecutar los query).

Funcionamiento:

Utiliza una función llamada addBook que añade un libro a la colección y actualiza el índice. Almacena una lista de libros para no duplicarlos y una lista ordenada alfabéticamente de los 500 tokens con mayor frecuencia.

```
def addBook(self, bookName, bookFileName):
    if bookName in self.books:
      return
    self.writeBook(bookName)
    text = self.getFileText(bookFileName)
    tokens = self.preProcess(text)
    self.addBookTokens(self.books[bookName], tokens)
    self.setFrequentTokens()
```

Preprocesamiento

Para el preprocesamiento realizamos los siguientes pasos:

1. Tokenizar el texto utilizando word_tokenize de la librería nltk.

```
tokens = []
for line in text:
  tokens += nltk.word_tokenize(line.lower())
```

Nota: text es una lista con las líneas leídas del documento.

2. Retirar los signos innecesarios utilizando un archivo personalizable con los símbolos a ignorar.

```
with open('symbols.txt') as file:
  symbol_list = [line.lower().strip() for line in file]
  for word in tokens:
    if word in symbol_list:
      tokens.remove(word)
```

3. Filtrar los stoptokens utilizando un archivo que contiene stoptokens del lenguaje español.

```
clean_tokens = []
with open(self.stopWordsFile) as file:
  stoplist = [line.lower().strip() for line in file]
for token in tokens:
  if not token in stoplist:
    cean_tokens.append(token)
```

4. Reducir palabras a su raiz utilizando SnowballStemmer.

```
stemmer = SnowballStemmer('spanish')
root_tokens = []
for word in clean_tokens:
  root_tokens.append(stemmer.stem(word))
return root_tokens
```

Construcción del índice invertido

Se itera la lista completa de tokens para actualizar el índice. Tenemos dos casos:

- El token ya se encuentra en el índice:
  Se actualiza su frecuencia total sumándole 1. Si ya apareció en ese libro se aumenta la frecuencia por el libro, caso contrario lo inicializa en 1.
- El token no se encuentra en el índice:
  Añade el token al índice e inicializa las frecuencias (total y en el libro).

```
for token in tokens:
  if token in self.index:
    self.index[token][0] += 1
    if bookId in self.index[token][1]:
      self.index[token][1][bookId] += 1
    else:
      self.index[token][1][bookId] = 1
  else:
    self.index[token] = [1, {bookId: 1}]
```

Aplicar Consultas Booleanas

Estregia: Siempre se mantienen las listas ordenadas para poder mantener constante la complejidad de las operaciones.

1. **OR:** Este operador retorna la unión de las dos listas. Se recorren ambas listas fijando un puntero al principio de cada una y recorriendolas con una estrategia comparativa que permite mantener el output ordenado.

```
    def OR(self, listA, listB):
      result = []
      i, j = (0,0)

      while i < len(listA) and j < len(listB):
        if listA[i] < listB[j]:
          result.append(listA[i])
          i+= 1
        elif listA[i] > listB[j]:
          result.append(listB[j])
          j+= 1
        else:
          result.append(listA[i])
          i += 1
          j += 1

      while i < len(listA):
        result.append(listA[i])
          i+= 1

      while j < len(listB):
        result.append(listB[j])
        j+= 1

      return result
```

2. **AND:** Este operador retorna la intersección entre las dos listas. Al igual que el operador **OR** se utiliza una estrategia que permite mantener ordenado el output. En este caso solo se agregan elementos a la lista resultante cuando los elementos comparados son iguales.

```
    def AND(self, listA, listB):
      result = []
      i, j = (0, 0)

      while i < len(listA) and j < len(listB):
        if listA[i] < listB[j]:
          i += 1
        elif listA[i] > listB[j]:
          j += 1
        else:
          result.append(listA[i])
          i += 1
          j += 1
      return result
```

3. **ANDNOT:** Este operador retorna la diferencia de la *listaA* menos la *listaB*.

```
    def ANDNOT(self, listA, listB):
      result = []
      i, j = (0,0)
      while i < len(listA) and j < len(listB):
      if listA[i] < listB[j]:
        result.append(listA[i])
        i += 1
      elif  listA[i] > listB[j]:
        j += 1
      else:
        i += 1
        j += 1

      while i < len(listA):
        result.append(listA[i])
        i += 1

      return result
```

Anexos

Memoria secundaria

- **InvertedIndex.json**

```
{"token":
    [
        frecuencia_total,
        {
            "id_libro": frecuencia_en_libro
        }
    ]
}
```

- **Books.txt**

```
    Gandalf
    Frodo
    Parth Galen
    Mordor
    Gandalf y Pippin
    Cirith Ungol
```

- **FrequentTokens.txt**

```
abaj
abiert
abism
acab

...

mordor
morgai
```

Inverted Index:

```
import os
import json
import nltk
import atexit
from nltk.stem.snowball import SnowballStemmer
nltk.download('punkt')
# from nltk.corpus import stoptokens
# nltk.download('stoptokens')

class InvertedIndex:
  def __init__(self, indexFile='./data/InvertedIndex.json',
booksFile='./data/Books.txt'):
    self.indexFile = indexFile
    self.booksFile = booksFile
    self.stopWordsFile = './stopWords/stop-words-spanish.txt'
    self.frequentTokensFile = './data/FrequentTokens.txt'
    self.index = {}
    self.books={}
    self.frequentTokens = []
    self.readIndexFile()
    self.readBooksFile()
    self.readFrequentTokensFile()
    atexit.register(self.cleanup)

  def readIndexFile(self):
    if os.path.exists('./'+self.indexFile):
      with open(self.indexFile, 'r') as file:
        self.index = json.load(file)
    else:
      f = open(self.indexFile, 'x')
      f.close()

  def readBooksFile(self):
    if os.path.exists('./'+self.booksFile):
      with open(self.booksFile) as file:
        i = 0
        for book in file:
          self.books[book[:-1]]=i
          i +=1
    else:
      f = open(self.booksFile, 'x')
      f.close()

  def readFrequentTokensFile(self):
    if os.path.exists('./'+self.frequentTokensFile):
      with open(self.frequentTokensFile) as file:
        for token in file:
          self.frequentTokens.append(token[:-1])
    else:
      f = open(self.frequentTokensFile, 'x')
      f.close()

  def cleanup(self):
    self.writeIndexJson()
    self.writeFrequentTokens()
```

```
def writeIndexJson(self):
  file = open(self.indexFile, 'w')
  file.write(json.dumps(self.index))
  file.close()

def writeFrequentTokens(self):
  file = open(self.frequentTokensFile, 'w')
  for token in self.frequentTokens:
    file.write(token+'\n')
  file.close()

def writeBook(self, bookName):
  self.books[bookName] = len(self.books)
  with open(self.booksFile, 'a') as file:
    file.write(f'{bookName}\n')

def getFileText(self, fileName):
  with open(fileName) as file:
    return [line.lower().strip() for line in file]


def preProcess(self, text):
  tokens = []
  # Tokenizar el texto
  for line in text:
    tokens += nltk.word_tokenize(line.lower())
  # Retirar signos innecesarios
  with open('symbols.txt') as file:
    symbol_list = [line.lower().strip() for line in file]
  for word in tokens:
    if word in symbol_list:
      tokens.remove(word)

  #Filtrar stoptokens
  clean_tokens = []
  with open(self.stopWordsFile) as file:
    stoplist = [line.lower().strip() for line in file]
  for token in tokens:
    if not token in stoplist:
      clean_tokens.append(token)

  # Reducir palabras a su raiz
  stemmer = SnowballStemmer('spanish')
  root_tokens = []
  for word in clean_tokens:
    root_tokens.append(stemmer.stem(word))
  #print(f'root_tokens: {root_tokens}')
  return root_tokens


def addBookTokens(self, bookId, tokens):
  for token in tokens:
    if token in self.index:
      self.index[token][0] += 1
      if bookId in self.index[token][1]:
        self.index[token][1][bookId] += 1
      else:
```

```
          self.index[token][1][bookId] = 1
      else:
        self.index[token] = [1, {bookId: 1}]

  def setFrequentTokens(self):
    self.index = dict(sorted(self.index.items(), key=lambda x: x[1][0],
reverse=True))
    self.frequentTokens = sorted(list(self.index.keys()))[:500]

  def addBook(self, bookName, bookFileName):
    if bookName in self.books:
      return
    self.writeBook(bookName)
    text = self.getFileText(bookFileName)
    tokens = self.preProcess(text)
    self.addBookTokens(self.books[bookName], tokens)
    self.setFrequentTokens()

  def recover(self, token): #L()
    if token in self.frequentTokens:
      return list(self.index[token][1].keys())
    raise Exception("Token not found in top 500 tokens.")

  def printTokens(self):
    print(self.index)
```

OperatorHandler

```
from InvertedIndex import InvertedIndex
from nltk.stem.snowball import SnowballStemmer


class OperatorHandler:
    def __init__(self):
        self.index = InvertedIndex()

    def addBook(self, bookName, bookFileName):
        self.index.addBook(bookName, bookFileName)

    def recover(self, token):
        stemmer = SnowballStemmer('spanish')
        root_token = stemmer.stem(token)
        return self.index.recover(root_token)

    def OR(self, listA, listB):
        result = []
        i, j = (0,0)

        while i < len(listA) and j < len(listB):
            if listA[i] < listB[j]:
                result.append(listA[i])
                i+= 1
            elif listA[i] > listB[j]:
                result.append(listB[j])
                j+= 1
            else:
```

```
                result.append(listA[i])
                i += 1
                j += 1

        while i < len(listA):
            result.append(listA[i])
            i+= 1

        while j < len(listB):
            result.append(listB[j])
            j+= 1

        return result


    def AND(self, listA, listB):
        result = []
        i, j = (0, 0)

        while i < len(listA) and j < len(listB):
            if listA[i] < listB[j]:
                i += 1
            elif listA[i] > listB[j]:
                j += 1
            else:
                result.append(listA[i])
                i += 1
                j += 1
        return result


    def ANDNOT(self, listA, listB):
        result = []
        i, j = (0,0)

        while i < len(listA) and j < len(listB):
            if listA[i] < listB[j]:
                result.append(listA[i])
                i += 1
            elif  listA[i] > listB[j]:
                j += 1
            else:
                i += 1
                j += 1

        while i < len(listA):
            result.append(listA[i])
            i += 1

        return result
```