LABORATORIUM VI – INDEKSOWANIE + LUCENE

1. Plan laboratorium VI

1.1. Indeks odwrotny

1.2. w-shingling

1.3. Suffix tree – algorytm naiwny oraz algorytm Ukkonena; zastosowanie struktur indeksujących

1.4. Suffix array – algorytm qsufsort

1.5. Lucene

2. Inverted index - indeks (plik) odwrócony

Struktura danych przechowująca poszczególne ***termy jako klucze***, oraz ***identyfikatory plików***, w których te termy wystąpiły jako wartości (najczęściej implementowana jako tablica hashująca lub drzewo binarne).

Cel: zwiększenie szybkości działania wyszukiwarki przy poniesieniu kosztu dodania informacji o dokumencie do indeksu

W indeksie najczęściej przechowywane są indeksy dokumentów (numery porządkowe). Dodatkowo dla każdego termu często przechowywana jest długość listy dokumentów, które go zawierają. W wersji pełnej indeksu przechowuje się pary (DocID, TermPos), gdzie TermPos jest pozycją termu w ramach dokumentu o identyfikatorze DocID.

Zwykle dokumenty dla danego termu są uszeregowane zgodnie z DocID. Inne pomysły polegają na wykorzystaniu tzw. statycznej jakości dokumentu (np. miary PageRank) lub miary TF.

3. w-shingling

- zbiór ciagłych podsekwencji tokenów w dokumencie
- „w” oznacza licznę tokenów w każdym podciągu
- wykorzystywany do badania podobieństwa dokumentów (w tym przede wszystkim wykrywania plagiatów) pod względem wspólnych sekwencji o określonym rozmiarze

$$sim(d_1, d_2) = \frac{|S(d_1) \cap S(d_2)|}{|S(d_1) \cup S(d_2)|}$$

4. Drzewo sufiksów - suffix tree

Prefiks – określenie początkowych znaków ciągu (np. S dla U jeśli U=SU’).

Sufiks – określenie końcowych znaków ciągu (np. S dla U jeśli U=U’S)

Drzewo sufiksów - struktura danych reprezentująca zbiór sufiksów danego słowa lub tekstu.

- umożliwia efektywną obsługę zapytań, które wymagają wyszukania w tekstach ciągów słów lub ciągów znaków
- indeksowany tekst jest traktowany jako jeden długi ciąg znaków;

Suffix trie - struktura drzewiasta do przechowywania ciągów znaków, której każdy możliwy sufiks można znaleźć na ścieżce od korzenia do któregoś liścia.

Drzewo sufiksów (suffix tree) to bardziej zwięzła reprezentacja struktury ‘suffix trie’ (każdy węzeł z pojedynczym potomkiem jest eliminowany (poprzez połączenie węzła z potomkiem w pojedynczy węzeł).

Drzewo sufiksów T (suffix tree) dla ciągu znaków S o długości m to drzewo o następujących właściwościach:

- posiada korzeń i jest skierowane
- m liści oznakowanych od 1 do m
- każda krawędź etykietowana podciągiem znaków S
- złączenie etykiet poszczególnych krawędzi na ścieżce od korzenia do liścia i daje sufiks i ciągu S (oznaczmy go jako $S_{i...m}$)
- każdy wewnętrzny wierzchołek ma co najmniej dwójkę dzieci
- krawędzie wychodzące z korzenia muszą zaczynać się od różnych znaków

Naive Suffix Tree Building (złożoność $O(m^2)$)

Dla i=1 do m

Dodaj sufiks $S_{i...m}$ do T, znajdując najdłuższy pasujący prefiks $S_{i...m}$, który jest już w T i rozgałęziając od tego miejsca

Algorytm Ukkonena

On-line – zaczyna od pierwszego znaku, potem drugi, itd.

Wersja podstawowa (złożoność $O(m^3)$)

Zbuduj I1 (dodaj pierwszy znak do drzewa).

Dla i=1 do m-1

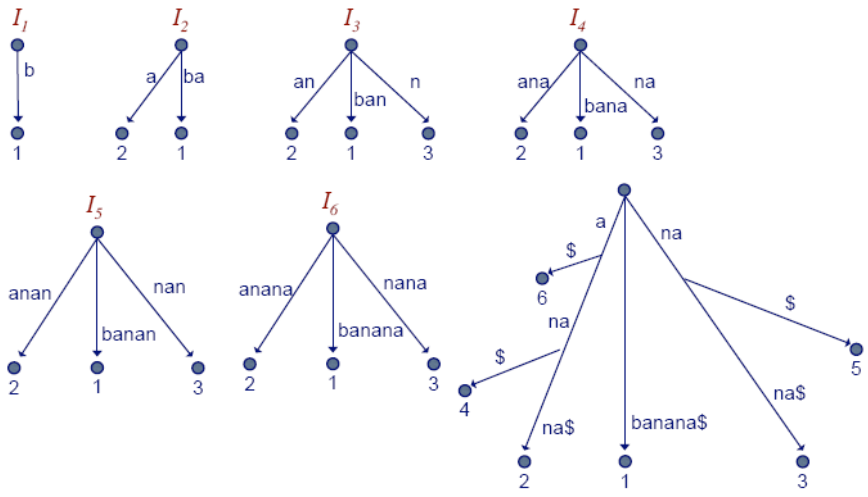
Dla j = 1 do i+1

Znajdź ścieżkę od korzenia zaetykietowaną $S_{j...i}$

Dodaj znak S_{i+1} na koniec ścieżki, jeśli to konieczne

Trzy reguły dodawania znaku na koniec ścieżki

Reguła 1	Reguła 2	Reguła 3
Jeśli ścieżka $S_{j...i}$ w drzewie kończy się liściem, dodaj S_{i+1} na koniec etykiety	Jeśli w drzewie są ścieżki $S_{j...i}$, które nie kończą się w liściu i nie następuje po nich S_{i+1}, utwórz nowy liść z etykietą S_{i+1} na końcu ścieżki $S_{j...i}$ (tworząc nowy wierzchołek wewnętrzny, jeśli $S_{j...i}$ kończy się w środku etykiety)	Jeśli w drzewie są ścieżki $S_{j...i}$, które nie kończą się w liściu, ale po jednej z nich następuje S_{i+1}, nie rób nic
$S_{j...i}$ = ...an $S_{j...i+1}$ = ...ana	$S_{j...i}$ = ...na $S_{j...i+1}$ = ...nay	$S_{j...i}$ = ...na $S_{j...i+1}$ = ...nan



Zmniejszenie złożoności opiera się na dwóch obserwacjach:

I. „Once a leaf always a leaf” – na krawędziach do liści można dodawać znak bez zastanowienia.

II. Jeśli w drzewie jest już jakiś sufiks, to wszystkie jego sufiksy też są już w drzewie, czyli jeśli w danej iteracji wykorzystano regułę 3 do rozszerzenia j, to można ją zastosować (czyli nic nie robić) dla wszystkich dalszych rozszerzeń w danej iteracji

Złożoność O(m)

Zbuduj I1.

Dla i=1 do m-1

Dla j takiego, że jL < j < jR

Znajdź ścieżkę od korzenia zaetykietowaną $S_{j...i}$

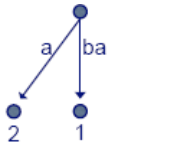
Dodaj znak S_{i+1} na koniec ścieżki

jL - w iteracji I to indeks ostatniego dodanego liścia

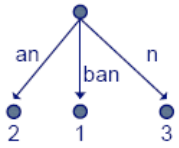
jR – w iteracji I to pierwszy indeks j taki, że $S_{j...i+1}$ jest już w drzewie

Wywołań pętli jest 2m; to, co jest wykonywane, da się zrobić w stałym czasie.

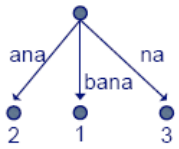
phase (*i*)	*j*	extension	rule
1	j_L → 1	ba	1
	2	a	2
	j_R →		



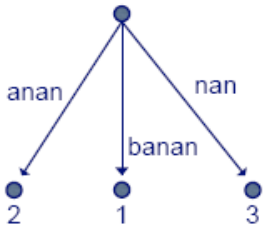
2	1	ban	1
	j_L → 2	an	1
	3	n	2
	j_R →		



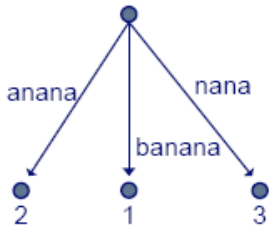
3	1	bana	1
	2	ana	1
	j_L → 3	na	1
	j_R → 4	a	3



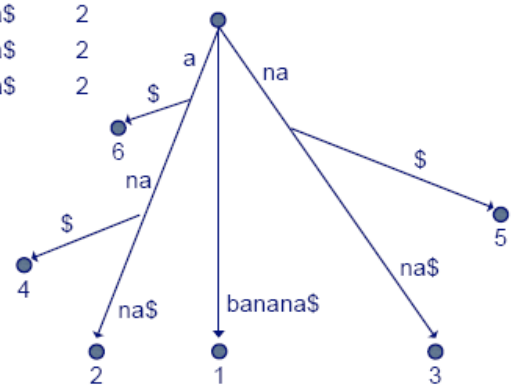
4	1	banan	1
	2	anan	1
	j_L → 3	nan	1
	j_R → 4	an	3
	5	n	3



5	1	banana	1
	2	anana	1
	j_L → 3	nana	1
	j_R → 4	ana	3
	5	na	3
	6	a	3



6	1	banana$	1
	2	anana$	1
	j_L → 3	nana$	1
	4	ana$	2
	5	na$	2
	6	a$	2
	j_R →		



Podstawowe zastosowania:

- znalezienie podciągu znaków P w tekście S;
- najdłuższa powtarzająca się sekwencja znaków w ramach jednego ciągu znaków;
- najdłuższy wspólny podciąg kilku ciągów znaków (alibaba.taliban.) – problem podciągów dla kolekcji wzorców
- najdłuższy palindrom (what do you think, do geese see God? -> whatdoyouthink dogeeseseegod.dogeeseseegodknihtuoytahw)
- problem dopasowania par suffix-prefix (dla ciągów Si oraz Sj, znajdź najdłuższą parę suffix Si i prefix Sj, które do siebie pasują – DNA sequence assembly, EST alignment)
- zanieczyszczenie DNA (DNA contamination) – procesy laboratoryjne powoduje wniknięcie niechcianego DNA do ciągu będącego przedmiotem badań; sekwencje DNA wielu możliwych „zanieczyszczaczy" są znane – problem: mając dane S1 oraz zbiór ciągów S (potencjalne zanieczyszczenia), znajdź podciągi S, które pojawiają się w S1 i są dłuższe niż zadane r

5. Tablica sufiksów – suffix array

Każdy sufiks może być jednoznacznie określony za pomocą indeksu swojego pierwszego znaku.

Tablica sufiksów to tablica indeksów sufiksów posortowanych w porządku leksykograficznym.

Tablica sufiksów jest zawsze permutacją indeksów sufiksów.

Tablica sufiksów

Suffix S(i)	I
banana$	0
anana$	1
nana$	2
Ana$	3
Na$	4
a$	5
$	6

-> sort ->

Suffix S(i)	I[i]
$	6
a·$	5
a·na·$	3
ana·na$	1
banana$	0
na·$	4
na·na$	2

Główne zastosowania:

Exact matching (wzór P[1…n] poszukiwany w T[1….m] – P jest podciągiem T jeśli jest prefiksem jakiegoś sufiksu w T; pozycja początkowe sufiksow w T, które zaczynają się od tego samego prefiksu znajdują się obok siebie w tablicy sufiksów; znajdź najmniejszy i największy indeks, w którym zaczyna się dany prefiks, wykorzystując np. wyszukiwanie binarne)

Substring problem

Tworzenie - algorytm qsufsort (Larsson, Sadakane)

I. W tablicy I umieść posortowane indeksy sufiksów 0,…,n. Do sortowania wykorzystaj znak na pozycji i-tej. Ustaw h=1.
II. Dla każdego i 0 [0,n], oblicz V[i] - numer grupy sufiksu i jako najgorszą pozycję w I jaką może zajmować sufiks zaczynający się od tej samej litery co sufiks i-ty.
III. Każdą grupę, która nie jest jeszcze posortowana posortuj zgodnie z ternary-split Quicksort, używając V[i+h] jako klucza dla sufiksu i-tego.
IV. Uzupełnij I[i]. Uzupełnij V[i]. Pomnóż h przez 2.
V. Jeśli I składa się z jednej posortowanej grupy, stop. W przeciwnym razie, idź do punktu III.

6. Ćwiczenia

1. Utwórz indeks odwrócony dla następujących dokumentów:

D1: new Home sales top forecasts

D2: home sales rise in july

D3: increase in home sales in july

D4: july new home sales rise

Jakie są wyniki wyszukiwania dla Zapytań:

sales AND rise

forecasts OR increase

Term/Doc	D1	D2	D3	D4
forecasts	1	0	0	0
home	1	1	1	1
in	0	1	1	0
increase	0	0	1	0
july	0	1	1	1
new	1	0	0	1
rise	0	1	0	1
sales	1	1	1	1
top	1	0	0	0

forecasts	
home	
In	
increase	
july	
new	
rise	
sales	
top	

2. Utwórz w-shingling dla tekstu „a banana is a banana is a banana” dla w=4.

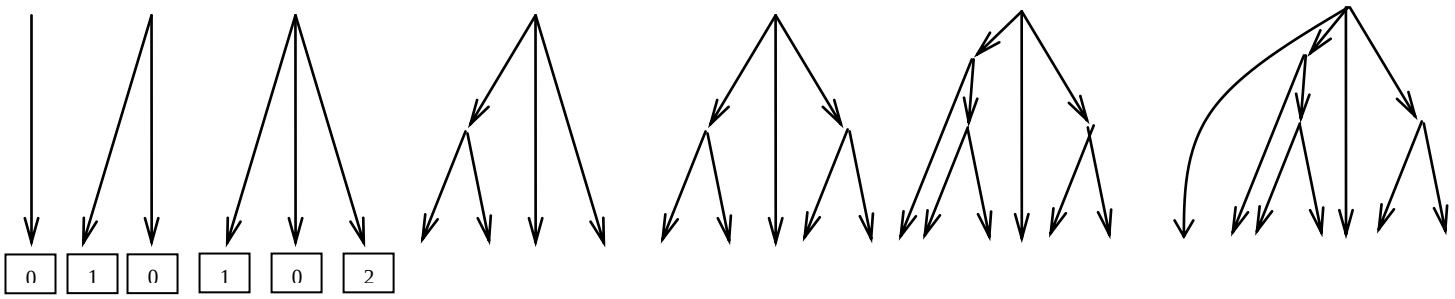
tokeny = (a, banana, is, a, banana, is, a, banana)

zbiór wszystkich kolejnych sekwencji 4 tokenów (N-gramów, 4-gramów) to:

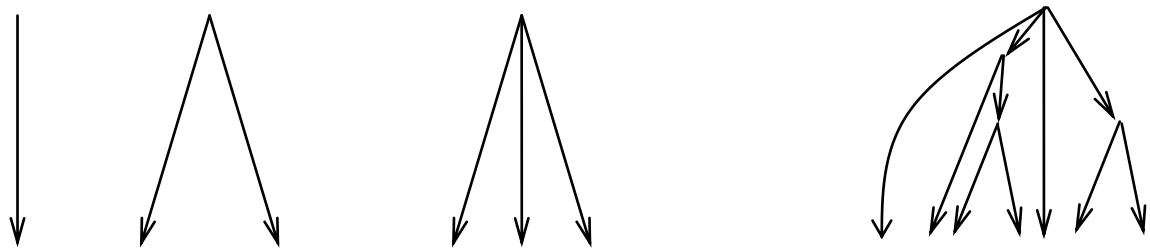
{(a, banana, is, a), (banana, is, a, banana), (is, a, banana, a), (a, banana, is, a), (banana, is, a, banana)}

4-shingling = { }

3. Utwórz drzewo sufiksów dla słowa „banana$”, wykorzystując algorytm naiwny.



oraz algorytm Ukonnena



4. Utwórz tablicę sufiksów dla słowa „banana$”, korzystając z algorytmu qsufsort.

	0	1	2	3	4	5	6	
X = [	b	a	n	a	n	a	$	]
I[i] = [								]
V[I[i]] = [								]
V[I[i+(h=1)]] = [								]
I[i] = [								]
V[I[i]] = [								]
V[I[i+(h=2)]] = [								]
I[i] = [								]

7. Zadania do samodzielnego wykonania

1. [1] Utwórz indeks odwrócony dla następujących dokumentów:

   D1: breakthrough drug for schizophrenia

   D2: new schizphrenia drug

   D3: new approach for treatment of schizophrenia

   D4: new hopes for schizophrenia patients

   Jakie będą wyniki wyszukiwania dla zapytań:

   Q1: schizophrenia AND drug

   Q2: for AND NOT(drug OR approach)

   Dla ułatwienia w arkuszu podano macierz term-dokument.

2. [1] Pokaż proces tworzenia tablicy sufiksów (suffix array) dla frazy „tobeornottobe$" z wykorzystaniem algorytmu qsufsort.

3. [3] Celem ćwiczenia jest zapoznanie się z Lucene Java API. Lucene to biblioteka udostępniająca funkcje niezbędne dla zastosowań z dziedziny Information Retrieval, np. indeksowanie dokumentów czy tworzenie rankingu stron dla zapytań. Udostępnia ona narzędzia takie jak parsery, systemy indeksujące i rangujące, pozwalając skupić się na projekcie architektury systemu. Co więcej, Lucene udostępnia proste algorytmy tokenizacji, stemmingu, itd., które mogą być zastąpione przez bardziej zaawansowane moduły w zależności od wymagań użytkowników.

Opis zadania jest długi – w rzeczywistości zadanie jest proste i większość rozwiązania znajduje się bezpośrednio w poniższym opisie. Zadanie sprowadza się do dopisania kilkunastu linii kodu, z których większość jest podana w treści ćwiczenia.

Zadanie polega na utworzeniu prostej wyszukiwarki, działającej na zadanej kolekcji tekstów. Szkielet rozwiązania znajduje się w klasie **LuceneLab6.java**. Klasę należy uzupełnić zgodnie z poniższymi instrukcjami. Wywołanie programu wymaga podania dwóch paramterów: args[0] - nazwa katalogu z kolekcją tekstów, args[1] – nazwa katalogu, w którym zostanie utworzony indeks. Zapytania jednowyrazowe podawane są przez użytkownika z poziomu konsoli.

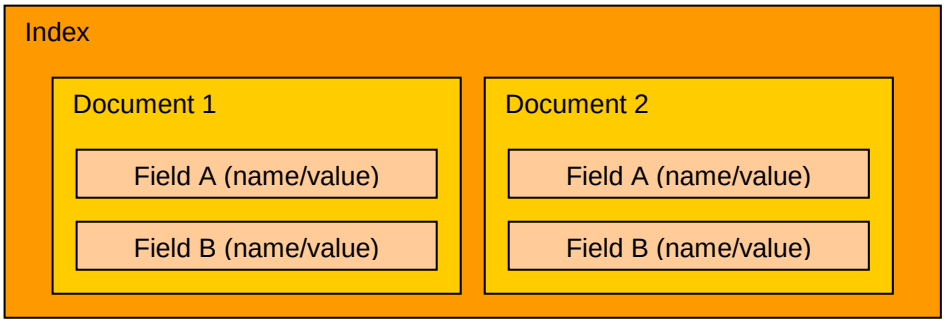
Część 1 - ustawienia środowiska

Ściągnij bibliotekę Lucene z katalogu lab6. Upewnij się, że umieściłeś .jar w katalogu Twojego projektu. W LuceneLab6.java dokonano już deklaracji importu wszystkich klas, których będziesz potrzebował dla celów zadania.

Kolekcja tekstów na której będziesz pracował znajduje się w katalogu lab6. Są to teksty Shakespeare pobrane ze strony projektu Gutenberg. Ścieżkę do kolekcji podajesz jako pierwszy parametr wywołania programu.

Część 2 - utworzenie indeksu, przeczytanie dokumentów, utworzenie reprezentacji dokumentów, wypełnienie indeksu

Napisz funkcję ***createIndex***, której celem jest utworzenie i wypełnienie indeksu dla zadanej kolekcji tekstów. Parametrem funkcji jest ścieżka do kolekcji tesktów. Indeks w Lucene wygląda następująco:



Do utworzenia indeksu wykorzystaj klasę IndexWriter :

```
Path iPath = Paths.get(indexPath);
Directory directory = FSDirectory.open(iPath);
Analyzer analyzer = new StandardAnalyzer();
IndexWriterConfig indexWriterConfig = new IndexWriterConfig(analyzer);
indexWriterConfig.setOpenMode(IndexWriterConfig.OpenMode.CREATE);
IndexWriter indexWriter = new IndexWriter(directory, indexWriterConfig);
```

gdzie pierwszy parametr jego konstuktora wskazuje na katalog, w którym ma być przechowywany indeks, drugi parametr to klasa zawierająca ustawienia, takie jak: typ "analizatora" (parsera dokumentu), który będzie wykorzystywany; tryb otwarcia - tutaj indeks zawsze jest nadpisywany (CREATE).
StandardAnalyzer jest najczęściej wykorzystywanym analizotorem ogólnego użycia. Inne możliwe to: WhiteSpaceAnalyzer (prosty analizator, które separuje tokeny na spacjach), StopAnalyzer (usuwa słowa stop-words), SnowballAnalyzer (używa pewnej formy stemmingu).

Utwórz obiekt klasy ***File***, korzystając z konstruktora, który jako parametr przyjmuje ścieżkę do kolekcji tekstów. Następnie wykorzystaj odpowiednią metodę klasy File tak, by nazwy dokumentów w tym katalogu reprezentować jako tablicę ***String[]*** i następnie odwołując się do tych nazwy, moć w pętli dodawać dokumenty do indeksu.

Po utworzeniu indeksu, wypełnij go zawartością, dodając poszczególne dokumenty. Każdy dokument odpowiada obiektowi klasy ***Document***. Dokument jest opisany za pomocą wielu własności w formie (nazwa, wartość). Każda własność odpowiada obiektowi klasy ***Field***. Korzystając z tych obiektów, można dodać informacje nt. dokumentu, takie jak tytuł, ścieżka do miejsca przechowywania, treść, itd.
Na przykład:

```
FileInputStream file = new FileInputStrea (docPath);
Document doc = new Document();

Field pathField = new StringField("path", docPath, Field.Store.YES);
doc.add(pathField);

Field contentField = new TextField("content",new InputStreamReader(file));
doc.add(contentField);
```

gdzie docPath zawiera ścieżkę do dokumentu; Field.Store.YES określa, czy własność jest przechowywana w ramach indeksu (użyteczne, jeśli chce się ją wyświetlić w ostatecznych wynikach), Klasa StringField: A field that is indexed but not tokenized: the entire String value is indexed as a single token. Klasa TextField: A field that is indexed and **tokenized**, without term vectors. ***Napisz osobną funkcję indexDoc dla utworzenia obiektu klasy Document.***

Po zdefiniowaniu obiektu klasy Document, dodaje się go do indeksu za pomocą polecenia:

indexWriter.addDocument(doc);

Następnie należy zakończyć tworzenie indeksu poleceniem:

indexWriter.close();

Część 3 - przeszukiwanie kolekcji

Jeśli utworzyłeś indeks, możesz przystąpić do napisania części przetwarzającej zapytania. Wykorzystaj klasy ***IndexSearcher*** oraz ***QueryParser***:

```
Path path = Paths.get(indexPath);
Directory directory = FSDirectory.open(path);
IndexReader indexReader = DirectoryReader.open(directory);
IndexSearcher indexSearcher = new IndexSearcher(indexReader);
```

gdzie ***indexPath*** to zmienna String, która zawiera ścieżkę do uprzednio utworzonego indeksu.

Dla zdefiniowania parsera zapytania, musisz wiedzieć, które pola dokumentu mają być analizowane pod względem zgodności z zapytaniem, a także jakiego typu parser zostanie użyty do parsowania zapytania (musi być on taki sam, jak parser dokumentu), np.:

```
Analyzer analyzer = new StandardAnalyzer();
QueryParser queryParser = new QueryParser("content", analyzer);
```

Parsowanie zapytania i wyszukiwanie dokumentów odbywa się za pomocą wykorzystania odpowiednich metod klasy ***QueryParser*** i ***TopDocs***:

```
Query query = queryParser.parse(queryString);
TopDocs topDocs = indexSearcher.search(query, 5);
ScoreDoc hits[] = topDocs.scoreDocs;
```

Metoda search zwraca ranking adekwatnych dokumentów (poprzez klasę ***Hits***). **Znalezienie wyników powinno być zaimplementowane w funkcji processQuery().**

Klasa Hits udostępnia iterator, z użyciem którego można przeglądać adekwatne dla zapytania dokumenty i miary podobieństwa. Należy wypisać te wartości, jak w poniższym przykładzie.

Przykładowe wyjście progamu:

```
Please enter your query: (lab6 to quit)
queen
12 result(s) found
Shakespeare/0ws4210.txt : 0.09321891
Shakespeare/0ws1410.txt : 0.04346354
Shakespeare/0ws0110.txt : 0.038874973
Shakespeare/0ws0210.txt : 0.038874973
Shakespeare/0ws2610.txt : 0.038874973
Shakespeare/0ws0410.txt : 0.03366671
Shakespeare/0ws0310.txt : 0.027488753
Shakespeare/0ws1710.txt : 0.027488753
Shakespeare/0ws0910.txt : 0.019437486
Shakespeare/0ws1210.txt : 0.019437486
Shakespeare/0ws1910.txt : 0.019437486
Shakespeare/0ws3910.txt : 0.019437486
Please enter your query: (lab6 to quit)
arthur
2 result(s) found
Shakespeare/0ws1410.txt : 0.16537577
Shakespeare/0ws2110.txt : 0.033757187
```

W razie potrzeby dokumentacja Lucene API jest do znalezienia w sieci, ale do wykonania tego zadania wystarczy realizacja (ze zrozumieniem) poleceń z powyższego opisu.

3. [4] Zadanie polega na wykorzystaniu Lucene do zaimplementowania metod indeksowania i wyszukiwania.
Kolekcją dokumentów jest RSS utworzony z BBC News – bbc_rss_feed.xml.
Dana jest klasa do parsowania RSS (RSSFeedParser), która zwraca dokumenty jako obiekty Javy zawierające istotne z punktu naszego zastosowania pola: tytuł, opis oraz datę publikacji (klasa RssFeedDcoument).
Klasą, którą należy uzupełnić to LuceneSearchApp (funkcje *index* oraz *search*):
Na etapie **indeksowania** dokumentów trzy pola (tytuł, opis, data) muszą być rozróżniane tak by zapytania mogły dotyczyć zawartości poszczególnych pól.
Metoda **wyszukiwania** powinna zwracać tytuły dokumentów, które odpowiadają zapytaniu.

Szkielet klasy LuceneSearchApp zawiera funkcję *main* do testowania Twojej implementacji metod indeksowania oraz wyszukiwania. Wywoływana jest w niej kolejno: istniejąca funkcja parsowania RSS (ścieżka do pliku RSS jest podawana jako argument wywołania w linii komend), funkcja indeksowania dokumentów oraz funkcje generujące kolejne zapytania testowe i wypisujące wynik.
W pliku test-output.txt dane jest wzorcowe wyjście, do którego możesz przyrównywać swoje rozwiązanie.

Implementacja wyszukiwania powinna uwzględniać następujące parametry:

- słowa, które występują w tytule (operator logiczny AND);
- słowa, które nie występują w tytule (operator logiczny NOT);
- słowa,które występują w opisie (operator logiczny AND);
- słowa, które nie występują w opisie (operator logiczny NOT);
- zakres daty publikacji (operator logiczny AND dla różnych dat)
- daty w formacie RRRR-MM-DD
  zakresy powinny być typu [początek, koniec], by data końcowa i początkowa były zawarte w przedziale (a nie typu (start, koniec))
  jeżeli data początkowa lub końcowa nie jest podana, przyjmuje się przedział otwarty z tej strony (nieograniczony)

Funkcja *search* przyjumuje więc na wejście sześć parametrów: tablice stringów inTitle, notInTitle, inDescription, notInDescription oraz stringi startDate oraz endDate.
Jako rozwiązanie trzeba przesłać kod źródłowy klasy LuceneSearchApp.

Dodatkowe 2 punkty (bonusowe w stosunku do 4 przewidzianych za to zadanie i nie zwiększające sumarycznej liczby punktów) można zdobyć, implementując własną klasę EziScoreQuery (rozszerzającą CustomScoreQuery), która w metodzie getCustomScoreProvider obliczałaby inną niż domyślna funkcja podobieństwa. Standardowo przy utworzeniu nowego IndexSearcher, Lucene wykorzystuje DefaultSimilarity, które jest miarą kosinusową opartą na tf-idf.
Dla uproszczenia możemy uznać, że funkcja będzie wykorzystywana tylko dla zapytań o słowa, które występują w tytule i/lub opisie i może polegać na prostym zliczaniu tych słów (dokument tym bardziej jest podobny, im częściej słowa z zapytania w nim występują).
Dla przejrzystości i ułatwienie sprawdzania, wersja standardowa rozwiązania (bez nowej funkcji podobieństwa) niech zostanie w LuceneSearchApp, a dla wersji rozszerzonej zróbcie nową klasę LuceneSearchSimApp.