Information Retrieval

Boolean Model

Matteo Ferfoglia
University of Trieste

- Introduction
- Boolean model
- Data structures
- Queries
- Configurability and extensibility
- Evaluation
- How to compile and execute

Outline

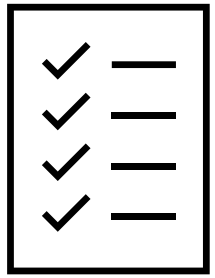
Introduction

Introduction

- Requirements
- Main goals
- Demo
- Platform
- Datasets

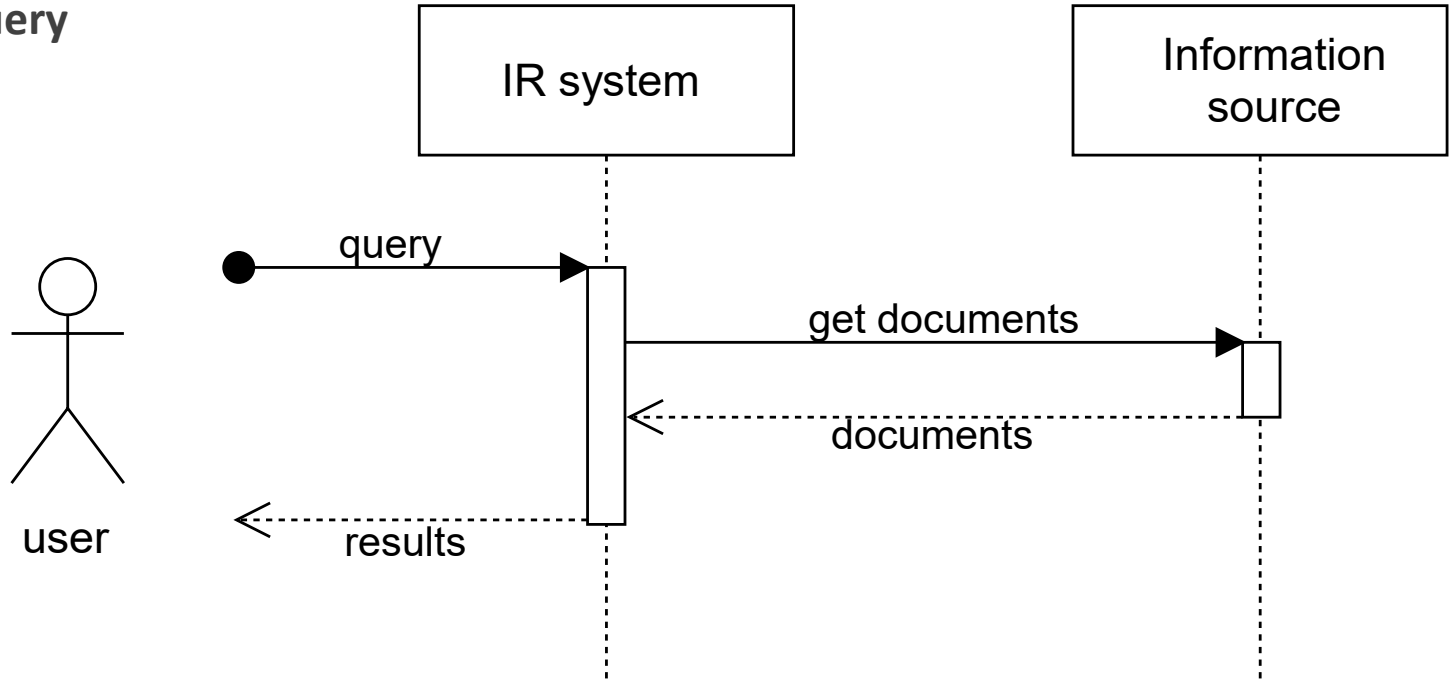
Requirements

- IR system able to answer
  - AND, OR, NOT **boolean queries**
  - **Wildcard** queries
  - **Phrase** queries
- **Normalization** or **stemming** can be performed
- **Spelling correction** can be implemented
- **Evaluation** of the system



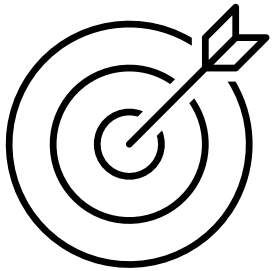
Main goals

- Satisfy an **information need**
  - formulated with a (**boolean**) **query**



Main goals

- Efficiency
  - Technical aspects
  - **Representation** of information
  - **Fast** retrieval
- Effectiveness
  - Stored information preserves its **meaning**
  - Retrieval of **relevant** documents



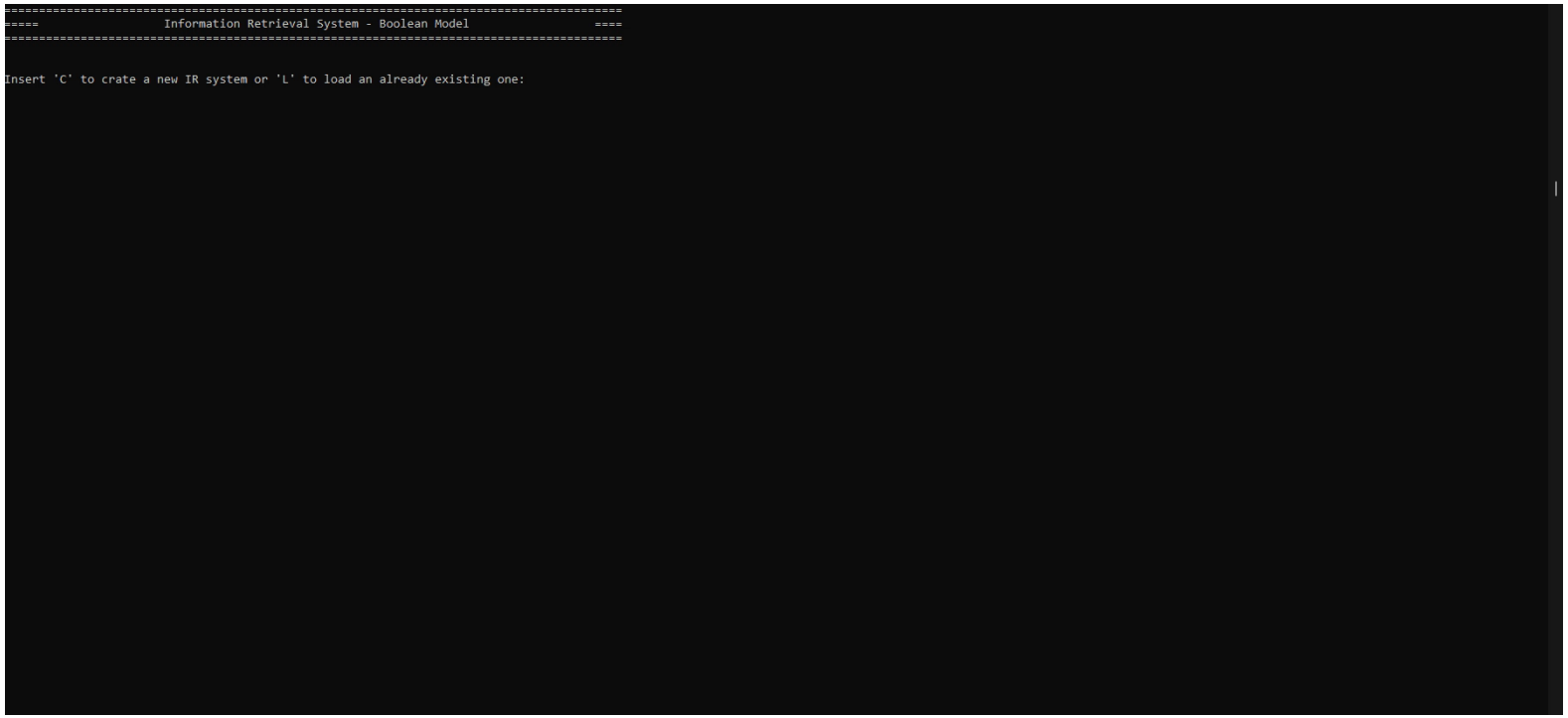
Demo

- IR System creation

```
================================================================================
=====                 Information Retrieval System - Boolean Model                 ====
================================================================================


Insert 'C' to crate a new IR system or 'L' to load an already existing one: c

Available collections to index:
        1)     Cranfield collections
        2)     Movie corpus
Insert the number of the collection that you want to index: 2
Creating IR system, please wait...
Indexing started
3.7 %    10.42 %         16.43 %         22.33 %         28.38 %         33.64 %         39.38 %         45.07 %
        50.83 %         56.06 %         60.42 %         65.29 %         70.63 %         75.86 %         81.11 %
        85.41 %         89.87 %         94.36 %         98.88 %         100.0 %
Indexing ended
Do you want to save the IR system to file? [y/n]: y
File "workingDirectory\irs\Movie corpus" created.
Serializing and saving the system on file, please wait...
IR System saved to file workingDirectory\irs\Movie corpus

System ready

INSTRUCTIONS
Insert:
       '-p' before the query string to enable the phonetic correction,
       '-s' before the query string to enable the spelling correction,
       '-a' before the query string to enable the automatic "word-wise"
            spelling correction, performed directly by the system,
            without specifying any further parameters (automatic
            correction will work only if no results are found by the
            system for the given query),
      the query string without any flags to retrieve exact matches only.
A combination of the previous flags can be used together. For phonetic
and spelling corrections, the number of corrections to attempt can be
specified by adding the number (the edit distance) after the flag (e.g.,
   -p2 foo&bar
means to try to correct (phonetic correction) two times; this does
not mean that the final query string (after the correction) will have
edit distance of 2 from the initial query string, but it means that
it will have an edit distance of at least 2, in fact, suppose that
no corrections are available at distance 1, so the system automatically
increases the edit-distance to 2 and re-try: after one single attempt
of correction, in this example, the overall edit-distance will be 2,
but in the input string we specified to try to correct 2 times, hence
another correction attempt will be made and the resulting edit distance
will be >=2).
The query string can have AND, OR or NOT operations (use the symbols '&',
'|', '!' respectively, spaces are interpreted as AND queries).
The system recognizes parenthesis, which can be used in any query string
to specify the precedences.

Insert a query or '-q' to exit:
-q
```

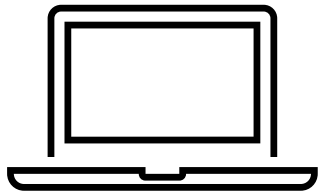
Demo

- IR System already created and now loaded



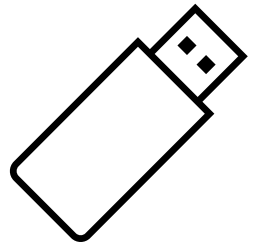
Platform

- Java 16
- Maven 3.5
  - Build automation tool
- Junit 5
  - Unit testing framework
- Operating system
  - Windows 11 Home, 64 bits
- Hardware
  - CPU 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
  - RAM 16.0 GB



Datasets

- Movie corpus
  - 87.9 MB
- Cranfield collection
  - 1.56 MB



Datasets

- Movie corpus
  - CMU Movie Summary Corpus
  - 81,741 movies metadata
    - 42,306 movie plot summaries and metadata
  - tab-separated values
  - not all metadata are available for all the movies
  - From: http://www.cs.cmu.edu/~ark/personas/

Wikipedia movie ID	Movie summary
23890098	Shlykov, a hard-working taxi driver and Lyosha, a saxophonist, develop …
31186339	The nation of Panem consists of a wealthy Capitol and twelve poorer ...
20663735	Poovalli Induchoodan is sentenced for six years prison life for murdering …
2231378	The Lemon Drop Kid , a New York City swindler, is illegally touting horses …
…	…

Figure plot summaries of the movie corpus

Wikipedia movie ID	Freebase movie ID	Movie name	Movie release date	Movie box office revenue	Movie runtime	Movie languages	Movie countries	Movie genres
23890098	/m/076w2lb	Taxi Blues	1990-09-07		110.0	{"/m/06b_j": "Russian Language"}	{"/m/0f8l9c": "France", "/m/05vz3zq": "Soviet Union", "/m/06bnz": "Russia"}	{"/m/07s9rl0": "Drama", "/m/03q4nz": "World cinema"}
31186339	/m/0gkz15s	The Hunger Games	2012-03-12	686533290	142.0	{"/m/02h40lc": "English Language"}	{"/m/09c7w0": "United States of America"}	{"/m/03btsm8": "Action/Adventure", "/m/06n90": "Science Fiction", "/m/02kdv5l": "Action", "/m/07s9rl0": "Drama"}
20663735	/m/051zjwb	Narasimham		2000	175.0	{"/m/0999q": "Malayalam Language"}	{"/m/03rk0": "India"}	{"/m/04t36": "Musical", "/m/02kdv5l": "Action", "/m/07s9rl0": "Drama", "/m/01chg": "Bollywood"}
2231378	/m/06xtz3	The Lemon Drop Kid	1951-03-08	2300000	91.0	{"/m/02h40lc": "English Language"}	{"/m/09c7w0": "United States of America"}	{"/m/06qm3": "Screwball comedy", "/m/01z4y": "Comedy"}
…	…	…	…	…	…	…	…	…

Figure metadata of the movie corpus

Datasets

- Cranfield collection
  - 1400 abstracts of scientific articles about astronautics
  - Used for system evaluation

.I 1
.T
experimental investigation of the aerodynamics of a
wing in a slipstream .
.A
brenckman,m.
.B
j. ae. scs. 25, 1958, 324.
.W
experimental investigation of the aerodynamics of a
wing in a slipstream .
an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios . the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
the comparative span loading curves, together with
supporting evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or
boundary-layer-control effect . the integrated remaining lift
increment, after subtracting this destalling lift, was found to agree
well with a potential flow theory .
an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .
.I 2
.T
simple shear flow past a flat plate in an incompressible fluid of small
viscosity .
.A
ting-yili
.B
department of aeronautical engineering, rensselaer polytechnic
institute
troy, n.y.
.W
simple shear flow past a flat plate in an incompressible fluid of small
viscosity .
in the study of high-speed viscous flow past a two-dimensional body it
is usually necessary to consider a curved shock wave emitting from the
nose or leading edge of the body . consequently, there exists an
inviscid rotational flow region between the shock wave and the boundary
layer . such a situation arises, for instance, in the study of the
hypersonic viscous flow past a flat plate . the situation is somewhat
different from prandtl's classical boundary-layer problem . in prandtl's
original problem the inviscid free stream outside the boundary layer is
irrotational while in a hypersonic boundary-layer problem the inviscid
free stream must be considered as rotational . the possible effects of
vorticity have been recently discussed by ferri and libby . in the
present paper, the simple shear flow past a flat plate in a fluid of small
viscosity is investigated . it can be shown that this problem can again
be treated by the boundary-layer approximation, the only novel feature
being that the free stream has a constant vorticity . the discussion
here is restricted to two-dimensional incompressible steady flow .
.I 3
.T
the boundary layer in simple shear flow past a flat plate .
.A
m. b. glauert
.B
department of mathematics, university of manchester, manchester,
england
.W
the boundary layer in simple shear flow past a flat plate .
the boundary-layer equations are presented for steady
incompressible flow with no pressure gradient .
.I 4
.T
approximate solutions of the incompressible laminar
boundary layer equations for a plate in shear flow .
.A
yen,k.t.
.B
j. ae. scs. 22, 1955, 728.
.W
approximate solutions of the incompressible laminar
boundary layer equations for a plate in shear flow .
the two-dimensional steady boundary-layer
problem for a flat plate in a
shear flow of incompressible fluid is considered .
solutions for the boundary-
layer thickness, skin friction, and the velocity
distribution in the boundary
layer are obtained by the karman-pohlhausen
technique . comparison with
the boundary layer of a uniform flow has also
been made to show the effect of
vorticity .
. . .

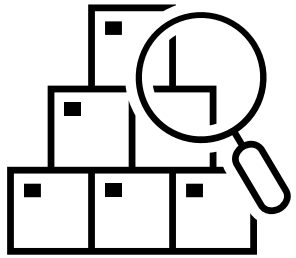
Introduction

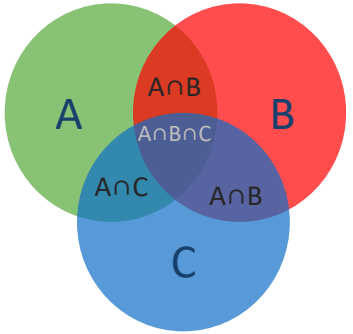
END

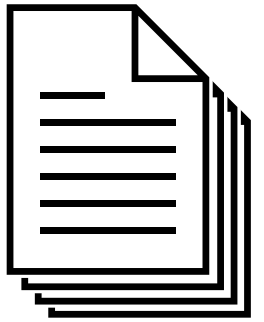
Boolean model

Boolean model

- Boolean retrieval
- Entities
- Inverted index creation
- Operations
- The IR System

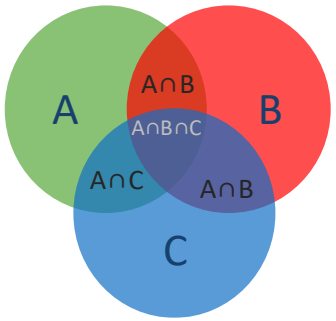






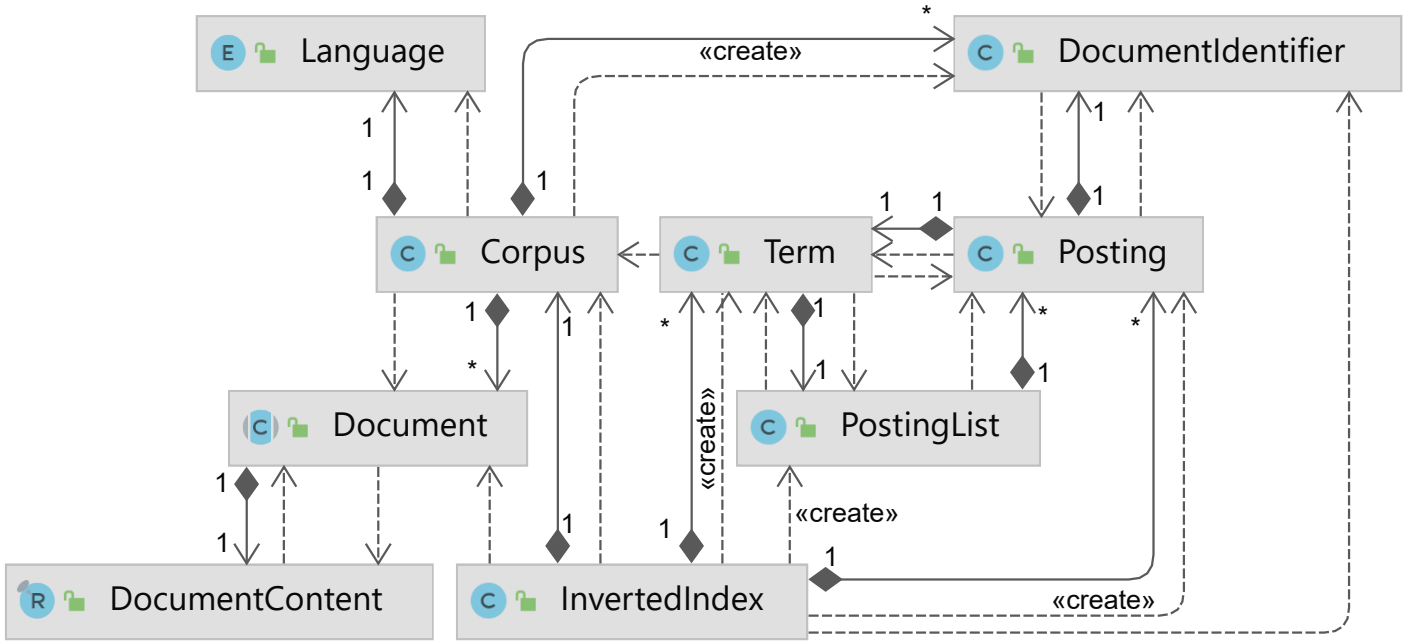
Boolean retrieval

- IR model is based on **boolean logic** and **set theory**
- Document and queries are conceived as **sets of terms**
- **Retrieval** is based on whether documents contain query terms or not
  - Only exact matching
    - a document is either relevant or not
  - Query expressed as a **boolean formula**
    - e.g., `( cat OR dog ) AND (NOT horse)`
- *Extended Boolean model* allows more powerful queries
  - **Phrase** queries (e.g., `"cat is on the table"`)
  - **Wildcard** queries (e.g., `"tab*e"`)
- **Ranking** of results according to their relevance is possible



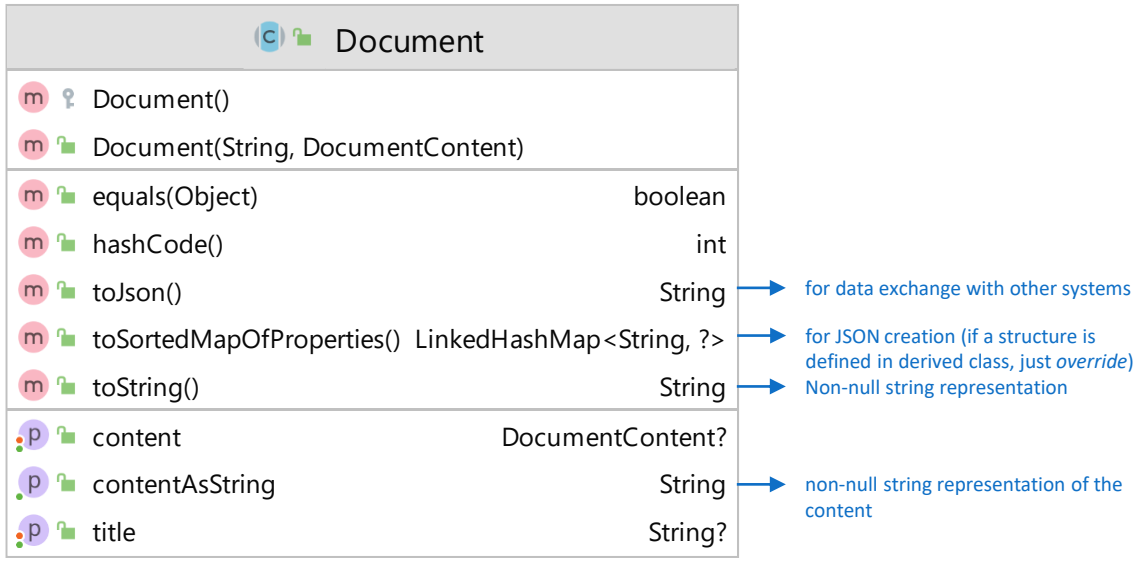
Entities

- Document
- DocumentIdentifier
- Corpus
- Posting
- PostingList
- Term
- InvertedIndex



Entities

- Document
  - Individual unit on which the IR system is built
  - Textual (in this system)
  - The class can be extended
    - The system defines an **abstract** class
  - Can have a structure
    - Title, author, abstract, …
    - To be implemented in derived classes

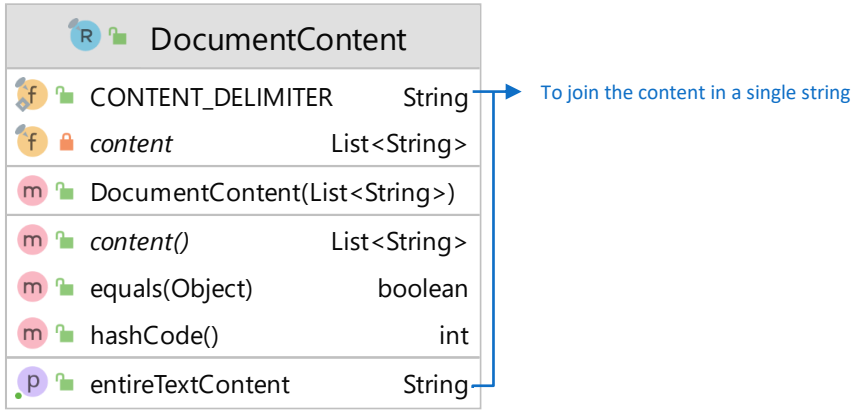


'?' near a field type means that it can be null

Entities

- Document
  - Contains a **DocumentContent** instance

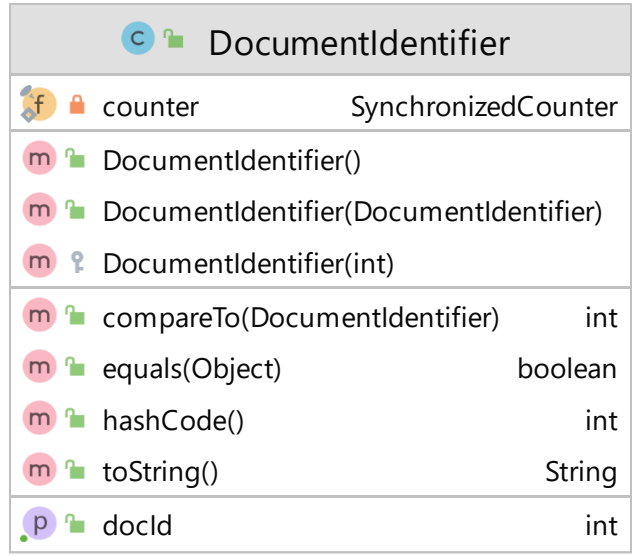
```
public record DocumentContent(@NotNull List<String> content) implements Serializable {

   public static final String CONTENT_DELIMITER = System.lineSeparator();

   public DocumentContent(@NotNull List<@NotNull String> content) {
      this.content = Objects.requireNonNull(content);
   }

   @NotNull
   public String getEntireTextContent() {
      return content.stream().sequential().collect(Collectors.joining(CONTENT_DELIMITER));
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      DocumentContent that = (DocumentContent) o;
      return Objects.equals(content, that.content);
   }

   @Override
   public int hashCode() {
      return content.hashCode();
   }
}
```



Entities

- DocumentIdentifier
  - Unique number used by the Corpus to identify each document
  - *Document*s do *not* need to know their own *DocumentIdentifier*
  - ***docId*** is the actual integer identifier
  - static (class variable) counter is used
    - avoid duplicates of *docId*s
    - Use a ***SynchronizedCounter***
      - Thread-safe

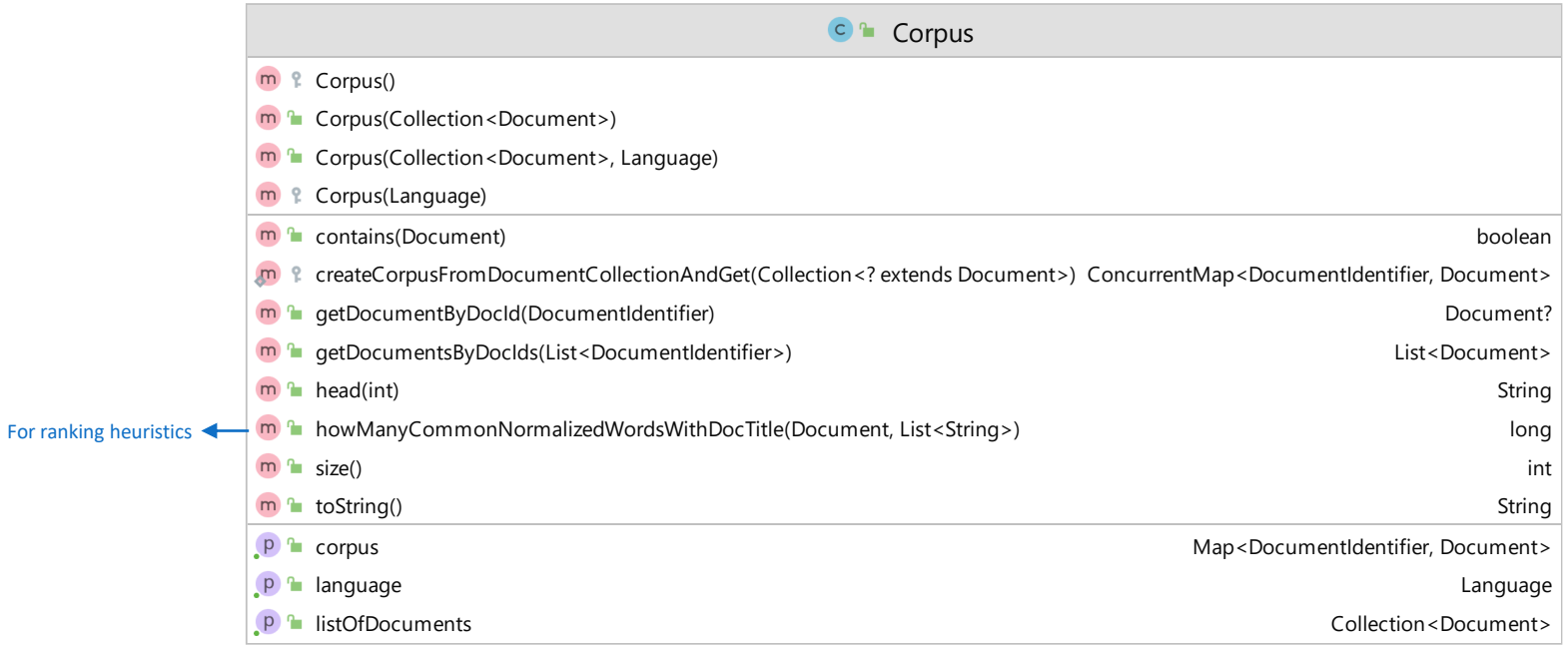
        ➔ Multithreaded approach to create the Corpus can be used



Entities

- Corpus
  - Group of *Document*s on which the search will be performed
  - Static
    - This project does not handle dynamic collections
    - If the collection changes

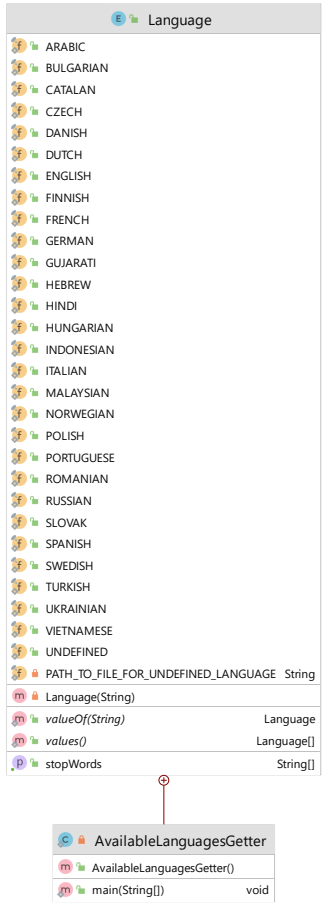
      ➔ Re-indexing needed



Entities

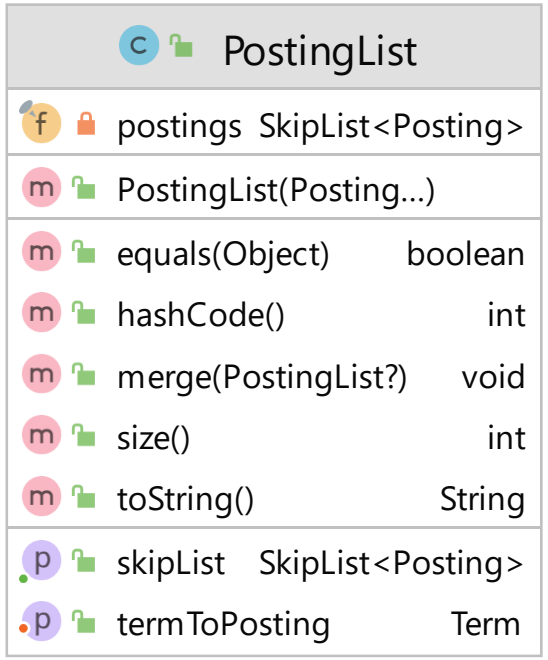
- Corpus
  - Has its own **Language**
    - It is the language of the documents
    - It is assumed to be the same for all the documents in the collection
  - Can be *UNDEFINED*
  - **Stemming** and **stop-words** are **language specific**

```
public enum Language {
    ARABIC("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/arabic.txt"),
    BULGARIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/bulgarian.txt"),
    CATALAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/catalan.txt"),
    CZECH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/czech.txt"),
    DANISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/danish.txt"),
    DUTCH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/dutch.txt"),
    ENGLISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/english.txt"),
    FINNISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/finnish.txt"),
    FRENCH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/french.txt"),
    GERMAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/german.txt"),
    GUJARATI("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/gujarati.txt"),
    HEBREW("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/hebrew.txt"),
    HINDI("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/hindi.txt"),
    HUNGARIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/hungarian.txt"),
    INDONESIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/indonesian.txt"),
    ITALIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/italian.txt"),
    MALAYSIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/malaysian.txt"),
    NORWEGIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/norwegian.txt"),
    POLISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/polish.txt"),
    PORTUGUESE("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/portuguese.txt"),
    ROMANIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/romanian.txt"),
    RUSSIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/russian.txt"),
    SLOVAK("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/slovak.txt"),
    SPANISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/spanish.txt"),
    SWEDISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/swedish.txt"),
    TURKISH("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/turkish.txt"),
    UKRAINIAN("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/ukrainian.txt"),
    VIETNAMESE("/it/units/informationretrieval/ir_boolean_model/stop_words/datasets/vietnamese.txt"),
    UNDEFINED("");

    private static final String PATH_TO_FILE_FOR_UNDEFINED_LANGUAGE = "";
    private final String[] stopWords;

    Language(@NotNull String pathToStopWordsDataset) {
        String[] tmpStopWords;  // tmp variable used to ensure the field will be initialized, also if any error occurs.
        tmpStopWords = pathToStopWordsDataset.equals(PATH_TO_FILE_FOR_UNDEFINED_LANGUAGE)
                ? new String[0] : Utility.readAllLines(getClass().getResourceAsStream(pathToStopWordsDataset))
                .stream().map(stopWord -> Utility.removeInvalidCharsAndToLowerCase(stopWord, false)).toArray(String[]::new);
        this.stopWords = tmpStopWords;
    }

    public String[] getStopWords() {
        return stopWords;
    }
}
```



Entities

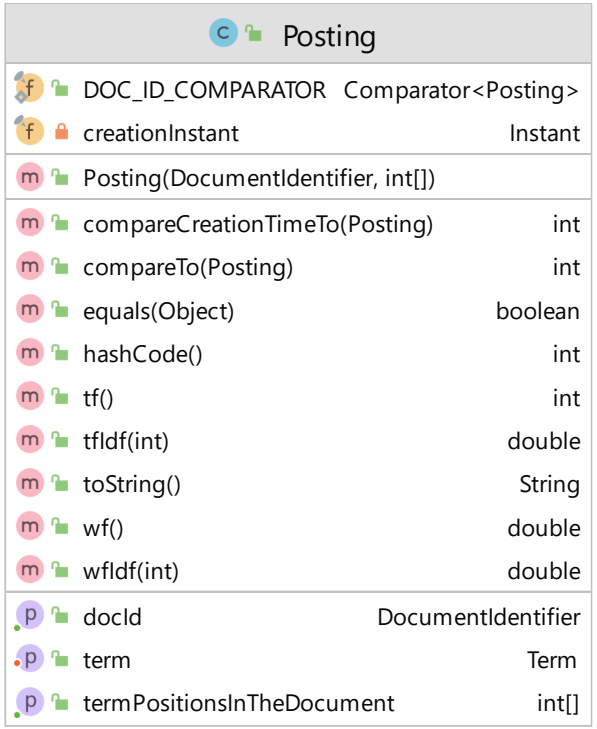
- PostingList
  - The list of *docId*s associated to a term
  - is a list of *posting*s, actually
  - uses a *SkipList* as data-structure



Entities

- Posting
  - An element of the *PostingList*
  - More than a *docId*
    - *docId* is simply a number
    - *posting* is associated with both the document and the term
      - for *tf(), tfIdf(corpusSize), wf(), wfIdf(corpusSize)*
      - Saves the positions of the term to which the posting refers

        ➔ **positional index**

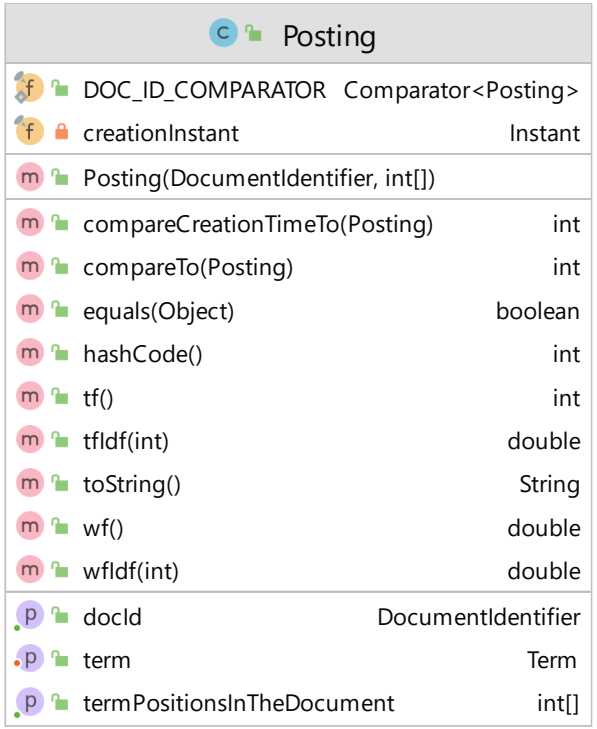


Entities

- Posting
  - Defines a “docId comparator”
    - Are two postings having the same *docId* equals?
      - **NO for storing** in skip lists
        - Skip lists contain only distinct sorted elements
        - A document may contain more than one distinct term

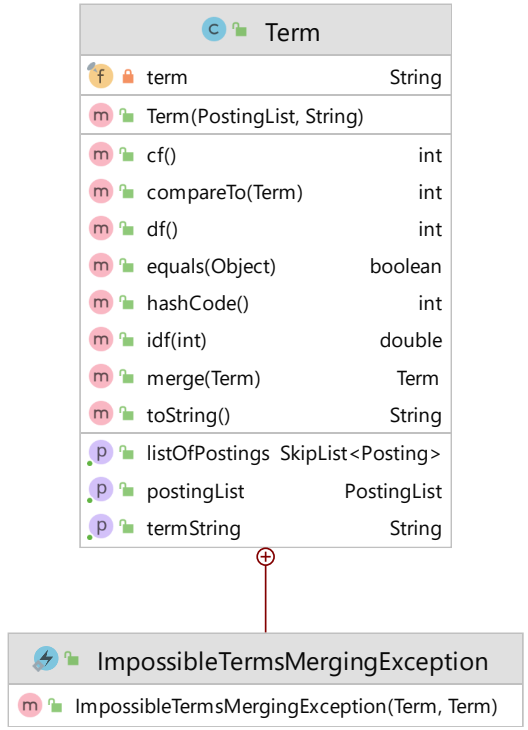
          ➔ more than one posting with same *docId* (but different term) may be present
        - Natural comparator (defined by *compareTo(..)* method) considers postings with different term positions as non-equal
        - *equals(..)* method considers postings with different term positions non-equal, too
      - **YES for intersection** operators
        - Suppose two posting lists have both a posting with the same *docId*, but referring to different terms
        - Their intersection must contain a posting with that *docId* (regardless of positions)

          ➔ use *DOC_ID_COMPARATOR* to consider *docId* equality only



Entities

- Term
  - Part of a *Document*
    - A (*normalized)* word (in this project) + attributes
  - throws *ImpossibleTermsMergingException*
    - if trying to merge term instances which are not equals each other.



Entities

- InvertedIndex
  - the main data-structure
  - saves, for each *Term*, the list of documents containing that *Term*
  - allows to avoid scanning the entire *Corpus* when a query must be answered
  - index update is not implemented
    - If the *Corpus* changes, it must be re-indexed
  - fields of class *InvertedIndex* are explained in the following slides

InvertedIndex	
WILDCARD	String
ESCAPED_WILDCARD_FOR_REGEX	String
END_OF_WORD	String
invertedIndex	ConcurrentMap<String, Term>
postings	SkipList<Posting>
phoneticHashes	ConcurrentMap<String, List<Term>>
permutermIndex	PatriciaTrie<Set<String>>
permutermIndexWithoutEndOfWord	PatriciaTrie<Set<String>>
unstemmedTerms	SynchronizedSet<String>
stemmer	AvailableStemmer
InvertedIndex(Corpus)	
cf(String)	int
createPermutermIndexAndGet(String)	PatriciaTrie<Set<String>>
getDictionaryTermsContainingPrefix(String, boolean)	Collection<String>
getDictionaryTermsFromSoundexCorrectionOf(String)	Collection<String>
getEntrySetOfTokensAndCorrespondingTermsFromADocument(Entry<DocumentIdentifier, Document>)	Set<Entry<String, Term>>
getListOfPostingsForToken(String)	SkipList<Posting>
indexCorpusAndGet(Corpus, AtomicLong)	ConcurrentMap<String, Term>
printIndexingProgressAndGetTheRunnableToInterruptPrinting(int, AtomicLong)	Runnable
readObject(ObjectInputStream)	void
allPostings	SkipList<Posting>
copyOfPermutermIndex	PatriciaTrie<Set<String>>
corpus	Corpus
dictionary	List<String>
phoneticHashesOfDictionary	ConcurrentMap<String, List<Term>>
unstemmedDictionary	List<String>

Entities

- InvertedIndex
  - *WILDCARD*
    - For wildcard queries
  - invertedIndex
    - The main data-structure
  - postings
    - For NOT queries
    - If no optimizations can be done
  - phoneticHashes
    - For **phonetic correction**

InvertedIndex	
WILDCARD	String
ESCAPED_WILDCARD_FOR_REGEX	String
END_OF_WORD	String
invertedIndex	ConcurrentMap<String, Term>
postings	SkipList<Posting>
phoneticHashes	ConcurrentMap<String, List<Term>>
permutermIndex	PatriciaTrie<Set<String>>
permutermIndexWithoutEndOfWord	PatriciaTrie<Set<String>>
unstemmedTerms	SynchronizedSet<String>
stemmer	AvailableStemmer
InvertedIndex(Corpus)	
cf(String)	int
createPermutermIndexAndGet(String)	PatriciaTrie<Set<String>>
getDictionaryTermsContainingPrefix(String, boolean)	Collection<String>
getDictionaryTermsFromSoundexCorrectionOf(String)	Collection<String>
getEntrySetOfTokensAndCorrespondingTermsFromADocument(Entry<DocumentIdentifier, Document>)	Set<Entry<String, Term>>
getListOfPostingsForToken(String)	SkipList<Posting>
indexCorpusAndGet(Corpus, AtomicLong)	ConcurrentMap<String, Term>
printIndexingProgressAndGetTheRunnableToInterruptPrinting(int, AtomicLong)	Runnable
readObject(ObjectInputStream)	void
allPostings	SkipList<Posting>
copyOfPermutermIndex	PatriciaTrie<Set<String>>
corpus	Corpus
dictionary	List<String>
phoneticHashesOfDictionary	ConcurrentMap<String, List<Term>>
unstemmedDictionary	List<String>

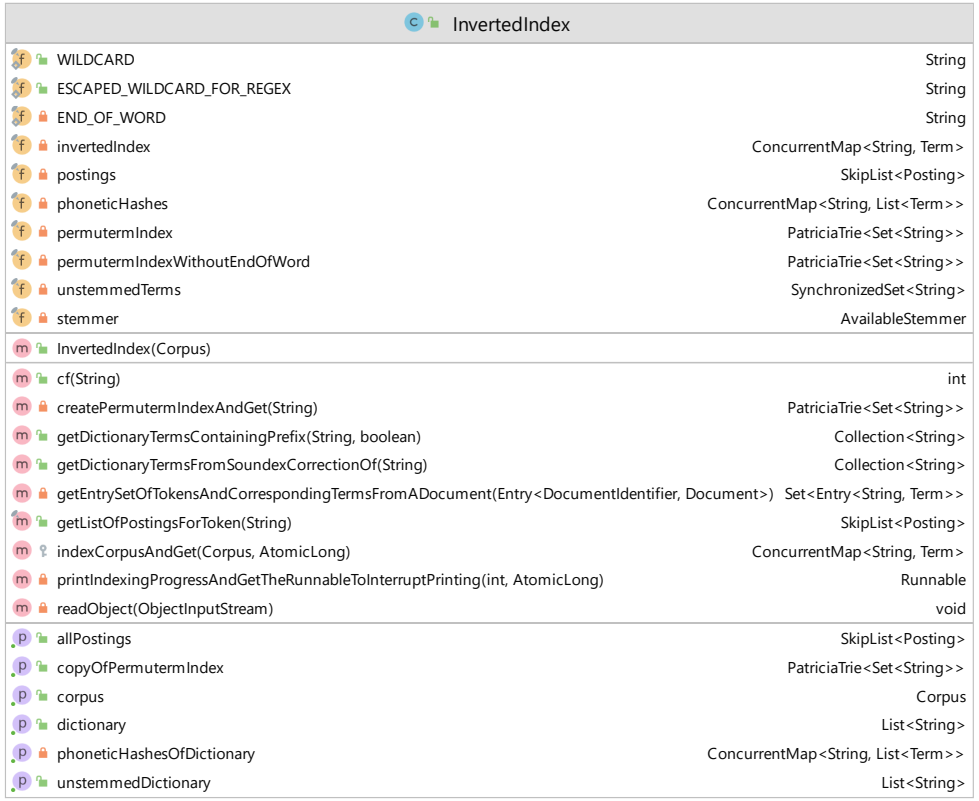
Entities

- InvertedIndex
  - permutermIndex
    - *PatriciaTrie*
    - For **wildcard queries**
    - Uses the *END_OF_WORD* symbol
  - permutermIndexWithoutEndOfWord
    - *PatriciaTrie*
    - For **spelling correction**
    - Like *permutermIndex* but does not use the *END_OF_WORD* symbol
      - \+ exploits the efficiency of *PatriciaTrie* data-structure
      - \- wastes memory (*keys* differ only for the absence of *END_OF_WORD*)
      - \- update problems

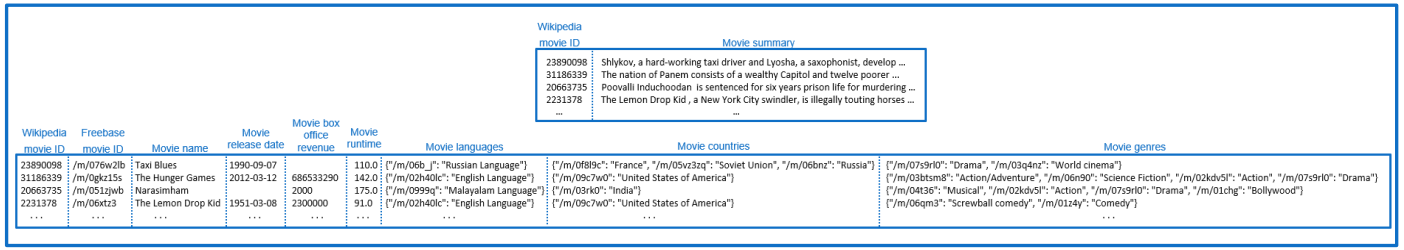
InvertedIndex	
WILDCARD	String
ESCAPED_WILDCARD_FOR_REGEX	String
END_OF_WORD	String
invertedIndex	ConcurrentMap<String, Term>
postings	SkipList<Posting>
phoneticHashes	ConcurrentMap<String, List<Term>>
permutermIndex	PatriciaTrie<Set<String>>
permutermIndexWithoutEndOfWord	PatriciaTrie<Set<String>>
unstemmedTerms	SynchronizedSet<String>
stemmer	AvailableStemmer
InvertedIndex(Corpus)	
cf(String)	int
createPermutermIndexAndGet(String)	PatriciaTrie<Set<String>>
getDictionaryTermsContainingPrefix(String, boolean)	Collection<String>
getDictionaryTermsFromSoundexCorrectionOf(String)	Collection<String>
getEntrySetOfTokensAndCorrespondingTermsFromADocument(Entry<DocumentIdentifier, Document>)	Set<Entry<String, Term>>
getListOfPostingsForToken(String)	SkipList<Posting>
indexCorpusAndGet(Corpus, AtomicLong)	ConcurrentMap<String, Term>
printIndexingProgressAndGetTheRunnableToInterruptPrinting(int, AtomicLong)	Runnable
readObject(ObjectInputStream)	void
allPostings	SkipList<Posting>
copyOfPermutermIndex	PatriciaTrie<Set<String>>
corpus	Corpus
dictionary	List<String>
phoneticHashesOfDictionary	ConcurrentMap<String, List<Term>>
unstemmedDictionary	List<String>

Entities

- InvertedIndex
  - stemmer
    - (eventually) used for **stemming**
  - unstemmedTerms
    - Used to create the permuterm index
      - \+ Improve the precision
        - Users that insert a wildcard query do not know how terms are stemmed
      - \- wastes memory
        - redundant with the dictionary (i.e., with the *keyset* of *invertedIndex*)



Inverted index creation



Wikipedia movie ID	Movie summary
23890098	Shlykov, a hard-working taxi driver and Lyosha, a saxophonist, develop ...
31186339	The nation of Panem consists of a wealthy Capitol and twelve poorer ...
20663735	Poovalli Induchoodan is sentenced for six years prison life for murdering ...
2231378	The Lemon Drop Kid , a New York City swindler, is illegally touting horses ...
...	...

Wikipedia movie ID	Freebase movie ID	Movie name	Movie release date	Movie box office revenue	Movie runtime	Movie languages	Movie countries	Movie genres
23890098	/m/076w2lb	Taxi Blues	1990-09-07		110.0	{"/m/06b_j": "Russian Language"}	{"/m/0f8l9c": "France", "/m/05vz3zq": "Soviet Union", "/m/06bnz": "Russia"}	{"/m/07s9rl0": "Drama", "/m/03q4nz": "World cinema"}
31186339	/m/0gkz15s	The Hunger Games	2012-03-12	686533290	142.0	{"/m/02h40lc": "English Language"}	{"/m/09c7w0": "United States of America"}	{"/m/03btsm8": "Action/Adventure", "/m/06n90": "Science Fiction", "/m/02kdv5l": "Action", "/m/07s9rl0": "Drama"}
20663735	/m/051zjwb	Narasimham		2000	175.0	{"/m/0999q": "Malayalam Language"}	{"/m/03rk0": "India"}	{"/m/04t36": "Musical", "/m/02kdv5l": "Action", "/m/07s9rl0": "Drama", "/m/01chg": "Bollywood"}
2231378	/m/06xtz3	The Lemon Drop Kid	1951-03-08	2300000	91.0	{"/m/02h40lc": "English Language"}	{"/m/09c7w0": "United States of America"}	{"/m/06qm3": "Screwball comedy", "/m/01z4y": "Comedy"}
...	...	...	...	...	...	...	...	...

Figure Input collection (movie corpus) [continues...]

- For each document
  1. Preprocess
     a. Normalize
     b. Stop-words removal
     c. Stemming
  2. Tokenize and get term positions
  3. Create a posting for each term
     - The posting is hence associated both with a term and a document
- Create posting lists
  - *SkipList* of *Postings* referring to the same *Term* in different *Documents*
  - Sorted
- Return all the distinct *Terms*
  - It is the dictionary of the IR system
  - Each *Term* has its own *PostingList* (as attribute)
- The *invertedIndex* is the resulting *Map*
  - *key*: the token for the *Term* (saved as *String)*
  - *value:* the *Term*

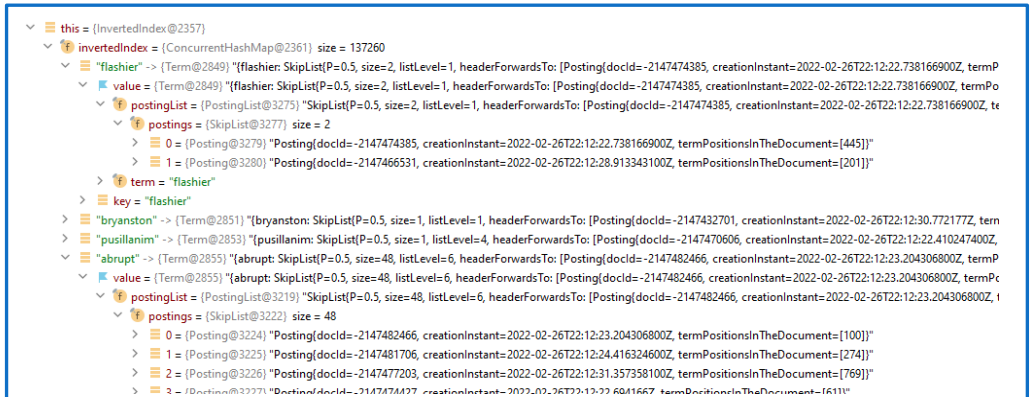
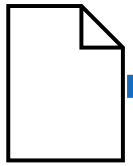


Figure Result of indexing (debug-view) of the *Movie Corpus* for the field *invertedIndex* [continues...]

Figure Result of indexing (debug-view) of the *Movie Corpus* for the field *corpus* [continues...]

Inverted index creation

- Document tokenization



Document + Language

TOKENIZATION

Split a document into words and *preprocess* them to obtain the *tokens*

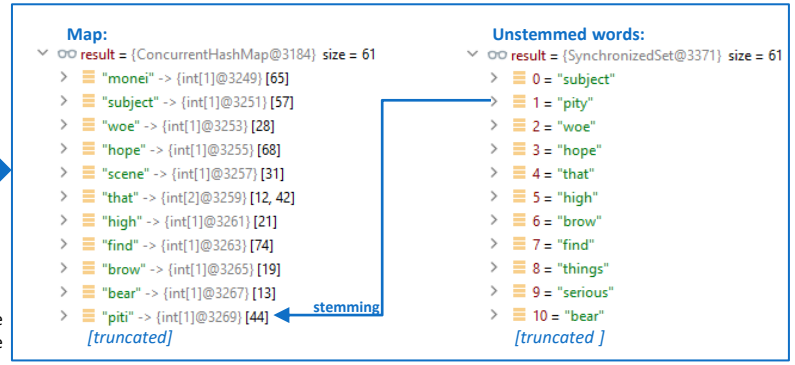
Map:
- key: a normalized token
- value: positions of the token in the document

No duplicated keys are present

If the same token appears more than once, the array of positions (a value of the map) saves its positions in the document

```
String textOfDocument = """
      I come no more to make you laugh: things now,
      That bear a weighty and a serious brow,
      Sad, high, and working, full of state and woe,
      Such noble scenes as draw the eye to flow,
      We now present. Those that can pity, here
      May, if they think it well, let fall a tear;
      The subject will deserve it. Such as give
      Their money out of hope they may believe,
      May here find truth too.
      """;
Document document = new FakeDocument (
   "Henry VIII", textOfDocument);
SynchronizedSet<String> unstemmedWords =
   new SynchronizedSet<>();
```

Utility.*tokenizeAndGetMapWithPositionsInDocument*(document, Language. *ENGLISH*, unstemmedWords)



Positions are computed taking the entire text of the document (including the title): e.g., token «that» would be at position 10 (first position is zero), but the title («Henry VIII») is two tokens long, hence «that» is at position 12.

Inverted index creation

- Document tokenization

```
@NotNull
public static Map<String, int[]> tokenizeAndGetMapWithPositionsInDocument(
      @NotNull final Document document, @NotNull Language language, @NotNull SynchronizedSet<String> unstemmedWords) {

   String[] tokensEventuallyDuplicatesSortedByPositionInDocument = tokenize(document, language, unstemmedWords);

   Map<String, Set<Integer>> mapTokenToPositionsInDoc = IntStream
         .range(0, tokensEventuallyDuplicatesSortedByPositionInDocument.length)
         .mapToObj(i -> new AbstractMap.SimpleEntry<>(
               tokensEventuallyDuplicatesSortedByPositionInDocument[i], i))
         .collect(Collectors.groupingByConcurrent(
               Map.Entry::getKey,
               Collectors.mapping(Map.Entry::getValue, toSet())));

   return mapTokenToPositionsInDoc.entrySet()
         .stream()
         .collect(Collectors.toConcurrentMap(
               Map.Entry::getKey,
               entry -> entry.getValue().stream().sorted().mapToInt(i -> i).toArray(),
               (a, b) -> {
                  throw new IllegalStateException("No duplicates should be present, but were.");
               },
               ConcurrentHashMap::new));
}
```

```
@NotNull
public static String[] tokenize(
      @NotNull Document document, @NotNull Language language, @NotNull SynchronizedSet<String> unstemmedWords) {

   return Arrays.stream(
               split(/*title is included in the content*/document.getContent().getEntireTextContent()
                     .replaceAll(REGEX_PUNCTUATION, " ")))
         .filter(text -> !text.isBlank())
         .map(token -> Utility.normalize(token, false, language, unstemmedWords))
         .filter(Objects::nonNull)
         .toArray(String[]::new);
}


public static String[] split(@NotNull String toSplit) {
   return toSplit.split(REGEX_MULTIPLE_SPACES);
}
```

```
REGEX_MULTIPLE_SPACES = "\s+"
```

Inverted index creation

- Document preprocessing
  - Normalization
  - Stop-words removal
  - Stemming


```
SynchronizedSet<String> unstemmedWord =
    new SynchronizedSet<>();
String token = "pity";
boolean fromQuery = false;
```

```
Utility.preprocess(
    token, fromQuery, Language.ENGLISH, unstemmedWord)
```

```
result = "piti"
unstemmedWord = {"pity"}
```

```
SynchronizedSet<String> unstemmedWord =
    new SynchronizedSet<>();
String token = "pit*";
boolean fromQuery = true;
```

```
Utility. preprocess(
    token, fromQuery, Language.ENGLISH, unstemmedWord)
```

```
result = "pit*"
unstemmedWord = {"pit*"}
```

- unstemmedWord is a SynchronizedSet<String> because the set is needed for further steps of preprocessing (see *tokenization)*
- the flag fromQuery is needed because special characters (like the wildcard symbol '*') must be kept during querying but removed during indexing
- the language is needed for language-specific preprocessing (stemming/stop word removal)

```
@Nullable
public static String preprocess(
      @NotNull String token, boolean fromQuery,
      @NotNull Language language, @NotNull SynchronizedSet<String> unstemmedWords) {

   String toReturn = normalize(token, fromQuery);

   Stemmer stemmer = getStemmer();

   // Stop-words handling (when stemming is not performed)
   if (stemmer == null && shouldExcludeStopWords()/*search in env properties*/ ) {
      if (isStopWord(toReturn, language, false)) {
         return null;
      }
   }

   // Stemming
   unstemmedWords.add(toReturn);  // before stemming, save the un-stemmed word
   if (stemmer != null && !toReturn.contains(InvertedIndex.WILDCARD)) {
      toReturn = stemmer.stem(toReturn, language);
   }

   // Stop-words handling (when stemming is performed)
   if (shouldExcludeStopWords()) {
      if (isStopWord(toReturn, language, true)) {
         return null;
      }
   }

   return toReturn.isBlank() ? null : toReturn;
}
```

Inverted index creation

- Document preprocessing
  - Normalization
  - Stop-words removal
  - Stemming

```
REGEX__NOT__VALID_CHARACTERS_WHEN_QUERYING = "[^a-zA-Z0-9\s\*]"
REGEX__NOT__VALID_CHARACTERS_WHEN_INDEXING = "[^a-zA-Z0-9\s]"
REGEX_MULTIPLE_SPACES                      = "\s+"
```

```
@NotNull
private static String normalize(@NotNull String token, boolean fromQuery) {
  return removeInvalidCharsAndToLowerCase(token, fromQuery);
}
```

```
@NotNull
public static String removeInvalidCharsAndToLowerCase(@NotNull String token, boolean fromQuery) {
  return token
      .replaceAll(
          fromQuery
              ? REGEX__NOT__VALID_CHARACTERS_WHEN_QUERYING
              : REGEX__NOT__VALID_CHARACTERS_WHEN_INDEXING,
          "")
      .replaceAll(REGEX_MULTIPLE_SPACES, " ")
      .toLowerCase(Locale.ROOT)
      .strip();
}
```

```
@Nullable
public static String preprocess(
    @NotNull String token, boolean fromQuery,
    @NotNull Language language, @NotNull SynchronizedSet<String> unstemmedWords) {

  String toReturn = normalize(token, fromQuery);

  Stemmer stemmer = getStemmer();

  // Stop-words handling (when stemming is not performed)
  if (stemmer == null && shouldExcludeStopWords()/*search in env properties*/ ) {
    if (isStopWord(toReturn, language, false)) {
      return null;
    }
  }

  // Stemming
  unstemmedWords.add(toReturn);  // before stemming, save the un-stemmed word
  if (stemmer != null && !toReturn.contains(InvertedIndex.WILDCARD)) {
    toReturn = stemmer.stem(toReturn, language);
  }

  // Stop-words handling (when stemming is performed)
  if (shouldExcludeStopWords()) {
    if (isStopWord(toReturn, language, true)) {
      return null;
    }
  }

  return toReturn.isBlank() ? null : toReturn;
}
```

Inverted index creation

- Document preprocessing
  - Normalization
  - Stop-words removal
  - Stemming

```
public static boolean isStopWord(String word, @NotNull Language language, boolean stemming) {
  return Arrays.asList(stemming
        ? Stemmer.getStemmedStopWords.apply(
        getStemmer() == null
              ? Stemmer.AvailableStemmer.NO_STEMMING
              : Stemmer.AvailableStemmer.fromStemmer(getStemmer()), language)
        : language.getStopWords()).contains(word);
}
```

- word is the token currently being preprocessed
- Stemmer.*getStemmedStopWords* is the (static) BiFunction<AvailableStemmer, Language, String[]> used for getting the stemmed stop words for the specified language with the currently used stemmer
- *getStemmer*() is the static method used for caching and fast retrieval of the currently used Stemmer
- language.getStopWords() returns the un-stemmed stop-words for the specified language

```
@Nullable
public static String preprocess(
      @NotNull String token, boolean fromQuery,
      @NotNull Language language, @NotNull SynchronizedSet<String> unstemmedWords) {

  String toReturn = normalize(token, fromQuery);

  Stemmer stemmer = getStemmer();

  // Stop-words handling (when stemming is not performed)
  if (stemmer == null && shouldExcludeStopWords()/*search in env properties*/ ) {
    if (isStopWord(toReturn, language, false)) {
      return null;
    }
  }

  // Stemming
  unstemmedWords.add(toReturn);  // before stemming, save the un-stemmed word
  if (stemmer != null && !toReturn.contains(InvertedIndex.WILDCARD)) {
    toReturn = stemmer.stem(toReturn, language);
  }

  // Stop-words handling (when stemming is performed)
  if (shouldExcludeStopWords()) {
    if (isStopWord(toReturn, language, true)) {
      return null;
    }
  }

  return toReturn.isBlank() ? null : toReturn;
}
```

Inverted index creation

- Document preprocessing
  - Normalization
  - Stop-words removal
  - Stemming

```
@Nullable  //  null if the IR system is currently configured for not performing stemming
public static Stemmer getStemmer() {
   if (!cachedStemmedInitialized) {       // caching for fast retrieval of the currently used stemmer (frequently used)
      cachedStemmed = null;
      try {
         String stemmerName = AppProperties.getInstance().get("app.stemmer"); // search the stemmer in the app props
         cachedStemmed = Stemmer.getStemmer(Stemmer.AvailableStemmer.valueOf_(stemmerName));
      } catch (IOException e) {
         System.err.println("Error while reading app properties. Stemming will not be performed.");
         e.printStackTrace();
      } catch (IllegalArgumentException e) {
         System.err.println("Stemmer not available while reading app properties. Stemming will not be performed.");
         e.printStackTrace();
      }
      cachedStemmedInitialized = true;
   }
   return cachedStemmed;
}
```

```
@Nullable
public static String preprocess(
      @NotNull String token, boolean fromQuery,
      @NotNull Language language, @NotNull SynchronizedSet<String> unstemmedWords) {

   String toReturn = normalize(token, fromQuery);

   Stemmer stemmer = getStemmer();

   // Stop-words handling (when stemming is not performed)
   if (stemmer == null && shouldExcludeStopWords()/*search in env properties*/ ) {
      if (isStopWord(toReturn, language, false)) {
         return null;
      }
   }

   // Stemming
   unstemmedWords.add(toReturn);   // before stemming, save the un-stemmed word
   if (stemmer != null && !toReturn.contains(InvertedIndex.WILDCARD)) {
      toReturn = stemmer.stem(toReturn, language);
   }

   // Stop-words handling (when stemming is performed)
   if (shouldExcludeStopWords()) {
      if (isStopWord(toReturn, language, true)) {
         return null;
      }
   }

   return toReturn.isBlank() ? null : toReturn;
}
```

Inverted index creation

- Stemming
  - language-specific
  - **Porter stemmer** is implemented
    - Implementation adapted from https://tartarus.org/martin/PorterStemmer/java.txt
  - *Stemmer* is an interface in this project
    - Additional stemmers can be implemented
  - What stemmer (and if) to use depends on the environment properties
  - Same stemmer is used both for indexing and querying
    - In queries, the wildcard symbol is not removed

```
public interface Stemmer {

  BiFunction<@NotNull AvailableStemmer, @NotNull Language, @NotNull String[]> getStemmedStopWords = new BiFunction<>() {

    @NotNull static final ConcurrentMap<@NotNull AvailableStemmer, @NotNull ConcurrentMap<@NotNull Language, @NotNull String[]>>
        stemmedStopWords =      // caches stemmed stop words, in-place initialization, but also lazy initialization could be done
        Arrays.stream(AvailableStemmer.values())
            .map(aStemmer -> new Pair<>(aStemmer, Arrays.stream(Language.values())
                .map(language -> new Pair<>(language, Arrays.stream(language.getStopWords())
                    .map(stopWord -> getStemmer(aStemmer).stem(stopWord, language))
                    .toArray(String[]::new)))
                .collect(Collectors.toConcurrentMap(Map.Entry::getKey, Map.Entry::getValue))))
            .collect(Collectors.toConcurrentMap(Map.Entry::getKey, Map.Entry::getValue));

    @Override
    @NotNull
    public String[] apply(@NotNull AvailableStemmer stemmer, @NotNull Language language) {
      return stemmedStopWords.get(stemmer).get(language);
    }
  };

  static Stemmer getStemmer(@NotNull AvailableStemmer stemmer) {
    return switch (stemmer) {
      case NO_STEMMING -> new NoStemming();
      case PORTER -> new PorterStemmer();
    };
  }

  String stem(@NotNull String input, @NotNull Language language);

  class NoStemming implements Stemmer {
    @Override
    public String stem(@NotNull String input, @NotNull Language ignored) {
      return input;
    }
  }
```

[continues in the next slide ...]

Inverted index creation

- Stemming
  - language-specific
  - **Porter stemmer** is implemented
    - Implementation adapted from https://tartarus.org/martin/PorterStemmer/java.txt
  - *Stemmer* is an interface in this project
    - Additional stemmers can be implemented
  - What stemmer (and if) to use depends on the environment properties
  - Same stemmer is used both for indexing and querying
    - In queries, the wildcard symbol is not removed

[… continues from the previous slide]

```
enum AvailableStemmer {
  NO_STEMMING,
  PORTER;
  public static AvailableStemmer fromStemmer(@Nullable Stemmer stemmer) {
    if (stemmer == null || stemmer instanceof NoStemming) {
      return NO_STEMMING;
    } else if (stemmer instanceof PorterStemmer) {
      return PORTER;
    } else {
      throw new IllegalArgumentException("Unknown stemmer " + stemmer);
    }
  }

  public static AvailableStemmer valueOf_(String valueAsString) {
    // "Override" {@link Enum#valueOf(Class, String)}.

    if (valueAsString == null || valueAsString.equalsIgnoreCase("null")) {
      return NO_STEMMING;
    }
    var tmp = Arrays.stream(AvailableStemmer.values())
        .filter(enumVal -> enumVal.name().equalsIgnoreCase(valueAsString))
        .toList();
    if (tmp.size() == 1) {
      return tmp.get(0);
    } else {
      System.err.println("Invalid enum \"" + valueAsString + "\". Valid possibilities are: "
          + Arrays.toString(AvailableStemmer.values()) + ". "
          + NO_STEMMING + " returned.");
      return NO_STEMMING;
    }
  }
}
```

Inverted index creation

- Soundex algorithm
  - For phonetic correction
  - creates the **phonetic hashes**

Word	Phonetic hash
Robert	r163
Rupert	r163
Rubin	r150
Robin	r150
Honeyman	h555
Onimen	o555

```
public abstract class Soundex {

   public static String getPhoneticHash(@NotNull final String inputWord) {
      var inputWord_ = inputWord.strip().toLowerCase();
      if (!inputWord_.isBlank()) {
         if (inputWord_.length() == 1) {
            return inputWord_;
         } else {
            return padWith0IfNeededAndGetFirst4LettersOf(
                  inputWord_.charAt(0) +
                        removeAllOccurrencesOf0ButFirstLetter(
                              replaceLettersWithSoundexDigits(
                                    keepOnlyTheFirstLetterIfTwoOrMoreLettersWithTheSameSoundexDigitAreAdjacentOrSeparatedByHorWorY(inputWord_)))
                              .substring(1));
         }
      } else {
         return "";
      }
   }
```

Inverted index creation

- Soundex algorithm
  - For phonetic correction
  - Uses the **phonetic hash**

```
private static String keepOnlyTheFirstLetterIfTwoOrMoreLettersWithTheSameSoundexDigitAreAdjacentOrSeparatedByHorWorY(
    String inputString) {

    char[] letters = inputString.toCharArray();
    char[] digits = replaceLettersWithSoundexDigits(inputString).toCharArray();
    StringBuilder output = new StringBuilder().append(letters[0]);
    int[] magicSeparationLetters = {'h', 'w', 'y'};
    for (int i = 1; i < digits.length; i++) {
        if (digits[i] != digits[i - 1]) {
            final int finalI = i;
            if (i < 2 /* impossible to have a magic letter in the middle if we are currently examining the 1st or 2nd letter */
                    /* if i>2 check not to be present magic letters in the middle of two letters with the same soundex digit */
                    || IntStream.of(magicSeparationLetters).noneMatch(x -> x == (int) letters[finalI - 1])
                    || digits[i] != digits[i - 2]) {
                output.append(letters[i]);
            }
        }
    }
    return output.toString();
}
```

```
    private static String replaceLettersWithSoundexDigits(String inputString) {
        return inputString
                .replaceAll("[aeiouhwy]", "0")
                .replaceAll("[bfpv]", "1")
                .replaceAll("[cgjkqsxz]", "2")
                .replaceAll("[dt]", "3")
                .replaceAll("[l]", "4")
                .replaceAll("[mn]", "5")
                .replaceAll("[r]", "6");
    }

    private static String removeAllOccurrencesOf0ButFirstLetter(String inputString) {
        return inputString.charAt(0) + inputString.substring(1).replaceAll("0", "");
    }

    private static String padWith0IfNeededAndGetFirst4LettersOf(String inputString) {
        final int TARGET_WORD_LENGTH = 4;
        inputString = inputString.length() > TARGET_WORD_LENGTH ? inputString.substring(0, TARGET_WORD_LENGTH) : inputString;
        return String.format("%-" + TARGET_WORD_LENGTH + "s", inputString).replaceAll("\\s+", "0");
    }

}
```

Inverted index creation

- *InvertedIndex* constructor

```
public InvertedIndex(@NotNull final Corpus corpus) {

    this.corpus = corpus;
    AtomicLong numberOfAlreadyProcessedDocuments = new AtomicLong(0L);

    try {
        invertedIndex = indexCorpusAndGet(corpus, numberOfAlreadyProcessedDocuments);
        postings = SkipList.createNewInstanceFromSortedCollection(
                invertedIndex.entrySet().stream().unordered().parallel()
                        .map(Map.Entry::getValue)
                        .map(Term::getListOfPostings)
                        .flatMap(Collection::stream)
                        .sorted()
                        .toList()/*unmodifiable list*/,
                Posting.DOC_ID_COMPARATOR);
        phoneticHashes = getPhoneticHashesOfDictionary();
        permutermIndex = createPermutermIndexAndGet(END_OF_WORD);
        permutermIndexWithoutEndOfWord = createPermutermIndexAndGet("");// NO end-of-word symbol
    } finally {
        System.out.println("\nIndexing ended");
    }

    Stemmer.AvailableStemmer stemmerTmp;   // defer assignment to final field
    try {
        stemmerTmp = Stemmer.AvailableStemmer.valueOf_(AppProperties.getInstance().get("app.stemmer"));
    } catch (IOException e) {
        stemmerTmp = Stemmer.AvailableStemmer.NO_STEMMING;
        Logger.getLogger(getClass().getCanonicalName())
                .log(Level.SEVERE, "Errors while reading app properties", e);
    }
    this.stemmer = stemmerTmp;

}
```

Inverted index creation

- *InvertedIndex* constructor
  - indexCorpusAndGet

```
@NotNull
protected ConcurrentMap<String, Term> indexCorpusAndGet(
    @NotNull Corpus corpus, @NotNull AtomicLong numberOfAlreadyProcessedDocuments /*another thread can print this value*/) {

  // Avoid documents with null content
  Predicate<Map.Entry<DocumentIdentifier, Document>> documentContentNotNullPredicate =
      entry -> entry != null
          && entry.getKey() != null
          && entry.getValue() != null
          && entry.getValue().getContent() != null;

  return corpus.getCorpus()
      .entrySet()
      .stream().unordered().parallel()
      .peek(ignored -> numberOfAlreadyProcessedDocuments.getAndIncrement()/*to save indexing progress*/)
      .filter(documentContentNotNullPredicate)
      .map(this::getEntrySetOfTokensAndCorrespondingTermsFromADocument)
      .flatMap(Collection::stream /*outputs all entries from all the documents*/)
      .collect(
          Collectors.toConcurrentMap(
              Map.Entry::getKey,
              Map.Entry::getValue,
              Term::merge /* Merge terms with the same token */,
              ConcurrentHashMap::new));
}
```

```
public InvertedIndex(@NotNull final Corpus corpus) {

  this.corpus = corpus;
  AtomicLong numberOfAlreadyProcessedDocuments = new AtomicLong(0L);

  try {
    invertedIndex = indexCorpusAndGet(corpus, numberOfAlreadyProcessedDocuments);
    postings = SkipList.createNewInstanceFromSortedCollection(
        invertedIndex.entrySet().stream().unordered().parallel()
            .map(Map.Entry::getValue)
            .map(Term::getListOfPostings)
            .flatMap(Collection::stream)
            .sorted()
            .toList()/*unmodifiable list*/,
        Posting.DOC_ID_COMPARATOR);
    phoneticHashes = getPhoneticHashesOfDictionary();
    permutermIndex = createPermutermIndexAndGet(END_OF_WORD);
    permutermIndexWithoutEndOfWord = createPermutermIndexAndGet("");// NO end-of-word symbol
  } finally {
    System.out.println("\nIndexing ended");
  }

  Stemmer.AvailableStemmer stemmerTmp;   // defer assignment to final field
  try {
    stemmerTmp = Stemmer.AvailableStemmer.valueOf_(AppProperties.getInstance().get("app.stemmer"));
  } catch (IOException e) {
    stemmerTmp = Stemmer.AvailableStemmer.NO_STEMMING;
    Logger.getLogger(getClass().getCanonicalName())
        .log(Level.SEVERE, "Errors while reading app properties", e);
  }
  this.stemmer = stemmerTmp;

}
```

Inverted index creation

- *InvertedIndex* constructor
  - indexCorpusAndGet

```
@NotNull
protected ConcurrentMap<String, Term> indexCorpusAndGet(
    @NotNull Corpus corpus, @NotNull AtomicLong numberOfAlreadyProcessedDocuments /*another thread can print this value*/) {

  // Avoid documents with null content
  Predicate<Map.Entry<DocumentIdentifier, Document>> documentContentNotNullPredicate =
      entry -> entry != null
          && entry.getKey() != null
          && entry.getValue() != null
          && entry.getValue().getContent() != null;

  return corpus.getCorpus()
      .entrySet()
      .stream().unordered().parallel()
      .peek(ignored -> numberOfAlreadyProcessedDocuments.getAndIncrement()/*to save indexing progress*/)
      .filter(documentContentNotNullPredicate)
      .map(this::getEntrySetOfTokensAndCorrespondingTermsFromADocument)
      .flatMap(Collection::stream /*outputs all entries from all the documents*/)
      .collect(
          Collectors.toConcurrentMap(
              Map.Entry::getKey,
              Map.Entry::getValue,
              Term::merge /* Merge terms with the same token */,
              ConcurrentHashMap::new));
}
```

```
public InvertedIndex(@NotNull final Corpus corpus) {

  this.corpus = corpus;
  AtomicLong numberOfAlreadyProcessedDocuments = new AtomicLong(0L);

  try {
    invertedIndex = indexCorpusAndGet(corpus, numberOfAlreadyProcessedDocuments);
    postings = SkipList.createNewInstanceFromSortedCollection(
```

```
@NotNull
private Set<Map.Entry<String, Term>> getEntrySetOfTokensAndCorrespondingTermsFromADocument(
    @NotNull Map.Entry<@NotNull DocumentIdentifier, @NotNull Document> entryFromCorpusRepresentingOneDocument) {
  DocumentIdentifier docIdThisDocument = entryFromCorpusRepresentingOneDocument.getKey();
  Document document = entryFromCorpusRepresentingOneDocument.getValue();
  Map<String, int[]> tokensFromCurrentDocument =
      Utility.tokenizeAndGetMapWithPositionsInDocument(document, corpus.getLanguage(), unstemmedTerms);

  return tokensFromCurrentDocument
      .entrySet()
      .stream().unordered()
      .map(tokenAndPositions -> new AbstractMap.SimpleEntry<>(
          tokenAndPositions.getKey(),
          new Term(
              new PostingList(new Posting(docIdThisDocument, tokenAndPositions.getValue())),
              tokenAndPositions.getKey())))
      .filter(e -> !e.getKey()/*the token*/.isBlank())
      .collect(
          Collectors.toConcurrentMap(
              Map.Entry::getKey,
              Map.Entry::getValue,
              (t1, t2) -> {
                throw new IllegalStateException("Duplicated keys should not be present, but were");
              }))
      .entrySet();
}
```

Inverted index creation

- *InvertedIndex* constructor
  - getPhoneticHashesOfDictionary
    - For **phonetic correction**

```
@NotNull
private ConcurrentMap<String, List<Term>> getPhoneticHashesOfDictionary() {
   return invertedIndex.entrySet()
         .stream().unordered().parallel()
         .map(stringTermEntry -> {
            var termList = new ArrayList<Term>();
            termList.add(stringTermEntry.getValue());
            return new Pair<>(Soundex.getPhoneticHash(stringTermEntry.getKey()), termList);
         })
         .collect(Collectors.toConcurrentMap(
               Map.Entry::getKey,
               AbstractMap.SimpleEntry::getValue,
               (a, b) -> {
                  a.addAll(b);
                  return a.stream().distinct().collect(Collectors.toList());
               },
               ConcurrentHashMap::new));
}
```

```
public InvertedIndex(@NotNull final Corpus corpus) {

   this.corpus = corpus;
   AtomicLong numberOfAlreadyProcessedDocuments = new AtomicLong(0L);

   try {
      invertedIndex = indexCorpusAndGet(corpus, numberOfAlreadyProcessedDocuments);
      postings = SkipList.createNewInstanceFromSortedCollection(
            invertedIndex.entrySet().stream().unordered().parallel()
                  .map(Map.Entry::getValue)
                  .map(Term::getListOfPostings)
                  .flatMap(Collection::stream)
                  .sorted()
                  .toList()/*unmodifiable list*/,
            Posting.DOC_ID_COMPARATOR);
      phoneticHashes = getPhoneticHashesOfDictionary();
      permutermIndex = createPermutermIndexAndGet(END_OF_WORD);
      permutermIndexWithoutEndOfWord = createPermutermIndexAndGet("");// NO end-of-word symbol
   } finally {
      System.out.println("\nIndexing ended");
   }

   Stemmer.AvailableStemmer stemmerTmp;   // defer assignment to final field
   try {
      stemmerTmp = Stemmer.AvailableStemmer.valueOf_(AppProperties.getInstance().get("app.stemmer"));
   } catch (IOException e) {
      stemmerTmp = Stemmer.AvailableStemmer.NO_STEMMING;
      Logger.getLogger(getClass().getCanonicalName())
            .log(Level.SEVERE, "Errors while reading app properties", e);
   }
   this.stemmer = stemmerTmp;

}
```

Inverted index creation

- *InvertedIndex* constructor
  - createPermutermIndexAndGet
    - For **spelling correction** and **wildcard queries**
    - Created from the unstemmed dictionary

```
private PatriciaTrie<Set<String>> createPermutermIndexAndGet(@NotNull String endOfWordSymbol) {
    Objects.requireNonNull(endOfWordSymbol);
    Stemmer stemmer = Utility.getStemmer() == null
            ? Stemmer.getStemmer(Stemmer.AvailableStemmer.NO_STEMMING)
            : Utility.getStemmer();
    PatriciaTrie<Set<String>> permutermIndex = new PatriciaTrie<>();
    getUnstemmedDictionary() // always use un-stemmed words
            .stream().distinct().unordered()
            .filter(Objects::nonNull).filter(str -> !str.isBlank())
            .flatMap(strFromDictionary -> {
                String str = strFromDictionary + endOfWordSymbol;
                return Arrays.stream(Utility.getAllRotationsOf(str))
                        .map(aRotation -> new Pair<>(aRotation, stemmer.stem(strFromDictionary, corpus.getLanguage())))
                        .filter(pair -> !pair.getValue().isBlank());
            })
            .forEachOrdered(pair -> {  // cannot use "collect" because PatriciaTrie is not thread-safe
                var key = pair.getKey();
                var rotation = pair.getValue();
                var rotationsForThisKey = permutermIndex.get(key);
                if (rotationsForThisKey == null) {  // true if the key is not present
                    Set<String> setOfRotations = ConcurrentHashMap.newKeySet();
                    setOfRotations.add(rotation);
                    permutermIndex.put(key, setOfRotations);
                } else {
                    rotationsForThisKey.add(rotation);
                }
                permutermIndex.put(pair.getKey(), pair.getValue());
            });
    return permutermIndex;
}
```

```
public InvertedIndex(@NotNull final Corpus corpus) {

    this.corpus = corpus;
    AtomicLong numberOfAlreadyProcessedDocuments = new AtomicLong(0L);

    try {
        invertedIndex = indexCorpusAndGet(corpus, numberOfAlreadyProcessedDocuments);
        postings = SkipList.createNewInstanceFromSortedCollection(
                invertedIndex.entrySet().stream().unordered().parallel()
                        .map(Map.Entry::getValue)
                        .map(Term::getListOfPostings)
                        .flatMap(Collection::stream)
                        .sorted()
                        .toList()/*unmodifiable list*/,
                Posting.DOC_ID_COMPARATOR);
        phoneticHashes = getPhoneticHashesOfDictionary();
        permutermIndex = createPermutermIndexAndGet(END_OF_WORD);
        permutermIndexWithoutEndOfWord = createPermutermIndexAndGet("");  // NO end-of-word symbol
    } finally {
        System.out.println("\nIndexing ended");
    }

    Stemmer.AvailableStemmer stemmerTmp;   // defer assignment to final field
    try {
        stemmerTmp = Stemmer.AvailableStemmer.valueOf_(AppProperties.getInstance().get("app.stemmer"));
    } catch (IOException e) {
        stemmerTmp = Stemmer.AvailableStemmer.NO_STEMMING;
        Logger.getLogger(getClass().getCanonicalName())
                .log(Level.SEVERE, "Errors while reading app properties", e);
    }
    this.stemmer = stemmerTmp;

}
```

Inverted index creation

- After indexing the *Movie* corpus

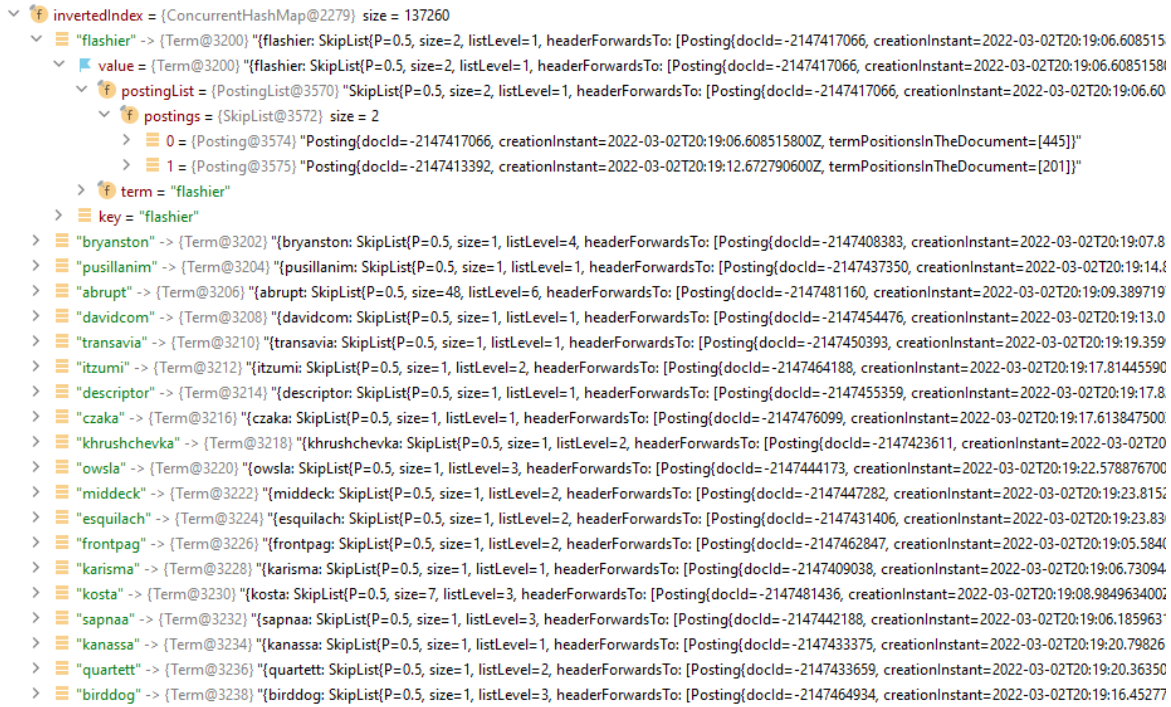


Figure Inverted index

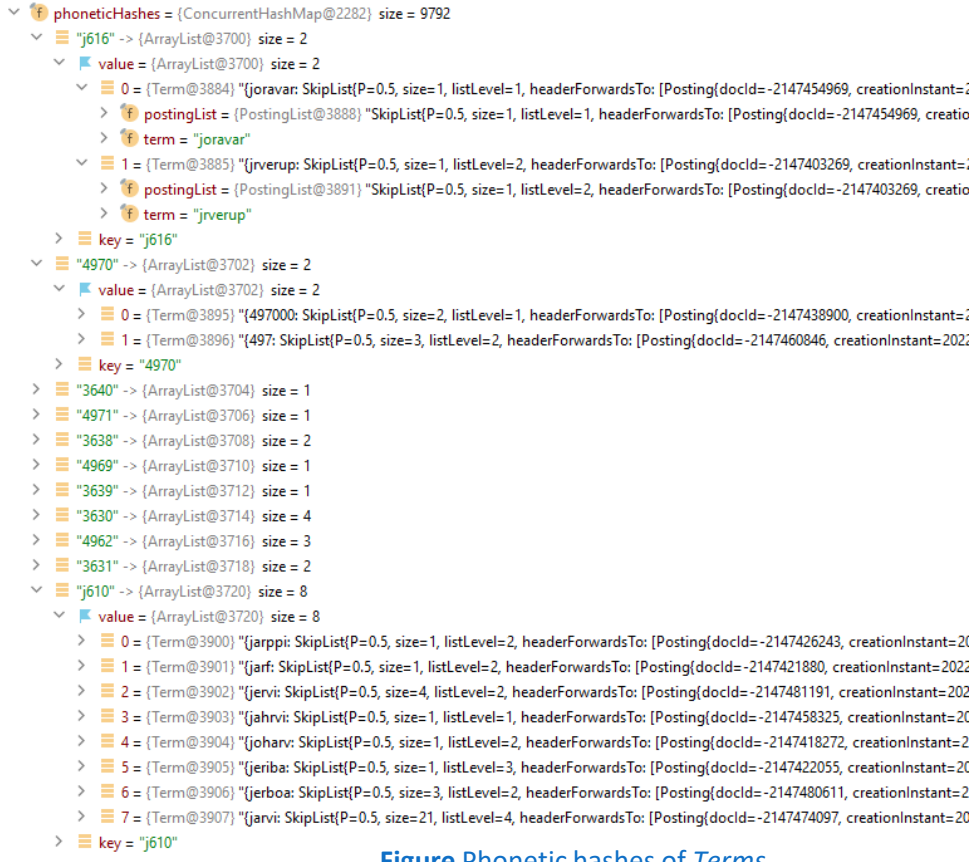


Figure Phonetic hashes of *Terms*

Inverted index creation

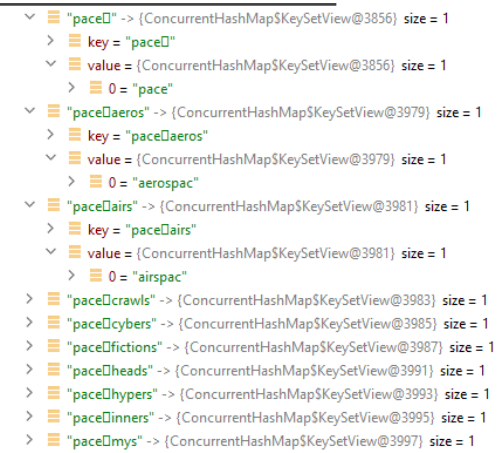
- After indexing the *Movie* corpus

Figure View of the permuterm index (for the *Movie* corpus), with the prefix "pace". The symbol □ is the *END_OF_WORD*. The view shows, e.g., that the key "pace□" is mapped to the un-rotated word "pace", while the key "pace□aeros" is mapped to the un-rotated word "aerospace" (the figure shows "aerospac", actually: it is an artifact caused by stemming). Entries following "pace□mys" are omitted.

Figure View of the permuterm index without the end of word symbol (for the *Movie* corpus), with the prefix "pace". The view shows, e.g., that the key "pace" is mapped to the un-rotated word "pace", while the key "paceaeros" is mapped to the un-rotated word "aerospace" (the figure shows "aerospac", actually: it is an artifact caused by stemming)

Inverted index creation

- Permuterm index considerations
  - The permuterm index (the one with the *END_OF_SYMBOL* "□") maps each rotated (and *unstemmed*) word to exactly one un-rotated (and *stemmed*) word
    - One-to-one mapping ➔ values might be *String* instead of *Set<String>*
      - But that might not be true for the permuterm index without "□"
        - E.g., "paces" maps to both "pace" and "space"
      - For uniformity and code-reuse, both are saved as *PatriciaTrie<Set<String>>*
        - *PatriciaTrie<?>* allows to **scan by prefix**
        - Key saved as *String*, values as *Set<String>*
  - Keys are always unstemmed
    - Both for wildcard and misspelled queries, users insert the word they want to query about
      - Not its stemmed version
  - Values are stemmed
    - If the system is configured to perform stemming
    - If they were not stemmed, they would not be found in the dictionary of the IR system



Operations

- Defined in *SkipList*
  - Union
  - Intersection
  - Difference
- Exploit the order of postings
  - fast calculation
  - $O(x + y)$
    - x = size of the first list
    - y = size of the second list
    - on average, better than $O(x + y)$ thanks to skip-pointers

```
public static <T extends Comparable<T>> SkipList<T> union(
      @NotNull final SkipList<T> a, @NotNull final SkipList<T> b,
      @NotNull final Comparator<@NotNull T> comparator) {

   SkipList<T> union = new SkipList<>(
         getBestMaxListLevelAccordingToExpectedSize(a.size() + b.size(), DEFAULT_P), comparator);

   var currentA = a.getFirstNodeOrNull();
   var currentB = b.getFirstNodeOrNull();

   while (currentA != null && currentB != null) {
      assert currentA.getKey() != null;
      assert currentB.getKey() != null;
      var comparison = comparator.compare(currentA.getKey(), currentB.getKey());
      if (comparison == 0) {
         union.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
         currentA = currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED);
         currentB = currentB.getNext(LOWEST_NODE_LEVEL_INCLUDED);
      } else if (comparison < 0) {
         union.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
         currentA = currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED);
      } else {
         union.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentB);
         currentB = currentB.getNext(LOWEST_NODE_LEVEL_INCLUDED);
      }
   }
   while (currentA != null) {   // add missing nodes from listA
      union.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
      currentA = currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED);
   }
   while (currentB != null) {   // add missing nodes from listB
      union.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentB);
      currentB = currentB.getNext(LOWEST_NODE_LEVEL_INCLUDED);
   }

   return union;
}

static int getBestMaxListLevelAccordingToExpectedSize(int expectedSize, double P) {
   return Math.max(MIN_ALLOWED_LIST_LEVEL, (int) Math.round(log(1 / P, expectedSize)));
}
```

Operations

- Defined in *SkipList*
  - Union
  - Intersection
  - Difference
- Exploit the order of postings
  - fast calculation
  - $O(x + y)$
    - x = size of the first list
    - y = size of the second list
    - on average, better than $O(x + y)$ thanks to skip-pointers

```
public static <T extends Comparable<T>> SkipList<T> intersection(
      @NotNull final SkipList<T> a, @NotNull final SkipList<T> b,
      @NotNull final BiPredicate<@NotNull T, @NotNull T> insertPredicate,
      @NotNull final Comparator<@NotNull T> comparator) {

   SkipList<T> intersection = new SkipList<>(comparator);
   if (a.isEmpty() || b.isEmpty()) {
      return intersection;   // empty intersection
   }

   var nodeFinderA = new NodeFinder<>(a.getHeader());
   var nodeFinderB = new NodeFinder<>(b.getHeader());

   var currentA = a.getFirstNodeOrNull();
   var currentB = b.getFirstNodeOrNull();

   while (currentA != null && currentB != null) {
      assert currentA.getKey() != null;
      assert currentB.getKey() != null;
      var comparison = comparator.compare(currentA.getKey(), currentB.getKey());
      if (comparison == 0) {
         if (insertPredicate.test(currentA.getKey(), currentB.getKey())) { // only keys matter for SkipList
            intersection.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
         }
         currentA = currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED);
         currentB = currentB.getNext(LOWEST_NODE_LEVEL_INCLUDED);
      } else if (comparison < 0) {
         var nextNode = nodeFinderA.findNextNode(currentB.getKey());
         currentA = nextNode == null ? currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED) : nextNode;
      } else {
         var nextNode = nodeFinderB.findNextNode(currentA.getKey());
         currentB = nextNode == null ? currentB.getNext(LOWEST_NODE_LEVEL_INCLUDED) : nextNode;
      }
   }

   return intersection;
}
```

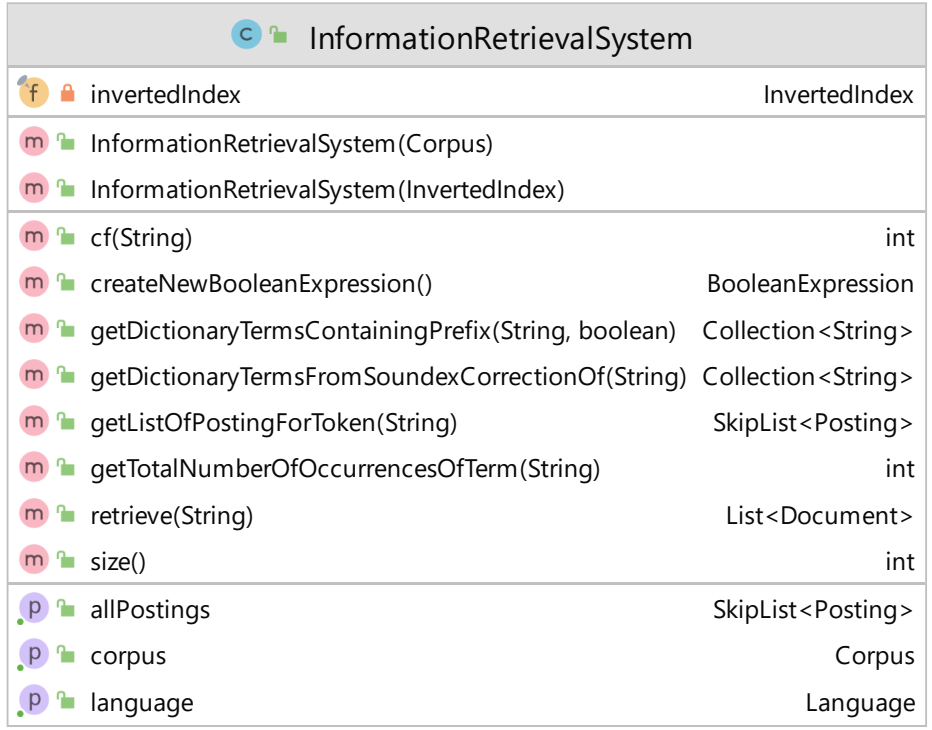
Operations

- Defined in *SkipList*
  - Union
  - Intersection
  - Difference
- Exploit the order of postings
  - fast calculation
  - $O(x + y)$
    - x = size of the first list
    - y = size of the second list
    - on average, better than $O(x + y)$ thanks to skip-pointers

```
public static <T extends Comparable<T>> SkipList<T> difference(
    @NotNull final SkipList<T> a, @NotNull final SkipList<T> b,
    @NotNull final BiPredicate<@NotNull T, @NotNull T> excludePredicate,
    @NotNull final Comparator<@NotNull T> comparator) {

  SkipList<T> difference = new SkipList<>(a.skipListMap.getMaxListLevel(), a.skipListMap.getP());
  if (a.isEmpty()) {
    return difference;   // empty difference
  }

  var nodeFinderB = new NodeFinder<>(b.getHeader());

  var currentA = a.getFirstNodeOrNull();
  var currentB = b.getFirstNodeOrNull();

  while (currentA != null && currentB != null) {
    assert currentA.getKey() != null;
    assert currentB.getKey() != null;
    var comparison = comparator.compare(currentA.getKey(), currentB.getKey());
    if (comparison == 0) {
      if (!excludePredicate.test(currentA.getKey(), currentB.getKey())) {
        difference.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
      }

      Function<@NotNull SkipListNode<T, ?>, @Nullable SkipListNode<T, ?>> moveCursor = node -> {
        SkipListNode<T, ?> tmp;
        SkipListNode<T, ?> current = node;
        while ((tmp = current.getNext(LOWEST_NODE_LEVEL_INCLUDED)) != null
            && current.getKey() != null && tmp.getKey() != null
            && comparator.compare(current.getKey(), tmp.getKey()) == 0) {
          current = tmp;
        }
        // if here, the comparator said that the next node is not equal to the current one
        return current.getNext(LOWEST_NODE_LEVEL_INCLUDED); // move cursor ahead
      };

      currentA = moveCursor.apply(currentA);
      currentB = moveCursor.apply(currentB);

    } else if (comparison < 0) {
      //noinspection unchecked
      difference.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
      currentA = currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED);
    } else {
      var nextNode = nodeFinderB.findNextNode(currentA.getKey());
      currentB = nextNode == null ? currentB.getNext(LOWEST_NODE_LEVEL_INCLUDED) : nextNode;
    }
  }
  while (currentA != null) {   // add missing nodes from listA
    difference.skipListMap.copyNodeAndInsertAtEnd((SkipListNode<T, Object>) currentA);
    currentA = currentA.getNext(LOWEST_NODE_LEVEL_INCLUDED);
  }

  return difference;
}
```

The IR System

- Creates and uses the entities
  - for storing and fast retrieval



The IR System

- Creates and uses the entities
  - for storing and fast retrieval

```
public class InformationRetrievalSystem implements Serializable {

   @NotNull private final Corpus corpus;
   @NotNull private final InvertedIndex invertedIndex;

   public InformationRetrievalSystem(@NotNull Corpus corpus) {
      this.corpus = Objects.requireNonNull(corpus);
      this.invertedIndex = new InvertedIndex(corpus);
   }

   public InformationRetrievalSystem(@NotNull InvertedIndex invertedIndex) {
      this.invertedIndex = invertedIndex;
      this.corpus = invertedIndex.getCorpus();
   }

   public SkipList<Posting> getListOfPostingForToken(@NotNull final String normalizedToken) {
      return invertedIndex.getListOfPostingsForToken(normalizedToken);
   }

   public int getTotalNumberOfOccurrencesOfTerm(@NotNull String term) {
      return invertedIndex.cf(term);
   }

   public Collection<String> getDictionaryTermsContainingPrefix(@NotNull String prefix, boolean ignoreEndOfWord) {
      return invertedIndex.getDictionaryTermsContainingPrefix(prefix, ignoreEndOfWord);
   }

   public BooleanExpression createNewBooleanExpression() {
      return new BooleanExpression(this);
   }

   public List<Document> retrieve(@NotNull String queryString) {
      return createNewBooleanExpression().parseQuery(queryString).evaluate();
   }

   public SkipList<Posting> getAllPostings() {
      return invertedIndex.getAllPostings();
   }

   @NotNull
   public Corpus getCorpus() {
      return corpus;
   }

   @NotNull
   public Language getLanguage() {
      return corpus.getLanguage();
   }

   public Collection<String> getDictionaryTermsFromSoundexCorrectionOf(@NotNull String word) {
      return invertedIndex.getDictionaryTermsFromSoundexCorrectionOf(word);
   }

   public int size() {
      return corpus.size();
   }

   public int cf(String str) {
      return invertedIndex.cf(str);
   }

}
```

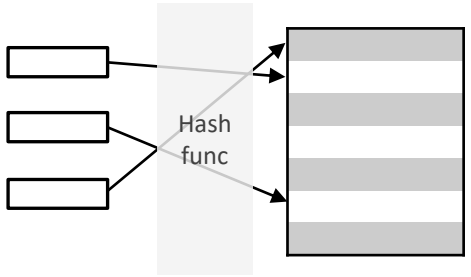
Boolean model

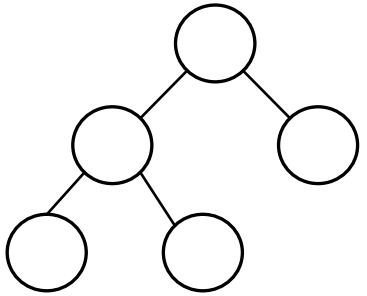
END

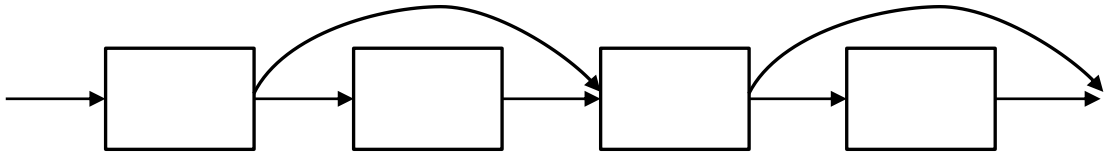
Data structures

Data structures

- Concurrent HashMap
- Patricia Trie
- SkipList







Concurrent HashMap

- O(1) access time
- Adding a new element can require copying all elements in a bigger HashMap
  - to avoid collisions
- *Concurrent* for the multithread environment
- Based on a *hash function*
- Cannot be used for fast prefix search
- Used for the corpus (access by docId) and the inverted index (access by token)

Patricia Trie

- Space-optimized trie
  - Prefix tree
- Sorted and hierarchical
  - Perfect for prefix search
- Used for the permuterm index

"cab□" -> "cab"
"cabal□" -> "cabal"
"cabalist□" -> "cabalist"
"cabalists□" -> "cabalist"
"caball□" -> "cabal"
"caballero□" -> "caballero"
"caballos□" -> "caballo"
"caballs□" -> "cabal"
"cabals□" -> "cabal"
"cabana□" -> "cabana"
"cabanas□" -> "cabana"

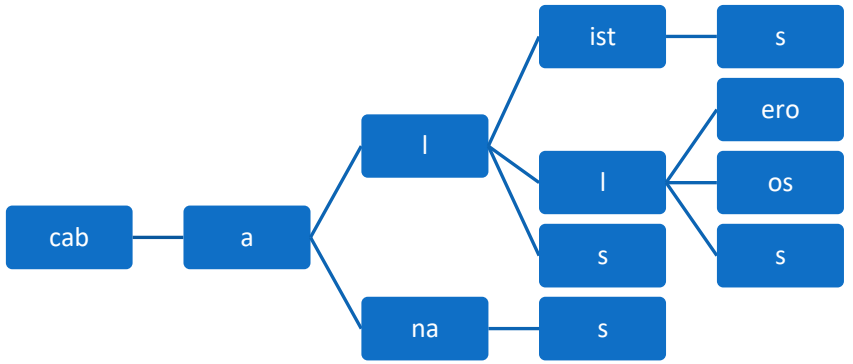


Figure Subtree of the permuterm index (saved as *Patricia Trie*) with some of the words starting with "cab" and not containing □ in the middle (this is just an illustrative view). The figure shows the keyset of the *Patricia Trie*.

Notice:
1) □ is the END OF WORD symbol, used for wildcard queries
2) all rotations of the words are saved in the permuterm index
3) several distinct keys may refer to the same term due to the preprocessing of tokens
4) invalid characters are filtered out, e.g.: "doña" (Spanish) becomes "doa"

SkipList

- Probabilistic data structure
- It is a **sorted set**
  - implemented as a linked list with **forward pointers**
  - which uses a **probabilistic balance**
  - It is a set because it does not contain duplicates
- Access time
  - Worst case: $\mathcal{O}(n)$
  - Average: $\mathcal{O}(\log n)$ thanks to **forward pointers**
- From:
  - W. Pugh, “Skip Lists: A Probabilistic alternative to balanced trees”, *Communications of the ACM*, Vol. 33, Number 6, June 1990.

SkipList

- Used for the posting lists in this project
- *Un*coupled with the IR system
  - SkipList can be used for the posting list, but it is a general data-structure
  - class *Posting* does not need any changes
  - class *PostingList* uses a *SkipList* as any other collection data-structure

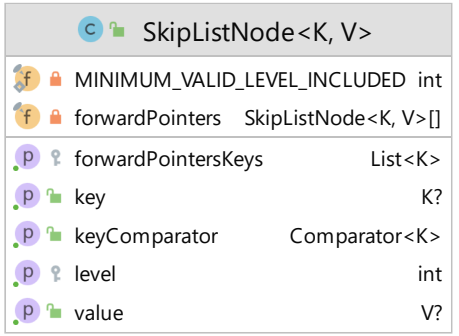
SkipList

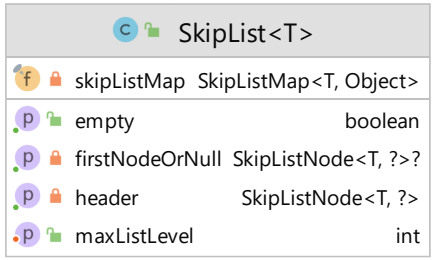
- Skip list vs Sorted Linked List
  - Forward pointers of skip lists allow to skip some elements

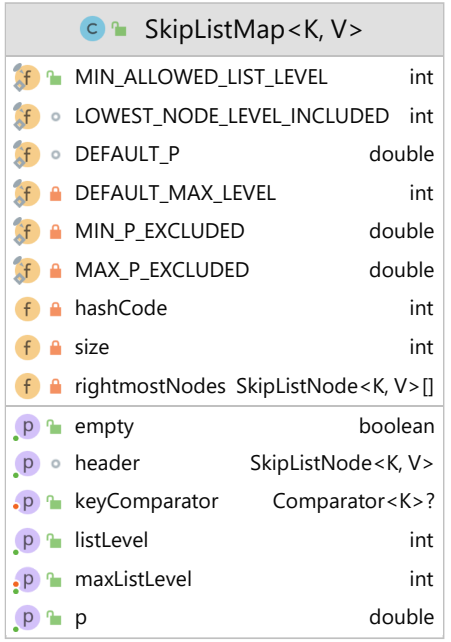
    ➔ Faster search
- Skip list vs Binary tree
  - Binary trees require
    - to be kept balanced for maintaining good performances
    - more complex algorithms
      - E.g., for tree traversal
      - not suitable for posting lists

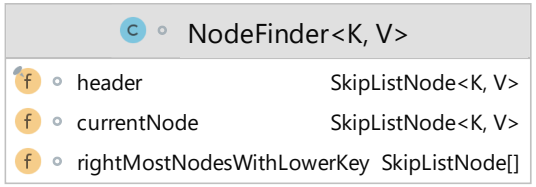
SkipList

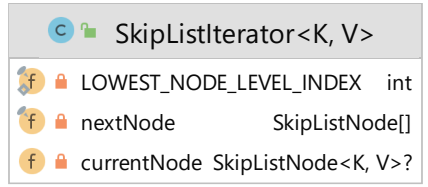
- *SkipList* is based on the more general *SkipListMap*
  - Using only the *KeySet*
- Each node has k forward pointers
  - k is chosen randomly between 1 and K (both included)
    - K is the *max level*
    - k is not constant for all the nodes
  - the node is called ***level k* node**
  - the **list level** is $\max_{nodes} k$











SkipList

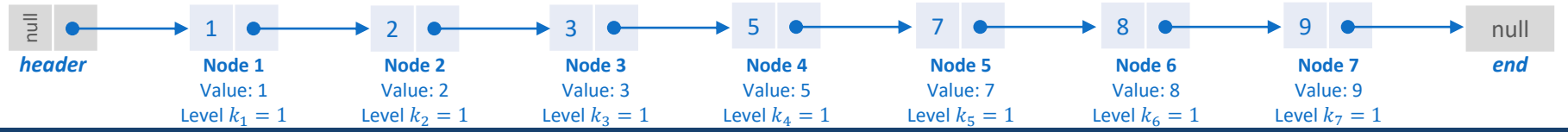
- If *MaxListLevel* is 1, then we have a linked list
  - Then at most n nodes must be examined to find a specific node (where n is the length of the list)

MaxListLevel is 1 for this example

```
final int MAX_LIST_LEVEL = 1;
SkipList<Integer> l1 = new SkipList<>(MAX_LIST_LEVEL);
l1.add(1);
l1.add(9);
l1.add(7);
l1.add(8);
l1.add(3);
l1.add(5);
l1.add(2);
System.out.println("l1:\t" + l1);
```

```
l1:     SkipList{P=0.5, size=7, listLevel=1, headerForwardsTo: [1],
        nodes=[
                1:      {value: 1, forwardsTo: [2]
                2:      {value: 2, forwardsTo: [3]
                3:      {value: 3, forwardsTo: [5]
                4:      {value: 5, forwardsTo: [7]
                5:      {value: 7, forwardsTo: [8]
                6:      {value: 8, forwardsTo: [9]
                7:      {value: 9, forwardsTo: [null]
        ]}
```

Notice: values are *not* inserted in the correct order, but the skip list is ordered by construction.



SkipList

- If *MaxListLevel* is 1, then we have a linked list
  - Then at most n nodes must be examined to find a specific node (where n is the length of the list)

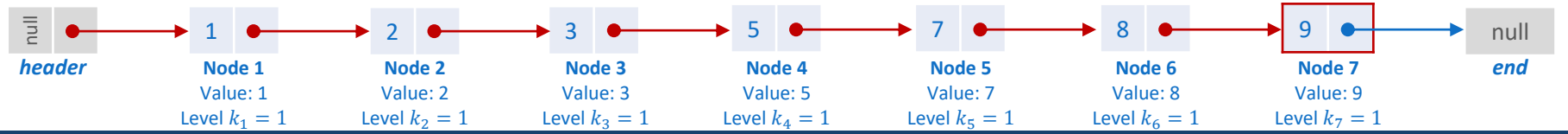
MaxListLevel is 1 for this example

```
final int MAX_LIST_LEVEL = 1;
SkipList<Integer> l1 = new SkipList<>(MAX_LIST_LEVEL);
l1.add(1);
l1.add(9);
l1.add(7);
l1.add(8);
l1.add(3);
l1.add(5);
l1.add(2);
System.out.println("l1:\t" + l1);
```

```
l1:     SkipList{P=0.5, size=7, listLevel=1, headerForwardsTo: [1],
        nodes=[
                1:      {value: 1, forwardsTo: [2]
                2:      {value: 2, forwardsTo: [3]
                3:      {value: 3, forwardsTo: [5]
                4:      {value: 5, forwardsTo: [7]
                5:      {value: 7, forwardsTo: [8]
                6:      {value: 8, forwardsTo: [9]
                7:      {value: 9, forwardsTo: [null]
        ]}
```

Notice: values are *not* inserted in the correct order, but the skip list is ordered by construction.

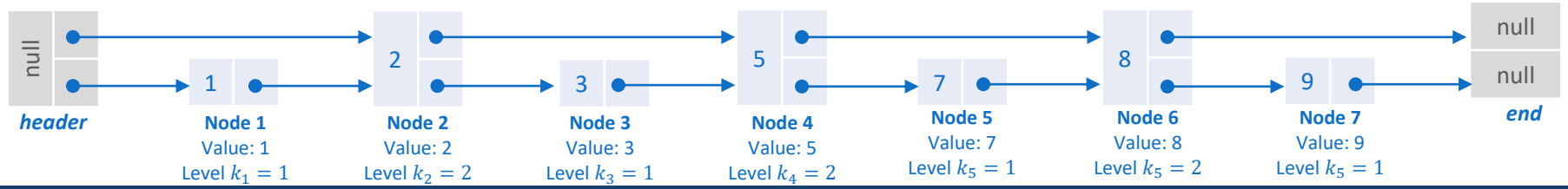
Example:
Searching for the value '9', starting from the *header*, requires to examine all nodes ($n = 7$) of the list must be examined in the worst- case.



SkipList

- If every two nodes in the skip list also have a pointer to the node two positions forward
  - Then at most $\left\lceil \frac{n}{2} \right\rceil$ nodes must be examined to find a specific node (where n is the length of the list)

Source code not available: this is just an illustrative example, hence, not reproducible	l1: SkipList{~~P=0.5~~, size=7, listLevel=2, headerForwardsTo: [1, 2], nodes=[1: {value: 1, forwardsTo: [2] 2: {value: 2, forwardsTo: [3, 5] 3: {value: 3, forwardsTo: [5] 4: {value: 5, forwardsTo: [7,8] 5: {value: 7, forwardsTo: [8] 6: {value: 8, forwardsTo: [9, null] 7: {value: 9, forwardsTo: [null]]}

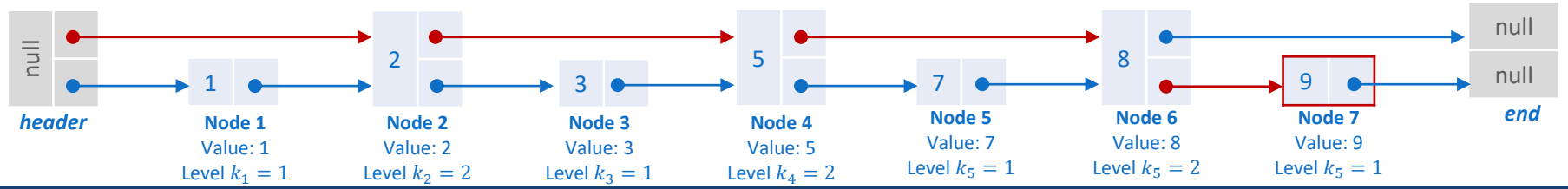


SkipList

- If every two nodes in the skip list also have a pointer to the node two positions forward
  - Then at most $\left\lceil \frac{n}{2} \right\rceil$ nodes must be examined to find a specific node (where n is the length of the list)

Source code not available: this is just an illustrative example, hence, not reproducible	l1: SkipList{~~P=0.5~~, size=7, listLevel=2, headerForwardsTo: [1, 2], nodes=[1: {value: 1, forwardsTo: [2] 2: {value: 2, forwardsTo: [3, 5] 3: {value: 3, forwardsTo: [5] 4: {value: 5, forwardsTo: [7,8] 5: {value: 7, forwardsTo: [8] 6: {value: 8, forwardsTo: [9, null] 7: {value: 9, forwardsTo: [null]]}

Example:
Searching for the value '9', starting from the *header*, requires to examine nodes 2, 4, 6 and 7 (the latest does containt the desired value), i.e., $\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{7}{2} \right\rceil = 4$ nodes, and this is the worst-case, because the value '9' is in the latest node of the skip list.



SkipList

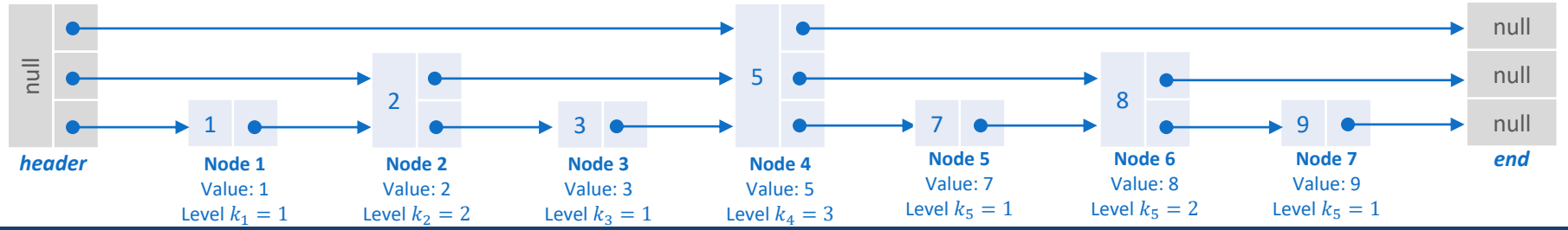
- In general:
  - If every (2^i)-th node has a pointer to 2^i nodes ahead
  - Then at most $\lceil \log_2 n \rceil$ nodes must be examined to find a specific node (where n is the length of the list)

Every (2^i)-th node has a pointer to 2^i nodes ahead:

- Every node (i=0) has a pointer to 1 node ahead (linked list)
- Every 2 nodes (i=1) has a pointer to the node 2 positions ahead
- Every 4 nodes (i=2) has a pointer to the node 4 positions ahead

Source code not available: this is just an illustrative example, hence, not reproducible

```
l1:   SkipList{~~P=0.5~~, size=7, listLevel=3, headerForwardsTo: [1, 2, 5],
      nodes=[
                1:   {value: 1, forwardsTo: [2]
                2:   {value: 2, forwardsTo: [3, 5]
                3:   {value: 3, forwardsTo: [5]
                4:   {value: 5, forwardsTo: [7,8, null]
                5:   {value: 7, forwardsTo: [8]
                6:   {value: 8, forwardsTo: [9, null]
                7:   {value: 9, forwardsTo: [null]
      ]}
```



SkipList

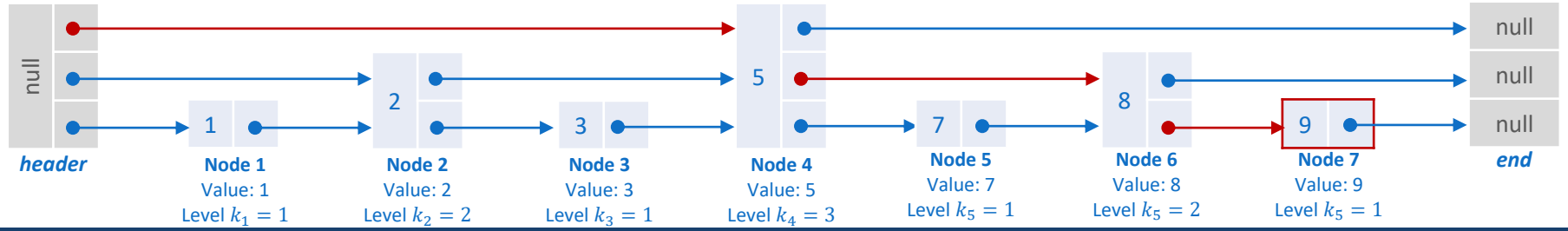
- In general:
  - If every (2^i)-th node has a pointer to 2^i nodes ahead
  - Then at most $\lceil \log_2 n \rceil$ nodes must be examined to find a specific node (where n is the length of the list)

Every (2^i)-th node has a pointer to 2^i nodes ahead:

- Every node (i=0) has a pointer to 1 node ahead (linked list)
- Every 2 nodes (i=1) has a pointer to the node 2 positions ahead
- Every 4 nodes (i=2) has a pointer to the node 4 positions ahead

Source code not available: this is just an illustrative example, hence, not reproducible

```
l1:   SkipList{~~P=0.5~~, size=7, listLevel=3, headerForwardsTo: [1, 2, 5],
      nodes=[
                1:   {value: 1, forwardsTo: [2]
                2:   {value: 2, forwardsTo: [3, 5]
                3:   {value: 3, forwardsTo: [5]
                4:   {value: 5, forwardsTo: [7,8, null]
                5:   {value: 7, forwardsTo: [8]
                6:   {value: 8, forwardsTo: [9, null]
                7:   {value: 9, forwardsTo: [null]
      ]}
```



Example:
Searching for the value '9', starting from the *header*, requires to examine nodes 2, 4 and 5 (the latest does contain the desired value), i.e., $\lceil \log_2 n \rceil = \lceil \log_2 7 \rceil = 3$ nodes, and this is the worst-case, because the value '9' is in the latest node of the skip list.

SkipList

- The probabilistic solution
  - Skip lists are balanced according to the number of forward pointers each node has
    - The number is randomly assigned between 1 and the *MaxListLevel* (parametrized)
    - with [on average] the same proportions as the general non-probabilistic case:
      - **$(n/2^i)$ nodes are (*level i*)-nodes**
        - 50% (level 2)-nodes
        - 25% (level 2)-nodes
        - 12.5% (level 3)-nodes

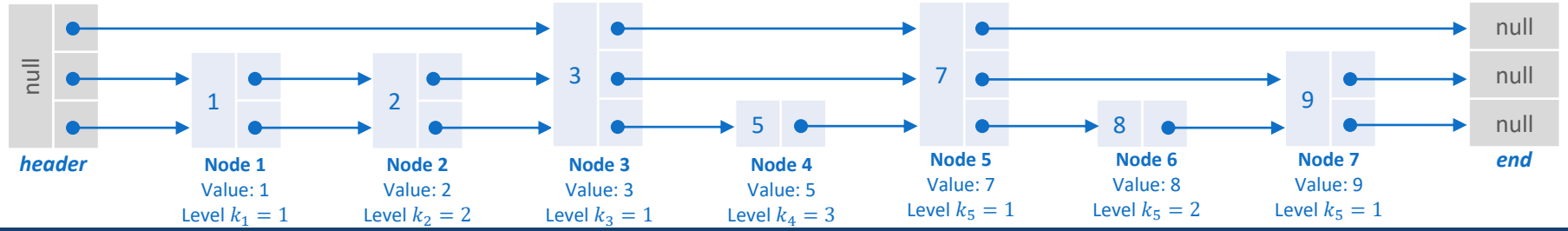
MaxListLevel is 3 for this example

```
final int MAX_LIST_LEVEL = 3;
SkipList<Integer> l1 = new SkipList<>(MAX_LIST_LEVEL);
l1.add(1);
l1.add(9);
l1.add(7);
l1.add(8);
l1.add(3);
l1.add(5);
l1.add(2);
System.out.println("l1:\t" + l1);
```

```
l1:     SkipList{P=0.5, size=7, listLevel=3, headerForwardsTo: [1, 1, 3],
        nodes=[
                1:      {value: 1, forwardsTo: [2, 2]
                2:      {value: 2, forwardsTo: [3, 3]
                3:      {value: 3, forwardsTo: [5, 7, 7]
                4:      {value: 5, forwardsTo: [7]
                5:      {value: 7, forwardsTo: [8, 9, null]
                6:      {value: 8, forwardsTo: [9]
                7:      {value: 9, forwardsTo: [null, null]
        ]}
```

Here proportions are not respected, in fact, there are:
- 2/7 ($\approx 30\%$) (level 1)-nodes
- 3/7 ($\approx 40\%$) (level 2)-nodes
- 2/7 ($\approx 30\%$) (level 3)-nodes

The list is too short to compute statistics.

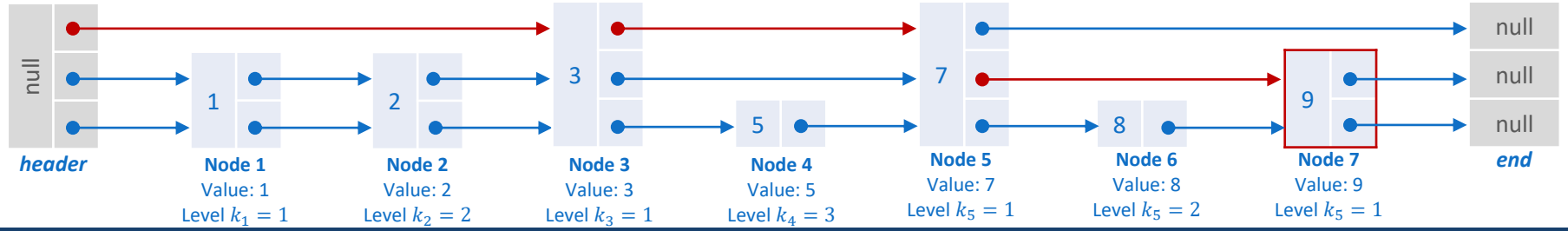


SkipList

- The probabilistic solution, for **searching**
  - In the worst case, all the nodes must be examined
  - In the average case, $\lceil \log_2 n \rceil$ nodes must be examined

```
final int MAX_LIST_LEVEL = 3;
SkipList<Integer> l1 = new SkipList<>(MAX_LIST_LEVEL);
l1.add(1);
l1.add(9);
l1.add(7);
l1.add(8);
l1.add(3);
l1.add(5);
l1.add(2);
System.out.println("l1:\t" + l1);
```

```
l1:     SkipList{P=0.5, size=7, listLevel=3, headerForwardsTo: [1, 1, 3],
        nodes=[
                1:      {value: 1, forwardsTo: [2, 2]
                2:      {value: 2, forwardsTo: [3, 3]
                3:      {value: 3, forwardsTo: [5, 7, 7]
                4:      {value: 5, forwardsTo: [7]
                5:      {value: 7, forwardsTo: [8, 9, null]
                6:      {value: 8, forwardsTo: [9]
                7:      {value: 9, forwardsTo: [null, null]
        ]}
```

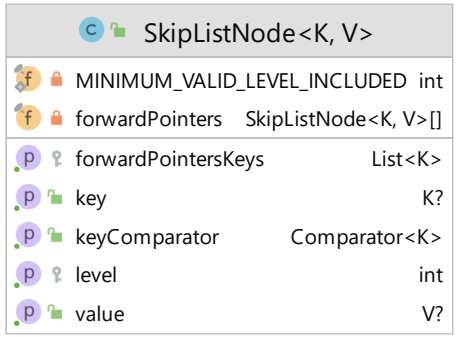


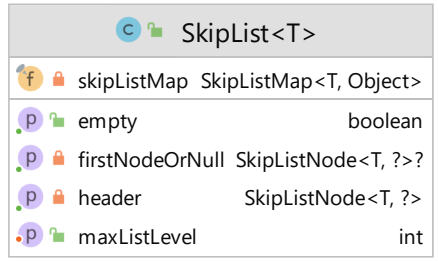
Example:
Searching for the value '9', starting from the *header*, requires to examine nodes 3, 7 and 9 (the latest does contain the desired value), i.e., 3 nodes (which is the average case ($\lceil \log_2 7 \rceil = 3$), even if '9' was at the latest node).

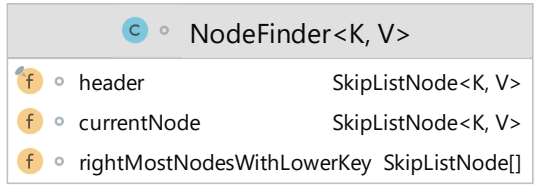
SkipList

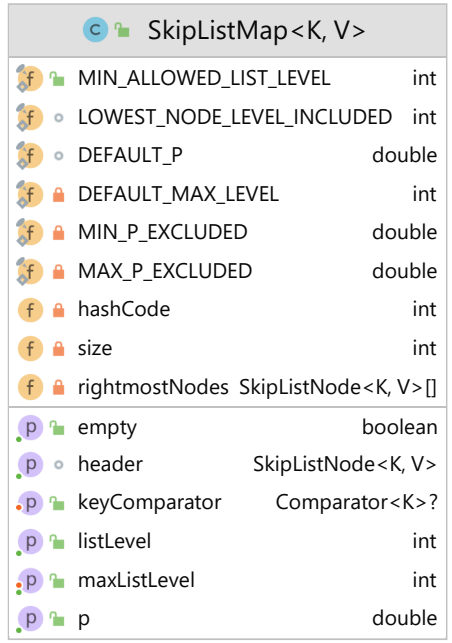
- Important **parameters** of the *SkipList*
  - P
    - Defines the distribution of (level i)-nodes
    - On average, there are $n\ P^i$ (level $i+1$)-nodes, with $0 < P < 1$
    - E.g., $P = \frac{1}{2} \Rightarrow \frac{n}{2^i}$ nodes are [on average] (level $i+1$)-nodes (like in the previous example)
      $P = \frac{1}{4} \Rightarrow \frac{n}{4^i}$ nodes are [on average] (level $i+1$)-nodes
  - List level
    - The level of the highest-level node currently in the list
  - Max list level
    - The maximum list level for the list
    - The optimal value
      - depends on *P* and the size of the list (n)
      - is $L(P) = 3\log_{1/P} n$

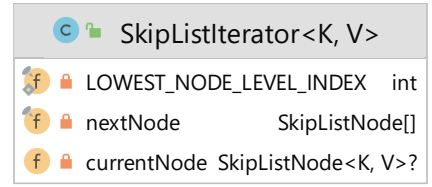
```
static int getBestMaxListLevelAccordingToExpectedSize(int expectedSize, double P) {
    return Math.max(MIN_ALLOWED_LIST_LEVEL, 3 * (int) Math.round(log(1 / P, expectedSize)));
}
```





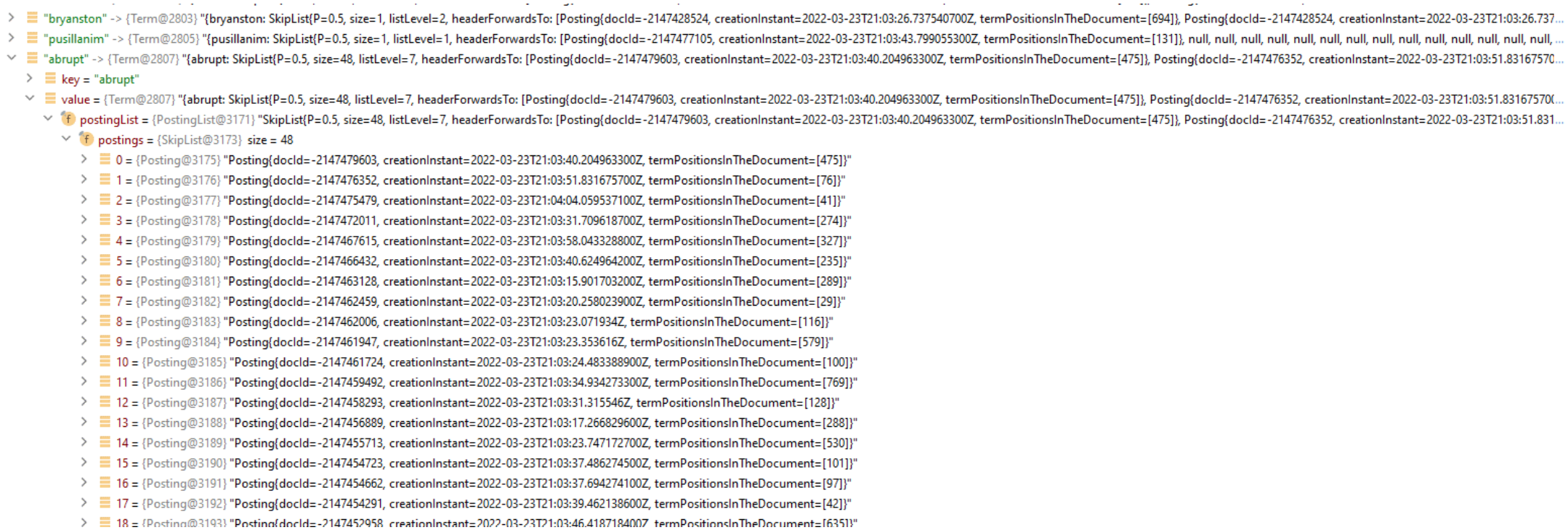






SkipList

- Used for the *PostingList*



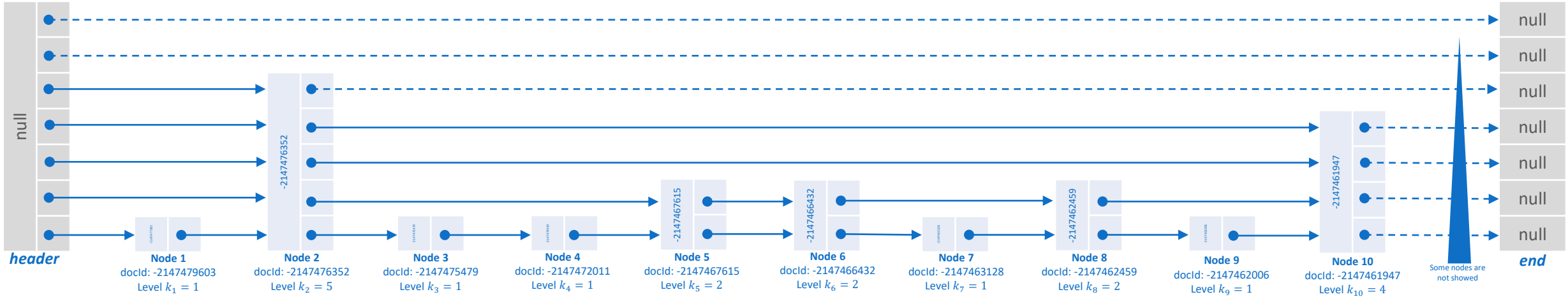
SkipList

- Used for the *PostingList*

SkipList{P=0.5, size=48, listLevel=7, headerForwardsTo: [Posting{docId=-2147479603, creationInstant=2022-03-23T21:03:40.204963300Z, termPositionsInTheDocument=[475]}, Posting{docId=-2147476352, creationInstant=2022-03-23T21:03:51.831675700Z, termPositionsInTheDocument=[76]}, Posting{docId=-2147476352, creationInstant=2022-03-23T21:03:51.831675700Z, terr
termPositionsInTheDocument=[402]}, null, null, null, null, null, null, null, null, null, null, null],
nodes=[

listLevel = 7 because no skiplist node has a level higher than 7

1: {value: Posting{docId=-2147479603, creationInstant=2022-03-23T21:03:40.204963300Z, termPositionsInTheDocument=[475]}, forwardsTo: [Posting{docId=-2147476352, creationInstant=2022-03-23T21:03:51.831675700Z, termPositionsInTheDocument=[76]}]}
2: {value: Posting{docId=-2147476352, creationInstant=2022-03-23T21:03:51.831675700Z, termPositionsInTheDocument=[76]}, forwardsTo: [Posting{docId=-2147475479, creationInstant=2022-03-23T21:04:04.059537100Z, termPositionsInTheDocument=[41]}, Posting{docId=-2147467615, creationInstant=2022-03-23T21:03:58.043328800Z, termPositionsI
3: {value: Posting{docId=-2147475479, creationInstant=2022-03-23T21:04:04.059537100Z, termPositionsInTheDocument=[41]}, forwardsTo: [Posting{docId=-2147472011, creationInstant=2022-03-23T21:03:31.709618700Z, termPositionsInTheDocument=[274]}]}
4: {value: Posting{docId=-2147472011, creationInstant=2022-03-23T21:03:31.709618700Z, termPositionsInTheDocument=[274]}, forwardsTo: [Posting{docId=-2147467615, creationInstant=2022-03-23T21:03:58.043328800Z, termPositionsInTheDocument=[327]}]}
5: {value: Posting{docId=-2147467615, creationInstant=2022-03-23T21:03:58.043328800Z, termPositionsInTheDocument=[327]}, forwardsTo: [Posting{docId=-2147466432, creationInstant=2022-03-23T21:03:40.624964200Z, termPositionsInTheDocument=[235]}, Posting{docId=-2147462459, creationInstant=2022-03-23T21:03:20.258023900Z, termPositio
6: {value: Posting{docId=-2147466432, creationInstant=2022-03-23T21:03:40.624964200Z, termPositionsInTheDocument=[235]}, forwardsTo: [Posting{docId=-2147463128, creationInstant=2022-03-23T21:03:15.901703200Z, termPositionsInTheDocument=[289]}]}
7: {value: Posting{docId=-2147463128, creationInstant=2022-03-23T21:03:15.901703200Z, termPositionsInTheDocument=[289]}, forwardsTo: [Posting{docId=-2147462459, creationInstant=2022-03-23T21:03:20.258023900Z, termPositionsInTheDocument=[29]}]}
8: {value: Posting{docId=-2147462459, creationInstant=2022-03-23T21:03:20.258023900Z, termPositionsInTheDocument=[29]}, forwardsTo: [Posting{docId=-2147462006, creationInstant=2022-03-23T21:03:23.071934Z, termPositionsInTheDocument=[116]}, Posting{docId=-2147461947, creationInstant=2022-03-23T21:03:23.353616Z, termPositionsInTheD
9: {value: Posting{docId=-2147462006, creationInstant=2022-03-23T21:03:23.071934Z, termPositionsInTheDocument=[116]}, forwardsTo: [Posting{docId=-2147461947, creationInstant=2022-03-23T21:03:23.353616Z, termPositionsInTheDocument=[579]}]}
10: {value: Posting{docId=-2147461947, creationInstant=2022-03-23T21:03:23.353616Z, termPositionsInTheDocument=[579]}, forwardsTo: [Posting{docId=-2147461724, creationInstant=2022-03-23T21:03:24.483388900Z, termPositionsInTheDocument=[100]}, Posting{docId=-2147459492, creationInstant=2022-03-23T21:03:34.934273300Z, termPositionsIn
...

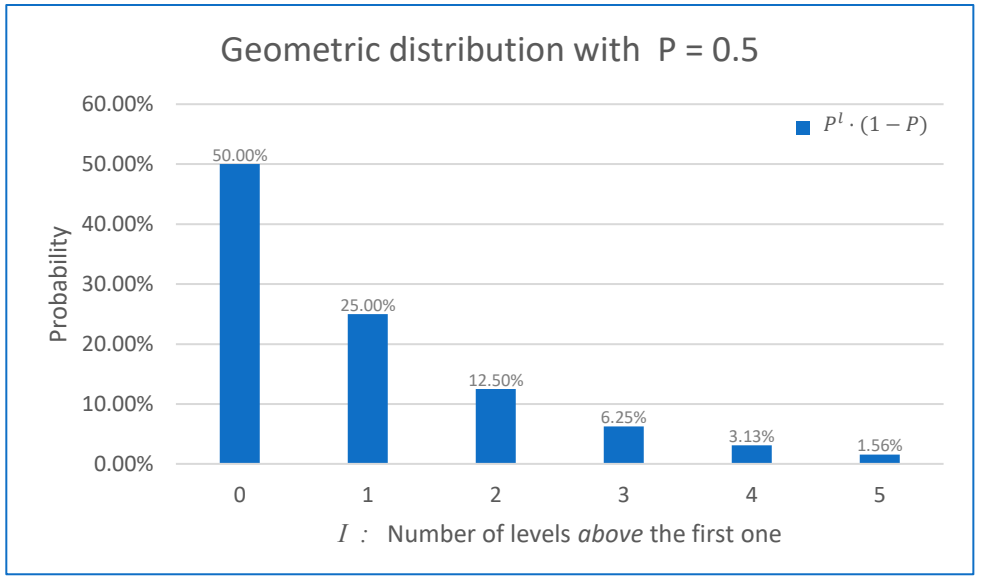


SkipList

- Probabilistic analysis
  - n is the number of nodes in the skip list (i.e., its length, without the header)
  - L_j is the **random variable** representing the **number of levels *above* the first one** of the j-th node,
    - $\forall j \in \{1,2,\dots,n\}$
    - "number of levels *above* the first one", because the first level is always present by construction
    - $L_j \in \mathbb{N}_{\geq 0}$, $\forall j \in \{1,2,\dots,n\}$
    - L_j is not (theoretically) bounded at the top
  - $M = 1 + \max_j L_j$ is the **random variable** representing the **level** of the skip list
    - " **1+** " is for the first level which is always present
    - Recall: L_j counts the number of level above the first one
    - The level of the skip list is the maximum actual level (counting also the first one) over all the n nodes of the skip list

SkipList

- Probabilistic analysis
  - $\Pr(L_j = l) = \begin{cases} 0\,, & \text{if } l \in \mathbb{Z}_{<0} \\ P^l \cdot (1-P)\,, & \text{if } l \in \mathbb{Z}_{\geq 0} \end{cases}, \quad \forall j \in \{1,2,\dots,n\}$ is the probability of the j-th node to have l more nodes above the first one
    - P^l is the probability for a node to be *at least* a (level $l+1$)-node
      - i.e., to have at least l levels above the first one
      - a node can be a (level $(i+1)$)-node only if it is a (level i)-node $\forall i \geq 1$
      - l levels must be climbed to reach the $(l+1)$-th level starting frome the first one
        - ➔ The product $\underbrace{P \cdot P \cdot \dots \cdot P}_{l \text{ levels}} = P^l$ must be considered
    - $(1-P)$ is the probability that the node has not any other upper levels
      - considering that the node already has $l+1$ levels
  - In practice, L_j follows a **geometric distribution**, where:
    - The **success** is the event "the node has another upper level to climb"
    - The **failure** is the event "the node has not any other upper levels to climb"
    - $P^l \cdot (1-P)$ is the probability to have l successes before the first failure



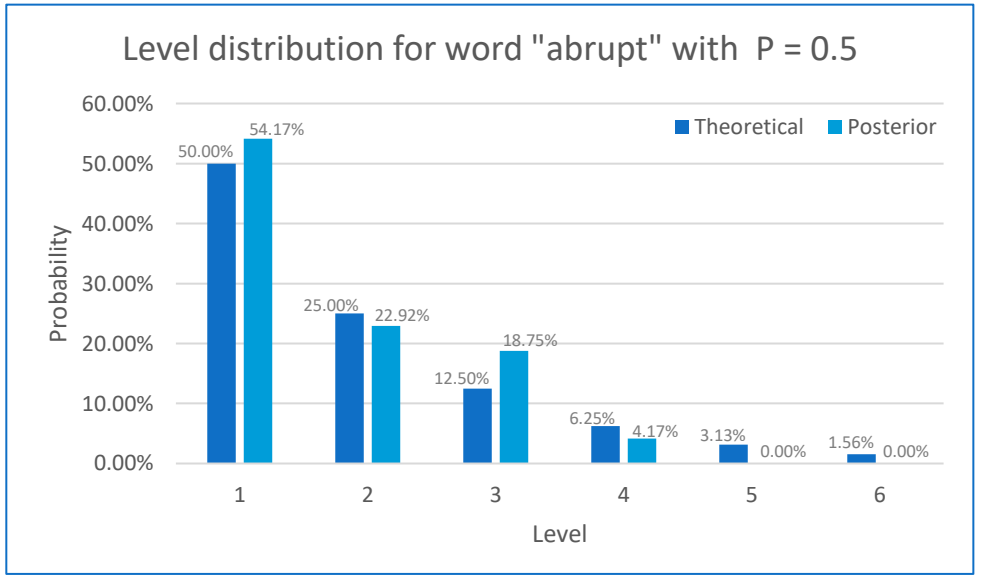
SkipList

- Probabilistic analysis
  - The SkipList is used for the *PostingList*
  - Level distribution for term “abrupt” is showed
    - *MaxListLevel* is 6

$P = 0.5$				
Level [l]	**Count** [N_l]	**Number of levels above the first one**	**Theoretical probability** [$P^l \cdot (1-P)$]	**Posterior probability**
1	26	0	50.00%	54.17%
2	11	1	25.00%	22.92%
3	9	2	12.50%	18.75%
4	2	3	6.25%	4.17%
5	0	4	3.13%	0.00%
6	0	5	1.56%	0.00%

Average number of forward pointers (i.e., average number of levels per node):

$$\frac{1}{n}\sum_{l} l\, N_l = \frac{1}{48}\,(1\cdot 26 + 2\cdot 11 + 3\cdot 9 + 4\cdot 2 + 5\cdot 0 + 6\cdot 0) = \frac{83}{48} \approx 1.73$$



SkipList

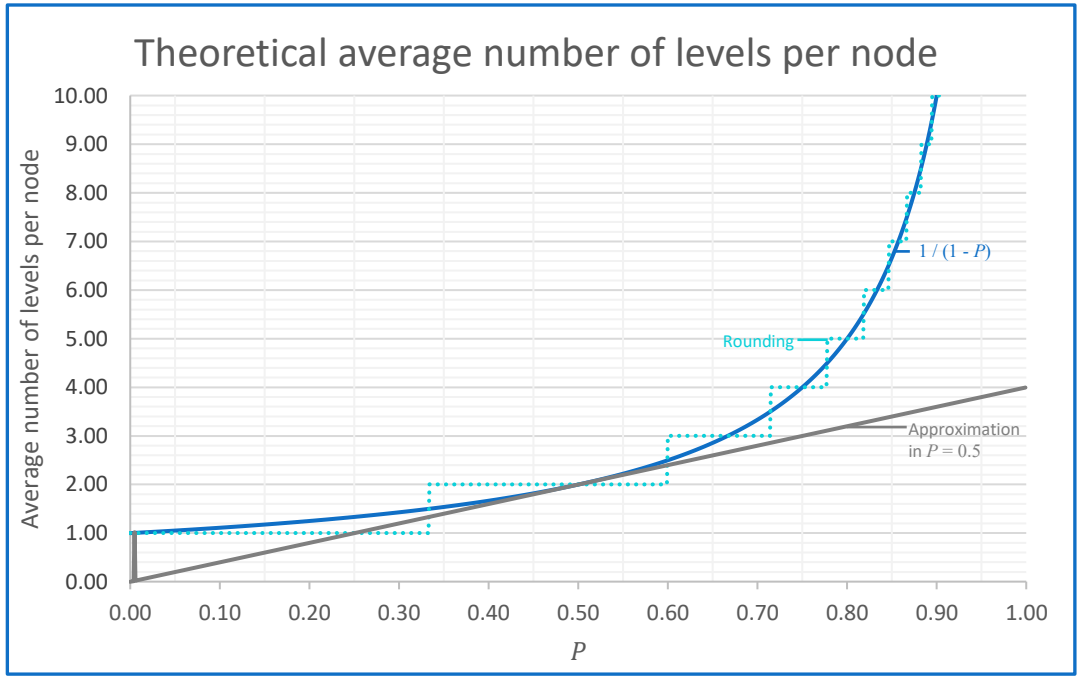
- Used space (memory usage)
  - The expected total number of pointers used in a skip list with n nodes is $\mathcal{O}(n)$
  - Proof
    - Each node has one pointer for each level
      - e.g., a level-3 node has 3 forward pointers
    - Each node is a (level l)-node with probability $\Pr(L_j = l-1) = P^{l-1} \cdot (1-P)\,,\ l \geq 1$
    - The average number of pointers of each node *above* the first one is

$$\mathrm{E}[L_j] = \sum_{l=1}^{\infty} l \Pr(L_j = l) = \sum_{l=1}^{\infty} l \cdot P^l \cdot (1-P) = (1-P)\sum_{l=1}^{\infty} l \cdot P^l$$

$$= (1-P)\sum_{l=1}^{\infty} \frac{P^l}{l^{-1}} = (1-P)\cdot \underbrace{\mathrm{Li}_{-1}(P)}_{\text{Polylogarithm of order -1 and argument } P} = (1-P)\cdot\frac{P}{(1-P)^2} = \frac{P}{1-P}$$

    - Counting the first level, the **average number of pointers of each node** is

$$1 + \mathrm{E}[L_j] = 1 + \frac{P}{1-P} = \boxed{\frac{1}{1-P}}$$



SkipList

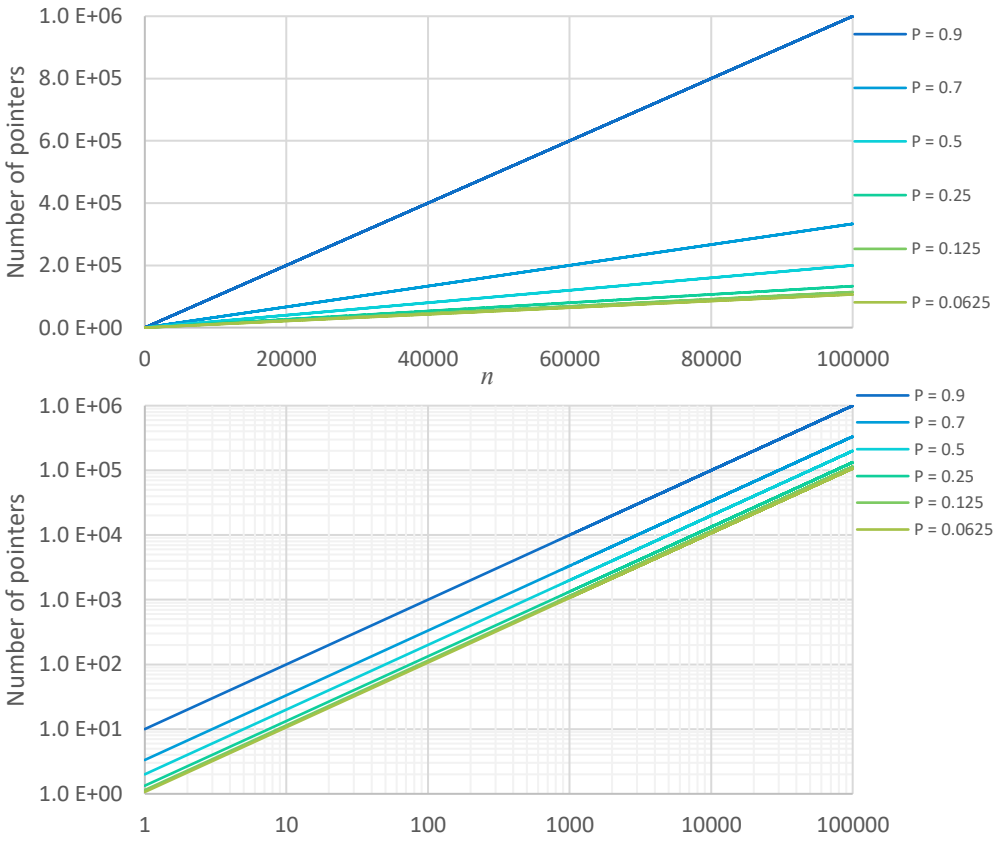
- Used space (memory usage)
  - *[… continues from the previous slide]*
  - The average total number of pointers is the sum over all the nodes of the list:

$$\sum_{j=1}^{n}\left(1+\mathrm{E}[L_j]\right)=\sum_{j=1}^{n}\frac{1}{1-P}=\frac{n}{1-P}$$

  - P is fixed ➔ **used space is $\mathcal{O}(n)$**

  - **Choosing P is a very important trade-off**
    - $0 < P < 1$
    - If P is too low, the most of nodes will be a (level 1)-node
      - The entire structure will be likely a linked list
      - Advantages of forward pointers will be lost
    - If P is too large, the occupied space grows very fast
    - In this project, **default value** is $P = 0.5$

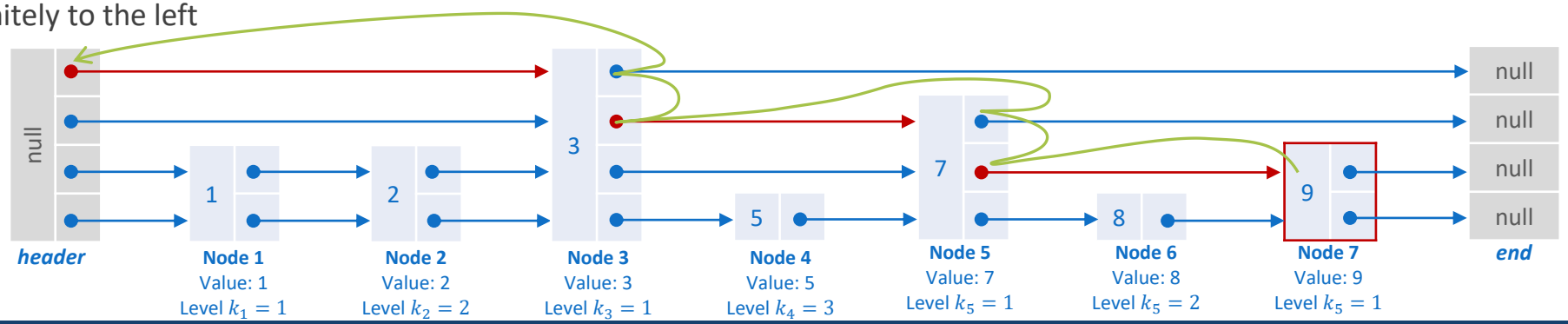
Expected used space is $\mathcal{O}$(n)



SkipList

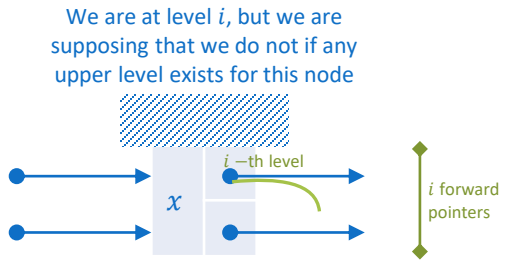
- Expected search cost
  - Analysis of the search path backwards
    - Starting from the searched (and found, by hypothesis) node
    - Travelling up and to the left
  - Hypothesis:
    - The level of each node is determined only after it is observed while backtracking the search path
    - The starting node is neither the head nor the tail
    - The list extends infinitely to the left

Example:
Searching for the value '9'.
For the expected search cost, we analyze the search path backwards.



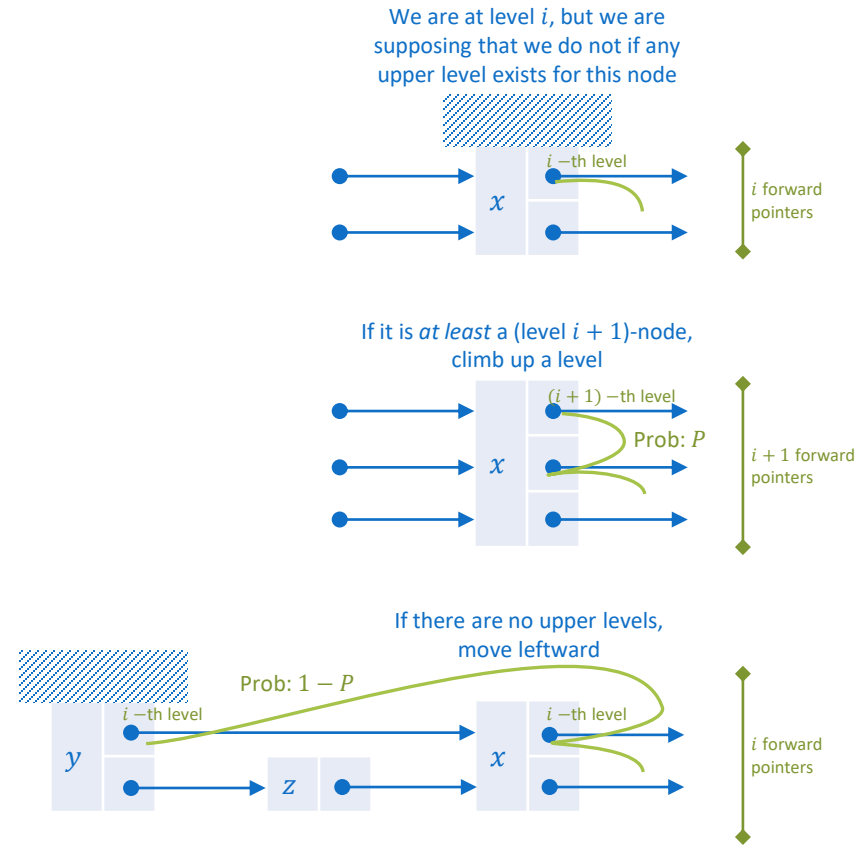
SkipList

- Expected search cost
  - We are at the i-th forward pointer of the currently examined node x
  - The actual level of the currently examined node is *unknown*
    - It must be *at least* a (level i)-node
    - It might be
      - a (level I)-node with $I > i$ (with probability P)
      - a (level i)-node (with probability $1 - P$)



SkipList

- Expected search cost
  - We are at the i-th forward pointer of the currently examined node x
  - The actual level of the currently examined node is *unknown*
    - If it is
      - a (level I)-node with $I > i$ (with probability P)
        - we climb a level up to the $(i+1)^{\text{th}}$ forward pointer of this node
      - a (level i)-node (with probability $1 - P$)
        - we move to the left, to the node y which is pointing the node x
          - the level of y is unknown
          - y is at least a (level i)-node



SkipList

- Expected search cost
  - We are at the i-th forward pointer of the currently examined node x
  - The expected cost to climb up l levels is

$$C_u[l] = \begin{cases} 0, & \text{if } l = 0 \\ \underbrace{(1-P)}_{\text{Probability that there are not upper levels}} \cdot \{\text{cost if } x \text{ has not any upper levels}\} + \underbrace{P}_{\text{Probability that there is at least another upper level}} \cdot \{\text{cost if } x \text{ has at least one more upper level}\}, & \text{if } l > 0 \end{cases}$$

$$= \begin{cases} 0, & \text{if } l = 0 \\ (1-P) \cdot (1 + C_u[l]) + P \cdot (1 + C_u[l-1]), & \text{if } l > 0 \end{cases}$$

Now we are at the i-th level. "1" is the cost for the current step (move backward to the previous node, because at the current node there were not upper levels to climb), then sum the cost $C_u[l]$ (after moving to the node to the left, we restart climbing from the i-th level).

Now we are at the i-th level."1" is the cost for the current step (climb one level up without changing node, because at the current node there was at least another upper level to climb), then sum the cost $C_u[l-1]$ (because, after climbing one level up, $l-1$ levels remain still to climb).

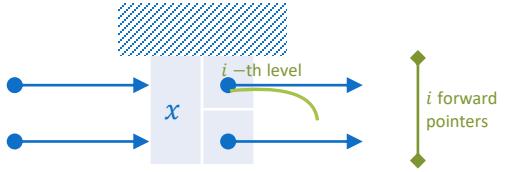
For $l > 0$:

$$C_u[l]_{l>0} = (1-P) \cdot (1 + C_u[l]) + P \cdot (1 + C_u[l-1]) \Leftrightarrow C_u[l] = \frac{P\, C_u[l-1]+1}{P} \cdot \mathbf{1}[l-1]$$

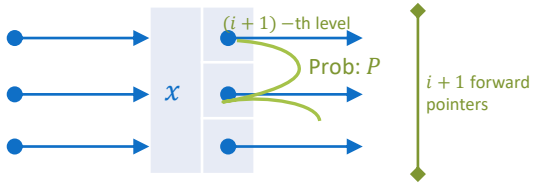
Hence: $C_u[0] = 0$, $C_u[1] = \frac{1}{P}$, $C_u[2] = \frac{2}{P}$, $C_u[3] = \frac{3}{P}$, … , $\boxed{C_u[l] = \frac{l}{P} \text{ with } l \geq 0}$

- $C_u[l]$ is the ***expected*** *cost* (i.e., the number of visited nodes in the path) *of* ***climbing*** (upward and backward) l levels
  - E.g., starting from level 1, $C_u[l]$ is the cost to climb (upward and backward) till the $(l+1)$-th level
    - assuming the currently examined list has at least $l+1$ levels

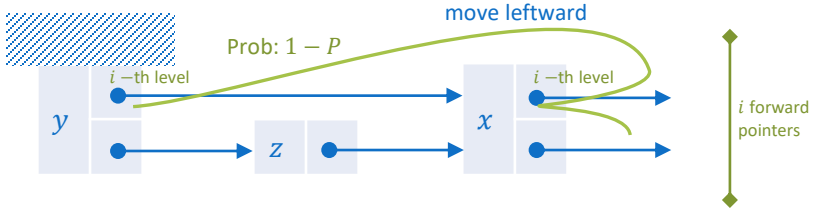
We are at level i, but we are supposing that we do not if any upper level exists for this node



If it is *at least* a (level $i+1$)-node, climb up a level



If there are no upper levels, move leftward

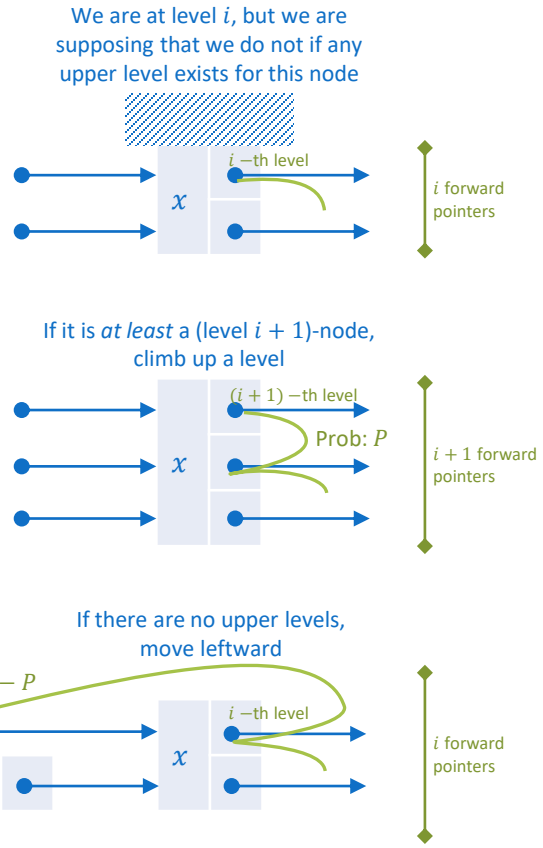


SkipList

- Expected search cost
  - Since the skip list has at most M levels (with M a random variable)
    - Starting from the first level, we have to climb $M-1$ levels to reach the uppermost one,
    - hence, we have the upper bound for the "vertical cost"

$$\mathrm{C_u}[l] = \frac{l}{P} \leq \frac{M-1}{P}\ ,\ \forall l \in \{0, 1, \dots, M-1\}$$

  - Recall:
    - $\mathrm{C_u}[l]$ *is not* the cost to climb till the l-th level
    - $\mathrm{C_u}[l]$ *is* the cost to climb l levels up
      - It depends on the starting level
        - If we start from level 1 and we must climb till level 5, the cost is $\mathrm{C_u}[5-1] = \mathrm{C_u}[4]$
        - If we start from level 3 and we must climb till level 5, the cost is $\mathrm{C_u}[5-3] = \mathrm{C_u}[2]$

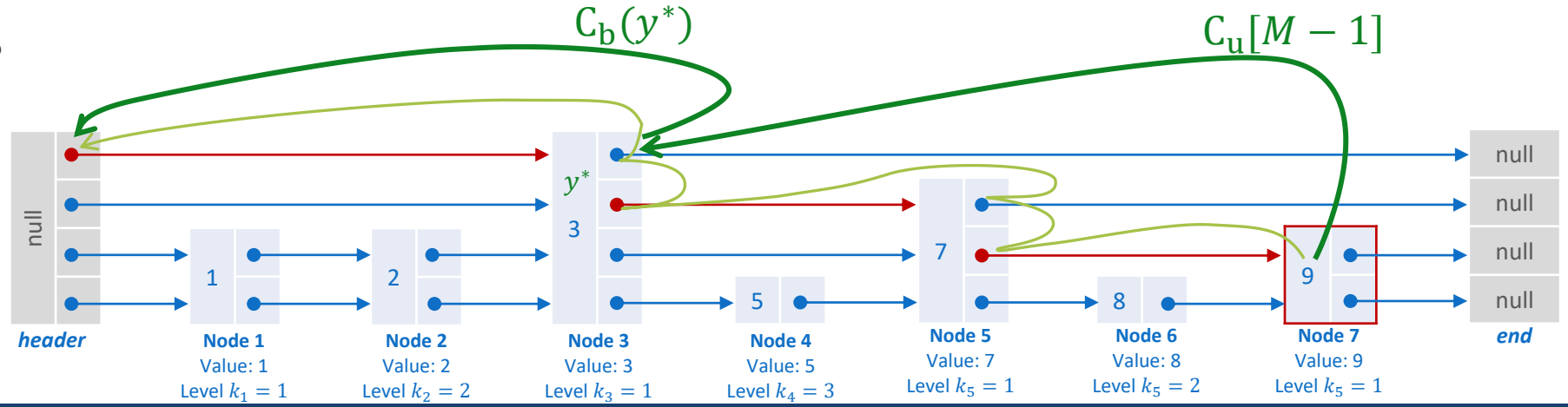


SkipList

- Expected search cost
  - Starting from level 1, we climb up to level M with "**vertical cost**" $\mathrm{C_u}[M-1]$
  - We might not have reached the header yet
    - When searching, we start from the header
  - Recall: M is the **random variable** representing the **height of the list**
  - Let $\mathrm{C_b}(y)$ the **cost to move backward till the header**, starting from a (level M)-node, let it be y, and **without changing level**
  - The total expected search cost is $\mathrm{C_u}[M-1] + \mathrm{C_b}(y^*)$
    - y^* is the (level M)-node reached by climbing up $M-1$ levels
      - It might be the header (in such case, $\mathrm{C_b}(y^*)|_{y^*=\mathrm{header}} = 0$)
    - " $+\mathrm{C_b}(y^*)$" means to sum the cost to move backward till the header
      - remaining at level $M-1$, which is the level of y^* (y^* reached with cost $\mathrm{C_u}[M-1]$)
      - $\mathrm{C_b}(y^*)$ is the number of (level M)-nodes that are between the header and the node reached by climbing $M-1$ levels up

SkipList

- Expected search cost
  - Example: search for the value ‘9’
  - When searching, we proceed forward and downward (→)
  - During this analysis, we proceed upward and backward (←)
  - In this example:
    - $M = 4\ ,\ P = \frac{1}{2}\ ,\ n = 7$
    - $\mathrm{C_u}[M-1] = \mathrm{C_u}[3] = \frac{3}{P} = 6$
    - $\mathrm{C_b}(y^*) = 1$
      - Only one step to reach the header from the 3rd node
    - Total search cost is $\mathrm{C_u}[3] + \mathrm{C_b}(y^*) = 6$



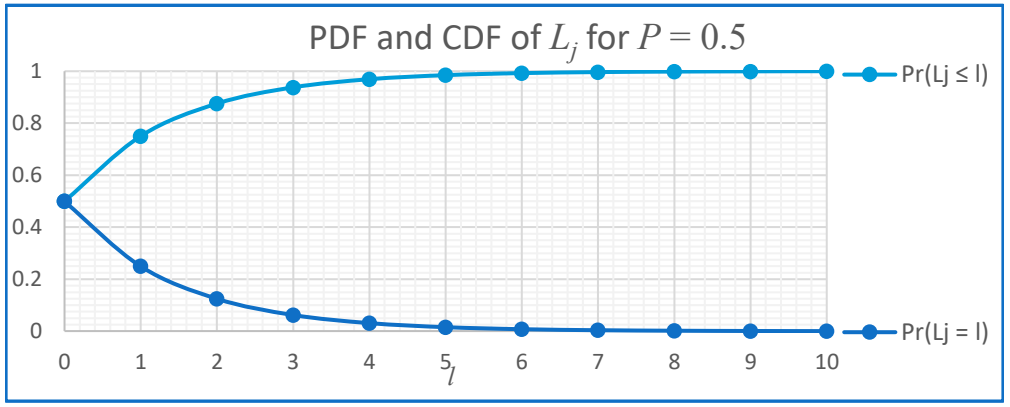
SkipList

- Expected search cost
  - We want to **find an upper bound for the overall expected search cost**
    - We already found the upper bound for the "**vertical cost**", i.e.: $\mathrm{C_u}[l] = \frac{l}{P} \leq \frac{M-1}{P}$, $\forall l \in \{0, 1, \dots, M-1\}$
    - Now: we are looking for an upper bound for the "**horizontal cost**" $\mathrm{C_b}(y^*)$
      - y^* is the (level M)-node of the skip list reached by climbing up $M-1$ levels from the searched node
  - Recall: $\Pr(L_j = l) = \begin{cases} 0\,, & \text{if } l \in \mathbb{Z}_{<0} \\ P^l \cdot (1-P)\,, & \text{if } l \in \mathbb{Z}_{\geq 0} \end{cases}$, $\forall j \in \{1,2,\dots,n\}$ is the probability of the j-th node to have l more nodes above the first one
  - Then, the CDF is:

$$\Pr(L_j \leq l) = \sum_{i=0}^{l} \Pr(L_j = l) = \sum_{i=0}^{l} P^l \cdot (1-P)$$

$$= (1-P) \sum_{i=0}^{l} P^l = (1-P) \cdot \frac{1-P^{l+1}}{1-P}$$

➡ $\Pr(L_j \leq l) = 1 - P^{l+1}$



SkipList

- Expected search cost
  - We want to **find an upper bound for the overall expected search cost**
    - We already found the upper bound for the "vertical cost", i.e.: $\mathrm{C_u}[l] = \frac{l}{P} \leq \frac{M-1}{P}\ ,\ \forall l \in \{0, 1, \dots, M-1\}$
    - Now: we are looking for an **upper bound for the "horizontal cost"** $\mathrm{C_b}(y^*)$
  - The probability that the list is higher than a given value m is:

$$\Pr(M > m) = 1 - \Pr(M \leq m) = 1 - \Pr(L_1 \leq m-1 \wedge L_2 \leq m-1 \wedge \cdots \wedge L_n \leq m-1)$$
$$= 1 - \prod_{j=1}^{n} \Pr(L_j \leq m-1) = 1 - \prod_{j=1}^{n} (1 - P^m) = 1 - (1-P^m)^n$$

$\Pr(M \leq m)$ is the probability that *all* nodes of the list have at most m levels, that is $\Pr(L_j \leq m-1)\ \forall j \in \{1,2,\dots,n\}$, i.e., the probability that all nodes have at most $m-1$ above the first one.

  - We can find the following upper bound:

$$\Pr(M > m) = 1 - (1-P^m)^n \leq nP^m$$

If $0 < a < 1$ then $1 - (1-a)^n \leq na\ ,\ \forall n \in \mathbb{N}_{\geq 0}$
Proof by induction:
If $n = 0 \vee n = 1$, then $1 - (1-a)^n \leq na$ holds
Supposing $1 - (1-a)^n \leq na$ holds for an arbitrary $n \in \mathbb{N}_{\geq 0}$, then, for $n+1$, we have:

$$1 - (1-a)^{n+1} = 1 - (1-a)(1-a)^n = (1-a) + a - (1-a)(1-a)^n\ ,$$ by adding and subtracting a
$$= (1-a)(1-(1-a)^n) + a \leq (1-a)\,na + a\ ,$$ for the inductive hypothesis
$$= (1 + n(1-a))a \leq (n+1)a$$ since $n(1-a) \leq n\ \forall (a,n) \in \{(a,n) | 0 < a < 1 \wedge n \in \mathbb{N}_{\geq 0}\}$

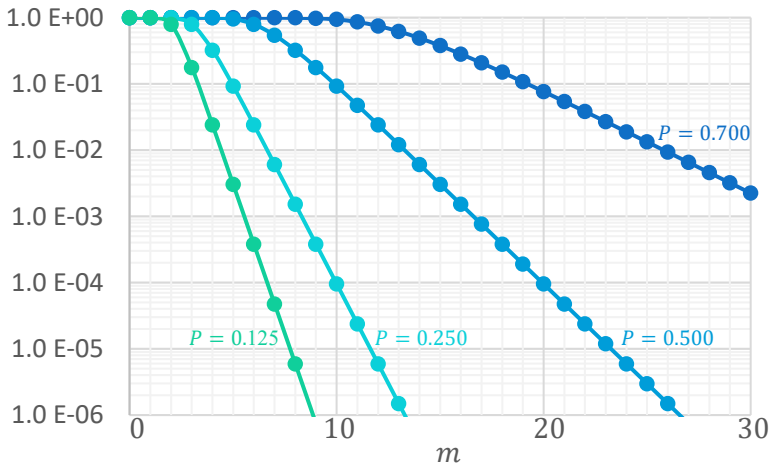
□

SkipList

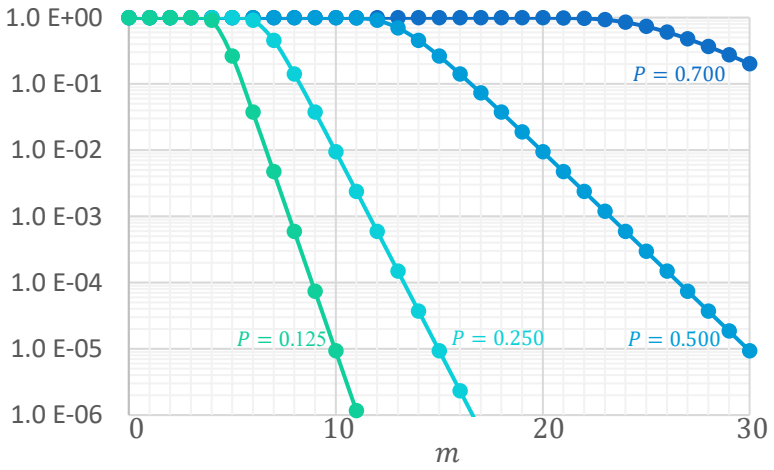
- Expected search cost
  - We can find the following upper bound:

$$\Pr(M > m) = 1 - (1 - P^m)^n \leq nP^m$$

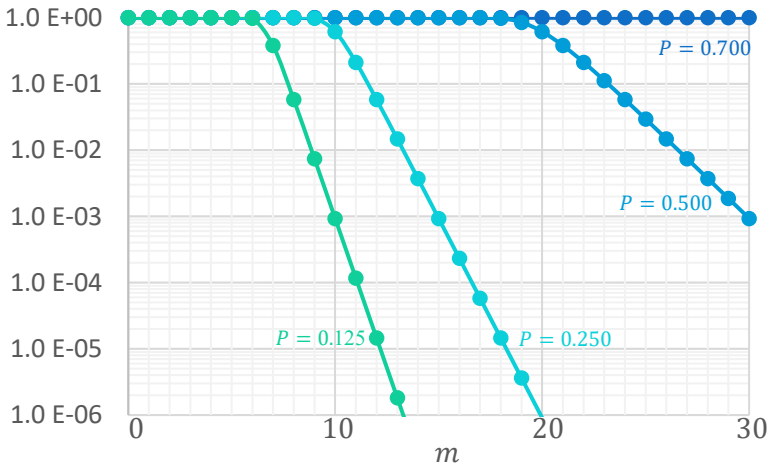
$\Pr(M > m), \quad n = 10^2$



$\Pr(M > m), \quad n = 10^4$



$\Pr(M > m), \quad n = 10^6$

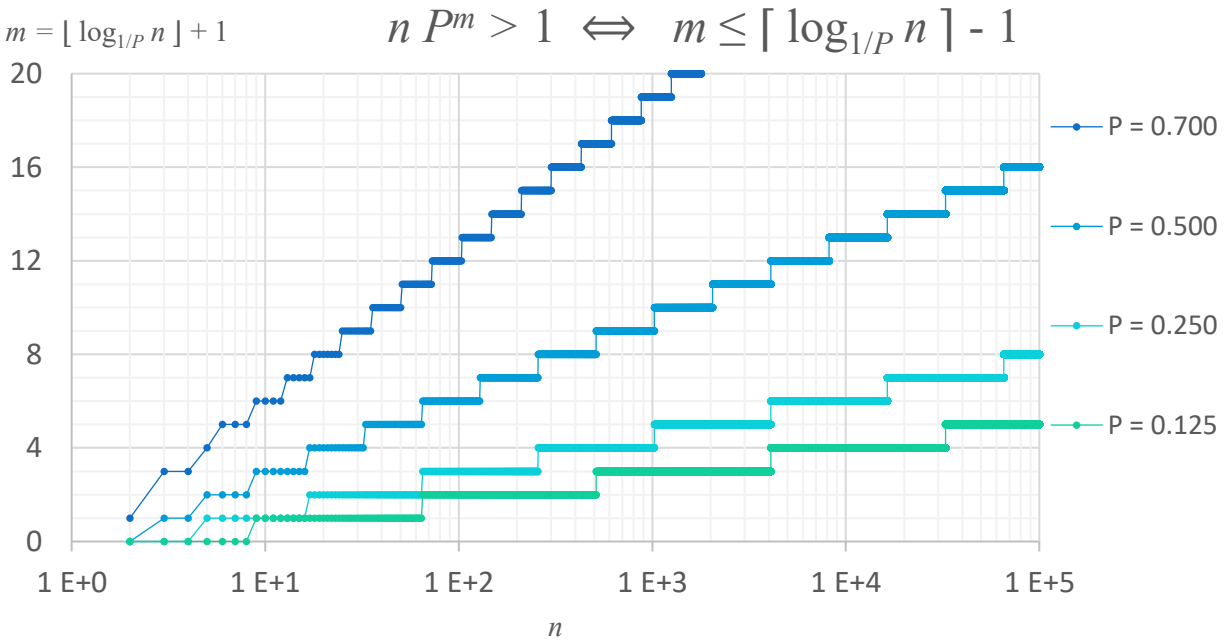


SkipList

- Expected search cost
  - We have found:
    - $\mathrm{C_u}[l] = \frac{l}{P} \leq \frac{M-1}{P}\ ,\ \forall l \in \{0, 1, \ldots, M-1\}$
    - $\Pr(M > m) = 1 - (1 - P^m)^n \leq nP^m = P^{m - \log_{\frac{1}{P}}(n)}$

Since $\Pr(M > m) \leq 1\ \forall m$, we have

$$\Pr(M > m) \leq \begin{cases} 1\ , & \text{if } nP^m > 1 \Leftrightarrow m \leq \left\lceil \log_{\frac{1}{P}} n \right\rceil - 1 \\ nP^m\ , & \text{otherwise} \end{cases}$$



SkipList

- Expected search cost
  - We have found:
    - $\mathrm{C_u}[l] = \frac{l}{P} \leq \frac{M-1}{P}\ ,\ \forall l \in \{0, 1, \dots, M-1\}$
    - $\Pr(M > m) = 1 - (1 - P^m)^n \leq \begin{cases} 1\ , \text{ if } m \leq \left\lceil \log_{\frac{1}{P}} n \right\rceil - 1 \\ nP^m\ , \text{ otherwise} \end{cases}$
  - The expected maximum list height is

$$\mathrm{E}[M] = \sum_{m=0}^{\infty} m \Pr(M = m) = \sum_{m=0}^{\infty} \Pr(M > m)$$

$$= \underbrace{\sum_{m=0}^{\left\lceil \log_{\frac{1}{P}}(n) \right\rceil - 1} \Pr(M > m)}_{\leq \left\lceil \log_{\frac{1}{P}}(n) \right\rceil} + \underbrace{\sum_{m=\left\lceil \log_{\frac{1}{P}}(n) \right\rceil}^{\infty} \Pr(M > m)}_{\leq \sum_{m=\left\lceil \log_{\frac{1}{P}}(n) \right\rceil}^{\infty} nP^m}$$

Since $\Pr(M > m) \leq 1\ \forall m$ and $nP^m > 1\ \forall m \leq \left\lceil \log_{\frac{1}{P}} n \right\rceil - 1$
Then $\Pr(M > m) \leq 1$ is the best upper bound we found $\forall m \leq \left\lceil \log_{\frac{1}{P}} n \right\rceil - 1$

If M is a discrete random variable which takes only positive integer values, then the following equalities hold for its expected value:

$$\mathrm{E}[M] = \sum_{m=0}^{\infty} m \Pr(M = m) = \sum_{m=1}^{\infty} \Pr(M \geq m) = \sum_{m=0}^{\infty} \Pr(M > m)$$

Proof

$$\Pr(M \geq m) = \Pr(M = m) + \Pr(M = m+1) + \cdots = \sum_{i=0}^{\infty} \Pr(M = m + i)$$

$$\Rightarrow \sum_{m=1}^{\infty} \Pr(M \geq m) = \sum_{m=1}^{\infty} \sum_{i=0}^{\infty} \Pr(M = m + i)$$

$$\begin{aligned}
&= \Pr(M = 1+0) + \Pr(M = 1+1) + \Pr(M = 1+2) + \cdots + && \text{for } m = 1 \\
&\quad \Pr(M = 2+0) + \Pr(M = 2+1) + \Pr(M = 2+2) + \cdots + && \text{for } m = 2 \\
&\quad \Pr(M = 3+0) + \Pr(M = 3+1) + \Pr(M = 3+2) + \cdots + && \text{for } m = 3 \\
&\quad \cdots && \cdots \\
&= \Pr(M = 1) + \Pr(M = 2) + \Pr(M = 3) + \cdots + && \text{for } m = 1 \\
&\quad \Pr(M = 2) + \Pr(M = 3) + \Pr(M = 4) + \cdots + && \text{for } m = 2 \\
&\quad \Pr(M = 3) + \Pr(M = 4) + \Pr(M = 5) + \cdots + && \text{for } m = 3 \\
&\quad \cdots && \cdots \\
&= \Pr(M = 1) + 2\Pr(M = 2) + 3\Pr(M = 3) + 4\Pr(M = 4) + \cdots \\
&= \sum_{m=1}^{\infty} m \Pr(M = m) = \mathrm{E}[M]
\end{aligned}$$

By substitution, if $n := m - 1$, then $\sum_{m=1}^{\infty} \Pr(M \geq m) = \sum_{n=0}^{\infty} \Pr(M \geq n+1) = \sum_{n=0}^{\infty} \Pr(M > n)$ □

SkipList

- Expected search cost
  - Hence, we have:

$$\mathrm{E}[M] = \sum_{m=0}^{\infty} \Pr(M > m) \leq \left\lceil \log_{\frac{1}{P}}(n) \right\rceil + n \sum_{m=\left\lceil \log_{\frac{1}{P}}(n) \right\rceil}^{\infty} P^m = \left\lceil \log_{\frac{1}{P}}(n) \right\rceil + \frac{n\, P^{\left\lceil \log_{\frac{1}{P}}(n) \right\rceil}}{1-P}$$

$$\leq \log_{\frac{1}{P}}(n+1) + \frac{n\, P^{\log_{\frac{1}{P}}(n+1)}}{1-P} = \log_{\frac{1}{P}}(n+1) + \frac{n \cdot \frac{1}{n+1}}{1-P} < \log_{\frac{1}{P}}(n+1) + \frac{1}{1-P}$$

$\left\lceil \log_{\frac{1}{P}} n \right\rceil \leq \log_{\frac{1}{P}}(n+1)$ — $P^{\log_{\frac{1}{P}}(n+1)} = \frac{1}{n+1}$ — $\frac{n}{n+1} < 1$

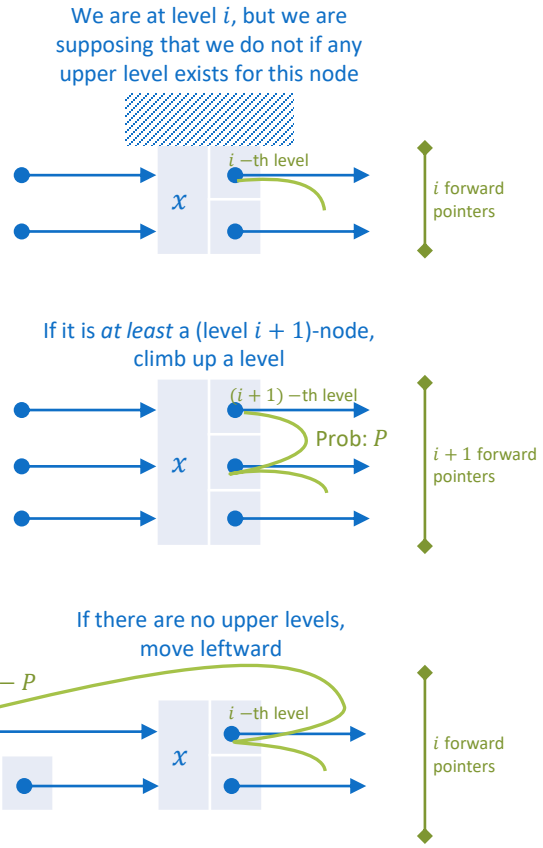
  - $\mathrm{E}[M]$ is the average maximum number of levels
    - **The average "vertical cost" is bounded**:
      - $\mathrm{E}\left[\mathrm{C_u}[l]\right] \leq \mathrm{E}\left[\frac{M-1}{P}\right] = \frac{\mathrm{E}[M]-1}{P} < \frac{\log_{\frac{1}{P}}(n+1)+\frac{P}{1-P}}{P} = \frac{1}{P}\log_{\frac{1}{P}}(n+1) + \frac{1}{1-P}\ ,\ \forall l \in \{0, 1, \ldots, M-1\}$

SkipList

- Expected search cost
  - The **average "horizontal cost"** must be **bounded**
    - In the probabilistic analysis, **starting from the searched** (and found) node
    - and **travelling up and to the left**
    - the average "horizontal cost" is the number of nodes traversed
      - from the (level M)-node
        - reached while climbing vertically, with cost $C_u[l]$
      - to the header

SkipList

- Expected search cost
  - The **average "horizontal cost"** must be **bounded**
  - Estimate the average numbers of nodes traversed **without changing level**
    - Suppose to be at a generic level $i \in \mathbb{N}_{\geq 0}$ of node x
      - x is neither the header nor the tail of the skip list
    - According to our probabilistic model, we can travel:
      - **One level up**
        - with probability P
          - it is the probabilty for the node to have one upper level
        - remamining at the **same node** x
      - **One position leftward**
        - with probability $1 - P$
        - remamining at the **same level**
        - supposing the list extends infinitely to the left



SkipList

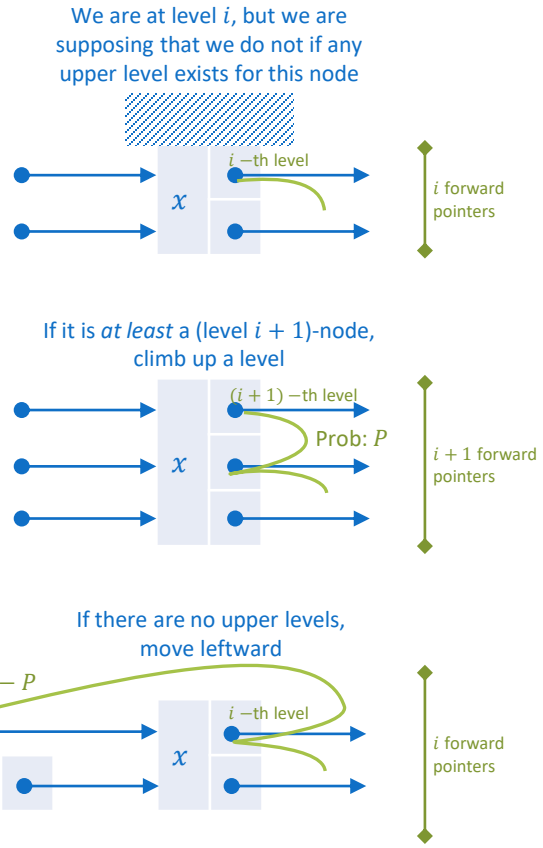
- Expected search cost
  - The **average "horizontal cost"** must be **bounded**
  - Estimate the average numbers of nodes traversed **without changing level**
  - Let B the random variable representing the number of nodes traversed backwards **without changing level**
    - $\Pr(B = b)$ is the probability to traverse b nodes without changing level
    - B is a geometric random variable

$$\Pr(B = b) = (1 - P)^b \cdot P\,, \qquad b \in \mathbb{N}_{\geq 0}$$

Pessimistic hypothesis: the list extends infinitely to the left. A more realistic bound might be $b \in \{0,1,\dots,n\}$

$$\mathrm{E}[B] = \sum_{b=1}^{\infty} b\,(1-P)^b\,P = \frac{1-P}{P}$$

Proof in the next slide



SkipList

- Expected search cost

$$\sum_{b=1}^{\infty} b\,(1-P)^b\,P = \frac{1-P}{P}$$

Proof

Since the geometric series ($\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, $\forall x \in (0,1)$) converges uniformly, we can calculate the derivative to both sides

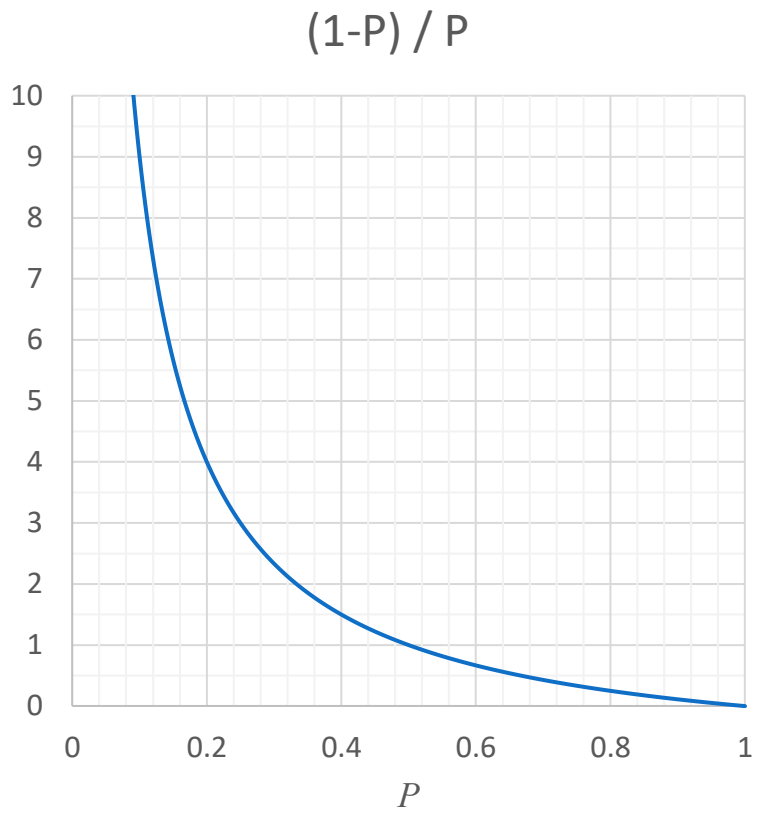
$$\frac{\mathrm{d}}{\mathrm{d}x}\sum_{i=0}^{\infty} x^i = \frac{\mathrm{d}}{\mathrm{d}x}\frac{1}{1-x} \Leftrightarrow \sum_{i=0}^{\infty} i\,x^{i-1} = \frac{1}{(1-x)^2}$$

$$\Rightarrow \sum_{i=0}^{\infty} i\,x^{i-1} = \frac{1}{x}\sum_{i=0}^{\infty} i\,x^i = \frac{1}{(1-x)^2} \Leftrightarrow \sum_{i=0}^{\infty} i\,x^i = \frac{x}{(1-x)^2}$$

In particular, for $x = 1-P$:

$$\sum_{b=1}^{\infty} b\,(1-P)^b\,P = \frac{1-P}{P^2}\cdot P = \frac{1-P}{P}$$

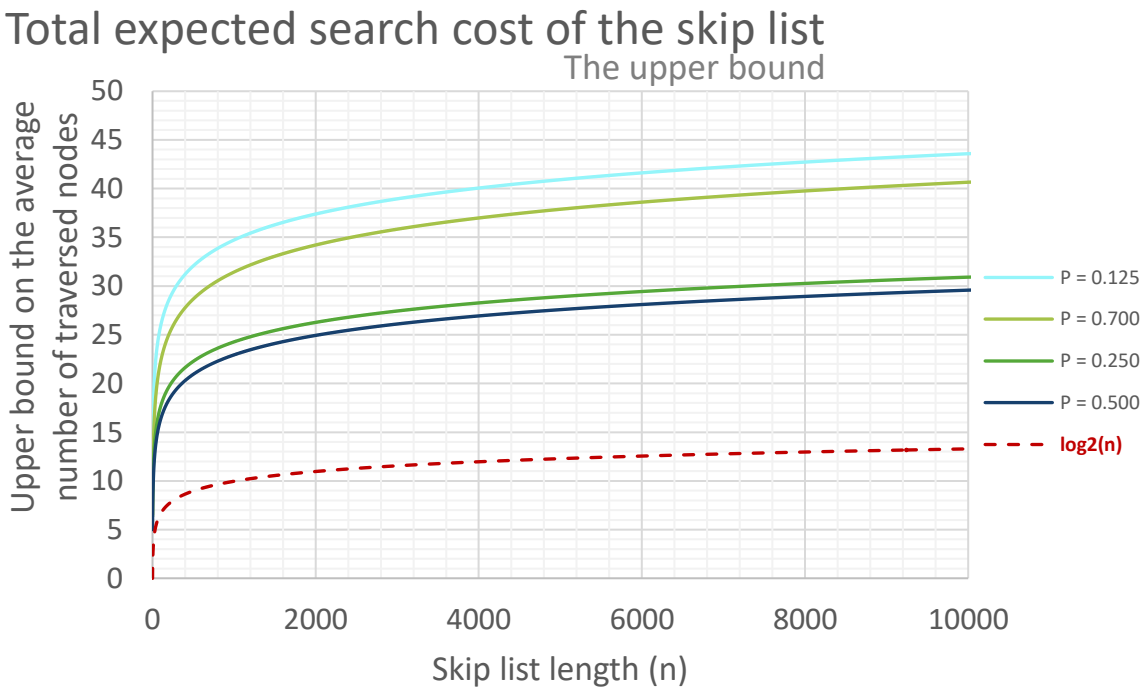
□



SkipList

- Expected search cost
  - We have found:
    - $\mathrm{E}\left[\mathrm{C_u}[l]\right] < \frac{1}{P}\log_{\frac{1}{P}}(n+1) + \frac{1}{1-P}\ ,\ \forall l \in \{0, 1, \dots, M-1\}$
    - $\mathrm{E}[\mathrm{C_b}(y^*)] \leq \frac{1-P}{P}$
      - y^* is the (level M)-node reached by climbing the skip list from the searched and found node
  - The **total expected search cost** is $\mathcal{O}(\log n)$
    - n is the total number of elements in the skip list

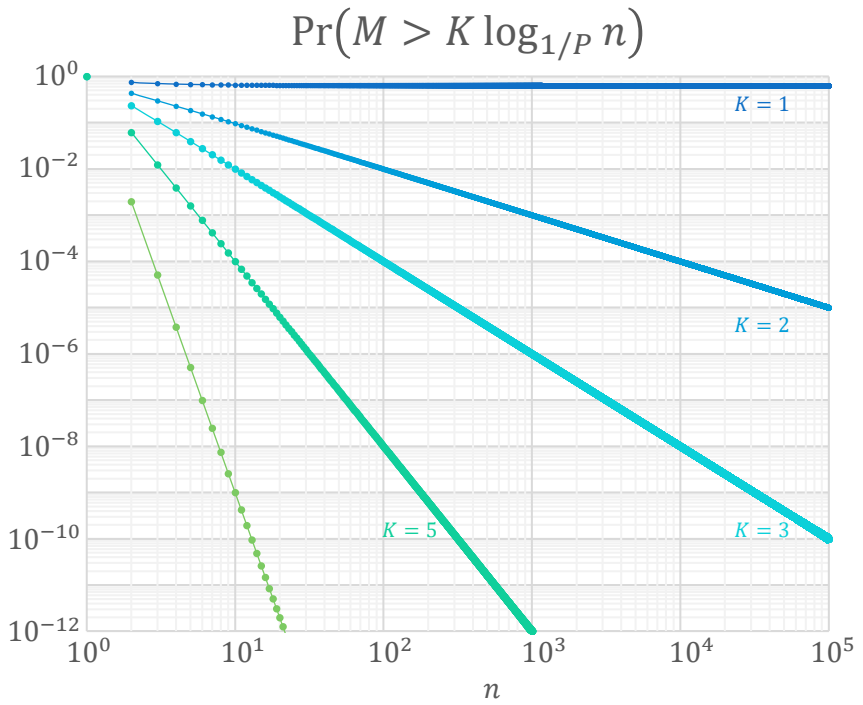
$$\mathrm{E}\left[\mathrm{C_u}[l]\right] + \mathrm{E}[\mathrm{C_b}\ (y^*)] < \frac{1}{P}\log_{\frac{1}{P}}(n+1) + \frac{1}{1-P} + \frac{1-P}{P}$$



SkipList

- Choosing the value for *MaxListLevel*
  - Levels can theoretically grow from $i = 1$ to infinity
    - In practice they must be limited, by *MaxListLevel*
    - Level $i = 0$ means that there are 0 levels above the first one
  - Recall: $M = \max_{j \in \{1,..,n\}} L_j$
    - is the **random variable** representing the **height of the list**
    - $\Pr(M > m) = 1 - (1 - P^m)^n$
  - the default value chosen for *MaxListLevel* is $3 \log_{1/P} n$
    - $\Pr(M > m)|_{m=K \log_{1/P} n} = 1 - \left(1 - P^{K \log_{1/P} n}\right)^n$

$$= 1 - \left(1 - \frac{1}{n^K}\right)^n \rightarrow_{n \rightarrow \infty} 0$$



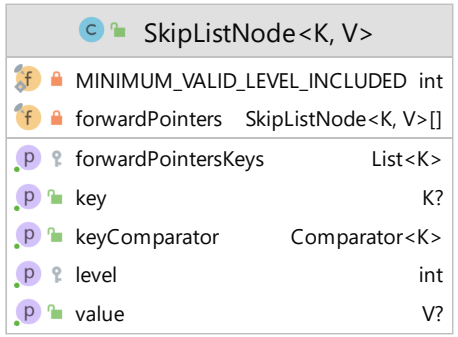
SkipList

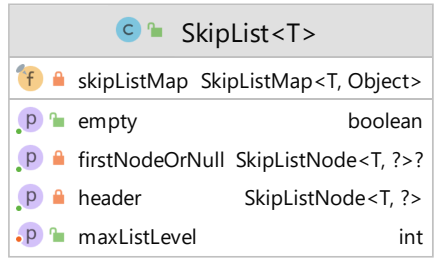
Summary

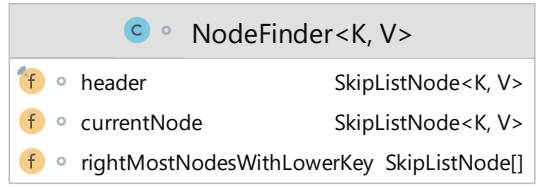
- Expected search cost
  - $\mathcal{O}(\log n)$
- Expected insertion cost
  - $\mathcal{O}(\log n)$
  - We have to find the position where to add the new element and then update pointers
- Expected deletion cost
  - $\mathcal{O}(\log n)$
  - We have to find the position where to remove the element from and then update pointers
- Expected memory usage
  - $\mathcal{O}(n)$

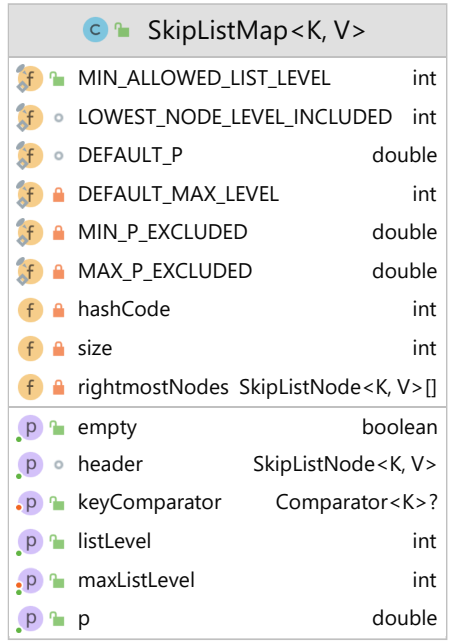
SkipList

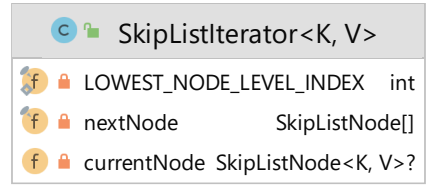
- Details and probabilistic analysis can be found at
  - W. Pugh, “Skip Lists: A Probabilistic alternative to balanced trees”, *Communications of the ACM*, Vol. 33, Number 6, June 1990.











Data structures comparison

- Expected costs of the *SkipList*

	Concurrent HashMap	Patricia Trie	Skip list
Search by key	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log n)$
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log n)$
Pros	Fast access by key; thread safe	Scan by prefix (useful for permuterm index), no collisions	Forward pointers for fast scanning of the list
Cons	Cannot scan by prefix; in the *worst* case the cost is $\mathcal{O}(n)$ due to collisions; might require re-hashing	m accesses required	Worst cases: $\mathcal{O}(n)$ for search/insertion/deletion, and $\mathcal{O}(n \log n)$ for used space

m is the size of the key (in tries)
n is the number of elements in the data structure

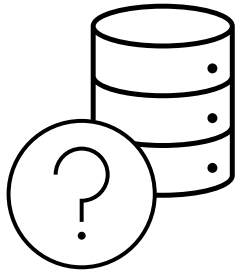
Data structures

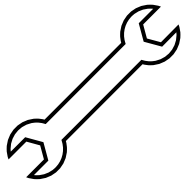
END

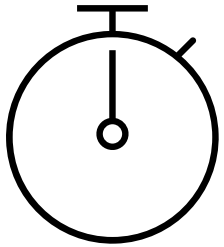
Queries

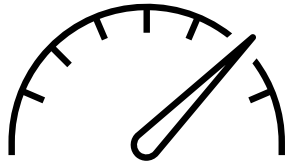
Queries

- Structure
- Parsing
- Optimization and simplification
- Evaluation
- Ranking
- Wildcard and phrase queries
- Spelling correction





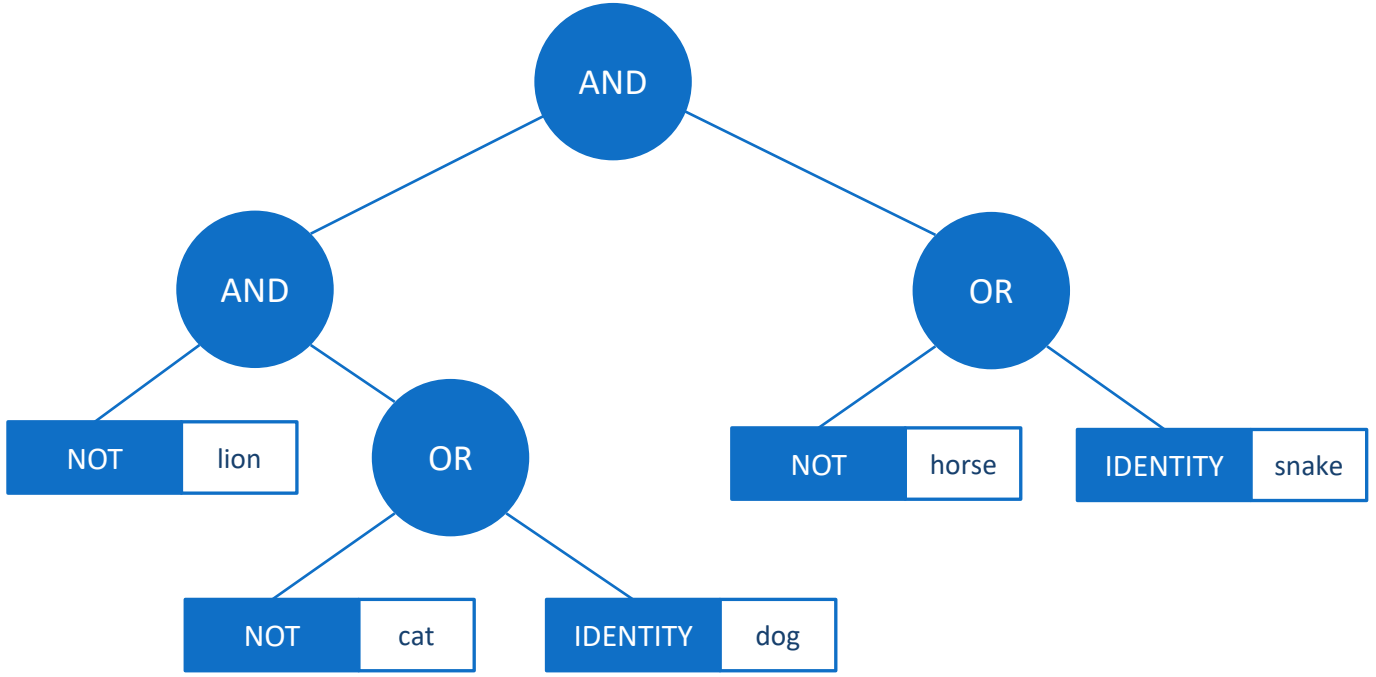




Structure

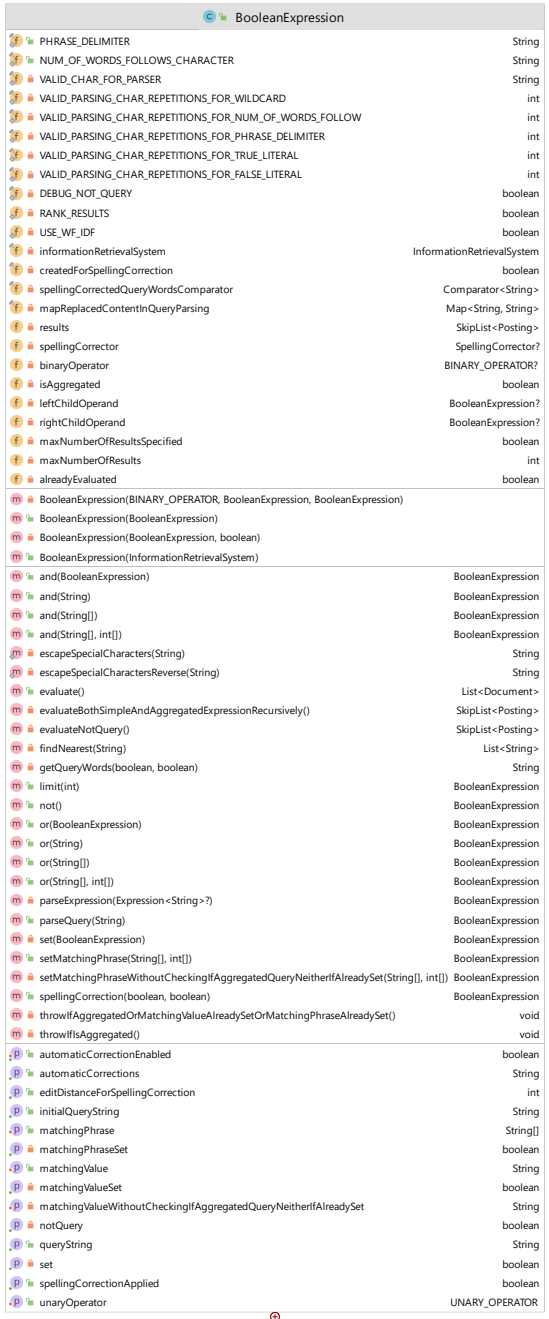
- Hierarchical structure
  - Based on
    - **Binary operations**
      - AND
      - OR
    - **Unary operations**
      - IDENTITY
      - NOT
  - Allows to use
    - recursive algorithms for query evaluation
    - parallelization

Hierarchical representation for query
`((NOT lion) AND ((NOT cat) OR dog)) AND ((NOT horse) OR snake)`



Structure

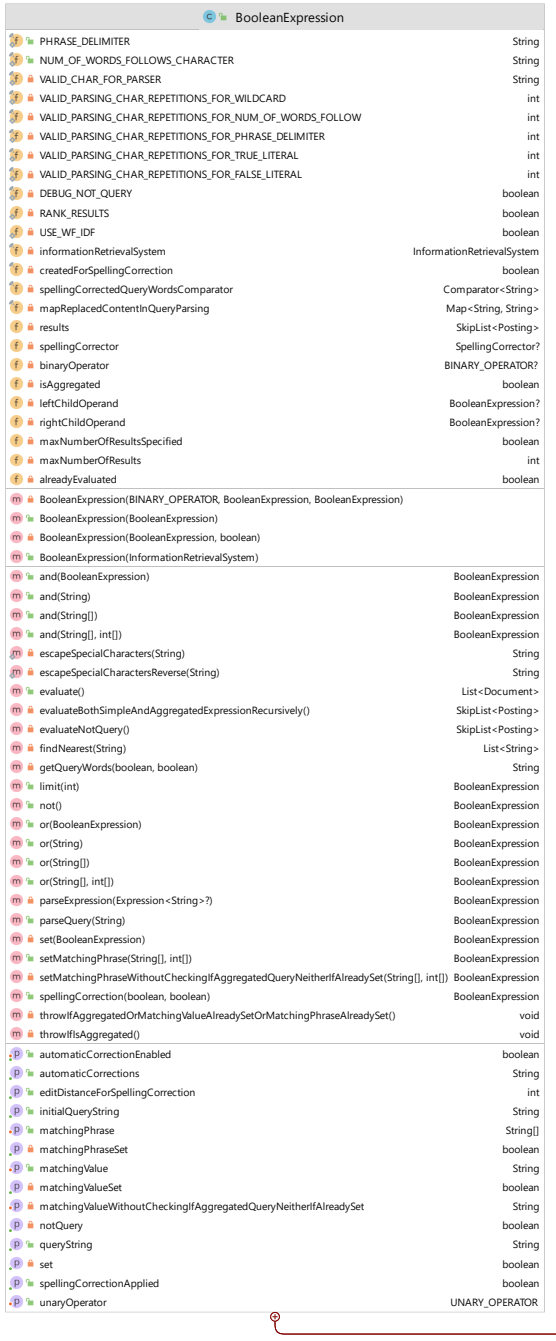
- Implementation
  - very complex
  - does much more than “simply” boolean queries evaluation
    - Interface with the IR system
      - *Inverted* index returns the *docID* of a document
      - *Corpus* returns a document from its *docID*
    - efficiency
    - phrase queries
    - wildcard queries
    - spelling correction
    - phonetic correction
    - ranking of results
    - parsing of queries inserted as text
    - consistency of the queries (saved hierarchically)



Structure

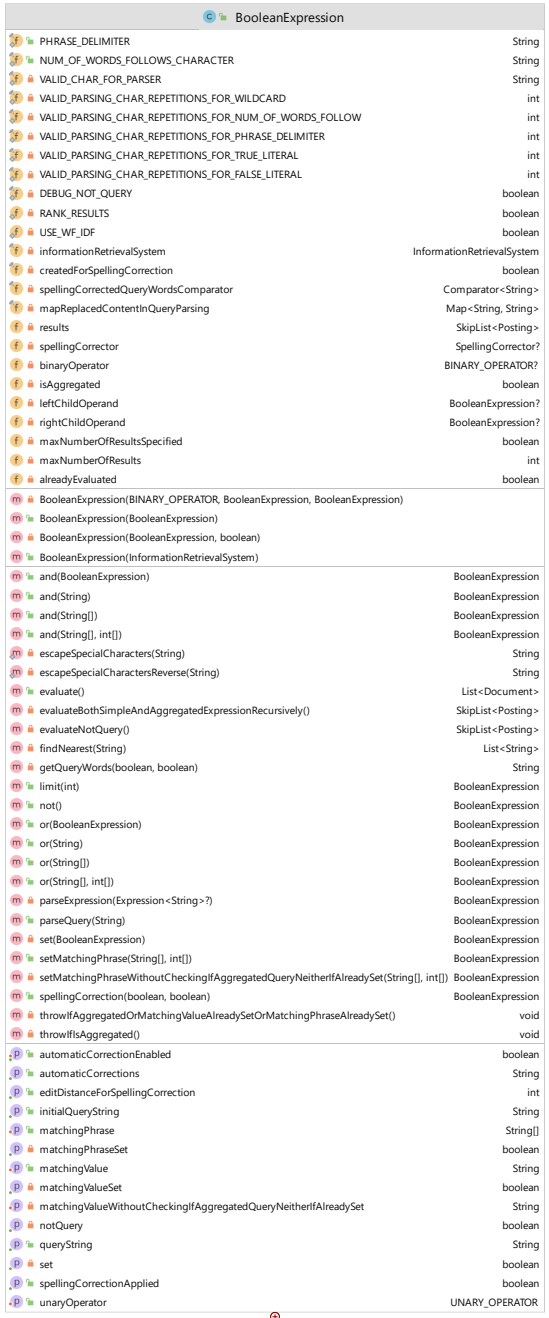
- Implementation
  - Important fields

```
@NotNull  private String queryString = "";
@Nullable private BINARY_OPERATOR binaryOperator;    // null if this expression is not aggregated
@NotNull  private UNARY_OPERATOR unaryOperator = UNARY_OPERATOR.IDENTITY; // default is the identity operator
private boolean isAggregated;
@Nullable private BooleanExpression leftChildOperand;  // null if this expression is not aggregated
@Nullable private BooleanExpression rightChildOperand; // null if this expression is not aggregated
@Nullable private String matchingValue;                // not null only if there is a matching value
@Nullable private Phrase matchingPhrase;               // not null only if there is a matching phrase
```



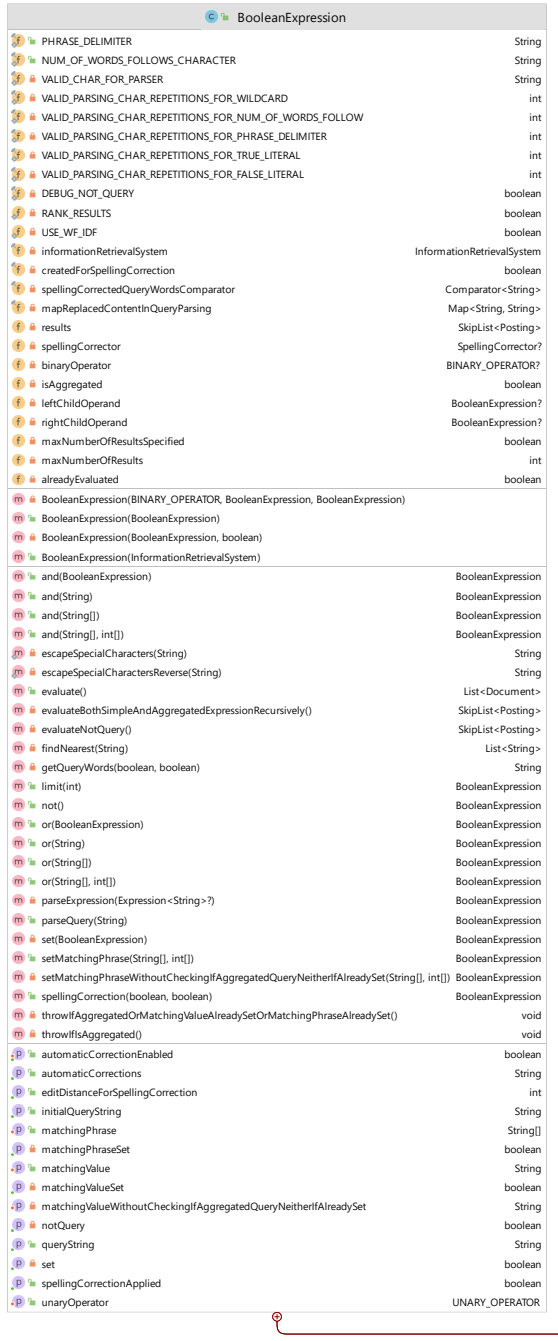
Structure

- Implementation
  - Important fields
    - public static final String *PHRASE_DELIMITER* = "\"";
      - to recognize double quotes delimiting a phrase (for phrase queries)
    - public static final String *NUM_OF_WORDS_FOLLOWS_CHARACTER* = "/";
      - to specify the number of words that must be present between the two words, e.g. `The/0cat/1on` might stands for `The cat is on`
    - private static boolean *RANK_RESULTS*;
      - to specify if results must be ranked
    - private static boolean *USE_WF_IDF*;
      - to specify if *wf-idf* weighting must be used (for ranking)
    - private final InformationRetrievalSystem informationRetrievalSystem;
      - to interface with the IR system, get the *docID*s and retrieve the documents
    - private final boolean createdForSpellingCorrection;
      - some instances are created by the recursive algorithms for spelling corrections
      - the flag specifies that the instance must not be selling-corrected
        - (if it has been created as consequence of another spelling correction)
    - private final Comparator<String> spellingCorrectedQueryWordsComparator;
      - The spelling corrector tries to correct a word by replacing it with the nearest one, according to the **edit distance**, among the words in the dictionary
      - This fields provides the order relationship between candidate words when they have the same edit distance



Structure

- Implementation
  - Important fields
    - private final Map<String, String> mapReplacedContentInQueryParsing = new HashMap<>(10);
      - Content between quotes in a query string represents a phrase query
      - It must not be modified (e.g., by the optimizer)
      - The solution is to replace that content with a particular string which will not be altered for sure by any step of query preprocessing, then to replace it back just before the query evaluation
      - This field maps that kind of replacements
    - private final SkipList<Posting> results = new SkipList<>(new ArrayList<>(), Posting.*DOC_ID_COMPARATOR*);
      - saves the results obtained from the query evaluation
    - private SpellingCorrector spellingCorrector = null;
      - Instantiated for spelling (and phonetic) correction if needed



Structure

- Query hierarchical structure is built with methods
  - not
  - and
  - or
  - setMatchingValue
  - setMatchingPhrase

```
public BooleanExpression not() {
  return isNotQuery() ?
        setUnaryOperator(UNARY_OPERATOR.IDENTITY/*negation of NOT is the identity*/) :
        setUnaryOperator(UNARY_OPERATOR.NOT);
}

public BooleanExpression and(@NotNull BooleanExpression other) {
  return set(isSet()
        ? new BooleanExpression(BINARY_OPERATOR.AND, new BooleanExpression(this), other)
        : other);
}

public BooleanExpression or(@NotNull BooleanExpression other) {
  return set(isSet()
        ? new BooleanExpression(BINARY_OPERATOR.OR, new BooleanExpression(this), other)
        : other);
}
```

Structure

- Query hierarchical structure is built with methods
  - not
  - and
  - or
  - setMatchingValue
  - setMatchingPhrase

```
public BooleanExpression setMatchingValue(@NotNull String matchingValue) {
   throwIfAggregatedOrMatchingValueAlreadySetOrMatchingPhraseAlreadySet();
   return setMatchingValueWithoutCheckingIfAggregatedQueryNeitherIfAlreadySet(matchingValue);
}

private BooleanExpression setMatchingValueWithoutCheckingIfAggregatedQueryNeitherIfAlreadySet(
      @NotNull String matchingValue) {
   // Save the query string inserted by the user, before any normalization is applied
   // This query string will NOT be used for the evaluation but only for toString methods
   this.queryString = matchingValue;
   this.matchingValue = Utility.preprocess(matchingValue, true, informationRetrievalSystem.getLanguage());
   return this;
}

private void throwIfAggregatedOrMatchingValueAlreadySetOrMatchingPhraseAlreadySet() throws IllegalStateException {
   throwIfIsAggregated();
   StringBuilder errorMsg = new StringBuilder();
   if (isMatchingValueSet()) {
      errorMsg.append("Matching value");
   } else if (isMatchingPhraseSet()) {
      errorMsg.append("Matching phrase");
   }
   if (!errorMsg.isEmpty()) {
      throw new IllegalStateException(errorMsg.append(" already set, cannot re-set.").toString());
   }
}
```

Structure

- Query hierarchical structure is built with methods
  - not
  - and
  - or
  - setMatchingValue
  - setMatchingPhrase

```
public BooleanExpression setMatchingPhrase(@NotNull String[] matchingPhrase, int[] matchingPhraseDistances) {
   throwIfAggregatedOrMatchingValueAlreadySetOrMatchingPhraseAlreadySet();
   return setMatchingPhraseWithoutCheckingIfAggregatedQueryNeitherIfAlreadySet(
         matchingPhrase, matchingPhraseDistances);

}
```

Structure

- Query hierarchical structure is built with methods
  - not
  - and
  - or
  - setMatchingValue
  - setMatchingPhrase

```
private BooleanExpression setMatchingPhraseWithoutCheckingIfAggregatedQueryNeitherIfAlreadySet(
    @NotNull String[] matchingPhrase, int[] matchingPhraseDistances)
    throws IllegalArgumentException {

  if (matchingPhrase.length < 1 || matchingPhraseDistances.length != matchingPhrase.length - 1) {
    throw new IllegalArgumentException(
        "The size of the array of distances must be equal to the phrase length minus 1, but it is not:"
            + System.lineSeparator()
            + "\t- phrase length: " + matchingPhrase.length
            + System.lineSeparator()
            + "\t- distances length: " + matchingPhraseDistances.length);
  }

  // Save the query string inserted by the user, before any normalization is applied
  // This query string will NOT be used for the evaluation but only for toString methods
  try {
    queryString = new Phrase(matchingPhrase, matchingPhraseDistances).toString();
  } catch (IllegalArgumentException e) {
    queryString = matchingPhrase[0];
  }

  String[] tmpPhrase = Arrays.stream(matchingPhrase)
      .map(word -> Utility.preprocess(word, true, informationRetrievalSystem.getLanguage()))
      .toArray(String[]::new);   // null elements might be present

  String[] phrase = new String[tmpPhrase.length];
  int[] distances = new int[tmpPhrase.length - 1];

  int wordCounter = 0;
  int nonNullWordsCounter = 0;

  for (; wordCounter < tmpPhrase.length && tmpPhrase[wordCounter] == null; wordCounter++) {
    // find the first non-null word and increment the counter
  }
```

[continues in the next slide …]

Structure

- Query hierarchical structure is built with methods
  - not
  - and
  - or
  - setMatchingValue
  - setMatchingPhrase

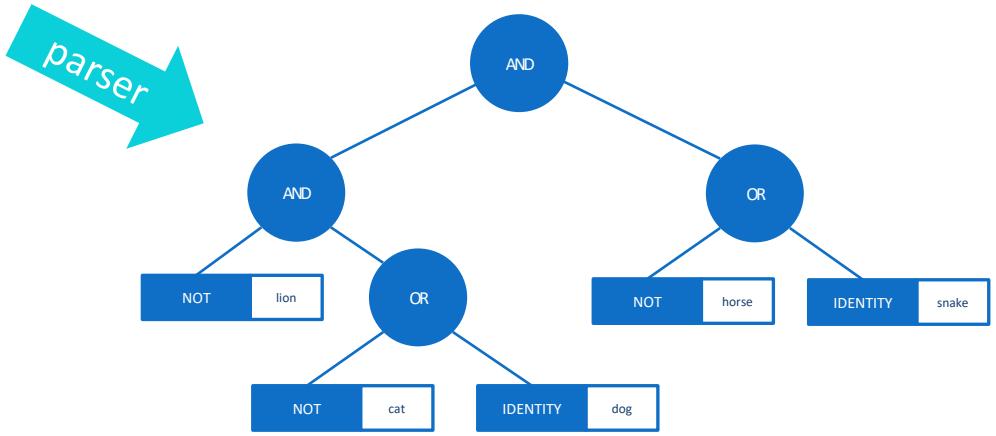
[… continues from the previous slide]

```
if (wordCounter == tmpPhrase.length) { // true if all words are null
   return this;   // do not set anything
} else {
   if (wordCounter > 0) {
      // Update distances due to removed words
      for (int i = 0; i < wordCounter; i++) {
         matchingPhraseDistances[i]--;
      }
   }
   for (; wordCounter < matchingPhraseDistances.length; wordCounter++) {
      if (tmpPhrase[wordCounter] != null) {
         phrase[nonNullWordsCounter] = tmpPhrase[wordCounter];
         distances[nonNullWordsCounter++] = matchingPhraseDistances[wordCounter];
      } else {
         // Update distances due to removed words
         for (int i = wordCounter + 1; i < matchingPhraseDistances.length; i++) {
            matchingPhraseDistances[i]--;  // remove the space occupied by the (one) just removed word
         }
      }
   }
   if (tmpPhrase[wordCounter] != null) {   // copy the last word if non-null
      phrase[nonNullWordsCounter++] = tmpPhrase[wordCounter];
   }
   assert Arrays.stream(distances).filter(val -> val >= 0).count() == distances.length;
   assert distances.length < 2
            || Arrays.stream(distances).sequential().reduce((a, b) -> a < b ? 1 : 0).orElse(1/*if here, array is empty*/) == 1;// check distances are monotonically growing
   assert wordCounter + 1 == tmpPhrase.length;
   assert nonNullWordsCounter >= 1;

   String[] finalPhrase = new String[nonNullWordsCounter];
   int[] finalDistances = new int[nonNullWordsCounter - 1];
   assert finalPhrase.length == Arrays.stream(tmpPhrase).filter(Objects::nonNull).count();
   assert finalDistances.length == finalPhrase.length - 1;

   System.arraycopy(phrase, 0, finalPhrase, 0, nonNullWordsCounter);
   System.arraycopy(distances, 0, finalDistances, 0, nonNullWordsCounter - 1);
   assert finalDistances.length == finalPhrase.length - 1;
   assert Arrays.stream(finalPhrase).filter(Objects::nonNull).count() == finalPhrase.length;
   assert Arrays.stream(finalDistances).filter(val -> val >= 0).count() == finalDistances.length;
   assert finalDistances.length < 2
            || Arrays.stream(finalDistances).sequential().reduce((a, b) -> a < b ? 1 : 0).orElse(1/*if here, array is empty*/) == 1;// check distances are monotonically growing

   if (finalPhrase.length == 1) {
      // normalization may lead to a single word (hence, it is not a phrase anymore)
      return setMatchingValueWithoutCheckingIfAggregatedQueryNeitherIfAlreadySet(finalPhrase[0]);
   } else {
      assert Arrays.stream(finalPhrase).noneMatch(Objects::isNull);
      this.matchingPhrase = new Phrase(finalPhrase, finalDistances);
      return this;
   }
}
}
```

Parsing

- Queries can be inserted as plain text
  - They will be parsed to create the hierarchical structure

```
(!lion & (!cat | dog)) & (!horse | snake)
```




```
public BooleanExpression parseQuery(@NotNull String queryString)
      throws IllegalArgumentException, IllegalStateException {

   Objects.requireNonNull(queryString);
   throwIfAggregatedOrMatchingValueAlreadySetOrMatchingPhraseAlreadySet();

   queryString = escapeSpecialCharacters(queryString);

   // Handle content between quotes in query string (it must not be modified)
   int counter = 0;
   final Pattern BETWEEN_QUOTES_REGEX = Pattern.compile("[\"][^\"]*[\"]");
   Matcher betweenQuotesMatcher = BETWEEN_QUOTES_REGEX.matcher(queryString);
   while (betweenQuotesMatcher.find()) {
      String match = betweenQuotesMatcher.group();
      final String replacementDelimiter = Utility.getNReplicationsOfString(
            VALID_PARSING_CHAR_REPETITIONS_FOR_PHRASE_DELIMITER, VALID_CHAR_FOR_PARSER);
      String replacement = replacementDelimiter + (counter++) + replacementDelimiter;
      mapReplacedContentInQueryParsing.put(replacement, match);
      queryString = queryString.replaceAll(match, replacement);
   }

   return set(parseExpression(QueryParsing.parse(queryString)));
}
```

Parsing

```
private static String escapeSpecialCharacters(@NotNull String queryString) {
   // escaping is done by replicating a valid char for the parser

   return queryString
         .replaceAll(VALID_CHAR_FOR_PARSER + "+", VALID_CHAR_FOR_PARSER) // remove duplicates
         .replaceAll(    // escape wildcard
               ESCAPED_WILDCARD_FOR_REGEX,
               Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_WILDCARD, VALID_CHAR_FOR_PARSER))

         // escape other used special characters
         .replaceAll(
               NUM_OF_WORDS_FOLLOWS_CHARACTER,
               Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_NUM_OF_WORDS_FOLLOW, VALID_CHAR_FOR_PARSER))

         // words with logical meaning are evaluated like literal expressions by the parser, hence they need kind of escaping
         .replaceAll(
               "true",
               Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_TRUE_LITERAL, VALID_CHAR_FOR_PARSER))
         .replaceAll(
               "false",
               Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_FALSE_LITERAL, VALID_CHAR_FOR_PARSER));

}

@NotNull
private static String escapeSpecialCharactersReverse(@NotNull String queryString) {

   //noinspection ConstantConditions // order in which replacement operations are made matters and this assertion checks if the value is changed for some reasons
   assert VALID_PARSING_CHAR_REPETITIONS_FOR_NUM_OF_WORDS_FOLLOW > VALID_PARSING_CHAR_REPETITIONS_FOR_WILDCARD;
   return queryString
         // replacements must be done in order: longest escaping sequence first
         .replaceAll(Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_FALSE_LITERAL, VALID_CHAR_FOR_PARSER), "false")
         .replaceAll(Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_TRUE_LITERAL, VALID_CHAR_FOR_PARSER), "true")
         .replaceAll(Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_NUM_OF_WORDS_FOLLOW, VALID_CHAR_FOR_PARSER), NUM_OF_WORDS_FOLLOWS_CHARACTER)
         .replaceAll(Utility.getNReplicationsOfString(VALID_PARSING_CHAR_REPETITIONS_FOR_WILDCARD, VALID_CHAR_FOR_PARSER), WILDCARD);
}
```

Parsing

```
/**
 * This method translates an instance of {@link Expression} to an instance of
 * this class.
 *
 * @param expression The {@link Expression} to be translated.
 * @return The resulting instance of this class obtained from the input parameter.
 */
@NotNull
private BooleanExpression parseExpression(@Nullable final Expression<String> expression) {
   if (expression == null) {  // invalid query string
      return this;
   } else {

      if (expression instanceof Variable variable) {

         String queryString = (String) variable.getValue();
         {
            // Handle the inverse operations for the escaping done for the parser
            for (Map.Entry<String, String> entry : mapReplacedContentInQueryParsing.entrySet()) {
               String replacement = entry.getKey();
               String initialValue = entry.getValue();
               queryString = queryString.replaceAll(replacement, initialValue);
            }
            queryString = escapeSpecialCharactersReverse(queryString);
         }

         final String REGEX_MATCHING_DELIMITED_PHRASE =
               "[" + PHRASE_DELIMITER + "][^" + PHRASE_DELIMITER + "]*[" + PHRASE_DELIMITER + "]";
         Matcher phraseMatcher = Pattern.compile(REGEX_MATCHING_DELIMITED_PHRASE).matcher(queryString);

         // new instance needed
         BooleanExpression be = new BooleanExpression(informationRetrievalSystem);

         if (phraseMatcher.matches()) {
            // phrase is present
            String phrase = phraseMatcher.group()
                  .replaceAll(PHRASE_DELIMITER, "").strip();
            String[] words;
            int[] distancesOfIthWordFromTheFirstWord = null;
            final String REGEX_SEARCHING_INTERMEDIATE_NUM_OF_WORDS =
                  "\\" + NUM_OF_WORDS_FOLLOWS_CHARACTER + "\\d+";
            if (Pattern.compile(REGEX_SEARCHING_INTERMEDIATE_NUM_OF_WORDS).matcher(phrase).find()) {
               phrase = phrase.replaceAll("\\s+(?!\\d+)", NUM_OF_WORDS_FOLLOWS_CHARACTER + "0");
                        // replace spaces with number of (0) intermediate words if not specified

               Matcher intermediateNumOfWordsMatcher =
                     Pattern.compile(REGEX_SEARCHING_INTERMEDIATE_NUM_OF_WORDS).matcher(phrase);
               words = phrase.split(REGEX_SEARCHING_INTERMEDIATE_NUM_OF_WORDS);
               int[] intermediateNumOfWords = new int[words.length - 1];
               for (int i = 0; i < intermediateNumOfWords.length && intermediateNumOfWordsMatcher.find(); i++) {
                  intermediateNumOfWords[i] = Integer.parseInt(
                        intermediateNumOfWordsMatcher.group()
                              .replaceAll("\\" + NUM_OF_WORDS_FOLLOWS_CHARACTER, ""));
               }
               distancesOfIthWordFromTheFirstWord = new int[intermediateNumOfWords.length];
               int cumulativeDistance = 0;
               for (int i = 0; i < distancesOfIthWordFromTheFirstWord.length; i++) {
                  cumulativeDistance += intermediateNumOfWords[i] + 1; // each word is at least one (+1) word ahead than the previous one
                  distancesOfIthWordFromTheFirstWord[i] = cumulativeDistance;
               }
            } else {
               // phrase does not specify the number of words which must be present between words composing the phrase itself
               words = Utility.split(phrase);
            }
            if (distancesOfIthWordFromTheFirstWord == null) {
               be.setMatchingPhrase(words);
            } else {
               be.setMatchingPhrase(words, distancesOfIthWordFromTheFirstWord);
            }
         } else {
            be.setMatchingValue(queryString);
         }

         return be;
      } else if (expression instanceof Not negation) {
         return parseExpression(negation.getE()).setUnaryOperator(UNARY_OPERATOR.NOT);
      } else if (expression instanceof Literal literal) {
         if (literal.getValue()) {
            System.err.println("Input query evaluated to literal true, this means that" +
                  " all indexed documents should be retrieved during the evaluation of" +
                  " this instance, but no results will be returned. Please, reformulate" +
                  " your query.");
         }

         // evaluation will return nothing (either if query expression evaluated true or false)
         return informationRetrievalSystem.createNewBooleanExpression();
      } else {
         // binary expression

         BinaryOperator<BooleanExpression> accumulator;
         if (expression instanceof And) {
            accumulator = BooleanExpression::and;
         } else if (expression instanceof Or) {
            accumulator = BooleanExpression::or;
         } else {
            throw new IllegalStateException("Unknown binary operation in expression " + expression);
         }
         return expression.getChildren()
               .stream()
               .map(this::parseExpression)
               .reduce(accumulator)
               .orElseThrow();
      }
   }
}
```

Parsing

- Class *Query Parsing*
  - Translates a textual query to *BooleanQuery*
    - a & b
    - a | b
    - ! a
  - spaces are not mandatory
  - Parenthesis impose priorities
    - (a|b)&c

```
class QueryParsing {

   static Expression<String> parse(@NotNull String queryString) {
      queryString = removeInvalidCharacters(queryString);
      queryString = replaceSpacesWithAndOperator(queryString);
      queryString = removeDuplicatedAdjacentBinaryOperators(queryString);
      if (queryString.isBlank()) {
         System.err.println("The query string is blank.");
         return null;
      }
      try {                          // Parse expression
         var parsedExpression = ExprParser.parse(queryString);

         // Simplify and convert the expression to a "sum of products"
         //  i.e., AND operations will have the priority, then the OR of results will be evaluated
         // This can be very helpful for query optimization, because performing AND queries
         //  first reduces sizes of intermediate results
         return RuleSet.toDNF(parsedExpression);  // Disjunctive Normal (Sum-Of-Products) Form
      } catch (RuntimeException e) {
         // invalid input
         return null;
      }
   }

   private static String removeDuplicatedAdjacentBinaryOperators(String queryString) {
      // e.g., "a &&& c" becomes "a & c"
      for (var bo : BINARY_OPERATOR.values()) {
         queryString = queryString
               .replaceAll("\\" + bo.getSymbol() + "{2,}", "\\" + bo.getSymbol());
      }
      return queryString;
   }

   private static String replaceSpacesWithAndOperator(@NotNull String queryString) {
      queryString = queryString.strip();
      final String LOOK_AHEAD_REGEX =
            "(?=(" + "[^\\" + PHRASE_DELIMITER + "]*\\" + PHRASE_DELIMITER
                  + "[^\\" + PHRASE_DELIMITER + "]*\\" + PHRASE_DELIMITER + ")*"
                  + "[^\\" + PHRASE_DELIMITER + "]*$)";
      final Pattern REGEX_REPLACE_SPACES_WITH_AND_OPERATOR = Pattern.compile(
            "\\s*(\\" + NOT.getSymbol() + "*\\w+\\s+)+\\" + NOT.getSymbol() + "*\\w+\\s*" // match substring like {  a b  c }
                  + LOOK_AHEAD_REGEX); // matched string must not be between phrase delimiters (i.e., neither {" d"} nor {"a b"} nor similar should match)
      Matcher m = REGEX_REPLACE_SPACES_WITH_AND_OPERATOR.matcher(queryString);
      while (m.find()) {
         queryString =
               queryString.replaceAll(m.group(), m.group().replaceAll("\\s+", AND.getSymbol()));
      }
      queryString = queryString                                  // Latest refinements
            .replaceAll("\\s+" + LOOK_AHEAD_REGEX, "")           // remove remaining spaces not inside phrase delimiters
            .replaceAll(NOT.getSymbol() + AND.getSymbol(), NOT.getSymbol()); // remove erroneously added AND symbols (not correctly matching in previous regex)
      return queryString;
   }
```

Parsing

- Class *Query Parsing*
  - Translates a textual query to *BooleanQuery*
    - a & b
    - a | b
    - ! a
  - spaces are not mandatory
  - Parenthesis impose priorities
    - (a|b)&c

```
@NotNull
private static String removeInvalidCharacters(@NotNull String queryString) {
   // Check if present and eventually remove invalid characters

   final String REGEX_INVALID_INPUT_CHARACTERS = "[^\\w\\s\"'&|!()]";
   Matcher invalidCharacterMatcher = Pattern.compile(REGEX_INVALID_INPUT_CHARACTERS).matcher(queryString);
   if (invalidCharacterMatcher.find()) {
      invalidCharacterMatcher.reset();
      final List<String> FOUND_INVALID_CHARACTERS =
            invalidCharacterMatcher.results().map(MatchResult::group).distinct().toList();
      final boolean MORE_THAN_1_INVALID_CHAR = FOUND_INVALID_CHARACTERS.size() > 1;
      final String CLEANED_QUERY_STRING = queryString.replaceAll(REGEX_INVALID_INPUT_CHARACTERS, "");
      System.err.println("Invalid character"
            + (MORE_THAN_1_INVALID_CHAR ? "s" : "")
            + " found {" + String.join(", ", FOUND_INVALID_CHARACTERS) + "}: "
            + (MORE_THAN_1_INVALID_CHAR ? "they" : "it")
            + " will be removed from the input query string " +
            "({" + queryString + "} becomes {" + CLEANED_QUERY_STRING + "}).");
      queryString = CLEANED_QUERY_STRING;
   }

   return queryString;
}

@NotNull
public static String toString(@Nullable Expression<?> expression) {
   if (expression == null) {
      return "";
   } else {
      String toString = expression.toString();
      for (var bo : BINARY_OPERATOR.values()) {
         toString = toString.replaceAll("\\" + bo.getSymbol(), bo.toString());
      }
      return toString.replaceAll("\\" + NOT.getSymbol(), "  " + NOT + " ");
   }
}

}
```

Optimization and simplification

- The class used for query parsing also tries to simplify expressions
  - `(false & A)` = `false`
  - `(true & A)` = `A`
  - `(false | A)` = `A`
  - `(true | A)` = `true`
  - `(!!A )` = `A`
  - `(A & !A)` = `false`
  - `(A | !A)` = `true`
  - `(A & A & (B & C))` = `(A & B & C)`
  - `(A | A | (B | C))` = `(A | B | C)`
- Further optimizations are made during the query evaluation
  - Optimization for NOT queries:
    - `NOT(A) AND B = B \ A`
    - `A AND NOT(B) = A \ B`
  - Motivation: the set difference (`A\B` or `B\A`, respectively) are faster to compute
    - Computing `NOT(A)`, or `NOT(B)`, among *all* postings in the system is not efficient

Evaluation

- Query evaluation is made mainly by the following methods of class *BooleanQuery*
  - public List<Document> evaluate() throws UnsupportedOperationException
  - private SkipList<Posting> evaluateBothSimpleAndAggregatedExpressionRecursively()
    throws UnsupportedOperationException
  - private SkipList<Posting> evaluateNotQuery()

Evaluation

```
private SkipList<Posting> evaluateBothSimpleAndAggregatedExpressionRecursively()
        throws UnsupportedOperationException {

    alreadyEvaluated = true;
    results = switch (unaryOperator) {
        case NOT -> evaluateNotQuery();
        case IDENTITY -> {
            if (isAggregated) {

                assert leftChildOperand != null;
                assert rightChildOperand != null;
                assert binaryOperator != null;
                yield switch (binaryOperator) {
                    case AND -> {

                        // Optimization for NOT queries:
                        //     NOT(A) AND B = B \ A
                        //     A AND NOT(B) = A \ B

                        if (leftChildOperand.unaryOperator.equals(UNARY_OPERATOR.NOT)) {         // NOT(A) AND B = B \ A
                            SkipList<Posting> fromRightChild =
                                    rightChildOperand.evaluateBothSimpleAndAggregatedExpressionRecursively();
                            SkipList<Posting> fromLeftChildNotNegated =
                                    new BooleanExpression(leftChildOperand)
                                            .setUnaryOperator(UNARY_OPERATOR.IDENTITY)
                                            .evaluateBothSimpleAndAggregatedExpressionRecursively();
                            yield SkipList.difference(fromRightChild, fromLeftChildNotNegated, Posting.DOC_ID_COMPARATOR);
                        } else if (rightChildOperand.unaryOperator.equals(UNARY_OPERATOR.NOT)) { // A AND NOT(B) = A \ B
                            SkipList<Posting> fromLeftChild =
                                    leftChildOperand.evaluateBothSimpleAndAggregatedExpressionRecursively();
                            SkipList<Posting> fromRightChildNotNegated =
                                    new BooleanExpression(rightChildOperand)
                                            .setUnaryOperator(UNARY_OPERATOR.IDENTITY)
                                            .evaluateBothSimpleAndAggregatedExpressionRecursively();
                            yield SkipList.difference(fromLeftChild, fromRightChildNotNegated, Posting.DOC_ID_COMPARATOR);
                        } else {
                            SkipList<Posting> fromLeftChild = leftChildOperand.evaluateBothSimpleAndAggregatedExpressionRecursively();
                            SkipList<Posting> fromRightChild = rightChildOperand.evaluateBothSimpleAndAggregatedExpressionRecursively();
                            yield Utility.intersection(fromLeftChild, fromRightChild, Posting.DOC_ID_COMPARATOR);
                        }
                    }
                    case OR -> {
                        SkipList<Posting> fromLeftChild = leftChildOperand.evaluateBothSimpleAndAggregatedExpressionRecursively();
                        SkipList<Posting> fromRightChild = rightChildOperand.evaluateBothSimpleAndAggregatedExpressionRecursively();
                        yield Utility.union(fromLeftChild, fromRightChild, Posting.DOC_ID_COMPARATOR);
                    }
                    default -> throw new UnsupportedOperationException("Unknown operator");
                };
```

Evaluation

```
} else {
   SkipList<Posting> tmpResults;

   if (isMatchingPhraseSet()) {
       BiPredicate<Posting, Posting>[] biPredicatesForCheckingPositionsForPhrasalQueries =
               IntStream.range(0, matchingPhrase.size() - 1)
                       .mapToObj(i -> (BiPredicate<Posting, Posting>)
                               (Posting posting1, Posting posting2) -> {
                                   // Note: order of input arg is important

                                   var positions1 = posting1.getTermPositionsInTheDocument();
                                   var positions2 = posting2.getTermPositionsInTheDocument();

                                   // assert positions were sorted
                                   assert Arrays.stream(positions1).sorted().boxed().toList().equals(Arrays.stream(positions1).boxed().toList());
                                   assert Arrays.stream(positions2).sorted().boxed().toList().equals(Arrays.stream(positions2).boxed().toList());

                                   int index1 = 0, index2 = 0;
                                   while (index1 < positions1.length && index2 < positions2.length) {
                                       int numberOfTermsBetweenWords =
                                               positions2[index2] - positions1[index1] - matchingPhrase.distanceFromFirstWord[i];   // exact distance allowed between *first* word and the word of posting 2
                                       if (numberOfTermsBetweenWords == 0) {
                                           // words are adjacent
                                           return true;
                                       } else if (numberOfTermsBetweenWords < 0) {
                                           // word from posting2 is present in the document before the word of posting1
                                           index2++;
                                       } else {
                                           // word from posting1 is present in the document before the word of posting2
                                           index1++;
                                       }
                                   }
                                   return false;   // no adjacency found
                               })
                       .toArray(BiPredicate[]::new);

       assert matchingPhrase.words.length >= 2;

       Map<String, SkipList<Posting>> cachedPostings =  // in phrases, common words (like articles) might be present more than once (if they were not excluded by normalization steps)
                                                        // and it would not be efficient to retrieve the corresponding posting list each time
               new ConcurrentHashMap<>(matchingPhrase.size());
```

Evaluation

```
IntFunction<@NotNull SkipList<Posting>> getPostingListOfIthWordInPhrase = i -> {
            assert i >= 0 && i < matchingPhrase.size();
            String word = matchingPhrase.words[i];
            assert word != null;
            var correspondingPostingList = cachedPostings.get(word);
            if (correspondingPostingList == null) {
                // posting list for the term was not cached
                correspondingPostingList = new SkipList<>(
                        informationRetrievalSystem.getListOfPostingForToken(
                                word.replaceAll(VALID_CHAR_FOR_PARSER, WILDCARD)),
                        Posting.DOC_ID_COMPARATOR);
                cachedPostings.put(word, correspondingPostingList);
            }
            return correspondingPostingList;
        };

        SkipList<Posting> phraseQueryIntersection = getPostingListOfIthWordInPhrase.apply(0);
        for (int i = 1; !phraseQueryIntersection.isEmpty() && i < matchingPhrase.size(); i++) {
            SkipList<Posting> postings = getPostingListOfIthWordInPhrase.apply(i);
            phraseQueryIntersection = SkipList.intersection(
                    phraseQueryIntersection, postings,
                    biPredicatesForCheckingPositionsForPhrasalQueries[i - 1], Posting.DOC_ID_COMPARATOR);
        }

        tmpResults = phraseQueryIntersection;
    } else if (isMatchingValueSet()) {
        tmpResults = new SkipList<>(
                informationRetrievalSystem.getListOfPostingForToken(matchingValue),
                Posting.DOC_ID_COMPARATOR);
    } else {
        // normalization of input matching value leads to null, hence no results can be found
        tmpResults = new SkipList<>(Posting.DOC_ID_COMPARATOR);
    }
```

```
                if (tmpResults.isEmpty() && automaticCorrectionEnabled) {
                    if (isMatchingValueSet()) {
                        var old = matchingValue;
                        var nearestWords = findNearest(matchingValue);
                        if (!nearestWords.isEmpty()) {
                            matchingValue = nearestWords.get(0);
                            for (int i = 1; i < nearestWords.size(); i++) {
                                set(or(nearestWords.get(i)));
                            }
                            automaticCorrections.put(old, nearestWords);
                            tmpResults = evaluateBothSimpleAndAggregatedExpressionRecursively();
                        }
                    }
                }

                yield tmpResults;
            }
        }
    };
    return results;
}
```

Evaluation

```
private SkipList<Posting> evaluateNotQuery() {
    // First: solve the direct query (create a new query without the NOT operator),
    // then take the difference to get the results for the NOT query.

    assert unaryOperator.equals(UNARY_OPERATOR.NOT);

    if (DEBUG_NOT_QUERY) {
        System.out.println("Evaluation of NOT query. DEBUG_NOT_QUERY flag is enabled, statistics are shown." +
                " It can be disabled by app properties.");
    }

    long start = 0, stop = 0;
    if (DEBUG_NOT_QUERY) {
        start = System.nanoTime();
    }
    SkipList<Posting> listOfPostingsToBeExcluded =
            new BooleanExpression(this)
                    .setUnaryOperator(UNARY_OPERATOR.IDENTITY)
                    .evaluateBothSimpleAndAggregatedExpressionRecursively();
    if (DEBUG_NOT_QUERY) {
        stop = System.nanoTime();
        System.out.println("SkipList of postings to exclude computed in       " + (stop - start) / 1e6 + " ms.");
    }

    if (DEBUG_NOT_QUERY) {
        start = System.nanoTime();
    }
    var allPostings = informationRetrievalSystem.getAllPostings();
    if (DEBUG_NOT_QUERY) {
        stop = System.nanoTime();
        System.out.println("SkipList of all postings in the index obtained in  " + (stop - start) / 1e6 + " ms.");
    }

    if (DEBUG_NOT_QUERY) {
        start = System.nanoTime();
    }
    var difference = SkipList.difference(
            allPostings, listOfPostingsToBeExcluded,
            Posting.DOC_ID_COMPARATOR); // "true predicate" means "classical" difference between sets
    if (DEBUG_NOT_QUERY) {
        stop = System.nanoTime();
        System.out.println("SkipList of difference computed in                 " + (stop - start) / 1e6 + " ms.");
    }

    return difference;
}
```

Evaluation

```
public List<Document> evaluate()
      throws UnsupportedOperationException {
   try {
      results = evaluateBothSimpleAndAggregatedExpressionRecursively();
   } catch (StackOverflowError e) {
      System.err.println("No more results will be shown due to low stack memory.");
      if (spellingCorrector != null) {    // Often spelling correction causes errors due to the high computational cost
         spellingCorrector.stop();
      }
   }
   assert results.stream().sorted().distinct().toList().equals(results.stream().toList());
   if (!maxNumberOfResultsSpecified) {
      maxNumberOfResults = results.size();
   }

   final var corpus = informationRetrievalSystem.getCorpus();
   if (RANK_RESULTS) {
      return new QueryResultsRanking().getRankedDocuments();
   } else {
      return results.stream()
            .map(Posting::getDocId)
            .map(corpus::getDocumentByDocId)
            .limit(maxNumberOfResults)
            .toList();
   }
}
```

Ranking

- Based on heuristics
  - Results are ranked according to their **wf-idf weighting**
  - An extra-score is assigned for each query term which is present in the title of the document

```
public class QueryResultsRanking {

  /**
   * @return the {@link List} of results opportunely ranked.
   */
  @NotNull
  public List<Document> getRankedDocuments() {

    if (!alreadyEvaluated) {
      System.err.println("Trying to rank results of a NON-evaluated query");
      return new ArrayList<>();
    } else {

      Corpus corpus = informationRetrievalSystem.getCorpus();
      final int entireCorpusSize = corpus.size();

      return results.stream()
          .collect(Collectors.toMap(
              posting -> corpus.getDocumentByDocId(posting.getDocId()),                          // docId as key
              posting -> USE_WF_IDF ? posting.wfIdf(entireCorpusSize) : posting.tfIdf(entireCorpusSize), // score as value
              Double::sum,   // sum scores if more query terms are present in the same document
              LinkedHashMap::new))
          .entrySet().stream().sequential()
          .map(entry_docToRank -> {
            // Assign extra rank if any query term is present in the title of the document
            double score = entry_docToRank.getValue();
            List<String> queryTerms = Arrays.asList(
                Utility.split(getQueryWords(false, true)));
            long extraScore = corpus.howManyCommonNormalizedWordsWithDocTitle(
                entry_docToRank.getKey(), queryTerms);
            BiFunction<Double, Long, Double> assignExtraScore = (initialScore, extraScore_) ->
                extraScore_ > 1 ? initialScore * extraScore_ : extraScore_ + initialScore;
            score = assignExtraScore.apply(score, extraScore);
            return new Pair<>(entry_docToRank.getKey(), score);
          })
          .sorted(Map.Entry.<Document, Double>comparingByValue().reversed()) // highest rank first
          .map(Map.Entry::getKey)
          .limit(maxNumberOfResults)
          .toList();
    }
  }
}
```

Wildcard queries

- Resolution of tokens to posting lists is a responsibility of class *InvertedIndex*
  1. Everything between the first and the last wildcards is folded in a single wildcard
     - If more than one wildcard is present
  2. The simplified token is rotated
     - The single wildcard must appear at the end
  3. Search by prefix in the dictionary
  4. Filter results and keep only terms matching the wildcard query
  5. Returns the *SkipList* of *Posting* for the query

Method **getListOfPostingsForToken** from class *InvertedIndex*

Wildcards handling

```
public final SkipList<Posting> getListOfPostingsForToken(String normalizedToken) {
  String tokenWithEndOfWord = normalizedToken + END_OF_WORD;
  int indexOfFirstWildcardIfPresent = tokenWithEndOfWord.indexOf(WILDCARD);
  if (indexOfFirstWildcardIfPresent > -1) {
    int indexOfLastWildcardIfPresent = tokenWithEndOfWord.lastIndexOf(WILDCARD);
    boolean moreThanOneWildcardIsPresent = indexOfLastWildcardIfPresent > indexOfFirstWildcardIfPresent;

    if (moreThanOneWildcardIsPresent) {
      // Consider the simplified wildcard input token where everything between the first and
      // the last wildcard is folded in a single wildcard
      tokenWithEndOfWord = tokenWithEndOfWord.replaceAll(
          ESCAPED_WILDCARD_FOR_REGEX + ".*" + ESCAPED_WILDCARD_FOR_REGEX,
          ESCAPED_WILDCARD_FOR_REGEX);
    }

    // Rotate the token such that the wildcard appears at the end
    String rotatedToken = tokenWithEndOfWord.substring(indexOfFirstWildcardIfPresent + 1)
        + tokenWithEndOfWord.substring(0, indexOfFirstWildcardIfPresent);
    // now the wildcard is removed (via substring) but the token has been rotated correctly
    // and, since we are here, we are sure that there is at least one wildcard

    return new SkipList<>(getDictionaryTermsContainingPrefix(rotatedToken, false)
        .stream().unordered().parallel()
        .distinct()
        .peek(tokenFromDictionary -> {
          assert tokenFromDictionary != null && !tokenFromDictionary.isBlank();
        })
        .filter(tokenFromDictionary -> MatcherForPermutermIndex
            .isWildcardQueryCompatibleWithStemmedTokenFromIndex(
                normalizedToken.replaceAll(ESCAPED_WILDCARD_FOR_REGEX, WILDCARD),
                tokenFromDictionary, corpus.getLanguage()))
        .map(invertedIndex::get)
        .filter(Objects::nonNull)
        .map(Term::getListOfPostings)
        .flatMap(Collection::stream)
        .collect(Collectors.toList()));
  } else {
    // no wildcards present in input token
    Term t = invertedIndex.get(normalizedToken);
    return t == null ? new SkipList<>() : t.getListOfPostings();
  }
}
```

Wildcard queries

- Resolution of tokens to posting lists is a responsibility of class *InvertedIndex*
  1. Everything between the first and the last wildcards is folded in a single wildcard
     - If more than one wildcard is present
  2. The simplified token is rotated
     - The single wildcard must appear at the end
  3. Search by prefix in the dictionary
  4. Filter results and keep only terms matching the wildcard query
  5. Returns the *SkipList* of *Posting* for the query

Method **isWildcardQueryCompatibleWithStemmedTokenFromIndex** from class *InvertedIndex*

```
public static boolean isWildcardQueryCompatibleWithStemmedTokenFromIndex(
      @NotNull String unstemmedInputWildcardQuery, @NotNull String stemmedTokenFromIndex, @NotNull Language language) {
   return new MatcherForPermutermIndex(unstemmedInputWildcardQuery, stemmedTokenFromIndex, language)
         .getResult().equals(Results.VALID);
}
```

MatcherForPermutermIndex is a **Finite State Machine**

Wildcard queries

- *MatcherForPermutermIndex*
  - is a **Finite State Machine**
  - handles more than one wildcard in the same token

(1) i < q.length AND j < t.length AND (q[i] = t[j] OR *stem(* t.substring(0,j) + q.substring(i).removeAll("*") *)* = t)
(2) i < q.length-1 AND j < t.length AND q[i] ≠ t[j] AND q[i] = '*'
(3) i < q.length-1 AND j < t.length AND q[i+1] ≠ t[j]
(4) (i < q.length-1 AND j < t.length AND q[i+1] = t[j]) OR (i = q.length-2 AND j = t.length-1 AND q[i] = '*' AND q[i+1] = t[j])
(5) (i = q.length OR i = q.length - 1) AND j = t.length
(6) i = q.length-1 AND j < t.length AND q[i] ≠ t[j] AND q[i] = '*'
(7) i = q.length-1 AND j < t.length AND q[i] ≠ t[j] AND q[i] ≠ '*'
(8) i < q.length-1 AND j < t.length AND q[i] ≠ t[j] AND q[i] ≠ '*'
(9) S.isEmpty()
(10) ! S.isEmpty()
(11) i < q.length AND j = t.length
(12) i = q.length AND j < t.length
(13) i < q.length AND j = t.length AND i < q.length AND *stem(* t + q.substring(i).removeAll("*") *)* = t
(14) q[i] = '*' AND i = q.length-1
(15) q[i] ≠ '*' OR i < q.length-1
(16) i < q.length-1 AND j < t.length AND q[i] ≠ t[j] AND q[i+1] = '*' AND q[i] ≠ '*'
(17) j = t.length AND i < q.length AND *stem(* t + q.substring(i+1).removeAll("*") *)* = t
(18) i < q.length AND j < t.length AND q[i] ≠ t[j] AND q[i] ≠ '*'

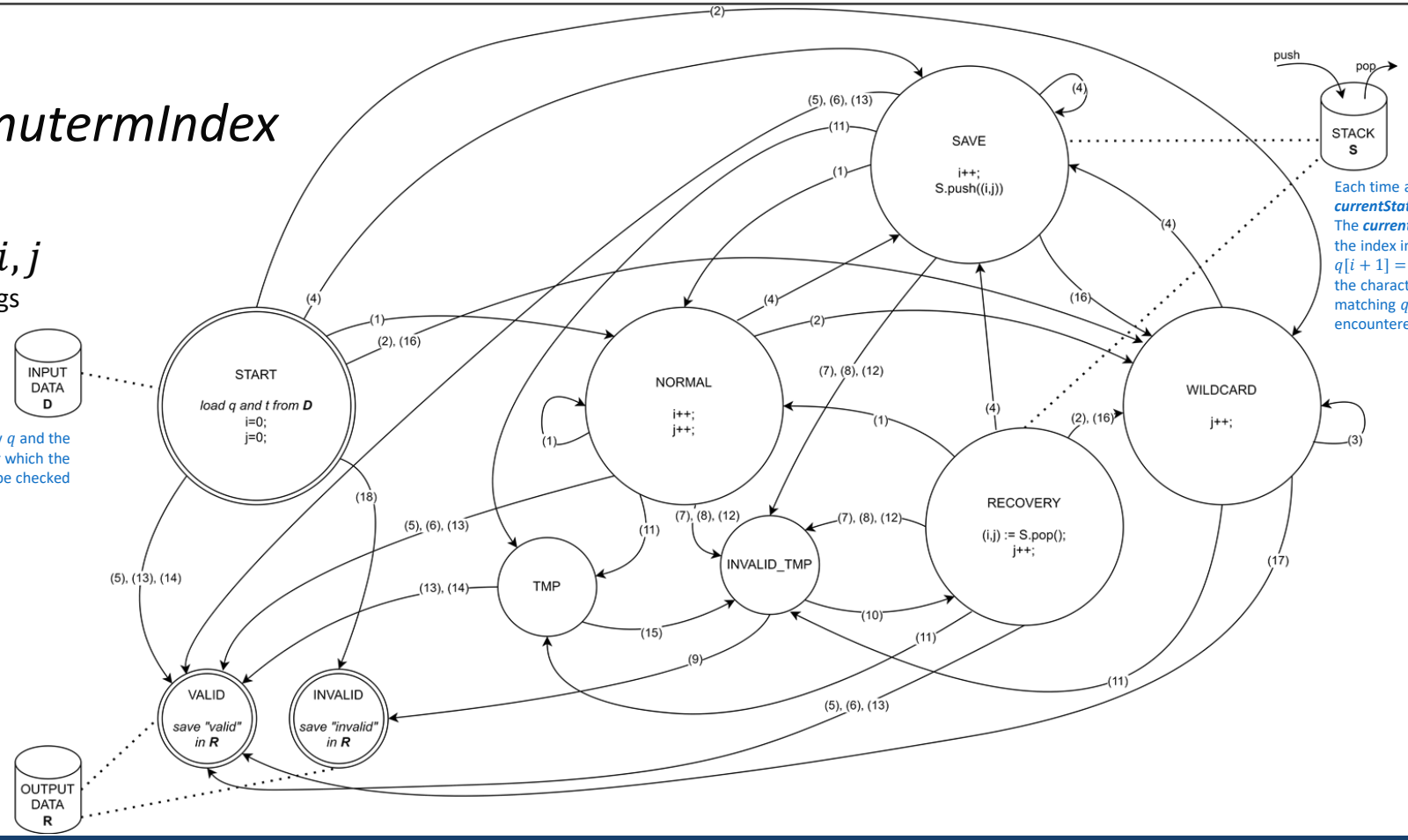
Goal: Given an un-stemmed **wildcard query** q and a stemmed token t, return **true** if the stemmed version of the wildcard query is compatible with the given stemmed token, **false** otherwise.

The problem is solved using an implementation of a modified finite-state machine, where each state can take some input parameter and its state evolves dynamically according to its current state and the input parameters.

current state / input	START	NORMAL	WILDCARD	VALID	INVALID	INVALID_TMP	TMP	RECOVERY	SAVE
no input (unstable state)	-	-	-	*end*	*end*	-	-	-	-
(1)	NORMAL	NORMAL	-	-	-	-	-	NORMAL	NORMAL
(2)	WILDCARD	WILDCARD	-	-	-	-	-	WILDCARD	-
(3)	-	-	WILDCARD	-	-	-	-	-	-
(4)	-	SAVE	SAVE	-	-	-	-	SAVE	SAVE
(5)	VALID	VALID	-	-	-	-	-	VALID	VALID
(6)	-	VALID	-	-	-	-	-	VALID	VALID
(7)	-	INVALID_TMP	-	-	-	-	-	INVALID_TMP	INVALID_TMP
(8)	-	INVALID_TMP	-	-	-	-	-	INVALID_TMP	INVALID_TMP
(9)	-	-	-	-	-	INVALID	-	-	-
(10)	-	-	-	-	-	RECOVERY	-	-	-
(11)	-	TMP	INVALID_TMP	-	-	-	-	TMP	TMP
(12)	-	INVALID_TMP	-	-	-	-	-	INVALID_TMP	INVALID_TMP
(13)	VALID	VALID	-	-	-	-	VALID	VALID	VALID
(14)	-	-	-	-	-	-	VALID	-	-
(15)	-	-	-	-	-	-	INVALID_TMP	-	-
(16)	WILDCARD	WILDCARD	-	-	-	-	-	WILDCARD	WILDCARD
(17)	-	-	VALID	-	-	-	-	-	-
(18)	INVALID	-	-	-	-	-	-	-	-

Wildcard queries

- *MatcherForPermutermIndex*
  - Memories: D, S, R
  - Working variables: i, j
    - Counters to scan the strings
      - i for q
      - j for t



Input data: the wildcard query q and the stemmed token t for which the compatibility of q must be checked

The **result** (either *true* or *false*) is returned when the FSM converges.

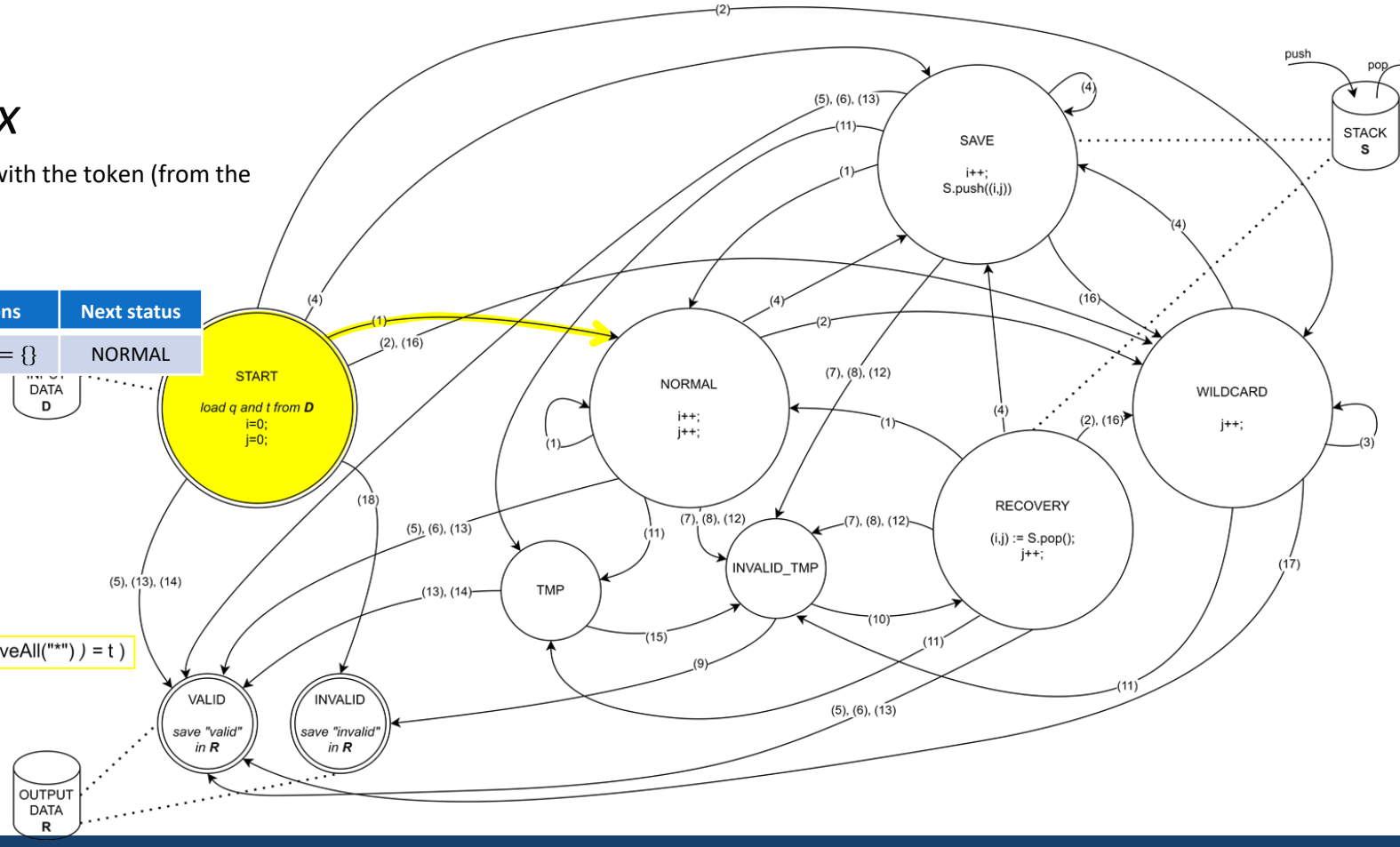
Each time a wildcard is found, the ***currentState*** is added to the *stack S*. The ***currentState*** is the pair (i, j), where j is the index in t such that the condition $q[i+1] = t[j]$ holds, i.e., the index in t of the character from which t restarts matching q after a wildcard was encountered in q.

Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	`tabl*e`	`table`	$i = 0, j = 0, S = \{\}$	NORMAL

(1) i < q.length AND j < t.length AND (q[i] = t[j] OR *stem(* t.substring(0,j) + q.substring(i).removeAll("*") *)* = t)

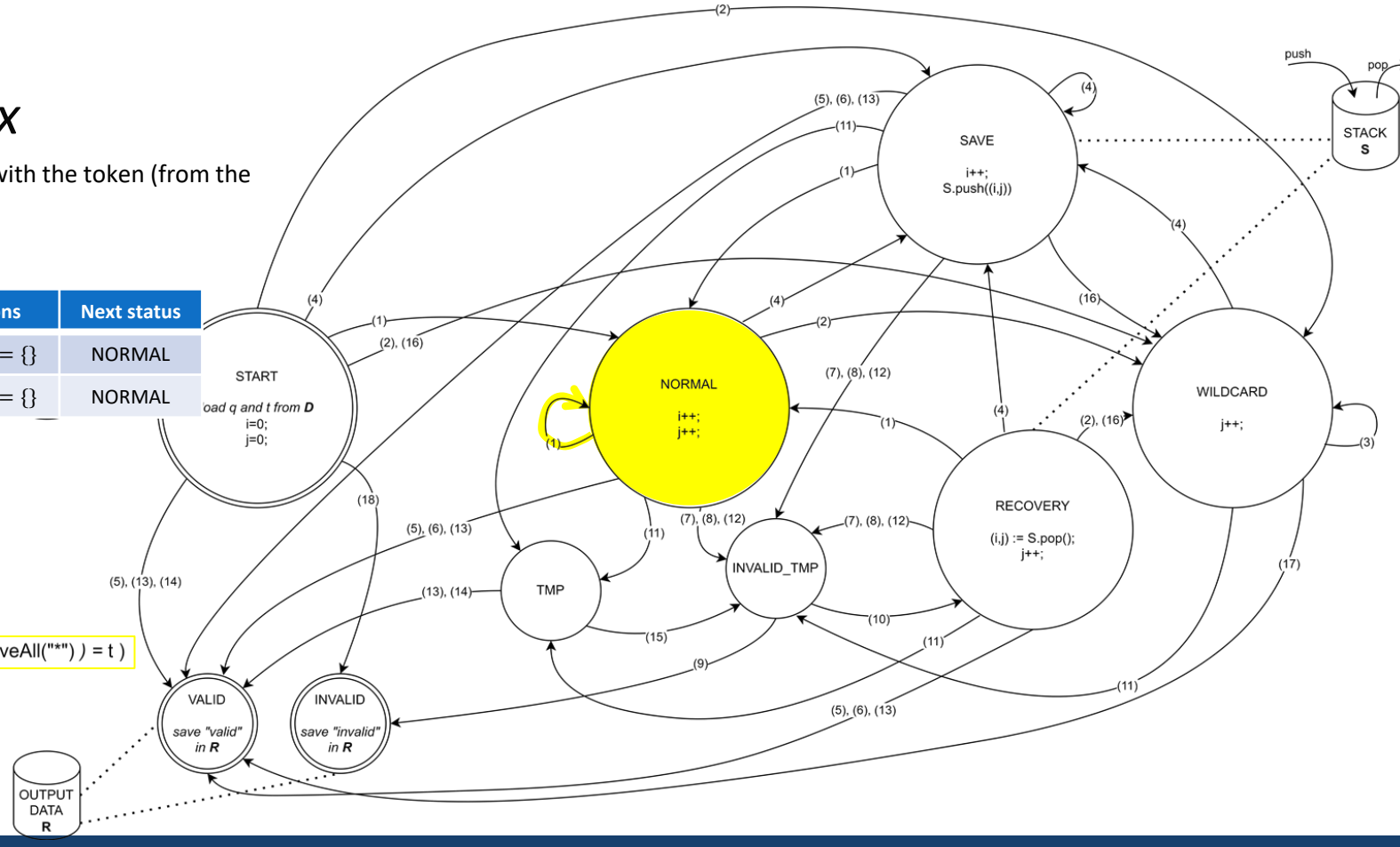


Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	`tabl*e`	`table`	$i = 0, j = 0, S = \{\}$	NORMAL
2	NORMAL	$i = 0, j = 0, S = \{\}$	`tabl*e`	`table`	$i = 1, j = 1, S = \{\}$	NORMAL

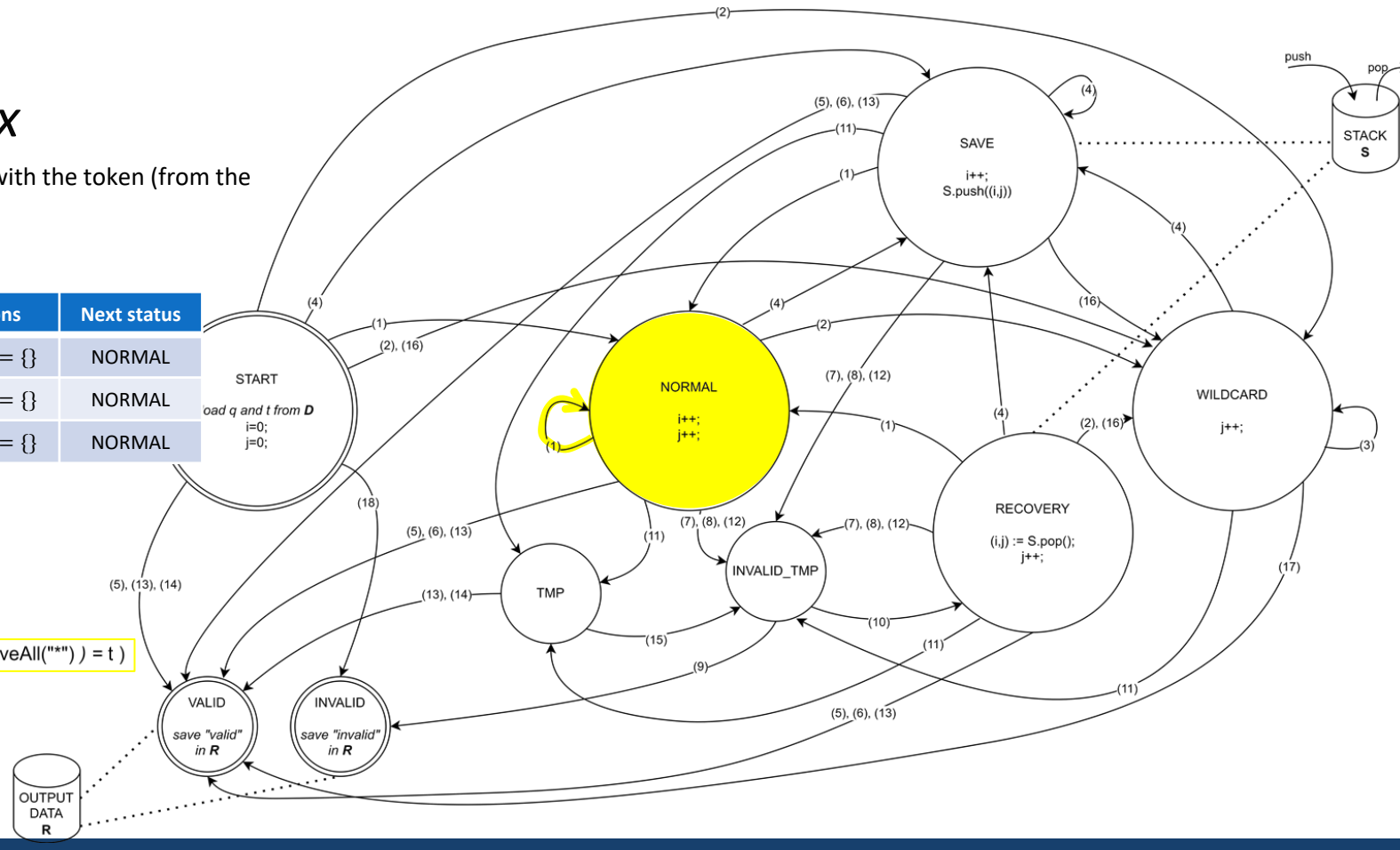
(1) i < q.length AND j < t.length AND (q[i] = t[j] OR *stem(* t.substring(0,j) + q.substring(i).removeAll("*") *)* = t)



Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	`tabl*e`	`table`	$i = 0, j = 0, S = \{\}$	NORMAL
2	NORMAL	$i = 0, j = 0, S = \{\}$	`tabl*e`	`table`	$i = 1, j = 1, S = \{\}$	NORMAL
3	NORMAL	$i = 1, j = 1, S = \{\}$	`tabl*e`	`table`	$i = 2, j = 2, S = \{\}$	NORMAL



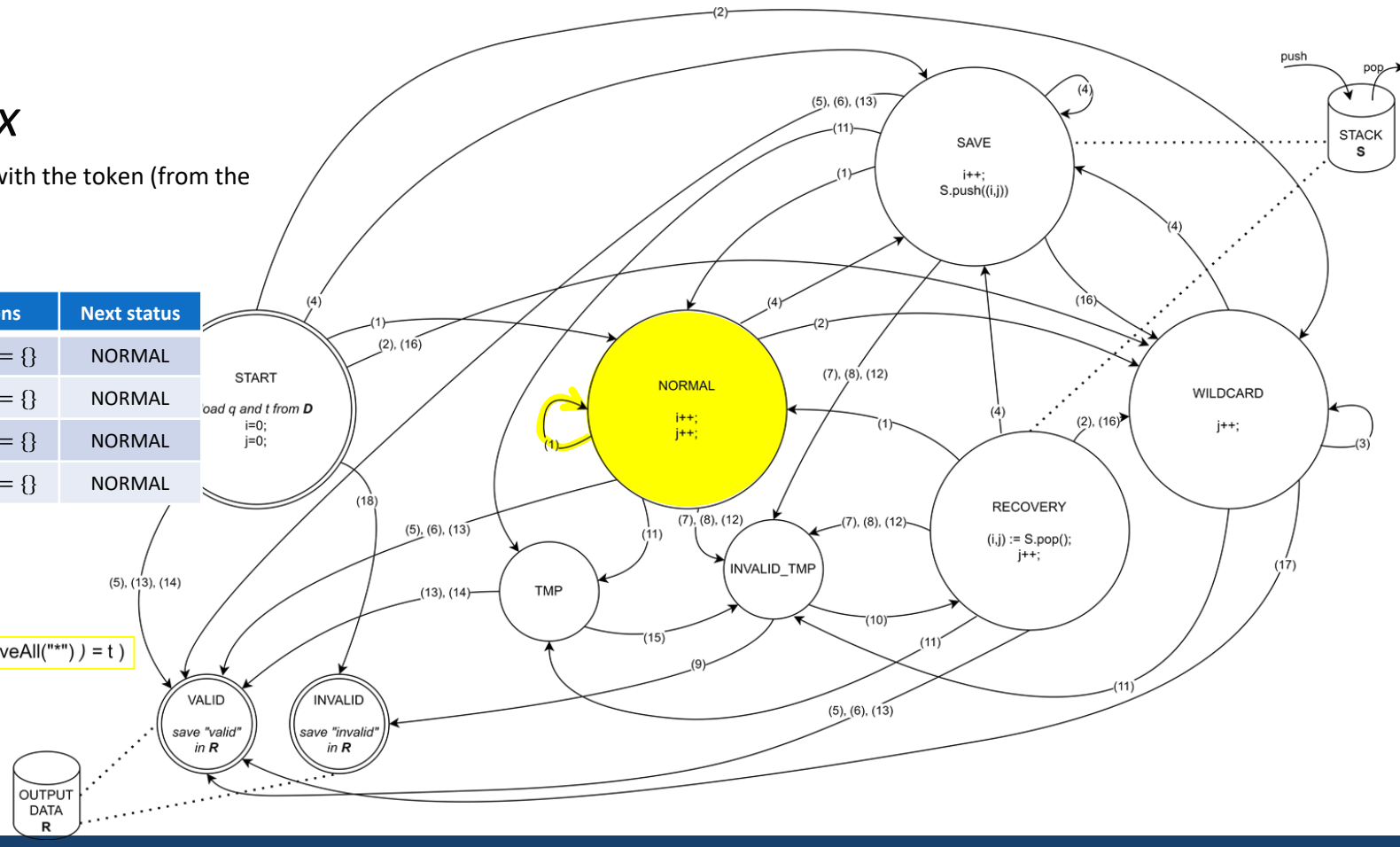
(1) i < q.length AND j < t.length AND (q[i] = t[j] OR *stem(* t.substring(0,j) + q.substring(i).removeAll("*") *)* = t)

Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	tabl*e	table	$i=0, j=0, S=\{\}$	NORMAL
2	NORMAL	$i=0, j=0, S=\{\}$	tabl*e	table	$i=1, j=1, S=\{\}$	NORMAL
3	NORMAL	$i=1, j=1, S=\{\}$	tabl*e	table	$i=2, j=2, S=\{\}$	NORMAL
4	NORMAL	$i=2, j=2, S=\{\}$	tabl*e	table	$i=3, j=3, S=\{\}$	NORMAL

(1) i < q.length AND j < t.length AND (q[i] = t[j] OR *stem(* t.substring(0,j) + q.substring(i).removeAll("*") *)* = t)

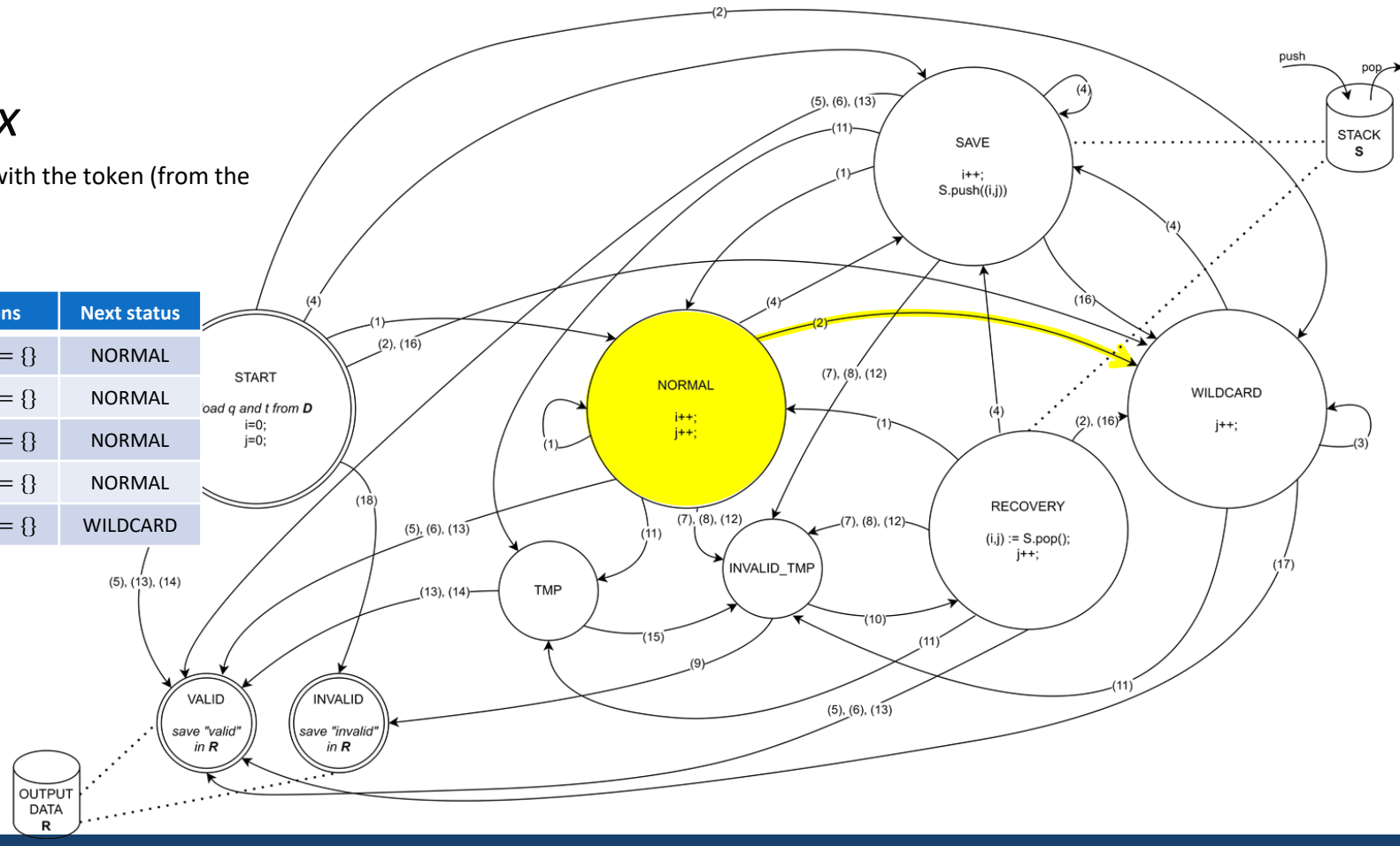


Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	tabl*e	table	$i=0, j=0, S=\{\}$	NORMAL
2	NORMAL	$i=0, j=0, S=\{\}$	tabl*e	table	$i=1, j=1, S=\{\}$	NORMAL
3	NORMAL	$i=1, j=1, S=\{\}$	tabl*e	table	$i=2, j=2, S=\{\}$	NORMAL
4	NORMAL	$i=2, j=2, S=\{\}$	tabl*e	table	$i=3, j=3, S=\{\}$	NORMAL
5	NORMAL	$i=3, j=3, S=\{\}$	tabl*e	table	$i=4, j=4, S=\{\}$	WILDCARD

(2) i < q.length-1 AND j < t.length AND q[i] ≠ t[j] AND q[i] = '*'

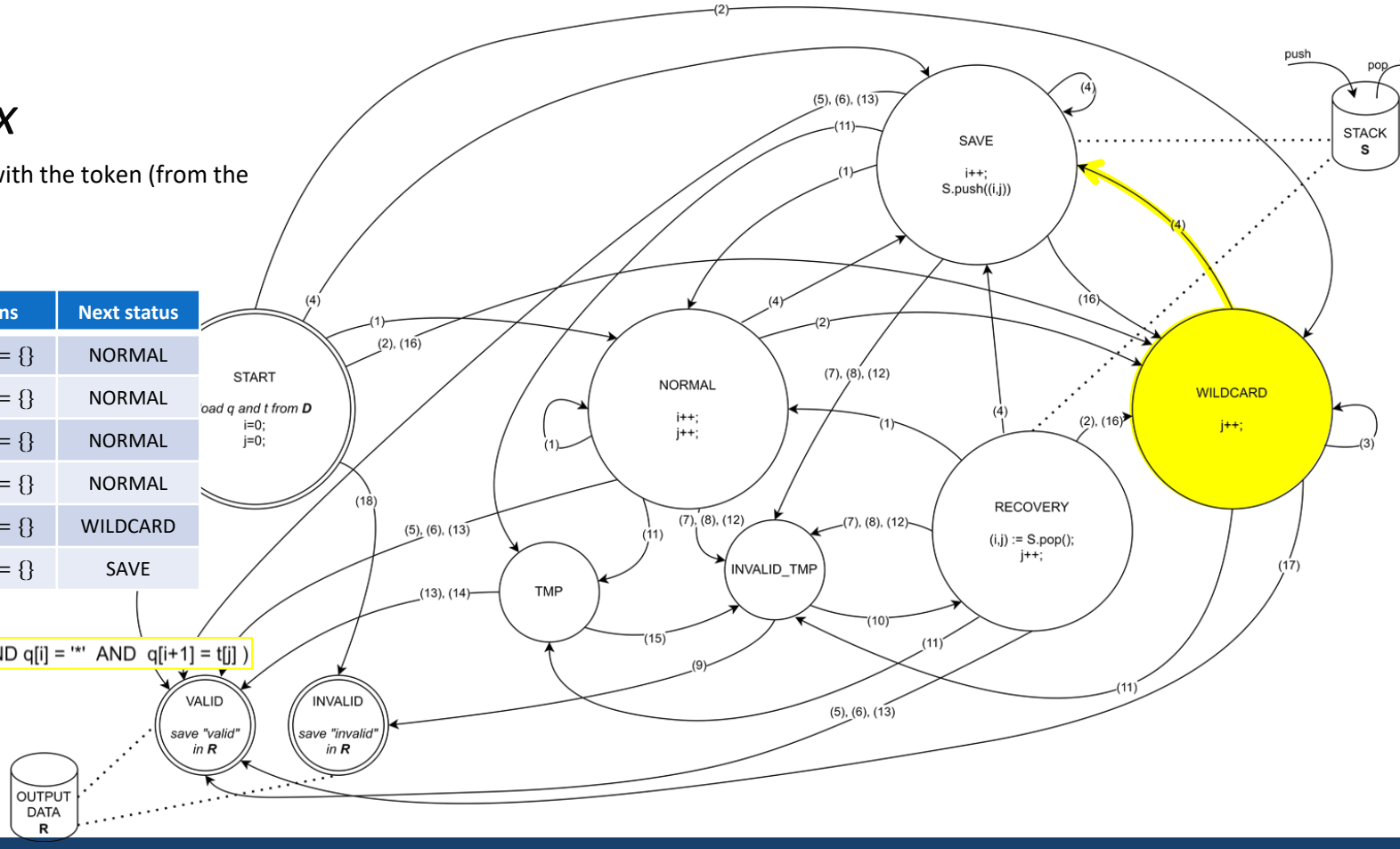


Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	tabl*e	table	$i=0, j=0, S=\{\}$	NORMAL
2	NORMAL	$i=0, j=0, S=\{\}$	tabl*e	table	$i=1, j=1, S=\{\}$	NORMAL
3	NORMAL	$i=1, j=1, S=\{\}$	tabl*e	table	$i=2, j=2, S=\{\}$	NORMAL
4	NORMAL	$i=2, j=2, S=\{\}$	tabl*e	table	$i=3, j=3, S=\{\}$	NORMAL
5	NORMAL	$i=3, j=3, S=\{\}$	tabl*e	table	$i=4, j=4, S=\{\}$	WILDCARD
6	WILDCARD	$i=4, j=4, S=\{\}$	tab*e	table	$i=4, j=5, S=\{\}$	SAVE

(4) (i < q.length-1 AND j < t.length AND q[i+1] = t[j]) OR (i = q.length-2 AND j = t.length-1 AND q[i] = '*' AND q[i+1] = t[j])

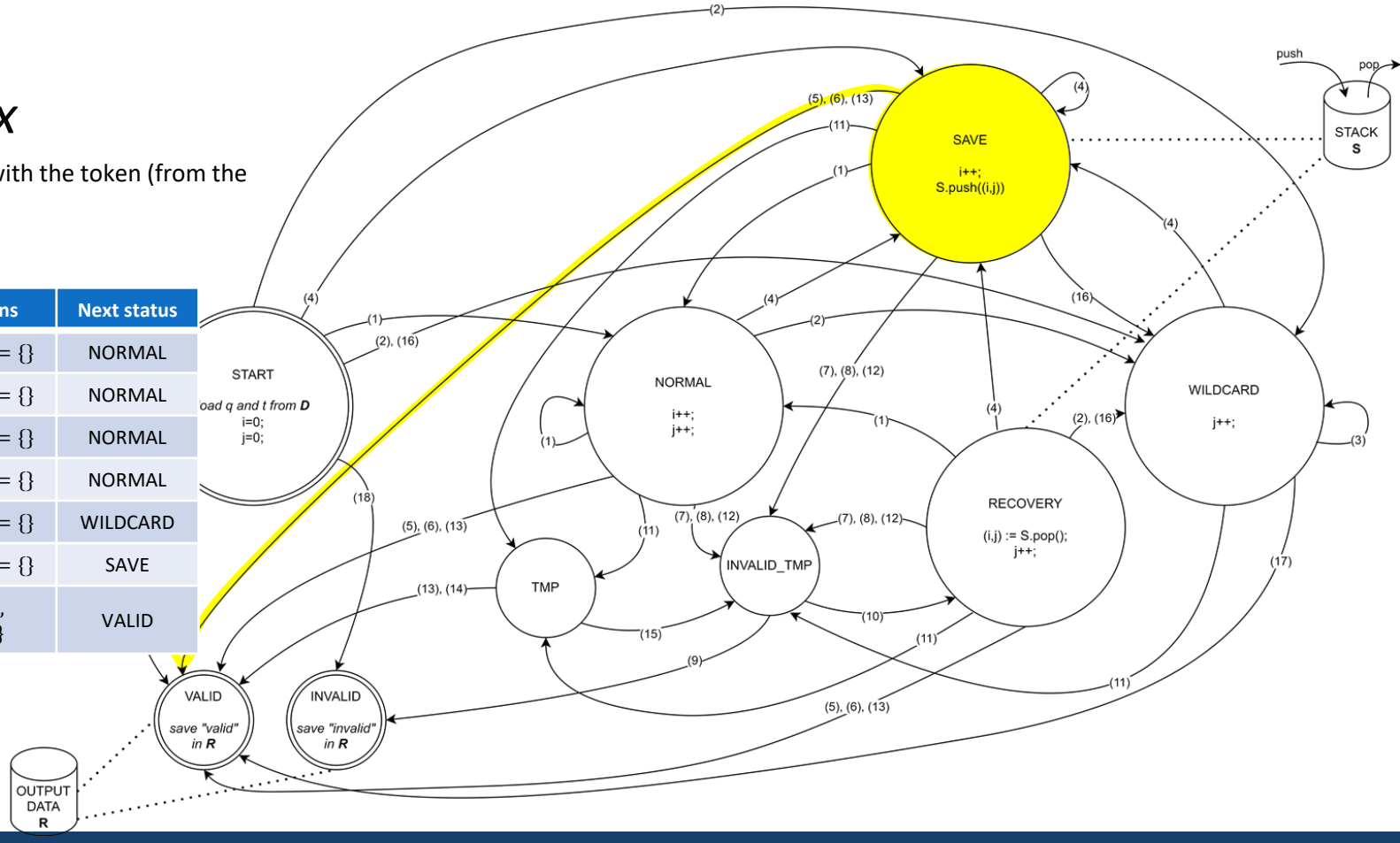


Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	tabl*e	table	$i=0, j=0, S=\{\}$	NORMAL
2	NORMAL	$i=0, j=0, S=\{\}$	tabl*e	table	$i=1, j=1, S=\{\}$	NORMAL
3	NORMAL	$i=1, j=1, S=\{\}$	tabl*e	table	$i=2, j=2, S=\{\}$	NORMAL
4	NORMAL	$i=2, j=2, S=\{\}$	tabl*e	table	$i=3, j=3, S=\{\}$	NORMAL
5	NORMAL	$i=3, j=3, S=\{\}$	tabl*e	table	$i=4, j=4, S=\{\}$	WILDCARD
6	WILDCARD	$i=4, j=4, S=\{\}$	tabl*e	table	$i=4, j=5, S=\{\}$	SAVE
7	SAVE	$i=4, j=5, S=\{\}$	tabl*e	table	$i=5, j=5,$ $S=\{(5,5)\}$	VALID

(5) (i = q.length OR i = q.length - 1) AND j = t.length

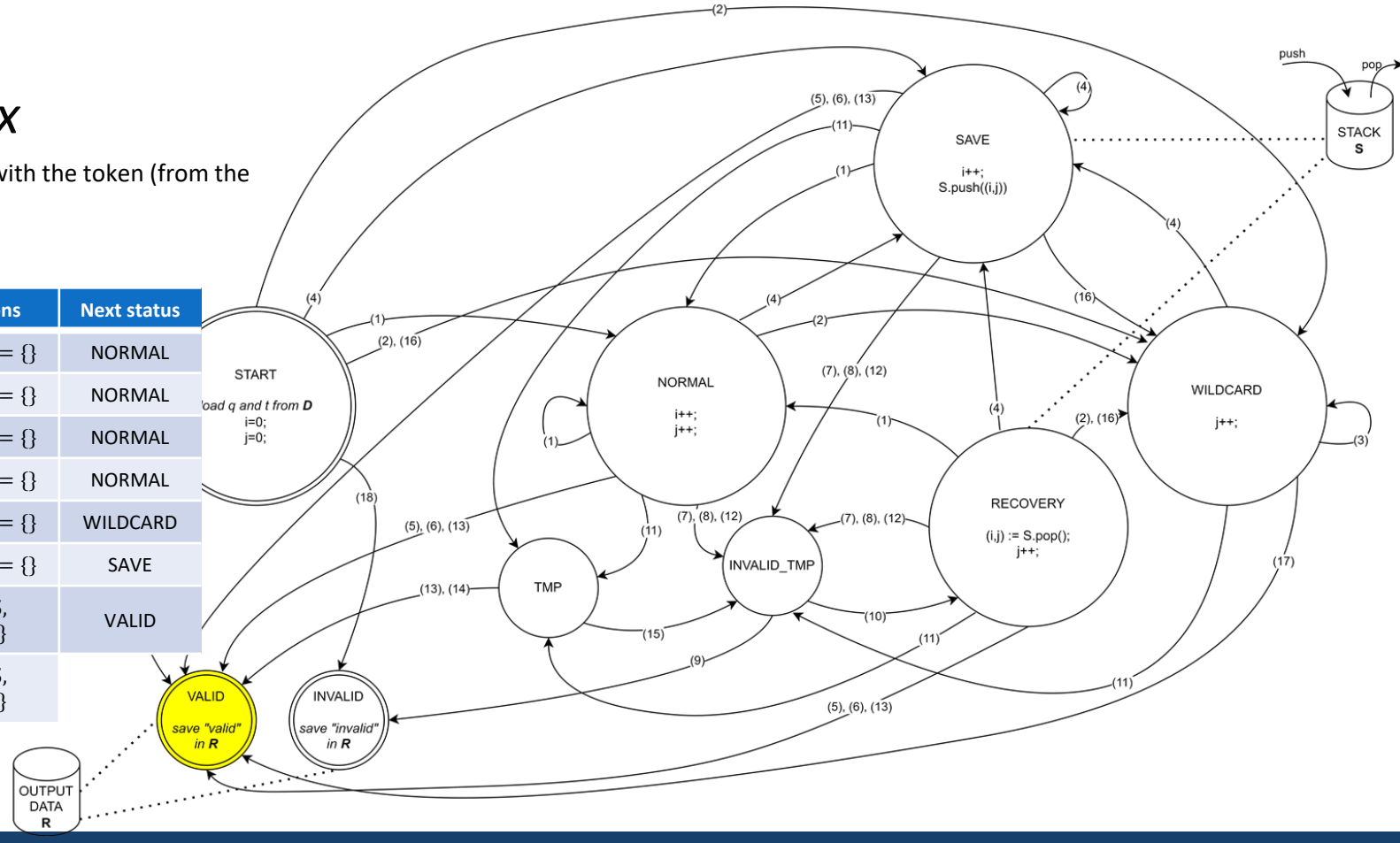


Wildcard queries

- *MatcherForPermutermIndex*
  - **Example**: is the wildcard query q =`tabl*e` compatible with the token (from the dictionary) t =`table` ?
    - Expected answer: yes, the wildcard '*' is replacing nothing.

Step	Input status	Preconditions	*q*	*t*	Postconditions	Next status
1	START	-	tabl*e	table	$i=0, j=0, S=\{\}$	NORMAL
2	NORMAL	$i=0, j=0, S=\{\}$	tabl*e	table	$i=1, j=1, S=\{\}$	NORMAL
3	NORMAL	$i=1, j=1, S=\{\}$	tabl*e	table	$i=2, j=2, S=\{\}$	NORMAL
4	NORMAL	$i=2, j=2, S=\{\}$	tabl*e	table	$i=3, j=3, S=\{\}$	NORMAL
5	NORMAL	$i=3, j=3, S=\{\}$	tabl*e	table	$i=4, j=4, S=\{\}$	WILDCARD
6	WILDCARD	$i=4, j=4, S=\{\}$	tabl*e	table	$i=4, j=5, S=\{\}$	SAVE
7	SAVE	$i=4, j=5, S=\{\}$	tabl*e	table	$i=5, j=5,$ $S=\{(5,5)\}$	VALID
8	VALID	$i=5, j=5,$ $S=\{(5,5)\}$	tabl*e	table	$i=5, j=5,$ $S=\{(5,5)\}$	

➔ FSM converged on the state "VALID", i.e., query q is compatible with token t



Phrase queries

- Recall
  - Structure of a *BooleanQuery* instance can save either a *value* (single word) or a *phrase*
  - *Postings* save the **positions** at which the term is present in the document
    - a posting refers both to a *Term* and a *Document* (thanks to the *docId*)

- During evaluation of the query
  - Preprocess the input phrase query
    - Words normalized or excluded by indexing creation cannot be found with a query
      ➔ queries must be preprocessed like documents were during the index creation
  - For each (preprocessed) word in the phrase query:
    1. Retrieve the list of postings referring to the word
    2. Get their intersection wrt. the *docId*
       - All words in the input phrase query must be present in the same document
       - Keep duplicates of postings pointing to the same *docId* but referring to distinct terms
         - The reference between the docId and the distinct words composing the input phrase must be kept
    3. For each *docId* (among those pointed by any of the postings in the list obtained from previous steps)
       1. Keep only postings for which term positions match the positions of the words in the input (preprocessed) phrase query

Posting	
DOC_ID_COMPARATOR	Comparator<Posting>
creationInstant	Instant
Posting(DocumentIdentifier, int[])	
compareCreationTimeTo(Posting)	int
compareTo(Posting)	int
equals(Object)	boolean
hashCode()	int
tf()	int
tfIdf(int)	double
toString()	String
wf()	double
wfIdf(int)	double
docId	DocumentIdentifier
term	Term
termPositionsInTheDocument	int[]

Phrase queries

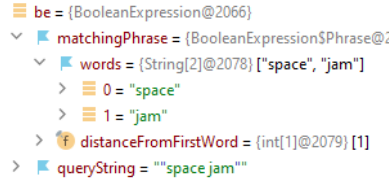
- Example
  - Input phrase query: `"space jam"`
  - Structure of a *BooleanQuery* instance can save either a *value* (single word) or a *phrase*
  - *Postings* save the **positions** at which the term is present in the document
    - a posting refers to a ***Term*** and a *Document* (thanks to the ***docId***)
- During evaluation of the query
  - Preprocess the input phrase query
    - Words normalized or excluded by indexing creation cannot be found with a query
      ➔ queries must be preprocessed like documents were during the index creation
  - For each (preprocessed) word in the phrase query:
    1. Retrieve the list of postings referring to the word
    2. Get their intersection wrt. the *docId*
       - All words in the input phrase query must be present in the same document
       - Keep duplicates of postings pointing to the same *docId* but referring to distinct terms
         - The reference between the docId and the distinct words composing the input phrase must be kept
    3. For each *docId* (among those pointed by any of the postings in the list obtained from previous steps)
       1. Keep only postings for which term positions match the positions of the words in the input (preprocessed) phrase query

Input: `"space jam"`

Quotes are part of the query to specify that it is a **phrase query**

Boolean expression creation

```
var be = irs.createNewBooleanExpression().parseQuery("\"space jam\"");
```



Some of the fields of the *BooleanExpression* instance created for the query "space jam". The *phrase* «space jam» is saved in an instance of class *BooleanExpression$Phrase* (inner class of *BoolanExpression*) and words composing the phrase are saved in *String array*

Words of the input query are **preprocessed** (here, only «to lower case»)

Query evaluation at the next slide

Phrase queries

- Example
  - Input phrase query: `"space jam"`
  - Structure of a *BooleanQuery* instance can save either a *value* (single word) or a *phrase*
  - *Postings* save the **positions** at which the term is present in the document
    - a posting refers to a ***Term*** and a *Document* (thanks to the ***docId***)
- During evaluation of the query
  - Preprocess the input phrase query
    - Words normalized or excluded by indexing creation cannot be found with a query
      ➔ queries must be preprocessed like documents were during the index creation
  - For each (preprocessed) word in the phrase query:
    1. Retrieve the list of postings referring to the word
    2. Get their intersection wrt. the *docId* ➔ Keeps only postings referring to documents containing both «space» and «jam»
       - All words in the input phrase query must be present in the same document
       - Keep duplicates of postings pointing to the same *docId* but referring to distinct terms
         - The reference between the docId and the distinct words composing the input phrase must be kept
    3. For each *docId* (among those pointed by any of the postings in the list obtained from previous steps)
       1. Keep only postings for which term positions match the positions of the words in the input (preprocessed) phrase query

➔ Group postings by *docId* and, if it equals, then keeps only postings referring to *docId* for which «space» is at position i and «jam» at position $i + 1$

[continues from the previous slide]

Code **extracted** from method *BooleanQuery. evaluateBothSimpleAndAggregatedExpressionRecursively():*

```
if (isMatchingPhraseSet()) {
  BiPredicate<Posting, Posting>[] biPredicatesForCheckingPositionsForPhrasalQueries =
      IntStream.range(0, matchingPhrase.size() - 1)
          .mapToObj(i -> (BiPredicate<Posting, Posting>)
              (Posting posting1, Posting posting2) -> {
                var positions1 = posting1.getTermPositionsInTheDocument();   // array of positions of j-th word in the phrase
                var positions2 = posting2.getTermPositionsInTheDocument();   // array of positions of (j+1)-th word in the phrase
                // assert positions were sorted
                assert Arrays.stream(positions1).sorted().boxed().toList().equals(Arrays.stream(positions1).boxed().toList());
                assert Arrays.stream(positions2).sorted().boxed().toList().equals(Arrays.stream(positions2).boxed().toList());

                int index1 = 0, index2 = 0;
                while (index1 < positions1.length && index2 < positions2.length) {
                  int numberOfTermsBetweenWords = positions2[index2] - positions1[index1] - matchingPhrase.distanceFromFirstWord[i];
                  if (numberOfTermsBetweenWords == 0) {          // words are adjacent
                    return true;
                  } else if (numberOfTermsBetweenWords < 0) {    // word from posting2 is present in the document before the word of posting1
                    index2++;
                  } else {                                       // word from posting1 is present in the document before the word of posting2
                    index1++;
                  }
                }
                return false;  // no adjacency found
              })
          .toArray(BiPredicate[]::new);

  assert matchingPhrase.words.length >= 2; // assertion failure means that the query is not a phrasal query and we should not be here…

  Map<String, SkipList<Posting>> cachedPostings =  // in phrases, common words (like articles) might be present more than once
      new ConcurrentHashMap<>(matchingPhrase.size());

  IntFunction<@NotNull SkipList<Posting>> getPostingListOfIthWordInPhrase = i -> {
    assert i >= 0 && i < matchingPhrase.size();
    String word = matchingPhrase.words[i];
    assert word != null;
    var correspondingPostingList = cachedPostings.get(word);
    if (correspondingPostingList == null) {
      // posting list for the term was not cached
      correspondingPostingList = new SkipList<>(
          informationRetrievalSystem.getListOfPostingForToken(
              word.replaceAll(VALID_CHAR_FOR_PARSER, WILDCARD)),
          /* wildcards ('*') were replaced with the symbol VALID_CHAR_FOR_PARSER,
             to avoid problems while parsing the input textual query, now they must
             be re-replaced with the correct symbol ('*') to query the IR system */
          Posting.DOC_ID_COMPARATOR /* only docId matters in comparisons*/);
      cachedPostings.put(word, correspondingPostingList);
    }
    return correspondingPostingList;
  };

  SkipList<Posting> phraseQueryIntersection = getPostingListOfIthWordInPhrase.apply(0);
  for (int i = 1; !phraseQueryIntersection.isEmpty() && i < matchingPhrase.size(); i++) {
    SkipList<Posting> postings = getPostingListOfIthWordInPhrase.apply(i);
    phraseQueryIntersection = SkipList.intersection(
        phraseQueryIntersection, postings,
        biPredicatesForCheckingPositionsForPhrasalQueries[i - 1], Posting.DOC_ID_COMPARATOR);
  }

  tmpResults = phraseQueryIntersection;
}
```

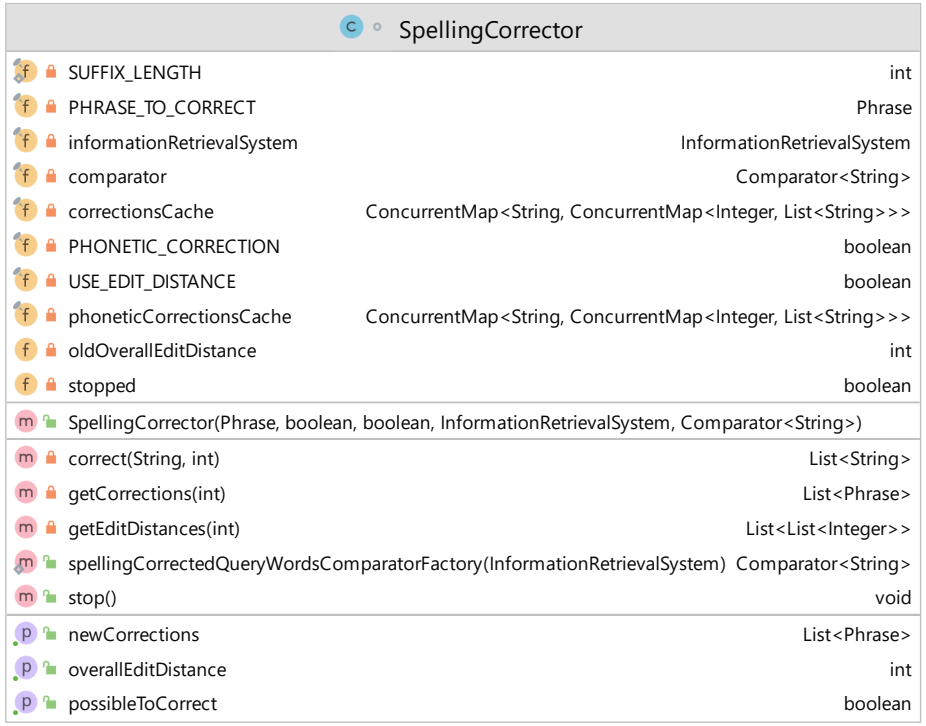
Retrieves all postings for the *i*-th word in the input phrase query

The actual intersection of postings

«tmpResults» because the code is «general» (either for phrase or simple queries), and, after the evaluation, saves the «tmpResults» in the variable named «results», but «tmpResults» already contains the *true* results.

Spelling and phonetic correction

- Class *SpellingCorrector*
  - makes use of the **edit distance**
  - works both for **spelling** and **phonetic** correction
  - A correction is applied on an instance of class *Phrase*
    - It can be a single word, too
  - Example
    - Input phrase: “spade jam”
    - Computation
      - Invocation of method *BooleanQuery.spellingCoorection(…)*
      - And then: *new SpellingCorrector(…).getNewCorrections()*
    - Spelling corrections: {“spade fam”, “spade jar”, “space jam”, …}
  - The spelling corrector creates new queries to be evaluated *(OR queries)* by class *BooleanQuery*



Spelling and phonetic correction

- Class *SpellingCorrector*
  - Fields
    - SUFFIX_LENGTH
      - Spelling correction algorithms correct a misspelled word by omitting a suffix and replacing the misspelled word with the (valid) **nearest** according to the **edit distance**
    - PHRASE_TO_CORRECT
      - The phrase to correct
      - Correcting one letter in each word of a phrase of *N* words would results in a correction at distance *N*
        - But we want to find the minimum distance
        - A *SpellingCorrector* instance saves the entire phrase and try to correct one word at a time, with all the combinations, trrying to minimize the **overall edit distance** of the resulting corrected phrase, wrt. the input misspelled phrase
    - informationRetrievalSystem
      - valid words are those of the dictionary
    - comparator
      - Several corrections might have the same edit distance wrt. the input
      - the comparator is used to sort the corrections

SpellingCorrector	
SUFFIX_LENGTH	int
PHRASE_TO_CORRECT	Phrase
informationRetrievalSystem	InformationRetrievalSystem
comparator	Comparator<String>
correctionsCache	ConcurrentMap<String, ConcurrentMap<Integer, List<String>>>
PHONETIC_CORRECTION	boolean
USE_EDIT_DISTANCE	boolean
phoneticCorrectionsCache	ConcurrentMap<String, ConcurrentMap<Integer, List<String>>>
oldOverallEditDistance	int
stopped	boolean
SpellingCorrector(Phrase, boolean, boolean, InformationRetrievalSystem, Comparator<String>)	
correct(String, int)	List<String>
getCorrections(int)	List<Phrase>
getEditDistances(int)	List<List<Integer>>
spellingCorrectedQueryWordsComparatorFactory(InformationRetrievalSystem)	Comparator<String>
stop()	void
newCorrections	List<Phrase>
overallEditDistance	int
possibleToCorrect	boolean

Spelling and phonetic correction

- Class *SpellingCorrector*
  - Fields
    - correctionsCache
      - caches corrections

Map< String, Map< Integer, List<String> > >

input word — Edit distance from the input word — List of corrections for the input word at the distance specified by the key

    - PHONETIC_CORRECTION
      - Flag: true if phonetic correction must be performed, false otherwise
    - overallEditDistance
      - The edit distance from the last returned corrections (by invoking *getNewCorrections()*) from the initial phrase
    - oldOverallEditDistance
      - The edit distance before the last invocation of *getNewCorrections()*
      - Used for determining if further corrections are still possible
        - If the overallEditDistance does not change anymore, then no further corrections are possible
    - USE_EDIT_DISTANCE
      - Flag:
        - if it is **true**, each time the method *getNewCorrections()* is invoked, only the corrections which are at most "overallEditDistance" far are returned (overallEditDistance is updated);
        - if it is **false**, all possible corrections are returned
      - It is recommended to set the flag to true
      - The flag is **ignored** for *non-phonetic* correction (edit distance *must* be used in such cases)
    - stop
      - becomes true by invoking method *stop()*
      - if it is true, it inhibits the *SpellingCorrector* instance to produce further corrections either if still possible
      - useful when the computational cost for a new correction is too high and unexpected behavior must be avoided

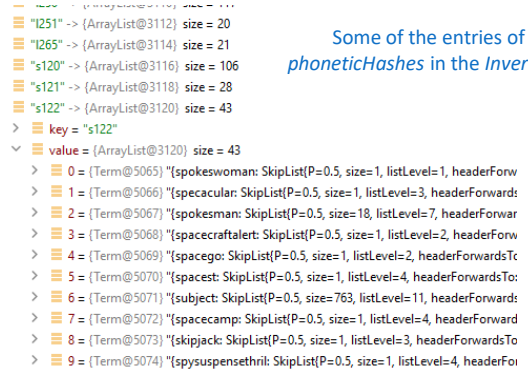
SpellingCorrector	
SUFFIX_LENGTH	int
PHRASE_TO_CORRECT	Phrase
informationRetrievalSystem	InformationRetrievalSystem
comparator	Comparator<String>
correctionsCache	ConcurrentMap<String, ConcurrentMap<Integer, List<String>>>
PHONETIC_CORRECTION	boolean
USE_EDIT_DISTANCE	boolean
phoneticCorrectionsCache	ConcurrentMap<String, ConcurrentMap<Integer, List<String>>>
oldOverallEditDistance	int
stopped	boolean
SpellingCorrector(Phrase, boolean, boolean, InformationRetrievalSystem, Comparator<String>)	
correct(String, int)	List<String>
getCorrections(int)	List<Phrase>
getEditDistances(int)	List<List<Integer>>
spellingCorrectedQueryWordsComparatorFactory(InformationRetrievalSystem)	Comparator<String>
stop()	void
newCorrections → Method *getNewCorrections()*	List<Phrase>
overallEditDistance	int
possibleToCorrect	boolean

Spelling and phonetic correction

Cost matrix obtained from the computation of the edit distance between "HOME" and "HOUSE", which is 2, by using the *EditDistanceCalculator*

	□	H	O	U	S	E
□	0	1	2	3	4	5
H	1	0	1	2	3	4
O	2	1	0	1	2	3
M	3	2	1	1	2	3
E	4	3	2	2	2	2

- Class *EditDistanceCalculator*
  - Used for **spelling correction**
  - **Edit distance** is computed wrt. words in the **dictionary** of the IR system
    - See class ***InvertedIndex***
- Soundex algorithm
  - Used for **phonetic corrections**
  - Corrections are made thanks to the **phonetic hashes**
    - See class ***InvertedIndex***
  - The algorithm is not perfect



Some of the entries of the *phoneticHashes* in the *InvertedIndex*

JUnit parametrized tests for the Soundex algorithm

```
@ParameterizedTest
@CsvSource({ "Robert, r163",  "Rupert, r163", "Rubin, r150", "Honeyman, h555"})
void testPhoneticHash(String inputWord, String expectedHash) {
   assertEquals(expectedHash, Soundex.getPhoneticHash(inputWord));
}
```

- Corrections are made by the method *SpellingCorrector.getNewCorrections()*
  - Returns corrections with the *minimum edit-distance*
  - If invoked more than once, the valid edit distance is increased at each invocation and further results (if available) are returned

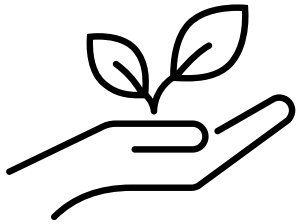
Queries

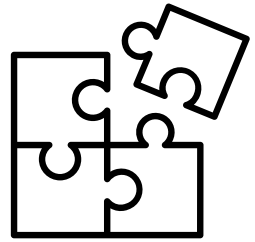
END

Configurability and extensibility

Configurability and extensibility

- Environment variables
- Modularity





Environment variables

- The following environment variables can be set to customize the system
  - workingDirectory_name = workingDirectory
    - The name for the working directory where to save files about the system (like log files and serialized version of an IR system instance)
  - app.name = "Information Retrieval System"
    - The name of the application
  - app.exclude_stop_words = false
    - Flag: if true, then stop words will not be indexed
  - app.stemmer = Porter
    - The stemmer to use, for now it can be “Porter” or “null” (to avoid using a stemmer)
    - Further stemmers can be implemented and added to the project
  - app.rank_query_results = true
    - Flag: if false, results of query evaluation are not ranked
  - app.use_wf_idf = true
    - Flag: if false, then *tf-idf* index is used for ranking, else, *wf-idf* is used (assuming that results must be ranked)
  - app.debug.NOTQueries = false
    - Flag: if true, several additional information are printed during the evaluation of NOT queries (for debugging purposes)

Modularity

- The system has been built following the modular approach
- It allows to
  - easily add new features (e.g., a new stemmer)
  - include the system in another project exploiting the API
  - adapt the system for indexing other collections
    - the following classes must be provided for each new collection
      - the **document descriptor**, i.e., a class extending class *Document* that represents a document of the new collection
      - the **corpus factory**, i.e., a class extending class *CorpusFactory* which specifies how the corpus must be created
    - See package “user_defined_contents”

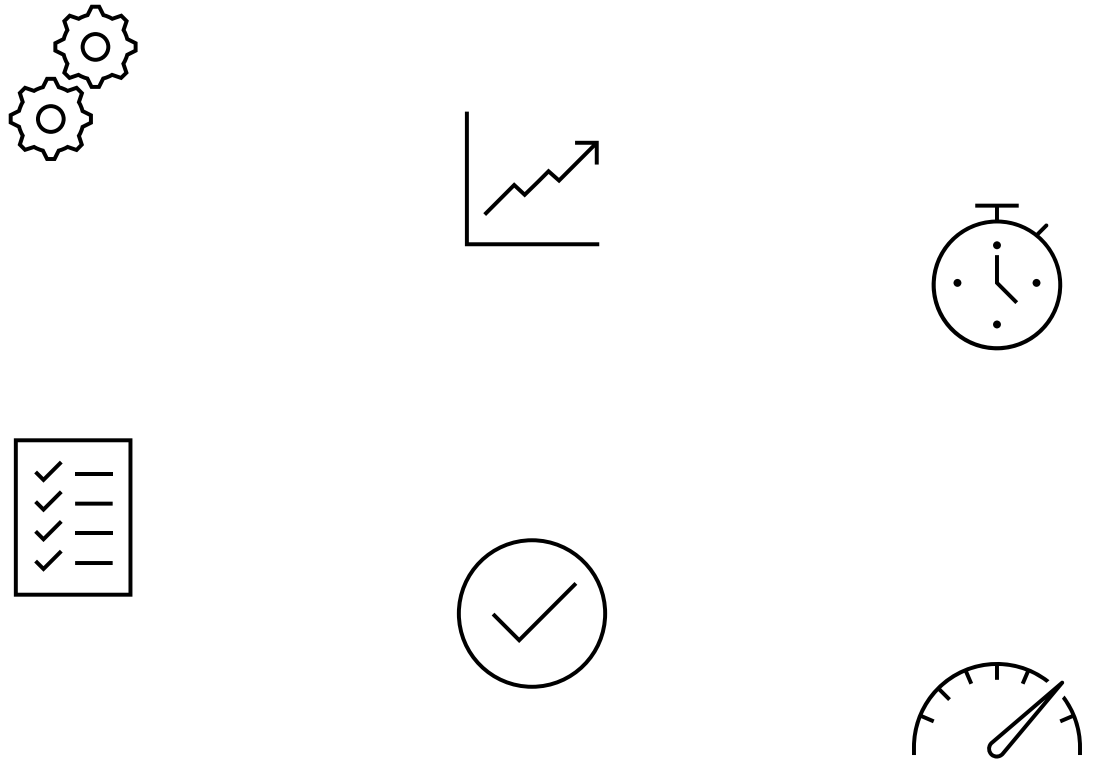
Configurability and extensibility

END

Evaluation

Evaluation

- Introduction to testing
- Precision
- Recall
- 11-point interpolated average precision
- MAP
- Precision@K
- R-precision
- Benchmark



Introduction to testing

- The system has been:
  - **deeply tested** (unit and integration tests)
    - More than **500** unit tests for the main IR system
    - More than **5000** units tests for the support code (e.g., *SkipList*)
  - **evaluated** with IR standard metrics
    - Precision, Recall, 11-point interpolated average precision, MAP, Precision@K, R-precision
    - Cranfield collection is used for evaluation
    - Queries were created with class *BooleanTestQueriesCreator*
    - Information needs provided as boolean queries (as strings)
  - and **benchmarked**
    - a benchmark framework for Java has been implemented
    - benchmarking performed on the Movie corpus
- Tests are available from the “evaluation” test package

Introduction to testing

- System configuration for tests
  - Results showed in the following tests have been obtained with the following environment variable values:
    - app.exclude_stop_words = false
    - app.stemmer = Porter
    - app.rank_query_results = true
    - app.use_wf_idf = true
    - app.debug.NOTQueries = false

Precision

- Fraction of retrieved documents which are relevant

$$\text{Precision} = \frac{|\text{relevant} \cap \text{retrieved}|}{|\text{retrieved}|}$$

	#queries	Average	Max	Min
Statistics	271	0.972	1.00 In 189 samples	0.500 In 2 samples
Excluding max and min	80	0.917	0.999 In 1 sample	0.565 In 1 sample

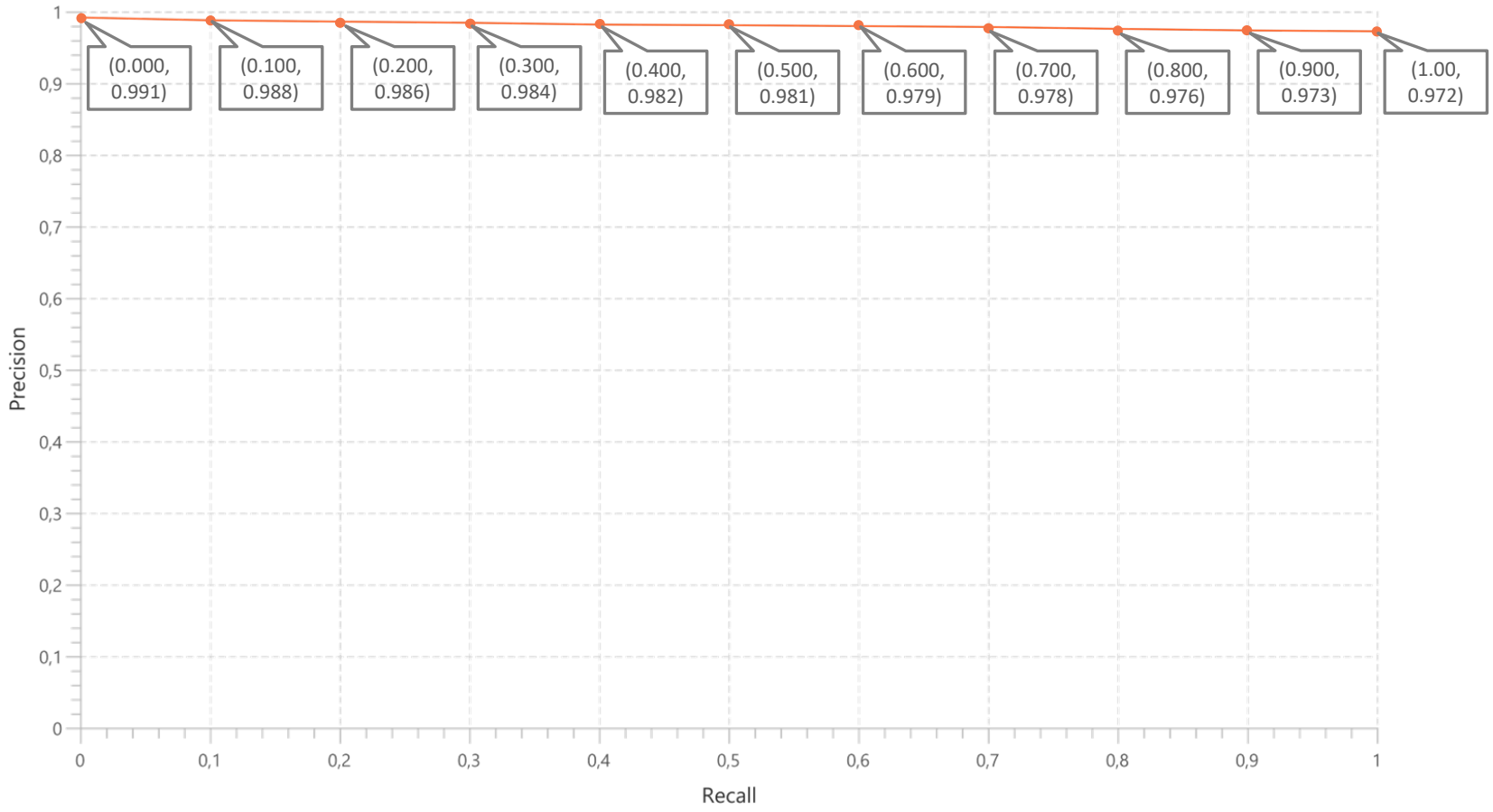
Recall

- Fraction of relevant documents which have been retrieved

$$\text{Recall} = \frac{|\text{relevant} \cap \text{retrieved}|}{|\text{relevant}|}$$

	#queries	Average	Max	Min
Statistics	271	0.996	1.00 In 159 samples	0.516 In 1 sample
Excluding max and min	111	0.995	0.999 In 1 sample	0.750 In 1 sample

11-point interpolated average precision



MAP

- Mean Average Precision
  - Provides a single measure of quality across recall levels
  - **Mean** of the **average precision**
    - **Average precision**
      - Average of the precision value obtained for the set of top *k* documents existing after each relevant document is retrieved for one information need
    - **Mean**
      - Average of the average precision over information needs
    - Information need provided as boolean query

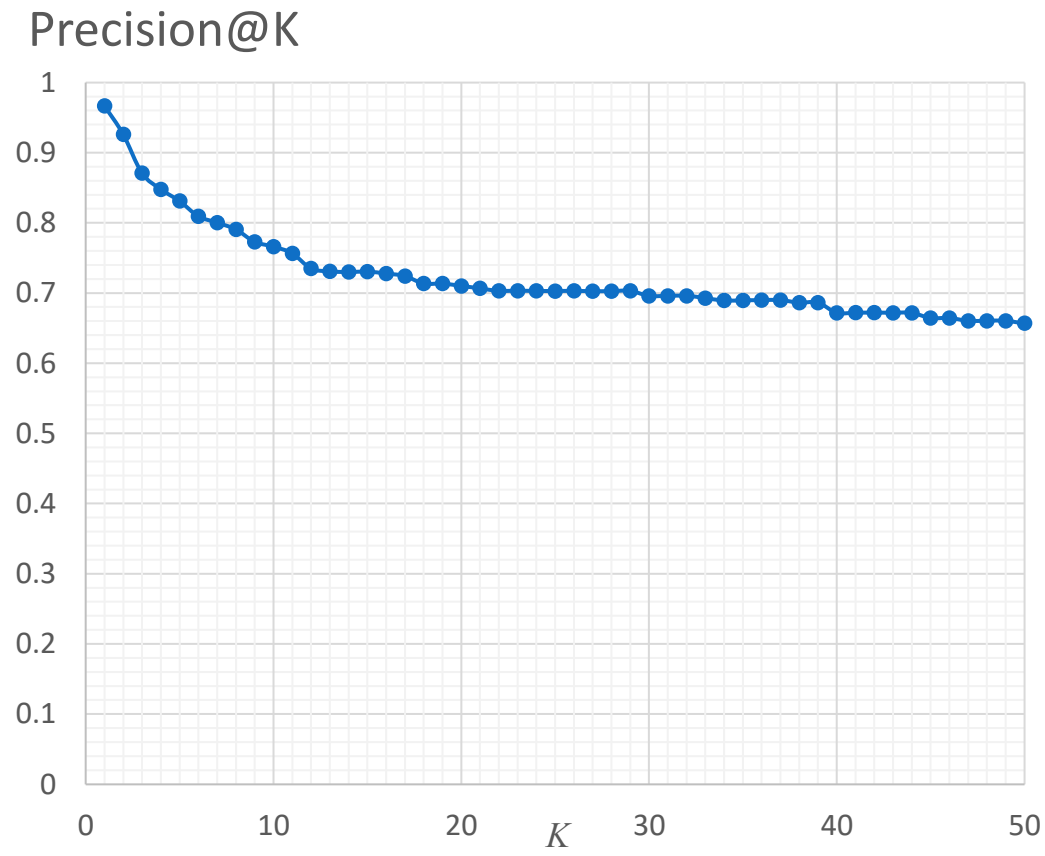
$$\text{MAP}(\text{Q}) = \underbrace{\frac{1}{|\text{Q}|}\sum_{j=1}^{|Q|}\underbrace{\frac{1}{m_j}\sum_{k=1}^{m_j}\text{Precision}\left(R_{jk}\right)}_{\text{Average precision for the } j\text{-th information need}}}_{\text{Average over information needs}} = \mathbf{0.968}$$

Q : set of information needs (boolean queries, in this case)
R_{jk} : set of ranked results for the *j*-th information need containing the *k* highest-ranked documents
m_j : number of available precision measures for the *j*-th information need

Precision@K

- is the precision computed after *K* documents have been retrieved
- Not very stable
  - If there are less the *K* relevant documents, then Precision@*K* cannot be 1
- averaged over 271 queries

K	1	2	3	5	10	20	50
Precision@*K*	0.967	0.923	0.870	0.830	0.765	0.711	0.657



R-precision

- If there are R relevant documents for a query,
  - R-precision is the precision of the top R ranked documents returned by the query

- R-precision: **0.701**
  - averaged over 271 queries

Benchmark

Test	#warm-up iterations	#iterations	#tear-down iterations	Fastest iteration [µs]	Slowest iteration [µs]	Average iteration [µs]
Retrieval of a posting list for a token randomly taken from the dictionary of the Movie corpus	100	100	100	0.6	30.6	1.7
Corpus creation for a fake collection of 1000 small documents (445 characters per document, documents already loaded in RAM)	1000	1000	1000	6.2	20.9	7.4
Creation of the inverted index for the Movie corpus	1	3	1	39,308,805.6	41,356,359.6	39,994,275.6
Creation of a posting list from 1000 unordered and eventually duplicated postings	100	100	100	134.7	478.6	218.1
Creation and evaluation of “one word” query (without string parsing)	10	10	10	22.5	245.9	62.0
Creation and evaluation of “single phrase” query (of five words, without string parsing)	10	10	10	21.3	422.3	131.7
Preprocessing of a document of size 140 KB	10	10	10	2.8	1,179.7	123.3

Benchmark

Test	#warm-up iterations	#iterations	#tear-down iterations	Fastest iteration [µs]	Slowest iteration [µs]	Average iteration [µs]
Creation and evaluation of “AND query” (between two words, without string parsing)	10	10	10	26.5	312.1	91.5
Creation and evaluation of “OR query” (between two words, without string parsing)	10	10	10	47.7	509.7	152.8
Creation and evaluation of “NOT query” (without string parsing)	10	10	10	230,822.1	299,520.9	260,300.6
Creation and evaluation of “AND phrase query” (between two phrases of five words each, without string parsing)	10	10	10	89.9	929.9	286.6
Creation and evaluation of “OR phrase query” (between two phrases of five words each, without string parsing)	10	10	10	60.3	6280.9	824.2
Intersection of two skip lists with 1000 random integers each	1000	1000	1000	212.0	710.9	330.7
Union of two skip lists with 1000 random integers each	1000	1000	1000	2,067.0	10,115.8	3,811.8

Notice: “NOT queries” in the form $(!\ w)$ require a lot of time because the system must compute the difference (set operation) between *all postings of the system* and all postings for the token w, in fact, this kind of queries are optimized when possible (e.g., $(w_1\ \&\ (!\ w_2))$ is evaluated by computing the difference (set operation) between the postings for w_1 and the postings for w_2, instead of computing the intersection between the postings for w_1 and the postings for $(!w_2)$.

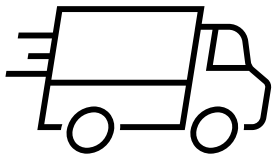
Evaluation

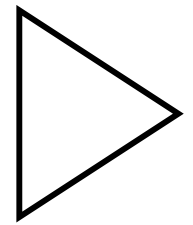
END

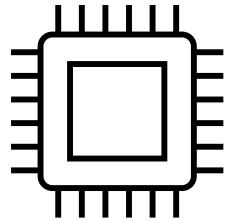
How to compile and execute

How to compile and execute

- Platform requirements
- Building and execution
- Distribution







Platform requirements

- No hard requirements are imposed for the application execution
  - assuming that a JAR file of the application is provided
  - Java 16 or higher is required
- Building the Java Application requires Maven 3.5
  - and an Internet connection to eventually download missing dependencies
- Test-phase (benchmark tests in particular) requires an high performance computer

Building and execution

- Maven commands (`mvn …`) must be executed from the root directory of the project
  - where the file "*pom.xml*" is present
- Building
  - `mvn package`
  - compiles, tests and creates the JAR executable file (in the directory named "*target*")
- Execution
  - `mvn exec:java@main`
- Benchmark
  - `mvn exec:java@benchmark`

Distribution

- During the building phase (`mvn package`), two JAR files are created
  - The larger includes also all the libraries, and can be distributed
- The final user
  - receive the larger JAR file (including all the libraries)
  - must have the correct Java version installed
  - can run the application
    - `java -jar *jarFileName.jar*`
    - from a terminal positioned where the JAR file is present

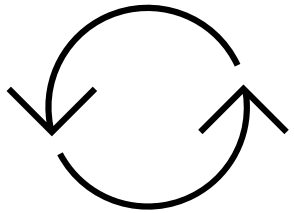
How to compile and execute

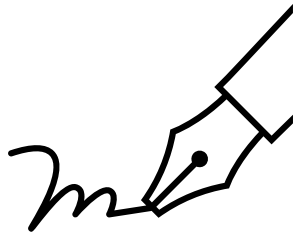
END

Summary

Summary

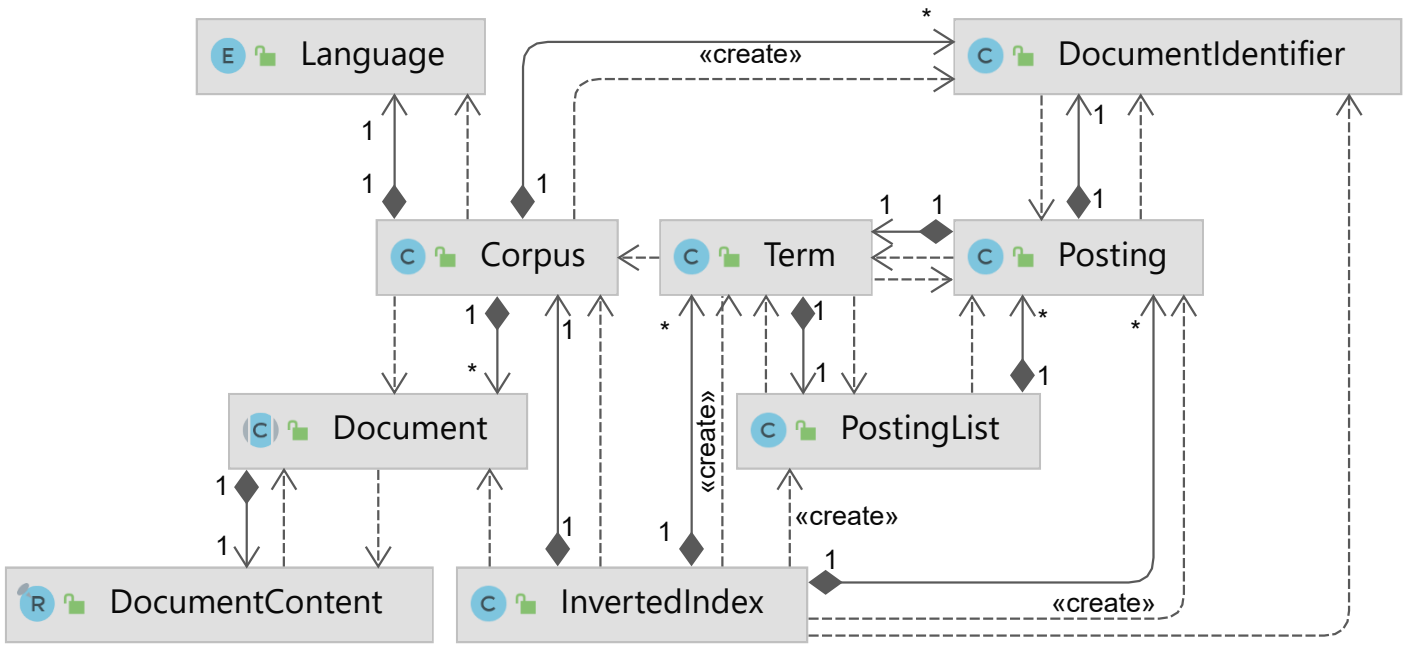
- Boolean model
- Data structures
- Queries





Boolean model

- Document
- DocumentIdentifier
- Corpus
- Posting
- PostingList
- Term
- InvertedIndex



Data structures

- Expected costs of the *SkipList*

	Concurrent HashMap	Patricia Trie	Skip list
Search by key	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log n)$
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log n)$
Pros	Fast access by key; thread safe	Scan by prefix (useful for permuterm index), no collisions	Forward pointers for fast scanning of the list
Cons	Cannot scan by prefix; in the *worst* case the cost is $\mathcal{O}(n)$ due to collisions; might require re-hashing	m accesses required	Worst cases: $\mathcal{O}(n)$ for search/insertion/deletion, and $\mathcal{O}(n \log n)$ for used space

m is the size of the key (in tries)
n is the number of elements in the data structure

Queries

- Hierarchical structure to evaluate aggregated queries
- AND, OR, NOT boolean queries
- Wildcard queries
- Spelling and phonetic corrections
- Queries can be inserted as strings and are parsed by the system

Summary

END

Thank you!

Matteo Ferfoglia
University of Trieste