ID1020 – Algorithms and Data Structures
Project 1 – VT14 P2

1 Organisation

The project is a programming task that is somewhat more involved than the lab. The results will be presented orally and a grade assigned based on the quality of the solution and understanding of the involved concepts.

The project itself is split into two parts, *Project 1 (P1)* and *Project 2 (P2)*, such that P1 forms the basis and P2 expands and improves on P1.

1.1 Dates

Submission Wednesday, Dec. 17th, 23:59 in Bilda

Presentation Thursday, Dec. 18th, time-slots will be assigned.

1.2 Goals

The project has the following goals:

- Work with the algorithms and data structures presented in the course
- Reason about usage patterns in a software system and leverage this to make implementation decisions
- Work on a real problem within the context of the course
- Get an idea of CFGs and their "real" applications

1.3 Requirements

For the project you will need the following:

- Java
- Maven

cf. Lab 2 for details.

1.4 Time

This depends heavily on your experience in writing code for more involved projects.
We can only recommend to start early, to get a feeling on how much time you will need to invest.
Make sure to think through the problem first before you throw yourself head first into coding. A good design from the beginning can save you many hours of wading through convoluted code later. Especially since you will have to reuse parts of your solution for P2.

1.5 Notations & Definitions

We denote the set of all natural numbers with $\mathbb{N} = \{1, 2, 3, 4, \ldots\}$ and $\mathbb{N}_0 = \{0\} \cup \mathbb{N}$. Similarly $\mathbb{R}$ denotes the set of all real numbers and $\mathbb{R}^+$ the set of all positive real numbers. The set of common complexity classes is denoted as follows:

$$\mathcal{C} = \left\{ f : \mathbb{N} \to \mathbb{R} \;\middle|\; f(n) \mapsto \left\{ \begin{array}{l} 1 \\ \log n \\ n \\ n \log n \\ n^r \text{ for some } r \in \mathbb{R}^+ \\ r^n \text{ for some } r \in \mathbb{R}^+ \\ n! \end{array} \right\} \right\}$$

Let $\mathbb{R}^{\mathbb{N}}$ refer to the set of all functions $f : \mathbb{N} \to \mathbb{R}$ then clearly $\mathcal{C} \subseteq \mathbb{R}^{\mathbb{N}}$.

2 Background

The goal of the project is to build a simple search engine for natural language text documents.
When completed the search engine will support the following operations:

- Index documents based on their content.
- Find and list documents which contain a single provided key word.
- Order search results by different properties (e.g. relevance, popularity).
- Use a *query language* to support more involved searches.

The documents themselves are provided in a specific format as part of the project framework and you don't have to worry about processing the raw files. However, it is important to notice that the given documents are english language texts of various types (e.g. short stories, newspaper articles, etc) and natural language texts differ in their treatment from more formal sources like programming languages, for example.

The specific documents we provide are from the so called Brown Corpus[1]. To give you some quick context, we provide a short summary of the field of natural language processing in section 2.1. Furthermore, the query language in the last task will require you to understand the concepts of parsing a limited syntax and to that end section 2.2 introduces the concept of context-free grammars and their relationship with parsing syntax.

2.1 Natural Language Processing

Natural language processing (NLP) is a field of computer science, artificial intelligence, and linguistics concerned with the interactions between computers and human (natural) languages. To a large extent the purpose of NLP is to allow computers to derive and subsequently leverage information from documents written for and by humans. This information could be anything from simply observing word usage pattern in certain kinds of documents, to understanding and classifying the content of said documents on a semantic basis. NLP has a number of sub-fields of which the following might be of interest to our search engine:

Part-of-speech tagging Given a sentence, determine the part of speech (POS) for each word. Many words, especially common ones, can serve as multiple parts of speech. For example, "book" can be a noun ("the book on the table") or verb ("to book a flight"). Some languages have more such ambiguity than others. Languages with little inflectional morphology, such as English are particularly prone to such ambiguity. POS tagging is a necessary pre-requisite in most cases for any kind of syntactical analysis.

Parsing Determine the parse tree (grammatical analysis) of a given sentence. The grammar for natural languages is ambiguous and typical sentences have multiple possible analyses. In fact, perhaps surprisingly, for a typical sentence there may be thousands of potential parses (most of which will seem completely nonsensical to a human).

Sentence breaking (also known as sentence boundary disambiguation)
Given a chunk of text, find the sentence boundaries. Sentence boundaries are often marked by periods or other punctuation marks, but these same characters can serve other purposes (e.g. marking abbreviations).

Word sense disambiguation Many words have more than one meaning; we have to select the meaning which makes the most sense in context. For this problem, we are typically given a list of words and associated word senses, e.g. from a dictionary or from an online resource such as WordNet.

If you are interested, http://en.wikipedia.org/wiki/Natural_language_processing give a good overview of the field.

[1] See http://en.wikipedia.org/wiki/Brown_Corpus for further reading, if you are interested.

2.2 Context-free Grammars

Context-free Grammars (CFGs) have their roots in formal language theory and were originally invented in an attempt to formalise the grammatical structure of natural languages. Since it has been shown that not all natural languages can be described by CFGs they have somewhat lost their importance in the NLP field. However, they retain immense importance in computer science as they form the basis of all programming languages (and also query languages, which is why we introduce them here).

Definition Formally a CFG is defined by a set of *production rules* of the form:

$$S \to s_1 s_2 \dots s_n \text{ for } n \in \mathbb{N}_0$$

where S is a *nonterminal* symbol and the s_i are either nonterminal or *terminal*, meaning that s_i is a *token* of the alphabet of the language. Terminal symbols may never appear on the left-hand side of a production rule and each nonterminal symbol has to appear on at least one left-hand side of some production rule.

Example Consider the following grammar for simple arithmetical expressions:

$$\begin{aligned}
&1 : E \to num \text{ where } num \in \mathbb{R} \text{ for example} \\
&2 : E \to E \cdot E \\
&3 : E \to \frac{E}{E} \\
&4 : E \to E + E \\
&5 : E \to E - E \\
&6 : E \to (E)
\end{aligned}$$

Also consider the following example sentence s in the language of arithmetical expressions $s = 2 \cdot (3 + 4)$.

Definition In order to show that a sentence is part of the language of a CFG we have to find a *derivation*, that starting with the start symbol and applying production rules to the right hand hand side, we arrive at the sentence.

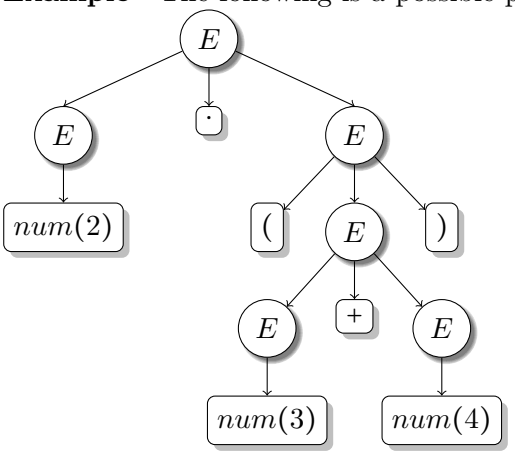
Example For the grammar above and sentence s the following would be a possible derivation:

E	start symbol
$E \cdot E$	rule 2
$num(2) \cdot E$	rule 1
$num(2) \cdot (E)$	rule 6
$num(2) \cdot (E + E)$	rule 4
$num(2) \cdot (num(3) + E)$	rule 1
$num(2) \cdot (num(3) + num(4))$	rule 1

Since we have derived s from the start symbol we have shown that s is part of the given grammar.

Definition While it is not difficult for a human to find a derivation that matches a sentence, a program would have try a (possibly large number) of different derivation until it by chance finds one that matches the input sentence. Since this is not feasible, programs usually apply the opposite process to derivation, which is called *parsing*. The result of the parsing process is a *parse tree* with the start symbol on top and the syntactical structure of the sentence represented in the structure of the tree. If parsing consumed all the symbols of a sentence that sentence is part of the language of the grammar.

Example The following is a possible parse tree for s:



Note The quick introduction to CFGs presented here is certainly far from exhaustive. There are important issues of ambiguity and associativity of operations we have not considered at all so far. However, it should be sufficient to perform the required tasks. If you require more information or are simply interested feel free to look at http://en.wikipedia.org/wiki/Context-free_grammar and follow references from there.

3 Helper Code

In order for you to focus on implementing the tasks, we are supplying a Java code skeleton which contains the basic functionality for you to start coding. It encapsulates the reading and parsing of the pre-tagged words from the Brown Corpus (see section 2 and supplies word by word using a *WordHandler* interface.

There are two main classes that you should know:

Word encapsulates the word itself and the POS (verb, noun, adverb, etc)

```
1 public class Word {
2     public final PartOfSpeech pos;
3     public final String word;
4 }
```

Attributes encapsulates the attributes that a word could have: a reference to the document in which the word was found and the occurrence (in number of preceding words) of the word in that document.

```
1 public class Attributes {
2     public final Document document;
3     public final int occurrence;
4 }
```

Document contains the name of the document and the popularity (think "number of views").

```
1 public class Document {
2     public final String name;
3     public final int popularity;
4 }
```

You are required to create your own *TinySeachEngine* which implements the *TinySearchEngineBase* inteface.

```
1 public interface TinySearchEngineBase {
2     //Build the index
3     public void insert(Word word, Attributes attr);
4
5     //Searching
6     public List<Document> search(String query);
7 }
```

As shown in the code above, you only have two methods:

insert: adds the word with the given attribute to your index.

search: returns the list of documents that matches the query.

3.1 Setup

First you need to create a maven project and add our helper project as a dependency in your *pom.xml* as follows:

```
1
2     <dependencies>
3         <dependency>
4             <groupId>se.kth.id1020</groupId>
5             <artifactId>tinySearchEngine</artifactId>
6             <version>1.0</version>
7         </dependency>
8     </dependencies>
9
10    <repositories>
11        <repository>
12            <id>sics-release</id>
13            <name>SICS Release Repository</name>
14            <url>http://kompics.sics.se/maven/repository</url>
15        </repository>
16    </repositories>
```

In your main method you need to call *Driver.run* which will starts reading the documents and will call *insert* in turns to build the index, also it will start a search REPL "read-eval-print-loop". To exit the REPL you can just write *exit* in front of search.

```
1     public static void main(String[] args) throws Exception{
2         TinySearchEngineBase searchEngine = new TinySearchEngine();
3         Driver.run(searchEngine);
4     }
```

Examples:

```
Building the index done in 47 seconds
Search:
```

As you can see first it prints that the indexing process is done in X seconds, and now on the next line you can enter your search query infront of search and hit enter for example, let's try to search for *nightmare*

```
Building the index done in 47 seconds
Search: nightmare
got 9 results in 52 microseconds
ca04
cc05
cf09
cl13
cl23
```

```
cp16
cp16
cp16
cr04
Search:
```

4 Tasks

4.1 Indexing – 30P

Build the index, your starting point is the insert method. You are allowed to use *Array*s, *ArrayList*s, *LinkedList*s but not *HashMap*s or *TreeMap*s. You are expected to analyse the complexity of your chosen data structures/algorithms during the oral examination.

For full points you implementation should allow for finding the documents for a word in $\sim \log n$ where there are n distinct words.

4.2 Simple Search – 20P

Implement the search for a simple query consists of only one word. Given a query "word" returns the list of documents that contains "word". You are expected to analyse the complexity of your chosen data structures/algorithms during the oral examination.

4.3 Sorting – 20P

Implement your own sorting algorithm to sort the resulted documents according to the query description in 4.4. You are expected to analyse the complexity of your chosen data structures/algorithms during the oral examination.

4.4 Queries – 30P

Implement your own query parser, your query language should support *union* and *ordering* as described by the following CFG:

$E \to T$	simple query
$E \to T$ *orderby Property Direction*	ordering
$T \to$ *word*	a search term
$T \to T\ T$	union
Property $\to$ *relevance*	how often do the search terms appear in a document?
Property $\to$ *popularity*	how popular is the document?
Property $\to$ *occurrence*	where in the document do the search terms appear?
Direction $\to$ *asc*	increasing order
Direction $\to$ *desc*	decreasing order

Note Do not confuse *union* with *join*! The query "word1 word2" should be executed in the following way (or equivalent):

1) Search for "word1" and place results in set $R_1 = \{doc_1, doc_2, doc_4\}$

2) Search for "word2" and place results in set $R_2 = \{doc_1, doc_3, doc_4\}$

3) Return $R_1 \cup R_2 = \{doc_1, doc_2, doc_3, doc_4\}$

Tip While the CFG above describes a very simple query language, make sure that you think your implementation through. In general parsers are a complicated field and you could have almost arbitrarily complicated implementations for this. But a simple and easy to read implementation could simply have one recursive function for each nonterminal that is basically a large switch-statement on the possible terminals of the production rules. Also remember that you will have to tokenise your query before parsing it, that is split it into a stream of tokens, similar to how you get the words from the documents in section 4.1. Try to avoid mixing the lexer (generating the token stream) and the parser (generating the parse tree) if possible. You should keep extendibility of your design in mind.
Take a look at `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html` if you need some ideas for the lexer.