CS210-001: Assignment 6
Fall 2019 (201930)

Due Date and Time: Thursday, November 7, 2019 at 11:20 AM

(100 marks) The term *search engine* is commonly used in reference to a computer program whose primary function is *information retrieval* (IR) from *documents* stored in some digital format. In IR, documents can be files stored locally on a user's computer and searched using operating system-specific tools (think Windows Explorer, Mac Finder, and the UNIX `grep` command), or on the World Wide Web (WWW) and searched using web-based tools (think Google and Bing). When using search engines on the WWW, most documents being searched are web pages, so we will use the term *pages* to refer to documents. The two main tasks performed by a search engine are *matching* and *ranking*. In the matching task, the pages are scanned to determine whether they contain any occurrences of user-specified *keywords* (i.e., terms representing the topics of interest). In the ranking task, any pages in which the keywords occur are scored according to relevance, typically ordered from most relevant to least relevant. The common challenge faced by all search engines is in determining what is relevant. Indeed, the algorithms used to determine relevance are closely guarded corporate secrets.

PART 1

The following extended example will give you some idea as to how the matching routine in a search engine works. Consider the three sample pages shown below.

Page: 1
```
|-----------------------|
| the cat sat on the mat |
|-----------------------|
```

Page: 2
```
|-------------------------|
| the dog stood on the mat |
|-------------------------|
```

Page: 3
```
|---------------------------------------|
| a cat stood while a dog sat on the cat |
|---------------------------------------|
```

The matching routine relies on *indexing* the pages based upon the occurrences of words on those pages. That is, each word is placed in an index, where, for simplicity in the first part of this example, the index merely identifies the page, or pages, on which a word occurs. An index for our three pages is shown below.

```
a     3
```

```
cat   1 3
dog   2 3
mat   1 2
on    1 2 3
sat   1 3
stood 2 3
the   1 2 3
while 3
```

For example, in this index, the word `a` occurs on page `3`, the word `cat` occurs on pages `1` and `3`, and the word `on` occurs on pages `1`, `2`, and `3`. This index can be used for querying pages to identify the occurrence of words in the pages.

PART 2

There are three kinds of queries: (1) single-keyword queries, (2) multiple-keyword queries, and (3) phrase queries.

Using a *single-keyword query*, you can identify those pages containing the occurrence of a single word. For example, if you query using the keyword `cat`, there is a *hit* (i.e., an occurrence of the word) on pages `1` and `3`, but not on `2`. Similarly, there is a hit for `while` on `3` and a hit for `the` on `1`, `2`, and `3`. Using a *multiple-keyword query*, you can identify those pages containing the co-occurrence of multiple words. For example, if you query using the keywords `cat dog`, there is a hit (i.e., an occurrence of *both* words) on page `3`. That is, in the index, the words `cat` and `dog` occur on a common page, namely, `3`.

A *phrase query* is a special type of multiple-keyword query in which there is a hit only when the words on the page occur consecutively in the same order as specified in the query. A phrase query is differentiated from an ordinary multiple-keyword query by enclosing the keywords within double quotes. For example, if you query using the phrase `"cat dog"`, there will be no hits because the words `cat` and `dog` do not appear consecutively anywhere on pages `1`, `2`, and `3`. If you query using the phrase `"the cat"`, there is a hit on pages `2` and `3`. However, there is clearly a problem with phrase queries and the way we have set up our index. That is, we have basically been "winging it" and looking back to the original documents to find out whether these phrases occur. The reason for this is that the index used for the single- and multi-keyword queries does not contain enough information to identify the occurrence of consecutive words.

The solution to the problem of identifying consecutive words is to not only store the page on which a word occurs, but also the *position* of the word within the page. The position of a word within a page is obtained by counting the words consecutively starting with `1`. For example, on page `1`, the words `the`, `cat`, `sat`, `on`, `the`, and `mat` are in positions `1`, `2`, `3`, `4`, `5`, and `6`, respectively. A revised index for our three pages that includes the positions of each word within the page is shown below.

```
a      (3, 1)  (3, 5)
cat    (1, 2)  (3, 2)  (3, 10)
dog    (2, 2)  (3, 6)
mat    (1, 6)  (2, 6)
on     (1, 4)  (2, 4)  (3, 8)
sat    (1, 3)  (3, 7)
stood  (2, 3)  (3, 3)
the    (1, 1)  (1, 5)  (2, 1)  (2, 5)  (3, 9)
while  (3, 4)
```

For example, in this revised index, the word `a` occurs on page `3` in positions `1` and `5`, the word `cat` occurs on page `1` in position `2` and on page `3` in positions `2` and `10`, and the word `on` occurs on page `1` in position `4`, on page `2` in position `4`, and on page `3` in position `8`. Now let's revisit the "`cat dog`" and "`the cat`" phrase queries. If you query using the phrase "`cat dog`", there will be a hit on the word `cat` because it occurs as the second word on page `1`, the second word on page `3`, and the tenth word on page `3`. However, there will be no hit on the phrase "`cat dog`" because the word `dog` does not occur as the third word on page `1`, or as the third or eleventh word on page `3`. If you query on the phrase "`the cat`", there will be a hit on the word `the` because it occurs on all three pages. The two hits of most interest are those on pages `1` and `3` where it is the first and ninth word, respectively. These hits lead to a hit on the phrase "`the cat`" because the word `cat` occurs as the second word on page `1` and as the tenth word on page `3`.

In the extended example, the revised index is actually a sorted list containing many unsorted lists. The sorted list consists of the words arranged in alphabetical order (i.e., `a`, `cat`, `dog`, etc.). The unsorted lists consist of the collection of page/position pairs associated with each word (e.g., `(1, 2)`, `(3, 2)`, and `(3, 10)` associated with the word `cat`). Although the unsorted lists appear to be sorted (and that is true), it is just a convenient coincidence that resulted from the way in which the sample pages were named.

PART 3

In this problem, you are required to develop a program for building the index for a simple search engine. Your program should be able to create an index like the revised index shown above in the extended example and it should be able to search the index using both a single-keyword query and a phrase query. The index should be implemented using the AVL tree and unsorted array-based list algorithms given on the CS210 Algorithms web page. The algorithms and data structures should be implemented as C++ classes. The *unsorted array-based list class* should be called `occurrenceType`. Its specification should be contained in a file called `occurrenceType.h` and its implementation should be contained in a file called `occurrenceType.cpp`. Each element of `occurrenceType` should be a `struct` consisting of variables called `page` and

`position`, where both `page` and `position` are defined as `int`s. The *AVL tree class* should be called `indexType`. Its specification should be contained in a file called `indexType.h` and its implementation should be contained in a file called `indexType.cpp`. Each element of `indexType` should be a `struct` consisting of variables called `word` and `occurrences`, where `word` is defined as a `string` and `occurrences` is defined as an `occurrenceType`. Your main program should be contained in files called `main.h` and `main.cpp`. Do not waste your time implementing any algorithms that you don't actually need.

So that you can take advantage of the convenient coincidence that resulted from the way in which the sample pages were named in the extended example, assume you will be indexing no more than 20 pages (text files actually) named `1` to `20`, respectively. Convert all alphabetic characters to lower case before inserting a word into the index.

PART 4

Once your index program is implemented, you can demonstrate that it works using the pages named `1` to `20` in the `/home/venus/hilder/cs210/assignment6/datafiles` directory as input. These pages contain course calendar descriptions for 20 different computer science courses. Demonstrate that you can:

a. Print the index, *without* the occurrences, in AVL tree format to show what the AVL tree representing the index actually looks like. You will have to modify one of the *depth-first* binary tree traversal algorithms to generate your AVL tree.

b. Print the index, *with* the occurrences, in list format like that shown above in the extended example. You will need to modify one of the *depth-first* binary tree traversal algorithms to generate your index.

c. Search for and print the pages and locations in which the following single keywords and phrases appear:

```
and
commutative
data
data structures
finite automata and formal language theory
history
machine-oriented
metrics
of
probabilistic reasoning and knowledge presentation
programming
qualified control of external specifications
topics include data
```

```
work
zero-based
```

After finding a single keyword, print the pages and locations in which it is found. For example, after finding the keyword `and`, your output should look like this:

```
and (2, 10) (7, 4) (13, 22) (20, 14)
```

After finding a phrase, print the pages and locations each keyword in the phrase is found, For example, after finding the phrase "`topics include data`", your output should look like this:

```
topics include data ((3, 15) (3, 16) (3, 17)) ((6, 7) (6, 8) (6, 9)) ((18, 27) (18, 28) (18, 29))
```

WHAT TO SUBMIT

If you are working alone, submit to UR Courses: (1) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles` and (2) a script file called `part4script` showing the compilation and execution of your program.

If you are working with a partner, *one of the partners* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%), (2) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles`, and (3) a script file called `part4script` showing the compilation and execution of your program. *The other partner* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contribution of the partners (should total 100%). Note that *both* partners need to submit the `partners` file.