Middle East Technical University — CENG 140

Department of Computer Engineering — **Spring 2015**

Take Home Exam 2

REGULATIONS

Due date: **31 May 2015**, Sunday, 23:59:59 (*No subject to postpone*)

Submission: Electronically. You will be submitting your program source code written in a file through the COW web system (see the specifications below for naming your file(s)). Resubmission is allowed (till the last moment of the due date), the last will replace the previous.

Team: There is **no** teaming up. The take home exam has to be done/turned in individually.

Cheating: <u>This is an exam</u>: all parts involved (source(s) and receiver(s)) get zero+parts will be subject to disciplinary action.

PROBLEM

General definition

In this assignment, our goal is simulating the "query result cache" of a typical search engine. The cache is a memory area where the search engine keeps the queries (and their results) it has seen previously; and hence, does not need to compute the answer each time a query is sent. For instance, the first time a user submits the query "justin bieber" its result will be computed and stored in the cache; and if another user sends the same query, the cached answer will be used. Of course, the cache has a finite capacity, so we can store the upcoming queries as long as there is free space. When all the available space is consumed, we will remove one of the cached queries to obtain the space for the new query. To decide on which query to be removed, we will count the number of times each query is submitted, and remove the query with the smallest count, as we discuss below.

Details and an example

Formally, assume a query string is submitted to a search engine. We keep a query table (typically using an **array**) that stores each query string and a pointer to corresponding node in the count-list (which will be discussed shortly). The size of the table is fixed as MAX_SIZE, which is the cache capacity. For simplicity, in this assignment, the table will be kept unsorted (see the example below) and the table will not store the query's actual answer (of course, in real life, this would not be the case).

A count-list is composed of several **doubly-linked lists** organized as follows. For each count value that is encountered, we will have a node in the list. Furthermore, for each count value, we will have a query-list including the nodes that represent the query strings with that count value (i.e., storing the queries asked that many times by the users).

To exemplify, suppose that the cache is initialized with the following information, which will be provided as a text file (“input.txt”). Note that, we show the query strings as single letters for brevity here; in the homework they will be one-word alphabetical strings (i.e., without space). The first line specifies the number of queries already in the cache. The following lines specify the query string and count separated with exactly one space character.

4
A 1
B 3
E 9
C 1

The corresponding query table and linked lists will look like as in Figure 1. We assume that the cache capacity (MAX_SIZE) is **5** in this example (so the last slot in the query table is empty).

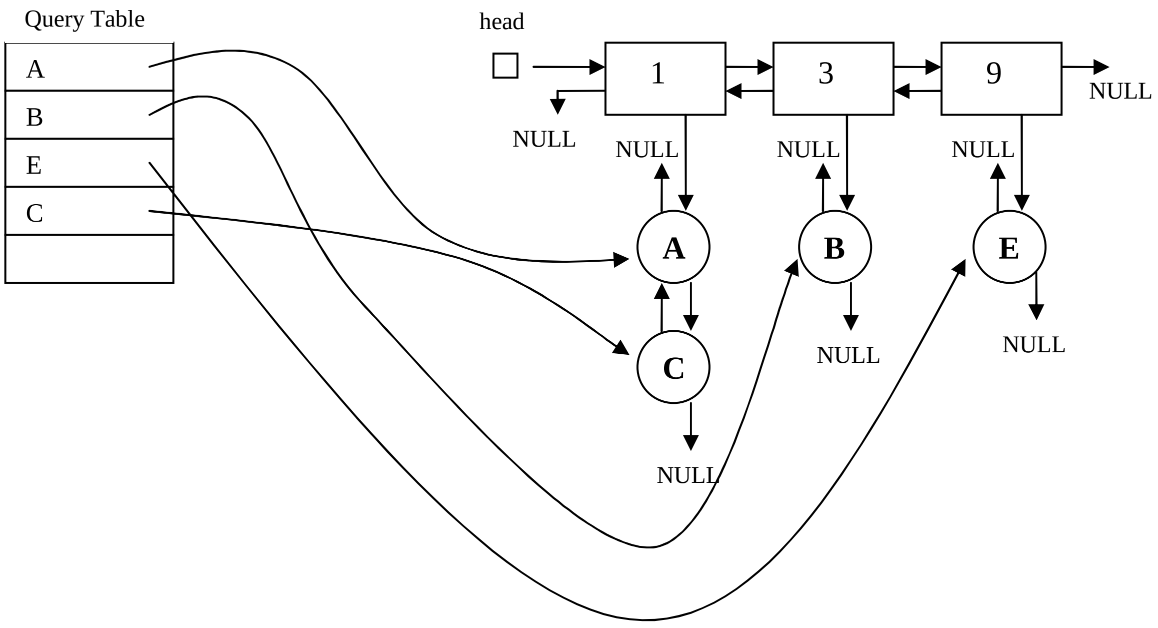

Figure 1. The cache contents after the initialization.

After the initialization information, the last line of the input file will include a sequence of “new” query submissions, as follows:

G A A Y

You should parse this line until you reach the “end of line”, each query is separated by a space character. Then, you need to update the cache contents for each query in the appropriate way, as described below.

If the new query is not in the table and there is space in the cache, then append it to the end of the table and arrange the query list of count-node 1 accordingly. Note that, in each query list, queries must be *sorted* in alphabetical order. Figure 2 shows the cache contents after seeing query G.

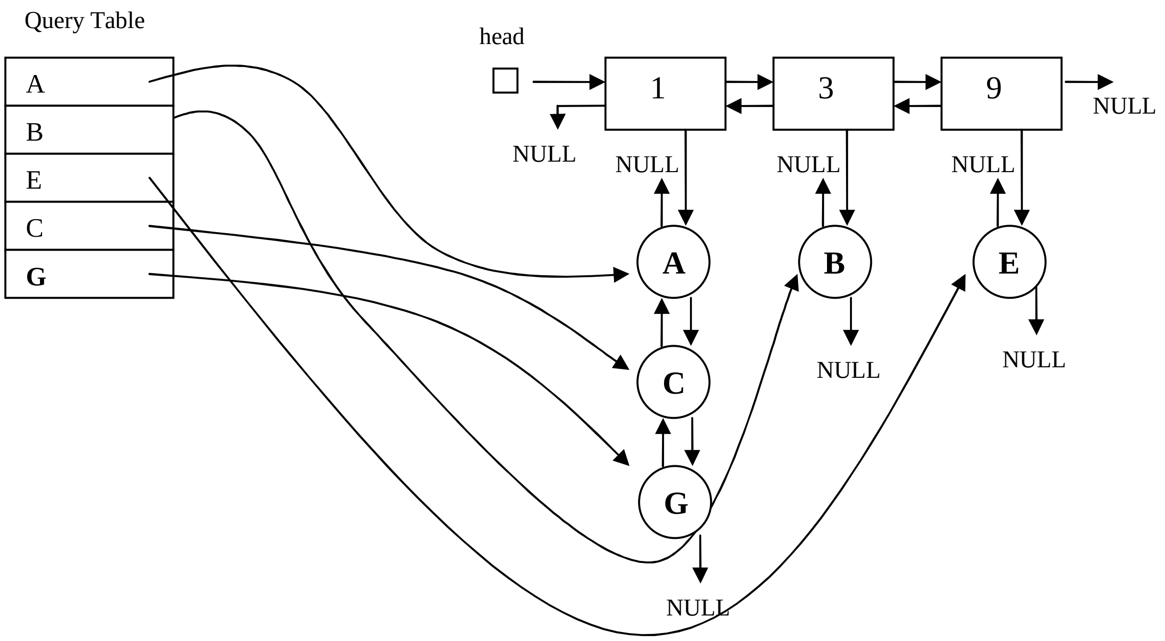

Figure 2. The cache contents after seeing query G.

If the new query is in the cache table, we simply update its count by moving it to the appropriate count-node's list. If the corresponding count-node is not in the list, it must be created; and if a count-node's list becomes empty, it musty be deleted. Figure 3 shows the cache contents after seeing query A; note that, as the count of A becomes 2, we need to create the count-node for 2. Figure 4 shows the cache contents after seeing the next query, which is again A; note that count-node 2 is now removed. (Except for query A, we don't show the pointers in the query table to make the figures simpler).

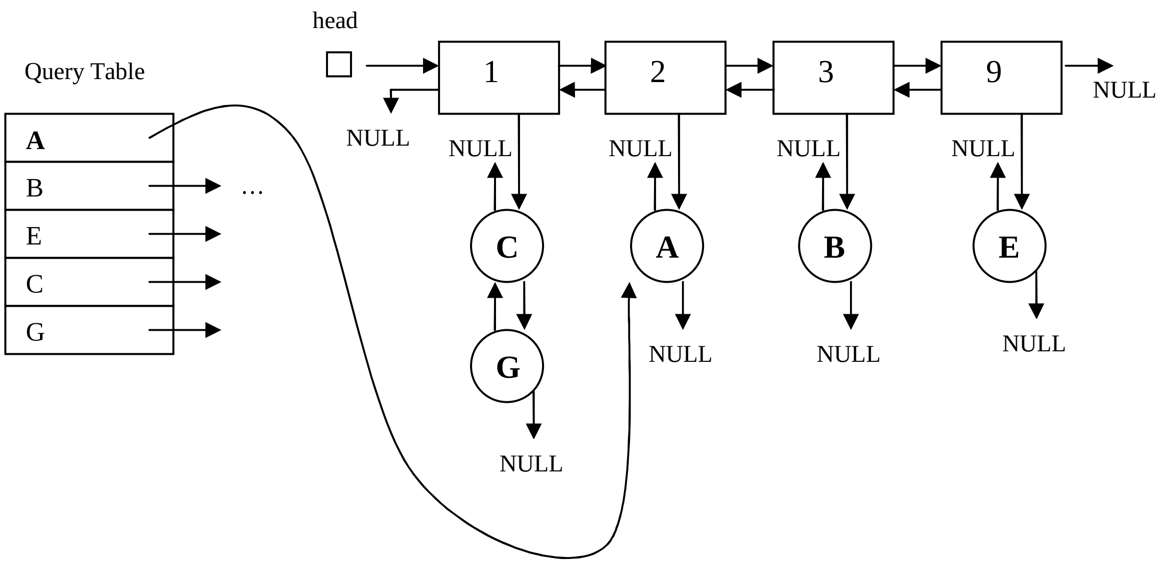

Figure 3. The cache contents after seeing query A.

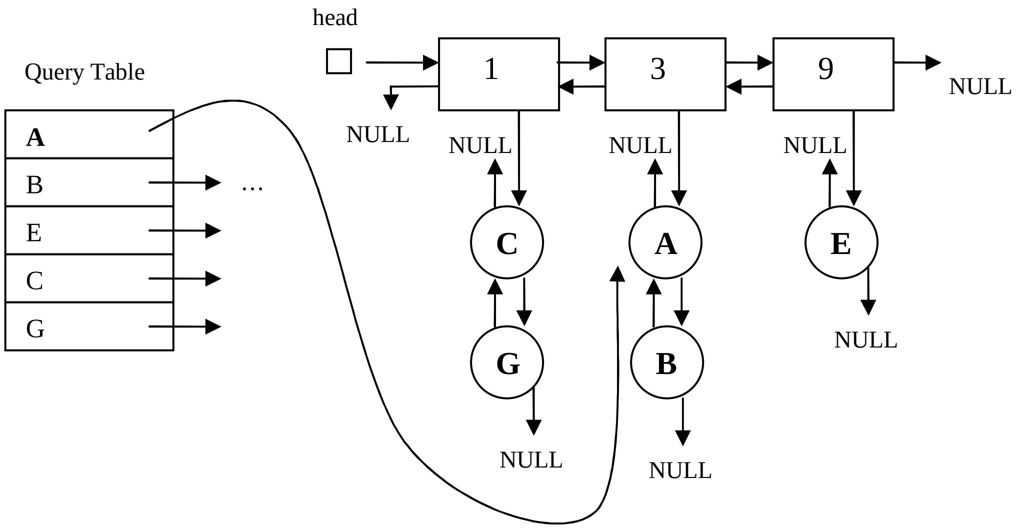

Figure 4. The cache contents after seeing query A one more time.

Finally, if the new query is not in the table and the cache is full, then remove the first query that has the *lowest* count (again, if the count-node's list becomes empty, also remove the count-node). Add the new query to this query's place in the query table and to the appropriate place in the query list of count-node 1 (as being a new query, it is seen only once). Figure 5 shows the cache contents after seeing the query Y. Note that query C is the first query in the lowest count (which is count-node 1 in this example, but need not to be so always), so it is removed from the list and table. Query Y is inserted to the table and the query list (kept sorted in alphabetical order).

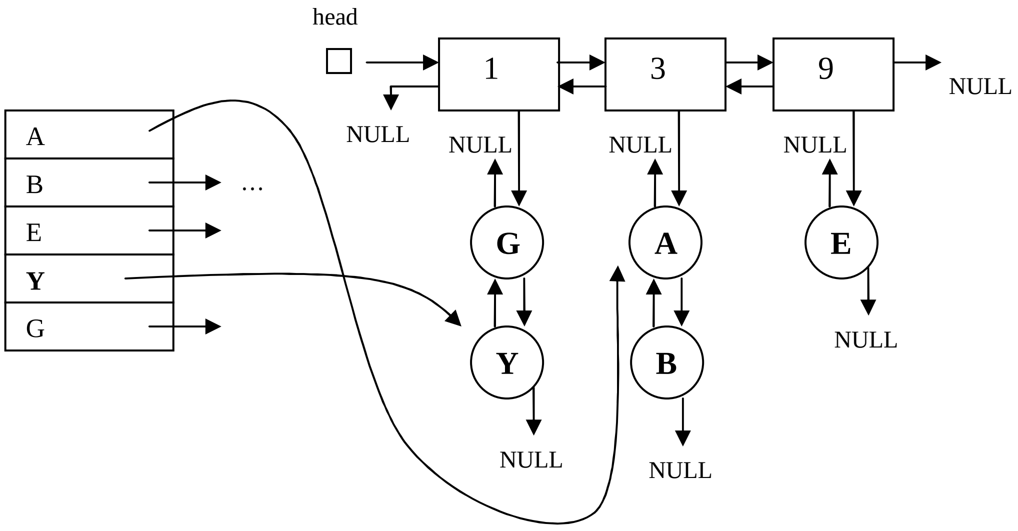

Figure 5. The cache contents after seeing query Y.

The output will be a text file (“output.txt”) that shows the cache contents by traversing the count-list and each query-list that is in the count-list, as follows (everything in each line separated by a single space):

```
1 G Y
3 A B
9 E
```

Specifications

1) For the query table, you must use an array in the way as exemplified above and use *linear search* to find whether a given query is in the table or not, and follow the pointer to access it (i.e., do **not** search for queries over the linked lists, which will be inefficient in real life). Assume cache capacity (i.e., MAX_SIZE) is 10.

2) The count list and each query list **must** be implemented using **doubly-linked lists** (trying to use an array will get **zero credit**).

3) The count list must be kept **sorted** with respect to the count values. The query lists must be kept **sorted** in the alphabetical order.

4) The query strings will be one word strings (like “book”) including only alphabetic characters without space, and will be all lower-case. The query strings will be at most 5 characters. The query strings used to initialize the cache will be all unique, so you don’t need to control this.

5) Input and output files must be in the format specified above. The input will be correct, there is no need for a sanity check. When executed, your code should read the text file “input.txt” and should write “output.txt”.