Homework - Homework 8

Independent work
Points: 120
Topics: hash tables (implementation and visual representation of the data structures).

You can organize your code however you want, but if you put some thought into it before you start it will go smoother. The Written part of this homework will ask you to show the data and it is meant to help you understand and debug your code. Start with the paper representation if the data and of what you want to be happening for each operation.

Programming task (100 points)

1. Implement a dictionary using a **hash table** with these specifications:

- For collision resolution it uses either quadratic hashing or double hashing.
- The table size should between 2 times and 4 times the number of lines in the dictionary file.
  - It cannot be more than 4 times the number of lines in the file.
  - It can be hardcoded to a number that works well for the data in the Spanish.txt file.
  - Hint: You may want to use a prime number for the size of your hash table.
- It must use a good function for hashing strings. You may use any resource for this function but you must cite it. You are allowed to use code you find on the web for this function alone, but you must cite it.

2. After the table is created display the statistics for building it. (Details will be provided below.)

3. Allow the user to use and update the dictionary through these operations: search, insert, delete, quit. Each operation (with all its needed data) will be given on one line as follows:

Line	Explanation and behavior
q	*q* - Quit Ends the loop that allows the user operations.
s word	*s* - Search Will search the dictionary for *word*. It will display the number of probes the search took. If found, will display the translation. If not found will display a message
d word	*d* - Delete Will search the dictionary for *word*. If found it will remove it. If not found it will display a message.
i word newTransl	*i* - Insert It will search the dictionary for *word*. If found, it will **update** the existing translation by appending ; and *newTransl*. If not found it will **add** this new entry to the dictionary. You can assume that *newTransl* is a single word of at most 100 characters. For example `el gato` cannot be given as a new translation for insertion in this loop because it consists of two words.

You can assume that for all operations above, *word* is at most 100 characters, and the operation (*s/d/i/q*) will have at most 2 characters.

4. Report the statistics for the user operations. (E.g. if the user performed 10 dictionary operations (insert, delete, or search), report the average number of probes per operation.

Given files
Spanish.txt – dictionary data. This is a data file that contains English words and their corresponding Spanish translation. This file name should NOT be hardcoded. The program will **take the file name as user input**.
(Citation for the Spanish.txt file: "#Copyright 1999 The Internet Dictionary Project/Tyler Chambers #http://www.june29.com/IDP/". Original file: Spanish_orig.txt)

- The maximum size of a line in the dictionary file is 1000.
- The maximum size of a query word (that will be used as a key in the dictionary) is 100.
- The lines in the dictionary file (e.g. Spanish.txt) have the format: *word tab translation*
- The translation may contain more than one word like in this line for `abandon`:
  ```
  abandon   desamparar, desertar, renunciar, evacuar, repudiar
  ```

input1.txt – Your program must work with this file using input redirection (< input1.txt). It provides the dictionary file name and a sequence of user operations on the built dictionary.
run1.txt - sample run file.

Dealing with duplicates

The dictionary file may contain duplicates: there may be several lines for the same word. E.g. in Spanish.txt:

```
aback     hacia atras
aback     hacia atrás,take aback, desconcertar. En facha.
aback     por sopresa, desprevenidamente, de improviso
aback     atra/s[Adverb]
```

When trying to insert an entry in the hash table, if there is already another entry for that word, APPEND the new translation (from the new entry) to the existing translation in the hash table. For example after loading the Spanish file, when searching for `aback`, the translation will have the concatenation of all four translations, separated by a ; (shown yellow highlight):

```
hacia atras;hacia atr▒s,take aback, desconcertar. En facha.;por sopresa,
desprevenidamente, de improviso;atra/s[Adverb]
```

Table statistics

As you build the hash table, keep track of the following statistics and display them after the table is built (but before the user operations):

- the total number of items that could not be hashed. All items must be hashed, so this number should be 0 every time.
- the total number of probes required to hash all objects,
- the average number of probes required to hash all objects,
- the largest number of probes required by any of the keys in the file to be hashed,
- an array that at index i stores the count of keys that required i probes to be hashed. In the posted sample run, I printed the count of objects that required up to 100 probes.

Self-study: try a bad string hashing function together with linear hashing and see how bad the performance (expressed as average probes per operation) becomes. It should be easy to change at least the hash function (and keep the same open addressing method).

Other requirements

GLOBAL VARIABLES are NOT ALLOWED: neither for the hash table, nor for counting probes.

Program output must be in the same format as shown in the sample run file.

- The actual numbers for the statistics you get with your implementation will not be the same as in my sample run, as they depend on the choices made for the hash table.
- **The formatting must be the same,** especially the table formatting and the result of user operations. You must display the operation and data read e.g. `READ op:s query:aback` and `Will insert pair [cat,***************]` s.t. the grader can verify the behavior of your program. Since input redirection will be used, it is hard to tell what operation is performed at a specific time.
- **If your program does not show the output as mine, there will be a heavy penalty. The percentages shown below are out of the full homework score, not the individual item score**:
  - **20%** of hw score for the table formatting: show the lines, and table cells have the same width. This penalty is more than the points for computing this table because the table allows verification of correct program behavior.
  - **25%** of hw score (additional to the above 20%) for the output format in the user operations loop. This part must clearly show:
    - the operation,
    - the data,
    - the probe count and
    - the result or action.
  - You do not need to match my output perfectly, but the overall formatting for readability and the table formatting for nicely aligning the numbers. E.g. your table can be wider than mine, but it cannot be misaligned.

No memory errors. Running the program with Valgrind reports no errors: memory leaks or bad memory access. Suppressed errors are ok (no other errors). (See past homeworks and the Slides and Resources page for more info on Valgrind and how to run the code with it)

Points Distribution (100 points)

Hash Table – 25 points
Hash table statistics collected– 8 points
Table with query count for each probe number built and displayed - 12
Search – 10 points
Insert new – 10 points
Insert duplicate (update) - 10
Delete (not only supported, but search and insert function correctly after delete) - 20
Average probes per operation user operations: - 5

Penalties:

Valgrind reports memory errors (other than the suppressed ones): -25 points
Global variables used: 50% penalty
See output format penalties in the requirements section.

Written part (20 points) - A USEFUL visual representation of the data in your program and its behavior.

In a file called hw8_written.pdf draw a representation of how you implemented the hash table from the programming task. You must show the table and any other data from the program that is relevant to the hash table (e.g. if the table keeps the index to data that is in an array, show that array as well).

Show the behavior of each operation.

The drawing must be EXACT e.g. a pointer and the memory it points to should be shown by two different boxes. Include the code (structs, array declaration, memory allocation, freeing memory).

The document must cover these **5 sections and the code relevant to them**:

1. empty table,
2. table with two entries at indexes 1 and 3 inserted. Do NOT include the code to compute the hash value, but rather assume you have the index (e.g. 1) and include the code that deals with the data: copies, allocates memory, frees data, etc….
   - At index 1 put the entry ["horse","caballo"]
   - At index 3 put the entry ["cat", "el gato"]
3. table after operation: *i cat gato* . the entry at index 3 should now be: ["cat", "el gato;gato"]. Show how the old translation, "el gato", was updated to "el gato;gato". Eg. was memory reallocated?
4. table after deleting item at index 1. Show what happens with the deleted item. If the memory for it is freed, include the code that does that. If some other data is updated, show that.
5. freed table. Here include the code that frees the memory if needed.

REDO the table drawing for each section (do not show all changes on the same drawing). See this template answer document as a guideline: 2320_H8_written.pdf (docx).

This part will be graded based on the information included in it alone: drawings, code, brief explanations. The grader will NOT go back on forth between the drawings and you .c file to see if the drawing matches the code. **Drawings without code will get NO CREDIT.**

The grading will be subjective based on how clear the representation is.

What and How to Submit

The assignment should be submitted via Blackboard.

The answer for the written part should be submitted in a file called hw8_written.pdf.

Name the programming portion of this assignment hw8.c (all lowercase).

If your program doesn't compile on Omega using gcc -std=c99 -o hw8 hw8.c, include a Readme file with the compilation instructions.

Zip all your files in a folder named hw8, and then submit it on Blackboard.

As stated on the course syllabus, programs must be in C, and must run on omega.uta.edu.

IMPORTANT: Pay close attention to all specifications on this page, including file names and submission format. Even in cases where your answers are correct, points will be taken off liberally for non-compliance with the instructions given on this page (such as wrong file names, wrong compression format for the submitted code, and so on). The reason is that non-compliance with the instructions makes the grading process significantly (and unnecessarily) more time consuming. Contact the instructor or TA if you have any questions.

Back to Homework page.