Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Assignment 6

Distributed Wednesday, March 15 2017.

Due 3:00PM, Wednesday, March 22 2017.

Instructions:

1. The difficulty rating of this assignment is *gold*. Please read the course web page for more information about assignment difficulty rating, late policy (no late assignments are accepted) and grace periods before you start.

2. Each question in this assignment requires you to create one or more regular files on bluenose. Use the exact names (case-sensitive) as specified by each question.

3. C programs will be marked for correctness, design/efficiency, and style/documentation. Please refer to the following web page for guidelines on style and documentation for C programs of median size (which applies to this assignment):

   `http://web.cs.dal.ca/~mhe/csci2132/assignments.htm`

   You will lose a lot of marks if you do not follow the guidelines on style and documentation.

4. Create a directory named `a6` that contains the following files (these are the files that assignment questions ask you to create): `a6q1.c` and `a6q2.c` (the file `a6q2.c` is for bonus marks only). Submit this directory electronically using the command `submit`. The instructions of using `submit` can be found at:

   `http://web.cs.dal.ca/~mhe/csci2132/assignments.htm`

5. Do NOT submit hard copies of your work.

As mentioned in the first lecture of this course, this assignment is a gold-level assignment. It also has a small number of bonus marks, to reward people who work hard on assignments of gold level: Recall that we have the best 6 out of 7 policy for assignments.

Since this assignment is supposed to be more difficult than previous assignments, start working on this assignment early. As usual, the instructor and the TAs are there to help.

To help you avoid losing marks, a draft of the marking scheme for the first question can be found at (this is subject to minor changes):

`http://web.cs.dal.ca/~mhe/csci2132/assignments/a6q1.pdf`

Background: As a major part of this assignment, you are required to implement a data structure called hash table, which will be used in solutions to assignment questions. We give information on hash table here, including specific requirements for your implementation.

A hash table is a data structure that uses a hash function to map keys to their associated values. In a previous lecture, we used a very simple strategy to map a key to a value. In particular, in the example of Latin square given in class, we used an array named `visited` to map a key `i` to a true/false value, where `i` is from `1` to `size`.

This simple approach worked because we knew that the key is from a small range, i.e. `1` to `size`. What should we do if the key is from a large range? For example, what should we do if the key can be any integer? This is what a hash table is for.

Hashing: Hashing is a classic example of time-space tradeoff. Instead of building a large array for the range of keys (optimizing for running time) or performing search without constructing any data structure (minimizing the memory space), a hash function (here function means a mathematical function, not a C function) can be used to transform a given key, k, to an integer, i, in a small range M, and then the value associated with key k can be stored at index i of an array `values` of length M. To enable searching, we also store the key itself at index i of an array `keys` of length M.

One popular type of hash function is called modular hashing. We choose the array size M to be a prime number, and for a key k, we hash it to $k \mod M$. That is, following the notation in the previous paragraph, we have $i = k \mod M$, where M is a prime number.

For example, if we choose M to be 11, then we can hash the key 35 to $35 \mod 11 = 2$, and the key 50 to $50 \mod 11 = 6$. Let 2000 and 1000 be the values associated with keys 35 and 50, respectively, then in array `keys` (length 11) and `values` (also of length 11), `keys[2]` and `keys[6]` store 35 and 50, respectively, while `values[2]` and `values[6]` store 2000 and 1000 respectively.

This way we can easily access elements in `values` and `keys` using the hash function when we are given keys.

Collision: The above strategy seems simple, but there is one major problem. What should we do if the hash function hashes two different keys to the same index of the arrays? For example, using the hash function given above, keys 35 and 24 are both hashed to index 2. This is called collision.

There are many different ways of handling collision, and they can typically be shown to be efficient by mathematical analysis using probability. Here we implement one approach that is called *linear probing.*

In this approach, when we hash a key, k, to an index, i, we check whether `keys[i]` already stores a key. If `keys[i]` does not, we can store the key, k, and its associated value in `keys[i]` and `values[i]`, respectively. If `keys[i]` stores the same key k, then we update its corresponding value according to the particular application. What's interesting is the case in which `keys[i]` stores a different key. In this case, we try `keys[(i+1) mod M]`, `keys[(i+2) mod M]`, `keys[(i+3) mod M]`, ..., until we find an entry of `keys` that either does not store any key, or stores the same key, and perform insertion or update on this entry.

The above is the process of inserting keys and their associate values into a hash table. To make sure that you understand this, work on the following example by hand. Let $M = 11$. We insert the following keys $35, 22, 24, 21, 120$, and their associated values are $28, 99, 240, 11, 5$. Then, the content of the array `keys` is (`NONE` means that nothing is stored in that array entry):

```
22, 120, 35, 24, NONE, NONE, NONE, NONE, NONE, NONE, 21
```

The following is the content of the array `values`:

```
99, 5, 28, 240, NONE, NONE, NONE, NONE, NONE, NONE, 11
```

Searching: Knowing the approach of handling collision, we can design the algorithm of searching for a key, k, in the array `keys`. We first complete $i = k \mod M$. We then check the entries `keys[i]`, `keys[(i+1) mod M]`, `keys[(i+2) mod M]`, `keys[(i+3) mod M]`, ..., until we find an entry that stores key k, or till we find an entry that does not store any key. In the former case, we have located where the key k is stored and we can also retrieve its associated value. In the latter case, we conclude that key k is not present in the hash table.

In the example given above, use the above algorithm to search for the keys 35, 120 and 79, to make sure that you know exactly how this algorithm works.

Load factor: There is one final detail. If we keep inserting keys into the hash table, eventually all array entries will store keys and we will not be able to insert more keys. Even before that, the more entries the array stores, the more likely there will be collisions, which makes the performance worse.

The *load factor* is defined to be the current number of keys divided by M. In the above example, the load factor after we insert five keys is $5/11$.

In many implementations, when the load factor exceeds $1/2$, that is, when the number of keys is larger than $M/2$, they enlarge the array. We however have not learned how to dynamically allocate arrays. Thus, in this assignment, you are required to print an error message and terminates the program when the load factor exceeds $1/2$.

Some comments about performance: It can be mathematically proved that, as long as the load factor is not too large, the number of operations required to insert a key or search for a key is constant in the average case. Thus, on average, storing a key in a hash table for searching is faster than binary trees.

Preventing the load factor from being more than $1/2$ is sufficient in practice.

Implementation requirements: The following is required for your implementation of the hash table:

1. You must implement your own hash table from scratch. We will not award any marks otherwise.

2. Strictly follow the approaches described above to implement your hash table. If you use different hashing strategies, we will deduct some marks for the design part.

3. Set the value of M to be 31 when you submit your work. This is **important** as we will automatically test whether your program prints an error message and terminates when the load factor exceeds 1/2.

4. You will get any marks for this assignment only if you use the hash table implemented to solve questions in this assignment, and pass some test cases in the automatic testing.

5. You may be inclined to use external variables, as we have seen how to use them to implement a stack in class. However, to receive full marks, you are required to avoid using any external variables at all. You will lose a small number of marks if you do use external variables. See suggestions below.

6. To simplify your work, you can assume that all keys have positive values.

To avoid using external variables, we can define variables in the main function and pass them as parameters to other functions. It is not hard to see that, to implement a hash table, we need maintain arrays `keys` and `value`, their lengths and the current number of keys stored. The following is a possible prototype of the function that inserts a key into the hash table:

```
int insert(int key, int keys[], int values[], int len, int num_keys);
```

In this prototype, `key` is the key to be inserted, `len` is the length of arrays, `num_keys` is the number of keys stored before calling this function. This function returns the number of keys stored after insertion. Thus, to call this function, we can write a statement like:

```
  num_keys = insert(key, keys, values, LEN, num_keys);
```

In this statement, `LEN` is a macro defined to be 31.

There are other strategies of avoiding external variables. For example, you can also use pointers. Feel free to come up with your own strategy. You are not required to use exactly the same function prototype as given above.

Finally, one useful suggestion is that each time you implement a function for your hash table, write a simple program to test its correctness. **This will make debugging a lot easier.**

Questions:

1. [30 marks] This question asks you to solve the *heavy hitter* problem by making use of a hash table. In this problem, you are giving an array of int keys, in which each key may appear more than once. You are also given a threshold value t, which is a positive integer. You are asked to report all the keys that appear at least t times in this array, in the order they first appear in this array. Each such key should be reported only once. These keys are called heavy hitters.

   For example, if the array stores the following 40 integers:

```
105 50 3 55 100
532 550 100 55 100
550 55 100 240 42
100 99 105 333 120
333 333 100 315 333
120 333 240 550 333
302 100 333 240 333
42 55 42 55 3
```

and the threshold is 4, you are expected to report $55, 100, 333$, as each of these three keys occurs at least four times in the given array, and the first occurrence of 55 in the array is at index 3, prior to index 4 which corresponds to the first entry that stores 100, and so on.

This problem has applications in information retrieval and data mining. For example, if you treat these numbers as identifiers of keywords that appear in one document, then those heavy hitters are likely to be the most relevant keywords.

This problem can be solved by making use of a hash table. How? It is a good exercise to find out how, so think about this before reading the solution below that you are required to implement.

Solution to this problem: First, read all the elements in the array and insert them into a hash table. The values associated with each key should be the number of times this key occurs in the given array. More precisely, the first time you insert a key into the hash table, you set its associated value to be 1. When you insert the same key again into the hash table, you increment its value.

Then, scan the array again, and for each element in the array, search for it in the hash table to find out if its corresponding value is greater than or equal to the threshold. If it is, report it and set its associated value to 0 so that it will not be reported again.

This solution is efficient, as on average, its running time is linear in the array length.

No marks will be given if you do not implement this solution at all.

General Requirements: Use `a6q1.c` as the name of your C source code file.

We will use the following command on bluenose to compile your program:

```
gcc -std=c99 -o heavy a6q1.c
```

Your program should NOT print anything to prompt for user input. As many numbers have to be read from stdin to test your programs, you are expected to use input redirection to read the input from a file. More precisely, you can run your program using a command like this:

```
./heavy < seq.1
```

In the above command, the file `seq.1` is a plain text file of the following format: The first line always has two integers. The first integer is the length of the array, and the second is the value of the threshold. The keys stored in the array start from the second line of this file, separated by while-space characters such as spaces and newlines.

For the example given above, the corresponding file should be:

```
40 4
105 50 3 55 100
532 550 100 55 100
550 55 100 240 42
100 99 105 333 120
333 333 100 315 333
120 333 240 550 333
302 100 333 240 333
42 55 42 55 3
```

Your program should output the heavy hitters separated by single space characters. There is no space characters after the last integer printed, but there is a trailing newline symbol. If there are no heavy hitters (in the above example, if we set the threshold to be 20), then your program should print a newline symbol only.

For the example given above, the output should be:

```
55 100 333
```

Error Handling: Your program should print an appropriate error message and terminate in each of the following cases:

(a) The given length of the array or the threshold is not a positive integer.

(b) When you try to read an integer from the input file, you encounter an illegal character. For example, your `scanf` function encounters `a100` in the input file and cannot read it as a decimal number.

(c) The load factor of the hash table exceeds 1/2.

If there is more than one problem with user input, your program just has to detect one of them. You can assume that everything else regarding the input file is correct.

Testing: To help you make sure that the output format of your program is exactly what this question asks for, several files are provided in the following folder on bluenose to show how exactly your program will be automatically tested:

```
/users/faculty/prof2132/public/a6test/
```

In this folder, open the file `a6q1test` to see the commands used to test the program with three test cases. The output files, generated using output redirection, are also given.

To check whether the output of your program on any test case is correct, redirect your output into a file, and use the UNIX utility `diff` (learned in Lab 3) to compare your output file with the output file in the above folder, to see whether they are identical.

Since these output files are given, we will **apply a penalty to any program that does not strictly meet the requirement on output format**. This includes the case in which your output has extra spaces, or has a missing newline at the end. Note that **it is extremely important to use diff here, as the number of values in the output file is large**.

Hints: If you use `gdb` to debug your program, the following command can run your program with input redirection (say, the input file is named `seq.1`):

```
run < seq.1
```

2. [5 marks extra credit only] In this bonus question, you are still working on the heavy hitter problem. The only difference is that the heavy hitters should be output in decreasing order of their numbers of occurrences. That is, the heavy hitter that occur most frequently in the input array should be printed first, and then the second most frequent heavy hitter, and so on.

   For example, using the input file given in Question 1, the output should be:

   ```
   333 100 55
   ```

   When two or more heavy hitters have the same number of occurrences, then the one whose first occurrence in the array is before others should be printed first, and so on.

   Your solution should also make use of the hash table to get any marks. Clearly this also requires some sorting. To receive full marks for the design/efficiency part, you are expected to implement a sorting algorithm (do not use any library functions) that is more efficient than insertion sort / bubble sort, and avoid using too many additional temporary arrays. Hint: make use of the fact that the mergesort algorithm is a stable sorting algorithm.

   If stable sorting was not taught in the version of CSCI 1100 that you took, check the index of the textbook for CSCI 1100 for stable sorting and it is probably there. If this is not the case, read the following wikipedia article:

   `http://en.wikipedia.org/wiki/Sorting_algorithm`

   You are allowed to modify the mergesort program that the instructor gave out and use it in your program, but you need add documentation.

   General Requirements: Use `a6q2.c` as the name of your C source code file.

   We will use the following command on bluenose to compile your program:

```
gcc -std=c99 -o sortedheavy a6q2.c
```

In the directory `/users/faculty/prof2132/public/a6test/`, make use of the file `a6q2test` and other files to make sure that the output format of your program is correct.

The same error handling as those specified in Question 1 are expected.