Experimental run times of hash tables with open-addressing

The purpose of this project is to implement hash tables with open-addressing, to explore empirically their performance under different situations (load factor, hash function, etc.), and to compare it with theoretical estimates. The more careful your implementation and experiments, and the more insightful your comments about the measured and theoretical run time, the higher your grade. Reread chapter 11 "Hash tables" in the textbook (in particular, the sections listed in the course web page). The coding part of the project is a minor extension of labs 5 (hash with chaining) and 6 (open-addressing), so it will be a small part of the grade. What is important (assuming the code is correct) is to understand its performance empirically and in the context of the theoretical estimates.

Work to do

Throughout, the key values are assumed to be `int` values starting at 0, with special values -1 for NIL (empty slot) and -2 for DEL (deleted slot). We call m the size (number of slots) of the hash table, n the number of elements in the table and $\alpha = n/m$ the load factor. All runtimes are assumed to be for a single key search (successful or unsuccessful). Since the runtime for a search depends on the key being searched and the state of the table, in the experiments below you should report the **average** of all the key searches in each case (e.g. search for all the keys in S and report their average runtime).

Code

First, write code for the following programs in C or C++.

1. A function `GenerateKeys` to generate key values. This takes as input an integer m (the table size) and an even integer $n \leq m$ (the number of keys to generate), and creates the following three arrays:
   - S (keys in table): containing $n/2$ keys randomly in $1, \ldots, k$ for $k = m \times 10^5$.
   - D (keys to delete): containing $n/2$ keys randomly in $1, \ldots, k$, but different from those in S ($S \cap D = \varnothing$).
   - U (keys not in table): containing 10^5 keys randomly in $k+1 \ldots, 2k$.

   Make sure that each of these 3 arrays contains distinct key values (no repetitions). The keys in these arrays will be used in multiple experiments to insert, search and delete from a hash table.
2. Functions OA-INSERT, OA-SEARCH and OA-DELETE that implement insert, search and delete in a hash table with open addressing (section 11.4 in the textbook).

Then, write a function `RunExpt` that takes as input S, D and U and runs the following experiment, reporting the average search times:

1. Create a hash table T and insert in it the $n/2$ keys from S, then the $n/2$ keys from D. Now $\alpha = n/m$.
2. Run the following and measure for each the runtime:
   - Successful search: search the keys in S (runtime t_s).
   - Unsuccessful search: search the keys in U (runtime t_u).
3. Delete the keys in D from the table. **Note α is now different, namely $\alpha = n/2m$.**
4. Again, measure the unsuccessful and successful search runtime (t'_s and t'_u).

The result of this experiment (which corresponds to the specific key sets S, D, U it was run on) is the four average runtimes (t_s, t_u, t'_s, t'_u).

Finally, write a function `ClusterRunLengths` to compute the distribution (histogram) of cluster run lengths in the table (see p. 272 in the textbook about the problem of clustering in open addressing). This takes as input an open-addressing hash table T (of size m) and returns an array $R[1\,..\,m]$, where $R[i]$ is the number of clusters of length i in the table. A cluster of length i means i consecutive slots are occupied. For example, for a table T with $m = 11$ slots `XX..XX.XXX.` (where `X` means occupied and `.` not occupied), $R = \langle 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$, since there are 2 clusters of length 2 and 1 of length 3. If you prefer, since R contains mostly zeros, you can output instead a list of pairs (length,frequency) for the clusters found; in the example: $(2, 2), (3, 1)$.

Experiments

The experiments consist of reporting the runtimes (t_s, t_u, t'_s, t'_u) and the cluster run length histogram for given key sets S, D and U (generated according to different load factors α) and using different hash functions. Specifically:

- Fix the table size at $m = 2^{20}$.
- Repeat the experiment for $\alpha \in \{0.05, 0.1, 0.5, 0.8, 0.9, 0.95\}$, i.e., generate $n = \alpha m$ keys (half in S, half in D).
- For each experiment, try the following hash functions:
  - Primary hash function $h_1(k)$: try
    - the divisive method: $h_1(k) = k \mod m$
    - the multiplication method with $A = (\sqrt{5} - 1)/2$: $h_1(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$.
  - Secondary hash function $h_2(k)$: try
    - linear probing: $h_2(k) = 1$
    - double hashing: try $h_2(k) = 2k + 1$ and $h_2(k) = 2k$.

Analysis

Given the runtimes and cluster lengths that you have measured in the different experiments, analyze the results. Here are some ideas to explore:

- Plot, as a function of α, your runtimes (successful and unsuccessful search, before and after deletion) and the theoretical estimates of p. 274–276. Do the empirical and theoretical curves match well? If they do not, why is that? Consider whether the following factors make a difference in the runtimes: whether doing deletions or not; the hash function we use.
- Plot the histogram of cluster run lengths (again, in the different experiments). Comment on the results.

Further ideas: would the runtimes be very different if the keys are not chosen at random? How are the measured runtimes affected by other processes running in the computer? What happens with multicore or multiple-CPU systems? Etc. Be inquisitive.

Optional extensions

- When reporting runtimes (e.g. corresponding to searching for all the keys in S), report not just their average runtime but also the maximum, minimum and standard deviation.
- Repeat several times the same identical experiment (e.g. successful search with $\alpha = 0.1$) but with different keys (generated using a different seed). This will allow you to obtain error bars (since no two runs will take the exact same time) and include them in the plot.
- Implement a hash table with chaining using as hash function h_1 above and repeat the experiments. For what value of α does chaining give better search time than open-addressing?
- Repeat the experiments but generating keys in a nonrandom way (e.g. picked as $0, 1, \dots$). Comment on the results.
- Change other things: the value of m, the sizes of U and D, the order in which we insert the keys in the table (S then D, or D then S, or picking from both at random), etc.

What you have to submit

Submit the following (as a single file `cse100.tar.gz`, by email to the TA):

1. A brief explanation of what each group member did.
2. A report `cse100.pdf` of your results, including the plots and a concise but insightful discussion of your analysis.
3. Your C or C++ code for the algorithms.
4. Give the basic specs for the computer that you used (processor, clock frequency, memory size, number of processors, operating system). **Note**: all run times must be measured with the same computer and (approximately) the same workload. I suggest you run your experiments in a single-core single-CPU system, or that you make sure that the experiments' code runs in a single CPU (see `taskset` in Linux).

Practical details

For all practical questions, see the TA.

- To generate random keys, use `rand()` in C. Use `srand(seed);` to set the seed to initialize the pseudorandom number generator. Note: fix the seed to your student id# for all your experiments.
- Use the C `gettimeofday()` function to measure elapsed time, by bracketing your code with `gettimeofday()` calls. For example:

```
#include <time.h>

struct timeval start,end; // time structure for starting and ending time
gettimeofday(&start,NULL);
THE SEGMENT OF YOUR CODE THAT YOU WANT TO MEASURE TIME FOR
gettimeofday(&end,NULL);
fprintf(stdout,"%f\n",(end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec)/1000000.0);
```

  The quantum time in `gettimeofday()` is around 10 μs in Linux. This will be sufficient for computing average times over a large number of searches. If you need to compute the time of a single search (for example, to compute the maximum or minimum search time), the time resolution of `gettimeofday()` will not be sufficient. In this case, you could try the functions `clock_getres()` and `clock_gettime()`.