ECE 223: Computer Systems Engineering **Fall 2015**
Machine Problem 7 **Due: 11:59 pm, Tuesday, December 1**

The goal of this machine problem is to design and implement a table ADT using hashing with open addressing and three approaches to collision resolution policies. You will investigate the performance in equilibrium with additions and deletions. The equilibrium driver will demonstrate that very poor performance is possible when there are a large number of deletions, and that rehashing the table is required to restore the table and achieve the expected performance.

Use a modular design similar to the design for the binary search tree module from MP5. A **`lab7.c`** file is provided with two test drivers, and you will add additional drivers to test your design. You will need to develop your own test drivers. Be sure to also submit a **`makefile`** that correctly builds your program and produces an executable "**`lab7`**".

Two additional documents should be submitted. One is a **test plan** that describes details of your implementation and demonstrates, with a test script, how you verified that the code works correctly. The verification should include detailed prints from your program to show that you program operates correctly and has no memory leaks. The second document describes your **performance evaluation**, and should describe your drivers that test performance and compare the results to the equations developed by Standish.

Interface specifications

The Table ADT should have a header **`table_t`**, and should store pointers to memory blocks based on a key **`hashkey_t`**. For testing purposes use keys that are non-negative integers in the range 4:999,999,995 (that is, to model student identification numbers use 9-digit keys). A small number of the lowest and highest 9-digit numbers are not permitted as valid keys.

```
typedef void *data_t;  /* pointer to the information, I, to be stored in the table */
typedef unsigned int hashkey_t;  /* the key, K, for the pair (K, I) */
typedef struct table_tag table_t;
```

The following functions are required (following the definitions in Table 11.1 on page 454 in Standish's book).

`table_t *table_construct (int table_size, int probe_type);`

The empty table is filled with empty table entries (K_0, I_0) where K_0 is a special empty key distinct from all other nonempty keys. The table must be dynamically allocated and have a total size of **`table_size`**. The **maximum** number of (K, I) entries that can be stored in the table is **`table_size-1`**. The **`probe_type`** specifies the type of hashing and probing, and is a constant that is one of LINEAR, DOUBLE, or QUAD. Do not "correct" the **`table_size`** or probe decrement if there is a chance that the combination of table size and probe decrement will not cover all entries in the table. Instead we will experiment to determine under what conditions an incorrect choice of table size and probe decrement results in poor performance.

`table_t *table_rehash (table_t *, int new_table_size);`

Sequentially remove each table entry (K, I) and insert into a new empty table with the new table size. Free the memory for the old table and return the pointer to the new table. The probe type should remain the same. Do **not** rehash the table during an insert or delete function call. Instead use drivers to verify under what conditions rehashing is required, and call the rehash function in the driver to show how the performance can be improved.

`int table_entries (table_t *);` /* returns number of entries in the table */
`int table_full(table_t *);` /* returns 1 if table is full and 0 if not full. */
`int table_deletekeys(table_t *);` /* returns number of table entries marked as deleted */

`int table_insert (table_t *, hashkey_t K, data_t I);`

Insert a new table entry (K, I) into the table provided the table is not already full. Return 0 if (K, I) is inserted, 1 if an older (K, I) is already in the table (in which case update with the new I), or **−1** if the (K, I) pair cannot be inserted.

`data_t table_delete (table_t *, hashkey_t);`

Delete the table entry (K, I) from the table and return I. Return null if (K, I) is not found in the table. See the note on page 490 in Standish's book about marking table entries for deletions

`data_t table_retrieve (table_t *, hashkey_t K);`

Given a key, K, retrieve the pointer to the information, I, from the table, but do not remove (K, I) from the table. Return NULL if the key is not found.

`void table_destruct (table_t *);`

Free all information in the table, the table itself, and any additional headers or other supporting data structures.

`int table_stats (table_t *);`

The number of probes (that is, the number of key comparisons) for the most recent call to **`table_retrieve`**, **`table_insert`**, or **`table_delete`**

`hashkey_t table_peek(table_t *T, int index);`

This function is for testing purposes only; see the description of the equilibrium driver below for the only situation where this function should be used. Given an index position into the hash table, this function returns the value of the key if data is stored in this index position. If the index position does not contain data, then the return value must be zero. Make the first line of this function

```
assert(0 <= index && index < table_size);
```

`void table_debug_print(table_t *);`

Print table showing index and key. Also, show if an index is marked as deleted.

Hashing with open addressing

Implement the hash function **`h(K) = K % M`**, where **`M`** is the size of the table (i.e., **`M`** = **`table_size`**). Implement three options for probe decrements.

1. linear
2. double hashing by division, **`p(K) = max(1, (K/M) % M)`**
3. quadratic probing as described on pages 484 and 485 in Standish. Notice he provides the code on pages 485 and 486.

Building Tables

While the hash function used for this machine problem works well in many situations it can perform poorly in certain cases. We will test the following methods for building a table and determine how the performance compares to the predicted results assuming random keys.

Let **`min`**=4, **`max`**=999,999,995.
Let **`num_addresses`** equal the number of addresses to insert into the table for a particular trial (i.e., it is equal to $\lfloor \alpha M \rfloor$ and set by the command line options **`–m`** and **`–a`**).
Random addresses. Let **`range = max – min + 1`**. Calculate a random key using

```
key = (int) (drand48() * range) + min.
```

Generating **`num_addresses`** keys using this approach should provide performance that is similar to the performance formulas in Standish's book.

Sequential addresses. Pick a starting addresses using one random key and insert addresses in the table by sequentially increasing the value of the key. This table contains one large cluster, but there are no collisions during insertions

Folded addresses. Pick a starting address using one random key and insert in two batches. Half of the keys are inserted sequentially in one batch, and the second batch hashes to the same table locations as the first batch. That is, the keys in the second bath are offset from the first by the table size. This table also contains a large cluster, but half of the insertions also experience collisions.

Worst addresses. Insert addresses that hash to the same table location. All keys hash to the same table location. That is, $K_2 = K_1 – M$. If there are a large number of keys and the table size is large, then it is possible to wrap around the space available for key addresses. So, multiple offsetting batches may be inserted when the table size is large. This approach creates one cluster when inserting keys with linear probing. You will investigate double and quadratic hashing to discover which collision resolution approach performance well even for worst case addresses.

Drivers

Retrieve driver (-r)

The load factor, α, for a hash table of size M with N occupied entries is defined by $\alpha = N/M$. For a given table size, M, and load factor, α, this driver measures the average number of probe addresses examined during a successful and unsuccessful search.

Recall that if we search for a key K already known to be in the table, the number of probes required to locate it will be exactly the same as the number of probes required when it was inserted in the first place. Thus, the average number of probe addresses examined during a successful search equals the number of probe addresses examined while building the initial table. A random key is generated for experimenting with unsuccessful searches. With high probability the key will not be found in the table (since the range of keys is much larger than any table size you can easily test). In the unlikely case that the retrieve driver finds the key, the trial is discarded. You should compare the experimental results generated with this driver with the analytical formulas that predict the expected performance. See equations 11.7 and 11.8 on page 479.

Equilibrium driver (-e)

The equilibrium driver builds an initial table with random addresses for the specified load factor, α. The equilibrium phase consists of a number of trials as specified on the command line. For each trial, with probability 0.5, a key is randomly generated and attempt to insert it into the table is made. Or, with probability 0.5, an attempt to remove a key from the table is made. To find a key to remove, a random number between $0{:}M – 1$ is generated, and the **`table_peek`** function is used to look at that position in the table for a key. If a key is found, then this is the key to use in the **`table_delete`** command. If the key is not found, keep repeating the steps of generating a random table location and looking for a key until a key is found. If $\alpha > 0.1$, making a random peek into the table works well.

The driver prints the number of different types of inserts, and the number of deletes. It then performs a retrieve test to determine the search times using a design similar to the retrieve driver. The driver examines how search times change after the table has been churned. Next, the table is rehashed, and the retrieve test is preformed again to allow experiments to determine when churning the table leads to poor performance that is improved by rehashing the table.

Rehash driver (-b)

The rehash driver tests cases requiring the rebuilding of the hash table, including increasing the size of the hash table.

Testing

You must add at least two of your own drivers for testing, and you must report on your drivers in your test plan.

1. Your *test-plan* driver must print the hash table, and show sequences of insertions and deletions that illustrate how the keys are stored in the table, collisions are resolved, and deletions are managed. Make sure to test special cases such as boundary conditions (e.g., inserting into a full table, inserting a duplicate key, deleting from an empty table, deleting a key not in table, inserting when using a probe decrement and table size combination that does not cover all addresses).
2. Illustrate one table size that is a small even number and show how your code reacts when using double hashing (it is okay to exit or abort since a poorly designed probe sequence and table size is a catastrophic error). Similarly, show the behavior for quadratic probing when the table size is not a power of 2. Explain what causes these combinations of table sizes and probe decrements to fail during an insert even though the table is not full. Use the Retrieve driver (-r).
3. You must create a driver, called *deletion driver*, which has a table size of 7 and uses linear probing. The driver must perform the following experiments (see also homework set 8, last two problems):
    a. Insert keys 5, 12, 11, and 19 into an empty table. Remove keys 5, 8, and 12. Insert keys 19 and 26. Key 19 must not occur in the table twice. Key 26 must be stored in location 5.
    b. Insert keys 7, 8, 9, 10, 11, 12 into a new empty table. Remove keys 7, 8, 9,10. Insert keys 13, 14. Retrieve 16 (not found in table).

   For each test, your output must show the contents of the table after each of the three groups of operations. Your output must match the solutions found in the homework set.
4. Show that your code does not have any memory leaks.

Performance evaluation

1. Using random addresses and the retrieve driver, test the three probe decrement options with two different table sizes:
   M = 65537 (a prime number is required for double hashing)
   M = 65536 (i.e., 2^{16} – quadratic probing requires a table that is a power of 2)

Try all load factors in this set {0.5, 0.75, 0.9, 0.95, 0.99}. How well does your experimental data match the predicted results and what are some reasons for any discrepancies?

2. Consider the four schemes for building a table (random, sequential, folded, worst), and for probing that is linear with M = 65537, double with M = 65537, and quadratic with M = 65536. Using the retrieve driver, consider trials with a load factor α=0.85. Find **both** the average number of successful and unsuccessful searches and create two tables with each table having a

format similar to the one shown below. For each table entry, do your results suggest that the performance is O(1) or O(*n*), or some other complexity class? Be sure to observe that all of the schemes perform well if the keys are random. But if the keys are not random than linear probing can perform poorly.

	random	sequential	folded	worst
Quadratic $M=2^{16}$				
Double $M=65537$				
Linear $M=65537$				

3. Using the equilibrium driver show that the performance degrades when the table size is large and the number of trials is large. Show that after the table is rehashed, the performance returns to values similar to experiments using the retrieve driver.

Notes

Command line arguments must be used to modify options for the table ADT and parameters for the test drivers. Here are the required options, and you may add additional ones. If an invalid option is given, print a list of valid options and their default values.

`-m` table size (*M*)
`-a` load factor (α) (build a table by inserting (int) ($\alpha \times M$) addresses)
`-h linear | double | quad` : the type of probing decrement
`-i rand | seq | fold | worst` : the type of addresses for the initial table for the drivers
`-r` run retrieve driver to build a table that tests inserts and unsuccessful retrieves
`-e` run equilibrium driver to build a table and test inserts and deletes
`-b` run rehash driver to rebuild hash table while full
`-t` number of trials for drivers
`-s` seed for random number generator
`-v` verbose printing for drivers

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. All code and documentation files must be turned in to Blackboard. **Work must be completed by each individual student, and see the course syllabus for additional policies.**