ECE 2230: Computer Systems Engineering **Fall 2020**
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. You will implement three collision resolution policies: open addressing with linear probing and double hashing, and separate chaining. The performance evaluation will consider larger numbers of additions and deletions in equilibrium. The equilibrium driver will demonstrate that very poor performance is possible for open addressing when there are a large number of deletions, and that rehashing the table is required to restore the table and achieve the expected performance when using open addressing.

Use a modular design similar to the design for the binary search tree module from MP5. A **`lab7.c`** file is provided with three 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`**. While a key could be any number of arbitrary bytes, for testing purposes the drivers use keys that are ASCII strings from the set **`[a-z]`** and **`[0-9]`** to facilitate printing. The string length will usually be between 5 and 8 characters but your code should be able to accept arbitrarily short or long strings as keys.

```
typedef void *data_t;  /* pointer to the information, I, to be stored in the table */
typedef char *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 (e.g., NULL). 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 CHAIN. 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. See the construct function*/
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. Note that both **K** and **I** are pointers to memory blocks created by **malloc()**.

```
data_t table_delete (table_t *, hashkey_t);
```

Delete the table entry (K, I) from the table and return I (and free K). 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 when using open addressing.

```
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 for the most recent call to **table_retrieve**, **table_insert**, or **table_delete**. Due to a quirk in the way Standish has defined comparisons in his performance formulas, we have two different rules for counting probes. For open addressing the function must return the number of probes into the hash table (that is, the number of comparisons includes checking if a position contains an empty key (or is marked as deleted) even if the empty is represented by a NULL pointer). For separate chaining, count the number of key comparisons required to insert, retrieve or delete (and don't count tests for a NULL pointer).

```
void table_debug_print(table_t *);
```

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

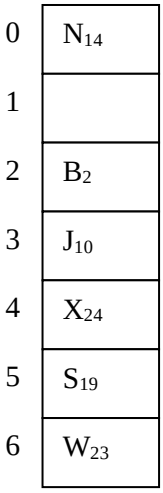
```
hashkey_t table_peek(table_t *T, int index, int list_position);
```

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. For separate chaining, return the key found in **list_position** at this index position. If the **list_position** is 0, the function returns the first key in the linked list; if 1 the second key, etc. If the **list_position** is greater than the number of items in the list, then return 0. Notice **list_position** is not used for open addressing. Make the first line of this function

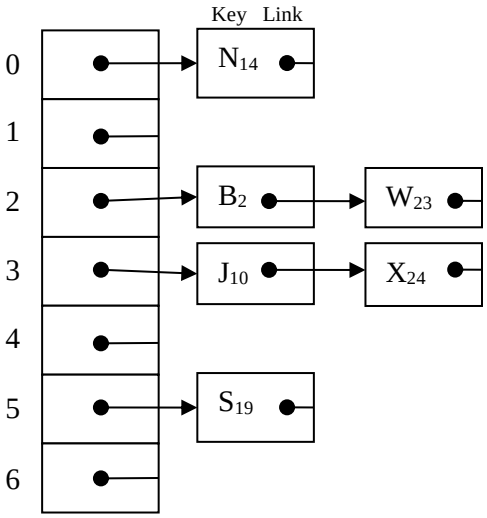
```
assert(0 <= index && index < table_size);
assert(0 <= list_position);
```

As an example of the **`table_peek`** command, consider these tables:

Open addressing



Separate chaining



`table_peek(T, 0, 0)` returns N_{14} in both tables. **`table_peek(T, 2, 1)`** returns W_{23} for separate chaining; for open addressing B_2 is returned since the **`list_position`** is ignored. **`table_peek(T, 2, 3)`** returns zero for separate chaining since there is not a third element in the linked list.

Hashing with open addressing

The hash function can use an algorithm of your choice. For double hashing a second hash function is required to calculate the probe decrement. Review the art of the hash function web site for many suggestions. (Notice some of the hash functions on this site are poor so pick a good one.)

http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx

In your comments you MUST cite the source of any code that is not your own. Since designing a hash function is a difficult art, you are permitted to adopt a well-known algorithm to generate a hash value. All other parts of your Hash Table ADT must be your own code developed using the example code provided in our textbook by Standish.

Hashing with separate chaining

Implement the same hash function as for open addressing. Place all keys that collide at a single hash address on a linked list starting at that address. Be careful to handle duplicate keys correctly!

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. (Except for separate chaining, in which case the number of key comparisons to retrieve a key will be one more than the number required to insert a key.) 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 an 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). Explain what causes the combination of table size and probe decrement to fail during an insert even though the table is not full. Use the Retrieve driver (-r).
3. Show that your code does not have any memory leaks.

Performance evaluation

1. Using random addresses and the retrieve driver, test the two open addressing options and separate chaining with a table size $M = 65537$ (a prime number). 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 two schemes for building a table (random and sequential), and the three types of hashing with $M = 65537$. 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
Separate chaining		
Double		
Linear		

3. Using the equilibrium driver and open addressing show that the performance degrades when the table size is large and the number of trials is large. For example, try a table size 65537 and 50,000 trials, and then repeat with 100,000 trials. Use the measure "Percent empty locations marked deleted" as a measure of how clogged the table is with deleted locations. Show that after the table is rehashed, the performance returns to values similar to experiments using the retrieve driver. Also explain why performance for separate chaining does not degrade compared to open addressing.
4. How much more memory does separate chaining require compared to the open addressing approaches? (Use the valgrind and the retrieve driver to find the heap size.)

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 | chain` : the type of probing decrement
`-i rand | seq` : 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, a makefile, one PDF document that contains your **test plan**, and one PDF document that contains your performance evaluation must be submitted to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#7. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes. If you do not get a confirmation email or the email reports an error, your submission was not successful. You must include

your files as attachments. Your email must be formatted as plain text (not html). You can make more than one submission but we will only grade the final submission. A re-submission must include all files. You cannot add files to an old submission.

Work must be completed by each individual student, and see the course syllabus for additional policies.