SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

COMP20007 Design of Algorithms, Semester 1, 2017

Assignment 2: Dynamic Hash Tables

Due: 12 noon, Monday 15 May

Overview

Hash table data structures are an incredibly popular and useful method for storing information, because they allow us to efficiently index a collection of records by something other than small integers (such as for a regular array). Using hashing to index a memory by strings or other more complex data types is extremely common, but hashing also enables us to index using numbers that are too large and too sparsely distributed to be used directly. For example, if we want to index some data using unsigned 64-bit integers, we can't use these numbers as array indices unless we have an array with 18 billion billion cells! Using hashing, we can map these large integers into a much smaller range for use as array indices in a hash table.

Moreover, a challenge encountered while designing a hash table is deciding how to grow a hash table to manage load and performance as many keys are inserted. When and how should the hash table grow in size to accommodate more keys?

For this assignment, you will implement and investigate various hash table structures for storing an arbitrarily large number of 64-bit integer keys. These keys will be stored alone, without corresponding 'values'. The hash tables will support 'insert' and 'look up' operations, and will grow in size dynamically as more keys are inserted.

The assignment consists of a 3-part coding component and a 2-part written report. Each task is described in detail in the sections below.

Provided Files

To assist with implementing this assignment, you're provided with a C program that reads and interprets commands to insert and look up 64-bit integers in a hash table. To see this program in action, download the following files from the LMS:

`Makefile`	containing instructions for compiling the program.
`main.c`	entry point and driver for the interpreter program.
`hashtbl.h/.c`	containing functions for using hash table data structures.
`inthash.h/.c`	containing hash functions for 64-bit integers.
`tables/`	folder containing various hash table modules.
`linear.h/.c`	complete implementation of a hash table using linear probing.
`xtndbl1.h/.c`	complete implementation of single-key extendible hashing.
`xtndbln.h/.c`	incomplete implementation of multi-key extendible hashing.
`cuckoo.h/.c`	incomplete implementation of cuckoo hashing.
`xuckoo.h/.c`	incomplete implementation of a hash table that combines cuckoo hashing and extendible hashing.

At this point, you should be able to compile the supplied code (listed above) by running `make`.

After compiling, you should be able to run the program with the command:

```
./a2 -t linear -s 16
```

When the program begins running, you should see the prompt 'enter a command' and a description of the available commands. These are summarised below:

format	description	example
`i number`	insert `number` to the hash table	`i 35`
`l number`	look up whether `number` is in the hash table	`l 100`
`p`	print the current contents of the hash table	`p`
`s`	print some statistics about the table state	`s`
`h`	print a list of available commands	`h`
`q`	quit the program	`q`

You should familiarise yourself with the interpreter program using the linear probing hash table and single-key extendible hash table, which are provided for you. Your task will be to implement the other three hash table types found inside the `tables` subfolder.

Once your project is completed, you will be able to execute it by running a command of the form

```
./a2 -t table_type [-s starting_size]
```

where

- `table_type` is a number or string representing the type of hash table to use:

part	table type	number	string
-	linear probing hash table	-	`linear`
-	single-key extendible hash table	-	`xtndbl1`
1	cuckoo hash table	1	`cuckoo`
2	multi-key extendible hash table	2	`xtndbln`
3	extendible cuckoo hash table	3	`xuckoo`

- `starting_size` is the initial size of the hash table, default 4.

For example, to run your solution to part 1 (Cuckoo Hashing) with initial size 8, you could use the command `./a2 -t cuckoo -s 8`, or the command `./a2 -t 1 -s 8`.

Generating input

For testing your programs on small example inputs, entering commands by hand one at a time is sufficient. For larger tests (including experiments for your written report), you will need to automatically enter a large sequence of commands. To achieve this, we provide a utility program `cmdgen.c` (command generator). Download this file from the LMS and compile it by running `make cmdgen`.

To generate some commands for testing your hash tables, run a command like:

```
./cmdgen 1000 100 > commands.txt
```

This will create a list of 1000 insert and 100 lookup commands inside a text file, `commands.txt`. You can use these commands as input to the interpreter program using a command like:

```
./a2 -t linear -s 16 < commands.txt
```

`a2` will follow the commands in `commands.txt`, instead of following commands typed in at runtime.

The provided `cmdgen.c` program is very basic. You are free to modify and extend it to generate different kinds of command sequences.

Coding Tasks

Part 1: Cuckoo Hashing (4 marks)

Complete `tables/cuckoo.c` with an implementation of a hash table that uses cuckoo hashing with two tables. An initial `struct cuckoo_table` is provided for you, but you may modify this as you see fit. You may find the code in `tables/linear.c` helpful. You will need to complete each of the functions described in `tables/cuckoo.h`:

- `new_cuckoo_hash_table(size)`: Create a hash table with two tables, each with space (initially) for `size` keys (so, 2× `size` keys in total). Return its pointer.
- `cuckoo_hash_table_insert(table, key)`: Insert `key` into `table`, if it is not there already. You should always begin by inserting `key` into the first of the two tables, using `h1()` from `inthash.h`. Upon a collision, the preexisting key should be moved to the second table using `h2()`. Upon collisions in the second table, the preexisting key should be moved back to the first table using `h1()` again, and so forth. Returns `true` if `key` was inserted, `false` if it was already in `table`.
- `cuckoo_hash_table_lookup(table, key)`: Returns `true` if `key` is in `table`, `false` otherwise.
- `cuckoo_hash_table_print(table)`: Implementation provided. If you modify the provided structs, you must ensure the output format is unchanged (it will be used to test your program).
- `cuckoo_hash_table_stats(table)`: Print any data you have gathered about hash table use. The output format is up to you.
- `free_cuckoo_hash_table(table)`: Free all memory allocated to `table`.

Your hash table must be capable of storing an arbitrary number of keys. Therefore, you must enable it to grow in size as more keys are added. You are free to decide on when and how to grow your hash table, for example you might like to double the size of each table whenever an insertion is about to fail because of a cycle, or whenever you encounter a very long chain of cuckoos.

Warning: a small bug could cause your hash table to continually grow in size, consuming a lot of memory. Also, the hash functions inside `inthash.c` only work for tables of bounded size. Whenever you increase the size of your table, you should use `assert` to ensure that your hash table is not larger than `inthash.h`'s `MAX_TABLE_SIZE` (see `tables/linear.c` for an example).

Part 2: Multi-key Extendible Hashing (4 mark)

The code in `tables/xtndbl1.c` provides a working implementation of extendible hashing where each bucket can store at most 1 key. Complete `tables/xtndbln.c` by adapting this code to provide an extendible hash table where each bucket can store up to `bucketsize` keys, where `bucketsize` is the parameter passed to the `new_xtndbln_hash_table()` function. Once again, initial structs are provided for you, but you may modify them as you see fit. You will need to complete each of the functions described in `tables/xtndbln.h`:

Note: in the context of this part, command-line parameter `starting_size` takes on a new meaning as the fixed size of each bucket, rather than the initial size of the internal table. The table of bucket pointers should grow like a regular extendible hash table.

- `new_xtndbln_hash_table(bucketsize)`: Create an empty extendible hash table with one slots, initially pointing to an empty bucket. The buckets in this hash table will contain up to `bucketsize` keys. Return the table's pointer.
- `xtndbln_hash_table_insert(table, key)`: Insert `key` into `table`, if it is not there already. To insert a key, take the right-most `depth` bits from the result of `inthash.h`'s `h1()` function, where `depth` is $\log_2$(current table size). The resulting value should be used as an index into the table

of bucket pointers, and the key should be inserted in the corresponding bucket. Returns `true` if `key` was inserted, `false` if it was already in `table`.

- `xtndbln_hash_table_lookup(table, key)`: Returns `true` if `key` is in `table`, `false` otherwise.
- `xtndbln_hash_table_print(table)`: Implementation provided. If you modify the provided structs, you must ensure the output format is unchanged (it will be used to test your program).
- `xtndbln_hash_table_stats(table)`: Print any data you have gathered about hash table use, for use gathering data for your report. The format of this function's output is up to you.
- `free_xtndbln_hash_table(table)`: Free all memory allocated to `table`.

Your hash table must be capable of storing an arbitrary number of keys. Therefore, you must enable it to grow in size as more keys are added. You should grow your hash table by splitting buckets when a key is to be inserted into a bucket that is already full. When splitting a bucket, the table may have to be doubled in size.

Warning: the same maximum table size warning from part 1 applies.

Part 3: Combining Cuckoo Hashing and Extendible Hashing (4 marks)

Complete `tables/xuckoo.c` with an implementation of a hash table that uses a combination of cuckoo hashing and extendible hashing, where each extendible hashing bucket store up to one key. Once again, initial structs are provided, and you can base your implementation off of `tables/xtndbl1.c`. You will need to complete each of the functions described in `tables/xuckoo.h`:

Note: in the context of this part, command-line parameter `starting_size` is ignored.

- `new_xuckoo_hash_table(size)`: Create an empty extendible hash table with two tables, each with one slot pointing to an empty bucket (separate buckets for each table). The buckets in this hash table will contain up to one key. Return the table's pointer.
- `xuckoo_hash_table_insert(table, key)`: Insert `key` into `table`, if it is not there already. Use `h1()` when inserting into `table`'s first table, and `h2()` when inserting into its second table. You should always begin by attempting to insert `key` into the table with fewer keys, or `table`'s first table if both have the same number of keys.

  When a key is hashed to a full bucket in the first table, before splitting the bucket, the preexisting key should be replaced with the new key, and the replaced key should be inserted into the second table. Likewise, if a key is hashed to a full bucket in the second table, the preexisting key should be replaced and inserted into the first table, and so forth.

  Returns `true` if `key` was inserted, `false` if it was already in `table`.
- `xuckoo_hash_table_lookup(table, key)`: Returns `true` if `key` is in `table`, `false` otherwise.
- `xuckoo_hash_table_print(table)`: Implementation provided. If you modify the provided structs, you must ensure the output format is unchanged (it will be used to test your program).
- `xuckoo_hash_table_stats(table)`: Print any data you have gathered about hash table use. The output format is up to you.
- `free_xuckoo_hash_table(table)`: Free all memory allocated to `table`.

Your hash table must be capable of storing an arbitrary number of keys. Therefore, you must enable it to grow in size as more keys are added. Therefore, you must enable it to grow in size as more keys are added. You are free to decide on when and how to grow your hash table. For example, you might like to stop moving between tables when you encounter a cycle, or after a very long chain of replacements. At this point, the final full bucket can be split like in a normal extendible hash table.

Warning: the same maximum table size warning from part 1 applies.

Coding Style (3 marks)

Three additional marks are available for overall good coding style. To receive these marks, your submission should meet the following criteria:

- **Readability (1 mark):** adequate comments describing each function and each logical section of code, clear and descriptive variable and function names, simple and easy-to-follow control structures (loops, if statements, etc.). No excessive use of global variables.
- **Approach (1 mark):** sensible functional decomposition (including using helper functions to avoid repeating code).
- **Safety (1 mark):** code compiles successfully on the School of Engineering student machines (a.k.a. dimefox) without alteration, contains no memory leaks, and runs without segmentation faults or infinite loops.

Written Tasks

The final two parts of the project require you to perform some experimentation and write a written report addressing the topics described below.

Your report must be **no more than three pages in length**, including all tables and graphs. You may find you do not need to use all three pages. Your report must contain your name and student number at the top of the first page. The file **must** be named `report.pdf` (you may use any document editor to create your report, but you must export the document in `.pdf` format for submission).

Part 4: Load Factor and Collisions (3 marks)

Modify the provided implementation of a linear probing hash table (`tables/linear.c`), and perform some experiments on the relationship between the hash table's load factor and the following statistics:

- Number of collisions — The number of keys which for which the first address they hash to is already occupied.
- Length of average probe sequence — The average number of slots which must be checked before a key is inserted into a free space.

Discuss these relationships in your report. You should support your discussion with one or more tables and/or graphs displaying the results of your experiments.

Note: You will need to add code to the functions in `tables/linear.c` to track the number of collisions and the length of the average probe sequence. You will also need to print these statistics from the `linear_hash_table_stats()` function.

Part 5: Keys Per Bucket (2 marks)

Using your implementation of an extendible hash table with a variable number of keys per bucket from part 2, perform some experiments on the relationship between the number of keys per bucket and the performance of the hash table after a sequence of insert and lookup operations.

Discuss this relationship in your report. You should support your discussion with one or more tables and/or graphs displaying the results of your experiments.

Note: you will similarly need to modify the `xtndbln_hash_table_stats()` function to print any collected statistics so that you can conduct performance experiments and gather results, for example CPU time of all inserts and lookups (see `tables/xtndbl1.c` for an example of measuring CPU time).

Bonus Challenge

Up to four bonus marks are available for students seeking an additional challenge. These marks will be awarded to students who successfully complete the following challenge.

Multi-key Extendible Cuckoo Hashing (4 marks)

Add a new type of hash table module to the `tables` folder, to `hashtbl.c`/`hashtbl.h`, and to `main.c`, such that this module is used when your program is run with a command like:

```
./a2 -t xuckoon -s 16
```

The module should be contained in two files `xuckoon.c` and `xuckoon.h`, and should provide a hash table implementation modelled after Part 3 (Combining Cuckoo Hashing and Extendible Hashing), but allowing up to a variable number of keys per bucket, like in Part 2 (Multi-key Extendible Hashing).

When a key is hashed to a full bucket in the first table (using `h1()`), a random key from among the preexisting keys should be replaced with the new key, and the replaced key should be inserted into the second table (using `h2()`). Likewise, if a key is hashed to a full bucket in the second table, a random preexisting key should be replaced and inserted into the first table, and so forth.

This process should continue up to a suitable maximum number of replacements, after which the final full bucket is split according to the rules of a normal extendible hash table.

Finally, add an extra page to the end of your report comparing the performance of Multi-key Extendible Cuckoo Hashing with Single-key Extendible Cuckoo Hashing (Part 3), Multi-key Extendible Hashing (Part 2), and Cuckoo Hashing (Part 1). You should modify the `hash_table_stats()` function for each module, record some interesting statistics, and perform experiments to support your discussion.

Sample Output

We provide some basic samples of correct output for each part of the assignment. Download the `samples` folder from the LMS. The files contain correct output from a selection of commands, as described in the table below.

Note that there may be multiple correct outputs for some inputs, depending on how you choose to grow your hash tables. Note also that these samples represent only a subset of the inputs your solution will be tested against after submission: you should use the interpreter and your own test cases to verify the correctness of your program, particularly for large sequences of insertions with more opportunities for table growth.

Filename	Command
`cuckoo-s4.txt`	`./a2 -t cuckoo  -s 4 < sample.txt`
`xtndbln-s4.txt`	`./a2 -t xtndbln -s 4 < sample.txt`
`xuckoo.txt`	`./a2 -t xuckoo < sample.txt`

To compare your program's output with the sample output files, you may find the following example commands helpful:

```
./a2 -t cuckoo -s 4 < sample.txt > my-output-1.txt
diff -sy cuckoo-s4.txt my-output-1.txt | less
```

These commands save the output of your program running cuckoo hashing on the sample command file in a file called `my-output-1.txt`, and then compare that output side-by-side with the sample, using `diff`. The output is displayed through `less`, which is useful for viewing long files.

Submission

Via the LMS, submit a single archive file (e.g. `.zip` or `.tar.gz`) containing **all files required to compile your solution (including the `Makefile`), plus your report (in `.pdf` format).** The archive should unpack into a folder, where the folder is named using your student number. Your submission should compile on the School of Engineering student machines (a.k.a. dimefox) without any errors, simply by running `make` inside this folder.

To make submission easier, we've added a `submission` target to the makefile. You need to locate the line `STUDENTNUM = STUDENT-NUMBER` and include your student number (for example, edit the line to `STUDENTNUM = 876543`). Then, running `make submission` will create the required archive file. If any files are missing, you will see an error. Don't forget to add any additional files you want to submit to the list, or they will not be included.

Submissions will close automatically at the deadline. As per the Subject Guide, the late penalty is 20% of the available marks for this project for each day (or part thereof) overdue. Note that network and machine problems right before the deadline is not a sufficient excuse for a late or missing submission. Please see the Subject Guide for more information on late submissions and applying for an extension.

Marking

The three coding parts of the assignment will be marked as follows. You will score full marks for a part if your solution produces correct output for all inputs tested. You will lose partial marks for minor discrepancies in output formatting, or minor mistakes in your solution. You will score no marks for a part if your solution crashes on certain inputs, or produces wrong answers.

The two written parts of your assignment will be marked as follows. You will score full marks for a clear and accurate solution that fully addresses the task, including at least some representation of your results (e.g. a graph or table). You will lose partial marks for minor inaccuracies, or misuse of terminology or notation, or failing to include your results. You will score no marks for a generic discussion with no evidence of experimentation.

Additional marks will be deducted if your report is too long (longer than three pages, or four pages including the bonus challenge), or is not in `.pdf` format.

Academic honesty

All work is to be done on an individual basis. Any code sourced from third parties must be attributed. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide for more information.