CMPT 201 Assignment 2

This assignment is worth 8% of your final grade

Overview

In this assignment, you will implement a dictionary-like data structure called a hash table as a C abstract data type (ADT). After validating its correctness, you will implement a program that makes use of your hash table.

I will be assuming basic familiarity with dictionary ADTs and hash tables. You may benefit from reviewing your CMPT 200 material on hash tables before proceeding with this assigment.

Getting started

As always, my strong recommendation is to do all your work in a CMPT 201-specific directory (name it A2), rather than just littering your home directory with unorganized files.

You are given a header file, `ht.h`, that describes the public interface for the hash table abstract data type. You will implement those functions prototypes in a file called `ht.c` as well as declare any additional "private" datatypes in another header, called `ht_impl.h`. The public interface must be complete and must not be changed! We will be testing your solution using this pre-determined API.

Deliverables

This assignment is to be delivered in two parts:

A2-Milestone#1 Due week 8 in your designated lab period. You will show your lab instructor any (partially) completed function or struct. Worth 5%.

A2-Final Submission Due Sunday March 16th at 23:59. You will submit your final implementation. See the "Submission" section for details.

Overview

By now you are very familiar with arrays, one of the fundamental derived types in C. Arrays allow *constant-time lookup* of elements given that element's index in the array.

```
>>> instructors = ["Ardy","Abdullah", "Dhara", "Salwa", "Mohammed"]
>>> instructors[2]
'Dhara'
```

You are also familiar, from your previous CS courses, with Python dictionaries. Dictionaries allow a similar sort of lookup, but with a non-integer arbitrary *key*. The upshot here is obvious: with a wellchosen hashing function and a sufficiently-large table, the average lookup time for keys in a hash table is `O(1)`, the same as indexing into an array!

```
>>> fav= {'Ardy':'MATLAB', 'Abdullah':'C', 'Dhara':'Python', 'Salwa':'C'}
>>> fav['Abdullah']
'C'
```

The downside, however, is that we can no longer assume anything about relative ordering between elements in a hash table, since internal organization happens based on how the hashing function slices and dices the key.

Data structure design is a surprisingly deep and fascinating topic. There are many approaches we could take based on the *user requirements* for a dictionary data structure. For instance, if we knew something about how users are going to use the structure (for instance, imagine a scenario where users never need to remove elements from a dictionary, or, a scenario where we know key values will fall into a relatively small range) then we can custom-tailor our structure to those needs. However! We are going to keep it simple in this assignment and assume nothing about how the user is going to use the hash table.

In this overview, and in the assignment, our keys will be C strings. However, in general, this need not necessarily be the case.

Our requirements for your hash table are:

- Mapping string keys to values of arbitrary type;
- Amortized `O(1)` lookup, insert, and deletion *for an arbitrary number of elements*;
- Safe, correct dynamic memory usage.

Design document

In Assignment 1, we laid out the requirements for the programs you wrote in reasonably complete detail. However, for this assignment, you will have to make more of those decisions on your own. This notion of you being given a *problem specification* and perhaps an *interface to implement*, but leaving internal design decisions to you, is something you'll start to see more of in upper-level Comp Sci courses and certainly out in the industry.

As a result, you may have to spend some time in the design portion of this assignment understanding the implementation possibilities of your hash table. The purpose of the design document is to codify and explain your decisions. (Often, in industrial settings, the design document lives on after the code has been written, either in internal wikis for people to read, or, even better, as comments inside the implementation code itself.)

Here are some topics which you should investigate and include in your design:

High-level operations and memory lifetimes

We have given you the public functions you must implement in the header file `ht.h`. Please do not change any of the declarations in that file! When we grade your solution we will be assuming that we can use your hash table in the expected way.

Take a moment to look at this file before proceeding; in particular, make sure you understand how the high-level requirements can be satisfied by the supplied interface.

Question: Note our use of the `const` keyword. Why is it reasonable for `ht_insert()` to take a nonconst `hashtable` as its argument when other functions, like `ht_lookup()` take a const `hashtable`?

Hash functions

A property of a good hashing function is that it should *uniformly* spread keys out. In other words, if we were to draw a histogram of a bunch of keys generated by the hash function, the histogram should converge to a uniform distribution. This should be the case even if the input keys themselves are not uniform in some way (say, if the string keys all begin with a common prefix or suffix, all share the same characters, or are of the same length).

Another property is *unpredictability*. This means that if we know the hash of some key, this doesn't help us figure out what the hash of another key might be. For *cryptographic hashing* this is of the utmost importance but for us it is not a strict necessity. One important way to achieve unpredictability is to ensure that all bits in the input key are taken into account.

A final property for us is *determinism*. Hashing the same key always needs to return the same hash value. Another way of saying this is that if `x` is equal to `y`, then `h(x)` is equal to `h(y)`.

Here is an example of a hashing function, with a TODO that we will figure out together. For the moment, assume TABLESIZE is a constant; your solution needs to be resizable and so in your implementation it will not be, though.

```
static long long unsigned hash(const char *s)
{     uint64_t ret = 5381;     char c;
while ((c = *s++)) {
  ret = (unsigned char)(c) + (33 * ret);     }

/* TODO: ret needs to be on [0, TABLESIZE) so we can use it as an
* index in the hash table.  How shall we do this? */

    return ret;
}
```

Designing a hash function from scratch is in general a very difficult problem and generally speaking should be left to cryptographers. You may find the choices for the constants `33` and `5381` to be nonobvious, and you are right! Some intuition here is that both numbers are prime, reducing the probability of the final computed value being coprime with `TABLESIZE`. Additionally, `33` has the property that CPUs can multiply it quickly; since `33 = 32 + 1`, `33 * x = (32 + 1) * x = 32 * x + x = (x << 5) + x`. That said, choosing hash constants is as much of an art as a science: the author of this hashing function, Daniel J. Bernstein, derived these particular values empirically by trying a bunch of candidate constant values and inspecting which minimized collisions in practice.

As the TODO suggests, we need a way of, given this slice-and-dice of the string's contents, getting a valid index into the hash table, between 0 and TABLESIZE, exclusive. Here are two potential ways of doing so:

The Division Method

The most straightforward way of doing so is simply dividing by `TABLESIZE` and taking the remainder. For this method to work optimally, though, the number of *common factors* between `ret` and `TABLESIZE` needs to be minimized in order to minimize collisions.

This puts constraints on possible sizes of the table. Generally this is done by making the size of the table prime.

Using the division method would require, during resize operations, ensuring that the new enlarged table's size remains prime. Additionally, taking the remainder requires executing a division instruction on the user's CPU - it might surprise you to learn that division is one of the more expensive instructions a CPU can execute, to the point of taking the remainder being the performance bottleneck for some real-world hash table implementations!

The multiplication method

(The following section is derived from Donald Knuth's *The Art of Computer Programming*, Vol. 3: *Searching and Sorting* (1973).)
There is another useful method that uses floating point multiplication and does not constrain the size of the table. It is based on the following observation: Take a number `x` and consider the *fractional part* of all the multiples of that number:

```
x, 2x, 3x, 4x, ...
```

If `x` is an integer, this is uninteresting: the fractional part is always 0.

If `x` is a rational number of the form `m/n`, this is marginally more interesting: the fractional part will cycle with period `n` through `0, 1/n, 2/n, ... (m-1)/n`. This sort of periodicity is not dissimilar to the division method when there are many common factors shared between `ret` and `TABLESIZE`.

However, if `x` is *irrational*, then it can be shown that this periodicity does not occur: the fractional parts will *all* be different, and will uniformly sample points on the unit interval. This means that if we scaled up the unit interval from `[0,1)` to `[0, k)`, irrespective of the value of k, we will have uniformly-sampled points on that interval, too.

This suggests a way to uniformly constrain `h` to `[0, TABLESIZE)`: Consider the following pseudocode, which uses the golden ratio as its irrational number:

```
double h, k = 1.61803398874989484820

h    = k * ret
h    = fractional part of h
h    = h * TABLESIZE
ret  = integer part of h
```

For the purposes of this assignment, you may either use the hashing function above or investigate another function and use that instead. You may also use the multiplication or division method for constraining the hash value to `[0, TABLESIZE)`, but if you use the former then your table size must always be prime, even after resizing.

Irrespective of your decision, in your design document, explain your choice. (As always: if you make use of any external resources to guide your thinking, cite them appropriately. Thank you!)

Collision resolution

Of course, it is possible to have two values we would like to insert into the hash table that hash to the same value. Indeed: it is a consequence of the Birthday Paradox that this is going to far more likely than what our intuition might suggest! As a result, we need a way of distinguishing between elements with the same hash.

Linear Probing

There are various designs for hash tables; however, in this assignment, you will be implementing, in particular, a hash table with *linear probing* used to resolve collisions. If two keys hash to the same index, then the table is to be probed until an empty position is found.

In your design document, indicate what data structure you have chosen to implement all the entries that correspond to a particular hash value. Provide type declarations as appropriate, and describe at a high level how they will be used in insertion, lookup, and deletion operations.

Table resizing

It can be shown (And, when you take CMPT 204, you may be asked to actually show it!) that hash tables maintain an average `O(1)` time complexity for inserts and lookups only so long as the table is no more than 2/3's full. Since your hash table needs to store an arbitrary number of keys and values, this means the size of your table may have to grow to accommodate many inserts.

To do so, your hash table will have to keep track of the current number elements in the table and it's capacity. When that fraction exceeds some threshold on an insert, the insert functionality will also trigger a table resize operation.

When this happens, your table will have to be reallocated to a larger size.

Question: What will need to happen to all the keys and values already inserted into the hash table? Will they necessarily reside in the same position as before the resize operation took place?

Hash Tables in Action: A postal code collator

You will now write a simple program that makes use of your hash table. We have provided you some input files containing the name of a city, town, or hamlet in Canada followed by a postal code in that place. Of course, larger communities will have more than one postal code.

```
$ cat postalcodes_20.in
Forest Grove,V0K1M0
Quebec,G1E7C2
Gloucester,K1B3L7
Saint-felicien,G8K1R8
North York,M3J2Z3
Montreal,H2Y3B6
Welland,L3C6T4
Montreal,H2Z1P3
Laval,H7A2E9
Oshawa,L1H1B8
Kinburn,K0A2H0
Montreal,H2G2G6
Winnipeg,R3L1K3
Williamswood,B3V1C5
Calgary,T3G3M9
Unionville,L3R3Y7
Leamington,N8H4W7
Markham,L3R0N8
Fort Erie,L2A3R4
Mississauga,L5T1E1
$
```

Write a program called `pcode` that reads such a file, the path of which you will provide as the argument to the program, and inserts every postal code into the hash table, keyed by its city. The value of each key will be a data structure that will store as many postal codes as the program has seen for that city. In other words, we will be inserting elements into the *value* of each key as the program runs.

Then, after the file has been loaded, on standard input, the user will enter names of various cities. The postal codes for those cities will be printed on standard output; if a city has more than one postal code, they should all be printed out, separated by commas, in the order that they appeared in the file. Therefore, your program will have to potentially *update* the values in the hash table to ensure that older values for an existing key are not overwritten.

Sample usage:

```
$ ./pcode ../datasets/postalcodes_20.in
Calgary
T3G3M9
Montreal
H2Y3B6,H2Z1P3,H2G2G6
Seattle
Markham
L3R0N8
$
```

Grading

Criterion	%
Milestone1	5%
Makefile (all targets present and working)	5%
Code quality and documentation	10%
Unit tests	5%
Hash table implementation	60%
pcode implementation	15%

Submission

Your solution must be archived and compressed before submission. The name of your tarball should be `cmpt201_assgn_2_X.tar.gz` X is your full initials. For example, Sam Porter Bridges would submit `cmpt201_assgn_2_SPB.tar.gz`.

Your tarball should contain:

1. makefile
2. testing directory including your testing strategy document
3. pcode.c
4. ht.c
5. ht.h
6. any .c and .h files you find necessary

Just remember any files pertaining to testing MUST go in the testing directory.

Getting started

You may be feeling right now that we have given you a lot to do and not a lot to go on. *Do not panic*. With a plan of attack you will be able to tackle this assignment!

Broadly speaking, I recommend building programs from the "inside-out". What I mean by that, concretely, is that you don't start tackling the postal code program until you've built the hash table, and that you don't start the actual hash table implementation until you have implemented the pertable element key-value data structure first.

I also recommend writing test programs *as you write the implementation*. A test program can be, as we've seen, a program that has expected standard output that you compare against with `diff`. Or, it can be a separate driver program that you link an object file against to call functions within.

In order to write tests as you write the implementation, you'll need a working `make testing` rule that compiles and runs your test programs. Running all your tests ensure that you haven't accidentally broken something that was working a moment ago!