Introduction

A hash table is a data structure which offers a fast implementation of the associative array API. As the terminology around hash tables can be confusing, I've added a summary below.

A hash table consists of an array of 'buckets', each of which stores a key-value pair. In order to locate the bucket where a key-value pair should be stored, the key is passed through a hashing function. This function returns an integer which is used as the pair's index in the array of buckets. When we want to retrieve a key-value pair, we supply the key to the same hashing function, receive its index, and use the index to find it in the array.

Array indexing has algorithmic complexity `O(1)`, making hash tables fast at storing and retrieving data.

Our hash table will map string keys to string values, but the principals given here are applicable to hash tables which map arbitrary key types to arbitrary value types. Only ASCII strings will be supported, as supporting unicode is non-trivial and out of scope of this tutorial.

API

Associative arrays are a collection of unordered key-value pairs. Duplicate keys are not permitted. The following operations are supported:

- `search(a, k)`: return the value `v` associated with key `k` from the associative array `a`, or `NULL` if the key does not exist.
- `insert(a, k, v)`: store the pair `k:v` in the associative array `a`.
- `delete(a, k)`: delete the `k:v` pair associated with `k`, or do nothing if `k` does not exist.

Setup

To set up C on your computer, please consult Daniel Holden's guide in the Build Your Own Lisp book. Build Your Own Lisp is a great book, and I recommend working through it.

Code structure

Code should be laid out in the following directory structure.

```
.
├── build
└── src
    ├── hash_table.c
    ├── hash_table.h
    ├── prime.c
    └── prime.h
```

`src` will contain our code, `build` will contain our compiled binaries.

Terminology

There are lots of names which are used interchangeably. In this article, we'll use the following:

- Associative array: an abstract data structure which implements the API described above. Also called a map, symbol table or dictionary.
- Hash table: a fast implementation of the associative array API which makes use of a hash function. Also called a hash map, map, hash or dictionary.

Associative arrays can be implemented with many different underlying data structures. A (non-performant) one can be implemented by simply storing items in an array, and iterating through the array when searching. Associative arrays and hash tables are often confused because associative arrays are so often implemented as hash tables.

Next section: Hash table structure Table of contents

Part 02: Hash table structure

Our key-value pairs (items) will each be stored in a `struct`:

```
// hash_table.h
typedef struct {
    char* key;
    char* value;
} ht_item;
```

Our hash table stores an array of pointers to items, and some details about its size and how full it is:

```
// hash_table.h
typedef struct {
    int size;
    int count;
    ht_item** items;
} ht_hash_table;
```

Initialising and deleting

We need to define initialisation functions for `ht_item`s. This function allocates a chunk of memory the size of an `ht_item`, and saves a copy of the strings `k` and `v` in the new chunk of memory. The function is marked as `static` because it will only ever be called by code internal to the hash table.

```
// hash_table.c
#include <stdlib.h>
#include <string.h>

#include "hash_table.h"

static ht_item* ht_new_item(const char* k, const char* v) {
    ht_item* i = malloc(sizeof(ht_item));
    i->key = strdup(k);
    i->value = strdup(v);
    return i;
}
```

`ht_new` initialises a new hash table. `size` defines how many items we can store. This is fixed at 53 for now. We'll expand this in the section on resizing. We initialise the array of items with `calloc`, which fills the allocated memory with `NULL` bytes. A `NULL` entry in the array indicates that the bucket is empty.

```
// hash_table.c
ht_hash_table* ht_new() {
    ht_hash_table* ht = malloc(sizeof(ht_hash_table));

    ht->size = 53;
    ht->count = 0;
    ht->items = calloc((size_t)ht->size, sizeof(ht_item*));
    return ht;
}
```

We also need functions for deleting `ht_item`s and `ht_hash_tables`, which `free` the memory we've allocated, so we don't cause memory leaks.

```
// hash_table.c
static void ht_del_item(ht_item* i) {
    free(i->key);
    free(i->value);
    free(i);
}


void ht_del_hash_table(ht_hash_table* ht) {
    for (int i = 0; i < ht->size; i++) {
        ht_item* item = ht->items[i];
        if (item != NULL) {
            ht_del_item(item);
        }
    }
    free(ht->items);
    free(ht);
}
```

We have written code which defines a hash table, and lets us create and destroy one. Although it doesn't do much at this point, we can still try it out.

```
// main.c
#include "hash_table.h"

int main() {
    ht_hash_table* ht = ht_new();
    ht_del_hash_table(ht);
}
```

Next section: Hash functions Table of contents

Part 03: Hash function

In this section, we'll write our hash function.

The hash function we choose should:

- Take a string as its input and return a number between `0` and `m`, our desired bucket array length.
- Return an even distribution of bucket indexes for an average set of inputs. If our hash function is unevenly distributed, it will put more items in some buckets than others. This will lead to a higher rate of collisions. Collisions reduce the efficiency of our hash table.

Algorithm

We'll make use of a generic string hashing function, expressed below in pseudocode.

```
function hash(string, a, num_buckets):
    hash = 0
    string_len = length(string)
    for i = 0, 1, ..., string_len:
        hash += (a ** (string_len - (i+1))) * char_code(string[i])
    hash = hash % num_buckets
    return hash
```

This hash function has two steps:

1. Convert the string to a large integer
2. Reduce the size of the integer to a fixed range by taking its remainder `mod m`

The variable `a` should be a prime number larger than the size of the alphabet. We're hashing ASCII strings, which has an alphabet size of 128, so we should choose a prime larger than that.

`char_code` is a function which returns an integer which represents the character. We'll use ASCII character codes for this.

Let's try the hash function out:

```
hash("cat", 151, 53)

hash = (151**2 * 99 + 151**1 * 97 + 151**0 * 116) % 53
hash = (2257299 + 14647 + 116) % 53
hash = (2272062) % 53
hash = 5
```

Changing the value of `a` give us a different hash function.

```
hash("cat", 163, 53) = 3
```

Implementation

```
// hash_table.c
static int ht_hash(const char* s, const int a, const int m) {
    long hash = 0;
    const int len_s = strlen(s);
    for (int i = 0; i < len_s; i++) {
        hash += (long)pow(a, len_s - (i+1)) * s[i];
        hash = hash % m;
    }
    return (int)hash;
}
```

Pathological data

An ideal hash function would always return an even distribution. However, for any hash function, there is a 'pathological' set of inputs, which all hash to the same value. To find this set of inputs, run a large set of inputs through the function. All inputs which hash to a particular bucket form a pathological set.

The existence of pathological input sets means there are no perfect hash functions for all inputs. The best we can do is to create a function which performs well for the expected data set.

Pathological inputs also poses a security issue. If a hash table is fed a set of colliding keys by some malicious user, then searches for those keys will take much longer (`O(n)`) than normal (`O(1)`). This can be used as a denial of service attack against systems which are underpinned by hash tables, such as DNS and certain web services.

Next section: Handling collisions Table of contents

Part 04: Handling collisions

Hash functions map an infinitely large number of inputs to a finite number of outputs. Different input keys will map to the same array index, causing bucket collisions. Hash tables must implement some method of dealing with collisions.

Our hash table will handle collisions using a technique called open addressing with double hashing. Double hashing makes use of two hash functions to calculate the index an item should be stored at after `i` collisions.

For an overview of other types of collision resolution, see the appendix.

Double hashing

The index that should be used after `i` collisions is given by:

```
index = hash_a(string) + i * hash_b(string) % num_buckets
```

We see that if no collisions have occurred, `i = 0`, so the index is just `hash_a` of the string. If a collision happens, the index is modified by the `hash_b`.

It is possible that `hash_b` will return 0, reducing the second term to 0. This will cause the hash table to try to insert the item into the same bucket over and over. We can mitigate this by adding 1 to the result of the second hash, making sure it's never 0.

```
index = (hash_a(string) + i * (hash_b(string) + 1)) % num_buckets
```

Implementation

```
// hash_table.c
static int ht_get_hash(
    const char* s, const int num_buckets, const int attempt
) {
    const int hash_a = ht_hash(s, HT_PRIME_1, num_buckets);
    const int hash_b = ht_hash(s, HT_PRIME_2, num_buckets);
    return (hash_a + (attempt * (hash_b + 1))) % num_buckets;
}
```

Next section: Hash table methods Table of contents

Part 05: Methods

Our hash function will implement the following API:

```
// hash_table.h
void ht_insert(ht_hash_table* ht, const char* key, const char* value);
char* ht_search(ht_hash_table* ht, const char* key);
void ht_delete(ht_hash_table* h, const char* key);
```

Insert

To insert a new key-value pair, we iterate through indexes until we find an empty bucket. We then insert the item into that bucket and increment the hash table's `count` attribute, to indicate a new item has been added. A hash table's `count` attribute will become useful when we look at resizing in the next section.

```
// hash_table.c
void ht_insert(ht_hash_table* ht, const char* key, const char* value) {
    ht_item* item = ht_new_item(key, value);
    int index = ht_get_hash(item->key, ht->size, 0);
    ht_item* cur_item = ht->items[index];
    int i = 1;
    while (cur_item != NULL) {
        index = ht_get_hash(item->key, ht->size, i);
        cur_item = ht->items[index];
        i++;
    }
    ht->items[index] = item;
    ht->count++;
}
```

Search

Searching is similar to inserting, but at each iteration of the `while` loop, we check whether the item's key matches the key we're searching for. If it does, we return the item's value. If the while loop hits a `NULL` bucket, we return `NULL`, to indicate that no value was found.

```
// hash_table.c
char* ht_search(ht_hash_table* ht, const char* key) {
    int index = ht_get_hash(key, ht->size, 0);
    ht_item* item = ht->items[index];
    int i = 1;
    while (item != NULL) {
        if (strcmp(item->key, key) == 0) {
            return item->value;
        }
        index = ht_get_hash(key, ht->size, i);
        item = ht->items[index];
        i++;
    }
    return NULL;
}
```

Delete

Deleting from an open addressed hash table is more complicated than inserting or searching. The item we wish to delete may be part of a collision chain. Removing it from the table will break that chain, and will make finding items in the tail of the chain impossible. To solve this, instead of deleting the item, we simply mark it as deleted.

We mark an item as deleted by replacing it with a pointer to a global sentinel item which represents that a bucket contains a deleted item.

```c
// hash_table.c
static ht_item HT_DELETED_ITEM = {NULL, NULL};


void ht_delete(ht_hash_table* ht, const char* key) {
    int index = ht_get_hash(key, ht->size, 0);
    ht_item* item = ht->items[index];
    int i = 1;
    while (item != NULL) {
        if (item != &HT_DELETED_ITEM) {
            if (strcmp(item->key, key) == 0) {
                ht_del_item(item);
                ht->items[index] = &HT_DELETED_ITEM;
            }
        }
        index = ht_get_hash(key, ht->size, i);
        item = ht->items[index];
        i++;
    }
    ht->count--;
}
```

After deleting, we decrement the hash table's `count` attribute.

We also need to modify `ht_insert` and `ht_search` functions to take account of deleted nodes.

When searching, we ignore and 'jump over' deleted nodes. When inserting, if we hit a deleted node, we can insert the new node into the deleted slot.

```c
// hash_table.c
void ht_insert(ht_hash_table* ht, const char* key, const char* value) {
    // ...
    while (cur_item != NULL && cur_item != &HT_DELETED_ITEM) {
        // ...
    }
    // ...
}


char* ht_search(ht_hash_table* ht, const char* key) {
    // ...
    while (item != NULL) {
        if (item != &HT_DELETED_ITEM) {
            if (strcmp(item->key, key) == 0) {
                return item->value;
            }
        }
        // ...
    }
    // ...
}
```

Update

Our hash table doesn't currently support updating a key's value. If we insert two items with the same key, the keys will collide, and the second item will be inserted into the next available bucket. When searching for the key, the original key will always be found, and we are unable to access the second item.

We can fix this my modifying `ht_insert` to delete the previous item and insert the new item at its location.

```
// hash_table.c
void ht_insert(ht_hash_table* ht, const char* key, const char* value) {
    // ...
    while (cur_item != NULL) {
        if (cur_item != &HT_DELETED_ITEM) {
            if (strcmp(cur_item->key, key) == 0) {
                ht_del_item(cur_item);
                ht->items[index] = item;
                return;
            }
        }
        // ...
    }
    // ...
}
```

Next section: Resizing tables Table of contents

Part 06: Resizing

Currently, our hash table has a fixed number of buckets. As more items are inserted, the table starts to fill up. This is problematic for two reasons:

1. The hash table's performance diminishes with high rates of collisions
2. Our hash table can only store a fixed number of items. If we try to store more than that, the insert function will fail.

To mitigate this, we can increase the size of the item array when it gets too full. We store the number of items stored in the hash table in the table's `count` attribute. On each insert and delete, we calculate the table's 'load', or ratio of filled buckets to total buckets. If it gets higher or lower than certain values, we resize the bucket up or down.

We will resize:

- up, if load > 0.7
- down, if load < 0.1

To resize, we create a new hash table roughly half or twice as big as the current, and insert all non-deleted items into it.

Our new array size should be a prime number roughly double or half the current size. Finding the new array size isn't trivial. To do so, we store a base size, which we want the array to be, and then define the actual size as the first prime larger than the base size. To resize up, we double the base size, and find the first larger prime, and to resize down, we halve the size and find the next larger prime.

Our base sizes start at 50. Instead of storing

We use a brute-force method to find the next prime, by checking if each successive number is prime. While brute-forcing anything sounds alarming, the number of values we actually have to check is low, and the time it takes is outweighed by the time spent re-hashing every item in the table.

First, let's define a function for finding the next prime. We'll do this in two new files, `prime.h` and `prime.c`.

```
// prime.h
int is_prime(const int x);
int next_prime(int x);
```

```c
// prime.c

#include <math.h>

#include "prime.h"



/*
 * Return whether x is prime or not
 *
 * Returns:
 *   1  - prime
 *   0  - not prime
 *   -1 - undefined (i.e. x < 2)
 */
int is_prime(const int x) {
    if (x < 2) { return -1; }
    if (x < 4) { return 1; }
    if ((x % 2) == 0) { return 0; }
    for (int i = 3; i <= floor(sqrt((double) x)); i += 2) {
        if ((x % i) == 0) {
            return 0;
        }
    }
    return 1;
}



/*
 * Return the next prime after x, or x if x is prime
 */
int next_prime(int x) {
    while (is_prime(x) != 1) {
        x++;
    }
    return x;
}
```

Next, we need to update our `ht_new` function to support creating hash tables of a certain size. To do this, we'll create a new function, `ht_new_sized`. We change `ht_new` to call `ht_new_sized` with the default starting size.

```c
// hash_table.c
static ht_hash_table* ht_new_sized(const int base_size) {
    ht_hash_table* ht = xmalloc(sizeof(ht_hash_table));
    ht->base_size = base_size;

    ht->size = next_prime(ht->base_size);

    ht->count = 0;
    ht->items = xcalloc((size_t)ht->size, sizeof(ht_item*));
    return ht;
}


ht_hash_table* ht_new() {
    return ht_new_sized(HT_INITIAL_BASE_SIZE);
}
```

Now we have all the parts we need to write our resize function.

In our resize function, we check to make sure we're not attempting to reduce the size of the hash table below its minimum. We then initialise a new hash table with the desired size. All non `NULL` or deleted items are inserted into the new hash table. We then swap the attributes of the new and old hash tables before deleting the old.

```c
// hash_table.c
static void ht_resize(ht_hash_table* ht, const int base_size) {
    if (base_size < HT_INITIAL_BASE_SIZE) {
        return;
    }
    ht_hash_table* new_ht = ht_new_sized(base_size);
    for (int i = 0; i < ht->size; i++) {
        ht_item* item = ht->items[i];
        if (item != NULL && item != &HT_DELETED_ITEM) {
            ht_insert(new_ht, item->key, item->value);
        }
    }

    ht->base_size = new_ht->base_size;
    ht->count = new_ht->count;

    // To delete new_ht, we give it ht's size and items
    const int tmp_size = ht->size;
    ht->size = new_ht->size;
    new_ht->size = tmp_size;

    ht_item** tmp_items = ht->items;
    ht->items = new_ht->items;
    new_ht->items = tmp_items;

    ht_del_hash_table(new_ht);
}
```

To simplify resizing, we define two small functions for resizing up and down.

```c
// hash_table.c
static void ht_resize_up(ht_hash_table* ht) {
    const int new_size = ht->base_size * 2;
    ht_resize(ht, new_size);
}


static void ht_resize_down(ht_hash_table* ht) {
    const int new_size = ht->base_size / 2;
    ht_resize(ht, new_size);
}
```

To perform the resize, we check the load on the hash table on insert and deletes. If it is above or below predefined limits of 0.7 and 0.1, we resize up or down respectively.

To avoid doing floating point maths, we multiply the count by 100, and check if it is above or below 70 or 10.

```c
// hash_table.c
void ht_insert(ht_hash_table* ht, const char* key, const char* value) {
    const int load = ht->count * 100 / ht->size;
    if (load > 70) {
        ht_resize_up(ht);
    }
    // ...
}


void ht_delete(ht_hash_table* ht, const char* key) {
    const int load = ht->count * 100 / ht->size;
    if (load < 10) {
        ht_resize_down(ht);
    }
    // ...
}
```

Next section: Appendix Table of contents

Part 07: Appendix: alternative collision handling

There are two common methods for handling collisions in hash tables:

- Separate chaining
- Open addressing

Separate chaining

Under separate chaining, each bucket contains a linked list. When items collide, they are added to the list. Methods:

- Insert: hash the key to get the bucket index. If there is nothing in that bucket, store the item there. If there is already an item there, append the item to the linked list.
- Search: hash the key to get the bucket index. Traverse the linked list, comparing each item's key to the search key. If the key is found, return the value, else return `NULL`.
- Delete: hash the key to get the bucket index. Traverse the linked list, comparing each item's key to the delete key. If the key is found, remove the item from the linked list. If there is only one item in the linked list, place the `NULL` pointer in the bucket, to indicate that it is empty.

This has the advantage of being simple to implement, but is space inefficient. Each item has to also store a pointer to the next item in the linked list, or the `NULL` pointer if no items come after it. This is space wasted on bookkeeping, which could be spent on storing more items.

Open addressing

Open addressing aims to solve the space inefficiency of separate chaining. When collisions happen, the collided item is placed in some other bucket in the table. The bucket that the item is placed into is chosen according some predetermined rule, which can be repeated when searching for the item. There are three common methods for choosing the bucket to insert a collided item into.

Linear probing.

When a collision occurs, the index is incremented and the item is put in the next available bucket in the array. Methods:

- Insert: hash the key to find the bucket index. If the bucket is empty, insert the item there. If it is not empty, repeatedly increment the index until an empty bucket is found, and insert it there.
- Search: hash the key to find the bucket index. Repeatedly increment the index, comparing each item's key to the search key, until an empty bucket is found. If an item with a matching key is found, return the value, else return `NULL`.
- Delete: hash the key to find the bucket index. Repeatedly increment the index, comparing each item's key to the delete key, until an empty bucket is found. If an item with a matching key is found, delete it. Deleting this item breaks the chain, so we have no choice but to reinsert all items in the chain after the deleted item.

Linear probing offers good cache performance, but suffers from clustering issues. Putting collided items in the next available bucket can lead to long contiguous stretches of filled buckets, which need to be iterated over when inserting, searching or deleting.

Quadratic probing.

Similar to linear probing, but instead of putting the collided item in the next available bucket, we try to put it in the buckets whose indexes follow the sequence: `i, i + 1, i + 4, i + 9, i + 16, ...`, where `i` is the original hash of the key. Methods:

- Insert: hash the key to find the bucket index. Follow the probing sequence until an empty or deleted bucket is found, and insert the item there.
- Search: hash the key to find the bucket index. Follow the probing sequence, comparing each item's key to the search key until an empty bucket is found. If a matching key is found, return the value, else return `NULL`.
- Delete: we can't tell if the item we're deleting is part of a collision chain, so we can't delete the item outright. Instead, we just mark it as deleted.

Quadratic probing reduces, but does not remove, clustering, and still offers decent cache performance.

Double hashing.

Double hashing aims to solve the clustering problem. To do so, we use a second hash function to choose a new index for the item. Using a hash function gives us a new bucket, the index of which should be evenly distributed across all buckets. This removes clustering, but also removes any boosted cache performance from locality of reference. Double hashing is a common method of collision management in production hash tables, and is the method we implement in this tutorial.