Exercise 37: Hashmaps

Hash Maps (Hashmaps, Hashes, or sometimes Dictionaries) are used frequently in many dynamic programming for storing key/value data. A Hashmap works by performing a "hashing" calculation on the keys to produce an integer, then uses that integer to find a bucket to get or set the value. It is a very fast practical data structure since it works on nearly any data and they are easy to implement.


(/book/)


(mailto:help@


(http://inculca

Here's an example of using a Hashmap (aka dict) in Python:

```
fruit_weights = {'Apples': 10, 'Oranges': 100, 'Grapes': 1.0}

for key, value in fruit_weights.items():
    print key, "=", value
```

Almost every modern language has something like this, so many people end up writing code and never understand how this actually works. By creating the `Hashmap` data structure in C I'll show you how this works. I'll start with the header file so I can talk about the data structure.

```
#ifndef _lcthw_Hashmap_h
#define _lcthw_Hashmap_h

#include <stdint.h>
#include <lcthw/darray.h>

#define DEFAULT_NUMBER_OF_BUCKETS 100

typedef int (*Hashmap_compare)(void *a, void *b);
typedef uint32_t (*Hashmap_hash)(void *key);

typedef struct Hashmap {
    DArray *buckets;
    Hashmap_compare compare;
    Hashmap_hash hash;
} Hashmap;

typedef struct HashmapNode {
    void *key;
    void *data;
    uint32_t hash;
} HashmapNode;

typedef int (*Hashmap_traverse_cb)(HashmapNode *node);

Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash);
void Hashmap_destroy(Hashmap *map);

int Hashmap_set(Hashmap *map, void *key, void *data);
void *Hashmap_get(Hashmap *map, void *key);

int Hashmap_traverse(Hashmap *map, Hashmap_traverse_cb traverse_cb);

void *Hashmap_delete(Hashmap *map, void *key);

#endif
```

The structure consists of a `Hashmap` that contains any number of `HashmapNode` structs. Looking at `Hashmap` you can see that it is structured like this:

`DArray *buckets`

A dynamic array that will be set to a fixed size of 100 buckets. Each bucket will in turn contain a `DArray` that will actually hold `HashmapNode` pairs.

`Hashmap_compare compare`

This is a comparison function that the `Hashmap` uses to actually find elements by their key. It should work like all of the other compare functions, and defaults to using `bstrcmp` so that keys are just bstrings.

`Hashmap_hash` hash

This is the hashing function and it's responsible for taking a key, processing its contents, and producing a single `uint32_t` index

number. You'll see the default one soon.

This almost tells you how the data is stored, but the `buckets` `DArray` isn't created yet. Just remember that it's kind of a two level mapping:

- There are 100 buckets that make up the first level, and things are in these buckets based on their hash.
- Each bucket is a `DArray` that then contains `HashmapNode` structs simply appended to the end as they're added.

The `HashmapNode` is then composed of these three elements:

`void *key`
The key for this key=value pair.

`void *value`
The value.

`uint32_t hash`
The calculated hash, which makes finding this

node quicker since we can just check the hash and skip any that don't match, only checking they key if it's equal.

The rest of the header file is nothing new, so now I can show you the implementation `hashmap.c` file:

```
#undef NDEBUG
#include <stdint.h>
#include <lcthw/hashmap.h>
#include <lcthw/dbg.h>
#include <lcthw/bstrlib.h>

static int default_compare(void *a, void *b)
{
    return bstrcmp((bstring)a, (bstring)b);
}

/**
 * Simple Bob Jenkins's hash algorithm taken from the
 * wikipedia description.
 */
static uint32_t default_hash(void *a)
{
    size_t len = blength((bstring)a);
    char *key = bdata((bstring)a);
    uint32_t hash = 0;
    uint32_t i = 0;

    for(hash = i = 0; i < len; ++i)
    {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}


Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash hash)
{
    Hashmap *map = calloc(1, sizeof(Hashmap));
    check_mem(map);

    map->compare = compare == NULL ? default_compare : compare;
    map->hash = hash == NULL ? default_hash : hash;
    map->buckets = DArray_create(sizeof(DArray *), DEFAULT_NUMBER_OF_BUCKETS);
    map->buckets->end = map->buckets->max; // fake out expanding it
    check_mem(map->buckets);

    return map;

error:
    if(map) {
        Hashmap_destroy(map);
    }

    return NULL;
```

```
}


void Hashmap_destroy(Hashmap *map)
{
    int i = 0;
    int j = 0;

    if(map) {
        if(map->buckets) {
            for(i = 0; i < DArray_count(map->buckets); i++) {
                DArray *bucket = DArray_get(map->buckets, i);
                if(bucket) {
                    for(j = 0; j < DArray_count(bucket); j++) {
                        free(DArray_get(bucket, j));
                    }
                    DArray_destroy(bucket);
                }
            }
            DArray_destroy(map->buckets);
        }

        free(map);
    }
}

static inline HashmapNode *Hashmap_node_create(int hash, void *key, void *data)
{
    HashmapNode *node = calloc(1, sizeof(HashmapNode));
    check_mem(node);

    node->key = key;
    node->data = data;
    node->hash = hash;

    return node;

error:
    return NULL;
}


static inline DArray *Hashmap_find_bucket(Hashmap *map, void *key,
        int create, uint32_t *hash_out)
{
    uint32_t hash = map->hash(key);
    int bucket_n = hash % DEFAULT_NUMBER_OF_BUCKETS;
    check(bucket_n >= 0, "Invalid bucket found: %d", bucket_n);
    *hash_out = hash; // store it for the return so the caller can use it


    DArray *bucket = DArray_get(map->buckets, bucket_n);

    if(!bucket && create) {
        // new bucket, set it up
        bucket = DArray_create(sizeof(void *), DEFAULT_NUMBER_OF_BUCKETS);
        check_mem(bucket);
        DArray_set(map->buckets, bucket_n, bucket);
    }

    return bucket;

error:
    return NULL;
}


int Hashmap_set(Hashmap *map, void *key, void *data)
{
    uint32_t hash = 0;
    DArray *bucket = Hashmap_find_bucket(map, key, 1, &hash);
    check(bucket, "Error can't create bucket.");

    HashmapNode *node = Hashmap_node_create(hash, key, data);
    check_mem(node);

    DArray_push(bucket, node);

    return 0;

error:
    return -1;
}

static inline int Hashmap_get_node(Hashmap *map, uint32_t hash, DArray *bucket, void *key)
{
    int i = 0;

    for(i = 0; i < DArray_end(bucket); i++) {
        debug("TRY: %d", i);
```

```
        HashmapNode *node = DArray_get(bucket, i);
        if(node->hash == hash && map->compare(node->key, key) == 0) {
            return i;
        }
    }

    return -1;
}

void *Hashmap_get(Hashmap *map, void *key)
{
    uint32_t hash = 0;
    DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
    if(!bucket) return NULL;

    int i = Hashmap_get_node(map, hash, bucket, key);
    if(i == -1) return NULL;

    HashmapNode *node = DArray_get(bucket, i);
    check(node != NULL, "Failed to get node from bucket when it should exist.");

    return node->data;

error: // fallthrough
    return NULL;
}


int Hashmap_traverse(Hashmap *map, Hashmap_traverse_cb traverse_cb)
{
    int i = 0;
    int j = 0;
    int rc = 0;

    for(i = 0; i < DArray_count(map->buckets); i++) {
        DArray *bucket = DArray_get(map->buckets, i);
        if(bucket) {
            for(j = 0; j < DArray_count(bucket); j++) {
                HashmapNode *node = DArray_get(bucket, j);
                rc = traverse_cb(node);
                if(rc != 0) return rc;
            }
        }
    }

    return 0;
}

void *Hashmap_delete(Hashmap *map, void *key)
{
    uint32_t hash = 0;
    DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
    if(!bucket) return NULL;

    int i = Hashmap_get_node(map, hash, bucket, key);
    if(i == -1) return NULL;

    HashmapNode *node = DArray_get(bucket, i);
    void *data = node->data;
    free(node);

    HashmapNode *ending = DArray_pop(bucket);

    if(ending != node) {
        // alright looks like it's not the last one, swap it
        DArray_set(bucket, i, ending);
    }

    return data;
}
```

There's nothing very complicated in the implementation, but the `default_hash` and `Hashmap_find_bucket` functions will need some explanation. When you use `Hashmap_create` you can pass in any compare and hash functions you want, but if you don't it uses the `default_compare` and `default_hash` functions.

The first thing to look at is how `default_hash` does its thing. This is a simple hash function called a "Jenkins hash" after Bob Jenkins. I got if from the Wikipedia page (http://en.wikipedia.org/wiki/Jenkins_hash_function) for the algorithm. It simply goes through each byte of the key to hash (a bstring) and works the bits so that the end result is a single `uint32_t`. It does this with some adding and xor operations.

There are many different hash functions, all with different properties, but once you have one you need a way to use it to find the right buckets. The `Hashmap_find_bucket` does it like this:

- First it calls `map->hash(key)` to get the hash for the key.
- It then finds the bucket using `hash % DEFAULT_NUMBER_OF_BUCKETS`, that way every hash will always find some bucket no matter how big it is.
- It then gets the bucket, which is also a `DArray`, and if it's not there it will create it. That depends on if the `create` variable says too.
- Once it has found the `DArray` bucket for the right hash, it returns it, and also the `hash_out` variable is used to give the caller the hash that was found.

All of the other functions then use `Hashmap_find_bucket` to do their work:

- Setting a key/value involves finding the bucket, then making a `HashmapNode`, and then adding it to the bucket.
- Getting a key involves finding the bucket, then finding the HashmapNode that matches the `hash` and `key` you want.
- Deleting an item again finds the bucket, finds where the requested node is, and then removes it by swapping the last node into its place.

The only other function that you should study is the `Hashmap_travers`. This simply walks every bucket, and for any bucket that has possible values, it calls the `traverse_cb` on each value. This is how you scan a whole `Hashmap` for its values.

The Unit Test

Finally you have the unit test that is testing all of these operations:

```
#include "minunit.h"
#include <lcthw/hashmap.h>
#include <assert.h>
#include <lcthw/bstrlib.h>

Hashmap *map = NULL;
static int traverse_called = 0;
struct tagbstring test1 = bsStatic("test data 1");
struct tagbstring test2 = bsStatic("test data 2");
struct tagbstring test3 = bsStatic("xest data 3");
struct tagbstring expect1 = bsStatic("THE VALUE 1");
struct tagbstring expect2 = bsStatic("THE VALUE 2");
struct tagbstring expect3 = bsStatic("THE VALUE 3");

static int traverse_good_cb(HashmapNode *node)
{
    debug("KEY: %s", bdata((bstring)node->key));
    traverse_called++;
    return 0;
}


static int traverse_fail_cb(HashmapNode *node)
{
    debug("KEY: %s", bdata((bstring)node->key));
    traverse_called++;

    if(traverse_called == 2) {
        return 1;
    } else {
        return 0;
    }
}


char *test_create()
{
    map = Hashmap_create(NULL, NULL);
    mu_assert(map != NULL, "Failed to create map.");

    return NULL;
}

char *test_destroy()
{
    Hashmap_destroy(map);

    return NULL;
}


char *test_get_set()
{
    int rc = Hashmap_set(map, &test1, &expect1);
    mu_assert(rc == 0, "Failed to set &test1");
```

```
    bstring result = Hashmap_get(map, &test1);
    mu_assert(result == &expect1, "Wrong value for test1.");

    rc = Hashmap_set(map, &test2, &expect2);
    mu_assert(rc == 0, "Failed to set test2");
    result = Hashmap_get(map, &test2);
    mu_assert(result == &expect2, "Wrong value for test2.");

    rc = Hashmap_set(map, &test3, &expect3);
    mu_assert(rc == 0, "Failed to set test3");
    result = Hashmap_get(map, &test3);
    mu_assert(result == &expect3, "Wrong value for test3.");

    return NULL;
}

char *test_traverse()
{
    int rc = Hashmap_traverse(map, traverse_good_cb);
    mu_assert(rc == 0, "Failed to traverse.");
    mu_assert(traverse_called == 3, "Wrong count traverse.");

    traverse_called = 0;
    rc = Hashmap_traverse(map, traverse_fail_cb);
    mu_assert(rc == 1, "Failed to traverse.");
    mu_assert(traverse_called == 2, "Wrong count traverse for fail.");

    return NULL;
}

char *test_delete()
{
    bstring deleted = (bstring)Hashmap_delete(map, &test1);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect1, "Should get test1");
    bstring result = Hashmap_get(map, &test1);
    mu_assert(result == NULL, "Should delete.");

    deleted = (bstring)Hashmap_delete(map, &test2);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect2, "Should get test2");
    result = Hashmap_get(map, &test2);
    mu_assert(result == NULL, "Should delete.");

    deleted = (bstring)Hashmap_delete(map, &test3);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect3, "Should get test3");
    result = Hashmap_get(map, &test3);
    mu_assert(result == NULL, "Should delete.");

    return NULL;
}

char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_get_set);
    mu_run_test(test_traverse);
    mu_run_test(test_delete);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

The only thing to learn about this unit test is that at the top I use a feature of `bstring` to create static strings to work with in the tests. I use the `tagbstring` and `bsStatic` to create them on lines 7-13.

How To Improve It

This is a very simple implementation of `Hashmap` as are most of the other data structures in this book. My goal isn't to give you insanely great hyper speed well tuned data structures. Usually those are much too complicated to discuss and only distract you from the real basic data structure at work. My goal is to give you an understandable starting point to then improve it or understand how they are implemented.

In this case, there's some things you can do with this implementation:

- You can use a sort on each bucket so that they are always sorted. This increases your insert time, but decreases your find time

because you can then use a binary search to find each node. Right now it's looping through all of the nodes in a bucket just to find one.
- You can dynamically size the number of buckets, or let the caller specify the number for each `Hashmap` created.
- You can use a better `default_hash`. There are tons of them.
- This (and nearly every `Hashmap` is vulnerable to someone picking keys that will fill only one bucket, and then tricking your program into processing them. This then makes your program run slower because it changes from processing a `Hashmap` to effectively processing a single `DArray`. If you sort the nodes in the bucket this helps, but you can also use better hashing functions, and for the really paranoid add a random salt so that keys can't be predicted.
- You could have it delete buckets that are empty of nodes to save space, or put empty buckets into a cache so you save on creating and destroying them.
- Right now it just adds elements even if they already exist. Write an alternative set method that only adds it if it isn't set already.

As usual you should go through each function and make it bullet proof. The `Hashmap` could also use a debug setting for doing an invariant check.

Extra Credit

- Research the `Hashmap` implementation of your favorite programming language to see what features they have.
- Find out what the major disadvantages of a `Hashmap` are and how to avoid them. For example, they do not preserve order without special changes and they don't work when you need to find things based on parts of keys.
- Write a unit test that demonstrates the defect of filling a `Hashmap` with keys that land in the same bucket, then test how this impact performance. A good way to do this is to just reduce the number of buckets to something stupid like 5.
