```
#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 *ke
{
    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;
}
```