

```
/*
 * Generic map implementation. This class is thread-safe.
 * free() must be invoked when only one thread has access to the hashmap.
 */
#include <stdlib.h>
#include <stdio.h>
#include <minithreads/hashmap.h>
#include <minithreads/synch.h>

#define INITIAL_SIZE 1024

// We need to keep keys and values
typedef struct _hashmap_element{
    int key;
    int in_use;
    any_t data;
} hashmap_element;

// A hashmap has some maximum size and current size,
// as well as the data to hold.
typedef struct _hashmap_map{
    int table_size;
    int size;
    hashmap_element *data;
    semaphore_t lock;
} hashmap_map;

/*
 * Return an empty hashmap, or NULL on failure.
 */
map_t hashmap_new() {
    hashmap_map* m = (hashmap_map*) malloc(sizeof(hashmap_map));
    if(!m) goto err;

    m->data = (hashmap_element*) calloc(INITIAL_SIZE, sizeof(hashmap_element));
    if(!m->data) goto err;

    m->lock = (semaphore_t) semaphore_create();
    if(!m->lock) goto err;
    semaphore_initialize(m->lock, 1);

    m->table_size = INITIAL_SIZE;
    m->size = 0;

    return m;
err:
    if (m)
        hashmap_free(m);
    return NULL;
}

/*
 * Hashing function for an integer
 */
unsigned int hashmap_hash_int(hashmap_map * m, unsigned int key){
    /* Robert Jenkins' 32 bit Mix Function */
    key += (key << 12);
    key ^= (key >> 22);
    key += (key << 4);
    key ^= (key >> 9);
    key += (key << 10);
    key ^= (key >> 2);
    key += (key << 7);
    key ^= (key >> 12);

    /* Knuth's Multiplicative Method */
}
```

```
    key = (key >> 3) * 2654435761;

    return key % m->table_size;
}

/*
 * Return the integer of the location in data
 * to store the point to the item, or MAP_FULLL.
 */
int hashmap_hash(map_t in, int key){
    int curr;
    int i;

    /* Cast the hashmap */
    hashmap_map* m = (hashmap_map *) in;

    /* If full, return immediately */
    if(m->size == m->table_size) return MAP_FULLL;

    /* Find the best index */
    curr = hashmap_hash_int(m, key);

    /* Linear probing */
    for(i = 0; i < m->table_size; i++){
        if(m->data[curr].in_use == 0)
            return curr;

        if(m->data[curr].key == key && m->data[curr].in_use == 1)
            return curr;

        curr = (curr + 1) % m->table_size;
    }

    return MAP_FULLL;
}

/*
 * Doubles the size of the hashmap, and rehashes all the elements
 */
int hashmap_rehash(map_t in){
    int i;
    int old_size;
    hashmap_element* curr;

    /* Setup the new elements */
    hashmap_map *m = (hashmap_map *) in;
    hashmap_element* temp = (hashmap_element *) calloc(2 * m->table_size, sizeof
    (hashmap_element));
    if(!temp) return MAP_OMEM;

    /* Update the array */
    curr = m->data;
    m->data = temp;

    /* Update the size */
    old_size = m->table_size;
    m->table_size = 2 * m->table_size;
    m->size = 0;

    /* Rehash the elements */
    for(i = 0; i < old_size; i++){
        int status = hashmap_put(m, curr[i].key, curr[i].data);
        if (status != MAP_OK)
            return status;
    }
}
```

```
    free(curr);

    return MAP_OK;
}

/*
 * Add a pointer to the hashmap with some key
 */
int hashmap_put(map_t in, int key, any_t value){
    int index;
    hashmap_map* m;

    /* Cast the hashmap */
    m = (hashmap_map *) in;

    /* Lock for concurrency */
    semaphore_P(m->lock);

    /* Find a place to put our value */
    index = hashmap_hash(in, key);
    while(index == MAP_FULL){
        if (hashmap_rehash(in) == MAP_OMEM) {
            semaphore_V(m->lock);
            return MAP_OMEM;
        }
        index = hashmap_hash(in, key);
    }

    /* Set the data */
    m->data[index].data = value;
    m->data[index].key = key;
    m->data[index].in_use = 1;
    m->size++;

    /* Unlock */
    semaphore_V(m->lock);

    return MAP_OK;
}

/*
 * Get your pointer out of the hashmap with a key
 */
int hashmap_get(map_t in, int key, any_t *arg){
    int curr;
    int i;
    hashmap_map* m;

    /* Cast the hashmap */
    m = (hashmap_map *) in;

    /* Lock for concurrency */
    semaphore_P(m->lock);

    /* Find data location */
    curr = hashmap_hash_int(m, key);

    /* Linear probing, if necessary */
    for(i = 0; i < m->table_size; i++){

        if(m->data[curr].key == key && m->data[curr].in_use == 1){
            *arg = (int *) (m->data[curr].data);
            semaphore_V(m->lock);
            return MAP_OK;
        }
    }
}
```

```
        curr = (curr + 1) % m->table_size;
    }

    *arg = NULL;

    /* Unlock */
    semaphore_V(m->lock);

    /* Not found */
    return MAP_MISSING;
}

/*
 * Get a random element from the hashmap
 */
int hashmap_get_one(map_t in, any_t *arg, int remove){
    int i;
    hashmap_map* m;

    /* Cast the hashmap */
    m = (hashmap_map *) in;

    /* On empty hashmap return immediately */
    if (hashmap_length(m) <= 0)
        return MAP_MISSING;

    /* Lock for concurrency */
    semaphore_P(m->lock);

    /* Linear probing */
    for(i = 0; i < m->table_size; i++){
        if(m->data[i].in_use != 0){
            *arg = (any_t) (m->data[i].data);
            if (remove) {
                m->data[i].in_use = 0;
                m->size--;
            }
            semaphore_V(m->lock);
            return MAP_OK;
        }
    }

    /* Unlock */
    semaphore_V(m->lock);

    return MAP_OK;
}

/*
 * Iterate the function parameter over each element in the hashmap. The
 * additional any_t argument is passed to the function as its first
 * argument and the hashmap element is the second.
 */
int hashmap_iterate(map_t in, PFany f, any_t item) {
    int i;

    /* Cast the hashmap */
    hashmap_map* m = (hashmap_map*) in;

    /* On empty hashmap, return immediately */
    if (hashmap_length(m) <= 0)
        return MAP_MISSING;

    /* Lock for concurrency */
    semaphore_P(m->lock);

    /* Linear probing */
```

```
    for(i = 0; i < m->table_size; i++)
        if(m->data[i].in_use != 0) {
            any_t data = (any_t) (m->data[i].data);
            int status = f(item, data);
            if (status != MAP_OK) {
                semaphore_V(m->lock);
                return status;
            }
        }

    /* Unlock */
    semaphore_V(m->lock);

    return MAP_OK;
}

/*
 * Remove an element with that key from the map
 */
int hashmap_remove(map_t in, int key){
    int i;
    int curr;
    hashmap_map* m;

    /* Cast the hashmap */
    m = (hashmap_map *) in;

    /* Lock for concurrency */
    semaphore_P(m->lock);

    /* Find key */
    curr = hashmap_hash_int(m, key);

    /* Linear probing, if necessary */
    for(i = 0; i < m->table_size; i++){
        if(m->data[curr].key == key && m->data[curr].in_use == 1){
            /* Blank out the fields */
            m->data[curr].in_use = 0;
            m->data[curr].data = NULL;
            m->data[curr].key = 0;

            /* Reduce the size */
            m->size--;
            semaphore_V(m->lock);
            return MAP_OK;
        }
        curr = (curr + 1) % m->table_size;
    }

    /* Unlock */
    semaphore_V(m->lock);

    /* Data not found */
    return MAP_MISSING;
}

/* Deallocate the hashmap */
void hashmap_free(map_t in){
    hashmap_map* m = (hashmap_map*) in;
    free(m->data);
    semaphore_destroy(m->lock);
    //free(m); FIXME: crashes
}

/* Return the length of the hashmap */
int hashmap_length(map_t in){
```

```
hashmap_map* m = (hashmap_map *) in;  
if(m != NULL) return m->size;  
else return 0;  
}
```