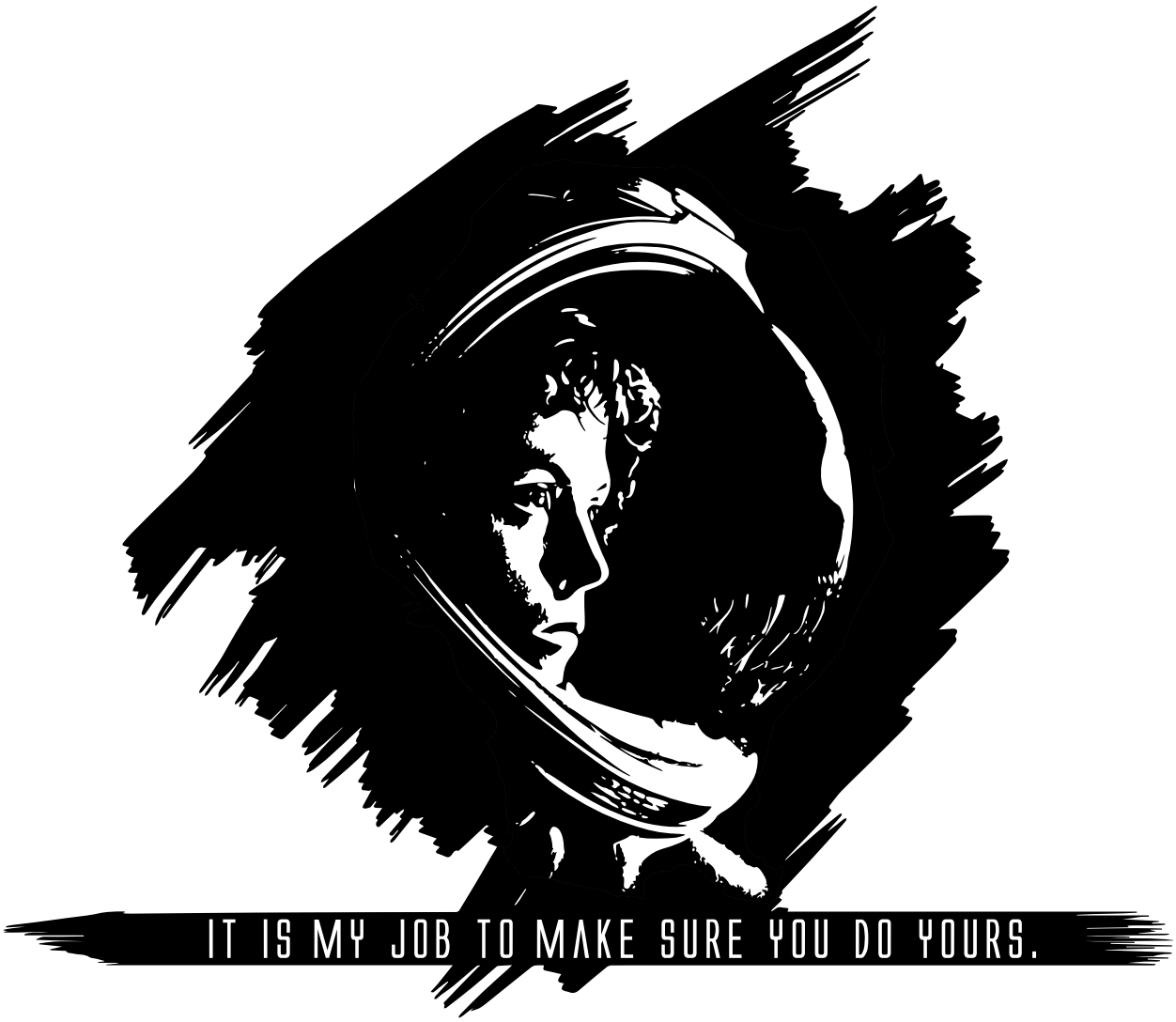
Exercises — Hash map

version #



Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

1 Hash map

Files to submit:

- hash_map/hash_map.c
- hash_map/dump.c
- hash_map/hash.c

Provided files:

- hash_map/hash_map.h
- hash_map/hash.c

Authorized functions: You are only allowed to use the following functions:

- malloc(3)
- realloc(3)
- calloc(3)
- free(3)
- printf(3)
- puts(3)
- putchar(3)

Authorized headers: You are only allowed to use the functions defined in the following headers:

- err.h
- errno.h
- assert.h
- stddef.h

1.1 Goal

In this exercise, you will implement a hash map (a.k.a. a dictionary).

> *"In computing, a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored."*
>
> ---Wikipedia https://en.wikipedia.org/wiki/Hash_table.

The file `hash_map.h` provides function prototypes as well as data structure definitions. The hash function is in the provided `hash.c` file. You are highly encouraged to change the hash function to see its effects on the data structure's performance.

```c
struct pair_list
{
    const char *key;
    char *value;
    struct pair_list *next;
};

struct hash_map
{
    struct pair_list **data;
    size_t size;
};
```

Notes:

- You **MUST** use the above structures as is.
- The case where the pointer to the dictionary is `NULL` will not be tested.

1.1.1 hash_map_init

```c
struct hash_map *hash_map_init(size_t size);
```

Returns an empty `hash_map` of size `size`. If an allocation fails, returns `NULL`.

1.1.2 hash_map_insert

```c
bool hash_map_insert(struct hash_map *hash_map, const char *key, char *value,
                     bool *updated);
```

Inserts the `key`/`value` pair in `hash_map`. If two keys have the same hash value, you **MUST** resolve the conflict by chaining them. Meaning you **MUST** use a linked list for the colliding elements and add the new element at the head of the list.

If this key is already present in the structure, the old value associated with the key **MUST** be updated and the value held by `updated` **MUST** be set to `true`. Otherwise, it must be set to `false`. If an allocation fails, the function **MUST** return `false` and the value held by `updated` should not be changed. `updated` **MIGHT** be `NULL`.

Example

If we insert `"SKI"` and then `"ACU"`, there is a collision, and the two elements need to be chained:

```
   -------------
0 |             |
1 |             |
2 |             |
3 | SKI(winter) |
4 |             |
```

(continues on next page)

(continued from previous page)

```
5 |             |
6 |             |
  |_____________|


   -------------
0 |             |
1 |             |
2 |             |
3 |   ACU(51)   | -> SKI(winter)
4 |             |
5 |             |
6 |             |
  |_____________|
```

1.1.3 hash_map_free

```
void hash_map_free(struct hash_map *hash_map);
```

Free all resources allocated by `hash_map`, including the pointer passed to the function.

1.1.4 hash_map_dump

```
void hash_map_dump(struct hash_map *hash);
```

Prints on the standard output the whole dictionary. You **MUST** print the elements in the same order as they appear in the array and each list **MUST** be separated by a line feed. When one of the lists is empty, you **MUST** skip it.

In case of a slot containing multiple key/value pairs, each pair **MUST** be separated by a comma followed by a space. Note that these delimiters **MUST** not appear at the end of the line.

In case of an empty list, it **MUST** be ignored.

Last but not least, each pair **MUST** be printed as follows:

```
key: value
```

Example

Let's suppose our dictionary looks like the following one (values are within parentheses) and that `"ACU"` was added after `"SKI"`:

```
   -------------
0 |             |
1 |   TARAN(W)  |
2 |             |
3 |   ACU(51)   | -> SKI(winter)
```

(continues on next page)

(continued from previous page)

```
4 |             |
5 |    C(42)    |
6 |   42(life)  |
  |_____________|
```

A call to `hash_map_dump` should print the following result on the standard output:

```
42sh$ ./hash_map_dump | cat -e
TARAN: W$
ACU: 51, SKI: winter$
C: 42$
42: life$
```

1.1.5 hash_map_get

```
const char *hash_map_get(const struct hash_map *hash, const char *key);
```

Return the value associated to `key` in `hash_map`.

Return `NULL` if the key was not found.

Example

```
hash_map_insert(my_hash, "test", "42", NULL);
/* do some clever stuff */
hash_map_get(my_hash, "test"); // == "42"
```

1.1.6 hash_map_remove

```
bool hash_map_remove(struct hash_map *hash, const char *key);
```

Remove `key` and its associated value from `hash_map`. You **MUST** free the removed `pair_list` but not its contained `value` and `key`.

Return `false` if the key was not found, `true` otherwise.

It is my job to make sure you do yours.