`libmap` User Guide

Aditya Singh

March 15, 2025

Contents

1 Introduction

`libmap` is a hashmap library (API) for C. It features dynamic resizing, collision resolution using chaining, and customizable key-value handling using user-defined functions.
Specifically, it provides functions for creating a hashmap, configuring the hashmap, inserting, printing, retrieving key-value pairs, removing keys, iterating, getting the size of the hashmap, and destroying the hashmap.

2 Building

2.1 Prerequisites

To use `libmap`, you should have the latest version of `gcc` or equivalent C compiler. `libmap` compiles under `-std=c99 -Wall -Werror -Wextra -pedantic`, so it should work with all compilers that support at least C99.

2.2 Compilation and Linking

To build, simply run `make` on any UNIX-y system. This generates `build/libmap.a`, which can then be statically linked into your program. Alternately, if you simply want to test `libmap`, you can add your program to the `tests/` directory and its binary will be generated in `bin/` after running `make`.

3 API Reference

3.1 Core Functions

3.1.1 `map_create`

Function Signature:

```
map_error_t map_create(map_t **map,
                       void *(*usr_key_clone)(void *key),
                       void *(*usr_value_clone)(void *value),
                       uint64_t (*usr_hash)(void *key),
                       char *(*usr_stringify)(void *key, void *value),
                       int32_t (*usr_compare)(void *key1, void *key2),
                       void (*usr_free_key)(void *key),
                       void (*usr_free_value)(void *value))
```

Description: Initializes a new hashmap, allowing users to store and manage key-value pairs. It allocates memory for the hashmap structure, sets up the initial number of buckets, and registers user-defined functions for key management, hashing, comparison, and memory handling.

Arguments:

- `map_t **map` — (Output) Pointer to the hashmap. This function allocates a new hashmap and stores the pointer in `*map`. The caller must pass a valid, uninitialized pointer (typically `NULL`). The user is responsible for calling `map_destroy` to free the allocated memory.
- `void *(*usr_key_clone)(void *key)` — (Input) Function pointer for cloning keys. This function is provided by the user and should return a newly allocated copy of the key. The library assumes ownership of the cloned key and will later free it using `usr_free_key`.
- `void *(*usr_value_clone)(void *value)` — (Input) Function pointer for cloning values. This function is provided by the user and should return a newly allocated copy of the value. The library takes ownership of this copy and will free it using `usr_free_value`.
- `uint64_t (*usr_hash)(void *key)` — (Input) Function pointer for hashing keys. This function is provided by the user and must return a deterministic hash value for a given key. Ownership: The library does not modify or free the key passed to this function.
- `char *(*usr_stringify)(void *key, void *value)` — (Input) Function pointer for converting key-value pairs into strings. The returned string must be dynamically allocated since the caller is responsible for freeing it.
- `int32_t (*usr_compare)(void *key1, void *key2)` — (Input) Function pointer for comparing two keys. This function must not modify either key. It returns: - ‘0‘ if ‘key1‘ and ‘key2‘ are equal - Nonzero if they differ
- `void (*usr_free_key)(void *key)` — (Input) Function pointer for freeing keys. This function is called by the hashmap when a key is removed or the map is destroyed.
- `void (*usr_free_value)(void *value)` — (Input) Function pointer for freeing values. The library calls this when a value is removed or overwritten. The function must not modify the pointer itself, only the memory it points to.

Ownership Table:

Argument	Ownership
`map_t **map`	Library allocates, user frees
`usr_key_clone`	User provides, library owns copy
`usr_value_clone`	User provides, library owns copy
`usr_hash`	User owns, library calls it
`usr_stringify`	User owns, caller frees result
`usr_compare`	User owns, library calls it
`usr_free_key`	User owns, library calls it
`usr_free_value`	User owns, library calls it

3.1.2 `map_destroy`

Function Signature:

```
map_error_t map_destroy(map_t **map)
```

Description: This function deallocates all memory used by the hashmap,including, stored keys,values and internal structures. After this function call, the pointer to the hashmap is set to `NULL` to avoid dangling pointer.

Arguments:

- `map_t **map` — (Input/Output) Pointer to the hashmap.This function frees all memory allocated for the hashmap and sets `*map` to `NULL`.

Ownership Table:

Argument	Ownership
`map_t **map`	Library frees

3.1.3 `map_insert`

Function Signature:

```
map_error_t map_insert(map_t *map, void* key, void* value)
```

Description: This functions adds a new key-value pair into the hashmap.If the key already exists, the function updates the existing value.If the load factor exceeds the threshold,the map resizes itself automatically.

Arguments:

- `map_t *map` — (Input) Pointer to an existing hashmap.The hashmap must be properly initialized before calling this function.
- `void *key` — (Input) Pointer to the key to be inserted.The key is cloned using the user-defined `usr_key_clone`function.
- `void *value` — (Input) Pointer to the value to be inserted.The value is cloned using the user-defined `usr_value_clone` function.

Ownership Table:

Argument	Ownership
`map_t *map`	User owns
`void *key`	User owns original, library owns copy
`void *value`	User owns original, library owns copy

Time Complexity:

- **Best Case:** O(1) — Direct insertion into an empty bucket.

- **Worst Case:** O(n) — Hash collision leads to a long linked list traversal.
- **Amortized:** O(1) — Assuming uniform hashing and resizing when needed.

Example Usage:

```c
// Define key and value
char *key = "name";
char *value = "Alice";

// Insert into hashmap
map_error_t result = map_insert(my_map, key, value);

if (result == MAP_OK) {
    printf("Insertion successful!\n");
} else {
    printf("Error: %d\n", result);
}
```

3.1.4 map_get

This function retrieves the value associated with a given key from the hashmap.

Function Signature:

```c
map_error_t map_get(const map_t *map, void *key, void **value);
```

Description: This function searches for a key in the hashmap and, if found, returns a pointer to its associated value. If the key is not present, the function returns an error.

Arguments:

- `const map_t *map` — (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.
- `void *key` — (Input) Pointer to the key to search for. The key is not modified and is compared using the user-defined `usr_compare` function.
- `void **value` — (Output) Pointer to store the retrieved value. If the key is found, this pointer is set to the value stored in the hashmap. If the key does not exist, this pointer remains unchanged.

Ownership Table:

Argument	Ownership
`const map_t *map`	User owns
`void *key`	User owns
`void **value`	User owns, library sets pointer

Time Complexity:

- **Best Case:** O(1) — Direct lookup if no collisions.
- **Worst Case:** O(n) — Searching through a long linked list (worst collision scenario).
- **Amortized:** O(1) — Assuming uniform hashing and proper resizing.

Example Usage:

```c
// Define key and expected value
char *key = "name";
char *retrieved_value = NULL;

// Retrieve value from hashmap
map_error_t result = map_get(my_map, key, (void**)&retrieved_value);
```

```c

if (result == MAP_OK) {
    printf("Retrieved: %s\n", retrieved_value);
} else {
    printf("Key not found.\n");
}
```

3.1.5 map_remove

This function removes a key—value pair from the hashmap.

Function Signature:

```c
map_error_t map_remove(map_t *map, void *key);
```

Description: This function searches for a key in the hashmap and, if found, removes it along with its associated value. If the key does not exist, the function returns an error.

Arguments:

- `map_t *map` — (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.
- `void *key` — (Input) Pointer to the key to remove. The function searches for this key and removes it if found.

Ownership Table:

Argument	Ownership
`map_t *map`	User owns
`void *key`	User owns

Time Complexity:

- **Best Case:** O(1) — Direct removal if no collisions.
- **Worst Case:** O(n) — Searching through a long linked list (worst collision scenario).
- **Amortized:** O(1) — Assuming uniform hashing and proper resizing.

Example Usage:

```c
// Define key to remove
char *key = "name";

// Remove key from hashmap
map_error_t result = map_remove(my_map, key);

if (result == MAP_OK) {
    printf("Key removed successfully!\n");
} else {
    printf("Key not found.\n");
}
```

3.1.6 map_configure

This function allows the user to adjust the hashmap's load factors and resizing behavior.

Function Signature:

```c
map_error_t map_configure(map_t *map,
                          float max_load_factor,
                          float min_load_factor,
                          float grow_factor);
```

Description: This function updates the hashmap's resizing thresholds and growth behavior. — If the number of elements exceeds `max_load_factor * num_buckets`, the hashmap grows.
— If the number of elements falls below `min_load_factor * num_buckets`, the hashmap shrinks.
— The `grow_factor` determines how much the hashmap expands when resized.

Arguments:

- `map_t *map` — (Input) Pointer to the hashmap to be configured. The hashmap must be initialized before calling this function.
- `float max_load_factor` — (Input) Maximum load factor before resizing. If the load factor exceeds this value, the hashmap grows.
- `float min_load_factor` — (Input) Minimum load factor before shrinking. If the load factor falls below this value, the hashmap shrinks.
- `float grow_factor` — (Input) Factor by which the hashmap grows. The number of buckets is multiplied by this factor when resizing.

Ownership Table:

Argument	Ownership
`map_t *map`	User owns
`float max_load_factor`	User owns
`float min_load_factor`	User owns
`float grow_factor`	User owns

Time Complexity:

- **Best Case:** O(1) — Directly updates struct values.
- **Worst Case:** O(1) — No complex operations.

Example Usage:

```
// Configure the hashmap
map_error_t result = map_configure(my_map, 2.0, 0.5, 2.0);

if (result == MAP_OK) {
    printf("Configuration updated!\n");
} else {
    printf("Invalid configuration parameters.\n");
}
```

3.1.7 `map_get_size`

This function retrieves the number of key-value pairs currently stored in the hashmap.

Function Signature:

```
map_error_t map_get_size(map_t *map, int *num_elements);
```

Description: This function provides the total count of elements in the hashmap without iterating over all entries. It stores the retrieved size in the integer pointed to by `num_elements`.

Arguments:

- `map_t *map` — (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.
- `int *num_elements` — (Output) Pointer to an integer where the function will store the number of elements in the hashmap. If `map` is valid, this value will be updated.

Ownership Table:

Argument	Ownership
`map_t *map`	User owns
`int *num_elements`	User owns, library writes to it

Time Complexity:

- **Best Case:** O(1) — Direct access to the size variable.
- **Worst Case:** O(1) — No complex operations involved.

Example Usage:

```c
// Define variable to store the hashmap size
int size = 0;

// Retrieve the number of elements in the hashmap
map_error_t result = map_get_size(my_map, &size);

if (result == MAP_OK) {
    printf("Hashmap contains %d elements.\n", size);
} else {
    printf("Failed to retrieve hashmap size.\n");
}
```

3.1.8 `map_get_num_buckets`

This function retrieves the number of buckets in the hashmap.

Function Signature:

```c
map_error_t map_get_num_buckets(map_t *map, int *num_buckets);
```

Description: This function returns the current number of buckets in the hashmap. The number of buckets affects the load factor and performance of insertions, lookups, and deletions. Users can use this function to analyze and fine-tune the resizing behavior of the hashmap.

Arguments:

- `map_t *map` — (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.
- `int *num_buckets` — (Output) Pointer to an integer where the function will store the number of buckets in the hashmap. If `map` is valid, this value will be updated.

Ownership Table:

Argument	Ownership
`map_t *map`	User owns
`int *num_buckets`	User owns, library writes to it.

Time Complexity:

- **Best Case:** O(1) — Direct access to the bucket count variable.
- **Worst Case:** O(1) — No complex operations involved.

Example Usage:

```c
// Define variable to store the number of buckets
int num_buckets = 0;

// Retrieve the number of buckets in the hashmap
map_error_t result = map_get_num_buckets(my_map, &num_buckets);

if (result == MAP_OK) {
```

```
    printf("Hashmap currently has %d buckets.\n", num_buckets);
} else {
    printf("Failed to retrieve bucket count.\n");
}
```

3.2 Iterating Functions

3.2.1 map_iter_start

This function initializes an iterator for traversing the hashmap.

Function Signature:

```
map_error_t map_iter_start(const map_t *map, map_iterator_t *iter);
```

Description: This function prepares an iterator to traverse the hashmap. It finds the first valid key-value pair and sets up the iterator to begin iteration. If the hashmap is empty, the iterator is set to indicate the end of the map.

Arguments:

- `const map_t *map` — (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.
- `map_iterator_t *iter` — (Output) Pointer to an iterator structure. This iterator will be initialized to point to the first valid entry in the hashmap.

Ownership Table:

Argument	Ownership
`const map_t *map`	User owns
`map_iterator_t *iter`	User owns, library initializes

Time Complexity:

- **Best Case:** O(1) — First bucket contains an entry.
- **Worst Case:** O(n) — All buckets are empty, requiring a full scan.

Example Usage:

```
// Initialize an iterator
map_iterator_t iter;
map_error_t result = map_iter_start(my_map, &iter);

if (result == MAP_OK) {
    printf("Iterator initialized successfully!\n");
} else {
    printf("Hashmap is empty.\n");
}
```

3.2.2 map_iter_next

This function retrieves the next key-value pair in the hashmap using an iterator.

Function Signature:

```
map_error_t map_iter_next(const map_t *map, map_iterator_t *iter,
                          void **out_key, void **out_value);
```

Description: This function advances the iterator to the next key-value pair in the hashmap. If the iterator reaches the end of the map, the function returns an error.

Arguments:

- `const map_t *map` - (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.
- `map_iterator_t *iter` — (Input/Output) Pointer to an iterator structure. This function updates the iterator to point to the next entry.
- `void **out_key` — (Output) Pointer to store the next key. The function updates this pointer to reference the next key.
- `void **out_value` — (Output) Pointer to store the next value. The function updates this pointer to reference the next value.

Ownership Table:

Argument	Ownership
`const map_t *map`	User owns
`map_iterator_t *iter`	User owns, library updates it
`void **out_key`	Library assigns existing key pointer
`void **out_value`	Library assigns existing value pointer

Time Complexity:

- **Best Case:** O(1) — Next element is in the same bucket.
- **Worst Case:** O(n) — Requires scanning multiple buckets for the next entry.

Example Usage:

```
// Initialize an iterator
map_iterator_t iter;
map_iter_start(my_map, &iter);

// Iterate through all elements
void *key, *value;
while (map_iter_next(my_map, &iter, &key, &value) == MAP_OK) {
    printf("Key: %s, Value: %s\n", (char *)key, (char *)value);
}
```

3.3 Utility Functions

3.3.1 `map_print`

This function prints the contents of the hashmap in a human-readable format.

Function Signature:

```
map_error_t map_print(const map_t *map);
```

Description: This function prints all key-value pairs in the hashmap. It is mainly used for debugging and visualization. If the hashmap is empty, it prints a message instead of crashing.

Arguments:

- `const map_t *map` — (Input) Pointer to an existing hashmap. The hashmap must be initialized before calling this function.

Ownership Table:

Argument	Ownership
`const map_t *map`	User owns

Time Complexity:

- **Best Case:** O(1) — The hashmap is empty, so nothing is printed.
- **Worst Case:** O(n) — All elements must be printed.

Example Usage:

```c
// Print the hashmap contents
map_error_t result = map_print(my_map);

if (result != MAP_OK) {
    printf("Failed to print hashmap.\n");
}
```

4 Internal Design

4.1 Overview

4.1.1 High-Level Structure

The hashmap (`map_t`) is implemented as an **array of pointers to linked lists**, where:

- **Buckets** (`buckets`): An array of pointers (`map_element_t **buckets`) where each entry points to the **head of a linked list**. If a bucket is empty, its pointer is `NULL`; otherwise, it points to the first element in the list.
- **Linked Lists**: Each node (`map_element_t`) stores a **key-value pair** and a **pointer to the next node**. Multiple nodes may exist in the same bucket due to collisions.
- **Resizing**: The hashmap grows and shrinks dynamically to maintain efficiency.

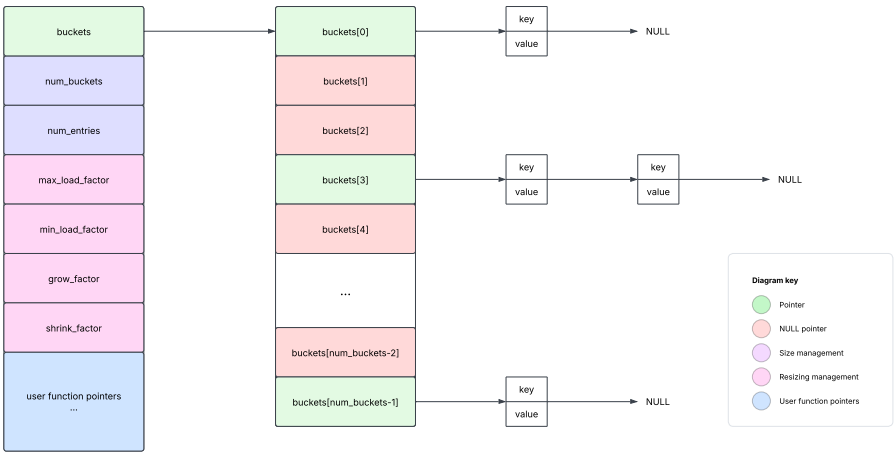


Figure 1: Hashmap Structure

4.1.2 Insertion Process

1. Hashing
The hashmap relies on a user-supplied hash function, defined as:

$$\text{usr\_hash(key)} \rightarrow \text{unsigned integer}$$

This function determines the bucket index as follows:

$$\text{index} = \text{usr\_hash(key)} \mod \text{num\_buckets}$$

The efficiency of hashmap operations depends on the quality of this hash function.

Example Hash Functions:

```c
// Simple Modulo Hash Function
uint64_t simple_hash(void *key) {
    return (uint64_t)key % 1024;
}

// FNV-1a Hash Function (More Effective)
uint64_t fnv1a_hash(void *key, size_t len) {
    uint64_t hash = 14695981039346656037ULL; // FNV offset basis
    for (size_t i = 0; i < len; i++) {
        hash ^= ((unsigned char*)key)[i];
        hash *= 1099511628211ULL; // FNV prime
```

```
    }
    return hash;
}
```

2. Bucket Selection
Each bucket is an **entry in an array** indexed using the computed hash value:

$$\text{buckets[index]}$$

If `buckets[index]` is `NULL`, the bucket is empty, and a new `map_element_t` is allocated. Otherwise, the key-value pair is inserted into the existing linked list at that bucket.

4.1.3 Collision Handling (Separate Chaining)

Why Do Collisions Happen?
A **collision** occurs when two different keys hash to the same bucket index. Since the hashmap uses a fixed number of buckets, multiple keys can map to the same index:

$$\text{index} = \text{usr\_hash(key)} \mod \text{num\_buckets}$$

To handle this, the hashmap uses separate chaining, where each bucket points to the head of a linked list storing multiple key-value pairs.

How the Hashmap Handles Collisions:
When inserting a new key-value pair:

- The bucket index is computed using the hash function.
- If the bucket is empty (`NULL`), a new element is created and assigned to the bucket.
- If the bucket already contains elements, the hashmap:
- Traverses the linked list to check if the key already exists.
- If the key exists, the value is updated.
- Otherwise, the new key-value pair is added to the front of the linked list.

Example: Handling Collisions in Code

```
map_error_t map_insert(map_t *map, void *key, void *value) {
    if (!map || !key || !value) return MAP_ERR_INVALID_ARG;

    // Compute bucket index
    uint64_t index = map->usr_hash(key) % map->num_buckets;

    // Get the head of the linked list at this bucket
    map_element_t *current = map->buckets[index];

    // Traverse the list to check if the key already exists
    while (current) {
        if (map->usr_compare(current->_key, key) == 0) {
            // Key already exists, update the value
            map->usr_free_value(current->_value);
            current->_value = map->usr_value_clone(value);
            return MAP_OK;
        }
        current = current->_next;
    }

    // If key does not exist, create a new element and insert it at the head
```

```
22      map_element_t *new_elem = malloc(sizeof(map_element_t));
23      if (!new_elem) return MAP_ERR_NO_MEM;
24
25      new_elem->_key = map->usr_key_clone(key);
26      new_elem->_value = map->usr_value_clone(value);
27      new_elem->_next = map->buckets[index]; // Insert at the front
28      map->buckets[index] = new_elem; // Update bucket pointer
29
30      map->num_entries++; // Increment entry count
31      return MAP_OK;
32  }
```

4.1.4 Resizing Strategy

The hashmap dynamically resizes to ensure efficient performance:

1. **Growth (Expanding the Number of Buckets)**
   - When the **load factor** exceeds 0.75, the hashmap **doubles** in size.
   - A larger bucket array reduces collisions and improves lookup speed.
   - All elements are rehashed and redistributed across the new bucket array.
2. **Shrinking (Reducing the Number of Buckets)**
   - When the **load factor** drops below 0.25, the hashmap **shrinks** to half its current size.
   - This prevents excessive memory usage when fewer elements are stored.
   - Elements are rehashed into the smaller bucket array.
3. **Why Resizing Matters**
   - Without resizing, a hashmap can become **inefficient**, leading to more collisions and slower lookups.
   - Resizing ensures that lookup time remains **close to** $O(1)$ on average.

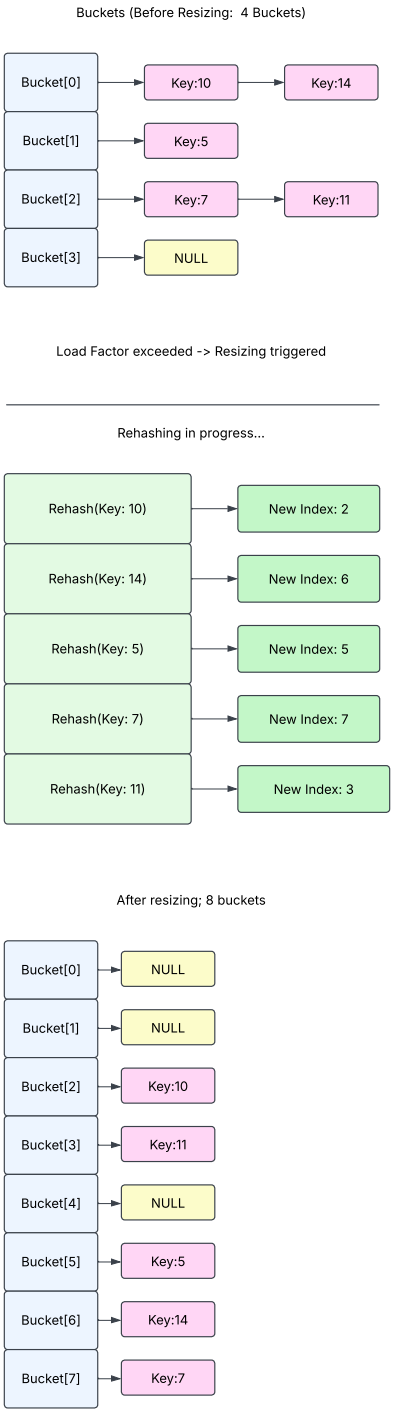


Figure 2: Resizing process in `libmap`: Elements are rehashed and moved to new buckets.

4.1.5 Key Takeaways

- **Buckets are pointers** (`map_element_t **buckets`)
- **Each bucket points to a linked list head** (or `NULL` if empty)
- **Collisions are handled by linked list chaining**
- **Resizing happens dynamically** to maintain efficiency

- **Hashmap grows when load factor** > 0.75 (to reduce collisions)
- **Hashmap shrinks when load factor** < 0.25 (to save memory)
- **Good hash functions reduce collisions and improve performance**

4.2 Data Structures

4.2.1 `map_t` (Hashmap Structure)

The `map_t` structure represents the main hashmap. It stores key-value pairs, handles collisions, and manages resizing.

Structure Definition:

```
typedef struct {
  map_element_t **buckets;  // Array of bucket pointers (linked list heads)
  int32_t num_buckets;      // Current number of buckets
  int32_t num_entries;      // Current number of stored key-value pairs
  float max_load_factor;    // Resizing threshold for growth
  float min_load_factor;    // Resizing threshold for shrinking
  float grow_factor;        // Factor by which the hashmap grows
  float shrink_factor;      // Factor by which the hashmap shrinks

  // User-defined function pointers
  void *(*usr_key_clone)(void *key);
  void *(*usr_value_clone)(void *value);
  uint64_t (*usr_hash)(void *key);
  char *(*usr_stringify)(void *key, void *data);
  int32_t (*usr_compare)(void *key1, void *key2);
  void (*usr_free_key)(void *key);
  void (*usr_free_value)(void *value);
} map_t;
```

Description: This structure contains all necessary information for managing a hashmap, including:

- Buckets (`buckets`) → Array of pointers to linked lists for separate chaining.
- Load Factors (`max_load_factor`, `min_load_factor`) → Control when resizing happens.
- Resizing Factors (`grow_factor`, `shrink_factor`) → Determines how much the hashmap expands or shrinks.
- User-Defined Functions → Custom key-value operations (hashing, comparison, cloning, freeing, stringifying).

Member Breakdown:

Member	Type	Description
buckets	map_element_t**	Array of bucket pointers (linked list heads)
num_buckets	int32_t	Number of buckets (size of buckets array)
num_entries	int32_t	Number of key-value pairs stored
max_load_factor	float	Threshold for hashmap growth
min_load_factor	float	Threshold for hashmap shrinking
grow_factor	float	Resizing growth multiplier (e.g., 2.0x)
shrink_factor	float	Resizing shrink multiplier (e.g., 0.5x)
User-Defined Functions		
usr_key_clone	void *(*)(void*)	Clones a key (provided by the user)
usr_value_clone	void *(*)(void*)	Clones a value (provided by the user)
usr_hash	uint64_t (*)(void*)	Hash function for keys
usr_stringify	char *(*)(void*, void*)	Converts a key-value pair to a string
usr_compare	int32_t (*)(void*, void*)	Compares two keys
usr_free_key	void (*)(void*)	Frees a key
usr_free_value	void (*)(void*)	Frees a value

Example Usage:

```
map_t *my_map;
map_create(&my_map, key_clone, value_clone, hash_func,
           stringify_func, compare_func, free_key, free_value);

// Insert some key-value pairs
map_insert(my_map, "name", "Alice");
map_insert(my_map, "age", "25");

// Retrieve a value
char *retrieved_value;
if (map_get(my_map, "name", (void**)&retrieved_value) == MAP_OK) {
    printf("Retrieved: %s\n", retrieved_value);
}

// Cleanup
map_destroy(&my_map);
```

4.2.2 map_element_t (Hashmap Node)

The map_element_t structure represents a single key-value pair stored in the hashmap. Each element is part of a linked list (chaining) used for collision handling.

Structure Definition:

```
typedef struct map_element {
  void *_key;              // Pointer to the key
  void *_value;            // Pointer to the value
  struct map_element *_next; // Pointer to the next element in the bucket
} map_element_t;
```

Description: Each map_element_t stores:

- A key (_key) → Points to a user-defined key.
- A value (_value) → Stores the associated value.
- A next pointer (_next) → Links to the next element in the same bucket (if any).

Since the hashmap uses separate chaining, multiple elements may exist in a single bucket, forming a linked list.

Member Breakdown:

Member	Type	Description
`_key`	`void*`	Pointer to the stored key
`_value`	`void*`	Pointer to the stored value
`_next`	`map_element_t*`	Pointer to the next element in the bucket (if any)

Example of a Bucket with Multiple Elements (Collisions)

```
// Assume 3 keys hash to the same bucket (index 5)
buckets[5] -> [_key="Alice", _value=25] -> [_key="Bob", _value=30] -> NULL
```

How It Works in the Hashmap: - New elements are added at the front of the linked list. - If a key already exists, its value is updated. - When an element is removed, the list is properly adjusted.

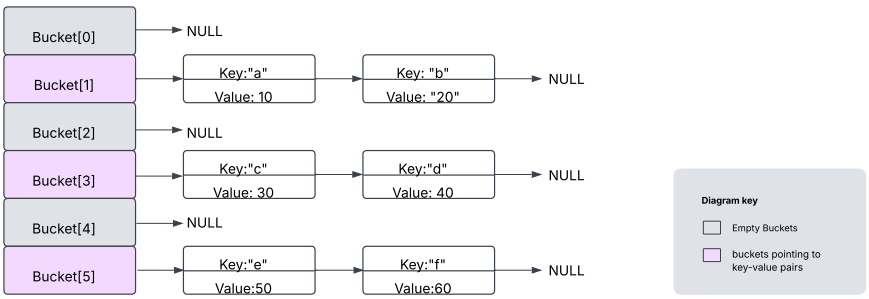


Figure 3: Illustration of Bucket Chaining in libmap

Example Usage:

```
map_element_t *elem = malloc(sizeof(map_element_t));
elem->_key = strdup("Alice");
elem->_value = malloc(sizeof(int));
*(int*)elem->_value = 25;
elem->_next = NULL; // First node in the list

// Insert another element in the same bucket
map_element_t *new_elem = malloc(sizeof(map_element_t));
new_elem->_key = strdup("Bob");
new_elem->_value = malloc(sizeof(int));
*(int*)new_elem->_value = 30;
new_elem->_next = elem; // Point to the previous element
elem = new_elem; // Update head of the list
```

4.2.3 `map_iterator_t` (Hashmap Iterator)

The `map_iterator_t` structure allows users to traverse all elements in the hashmap.

Structure Definition:

```
typedef struct {
  int32_t current_bucket;          // Index of the current bucket
  map_element_t *current_element; // Pointer to the current element in the list
} map_iterator_t;
```

Description: This structure maintains the current position in the hashmap while iterating over elements.

- `current_bucket` → Keeps track of the bucket index in the `buckets` array.
- `current_element` → Points to the current element in the linked list.

- The iterator starts at the first non-empty bucket and moves through the linked lists.

Iteration Process: 1. Find the first non-empty bucket and set `current_bucket`. 2. Set `current_element` to the first key-value pair in that bucket. 3. Move to the next element in the linked list. 4. If the list ends, move to the next non-empty bucket. 5. Repeat until all elements have been visited.

Member Breakdown:

Member	Type	Description
`current_bucket`	`int32_t`	Index of the current bucket being visited
`current_element`	`map_element_t*`	Pointer to the current element in the linked list

Example Usage:

```
map_iterator_t iter;
map_iter_start(my_map, &iter);

void *key, *value;
while (map_iter_next(my_map, &iter, &key, &value) == MAP_OK) {
    printf("Key: %s, Value: %s\n", (char*)key, (char*)value);
}
```

How It Works in the Hashmap: - `map_iter_start` initializes the iterator at the first valid element. - `map_iter_next` moves to the next element and retrieves its key-value pair. - The iteration stops when all elements have been visited.

4.3 Hashing and Collision Handling

4.3.1 What is Hashing?

A **hash function** is a function that maps keys to numeric values called **hash codes**, which determine the bucket index in the hashmap.

$$\text{index} = \text{usr\_hash(key)} \mod \text{num\_buckets}$$

The quality of the hash function significantly impacts hashmap performance.

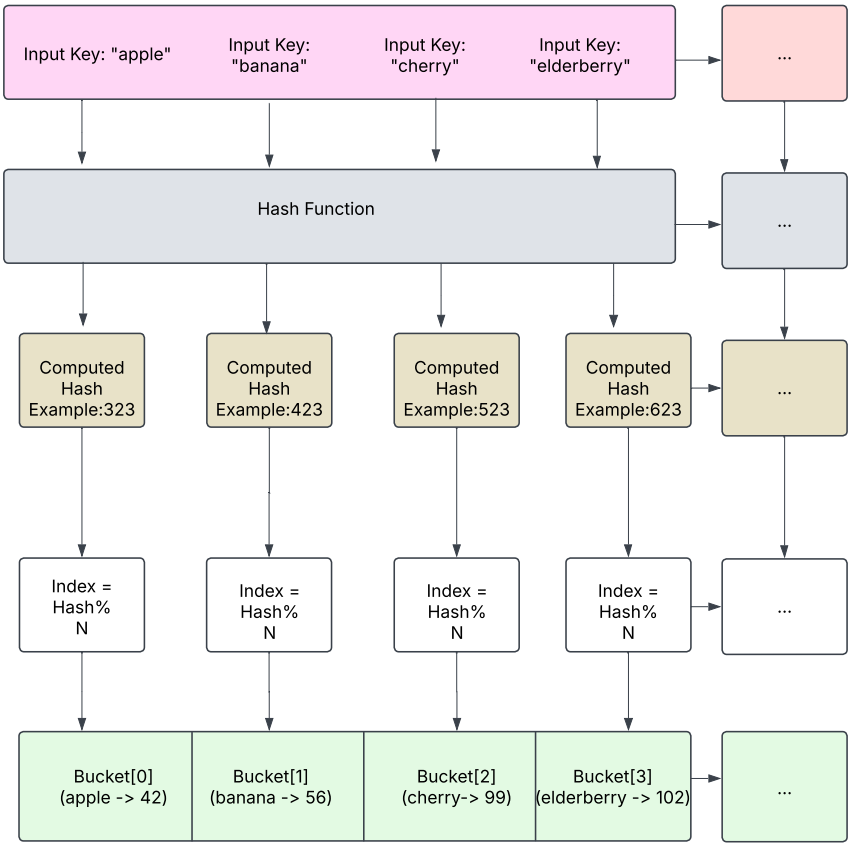


Figure 4: Example of Hash Computation and Bucket Indexing

4.3.2 User-Defined Hash Functions

Unlike some hashmaps that provide built-in hashing, `libmap` requires the user to define their own hash function via:

```
uint64_t (*usr_hash)(void *key);
```

This allows maximum flexibility but requires the user to ensure the function is efficient and well-distributed.

Example of a Poor Hash Function (Bad Distribution)

```
uint64_t bad_hash(void *key) {
    return (uint64_t) key; // Directly returning pointer value (BAD)
}
```

This function results in poor distribution, as pointers are not evenly spread across buckets.

Example of a Good Hash Function (FNV-1a)

```
uint64_t fnv1a_hash(void *key, size_t len) {
    uint64_t hash = 14695981039346656037ULL; // FNV offset basis
    for (size_t i = 0; i < len; i++) {
        hash ^= ((unsigned char*)key)[i];
        hash *= 1099511628211ULL; // FNV prime
    }
    return hash;
}
```

This function provides strong distribution and reduces clustering.

4.3.3 Why Do Hash Collisions Happen?

Since the number of possible keys is much larger than the number of buckets, different keys may hash to the same bucket, causing a **collision**. Collisions are unavoidable, so the hashmap must handle them efficiently.

4.3.4 Collision Handling via Separate Chaining

`libmap` resolves collisions using separate chaining, where each bucket contains a linked list of elements. If multiple keys hash to the same bucket: 1. The first key-value pair is stored in the bucket. 2. The second key-value pair is added as a linked list node. 3. The hashmap traverses the list to retrieve, update, or delete keys.

Example: Hash Collision Scenario

```
// Assume "Alice" and "Bob" hash to the same bucket index (e.g., index 5)
buckets[5] -> [_key="Alice", _value=25] -> [_key="Bob", _value=30] -> NULL
```

Inserting "Charlie" at index 5:

```
buckets[5] -> [_key="Charlie", _value=40] -> [_key="Alice", _value=25] ->
↪  [_key="Bob", _value=30] -> NULL
```

This maintains $O(1)$ insertion in most cases.

4.3.5 Choosing a Good Hash Function

A good hash function must: - Distribute keys uniformly across all buckets. - Avoid clustering (grouping too many keys in the same bucket). - Be fast to compute for efficiency. - Be deterministic (same input always produces the same hash).

Comparison of Hash Functions

Hash Function	Speed	Uniformity	Collisions
Pointer Casting	Fast	Poor	High
Simple Modulo	Fast	Moderate	Moderate
FNV-1a	Moderate	Good	Low
MurmurHash	Slow	Excellent	Very Low

4.3.6 Alternative Collision Handling Strategies

`libmap` uses separate chaining, but other approaches include: 1. Open Addressing – Instead of linked lists, elements are stored directly in the array, using: - Linear Probing → Find next available slot. - Quadratic Probing → Skip slots in increasing steps. 2. Perfect Hashing – A specialized technique to eliminate collisions but requires extra preprocessing.

Why Separate Chaining? - Pros: Simple, dynamic resizing is easy, supports variable key sizes. - Cons: Extra memory for linked lists, can degrade with excessive collisions.

Key Takeaways:

- Collisions are inevitable, so they must be handled properly.
- **Separate chaining** is used because it is simple and effective.
- A good hash function is crucial for reducing collisions.
- Open addressing is another option but is not used in ‘libmap‘.

4.4 Resizing Mechanism

4.4.1 Why Resizing is Necessary

A hashmap must maintain a balance between: - Efficiency: Ensuring lookups remain fast (O(1)). - Memory Usage: Preventing excessive memory allocation. - Load Factor: Maintaining an optimal

bucket-to-element ratio.

Without resizing: - If the hashmap is too full → More collisions occur, degrading performance. - If the hashmap is too empty → Wasted memory, inefficient cache usage.

4.4.2 When Does Resizing Occur?

The hashmap automatically resizes when:

- Growth Condition: If the number of elements exceeds:

$$\text{max\_load\_factor} \times \text{num\_buckets}$$

  The number of buckets increases.

- Shrink Condition: If the number of elements falls below:

$$\text{min\_load\_factor} \times \text{num\_buckets}$$

  The number of buckets decreases.

Default Load Factor Values:

Parameter	Default Value
`max_load_factor`	2.0 (Grow when `num_entries` exceeds 2x `num_buckets`)
`min_load_factor`	0.5 (Shrink when `num_entries` falls below 0.5x `num_buckets`)
`grow_factor`	2.0 (Buckets double in size)
`shrink_factor`	0.5 (Buckets shrink to half)

4.4.3 How Resizing Works (Step-by-Step)

When resizing is triggered: 1. A new bucket array is allocated with size:

$$\text{new\_num\_buckets} = \text{num\_buckets} \times \text{grow\_factor}$$

2. All existing elements are rehashed into the new array. 3. The old bucket array is freed to reclaim memory.

Code Implementation of Resizing:

```
map_error_t __map_resize(map_t *map, float resize_factor) {
    if (!map || resize_factor <= 0) return MAP_ERR_INVALID_ARG;

    uint64_t new_num_buckets = (uint64_t)(map->num_buckets * resize_factor);
    if (new_num_buckets < 1) return MAP_ERR_INVALID_ARG;

    // Allocate new bucket array
    map_element_t **new_buckets = malloc(new_num_buckets * sizeof(map_element_t
    ↪  *));
    if (!new_buckets) return MAP_ERR_NO_MEM;

    // Initialize new buckets
    for (uint64_t i = 0; i < new_num_buckets; i++) new_buckets[i] = NULL;

    // Rehash elements into new bucket array
    for (int i = 0; i < map->num_buckets; i++) {
        map_element_t *current = map->buckets[i];
        while (current) {
            map_element_t *next = current->_next;
            uint64_t new_index = map->usr_hash(current->_key) % new_num_buckets;
            current->_next = new_buckets[new_index];
            new_buckets[new_index] = current;
```

```
            current = next;
        }
    }

    // Free old buckets and update hashmap
    free(map->buckets);
    map->buckets = new_buckets;
    map->num_buckets = new_num_buckets;

    return MAP_OK;
}
```

4.4.4 Growth Strategy

The hashmap grows when:

$$\text{num\_entries} > \text{max\_load\_factor} \times \text{num\_buckets}$$

Effects of Growing:

- Reduces Collisions → More buckets spread out key-value pairs.
- Maintains O(1) Performance → Faster lookup times.
- Requires Rehashing → All elements are redistributed.

Example: Growth from 4 to 8 Buckets

```
Before Growth:
buckets[4] -> ["Alice"] -> ["Bob"] -> NULL
buckets[5] -> ["Charlie"] -> NULL

After Growth:
buckets[8] -> ["Alice"] -> NULL
buckets[9] -> ["Bob"] -> NULL
buckets[10] -> ["Charlie"] -> NULL
```

4.4.5 Shrink Strategy

The hashmap shrinks when:

$$\text{num\_entries} < \text{min\_load\_factor} \times \text{num\_buckets}$$

Effects of Shrinking:

- Saves Memory → Frees unused space.
- May Increase Collisions → Fewer buckets = more chance of collision.

Example: Shrinking from 8 to 4 Buckets

```
Before Shrinking:
buckets[8] -> ["Alice"] -> NULL
buckets[9] -> ["Bob"] -> NULL
buckets[10] -> ["Charlie"] -> NULL

After Shrinking:
buckets[4] -> ["Alice"] -> ["Bob"] -> ["Charlie"] -> NULL
```

4.4.6 Impact of Resizing on Performance

Time Complexity:

- Lookup / Insert / Delete: O(1) (Amortized)

- Resizing (Rehashing): O(n) (All elements must be moved)

Trade-offs:

- Pro: Ensures lookup stays efficient.
- Pro: Prevents excessive memory use.
- Con: Rehashing is expensive (happens rarely).

Key Takeaways:

- The hashmap grows when it becomes too full (> max load factor).
- The hashmap shrinks when it becomes too empty (< min load factor).
- Resizing helps maintain $O(1)$ efficiency but has an $O(n)$ cost.
- Choosing a good load factor is essential for performance.

4.5 Iterator Mechanism

4.5.1 Why an Iterator is Needed?

A hashmap stores key-value pairs, but since elements are distributed across multiple buckets (and may have linked lists for collisions), a direct traversal is not possible. To allow users to efficiently access all elements without exposing internal structures, `libmap` provides an **iterator-based API**.

4.5.2 How Iteration Works

Iteration involves: 1. Starting at the first non-empty bucket. 2. Retrieving elements from the linked list in that bucket. 3. Moving to the next bucket when the list ends. 4. Repeating until all elements are visited.

4.5.3 Iterator Structure (`map_iterator_t`)

The iterator tracks the traversal position:

```
typedef struct {
  int32_t current_bucket;          // Index of the current bucket
  map_element_t *current_element; // Pointer to the current element in the list
} map_iterator_t;
```

Member Breakdown:

Member	Type	Description
`current_bucket`	`int32_t`	Index of the bucket currently being iterated
`current_element`	`map_element_t*`	Pointer to the current element in the linked list

4.5.4 Iterator Functions

Two functions are provided for iteration:

1. Start Iteration (`map_iter_start`) Initializes the iterator to the first valid key-value pair.

```
map_error_t map_iter_start(const map_t *map, map_iterator_t *iter) {
    if (!map || !iter) return MAP_ERR_INVALID_ARG;

    iter->current_bucket = -1;
    iter->current_element = NULL;

    for (int i = 0; i < map->num_buckets; i++) {
        if (map->buckets[i] != NULL) {
            iter->current_bucket = i;
            iter->current_element = map->buckets[i];
```

```
            return MAP_OK;
        }
    }

    return MAP_ERR_END_OF_MAP;
}
```

2. Retrieve Next Element (`map_iter_next`) Returns the next key-value pair and advances the iterator.

```
map_error_t map_iter_next(const map_t *map, map_iterator_t *iter,
                          void **out_key, void **out_value) {
    if (!map || !iter || !out_key || !out_value) return MAP_ERR_INVALID_ARG;

    // If inside a bucket's linked list, move to next element
    if (iter->current_element != NULL) {
        *out_key = iter->current_element->_key;
        *out_value = iter->current_element->_value;
        iter->current_element = iter->current_element->_next;
        return MAP_OK;
    }

    // Move to the next non-empty bucket
    while (++iter->current_bucket < map->num_buckets) {
        if (map->buckets[iter->current_bucket] != NULL) {
            iter->current_element = map->buckets[iter->current_bucket];
            *out_key = iter->current_element->_key;
            *out_value = iter->current_element->_value;
            iter->current_element = iter->current_element->_next;
            return MAP_OK;
        }
    }

    return MAP_ERR_END_OF_MAP;
}
```

4.5.5 Example Usage

```
map_iterator_t iter;
map_iter_start(my_map, &iter);

void *key, *value;
while (map_iter_next(my_map, &iter, &key, &value) == MAP_OK) {
    printf("Key: %s, Value: %s\n", (char*)key, (char*)value);
}
```

4.5.6 Performance Considerations

Time Complexity:

- Best Case: O(1) per element (if minimal collisions).
- Worst Case: O(n) (if all elements are in a single bucket).

Trade-offs:

- Pro: Provides a clean way to iterate without exposing internal structure.
- Pro: Avoids modifying hashmap state.
- Con: Linked list traversal can be slower in high-collision scenarios.

Key Takeaways:

- The iterator allows users to traverse all elements in a controlled way.
- Uses separate chaining, so it must handle linked lists.
- Lookup time is O(1) per element on average, but depends on bucket distribution.

4.6 Trade-offs and Design Choices

4.6.1 Choosing Separate Chaining vs. Open Addressing

A key design choice in `libmap` was using separate chaining rather than open addressing for collision handling.

Why Separate Chaining?

- Handles collisions well – Each bucket stores a linked list, so multiple keys can exist in the same bucket.
- Supports dynamic resizing easily – Rehashing only requires updating pointers rather than shifting elements.
- Works well with variable-sized keys – Open addressing struggles with non-fixed size data.
- Deletion is simple – Removing an element just involves updating linked list pointers.

Why Not Open Addressing?

- Suffers from clustering – Linear/quadratic probing leads to long lookup chains.
- More expensive resizing – Requires rehashing all elements into a new array.
- Poor performance at high load factors – Performance degrades as the table fills up.

Comparison Table:

Technique	Pros	Cons
Separate Chaining	Simple, handles collisions well	Uses extra memory for linked lists
Open Addressing	Memory efficient	Bad performance with collisions, complex resizing

4.6.2 Dynamic Resizing Strategy

`libmap` dynamically resizes when the load factor exceeds a threshold.

Why Resize at Load Factor 2.0?

- Keeps collision probability low – Ensures efficient lookups.
- Balances memory usage – Prevents excessive growth while keeping performance optimal.

Why Shrink at Load Factor 0.5?

- Avoids wasted space – Reduces memory footprint when usage decreases.
- Prevents excessive allocations/deallocations – Helps maintain efficiency.

4.6.3 User-Defined Hash Functions: Pros and Cons

Unlike some libraries that provide built-in hashing, `libmap` requires the user to define a hash function.

Why Allow User-Defined Hash Functions?

- Flexibility – Users can optimize hash functions for their specific data.
- Supports non-standard key types – Works with custom structs, non-string keys.
- Avoids unnecessary computation – Some applications may already have hashed keys.

Risks and Potential Issues of User-Defined Functions While user-defined functions provide flexibility, they introduce risks that can negatively impact performance and correctness if not implemented carefully.

1. Poor Hash Functions Cause Performance Degradation

```
uint64_t bad_hash(void *key) {
    return 1; // Maps all keys to the same bucket (worst case scenario)
}
```

2. Hash Functions Must Be Deterministic

```
uint64_t nondet_hash(void *key) {
    return rand(); // Random values cause lookups to fail
}
```

3. Comparison Function Errors Can Cause Incorrect Lookups

```
int bad_compare(void *key1, void *key2) {
    return 2; // Should return 0 for equal keys, nonzero otherwise
}
```

4. Stringification Functions Can Leak Memory

```
char *bad_stringify(void *key, void *value) {
    char buffer[100];
    snprintf(buffer, sizeof(buffer), "%s -> %s", (char*)key, (char*)value);
    return buffer; // Returning stack memory! Causes undefined behavior.
}
```

Best Practices for User-Defined Functions To avoid the above issues, follow these rules:

- Use strong hash functions like MurmurHash or xxHash.
- Ensure the comparison function follows strict equality rules.
- Properly free cloned keys and values in deallocation functions.
- Test hash distribution by inserting multiple keys and checking for collisions.

4.6.4 Balancing Memory vs. Performance

Hashmaps must balance speed and memory efficiency. `libmap` makes the following design choices:

- Optimized for speed – Keeps lookup, insertion, and deletion at O(1) time complexity.
- Accepts extra memory usage – Uses linked lists in buckets to handle collisions.

Memory vs. Performance Trade-offs:

Optimization	Effect
More Buckets	Faster lookups, more memory usage
Fewer Buckets	Saves memory, increases collisions
Separate Chaining	Easy deletions, more memory overhead
Open Addressing	Compact memory, hard to resize

4.6.5 Future Optimizations

Potential areas for improvement in future versions of `libmap`:

1. Faster Hashing Algorithms - Support built-in fast hash functions for common data types.

2. Improved Collision Resolution - Use linked list sorting to make lookups faster in high-collision scenarios.

3. Better Memory Management - Reduce malloc/free calls.

4.6.6 Key Takeaways

- **Separate Chaining** was chosen over Open Addressing for simplicity and flexibility.
- Resizing keeps performance optimal by maintaining a low load factor.
- **User-defined hashing** provides flexibility but requires careful implementation.
- Future versions can explore **built-in fast hash functions, optimized memory management, and smarter iteration**.

5 Testing

5.1 Introduction to Testing

Testing is a **critical** part of `libmap`, ensuring that the hashmap implementation behaves correctly under different scenarios. To achieve this, we have designed a **comprehensive suite of automated tests** that cover various aspects of functionality, correctness, and performance.

All tests in `libmap` are **automated**, meaning they run **without manual intervention**. Each test case uses **assertions** (`assert()`) to verify expected behavior, ensuring that any failure **immediately stops execution** and prints an error message. This approach eliminates the need for manual verification and makes debugging easier.

Additionally, performance testing is a key part of `libmap` validation. We use **Flamegraphs** to analyze execution time and identify performance bottlenecks, ensuring that operations like **insertion, retrieval, and resizing** are efficient even for large datasets.

5.2 Testing Categories

Our tests are divided into the following categories:

- **Unit Tests** (Verify individual functions)
  - Each function (e.g., `map_create()`, `map_insert()`, `map_get()`) is tested independently.
  - Tests include **edge cases**, invalid inputs, and function-specific logic.
- **Integration Tests** (Verify end-to-end correctness)
  - Tests real-world scenarios like **inserting thousands of elements** and ensuring correctness through iteration, retrieval, and removal.
- **Performance Testing with Flamegraphs**
  - `test_map_torture.c` runs a **large-scale performance test** by inserting and manipulating **10 million key-value pairs**.
  - A **Flamegraph** is generated to analyze function execution time and detect slowdowns.
- **Error Handling Tests**
  - We simulate failure cases, such as:
    - Passing `NULL` pointers.
    - Providing broken user-defined functions.
    - Ensuring correct error codes are returned.

By structuring our tests into these categories, `libmap` ensures **correctness, robustness, and high performance** under different conditions.

5.2.1 `test_map_create.c` - Hashmap Creation Test

Purpose: This test verifies that the `map_create()` function correctly initializes an empty hashmap and properly handles invalid arguments. It ensures that memory is allocated, internal structures are initialized, and invalid inputs are properly rejected.

Test Cases:

- **Valid Initialization:** Ensures that `map_create()` correctly allocates and initializes a hashmap.
- **Invalid Argument Handling:** Tests whether passing `NULL` as an argument returns an error.
- **Resource Cleanup:** Ensures that the hashmap is correctly deallocated using `map_destroy()`.

Breakdown and Explanation: This test covers the following key aspects of `map_create()`:

- **Valid Initialization:** The test calls `map_create()` with valid function pointers for key cloning, value cloning, hashing, stringifying, comparison, and memory management. If successful, it returns `MAP_OK`, confirming that the hashmap has been allocated and initialized properly.
- **Memory Allocation:** The user-defined `dummy_int_key_clone()` and `dummy_int_value_clone()` functions allocate memory for cloned keys and values. This ensures that the hashmap owns its copies of inserted data.
- **Hash Function Integration:** The test includes a simple modulus-based hash function `dummy_user_hash()`, which ensures uniform distribution of keys across hashmap buckets.
- **Invalid Argument Handling:** Passing a `NULL` pointer to `map_create()` should return an error (`MAP_ERR_INVALID_ARG`), preventing segmentation faults or undefined behavior.
- **Cleanup and Destruction:** The test ensures that after the map is created, it is properly cleaned up using `map_destroy()`. If `map_destroy()` does not correctly free memory, tools like `valgrind` will report memory leaks.

Key Validations:

- `map_create()` successfully initializes a hashmap.
- Passing a `NULL` pointer returns `MAP_ERR_INVALID_ARG`.
- The allocated hashmap is properly destroyed with `map_destroy()`.

5.2.2 `test_map_destroy.c` - Hashmap Destruction Test

Purpose: This test ensures that the `map_destroy()` function correctly deallocates the hashmap, preventing memory leaks and handling edge cases safely.

Test Cases:

- **Valid Destruction:** Ensures that calling `map_destroy()` frees all allocated memory.
- **Repeated Destruction:** Checks if calling `map_destroy()` multiple times on the same hashmap is safe.
- **Null Pointer Handling:** Ensures that passing a `NULL` pointer to `map_destroy()` does not cause segmentation faults.

Breakdown and Explanation:

This test covers the following key aspects of `map_destroy()`:

- **Proper Memory Deallocation:** The function should free all allocated memory, including:
  - The main `map_t` structure.
  - All stored `map_element_t` nodes.
  - User-defined keys and values using `usr_free_key()` and `usr_free_value()`.
- **Ensuring Safe Double-Free Protection:** - After calling `map_destroy()`, the map pointer should be set to `NULL`. - Calling `map_destroy()` on an already destroyed hashmap should return `MAP_ERR_INVALID_ARG` instead of crashing.
- **Handling Edge Cases:** - If `map_destroy()` is called on an empty map, it should still function correctly. - If `map_destroy()` is called on a `NULL` pointer, it should not crash.

Key Validations:

- `map_destroy()` successfully deallocates the hashmap.
- Repeated destruction of the same hashmap does not cause errors.
- The `map` pointer is correctly set to `NULL` after destruction.

5.2.3 `test_map_insert.c` - Hashmap Insertion Test

Purpose: This test ensures that the `map_insert()` function correctly adds key-value pairs, handles duplicate keys, and triggers resizing when necessary.

Test Cases:

- **Basic Insertion:** Verify that key-value pairs are inserted correctly.
- **Duplicate Key Handling:** Ensure that inserting an existing key updates the associated value.
- **Resizing Behavior:** Test whether the hashmap resizes when the load factor exceeds the threshold.
- **Invalid Arguments:** Ensure that inserting `NULL` as a key or value is handled properly.

Breakdown and Explanation:

This test covers the following key aspects of `map_insert()`:

- **Correct Data Insertion:** - The test ensures that new key-value pairs are inserted into the hashmap and can be retrieved correctly.
- **Duplicate Key Handling:** - When a key that already exists is inserted again, its value should be updated rather than inserting a duplicate entry.
- **Resizing Behavior:** - The hashmap dynamically resizes when the number of elements exceeds the configured load factor, preventing excessive collisions.
- **Invalid Argument Handling:** - If `NULL` is passed as a key, `map_insert()` should return `MAP_ERR_INVALID_ARG`.
- **Memory Safety:** - Keys and values are cloned before insertion, ensuring that external modifications to original data do not affect stored values.

Key Validations:

- `map_insert()` successfully inserts key-value pairs.
- Duplicate keys update existing values instead of creating duplicates.
- Hashmap resizes when the threshold is exceeded.
- Inserting a `NULL` key returns `MAP_ERR_INVALID_ARG`.
- The hashmap is properly cleaned up after the test.

5.2.4 `test_map_get.c` - Hashmap Retrieval Test

Purpose: This test ensures that the `map_get()` function correctly retrieves values associated with keys, properly handles missing keys, and correctly validates the `usr_compare` function.

Test Cases:

- **Valid Retrieval:** Ensures that values inserted into the hashmap can be correctly retrieved.
- **Non-Existent Key Handling:** Tests whether retrieving a key that does not exist returns `MAP_ERR_NOT_FOUND`.
- **Broken Comparison Function:** Checks the behavior when `usr_compare` is incorrectly implemented.

Breakdown and Explanation:

This test covers the following key aspects of `map_get()`:

- **Correct Retrieval:** - The test ensures that key-value pairs inserted into the hashmap can be retrieved correctly.

- **Handling of Missing Keys:** - If a key is not found, `map_get()` should return `MAP_ERR_NOT_FOUND` instead of an invalid value.
- **Broken Comparison Function Handling:** - If the user provides a faulty `usr_compare` function that does not return valid comparison values ({-1, 0, 1}), the test ensures that `MAP_ERR_UNKNOWN` is returned.
- **Memory Safety:** - The function does not modify or free memory that it does not own. The retrieved value is still managed by the hashmap.

Key Validations:

- `map_get()` correctly retrieves values for valid keys.
- `map_get()` returns `MAP_ERR_NOT_FOUND` for missing keys.
- A faulty `usr_compare` function results in `MAP_ERR_UNKNOWN`.
- The hashmap remains consistent and free of memory leaks after operations.

5.2.5 `test_map_remove.c` - Hashmap Deletion Test

Purpose: This test ensures that the `map_remove()` function correctly deletes key-value pairs, properly updates the hashmap size, and handles cases where keys do not exist.

Test Cases:

- **Valid Removal:** Ensures that an existing key-value pair can be removed.
- **Non-Existent Key Handling:** Checks whether attempting to remove a key that does not exist returns `MAP_ERR_NOT_FOUND`.
- **Hashmap Shrinking:** If many elements are removed, the hashmap should shrink based on the load factor.

Breakdown and Explanation:

This test covers the following key aspects of `map_remove()`:

- **Correct Deletion:** - Ensures that calling `map_remove()` correctly deletes a key-value pair.
- **Handling of Missing Keys:** - If a key is not found, `map_remove()` should return `MAP_ERR_NOT_FOUND`.
- **Updating Size:** - The test verifies that the hashmap size decreases after each successful removal.
- **Shrinking Mechanism:** - When many elements are removed, the hashmap should shrink its bucket array if the load factor drops below the configured minimum.
- **Memory Safety:** - The function should not modify or free memory that it does not own.

Key Validations:

- `map_remove()` correctly deletes existing keys.
- `map_remove()` returns `MAP_ERR_NOT_FOUND` for missing keys.
- The hashmap correctly updates its size after removals.
- The hashmap correctly shrinks when necessary.
- Memory remains consistent and free of leaks.

5.2.6 `test_map_chaining.c` - Collision Handling Test

Purpose: This test ensures that the hashmap correctly handles hash collisions using separate chaining. A custom hash function is used to force all keys into the same bucket, ensuring that chaining is tested.

Test Cases:

- **Forced Collisions:** Ensures multiple keys hash to the same bucket.
- **Retrieval After Collision:** Confirms that all inserted values are retrievable despite collisions.
- **Deletion in Chained Buckets:** Checks that deleting an element from a chained list does not break the structure.

Breakdown and Explanation:

This test ensures that separate chaining is working correctly by forcing all keys to hash to the same bucket.

- **Forced Collisions:** - A fake hash function returns the same value for all keys, ensuring all elements collide into the same bucket.
- **Insertion and Linked List Growth:** - The hashmap must dynamically manage multiple elements in a single bucket as a linked list.
- **Retrieval from a Chained Bucket:** - The test confirms that all values can be retrieved even when they are stored in a linked list at the same bucket.
- **Deletion in Chained Buckets:** - When an element is removed, the linked list should remain intact, and the remaining elements should still be accessible.
- **Correct Size Updates:** - After deletions, `map_get_size()` should reflect the correct number of elements remaining.
- **Memory Safety:** - The test ensures that all allocated memory is freed correctly when `map_destroy()` is called.

Key Validations:

- All inserted elements are retrievable, proving chaining works.
- After removing an element, the remaining elements in the bucket remain accessible.
- The hashmap correctly updates its size after insertions and deletions.
- Memory is correctly managed and freed upon destruction.

5.3 Integration Tests

Purpose: Integration tests verify that multiple components of the hashmap work together correctly. While unit tests check individual functions in isolation, integration tests ensure that operations like insertion, retrieval, iteration, and removal work as expected in real-world scenarios.

Key Focus Areas:

- Ensuring the hashmap maintains consistency when handling a large number of entries.
- Validating that insertion, retrieval, and removal operations function correctly when used together.
- Testing edge cases like hashmap resizing, iteration over entries, and high-load conditions.
- Evaluating the performance of the hashmap under stress (e.g., inserting 10 million elements).

The following integration tests have been implemented:

- `test_map_consistency.c` – Inserts and verifies a large number of key-value pairs, removes a range of elements, and checks iteration behavior.
- `test_mega_map.c` – General large-scale test (user-defined behavior).
- `test_map_torture.c` – Inserts 10 million random key-value pairs, modifies them, retrieves values, and then removes all elements.

Each test case is designed to ensure that `libmap` remains reliable, even under extreme workloads.

5.3.1 `test_map_consistency.c` - Consistency and Iteration Test

Purpose: This test verifies the integrity of the hashmap after sequential insertions, retrievals, iterations, and removals. It ensures:

- Elements are stored and retrieved correctly.
- The iterator correctly traverses all elements.
- Removal correctly reduces the number of elements.

Test Cases:

- Insert keys in the range $[1, 1000]$, mapping each key to its square.
- Verify all inserted keys exist using `map_get()`.
- Iterate over the map and verify all elements are counted.
- Remove elements in the range $[250, 750)$.
- Iterate again to confirm that exactly 500 elements remain.

Breakdown and Explanation: This test ensures that:

- **Insertion Consistency:** All keys from 1 to 1000 are correctly stored with their square as values.
- **Accurate Retrieval:** Each key is retrieved and checked for correctness.
- **Iterator Validation:** The iterator successfully counts all elements.
- **Correct Removal Behavior:** Keys in the range $[250, 750)$ are removed.
- **Final Consistency Check:** The iterator confirms that 500 elements remain.

5.3.2 `test_mega_map.c` - Large-Scale Hashmap Operations

Purpose: This test performs large-scale hashmap operations, including insertion, retrieval, iteration, and removal. It ensures:

- The hashmap can store and manage a variety of string keys.
- Retrieval and removal operations work correctly.
- The iterator traverses all elements as expected.
- The hashmap correctly deallocates memory upon destruction.

Test Cases:

- Insert multiple key-value pairs into the hashmap.
- Retrieve specific elements to verify correctness.
- Iterate over the entire hashmap and print all elements.
- Remove a key and ensure it is no longer retrievable.
- Destroy the hashmap and verify cleanup.

Breakdown and Explanation: This test validates the correctness of:

- **Insertion:** Multiple string keys are inserted into the hashmap.
- **Retrieval:** The function `map_get()` correctly retrieves a key-value pair.
- **Iteration:** The iterator traverses and prints all elements.
- **Removal:** The key `"banana"` is successfully removed.
- **Destruction:** The hashmap is properly deallocated with `map_destroy()`.

5.3.3 `test_map_torture.c` - Large-Scale Hashmap Stress Test

Purpose: This test evaluates the performance and robustness of the hashmap by inserting, modifying, retrieving, and deleting a large number of elements. It ensures:

- The hashmap can handle a large number of entries (10 million).
- The insertions and retrievals perform efficiently under high load.
- The iteration mechanism functions correctly with large datasets.
- The hashmap correctly deallocates memory after bulk removals.

Test Cases:

- Insert 10 million random key-value pairs.
- Iterate over the hashmap and modify all values.
- Retrieve random keys and validate modified values.
- Remove all elements from the hashmap.
- Ensure proper cleanup and memory deallocation.

Breakdown and Explanation: This test ensures the hashmap remains performant and functional under extreme load:

- **Insertion:** 10 million elements are inserted into the hashmap.
- **Iteration and Modification:** The iterator updates all values in the hashmap.
- **Retrieval:** The test verifies the correctness of modified values.
- **Removal:** All elements are deleted, and the hashmap is deallocated.

Performance Considerations:

- Since the test involves 10 million elements, memory usage should be monitored.
- Hash collisions may occur, but the chaining mechanism ensures proper handling.
- Using a large prime number (1,000,003) in the hash function improves distribution.

5.4 Summary of Testing

To ensure the correctness, stability, and efficiency of `libmap`, we designed a comprehensive test suite covering both **unit tests** and **integration tests**. These tests validate that the hashmap behaves as expected across different scenarios, including boundary conditions, error handling, and performance under stress.

What is Testing and Why is it Important? In software development, testing is the process of running a program with various inputs to verify its correctness. A well-tested library ensures:

- **Correct behavior**: The library functions work as intended.
- **Reliability**: The library performs consistently under different conditions.
- **Error handling**: The library can detect and gracefully handle invalid inputs.
- **Performance**: The library remains efficient even with large datasets.
- **Memory safety**: No memory leaks or segmentation faults occur.

For `libmap`, we use two main testing strategies: unit testing and integration testing.

Recap of Unit Tests Unit tests focus on testing **one function at a time in isolation**. Each unit test targets a **specific function** in `libmap`, ensuring it behaves correctly under normal and edge cases.

- **Correctness**: Ensures functions perform as expected for valid inputs.
- **Error Handling**: Verifies that invalid inputs return appropriate error codes.
- **Memory Management**: Confirms that allocated memory is properly freed to prevent leaks.

Unit Tests Implemented:

Test Name	Purpose
`test_map_create.c`	Ensures correct hashmap initialization.
`test_map_destroy.c`	Checks that memory is freed properly.
`test_map_insert.c`	Validates insertion and automatic resizing.
`test_map_get.c`	Ensures key retrieval works and missing keys are handled correctly.
`test_map_remove.c`	Tests removal, ensuring resizing occurs when needed.
`test_map_chaining.c`	Forces hash collisions and verifies separate chaining works correctly.

Table 1: Overview of Unit Tests in `libmap`

Recap of Integration Tests Unit tests ensure individual functions work, but integration tests check how multiple functions work together. These tests simulate real-world usage of the hashmap.

- **Consistency**: Ensures that repeated insertions, retrievals, and removals maintain data integrity.
- **Resilience**: Tests whether `libmap` can handle large datasets and still perform efficiently.
- **Iterator Safety**: Ensures that iterators behave correctly while modifying the hashmap.

Integration Tests Implemented:

Test Name	Purpose
`test_map_consistency.c`	Inserts 1000 elements, verifies retrieval, iterates, removes a range, and checks consistency.
`test_map_torture.c`	Inserts ten million elements, modifies values, retrieves keys, and removes all entries.
`test_mega_map.c`	Large-scale tests simulating real-world use cases.

Table 2: Overview of Integration Tests in `libmap`

Aspect	Unit Testing	Integration Testing
Scope	Single function	Multiple functions working together
Purpose	Verify function correctness	Ensure system-wide stability
Complexity	Simple, focused tests	More complex interactions
Example	Testing `map_insert()` in isolation	Testing insertion, retrieval, and deletion together

Table 3: Comparison of Unit Testing and Integration Testing

Key Differences Between Unit and Integration Testing

Why Testing Matters? Testing is not just about catching bugs – it is about guaranteeing that the library is **reliable, predictable, and performant**. The tests designed for `libmap` ensure:

- **Correct behavior** under normal and edge-case scenarios.
- **Robust error handling** to prevent crashes and undefined behavior.
- **Memory safety** by ensuring no memory leaks or invalid accesses occur.
- **Performance validation** under high loads.

By running these tests across different environments, we increase confidence that `libmap` can be used in real-world applications without unexpected failures.

Key Takeaways

- **Unit tests** verify that individual components of `libmap` function correctly.
- **Integration tests** ensure these components work together seamlessly.
- **Error handling tests** confirm that `libmap` gracefully handles invalid operations.

With a well-structured test profile, we can confidently say that `libmap` is stable, efficient, and ready for production use.

5.5 Performance Analysis

Why Analyze Performance?

Hashmaps are widely used due to their expected $O(1)$ average-case time complexity for insertions, deletions, and lookups. However, real-world performance is influenced by several factors:

- **Collision Resolution:** Excessive collisions can degrade performance to `O(n)` in worst-case scenarios.
- **Resizing Overhead:** Frequent growth and reallocation of buckets may introduce performance penalties.
- **User-Defined Functions:** Inefficient `usr_hash`, `usr_compare`, and memory management functions can cause slowdowns.

To ensure optimal performance, we conduct runtime profiling to identify bottlenecks and analyze execution time.

Profiling and Runtime Performance

Profiling helps analyze how much CPU time is spent in different functions, allowing optimization of critical sections. The key profiling metrics include:

Metric	Description
Execution Time	Total runtime of hashmap operations.
Function Hotspots	Functions consuming the most CPU time.
Resizing Frequency	Number of hashmap resizes triggered.
Collision Handling Overhead	Time spent resolving key collisions.

Flamegraph Generation for `test_map_torture.c`

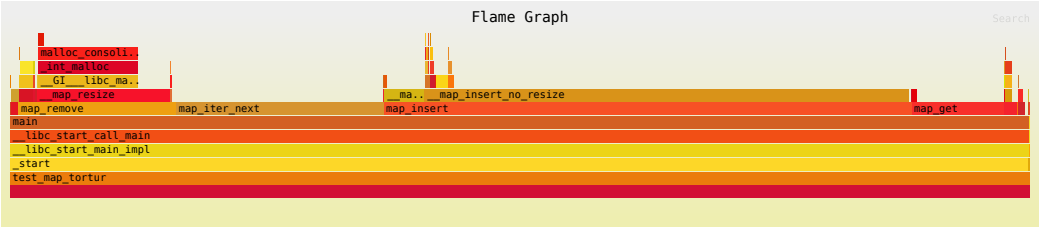
Since `test_map_torture.c` performs over ten million operations, it is the most suitable test for profiling and stress testing. A Flamegraph, which visualizes function execution time, was generated to identify potential inefficiencies.

The next section presents an in-depth analysis of the profiling results from `test_map_torture.c`.

5.5.1 Performance Analysis: Flamegraph Interpretation

Introduction to Flamegraphs: A **flamegraph** is a visualization of CPU time spent in different functions during execution. The width of each function block represents its proportion of total execution time. Functions lower in the stack are called by the ones above them.

Flamegraph for `test_map_torture.c`: The performance profile for the large-scale stress test `test_map_torture.c` reveals key computational hotspots, particularly in insertion, resizing, iteration, and removal.



5.6 Key Performance Insights and Optimization Strategies

Introduction: Performance analysis revealed several computational bottlenecks in `libmap`, primarily during insertion, resizing, iteration, and removal. This section breaks down these challenges, explains why they occur, and proposes optimization strategies to enhance efficiency.

5.6.1 Identified Bottlenecks

- **Insertion Costs (`map_insert()`)** Frequent memory allocations for keys, values, and hashmap nodes make insertion expensive, especially when the hashmap grows dynamically.
- **Resizing Overhead (`__map_resize()`)** Resizing involves allocating new buckets, Rehashing all existing keys and moving them to new buckets increases execution time.
- **Iteration Costs (`map_iter_next()`)** Iterating through a hashmap with chained buckets requires traversing linked lists, which becomes costly when chains grow due to collisions.
- **Deletion Overhead (`map_remove()`)** Removing elements requires searching a bucket and handling chained elements, making it slower when collisions are frequent.
- **Memory Allocation Overhead (`malloc()`)** Frequent allocations and deallocations cause fragmentation, impacting overall performance.

5.6.2 Understanding Why These Bottlenecks Occur

Each of these bottlenecks stems from specific design choices:

- **Insertion Slowness:** Each insertion dynamically allocates memory for keys, values, and nodes, increasing the number of `malloc()` calls.
- **Resizing Complexity:** When the load factor threshold is exceeded, resizing and rehashing require redistributing all stored elements.
- **Iteration Overhead:** Since elements are stored in linked lists within buckets, iterating requires multiple pointer dereferences.
- **Deletion Cost:** Searching and removing from linked lists involves additional pointer traversals and memory management.

5.6.3 Proposed Optimizations

The following strategies address the identified bottlenecks:

Problem	Optimization Strategy	Expected Impact
Insertion overhead	Use memory pooling to reduce calls to `malloc()`	Faster insertions, reduced memory fragmentation.
Slow iteration	Optimize bucket traversal (e.g., store precomputed element count)	Faster hashmap iteration.
High deletion costs	Use tombstones instead of immediate deallocation	Faster key removal with deferred memory reuse.
Memory fragmentation	Pre-allocate memory in bulk instead of per-insertion	Reduced allocation overhead and fragmentation.

Table 4: Optimization Strategies for `libmap` Performance Improvement

5.6.4 Conclusion

The primary performance constraints in `libmap` arise from dynamic memory allocation, resizing overhead, and chained traversal during iteration. The proposed optimizations – reducing unnecessary `malloc()` calls, improving bucket resizing policies, and refining traversal mechanisms – can significantly enhance performance.

Further improvements could involve benchmarking different allocation strategies and testing alternate hashing techniques to optimize load distribution.

5.7 CI/CD using GitHub Actions

5.7.1 Introduction to CI/CD

Overview: CI/CD (**Continuous Integration and Continuous Deployment**) is an automated pipeline that ensures code quality, correctness, and stability. By using **GitHub Actions**, every commit and push to the repository triggers automated builds and tests, preventing regressions before they reach production.

Why CI/CD for `libmap`? Since `libmap` is a low-level C library with a strong focus on performance and correctness, CI/CD plays a critical role in ensuring:

- **Automated testing:** Every push runs the test suite, catching bugs early.
- **Memory safety:** Valgrind is used to detect memory leaks and invalid accesses.
- **Consistent builds:** Ensures that `libmap` compiles and runs correctly across environments.
- **Faster debugging:** Immediate feedback from CI logs helps identify and resolve issues quickly.

How GitHub Actions Helps: `libmap` uses **GitHub Actions** for CI/CD. This means that **whenever a developer pushes code**, GitHub automatically:

1. Checks out the latest code from the repository.
2. Installs necessary dependencies (e.g., Valgrind for memory checking).
3. Compiles and builds `libmap`.
4. Runs all unit tests and integration tests.
5. Runs Valgrind to check for memory leaks.
6. Reports results and logs failures if tests fail.

5.7.2 GitHub Actions Workflow

Overview: The CI/CD pipeline for `libmap` is implemented using a GitHub Actions workflow file. This workflow automates the process of building, testing, and validating the library on every push.

The workflow file is located at:

`.github/workflows/run_hashmap_tests.yml`

Workflow Structure: The workflow consists of multiple steps, each performing a key task in the CI/CD pipeline:

1. The workflow runs on every push to the repository.
2. Clones the latest version of the repository.
3. Installs necessary tools (e.g., Valgrind).
4. Compiles the `libmap` library.
5. Executes all unit and integration tests.
6. Runs Valgrind to detect memory leaks.

Workflow YAML File: Below is the actual GitHub Actions workflow file used for `libmap`:

```yaml
name: hashmap regression test

on: [push]

jobs:
  run_hashmap_tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Install dependencies
        run: |
          sudo rm /var/lib/man-db/auto-update
          sudo apt-get remove --purge man-db
          sudo apt-get install valgrind

      - name: Build libmap
        run: |
          cd hashmap
          make

      - name: Run all tests
        run: |
          find hashmap/bin -type f -executable | xargs -I {} bash -c '{}; exit $?'

      - name: Valgrind all tests
        run: |
          find hashmap/bin -type f -executable | xargs -I {} bash -c 'valgrind
          ↪  --error-exitcode=1 --leak-check=full --track-origins=yes "{}"; exit
          ↪  $?'
```

Explanation of Key Steps:

- **Checkout Repository:** Retrieves the latest code.
- **Install Dependencies:** Ensures Valgrind is available.
- **Build:** Runs `make` to compile `libmap`.
- **Run Tests:** Executes all test cases inside `hashmap/bin`.
- **Valgrind Analysis:** Runs all tests under Valgrind to detect memory issues.

Why This Workflow? This workflow ensures that `libmap` is continuously tested for correctness and memory safety without manual intervention. If any test fails, the workflow provides immediate feedback, allowing developers to fix issues before pushing changes.

5.7.3 Interpreting CI Results

Overview: Once the CI/CD workflow is triggered, GitHub Actions provides a detailed log of each step's execution. This section explains how to analyze the results and debug potential failures.

Accessing Workflow Logs: To view the CI/CD results:

1. Navigate to the repository on GitHub.
2. Click on the **Actions** tab.
3. Select the most recent workflow run.
4. Click on individual steps to expand detailed logs.

Understanding Test Results: Each step in the workflow outputs a success or failure status. The following table summarizes possible outcomes and their meanings:

Step	Expected Outcome
`Install dependencies`	System dependencies are installed successfully (e.g., Valgrind).
`Build libmap`	The `make` command compiles the library without errors.
`Run all tests`	All unit and integration tests pass with exit code 0.
`Valgrind all tests`	No memory leaks or invalid accesses detected.

Table 5: Expected Outcomes for CI/CD Steps

Common Errors and Debugging: If a step fails, the logs provide useful debugging information. Below are common failure scenarios and their solutions:

Failure	Possible Cause and Fix
`Build failed`	Syntax error, missing files, or incorrect `Makefile`. Check the logs for compilation errors.
`Tests failed`	A test case failed. Run the tests locally and verify output.
`Valgrind errors`	Memory leak or invalid access detected. Run tests locally with `valgrind -leak-check=full`.

Table 6: Common CI/CD Failures and Debugging Tips

Example: Test Failure Log If a test fails, the logs might show output like this:

```
Running test_map_insert...
Assertion failed...
```

This indicates that `map_insert()` did not behave as expected. Debugging steps:

- Reproduce the failure locally by running `./bin/test_map_insert`.
- Use `printf()` or a debugger to inspect values.
- If Valgrind detected a memory issue, analyze the leak report.

Conclusion: Understanding CI/CD logs is essential for debugging failures efficiently. By analyzing workflow outputs and using local debugging tools, developers can quickly identify and resolve issues.

5.7.4 Conclusion

CI/CD Ensures Stability: The integration of GitHub Actions into the development workflow of `libmap` has significantly improved code quality by ensuring that:

- Every commit and pull request undergoes rigorous testing.
- Memory issues and regressions are detected early using Valgrind.
- The build remains stable, preventing broken commits from being merged.
- Performance and correctness are continuously validated.

Key Takeaways:

- Automated Testing: Every push triggers unit and integration tests.
- Memory Safety Checks: Valgrind prevents memory leaks.

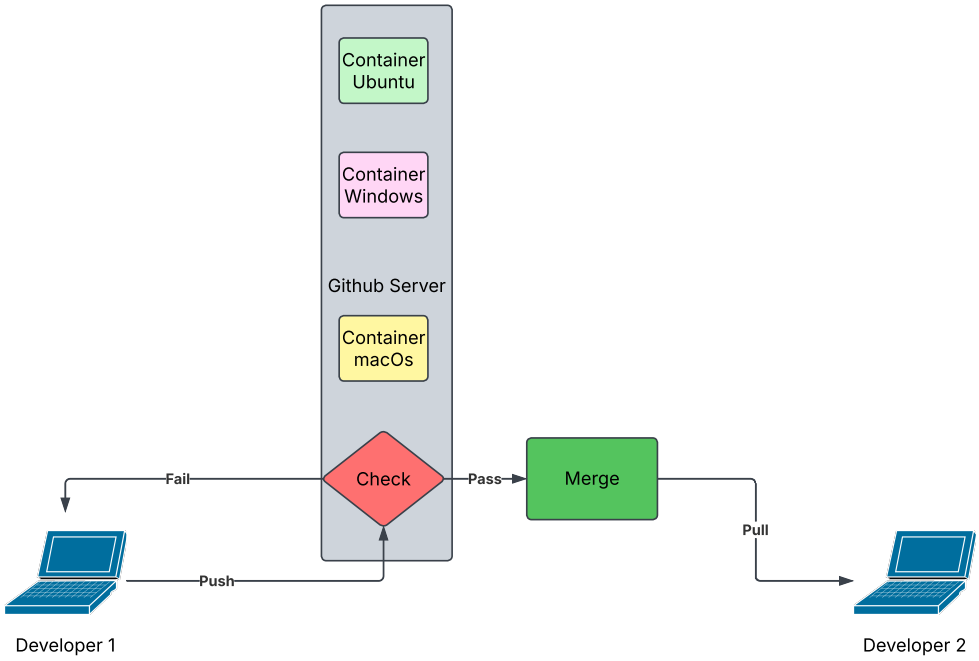


Figure 5: GitHub Actions Workflow Execution

Final Thoughts: CI/CD is a crucial component of modern software development. By leveraging GitHub Actions, `libmap` ensures stability, reliability, and high performance, allowing developers to focus on feature development rather than manual testing.

5.8 Final Thoughts on Testing and Performance

Summary of Testing Insights: The testing framework for `libmap` ensures correctness, stability, and efficiency through rigorous unit tests, integration tests, and performance profiling. The unit tests validate individual components such as insertion, retrieval, and removal, while integration tests assess overall consistency and behavior in real-world scenarios. The flamegraph analysis further identifies computational bottlenecks in large-scale operations.

5.8.1 Strengths of `libmap`

The testing and profiling results highlight several strengths of `libmap`:

- **Efficient Insertion and Retrieval:** The average case complexity for insertion and lookup remains O(1) with proper hashing.
- **Dynamic Resizing:** Automatic expansion and contraction ensure efficient memory utilization.

- **Customizability:** The library allows user-defined hashing, key comparison, and memory management functions.
- **Iterator Support:** Iterators enable efficient traversal of stored elements.

5.8.2 Identified Weaknesses and Bottlenecks

Despite its strengths, testing and profiling reveal some areas for improvement:

- **Frequent Resizing Costs:** The flamegraph analysis indicates that reallocation overhead is significant for large datasets.
- **Memory Allocation Overhead:** A considerable portion of execution time is spent in `malloc()` and `free()`.
- **Iteration Complexity:** Traversing all elements incurs additional overhead in large hashmaps.

5.8.3 Future Enhancements

Potential optimizations to enhance performance:

- **Optimized Resizing Strategy:** Switching to power-of-two bucket sizes may reduce resizing overhead.
- **Memory Pooling:** Preallocating memory for elements could reduce dynamic allocation overhead.
- **Improved Collision Resolution:** Exploring alternative techniques like open addressing may improve performance.

Closing Thoughts: The extensive testing and performance profiling of `libmap` demonstrate its robustness in handling dynamic key-value storage. While the library is efficient for most use cases, optimizations in resizing and memory management can further improve its performance. Future iterations may explore advanced hash functions, alternative data structures, and better memory allocation strategies to enhance overall efficiency.

5.8.4 Conclusion

CI/CD Ensures Stability: The integration of GitHub Actions into the development workflow of `libmap` has significantly improved code quality by ensuring that:

- Every commit and pull request undergoes rigorous testing.
- Memory issues and regressions are detected early using Valgrind.
- The build remains stable, preventing broken commits from being merged.
- Performance and correctness are continuously validated.

Key Takeaways:

- Automated Testing: Every push triggers unit and integration tests.
- Memory Safety Checks: Valgrind prevents memory leaks.
- Optimized Execution: Caching, parallel execution, and logging improve efficiency.

Final Thoughts: CI/CD is a crucial component of modern software development. By leveraging GitHub Actions, `libmap` ensures stability, reliability, and high performance, allowing developers to focus on feature development rather than manual testing.

6 Usage Examples

6.1 Counting Unique Elements in an Array

Introduction: One of the most common problems in programming is counting the number of unique elements in a dataset. Traditionally, solutions rely on brute-force checking ($O(n^2)$) or sorting followed by a single pass ($O(n \log n)$). However, these methods can be slow, especially for large datasets.

With `libmap`, we leverage a hashmap-based approach, achieving an optimal $O(n)$ complexity for insertion and lookup. This makes it significantly faster than sorting, and it scales well for large datasets.

This example demonstrates how `libmap` makes counting unique elements efficient, readable, and scalable.

Implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <map.h>

// Dummy key clone function
void* dummy_key_clone(void *key) {
    int *copy = malloc(sizeof(int));
    if (copy) *copy = *(int*)key;
    return copy;
}

// Dummy value clone function
void* dummy_value_clone(void *value) {
    int *new_value = malloc(sizeof(int));
    if (new_value) *new_value = *(int *)value;
    return new_value;
}

// Simple Modulus Hashing
uint64_t dummy_hash(void *key) {
    return (*(int *)key) % 1000003;  // Large prime for better distribution
}

// Dummy stringify function
char* dummy_stringify(void *key, void *value) {
    char *str = malloc(100 * sizeof(char));
    if (str) snprintf(str, 100, "(Key: %d, Value: %d)", *(int *)key, *(int
    ↪  *)value);
    return str;
}

// Dummy compare function
int32_t dummy_compare(void *key1, void *key2) {
    int a = *(int*)key1;
    int b = *(int*)key2;
    return (a > b) - (a < b);
}

// Dummy free functions
void dummy_free_key(void *key) { free(key); }
```

```c
void dummy_free_value(void *value) { free(value); }

// Function to count unique integers in an array
int count_unique(int *arr, size_t size) {
    map_t *map;
    map_error_t result;

    // Create hashmap
    result = map_create(&map, dummy_key_clone, dummy_value_clone, dummy_hash,
                        dummy_stringify, dummy_compare, dummy_free_key,
                        ↪  dummy_free_value);
    assert(result == MAP_OK && "Map creation failed");

    for (size_t i = 0; i < size; i++) {
        int *key = malloc(sizeof(int));
        assert(key && "Memory allocation failed");
        *key = arr[i];

        // Check if the key already exists
        int *existing_freq = NULL;
        result = map_get(map, key, (void **)&existing_freq);

        if (result == MAP_OK) {
            // Key exists, increment its frequency
            (*existing_freq)++;
            free(key);  // Free the unused key
        } else {
            // Key doesn't exist, insert it with frequency = 1
            int *freq = malloc(sizeof(int));
            assert(freq && "Memory allocation failed");
            *freq = 1;

            result = map_insert(map, key, freq);
            assert(result == MAP_OK && "Map insertion failed");
        }
    }

    // Get the number of unique elements
    int unique_count;
    map_get_size(map, &unique_count);

    // Destroy hashmap
    map_destroy(&map);

    return unique_count;
}

// Main function
int main(void) {
    int arr[] = {1,2,2,3,3,4,5,5,6,7,8,8,8,9,10,11,11,11,12};
    size_t size = sizeof(arr) / sizeof(arr[0]); // Convert bytes to elements

    printf("The given array is:\n[");
    for (size_t i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) printf(",");
    }
```

```
    printf("]\n");

    int unique_count = count_unique(arr, size);
    printf("The number of unique elements in the array is: %d\n", unique_count);

    return 0;
}
```

6.1.1 Breakdown and Explanation

- **Hashmap-Based Approach:** - Instead of iterating multiple times over the array or sorting, we use a hashmap for constant-time lookups. - This allows us to count unique elements in linear time $O(n)$.
- **Hashmap Initialization:** - The hashmap is created using `map_create()` with user-defined functions for key cloning, hashing, comparison, and memory management.
- **Efficient Insertion and Lookup:** - Each element is checked in `map_get()`. - If found, its count is increased. - Otherwise, it is inserted into the map.
- **Unique Count Retrieval:** - Once all elements are processed, `map_get_size()` returns the number of unique elements.
- **Memory Efficiency:** - Since the map dynamically resizes, it scales well with larger datasets. - Proper cleanup is ensured with `map_destroy()`.

6.1.2 Performance Comparison

Method	Time Complexity
Brute Force	$O(n^2)$
Sorting + Pass	$O(n \log n)$
`libmap` Hashmap	O(n)

Table 7: Performance Comparison for Counting Unique Elements

Expected Output:

```
The given array is:
[1,2,2,3,3,4,5,5,6,7,8,8,8,9,10,11,11,11,12]
The number of unique elements in the array is: 10
```

Key Takeaways:

- Speed: `libmap` makes the solution fast and scalable with an optimal `O(n)` complexity.
- Readability and Maintainability: Using a hashmap makes the logic clearer and cleaner than manual iteration.
- Real-World Impact: This approach can be extended to massive datasets like log analysis, frequency counting, and statistical modeling.

6.2 Other Applications of Hashmaps

Introduction: Hashmaps are a fundamental data structure widely used in software development due to their constant-time average complexity for insertions, lookups, and deletions. The `libmap` library provides a robust, customizable hashmap implementation in C, making it ideal for solving various real-world problems efficiently. Below are some common applications where `libmap` can be used.

Fast Lookup for the Two-Sum Problem

Use Case: Given an array of numbers and a target sum, find two numbers that add up to the target.

Why Hashmaps? Instead of a brute-force $O(n^2)$ approach, a hashmap can store numbers as keys and their indices as values, allowing for quick lookups in O(n).

How `libmap` Helps:

- Insert each number into the hashmap with its index as the value.
- For each number, check if `target - num` exists in the hashmap.
- If found, return the indices of the two numbers.

Anagram Checker

Use Case: Determine if two strings are anagrams (contain the same characters in different orders).

Why Hashmaps? Instead of sorting strings ($O(n \log n)$), a hashmap can store character frequencies in O(n).

How `libmap` Helps:

- Insert characters from the first string into a hashmap as keys, with their counts as values.
- Iterate through the second string, decrementing counts.
- If all values are zero, the strings are anagrams.

Key Takeaways:

- Hashmaps provide fast lookups and efficient storage for various problems.
- `libmap` enables C programmers to implement these solutions flexibly with custom key-value types.
- Using `libmap`, users can optimize performance-critical applications with frequency counting and efficient data retrieval.

7 Error Handling

7.1 Introduction to Error Handling

Why is error handling important? Robust error handling is essential in any library to prevent undefined behavior, crashes, or memory corruption. `libmap` is designed with structured error reporting to help users detect and handle failures safely.

How does `libmap` handle errors? Unlike some C libraries that rely on `assert()` or abrupt termination, `libmap` returns error codes of type `map_error_t`. This allows applications to gracefully handle errors instead of crashing.

User responsibility: Users of `libmap` should always check return values from functions and handle errors appropriately. Ignoring errors may lead to undefined behavior or memory leaks.

Common failure scenarios:

- **Invalid arguments:** Passing `NULL` pointers or incorrect data types.
- **Memory allocation failures:** When `malloc()` fails due to insufficient memory.
- **Hash collisions:** Large numbers of key collisions could affect performance.
- **Key not found errors:** Attempting to retrieve or remove a key that does not exist.

7.2 `map_error_t`: Error Codes

What is `map_error_t`? `map_error_t` is an enumeration that defines error codes returned by `libmap` functions. Instead of crashing on errors, functions return meaningful error codes, allowing the user to handle issues safely.

Error Code Reference Table: The table below summarizes all possible error codes in `libmap`:

Error Code	Description
`MAP_OK`	Operation completed successfully.
`MAP_ERR_NO_MEM`	Memory allocation failed due to insufficient system memory.
`MAP_ERR_NOT_FOUND`	The requested key was not found in the hashmap.
`MAP_ERR_INVALID_ARG`	A function received an invalid argument (e.g., NULL pointer).
`MAP_ERR_OVERFLOW`	The hashmap has grown too large to resize further.
`MAP_ERR_END_OF_MAP`	Iterator has reached the end of the hashmap.
`MAP_ERR_UNKNOWN`	An unexpected error occurred (should not normally happen).

Table 8: Error Codes in `libmap`

When do these errors occur?

- `MAP_ERR_NO_MEM`: Happens if `malloc()` fails during allocation.
- `MAP_ERR_NOT_FOUND`: Returned when attempting to retrieve or remove a non-existent key.
- `MAP_ERR_INVALID_ARG`: Occurs when passing a `NULL` pointer or invalid parameters to a function.
- `MAP_ERR_OVERFLOW`: Triggered when the hashmap reaches its maximum allowable size and cannot be resized further.
- `MAP_ERR_END_OF_MAP`: Used by iterators to signal that no more elements remain in the map.
- `MAP_ERR_UNKNOWN`: Indicates an unexpected internal error (should not normally occur).

7.3 How Functions Return Errors

Overview: All public functions in `libmap` return an error code of type `map_error_t`, allowing users to detect and handle issues gracefully.

7.3.1 Checking Return Values

To ensure robustness, users should always check the return value of `libmap` functions before proceeding. Ignoring error codes can lead to undefined behavior, crashes, or memory leaks.

```
map_error_t result = map_insert(my_map, key, value);
if (result != MAP_OK) {
    printf("Error: Failed to insert key (Error Code: %d)\n", result);
}
```

7.3.2 Functions That Return `map_error_t`

All `libmap` functions return `map_error_t` to indicate success or failure:

Function	Return Type	When to Check Error?
`map_create()`	`map_error_t`	Always check after creation.
`map_insert()`	`map_error_t`	Check to ensure insertion succeeded.
`map_get()`	`map_error_t`	Handle missing keys properly.
`map_remove()`	`map_error_t`	Verify removal success.
`map_destroy()`	`map_error_t`	Ensure no double-free occurs.

Table 9: Error Handling in libmap Functions

7.3.3 Best Practices for Error Handling

- **Always Check Return Values:** Never assume a function succeeded – always check its return value.
- **Use Meaningful Error Messages:** Instead of just printing `"Error"`, specify which function failed and why.
- **Handle Critical Errors Immediately:** For issues like `MAP_ERR_NO_MEM`, consider freeing other resources and aborting gracefully.
- **Log Errors for Debugging:** Use logs to track recurring issues and improve debugging.

7.3.4 Example: Handling Errors Gracefully

```
map_t *map;
map_error_t result = map_create(&map, key_clone, value_clone, hash, stringify,
↪  compare, free_key, free_value);
if (result != MAP_OK) {
    printf("Failed to create hashmap (Error Code: %d)\n", result);
    return 1;
}

char *key = "test";
int value = 42;
result = map_insert(map, key, &value);
if (result != MAP_OK) {
    printf("Failed to insert key: %s (Error Code: %d)\n", key, result);
}
```

7.3.5 Conclusion

Understanding how `libmap` functions return errors is crucial for writing stable and reliable code. By checking return values, handling failures properly, and logging errors, users can prevent crashes and ensure smooth hashmap operations.

7.4 Common Error Scenarios

Overview: Errors in `libmap` typically arise from invalid arguments, memory allocation failures, missing keys, and configuration mistakes. This section covers frequent error cases, their causes, and best practices for handling them.

7.4.1 Summary of Error Codes

The following table provides an overview of all error codes defined in `libmap`:

Error Code	Description
`MAP_OK`	Operation completed successfully.
`MAP_ERR_NO_MEM`	Memory allocation failed during an operation.
`MAP_ERR_NOT_FOUND`	Requested key was not found in the hashmap.
`MAP_ERR_INVALID_ARG`	One or more arguments passed to a function were invalid.
`MAP_ERR_OVERFLOW`	The table is too large to resize further.
`MAP_ERR_END_OF_MAP`	Iterator reached the end of the hashmap.
`MAP_ERR_UNKNOWN`	An unspecified error occurred.

Table 10: Summary of Error Codes in libmap

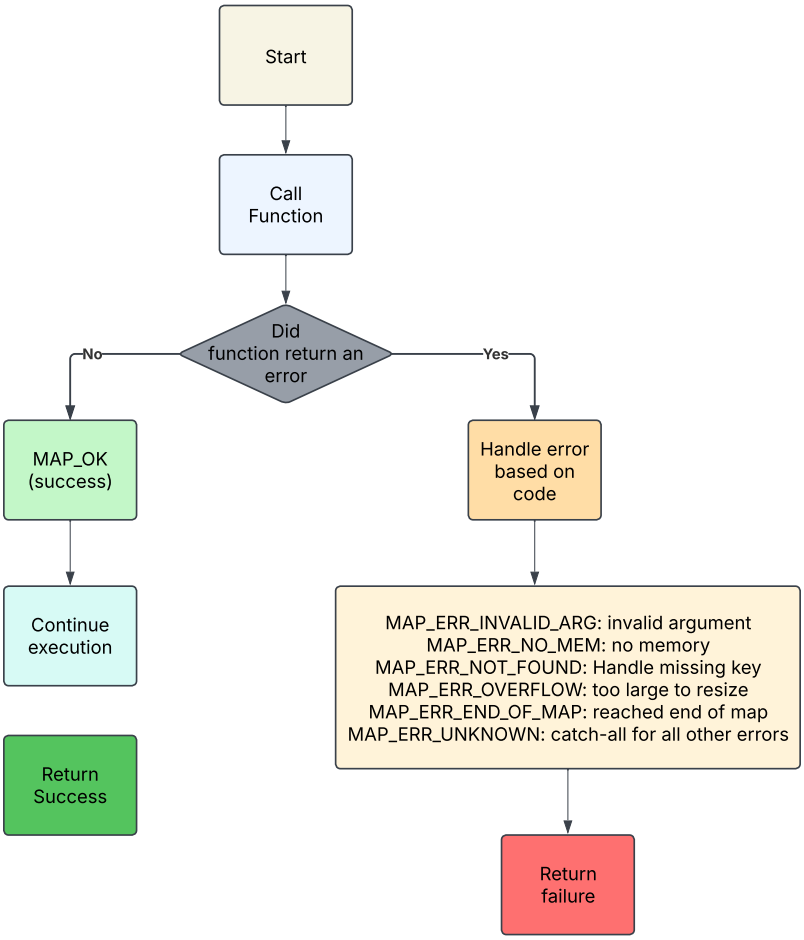


Figure 6: Error Handling Flowchart in `libmap`

7.4.2 1. Invalid Arguments (`MAP_ERR_INVALID_ARG`)

Cause: Occurs when passing `NULL` pointers or invalid parameters to a function.

```
// Invalid: Passing NULL as the map pointer
map_error_t result = map_insert(NULL, key, value);
if (result == MAP_ERR_INVALID_ARG) {
    printf("Error: Cannot insert into a NULL map\n");
}
```

Best Practices:

- Always validate function arguments before calling `libmap` functions.
- Ensure pointers are initialized properly.
- Use `assert()` to catch issues during development.

7.4.3 2. Memory Allocation Failure (`MAP_ERR_NO_MEM`)

Cause: Occurs when `malloc()` fails due to insufficient memory.

```
// Simulating memory failure
void *key = malloc(SIZE_MAX); // Likely to fail
if (key == NULL) {
```

```
    printf("Error: Out of memory!\n");
}
```

Best Practices:

- Always check if `malloc()` succeeds.
- Free memory when no longer needed.
- Use tools like Valgrind to detect memory leaks.

7.4.4 3. Key Not Found (`MAP_ERR_NOT_FOUND`)

Cause: Occurs when trying to retrieve or remove a key that does not exist.

```
void *value;
map_error_t result = map_get(my_map, "nonexistent", &value);
if (result == MAP_ERR_NOT_FOUND) {
    printf("Error: Key not found in hashmap\n");
}
```

Best Practices:

- Ensure the key exists before attempting to retrieve it.
- Provide meaningful error messages when a key is missing.
- Implement fallback behavior for missing keys.

7.4.5 4. Resizing Errors and Configuration Issues (`MAP_ERR_INVALID_ARG`)

Cause: Occurs when `map_configure()` is called with invalid parameters.

```
// Invalid: grow_factor must be greater than 1
map_error_t result = map_configure(my_map, 100.0, 0.5, 0.9);
if (result == MAP_ERR_INVALID_ARG) {
    printf("Error: Invalid parameters for map configuration\n");
}
```

Best Practices:

- Set `max_load_factor` and `min_load_factor` carefully.
- Ensure `grow_factor` is greater than 1.
- Validate input parameters before calling `map_configure()`.

7.4.6 5. Iterator Errors (`MAP_ERR_END_OF_MAP`)

Cause: Occurs when an iterator reaches the end of the hashmap.

```
map_iterator_t iter;
void *key, *value;
map_error_t result = map_iter_start(my_map, &iter);
while ((result = map_iter_next(my_map, &iter, &key, &value)) == MAP_OK) {
    printf("Key: %s, Value: %d\n", (char *)key, *(int *)value);
}
if (result == MAP_ERR_END_OF_MAP) {
    printf("End of map reached.\n");
}
```

Best Practices:

- Always check for `MAP_ERR_END_OF_MAP` when iterating.
- Do not modify the hashmap while iterating.

- Reset iterators before reuse.

7.4.7 6. Debugging Tips

Using Assertions for Debugging:

```
assert(map_insert(my_map, key, value) == MAP_OK);
```

Using Logging for Error Detection:

```
map_error_t result = map_insert(my_map, key, value);
if (result != MAP_OK) {
    printf("Error: Failed to insert key. Error code: %d\n", result);
}
```

Using Valgrind for Memory Errors: Run the program with:

```
valgrind --leak-check=full ./test_program
```

7.4.8 Conclusion

By understanding these common error scenarios and their proper handling, users can ensure robust and predictable behavior when using `libmap`. Proper error handling minimizes crashes, improves debugging, and enhances application stability.

7.5 Debugging Tips

Overview: Effective debugging of `libmap` requires both **preventative measures** (like assertions and compiler warnings) and **diagnostic tools** (like Valgrind). This section provides techniques to detect and resolve issues efficiently.

7.5.1 Using Assertions for Error Detection

Assertions help catch invalid states at runtime. The `assert()` function from `<assert.h>` ensures that assumptions hold true during execution.

```
#include <assert.h>

map_t *map = NULL;
assert(map_create(&map, key_clone, value_clone, hash, stringify, compare,
↪  free_key, free_value) == MAP_OK);
```

Use cases:

- Ensuring `map_create()` succeeds before proceeding.
- Checking if `map_get()` successfully retrieves an entry.
- Verifying pointer validity before dereferencing.

If an assertion fails, the program **terminates immediately**, helping catch errors early during development.

7.5.2 Enabling Compiler Warnings

Compilers provide powerful warnings that help detect potential issues.

GCC/Clang Warning Flags:

```
gcc -Wall -Wextra -Wpedantic -o test test_map.c -lmap
```

Use cases:

- Catching **unused variables, missing return values, and implicit conversions**.
- Detecting **mismatched function signatures** (e.g., passing the wrong argument type).

- Preventing **potential undefined behavior**.

Using these flags helps maintain code quality and prevents subtle bugs from slipping into production.

7.5.3 Using Valgrind for Memory Leak Detection

Since `libmap` uses dynamic memory allocation, **detecting memory leaks** is crucial.

To install Valgrind:

```
sudo apt install valgrind
```

Run it with:

```
valgrind --leak-check=full --show-leak-kinds=all ./test_map
```

Use cases:

- Identifying **memory leaks** in `map_create()`, `map_insert()`, or `map_remove()`.
- Checking for **use-after-free** errors.
- Validating that `map_destroy()` properly frees memory.

Example Valgrind output (bad case):

```
HEAP SUMMARY:
  definitely lost: 32 bytes in 2 blocks
  indirectly lost: 0 bytes in 0 blocks
```

If you see "definitely lost" memory, it means some allocated memory was **not freed**, indicating a potential memory leak.

7.5.4 Summary of Debugging Techniques

Technique	Purpose
`assert()`	Catches invalid states during execution.
Compiler Warnings (`-Wall`, `-Wextra`)	Detects potential mistakes at compile time.
Valgrind	Identifies memory leaks and use-after-free errors.

Table 11: Debugging Techniques in libmap

7.5.5 Final Thoughts

- **Use assertions** to detect invalid states early.
- **Enable compiler warnings** to catch issues at compile time.
- **Run Valgrind** to check for memory leaks and improper deallocations.

By following these debugging techniques, users can efficiently troubleshoot `libmap`, prevent crashes, and improve code reliability.

8 License

9 Contact

For any inquiries, feedback, or contributions regarding `libmap`, feel free to reach out.

Name: Aditya Singh
Email: aditherealone@gmail.com

If you encounter issues or have suggestions for improvements, you are encouraged to report them. Thank you for using `libmap`!