# Contents

# 1 Basic Test Results

```
1
2  Running presubmission script...
3
4
5  Opening tar file
6  Vector.c
7  HashMap.c
8  OK
9  Tar extracted O.K.
10 For your convenience, the MD5 checksum for your submission is f2754b0ee18adbe94fc47d59213687c7
11 Checking files...
12 OK
13 Making sure files are not empty...
14 OK
15 Compilation check...
16 Compiling...
17 OK
18 Compilation seems OK! Check if you got warnings!
19 Checking CodingStyle...
20 Checking file Vector.c...
21 Checking file HashMap.c...
22 Passed codingStyle check.
23
24 =====================
25  Public test cases
26 =====================
27
28 =====================
29 Running test...
30 [Pre-Submission] 3 ...
31 [Pre-Submission] 2 ...
32 [Pre-Submission] 1 ...
33 [Pre-Submission] Ignition ...
34 [Pre-Submission] Creating Vector of Ints ...
35 [Pre-Submission] Check vector size with assert ...
36 [Pre-Submission] Free Vector ...
37 [Pre-Submission] Creating Pairs of {Char : Int} ...
38 [Pre-Submission] Creating HashMap of {Char : Int} ...
39 [Pre-Submission] Inserting Pairs to HashMap ...
40 [Pre-Submission] Free Pairs ...
41 [Pre-Submission] Check hash-map size with assert ...
42 [Pre-Submission] Free hash map ...
43 [Pre-Submission] all tests ended.
44 OK
45 =====================
46
47 ****************************************
48 *                                      *
49 *          Passed all tests!!          *
50 *                                      *
51 ****************************************
```

# 2 HashMap.c

```c
#include "HashMap.h"


HashMap *HashMapAlloc(
        HashFunc hash_func, HashMapPairCpy pair_cpy,
        HashMapPairCmp pair_cmp, HashMapPairFree pair_free) {
    if (!hash_func || !pair_cpy || !pair_cmp || !pair_free) { // check if the functions are not null,
        return NULL;
    }
    HashMap *new_hash = calloc(1, sizeof(HashMap));
    if (!new_hash ) {
        return NULL;
    }
    new_hash->buckets = calloc(HASH_MAP_INITIAL_CAP, sizeof(Vector ));
    if (!new_hash->buckets) {
        return NULL;
    }
    new_hash->capacity = HASH_MAP_INITIAL_CAP;
    new_hash->hash_func = hash_func;
    new_hash->pair_cpy = pair_cpy;
    new_hash->pair_cmp = pair_cmp;
    new_hash->pair_free = pair_free;
    for (unsigned long i = 0; i < HASH_MAP_INITIAL_CAP; ++i) {
        new_hash->buckets[i] = VectorAlloc(new_hash->pair_cpy, new_hash->pair_cmp, new_hash->pair_free);
    }
    return new_hash;
}

void HashMapFree(HashMap **p_hash_map) {
    if (!*p_hash_map) {
        return;
    }
    HashMap *map = (*(p_hash_map));
    for (size_t k_i = 0; k_i < map->capacity; k_i++) {
        VectorFree(&map->buckets[k_i]);
    }
    free(map->buckets);
    map->buckets = NULL;
    free(*p_hash_map);
    *p_hash_map = NULL;
}

void CreatArrOfVector(HashMap *hash_map, Pair *arr[]) {
    /**
     * get the hash map and empty array and insert all the pairs that was in the hash map into the array
     */
    int k = 0;
    for (size_t j = 0; j < hash_map->capacity; ++j) {
        if (hash_map->buckets[j]) {
            for (size_t i = 0; i < hash_map->buckets[j]->size; ++i) {
                Pair *pair = VectorAt(hash_map->buckets[j], i);
                arr[k++] = hash_map->buckets[j]->elem_copy_func(
                        pair);

            }
        }
    }
}

```

3.1

```c
void CheckCapAndIncreaseIfNessr(HashMap *hash_map) {
    /**
     * check if we need to increase the capacity of the hash map depend on the load factor .
     * and if we need to change the capacity we rehash the hash map : we creat buffer - an array - and put all
     * the elements of the hash map into it .
     * after that we change the capacity and insert all the elements to th hash map
     */
    double loaf = HashMapGetLoadFactor(hash_map);
    if (loaf != -1 && loaf > HASH_MAP_MAX_LOAD_FACTOR) {
        size_t arr_size = hash_map->size;
        size_t num_of_vectors = hash_map->capacity;
        Pair** arr = malloc(hash_map->size*sizeof(Pair*));
        CreatArrOfVector(hash_map, arr); // i save all the pairs in an array for the rehashing
        size_t new_cap = hash_map->capacity * HASH_MAP_GROWTH_FACTOR;
        HashMapClear(hash_map);
        for (size_t i = 0; i < num_of_vectors ; ++i) {
            VectorFree(&hash_map->buckets[i]);
        }
        hash_map->capacity = new_cap;
        Vector **tmp = realloc(hash_map->buckets, new_cap * sizeof(Vector ));
        if (tmp) {
            hash_map->buckets = tmp;
        }
        for (size_t i = 0; i < new_cap ; ++i) {
            hash_map->buckets[i] = VectorAlloc(hash_map->pair_cpy, hash_map->pair_cmp, hash_map->pair_free);
        }
        for (size_t i = 0; i < arr_size; ++i) {
            HashMapInsert(hash_map, arr[i]);
        }
        for (size_t i = 0; i < arr_size; ++i) {
            hash_map->pair_free((void**)&arr[i]);
        }
        free(arr);
    }
}

void VectorFindByKey(Vector *vec, Pair *pair) {
    /**
     *     find the vector that the current key is in it and change the old pair to the new pair
     */
    for (size_t i = 0; i < vec->size; ++i) {
        Pair *old_pair = (Pair *) vec->data[i];
        if (pair->key_cmp(pair->key, old_pair->key)) {
            vec->elem_free_func((void**)&old_pair);
            vec->data[i] = pair;

        }
    }
}

void FindVectorAndChangePair(HashMap *hash_map, Pair *pair) {
    for (size_t i = 0; i < hash_map->capacity; ++i) {
        Vector * vec = hash_map->buckets[i];
        if (vec && vec->size) { // the currnet bucket is with vector of at list one pair
            VectorFindByKey(vec, pair);
        }
    }
}

int HashMapInsert(HashMap *hash_map, Pair *pair) {
    if (hash_map && pair) {
        Pair *new_pair = hash_map->pair_cpy(pair);
        if (HashMapContainsKey(hash_map,
                               new_pair->key)) {
            /** if the key of the new pair is in the hash map
            * we need to find the vector that the new key is in it
            * and change the old pair to the new pair
            * we dont need to check the capacity because the size still the same
```

```
128             */
129             FindVectorAndChangePair(hash_map, new_pair);
130             return 1;
131         }
132         size_t hash_ind = hash_map->hash_func(new_pair->key) & (hash_map->capacity - 1);
133         if (!hash_map->buckets[hash_ind]->size ) {
134             Vector * vec = hash_map->buckets[hash_ind];
135             VectorPushBack(vec, new_pair);
136             hash_map->pair_free((void**)(&new_pair));
137             hash_map->buckets[hash_ind] = vec;
138             hash_map->size++;
139             CheckCapAndIncreaseIfNessr(hash_map);
140             return 1;
141         } else { // if there is at list one pair in the vector
142             Vector *cur_vec = hash_map->buckets[hash_ind];
143             VectorPushBack(cur_vec, new_pair);
144             hash_map->pair_free((void**)(&new_pair));
145             hash_map->size++;
146             CheckCapAndIncreaseIfNessr(hash_map);
147             return 1;
148         }
149
150
151     }
152
153     return 0;
154 }
155
156 int HashMapContainsKey(HashMap *hash_map, KeyT key) {
157     if (!hash_map) {
158         return 0;
159     }
160
161     for (size_t i = 0; i < hash_map->capacity  ; ++i) {
162         if (hash_map->buckets[i]) {
163             Vector *vec = hash_map->buckets[i];
164             for (size_t j = 0; j < vec->size; ++j) {
165                 Pair *cur_pair = (Pair *) vec->data[j];
166                 KeyT cur_key = cur_pair->key;
167                 if (cur_pair->key_cmp(cur_key, key)) {
168                     return 1;
169                 }
170             }
171         }
172     }
173     return 0;
174
175 }
176
177 int HashMapContainsValue(HashMap *hash_map, ValueT value) {
178     if (!hash_map) {
179         return 0;
180     }
181     for (size_t i = 0; i < hash_map->capacity; ++i) {
182         if (hash_map->buckets[i]) { // if there is vector in the the bucket (it means that there is at list one pair
183             Vector *vec = hash_map->buckets[i];
184             for (size_t j = 0; j < vec->size; ++j) {
185                 Pair *cur_pair = (Pair *) vec->data[j];
186                 ValueT cur_val = cur_pair->value;
187                 if (cur_pair->value_cmp(cur_val, value)) {
188                     return 1;
189                 }
190             }
191         }
192     }
193     return 0;
194 }
195
```

```
196 ValueT HashMapAt(HashMap *hash_map, KeyT key) {
197     if (!hash_map) {
198         return NULL;
199     }
200     for (size_t i = 0; i < hash_map->capacity; ++i) {
201         if (hash_map->buckets[i]) {
202             Vector *vec = hash_map->buckets[i];
203             for (size_t j = 0; j < vec->size; ++j) {
204                 Pair *pair = vec->data[j];
205                 if (pair->key_cmp(pair->key, key)) {
206                     return pair->value;
207                 }
208             }
209         }
210     }
211     return NULL;
212 }
213
214
215 void CheckCapAndDecreaseIfNessr(HashMap *hash_map) {
216     /** check if the current min load factor is lower then the MIN_LOAD_FACTOR 6.1 if it is the function send the
217     * hash map and an empty array and fil in the array with all the pairs that was in the hash map.
218     * after that it realloc the buckets of the hash map , clear the hash map ,and rehash the pairs into the the clear
219     * hash map.
220      */
221     double loaf = HashMapGetLoadFactor(hash_map);
222     if (loaf != -1 && loaf < HASH_MAP_MIN_LOAD_FACTOR) {
223         size_t arr_size = hash_map->size;
224         Pair** arr = malloc(hash_map->size*sizeof(Pair*));
225         CreatArrOfVector(hash_map, arr);          // save all the pairs in an array for the rehashing.
226         size_t new_cap = hash_map->capacity / HASH_MAP_GROWTH_FACTOR;
227         for (size_t i = 0; i < hash_map->capacity; ++i) {
228             VectorFree(&hash_map->buckets[i]);
229         }
230         HashMapClear(hash_map);  // clear the hash map before insert  the pairs into it
231         hash_map->capacity = new_cap;
232         Vector **tmp = realloc(hash_map->buckets, new_cap * sizeof(Vector *));
233         if (tmp) {
234             hash_map->buckets = tmp;
235         }
236         for (size_t i = 0; i < new_cap ; ++i) { // insert vector to all the buckets
237             hash_map->buckets[i] = VectorAlloc(hash_map->pair_cpy, hash_map->pair_cmp, hash_map->pair_free);
238         }
239         for (size_t i = 0; i < arr_size; ++i) { // insert the pairs into the hashmap
240             HashMapInsert(hash_map, arr[i]);
241         }
242         for (size_t i = 0; i < arr_size; ++i) {
243             hash_map->pair_free((void**)&arr[i]);
244         }
245         free(arr);
246     }
247 }
248
249 int HashMapErase(HashMap *hash_map, KeyT key) {
250     if (!hash_map) {
251         return 0;
252     }
253     if (!HashMapContainsKey(hash_map, key)) {      6.2
254         return 0;
255     }
256     for (size_t i = 0; i < hash_map->capacity; ++i) {
257         Vector *vec = hash_map->buckets[i];
258         if (vec) {
259             for (size_t j = 0; j < vec->size; ++j) {
260                 Pair *pair = vec->data[j];
261                 if (pair->key_cmp(key, pair->key)) {
262                     int ind = VectorFind(vec, pair);
263                     if (ind != -1) {
```

```
                        VectorErase(vec, ind);
                        hash_map->size--;
                        if (!vec->data[j]) {// the bucket is now empty and dont contain any pairs
                            CheckCapAndDecreaseIfNessr(hash_map);
                            return 1;
                        }
                    }
                }
            }
        }
    }
    return 0;
}

double HashMapGetLoadFactor(HashMap *hash_map) {
    if (hash_map) {
        double size = hash_map->size;
        double cap = hash_map->capacity;
        return size / cap;
    }
    return -1;
}

void ChooseCapOfClearMap(HashMap *hash_map, size_t start_size, size_t start_cap) {
    /** this function choose the final capacity of the hash map .
   * there are three options to the final capacity depend in what the start size and the start capacity
   * was in the first
     *
   */
    if (start_size == 1 && start_cap == HASH_MAP_INITIAL_CAP) {
        hash_map->capacity = HASH_MAP_INITIAL_CAP/HASH_MAP_GROWTH_FACTOR;
    } else if (start_size == 2 && start_cap == HASH_MAP_INITIAL_CAP) {
        hash_map->capacity = HASH_MAP_INITIAL_CAP / HASH_MAP_GROWTH_FACTOR / HASH_MAP_GROWTH_FACTOR;
    } else {
        hash_map->capacity = HASH_MAP_INITIAL_CAP/HASH_MAP_GROWTH_FACTOR/HASH_MAP_GROWTH_FACTOR/HASH_MAP_GROWTH_FACTOR;
    }
}

void HashMapClear(HashMap *hash_map) {
    if (!hash_map) {
        return;
    }
    size_t start_size = hash_map->size;
    size_t start_cap = hash_map->capacity;
    for (size_t i = 0; i < hash_map->capacity ; ++i) {
        if (hash_map->buckets[i]) {
            Vector *vec = hash_map->buckets[i];
            VectorClear(vec);
        }
    }
    ChooseCapOfClearMap(hash_map, start_size, start_cap);
    for (size_t i = hash_map->capacity ; i < start_cap; ++i) {
        VectorFree(&hash_map->buckets[i]);
    }
    hash_map->size = 0;
}
```

# 3 Vector.c

```c
#include "Vector.h"
#include <stdlib.h>


Vector *VectorAlloc(VectorElemCpy elem_copy_func, VectorElemCmp elem_cmp_func, VectorElemFree elem_free_func) {
    if(!elem_copy_func || !elem_cmp_func || !elem_free_func){ // check uf the functions are not Null
        return  NULL;
    }
    Vector *new_vector = calloc(1, sizeof(Vector));
    if (!new_vector) {
        return NULL;
    }
    new_vector->data = calloc(VECTOR_INITIAL_CAP, sizeof(void *));
    if (!new_vector->data) {
        return NULL;
    }
    new_vector->capacity = VECTOR_INITIAL_CAP;
    new_vector->size = 0;
    new_vector->elem_copy_func = elem_copy_func;
    new_vector->elem_cmp_func = elem_cmp_func;
    new_vector->elem_free_func = elem_free_func;
    return new_vector;
}

void VectorFree(Vector **p_vector) {
    if (!*p_vector) {
        return;
    }
    for (size_t k_i = 0; k_i < (*p_vector)->size; k_i++) {
        (*p_vector)->elem_free_func(&(*p_vector)->data[k_i]);
        (*p_vector)->data[k_i] = NULL;
    }
    free((*p_vector)->data);
    (*p_vector)->data = NULL;
    free(*p_vector);
    (*p_vector) = NULL;


}
void *VectorAt(Vector *vector, size_t ind) {
    if (!vector || ind >= vector->size || !vector->data ) {
        return NULL;
    }
    int *a = vector->data[ind];
    return a;
}

int VectorFind(Vector *vector, void *value) {
    if (!vector || !value) {
        return -1; // need to think about it
    }
    for (size_t k_i = 0; k_i < vector->size; k_i++) {
        void const *a = vector->data[k_i];
        void const *b = value;
        if (vector->elem_cmp_func(a, b)) {
            return k_i;
        }
    }
    return -1;
```

```
}

static void *CheckError(void *ptr, void *return_value) {
    /**
     * check error of the parameters that the function get
     */
    if (!ptr) {
        return return_value;
    }
    return NULL;
}

int VectorIncreaseCapIfNessry(Vector *vector) {
    /**
     * this function check if we need to increase the capacity of the vector depend on the load factor
     * and if it was it change the capacity
     */
    if (VectorGetLoadFactor(vector) >= VECTOR_MAX_LOAD_FACTOR) {
        void **tmp = realloc(vector->data, vector->capacity * VECTOR_GROWTH_FACTOR * sizeof(void *));
        CheckError(tmp, 0);
        vector->data = tmp;
        vector->capacity *= VECTOR_GROWTH_FACTOR;

    }
    return 1;
}

int VectorDecreaseCapIfNessry(Vector *vector) {
    /**
     * this function check if we need to decrease the capacity of the vector depend on the load factor
     * and if it was it change the capacity
     */
    if (VectorGetLoadFactor(vector) != 0 && VectorGetLoadFactor(vector) < VECTOR_MIN_LOAD_FACTOR) {
        void **tmp = realloc(vector->data, (vector->capacity / VECTOR_GROWTH_FACTOR) * sizeof(void *));
        CheckError(tmp, 0);
        vector->data = tmp;
        vector->capacity /= VECTOR_GROWTH_FACTOR;
    }
    return 1;
}


int VectorPushBack(Vector *vector, void *value) {
    CheckError(vector, 0);
    CheckError(value, 0);
    void *cpy_val = vector->elem_copy_func(value);
    vector->data[vector->size++] = cpy_val;
    VectorIncreaseCapIfNessry(vector); // increase the capacity of the vector and change the vector size and the vector cap
    return 1;

}

double VectorGetLoadFactor(Vector *vector) {
    if (!vector) {
        return -1;
    }
    return (double) vector->size / vector->capacity;
}

int VectorErase(Vector *vector, size_t ind) {
    CheckError(vector, 0);
    if (ind >= vector->size) {
        return 0;
    }

    vector->elem_free_func(&vector->data[ind]);
    for (size_t k_i = ind; k_i < vector->size; k_i++) {

```

```
128             if (k_i != vector->size) { // if the index is not the last value in the vector
129                 vector->data[k_i] = vector->data[k_i + 1];
130             } else {
131                 vector->data[k_i] = NULL;
132             }
133         }
134         vector->size--;
135         VectorDecreaseCapIfNessry(vector);
136         return 1;
137 
138     }
139     void ChooseCapOfClearVec(Vector *vec, size_t start_size, size_t start_cap) {
140         /** this function choose the final capacity of the vector .
141          * there are three options to the final capacity depend in what the start size and the start capacity
142          * was in the first
143          */
144         if (start_size == 1 && start_cap == VECTOR_INITIAL_CAP) {
145             vec->capacity = VECTOR_INITIAL_CAP/VECTOR_GROWTH_FACTOR;
146         } else if (start_size == 2 && start_cap == VECTOR_INITIAL_CAP) {
147             vec->capacity = VECTOR_INITIAL_CAP / VECTOR_GROWTH_FACTOR / VECTOR_GROWTH_FACTOR;
148         } else {
149             vec->capacity = VECTOR_INITIAL_CAP/VECTOR_GROWTH_FACTOR/VECTOR_GROWTH_FACTOR/ \
150             VECTOR_GROWTH_FACTOR/VECTOR_GROWTH_FACTOR;
151         }
152     }
153     void VectorClear(Vector *vector) {
154         if (!vector) {
155             return;
156         }
157         size_t start_size = vector->size;
158         size_t start_cap = vector->capacity;
159         int i = 0;
160         while (vector->size){
161             vector->elem_free_func(&vector->data[i++]);
162             vector->size--;
163         }
164         ChooseCapOfClearVec(vector,start_size,start_cap);
165 
166     }
```

# Index of comments