Output :

```
Number of buckets in the table: 5

Number of elements in the table: 1

Table's load factor: 0.200000

Table's max load factor: 0.500000

Table's resize multiplier: 2.000000


Number of buckets in the table: 5

Number of elements in the table: 2

Table's load factor: 0.400000

Table's max load factor: 0.500000

Table's resize multiplier: 2.000000


Number of buckets in the table: 5

Number of elements in the table: 3

Table's load factor: 0.600000

Table's max load factor: 0.500000

Table's resize multiplier: 2.000000


Number of buckets in the table: 10

Number of elements in the table: 4

Table's load factor: 0.400000

Table's max load factor: 0.500000

Table's resize multiplier: 2.000000


Number of buckets in the table: 10

Number of elements in the table: 5

Table's load factor: 0.500000

Table's max load factor: 0.500000
```

```
Table's resize multiplier: 2.000000


Number of buckets in the table: 10

Number of elements in the table: 6

Table's load factor: 0.600000

Table's max load factor: 0.500000

Table's resize multiplier: 2.000000
```

Demo.c

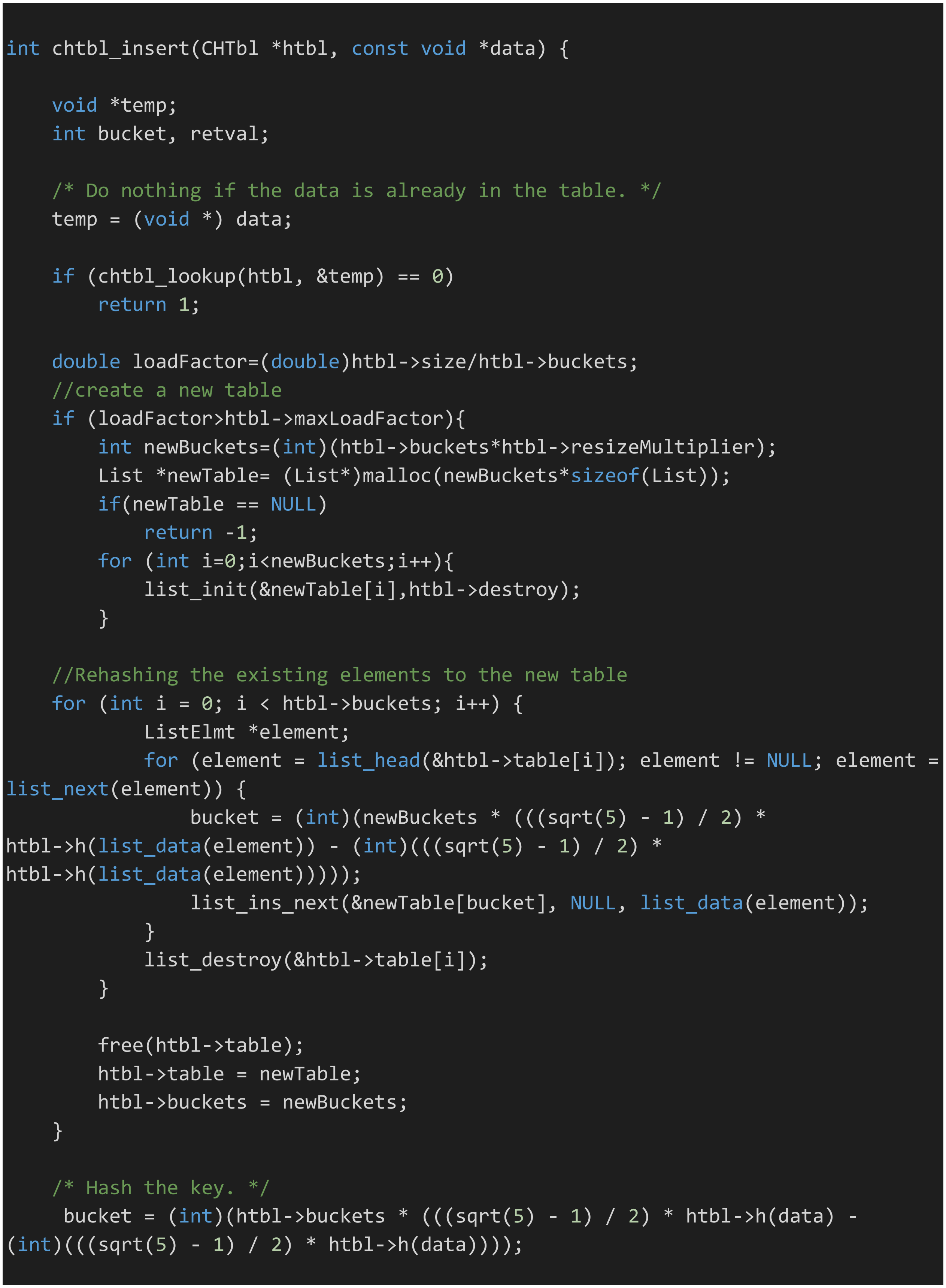
```c
#include "chtbl.h"
#include <stdio.h>

// Simple hash function: modulus by number of buckets
int hash(const void *key) {
    return (*(int *)key);
}

// Matching function
int match(const void *key1, const void *key2) {
    return (*(int *)key1 == *(int *)key2);
}

int main() {
    CHTbl table;

    chtbl_init(&table, 5, hash, match, free, 0.5, 2);  // initial size: 5,
maxLoadFactor: 0.5, resizeMultiplier: 2

    int i = 0;
    while (1) {
        int *data = (int*) malloc(sizeof(int));
        if (!data) {
            printf("Memory allocation error\n");
            return -1;
        }
        *data = i;

        if (chtbl_insert(&table, data) != 0) {
            free(data);
        }
        printf("Number of buckets in the table: %d\n", table.buckets);
        printf("Number of elements in the table: %d\n", table.size);
        printf("Table's load factor: %f\n", (double)table.size / table.buckets);
        printf("Table's max load factor: %f\n", table.maxLoadFactor);
        printf("Table's resize multiplier: %f\n\n", table.resizeMultiplier);

        if (table.size > 50) {  // arbitrary stopping condition
            break;
        }
        i++;
    }

    chtbl_destroy(&table);
    return 0;
}
```

Chtbl.h

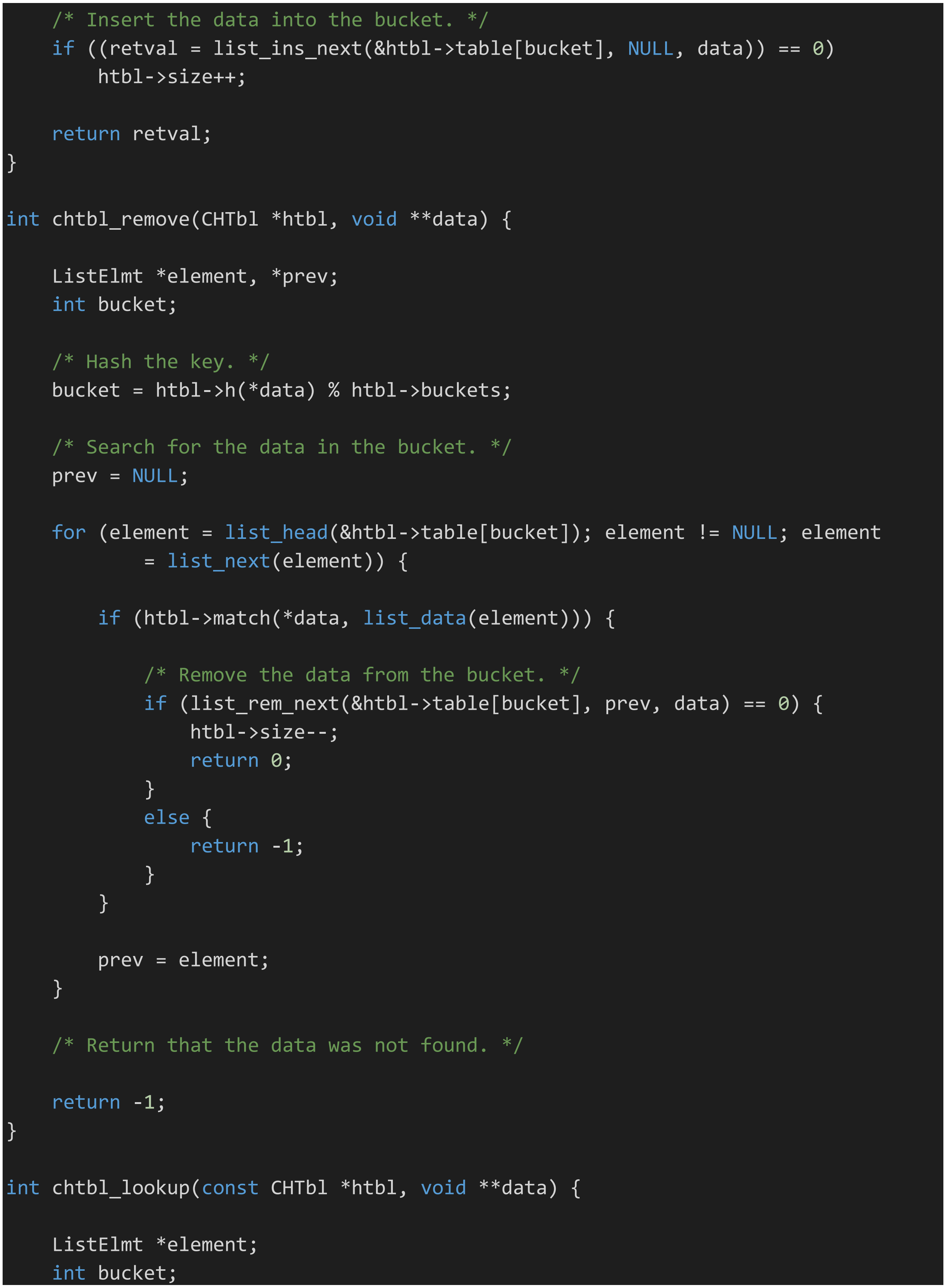
```
/*
 * chtbl.h
 */
#ifndef CHTBL_H
#define CHTBL_H

#include <stdlib.h>

#include "list.h"

/* Define a structure for chained hash tables. */
typedef struct CHTbl_ {

    int buckets;

    int (*h)(const void *key);
    int (*match)(const void *key1, const void *key2);
    void (*destroy)(void *data);

    int size;
    List *table;
    /* the maximum load factor the hash table should be
    allowed to reach before being auto-resized*/
    double maxLoadFactor;
    /*the number of buckets should be multiplied when a resize occurs*/
    double resizeMultiplier;
} CHTbl;

/* Public Interface */
//Modified chtbl_init prototype.
int chtbl_init(CHTbl *htbl,
            int buckets,
            int (*h)(const void *key),
            int (*match)(const void *key1, const void *key2),
            void (*destroy)(void *data),
            double maxLoadFactor,
            double resizeMultiplier);

void chtbl_destroy(CHTbl *htbl);

int chtbl_insert(CHTbl *htbl, const void *data);

int chtbl_remove(CHTbl *htbl, void **data);

int chtbl_lookup(const CHTbl *htbl, void **data);

#define chtbl_size(htbl) ((htbl)->size)

#endif
```

Chtbl.c

```
/*
 * chtbl.c
 */
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "list.h"
#include "chtbl.h"

int chtbl_init(CHTbl *htbl,
                int buckets,
                int (*h)(const void *key),
                int (*match)(const void *key1, const void *key2),
                void (*destroy)(void *data),
                double maxLoadFactor,
                double resizeMultiplier) {
    if (resizeMultiplier<=1.0) //resizeMultiplier must bigger than 1
        return -1;
    int i;

    /* Allocate space for the hash table. */
    if ((htbl->table = (List *) malloc(buckets * sizeof(List))) == NULL)
        return -1;

    /* Initialize the buckets. */
    htbl->buckets = buckets;

    for (i = 0; i < htbl->buckets; i++)
        list_init(&htbl->table[i], destroy);

    /* Encapsulate the functions. */
    htbl->h = h;
    htbl->match = match;
    htbl->destroy = destroy;

    /* Initialize the number of elements in the table. */
    htbl->size = 0;

    /*The maxLoadFactor and resizeMultiplier values should be stored in new fields in the CHTbl struct*/
    htbl->maxLoadFactor=maxLoadFactor;
    htbl->resizeMultiplier=resizeMultiplier;

    return 0;
}

void chtbl_destroy(CHTbl *htbl) {

    int i;

    /* Destroy each bucket. */
    for (i = 0; i < htbl->buckets; i++) {
        list_destroy(&htbl->table[i]);
    }

    /* Free the storage allocated for the hash table. */
    free(htbl->table);

    /* No operations are allowed now, but clear the structure as a
     * precaution. */
    memset(htbl, 0, sizeof(CHTbl));
}
```



```
int chtbl_insert(CHTbl *htbl, const void *data) {

    void *temp;
    int bucket, retval;

    /* Do nothing if the data is already in the table. */
    temp = (void *) data;

    if (chtbl_lookup(htbl, &temp) == 0)
        return 1;

    double loadFactor=(double)htbl->size/htbl->buckets;
    //create a new table
    if (loadFactor>htbl->maxLoadFactor){
        int newBuckets=(int)(htbl->buckets*htbl->resizeMultiplier);
        List *newTable= (List*)malloc(newBuckets*sizeof(List));
        if(newTable == NULL)
            return -1;
        for (int i=0;i<newBuckets;i++){
            list_init(&newTable[i],htbl->destroy);
        }

    //Rehashing the existing elements to the new table
    for (int i = 0; i < htbl->buckets; i++) {
            ListElmt *element;
            for (element = list_head(&htbl->table[i]); element != NULL; element =
list_next(element)) {
                bucket = (int)(newBuckets * (((sqrt(5) - 1) / 2) *
htbl->h(list_data(element)) - (int)(((sqrt(5) - 1) / 2) *
htbl->h(list_data(element)))));
                list_ins_next(&newTable[bucket], NULL, list_data(element));
            }
            list_destroy(&htbl->table[i]);
        }

        free(htbl->table);
        htbl->table = newTable;
        htbl->buckets = newBuckets;
    }

    /* Hash the key. */
     bucket = (int)(htbl->buckets * (((sqrt(5) - 1) / 2) * htbl->h(data) -
(int)(((sqrt(5) - 1) / 2) * htbl->h(data))));
```



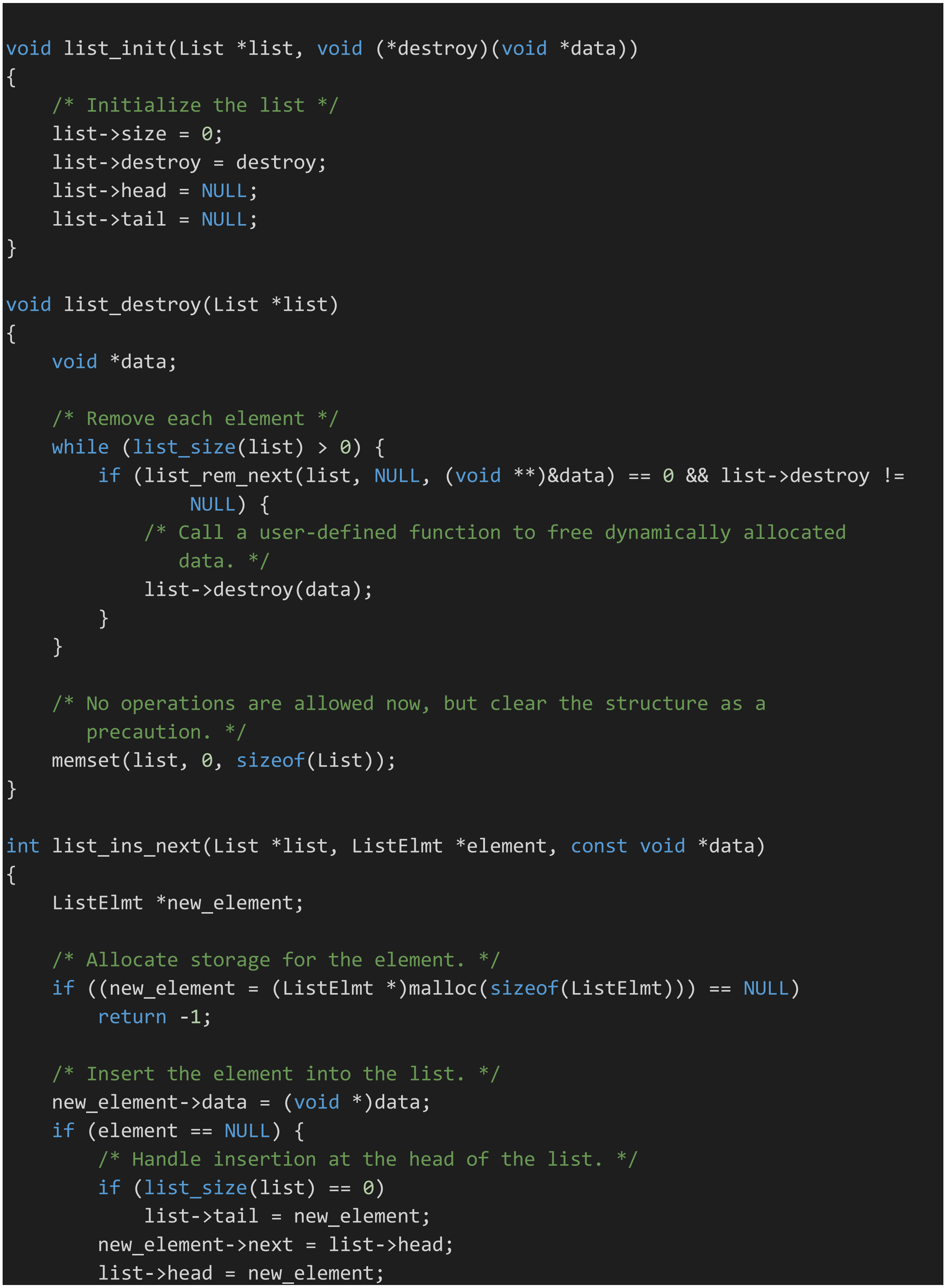
```c
    /* Insert the data into the bucket. */
    if ((retval = list_ins_next(&htbl->table[bucket], NULL, data)) == 0)
        htbl->size++;

    return retval;
}

int chtbl_remove(CHTbl *htbl, void **data) {

    ListElmt *element, *prev;
    int bucket;

    /* Hash the key. */
    bucket = htbl->h(*data) % htbl->buckets;

    /* Search for the data in the bucket. */
    prev = NULL;

    for (element = list_head(&htbl->table[bucket]); element != NULL; element
            = list_next(element)) {

        if (htbl->match(*data, list_data(element))) {

            /* Remove the data from the bucket. */
            if (list_rem_next(&htbl->table[bucket], prev, data) == 0) {
                htbl->size--;
                return 0;
            }
            else {
                return -1;
            }
        }

        prev = element;
    }

    /* Return that the data was not found. */

    return -1;
}

int chtbl_lookup(const CHTbl *htbl, void **data) {

    ListElmt *element;
    int bucket;
    /* Hash the key. */
    bucket = htbl->h(*data) % htbl->buckets;

    /* Search for the data in the bucket. */
    for (element = list_head(&htbl->table[bucket]); element != NULL; element
            = list_next(element)) {

        if (htbl->match(*data, list_data(element))) {

            /* Pass back the data from the table. */
            *data = list_data(element);
            return 0;
        }
    }

    /* Return that the data was not found. */

    return -1;
}
```

list.h

```
/*
 * list.h
 */
#ifndef LIST_H
#define LIST_H

#include <stdlib.h>

/*
 * Singly-linked list element
 */
typedef struct ListElmt_
{
    void               *data;
    struct ListElmt_   *next;
} ListElmt;

/*
 * Singly-linked list
 */
typedef struct List_
{
    int                size;

    int                (*match)(const void *key1, const void *key2);
    void               (*destroy)(void *data);

    ListElmt           *head;
    ListElmt           *tail;
} List;

/*
 * Public interface
 */
void list_init(List *list, void (*destroy)(void *data));

void list_destroy(List *list);

int list_ins_next(List *list, ListElmt *element, const void *data);

int list_rem_next(List *list, ListElmt *element, void **data);

#define list_size(list) ((list)->size)

#define list_head(list) ((list)->head)

#define list_tail(list) ((list)->tail)

#define list_is_head(list, element) ((element) == (list)->head ? 1 : 0)

#define list_is_tail(element) ((element)->next == NULL ? 1 : 0)

#define list_data(element) ((element)->data)

#define list_next(element) ((element)->next)

#endif
```

List.c

```
/*
 * list.c
 */
#include <stdlib.h>
#include <string.h>

#include "list.h"
```



```c
void list_init(List *list, void (*destroy)(void *data))
{
    /* Initialize the list */
    list->size = 0;
    list->destroy = destroy;
    list->head = NULL;
    list->tail = NULL;
}

void list_destroy(List *list)
{
    void *data;

    /* Remove each element */
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 && list->destroy != NULL) {
            /* Call a user-defined function to free dynamically allocated data. */
            list->destroy(data);
        }
    }

    /* No operations are allowed now, but clear the structure as a precaution. */
    memset(list, 0, sizeof(List));
}

int list_ins_next(List *list, ListElmt *element, const void *data)
{
    ListElmt *new_element;

    /* Allocate storage for the element. */
    if ((new_element = (ListElmt *)malloc(sizeof(ListElmt))) == NULL)
        return -1;

    /* Insert the element into the list. */
    new_element->data = (void *)data;
    if (element == NULL) {
        /* Handle insertion at the head of the list. */
        if (list_size(list) == 0)
            list->tail = new_element;
        new_element->next = list->head;
        list->head = new_element;
    }
    else {
        /* Handle insertion somewhere other than at the head. */
        if (element->next == NULL)
            list->tail = new_element;
        new_element->next = element->next;
        element->next = new_element;
    }

    /*  Adjust the size of the list to account for the inserted element. */
    list->size++;

    return 0;
}

int list_rem_next(List *list, ListElmt *element, void **data)
{
    ListElmt *old_element;

    /* Do not allow removal from an empty list. */
    if (list_size(list) == 0)
        return -1;

    /* Remove the element from the list. */
    if (element == NULL) {
        /* Handle removal from the head of the list. */
        *data = list->head->data;
        old_element = list->head;
        list->head = list->head->next;

        if (list_size(list) == 1)
            list->tail = NULL;
    }
    else {
        /* Handle removal from somewhere other than the head. */
        if (element->next == NULL)
            return -1;

        *data = element->next->data;
        old_element = element->next;
        element->next = element->next->next;

        if (element->next == NULL)
            list->tail = element;
    }
    /* Free the storage allocated by the abstract data type. */
    free(old_element);

    /* Adjust the size of the list to account for the removed element. */
    list->size--;

    return 0;
}
```

1. **What is the Big-O execution performance of an insert now that auto-resizing can take place?**
   The amortized time of insert is O(1).
   In the worst case, the time complexity of insert operation is O(N)
2. **Why do you think you were required to change chtbl_insert to use the multiplication method instead of the division method to map hash codes to buckets?**
   When using the division method, the number of buckets is often chosen to be prime to reduce the likelihood of collision. On the other hand, the multiplication method works with a real number between 0 and 1 and the size of the table doesn't need to be prime.