**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**

#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))));

    /\* 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"

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)

1. **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.