Homework #5

a) (5 points) Modify the chained hash table from the book (*chtbl.h* and *chtbl.c*) so that it auto-grows when its load factor exceeds a given value. (Auto-growing means increasing the number of buckets in the hash table then rehashing the existing elements.)

Begin by modifying the chtb1_init function from the book to have the following prototype:

```
Int chtb1_init (CHTb1 *htbl,
                int buckets,
                int (*h) (const void *key),
                int (*match) (const void key1, const void *key2),
                void (*destroy) (void *data),
                double maxLoadFactor,
                double resizeMultiplier);
```

The *htbl*, *buckets*, *h*, *match*, and *destroy* parameters have the same meaning as defined in the book. The *maxLoadFactor* parameter is the maximum load factor the hash table should be allowed to reach before getting auto-resized. The *resizeMultiplier* parameter is the amount by which the number of buckets should be multiplied when a resize occurs (this value must be greater than 1). The *maxLoadFactor* and *resizeMultiplier* values should be stored in new fields in *CHTb1* struct.

Modify any other parts of the *CHTb1* as needed. Note that removing items from the hash table should not cause the number of buckets in the table to shrink.

Next, modify the code in the *chtb1_insert* function as follows:

1. Resize the hash table when the load factor exceeds the maximum load factor. The new size of the hash table should be the old size times the *resizeMultiplier*. All elements currently in the hash table must be reshaped and placed into new buckets.

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

    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;
    htbl->maxLoadFactor = maxLoadFactor;
    htbl->resizeMultiplier = resizeMultiplier;

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

    return 0;
}
```

```c
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));
}
```

```c
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;

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

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

        double loadFactor = (double)htbl->size / htbl->buckets;

        // When calculated load factor exceeds maximum load factor
        if (loadFactor > htbl->maxLoadFactor) {
            // New hash size
            int newBuckets = (int)(htbl->buckets * htbl->resizeMultiplier);

            // Create new table
            List *newTable = (List *)malloc(newBuckets * sizeof(List));
            if (newTable == NULL)
                return -1;

            // init new buckets
            for (int i = 0; i < newBuckets; i++) {
                list_init(&newTable[i], htbl->destroy);
            }

            // Rehash existing elements
            ListElmt *element;
            void *tempData;
            for (int i = 0; i < htbl->buckets; i++) {
                element = list_head(&htbl->table[i]);
                while (element != NULL) {
                    tempData = list_data(element);
                    int newBucket = htbl->h(tempData) % newBuckets;

                    // If insertion fails
                    if (list_ins_next(&newTable[newBucket], NULL, tempData) != 0) {
                        // clear the bucket
                        for (int j = 0; j < newBuckets; j++) {
                            list_destroy(&newTable[j]);
                        }
                        free(newTable);
                        return -1;
                    }
                    element = list_next(element);
                }
            }

            // Free old table buckets but not data
            for (int i = 0; i < htbl->buckets; i++) {
                void (*temp_destroy)(void *data) = htbl->table[i].destroy;
                htbl->table[i].destroy = NULL;
                list_destroy(&htbl->table[i]);
                htbl->table[i].destroy = temp_destroy;
            }

            // Free old table and update htbl with new table
            free(htbl->table);
            htbl->table = newTable;
            htbl->buckets = newBuckets;
        }

    return retval;
}
```

```c
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;
}
```

2. Change the method by which hash codes are mapped to buckets to use the multiplication method instead of division method.

```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) {

    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;
    htbl->maxLoadFactor = maxLoadFactor;
    htbl->resizeMultiplier = resizeMultiplier;

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

    return 0;
}
```

```c
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));
}
```

```c
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;

    /* Hash the key

    Change the method by which hash codes are mapped to buckets to use the
    multiplication method instead of division method. */

    const double A = 0.618034;
    double hashValue = (double)htbl->h(data);
    double product = hashValue * A;
    double fractional = product - floor(product);
    bucket = (int)(htbl->buckets * fractional);

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

        double loadFactor = (double)htbl->size / htbl->buckets;

        // When calculated load factor exceeds maximum load factor
        if (loadFactor > htbl->maxLoadFactor) {
            // New hash size
            int newBuckets = (int)(htbl->buckets * htbl->resizeMultiplier);

            // Create new table
            List *newTable = (List *)malloc(newBuckets * sizeof(List));
            if (newTable == NULL)
                return -1;

            // init new buckets
            for (int i = 0; i < newBuckets; i++) {
                list_init(&newTable[i], htbl->destroy);
            }

            // Rehash existing elements
            ListElmt *element;
            void *tempData;
            for (int i = 0; i < htbl->buckets; i++) {
                element = list_head(&htbl->table[i]);
                while (element != NULL) {
                    tempData = list_data(element);

                    // Calculation for new bucket with multiplicative method
                    hashValue = (double)htbl->h(tempData);
                    product = hashValue * A;
                    fractional = product - floor(product);
                    int newBucket = (int)(newBuckets * fractional);

                    // If insertion fails
                    if (list_ins_next(&newTable[newBucket], NULL, tempData) != 0) {
                        // clear the bucket
                        for (int j = 0; j < newBuckets; j++) {
                            list_destroy(&newTable[j]);
                        }
                        free(newTable);
                        return -1;
                    }
                    element = list_next(element);
                }
            }

            // Free old table buckets but not data
            for (int i = 0; i < htbl->buckets; i++) {
                void (*temp_destroy)(void *data) = htbl->table[i].destroy;
                htbl->table[i].destroy = NULL;
                list_destroy(&htbl->table[i]);
                htbl->table[i].destroy = temp_destroy;
            }

            // Free old table and update htbl with new table
            free(htbl->table);
            htbl->table = newTable;
            htbl->buckets = newBuckets;
        }

    return retval;
}
```

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

    ListElmt *element, *prev;
    int bucket;
    const double A = 0.618034;

    /* Hash the key. */
    double hashValue = (double)htbl->h(*data);
    double product = hashValue * A;
    double fractional = product - floor(product);
    bucket = (int)(htbl->buckets * fractional);

    /* 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;
}
```

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

    ListElmt *element;
    int bucket;
    const double A = 0.618034;

    /* Hash the key. */
    double hashValue = (double)htbl->h(*data);
    double product = hashValue * A;
    double fractional = product - floor(product);
    bucket = (int)(htbl->buckets * fractional);

    /* 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;
}
```

b) (3 points) Implement a program that demonstrates inserts and lookups with an auto-resizing hash table. THis program should initialize the hash table to a small number of buckets then begin inserting integers until a resize occurs. After each insert the program should output the following information:

- Number of buckets in the table
- Number of elements in the table
- The table’s load factor
- The table’s max load factor
- The tables’s resize multiplier

For example, if the hash table was initialized to start with 5 buckets, a max load factor of 0.5, and a resize multiplier of 2.0, the following output should be displayed as elements are inserted:

> buckets 5, elements 1, lf 0.20, max lf 0.5, resize multiplier 2.0
> buckets 5, elements 2, lf 0.40, max lf 0.5, resize multiplier 2.0
> buckets 10, elements 3, lf 0.33, max lf 0.5, resize multiplier 2.0

Note that in the 3rd line of output the number of buckets has doubled. This happened because the load factor that would result from inserting the 3rd element would have caused the load factor to exceed the max load factor (0.5) so the hash table was auto-resized.

After the resize occurs, your program must demonstrate successfully looking up a value that was inserted before the resize and must also demonstrate unsuccessfully looking up a value that does not exist in the table.

hw5 > hw5.cpp > main()

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

int hash(const void* key) {
    return *(const int*)key;
}

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

int main() {
    // initailize variables
    int numBuckets = 5;
    double maxLoadFactor = 0.5;
    double resizeMultiplier = 2.0;

    // Initialialize hash table
    CHTbl htbl;
    chtbl_init(&htbl, numBuckets, hash, match, free, maxLoadFactor, resizeMultiplier);

    // Insert new data onto the table
    int *data;
    for (int i = 1; i <= 3; i++) {
        data = (int *)malloc(sizeof(int));
        if (data == NULL) {
            fprintf(stderr, "Failed to allocate memory for data\n");
            chtbl_destroy(&htbl);
            return 1;
        }
        *data = i;

        chtbl_insert(&htbl, data);

        /* Output current state after each insert */
        printf("buckets %d, elements %d, lf %.2f, max lf %.2f, resize multiplier %.1f\n",
               htbl.buckets, htbl.size, (double)htbl.size / htbl.buckets, htbl.maxLoadFactor, htbl.resizeMultiplier);
    }

    // Clean up table
    chtbl_destroy(&htbl);

    return 0;
}
```

c) (1 point) Answer the following questions:

1. **What is the Big-O execution performance of an insert now that auto-resizing can take place?**

    The Big-O execution performance of an insert with auto-resizing is O(1) constant time complexity. The worst case is O(n) linear complexity. It is only O(n) when resizing takes place, and O(1) for all other cases.

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?**

    The multiplicative method uses a constant value (A), which makes the hash numbers more uniformly distributed despite the number of buckets. With that being said, it has better performance with resizing because there is no calculation needed. Overall, the multiplicative method has better performance and is more flexible.