Please NOTE: Changes made to the chtbl.h and chtlb.c are in yellow.

```
/*
 * 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;
    double maxLoadFactor;
   double resizeMultiplier;

} CHTbl;

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

void chtbl_destroy(CHTbl *htbl);

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

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

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

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

#endif
```

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

int chtbl_init(CHTbl *htbl, int buckets, double maxLoadFactor, double resizeMultiplier,
int(*h)(const void *key), int(*match)(const void *key1, const void *key2),
void(*destroy)(void *data)){
   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;
}



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, int newElementFlag){
   void *temp;
   int bucket, retval;
   double loadFactor;
   int newBuckets;
   int oldBuckets;
   ListElmt *element, *prev;
   int i;

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

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

   /*Compute load factor in advance for new element to be inserted */
   loadFactor = (double)((*htbl)->size+1)/(double)(*htbl)->buckets;

   /*If load factor > maxLoadFactor, compute new bucket size, init new hash table
   and repopulate it*/
   if(loadFactor >= (*htbl)->maxLoadFactor){
          CHTbl* newHashTable;
         oldBuckets = (*htbl)->buckets;
         /*Create a new hashtable*/
         if((newHashTable = (CHTbl*) malloc(sizeof(CHTbl))) == NULL){
                  printf("Malloc failed\n");
         }
         /*Compute new bucket size and initialize hashtable with updated bucket size*/
         newBuckets = oldBuckets * (*htbl)->resizeMultiplier;
         chtbl_init(newHashTable, newBuckets, (*htbl)->maxLoadFactor,(*htbl)-
         >resizeMultiplier, (*htbl)->h,(*htbl)->match,(*htbl)->destroy);
         /*Loop through each bucket (i.e. each linked list) of old hash table*/
         for(i=0;i<oldBuckets;i++){
```

```
                    prev = NULL;
                    /*Retrieve all elements of a bucket, rehash it and place it in new hashtable*/
                    for (element = list_head(&(*htbl)->table[i]); element != NULL; element=
                    list_next(element)){
                              /* Extract data , remove the element from the current linked list */
                              if (list_rem_next(&(*htbl)->table[i], prev, &temp) != 0) {
                                        return -1;
                              }
                              prev = element;
                              /*insert the extracted element into new hash table. we set the last
                              parameter to zero for old elements*/
                              chtbl_insert(&newHashTable, temp,0);
                    }
                    /*destroy the old list*/
                    list_destroy( &(*htbl)->table[i] );
          }
           /*Point to the new hashtable now*/
           *htbl = newHashTable;
 }

   /*Hash the key using multiplication method*/
   bucket = floor(fmod((*htbl)->h(data)*0.618,1.0) * (*htbl)->buckets);

   /* Insert the data into the bucket. */
   if ((retval = list_ins_next(&(*htbl)->table[bucket], NULL, data)) == 0){
           (*htbl)->size++;
          /*Compute actual load factor*/
          loadFactor = (double)((*htbl)->size)/(double)(*htbl)->buckets;
           if(newElementFlag){
                    newElementFlag = 0;
                    printf("buckets = %4d ,elements = %3d, data = %3d ,lf = %3.2g, max lf =
                    %3.2g, resize multiplier = %3.2g\n",
                    (*htbl)->buckets,(*htbl)->size,*(int*)data,loadFactor,(*htbl)-
                    >maxLoadFactor,(*htbl)->resizeMultiplier);
           }
   }
   return retval;
}




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

   ListElmt *element, *prev;
   int bucket;
   /*Hash the key using multitplication method */
   bucket = floor(fmod((htbl)->h(*data)*0.618,1.0) * (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 using multiplication method*/
   bucket = floor(fmod((htbl)->h(*data)*0.618,1.0) * (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;
}
```

TEST PROGRAM

```
/*
 * hashResizeTest.c
 */

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

#define MAX_LOAD_FACTOR 0.5
#define RESIZE_MULTIPLIER 2.0

/*Hashing function is same as key for integers*/
int hashId(const void* key){
	int code = *((int*)key);
	return code;
}

/*Define the matching function for integers*/
int matchIntegers(const void *key1, const void* key2){
	if (*((int*)key1) == *((int*)key2) )
		return 1;
	else
		return 0;
}

int main(){
	int numbers[]={35,67,32,70,18,44,91};
	int i;
	/*Two numbers to look up in hash table. 67 is present,113 is not*/
	int validNum = 67;
	int invalidNum = 113;
	int *vPtr = &validNum;
	int *invPtr = &invalidNum;

	CHTbl* hTable = (CHTbl*)malloc(sizeof(CHTbl));
	chtbl_init(hTable,5,MAX_LOAD_FACTOR,RESIZE_MULTIPLIER,
hashId,matchIntegers,free);

	for (i=0;i<sizeof(numbers)/sizeof(*numbers);i++){
		if(chtbl_insert(&hTable,&numbers[i],1) != 0){
			printf("Hash Table insertion failed\n");
			exit(1);
		}
	}

	/*Look up a known value in the hash table*/
```

```
        if(chtbl_lookup(hTable,(void**)&vPtr) == 0){
                printf("YES, Hash table found %d\n",*vPtr);
        }
        else{
                printf("NO,Hash table  didn't find  %d\n",*vPtr);
        }
        /*Look up a non-existent value in the hash table*/
        if(chtbl_lookup(hTable,(void**)&invPtr) == 0)
                printf("YES, Hash table returned a value of %d\n",*invPtr);
        else
                printf("NO,Hash table  didn't find  %d\n",*invPtr);

        free(hTable);
        return 0;

}
```

OUTPUT

```
Ram (master *) HashTableResizing $ ./hw5
buckets  5, elements  1, lf 0.2, max lf 0.5, resize multiplier  2
buckets  5, elements  2, lf 0.4, max lf 0.5, resize multiplier  2
buckets 10, elements  3, lf 0.3, max lf 0.5, resize multiplier  2
buckets 10, elements  4, lf 0.4, max lf 0.5, resize multiplier  2
buckets 20, elements  5, lf 0.25,max lf 0.5, resize multiplier  2
buckets 20, elements  6, lf 0.3, max lf 0.5, resize multiplier  2
buckets 20, elements  7, lf 0.35, max lf 0.5,resize multiplier  2

YES, Hash table found 67
NO,Hash table  didn't find  113
```

c) What is the Big-O execution performance of an insert now that auto-resizing can take place?

ANS: If m= no. of bucket,

n= no. of elements

When we exceed the maximum load factor and need to resize the hash table,, we have to loop through each of the buckets in the old hash table, and rehash all the elements. This operation is O(m) with m>n here.

In all other instances, insert is O(c). Cumulatively, the big-O should be weighted as follows:

(maxLoadFactor)*O(m) + (1-maxLoadFactor)*O(c)

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?

ANS: Division method won't generally yield a prime number for number of buckets every time resizing takes place. As a result, division method can result in frequent collisions upon resizing due to poor hashing.. Whereas multiplication method doesn't suffer from the drawback upon resizing.