UCSD 167113 Data Structure And Algorithms in C/C++
HW5
Cheng FEI

Source Code:

```
//
//  main.c
//  autogrowing-hash-table
//
//  Created by Cheng FEI on 2022/10/26.
//

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

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

int hash(const void *key);
int match(const void *key1, const void *key2);

int main(int argc, const char * argv[]) {
    /* Initialize the htbl. */
    CHTbl htbl;
    chtbl_init(&htbl, 5, &hash, &match, NULL, 0.5, 2.0);
    int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *exist = nums+9;
    int *notExist = (int *) malloc(sizeof(int));
    *notExist = 100;
    int found = 0;

    /* Insert elements and output the statistics of the htbl. */
    for (int i = 0; i < 10; i++) {
        chtbl_insert(&htbl, &nums[i]);
        printf("buckets %d, elements %d, lf %.2f, max lf %.1f, resize multiplier %.1f\n",
htbl.buckets, htbl.size, 1.0*htbl.size / htbl.buckets, htbl.maxLoadFactor, htbl.resizeMultiplier);
    }

    /* Look up values. */
    found = chtbl_lookup(&htbl, (void *) &exist);
    if (0 == found) {
        printf("The value %d has been successfully inserted!\n", *exist);
    }
    else printf("The value %d has not been inserted at all!\n", *exist);
    found = chtbl_lookup(&htbl, (void *) &notExist);
    if (0 == found) {
        printf("The value %d has been successfully inserted!\n", *notExist);
    }
    else printf("The value %d has not been inserted at all!\n", *notExist);


    /* Destroy the htbl. */
    chtbl_destroy(&htbl);
    free(notExist);

    return 0;
}

// Hash an integer.
int hash(const void *key) {
    return *(int *)key;
}

// Verify whether two elements are identical.
int match(const void *key1, const void *key2) {
    return *(int *)key1 == *(int *)key2;
}
```

```
//
//  chtbl.c
//  autogrowing-hash-table
//
//  Created by Cheng FEI on 2022/10/26.
//

#include <stdlib.h>
#include <string.h>
#include <math.h>

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

#define A (sqrt(5)-1)/2

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;

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

    /* Initialize the resize flags. */
    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;

    /* Auto-grow the buckets when the load factor exceeds the maximum load factor. */
    if (1.0*(htbl->size + 1) / htbl->buckets >= htbl->maxLoadFactor) {
        chtbl_resize(htbl);
    }

    /* Hash the key. */
    bucket = floor(htbl->buckets * fmod(htbl->h(data)*A, 1.0));

    /* 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 = floor(htbl->buckets * fmod(htbl->h(data)*A, 1.0));

    /* 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 = floor(htbl->buckets * fmod(htbl->h(*data)*A, 1.0));

    /* 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;
}

int chtbl_resize(CHTbl *htbl) {
    List *table;
    ListElmt *listElmt;
    void * data;
    int newBucket;
    int newBuckets = htbl->buckets * htbl->resizeMultiplier;

    /* Allocate space for the hash table. */
    if ((table = (List *) malloc(sizeof(List) * newBuckets)) == NULL) {
        return -1;
    }

    for (int i = 0; i < newBuckets; i++) {
        list_init(&table[i], htbl->destroy);
    }

    /* Pass all elements stored in the old htbl into the new htbl. */
    for (int i = 0; i < htbl->buckets; i++) {
        for (listElmt = list_head(&htbl->table[i]); listElmt != NULL; listElmt =
list_next(listElmt)) {
            data = listElmt->data;
            newBucket = floor(newBuckets * fmod(htbl->h(data) * A, 1.0));
            if (list_ins_next(&table[newBucket], NULL, data) != 0) {
                return -1;
            }
        }
    }

    list_destroy(htbl->table);
    htbl->table = table;

    htbl->buckets = newBuckets;

    return 0;
}
```

Console Output:

```
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 0.30, max lf 0.5, resize multiplier 2.0
buckets 10, elements 4, lf 0.40, max lf 0.5, resize multiplier 2.0
buckets 20, elements 5, lf 0.25, max lf 0.5, resize multiplier 2.0
buckets 20, elements 6, lf 0.30, max lf 0.5, resize multiplier 2.0
buckets 20, elements 7, lf 0.35, max lf 0.5, resize multiplier 2.0
buckets 20, elements 8, lf 0.40, max lf 0.5, resize multiplier 2.0
buckets 20, elements 9, lf 0.45, max lf 0.5, resize multiplier 2.0
buckets 40, elements 10, lf 0.25, max lf 0.5, resize multiplier 2.0
The value 10 has been successfully inserted!
The value 100 has not been inserted at all!
Program ended with exit code: 0
```

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

Answer:
1.
Time complexity - O(n)/amortized O(1).
Space complexity - O(n)/amortized O(1).

Reason: The following code snippet has the largest time/space complexity in the program.

```
// O(n) time/space complexity over the loops.
for (int i = 0; i < htbl->buckets; i++) {
        for (listElmt = list_head(&htbl->table[i]); listElmt != NULL; listElmt=list_next(listElmt) {
                // O(1) time/space complexity in one loop.
                data = listElmt->data;
                newBucket = floor(newBuckets * fmod(htbl->h(data) * A, 1.0));
                if (list_ins_next(&table[newBucket], NULL, data) != 0) {
                        return -1;
                }
         }
}
```

Hence, the overall time/space complexity is O(n). Also, since in every n inserts, there will only be 1 auto-resizing, the amortized time/space complexity are both O(1).

2. Using the division method can likely cause most of the existing elements to cluster partially in the buckets. But the multiplication method improves the randomness of the distribution of the existing elements and further distributes these elements among all new buckets.