

KAIST

EE 209: Programming Structures for EE

Assignment 3: Customer Management Table Assignment

Purpose

The purpose of this assignment is to help you learn how to implement common data structures in C and how to exploit them to achieve modularity in a real-world application. It also will give you the opportunity to gain more experience with the GNU/Linux programming tools, especially bash, emacs, and gdb.

Background

A data structure is a way of organizing data for efficient operation. In this assignment, you will implement the required functionalities (register, unregister, and find) of a customer management program using the following data structures.

- Array: an array is a collection of the same data type, where its elements are stored linearly in memory.
- Linked list: a linked list is a data structure that its elements can be added or removed in efficient way than array.
- Hash table: a more efficient implementation might use a *hash table*, which reduces the complexity to search for data. Hash tables are described in Lecture 10 of this course, or Section 2.9 of *The Practice of Programming* (Kernighan & Pike) and Chapter 14 of *Algorithms in C, Parts 1-4* (Sedgewick).

Your Task

You will implement and improve the customer data management API using various data structures. Your task in this assignment is threefold:

- [Task 1] Implement an API library for the customer data management, using a **dynamically resizable** array.
- [Task 2] Implement the same API library using a linked list.
- [Task 3] Implement the same API library using a hash table.
- [Task 4] Test the correctness of your three libraries and measure the performance.

The customer_manager Interface

`customer_manager` is an API library for the customer data management, where the users can register the customer information and perform lookup operations to retrieve the purchase amount information.

The `customer_manager` interface introduces a structure type definition, `DB_T`:

- Conceptually, the `DB_T` structure is used for saving the pointer(s) to the entire customer data.
 - The per-customer data includes the customer name, ID, and purchase amount.

The `customer_manager` interface is described in a file named `customer_manager.h`, and it contains these function declarations:

```

DB_T CreateCustomerDB(void);
void DestroyCustomerDB(DB_T d);
int RegisterCustomer(DB_T d, const char *id, const char *name, const int purchase);
int UnregisterCustomerByID(DB_T d, const char *id);
int UnregisterCustomerByName(DB_T d, const char *name);
int GetPurchaseByID(DB_T d, const char *id);
int GetPurchaseByName(DB_T d, const char *name);
typedef int (*FUNCPTR_T)(const char* id, const char* name, const int purchase);
int GetSumCustomerPurchase(DB_T d, FUNCPTR_T fp);

```



customer_manager.h

What each function does is as follows:

- CreateCustomerDB should allocate memory for a new DB_T object and any underlying objects.
 - On success, it should return a pointer to the memory block for the new DB_T object.
 - If the function fails to allocate the memory, it should return NULL.
- DestroyCustomerDB should free all memory occupied by the DB_T object and all the underlying objects.
 - If the given pointer to the DB_T object is NULL, it should do nothing.
- RegisterCustomer registers a new item, that is, a new user whose ID is id, name is name, and purchase amount is purchase to the DB_T object d.
 - This function should store a new user item with her customer information with d.
 - The new item should **own** the id and name strings by copying them to new buffers. Consider using strdup(). If you use strdup(), please add -D_GNU_SOURCE after gcc209. (e.g., \$ gcc209 -D_GNU_SOURCE -o testclient testclient.c)
 - On success, it should return 0. Otherwise (e.g., on failure), it should return -1.
 - If any of d, id, or name is NULL, it is a failure. If purchase is zero or a negative number, it is a failure.
 - If an item with the same id _or_ with the same name already exists, it is a failure. You should not modify the existing item in this case, and should not leak memory for the new item. A good strategy is to check if such an item already exists, and allocate the new item only if it does not exist.
- UnregisterCustomerByID unregisters a user whose ID is id from the DB_T object, d.
 - On success, it should return 0. Otherwise, it should return -1.
 - If d or id is NULL, it is a failure. If no such item exists, it is a failure.
 - Make sure that you free all the memory allocated for the item being unregistered.
- UnregisterCustomerByName unregisters a user whose name is name from the DB_T object, d.
 - It is identical to UnregisterCustomerByID except that it matches name instead of id.
 - On success, it should return 0. Otherwise, it should return -1.
 - If d or name is NULL, it is a failure. If no such item exists, it is a failure.
 - Make sure that you free all the memory allocated for the item being unregistered.
- GetPurchaseByID searches for the purchase amount of the customer whose ID is id from a DB_T object d.
 - On success, it should return the purchase amount. Otherwise, it should return -1
 - If d or id is NULL, it is a failure. If there is no customer whose ID matches the given one, it is a failure.
- GetPurchaseByName searches for the purchase amount of the customer whose name is name from a DB_T object d.
 - It is identical to GetPurchaseByID except that it matches name instead of id.
 - On success, it should return the purchase amount. Otherwise, it should return -1
 - If d or name is NULL, it is a failure. If there is no customer whose name matches the given one, it is a failure.

- GetSumCustomerPurchase calculates the sum of the numbers returned by fp by calling fp for each user item. That is, this function iterates every user item in d, calls fp once for each user item, and returns the sum of the numbers returned by fp.
 - On success, GetCustomerPurchase should return the sum of all numbers returned by fp by iterating each user item in d.
 - If d or fp is NULL, it should return -1.
 - Note that fp is provided by the caller of GetSumCustomerPurchase. fp is a function pointer that takes user's id, name, and purchase as parameters, evaluates a specific condition on it, and returns a non-negative number. For example, the following code snippet shows the example of the function pointed by fp, which returns the half of the purchase amount of the user whose name contains with "Gorilla".

```
int GetHalfAmountOfUserWhoseNameContainsGorilla(const char *id, const char *name, const int purchase) {
    if (strstr(name, "Gorilla") != NULL)
        return (purchase/2);
    return 0;
}
...

// call of GetSumCustomerPurchase() with the above function
int value = GetSumCustomerPurchase(d, GetHalfAmountOfUserWhoseNameContainsGorilla);
printf("Half the purchase amount of the users whose name contains Gorilla is %d\n", value);
```

[Task 1] The customer_manager Array Implementation

The goal of the first task is to implement the customer_manager API using a dynamically resizable array. Array is the simplest data structure that works well for a small number of user items.

Your first customer_manager implementation should be as follows:

- Follow the function prototype described in [customer_manager.h](#).
- Write your code in a file named customer_manager1.c ([skeleton code](#)).
- Avoid any memory leaks. Your customer_manager implementation should use a dynamically-allocated memory (i.e., in CreateCustomerDB() and DestroyCustomerDB()). It should make sure to free all dynamically-allocated memory when the memory is no longer needed.
- Whenever is possible, validate function parameters by calling the assert macro. Determining which invariant should be maintained is a part of your task.



customer_manager1.c

Implementation tips:

- Define the structure for a user item, and allocate an array of the user item structure when CreateCustomerDB is called. You may start with the initial array size as 64.
- A reasonable user item structure, a DB structure, and CreateCustomerDB is as follows. You can use the following code or define your own structure differently.

```

#define UNIT_ARRAY_SIZE 64

struct UserInfo {
    char *name;    // user name
    char *id;      // user ID
    int purchase;  // purchase amount
};

struct DB {
    struct UserInfo *pArray; // pointer to the array
    int curArrSize;          // current array size (max # of elements)
    int numItems;            // # of stored items,
                             // needed to determine when the array should be expanded

    // add more records below if needed
    ...
};

...
DB_T CreateCustomerDB(void)
{
    DB_T d;

    d = (DB_T) calloc(1, sizeof(struct DB));
    if (d == NULL) {
        fprintf(stderr, "Can't allocate a memory for DB_T\n");
        return NULL;
    }
    d->curArrSize = UNIT_ARRAY_SIZE; // start with 1024 elements
    d->pArray = (struct UserInfo *)calloc(d->curArrSize, sizeof(struct UserInfo));
    if (d->pArray == NULL) {
        fprintf(stderr, "Can't allocate a memory for array of size %d\n", d->curArrSize);
        free(d);
        return NULL;
    }
    return d;
}

```

- Using `calloc()` to allocate the array is often helpful since it initializes all elements (and their fields) to 0.
- For `RegisterCustomer`, find an empty element in the array, and store the new user data in it. Make sure that you copy id and name strings instead of only their pointers. If there is no empty element, expand the array by calling `realloc()`. The increment amount should be 64 elements each time you expand.
- In `UnregisterCustomerByID` or `UnregisterCustomerByName`, you search for the matching item, and deallocate the name and id. Optionally, you can set the name to `NULL`. This way, you can easily know which element is empty by checking each element's name with `NULL`.
- For `GetSumCustomerPurchase`, scan the array from index 0 till the max index, and call `fp` for each **valid** element.
- Feel free to deviate from the above implementation tips if you have your own idea to make the code run faster or more efficiently.

You must implement dynamically resizable array. You will not get a full credit if you do not implement array expansion.

[Task 2] The customer_manager Linked List Implementation

Unfortunately, using an array is slow when you deal with a large number of user items. Frequent registration and unregistration of a user item creates many holes (empty elements) scattered across the array, which, in turn, makes these operations slow. Adding, deleting, and searching

of a user item would eventually depend on linear search (unless you take extra measures to manage the holes separately). We improve the performance and memory usage efficiency of `customer_manager` operations with a linked list in this task.

Your linked list-based `customer_manager` implementation should:

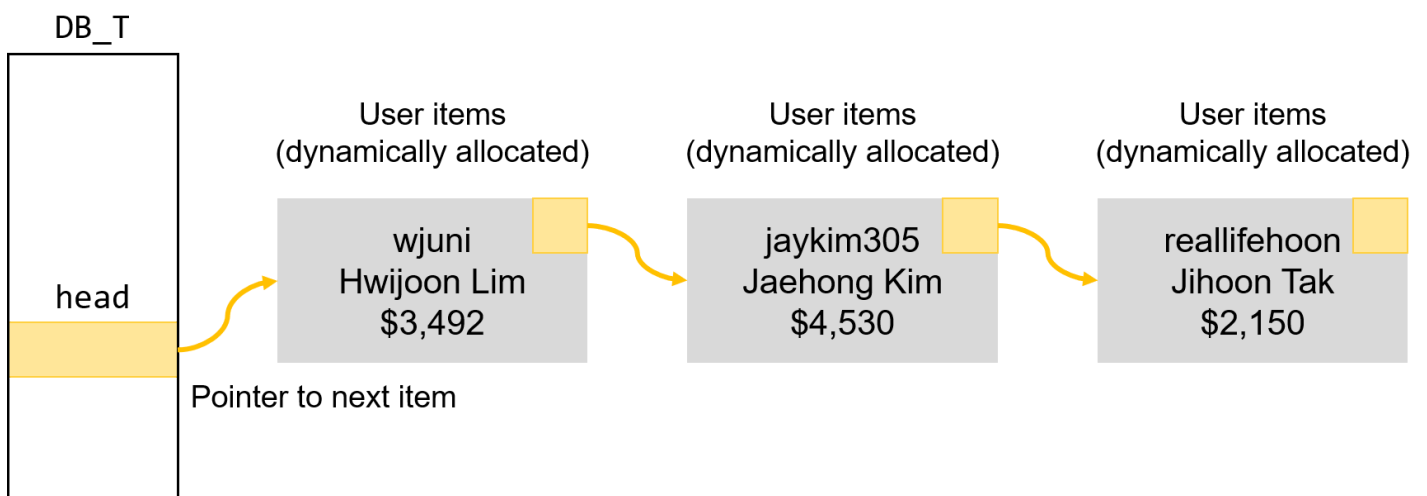
- Follow the function prototype described in `customer_manager.h`.
- Reside in a file named `customer_manager2.c`. (You can copy `customer_manager1.c` and change your implementation into linked list.)
- Avoid any memory leaks. For each call of `malloc` or `calloc` in any object, eventually there should be exactly one call of `free`.
- Only implementing singly linked list would be fine. (You don't have to implement a doubly linked list.)

Implementation tips:

- Reuse the user item data structure that you defined in task 1, but add proper fields to keep track of the next node in a linked list.
- Store the head of the linked list at your DB structure.
- Unlike the array-based implementation, each new registration would require memory allocation of a user item structure. Please review the code in Lecture 10 since your implementation could be similar to it.

The following figure represents an example of linked list-based `customer_manager` implementation.

- Your DB structure must store a pointer to the first item of the linked list.
- Each user item should be allocated dynamically.
- Each user item must link to the next item of the linked list.



[Task 3] The `customer_manager` Hash Table Implementation

Unfortunately, the linked list implementation still requires a large time when searching for certain user items in a very large database. We can improve the performance of `customer_manager` operations with a hash table in this task. Actually, you would need two hash tables. One is for looking up a user item with ID as a key, and the other is for a lookup with a name as a key.

Your hash table-based `customer_manager` implementation should:

- Follow the function prototype described in `customer_manager.h`.
- Reside in a file named `customer_manager3.c`.
- Avoid any memory leaks. For each call of `malloc` or `calloc` in any object, eventually there should be exactly one call of `free`.
- Use a reasonable hash function. You are welcome to use this one. Feel free to modify it or define your own hash function if you'd like.

```

enum {HASH_MULTIPLIER = 65599};
...
static int hash_function(const char *pcKey, int iBucketCount)

/* Return a hash code for pcKey that is between 0 and iBucketCount-1,
   inclusive. Adapted from the EE209 lecture notes. */
{
    int i;
    unsigned int uiHash = 0U;
    for (i = 0; pcKey[i] != '\0'; i++)
        uiHash = uiHash * (unsigned int)HASH_MULTIPLIER
            + (unsigned int)pcKey[i];
    return (int)(uiHash % (unsigned int)iBucketCount);
}

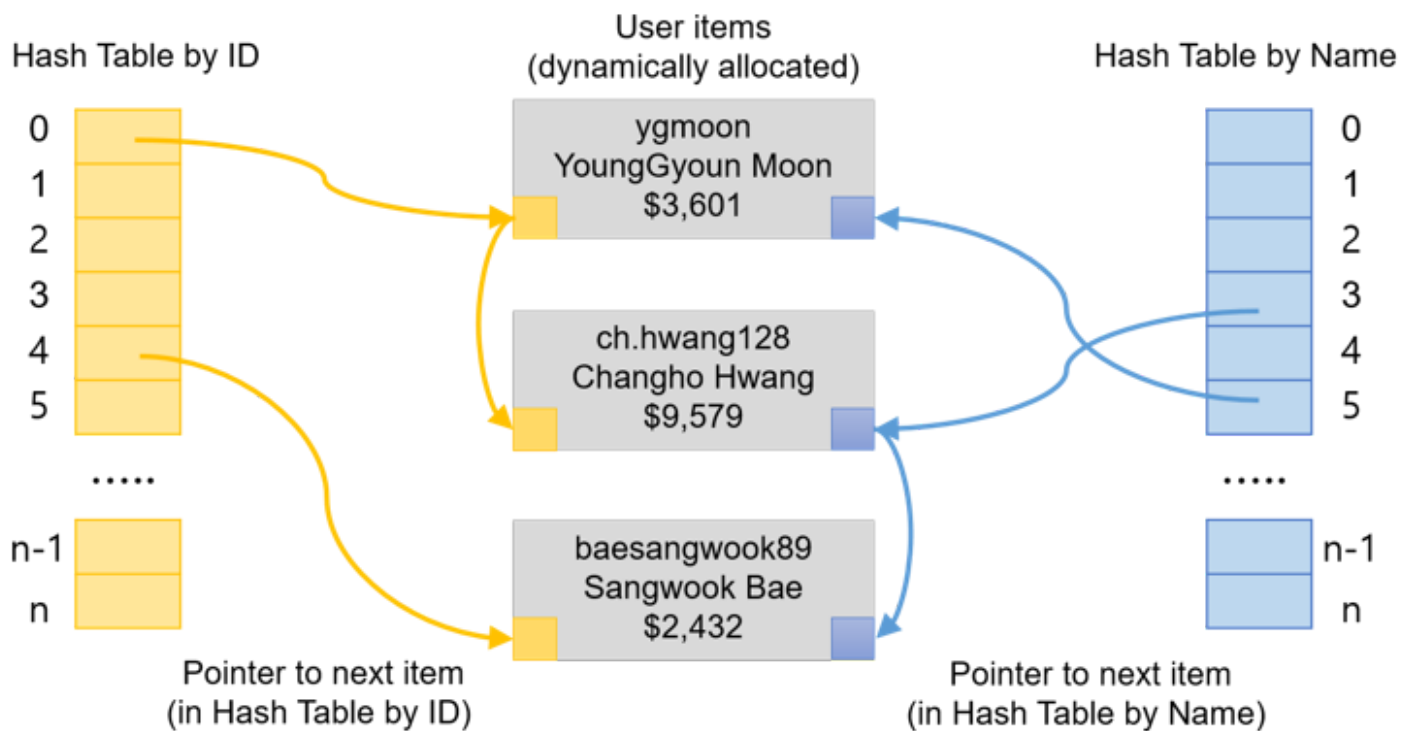
```

Implementation tips:

- Reuse the user item data structure that you defined in task 2, but add proper fields to keep track of the next node in a hash table. You would need two such pointers in the new user item data structure since you will need to maintain two hash tables.
- Unlike the array-based implementation, each new registration would require memory allocation of a user item structure. Please review the code in Lecture 10 since your implementation could be similar to it except that you need to maintain two hash tables here.
- Avoid gross inefficiencies. In particular, it would be grossly inefficient to hash the given key multiple times when a single time would suffice.
- Start the hash table bucket size as 1024 entries.
- **(Extra credit: 15%)** Implement hash table expansion. When the number of nodes (user items) in a hash table reaches 75% of the number of buckets, expand the hash table to double the number of buckets. That is, your initial number of buckets is 1024. When the number of nodes reaches $(0.75 * 1024)$, you expand the number of buckets to 2048. Again, when the number of nodes reaches $(0.75 * 2048)$, the next number of buckets should be 4096. The max number of buckets is 1 million ($1,048,576 = 2^{20}$), so even if the number of nodes exceeds 1 million, don't expand the hash table. Note that even after hash table expansion, you should be able to retrieve all existing user items already registered before hash table expansion. Mark if you implement hash table expansion in your readme file.

The following figure represents an example hash table-based `customer_manager` implementation (it uses the `hash_function` mentioned above).

- `hash_function("ygmoon", n)` and `hash_function ("ch.hwang128", n)` are 0.
- `hash_function("baesangwook89", n)` is 4.
- `hash_function("Changho Hwang", n)` and `hash_function ("Sangwook Bae", n)` are 3.
- `hash_function("YoungGyoun Moon", n)` is 5.



[Task 4] Testing Your Library and Measuring the Performance

We provide [testclient.c](#) to test your implementations. It first checks the correctness of your library functions and measures the performance over various user items. Note that we may use other programs for grading.



testclient.c

To compile your code, do the following:

```
// test your array-based implementation
$ gcc209 -o testclient1 testclient.c customer_manager1.c
$ ./testclient1

// test your hash table-based implementation
$ gcc209 -o testclient2 testclient.c customer_manager2.c
$ ./testclient2
```

(Extra credit: 15%) We will give an extra credit to the students whose implementation is the fastest among all students. Note that only assignments whose basic functionality is implemented without problems deserve the extra credit. We may use our own program to measure the performance.

| Rank # | Extra credit |
|--------|--------------|
| 1 | 15% |
| 2-3 | 10% |
| 4-6 | 5% |
| 7-10 | 3% |

Logistics

Develop in your own environment using emacs to create source code and gdb to debug. Make sure to compile with gcc209 and test your code on lab machine before submission.

Please follow the steps through Task 1 to Task 4 to complete the customer_manager API, and test your libraries.

We give two opportunities for getting an extra credit (each 15%).

- (Extra credit 15%): Implement hash table expansion.
- (Extra credit 15%): Make your implementation the fastest among all submissions.

Create a readme text file that contains:

- Your name.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course "Policy" web page.
- Compare the implementation of customer_manager using array and the one using hash table respectively. Please try to provide reasonable explanation for the pros and cons of each implementation.
- Whether you implemented hash table expansion or not. If you implemented it, state which functions implement it.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Submission

Use [KAIST KLMS](#) to submit your assignments. Your submission should be one gzipped tar file whose name is YourStudentID_assign3.tar.gz

For example, if your student ID is 20191234, please name the file as 20191234_assign3.tar.gz

Create a local directory named 'YourStudentID_assign3' and place all your files in it. Then, tar your submission file. Please refer [here](#) for how to archive your assignment. **Do not archive your submission file without creating a directory and moving your files inside.**

Your submission need to include the following files:

- (Task 1) customer_manager1.c
- (Task 2 and 4) customer_manager2.c
- (Task 3 and 4) customer_manager3.c
- readme text file
- [Observance of Ethics](#). Sign on the document, save it into a PDF file, and submit it.

Your submission file should look like this:

```
20191234_assign3.tar.gz
├── 20191234_assign3
│   ├── customer_manager1.c
│   ├── customer_manager2.c
│   ├── customer_manager3.c
│   ├── readme
│   └── EthicsOath.pdf
```

You can use this [tester](#) before you submit.

Do not add files which has not been mentioned above.

Do not put `customer_manager.h` or another source files in your submission. Your implementation must lay in `customer_manager1.c`, `customer_manager2.c`, and `customer_manager3.c`, not in other files.
(You MUST not modify `customer_manager.h`, either.)

The name of the files must be exact. Note that file name is case sensitive. You might not get the full credit if the names are not correct.

Late Submission

You can use the late submission (late pass; as known as a **token**) which can be late up to three days without penalty for the first four programming assignments. That is, you can apply your late submission days (within 3 days in total) spread over the first four programming assignments. The minimum granularity is one day: if you are 1 hour late, that's still counted as one day late. If you're going to spend your free late days, please say so in your readme file.

You need to notify your token usage in your readme file in following format; `[TOKEN=n]` where `n` is the number of tokens that you want to use. For example, if you want to use 2 tokens, following text MUST be included in your readme.

```
[TOKEN=2]
```

You need to strictly follow this format. Otherwise, TAs will not consider your late token usage.

Grading

If your code cannot be compiled at `ee1ab5` with `gcc209`, we cannot give you any points (0 point). Please double check before you submit.

We will grade your work on quality from the user's point of view and from the programmer's point of view. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages.

From the user's point of view, your module has quality if it behaves as it should.

In part, style is defined by the rules given in *The Practice of Programming* (Kernighan and Pike), as summarized by the [Rules of Programming Style](#) document. These additional rules apply:

Names: You should use a clear and consistent style for variable and function names. One example of such a style is to prefix each variable name with characters that indicate its type. For example, the prefix `c` might indicate that the variable is of type `char`, `i` might indicate `int`, `pc` might mean `char*`, `ui` might mean `unsigned int`, etc. But it is fine to use another style -- a style which does not include the type of a variable in its name -- as long as the result is a readable program.

Line lengths: Limit line lengths in your source code to 72 characters. Doing so allows us to print your work in two columns, thus saving paper.

Comments: Each source code file should begin with a comment that includes your name, the number of the assignment, and the name of the file.

Comments: Each function should begin with a comment that describes what the function does from the caller's point of view. The function comment should:

- Explicitly refer to the function's parameters (by name) and the function's return value.
- State what, if anything, the function reads from standard input or any other stream, and what, if anything, the function writes to standard output, standard error, or any other stream.
- State which global variables the function uses or affects.
- Appear in both the interface (.h) file for the sake of the *clients* of the function and the implementation (.c) file for the sake of the *maintainers* of the function.

Comments: Each structure type definition and each structure field definition should have a comment that describes it.

Please note that you might not get a full credit even if you pass the test with your `testclient`. TAs might use another `testclient` to test functionality and robustness of your implementation.

Comments should be written in English.

FAQ

Question 1

Can I use `assert` macro in my submission source code?

Answer

You can use `assert` for your debugging. However, your code **MUST** still be able to return `NULL` to the caller without your `assert` code.

It is highly recommended to test if your code compiles without the `assert` macro (use `-DNDEBUG` flag when compile). We will use `-DNDEBUG` option with `gcc209` for compiling and grading. Make sure to check your source code before submission. **If your code does not compile with `-DNDEBUG`, you will not get full credit and may get huge deduction in grading.**

Question 2

In Part 1, do I need to shrink the `pArray` back when unregistering customers?

Answer

You don't need to shrink `pArray`.

Question 3

When implementing `RegisterCustomer`, should we consider the case where name or id is blank string (e.g, "") as error and return -1?

Answer

Yes, name or id can be a blank string. Therefore, you should consider it as normal case(not error case).

Question 4

If the unregistering functions cause free item space in the array, should we fill the free space?

Answer

It's up to you handle the free space. As long as it register/unregister function works fine, it won't be a problem.

Question 5

For the following code,

```
struct UserInfo {  
    char *name;  
};  
  
struct UserInfo *p;  
p = (struct UserInfo *) calloc(1, sizeof(struct UserInfo));  
p->name = strdup("Kevin");
```

In order to free all the allocated memory, is it fine just to free(p) or do I need to free(p->name) as well?

Answer

You need to free all the allocated memory including p->name for this case, or else there will be memory leak.

Question 6

In the hash table expansion of the task 3, it states that we need to expand "when the number of nodes in a hash table reaches 75% of the number of buckets". What does it mean?

Answer

It means that you need to expand when total number of User item in hash table reaches 75 % of the number of buckets.