CSc 352 (Summer 04)
Systems Programming and UNIX

THE UNIVERSITY OF ARIZONA
DEPARTMENT OF COMPUTER SCIENCE

Homework #5: C Programming (Structs, ADT, More Pointers) **Version 5**

Assigned: 07/27/2004
Due: 11:59 08/11/2004
Turnin ID: cs352_hw5

I have now provided several files to help you out. I added a compare_func.c and print_func.c that has a bunch of functions for comparing and printing all different types. These are very useful for both arraylist and hashtable. I also have some common hashcodes in hashcode_func.c to help with testing. These files are located at: /home/cs352/summer04/assignments/hw5. Also, to make building your project easier, I have provided a Makefile (even though we have not covered them yet). All you need to do to compile everything is type "make" at the command line. For this to work, all of the following files must be in the same directory:

Makefile arraylist.h hashtable.h compare_func.h hashcode_func.h print_func.h arraylist.c al_test.c hashtable.c compare_func.c hashcode_func.c hashtable_test.c

This makefile is available at: /home/cs352/summer04/assigments/hw5/Makefile

Assignment Description and Requirements

In this assignment, you will be required to write ***2 separate C programs*** and turn them in as follows:
`turnin cs352_hw5 arraylist.c hashtable.c`

For full credit on this assignment, your code must compile cleanly without any compiler warnings. Compile your code as follows to ensure there are no warnings: `gcc -Wall -o -c arraylist.o arraylist.c` The "-W" flag tells the gcc compiler to turn on warnings, and the "all" option tells it to display all warnings. You will be penalized for each compiler warning. Finally, recall that you will also be graded on style. You should never use "magic numbers" or random numerical constants peppered throughout your code. Instead, you should use the preprocessor directive "#define" to define those constants as macros. Also, you should use functions or macros instead of repetitive code.

Program #1: ** [150/100 points] arraylist.c

Write a complete, well-written, documented C file named **arraylist.c** that implements the Abstract Data Type implemented in Java known as an ArrayList. Officially, an ArrayList is described as follows:

"`Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.`"

Similar to the Java implementation of ArrayList, you will write an ArrayList in C that has the ability to store any data type in an internal data structure. Specifically, you will use an array of void pointers so that your ArrayList is generic enough to hold any data type. As described above, the arraylist internally stores data within a resizable array. Since arrays cannot be **reassigned** in C, you will have to use a pointer to a void pointer to implement this correctly. Also, since there is no encapsulation in C, you will have to manage a struct to keep track of all of the data and functions associated with any given arraylist. Specifically, you will represent the arraylist using the following typedefs and structures:

```
typedef struct al *arraylist;

typedef struct node{
  int data_size;
  void *data;
}node;
```

```
typedef struct al{
  int size;
  int capacity;
  node **array;
  int (*comp)(const void *, const void *);
  void (*print)(const void *);
}al;
```

In the above structures, node that an "arraylist" is a pointer to a "struct al". A "struct al" stores all of the internal data associated with a given arraylist. Specifically, it stores the size (the number of actual elements inserted), the capacity (the number of total slots in this arraylist), an array of node pointers, and two ***function pointers.*** The first function pointer, "comp" is a function that is used to compare two objects inserted into the arraylist. Notice the format defines this function as returning an int. This int should return 0 if the two are identical, ~~-1~~ <0 if the first is less than the second, and ~~1~~ >0 if the first is greater than the second. Finally, we have another function pointer, "print", which defines how to print ***one element*** in the arraylist. This function will be a wrapper function around printf which uses the write format based on the data type for that arraylist. ~~For example, if you are inserting strings in the arraylist, the function you will use for "print" is:~~

```
~~void print(void *a){~~
~~  printf("%s", (char *)a);~~
~~}~~
```

For all function pointers where you want to **compare** the values, you can now make use of files I am providing: compare_func.c and compare_func.h. These provides common compare functions. These are both already included in arraylist.h and hashtable.h, so you can simply use these functions without having to create them all over again. Just look at the file, read the comments, and pick the function you want to use based on the types you are inserting into the arraylist. Also, the same is true of the print funtions as well. There are two files, print_func.c and print_func.h that allow you to print lots of different combinations for both arraylist values and hashtable key/value pairs. You only need to be able to support the 4 types I provide functions for, namely: char, int, double, char *.

You must implement the following externally callable functions exactly as defined in the following descriptions. You **must not** include a main method inside arraylist.c, however you may put a main method in another file and compile them together to form an executable and thereby test your program. This was described in the last assignment. If you are confused how to do this, please read the last assignment and talk to me if you are still confused.

For all the functions described below, if you ever have any MALLOC errors, you must execute the following: `printf("%s @ %d: Memory allocation failed!\n", __FUNCTION__, __LINE__);` *then exit with MEM_ERROR. __FUNCTION__ and __LINE__ are macros defined in stdlib.h that evaluate to the function name and line number of the executing code.*

- **arraylist al_init(int (*comp)(const void *, const void *), void (*print)(const void *));**
  This function should initialize all required structures you need to represent an arraylist. Among other things, you will need to malloc space for one **al** (this is the struct that stores the data and function pointers for an arraylist). You should also set all the fields in this struct to default values. It is very important that you remember to also **malloc** space for the array during this initialization process. The array should be malloc'd with DEFAULT_CAPACITY (defined in the header file). Finally, return a pointer to the arraylist that has been initialized.

  Returns:
  A pointer to the arraylist, or NULL if there is a malloc error.

- **arraylist al_init_size(int capacity, int (*comp)(const void *, const void *), void (*print)(const void *));**
  This function should do the same thing as **al_init** except that instead of using DEFAULT_CAPACITY as the size of the array, you instead use the parameter **capacity**. To avoid having two functions that both

do the same thing, **al_init** should simply call **al_init_size** with a capacity value of DEFAULT_CAPACITY.

Returns:
A pointer to the arraylist, or NULL if there is a malloc error.

- **int al_add_at(arraylist list, int index, void *data, int size);**
  Inserts the specified element indicated by **data** at the specified position (**index**) in the arraylist **list**. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices). When data is inserted into the arraylist, you must not assign the contents of the array to the pointer, but must instead **memcpy** the contents of that pointer into the arraylist. This way, if the caller modifies or changes the object it inserted after insertion, it will remain unchanged in the arraylist. In order to perform a memcpy, you need to know the size of the element being copied. Unfortunately, we cannot determine this from a void *, as sizeof() will always report **4** for any pointer. Thus, we need to pass an additional parameter , **size,** that specifies the size of the element to be inserted. To keep the void * conjoined with the size of the data itself, you should make use of the aforementioned structure:

```
typedef struct node{
  int data_size;
  void *data;
}node;
```

Thus, all the elements in your array will **NOT** be void *, but instead will be **node *.** The **node *** stores a pointer to the data itself, along with an int representing the size of that data. This will be needed in several functions in this program. To help make it easier for you to create **nodes** and get **nodes** from the arraylist, I have provied two very useful functions (in the provided skeleton file):

static node *getnode(arraylist list, int index);
static node *make_node(void *data, int size);

Returns:
- NULL_LIST if the specified list is not a valid list (the arraylist pointer is NULL)
- OK if successfully inserted

- **int al_remove(arraylist list, void *data);**
  Removes the specified element indicated by **data** from the arraylist **list**. Because an arraylist can contain duplicates, al_remove should remove the first occurrence of **data** in the arraylist. You should use the provided compare function when the arraylist was created (stored in list->comp) to find **data** in the arraylist. After removing the specified element, you should shift all elements to the ***right*** of that element ***left*** in order to fill in the hole created by the removal. This should also decrement the size of the arraylist. Recall that the arraylist stores ***nodes*** not void *. Thus, as you compare each element of the arraylist with the specified element, you have to extract the data in that element to do the comparision. To make this easier, I have provided a static helper function (in the provided skelaton file): `static void *getdata(arraylist list, int index);` This function should make it significantly easier for you to access the data in each node. Also, when the data is removed, you should call **free_node** (provided in skeleton file) in order to clear out the associated node from the arraylist and free all associated memory.

Returns:
- NULL_LIST if the specified list is not a valid list (the arraylist pointer is NULL)
- ~~**OK**~~ **TRUE** if successfully removed
- **FALSE if not successfully removed (i.e. if data is not in the arraylist)**

- **int al_remove_all(arraylist list, void *data);**
  This function is identical to **al_remove**, except that instead of removing only the first occurrence of **data** in **list**, you instead must remove ***all*** occurrences. On each match, you should perform all of the tasks described in **al_remove**, including calling **free_node**.

Returns:

- NULL_LIST if the specified list is not a valid list (the arraylist pointer is NULL)
- ~~**OK**~~ **TRUE** if successfully removed **ALL the elements matching "data"**
- **FALSE if not successfully removed ALL the elements matching "data"**

➢ **void *al_remove_at(arraylist list, int index);**
This function is identical to **al_remove**, except that instead of removing the first occurrence of some **data** in **list**, you instead must remove the node located at **index** inside the arraylist. If the list and index are valid, you should remove the specified node as described in the previous remove methods. Ensure you call **free_node** and decrement the **size** of the arraylist**..**

Returns:

- NULL if the specified list is not a valid list (the arraylist pointer is NULL), or the index is invalid (<0 or > list->size).
- A void * corresponding to the data that was held at the corresponding index. Observe that you are required to call **free_node** before you return from this function. As described below, **free_node** not only frees the memory occupied by the node, but also the data inside the node. Thus, you must ensure you **memcpy** this data into a new void *, and then return that void *.

➢ **int *al_remove_range(arraylist list, int begin, int end);**
This function is identical to **al_remove_at**, except that instead of removing only one index, you remove from the list all of the elements whose index is between begin, inclusive and end, exclusive. If the list and indexes are valid, you should remove the specified nodes as described in the previous remove methods. Ensure you call **free_node** and decrement the **size** of the arraylist on each removal.

Returns:

- NULL_LIST if the specified list is not a valid list
- BOUNDS if the bounds are illegal (if either is < 0 or either is > list->size, or if end < begin).
- ~~**OK**~~ **TRUE** if successfully deleted **ALL** the corresponding nodes in the appropriate range.
- **FALSE if did not successfully delete ALL the corresponding nodes in the appropriate range**

➢ **int al_clear(arraylist list);**
This function removes all of the elements from the list. The list will be empty after this call returns. Specifically, you must **remove** each element from the list, and ensure you call **free_node** on each element in order to free all memory occupied by the arraylist. Size should be decremented each time, such that it equals 0 when the arraylist is cleared.

Returns:

- NULL_LIST if the specified list is not a valid list
- OK if successfully deleted the corresponding nodes in the appropriate range.

➢ **int al_contains(arraylist list, void *data);**
This function determines whether **list** contains **data.** This should be determined by using a call to the compare function (list->comp). The compare function should return 0 if two data objects are equal. Thus, you should iterate through the list, and if the compare function ever returns 0, you should return **TRUE** to indicate that the list contains the required data. Otherwise, you should return **FALSE** (TRUE and FALSE are declare in the provided header file).

Returns:

- NULL_LIST if the specified list is not a valid list
- TRUE if the list contains **data**
- FALSE if the list does not contain **data**

➢ **int al_ensure_capacity(arraylist list, int capacity);**
This function increases the capacity of this arraylist instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. If the capacity of this

arraylist list is already greater than or equal to **capacity**, this function should simply return OK. Otherwise it should **grow** the array to the required **capacity.** It should do this by creating a new array (using malloc) with the larger size. Then, it should copy all of the elements from the old array into the new array. Once that is finished, ensure you update the capacity in the arraylist structure, and the pointer that points to the array since we have created a new array.

Returns:

- NULL_LIST if the specified list is not a valid list
- OK if the arraylist already has the required capacity
- OK if the arraylist is successfully resized to have the required capacity

- **void *al_get(arraylist list, int index);**
  This function returns the element at the specified location (**index**) within the arraylist. This function should simply check the paremeters, and if they are valid, it should just return the void * stored in the node at the specified index. The value it returns should not be a copy, but instead should be the actual pointer. Thus, if the caller changes the value of this pointer after returned to them, it will also modify it within this data structure. Again, you will likely want to make use of the **getdata** function as it is very useful for this function.

Returns:

- NULL if the specified list is not a valid list or the index is illegal (as described above).
- A void * pointing to the void * inside the node stored at this index
- OK if the arraylist is successfully resized to have the required capacity

- **int al_indexof(arraylist list, void *data);**
  This function should simply iterate over the arrylist, and utilize the comp function to dtermine if **data** is in the arraylist.

Returns:

- NULL_LIST if the specified list is not a valid list (as described above).
- ERR if the arraylist does not contain **dest**
- The index of the first occurrence of **dest** in **list.**

- **void *al_isempty(arraylist list);**
  This function should simply check to see if the array list is empty. If so, it should return TRUE, otherwise it should return FALSE. An arraylist is empty if it contains no elements (it's size should report 0).

Returns:

- NULL_LISt if the specified list is not a valid list(as described above).
- TRUE if the list is empty
- FALSE if the list is not empty

- **void *al_set(arraylist list, int index, void *data, int size);**
  This function should replace the specified node (**index**)within the arraylist with the specified node (created from **data** and **size** using **make_node**) and return a void * corresponding to the old node that is overwritten. You must follow the same basic guidelines for when you insert a node. You first must create a node, then you have to extract the void * from the node currently in the list using memcpy. Finally, you can overwrite the old node with the new node. The size should remain unchanged.

Returns:

- NULL if the list is not a valid list (as described above) or the index is illegal.
- A void * corresponding to the old node that gets overwritten (which could be NULL if that index stores the pointer NULL).

- **void *al_size(arraylist list);**

Simply return the size (as opposed to the capacity) of this arraylist. You should NOT iterate over the list, but instead in constant time just return the value stored in the struct for this arraylist.

Returns:
- NULL_LIST if the list is invalid (as described above).
- >=0 for the number of actual nodes stored in the arraylist.

- **int al_trim(arraylist list);**
  Trims the capacity of the arraylist **list** to the arraylist's current size. The purpose of this function to reduce the amount of memory consumed by an arraylist by getting rid of all the open slots available for future writes. The easiest way to manage this is to create a new array of the required smaller size, then iterate over the old array and copy the nodes into the new array. Then, call **free_node** on each of the nodes in the old array. Finally, update the array pointer and the capacity.

  Returns:
- NULL_LIST if the list is invalid (as described above).
- OK upon successful trimming

- **void *al_printlist(arraylist list);**
  The purpose of this function is to print the arraylist to stdout. This function is basically a glorified wrapper around the provided **print** function pointer provided on creation time. You should just repeatedly call **print** on each element of the arraylist, allowing the user-defined function to take care of the actual printing of one element to stdout.

  Returns:
- NULL_LIST if the list is invalid (as described above).
- OK if the arraylist is successfully printed to stdout.

- **void al_destroy (arraylist *list);**
  This function should simply clear out the arraylist by calling **al_clear**. Once cleared out, it should then free any other internal data structures used to represent the arraylist itself. Thus, the actual pointer used by the caller to access the arraylist (**list** in this case) should be freed. Any subsequent operations or function calls on **list** should fail, producing the error code NULL_LIST. Thus, in addition to explicitly freeing **list,** you must **also** set **list = NULL** as free does not do that!

  Returns:
  - None.

*A sample *.o file, header file and skeleton file at: /home/cs352/summer04/assignments/hw5/~~arraylist/~~*

Program #2: * [150/100 points] hashtable.c**

Write a complete, well-written, documented C file named **hashtable.c** that implements the Abstract Data Type "hashtable" using ***separate chaining.*** You should already be familiar with how a hashtable works as it is covered in cs127b and cs227. If you are not, please see one of us for more detailed help. In general, however, a hashtable is a data structure which maps keys to values. Any non-null pointer can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the pointers used as keys must have a defined hashcode function and an equals function. All hashtables must have a specific mechanism for dealing with values which hash to the same key. Since we are using ***separate chaining,*** the hashtable will insert values which hash to the same key onto the end of a linked list associated with that key.

You will write a hashtable in C that has the ability to store any data type in an internal data structure. Specifically, you will use an array of **hash_element** pointers, which in-turn is composed of a void * for the key, and another void * for the data (and a **next**) poitner to point to the next element in that linked list. Using void * like this allows your hashtable to be generic enough to hold any data type. Specifically,

you will represent the hashtable using the following typedefs and structures:

```
/* The hashtable itself */
typedef struct ht_struct *hashtable;

/* One element or node in the hashtable */
typedef struct hash_node {
  void *key;                        /* unhashed key */
  int key_size;                     /* size of key */
  void *value;                      /* value */
  int value_size;                        /* size of value */
  struct hash_node *next;           /* Pointer to next node */
} hash_node;


/* Struct representing the actual hashtable */
typedef struct ht_struct {
  hash_node **buckets;                  /* an array of pointers to buckets */
  int num_buckets;                      /* Number of buckets, or indexes into array */
  int num_elements;                     /* Number of actual elements inserted == size */
  int (*comp_key)(const void*, const void*);/* compare keys */
  int (*comp_val)(const void*, const void*);/* compare values */
  int (*print)(const void*);            /* print function */
  int (*hashcode)(const void*, int);             /* hashcode function */
}ht_struct;
```

In the above structures, node that a "hashtable" is a pointer to a "struct ht_struct". A "struct ht_struct" stores all of the internal data associated with a given hashtable. Specifically, it stores the array of poitners to buckets, the number of buckets, the number of elements, and three functions: a compare function, a print function, and a hashcode function.

The **comp_key** is a function pointer used to compare keys, it takes two void * and returns 0 if they are the same, <0 if the first one is less than the second one, and >0 if the first one is greater than the second one. Note that these are defined as "const", which means your functions cannot modify these pointers.

Similarly, **comp_val** is used to compare values. It performs the same as above.

Finally, we have another function pointer, "print", which defines how to print ***one element*** in the hashtable. Recall that one element is a **hash_node,** thus you have to parse that hash_node appropriately and print the key/value pairs as described above. This function will be a wrapper function around printf which uses the write format based on the data type for that hashtable. ~~For example, if you are inserting strings in the hashtable, the function you will use for "print" is:~~

```
int printHashEl (const void *a)
{
  hash_node *aa = (hash_node*) a;
  if (a) {
    printf ("(%s, %d)", (char *) (aa->key), *(int *) (aa->value));
    return 1;
  }
  return 0;
}
```

For all function pointers where you want to compare the keys or the values, you can now make use of files I am providing: compare_func.c and compare_func.h. These provides common compare functions. These are both already included in hashtable.h, so you can simply use these functions without having to create them all over again. Just look at the file, read the comments, and pick the function you want to use based on the types you are inserting into the arraylist. Also, the same is true of the print funtions as well. There are two files, print_func.c and print_func.h that allow you to print lots of different combinations of key/value types. You only need to be able to support the 4 types I provide functions for, namely: char,

int, double, char *. Also, for hash functions, I have provided all that you need in hashcode_func.c. Thus, you do not need to write your own hashcode functions. They are relatively simply functions, but they work.

You must implement the following externally callable functions exactly as defined in the following descriptions. You **must not** include a main method inside hashtable.c, however you may put a main method in another file and compile them together to form an executable and thereby test your program. This was described in the last assignment. If you are confused how to do this, please read the last assignment and talk to me if you are still confused.

For all the functions described below, if you ever have any MALLOC errors, you must execute the following: `printf("%s @ %d: Memory allocation failed!\n", __FUNCTION__, __LINE__);` *then exit with MEM_ERROR. __FUNCTION__ and __LINE__ are macros defined in stdlib.h that evaluate to the function name and line number of the executing code.*

- **hashtable ht_init(int (*comp_key)(const void *, const void *),**
  int (*com_val)(const void*, const void *),
  void (*print)(const void *),
  int (*hashcode)(const void *, int));

  This function should initialize all required structures you need to represent a hashtable. Among other things, you will need to malloc space for one ht_struct (this is the struct that stores the data and function pointers for a hashtable). You should also set all the fields in this struct to default values. It is very important that you remember to also **malloc** space for the array during this initialization process. The array should be malloc'd with DEFAULT_CAPACITY (defined in the header file). Finally, return a pointer to the hashtable that has been initialized.

  Returns:
  A pointer to the hashtable, or NULL if there is a malloc error.

- **hashtable ht_init_size(int capacity,**
  int (*comp_key)(const void *, const void *),
  int (*com_val)(const void*, const void *),
  void (*print)(const void *),
  int (*hashcode)(const void *, int));

- This function should do the same thing as **ht_init** except that instead of using DEFAULT_CAPACITY as the size of the array, you instead use the parameter **capacity**. To avoid having two functions that both do the same thing, **ht_init** should simply call **ht_init_size** with a capacity value of DEFAULT_CAPACITY.

  Returns:
  A pointer to the hashtable, or NULL if there is a malloc error.

- **int ht_put(hashtable ht, void *key, int key_size, void *value, int value_size);**
  Inserts the specified element indicated by **value** in the hashtable by getting a hash code for **key** using the hashcode function. It then inserts a **hash_node** into the linked list corresponding to that key within the array of linked lists for that hashtable. When data is inserted, you must not merely set a pointer, but must instead **memcpy** the contents of that pointer into the appropriate hash_element. This way, if the caller modifies or changes the object it inserted after insertion, it will remain unchanged in the hashtable. In order to perform a memcpy, you need to know the size of the element being copied. Unfortunately, we cannot determine this from a void *, as sizeof() will always report **4** for any pointer. Thus, we need to pass an additional 2 parameters:, **key_size,** that specifies the size of the key, and **value_size** that specifies the size of the element to be inserted. To keep the void * conjoined with the size of the data itself, you should make use of the aforementioned structure:

```
typedef struct hash_node {
  void *key;                        /* unhashed key */
```

```
  int key_size;                    /* size of key */
  void *value;                     /* value */
  int value_size;                      /* size of value */
  struct hash_node *next;          /* Pointer to next node */
} hash_node;
```

Thus, all the elements in your array will **NOT** be void *, but instead will be **hash_node *.** The **hash_node *** stores the size of the data, a pointer to the key, and a pointer to the value. This will be needed in several functions in this program. To help make it easier for you to create **hash_nodes** and get **hash_nodes** from the hashtable, you may want to write functions similar to the ones we used in arraylist to get nodes and make nodes.

Returns:
- NULL_HT if the specified hashtable is not a valid hashtable (the pointer is NULL)
- OK if successfully inserted

➢ **void * ht_get(hashtable ht, char *key);**
Returns the value to which the specified key is mapped to in this hashtable.

Returns:
- NULL if the specified hashtable is not a valid hashtable (the pointer is NULL)
- NULL if the key is not mapped to a value in the hashtable
- A void * corresponding to the data value that the key maps to. Note that you are returning the actual void *, you do **not** memcpy a new one. This way, the caller can manipulate what is in the hashtable.

➢ **void *ht_remove(hashtable ht, char *key);**
Removes the key, and its corresponding value, from the hashtable. This function should obtain the hashcode value for key, and go that index in the hashtable. If there is no element in that index in the hashtable, then simply return NULL. Otherwise, iterate through the linked list associated with that code in the hashtable and look for the corresponding key. Once the correct key is found, you should remove that **node** from the hashtable by updating the pointers such that this node is no longer in the linked list. You should also call **free_node** (provided in skeleton file) in order to liberate that memory occupied by the node. Note that you should NOT free the value itself since you are to return that to the caller.

Returns:
- NULL if the specified key does not map to any value
- A void * corresponding to the value associated with this key

➢ **int ht_print(hashtable ht);**
The purpose of this function is to print the hashtable to stdout. This function prints to stdout a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space). Eac set is rendereda s the key, and equals sign =, and the associated element, where the **print** function is used to convert the key and elements to strings.

Returns:
- NULL_HT if the specified hashtable is not a valid hashtable (the pointer is NULL)
- OK if successfully printed

➢ **int ht_contains_value(hashtable ht, void *value);**
This function returns TRUE if this hashtable maps one or moe keys to this value. It returns FALSE if it is not the case that this hashtable maps on or more keys to this value. To implement this function, you should simply traverse the entire hashtable and search for any node which has a value which, when compared using **comp**, evaluates to 0. If so, you return TRUE. Otherwise, you return FALSE.

Returns:
- NULL_LIST if the specified list is not a valid list (the arraylist pointer is NULL)

- TRUE if the hashtable contains one or more keys that map to this value.
- FALSE if the hashtable does not contain one or more keys that map to this value.

➢ **int ht_contains_key(hashtable ht, void *key);**
This function returns TRUE if this hashtable contains the specified key. Note, it should return FALSE if key is NULL, or if the hashtable does not contain the requested key. Otherwise, it should return TRUE if the key is found in the hashtable. To implement this function, you should simply call your ht_get function and see if that returns NULL. If it does return NULL, then the key is not found. If it returns a non-NULL value, then the key exists in the hashtable.

Returns:
- FALSE if key is NULL
- FALSE if the hashtable does not contain this key
- TRUE if the hashtable does contain this key

➢ **int ht_clear(hashtable ht);**
This function should clear the hashtable so that it contains no keys or values. It should free all data structures occupied my the entire hashtable, except for the hashtable itself (the hashtable pointer). Thus, further insertions and operations on the hashtable should be possible after calling ht_clear. The hashtable will be empty after this call returns. Specifically, you must **remove** each element from the list, and ensure you call **free_node** on each element in order to free all memory occupied by the hashtable. Size should be decremented each time, such that it equals 0 when the hashtable is cleared.

Returns:
- NULL_HT if the specified list is not a valid hashtable
- OK if the hashtable is successfully cleared

➢ **void ht_destroy(hashtable *ht);**
This function should simply clear out the hashtable by calling **ht_clear**. Once cleared out, it should then free any other internal data structures used to represent the hashtable itself. Thus, the actual pointer used by the caller to access the hashtable (**ht** in this case) should be freed. Any subsequent operations or function calls on **ht** should fail, producing the error code NULL_HT. Thus, in addition to explicitly freeing **ht,** you must **also** set **ht = NULL** as free does not do that!

Returns:
- None.

➢ **int ht_isempty(hashtable ht);**
Tests if the hashtable is empty or not. An empty hashtable should have a size of 0, and should not map any keys to any values. This should be a constant time operation whereby you simply lookup the value of size.

Returns:
- NULL_HT if the specified hashtable is not a valid hashtable
- TRUE if the hashtable is empty
- FALSE if the hashtable is not empty

➢ **int ht_size(hashtable ht);**
Returns the number of keys in this hashtable.

Returns:
- NULL_HT if the specified hashtable is not a valid hashtable
- >= 0, the size of the hashtable, which is counted by the number of keys->value mappings in the table.

*A sample *.o file, header file and skeleton file at: /home/cs352/summer04/assignments/hw5/~~hashtable/~~*