Draft Report - Assignment 6

By Mira Saini

Purpose

This assignment is meant to walk us through the best, most optimized way to create a hash table. The dynamic nature of this assignment means that steps will have to be done chronologically and means that the Makefile will be significantly edited through the course of the assignment. First, a linked list structure will be created, then optimized. Then a hash table structure using the linked list will be created, then optimized. The linked list and hash table functions should be written in a way that allows the program to run through several iterations without spending too much time. After the structures and optimization is successful, a separate program called uniqq should be implemented to test the hash functions capabilities. In the uniqq program, several lines of text, separated by a newline, will be compared. The program should return the number of unique lines in the program followed by a newline.

Draft Report

For Part I, describe your approach to implementing list_remove and list_destroy. Also, mention how you plan to ensure that all dynamically-allocated memory is cleaned up.
To implement list_remove, I will first check to make sure that ‘ll’ or the head of ‘ll’ does not equal NULL. I will then create a current and previous Node struct variable. The previous var should be assigned to the head of ‘ll’ and the previous should be set to NULL, since nothing is previous to the head. I’ll use a while loop, which will exit if the current equals NULL meaning that it's the end of ‘ll.’ Inside the loop, I’ll use the cmpfn function to compare the current element to iptr. If the same, I’ll then check if the current is the head by checking if the previous == NULL. If so, then the head of ll will be assigned to the next node after the current one (to make sure that ll still has a head). If it's not the head, then I will take the pointer to the one after previous and assign it to the next one after the current, skipping over the current. Then I will free current and return. If cmpfn returns false, then I will make the previous one become the current one and the current one become the one after (current → next).
To implement list_destroy, I will first check to make sure that the pointer ll or ll do not equal NULL. I will then create a current var with struct type Node that will be assigned to the head of the double pointer. I will also create a next var with struct type Node. Then I will use a while loop that checks that current != NULL. Inside the loop, I will assign the next var as the next of current. I will free current and then assign current to next and repeat this until I get to the end of the list. Then I will free my ll pointer and then assign it to NULL.

UPDATED VERSION (finished program):
In order to optimize the linked list to perform at faster speeds, I had to change the way I implemented the linked list in ll.c. I will describe this optimization in Part II.

For Part II, explain any initial ideas you have for optimizing the linked list to improve performance.
Implementing a different type of linked list, such a circular or doubly linked list can improve performance. That way the list can be moved through in a multidirectional way, instead of just straight through which can cause a lot more time to be wasted. Instead of traversing the list from beginning to end, we can go both ways and even circle back around.

For Part III, outline your plan for implementing the hash table, including the choice of hash function and collision handling method.
To prevent collisions, I plan to have a lot of “buckets" in my hash table. I will be using the hash function from the badhash.h file, but will implement more buckets to avoid collisions and long lists that take too much time to traverse. In order to do this, I will set the size of my table to be an extremely large number. Then when I return my hash(key) function, I will use the modulus operator on the returned value and on my extremely large number to diversify the number of buckets that the item can go in. If some values unfortunately are placed in the same bucket, I will use my implementation of a linked list to handle possible collisions in the buckets. To implement each hash function, like to remove/find a certain entry, I will again use hash(key) % size to receive the bucket number and then traverse the linked list inside that bucket (if necessary).

Describe how you plan to use the hash table to implement the uniqq application. Include any initial thoughts on how you will create test inputs and check the output of uniqq.
To implement the hash table in my uniqq application, I plan on first creating a table using hash_create, then using fgets to read from stdin. I will also have a counter variable to keep count of the number of unique lines and a line[255] buffer. As fgets reads input from stdin into my line buffer, I will set the end of line to equal ‘\0’ to indicate it's a string. Then I will use hash_get to check if that line is already in my table. If it returns NULL, then I will use hash_put to place that line in my table and then increment count. After the while loop breaks as no more lines are being read in, I will print the value of the count and use hash_destroy to destroy the table and free up any memory.
To create test inputs and check the output of uniqq, I can create test scripts that test different inputs using my program and compare my output to the outputs of the UNIXX uniq command. I can use ‘diff’ in my test script to check if the values are the same or different.
To test my uniqq program, as I mentioned earlier in my report, I can create a test file called test.txt and then do “./uniqq < test.txt” to see the output of my program. I can then do “sort test.txt | uniq | wc -l” and compare that output to my program’s output.

Final Report

In Part I, you implemented garbage collection—routines that clean up dynamically-allocated memory. How did you make sure the memory was all cleaned up? How did you check?

To check that all of the memory was cleaned up and freed at the end, I ran the command "valgrind ./program_name" to ensure that the number of allocations matched the number of frees. I also used this command to check for any memory leaks caused by a segmentation fault.

In Part II, you made a major optimization to the linked list optimization. What was it, and why do you think it changed the performance of bench1 so much?

To optimize my linked list, I added a tail node. In the first version of the linked list, to add a node required the list to be traversed starting from the head. In my current optimized version, instead of traversing the whole list, the new node is appended to the tail node and then the tail pointer points to the new node. So instead of having to traverse the whole list each time from beginning to end, which takes more time adding a new element as the list grows, the time becomes constant and much faster because of the tail node.

In Part III, you implemented hash tables. What happens to the performance of bench2 as you vary the number of buckets in your hash table? How many buckets did you ultimately choose to use? How did you make sure that your code was free from bugs? What did you test, and how did you test it? In particular, how did you create inputs and check the output of uniqq?

As the number of buckets increased in my hash table, the performance of bench2 became much more optimized and quicker. I ended up creating 9,000,000 buckets to reduce the number of chains and collisions in my table. I ended up playing around with the number of buckets, increasing it until I saw that it led to erroneous behavior and ended up actually decreasing performance. To check my functions (because I had a segmentation fault at one point), I used a ton of conditional statements, checking to see if specific points were equal to NULL. I also used "valgrind -s ./bench2" to check which functions were leading to the segmentation fault. To also test my hash functions, I wrote a short program called trial.c to create a hash table, to add two key:value pairs to it, to retrieve one of those two values and then to destroy the table which looked like this:

```
//header files
Hashtable *table = hash_create();
hash_put(table, "burrito", 3);
hash_put(table, "hotdog", 4);
```

```
int *value = hash_get(table, "hotdog");

printf("hotdog: : %d", *value);
hash_destroy(table);
```

I then added this target to my Makefile. At points where my functions weren't doing the right thing, I used print statements to debug.
To ensure that my code was free from bugs, I used many 'if' statements in my functions and also compared the output of my uniqq program to the UNIX uniq program using "sort test.txt | uniq | wc -l" creating multiple files that housed different inputs. For my bench1, bench2, and toy files, I checked that all of the memory was cleared up and for my uniqq file, I checked that the output was the same as UNIX's uniq using the "sort test.txt | uniq | wc -l" command.

Testing

List what you will do to test your code. Make sure this is comprehensive. This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought. Be sure to test inputs with delays and a wide range of files/characters.

To test my structures and optimization, bench1, bench2, and toy have been provided to me. Toy is supposed to test the linked list functions, bench1 tests the linked list's optimization and speed, and bench2 should test the hash table's optimization and speed. Then uniqq, the program that I build, will be used to implement the hash table functions as well. If I want to ensure that my structures do not result in memory leaks, I can use the command valgrind ./bench1 or valgrind ./bench2 or valgrind ./toy depending on which structure I'm trying to test. To test the speed/optimization of my structures, I can run "time ./<program_name>" depending on which one I want to test. This will return the amount of time it takes for the program to run through. Before optimization of my linked list, this time was almost a minute long, but after, less than a second. Focusing on debugging, I can use multiple if statements to make sure that my pointers are not set to NULL when they are used and that I'm not freeing my memory prematurely. I can also open up GDB if I find that there is memory leakage denoted by a segmentation fault.
To test that the implementation of my uniqq file works correctly, I should be able to create a test.txt file with varying and non-unique lines of text inside. Then when I run "./uniqq < test.txt" the number of unique lines should appear followed by a newline. To make sure the results are correct, I can run sort test.txt | uniq | wc -l which will give me the expected result. If I wish to test this using multiple inputs simultaneously, I can create a few test scripts that test different inputs. I can compare my program output to the standard UNIXX uniq output using 'diff' and ensure that they are the same.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, ``How do I use this thing?''. Don't copy the assignment exactly; explain this in your own words.

To implement my hash table structure into a program, first use the hash_create function to create your table. Then you can implement hash_put to add key:value pairs into the table, using a bucket. Adding more key:value pairs will use different buckets from each other, creating a more optimized hash table. You can use hash_get in your program to retrieve a value based on a key and you can use hash_destroy to remove the entire hash table and clear up any of the memory it was using. To compile and run my uniqq program, first type in "make" to compile and then do "./uniqq > file.txt" to return the number of unique lines in the file called file.txt.
If you would like to test my structures using bench1, bench2, or toy, make sure to add it to the Makefile and then you may simple run ./program_name, or you can use "time" to test the speed/performance or "valgrind" to check for any memory leaks in the program.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, ``How is this thing organized so that I can have a chance of fixing it?''. This section will be longer for a more complicated program and shorter for a less complicated program.

<u>FILE ll.c</u>
Contains my list structure functions: list_create, list_add, list_find, list_destroy, list_remove

<u>FILE ll.h</u>
Contains my Node and LL (list) structure. The Node structure contains item data, pointer to char key, pointer to int value, Node pointer to temp, and a Node pointer to next. In my LL (list) structure, I have a Node pointer to head and a Node pointer to tail. Also I declare all of my list functions from above.

<u>FILE item.c</u>
Contains my string comparison function

<u>FILE item.h</u>
Contains my item structure: contains char key[255] and an int id, also declared my string comparison function here

<u>FILE hash.c</u>

In this file, I define all of my hash functions: hash_destroy, hash_create, hash_get, hash_put.

<u>FILE hash.h</u>
In this file, I create my Hashtable structure which contains LL struct pointer to buckets, a Hashtable pointer to table, and int size. I also declare my hash functions from above here.

<u>FILE uniqq.c</u>
In this file, I created a simple program that returns the number of unique lines in a text file using my hash functions from above.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function?

<u>FILE ll.c:</u>

```
//header files
LL *list_create(void){
   LL *l = (LL*)malloc(sizeof(LL));
   //do an if check, if l == NULL then return NULL
   l->head = NULL
   l->tail = NULL;
   return l;
}
bool list_add(LL *l, item i){
   //allocate memory assign to Node *n using malloc
   //check that n does not equal NULL, otherwise return false
   n->data = *i;
   n->next = NULL;
   if(l->head)==NULL{
      l->head = n
      l->tail = n
   }else{
      //next after tail = n
      l->tail = n
   }
   return true
}

item *list_find(LL *l, bool(*cmpfn)(item*, item*),item*i){
    Node *n = l->head; //start at the beginning
    while(n!=NULL){
        //use cmpfn to compare n and i
        //if the same, return &n->data
        //if not then...
        n = n->next // move on to the next one
    }
    return NULL;
}

void list_destroy(LL **ll){
    //check that ll and *ll don't equal NULL
    Node *current = (*ll)->head
    Node *next;
    while current != NULL{
        //assign next to the next of current
        //free current
        //assign current to next
    }
    free(*ll)
    *ll = NULL
}

void list_remove(LL *ll, bool(*cmpfn)(item*, item*), item *iptr){
    //check that ll and ll->head don't equal NULL
    if cmpfn(&ll->head->data, iptr){
        Node *next = //next after head of ll
        free(ll->head)
        ll->head = next
        //check that head doesn't equal NULL, otherwise tail equals NULL
    }return;
}
Node *current = head //start at beginning
Node *previous = NULL
while current != NULL and !cmpfn(current, iptr){
    previous = current
    current = current->next
}//once loop breaks then...
//check that current != NULL
//set next after previous to be the next after head, skips the middle
```

```
//check that current->next == NULL, if so then ll->tail = previous
free(current)}}
\end{lstlisting}
```

FILE ll.h

```
\begin{lstlisting}
//include header files
typedef struct Node Node;
struct Node{
   item data
   char *key //key of key:value pair
   int *value //value of key:value pair
   struct Node *temp //point back to Node
   struct Node *next //point back to Node
}

typedef LL LL;
struct LL{
   Node *head
   Node *tail
}
//here do all of the function declarations for the ll.c functions
```

FILE item.c:

```
//header files
//define cmp function
bool cmp(item *i1, item *i2){
   return !strcmp(i1->key,i2->key);
}
```

FILE item.h:

```
//header files
typedef struct item item;
struct item{
   char key[255]
   int id
}
//declare cmp function here
```

<u>FILE hash.c:</u>

```
//declare header files
Hashtable *hash_create(void){
    int size = //a really giant int (serves as num of buckets)
    //assign Hashtable *table memory for size of Hashtable using malloc
    //if statement checks that table does not equal NULL
    table -> size = size
    table-> buckets = (LL *)malloc((unsigned long) size *sizeof(LL))
    //check that table->buckets does not equal NULL
    for i in range(size){
        //set each head and tail of each bucket to NULL
    }
    return table
}

bool hash_put(Hashtable *table, char *key, int value){
    //check that key and table don't equal NULL
    size_t index = hash(key) % (unsigned long) table->size
    //check node doesn't equal NULL
    node->key = strdup(key)
    check that node->key doesn't equal NULL
    //use malloc to create memory for node->value
    //check node->value doesn't equal NULL
    *node->value = value
    node->next = NULL
    if table->buckets[index].head == NULL{
        //assign above to node and assign same but .tail to node
    }else{
        table->buckets[index].tail->next = node
        table->buckets[index].tail = node
    }
    return true;
}

int *hash_get(Hashtable *table, char *key){
    //check that table and key don't equal NULL
    size_t index = hash(key) % (unsigned long) table->size
    Node *node = table -> buckets[index].head
    while node != NULL{
        if node->key != NULL and strcmp(node->key, key) == 0){
            return node->value
        }
        node = node-> next //move to next
    }
    return NULL
}

void hash_destroy(Hashtable **table){
    //check that table and *table don't equal NULL
    //loop the number equal to size of table using for loop{
        //assign node to the buckets[i].head to start at beginning
        while(node != NULL){
            node *temp = node
            node = node->next //move to next value
            //free up any memory
        }
    }
    //free up remaining memory and *table pointer
    *table = NULL
}
```

FILE hash.h:

```
//header files
typedef struct Hashtable Hashtable
struct Hashtable{
    LL *buckets
    Hashtable *table
    int size
}
//declare all the functions from hash.c here
```

FILE uniqq.c:

```
//header files, include hash.h

int main(int argc, char **argv){
    Hashtable *table = hash_create()
    char line[255]
    int count = 0
    while loop{
    //use fgets to read from stdin, place into line
    //set end of line to '\0' to indicate string
    if hash_get(table, line) == NULL{ //line isn't in table
        hash_put(table,line,1) //put in table
        count ++ //increment unique counter
    }
    printf("%d\n", count)
    hash_destroy(&table)
    return 0
    }
```

Function Descriptions

The inputs of every function (even if it's not a parameter)
The outputs of every function (even if it's not the return value)
The purpose of each function, a brief description about a sentence long.

***note: all of the pseudocode for each function is in the pseudocode section**

FILE ll.c functions

1. LL *list_create(void): This function creates a new linked list, returning a pointer to the new linked list. There are no parameters, but the function does use the LL structure to create the list of type LL.
2. bool list_add(LL *l, item *i): This function takes a linked list and adds an item to it. The parameters are a pointer to a linked list and a pointer to a value of type item to be added to the linked list. The function will return if successful, else false.
3. item *list_find(LL *l, bool (*cmpfn)(item *, item *), item *i): This function takes a pointer to a linked list and uses the comparison function cmpfn to compare two items. One item 'i' is being compared to elements of the linked list. If cmpfn finds a match, then true is returned and a pointer to the item is returned.
4. void list_destroy(LL **ll): This function frees the memory used by the linked list that is pointed to by ll, which is the function's parameter. Nothing is returned.
5. void list_remove(LL *ll, bool(*cmpfn)(item *, item *), item *iptr): This function removes the node that matches the data pointed to by iptr in the linked list using the cmpfn function to find the matching item. LL *ll is the pointer to the linked list. Cmpfn will return true if the data pointed to by iptr matches the data pointed to by item.

FILE hash.c functions

1. Hashtable *hash_create(void): This function creates a new hashtable. The function should return a pointer to the newly created hash table.

2. bool hash_put(Hashtable *table, char *key, int value): This function inserts a new key:value pair into the hash table. If successful, then the function returns true. The parameters are a pointer to a created hash table, a pointer to the key, and an int value that will be the value part in the key:value pair.
3. int *hash_get(Hashtable *table, char *key): This function returns the int value associated with the key (a parameter). The other parameter is a pointer to a hash table.
4. void hash_destroy(Hashtable **table): This function frees up the memory that was originally allocated to the hash table and sets the pointer Hashtable to NULL. This function does not return anything.

FILE item.c functions

1. bool cmp(item *i1, item *i2): This function compares two item structs using their keys. If they match, then true is returned, else false. The parameters are two pointers to two items.