Assignment 5 – towers AKA mm..OOMs

Jason Maheru

CSE 13S – Winter 24

Purpose

- The purpose of the assignment is to take our first step into abstract datatypes. In this assignment, we will be focusing on hash tables. In Python hash tables are known as dicts. We will also be implementing two performance optimization, which will help improve the performance of the ADTs by over a factor of 100.

Questions

- 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?
- 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?
- 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?

1. I made sure the memory was all cleaned up when listremove and list destroy functions are run. I made sure the existing memory was set to null so there weren't any leaks. I checked the by running valgrind.
2. The major thing was getting rid of the while loop in the list add function and making it into an if-else statement. The while loop iterates through the whole input while with the if-else it checks if the input is worth checking.
3. As you add more buckets the faster the bench2 would run. Ultimately I used 100 buckets. I believe if you run too many buckets it doesn't run as fast.
4. For uniqq I would creat text files with different number of lines and had uniqq print them out and see if it was printing out the right number of unique lines.

Testing

- In the testing phase I would check to see how uniqq handles when being tested with many lines as in different amounts of lines in a file each time.

How to Use the Program

For this program, you run the Makefile and run the .c files. The whole point of the assignment is to see how efficient you can make your ADTS. There isn't much to explain how to run the program. Run ./toy.c and repeat the same for the other .c programs as `./<FILENAME>`. To run uniqq.c you run the command `./uniqq <filename>`.

Program Design

The main purpose of the assignment is to implement data structures in our program. Using linked list we implement hash tables to make our code 10x more efficient than it normally would be.

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.

- For ll.h I need to implement tails so that bench1 can run faster,
- For ll.c I need to add the tail pointer to the list add function and get rid of the while loop and replace it with an else statement with the if statments initially there.
- For hash.h you implement the structure hashtable.
- For hash.c you implement the functions that are used(defined) in hash.h. I'll be including the following headers: badhash.h and hash.h.

Function Descriptions

- HASH.c
- `size_t hash_function(const char *key)`: Computes a hash value for a given key by iterating through its characters and applying a simple hashing algorithm. Takes a pointer to a character array (char *key) as input and returns the computed hash value as a size_t.
- `Hashtable *hash_create(void)`: Creates a new hashtable and returns a pointer to it. Initializes the hashtable's internal data structure and allocates memory for it. Takes no input arguments.
- `bool hash_put(Hashtable *ht, char *key, int val)`: Inserts a key-value pair into the hashtable. Takes a pointer to the hashtable (Hashtable *ht), a string representing the key (char *key), and an integer representing the value (int val). Returns true if the insertion is successful, false otherwise.
- `int *hash_get(Hashtable *ht, char *key)`: Retrieves the value associated with a given key from the hashtable. Takes a pointer to the hashtable (Hashtable *ht) and a string representing the key (char *key). Returns a pointer to the value associated with the key if found, NULL otherwise.
- `void hash_destroy(Hashtable **ht)`: Destroys the hashtable and frees all allocated memory. Takes a pointer to a pointer to the hashtable (Hashtable **ht) to update it to NULL after freeing the memory. Frees memory allocated for hashtable entries and the hashtable itself.