Assignment 5

Caden Roberts

CSE 13S – Winter 24

Purpose

The purpose of this assignment is to experiment with implementing a linked list and a hash table, allowing us to effectively and efficiently store massive amounts of data.

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?

  ANS

  I used the free() function to free all Node *'s and set the Node * passed into list_destroy to NULL. list_remove uses free() to free the Node * and updates the Node before the removed Node to point to Node after the removed Node. I used valgrind ./toy to verify that all memory was freed properly, and the following was output:

```
cadenroberts@cse13svm:~/cawrober/asgn5$ valgrind ./toy
==42240== Memcheck, a memory error detector
==42240== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==42240== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==42240== Command: ./toy
==42240==
==42240==
==42240== HEAP SUMMARY:
==42240==     in use at exit: 0 bytes in 0 blocks
==42240==   total heap usage: 1,001 allocs, 1,001 frees, 16,008 bytes allocated
==42240==
==42240== All heap blocks were freed -- no leaks are possible
==42240==
==42240== For lists of detected and suppressed errors, rerun with: -s
==42240== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- 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?

  ANS

  The optimization was adding a tail pointer to the LL implementation. It allowed us to not have to traverse the entire list every time to add a new item.

- 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?

ANS

Performance improves as we increase the number of buckets. I chose to use 1000 buckets.

- 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?

ANS

I used valgrind, time, toy, bench1, and bench2 to check my codes functionality and make sure it was free of bugs. I tested uniqq with .txt files and manually entering input into stdin.

How to Use the Program

Run make, and uniqq can be ran with a .txt file or input from stdin. The tester files can be ran with time, valgrind, or on their own.

Program Design

The program used an array of user-defined struct ”Linked List” which utilize user-defined struct ”Node” which utilize char arrays and integers for their data.

Pseudocode

File - uniqq.c

```
Function: main(int argc, char *argv[])
Input: Number of command-line arguments argc, array of command-line arguments argv

1. Declare variables:
   1.1. fd as integer
   1.2. file as FILE pointer
   1.3. h as pointer to Hashtable
   1.4. a as integer
   1.5. i as integer
   1.6. ch as character
   1.7. s as character array
   1.8. n as pointer to Node

2. Check the number of command-line arguments:
   2.1. If argc is 1:
        2.1.1. Set fd to STDIN_FILENO
   2.2. Else if argc is 2:
        2.2.1. Open the file specified by argv[1] in read-only mode and assign the file descriptor
            to fd
        2.2.2. If fd is -1 (indicating failure to open the file):
                 2.2.2.1. Return 1 (indicating error)
   2.3. Else:
        2.3.1. Return 1 (indicating error)

3. Open a FILE stream from the file descriptor fd in read mode and assign it to file

4. Create a hashtable h using hash_create function
   4.1. If h is NULL:
        4.1.1. Return 1 (indicating error)
```

```
5. Initialize variable a to 0

6. Initialize character array s with a very large size (e.g., 2147483647)

7. Loop:
   7.1. Read characters from the file stream until end of file (-1) is reached:
        7.1.1. Read a character and assign it to i
        7.1.2. If i is within the printable ASCII range (32 to 126):
                  7.1.2.1. Assign the character to ch
               Else:
                  7.1.2.2. Assign newline character to ch
        7.1.3. If ch is not newline:
                  7.1.3.1. Store ch in the s array at index a and increment a
               Else:
                  7.1.3.2. Terminate s with null character ('\0')
                  7.1.3.3. Add the string in s to the hashtable using hash_put function
                  7.1.3.4. Clear the s array and reset a to 0
   7.2. End loop

8. Initialize a to 0

9. Loop over the hashtable and count the number of elements:
   9.1. Loop over each bucket of the hashtable:
        9.1.1. If the bucket is not NULL:
                  9.1.1.1. Get the head of the linked list from the bucket
                  9.1.1.2. Loop over each node in the linked list and increment a
   9.2. End loop

10. Print the value of a

11. If fd is not STDIN_FILENO:
      11.1. Close the file descriptor fd

12. Return 0 (indicating success)
```

Function Descriptions

bool cmp(item *, item *); Compares 2 item keys, passed in from their pointers, returning true if the keys match.

LL *list_create(void); Initializes a linked list and returns a pointer to the linked list, and NULL if the operation failed.

bool list_add(LL *, item *); Adds an item to a linked list, both passed in from their pointers, and returns whether the operation was successful.

item *list_find(LL *, bool (*)(item *, item *), item *); Searches for an item in a linked list, utilizing our user-defined cmp function, all of which are passed in from their pointers. Returns the pointer to the matching item and NULL otherwise.

void list_destroy(LL **); Takes a pointer to a pointer to a linked list and frees every node in the linked list, followed by freeing the linked list pointer.

void list_remove(LL *, bool (*)(item *, item *), item *); Removes an item from a linked list, using our user defined cmp function to check for the item. All are passed in by their pointers.

Hashtable *hash_create(void); Initializes a hashtable and returns a pointer to the hashtable, and NULL if the operation wasn't successful.

bool hash_put(Hashtable *, char *key, int val); Inserts an int val into a particular key on the hashtable, both are passed in by pointers. Returns whether the operation was successful or not.

int *hash_get(Hashtable *, char *key); Looks for a key in a hashtable, both of which are passed in by their pointers. Returns a pointer to the associated int id.

void hash_destroy(Hashtable **); Takes a pointer to a pointer to a hashtable, frees all memory of the linked lists and contained nodes in the hashtable, and finally frees the hashtable pointer.

Results

Using files toy, bench1, bench2, and uniqq:

```
cadenroberts@cse13svm:~/cawrober/asgn5$ valgrind ./toy
==2253== Memcheck, a memory error detector
==2253== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2253== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2253== Command: ./toy
==2253==
==2253==
==2253== HEAP SUMMARY:
==2253==     in use at exit: 0 bytes in 0 blocks
==2253==   total heap usage: 1,001 allocs, 1,001 frees, 272,016 bytes allocated
==2253==
==2253== All heap blocks were freed -- no leaks are possible
==2253==
==2253== For lists of detected and suppressed errors, rerun with: -s
==2253== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cadenroberts@cse13svm:~/cawrober/asgn5$ time ./bench1
SUCCESS!

real    0m0.039s
user    0m0.026s
sys     0m0.013s
cadenroberts@cse13svm:~/cawrober/asgn5$ ./bench2
SUCCESS!
cadenroberts@cse13svm:~/cawrober/asgn5$ ./uniqq test.txt
4
cadenroberts@cse13svm:~/cawrober/asgn5$ ./uniqq
hey
there
test
buddy
hey
there
test
4
cadenroberts@cse13svm:~/cawrober/asgn5$
```