Assignment 5 – Towers Report

Jackson Dawson

CSE 13S – Winter 24

Purpose

This program provides the ability to quickly and efficiently track the number of unique lines of a given input. This functionality is implemented using a combination of a linked lists and hash tables. The user is able to input via STDIN or give a file to read from. Behind the scenes, a hash table containing buckets of entries, organized using linked-lists makes counting and retrievable fast, effective, and efficient.

Garbage Collection Implementation (Part I)

Explanation of Memory Cleanup:

- **list_destroy** traverses through the linked list, freeing each individual node as it progresses. Then, it frees the pointer to the linked list itself, and sets that pointer to NULL.

```
Function list_destroy(l: Pointer to Pointer to LL)
  n := l->head
  while n != NULL do
    next := n->next
    free(n)
    n := next
  end while
  free(*l)
  *l := NULL
End Function
```

- **list_remove** traverses through the linked list, tracking the current node as well as the previous node. If currently identified node is the node that we are looking to remove, the program will check if it is the head node or not. If it is the head, the head of the list will be set to the next node. If it is not the head, then the previous node is set to point toward the next node. After either condition, the node is freed. Addtionally, the tail is maintained during execution in order to avoid removing additional nodes unintentionally.

```
Function list_remove(l: LL, cmp: Function, i: item)
  n := l->head
  prev := NULL
  while n != NULL do
    if cmp(&n->data, i) then
      if prev == NULL then
        l->head := n->next
        if l->head == NULL then
          l->tail := NULL
        end if
```

```
          else
            prev->next := n->next
            if n->next == NULL then
              l->tail := prev
            end if
          end if
          free(n)
          return
        end if
        prev := n
        n := n->next
      end while
    End Function
```

Verification of Memory Cleanup:

- Valgrind was used to check for memory leaks and ensure that all dynamically allocated memory is properly freed.

Linked List Optimization (Part II)

The necessary improvement that I found was to keep track of the tail of the linked list. This meant initializing every linked list with a tail attribute, so that the end of the list only needs to be found once. Instead of traversing the linked list, starting from the head, each time you want to append a new node at the end, you can simply append the node to the end using access via the tail. This drastically improves performance, transforming the operation from O(n) for each append operation to O(1). Running bench1 initially took me more than 3 minutes, and is now complete in significantly less than one second.

e.g.

```
Function list_add(l: LL, i: item) -> bool
  n := new Node
  if n == NULL then
    return false
  end if
  n->data := i
  n->next := NULL
  // if the list is empty...
  if l->head == NULL then
    l->head := n
    l->tail := n
  // if the list is not empty...
  else
    // add the new node to the end of the list
    l->tail->next := n
    // set the tail of the list to the new node
    l->tail := n
  end if
  return true
End Function
```

Hash Table Implementation (Part III)

The amount of buckets can have a drastic impact on the performance of our hash functions. I began by using 100 buckets, but found that 1000 buckets got me safely under the 1 second time goal for bench2.c. I suppose I could have used more, but it didn't seem necessary.

e.g.

```
Function hash_create() -> HashTablePointer
  Allocate memory for a new hash table structure
  If memory allocation fails
    Return NULL
  End If

  Allocate memory for the table's array of linked list pointers with size HASH_TABLE_SIZE
  If memory allocation for the array fails
    Free the allocated hash table
    Return NULL
  End If

  For each index from 0 to HASH_TABLE_SIZE - 1
    Initialize each linked list in the array
    If initialization of a linked list fails
      Clean up previously initialized linked lists
      Free the array and the hash table
      Return NULL
    End If
  End For

  Return the pointer to the newly created hash table
End Function
```

Bug-Free Code and Testing

- Linked-List
  - Implementations for list_destroy and list_remove were tested using Valgrind to ensure that there were no memory leaks. GDB was used as necessary to verify functionality when updating from "int" items to "struct" items.
- Hash Table
  - Testing included unit testing for important and problematic functions such as hash_put and hash_-create. Valgrind was used to inspect the state of memory leaks and gdb was used to walk through some of the code during development.
- uniqq
  - Bash scripts will be run to determine whether the program outputs the expected results. Various inputs will be tested. Very large file inputs will be essential to test.

How to Use the Program

Start by compiling the program to make it usable. This is achieved by typing the following commands into the terminal:

```
make format
make
```

Now the program "uniqq" is ready to use. Run uniqq with a file you want to read:

```
> uniqq <file_name>
```

To end the program's execution, press:

```
> {CTRL + c}
```

To clean up after yourself, do:

```
make clean
```

You're all done!