Assignment 5 – Towers

Debi Majumdar

CSE 13S – Winter 24

Purpose

This assignment is designed to enhance proficiency in implementing essential data structures and algorithms using C programming. By tackling tasks like implementing linked lists and hash tables, gaining hands-on experience in optimizing code performance, managing memory efficiently, and applying key software engineering principles. Through practical exercises, such as processing large text files and implementing unique word counting algorithms, we learn to effectively apply these data structures to solve real-world problems. Overall, this assignment helps us bridge theoretical knowledge with practical skills, preparing us to tackle complex software development challenges effectively.

Hashes and Cubbies

Hashes and cubbies serve as a metaphor for understanding the concept of hash tables, a fundamental data structure in computer science. Imagine a collection of cubbies, each labeled with a unique number. Within each cubby, items such as books or pens are stored. Similarly, in a hash table, data items are stored in "buckets" identified by unique hash values. Just as finding a cubby based on its number is efficient, hashing allows for quick retrieval of data based on its hashed value. However, like searching through a cubby's contents, accessing data within a hash bucket may require traversing through multiple items. The efficiency of a hash table depends on the distribution of items across buckets and the quality of the hash function used to generate unique hash values. Overall, understanding hashes and cubbies provides a conceptual framework for comprehending the workings of hash tables in managing and accessing data efficiently.

What is a hash table

A hash table is a fundamental data structure used in computer science to efficiently store and retrieve data based on keys. It consists of an array of "buckets" or "slots," each capable of holding one or more key-value pairs. The key is hashed using a hash function, which converts the key into a unique numerical value called a hash code. This hash code determines the index or location within the array where the key-value pair will be stored. Hash tables offer fast access to data because the time complexity for inserting, retrieving, and deleting elements is typically O(1) on average, assuming a good hash function and minimal collisions. However, collisions can occur when multiple keys map to the same hash code, requiring collision resolution techniques such as chaining (using linked lists) or open addressing (probing). Hash tables are widely used in various applications, including database indexing, symbol tables in compilers, caching mechanisms, and implementing associative arrays in programming languages like Python and Java.

Why?

People use hash tables because they offer efficient data storage and retrieval based on keys, making them ideal for a wide range of applications. With constant-time access on average, hash tables excel in rapid operations such as insertion, retrieval, and deletion, making them indispensable for databases, caching systems, and symbol tables. Their flexibility allows for storage of diverse data types, from key-value pairs to sets, accommodating various use cases. Additionally, hash tables use memory efficiently by dynamically allocating space, ensuring minimal wastage. Their scalability enables seamless handling of large datasets

without compromising performance, making them suitable for applications dealing with extensive data. While collisions can occur, modern hashing techniques and collision resolution strategies effectively manage them, maintaining the hash table's overall efficiency. These characteristics, including speed, flexibility, memory efficiency, scalability, and collision handling, make hash tables a preferred choice for implementing data structures and algorithms in software development.

Testing

To comprehensively test the code, I will conduct rigorous testing procedures covering various aspects. This includes unit tests for individual functions, assessing their correctness and functionality, as well as integration testing to evaluate interactions between different components. I will test diverse input scenarios, including valid and invalid data, inputs with delays, and files of varying sizes and characters, ensuring compatibility, scalability, and efficiency. Additionally, I will perform stress testing to assess performance under heavy loads and analyze output against expected results to verify accuracy. By employing automated testing tools and seeking feedback from peers, I aim to ensure the robustness, reliability, and effectiveness of the code across different scenarios and use cases.

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. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags that your program uses, and what they do.

To utilize this program, you'll first need to compile it using a C compiler like Clang or GCC. Begin by compiling the source code files provided, including "uniqq.c", "ll.c", "item.c", "hash.c", and "toy.c". You can compile these files using a command similar to the following in your terminal or command prompt:

```
clang -o uniqcounter uniqq.c ll.c item.c hash.c toy.c main.c
```

After successful compilation, you'll have an executable file named "uniqcounter". To execute the program, use the following command:

```
./uniqcounter
```

This command will run the program, and you can interact with it as required. Additionally, the program may support optional flags to modify its behavior or provide additional functionalities. You can explore these flags by using the "–help" or "-h" flag, which displays a help message with information about available flags and their usage. By following these steps, you can effectively compile, run, and utilize the program, including its toy functionality, for various tasks such as unique word counting and more.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{lstlisting}`

which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[1]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][2][3].

[1] This is my footnote.

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.

The program is organized around several key data structures and algorithms to facilitate efficient maintenance and debugging. The main data structures used in the program include linked lists for managing collections of items and hash tables for efficient key-value pair storage and retrieval. The linked list data structure is implemented using a Node struct to represent each node in the list, with a pointer to the next node. The main algorithms employed in the program include operations for creating, adding to, finding items in, and destroying linked lists. Additionally, hash table operations such as creating, putting, getting, and destroying key-value pairs are implemented to manage hash table functionality. The program design emphasizes modularity and encapsulation, with each data structure and algorithm contained within its respective module (e.g., ll.c, hash.c). This organization allows for easier debugging and maintenance, as changes or fixes can be isolated to specific modules without affecting the overall functionality of the program. Additionally, the use of clear and descriptive function names and comments throughout the codebase aids in understanding and maintaining the program.

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

0.0.1 Main Program (uniqq.c)

The main program serves as the entry point of the application and is responsible for handling user interactions. It reads input from standard input (stdin), typically a text file, and utilizes the hash table implementation to count unique words in the input file. Finally, it outputs the number of unique words to standard output (stdout).

0.0.2 Linked List Module (ll.c, ll.h)

The linked list module implements a linked list data structure for storing key-value pairs. It provides functions for creating, adding to, finding, removing, and destroying linked lists. Additionally, it supports generic item types by using function pointers for comparison.

0.0.3 Hash Table Module (hash.c, hash.h)

The hash table module implements a hash table data structure for efficient storage and retrieval of key-value pairs. It utilizes the linked list module for handling collisions using chaining. This module defines functions for creating, putting, getting, and destroying hash tables.

0.0.4 Item Module (item.c, item.h)

The item module defines the item datatype and comparison function used by the linked list and hash table modules. It allows for flexibility in the type of items stored in the data structures.

0.0.5 Utility Functions

Additional utility functions may be implemented to aid in various operations, such as memory management, input/output handling, and string manipulation.

By structuring the program in this manner, each component can be managed and maintained independently, facilitating easier understanding, debugging, and enhancement of the overall functionality. The

modular design also promotes code reuse and scalability, enabling seamless integration of new features or modifications in the future.

Debugging

To ensure that the code was free from bugs, a comprehensive testing approach was adopted. This included unit testing individual functions to verify their correctness and integration testing to ensure proper interaction between different components of the program. Input data was carefully crafted to cover various edge cases and potential failure scenarios, helping to uncover bugs and validate the program's behavior under different conditions. Additionally, automated testing frameworks and tools were utilized to automate the testing process and detect regressions. For the `uniqq` program, input files with known contents were used to verify that the output matched the expected results, confirming the correctness of the word counting functionality.

Optimization

The major optimization made to the linked list implementation was likely the introduction of a tail pointer. By keeping track of the tail node of the linked list, the list_add function could append new nodes directly to the end of the list without traversing the entire list each time. This optimization significantly reduced the time complexity of adding elements to the linked list, leading to a drastic improvement in performance for bench1. With the tail pointer, the list_add function operates in constant time complexity, making the insertion of elements much faster, especially for large lists.

Garbage Collection

The memory cleanup was ensured by implementing the list_destroy function, which recursively frees all dynamically allocated memory associated with the linked list. This function iterates through each node of the list and deallocates memory for both the nodes and their associated data. To check for memory leaks and ensure that all memory was properly cleaned up, the program was tested using a memory debugging tool such as Valgrind. Valgrind analyzes memory usage during program execution and reports any memory leaks or invalid memory accesses, helping to verify that memory cleanup routines are functioning correctly.

Implementation of Hash Tables

Varying the number of buckets in the hash table affected the performance of bench2. Initially, with a small number of buckets, collisions were more likely to occur, leading to longer probe chains and slower lookups. As the number of buckets increased, the likelihood of collisions decreased, resulting in shorter probe chains and faster lookups. However, increasing the number of buckets also increased memory overhead. Ultimately, the number of buckets chosen for the hash table was determined through experimentation and performance testing. The goal was to strike a balance between minimizing collisions and minimizing memory usage, resulting in optimal performance for bench2.

Function Descriptions

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge.

0.0.6 Main Program Functions

- **main**: This function serves as the entry point of the program. It initializes necessary data structures, reads input files, performs word counting using hash tables, and displays the results to the user.

0.0.7 Linked List Module Functions

- **list_create**: Allocates memory for a new linked list and initializes it to be empty.

- **list_add**: Inserts a new key-value pair into the linked list. If the key already exists, it updates the corresponding value.
- **list_find**: Searches for a key in the linked list. If found, it returns a pointer to the corresponding value; otherwise, it returns NULL.
- **list_remove**: Deletes a key-value pair from the linked list based on the provided key.
- **list_destroy**: Frees memory allocated for the linked list, including all nodes and their associated data.

0.0.8 Hash Table Module Functions

- **hash_create**: Creates a new hash table by allocating memory and initializing its components.
- **hash_put**: Inserts a new key-value pair into the hash table. If the key already exists, it updates the corresponding value.
- **hash_get**: Retrieves the value associated with a given key from the hash table.
- **hash_destroy**: Frees memory allocated for the hash table, including all buckets and their contents.

0.0.9 Item Module Functions

- **cmp**: Compares two items to determine if they are equal. The comparison logic depends on the type of items being compared.

These functions encapsulate the core functionality of the program, providing essential operations for managing data structures, performing word counting efficiently, and ensuring proper memory management throughout the program's execution.

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].

[2] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd ed.* O'Reilly, Cambridge, Mass., 2005.

[3] Walter R. Tschinkel. Just scoring points. *The Chronicle of Higher Education*, 53(32):B13, 2007.