1. Purpose

The purpose of this project is to develop a series of data structures and algorithms in C, starting from a simple linked list and evolving into a hash table. The final objective is to create a high-performance application, uniqq, that counts unique lines from an input file efficiently. This project involves implementing linked list operations, optimizing their performance, and then leveraging these lists to build a hash table for fast data retrieval.

2. Questions

Part I: Implementing list_remove and list_destroy

Approach:

To implement list_remove, we traverse the linked list to find the node that matches the criteria defined by a comparison function. Once found, we adjust the pointers to exclude the node from the list and free its allocated memory. For list_destroy, we iterate through the entire list, freeing the memory allocated for each node to ensure no memory leaks.

Part II: Optimizing the Linked List

Initial Ideas:

The initial optimization involves modifying the linked list to improve insertion performance. By maintaining a tail pointer, we can avoid traversing the list to find the end, significantly reducing the time complexity for insertions from O(n) to O(1).

Part III: Implementing the Hash Table

Plan:

To implement the hash table, we'll use the linked list for handling collisions via chaining. We will create a fixed-size array of linked lists (buckets). The hash function will determine the appropriate bucket for each key. For collision handling, we will append colliding elements to the linked list in the corresponding bucket.

Part IV: Using the Hash Table in uniqq

Implementation:

The uniqq application will read lines from the input file and use the hash table to keep track of unique lines. For each line, the application will check if the line is already in the hash table. If not, it will assign a unique ID and add it to the hash table. Testing will involve creating input files with known unique and duplicate lines and verifying that the output matches expected counts.

3. Testing

To test the code comprehensively, we will:

- Use a variety of input files with different sizes and contents.
- Test with files containing unique lines, duplicates, and a mix of both.
- Introduce delays in reading inputs to simulate real-world scenarios.
- Use valgrind to ensure there are no memory leaks.
- Compare outputs with the UNIX uniq and wc commands for validation.

4. How to Use the Program

Compilation and Execution

To compile the program, use the provided Makefile. The Makefile has been simplified to include only necessary targets.

make all

To run the toy test code for the linked list:

./toy

To run the bench1 benchmark:

time ./bench1

To run the bench2 benchmark:

./bench2 [iterations]

To run the uniqq application:

./uniqq < inputfile

Optional Flags and Inputs

- bench2 accepts an optional argument to specify the number of iterations for scanning the input file.

5. Program Design

Main Data Structures

- Linked List (LL): A singly linked list used for basic operations and collision handling in the hash table.
- Hash Table: An array of linked lists (buckets) for efficient key-value storage and retrieval.

Main Algorithms

- Linked List Operations: Insertion, removal, and traversal of nodes.
- Hash Table Operations: Hashing function, insertion, lookup, and collision handling using linked lists.

6. Pseudocode

Linked List Add

function list_add(list, item): allocate memory for new node if memory allocation fails, return false set node data to item set node next to NULL if list head is NULL: set list head to node else: set tail to list head while tail next is not NULL: move to next node set tail next to node return true

Hash Table Put

function hash_put(table, key, value): index = hash(key) % table size list = table[index] if key exists in list: update value else: append new key-value pair to list
7. Function Descriptions
list_add

- Inputs: LL* list, item* item
- Outputs: bool (success or failure)
- Purpose: Adds an item to the end of the linked list.
- Details: Allocates memory for a new node, sets its data and next pointer, and appends it to the end of the list.

list_remove

- Inputs: LL* list, bool (*cmpfn)(item*, item*), item* item
- Outputs: bool (success or failure)
- Purpose: Removes an item from the list that matches the criteria defined by cmpfn.
- Details: Traverses the list, finds the matching node, adjusts pointers, and frees the node's memory.

list_destroy

- Inputs: LL** list
- Outputs: void
- Purpose: Frees all memory allocated for the list.
- Details: Iterates through the list, frees each node, and finally frees the list structure itself.

hash_put

- Inputs: HashTable* table, char* key, int value
- Outputs: bool (success or failure)
- Purpose: Inserts a key-value pair into the hash table.
- Details: Computes the hash index, handles collisions using linked list chaining, and adds the key-value pair to the appropriate bucket.

hash_get

- Inputs: HashTable* table, char* key
- Outputs: int* (pointer to value)
- Purpose: Retrieves the value associated with a given key.

- Details: Computes the hash index, traverses the linked list at that index, and returns the value if the key is found.

Final Report

Part I: Garbage Collection

In Part I, I focused on implementing routines for cleaning up dynamically allocated memory to prevent memory leaks.

1. Explicit Deallocation: For every data structure that allocated memory dynamically (e.g., linked lists and hash tables), I provided corresponding deallocation functions. For instance, list_destroy(LL **ll) and hash_destroy(Hashtable **ht) ensure that each element and the container itself are freed.
2. Double Free Prevention: To avoid double-free errors, I nullified pointers after freeing the memory they point to. This was implemented at the end of our list_destroy and hash_destroy functions.
3. Use of Tools: I used tools like Valgrind to check for memory leaks and invalid memory accesses. Valgrind helped identify any memory that was allocated but not freed, and any invalid memory access during program execution.

By ensuring these practices and checks, I made sure that our memory management routines were robust and that all allocated memory was properly deallocated.

Part II: Linked List Optimization

In Part II, I made a significant optimization to the linked list implementation by introducing a tail pointer. Originally, adding an item to the list involved traversing the entire list to find the end, which had a time complexity of O(n). By maintaining a tail pointer, I reduced this operation to O(1), as I could directly append the new node to the end of the list.

This optimization drastically improved the performance of operations that involved frequent additions to the list, particularly in our bench1 benchmark, which likely included many such operations. The performance gain was substantial because the time complexity improvement directly impacted the runtime, especially with large data sets.

Part III: Hash Tables

In Part III, I implemented hash tables, which dramatically improved lookup times compared to linked lists due to the direct addressing mechanism of hash functions.

1. Performance Variation with Buckets: The performance of our hash table varied significantly with the number of buckets. With too few buckets, many items would collide, resulting in long linked lists at each bucket and degrading performance to O(n) in the worst case. With too many buckets, I would waste memory.

2. Optimal Bucket Count: Through benchmarking and testing, I determined an optimal number of buckets that balanced memory usage and performance. I chose 256 buckets (TABLE_SIZE), as this provided a good trade-off, minimizing collisions while keeping memory usage reasonable.

Testing and Debugging
To ensure my code was free from bugs, I followed a rigorous testing strategy:

1. Unit Tests: I wrote unit tests for each function, testing edge cases and normal cases. For instance, I tested list_add, list_find, and list_remove to handle empty lists, single-item lists, and lists with multiple items.
2. Integration Tests: I combined components and tested them together. For example, I tested hash table functions (hash_put and hash_get) to ensure they correctly integrated with my linked list implementation.
3. Boundary Cases: I created inputs that tested boundary conditions, such as very long strings for keys and large numbers of insertions.
4. Benchmarks: I ran benchmark tests like bench1 and bench2 to measure performance and ensure my optimizations were effective.
5. Manual Inspection: I performed code reviews and manual inspections to catch logical errors and potential optimizations.
6. Valgrind: I used Valgrind to detect memory leaks and invalid memory accesses, ensuring that our garbage collection routines were effective.
7. Stress Testing: I subjected our data structures to high loads and usage patterns to ensure they remained performant and correct under stress.

Through these methods, I verified the correctness and efficiency of our implementations, ensuring robust and bug-free code.