

Part 1: Data Structures and Algorithms

Problem Overview

The task is to implement a fixed-size hash table with linear probing as the collision resolution method. This hash table will store words parsed from a text file, specifically the novel "A Tale of Two Cities" from Project Gutenberg. The keys are words, and the associated values are integers, with functionalities to insert, remove, and retrieve keys, as well as retrieve the most and least recently inserted or modified keys. The time complexity requirements for these operations are $O(1)$.

Steps taken to solve the task:

The project is divided into key components:

1. **Text Parsing:** Extract words from the text file.
2. **Hash Table Implementation:** Create a hash table using linear probing to handle collision.
3. **Implementing Key Functions**
4. **Testing for $O(1)$ -complexity**

Text Parsing:

To begin, the book text file "98-0.txt" is parsed to extract words, which form the dataset for the hash table operations. The assumption was made that all the words in the file will be used (i.e. no pre-processing is required therefore meta data and contents section will be included). The function `parseTextFile` from `TextParser.cpp` reads the file character by character. Alphabetic characters are converted to lowercase to ensure uniformity (avoiding case-sensitive mismatches). Non-alphabetic characters are treated as word boundaries, splitting the text into individual words.

- **Input:** A text file, where each word is extracted sequentially.
- **Output:** A vector of words, which will later be inserted into the hash table.

Hash Table Implementation:

The hash table is implemented with linear probing for collision resolution. The basic structure includes an array to store key-value pairs, with each key being a word from the parsed text and each value being an integer. The table is of fixed size, and the operations (insert, remove, and get) are designed to execute in constant time on average.

The hash table has a fixed capacity, meaning that the number of slots is predetermined at initialization and does not change (no dynamic resizing). It is a balance between memory efficiency and performance. The capacity is chosen to be a large prime number

to reduce the likelihood of hash collisions, which improves the distribution of the keys in the table. The fixed-size capacity (200003) will be large enough to handle the dataset without a high load factor, ensuring that operations maintain constant time complexity. A prime number (200003) is beneficial as the size of a hash table because it ensures a more uniform distribution of keys across the table, reducing collisions. This prevents alignment with common factors in hash values, optimizing performance for operations like insertion and lookup.

The hash function was initially implemented using `std::hash<std::string>`, which generates a hash code from a word (string). This hash code is then modulo-divided by the table's capacity to determine the index at which the word will be stored in the table.

$$\text{index} = \text{hash}(\text{key}) \% \text{capacity}$$

However, this caused 30% more collisions compared to FNV-1a, which could indicate that the hash function doesn't distribute words uniformly across the table. Therefore FNV-1a will be used since it is known to be a well-distributed hash for string inputs. FNV-1a hashing works by iteratively applying a combination of XOR and multiplication with a prime number to each byte of the input, producing a well-distributed and efficient hash value. Collisions can be greatly reduced by increasing the table's capacity. In a real world scenario it is crucial to find the balance between memory efficiency and performance. With a hash table capacity of 200003 and approximately 141 000 entries, the load factor is about 0.7.

When a collision occurs (i.e., two different keys hash to the same index), linear probing is used to find the next available slot. This involves checking subsequent slots sequentially until an empty slot is found.

To manage memory efficiently, I used `std::unique_ptr` for handling the dynamic memory of hash table entries. This ensures that each slot in the table can either store a valid entry or remain empty without memory leaks. Using `std::unique_ptr` allows us to manage the lifecycle of hash table entries without manually deleting objects. It simplifies memory management and prevents potential issues like double-deletion or dangling pointers. `std::optional` was considered for managing the presence or absence of hash table entries, but `std::unique_ptr` was chosen instead due to concerns about broader compatibility with C++ standards prior to C++17, which may be required in some development environments.

Implementing Key Functions

1. `insert(key, value):`

The insert function begins by calculating the **hash** of the given key using the hash function. This is done to determine the index where the key-value pair should be inserted into the hash table. The hash function maps the key (which is a string) to an integer that is then **modulus-divided** by the table capacity to ensure that the index is within the bounds of the hash table.

After computing the hash, the function checks if the slot at the computed index is **empty** (i.e., `nullptr` or `is_deleted` flag is true). If the slot is available, the key-value pair is inserted directly. If the slot is occupied by another key (i.e., a **collision** has occurred), the function enters the **linear probing** phase to find the next available slot. Linear probing sequentially checks subsequent slots (i.e., `index + 1`, `index + 2`, etc.) until an empty slot is found. Once an available slot is found (either empty or marked as deleted), the function creates a new entry for the key-value pair or updates the existing key's value.

After the insertion or update is complete, the function updates the pointer/index tracking the **most recently inserted key**. This allows the `get_last()` function to retrieve the most recent key-value pair in constant time ($O(1)$). The performance of the insert function relies heavily on the **load factor**, which is the ratio of the number of elements to the capacity of the table. As long as the load factor remains low (typically < 0.7), the insert operation is **amortized $O(1)$** . However, as the table fills up, linear probing takes longer, leading to more collisions and slower performance.

2. **remove(key):**

The `remove(key)` function in the hash table is designed to **mark a key as deleted** rather than physically removing it from the table. This method avoids the overhead of rehashing or shifting elements, ensuring the operation maintains **$O(1)$ complexity**.

The first step is to locate the key in the hash table. This is accomplished using the `find_key` function, which uses **linear probing** to search for the key. The `find_key` function starts by computing the hash for the key, which gives an initial index. It then performs linear probing by checking subsequent slots (i.e., `index + 1`, `index + 2`, etc.) if necessary, until it either finds the key or loops back to the original index (indicating that the key is not present in the table).

Once the `find_key` function returns the index where the key is found, the `remove` function checks if the key at the index matches the desired key and whether it has already been marked as deleted. If the key is present and not deleted, it marks the key as deleted by setting the `is_deleted` flag to true. If the `find_key` function returns an index where the key is either not present or has already been deleted, the `remove` function displays an error message indicating that the key was not found.

3. **get(key):**

The function first checks if the hash table is empty (i.e., if `size == 0`). If the table is empty, it returns `nullptr` and displays an error message indicating that there are no keys to retrieve. The function uses the `find_key(key)` function to locate the

index of the key. This function calculates the hash of the key and checks the corresponding index. If the key is not found at the initial index due to a collision, **linear probing** is used to search subsequent slots for the key.

Once the key is located, if the key is found and is not marked as deleted, the function returns the value wrapped in a `std::unique_ptr<int>`. This ensures proper memory management and safe return of the value. If the key is not found (i.e., the slot is empty or marked as deleted), the function returns `nullptr` and displays an error message indicating that the key was not found.

Under ideal conditions (low load factor, no collisions), the `get(key)` function operates in **$O(1)$** time, since it performs a single hash calculation and a constant number of steps to retrieve the value. However, if the load factor is high and many collisions occur, the function can degrade to **$O(n)$** in the worst case due to the need for linear probing. But as long as the load factor is kept low (typically below 0.7), the average-case complexity remains **$O(1)$** .

4. **get_last():**

The `get_last()` function retrieves the **most recently inserted key-value pair** from the hash table. The function first checks if the hash table is empty or if no valid last inserted key is recorded. This is done by checking two conditions:

- If `size == 0`, meaning there are no elements in the hash table.
- If `last_inserted_index == -1`, meaning no valid key has been inserted yet.

If either condition is met, the function outputs an error message and returns `nullptr` to indicate that there is no key to retrieve.

If the hash table is not empty, the function returns the **most recently inserted key-value pair**. This is done by accessing the `table[last_inserted_index]` entry, where the last key-value pair is stored, and wrapping the result in a `std::unique_ptr<std::pair<std::string, int>>`.

The `get_last()` function operates in constant time ($O(1)$) because the `last_inserted_index` is updated with every insertion, allowing direct access to the most recently inserted key-value pair without the need for searching or probing.

5. **get_first():**

The `get_first()` function retrieves the first inserted key-value pair from the hash table. Similar to `get_last()`, the `get_first()` function first checks whether the hash table is empty or if no valid first inserted key exists. It checks:


- If `size == 0`, meaning there are no elements in the hash table.
- If `first_inserted_index == -1`, meaning no valid first key has been inserted.

If either condition is true, the function outputs an error message and returns `nullptr`. If the hash table contains elements, the function returns the **first inserted key-value pair** by accessing the `table[first_inserted_index]` entry and wrapping the result in a `std::unique_ptr<std::pair<std::string, int>>`.

The `get_first()` function also operates in constant time ($O(1)$) because the `first_inserted_index` is set when the first element is inserted into the table, allowing direct access to this key-value pair without additional search operations.

Testing for $O(1)$ -complexity:

To verify that the functions adhered to $O(1)$ complexity, I tested the project with the provided dataset of approximately 141 000 words from the parsed text file. By measuring the execution times for each operation (insert, get, remove, `get_last`, `get_first`), I confirmed that the hash table functions maintain $O(1)$ performance.

 Microsoft Visual Studio Debug Console

```
Total number of collisions during insertions: 699
Time taken to insert 141633 words: 0.0918875 seconds.
Value for 'the': 141608
Time taken to get 'the': 0.0002807 seconds.
Value for 'and': 141621
Time taken to get 'and': 0.0002701 seconds.
Time taken to remove 'the': 5.5e-06 seconds.
Last inserted key: ebooks, value: 141633
Time taken to get the last inserted key: 4.7e-06 seconds.
First inserted key: the, value: 141608
Time taken to get the first inserted key: 4.3e-06 seconds.
```