ECE 250 — Winter 2024
Electrical and Computer Engineering
University of Waterloo

Due Date: Mar 25, 2024
Professor: Ziqiang Patrick Huang
Lab Instructor: Ahmed Fahmy

Lab 3:

Tokenizing

1 Overview

The objectives of Lab 3 are:

- Solve the problem of tokenization in AI systems.
- Implement and utilize a hash table data structure.

2 Understanding Tokenization

Large Language Models (LLMs) such as ChatGPT take text as their input and return text as their output. However, their intermediate processing is performed using numeric computations. The process by which text is turned into numbers and back again is referred to as *tokenization*. There are two major sub-problems to solve while tokenizing:

1. **Map Tokens to Words**: Given a numeric token, retrieve the corresponding word.
2. **Map Words to Tokens**: Given a word, retrieve the corresponding numeric token.

To address the two sub-problems of tokenization, we shall implement a Bidirectional Hash Table (i.e. Two-Way Dictionary). Such a structure consists of an array and a hash table: **The array is used for tokens $\Rightarrow$ words mapping, and the hash table is used for words $\Rightarrow$ tokens mapping. As a result, we end up having a words $\Leftrightarrow$ tokens mapping**.

Mapping Tokens to Words

We address the first sub-problem using a resizable array in the following manner: Upon inserting words into the two-way dictionary, each word is sequentially pushed back into the array. Essentially, the insertion order of a word, which is also its index in the array (using 1-based indexing) serves as its token. **No duplicates are permitted**. Consequently, each word is assigned a unique token (i.e. index).

Note: You are allowed to use the STL's `std::vector` in solving **this sub-problem**.

Mapping Words to Tokens

We can utilize a hash table in the two-way dictionary to efficiently map words to tokens by storing them as key-value pairs. A word (case sensitive) is used as the *key* and its token is the value.

3 Hash Table Specifications

You should implement a hash table to solve the second sub-problem (i.e. words $\Rightarrow$ tokens mapping). The features and specifications of the hash table are presented in this section.

Underlying Structure

The hash table must be implemented using the standard C++ dynamic array as the underlying data structure. **Do not use STL's `std::vector` in implementing the hash table**.

Hash Function

For a key word (`str`) and a hash table of size (`m`), the hash value can be calculated as:

```cpp
unsigned hash(const std::string & str) {
    unsigned ans = 0;
    for (auto it = str.begin(); it != str.end(); ++it)
        ans = ans * 29 + *it;
    return ans % m;
}
```

Code 1: Hash function for keys of type `std::string`.

Collisions

Hashing collisions are addressed through the external chaining method. **Note**: You may employ either your own implementation of a linked list or utilize the STL's `std::forward_list`.

Expandability and Load Factor

For a hash table of size m that has n key-value pairs, the load factor $\alpha = \frac{n}{m}$ is defined as the average chain length (linked list size). A hash table operation (insert, delete or search) has an amortized run-time complexity of $O(\alpha)$. We seek to make the amortized complexity $O(1)$ for all operations by fixing the load factor. In this project, to make α constant, whenever $\alpha = \frac{n}{m} \geq 0.8$, the hash table is doubled in size (i.e. expanded) to be of size $2 * m$. **To resize a hash table, you need to internally create a new underlying table (i.e. array) of size $2*m$, then re-hash all elements and insert them into the new table, and finally de-allocate the old array**. Typically, when deleting key-value pairs in a hash table, the table shrinks to a smaller size. **For simplicity, you are not required to deleting key-value pairs or shrink the hash table in this project**.

4 Program Requirements

Your program must read commands from standard input (e.g. `cin`) and write to standard output (e.g. `cout`). For each input, the program is expected to execute a designated action with a specified run-time requirement and provide the corresponding output[1] message, as outlined in Table 1.

You may assume that all commands are valid and all input strings are of maximum length of 256 character ASCII. **Strings containing non-alphabetic characters are not permitted in the two-way dictionary. For instance, strings like "Here's" and "end." are not allowed due to the presence of an apostrophe in the first and a period at the end of the second.** For run-time requirements, we define n as the number of words/tokens in an input string, and denote p as the number of token-word pairs in the dictionary. The "Run-time" column in Table 1 specifies complexity requirements based on **amortized analysis**.

[1]The outputs specified in the "Output" column must be displayed as indicated. For example, in the first row, the expected output is the string "success" in lowercase.

Command	Parameters	Action	Output	Run-time
`create`	`size` Range: $[1, 256]$	Create a hash table with a size of `size`. This command appears **exactly once** at the beginning.	`success`	$O(1)$
`insert`	`word`	Insert the `word` into the dictionary **only if** it does not already exist and is entirely alphabetic.	`success` OR `failure`	$O(1)$
`load`	`filename`	Insert all words in the file `filename`. Words in `filename` are separated by white spaces. This operation is successful **only if** at least one word is inserted into the dictionary.	`success` OR `failure`	$O(n)$
`tok`	`word`	Return the numeric token associated with `word`. Return `-1` **if** `word` does not exist in the dictionary.	`token` OR `-1`	$O(1)$
`ret`	`token` Range: $[1, \texttt{MAX\_INT}]$	Retrieve the word associated with `token`, **only if** it exists in the dictionary.	`word` OR `N/A`	$O(1)$
`tok_all`	`W1 W2 W3...`	Tokenize all words, which are separated by white spaces only. **For each** non-existing word print `-1`.	`T1 T2 T3...`	$O(n)$
`ret_all`	`T1 T2 T3...` Range: $[1, \texttt{MAX\_INT}]$	Retrieve the words of all tokens, which are separated by white spaces. **For each** non-existing token print `N/A`.	`W1 W2 W3...`	$O(n)$
`print`	`k` Range: $[0, \texttt{MAX\_INT}]$	Print the keys in the chain at hash table position `k` ($0 \leq$ `k` $< m$). **No output** for invalid `k` or empty chain.	`K1 K2 K3...`	$O(p)$
`exit`	N/A	End the program	N/A	N/A

Table 1: Input/Output Requirements

Provided Files

You are provided with the following files: `driver.cpp` where the `main()` function is implemented, `tokenize.h` where your classes are *defined*, and `tokenize.cpp` where your classes are *implemented*. **Do not create, delete or change the name of the provided files**. In addition, there are some examples of input files along with their corresponding output files. The test files are named `test01.in`, `test02.in`, and so on, with their respective output files named `test01.out`, `test02.out`, etc.

5 GitLab Repository

We are going to use GitLab for code management and submission. The repository is created for you with the URL: `https://git.uwaterloo.ca/ece250-w24/lab3/ece250-w24-lab3-USERID`

6 Evaluation Criteria

Evaluation Focus

Output Accuracy and Performance: The primary focus of the evaluation for Lab 3 will be on the correctness of the output generated by your program and the Performance (Time complexity of the functions designed for different commands). We will use a variety of inputs and edge cases to test the accuracy of your results.

Deductions for Non-Compliance (up to -50 points): Marks will be deducted if your code does not adhere to the specified guidelines, including the prohibition of using C++ STL.

Note: It is imperative that you adhere to the prescribed utilization of a hash table as outlined in the lab manual for your assignment. Failure to comply with this directive will result in a substantial deduction of points from your overall assignment score.

Additional Considerations

Code Inspection: Following the output evaluation, we will review your code to ensure it aligns with the provided notes and guidelines.

Code Comments: Include comments to explain your code logic and design choices.

Testing: Thoroughly test your program with various inputs to ensure its correctness.

No STL: Remember that the utilization of any C++ STL containers or algorithms is prohibited unless otherwise specified.

Performance Evaluation: For Lab 3, we will be checking the time complexity of functions you have designed for different commands.

Memory Leaks: We will be checking for memory leaks in Lab 3.

Use of AI Tools: If you utilized any AI tools in the development of your code, please include a comment in your source code indicating their use.

Code Compilation: Ensure that your code compiles without errors on ECE Linux servers (e.g., eceubuntu, ecetesla). We will make and test your code on these servers. If we cannot compile your code, you may receive a score of 0 for the marking.

Object-Oriented Design Principles:

- You must use proper Object-Oriented design principles in the implementation of your code.
- Create a design using classes that have appropriately private/protected member variables and appropriate public member functions.
- It is not acceptable to simply make all member variables public.
- You should consider separating the interface and implementation of class member functions by placing the declarations in a .h (header) file and the definitions in a .cpp (source code) file.
- Implement constructors for initialization and destructors for resource release in classes.