Building a High-Performance URL Shortener in C

This presentation outlines the design and implementation of a robust and efficient URL shortener in C, engineered to handle essential operations and demonstrate advanced data structure concepts.

The Challenge: Designing a URL Shortener

Our mission is to design and implement a URL shortener in C that efficiently maps long URLs to concise, unique tokens. The system must support core functionalities:

- **Create (Shorten):** Generate a short token for a given long URL.
- **Retrieve (Expand):** Quickly return the original long URL from its short token.
- **Delete:** Remove existing URL mappings.

Crucially, the system must robustly handle **collisions** and ensure **efficient lookups**, even under heavy load.

Why a C-Based Solution?

Beyond the practical benefit of reducing URL length for sharing, this project serves as a concrete demonstration of:

- Effective hash-table usage for rapid data access.
- Sophisticated collision handling strategies.
- Memory-efficient data structures, a hallmark of C programming.

Our Team: Power Rangers

A collaborative effort bringing together specialized expertise:

Nagaraj Hegde

Design & Hashing Strategy

Member B

Storage & Memory Optimization

Member C

I/O & Rigorous Testing

Our URL shortener is a **simple, memory-optimized solution** implemented in C, leveraging a hash table with robust collision handling and Base62-like encoding for compact tokens.

The Core: Hash Table with Chaining

The foundation of our URL shortener's efficiency lies in a carefully chosen data structure. We opted for a classic yet powerful approach:

Primary Data Structure: Hash Table

An **array of buckets**, where each bucket uses **linked-list chaining** to resolve collisions. This structure provides a compelling balance of performance and simplicity.

Efficiency

Achieves an average **O(1) expected time complexity** for insert, lookup, and delete operations, crucial for high-performance systems.

Implementability

Relatively **easy to implement in C**, allowing us to focus on optimization and specific system constraints.

Collision Handling

Chaining provides a **simple, robust mechanism** for managing collisions, accommodating variable bucket loads without complex rehashing schemes.

Auxiliary Structures & Design Choices

To further enhance performance and memory efficiency, we made several strategic design decisions:

- **Unified URL Node Struct:** A single, consolidated structure for URL data prevents duplication, ensuring only one copy of each long URL resides in memory.
- **Fixed-size Table:** Initially, a fixed-size hash table simplifies management, with dynamic resizing as a potential future enhancement if load patterns demand it.
- **Optimized Token Storage:** Short tokens are stored as fixed-length strings (e.g., 8 bytes) or as integers with Base62 conversion, ensuring compact representation.

URLShortener: Abstract Data Type

Understanding the Abstract Data Type (ADT) for our URL shortener is key to grasping its functionality. It defines both the internal state and the external operations.

State & Data Components

table[]

An array of bucket pointers, representing the hash table itself, with a defined size **N**.

N

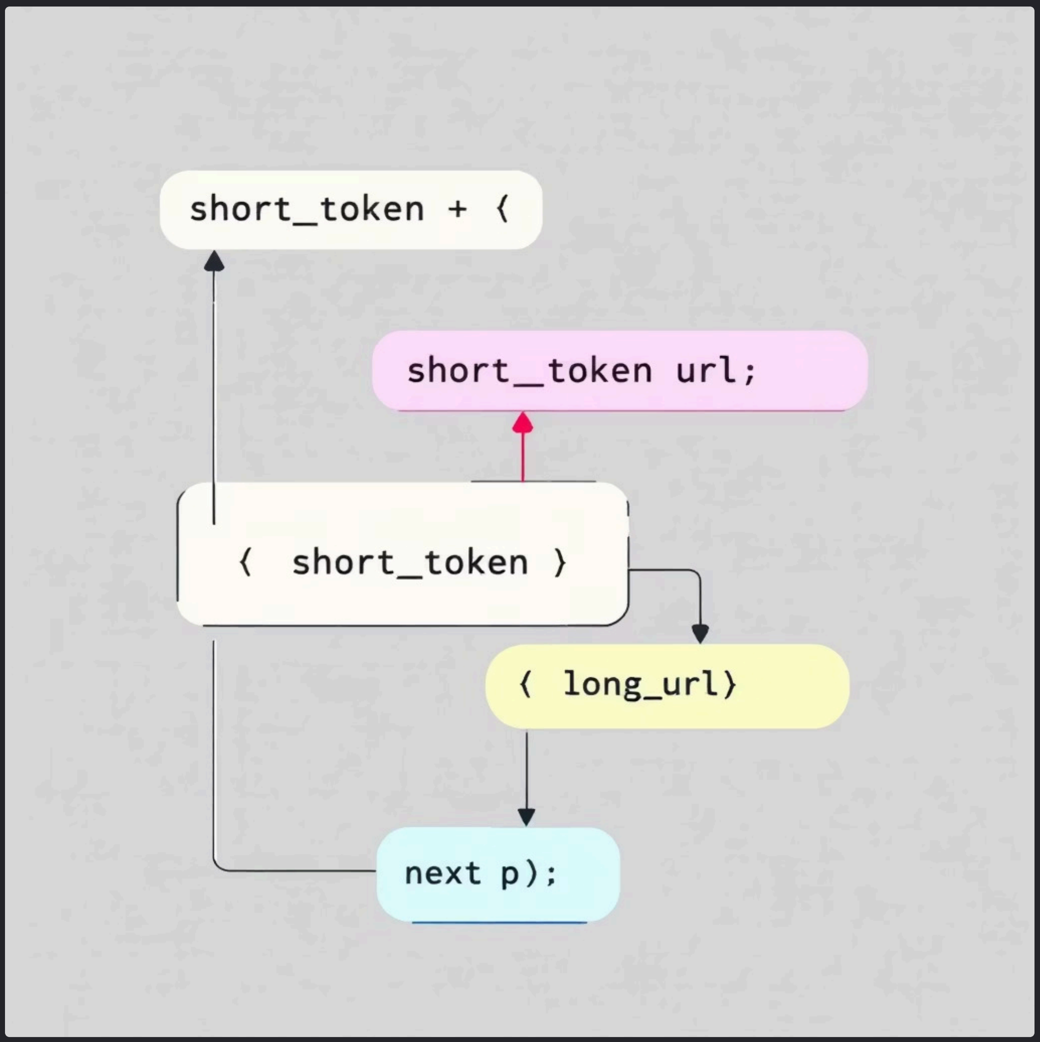
The total number of buckets (or slots) available in the hash table, determining its capacity.

count

A counter tracking the current number of unique URLs (mappings) stored within the shortener.

Node / Data Item Structure

Each entry in our hash table is represented by a node with the following critical fields:



- **short_token:** The compact, unique string or integer representing the shortened URL.
- **long_url:** The original, full-length URL string.
- **next pointer:** A pointer used for chaining, linking to the next node in the event of a hash collision.

Implementation Details & Algorithms

Behind the simple API, several algorithms and design choices ensure the robust and efficient operation of our URL shortener.

Hashing Strategy

Effective hashing is fundamental for hash table performance:

- We will employ well-known **string hash functions**, such as `djb2` or `FNV-1a`, known for their good distribution properties.
- The hash function will be applied to either the **short_token** (for lookups) or the **long_url** (for initial storage/creation) depending on the specific design choice.
- The resulting hash value is then mapped to an array index using the modulo operator: `Index = hash % N`.

Token Generation Methods

Two primary methods for generating unique short tokens are considered:

Monotonically Increasing ID

- Generate a unique, sequentially increasing integer ID.
- Encode this ID into a **Base62-like string** (using characters 0-9, a-z, A-Z) to produce the short token.
- This ensures uniqueness and can be decoded back to the original ID.

Random Token Generation

- Generate a random 6–8 character alphanumeric token.
- Crucially, **check for collisions** in the hash table and regenerate if a duplicate is found.
- This provides good distribution but requires collision detection logic.

Collision Handling: Linked-List Chaining

Our chosen method for collision resolution is elegant and effective:

- When a hash collision occurs, the new URL node is simply **appended to the linked list** at the corresponding bucket.
- Nodes can be added to the head or tail of the list for simplicity.
- During lookup, the linked list is **traversed**, and **short_token strings are compared** to find the exact match.

Algorithmic Complexities

shorten	expand	delete
Expected **O(1)** average time (hash computation + insertion). Worst-case **O(k)** if a bucket becomes excessively long (where 'k' is the bucket length).	Expected **O(1)** average time. Worst-case **O(k)** if the target bucket's linked list needs full traversal.	Expected **O(1)** average time. Worst-case **O(k)** requiring traversal and node removal from a linked list.

Memory Footprint: The total memory usage is **O(M)**, where M represents the cumulative number of characters stored for all URLs, plus a minimal overhead per node (pointers, struct size).