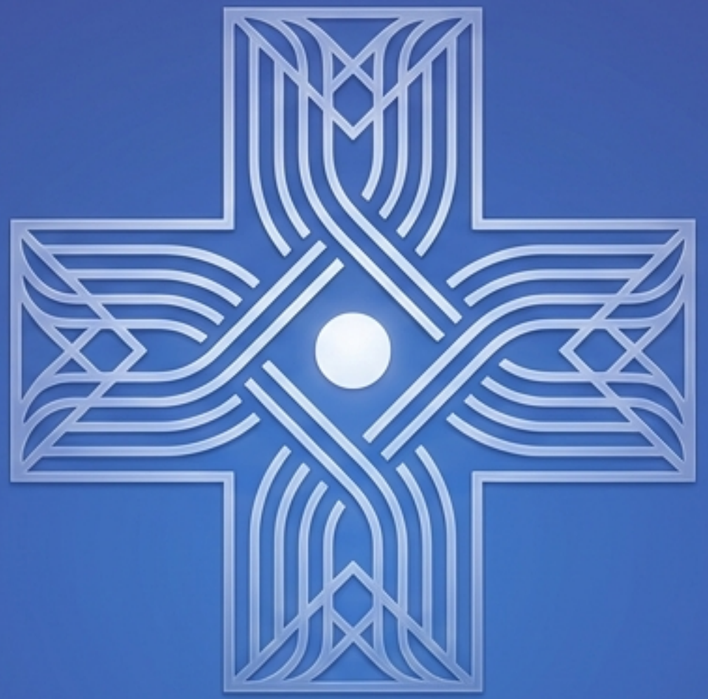


Smart Blood Donation System

A Case Study in Applied Data Structures & Algorithms

Engineering for Millisecond Response in Critical Scenarios

📊 High-Performance Algorithms • 🔒 Custom Data Structures • 🩺 Real-World Application

The Mandate: From Hours to Seconds When Lives Are at Stake

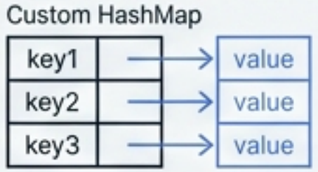
The Problem

- Blood shortages and slow manual search processes lead to **preventable deaths**.
- In emergencies, searching for donors is a race against **time**, where **hours** of **manual calls** are the norm.

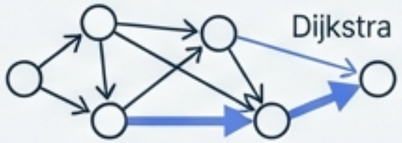
The Engineered Solution

- An automated, **intelligent** system that connects donors with recipients instantly.
- The core mission is to **leverage custom Data** Structures and Algorithms to eliminate search latency.

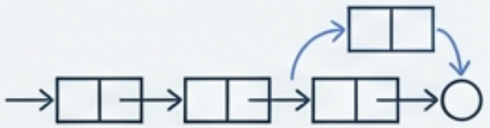
The Architectural Goals



- **O(1) Donor Lookup**: Instantaneous retrieval of donors by blood type using a custom HashMap.



- **Geographic Optimization**: Find the *nearest* available donor, not just any donor, using a Graph and Dijkstra's algorithm.



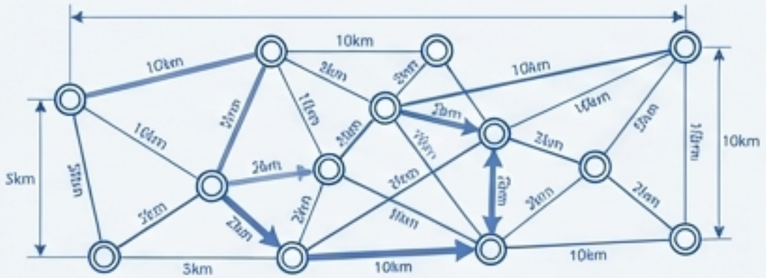
- **Efficient History**: Maintain immutable transaction records with O(1) insertion via a custom LinkedList.

The Core Challenge: Why a Linear Search is Not an Option

With a dataset of over 10,000 users, a linear search (`O(n)`) is dangerously slow in an emergency. Every second of computation is a second lost for a patient.

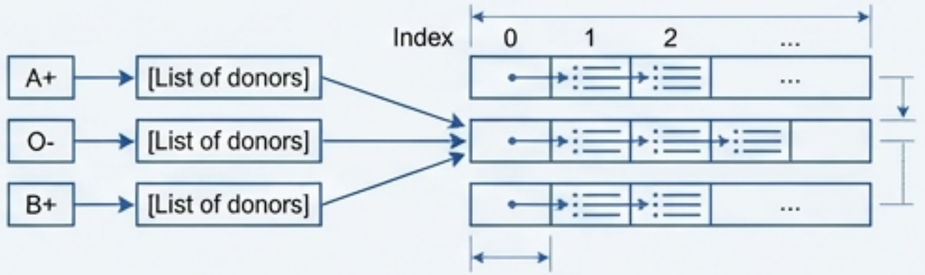
The DSA-Driven Architecture: Our solution is built on a foundation of custom data structures, each engineered for a specific performance objective.

Geographic Matching



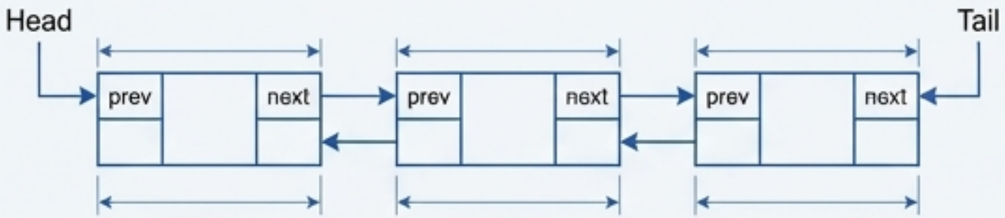
Custom Graph with Dijkstra's algorithm

Blood Type Matching



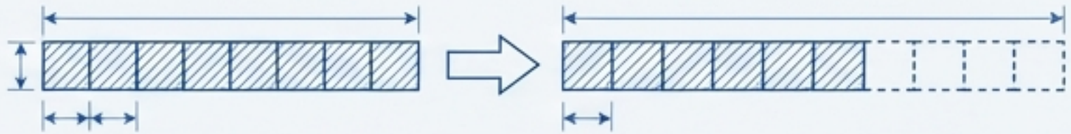
Custom HashMap for `O(1)` lookup

History Tracking



Custom LinkedList for `O(1)` insertion

Dynamic Storage



Custom Vector to handle variable results

Performance Impact

- **Without Custom DSA:** `O(n)` - Scanning 10,000 donors results in unacceptable delays.
- **With Custom DSA:** `O(1) + O(E log V)` - Operations complete in milliseconds.

The User-Facing Architecture: Lightweight and Responsive

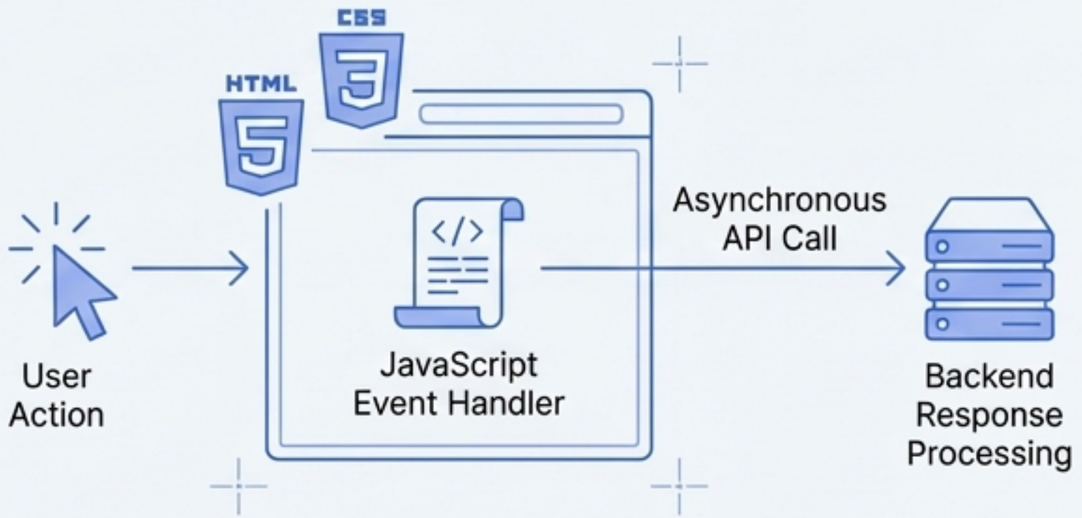
Technology Stack

- **HTML5**: Semantic structure for forms and content.
- **CSS3**: Responsive design, modern gradients, and animations for a clean user experience.
- **Vanilla JavaScript**: Direct DOM manipulation for speed. We intentionally avoided heavy frameworks to minimize load times.

Key Components

- User-facing pages include `index.html`, `login.html`, `register.html`, and `dashboard-donor.html`.
- Client-side session state is managed using `localStorage`.

Interaction Flow

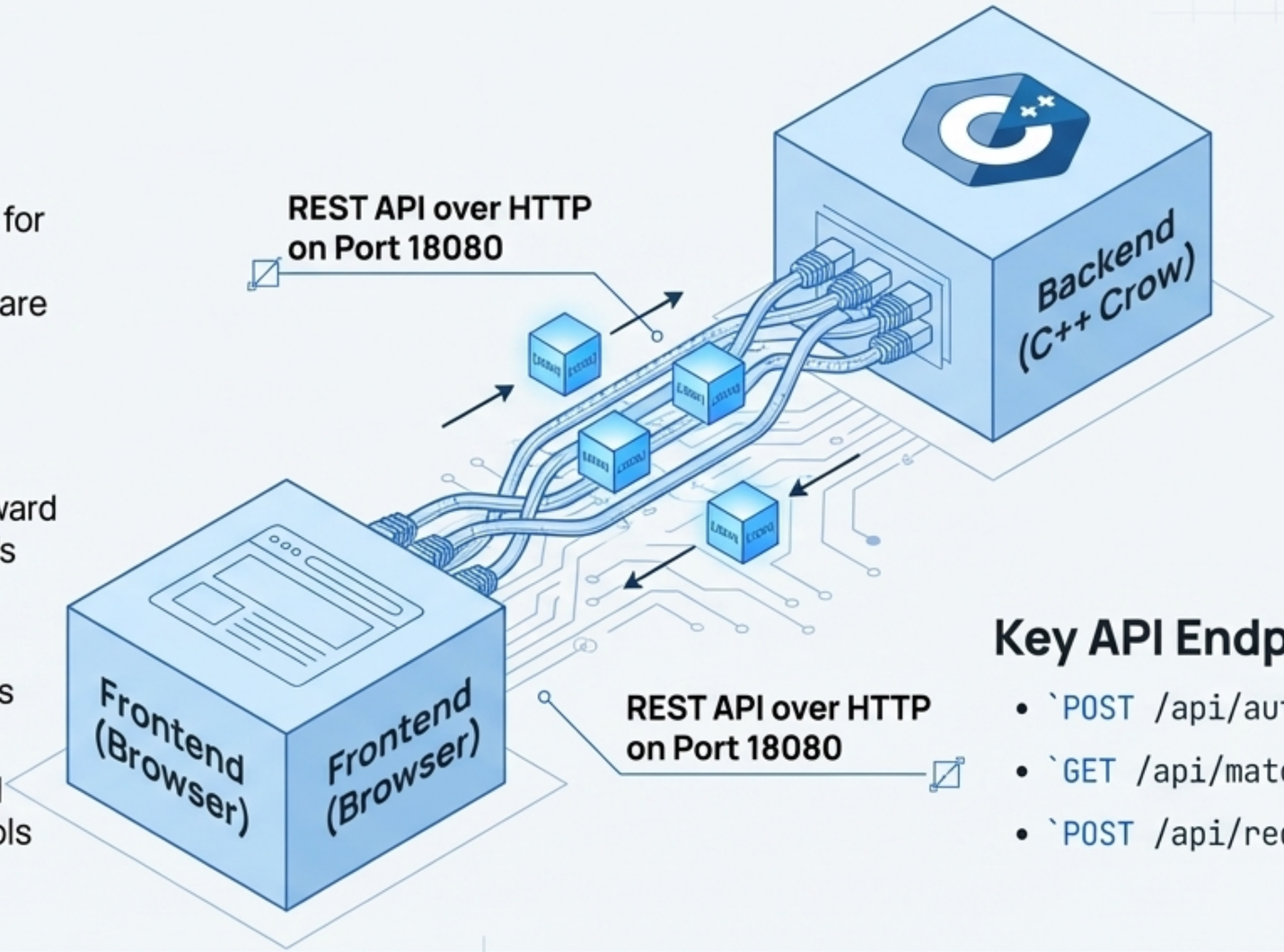


A clear, linear process: User Action -> JavaScript Event Handler -> Asynchronous API Call -> Backend Response Processing.

The Communication Bridge: REST API & C++ Backend

Justification for Technology Choices

- **Why Crow?**: A modern C++17 framework built on ASIO, chosen for its high-performance, low-overhead characteristics, which are essential for our speed requirements.
- **Why REST?:**
  - **Simplicity**: More straightforward than GraphQL for the project's well-defined scope.
  - **Native Support**: Works seamlessly with the browser's native `fetch()` API.
  - **Testability**: Easily tested and debugged using standard tools like Postman and cURL.



Key API Endpoints

- `POST /api/auth/register`
- `GET /api/match/find`
- `POST /api/request/blood`

The Backend Blueprint: Code Structure and Data Flow

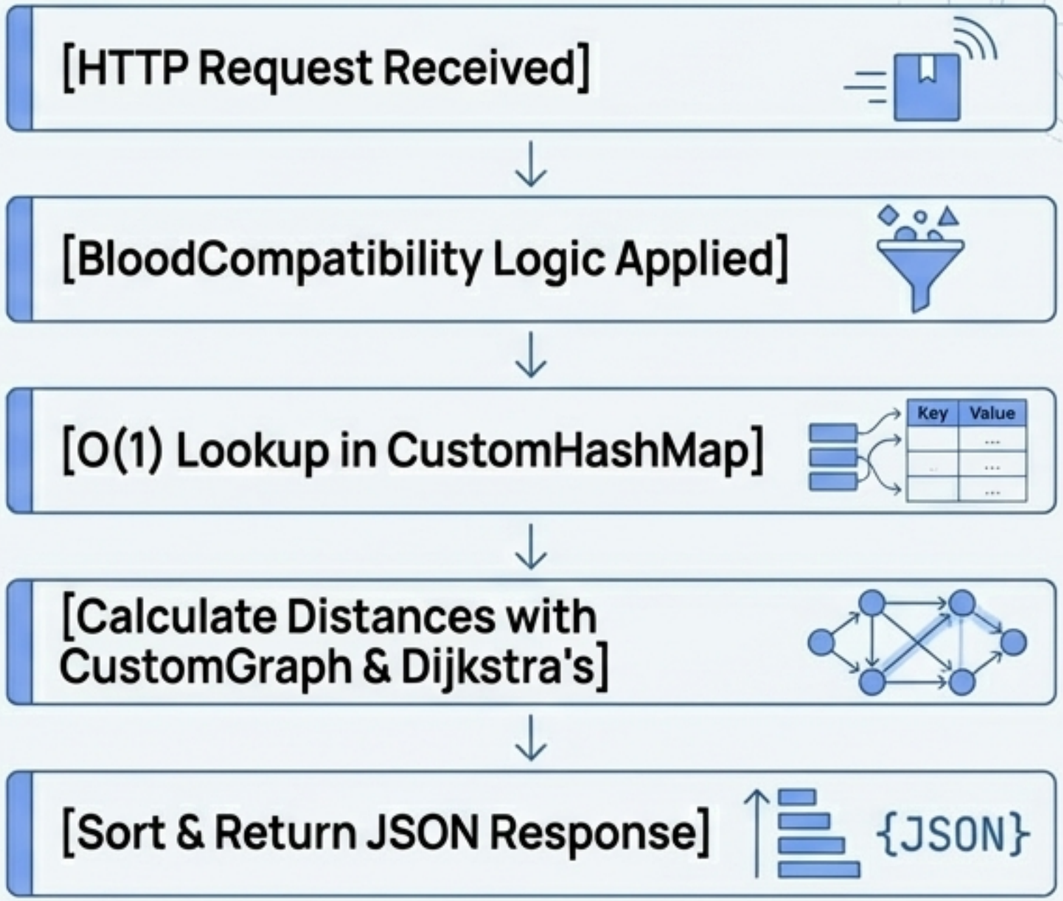
Project Directory Structure

```
src/
├── main.cpp          (API Route Handling)
├── dsa/
│   ├── CustomHashMap
│   ├── CustomLinkedList
│   └── CustomGraph
├── logic/
│   ├── MatchingEngine
│   └── BloodCompatibility
└── models/
    ├── Donor
    └── Recipient
```

Data Flow Example 1: User Registration

1. HTTP Request hits `main.cpp`.
2. A `Donor` object is created.
3. The object is inserted into the `CustomHashMap` in memory and persisted via a `CSVHandler`.

Data Flow Example 2: Finding a Match



Component Specifications: An Overview of Core Algorithms

Key Algorithms Employed

1. **Hash Function & Linear Probing**: For `O(1)` average time donor lookup.
2. **Dijkstra's Algorithm**: For `O((V+E)log V)` shortest path calculation to find the nearest donor.
3. **Breadth-First Search (BFS)*: For `O(V+E)` graph traversal.
4. **Doubly Linked List**: For `O(1)` insertion of transaction history.
5. **Merge Concept**: For `O(n)` history traversal and reporting.

Performance Comparison Table

Algorithm	Use Case	Time Complexity	Space Complexity
Custom HashMap	Donor Lookup by Email	O(1) Avg, O(n) Worst	O(n)
Dijkstra's	Find Nearest Donor	O(E log V)	O(V)
Custom LinkedList	Transaction History	O(1) Insertion	O(n)

Custom Component #1: The Instant-Access Indexing System (HashMap)

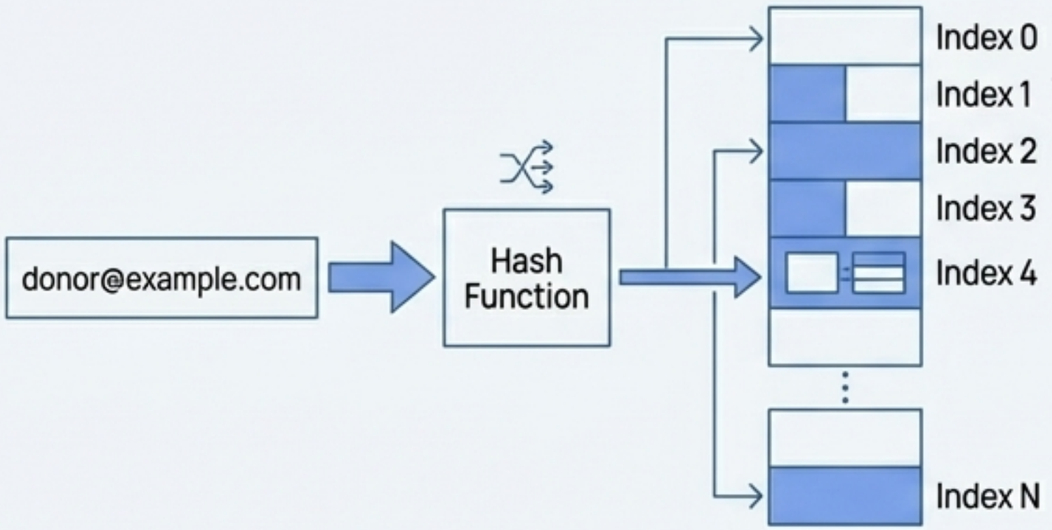
****Engineering Purpose****
To find any donor by their unique email in constant time, `O(1)`. This is critical for fast dashboard lookups and authentication.

****Technical Implementation****

- A hash table implemented with an array of key-value pairs.
- **Collision Handling**: Uses linear probing to resolve hash collisions efficiently.
- **Hash Function**: A custom function converts the string-based email key into an integer array index.

Performance

- **Best/Average Time Complexity**: `O(1)`
- **Worst-Case Time Complexity**: `O(n)` (in the rare case of extreme collisions)




```
// Converts a key into a table index
int hash(K key) {
  return std::hash<K>()(key) % TABLE_SIZE;
}

// Inserts a value using linear probing for collision resolution
void insert(K key, V value) { ... }
```

Custom Component #2: The Immutable Transaction Ledger (LinkedList)

Engineering Purpose
To maintain a chronological history of all donation requests and fulfillments with zero performance overhead on insertion.

Technical Implementation

- A Doubly Linked List, where each node contains data and pointers to both the `previous` and `next` nodes.
- **push_front Operation**: Adds a new donation record to the head of the list in `O(1)` time.
- **pop_front Operation**: Removes an old or invalid record in `O(1)` time.

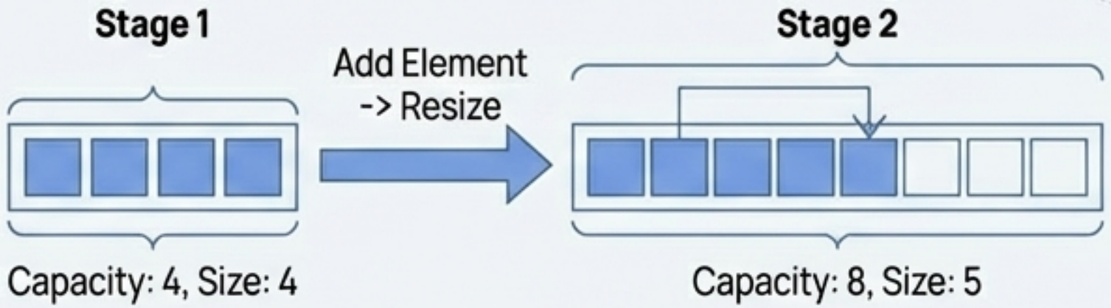
```
template <typename T>
struct Node {
  T data;
  Node* prev;
  Node* next;
};

void push_front(T data) { ... }
```

Why a LinkedList?
This structure avoids the costly process of array shifting or resizing required by a standard vector when inserting elements at the beginning.

Custom Component #3: The Dynamic Results Container (Vector)

Engineering Purpose
To store a list of matching donors when the final count is unknown beforehand. The container must grow dynamically as more matches are found.

Technical Implementation
- A dynamic array that automatically manages its own storage.
- **Growth Strategy**: Implements an exponential growth model. When the internal array is full, its capacity is doubled (e.g., 10 -> 20 -> 40).



Performance
- **Element Access**: `O(1)`
- **push_back (Adding an element)**: `O(1)` amortized time. While resizing is `O(n), it happens so infrequently that the average cost per insertion remains constant.

**Why a Custom Vector?*: It provides the perfect balance of fast access and automatic memory management for returning search results to the frontend.

The Intelligent Logistics Engine: Custom Graph + Dijkstra's Algorithm

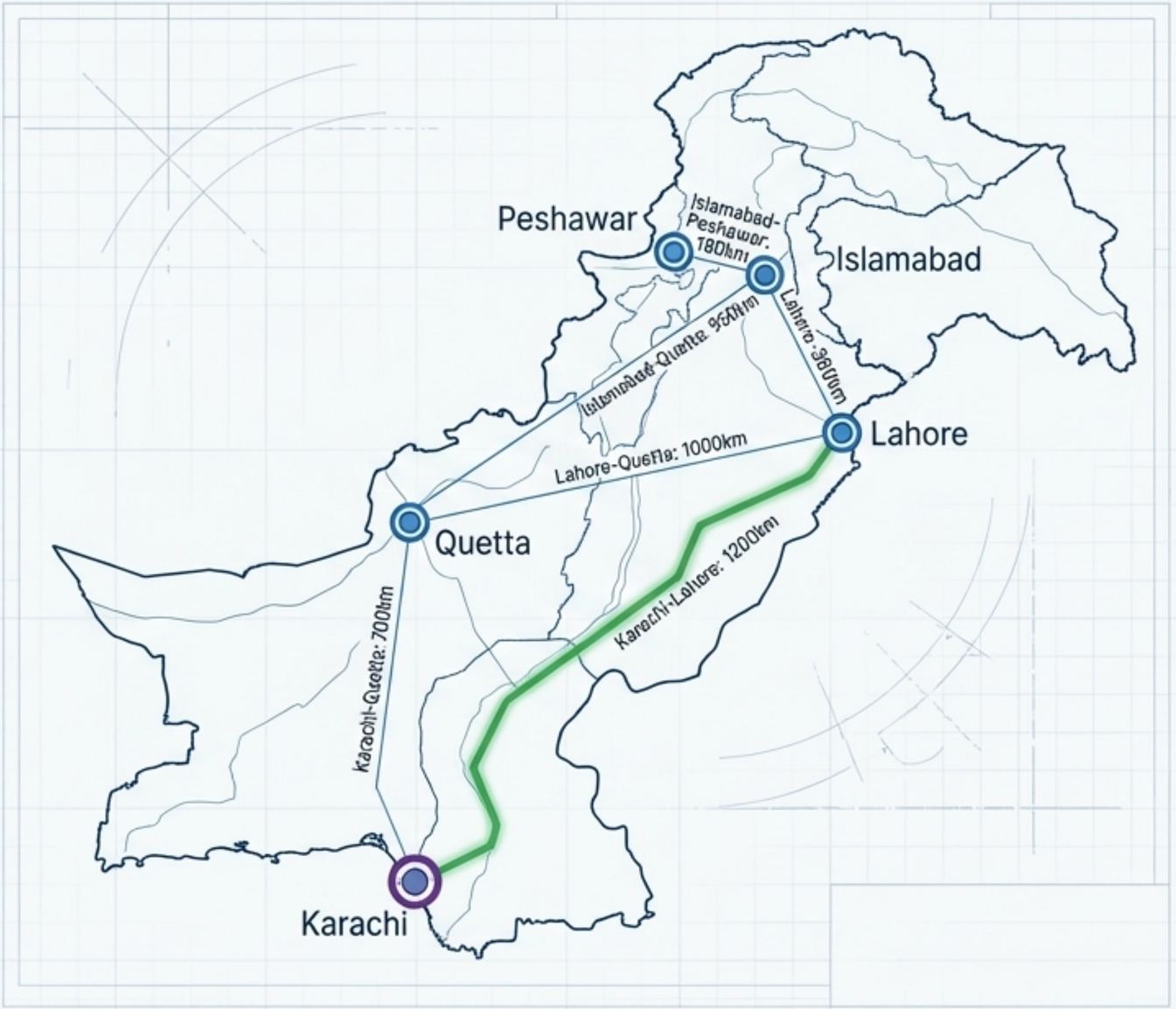
The Core Problem

A patient in Karachi needs O+ blood. We have compatible donors in Lahore, Islamabad, and Quetta. Who is the closest and can respond the fastest?

The Solution: Dijkstra's Shortest Path Algorithm

1. **Model the Network:** Represent cities as nodes (Vertices) and routes between them as weighted edges (Distances in km).
2. **Execute Algorithm:** Starting from the patient's city (**the source**), the algorithm greedily explores the graph.
3. **Priority Queue:** A Min-Heap Priority Queue is used to always select the nearest unvisited city, ensuring optimal pathfinding.
4. **Update Paths:** Iteratively updates the shortest known distance to all other cities from the source.

****Performance**:** `O(E log V)`, highly efficient for this type of problem.



The Efficiency Multiplier: Custom Priority Queue (Min-Heap)

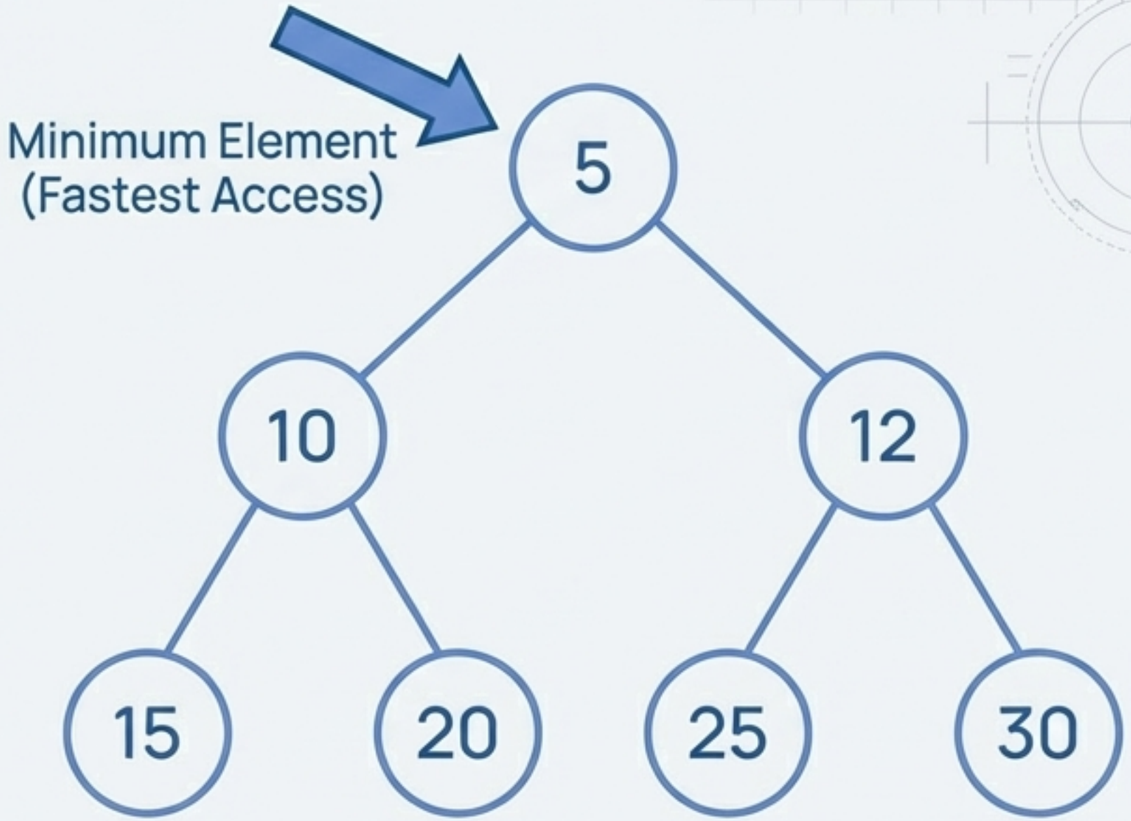
Engineering Purpose
The Priority Queue is the engine inside Dijkstra's algorithm. Its job is to provide the next closest node to visit in `O(log n)` time.

Technical Implementation

- A Min-Heap, which is a binary tree with a specific property: a parent node's value is always less than or equal to its children's values.
- This guarantees that the root of the tree is always the element with the minimum value (the shortest distance).

Performance Impact

- **Push/Pop**: `O(log n)`
- **Get Minimum**: `O(1)`



Why is this critical?: Using a Min-Heap is exponentially faster than scanning an entire list (`O(n)`) at every step of Dijkstra's algorithm to find the minimum distance node. It's a fundamental optimization.

The Integrated System in Motion: A Millisecond Journey

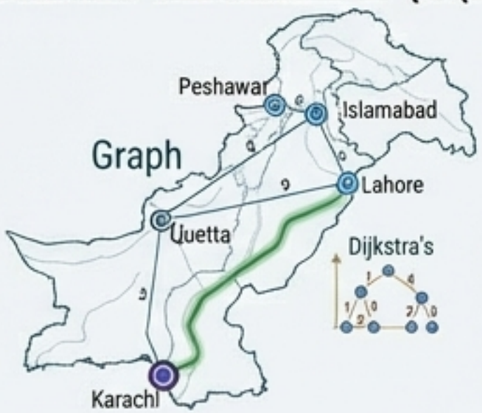
Scenario A patient in Karachi submits an urgent request for O+ blood.

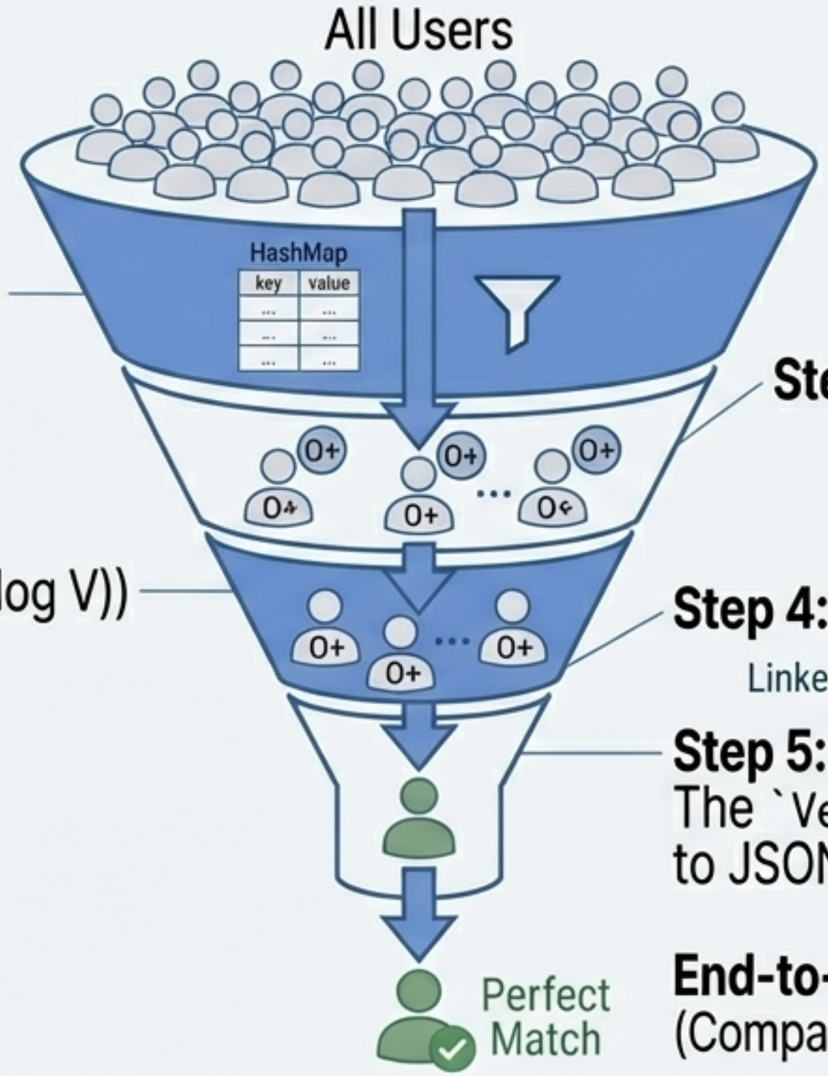
Step 1: Blood Type Filtering (O(1))

HashMap

key	value
...	...
...	...

Step 2: Distance Calculation (O(E log V))





Step 3: Sorting and Packaging

CustomVector

1	2	3	4	5	6	7	8	...

Array

Step 4: Transaction Logging (O(1))

LinkedList

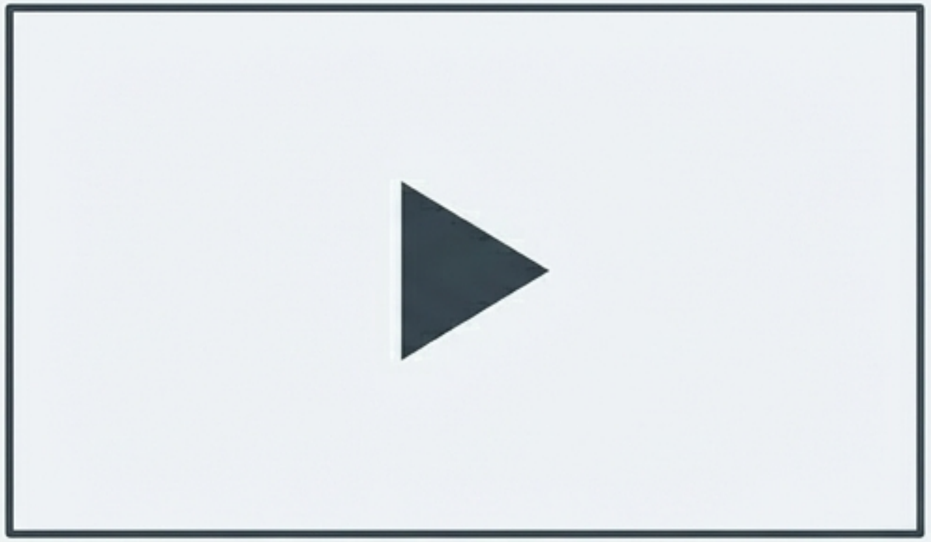
data	data	data
...	...	...

Step 5: Response

The `Vector` of sorted donors is serialized to JSON and sent to the frontend.

End-to-End Performance: ~2 milliseconds.
(Compared to 2000+ ms for a naive approach).

As-Built Verification: System Demonstration



[VIDEO: DS-Final-Demo.mp4]

****Key Features to Observe in the Demo****

✅ **Real-time Matching:** Watch how quickly the system responds to a new blood request.
✅ **Geographic Optimization:** Note how the results are correctly sorted by geographic proximity.
✅ **Instantaneous UI Updates:** See the seamless performance from user click to result display, proving the efficiency of the full stack.

This is not just theory; it's high-performance algorithms in action, solving a real-world problem.