Contact Management Application with Data Structure Benchmarking

1. Introduction

This project implements a **Contact Management Application (mini phonebook)** that allows users to store, update, and manage contact information (Name, Phone, Email). Unlike a simple phonebook, this system is designed to let users **choose different data structures** for storing contacts and directly compare their performance.

It also includes a **benchmarking module** that measures the performance of these data structures in terms of insertion, search, deletion, listing, and memory usage. A web-based frontend provides an interactive dashboard to both manage contacts and visualize benchmark results.

2. Objectives

- Build a modular **contact storage system** supporting multiple data structures.
- Support **CRUD operations** (Create, Read, Update, Delete) on contacts.
- Allow users to **switch between data structures** dynamically.
- Benchmark the performance (time & memory) of each data structure.
- Display both **theoretical complexities** and **practical results** via graphs and tables.

3. System Architecture

3.1 High-Level Design

The project follows a **client-server architecture**:

- **Frontend (UI)** → Runs in the browser, provides forms, buttons, and interactive charts.
- **Backend (Flask)** → Handles requests, manages data structures, and runs benchmarks.
- **Data Structures Module** → Contains separate implementations for Array, Linked List, BST, Hash Map, and Heap.
- **Benchmark Module** → Generates synthetic contacts, runs experiments, measures time and memory usage, and visualizes results.

3.2 Folder Structure

```
project/
├── frontend/
│   ├── index.html    # Main UI
│   ├── style.css     # Styling
│   ├── script.js     # Frontend logic
│
├── backend/
│   ├── app.py        # Flask app (API + serving frontend)
│   ├── benchmark.py  # Benchmark runner
│   ├── data_structures/
│       ├── __init__.py
│       ├── contact.py
│       ├── array_store.py
│       ├── linkedlist_store.py
│       ├── bst_store.py
│       ├── hashmap_store.py
│       ├── heap_store.py
│       ├── complexities.py
│
├── results/
│   ├── benchmark_results.png
│
└── README.md
```

4. Features

4.1 Contact Management

- **Insert Contact** → Adds a new contact.
- **Search Contact** → Finds a contact by name.
- **Delete Contact** → Removes a contact by name.
- **Update Contact (NEW)** → Modify an existing contact's name, phone, or email.
- **List All Contacts** → Displays all stored contacts.

4.2 Data Structure Support

The system supports multiple storage mechanisms:

- **Array (Python list)**
- **Linked List**
- **Binary Search Tree (BST)**
- **Hash Map (Python dict)**
- **Heap (priority queue-based)**

Each data structure implements a common interface:

insert(contact)

search(name)

delete(name)

list_all()

update(name, new_contact) # added for update feature

4.3 Benchmarking

Two types of benchmarking are supported:

Basic Benchmark

- Run for a given number of operations (e.g., 2000).
- Returns **execution time comparison** as a bar chart.

Full Benchmark (NEW)

- Tests across dataset sizes (100, 1000, 10,000).
- Measures **execution time** (insert/search/delete/list).
- Tracks **memory usage** using tracemalloc.
- Includes **theoretical complexities** (O(1), O(n), O(log n) etc.).
- Returns results as **tables + styled charts** in the frontend.

5. Backend Implementation

5.1 API Endpoints

- /insert → Insert a contact.
- /search/<ds>/<name> → Search by name in a chosen data structure.
- /delete/<ds>/<name> → Delete a contact by name.
- /update (NEW) → Update an existing contact.
- /list/<ds> → List all contacts.
- /benchmark?n=5000 → Run a benchmark with given operations.
- /full_benchmark (NEW) → Return complexities + experimental results + memory usage.

5.2 Benchmarking Code Snippet (Core Loop)

start = time.perf_counter()

for c in contacts:

  store.insert(c)

insert_time = time.perf_counter() - start

Memory usage measured with:

tracemalloc.start()

```
# ... run ops ...
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
```

---

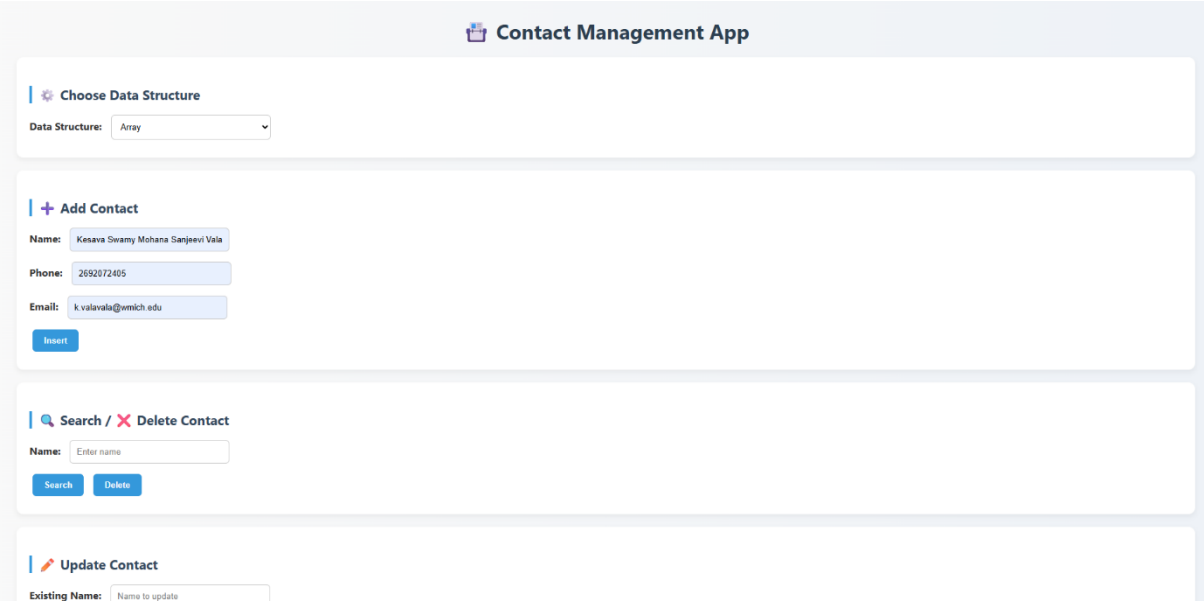
6. Frontend Implementation

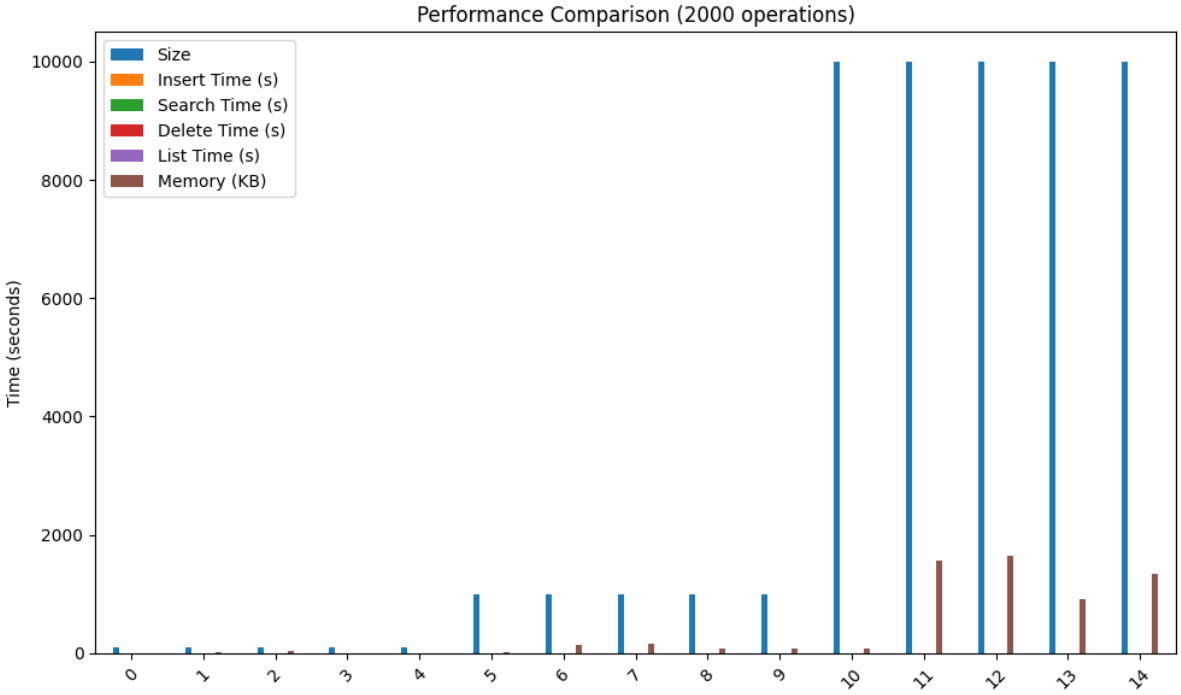
- **index.html** → UI for managing contacts and running benchmarks.
- **script.js** → Handles API calls using fetch.
- **style.css** → Provides clean table/chart styling.

6.1 New UI Features

- **Update Contact Form** for modifying existing entries.
- **Benchmark Dashboard**:
  - Theoretical Complexity Table
  - Benchmark Results Table
  - Interactive Line Charts using Chart.js

7. Benchmark Results





Benchmark Results

Theoretical Complexities

Data Structure	Insert	Search	Delete	List
array	O(1) amortized	O(n)	O(n)	O(n)
bst	O(log n) avg, O(n) worst	O(log n) avg, O(n) worst	O(log n) avg, O(n) worst	O(n)
hashmap	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(n)
heap	O(log n)	O(n)	O(log n) for root, O(n) general	O(n log n) if sorted
linkedlist	O(1)	O(n)	O(n)	O(n)

Experimental Results

Data Structure	Size	Insert Time (s)	Search Time (s)	Delete Time (s)	List Time (s)	Memory (KB)
array	100	0.000040	0.003055	0.002932	0.000002	2.44
linkedlist	100	0.000123	0.000184	0.000162	0.000045	10.19
bst	100	0.000384	0.000098	0.000292	0.000066	36.37
hashmap	100	0.000025	0.000062	0.000059	0.000020	4.78
heap	100	0.000037	0.000189	0.000486	0.000049	4.27
array	1000	0.000158	0.015521	0.026054	0.000044	12.70
linkedlist	1000	0.001316	0.001215	0.001620	0.001282	143.61
bst	1000	0.005294	0.000225	0.000444	0.001457	157.29
hashmap	1000	0.000151	0.000099	0.000087	0.000613	87.67
heap	1000	0.000330	0.001203	0.012319	0.001822	70.70
array	10000	0.000934	0.176985	0.281329	0.000411	96.31
linkedlist	10000	0.009700	0.012542	0.012037	0.012903	1555.28
bst	10000	0.066363	0.000322	0.000750	0.014283	1653.13
hashmap	10000	0.001441	0.000147	0.000107	0.011629	908.30
heap	10000	0.007562	0.012296	0.177156	0.010165	1330.37

7.1 Theoretical Complexities

7.2 Experimental Insights

- **Hash Map** is consistently fastest across all operations.
- **BST** performs well on average, but performance drops when unbalanced.
- **Array & Linked List** are slower in search/delete due to linear scanning.
- **Heap** is efficient for root operations but not general searches.
- **Memory usage** grows differently across structures (Hash Map higher overhead, Linked List node pointers add cost).

---

8. Conclusion

This project demonstrates how **data structure choice impacts performance** in a real-world application (contact management).

Key learnings:

- Theoretical complexities align closely with experimental results.
- Hash Maps provide the best practical performance for phonebook-like tasks.
- Benchmarking reveals trade-offs between memory usage and speed.
- Adding **update** and **full benchmarks** made the system more complete and realistic.

Future work could include:

- Adding **self-balancing BSTs (AVL/Red-Black Trees)**.
- Persisting contacts in a **database**.
- Extending benchmarking to distributed systems.