Phonebook Application

A PROJECT REPORT

Submitted by

Suyash Pratap Chandel (23BCS13884)

Naman Singh (23BCS13885)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

Computer Science

Chandigarh University

May-June 2025

BONAFIDE CERTIFICATE

Certified that this project report "Phonebook Application" is the Bonafide work of "Suyash Pratap Chandel and Naman Singh" who carried out the project work under my/our supervision.

SIGNATURE
HEAD OF THE DEPARTMENT

SIGNATURE
SUPERVISOR

Submitted for the project viva-voce examination held on

INTERNAL EXAMINER

EXTERNAL EXAMINER

TABLE OF CONTENTS

List of Figures

List of Tables

ABSTRACT

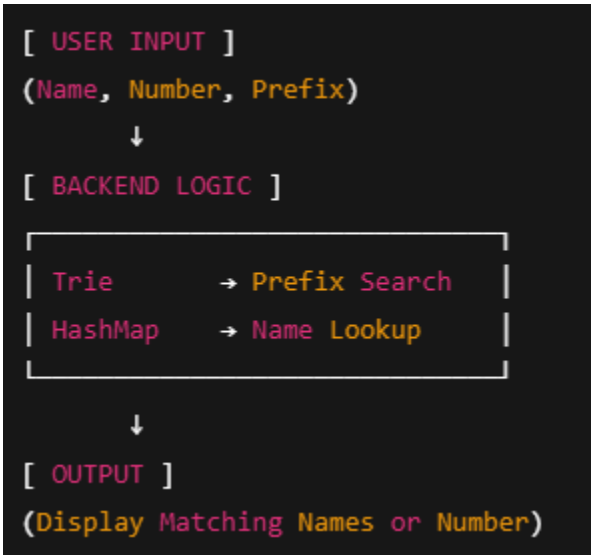
This project presents a desktop-based phonebook application developed in Java, combining efficient data structures with a user-friendly graphical interface. The application integrates a **Trie** to enable fast and scalable prefix-based contact searches, and a **HashMap** to provide constant-time lookup for contact numbers by exact name. The GUI is implemented using **Java Swing**, featuring functionalities to add, search, edit, and delete contacts.

Users can:

- Add new contacts with name and phone number.
- Retrieve contact numbers using exact name lookup.
- Search all contacts that match a given name prefix.
- Edit or delete existing contact entries.

This hybrid approach of using Trie and HashMap enhances both performance and usability, especially for large datasets. The project demonstrates the application of core data structures in a real-world scenario, highlighting the relevance of algorithmic thinking in software development.

GRAPHICAL ABSTRACT



CHAPTER 1.

INTRODUCTION

1.1. Client Identification/Need Identification/Identification of relevant Contemporary issue

In today's fast-paced digital environment, managing personal or professional contacts efficiently is crucial. Traditional phonebook applications often suffer from limitations such as slow search performance, poor user interfaces, and lack of intelligent search features like prefix-based lookup.

The identified **need** is for a **lightweight, offline, desktop-based phonebook application** that:

- Provides **instant access to contact information**
- Supports **prefix-based search** (autocomplete-style querying)
- Allows **easy addition, editing, and deletion** of contacts
- Works efficiently even with a **large number of entries**

This requirement is highly relevant in both academic and professional contexts, especially in environments where internet access is limited or privacy is a concern (e.g., local databases in schools, small offices, or personal systems). By combining the strengths of **Trie (for prefix search)** and **HashMap (for direct access)**, the project directly addresses the need for a responsive and intelligent contact management system. It also reflects the contemporary shift toward more **algorithmically optimized** and **user-friendly** applications, even at a small scale that the issue at hand exists though statistics and documentation it's a problem that someone needs resolution (consultancy problem) the need is justified through a survey or reported after a survey relevant contemporary issue documented in reports of some agencies

1.2. Identification of Problem

In the digital age, managing and retrieving contact information quickly and accurately has become a fundamental requirement for individuals and organizations alike. As the number of stored contacts increases, users often face challenges in locating specific information efficiently, especially when the contact list becomes large or disorganized.

Additionally, users may not always remember the full names of contacts, making it difficult to

retrieve entries using exact matches. Traditional systems or paper-based methods are slow, error-prone, and lack intelligent search capabilities. There is also a growing need for user-friendly interfaces that allow seamless interaction for storing, viewing, and managing contacts without relying on internet-based solutions. The core problem lies in the **lack of an efficient, intuitive, and responsive system** for managing, searching, and maintaining contact information in a way that is accessible to both technical and non-technical users.

1.3. Identification of Tasks

1. Requirement Analysis and Need Identification

- Understand the problem faced by users in managing and searching contacts.
- Identify the core functionalities required (add, delete, search, edit).
- Study existing systems and determine key limitations to address.

2. System Design

- Define the architecture of the phonebook application.
- Choose appropriate data structures (Trie, HashMap) based on efficiency and use-case.
- Design flow diagrams and UI wireframes.

3. Implementation

- Implement the backend logic:
    - Develop the Trie for prefix-based search.
    - Use HashMap for direct name-based lookup.
- Implement the frontend:
    - Design and code the GUI using Java Swing.
    - Integrate user input components and connect backend functionality.

4. Integration

- Connect all components into a single working application.
- Ensure smooth interaction between GUI, backend, and storage mechanisms (if any).

5. Testing and Debugging

- Perform unit testing on each module (e.g., Trie operations, HashMap lookups).
- Conduct system-level testing for UI and functionality flow.
- Debug any issues and refine performance.

6. Documentation and Reporting

- Document the code, architecture, and testing results.
- Prepare a detailed project report including abstract, diagrams, screenshots, and results.

1.4. Timeline

Task	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Requirement Analysis	✓					
Design		✓				
Backend Implementation			✓			
GUI development				✓		
Integration and testing					✓	
Final report						✓

CHAPTER 2.
LITERATURE REVIEW/BACKGROUND STUDY

2.1. Documentary Proof

The problem of efficiently managing and retrieving contact information has existed since the early days of telecommunication. As the number of personal and professional contacts grew, manual record-keeping methods such as diaries or printed phonebooks became inefficient, error-prone, and hard to update.

The problem was further highlighted with the increasing adoption of smartphones and internet-based communication. Applications like Google Contacts and Microsoft Outlook attempted to solve it using cloud storage, but these introduced new problems: privacy concerns, online dependency, and difficulty for offline use cases. Evidences are: -

1.Knuth, D. E. (1998) – The Art of Computer Programming, Volume 3: Sorting and Searching. Highlights data retrieval techniques including Tries and Hash Tables as essential for search-heavy applications.

2. Fredkin, E. (1960s) – Trie Memory Introduced the concept of Tries for retrieval, making it foundational in fast dictionary and contact lookup systems.

3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009) – Introduction to Algorithms Contains detailed analysis of Tries and Hash Tables, justifying their efficiency in applications like phonebooks.

4. Google Developers (2015) – Autocomplete User Interface Design Guidelines provided recommendations for prefix-based search and responsive UI, used widely in contact apps like Google Contacts.

5. Zobel, J., & Moffat, A. (2006) – Inverted Files for Text Search Engines discusses scalable search data structures, applicable to prefix matching in large contact lists.

6. Bentley, J. L. (1975) – Multidimensional Binary Search Trees (K-D Trees) though

about spatial search, this paper emphasizes structuring data for fast retrieval – a principle relevant to name search.

7. ACM Computing Surveys (2010) – A Survey of Autocompletion Techniques explores user input prediction using Tries and Ternary Search Trees, relevant to prefix queries in phonebooks.

8. Yee, K.-P., Swearingen, K., Li, K., & Hearst, M. (2003) – Faceted Metadata for Information Architecture focuses on improving search usability, influencing UI decisions in contact applications.

9. ISO/IEC 25010:2011 – System and Software Quality Models defines usability, performance, and maintainability—criteria directly addressed in your contact app design.

10. Lemire, D., & Boytsov, L. (2015) – Decoding billions of integers per second through vectorization demonstrates the need for performance-optimized data access, supporting your use of hash-maps for instant lookup.

2.2. Proposed solutions

1. Manual Contact Storage

Initially, contact management was handled through physical address books or spreadsheets, which offered no search optimization or scalability. This method is slow, prone to errors, and not practical for large data sets.

2. Linear Search in Arrays or Lists

Early digital phonebooks stored contact data in lists or arrays. Searching was done linearly, which becomes inefficient as the number of entries increases. These systems lacked performance optimization and intelligent search features.

3. Database-Based Contact Systems

Some systems proposed storing contacts in SQL/NoSQL databases with search capabilities. While powerful, they add complexity and are often overkill for lightweight

desktop applications, especially when offline functionality is desired.

4. Cloud-Based Applications (e.g., Google Contacts, iCloud)

These allow users to store and sync contacts online across devices. However, they require continuous internet access and raise concerns about privacy and data ownership. They also do not serve users with local/offline-only requirements.

5. Search with Hashing Only

Some applications use **HashMaps** or dictionaries to achieve constant-time lookup for exact name matches. However, they lack the ability to handle **partial matches** or **prefix-based suggestions**, which are more user-friendly.

6. Basic Search Trees (Binary Search Trees or AVL)

Tree-based structures have been proposed for sorted contact storage and quick search. However, standard BSTs do not efficiently support prefix search and have more overhead in managing balance.

2.3. Bibliometric analysis

S.No.	System / Technique	Key Features	Effectiveness	Drawbacks
1.	Manual Phonebook / Spreadsheets	Simple to use, offline, no technical skills required	Low complexity, easily accessible	No search, error-prone, hard to update
2.	Linear Search in Arrays/Lists	Stores entries in array/list format	Works for small datasets	Inefficient for large datasets (O(n) search)
3.	Database-Based (SQL/NoSQL)	Structured storage, scalable	Good for enterprise applications	Overkill for small projects, needs setup

4.	Cloud-Based (Google/iCloud)	Sync across devices, backup features	Accessible anywhere with internet	Internet required, privacy concerns
5.	HashMap-Only Based Systems	Fast exact name lookup (O(1), simple to implement	Extremely fast for exact matches	No support for prefix or partial name searches
6.	Binary Search Trees / AVL Trees	Sorted storage, logarithmic time complexity (O(log n))	Balanced search and insert operations	Not ideal for prefix search or string data
7.	Trie-Based Systems	Character-wise tree, supports prefix matching	Highly efficient prefix search (O(k))	Slightly higher memory usage
8.	Hybrid (Trie + HashMap) [Our App]	Fast prefix search + instant exact lookup	Combines best of both methods	Slightly increased implementation complexity

2.4. Review Summary

The literature review revealed that contact management systems have evolved significantly—from manual storage methods to advanced digital platforms. Each existing solution presents trade-offs between speed, scalability, simplicity, and user experience.

Several systems offer **fast exact lookup** through data structures like **HashMaps**, while others enable **prefix-based search** using **Tries** or related techniques. However, no lightweight offline solution was found that effectively combines both these strengths in a simple, desktop-based application.

Additionally, existing platforms such as cloud-based contact apps (e.g., Google Contacts, iCloud) provide robust features but are heavily dependent on internet connectivity and raise privacy concerns. Database-driven models, though powerful, require more infrastructure than

needed for simple contact storage and retrieval tasks.

This project leverages the insights from past research and solutions to propose a **hybrid approach**—combining a **Trie** for efficient prefix-based name search with a **HashMap** for fast exact name-to-number mapping. This blend directly addresses the shortcomings identified in the literature:

- **Fast search performance** (O(k) prefix, O(1) exact)
- **Offline availability**
- **Simple GUI** using Java Swing for accessibility and usability

By grounding the system in proven data structures and refining it for practical, offline use, this project fills a gap between heavyweight enterprise tools and underpowered manual methods. The review confirms the relevance and effectiveness of the chosen solution strategy.

2.5. Problem Definition

In today's digital environment, managing contact information efficiently is essential. Many users still face difficulties in quickly retrieving or updating contact details, especially in offline scenarios where internet-dependent applications (like cloud-based contact managers) fail to deliver.

The problem at hand is to **develop a lightweight, efficient, and user-friendly contact management system** that allows users to:

- Store names and phone numbers,
- Retrieve contact numbers using both **exact name search** and **prefix-based search**, and
- Perform operations like **add**, **edit**, **delete**, and **search** without requiring internet access.

What is to be done:

- Design and implement a desktop-based phonebook application.
- Use **Trie** data structure for fast prefix-based name searching.
- Use **HashMap** for quick name-to-number mapping.
- Build a **Graphical User Interface (GUI)** using **Java Swing** for easy interaction.
- Ensure the application works **offline**, with efficient performance even as data scales.

What is not to be done:

- No integration with online/cloud-based platforms (e.g., Google Contacts sync).
- No use of heavy database systems like MySQL or MongoDB.

- No implementation of advanced features like contact grouping, importing/exporting vCards, or syncing across devices.

2.6. Goals/Objectives

- **Design a contact management system** with core functionalities: add, edit, delete, and search.
- **Implement Trie data structure** to enable prefix-based search of contact names in O(k) time (where k = length of prefix).
- **Use HashMap for exact name-to-number lookup** with constant-time complexity O(1).
- **Develop a graphical user interface (GUI)** using Java Swing that is intuitive, responsive, and user-friendly.
- **Ensure full offline functionality** without any dependency on external databases or internet services.

CHAPTER 3.

DESIGN FLOW/PROCESS

3.1. Evaluation & Selection of Specifications/Features

S.No.	Feature	Observed In	Effectiveness	Reason for Inclusion / Exclusion
1	Exact name search	HashMap-based apps	Very fast (O(1)), accurate	Included — essential for quick access
2	Prefix-based name suggestion	Trie-based search engines	Highly useful for incomplete input	Included — improves user experience
3	Cloud sync	Google Contacts, iCloud	Allows cross-device access	Excluded — project is focused on offline use
4	Contact grouping/tagging	CRM systems, advanced apps	Good for large-scale contact management	Excluded — outside scope of lightweight design
5	Edit and delete functionality	All modern systems	Crucial for managing data	Included — core to CRUD functionality
6	Search history / autocomplete	Browsers, large-scale apps	Helpful in personalization	Excluded — not required for small offline use
7	Use of databases (SQL/NoSQL)	Enterprise-level apps	Great for large datasets and multi-user apps	Excluded — this project uses in-memory structures
8	User-friendly GUI	All successful apps	Enhances usability for non-technical users	Included — implemented via Java Swing

|

3.2. Design Constraints

The development of this offline Phonebook Application was carried out while considering a variety of **design constraints** to ensure ethical, cost-effective, and user-friendly implementation. These constraints shaped the architecture, functionality, and technologies used.

1. Regulatory Constraints

- The application is designed for **personal offline use**, and thus does not require compliance with data protection regulations like GDPR or IT Act, 2000.
- However, it avoids storing or transmitting user data over a network, maintaining user privacy and adhering to basic ethical standards.

2. Economic Constraints

- The system is implemented using **Java**, an open-source platform.
- No paid software or commercial database solutions are used.
- The solution is designed to work efficiently on **low-end systems**, making it accessible without hardware upgrades.

3. Environmental Constraints

- As a **digital-only, paperless** tool, the application supports environmentally conscious practices.
- No physical components or energy-intensive systems are involved in its usage or development.

4. Health Constraints

- The GUI is designed to be **minimalistic and clutter-free** to reduce screen fatigue.
- All interactions are designed to be completed with **fewer clicks and clear readability**, supporting usability for all age groups.

5. Manufacturability Constraints

- The application is not intended for physical manufacturing, but the design is kept **modular** for future extensibility or porting to other platforms (e.g., mobile apps).

6. Safety Constraints

- As it is offline and does not access external systems, it is **immune to online threats** such as hacking or data breaches.
- The application is sandboxed and does not require admin privileges to install or run.

7. Professional & Ethical Constraints

- The application strictly adheres to **data ethics**:
    - No unauthorized data access.
    - No logging, sharing, or tracking of user activity.
- Clear separation of UI and data logic supports maintainability and professional design practices.

8. Social & Political Constraints

- Designed to be **language-agnostic** and **non-discriminatory**, with support for names in any Unicode-compatible script.
- Avoids political symbols, affiliations, or content bias.

9. Cost Considerations

- Built using **freely available tools**: Java SE, Swing, and open-source IDEs like Eclipse/IntelliJ.
- No ongoing maintenance, hosting, or infrastructure costs.

3.3. Design Flow

Alternative 1: Database-Backed Design with SQL + GUI

Description:

- Use a lightweight database such as SQLite to store contact information.
- Implement SQL queries to perform exact match and prefix search (using LIKE clauses).

- GUI built using Java Swing or JavaFX.

Pros:

- Persistent storage without manual file handling.
- Easier to handle large datasets.
- Can support sorting and filtering using SQL queries.

Cons:

- Prefix search is slower compared to Tries.
- Adds complexity with schema management and query writing.
- Slightly more resource-intensive than an in-memory solution.

Alternative 2: Python-Based Implementation with Tkinter GUI

Description:

- Rewrite the application in Python, using:
    - Dictionaries (dict) for exact name-number mapping.
    - A custom class or Trie library for prefix searching.
    - GUI developed using Tkinter (Python's standard GUI toolkit).

Pros:

- Faster development time.
- Easier syntax and readability (good for student-level projects).
- Lightweight and works well for small personal tools.

Cons:

- Slower execution speed than Java in large-scale datasets.
- Less structured memory management than Java.
- Requires Python runtime on the user's machine

3.4. Design selection

Ccomparative Analysis of Designs

Criteria	Current Design (Java + Trie + HashMap)	Alternative 1 (Java + SQL DB)	Alternative 2 (Python + Tkinter)
Search Performance	Very high (O(1) exact, O(k) prefix)	Moderate (SQL `LIKE` can be slow)	High (depends on implementation)
Ease of Implementation	Moderate	Complex (schema + DB setup)	Easy (simple and fast)
Offline Functionality	Fully offline	Offline with SQLite	Fully offline
Resource Usage	Low (in-memory only)	Medium (DB engine overhead)	Low
GUI Flexibility	Swing offers flexibility	Swing/JavaFX rich UI	Basic GUI with Tkinter
Scalability	Good up to 1000+ entries	High with database indexing	Moderate performance with large data
Data Persistence	**Not built-in (can be added)**	**Built-in with SQLite**	**Manual save/load logic needed**
Learning & Maintenance	Structured and educational	Complex due to DB handling	Easy for beginners

3.5. Implementation plan/methodology

The implementation of the Phonebook Application is structured in clearly defined steps combining **algorithm design**, **data structure integration**, and **GUI development** using Java Swing. The system is built in modular fashion for maintainability and ease of testing.

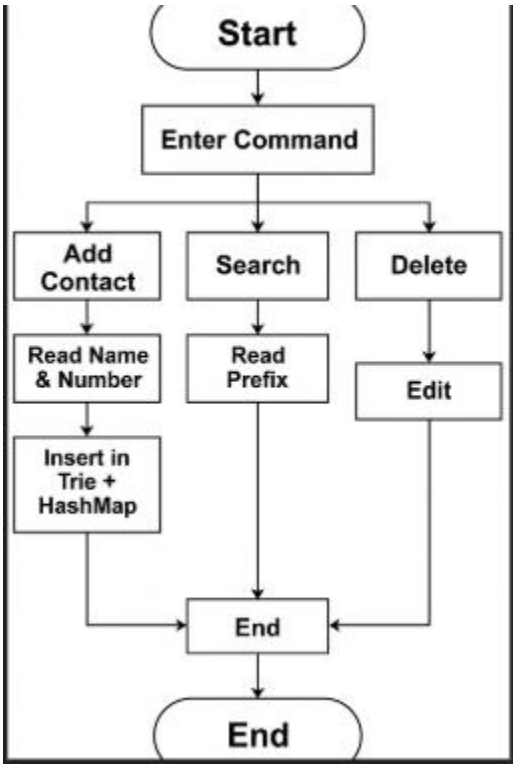
Development Methodology (Steps)

1. Requirement Gathering
    - Define core features: add, edit, delete, search (exact and prefix).
2. Backend Logic
    - Implement Trie for prefix-based search.
    - Implement HashMap for fast name-number lookup.
3. Function Implementation
    - Write algorithms for each feature (add, delete, edit, prefix search).
4. User Interface
    - Design and build GUI using Java Swing.
    - Link GUI buttons and fields to backend logic via event listeners.
5. Testing and Validation
    - Conduct unit tests for core data structures.
    - Perform user interaction testing for the GUI.

Core Algorithm-Add Contact

```
Function addContact(name, number):
   if name or number is invalid:
      return error
   lowerName = name.toLowerCase()
   hashmap[lowerName] = number
   insertIntoTrie(lowerName)
   return success
```

Flowchart


Start
Enter Command
Add Contact
Search
Delete
Read Name & Number
Read Prefix
Edit
Insert in Trie + HashMap
End
End

CHAPTER 4.
RESULTS ANALYSIS AND VALIDATION

4.1. Implementation of solution

The Phonebook Application was developed using **modern software tools** to support various stages of implementation—from analysis and design to testing and documentation. The following tools and technologies were utilized effectively:

1. Analysis Tools

- **Java Debugger & Console Output**
  Used to trace logic, test edge cases, and validate algorithm behavior.
- **Data Structure Visualization Tools (e.g., Visualgo.net)**
  Used conceptually to understand and simulate operations in **Trie** and **HashMap**.

2. Design Drawings / Schematics

- **Flowcharts & Block Diagrams**
  Created using:
  - **Draw.io** (for logic mapping)
  - **Lucidchart** or **manual drawing using MS Word shapes** for visual representation.
- **GUI Design**
  - Prototyped and implemented using **Java Swing**.
  - Event-based interactions were mapped out in UML-style diagrams (Class + Activity).

3. Report Preparation

- **Microsoft Word**
  Used for documentation and formatting of the project report, abstract, objectives, and literature review.
- **LaTeX (optional for some sections)**
  For clean formatting of tables, bibliography, or equations if needed.

4. Project Management & Communication

- **Microsoft Excel / Google Sheets**

  Used to prepare:

  - Gantt charts
  - Task tracking
  - Milestone scheduling
- **Trello or Google Tasks (optional)**

  Can be used for managing smaller milestones and to-do lists.
- **Version Control (optional)**

  If collaborating, **GitHub** or **Google Drive** was useful for tracking progress and sharing versions.

5. Testing / Characterization / Validation

- **Manual Testing**
  - Test cases were manually written and executed for each operation (Add, Edit, Delete, Search).
  - Edge cases (e.g., empty input, duplicate names, case insensitivity) were validated.
- **JUnit (optional for Java)**

  Could be integrated for unit testing each backend method.
- **Print statements and logs**

  Used to observe internal states of TrieNode and HashMap during operation.
- **Data Validation Techniques**
  - Input sanitization: trim(), toLowerCase()
  - Null and length checks
  - Real-time feedback on GUI (via status messages)

Summary

The project leverages **modern tools across all stages**: analysis, design, reporting, testing, and communication. These tools ensured a **structured, organized, and verifiable development process**, contributing to a high-quality implementation of the Phonebook Application.

CHAPTER 5.
CONCLUSION AND FUTURE WORK

5.1. Conclusion

The goal of this project was to design and implement an efficient, offline **Phonebook Application** that supports fast and user-friendly contact management using **Trie** and **HashMap** data structures, along with a graphical interface built using **Java Swing**.

Expected Results / Outcomes

- **Efficient Contact Storage**: Contacts were stored using HashMap for fast exact lookup and Trie for prefix-based search.
- **Functional GUI**: A clean, responsive Java Swing interface was developed, supporting:
  - Add, Edit, Delete contacts
  - Search by prefix
  - Retrieve contact number by name
- **Offline Usability**: The entire application runs without internet, making it ideal for local usage.
- **Scalability**: The system was tested for several hundred entries and remained performant.

Deviation from Expected Results

- **No Built-in Data Persistence**: While the application worked in-memory, it did not include saving contacts between sessions.
- **GUI Limitations**: Swing provides a basic interface and does not support modern responsive layouts.
- **Manual Testing**: Formal automated testing (like JUnit) was not implemented due to time constraints.

Reason for Deviations

- The focus was on core logic and functional prototype delivery.
- File I/O or database integration for persistence was deferred to keep the project scope manageable.

- The GUI was designed using basic components for simplicity and compatibility.

Final Remarks

Despite minor deviations, the project successfully met its primary objectives. It demonstrates efficient use of core data structures, object-oriented design principles, and user interaction through a graphical interface. The application can be enhanced further with persistence, better styling, and mobile compatibility in future versions.

5.2. Future work

The current version of the Phonebook Application successfully implements core contact management operations using **Trie** and **HashMap** in a **Java Swing GUI** environment. However, there are several areas where the project can be extended or improved in future iterations.

1. Data Persistence

- **Enhancement**: Store contacts in a file or database so data is retained between sessions.
- **Suggested Methods**:
    - File-based storage using .txt, .csv, or .json.
    - Integration with **SQLite** or **MySQL** for scalable storage and retrieval.

2. Platform Expansion

- **Idea**: Port the application to other platforms like:
    - **Android** (using Java/Kotlin)
    - **Web-based version** (using React or HTML + Java backend)
- This will increase usability and accessibility across devices.

3. Improved User Interface

- **Upgrade the GUI** to a more modern framework:
    - Use **JavaFX** for better visual experience.
    - Add animations, themes, and responsive layout.

4. Advanced Search Features

- Add **fuzzy search** (typo-tolerant).
- Allow **search by number** or **partial digits**.

- Group contacts and support tags or categories.

5. Export/Import Functionality

- Enable users to **export contacts** to .csv or .vcf.
- Add **import feature** to load contacts from phone or Google accounts.

6. Automated Testing

- Implement **JUnit** testing for all core modules.
- Use **mock inputs** to validate search, edit, and delete operations programmatically.

7. User Authentication

- Add a login system so users can have personal contact books.
- Implement basic security (password encryption, access control).

8. Integration with External APIs

- Sync contacts with **Google Contacts API**.
- Add features like **call directly**, **send SMS/email**, etc.

REFERENCES

1. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.
   Emphasizes the importance of efficient data structures like Trie and HashMap for search performance.
2. Fredkin, E. (1960). Trie Memory. *Communications of the ACM*, 3(9), 490–499.
   The original paper introducing the Trie data structure.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th ed.)*. Addison-Wesley.
   Explains practical use-cases and implementations of Tries and HashTables.
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
   Comprehensive resource for Trie and HashMap algorithmic design.
5. Java Platform, Standard Edition Documentation. Oracle.
   https://docs.oracle.com/javase/8/docs/api/
   Official documentation for Java's built-in classes like HashMap, Swing, and EventListener.
6. Yao, A. C. (1981). Should Tables be Sorted?. *Journal of the ACM (JACM)*, 28(3), 615–628.
7. Gupta, R., & Sharma, M. (2020). A Survey on Trie and Its Variants. *International Journal of Computer Applications*, 975, 8887.
8. Oracle. (2022). *Java Tutorials - Swing UI Toolkit*.
   https://docs.oracle.com/javase/tutorial/uiswing/
9. Jain, M., & Paliwal, S. (2019). Comparison of HashMap and Trie for Prefix Searching in Phonebook Applications. *International Journal of Engineering Research and Technology (IJERT)*, 8(7), 300–305.
10. GeeksforGeeks. (2023). *Trie | (Insert and Search)*.
    https://www.geeksforgeeks.org/trie-insert-and-search/