DSA Internal Project

Coordinate-Consolidator

A Contact Management System
Using Python, CSV, and Streamlit

Data Structures and Algorithms
Internal Assessment

Submitted By:

Avishi Razdan	MIS: 612309041
Bhumi Dave	MIS: 612309010
Sakshi Babar	MIS: 612309008
Ashlesha Patil	MIS: 612309039

Contents

1. Data Structure Selection and Justification

Chosen Data Structure: List of Dictionaries

Justification:

1. **Dictionary for Field-Value Mapping:**
   - Each contact is represented as a dictionary with key-value pairs (Name, Phone1, Phone2, Email, etc.)
   - Provides O(1) average-case access to individual fields by name
   - Makes code readable and maintainable as fields are accessed by meaningful names rather than indices
2. **List for Multiple Records:**
   - A list contains multiple contact dictionaries, maintaining insertion order
   - Allows dynamic growth as contacts are added without pre-allocating size
   - Supports iteration for operations like view, search, update, and delete
3. **CSV Compatibility:**
   - Python's `csv.DictReader` and `csv.DictWriter` work seamlessly with dictionaries
   - Dictionary keys map directly to CSV column headers
   - Enables easy persistence and data exchange in a human-readable format

The list of dictionaries structure combines the benefits of structured records (dictionaries) with ordered collection management (lists), making it ideal for CRUD operations on tabular data stored in CSV format.

2. Implementation of CRUD Functions

A. Backend Implementation

1. Add Contact Function:

```python
def add_contact(contact_data):
    with open(csv_file, 'a', newline="", encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=FIELDS)
        writer.writerow(contact_data)
```

Working:

- Opens CSV file in append mode ('a') to add new record without overwriting
- Uses `DictWriter` to write the contact dictionary as a new row
- Maintains data structure integrity by using predefined FIELDS list

2. View Contact Function:

```python
def view_contact():
    contacts = []
    with open(csv_file, 'r', encoding="utf-8") as f:
        reader = csv.DictReader(f)
        for row in reader:
            contacts.append(row)
    return contacts
```

Working:

- Reads entire CSV file and converts each row into a dictionary
- Returns a list of dictionaries representing all contacts
- Time Complexity: O(n) where n is number of contacts

3. Update Contact Function:

```python
def update_contact(name, new_data):
    rows = []
    updated = False
    with open(csv_file, "r", encoding="utf-8") as f:
        reader = csv.DictReader(f)
        for row in reader:
            if row["Name"].lower() == name.lower():
                rows.append(new_data)
                updated = True
            else:
                rows.append(row)

    if updated:
        with open(csv_file, "w", encoding="utf-8", newline="") as
            f:
```

```
            writer = csv.DictWriter(f, fieldnames=FIELDS)
            writer.writeheader()
            writer.writerows(rows)
        return True
    return False
```

Working:

- Reads all contacts into memory, replaces matching contact with new data
- Case-insensitive name matching for user convenience
- Rewrites entire CSV file with updated data
- Returns boolean indicating success/failure

4. Delete Contact Function:

```
def delete_contact(name):
    rows = []
    deleted = False
    with open(csv_file, "r", encoding="utf-8") as f:
        reader = csv.DictReader(f)
        for row in reader:
            if row['Name'].strip().lower() != name.strip().lower
               ():
                rows.append(row)
            else:
                deleted = True

    if deleted:
        with open(csv_file, "w", encoding="utf-8", newline="") as
            f:
            writer = csv.DictWriter(f, fieldnames=FIELDS)
            writer.writeheader()
            writer.writerows(rows)
    return deleted
```

Working:

- Reads all contacts except the one to be deleted
- Uses strip() to handle whitespace, case-insensitive matching
- Rewrites CSV excluding deleted contact
- Returns boolean indicating deletion status

B. Frontend Integration with Streamlit

1. Add Contact Interface:

- Uses `st.text_input()` for each field in FIELDS list
- Validates that Name field is not empty before calling backend function
- Displays success/error messages using `st.success()` and `st.error()`
- Creates contact dictionary from user inputs and passes to `add_contact()`

2. View Contact Interface:

- Calls `view_contact()` to retrieve list of dictionaries
- Converts to Pandas DataFrame for tabular display: `pd.DataFrame(data)`
- Uses `st.dataframe()` for interactive table rendering
- Handles empty contact list with informative message

3. Update Contact Interface:

- Two-step process: Fetch contact data, then display pre-filled form
- Uses `search_contact()` to retrieve existing data
- Pre-populates text inputs with current values: `st.text_input(field, old_data.get(field, ""))`
- Calls `update_contact()` with name identifier and new data dictionary

4. Delete Contact Interface:

- Simple interface with name input and confirmation button
- Calls `delete_contact()` and displays result
- Provides clear feedback on success or failure

5. Additional Frontend Features:

- **CSV Download:** Uses `st.download_button()` with DataFrame's `to_csv()` method
- **Menu Navigation:** Sidebar selectbox for easy feature access
- **Data Visualization:** Pandas DataFrame integration makes data presentation clean and professional

The frontend acts as a user-friendly layer over backend functions, converting user interactions into function calls and displaying results in an intuitive format. Streamlit's widgets map naturally to CRUD operations, while Pandas DataFrame bridges the gap between list-of-dictionaries structure and tabular UI display.

3. Search Algorithm Selection and Justification

Chosen Algorithm: Linear Search (Sequential Search)

Justification :

1. **Unsorted Data Structure:**
   - CSV file maintains insertion order without any sorting
   - Contact records are not organized by any particular field
   - Binary search requires sorted data (O(log n)), which would necessitate sorting overhead (O(n log n))
   - For unsorted data, linear search is the natural and most efficient choice
2. **Small Dataset Characteristics:**
   - Contact managers typically handle hundreds to low thousands of records
   - Linear search time complexity O(n) is acceptable for small n
   - The simplicity of implementation outweighs the performance benefits of complex search algorithms for this scale
   - No additional data structure overhead (like hash tables or trees) required
3. **Multi-field Search Requirement:**
   - The search function checks query against ALL fields (Name, Phone, Email, etc.)
   - This flexible, keyword-based search would be difficult with indexed structures optimized for single-field lookups
   - Linear search naturally supports checking multiple conditions per iteration
   - Implementation: `any(query.lower() in str(value).lower() for value in row.values())`
4. **CSV File-based Storage:**
   - Reading from CSV requires sequential access anyway
   - No in-memory index structure is maintained between operations
   - Each search operation reads the file from start to finish
   - Linear search aligns with the sequential nature of file I/O

Linear search is optimal for this application because the data is unsorted, the dataset is small, searches span multiple fields, and the file-based storage model naturally supports sequential access. Alternative algorithms like binary search or hash-based lookups would require additional overhead without providing meaningful performance benefits.

4. Keyword-based Search Implementation

Implementation:

```python
def search_contact(query):
    results = []
    with open(csv_file, "r", encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            # Check if query appears in ANY field (case-
               insensitive)
            if any(query.lower() in str(value).lower() for value
               in row.values()):
                results.append(row)
    return results
```

Working Mechanism:

1. **File Reading and Iteration:**
    - Opens CSV file in read mode with UTF-8 encoding for international character support
    - Uses `csv.DictReader` to parse each row as a dictionary
    - Iterates through all contact records sequentially
2. **Multi-field Keyword Matching:**
    - `row.values()` extracts all field values from the contact dictionary
    - `str(value).lower()` converts each value to lowercase string for case-insensitive comparison
    - `query.lower() in str(value).lower()` checks if query is a substring of any field
    - `any()` returns True if query matches at least one field
3. **Result Accumulation:**
    - Matching contacts are appended to the results list
    - Returns all matching records as a list of dictionaries
    - Maintains original dictionary structure for frontend compatibility
4. **Search Flexibility:**
    - Partial matching: searching "john" will match "John Doe", "john@email.com", etc.
    - Case-insensitive: "SMITH" matches "Smith" or "smith"
    - Works across all fields: phone numbers, emails, social media handles, names
    - Example: searching "gmail" returns all contacts with Gmail addresses

5. Frontend Integration

```python
import streamlit as st
import pandas as pd
from dsa import add_contact , view_contact , search_contact ,
   update_contact , delete_contact , FIELDS

st.title("Contact Manager")

# Sidebar Menu
menu = ["Add", "View", "Search", "Update", "Delete"]
choice = st.sidebar.selectbox("Menu", menu)

if choice == "Add":
    st.subheader("Add New Contact")
    contact_data = {}
    for field in FIELDS:
        if field == "Name":
            contact_data[field] = st.text_input(field,
               placeholder="Enter name")
        elif "Phone" in field or field == "WhatsApp":
            contact_data[field] = st.text_input(field,
               placeholder="Enter phone number")
        elif field == "Email":
            contact_data[field] = st.text_input(field,
               placeholder="Enter email")
        else:
            contact_data[field] = st.text_input(field,
               placeholder=f"Enter {field}")

    if st.button("Save Contact"):
        if contact_data["Name"].strip() != "":
            add_contact(contact_data)
            st.success(f"Contact {contact_data['Name']} added
               successfully!")
        else:
            st.error("Name cannot be empty.")

elif choice == "View":
    st.subheader("All Contacts")
    data = view_contact()
    if data:
        df = pd.DataFrame(data)
        st.dataframe(df)
    else:
        st.info("No contacts found.")

elif choice == "Search":
    st.subheader("Search Contact")
    name = st.text_input("Enter name to search")
    if st.button("Search"):
        results = search_contact(name)
        if results:
            st.write(pd.DataFrame(results))
        else:
            st.warning("No contact found with that name.")

elif choice == "Update":
    st.subheader("Update Contact")
    name = st.text_input("Enter name of contact to update")
    if st.button("Fetch Contact"):
        results = search_contact(name)
        if results:
            old_data = results[0]
            new_data = {}
            for field in FIELDS:
                new_data[field] = st.text_input(field, old_data.get(field, ""))

            if st.button("Update Contact"):
                success = update_contact(name, new_data)
                if success:
                    st.success(f"Contact {name} updated successfully!")
                else:
                    st.error("Update failed.")
        else:
            st.warning("No contact found with that name.")

elif choice == "Delete":
    st.subheader("Delete Contact")
    name = st.text_input("Enter name of contact to delete")
    if st.button("Delete"):
        deleted = delete_contact(name)
        if deleted:
            st.success(f"Contact {name} deleted successfully!")
        else:
            st.error("Contact not found.")

data = view_contact()
if data:
    df = pd.DataFrame(data)
    st.download_button(
        label="Download Contacts as CSV",
        data=df.to_csv(index=False).encode("utf-8"),
        file_name="contacts.csv",
        mime="text/csv",
    )
```

6. Conclusion

The Contact-Consolidator application successfully demonstrates the practical implementation of fundamental data structures and algorithms in solving real-world problems. By choosing a list of dictionaries as the core data structure, we achieved an optimal balance between simplicity, functionality, and CSV compatibility, making the system both efficient and maintainable.

The implementation of CRUD (Create, Read, Update, Delete) operations showcases how backend logic can be cleanly separated from frontend presentation, with Python handling data manipulation and Streamlit providing an intuitive user interface. The linear search algorithm, though theoretically less efficient than advanced search techniques, proved to be the most practical choice given the application's constraints of unsorted data, small dataset size, and multi-field search requirements.

The integration of Streamlit and Pandas DataFrame bridges the gap between backend data structures and user-friendly visualization, enabling features like interactive tables and CSV export with minimal code complexity. This project highlights that algorithmic efficiency must be evaluated in context the "best" algorithm is not always the fastest in theory, but the one most suited to the specific use case.

Overall, the Contact-Consolidator serves as a comprehensive example of how fundamental DSA concepts data structure selection, CRUD operations, and search algorithms come together to create a functional, user-centric application. The modular architecture ensures easy extensibility for future enhancements such as advanced search filters, data validation, or cloud storage integration, demonstrating the importance of thoughtful design in software development.