<u>DATA STRUCTURE AND ALGORITHM</u> 1

<u>SECTION B</u>

<u>The source code</u>

```
// Import the necessary classes from the Java Swing and AWT libraries to build the GUI

import javax.swing.*;  // Provides classes for creating a graphical interface, like buttons, text fields, etc.

import java.awt.*;     // Provides classes for windowing and layouts, like positioning components

import java.awt.event.ActionEvent;  // Provides classes to capture actions, like button clicks

import java.awt.event.ActionListener; // Interface for receiving action events (e.g., a button click)

import java.awt.event.MouseAdapter;   // Adapter class for receiving mouse events (e.g., clicking)

import java.awt.event.MouseEvent;     // Provides information about a mouse event (e.g., which button was clicked)

import java.util.ArrayList;  // Provides a dynamic array for storing multiple items, like contacts

import java.util.Comparator; // Provides functionality to compare objects for sorting (e.g., sort contacts by name)
```

```
// Define the main class for the PhoneBook application,
which extends JFrame, making it a GUI window
public class PhoneBookApp extends JFrame {
   // An ArrayList is used to store a list of contacts,
which can grow and shrink as needed
   private ArrayList<Contact> phonebook = new
ArrayList<>();

   // A JTextArea where information about contacts will
be displayed to the user
   private JTextArea displayArea;

   // JTextFields are text input boxes where users can
type in contact information
   private JTextField nameField, phoneField, emailField,
searchField;

   // JLabel for displaying a background image behind all
components
   private JLabel backgroundLabel;

   // JLayeredPane allows multiple layers in a single
container; we use it to manage components over a
background image
```

```
  private JLayeredPane layeredPane;

  // A temporary variable to hold a contact that the user
wants to update
  private Contact contactToUpdate = null;

  // The Contact class is a blueprint for creating
individual contact objects with name, phone, and
email
  public static class Contact {
    // Each contact has a name, phone number, and
email, stored in these three variables
    String name, phoneNumber, email;

    // Constructor that sets the name, phone number,
and email of a new contact when it's created
    public Contact(String name, String phoneNumber,
String email) {
      this.name = name;
      this.phoneNumber = phoneNumber;
      this.email = email;
    }
    // A method to get the name of the contact
    public String getName() {
      return name;
    }

    // A method to update the name of the contact
    public void setName(String name) {
      this.name = name;
    }

    // A method to update the phone number of the
contact
    public void setPhoneNumber(String phoneNumber)
{
      this.phoneNumber = phoneNumber;
    }

    // A method to update the email of the contact
    public void setEmail(String email) {
      this.email = email;
    }
```

```
        // This method returns a formatted string with the
contact's details
        public String toString() {
            return "Name: " + name + ", Phone: " +
phoneNumber + ", Email: " + email;
        }
    }

    // The main constructor for setting up the PhoneBook
application’s GUI and components
    public PhoneBookApp() {
        setTitle("PhoneBook App with Background Image");
// Sets the window title at the top of the app
        setSize(500, 500);                    // Sets the size of
the application window to 500x500 pixels
        setDefaultCloseOperation(EXIT_ON_CLOSE);      //
Closes the application when the user closes the
window

        // Initialize the layeredPane to manage multiple
overlapping layers, like a background and input fields
        layeredPane = new JLayeredPane();
    layeredPane.setLayout(new BorderLayout());      //
Set layout manager to arrange components in the
center, top, etc.


    // Create a JTextArea where contact information
will be shown, and make it non-editable
    displayArea = new JTextArea();
    displayArea.setEditable(false);             // Prevents
user from typing in the display area
    JScrollPane scrollPane = new
JScrollPane(displayArea);  // Add scroll functionality in
case text overflows
    layeredPane.add(scrollPane,
BorderLayout.CENTER,
JLayeredPane.DEFAULT_LAYER);  // Place scroll pane in
center of layeredPane


    // Create a JPanel to hold input fields for adding or
updating contacts
    JPanel inputPanel = new JPanel(new GridLayout(5,
2));  // 5 rows, 2 columns grid layout


    inputPanel.add(new JLabel("Name:"));          //
Create and add label "Name:" next to the name input
field
```

```
    nameField = new JTextField();              // Initialize a
text field for the user to type the name
    inputPanel.add(nameField);                 // Add the
name field to the input panel


    inputPanel.add(new JLabel("Phone:"));         //
Create and add label "Phone:" next to the phone input
field
    phoneField = new JTextField();             // Initialize a
text field for the phone number
    inputPanel.add(phoneField);                // Add the
phone field to the input panel


    inputPanel.add(new JLabel("Email:"));         //
Create and add label "Email:" next to the email input
field
    emailField = new JTextField();             // Initialize a
text field for the email
    inputPanel.add(emailField);                // Add the
email field to the input panel


    inputPanel.add(new
JLabel("Search/Update/Delete:")); // Label for search
field, which can also be used for delete and update
```

```
    searchField = new JTextField();              // Initialize
the search field
    inputPanel.add(searchField);                 // Add
search field to the input panel

    layeredPane.add(inputPanel,
BorderLayout.NORTH, JLayeredPane.PALETTE_LAYER);
// Place input panel at the top of the layeredPane

    // Create a JPanel for the buttons that will perform
different actions (Insert, Search, Display, Delete,
Update, Sort)
    JPanel buttonPanel = new JPanel(new GridLayout(3,
2)); // 3 rows, 2 columns grid layout for the buttons

    // Create button to insert a new contact, and add an
action listener to define what happens when it’s
clicked
    JButton insertButton = new JButton("Insert
Contact");  // Button to add a new contact
    insertButton.addActionListener(new
InsertAction());    // Set up InsertAction to handle button
click
    buttonPanel.add(insertButton);                   // Add
button to the button panel
```

```
        // Create button to search for a contact, and define
what happens when it’s clicked
        JButton searchButton = new JButton("Search
Contact");  // Button to search for a contact
        searchButton.addActionListener(new
SearchAction());    // Set up SearchAction to handle
button click
        buttonPanel.add(searchButton);                    // Add
button to the button panel


        // Create button to display all contacts, and set up
the action to display them when clicked
        JButton displayButton = new JButton("Display
Contacts");  // Button to show all contacts
        displayButton.addActionListener(new
DisplayAction());     // Set up DisplayAction to handle
button click
        buttonPanel.add(displayButton);                    // Add
button to the button panel


        // Create button to delete a contact, and set up the
action to delete it when clicked
        JButton deleteButton = new JButton("Delete
Contact");     // Button to delete a contact
```

```
    deleteButton.addActionListener(new
DeleteAction());     // Set up DeleteAction to handle
button click
    buttonPanel.add(deleteButton);                    // Add
button to the button panel


    // Create button to update an existing contact, and
set up the action to perform the update when clicked
    JButton updateButton = new JButton("Update
Contact");    // Button to update a contact
    updateButton.addActionListener(new
UpdateAction());     // Set up UpdateAction to handle
button click
    buttonPanel.add(updateButton);                    // Add
button to the button panel


    // Create button to sort contacts, and set up the
action to sort them based on the chosen criteria when
clicked
    JButton sortButton = new JButton("Sort Contacts");
// Button to sort contacts by name, phone, or email
    sortButton.addActionListener(new SortAction());
// Set up SortAction to handle button click
    buttonPanel.add(sortButton);                    // Add
button to the button panel
```

```
        layeredPane.add(buttonPanel,
BorderLayout.SOUTH, JLayeredPane.PALETTE_LAYER);
// Place button panel at the bottom of the layeredPane


        // Add the layeredPane (containing all other
components) to the main JFrame window
        add(layeredPane);


        // Load a background image to give a visually
appealing look to the app

loadBackgroundImage("path/to/your/background.jpg")
;  // Replace "path/to/your/background.jpg" with actual
image file path


        setVisible(true);  // Make the application window
visible to the user
    }


    // This method loads an image and sets it as the
background of the app
    private void loadBackgroundImage(String imagePath)
{
        ImageIcon backgroundImage = new
ImageIcon(imagePath);  // Load image from the given
file path
        backgroundLabel = new JLabel(new
ImageIcon(backgroundImage.getImage().getScaledInst
ance(getWidth(), getHeight(),
Image.SCALE_SMOOTH)));
        setContentPane(backgroundLabel);                  // Set
the background label as the content pane of the
JFrame
        backgroundLabel.setLayout(new BorderLayout());
// Set layout for background label so other components
can be added on top
        backgroundLabel.add(layeredPane);                 //
Place layeredPane (with buttons, input fields, etc.) over
the background
    }


    // This class handles inserting a new contact when
the user clicks "Insert Contact"
    private class InsertAction implements ActionListener
{
        public void actionPerformed(ActionEvent e) {
            String name = nameField.getText();           //
Retrieve the name typed by the user
```

```
        String phone = phoneField.getText();          //
Retrieve the phone number typed by the user

        String email = emailField.getText();          //
Retrieve the email typed by the user


        // Only proceed if the name and phone fields are
not empty

        if (!name.isEmpty() && !phone.isEmpty()) {

          Contact newContact = new Contact(name,
phone, email);  // Create a new contact with the
entered information

          phonebook.add(newContact);              // Add the
new contact to the phonebook list

          displayArea.setText("Contact added: " +
newContact);  // Show a message confirming the
contact was added

          clearFields();                         // Clear the input
fields to prepare for new input

        } else {

          displayArea.setText("Name and Phone are
required."); // Inform the user that name and phone are
necessary

        }

      }

   }
```

```
  // This class handles searching for contacts based on
name or phone number when the user clicks "Search
Contact"

  private class SearchAction implements
ActionListener {

    public void actionPerformed(ActionEvent e) {

      String query = searchField.getText();       // Get
the search query from the search input field

      ArrayList<Contact> matches = new ArrayList<>();
// Create an empty list to store any contacts that match
the search


      // Check each contact in the phonebook to see if
it matches the search query

      for (Contact contact : phonebook) {

        // Compare the search query with the contact's
name or phone number (ignoring case differences)

        if (contact.getName().equalsIgnoreCase(query)
|| contact.phoneNumber.equals(query)) {

          matches.add(contact);          // If there’s a
match, add the contact to the matches list

        }

      }
```

```
        if (matches.size() == 0) {                // If no contacts
match the search query
            displayArea.setText("No contacts found with
that name or phone number.");
        } else if (matches.size() == 1) {          // If there is
exactly one matching contact
            contactToUpdate = matches.get(0);        // Set
the contact to be updated
            displayArea.setText("Contact found. Edit fields
and click Update to modify.");
            populateFields(contactToUpdate);         //
Display the contact's details in the input fields
        } else {
            displayMatchingContacts(matches);        // If
multiple matches, show them in a list for selection
        }
      }
   }

   // This method shows a list of contacts when there
are multiple matches for a search
```

```
  private void
displayMatchingContacts(ArrayList<Contact>
matches) {
    JFrame selectionFrame = new JFrame("Select a
Contact"); // Create a new window to show the list of
matching contacts
    selectionFrame.setSize(300, 200);           // Set
the size of this window
    selectionFrame.setLayout(new BorderLayout());
// Use a BorderLayout for the window

    // Create a JList containing the matched contacts,
allowing the user to pick one
    JList<Contact> contactList = new
JList<>(matches.toArray(new Contact[0]));

contactList.setSelectionMode(ListSelectionModel.SIN
GLE_SELECTION); // Only one contact can be selected
at a time

    // Listen for a mouse click on any contact in the list
    contactList.addMouseListener(new
MouseAdapter() {
      public void mouseClicked(MouseEvent evt) {
        if (evt.getClickCount() == 1) {          // If user
clicks a contact once
          contactToUpdate =
contactList.getSelectedValue();  // Get the selected
contact
          if (contactToUpdate != null) {
            populateFields(contactToUpdate);     //
Display the selected contact's info in the input fields
            displayArea.setText("Contact selected. Edit
fields and click Update to modify.");
            selectionFrame.dispose();          // Close the
selection window
          }
        }
      }
    });

    selectionFrame.add(new JScrollPane(contactList),
BorderLayout.CENTER); // Add the list to the window
with scroll enabled
    selectionFrame.setLocationRelativeTo(this);       //
Center the selection window over the main app window
    selectionFrame.setVisible(true);              // Make
the selection window visible
```

```
   }

   // This method fills the input fields with information
from a given contact
   private void populateFields(Contact contact) {
      nameField.setText(contact.name);               // Set
the name field with the contact’s name
      phoneField.setText(contact.phoneNumber);         //
Set the phone field with the contact’s phone number
      emailField.setText(contact.email);             // Set
the email field with the contact’s email
   }

   // Handles displaying all contacts in the phonebook
when the "Display Contacts" button is clicked
   private class DisplayAction implements
ActionListener {
      public void actionPerformed(ActionEvent e) {
         if (phonebook.isEmpty()) {                // Check if
there are any contacts in the phonebook
            displayArea.setText("Phonebook is empty.");
         } else {
            StringBuilder contacts = new StringBuilder(); //
Use a StringBuilder to gather all contact info
```

```
            for (Contact contact : phonebook) {
              contacts.append(contact).append("\n");   //
Append each contact's info to the display
            }
            displayArea.setText(contacts.toString());    //
Show all contacts in the display area
          }
        }
    }

    // Handles deleting a contact based on the search
query when "Delete Contact" button is clicked
    private class DeleteAction implements
ActionListener {
        public void actionPerformed(ActionEvent e) {
          String query = searchField.getText();         // Get
the query from the search field
          for (Contact contact : phonebook) {
            if (contact.getName().equalsIgnoreCase(query)
|| contact.phoneNumber.equals(query)) {
              phonebook.remove(contact);           // Remove
the matching contact from the phonebook
              displayArea.setText("Contact deleted."); //
Display a confirmation message
```

```
            clearFields();                    // Clear input fields
            return;                           // Exit after deletion
          }
        }
        displayArea.setText("Contact not found.");     //
Display message if no matching contact is found
      }
    }

    // Handles updating the selected contact's
information when "Update Contact" is clicked
    private class UpdateAction implements
ActionListener {
      public void actionPerformed(ActionEvent e) {
        if (contactToUpdate != null) {              // Check if a
contact has been selected for updating

contactToUpdate.setName(nameField.getText()); //
Update the name with the new value

contactToUpdate.setPhoneNumber(phoneField.getTex
t()); // Update the phone number
```

```
contactToUpdate.setEmail(emailField.getText()); //
Update the email
            displayArea.setText("Contact updated
successfully: " + contactToUpdate); // Confirm update
            clearFields();                      // Clear input fields
            contactToUpdate = null;                // Reset
selected contact
         } else {
            displayArea.setText("Please search for a
contact first to update."); // Prompt to search before
updating
         }
      }
   }

   // Handles sorting the contacts when "Sort Contacts"
is clicked, showing options to sort by name, phone, or
email
   private class SortAction implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         // Prompt the user to select a sorting criterion
         String[] options = {"Name", "Phone Number",
"Email"};
```

```
        String choice = (String)
JOptionPane.showInputDialog(
            null, "Sort contacts by:", "Sort Options",
            JOptionPane.QUESTION_MESSAGE, null,
options, options[0]);

        if (choice != null) {                    // Proceed only if
the user made a choice
          // Sort the contacts based on the chosen
criterion
          switch (choice) {
            case "Name":

phonebook.sort(Comparator.comparing(Contact::getN
ame, String.CASE_INSENSITIVE_ORDER));
              break;
            case "Phone Number":

phonebook.sort(Comparator.comparing(contact ->
contact.phoneNumber));
              break;
            case "Email":
}

phonebook.sort(Comparator.comparing(contact ->
contact.email, String.CASE_INSENSITIVE_ORDER));
            break;
        }
        displaySortedContacts();          // Show the
sorted contacts in the display area
      }
    }
  }

  // Method to display all contacts after sorting
  private void displaySortedContacts() {
    StringBuilder sortedContacts = new StringBuilder();
// Use StringBuilder to gather sorted contacts
    for (Contact contact : phonebook) {
      sortedContacts.append(contact).append("\n");
// Append each contact's info
    }
    displayArea.setText(sortedContacts.toString());   //
Display sorted contacts in the display area
  }
```

```
    // Clears the input fields (name, phone, email, and
search) to make them ready for new input
    private void clearFields() {
        nameField.setText("");                    // Clear the
name field
        phoneField.setText("");                   // Clear the
phone field
        emailField.setText("");                  // Clear the
email field
        searchField.setText("");                  // Clear the
search field
    }

    // The main method to start the application
    public static void main(String[] args) {
        new PhoneBookApp();                        // Create an
instance of PhoneBookApp, launching the GUI
    }
}
```