The provided code is an implementation of an address book application using a Binary Search Tree (BST) data structure to store and manage contact information. The BST provides an efficient way to store and manage a large number of contacts while allowing for quick search, insertion, deletion, and updating of contacts based on their names. The BST's inherent property of maintaining contacts in sorted order allows for easy retrieval and display of contacts in alphabetical order. This code used file handling operations error handling and input validation to properly save and load contacts from a file, handle invalid user inputs, and enhance code organization and modularity. With these enhancements, the code can become a functional and efficient address book application.

**Design and Implementation of Address Book Application:**

The address book application is designed to store and manage contact information using a Binary Search Tree (BST) data structure. Here are the design and implementation details:

* Contact Structure:
* The **Contact** structure represents a single contact and contains the following fields:
* **name**: Holds the name of the contact.
* **phone**: Stores the phone number of the contact.
* **email**: Stores the email address of the contact.
* Node Structure:
* The **Node** structure represents a node in the BST and contains the following fields:
* **contact**: Holds a **Contact** object that represents a contact.
* **left**: A pointer to the left child node.
* **right**: A pointer to the right child node.
* AddressBook Class:
* The **AddressBook** class manages the contacts using the BST.
* Private Data Members:
* **root**: A pointer to the root node of the BST.
* **filename**: A string that holds the name of the file to save/load contacts.
* Public Member Functions:
* **AddressBook(string& file)**: Constructor that initializes the BST and loads contacts from the file if available.
* **~AddressBook()**: Destructor that saves contacts to the file and destroys the BST.
* **void addContact(Contact& contact)**: Adds a new contact to the BST.
* **void searchContact(string& name)**: Searches for a contact by name and displays its details if found.
* **void updateContact(string& name, Contact& newContact)**: Updates the details of an existing contact.
* **void deleteContact(string& name)**: Deletes a contact from the BST.
* **void displayAllContacts()**: Displays all contacts in the address book.
* Private Member Functions:
* **Node\* add(Node\* node, Contact& contact)**: Inserts a new contact into the BST.
* **Node\* srch(Node\* node, string& name)**: Searches for a contact by name in the BST.
* **Node\* findMin(Node\* node)**: Finds the node with the minimum value in a given subtree.
* **Node\* rmv(Node\* node, string& name)**: Removes a contact from the BST.
* **void displayInOrder(Node\* node)**: Displays the contacts in the BST in sorted order.
* **void saveContactsToFile()**: Saves the contacts in the BST to the file.
* **void saveInOrder(Node\* node, ofstream& file)**: Saves the contacts in the BST to the file in sorted order.
* **void loadContactsFromFile()**: Loads contacts from the file into the BST.
* **void destroyBST(Node\* node)**: Deletes all nodes in the BST.
* Main Function:
* The **main** function serves as the entry point of the program.
* It interacts with the user to provide a menu-based interface for managing the address book using the **AddressBook** class.
* The user can choose options such as adding a contact, searching for a contact, updating a contact, deleting a contact, displaying all contacts, and exiting the program.

Data Structures and Algorithms:

* Binary Search Tree (BST):
* The BST data structure is chosen to efficiently store and manage the contact information.
* The contacts are sorted alphabetically by name in the BST, which allows for efficient search, insertion, deletion, and traversal operations.
* The BST property ensures that the contacts are maintained in sorted order, facilitating quick searching and displaying of contacts.
* Operations:
* Add Contact: The **addContact** function inserts a new contact into the BST using the appropriate ordering based on the name.
* Search Contact: The **searchContact** function performs a binary search in the BST based on the contact name to find and display the details of the contact if found.
* Update Contact: The **updateContact** function searches for a contact by name and updates its details if found.
* Delete Contact: The **deleteContact** function removes a contact from the BST while maintaining the BST property.
* Display All Contacts: The **displayAllContacts** function performs an in-order traversal of the BST and displays the contacts in sorted order.
* File Handling:
* The contacts are saved and loaded from a file using the **saveContactsToFile** and **loadContactsFromFile** functions, respectively.
* The contacts are saved in the file in sorted order using an in-order traversal of the BST.
* During program initialization, contacts are loaded from the file into the BST to maintain persistence across program executions.

The chosen design and implementation using a Binary Search Tree provide efficient operations for adding, searching, updating, and deleting contacts in the address book application.

**Testing Process:**

* Unit Testing: Each individual function in the AddressBook class can be tested independently to ensure their correctness. Test cases should cover different scenarios, such as adding, searching, updating, and deleting contacts, as well as edge cases like empty contact list, duplicate contacts, and invalid inputs.
* Integration Testing: The interaction between different functions and the overall flow of the application should be tested. This involves testing the main menu options and verifying that the application behaves as expected when performing multiple operations.
* File Handling Testing: Test cases should be created to check the saving and loading of contacts from the file. This includes verifying that contacts are saved correctly in the expected file format and loaded back into the address book correctly.

Performance Evaluation: To evaluate the performance of the application with a large number of contacts, you can consider the following aspects:

* Time Complexity:
* Insertion, search, update, and deletion operations in the BST have an average time complexity of O(log n), where n is the number of contacts in the address book. This allows for efficient handling of a large number of contacts.
* The in-order traversal to display all contacts has a time complexity of O(n), where n is the number of contacts. This operation is required to display all contacts in sorted order.
* Memory Usage:
* The BST data structure requires additional memory to store the nodes and contact details. However, it is memory-efficient compared to other data structures like arrays or linked lists when dealing with a large number of contacts. Each contact requires a constant amount of memory, and the memory usage grows with the number of contacts.
* Scalability:
* The BST-based implementation allows the application to scale well with a large number of contacts. The logarithmic time complexity of operations ensures that the performance remains reasonable even as the number of contacts increases.
* Performance Testing:
* Create test cases with a large number of contacts (e.g., thousands or more) and measure the execution time of various operations like adding, searching, updating, and deleting contacts.
* Monitor the memory usage of the application while handling a large number of contacts to ensure it remains within acceptable limits.

By conducting comprehensive testing and evaluating the performance, you can ensure that the address book application functions correctly and efficiently handle a large number of contacts.

**Sample Pictures:**



Fig1: Add Contact



Fig2: Search Contact



Fig3: Update Contact



Fig4: Delete Contact

**Conclusion:**

Solving the real-world problem of developing an address book application using data structures was a valuable experience. It allowed us to apply fundamental concepts of data structures and algorithms to create a practical solution.

Challenges Faced:

* Designing the data structure: Choosing the appropriate data structure to efficiently store and manage contacts was crucial. Initially, we used a map-based implementation, but later switched to a Binary Search Tree (BST) for better efficiency. Adapting the code to work with a BST required careful consideration of the BST operations and modifications to the existing code.
* File handling: Implementing file read/write operations and integrating them with the address book application posed some challenges. Ensuring the correct format for storing and loading contacts from the file required attention to detail.
* Variable: For selection menu at first it was “Integer” data type. If choice number 6 was selected some errors were occurring where the choice menu was running infinitively. So, the variable was changed from” Integer” to “Character” data type so in switch case the previously used numbers for menu was counting as individual character like ‘1’,’2’,’3’ etc.
* Use of Default Constructor: When creating a new Node in the insert function, the code was causing an error because the Contact structure did not have a default constructor defined. To resolve this issue, the Contact structure was modified to include a default constructor. By adding the default constructor Contact() = default; ,it allowed the creation of Contact objects without any arguments.

Lessons Learned:

* Choosing the right data structure: Selecting the appropriate data structure is essential for efficient operations. The BST provided a suitable balance between efficient search, insert, update, and delete operations, which are common requirements in an address book application.
* Modularity and reusability: Designing the code in a modular and reusable manner facilitated easier implementation of new features and changes. Separating concerns into classes and functions helped improve code organization and maintainability.
* Testing and error handling: Rigorous testing of different scenarios and edge cases helped uncover bugs and ensure the correctness of the application. Implementing robust error handling mechanisms, such as handling file I/O errors, improved the application's reliability.
* Performance considerations: Evaluating the performance of the application, especially with a large number of contacts, highlighted the importance of choosing efficient data structures and algorithms. Monitoring memory usage and execution times provided insights into the scalability and efficiency of the application.

Overall, developing the address book application provided hands-on experience in solving a real-world problem using data structures. It reinforced the importance of choosing the right data structures, designing modular and reusable code, thorough testing, and considering performance implications. These lessons can be applied to future projects to build efficient and robust software solutions.