academic publishers

INTERNATIONAL JOURNAL OF ARTIFICIAL INTELLIGENCE (ISSN: 2692-5206)

Volume 04, Issue 06, 2024

Published Date: 28-08-2024

DEVELOPMENT OF A PHONE BOOK MANAGEMENT SYSTEM USING DOUBLY LINKED LISTS IN DATA STRUCTURES WITH C++ PROGRAMMING LANGUAGE

Abror Musulmonov
Inha University in Tashkent School of Computer and Information Engineering
Phone: *+998934436004* ***Email:*** *abrorjon.musulmonov@gmail.com*

Abstract: This paper presents the development of a Phone Book Management System utilizing Doubly Linked Lists (DLLs) in C++. The system was designed to manage, store, and retrieve contact information efficiently, focusing on minimizing computational overhead. DLLs were chosen for their ability to allow bidirectional traversal, facilitating complex operations such as insertion, deletion, and search functions. The system's implementation, performance analysis, and potential enhancements are discussed, along with relevant code excerpts to illustrate the methodologies used.

Introduction: Efficient data management is critical in applications like contact management systems, where frequent updates and retrievals are required. Traditional data structures often fail to provide optimal performance under dynamic conditions, leading to inefficiencies. This study explores the utilization of Doubly Linked Lists (DLLs) to overcome these limitations. DLLs, with their bidirectional node connections, offer significant advantages in operations like insertion, deletion, and searching, making them ideal for a phone book management system.

Materials and Methods: The Phone Book Management System was implemented using C++ programming language, with the primary data structure being the Doubly Linked List. The DLL was chosen due to its bidirectional traversal capability, which allows for efficient insertion, deletion, and searching of nodes containing contact information.

The system's operations are defined as follows:

1. **Insertion**: Adding new contacts to the phone book.
2. **Deletion**: Removing existing contacts from the phone book.
3. **Searching**: Locating contacts by name, number, or email.
4. **Updating**: Modifying the details of existing contacts.
5. **Displaying**: Listing all saved contacts.

The choice of DLLs over other data structures was driven by their ability to handle dynamic data efficiently. Unlike singly linked lists, DLLs maintain pointers to both the next and previous nodes, allowing for easier node removal and list reversal, which are advantageous in the context of a phone book system.

Data Structures and Algorithms: The Phone Book Management System was developed using C++ programming language. The core data structure used is the Doubly Linked List (DLL). A DLL is a type of linked list where each node contains three components:

1. Data (in this case, contact information such as name, phone number, and email).

2. A pointer to the next node in the list.
3. A pointer to the previous node in the list.

The general structure of a node in a DLL is defined as follows:

```
struct Contact {
    string name;
    string phoneNumber;
    string email;
    Contact* next;
    Contact* prev;
};
```

Operations

The system supports the following operations:

1. **Insertion of a New Contact**
    - Inserting at the beginning, middle, or end of the list.
    - Time Complexity: O(1) for beginning, O(n) for end or middle.
2. **Deletion of an Existing Contact**
    - Removing a node based on contact name, phone number, or email.
    - Time Complexity: O(1) if at the beginning, O(n) if elsewhere.
3. **Searching for a Contact**
    - By name, phone number, or email.
    - Time Complexity: O(n).
4. **Updating Contact Information**
    - Modifying the details of an existing contact.
    - Time Complexity: O(n).
5. **Displaying All Contacts**
    - Traversing the DLL to display all stored contacts.
    - Time Complexity: O(n).

Formulas for Time Complexity Analysis

The time complexity for various operations in the DLL is derived based on the following:

- **Insertion**:

$$T_{\text{insert}} = \begin{cases} O(1) & \text{if at the beginning} \\ O(n) & \text{if at the end/middle} \end{cases}$$

- **Deletion**:

$$T_{\text{delete}} = \begin{cases} O(1) & \text{if at the beginning} \\ O(n) & \text{if at the end/middle} \end{cases}$$

- **Searching**:

$$T_{\text{search}} = O(n)$$

- **Updating**:

$$T_{\text{update}} = O(n)$$

Implementation Details: Below are key code snippets demonstrating the implementation of the system:

Insertion Function:

The insert() function allows for adding a new contact into the list. Depending on where the new contact needs to be inserted, the function adjusts the pointers accordingly.

```
void insert(string name, string phoneNumber, string email) {
    Contact* newContact = new Contact();
    newContact->name = name;
    newContact->phoneNumber = phoneNumber;
    newContact->email = email;
    newContact->next = head;
    newContact->prev = nullptr;

    if (head != nullptr) {
        head->prev = newContact;
    }
    head = newContact;
}
```

Deletion Function:

The deleteContact() function is responsible for removing a contact from the list. It locates the node and adjusts the pointers to maintain the DLL structure.

```
void deleteContact(string name) {
    Contact* current = head;
    while (current != nullptr && current->name != name) {
        current = current->next;
    }
    if (current == nullptr) return; // Contact not found

    if (current->prev != nullptr) {
        current->prev->next = current->next;
    } else {
        head = current->next;
    }
    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }
    delete current;
}
```

Search Function:

The searchByName() function searches the DLL for a contact by name. It traverses the list from the head to the end, comparing each node's name with the search query.

```
Contact* searchByName(string name) {
    Contact* current = head;
    while (current != nullptr) {
        if (current->name == name) return current;
        current = current->next;
    }
    return nullptr; // Not found
}
```

Results: The implementation of the Phone Book Management System demonstrated the effectiveness of DLLs in managing contact information. The system successfully performed all intended operations, including insertion, deletion, searching, updating, and displaying contacts. The bidirectional nature of DLLs facilitated efficient data manipulation, reducing the complexity of operations such as node deletion, which only required access to the target node.

The time complexity for basic operations was observed as follows:

- **Insertion**: O(1) for adding at the beginning, O(n) for adding at the end.
- **Deletion**: O(1) for deleting at the beginning, O(n) for deleting at the end or middle.
- **Searching**: O(n) for finding a contact by name, number, or email.

These performance metrics confirm the suitability of DLLs for applications requiring frequent modifications

and access to data, such as a phone book system.

Discussion: The implementation of a Phone Book Management System using DLLs revealed the strengths and limitations of this data structure in managing contact information. DLLs provide efficient insertion and deletion operations due to their bidirectional nature, which is particularly useful in applications requiring frequent updates.

However, the linear time complexity for searching and updating may become a performance bottleneck as the size of the contact list grows. Future work could explore the integration of more advanced data structures, such as balanced binary search trees or hash tables, to optimize these operations.

Additionally, the current implementation can be extended to include features such as:

- **Sorting Contacts**: Implementing a sorting algorithm to maintain contacts in alphabetical order.
- **Backup and Restore Functionality**: Adding features to export the contact list to a file and restore it from a backup.

Conclusion: The Phone Book Management System developed using Doubly Linked Lists effectively demonstrates the benefits of this data structure in managing dynamic contact information. While the system meets the basic requirements for a functional phone book, there is significant potential for optimization and feature expansion. This project lays the groundwork for future improvements and serves as a practical example of applying data structures in software development.

References:

1. Tutorialspoint. (n.d.). Doubly Linked List Data Structure. Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list.html
2. Programiz. (n.d.). Doubly Linked List in C Programming. Retrieved from https://www.programiz.com/dsa/doubly-linked-list
3. Studytonight. (n.d.). Doubly Linked List in Data Structure. Retrieved from https://www.studytonight.com/data-structures/doubly-linked-list
4. Codementor. (n.d.). Understanding and Implementing a Doubly Linked List in C++. Retrieved from https://www.codementor.io/@codementorteam/understanding-and-implementing-a-doubly-linked-list-in-c-du107y8f2
5. GeeksforGeeks. Introduction and Insertion in a Doubly Linked List. Retrieved from https://www.geeksforgeeks.org/introduction-and-insertion-in-a-doubly-linked-list/
6. Towards Data Science. Data Structures in C++ Part 1. Retrieved from https://towardsdatascience.com/data-structures-in-c-part-1-b64613b0138d/
7. PrepBytes. Advantages, Disadvantages, and Uses of a Doubly Linked List. Retrieved from https://www.prepbytes.com/blog/linked-list/advantages-disadvantages-and-uses-of-a-doubly-linked-list/
8. GitHub Repository by Rustam-Z. Data Structures and Algorithms. Retrieved from https://github.com/Rustam-Z/data-structures-and-algorithms