Contents

List of Figures

List of Tables

1 Introduction **1**
1.1 Overview . . . 1
1.2 Objectives . . . 1
1.3 Problem Statement . . . 2

2 Methodology **3**
2.1 Data Structure Design . . . 3
2.2 Implementation Approach . . . 3
2.2.1 Core Operations . . . 3
2.3 Error Handling and Input Validation . . . 4

3 Implementation Details **5**
3.1 Complete Source Code . . . 5

4 Simulation Scenarios **20**
4.1 Scenario 1: Basic List Creation and Display . . . 20
4.1.1 Initial Setup . . . 20
4.1.2 Resulting List Structure . . . 20
4.1.3 Display Operation . . . 20
4.2 Scenario 2: Insertion Operations . . . 21
4.2.1 Insert at Beginning . . . 21
4.2.2 Insert at End . . . 21
4.2.3 Insert at Position . . . 21
4.3 Scenario 3: Deletion Operations . . . 21
4.3.1 Delete from Beginning . . . 21
4.3.2 Delete from End . . . 22
4.3.3 Delete from Position . . . 22
4.4 Scenario 4: Search Operation . . . 22
4.5 Scenario 5: List Reversal . . . 23
4.5.1 Step-by-Step Reversal Process . . . 23
4.6 Scenario 6: List Sorting . . . 23
4.6.1 Bubble Sort Progress . . . 23
4.7 Scenario 7: Error Handling . . . 24
4.7.1 Invalid Position Insertion . . . 24
4.7.2 Empty List Operations . . . 24
4.7.3 Invalid Input Handling . . . 24

5 Algorithm Analysis **25**
5.1 Time Complexity Analysis . . . 25
5.2 Space Complexity Analysis . . . 25

5.3 Algorithm Efficiency Discussion 26
5.3.1 Insertion and Deletion at Beginning 26
5.3.2 Insertion and Deletion at End 26
5.3.3 Sorting Algorithm 26

6 Discussion **27**
6.1 Implementation Strengths 27
6.1.1 Robust Error Handling 27
6.1.2 User-Friendly Interface 27
6.1.3 Code Modularity 27
6.2 Potential Improvements 28
6.2.1 Performance Optimizations 28
6.2.2 Feature Enhancements 28
6.3 Real-World Applications 28

7 Conclusion **29**

List of Figures

List of Tables

5.1 Time Complexity of Linked List Operations 25

Chapter 1

Introduction

This report presents a comprehensive analysis and implementation of a singly linked list data structure in C programming language. The implementation includes various fundamental operations such as insertion, deletion, searching, sorting, and list manipulation, providing a complete framework for understanding dynamic data structures.

1.1 Overview

A linked list is a linear data structure where elements are stored in nodes, with each node containing data and a reference (pointer) to the next node in the sequence. Unlike arrays, linked lists provide dynamic memory allocation, allowing for efficient insertion and deletion operations at any position without the need for memory reallocation or element shifting.

The implementation presented in this report demonstrates a menu-driven program that allows users to interact with the linked list through various operations, showcasing the practical application of pointer manipulation and dynamic memory management in C.

1.2 Objectives

The primary objectives of this implementation are:

1. To develop a robust and complete implementation of a singly linked list in C
2. To implement all fundamental linked list operations with proper error handling
3. To demonstrate effective memory management using dynamic allocation
4. To provide comprehensive input validation and user-friendly interface

5. To implement advanced operations such as list reversal and sorting

6. To create a modular code structure that promotes code reusability and maintainability

1.3 Problem Statement

Traditional static data structures like arrays have limitations in terms of fixed size and inefficient insertion/deletion operations. This implementation addresses these limitations by creating a dynamic linked list structure that:

- Allows dynamic growth and shrinkage based on requirements
- Provides O(1) insertion and deletion at the beginning
- Enables flexible memory utilization
- Supports various operations without size constraints
- Handles edge cases and provides robust error handling

Chapter 2

Methodology

2.1 Data Structure Design

The linked list is implemented using a self-referential structure in C:

```c
struct Node {
    int data;
    struct Node* next;
};
```

Listing 2.1: Node Structure Definition

Each node contains:

- `data`: An integer value stored in the node
- `next`: A pointer to the next node in the list

2.2 Implementation Approach

The implementation follows a modular approach with separate functions for each operation:

2.2.1 Core Operations

1. **Node Creation**: Dynamic memory allocation for new nodes
2. **List Creation**: Initial list setup with user input
3. **Insertion Operations**: At beginning, end, and specific positions
4. **Deletion Operations**: From beginning, end, and specific positions

5. **Display**: Traversal and printing of list elements
6. **Search**: Linear search for specific elements
7. **Advanced Operations**: List reversal and sorting

2.3 Error Handling and Input Validation

The implementation includes comprehensive error handling:

- Memory allocation failure detection
- Input buffer clearing for invalid inputs
- Boundary checking for position-based operations
- Empty list condition handling
- Type validation using `scanf` return value checking and `strtol` for robust parsing

Chapter 3

Implementation Details

3.1 Complete Source Code

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data);
struct Node* createLinkedList();
void displayList(struct Node* head);
void insertAtBeginning(struct Node** head, int data);
void insertAtEnd(struct Node** head, int data);
void insertAtPosition(struct Node** head, int data, int position);
void deleteFromBeginning(struct Node** head);
void deleteFromEnd(struct Node** head);
void deleteAtPosition(struct Node** head, int position);
int searchElement(struct Node* head, int key);
int calculateLength(struct Node* head);
void reverseList(struct Node** head);
void sortList(struct Node** head);
void clearInputBuffer();
```

```c

int main() {
    struct Node* head = NULL;
    int choice, data, position;

    // Initial step to create a linked list
    head = createLinkedList();

    do {
        printf("\n----- Linked List Operations -----\n");
        printf("1. Insert at the Beginning\n");
        printf("2. Insert at the End\n");
        printf("3. Insert at a Specific Position\n");
        printf("4. Delete from the Beginning\n");
        printf("5. Delete from the End\n");
        printf("6. Delete from a Specific Position\n");
        printf("7. Display List\n");
        printf("8. Search Element\n");
        printf("9. Calculate Length\n");
        printf("10. Reverse List\n");
        printf("11. Sort List\n");
        printf("0. Exit\n");
        printf("Enter your choice: ");

        if (scanf("%d", &choice) != 1) {
            printf("Invalid choice. Please try again.\n");
            clearInputBuffer();
            continue;
        }


```

```
        switch (choice) {
            case 1:
                printf("Enter data to insert at the beginning: ");
                // Check the return value of scanf
                if (scanf("%d", &data) != 1) {
                    printf("Invalid input. Please enter a valid integer.\n");
                    clearInputBuffer();  // Clear the input buffer
                    break;
                }
                insertAtBeginning(&head, data);
                break;
            case 2:
                printf("Enter data to insert at the end: ");
                // Check the return value of scanf
                if (scanf("%d", &data) != 1) {
                    printf("Invalid input. Please enter a valid integer.\n");
                    clearInputBuffer();  // Clear the input buffer
                    break;
                }
                insertAtEnd(&head, data);
                break;
            case 3:
                printf("Enter data to insert: ");
                // Check the return value of scanf
                if (scanf("%d", &data) != 1) {
                    printf("Invalid input. Please enter a valid integer.\n");
                    clearInputBuffer();  // Clear the input buffer
                    break;
                }
                printf("Enter position to insert at: ");
                // Check the return value of scanf
```

```
                if (scanf("%d", &position) != 1) {
                    printf("Invalid input. Please enter a valid integer.\n");
                    clearInputBuffer();  // Clear the input buffer
                    break;
                }
                insertAtPosition(&head, data, position);
                break;
            case 4:
                deleteFromBeginning(&head);
                break;
            case 5:
                deleteFromEnd(&head);
                break;
            case 6:
                printf("Enter position to delete: ");
                // Check the return value of scanf
                if (scanf("%d", &position) != 1) {
                    printf("Invalid input. Please enter a valid integer.\n");
                    clearInputBuffer();  // Clear the input buffer
                    break;
                }
                deleteAtPosition(&head, position);
                break;
            case 7:
                displayList(head);
                break;
            case 8:
                printf("Enter element to search: ");
                // Check the return value of scanf
                if (scanf("%d", &data) != 1) {
                    printf("Invalid input. Please enter a valid integer.\n");
```

```
                clearInputBuffer();  // Clear the input buffer
                break;
            }
            position = searchElement(head, data);
            if (position != -1)
                printf("Element found at position %d\n", position);
            else
                printf("Element not found\n");
            break;
        case 9:
            printf("Length of the list: %d\n", calculateLength(head));
            break;
        case 10:
            reverseList(&head);
            if (head == NULL) {
                printf("List is empty. Nothing to reverse.\n");
            } else {
                printf("List reversed successfully.\n");
            }
            break;
        case 11:
            sortList(&head);
            if (head == NULL) {
                printf("List is empty. Nothing to sort.\n");
            } else {
                printf("List sorted successfully.\n");
            }
            break;
        case 0:
            printf("Exiting program.\n");
            break;
```

```
            default:
                printf("Invalid choice. Please try again.\n");
                clearInputBuffer();  // Clear the input buffer in case of invalid input
        }
    } while (choice != 0);

    return 0;
}

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Node* createLinkedList() {
    struct Node* head = NULL;
    struct Node* tail = NULL;
    char buffer[100];  // Adjust the buffer size as needed
    char *endptr;

    int n;  // Variable to store the number of elements

    do {
        printf("Enter the number of elements: ");
```

```

        // Read a line of input for the number of elements
        if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
            printf("Error reading input.\n");
            exit(EXIT_FAILURE);
        }

        // Use strtol to parse the integer
        n = strtol(buffer, &endptr, 10);

        // Check for conversion errors
        if (*endptr != '\n' && *endptr != '\0') {
            printf("Invalid input. Please enter a valid integer.\n");
        } else if (n < 0) {
            printf("Number of elements cannot be negative.\n");
            // Force loop to continue
            *endptr = 'x';
        }
    } while (*endptr != '\n' && *endptr != '\0');

    for (int i = 0; i < n; ++i) {
        do {
            printf("Enter element %d: ", i + 1);

            // Read a line of input for the element
            if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
                printf("Error reading input.\n");
                exit(EXIT_FAILURE);
            }


```

```
            long data = strtol(buffer, &endptr, 10);

            // Check for conversion errors
            if (*endptr != '\n' && *endptr != '\0') {
                printf("Invalid input. Please enter a valid integer for
    element %d: ", i + 1);
                clearInputBuffer();  // Clear the input buffer
            } else {
                struct Node* newNode = createNode((int)data);
                if (head == NULL) {
                    head = newNode;
                    tail = newNode;
                } else {
                    tail->next = newNode;
                    tail = newNode;
                }
                break;  // Break the loop if input is valid
            }
        } while (1);
    }

    return head;
}

// Function to display the elements of the linked list
void displayList(struct Node* head) {
    struct Node* current = head;

    if (current == NULL) {
        printf("List is empty.\n");
        return;
```

```
    }

    printf("Linked List: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Function to insert a node at the beginning of the linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
    printf("Element inserted at the beginning.\n");
}

// Function to insert a node at the end of the linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
    printf("Element inserted at the end.\n");
```

```
}

// Function to insert a node at a specified position in the linked list
void insertAtPosition(struct Node** head, int data, int position) {
    if (position < 1) {
        printf("Invalid position. Position should be >= 1.\n");
        return;
    }

    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }

    struct Node* newNode = createNode(data);
    struct Node* current = *head;
    for (int i = 1; i < position - 1 && current != NULL; ++i) {
        current = current->next;
    }

    if (current == NULL) {
        printf("Position out of bounds.\n");
    } else {
        newNode->next = current->next;
        current->next = newNode;
        printf("Element inserted at position %d.\n", position);
    }
}

// Function to delete a node from the beginning of the linked list
void deleteFromBeginning(struct Node** head) {
```

```
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
    printf("Element deleted from the beginning.\n");
}

// Function to delete a node from the end of the linked list
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        printf("Element deleted from the end.\n");
        return;
    }

    struct Node* current = *head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    free(current->next);
```

```
    current->next = NULL;
    printf("Element deleted from the end.\n");
}

// Function to delete a node from a specified position in the linked list
void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    if (position < 1) {
        printf("Invalid position. Position should be >= 1.\n");
        return;
    }

    if (position == 1) {
        deleteFromBeginning(head);
        return;
    }

    struct Node* current = *head;
    struct Node* previous = NULL;
    for (int i = 1; i < position && current != NULL; ++i) {
        previous = current;
        current = current->next;
    }

    if (current == NULL) {
        printf("Position out of bounds.\n");
    } else {
```

```c
        previous->next = current->next;
        free(current);
        printf("Element deleted from position %d.\n", position);
    }
}

// Function to search for a specific element in the linked list and display its position
int searchElement(struct Node* head, int key) {
    struct Node* current = head;
    int position = 1;

    while (current != NULL) {
        if (current->data == key) {
            return position;
        }
        current = current->next;
        position++;
    }

    return -1; // Element not found
}

// Function to calculate and display the length of the linked list
int calculateLength(struct Node* head) {
    struct Node* current = head;
    int length = 0;

    while (current != NULL) {
        length++;
        current = current->next;
```

```c
    }

    return length;
}

// Function to reverse the linked list
void reverseList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head = prev;
}

// Function to sort the linked list using bubble sort
void sortList(struct Node** head) {
    int swapped, temp;
    struct Node* current;
    struct Node* last = NULL;

    if (*head == NULL) {
        printf("List is empty. Nothing to sort.\n");
        return;
    }
```

```c

    do {
        swapped = 0;
        current = *head;

        while (current->next != last) {
            if (current->data > current->next->data) {
                temp = current->data;
                current->data = current->next->data;
                current->next->data = temp;
                swapped = 1;
            }
            current = current->next;
        }

        last = current;
    } while (swapped);
}

// Function to clear the input buffer
void clearInputBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}
```

Listing 3.1: Complete LinkedList.c Implementation

Chapter 4

Simulation Scenarios

This chapter presents various simulation scenarios demonstrating the functionality of the linked list implementation. Each scenario shows the step-by-step execution and the resulting state of the linked list.

4.1 Scenario 1: Basic List Creation and Display

4.1.1 Initial Setup

```
Enter the number of elements: 4
Enter element 1: 10
Enter element 2: 20
Enter element 3: 30
Enter element 4: 40
```

4.1.2 Resulting List Structure

```
[10] -> [20] -> [30] -> [40] -> NULL
```

4.1.3 Display Operation

```
Choice: 7
Linked List: 10 20 30 40
```

4.2 Scenario 2: Insertion Operations

4.2.1 Insert at Beginning

```
Initial List: [10] -> [20] -> [30] -> [40] -> NULL

Choice: 1

Enter data to insert at the beginning: 5

Element inserted at the beginning.

Resulting List: [5] -> [10] -> [20] -> [30] -> [40] -> NULL
```

4.2.2 Insert at End

```
Current List: [5] -> [10] -> [20] -> [30] -> [40] -> NULL

Choice: 2

Enter data to insert at the end: 50

Element inserted at the end.

Resulting List: [5] -> [10] -> [20] -> [30] -> [40] -> [50] -> NULL
```

4.2.3 Insert at Position

```
Current List: [5] -> [10] -> [20] -> [30] -> [40] -> [50] -> NULL

Choice: 3

Enter data to insert: 25

Enter position to insert at: 4

Element inserted at position 4.

Resulting List: [5] -> [10] -> [20] -> [25] -> [30] -> [40] -> [50] -> NULL
```

4.3 Scenario 3: Deletion Operations

4.3.1 Delete from Beginning

```
Initial List: [5] -> [10] -> [20] -> [25] -> [30] -> [40] -> [50] -> NULL
```

```
Choice: 4
Element deleted from the beginning.
Resulting List: [10] -> [20] -> [25] -> [30] -> [40] -> [50] -> NULL
```

4.3.2 Delete from End

```
Current List: [10] -> [20] -> [25] -> [30] -> [40] -> [50] -> NULL
Choice: 5
Element deleted from the end.
Resulting List: [10] -> [20] -> [25] -> [30] -> [40] -> NULL
```

4.3.3 Delete from Position

```
Current List: [10] -> [20] -> [25] -> [30] -> [40] -> NULL
Choice: 6
Enter position to delete: 3
Element deleted from position 3.
Resulting List: [10] -> [20] -> [30] -> [40] -> NULL
```

4.4 Scenario 4: Search Operation

```
Current List: [10] -> [20] -> [30] -> [40] -> NULL
Choice: 8
Enter element to search: 30
Element found at position 3

Choice: 8
Enter element to search: 25
Element not found
```

4.5 Scenario 5: List Reversal

```
Initial List: [10] -> [20] -> [30] -> [40] -> NULL

Choice: 10

List reversed successfully.

Resulting List: [40] -> [30] -> [20] -> [10] -> NULL
```

4.5.1 Step-by-Step Reversal Process

1. Initial: prev=NULL, current=[10], next=NULL
2. Iteration 1: prev=[10], current=[20], [10]-¿next=NULL
3. Iteration 2: prev=[20], current=[30], [20]-¿next=[10]
4. Iteration 3: prev=[30], current=[40], [30]-¿next=[20]
5. Iteration 4: prev=[40], current=NULL, [40]-¿next=[30]
6. Final: head=[40]

4.6 Scenario 6: List Sorting

```
Initial List: [40] -> [10] -> [30] -> [20] -> [50] -> NULL

Choice: 11

List sorted successfully.

Resulting List: [10] -> [20] -> [30] -> [40] -> [50] -> NULL
```

4.6.1 Bubble Sort Progress

```
Pass 1: [10] -> [30] -> [20] -> [40] -> [50] -> NULL

Pass 2: [10] -> [20] -> [30] -> [40] -> [50] -> NULL

Pass 3: No swaps needed - List is sorted
```

4.7 Scenario 7: Error Handling

4.7.1 Invalid Position Insertion

```
Current List: [10] -> [20] -> [30] -> NULL
Choice: 3
Enter data to insert: 25
Enter position to insert at: 10
Position out of bounds.
```

4.7.2 Empty List Operations

```
Empty List: NULL
Choice: 4
List is empty. Nothing to delete.

Choice: 7
List is empty.
```

4.7.3 Invalid Input Handling

```
Choice: abc
Invalid choice. Please try again.

Choice: 1
Enter data to insert at the beginning: xyz
Invalid input. Please enter a valid integer.
```

Chapter 5

Algorithm Analysis

5.1 Time Complexity Analysis

Table 5.1: Time Complexity of Linked List Operations

Operation	Best Case	Average Case	Worst Case
Insert at Beginning	O(1)	O(1)	O(1)
Insert at End	O(n)	O(n)	O(n)
Insert at Position	O(1)	O(n)	O(n)
Delete from Beginning	O(1)	O(1)	O(1)
Delete from End	O(n)	O(n)	O(n)
Delete from Position	O(1)	O(n)	O(n)
Search	O(1)	O(n)	O(n)
Display	O(n)	O(n)	O(n)
Calculate Length	O(n)	O(n)	O(n)
Reverse List	O(n)	O(n)	O(n)
Sort List	O(n)	$O(n^2)$	$O(n^2)$

5.2 Space Complexity Analysis

The space complexity for the linked list implementation is O(n), where n is the number of elements in the list. Additional space requirements:

- Each node requires space for data (int) and pointer (struct Node*)
- Temporary variables for operations: O(1)
- No additional arrays or data structures used
- Recursive functions not used, preventing stack overflow

5.3 Algorithm Efficiency Discussion

5.3.1 Insertion and Deletion at Beginning

These operations are highly efficient with O(1) time complexity, making linked lists ideal for stack-like operations.

5.3.2 Insertion and Deletion at End

These operations require traversal to the last node, resulting in O(n) complexity. This could be optimized by maintaining a tail pointer.

5.3.3 Sorting Algorithm

The implementation uses bubble sort with $O(n^2)$ complexity. For better performance, merge sort or quicksort could be implemented for O(n log n) average case.

Chapter 6

Discussion

6.1 Implementation Strengths

6.1.1 Robust Error Handling

The implementation demonstrates comprehensive error handling through multiple mechanisms:

- Memory allocation failure detection with graceful exit
- Input validation using both `scanf` return checking and `strtol` parsing
- Boundary checking for all position-based operations
- Empty list condition handling in all relevant operations

6.1.2 User-Friendly Interface

The menu-driven interface provides:

- Clear operation descriptions
- Immediate feedback for all operations
- Input error recovery without program termination
- Intuitive navigation and operation selection

6.1.3 Code Modularity

Each operation is implemented as a separate function, promoting:

- Code reusability

- Easy maintenance and debugging
- Clear separation of concerns
- Potential for unit testing

6.2 Potential Improvements

6.2.1 Performance Optimizations

1. **Tail Pointer**: Maintaining a tail pointer would reduce insertion at end from O(n) to O(1)
2. **Doubly Linked List**: Would enable O(1) deletion from end and bidirectional traversal
3. **Better Sorting**: Implementing merge sort would improve sorting from $O(n^2)$ to O(n log n)

6.2.2 Feature Enhancements

1. **Generic Data Type**: Using void pointers to support any data type
2. **File Operations**: Save and load list from files
3. **Memory Leak Prevention**: Implement a function to free entire list before program exit
4. **Duplicate Handling**: Options to handle duplicate values during insertion

6.3 Real-World Applications

Linked lists find applications in various domains:

- **Operating Systems**: Process scheduling, memory management
- **Compiler Design**: Symbol table management
- **Music Players**: Playlist implementation
- **Web Browsers**: Forward and backward navigation
- **Image Viewers**: Previous/next image navigation

Chapter 7

Conclusion

This report has presented a comprehensive implementation of a singly linked list in C, demonstrating fundamental data structure concepts and programming techniques. The implementation successfully achieves all stated objectives:

1. Complete implementation of all basic linked list operations
2. Robust error handling and input validation
3. Efficient memory management using dynamic allocation
4. User-friendly interface with clear feedback
5. Advanced operations including reversal and sorting
6. Modular code structure promoting maintainability

The simulation scenarios demonstrate the practical functionality of each operation, while the algorithm analysis provides insights into performance characteristics. The implementation serves as an excellent educational tool for understanding dynamic data structures and pointer manipulation in C.

Future enhancements could include performance optimizations such as tail pointer maintenance, implementation of doubly linked lists, and more efficient sorting algorithms. The modular design facilitates such improvements without major structural changes.

This linked list implementation provides a solid foundation for understanding more complex data structures and algorithms, making it valuable for both educational purposes and as a reference for practical applications.