CAM2003C - Data Structures and Algorithms with C and C++

Lab Exercise: Singly Linked Lists in C and C++

Solve the below problems in C/C++ and give the time and space complexities

1. Write Algorithm/Pseudocode for following operations on a Singly Linked Lists

a) Inserting a new node in a Linked List at the beginning
b) Inserting a new node in a Linked List at the end.
c) Inserting a new node in a Linked List after a given node.
d) Inserting a new node in a Linked List before a given node
e) Deleting a node from a Linked List from beginning
f) Deleting a node from a Linked List from end
g) Deleting a node after a given node
h) Display the entire linked list (Traversal)
i) Search for an element in linked list.
j) Count the number of nodes in a Singly Linked List
k) Reverse the singly linked list.

2. Do an Apriori analysis on the Time and Space complexities of above algorithms on a Singly Linked List.
3. How can you optimise the insertion of an element at the end. Discuss.
4. Discuss about why it's not possible to delete a node before a given node
5. Write a C and C++ program for implementation of Singly Linked List consisting of following operations:

- Inserting a new node in a Linked List at the beginning
- Inserting a new node in a Linked List at the end.
- Deleting a node from a Linked List from beginning
- Deleting a node from a Linked List from end
- Display the entire linked list (Traversal)
- Now test the working of the program by calling these functions as follows:
- Inserting a new node in a Linked List at the beginning: 45
- Inserting a new node in a Linked List at the beginning:55
- Inserting a new node in a Linked List at the beginning:65
- Inserting a new node in a Linked List at the end: 67
- Inserting a new node in a Linked List at the end: 77
- Inserting a new node in a Linked List at the end: 87
- Display the entire linked list (Traversal)
- Deleting a node from a Linked List from beginning
- Deleting a node from a Linked List from beginning
- Display the entire linked list (Traversal)
- Deleting a node from a Linked List from end
- Deleting a node from a Linked List from end

- Display the entire linked list (Traversal)

// Program in C to implement a Singly Linked List

//singlylinkedlist.c

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

// Define structure for a node in the singly linked list
struct Node {
    int data;              // data to store
    struct Node* next;     // pointer to the next node
};

// Global pointer to the head of the list
struct Node* head = NULL;

// Function declarations
void insertAtBeginning(int val);
void insertAtEnd(int val);
void deleteFromBeginning();
void deleteFromEnd();
void display();

int main() {

insertAtBeginning(45);
insertAtBeginning(55);
insertAtBeginning(65);

insertAtEnd(67);
insertAtEnd(77);
insertAtEnd(87);
```

```
display();

deleteFromBeginning();
deleteFromBeginning();

display();

deleteFromEnd();
deleteFromEnd();

display();



    return 0;
}

// Insert a new node at the beginning
void insertAtBeginning(int val) {
    // Allocate memory for new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = val;        // set data
    newNode->next = head;       // new node points to current head
    head = newNode;             // head is updated to new node
}

// Insert a new node at the end
void insertAtEnd(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = val;
    newNode->next = NULL;       // end node always points to NULL
```

```
    // If list is empty, new node becomes head
    if (head == NULL) {
        head = newNode;
        return;
    }

    // Otherwise, traverse to last node
    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;       // last node points to new node
}

// Delete a node from the beginning
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;   // store current head in temp
    head = head->next;          // move head to next node
    free(temp);                 // delete old head
    printf("Deleted from beginning.\n");
}

// Delete a node from the end
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
```

```
        return;
    }

    // If there's only one node
    if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("Deleted last node.\n");
        return;
    }

    // Traverse to second-last node
    struct Node* temp = head;
    while (temp->next->next != NULL)
        temp = temp->next;

    free(temp->next);           // delete last node
    temp->next = NULL;          // update next of second last to
NULL
    printf("Deleted from end.\n");
}

// Traverse and display the linked list
void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);  // print node data
```

```
        temp = temp->next;              // move to next node
    }
    printf("NULL\n");
}
```

// Program in C++ to implement a Singly Linked List

//singlylinkedlist.cpp

```
#include <iostream>
using namespace std;

// Node class for singly linked list
class Node {
public:
    int data;      // Data in the node
    Node* next;    // Pointer to next node

    // Constructor initializes the data and sets next to nullptr
    Node(int val) {
        data = val;
        next = nullptr;
    }
};

// LinkedList class to manage list operations
class LinkedList {
    Node* head;    // Pointer to the head of the list

public:
    LinkedList() { head = nullptr; }   // Constructor initializes
head to null

    void insertAtBeginning(int val);
    void insertAtEnd(int val);
```

```
    void deleteFromBeginning();
    void deleteFromEnd();
    void display();
};

// Insert at beginning of list
void LinkedList::insertAtBeginning(int val) {
    Node* newNode = new Node(val); // allocate new node with value
    newNode->next = head;          // link new node to current head
    head = newNode;                // update head to new node
}

// Insert at end of list
void LinkedList::insertAtEnd(int val) {
    Node* newNode = new Node(val); // allocate new node

    if (!head) {
        head = newNode;            // if list is empty
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) // traverse to end
        temp = temp->next;

    temp->next = newNode;         // link last node to new node
}

// Delete first node
void LinkedList::deleteFromBeginning() {
    if (!head) {
        cout << "List is empty.\n";
```

```
        return;
    }

    Node* temp = head;
    head = head->next;            // move head forward
    delete temp;                  // delete old head
    cout << "Deleted from beginning.\n";
}

// Delete last node
void LinkedList::deleteFromEnd() {
    if (!head) {
        cout << "List is empty.\n";
        return;
    }

    if (!head->next) {
        delete head;
        head = nullptr;
        cout << "Deleted last node.\n";
        return;
    }

    Node* temp = head;
    while (temp->next->next != nullptr) // go to second-last node
        temp = temp->next;

    delete temp->next;             // delete last node
    temp->next = nullptr;          // update second-last node's next
    cout << "Deleted from end.\n";
}
```

```
// Display the list
void LinkedList::display() {
    if (!head) {
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;
    cout << "List: ";
    while (temp) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}

// Main function with menu
int main() {

                LinkedList list;
                list.insertAtBeginning(45);
                list.insertAtBeginning(55);
                list.insertAtBeginning(65);


                list.insertAtEnd(67);
                list.insertAtEnd(77);
                list.insertAtEnd(87);

                list.display();

                list.deleteFromBeginning();
```

```
    list.deleteFromBeginning();

    list.display();

    list.deleteFromEnd();
    list.deleteFromEnd();

    list.display();


    return 0;
}
```

6. Modify the above program (C and C++ both) to make a Menu Driven application. And test the code by invoking the the function as per above given order.

// Menu Driven Program in C to implement a Singly Linked List

//singlylinkedlist_menudriven.c

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

// Define structure for a node in the singly linked list
struct Node {
    int data;              // data to store
    struct Node* next;     // pointer to the next node
};

// Global pointer to the head of the list
struct Node* head = NULL;

// Function declarations
void insertAtBeginning(int val);
void insertAtEnd(int val);
void deleteFromBeginning();
```

```
void deleteFromEnd();
void display();

int main() {
    int choice, value;

    while (1) {
        printf("\n--- Singly Linked List Menu ---\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Delete from Beginning\n");
        printf("4. Delete from End\n");
        printf("5. Display\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Menu-driven interface
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;

            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;

            case 3:
```

```
                deleteFromBeginning();
                break;

            case 4:
                deleteFromEnd();
                break;

            case 5:
                display();
                break;

            case 6:
                exit(0);

            default:
                printf("Invalid choice!\n");
        }
    }

    return 0;
}

// Insert a new node at the beginning
void insertAtBeginning(int val) {
    // Allocate memory for new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = val;        // set data
    newNode->next = head;       // new node points to current head
    head = newNode;             // head is updated to new node
}

// Insert a new node at the end
```

```
void insertAtEnd(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = val;
    newNode->next = NULL;       // end node always points to NULL

    // If list is empty, new node becomes head
    if (head == NULL) {
        head = newNode;
        return;
    }

    // Otherwise, traverse to last node
    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;       // last node points to new node
}

// Delete a node from the beginning
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;   // store current head in temp
    head = head->next;          // move head to next node
    free(temp);                 // delete old head
    printf("Deleted from beginning.\n");
}
```

```
// Delete a node from the end
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    // If there's only one node
    if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("Deleted last node.\n");
        return;
    }

    // Traverse to second-last node
    struct Node* temp = head;
    while (temp->next->next != NULL)
        temp = temp->next;

    free(temp->next);           // delete last node
    temp->next = NULL;          // update next of second last to
NULL
    printf("Deleted from end.\n");
}

// Traverse and display the linked list
void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
```

```
    struct Node* temp = head;
    printf("List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);  // print node data
        temp = temp->next;             // move to next node
    }
    printf("NULL\n");
}
```

// Menu Driven Program in C++ to implement a Singly Linked List

//singlylinkedlist_menudriven.cpp

```
#include <iostream>
using namespace std;

// Node class for singly linked list
class Node {
public:
    int data;      // Data in the node
    Node* next;    // Pointer to next node

    // Constructor initializes the data and sets next to nullptr
    Node(int val) {
        data = val;
        next = nullptr;
    }
};

// LinkedList class to manage list operations
class LinkedList {
    Node* head;    // Pointer to the head of the list

public:
```

```
    LinkedList() { head = nullptr; }   // Constructor initializes
head to null

    void insertAtBeginning(int val);
    void insertAtEnd(int val);
    void deleteFromBeginning();
    void deleteFromEnd();
    void display();
};

// Insert at beginning of list
void LinkedList::insertAtBeginning(int val) {
    Node* newNode = new Node(val); // allocate new node with value
    newNode->next = head;          // link new node to current head
    head = newNode;                // update head to new node
}

// Insert at end of list
void LinkedList::insertAtEnd(int val) {
    Node* newNode = new Node(val); // allocate new node

    if (!head) {
        head = newNode;            // if list is empty
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) // traverse to end
        temp = temp->next;

    temp->next = newNode;         // link last node to new node
}
```

```
// Delete first node
void LinkedList::deleteFromBeginning() {
    if (!head) {
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;
    head = head->next;            // move head forward
    delete temp;                  // delete old head
    cout << "Deleted from beginning.\n";
}

// Delete last node
void LinkedList::deleteFromEnd() {
    if (!head) {
        cout << "List is empty.\n";
        return;
    }

    if (!head->next) {
        delete head;
        head = nullptr;
        cout << "Deleted last node.\n";
        return;
    }

    Node* temp = head;
    while (temp->next->next != nullptr) // go to second-last node
        temp = temp->next;

    delete temp->next;             // delete last node
```

```
    temp->next = nullptr;          // update second-last node's next
    cout << "Deleted from end.\n";
}

// Display the list
void LinkedList::display() {
    if (!head) {
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;
    cout << "List: ";
    while (temp) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}

// Main function with menu
int main() {
    LinkedList list;
    int choice, value;

    while (true) {
        cout << "\n--- Singly Linked List Menu ---\n";
        cout << "1. Insert at Beginning\n";
        cout << "2. Insert at End\n";
        cout << "3. Delete from Beginning\n";
        cout << "4. Delete from End\n";
        cout << "5. Display\n";
```

```
cout << "6. Exit\n";
cout << "Enter your choice: ";
cin >> choice;

// Switch case for operations
switch (choice) {
    case 1:
        cout << "Enter value to insert: ";
        cin >> value;
        list.insertAtBeginning(value);
        break;

    case 2:
        cout << "Enter value to insert: ";
        cin >> value;
        list.insertAtEnd(value);
        break;

    case 3:
        list.deleteFromBeginning();
        break;

    case 4:
        list.deleteFromEnd();
        break;

    case 5:
        list.display();
        break;

    case 6:
        return 0;
```

```
            default:
                cout << "Invalid choice!\n";
        }
    }

    return 0;
}
```

7. In the above C code, also add the code of Count the no. of nodes and search a node operation.

8. Modify the code of C implementation of Question 6 and use calloc() to allocate the memory for a node.

9. Write code for a function to create a Node of Linked List and call that function in the insert operations of Linekd List implementation of Question 6.

```
struct Node* createNode(int value) {
    // Allocate memory for a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));

    // Check if malloc failed
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }

    // Assign data and next
    newNode->data = value;
    newNode->next = NULL;

    return newNode;
```

```
}
```

10. Create a menu-driven C program using a Singly Linked List to manage student records. Each node stores:

- Roll No. (treated as PRN)
- Name
- Phone Number
- City
- HSC Marks

The list supports:

- Insert at beginning
- Insert at end
- Delete from beginning
- Delete from end
- Display all records
- Count records
- Search by PRN (Roll No)

Solution

// A menu-driven C program using a Singly Linked List to manage student records

//student_record_processing_linkedlist.c

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

// Define the structure for a student record
struct Student {
    int rollNo;                // Unique identifier (PRN)
    char name[50];
    char phone[15];
    char city[30];
```

```
    float hscMarks;
    struct Student* next;
};

// Global head pointer
struct Student* head = NULL;

// Function to create a new student node
struct Student* createNode() {
    struct Student* newNode = (struct Student*)malloc(sizeof(struct
Student));
    if (!newNode) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    printf("Enter Roll No (PRN): ");
    scanf("%d", &newNode->rollNo);
    printf("Enter Name: ");
    scanf(" %[^\n]", newNode->name);
    printf("Enter Phone No: ");
    scanf(" %[^\n]", newNode->phone);
    printf("Enter City: ");
    scanf(" %[^\n]", newNode->city);
    printf("Enter HSC Marks: ");
    scanf("%f", &newNode->hscMarks);

    newNode->next = NULL;
    return newNode;
}

// Insert at the beginning
void insertAtBeginning() {
    struct Student* newNode = createNode();
```

```
    newNode->next = head;
    head = newNode;
    printf("Record inserted at beginning.\n");
}

// Insert at the end
void insertAtEnd() {
    struct Student* newNode = createNode();
    if (head == NULL) {
        head = newNode;
    } else {
        struct Student* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("Record inserted at end.\n");
}

// Delete from beginning
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty! No record to delete.\n");
        return;
    }
    struct Student* temp = head;
    head = head->next;
    printf("Deleted record of Roll No: %d\n", temp->rollNo);
    free(temp);
}

// Delete from end
```

```
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty! No record to delete.\n");
        return;
    }

    struct Student* temp = head;

    // If only one node
    if (head->next == NULL) {
        printf("Deleted record of Roll No: %d\n", head->rollNo);
        free(head);
        head = NULL;
        return;
    }

    // Traverse to second last node
    while (temp->next->next != NULL)
        temp = temp->next;

    printf("Deleted record of Roll No: %d\n", temp->next->rollNo);
    free(temp->next);
    temp->next = NULL;
}

// Display all records
void displayRecords() {
    if (head == NULL) {
        printf("No records to display.\n");
        return;
    }
    struct Student* temp = head;
```

```
    printf("\nStudent Records:\n");
    printf("-------------------------------------------------------
----\n");
    while (temp != NULL) {
        printf("Roll No: %d\n", temp->rollNo);
        printf("Name   : %s\n", temp->name);
        printf("Phone  : %s\n", temp->phone);
        printf("City   : %s\n", temp->city);
        printf("HSC Marks: %.2f\n", temp->hscMarks);
        printf("-------------------------------------------------
--------\n");
        temp = temp->next;
    }
}

// Count total records
void countRecords() {
    int count = 0;
    struct Student* temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    printf("Total number of student records: %d\n", count);
}

// Search by Roll No (PRN)
void searchByPRN() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
```

```
    int key;
    printf("Enter Roll No (PRN) to search: ");
    scanf("%d", &key);

    struct Student* temp = head;
    while (temp != NULL) {
        if (temp->rollNo == key) {
            printf("Record Found:\n");
            printf("Name   : %s\n", temp->name);
            printf("Phone  : %s\n", temp->phone);
            printf("City   : %s\n", temp->city);
            printf("HSC Marks: %.2f\n", temp->hscMarks);
            return;
        }
        temp = temp->next;
    }
    printf("No record found with Roll No: %d\n", key);
}

// Menu function
void menu() {
    int choice;
    do {
        printf("\n===== Student Records Menu =====\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Delete from Beginning\n");
        printf("4. Delete from End\n");
        printf("5. Display Records\n");
        printf("6. Count Total Records\n");
        printf("7. Search by Roll No (PRN)\n");
        printf("0. Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);
        printf("\n");

        switch (choice) {
            case 1: insertAtBeginning(); break;
            case 2: insertAtEnd(); break;
            case 3: deleteFromBeginning(); break;
            case 4: deleteFromEnd(); break;
            case 5: displayRecords(); break;
            case 6: countRecords(); break;
            case 7: searchByPRN(); break;
            case 0: printf("Exiting...\n"); break;
            default: printf("Invalid choice! Try again.\n");
        }
    } while (choice != 0);
}

// Main function
int main() {
    menu();
    return 0;
}
```

11. Discuss about the Time and Space complexity of various operations on Singly Linked Lists with your friends.

Time Complexity Table of Singly Linked List Operations

Operation	**Best Case**	**Average Case**	**Worst Case**	**Notes**
Insertion at Beginning	O(1)	O(1)	O(1)	Head insertion is always constant time.

Insertion at End (with Tail)	O(1)	O(1)	O(1)	Only if we maintain a tail pointer.
Insertion at End (no Tail)	O(n)	O(n)	O(n)	Traverses the entire list to insert.
Insertion at Given Position	O(1)	O(n)	O(n)	Position near head is fast; near tail takes longer.
Deletion at Beginning	O(1)	O(1)	O(1)	Just move the head pointer.
Deletion at End	O(n)	O(n)	O(n)	Requires traversal to the second last node.
Deletion by Key (Search + Del)	O(1)	O(n)	O(n)	Best case when found at head; otherwise needs search.
Search (by Value/Key)	O(1)	O(n)	O(n)	Sequential search only, no indexing.
Traversal / Display All	O(1)	O(n)	O(n)	O(n) always if you want to print/display all nodes.
Count Nodes	O(1)*	O(n)	O(n)	*If a global count is maintained, otherwise traversal needed.

Space Complexity Table of Singly Linked List Operations

Operation	Space Complexity	Notes
Insertion	O(1)	Memory required only for one new node.
Deletion	O(1)	No extra space; just frees existing node.
Search	O(1)	No extra space used; just traversal.
Traversal / Display	O(1)	Only one pointer used at a time during traversal.
Entire List Storage	O(n)	n nodes each storing data + next pointer.