Data Structure - Module 3 - linkedlist

1. **Singly Linked List (SLL)**:
    - In a singly linked list, each node contains a data element and a pointer to the next node in the sequence.
    - The last node's pointer is set to NULL, indicating the end of the list.
    - Traversal in a singly linked list is only possible in one direction, from the head (the first node) to the last node.
2. **Doubly Linked List (DLL)**:
    - In a doubly linked list, each node contains a data element and two pointers, one to the next node and one to the previous node.
    - Both the first and last nodes have one of their pointers set to NULL.
    - Traversal in a doubly linked list can be done in both forward and backward directions.

    Advantages of DLLs over SLLs:
    - **Bidirectional Traversal**: DLLs support traversal in both forward and backward directions, allowing easier navigation through the list.
    - **Insertion and Deletion Flexibility**: DLLs facilitate efficient insertion and deletion at both the beginning and end of the list, providing more flexibility in various scenarios.
    - **Efficient Removal of Last Element**: Removing the last element in a DLL is more efficient as it doesn't require traversal through the entire list.
    - **Simpler Implementation of Reverse Operations**: DLLs simplify the implementation of reverse traversal, making it easier to perform operations like printing the list in reverse order or implementing undo functionality.
3. **Circular Singly Linked List (CSLL)**:
    - In a circular singly linked list, the last node's pointer points back to the first node, forming a circle.
    - Traversal in a circular singly linked list starts from any node and continues until you reach the starting node again.
4. **Circular Doubly Linked List (CDLL)**:
    - Each node contains a data element and two pointers, one to the next node and one to the previous node.
    - Both the first and last nodes have one of their pointers set to point to each other, forming a circle.

- Traversal in a circular doubly linked list can be done in both forward and backward directions, and it forms a continuous loop.

Linked List operation

1. insertion at front

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


typedef struct Node
{
    int data;
    struct Node* link;
} Node;


Node* create_node(int data) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    if (new_node == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    new_node->data = data;
    new_node->link = NULL;
    return new_node;
}


Node* insert_front(Node* first, int data)
{
    Node* new_node = create_node(data);
    if (new_node == NULL) {
        return first;
    }
    new_node->link = first;
    return new_node;
}
```

2. Insertion at rear

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


typedef struct Node
{
    int data;
    struct Node* link;
} Node;


Node* create_node(int data) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    if (new_node == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    new_node->data = data;
    new_node->link = NULL;
    return new_node;
}

Node* rear_insert(Node* head, int data) {

    Node* current = first;
    while (current->link != NULL)
    {
        current = current->link;
    }


    current->link = new_node;

    return first;
}
```

3. Deletion at rear

```
Node* delete_rear(Node* first) {

    if (first == NULL || first->link == NULL)
    {
        free(first);
        return NULL;
    }

    // Traverse the list to find the second last node
    Node* prev = NULL;
    Node* current = first;
    while (current->link != NULL) {
        prev = current;
        current = current->link;
    }

    // Free the last node and update the second last node's link to NULL
    free(current);
    prev->link = NULL;

    return first;
}
```

4. Display

```
void display_sll(Node* first) {
    Node* current = first;
    printf("Linked list elements: ");
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->link;
    }
    printf("NULL\n");
}
```

5. Count elements

```c
int length_of_sll(Node* first) {
    int count = 0;
    Node* current = first;
    while (current != NULL) {
        count++;
        current = current->link;
    }
    return count;
}
```

6. Search

```c
int linear_search(Node* first, int target) {
    Node* current = first;
    while (current != NULL) {
        if (current->data == target) {
            return 1; // Element found
        }
        current = current->link;
    }
    return 0;
}
```

7. Concatenate

```c
Node* concatenate_lists(Node* list1, Node* list2) {

    if (list1 == NULL) {
        return list2;
    }
    if (list2 == NULL) {
        return list1;
    }


    Node* current = list1;
    while (current->link != NULL) {
        current = current->link;
    }
```

```
    current->link = list2;


    return list1;
}
```

9. Deletion using data

```
Node* delete_node(Node* head, int target)
{
    if (head == NULL) {
        return NULL;
}



    if (head->data == target) {
        Node* temp = head;
        head = head->link;
        free(temp);
        return head;
    }



    Node* current = head;
    Node* prev = NULL;
    while (current != NULL && current->data != target)
     {
        prev = current;
        current = current->link;
    }



    if (current != NULL) {
        prev->link = current->link;
        free(current);
    }

    return head;
```

10. Insertion before and after

```c
Node* insert_between(Node* first, int data_before, int data_after, int
data_to_insert) {

    Node* new_node = (Node*)malloc(sizeof(Node));
    if (new_node == NULL) {
        printf("Memory allocation failed!\n");
        return first; // Return original first if memory allocation fails
    }
    new_node->data = data_to_insert;


    if (first == NULL) {
        new_node->link = NULL;
        return new_node;
    }


    Node* current = first;
    while (current->link != NULL && current->data != data_before) {
        current = current->link;
    }


    if (current->data == data_before) {

        Node* node_after = current->link;
        new_node->link = node_after;
        current->link = new_node;
        return first;
    } else {
        printf("Node with data %d not found!\n", data_before);
        free(new_node); // Free the memory allocated for the new node
        return first;
    }
}
```

Doubly linked list

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

typedef struct Node {
```

```c
    int data;
    struct Node* llink;
    struct Node* rlink;
} Node;

Node* head = NULL;

Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = value;
    newNode->llink = NULL;
    newNode->rlink = NULL;
    return newNode;
}

void insertFront(int value) {
    Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
        return;
    }
    newNode->rlink = head;
    head->llink = newNode;
    head = newNode;
}

void insertRear(int value) {
    Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->rlink != NULL) {
        temp = temp->rlink;
    }
    temp->rlink = newNode;
    newNode->llink = temp;
}

void insertIntermediate(int position, int value) {
    if (position < 1) {
```

```
        printf("Invalid position!\n");
        return;
    }
    if (position == 1) {
        insertFront(value);
        return;
    }
    Node* newNode = createNode(value);
    Node* temp = head;
    int count = 1;
    while (temp != NULL && count < position - 1) {
        temp = temp->rlink;
        count++;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        return;
    }
    newNode->rlink = temp->rlink;
    if (temp->rlink != NULL) {
        temp->rlink->llink = newNode;
    }
    newNode->llink = temp;
    temp->rlink = newNode;
}

void deleteFront() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    Node* temp = head;
    head = head->rlink;
    if (head != NULL) {
        head->llink = NULL;
    }
    free(temp);
}

void deleteRear() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    if (head->rlink == NULL) {
        free(head);
```

```c
        head = NULL;
        return;
    }
    Node* temp = head;
    while (temp->rlink != NULL) {
        temp = temp->rlink;
    }
    temp->llink->rlink = NULL;
    free(temp);
}

void deleteNodeWithValue(int value) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    Node* temp = head;
    while (temp != NULL && temp->data != value) {
        temp = temp->rlink;
    }
    if (temp == NULL) {
        printf("Value not found in the list!\n");
        return;
    }
    if (temp->llink != NULL) {
        temp->llink->rlink = temp->rlink;
    } else {
        head = temp->rlink;
    }
    if (temp->rlink != NULL) {
        temp->rlink->llink = temp->llink;
    }
    free(temp);
}

void display() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->rlink;
    }
```

```
    printf("\n");
}

void concatenate(Node* list2Head) {
    if (head == NULL) {
        head = list2Head;
        return;
    }
    Node* temp = head;
    while (temp->rlink != NULL) {
        temp = temp->rlink;
    }
    temp->rlink = list2Head;
    if (list2Head != NULL) {
        list2Head->llink = temp;
    }
}

int search(int key) {
    if (head == NULL) {
        printf("List is empty!\n");
        return 0;
    }
    Node* temp = head;
    int position = 1;
    while (temp != NULL && temp->data != key) {
        temp = temp->rlink;
        position++;
    }
    if (temp == NULL) {
        printf("Key not found in the list!\n");
        return 0;
    }
    printf("Key %d found at position %d.\n", key, position);
    return 1;
}

int main() {
    // Example usage of doubly linked list operations
    insertFront(5);
    insertRear(7);
    insertIntermediate(2, 6);
    display(); // Output: Doubly Linked List: 5 6 7
    search(5);
    deleteFront();
    deleteRear();
```

```
    deleteNodeWithValue(6);
    display(); // Output: List is empty!

    return 0;
}
```

LinkedList

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

#define MAX_STACKS 10 /* maximum number of stacks */

typedef struct {
    int key;
    /* other fields */
} Element;

typedef struct StackNode *StackPointer;
typedef struct StackNode {
    Element data;
    StackPointer next;
} StackNode;

StackPointer top[MAX_STACKS]; // Array of pointers to the tops of stacks

// Function to push an element onto a stack
void push(int stackNumber, Element item) {
    StackPointer newNode = (StackPointer)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    newNode->data = item;
    newNode->next = top[stackNumber];
    top[stackNumber] = newNode;
}

// Function to pop an element from a stack
Element pop(int stackNumber) {
    if (top[stackNumber] == NULL) {
        printf("Stack underflow.\n");
        exit(EXIT_FAILURE);
    }
```

```
    StackPointer temp = top[stackNumber];
    Element item = temp->data;
    top[stackNumber] = temp->next;
    free(temp);
    return item;
}

// Function to display the elements of a stack
void display(int stackNumber) {
    if (top[stackNumber] == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack %d: ", stackNumber);
    StackPointer temp = top[stackNumber];
    while (temp != NULL) {
        printf("%d ", temp->data.key); // Assuming key is an integer for
simplicity
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    // Example usage
    Element item1 = {1}; // Initialize element with key value 1
    Element item2 = {2}; // Initialize element with key value 2

    push(0, item1); // Push item1 onto stack 0
    push(0, item2); // Push item2 onto stack 0

    display(0); // Display stack 0

    Element poppedItem = pop(0); // Pop an item from stack 0
    printf("Popped item: %d\n", poppedItem.key);

    display(0); // Display stack 0 after popping

    return 0;
}
```

LinkeQueue

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

// Define a structure for queue nodes
typedef struct QueueNode {
    int data;
    struct QueueNode *next;
} QueueNode;

// Define a structure for the queue itself
typedef struct {
    QueueNode *front;
    QueueNode *rear;
} Queue;

// Function to create a new node
QueueNode* createNode(int data) {
    QueueNode *newNode = (QueueNode*)malloc(sizeof(QueueNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize the queue
void initQueue(Queue *q) {
    q->front = q->rear = NULL;
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return q->front == NULL;
}

// Function to enqueue an element into the queue
void enqueue(Queue *q, int data) {
    QueueNode *newNode = createNode(data);
    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
```

```
    }
}

// Function to dequeue an element from the queue
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        exit(EXIT_FAILURE);
    }
    int data = q->front->data;
    QueueNode *temp = q->front;
    q->front = q->front->next;
    free(temp);
    return data;
}

// Function to get the front element of the queue without removing it
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        exit(EXIT_FAILURE);
    }
    return q->front->data;
}

// Function to display the elements of the queue
void display(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    QueueNode *current = q->front;
    printf("Queue: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    Queue q;
    initQueue(&q);

    // Enqueue some elements
    enqueue(&q, 10);
```

```c
    enqueue(&q, 20);
    enqueue(&q, 30);

    // Display the queue
    display(&q);

    // Dequeue an element
    printf("Dequeued element: %d\n", dequeue(&q));

    // Display the queue after dequeue
    display(&q);

    return 0;
}
```

Poly addition using circular SLL

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

// Define a structure for polynomial terms
typedef struct PolyNode *polyPointer;
typedef struct PolyNode {
    int coef;   // Coefficient
    int expon;  // Exponent
    polyPointer link; // Pointer to the next term
} PolyNode;

// Function to attach a new term to the polynomial
void attach(int coefficient, int exponent, polyPointer *rear) {
    polyPointer temp;
    temp = (polyPointer)malloc(sizeof(struct PolyNode));
    if (temp == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    temp->coef = coefficient;
    temp->expon = exponent;
    if (*rear == NULL) {
        temp->link = temp; // For the first term, link it to itself
        *rear = temp;
    } else {
        temp->link = (*rear)->link; // Link the new term to the first term
        (*rear)->link = temp; // Update the link of the last term to point to
```

```
the new term
        *rear = temp; // Update rear to point to the new last term
    }
}

// Function to compare two exponents
#define COMPARE(x, y) ((x) == (y) ? 0 : ((x) < (y) ? -1 : 1))

// Function to add two polynomials
polyPointer padd(polyPointer a, polyPointer b) {
    polyPointer c, rear, temp;
    int sum;

    // Create a dummy node for the resulting polynomial
    c = (polyPointer)malloc(sizeof(struct PolyNode));
    if (c == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    rear = NULL;

    // Loop until both polynomials reach the end
    do {
        switch (COMPARE(a->expon, b->expon)) {
            case -1: // a->expon < b->expon
                attach(b->coef, b->expon, &rear);
                b = b->link;
                break;
            case 0: // a->expon == b->expon
                sum = a->coef + b->coef;
                if (sum != 0)
                    attach(sum, a->expon, &rear);
                a = a->link;
                b = b->link;
                break;
            case 1: // a->expon > b->expon
                attach(a->coef, a->expon, &rear);
                a = a->link;
                break;
        }
    } while (a != c && b != c); // Continue until both pointers reach the dummy
node

    // Set the link of the last node to the dummy node to make it circular
    rear->link = c;
```

```
    // Return the resulting polynomial
    return c;
}
```

Problems

`linked representation of polynomial`

`linked list representation of sparse matrix`