Folder SEM 3\Exp1

1 printable files

(file list disabled)

SEM 3\Exp1\SLL_implementation.c

```c
// program to implement singly linked list

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

// Core structure of Node that forms Linked List
struct node
{
    int data;
    struct node *next;
};

// End of the linked list should not point to anything(NULL)
struct node *head = NULL;

// Function to insert a node at the beginning of the list
void insertFirst(int data)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));

    new_node->data = data;
    new_node->next = head;

    head = new_node;
}

// Function to insert a node at the end of the list
void insertEnd(int data)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));

    new_node->data = data;
    new_node->next = NULL;

    if (head == NULL)
    {
        head = new_node;
        return;
    }

    struct node *temp = head;

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

    temp->next = new_node;
}

void insertPos(int data, int pos)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));

    new_node->data = data;

    int curr_pos = 0;
    struct node *temp = head;

    while (temp->next != NULL && curr_pos < pos - 1)
    {
        temp = temp->next;
        curr_pos++;
    }

    new_node->next = temp->next;
    temp->next = new_node;
}

void deleteFirst()
{
    if (head == NULL)
    {
        printf("List is empty");
        return;
    }

    struct node *temp = head;
    head = head->next;
    free(temp);
}

void deleteEnd()
{
    if (head == NULL)
    {
        printf("List is empty");
        return;
    }

    struct node *temp = head;
    struct node *prev = NULL;

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

    prev->next = NULL;
    free(temp);
}

void deletePos(int pos)
{
    if (head == NULL)
    {
        printf("List is empty");
        return;
    }

    struct node *temp = head;
    struct node *prev = NULL;
    int curr_pos = 0;

    while (temp->next != NULL && curr_pos < pos - 1)
    {
        prev = temp;
        temp = temp->next;
        curr_pos++;
    }

    prev->next = temp->next;
    free(temp);
}

void display()
{
    struct node *temp = head;

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

int main()
{
    printf("Linked List creation and Manipulation\n");
    printf("Enter from the following options:\n");
    printf("1. Insert at the beginning of the list\n");
    printf("2. Insert at the end of the list\n");
    printf("3. Insert at a specific position in the list\n");
    printf("4. Delete from the beginning of the list\n");
    printf("5. Delete from the end of the list\n");
    printf("6. Delete from a specific position in the list\n");
    printf("7. Display the list\n");
    printf("8. Exit\n");

    int choice;
    int data;
    int pos;

    while (1)
    {
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the data to be inserted: ");
            scanf("%d", &data);
            insertFirst(data);
            break;
        case 2:
            printf("Enter the data to be inserted: ");
            scanf("%d", &data);
            insertEnd(data);
            break;
        case 3:
            printf("Enter the data to be inserted: ");
            scanf("%d", &data);
            printf("Enter the position to insert the data: ");
            scanf("%d", &pos);
            insertPos(data, pos);
            break;
        case 4:
            deleteFirst();
            break;
        case 5:
            deleteEnd();
            break;
        case 6:
            printf("Enter the position to delete the data: ");
            scanf("%d", &pos);
            deletePos(pos);
            break;
        case 7:
            display();
            break;
        case 8:
            exit(0);
        default:
            printf("Invalid choice");
            break;
        }
    }

    return 0;
}
```

```
Linked List creation and Manipulation
Enter from the following options:
1. Insert at the beginning of the list
2. Insert at the end of the list
3. Insert at a specific position in the list
4. Delete from the beginning of the list
5. Delete from the end of the list
6. Delete from a specific position in the list
7. Display the list
8. Exit
Enter your choice: 1
Enter the data to be inserted: 5
Enter your choice: 1
Enter the data to be inserted: 8
Enter your choice: 1
Enter the data to be inserted: 6
Enter your choice: 2
Enter the data to be inserted: 7
Enter your choice: 2
Enter the data to be inserted: 14
Enter your choice: 7
6 -> 8 -> 5 -> 7 -> 14 -> NULL
Enter your choice: 4
Enter your choice: 7
8 -> 5 -> 7 -> 14 -> NULL
Enter your choice: 5
Enter your choice: 7
8 -> 5 -> 7 -> NULL
Enter your choice: 6
Enter the position to delete the data: 2
Enter your choice: 7
8 -> 7 -> NULL
Enter your choice: 8
```

Folder SEM 3\Exp2

1 printable files

(file list disabled)

SEM 3\Exp2\DLL_implementation.c

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

// Node structure for the doubly linked list
struct Node
{
    int data;
    struct Node *prev;
    struct Node *next;
};

// Insert at the end of the doubly linked list
void insert(struct Node **head_ref, int new_data)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    struct Node *last = *head_ref;
    new_node->data = new_data;
    new_node->next = NULL;

    if (*head_ref == NULL)
    {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    while (last->next != NULL)
        last = last->next;

    last->next = new_node;
    new_node->prev = last;
}

// Display the doubly linked list
void display(struct Node *node)
{
    struct Node *last;
    printf("Traversal in forward direction:\n");
    while (node != NULL)
    {
        printf("%d ", node->data);
        last = node;
        node = node->next;
    }
    printf("\n");
```

```c
}

// Delete a node from the doubly linked list
void deleteNode(struct Node **head_ref, int key)
{
    struct Node *temp = *head_ref;

    if (*head_ref == NULL)
        return;

    while (temp != NULL && temp->data != key)
        temp = temp->next;

    if (temp == NULL)
        return;

    if (*head_ref == temp)
        *head_ref = temp->next;

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    if (temp->prev != NULL)
        temp->prev->next = temp->next;

    free(temp);
}

// Search for a key in the doubly linked list
void search(struct Node *head, int key)
{
    struct Node *temp = head;
    int pos = 0;
    while (temp != NULL)
    {
        if (temp->data == key)
        {
            printf("Element %d found at position %d\n", key, pos);
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("Element %d not found in the list\n", key);
}

// Count the number of nodes in the doubly linked list
int count(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;
    while (temp != NULL)
    {
        count++;
        temp = temp->next;
    }
    return count;
}

int main()
{
    struct Node *head = NULL;
    int choice, value, key;

    while (1)
    {
        printf("\nDoubly Linked List Operations:\n");
        printf("1. Insert\n");
        printf("2. Display\n");
        printf("3. Delete\n");
        printf("4. Search\n");
        printf("5. Count\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            insert(&head, value);
            break;
        case 2:
            display(head);
            break;
        case 3:
            printf("Enter the value to delete: ");
            scanf("%d", &key);
            deleteNode(&head, key);
            break;
        case 4:
            printf("Enter the value to search: ");
            scanf("%d", &key);
            search(head, key);
            break;
        case 5:
            printf("The number of nodes in the list: %d\n", count(head));
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

```
Doubly Linked List Operations:
1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit
Enter your choice: 1
Enter the value to insert: 2

Doubly Linked List Operations:
1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit
Enter your choice: 1
Enter the value to insert: 3

Doubly Linked List Operations:
1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit
Enter your choice: 1
Enter the value to insert: 5

Doubly Linked List Operations:
1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit
Enter your choice: 1
Enter the value to insert: 7

Doubly Linked List Operations:
1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit
Enter your choice: 1
Enter the value to insert: 11
```

Folder SEM 3\Exp3

1 printable files

(file list disabled)

SEM 3\Exp3\CSLL.c

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

// Node structure for the circular linked list
struct Node
{
    int data;
    struct Node *next;
};

// Insert a new node at the end of the circular linked list
void insert(struct Node **head_ref, int new_data)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    struct Node *temp = *head_ref;
    new_node->data = new_data;
    new_node->next = *head_ref;

    if (*head_ref == NULL)
    {
        new_node->next = new_node;
        *head_ref = new_node;
        return;
    }

    while (temp->next != *head_ref)
        temp = temp->next;

    temp->next = new_node;
}

// Display the circular linked list
void display(struct Node *head)
{
    struct Node *temp = head;
    if (head != NULL)
    {
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        } while (temp != head);
        printf("\n");
    }
    else
    {
        printf("List is empty.\n");
    }
}

// Delete a node with a specific value from the circular linked list
void deleteNode(struct Node **head_ref, int key)
{
    if (*head_ref == NULL)
        return;

    struct Node *temp = *head_ref, *prev;

    // If the node to be deleted is the head
    if (temp->data == key && temp->next == *head_ref)
    {
        *head_ref = NULL;
        free(temp);
        return;
    }

    // If the node to be deleted is the head and the list has more than one node
    if (temp->data == key)
    {
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = (*head_ref)->next;
        free(*head_ref);
        *head_ref = temp->next;
        return;
    }

    // If the node to be deleted is not the head
    prev = temp;
    while (temp->next != *head_ref && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    if (temp->data == key)
    {
        prev->next = temp->next;
        free(temp);
    }
}

// Search for a specific value in the circular linked list
void search(struct Node *head, int key)
{
    struct Node *temp = head;
    int pos = 0;

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

    do
    {
        if (temp->data == key)
        {
            printf("Element %d found at position %d\n", key, pos);
            return;
        }
        temp = temp->next;
        pos++;
    } while (temp != head);

    printf("Element %d not found in the list\n", key);
}

// Count the number of nodes in the circular linked list
int count(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;

    if (head == NULL)
        return 0;

    do
    {
        count++;
        temp = temp->next;
    } while (temp != head);

    return count;
}

int main()
{
    struct Node *head = NULL;
    int choice, value, key;

    while (1)
    {
        printf("\nCircular Linked List Operations:\n");
        printf("1. Insert\n");
        printf("2. Display\n");
        printf("3. Delete\n");
        printf("4. Search\n");
        printf("5. Count\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            insert(&head, value);
            break;
        case 2:
            display(head);
            break;
        case 3:
            printf("Enter the value to delete: ");
            scanf("%d", &key);
            deleteNode(&head, key);
            break;
        case 4:
            printf("Enter the value to search: ");
            scanf("%d", &key);
            search(head, key);
            break;
        case 5:
            printf("The number of nodes in the list: %d\n", count(head));
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

```

Folder SEM 3\Exp4

1 printable files

(file list disabled)

SEM 3\Exp4\CDLL.c

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

// Node structure for the circular doubly linked list
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

// Insert a node at the end of the circular doubly linked list
void insert(struct Node **head_ref, int new_data)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;

    if (*head_ref == NULL)
    {
        new_node->next = new_node;
        new_node->prev = new_node;
        *head_ref = new_node;
        return;
    }

    struct Node *last = (*head_ref)->prev;

    new_node->next = *head_ref;
    (*head_ref)->prev = new_node;
    new_node->prev = last;
    last->next = new_node;
}

// Display the circular doubly linked list
void display(struct Node *head)
{
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = head;
    printf("Traversal in forward direction:\n");
    do
    {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("\n");

    printf("Traversal in reverse direction:\n");
    temp = head->prev;
    do
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    } while (temp->next != head);
    printf("\n");
}

// Delete a node from the circular doubly linked list
void deleteNode(struct Node **head_ref, int key)
{
    if (*head_ref == NULL)
        return;

    struct Node *current = *head_ref;

    while (current->data != key)
    {
        current = current->next;
        if (current == *head_ref)
        {
            printf("Element %d not found in the list.\n", key);
            return;
        }
    }

    if (current->next == *head_ref && current->prev == *head_ref)
    {
        *head_ref = NULL;
        free(current);
        return;
    }

    if (current == *head_ref)
    {
        struct Node *last = (*head_ref)->prev;
        *head_ref = current->next;
        last->next = *head_ref;
        (*head_ref)->prev = last;
        free(current);
        return;
    }

    current->prev->next = current->next;
    current->next->prev = current->prev;

    free(current);
}

// Search for a specific value in the circular doubly linked list
void search(struct Node *head, int key)
{
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = head;
    int pos = 0;

    do
    {
        if (temp->data == key)
        {
            printf("Element %d found at position %d\n", key, pos);
            return;
        }
        temp = temp->next;
        pos++;
    } while (temp != head);

    printf("Element %d not found in the list\n", key);
}

// Count the number of nodes in the circular doubly linked list
int count(struct Node *head)
{
    if (head == NULL)
        return 0;

    struct Node *temp = head;
    int count = 0;

    do
    {
        count++;
        temp = temp->next;
    } while (temp != head);

    return count;
}

int main()
{
    struct Node *head = NULL;
    int choice, value, key;

    while (1)
    {
        printf("\nCircular Doubly Linked List Operations:\n");
        printf("1. Insert\n");
        printf("2. Display\n");
        printf("3. Delete\n");
        printf("4. Search\n");
        printf("5. Count\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            insert(&head, value);
            break;
        case 2:
            display(head);
            break;
        case 3:
            printf("Enter the value to delete: ");
            scanf("%d", &key);
            deleteNode(&head, key);
            break;
        case 4:
            printf("Enter the value to search: ");
            scanf("%d", &key);
            search(head, key);
            break;
        case 5:
            printf("The number of nodes in the list: %d\n", count(head));
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

Folder SEM 3\Exp5

2 printable files

(file list disabled)

SEM 3\Exp5\Stack_ARR.c

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

#define MAX 100 // Maximum size of the stack

// Stack structure using arrays
struct StackArray
{
    int top;
    int arr[MAX];
};

// Function to create a stack
struct StackArray *createStack()
{
    struct StackArray *stack = (struct StackArray *)malloc(sizeof(struct
StackArray));
    stack->top = -1; // Initialize the top index
    return stack;
}

// Check if the stack is full
int isFull(struct StackArray *stack)
{
    return stack->top == MAX - 1;
}

// Check if the stack is empty
int isEmpty(struct StackArray *stack)
{
    return stack->top == -1;
}

// Push an element onto the stack
void push(struct StackArray *stack, int value)
{
    if (isFull(stack))
    {
        printf("Stack overflow!\n");
        return;
    }
    stack->arr[++stack->top] = value;
    printf("%d pushed onto stack\n", value);
}

```

```c
// Pop an element from the stack
int pop(struct StackArray *stack)
{
    if (isEmpty(stack))
    {
        printf("Stack underflow!\n");
        return -1; // Return -1 for underflow
    }
    return stack->arr[stack->top--];
}

// Peek at the top element of the stack
int peek(struct StackArray *stack)
{
    if (isEmpty(stack))
    {
        printf("Stack is empty!\n");
        return -1; // Return -1 if empty
    }
    return stack->arr[stack->top];
}

// Display the stack
void display(struct StackArray *stack)
{
    if (isEmpty(stack))
    {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = stack->top; i >= 0; i--)
    {
        printf("%d ", stack->arr[i]);
    }
    printf("\n");
}

int main()
{
    struct StackArray *stack = createStack();
    int choice, value;

    while (1)
    {
        printf("\nStack Operations (Array Implementation):\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to push: ");
            scanf("%d", &value);
            push(stack, value);
            break;
        case 2:
            value = pop(stack);
            if (value != -1)
                printf("Popped value: %d\n", value);
            break;
        case 3:
            value = peek(stack);
            if (value != -1)
                printf("Top value: %d\n", value);
            break;
        case 4:
            display(stack);
            break;
        case 5:
            free(stack);
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

```

SEM 3\Exp5\Stack_LL.c

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

// Node structure for the linked list
struct Node
{
    int data;
    struct Node *next;
};

// Stack structure using linked lists
struct StackLinkedList
{
    struct Node *top;
};

// Function to create a stack
struct StackLinkedList *createStack()
{
    struct StackLinkedList *stack = (struct StackLinkedList *)malloc(sizeof(struct
StackLinkedList));
```

```c
    stack->top = NULL; // Initialize the top pointer
    return stack;
}

// Check if the stack is empty
int isEmpty(struct StackLinkedList *stack)
{
    return stack->top == NULL;
}

// Push an element onto the stack
void push(struct StackLinkedList *stack, int value)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = value;
    new_node->next = stack->top;
    stack->top = new_node;
    printf("%d pushed onto stack\n", value);
}

// Pop an element from the stack
int pop(struct StackLinkedList *stack)
{
    if (isEmpty(stack))
    {
        printf("Stack underflow!\n");
        return -1; // Return -1 for underflow
    }
    struct Node *temp = stack->top;
    int popped_value = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return popped_value;
}

// Peek at the top element of the stack
int peek(struct StackLinkedList *stack)
{
    if (isEmpty(stack))
    {
        printf("Stack is empty!\n");
        return -1; // Return -1 if empty
    }
    return stack->top->data;
}

// Display the stack
void display(struct StackLinkedList *stack)
{
    if (isEmpty(stack))
    {
        printf("Stack is empty!\n");
        return;
    }
    struct Node *temp = stack->top;
    printf("Stack elements: ");
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main()
{
    struct StackLinkedList *stack = createStack();
    int choice, value;

    while (1)
    {
        printf("\nStack Operations (Linked List Implementation):\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to push: ");
            scanf("%d", &value);
            push(stack, value);
            break;
        case 2:
            value = pop(stack);
            if (value != -1)
                printf("Popped value: %d\n", value);
            break;
        case 3:
            value = peek(stack);
            if (value != -1)
                printf("Top value: %d\n", value);
            break;
        case 4:
            display(stack);
            break;
        case 5:
            // Free linked list nodes (cleanup)
            while (!isEmpty(stack))
            {
                pop(stack);
            }
            free(stack);
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

```

Folder SEM 3\Exp6

2 printable files

(file list disabled)

SEM 3\Exp6\Queue_ARR.c

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

#define MAX 100 // Maximum size of the queue

// Queue structure using arrays
struct QueueArray
{
    int front, rear;
    int arr[MAX];
};

// Function to create a queue
struct QueueArray *createQueue()
{
    struct QueueArray *queue = (struct QueueArray *)malloc(sizeof(struct
QueueArray));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

// Check if the queue is full
int isFull(struct QueueArray *queue)
{
    return queue->rear == MAX - 1;
}

// Check if the queue is empty
int isEmpty(struct QueueArray *queue)
{
    return queue->front == -1 || queue->front > queue->rear;
}

// Enqueue an element into the queue
void enqueue(struct QueueArray *queue, int value)
{
    if (isFull(queue))
    {
        printf("Queue overflow!\n");
        return;
    }
    if (isEmpty(queue))
    {
        queue->front = 0; // Initialize front if queue was empty
    }
    queue->arr[++queue->rear] = value;
    printf("%d enqueued to queue\n", value);
}

// Dequeue an element from the queue
int dequeue(struct QueueArray *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue underflow!\n");
        return -1; // Return -1 for underflow
    }
    return queue->arr[queue->front++];
}

// Peek at the front element of the queue
int peek(struct QueueArray *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue is empty!\n");
        return -1; // Return -1 if empty
    }
    return queue->arr[queue->front];
}

// Display the queue
void display(struct QueueArray *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = queue->front; i <= queue->rear; i++)
    {
        printf("%d ", queue->arr[i]);
    }
    printf("\n");
}

int main()
{
    struct QueueArray *queue = createQueue();
    int choice, value;

    while (1)
    {
        printf("\nQueue Operations (Array Implementation):\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to enqueue: ");
            scanf("%d", &value);
            enqueue(queue, value);
            break;
        case 2:
            value = dequeue(queue);
            if (value != -1)
                printf("Dequeued value: %d\n", value);
            break;
        case 3:
            value = peek(queue);
            if (value != -1)
                printf("Front value: %d\n", value);
            break;
        case 4:
            display(queue);
            break;
        case 5:
            free(queue);
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

```

SEM 3\Exp6\Queue_LL.c

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

// Node structure for the linked list
struct Node
{
    int data;
    struct Node *next;
};

// Queue structure using linked lists
struct QueueLinkedList
{
    struct Node *front;
    struct Node *rear;
```

```c
};

// Function to create a queue
struct QueueLinkedList *createQueue()
{
    struct QueueLinkedList *queue = (struct QueueLinkedList *)malloc(sizeof(struct
QueueLinkedList));
    queue->front = queue->rear = NULL; // Initialize front and rear
    return queue;
}

// Check if the queue is empty
int isEmpty(struct QueueLinkedList *queue)
{
    return queue->front == NULL;
}

// Enqueue an element into the queue
void enqueue(struct QueueLinkedList *queue, int value)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = value;
    new_node->next = NULL;

    if (isEmpty(queue))
    {
        queue->front = queue->rear = new_node; // First node
        printf("%d enqueued to queue\n", value);
        return;
    }

    queue->rear->next = new_node; // Add new node at the end
    queue->rear = new_node;       // Update the rear pointer
    printf("%d enqueued to queue\n", value);
}

// Dequeue an element from the queue
int dequeue(struct QueueLinkedList *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue underflow!\n");
        return -1; // Return -1 for underflow
    }
    struct Node *temp = queue->front;
    int dequeued_value = temp->data;
    queue->front = queue->front->next;

    // If the front becomes NULL, set rear to NULL as well
    if (queue->front == NULL)
    {
        queue->rear = NULL;
    }

    free(temp);
    return dequeued_value;
}

// Peek at the front element of the queue
int peek(struct QueueLinkedList *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue is empty!\n");
        return -1; // Return -1 if empty
    }
    return queue->front->data;
}

// Display the queue
void display(struct QueueLinkedList *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue is empty!\n");
        return;
    }
    struct Node *temp = queue->front;
    printf("Queue elements: ");
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main()
{
    struct QueueLinkedList *queue = createQueue();
    int choice, value;

    while (1)
    {
        printf("\nQueue Operations (Linked List Implementation):\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the value to enqueue: ");
            scanf("%d", &value);
            enqueue(queue, value);
            break;
        case 2:
            value = dequeue(queue);
            if (value ≠ -1)
                printf("Dequeued value: %d\n", value);
            break;
        case 3:
            value = peek(queue);
            if (value ≠ -1)
                printf("Front value: %d\n", value);
            break;
        case 4:
            display(queue);
            break;
        case 5:
            // Free linked list nodes (cleanup)
            while (!isEmpty(queue))
            {
                dequeue(queue);
            }
            free(queue);
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

```

Folder SEM 3\Exp7

2 printable files

(file list disabled)

SEM 3\Exp7\Bin_Search.c

```c
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int size, int key)
{
    int left = 0;
    int right = size - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (arr[mid] == key)
        {
            return mid; // Return the index of the found element
        }
        if (arr[mid] < key)
        {
            left = mid + 1; // Search in the right half
        }
        else
        {
            right = mid - 1; // Search in the left half
        }
    }
    return -1; // Return -1 if the element is not found
}

int main()
{
    int arr[] = {1, 2, 5, 5, 6, 9}; // Note: Array must be sorted for binary search
    int size = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the element to search for (Binary Search): ");
    scanf("%d", &key);

    int index = binarySearch(arr, size, key);
    if (index != -1)
    {
        printf("Element %d found at index %d.\n", key, index);
    }
    else
    {
        printf("Element %d not found in the array.\n", key);
    }

    return 0;
}

```

SEM 3\Exp7\Linear_Search.c

```c
#include <stdio.h>

// Function to perform linear search
int linearSearch(int arr[], int size, int key)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == key)
        {
            return i; // Return the index of the found element
        }
    }
    return -1; // Return -1 if the element is not found
}

int main()
{
    int arr[] = {5, 2, 9, 1, 5, 6};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the element to search for (Linear Search): ");
    scanf("%d", &key);

    int index = linearSearch(arr, size, key);
    if (index != -1)
    {
        printf("Element %d found at index %d.\n", key, index);
    }
    else
    {
        printf("Element %d not found in the array.\n", key);
    }

    return 0;
}

```

Folder SEM 3\Exp8

7 printable files

(file list disabled)

SEM 3\Exp8\Makefile

```
# Compiler to use
CC = gcc

# Compiler flags
CFLAGS = -Wall -Wextra -g

# Object files to compile
OBJS = main.o bubble_Sort.o insertion_Sort.o Selection_sort.o quick_sort.o
shell_Sort.o

# The final executable name
TARGET = Exp8_sorting_program

# Default target to build the executable
all: $(TARGET)

# Rule to link object files into the final executable
$(TARGET): $(OBJS)
	$(CC) -o $(TARGET) $(OBJS)

# Rule to compile each .c file into a .o file
%.o: %.c
	$(CC) $(CFLAGS) -c $<

# Clean target to remove object files and the executable
clean:
	rm -f $(OBJS) $(TARGET)

```

SEM 3\Exp8\Selection_sort.c

```c
// Function to perform selection sort
void selectionSort(int arr[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < size; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j; // Find the index of the minimum element
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

```

SEM 3\Exp8\bubble_Sort.c

```c
#include <stdio.h>

// Function to perform bubble sort
void bubbleSort(int arr[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        for (int j = 0; j < size - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to display the array
void display(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

```

SEM 3\Exp8\insertion_Sort.c

```c
// Function to perform insertion sort
void insertionSort(int arr[], int size)
{
    for (int i = 1; i < size; i++)
    {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key to one position ahead
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

SEM 3\Exp8\main.c

```c
#include <stdio.h>

// Function declarations for sorting algorithms
void bubbleSort(int arr[], int size);
void insertionSort(int arr[], int size);
void selectionSort(int arr[], int size);
void quickSort(int arr[], int low, int high);
void shellSort(int arr[], int size);

// Function to display the array
void display(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr1[] = {64, 34, 25, 12, 22, 11, 90};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);

    printf("Original array for Bubble Sort: ");
    display(arr1, size1);
    bubbleSort(arr1, size1);
    printf("Sorted array using Bubble Sort: ");
    display(arr1, size1);

    // Reset the array for next sorting
    int arr2[] = {64, 34, 25, 12, 22, 11, 90};
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    printf("\nOriginal array for Insertion Sort: ");
    display(arr2, size2);
    insertionSort(arr2, size2);
    printf("Sorted array using Insertion Sort: ");
    display(arr2, size2);

    // Reset the array for next sorting
    int arr3[] = {64, 34, 25, 12, 22, 11, 90};
    int size3 = sizeof(arr3) / sizeof(arr3[0]);

    printf("\nOriginal array for Selection Sort: ");
    display(arr3, size3);
    selectionSort(arr3, size3);
    printf("Sorted array using Selection Sort: ");
    display(arr3, size3);

    // Reset the array for next sorting
    int arr4[] = {64, 34, 25, 12, 22, 11, 90};
    int size4 = sizeof(arr4) / sizeof(arr4[0]);

    printf("\nOriginal array for Quick Sort: ");
    display(arr4, size4);
    quickSort(arr4, 0, size4 - 1);
    printf("Sorted array using Quick Sort: ");
    display(arr4, size4);

    // Reset the array for next sorting
    int arr5[] = {64, 34, 25, 12, 22, 11, 90};
    int size5 = sizeof(arr5) / sizeof(arr5[0]);

    printf("\nOriginal array for Shell Sort: ");
    display(arr5, size5);
    shellSort(arr5, size5);
    printf("Sorted array using Shell Sort: ");
    display(arr5, size5);

    return 0;
}

```

SEM 3\Exp8\quick_sort.c

```c
// Function to perform quick sort
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // Choosing the rightmost element as pivot
    int i = (low - 1);     // Index of smaller element

    for (int j = low; j < high; j++)
    {
        // If the current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // Increment index of smaller element
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap the pivot element with the element at i + 1
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1; // Return the partitioning index
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);  // Partitioning index
        quickSort(arr, low, pi - 1);         // Recursively sort elements before partition
        quickSort(arr, pi + 1, high);        // Recursively sort elements after partition
    }
}

```

SEM 3\Exp8\shell_Sort.c

```c
// Function to perform shell sort
void shellSort(int arr[], int size)
{
    for (int gap = size / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < size; i++)
        {
            int temp = arr[i];
            int j;

            // Shift earlier gap-sorted elements up until the correct location for arr[i] is found
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

```

Folder SEM 3\Exp9

7 printable files

(file list disabled)

SEM 3\Exp9\Bucket_Sort.c

```c
// Function to perform bucket sort
void bucketSort(float arr[], int size)
{
    // Create buckets
    int bucketCount = 10;                // Number of buckets
    float buckets[bucketCount][size]; // 2D array to hold buckets
    int bucketSizes[bucketCount];      // Array to hold the size of each bucket

    // Initialize bucket sizes to 0
    for (int i = 0; i < bucketCount; i++)
        bucketSizes[i] = 0;

    // Insert elements into buckets
    for (int i = 0; i < size; i++)
    {
        int bucketIndex = (int)(bucketCount * arr[i]);              // Determine bucket index
        buckets[bucketIndex][bucketSizes[bucketIndex]++] = arr[i]; // Place element in bucket
    }

    // Sort individual buckets
    for (int i = 0; i < bucketCount; i++)
    {
        if (bucketSizes[i] > 0)
        {
            // Sort the bucket (using insertion sort here)
            insertionSort(buckets[i], bucketSizes[i]);
        }
    }

    // Concatenate all buckets into the original array
    int index = 0;
    for (int i = 0; i < bucketCount; i++)
    {
        for (int j = 0; j < bucketSizes[i]; j++)
        {
            arr[index++] = buckets[i][j];
        }
    }
}

```

SEM 3\Exp9\Heap_Sort.c

```c
// Function to heapify a subtree rooted at index i
```

```c
void heapify(int arr[], int size, int i)
{
    int largest = i;       // Initialize largest as root
    int left = 2 * i + 1;  // left = 2*i + 1
    int right = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (left < size && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < size && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i)
    {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected subtree
        heapify(arr, size, largest);
    }
}

// Function to perform heap sort
void heapSort(int arr[], int size)
{
    // Build heap (rearrange array)
    for (int i = size / 2 - 1; i >= 0; i--)
        heapify(arr, size, i);

    // One by one extract an element from heap
    for (int i = size - 1; i > 0; i--)
    {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

```

SEM 3\Exp9\Makefile

```makefile
# Compiler to use
CC = gcc

# Compiler flags
CFLAGS = -Wall -Wextra -g
```

```

# Object files to compile
OBJS = main.o Bucket_Sort.o counting_sort.o Heap_Sort.o Merge_Sort.o Radix_sort.o

# The final executable name
TARGET = Exp9_sorting_program

# Default target to build the executable
all: $(TARGET)

# Rule to link object files into the final executable
$(TARGET): $(OBJS)
	$(CC) -o $(TARGET) $(OBJS)

# Rule to compile each .c file into a .o file
%.o: %.c
	$(CC) $(CFLAGS) -c $<

# Clean target to remove object files and the executable
clean:
	rm -f $(OBJS) $(TARGET)

```

SEM 3\Exp9\Merge_Sort.c

```c
#include <stdio.h>

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays
    i = 0;    // Initial index of first subarray
    j = 0;    // Initial index of second subarray
    k = left; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[], if any
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[], if any
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to perform merge sort
void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

```

SEM 3\Exp9\Radix_sort.c

```c
// Function to get the maximum value in an array
int getMax(int arr[], int size)
{
    int max = arr[0];
    for (int i = 1; i < size; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// Function to perform counting sort based on a specific digit
void countingSort(int arr[], int size, int exp)
```

```c
{
    int output[size];
    int count[10] = {0}; // Initialize count array

    // Store the count of occurrences in count[]
    for (int i = 0; i < size; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] contains actual position of this digit in output[]
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (int i = size - 1; i >= 0; i--)
    {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now contains sorted numbers
    for (int i = 0; i < size; i++)
        arr[i] = output[i];
}

// Function to perform radix sort
void radixSort(int arr[], int size)
{
    // Get the maximum number to know the number of digits
    int max = getMax(arr, size);

    // Apply counting sort to sort elements based on each digit
    for (int exp = 1; max / exp > 0; exp *= 10)
        countingSort(arr, size, exp);
}

```

SEM 3\Exp9\counting_sort.c

```c
// Function to perform counting sort
void countingSort(int arr[], int size)
{
    int output[size];
    int count[100] = {0}; // Assuming the range of input numbers is known (0-99)

    // Store the count of occurrences
    for (int i = 0; i < size; i++)
        count[arr[i]]++;

    // Build the output array
    for (int i = 0, j = 0; i < 100; i++)
    {
        while (count[i] > 0)
        {
            output[j++] = i;
```

```
                count[i]--;
            }
        }

        // Copy the output array to arr[]
        for (int i = 0; i < size; i++)
            arr[i] = output[i];
}

```

SEM 3\Exp9\main.c

```
#include <stdio.h>

// Function declarations (you can also include headers for better organization)
void mergeSort(int arr[], int left, int right);
void radixSort(int arr[], int size);
void countingSort(int arr[], int size);
void bucketSort(float arr[], int size);
void heapSort(int arr[], int size);

// Function to display the array (add this in main.c)
void display(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    // Array for testing sorting algorithms
    int arr1[] = {64, 34, 25, 12, 22, 11, 90};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);

    // Merge Sort
    printf("Original array for Merge Sort: ");
    display(arr1, size1);
    mergeSort(arr1, 0, size1 - 1);
    printf("Sorted array using Merge Sort: ");
    display(arr1, size1);

    // Reset the array for next sorting
    int arr2[] = {64, 34, 25, 12, 22, 11, 90};
    printf("\nOriginal array for Radix Sort: ");
    display(arr2, size1);
    radixSort(arr2, size1);
    printf("Sorted array using Radix Sort: ");
    display(arr2, size1);

    // Reset the array for next sorting
    int arr3[] = {64, 34, 25, 12, 22, 11, 90};
    printf("\nOriginal array for Counting Sort: ");
    display(arr3, size1);
```

```c
    countingSort(arr3, size1);
    printf("Sorted array using Counting Sort: ");
    display(arr3, size1);

    // Reset the array for next sorting
    float arr4[] = {0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23};
    int size4 = sizeof(arr4) / sizeof(arr4[0]);
    printf("\nOriginal array for Bucket Sort: ");
    display(arr4, size4);
    bucketSort(arr4, size4);
    printf("Sorted array using Bucket Sort: ");
    display(arr4, size4);

    // Reset the array for next sorting
    int arr5[] = {64, 34, 25, 12, 22, 11, 90};
    int size5 = sizeof(arr5) / sizeof(arr5[0]);
    printf("\nOriginal array for Heap Sort: ");
    display(arr5, size5);
    heapSort(arr5, size5);
    printf("Sorted array using Heap Sort: ");
    display(arr5, size5);

    return 0;
}

```