R23 Data Structures Lab manual

S.No	Name of the Experiment
1	Array Reverse
2	Search for an element using Linear search
3	Bubble Sort
4	Non-recursive Binary search
5	C program that implements the Insertion sort
6	C program that implements the Selection sort
7	Write a C program to Insert an element at End in Singly Linked List
8	Write a C program to Insert an element at Begin and Delete at End in Singly Linked List.
9	Write a C program to reverse the Singly Linked List.
10	Reverse of a single linked list recursively.
11	Single Linked List operations
12	Write code for removing duplicate elements in Singly Linked List
13	Polynomial Operations - Creating and Printing Polynomials using Linked Lists
14	Polynomial Operations - Adding Polynomials using Linked List
15	Write a Program to Insert an element at End and Traverse the Nodes in Doubly Linked List
16	Implement double linked list
17	Double Linked List Operations
18	Write Code for insertAtBeginInCLL() and countInCLL() functions in CLL
19	C program which performs all operations in Circular linked list.
20	Implementation of double ended queue using linked list - inject, eject and display operations
21	Stack using Arrays
22	Write a C program to implement different Operations on Stack using Linked Lists
23	Write a C program to evaluate a Postfix expression
24	Check for the balanced parenthesis using a stack
25	C program to implement Operations on Queue using static Array
26	Queue using Linked Lists
27	Simulation of a simple printer queue system.
28	Circular Queues using arrays
29	Implementation of Circular Queue using Linked List
30	Convert an Infix Expression to Postfix and Evaluate it.
31	Check for Symmetry of a String using Stack

32	Check for Symmetry of a String using Queue
33	Program to insert into BST and traversal using In-order, Pre-order and Post-order
34	To implement binary search tree using Linked List.
35	Collision resolution techniques : linear probing
36	Collision resolution techniques - Separate chaining
37	Implement a simple cache using hashing.

Exercise 2: Linked List Implementation

1. Implement a singly linked list and perform insertion and deletion operations.

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // Set the data and next pointer of the new node
    new_node->data = new_data;
    new_node->next = *head_ref;

    // Update the head to point to the new node
    *head_ref = new_node;
}

// Function to delete a node with a given key from the linked list
void deleteNode(struct Node** head_ref, int key) {
    // Store head node
    struct Node* temp = *head_ref;
    struct Node* prev = NULL;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head_ref = temp->next; // Changed head
        free(temp);             // Free old head
        return;
    }
```

```
    // Search for the key to be deleted, keep track of the previous node as we need to change
'prev->next'
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL)
        return;

    // Unlink the node from linked list
    prev->next = temp->next;
    free(temp); // Free memory
}

// Function to print the linked list
void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    // Initialize an empty linked list
    struct Node* head = NULL;

    // Insert elements into the linked list
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    // Print the original linked list
    printf("Original Linked List: ");
    printLinkedList(head);

    // Delete an element from the linked list
    deleteNode(&head, 2);

    // Print the modified linked list
    printf("Modified Linked List: ");
    printLinkedList(head);
```

```
    return 0;
}
```

OUTPUT:
Original Linked List: 1 2 3
Modified Linked List: 1 3

2.a)

Develop a program to reverse a linked list iteratively and recursively.

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // Set the data and next pointer of the new node
    new_node->data = new_data;
    new_node->next = *head_ref;

    // Update the head to point to the new node
    *head_ref = new_node;
}

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

    while (current != NULL) {
        // Store the next node
        next = current->next;

        // Reverse the current node's pointer
        current->next = prev;

        // Move pointers one position ahead
        prev = current;
```

```
        current = next;
    }

    // Update the head to point to the new first node
    *head_ref = prev;
}

// Function to print the linked list
void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    // Initialize an empty linked list
    struct Node* head = NULL;

    // Insert elements into the linked list
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    // Print the original linked list
    printf("Original Linked List: ");
    printLinkedList(head);

    // Reverse the linked list
    reverseLinkedList(&head);

    // Print the reversed linked list
    printf("Reversed Linked List: ");
    printLinkedList(head);

    return 0;
}
```

OUTPUT:
Original Linked List: 1 2 3
Reversed Linked List: 3 2 1

2.b) Develop a program to reverse a linked list recursively.

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

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

struct Node* head = NULL;

void insert(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

void reverse(struct Node* curr, struct Node* prev) {
    if (curr == NULL) {
        head = prev;
        return;
    }
    struct Node* nextNode = curr->next;
    curr->next = prev;
    reverse(nextNode, curr);
}

void reverseLinkedList() {
    if (head == NULL)
        return;
    reverse(head, NULL);
}

void printList() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

```
int main() {
    insert(1);
    insert(2);
    insert(3);
    insert(4);
    insert(5);

    printf("Original Linked List: ");
    printList();

    reverseLinkedList();

    printf("Reversed Linked List: ");
    printList();

    return 0;
}
```

OUTPUT:
Original Linked List: 5 4 3 2 1
Reversed Linked List: 1 2 3 4 5

3. Solve problems involving linked list traversal and manipulation.

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

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

void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
}

int main() {
    // Create a sample linked list
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 1;
    head->next = NULL;
```

```
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    second->data = 2;
    second->next = NULL;
    head->next = second;

    struct Node* third = (struct Node*)malloc(sizeof(struct Node));
    third->data = 3;
    third->next = NULL;
    second->next = third;

    // Print the elements of the linked list
    printLinkedList(head);

    return 0;
}
```

OUTPUT:
1 2 3

Exercise 3: Linked List Applications
1)Create a program to detect and remove duplicates from a linked list

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // Set the data and next pointer of the new node
    new_node->data = new_data;
    new_node->next = *head_ref;

    // Update the head to point to the new node
    *head_ref = new_node;
}
```

```
// Function to remove duplicates from the linked list
void removeDuplicates(struct Node* head) {
    struct Node* current = head;
    struct Node* next_next;

    // If the linked list is empty, return
    if (current == NULL)
        return;

    // Traverse the list till the last node
    while (current->next != NULL) {
        // Compare current node with the next node
        if (current->data == current->next->data) {
            // The next_next pointer now points to the node after the next node
            next_next = current->next->next;

            // Free the memory allocated to the next node
            free(current->next);

            // Unlink the duplicate node from the list
            current->next = next_next;
        } else {
            // Move to the next node if there is no duplicate
            current = current->next;
        }
    }
}

// Function to print the linked list
void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    // Initialize an empty linked list
    struct Node* head = NULL;

    // Insert elements into the linked list
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
```

```
    insertAtBeginning(&head, 1);
    insertAtBeginning(&head, 1);

    // Print the original linked list
    printf("Original Linked List: ");
    printLinkedList(head);

    // Remove duplicates from the linked list
    removeDuplicates(head);

    // Print the linked list after removing duplicates
    printf("Linked List after removing duplicates: ");
    printLinkedList(head);

    return 0;
}
```

OUTPUT:

Original Linked List: 1 1 2 3 3

Linked List after removing duplicates: 1 2 3

ii) Implement a linked list to represent polynomials and perform addition.

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

// Define the structure for a node representing a term in a polynomial
struct Node {
    int coefficient;
    int exponent;
    struct Node* next;
};

// Function to insert a new term into the polynomial
void insertTerm(struct Node** head_ref, int coeff, int exp) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->coefficient = coeff;
    new_node->exponent = exp;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

// Function to add two polynomials
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;
    struct Node* temp = NULL;
    struct Node* prev = NULL;
```

```
    while (poly1 != NULL && poly2 != NULL) {
        // If the exponents are equal, add the coefficients
        if (poly1->exponent == poly2->exponent) {
            int coeff_sum = poly1->coefficient + poly2->coefficient;
            insertTerm(&result, coeff_sum, poly1->exponent);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
        // If the exponent of the first polynomial is greater, add its term
        else if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        }
        // If the exponent of the second polynomial is greater, add its term
        else {
            insertTerm(&result, poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        }
    }

    // Add remaining terms of poly1, if any
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }

    // Add remaining terms of poly2, if any
    while (poly2 != NULL) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }

    // Reverse the result list
    struct Node* current = result;
    struct Node* next = NULL;
    struct Node* prev_node = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev_node;
        prev_node = current;
        current = next;
    }
    result = prev_node;

    return result;
}
```

```
// Function to print the polynomial
void printPolynomial(struct Node* poly) {
    struct Node* current = poly;
    while (current != NULL) {
        printf("(%dx^%d) ", current->coefficient, current->exponent);
        current = current->next;
        if (current != NULL)
            printf("+ ");
    }
    printf("\n");
}

int main() {
    // Initialize polynomial 1: 3x^2 + 2x + 1
    struct Node* poly1 = NULL;
    insertTerm(&poly1, 3, 2);
    insertTerm(&poly1, 2, 1);
    insertTerm(&poly1, 1, 0);

    // Initialize polynomial 2: 4x^3 + 2x^2 + 1
    struct Node* poly2 = NULL;
    insertTerm(&poly2, 4, 3);
    insertTerm(&poly2, 2, 2);
    insertTerm(&poly2, 1, 0);

    printf("Polynomial 1: ");
    printPolynomial(poly1);
    printf("Polynomial 2: ");
    printPolynomial(poly2);

    struct Node* result = addPolynomials(poly1, poly2);

    printf("Resultant Polynomial: ");
    printPolynomial(result);

    return 0;
}
```

OUTPUT:

```
Polynomial 1: (1x^0) + (2x^1) + (3x^2)
Polynomial 2: (1x^0) + (2x^2) + (4x^3)
Resultant Polynomial: (2x^0) + (2x^2) + (4x^3) + (2x^1) + (3x^2)
```

iii) Implement a double-ended queue (deque) with essential operations.

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

```
// Define the structure for a node
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to add an element to the front of the deque
void addToFront(struct Node** front_ref, struct Node** rear_ref, int data) {
    struct Node* newNode = createNode(data);
    if (*front_ref == NULL) {
        *front_ref = newNode;
        *rear_ref = newNode;
    } else {
        newNode->next = *front_ref;
        (*front_ref)->prev = newNode;
        *front_ref = newNode;
    }
}

// Function to add an element to the rear of the deque
void addToRear(struct Node** front_ref, struct Node** rear_ref, int data) {
    struct Node* newNode = createNode(data);
    if (*rear_ref == NULL) {
        *front_ref = newNode;
        *rear_ref = newNode;
    } else {
        newNode->prev = *rear_ref;
        (*rear_ref)->next = newNode;
        *rear_ref = newNode;
    }
}
```

```
// Function to remove an element from the front of the deque
int removeFromFront(struct Node** front_ref, struct Node** rear_ref) {
    if (*front_ref == NULL) {
        printf("Deque is empty.\n");
        exit(1);
    }
    int data = (*front_ref)->data;
    struct Node* temp = *front_ref;
    if (*front_ref == *rear_ref) {
        *front_ref = NULL;
        *rear_ref = NULL;
    } else {
        *front_ref = (*front_ref)->next;
        (*front_ref)->prev = NULL;
    }
    free(temp);
    return data;
}

// Function to remove an element from the rear of the deque
int removeFromRear(struct Node** front_ref, struct Node** rear_ref) {
    if (*rear_ref == NULL) {
        printf("Deque is empty.\n");
        exit(1);
    }
    int data = (*rear_ref)->data;
    struct Node* temp = *rear_ref;
    if (*front_ref == *rear_ref) {
        *front_ref = NULL;
        *rear_ref = NULL;
    } else {
        *rear_ref = (*rear_ref)->prev;
        (*rear_ref)->next = NULL;
    }
    free(temp);
    return data;
}

// Function to print the elements of the deque
void printDeque(struct Node* front) {
    printf("Deque: ");
    struct Node* current = front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
```

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

int main() {
    struct Node* front = NULL;
    struct Node* rear = NULL;

    // Adding elements to the front of the deque
    addToFront(&front, &rear, 1);
    addToFront(&front, &rear, 2);
    addToFront(&front, &rear, 3);

    // Adding elements to the rear of the deque
    addToRear(&front, &rear, 4);
    addToRear(&front, &rear, 5);
    addToRear(&front, &rear, 6);

    printDeque(front);

    // Removing elements from the front and rear of the deque
    printf("Removed from front: %d\n", removeFromFront(&front, &rear));
    printf("Removed from rear: %d\n", removeFromRear(&front, &rear));

    printDeque(front);

    return 0;
}
```

OUTPUT:

```
Deque: 3 2 1 4 5 6
Removed from front: 3
Removed from rear: 6
Deque: 2 1 4 5
```

Exercise 4: Double Linked List Implementation

i) Implement a doubly linked list and perform various operations to understand its properties and applications

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

// Define the structure for a node in the doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
```

```
// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a new node at the beginning of the doubly linked list
void insertAtBeginning(struct Node** head_ref, int data) {
    struct Node* newNode = createNode(data);
    if (*head_ref != NULL) {
        (*head_ref)->prev = newNode;
    }
    newNode->next = *head_ref;
    *head_ref = newNode;
}

// Function to insert a new node after a given node in the doubly linked list
void insertAfter(struct Node* prev_node, int data) {
    if (prev_node == NULL) {
        printf("Previous node cannot be NULL.\n");
        return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = prev_node->next;
    if (prev_node->next != NULL) {
        prev_node->next->prev = newNode;
    }
    prev_node->next = newNode;
    newNode->prev = prev_node;
}

// Function to insert a new node at the end of the doubly linked list
void insertAtEnd(struct Node** head_ref, int data) {
    struct Node* newNode = createNode(data);
    if (*head_ref == NULL) {
        *head_ref = newNode;
        return;
    }
```

```
    struct Node* last = *head_ref;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
    newNode->prev = last;
}

// Function to delete a node from the doubly linked list
void deleteNode(struct Node** head_ref, struct Node* del) {
    if (*head_ref == NULL || del == NULL) {
        printf("List is empty or node to be deleted is NULL.\n");
        return;
    }
    if (*head_ref == del) {
        *head_ref = del->next;
    }
    if (del->next != NULL) {
        del->next->prev = del->prev;
    }
    if (del->prev != NULL) {
        del->prev->next = del->next;
    }
    free(del);
}

// Function to print the doubly linked list in forward order
void printListForward(struct Node* head) {
    printf("Forward: ");
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Function to print the doubly linked list in reverse order
void printListReverse(struct Node* head) {
    printf("Reverse: ");
    struct Node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    while (current != NULL) {
        printf("%d ", current->data);
```

```
        current = current->prev;
    }
    printf("\n");
}

int main() {
    // Initialize an empty doubly linked list
    struct Node* head = NULL;

    // Insert elements into the doubly linked list
    insertAtBeginning(&head, 1);
    insertAtEnd(&head, 2);
    insertAfter(head->next, 3);

    // Print the doubly linked list in forward and reverse order
    printListForward(head);
    printListReverse(head);

    // Delete a node from the doubly linked list
    deleteNode(&head, head->next->next);

    // Print the doubly linked list after deletion
    printListForward(head);

    return 0;
}
```

OUTPUT:

```
Forward: 1 2 3
Reverse: 3 2 1
Forward: 1 2
```

ii) Implement a circular linked list and perform insertion, deletion, and traversal.

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

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

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
```

```
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a new node at the beginning of the circular linked list
void insertAtBeginning(struct Node** head_ref, int data) {
    struct Node* newNode = createNode(data);
    if (*head_ref == NULL) {
        newNode->next = newNode; // Point to itself if list is empty
    } else {
        struct Node* last = *head_ref;
        while (last->next != *head_ref) {
            last = last->next;
        }
        last->next = newNode; // Link the last node to the new node
        newNode->next = *head_ref; // Link the new node to the head
    }
    *head_ref = newNode; // Update the head to point to the new node
}

// Function to delete a node from the circular linked list
void deleteNode(struct Node** head_ref, int key) {
    if (*head_ref == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* current = *head_ref;
    struct Node* prev = NULL;

    // If the node to be deleted is the head node
    if (current->data == key) {
        struct Node* last = *head_ref;
        while (last->next != *head_ref) {
            last = last->next;
        }
        last->next = current->next; // Update the next pointer of the last node
        *head_ref = current->next; // Update the head to point to the next node
        free(current); // Free memory of the deleted node
        return;
    }

    // Search for the node with the given key
```

```
    do {
        prev = current;
        current = current->next;
    } while (current != *head_ref && current->data != key);

    // If the node with the given key is found
    if (current != *head_ref) {
        prev->next = current->next; // Update the next pointer of the previous node
        free(current); // Free memory of the deleted node
    } else {
        printf("Node with key %d not found.\n", key);
    }
}

// Function to print the circular linked list
void printList(struct Node* head) {
    struct Node* current = head;
    if (current == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Circular Linked List: ");
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != head);
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the circular linked list
    insertAtBeginning(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);

    // Print the circular linked list
    printList(head);

    // Delete a node from the circular linked list
    deleteNode(&head, 2);

    // Print the circular linked list after deletion
    printList(head);
```

```
    return 0;
}
```
OUTPUT:
Circular Linked List: 3 2 1
Circular Linked List: 3 1

Exercise 5: Stack Operations

i) Implement a stack using arrays and linked lists

Array:

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("**********Stack operations using array*********");

printf("\n----------------------------------------------\n");
   while(choice != 4)
   {
      printf("Chose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            show();
```

```
            break;
        }
        case 4:
        {
            printf("Exiting....");
            break;
        }
        default:
        {
            printf("Please Enter valid choice ");
        }
    };
  }
}

void push ()
{
  int val;
  if (top == n )
  printf("\n Overflow");
  else
  {
    printf("Enter the value?");
    scanf("%d",&val);
    top = top +1;
    stack[top] = val;
  }
}

void pop ()
{
  if(top == -1)
  printf("Underflow");
  else
  top = top -1;
}
void show()
{
  for (i=top;i>=0;i--)
  {
    printf("%d\n",stack[i]);
  }
  if(top == -1)
  {
    printf("Stack is empty");
  }
```

```
}
OUTPUT:
/tmp/8Qwlx2oGQD.o
Enter the number of elements in the stack 3
*********Stack operations using array*********
----------------------------------------------
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?23
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
23
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?23
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
23
23
Chose one from the below options...

1.Push
```

```
2.Pop
3.Show
4.Exit
 Enter your choice
2
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
2
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
Stack is emptyChose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
4
Exiting....
```

Using Linked List

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

// Define the structure for a node in the stack
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
```

```
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Structure to represent the stack
struct Stack {
    struct Node* top; // Pointer to the top element of the stack
};

// Function to create a new stack
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    if (stack == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    stack->top = NULL;
    return stack;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
    return (stack->top == NULL);
}

// Function to push an element onto the stack
void push(struct Stack* stack, int item) {
    struct Node* newNode = createNode(item);
    newNode->next = stack->top;
    stack->top = newNode;
    printf("%d pushed to stack\n", item);
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow. Cannot pop.\n");
        exit(1);
    }
    struct Node* temp = stack->top;
    int data = temp->data;
    stack->top = stack->top->next;
    free(temp);
```

```
    return data;
}

// Function to peek the top element of the stack without removing it
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. No top element.\n");
        exit(1);
    }
    return stack->top->data;
}

int main() {
    struct Stack* stack = createStack();

    push(stack, 1);
    push(stack, 2);
    push(stack, 3);

    printf("Top element: %d\n", peek(stack));

    printf("%d popped from stack\n", pop(stack));
    printf("%d popped from stack\n", pop(stack));

    printf("Top element: %d\n", peek(stack));

    return 0;
}
OUTPUT:
1 pushed to stack
2 pushed to stack
3 pushed to stack
Top element: 3
3 popped from stack
2 popped from stack
Top element: 1
```

b)Write a program to evaluate a postfix expression using a stack.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
// Stack implementation
int stack[MAX_SIZE];
int top = -1;
void push(int item) {
   if (top >= MAX_SIZE - 1) {
```

```
printf("Stack Overflow\n");
        return;
    }
    top++;
    stack[top] = item;
}
int pop() {
    if (top < 0) {
printf("Stack Underflow\n");
        return -1;
    }
    int item = stack[top];
    top--;
    return item;
}
int is_operator(char symbol) {
    if (symbol == '+' || symbol == '-' || symbol == '*' || symbol == '/') {
        return 1;
    }
    return 0;
}
int evaluate(char* expression) {
    int i = 0;
    char symbol = expression[i];
    int operand1, operand2, result;

    while (symbol != '\0') {
        if (symbol >= '0' && symbol <= '9') {
            int num = symbol - '0';
            push(num);
        }
        else if (is_operator(symbol)) {
            operand2 = pop();
            operand1 = pop();
            switch(symbol) {
                case '+': result = operand1 + operand2; break;
                case '-': result = operand1 - operand2; break;
                case '*': result = operand1 * operand2; break;
                case '/': result = operand1 / operand2; break;
            }
            push(result);
        }
i++;
        symbol = expression[i];
    }
    result = pop();
```

```
    return result;
}

int main() {
    char expression[] = "5 6 7 + * 8 -";
    int result = evaluate(expression);
printf("Result= %d\n", result);
return 0;
}
```

OUTPUT:

Result= 57

ii) Implement a program to check for balanced parentheses using a stack.

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

// Structure to represent a stack node
struct StackNode {
    char data;
    struct StackNode* next;
};

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

// Structure to represent a stack
struct Stack {
    struct StackNode* top;
};

// Function to create a new stack
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    if (stack == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
```

```
    }
    stack->top = NULL;
    return stack;
}

// Function to check if the stack is empty
bool isEmpty(struct Stack* stack) {
    return (stack->top == NULL);
}

// Function to push an element onto the stack
void push(struct Stack* stack, char data) {
    struct StackNode* newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
}

// Function to pop an element from the stack
char pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(EXIT_FAILURE);
    }
    struct StackNode* temp = stack->top;
    char data = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return data;
}

// Function to check if the given character is an opening parenthesis
bool isOpeningParenthesis(char ch) {
    return (ch == '(' || ch == '{' || ch == '[');
}

// Function to check if the given character is a closing parenthesis
bool isClosingParenthesis(char ch) {
    return (ch == ')' || ch == '}' || ch == ']');
}

// Function to check if the given characters form a balanced pair of parentheses
bool isPair(char open, char close) {
    return ((open == '(' && close == ')') ||
            (open == '{' && close == '}') ||
            (open == '[' && close == ']'));
}
```

```
// Function to check for balanced parentheses in the given string
bool checkBalancedParentheses(char* exp) {
    struct Stack* stack = createStack();
    int i;

    for (i = 0; exp[i] != '\0'; ++i) {
        if (isOpeningParenthesis(exp[i])) {
            push(stack, exp[i]);
        } else if (isClosingParenthesis(exp[i])) {
            if (isEmpty(stack) || !isPair(pop(stack), exp[i])) {
                return false;
            }
        }
    }
    return isEmpty(stack);
}

int main() {
    char exp[100];
    printf("Enter a string with parentheses: ");
    scanf("%s", exp);

    if (checkBalancedParentheses(exp)) {
        printf("Balanced parentheses.\n");
    } else {
        printf("Unbalanced parentheses.\n");
    }

    return 0;
}
```

OUTPUT:

```
Enter a string with parentheses: (siva)
Balanced parentheses.
```

Exercise 6: Queue Operations

i) Implement a queue using arrays and linked lists.

Using Arrays

```
/*
 * C Program to Implement a Queue using an Array
 */
#include <stdio.h>

#define MAX 50

void insert();
```

```
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      printf("1.Insert element to queue \n");
      printf("2.Delete element from queue \n");
      printf("3.Display all elements of queue \n");
      printf("4.Quit \n");
      printf("Enter your choice : ");
      scanf("%d", &choice);
      switch (choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(1);
         default:
         printf("Wrong choice \n");
      } /* End of switch */
   } /* End of while */
} /* End of main() */

void insert()
{
   int add_item;
   if (rear == MAX - 1)
   printf("Queue Overflow \n");
   else
   {
      if (front == - 1)
      /*If queue is initially empty */
      front = 0;
      printf("Inset the element in queue : ");
```

```
        scanf("%d", &add_item);
        rear = rear + 1;
        queue_array[rear] = add_item;
    }
} /* End of insert() */

void delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front + 1;
    }
} /* End of delete() */

void display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
} /* End of display() */
```

OUTPUT:

```
/tmp/8Qwlx2oGQD.o
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 12
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
```

```
Inset the element in queue : 23
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
12 23
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
```

Using Linked List

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

// Structure to represent a node in the queue
struct Node {
    int data;
    struct Node* next;
};

// Structure to represent a queue
struct Queue {
    struct Node *front, *rear;
};

// Function to create a new node
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;
}

// Function to create a new queue
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    if (queue == NULL) {
        printf("Memory allocation failed.\n");
```

```
        exit(EXIT_FAILURE);
    }
    queue->front = queue->rear = NULL;
    return queue;
}

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

// Function to enqueue an element into the queue
void enqueue(struct Queue* queue, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
    printf("%d enqueued to queue\n", data);
}

// Function to dequeue an element from the queue
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        exit(EXIT_FAILURE);
    }
    struct Node* temp = queue->front;
    int data = temp->data;
    queue->front = queue->front->next;
    free(temp);
    return data;
}

// Function to get the front element of the queue without removing it
int front(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. No front element.\n");
        exit(EXIT_FAILURE);
    }
    return queue->front->data;
}

// Function to print the elements of the queue
```

```
void printQueue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = queue->front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Queue* queue = createQueue();

    enqueue(queue, 1);
    enqueue(queue, 2);
    enqueue(queue, 3);

    printQueue(queue);

    printf("%d dequeued from queue\n", dequeue(queue));
    printf("%d dequeued from queue\n", dequeue(queue));

    printQueue(queue);

    return 0;
}
```

OUTPUT:

```
1 enqueued to queue
2 enqueued to queue
3 enqueued to queue
Queue: 1 2 3
1 dequeued from queue
2 dequeued from queue
Queue: 3
```

ii) Develop a program to simulate a simple printer queue system.

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

// Structure to represent a print job
struct PrintJob {
```

```
    int jobNumber;
    int numPages;
};

// Structure to represent a node in the print queue
struct Node {
    struct PrintJob data;
    struct Node* next;
};

// Structure to represent the print queue
struct Queue {
    struct Node *front, *rear;
};

// Function to create a new node
struct Node* createNode(struct PrintJob 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;
}

// Function to create a new queue
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    if (queue == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    queue->front = queue->rear = NULL;
    return queue;
}

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

// Function to enqueue a print job into the queue
void enqueue(struct Queue* queue, struct PrintJob data) {
    struct Node* newNode = createNode(data);
```

```
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Function to dequeue a print job from the queue
struct PrintJob dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        exit(EXIT_FAILURE);
    }
    struct Node* temp = queue->front;
    struct PrintJob data = temp->data;
    queue->front = queue->front->next;
    free(temp);
    return data;
}

// Function to print the contents of the queue
void printQueue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = queue->front;
    printf("Printer Queue:\n");
    while (temp != NULL) {
        printf("Job Number: %d, Pages: %d\n", temp->data.jobNumber, temp->data.numPages);
        temp = temp->next;
    }
}

int main() {
    struct Queue* printerQueue = createQueue();
    srand(time(NULL));

    // Simulate print job generation and enqueueing
    for (int i = 1; i <= 5; ++i) {
        struct PrintJob newJob;
        newJob.jobNumber = i;
        newJob.numPages = rand() % 20 + 1; // Random number of pages between 1 and 20
        enqueue(printerQueue, newJob);
    }
```

```
    // Print the initial printer queue
    printQueue(printerQueue);

    // Simulate print job processing (dequeueing)
    printf("\nProcessing print jobs...\n");
    while (!isEmpty(printerQueue)) {
        struct PrintJob job = dequeue(printerQueue);
        printf("Printing Job Number: %d, Pages: %d\n", job.jobNumber, job.numPages);
    }

    return 0;
}
```

OUTPUT:

```
Printer Queue:
Job Number: 1, Pages: 1
Job Number: 2, Pages: 15
Job Number: 3, Pages: 12
Job Number: 4, Pages: 13
Job Number: 5, Pages: 12

Processing print jobs...
Printing Job Number: 1, Pages: 1
Printing Job Number: 2, Pages: 15
Printing Job Number: 3, Pages: 12
Printing Job Number: 4, Pages: 13
Printing Job Number: 5, Pages: 12
```

iii) Solve problems involving circular queues.

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

#define MAX_SIZE 5

// Structure to represent a circular queue
struct CircularQueue {
    int front, rear;
    int array[MAX_SIZE];
    unsigned int size;
};

// Function to create a new circular queue
struct CircularQueue* createCircularQueue() {
    struct CircularQueue* queue = (struct CircularQueue*)malloc(sizeof(struct CircularQueue));
    if (queue == NULL) {
        printf("Memory allocation failed.\n");
```

```
        exit(EXIT_FAILURE);
    }
    queue->front = -1;
    queue->rear = -1;
    queue->size = 0;
    return queue;
}

// Function to check if the circular queue is empty
bool isEmpty(struct CircularQueue* queue) {
    return (queue->size == 0);
}

// Function to check if the circular queue is full
bool isFull(struct CircularQueue* queue) {
    return (queue->size == MAX_SIZE);
}

// Function to enqueue an element into the circular queue
void enqueue(struct CircularQueue* queue, int data) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (isEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }
    queue->array[queue->rear] = data;
    queue->size++;
    printf("%d enqueued to queue\n", data);
}

// Function to dequeue an element from the circular queue
int dequeue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        exit(EXIT_FAILURE);
    }
    int data = queue->array[queue->front];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }
```

```
    queue->size--;
    return data;
}

// Function to get the front element of the circular queue without removing it
int front(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. No front element.\n");
        exit(EXIT_FAILURE);
    }
    return queue->array[queue->front];
}

// Function to print the elements of the circular queue
void printQueue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }
    int i = queue->front;
    printf("Circular Queue: ");
    do {
        printf("%d ", queue->array[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (queue->rear + 1) % MAX_SIZE);
    printf("\n");
}

int main() {
    struct CircularQueue* queue = createCircularQueue();

    enqueue(queue, 1);
    enqueue(queue, 2);
    enqueue(queue, 3);

    printQueue(queue);

    printf("%d dequeued from queue\n", dequeue(queue));
    printf("%d dequeued from queue\n", dequeue(queue));

    printQueue(queue);

    enqueue(queue, 4);
    enqueue(queue, 5);
    enqueue(queue, 6);
```

```
    printQueue(queue);

    return 0;
}
```

OUTPUT:

```
/tmp/8Qwlx2oGQD.o
1 enqueued to queue
2 enqueued to queue
3 enqueued to queue
Circular Queue: 1 2 3
1 dequeued from queue
2 dequeued from queue
Circular Queue: 3
4 enqueued to queue
5 enqueued to queue
6 enqueued to queue
Circular Queue: 3 4 5 6
```

Exercise 7: Stack and Queue Applications

1) Use a stack to evaluate an infix expression and convert it to postfix.

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

// Structure to represent a stack node
struct StackNode {
    char data;
    struct StackNode* next;
};

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

// Structure to represent a stack
struct Stack {
```

```
    struct StackNode* top;
};

// Function to create a new stack
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    if (stack == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    stack->top = NULL;
    return stack;
}

// Function to check if the stack is empty
bool isEmpty(struct Stack* stack) {
    return (stack->top == NULL);
}

// Function to push an element onto the stack
void push(struct Stack* stack, char data) {
    struct StackNode* newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
}

// Function to pop an element from the stack
char pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(EXIT_FAILURE);
    }
    struct StackNode* temp = stack->top;
    char data = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return data;
}

// Function to peek the top element of the stack without removing it
char peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. No top element.\n");
        exit(EXIT_FAILURE);
    }
    return stack->top->data;
```

```
}

// Function to return the precedence of an operator
int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return -1;
    }
}

// Function to convert infix expression to postfix
char* infixToPostfix(char* exp) {
    struct Stack* stack = createStack();
    char* postfix = (char*)malloc((strlen(exp) + 1) * sizeof(char));
    if (postfix == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    int i = 0, k = 0;

    while (exp[i] != '\0') {
        if (isdigit(exp[i]) || isalpha(exp[i])) {
            postfix[k++] = exp[i++];
        } else if (exp[i] == '(') {
            push(stack, exp[i++]);
        } else if (exp[i] == ')') {
            while (!isEmpty(stack) && peek(stack) != '(') {
                postfix[k++] = pop(stack);
            }
            if (!isEmpty(stack) && peek(stack) != '(') {
                printf("Invalid expression.\n");
                exit(EXIT_FAILURE);
            } else {
                pop(stack);
            }
            i++;
        } else {
            while (!isEmpty(stack) && precedence(exp[i]) <= precedence(peek(stack))) {
```

```
            postfix[k++] = pop(stack);
        }
        push(stack, exp[i++]);
      }
   }

   while (!isEmpty(stack)) {
      postfix[k++] = pop(stack);
   }

   postfix[k] = '\0';
   return postfix;
}

int main() {
   char exp[100];
   printf("Enter an infix expression: ");
   scanf("%s", exp);

   char* postfix = infixToPostfix(exp);
   printf("Postfix expression: %s\n", postfix);
   free(postfix);

   return 0;
}
```

OUTPUT:

Enter an infix expression: 2+3*4/5

Postfix expression: 234*5/+

iii) Create a program to determine whether a given string is a palindrome or not.

```
// C implementation to check if a given
// string is palindrome or not
#include <stdio.h>
#include <string.h>

int main()
{
        char str[] = { "abbba" };

        // Start from first and
        // last character of str
        int l = 0;
        int h = strlen(str) - 1;

        // Keep comparing characters
        // while they are same
        while (h > l) {
```

```
            if (str[l++] != str[h--]) {
                printf("%s is not a palindrome\n", str);
                return 0;
                // will return from here
            }
        }

        printf("%s is a palindrome\n", str);

        return 0;
}
```

OUTPUT: abbba is a palindrome

iv) Implement a stack or queue to perform comparison and check for symmetry.

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

// Structure to represent a stack node
struct StackNode {
    char data;
    struct StackNode* next;
};

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

// Structure to represent a stack
struct Stack {
    struct StackNode* top;
};

// Function to create a new stack
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    if (stack == NULL) {
        printf("Memory allocation failed.\n");
```

```
        exit(EXIT_FAILURE);
    }
    stack->top = NULL;
    return stack;
}

// Function to check if the stack is empty
bool isEmpty(struct Stack* stack) {
    return (stack->top == NULL);
}

// Function to push an element onto the stack
void push(struct Stack* stack, char data) {
    struct StackNode* newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
}

// Function to pop an element from the stack
char pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(EXIT_FAILURE);
    }
    struct StackNode* temp = stack->top;
    char data = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return data;
}

// Function to check if the given string is symmetric
bool isSymmetric(char* str) {
    struct Stack* stack = createStack();
    int len = strlen(str);
    int i;

    // Push the first half of characters onto the stack
    for (i = 0; i < len / 2; ++i) {
        push(stack, str[i]);
    }

    // If the length is odd, skip the middle character
    if (len % 2 != 0) {
        i++;
    }
```

```
    // Compare the second half of characters with the stack
    while (str[i] != '\0') {
        if (isEmpty(stack) || pop(stack) != str[i]) {
            return false;
        }
        i++;
    }

    return isEmpty(stack); // Check if stack is empty at the end
}

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    if (isSymmetric(str)) {
        printf("The string \"%s\" is symmetric.\n", str);
    } else {
        printf("The string \"%s\" is not symmetric.\n", str);
    }

    return 0;
}
```

OUTPUT:

```
Enter a string: siva
The string "siva" is not symmetric.
```

Exercise 8: Binary Search Tree

i) Implementing a BST using Linked List.

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

// Structure to represent a node in the BST
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
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->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a new node into the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

// Function to perform an inorder traversal of the BST
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Function to perform a preorder traversal of the BST
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Function to perform a postorder traversal of the BST
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
```

```
int main() {
    struct Node* root = NULL;

    // Insert elements into the BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Perform inorder traversal
    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    // Perform preorder traversal
    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    // Perform postorder traversal
    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

OUTPUT:

```
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
```

ii) Traversing of BST.

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

// Structure to represent a node in the BST
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
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->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a new node into the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

// Function to perform an inorder traversal of the BST
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Function to perform a preorder traversal of the BST
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Function to perform a postorder traversal of the BST
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
```

```
        printf("%d ", root->data);
    }
}

int main() {
    struct Node* root = NULL;

    // Insert elements into the BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Perform inorder traversal
    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    // Perform preorder traversal
    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    // Perform postorder traversal
    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Exercise 9: Hashing

i) Implement a hash table with collision resolution techniques

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

#define SIZE 10

// Structure to represent a node in the hash table
struct Node {
    int key;
    int data;
    struct Node* next;
```

```
};

// Structure to represent a hash table
struct HashTable {
    struct Node* table[SIZE];
};

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

// Function to create a new hash table
struct HashTable* createHashTable() {
    struct HashTable* hashTable = (struct HashTable*)malloc(sizeof(struct HashTable));
    if (hashTable == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < SIZE; ++i) {
        hashTable->table[i] = NULL;
    }
    return hashTable;
}

// Function to calculate the hash value
int hashFunction(int key) {
    return key % SIZE;
}

// Function to insert a key-value pair into the hash table
void insert(struct HashTable* hashTable, int key, int data) {
    int index = hashFunction(key);
    struct Node* newNode = createNode(key, data);
    if (hashTable->table[index] == NULL) {
        hashTable->table[index] = newNode;
    } else {
        struct Node* temp = hashTable->table[index];
```

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

// Function to search for a key in the hash table and return its corresponding data
int search(struct HashTable* hashTable, int key) {
    int index = hashFunction(key);
    struct Node* temp = hashTable->table[index];
    while (temp != NULL) {
        if (temp->key == key) {
            return temp->data;
        }
        temp = temp->next;
    }
    return -1; // Key not found
}

// Function to delete a key-value pair from the hash table
void delete(struct HashTable* hashTable, int key) {
    int index = hashFunction(key);
    struct Node* current = hashTable->table[index];
    struct Node* prev = NULL;
    while (current != NULL && current->key != key) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        printf("Key not found. Cannot delete.\n");
        return;
    }
    if (prev == NULL) {
        hashTable->table[index] = current->next;
    } else {
        prev->next = current->next;
    }
    free(current);
    printf("Key-value pair with key %d deleted successfully.\n", key);
}

// Function to print the hash table
void display(struct HashTable* hashTable) {
    printf("Hash Table:\n");
    for (int i = 0; i < SIZE; ++i) {
```

```
        printf("[%d]: ", i);
        struct Node* temp = hashTable->table[i];
        while (temp != NULL) {
            printf("(%d, %d) ", temp->key, temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    struct HashTable* hashTable = createHashTable();

    // Insert key-value pairs into the hash table
    insert(hashTable, 10, 100);
    insert(hashTable, 20, 200);
    insert(hashTable, 30, 300);
    insert(hashTable, 15, 150);
    insert(hashTable, 25, 250);

    // Display the hash table
    display(hashTable);

    // Search for a key in the hash table
    int keyToSearch = 20;
    int result = search(hashTable, keyToSearch);
    if (result != -1) {
        printf("Value for key %d is %d.\n", keyToSearch, result);
    } else {
        printf("Key %d not found.\n", keyToSearch);
    }

    // Delete a key-value pair from the hash table
    int keyToDelete = 15;
    delete(hashTable, keyToDelete);
    display(hashTable);

    return 0;
}
```

OUTPUT:

```
Hash Table:
[0]: (10, 100) (20, 200) (30, 300)
[1]:
[2]:
[3]:
[4]:
```

```
[5]: (15, 150) (25, 250)
[6]:
[7]:
[8]:
[9]:
Value for key 20 is 200.
Key-value pair with key 15 deleted successfully.
Hash Table:
[0]: (10, 100) (20, 200) (30, 300)
[1]:
[2]:
[3]:
[4]:
[5]: (25, 250)
[6]:
[7]:
[8]:
[9]:
```

ii) Write a program to implement a simple cache using hashing.

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

#define CACHE_SIZE 10

// Structure to represent a node in the cache
struct CacheNode {
    int key;
    int data;
    struct CacheNode* next;
};

// Structure to represent the cache
struct Cache {
    struct CacheNode* table[CACHE_SIZE];
};

// Function to create a new cache node
struct CacheNode* createCacheNode(int key, int data) {
    struct CacheNode* newNode = (struct CacheNode*)malloc(sizeof(struct CacheNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    newNode->key = key;
```

```
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to create a new cache
struct Cache* createCache() {
    struct Cache* cache = (struct Cache*)malloc(sizeof(struct Cache));
    if (cache == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < CACHE_SIZE; ++i) {
        cache->table[i] = NULL;
    }
    return cache;
}

// Function to calculate the hash value
int hashFunction(int key) {
    return key % CACHE_SIZE;
}

// Function to insert a key-value pair into the cache
void insert(struct Cache* cache, int key, int data) {
    int index = hashFunction(key);
    struct CacheNode* newNode = createCacheNode(key, data);
    if (cache->table[index] == NULL) {
        cache->table[index] = newNode;
    } else {
        struct CacheNode* temp = cache->table[index];
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("Key-value pair (%d, %d) inserted into cache.\n", key, data);
}

// Function to search for a key in the cache and return its corresponding data
int search(struct Cache* cache, int key) {
    int index = hashFunction(key);
    struct CacheNode* temp = cache->table[index];
    while (temp != NULL) {
        if (temp->key == key) {
            return temp->data;
```

```
        }
        temp = temp->next;
    }
    return -1; // Key not found
}

// Function to print the cache
void display(struct Cache* cache) {
    printf("Cache:\n");
    for (int i = 0; i < CACHE_SIZE; ++i) {
        printf("[%d]: ", i);
        struct CacheNode* temp = cache->table[i];
        while (temp != NULL) {
            printf("(%d, %d) ", temp->key, temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    struct Cache* cache = createCache();

    // Insert key-value pairs into the cache
    insert(cache, 10, 100);
    insert(cache, 20, 200);
    insert(cache, 30, 300);
    insert(cache, 15, 150);
    insert(cache, 25, 250);

    // Display the cache
    display(cache);

    // Search for a key in the cache
    int keyToSearch = 20;
    int result = search(cache, keyToSearch);
    if (result != -1) {
        printf("Value for key %d is %d.\n", keyToSearch, result);
    } else {
        printf("Key %d not found.\n", keyToSearch);
    }

    return 0;
}
```

Output:

Key-value pair (10, 100) inserted into cache.
Key-value pair (20, 200) inserted into cache.
Key-value pair (30, 300) inserted into cache.
Key-value pair (15, 150) inserted into cache.
Key-value pair (25, 250) inserted into cache.
Cache:
[0]: (10, 100) (20, 200) (30, 300)
[1]:
[2]:
[3]:
[4]:
[5]: (15, 150) (25, 250)
[6]:
[7]:
[8]:
[9]:
Value for key 20 is 200.