DATA STRUCTURES USING C LAB

M.Sc (COMPUTER SCIENCE)
SEMESTER-I, PAPER-VI

Lesson Writer:

Dr. Kampa Lavanya
Asst. Professor
Dept. of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Editor

Dr. Neelima Guntupalli
Asst. Professor,
Dept. of Comp. Science & Eng.,
Acharya Nagarjuna University
Nagarjunanagar – 522 510.

Director, I/c.
Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W ., M.Phil., Ph.D.
Professor
Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

M.Sc Computer Science

First Edition : 2025

No. of Copies :

This book is exclusively prepared for the use of students of M.Sc (Computer Science), Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Published by:

Director **I/c**
Prof. V. Venkateswarlu,
M.A., M.P.S., M.S.W . M.Phil., Ph.D.
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University

M.Sc. Computer Science
Semester-I, Paper-VI
DATA STRUCTURES USING C LAB

Lab Cycle

1. Program for Sorting 'n' elements Using bubble sort technique.
2. Sort given elements using Selection Sort.
3. Sort given elements using Insertion Sort.
4. Sort given elements using Merge Sort.
5. Sort given elements using Quick Sort.
6. Implement the following operations on single linked list.
(i) Creation
(ii) Insertion
(iii) Deletion
(iv) Display

7. Implement the following operations on double linked list.
(i) Creation
(ii) Insertion
(iii) Deletion
(iv) Display

8. Implement the following operations on circular linked list.
(i) Creation
(ii) Insertion
(iii) Deletion
(iv) Display

9. Program for splitting given linked list.

10.Program for traversing the given linked list in reverse order.

11.Merge two given linked lists.

12. Create a linked list to store the names of colors.

13. Implement Stack Operations Using Arrays.

14. Implement Stack Operations Using Linked List.

15. Implement Queue Operations Using Arrays.

16. Implement Queue Operations Using Linked List.

17. Implement Operations on Circular Queue.

18. Construct and implement operations on Priority Queue.

19. Implement Operations on double ended Queue.

20. Converting infix expression to postfix expression by using stack.

21. Write program to evaluate post fix expression.

22. Implement Operations on two-way stack.

23. Add two polynomials using Linked List.
24. Multiply Two polynomials using Linked List.
25. Construct BST and implement traversing techniques recursively.
26. Implement preorder traversal on BST non recursively.
27. Implement in order traversal on BST non recursively.
28. Implement post order traversal on BST non recursively.
29. To Convert matrix into sparse matrix.
30. Implement binary search techniques recursively.
31. Program to implement graph traversing techniques DFS AND DFS.
32. Program to estimate shortest path for a graph.

DATA STRUCTURES USING C LAB

OBJECTIVES:

The objective of this lab is to master the DATA STRUCTURES concepts with C programming and to learn how to work with real time applications using DATA STRUCTURES USING C. These programs are widely used in most real-time scenarios. After the end of lab, students will be able know the complete practical exposure on DATA STRUCTURES USING C.

STRUCTURE

1. Program for Sorting 'n' elements Using bubble sort technique.
2. Sort given elements using Selection Sort.
3. Sort given elements using Insertion Sort.
4. Sort given elements using Merge Sort.
5. Sort given elements using Quick Sort.
6. Implement the following operations on single linked list.
   (i) Creation (ii) Insertion (iii) Deletion (iv) Display
7. Implement the following operations on double linked list.
   (i) Creation (ii) Insertion (iii) Deletion (iv) Display
8. Implement the following operations on circular linked list.
   (i) Creation (ii) Insertion (iii) Deletion (iv) Display
9. Program for splitting given linked list.
10. Program for traversing the given linked list in reverse order.
11. Merge two given linked lists.
12. Create a linked list to store the names of colors.
13. Implement Stack Operations Using Arrays.
14. Implement Stack Operations Using Linked List.
15. Implement Queue Operations Using Arrays.
16. Implement Queue Operations Using Linked List.
17. Implement Operations on Circular Queue.
18. Construct and implement operations on Priority Queue.
19. Implement Operations on double ended Queue.
20. Converting infix expression to postfix expression by using stack.
21. Write program to evaluate post fix expression.
22. Implement Operations on two-way stack.
23. Add two polynomials using Linked List.
24. Multiply Two polynomials using Linked List.
25. Construct BST and implement traversing techniques recursively.
26. Implement preorder traversal on BST non recursively.
27. To Convert matrix into sparse matrix.
28. Implement binary search techniques recursively.
29. Program to implement graph traversing techniques DFS AND DFS.
30. Program to estimate shortest path for a graph.

1. Program for Sorting 'n' elements Using bubble sort technique.

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
```

```
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Input:

Enter the number of elements: 5

Enter 5 elements:
64 34 25 12 22
Output:
Original array: 64 34 25 12 22
Sorted array: 12 22 25 34 64

2. Sort given elements using Selection Sort.

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;

    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in the unsorted part
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the found minimum element with the first element of the unsorted part
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
```

```
    printf("Original array: ");
    printArray(arr, n);
    selectionSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

Input:

Enter the number of elements: 6
Enter 6 elements:
29 10 14 37 13 5

Output:

Original array: 29 10 14 37 13 5
Sorted array: 5 10 13 14 29 37

3. Sort given elements using Insertion Sort.

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = arr[i]; // The element to be inserted
        j = i - 1;
        // Move elements of arr[0..i-1] that are greater than key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
```

```
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Original array: ");
    printArray(arr, n);
    insertionSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

Input:

Enter the number of elements: 5
Enter 5 elements:
12 11 13 5 6

Output:

Original array: 12 11 13 5 6
Sorted array: 5 6 11 12 13

4. Sort given elements using Merge Sort.

```
#include <stdio.h>
// Function to merge two halves of the array
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 L[] and R[]
    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 back into arr[left..right]
    i = 0; // Initial index of the first subarray
    j = 0; // Initial index of the second subarray
    k = left; // Initial index of the 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 the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the 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 the first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
```

```
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original array: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Input:

Enter the number of elements: 6
Enter 6 elements:
38 27 43 3 9 82

Output:

Original array: 38 27 43 3 9 82
Sorted array: 3 9 27 38 43 82

5. Sort given elements using Quick Sort.

```
#include <stdio.h>
// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as the pivot
    int i = (low - 1); // Index of the smaller element
    int temp;

    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++;
            // Swap arr[i] and arr[j]
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
```

```
    // Swap arr[i+1] and arr[high] (or pivot)
    temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array around the pivot
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original array: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);
```

```
    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Input:

Enter the number of elements: 6
Enter 6 elements:
10 80 30 90 40 50

Output:

Original array: 10 80 30 90 40 50
Sorted array: 10 30 40 50 80 90

6. Implement the following operations on single linked list.
(i) Creation (ii) Insertion (iii) Deletion (iv) Display

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

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

// Function prototypes
void createNode(int data);
void insertNode(int data, int position);
void deleteNode(int position);
void displayList();

// Head of the linked list
struct Node* head = NULL;

// Function to create a new node and add it to the end
void createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
```

```
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to insert a node at a specific position
void insertNode(int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

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

        if (temp == NULL) {
            printf("Invalid position!\n");
            free(newNode);
        } else {
            newNode->next = temp->next;
            temp->next = newNode;
        }
    }
}

// Function to delete a node at a specific position
void deleteNode(int position) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = head;

    if (position == 1) {
        head = temp->next;
        free(temp);
    } else {
        struct Node* prev = NULL;
```

```
        for (int i = 1; i < position && temp != NULL; i++) {
            prev = temp;
            temp = temp->next;
        }

        if (temp == NULL) {
            printf("Invalid position!\n");
        } else {
            prev->next = temp->next;
            free(temp);
        }
    }
}

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

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

// Main function
int main() {
    int choice, data, position;

    while (1) {
        printf("\nLinked List Operations:\n");
        printf("1. Create Node\n");
        printf("2. Insert Node\n");
        printf("3. Delete Node\n");
        printf("4. Display List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
```

```
            printf("Enter data to create node: ");
            scanf("%d", &data);
            createNode(data);
            break;

        case 2:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            printf("Enter position to insert: ");
            scanf("%d", &position);
            insertNode(data, position);
            break;

        case 3:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteNode(position);
            break;

        case 4:
            printf("Linked List: ");
            displayList();
            break;

        case 5:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 1
Enter data to create node: 10

```
Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 1
Enter data to create node: 20

Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 4
Linked List: 10 -> 20 -> NULL
```

7. Implement the following operations on double linked list.
(i) Creation (ii) Insertion (iii) Deletion (iv) Display

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

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

// Head of the doubly linked list
struct Node* head = NULL;

// Function to create a new node and add it to the end
void createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
```

```
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}
// Function to insert a node at a specific position
void insertNode(int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (position == 1) {
        newNode->next = head;
        newNode->prev = NULL;

        if (head != NULL) {
            head->prev = newNode;
        }
        head = newNode;
    } else {
        struct Node* temp = head;
        for (int i = 1; i < position - 1 && temp != NULL; i++) {
            temp = temp->next;
        }

        if (temp == NULL) {
            printf("Invalid position!\n");
            free(newNode);
        } else {
            newNode->next = temp->next;
            newNode->prev = temp;

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

// Function to delete a node at a specific position
void deleteNode(int position) {
```

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

    struct Node* temp = head;

    if (position == 1) {
        head = temp->next;
        if (head != NULL) {
            head->prev = NULL;
        }
        free(temp);
    } else {
        for (int i = 1; i < position && temp != NULL; i++) {
            temp = temp->next;
        }

        if (temp == NULL) {
            printf("Invalid position!\n");
        } else {
            if (temp->prev != NULL) {
                temp->prev->next = temp->next;
            }
            if (temp->next != NULL) {
                temp->next->prev = temp->prev;
            }
            free(temp);
        }
    }
}

// Function to display the list in forward direction
void displayList() {
    struct Node* temp = head;

    if (temp == NULL) {
        printf("List is empty!\n");
    } else {
        while (temp != NULL) {
            printf("%d <-> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
```

```
    }
}

// Main function
int main() {
    int choice, data, position;

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

        switch (choice) {
            case 1:
                printf("Enter data to create node: ");
                scanf("%d", &data);
                createNode(data);
                break;

            case 2:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter position to insert: ");
                scanf("%d", &position);
                insertNode(data, position);
                break;

            case 3:
                printf("Enter position to delete: ");
                scanf("%d", &position);
                deleteNode(position);
                break;
            case 4:
                printf("Doubly Linked List: ");
                displayList();
                break;

            case 5:
                printf("Exiting program.\n");
```

```
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
    }
  }

  return 0;
}
```

Menu Example:
mathematica
Doubly Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 1
Enter data to create node: 10

Doubly Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 1
Enter data to create node: 20

Doubly Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 4
Doubly Linked List: 10 <-> 20 <-> NULL

8. Implement the following operations on circular linked list.
(i) Creation (ii) Insertion (iii) Deletion (iv) Display

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

```
// Define the structure of a node
struct Node {
    int data;
    struct Node* next;
};
// Head of the circular linked list
struct Node* head = NULL;

// Function to create a new node and add it to the end
void createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }
}

// Function to insert a node at a specific position
void insertNode(int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    if (position == 1) {
        if (head == NULL) {
            head = newNode;
            newNode->next = head;
        } else {
            struct Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            newNode->next = head;
            head = newNode;
            temp->next = head;
        }
    } else {
```

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

        if (temp->next == head && position != 2) {
            printf("Invalid position!\n");
            free(newNode);
        } else {
            newNode->next = temp->next;
            temp->next = newNode;
        }
    }
}

// Function to delete a node at a specific position
void deleteNode(int position) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = head;

    if (position == 1) {
        if (head->next == head) {
            free(head);
            head = NULL;
        } else {
            struct Node* last = head;
            while (last->next != head) {
                last = last->next;
            }
            head = head->next;
            last->next = head;
            free(temp);
        }
    } else {
        struct Node* prev = NULL;
        for (int i = 1; i < position && temp->next != head; i++) {
            prev = temp;
            temp = temp->next;
        }
```

```
        if (temp->next == head && position != 2) {
            printf("Invalid position!\n");
        } else {
            prev->next = temp->next;
            free(temp);
        }
    }
}

// Function to display the circular linked list
void displayList() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

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

// Main function
int main() {
    int choice, data, position;

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

        switch (choice) {
            case 1:
                printf("Enter data to create node: ");
                scanf("%d", &data);
                createNode(data);
                break;
```

```
            case 2:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter position to insert: ");
                scanf("%d", &position);
                insertNode(data, position);
                break;

            case 3:
                printf("Enter position to delete: ");
                scanf("%d", &position);
                deleteNode(position);
                break;

            case 4:
                printf("Circular Linked List: ");
                displayList();
                break;

            case 5:
                printf("Exiting program.\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

```
Circular Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 1
Enter data to create node: 10

Circular Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
```

```
4. Display List
5. Exit
Enter your choice: 1
Enter data to create node: 20

Circular Linked List Operations:
1. Create Node
2. Insert Node
3. Delete Node
4. Display List
5. Exit
Enter your choice: 4
Circular Linked List: 10 -> 20 -> (head)
```

9. Program for splitting given linked list.

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

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

// Head of the linked list
struct Node* head = NULL;
// Function prototypes
void createNode(int data);
void splitList(struct Node** firstHalf, struct Node** secondHalf);
void displayList(struct Node* list);

// Function to create a new node and add it to the end
void createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   if (head == NULL) {
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
```

```
    }
}
// Function to split the linked list into two halves
void splitList(struct Node** firstHalf, struct Node** secondHalf) {
    if (head == NULL) {
        *firstHalf = NULL;
        *secondHalf = NULL;
        return;
    }
    struct Node* slow = head;
    struct Node* fast = head;
    struct Node* prev = NULL;

    // Use the slow and fast pointer approach
    while (fast != NULL && fast->next != NULL) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    // Split the list into two halves
    *firstHalf = head;

    if (fast == NULL) {
        // Even number of nodes
        *secondHalf = slow;
        prev->next = NULL;
    } else {
        // Odd number of nodes
        *secondHalf = slow->next;
        slow->next = NULL;
    }
}

// Function to display the linked list
void displayList(struct Node* list) {
    if (list == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = list;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
```

```
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    int choice, data;
    struct Node* firstHalf = NULL;
    struct Node* secondHalf = NULL;

    while (1) {
        printf("\nLinked List Operations:\n");
        printf("1. Create Node\n");
        printf("2. Split List\n");
        printf("3. Display Original List\n");
        printf("4. Display First Half\n");
        printf("5. Display Second Half\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to create node: ");
                scanf("%d", &data);
                createNode(data);
                break;

            case 2:
                splitList(&firstHalf, &secondHalf);
                printf("List split successfully!\n");
                break;

            case 3:
                printf("Original List: ");
                displayList(head);
                break;
            case 4:
                printf("First Half: ");
                displayList(firstHalf);
                break;
            case 5:
                printf("Second Half: ");
```

```
            displayList(secondHalf);
            break;
        case 6:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Input:

1. Create nodes: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
2. Split the list.

Output:

Original List: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
First Half: 10 -> 20 -> 30 -> NULL
Second Half: 40 -> 50 -> NULL

Input:

1. Create nodes: 10 -> 20 -> 30 -> 40 -> NULL
2. Split the list.

Output:

Original List: 10 -> 20 -> 30 -> 40 -> NULL
First Half: 10 -> 20 -> NULL
Second Half: 30 -> 40 -> NULL

10. Program for traversing the given linked list in reverse order.

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

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

// Head of the linked list
struct Node* head = NULL;
// Function prototypes
void createNode(int data);
void reverseTraversal(struct Node* current);
```

```
void displayList();

// Function to create a new node and add it to the end
void createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Recursive function to traverse the list in reverse order
void reverseTraversal(struct Node* current) {
    if (current == NULL) {
        return;
    }

    // Recursive call to the next node
    reverseTraversal(current->next);

    // Print the current node data after recursion
    printf("%d -> ", current->data);
}

// Function to display the linked list
void displayList() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
```

```
}

// Main function
int main() {
   int choice, data;

   while (1) {
      printf("\nLinked List Operations:\n");
      printf("1. Create Node\n");
      printf("2. Display List\n");
      printf("3. Reverse Traversal\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter data to create node: ");
            scanf("%d", &data);
            createNode(data);
            break;

         case 2:
            printf("Linked List: ");
            displayList();
            break;

         case 3:
            printf("Reverse Order: ");
            reverseTraversal(head);
            printf("NULL\n");
            break;

         case 4:
            printf("Exiting program.\n");
            exit(0);

         default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Input:

1. Create nodes: 10 -> 20 -> 30 -> 40 -> NULL
2. Reverse traversal.

Output:

Linked List: 10 -> 20 -> 30 -> 40 -> NULL
Reverse Order: 40 -> 30 -> 20 -> 10 -> NULL

11. Merge two given linked lists.

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

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

// Function prototypes
void createNode(struct Node** head, int data);
void displayList(struct Node* head);
struct Node* mergeLists(struct Node* list1, struct Node* list2);

// Function to create a new node and add it to the end of the list
void createNode(struct Node** head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;

   if (*head == NULL) {
      *head = newNode;
   } else {
      struct Node* temp = *head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}

// Function to display the linked list
void displayList(struct Node* head) {
   if (head == NULL) {
      printf("List is empty!\n");
```

```
        return;
    }

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

// Function to merge two sorted linked lists
struct Node* mergeLists(struct Node* list1, struct Node* list2) {
    if (list1 == NULL) {
        return list2;
    } else if (list2 == NULL) {
        return list1;
    }

    // Create a new head for the merged list
    struct Node* mergedHead = NULL;

    // If list1's data is smaller, merge it
    if (list1->data < list2->data) {
        mergedHead = list1;
        mergedHead->next = mergeLists(list1->next, list2);
    } else {
        mergedHead = list2;
        mergedHead->next = mergeLists(list1, list2->next);
    }

    return mergedHead;
}

// Main function
int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;
    struct Node* mergedList = NULL;
    int choice, data;

    while (1) {
        printf("\nLinked List Operations:\n");
        printf("1. Create Node for List 1\n");
```

```
printf("2. Create Node for List 2\n");
printf("3. Display List 1\n");
printf("4. Display List 2\n");
printf("5. Merge Lists\n");
printf("6. Display Merged List\n");
printf("7. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter data to create node for List 1: ");
      scanf("%d", &data);
      createNode(&list1, data);
      break;

   case 2:
      printf("Enter data to create node for List 2: ");
      scanf("%d", &data);
      createNode(&list2, data);
      break;
   case 3:
      printf("List 1: ");
      displayList(list1);
      break;

   case 4:
      printf("List 2: ");
      displayList(list2);
      break;

   case 5:
      mergedList = mergeLists(list1, list2);
      printf("Lists merged successfully!\n");
      break;

   case 6:
      printf("Merged List: ");
      displayList(mergedList);
      break;

   case 7:
      printf("Exiting program.\n");
      exit(0);
```

```
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Input:

1. Create nodes for List 1: 10 -> 20 -> 30
2. Create nodes for List 2: 5 -> 15 -> 25
3. Merge the lists.

Output:

List 1: 10 -> 20 -> 30 -> NULL
List 2: 5 -> 15 -> 25 -> NULL
Lists merged successfully!
Merged List: 5 -> 10 -> 15 -> 20 -> 25 -> 30 -> NULL

12. Create a linked list to store the names of colors.

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

// Define the structure of a node
struct Node {
    char color[20];        // Store color name (up to 19 characters + null terminator)
    struct Node* next;     // Pointer to the next node
};

// Head of the linked list
struct Node* head = NULL;

// Function prototypes
void createColorNode(char* color);
void displayColors();

// Function to create a new node and add a color to the list
void createColorNode(char* color) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->color, color);   // Copy color name to the node
    newNode->next = NULL;

    // If the list is empty, set the new node as head
    if (head == NULL) {
```

```
        head = newNode;
    } else {
        // Traverse to the end of the list and add the new node
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to display the colors stored in the linked list
void displayColors() {
    if (head == NULL) {
        printf("No colors in the list!\n");
        return;
    }
    struct Node* temp = head;
    printf("Colors in the list:\n");
    while (temp != NULL) {
        printf("%s -> ", temp->color);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    int choice;
    char color[20];

    while (1) {
        printf("\nLinked List Operations for Colors:\n");
        printf("1. Add Color\n");
        printf("2. Display Colors\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter color name: ");
                scanf("%s", color);  // Read the color name
                createColorNode(color);
```

```
            break;

        case 2:
            displayColors();
            break;

        case 3:
            printf("Exiting program.\n");
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Input:

1. Add colors: Red, Green, Blue.
2. Display the list of colors.

Output:

```
Linked List Operations for Colors:
1. Add Color
2. Display Colors
3. Exit
Enter your choice: 1
Enter color name: Red

Linked List Operations for Colors:
1. Add Color
2. Display Colors
3. Exit
Enter your choice: 1
Enter color name: Green

Linked List Operations for Colors:
1. Add Color
2. Display Colors
3. Exit
Enter your choice: 1
Enter color name: Blue

Linked List Operations for Colors:
1. Add Color
2. Display Colors
```

3. Exit
Enter your choice: 2
Colors in the list:
Red -> Green -> Blue -> NULL

13. Implement Stack Operations Using Arrays.

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

#define MAX 5  // Define maximum size of the stack

// Stack structure
struct Stack {
   int arr[MAX];  // Array to store stack elements
   int top;       // Index of the top element
};
// Function prototypes
void initializeStack(struct Stack* stack);
int isFull(struct Stack* stack);
int isEmpty(struct Stack* stack);
void push(struct Stack* stack, int value);
int pop(struct Stack* stack);
int peek(struct Stack* stack);
void displayStack(struct Stack* stack);

// Function to initialize the stack
void initializeStack(struct Stack* stack) {
   stack->top = -1;  // Set top to -1 to indicate an empty stack
}
// Function to check if the stack is full
int isFull(struct Stack* stack) {
   return stack->top == MAX - 1;  // Check if top is at the maximum index
}
// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
   return stack->top == -1;  // Check if top is -1 (stack is empty)
}
// Function to push an element onto the stack
void push(struct Stack* stack, int value) {
   if (isFull(stack)) {
      printf("Stack Overflow! Unable to push %d\n", value);
   } else {
      stack->arr[++(stack->top)] = value;  // Increment top and add element
```

```
        printf("Pushed %d to the stack\n", value);
    }
}
// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow! The stack is empty\n");
        return -1;  // Return -1 if stack is empty
    } else {
        return stack->arr[(stack->top)--];  // Return top element and decrement top
    }
}

// Function to peek at the top element of the stack
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Nothing to peek\n");
        return -1;
    } else {
        return stack->arr[stack->top];  // Return top element without removing it
    }
}

// Function to display the elements of the stack
void displayStack(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Nothing to display\n");
    } else {
        printf("Stack elements: ");
        for (int i = 0; i <= stack->top; i++) {
            printf("%d ", stack->arr[i]);
        }
        printf("\n");
    }
}

// Main function
int main() {
    struct Stack stack;
    int choice, value;

    // Initialize the stack
    initializeStack(&stack);
```

```
while (1) {
    printf("\nStack Operations:\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 element is: %d\n", value);
            }
            break;

        case 4:
            displayStack(&stack);
            break;

        case 5:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
    }
}
```

```
    return 0;
}
```

```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to push: 10
Pushed 10 to the stack

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to push: 20
Pushed 20 to the stack

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements: 10 20
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
Popped value: 20
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
```

5. Exit
Enter your choice: 3
Top element is: 10

14. Implement Stack Operations Using Linked List.

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

// Define the structure of a node
struct Node {
   int data;        // Store the stack element
   struct Node* next; // Pointer to the next node in the stack
};

// Stack structure with a pointer to the top node
struct Stack {
   struct Node* top; // Pointer to the top of the stack
};

// Function prototypes
void initializeStack(struct Stack* stack);
int isEmpty(struct Stack* stack);
void push(struct Stack* stack, int value);
int pop(struct Stack* stack);
int peek(struct Stack* stack);
void displayStack(struct Stack* stack);

// Function to initialize the stack
void initializeStack(struct Stack* stack) {
   stack->top = NULL;  // Set top to NULL, indicating an empty stack
}
// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
   return stack->top == NULL;  // Stack is empty if top is NULL
}
// Function to push an element onto the stack
void push(struct Stack* stack, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed\n");
      return;
   }
   newNode->data = value;    // Set data of the new node
```

```
    newNode->next = stack->top; // Link the new node to the current top
    stack->top = newNode;      // Update the top to the new node
    printf("Pushed %d to the stack\n", value);
}
// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow! The stack is empty\n");
        return -1;  // Return -1 if the stack is empty
    }
    struct Node* temp = stack->top;  // Store the top node temporarily
    int poppedValue = temp->data;    // Get the value of the top node
    stack->top = stack->top->next;   // Update top to the next node
    free(temp);                      // Free the memory of the old top node

    return poppedValue;
}
// Function to peek at the top element of the stack
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Nothing to peek\n");
        return -1;
    }
    return stack->top->data;  // Return the data of the top node
}
// Function to display the elements of the stack
void displayStack(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Nothing to display\n");
        return;
    }
    struct Node* temp = stack->top;
    printf("Stack elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Main function
int main() {
    struct Stack stack;
    int choice, value;
```

```
// Initialize the stack
initializeStack(&stack);
while (1) {
   printf("\nStack Operations using Linked List:\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 element is: %d\n", value);
         }
         break;
      case 4:
         displayStack(&stack);
         break;

      case 5:
         printf("Exiting program.\n");
         exit(0);

      default:
         printf("Invalid choice! Please try again.\n");
   }
}

return 0;
```

```
}
```

Explanation

1. Node Structure:
    - Each node in the stack is represented by the Node structure, which contains:
        - data: The actual data stored in the node.
        - next: A pointer to the next node in the stack.
2. Stack Structure:
    - The stack itself is represented by the Stack structure, which contains:
        - top: A pointer to the top node of the stack.
3. Operations:
    - initializeStack(): Initializes the stack by setting top to NULL, indicating that the stack is empty.
    - isEmpty(): Checks if the stack is empty by checking if top is NULL.
    - push(): Adds a new element to the top of the stack by creating a new node, setting its next to the current top, and updating top to point to the new node.
    - pop(): Removes the top element from the stack. The top pointer is updated to point to the next node, and the memory for the old top node is freed.
    - peek(): Returns the value of the top element without removing it.
    - displayStack(): Traverses the stack and prints each element, starting from the top.
4. Main Function:
    - A menu is provided to allow the user to perform different stack operations like push, pop, peek, display, or exit the program.

Sample Run
mathematica
Copy code

```
Stack Operations using Linked List:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to push: 10
Pushed 10 to the stack

Stack Operations using Linked List:
1. Push
2. Pop
```

3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to push: 20
Pushed 20 to the stack

Stack Operations using Linked List:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements: 20 10

Stack Operations using Linked List:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
Popped value: 20

Stack Operations using Linked List:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element is: 10

15. Implement Queue Operations Using Arrays.

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

#define MAX 5  // Define the maximum size of the queue

// Queue structure
struct Queue {
   int arr[MAX];  // Array to store queue elements
```

```
    int front;     // Index of the front element
    int rear;      // Index of the rear element
};

// Function prototypes
void initializeQueue(struct Queue* queue);
int isFull(struct Queue* queue);
int isEmpty(struct Queue* queue);
void enqueue(struct Queue* queue, int value);
int dequeue(struct Queue* queue);
int peek(struct Queue* queue);
void displayQueue(struct Queue* queue);

// Function to initialize the queue
void initializeQueue(struct Queue* queue) {
    queue->front = -1;  // Set front to -1 (indicating empty queue)
    queue->rear = -1;   // Set rear to -1 (indicating empty queue)
}

// Function to check if the queue is full
int isFull(struct Queue* queue) {
    return queue->rear == MAX - 1;  // Check if rear has reached the maximum index
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    return queue->front == -1;  // Check if front is -1 (empty queue)
}

// Function to enqueue (add an element) into the queue
void enqueue(struct Queue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue Overflow! Unable to enqueue %d\n", value);
    } else {
        if (queue->front == -1) {
            queue->front = 0;  // Set front to 0 when first element is added
        }
        queue->rear++;  // Move rear to the next position
        queue->arr[queue->rear] = value;  // Add the value at the rear
        printf("Enqueued %d to the queue\n", value);
    }
}

// Function to dequeue (remove an element) from the queue
```

```
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! The queue is empty\n");
        return -1;  // Return -1 if the queue is empty
    } else {
        int dequeuedValue = queue->arr[queue->front];  // Get the front element
        if (queue->front == queue->rear) {
            queue->front = queue->rear = -1;  // Reset the queue to empty
        } else {
            queue->front++;  // Move front to the next element
        }
        return dequeuedValue;
    }
}

// Function to peek (get the front element) from the queue
int peek(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty! Nothing to peek\n");
        return -1;
    } else {
        return queue->arr[queue->front];  // Return the front element
    }
}

// Function to display the elements of the queue
void displayQueue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty! Nothing to display\n");
    } else {
        printf("Queue elements: ");
        for (int i = queue->front; i <= queue->rear; i++) {
            printf("%d ", queue->arr[i]);
        }
        printf("\n");
    }
}

// Main function
int main() {
    struct Queue queue;
    int choice, value;

    // Initialize the queue
```

```
initializeQueue(&queue);

while (1) {
   printf("\nQueue Operations:\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 element is: %d\n", value);
         }
         break;

      case 4:
         displayQueue(&queue);
         break;

      case 5:
         printf("Exiting program.\n");
         exit(0);

      default:
         printf("Invalid choice! Please try again.\n");
   }
```

```
    }

    return 0;
}
```

```
Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to enqueue: 10
Enqueued 10 to the queue

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to enqueue: 20
Enqueued 20 to the queue

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 10 20

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
Dequeued value: 10

Queue Operations:
1. Enqueue
```

2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element is: 20

16. Implement Queue Operations Using Linked List.

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

// Define the structure of a node
struct Node {
   int data;        // Store the data of the node
   struct Node* next; // Pointer to the next node in the queue
};

// Queue structure with pointers to the front and rear nodes
struct Queue {
   struct Node* front; // Pointer to the front of the queue
   struct Node* rear;  // Pointer to the rear of the queue
};

// Function prototypes
void initializeQueue(struct Queue* queue);
int isEmpty(struct Queue* queue);
void enqueue(struct Queue* queue, int value);
int dequeue(struct Queue* queue);
int peek(struct Queue* queue);
void displayQueue(struct Queue* queue);

// Function to initialize the queue
void initializeQueue(struct Queue* queue) {
   queue->front = NULL; // Set front to NULL, indicating an empty queue
   queue->rear = NULL;  // Set rear to NULL, indicating an empty queue
}

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

// Function to enqueue (add an element) into the queue
```

```
void enqueue(struct Queue* queue, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    newNode->data = value;  // Set data for the new node
    newNode->next = NULL;    // Set next to NULL as it's going to be the last node

    if (queue->rear == NULL) {
        queue->front = newNode; // If the queue is empty, set the front to the new node
        queue->rear = newNode;  // Set the rear to the new node
    } else {
        queue->rear->next = newNode; // Link the current rear to the new node
        queue->rear = newNode;       // Update the rear to the new node
    }

    printf("Enqueued %d to the queue\n", value);
}

// Function to dequeue (remove an element) from the queue
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! The queue is empty\n");
        return -1; // Return -1 if the queue is empty
    }

    struct Node* temp = queue->front;   // Store the front node temporarily
    int dequeuedValue = temp->data;     // Get the data of the front node
    queue->front = queue->front->next;  // Move the front to the next node

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

    free(temp);  // Free the memory of the old front node

    return dequeuedValue;  // Return the dequeued value
}

// Function to peek (get the front element) from the queue
int peek(struct Queue* queue) {
    if (isEmpty(queue)) {
```

```
        printf("Queue is empty! Nothing to peek\n");
        return -1; // Return -1 if the queue is empty
    }

    return queue->front->data;  // Return the data of the front node
}

// Function to display the elements of the queue
void displayQueue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty! Nothing to display\n");
        return;
    }

    struct Node* temp = queue->front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);  // Print each node's data
        temp = temp->next;          // Move to the next node
    }
    printf("\n");
}

// Main function
int main() {
    struct Queue queue;
    int choice, value;

    // Initialize the queue
    initializeQueue(&queue);

    while (1) {
        printf("\nQueue Operations using Linked List:\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 element is: %d\n", value);
            }
            break;

        case 4:
            displayQueue(&queue);
            break;

        case 5:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Queue Operations using Linked List:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to enqueue: 10
Enqueued 10 to the queue

Queue Operations using Linked List:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to enqueue: 20
Enqueued 20 to the queue

Queue Operations using Linked List:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 10 20

Queue Operations using Linked List:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
Dequeued value: 10

Queue Operations using Linked List:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element is: 20

17. Implement Operations on Circular Queue.

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

#define MAX 5  // Define the maximum size of the circular queue
```

```
// Circular Queue structure
struct CircularQueue {
    int arr[MAX];  // Array to store elements of the queue
    int front;     // Index of the front element
    int rear;      // Index of the rear element
};

// Function prototypes
void initializeQueue(struct CircularQueue* queue);
int isFull(struct CircularQueue* queue);
int isEmpty(struct CircularQueue* queue);
void enqueue(struct CircularQueue* queue, int value);
int dequeue(struct CircularQueue* queue);
int peek(struct CircularQueue* queue);
void displayQueue(struct CircularQueue* queue);

// Function to initialize the queue
void initializeQueue(struct CircularQueue* queue) {
    queue->front = -1;  // Set front to -1, indicating an empty queue
    queue->rear = -1;   // Set rear to -1, indicating an empty queue
}

// Function to check if the queue is full
int isFull(struct CircularQueue* queue) {
    return (queue->rear + 1) % MAX == queue->front;  // Check if next position of rear
is front
}

// Function to check if the queue is empty
int isEmpty(struct CircularQueue* queue) {
    return queue->front == -1;  // Queue is empty if front is -1
}

// Function to enqueue (add an element) into the circular queue
void enqueue(struct CircularQueue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue Overflow! Unable to enqueue %d\n", value);
    } else {
        if (queue->front == -1) {  // If the queue is empty, set front and rear to 0
            queue->front = 0;
        }
        queue->rear = (queue->rear + 1) % MAX;  // Circular increment of rear
        queue->arr[queue->rear] = value;  // Add the value at the rear
        printf("Enqueued %d to the queue\n", value);
```

```
    }
}

// Function to dequeue (remove an element) from the circular queue
int dequeue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! The queue is empty\n");
        return -1;  // Return -1 if the queue is empty
    } else {
        int dequeuedValue = queue->arr[queue->front];  // Get the value at the front
        if (queue->front == queue->rear) {  // If there is only one element left
            queue->front = queue->rear = -1;  // Set front and rear to -1 (empty queue)
        } else {
            queue->front = (queue->front + 1) % MAX;  // Circular increment of front
        }
        return dequeuedValue;  // Return the dequeued value
    }
}

// Function to peek (get the front element) from the circular queue
int peek(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty! Nothing to peek\n");
        return -1;  // Return -1 if the queue is empty
    } else {
        return queue->arr[queue->front];  // Return the element at the front
    }
}

// Function to display the elements of the circular queue
void displayQueue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty! Nothing to display\n");
    } else {
        printf("Queue elements: ");
        int i = queue->front;
        while (i != queue->rear) {  // Traverse until we reach the rear
            printf("%d ", queue->arr[i]);
            i = (i + 1) % MAX;  // Circular increment
        }
        printf("%d\n", queue->arr[queue->rear]);  // Print the rear element
    }
}
```

```
// Main function
int main() {
    struct CircularQueue queue;
    int choice, value;
    // Initialize the circular queue
    initializeQueue(&queue);

    while (1) {
        printf("\nCircular Queue Operations:\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 element is: %d\n", value);
                }
                break;

            case 4:
                displayQueue(&queue);
                break;

            case 5:
                printf("Exiting program.\n");
```

```
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
    }
  }

  return 0;
}
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to enqueue: 10
Enqueued 10 to the queue

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to enqueue: 20
Enqueued 20 to the queue

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 10 20

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
```

```
5. Exit
Enter your choice: 2
Dequeued value: 10

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element is: 20
```

18. Construct and implement operations on Priority Queue.

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

#define MAX 5  // Define the maximum size of the priority queue

// Structure for an element in the priority queue
struct Element {
   int data;      // Store the data of the element
   int priority;  // Store the priority of the element
};

// Priority Queue structure
struct PriorityQueue {
   struct Element arr[MAX];  // Array to store elements and their priorities
   int size;                 // Number of elements in the priority queue
};

// Function prototypes
void initializeQueue(struct PriorityQueue* pq);
int isFull(struct PriorityQueue* pq);
int isEmpty(struct PriorityQueue* pq);
void insert(struct PriorityQueue* pq, int value, int priority);
int removeMax(struct PriorityQueue* pq);
int peek(struct PriorityQueue* pq);
void displayQueue(struct PriorityQueue* pq);

// Function to initialize the priority queue
void initializeQueue(struct PriorityQueue* pq) {
   pq->size = 0;  // Set size to 0, indicating an empty queue
```

```
}

// Function to check if the queue is full
int isFull(struct PriorityQueue* pq) {
   return pq->size == MAX;  // Queue is full if size is equal to MAX
}

// Function to check if the queue is empty
int isEmpty(struct PriorityQueue* pq) {
   return pq->size == 0;  // Queue is empty if size is 0
}

// Function to insert an element into the priority queue
void insert(struct PriorityQueue* pq, int value, int priority) {
   if (isFull(pq)) {
      printf("Queue Overflow! Unable to insert %d with priority %d\n", value,
priority);
      return;
   }

   int i;
   for (i = pq->size - 1; i >= 0; i--) {
      // Move elements with lower priority to the right
      if (pq->arr[i].priority < priority) {
         pq->arr[i + 1] = pq->arr[i];  // Shift element to the right
      } else {
         break;  // Stop when we find an element with equal or higher priority
      }
   }

   // Insert the new element at the correct position
   pq->arr[i + 1].data = value;
   pq->arr[i + 1].priority = priority;
   pq->size++;  // Increase the size of the queue
   printf("Inserted %d with priority %d\n", value, priority);
}

// Function to remove and return the element with the highest priority
int removeMax(struct PriorityQueue* pq) {
   if (isEmpty(pq)) {
      printf("Queue Underflow! The queue is empty\n");
      return -1;  // Return -1 if the queue is empty
   }
```

```
    // The element with the highest priority is always at the front (index 0)
    int maxValue = pq->arr[0].data;

    // Shift all elements to the left to fill the gap
    for (int i = 0; i < pq->size - 1; i++) {
        pq->arr[i] = pq->arr[i + 1];
    }

    pq->size--;  // Decrease the size of the queue
    return maxValue;  // Return the removed value
}

// Function to peek the element with the highest priority
int peek(struct PriorityQueue* pq) {
    if (isEmpty(pq)) {
        printf("Queue is empty! Nothing to peek\n");
        return -1;  // Return -1 if the queue is empty
    }

    return pq->arr[0].data;  // Return the data of the highest priority element
}

// Function to display the elements of the priority queue
void displayQueue(struct PriorityQueue* pq) {
    if (isEmpty(pq)) {
        printf("Queue is empty! Nothing to display\n");
    } else {
        printf("Priority Queue elements: \n");
        for (int i = 0; i < pq->size; i++) {
            printf("Value: %d, Priority: %d\n", pq->arr[i].data, pq->arr[i].priority);
        }
    }
}

// Main function
int main() {
    struct PriorityQueue pq;
    int choice, value, priority;

    // Initialize the priority queue
    initializeQueue(&pq);

    while (1) {
        printf("\nPriority Queue Operations:\n");
```

```
    printf("1. Insert\n");
    printf("2. Remove Max (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 insert: ");
            scanf("%d", &value);
            printf("Enter the priority of the value: ");
            scanf("%d", &priority);
            insert(&pq, value, priority);
            break;

        case 2:
            value = removeMax(&pq);
            if (value != -1) {
                printf("Removed value: %d\n", value);
            }
            break;

        case 3:
            value = peek(&pq);
            if (value != -1) {
                printf("Element with highest priority is: %d\n", value);
            }
            break;

        case 4:
            displayQueue(&pq);
            break;

        case 5:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
    }
}
```

```
    return 0;
}
```

```
Priority Queue Operations:
1. Insert
2. Remove Max (Dequeue)
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to insert: 10
Enter the priority of the value: 2
Inserted 10 with priority 2

Priority Queue Operations:
1. Insert
2. Remove Max (Dequeue)
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the value to insert: 20
Enter the priority of the value: 5
Inserted 20 with priority 5

Priority Queue Operations:
1. Insert
2. Remove Max (Dequeue)
3. Peek
4. Display
5. Exit
Enter your choice: 4
Priority Queue elements:
Value: 20, Priority: 5
Value: 10, Priority: 2

Priority Queue Operations:
1. Insert
2. Remove Max (Dequeue)
3. Peek
4. Display
5. Exit
Enter your choice: 2
Removed value: 20
```

Priority Queue Operations:
1. Insert
2. Remove Max (Dequeue)
3. Peek
4. Display
5. Exit
Enter your choice: 3
Element with highest priority is: 10

19. Implement Operations on double ended Queue.

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

#define MAX 5  // Define the maximum size of the deque

// Structure for Deque
struct Deque {
   int arr[MAX];  // Array to store elements of the deque
   int front;     // Index of the front element
   int rear;      // Index of the rear element
};

// Function prototypes
void initializeDeque(struct Deque* dq);
int isFull(struct Deque* dq);
int isEmpty(struct Deque* dq);
void insertFront(struct Deque* dq, int value);
void insertRear(struct Deque* dq, int value);
int deleteFront(struct Deque* dq);
int deleteRear(struct Deque* dq);
int peekFront(struct Deque* dq);
int peekRear(struct Deque* dq);
void displayDeque(struct Deque* dq);

// Function to initialize the deque
void initializeDeque(struct Deque* dq) {
   dq->front = -1;
   dq->rear = -1;
}

// Function to check if the deque is full
int isFull(struct Deque* dq) {
```

```
    return (dq->front == 0 && dq->rear == MAX - 1) || (dq->rear + 1 == dq->front);  // Check if front and rear are at the same position in a circular way
}

// Function to check if the deque is empty
int isEmpty(struct Deque* dq) {
    return dq->front == -1;  // Deque is empty if front is -1
}

// Function to insert an element at the front of the deque
void insertFront(struct Deque* dq, int value) {
    if (isFull(dq)) {
        printf("Deque Overflow! Unable to insert %d at front\n", value);
        return;
    }

    if (dq->front == -1) {  // If the deque is empty, set front and rear to 0
        dq->front = dq->rear = 0;
    } else if (dq->front == 0) {  // If front is at the beginning, wrap around to the end
        dq->front = MAX - 1;
    } else {
        dq->front--;  // Move the front pointer to the previous position
    }
    dq->arr[dq->front] = value;  // Insert the element at the front
    printf("Inserted %d at the front of the deque\n", value);
}

// Function to insert an element at the rear of the deque
void insertRear(struct Deque* dq, int value) {
    if (isFull(dq)) {
        printf("Deque Overflow! Unable to insert %d at rear\n", value);
        return;
    }

    if (dq->front == -1) {  // If the deque is empty, set front and rear to 0
        dq->front = dq->rear = 0;
    } else if (dq->rear == MAX - 1) {  // If rear is at the end, wrap around to the beginning
        dq->rear = 0;
    } else {
        dq->rear++;  // Move the rear pointer to the next position
    }
    dq->arr[dq->rear] = value;  // Insert the element at the rear
    printf("Inserted %d at the rear of the deque\n", value);
```

```
}

// Function to delete an element from the front of the deque
int deleteFront(struct Deque* dq) {
    if (isEmpty(dq)) {
        printf("Deque Underflow! Unable to delete from front\n");
        return -1;  // Return -1 if the deque is empty
    }

    int deletedValue = dq->arr[dq->front];
    if (dq->front == dq->rear) {  // If there is only one element, set both front and rear
to -1
        dq->front = dq->rear = -1;
    } else if (dq->front == MAX - 1) {  // If front is at the end, wrap around to the
beginning
        dq->front = 0;
    } else {
        dq->front++;  // Move the front pointer to the next position
    }

    return deletedValue;  // Return the deleted value
}

// Function to delete an element from the rear of the deque
int deleteRear(struct Deque* dq) {
    if (isEmpty(dq)) {
        printf("Deque Underflow! Unable to delete from rear\n");
        return -1;  // Return -1 if the deque is empty
    }

    int deletedValue = dq->arr[dq->rear];

    if (dq->front == dq->rear) {  // If there is only one element, set both front and rear
to -1
        dq->front = dq->rear = -1;
    } else if (dq->rear == 0) {  // If rear is at the beginning, wrap around to the end
        dq->rear = MAX - 1;
    } else {
        dq->rear--;  // Move the rear pointer to the previous position
    }

    return deletedValue;  // Return the deleted value
}
```

```
// Function to peek the element at the front of the deque
int peekFront(struct Deque* dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty! Nothing to peek at the front\n");
        return -1;
    }
    return dq->arr[dq->front];  // Return the element at the front
}
// Function to peek the element at the rear of the deque
int peekRear(struct Deque* dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty! Nothing to peek at the rear\n");
        return -1;
    }
    return dq->arr[dq->rear];  // Return the element at the rear
}

// Function to display the elements in the deque
void displayDeque(struct Deque* dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty! Nothing to display\n");
        return;
    }

    printf("Deque elements: ");
    int i = dq->front;
    while (i != dq->rear) {  // Traverse the deque from front to rear
        printf("%d ", dq->arr[i]);
        i = (i + 1) % MAX;  // Circular increment
    }
    printf("%d\n", dq->arr[dq->rear]);  // Print the rear element
}

// Main function
int main() {
    struct Deque dq;
    int choice, value;

    // Initialize the deque
    initializeDeque(&dq);

    while (1) {
        printf("\nDouble Ended Queue (Deque) Operations:\n");
        printf("1. Insert Front\n");
```

```
printf("2. Insert Rear\n");
printf("3. Delete Front\n");
printf("4. Delete Rear\n");
printf("5. Peek Front\n");
printf("6. Peek Rear\n");
printf("7. Display\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter the value to insert at the front: ");
      scanf("%d", &value);
      insertFront(&dq, value);
      break;

   case 2:
      printf("Enter the value to insert at the rear: ");
      scanf("%d", &value);
      insertRear(&dq, value);
      break;

   case 3:
      value = deleteFront(&dq);
      if (value != -1) {
         printf("Deleted value from front: %d\n", value);
      }
      break;

   case 4:
      value = deleteRear(&dq);
      if (value != -1) {
         printf("Deleted value from rear: %d\n", value);
      }
      break;

   case 5:
      value = peekFront(&dq);
      if (value != -1) {
         printf("Front element is: %d\n", value);
      }
      break;
```

```
            case 6:
                value = peekRear(&dq);
                if (value != -1) {
                    printf("Rear element is: %d\n", value);
                }
                break;

            case 7:
                displayDeque(&dq);
                break;

            case 8:
                printf("Exiting program.\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

```
Double Ended Queue (Deque) Operations:
1. Insert Front
2. Insert Rear
3. Delete Front
4. Delete Rear
5. Peek Front
6. Peek Rear
7. Display
8. Exit
Enter your choice: 1
Enter the value to insert at the front: 10
Inserted 10 at the front of the deque

Double Ended Queue (Deque) Operations:
1. Insert Front
2. Insert Rear
3. Delete Front
4. Delete Rear
5. Peek Front
6. Peek Rear
7. Display
8. Exit
```

```
Enter your choice: 2
Enter the value to insert at the rear: 20
Inserted 20 at the rear of the deque

Double Ended Queue (Deque) Operations:
1. Insert Front
2. Insert Rear
3. Delete Front
4. Delete Rear
5. Peek Front
6. Peek Rear
7. Display
8. Exit
Enter your choice: 7
Deque elements: 10 20
```

20. Converting infix expression to postfix expression by using stack.

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

#define MAX 100

// Stack structure
struct Stack {
   int top;
   char items[MAX];
};

// Function to initialize the stack
void initStack(struct Stack* stack) {
   stack->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
   return stack->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack* stack) {
   return stack->top == (MAX - 1);
}
```

```
// Function to push an element onto the stack
void push(struct Stack* stack, char value) {
    if (isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->items[++stack->top] = value;
}

// Function to pop an element from the stack
char pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;  // Error value
    }
    return stack->items[stack->top--];
}

// Function to peek the top element of the stack
char peek(struct Stack* stack) {
    if (!isEmpty(stack)) {
        return stack->items[stack->top];
    }
    return -1;  // Error value
}

// Function to get the precedence of operators
int precedence(char c) {
    if (c == '+' || c == '-') {
        return 1;
    } else if (c == '*' || c == '/') {
        return 2;
    } else {
        return 0;  // For parenthesis
    }
}

// Function to check if a character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to convert infix expression to postfix expression
```

```
void infixToPostfix(char* infix, char* postfix) {
    struct Stack stack;
    initStack(&stack);
    int i = 0, j = 0;

    while (infix[i] != '\0') {
        char current = infix[i];

        // If current character is an operand, add it to the postfix expression
        if (isalnum(current)) {
            postfix[j++] = current;
        }
        // If current character is '(', push it to the stack
        else if (current == '(') {
            push(&stack, current);
        }
        // If current character is ')', pop and append to postfix until '(' is encountered
        else if (current == ')') {
            while (!isEmpty(&stack) && peek(&stack) != '(') {
                postfix[j++] = pop(&stack);
            }
            pop(&stack);  // Discard the '('
        }
        // If current character is an operator
        else if (isOperator(current)) {
            while (!isEmpty(&stack) && precedence(peek(&stack)) >=
precedence(current)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, current);
        }
        i++;
    }

    // Pop all remaining operators from the stack and append to postfix
    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }
    postfix[j] = '\0';  // Null-terminate the postfix expression
}

int main() {
    char infix[MAX], postfix[MAX];
```

```
    // Read infix expression
    printf("Enter infix expression: ");
    scanf("%s", infix);

    // Convert infix to postfix
    infixToPostfix(infix, postfix);

    // Display the postfix expression
    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

Input:

Enter infix expression: A+B*(C-D)

Output:

Postfix expression: ABCD-*+

21. Write program to evaluate post fix expression.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX 100

// Stack structure
struct Stack {
    int top;
    int items[MAX];
};

// Function to initialize the stack
void initStack(struct Stack* stack) {
    stack->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack* stack) {
    return stack->top == (MAX - 1);
}
```

```
// Function to push an element onto the stack
void push(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->items[++stack->top] = value;
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;  // Error value
    }
    return stack->items[stack->top--];
}

// Function to evaluate a postfix expression
int evaluatePostfix(char* expression) {
    struct Stack stack;
    initStack(&stack);

    for (int i = 0; expression[i] != '\0'; i++) {
        char current = expression[i];

        // If the current character is a number, push it onto the stack
        if (isdigit(current)) {
            push(&stack, current - '0');  // Convert char to integer
        }
        // If the current character is an operator, pop two operands, apply the operator,
and push the result
        else if (current == '+' || current == '-' || current == '*' || current == '/') {
            int operand2 = pop(&stack);
            int operand1 = pop(&stack);

            switch (current) {
                case '+': push(&stack, operand1 + operand2); break;
                case '-': push(&stack, operand1 - operand2); break;
                case '*': push(&stack, operand1 * operand2); break;
                case '/': push(&stack, operand1 / operand2); break;
            }
        }
```

```
    }

    // The final result will be the only element left in the stack
    return pop(&stack);
}

int main() {
    char postfix[MAX];

    // Read postfix expression
    printf("Enter postfix expression: ");
    scanf("%s", postfix);

    // Evaluate the postfix expression
    int result = evaluatePostfix(postfix);

    // Display the result
    printf("Result of the postfix expression: %d\n", result);

    return 0;
}
```

Input:

Enter postfix expression: 23*54*+

Output:

Result of the postfix expression: 23

22. Implement Operations on two-way stack.

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

#define MAX 100

// Structure for two-way stack
struct TwoWayStack {
    int arr[MAX];
    int left_top;
    int right_top;
};

// Function to initialize the two-way stack
void initTwoWayStack(struct TwoWayStack* stack) {
    stack->left_top = -1;  // Left stack starts at -1
    stack->right_top = MAX;  // Right stack starts at MAX
```

```
}

// Function to check if the left stack is full
int isLeftStackFull(struct TwoWayStack* stack) {
    return stack->left_top == stack->right_top - 1;
}

// Function to check if the right stack is full
int isRightStackFull(struct TwoWayStack* stack) {
    return stack->right_top == stack->left_top + 1;
}

// Function to push an element onto the left stack
void pushLeft(struct TwoWayStack* stack, int value) {
    if (isLeftStackFull(stack)) {
        printf("Left stack overflow\n");
        return;
    }
    stack->arr[++stack->left_top] = value;
}

// Function to push an element onto the right stack
void pushRight(struct TwoWayStack* stack, int value) {
    if (isRightStackFull(stack)) {
        printf("Right stack overflow\n");
        return;
    }
    stack->arr[--stack->right_top] = value;
}

// Function to pop an element from the left stack
int popLeft(struct TwoWayStack* stack) {
    if (stack->left_top == -1) {
        printf("Left stack underflow\n");
        return -1;  // Error value
    }
    return stack->arr[stack->left_top--];
}

// Function to pop an element from the right stack
int popRight(struct TwoWayStack* stack) {
    if (stack->right_top == MAX) {
        printf("Right stack underflow\n");
        return -1;  // Error value
```

```
    }
    return stack->arr[stack->right_top++];
}

// Function to peek the top element of the left stack
int peekLeft(struct TwoWayStack* stack) {
    if (stack->left_top == -1) {
        printf("Left stack is empty\n");
        return -1;  // Error value
    }
    return stack->arr[stack->left_top];
}

// Function to peek the top element of the right stack
int peekRight(struct TwoWayStack* stack) {
    if (stack->right_top == MAX) {
        printf("Right stack is empty\n");
        return -1;  // Error value
    }
    return stack->arr[stack->right_top];
}

// Function to display the elements of the left stack
void displayLeftStack(struct TwoWayStack* stack) {
    if (stack->left_top == -1) {
        printf("Left stack is empty\n");
        return;
    }
    printf("Left stack: ");
    for (int i = 0; i <= stack->left_top; i++) {
        printf("%d ", stack->arr[i]);
    }
    printf("\n");
}

// Function to display the elements of the right stack
void displayRightStack(struct TwoWayStack* stack) {
    if (stack->right_top == MAX) {
        printf("Right stack is empty\n");
        return;
    }
    printf("Right stack: ");
    for (int i = MAX - 1; i >= stack->right_top; i--) {
        printf("%d ", stack->arr[i]);
```

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

int main() {
    struct TwoWayStack stack;
    initTwoWayStack(&stack);

    // Push elements to the left stack
    pushLeft(&stack, 10);
    pushLeft(&stack, 20);
    pushLeft(&stack, 30);

    // Push elements to the right stack
    pushRight(&stack, 40);
    pushRight(&stack, 50);
    pushRight(&stack, 60);

    // Display the left and right stacks
    displayLeftStack(&stack);
    displayRightStack(&stack);

    // Pop elements from the left stack
    printf("Popped from left stack: %d\n", popLeft(&stack));

    // Pop elements from the right stack
    printf("Popped from right stack: %d\n", popRight(&stack));

    // Display the left and right stacks after popping
    displayLeftStack(&stack);
    displayRightStack(&stack);

    // Peek the top element from both stacks
    printf("Top element of left stack: %d\n", peekLeft(&stack));
    printf("Top element of right stack: %d\n", peekRight(&stack));

    return 0;
}
```

Input:

Pushing to left stack: 10, 20, 30

Pushing to right stack: 40, 50, 60

Output:

Left stack: 10 20 30

Right stack: 60 50 40

Popped from left stack: 30
Popped from right stack: 60
Left stack: 10 20
Right stack: 50 40
Top element of left stack: 20
Top element of right stack: 50

23. Add two polynomials using Linked List.

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

// Structure to represent a polynomial term
struct Node {
   int coefficient;
   int exponent;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int coefficient, int exponent) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->coefficient = coefficient;
   newNode->exponent = exponent;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a term into the polynomial in descending order of exponents
void insertTerm(struct Node** head, int coefficient, int exponent) {
   struct Node* newNode = createNode(coefficient, exponent);
   if (*head == NULL || (*head)->exponent < exponent) {
      newNode->next = *head;
      *head = newNode;
   } else {
      struct Node* temp = *head;
      while (temp->next != NULL && temp->next->exponent > exponent) {
         temp = temp->next;
      }
      if (temp->next != NULL && temp->next->exponent == exponent) {
         temp->next->coefficient += coefficient;
         free(newNode);  // No need to insert a new node, just update coefficient
      } else {
         newNode->next = temp->next;
         temp->next = newNode;
```

```
        }
    }
}

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

    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
            insertTerm(&result, poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else {  // When exponents are equal
            int sum = poly1->coefficient + poly2->coefficient;
            if (sum != 0) {
                insertTerm(&result, sum, poly1->exponent);
            }
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }

    // Append remaining terms from poly1 or poly2
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }

    return result;
}

// Function to display the polynomial
void displayPolynomial(struct Node* poly) {
    if (poly == NULL) {
        printf("0\n");
        return;
```

```
    }

    while (poly != NULL) {
        printf("%dx^%d", poly->coefficient, poly->exponent);
        if (poly->next != NULL && poly->next->coefficient > 0) {
            printf(" + ");
        }
        poly = poly->next;
    }
    printf("\n");
}

int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;

    // Polynomial 1: 5x^3 + 4x^2 + 2x + 3
    insertTerm(&poly1, 5, 3);
    insertTerm(&poly1, 4, 2);
    insertTerm(&poly1, 2, 1);
    insertTerm(&poly1, 3, 0);

    // Polynomial 2: 3x^3 + 2x^2 + 4x + 1
    insertTerm(&poly2, 3, 3);
    insertTerm(&poly2, 2, 2);
    insertTerm(&poly2, 4, 1);
    insertTerm(&poly2, 1, 0);

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

    // Add the polynomials
    result = addPolynomials(poly1, poly2);

    printf("Sum of the polynomials: ");
    displayPolynomial(result);

    return 0;
}
```

Input:

Polynomial 1: 5x^3 + 4x^2 + 2x + 3

Polynomial 2: 3x^3 + 2x^2 + 4x + 1

Output:

Polynomial 1: 5x^3 + 4x^2 + 2x + 3
Polynomial 2: 3x^3 + 2x^2 + 4x + 1
Sum of the polynomials: 8x^3 + 6x^2 + 6x + 4

24. Multiply Two polynomials using Linked List.

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

// Structure to represent a polynomial term
struct Node {
   int coefficient;
   int exponent;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int coefficient, int exponent) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->coefficient = coefficient;
   newNode->exponent = exponent;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a term into the polynomial in descending order of exponents
void insertTerm(struct Node** head, int coefficient, int exponent) {
   struct Node* newNode = createNode(coefficient, exponent);
   if (*head == NULL || (*head)->exponent < exponent) {
      newNode->next = *head;
      *head = newNode;
   } else {
      struct Node* temp = *head;
      while (temp->next != NULL && temp->next->exponent > exponent) {
         temp = temp->next;
      }
      if (temp->next != NULL && temp->next->exponent == exponent) {
         temp->next->coefficient += coefficient;
         free(newNode);  // No need to insert a new node, just update coefficient
      } else {
         newNode->next = temp->next;
         temp->next = newNode;
```

```
        }
    }
}

// Function to multiply two polynomials
struct Node* multiplyPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;
    struct Node *temp1 = poly1, *temp2 = poly2;

    while (temp1 != NULL) {
        while (temp2 != NULL) {
            int newCoefficient = temp1->coefficient * temp2->coefficient;
            int newExponent = temp1->exponent + temp2->exponent;
            insertTerm(&result, newCoefficient, newExponent);
            temp2 = temp2->next;
        }
        temp1 = temp1->next;
        temp2 = poly2;  // Reset temp2 to the start of poly2
    }

    return result;
}

// Function to display the polynomial
void displayPolynomial(struct Node* poly) {
    if (poly == NULL) {
        printf("0\n");
        return;
    }

    while (poly != NULL) {
        printf("%dx^%d", poly->coefficient, poly->exponent);
        if (poly->next != NULL && poly->next->coefficient > 0) {
            printf(" + ");
        }
        poly = poly->next;
    }
    printf("\n");
}

int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;
```

```
    // Polynomial 1: 5x^3 + 4x^2 + 2x + 3
    insertTerm(&poly1, 5, 3);
    insertTerm(&poly1, 4, 2);
    insertTerm(&poly1, 2, 1);
    insertTerm(&poly1, 3, 0);

    // Polynomial 2: 3x^3 + 2x^2 + 4x + 1
    insertTerm(&poly2, 3, 3);
    insertTerm(&poly2, 2, 2);
    insertTerm(&poly2, 4, 1);
    insertTerm(&poly2, 1, 0);

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

    // Multiply the polynomials
    result = multiplyPolynomials(poly1, poly2);

    printf("Product of the polynomials: ");
    displayPolynomial(result);

    return 0;
}
```

Input:

Polynomial 1: 5x^3 + 4x^2 + 2x + 3
Polynomial 2: 3x^3 + 2x^2 + 4x + 1

Output:

Polynomial 1: 5x^3 + 4x^2 + 2x + 3
Polynomial 2: 3x^3 + 2x^2 + 4x + 1
Product of the polynomials: 15x^6 + 22x^5 + 37x^4 + 22x^3 + 14x^2 + 14x + 3

25. Construct BST and implement traversing techniques recursively.

```
#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));
    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 the tree is empty, return a new node
    if (root == NULL) {
        return createNode(data);
    }

    // Otherwise, recur down the tree
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }

    // return the (unchanged) node pointer
    return root;
}

// Inorder Traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);  // Visit left subtree
        printf("%d ", root->data);     // Visit root
        inorderTraversal(root->right); // Visit right subtree
    }
}

// Preorder Traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);     // Visit root
        preorderTraversal(root->left);  // Visit left subtree
```

```
        preorderTraversal(root->right); // Visit right subtree
    }
}

// Postorder Traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);  // Visit left subtree
        postorderTraversal(root->right); // Visit right subtree
        printf("%d ", root->data);       // Visit root
    }
}

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

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

    // Display the BST with different traversal techniques
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

Input:

We insert the following values into the BST: 50, 30, 20, 40, 70, 60, 80.

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

26. Implement preorder traversal on BST non recursively.

```
#include <stdio.h>
#include <stdlib.h>
// Structure to represent a node in the BST
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

// Structure for stack
struct Stack {
   struct Node* node;
   struct Stack* next;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

// Function to push a node onto the stack
void push(struct Stack** stack, struct Node* node) {
   struct Stack* newStackNode = (struct Stack*)malloc(sizeof(struct Stack));
   newStackNode->node = node;
   newStackNode->next = *stack;
   *stack = newStackNode;
}

// Function to pop a node from the stack
struct Node* pop(struct Stack** stack) {
   if (*stack == NULL) {
      return NULL;
   }
   struct Stack* temp = *stack;
```

```
    *stack = (*stack)->next;
    struct Node* poppedNode = temp->node;
    free(temp);
    return poppedNode;
}

// Function to perform non-recursive preorder traversal
void nonRecursivePreorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    struct Stack* stack = NULL;  // Initialize the stack as empty
    push(&stack, root);  // Push the root onto the stack

    while (stack != NULL) {
        struct Node* current = pop(&stack);  // Pop a node from the stack
        printf("%d ", current->data);  // Visit the node (print its data)

        // Push right child first so that left child is processed first
        if (current->right != NULL) {
            push(&stack, current->right);
        }

        // Push left child onto the stack
        if (current->left != NULL) {
            push(&stack, current->left);
        }
    }
}

// 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;
```

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

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

   printf("Non-Recursive Preorder Traversal: ");
   nonRecursivePreorderTraversal(root);
   printf("\n");

   return 0;
}
```

Input: We insert the following values into the BST: 50, 30, 20, 40, 70, 60, 80.
Output:
Non-Recursive Preorder Traversal: 50 30 20 40 70 60 80

27. To Convert matrix into sparse matrix.

```
#include <stdio.h>

// Function to convert a matrix to sparse matrix
void convertToSparseMatrix(int matrix[4][4], int rows, int cols) {
   int sparseMatrix[rows * cols][3]; // Array to store sparse matrix (max size is rows *
cols)
   int nonZeroCount = 0;

   // Traverse the matrix to count non-zero elements
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         if (matrix[i][j] != 0) {
            sparseMatrix[nonZeroCount][0] = i;        // Row index
            sparseMatrix[nonZeroCount][1] = j;        // Column index
            sparseMatrix[nonZeroCount][2] = matrix[i][j];  // Value
            nonZeroCount++;
         }
      }
   }
```

```
    // Print the sparse matrix
    printf("Row   Column   Value\n");
    for (int i = 0; i < nonZeroCount; i++) {
        printf("%d      %d        %d\n", sparseMatrix[i][0], sparseMatrix[i][1],
sparseMatrix[i][2]);
    }
}

int main() {
    // Input matrix
    int matrix[4][4] = {
        {0, 0, 3, 0},
        {5, 0, 0, 0},
        {0, 0, 0, 7},
        {8, 0, 0, 0}
    };

    int rows = 4, cols = 4;

    printf("Original Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Convert to sparse matrix
    printf("\nSparse Matrix:\n");
    convertToSparseMatrix(matrix, rows, cols);

    return 0;
}
```

Input Matrix:

0 0 3 0
5 0 0 0
0 0 0 7
8 0 0 0

Output:

Original Matrix:
0 0 3 0

```
5 0 0 0
0 0 0 7
8 0 0 0

Sparse Matrix:
Row   Column   Value
---------------------
0     2        3
1     0        5
2     3        7
3     0        8
```

28. Implement binary search techniques recursively.

```
#include <stdio.h>

// Function for recursive binary search
int binarySearch(int arr[], int start, int end, int target) {
   // Base case: if the search interval is empty
   if (start > end) {
      return -1;  // Element is not present in the array
   }

   // Find the middle index
   int mid = start + (end - start) / 2;

   // If the target is found at the middle index
   if (arr[mid] == target) {
      return mid;  // Return the index of the target element
   }

   // If the target is smaller than the middle element, search in the left half
   if (arr[mid] > target) {
      return binarySearch(arr, start, mid - 1, target);
   }

   // If the target is greater than the middle element, search in the right half
   return binarySearch(arr, mid + 1, end, target);
}

int main() {
   // Sorted array for binary search
   int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21};
   int n = sizeof(arr) / sizeof(arr[0]);  // Calculate the number of elements in the array
   int target = 13;  // Element to search for
```

```
    // Call binarySearch function with initial parameters
    int result = binarySearch(arr, 0, n - 1, target);

    // Output the result
    if (result == -1) {
        printf("Element %d is not present in the array.\n", target);
    } else {
        printf("Element %d is present at index %d.\n", target, result);
    }

    return 0;
}
```

Input:

- Sorted array: {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
- Target to search: 13

Output:

Element 13 is present at index 6.

Input (when element is not found):

- Sorted array: {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
- Target to search: 10

Output:

Element 10 is not present in the array.

29. Program to implement graph traversing techniques DFS AND DFS.

```
#include <stdio.h>
#include <stdlib.h>
// Define the maximum number of vertices in the graph
#define MAX_VERTICES 10

// Graph represented as an adjacency list
struct Graph {
    int numVertices;
    int adjMatrix[MAX_VERTICES][MAX_VERTICES]; // Adjacency matrix
};

// Function to create a graph
void createGraph(struct Graph* graph, int vertices) {
    graph->numVertices = vertices;

    // Initialize the adjacency matrix with 0
    for (int i = 0; i < vertices; i++) {
```

```
        for (int j = 0; j < vertices; j++) {
            graph->adjMatrix[i][j] = 0;
        }
    }
}

// Function to add an edge between two vertices
void addEdge(struct Graph* graph, int src, int dest) {
    graph->adjMatrix[src][dest] = 1;
    graph->adjMatrix[dest][src] = 1;  // For undirected graph
}

// DFS function
void DFS(struct Graph* graph, int visited[], int vertex) {
    // Mark the vertex as visited and print it
    visited[vertex] = 1;
    printf("%d ", vertex);

    // Recur for all adjacent vertices
    for (int i = 0; i < graph->numVertices; i++) {
        if (graph->adjMatrix[vertex][i] == 1 && !visited[i]) {
            DFS(graph, visited, i);
        }
    }
}
// BFS function
void BFS(struct Graph* graph, int startVertex) {
    int visited[MAX_VERTICES] = {0};  // Initialize visited array
    int queue[MAX_VERTICES], front = -1, rear = -1;

    // Mark the start vertex as visited and enqueue it
    visited[startVertex] = 1;
    queue[++rear] = startVertex;

    // Process the vertices in the queue
    while (front != rear) {
        int currentVertex = queue[++front];
        printf("%d ", currentVertex);

        // Check all adjacent vertices
        for (int i = 0; i < graph->numVertices; i++) {
            if (graph->adjMatrix[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
                queue[++rear] = i;
```

```
            }
        }
    }
}

int main() {
    struct Graph graph;
    int vertices, edges, src, dest;
    // Input the number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    createGraph(&graph, vertices);

    // Input edges
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    for (int i = 0; i < edges; i++) {
        printf("Enter edge (src dest): ");
        scanf("%d %d", &src, &dest);
        addEdge(&graph, src, dest);
    }
    // DFS Traversal
    int visited[MAX_VERTICES] = {0};  // Initialize visited array
    printf("\nDFS Traversal starting from vertex 0: ");
    DFS(&graph, visited, 0);

    // BFS Traversal
    printf("\nBFS Traversal starting from vertex 0: ");
    BFS(&graph, 0);

    return 0;
}
```

Explanation of the Program:

1. Graph Representation:
    - The graph is represented using an adjacency matrix (adjMatrix[MAX_VERTICES][MAX_VERTICES]).
    - Each element adjMatrix[i][j] = 1 indicates an edge between vertex i and vertex j.
2. Graph Creation and Edge Addition:
    - createGraph: Initializes the adjacency matrix for the graph.
    - addEdge: Adds an edge between two vertices (for an undirected graph, the edge is added in both directions).
3. DFS (Depth First Search):

- The DFS function uses recursion to visit all the vertices starting from a given vertex.
- We maintain an array visited[] to track the visited vertices.
- The function visits the current vertex, prints it, and recursively visits all its unvisited neighbors.

4. BFS (Breadth First Search):
   - The BFS function uses a queue to explore all vertices level by level.
   - We start by marking the start vertex as visited and enqueue it.
   - We then dequeue a vertex, print it, and enqueue all its unvisited neighbors.
   - This process continues until the queue is empty.
5. Main Function:
   - We first create the graph and add edges.
   - Then, we perform DFS and BFS starting from vertex 0 (you can change this to any starting vertex).

Input:

Enter the number of vertices: 5
Enter the number of edges: 4
Enter edge (src dest): 0 1
Enter edge (src dest): 0 2
Enter edge (src dest): 1 3
Enter edge (src dest): 1 4
DFS Traversal starting from vertex 0: 0 1 3 4 2
BFS Traversal starting from vertex 0: 0 1 2 3 4

Output Explanation:

- The graph is represented as:

```
0 - 1 - 3
|   |
2   4
```

30. Program to estimate shortest path for a graph.

```
#include <stdio.h>
#include <limits.h>
#define MAX_VERTICES 10
// Function to find the vertex with the minimum distance that is not yet processed
int minDistance(int dist[], int visited[], int vertices) {
   int min = INT_MAX, minIndex;
   for (int v = 0; v < vertices; v++) {
      if (!visited[v] && dist[v] <= min) {
         min = dist[v];
         minIndex = v;
      }
   }
   return minIndex;
}
```

```
// Function to implement Dijkstra's algorithm
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int vertices, int start)
{
   int dist[vertices];  // Array to store the shortest distance to each vertex
   int visited[vertices];  // Boolean array to check if a vertex has been processed
   // Initialize the distance array and visited array
   for (int i = 0; i < vertices; i++) {
      dist[i] = INT_MAX;  // Set initial distance to infinity
      visited[i] = 0;  // Mark all vertices as unvisited
   }

   // Set the distance of the start vertex to 0
   dist[start] = 0;
   // Find the shortest path for each vertex
   for (int count = 0; count < vertices - 1; count++) {
      int u = minDistance(dist, visited, vertices);  // Get the vertex with minimum
distance
      visited[u] = 1;  // Mark the selected vertex as visited

      // Update the distance of the adjacent vertices of the selected vertex
      for (int v = 0; v < vertices; v++) {
         if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
         }
      }
   }

   // Print the shortest distances
   printf("Vertex   Distance from Source\n");
   for (int i = 0; i < vertices; i++) {
      printf("%d \t\t %d\n", i, dist[i]);
   }
}
int main() {
   int graph[MAX_VERTICES][MAX_VERTICES], vertices, edges, src, dest, weight;

   // Input the number of vertices and edges
   printf("Enter the number of vertices: ");
   scanf("%d", &vertices);
   // Initialize the graph as a 2D array
   printf("Enter the adjacency matrix representation of the graph (use 0 for no
edge):\n");
   for (int i = 0; i < vertices; i++) {
```

```
        for (int j = 0; j < vertices; j++) {
            graph[i][j] = 0;  // Initialize all edges to 0 (no edge)
        }
    }
    // Input edges and weights
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    for (int i = 0; i < edges; i++) {
        printf("Enter edge (src, dest, weight): ");
        scanf("%d %d %d", &src, &dest, &weight);
        graph[src][dest] = weight;
        graph[dest][src] = weight;  // For undirected graph
    }

    // Input the source vertex for Dijkstra's algorithm
    printf("Enter the source vertex: ");
    scanf("%d", &src);

    // Call Dijkstra's algorithm
    dijkstra(graph, vertices, src);
    return 0;
}
```

Input:

```
Enter the number of vertices: 5
Enter the adjacency matrix representation of the graph (use 0 for no edge):
Enter the number of edges: 6
Enter edge (src, dest, weight): 0 1 10
Enter edge (src, dest, weight): 0 2 5
Enter edge (src, dest, weight): 1 2 2
Enter edge (src, dest, weight): 1 3 1
Enter edge (src, dest, weight): 2 3 9
Enter edge (src, dest, weight): 3 4 4
Enter the source vertex: 0
```

Output:

```
        Vertex   Distance from Source
        0        0
        1        8
        2        5
        3        9
        4        13
```