Lenora College Of Engineering

Rampachodavaram

B.Tech – R23

Data Structures Using C Lab

L	T	P	C
0	0	3	1.5

DATA STRUCTURES LAB

(Common to CSE, IT & allied branches)

Course Objectives:

The course aims to strengthen the ability of the students to identify and apply the suitable data structure for the given real-world problem. It enables them to gain knowledge in practical applications of data structures.

Course Outcomes: At the end of the course, Student will be able to

CO1: Explain the role of linear data structures in organizing and accessing data efficiently in algorithms.
CO2: Design, implement, and apply linked lists for dynamic data storage, demonstrating understanding of memory allocation.
CO3: Develop programs using stacks to handle recursive algorithms, manage program states, and solve related problems.
CO4: Apply queue-based algorithms for efficient task scheduling and breadth-first traversal in graphs and distinguish between deques and priority queues and apply them appropriately to solve data management challenges.
CO5: Recognize scenarios where hashing is advantageous, and design hash-based solutions for specific problems.

List of Experiments:

Exercise 1: Array Manipulation

i) Write a program to reverse an array.
ii) C Programs to implement the Searching Techniques – Linear & Binary Search
iii) C Programs to implement Sorting Techniques – Bubble, Selection and Insertion Sort

Exercise 2: Linked List Implementation

i) Implement a singly linked list and perform insertion and deletion operations.
ii) Develop a program to reverse a linked list iteratively and recursively.
iii) Solve problems involving linked list traversal and manipulation.

Exercise 3: Linked List Applications

i) Create a program to detect and remove duplicates from a linked list.
ii) Implement a linked list to represent polynomials and perform addition.
iii) Implement a double-ended queue (deque) with essential operations.

Exercise 4: Double Linked List Implementation

i) Implement a doubly linked list and perform various operations to understand its properties and applications.
ii) Implement a circular linked list and perform insertion, deletion, and traversal.

Exercise 5: Stack Operations

i) Implement a stack using arrays and linked lists.
ii) Write a program to evaluate a postfix expression using a stack.
iii) Implement a program to check for balanced parentheses using a stack.

Exercise 6: Queue Operations

i) Implement a queue using arrays and linked lists.
ii) Develop a program to simulate a simple printer queue system.
iii) Solve problems involving circular queues.

Exercise 7: Stack and Queue Applications

i) Use a stack to evaluate an infix expression and convert it to postfix.
ii) Create a program to determine whether a given string is a palindrome or not.
iii) Implement a stack or queue to perform comparison and check for symmetry.

Exercise 8: Binary Search Tree

i) Implementing a BST using Linked List.
ii) Traversing of BST.

Exercise 9: Hashing

i) Implement a hash table with collision resolution techniques.
ii) Write a program to implement a simple cache using hashing.

Textbooks:

1. Data Structures and algorithm analysis in C, Mark Allen Weiss, Pearson, 2nd Edition.
2. Fundamentals of data structures in C, Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, Silicon Press, 2008

Reference Books:

1. Algorithms and Data Structures: The Basic Toolbox by Kurt Mehlhorn and Peter Sanders
2. C Data Structures and Algorithms by Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft
3. Problem Solving with Algorithms and Data Structures" by Brad Miller and David Ranum
4. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
5. Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms by Robert Sedgewick.

Exercise 1: Array Manipulation

i) Write a program to reverse an array.

```
#include <stdio.h>
int main() {
   int arr[100], n, i, temp;
   // Get the size of the array
   printf("Enter the number of elements in the array: ");
   scanf("%d", &n);
   // Get array elements from the user
   printf("Enter %d elements:\n", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   // Reverse the array in-place
   for(i = 0; i < n / 2; i++) {
      temp = arr[i];
      arr[i] = arr[n - i - 1];
      arr[n - i - 1] = temp;
   }
   // Print the reversed array
   printf("Reversed array:\n");
   for(i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
   return 0;
}
```

Output:

```
Output

Enter the number of elements in the array: 8
Enter 8 elements:
41
5
6
1365
4128
125
1
5
Reversed array:
5 1 125 4128 1365 6 5 41
```

ii) C Programs to implement the Searching Techniques – Linear & Binary Search

C Program to implement the Searching Technique – Linear Search

Linear Search (works on unsorted arrays)

Program:

```
#include <stdio.h>
int main() {

   int arr[100], n, i, key, found = 0;

   printf("Enter number of elements: ");

   scanf("%d", &n);

   printf("Enter %d elements:\n", n);

   for(i = 0; i < n; i++) {

      scanf("%d", &arr[i]);

   }

   printf("Enter the element to search: ");

   scanf("%d", &key);

   for(i = 0; i < n; i++) {

      if(arr[i] == key) {

         found = 1;
```

```
            printf("Element found at position %d (index %d)\n", i + 1, i);

            break;

        }

    }
    if(!found) {

        printf("Element not found in the array.\n");

    }

    return 0;

}
```

Output(s):

Output

```
Enter number of elements: 5
Enter 5 elements:
41
4
5
6

7
Enter the element to search: 7
Element found at position 5 (index 4)
```

Output

```
Enter number of elements: 5
Enter 5 elements:
12
4
8
55
55
Enter the element to search: 55
Element found at position 4 (index 3)
```

Output

```
Enter number of elements: 5
Enter 5 elements:
41
8
4
11
1
Enter the element to search: 2
Element not found in the array.
```

C Program to implement the Searching Technique - Binary Search

Binary Search (only works on sorted arrays)

Program:

```
#include <stdio.h>

int main() {

   int arr[100], n, i, key, low, high, mid;

   printf("Enter number of elements: ");

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

   }
   printf("Enter the element to search: ");

   scanf("%d", &key);
   low = 0;
   high = n - 1;
   while(low <= high) {

      mid = (low + high) / 2;
      if(arr[mid] == key) {
```

```
            printf("Element found at position %d (index %d)\n", mid + 1, mid);

            return 0;

        } else if(arr[mid] < key) {

            low = mid + 1;

        } else {

            high = mid - 1;

        }

    }
    printf("Element not found in the array.\n");

    return 0;

}
```

Output(s):

Output

```
Enter number of elements: 5
Enter 5 elements in sorted order:
1
2
3
4
5
Enter the element to search: 2
Element found at position 2 (index 1)
```

Output

```
Enter number of elements: 5
Enter 5 elements in sorted order:
1
5
7
9
11
Enter the element to search: 12
Element not found in the array.
```

iii) C Programs to implement Sorting Techniques – Bubble, Selection and Insertion Sort

Bubble Sort in C

```
#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 elements
            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 arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Original array: ");
```

```
    printArray(arr, n);

    bubbleSort(arr, n);

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

    return 0;
}
```

Output:

```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

Selection sort in C

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
   int i, j, min_idx, temp;
   for (i = 0; i < n - 1; i++) {
      min_idx = i;
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[min_idx]) {
            min_idx = j;
         }
      }
      // Swap the found minimum element with the first element
      temp = arr[min_idx];
      arr[min_idx] = 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 arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Original array: ");
```

```
    printArray(arr, n);

    selectionSort(arr, n);

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

    return 0;
}
```

Output:

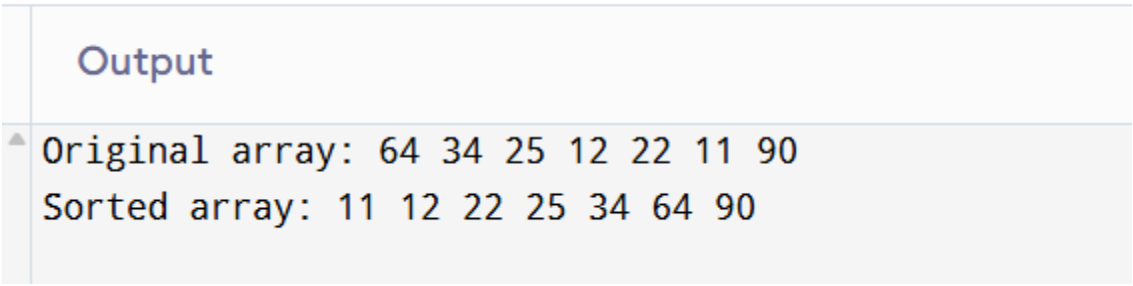
```
Output

Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

Insertion Sort in C

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      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 arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);

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

```
    insertionSort(arr, n);

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

    return 0;
}
```

Output:



Exercise 2: Linked List Implementation

i) Implement a singly linked list and perform insertion and deletion operations.

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

// Structure for a node in the singly linked list
struct Node {
   int data;
   struct Node* next;
};

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

// Insertion at the beginning
void insertAtBeginning(struct Node** head, int data) {
   struct Node* newNode = createNode(data);
   newNode->next = *head;
   *head = newNode;
   printf("Inserted %d at the beginning\n", data);
}
```

```
// Insertion at the end
void insertAtEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
      printf("Inserted %d at the end\n", data);
      return;
   }
   struct Node* temp = *head;
   while (temp->next != NULL) {
      temp = temp->next;
   }
   temp->next = newNode;
   printf("Inserted %d at the end\n", data);
}

// Insertion after a given node (by value)
void insertAfter(struct Node* head, int afterData, int data) {
   struct Node* temp = head;
   while (temp != NULL && temp->data != afterData) {
      temp = temp->next;
   }
   if (temp == NULL) {
      printf("Node with data %d not found\n", afterData);
      return;
   }
   struct Node* newNode = createNode(data);
   newNode->next = temp->next;
   temp->next = newNode;
   printf("Inserted %d after %d\n", data, afterData);
}
```

```
// Deletion of the first node
void deleteFirst(struct Node** head) {
   if (*head == NULL) {
      printf("List is empty\n");
      return;
   }
   struct Node* temp = *head;
   *head = (*head)->next;
   printf("Deleted %d from the beginning\n", temp->data);
   free(temp);
}

// Deletion of the last node
void deleteLast(struct Node** head) {
   if (*head == NULL) {
      printf("List is empty\n");
      return;
   }
   if ((*head)->next == NULL) {
      printf("Deleted %d from the end\n", (*head)->data);
      free(*head);
      *head = NULL;
      return;
   }
   struct Node* temp = *head;
   while (temp->next->next != NULL) {
      temp = temp->next;
   }
   printf("Deleted %d from the end\n", temp->next->data);
   free(temp->next);
```

```
    temp->next = NULL;
}

// Deletion of a node with a specific value
void deleteByValue(struct Node** head, int data) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    if ((*head)->data == data) {
        struct Node* temp = *head;
        *head = (*head)->next;
        printf("Deleted %d\n", data);
        free(temp);
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->data != data) {
        temp = temp->next;
    }
    if (temp->next == NULL) {
        printf("Node with data %d not found\n", data);
        return;
    }
    struct Node* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    printf("Deleted %d\n", data);
    free(nodeToDelete);
}

// Display the linked list
```

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

// Main function to test the linked list operations
int main() {
    struct Node* head = NULL;

    // Insert some nodes
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    displayList(head);

    // Insert at the beginning
    insertAtBeginning(&head, 5);
    displayList(head);

    // Insert after a node
    insertAfter(head, 20, 25);
    displayList(head);
```

```
    // Delete operations
    deleteFirst(&head);
    displayList(head);

    deleteLast(&head);
    displayList(head);

    deleteByValue(&head, 20);
    displayList(head);

    return 0;
}
```

Output:

Output

```
Inserted 10 at the end
Inserted 20 at the end
Inserted 30 at the end
Linked List: 10 -> 20 -> 30 -> NULL
Inserted 5 at the beginning
Linked List: 5 -> 10 -> 20 -> 30 -> NULL
Inserted 25 after 20
Linked List: 5 -> 10 -> 20 -> 25 -> 30 -> NULL
Deleted 5 from the beginning
Linked List: 10 -> 20 -> 25 -> 30 -> NULL
Deleted 30 from the end
Linked List: 10 -> 20 -> 25 -> NULL
Deleted 20
Linked List: 10 -> 25 -> NULL
```

ii) **Develop a program to reverse a linked list iteratively and recursively.**

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

// Structure for a node in the singly linked list
struct Node {
   int data;
   struct Node* next;
};

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

// Function to insert a node at the end of the list
void insertAtEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
      return;
   }
   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;
    }
    printf("Linked List: ");
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Iterative function to reverse the linked list
void reverseIterative(struct Node** head) {
    struct Node *prev = NULL, *current = *head, *next = NULL;
    while (current != NULL) {
        // Store next
        next = current->next;
        // Reverse current node's pointer
        current->next = prev;
        // Move pointers one position ahead
        prev = current;
        current = next;
```

```
    }
    *head = prev;
    printf("Linked list reversed iteratively\n");
}

// Recursive function to reverse the linked list
struct Node* reverseRecursiveUtil(struct Node* current, struct Node* prev) {
    if (current == NULL) {
        return prev;
    }
    // Store next
    struct Node* next = current->next;
    // Reverse current node's pointer
    current->next = prev;
    // Recurse for the remaining list
    return reverseRecursiveUtil(next, current);
}

void reverseRecursive(struct Node** head) {
    *head = reverseRecursiveUtil(*head, NULL);
    printf("Linked list reversed recursively\n");
}

// Main function to test the linked list reversal
int main() {
    struct Node* head = NULL;

    // Create a linked list: 10 -> 20 -> 30 -> 40 -> 50
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
```

```
    insertAtEnd(&head, 40);
    insertAtEnd(&head, 50);

    printf("Original ");
    displayList(head);

    // Reverse the linked list iteratively
    reverseIterative(&head);
    displayList(head);

    // Recreate the linked list for recursive reversal
    head = NULL;
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);
    insertAtEnd(&head, 50);

    printf("\nOriginal ");
    displayList(head);

    // Reverse the linked list recursively
    reverseRecursive(&head);
    displayList(head);

    return 0;
}
```

Output:

```
Output
Original Linked List: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
Linked list reversed iteratively
Linked List: 50 -> 40 -> 30 -> 20 -> 10 -> NULL

Original Linked List: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
Linked list reversed recursively
Linked List: 50 -> 40 -> 30 -> 20 -> 10 -> NULL
```

iv) Solve problems involving linked list traversal and manipulation.

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

// Structure for a node in the singly linked list
struct Node {
  int data;
  struct Node* next;
};

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

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int data) {
  struct Node* newNode = createNode(data);
  if (*head == NULL) {
    *head = newNode;
    return;
  }
  struct Node* temp = *head;
  while (temp->next != NULL) {
```

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

// 1. Traversal: Print all elements in the linked list
void traverseList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Linked List: ");
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// 2. Find the length of the linked list
int getLength(struct Node* head) {
    int length = 0;
    struct Node* temp = head;
    while (temp != NULL) {
        length++;
        temp = temp->next;
    }
    return length;
}
```

```
// 3. Search for an element in the linked list
int searchElement(struct Node* head, int key) {
   struct Node* temp = head;
   int position = 1;
   while (temp != NULL) {
      if (temp->data == key) {
         return position;
      }
      temp = temp->next;
      position++;
   }
   return -1; // Element not found
}

// 4. Find the middle element of the linked list (Slow and Fast Pointer)
int findMiddle(struct Node* head) {
   if (head == NULL) {
      printf("List is empty\n");
      return -1;
   }
   struct Node *slow = head, *fast = head;
   while (fast != NULL && fast->next != NULL) {
      slow = slow->next;
      fast = fast->next->next;
   }
   return slow->data;
}

// 5. Detect a loop in the linked list (Floyd's Cycle Detection)
int detectLoop(struct Node* head) {
   struct Node *slow = head, *fast = head;
```

```
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return 1; // Loop detected
        }
    }
    return 0; // No loop
}

// 6. Merge two sorted linked lists
struct Node* mergeSortedLists(struct Node* head1, struct Node* head2) {
    struct Node dummy;
    struct Node* tail = &dummy;
    dummy.next = NULL;

    while (head1 != NULL && head2 != NULL) {
        if (head1->data <= head2->data) {
            tail->next = head1;
            head1 = head1->next;
        } else {
            tail->next = head2;
            head2 = head2->next;
        }
        tail = tail->next;
    }
    // Attach remaining nodes
    if (head1 != NULL) {
        tail->next = head1;
    } else {
        tail->next = head2;
```

```
    }
    return dummy.next;
}

// Main function to test the linked list operations
int main() {
    // Create first linked list: 1 -> 3 -> 5 -> 7
    struct Node* head1 = NULL;
    insertAtEnd(&head1, 1);
    insertAtEnd(&head1, 3);
    insertAtEnd(&head1, 5);
    insertAtEnd(&head1, 7);

    // Create second linked list: 2 -> 4 -> 6
    struct Node* head2 = NULL;
    insertAtEnd(&head2, 2);
    insertAtEnd(&head2, 4);
    insertAtEnd(&head2, 6);

    // 1. Traversal
    printf("First ");
    traverseList(head1);
    printf("Second ");
    traverseList(head2);

    // 2. Length of the first linked list
    printf("Length of first linked list: %d\n", getLength(head1));

    // 3. Search for an element
    int key = 5;
    int pos = searchElement(head1, key);
```

```
    if (pos != -1) {
        printf("Element %d found at position %d in first list\n", key, pos);
    } else {
        printf("Element %d not found in first list\n", key);
    }

    // 4. Find middle element
    printf("Middle element of first list: %d\n", findMiddle(head1));

    // 5. Detect loop (create a loop for testing)
    struct Node* temp = head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head1->next; // Create a loop (7 -> 3)
    printf("Loop detected in first list: %s\n", detectLoop(head1) ? "Yes" : "No");
    temp->next = NULL; // Remove the loop for further operations

    // 6. Merge two sorted linked lists
    printf("\nMerging two sorted lists:\n");
    struct Node* mergedHead = mergeSortedLists(head1, head2);
    traverseList(mergedHead);

    return 0;
}
```

Output:

```
First Linked List: 1 -> 3 -> 5 -> 7 -> NULL
Second Linked List: 2 -> 4 -> 6 -> NULL
Length of first linked list: 4
Element 5 found at position 3 in first list
Middle element of first list: 5
Loop detected in first list: Yes

Merging two sorted lists:
Linked List: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> NULL
```

Exercise 3: Linked List Applications

i) Create a program to detect and remove duplicates from a linked list

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

// Structure for a node in the singly linked list
struct Node {
   int data;
   struct Node* next;
};

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

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
      return;
```

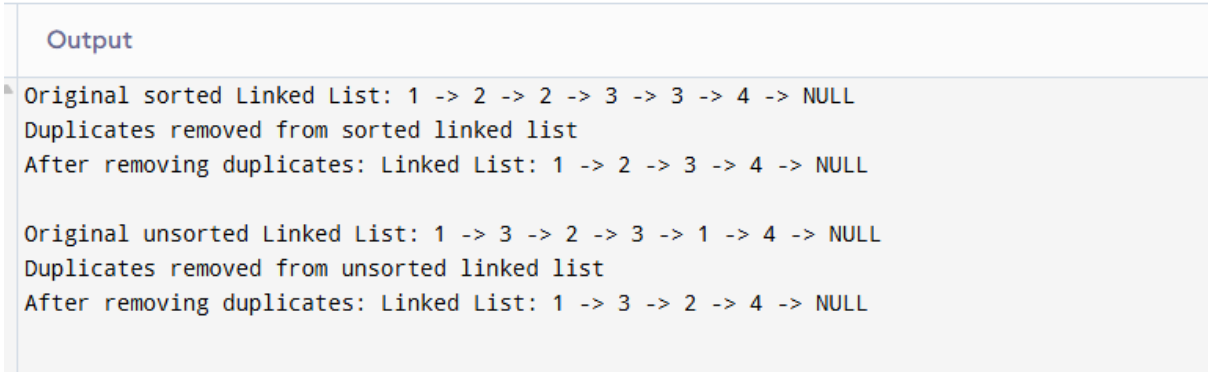
```
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

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

// 1. Remove duplicates from a sorted linked list
void removeDuplicatesSorted(struct Node* head) {
    if (head == NULL || head->next == NULL) {
        return; // No duplicates possible
    }
    struct Node* current = head;
    struct Node* nextNode;
    while (current != NULL && current->next != NULL) {
```

```
        if (current->data == current->next->data) {
            nextNode = current->next;
            current->next = nextNode->next;
            free(nextNode); // Free the duplicate node
        } else {
            current = current->next;
        }
    }
    printf("Duplicates removed from sorted linked list\n");
}

// 2. Remove duplicates from an unsorted linked list
// Using a simple array-based hash table for demonstration
#define MAX_SIZE 100 // Assuming data values are in range [0, MAX_SIZE)
void removeDuplicatesUnsorted(struct Node** head) {
    if (*head == NULL || (*head)->next == NULL) {
        return; // No duplicates possible
    }
    // Initialize hash table (array) to track seen values
    bool seen[MAX_SIZE] = {false};
    struct Node* current = *head;
    struct Node* prev = NULL;
    struct Node* temp;

    while (current != NULL) {
        // If current data is already seen, remove the node
        if (seen[current->data]) {
            temp = current;
            prev->next = current->next;
            current = current->next;
            free(temp);
```

```
        } else {
            // Mark data as seen and move to next node
            seen[current->data] = true;
            prev = current;
            current = current->next;
        }
    }
    printf("Duplicates removed from unsorted linked list\n");
}

// Main function to test the linked list operations
int main() {
    // Test 1: Sorted linked list with duplicates
    struct Node* sortedHead = NULL;
    insertAtEnd(&sortedHead, 1);
    insertAtEnd(&sortedHead, 2);
    insertAtEnd(&sortedHead, 2);
    insertAtEnd(&sortedHead, 3);
    insertAtEnd(&sortedHead, 3);
    insertAtEnd(&sortedHead, 4);

    printf("Original sorted ");
    displayList(sortedHead);
    removeDuplicatesSorted(sortedHead);
    printf("After removing duplicates: ");
    displayList(sortedHead);

    // Test 2: Unsorted linked list with duplicates
    struct Node* unsortedHead = NULL;
    insertAtEnd(&unsortedHead, 1);
    insertAtEnd(&unsortedHead, 3);
```

```
    insertAtEnd(&unsortedHead, 2);
    insertAtEnd(&unsortedHead, 3);
    insertAtEnd(&unsortedHead, 1);
    insertAtEnd(&unsortedHead, 4);

    printf("\nOriginal unsorted ");
    displayList(unsortedHead);
    removeDuplicatesUnsorted(&unsortedHead);
    printf("After removing duplicates: ");
    displayList(unsortedHead);

    return 0;
}
```

Output:



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

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

// Structure for a node representing a term in the polynomial
struct Node {
   int coeff; // Coefficient of the term
   int exp; // Exponent of the term
   struct Node* next;
};

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

// Function to insert a term into the polynomial in descending order of exponents
void insertTerm(struct Node** head, int coeff, int exp) {
   // Skip terms with zero coefficient
   if (coeff == 0) {
      return;
   }
```

```
struct Node* newNode = createNode(coeff, exp);
// If list is empty or new term has higher exponent
if (*head == NULL || (*head)->exp < exp) {
   newNode->next = *head;
   *head = newNode;
   return;
}
struct Node* current = *head;
struct Node* prev = NULL;
// Find the correct position to insert
while (current != NULL && current->exp > exp) {
   prev = current;
   current = current->next;
}
// If term with same exponent exists, add coefficients
if (current != NULL && current->exp == exp) {
   current->coeff += coeff;
   free(newNode); // Free the new node as it's not needed
   // If coefficient becomes zero, remove the term
   if (current->coeff == 0) {
      if (prev == NULL) {
         *head = current->next;
      } else {
         prev->next = current->next;
      }
      free(current);
   }
   return;
}
// Insert the new node
newNode->next = current;
```

```
    if (prev == NULL) {
        *head = newNode;
    } else {
        prev->next = newNode;
    }
}

// Function to display the polynomial
void displayPolynomial(struct Node* head) {
    if (head == NULL) {
        printf("0\n");
        return;
    }
    struct Node* temp = head;
    while (temp != NULL) {
        // Print coefficient
        if (temp->coeff > 0 && temp != head) {
            printf(" + ");
        } else if (temp->coeff < 0) {
            printf(" - ");
        }
        // Print absolute value of coefficient
        if (abs(temp->coeff) != 1 || temp->exp == 0) {
            printf("%d", abs(temp->coeff));
        } else if (temp->coeff == -1 && temp->exp != 0) {
            printf("1");
        }
        // Print variable and exponent
        if (temp->exp > 0) {
            printf("x");
            if (temp->exp > 1) {
```

```
            printf("^%d", temp->exp);
        }
      }
      temp = temp->next;
   }
   printf("\n");
}

// Function to add two polynomials
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
   struct Node* result = NULL;
   struct Node *p1 = poly1, *p2 = poly2;

   while (p1 != NULL && p2 != NULL) {
      if (p1->exp > p2->exp) {
         insertTerm(&result, p1->coeff, p1->exp);
         p1 = p1->next;
      } else if (p1->exp < p2->exp) {
         insertTerm(&result, p2->coeff, p2->exp);
         p2 = p2->next;
      } else {
         // Add coefficients for same exponent
         int sumCoeff = p1->coeff + p2->coeff;
         if (sumCoeff != 0) {
            insertTerm(&result, sumCoeff, p1->exp);
         }
         p1 = p1->next;
         p2 = p2->next;
      }
   }
   // Add remaining terms from poly1
```

```
    while (p1 != NULL) {
        insertTerm(&result, p1->coeff, p1->exp);
        p1 = p1->next;
    }
    // Add remaining terms from poly2
    while (p2 != NULL) {
        insertTerm(&result, p2->coeff, p2->exp);
        p2 = p2->next;
    }
    return result;
}

// Main function to test polynomial addition
int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;

    // Create first polynomial: 3x^3 + 2x^2 + 1
    insertTerm(&poly1, 3, 3);
    insertTerm(&poly1, 2, 2);
    insertTerm(&poly1, 1, 0);

    // Create second polynomial: 4x^2 + 5x^1 + 2
    insertTerm(&poly2, 4, 2);
    insertTerm(&poly2, 5, 1);
    insertTerm(&poly2, 2, 0);

    // Display polynomials
    printf("Polynomial 1: ");
    displayPolynomial(poly1);
```

```
    printf(" polynomial 2: ");
    displayPolynomial(poly2);

    // Add polynomials
    result = addPolynomials(poly1, poly2);
    printf("Result of addition: ");
    displayPolynomial(result);

    return 0;
}
```

Output:

```
Polynomial 1: 3x^3 + 2x^2 + 1
 polynomial 2: 4x^2 + 5x + 2
Result of addition: 3x^3 + 6x^2 + 5x + 3
```

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

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

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

// Structure for the deque
struct Deque {
   struct Node* front;
   struct Node* rear;
};

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

```
// Function to initialize an empty deque
struct Deque* createDeque() {
    struct Deque* deque = (struct Deque*)malloc(sizeof(struct Deque));
    if (deque == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    deque->front = NULL;
    deque->rear = NULL;
    return deque;
}

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

// Function to insert an element at the front
void insertFront(struct Deque* deque, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(deque)) {
        deque->front = deque->rear = newNode;
    } else {
        newNode->next = deque->front;
        deque->front->prev = newNode;
        deque->front = newNode;
    }
    printf("Inserted %d at front\n", data);
}
```

```
// Function to insert an element at the rear
void insertRear(struct Deque* deque, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(deque)) {
        deque->front = deque->rear = newNode;
    } else {
        newNode->prev = deque->rear;
        deque->rear->next = newNode;
        deque->rear = newNode;
    }
    printf("Inserted %d at rear\n", data);
}

// Function to delete an element from the front
int deleteFront(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty, cannot delete from front\n");
        return -1; // Indicate error
    }
    struct Node* temp = deque->front;
    int data = temp->data;
    deque->front = deque->front->next;
    if (deque->front == NULL) {
        deque->rear = NULL; // Deque is now empty
    } else {
        deque->front->prev = NULL;
    }
    free(temp);
    printf("Deleted %d from front\n", data);
    return data;
}
```

```
// Function to delete an element from the rear
int deleteRear(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty, cannot delete from rear\n");
        return -1; // Indicate error
    }
    struct Node* temp = deque->rear;
    int data = temp->data;
    deque->rear = deque->rear->prev;
    if (deque->rear == NULL) {
        deque->front = NULL; // Deque is now empty
    } else {
        deque->rear->next = NULL;
    }
    free(temp);
    printf("Deleted %d from rear\n", data);
    return data;
}

// Function to get the front element
int getFront(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty, no front element\n");
        return -1; // Indicate error
    }
    return deque->front->data;
}

// Function to get the rear element
int getRear(struct Deque* deque) {
```

```
    if (isEmpty(deque)) {
        printf("Deque is empty, no rear element\n");
        return -1; // Indicate error
    }
    return deque->rear->data;
}

// Function to display the deque
void displayDeque(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty\n");
        return;
    }
    printf("Deque: ");
    struct Node* temp = deque->front;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function to test the deque operations
int main() {
    struct Deque* deque = createDeque();

    // Test insertion operations
    insertFront(deque, 10); // 10
    insertRear(deque, 20); // 10 20
    insertFront(deque, 5); // 5 10 20
    insertRear(deque, 30); // 5 10 20 30
```

```
    displayDeque(deque);

    // Test get operations
    printf("Front element: %d\n", getFront(deque)); // 5
    printf("Rear element: %d\n", getRear(deque)); // 30

    // Test deletion operations
    deleteFront(deque); // Remove 5, deque: 10 20 30
    deleteRear(deque); // Remove 30, deque: 10 20
    displayDeque(deque);

    // Test empty deque
    deleteFront(deque); // Remove 10, deque: 20
    deleteRear(deque); // Remove 20, deque: empty
    displayDeque(deque);

    // Test operations on empty deque
    deleteFront(deque); // Should print error
    printf("Front element: %d\n", getFront(deque)); // Should print error

    // Add more elements to test further
    insertFront(deque, 40); // 40
    insertRear(deque, 50); // 40 50
    displayDeque(deque);

    return 0;
}
```

Output:

```
Inserted 10 at front
Inserted 20 at rear
Inserted 5 at front
Inserted 30 at rear
Deque: 5 10 20 30
Front element: 5
Rear element: 30
Deleted 5 from front
Deleted 30 from rear
Deque: 10 20
Deleted 10 from front
Deleted 20 from rear
Deque is empty
Deque is empty, cannot delete from front
Deque is empty, no front element
Front element: -1
Inserted 40 at front
Inserted 50 at rear
Deque: 40 50
```

Exercise 4: Double Linked List Implementation

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

```
// C program to implement all functions
// used in Doubly Linked List

#include <stdio.h>
#include <stdlib.h>
int i = 0;

// Node for Doubly Linked List
typedef struct node {
        int key;
        struct node* prev;
        struct node* next;

} node;

// Head, Tail, first & temp Node
node* head = NULL;
node* first = NULL;
node* temp = NULL;
node* tail = NULL;

// Function to add a node in the
// Doubly Linked List
void addnode(int k)
{

        // Allocating memory
        // to the Node ptr
```

```
        node* ptr
                = (node*)malloc(sizeof(node));

        // Assign Key to value k
        ptr->key = k;

        // Next and prev pointer to NULL
        ptr->next = NULL;
        ptr->prev = NULL;

        // If Linked List is empty
        if (head == NULL) {
                head = ptr;
                first = head;
                tail = head;
        }

        // Else insert at the end of the
        // Linked List
        else {
                temp = ptr;
                first->next = temp;
                temp->prev = first;
                first = temp;
                tail = temp;
        }

        // Increment for number of Nodes
        // in the Doubly Linked List
        i++;
}
```

```
// Function to traverse the Doubly
// Linked List
void traverse()
{
	// Nodes points towards head node
	node* ptr = head;

	// While pointer is not NULL,
	// traverse and print the node
	while (ptr != NULL) {

		// Print key of the node
		printf("%d ", ptr->key);
		ptr = ptr->next;
	}

	printf("\n");
}

// Function to insert a node at the
// beginning of the linked list
void insertatbegin(int k)
{

	// Allocating memory
	// to the Node ptr
	node* ptr
		= (node*)malloc(sizeof(node));

	// Assign Key to value k
```

```
        ptr->key = k;

        // Next and prev pointer to NULL
        ptr->next = NULL;
        ptr->prev = NULL;

        // If head is NULL
        if (head == NULL) {
                first = ptr;
                first = head;
                tail = head;
        }

        // Else insert at beginning and
        // change the head to current node
        else {
                temp = ptr;
                temp->next = head;
                head->prev = temp;
                head = temp;
        }
        i++;
}

// Function to insert Node at end
void insertatend(int k)
{

        // Allocating memory
        // to the Node ptr
        node* ptr
```

```
                = (node*)malloc(sizeof(node));

        // Assign Key to value k
        ptr->key = k;

        // Next and prev pointer to NULL
        ptr->next = NULL;
        ptr->prev = NULL;

        // If head is NULL
        if (head == NULL) {
                first = ptr;
                first = head;
                tail = head;
        }

        // Else insert at the end
        else {
                temp = ptr;
                temp->prev = tail;
                tail->next = temp;
                tail = temp;
        }
        i++;
}

// Function to insert Node at any
// position pos
void insertatpos(int k, int pos)
{
```

```
// For Invalid Position
if (pos < 1 || pos > i + 1) {
        printf("Please enter a"
                " valid position\n");
}

// If position is at the front,
// then call insertatbegin()
else if (pos == 1) {
        insertatbegin(k);
}

// Position is at length of Linked
// list + 1, then insert at the end
else if (pos == i + 1) {
        insertatend(k);
}

// Else traverse till position pos
// and insert the Node
else {
        node* src = head;

        // Move head pointer to pos
        while (pos--) {
                src = src->next;
        }

        // Allocate memory to new Node
        node **da, **ba;
        node* ptr
```

```
                        = (node*)malloc(
                                sizeof(node));
                ptr->next = NULL;
                ptr->prev = NULL;
                ptr->key = k;

                // Change the previous and next
                // pointer of the nodes inserted
                // with previous and next node
                ba = &src;
                da = &(src->prev);
                ptr->next = (*ba);
                ptr->prev = (*da);
                (*da)->next = ptr;
                (*ba)->prev = ptr;
                i++;
        }
}

// Function to delete node at the
// beginning of the list
void delatbegin()
{
        // Move head to next and
        // decrease length by 1
        head = head->next;
        i--;
}

// Function to delete at the end
// of the list
```

```
void delatend()
{
	// Mode tail to the prev and
	// decrease length by 1
	tail = tail->prev;
	tail->next = NULL;
	i--;
}

// Function to delete the node at
// a given position pos
void delatpos(int pos)
{

	// If invalid position
	if (pos < 1 || pos > i + 1) {
		printf("Please enter a"
			" valid position\n");
	}

	// If position is 1, then
	// call delatbegin()
	else if (pos == 1) {
		delatbegin();
	}

	// If position is at the end, then
	// call delatend()
	else if (pos == i) {
		delatend();
	}
```

```
        // Else traverse till pos, and
        // delete the node at pos
        else {
                // Src node to find which
                // node to be deleted
                node* src = head;
                pos--;

                // Traverse node till pos
                while (pos--) {
                        src = src->next;
                }

                // previous and after node
                // of the src node
                node **pre, **aft;
                pre = &(src->prev);
                aft = &(src->next);

                // Change the next and prev
                // pointer of pre and aft node
                (*pre)->next = (*aft);
                (*aft)->prev = (*pre);

                // Decrease the length of the
                // Linked List
                i--;
        }
}
```

```
// Driver Code
int main()
{
	// Adding node to the linked List
	addnode(2);
	addnode(4);
	addnode(9);
	addnode(1);
	addnode(21);
	addnode(22);

	// To print the linked List
	printf("Linked List: ");
	traverse();

	printf("\n");

	// To insert node at the beginning
	insertatbegin(1);
	printf("Linked List after"
		" inserting 1 "
		"at beginning: ");
	traverse();

	// To insert at the end
	insertatend(0);
	printf("Linked List after "
		"inserting 0 at end: ");
	traverse();

	// To insert Node with
```

```
    // value 44 after 3rd Node
    insertatpos(44, 3);
    printf("Linked List after "
           "inserting 44 "
           "after 3rd Node: ");
    traverse();

    printf("\n");

    // To delete node at the beginning
    delatbegin();
    printf("Linked List after "
           "deleting node "
           "at beginning: ");
    traverse();

    // To delete at the end
    delatend();
    printf("Linked List after "
           "deleting node at end: ");
    traverse();

    // To delete Node at position 5
    printf("Linked List after "
           "deleting node "
           "at position 5: ");
    delatpos(5);
    traverse();

    return 0;
}
```

Output:

```
Linked List: 2 4 9 1 21 22

Linked List after inserting 1 at beginning: 1 2 4 9 1 2
1 22
Linked List after inserting 0 at end: 1 2 4 9 1 21 22 0

Linked List after inserting 44 after 3rd Node: 1 2 4 44
 9 1 21 22 0

Linked List after deleting node at beginning: 2 4 44 9
1 21 22 0
Linked List after deleting node at end: 2 4 44 9 1 21 2
2
Linked List after deleting node at position 5: 2 4 44 9
 21 22
```

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

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

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

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

// Function to insert a node at the front
void insertAtFront(struct Node** tail, int data) {
   struct Node* newNode = createNode(data);
   if (*tail == NULL) {
      *tail = newNode;
      newNode->next = newNode; // Point to itself
   } else {
      newNode->next = (*tail)->next; // New node points to head
```

```
        (*tail)->next = newNode; // Tail's next points to new node
    }
    printf("Inserted %d at front\n", data);
}

// Function to insert a node at the end
void insertAtEnd(struct Node** tail, int data) {
    struct Node* newNode = createNode(data);
    if (*tail == NULL) {
        *tail = newNode;
        newNode->next = newNode; // Point to itself
    } else {
        newNode->next = (*tail)->next; // New node points to head
        (*tail)->next = newNode; // Current tail points to new node
        *tail = newNode; // Update tail to new node
    }
    printf("Inserted %d at end\n", data);
}

// Function to delete the first node
void deleteFirst(struct Node** tail) {
    if (*tail == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* head = (*tail)->next;
    int data = head->data;
    if (head == *tail) { // Only one node
        free(head);
        *tail = NULL;
    } else {
```

```
        (*tail)->next = head->next; // Tail points to second node
        free(head);
    }
    printf("Deleted %d from front\n", data);
}

// Function to delete the last node
void deleteLast(struct Node** tail) {
    if (*tail == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = (*tail)->next;
    int data = (*tail)->data;
    if (current == *tail) { // Only one node
        free(*tail);
        *tail = NULL;
    } else {
        // Traverse to the second-to-last node
        while (current->next != *tail) {
            current = current->next;
        }
        current->next = (*tail)->next; // Point to head
        free(*tail);
        *tail = current; // Update tail
    }
    printf("Deleted %d from end\n", data);
}

// Function to delete a node by value
void deleteByValue(struct Node** tail, int data) {
```

```
    if (*tail == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = (*tail)->next;
    struct Node* prev = *tail;
    // If head node has the data
    if (current->data == data) {
        deleteFirst(tail);
        return;
    }
    // Search for the node
    do {
        if (current->data == data) {
            prev->next = current->next;
            if (current == *tail) {
                *tail = prev; // Update tail if last node is deleted
            }
            free(current);
            printf("Deleted %d\n", data);
            return;
        }
        prev = current;
        current = current->next;
    } while (current != (*tail)->next);
    printf("Node with data %d not found\n", data);
}

// Function to traverse the circular linked list
void traverse(struct Node* tail) {
    if (tail == NULL) {
```

```
        printf("List is empty\n");
        return;
    }
    printf("Circular Linked List: ");
    struct Node* current = tail->next; // Start from head
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != tail->next);
    printf("\n");
}

// Function to get the length of the list
int getLength(struct Node* tail) {
    if (tail == NULL) {
        return 0;
    }
    int length = 0;
    struct Node* current = tail->next;
    do {
        length++;
        current = current->next;
    } while (current != tail->next);
    return length;
}

// Main function to test the circular linked list operations
int main() {
    struct Node* tail = NULL;

    // Insert nodes
```

```
    insertAtFront(&tail, 10); // 10
    insertAtEnd(&tail, 20); // 10 -> 20
    insertAtFront(&tail, 5); // 5 -> 10 -> 20
    insertAtEnd(&tail, 30); // 5 -> 10 -> 20 -> 30
    traverse(tail);
    printf("Length: %d\n", getLength(tail));

    // Delete nodes
    deleteFirst(&tail); // 10 -> 20 -> 30
    traverse(tail);
    deleteLast(&tail); // 10 -> 20
    traverse(tail);
    deleteByValue(&tail, 20); // 10
    traverse(tail);

    // Test edge cases
    deleteByValue(&tail, 100); // Not found
    deleteFirst(&tail); // Empty
    traverse(tail);
    printf("Length: %d\n", getLength(tail));

    // Insert again to test
    insertAtEnd(&tail, 40); // 40
    insertAtFront(&tail, 50); // 50 -> 40
    traverse(tail);

    return 0;
}
```

Output:

Output

```
Inserted 10 at front
Inserted 20 at end
Inserted 5 at front
Inserted 30 at end
Circular Linked List: 5 10 20 30
Length: 4
Deleted 5 from front
Circular Linked List: 10 20 30
Deleted 30 from end
Circular Linked List: 10 20
Deleted 20
Circular Linked List: 10
Node with data 100 not found
Deleted 10 from front
List is empty
Length: 0
Inserted 40 at end
Inserted 50 at front
Circular Linked List: 50 40
```

Exercise 5: Stack Operations

i) Implement a stack using arrays and linked lists.

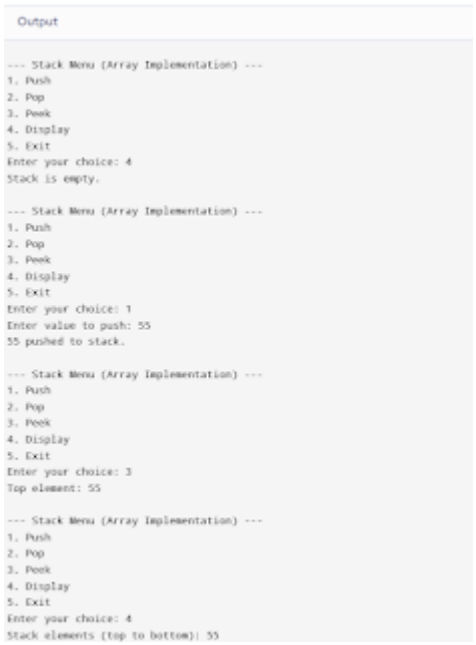
Stack using Array

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100



// Array-based stack

int stack[MAX];

int top = -1;



// Push operation

void pushArray(int value) {

  if (top == MAX - 1) {

    printf("Stack Overflow (Array)\n");

    return;
```

```
    }

    stack[++top] = value;

}



// Pop operation

int popArray() {

    if (top == -1) {

        printf("Stack Underflow (Array)\n");

        return -1;

    }

    return stack[top--];

}



// Peek operation

int peekArray() {

    if (top == -1) return -1;
```

```
    return stack[top];
}

// Display stack
void displayArrayStack() {
    printf("Array Stack: ");
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}
```

Output:



```
Output

--- Stack Menu (Array Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack is empty.

--- Stack Menu (Array Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 55
55 pushed to stack.

--- Stack Menu (Array Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element: 55

--- Stack Menu (Array Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements (top to bottom): 55
```

Stack using Linked List

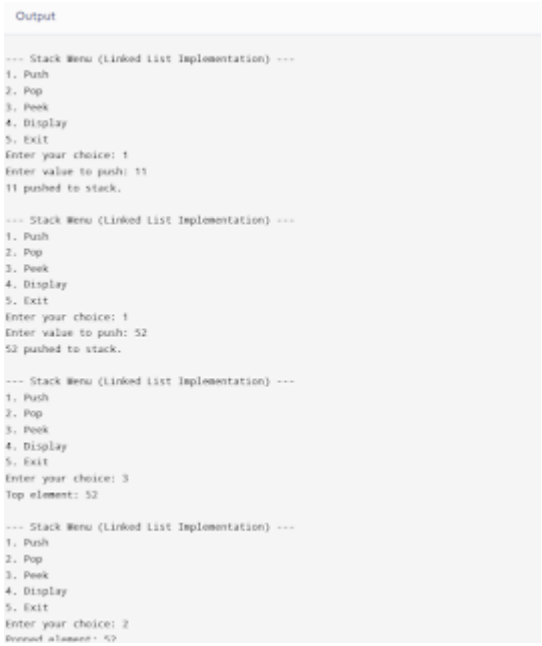
```
 #include <stdio.h>
#include <stdlib.h>

// Node structure for stack
struct Node {
   int data;
   struct Node* next;
};

// Top pointer for stack
struct Node* top = NULL;

// Push operation
void push(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (!newNode) {
      printf("Heap Overflow\n");
      return;
   }
   newNode->data = value;
   newNode->next = top;
   top = newNode;
   printf("%d pushed to stack.\n", value);
}

// Pop operation
int pop() {
   if (top == NULL) {
      printf("Stack Underflow! Cannot pop.\n");
```

```
        return -1;
    }
    int popped = top->data;
    struct Node* temp = top;
    top = top->next;
    free(temp);
    return popped;
}

// Peek operation
int peek() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return -1;
    }
    return top->data;
}

// Check if stack is empty
int isEmpty() {
    return top == NULL;
}

// Display the stack
void display() {
    if (isEmpty()) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack elements (top to bottom): ");
```

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

// Main function to test stack
int main() {
    int choice, value;

    while (1) {
        printf("\n--- Stack Menu (Linked List Implementation) ---\n");
        printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            value = pop();
            if (value != -1)
                printf("Popped element: %d\n", value);
            break;
        case 3:
            value = peek();
            if (value != -1)
```

```
            printf("Top element: %d\n", value);
            break;
        case 4:
            display();
            break;
        case 5:
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice! Please try again.\n");
        }
    }
}
```

Output:



```
Output

--- Stack Menu (Linked List Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 11
11 pushed to stack.

--- Stack Menu (Linked List Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 52
52 pushed to stack.

--- Stack Menu (Linked List Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element: 52

--- Stack Menu (Linked List Implementation) ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
```

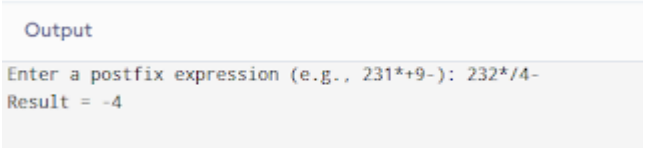
ii) Write a program to evaluate a postfix expression using a stack.

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

#define MAX 100

// Stack implementation
int stack[MAX];
int top = -1;

// Push function
void push(int value) {
   if (top == MAX - 1) {
      printf("Stack overflow!\n");
      exit(1);
   }
   stack[++top] = value;
}

// Pop function
int pop() {
   if (top == -1) {
      printf("Stack underflow!\n");
      exit(1);
   }
   return stack[top--];
```

```
}

// Function to evaluate postfix expression
int evaluatePostfix(char* exp) {
   for (int i = 0; exp[i]; i++) {
      char ch = exp[i];

      if (isspace(ch)) {
         continue;
      }
      else if (isdigit(ch)) {
         // Convert char digit to int and push
         push(ch - '0');
      }
      else {
         int val2 = pop();
         int val1 = pop();

         switch (ch) {
            case '+': push(val1 + val2); break;
            case '-': push(val1 - val2); break;
            case '*': push(val1 * val2); break;
            case '/': push(val1 / val2); break;
            case '^': push(pow(val1, val2)); break;
            default:
               printf("Invalid operator: %c\n", ch);
               exit(1);
         }
      }
   }
   return pop();  // Final result
```

```
}

// Main function
int main() {
    char expression[MAX];

    printf("Enter a postfix expression (e.g., 231*+9-): ");
    scanf("%s", expression);

    int result = evaluatePostfix(expression);
    printf("Result = %d\n", result);

    return 0;
}
```

Output:



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

```
#include <stdio.h>

#include <stdlib.h>



// Stack Node

struct StackNode {

   char data;

   struct StackNode* next;

};



// Push to stack

void push(struct StackNode** top, char data) {

   struct StackNode* new_node = (struct StackNode*)malloc(sizeof(struct StackNode));

   new_node->data = data;

   new_node->next = *top;

   *top = new_node;
```

```
}



// Pop from stack

char pop(struct StackNode** top) {

   if (*top == NULL) return '\0'; // stack underflow

   struct StackNode* temp = *top;

   char popped = temp->data;

   *top = temp->next;

   free(temp);

   return popped;

}



// Peek at top element

char peek(struct StackNode* top) {

   if (top == NULL) return '\0';
```

```
    return top->data;
}

// Check for matching pair
int isMatchingPair(char opening, char closing) {
    return (opening == '(' && closing == ')') ||
           (opening == '{' && closing == '}') ||
           (opening == '[' && closing == ']');
}

// Function to check balanced parentheses
int isBalanced(const char* expr) {
    struct StackNode* stack = NULL;

    for (int i = 0; expr[i] != '\0'; i++) {
        char ch = expr[i];
```

```
        if (ch == '(' || ch == '{' || ch == '[') {
            push(&stack, ch);
        } else if (ch == ')' || ch == '}' || ch == ']') {
            if (stack == NULL || !isMatchingPair(pop(&stack), ch)) {
                return 0; // Not balanced
            }
        }
    }

    // If stack is empty, it's balanced
    return stack == NULL;
}

// Driver code
```

```
int main() {
    char expression[100];

    printf("Enter an expression: ");
    scanf("%s", expression);

    if (isBalanced(expression))
        printf("Balanced\n");
    else
        printf("Not Balanced\n");

    return 0;
}
```

Output:

```
Output

Enter an expression: {[(())]}
Balanced
```

```
Output

Enter an expression: {[({})]}
Not Balanced
```

Exercise 6: Queue Operations

i) Implement a queue using arrays and linked lists

1. Queue Using Array (Linear Queue)

```
// C Program to implement queue using arrays
#include <stdio.h>
// Define the maximum size for the queue
#define MAX_SIZE 100

// Define a structure for the queue
struct Queue {
   int queue[MAX_SIZE];
   int front;
   int rear;
};

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

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

// Function to check if the queue is full
int isFull(struct Queue *q) {
   return (q->rear == MAX_SIZE - 1);
}
```

```
// Function to insert an element into the queue
void enqueue(struct Queue *q, int data) {
   if (isFull(q)) {
      printf("Queue is full\n");
      return;
   }
   if (isEmpty(q)) {
      q->front = 0;
   }
   q->rear++;
   q->queue[q->rear] = data;
   printf("Enqueued %d in queue\n", data);
}

// Function to remove an element from the queue
int dequeue(struct Queue *q) {
   if (isEmpty(q)) {
      printf("Queue is empty\n");
      return -1;
   }
   int data = q->queue[q->front];
   // If the queue is empty reset the pointers
   if (q->front == q->rear) {
      q->front = -1;
      q->rear = -1;
   } else {
      q->front++;
   }
   printf("Deleted element: %d\n", data);
   return data;
```

```
}

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

int main() {
    // Initialize a queue
    struct Queue q;
    initializeQueue(&q);

    enqueue(&q, 1);
    enqueue(&q, 2);
    enqueue(&q, 3);
    printf("Elements in the queue after enqueue operation: ");
    display(&q);

    dequeue(&q);
    printf("Elements in the queue after dequeue operation: ");
    display(&q);

    return 0;
}
```

Output:

```
Output

Enqueued 1 in queue
Enqueued 2 in queue
Enqueued 3 in queue
Elements in the queue after enqueue operation: 1 2 3
Deleted element: 1
Elements in the queue after dequeue operation: 2 3
```

2. Queue using Linked List

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

// Node structure definition
struct Node {
   int data;
   struct Node* next;
};

// Queue structure definition
struct Queue {
   struct Node* front;
   struct Node* rear;
};

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

// Function to initialize the queue
struct Queue* createQueue() {
   struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
   q->front = q->rear = NULL;
   return q;
}
```

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

// Function to print the current state of the queue
void printQueue(struct Queue* q) {
   if (isEmpty(q)) {
      printf("Queue is empty\n");
      return;
   }
   struct Node* temp = q->front;
   printf("Current Queue: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}

// Function to add an element to the queue
void enqueue(struct Queue* q, int new_data) {
   struct Node* new_node = newNode(new_data);
   if (isEmpty(q)) {
      q->front = q->rear = new_node;
   } else {
      q->rear->next = new_node;
      q->rear = new_node;
   }
   printQueue(q);
```

```
}

// Function to remove an element from the queue
void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Cannot dequeue. Queue is empty.\n");
        return;
    }
    struct Node* temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);
    printQueue(q);
}

// Driver code to test the queue implementation
int main() {
    struct Queue* q = createQueue();

    // Enqueue elements into the queue
    enqueue(q, 10);
    enqueue(q, 20);

    // Dequeue elements from the queue
    dequeue(q);
    dequeue(q);

    // Enqueue more elements into the queue
    enqueue(q, 30);
```

```
    enqueue(q, 40);
    enqueue(q, 50);

    // Dequeue an element from the queue
    dequeue(q);

    return 0;
}
    return q->front == NULL;
}

// Add an element to the queue (enqueue)
void linkedListEnqueue(LinkedListQueue *q, int value) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed.\n");
        return;
    }

    newNode->data = value;
    newNode->next = NULL;

    if (isLinkedListQueueEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }

    printf("Enqueued %d\n", value);
}
```

```
// Remove an element from the queue (dequeue)
int linkedListDequeue(LinkedListQueue *q) {
    if (isLinkedListQueueEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    Node *temp = q->front;
    int item = temp->data;

    q->front = q->front->next;

    // If queue becomes empty
    if (q->front == NULL) {
        q->rear = NULL;
    }

    free(temp);
    printf("Dequeued %d\n", item);
    return item;
}

// Display the queue
void displayLinkedListQueue(LinkedListQueue *q) {
    if (isLinkedListQueueEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Queue elements: ");
```

```
    Node *current = q->front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Free memory when done
void freeLinkedListQueue(LinkedListQueue *q) {
    while (!isLinkedListQueueEmpty(q)) {
        linkedListDequeue(q);
    }
}
```

Output:

Output

```
Current Queue: 10
Current Queue: 10 20
Current Queue: 20
Queue is empty
Current Queue: 30
Current Queue: 30 40
Current Queue: 30 40 50
Current Queue: 40 50
```

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

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

#define MAX_QUEUE_SIZE 10

// Structure to represent a print job
typedef struct {
   int job_id;
   int pages;
} PrintJob;

// Structure to represent the printer queue
typedef struct {
   PrintJob jobs[MAX_QUEUE_SIZE];
   int front, rear;
} PrinterQueue;

// Function to initialize the printer queue
void initializeQueue(PrinterQueue *queue) {
   queue->front = -1;
   queue->rear = -1;
}

// Function to check if the queue is empty
int isEmpty(PrinterQueue *queue) {
   return (queue->front == -1 && queue->rear == -1);
}

// Function to check if the queue is full
```

```
int isFull(PrinterQueue *queue) {
    return (queue->rear + 1) % MAX_QUEUE_SIZE == queue->front;
}

// Function to add a job to the queue (enqueue)
void enqueue(PrinterQueue *queue, PrintJob job) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot add more jobs.\n");
        return;
    }

    if (isEmpty(queue)) {
        queue->front = 0;
        queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_QUEUE_SIZE;
    }

    queue->jobs[queue->rear] = job;
    printf("Job %d with %d pages added to the queue.\n", job.job_id, job.pages);
}

// Function to remove a job from the queue (dequeue)
PrintJob dequeue(PrinterQueue *queue) {
    PrintJob job;

    if (isEmpty(queue)) {
        printf("Queue is empty. No jobs to dequeue.\n");
        job.job_id = -1; // Return a dummy job ID to indicate empty queue
        return job;
    }
```

```
    job = queue->jobs[queue->front];

    if (queue->front == queue->rear) {
        // Reset the queue when the last element is dequeued
        queue->front = -1;
        queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_QUEUE_SIZE;
    }

    printf("Job %d with %d pages removed from the queue.\n", job.job_id, job.pages);
    return job;
}

// Function to display all jobs in the queue
void displayQueue(PrinterQueue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Jobs in the queue:\n");
    int i = queue->front;
    while (i != queue->rear) {
        printf("Job %d with %d pages\n", queue->jobs[i].job_id, queue->jobs[i].pages);
        i = (i + 1) % MAX_QUEUE_SIZE;
    }
    printf("Job %d with %d pages\n", queue->jobs[i].job_id, queue->jobs[i].pages);
}
```

```
int main() {
    PrinterQueue queue;
    initializeQueue(&queue);

    // Example: Adding jobs to the queue
    PrintJob job1 = {1, 5};
    PrintJob job2 = {2, 3};
    PrintJob job3 = {3, 7};

    enqueue(&queue, job1);
    enqueue(&queue, job2);
    enqueue(&queue, job3);

    // Display the current queue
    displayQueue(&queue);

    // Example: Removing a job from the queue
    dequeue(&queue);

    // Display the updated queue
    displayQueue(&queue);

    return 0;
}
```

Output:

```
Output
Job 1 with 5 pages added to the queue.
Job 2 with 3 pages added to the queue.
Job 3 with 7 pages added to the queue.
Jobs in the queue:
Job 1 with 5 pages
Job 2 with 3 pages
Job 3 with 7 pages
Job 1 with 5 pages removed from the queue.
Jobs in the queue:
Job 2 with 3 pages
Job 3 with 7 pages
```

iii) Solve problems involving circular queues.

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

#define MAX_SIZE 5

typedef struct {
   int items[MAX_SIZE];
   int front;
   int rear;
   int size;
} CircularQueue;

// Initialize the circular queue
void initQueue(CircularQueue *q) {
   q->front = -1;
   q->rear = -1;
   q->size = 0;
}

// Check if the queue is empty
bool isEmpty(CircularQueue *q) {
   return q->size == 0;
}

// Check if the queue is full
bool isFull(CircularQueue *q) {
   return q->size == MAX_SIZE;
```

```
}

// Add an element to the queue (enqueue)
void enqueue(CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue %d.\n", value);
        return;
    }

    if (isEmpty(q)) {
        q->front = 0;
    }

    q->rear = (q->rear + 1) % MAX_SIZE;
    q->items[q->rear] = value;
    q->size++;
    printf("Enqueued %d\n", value);
}

// Remove an element from the queue (dequeue)
int dequeue(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    int item = q->items[q->front];
    if (q->front == q->rear) {
        // Queue has only one element, reset after removal
        q->front = -1;
        q->rear = -1;
```

```
    } else {
        q->front = (q->front + 1) % MAX_SIZE;
    }
    q->size--;
    printf("Dequeued %d\n", item);
    return item;
}

// Get the front element without removing it
int peek(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return -1;
    }
    return q->items[q->front];
}

// Display the queue
void display(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Queue elements: ");
    int i = q->front;
    int count = 0;

    while (count < q->size) {
        printf("%d ", q->items[i]);
        i = (i + 1) % MAX_SIZE;
```

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

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

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);

    display(&q);

    // This will fail because queue is full
    enqueue(&q, 60);

    dequeue(&q);
    dequeue(&q);

    display(&q);

    enqueue(&q, 60);
    enqueue(&q, 70);

    display(&q);

    printf("Front element: %d\n", peek(&q));
```

```
    return 0;
}
```

Output:

Output

```
Enqueued 10
Enqueued 20
Enqueued 30
Enqueued 40
Enqueued 50
Queue elements: 10 20 30 40 50
Queue is full. Cannot enqueue 60.
Dequeued 10
Dequeued 20
Queue elements: 30 40 50
Enqueued 60
Enqueued 70
Queue elements: 30 40 50 60 70
Front element: 30
```

Exercise 7: Stack and Queue Applications

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

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

#define MAX_SIZE 100 // Maximum stack size

// Structure for the stack
struct Stack {
   char items[MAX_SIZE]; // For infix-to-postfix (operators)
   int top;
};

// 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_SIZE - 1;
}
```

```
// Function to push an item onto the stack
void push(struct Stack* stack, char item) {
    if (isFull(stack)) {
        printf("Stack overflow!\n");
        exit(1);
    }
    stack->items[++stack->top] = item;
}

// Function to pop an item from the stack
char pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow!\n");
        exit(1);
    }
    return stack->items[stack->top--];
}

// Function to peek at the top item of the stack
char peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        return '\0';
    }
    return stack->items[stack->top];
}

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

```
// Function to get the precedence of an operator
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    }
    if (op == '*' || op == '/') {
        return 2;
    }
    return 0; // For parentheses or non-operators
}

// Function to convert infix expression to postfix
void infixToPostfix(const char* infix, char* postfix) {
    struct Stack stack;
    initStack(&stack);
    int j = 0; // Index for postfix string

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

        // Skip spaces
        if (isspace(c)) {
            continue;
        }

        // If operand (letter or digit), add to postfix
        if (isalnum(c)) {
            postfix[j++] = c;
        }
        // If opening parenthesis, push to stack
```

```
        else if (c == '(') {
            push(&stack, c);
        }
        // If closing parenthesis, pop until opening parenthesis
        else if (c == ')') {
            while (!isEmpty(&stack) && peek(&stack) != '(') {
                postfix[j++] = pop(&stack);
            }
            if (!isEmpty(&stack) && peek(&stack) == '(') {
                pop(&stack); // Remove '('
            }
        }
        // If operator
        else if (isOperator(c)) {
            while (!isEmpty(&stack) && precedence(peek(&stack)) >= precedence(c)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, c);
        }
    }

    // Pop remaining operators from stack
    while (!isEmpty(&stack)) {
        char op = pop(&stack);
        if (op != '(') {
            postfix[j++] = op;
        }
    }

    postfix[j] = '\0'; // Null-terminate the postfix string
}
```

```
// Structure for integer stack (for evaluation)
struct IntStack {
    int items[MAX_SIZE];
    int top;
};

// Integer stack operations
void initIntStack(struct IntStack* stack) {
    stack->top = -1;
}

int isIntEmpty(struct IntStack* stack) {
    return stack->top == -1;
}

int isIntFull(struct IntStack* stack) {
    return stack->top == MAX_SIZE - 1;
}

void pushInt(struct IntStack* stack, int item) {
    if (isIntFull(stack)) {
        printf("Stack overflow!\n");
        exit(1);
    }
    stack->items[++stack->top] = item;
}

int popInt(struct IntStack* stack) {
    if (isIntEmpty(stack)) {
        printf("Stack underflow!\n");
```

```
        exit(1);
    }
    return stack->items[stack->top--];
}

// Function to evaluate a postfix expression
int evaluatePostfix(const char* postfix, int values[]) {
    struct IntStack stack;
    initIntStack(&stack);

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

        // Skip spaces
        if (isspace(c)) {
            continue;
        }

        // If operand (digit), push its value
        if (isdigit(c)) {
            pushInt(&stack, values[c - '0']);
        }
        // If operator, pop two operands, compute, and push result
        else if (isOperator(c)) {
            int b = popInt(&stack); // Second operand
            int a = popInt(&stack); // First operand
            int result;
            switch (c) {
                case '+': result = a + b; break;
                case '-': result = a - b; break;
                case '*': result = a * b; break;
```

```
            case '/':
                if (b == 0) {
                    printf("Division by zero!\n");
                    exit(1);
                }
                result = a / b;
                break;
            }
            pushInt(&stack, result);
        }
    }

    return popInt(&stack); // Final result
}

// Main function to test infix-to-postfix conversion and evaluation
int main() {
    char infix[MAX_SIZE];
    char postfix[MAX_SIZE];
    int values[10]; // Store values for digits 0-9

    // Get infix expression
    printf("Enter an infix expression (e.g., (2 + 3) * 4): ");
    fgets(infix, MAX_SIZE, stdin);
    infix[strcspn(infix, "\n")] = '\0'; // Remove newline

    // Convert to postfix
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);

    // Get values for operands (assuming digits 0-9)
```

```
    printf("Enter values for operands (digits 0-9):\n");
    for (int i = 0; i < 10; i++) {
        printf("Value for %d: ", i);
        scanf("%d", &values[i]);
    }

    // Evaluate postfix expression
    int result = evaluatePostfix(postfix, values);
    printf("Evaluation result: %d\n", result);

    return 0;
}
```

Output:

Output

```
Enter an infix expression (e.g., (2 + 3) * 4): 2-(8(9+2)*1)
Postfix expression: 2892+1*-
Enter values for operands (digits 0-9):
Value for 0: 4
Value for 1: 1
Value for 2: 2
Value for 3: 5
Value for 4: 5
Value for 5: 6
Value for 6: 7
Value for 7: 5
Value for 8: 9
Value for 9: 100
Evaluation result: -93
```

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

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

#define MAX_SIZE 100 // Maximum stack size

// Structure for the stack
struct Stack {
   char items[MAX_SIZE];
   int top;
};

// 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_SIZE - 1;
}

// Function to push an item onto the stack
void push(struct Stack* stack, char item) {
```

```
    if (isFull(stack)) {
        printf("Stack overflow!\n");
        exit(1);
    }
    stack->items[++stack->top] = item;
}

// Function to pop an item from the stack
char pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow!\n");
        exit(1);
    }
    return stack->items[stack->top--];
}

// Function to clean the string (remove non-alphanumeric, convert to lowercase)
void cleanString(const char* input, char* output) {
    int j = 0;
    for (int i = 0; input[i] != '\0'; i++) {
        if (isalnum(input[i])) {
            output[j++] = tolower(input[i]);
        }
    }
    output[j] = '\0';
}

// Function to check if the string is a palindrome using a stack
int isPalindrome(const char* input) {
    // Clean the input string
    char cleaned[MAX_SIZE];
```

```
    cleanString(input, cleaned);
    int len = strlen(cleaned);

    if (len == 0) {
        return 1; // Empty string is a palindrome
    }

    // Initialize stack
    struct Stack stack;
    initStack(&stack);

    // Push first half of the string onto the stack
    int mid = len / 2;
    for (int i = 0; i < mid; i++) {
        push(&stack, cleaned[i]);
    }

    // Skip the middle character for odd-length strings
    int start = (len % 2 == 0) ? mid : mid + 1;

    // Compare second half with popped characters
    for (int i = start; i < len; i++) {
        if (pop(&stack) != cleaned[i]) {
            return 0; // Not a palindrome
        }
    }

    return 1; // Palindrome
}

// Main function to test the palindrome check
```

```
int main() {
    // Test cases
    const char* testCases[] = {
        "A man, a plan, a canal: Panama",
        "race a car",
        "Was it a car or a cat I saw?",
        "",
        "hello",
        "12321",
        "Madam, I'm Adam"
    };
    int numTests = sizeof(testCases) / sizeof(testCases[0]);

    for (int i = 0; i < numTests; i++) {
        printf("Input: \"%s\"\n", testCases[i]);
        printf("Output: %s\n", isPalindrome(testCases[i]) ? "Palindrome" : "Not a Palindrome");
        printf("\n");
    }

    return 0;
}
```

Output:

Output

```
Input: "A man, a plan, a canal: Panama"
Output: Palindrome

Input: "race a car"
Output: Not a Palindrome

Input: "Was it a car or a cat I saw?"
Output: Palindrome

Input: ""
Output: Palindrome

Input: "hello"
Output: Not a Palindrome

Input: "12321"
Output: Palindrome

Input: "Madam, I'm Adam"
Output: Palindrome
```

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

```
#include <stdio.h>
#include <string.h>
#define MAX 100

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

// Initialize stack
void init(Stack *s) {
   s->top = -1;
}

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

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

// Push element to stack
void push(Stack *s, char ch) {
   if (isFull(s)) {
      printf("Stack is full.\n");
```

```
        return;
    }
    s->data[++(s->top)] = ch;
}

// Pop element from stack
char pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty.\n");
        return '\0';
    }
    return s->data[(s->top)--];
}

// Check symmetry using stack
void checkSymmetry(char str[]) {
    Stack s;
    init(&s);

    int len = strlen(str);
    int i;

    // Push first half to stack
    for (i = 0; i < len / 2; i++) {
        push(&s, str[i]);
    }

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

```
    // Compare second half with stack
    while (str[i] != '\0') {
        char ch = pop(&s);
        if (ch != str[i]) {
            printf("The string '%s' is NOT symmetric.\n", str);
            return;
        }
        i++;
    }

    printf("The string '%s' is symmetric.\n", str);
}

int main() {
    char str1[MAX];

    printf("Enter a string to check symmetry: ");
    scanf("%s", str1);

    checkSymmetry(str1);

    return 0;
}
```

Output:

```
Output
Enter a string to check symmetry: madam
The string 'madam' is symmetric.
```

```
Output
Enter a string to check symmetry: tomcruise
The string 'tomcruise' is NOT symmetric.
```

Exercise 8: Binary Search Tree

i) Implementing a BST using Linked List.

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

// Structure for a node in the Binary Search Tree
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

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

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

```
        root->left = insertNode(root->left, data);
    } else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }
    // Ignore duplicates for simplicity
    return root;
}

// Function to search for a key in the BST
struct Node* searchNode(struct Node* root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }
    if (key < root->data) {
        return searchNode(root->left, key);
    }
    return searchNode(root->right, key);
}

// Function to find the node with the minimum value in a BST
struct Node* findMin(struct Node* root) {
    while (root && root->left != NULL) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the BST
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) {
        return root;
```

```
    }
    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        // Node with two children: Get the inorder successor (smallest in right subtree)
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

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

```
}

// Main function to test BST operations
int main() {
    struct Node* root = NULL;

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

    // Print the BST using inorder traversal
    printf("Inorder Traversal (Sorted Order): ");
    inorderTraversal(root);
    printf("\n");

    // Test search
    int key = 40;
    struct Node* result = searchNode(root, key);
    if (result) {
        printf("Found key %d in the BST\n", key);
    } else {
        printf("Key %d not found in the BST\n", key);
    }

    // Test deletion
    printf("Deleting key 30...\n");
```

```
    root = deleteNode(root, 30);
    printf("Inorder Traversal after deletion: ");
    inorderTraversal(root);
    printf("\n");

    // Test deletion of a non-existent key
    printf("Deleting key 100...\n");
    root = deleteNode(root, 100);
    printf("Inorder Traversal after deletion: ");
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output:

Output

```
Inorder Traversal (Sorted Order): 20 30 40 50 60 70 80
Found key 40 in the BST
Deleting key 30...
Inorder Traversal after deletion: 20 40 50 60 70 80
Deleting key 100...
Inorder Traversal after deletion: 20 40 50 60 70 80
```

ii) Traversing of BST.

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

// Structure for a node in the Binary Search Tree
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

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

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

```
    } else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }
    return root; // Return unchanged root if data equals root->data (no duplicates)
}

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

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

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

```
}

// Main function to test BST traversals
int main() {
    struct Node* root = NULL;

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

    // Perform traversals
    printf("Inorder Traversal (Sorted Order): ");
    inorderTraversal(root);
    printf("\n");

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

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

    return 0;
}
```

Output:

Output

```
Inorder Traversal (Sorted Order): 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
```

Exercise 9: Hashing

i) Implement a hash table with collision resolution techniques.

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

// Constants
#define TABLE_SIZE 10
#define DELETED -999 // Marker for deleted slots in linear probing

// Structure for a node in the linked list (for chaining)
struct ChainNode {
   int key;
   int value;
   struct ChainNode* next;
};

// Structure for chaining hash table
struct ChainHashTable {
   struct ChainNode** table; // Array of linked list pointers
};

// Structure for open addressing hash table
struct OpenHashTable {
   int* keys; // Array to store keys
   int* values; // Array to store values
   int size; // Current number of elements
};

// --- Chaining Hash Table Functions ---
```

```
// Create a new chain node
struct ChainNode* createChainNode(int key, int value) {
    struct ChainNode* node = (struct ChainNode*)malloc(sizeof(struct ChainNode));
    if (!node) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    node->key = key;
    node->value = value;
    node->next = NULL;
    return node;
}

// Initialize chaining hash table
struct ChainHashTable* createChainHashTable() {
    struct ChainHashTable* ht = (struct ChainHashTable*)malloc(sizeof(struct ChainHashTable));
    if (!ht) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    ht->table = (struct ChainNode**)calloc(TABLE_SIZE, sizeof(struct ChainNode*));
    if (!ht->table) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    return ht;
}

// Hash function
unsigned int hashFunction(int key) {
    return key % TABLE_SIZE; // Simple modulo
```

```
}

// Insert into chaining hash table
void insertChain(struct ChainHashTable* ht, int key, int value) {
    unsigned int index = hashFunction(key);
    struct ChainNode* current = ht->table[index];
    // Check if key exists
    while (current) {
        if (current->key == key) {
            current->value = value;
            printf("Updated key %d with value %d\n", key, value);
            return;
        }
        current = current->next;
    }
    // Insert new node at the front of the list
    struct ChainNode* newNode = createChainNode(key, value);
    newNode->next = ht->table[index];
    ht->table[index] = newNode;
    printf("Inserted key %d with value %d\n", key, value);
}

// Search in chaining hash table
int searchChain(struct ChainHashTable* ht, int key) {
    unsigned int index = hashFunction(key);
    struct ChainNode* current = ht->table[index];
    while (current) {
        if (current->key == key) {
            printf("Found key %d with value %d\n", key, current->value);
            return current->value;
        }
```

```
        current = current->next;
    }
    printf("Key %d not found\n", key);
    return -1;
}

// Delete from chaining hash table
void deleteChain(struct ChainHashTable* ht, int key) {
    unsigned int index = hashFunction(key);
    struct ChainNode* current = ht->table[index];
    struct ChainNode* prev = NULL;
    while (current) {
        if (current->key == key) {
            if (prev) {
                prev->next = current->next;
            } else {
                ht->table[index] = current->next;
            }
            free(current);
            printf("Deleted key %d\n", key);
            return;
        }
        prev = current;
        current = current->next;
    }
    printf("Key %d not found\n", key);
}

// Display chaining hash table
void displayChain(struct ChainHashTable* ht) {
    printf("Chaining Hash Table:\n");
```

```
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Index %d: ", i);
        struct ChainNode* current = ht->table[i];
        while (current) {
            printf("(%d, %d) ", current->key, current->value);
            current = current->next;
        }
        printf("\n");
    }
}

// --- Open Addressing (Linear Probing) Hash Table Functions ---

// Initialize open addressing hash table
struct OpenHashTable* createOpenHashTable() {
    struct OpenHashTable* ht = (struct OpenHashTable*)malloc(sizeof(struct OpenHashTable));
    if (!ht) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    ht->keys = (int*)calloc(TABLE_SIZE, sizeof(int));
    ht->values = (int*)calloc(TABLE_SIZE, sizeof(int));
    if (!ht->keys || !ht->values) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    ht->size = 0;
    // Initialize with -1 to indicate empty slots
    for (int i = 0; i < TABLE_SIZE; i++) {
        ht->keys[i] = -1;
    }
```

```
    return ht;
}

// Linear probing function
int probe(struct OpenHashTable* ht, int key, int i) {
    return (hashFunction(key) + i) % TABLE_SIZE;
}

// Insert into open addressing hash table
void insertOpen(struct OpenHashTable* ht, int key, int value) {
    if (ht->size >= TABLE_SIZE) {
        printf("Hash table is full\n");
        return;
    }
    int i = 0;
    int index = probe(ht, key, i);
    // Search for key or empty/deleted slot
    while (ht->keys[index] != -1 && ht->keys[index] != key && i < TABLE_SIZE) {
        i++;
        index = probe(ht, key, i);
    }
    if (ht->keys[index] == key) {
        ht->values[index] = value;
        printf("Updated key %d with value %d\n", key, value);
    } else {
        ht->keys[index] = key;
        ht->values[index] = value;
        ht->size++;
        printf("Inserted key %d with value %d\n", key, value);
    }
}
```

```
// Search in open addressing hash table
int searchOpen(struct OpenHashTable* ht, int key) {
    int i = 0;
    int index = probe(ht, key, i);
    while (ht->keys[index] != -1 && i < TABLE_SIZE) {
        if (ht->keys[index] == key) {
            printf("Found key %d with value %d\n", key, ht->values[index]);
            return ht->values[index];
        }
        i++;
        index = probe(ht, key, i);
    }
    printf("Key %d not found\n", key);
    return -1;
}

// Delete from open addressing hash table
void deleteOpen(struct OpenHashTable* ht, int key) {
    int i = 0;
    int index = probe(ht, key, i);
    while (ht->keys[index] != -1 && i < TABLE_SIZE) {
        if (ht->keys[index] == key) {
            ht->keys[index] = DELETED; // Mark as deleted
            ht->size--;
            printf("Deleted key %d\n", key);
            return;
        }
        i++;
        index = probe(ht, key, i);
    }
```

```
    printf("Key %d not found\n", key);
}

// Display open addressing hash table
void displayOpen(struct OpenHashTable* ht) {
    printf("Open Addressing Hash Table:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (ht->keys[i] != -1 && ht->keys[i] != DELETED) {
            printf("Index %d: (%d, %d)\n", i, ht->keys[i], ht->values[i]);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}

// Main function to test both hash tables
int main() {
    // Test Chaining Hash Table
    printf("--- Testing Chaining Hash Table ---\n");
    struct ChainHashTable* chainHT = createChainHashTable();
    insertChain(chainHT, 1, 100);
    insertChain(chainHT, 11, 110); // Collides with key 1
    insertChain(chainHT, 2, 200);
    insertChain(chainHT, 1, 150); // Update key 1
    displayChain(chainHT);
    searchChain(chainHT, 11);
    deleteChain(chainHT, 11);
    displayChain(chainHT);

    // Test Open Addressing Hash Table
    printf("\n--- Testing Open Addressing Hash Table (Linear Probing) ---\n");
```

```
    struct OpenHashTable* openHT = createOpenHashTable();
    insertOpen(openHT, 1, 100);
    insertOpen(openHT, 11, 110); // Collides with key 1, probes to next slot
    insertOpen(openHT, 2, 200);
    insertOpen(openHT, 1, 150); // Update key 1
    displayOpen(openHT);
    searchOpen(openHT, 11);
    deleteOpen(openHT, 11);
    displayOpen(openHT);

    return 0;
}
```

Output:

Output

```
--- Testing Chaining Hash Table ---
Inserted key 1 with value 100
Inserted key 11 with value 110
Inserted key 2 with value 200
Updated key 1 with value 150
Chaining Hash Table:
Index 0:
Index 1: (11, 110) (1, 150)
Index 2: (2, 200)
Index 3:
Index 4:
Index 5:
Index 6:
Index 7:
Index 8:
Index 9:
Found key 11 with value 110
Deleted key 11
Chaining Hash Table:
```

```
Chaining Hash Table:
Index 0:
Index 1: (1, 150)
Index 2: (2, 200)
Index 3:
Index 4:
Index 5:
Index 6:
Index 7:
Index 8:
Index 9:

--- Testing Open Addressing Hash Table (Linear Probing) ---
Inserted key 1 with value 100
Inserted key 11 with value 110
Inserted key 2 with value 200
Updated key 1 with value 150
Open Addressing Hash Table:
Index 0: Empty
Index 1: (1, 150)
Index 2: (11, 110)
Index 3: (2, 200)
Index 4: Empty
Index 5: Empty
Index 6: Empty
```

```
Index 7: Empty
Index 8: Empty
Index 9: Empty
Found key 11 with value 110
Deleted key 11
Open Addressing Hash Table:
Index 0: Empty
Index 1: (1, 150)
Index 2: Empty
Index 3: (2, 200)
Index 4: Empty
Index 5: Empty
Index 6: Empty
Index 7: Empty
Index 8: Empty
Index 9: Empty
```

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

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

// Structure for a node in the doubly linked list (LRU order)
struct CacheNode {
  int key;
  int value;
  struct CacheNode* prev;
  struct CacheNode* next;
};

// Structure for a hash table entry (linked list for collision handling)
struct HashNode {
  int key;
  int value;
  struct CacheNode* cachePtr; // Pointer to the corresponding LRU node
  struct HashNode* next;
};

// Structure for the cache
struct Cache {
  int capacity; // Maximum number of items
  int size; // Current number of items
  struct HashNode** hashTable; // Array of linked lists for hashing
  struct CacheNode* lruHead; // Most recently used
  struct CacheNode* lruTail; // Least recently used
};

// Function to create a cache node
```

```
struct CacheNode* createCacheNode(int key, int value) {
    struct CacheNode* node = (struct CacheNode*)malloc(sizeof(struct CacheNode));
    if (!node) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    node->key = key;
    node->value = value;
    node->prev = node->next = NULL;
    return node;
}

// Function to create a hash node
struct HashNode* createHashNode(int key, int value, struct CacheNode* cachePtr) {
    struct HashNode* node = (struct HashNode*)malloc(sizeof(struct HashNode));
    if (!node) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    node->key = key;
    node->value = value;
    node->cachePtr = cachePtr;
    node->next = NULL;
    return node;
}

// Function to initialize the cache
struct Cache* createCache(int capacity) {
    struct Cache* cache = (struct Cache*)malloc(sizeof(struct Cache));
    if (!cache) {
        printf("Memory allocation failed!\n");
```

```
        exit(1);
    }
    cache->capacity = capacity;
    cache->size = 0;
    cache->lruHead = cache->lruTail = NULL;
    // Initialize hash table (assuming keys are non-negative integers)
    cache->hashTable = (struct HashNode**)calloc(capacity, sizeof(struct HashNode*));
    if (!cache->hashTable) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    return cache;
}

// Hash function (simple modulo for demonstration)
unsigned int hashFunction(int key, int capacity) {
    return key % capacity;
}

// Function to move a node to the front of the LRU list (most recently used)
void moveToFront(struct Cache* cache, struct CacheNode* node) {
    if (node == cache->lruHead) {
        return; // Already at front
    }
    // Remove node from current position
    if (node->prev) {
        node->prev->next = node->next;
    }
    if (node->next) {
        node->next->prev = node->prev;
    }
```

```
    if (node == cache->lruTail) {
        cache->lruTail = node->prev;
    }
    // Move to front
    node->next = cache->lruHead;
    node->prev = NULL;
    if (cache->lruHead) {
        cache->lruHead->prev = node;
    }
    cache->lruHead = node;
    if (!cache->lruTail) {
        cache->lruTail = node;
    }
}

// Function to put a key-value pair in the cache
void put(struct Cache* cache, int key, int value) {
    unsigned int index = hashFunction(key, cache->capacity);
    // Check if key already exists
    struct HashNode* current = cache->hashTable[index];
    while (current) {
        if (current->key == key) {
            // Update value and move to front
            current->value = value;
            current->cachePtr->value = value;
            moveToFront(cache, current->cachePtr);
            printf("Updated key %d with value %d\n", key, value);
            return;
        }
        current = current->next;
    }
```

```
// If cache is full, remove the least recently used item
if (cache->size >= cache->capacity) {
   // Remove from LRU list
   struct CacheNode* lruNode = cache->lruTail;
   int lruKey = lruNode->key;
   cache->lruTail = lruNode->prev;
   if (cache->lruTail) {
      cache->lruTail->next = NULL;
   } else {
      cache->lruHead = NULL;
   }
   // Remove from hash table
   index = hashFunction(lruKey, cache->capacity);
   struct HashNode* prev = NULL;
   current = cache->hashTable[index];
   while (current && current->key != lruKey) {
      prev = current;
      current = current->next;
   }
   if (prev) {
      prev->next = current->next;
   } else {
      cache->hashTable[index] = current->next;
   }
   free(current);
   free(lruNode);
   cache->size--;
   printf("Evicted key %d\n", lruKey);
}
// Insert new key-value pair
struct CacheNode* cacheNode = createCacheNode(key, value);
```

```
    moveToFront(cache, cacheNode);
    struct HashNode* hashNode = createHashNode(key, value, cacheNode);
    hashNode->next = cache->hashTable[index];
    cache->hashTable[index] = hashNode;
    cache->size++;
    printf("Inserted key %d with value %d\n", key, value);
}

// Function to get the value for a key
int get(struct Cache* cache, int key) {
    unsigned int index = hashFunction(key, cache->capacity);
    struct HashNode* current = cache->hashTable[index];
    while (current) {
        if (current->key == key) {
            moveToFront(cache, current->cachePtr);
            printf("Retrieved key %d with value %d\n", key, current->value);
            return current->value;
        }
        current = current->next;
    }
    printf("Key %d not found\n", key);
    return -1; // Key not found
}

// Function to display the cache contents and LRU order
void displayCache(struct Cache* cache) {
    printf("Cache Contents (Hash Table):\n");
    for (int i = 0; i < cache->capacity; i++) {
        printf("Index %d: ", i);
        struct HashNode* current = cache->hashTable[i];
        while (current) {
```

```
            printf("(%d, %d) ", current->key, current->value);
            current = current->next;
        }
        printf("\n");
    }
    printf("LRU Order (Most to Least Recent): ");
    struct CacheNode* current = cache->lruHead;
    while (current) {
        printf("(%d, %d) ", current->key, current->value);
        current = current->next;
    }
    printf("\n");
}

// Main function to test the cache
int main() {
    struct Cache* cache = createCache(3); // Cache with capacity 3

    // Test insertions
    put(cache, 1, 100); // Insert (1, 100)
    put(cache, 2, 200); // Insert (2, 200)
    put(cache, 3, 300); // Insert (3, 300)
    displayCache(cache);

    // Test get
    get(cache, 2); // Should move 2 to front
    displayCache(cache);

    // Test update
    put(cache, 1, 150); // Update value of key 1
    displayCache(cache);
```

```
    // Test eviction
    put(cache, 4, 400); // Should evict least recently used (3)
    displayCache(cache);

    // Test get for non-existent key
    get(cache, 3); // Should return -1

    return 0;
}
```

Output:

```
Output
Inserted key 1 with value 100
Inserted key 2 with value 200
Inserted key 3 with value 300
Cache Contents (Hash Table):
Index 0: (3, 300)
Index 1: (1, 100)
Index 2: (2, 200)
LRU Order (Most to Least Recent): (3, 300) (2, 200) (1, 100)
Retrieved key 2 with value 200
Cache Contents (Hash Table):
Index 0: (3, 300)
Index 1: (1, 100)
Index 2: (2, 200)
LRU Order (Most to Least Recent): (2, 200) (3, 300) (1, 100)
Updated key 1 with value 150
Cache Contents (Hash Table):
Index 0: (3, 300)
Index 1: (1, 150)
Index 2: (2, 200)
LRU Order (Most to Least Recent): (1, 150) (2, 200) (3, 300)
Evicted key 3
Inserted key 4 with value 400
Cache Contents (Hash Table):
Index 0: (4, 400)
Index 1: (1, 150)
Index 2: (2, 200)
LRU Order (Most to Least Recent): (4, 400) (1, 150) (2, 200)
Key 3 not found
```