**DELHI TECHNOLOGICAL UNIVERSITY**

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi 110042

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**



**CO327: MACHINE LEARNING**

**LAB FILE**

**Submitted To: Submitted By:**

**Mr. Saurabh Negi ARINDAM SINGH**

**Assistant Professor B.Tech ECE IIIrd Year**

**Department of 2K22/EC/49**

**Computer Science Engineering Section-3, Group-2**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Objective** | **Date** | **Signature** |
| 1 | To study about numpy, pandas and matplotlib libraries in python. |  |  |
| 2 | To perform data preprocessing and data summarization on iris dataset. |  |  |
| 3 | To perform data preprocessing and data visualization on iris dataset. |  |  |
| 4 | To implement k means clustering. |  |  |
| 5 | To implement data classification using KNN. |  |  |
| 6 | To implement decision tree using ID3 algorithm. |  |  |
| 7 | To implement decision tree using CART algorithm. |  |  |
| 8 | To implement decision tree using C4.5 algorithm. |  |  |
| 9 | To implement multi layer neural network. |  |  |

**Experiment 1**

**Objective:**

To study about numpy, pandas and matplotlib libraries in python..

**Theory**

Binary search is an efficient algorithm for finding an element in a sorted array. It works by repeatedly dividing the array in half. Starting from the middle, the algorithm compares the target with the middle element. If they match, the search is successful. If the target is smaller than the middle element, the search continues in the left half, and if larger, it proceeds in the right half. This process repeats until the target is found or the search space is reduced to zero.

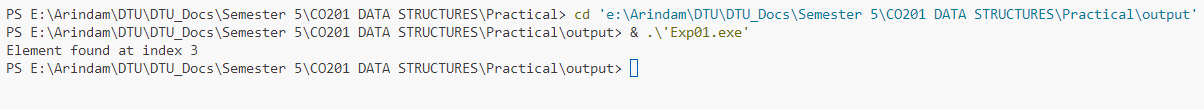
Time Complexity: O(log(n))

Space Complexity: O(1) for the iterative approach.

It is more efficient than linear search, especially for large datasets, due to the reduction in the search space after each comparison.

**Code & OUTPUT**

**Output**

**Result**

As a result of this Experiment, we successfully wrote and executed the program on Binary search in C language.

**Experiment 2**

**Objective:**

To make a program on Quick Sort using Divide and Conquer.

**Theory**

Quick Sort is an efficient sorting algorithm based on the Divide and Conquer paradigm. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays: one with elements less than the pivot and one with elements greater than the pivot. These sub-arrays are then recursively sorted.

Steps of Quick Sort:

1. Divide: Choose a pivot element and partition the array into two parts.
2. Conquer: Recursively apply the same process to the left and right sub-arrays.
3. Combine: Combine the sorted sub-arrays to get the fully sorted array (implicit in recursion).

Time Complexity:

* Best case: O(nlogn)
* Worst case: O(n2) (if pivot is poorly chosen)
* Space Complexity: O(logn)

**Program Code**

#include <stdio.h>

// Function to swap two elements

void swap(int\* a, int\* b) {

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

// Partition function

int partition(int arr[], int low, int high) {

    int pivot = arr[high];  // Taking last element as pivot

    int i = (low - 1);      // Index of smaller element

    for (int j = low; j <= high - 1; j++) {

        // If the current element is smaller than or equal to pivot

        if (arr[j] <= pivot) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

// Quick Sort function

void quickSort(int arr[], int low, int high) {

    if (low < high) {

        // Partition the array

        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

// Function to print an array

void printArray(int arr[], int size) {

    for (int i = 0; i < size; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

// Main function

int main() {

    int arr[] = {10, 7, 8, 9, 1, 5};

    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

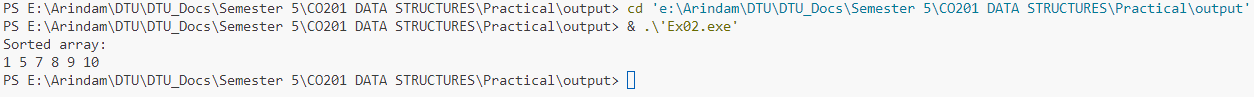
    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to implement Quick Sort using Divide and Conquer in C language.

**Experiment 3**

**Objective:**

To write a program on Quick Sort to sort a list.

**Theory**

Quick Sort is a highly efficient sorting algorithm based on the Divide and Conquer principle. It works by selecting a "pivot" element and partitioning the list such that elements smaller than the pivot are placed to its left and larger elements to its right. The process is then recursively applied to the left and right sublists. This partitioning ensures that after each step, the pivot is in its final sorted position.

Quick Sort is preferred for large datasets due to its average-case time complexity of O(nlogn), though in the worst case (when the pivot is poorly chosen), it degrades to O(n2). However, with good pivot selection strategies, such as choosing the median, Quick Sort performs well in practice.

**Program Code**

#include <stdio.h>

void swap(int\* a, int\* b) {

    int t = \*a; \*a = \*b; \*b = t;

}

int partition(int arr[], int low, int high) {

    int pivot = arr[high], i = (low - 1);

    for (int j = low; j < high; j++) {

        if (arr[j] <= pivot) {

            i++; swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

void quickSort(int arr[], int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(int arr[], int size) {

    for (int i = 0; i < size; i++) printf("%d ", arr[i]);

    printf("\n");

}

int main() {

    int n;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");

    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    quickSort(arr, 0, n - 1);

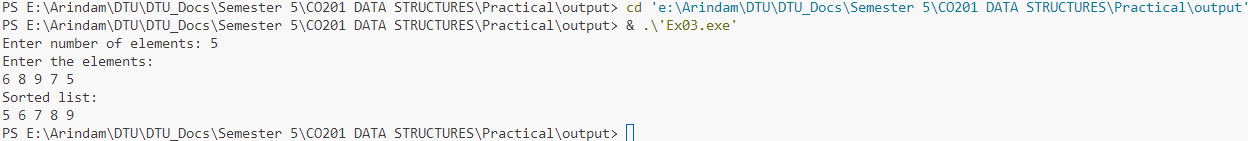
    printf("Sorted list: \n");

    printArray(arr, n);

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to use Quick Sort to sort a list in C language.

**Experiment 4**

**Objective:**

To sort an array of numbers using Heap Sort.

**Theory**

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It works by first building a max heap from the input data, where the largest element is at the root. The algorithm then repeatedly removes the largest element (the root), places it at the end of the array, and re-establishes the heap property on the remaining elements.

Heap Sort operates in two phases:

1. Heapify: Build a max heap from the unsorted array.
2. Extract and Sort: Repeatedly extract the maximum element from the heap, swapping it with the last unsorted element, and heapifying the reduced heap.

Heap Sort has a time complexity of O(n log n) in all cases (best, average, and worst), making it an efficient and stable sorting algorithm for large datasets. Its space complexity is O(1), as it operates in-place.

**Program Code**

#include <stdio.h>

// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {

    int largest = i;    // Initialize largest as root

    int left = 2 \* i + 1;  // Left child

    int right = 2 \* i + 2; // Right child

    // If left child is larger than root

    if (left < n && arr[left] > arr[largest])

        largest = left;

    // If right child is larger than largest so far

    if (right < n && arr[right] > arr[largest])

        largest = right;

    // If largest is not root

    if (largest != i) {

        int temp = arr[i];

        arr[i] = arr[largest];

        arr[largest] = temp;

        // Recursively heapify the affected sub-tree

        heapify(arr, n, largest);

    }

}

// Main function to perform Heap Sort

void heapSort(int arr[], int n) {

    // Build max heap

    for (int i = n / 2 - 1; i >= 0; i--)

        heapify(arr, n, i);

    // Extract elements from heap one by one

    for (int i = n - 1; i > 0; i--) {

        // Move current root to end

        int temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

        // Call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

// Function to print an array

void printArray(int arr[], int size) {

    for (int i = 0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

// Main function

int main() {

    int n;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");

    for (int i = 0; i < n; i++) {

        scanf("%d", &arr[i]);

    }

    heapSort(arr, n);

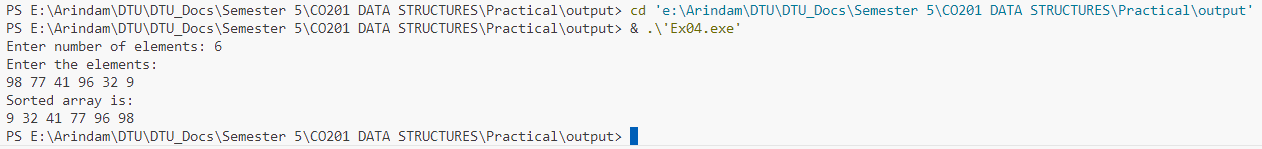
    printf("Sorted array is:\n");

    printArray(arr, n);

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to sort an array of numbers using Heap Sort in C language.

**Experiment 5**

**Objective:**

Convert a valid prefix expression into postfix expression.

**Theory**

To convert a prefix expression (also known as Polish notation) into a postfix expression (Reverse Polish notation), follow a systematic approach using a stack. Prefix expressions have the operator before the operands, while postfix expressions have the operator after the operands. The conversion process ensures that the operands are arranged correctly before their operators are applied.

**Steps:**

1. Traverse the prefix expression from right to left.
2. Push operands onto the stack as you encounter them.
3. When you encounter an operator, pop two operands from the stack, create a string in postfix form (operand1 operand2 operator), and push this result back onto the stack.
4. Repeat the process until the expression is fully traversed.
5. The final element in the stack will be the postfix expression.

**Time Complexity:**

* Time Complexity: O(n) for traversing the expression once.

**Program Code**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

// Stack to store postfix strings

struct Stack {

    int top;

    char arr[MAX][MAX];

} stack;

// Function to push an element onto the stack

void push(char\* str) {

    strcpy(stack.arr[++stack.top], str);

}

// Function to pop an element from the stack

char\* pop() {

    return stack.arr[stack.top--];

}

// Function to check if a character is an operator

int isOperator(char ch) {

    return (ch == '+' || ch == '-' || ch == '\*' || ch == '/');

}

// Function to convert prefix expression to postfix

void prefixToPostfix(char\* prefix) {

    stack.top = -1; // Initialize stack

    int length = strlen(prefix);

    // Traverse the prefix expression from right to left

    for (int i = length - 1; i >= 0; i--) {

        // If the character is an operand, push it onto the stack

        if (isalnum(prefix[i])) {

            char operand[2] = {prefix[i], '\0'};

            push(operand);

        }

        // If the character is an operator

        else if (isOperator(prefix[i])) {

            // Pop two operands from the stack

            char operand1[MAX], operand2[MAX];

            strcpy(operand1, pop());

            strcpy(operand2, pop());

            // Create the postfix expression: operand1 operand2 operator

            char temp[MAX];

            snprintf(temp, sizeof(temp), "%s %s %c", operand1, operand2, prefix[i]);

            // Push the resultant postfix expression back to the stack

            push(temp);

        }

    }

    // The final result in the stack will be the postfix expression

    printf("Postfix expression: %s\n", pop());

}

// Main function

int main() {

    char prefix[MAX];

    printf("Enter a valid prefix expression: ");

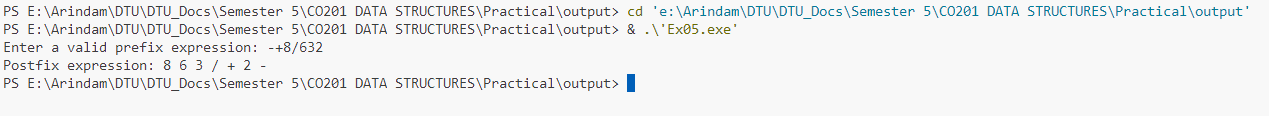
    scanf("%s", prefix);

    prefixToPostfix(prefix);

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to Convert a valid prefix expression into postfix expression in C language.

**Experiment 6**

**Objective:**

Write a C program to evaluate a given postfix expression and its values for the variables.

**Theory**

In postfix expressions (Reverse Polish Notation), operators come after their operands, and there is no need for parentheses to define the order of operations. Evaluation of a postfix expression is done by scanning the expression from left to right and using a stack. When an operand is encountered, it is pushed onto the stack. When an operator is encountered, two operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack. The final result will be at the top of the stack.

Steps for Evaluating a Postfix Expression:

1. Scan the postfix expression from left to right.
2. Push operands onto the stack.
3. When an operator is encountered, pop two operands from the stack, apply the operator, and push the result back onto the stack.
4. At the end, the stack will contain the final result of the postfix expression.

**Program Code**

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Stack type

struct Stack {

    int top;

    unsigned capacity;

    int\* array;

};

// Stack Operations

struct Stack\* createStack(unsigned capacity) {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;

    stack->capacity = capacity;

    stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

    if (!stack->array) return NULL;

    return stack;

}

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

int peek(struct Stack\* stack) {

    return stack->array[stack->top];

}

int pop(struct Stack\* stack) {

    if (!isEmpty(stack))

        return stack->array[stack->top--];

    return -1; // Return -1 if stack is empty

}

void push(struct Stack\* stack, int op) {

    stack->array[++stack->top] = op;

}

// Function to evaluate the value of a given postfix expression

int evaluatePostfix(char\* exp) {

    // Create a stack of capacity equal to expression size

    struct Stack\* stack = createStack(strlen(exp));

    int i;

    // See if stack was created successfully

    if (!stack)

        return -1;

    // Scan all characters one by one

    for (i = 0; exp[i]; ++i) {

        // If the scanned character is an operand (number here), push it to the stack.

        if (isdigit(exp[i]))

            push(stack, exp[i] - '0'); // Convert char to int

        // If the scanned character is an operator, pop two elements from stack and apply the operator

        else {

            int val1 = pop(stack);

            int val2 = pop(stack);

            switch (exp[i]) {

                case '+':

                    push(stack, val2 + val1);

                    break;

                case '-':

                    push(stack, val2 - val1);

                    break;

                case '\*':

                    push(stack, val2 \* val1);

                    break;

                case '/':

                    push(stack, val2 / val1);

                    break;

            }

        }

    }

    return pop(stack); // Return the final result

}

// Driver code

int main() {

    char exp[100];

    // Input the postfix expression

    printf("Enter a postfix expression: ");

    scanf("%s", exp);

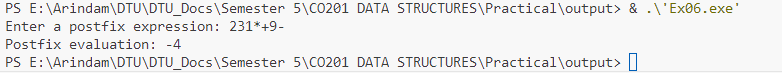
    // Function call

    printf("Postfix evaluation: %d\n", evaluatePostfix(exp));

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to evaluate a given postfix expression and its values for the variables in C language.

**Experiment 7**

**Objective:**

Write a C program to implement a stacks and following operations using an array.

**Theory**

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where the last element added to the stack is the first one to be removed. Stacks are used in various applications such as expression evaluation, backtracking algorithms, and function call management in programming languages.

In this implementation, a stack is created using an array with a defined maximum size. Key operations include **push**, which adds an element to the top of the stack, and **pop**, which removes the top element. The **peek** operation allows users to view the top element without modifying the stack. The **isEmpty** and **isFull** functions check whether the stack is empty or full, respectively. The array-based implementation ensures constant time complexity (O(1)) for push and pop operations, but it has a fixed size, leading to possible overflow (stack full) and underflow (stack empty) conditions.

**Program Code**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum size of the stack

// Stack structure

struct Stack {

    int top;

    int arr[MAX];

};

// Function to create a stack and initialize top

struct Stack\* createStack() {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->top = -1; // Stack is initially empty

    return stack;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

    return stack->top == MAX - 1;

}

// Function to push an element onto the stack

void push(struct Stack\* stack, int value) {

    if (isFull(stack)) {

        printf("Stack Overflow! Cannot push %d onto the stack.\n", value);

        return;

    }

    stack->arr[++stack->top] = value;

    printf("Pushed %d onto the stack.\n", value);

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

    if (isEmpty(stack)) {

        printf("Stack Underflow! Cannot pop from an empty stack.\n");

        return -1; // Return -1 to indicate an error

    }

    return stack->arr[stack->top--];

}

// Function to peek at the top element of the stack

int peek(struct Stack\* stack) {

    if (isEmpty(stack)) {

        printf("Stack is empty! No top element.\n");

        return -1; // Return -1 to indicate an error

    }

    return stack->arr[stack->top];

}

// Function to display the elements of the stack

void display(struct Stack\* stack) {

    if (isEmpty(stack)) {

        printf("Stack is empty! Nothing to display.\n");

        return;

    }

    printf("Stack elements: ");

    for (int i = stack->top; i >= 0; i--) {

        printf("%d ", stack->arr[i]);

    }

    printf("\n");

}

// Driver code

int main() {

    struct Stack\* stack = createStack();

    int choice, value;

    while (1) {

        printf("\nStack Operations:\n");

        printf("1. Push\n");

        printf("2. Pop\n");

        printf("3. Peek\n");

        printf("4. Display\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter value to push: ");

                scanf("%d", &value);

                push(stack, value);

                break;

            case 2:

                value = pop(stack);

                if (value != -1) {

                    printf("Popped %d from the stack.\n", value);

                }

                break;

            case 3:

                value = peek(stack);

                if (value != -1) {

                    printf("Top element is %d\n", value);

                }

                break;

            case 4:

                display(stack);

                break;

            case 5:

                free(stack); // Free allocated memory

                printf("Exiting...\n");

                return 0;

            default:

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

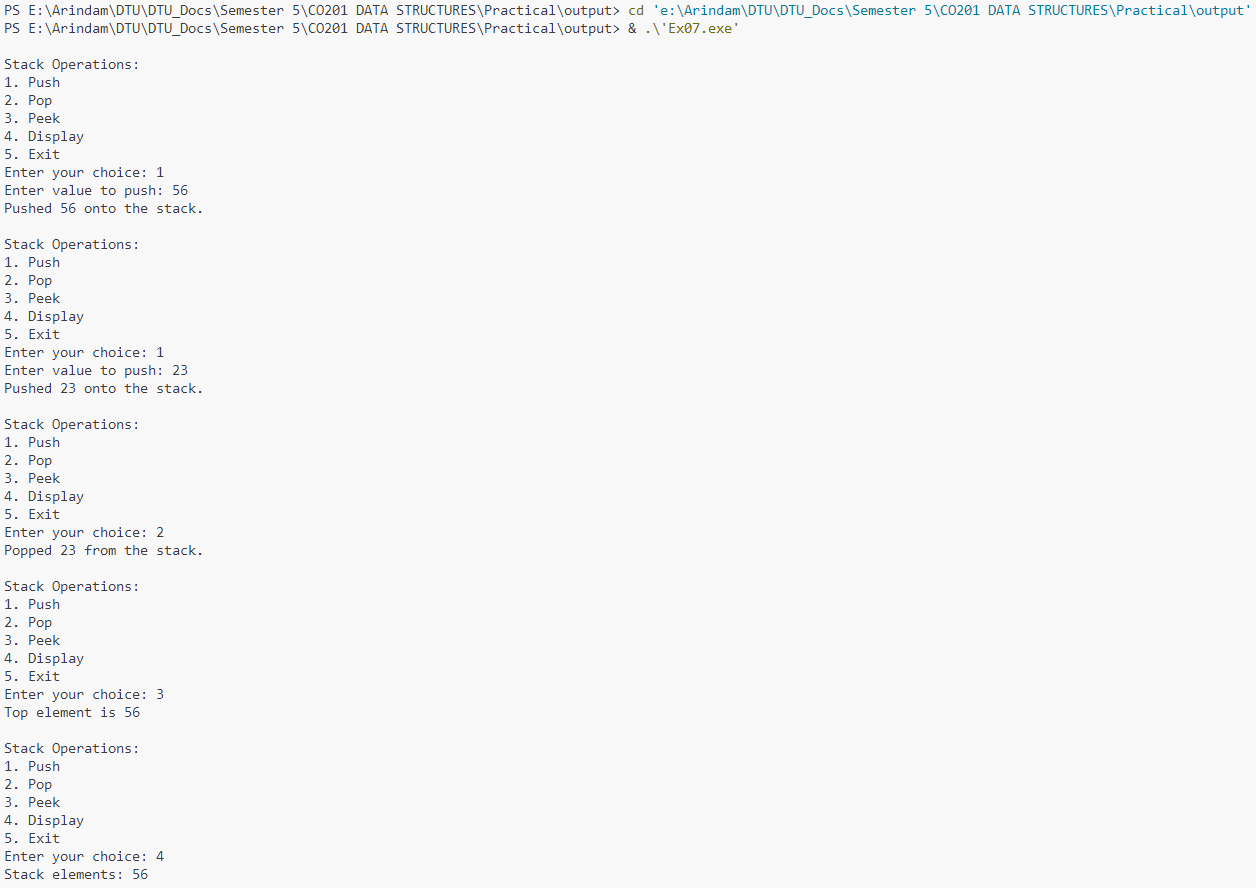
        }

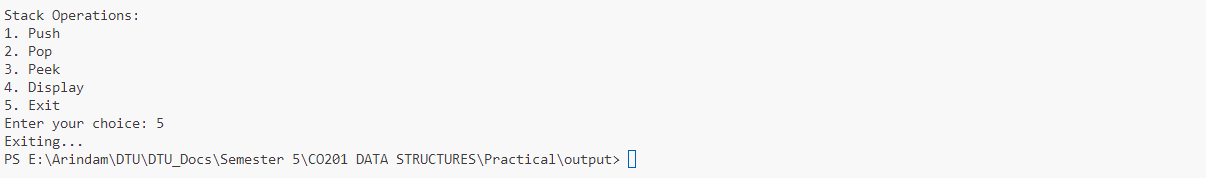
    }

    return 0;

}

**Output**

****

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to implement a stacks and operations using an array in C language.

**Experiment 8**

**Objective:**

To write a C program to implement a Queue using an array and perform basic queue operations**.**

**Theory**

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, where the first element added to the queue is the first one to be removed. This behavior makes queues ideal for scenarios such as task scheduling, breadth-first search in graphs, and managing requests in a server.

In this implementation, a queue is created using an array with a fixed maximum size. Key operations include **enqueue**, which adds an element to the rear of the queue, and **dequeue**, which removes an element from the front. The **peek** operation allows users to view the front element without removing it. Additionally, the **isEmpty** and **isFull** functions check the status of the queue. To efficiently manage space, circular indexing is used to wrap around the array when adding or removing elements. This ensures that the queue can utilize available space effectively without shifting elements, providing O(1) time complexity for enqueue and dequeue operations.

**Program Code**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum size of the queue

// Queue structure

struct Queue {

    int front;

    int rear;

    int size;

    int arr[MAX];

};

// Function to create a queue and initialize its attributes

struct Queue\* createQueue() {

    struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

    queue->front = 0;

    queue->rear = -1;

    queue->size = 0; // Initially, the queue is empty

    return queue;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* queue) {

    return queue->size == 0;

}

// Function to check if the queue is full

int isFull(struct Queue\* queue) {

    return queue->size == MAX;

}

// Function to add an element to the queue

void enqueue(struct Queue\* queue, int value) {

    if (isFull(queue)) {

        printf("Queue Overflow! Cannot enqueue %d\n", value);

        return;

    }

    queue->rear = (queue->rear + 1) % MAX; // Circular increment

    queue->arr[queue->rear] = value;

    queue->size++;

    printf("Enqueued %d into the queue.\n", value);

}

// Function to remove an element from the queue

int dequeue(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue Underflow! Cannot dequeue from an empty queue.\n");

        return -1; // Return -1 to indicate an error

    }

    int value = queue->arr[queue->front];

    queue->front = (queue->front + 1) % MAX; // Circular increment

    queue->size--;

    return value;

}

// Function to get the front element of the queue without removing it

int peek(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty! No front element.\n");

        return -1; // Return -1 to indicate an error

    }

    return queue->arr[queue->front];

}

// Function to display the elements of the queue

void display(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty! Nothing to display.\n");

        return;

    }

    printf("Queue elements: ");

    for (int i = 0; i < queue->size; i++) {

        printf("%d ", queue->arr[(queue->front + i) % MAX]);

    }

    printf("\n");

}

// Driver code

int main() {

    struct Queue\* queue = createQueue();

    int choice, value;

    while (1) {

        printf("\nQueue Operations:\n");

        printf("1. Enqueue\n");

        printf("2. Dequeue\n");

        printf("3. Peek\n");

        printf("4. Display\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter value to enqueue: ");

                scanf("%d", &value);

                enqueue(queue, value);

                break;

            case 2:

                value = dequeue(queue);

                if (value != -1) {

                    printf("Dequeued %d from the queue.\n", value);

                }

                break;

            case 3:

                value = peek(queue);

                if (value != -1) {

                    printf("Front element is %d\n", value);

                }

                break;

            case 4:

                display(queue);

                break;

            case 5:

                free(queue); // Free allocated memory

                printf("Exiting...\n");

                return 0;

            default:

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

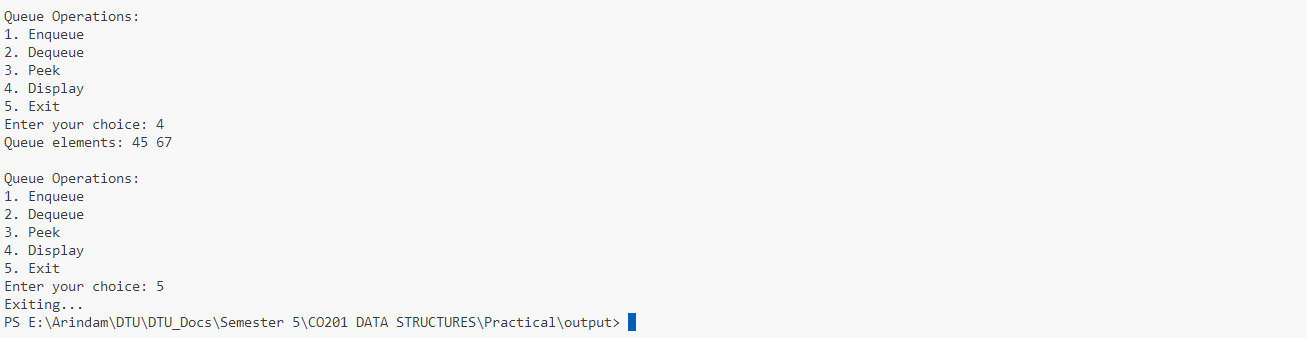
        }

    }

    return 0;

}

**Output**

**** ****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to implement a Queue using an array and perform basic queue operations in C language.

**Experiment 9**

**Objective:**

To write a program to implement a Circular Queue.

**Theory**

A circular queue is a linear data structure that follows the First In, First Out (FIFO) principle while efficiently utilizing space. Unlike a standard queue, where elements are removed from the front and added to the rear, a circular queue connects the rear to the front, allowing it to wrap around when reaching the end of the array. This prevents wasted space when elements are dequeued, enabling continuous usage of the available slots.

In this implementation, a circular queue is created using a fixed-size array. Key operations include **enqueue** (adding elements to the rear), **dequeue** (removing elements from the front), and **peek** (viewing the front element without removal). The **isEmpty** and **isFull** functions determine the status of the queue. The circular nature is managed through modular arithmetic, ensuring that both front and rear indices loop back to the beginning of the array when necessary. This design provides O(1) time complexity for enqueue and dequeue operations while maintaining efficient space utilization.

**Program Code**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum size of the queue

// Circular Queue structure

struct CircularQueue {

    int front;

    int rear;

    int size;

    int arr[MAX];

};

// Function to create a circular queue and initialize its attributes

struct CircularQueue\* createCircularQueue() {

    struct CircularQueue\* queue = (struct CircularQueue\*)malloc(sizeof(struct CircularQueue));

    queue->front = -1; // Initialize front to -1

    queue->rear = -1;  // Initialize rear to -1

    queue->size = 0;   // Initially, the queue is empty

    return queue;

}

// Function to check if the queue is empty

int isEmpty(struct CircularQueue\* queue) {

    return queue->size == 0;

}

// Function to check if the queue is full

int isFull(struct CircularQueue\* queue) {

    return queue->size == MAX;

}

// Function to add an element to the circular queue

void enqueue(struct CircularQueue\* queue, int value) {

    if (isFull(queue)) {

        printf("Queue Overflow! Cannot enqueue %d\n", value);

        return;

    }

    // Update rear

    if (queue->front == -1) {

        queue->front = 0; // Set front to 0 if it's the first element

    }

    queue->rear = (queue->rear + 1) % MAX; // Circular increment

    queue->arr[queue->rear] = value;

    queue->size++;

    printf("Enqueued %d into the queue.\n", value);

}

// Function to remove an element from the circular queue

int dequeue(struct CircularQueue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue Underflow! Cannot dequeue from an empty queue.\n");

        return -1; // Return -1 to indicate an error

    }

    int value = queue->arr[queue->front];

    // Update front

    queue->front = (queue->front + 1) % MAX; // Circular increment

    queue->size--;

    if (isEmpty(queue)) {

        queue->front = -1; // Reset front and rear when queue is empty

        queue->rear = -1;

    }

    return value;

}

// Function to get the front element of the circular queue without removing it

int peek(struct CircularQueue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty! No front element.\n");

        return -1; // Return -1 to indicate an error

    }

    return queue->arr[queue->front];

}

// Function to display the elements of the circular queue

void display(struct CircularQueue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty! Nothing to display.\n");

        return;

    }

    printf("Queue elements: ");

    for (int i = 0; i < queue->size; i++) {

        printf("%d ", queue->arr[(queue->front + i) % MAX]);

    }

    printf("\n");

}

// Driver code

int main() {

    struct CircularQueue\* queue = createCircularQueue();

    int choice, value;

    while (1) {

        printf("\nCircular Queue Operations:\n");

        printf("1. Enqueue\n");

        printf("2. Dequeue\n");

        printf("3. Peek\n");

        printf("4. Display\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter value to enqueue: ");

                scanf("%d", &value);

                enqueue(queue, value);

                break;

            case 2:

                value = dequeue(queue);

                if (value != -1) {

                    printf("Dequeued %d from the queue.\n", value);

                }

                break;

            case 3:

                value = peek(queue);

                if (value != -1) {

                    printf("Front element is %d\n", value);

                }

                break;

            case 4:

                display(queue);

                break;

            case 5:

                free(queue); // Free allocated memory

                printf("Exiting...\n");

                return 0;

            default:

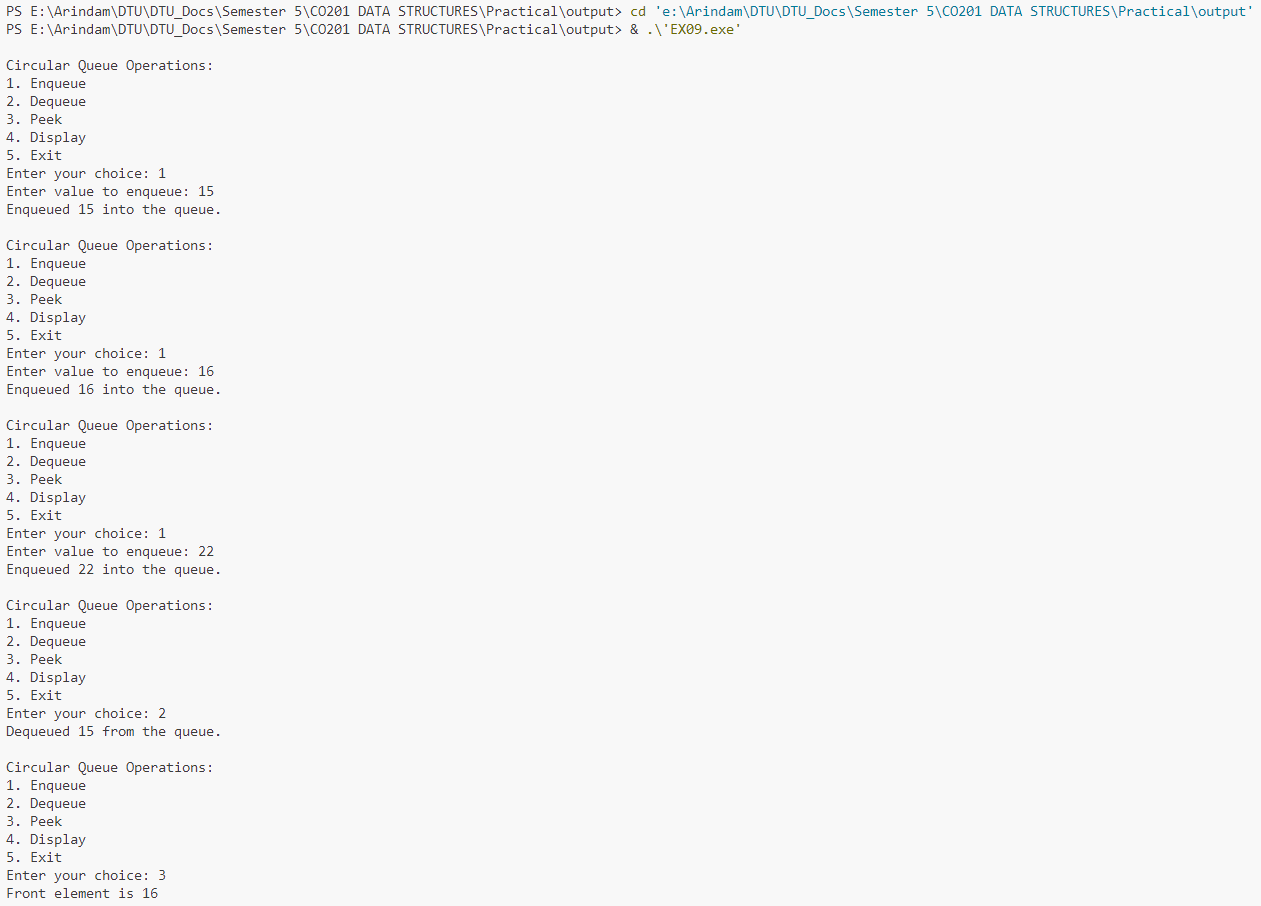
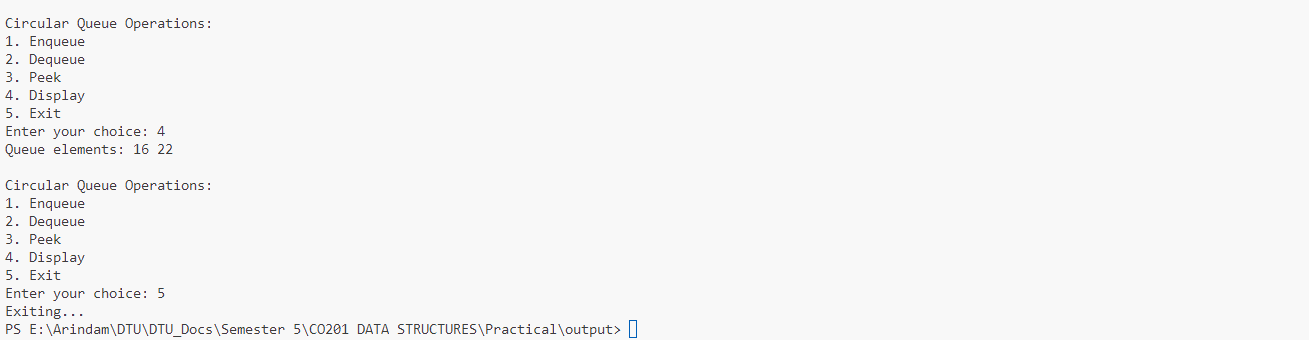
                printf("Invalid choice! Please try again.\n");

        }

    }

    return 0;

}

**Output ** ****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to implement a Circular Queue in C language.

**Experiment 10**

**Objective:**

To write a program to perform insertion and deletion in a linked list.

**Theory**

A linked list is a dynamic data structure composed of nodes, where each node contains data and a pointer to the next node. Unlike arrays, linked lists allow for efficient insertion and deletion operations since they do not require contiguous memory allocation. This characteristic makes linked lists ideal for scenarios where the size of the data set is unknown or changes frequently.

Insertion in a linked list can occur at various positions, such as the beginning, the end, or a specified location. When inserting, new nodes are created and linked appropriately, adjusting the pointers of adjacent nodes. Deletion involves finding a node with a specified value, unlinking it from the list, and freeing the associated memory.

These operations, while simple, require careful management of pointers to avoid memory leaks and ensure data integrity. The dynamic nature of linked lists provides flexibility but also demands proper handling of memory allocation and deallocation.

**Program Code**

#include <stdio.h>

#include <stdlib.h>

// Node structure

struct Node {

    int data;

    struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int value) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = NULL;

    return newNode;

}

// Function to insert a node at the beginning

void insertAtBeginning(struct Node\*\* head, int value) {

    struct Node\* newNode = createNode(value);

    newNode->next = \*head;

    \*head = newNode;

    printf("Inserted %d at the beginning.\n", value);

}

// Function to insert a node at the end

void insertAtEnd(struct Node\*\* head, int value) {

    struct Node\* newNode = createNode(value);

    if (\*head == NULL) {

        \*head = newNode;

        printf("Inserted %d at the end.\n", value);

        return;

    }

    struct Node\* temp = \*head;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = newNode;

    printf("Inserted %d at the end.\n", value);

}

// Function to delete a node by value

void deleteNode(struct Node\*\* head, int value) {

    struct Node\* temp = \*head;

    struct Node\* prev = NULL;

    // If head node itself holds the key

    if (temp != NULL && temp->data == value) {

        \*head = temp->next; // Change head

        free(temp); // Free old head

        printf("Deleted %d from the list.\n", value);

        return;

    }

    // Search for the key to be deleted

    while (temp != NULL && temp->data != value) {

        prev = temp;

        temp = temp->next;

    }

    // If key was not present in linked list

    if (temp == NULL) {

        printf("%d not found in the list.\n", value);

        return;

    }

    // Unlink the node from linked list

    prev->next = temp->next;

    free(temp); // Free memory

    printf("Deleted %d from the list.\n", value);

}

// Function to display the linked list

void displayList(struct Node\* node) {

    if (node == NULL) {

        printf("The list is empty.\n");

        return;

    }

    printf("Linked List: ");

    while (node != NULL) {

        printf("%d -> ", node->data);

        node = node->next;

    }

    printf("NULL\n");

}

// Driver code

int main() {

    struct Node\* head = NULL; // Initialize head to NULL

    int choice, value;

    while (1) {

        printf("\nLinked List Operations:\n");

        printf("1. Insert at Beginning\n");

        printf("2. Insert at End\n");

        printf("3. Delete a Node\n");

        printf("4. Display List\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter value to insert at the beginning: ");

                scanf("%d", &value);

                insertAtBeginning(&head, value);

                break;

            case 2:

                printf("Enter value to insert at the end: ");

                scanf("%d", &value);

                insertAtEnd(&head, value);

                break;

            case 3:

                printf("Enter value to delete: ");

                scanf("%d", &value);

                deleteNode(&head, value);

                break;

            case 4:

                displayList(head);

                break;

            case 5:

                printf("Exiting...\n");

                return 0;

            default:

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

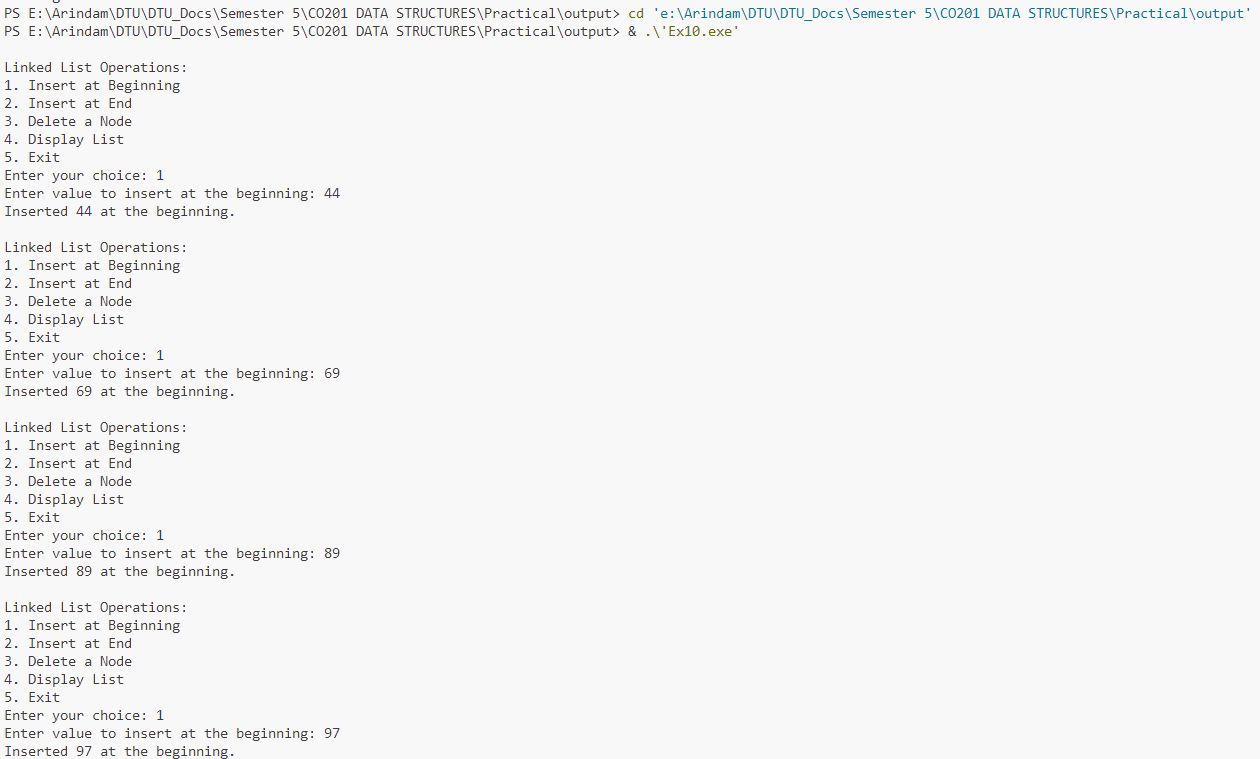
        }

    }

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to perform insertion and deletion in a linked list in C language.

**Experiment 11**

**Objective:**

To implement a queue using a linked list.

**Theory**

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning that the first element added is the first one to be removed. This structure is particularly useful in scenarios such as scheduling tasks, managing requests, and handling asynchronous data.

Implementing a queue using a linked list provides dynamic memory allocation, allowing for efficient use of space as the size of the queue can grow or shrink as needed. Each element in the queue is represented as a node, containing data and a pointer to the next node.

Key operations include **enqueue** (adding an element to the rear), **dequeue** (removing an element from the front), and **peek** (viewing the front element without removal). Since linked lists do not require shifting elements like arrays do, insertion and deletion operations can be performed in constant time, making this implementation efficient for varying workloads.

**Program Code**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the linked list

struct Node {

    int data;

    struct Node\* next;

};

// Queue structure

struct Queue {

    struct Node\* front;

    struct Node\* rear;

};

// Function to create a new node

struct Node\* createNode(int value) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = NULL;

    return newNode;

}

// Function to create a queue and initialize its attributes

struct Queue\* createQueue() {

    struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

    queue->front = NULL;

    queue->rear = NULL;

    return queue;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* queue) {

    return queue->front == NULL;

}

// Function to add an element to the queue

void enqueue(struct Queue\* queue, int value) {

    struct Node\* newNode = createNode(value);

    if (queue->rear == NULL) {

        // Queue is empty

        queue->front = newNode;

        queue->rear = newNode;

    } else {

        // Add new node at the end and update rear

        queue->rear->next = newNode;

        queue->rear = newNode;

    }

    printf("Enqueued %d into the queue.\n", value);

}

// Function to remove an element from the queue

int dequeue(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue Underflow! Cannot dequeue from an empty queue.\n");

        return -1; // Return -1 to indicate an error

    }

    struct Node\* temp = queue->front;

    int value = temp->data;

    queue->front = queue->front->next; // Move front to the next node

    // If the queue becomes empty after dequeuing, update rear

    if (queue->front == NULL) {

        queue->rear = NULL;

    }

    free(temp); // Free memory

    printf("Dequeued %d from the queue.\n", value);

    return value;

}

// Function to get the front element of the queue without removing it

int peek(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty! No front element.\n");

        return -1; // Return -1 to indicate an error

    }

    return queue->front->data;

}

// Function to display the elements of the queue

void displayQueue(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty! Nothing to display.\n");

        return;

    }

    struct Node\* temp = queue->front;

    printf("Queue elements: ");

    while (temp != NULL) {

        printf("%d -> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

// Driver code

int main() {

    struct Queue\* queue = createQueue();

    int choice, value;

    while (1) {

        printf("\nQueue Operations:\n");

        printf("1. Enqueue\n");

        printf("2. Dequeue\n");

        printf("3. Peek\n");

        printf("4. Display\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter value to enqueue: ");

                scanf("%d", &value);

                enqueue(queue, value);

                break;

            case 2:

                value = dequeue(queue);

                if (value != -1) {

                    printf("Dequeued %d from the queue.\n", value);

                }

                break;

            case 3:

                value = peek(queue);

                if (value != -1) {

                    printf("Front element is %d\n", value);

                }

                break;

            case 4:

                displayQueue(queue);

                break;

            case 5:

                printf("Exiting...\n");

                // Free memory (not shown for brevity)

                free(queue);

                return 0;

            default:

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

        }

    }

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to implement a queue using a linked list in C language.

**Experiment 12**

**Objective:**

Program to traverse all the nodes of a Binary Tree using the following traversals.

**Theory**

Binary tree traversal is the process of visiting all the nodes in a binary tree in a specific order. There are three primary methods for traversing a binary tree: **in-order**, **pre-order**, and **post-order**.

* **In-order Traversal**: This method visits the left subtree first, then the root, and finally the right subtree. It is commonly used to retrieve data in a sorted order for binary search trees.
* **Pre-order Traversal**: In this approach, the root node is visited first, followed by the left and then the right subtree. This traversal is useful for creating a copy of the tree or obtaining a prefix expression of an expression tree.
* **Post-order Traversal**: This method visits the left subtree first, then the right subtree, and finally the root node. It is often used for deleting the tree or evaluating postfix expressions in expression trees.

Each traversal method has unique applications and is fundamental for tree data structure operations.

**Program Code**

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

};

// Function to create a new node

struct Node\* createNode(int value) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

// In-order traversal: Left, Root, Right

void inOrderTraversal(struct Node\* root) {

    if (root != NULL) {

        inOrderTraversal(root->left);

        printf("%d ", root->data);

        inOrderTraversal(root->right);

    }

}

// Pre-order traversal: Root, Left, Right

void preOrderTraversal(struct Node\* root) {

    if (root != NULL) {

        printf("%d ", root->data);

        preOrderTraversal(root->left);

        preOrderTraversal(root->right);

    }

}

// Post-order traversal: Left, Right, Root

void postOrderTraversal(struct Node\* root) {

    if (root != NULL) {

        postOrderTraversal(root->left);

        postOrderTraversal(root->right);

        printf("%d ", root->data);

    }

}

// Driver code

int main() {

    // Create a sample binary tree

    struct Node\* root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(3);

    root->left->left = createNode(4);

    root->left->right = createNode(5);

    root->right->left = createNode(6);

    root->right->right = createNode(7);

    printf("In-order Traversal: ");

    inOrderTraversal(root);

    printf("\n");

    printf("Pre-order Traversal: ");

    preOrderTraversal(root);

    printf("\n");

    printf("Post-order Traversal: ");

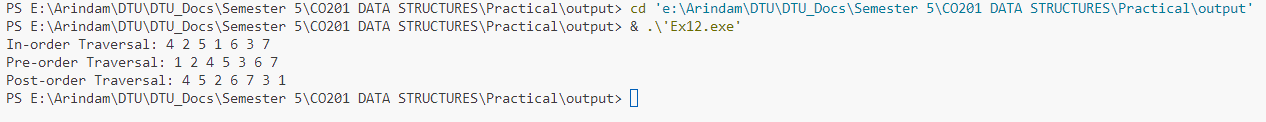
    postOrderTraversal(root);

    printf("\n");

    return 0;

}

**Output**

****

**Result**

As a result of this Experiment, we successfully wrote and executed the program to traverse all the nodes of a Binary Tree using the traversals in C language.