**ARRAY**

1] Write a program to sort the array in ascending order.

Hint: Use Bubble Sort or Python's built-in sort() method.

```
#include <stdio.h>
int main() {
  int a[50], n, i, j, t;
  printf("Enter number of elements: ");
  scanf("%d", &n);
  printf("Enter the elements:\n");
  for(i = 0; i < n; i++)
    scanf("%d", &a[i]);
  // Sorting
  for(i = 0; i < n-1; i++) {
    for(j = 0; j < n-1; j++) {
      if(a[j] > a[j+1]) {
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    }
  }
  printf("Sorted array: ");
  for(i = 0; i < n; i++)
    printf("%d ", a[i]);
  return 0;
}
```

2] Write a program to reverse the elements of a 1D array without using the built-in reverse() function.

Hint: Swap elements from both ends using two-pointer technique.

```
#include <stdio.h>

int main() {

  int n, temp;

  printf("Enter the number of elements: ");

  scanf("%d", &n);

  int arr[n];

  printf("Enter the elements: ");

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

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

  }

  int left = 0;

  int right = n - 1;

  while(left < right) {

    // Swap elements at left and right

    temp = arr[left];

    arr[left] = arr[right];

    arr[right] = temp;

    left++;

    right--;

  }

  printf("Reversed array: ");

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

    printf("%d ", arr[i]);

  }

  return 0;
```

}

3] Write a program to: Create a 2D array (matrix). Take input from the user. Display it in matrix format. Hint:Use nested loops for rows and columns.

```
#include <stdio.h>
int main() {
   int rows, cols;
   printf("Enter number of rows: ");
   scanf("%d", &rows);
   printf("Enter number of columns: ");
   scanf("%d", &cols);
   int matrix[rows][cols];
   printf("Enter matrix elements:\n");
   for(int i = 0; i < rows; i++) {
      for(int j = 0; j < cols; j++) {
         scanf("%d", &matrix[i][j]);
      }
   }
   printf("Matrix:\n");
   for(int i = 0; i < rows; i++) {
      for(int j = 0; j < cols; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

4]Write a program to perform addition of two matrices of the same order.

Hint: C[i][j] = A[i][j] + B[i][j]

```
#include <stdio.h>

int main() {

    int rows, cols;

    printf("Enter number of rows and columns: ");

    scanf("%d %d", &rows, &cols);

    int A[rows][cols], B[rows][cols], C[rows][cols];

    printf("Enter elements of matrix A:\n");

    for(int i = 0; i < rows; i++) {

        for(int j = 0; j < cols; j++) {

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

        }

    }

    printf("Enter elements of matrix B:\n");

    for(int i = 0; i < rows; i++) {

        for(int j = 0; j < cols; j++) {

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

        }

    }

    for(int i = 0; i < rows; i++) {

        for(int j = 0; j < cols; j++) {

            C[i][j] = A[i][j] + B[i][j];

        }

    }

    printf("Sum of matrices:\n");

    for(int i = 0; i < rows; i++) {

        for(int j = 0; j < cols; j++) {
```

```
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

5].Write a program to find the transpose of a given 2D matrix.

Hint: Swap matrix[i][j] with matrix[j][i].

```
#include <stdio.h>
int main() {
    int rows, cols;
    printf("Enter number of rows and columns: ");
    scanf("%d %d", &rows, &cols);
    int matrix[rows][cols];
    printf("Enter matrix elements:\n");
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
    printf("Transpose of the matrix:\n");
    for(int j = 0; j < cols; j++) {
        for(int i = 0; i < rows; i++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
```

```
    return 0;
}
```

**LINKED LIST**

6] Create a singly linked list from input and support push front, push back, delete value, print. Operations to Practice: Node creation, head/tail management, traversal, delete by value.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void pushFront(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;
}
void pushBack(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;
```

```
    temp->next = newNode;
}
void deleteValue(struct Node** head, int value) {
    struct Node* temp = *head;
    struct Node* prev = NULL;
    if (temp != NULL && temp->data == value) {
        *head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) return;
    prev->next = temp->next;
    free(temp);
}
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* head = NULL;
    pushBack(&head, 10);
```

```
    pushBack(&head, 20);
    pushFront(&head, 5);
    printList(head);
    deleteValue(&head, 10);
    printList(head);
    pushBack(&head, 30);
    printList(head);
    return 0;
}
```

7].Given a list and integer k, reverse nodes in groups of k. Last group ¡ k remains as-is. Operations to Practice:Pointer reversal, group detection, reconnect sublists. Target: O(n) time, O(1) extra.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void pushBack(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL)
```

```
        temp = temp->next;
    temp->next = newNode;
}
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
struct Node* reverseKGroup(struct Node* head, int k) {
    struct Node* prev = NULL;
    struct Node* curr = head;
    struct Node* next = NULL;
    int count = 0;
    struct Node* temp = head;
    int nodes = 0;
    while (temp != NULL && nodes < k) {
        temp = temp->next;
        nodes++;
    }
    if (nodes < k) return head;
    count = 0;
    while (curr != NULL && count < k) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
```

```
        count++;
    }
    if (next != NULL)
        head->next = reverseKGroup(next, k);
    return prev;
}
int main() {
    struct Node* head = NULL;
    int n, value, k;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &value);
        pushBack(&head, value);
    }
    printf("Original List: ");
    printList(head);
    printf("Enter k: ");
    scanf("%d", &k);
    head = reverseKGroup(head, k);
    printf("List after reversing in groups of %d: ", k);
    printList(head);
    return 0;
}
```

8] Remove duplicate values from an unsorted singly list; keep first occurrence. Operations to Practice: Hashingor nested scan. Target: O(n) with hash; O(n2 )withoutextraspace.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 1000
struct Node {
   int data;
   struct Node* next;
};
void pushBack(struct Node** head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if (*head == NULL) {
      *head = newNode;
      return;
   }
   struct Node* temp = *head;
   while (temp->next != NULL)
      temp = temp->next;
   temp->next = newNode;
}
void printList(struct Node* head) {
   while (head != NULL) {
      printf("%d -> ", head->data);
      head = head->next;
   }
   printf("NULL\n");
}
```

```c
void removeDuplicatesNested(struct Node* head) {
    struct Node* curr = head;
    while (curr != NULL) {
        struct Node* runner = curr;
        while (runner->next != NULL) {
            if (runner->next->data == curr->data) {
                struct Node* temp = runner->next;
                runner->next = runner->next->next;
                free(temp);
            } else {
                runner = runner->next;
            }
        }
        curr = curr->next;
    }
}
void removeDuplicatesHash(struct Node* head) {
    bool seen[MAX] = {false};
    struct Node* curr = head;
    struct Node* prev = NULL;
    while (curr != NULL) {
        if (seen[curr->data]) {
            prev->next = curr->next;
            free(curr);
            curr = prev->next;
        } else {
            seen[curr->data] = true;
            prev = curr;
```

```
            curr = curr->next;
        }
    }
}
int main() {
    struct Node* head = NULL;
    int n, value, choice;
    printf("How many elements? ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &value);
        pushBack(&head, value);
    }
    printf("Original List: ");
    printList(head);
    printf("Choose method: 1=Nested scan, 2=Hashing: ");
    scanf("%d", &choice);
    if (choice == 1)
        removeDuplicatesNested(head);
    else if (choice == 2)
        removeDuplicatesHash(head);
    else
        printf("Invalid choice!\n");
    printf("List after removing duplicates: ");
    printList(head);
    return 0;
}
```

9]  Detect if the list has a cycle. If yes, remove it and print linear list. Operations to Practice: Floyd's tortoise-hare, meeting point, cycle length or entry node; pointer fix

```
#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

   int data;

   struct Node* next;

} Node;

Node* newNode(int data) {

   Node* n = (Node*)malloc(sizeof(Node));

   n->data = data;

   n->next = NULL;

   return n;

}

void removeCycle(Node* head) {

   Node *slow = head, *fast = head;

   while (fast && fast->next) {

      slow = slow->next;

      fast = fast->next->next;

      if (slow == fast)

         break;

   }

   if (!fast || !fast->next)

      return;

   slow = head;

   while (slow != fast) {

      slow = slow->next;

      fast = fast->next;
```

```
    }
    Node* entry = slow;
    Node* ptr = entry;
    while (ptr->next != entry)
        ptr = ptr->next;
    ptr->next = NULL;
}
void printList(Node* head) {
    while (head) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
int main() {
    int n, val, pos;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Empty list.\n");
        return 0;
    }
    Node *head = NULL, *tail = NULL;
    printf("Enter %d values:\n", n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &val);
        Node* temp = newNode(val);
        if (head == NULL) {
```

```
            head = tail = temp;
        } else {
            tail->next = temp;
            tail = temp;
        }
    }
    printf("Enter position to link last node to (0 for no cycle): ");
    scanf("%d", &pos);
    if (pos > 0 && pos <= n) {
        Node* cycleNode = head;
        for (int i = 1; i < pos; i++)
            cycleNode = cycleNode->next;
        tail->next = cycleNode;
        printf("Cycle created at position %d.\n", pos);
    }
    removeCycle(head);
    printf("Linear list: ");
    printList(head);
    return 0;
}
```

10] .Check whether a singly linked list is a palindrome. Restore list if modified. Operations to Practice: Middle viaslow/fast, reverse second half, compare, optional restore.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int val;
    struct Node* next;
```

```
} Node;
Node* newNode(int val) {
   Node* n = (Node*)malloc(sizeof(Node));
   n->val = val;
   n->next = NULL;
   return n;
}
Node* reverse(Node* head) {
   Node *prev = NULL, *curr = head, *next;
   while (curr) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   return prev;
}
int isPalindrome(Node* head) {
   if (!head || !head->next)
      return 1;
   Node *slow = head, *fast = head;
   while (fast->next && fast->next->next) {
      slow = slow->next;
      fast = fast->next->next;
   }
   Node* second = reverse(slow->next);
   Node *p1 = head, *p2 = second;
   int palindrome = 1;
```

```
  while (p2) {
    if (p1->val != p2->val) {
      palindrome = 0;
      break;
    }
    p1 = p1->next;
    p2 = p2->next;
  }
  slow->next = reverse(second);
  return palindrome;
}
void printList(Node* head) {
  Node* curr = head;
  while (curr) {
    printf("%d ", curr->val);
    curr = curr->next;
  }
  printf("\n");
}
int main() {
  int n, val;
  printf("Enter number of nodes: ");
  scanf("%d", &n);
  if (n <= 0) {
    printf("Empty list.\n");
    return 0;
  }
  Node *head = NULL, *tail = NULL;
```

```
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        Node* node = newNode(val);
        if (!head) {
            head = tail = node;
        } else {
            tail->next = node;
            tail = node;
        }
    }
    printf("Original list: ");
    printList(head);
    if (isPalindrome(head))
        printf("The list IS a palindrome.\n");
    else
        printf("The list is NOT a palindrome.\n");
    printf("List after restore: ");
    printList(head);
    return 0;
}
```

**Stack**

Q1.Given a string of ()[]{}, determine if it is balanced. Operations to Practice: Push on opening, pop on matching closing; check underflow and leftovers.

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

```c
// Stack structure
typedef struct Stack {
    char *arr;
    int top;
    int capacity;
} Stack;

// Create a stack
Stack* createStack(int size) {
    Stack *s = (Stack*)malloc(sizeof(Stack));
    s->arr = (char*)malloc(size * sizeof(char));
    s->top = -1;
    s->capacity = size;
    return s;
}

// Push to stack
void push(Stack *s, char c) {
    if (s->top == s->capacity - 1) {
        printf("Stack overflow!\n");
        return;
    }
    s->arr[++s->top] = c;
}

// Pop from stack
char pop(Stack *s) {
```

```
    if (s->top == -1) {
        return '\0'; // underflow
    }
    return s->arr[s->top--];
}

// Peek top element
char peek(Stack *s) {
    if (s->top == -1) return '\0';
    return s->arr[s->top];
}

// Check if opening and closing match
int isMatching(char open, char close) {
    if (open == '(' && close == ')') return 1;
    if (open == '[' && close == ']') return 1;
    if (open == '{' && close == '}') return 1;
    return 0;
}

// Check if string is balanced
int isBalanced(char *str) {
    int n = strlen(str);
    Stack *s = createStack(n);

    for (int i = 0; i < n; i++) {
        char c = str[i];
```

```
        if (c == '(' || c == '[' || c == '{') {
            push(s, c);  // push opening
        } else if (c == ')' || c == ']' || c == '}') {
            char topChar = pop(s);  // pop and check
            if (topChar == '\0' || !isMatching(topChar, c)) {
                free(s->arr);
                free(s);
                return 0;  // not balanced (underflow or mismatch)
            }
        }
        // ignore other characters
    }

    int balanced = (s->top == -1); // check leftovers
    free(s->arr);
    free(s);
    return balanced;
}

int main() {
    char str[100];

    printf("Enter a string of (), [], {}: ");
    scanf("%s", str);

    if (isBalanced(str))
        printf("The string is balanced.\n");
    else
```

```
        printf("The string is NOT balanced.\n");
    return 0;
}
```

2] For each element, find the next greater element to its right; if none, -1. Operations to Practice: Monotonic stack (indices). Target: O(n).

```
#include <stdio.h>
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[100], nge[100], stack[100];
    int top = -1;
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        nge[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        while (top != -1 && arr[i] > arr[stack[top]]) {
            nge[stack[top--]] = arr[i];
        }
        stack[++top] = i;
    }
    printf("Next Greater Elements:\n");
    for (int i = 0; i < n; i++)
        printf("%d -> %d\n", arr[i], nge[i]);
    return 0;     }
```

Q3.Evaluate a space-separated postfix expression with +, -, *, / and integers. Operations to Practice: Operand push, operator pop-2, compute, push result.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

int main() {

  char expr[100];

  printf("Enter postfix expression (space-separated): ");

  fgets(expr, sizeof(expr), stdin);

  int stack[100], top = -1;

  char *token = strtok(expr, " \n");

  while (token != NULL) {

    // If token is a number (including negative)

    if (isdigit(token[0]) || (token[0] == '-' && isdigit(token[1]))) {

      stack[++top] = atoi(token);

    } else { // Operator

      int b = stack[top--];

      int a = stack[top--];

      if (token[0] == '+') stack[++top] = a + b;

      else if (token[0] == '-') stack[++top] = a - b;

      else if (token[0] == '*') stack[++top] = a * b;

      else if (token[0] == '/') stack[++top] = a / b;

    }

    token = strtok(NULL, " \n");

  }

  printf("Result: %d\n", stack[top]);

  return 0; }
```

Q4.: Convert an infix expression (with parentheses) to postfix. Operations to Practice: Operator stack with precedence/associativity; output queue/string

```
#include <stdio.h>

#include <ctype.h>

#define MAX 100

int precedence(char op) {

    if (op == '+' || op == '-') return 1;

    if (op == '*' || op == '/') return 2;

    return 0;

}

int main() {

    char infix[MAX], postfix[MAX], stack[MAX];

    int top = -1, k = 0;

    printf("Enter infix expression: ");

    scanf("%s", infix);

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

        char c = infix[i];

        if (isalnum(c)) {

            postfix[k++] = c;

        } else if (c == '(') {

            stack[++top] = c;

        } else if (c == ')') {

            while (top != -1 && stack[top] != '(')

                postfix[k++] = stack[top--];

            top--;

        } else {

            while (top != -1 && precedence(stack[top]) >= precedence(c))

                postfix[k++] = stack[top--];
```

```
            stack[++top] = c;
        }
    }
    while (top != -1)
        postfix[k++] = stack[top--];
    postfix[k] = '\0';
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

Q5.Implement a stack with two queues (push/pop/top/size). Operations to Practice: Two-queue method: costly push or costly pop variant.

```
#include <stdio.h>
#define MAX 100
int q1[MAX], q2[MAX];
int front1 = 0, rear1 = 0;
int front2 = 0, rear2 = 0;
void enqueue(int q[], int *rear, int x) {
    q[(*rear)++] = x;
}
int dequeue(int q[], int *front) {
    return q[(*front)++];
}
int isEmpty(int front, int rear) {
    return front == rear;
}
void push(int x) {
    enqueue(q2, &rear2, x);
```

```
    while (!isEmpty(front1, rear1)) {
        enqueue(q2, &rear2, dequeue(q1, &front1));
    }
    int temp[MAX];
    int tempFront = front1, tempRear = rear1;
    for (int i = 0; i < rear1 - front1; i++)
        temp[i] = q1[front1 + i];
    for (int i = 0; i < rear2 - front2; i++)
        q1[i] = q2[front2 + i];
    front1 = 0;
    rear1 = rear2 - front2;
    for (int i = 0; i < tempRear - tempFront; i++)
        q2[i] = temp[i];
    front2 = 0;
    rear2 = 0;
}
int pop() {
    if (isEmpty(front1, rear1)) {
        printf("Stack underflow!\n");
        return -1;
    }
    return dequeue(q1, &front1);
}
int top() {
    if (isEmpty(front1, rear1)) {
        printf("Stack is empty!\n");
        return -1;
    }
```

```
    return q1[front1];
}
int size() {
    return rear1 - front1;
}
int main() {
    push(10);
    push(20);
    push(30);
    printf("Top: %d\n", top());    // 30
    printf("Size: %d\n", size());  // 3
    printf("Pop: %d\n", pop());    // 30
    printf("Pop: %d\n", pop());    // 20
    printf("Top: %d\n", top());    // 10
    printf("Size: %d\n", size());  // 1
    return 0;
}
```

Queue

Q1.Implement a fixed-size circular queue with enqueue, dequeue, front, isFull, isEmpty. Operations to Practice: Array ring buffer; head/tail modulo arithmetic.

```
#include <stdio.h>
#define SIZE 3
int queue[SIZE];
int head = 0;
int tail = 0;  // points to next insertion
int count = 0; // number of elements
int isEmpty() {
```

```
    return count == 0;
}
int isFull() {
    return count == SIZE;
}
int enqueue(int value) {
    if (isFull()) return 0; // fail
    queue[tail] = value;
    tail = (tail + 1) % SIZE;
    count++;
    return 1; // success
}
int dequeue() {
    if (isEmpty()) return 0; // fail
    head = (head + 1) % SIZE;
    count--;
    return 1; // success
}
int front() {
    if (isEmpty()) return -1; // empty
    return queue[head];
}
int rear() {
    if (isEmpty()) return -1;
    return queue[(tail - 1 + SIZE) % SIZE];
}
int main() {
    enqueue(1);
```

```c
    enqueue(2);
    enqueue(3);
    printf("Enqueue 4: %d\n", enqueue(4)); // 0, full
    printf("Front: %d\n", front()); // 1
    printf("Rear: %d\n", rear());   // 3
    dequeue();
    printf("Front after dequeue: %d\n", front()); // 2
    enqueue(4);
    printf("Rear after enqueue 4: %d\n", rear()); // 4
    return 0;
}
```

Q2.Given array and window size k, print max of each window. Operations to Practice: Monotonic deque storing indices. Target: O(n).

```c
#include <stdio.h>
#define MAX 1000
int deque[MAX];  // stores indices
int front = 0, back = -1;
void pushBack(int idx) {
    back++;
    deque[back] = idx;
}
void popBack() {
    if (front <= back) back--;
}
void popFront() {
    if (front <= back) front++;
}
```

```
int getFront() {
    return deque[front];
}
int main() {
    int n, k;
    printf("Enter array size: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter array elements: ");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);
    printf("Enter window size k: ");
    scanf("%d", &k);
    for (int i = 0; i < n; i++) {
        if (front <= back && deque[front] <= i - k) popFront();
        while (front <= back && arr[deque[back]] < arr[i]) popBack();
        pushBack(i);
        if (i >= k - 1) printf("%d ", arr[getFront()]);
    }
    printf("\n");
    return 0;
}
```

Q3.Implement a FIFO queue using two stacks. Operations to Practice: In-stack for enqueue, out-stack for dequeue; transfer lazily.

```
#include <stdio.h>
#define MAX 100
int stack_in[MAX], top_in = -1;
int stack_out[MAX], top_out = -1;
```

```
void push_in(int val) { stack_in[++top_in] = val; }

int pop_in() { return (top_in >= 0) ? stack_in[top_in--] : -1; }

int peek_in() { return (top_in >= 0) ? stack_in[top_in] : -1; }

void push_out(int val) { stack_out[++top_out] = val; }

int pop_out() { return (top_out >= 0) ? stack_out[top_out--] : -1; }

int peek_out() { return (top_out >= 0) ? stack_out[top_out] : -1; }

void transfer() {
  while (top_in >= 0) {
    push_out(pop_in());
  }
}

void enqueue(int val) {
  push_in(val);
}

int dequeue() {
  if (top_out < 0) transfer();  // lazy transfer
  return pop_out();
}

int front() {
  if (top_out < 0) transfer();
  return peek_out();
}

int isEmpty() {
  return top_in < 0 && top_out < 0;
}

int main() {
  enqueue(10);
  enqueue(20);
```

```
    enqueue(30);

    printf("Front: %d\n", front());  // 10

    printf("Dequeue: %d\n", dequeue());  // 10

    printf("Front: %d\n", front());  // 20

    enqueue(40);

    printf("Dequeue: %d\n", dequeue());  // 20

    printf("Dequeue: %d\n", dequeue());  // 30

    printf("Front: %d\n", front());  // 40

    printf("Is queue empty? %s\n", isEmpty() ? "Yes" : "No");

    return 0;

}
```

Q4.Simulate round-robin scheduling: each job has burst time; given quantum q, print completion order and turnaround times. Operations to Practice: Queue rotation, decrement remaining time, track finish time.

```
#include <stdio.h>

#define MAX 100

int main() {

    int n, q;

    int burst[MAX], rem[MAX], comp[MAX];

    int queue[MAX], front = 0, back = -1; // simple queue

    printf("Enter number of processes: ");

    scanf("%d", &n);

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

        printf("Enter burst time for process %d: ", i + 1);

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

        rem[i] = burst[i];

        comp[i] = 0;

        queue[++back] = i; // enqueue all processes initially
```

```
  }
  printf("Enter time quantum: ");
  scanf("%d", &q);
  int time = 0; // current time
  printf("\nCompletion order: ");
  while (front <= back) {
    int i = queue[front++]; // dequeue
    if (rem[i] > q) {
      time += q;
      rem[i] -= q;
      queue[++back] = i; // enqueue back
    } else {
      time += rem[i];
      rem[i] = 0;
      comp[i] = time;
      printf("P%d ", i + 1);
    }
  }
  printf("\n\nTurnaround times:\n");
  for (int i = 0; i < n; i++) {
    printf("P%d: %d\n", i + 1, comp[i]); // arrival time = 0
  }
  return 0;
}
```

Q5.Given documents with priorities, simulate printing: at each step if any doc has higher priority than front, move front to back; else print it. Report when target index prints. Operations to Practice: Queue of (prio5

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

```
#define MAX 100

int main() {

   int n, target;

   int prio[MAX], idx[MAX]; // priority and original indices

   int front = 0, back = -1; // queue indices

   printf("Enter number of documents: ");

   scanf("%d", &n);

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

      printf("Enter priority of document %d: ", i);

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

      idx[++back] = i; // enqueue index

   }

   printf("Enter target document index: ");

   scanf("%d", &target);

   int printed = 0;

   while (front <= back) {

      int cur = idx[front];

      int highest = prio[idx[front]];

      for (int i = front + 1; i <= back; i++) {

         if (prio[idx[i]] > highest)

            highest = prio[idx[i]];

      }

      if (prio[cur] < highest) {

         idx[++back] = idx[front];

         front++;

      } else {

         printed++;

         if (cur == target) {
```

```
            printf("\nTarget document printed at order: %d\n", printed);
            break;
        }
        front++;
    }
  }
  return 0;
}
```

**Trees**

Question 1: Binary Tree Traversals Problem Statement: Create a binary tree and perform the following traversals:1. Inorder Traversal 2. Preorder Traversal 3. Postorder Traversal

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node* left;
  struct Node* right;
};
struct Node* createNode(int data) {
  struct Node* node = (struct Node*)malloc(sizeof(struct Node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return node;
}
void inorder(struct Node* root) {
  if (root == NULL) return;
```

```
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}
void postorder(struct Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    printf("Inorder: ");
    inorder(root);
    printf("\n");
    printf("Preorder: ");
    preorder(root);
    printf("\n");
```

```
    printf("Postorder: ");
    postorder(root);
    printf("\n");
    return 0;
}
```

2: Height of a Binary Tree Problem Statement: Write a program to find the height (or depth) of a binary tree. The height of a tree is the number of edges on the longest path from the root node to a leaf node. Hint: Use recursion to calculate height.

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

// Node structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

// Build tree with user input (-1 means no child)
```

```
struct Node* buildTree() {
    int val;
    printf("Enter node value (-1 for NULL): ");
    scanf("%d", &val);
    if (val == -1) return NULL;

    struct Node* root = createNode(val);
    printf("Enter left child of %d:\n", val);
    root->left = buildTree();
    printf("Enter right child of %d:\n", val);
    root->right = buildTree();
    return root;
}

// Calculate height recursively
int height(struct Node* root) {
    if (root == NULL) return -1; // -1 edges; use 0 if counting nodes
    int leftH = height(root->left);
    int rightH = height(root->right);
    return (leftH > rightH ? leftH : rightH) + 1;
}

int main() {
    printf("Build your binary tree:\n");
    struct Node* root = buildTree();

    int h = height(root);
    printf("\nHeight of the binary tree: %d\n", h);
```

```
    return 0;
}
```

3: Count Leaf and Non-Leaf Nodes Problem Statement: Construct a binary tree and write a program to count the number of leaf nodes and non-leaf nodes.

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

// Node structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

// Count leaf nodes
int countLeaf(struct Node* root) {
    if (root == NULL) return 0;
```

```
    if (root->left == NULL && root->right == NULL) return 1;
    return countLeaf(root->left) + countLeaf(root->right);
}

// Count non-leaf nodes
int countNonLeaf(struct Node* root) {
    if (root == NULL || (root->left == NULL && root->right == NULL)) return 0;
    return 1 + countNonLeaf(root->left) + countNonLeaf(root->right);
}

int main() {
    // Build tree manually in main
    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);

    int leaf = countLeaf(root);
    int nonLeaf = countNonLeaf(root);

    printf("Number of leaf nodes: %d\n", leaf);
    printf("Number of non-leaf nodes: %d\n", nonLeaf);

    return 0;
}
```

4: Expression Tree Construction and Traversals Problem Statement: Construct an Expression Tree for the postfix expression: AB + CD – and display its Inorder, Preorder, and Postorder traversals

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


// Node structure

struct Node {

   char data;

   struct Node* left;

   struct Node* right;

};


// Stack for nodes

struct Stack {

   struct Node* items[100];

   int top;

};


// Stack operations

void push(struct Stack* s, struct Node* node) {

   s->items[++(s->top)] = node;

}


struct Node* pop(struct Stack* s) {

   return s->items[(s->top)--];

}
```

```
// Create a new node
struct Node* createNode(char data) {
  struct Node* node = (struct Node*)malloc(sizeof(struct Node));
  node->data = data;
  node->left = node->right = NULL;
  return node;
}

// Check if operator
int isOperator(char ch) {
  return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

// Build expression tree from postfix
struct Node* buildExpressionTree(char* postfix) {
  struct Stack s;
  s.top = -1;

  for (int i = 0; postfix[i] != '\0'; i++) {
    char ch = postfix[i];
    if (isalnum(ch)) { // Operand
      push(&s, createNode(ch));
    } else if (isOperator(ch)) { // Operator
      struct Node* node = createNode(ch);
      node->right = pop(&s);
      node->left = pop(&s);
      push(&s, node);
```

```
        }
    }
    return pop(&s);
}

// Traversals
void inorder(struct Node* root) {
    if (root == NULL) return;
    if (root->left || root->right) printf("(");
    inorder(root->left);
    printf("%c", root->data);
    inorder(root->right);
    if (root->left || root->right) printf(")");
}

void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%c", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%c", root->data);
}
```

```
int main() {
    char postfix[100];
    printf("Enter postfix expression: ");
    scanf("%s", postfix);

    struct Node* root = buildExpressionTree(postfix);

    printf("Inorder Traversal: ");
    inorder(root);
    printf("\n");

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

    printf("Postorder Traversal: ");
    postorder(root);
    printf("\n");
    return 0;
}
```

5: Level Order Traversal (BreadthFirst Search) Problem Statement: Write a program to perform Level Order Traversal of a binary tree using a queue

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

// Node structure
```

```
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

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

// Queue structure
struct Queue {
   struct Node* nodes[100];
   int front, rear;
};

// Initialize queue
void initQueue(struct Queue* q) {
   q->front = q->rear = -1;
}

// Check if queue is empty
int isEmpty(struct Queue* q) {
```

```
    return q->front == -1 || q->front > q->rear;
}

// Enqueue
void enqueue(struct Queue* q, struct Node* node) {
    if (q->rear == 99) return;
    if (q->front == -1) q->front = 0;
    q->nodes[++(q->rear)] = node;
}

// Dequeue
struct Node* dequeue(struct Queue* q) {
    if (isEmpty(q)) return NULL;
    return q->nodes[(q->front)++];
}

// Level Order Traversal
void levelOrder(struct Node* root) {
    if (!root) return;

    struct Queue q;
    initQueue(&q);
    enqueue(&q, root);

    while (!isEmpty(&q)) {
        struct Node* curr = dequeue(&q);
        printf("%d ", curr->data);
```

```
        if (curr->left) enqueue(&q, curr->left);

        if (curr->right) enqueue(&q, curr->right);

    }

}


int main() {

    // Build tree manually

    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("Level Order Traversal: ");

    levelOrder(root);

    printf("\n");


    return 0;

}
```

**Graphs :**

Q1.Representation of Graph using Adjacency Matrix and List :Write a program to represent a graph using: 1. Adjacency Matrix 2. Adjacency List Display both representations for the given graph

```
#include <stdio.h>

#include <stdlib.h>
```

```c
// Node structure for adjacency list
struct Node {
    int vertex;
    struct Node* next;
};

// Create a new adjacency list node
struct Node* createNode(int v) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->vertex = v;
    node->next = NULL;
    return node;
}

// Display adjacency matrix
void displayMatrix(int n, int matrix[n][n]) {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%d ", matrix[i][j]);
        printf("\n");
    }
}

// Display adjacency list
void displayList(int n, struct Node* list[]) {
    printf("Adjacency List:\n");
    for (int i = 0; i < n; i++) {
```

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

int main() {
    int n, e;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);

    int matrix[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            matrix[i][j] = 0;

    struct Node* list[n];
    for (int i = 0; i < n; i++)
        list[i] = NULL;

    printf("Enter edges (u v):\n");
    for (int i = 0; i < e; i++) {
```

```
        int u, v;
        scanf("%d %d", &u, &v);
        // adjacency matrix
        matrix[u][v] = 1;
        matrix[v][u] = 1; // For undirected graph

        // adjacency list
        struct Node* node = createNode(v);
        node->next = list[u];
        list[u] = node;

        struct Node* node2 = createNode(u); // undirected
        node2->next = list[v];
        list[v] = node2;
    }

    displayMatrix(n, matrix);
    displayList(n, list);

    return 0;
}
```

2: Depth First Search (DFS) Traversal Problem Statement: Implement Depth First Search (DFS) traversal of a graph using recursion.

```
#include <stdio.h>

#define MAX 100
```

```
int visited[MAX];

// DFS function using recursion
void DFS(int n, int graph[n][n], int v) {
   visited[v] = 1;
   printf("%d ", v);

   for (int i = 0; i < n; i++) {
      if (graph[v][i] == 1 && !visited[i]) {
         DFS(n, graph, i);
      }
   }
}

int main() {
   int n, e;
   printf("Enter number of vertices: ");
   scanf("%d", &n);
   printf("Enter number of edges: ");
   scanf("%d", &e);

   int graph[n][n];
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         graph[i][j] = 0;

   printf("Enter edges (u v):\n");
```

```c
    for (int i = 0; i < e; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        graph[u][v] = 1;
        graph[v][u] = 1; // For undirected graph
    }

    // Initialize visited array
    for (int i = 0; i < n; i++)
        visited[i] = 0;

    printf("DFS Traversal starting from vertex 0: ");
    DFS(n, graph, 0);
    printf("\n");

    return 0;
}
```

3: Breadth First Search (BFS) Traversal Problem Statement: Implement Breadth First Search (BFS) traversal of a graph using a queue.

```c
#include <stdio.h>

#define MAX 100

int queue[MAX], front = 0, rear = 0;
int visited[MAX];

// Enqueue
```

```
void enqueue(int v) {
    queue[rear++] = v;
}

// Dequeue
int dequeue() {
    return queue[front++];
}

// Check if queue is empty
int isEmpty() {
    return front == rear;
}

// BFS function
void BFS(int n, int graph[n][n], int start) {
    visited[start] = 1;
    enqueue(start);

    while (!isEmpty()) {
        int v = dequeue();
        printf("%d ", v);

        for (int i = 0; i < n; i++) {
            if (graph[v][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(i);
            }
```

```
        }
    }
}

int main() {
    int n, e;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);

    int graph[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            graph[i][j] = 0;

    printf("Enter edges (u v):\n");
    for (int i = 0; i < e; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        graph[u][v] = 1;
        graph[v][u] = 1; // for undirected graph
    }

    for (int i = 0; i < n; i++) visited[i] = 0;

    printf("BFS Traversal starting from vertex 0: ");
    BFS(n, graph, 0);
```

```
    printf("\n");

    return 0;
}
```

4: Minimum Spanning Tree using Kruskal’s Algorithm Problem Statement: Write a program to find the

Minimum Spanning Tree (MST) of a connected weighted graph using Kruskal’s Algorithm.

```
#include <stdio.h>

#define MAX 100

struct Edge {
    int u, v, weight;
};

int parent[MAX];

// Find parent of a vertex (for union-find)
int find(int i) {
    if (parent[i] == i) return i;
    return find(parent[i]);
}

// Union two sets
void unionSets(int x, int y) {
    int a = find(x);
    int b = find(y);
```

```
    parent[a] = b;
}

// Simple sort edges by weight (bubble sort for simplicity)
void sortEdges(struct Edge edges[], int e) {
    for (int i = 0; i < e-1; i++) {
        for (int j = 0; j < e-i-1; j++) {
            if (edges[j].weight > edges[j+1].weight) {
                struct Edge temp = edges[j];
                edges[j] = edges[j+1];
                edges[j+1] = temp;
            }
        }
    }
}

int main() {
    int n, e;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);

    struct Edge edges[e];
    printf("Enter edges (u v weight):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);
    }
```

```c
    // Initialize parent array
    for (int i = 0; i < n; i++) parent[i] = i;

    // Sort edges by weight
    sortEdges(edges, e);

    printf("Edges in MST:\n");
    int mst_weight = 0;

    for (int i = 0; i < e; i++) {
        int u = edges[i].u;
        int v = edges[i].v;
        int w = edges[i].weight;

        if (find(u) != find(v)) {
            printf("%d - %d : %d\n", u, v, w);
            mst_weight += w;
            unionSets(u, v);
        }
    }

    printf("Total weight of MST: %d\n", mst_weight);

    return 0;
}
```

5: Shortest Path using Dijkstra's Algorithm Problem Statement: Write a program to find the shortest path

from a given source vertex to all other vertices using Dijkstra's Algorithm.

```c
#include <stdio.h>

#define MAX 100

#define INF 100000


int main() {

   int n, src;

   printf("Enter number of vertices: ");

   scanf("%d", &n);


   int graph[n][n];

   printf("Enter adjacency matrix (0 if no edge):\n");

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

      for (int j = 0; j < n; j++)

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


   printf("Enter source vertex: ");

   scanf("%d", &src);


   int dist[n], visited[n];

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

      dist[i] = INF;

      visited[i] = 0;

   }

   dist[src] = 0;
```

```
    for (int count = 0; count < n; count++) {
        int min = INF, u = -1;
        for (int i = 0; i < n; i++)
            if (!visited[i] && dist[i] < min) {
                min = dist[i];
                u = i;
            }

        if (u == -1) break; // all remaining vertices are unreachable
        visited[u] = 1;

        for (int v = 0; v < n; v++)
            if (graph[u][v] != 0 && !visited[v] && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    printf("Vertex\tDistance from Source\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\n", i, dist[i]);

    return 0;
}
```