Q. Write a program to create a Binary Search Tree (BST) and implement the following operations:

- **Insert nodes into the BST**
- **Traverse the BST using:**
  - **Inorder traversal**
  - **Preorder traversal**
  - **Postorder traversal**
- **Count the total number of nodes in the BST**
- **Count the number of leaf nodes**
- **Count the number of nodes with only one child**
- **Count the number of nodes with only a left child**

Solution:

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

// Define a node structure
struct Node {
    int key;
    struct Node *left, *right;
};

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

// Function to insert a node in the BST
struct Node* insert(struct Node* node, int key) {
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    return node;
}

// Inorder traversal
void inorder(struct Node* root) {
```

```
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Preorder traversal
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder traversal
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->key);
    }
}

// Function to count total number of nodes
void countNodes(struct Node* root, int* count) {
    if (root != NULL) {
        (*count)++;
        countNodes(root->left, count);
        countNodes(root->right, count);
    }
}

// Function to count total number of leaf nodes
void countLeafNodes(struct Node* root, int* leafCount) {
    if (root != NULL) {
        if (root->left == NULL && root->right == NULL) {
            (*leafCount)++;
        }
        countLeafNodes(root->left, leafCount);
        countLeafNodes(root->right, leafCount);
    }
}

// Function to count nodes with only one child
void countSingleChildNodes(struct Node* root, int* count) {
    if (root != NULL) {
        if ((root->left == NULL && root->right != NULL) ||
            (root->left != NULL && root->right == NULL)) {
            (*count)++;
```

```
        }
        countSingleChildNodes(root->left, count);
        countSingleChildNodes(root->right, count);
    }
}

// Function to count nodes with only a left child
void countLeftChildOnly(struct Node* root, int* count) {
    if (root != NULL) {
        if (root->left != NULL && root->right == NULL) {
            (*count)++;
        }
        countLeftChildOnly(root->left, count);
        countLeftChildOnly(root->right, count);
    }
}

int main() {
    struct Node* root = NULL;
    int choice, key;
    int count, leafCount, singleChildCount, leftOnlyCount;

    // Initialize with some values
    int elements[] = {50, 30, 20, 40, 70, 60, 80};
    int n = sizeof(elements)/sizeof(elements[0]);
    for (int i = 0; i < n; i++) {
        root = insert(root, elements[i]);
    }

    while (1) {
        printf("\n\nBinary Search Tree Operations:\n");
        printf("1. Insert a node\n");
        printf("2. Inorder Traversal\n");
        printf("3. Preorder Traversal\n");
        printf("4. Postorder Traversal\n");
        printf("5. Count total nodes\n");
        printf("6. Count leaf nodes\n");
        printf("7. Count nodes with one child\n");
        printf("8. Count nodes with only left child\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                printf("Node inserted successfully.\n");
                break;
            case 2:
                printf("Inorder traversal:\n");
```

```
            inorder(root);
            printf("\n");
            break;
        case 3:
            printf("Preorder traversal:\n");
            preorder(root);
            printf("\n");
            break;
        case 4:
            printf("Postorder traversal:\n");
            postorder(root);
            printf("\n");
            break;
        case 5:
            count = 0;
            countNodes(root, &count);
            printf("Total number of nodes: %d\n", count);
            break;
        case 6:
            leafCount = 0;
            countLeafNodes(root, &leafCount);
            printf("Total leaf nodes: %d\n", leafCount);
            break;
        case 7:
            singleChildCount = 0;
            countSingleChildNodes(root, &singleChildCount);
            printf("Nodes with one child: %d\n", singleChildCount);
            break;
        case 8:
            leftOnlyCount = 0;
            countLeftChildOnly(root, &leftOnlyCount);
            printf("Nodes with only left child: %d\n", leftOnlyCount);
            break;
        case 9:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }   return 0; }
```

Output:

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit

Enter your choice: 1
Enter key to insert: 10
Node inserted successfully.

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 1
Enter key to insert: 20
Node inserted successfully.

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 1
Enter key to insert: 30
Node inserted successfully.

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 1
Enter key to insert:
50
Node inserted successfully.

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 2
Inorder traversal:
10 20 30 40 50 60 70 80

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 3
Preorder traversal:
50 30 20 10 40 70 60 80

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 4
Postorder traversal:
10 20 40 30 60 80 70 50

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 5
Total number of nodes: 8

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 6
Total leaf nodes: 4

Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes

```
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 7
Nodes with one child: 1


Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 8
Nodes with only left child: 1


Binary Search Tree Operations:
1. Insert a node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Count total nodes
6. Count leaf nodes
7. Count nodes with one child
8. Count nodes with only left child
9. Exit
Enter your choice: 9


=== Code Execution Successful ===
```

Q. Write a C program to get the n number of nodes from the end of a singly linked list.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void insert(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
int getNthFromEnd(struct Node* head, int n) {
    struct Node *main_ptr = head, *ref_ptr = head;
    int count = 0;

    if (head == NULL) {
        printf("List is empty\n");
        return -1;
    }

    while (count < n) {
        if (ref_ptr == NULL) {
            printf("%d is greater than the number of nodes in list\n", n);
            return -1;
        }
        ref_ptr = ref_ptr->next;
        count++;
    }

    while (ref_ptr != NULL) {
        main_ptr = main_ptr->next;
        ref_ptr = ref_ptr->next;
    }

    return main_ptr->data;
}

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

```c
    int choice, data, n;

    while (1) {
        printf("\n1. Insert node\n2. Print list\n3. Get nth node from end\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insert(&head, data);
                break;
            case 2:
                printf("List: ");
                printList(head);
                break;
            case 3:
                printf("Enter n: ");
                scanf("%d", &n);
                int result = getNthFromEnd(head, n);
                if (result != -1)
                    printf("Node %d from end is %d\n", n, result);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output:

```
1. Insert node
2. Print list
3. Get nth node from end
4. Exit
Enter your choice: 1
Enter data to insert: 10

1. Insert node
2. Print list
3. Get nth node from end
4. Exit
Enter your choice: 1
Enter data to insert: 20

1. Insert node
2. Print list
3. Get nth node from end
4. Exit
Enter your choice: 1
Enter data to insert: 30

1. Insert node
2. Print list
3. Get nth node from end
4. Exit
Enter your choice: 2
List: 30 20 10

1. Insert node
2. Print list
3. Get nth node from end
4. Exit
Enter your choice: 3
Enter n: 3
Node 3 from end is 30

1. Insert node
2. Print list
3. Get nth node from end
4. Exit
Enter your choice: 4


=== Code Execution Successful ===
```

Q. Write a C program to create a linked list P, then write a 'C' function named split to create two linked lists Q & R from P So that Q contains all elements in odd positions of P and R contains the remaining elements. Finally print both linked lists i.e. Q and R.

Solution:

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

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

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

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

void split(struct Node* P, struct Node** Q, struct Node** R) {
    struct Node *current = P;
    int position = 1;

    while (current != NULL) {
        if (position % 2 == 1) {
            insert(Q, current->data);
        } else {
            insert(R, current->data);
        }
        current = current->next;
        position++;
    }
}

int main() {
    struct Node *P = NULL, *Q = NULL, *R = NULL;
    int choice, data;
```

```
    while (1) {
            printf("\n1. Insert into P\n2. Print P\n3. Split P into Q and R\n4.
        Print Q and R\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insert(&P, data);
                break;
            case 2:
                printf("List P: ");
                printList(P);
                break;
            case 3:
                split(P, &Q, &R);
                printf("Split completed\n");
                break;
            case 4:
                printf("List Q (odd positions): ");
                printList(Q);
                printf("List R (even positions): ");
                printList(R);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output:

```
1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 1
Enter data to insert: 21

1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 1
Enter data to insert: 31

1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 1
Enter data to insert: 41

1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 2
List P: 41 31 21
```

```
1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 3
Split completed

1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 4
List Q (odd positions): 21 41
List R (even positions): 31

1. Insert into P
2. Print P
3. Split P into Q and R
4. Print Q and R
5. Exit
Enter your choice: 5


=== Code Execution Successful ===
```

Q. Write a program to add of two polynomials of degree n, using linked list

For example $p1=a_nx^n+a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots\ldots.. a^0x^0$

$P2=b_nx^n+b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots\ldots.. b^0x^0$

p1 = first polynomial

p2 = second polynomial

Find out p3= p1+p2

Solution:

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

struct Term {
    int coeff;
    int exp;
    struct Term* next;
};

void insertTerm(struct Term** poly, int coeff, int exp) {
    struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));
    newTerm->coeff = coeff;
    newTerm->exp = exp;
    newTerm->next = *poly;
    *poly = newTerm;
}

void printPoly(struct Term* poly) {
    if (poly == NULL) {
        printf("0\n");
        return;
    }

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

struct Term* addPolynomials(struct Term* p1, struct Term* p2) {
    struct Term* result = NULL;
```

```
 while (p1 != NULL || p2 != NULL) {
        if (p1 == NULL) {
            insertTerm(&result, p2->coeff, p2->exp);
            p2 = p2->next;
        } else if (p2 == NULL) {
            insertTerm(&result, p1->coeff, p1->exp);
            p1 = p1->next;
        } else {
            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 {
                insertTerm(&result, p1->coeff + p2->coeff, p1->exp);
                p1 = p1->next;
                p2 = p2->next;
            }
        }
    }

    return result;
}

int main() {
    struct Term *p1 = NULL, *p2 = NULL, *p3 = NULL;
    int choice, coeff, exp;
    while (1) {
        printf("\n1. Add term to p1\n2. Add term to p2\n3. Print polynomials\n4. 
Add p1 and p2\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter coefficient and exponent for p1: ");
                scanf("%d %d", &coeff, &exp);
                insertTerm(&p1, coeff, exp);
                break;
            case 2:
                printf("Enter coefficient and exponent for p2: ");
                scanf("%d %d", &coeff, &exp);
                insertTerm(&p2, coeff, exp);
                break;
            case 3:
                printf("p1: ");
                printPoly(p1);
                printf("p2: ");
                printPoly(p2);
                break;
```

```
            case 4:
                p3 = addPolynomials(p1, p2);
                printf("p1 + p2: ");
                printPoly(p3);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output:

```
1. Add term to p1
2. Add term to p2
3. Print polynomials
4. Add p1 and p2
5. Exit
Enter your choice: 1
Enter coefficient and exponent for p1: 3 2

1. Add term to p1
2. Add term to p2
3. Print polynomials
4. Add p1 and p2
5. Exit
Enter your choice: 1
Enter coefficient and exponent for p1: 5 3

1. Add term to p1
2. Add term to p2
3. Print polynomials
4. Add p1 and p2
5. Exit
Enter your choice: 2
Enter coefficient and exponent for p2: 2 1

1. Add term to p1
2. Add term to p2
3. Print polynomials
4. Add p1 and p2
5. Exit
Enter your choice: 2
```

Enter coefficient and exponent for p2: 4 3

1. Add term to p1

2. Add term to p2

3. Print polynomials

4. Add p1 and p2

5. Exit

Enter your choice: 3

p1: 5x^3 + 3x^2

p2: 4x^3 + 2x^1

1. Add term to p1

2. Add term to p2

3. Print polynomials

4. Add p1 and p2

5. Exit

Enter your choice: 4

p1 + p2: 2x^1 + 3x^2 + 9x^3

1. Add term to p1

2. Add term to p2

3. Print polynomials

4. Add p1 and p2

5. Exit

Enter your choice: 5

=== Code Execution Successful ===

Q. Write a C program to sort an array using merge sort technique.

Solution:

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

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```c
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

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

int main() {
    int n, choice;

    printf("Enter size of array: ");
    scanf("%d", &n);
    int arr[n];

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

    while (1) {
        printf("\n1. Print array\n2. Sort array using merge sort\n3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Array: ");
                printArray(arr, n);
                break;
            case 2:
                mergeSort(arr, 0, n - 1);
                printf("Array sorted\n");
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output:

```
Enter size of array: 5
Enter 5 elements: 10 200 3 23 1

1. Print array
2. Sort array using merge sort
3. Exit
Enter your choice: 1
Array: 10 200 3 23 1

1. Print array
2. Sort array using merge sort
3. Exit
Enter your choice: 2
Array sorted

1. Print array
2. Sort array using merge sort
3. Exit
Enter your choice: 1
Array: 1 3 10 23 200

1. Print array
2. Sort array using merge sort
3. Exit
Enter your choice: 3


=== Code Execution Successful ===
```

Q. Write a C program using circular linked list do processor scheduling for n processes. Every process is given a CPU time slot of 10 seconds at a time. Find out which process will be complete when and what will be the total waiting time for every process.

Solution:

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

struct Process {
    int pid;
    int burst_time;
    int remaining_time;
    int waiting_time;
    struct Process* next;
};

// Function to add a new process to the circular linked list
void addProcess(struct Process** head, int pid, int burst_time) {
    struct Process* newProcess = (struct Process*)malloc(sizeof(struct
Process));
    newProcess->pid = pid;
    newProcess->burst_time = burst_time;
    newProcess->remaining_time = burst_time;
    newProcess->waiting_time = 0;
    newProcess->next = NULL;

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

// Function to simulate Round Robin scheduling
void scheduleProcesses(struct Process* head, int n) {
    if (head == NULL) return;

    struct Process* current = head;
    int time = 0;
    int completed = 0;
    int time_slot = 0;

    printf("\nTime\tProcess\tRemaining\tWaiting\n");
```

```
 while (completed < n) {
        if (current->remaining_time > 0) {
            time_slot = (current->remaining_time > 10) ? 10 : current-
>remaining_time;

            printf("%d\tP%d\t%d\t\t%d\n", time, current->pid, current-
>remaining_time, current->waiting_time);

            time += time_slot;
            current->remaining_time -= time_slot;

            if (current->remaining_time == 0) {
                completed++;
                printf("Process P%d completed at time %d\n", current->pid, time);
            }

            // Update waiting time for other processes
            struct Process* temp = current->next;
            while (temp != current) {
                if (temp->remaining_time > 0) {
                    temp->waiting_time += time_slot;
                }
                temp = temp->next;
            }
        }

        current = current->next;
    }
}

int main() {
    struct Process* head = NULL;
    int n, burst_time, choice;

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

    for (int i = 1; i <= n; i++) {
        printf("Enter burst time for process P%d: ", i);
        scanf("%d", &burst_time);
        addProcess(&head, i, burst_time);
    }

    while (1) {
        printf("\n1. Schedule processes\n2. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
 switch (choice) {
           case 1:
               scheduleProcesses(head, n);
               break;
           case 2:
               exit(0);
           default:
               printf("Invalid choice\n");
       }
   }

   return 0;
}
```

Output:

```
Enter number of processes: 3
Enter burst time for process P1: 18
Enter burst time for process P2: 5
Enter burst time for process P3: 2

1. Schedule processes
2. Exit
Enter your choice: 1

Time    Process         Remaining     Waiting
0       P1      18              0
10      P2      5               10
Process P2 completed at time 15
15      P3      2               15
Process P3 completed at time 17
17      P1      8               7
Process P1 completed at time 25

1. Schedule processes
2. Exit
Enter your choice: 1

Time    Process         Remaining     Waiting
```

Q. Write a C program to store the details of a weighted graph (Use array of pointers concept).

Solution:

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

struct AdjListNode {
    int dest;
    int weight;
    struct AdjListNode* next;
};

struct AdjList {
    struct AdjListNode* head;
};

struct Graph {
    int V;
    struct AdjList* array;
};

struct AdjListNode* newAdjListNode(int dest, int weight) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));

    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

void addEdge(struct Graph* graph, int src, int dest, int weight) {
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
```

```
    // For undirected graph, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->V; ++v) {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\nAdjacency list of vertex %d\nhead ", v);
        while (pCrawl) {
            printf("-> %d (w:%d)", pCrawl->dest, pCrawl->weight);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

int main() {
    int V, choice, src, dest, weight;

    printf("Enter number of vertices: ");
    scanf("%d", &V);
    struct Graph* graph = createGraph(V);

    while (1) {
        printf("\n1. Add edge\n2. Print graph\n3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter source, destination and weight: ");
                scanf("%d %d %d", &src, &dest, &weight);
                addEdge(graph, src, dest, weight);
                break;
            case 2:
                printGraph(graph);
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output :

```
Enter number of vertices: 3

1. Add edge
2. Print graph
3. Exit
Enter your choice: 1
Enter source, destination and weight: 0 1 10

1. Add edge
2. Print graph
3. Exit
Enter your choice: 1
Enter source, destination and weight: 1 2 5

1. Add edge
2. Print graph
3. Exit
Enter your choice: 2

Adjacency list of vertex 0
head -> 1 (w:10)

Adjacency list of vertex 1
head -> 2 (w:5)-> 0 (w:10)

Adjacency list of vertex 2
head -> 1 (w:5)

1. Add edge
2. Print graph
3. Exit
Enter your choice: 3


=== Code Execution Successful ===
```

Q. Write a C program to implement DFS.

Solution:

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

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

struct Graph {
    int numVertices;
    int* visited;
    struct Node** adjLists;
};

struct Node* createNode(int v) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct Node*));
    graph->visited = malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (for undirected graph)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

```
void DFS(struct Graph* graph, int vertex) {
    struct Node* adjList = graph->adjLists[vertex];
    struct Node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("\nAdjacency list of vertex %d\n", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    int V, choice, src, dest, start;

    printf("Enter number of vertices: ");
    scanf("%d", &V);
    struct Graph* graph = createGraph(V);

    while (1) {
        printf("\n1. Add edge\n2. Print graph\n3. Perform DFS\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter source and destination: ");
                scanf("%d %d", &src, &dest);
                addEdge(graph, src, dest);
                break;
            case 2:
                printGraph(graph);
                break;
```

```
            case 3:
                printf("Enter starting vertex for DFS: ");
                scanf("%d", &start);
                printf("DFS traversal:\n");
                DFS(graph, start);
                // Reset visited array for next DFS
                for (int i = 0; i < graph->numVertices; i++)
                    graph->visited[i] = 0;
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output :

```
Enter number of vertices: 4

1. Add edge
2. Print graph
3. Perform DFS
4. Exit
Enter your choice: 1
Enter source and destination: 0 1

1. Add edge
2. Print graph
3. Perform DFS
4. Exit
Enter your choice: 1
Enter source and destination: 0 2

1. Add edge
2. Print graph
3. Perform DFS
4. Exit
Enter your choice: 1
Enter source and destination: 1 3

1. Add edge
2. Print graph
3. Perform DFS
4. Exit
Enter your choice: 2

Adjacency list of vertex 0
2 -> 1 ->

Adjacency list of vertex 1
3 -> 0 ->

Adjacency list of vertex 2
0 ->

Adjacency list of vertex 3
1 ->

1. Add edge
2. Print graph
3. Perform DFS
4. Exit
Enter your choice: 3
Enter starting vertex for DFS: 0
```

```
DFS traversal:
Visited 0
Visited 2
Visited 1
Visited 3

1. Add edge
2. Print graph
3. Perform DFS
4. Exit
Enter your choice: 4


=== Code Execution Successful ===
```

Q. Write a C program to implement BFS.

Solution:

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

#define MAX 100

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

struct Graph {
    int numVertices;
    int* visited;
    struct Node** adjLists;
};

struct Node* createNode(int v) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct Node*));
    graph->visited = malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (for undirected graph)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

```
void BFS(struct Graph* graph, int startVertex) {
    int queue[MAX];
    int front = -1, rear = -1;

    graph->visited[startVertex] = 1;
    queue[++rear] = startVertex;

    while (front != rear) {
        int currentVertex = queue[++front];
        printf("Visited %d\n", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0) {
                queue[++rear] = adjVertex;
                graph->visited[adjVertex] = 1;
            }
            temp = temp->next;
        }
    }
}

void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("\nAdjacency list of vertex %d\n", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    int V, choice, src, dest, start;

    printf("Enter number of vertices: ");
    scanf("%d", &V);
    struct Graph* graph = createGraph(V);

    while (1) {
        printf("\n1. Add edge\n2. Print graph\n3. Perform BFS\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
  switch (choice) {
          case 1:
              printf("Enter source and destination: ");
              scanf("%d %d", &src, &dest);
              addEdge(graph, src, dest);
              break;
          case 2:
              printGraph(graph);
              break;
          case 3:
              printf("Enter starting vertex for BFS: ");
              scanf("%d", &start);
              printf("BFS traversal:\n");
              BFS(graph, start);
              // Reset visited array for next BFS
              for (int i = 0; i < graph->numVertices; i++)
                  graph->visited[i] = 0;
              break;
          case 4:
              exit(0);
          default:
              printf("Invalid choice\n");
      }
  }

  return 0;
}
```

Output:

```
Enter number of vertices: 5

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 1
Enter source and destination: 0 1

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 1
Enter source and destination: 0 4

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 1
Enter source and destination: 4 1

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 1
Enter source and destination: 1 3

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 1
Enter source and destination: 3 2

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 2

Adjacency list of vertex 0
4 -> 1 ->

Adjacency list of vertex 1
3 -> 4 -> 0 ->
```

```
Adjacency list of vertex 2
3 ->

Adjacency list of vertex 3
2 -> 1 ->

Adjacency list of vertex 4
1 -> 0 ->

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 3
Enter starting vertex for BFS: 0
BFS traversal:
Visited 0
Visited 4
Visited 1
Visited 3
Visited 2

1. Add edge
2. Print graph
3. Perform BFS
4. Exit
Enter your choice: 4


=== Code Execution Successful ===
```

Q. Write a C program to implement Kruskal's algorithm.

Solution:

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

struct Edge {
   int src, dest, weight;
};

struct Graph {
   int V, E;
   struct Edge* edge;
};

struct Graph* createGraph(int V, int E) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->E = E;
   graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
   return graph;
}

struct subset {
   int parent;
   int rank;
};

int find(struct subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y) {
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);

   if (subsets[xroot].rank < subsets[yroot].rank)
      subsets[xroot].parent = yroot;
   else if (subsets[xroot].rank > subsets[yroot].rank)
      subsets[yroot].parent = xroot;
   else {
      subsets[yroot].parent = xroot;
      subsets[xroot].rank++;
   }
}
```

```
int compare(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V];
    int e = 0;
    int i = 0;

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compare);

    struct subset* subsets = (struct subset*)malloc(V * sizeof(struct subset));

    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < graph->E) {
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
}

int main() {
    int V, E, choice;

    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);
    struct Graph* graph = createGraph(V, E);

    for (int i = 0; i < E; i++) {
        printf("Enter source, destination and weight for edge %d: ", i + 1);
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph-
>edge[i].weight);    }
```

```
    while (1) {
        printf("\n1. Find MST using Kruskal's algorithm\n2. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                KruskalMST(graph);
                break;
            case 2:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output:

```
Enter number of vertices: 4
Enter number of edges: 5
Enter source, destination and weight for edge 1: 0 1 10
Enter source, destination and weight for edge 2: 0 2 6
Enter source, destination and weight for edge 3: 0 3 5
Enter source, destination and weight for edge 4: 1 3 15
Enter source, destination and weight for edge 5: 2 3 4

1. Find MST using Kruskal's algorithm
2. Exit
Enter your choice: 1
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10

1. Find MST using Kruskal's algorithm
2. Exit
Enter your choice: 2


=== Code Execution Successful ===
```