Question 1:

Implement Queue using two stacks

Code:

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the stack

struct Node {

int data;

struct Node\* next;

};

// Structure to represent a stack

struct Stack {

struct Node\* top;

};

// Function to create a new node

struct Node\* newNode(int data) {

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

node->data = data;

node->next = NULL;

return node;

}

// Function to initialize a stack

struct Stack\* createStack() {

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

stack->top = NULL;

return stack;

}

// Function to check if a stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

// Function to push an element onto the stack

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

struct Node\* node = newNode(data);

node->next = stack->top;

stack->top = node;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty!\n");

return -1;

}

int data = stack->top->data;

struct Node\* temp = stack->top;

stack->top = stack->top->next;

free(temp);

return data;

}

// Structure to represent a queue

struct Queue {

struct Stack\* stack\_enqueue;

struct Stack\* stack\_dequeue;

};

// Function to create a new queue

struct Queue\* createQueue() {

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

queue->stack\_enqueue = createStack();

queue->stack\_dequeue = createStack();

return queue;

}

// Function to enqueue an element

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

// Push the element onto the enqueue stack

push(queue->stack\_enqueue, data);

}

// Function to dequeue an element

int dequeue(struct Queue\* queue) {

if (isEmpty(queue->stack\_enqueue) && isEmpty(queue->stack\_dequeue)) {

printf("Queue is empty!\n");

return -1;

}

// If dequeue stack is empty, transfer elements from enqueue stack

if (isEmpty(queue->stack\_dequeue)) {

while (!isEmpty(queue->stack\_enqueue)) {

push(queue->stack\_dequeue, pop(queue->stack\_enqueue));

}

}

// Pop from the dequeue stack

return pop(queue->stack\_dequeue);

}

int main() {

struct Queue\* queue = createQueue();

enqueue(queue, 1);

enqueue(queue, 2);

enqueue(queue, 3);

printf("Dequeued: %d\n", dequeue(queue)); // Output: 1

printf("Dequeued: %d\n", dequeue(queue)); // Output: 2

enqueue(queue, 4);

printf("Dequeued: %d\n", dequeue(queue)); // Output: 3

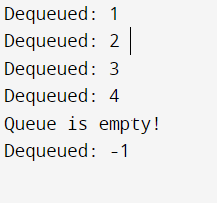
printf("Dequeued: %d\n", dequeue(queue)); // Output: 4

printf("Dequeued: %d\n", dequeue(queue)); // Output: Queue is empty!

return 0;

}

Output



Question 2:

Implement two stacks within single array optimally.

Code:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 10

// Structure to represent the two stacks

struct TwoStacks {

int\* arr;

int top1;

int top2;

};

// Function to create a new instance of the two stacks

struct TwoStacks\* createTwoStacks() {

struct TwoStacks\* twoStacks = (struct TwoStacks\*)malloc(sizeof(struct TwoStacks));

twoStacks->arr = (int\*)malloc(MAX\_SIZE \* sizeof(int));

twoStacks->top1 = -1; // Initial top of stack 1

twoStacks->top2 = MAX\_SIZE; // Initial top of stack 2

return twoStacks;

}

// Function to check if stack 1 is empty

int isEmptyStack1(struct TwoStacks\* twoStacks) {

return twoStacks->top1 == -1;

}

// Function to check if stack 2 is empty

int isEmptyStack2(struct TwoStacks\* twoStacks) {

return twoStacks->top2 == MAX\_SIZE;

}

// Function to check if stack 1 is full

int isFullStack1(struct TwoStacks\* twoStacks) {

return twoStacks->top1 + 1 == twoStacks->top2;

}

// Function to check if stack 2 is full

int isFullStack2(struct TwoStacks\* twoStacks) {

return twoStacks->top2 - 1 == twoStacks->top1;

}

// Function to push an element onto stack 1

void pushStack1(struct TwoStacks\* twoStacks, int data) {

if (isFullStack1(twoStacks)) {

printf("Stack 1 is full!\n");

return;

}

twoStacks->arr[++twoStacks->top1] = data;

}

// Function to push an element onto stack 2

void pushStack2(struct TwoStacks\* twoStacks, int data) {

if (isFullStack2(twoStacks)) {

printf("Stack 2 is full!\n");

return;

}

twoStacks->arr[--twoStacks->top2] = data;

}

// Function to pop an element from stack 1

int popStack1(struct TwoStacks\* twoStacks) {

if (isEmptyStack1(twoStacks)) {

printf("Stack 1 is empty!\n");

return -1;

}

return twoStacks->arr[twoStacks->top1--];

}

// Function to pop an element from stack 2

int popStack2(struct TwoStacks\* twoStacks) {

if (isEmptyStack2(twoStacks)) {

printf("Stack 2 is empty!\n");

return -1;

}

return twoStacks->arr[twoStacks->top2++];

}

int main() {

struct TwoStacks\* twoStacks = createTwoStacks();

pushStack1(twoStacks, 1);

pushStack1(twoStacks, 2);

pushStack1(twoStacks, 3);

pushStack2(twoStacks, 10);

pushStack2(twoStacks, 20);

pushStack2(twoStacks, 30);

printf("Popped from Stack 1: %d\n", popStack1(twoStacks)); // Output: 3

printf("Popped from Stack 2: %d\n", popStack2(twoStacks)); // Output: 30

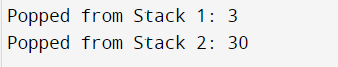
free(twoStacks->arr);

free(twoStacks);

return 0;

}

Output



Question 3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

Write a code to print the matrix in a manner that the output is

1 2 5 9 6 3 4 7 10 13 14 11 8 12 15 16

Code:

#include <stdio.h>

#define ROWS 4

#define COLS 4

// Function to print values from matrix in zigzag manner

void printZigZag(int matrix[ROWS][COLS]) {

// Variables to keep track of current diagonal and direction

int row = 0, col = 0;

int rowEnd = ROWS - 1, colEnd = COLS - 1;

int isUp = 1; // Direction flag: 1 for up, 0 for down

// Iterate through each diagonal

while (row <= rowEnd && col <= colEnd) {

// Print elements of the current diagonal

if (isUp) {

// Move diagonally up

for (; row >= 0 && col <= colEnd; row--, col++) {

printf("%d ", matrix[row][col]);

}

// Adjust row and col after reaching boundary

if (row < 0 && col <= colEnd) {

row = 0;

}

if (col > colEnd) {

col = colEnd;

row += 2;

}

} else {

// Move diagonally down

for (; col >= 0 && row <= rowEnd; col--, row++) {

printf("%d ", matrix[row][col]);

}

// Adjust row and col after reaching boundary

if (col < 0 && row <= rowEnd) {

col = 0;

}

if (row > rowEnd) {

row = rowEnd;

col += 2;

}

}

// Change direction for next diagonal

isUp = !isUp;

}

}

int main() {

int matrix[ROWS][COLS] = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12},

{13, 14, 15, 16}

};

printf("Matrix in zigzag order:\n");

printZigZag(matrix);

return 0;

}

Output



Question 4.

Implement Prims Algorithm

Code:

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define V 5 // Number of vertices in the graph

// Function to find the vertex with the minimum key value,

// from the set of vertices not yet included in the MST

int minKey(int key[], bool mstSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (mstSet[v] == false && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// Function to print the MST using Prim's algorithm

void printMST(int parent[], int graph[V][V]) {

printf("Edge \tWeight\n");

for (int i = 1; i < V; i++)

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to implement Prim's algorithm for finding MST

void primMST(int graph[V][V]) {

int parent[V]; // Array to store constructed MST

int key[V]; // Key values used to pick minimum weight edge in cut

bool mstSet[V]; // To represent set of vertices included in MST

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX;

mstSet[i] = false;

}

// Always include first vertex in MST.

key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the set of vertices not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST set

mstSet[u] = true;

// Update key value and parent index of the adjacent vertices of the picked vertex.

// Consider only those vertices which are not yet included in MST

for (int v = 0; v < V; v++) {

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// Print the constructed MST

printMST(parent, graph);

}

int main() {

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

int graph[V][V] = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0},

};

// Print the MST using Prim's algorithm

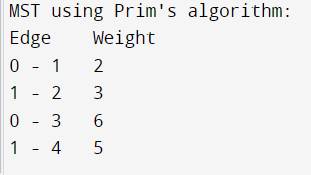
printf("MST using Prim's algorithm:\n");

primMST(graph);

return 0;

}

Output



Question 5

Implement Kruskal’s Algorithm

Code:

#include <stdio.h>

#include <stdlib.h>

// Structure to represent an edge in the graph

struct Edge {

int src, dest, weight;

};

// Structure to represent a subset for union-find

struct Subset {

int parent;

int rank;

};

// A structure to represent a graph

struct Graph {

int V, E;

struct Edge\* edge;

};

// Function prototypes

struct Graph\* createGraph(int V, int E);

void Union(struct Subset subsets[], int x, int y);

int find(struct Subset subsets[], int i);

int myComp(const void\* a, const void\* b);

void KruskalMST(struct Graph\* graph);

int main() {

int V = 5; // Number of vertices

int E = 7; // Number of edges

struct Graph\* graph = createGraph(V, E);

// Add edge 0-1

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 10;

// Add edge 0-2

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 6;

// Add edge 0-3

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 5;

// Add edge 1-3

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 15;

// Add edge 2-3

graph->edge[4].src = 2;

graph->edge[4].dest = 3;

graph->edge[4].weight = 4;

// Add edge 1-4

graph->edge[5].src = 1;

graph->edge[5].dest = 4;

graph->edge[5].weight = 100;

// Add edge 3-4

graph->edge[6].src = 3;

graph->edge[6].dest = 4;

graph->edge[6].weight = 40;

KruskalMST(graph);

return 0;

}

// Creates a graph with V vertices and E edges

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;

}

// A utility function to find the subset of an element i

int find(struct Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of two sets of x and y (uses union by rank)

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++;

}

}

// Compare two edges according to their weights

int myComp(const void\* a, const void\* b) {

struct Edge\* a1 = (struct Edge\*)a;

struct Edge\* b1 = (struct Edge\*)b;

return a1->weight > b1->weight;

}

// The main function to construct MST using Kruskal's algorithm

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]), myComp);

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");

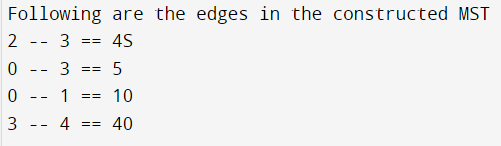
for (i = 0; i < e; ++i)

printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

return;

}

Output



Question 6: Implement Shell sort

Code:

#include <stdio.h>

// Function to perform shell sort

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

// Start with a big gap, then reduce the gap

for (int gap = n / 2; gap > 0; gap /= 2) {

// Do a gapped insertion sort for this gap size.

// The first gap elements arr[0..gap-1] are already in gapped order

// keep adding one more element until the entire array is gap sorted

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

// add arr[i] to the elements that have been gap sorted

// save arr[i] in temp and make a hole at position i

int temp = arr[i];

// shift earlier gap-sorted elements up until the correct location for arr[i] is found

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

arr[j] = arr[j - gap];

// put temp (the original arr[i]) in its correct location

arr[j] = temp;

}

}

}

// Function to print an array

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

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

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

printf("\n");

}

int main() {

int arr[] = {12, 34, 54, 2, 3};

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

printf("Array before sorting: \n");

printArray(arr, n);

shellSort(arr, n);

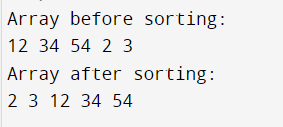
printf("Array after sorting: \n");

printArray(arr, n);

return 0;

}

Output



Question 7

Given the array of integers written the maximum difference of any pair of numbers such that the larger integer in the pair occurs at higher index of array then the smaller integer returns -1 if you can’t find a pair that satisfies this condition.

Code:

#include <stdio.h>

int maxDifference(int arr[], int size) {

if (size < 2)

return -1; // There must be at least two elements to form a pair

int maxDiff = -1;

int minElement = arr[0];

for (int i = 1; i < size; i++) {

if (arr[i] > minElement) {

int diff = arr[i] - minElement;

if (diff > maxDiff)

maxDiff = diff;

} else {

minElement = arr[i];

}

}

return maxDiff;

}

int main() {

int arr[] = {2, 3, 10, 6, 4, 8, 1};

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

int max\_diff = maxDifference(arr, size);

if (max\_diff != -1)

printf("Maximum difference satisfying the condition: %d\n", max\_diff);

else

printf("No pair satisfying the condition found. Maximum difference: -1\n");

return 0;

}

Output



Question 8:

Given n sticks where each stick is of the integer length. A cut operation is performed on stick such that all of them are reduced by length of the smallest stick. Length of the sticks are given, print the number of sticks that are cut in subsequent cut operations.

Code: #include <stdio.h>

#include <stdlib.h>

// Function to find the smallest positive stick length

int find\_min(int arr[], int n) {

int min = \_\_INT\_MAX\_\_;

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

if (arr[i] > 0 && arr[i] < min) {

min = arr[i];

}

}

return min;

}

// Function to perform the cut operations

void cut\_sticks(int arr[], int n) {

int sticks\_cut;

while (1) {

sticks\_cut = 0;

int min = find\_min(arr, n);

// If the smallest stick is zero, all sticks are zero, break the loop

if (min == \_\_INT\_MAX\_\_) {

break;

}

// Perform the cut operation

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

if (arr[i] > 0) {

arr[i] -= min;

sticks\_cut++;

}

}

// Print the number of sticks cut in this operation

printf("%d\n", sticks\_cut);

}

}

int main() {

int n;

// Read the number of sticks

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

scanf("%d", &n);

// Allocate memory for the stick lengths array

int \*arr = (int \*)malloc(n \* sizeof(int));

// Read the length of each stick

printf("Enter the lengths of the sticks: ");

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

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

}

// Perform the cut operations and print the result

cut\_sticks(arr, n);

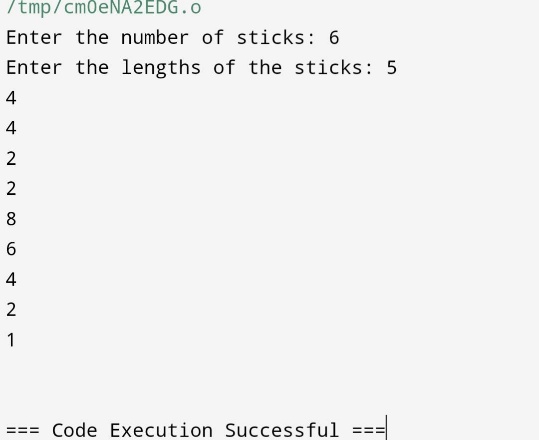
// Free the allocated memory

free(arr);

return 0;

}

Output:



Question 9:

Implement infix to postfix conversion

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIZE 100

// Stack structure

struct Stack {

int top;

unsigned capacity;

char\* array;

};

// Function to create a stack of given capacity

struct Stack\* createStack(unsigned capacity) {

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

stack->capacity = capacity;

stack->top = -1;

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

return stack;

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == stack->capacity - 1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element to the stack

void push(struct Stack\* stack, char item) {

if (isFull(stack))

return;

stack->array[++stack->top] = item;

}

// Function to pop an element from the stack

char pop(struct Stack\* stack) {

if (isEmpty(stack))

return '\0';

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

}

// Function to return the top element of the stack without popping it

char peek(struct Stack\* stack) {

if (isEmpty(stack))

return '\0';

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

}

// Function to check if a character is an operator

int isOperator(char ch) {

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

}

// Function to get the precedence of an operator

int precedence(char ch) {

switch (ch) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

default:

return -1;

}

}

// Function to convert infix expression to postfix expression

void infixToPostfix(char\* infix, char\* postfix) {

struct Stack\* stack = createStack(MAX\_SIZE);

int i, k;

for (i = 0, k = -1; infix[i]; ++i) {

if (isalnum(infix[i])) {

postfix[++k] = infix[i];

} else if (infix[i] == '(') {

push(stack, infix[i]);

} else if (infix[i] == ')') {

while (!isEmpty(stack) && peek(stack) != '(')

postfix[++k] = pop(stack);

if (!isEmpty(stack) && peek(stack) != '(')

return; // Invalid expression

else

pop(stack);

} else {

while (!isEmpty(stack) && precedence(infix[i]) <= precedence(peek(stack)))

postfix[++k] = pop(stack);

push(stack, infix[i]);

}

}

while (!isEmpty(stack))

postfix[++k] = pop(stack);

postfix[++k] = '\0';

}

int main() {

char infix[MAX\_SIZE];

printf("Enter infix expression: ");

fgets(infix, MAX\_SIZE, stdin);

infix[strcspn(infix, "\n")] = '\0'; // Remove newline character from input

char postfix[MAX\_SIZE];

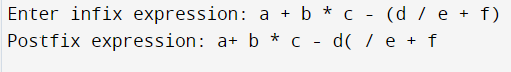
infixToPostfix(infix, postfix);

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

return 0;

}

Output



Question 10:

Evaluate a given postfix expression

Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIZE 100

// Stack structure

struct Stack {

int top;

unsigned capacity;

int\* array;

};

// Function to create a stack of given capacity

struct Stack\* createStack(unsigned capacity) {

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

stack->capacity = capacity;

stack->top = -1;

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

return stack;

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == stack->capacity - 1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element to the stack

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

if (isFull(stack))

return;

stack->array[++stack->top] = item;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack))

return -1;

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

}

// Function to evaluate postfix expression

int evaluatePostfix(char\* postfix) {

struct Stack\* stack = createStack(MAX\_SIZE);

int i;

for (i = 0; postfix[i]; ++i) {

if (isdigit(postfix[i])) {

push(stack, postfix[i] - '0');

} else {

int operand2 = pop(stack);

int operand1 = pop(stack);

switch (postfix[i]) {

case '+':

push(stack, operand1 + operand2);

break;

case '-':

push(stack, operand1 - operand2);

break;

case '\*':

push(stack, operand1 \* operand2);

break;

case '/':

push(stack, operand1 / operand2);

break;

}

}

}

return pop(stack);

}

int main() {

char postfix[MAX\_SIZE];

printf("Enter postfix expression: ");

fgets(postfix, MAX\_SIZE, stdin);

postfix[strcspn(postfix, "\n")] = '\0'; // Remove newline character from input

int result = evaluatePostfix(postfix);

printf("Result of evaluation: %d\n", result);

return 0;

}

Output



Question 11:

Implement Dijkstra’s Algorithm

Code:

#include <stdio.h>

#include <limits.h>

#define V 6 // Number of vertices in the graph

// Function to find the vertex with the minimum distance value, from the set of vertices not yet included in the shortest path tree

int minDistance(int dist[], int sptSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

// Function to print the constructed distance array

void printSolution(int dist[]) {

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

// Function that implements Dijkstra's algorithm for a graph represented using adjacency matrix representation

void dijkstra(int graph[V][V], int src) {

int dist[V]; // The output array. dist[i] will hold the shortest distance from src to i

int sptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest path tree or shortest distance from src to i is finalized

// Initialize all distances as INFINITE and sptSet[] as false

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

dist[i] = INT\_MAX, sptSet[i] = 0;

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum distance vertex from the set of vertices not yet processed.

// u is always equal to src in the first iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = 1;

// Update dist value of the adjacent vertices of the picked vertex.

for (int v = 0; v < V; v++) {

// Update dist[v] only if it is not in sptSet, there is an edge from u to v, and the total weight of path from src to v through u is smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

}

// Print the constructed distance array

printSolution(dist);

}

int main() {

// Graph representation using adjacency matrix

int graph[V][V] = {

{0, 7, 0, 5, 0, 0},

{7, 0, 8, 9, 7, 0},

{0, 8, 0, 0, 5, 0},

{5, 9, 0, 0, 15, 6},

{0, 7, 5, 15, 0, 8},

{0, 0, 0, 6, 8, 0}

};

// Source vertex

int src = 0;

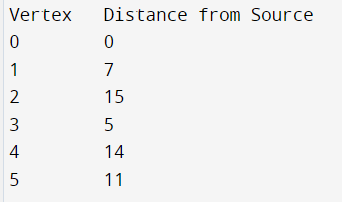
// Run Dijkstra's algorithm

dijkstra(graph, src);

return 0;

}

Output



Question 12:

Implement Floyd Warshall Algorithm

Code:

#include <stdio.h>

#include <limits.h>

#define V 4 // Number of vertices in the graph

// Function to print the solution matrix

void printSolution(int dist[][V]) {

printf("Shortest distances between every pair of vertices:\n");

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

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

if (dist[i][j] == INT\_MAX)

printf("INF\t");

else

printf("%d\t", dist[i][j]);

}

printf("\n");

}

}

// Function to implement Floyd-Warshall algorithm for a graph represented using adjacency matrix representation

void floydWarshall(int graph[][V]) {

int dist[V][V]; // Output matrix that will have the shortest distances between every pair of vertices

// Initialize the solution matrix with the same values as the input graph

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

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

dist[i][j] = graph[i][j];

}

}

// Update dist[][] to include the shortest path from every pair of vertices

for (int k = 0; k < V; k++) {

// Pick all vertices as source one by one

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

// Pick all vertices as destination for the above picked source

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

// If vertex k is on the shortest path from i to j, then update the value of dist[i][j]

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

// Print the solution matrix

printSolution(dist);

}

int main() {

// Graph representation using adjacency matrix

int graph[V][V] = {

{0, 5, INT\_MAX, 10},

{INT\_MAX, 0, 3, INT\_MAX},

{INT\_MAX, INT\_MAX, 0, 1},

{INT\_MAX, INT\_MAX, INT\_MAX, 0}

};

// Run Floyd-Warshall algorithm

floydWarshall(graph);

return 0;

}

Output

