Index

|  |  |  |  |
| --- | --- | --- | --- |
| Title | Date of Experiment | Date of Submission | Remark / Signature |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Subject: Algorithm Implementation Lab (CSE2251)

Day 1 Assignment

1. WAP to solve Towers-of-Hanoi problem using Recursive procedure

Code-

#include<stdio.h>

*int* count **=** 0 **;** *//global counter*

*void* toh(*int* **n** **,** *char* **A,** *char* **B,** *char* **C**)

{

**if**(**n==**0)

**return** **;**

**else**

    {

        toh(**n-**1**,A,C,B**)**;**

        printf("Move *%d* from *%c* to *%c* \n" **,** **n,A,C**)**;**

            count**++;** *//incre,ent counter at every move*

        toh(**n-**1**,B,A,C**)**;**

    }

}

*int* main()

{

    printf("Enter the number of disks: ")**;**

*int* n **;**

    scanf("*%d*"**,&**n)**;**

*char* A **=** 'S'**;** *//Source Rod*

*char* B **=** 'A'**;** *//Auxiliary Rod*

*char* C **=** 'D'**;** *//Destination Rod*

    toh ( n **,** A**,**  B**,**  C)**;**

    printf("The number of count is : *%d*"**,** count)**;**

}

Output –

Enter the number of disks: 3

Move 1 from S to D

Move 2 from S to A

Move 1 from D to A

Move 3 from S to D

Move 1 from A to S

Move 2 from A to D

Move 1 from S to D

The number of count is : 7

b) Write two recursive programs to compute X n , where both X &amp; n are integers, one

computes it in O(n) time &amp; the other in O(log n) time.

Code-

#include <stdio.h>

*int* linSteps **=** 0**,** binSteps **=** 0**;** *// Global counters*

*// O(n)*

*int* linPow(*int* **X,** *int* **n**) {

    linSteps**++;**

**if** (**n** **==** 0) **return** 1**;**

**return** **X** **\*** linPow(**X,** **n** **-** 1)**;**

}

*// O(log n)*

*int* binPow(*int* **X,** *int* **n**) {

    binSteps**++;**

**if** (**n** **==** 0) **return** 1**;**

*int* halfPower **=** binPow(**X,** **n** **/** 2)**;**

**if** (**n** **%** 2 **==** 0)

**return** halfPower **\*** halfPower**;**

**else**

**return** **X** **\*** halfPower **\*** halfPower**;**

}

*int* main() {

    printf("Enter the base and power : ")**;**

*int* X**,** n**;**

    scanf("*%d* *%d*" **,** **&**X **,** **&**n)**;**

*int* result1 **=** linPow(X**,** n)**;**

    printf("linPow(*%d*, *%d*) = *%d*, Steps = *%d*\n"**,** X**,** n**,** result1**,** linSteps)**;**

*int* result2 **=** binPow(X**,** n)**;**

    printf("binPow(*%d*, *%d*) = *%d*, Steps = *%d*\n"**,** X**,** n**,** result2**,** binSteps)**;**

**return** 0**;**

}

Output –

Enter the base and power : 2 10

linPow(2, 10) = 1024, Steps = 11

binPow(2, 10) = 1024, Steps = 5

Day 2 Assignment

1. Wap to implement merge sort algorithm on a randomly generated array that is stored in input.txt and give output in the output.txt in c language

Code-

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 10

*void* merge(*int* **arr**[]**,** *int* **left,** *int* **mid,** *int* **right**) {

*int* i**,** j**,** k**;**

*int* n1 **=** **mid** **-** **left** **+** 1**;**

*int* n2 **=** **right** **-** **mid;**

*int* L[n1]**,** R[n2]**;**

**for** (i **=** 0**;** i **<** n1**;** i**++**)

        L[i] **=** **arr**[**left** **+** i]**;**

**for** (j **=** 0**;** j **<** n2**;** j**++**)

        R[j] **=** **arr**[**mid** **+** 1 **+** j]**;**

    i **=** 0**;**

    j **=** 0**;**

    k **=** **left;**

**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**++;**

    }

}

*// Merge Sort function*

*void* mergeSort(*int* **arr**[]**,** *int* **left,** *int* **right**) {

**if** (**left** **<** **right**) {

*int* mid **=** **left** **+** (**right** **-** **left**) **/** 2**;**

        mergeSort(**arr,** **left,** mid)**;**

        mergeSort(**arr,** mid **+** 1**,** **right**)**;**

        merge(**arr,** **left,** mid**,** **right**)**;**

    }

}

*int* main() {

    FILE **\***inputFile **=** fopen("input.txt"**,** "w")**;**

**if** (inputFile **==** NULL) {

        printf("Error opening input file for writing!\n")**;**

**return** 1**;**

    }

    srand(time(0))**;**

**for** (*int* i **=** 0**;** i **<** SIZE**;** i**++**) {

        fprintf(inputFile**,** "*%d* "**,** (rand() **%** SIZE))**;**

    }

    fclose(inputFile)**;**

    inputFile **=** fopen("input.txt"**,** "r")**;**

**if** (inputFile **==** NULL) {

        printf("Error opening input file for reading!\n")**;**

**return** 1**;**

    }

*int* arr[SIZE]**,** n **=** 0**;**

**while** (fscanf(inputFile**,** "*%d*"**,** **&**arr[n]) **!=** EOF) {

        n**++;**

    }

    fclose(inputFile)**;**

    mergeSort(arr**,** 0**,** n **-** 1)**;**

    FILE **\***outputFile **=** fopen("output.txt"**,** "w")**;**

**if** (outputFile **==** NULL) {

        printf("Error opening output file!\n")**;**

**return** 1**;**

    }

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

        fprintf(outputFile**,** "*%d* "**,** arr[i])**;**

    }

    fclose(outputFile)**;**

    printf("Sorting complete. Check output.txt\n")**;**

**return** 0**;**

}

Input.txt Output.txt

0 0 2 4 5 6 6 8 8 9

5 6 0 6 8 9 2 8 4 0

1. Implement the same merge sort on ‘k’ different number of arrays where k will be the input from the user

Code-

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 100

*void* merge(*int* **arr**[]**,** *int* **left,** *int* **mid,** *int* **right**) {

*int* i**,** j**,** k**;**

*int* n1 **=** **mid** **-** **left** **+** 1**;**

*int* n2 **=** **right** **-** **mid;**

*int* L[n1]**,** R[n2]**;**

**for** (i **=** 0**;** i **<** n1**;** i**++**)

        L[i] **=** **arr**[**left** **+** i]**;**

**for** (j **=** 0**;** j **<** n2**;** j**++**)

        R[j] **=** **arr**[**mid** **+** 1 **+** j]**;**

    i **=** 0**;**

    j **=** 0**;**

    k **=** **left;**

**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**++;**

    }

}

*void* mergeSort(*int* **arr**[]**,** *int* **left,** *int* **right**) {

**if** (**left** **<** **right**) {

*int* mid **=** **left** **+** (**right** **-** **left**) **/** 2**;**

        mergeSort(**arr,** **left,** mid)**;**

        mergeSort(**arr,** mid **+** 1**,** **right**)**;**

        merge(**arr,** **left,** mid**,** **right**)**;**

    }

}

*int* main() {

*int* k**;**

    printf("Enter the number of arrays (k): ")**;**

    scanf("*%d*"**,** **&**k)**;**

    srand(time(0))**;**

    FILE **\***inputFile **=** fopen("input.txt"**,** "w")**;**

**if** (inputFile **==** NULL) {

        printf("Error opening input.txt for writing!\n")**;**

**return** 1**;**

    }

*int* arr[k][10]**;**

**for** (*int* i **=** 0**;** i **<** k**;** i**++**) {

**for** (*int* j **=** 0**;** j **<** 10**;** j**++**) {

            arr[i][j] **=** (rand() **%** SIZE)**;**

            fprintf(inputFile**,** "*%d* "**,** arr[i][j])**;**

        }

        fprintf(inputFile**,** "\n")**;**

    }

    fclose(inputFile)**;**

    FILE **\***outputFile **=** fopen("output.txt"**,** "w")**;**

**if** (outputFile **==** NULL) {

        printf("Error opening output.txt for writing!\n")**;**

**return** 1**;**

    }

**for** (*int* i **=** 0**;** i **<** k**;** i**++**) {

        mergeSort(arr[i]**,** 0**,** 9)**;**

**for** (*int* j **=** 0**;** j **<** 9**;** j**++**) {

            fprintf(outputFile**,** "*%d* "**,** arr[i][j])**;**

        }

        fprintf(outputFile**,** "\n")**;**

    }

    fclose(outputFile)**;**

    printf("Sorting complete. Check output.txt\n")**;**

**return** 0**;**

}

Input.txt Output.txt

7 18 42 46 49 51 51 66 82

10 13 16 29 32 48 63 74 86

10 14 20 28 39 49 65 69 90

29 30 35 39 40 51 54 67 71

5 6 8 24 32 40 72 93 98

1 10 14 14 45 52 54 57 60

6 14 34 52 56 61 72 75 78

8 12 14 23 24 39 67 76 91

7 10 13 14 27 48 52 61 63

0 2 3 7 54 66 72 72 83

66 49 51 7 82 18 51 46 42 82

94 32 74 16 29 48 63 13 86 10

10 69 28 65 90 39 14 20 99 49

39 35 67 76 29 71 54 30 40 51

5 93 24 8 72 99 6 32 40 98

10 14 14 54 60 45 69 52 57 1

34 78 52 14 6 56 86 72 75 61

97 67 39 23 76 12 14 91 8 24

14 52 10 63 27 13 65 7 48 61

72 83 0 7 3 66 2 54 90 72

Day 3 Assignment

1. WAP to implement QUICK sort using Divide-and-Conquer strategy. Estimate the number of comparisons .

Code- using Lomuto Partition scheme

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 10

int comparison\_count = 0;

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

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int lomuto\_partition(int arr[], int low, int high) {

    int pivot = arr[high];

    int i = low - 1;

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

        comparison\_count++;

        if (arr[j] < pivot) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return i + 1;

}

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

    if (low < high) {

        int pi = lomuto\_partition(arr, low, high);

        quicksort(arr, low, pi - 1);

        quicksort(arr, pi + 1, high);

    }

}

void generate\_input\_file() {

    FILE \*file = fopen("input.txt", "w");

    if (file == NULL) return;

    srand(time(0));

    for (int i = 0; i < SIZE; i++) {

        fprintf(file, "%d ", rand() % 100);

    }

    fclose(file);

}

void read\_input\_file(int arr[], int \*size) {

    FILE \*file = fopen("input.txt", "r");

    if (file == NULL) return;

    \*size = 0;

    while (fscanf(file, "%d", &arr[\*size]) != EOF) {

        (\*size)++;

    }

    fclose(file);

}

void write\_output\_file(int arr[], int size) {

    FILE \*file = fopen("output.txt", "w");

    if (file == NULL) return;

    for (int i = 0; i < size; i++) {

        fprintf(file, "%d ", arr[i]);

    }

    fclose(file);

}

int main() {

    generate\_input\_file();

    int arr[SIZE], size = 0;

    read\_input\_file(arr, &size);

    quicksort(arr, 0, size - 1);

    write\_output\_file(arr, size);

    printf("Sorting complete. Check output.txt for results.\n");

    printf("Total comparisons made: %d\n", comparison\_count);

    return 0;

}

OUTPUT-

INPUT.TXT OUTPUT.TXT

2 5 6 9 18 29 51 67 91 98

6 51 91 29 9 98 2 18 67 5

Sorting complete. Check output.txt for results.

Total comparisons made: 22

1. Implement randomized Quick sort and compare its performance with the previous algorithm by testing average case running time. You should write a separate sub program to generate random permutation for 10 numbers (0 - 9), that has to be used as a sub-part of the complete program.

CODE –

#include <cstdio>

#include <cstdlib>

using namespace std;

int totalComparisons = 0;

int permutationCount = 0;

int comparisons = 0;

void swap(int &a, int &b) {

    int temp = a;

    a = b;

    b = temp;

}

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

    int pivot = arr[high];

    int i = low - 1;

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

        comparisons++;

        if (arr[j] <= pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], arr[high]);

    return i + 1;

}

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

    if (low < high) {

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

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void processPermutation(int arr[], int n, FILE \*inputFile, FILE \*outputFile) {

    for (int i = 0; i < n; i++) {

        fprintf(inputFile, "%d ", arr[i]);

    }

    fprintf(inputFile, "\n");

    int \*temp = new int[n];

    for (int i = 0; i < n; i++) {

        temp[i] = arr[i];

    }

    comparisons = 0;

    quickSort(temp, 0, n - 1);

    for (int i = 0; i < n; i++) {

        fprintf(outputFile, "%d ", temp[i]);

    }

    fprintf(outputFile, "\n");

    printf("Permutation %d sorted with %d comparisons.\n", permutationCount + 1, comparisons);

    totalComparisons += comparisons;

    permutationCount++;

    delete [] temp;

}

void generatePermutations(int arr[], int start, int n, FILE \*inputFile, FILE \*outputFile) {

    if (start == n) {

        processPermutation(arr, n, inputFile, outputFile);

        return;

    }

    for (int i = start; i < n; i++) {

        swap(arr[start], arr[i]);

        generatePermutations(arr, start + 1, n, inputFile, outputFile);

        swap(arr[start], arr[i]);

    }

}

int main() {

    int n;

    printf("Enter number of elements: ");

    if (scanf("%d", &n) != 1) {

        fprintf(stderr, "Invalid input.\n");

        return EXIT\_FAILURE;

    }

    int \*arr = new int[n];

    for (int i = 0; i < n; i++) {

        arr[i] = i;

    }

    FILE \*inputFile = fopen("input.txt", "w");

    if (!inputFile) {

        perror("Failed to open input.txt");

        delete [] arr;

        return EXIT\_FAILURE;

    }

    FILE \*outputFile = fopen("output.txt", "w");

    if (!outputFile) {

        perror("Failed to open output.txt");

        fclose(inputFile);

        delete [] arr;

        return EXIT\_FAILURE;

    }

    generatePermutations(arr, 0, n, inputFile, outputFile);

    if (permutationCount > 0) {

        double average = (double)totalComparisons / permutationCount;

        printf("Total permutations processed: %d\n", permutationCount);

        printf("Average comparisons: %.2f\n", average);

    } else {

        printf("No permutations processed.\n");

    }

    delete [] arr;

    fclose(inputFile);

    fclose(outputFile);

    return 0;

}

OUTPUT –

Enter number of elements: 3

Permutation 1 sorted with 3 comparisons.

Permutation 2 sorted with 2 comparisons.

Permutation 3 sorted with 3 comparisons.

Permutation 4 sorted with 3 comparisons.

Permutation 5 sorted with 3 comparisons.

Permutation 6 sorted with 2 comparisons.

Total permutations processed: 6

Average comparisons: 2.67

0 1 2

0 2 1

1 0 2

1 2 0

2 1 0

2 0 1

0 1 2

0 1 2

0 1 2

0 1 2

0 1 2

0 1 2

DAY 4 ASSIGNMENT

WAP to multiply a chain of matrices optimally using iterative version of Dynamic Programming approach. Check your program for the sequence of 4 matrices < M1, M2, M3, M4 > whose sequence of dimensions is < 13, 5, 89, 3, 34 >

CODE

#include <stdio.h>

#include <limits.h>

// optimal parenthesization recursively

void printOptimalParenthesis(int s[][5], int i, int j, char \*name) {

    if (i == j) {

        printf("%c", (\*name)++);

        return;

    }

    printf("(");

    printOptimalParenthesis(s, i, s[i][j], name);

    printOptimalParenthesis(s, s[i][j] + 1, j, name);

    printf(")");

}

// display matrices

void displayMatrices(int m[][5], int s[][5], int n) {

    // Print m matrix

    printf("\nMatrix m:\n");

    for (int i = 1; i < n; i++) {

        for (int j = 1; j < n; j++) {

            if (j >= i)

                printf("%7d ", m[i][j]);

            else

                printf("       ");

        }

        printf("\n");

    }

    // s matrix

    printf("\nMatrix s:\n");

    for (int i = 1; i < n - 1; i++) {

        for (int j = 2; j < n; j++) {

            if (j > i)

                printf("%7d ", s[i][j]);

            else

                printf("       ");

        }

        printf("\n");

    }

}

// minimum cost of multiplying matrices

void matrixChainOrder(int p[], int n) {

    int m[n][n];

    int s[n][n];

    for (int i = 1; i < n; i++) {

        m[i][i] = 0;

    }

    for (int chainLength = 2; chainLength < n; chainLength++) {

        for (int i = 1; i < n - chainLength + 1; i++) {

            int j = i + chainLength - 1;

            m[i][j] = INT\_MAX;

            for (int k = i; k < j; k++) {

                int cost = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

                if (cost < m[i][j]) {

                    m[i][j] = cost;

                    s[i][j] = k;

                }

            }

        }

    }

    printf("Minimum number of multiplications is %d\n", m[1][n - 1]);

    // Display matrices

    displayMatrices(m, s, n);

    // Print optimal parenthesization

    printf("\nOptimal Parenthesization: ");

    char name = 'A';

    printOptimalParenthesis(s, 1, n - 1, &name);

    printf("\n");

}

int main() {

    int p[] = {13, 5, 89, 3, 34 }; //dimensions

    int n = sizeof(p) / sizeof(p[0]);

    matrixChainOrder(p, n);

    return 0;

}

OUTPUT

Minimum number of multiplications is 2856

Matrix m:

0 5785 1530 2856

0 1335 1845

0 9078

0

Matrix s:

1 1 3

2 3

3

Optimal Parenthesization: ((A(BC))D)

DAY 5 ASSIGNMENT

WAP to implement a skip list.

CODE

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

#define PLUS\_INFINITY INT\_MAX

#define MINUS\_INFINITY INT\_MIN

int c = 0; // Global comparison counter

typedef struct node  //Quad-node

{

    int key;

    struct node \*f;

    struct node \*b;

    struct node \*u;

    struct node \*d;

} node;

typedef struct skiplist // list

{

    node \*head;

    node \*tail;

    int level;

    int size;

} skiplist;

int toss()

{

    return rand() % 2; //genrates 2 values 0 and 1

}

node \*createNode(int key) //will create a quad node pointing to null on all four sides

{

    node \*newNode = (node \*)malloc(sizeof(node));

    if (newNode == NULL)

    {

        fprintf(stderr, "Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    newNode->key = key;

    newNode->f = NULL;

    newNode->b = NULL;

    newNode->u = NULL;

    newNode->d = NULL;

    return newNode;

}

skiplist \*createSkiplist() //will initialize a list at levvel 0

{

    skiplist \*sl = (skiplist \*)malloc(sizeof(skiplist));

    if (sl == NULL)

    {

        fprintf(stderr, "Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    node \*p2 = createNode(PLUS\_INFINITY); //node for +∞

    node \*p1 = createNode(MINUS\_INFINITY); //node for -∞

    p1->f = p2;

    p2->b = p1;

    sl->head = p1;

    sl->tail = p2;

    sl->level = 0;

    sl->size = 0;

    return sl;

}

void addEmptyList(skiplist \*sl) //will add an empty list to a level up of current level

{

    node \*p1 = createNode(MINUS\_INFINITY);

    node \*p2 = createNode(PLUS\_INFINITY);

    p1->f = p2;

    p1->d = sl->head;

    p2->b = p1;

    p2->d = sl->tail;

    sl->head->u = p1;

    sl->tail->u = p2;

    sl->head = p1;

    sl->tail = p2;

    sl->level++;

}

node \*search(skiplist \*sl, int value ) //searching for the element

{

    node \*t = sl->head;

    while (1)

    {

        while (t->f->key != PLUS\_INFINITY && t->f->key <= value) //if next element is not +∞ , and next element is lesser than key then ...

        {

            t = t->f; // move tail

            c++; // incrementing comparision counter

        }

        if (t->d)

        {

            t = t->d;

            c++; // one counter gets increment when compared with the next greater element and traversal moves down

        }

        else

            break;

    }

    return t;

}

void insert(skiplist \*sl, int value)

{

    c = 0;

    node \*p = search(sl, value); //searching for the value, returns a pointer to the node <= the value

    if (p->key == value) // if duplicates found

    {

        printf("Value %d already exists in the skiplist. Comparisons: %d\n", value, c);

        return;

    }

    node \*newNode = createNode(value); // quad node created

    newNode->f = p->f; // setting up the node pointers

    newNode->b = p;

    p->f->b = newNode;

    p->f = newNode;

    int current\_level = 0;

    node \*curr\_node = newNode;

    while (toss()) //tossing the coin

    {

        if (current\_level >= sl->level)

            addEmptyList(sl); //adding one extra list on top

        while (p->b && p->u == NULL)

            p = p->b;

        if (p->u == NULL)

            break;

        p = p->u;

        node \*newNodeUp = createNode(value); //adding the node to new levels if tosses a head (1)

        newNodeUp->f = p->f;

        newNodeUp->b = p;

        newNodeUp->d = curr\_node;

        p->f->b = newNodeUp;

        p->f = newNodeUp;

        curr\_node->u = newNodeUp;

        curr\_node = newNodeUp;

        current\_level++; //incrementing the level

    }

    // Always ensure the top level is empty.

    // If the top level contains a promoted node (i.e. it’s not just sentinels), add one extra level.

    if (sl->head->f != sl->tail)

    {

        addEmptyList(sl);

    }

    sl->size++; // incrementing the size of list

    printf("Value %d inserted. Comparisons: %d\n", value, c);

}

void deleteExtras(skiplist \*sl)

{

    // While there is a level below and the current top level is empty (only sentinels)

    while (sl->head->d != NULL && sl->head->f->key == PLUS\_INFINITY)

    {

        node\* old\_left = sl->head;      // current top level - left sentinel

        node\* old\_right = sl->tail;    // current top level - right sentinel

        // Move down one level

        sl->head = old\_left->d;

        sl->tail = old\_right->d;

        // Remove upward links from the new top level

        if (sl->head)

            sl->head->u = NULL;

        if (sl->tail)

            sl->tail->u = NULL;

        // Free the sentinel nodes of the old top level

        free(old\_left);

        free(old\_right);

        sl->level--;

    }

}

void deleteValue(skiplist \*sl, int value)

{

    c = 0;

    node \*p = search(sl, value); //searching for the value, returns a pointer to the node <= the value

    if (p->key != value)  // checks whether present or not

    {

        printf("Value %d not found. Comparisons: %d\n", value, c);

        return;

    }

    while (p)

    {

        p->b->f = p->f; //adjusting the pointer so to detach and hence delete the node

        p->f->b = p->b;

        node \*temp = p;

        p = p->u; //going up and deleting all instances

        free(temp);

    }

    sl->size--;

    printf("Value %d deleted. Comparisons: %d\n", value, c);

    deleteExtras(sl);//deleting the extra lists if present after deletion  (as only 1 list should be present in extra)

    if (sl->head->f != sl->tail) {

        addEmptyList(sl);

    }

}

void searchValue(skiplist \*sl, int value)

{

    c = 0;

    node \*p = search(sl, value); // will also return the pointer pointing to key <= value

    if (p->key == value)

        printf("Value %d found. Comparisons: %d\n", value, c);

    else

        printf("Value %d not found. Comparisons: %d\n", value, c);

}

void display(skiplist \*sl) // for displaying the skiplist

{

    node \*t = sl->head;

    while (t)

    {

        node \*q = t;

        printf("-INF -> ");

        while (q->f->key != PLUS\_INFINITY)

        {

            printf("%d -> ", q->f->key);

            q = q->f;

        }

        printf("+INF\n");

        t = t->d; //stops when t->d points to null

    }

}

int main()

{

    srand(time(NULL));

    skiplist \*sl = createSkiplist();

    int choice, value;

    do

    {

        printf("\n1. Insert\n2. Search\n3. Delete\n4. Display\n5. Exit\nEnter your choice: ");

        scanf("%d", &choice);

        switch (choice)

        {

        case 1:

            printf("Enter value to insert: ");

            scanf("%d", &value);

            insert(sl, value);

            break;

        case 2:

            printf("Enter value to search: ");

            scanf("%d", &value);

            searchValue(sl, value);

            break;

        case 3:

            printf("Enter value to delete: ");

            scanf("%d", &value);

            deleteValue(sl, value);

            break;

        case 4:

            display(sl);

            break;

        case 5:

            printf("Exiting...\n");

            break;

        default:

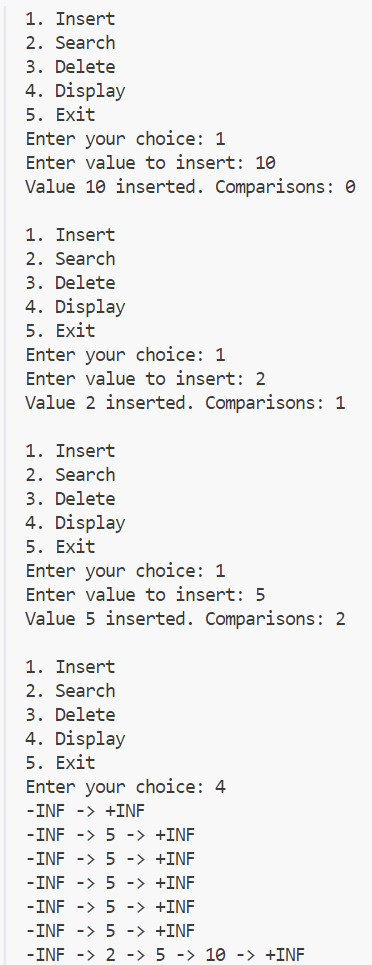
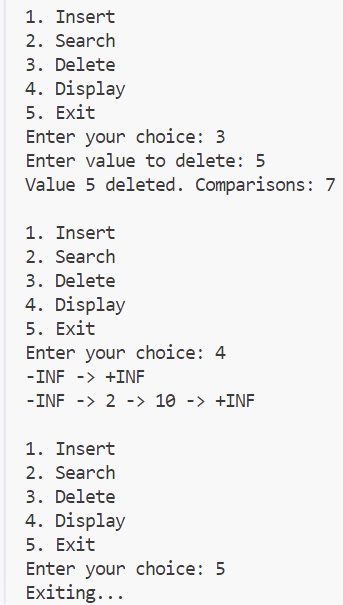
            printf("Invalid choice. Please try again.\n");

        }

    } while (choice != 5);

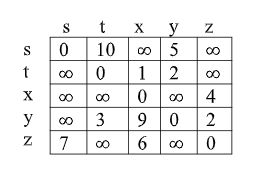
    return 0;

}

OUTPUT

DAY 6 ASSIGNMENT

a) Write a program to implement Djikstra’s Algorithm for Single Source Shortest Path problem. Your program should use priority queue implementation. Check your program on the directed graph represented by the following length matrix:



Code –

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define INF INT\_MAX

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

// Structure for adjacency list node

typedef struct Node {

int vertex;

int weight;

struct Node\* next;

} Node;

// Structure for adjacency list

typedef struct {

Node\* head;

} AdjList;

// Structure for the graph

typedef struct {

AdjList\* array;

} Graph;

// Structure for Min Heap Node

typedef struct {

int vertex;

int dist;

} MinHeapNode;

// Structure for Min Heap

typedef struct {

int size;

int capacity;

int \*pos;

MinHeapNode \*\*array;

} MinHeap;

// Function to create a new node

Node\* newNode(int vertex, int weight) {

Node\* temp = (Node\*)malloc(sizeof(Node));

temp->vertex = vertex;

temp->weight = weight;

temp->next = NULL;

return temp;

}

// Function to create a graph

Graph\* createGraph() {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->array = (AdjList\*)malloc(V \* sizeof(AdjList));

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

graph->array[i].head = NULL;

return graph;

}

// Function to add an edge to the graph

void addEdge(Graph\* graph, int src, int dest, int weight) {

Node\* temp = newNode(dest, weight);

temp->next = graph->array[src].head;

graph->array[src].head = temp;

}

MinHeap\* createMinHeap(int capacity) {

MinHeap\* minHeap = (MinHeap\*)malloc(sizeof(MinHeap));

minHeap->pos = (int\*)malloc(V \* sizeof(int));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (MinHeapNode\*\*)malloc(capacity \* sizeof(MinHeapNode\*));

return minHeap;

}

MinHeapNode\* newMinHeapNode(int vertex, int dist) {

MinHeapNode\* minHeapNode = (MinHeapNode\*)malloc(sizeof(MinHeapNode));

minHeapNode->vertex = vertex;

minHeapNode->dist = dist;

return minHeapNode;

}

void swapMinHeapNode(MinHeapNode\*\* a, MinHeapNode\*\* b) {

MinHeapNode\* temp = \*a;

\*a = \*b ;

\*b = temp;

}

// Min Heapify function

void minHeapify(MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->dist < minHeap->array[smallest]->dist)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->dist < minHeap->array[smallest]->dist)

smallest = right;

if (smallest != idx) {

MinHeapNode \*smallestNode = minHeap->array[smallest];

MinHeapNode \*idxNode = minHeap->array[idx];

minHeap->pos[smallestNode->vertex] = idx;

minHeap->pos[idxNode->vertex] = smallest;

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

//Extract-Min(V times)

MinHeapNode\* extractMin(MinHeap\* minHeap) {

if (minHeap->size == 0)

return NULL;

MinHeapNode\* root = minHeap->array[0];

MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];

minHeap->array[0] = lastNode;

minHeap->pos[root->vertex] = minHeap->size - 1;

minHeap->pos[lastNode->vertex] = 0;

minHeap->size--;

minHeapify(minHeap, 0);

return root;

}

//Decrease Key (E times)

void decreaseKey(MinHeap\* minHeap, int vertex, int dist) {

int i = minHeap->pos[vertex];

minHeap->array[i]->dist = dist;

while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist) {

minHeap->pos[minHeap->array[i]->vertex] = (i - 1) / 2;

minHeap->pos[minHeap->array[(i - 1) / 2]->vertex] = i;

swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

int isEmpty(MinHeap\* minHeap) {

return minHeap->size == 0;

}

// Printing the Path

void printPath(int parent[], int j) {

if (parent[j] == -1) {

printf("%c", 's' + j);

return;

}

printPath(parent, parent[j]);

printf(" -> %c", 's' + j);

}

void printAdjList(Graph\* graph) {

printf("Adjacency List:\n");

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

Node\* temp = graph->array[i].head;

printf("%c -> ", 's' + i);

while (temp != NULL) {

printf("%c(%d) -> ", 's' + temp->vertex, temp->weight);

temp = temp->next;

}

printf("NULL\n");

}

}

void dijkstra(Graph\* graph, int src) {

int dist[V];

int parent[V];

MinHeap\* minHeap = createMinHeap(V);

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

parent[v] = -1;

dist[v] = INF;

minHeap->array[v] = newMinHeapNode(v, dist[v]);

minHeap->pos[v] = v;

}

minHeap->array[src] = newMinHeapNode(src, 0);

minHeap->pos[src] = src;

dist[src] = 0;

decreaseKey(minHeap, src, dist[src]);

minHeap->size = V;

while (!isEmpty(minHeap)) {

MinHeapNode\* minNode = extractMin(minHeap);

int u = minNode->vertex;

Node\* temp = graph->array[u].head;

while (temp != NULL) {

int v = temp->vertex;

if (minHeap->pos[v] < minHeap->size && dist[u] != INF && temp->weight + dist[u] < dist[v]) {

dist[v] = dist[u] + temp->weight;

parent[v] = u;

decreaseKey(minHeap, v, dist[v]);

}

temp = temp->next;

}

}

printf("Vertex\tDistance\tPath\n");

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

printf("%c\t%d\t\t", 's' + i, dist[i]);

printPath(parent, i);

printf("\n");

}

}

int main() {

Graph\* graph = createGraph();

addEdge(graph, 0, 1, 10);

addEdge(graph, 0, 3, 5);

addEdge(graph, 1, 2, 1);

addEdge(graph, 1, 4, 2);

addEdge(graph, 2, 4, 4);

addEdge(graph, 3, 1, 3);

addEdge(graph, 3, 2, 9);

addEdge(graph, 3, 4, 2);

addEdge(graph, 4, 0, 7);

addEdge(graph, 4, 2, 6);

printAdjList(graph);

printf("\n");

printf("Dijkstra's Algorithm:\n");

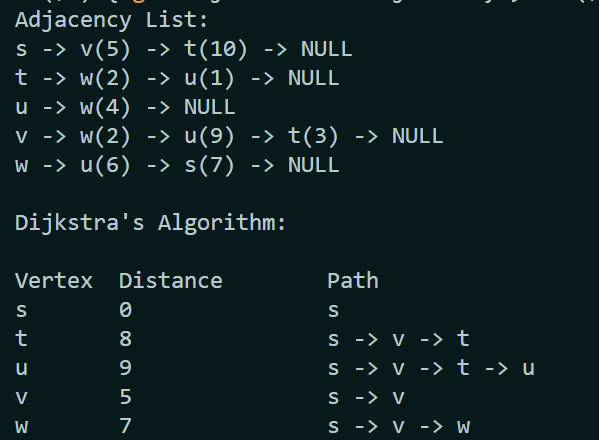
printf("\n");

dijkstra(graph, 0);

return 0;

}

output



DAY 7 ASSIGNMENT

WAP to find the MST of a given graph using Prim’s algorithm. Check your

program on a graph whose length matrix is given as follows. Your program

should use priority queue implementation.

CODE  
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

#define MAX\_VERTICES 1000

typedef struct AdjListNode {

    int dest;

    int weight;

    struct AdjListNode\* next;

} AdjListNode;

typedef struct AdjList {

    AdjListNode\* head;

} AdjList;

typedef struct Graph {

    int V;

    AdjList\* array;

} Graph;

typedef struct MinHeapNode {

    int v;

    int key;

} MinHeapNode;

typedef struct MinHeap {

    int size;

    int capacity;

    int\* pos;               // position of vertex in heap array

    MinHeapNode\*\* array;

} MinHeap;

AdjListNode\* newAdjListNode(int dest, int weight) {

    AdjListNode\* newNode = (AdjListNode\*) malloc(sizeof(AdjListNode));

    newNode->dest = dest;

    newNode->weight = weight;

    newNode->next = NULL;

    return newNode;

}

Graph\* createGraph(int V) {

    Graph\* graph = (Graph\*) malloc(sizeof(Graph));

    graph->V = V;

    graph->array = (AdjList\*) malloc(V \* sizeof(AdjList));

    for (int i = 0; i < V; ++i)

        graph->array[i].head = NULL;

    return graph;

}

void addEdge(Graph\* graph, int src, int dest, int weight) {

    AdjListNode\* newNode = newAdjListNode(dest, weight);

    newNode->next = graph->array[src].head;

    graph->array[src].head = newNode;

    newNode = newAdjListNode(src, weight);

    newNode->next = graph->array[dest].head;

    graph->array[dest].head = newNode;

}

MinHeapNode\* newMinHeapNode(int v, int key) {

    MinHeapNode\* minHeapNode = (MinHeapNode\*) malloc(sizeof(MinHeapNode));

    minHeapNode->v = v;

    minHeapNode->key = key;

    return minHeapNode;

}

MinHeap\* createMinHeap(int capacity) {

    MinHeap\* minHeap = (MinHeap\*) malloc(sizeof(MinHeap));

    minHeap->pos = (int\*) malloc(capacity \* sizeof(int));

    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array = (MinHeapNode\*\*) malloc(capacity \* sizeof(MinHeapNode\*));

    return minHeap;

}

void swapMinHeapNode(MinHeapNode\*\* a, MinHeapNode\*\* b) {

    MinHeapNode\* t = \*a;

    \*a = \*b;

    \*b = t;

}

void minHeapify(MinHeap\* minHeap, int idx) {

    int smallest = idx;

    int left = 2 \* idx + 1;

    int right = 2 \* idx + 2;

    if (left < minHeap->size &&

        minHeap->array[left]->key < minHeap->array[smallest]->key)

        smallest = left;

    if (right < minHeap->size &&

        minHeap->array[right]->key < minHeap->array[smallest]->key)

        smallest = right;

    if (smallest != idx) {

        MinHeapNode\* smallestNode = minHeap->array[smallest];

        MinHeapNode\* idxNode = minHeap->array[idx];

        minHeap->pos[smallestNode->v] = idx;

        minHeap->pos[idxNode->v] = smallest;

        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);

    }

}

// Check if MinHeap is empty

int isEmpty(MinHeap\* minHeap) {

    return minHeap->size == 0;

}

// Extract min node from heap

MinHeapNode\* extractMin(MinHeap\* minHeap) {

    if (isEmpty(minHeap))

        return NULL;

    MinHeapNode\* root = minHeap->array[0];

    MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];

    minHeap->array[0] = lastNode;

    minHeap->pos[root->v] = minHeap->size-1;

    minHeap->pos[lastNode->v] = 0;

    --minHeap->size;

    minHeapify(minHeap, 0);

    return root;

}

// Decrease key value

void decreaseKey(MinHeap\* minHeap, int v, int key) {

    int i = minHeap->pos[v];

    minHeap->array[i]->key = key;

    while (i && minHeap->array[i]->key < minHeap->array[(i-1)/2]->key) {

        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;

        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;

        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i-1)/2]);

        i = (i-1)/2;

    }

}

// Check if vertex v is in minHeap

int isInMinHeap(MinHeap \*minHeap, int v) {

    if (minHeap->pos[v] < minHeap->size)

        return 1;

    return 0;

}

// Main Prim's algorithm

void PrimMST(Graph\* graph) {

    int V = graph->V;

    int parent[V];    // Store constructed MST

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

    MinHeap\* minHeap = createMinHeap(V);

    for (int v = 0; v < V; ++v) {

        parent[v] = -1;

        key[v] = INT\_MAX;

        minHeap->array[v] = newMinHeapNode(v, key[v]);

        minHeap->pos[v] = v;

    }

    // Pick random source vertex

    srand(time(NULL));

    int src = rand() % V;

    printf("Random starting vertex: %d\n", src);

    minHeap->array[src] = newMinHeapNode(src, key[src]);

    minHeap->pos[src] = src;

    key[src] = 0;

    decreaseKey(minHeap, src, key[src]);

    minHeap->size = V;

    while (!isEmpty(minHeap)) {

        MinHeapNode\* minHeapNode = extractMin(minHeap);

        int u = minHeapNode->v;

        AdjListNode\* pCrawl = graph->array[u].head;

        while (pCrawl != NULL) {

            int v = pCrawl->dest;

            if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {

                key[v] = pCrawl->weight;

                parent[v] = u;

                decreaseKey(minHeap, v, key[v]);

            }

            pCrawl = pCrawl->next;

        }

    }

    printf("Edge \tWeight\n");

    int totalWeight = 0;

    for (int i = 0; i < V; ++i) {

        if (parent[i] != -1) {

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

            totalWeight += key[i];

        }

    }

    printf("Total MST Weight: %d\n", totalWeight);

}

// Read graph from file

Graph\* readGraphFromFile(const char\* filename) {

    FILE\* file = fopen(filename, "r");

    if (!file) {

        printf("Error opening file.\n");

        exit(1);

    }

    int V;

    fscanf(file, "%d", &V);

    Graph\* graph = createGraph(V);

    int src, dest, weight;

    while (fscanf(file, "%d %d %d", &src, &dest, &weight) == 3) {

        addEdge(graph, src, dest, weight);

    }

    fclose(file);

    return graph;

}

int main() {

    Graph\* graph = readGraphFromFile("graph.txt");

    PrimMST(graph);

    return 0;

}

OUTPUT graph.txt

9

0 1 4

0 7 8

1 2 8

1 7 11

2 3 7

2 5 4

2 8 2

3 4 9

3 5 14

4 5 10

5 6 2

6 7 1

6 8 6

7 8 7

Random starting vertex: 2

Edge Weight

7 - 0 8

0 - 1 4

2 - 3 7

3 - 4 9

2 - 5 4

5 - 6 2

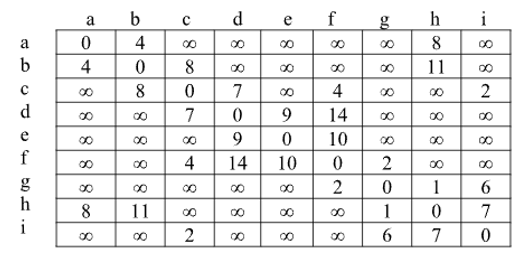
6 - 7 1

2 - 8 2

Total MST Weight: 37

DAY 8 ASSIGNMENT

Write a Program to find the MST of a given graph using Kruskal’s algorithm. Check your program on a graph whose length matrix is given above. Your program should use UNION & FIND operations on Disjoint data structure.



Code

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_NODES 100

typedef struct Node

{

    int vertex;

    int weight;

    struct Node \*next;

} Node;

typedef struct Graph

{

    int numVertices;

    struct Node \*\*adjLists;

} Graph;

typedef struct

{

    int u, v, weight;

} Edge;

int parent[MAX\_NODES], rank[MAX\_NODES];

Edge edges[MAX\_NODES \* MAX\_NODES];

int edgeCount = 0;

Node \*createNode(int v, int w)

{

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

    newNode->vertex = v;

    newNode->weight = w;

    newNode->next = NULL;

    return newNode;

}

Graph \*createGraph(int vertices)

{

    Graph \*graph = (Graph \*)malloc(sizeof(Graph));

    graph->numVertices = vertices;

    graph->adjLists = (struct Node \*\*)malloc(vertices \* sizeof(struct Node \*));

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

        graph->adjLists[i] = NULL;

    return graph;

}

void addEdge(Graph \*graph, int u, int v, int w)

{

    Node \*newNode = createNode(v, w);

    newNode->next = graph->adjLists[u];

    graph->adjLists[u] = newNode;

    newNode = createNode(u, w);

    newNode->next = graph->adjLists[v];

    graph->adjLists[v] = newNode;

}

void makeSet(int n)

{

    for (int i = 0; i < n; i++)

    {

        parent[i] = i;

        rank[i] = 0;

    }

}

int findSet(int v)

{

    if (parent[v] != v)

        parent[v] = findSet(parent[v]);

    return parent[v];

}

void unionSets(int u, int v)

{

    int rootU = findSet(u);

    int rootV = findSet(v);

    if (rootU != rootV)

    {

        if (rank[rootU] < rank[rootV])

            parent[rootU] = rootV;

        else if (rank[rootU] > rank[rootV])

            parent[rootV] = rootU;

        else

        {

            parent[rootV] = rootU;

            rank[rootU]++;

        }

    }

}

void swap(Edge \*a, Edge \*b)

{

    Edge temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int randomizedPartition(Edge arr[], int low, int high)

{

    int p = low + rand() % (high - low + 1);

    Edge temp = arr[p];

    arr[p] = arr[high];

    arr[high] = temp;

    int pivot = arr[high].weight;

    int i = low - 1;

    for (int j = low; j < high; j++)

    {

        if (arr[j].weight <= pivot)

        {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return i + 1;

}

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

{

    if (low < high)

    {

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

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void buildEdgeList(Graph \*graph, int n)

{

    int visited[MAX\_NODES][MAX\_NODES] = {0};

    edgeCount = 0;

    for (int u = 0; u < n; u++)

    {

        Node \*temp = graph->adjLists[u];

        while (temp)

        {

            int v = temp->vertex;

            int w = temp->weight;

            if (!visited[u][v])

            {

                edges[edgeCount].u = u;

                edges[edgeCount].v = v;

                edges[edgeCount].weight = w;

                edgeCount++;

                visited[u][v] = visited[v][u] = 1;

            }

            temp = temp->next;

        }

    }

}

void kruskalMST(Graph \*graph, int vertices)

{

    makeSet(vertices);

    buildEdgeList(graph, vertices);

    quickSort(edges, 0, edgeCount - 1);

    int totalWeight = 0;

    printf("Edges in the Minimum Spanning Tree:\n");

    printf("Vertices    Weight\n");

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

    {

        int u = edges[i].u;

        int v = edges[i].v;

        int w = edges[i].weight;

        if (findSet(u) != findSet(v))

        {

            printf("%d - %d\t\t%d\n", u, v, w);

            totalWeight += w;

            unionSets(u, v);

        }

    }

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

}

int main()

{

    srand(time(NULL));

    int vertices;

    FILE \*fptr = fopen("graph.txt", "r");

    fscanf(fptr, "%d", &vertices);

    struct Graph \*graph = createGraph(vertices);

    while (!feof(fptr))

    {

        int a, b, c;

        fscanf(fptr, "%d%d%d", &a, &b, &c);

        addEdge(graph, a, b, c);

    }

    kruskalMST(graph, vertices);

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

    {

        Node \*curr = graph->adjLists[i];

        while (curr)

        {

            Node \*temp = curr;

            curr = curr->next;

            free(temp);

        }

    }

    free(graph);

    return 0;

}

#OUTPUT

Edges in the Minimum Spanning Tree:

Vertices Weight

6 - 7 1

2 - 8 2

5 - 6 2

2 - 5 4

0 - 1 4

2 - 3 7

0 - 7 8

3 - 4 9

Total weight of MST: 37

DAY 9 ASSIGNMENT

Write a Program to implement Knuth-Morris-Pratt (KMP) algorithm for pattern matching. Check your program for the following Text & Pattern:

T: b a c b a b a b a c a b a b a c a

P: a b a b a c a

Code

#include <stdio.h>

#include <string.h>

// Function to compute the LPS (Longest Prefix Suffix) array

void computeLPSArray(char\* pattern, int M, int\* lps) {

    int length = 0;

    lps[0] = 0; // LPS of first character is always 0

    int i = 1;

    while (i < M) {

        if (pattern[i] == pattern[length]) {

            length++;

            lps[i] = length;

            i++;

        } else {

            if (length != 0) {

                length = lps[length - 1];

            } else {

                lps[i] = 0;

                i++;

            }

        }

    }

}

// KMP search function

void KMPSearch(char\* pattern, char\* text) {

    int M = strlen(pattern);

    int N = strlen(text);

    int lps[M];

    computeLPSArray(pattern, M, lps);

    int i = 0; // index for text[]

    int j = 0; // index for pattern[]

    while (i < N) {

        if (pattern[j] == text[i]) {

            i++;

            j++;

        }

        if (j == M) {

            printf("Pattern found at index %d\n", i - j);

            j = lps[j - 1]; // Look for next possible match

        } else if (i < N && pattern[j] != text[i]) {

            if (j != 0) {

                j = lps[j - 1];

            } else {

                i++;

            }

        }

    }

}

int main() {

    char text[] = "bacbababacababaca";

    char pattern[] = "ababaca";

    KMPSearch(pattern, text);

    return 0;

}  
  
  
#OUTPUT

Pattern found at index 4

Pattern found at index 10