FALL – SEMESTER **Name: Nidhi Singh**
Course Code: MCSE501P **Reg. No:22MAI0015**
Course-Title: – Data Structures and Algorithms
DIGITAL ASSIGNMENT - V
(LAB) **Slot-** L37+L38

Faculty: SARAVANAN R - SCOPE

List of programs for lab 5 :-

19. Write a program in C to implement a B+ tree and operations on it
20. Write a program to implement splay tree
21. Write a program to implement topological sort
22. Write a program to implement Dijkstra's algorithm
23. Write a program to implement Floyd Warshall algorithm
24. Write a program in C to implement Heap sort

19. Write a program in C to implement a B+ tree and operations on it

```c
// Searching on a B+ Tree in C

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

// Default order
#define ORDER 3

typedef struct record {
  int value;
} record;

// Node
typedef struct node {
  void **pointers;
  int *keys;
  struct node *parent;
  bool is_leaf;
  int num_keys;
  struct node *next;
} node;

int order = ORDER;
node *queue = NULL;
```

```
bool verbose_output = false;

// Enqueue
void enqueue(node *new_node);

// Dequeue
node *dequeue(void);
int height(node *const root);
int pathToLeaves(node *const root, node *child);
void printLeaves(node *const root);
void printTree(node *const root);
void findAndPrint(node *const root, int key, bool verbose);
void findAndPrintRange(node *const root, int range1, int range2, bool
verbose);
int findRange(node *const root, int key_start, int key_end, bool
verbose,
        int returned_keys[], void *returned_pointers[]);
node *findLeaf(node *const root, int key, bool verbose);
record *find(node *root, int key, bool verbose, node **leaf_out);
int cut(int length);

record *makeRecord(int value);
node *makeNode(void);
node *makeLeaf(void);
int getLeftIndex(node *parent, node *left);
node *insertIntoLeaf(node *leaf, int key, record *pointer);
node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
                   record *pointer);
node *insertIntoNode(node *root, node *parent,
           int left_index, int key, node *right);
node *insertIntoNodeAfterSplitting(node *root, node *parent,
                   int left_index,
                   int key, node *right);
node *insertIntoParent(node *root, node *left, int key, node *right);
node *insertIntoNewRoot(node *left, int key, node *right);
node *startNewTree(int key, record *pointer);
node *insert(node *root, int key, int value);

// Enqueue
void enqueue(node *new_node) {
  node *c;
  if (queue == NULL) {
    queue = new_node;
    queue->next = NULL;
  } else {
    c = queue;
    while (c->next != NULL) {
      c = c->next;
    }
    c->next = new_node;
    new_node->next = NULL;
```

```
  }
}

// Dequeue
node *dequeue(void) {
  node *n = queue;
  queue = queue->next;
  n->next = NULL;
  return n;
}

// Print the leaves
void printLeaves(node *const root) {
  if (root == NULL) {
    printf("Empty tree.\n");
    return;
  }
  int i;
  node *c = root;
  while (!c->is_leaf)
    c = c->pointers[0];
  while (true) {
    for (i = 0; i < c->num_keys; i++) {
      if (verbose_output)
        printf("%p ", c->pointers[i]);
      printf("%d ", c->keys[i]);
    }
    if (verbose_output)
      printf("%p ", c->pointers[order - 1]);
    if (c->pointers[order - 1] != NULL) {
      printf(" | ");
      c = c->pointers[order - 1];
    } else
      break;
  }
  printf("\n");
}

// Calculate height
int height(node *const root) {
  int h = 0;
  node *c = root;
  while (!c->is_leaf) {
    c = c->pointers[0];
    h++;
  }
  return h;
}

// Get path to root
int pathToLeaves(node *const root, node *child) {
```

```
  int length = 0;
  node *c = child;
  while (c != root) {
    c = c->parent;
    length++;
  }
  return length;
}

// Print the tree
void printTree(node *const root) {
  node *n = NULL;
  int i = 0;
  int rank = 0;
  int new_rank = 0;

  if (root == NULL) {
    printf("Empty tree.\n");
    return;
  }
  queue = NULL;
  enqueue(root);
  while (queue != NULL) {
    n = dequeue();
    if (n->parent != NULL && n == n->parent->pointers[0]) {
      new_rank = pathToLeaves(root, n);
      if (new_rank != rank) {
        rank = new_rank;
        printf("\n");
      }
    }
    if (verbose_output)
      printf("(%p)", n);
    for (i = 0; i < n->num_keys; i++) {
      if (verbose_output)
        printf("%p ", n->pointers[i]);
      printf("%d ", n->keys[i]);
    }
    if (!n->is_leaf)
      for (i = 0; i <= n->num_keys; i++)
        enqueue(n->pointers[i]);
    if (verbose_output) {
      if (n->is_leaf)
        printf("%p ", n->pointers[order - 1]);
      else
        printf("%p ", n->pointers[n->num_keys]);
    }
    printf("| ");
  }
  printf("\n");
}
```

```
// Find the node and print it
void findAndPrint(node *const root, int key, bool verbose) {
  node *leaf = NULL;
  record *r = find(root, key, verbose, NULL);
  if (r == NULL)
    printf("Record not found under key %d.\n", key);
  else
    printf("Record at %p -- key %d, value %d.\n",
         r, key, r->value);
}

// Find and print the range
void findAndPrintRange(node *const root, int key_start, int key_end,
             bool verbose) {
  int i;
  int array_size = key_end - key_start + 1;
  int returned_keys[array_size];
  void *returned_pointers[array_size];
  int num_found = findRange(root, key_start, key_end, verbose,
                returned_keys, returned_pointers);
  if (!num_found)
    printf("None found.\n");
  else {
    for (i = 0; i < num_found; i++)
      printf("Key: %d   Location: %p  Value: %d\n",
           returned_keys[i],
           returned_pointers[i],
           ((record *)
            returned_pointers[i])
             ->value);
  }
}

// Find the range
int findRange(node *const root, int key_start, int key_end, bool
verbose,
        int returned_keys[], void *returned_pointers[]) {
  int i, num_found;
  num_found = 0;
  node *n = findLeaf(root, key_start, verbose);
  if (n == NULL)
    return 0;
  for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++)
    ;
  if (i == n->num_keys)
    return 0;
  while (n != NULL) {
    for (; i < n->num_keys && n->keys[i] <= key_end; i++) {
      returned_keys[num_found] = n->keys[i];
      returned_pointers[num_found] = n->pointers[i];
```

```
      num_found++;
    }
    n = n->pointers[order - 1];
    i = 0;
  }
  return num_found;
}

// Find the leaf
node *findLeaf(node *const root, int key, bool verbose) {
  if (root == NULL) {
    if (verbose)
      printf("Empty tree.\n");
    return root;
  }
  int i = 0;
  node *c = root;
  while (!c->is_leaf) {
    if (verbose) {
      printf("[");
      for (i = 0; i < c->num_keys - 1; i++)
        printf("%d ", c->keys[i]);
      printf("%d] ", c->keys[i]);
    }
    i = 0;
    while (i < c->num_keys) {
      if (key >= c->keys[i])
        i++;
      else
        break;
    }
    if (verbose)
      printf("%d ->\n", i);
    c = (node *)c->pointers[i];
  }
  if (verbose) {
    printf("Leaf [");
    for (i = 0; i < c->num_keys - 1; i++)
      printf("%d ", c->keys[i]);
    printf("%d] ->\n", c->keys[i]);
  }
  return c;
}

record *find(node *root, int key, bool verbose, node **leaf_out) {
  if (root == NULL) {
    if (leaf_out != NULL) {
      *leaf_out = NULL;
    }
    return NULL;
  }
```

```
  int i = 0;
  node *leaf = NULL;

  leaf = findLeaf(root, key, verbose);

  for (i = 0; i < leaf->num_keys; i++)
    if (leaf->keys[i] == key)
      break;
  if (leaf_out != NULL) {
    *leaf_out = leaf;
  }
  if (i == leaf->num_keys)
    return NULL;
  else
    return (record *)leaf->pointers[i];
}

int cut(int length) {
  if (length % 2 == 0)
    return length / 2;
  else
    return length / 2 + 1;
}

record *makeRecord(int value) {
  record *new_record = (record *)malloc(sizeof(record));
  if (new_record == NULL) {
    perror("Record creation.");
    exit(EXIT_FAILURE);
  } else {
    new_record->value = value;
  }
  return new_record;
}

node *makeNode(void) {
  node *new_node;
  new_node = malloc(sizeof(node));
  if (new_node == NULL) {
    perror("Node creation.");
    exit(EXIT_FAILURE);
  }
  new_node->keys = malloc((order - 1) * sizeof(int));
  if (new_node->keys == NULL) {
    perror("New node keys array.");
    exit(EXIT_FAILURE);
  }
  new_node->pointers = malloc(order * sizeof(void *));
  if (new_node->pointers == NULL) {
    perror("New node pointers array.");
```

```
    exit(EXIT_FAILURE);
  }
  new_node->is_leaf = false;
  new_node->num_keys = 0;
  new_node->parent = NULL;
  new_node->next = NULL;
  return new_node;
}

node *makeLeaf(void) {
  node *leaf = makeNode();
  leaf->is_leaf = true;
  return leaf;
}

int getLeftIndex(node *parent, node *left) {
  int left_index = 0;
  while (left_index <= parent->num_keys &&
       parent->pointers[left_index] != left)
    left_index++;
  return left_index;
}

node *insertIntoLeaf(node *leaf, int key, record *pointer) {
  int i, insertion_point;

  insertion_point = 0;
  while (insertion_point < leaf->num_keys && leaf->keys[insertion_point]
< key)
    insertion_point++;

  for (i = leaf->num_keys; i > insertion_point; i--) {
    leaf->keys[i] = leaf->keys[i - 1];
    leaf->pointers[i] = leaf->pointers[i - 1];
  }
  leaf->keys[insertion_point] = key;
  leaf->pointers[insertion_point] = pointer;
  leaf->num_keys++;
  return leaf;
}

node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
record *pointer) {
  node *new_leaf;
  int *temp_keys;
  void **temp_pointers;
  int insertion_index, split, new_key, i, j;

  new_leaf = makeLeaf();

  temp_keys = malloc(order * sizeof(int));
```

```
  if (temp_keys == NULL) {
    perror("Temporary keys array.");
    exit(EXIT_FAILURE);
  }

  temp_pointers = malloc(order * sizeof(void *));
  if (temp_pointers == NULL) {
    perror("Temporary pointers array.");
    exit(EXIT_FAILURE);
  }

  insertion_index = 0;
  while (insertion_index < order - 1 && leaf->keys[insertion_index] <
key)
    insertion_index++;

  for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
    if (j == insertion_index)
      j++;
    temp_keys[j] = leaf->keys[i];
    temp_pointers[j] = leaf->pointers[i];
  }

  temp_keys[insertion_index] = key;
  temp_pointers[insertion_index] = pointer;

  leaf->num_keys = 0;

  split = cut(order - 1);

  for (i = 0; i < split; i++) {
    leaf->pointers[i] = temp_pointers[i];
    leaf->keys[i] = temp_keys[i];
    leaf->num_keys++;
  }

  for (i = split, j = 0; i < order; i++, j++) {
    new_leaf->pointers[j] = temp_pointers[i];
    new_leaf->keys[j] = temp_keys[i];
    new_leaf->num_keys++;
  }

  free(temp_pointers);
  free(temp_keys);

  new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
  leaf->pointers[order - 1] = new_leaf;

  for (i = leaf->num_keys; i < order - 1; i++)
    leaf->pointers[i] = NULL;
  for (i = new_leaf->num_keys; i < order - 1; i++)
```

```
    new_leaf->pointers[i] = NULL;

  new_leaf->parent = leaf->parent;
  new_key = new_leaf->keys[0];

  return insertIntoParent(root, leaf, new_key, new_leaf);
}

node *insertIntoNode(node *root, node *n,
           int left_index, int key, node *right) {
  int i;

  for (i = n->num_keys; i > left_index; i--) {
    n->pointers[i + 1] = n->pointers[i];
    n->keys[i] = n->keys[i - 1];
  }
  n->pointers[left_index + 1] = right;
  n->keys[left_index] = key;
  n->num_keys++;
  return root;
}

node *insertIntoNodeAfterSplitting(node *root, node *old_node, int
left_index,
                   int key, node *right) {
  int i, j, split, k_prime;
  node *new_node, *child;
  int *temp_keys;
  node **temp_pointers;

  temp_pointers = malloc((order + 1) * sizeof(node *));
  if (temp_pointers == NULL) {
    exit(EXIT_FAILURE);
  }
  temp_keys = malloc(order * sizeof(int));
  if (temp_keys == NULL) {
    exit(EXIT_FAILURE);
  }

  for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
    if (j == left_index + 1)
      j++;
    temp_pointers[j] = old_node->pointers[i];
  }

  for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
    if (j == left_index)
      j++;
    temp_keys[j] = old_node->keys[i];
  }
```

```
  temp_pointers[left_index + 1] = right;
  temp_keys[left_index] = key;

  split = cut(order);
  new_node = makeNode();
  old_node->num_keys = 0;
  for (i = 0; i < split - 1; i++) {
    old_node->pointers[i] = temp_pointers[i];
    old_node->keys[i] = temp_keys[i];
    old_node->num_keys++;
  }
  old_node->pointers[i] = temp_pointers[i];
  k_prime = temp_keys[split - 1];
  for (++i, j = 0; i < order; i++, j++) {
    new_node->pointers[j] = temp_pointers[i];
    new_node->keys[j] = temp_keys[i];
    new_node->num_keys++;
  }
  new_node->pointers[j] = temp_pointers[i];
  free(temp_pointers);
  free(temp_keys);
  new_node->parent = old_node->parent;
  for (i = 0; i <= new_node->num_keys; i++) {
    child = new_node->pointers[i];
    child->parent = new_node;
  }

  return insertIntoParent(root, old_node, k_prime, new_node);
}

node *insertIntoParent(node *root, node *left, int key, node *right) {
  int left_index;
  node *parent;

  parent = left->parent;

  if (parent == NULL)
    return insertIntoNewRoot(left, key, right);

  left_index = getLeftIndex(parent, left);

  if (parent->num_keys < order - 1)
    return insertIntoNode(root, parent, left_index, key, right);

  return insertIntoNodeAfterSplitting(root, parent, left_index, key,
right);
}

node *insertIntoNewRoot(node *left, int key, node *right) {
  node *root = makeNode();
  root->keys[0] = key;
```

```
  root->pointers[0] = left;
  root->pointers[1] = right;
  root->num_keys++;
  root->parent = NULL;
  left->parent = root;
  right->parent = root;
  return root;
}

node *startNewTree(int key, record *pointer) {
  node *root = makeLeaf();
  root->keys[0] = key;
  root->pointers[0] = pointer;
  root->pointers[order - 1] = NULL;
  root->parent = NULL;
  root->num_keys++;
  return root;
}

node *insert(node *root, int key, int value) {
  record *record_pointer = NULL;
  node *leaf = NULL;

  record_pointer = find(root, key, false, NULL);
  if (record_pointer != NULL) {
    record_pointer->value = value;
    return root;
  }

  record_pointer = makeRecord(value);

  if (root == NULL)
    return startNewTree(key, record_pointer);

  leaf = findLeaf(root, key, false);

  if (leaf->num_keys < order - 1) {
    leaf = insertIntoLeaf(leaf, key, record_pointer);
    return root;
  }

  return insertIntoLeafAfterSplitting(root, leaf, key, record_pointer);
}

int main() {
       int n, value;
       printf("\nenter the number of item want to enter :\t");
       scanf("%d",&n);
       int a[n];
  node *root;
  char instruction;
```

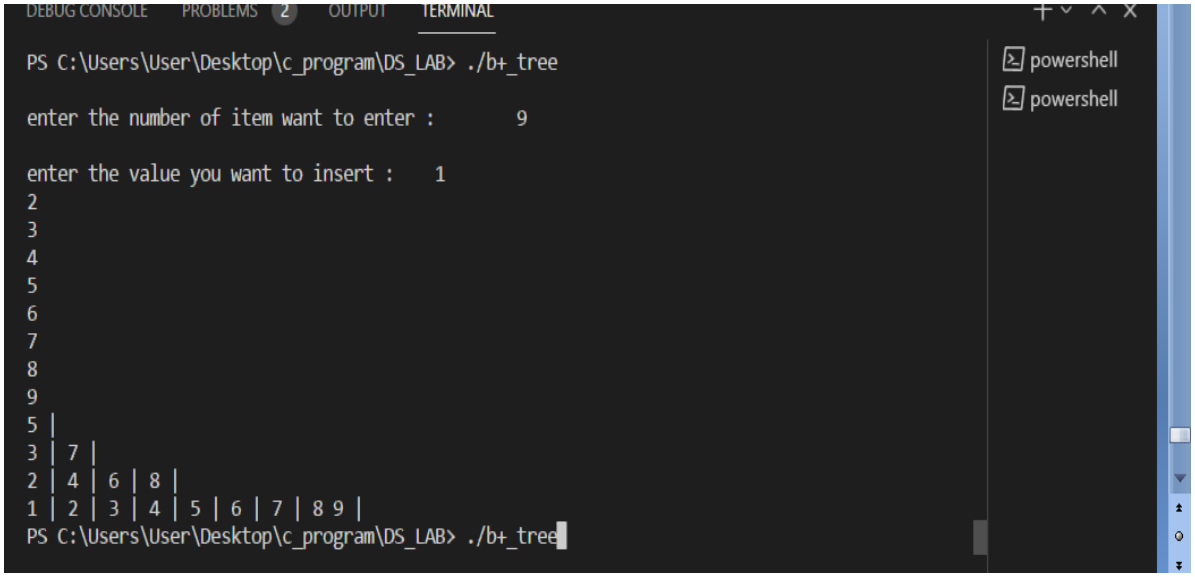
```
 root = NULL;
printf("\nenter the value you want to insert :\t");
for(int i=0;i<n;i++)
{
     scanf("%d", &a[i]);
     root = insert(root, a[i], 21);

}

 printTree(root);

 }
```

Output:-



20. Write a program to implement splay tree

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

struct node
{
    int key;
    struct node *left, *right;
};

struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

struct node *splay(struct node *root, int key)
{
     if (root == NULL || root->key == key)
        return root;
   if (root->key > key)
    {
         if (root->left == NULL) return root;

         if (root->left->key > key)
        {
             root->left->left = splay(root->left->left, key);

             root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
```

```
        {
             root->left->right = splay(root->left->right, key);
             if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }

         return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {
         if (root->right == NULL) return root;
       if (root->right->key > key)
        {
             root->right->left = splay(root->right->left, key);
             if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {
             root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

         return (root->right == NULL)? root: leftRotate(root);
    }
}

struct node *insert(struct node *root, int k)
{
     if (root == NULL) return newNode(k);
    root = splay(root, k);
    if (root->key == k) return root;
 struct node *newnode = newNode(k);
 if (root->key > k)
    {
        newnode->right = root;
        newnode->left = root->left;
        root->left = NULL;
    }
 else
    {
        newnode->left = root;
        newnode->right = root->right;
        root->right = NULL;
    }

    return newnode; // newnode becomes new root
}
void preOrder(struct node *root)
{
    if (root != NULL)
```

```
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);
    printf("Preorder traversal of the Existing  Splay tree is \n");
    preOrder(root);
    int n;
    printf("\nenter the number of element you want to enter:\t");
    scanf("%d", &n);
    int a[n];
    printf("\nenter the elements :\t");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
        root = insert(root, a[i]);
    }

    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Output :-

```
PS C:\Users\User\Desktop\c_program\DS_LAB> ./SPLAY_TREE
Preorder traversal of the Existing  Splay tree is
Preorder traversal of the modified Splay tree is
25 PS C:\Users\User\Desktop\c_program\DS_LAB> gcc SPLAY_TREE.c -o SPLAY_TREE.exe
PS C:\Users\User\Desktop\c_program\DS_LAB> ./SPLAY_TREE
Preorder traversal of the Existing  Splay tree is
100 50 40 30 20 200
enter the number of element you want to enter:  5

enter the elements :    1
2
3
4
5
Preorder traversal of the modified Splay tree is
5 4 3 2 1 20 50 30 40 100 200 PS C:\Users\User\Desktop\c_program\DS_LAB>
```

21. Write a program to implement topological sort

Code :-

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

int s[100], j, res[100];

void AdjacencyMatrix(int a[][100], int n) {

    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j <= n; j++) {
            a[i][j] = 0;
        }
    }
    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) {
            a[i][j] = rand() % 2;
            a[j][i] = 0;
        }
    }
}

void dfs(int u, int n, int a[][100]) {

    int v;
```

```
    s[u] = 1;
    for (v = 0; v < n - 1; v++) {
        if (a[u][v] == 1 && s[v] == 0) {
            dfs(v, n, a);
        }
    }
    j += 1;
    res[j] = u;
}

void topological_order(int n, int a[][100]) {

    int i, u;
    for (i = 0; i < n; i++) {
        s[i] = 0;
    }
    j = 0;
    for (u = 0; u < n; u++) {
        if (s[u] == 0) {
            dfs(u, n, a);
        }
    }
    return;
}
int main() {
    int a[100][100], n, i, j;

    printf("Enter number of vertices\n");    scanf("%d", &n);

    AdjacencyMatrix(a, n);
    printf("\t\tAdjacency Matrix of the graph\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("\t%d", a[i][j]);
        }
        printf("\n");
    }
    printf("\nTopological order:\n");

    topological_order(n, a);

    for (i = n; i >= 1; i--) {
        printf("-->%d", res[i]);
    }
    return 0;
}
```

Output:-

```
PS C:\Users\User\Desktop\c_program\DS_LAB> gcc topological.c -o topological.exe
PS C:\Users\User\Desktop\c_program\DS_LAB> ./topological
Enter number of vertices
4

                Adjacency Matrix of the graph
        0       0       0       0
        1       0       0       0
        1       0       0       0
        0       1       0       0

Topological order:
-->3-->2-->1-->0PS C:\Users\User\Desktop\c_program\DS_LAB>
```

22. Write a program to implement Dijkstra's algorithm

Code :-

```c
#include <stdio.h>
#define INFINITY 9999
#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
  int cost[MAX][MAX], distance[MAX], pred[MAX];
  int visited[MAX], count, mindistance, nextnode, i, j;

   for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (Graph[i][j] == 0)
        cost[i][j] = INFINITY;
      else
        cost[i][j] = Graph[i][j];

  for (i = 0; i < n; i++) {
    distance[i] = cost[start][i];
    pred[i] = start;
    visited[i] = 0;
  }

  distance[start] = 0;
  visited[start] = 1;
  count = 1;

  while (count < n - 1) {
    mindistance = INFINITY;

    for (i = 0; i < n; i++)
      if (distance[i] < mindistance && !visited[i]) {
```

```
        mindistance = distance[i];
        nextnode = i;
      }

    visited[nextnode] = 1;
    for (i = 0; i < n; i++)
      if (!visited[i])
        if (mindistance + cost[nextnode][i] < distance[i]) {
          distance[i] = mindistance + cost[nextnode][i];
          pred[i] = nextnode;
        }
    count++;
  }

  // Printing the distance
  for (i = 0; i < n; i++)
    if (i != start) {
      printf("\nDistance from source to %d: %d\n", i, distance[i]);
    }
}
int main() {
  int Graph[MAX][MAX], i, j, n, u;
  n = 7;

  Graph[0][0] = 0;
  Graph[0][1] = 0;
  Graph[0][2] = 1;
  Graph[0][3] = 2;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 0;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 3;
  Graph[1][6] = 0;

  Graph[2][0] = 1;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 1;
  Graph[2][4] = 3;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 2;
  Graph[3][1] = 0;
  Graph[3][2] = 1;
```

```
  Graph[3][3] = 0;
  Graph[3][4] = 0;
  Graph[3][5] = 0;
  Graph[3][6] = 1;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 3;
  Graph[4][3] = 0;
  Graph[4][4] = 0;
  Graph[4][5] = 2;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 3;
  Graph[5][2] = 0;
  Graph[5][3] = 0;
  Graph[5][4] = 2;
  Graph[5][5] = 0;
  Graph[5][6] = 1;

  Graph[6][0] = 0;
  Graph[6][1] = 0;

  Graph[6][2] = 0;
  Graph[6][3] = 1;
  Graph[6][4] = 0;
  Graph[6][5] = 1;
  Graph[6][6] = 0;

  u = 0;
  Dijkstra(Graph, n, u);

  return 0;
}
```

Output :-

```
stra.exe
PS C:\Users\User\Desktop\c_program\DS_LAB> ./dijkastra

Distance from source to 1: 3

Distance from source to 2: 1

Distance from source to 3: 2

Distance from source to 4: 4

Distance from source to 5: 4

Distance from source to 6: 3
PS C:\Users\User\Desktop\c_program\DS_LAB>
```

23. Write a program to implement Floyd Warshall algorithm

Code :-

```c
#include <stdio.h>
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];
 for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
```

```
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
              {2, 0, INF, 4},
              {INF, 1, 0, INF},
              {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

Output :-

```
PS C:\Users\User\Desktop\c_program\DS_LAB> gcc flod.c -o flod.exe
PS C:\Users\User\Desktop\c_program\DS_LAB> ./flod
   0   3   7   5
   2   0   6   4
   3   1   0   5
   5   3   2   0
PS C:\Users\User\Desktop\c_program\DS_LAB>
```

24. Write a program in C to implement Heap sort

Code :-

```
#include <stdio.h>
  void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
 }
  void heapify(int arr[], int n, int i) {
    int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
   if (left < n && arr[left] > arr[largest])
```

```c
    largest = left;
  if (right < n && arr[right] > arr[largest])
    largest = right;
  if (largest != i) {
    swap(&arr[i], &arr[largest]);
    heapify(arr, n, largest);
  }
}
 void heapSort(int arr[], int n) {
   for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
  for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);
  heapify(arr, i, 0);
  }
}
 void printArray(int arr[], int n) {
  for (int i = 0; i < n; ++i)
    printf("%d ", arr[i]);
  printf("\n");
}
 int main() {
  int arr[] = {1, 12, 9, 5, 6, 10};
  int n = sizeof(arr) / sizeof(arr[0]);

  heapSort(arr, n);

  printf("Sorted array is \n");
  printArray(arr, n);
}
```

Output :-

```
PS C:\Users\User\Desktop\c_program\DS_LAB> gcc heap.c -o heap.exe
PS C:\Users\User\Desktop\c_program\DS_LAB> ./heap
Sorted array is
1 5 6 9 10 12
PS C:\Users\User\Desktop\c_program\DS_LAB>
```

Ln 62, Col 4 Spaces: 4 UTF-8 CRLF C Win32