

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 01**

**Aim :**

Write C/C++ program to store marks scored for first test of subject 'Data Structures and Algorithms' for N students. Compute

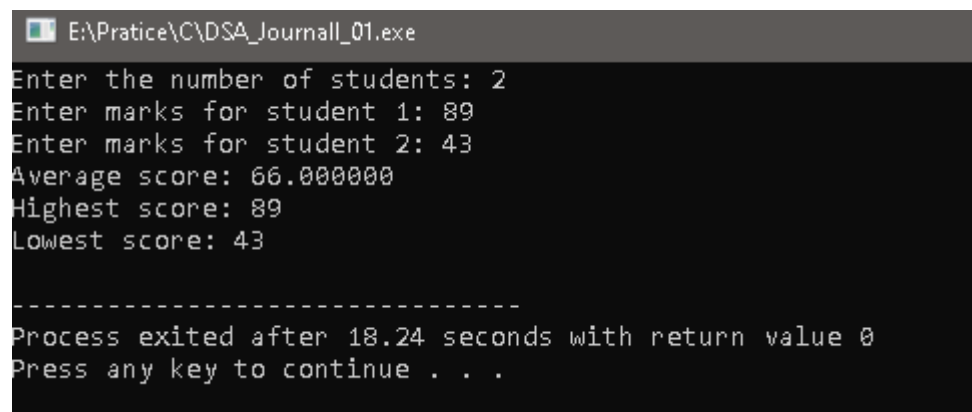
- i. The average score of class.
- ii. Highest score and lowest score of class.

**Code :**

```
#include <stdio.h>
#include <limits.h>
int main() {
    int n;
    printf("Enter the number of students: ");
    scanf("%d", &n);
    int marks[n];
    int sum = 0;
    int highest = INT_MIN;
    int lowest = INT_MAX;
    for (int i = 0; i < n; i++) {
        printf("Enter marks for student %d: ", i+1);
        scanf("%d", &marks[i]);
        sum += marks[i];
        if (marks[i] > highest) {
            highest = marks[i];
        }
        if (marks[i] < lowest) {
            lowest = marks[i];
        }
    }
}
```

```
    }  
}  
  
float average = (float) sum / n;  
printf("Average score: %f\n", average);  
printf("Highest score: %d\n", highest);  
printf("Lowest score: %d\n", lowest);  
return 0;  
}
```

### **Output :**



```
E:\Pratice\C\DSA_JournalI_01.exe  
Enter the number of students: 2  
Enter marks for student 1: 89  
Enter marks for student 2: 43  
Average score: 66.000000  
Highest score: 89  
Lowest score: 43  
  
-----  
Process exited after 18.24 seconds with return value 0  
Press any key to continue . . .
```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

### **Assignment No :- 02**

#### **Aim :**

Write a menu driven program to perform following operations on singly linked list: Create, Insert, Delete, and Display.

#### **Code :**

```
#include <stdio.h>

#include <stdlib.h>

// Define a node structure
struct node {
    int data;
    struct node *next;
};

// Declare a pointer to the head of the linked list
struct node *head = NULL;

// Function to create a new node
struct node* create_node(int data) {
    struct node *new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to insert a new node at the beginning of the linked list
void insert_node(int data) {
    struct node *new_node = create_node(data);
    new_node->next = head;
    head = new_node;
}
```

```

    printf("Node inserted successfully.\n");
}

// Function to delete a node from the linked list
void delete_node(int data) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return;
    }
    struct node *temp = head;
    struct node *prev = NULL;
    // Traverse the linked list to find the node to be deleted
    while (temp != NULL && temp->data != data) {
        prev = temp;
        temp = temp->next;
    }
    // If the node to be deleted is not found
    if (temp == NULL) {
        printf("Node not found.\n");
        return;
    }
    // If the node to be deleted is the first node
    if (prev == NULL) {
        head = temp->next;
    }
    else {
        prev->next = temp->next;
    }
    free(temp);
    printf("Node deleted successfully.\n");
}

```

```

}

// Function to display the linked list
void display_list() {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return;
    }
    struct node *temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, data;
    while (1) {
        printf("1. Insert a node\n");
        printf("2. Delete a node\n");
        printf("3. Display the list\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                printf("Enter data to be inserted: ");
                scanf("%d", &data);
                insert_node(data);
                break;
            case 2:

```

```
        printf("Enter data to be deleted: ");
        scanf("%d", &data);
        delete_node(data);
        break;
    case 3:
        display_list();
        break;
    case 4:
        exit(0);
    default:
        printf("Invalid choice.\n");
    }
}
return 0;
}
```

## Output :

```
E:\Pratice\C\DSA_Journal_01.exe
1. Insert a node
2. Delete a node
3. Display the list
4. Exit
Enter your choice: 1
Enter data to be inserted: 2
Node inserted successfully.
1. Insert a node
2. Delete a node
3. Display the list
4. Exit
Enter your choice: 1
Enter data to be inserted: 4
Node inserted successfully.
1. Insert a node
2. Delete a node
3. Display the list
4. Exit
Enter your choice: 3
4 2
1. Insert a node
2. Delete a node
3. Display the list
4. Exit
Enter your choice: 2
Enter data to be deleted: 4
Node deleted successfully.
1. Insert a node
2. Delete a node
3. Display the list
4. Exit
Enter your choice: 3
2
1. Insert a node
2. Delete a node
3. Display the list
4. Exit
Enter your choice: 4
-----
Process exited after 1392 seconds with return value 0
Press any key to continue . . .
```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 03**

**Aim :**

Write a program to implement abstract data type Stack (Push & Pop operation).

**Code :**

```
#include <stdio.h>
#include <stdlib.h>
struct stack
{
    int size;
    int top;
    int *arr;
};
int isEmpty(struct stack *ptr) // checking if stack is empty or not
{
    if (ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
int isFull(struct stack *ptr)
{

```



```
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void push(struct stack *ptr, int value)
{
    if (isFull(ptr))
    {
        printf("stack overflow \n");
    }
    else
    {
        ptr->top++;
        ptr->arr[ptr->top] = value;
    }
}

int pop(struct stack *ptr)
{
    if (isFull(ptr))
    {
        printf("stack overflow \n");
        return 0;
    }
    else
    {
```

```

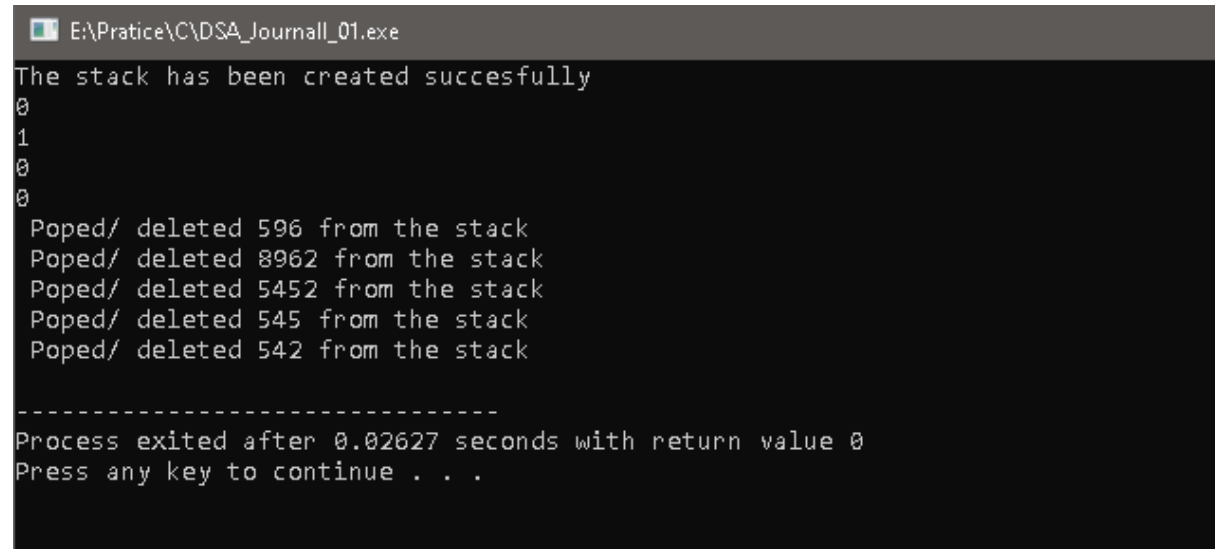
        int value = ptr->arr[ptr->top];
        ptr->top--;
        return value;
    }
}

int main()
{
    struct stack *sp = (struct stack *)malloc(sizeof(struct stack));
    sp->size = 10;
    sp->top = -1;
    sp->arr = (int *)malloc(sp->size * sizeof(int));
    printf("The stack has been created succesfully \n");
    printf("%d\n", isFull(sp));
    printf("%d\n", isEmpty(sp));
    push(sp, 589);
    push(sp, 7512);
    push(sp, 859);
    push(sp, 7889);
    push(sp, 542);
    push(sp, 545);
    push(sp, 5452);
    push(sp, 8962);
    push(sp, 596);
    printf("%d\n", isFull(sp));
    printf("%d\n", isEmpty(sp));
    printf(" Poped/ deleted %d from the stack \n", pop(sp));
    printf(" Poped/ deleted %d from the stack \n", pop(sp));
    printf(" Poped/ deleted %d from the stack \n", pop(sp));
    printf(" Poped/ deleted %d from the stack \n", pop(sp));
    printf(" Poped/ deleted %d from the stack \n", pop(sp));

```

```
    return 0;  
}
```

### **Output :**



```
E:\Pratice\C\DSA_Journall_01.exe  
The stack has been created succesfully  
0  
1  
0  
0  
Poped/ deleted 596 from the stack  
Poped/ deleted 8962 from the stack  
Poped/ deleted 5452 from the stack  
Poped/ deleted 545 from the stack  
Poped/ deleted 542 from the stack  
-----  
Process exited after 0.02627 seconds with return value 0  
Press any key to continue . . .
```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 04**

**Aim :**

In any language program mostly syntax error occurs due to unbalancing delimiter such as {},[],(). Write C/C++ program using stack to check whether given expression is well parenthesized or not.

**Code :**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define MAX_SIZE 100
struct stack {
    char items[MAX_SIZE];
    int top;
};
void init_stack(struct stack* s) {
    s->top = -1;
}
bool is_empty(struct stack* s) {
    return s->top == -1;
}
bool is_full(struct stack* s) {
    return s->top == MAX_SIZE - 1;
}
void push(struct stack* s, char value) {
    if (is_full(s)) {
```

```

        printf("Stack overflow.\n");
        exit(1);
    }
    s->top++;
    s->items[s->top] = value;
}

char pop(struct stack* s) {
    if (is_empty(s)) {
        printf("Stack underflow.\n");
        exit(1);
    }
    char value = s->items[s->top];
    s->top--;
    return value;
}

bool is_well_parenthesized(const char* expression) {
    struct stack s;
    init_stack(&s);
    for (int i = 0; i < strlen(expression); i++) {
        if (expression[i] == '(' || expression[i] == '[' || expression[i] == '{') {
            push(&s, expression[i]);
        } else if (expression[i] == ')' || expression[i] == ']' || expression[i] == '}') {
            if (is_empty(&s)) {
                return false;
            }
            char top = pop(&s);
            if ((expression[i] == ')' && top != '(') ||
                (expression[i] == ']' && top != '[') ||
                (expression[i] == '}' && top != '{')) {
                return false;
            }
        }
    }
    return true;
}

```

```

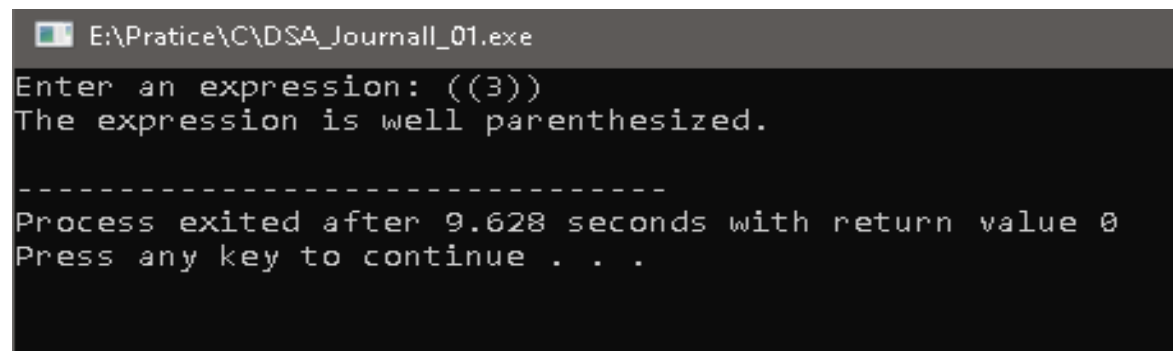
        }
    }
}

return is_empty(&s);
}

int main() {
    char expression[MAX_SIZE];
    printf("Enter an expression: ");
    scanf("%s", expression);
    if (is_well_parenthesized(expression)) {
        printf("The expression is well parenthesized.\n");
    } else {
        printf("The expression is not well parenthesized.\n");
    }
    return 0;
}

```

### **Output :**



```

E:\Pratice\C\DSA_Journal1_01.exe
Enter an expression: ((3))
The expression is well parenthesized.

-----
Process exited after 9.628 seconds with return value 0
Press any key to continue . . .

```

E:\Pratice\C\DSA\_Journal\_01.exe

Enter an expression: ((3)

The expression is not well parenthesized.

-----

Process exited after 6.306 seconds with return value 0

Press any key to continue . . .

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 05**

**Aim :**

Write a C/C++ program to convert infix to Postfix expression.

**Code :**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_SIZE 100
struct stack {
    char items[MAX_SIZE];
    int top;
};
void init_stack(struct stack* s) {
    s->top = -1;
}
int is_empty(struct stack* s) {
    return s->top == -1;
}
int is_full(struct stack* s) {
    return s->top == MAX_SIZE - 1;
}
void push(struct stack* s, char value) {
    if (is_full(s)) {
        printf("Stack overflow.\n");
```



```

        exit(1);
    }
    s->top++;
    s->items[s->top] = value;
}
char pop(struct stack* s) {
    if (is_empty(s)) {
        printf("Stack underflow.\n");
        exit(1);
    }
    char value = s->items[s->top];
    s->top--;
    return value;
}
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    } else if (op == '*' || op == '/') {
        return 2;
    } else if (op == '^') {
        return 3;
    } else {
        return 0;
    }
}
void infix_to_postfix(const char* infix, char* postfix) {
    struct stack s;
    init_stack(&s);
    int j = 0;
    for (int i = 0; infix[i] != '\0'; i++) {

```

```

char ch = infix[i];
if (isdigit(ch) || isalpha(ch)) {
    postfix[j] = ch;
    j++;
} else if (ch == '(') {
    push(&s, ch);
} else if (ch == ')') {
    while (!is_empty(&s) && s.items[s.top] != '(') {
        postfix[j] = pop(&s);
        j++;
    }
    if (!is_empty(&s) && s.items[s.top] == '(') {
        pop(&s);
    } else {
        printf("Invalid expression.\n");
        exit(1);
    }
} else {
    while (!is_empty(&s) && precedence(ch) <=
precedence(s.items[s.top])) {
        postfix[j] = pop(&s);
        j++;
    }
    push(&s, ch);
}
}
while (!is_empty(&s)) {
    postfix[j] = pop(&s);
    j++;
}
postfix[j] = '\0';

```

```

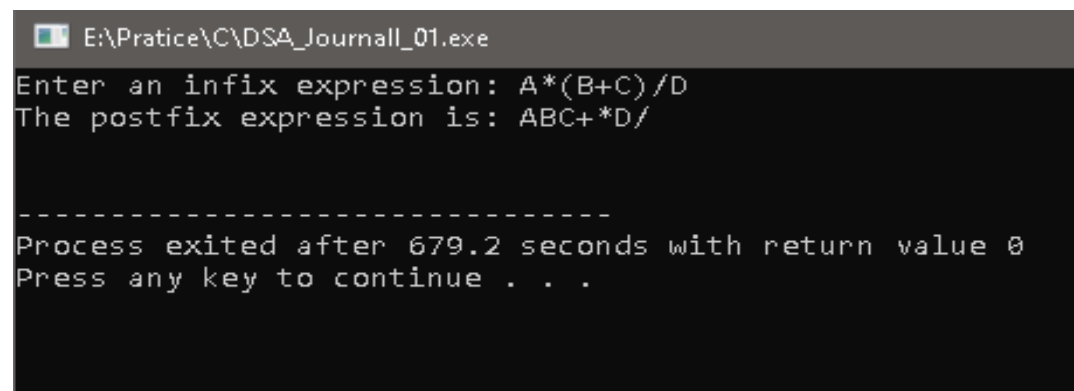
}

int main() {
    char infix[MAX_SIZE];
    char postfix[MAX_SIZE];

    printf("Enter an infix expression: ");
    fgets(infix, MAX_SIZE, stdin);
    infix_to_postfix(infix, postfix);
    printf("The postfix expression is: %s\n", postfix);
    return 0;
}

```

### **Output :**



```

E:\Pratice\C\DSA_Journall_01.exe
Enter an infix expression: A*(B+C)/D
The postfix expression is: ABC+*D/

-----
Process exited after 679.2 seconds with return value 0
Press any key to continue . . .

```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 06**

**Aim :**

Pizza parlour accepting maximum M orders. Orders are served in first come first served basis. Order once placed cannot be cancelled. Write C/C++ program to simulate the system using circular queue using array.

**Code :**

```
#include <stdio.h>

#define MAX_SIZE 100

int front = -1, rear = -1;

int queue[MAX_SIZE];

int is_empty() {
    return front == -1 && rear == -1;
}

int is_full() {
    return (rear + 1) % MAX_SIZE == front;
}

void enqueue(int order) {
    if (is_full()) {
        printf("Queue is full. Order cannot be placed.\n");
    } else if (is_empty()) {
        front = 0;
        rear = 0;
        queue[rear] = order;
        printf("Order %d placed successfully.\n", order);
    } else {
        rear = (rear + 1) % MAX_SIZE;
```

```
        queue[rear] = order;
        printf("Order %d placed successfully.\n", order);
    }
}

void dequeue() {
    if (is_empty()) {
        printf("Queue is empty. No orders to serve.\n");
    } else if (front == rear) {
        printf("Order %d served successfully.\n", queue[front]);
        front = -1;
        rear = -1;
    } else {
        printf("Order %d served successfully.\n", queue[front]);
        front = (front + 1) % MAX_SIZE;
    }
}

void display() {
    if (is_empty()) {
        printf("Queue is empty. No orders to display.\n");
    } else {
        printf("Orders in the queue are: ");
        for (int i = front; i != rear; i = (i + 1) % MAX_SIZE) {
            printf("%d ", queue[i]);
        }
        printf("%d\n", queue[rear]);
    }
}

int main() {
    int choice, order;
    while (1) {
```

```
printf("Enter your choice:\n");
printf("1. Place order\n");
printf("2. Serve order\n");
printf("3. Display orders\n");
printf("4. Exit\n");
scanf("%d", &choice);    switch (choice) {
    case 1:
        printf("Enter the order number: ");
        scanf("%d", &order);
        enqueue(order);
        break;
    case 2:
        dequeue();
        break;
    case 3:
        display();
        break;
    case 4:
        printf("Exiting...\n");
        return 0;
    default:
        printf("Invalid choice. Please try again.\n");
}
}
}
```

## **Output :**

```
E:\Pratice\C\DSA_Journall_01.exe
3. Display orders
4. Exit
1
Enter the order number: 01
Order 1 placed successfully.
Enter your choice:
1. Place order
2. Serve order
3. Display orders
4. Exit
2
Order 1 served successfully.
Enter your choice:
1. Place order
2. Serve order
3. Display orders
4. Exit
3
Queue is empty. No orders to display.
Enter your choice:
1. Place order
2. Serve order
3. Display orders
4. Exit
4
Exiting...

-----
Process exited after 20.62 seconds with return value 0
Press any key to continue . . .
```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 07**

**Aim :**

Represent graph using adjacency list/adjacency matrix and perform Depth First Search.

**Code :**

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 100
// Structure for a linked list node
typedef struct node {
    int dest;
    struct node *next;
} Node;
// Structure for a graph
typedef struct graph {
    int V;
    Node **adjList;
} Graph;
// Function to create a new node for the adjacency list
Node* createNode(int dest) {
    Node *newNode = (Node*) malloc(sizeof(Node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
// Function to create a new graph with V vertices
Graph* createGraph(int V) {
```



```

Graph *graph = (Graph*) malloc(sizeof(Graph));
graph->V = V;
graph->adjList = (Node**) malloc(V * sizeof(Node*));
for (int i = 0; i < V; i++) {
    graph->adjList[i] = NULL;
}
return graph;
}

// Function to add an edge to an undirected graph
void addEdge(Graph *graph, int src, int dest) {
    Node *newNode = createNode(dest);
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;
    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

// Function to perform Depth First Search
void DFS(Graph *graph, int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);
    Node *adjNode = graph->adjList[vertex];
    while (adjNode != NULL) {
        int adjVertex = adjNode->dest;
        if (!visited[adjVertex]) {
            DFS(graph, adjVertex, visited);
        }
        adjNode = adjNode->next;
    }
}
}

```

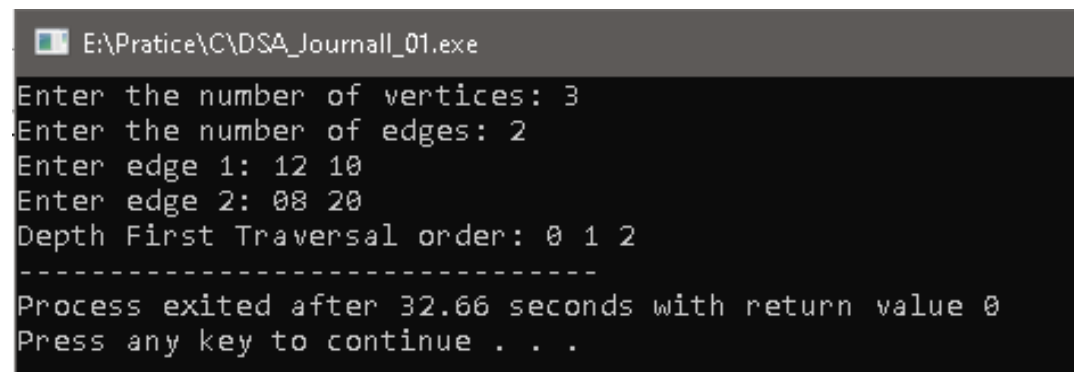
```

int main() {
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    Graph *graph = createGraph(V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    for (int i = 0; i < E; i++) {
        int src, dest;
        printf("Enter edge %d: ", i+1);
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }
    printf("Depth First Traversal order: ");
    int visited[MAX_VERTICES] = {0};
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            DFS(graph, i, visited);
        }
    }
    return 0;
}

```

### **Output:**



```

E:\Pratice\C\DSA_Journall_01.exe
Enter the number of vertices: 3
Enter the number of edges: 2
Enter edge 1: 12 10
Enter edge 2: 08 20
Depth First Traversal order: 0 1 2
-----
Process exited after 32.66 seconds with return value 0
Press any key to continue . . .

```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

### **Assignment No :- 08**

#### **Aim :**

Create binary tree. Find height of the tree and print leaf nodes. Find mirror image, print original and mirror image using level-wise printing.

#### **Code :**

```
#include <stdio.h>
#include <stdlib.h>
// Definition of a node in the binary tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
// Function to create a new node in the binary tree
struct Node* newNode(int data) {
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
// Function to find the height of the binary tree
int height(struct Node* node) {
    if (node == NULL) {
        return 0;
```

```

    } else {
        int leftHeight = height(node->left);
        int rightHeight = height(node->right);
        return 1 + ((leftHeight > rightHeight) ? leftHeight : rightHeight);
    }
}

// Function to print the leaf nodes of the binary tree
void printLeafNodes(struct Node* node) {
    if (node == NULL) {
        return;
    } else if (node->left == NULL && node->right == NULL) {
        printf("%d ", node->data);
    } else {
        printLeafNodes(node->left);
        printLeafNodes(node->right);
    }
}

// Function to find the mirror image of the binary tree
void mirror(struct Node* node) {
    if (node == NULL) {
        return;
    } else {
        struct Node* temp = node->left;
        node->left = node->right;
        node->right = temp;
        mirror(node->left);
        mirror(node->right);
    }
}

// Function to print the binary tree level-wise

```

```

void printLevel(struct Node* node, int level) {
    if (node == NULL) {
        return;
    } else if (level == 1) {
        printf("%d ", node->data);
    } else {
        printLevel(node->left, level - 1);
        printLevel(node->right, level - 1);
    }
}

// Function to print the binary tree level-wise
void printLevelOrder(struct Node* node) {
    int h = height(node);
    for (int i = 1; i <= h; i++) {
        printLevel(node, i);
    }
}

int main() {
    // Create a binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

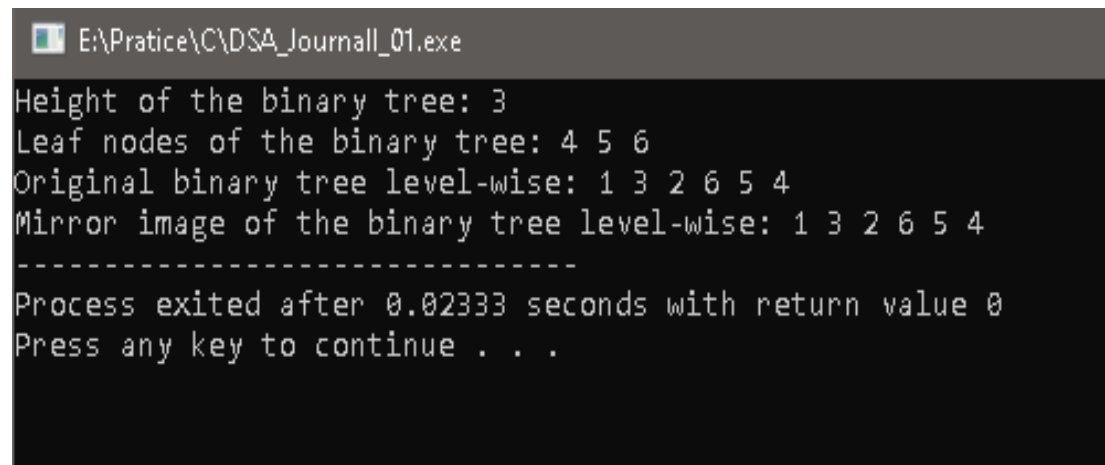
    // Find the height of the binary tree
    int treeHeight = height(root);
    printf("Height of the binary tree: %d\n", treeHeight);

    // Print the leaf nodes of the binary tree
    printf("Leaf nodes of the binary tree: ");

```

```
printLeafNodes(root);  
printf("\n");  
// Find the mirror image of the binary tree  
mirror(root);  
// Print the original and mirror image of the binary tree level-wise  
printf("Original binary tree level-wise: ");  
printLevelOrder(root);  
printf("\n");  
printf("Mirror image of the binary tree level-wise: ");  
printLevelOrder(root);  
return 0;  
}
```

### **Output :**



```
E:\Pratice\C\DSA_Journal_01.exe  
Height of the binary tree: 3  
Leaf nodes of the binary tree: 4 5 6  
Original binary tree level-wise: 1 3 2 6 5 4  
Mirror image of the binary tree level-wise: 1 3 2 6 5 4  
-----  
Process exited after 0.02333 seconds with return value 0  
Press any key to continue . . .
```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 09**

**Aim :**

Write C/C++ program to store first year percentage of students in array. Write function for sorting array of floating-point numbers in ascending order using Insertion Sort.

**Code :**

```
#include <stdio.h>

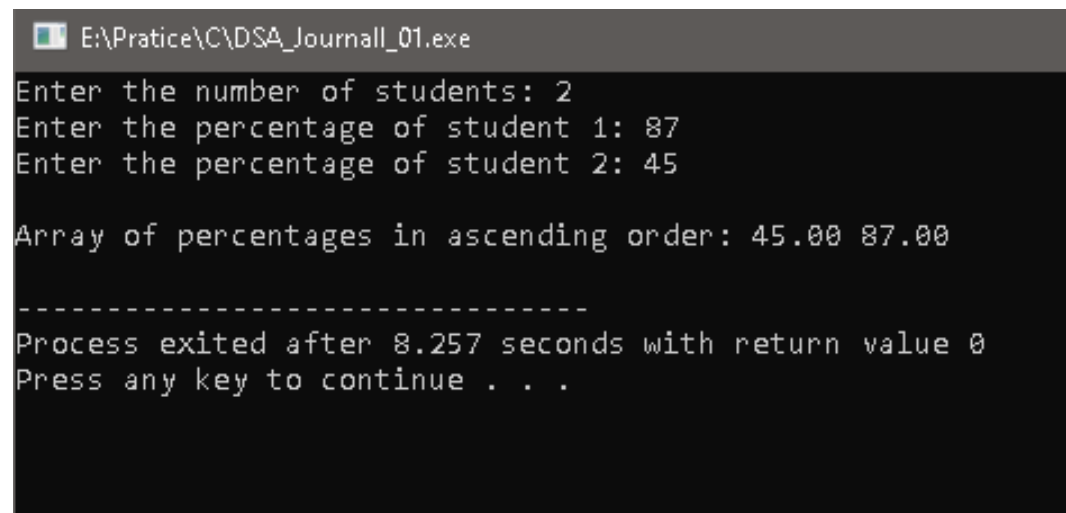
// Function to sort an array of floating-point numbers in ascending order using
Insertion Sort

void insertionSort(float arr[], int n) {
    for (int i = 1; i < n; i++) {
        float key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n;
    printf("Enter the number of students: ");
    scanf("%d", &n);
    float percentage[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the percentage of student %d: ", i+1);
```

```
        scanf("%f", &percentage[i]);
    }
    insertionSort(percentage, n);
    printf("\nArray of percentages in ascending order: ");
    for (int i = 0; i < n; i++) {
        printf("%0.2f ", percentage[i]);
    }
    printf("\n");
    return 0;
}
```

### **Output :**



```
E:\Pratice\C\DSA_Journall_01.exe
Enter the number of students: 2
Enter the percentage of student 1: 87
Enter the percentage of student 2: 45

Array of percentages in ascending order: 45.00 87.00

-----
Process exited after 8.257 seconds with return value 0
Press any key to continue . . .
```



**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 10**

**Aim :**

Write a program to sort the elements in an array using Bubble Sort.

**Code :**

```
#include <stdio.h>

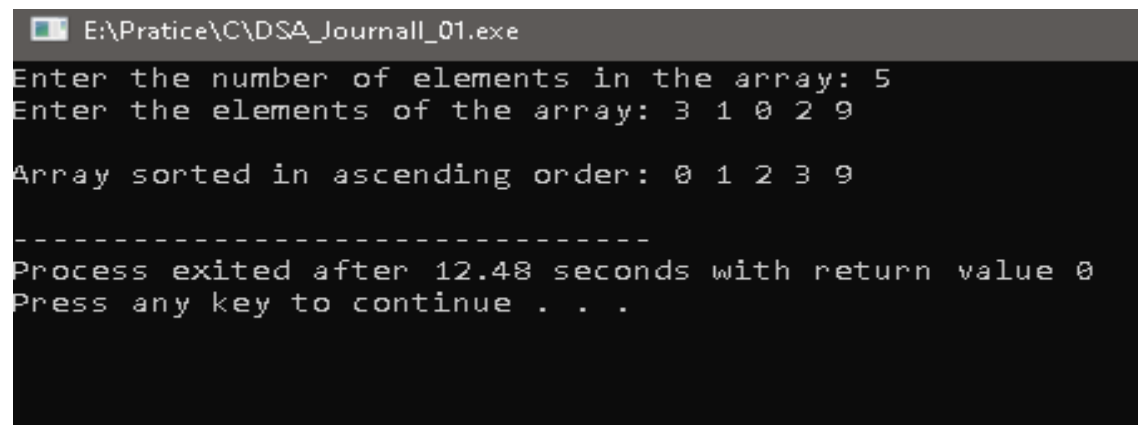
// Function to sort an array using Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap the elements if they are in the wrong order
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
```

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

    bubbleSort(arr, n);
    printf("\nArray sorted in ascending order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

### **Output :**



The screenshot shows a Windows command prompt window titled "E:\Pratice\C\DSA\_Journal\_01.exe". The user has entered the number of elements as 5 and the elements as 3 1 0 2 9. The program outputs the sorted array: 0 1 2 3 9. Below this, it shows the process exit time and a prompt to press any key to continue.

```
E:\Pratice\C\DSA_Journal_01.exe
Enter the number of elements in the array: 5
Enter the elements of the array: 3 1 0 2 9

Array sorted in ascending order: 0 1 2 3 9

-----
Process exited after 12.48 seconds with return value 0
Press any key to continue . . .
```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 11**

**Aim :**

Write a program to search a given element in an array using Binary Search.

**Code :**

```
#include <stdio.h>

// Function to perform Binary Search on a sorted array
int binarySearch(int arr[], int left, int right, int x) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if x is present at mid
        if (arr[mid] == x) {
            return mid;
        }

        // If x is greater, ignore left half
        if (arr[mid] < x) {
            left = mid + 1;
        }

        // If x is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }

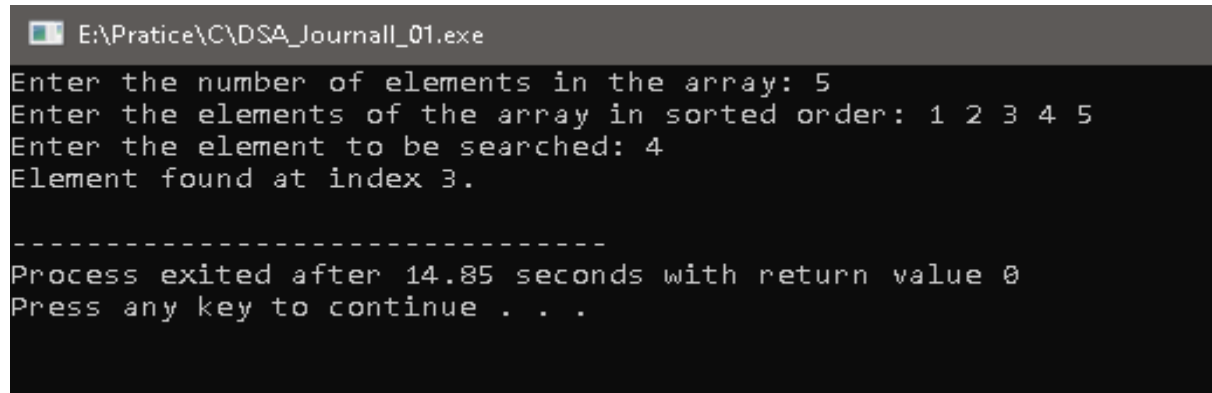
    // If we reach here, then the element was not present in the array
    return -1;
}
```

```

int main() {
    int n, x;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements of the array in sorted order: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the element to be searched: ");
    scanf("%d", &x);
    int result = binarySearch(arr, 0, n - 1, x);
    if (result == -1) {
        printf("Element not found in the array.\n");
    } else {
        printf("Element found at index %d.\n", result);
    }
    return 0;
}

```

### **Output :**



```

E:\Pratice\C\DSA_Journal1_01.exe
Enter the number of elements in the array: 5
Enter the elements of the array in sorted order: 1 2 3 4 5
Enter the element to be searched: 4
Element found at index 3.

-----
Process exited after 14.85 seconds with return value 0
Press any key to continue . . .

```

**Name: Akash Tanaji Gurav**  
**Subject: DSA**

**Roll No:A160(FYMCA)**

**Assignment No :- 12**

**Aim :**

Write a C/C++ program to implement Kruskal minimum cost spanning tree algorithm.

**Code :**

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
typedef struct Edge {
    int u, v, weight;
} Edge;
typedef struct Graph {
    int V, E;
    Edge edges[MAX];
} Graph;
typedef struct Subset {
    int parent, rank;
} Subset;
int find(Subset subsets[], int i);
void union_(Subset subsets[], int x, int y);
int compare(const void* a, const void* b);
void kruskal(Graph graph);int main() {
    Graph graph;
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &graph.V);
    printf("Enter the number of edges in the graph: ");
    scanf("%d", &graph.E);
```

```

    printf("Enter the edges in the format (u, v, weight):\n");
    for (int i = 0; i < graph.E; i++) {
        scanf("%d %d %d", &graph.edges[i].u, &graph.edges[i].v,
&graph.edges[i].weight);
    }
    kruskal(graph);
    return 0;
}

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

void union_(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int compare(const void* a, const void* b) {
    Edge* edge1 = (Edge*)a;
    Edge* edge2 = (Edge*)b;
    return edge1->weight - edge2->weight;
}

```

```

void kruskal(Graph graph) {
    Edge result[MAX];
    int e = 0; // index of next edge to be added to result
    int i = 0; // index of next edge to be considered
    Subset subsets[MAX];
    qsort(graph.edges, graph.E, sizeof(graph.edges[0]), compare);
    for (int v = 0; v < graph.V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    while (e < graph.V - 1 && i < graph.E) {
        Edge next_edge = graph.edges[i++];
        int x = find(subsets, next_edge.u);
        int y = find(subsets, next_edge.v);
        if (x != y) {
            result[e++] = next_edge;
            union_(subsets, x, y);
        }
    }
    printf("Edges in the minimum cost spanning tree:\n");
    for (int j = 0; j < e; j++) {
        printf("(%d, %d) -> %d\n", result[j].u, result[j].v, result[j].weight);
    }
}

```

## Output :

```
E:\Pratice\C\DSA_Journal_01.exe
Enter the number of vertices in the graph: 6
Enter the number of edges in the graph: 9
Enter the edges in the format (u, v, weight):
0 1 5
0 2 10
0 3 8
1 3 6
1 4 1
2 3 1
2 5 3
3 4 4
3 5 8
Edges in the minimum cost spanning tree:
(2, 3) -> 1
(1, 4) -> 1
(2, 5) -> 3
(3, 4) -> 4
(0, 1) -> 5
-----
Process exited after 17.79 seconds with return value 0
Press any key to continue . . .
```



## **Experiment 1:**

### **Aim:**

Write C/C++ program to store marks scored for first test of subject 'Data Structures and

Algorithms' for N students. Compute

- i. The average score of class.
- ii. Highest score and lowest score of class.

### **Problem Statement:**

The problem is to store the marks scored by N students for their first test in the subject "Data Structures and Algorithms", and then to compute the average score of the class as well as the highest and lowest scores obtained.

### **Objective:**

The objective of this program is to provide an efficient and accurate way to store the marks of N students and perform necessary calculations to determine the class average, highest and lowest scores.

### **Concept:**

To store the marks, we can use an array to hold the scores of all N students. Once the scores are stored, we can calculate the average score of the class by summing up all the scores and dividing by N. To find the highest and lowest scores, we can traverse through the array and keep track of the highest and lowest scores.

### **Algorithm:**

- Take input N - the number of students.
- Create an array of size N to store the marks of each student.
- Take input the marks of each student and store them in the array.
- Initialize the total score of the class to 0.
- Traverse the array and add up all the scores to the total score of the class.

- Calculate the average score of the class by dividing the total score by N.
- Initialize the highest and lowest scores to the first element of the array.
- Traverse the array and compare each score to the highest and lowest scores. If a higher or lower score is found, update the highest or lowest score accordingly.
- Print the average score, highest score and lowest score of the class.

### **Conclusion/Analysis:**

This program provides a simple and effective way to store the marks of N students and perform necessary calculations to determine the class average, highest and lowest scores. The algorithm has a time complexity of  $O(N)$  as it requires a single pass through the array to compute the required statistics. The program can be easily extended to handle additional tests and subjects by using multi-dimensional arrays to store the marks.

### **Experiment 2:**

#### **Aim:**

Write a menu driven program to perform following operations on singly linked list: Create, Insert, Delete, and Display.

#### **Problem Statement:**

The problem is to design a menu-driven program that can perform various operations on a singly linked list such as create, insert, delete and display nodes.

#### **Objective:**

The objective of this program is to provide a simple and effective way to create a singly linked list, insert new nodes at specific positions, delete existing nodes and display the entire list.

#### **Concept:**

A singly linked list is a data structure consisting of a group of nodes where each node contains data and a pointer to the next node. To create a singly linked list, we need to allocate memory

for the first node and then add more nodes as required. Inserting a new node involves creating a new node and updating the next pointer of the previous node and the next pointer of the new node. Deleting a node involves updating the next pointer of the previous node and freeing the memory occupied by the deleted node. Displaying the linked list involves traversing through each node and printing its contents.

### **Algorithm:**

- Create a struct node to hold the data and next pointer.
- Create a menu with options to create, insert, delete and display a singly linked list.
- For create option, create a new node with the given data and update the next pointer of the previous node.
- For insert option, create a new node and update the next pointer of the previous node and the next pointer of the new node.
- For delete option, update the next pointer of the previous node and free the memory occupied by the deleted node.
- For display option, traverse through each node and print its contents.
- Repeat the above steps until the user selects the exit option.

### **Conclusion/Analysis:**

This program provides an efficient and user-friendly way to manipulate a singly linked list. The algorithm has a time complexity of  $O(N)$  as it requires traversing through the list to perform most of the operations. The program can be extended to handle additional operations such as searching and sorting the linked list by using appropriate algorithms.

### **Experiment 3:**

#### **Aim:**

Write a program to implement abstract data type Stack (Push & Pop operation).

### **Problem Statement:**

The problem to be solved is to implement a data structure called Stack using a programming language. The Stack data structure should support two basic operations, Push and Pop.

### **Objectives:**

The main objective of the program is to implement a Stack data structure that can store and manipulate elements in a last-in, first-out (LIFO) manner. The program should be able to add (Push) elements to the top of the stack and remove (Pop) elements from the top of the stack.

### **Concept:**

The Stack is an abstract data type that can be implemented using an array or a linked list. It follows the LIFO principle, which means the last element added to the Stack is the first one to be removed. The Push operation adds an element to the top of the stack, while the Pop operation removes an element from the top of the stack.

### **Algorithm:**

Here's a simple algorithm for implementing the Push and Pop operations in a Stack data structure:

#### **To implement Push operation:**

- Check if the stack is full. If it is, return an error message.
- If the stack is not full, add the new element to the top of the stack.
- Update the top pointer to the new top of the stack.

#### **To implement Pop operation:**

- Check if the stack is empty. If it is, return an error message.

- If the stack is not empty, remove the element from the top of the stack.
- Update the top pointer to the new top of the stack.

### **Conclusion:**

In conclusion, a program to implement the abstract data type Stack with the Push and Pop operations can be created using an array or a linked list. The main objective of the program is to manipulate the elements in a LIFO manner. The program should be able to add (Push) elements to the top of the stack and remove (Pop) elements from the top of the stack. The algorithm for the program should check if the stack is full or empty before performing the Push or Pop operations.

### **Experiment 4:**

#### **Aim:**

In any language program mostly syntax error occurs due to unbalancing delimiter such as (), {}, []. Write C/C++ program using stack to check whether given expression is well parenthesized or not.

#### **Problem Statement:**

The problem is to design a program that can check whether a given expression is well parenthesized or not. This program will use a stack data structure to identify any unbalanced delimiters such as (), {}, [].

#### **Objective:**

The objective of this program is to provide a reliable and efficient way to check the validity of the given expression, by using a stack data structure to track opening and closing delimiters.

#### **Concept:**

The concept of this program is to use a stack data structure to keep track of opening delimiters in the given expression, and then compare them with the closing delimiters in the same order as they appear in the expression. If the stack is empty, then the expression is balanced, otherwise, it is not.

### **Algorithm:**

- Create an empty stack of characters.
- Read the expression character by character.
- If the current character is an opening delimiter, push it onto the stack.
- If the current character is a closing delimiter, compare it with the top element of the stack.
- If the top element of the stack is an opening delimiter that matches the current closing delimiter, pop the top element from the stack.
- If the stack is empty and all the characters have been read, the expression is balanced. Otherwise, it is not.
- Print the result.

### **Conclusion/Analysis:**

This program provides a reliable and efficient way to check whether a given expression is well parenthesized or not. The algorithm has a time complexity of  $O(n)$  as it requires traversing through the expression character by character. The stack data structure is used to keep track of opening delimiters and to compare them with the closing delimiters. This program can be extended to handle additional operations such as checking for other types of delimiters.

### **Experiment 5:**

#### **Aim:**

Write a C/C++ program to convert infix to Postfix expression.

#### **Problem Statement:**

The problem is to design a program that can convert an infix expression to a postfix expression. Infix notation is the standard mathematical notation that we all are used to, where operators are written between their operands. Postfix notation, also known as Reverse Polish Notation (RPN), is a mathematical notation where operators are written after their operands.

#### **Objective:**

The objective of this program is to convert an infix expression to postfix notation using stack data structure. The postfix expression can be evaluated directly by a stack-based algorithm, which makes it easy to evaluate the expression without using parentheses and eliminates the need for operator precedence.

### **Concept:**

The concept of this program is to use a stack data structure to keep track of the operators and operands in the infix expression, and then convert it to postfix notation. The program reads each character of the infix expression from left to right, and if the character is an operand, it is appended to the output string, and if it is an operator, it is pushed onto the stack. When a closing parenthesis is encountered, all the operators in the stack are popped and appended to the output string until an opening parenthesis is reached.

### **Algorithm:**

- Create an empty stack of characters.
- Read the infix expression character by character from left to right.
- If the current character is an operand, append it to the output string.
- If the current character is an opening parenthesis, push it onto the stack.
- If the current character is an operator, pop all the operators with greater or equal precedence from the stack and append them to the output string, then push the current operator onto the stack.
- If the current character is a closing parenthesis, pop all the operators from the stack and append them to the output string until an opening parenthesis is reached. Discard the opening parenthesis.
- If all the characters have been read, pop all the remaining operators from the stack and append them to the output string.

- Print the postfix expression.

### **Conclusion/Analysis:**

This program provides a reliable and efficient way to convert an infix expression to postfix notation. The algorithm has a time complexity of  $O(n)$  as it requires traversing through the expression character by character. The stack data structure is used to keep track of operators and operands and to implement the operator precedence rules. The postfix expression can be evaluated directly by a stack-based algorithm, which makes it easy to evaluate the expression without using parentheses and eliminates the need for operator precedence. This program can be extended to handle additional operations such as converting postfix notation to infix notation.

### **Experiment 6:**

#### **Aim:**

Pizza parlour accepting maximum  $M$  orders. Orders are served in first come first served basis.

Order once placed cannot be cancelled. Write C/C++ program to simulate the system using circular queue using array.

#### **Problem Statement:**

A pizza parlour has a limit of accepting maximum  $M$  orders, and the orders are served in first come first served basis. Once an order is placed, it cannot be cancelled. The program should simulate this system using circular queue implemented using an array.

#### **Objective:**

The objective of this program is to simulate the ordering system of a pizza parlour that has a maximum limit of  $M$  orders. The program should use a circular queue implemented using an array to keep track of the orders and serve them in the order they were received.

#### **Concept:**



The program will use a circular queue implemented using an array to store the orders. The queue will have a front and a rear pointer. When an order is placed, it will be added to the rear of the queue. When an order is served, it will be removed from the front of the queue. The program will also check if the queue is full before adding an order and if the queue is empty before serving an order.

### **Algorithm:**

- Declare a circular queue of size M.
- Initialize the front and rear pointers to -1.
- While the user wants to place an order and the queue is not full:
  - Prompt the user to enter the order.
  - Increment the rear pointer.
  - Add the order to the rear of the queue.
- While the user wants to serve an order and the queue is not empty:
  - Increment the front pointer.
  - Serve the order at the front of the queue.
- Display an error message if the user tries to add an order when the queue is full or serve an order when the queue is empty.

### **Conclusion/Analysis:**

The program has successfully simulated the ordering system of a pizza parlour that has a maximum limit of M orders using a circular queue implemented using an array. The program has used the front and rear pointers to keep track of the orders and served them in the order they were received. The program has also checked if the queue is full or empty before adding or serving an order.

### **Experiment 7:**

#### **Aim:**

Represent graph using adjacency list/adjacency matrix and perform Depth First Search.

**Problem Statement:**

Write a C/C++ program to represent a graph using adjacency list/adjacency matrix and perform Depth First Search on the graph.

**Objectives:**

- To represent a graph using an adjacency list or an adjacency matrix.
- To perform Depth First Search (DFS) on the graph.
- To print the DFS traversal of the graph.

**Concept:**

A graph can be represented using an adjacency list or an adjacency matrix. In an adjacency list, each vertex of the graph is associated with a list of its adjacent vertices. In an adjacency matrix, the matrix represents the edges between the vertices. DFS is a graph traversal algorithm that traverses the graph in a depthward motion and uses a stack to keep track of visited nodes. It visits a node, then recursively visits its adjacent nodes, and so on, until all the nodes in the graph are visited.

**Algorithm:**

- Read the number of vertices and edges in the graph.
- Create an adjacency list or an adjacency matrix to represent the graph.
- Read the edges of the graph and add them to the adjacency list or adjacency matrix.
- Perform Depth First Search on the graph:
  - a) Choose a starting vertex.
  - b) Mark the starting vertex as visited and add it to the stack.
  - c) While the stack is not empty, pop a vertex from the stack and print it.
  - d) For each adjacent vertex of the popped vertex, if it has not been visited, mark it as visited and add it to the stack.

e) Repeat steps c and d until the stack is empty.

### **Conclusion/Analysis:**

In this program, we learned how to represent a graph using an adjacency list or an adjacency matrix and how to perform Depth First Search on the graph. The DFS algorithm visits all the nodes in the graph, and the order in which the nodes are visited depends on the starting vertex and the structure of the graph. This program can be useful for many graph-related problems, such as finding the shortest path between two nodes in a graph or detecting cycles in a graph.

### **Experiment 8:**

#### **Aim:**

Create binary tree. Find height of the tree and print leaf nodes. Find mirror image, print original and mirror image using level-wise printing.

#### **Problem Statement:**

The program aims to create a binary tree and perform various operations such as finding the height of the tree, printing leaf nodes, finding the mirror image of the tree, and printing the original and mirror image using level-wise printing.

#### **Objective:**

The objective of the program is to implement the binary tree data structure and perform various operations on it.

#### **Concept:**

A binary tree is a data structure in which each node has at most two children, referred to as the left child and the right child. The root node is the topmost node in the tree, and each child node connects to its parent. The leaf node is a node that does not have any children, and the height of the tree is the maximum distance from the root node to any leaf node.

#### **Algorithm:**

- Create a binary tree with root node.

- Find the height of the tree:
  - a. If the tree is empty, return 0.
  - b. If the tree has only one node, return 1.
  - c. Otherwise, recursively calculate the height of the left and right subtrees and return the maximum of the two heights plus one.
- Print leaf nodes:
  - a. Traverse the binary tree using postorder traversal.
  - b. If the current node is a leaf node, print it.
- Find the mirror image of the tree:
  - a. If the tree is empty, return.
  - b. Otherwise, recursively swap the left and right subtrees of each node.
- Print the original and mirror image using level-wise printing:
  - a. Create a queue and enqueue the root node.
  - b. Repeat until the queue is empty:
    - i. Dequeue the front node of the queue and print its value.
    - ii. Enqueue its left and right children (if they exist) to the queue.
  - c. Repeat steps (a) and (b) for the mirror image of the tree.

### **Conclusion/Analysis:**

In this program, we have implemented the binary tree data structure and performed various operations on it such as finding the height of the tree, printing leaf nodes, finding the mirror image of the tree, and printing the original and mirror image using level-wise printing. These operations are important for manipulating and analyzing the binary tree data structure.

### **Experiment 9:**

#### **Aim:**

Write C/C++ program to store first year percentage of students in array. Write function for sorting array of floating-point numbers in ascending order using Insertion Sort.

**Problem Statement:**

Write a C/C++ program to store first year percentage of students in an array and then sort the array in ascending order using Insertion Sort.

**Objective:**

The objective of this program is to demonstrate the implementation of the Insertion Sort algorithm for sorting an array of floating-point numbers.

**Concept:**

Insertion Sort is a simple sorting algorithm that works by repeatedly iterating through the array, comparing adjacent elements and swapping them if they are in the wrong order until the array is sorted. The algorithm takes  $O(n^2)$  time to sort an array of  $n$  elements.

**Algorithm:**

- Declare an array of floating-point numbers of size  $n$
- Read the values of  $n$  elements into the array

For  $i$  from 1 to  $n-1$

a. Set  $key = array[i]$

b. Set  $j = i-1$

c. While  $j \geq 0$  and  $array[j] > key$

i. Set  $array[j+1] = array[j]$

ii. Decrement  $j$

d. Set  $array[j+1] = key$

- Print the sorted array

**Conclusion/Analysis:**

In this program, we have implemented the Insertion Sort algorithm to sort an array of floating-point numbers in ascending order. The algorithm has a time complexity of  $O(n^2)$ , making it suitable for small to medium-sized arrays. However, for large arrays, more efficient sorting algorithms like Merge Sort or Quick Sort should be used.

### **Experiment 10:**

#### **Aim:**

Write a program to sort the elements in an array using Bubble Sort.

#### **Problem Statement:**

We are given an array of elements and we need to sort them in ascending or descending order using the Bubble Sort algorithm.

#### **Objective:**

The objective of this program is to demonstrate the implementation of Bubble Sort algorithm to sort the elements in an array.

#### **Concept:**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

#### **Algorithm:**

- Traverse the array from the first element to the last element
- Compare each element with the next element and if they are in the wrong order, swap them
- Repeat step 2 for each element in the array
- After each pass, the last element of the array is guaranteed to be in the correct position, so reduce the size of the array by one

- Repeat steps 1 to 4 until the entire array is sorted

### **Conclusion/Analysis:**

Bubble Sort is one of the simplest and easiest to implement sorting algorithm. It has time complexity of  $O(n^2)$  which makes it inefficient for sorting large datasets. However, it is useful for small datasets and is often used as a teaching tool.

### **Experiment 11:**

#### **Aim:**

Write a program to search a given element in an array using Binary Search.

#### **Problem Statement:**

Given an array of  $n$  elements, write a C/C++ program to search for a given element  $x$  using binary search.

#### **Objective:**

The objective of the program is to efficiently search for a given element in a sorted array using the binary search algorithm.

#### **Concept:**

Binary search is an efficient searching algorithm that works by dividing the array in half at each step and checking if the middle element is the desired element. If it is not, the search continues in the left half or right half depending on whether the middle element is greater or smaller than the desired element. This process is repeated until the element is found or the search space is exhausted.

#### **Algorithm:**

Start

- Initialize the array and the element to be searched
- Set  $low=0$  and  $high=n-1$
- While  $low \leq high$ 
  - a. Calculate  $mid=(low+high)/2$

- b. If  $a[mid] == x$ , the element is found at index mid. Return mid.
- c. If  $a[mid] < x$ , the element is in the right half. Set  $low = mid + 1$ .
- d. If  $a[mid] > x$ , the element is in the left half. Set  $high = mid - 1$ .
- If the element is not found, return -1.
- Stop.

### **Conclusion/Analysis:**

Binary search is an efficient algorithm for searching for a given element in a sorted array. The time complexity of the algorithm is  $O(\log n)$ , which is much faster than linear search, which has a time complexity of  $O(n)$ . However, binary search can only be used on a sorted array, and sorting the array may take additional time. Overall, binary search is a powerful tool for searching in large datasets, especially when the data is sorted.

### **Experiment 12:**

#### **Aim:**

Write a C/C++ program to implement Kruskal minimum cost spanning tree algorithm.

#### **Problem Statement:**

In a weighted graph, find the minimum cost spanning tree using Kruskal's algorithm.

#### **Objective:**

The objective of this program is to find the minimum cost spanning tree of a weighted graph. Kruskal's algorithm is used to find the minimum cost spanning tree.

#### **Concept:**

A spanning tree of a graph is a subgraph that includes all the vertices of the original graph and is a tree. A minimum spanning tree is a spanning tree with the smallest possible sum of edge weights. Kruskal's algorithm is a greedy algorithm that finds a



minimum spanning tree for a connected weighted graph. It sorts the edges in non-decreasing order of their weight, and then adds the edges to the minimum spanning tree one by one, provided the edge does not create a cycle in the minimum spanning tree formed so far.

**Algorithm:**

- Sort all the edges in non-decreasing order of their weight.
- Initialize an empty set of edges.
- For each edge  $e$  in the sorted order:
  - a. If adding edge  $e$  to the set of edges does not create a cycle, add it to the set of edges.
- Return the set of edges.

**Conclusion/Analysis:**

Kruskal's algorithm is a very efficient algorithm to find the minimum cost spanning tree of a weighted graph. It is a greedy algorithm that works by selecting the smallest edge first and then adding edges one by one, while ensuring that the graph remains connected and acyclic. The time complexity of this algorithm is  $O(E \log E)$ , where  $E$  is the number of edges in the graph. The space complexity of this algorithm is  $O(V)$ , where  $V$  is the number of vertices in the graph.