SRI CHANDRASEKHARENDRA SARASWATHI VISWA MAHAVIDYALAYA

(UNIVERSITY ESTABLISHED UNDER SECTION 3 OF UGC ACT 1956)
ENATHUR, KANCHIPURAM – 631 561

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Name : S.Reshmitha

Reg. No : 11249A339

Class : II B.E. (CSE)

Course Code: BCSF183P80

Course Name: Python programming

SRI CHANDRASEKHARENDRA SARASWATHI VISWA MAHAVIDYALAYA

(UNIVERSITY ESTABLISHED UNDER SECTION 3 OF UGC ACT 1956)
ENATHUR, KANCHIPURAM – 631 561

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this is the bonafide record of work done by **Mr/Ms.** S.Reshmitha,

with **Reg. No** 11249A339 of **II-B.E.(CSE)** during the academic year 2025 – 2026.

Station: ENNATHUR

Date:

Staff-in-charge **Head of the Department**

Submitted for the Practical examination held on________________________.

✅ 1. LINEAR SEARCH

Aim

To search for an element in an array using Linear Search.

Algorithm

1. Start

2. Read n and the array elements

3. Read key to be searched

4. Traverse array from i = 0 to n-1

5. If a[i] == key, print position and stop

6. If not found, print “Not Found”

7. End

C Program

```
#include <stdio.h>

int main() {
    int a[20], n, key, i;
    printf("Enter size: ");
    scanf("%d", &n);

    printf("Enter elements: ");
    for(i=0; i<n; i++)
```

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

    printf("Enter key: ");
    scanf("%d", &key);

    for(i=0; i<n; i++) {
        if(a[i] == key) {
            printf("Element found at position %d\n", i+1);
            return 0;
        }
    }

    printf("Element not found\n");
    return 0;
}
```

Sample Output

Enter size: 5
Enter elements: 4 9 2 7 1
Enter key: 7
Element found at position 4

✅ 2. BINARY SEARCH

Aim

To search an element in a sorted array using Binary Search.

Algorithm

1. Start

2. Read n & sorted array

3. Read key

4. Set low = 0, high = n-1

5. Repeat until low ≤ high

mid = (low + high)/2

If a[mid] == key → Found

If key < a[mid] → search left (high = mid -1)

Else → search right (low = mid + 1)

6. If not found → print “Not Found”

7. End

C Program

```
#include <stdio.h>

int main() {
    int a[20], n, key, low, high, mid;

    printf("Enter size: ");
    scanf("%d", &n);

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

    printf("Enter key: ");
    scanf("%d", &key);

    low = 0;
    high = n - 1;

    while(low <= high) {
        mid = (low + high) / 2;
```

```
        if(a[mid] == key) {
            printf("Element found at position %d\n", mid+1);
            return 0;
        }
        else if(key < a[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }

    printf("Element not found\n");
}
```

Sample Output

```
Enter sorted elements: 1 3 5 7 9
Enter key: 7
Element found at position 4
```

✅ 3. IMPLEMENTATION OF STACK

Aim

To implement stack operations using array.

Algorithm

Operations:

Push

Pop

Display

C Program

```
#include <stdio.h>
#define MAX 10
```

```
int stack[MAX], top = -1;

void push(int x) {
    if(top == MAX-1)
        printf("Stack Overflow\n");
    else
        stack[++top] = x;
}

void pop() {
    if(top == -1)
        printf("Stack Underflow\n");
    else
        printf("Popped: %d\n", stack[top--]);
}

void display() {
    if(top == -1)
        printf("Stack Empty\n");
    else {
        printf("Stack: ");
        for(int i=0; i<=top; i++)
            printf("%d ", stack[i]);
        printf("\n");
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
}
```

Sample Output

```
Stack: 10 20 30
Popped: 30
Stack: 10 20
```

✅ 4. APPLICATION OF STACK (INFIX TO POSTFIX)

Aim

To convert an infix expression to postfix using stack.

Algorithm (Simple)

1. Scan infix from left
2. If operand → add to postfix
3. If operator → push based on precedence
4. If (→ push
5. If) → pop till (
6. Pop remaining operators

C Program

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

char stack[20];
int top = -1;

void push(char x) { stack[++top] = x; }
char pop() { return stack[top--]; }
int precedence(char x) {
    if(x=='+' || x=='-') return 1;
    if(x=='*' || x=='/') return 2;
    return 0;
}
```

```
int main() {
    char infix[20], postfix[20], ch;
    int i=0, j=0;

    printf("Enter expression: ");
    scanf("%s", infix);

    while((ch = infix[i++]) != '\0') {
        if(isalnum(ch))
            postfix[j++] = ch;
        else if(ch == '(')
            push(ch);
        else if(ch == ')') {
            while(stack[top] != '(')
                postfix[j++] = pop();
            pop();
        }
        else {
            while(top != -1 && precedence(stack[top]) >= precedence(ch))
                postfix[j++] = pop();
            push(ch);
        }
    }

    while(top != -1)
        postfix[j++] = pop();

    postfix[j] = '\0';
    printf("Postfix: %s\n", postfix);
}
```

Output

Enter expression: A*(B+C)
Postfix: ABC+*

✅ 5. IMPLEMENTATION OF QUEUE (Array)

Aim

To implement insertion and deletion operations of a Queue using arrays.

Algorithm

Enqueue(x):

1. If rear == MAX-1 → Overflow

2. Else rear++

3. queue[rear] = x

4. If queue was empty → front = 0

Dequeue():

1. If front == -1 → Underflow

2. Print and remove queue[front]

3. If front == rear → set both to -1 (queue empty)

4. Else front++

C Program

```
#include <stdio.h>
#define MAX 10

int queue[MAX], front = -1, rear = -1;
```

```
void enqueue(int x) {
    if(rear == MAX-1)
        printf("Queue Overflow\n");
    else {
        if(front == -1) front = 0;
        queue[++rear] = x;
    }
}

void dequeue() {
    if(front == -1)
        printf("Queue Underflow\n");
    else {
        printf("Deleted: %d\n", queue[front]);
        if(front == rear)
            front = rear = -1;
        else
            front++;
    }
}

void display() {
    if(front == -1)
        printf("Queue Empty\n");
    else {
        printf("Queue: ");
        for(int i=front; i<=rear; i++)
            printf("%d ", queue[i]);
        printf("\n");
    }
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
}
```

Sample Output

Queue: 10 20 30
Deleted: 10
Queue: 20 30

✅ 6. IMPLEMENTATION OF SINGLY LINKED LIST (SLL)

Aim

To perform insertion and deletion operations on a Singly Linked List.

Algorithm

Insert at end:

1. Create new node

2. If list empty → head = new node

3. Else traverse to last node and link new node

Delete from beginning:

1. If empty → print message

2. Else set head = head->next

C Program

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

struct node {
    int data;
    struct node *next;
```

```
};

struct node *head = NULL;

void insertEnd(int x) {
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = x;
    newNode->next = NULL;

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

void deleteBegin() {
    if(head == NULL)
        printf("List Empty\n");
    else {
        struct node *temp = head;
        head = head->next;
        free(temp);
        printf("Deleted from beginning.\n");
    }
}

void display() {
    struct node *temp = head;
    printf("List: ");
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    insertEnd(10);
    insertEnd(20);
    insertEnd(30);
```

```
    display();
    deleteBegin();
    display();
}
```

Sample Output

List: 10 20 30
Deleted from beginning.
List: 20 30

✅ 7. IMPLEMENTATION OF DOUBLY LINKED LIST (DLL)

Aim

To implement insertion at end and deletion at beginning of a Doubly Linked List.

Algorithm

Insert End:

1. Create new node

2. If list empty → head = new node

3. Otherwise traverse and attach at end

Delete Beginning:

1. If empty → stop

2. Move head to next node

3. Free old head

C Program

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

struct node {
    int data;
    struct node *prev, *next;
};

struct node *head = NULL;

void insertEnd(int x) {
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = x;
    newNode->next = NULL;

    if(head == NULL) {
        newNode->prev = NULL;
        head = newNode;
    }
    else {
        struct node *temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void deleteBegin() {
    if(head == NULL)
        printf("List empty\n");
    else {
        struct node *temp = head;
        head = head->next;
        if(head != NULL)
            head->prev = NULL;
        free(temp);
        printf("Deleted from beginning\n");
    }
}
```

```
void display() {
    struct node *temp = head;
    printf("DLL: ");
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    insertEnd(10);
    insertEnd(20);
    insertEnd(30);
    display();
    deleteBegin();
    display();
}
```

Sample Output

DLL: 10 20 30
Deleted from beginning
DLL: 20 30

✅ 8. BINARY TREE TRAVERSALS (Inorder, Preorder, Postorder)

Aim

To perform inorder, preorder, and postorder traversal of a binary tree.

Algorithm

Inorder: Left → Root → Right

Preorder: Root → Left → Right

Postorder: Left → Right → Root

C Program

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

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

struct node* create(int x) {
    struct node *temp = malloc(sizeof(struct node));
    temp->data = x;
    temp->left = temp->right = NULL;
    return temp;
}

void inorder(struct node *root) {
    if(root != NULL) !
```

OUTPUT
Inorder: 2 1 3
Preorder: 1 2 3
Postorder: 2 3 1

✅ 9. GRAPH SEARCH – BFS and DFS

Aim

To perform BFS and DFS on a graph using adjacency matrix.

Algorithm

BFS:

1. Use queue

2. Visit start node

3. Add its neighbors to queue

4. Continue

DFS:

1. Use stack/recursion

2. Visit node

3. Go deeper to neighbors

C Program

```
#include <stdio.h>

int visited[10], a[10][10], n;

void dfs(int v) {
    visited[v] = 1;
    printf("%d ", v);

    for(int i=1; i<=n; i++)
        if(a[v][i] == 1 && !visited[i])
            dfs(i);
}

void bfs(int v) {
    int q[10], front=0, rear=0;

    visited[v] = 1;
    q[rear] = v;

    while(front <= rear) {
        int node = q[front++];
        printf("%d ", node);

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

```
            if(a[node][i]==1 && !visited[i]) {
                visited[i] = 1;
                q[++rear] = i;
            }
        }
    }
}

int main() {
    int i, j, start;
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d", &a[i][j]);

    printf("Enter start node: ");
    scanf("%d", &start);

    printf("DFS: ");
    for(i=1;i<=n;i++) visited[i]=0;
    dfs(start);

    printf("\nBFS: ");
    for(i=1;i<=n;i++) visited[i]=0;
    bfs(start);
}
```

Sample Output

DFS: 1 2 4 3
BFS: 1 2 3 4

✅ 10. IMPLEMENTATION OF HASHING (Linear Probing)

Aim

To implement hashing using linear probing for collision resolution.

Algorithm

1. Initialize hash table with -1

2. Compute index = key % size

3. If occupied, move to next index (i+1 % size)

4. Insert key

C Program

```
#include <stdio.h>

int main() {
    int size = 10, hash[10], key, index, i;

    for(i=0;i<size;i++)
        hash[i] = -1;

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

    for(i=0;i<n;i++) {
        printf("Enter key: ");
        scanf("%d", &key);

        index = key % size;

        while(hash[index] != -1)
            index = (index + 1) % size;

        hash[index] = key;
    }

    printf("Hash Table:\n");
    for(i=0;i<size;i++)
        printf("%d ", hash[i]);
```

```
}
```

Sample Output

```
Enter keys: 10 21 32 43
Hash Table:
10 21 32 43 -1 -1 -1 -1 -1 -1
```