DEPARTMENT OF ECE

U23CSP33 C PROGRAMMING AND DATA STRUCTURES LAB

EXP. NO.	DATE	NAME OF THE EXPERIMENT	PAGE NO.	STAFF SIGN
1.		Practice of C programming using statements, expressions, decision making and iterative statements		
2.		Practice of C programming using Functions and Arrays		
3.		Implement C programs using Pointers and Structures		
4.		Implement C programs using Files		
5.		Development of real time C applications		
6.		Array implementation of List ADT		
7.		Array implementation of Stack and Queue ADTs		
8.		Linked list implementation of List, Stack and Queue ADTs		
9.		Applications of List, Stack and Queue ADTs		
10.		Implementation of Binary Trees and operations of Binary Trees		
11.		Implementation of Binary Search Trees		
12.		Implementation of searching techniques		
13.		Implementation of Sorting algorithms : Insertion Sort, Quick Sort, Merge Sort		
14.		Implementation of Hashing – any two collision techniques		

1. Practice of C Programming using statements expression, Decision making iterative Statement

Program:

```
#include <stdio.h>
int main() {
int num, i;
printf ("Enter a number: ");
scanf("%d", &num);
if (num % 2 == 0) {
printf("%d is Even\n", num);
  } else {
printf("%d is Odd\n", num);
  }
printf("Multiplication table using for loop:\n");
for (i = 1; i <= 10; i++) {
printf("%d x %d = %d\n", num, i, num * i);
  }
printf("\nMultiplication table using while loop:\n");
  i = 1;
while (i <= 10) {
printf("%d x %d = %d\n", num, i, num * i);
i++;
  }
printf("\nMultiplication table using do-while loop:\n");
  i = 1;
do {
printf("%d x %d = %d\n", num, i, num * i);
i++;
  } while (i <= 10);

return 0;
}
```

Output:

```
Enter a number: 5
5 is Odd
Multiplication table using for loop:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

2. Practice of C program using functions and array

Program:

```
include<stdio.h>
int findSum(int arr[], int size) {
int sum = 0;
for(int i = 0; i < size; i++) {
sum += arr[i];
  }
return sum;
}
int main() {
int numbers[100], n, total;
printf("Enter number of elements: ");
scanf("%d", &n);
printf("Enter %d numbers:\n", n);
for(int i = 0; i < n; i++) {
scanf("%d", &numbers[i]);
  }
total = findSum(numbers, n);
printf("Sum of array elements = %d\n", total);

return 0;
}
```

Output:

Enter number of elements: 4

Enter 4 numbers:

5

10

15

20

Sum of array elements = 50

3. Implement C programming pointers and structures

Program:

```
#include <stdio.h>
struct Student {
introllNo;
char name[50];
};
int main() {
   struct Student s = {101, "Rahul"};
struct Student *ptr;
ptr = &s;
printf("Student Roll No: %d\n", ptr->rollNo);
printf("Student Name: %s\n", ptr->name);
return 0;
}
```

Output:

Student Roll No: 101

Student Name: Rahul

4. Implement C program using Files

Program:

```
#include <stdio.h>
int main() {
   FILE *fp;
char name[50];
fp = fopen("name.txt", "w");
if (fp == NULL) {
printf("Error opening file!\n");
return 1;
   }
printf("Enter your name: ");
scanf("%s", name);
fprintf(fp, "%s", name);
fclose(fp);
fp = fopen("name.txt", "r");
if (fp == NULL) {
printf("Error reading file!\n");
return 1;
   }
fscanf(fp, "%s", name);
printf("Name read from file: %s\n", name);
fclose(fp);  // close the file
return 0;
}
```

Sample Output:

```
Enter your name: Arjun
Name read from file: Arjun
```

5. Development of real time applications

Program:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>  // For delay()
int main() {
time_t now;
struct tm *local;
while (1) {
time(&now);
local = localtime(&now);
printf("Current Time: %02d:%02d:%02d\n",
local->tm_hour, local->tm_min, local->tm_sec);
delay(10);  // wait for 1000 milliseconds = 1 second
   }

return 0;
}
```

Output:

Current Time: 12:30:01

Current Time: 12:30:02

Current Time: 12:30:03

Current Time: 12:30:04

6. Array Implementation of list ADT

Program:

```
#include <stdio.h>
#define MAX 100
typedef struct {
int data[MAX];
int size;
} List;
// Insert element at the end
void insert(List *list, int value) {
if (list->size == MAX) {
printf("List is full.\n");
return;
  }
list->data[list->size] = value;
list->size++;
printf("Inserted %d\n", value);
}
// Delete element from the end
void delete(List *list) {
if (list->size == 0) {
printf("List is empty.\n");
return;
  }
list->size--;
printf("Deleted %d\n", list->data[list->size]);
}
// Display all elements
int i;
void display(List *list) {
if (list->size == 0) {
```

```
printf("List is empty.\n");
      return;
   }
printf("List: ");
for ( i = 0; i < list->size; i++) {
printf("%d ", list->data[i]);
   }
printf("\n");
}
int main() {
   List myList;
   myList.size = 0;
   insert(&myList, 10);
   insert(&myList, 20);
   insert(&myList, 30);
   display(&myList);
   delete(&myList);
display(&myList);
getch ();
return 0;
}
```

Output:

Inserted 10

Inserted 20

Inserted 30

List: 10 20 30

Deleted 30

List: 10 20

7. Array implementation of Stack and Queue ADTs

(i)Array Implementation of stack

Program:

```
#include <stdio.h>
#include<conio.h>
#define MAX 5  // Maximum size of the stack
int stack[MAX];  // Array to hold stack elements
int top = -1;    // Stack pointer (initially empty)
// Push operation
void push(int value) {
if (top == MAX - 1) {
printf("Stack is Full\n");
   } else {
top++;
stack[top] = value;
printf("Pushed: %d\n", value);
   }
}
// Pop operation
void pop() {
if (top == -1) {
printf("Stack is Empty\n");
   } else {
printf("Popped: %d\n", stack[top]);
top--;
   }
}
// Display stack
void display() {
if (top == -1) {
```

```
printf("Stack is Empty\n");
   } else {
inti;
printf("Stack: ");
for (i = 0; i<= top; i++) {
printf("%d ", stack[i]);
      }
printf("\n");
   }
}
// Main function
int main() {
push(10);
push(20);
push(30);
display();
pop();
display();
getch ();
return 0;
}
```

Output :

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

(ii)Array implementation of Queue

Program:

```
#include <stdio.h>
#define SIZE 5  // Maximum size of the queue
int queue[SIZE];
int front = 0;
int rear = -1;
// Enqueue function
voidenqueue(int value) {
if (rear == SIZE - 1) {
printf("Queue is Full\n");
   } else {
rear++;
queue[rear] = value;
printf("Enqueued: %d\n", value);
   }
}
// Dequeue function
voiddequeue() {
if (front > rear) {
printf("Queue is Empty\n");
   } else {
printf("Dequeued: %d\n", queue[front]);
front++;
   }
}
// Display function
void display() {
if (front > rear) {
printf("Queue is Empty\n");
   } else {
```

```
inti;
printf("Queue: ");
for (
i = front; i<= rear; i++) {
printf("%d ", queue[i]);
      }
printf("\n");
   }
}
// Main function
int main() {
enqueue(10);
enqueue(20);
enqueue(30);
display();
dequeue();
display();
getch ();
return 0;
}
```

Output:

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue: 10 20 30
Dequeued: 10
Queue: 20 30
```

8. Linked list implementation of List, Stack and Queue ADTs

Program:

Linked list implementation of List ADT

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
// Define Node structure
struct Node {
   int data;
   struct Node* next;
};
// Create a new node
struct Node* createNode(int value) {
   struct Node* newNode;
   newNode = (struct Node*) malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   return newNode;
}
// Display the linked list
void displayList(struct Node* head) {
   struct Node* temp = head;
   printf("List: ");
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}
// Main function
```

```
void main() {
   struct Node* head = NULL;
   struct Node* node1 = createNode(10);
   struct Node* node2 = createNode(20);
   struct Node* node3 = createNode(30);
   clrscr();
   // Link the nodes
   head = node1;
   node1->next = node2;
   node2->next = node3;
   // Display the list once
   displayList(head);
   getch();
}
```

Output

```
List: 10 -> 20 -> 30 -> NULL
```

Linked list implementation of stack ADT

Program:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
   int data;
   struct Node *next;
} Node;
void push(Node **top, int data) {
Node*newnode;
   newNode = (Node *)malloc(sizeof(Node));
   newNode->data = data;
   newNode->next = *top;
   *top = newNode;
}
int pop(Node **top) {
Node*temp;
int popped;
   if (*top == NULL) {
      printf("Stack is empty.\n");
      return -1;
   }
   temp = *top;
popped=temp->data;
   *top = temp->next;
   free(temp);
   return popped;
}
```

```
void displayStack(Node *top) {
   while (top != NULL) {
      printf("%d ", top->data);
      top = top->next;
   }
   printf("\n");
}
int main() {
   Node *stack = NULL;
   push(&stack, 10);
   push(&stack, 20);
   push(&stack, 30);
   displayStack(stack);
   printf("Popped: %d\n", pop(&stack));
   displayStack(stack);
   getch ();
   return 0;
}
```

Output

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

Linked list implementation of Queue ADT

Program:

```
#include <stdio.h>
#include <stdlib.h>
// Define node structure
typedef struct node {
   int data;
   struct node *next;
} Node;
// Initialize front and rear pointers
Node *front = NULL, *rear = NULL;
// Function to enqueue (insert element)
void enqueue(int value) {
   Node *newnode = (Node *)malloc(sizeof(Node));
   newnode->data = value;
   newnode->next = NULL;
   if (rear == NULL) {
      front = rear = newnode;
   } else {
      rear->next = newnode;
      rear = newnode;
   }}
// Function to dequeue (remove element)
int dequeue() {
   int value;
   Node *temp;
   if (front == NULL) {
      printf("Queue is empty.\n");
      return -1;
```

```
    }
    temp = front;
    value = temp->data;
    front = front->next;
    if (front == NULL) {
        rear = NULL;
    }
    free(temp);
    return value;
}
// Function to display the queue
void display() {
    Node *temp = front;
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Main function to test queue
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
```

```
    display();
    printf("Dequeued: %d\n", dequeue());
    display();
    return 0;
}
```

Output:

```
Queue: 10 20 30
Dequeued: 10
Queue: 20 30
```

9. Application of List, Stack and Queue ADTs

Program:

```
#include <stdio.h>
#define MAX 100
// ---------- LIST ADT ----------
typedef struct {
int data[MAX];
int size;
} List;
void list_insert(List *list, int value) {
if (list->size < MAX) {
list->data[list->size++] = value;
  }
}
void list_display(List *list) {
printf("List Elements: ");
for (int i = 0; i < list->size; i++) {
printf("%d ", list->data[i]);
  }
printf("\n");
}
// ---------- STACK ADT ----------
typedef struct {
int data[MAX];
int top;
} Stack;
void push(Stack *s, int value) {
s->data[++(s->top)] = value;
}
int pop(Stack *s) {
return s->data[(s->top)--];
```

```
}
void stack_display(Stack *s) {
printf("Stack Elements (Top to Bottom): ");
for (int i = s->top; i >= 0; i--) {
printf("%d ", s->data[i]);
  }
printf("\n");
}
// ---------- QUEUE ADT ----------
typedef struct {
int data[MAX];
int front, rear;
} Queue;
voidenqueue(Queue *q, int value) {
if (q->rear < MAX - 1) {
     q->data[++(q->rear)] = value;
if (q->front == -1) q->front = 0;
  }
}
int dequeue(Queue *q) {
if (q->front == -1 || q->front > q->rear)
return -1;
return q->data[(q->front)++];
}
void queue_display(Queue *q) {
printf("Queue Elements (Front to Rear): ");
for (int i = q->front; i <= q->rear; i++) {
printf("%d ", q->data[i]);
  }
printf("\n");
}
```

```
// ---------- MAIN FUNCTION ----------
int main() {
   List list = {.size = 0};
   Stack stack = {.top = -1};
   Queue queue = {.front = -1, .rear = -1};

   // Insert 3 numbers into list
   list_insert(&list, 10);
   list_insert(&list, 20);
   list_insert(&list, 30);
   list_display(&list);
   // Push same numbers into stack
   push(&stack, 10);
   push(&stack, 20);
   push(&stack, 30);
   stack_display(&stack);
   // Enqueue same numbers into queue
enqueue(&queue, 10);
enqueue(&queue, 20);
enqueue(&queue, 30);
   queue_display(&queue);
   return 0;
}
```

Output:

List Elements: 10 20 30

Stack Elements (Top to Bottom): 30 20 10

Queue Elements (Front to Rear): 10 20 30

10. Implementation of binary trees and operation of binary trees

Program:

```
#include <stdio.h>
#include <stdlib.h>
// Define structure of a binary tree node
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};
// Function to create a new node
struct Node* createNode(int value) {
   struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}
// Inorder traversal (Left, Root, Right)
void inorder(struct Node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
   }
}
// Main function
int main() {
```

```
   // Manually creating the binary tree
//      1
  //     / \
  //    2   3
struct Node* root = createNode(1);
   root->left = createNode(2);
  root->right = createNode(3);
   printf("Inorder Traversal of Binary Tree: ");
inorder(root);  // Output: 2 1 3
getch ();
return 0;
}
```

Output:

Inorder Traversal of Binary Tree: 2 1 3

11. Implementation of Binary Search Trees

Program:

```
#include <stdio.h>
#include <stdlib.h>
// Define structure of a BST node
struct Node {
int data;
struct Node *left, *right;
};
// Function to create a new node
struct Node* createNode(int value) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->left = newNode->right = NULL;
returnnewNode;
}
// Function to insert a node into BST
struct Node* insert(struct Node* root, int value) {
if (root == NULL) {
returncreateNode(value);
  }
if (value < root->data) {
root->left = insert(root->left, value);
  } else if (value > root->data) {
root->right = insert(root->right, value);
  }
return root;
}
```

```
// Inorder traversal (left, root, right)
voidinorder(struct Node* root) {
if (root != NULL) {
inorder(root->left);
printf("%d ", root->data);
inorder(root->right);
   }
}
// Main function
int main() {
struct Node* root = NULL;
   // Insert elements into BST
root = insert(root, 50);
insert(root, 30);
insert(root, 70);
insert(root, 20);
insert(root, 40);
insert(root, 60);
insert(root, 80);
   // Print BST using inorder traversal
printf("Inorder Traversal of BST: ");
inorder(root);
printf("\n");
getch ();
return 0;
}
```

Output:

Inorder Traversal of BST: 20 30 40 50 60 70 80

12. Implementation of searching techniques

Program:

Linear Search

```
#include <stdio.h>
int linearSearch(int arr[], int n, int key) {
int i;
   for ( i = 0; i < n; i++) {
      if (arr[i] == key)
         return i;  // Return index if found
   }
   return -1;  // Not found
}
int main() {
   int arr[100], n, key, i, result;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("Enter the element to search: ");
   scanf("%d", &key);
   result = linearSearch(arr, n, key);
   if (result != -1)
      printf("Element %d found at position %d\n", key, result + 1);
   else
      printf("Element %d not found in the array\n", key);
   return 0;
```

```
}
```

Output:

```
Enter number of elements: 5
Enter 5 elements:
10 20 30 40 50
Enter the element to search: 30
Element 30 found at position 3
```

Program:

Binary Search

```
#include <stdio.h>
int linearSearch(int arr[], int n, int key) {
int i;
   for ( i = 0; i < n; i++) {
      if (arr[i] == key)
         return i;  // Return index if found
   }
   return -1;  // Not found
}
int main() {
   int arr[100], n, key, i, result;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("Enter the element to search: ");
   scanf("%d", &key);
   result = linearSearch(arr, n, key);

   if (result != -1)
```

```
        printf("Element %d found at position %d\n", key, result + 1);
    else
        printf("Element %d not found in the array\n", key);
    return 0;
}
```

Output:

```
Enter number of elements: 5
Enter 5 elements:
10 20 30 40 50
Enter the element to search: 30
Element 30 found at position 3
```

13. Implementation of sorting algorithms: Insertion Sort, Quick Sort, Merge Sort

Program:

Insertion Sort

```
#include <stdio.h>
// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];    // current element
      j = i - 1;
      // Shift elements of arr[0..i-1] greater than key
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key; // insert key in correct position
   }}
// Function to print the array
void printArray(int arr[], int n) {
int i;
   for ( i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}
// Main function
int main() {
   int arr[] = {10, 2, 8, 6, 7};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    insertionSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
   getch ();
    return 0;
}
```

Output:

Original array: 10 2 8 6 7

Sorted array: 2 6 7 8 10

Program:

Quick sort:

```
#include <stdio.h>
// Function to swap two numbers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choose last element as pivot
    int i = low - 1;
    int j;
    for (j = low; j < high; j++) {
        if (arr[j] < pivot) {
```

```
            i++;
            swap(&arr[i], &arr[j]);
        }}
    // Place pivot at correct position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partition index
        quickSort(arr, low, pi - 1);  // Left side
        quickSort(arr, pi + 1, high); // Right side
    }
}
// Function to print the array
void printArray(int arr[], int n) {
   int i;
    for ( i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
// Main function
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
```

```
    return 0;
}
```

Output:

Original array: 10 7 8 9 1 5

Sorted array: 1 5 7 8 9 10

Program:

Merge sort:

```
#include <stdio.h>
// Function to merge two parts
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int a1 = mid - left + 1;
    int a2 = right - mid;
    int L[a1], R[a2];
    for (i = 0; i < a1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < a2; j++)
        R[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < a1 && j < a2) {
        if (L[i] < R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }
    while (i < a1)
        arr[k++] = L[i++];
    while (j < a2)
        arr[k++] = R[j++];
```

```
}
// Merge sort function
void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = (left + right) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }}
// Main function
int main() {
   int arr[] = {8, 3, 1, 7, 0, 10, 2};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Before sorting: ");
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   mergeSort(arr, 0, n - 1);
   printf("\nAfter sorting: ");
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   return 0;
}
```

Output:

Before sorting: 8 3 1 7 0 10 2

After sorting: 0 1 2 3 7 8 10

14. Implementation of Hashing – any two collision techniques

Program:

```
#include <stdio.h>
#define SIZE 10
int hashTable[SIZE];
// Initialize hash table with -1
void initialize() {
int i;
   for ( i = 0; i < SIZE; i++)
      hashTable[i] = -1;
} // Hash function
int hashFunction(int key) {
   return key % SIZE;
} // Insert using Linear Probing
void insertLinearProbing(int key) {
   int index = hashFunction(key);
   while (hashTable[index] != -1) {
      index = (index + 1) % SIZE;  // Move to next slot
   }
   hashTable[index] = key;
} // Insert using Quadratic Probing
void insertQuadraticProbing(int key) {
   int index = hashFunction(key);
   int i = 1;
   while (hashTable[index] != -1) {
      index = (index + i * i) % SIZE;  // Quadratic probing
      i++;
   }
```

```
    hashTable[index] = key;
} // Display hash table
void display() {
int i;
    printf("\nHash Table:\n");
    for ( i = 0; i < SIZE; i++) {
        if (hashTable[i] != -1)
            printf("Index %d -> %d\n", i, hashTable[i]);
        else
            printf("Index %d ->NULL\n", i);
    }} // Main function
int main() {
    int keys[] = {25, 35, 15, 5, 65};
    int n = sizeof(keys) / sizeof(keys[0]);
    int choice;
    int i;
    printf("Choose collision resolution method:\n");
    printf("1. Linear Probing\n");
    printf("2. Quadratic Probing\n");
    printf("Enter your choice (1 or 2): ");
    scanf("%d", &choice);
    initialize();
    for ( i = 0; i < n; i++) {
        if (choice == 1)
            insertLinearProbing(keys[i]);
        else if (choice == 2)
            insertQuadraticProbing(keys[i]);
        else {
            printf("Invalid choice.\n");
            return 0;
        }}
```

```
    display();
    return 0;
}
```

Output:

```
Choose collision resolution method:
1. Linear Probing
2. Quadratic Probing
Enter your choice (1 or 2): 1
Hash Table:
Index 0 → NULL
Index 1 → NULL
Index 2 → NULL
Index 3 → NULL
Index 4 → NULL
Index 5 → 25
Index 6 → 35
Index 7 → 15
Index 8 → 5
Index 9 → 65
```