KIT-KALAIGNARKARUNANIDHI INSTITUTE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTION)
(Accredited by NAAC&NBA with 'A' Grade)
Approved by AICTE & Affiliated to Anna University, Chennai)
Kannampalayam Post, Coimbatore-641402

Department of Artificial Intelligence and Data Science

B23CSP301 – DATA STRUCTURES LABORATORY

Name : ……………………………………………………………………………….

Batch : …………………… Reg.No.…………………………….……

Branch : ……………………. Year ………………………………..

KIT-KALAIGNARKARUNANIDHI INSTITUTE OF TECHNOLOGY
(ANAUTONOMOUS INSTITUTION)
(Accredited by NAAC & NBA with 'A' Grade)
Approved by AICTE & Affiliated to Anna University, Chennai)
Kannampalayam Post, Coimbatore-641402

Department of Artificial Intelligence and Data Science

BONAFIDE CERTIFICATE

Name :…..………………………………………………………………………………………

Roll No. : ……………………………………… Reg. No.: ……………………………………

Branch : B.Tech – Artificial Intelligence and Data Science

Certified that this is Bona-fide record work done by Mr./Ms. ……………………………………………

of II – Year Artificial Intelligence and Data Science during the academic year 2024-2025.

FACULTY IN-CHARGE **HOD**

Submitted for the University Practical Examination held on ……………………….

INTERNAL EXAMINER **EXTERNAL EXAMINER**

Instructions for Laboratory Classes

1. Enter the lab with record work book & necessary things.
2. Enter the lab without bags and footwear.
3. Footwear should be kept in the outside shoe rack neatly.
4. Maintain silence during the Lab hours.
5. Read and follow the work instructions inside the laboratory.
6. Handle the computer systems with care.
7. Shutdown the Computer properly and arrange chairs in order before leaving the lab.
8. The program should be written on the left side pages of the record workbook.
9. The record work book should be completed in all aspects and submitted in the next class itself.
10. Experiment number with date should be written at the top left-hand corner of the record work book page.
11. Strictly follow the uniform dress code for Laboratory classes.
12. Maintain punctuality for lab classes.
13. Avoid eatables inside and maintain the cleanliness of the lab.

VISION

To produce competent professionals to the dynamic needs of the emerging field of Artificial Intelligence and Data Science

MISSION

- To empower students with the knowledge and skills necessary to create intelligent systems and innovative solutions that address societal issues.
- Providing technical knowledge on par with Industry to the students through qualified faculty members having knowledge in recent trends and technologies.
- To produce competent engineers who are both professional and life-skills oriented.
- Providing opportunities for students to improve their research skills in order to address a variety of societal concerns through innovative projects.

PROGRAMME OUTCOMES (POs)

Students graduating from Artificial Intelligence and Data Science should be able to:

PO1 Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex Artificial Intelligence and Data Science problems.

PO2 Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and Artificial Intelligence and Data Sciences.

PO3 Design/development to solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations in the field of Artificial Intelligence and Data Science.

PO4 Conduct investigations of complex problems: Using research-based knowledge and Artificial Intelligence & Data Science oriented research methodologies including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex Artificial Intelligence and Data Science Engineering activities with an understanding of the limitations.

PO6 The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7Environmentandsustainability: Understand the impact of the professional Artificial Intelligence and Data Science Engineering solutions in societal and environmental contexts, and demonstrate the knowledge, and need for the sustainable development.

PO8 Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9 Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams and in multidisciplinary settings.

PO10 Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 Project management and finance: Demonstrate knowledge and understanding of the Artificial Intelligence and Data Science engineering and management principles and apply these to one's own work, as a member and leader in a team and, to manage projects in multidisciplinary environments.

PO12 Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Graduates will have a strong foundation in mathematics, programming, machine learning, artificial intelligence, and data science, as well as advanced skills in these areas to solve technical problems.

PEO2: Graduates will have the capability to apply their knowledge and skills to identify and solve the issues in real world Artificial Intelligence and Data Science related applications.

PEO3: Graduates will be able to engage in life-long learning by completing advanced software Technologies, certificates, and/or other professional development.

PROGRAM SPECIFIC OUTCOME(PSOs)

Graduates of Artificial Intelligence and Data Science Programmed should be able to:

PSO1: Apply fundamental concepts of Artificial Intelligence and Data Science according to the environmental needs.

PSO2: Ability to develop skills to address and solve Artificial Intelligence based social and environmental problem using Data Science to deal multidisciplinary projects using modern tools.

COURSE OUTCOMES:

Students will be able to:

Course Outcome	Knowledge Level
CO1: Categorize basic ADT's	K4
CO2: Examine the applications of ADT's	K4
CO3: Develop Algorithms to perform search and sort operations.	K3
CO4: Analyzing the functioning of various types of tree and graph structures.	K4
CO5: Assess different types of Hashing Techniques and collision avoidance strategies.	K5

CO/PO &PSO		PO1 (K3)	PO2 (K4)	PO3 (K5)	PO4 (K5)	PO5 (K6)	PO6 (K3) (A3)	PO7 (K2) (A3)	PO8 (K3) (A3)	PO9 (A3)	PO10 (A3)	PO11 (K3) (A3)	PO12 (A3)	PSO1 (K3,A3)	PSO2 (K3,A3)
CO1	K3	3	3	3	2	2	-	-	-	2	2	3	3	2	2
CO2	K3	3	3	3	2	2	-	-	-	2	2	2	2	3	2
CO3	K3	3	3	3	2	2	-	-	-	2	2	3	2	2	3
CO4	K3	3	3	3	2	2	-	-	-	2	2	2	2	2	2
CO5	K3	3	3	3	2	2	-	-	-	2	2	2	3	3	2
Weighted Average		3	3	3	2	2	-	-	-	2	2	2.4	2.4	2.4	2.2

SYLLABUS

LIST OF EXPERIMENTS

1. Implementation of Stack and Queue using Array.
2. Array implementation of List.
3. Linked List implementation of Stack and Queue.
4. Implementation of Doubly Linked List using Pointers.
5. Applications of List in Polynomial Addition and Subtraction.
6. Implementation of Infix to Postfix Conversion using Stack ADT.
7. Implementation of Linear and Binary Search.
8. Implementation of AVL Tree.
9. Graph Representation and Traversal using DFS and BFS.
10. Implementation of Insertion, Bubble, Merge and Quick sort techniques.
11. Implementation of Heap using Priority Queue.
12. Implementation of Hash Functions and Collision Resolution Techniques.

TOTAL HRS : 45

INDEX

CONTENT BEYOND THE SYLLABUS		
13	Functions of Dictionary (ADT) Using Hashing	123

	EXPERIMET	**PREREQUISITS**	**LEARNING OBJECTIVES**
	FOR ALL EXPERIMENTS	PROGRAMMING IN C	1. To understand the practical application of linear data structures. 2. To understand the different operations of search trees. 3. To familiarize graphs and their applications. 4. To demonstrate different sorting and searching techniques. 5. To implement the different hashing techniques.

Sl. No.	Date	Name of the Experiment	Page Number	Aim & Algorithm (20 Marks)	Program (30 Marks)	Output & Inference (15 Marks)	Viva-Voce (10Marks)	Total (75Marks)	Signature of the Faculty Member

Model Exam Marks (25):___________Total (100):________

Practical Record Book Index Page

Sl. No.	Date	Name of the Experiment	Page Number	Aim & Algorithm (20 Marks)	Program (30 Marks)	Output & Inference (15Marks)	Viva-Voce (10Marks)	Total (75Marks)	Signature of the Faculty Member

Practical Record Book Index Page

Sl. No.	Date	Name of the Experiment	Page Number	Aim & Algorithm (20 Marks)	Program (30 Marks)	Output & Inference (15Marks)	Viva-Voce (10Marks)	Total (75Marks)	Signature of the Faculty Member

Model Exam Marks (25):________________Total (100):_____

Signature of the Faculty Member

Practical Record Book Index Page

Sl. No.	Date	Name of the Experiment	Page Number	Aim & Algorithm (20 Marks)	Program (30 Marks)	Output & Inference (15 Marks)	Viva-Voce (10 Marks)	Total (75 Marks)	Signature of the Faculty Member

Model Exam Marks (25):________________Total (100):____ **Signature of the Faculty Member**

x. No:	**1(A)**	**IMPLEMENTATION OF STACK USING ARRAY**
Date:		

AIM

To Write a C program for implementation of stack using array.

ALGORITHM

Step 1: Start the program
Step 2: Check if the stack is full (i.e., if top == max_size - 1).
Step 3: If full, print an overflow message.
Step 4: Otherwise, increment the top pointer and insert the element at the top position.
Step 5: Check if the stack is empty (i.e., if top == -1).
Step 6: If empty, print an underflow message.
Step 7: Otherwise, return the element at the top position and decrement the top pointer.
Step 8: Check if the stack is empty (i.e., if top == -1).
Step 9: If empty, print a message indicating the stack is empty.
Step 10: Otherwise, return the element at the top position without modifying top.
Step 11: Stop the Program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5  // Maximum size of the stack

// Stack structure
struct Stack {
    int arr[MAX];
    int top;
};

// Function to initialize the stack
void initStack(struct Stack* s) {
    s->top = -1;
}

// Function to check if the stack is full
int isFull(struct Stack* s) {
    return s->top == MAX - 1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* s) {
    return s->top == -1;
}

// Push operation
void push(struct Stack* s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow! Cannot push %d\n", value);
    } else {
        s->arr[++(s->top)] = value;
        printf("%d pushed to the stack\n", value);
    }
}

// Pop operation
int pop(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;  // Indicates an error
    } else {
        int poppedValue = s->arr[(s->top)--];
        return poppedValue;
    }
}

// Peek operation
int peek(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack is empty! No element to peek\n");
        return -1;  // Indicates an error
    } else {
        return s->arr[s->top];
    }
}

// Main program to test stack operations
int main() {
    struct Stack stack;
    initStack(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Top element is %d\n", peek(&stack));

    printf("Popped element is %d\n", pop(&stack));
    printf("Top element after pop is %d\n", peek(&stack));

    push(&stack, 40);
    printf("Top element is %d\n", peek(&stack));

    return 0; }
```

OUTPUT:

```
10 pushed to the stack
20 pushed to the stack
30 pushed to the stack
Top element is 30
Popped element is 30
Top element after pop is 20
40 pushed to the stack
Top element is 40
```

RESULT

Thus the program for stack using array is executed successfully & verified.

Ex.No:	**1 (B)**	**IMPLEMENTATION OF QUEUE USING ARRAY**
Date:		

AIM

To write a C program for implementation of queue using Array.

ALGORITHM

Step 1: Start the program.
Step 2: Check if the stack is full (i.e., if top == max_size - 1).
Step 3: If full, print an overflow message.
Step 5: Otherwise, increment the top pointer and insert the element at the top position.
Step 6: Check if the stack is empty (i.e., if top == -1).
Step 7: If empty, print an underflow message.
Step 8: Otherwise, return the element at the top position and decrement the top pointer.
Step 9: Check if the stack is empty (i.e., if top == -1).
Step 10: If empty, print a message indicating the stack is empty.
Step 11: Otherwise, return the element at the top position without modifying top.
Step 12: Stop the Program.

PROGRAM

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

#define MAX 5  // Maximum size of the queue

// Queue structure
struct Queue {
    int arr[MAX];
    int front;
    int rear;
};

// Function to initialize the queue
void initQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(struct Queue* q) {
    return q->rear == MAX - 1;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
    return q->front == -1 || q->front > q->rear;
}

// Enqueue operation
void enqueue(struct Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
    } else {
        if (q->front == -1) {
            q->front = 0;  // Set front to 0 if it's the first element
        }
        q->arr[++(q->rear)] = value;
        printf("%d enqueued to the queue\n", value);
    }
}

// Dequeue operation
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow! Cannot dequeue\n");
        return -1;  // Indicates an error
    } else {
        int dequeuedValue = q->arr[(q->front)++];
        return dequeuedValue;
    }
}

// Peek operation
int peek(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! No element to peek\n");
        return -1;  // Indicates an error
    } else {
        return q->arr[q->front];
    }
}

// Main program to test queue operations
int main() {
    struct Queue queue;
    initQueue(&queue);

    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);

    printf("Front element is %d\n", peek(&queue));

    printf("Dequeued element is %d\n", dequeue(&queue));
    printf("Front element after dequeue is %d\n", peek(&queue));

    enqueue(&queue, 40);
    printf("Front element is %d\n", peek(&queue));

    return 0;
}
```

OUTPUT:

```
10 enqueued to the queue
20 enqueued to the queue
30 enqueued to the queue
Front element is 10
Dequeued element is 10
Front element after dequeue is 20
40 enqueued to the queue
Front element is 20
```

RESULT

Thus the program for Queue using array is executed successfully & verified.

Ex.No:	**2**	**ARRAY IMPLEMENTATION OF LIST**
Date:		

AIM

To write a C program for array implementation of List.

ALGORITHM

Step 1: Start the program.

Step 2: Check if the list is full (i.e., size == max_size).

Step 3: If full, dynamically increase the size of the array (by reallocating memory).

Step 4: Insert the new element at the end (i.e., at index size), and increment the size counter.

Step 5: Check if the index to remove is valid (i.e., index >= 0 && index < size).

Step 6: Shift all elements to the left starting from the removal index.

Step 7: Decrease the size counter.

Step 8: Check if the index is valid (i.e., index >= 0 && index < size).

Step 9 : Return the element at the given index.

PROGRAM

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

// Define the initial size of the list
#define INITIAL_CAPACITY 2

// List structure
struct List {
    int* array;  // Dynamic array to store list elements
    int size;    // Current number of elements
    int capacity;  // Maximum capacity of the list
};

// Function to initialize the list
void initList(struct List* list) {
    list->capacity = INITIAL_CAPACITY;
    list->size = 0;
    list->array = (int*)malloc(list->capacity * sizeof(int));
}

// Function to check if resizing is needed and expand the array
void resizeList(struct List* list) {
    if (list->size == list->capacity) {
        list->capacity *= 2;  // Double the capacity
        list->array = (int*)realloc(list->array, list->capacity * sizeof(int));
        printf("List resized to capacity %d\n", list->capacity);
    }
}

// Add element to the list
void add(struct List* list, int value) {
    resizeList(list);  // Resize if needed
    list->array[list->size] = value;
    list->size++;
    printf("%d added to the list\n", value);
}

// Remove element from the list at the given index
```

```
void removeAt(struct List* list, int index) {
    if (index < 0 || index >= list->size) {
        printf("Invalid index!\n");
        return;
    }
    // Shift elements to the left to fill the gap
    for (int i = index; i < list->size - 1; i++) {
        list->array[i] = list->array[i + 1];
    }
    list->size--;
    printf("Element at index %d removed\n", index);
}

// Get element at the given index
int get(struct List* list, int index) {
    if (index < 0 || index >= list->size) {
        printf("Invalid index!\n");
        return -1;  // Indicates an error
    }
    return list->array[index];
}

// Main program to test the list implementation
int main() {
    struct List myList;
    initList(&myList);

    // Adding elements
    add(&myList, 10);
    add(&myList, 20);
    add(&myList, 30);

    // Getting elements
    printf("Element at index 1: %d\n", get(&myList, 1));

    // Removing an element
    removeAt(&myList, 1);

    // Getting elements after removal
    printf("Element at index 1 after removal: %d\n", get(&myList, 1));

    // Free allocated memory
    free(myList.array);

    return 0;
}
```

OUTPUT:

```
10 added to the list
20 added to the list
List resized to capacity 4
30 added to the list
Element at index 1: 20
Element at index 1 removed
Element at index 1 after removal: 30
```

RESULT

Thus the program for Array implementation of list is executed successfully & verified.

Ex.No:	**3 (A)**	**LINKED LIST IMPLEMENTATION OF STACK**
Date:		

AIM

To write a program for linked list implementation of stack.

ALGORITHM

Step1: Start the Program.

Step 2: Create a new node.

Step 3: Set the new node’s data field to the value to be pushed.

Step 4: Set the new node’s next field to the current top of the stack.

Step 5: Update the top to point to the new node.

Step 6: Check if the stack is empty (i.e., if top == NULL).

Step 7: If empty, print an underflow message.

Step 8: Otherwise, retrieve the data of the node at the top.

Step 9: Update the top to point to the next node.

Step 10: Free the memory of the popped node.

Step 11: Check if the stack is empty (i.e., if top == NULL).

Step 12: If empty, print a message indicating the stack is empty.

Step 13: Otherwise, return the data of the node at the top.

Step 14: Stop the Program.

PROGRAM :

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

// Structure to represent a stack node
struct Node {
   int data;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (!newNode) {
      printf("Memory allocation failed\n");
      exit(1);  // Exit if memory allocation fails
   }
   newNode->data = value;
   newNode->next = NULL;
   return newNode;
}

// Function to check if the stack is empty
int isEmpty(struct Node* top) {
   return top == NULL;
}

// Push operation
void push(struct Node** top, int value) {
   struct Node* newNode = createNode(value);
   newNode->next = *top;
   *top = newNode;
   printf("%d pushed to the stack\n", value);
}

// Pop operation
int pop(struct Node** top) {
   if (isEmpty(*top)) {
      printf("Stack Underflow! Cannot pop\n");
```

```
        return -1;  // Indicates an error
    } else {
        struct Node* temp = *top;
        int poppedValue = temp->data;
        *top = (*top)->next;
        free(temp);
        return poppedValue;
    }
}

// Peek operation
int peek(struct Node* top) {
    if (isEmpty(top)) {
        printf("Stack is empty! No element to peek\n");
        return -1;  // Indicates an error
    } else {
        return top->data;
    }
}

// Main program to test stack operations
int main() {
    struct Node* stack = NULL;  // Stack is initially empty

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Top element is %d\n", peek(stack));

    printf("Popped element is %d\n", pop(&stack));
    printf("Top element after pop is %d\n", peek(stack));

    push(&stack, 40);
    printf("Top element is %d\n", peek(stack));

    return 0;
         }
```

OUTPUT:

```
10 pushed to the stack
20 pushed to the stack
30 pushed to the stack
Top element is 30
Popped element is 30
Top element after pop is 20
40 pushed to the stack
Top element is 40
```

RESULT

Thus, the above program of linked list implementation of stack is executed and the output is verified successfully.

Ex.No:	3 (B)	LINKED LIST IMPLEMENTATION OF QUEUE
Date:		

AIM

To write a program for implementation of Queue using Linked list.

ALGORITHM :

Step 1: Start the Program.

Step 2: Create a new node.

Step 3: Set the new node’s data field to the value to be enqueued.

Step 4: Set the new node’s next field to NULL.

Step 5: If the queue is empty (i.e., front == NULL), set both front and rear to the new node.

Step 6: Otherwise, set the next of the current rear to the new node, and update rear to point to the new node.

Step 7: Check if the queue is empty (i.e., front == NULL).

Step 8: If empty, print an underflow message.

Step 9: Otherwise, retrieve the data of the node at the front.

Step 10: Update front to point to the next node.

Step 11: If front becomes NULL, set rear to NULL (i.e., the queue is now empty).

Step 12: Free the memory of the dequeued node.

Step 13: Check if the queue is empty (i.e., front == NULL).

Step 14: If empty, print a message indicating the queue is empty.

Step 15: Otherwise, return the data of the node at the front.

PROGRAM:

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

// Structure to represent a queue node
struct Node {
    int data;
    struct Node* next;
};

// Queue structure with front and rear pointers
struct Queue {
    struct Node* front;
    struct Node* rear;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);  // Exit if memory allocation fails
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize a queue
void initQueue(struct Queue* q) {
    q->front = NULL;
    q->rear = NULL;
}

// Enqueue operation (insert element at the rear)
void enqueue(struct Queue* q, int value) {
    struct Node* newNode = createNode(value);
    if (q->rear == NULL) {  // If the queue is empty
        q->front = q->rear = newNode;
        printf("%d enqueued to the queue\n", value);
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
    printf("%d enqueued to the queue\n", value);
}

// Dequeue operation (remove element from the front)
int dequeue(struct Queue* q) {
    if (q->front == NULL) {  // If the queue is empty
        printf("Queue Underflow! Cannot dequeue\n");
        return -1;  // Indicates an error
    }
    struct Node* temp = q->front;
    int dequeuedValue = temp->data;
    q->front = q->front->next;

    if (q->front == NULL) {  // If the queue becomes empty after dequeue
        q->rear = NULL;
    }
    free(temp);  // Free the memory of the dequeued node
    return dequeuedValue;
}

// Peek operation (get the front element)
int peek(struct Queue* q) {
    if (q->front == NULL) {  // If the queue is empty
        printf("Queue is empty! No element to peek\n");
        return -1;  // Indicates an error
    }
    return q->front->data;
}

// Main program to test queue operations
int main() {
    struct Queue q;
    initQueue(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);

    printf("Front element is %d\n", peek(&q));

    printf("Dequeued element is %d\n", dequeue(&q));
    printf("Front element after dequeue is %d\n", peek(&q));

    enqueue(&q, 40);
    printf("Front element is %d\n", peek(&q));

    return 0;
}
```

OUTPUT:

```
10 enqueued to the queue
20 enqueued to the queue
30 enqueued to the queue
Front element is 10
Dequeued element is 10
Front element after dequeue is 20
40 enqueued to the queue
Front element is 20
```

RESULT :

Thus the program for linked list implementation of queue is executed successfully and verified.

Ex.No:	**4**	**IMPLEMENTATION OF DOUBLY LINKED LIST USING POINTERS**
Date:		

AIM

To Write a program for implementing doubly linked list using pointers.

ALGORITHM

Step 1: Start the Program.
Step 2: Create a new node.
Step 3: Set the new node's data field to the value to be inserted.
Step 4: If the list is empty, set both head and tail to the new node.
Step 5: Otherwise, set the next of the current tail to the new node and the prev of the new node to the current tail. Then, update tail to point to the new node.
Step 6: Check if the list is empty (i.e., head == NULL).
Step 7: If not, traverse the list to find the node at the given position.
Step 8: If the node to be deleted is the head, update head to point to the next node.
Step 10: If the node to be deleted is the tail, update tail to point to the previous node.
Step 11: Adjust the next and prev pointers of the neighboring nodes to bypass the node being deleted.
Step 12: Free the memory of the deleted node.
Step 13: Start from the head and traverse the list, printing the data of each node until the end of the list is reached.
Step 14: Start from the tail and traverse the list in reverse by following the prev pointers, printing the data of each node until the head is reached.

PROGRAM:

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

// Structure to represent a node in the doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Insert a new node at the end of the list
void insertAtEnd(struct Node** head, struct Node** tail, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    newNode->prev = *tail;

    if (*head == NULL) {  // If the list is empty
        *head = *tail = newNode;
    } else {
        (*tail)->next = newNode;
        *tail = newNode;
    }
    printf("%d inserted at the end\n", value);
}

// Delete a node from the list at a given position
void deleteAtPosition(struct Node** head, struct Node** tail, int position) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete\n");
        return;
    }

    struct Node* temp = *head;
    int i;
    // Traverse to the node at the specified position
    for (i = 1; temp != NULL && i < position; i++) {
        temp = temp->next;
    }

    // If position is beyond list length
    if (temp == NULL) {
        printf("Position %d is out of range\n", position);
        return;
    }

    // If node to be deleted is the head node
    if (temp == *head) {
        *head = temp->next;
    }

    // If node to be deleted is the tail node
    if (temp == *tail) {
        *tail = temp->prev;
    }

    // Update links of the previous and next nodes
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }

    free(temp);  // Free memory of the deleted node
    printf("Node at position %d deleted\n", position);
}

// Display the list from the head to the tail
void displayList(struct Node* head) {
    struct Node* temp = head;
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Display the list in reverse from tail to head
void reverseDisplayList(struct Node* tail) {
    struct Node* temp = tail;
    if (tail == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Reversed List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
}

// Main function to test doubly linked list operations
int main() {
    struct Node* head = NULL;
    struct Node* tail = NULL;

    insertAtEnd(&head, &tail, 10);
    insertAtEnd(&head, &tail, 20);
    insertAtEnd(&head, &tail, 30);
    insertAtEnd(&head, &tail, 40);
    displayList(head);

    deleteAtPosition(&head, &tail, 2);  // Delete node at position 2

    displayList(head);

    reverseDisplayList(tail);

    return 0;
}
```

OUTPUT:

```
10 inserted at the end
20 inserted at the end
30 inserted at the end
40 inserted at the end
List: 10 20 30 40
Node at position 2 deleted
List: 10 30 40
Reversed List: 40 30 10
```

RESULT

Thus the program for doubly linked list using pointers is executed successfully and verified.

Ex.No:	**5 (A)**	**APPLICATIONS OF POLYNOMIAL ADDITION**
Date:		

AIM

Write a program for polynomial addition using C.

ALGORITHM

Step 1: Start the program.
Step 2: **Create two linked lists** representing two polynomials. Each node of the linked list will store a coefficient and an exponent.
Step 3: **Traverse both polynomials** simultaneously:

Step4: Compare the exponents of the current terms of both polynomials.
Step 5: If the exponents are equal, add the coefficients and create a new term with the resulting coefficient and exponent.

Step 6: If the exponent of the first polynomial is larger, add its term to the result.

Step 7: If the exponent of the second polynomial is larger, add its term to the result.

Step 8: Continue adding remaining terms if any polynomial still has terms after the other polynomial finishes.

Step 9: Display the result.

Step 10: Stop the program.

PROGRAM:

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

// Define a structure to represent a term in the polynomial
struct Node {
    int coeff;
    int exp;
    struct Node* next;
};

// Function to create a new node (term) in the polynomial
struct Node* createNode(int coeff, int exp) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node (term) into the polynomial
void insertNode(struct Node** poly, int coeff, int exp) {
    struct Node* newNode = createNode(coeff, exp);
    if (*poly == NULL) {
        *poly = newNode;
    } else {
        struct Node* temp = *poly;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to add two polynomials and return the result
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;
    struct Node* p1 = poly1;
    struct Node* p2 = poly2;

    while (p1 != NULL && p2 != NULL) {
        if (p1->exp > p2->exp) {
            insertNode(&result, p1->coeff, p1->exp);
            p1 = p1->next;
        } else if (p1->exp < p2->exp) {
            insertNode(&result, p2->coeff, p2->exp);
            p2 = p2->next;
        } else {
            // Add coefficients if exponents are the same
            int sumCoeff = p1->coeff + p2->coeff;
            if (sumCoeff != 0) {  // Only add non-zero coefficients
                insertNode(&result, sumCoeff, p1->exp);
            }
            p1 = p1->next;
            p2 = p2->next;
        }
    }

    // Add remaining terms from poly1
    while (p1 != NULL) {
        insertNode(&result, p1->coeff, p1->exp);
        p1 = p1->next;
    }

    // Add remaining terms from poly2
    while (p2 != NULL) {
        insertNode(&result, p2->coeff, p2->exp);
        p2 = p2->next;
    }

    return result;
}

// Function to display the polynomial
void displayPolynomial(struct Node* poly) {
    if (poly == NULL) {
        printf("0");
        return;
    }
    while (poly != NULL) {
        printf("%d", poly->coeff);
        if (poly->exp != 0) {
            printf("x^%d", poly->exp);
        }
        poly = poly->next;
        if (poly != NULL) {
            printf(" + ");
        }
    }
    printf("\n");
}

// Main function to test polynomial addition
int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;

    // Create first polynomial: 3x^3 + 5x^2 + 6
    insertNode(&poly1, 3, 3);
    insertNode(&poly1, 5, 2);
    insertNode(&poly1, 6, 0);

    // Create second polynomial: 4x^2 + 2x^1 + 1
    insertNode(&poly2, 4, 2);
    insertNode(&poly2, 2, 1);
    insertNode(&poly2, 1, 0);

    printf("First Polynomial: ");
    displayPolynomial(poly1);

    printf("Second Polynomial: ");
    displayPolynomial(poly2);

    // Add the two polynomials
    result = addPolynomials(poly1, poly2);
    printf("Resultant Polynomial: ");
    displayPolynomial(result);

    return 0;
}
```

OUTPUT:

```
First Polynomial: 3x^3 + 5x^2 + 6
Second Polynomial: 4x^2 + 2x^1 + 1
Resultant Polynomial: 3x^3 + 9x^2 + 2x^1 + 7
```

RESULT:

Thus the program applications of polynomial addition is executed successfully & verified.

Ex.No:	5 (B)	APPLICATIONS OF POLYNOMIAL SUBTRACTION
Date:		

AIM

To write a C Program for applications of Polynomial subtraction.

ALGORITHM

Step 1: Start the program.

Step 2: **Create two linked lists** representing the two polynomials. Each node in the linked list will store a coefficient and an exponent.

Step 3: **Traverse both polynomials** simultaneously:

Step 4: Compare the exponents of the current terms of both polynomials.

Step 5: If the exponents are equal, subtract the coefficients and create a new term with the resulting coefficient and the same exponent.

Step 6: If the exponent of the first polynomial is larger, add its term to the result.

Step 7: If the exponent of the second polynomial is larger, subtract its term and add it to the result (with a negative sign).

Step 8: **Continue subtracting remaining terms** if one polynomial has more terms than the other.

Step 9: Display the result.

Step 10: Stop the Program.

PROGRAM :

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

// Define a structure to represent a term in the polynomial
struct Node {
    int coeff;
    int exp;
    struct Node* next;
};

// Function to create a new node (term) in the polynomial
struct Node* createNode(int coeff, int exp) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node (term) into the polynomial
void insertNode(struct Node** poly, int coeff, int exp) {
    struct Node* newNode = createNode(coeff, exp);
    if (*poly == NULL) {
        *poly = newNode;
    } else {
        struct Node* temp = *poly;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to subtract two polynomials and return the result
```

```
struct Node* subtractPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;
    struct Node* p1 = poly1;
    struct Node* p2 = poly2;

    while (p1 != NULL && p2 != NULL) {
        if (p1->exp > p2->exp) {
            insertNode(&result, p1->coeff, p1->exp);
            p1 = p1->next;
        } else if (p1->exp < p2->exp) {
            insertNode(&result, -p2->coeff, p2->exp);
            p2 = p2->next;
        } else {
            // Subtract coefficients if exponents are the same
            int diffCoeff = p1->coeff - p2->coeff;
            if (diffCoeff != 0) {  // Only insert non-zero terms
                insertNode(&result, diffCoeff, p1->exp);
            }
            p1 = p1->next;
            p2 = p2->next;
        }
    }

    // Add remaining terms from poly1
    while (p1 != NULL) {
        insertNode(&result, p1->coeff, p1->exp);
        p1 = p1->next;
    }

    // Add remaining terms from poly2 with negative sign
    while (p2 != NULL) {
        insertNode(&result, -p2->coeff, p2->exp);
        p2 = p2->next;
    }

    return result;
```

```
}

// Function to display the polynomial
void displayPolynomial(struct Node* poly) {
    if (poly == NULL) {
        printf("0");
        return;
    }
    while (poly != NULL) {
        if (poly->coeff > 0 && poly != poly->next) printf("+");  // Format output for positive terms
        printf("%d", poly->coeff);
        if (poly->exp != 0) {
            printf("x^%d", poly->exp);
        }
        poly = poly->next;
        if (poly != NULL) {
            printf(" ");
        }
    }
    printf("\n");
}

// Main function to test polynomial subtraction
int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;

    // Create first polynomial: 3x^3 + 5x^2 + 6
    insertNode(&poly1, 3, 3);
    insertNode(&poly1, 5, 2);
    insertNode(&poly1, 6, 0);

    // Create second polynomial: 4x^2 + 2x^1 + 1
    insertNode(&poly2, 4, 2);
    insertNode(&poly2, 2, 1);
    insertNode(&poly2, 1, 0);

    printf("First Polynomial: ");
    displayPolynomial(poly1);

    printf("Second Polynomial: ");
    displayPolynomial(poly2);

    // Subtract the two polynomials
    result = subtractPolynomials(poly1, poly2);

    printf("Resultant Polynomial (Poly1 - Poly2): ");
    displayPolynomial(result);

    return 0;
}
```

OUTPUT:

```
First Polynomial: +3x^3 +5x^2 +6
Second Polynomial: +4x^2 +2x^1 +1
Resultant Polynomial (Poly1 - Poly2): +3x^3 +1x^2 -2x^1 +5
```

.

RESULT

Thus the program for applications of polynomial subtraction is executed successfully and verified.

Ex.No:	6	UTILIZATION OF STACK ADT FOR INFIX TO POSTFIX CONVERSION
Date:		

AIM

To Write a C Program for utilization of stack ADT for infix and postfix conversion.

ALGORITHM

Step 1: Start the program.

Step 2: **Initialize an empty stack** to store operators and parentheses.
Step 3: **Traverse the infix expression** from left to right for each character:

Step 4: If the character is an operand (number or variable), add it to the output (postfix expression).

Step 5: If the character is an operator:

Step 6: Pop operators from the stack to the output until you find one with lower precedence or encounter an opening parenthesis.
Step 7: Push the current operator onto the stack.

Step 8: If the character is an opening parenthesis, push it onto the stack.

Step 9: If the character is a closing parenthesis, pop operators from the stack to the output until an opening parenthesis is found. Discard both parentheses.

Step 10: **Pop all remaining operators** from the stack to the output when the expression is fully processed.
Step 11: **The final output** is the postfix expression.

Step 12: Stop the program.

PROGRAM :

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

#define MAX 100 // Maximum size of the stack

// Stack structure
struct Stack {
    int top;
    char items[MAX];
};

// Function to initialize the stack
void initStack(struct Stack* s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* s) {
    return s->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack* s) {
    return s->top == MAX - 1;
}

// Function to push an element onto the stack
void push(struct Stack* s, char value) {
    if (isFull(s)) {
        printf("Stack overflow\n");
        return;
    }
    s->items[++(s->top)] = value;
}

// Function to pop an element from the stack
char pop(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        return '\0';
    }
    return s->items[(s->top)--];
}

// Function to peek the top element of the stack
char peek(struct Stack* s) {
    if (isEmpty(s)) {
        return '\0';
    }
    return s->items[s->top];
}

// Function to check if a character is an operator
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

// Function to return precedence of operators
int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}
```

```
// Function to convert infix expression to postfix
void infixToPostfix(char* infix, char* postfix) {
    struct Stack s;
    initStack(&s);

    int i, k = 0;
    for (i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];

        // If the character is an operand, add it to the output
        if (isalnum(ch)) {
            postfix[k++] = ch;
        }
        // If the character is an opening parenthesis, push it onto the stack
        else if (ch == '(') {
            push(&s, ch);
        }
        // If the character is a closing parenthesis, pop until an opening parenthesis is found
        else if (ch == ')') {
            while (!isEmpty(&s) && peek(&s) != '(') {
                postfix[k++] = pop(&s);
            }
            pop(&s); // Pop the opening parenthesis
        }
        // If the character is an operator
        else if (isOperator(ch)) {
            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(ch)) {
                postfix[k++] = pop(&s);
            }
            push(&s, ch);
        }
    }

    // Pop all remaining operators from the stack
    while (!isEmpty(&s)) {
        postfix[k++] = pop(&s);
    }

    postfix[k] = '\0'; // Null terminate the postfix expression
}

// Main function to test the conversion
int main() {
    char infix[MAX], postfix[MAX];

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

    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

OUTPUT:

```
Input: (a+b)*c

Output: ab+c*

Input: a+b*c

Output: abc*+
```

RESULT :

Thus the program for utilization of Stack ADT for infix and postfix conversion is executed successfully.

Ex.No:	**7 (A)**	**EXECUTION OF LINEAR SEARCH**
Date:		

AIM

To write a program for the execution of Linear Search.

ALGORITHM

Step 1: Start at the first element of the array.
Step 2: Compare the target element with the current element of the array.
Step 3: If the current element matches the target element, return the index of the element.
Step 4: If the current element does not match, move to the next element.
Step 5: Repeat steps 2–4 until the target element is found or the array ends.
Step 6: If the target element is not found after checking all elements, return -1 (indicating the element is not present in the array).

PROGRAM:

```
#include <stdio.h>

// Function to perform linear search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;  // Return the index if target is found
        }
    }
    return -1;  // Return -1 if target is not found
}

int main() {
    int n, target, result;

    // Asking the user for array size
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    int arr[n];  // Declaring an array of size 'n'

    // Taking array input from the user
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Asking the user for the target element
    printf("Enter the element to search: ");
    scanf("%d", &target);

    // Calling linear search function
    result = linearSearch(arr, n, target);

    // Displaying the result
    if (result == -1) {
        printf("Element not found in the array.\n");
    } else {
        printf("Element found at index %d.\n", result);
    }

    return 0;
}
```

OUTPUT:

```
Enter the number of elements in the array: 5
Enter 5 elements:
10 25 30 45 50
Enter the element to search: 30
Element found at index 2.
```

RESULT

Thus the program for the execution of linear search has been executed and verified successfully.

Ex.No:	**7 (B)**	**IMPLEMENTATION OF BINARY SEARCH**
Date:		

AIM

To Write a C Program for implementing Binary Search.

ALGOGITHM

Step 1: Start with the entire sorted array.
Step 2: Find the middle element.
Step 3: If the middle element matches the target, return the index.
Step 4: If the middle element is greater than the target, search in the left half of the array (ignore the right half).
Step 5: If the middle element is less than the target, search in the right half of the array (ignore the left half).
Step 6: Repeat steps 2–5 until the target element is found or the interval is empty.
Step 7: If the interval becomes empty, return -1 (indicating the element is not present in the array).

PROGRAM

```
 #include <stdio.h>

// Function to perform binary search
  int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;  // Finding the middle element

        // Check if target is present at mid
        if (arr[mid] == target) {
            return mid;  // Target found at index 'mid'
        }

        // If target is smaller than mid element, it is in the left half
        if (arr[mid] > target) {
            right = mid - 1;
        }
        // If target is larger than mid element, it is in the right half
        else {
            left = mid + 1;
        }
    }

    return -1;  // Target not found
}

int main() {
    int n, target, result;

    // Asking the user for array size
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    int arr[n];  // Declaring an array of size 'n'

    // Taking array input from the user
    printf("Enter %d sorted elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Asking the user for the target element
    printf("Enter the element to search: ");
    scanf("%d", &target);

    // Calling binary search function
    result = binarySearch(arr, n, target);

    // Displaying the result
    if (result == -1) {
        printf("Element not found in the array.\n");
    } else {
        printf("Element found at index %d.\n", result);
    }

    return 0;
}
```

OUTPUT:

```
Enter the number of elements in the array: 6
Enter 6 sorted elements:
5 10 15 20 25 30
Enter the element to search: 20
Element found at index 3.
```

RESULT

Thus the program for Binary search implementation is executed successfully and Verified.

Ex.No:	**8**	**IMPLEMENTATION OF AVL TREE**
Date:		

AIM

To write a C program for implementing AVL Tree.

ALGORITHM

Step 1: Perform a standard BST insert.
Step 2: After insertion, update the heights of all ancestor nodes.
Step 3: Check if the tree is balanced by calculating the balance factor.
Step 4: If the balance factor of any node is more than 1 or less than -1, perform appropriate rotations to restore balance.
Step 5: **Right Rotation**: Used to fix Left-Left imbalance.
Step 6: Left Rotation: Used to fix Right-Right imbalance.
Step 7: **Left-Right Rotation**: Used to fix Left-Right imbalance.
Step 8: **Right-Left Rotation**: Used to fix Right-Left imbalance.
Step 9: Calculated as the height of the left subtree minus the height of the right subtree.
Step 10: Balance factor must be between -1 and 1 for the tree to be balanced.

PROGRAM

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

// Node structure
struct Node {
   int key;
   struct Node* left;
   struct Node* right;
   int height;
};

// Function to get the height of the tree
int height(struct Node* node) {
   if (node == NULL)
      return 0;
   return node->height;
}

// Utility function to get the maximum of two integers
int max(int a, int b) {
   return (a > b) ? a : b;
}

// Function to create a new node
struct Node* newNode(int key) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->key = key;
   node->left = NULL;
   node->right = NULL;
   node->height = 1;  // New node is initially added at leaf
   return node;
}

// Right rotate the subtree rooted with y
struct Node* rightRotate(struct Node* y) {
   struct Node* x = y->left;
```

```
    struct Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// Left rotate the subtree rooted with x
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

// Get balance factor of node N
int getBalance(struct Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}
```

```
// Function to insert a node into the AVL tree
struct Node* insert(struct Node* node, int key) {
    // Perform the normal BST insertion
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else  // Equal keys are not allowed in AVL Tree
        return node;

    // Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // Get the balance factor of this node
    int balance = getBalance(node);

    // If the node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// Function to print the pre-order traversal of the tree
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main() {
    struct Node* root = NULL;

    // Insert nodes
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    // Print pre-order traversal of the AVL tree
    printf("Pre-order traversal of the AVL tree is: \n");
    preOrder(root);

    return 0;
}
```

OUTPUT

```
Pre-order traversal of the AVL tree is:
30 20 10 25 40 50
```

RESULT

Thus the program for implementing AVL tree is executed and verified successfully**.**

Ex.No:	**9 (A)**	**GRAPH REPRESENTATION AND TRAVERSAL USING DFS**
Date:		

**A
IM**

To write a c program for DFS.

ALGORITHM

Step 1: Start with a source node.
Step 2: Mark the node as visited.
Step 3: Recursively visit all unvisited neighbors of the node.
Step 4: Backtrack when no unvisited neighbors remain.

PROGRAM

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

struct Node {

    int vertex;

    struct Node* next;

};

// Structure to represent the adjacency list of the graph

struct Graph {

    int numVertices;

    struct Node** adjLists;  // Array of adjacency lists

    int* visited;            // Array to track visited vertices

};


// Function to create a new node

struct Node* createNode(int v) {

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

    newNode->vertex = v;

    newNode->next = NULL;

    return newNode;

}


// Function to create a graph with numVertices vertices

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

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

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;  // Initially, all vertices are unvisited
    }

    return graph;
}
// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add an edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Since the graph is undirected, add an edge from dest to src as well
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to perform DFS traversal
```

```
void DFS(struct Graph* graph, int vertex) {
   struct Node* adjList = graph->adjLists[vertex];
   struct Node* temp = adjList;

   // Mark the current vertex as visited
   graph->visited[vertex] = 1;
   printf("Visited %d\n", vertex);

   // Recursively visit all adjacent vertices
   while (temp != NULL) {
      int connectedVertex = temp->vertex;

      if (graph->visited[connectedVertex] == 0) {
         DFS(graph, connectedVertex);
      }
      temp = temp->next;
   }
}

int main() {
   int vertices = 6;

   // Create a graph
   struct Graph* graph = createGraph(vertices);

   // Add edges
   addEdge(graph, 0, 1);
```

```
    addEdge(graph, 0, 2);

    addEdge(graph, 1, 2);

    addEdge(graph, 1, 3);

    addEdge(graph, 2, 4);

    addEdge(graph, 3, 5);

    addEdge(graph, 4, 5);


    // Perform DFS starting from vertex 0

    printf("Depth First Traversal starting from vertex 0:\n");

    DFS(graph, 0);


    return 0;

}
```

OUTPUT

```
 Depth First Traversal starting from vertex 0:
Visited 0
Visited 2
Visited 4
Visited 5
Visited 3
Visited 1
```

RESULT

Thus the c program for Depth First Search is executed & verified successfully.

Ex.No:	**9 (B)**	**GRAPH REPRESENTATION AND TRAVERSAL USING BFS**
Date:		

AIM

To write a program for BFS.

ALGORITHM

Step 1: Start from a source node and mark it as visited.

Step 2: Add the node to the queue.

Step 3: While the queue is not empty:

- Dequeue a node.
- For each unvisited neighbor of the dequeued node, mark it as visited and enqueue it.

Step 4: Repeat the process until all vertices are visited.

Step 5: Stop the Program.

PROGRAM:

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

// Structure to represent a node in the adjacency list
struct Node {
    int vertex;
    struct Node* next;
};

// Structure to represent the adjacency list of the graph
struct Graph {
    int numVertices;
    struct Node** adjLists;  // Array of adjacency lists
    int* visited;            // Array to track visited vertices
};

// Queue structure for BFS
struct Queue {
    int items[100];
    int front;
    int rear;
};

// Function to create a new node
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph with numVertices vertices
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;  // Initially, all vertices are unvisited
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add an edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Since the graph is undirected, add an edge from dest to src as well
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to create a queue
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    if (queue->rear == -1)
        return 1;
    else
        return 0;
}
```

```
// Function to enqueue an element
void enqueue(struct Queue* queue, int value) {
    if (queue->rear == 99)
        printf("\nQueue is full!");
    else {
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}

// Function to dequeue an element
int dequeue(struct Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty");
        item = -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}

// Function to perform BFS traversal
void BFS(struct Graph* graph, int startVertex) {
    struct Queue* queue = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(queue, startVertex);

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        printf("Visited %d\n", currentVertex);
        struct Node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(queue, adjVertex);
            }
            temp = temp->next;
        }
    }
}

int main() {
    int vertices = 6;

    // Create a graph
    struct Graph* graph = createGraph(vertices);

    // Add edges
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 5);
    addEdge(graph, 4, 5);

    // Perform BFS starting from vertex 0
    printf("Breadth First Traversal starting from vertex 0:\n");
    BFS(graph, 0);

    return 0;
}
```

OUTPUT:

```
Breadth First Traversal starting from vertex 0:
Visited 0
Visited 1
Visited 2
Visited 3
Visited 4
Visited 5
```

RESULT

Thus the program for implementing BFS is executed & verified successfully.

Ex.No:	**10**	**IMPLEMENTATION OF INSERTION, BUBBLE, MERGE AND QUICK SORT TECHNIQUES**
Date:		

**A
IM**

To write a C program for insertion, bubble, merge and quick sort techniques.

ALGORITHM

Step 1: Insertion sort :

- Start from the second element (index 1) and compare it to the elements before it.
- Shift the elements larger than the current element to the right.
- Insert the current element into its correct position.
- Repeat for all elements.

Step 2: Bubble Sort :

- Compare adjacent elements and swap them if they are in the wrong order.
- Repeat the process for every element in the list.
- With each pass, the largest unsorted element gets placed at its correct position.
- Stop when no more swaps are needed.

Step 3: Merge Sort :

- Divide the array into two halves.
- Recursively sort the two halves.
- Merge the two sorted halves to get the fully sorted array.

Step 4: Quick Sort :

- Choose a pivot element from the array.
- Partition the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
- Recursively apply the same procedure to the subarrays.

Step 5: Stop the Program.

PROGRAM:

1. **Insertion Sort:**

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key, to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printf("Sorted array using Insertion Sort: \n");
    printArray(arr, n);

    return 0;
      }
```

2. **Bubble Sort:**

```c
#include <stdio.h>

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 arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);
    printf("Sorted array using Bubble Sort: \n");
    printArray(arr, n);

    return 0;
     }
```

3. **Merge Sort:**

```c
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2]; // Temporary arrays

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is: \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array using Merge Sort: \n");
    printArray(arr, arr_size);

    return 0;
      }
```

4. **Quick Sort:**

```
#include <stdio.h>

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Pivot
    int i = (low - 1);  // Index of smaller element
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);  // Partitioning index

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    printf("Sorted array using Quick Sort: \n");
    printArray(arr, n);

    return 0;
      }
```

OUTPUT:

1. **Insertion Sort Output:**

```
Input Array: {12, 11, 13, 5, 6}
Sorted Array using Insertion Sort:
5 6 11 12 13
```

2. **Bubble Sort Output:**

```
Input Array: {64, 34, 25, 12, 22, 11, 90}
Sorted Array using Bubble Sort:
11 12 22 25 34 64 90
```

3. **Merge Sort Output:**

```
Input Array: {12, 11, 13, 5, 6, 7}
Given array is:
12 11 13 5 6 7

Sorted array using Merge Sort:
5 6 7 11 12 13
```

4. **Quick Sort Output:**

```
Input Array: {10, 7, 8, 9, 1, 5}
Sorted array using Quick Sort:
1 5 7 8 9 10
```

RESULT

Thus the program for implementing sorting is executed & verified successfully.

Ex.No:	**11**	**IMPLEMENTATION OF HEAP USING PRIORITY QUEUE**
Date:		

AIM

To write a C Program for implementing priority queue using Heap.

ALGORITHM

Step 1: Insertion:
Step 2: Add the element at the end of the heap (complete tree property).
Step 3: Adjust the heap by comparing the new element with its parent. Swap if necessary (heap property).
Step 4: Deletion:
Step 5: Replace the root (max/min) with the last element.
Step 6: Remove the last element.
Step 7: Adjust the heap from the root down (heapify) by comparing with the children and swapping with the larger (max-heap) or smaller (min-heap) child.
Step 8: Stop the Program.

PROGRAM :

```
#include <stdio.h>

#define MAX 100  // Maximum size of the heap

int heap[MAX];  // Array representation of the heap
int heapSize = 0;

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to insert an element into the heap
void insert(int value) {
    // Insert the element at the end of the heap
    heap[heapSize] = value;
    int i = heapSize;
    heapSize++;

    // Heapify (move the new element up if necessary)
    while (i != 0 && heap[i] > heap[(i - 1) / 2]) {
        swap(&heap[i], &heap[(i - 1) / 2]);  // Swap with parent
        i = (i - 1) / 2;  // Move to the parent index
    }
}

// Function to heapify (adjust the heap after deletion)
void heapify(int i) {
    int largest = i;  // Initialize largest as root
    int left = 2 * i + 1;  // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is larger than root
    if (left < heapSize && heap[left] > heap[largest])
        largest = left;

    // If right child is larger than the largest so far
    if (right < heapSize && heap[right] > heap[largest])
        largest = right;

    // If the largest is not root
    if (largest != i) {
        swap(&heap[i], &heap[largest]);  // Swap root with the largest
        heapify(largest);  // Recursively heapify the affected sub-tree
    }
}

// Function to remove and return the maximum element (root)
int deleteMax() {
    if (heapSize == 0) {
        printf("Heap is empty\n");
        return -1;
    }

    int maxElement = heap[0];  // Root of the heap
    heap[0] = heap[heapSize - 1];  // Replace root with the last element
    heapSize--;  // Reduce the size of the heap

    heapify(0);  // Adjust the heap (heapify)

    return maxElement;  // Return the max element
}

// Function to display the heap
void displayHeap() {
    for (int i = 0; i < heapSize; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");
}
```

```
int main() {
    insert(10);
    insert(30);
    insert(20);
    insert(50);
    insert(40);

    printf("Heap after inserting elements: ");
    displayHeap();

    printf("Deleted max element: %d\n", deleteMax());
    printf("Heap after deleting max: ");
    displayHeap();

    return 0;
}
```

OUTPUT:

```
Heap after inserting elements: 50 40 20 10 30
Deleted max element: 50
Heap after deleting max: 40 30 20 10
```

RESULT :

Thus the program of implementation of Heap using priority Queue is executed & verified successfully.

Ex.No:	12	IMPLEMENTATION OF HASH FUNCTIONS AND COLLISION RESOLUTION TECHNIQUE
Date:		

AIM

To write a C Program for implementing Hash functions and Collision Resolution technique.

ALGORITHM

Step 1: Hash Function: Convert a key into an index in the hash table.
Step 2: Insert: If the index is empty, insert the key. If occupied, insert the key into the linked list at that index (Separate Chaining).
Step 3: Search: Use the hash function to find the index, then search through the linked list at that index.
Step 4: **Delete**: Use the hash function to find the index, then remove the key from the linked list.

PROGRAM : (Separate Chaining)

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

#define SIZE 10  // Size of the hash table

// Node structure for linked list (used in separate chaining)
struct Node {
    int key;
    struct Node* next;
};

// Hash table of linked lists
struct Node* hashTable[SIZE];

// Hash function (returns index based on the key)
int hashFunction(int key) {
    return key % SIZE;
}

// Insert a key into the hash table
void insert(int key) {
    int index = hashFunction(key);
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->next = hashTable[index];  // Insert at the beginning of the linked list
    hashTable[index] = newNode;
}

// Search for a key in the hash table
struct Node* search(int key) {
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];

    while (temp != NULL) {
        if (temp->key == key)
            return temp;
        temp = temp->next;
    }
    return NULL;
}

// Delete a key from the hash table
void deleteKey(int key) {
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];
    struct Node* prev = NULL;

    while (temp != NULL && temp->key != key) {
        prev = temp;
        temp = temp->next;
    }

    // Key not found
    if (temp == NULL) {
        printf("Key %d not found in the hash table.\n", key);
        return;
    }

    // Remove the node from the linked list
    if (prev == NULL) {
        hashTable[index] = temp->next;  // Remove the head node
    } else {
        prev->next = temp->next;
    }
    free(temp);
    printf("Key %d deleted from the hash table.\n", key);
}

// Print the hash table
void printHashTable() {
    for (int i = 0; i < SIZE; i++) {
        struct Node* temp = hashTable[i];
        printf("Index %d: ", i);
        while (temp != NULL) {
            printf("%d -> ", temp->key);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

int main() {
    // Initialize the hash table with NULL pointers
    for (int i = 0; i < SIZE; i++) {
        hashTable[i] = NULL;
    }

    insert(10);
    insert(20);
    insert(30);
    insert(40);
    insert(25);
    insert(35);

    printf("Hash table after insertion:\n");
    printHashTable();

    struct Node* found = search(30);
    if (found != NULL) {
        printf("Key 30 found in the hash table.\n");
    } else {
        printf("Key 30 not found in the hash table.\n");
    }

    deleteKey(25);
    printf("Hash table after deletion:\n");
    printHashTable();

    return 0;
}
```

PROGRAM : (Linear Probing)

```
  #include <stdio.h>

#define SIZE 10  // Size of the hash table

int hashTable[SIZE];  // Hash table array
int deletedMarker = -9999;  // Marker to indicate a deleted key

// Hash function (returns index based on the key)
int hashFunction(int key) {
    return key % SIZE;
}

// Insert a key into the hash table using linear probing
void insert(int key) {
    int index = hashFunction(key);

    // Linear probing
    while (hashTable[index] != 0 && hashTable[index] != deletedMarker) {
        index = (index + 1) % SIZE;  // Move to the next slot
    }

    hashTable[index] = key;
}

// Search for a key in the hash table
int search(int key) {
    int index = hashFunction(key);

    // Linear probing
    int startIndex = index;
    while (hashTable[index] != 0) {
        if (hashTable[index] == key)
            return index;  // Key found

        index = (index + 1) % SIZE;
        if (index == startIndex)
            break;  // We've looped around the entire table
    }

    return -1;  // Key not found
}
// Delete a key from the hash table using linear probing
void deleteKey(int key) {
    int index = search(key);
    if (index == -1) {
        printf("Key %d not found in the hash table.\n", key);
    } else {
        hashTable[index] = deletedMarker;
        printf("Key %d deleted from the hash table.\n", key);
    }
}
// Print the hash table
void printHashTable() {
    for (int i = 0; i < SIZE; i++) {
        if (hashTable[i] == 0) {
            printf("Index %d: NULL\n", i);
        } else if (hashTable[i] == deletedMarker) {
            printf("Index %d: DELETED\n", i);
        } else {
            printf("Index %d: %d\n", i, hashTable[i]);
        }
    }
}
int main() {
    // Initialize the hash table with 0 (indicating empty slots)
    for (int i = 0; i < SIZE; i++) {
        hashTable[i] = 0;
    }
    insert(10);
    insert(20);
    insert(30);
    insert(40);
    insert(25);
    insert(35);
    printf("Hash table after insertion:\n");
    printHashTable();

    int index = search(30);
    if (index != -1) {
        printf("Key 30 found at index %d in the hash table.\n", index);
    } else {
        printf("Key 30 not found in the hash table.\n");
    }
    deleteKey(25);
    printf("Hash table after deletion:\n");
    printHashTable();

    return 0;
}
```

OUTPUT : (Separate Chaining)

```
Hash table after insertion:
Index 0: NULL
Index 1: NULL
Index 2: NULL
Index 3: 30 -> NULL
Index 4: 40 -> NULL
Index 5: 25 -> NULL
Index 6: NULL
Index 7: 10 -> NULL
Index 8: 20 -> NULL
Index 9: 35 -> NULL

Key 30 found in the hash table.
Key 25 deleted from the hash table.
Hash table after deletion:
Index 0: NULL
Index 1: NULL
Index 2: NULL
Index 3: 30 -> NULL
Index 4: 40 -> NULL
Index 5: NULL
Index 6: NULL
Index 7: 10 -> NULL
Index 8: 20 -> NULL
Index 9: 35 -> NULL
```

OUTPUT : (Linear Probing)

```
Hash table after insertion:
Index 0: NULL
Index 1: NULL
Index 2: 20
Index 3: 30
Index 4: 40
Index 5: 25
Index 6: 35
Index 7: 10
Index 8: NULL
Index 9: NULL

Key 30 found at index 3 in the hash table.
Key 25 deleted from the hash table.
Hash table after deletion:
Index 0: NULL
Index 1: NULL
Index 2: 20
Index 3: 30
Index 4: 40
Index 5: DELETED
Index 6: 35
Index 7: 10
Index 8: NULL
Index 9: NULL
```

RESULT

Thus the program for implementation of Hash functions is executed & verified successfully.

CONTENT BEYOND THE SYLLABUS

Functions of Dictionary (ADT) Using Hashing

Ex.No:	**13**	**FUNCTIONS OF DICTIONARY (ADT) USING HASHING**
Date:		

AIM

To implement all the functions of a Dictionary (ADT) using hashing.

PROGRAM LOGIC

Step 1: Hash Function: Convert the key to an array index.
Step 2: **Insert**: Add a key-value pair. If the slot is occupied, handle the collision.
Step 3: Search: Use the key to find the associated value.
Step 4: Delete: Remove the key-value pair.

PROGRAM:

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

#define SIZE 10  // Size of the hash table

// Node structure for a linked list (used in separate chaining)
struct Node {
    int key;
    char value[100];
    struct Node* next;
};

// Hash table as an array of linked list heads
struct Node* hashTable[SIZE];

// Hash function (returns index based on the key)
int hashFunction(int key) {
    return key % SIZE;
}

// Insert a key-value pair into the dictionary
void insert(int key, char* value) {
    int index = hashFunction(key);
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    strcpy(newNode->value, value);
    newNode->next = hashTable[index];  // Insert at the beginning of the linked list
    hashTable[index] = newNode;
    printf("Inserted key %d with value '%s'\n", key, value);
}

// Search for a value by its key in the dictionary
void search(int key) {
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];

    while (temp != NULL) {
        if (temp->key == key) {
            printf("Key %d found with value '%s'\n", key, temp->value);
            return;
        }
        temp = temp->next;
    }

    printf("Key %d not found in the dictionary.\n", key);
}

// Delete a key-value pair from the dictionary
void deleteKey(int key) {
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];
    struct Node* prev = NULL;

    // Traverse the list to find the key
    while (temp != NULL && temp->key != key) {
        prev = temp;
        temp = temp->next;
    }

    // Key not found
    if (temp == NULL) {
        printf("Key %d not found in the dictionary.\n", key);
        return;
    }

    // Key found, remove the node
    if (prev == NULL) {
        hashTable[index] = temp->next;  // Remove head node
    } else {
        prev->next = temp->next;  // Bypass the node to be deleted
    }

    printf("Key %d with value '%s' deleted from the dictionary.\n", key, temp->value);
    free(temp);  // Free the memory
}

// Print the entire dictionary
void printDictionary() {
    for (int i = 0; i < SIZE; i++) {
        struct Node* temp = hashTable[i];
        printf("Index %d: ", i);
        while (temp != NULL) {
            printf("(%d, %s) -> ", temp->key, temp->value);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

int main() {
    // Initialize hash table
    for (int i = 0; i < SIZE; i++) {
        hashTable[i] = NULL;
    }

    // Insert key-value pairs
    insert(10, "Apple");
    insert(20, "Banana");
    insert(30, "Cherry");
    insert(40, "Date");
    insert(25, "Elderberry");

    // Display the dictionary
    printf("\nDictionary after insertions:\n");
    printDictionary();

    // Search for keys
    printf("\nSearching for key 30:\n");
    search(30);

    printf("\nSearching for key 50:\n");
    search(50);

    // Delete a key-value pair
    printf("\nDeleting key 25:\n");
    deleteKey(25);

    // Display the dictionary after deletion
    printf("\nDictionary after deletion:\n");
    printDictionary();

    return 0;
}
```

OUTPUT:

```
Inserted key 10 with value 'Apple'
Inserted key 20 with value 'Banana'
Inserted key 30 with value 'Cherry'
Inserted key 40 with value 'Date'
Inserted key 25 with value 'Elderberry'
Dictionary after insertions:
Index 0: NULL
Index 1: NULL
Index 2: NULL
Index 3: NULL
Index 4: (40, Date) -> NULL
Index 5: (25, Elderberry) -> NULL
Index 6: NULL
Index 7: (10, Apple) -> NULL
Index 8: (20, Banana) -> NULL
Index 9: (30, Cherry) -> NULL
Searching for key 30:
Key 30 found with value 'Cherry'

Searching for key 50:
Key 50 not found in the dictionary.

Deleting key 25:
Key 25 with value 'Elderberry' deleted from the dictionary.
Dictionary after deletion:
Index 0: NULL
Index 1: NULL
Index 2: NULL
Index 3: NULL
Index 4: (40, Date) -> NULL
Index 5: NULL
Index 6: NULL
Index 7: (10, Apple) -> NULL
Index 8: (20, Banana) -> NULL
Index 9: (30, Cherry) -> NULL
```

RESULT

Thus the program of Directory of ADT using Hashing is executed & verified successfully.

DATA STRUCTURES LABORATORY VIVA VOCE QUESTIONS

1. What is a Data Structure?

2. Define the types of Data Structures and give examples?

3. What is a Singly Linked List?

4. What is Doubly Linked List?

5. Differentiate Array and Linked List?

6. What is a Stack?

7. What is Stack Underflow?

8. What is Stack Overflow?

9. What are the Applications of Stack?

10. What is the use of Postfix expressions?

11. What is a Queue?

12. What are the applications of Queues?

13. What is a Circular Queue?

14. Differentiate Linear Queue and Circular Queue?

15. What is Dequeue?

16. What is a Tree?

17. What is a Binary Tree?

18. What is Tree Traversal? List different Tree Traversal Techniques?

19. **What is a Binary Search Tree? Give one example?**

20. **What is a Full (perfect) Binary Tree?**

21. **What is a Complete Binary Tree?**

22. **List one application of trees (Search trees)?**

23. **What is Priority Queue and differentiate Priority Queue and Queue?**

24. **What is a Graph? Name various Graph Representation Techniques?**

25. What is difference between a Graph and a Tree?

26. What is Linear Search?

27. What is Binary Search method?

28. What is Sorting?

29. What is Hashing?

30. List one application of Hashing.