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

Department of Electrical & Electronics Engineering

B23CSI303 – FUNDAMENTALS OF DATA STRUCTURES USING C LABORATORY

Name.....

Batch ..... Reg.No.....

Branch..... Year .....

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

Department of Electrical & Electronics Engineering

BONAFIDE CERTIFICATE

Name :……

Roll No. : …… Reg. No.: ……

Branch : B.E. Electrical & Electronics Engineering

Certified that this is Bonafide record work done by Mr./Ms. …… of

II – Year Electrical & Electronics Engineering during the academic year 2024-2025.

FACULTYIN-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

- The Department strives to become one of the recognized National centres with particular focus in Electrical and Electronics Engineering.

MISSION

- Quality Education
- Entrepreneurship Skill Development
- Result Oriented Research by Creating Research Centres
- Centre of Excellence
- Offering Problem Solving Community Services

PROGRAMME OUTCOMES (POs)

Students graduating from Electrical & Electronics Engineering should be able to:

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

PO2 Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3 Design/Development of 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.

PO4 Conduct Investigations of Complex Problems: Use research-based knowledge and research methods 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 modelling to complex 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.

PO7 Environment and Sustainability: Understand the impact of the professional engineering solutions in

societal and environmental contexts, and demonstrate the knowledge of, and need for 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 engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and 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

4. PROGRAMME SPECIFIC OUTCOMES(PSOs)

After the successful completion of UG programme in Electrical and Electronics Engineering graduates will be able to:

PSO1: Develop models, assess and analyze the components and systems that effectively generate, transmit and distribute electric power and protection mechanism in power systems.

PSO2: Design and test advanced electronics systems to perform analog and digital control and deploy control strategies for Power Electronics related and other applications.

5.PROGRAMME EDUCATIONAL OBJECTIVES(PEOS)

PEO1: Graduates of the programme will have successful technical or professional career.

PEO2: Graduates will demonstrate core competence in Electrical and Electronics Engineering and leadership qualities in their chosen fields of employment.

PEO3: Graduates will continue to learn and adapt in a world of constantly evolving technology.

COURSE OUTCOMES:

Students will be able to:

Course Outcome	Knowledge Level
CO1: Develop functions to implement linear and non-linear data structureOperations.	K3
CO2: Choose the appropriate linear data structures for solving a givenproblem.	K3
CO3: Select, implement and use the appropriate non-linear data structuresfor solving a given problem.	K3
CO4: Develop and design optimal algorithms for searching and sorting.	K3
CO5: Apply appropriate hash functions that result in a collision freescenario for data storage and retrieval.	K3

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	2	2	1	-	-	-	-	-	-	-		3	3
CO2	K3	3	3	3	3	-	-	-	-	-	-	-		3	2
CO3	K3	3	3	3	3	-	-	-	-	-	-	-		3	2
CO4	K3	3	1	3	2	-	-	-	-	-	-	-		2	3
CO5	K3	3	1	2	1	-	-	-	-	-	-	-		2	3
Weighted average		3	3	3	1	-	-	-	-	-	-	-		3	3

SYLLABUS

LIST OF EXPERIMENTS

1. **IMPLEMENTATION OF SINGLY, DOUBLY AND CIRCULAR LINKED LIST.**
2. **ARRAY IMPLEMENTATION OF STACK AND QUEUE ADTS.**
3. **LINKED LIST IMPLEMENTATION OF STACK AND QUEUE ADTS.**
4. **IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS.**
5. **IMPLEMENTATION OF BINARY SEARCH TREES.**
6. **IMPLEMENTATION OF AVL TREES.**
7. **IMPLEMENTATION OF SHORTEST PATH AND MST ALGORITHMS**
8. **IMPLEMENTATION OF SEARCHING ALGORITHMS.**
9. **IMPLEMENTATION OF SORTING ALGORITHMS.**
10. **HASHING – COLLISION RESOLUTION TECHNIQUES**

S.NO	EXPERIMENT	PREREQUISITS	LEARNING ONJECTIVES
	FOR ALL EXPERIMENTS	PROGRAMMING IN C	1. To understand the practical application of lineardata structures. 2. To understand the different operations of searchtrees. 3. To familiarize graphs and their applications. 4. To demonstrate different sorting and searchingtechniques. 5. To implement the different hashing techniques.

Practical Record Book Index Page

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

Model Exam Marks (25):_____ Total (100):_____ **Signature of the Faculty Member**

Practical Record Book Index Page

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

Model Exam Marks (25):_____ Total (100):_____ **Signature of the Faculty Member**

Practical Record Book Index Page

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

Model Exam Marks (25):_____ Total (100):_____ **Signature of the Faculty Member**

Ex. No:	1(A)	IMPLEMENTATION OF SINGLY LINKED LIST
Date:		

AIM

To Write a C program for Singly Linked List.

ALGORITHM

Step 1: Start the program
Step 2: Get the size of the list from user
Step 3: To create a new singly list
Step 4: Get the choice from the user.
Step 5: If the choice is to insert, then get the data from the user and add to the list.
Step 6: If the choice is to delete, then get the position from the user and delete it from the list.
Step 7: If the choice is to display, then print the list elements.
Step 8: If the choice is to exit, then exit the program.
Step 9: Stop the program.

PROGRAM

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

struct Node {
    int data;
    struct Node* next;
};

// Insert at the beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}

// Insert at the end
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
}

// Delete a node by value
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }

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

```
    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

// Display the list
void displayList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

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

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);

    printf("Singly Linked List:\n");
    displayList(head);

    deleteNode(&head, 20);

    printf("After deletion:\n");
    displayList(head);

    return 0;
}
```

OUTPUT

insertAtEnd(&head, 10);
insertAtEnd(&head, 20);
insertAtEnd(&head, 30);
insertAtEnd(&head, 40);
insertAtEnd(&head, 50);
displayList(head);
10 20 30 40 50
deleteNode(&head, 20);
deleteNode(&head, 40);
displayList(head);
10 30 50

INFERENCE

VIVA QUESTIONS

1. **What is a singly linked list?**

2. **How do you traverse a singly linked list?**

3. **How do you find the length of a singly linked list?**

4. **How do you insert a node at the end of a singly linked list?**

5. **How do you display all elements in a singly linked list?**

RESULT

Thus the program for singly linked list was executed successfully and verified.

Ex.No:	1.(B)	IMPLEMENTATION OF DOUBLY LINKED LIST
Date:		

AIM

To write a doubly linked list program

ALGORITHM

Step 1: Start the program.

Step 2: Get the size of the list from the user**.**

Step 3: To create a new doubly linked list**.**

Step 4: Get the choice from the user.

Step 5: If the choice is to insert, then get the data from the user and add to the list.

Step 6: If the choice is to delete, then get the position from the user and delete it from the list.

Step 7: If the choice is to display, then print the list elements.

Step 8: If the choice is to exit, then exit the program.

Step 9: Stop the program.

PROGRAM

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

struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};

// Insert at the beginning
void insertAtBeginning(struct Node** head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = *head;
   newNode->prev = NULL;
   if (*head != NULL)
      (*head)->prev = newNode;
   *head = newNode;
}

// Insert at the end
void insertAtEnd(struct Node** head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   if (*head == NULL) {
      newNode->prev = NULL;
      *head = newNode;
      return;
   }
   struct Node* temp = *head;
   while (temp->next != NULL)
      temp = temp->next;
   temp->next = newNode;
   newNode->prev = temp;
}

// Delete a node by value
void deleteNode(struct Node** head, int key) {
   struct Node* temp = *head;

   while (temp != NULL && temp->data != key)
      temp = temp->next;

   if (temp == NULL) return;
```

```
    if (temp->prev != NULL)
        temp->prev->next = temp->next;
    else
        *head = temp->next;

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    free(temp);
}

// Display the list
void displayList(struct Node* node) {
    while (node != NULL) {
        printf("%d <-> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

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

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);

    printf("Doubly Linked List:\n");
    displayList(head);

    deleteNode(&head, 20);

    printf("After deletion:\n");
    displayList(head);

    return 0;
}
```

OUTPUT

```
    // Inserting nodes at the beginning
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);

    // Inserting nodes at the end
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);

    // Display the list after insertions
    printf("After insertions:\n");
    displayList(head);

    // Deleting a node in the middle
    deleteNode(&head, 20);
    printf("After deleting 20:\n");
    displayList(head);

    // Deleting the head node
    deleteNode(&head, 10);
    printf("After deleting 10 (head):\n");
    displayList(head);

    // Deleting the last node
    deleteNode(&head, 40);
    printf("After deleting 40 (tail):\n");
    displayList(head);

    // Inserting again at both ends
    insertAtBeginning(&head, 50);
    insertAtEnd(&head, 60);
    printf("After reinserting 50 (beginning) and 60 (end):\n");
    displayList(head);

    // Deleting a non-existent node
    deleteNode(&head, 100);
    printf("After attempting to delete 100 (non-existent):\n");
    displayList(head);
```

```
After insertions:
20 <-> 10 <-> 30 <-> 40 <-> NULL

After deleting 20:
10 <-> 30 <-> 40 <-> NULL

After deleting 10 (head):
```

30 <-> 40 <-> NULL

After deleting 40 (tail):
30 <-> NULL

After reinserting 50 (beginning) and 60 (end):
50 <-> 30 <-> 60 <-> NULL

After attempting to delete 100 (non-existent):
50 <-> 30 <-> 60 <-> NULL

INFERENCE

VIVA QUESTIONS

1. **What is a doubly linked list, and how is it different from a singly linked list?**

2. **How many pointers does a node in a doubly linked list have? What are they used for?**

3. **What is the time complexity for insertion at the beginning of a doubly linked list?**

4. **State one advantage of a doubly linked list over a singly linked list.**

5. **What happens to the prev pointer of the head node in a doubly linked list?**

RESULT

Thus the program for Doubly linked list was executed successfully and verified.

Ex.No:	1.(C)	IMPLEMENTATION OF CIRCULAR LINKED LIST
Date:		

AIM

To implement Circular Linked list program.

ALGORITHM

Step 1: Start the program.

Step 2: Get the size of the list from the user.

Step 3: To create a new circular linked list.

Step 4: Get the choice from the user.

Step 5: If the choice is to insert, then get the data from the user and add to the list.

Step 6: If the choice is to delete, then get the position from the user and delete it from the list.

Step 7: If the choice is to display, then print the list elements.

Step 8: If the choice is to exit, then exit the program.

Step 9: Stop the program

PROGRAM

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

struct Node {
   int data;
   struct Node* next;
};

// Insert at the beginning
void insertAtBeginning(struct Node** head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   struct Node* temp = *head;
   newNode->data = data;
   newNode->next = *head;

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

// Insert at the end
void insertAtEnd(struct Node** head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   struct Node* temp = *head;
   newNode->data = data;
   newNode->next = *head;

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

// Delete a node by value
```

```
void deleteNode(struct Node** head, int key) {
   if (*head == NULL) return;

   struct Node *temp = *head, *prev = NULL;

   while (temp->data != key) {
      if (temp->next == *head) return;
      prev = temp;
      temp = temp->next;
   }

   if (temp == *head && temp->next == *head) {
      *head = NULL;
      free(temp);
      return;
   }

   if (temp == *head) {
      prev = *head;
      while (prev->next != *head)
         prev = prev->next;
      *head = temp->next;
      prev->next = *head;
      free(temp);
   } else if (temp->next == *head) {
      prev->next = *head;
      free(temp);
   } else {
      prev->next = temp->next;
      free(temp);
   }
}

// Display the list
void displayList(struct Node* head) {
   if (head == NULL) return;
   struct Node* temp = head;
   do {
      printf("%d -> ", temp->data);
      temp = temp->next;
   } while (temp != head);
   printf("HEAD\n");
}

int main() {
```

```
    struct Node* head = NULL;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);

    printf("Circular Linked List:\n");
    displayList(head);

    deleteNode(&head, 20);

    printf("After deletion:\n");
    displayList(head);

    return 0;
}
```

OUTPUT

```
// Inserting nodes at the beginning
insertAtBeginning(&head, 10);  // List: 10 -> HEAD
insertAtBeginning(&head, 20);  // List: 20 -> 10 -> HEAD

// Inserting nodes at the end
insertAtEnd(&head, 30);        // List: 20 -> 10 -> 30 -> HEAD
insertAtEnd(&head, 40);        // List: 20 -> 10 -> 30 -> 40 -> HEAD

// Display the list after insertions
printf("After insertions:\n");
displayList(head);

// Deleting a node from the middle
deleteNode(&head, 10);         // List: 20 -> 30 -> 40 -> HEAD
printf("After deleting 10:\n");
displayList(head);

// Deleting the head node
deleteNode(&head, 20);         // List: 30 -> 40 -> HEAD
printf("After deleting 20 (head):\n");
displayList(head);

// Deleting the last node
deleteNode(&head, 40);         // List: 30 -> HEAD
printf("After deleting 40 (tail):\n");
displayList(head);

// Deleting the remaining node
deleteNode(&head, 30);         // List: Empty
printf("After deleting 30 (last node):\n");
displayList(head);
```

```
After insertions:
20 -> 10 -> 30 -> 40 -> HEAD

After deleting 10:
20 -> 30 -> 40 -> HEAD

After deleting 20 (head):
30 -> 40 -> HEAD

After deleting 40 (tail):
30 -> HEAD

After deleting 30 (last node):
```

INFERENCE

VIVA QUESTIONS

1. What is a circular linked list, and how is it different from a singly linked list?

2. What is the key property of the next pointer in the last node of a circular linked list?

3. What is the time complexity of traversing a circular linked list?

4. What happens when you attempt to delete the only node in a circular linked list?

5. State one advantage of a circular linked list over a singly linked list.

RESULT

Thus the program for Circular linked list was executed successfully and verified.

Ex.No:	2.(A)	ARRAY IMPLEMENTATION OF STACK ADT
Date:		

AIM

To write a program for stack using array implementation.

ALGORITHM

Step1 : Define an array which stores stack element

Step 2: The operations on the stack are,

PUSH data into the stack

POP data out of stack

Step 3 : PUSH DATA INTO STACK

3a : Enter the data to be inserted into stack.

3b : If TOP is NULLthe input data is the first node in stack. The link of the node isNULL.TOP points to that node.

3c : If TOP is NOT NULLthe link of TOP points to the new node.TOP points to that node.

Step 4a:POP DATA FROM STACK 4a.If TOP is NULLthe stack is empty

4b:If TOP is NOT NULLthe link of TOP is the current TOP. The pervious TOP is popped from stack. .

Step 5: The stack represented by linked list is traversed to display its content.

PROGRAM

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

#define MAX 100 // Maximum size of the stack

typedef struct Stack {
    int data[MAX]; // Array to store stack elements
    int top;       // Index of the top element
} Stack;

// Initialize the stack
void initStack(Stack* stack) {
    stack->top = -1; // Stack is empty
}

// Check if the stack is full
int isFull(Stack* stack) {
    return stack->top == MAX - 1;
}

// Check if the stack is empty
int isEmpty(Stack* stack) {
    return stack->top == -1;
}

// Push an element onto the stack
void push(Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    stack->data[++stack->top] = value; // Increment top and add value
    printf("%d pushed onto stack.\n", value);
}

// Pop an element from the stack
int pop(Stack* stack) {
```

```
    if (isEmpty(stack)) {
        printf("Stack Underflow! Cannot pop.\n");
        return -1; // Return -1 to indicate underflow
    }
    return stack->data[stack->top--]; // Return top value and decrement top
}

// Peek at the top element of the stack
int peek(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Cannot peek.\n");
        return -1; // Return -1 to indicate empty stack
    }
    return stack->data[stack->top]; // Return top value
}

// Display all elements in the stack
void display(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = stack->top; i >= 0; i--)
        printf("%d ", stack->data[i]);
    printf("\n");
}

int main() {
    Stack stack;
    initStack(&stack);

    // Test the 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));
```

```
    display(&stack);

    return 0;
}
```

OUTPUT

```
// Push operations
push(&stack, 10);       // Stack: 10
push(&stack, 20);       // Stack: 10, 20
push(&stack, 30);       // Stack: 10, 20, 30

// Display stack
printf("Stack after pushing elements:\n");
display(&stack);

// Peek at the top element
printf("Top element is %d\n", peek(&stack));

// Pop operations
printf("Popped element is %d\n", pop(&stack)); // Stack: 10, 20
printf("Popped element is %d\n", pop(&stack)); // Stack: 10

// Display stack after popping
printf("Stack after popping elements:\n");
display(&stack);

// Push additional elements
push(&stack, 40);       // Stack: 10, 40
push(&stack, 50);       // Stack: 10, 40, 50

// Display stack
printf("Stack after pushing more elements:\n");
display(&stack);

// Edge case: Pop all elements and attempt another pop
printf("Popped element is %d\n", pop(&stack)); // Stack: 10, 40
printf("Popped element is %d\n", pop(&stack)); // Stack: 10
printf("Popped element is %d\n", pop(&stack)); // Stack: Empty
printf("Popped element is %d\n", pop(&stack)); // Stack Underflow!

// Edge case: Peek at an empty stack
printf("Peek at empty stack: %d\n", peek(&stack));

// Edge case: Push elements until stack is full
for (int i = 1; i <= MAX; i++) {
        push(&stack, i); // Attempt to fill the stack
}
```

```
10 pushed onto stack.
20 pushed onto stack.
30 pushed onto stack.
Stack after pushing elements:
Stack elements: 30 20 10
Top element is 30
Popped element is 30
Popped element is 20
Stack after popping elements:
Stack elements: 10
40 pushed onto stack.
50 pushed onto stack.
Stack after pushing more elements:
Stack elements: 50 40 10
Popped element is 50
Popped element is 40
Popped element is 10
Stack Underflow! Cannot pop.
Popped element is -1
Peek at empty stack: Stack is empty! Cannot peek.
Peek at empty stack: -1
1 pushed onto stack.
2 pushed onto stack.
...
Stack Overflow! Cannot push 101
```

INFERENCE

VIVA QUESTIONS

1. What is a stack, and what are its two primary operations?

2. What is the condition for a stack overflow in an array-based implementation?

3. What is the condition for a stack underflow in an array-based implementation?

4. What is the time complexity of the push and pop operations in a stack implemented using an array?

5. What is the role of the top variable in a stack implemented using an array?

RESULT

Thus, the above program Array implementation of stack is executed and the output isverified successfully.

Ex.No:2(B) Date:	ARRAY IMPLEMENTATION OF QUEUE ADT

AIM

To write a program for Queue using array implementation.

ALGORITHM

Step1: Define a array which stores queue elements.

Step2: The operations on the queue are

a. INSERT data into the queue

b. DELETE data out of queue

Step3: INSERT DATA INTO queue.

Enter the data to be inserted into queue.

If TOP is NULL the input data is the first node in queue The link of the node is NULL.TOP points to that node.

If TOP is NOT NULL the link of TOP points to the new node. TOP points to that node.

Step 4: DELETE DATA FROM queue

If TOP is NULL the queue is empty

If TOP is NOT NULL the link of TOP is the current TOP.

The pervious TOP is popped from queue.

Step 5: The queue represented by linked list is traversed to display its content.

PROGRAM

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

#define MAX 100 // Maximum size of the queue

typedef struct Queue {
    int front, rear, size;
    int data[MAX];
} Queue;

// Initialize the queue
void initializeQueue(Queue* q) {
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}

// Check if the queue is full
int isFull(Queue* q) {
    return q->size == MAX;
}

// Check if the queue is empty
int isEmpty(Queue* q) {
    return q->size == 0;
}

// Enqueue an element into the queue
void enqueue(Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full! Cannot enqueue %d\n", value);
        return;
    }
    q->rear = (q->rear + 1) % MAX; // Circular increment
    q->data[q->rear] = value;
    q->size++;
    printf("Enqueued: %d\n", value);
}

// Dequeue an element from the queue
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! Cannot dequeue\n");
        return -1; // Return -1 to indicate the queue is empty
    }
    int dequeuedValue = q->data[q->front];
```

```
    q->front = (q->front + 1) % MAX; // Circular increment
    q->size--;
    printf("Dequeued: %d\n", dequeuedValue);
    return dequeuedValue;
}

// Peek the front element of the queue
int peek(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! Nothing to peek\n");
        return -1; // Return -1 to indicate the queue is empty
    }
    return q->data[q->front];
}

// Display the queue elements
void displayQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue: ");
    for (int i = 0; i < q->size; i++) {
        int index = (q->front + i) % MAX;
        printf("%d ", q->data[index]);
    }
    printf("\n");
}

int main() {
    Queue q;
    initializeQueue(&q);

    // Test the queue implementation
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    displayQueue(&q);

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

    dequeue(&q);
    displayQueue(&q);

    enqueue(&q, 40);
    enqueue(&q, 50);
    displayQueue(&q);

    dequeue(&q);
```

```
    dequeue(&q);
    displayQueue(&q);

    return 0;

}
```

OUTPUT

```
// Enqueue operations
enqueue(&q, 10);        // Queue: 10
enqueue(&q, 20);        // Queue: 10, 20
enqueue(&q, 30);        // Queue: 10, 20, 30

// Display queue
printf("After enqueueing 10, 20, 30:\n");
displayQueue(&q);

// Peek at the front element
printf("Front element: %d\n", peek(&q)); // Should print 10

// Dequeue operations
dequeue(&q);            // Queue: 20, 30
printf("After dequeuing an element:\n");
displayQueue(&q);

// Enqueue additional elements
enqueue(&q, 40);        // Queue: 20, 30, 40
enqueue(&q, 50);        // Queue: 20, 30, 40, 50
printf("After enqueueing 40, 50:\n");
displayQueue(&q);

// Dequeue multiple elements
dequeue(&q);            // Queue: 30, 40, 50
dequeue(&q);            // Queue: 40, 50
printf("After dequeuing two elements:\n");
displayQueue(&q);

// Edge case: Dequeue until the queue is empty
dequeue(&q);            // Queue: 50
dequeue(&q);            // Queue: Empty
printf("After dequeuing all elements:\n");
displayQueue(&q);

// Edge case: Attempt to dequeue from an empty queue
dequeue(&q);            // Should print "Queue is empty!"

// Edge case: Fill the queue to its maximum capacity
printf("Filling the queue to its maximum capacity...\n");
for (int i = 1; i <= MAX; i++) {
        enqueue(&q, i);
}
```

```
// Attempt to enqueue into a full queue
enqueue(&q, 101);       // Should print "Queue is full!"
```

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
After enqueueing 10, 20, 30:
Queue: 10 20 30
Front element: 10
Dequeued: 10
After dequeuing an element:
Queue: 20 30
Enqueued: 40
Enqueued: 50
After enqueueing 40, 50:
Queue: 20 30 40 50
Dequeued: 20
Dequeued: 30
After dequeuing two elements:
Queue: 40 50
Dequeued: 40
Dequeued: 50
After dequeuing all elements:
Queue is empty!
Queue is empty! Cannot dequeue
Filling the queue to its maximum capacity...
Enqueued: 1
Enqueued: 2
...
Queue is full! Cannot enqueue 101
```

INFERENCE

VIVA QUESTIONS

1. What is a queue, and what are its two primary operations?

2. What is the condition for a queue overflow in an array-based implementation?

3. What is the condition for a queue underflow in an array-based implementation?

4. What are the advantages of implementing a queue using a circular array?

5. What is the time complexity of the enqueue and dequeue operations in an array-based queue?

RESULT

Thus, the above program Array implementation of queue is executed and the output is verified successfully.

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

AIM

To Write a program for implementing Stack using ADT.

ALGORITHM

Step 1: Initialize the data variables.
Step 2: In a switch case, in case 1 the number to be pushed is got
Step 3: Top is now equal to top+1
Step 4: Else the stack is full
Step 5: In case 2 the number to be deleted is got
Step 6: Top is now equal to top-1
Step 7: Else the stack is empty
Step 8: In case 3 if the top is less than 0 the stack is full Step 9: Else the stack is printed
Step 10: Default is no such choice.

PROGRAM

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

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

// Push a value onto the stack
void push(struct Node** top, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack Overflow! Unable to allocate memory.\n");
        return;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
    printf("%d pushed onto stack.\n", value);
}

// Pop a value from the stack
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow! The stack is empty.\n");
        return -1; // Return -1 to indicate underflow
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = temp->next;
    free(temp);
    return poppedValue;
}

// Peek at the top value of the stack
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty! Nothing to peek.\n");
        return -1; // Return -1 to indicate an empty stack
    }
    return top->data;
}
```

```
// Display the stack elements
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    struct Node* temp = top;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node* stack = NULL; // Initialize an empty stack

    // Test the stack implementation
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

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

    display(stack);

    printf("Popped: %d\n", pop(&stack));
    display(stack);

    printf("Popped: %d\n", pop(&stack));
    printf("Popped: %d\n", pop(&stack));
    printf("Popped: %d\n", pop(&stack)); // Attempt to pop from an empty stack

    return 0;
}
```

OUTPUT

```
// Push operations
push(&stack, 10);       // Stack: 10
push(&stack, 20);       // Stack: 20 -> 10
push(&stack, 30);       // Stack: 30 -> 20 -> 10
push(&stack, 40);       // Stack: 40 -> 30 -> 20 -> 10

// Display stack
printf("After pushing 10, 20, 30, 40:\n");
display(stack);

// Peek at the top element
printf("Top element after pushes: %d\n", peek(stack)); // Should print 40

// Pop operations
printf("Popped: %d\n", pop(&stack)); // Stack: 30 -> 20 -> 10
printf("Popped: %d\n", pop(&stack)); // Stack: 20 -> 10

// Display stack after pops
printf("Stack after popping twice:\n");
display(stack);

// Push additional elements
push(&stack, 50);       // Stack: 50 -> 20 -> 10
push(&stack, 60);       // Stack: 60 -> 50 -> 20 -> 10

// Display stack after new pushes
printf("Stack after pushing 50, 60:\n");
display(stack);

// Peek the top element
printf("Top element after additional pushes: %d\n", peek(stack)); // Should print 60

// Edge case: Pop all elements
while (stack != NULL) {
   printf("Popped: %d\n", pop(&stack));
}

// Attempt to pop from an empty stack
printf("Attempting to pop from an empty stack:\n");
pop(&stack); // Should print "Stack Underflow!"

// Peek from an empty stack
printf("Attempting to peek from an empty stack:\n");
peek(stack); // Should print "Stack is empty!"
```

10 pushed onto stack.
20 pushed onto stack.
30 pushed onto stack.
40 pushed onto stack.
After pushing 10, 20, 30, 40:
Stack elements: 40 30 20 10
Top element after pushes: 40
Popped: 40
Popped: 30
Stack after popping twice:
Stack elements: 20 10
50 pushed onto stack.
60 pushed onto stack.
Stack after pushing 50, 60:
Stack elements: 60 50 20 10
Top element after additional pushes: 60
Popped: 60
Popped: 50
Popped: 20
Popped: 10
Attempting to pop from an empty stack:
Stack Underflow! The stack is empty.
Attempting to peek from an empty stack:
Stack is empty! Nothing to peek.

INFERENCE

VIVA QUESTIONS

1. **What is a stack, and how is it implemented using a linked list?**

2. **What is the advantage of implementing a stack using a linked list over using an array?**

3. **What is the role of the top pointer in a stack implemented using a linked list?**

4. **What happens when you attempt to pop an element from an empty stack implemented using a linked list?**

5. **How does the peek operation work in a stack implemented using a linked list?**

RESULT

Thus the program for implementing Stack using ADT is executed successfully and verified.

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

AIM

To implement queue using linked list, we need to set the following things before implementing actual operations.

ALGORITHM

Step 1: Include all the header files which are used in the program. And declare all the userdefined functions.

Step 2: Define a 'Node' structure with two members data and next.

Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.

Step4: Implement the main method by displaying Menu of list of operations and makesuitable function calls in the main method to perform user selected operation. **enQueue(value) -** Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...nn.

Step 1: Create a new Node with given value and set 'new Node! next' to NULL.

Step 2: Check whether queue is Empty (rear == NULL)

Step 3: If it is Empty then, set front =new Node and rear =new Node.

Step 4: If it is Not Empty then, set rear ! next =new Node and rear =new Node.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

Step 4: Then set 'front =front ! next' and delete 'temp' (free(temp)).

PROGRAM

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

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

// Define a Queue structure
struct Queue {
    struct Node* front; // Pointer to the front of the queue
    struct Node* rear;  // Pointer to the rear of the queue
};

// Initialize the queue
void initializeQueue(struct Queue* q) {
    q->front = q->rear = NULL;
}

// Check if the queue is empty
int isEmpty(struct Queue* q) {
    return q->front == NULL;
}

// Enqueue an element into the queue
void enqueue(struct Queue* q, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    if (!newNode) {
        printf("Memory allocation failed! Unable to enqueue %d\n", value);
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (isEmpty(q)) {
        q->front = q->rear = newNode; // Queue is empty, both front and rear point to the new node
    } else {
        q->rear->next = newNode; // Link the new node to the rear
        q->rear = newNode;       // Update the rear pointer
    }
    printf("Enqueued: %d\n", value);
}

// Dequeue an element from the queue
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow! The queue is empty.\n");
        return -1; // Return -1 to indicate the queue is empty
    }

    struct Node* temp = q->front;
    int dequeuedValue = temp->data;

    q->front = q->front->next; // Update the front pointer
    if (q->front == NULL) {    // If the queue becomes empty
        q->rear = NULL;        // Reset the rear pointer as well
    }
```

```
    free(temp); // Free the dequeued node
    printf("Dequeued: %d\n", dequeuedValue);
    return dequeuedValue;
}

// Peek at the front element of the queue
int peek(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! Nothing to peek.\n");
        return -1; // Return -1 to indicate the queue is empty
    }
    return q->front->data;
}

// Display the queue elements
void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }

    printf("Queue elements: ");
    struct Node* temp = q->front;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

```
int main() {
    struct Queue q;
    initializeQueue(&q);

    // Test the queue implementation
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    displayQueue(&q);

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

    dequeue(&q);
    displayQueue(&q);

    enqueue(&q, 40);
    enqueue(&q, 50);
    displayQueue(&q);

    dequeue(&q);
    dequeue(&q);
    displayQueue(&q);

    return 0;
}
```

OUTPUT

Enqueued: 10

Enqueued: 20

Enqueued: 30

Queue elements: 10 20 30

Front element: 10

Dequeued: 10

Queue elements: 20 30

Enqueued: 40

Enqueued: 50

Queue elements: 20 30 40 50

Dequeued: 20

Dequeued: 30

Queue elements: 40 50

INFERENCE

VIVA QUESTIONS

1. **What is the difference between a queue implemented using an array and a queue implemented using a linked list?**

2. Explain how the enqueue operation works in a queue implemented using a linked list.

3. What happens when a dequeue operation is performed on an empty queue in a linked list implementation?

4. How do you ensure that the rear pointer is correctly updated when an element is dequeued in a queue implemented using a linked list?

5. What is the role of the front pointer in a queue implemented using a linked list?

RESULT

Thus the program for implementing queue using linked list was executed successfully and verified.

Ex. No: 4(A) Date:	IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS.

AIM

To write a C Program for implementing BFS traversal algorithms.

ALGORITHM

Inorder traversal :

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: INORDER(TREE -> LEFT)
Step 3: Write TREE -> DATA
Step 4: INORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END

Preorder traversal:

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER(TREE -> LEFT)
Step 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END

Post order traversal:

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER(TREE -> LEFT)
Step 3: POSTORDER(TREE -> RIGHT)
Step 4: Write TREE -> DATA
[END OF LOOP]
Step 5: END

PROGRAM

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

#define MAX 100 // Maximum number of vertices

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

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

// Check if the queue is empty
int isEmpty(struct Queue* q) {
    return q->front == -1;
}

// Enqueue an element into the queue
void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX - 1) {
        printf("Queue overflow\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->rear++;
    q->items[q->rear] = value;
}

// Dequeue an element from the queue
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue underflow\n");
        return -1;
    }
    int value = q->items[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1; // Reset the queue
```

```
    } else {
        q->front++;
    }
    return value;
}

// Perform BFS on the graph
void BFS(int graph[MAX][MAX], int startVertex, int numVertices) {
    int visited[MAX] = {0}; // Array to keep track of visited vertices
    struct Queue q;
    initQueue(&q);

    // Mark the start vertex as visited and enqueue it
    visited[startVertex] = 1;
    enqueue(&q, startVertex);

    printf("BFS Traversal: ");
    while (!isEmpty(&q)) {
        // Dequeue a vertex
        int currentVertex = dequeue(&q);
        printf("%d ", currentVertex);

        // Enqueue all adjacent unvisited vertices
        for (int i = 0; i < numVertices; i++) {
            if (graph[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(&q, i);
            }
        }
    }
    printf("\n");
}

int main() {
    int numVertices, startVertex;
    int graph[MAX][MAX];

    // Input number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);

    // Input adjacency matrix
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            scanf("%d", &graph[i][j]);
        }
```

```
    }

    // Input starting vertex
    printf("Enter the starting vertex (0-based index): ");
    scanf("%d", &startVertex);

    // Perform BFS
    BFS(graph, startVertex, numVertices);

    return 0;
}
```

OUTPUT

Enter the number of vertices: 4
Enter the adjacency matrix:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter the starting vertex (0-based index): 0

BFS Traversal: 0 1 2 3

INFERENCE

VIVA QUESTIONS

1. What is BFS, and how is it different from DFS?

2. What data structure is typically used to implement BFS?

3. What happens if there are cycles in the graph during BFS traversal?

4. How does BFS ensure that it visits all nodes in the shortest path in an unweighted graph?

5. How does BFS handle disconnected graphs?

RESULT

Thus the program for implementing breadth first search was executed successfully and verified.

Ex. No: 4(B) Date:	IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS.

AIM

To write a C Program for implementing DFS traversal algorithms.

ALGORITHM

Inorder traversal :

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: INORDER(TREE -> LEFT)
Step 3: Write TREE -> DATA
Step 4: INORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END

Preorder traversal:

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER(TREE -> LEFT)
Step 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END

Post order traversal:

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER(TREE -> LEFT)
Step 3: POSTORDER(TREE -> RIGHT)
Step 4: Write TREE -> DATA
[END OF LOOP]
Step 5: END

PROGRAM

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

// Define a structure for the adjacency list node
struct Node {
   int vertex;
   struct Node* next;
};

// Define a structure for the adjacency list
struct Graph {
   int numVertices;
   struct Node** adjLists;
   int* visited; // Array to track visited vertices
};

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

// Create a graph
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; // Initialize all vertices as unvisited
   }

   return graph;
}

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

```
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (for undirected graph)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Perform DFS traversal
void DFS(struct Graph* graph, int vertex) {
    graph->visited[vertex] = 1; // Mark the current vertex as visited
    printf("%d ", vertex);

    struct Node* temp = graph->adjLists[vertex];
    while (temp) {
        int connectedVertex = temp->vertex;

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

int main() {
    int vertices = 6;
    struct Graph* graph = createGraph(vertices);

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

    printf("DFS Traversal starting from vertex 0:\n");
    DFS(graph, 0);

    return 0;
}
```

OUTPUT

DFS Traversal starting from vertex 0:
0 2 5 1 4 3

INFERENCE

VIVA QUESTIONS

1. What is DFS and how does it traverse a graph?

2. What data structure is used to implement DFS?

3. How does DFS handle cycles in a graph?

4. What is the main difference between DFS and BFS?

5. What are the advantages of using DFS over BFS in graph traversal?

RESULT

Thus the program for implementing depth first search was executed successfully and verified.

Ex. No: 5 Date:	IMPLEMENTATION OF BINARY SEARCH TREES

AIM

To implement and demonstrate the operations of a Binary Search Tree (BST) including insertion, traversal (in-order, pre-order, and post-order), and searching.

ALGORITHM

Step 1: Start with an empty tree or a given root node.
Step 2: For insertion, start at the root node.
Step 3: Compare the value to be inserted with the current node's value.
If smaller, move to the left child.
If larger, move to the right child.
Step 4: Repeat step 3 until an empty position is found.
Step 5: Insert the new value at the empty position as a leaf node.
Step 6: For searching, start at the root node.
Step 7: Compare the target value with the current node:
If equal, the value is found.
If smaller, move to the left child.
If larger, move to the right child.
Step 8: Repeat step 7 until the value is found or a NULL node is reached.
Step 9: For traversal:
Use In-order (Left, Root, Right), Pre-order (Root, Left, Right), or Post-order (Left, Right, Root) traversal methods.
Step 10: Stop when all nodes are processed or the required operation is complete.

PROGRAM

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

// Define a Node structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

// Function to insert a node into the
BST
struct Node* insert(struct Node* root,
int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left,
data);
    } else if (data > root->data) {
        root->right = insert(root->right,
data);
    }
    return root;
}
```

```
// Function to perform in-order
traversal (Left, Root, Right)
void inOrder(struct Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

// Function to perform pre-order
traversal (Root, Left, Right)
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Function to perform post-order
traversal (Left, Right, Root)
void postOrder(struct Node* root) {
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}

// Function to search for a value in the
BST
int search(struct Node* root, int key)
{
    if (root == NULL) {
        return 0; // Key not found
    }
    if (root->data == key) {
        return 1; // Key found
```

```
    }
    if (key < root->data) {
        return search(root->left, key);
    }
    return search(root->right, key);
}

int main() {
    struct Node* root = NULL;
    int choice, value, key;

    do {
        printf("\nBinary Search Tree
Operations:\n");
        printf("1. Insert\n");
        printf("2. In-order Traversal\n");
        printf("3. Pre-order
Traversal\n");
        printf("4. Post-order
Traversal\n");
        printf("5. Search\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value to insert:
");
            scanf("%d", &value);
            root = insert(root, value);
            printf("%d inserted.\n",
value);
            break;

        case 2:
            printf("In-order Traversal: ");
            inOrder(root);
            printf("\n");
            break;
```

```
        case 3:
            printf("Pre-order Traversal: ");
            preOrder(root);
            printf("\n");
            break;

        case 4:
            printf("Post-order
Traversal:");
            postOrder(root);
            printf("\n");
            break;

        case 5:
            printf("Enter value to search:
");
            scanf("%d", &key);
            if (search(root, key)) {
                printf("%d found in the
BST.\n", key);
            } else {
                printf("%d not found in the
BST.\n", key);
            }
            break;

        case 6:
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice! Please
try again.\n");
        }
    } while (choice != 6);

    return 0;
}
```

OUTPUT

Binary Search Tree Operations:
1. Insert
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Search
6. Exit
Enter your choice: 1
Enter value to insert: 50
50 inserted.

Enter your choice: 1
Enter value to insert: 30
30 inserted.

Enter your choice: 1
Enter value to insert: 70
70 inserted.

Enter your choice: 2
In-order Traversal: 30 50 70

Enter your choice: 3
Pre-order Traversal: 50 30 70

Enter your choice: 4
Post-order Traversal: 30 70 50

Enter your choice: 5
Enter value to search: 30
30 found in the BST.

Enter your choice: 6
Exiting...

INFERENCE

VIVA QUESTIONS

1. What is a Binary Search Tree (BST)?

2. What are the key properties of a Binary Search Tree?

3. What is the time complexity of searching in a Binary Search Tree?

4. How do you insert a node in a Binary Search Tree?

5. What is the in-order traversal of a Binary Search Tree, and what does it produce?

RESULT

Thus the program for implementing binary search tree was executed and verified successfully.

Ex. No : 6 Date:	IMPLEMENTATION OF AVL TREE

AIM

To Write a C Program for implementing AVL tree operations.

ALGOGITHM

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2: After insertion, check the Balance Factor of every node.
Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

PROGRAM

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

// Define a structure for an AVL tree node
struct Node {
   int key;
   struct Node* left;
   struct Node* right;
   int height;
};

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

// Function to calculate the balance factor of a node
int getBalance(struct Node* node) {
   return (node == NULL) ? 0 : height(node->left) - height(node->right);
}

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

// Right rotate
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 = 1 + ((height(y->left) > height(y->right)) ? height(y->left) : height(y->right));
   x->height = 1 + ((height(x->left) > height(x->right)) ? height(x->left) : height(x->right));

   return x; // New root
}

// Left rotate
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 = 1 + ((height(x->left) > height(x->right)) ? height(x->left) : height(x->right));
    y->height = 1 + ((height(y->left) > height(y->right)) ? height(y->left) : height(y->right));

    return y; // New root
}

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

    if (key < node->key) {
        node->left = insert(node->left, key);
    } else if (key > node->key) {
        node->right = insert(node->right, key);
    } else {
        return node; // Duplicates not allowed
    }

    // Update the height of the current node
    node->height = 1 + ((height(node->left) > height(node->right)) ? height(node->left) : height(node->right));

    // Get the balance factor to check if the node is balanced
    int balance = getBalance(node);

    // If unbalanced, perform rotations

    // 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 node; // Return the unchanged node
}

// In-order traversal
void inOrder(struct Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->key);
        inOrder(root->right);
    }
}

// Pre-order traversal
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

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

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
    printf("In-order traversal of the AVL tree:\n");
    inOrder(root);
    printf("\nPre-order traversal of the AVL tree:\n");
    preOrder(root);
    return 0;
}
```

OUTPUT

In-order traversal of the AVL tree:
10 20 25 30 40 50

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

INFERENCE

VIVA QUESTIONS

1. **What is an AVL tree?**

2. What is the balance factor in an AVL tree?

3. How do you balance an AVL tree after an insertion?

4. What is a left rotation in an AVL tree?

5. What is a right rotation in an AVL tree?

RESULT

Thus the program for AVL implementation was executed successfully and Verified.

EX. NO: 7(A) DATE:	IMPLEMENTATION OF SHORTEST PATH ALGORITHM

AIM

To write a C program for implementing shortest path Dijkstra's algorithm.

ALGORITHM :

Step 1: Create an array **dist[]** to store the shortest distance from the source to each vertex.
Step 2: Initialize all distances as INFINITE except the source vertex (set to 0).
Step 3: Create an array **included[]** to track which vertices are included in the shortest path tree.
Step 4: Implement a function **findminDistance() t**o find the vertex with the minimum distance value from the set of vertices that are not yet included in the shortest path tree.
Step 5: Repeat the following steps V-1 times (where V is the number of vertices):
Step 5(a): Call findminDistance() to select the vertex 'u' with the minimum distance from the source, among the vertices not yet processed.
Step 5(b): Mark 'u' as included by setting included[u] = 1.
Step 5(c): For each adjacent vertex 'v' of 'u':
Step 5(c.i) If 'v' is not included in the shortest path tree and there's an edge from 'u' to 'v':
Step 5(c.ii) If the total distance from the source to 'u' plus the weight of the edge (u,v) is less than the current distance to 'v', update the distance of'v'.
Step 6: After the loop completes, the dist[] array will contain the shortest distances from the source to all vertices.
Step 7: Print the final dist[] array to show the shortest distance from the source to each vertex in the graph .

PROGRAM

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

#define V 9 // Number of vertices in the graph

// Function to find the vertex with the minimum distance value
int minDistance(int dist[], int visited[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++) {
      if (!visited[v] && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   }

   return min_index;
}

// Function to print the shortest distance from the source to all vertices
void printSolution(int dist[]) {
   printf("Vertex\tDistance from Source\n");
   for (int i = 0; i < V; i++) {
      printf("%d\t\t%d\n", i, dist[i]);
   }
}

// Function to implement Dijkstra's shortest path algorithm
void dijkstra(int graph[V][V], int src) {
   int dist[V];     // Array to store the shortest distance from src to i
   int visited[V];  // visited[i] is true if vertex i is included in the shortest path tree

   // Initialize distances as INFINITE and visited as false
   for (int i = 0; i < V; i++) {
      dist[i] = INT_MAX;
      visited[i] = 0;
   }

   dist[src] = 0; // Distance of source vertex from itself is always 0

   // Find the shortest path for all vertices
   for (int count = 0; count < V - 1; count++) {
      // Pick the minimum distance vertex from the set of unvisited vertices
      int u = minDistance(dist, visited);
```

```
 // Mark the picked vertex as visited
        visited[u] = 1;

        // Update dist value of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            // Update dist[v] if:
            // 1. It's not visited.
            // 2. There's an edge from u to v.
            // 3. Total weight of path from src to v through u is smaller than the current dist[v].
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Print the shortest distance array
    printSolution(dist);
}

int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    int src = 0; // Source vertex
    dijkstra(graph, src);

    return 0;
}
```

OUTPUT

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

INFERENCE

VIVA QUESTIONS

1. What is Dijkstra's algorithm used for?

2. What data structure is commonly used to implement Dijkstra's algorithm efficiently?

3. What is the time complexity of Dijkstra's algorithm using a binary heap?

4. Can Dijkstra's algorithm handle graphs with negative edge weights?

5. Explain the working of Dijkstra's algorithm.

RESULT

Thus, the Dijkstra’s algorithm implementation was implemented successfully.

EX. NO: 7(B) DATE:	IMPLEMENTATION OF MST ALGORITHM

Aim:

To write a C program to implement Prims's algorithm

Algorithm:

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: Return the MST and exit

PROGRAM

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

#define V 5 // Number of vertices in the graph

// Function to find the vertex with the minimum key value
int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++) {
      if (!mstSet[v] && key[v] < min) {
         min = key[v];
         min_index = v;
      }
   }

   return min_index;
}

// Function to print the constructed MST
void printMST(int parent[], int graph[V][V]) {
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++) {
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
   }
}

// Function to construct and print MST using Prim's algorithm
void primMST(int graph[V][V]) {
   int parent[V];   // Array to store the MST
   int key[V];      // Key values to pick the minimum weight edge
   int mstSet[V];   // mstSet[i] is true if vertex i is included in the MST

   // Initialize all keys to infinity and mstSet[] to false
   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = 0;
   }

   key[0] = 0;       // Start from the first vertex
   parent[0] = -1;   // First node is the root of the MST

   // The MST will have V-1 edges
   for (int count = 0; count < V - 1; count++) {
```

```
        // Pick the minimum key vertex that is not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST set
        mstSet[u] = 1;

        // Update key values and parent index of the adjacent vertices
        for (int v = 0; v < V; v++) {
            // Update key[v] only if:
            // 1. There's an edge from u to v.
            // 2. v is not yet included in MST.
            // 3. The weight of the edge is smaller than key[v].
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the MST
    printMST(parent, graph);
}

int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0},
    };

    // Find and print the MST using Prim's algorithm
    primMST(graph);

    return 0;
}
```

OUTPUT:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

INFERENCE

VIVA QUESTIONS

1. **What is a Minimum Spanning Tree?**

2. What is the purpose of Prim's algorithm?

3. Explain how Prim’s algorithm works.

4. What data structure is commonly used to implement Prim's algorithm efficiently?

5. How does Prim's algorithm differ from Kruskal's algorithm?

RESULT

Thus MST and shortest path algorithm implementation was implemented successfully.

Ex.No:8 Date:	IMPLEMENTATION OF SEARCHING ALGORITHM

AIM

To find a given target number using linear search from the list of numbers.

ALGORITHM

Step 1: Initialize the integer variables.

Step 2: Get the target number from User.

Step 3: Get the list of numbers to be searched.

Step 4: If the counter is equal to target the print the location.

Step 5: If it is not equal to then print that the location is not found.

PROGRAM

```
#include <stdio.h>

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

int main() {
    int n, target;

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

    int arr[n];

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

    // Input the target number to search for
    printf("Enter the target number to search: ");
    scanf("%d", &target);

    // Perform linear search
    int result = linearSearch(arr, n, target);

    // Output the result
    if (result != -1) {
```

```
        printf("Target number %d found at index %d.\n", target, result);
    } else {
        printf("Target number %d not found in the array.\n", target);
    }

    return 0;
}
```

OUTPUT

Enter the number of elements in the array: 5
Enter the elements of the array:
10 20 30 40 50

Enter the target number to search: 30
Target number 30 found at index 2.

Enter the target number to search: 60
Target number 60 not found in the array.

INFERENCE

VIVA QUESTIONS

1. What is Linear Search?

2. What is the time complexity of Linear Search?

3. In which type of data structure is Linear Search most suitable?

4. How does Linear Search differ from Binary Search?

5. What are the advantages and disadvantages of Linear Search?

RESULT

Thus the program for implementing linear search was executed and verified successfully.

Ex.No: 9 Date:	IMPLEMENTATION OF SORTING ALGORITHM

AIM

To sort the given data using Insertion sort.

ALGORITHM

Step 1: Initialize the variables i,j,n, curr and temp.

Step 2: Get the number of elements.

Step 3: Get the numbers to be sorted.

Step 4: If the second element is less than the first element then assign it to temp.

Step 5: Assign the first element to second element.

Step 6: Assign Temp to first element.

Step 7: The swapping is done.

Step 8: The data is sorted

PROGRAM

```
#include <stdio.h>

// Function to perform insertion sort
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 of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Function to display the array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n;

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

    int arr[n];

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

    // Sort the array using insertion sort
    printf("Sorting the array using Insertion Sort...\n");
    insertionSort(arr, n);

    // Display the sorted array
    printf("Sorted array:\n");
    displayArray(arr, n);

    return 0;
}
```

OUTPUT

```
Enter the number of elements in the array: 5
Enter the elements of the array:
34 8 64 51 32

Sorting the array using Insertion Sort...
Sorted array:
8 32 34 51 64
```

INFERENCE

VIVA QUESTIONS

1. **What is a sorting algorithm?**

2. **What are the different types of sorting algorithms?**

3. **What is the main idea behind Insertion Sort?**

4. **Why is Insertion Sort considered inefficient for large datasets?**

5. **What are the advantages of using Insertion Sort over other sorting algorithms like Bubble Sort or Selection Sort?**

RESULT

Thus the program for implementing insertion sort was executed and verified successfully.

Ex.No: 10 Date :	HASHING TECHNIQUES

AIM

To write a C Program for implementing hashing techniques using separate chaining.

ALGORITHM

Step 1: Start the program.

Step 2: Perform the following stages.

m = Number of slots in hash table

n = Number of keys to be inserted in hash table Load factor $\alpha = n/m$

Expected time to search = $O(1 + \alpha)$ Expected time to insert/delete = $O(1 + \alpha)$

Time complexity of search insert and delete is $O(1)$ if α is $O(1)$

Step 3: Stop the program.

PROGRAM

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

#define TABLE_SIZE 10 // Size of the hash table

// Structure for a node in the linked list
struct Node {
    int data;
    struct Node* next;
};

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

// Function to initialize the hash table
void initializeTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }
}

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Function to 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->data = key;
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
    printf("Inserted %d at index %d\n", key, index);
}

// Function to search for a key in the hash table
int search(int key) {
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];
    while (temp != NULL) {
        if (temp->data == key) {
            return 1; // Key found
```

```
        }
        temp = temp->next;
    }
    return 0; // Key not found
}

// Function to display the hash table
void displayTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Index %d: ", i);
        struct Node* temp = hashTable[i];
        while (temp != NULL) {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

int main() {
    initializeTable();

    // Insert elements into the hash table
    insert(12);
    insert(22);
    insert(32);
    insert(42);
    insert(52);

    // Display the hash table
    printf("\nHash Table:\n");
    displayTable();

    // Search for elements
    int key = 22;
    if (search(key)) {
        printf("\nKey %d found in the hash table.\n", key);
    } else {
        printf("\nKey %d not found in the hash table.\n", key);
    }

    key = 15;
    if (search(key)) {
        printf("Key %d found in the hash table.\n", key);
    } else {
        printf("Key %d not found in the hash table.\n", key);
```

```
}

return 0; }
```

INFERENCE

VIVA QUESTONS

1. **What is hashing?**

2. **What is the difference between hash tables and hashing?**

3. **What is hash collision?**

4. **What are the basic operations for a hash table?**

5. **What are the types of hashing?**

RESULT:

Thus the program for implementing hashing techniques was executed and verified successfully