SRM VALLIAMMAI ENGINEERINGCOLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203.

DEPARTMENT OF INFORMATION TECHNOLOGY

Regulation 2023 - Lab Manual *for* III Semester

IT3363 PROGRAMMING AND DATA STRUCTURES LABORATORY

Academic Year 2024-2025 (ODD Semester)

Prepared by

Mr. A. Aswin Jeba Mahir, Assistant Professor (O.G) / IT

IT3363 PROGRAMMING AND DATA STRUCTURES LABORATORY L T P C

0 0 3 1.5

OBJECTIVES:

- To develop C programs using basic constructs.
- To implement Linear Data Structures.
- To implement Non-Linear Data Structures.
- To implement Tree Traversal Algorithms.
- To implement Graph Traversal Algorithms.

LIST OF EXPERIMENTS:

1. Implement a C program using I/O Statements, Operators, and Expressions
2. a. Decision-making constructs: if-else, goto, switch-case, break-continue
   b. Loops: for, while, do-while
3. Arrays: 1D and 2D, multi-dimensional arrays, traversal
4. Array implementation of Stack, Queue, and Circular Queue ADTs
5. Implementation of Singly Linked List
6. Linked list implementation of Stack and Linear Queue ADTs
7. Implementation of Polynomial Manipulation using Linked list
8. Implementation of Evaluating Postfix Expressions, Infix to Postfix conversion
9. Implementation of Binary Search Trees
10. Implementation of Tree Traversal Algorithms
11. Implementation of Graph Traversal Algorithms
12. Implementation of Dijkstra's Algorithm

TOTAL: 45 PERIODS

OUTCOMES:

At the end of the course, the student should be able to:

- Develop C programs for real-world problems
- Implement Linear Data Structures and their applications.
- Implement Non-Linear Data Structures and their applications.

- Implement Binary Search tree operations.
- Implement graph algorithms.

SOFTWARE REQUIREMENTS

Operating Systems: Linux / Windows 7 or higher

Software: Turbo C / C / C++ / Equivalent Software.

CO – PO – PSO Mapping

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
IT3363.1	3	-	-	-	2	-	-	-	-	-	-	-	3	2	-	-
IT3363.2	2	2	-	-	-	-	-	-	-	-	-	-	3	2	-	-
IT3363.3	2	2	-	-	-	-	-	-	-	-	-	-	3	-	-	-
IT3363.4	3	-	-	-	-	-	-	-	-	-	-	-	3	2	-	-
IT3363.5	3	3	3	-	2	-	-	-	-	-	-	-	3	2	-	-
IT3363	2.6	2.3	3	-	2.0	-	-	-	-	-	-	-	3	2	-	-

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1. To afford the necessary background in the field of Information Technology to deal with engineering problems to excel as engineering professionals in industries.
2. To improve the qualities like creativity, leadership, teamwork, and skill thus contributing towards the growth and development of society.
3. To develop the ability among students towards innovation and entrepreneurship that caters to the needs of Industry and society.
4. To inculcate an attitude toward a life-long learning process through the use of information technology sources.
5. To prepare them to be innovative and ethical leaders, both in their chosen profession and in other activities.

PROGRAM OUTCOMES (POs)

After going through the four years of study, Bachelor of Technology in Information Technology Graduates will exhibit the ability to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to solve complex engineering problems.

2. **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.
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.
8. **Ethics**: Apply ethical principles and commit to professional ethics, responsibilities, and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **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.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one‘s work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **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 SPECIFIC OBJECTIVES (PSOs)

By the completion Bachelor of Technology in Information Technology program the student will have the following Program specific outcomes.

1. Design secured database applications involving planning, development, and maintenance using state-of-the-art methodologies based on ethical values.
2. Design and develop solutions for modern business environments coherent with advanced technologies and tools.
3. Design, plan, and set up a network that is helpful for contemporary business environments using the latest hardware components.
4. Planning and defining test activities by preparing test cases that can predict and correct errors ensuring a socially transformed product catering all technological needs.

COURSE OUTCOMES:

Course Name: IT3363 - PROGRAMMING AND DATA STRUCTURES LABORATORY

Year of study: 2024 –2025

IT3363.1	Develop C programs for real-world problems
IT3363.2	Implement Linear Data Structures and their applications.
IT3363.3	Implement Non-Linear Data Structures and their Applications.
IT3363.4	Implement Binary Search tree operations.
IT3363.5	Implement graph algorithms.

CO-PO Matrix:

CO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
IT3363.1	3	-	-	-	2	-	-	-	-	-	-	-
IT3363.2	2	2	-	-	-	-	-	-	-	-	-	-
IT3363.3	2	2	-	-	-	-	-	-	-	-	-	-
IT3363.4	3	-	-	-	-	-	-	-	-	-	-	-
IT3363.5	3	3	3	-	2	-	-	-	-	-	-	-

Justification:

Course Outcome	Program Outcome	Value	Justification
IT3363.1	PO1	3	Students will be able to write, debug, and execute programs in a chosen programming language
	PO5	2	Students will develop a strong understanding of core data structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables.
IT3363.2	PO1	2	Students will be able to implement and analyze fundamental algorithms for searching, sorting, and traversal.
	PO2	2	Students will enhance their problem-solving skills by applying appropriate data structures and algorithms to solve computational problems.
IT3363.3	PO1	2	Students will gain practical experience by working on projects that require the application of programming and data structure concepts.
	PO2	2	Students will develop collaboration skills by working in teams to complete lab assignments and projects.
IT3363.4	PO1	3	Students will learn effective debugging techniques and how to write test cases to ensure their code is robust and error-free.
IT3363.5	PO1	3	Students will be able to apply theoretical concepts learned in lectures to practical scenarios in the laboratory.
	PO2	3	Students will learn how to optimize code for performance, including memory management and efficient algorithm design.
	PO3	3	Develop the ability to analyze problems, design algorithms, and implement solutions using appropriate data structures and programming techniques.
	PO5	2	Apply data structures to solve real-world problems and understand their practical applications.

CO-PO Average:

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
IT3363	2.6	2.3	3	-	2.0	-	-	-	-	-	-	-

CO-PSO Matrix:

CO	PSO1	PSO2	PSO3	PSO4
IT3363.1	3	2	-	-
IT3363.2	3	2	-	-
IT3363.3	3	-	-	-
IT3363.4	3	2	-	-
IT3363.5	3	2	-	-

Justification:

Course Outcome	PSO	Value	Justification
IT3363.1	PSO1	3	Design and develop applications with C using state of the art methodologies based on ethical values.
	PSO2	2	Teach students the importance of writing efficient and optimized code.
IT3363.2	PSO1	3	Provide students with the ability to apply appropriate data structures to solve practical problems.
	PSO2	2	Improve students' problem-solving skills and critical thinking through the application of programming and data structures.
IT3363.3	PSO1	3	Exhibit a deep understanding of fundamental and advanced data structures and algorithms, applying this knowledge to design, analyze, and implement robust software solutions.
IT3363.4	PSO1	3	Students will be well-prepared for advanced studies and research in computer science and related fields, building on their solid foundation in programming and data structures.
	PSO2	2	Apply their programming and data structures knowledge to solve real-world problems, contributing effectively to industry projects and societal challenges.
IT3363.5	PSO1	3	Develop critical problem-solving and analytical thinking abilities, enabling them to tackle complex computational problems and devise innovative solutions.
	PSO2	2	Instill best practices in software development, including code readability, maintainability, and version control usage.

CO-PSO Average:

CO	**PSO 1**	**PSO 2**	**PSO 3**	**PSO 4**
IT3363	3	2	-	-

ASSESSMENT METHOD

i) EVALUATION PROCEDURE FOR EACH EXPERIMENTS

S.No	Description	Mark
1.	Aim & Pre-Lab discussion	20
2.	Observation	20
3.	Conduction and Execution	30
4.	Output & Result	10
5.	Viva	20
	Total	**100**

ii) INTERNAL ASSESSMENT FOR LABORATORY

S.No	Description	Mark
1.	Conduction & Execution of Experiment	25
2.	Record	10
3.	Model Test	15
	Total	**50**

TABLE OF CONTENTS

***Topic beyond the syllabus**

Ex.No:1 PROGRAM USING I/O STATEMENTS, OPERATORS, AND EXPRESSIONS

AIM
To write a C Program to perform I/O statements, Operators, and Expressions for Arithmetic operations.

PRE-LAB DISCUSSION:

C is a general-purpose programming language that is widely used for system programming, developing operating systems, and embedded system applications. Understanding C provides a solid foundation for learning other programming languages and concepts.

Input and Output in C

- **printf Function:** Used to output/display information on the screen.
  - Example: printf("Hello, World!\n");
  - Format specifiers like %d for integers, %f for floats, and %c for characters are used to format the output.
- **scanf Function:** Used to read/input data from the user.
  - Example: scanf("%d", &num);
  - The & symbol is used to store the input value in the variable's address.

Operators in C

1. **Arithmetic Operators:** Used for basic mathematical operations.
   - + (Addition)
   - - (Subtraction)
   - * (Multiplication)
   - / (Division)
   - % (Modulus)
2. **Relational Operators:** Used to compare two values.
   - == (Equal to)
   - != (Not equal to)
   - > (Greater than)
   - < (Less than)
   - >= (Greater than or equal to)
   - <= (Less than or equal to)
3. **Logical Operators:** Used to perform logical operations.
   - && (Logical AND)
   - || (Logical OR)
   - ! (Logical NOT)
4. **Assignment Operators:** Used to assign values to variables.
   - = (Assignment)

- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

Expressions in C

An expression is a combination of variables, constants, and operators that yields a value. For example, in the expression a + b, a and b are operands, and + is the operator.

ALGORITHM

1. Start
2. Declare variables and initializations
3. Read the Input variable.
4. Using I/O statements and expressions for computational processing.
5. Display the output of the calculations.
6. Stop

PROGRAM

This program takes two integer inputs from the user, performs various arithmetic operations, and displays the results.

```
#include <stdio.h>

int main() {
    int num1, num2;
    int sum, difference, product, quotient, remainder;

    // Input: Read two integers from the user
    printf("Enter the first integer: ");
    scanf("%d", &num1);

    printf("Enter the second integer: ");
    scanf("%d", &num2);

    // Operators and Expressions
    sum = num1 + num2;
    difference = num1 - num2;
    product = num1 * num2;
    quotient = num1 / num2;
    remainder = num1 % num2;

    // Output: Display the results of the operations
    printf("Sum: %d + %d = %d\n", num1, num2, sum);
    printf("Difference: %d - %d = %d\n", num1, num2, difference);
    printf("Product: %d * %d = %d\n", num1, num2, product);
    printf("Quotient: %d / %d = %d\n", num1, num2, quotient);
    printf("Remainder: %d %% %d = %d\n", num1, num2, remainder);

    return 0;
}
```

Output:

```
Enter the first integer: 12
Enter the second integer: 5
Sum: 12 + 5 = 17
Difference: 12 - 5 = 7
Product: 12 * 5 = 60
Quotient: 12 / 5 = 2
Remainder: 12 % 5 = 2
```

EXPLANATION

i) Input Statements:

```
printf("Enter the first integer: ");
scanf("%d", &num1);
```

These statements prompt the user to enter an integer and store it in num1.

ii) Operators and Expressions:

```
sum = num1 + num2;
difference = num1 - num2;
product = num1 * num2;
quotient = num1 / num2;
remainder = num1 % num2;
```

These statements perform arithmetic operations on num1 and num2 and store the results in corresponding variables.

iii) Output Statements:

```
printf("Sum: %d + %d = %d\n", num1, num2, sum);
```

These statements display the results of the arithmetic operations in a formatted manner.

VIVA QUESTIONS

1. How does scanf() handle different data types (integers, floats, characters)?
2. What are format specifiers in printf and how are they used?
3. Discuss the use of formatted I/O functions (fprintf, fscanf, etc.) for more advanced file operations in C.
4. What factors should you consider when choosing between different I/O functions (printf, scanf, getchar, putchar, file I/O)?
5. What is operator precedence in C? How does it affect expression evaluation?

Result:

Thus, a C Program using i/o statements and expressions was executed and the output was obtained.

Ex.No:2a PROGRAM USING DECISION-MAKING CONSTRUCTS

AIM

To understand and implement various decision-making constructs in C programming: if-else, goto, switch-case, break, and continue.

PRE-LAB DISCUSSION

Decision-Making Constructs in C Programming

Decision-making constructs allow programs to make decisions and perform different actions based on conditions. Understanding these constructs is fundamental for controlling program flow and logic.

i) if-else Statement:

Purpose: Executes a block of code if a specified condition is true, otherwise executes an alternative block of code.

Syntax:

```
if (condition) {
   // Code to be executed if condition is true
} else {
   // Code to be executed if condition is false
}
```

ii) goto Statement:

Purpose: Unconditionally transfers control to a labeled statement in the same function.

Syntax:

```
goto label;
//...
label:
// Statement(s)
```

iii) switch-case Statement:

Purpose: Unconditionally transfers control to a labeled statement in the same function.

Syntax:

```
switch (expression) {
   case value1:
      // Code to be executed if the expression matches value1
      break;
   case value2:
      // Code to be executed if the expression matches value2
      break;
   default:
        // Code to be executed if the expression doesn't match any case
}
```

iv) break and continue Statements:

- **break**:

Purpose: Terminates the current loop or switch-case statement.
Used within loops (for, while, do-while) or switch-case to exit the block.

- **continue**:

Purpose: Skips the remaining code inside loop and proceeds with the next iteration.
Used to jump to the next iteration in loops without executing the remaining code.

ALGORITHM

1. Start
2. Declare variables and initializations
3. Read the Input variable.
4. Using Decision making constructs for computational processing.
5. Display the output.
6. Stop

PROGRAM

```
#include <stdio.h>

int main() {
    int num,choice,i;

    // if-else statement
    printf("Enter a number: ");
    scanf("%d", &num);

    if (num > 0) {
        printf("%d is positive.\n", num);
    } else {
        printf("%d is non-positive.\n", num);
    }

    // goto statement
    if (num > 0) {
        goto positive;
    } else {
        goto non_positive;
    }
    positive:
    printf("Using goto: Number is positive.\n");
    goto end_label;

    non_positive:
    printf("Using goto: Number is non-positive.\n");

    end_label:
    printf("End of goto example.\n");

    // switch-case statement
    printf("\nMenu:\n");
    printf("1. Print Hello\n");
    printf("2. Print World\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Hello\n");
            break;
        case 2:
            printf("World\n");
            break;
        case 3:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice.\n");
    }

  // break statement in loop
    printf("\nCounting even numbers from 1 to 10 until reaching 6:\n");
    for (i = 1; i <= 10; i++) {
        if (i % 2 != 0) {
            continue; // Skip odd numbers
            }
            if (i > 6) {
                break; // Exit the loop when i exceeds 6
            }
            printf("%d ", i);
    }
    printf("\n");

   // continue statement in loop
    printf("\nPrinting numbers from 1 to 5 except 3:\n");
    for (i = 1; i <= 5; i++) {
            if (i == 3) {
                continue; // Skip printing number 3
            }
            printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

OUTPUT

```
Enter a number: 7
7 is positive.

Using goto: Number is positive.
End of goto example.

Menu:
1. Print Hello
2. Print World
3. Exit
Enter your choice: 2
World

Counting even numbers from 1 to 10 until reaching 6:
2 4 6

Printing numbers from 1 to 5 except 3:
1 2 4 5
```

EXPLANATION

i) if-else Statement: Checks if num is positive or non-positive and prints appropriate messages.

ii) goto Statement: Demonstrates the use of goto to jump to specific labels (positive or nonpositive) based on the condition.

iii) switch-case Statement: Displays a menu and performs actions based on user input (choice).

iv) break Statement: Used in loops to exit early when conditions are met (Exit loop when i exceeds 6).

v) continue Statement: Skips the current iteration of a loop based on a condition (skips printing number 3 in the second loop).

VIVA QUESTIONS

1. Compare and contrast the use of if-else and switch statements for decision-making in C.
2. When would you prefer to use a switch statement over nested if-else statements?
3. Describe how decision-making constructs are used in embedded systems programming.
4. Explain the usage of &&, ||, and ! operators in decision-making constructs.
5. Why are decision-making constructs important in programming?
6. How do you use the ternary operator to replace an if-else statement?

RESULT

Thus, a C Program using decision-making constructs was executed and the output was obtained.

Ex.No:2b **PROGRAM USING LOOPS**

AIM

To understand and implement different types of loops (for, while, do-while) in C programming to control the flow and repetition of statements.

PRE-LAB DISCUSSION

Loops in C Programming

Loops are used to repeat a block of code multiple times until a specified condition is met. They are essential for automating repetitive tasks and iterating over data structures. In C programming, there are three main types of loops:

i) for Loop:
Purpose: Executes a block of code repeatedly based on a specified number of iterations.
Syntax:

```
for (initialization; condition; update) {
   // Code to be executed repeatedly
}
```

Explanation:
initialization: Executes once at the beginning of the loop (typically initializes loop control variable).
condition: Checked before each iteration. If true, the loop continues; if false, the loop terminates.
update: Executed at the end of each iteration (typically increments or decrements loop control variable).

ii) while Loop:
Purpose: Executes a block of code repeatedly as long as a specified condition is true.
Syntax:

```
while (condition) {
   // Code to be executed repeatedly
}
```

Explanation:
condition: Checked before each iteration. If true, the loop continues; if false, the loop terminates.

iii) do-while Loop:
Purpose: Similar to while loop, but guarantees at least one execution of the block of code, even if the condition is initially false.
Syntax:

```
do {
    // Code to be executed repeatedly
} while (condition);
```

Explanation:

The block of code executes once before checking the condition. If the condition is true, the loop continues; if false, the loop terminates.

ALGORITHM

1. Start
2. Declare variables and initializations
3. Read the Input variable.
4. Using different kinds of loops for computational processing.
5. Display the output.

PROGRAM

C program that calculates electricity bills using for, while, and do-while loops:

```
#include <stdio.h>

int main() {
    int units, total_bill;
    int rate_per_unit = 5; // Rate per unit in Rs.
    int j=1;
    int k=1;
    // for loop version
    printf("Using for loop:\n");
    for (int i=1;i<=3;i++) {
            printf("Enter units consumed for customer %d: ", i);
            scanf("%d", &units);

            total_bill = units * rate_per_unit;

            printf("Electricity bill for customer %d: Rs. %d\n", i, total_bill);
    }

    // while loop version
    printf("\nUsing while loop:\n");
    while (j <= 3) {
            printf("Enter units consumed for customer %d: ", j);
            scanf("%d", &units);

            total_bill = units * rate_per_unit;

            printf("Electricity bill for customer %d: Rs. %d\n", j, total_bill);

            j++;
    }

    // do-while loop version
    printf("\nUsing do-while loop:\n");

    do {
            printf("Enter units consumed for customer %d: ", k);
            scanf("%d", &units);

            total_bill = units * rate_per_unit;

            printf("Electricity bill for customer %d: Rs. %d\n", k, total_bill);

            k++;
    } while (k <= 3);

    return 0;
}
```

OUTPUT

Using for loop:

```
Enter units consumed for customer 1: 100

Electricity bill for customer 1: Rs. 500

Enter units consumed for customer 2: 150

Electricity bill for customer 2: Rs. 750

Enter units consumed for customer 3: 80
```

```
Electricity bill for customer 3: Rs. 400
```

Using while loop:

```
Enter units consumed for customer 1: 120

Electricity bill for customer 1: Rs. 600

Enter units consumed for customer 2: 90

Electricity bill for customer 2: Rs. 450

Enter units consumed for customer 3: 200

Electricity bill for customer 3: Rs. 1000
```

Using do-while loop:

```
Enter units consumed for customer 1: 180

Electricity bill for customer 1: Rs. 900

Enter units consumed for customer 2: 60

Electricity bill for customer 2: Rs. 300

Enter units consumed for customer 3: 140

Electricity bill for customer 3: Rs. 700
```

EXPLANATION

i) Initialization: Variables units, total_bill, and rate_per_unit are declared.

ii) for Loop: Iterates three times (i from 1 to 3) to simulate calculations for three customers.

iii) while Loop: Performs the same calculation as for loop but using a while loop structure.

iv) do-while Loop: Similar to the while loop but guarantees at least one execution of the loop body.

Each loop type (for, while, do-while) demonstrates how the program can repeatedly calculate electricity bills based on user input for multiple customers. Adjust the number of iterations (3 in this case) and other details as per your specific requirements.

VIVA QUESTIONS

1. How does a do-while loop differ from a while loop?
2. What is the purpose of the break statement in loops?
3. How does the continue statement work within a loop?
4. What is a nested loop?
5. How do nested loops work in matrix multiplication?
6. Explain the concept of loop unrolling and its advantages.
7. What is an off-by-one error, and how does it occur in loops?
8. How can loops be used in file processing tasks?

RESULT

Thus, a C Program using Loops was executed and the output was obtained.

Ex.No:3a **ONE DIMENSIONAL ARRAY**

AIM

To write a C program to calculate the sum of elements in a 1D array.

PRE-LAB DISCUSSION

Definition of 1D Array: A one-dimensional array is a list of elements, all of the same type, accessible using a single index.
Syntax: data_type array_name[size1];
Declaration: int arr[n]; where n is the number of elements.
Accessing Elements: Use indices starting from 0 up to n-1.
Traversal: Loop through each element using a for loop.
Loops: The most common way to traverse arrays is using loops (for loops, while loops).

ALGORITHM

1. Start.

2. Initialize the array and variables.

3. Traverse the array using a loop.

4. Sum the elements.

5. Print the result.

6. End.

PROGRAM

```
#include <stdio.h>

int main() {
    int n, i, sum = 0;

    // Input number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[500];

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

    // Calculate sum of elements
    for(i = 0; i < n; i++) {
        sum += arr[i];
    }

    // Print the sum
    printf("Sum of elements: %d\n", sum);

    return 0;
}
```

OUTPUT

```
Enter the number of elements: 5

Enter the elements:

1 2 3 4 5

Sum of elements: 15
```

EXPLANATION

1.The program reads the number of elements and then the elements themselves into an array.

2. It then traverses the array using a for loop to calculate the sum of its elements.

3. Finally, it prints the sum.

VIVA QUESTIONS

1. How would you handle operations on one-dimensional arrays with variable sizes?
2. How would you dynamically allocate memory for a one-dimensional array in C?
3. What is a one-dimensional array?
4. How do you declare a one-dimensional array in C?
5. What is the significance of the index in a one-dimensional array?

RESULT

Thus, a C Program using One Dimensional Array was executed and the output was obtained.

Ex.No:3b **TWO-DIMENSIONAL ARRAY**

AIM

To write a C program to calculate the sum of elements in each row of a 2D array.

PRE-LAB DISCUSSION

Definition of 2D Array: Two-dimensional arrays are arrays of arrays, forming a matrix where elements are accessed using two indices.
Syntax: data_type array_name[size1][size2];
Declaration: int arr[m][n]; where m is the number of rows and n is the number of columns.
Accessing Elements: Use two indices, one for the row and one for the column.
Traversal: Use nested loops to traverse the matrix.

ALGORITHM

```
1. Start.
2. Initialize the variables.
3. Allocate the memory for array and declare array.
4. Traverse each row using nested loops.
5.Sum the elements of each row.
6. Print the result.
7. End.
```

PROGRAM

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

int main() {
    int m, n, i, j;

    // Input dimensions of the array
    printf("Enter the number of rows: ");
    scanf("%d", &m);
    printf("Enter the number of columns: ");
    scanf("%d", &n);
    // Allocate memory for the 2D array
    int** arr = (int**)malloc(m * sizeof(int*));
    for(i = 0; i < m; i++) {
        arr[i] = (int*)malloc(n * sizeof(int));
    }

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

    // Calculate sum of elements in each row
    for(i = 0; i < m; i++) {
        int sum = 0;
        for(j = 0; j < n; j++) {
            sum += arr[i][j];
        }
        printf("Sum of elements in row %d: %d\n", i + 1, sum);
    }

    // Free allocated memory
    for(i = 0; i < m; i++) {
        free(arr[i]);
    }
    free(arr);

    return 0;
}
```

OUTPUT

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements:
1 2 3
4 5 6
Sum of elements in row 1: 6
Sum of elements in row 2: 15
```

EXPLANATION

Dynamic Memory Allocation: We use malloc to allocate memory for the 2D array. This avoids the issue with VLAs.
Input and Processing: The rest of the program remains the same, where we input the elements and calculate the sum of each row.
Memory Deallocation: We use free to deallocate the memory once it is no longer needed to avoid memory leaks.

The program reads the dimensions of the array and then its elements. It then traverses each row using nested for loops to calculate the sum of elements in each row. Finally, it prints the sum for each row.

VIVA QUESTIONS

1. What common errors might you encounter when working with two-dimensional arrays in C?
2. How do you initialize a two-dimensional array at the time of declaration?
3. Can you explain the memory layout of a two-dimensional array in C?
4. How can you copy the contents of one two-dimensional array to another?
5. What are some alternative data structures you can use if two-dimensional arrays become too cumbersome?

RESULT

Thus, a C Program using Two-Dimensional Array was executed and the output was obtained.

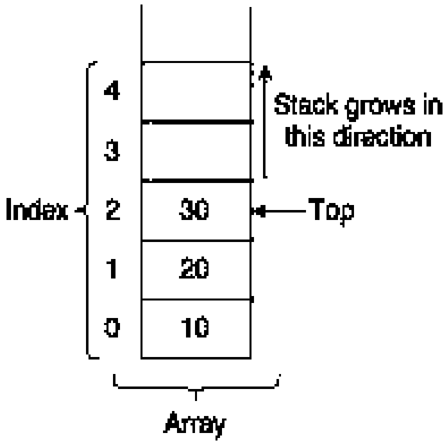
Ex.No:4a **ARRAY IMPLEMENTATION OF STACK ADT**

AIM

To implement a stack data structure using arrays in C programming.

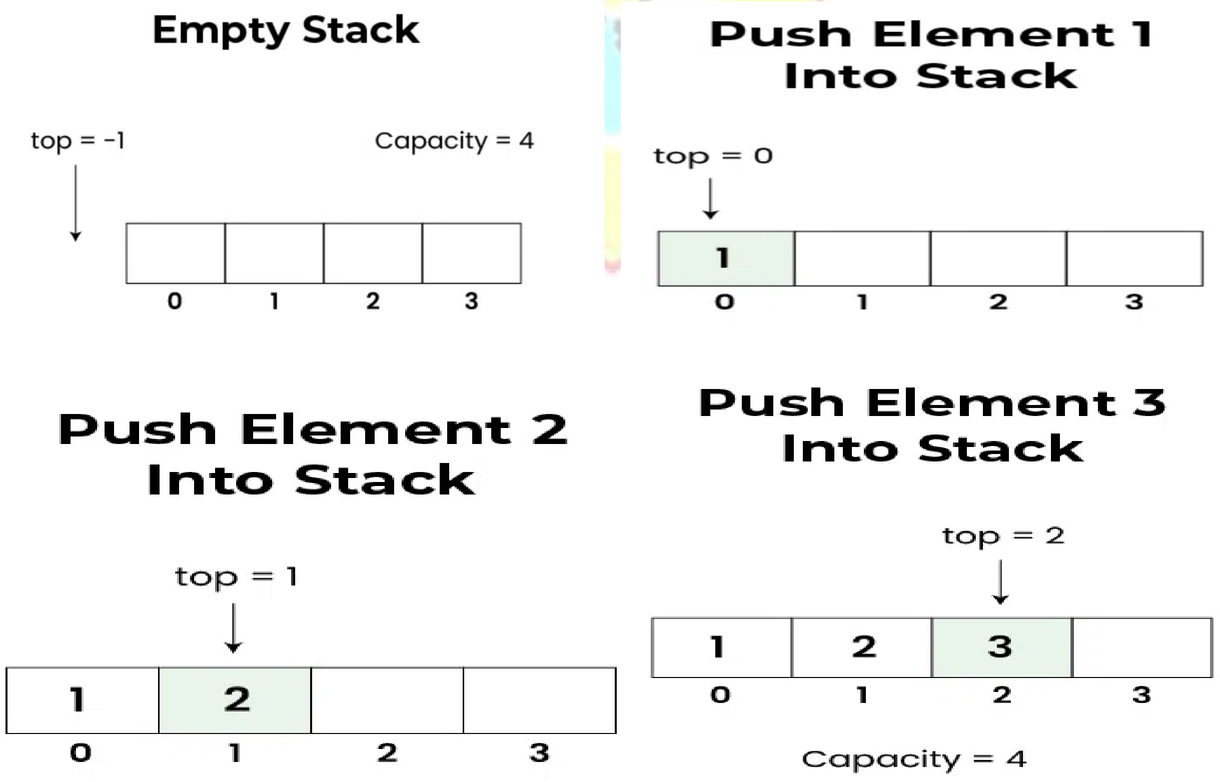
PRELAB DISCUSSION

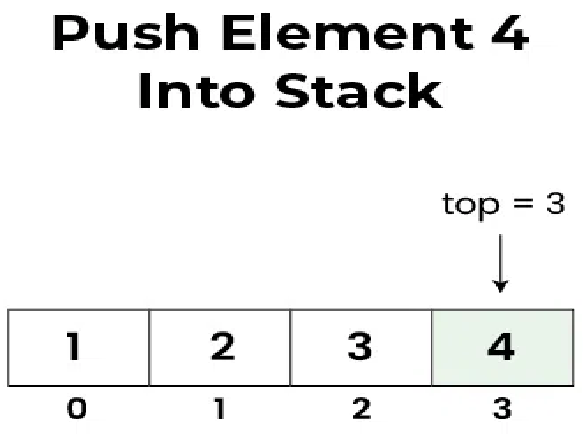
Stack Data Structure: A stack is a Last-In-First-Out (LIFO) data structure where elements are added and removed from one end called the top.

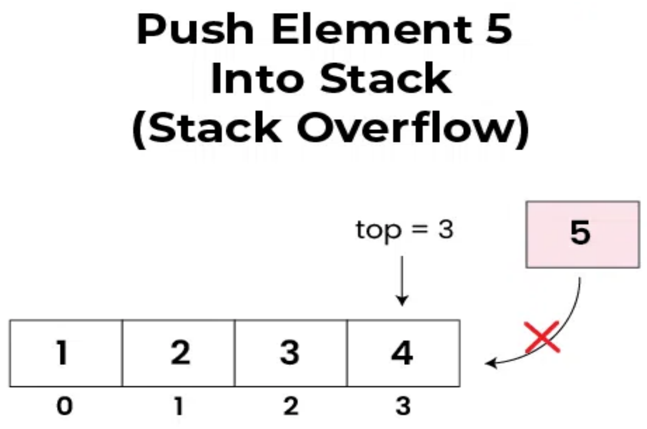


Operations:

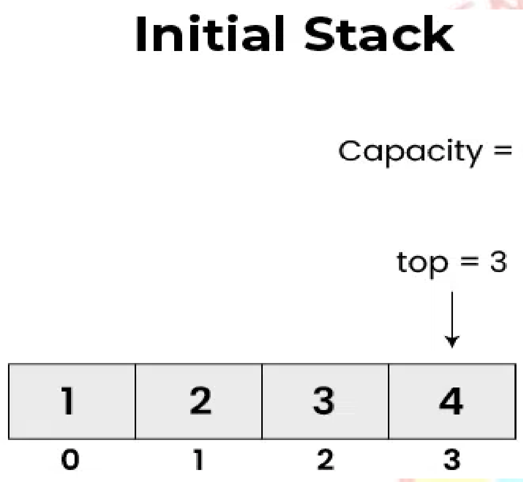
Push: Adds an element to the top of the stack.

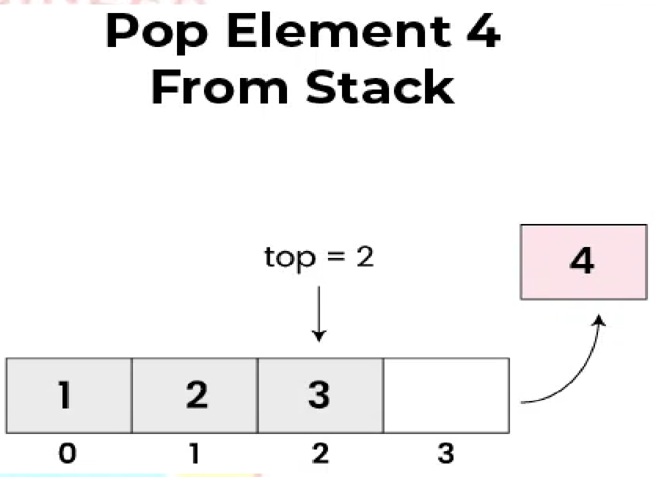


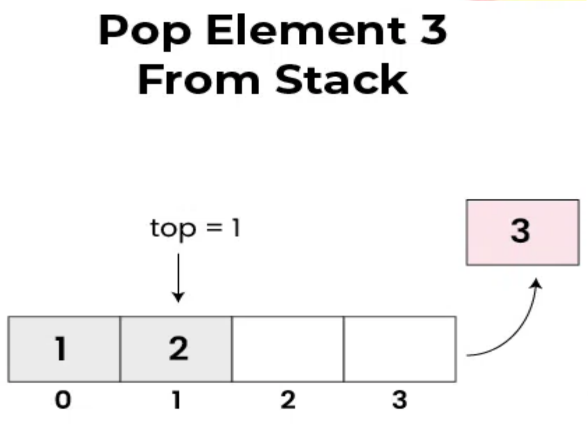


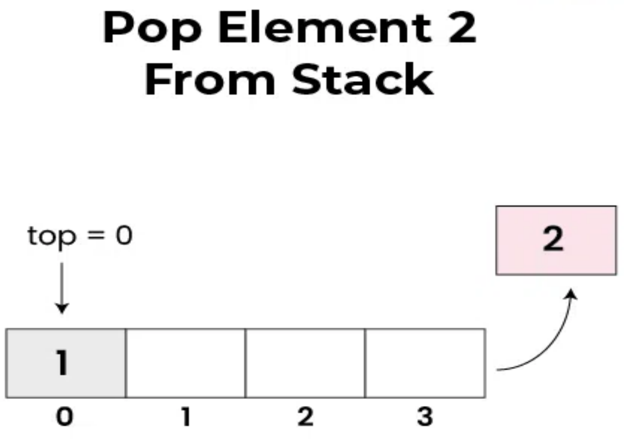


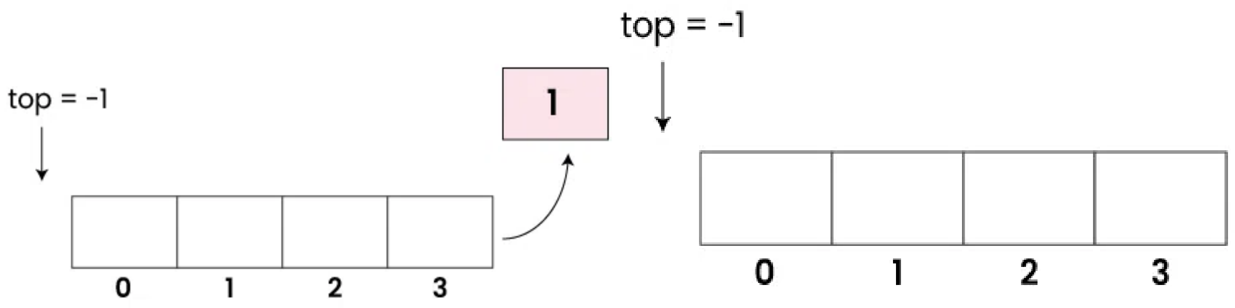
Pop: Removes and returns the element from the top of the stack.











Peek: Returns the element at the top of the stack without removing it.
isEmpty: Checks if the stack is empty.
Array Implementation: In C, a stack can be efficiently implemented using an array due to its simplicity and direct access properties.

ALGORITHM

1. Start.
2. Initialize top = -1;

3: Push operation increases top by one and writes pushed element to storage[top];

4: Pop operation checks that top is not equal to -1 and decreases top variable by 1;

5: Display operation checks that top is not equal to -1 and returns storage[top];

6: Stop.

PROGRAM

```
// Array implementation of Stack ADT's

#include<stdio.h>
#include<conio.h>
#include<process.h>
#define size 5

int item;
int s[10];
int top;

// Function to display the content of the stack
void display()
{
    int i;
    if (top == -1)
    {
        printf("\nThe stack is empty.");
        return;
    }
    printf("\nThe content of the stack is:\n");
    for (i = 0; i <= top; i++)
    {
        printf("%d\t", s[i]);
    }
    printf("\n\n");
}

// Function to push an item onto the stack
void push()
{
    if (top == size - 1)
    {
        printf("\nThe stack is full.");
        return;
    }
    printf("\nEnter an item: ");
    scanf("%d", &item);
    printf("\n");
    s[++top] = item;
}

// Function to pop an item from the stack
void pop()
{
    if (top == -1)
    {
        printf("\nThe stack is empty.");
        return;
    }
    printf("\nThe deleted item is: %d", s[top]);
    printf("\n\n");
    top--;
}
```

```
void main()
{
   int ch;
   top = -1;

   printf("ARRAY IMPLEMENTATION OF STACK ADTs\n\n");
   do
   {
      printf("**************************\n");
      printf("*       MAIN MENU       *\n");
      printf("**************************\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &ch);
      switch (ch)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("\nThank you for using the program. Goodbye!");
            exit(0);
         }
         default:
            {
                printf("\nWrong entry! Please try again.");
            }
        }
    }
    while (ch <= 4);
    getch();
}
```

OUTPUT

```
     ARRAY IMPLEMENTATION OF STACK ADTs

*************************
*          MAIN MENU *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter an item: 12

*************************
*          MAIN MENU *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter an item: 23

*************************
*          MAIN MENU *
*************************
1. Push
```

```
2. Pop
3. Display
4. Exit
Enter your choice: 2

The deleted item is: 23

*************************
*         MAIN MENU *
*************************
1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 1

Enter an item: 34

*************************
*         MAIN MENU *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3

The content of the stack is: 12          34

*************************
*         MAIN MENU *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4

Thank you for using the program. Goodbye!
```

EXPLANATION

Program Execution:

The program starts by initializing an array s of size 10 and a variable top set to -1 to indicate an empty stack. It provides a menu-driven interface using a do-while loop where the user can choose to push an item onto the stack (1), pop an item from the stack (2), display the stack (3), or exit the program (4).

Push Operation (push() function):

If the stack (top) is not full (size - 1), the user is prompted to enter an item which is then pushed onto the stack (s[++top] = item).

Pop Operation (pop() function):

If the stack is not empty (top is not -1), the top item from the stack (s[top]) is displayed as the deleted item, and top is decremented to remove it from the stack.

Display Operation (display() function):

Displays all elements currently in the stack (s) if the stack is not empty (top != -1). If the stack is empty, it prints a message indicating so.

Exiting the Program:

If the user chooses 4 (exit), the program displays a goodbye message and exits using exit(0).

Handling Invalid Choices:

If the user enters an invalid choice (not 1 to 4), the program prompts the user to try again with a valid option.

VIVA QUESTIONS

1. How is a stack used in expression evaluation and syntax parsing?
2. Explain how a stack can be used to reverse a string.
3. How can you extend the array-based stack implementation to support multiple stacks in a single array?
4. How can you avoid memory wastage when implementing a stack using an array?
5. How can you ensure that the array-based stack implementation is efficient and robust?

RESULT

Thus, the C program to implement stack using array was executed successfully.

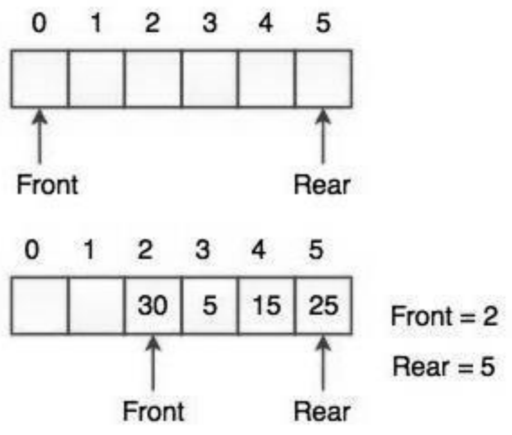
Ex.No:4b ARRAY IMPLEMENTATION OF QUEUE ADT

AIM

To write a C program to implement Queue operations such as enqueue, dequeue and display using array.

PRE LAB DISCUSSION

Queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front'and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.



The Front and Rear of the queue point at the first index of the array. (Array index starts from0). While adding an element into the queue, the Rear keeps on moving ahead and always points to the position where the next element will be inserted. Front remains at the first index.

Queue Operations

1. **Enqueue (Insert)**:
   - Adds an element to the rear (end) of the queue.
   - If the queue is full, it results in a queue overflow.
   - Time complexity: O(1)
2. **Dequeue (Delete)**:
   - Removes and returns the element from the front (beginning) of the queue.
   - If the queue is empty, it results in a queue underflow.
   - Time complexity: O(1)
3. **Front (Peek)**:

        - Returns the element at the front of the queue without removing it.
        - Time complexity: O(1)
4. **isEmpty**:
    - Checks if the queue is empty.
    - Returns true if the queue is empty; otherwise, returns false.
    - Time complexity: O(1)
5. **isFull**:
    - Checks if the queue is full.
    - Returns true if the queue is full; otherwise, returns false.
    - Time complexity: O(1)

ALGORITHM

```
1: Start.
2: Initialize front=0; rear=-1.
3: Enqueue operation moves a rear by one position and inserts a element at the rear.
4: Dequeue operation deletes a element at the front of the list and moves the front by
   one Position.
5: Display operation displays all the element in the list.
6: Stop.
```

PROGRAM

```
// Array implementation of Queue ADT's

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 5

int queue[SIZE], front = 0, rear = -1;

int isFull();
int isEmpty();

// Function to insert an element into the queue
void enqueue(int element) {
   if (isFull())
      printf("\nQUEUE OVERFLOW\n");
   else {
        ++rear;
        queue[rear] = element;
    }
    printf("\n");
}

// Function to delete an element from the queue
int dequeue() {
    int element;
    if (isEmpty()) {
        printf("QUEUE UNDERFLOW\n\n");
        return (-1);
    }
    else {
        element = queue[front];
        front = front + 1;
        return (element);
    }
}

// Function to check if the queue is full
int isFull() {
    if (rear == SIZE - 1)
        return 1;
    return 0;
}

// Function to check if the queue is empty
int isEmpty() {
    if (front > rear)
        return 1;
    return 0;
}

// Function to display the elements of the queue
void display() {
    int i;
    if (isEmpty())
        printf("EMPTY QUEUE\n\n");
    else {
        printf("Front ->");
        for (i = front; i <= rear; i++)
            printf("  %d  ", queue[i]);
        printf("<- Rear");
        printf("\n\n");
    }
}

void main() {
    int option, element;
    printf("ARRAY IMPLEMENTATION OF QUEUE ADTs\n\n");

    do {
        printf("**************************\n");
        printf("***     MAIN MENU     ***\n");
        printf("**************************\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &option);
        printf("\n");
        switch (option) {
            case 1:
                printf("Enter the element to be inserted: ");
                scanf("%d", &element);
                enqueue(element);
                break;
            case 2:
                element = dequeue();
                if (element != -1)
                    printf("Deleted Element is %d\n\n", element);
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Thank you for using the program. Goodbye!\n");
                exit(0);
            break;
        default:
            printf("Invalid Option. Try Again.");
            break;
        }
    } while (option != 4);
    getch();
}
```

OUTPUT

```
ARRAY IMPLEMENTATION OF QUEUE ADTs

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be inserted: 12

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be inserted: 23

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
```

```
3. Display
4. Exit
Enter your choice: 2

Deleted Element is 12

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be inserted: 34

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Front ->  23    34  <- Rear

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
Thank you for using the program. Goodbye!
```

EXPLANATION

Insert (enqueue()): User inserts 10 and 20 into the queue using option 1. Each insertion updates rear and adds elements to the queue.

Display (display()): Option 3 displays the current elements in the queue from front to rear.

Delete (dequeue()): Option 2 removes and returns the element at front (10 in this case). Displays the deleted element.

Exit (option 4): Option 4 terminates the program with a goodbye message.

VIVA QUESTIONS

1. What are the primary operations of a queue?
2. What is the difference between a queue and a stack?
3. What are some practical applications of queue data structures?
4. Why is a queue called a First-In-First-Out (FIFO) data structure?
5. How does the fixed size of an array affect the implementation of a queue?

RESULT

Thus, the C program to implement Queue using array was completed successfully.

Ex.No:4c ARRAY IMPLEMENTATION OF CIRCULAR QUEUE ADT

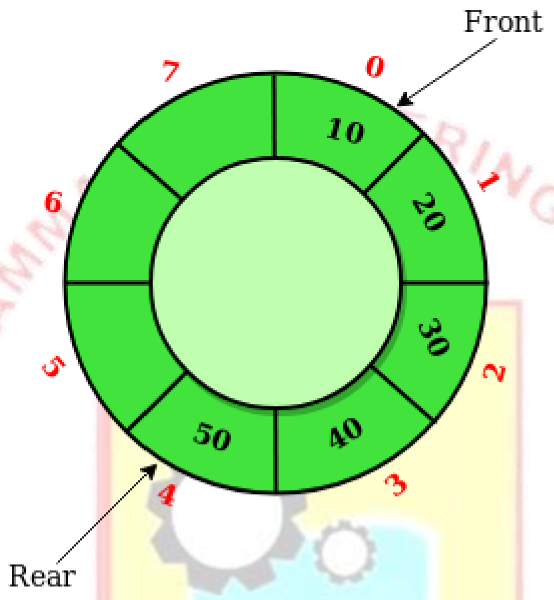
AIM

To write a C program to implement Queue operations such as enqueue, dequeue and display using array.

PRE LAB-DISCUSSION

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called 'Ring Buffer'.



Operations on Circular Queue:

Front: Get the front item from the queue.

Rear: Get the last item from the queue.

enQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.

Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].

If it is full then display Queue is full.

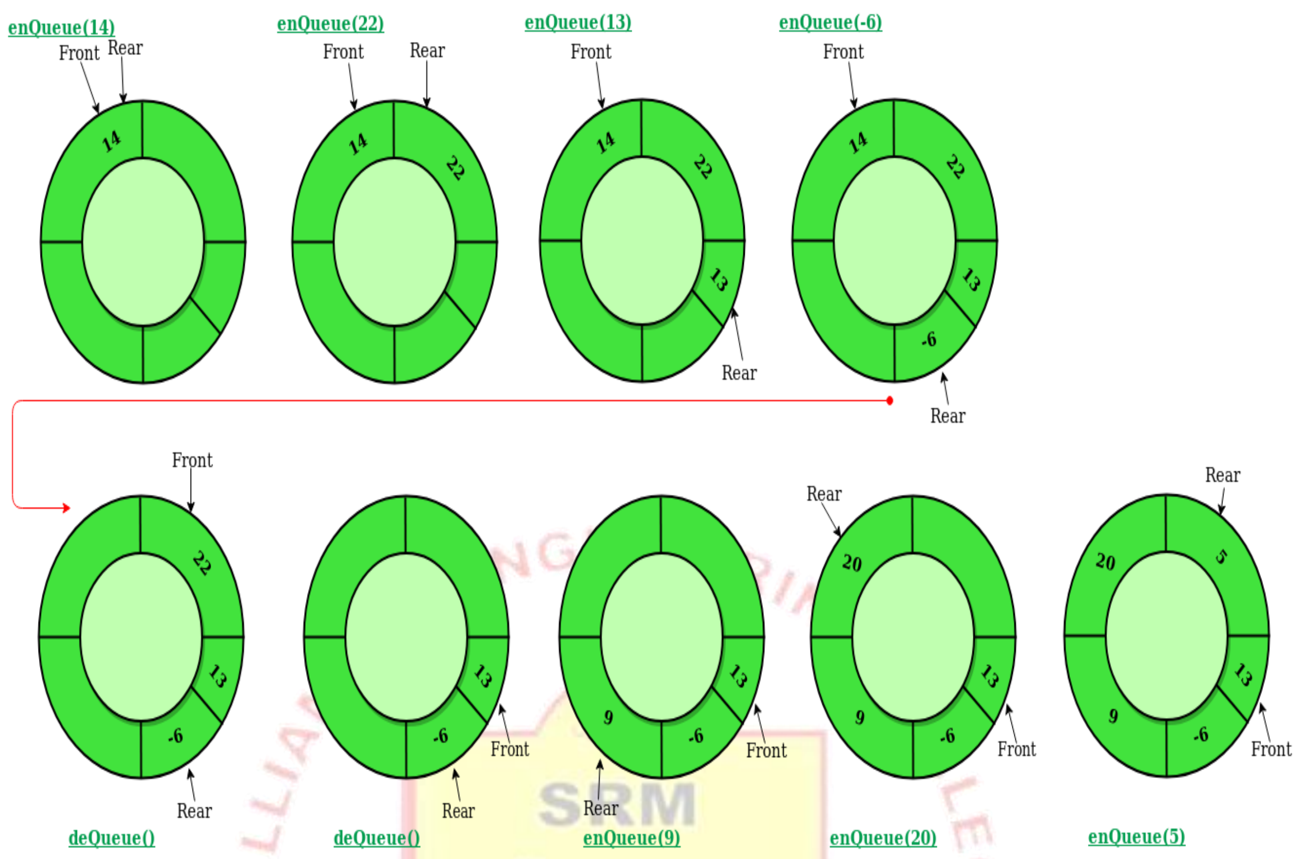
If the queue is not full then, insert an element at the end of the queue.

deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.

Check whether the queue is Empty.

If it is empty then display Queue is empty.

If the queue is not empty, then get the last element and remove it from the queue.



ALGORITHM

1: Start.

2: Initialize front=0; rear=-1.

3: Enqueue operation moves a rear by one position and inserts a element at the rear.

4: Dequeue operation deletes a element at the front of the list and moves the front by one Position.

5: Display operation displays all the element in the list.

6: Stop.

PROGRAM

```
// Array implementation of Circular Queue ADT's
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

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

```
// Function to check if the queue is empty
int isEmpty() {
    return (front == -1 && rear == -1);
}

// Function to check if the queue is full
int isFull() {
    return ((rear + 1) % SIZE == front);
}

// Function to insert an element into the circular queue
void enqueue(int element) {
    if (isFull()) {
        printf("\nQUEUE OVERFLOW\n");
        return;
    } else if (isEmpty()) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % SIZE;
    }
    queue[rear] = element;
    printf("Enqueued element: %d\n", element);
}

// Function to delete an element from the circular queue
int dequeue() {
    int element;
    if (isEmpty()) {
        printf("QUEUE UNDERFLOW\n");
        return -1;
    } else if (front == rear) {
        element = queue[front];
        front = rear = -1;
    } else {
        element = queue[front];
        front = (front + 1) % SIZE;
    }
    return element;
}
```

```
// Function to display the elements of the circular queue
void display() {
    int i;
    if (isEmpty()) {
        printf("EMPTY QUEUE\n");
        return;
    }
    printf("Queue elements: ");
    i = front;
    do {
        printf("%d ", queue[i]);
        i = (i + 1) % SIZE;
    } while (i != (rear + 1) % SIZE);
    printf("\n");
}

int main() {
    int option, element;
    printf("CIRCULAR QUEUE ADTs IMPLEMENTATION\n\n");

    do {
        printf("**************************\n");
        printf("***     MAIN MENU     ***\n");
        printf("**************************\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &option);
        printf("\n");
        switch (option) {
            case 1:
                printf("Enter the element to be inserted: ");
                scanf("%d", &element);
                enqueue(element);
                break;
            case 2:
                element = dequeue();
                if (element != -1)
                    printf("Deleted Element is %d\n", element);
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Thank you for using the program. Goodbye!\n");
                exit(0);
            default:
                printf("Invalid Option. Try Again.\n");
                break;
        }
    } while (option != 4);

    return 0;
}
```

OUTPUT

```
CIRCULAR QUEUE ADTs IMPLEMENTATION

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be inserted: 10
Enqueued element: 10

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter your choice: 1

Enter the element to be inserted: 20
Enqueued element: 20

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be inserted: 30
Enqueued element: 30

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Queue elements: 10 20 30

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2

Deleted Element is 10
```

```
*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Queue elements: 20 30

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be inserted: 40
Enqueued element: 40

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Queue elements: 20 30 40

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
```

```
4. Exit
Enter your choice: 4

Thank you for using the program. Goodbye!
```

EXPLANATION

Circular Queue Implementation:

Enqueue Operation: Checks if the queue is full using (rear + 1) % SIZE == front. If empty, initializes front and rear to 0. Otherwise, increments rear using modulo operation (rear + 1) % SIZE.
Dequeue Operation: Checks if the queue is empty using front == -1. If only one element, resets front and rear to -1. Otherwise, increments front using modulo operation (front + 1) % SIZE.
Display Operation: Iterates through the circular queue using modulo operation to wrap around at the end.
Main Function: Implements a menu-driven approach to interact with the circular queue (enqueue, dequeue, display, exit).

Initially, elements 10, 20, and 30 are enqueued into the circular queue. After each operation (enqueue, dequeue, display), the current state of the queue is shown. Dequeuing the element 10 results in 10 being removed from the queue. Enqueuing 40 adds 40 to the rear of the queue. Finally, upon choosing option 4 (Exit), the program ends with a farewell message.

VIVA QUESTIONS

1. What considerations should be taken into account when implementing a circular queue for real-time applications?
2. In what scenarios would you prefer using a circular queue over other types of queues?
3. What happens if you try to dequeue an element from an empty circular queue?
4. What happens if you try to enqueue an element into a full circular queue?
5. Why is a circular queue called a circular queue?

RESULT

Thus, the C program to implement Circular Queue was completed successfully.

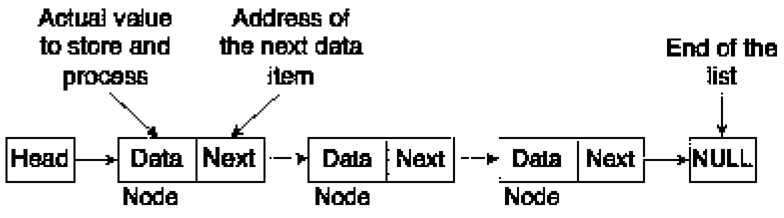
Ex.No:5 IMPLEMENTATION OF SINGLY LINKED LIST

AIM

To write a C program to implement singly linked list.

PRE LAB-DISCUSSION

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer. In linked list, each node consists of its own data and the address of the next node and forms a chain.



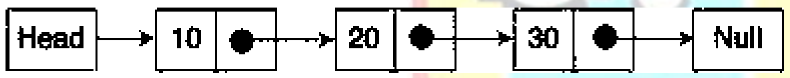
Actual value to store and process
Address of the next data item
End of the list
Head
Data
Next
Node
NULL

Fig. Linked List

Linked list contains a link element called first and each link carries a data item. Entry point intothe linked list is called the head of the list. Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of thelist.

Linked list is a dynamic data structure. While accessing a particular item, start at the head and follow the references until you get that data item.

Linked list is used while dealing with an unknown number of objects:



Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL. The real-life example of Linked List is that of Railway Carriage. It starts from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

ALGORITHM

1: Start

2: Creation: Get the number of elements, and create the nodes having structures DATA,LINK and store the element in Data field, link them together to form a linked list.

3: Insertion: Get the number to be inserted and create a new node store the value in DATA field and insert the node in the required position.

4: Deletion: Get the number to be deleted. Search the list from the beginning and locate the node then delete the node.

5: Display: Display all the nodes in the list.

6: Stop.

PROGRAM

```
// Linked list implementation of list ADT's

// Include necessary libraries and define constants
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

typedef struct list
{
    int no; // number
    struct list *next; // pointer to next node
} LIST;

LIST *p, *t, *h, *y, *ptr, *pt;

void create(void);
void insert(void);
void delet(void);
void display(void);

int j, pos, k = 1, count;

void main()
{
    // Code to initialize the list and perform operations on it
    int n, i = 1, opt;
    p = NULL;


    printf("LINKED LIST IMPLEMENTATION OF LIST ADTs\n\n");
    printf("Enter the number of nodes: ");
    scanf("%d", &n);
    printf("\n");

    count = n;
```

```
while (i <= n)
{
   create(); // create nodes
   i++;
}
printf("\n");

do
{
   printf("*************************\n");
   printf("*       MAIN MENU       *\n");
   printf("*************************\n");
   printf("1. Insert\n");
   printf("2. Delete\n");
   printf("3. Display\n");
   printf("4. Exit\n");
   printf("Enter your option: ");
   scanf("%d", &opt);
   printf("\n");
   switch (opt)
   {
      case 1:
         insert(); // insert node
         count++;
         break;
      case 2:
         delet(); // delete node
         count--;
         if (count == 0)
         {
            printf("\nList is empty\n");
         }
         break;
      case 3:
         printf("List elements are:\n");
         display(); // display nodes
         break;
      case 4:
         printf("Thank you for using the program!\n");
```

```
                exit(0);
                break;
        }
    } while (opt != 4);
    getch();
}

void create()
{
    // Code to create a new node and add it to the list
    if (p == NULL)
    {
        p = (LIST *)malloc(sizeof(LIST));
        printf("Enter the element: ");
        scanf("%d", &p->no);
        p->next = NULL;
        h = p;
    }
    else
    {
        t = (LIST *)malloc(sizeof(LIST));
        printf("Enter the element: ");
        scanf("%d", &t->no);
        t->next = NULL;
        p->next = t;
        p = t;
    }
}


void insert()
{
    // Code to insert an element into the list
    t = h;
    p = (LIST *)malloc(sizeof(LIST));
    printf("Enter the element to be inserted: ");
    scanf("%d", &p->no);
    printf("Enter the position to insert: ");
    scanf("%d", &pos);
    if (pos == 1)
    {
        h = p;
        h->next = t;
    }
    else
    {
        for (j = 1; j < (pos - 1); j++)
            t = t->next;
        p->next = t->next;
        t->next = p;
        t = p;
    }
    printf("\n");
}

void delet()
{
    // Code to delete an element from the list
    printf("Enter the position to delete: ");
    scanf("%d", &pos);
    if (pos == 1)
    {
        h = h->next;
    }
    else
    {
        t = h;
        for (j = 1; j < (pos - 1); j++)
            t = t->next;
        pt = t->next->next;
        free(t->next);
        t->next = pt;
    }
    printf("\n");
}

void display()
{
    // Code to display the elements of the list
    t = h;
    while (t->next != NULL)
    {
        printf("%d\t", t->no);
        t = t->next;
    }
    printf("%d", t->no);
    printf("\n\n");
}
```

OUTPUT

```
LINKED LIST IMPLEMENTATION OF LIST ADTs

Enter the number of nodes: 5

Enter the element: 12
Enter the element: 23
Enter the element: 34
Enter the element: 45
Enter the element: 56

*************************
*       MAIN MENU       *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 1

Enter the element to be inserted: 27
Enter the position to insert: 3

*************************
*       MAIN MENU       *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 2
Enter the position to delete: 4

*************************
*      MAIN MENU      *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 3

List elements are:
12      23      27      45      56

*************************
*      MAIN MENU      *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 4

Thank you for using the program!
```

EXPLANATION

Node Definition: The LIST structure defines a node containing an integer (no) and a pointer to the next node (next).
Pointer Variables: p, t, h, y, ptr, and pt are used as pointers to LIST nodes, indicating operations like insertion, deletion, and traversal within the linked list.
Functions:
create(): Creates a new node and adds it to the end of the linked list.
insert(): Inserts a new node at a specified position in the linked list.
delet(): Deletes a node at a specified position from the linked list.
display(): Displays all elements of the linked list.
Main Function:
Initializes the linked list with user-defined number of nodes. Provides a menu-driven interface (opt) to perform insert, delete, display, and exit operations on the linked list.

Dynamic Memory Allocation: Uses malloc() to allocate memory for new nodes dynamically, which is a characteristic feature of linked lists.
Traversal: Traverses through the linked list using pointers (t) to access and manipulate nodes.

VIVA QUESTIONS

1. How would you define a node in a singly linked list?
2. What is the role of the head pointer in a singly linked list?
3. Can you provide an example of a problem where using a singly linked list would be advantageous?
4. How would you handle edge cases such as inserting or deleting from an empty list?
5. Compare and contrast singly linked lists with doubly linked lists and circular linked lists.

RESULT

Thus, the C program to implement Singly Linked List was completed successfully.

Ex.No:6a **LINKED LIST IMPLEMENTATION OF STACK ADT**

AIM

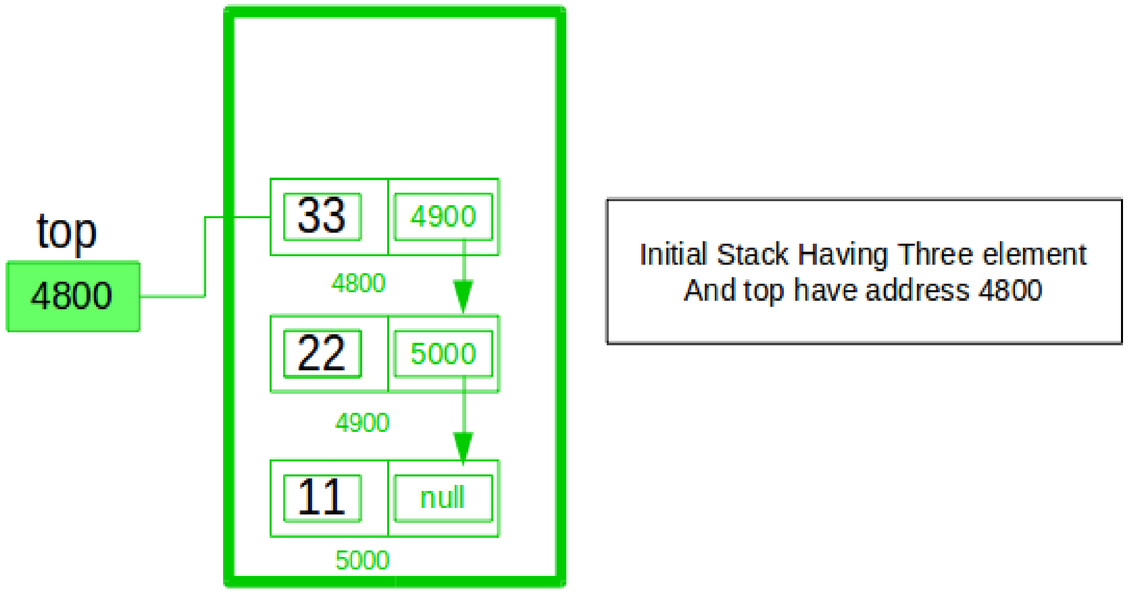
To write a C program to implement Stack operations such as push, pop and display using linked list.

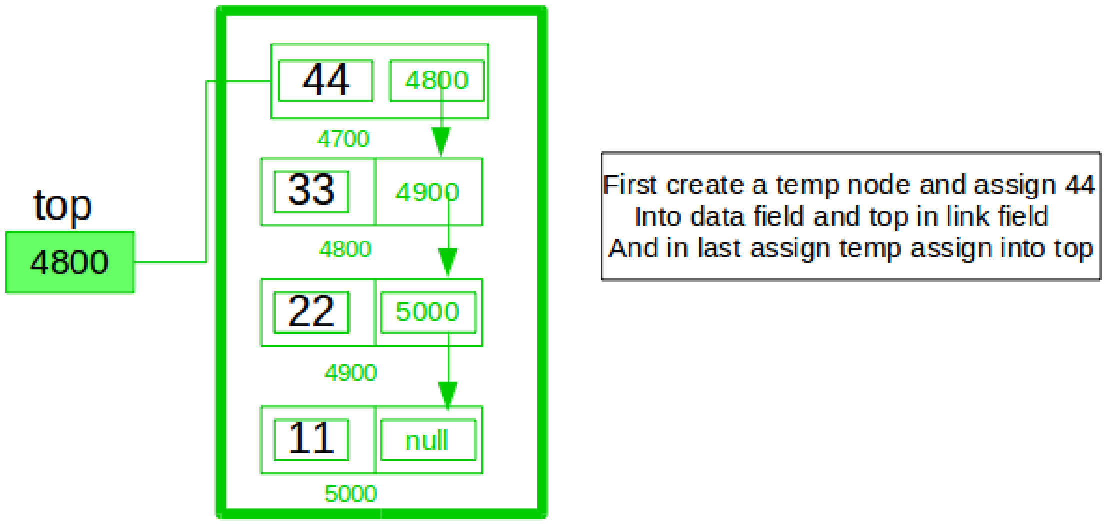
PRE LAB-DISCUSSION

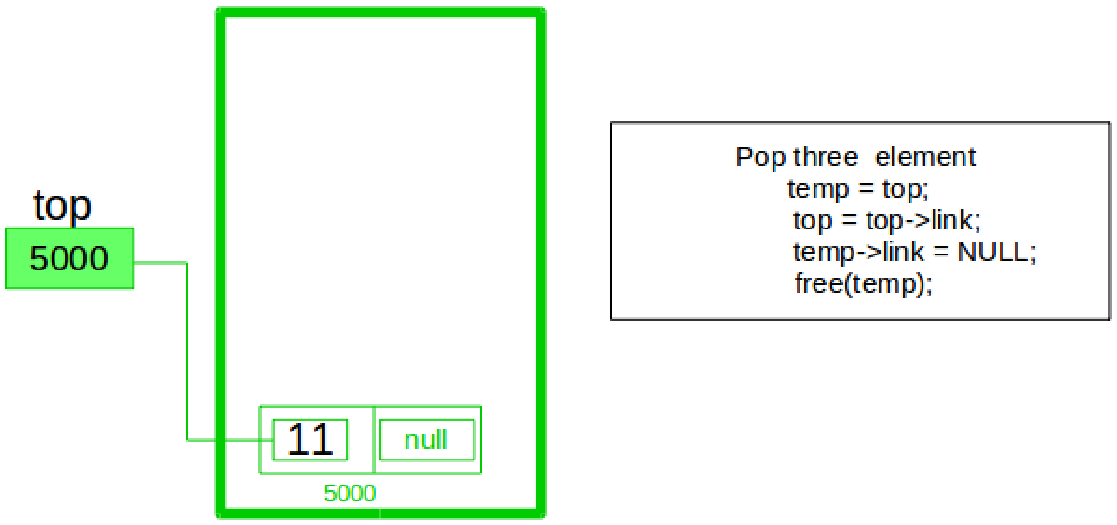
The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

Example:




top
4800
44
4800
4700
33
4900
4800
22
5000
4900
11
null
5000
First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top



There are two basic operations performed in a Stack:
1. Push():is used to add or insert new elements into thestack.
2. Pop():is used to delete or remove an element from thestack.

ALGORITHM

1: Start.

2: push operation inserts an element at the front.

3: pop operation deletes an element at the front of the list;

4: display operation displays all the elements in the list.

5: Stop.

PROGRAM

```
// Linked list implementation of stack

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

void pop();
void push(int value);
void display();

struct node
{
    int data;
    struct node *link;
};

struct node *top = NULL, *temp;

void main()
{
    int choice, data;
    printf("LINKED LIST IMPLEMENTATION OF STACK\n\n");

    while (1) // infinite loop is used to insert/delete infinite number of elements in stack
    {
        printf("**************************\n");
        printf("*       MAIN MENU       *\n");
        printf("**************************\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice)
        {
            case 1: // To push a new element into stack
            {
                printf("Enter a new element: ");
                scanf("%d", &data);
                push(data);
                break;
            }
            case 2: // pop the element from stack
            {
                pop();
                break;
            }
            case 3: // Display the stack elements
            {
                display();
                break;
            }
            case 4: // To exit
            {
                printf("Thank you for using the program. Exiting...\n");
                exit(0);
            }
        }
    }
    getch();
    // return 0;
}

void display()
{
    temp = top;
    if (temp == NULL)
    {
        printf("\nStack is empty\n");
    }
    printf("\nThe Contents of the Stack are...\n");
    while (temp != NULL)
    {
        printf(" %d->", temp->data);
        temp = temp->link;
    }
    printf("\n\n");
```

```
}

void push(int data)
{
    temp = (struct node *)malloc(sizeof(struct node)); // creating a space for the new element.
    temp->data = data;
    temp->link = top;
    top = temp;
    display();
}

void pop()
{
    if (top != NULL)
    {
        printf("The popped element is %d", top->data);
        printf("\n");
        top = top->link;
    }
    else
    {
        printf("\nStack Underflow");
    }
    display();
}
```

OUTPUT

```
LINKED LIST IMPLEMENTATION OF STACK

*************************
*      MAIN MENU      *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter a new element: 12
```

```
The Contents of the Stack are...
 12->

*************************
*      MAIN MENU      *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter a new element: 23

The Contents of the Stack are...
 23-> 12->

*************************
*      MAIN MENU      *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2

The popped element is 23

The Contents of the Stack are...
 12->

*************************
*      MAIN MENU      *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter a new element: 34

The Contents of the Stack are...
 34-> 12->

*************************
*       MAIN MENU       *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3


The Contents of the Stack are...
 34-> 12->

*************************
*       MAIN MENU       *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4

Thank you for using the program. Exiting...
```

EXPLANATION

Initial Display: The program starts by displaying the main menu and prompts the user to enter a choice.

Push Operation (Choice 1): User chooses to push a new element (12) onto the stack. After pushing, the stack contents (12) are displayed.

Push Operation Again (Choice 1): User pushes another element (23) onto the stack. Updated stack contents (23-> 12->) are displayed.

Pop Operation (Choice 2): User chooses to pop an element from the stack. The element 23 is removed (popped), and the updated stack contents (12->) are displayed.

Push Operation Again (Choice 1): User pushes another element (34) onto the stack. Updated stack contents (34-> 12->) are displayed.

Display Operation (Choice 3): User chooses to display the stack contents. The current stack contents (34-> 12->) are displayed.
Exit Operation (Choice 4): User chooses to exit the program.

VIVA QUESTIONS

1. What challenges might arise when implementing stack operations using a linked list?
2. How would you handle stack overflow or underflow conditions in a linked list-based stack?
3. What are some practical applications of stack data structures in computer science?
4. What is the top of the stack, and why is it important?
5. How do you check if a stack implemented using a linked list is empty?
6. How do you retrieve the top element of a stack without removing it?

RESULT

Thus, the C program to implement Stack using linked list was completed successfully.

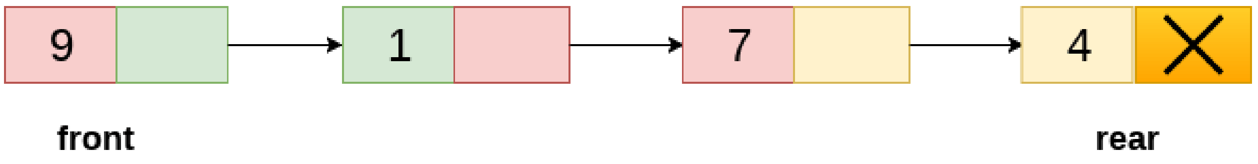
Ex.No:6b LINKED LIST IMPLEMENTATION OF LINEAR QUEUE ADT

AIM

To write a C program to implement Queue operations such as enqueue, dequeue and display using linked list.

PRE LAB-DISCUSSION

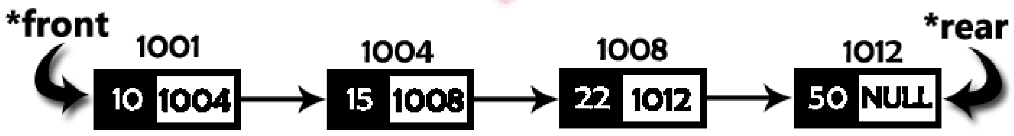
The major problem with the queue implemented using array is, It will work for only fixed number of data. That means the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation).



Linked Queue

The Queue implemented using linked list can organize as many data values as we want. In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

There are two basic operations performed on a Queue.

Enqueue(): This function defines the operation for adding an element into queue.

Dequeue(): This function defines the operation for removing an element from queue.

Key Characteristics of a Linear Queue ADT:
FIFO Order: Elements are inserted at the rear (end) and removed from the front (beginning).
Single Directional: The queue grows in one direction, from front to rear.
Operations: Basic operations include insertion (enqueue), deletion (dequeue), and display.

ALGORITHM

```
1: Start.
2: Enqueue operation inserts an element at the rear of the list.
3: Dequeue operation deletes an element at the front of the list.
4: Display operation display all the element in the list.
5: Stop.
```

PROGRAM

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

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

struct Node *front = NULL, *rear = NULL;

void insert(int);
void delet();
void display();

void main()
{
   int choice, value;
   printf("LINKED LIST IMPLEMENTATION OF QUEUES\n\n");

   while (1) {
          printf("*************************\n");
          printf("***     MAIN MENU     ***\n");
          printf("*************************\n");
```

```
            printf("1. Insert\n");
            printf("2. Delete\n");
            printf("3. Display\n");
            printf("4. Exit\n");
            printf("Enter your choice: ");
            scanf("%d", &choice);
            printf("\n");

            switch (choice) {
               case 1:
                    printf("Enter the value to be inserted: ");
                    scanf("%d", &value);
                    insert(value);
                    break;
               case 2:
                    delet();
                    break;
               case 3:
                    display();
                    break;
               case 4:
                    printf("\nThank you for using the program. Exiting...\n");
                    exit(0);
               default:
                    printf("\nInvalid option. Try again.\n");
            }
    }
}

void insert(int value)
{
    struct Node *newNode;
    newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> next = NULL;
    if (front == NULL)
            front = rear = newNode;
    else {
            rear -> next = newNode;
            rear = newNode;
    }
    printf("\nINSERTION SUCCESSFULL\n\n");
}

void delet()
{
    if (front == NULL)
        printf("EMPTY QUEUE\n\n");
    else {
        struct Node *temp = front;
        front = front->next;
        printf("Deleted element: %d\n\n", temp -> data);
        free(temp);
    }
}

void display()
{
    if (front == NULL)
        printf("EMPTY QUEUE\n\n");
    else {
        struct Node *temp = front;
        while (temp -> next != NULL) {
            printf("%d -> ", temp -> data);
            temp = temp->next;
        }
        printf("%d -> NULL\n\n", temp -> data);
    }
}
```

OUTPUT

```
LINKED LIST IMPLEMENTATION OF QUEUES

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter your choice: 1

Enter the value to be inserted: 12

INSERTION SUCCESSFULL

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be inserted: 23

INSERTION SUCCESSFULL

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2

Deleted element: 12

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be inserted: 34
```

```
INSERTION SUCCESSFULL

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

23 -> 34 -> NULL

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4

Thank you for using the program. Exiting...
```

EXPLANATION

Node Structure: Each node contains data and a pointer to the next node.

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

Global Pointers: front and rear pointers are used to track the front and rear of the queue.

```
struct Node *front = NULL, *rear = NULL;
```

Insertion (Enqueue):

New elements are added at the rear of the queue. If the queue is empty, both front and rear are set to the new node. Otherwise, the new node is added after the rear, and rear is updated.

Deletion (Dequeue):

Elements are removed from the front of the queue. If the queue is empty, a message is displayed. Otherwise, the front element is removed, and front is updated to the next node.

Display:

The queue elements are displayed from the front to the rear. If the queue is empty, a message is displayed. The primary operations (insert, delete, display) maintain the FIFO order, and the use of linked nodes allows dynamic memory management, accommodating the dynamic nature of queues.

VIVA QUESTIONS

1. Compare and contrast a queue implemented using a linked list versus an array.
2. How does a linked list-based queue handle memory allocation compare to an array-based queue?
3. What are the limitations or drawbacks of using a linked list for implementing a queue?
4. How would you handle queue underflow or overflow conditions in a linked list-based queue?
5. What modifications would you make to the linked list-based queue implementation to support priority queues?
6. How do you check if a queue implemented using a linked list is empty?
7. How do you define a node in the linked list for queue implementation?

RESULT

Thus, the C program to implement linear queue using linked list was completed successfully.

Ex.No:7 IMPLEMENTATION OF POLYNOMIAL MANIPULATION USING LINKED LIST

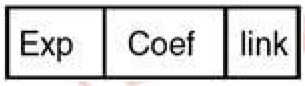
AIM

To write a 'C' program to represent a polynomial as a linked list and write functions for polynomial addition

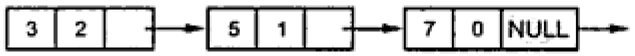
PRE LAB-DISCUSSION

A **polynomial equation** is an **equation** that can be written in the form. axn + bxn-1 + . . . + rx+ s = 0, where a, b, . . . , r and s are constants. We call the largest exponent of x appearing in a nonzero term of a **polynomial** the degree of that **polynomial**.A Polynomial has mainly two fields Exponent and coefficient.

Node of a Polynomial:



For example $3x^2 + 5x + 7$ will represent as follows.



In each node the exponent field will store the corresponding exponent and the coefficient field will store the corresponding coefficient. Link field points to the next item in the polynomial.Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients which have same variable powers.

Example:

Input: 1st number = $5x^2 + 4x + 2$ 2nd number = $5x + 5$ **Output:** $5x^2 + 9x^1 + 7$	**Input:** 1st number = 5x^2 + 4x^1 + 2x^0 2nd number = 5x^1 + 5x^0 **Output:** 5x^2 + 9x^1 + 7x^0

ALGORITHM

1: Start the program

2: Get the coefficients and powers for the two polynomials to be added.

3: Sort the polynomial based upon the powers.

4: Add the coefficients of the respective powers.

5: Display the added polynomial.

6: Terminate the program.

PROGRAM

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

struct Polynomial
{
    int coefficient;
    int power;
    struct Polynomial *next;
} *ptr, *start1, *node, *start2, *start3, *ptr1, *ptr2;
typedef struct Polynomial Poly;

int temp1, temp2;

void main()
{
    void create(void);
    void print(void);
    void sum(void);
    void sort(void);

    printf("APPLICATIONS OF LISTS\n\n");
    printf("Enter the elements of the first polynomial:\n");
    node = (Poly *) malloc(sizeof (Poly));
    start1 = node;
    if (start1 == NULL)
    {
        printf("UNABLE TO ALLOCATE MEMORY.");
        exit(0);
    }
    create();
    printf("\nEnter the elements of the second polynomial:\n");
    node = (Poly *) malloc(sizeof (Poly));
    start2 = node;
    if (start2 == NULL)
    {
        printf("UNABLE TO ALLOCATE MEMORY.");
        exit(0);
    }
    create();

    printf("\nThe elements of the first polynomial are:\n");
    ptr = start1;
    print();
    printf("\nThe elements of the second polynomial are:\n");
    ptr = start2;
    print();
    printf("\nThe first sorted list is:\n");
    ptr = start1;
    sort();
    ptr = start1;
    print();
    printf("\nThe second sorted list is:\n");
    ptr = start2;
    sort();
    ptr = start2;
    print();
    printf("\nThe sum of the two polynomials is: \n");
    sum();
    ptr = start3;
    print();
    printf("\n");
}

void create()
{
    char ch;
    while (1)
    {
        printf("Enter the coefficient and power: ");
        scanf("%d%d", &node->coefficient, &node->power);
        if (node->power == 0)
        {
            ptr = node;
            node = (Poly *) malloc(sizeof(Poly));
            node = NULL;
            ptr->next = node;
            break;
        }
        printf("Do you want to enter more coefficients? (y/n): ");
        scanf(" %c", &ch);
        if (ch == 'n')
        {
            ptr = node;
            node = (Poly *) malloc(sizeof(Poly));
            node = NULL;
            ptr->next = node;
            break;
        }
        ptr = node;
        node = (Poly *) malloc(sizeof(Poly));
        ptr->next = node;
    }
}

void print()
{
    int i = 1;
    while (ptr != NULL)
    {
        if (i != 1)
            printf("+ ");
        printf("%d x^%d ", ptr->coefficient, ptr->power);
        ptr = ptr->next;
        i++;
    }
    printf("\n");
}

void sort()
{
    for (; ptr != NULL; ptr = ptr->next)
        for (ptr2 = ptr->next; ptr2 != NULL; ptr2 = ptr2->next)
        {
            if (ptr->power > ptr2->power)
            {
                temp1 = ptr->coefficient;
                temp2 = ptr->power;
                ptr->coefficient = ptr2->coefficient;
                ptr->power = ptr2->power;
                ptr2->coefficient = temp1;
                ptr2->power = temp2;
            }
        }
}

void sum()
{
    node = (Poly *) malloc(sizeof(Poly));
    start3 = node;
    ptr1 = start1;
    ptr2 = start2;
    while (ptr1 != NULL && ptr2 != NULL)
    {
        ptr = node;
        if (ptr1->power > ptr2->power)
        {
            node->coefficient = ptr2->coefficient;
            node->power = ptr2->power;
            ptr2 = ptr2->next;
        }
        else if (ptr1->power < ptr2->power)
        {
            node->coefficient = ptr1->coefficient;
            node->power = ptr1->power;
            ptr1 = ptr1->next;
        }
        else
        {
            node->coefficient = ptr2->coefficient + ptr1->coefficient;
            node->power = ptr2->power;
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
        }
        node = (Poly *) malloc(sizeof(Poly));
        ptr->next = node;
    }
    if (ptr1 == NULL)
    {
        while (ptr2 != NULL)
        {
            node->coefficient = ptr2->coefficient;
            node->power = ptr2->power;
            ptr2 = ptr2->next;
            ptr = node;
            node = (Poly *) malloc(sizeof(Poly));
            ptr->next = node;
        }
    }
    else if (ptr2 == NULL)
    {
        while (ptr1 != NULL)
        {
            node->coefficient = ptr1->coefficient;
            node->power = ptr1->power;
            ptr1 = ptr1->next;
            ptr = node;
            node = (Poly *) malloc(sizeof(Poly));
            ptr->next = node;
        }
    }
    node = NULL;
    ptr->next = node;
}
```

OUTPUT

```
APPLICATIONS OF LISTS

Enter the elements of the first polynomial:
Enter the coefficient and power: 4 2
Do you want to enter more coefficients? (y/n): y
Enter the coefficient and power: 4 1
Do you want to enter more coefficients? (y/n): y
Enter the coefficient and power: 4 0
Enter the elements of the second polynomial:
Enter the coefficient and power: 2 2
Do you want to enter more coefficients? (y/n): y
Enter the coefficient and power: 2 1
Do you want to enter more coefficients? (y/n): y
Enter the coefficient and power: 2 0

The elements of the first polynomial are:
4 x^2 + 4 x^1 + 4 x^0

The elements of the second polynomial are:
2 x^2 + 2 x^1 + 2 x^0

The first sorted list is:
4 x^0 + 4 x^1 + 4 x^2

The second sorted list is:
2 x^0 + 2 x^1 + 2 x^2

The sum of the two polynomials is:
6 x^0 + 6 x^1 + 6 x^2
```

EXPLANATION

Node Structure:

```
struct Polynomial

{

    int coefficient;

    int power;

    struct Polynomial *next;

} *ptr, *start1, *node, *start2, *start3, *ptr1, *ptr2;

typedef struct Polynomial Poly;
```

Global Variables:

start1, start2, and start3 are pointers to the head of the first, second, and resultant polynomial lists, respectively. ptr, ptr1, and ptr2 are used to traverse the lists. node is used to create new nodes. temp1 and temp2 are used for temporary storage during sorting.

Main Function:

The main function manages the flow of the program, calling other functions to create polynomials, print them, sort them, and find their sum.

Creating Polynomials:

The create function allows the user to input the terms of a polynomial. Each term includes a coefficient and a power. The user can decide whether to add more terms or stop input.

Displaying Polynomials:

The print function traverses the linked list and prints each term of the polynomial.

Sorting Polynomials:

The sort function sorts the terms of the polynomial in ascending order of their powers using a basic sorting algorithm.

Summing Polynomials:

The sum function adds two polynomials by combining terms with the same powers. It creates a new polynomial (linked list) to store the result.

VIVA QUESTIONS

1. Explain why polynomial addition and multiplication are analogous to arithmetic operations on numbers.
2. How does the degree of a polynomial affect its representation and operations using linked lists?
3. If a polynomial operation is not producing the expected result, what steps would you take to debug it?
4. How do you update the coefficient of a specific power in a polynomial?
5. How are polynomials typically represented in mathematical notation?
6. Why would you use a linked list to represent a polynomial?

RESULT

Thus, the C program to implement Polynomial using linked list was completed successfully.

Ex.No:8a IMPLEMENTATION OF EVALUATING POSTFIX EXPRESSIONS

AIM

To write a C program to evaluating Postfix Expression using stack

PRE LAB-DISCUSSION

Stack Operations:

Push: Adds an element to the top of the stack.

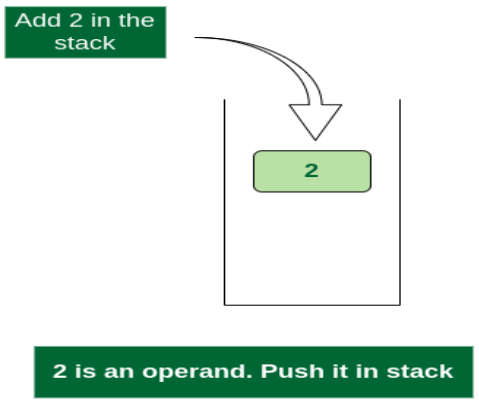
Pop: Removes and returns the top element of the stack.

Postfix Evaluation:

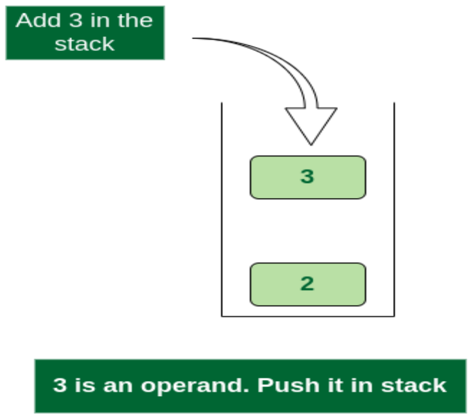
evaluatePostfix: Iterates through the postfix expression. If a digit is encountered, it is pushed onto the stack. If an operator is encountered, two elements are popped from the stack, the operation is performed, and the result is pushed back onto the stack.

Consider the expression: exp = **"2 3 1 * + 9 -"**

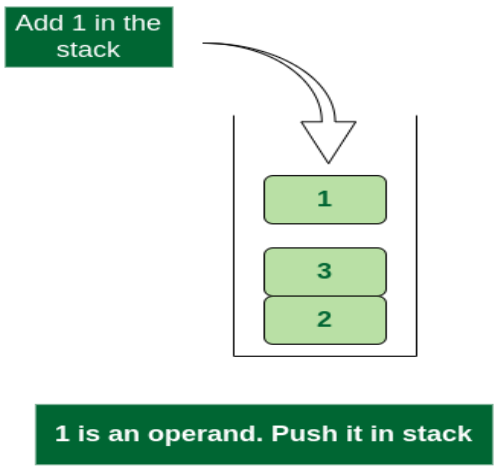
- Scan **2**, it's a number, So push it into stack. Stack contains '2'. Push 2 into stack



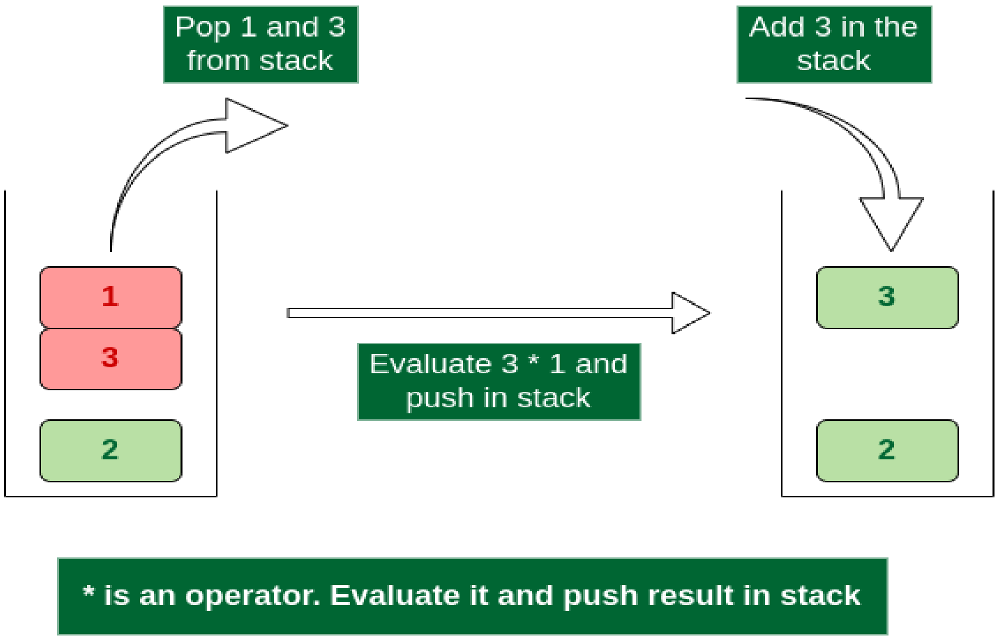
- Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top) Push 3 into stack



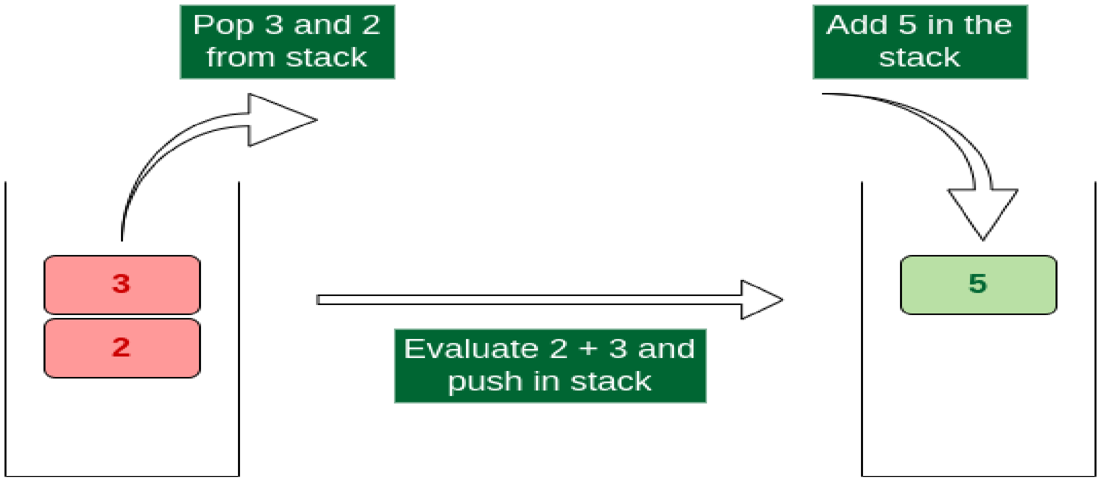
- Scan 1, again a number, push it to stack, stack now contains ‘2 3 1’ Push 1 into stack



- Scan *, it’s an operator. Pop two operands from stack, apply the * operator on operands. We get 3*1 which results in 3. We push the result 3 to stack. The stack now becomes ‘2 3’. Evaluate * operator and push result in stack

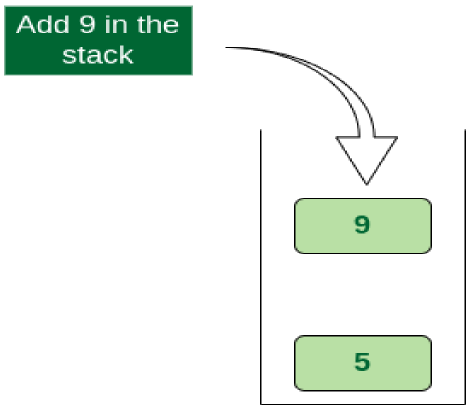

Pop 1 and 3 from stack
Add 3 in the stack
1
3
2
Evaluate 3 * 1 and push in stack
3
2
* is an operator. Evaluate it and push result in stack

- Scan +, it’s an operator. Pop two operands from stack, apply the + operator on operands. We get 3 + 2 which results in 5. We push the result 5 to stack. The stack now becomes 5. Evaluate + operator and push result in stack.



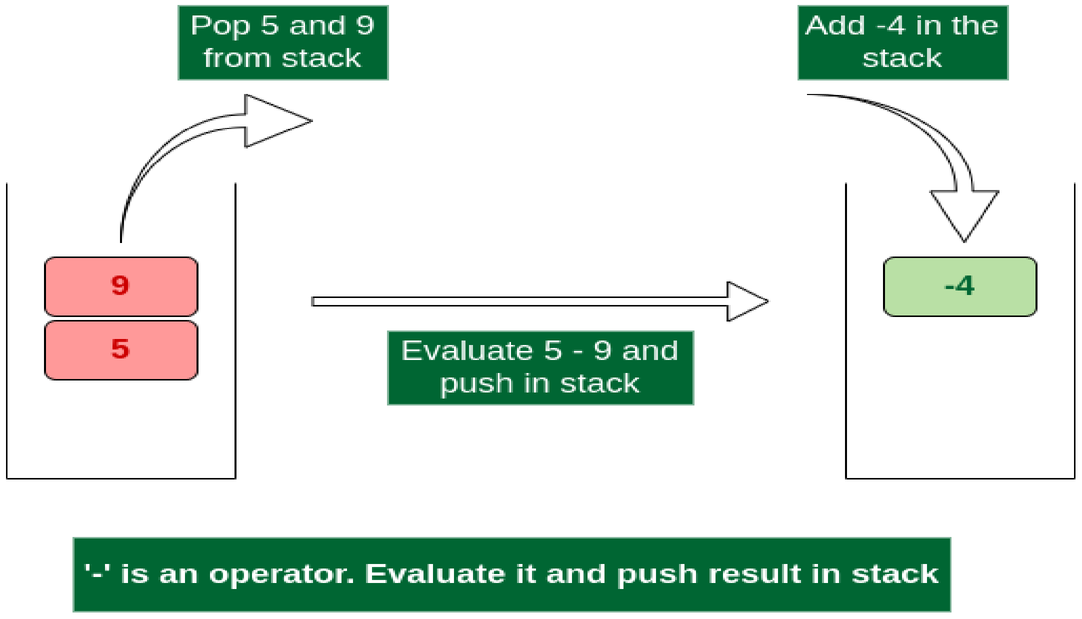
+ is an operator. Evaluate it and push result in stack

- Scan 9, it's a number. So we push it to the stack. The stack now becomes '5 9'. Push 9 into stack



9 is an operand. Push it in stack

- Scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result -4 to the stack. The stack now becomes '-4'. Evaluate '-' operator and push result in stack


Pop 5 and 9 from stack
Add -4 in the stack
9
5
-4
Evaluate 5 - 9 and push in stack
'-' is an operator. Evaluate it and push result in stack

- There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack). So, the result becomes **-4**.

ALGORITHM

1. Create an empty stack.

2. Initialize top to -1.

3. Push Operation:

    Input: Element item

    Check: If the stack is full (top >= MAX - 1), print "Stack Overflow" and return.

    Otherwise: Increment top and assign stack[top] = item

4. Pop Operation:

    Check: If the stack is empty (top < 0), print "Stack Underflow" and return 0.

    Otherwise: Return stack[top--].

5.Evaluate Postfix Expression

- **Input**: A postfix expression postfix.
- Initialize i to 0.
- Loop through each character ch in postfix until the end of the string:
    - **If ch is a digit**:
        - Convert ch to integer by ch - '0'.
        - Push the integer value onto the stack.
    - **If ch is an operator**:
        - Pop the top two elements from the stack. Let them be A and B.

        - Perform the operation B <operator> A:
            - + : val = B + A
            - - : val = B - A
            - * : val = B * A
            - / : val = B / A
        - Push the result val back onto the stack.
- **End Loop**.

6. The final result of the postfix expression is the top element of the stack.

PROGRAM

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

#define MAX 20

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

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

int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        return 0;
    }
    return stack[top--];
}

int evaluatePostfix(char postfix[]) {
    int i;
    char ch;
    int val;
    int A, B;

    for (i = 0; postfix[i] != '\0'; i++) {
        ch = postfix[i];

        if (isdigit(ch)) {
            push(ch - '0'); // Convert char to int and push to stack
        } else {
            A = pop();
            B = pop();
            switch (ch) {
                case '+':
                    val = B + A;
                    break;
                case '-':
                    val = B - A;
                    break;
                case '*':
                    val = B * A;
                    break;
                case '/':
                    val = B / A;
                    break;
            }
            push(val);
        }
    }
    return pop();
}

int main() {
    char postfix[MAX];
    printf("Enter the postfix expression: ");
    scanf("%s", postfix);
    printf("The result of the postfix expression is: %d\n", evaluatePostfix(postfix));
    return 0;
}
```

OUTPUT

Example 1:

```
Enter the postfix expression: 231*+9-
The result of the postfix expression is: -4
```

Example 2:

```
Enter the postfix expression: 52+83-*
The result of the postfix expression is: 35
```

EXPLANATION

Example 1: Input: "231*+9-"

Steps:

1. Push 2
2. Push 3
3. Push 1
4. Encounter *, pop 1 and 3, compute 3 * 1 = 3, push 3
5. Encounter +, pop 3 and 2, compute 2 + 3 = 5, push 5
6. Push 9
7. Encounter -, pop 9 and 5, compute 5 - 9 = -4, push -4

Output: -4

Example 2: Input: "52+83-*"

Steps:

1. Push 5
2. Push 2
3. Encounter +, pop 2 and 5, compute 5 + 2 = 7, push 7
4. Push 8
5. Push 3
6. Encounter -, pop 3 and 8, compute 8 - 3 = 5, push 5
7. Encounter *, pop 5 and 7, compute 7 * 5 = 35, push 35

Output: 35

VIVA QUESTIONS

1. Explain why postfix evaluation eliminates the need for parentheses in expressions.
2. What is the significance of postfix notation in terms of stack-based computation?
3. Why is postfix evaluation considered more efficient than infix evaluation in terms of computation and memory usage?
4. What happens if the postfix expression contains more operands than operators?
5. How are unary operators (like negation or factorial) handled in the evaluation process?

RESULT

Thus, the C program Evaluating Postfix Expressions using Stack was completed successfully.

Ex.No:8b **CONVERSION OF INFIX TO POSTFIX EXPRESSION**

AIM

To write a C program to implement the conversion of infix to postfix expression using Stack.

PRE LAB-DISCUSSION

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. An arithmetic expression may consist of more than one operator and two operands e.g. (A+B)*C(D/(J+D)).These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

Infix Expression:

It follows the scheme of <operand><operator><operand>i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g.,A+B

Postfix Expression:

It follows the scheme of <operand><operand><operator>i.e. an <operator> is succeeded by both the <operand>. E.g., AB+

Steps to convert Infix to Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

```
1.  Push “(“onto Stack, and add “)” to the end of X.
2.  Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is
empty.
3.  If an operand is encountered, add it to Y.
4.  If a left parenthesis is encountered, push it onto Stack.
5.  If an operator is encountered, then:
      1.  Repeatedly pop from Stack and add to Y each operator (on the top of Stack)
          which has the same precedence as or higher precedence than operator.
      2.  Add operator to Stack.
         [End ofIf]
6.  If a right parenthesis is encountered,then:
     1.  Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until
         a left parenthesis is encountered.
     2.  Remove the left Parenthesis.
        [End ofIf]
        [End of If]
7.  END.
```

ALGORITHM

1: Start.

2: Create a stack to store operand and operator.

3: In Postfix notation the operator follows the two operands and in the infix notation the operator is in between the two operands.

4: Consider the sum of A and B. Apply the operator "+" to the operands A and B and write the sum as A+B is INFIX. + AB is PREFIX. AB+ is POSTFIX

5: Get an Infix Expression as input and evaluate it by first converting it to postfix and then evaluating the postfix expression.

6: The expressions with in innermost parenthesis must first be converted to postfix so that they can be treated as single operands. In this way Parentheses can be successively eliminated until the entire expression is converted.

7: The last pair of parentheses to be opened with in a group of parentheses encloses the first expression with in that group to be transformed. This last-in first-out immediately suggests the use of Stack. Precedence plays an important role in the transforming infix to postfix.

8: Stop.

PROGRAM

```
// Applications of stack

#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 20

int top = -1;
char pop();
char stack[MAX];
void push(char item);

int operatorPrecedence(char symbol) {
    switch (symbol) {
        case '+': {
            return 2;
            break;
        }
        case '-': {
            return 2;
            break;
        }
        case '*': {
            return 4;
            break;
        }
        case '/': {
            return 4;
            break;
        }
        case '^': {
            return 6;
            break;
        }
        case '$': {
            return 6;
            break;
        }
        case '(': {
            return 1;
            break;
        }
        case ')': {
            return 1;
            break;
        }
        case '#': {
            return 1;
            break;
        }
    }
}

int isOperator(char symbol) {
    switch (symbol) {
        case '+': {
            return 1;
            break;
        }
        case '-': {
            return 1;
            break;
        }
        case '*': {
            return 1;
            break;
        }
        case '/': {
            return 1;
            break;
        }
        case '^': {
            return 1;
            break;
        }
        case '$': {
            return 1;
            break;
        }
        case '(': {
            return 1;
            break;
        }
        case ')': {
            return 1;
            break;
        }
        default: {
            return 0;
        }
    }
}

void convertInfixToPostfix(char infix[], char postfix[]) {
    int i, symbol, j = 0;
    stack[++top] = '#';
    for (i = 0; i < strlen(infix); i++) {
        symbol = infix[i];
        if (isOperator(symbol) == 0) {
            postfix[j] = symbol;
            j++;
        } else {
            if (symbol == '(') {
                push(symbol);
            } else if (symbol == ')') {
                while (stack[top] != '(') {
                    postfix[j] = pop();
                    j++;
                }
                pop();
            } else {
                if (operatorPrecedence(symbol) > operatorPrecedence(stack[top])) {
                    push(symbol);
                } else {
                    while (operatorPrecedence(symbol) <= operatorPrecedence(stack[top])) {
                        postfix[j] = pop();
                        j++;
                    }
                    push(symbol);
                }
            }
        }
    }
    while (stack[top] != '#') {
        postfix[j] = pop();
        j++;
    }
    postfix[j] = '\0';
}

void main() {
    char infix[20], postfix[20];
    printf("Enter the valid infix string:\n");
    gets(infix);
    convertInfixToPostfix(infix, postfix);
    printf("The corresponding postfix string is:\n");
    puts(postfix);
    getch();
```

```
}

void push(char item) {
    top++;
    stack[top] = item;
}

char pop() {
    char a;
    a = stack[top];
    top--;
    return a;
}
```

Output:

```
Enter the valid infix string: A+B*(C-D)
The corresponding postfix string is: ABCD-*+
```

EXPLANATION

1. Initialization:

Stack: #

2. Read 'A':

Operand, so append to postfix.
Postfix: A
Stack: #

3. Read '+':

Operator, push onto the stack.
Postfix: A
Stack: #+

4. Read 'B':

Operand, so append to postfix.
Postfix: AB

Stack: #+

5. Read '*':

Operator with higher precedence than '+', so push onto the stack.
Postfix: AB
Stack: #+*

6. Read '(':

Opening parenthesis, push onto the stack.
Postfix: AB
Stack: #+*(

7. Read 'C':

Operand, so append to postfix.
Postfix: ABC
Stack: #+*(

8. Read '-':

Operator, push onto the stack.
Postfix: ABC
Stack: #+*(-

9. Read 'D':

Operand, so append to postfix.
Postfix: ABCD
Stack: #+*(-

10. Read ')':

Closing parenthesis, pop until '(' is found.
Postfix: ABCD-
Stack: #+*

11. End of expression, pop remaining operators:

Postfix: ABCD-*+
Stack: #

VIVA QUESTIONS

1. What is the significance of the order of operations in infix expressions?
2. How does the stack ensure that the postfix expression is generated correctly?
3. What are the limitations of the infix to postfix conversion algorithm?
4. What happens when the algorithm encounters a closing parenthesis)?
5. What are some practical applications of converting infix expressions to postfix expressions?

RESULT

Thus, the C program to convert infix to postfix expression using Stack was completed successfully.

Ex.No:9 IMPLEMENTATION OF BINARY SEARCH TREES

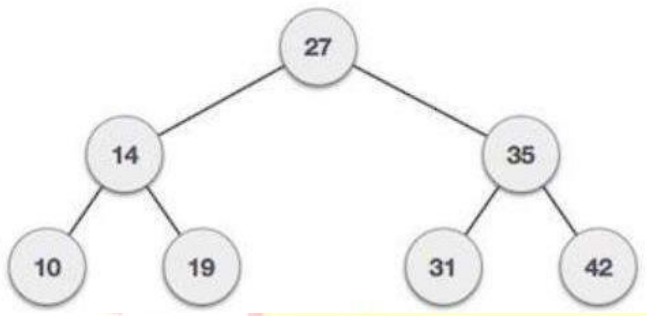
AIM

To write a C program to implement the binary search trees.

PRE LAB-DISCUSSION

A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties

- The left sub-tree of a node has key less than or equal to its parent node'skey.
- The right sub-tree of a node has key greater than or equal to its parent node's key.
- Thus, a binary search tree (BST) divides all its sub-trees into two segments;
- left sub- tree and right sub-tree and can be definedas
  left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)



Following are basic primary operations of a tree which are following.

- Search − search an element in atree.
- Insert − insert an element in atree.
- Delete – removes an existing node from thetree
- Preorder Traversal − traverse a tree in a preordermanner.
- Inorder Traversal − traverse a tree in an inordermanner.
- Postorder Traversal − traverse a tree in a postordermanner.

ALGORITHM

1: Start the process.

2: Initialize and declare variables.

3: Construct the Tree

4: Data values are given which we call a key and a binary search tree

5: To search for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node. If they match, return the root pointer.

6: If the key is less than the data value of the root node, repeat the process by using the left subtree.

7: Otherwise, repeat the same process with the right subtree until either a match is found or the subtree under consideration becomes an empty tree.

8: Terminate

PROGRAM

```
// Implementation of binary search trees

#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <stdlib.h>

struct tree
{
    int data;
    struct tree *lchild;
    struct tree *rchild;
} *t, *temp;

int element;

void inorder(struct tree *);
void preorder(struct tree *);
void postorder(struct tree *);

struct tree *create(struct tree *, int);
struct tree *find(struct tree *, int);
struct tree *insert(struct tree *, int);
struct tree *del(struct tree *, int);
struct tree *findmin(struct tree *);
struct tree *findmax(struct tree *);

int main(void)
{
    int ch;
    printf("BINARY SEARCH TREE\n\n");
    do
    {
```

```
printf("**************************\n");
printf("***      MAIN MENU      ***\n");
printf("**************************\n");
printf("1. Create\n");
printf("2. Insert\n");
printf("3. Delete\n");
printf("4. Find\n");
printf("5. FindMin\n");
printf("6. FindMax\n");
printf("7. Inorder\n");
printf("8. Preorder\n");
printf("9. Postorder\n");
printf("10. Exit\n");
printf("Enter your choice: ");
scanf("%d", &ch);
printf("\n");
switch (ch)
{
   case 1:
      printf("Enter the data: ");
      scanf("%d", &element);
      printf("\n");
      t = create(t, element);
      inorder(t);
      printf("\n\n");
      break;
   case 2:
      printf("Enter the data: ");
      scanf("%d", &element);
      printf("\n");
      t = insert(t, element);
      inorder(t);
      printf("\n\n");
      break;
   case 3:
      printf("Enter the data: ");
      scanf("%d", &element);
      printf("\n");
      t = del(t, element);
      inorder(t);
```

```
                printf("\n\n");
                break;
            case 4:
                printf("Enter the data: ");
                scanf("%d", &element);
                temp = find(t, element);
                if (temp->data == element)
                    printf("\nElement %d is at %d", element, temp);
                else
                    printf("\nElement is not found");
                printf("\n\n");
                break;
            case 5:
                temp = findmin(t);
                printf("Min element = %d", temp->data);
                printf("\n\n");
                break;
            case 6:
                temp = findmax(t);
                printf("Max element = %d", temp->data);
                printf("\n\n");
                break;
            case 7:
                inorder(t);
                printf("\n\n");
                break;
            case 8:
                preorder(t);
                printf("\n\n");
                break;
            case 9:
                postorder(t);
                printf("\n\n");
                break;
            case 10:
                printf("Thank you for using the binary search tree program!\n");
                exit(0);
        }
    } while (ch <= 10);
}
```

```
struct tree *create(struct tree *t, int element)
{
    t = (struct tree *)malloc(sizeof(struct tree));
    t->data = element;
    t->lchild = NULL;
    t->rchild = NULL;
    return t;
}

struct tree *find(struct tree *t, int element)
{
    if (t == NULL)
        return NULL;
    if (element < t->data)
        return (find(t->lchild, element));
    else if (element > t->data)
        return (find(t->rchild, element));
    else
        return t;
}

struct tree *findmin(struct tree *t)
{
    if (t == NULL)
        return NULL;
    else if (t->lchild == NULL)
        return t;
    else
        return (findmin(t->lchild));
}

struct tree *findmax(struct tree *t)
{
    if (t != NULL)
    {
        while (t->rchild != NULL)
            t = t->rchild;
    }
    return t;
```

```
}

struct tree *insert(struct tree *t, int element)
{
   if (t == NULL)
   {
      t = (struct tree *)malloc(sizeof(struct tree));
      t->data = element;
      t->lchild = NULL;
      t->rchild = NULL;
      return t;
   }
   else if (element < t->data)
   {
      t->lchild = insert(t->lchild, element);
   }
   else if (element > t->data)
   {
      t->rchild = insert(t->rchild, element);
   }
   else if (element == t->data)
   {
      printf("Element already present\n");
   }
   return t;
}

struct tree *del(struct tree *t, int element)
{
   if (t == NULL)
      printf("Element not found\n");
   else if (element < t->data)
      t->lchild = del(t->lchild, element);
   else if (element > t->data)
      t->rchild = del(t->rchild, element);
   else if (t->lchild && t->rchild)
   {
      temp = findmin(t->rchild);
      t->data = temp->data;
      t->rchild = del(t->rchild, t->data);
```

```
        temp = t;
        if (t->lchild == NULL)
            t = t->rchild;
        return t;
    }
    else if (t->rchild == NULL)
        t = t->lchild;
    free(temp);
    return t;
}

void inorder(struct tree *t)
{
    if (t == NULL)
        return;
    else
    {
        inorder(t->lchild);
        printf("\t%d", t->data);
        inorder(t->rchild);
    }
}

void preorder(struct tree *t)
{
    if (t == NULL)
        return;
    else
    {
        printf("\t%d", t->data);
        preorder(t->lchild);
        preorder(t->rchild);
    }
}

void postorder(struct tree *t)
{
    if (t == NULL)
        return;
    else
    {
        postorder(t->lchild);
        postorder(t->rchild);
        printf("\t%d", t->data);
    }
}
```

OUTPUT

```
BINARY SEARCH TREE

**************************
***     MAIN MENU     ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 1

Enter the data: 12

        12

**************************
***     MAIN MENU     ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
```

```
9. Postorder
10. Exit
Enter your choice: 2

Enter the data: 11

        11      12

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 2

Enter the data: 23

        11      12      23

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
```

```
Enter your choice: 3

Enter the data: 23

        11      12

**************************
***     MAIN MENU     ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 2

Enter the data: 24

        11      12      24

**************************
***     MAIN MENU     ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 4
Enter the data: 11

Element 11 is at 1395266672

**************************
***     MAIN MENU     ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 5

Min element = 11

**************************
***     MAIN MENU     ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 6

Max element = 24

**************************
***     MAIN MENU     ***
```

```
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 7

        11      12      24

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 8

        12      11      24

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
```

```
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 9

	11	24	12

**************************
***	MAIN MENU	***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 10

Thank you for using the binary search tree program!
```

EXPLANATION

Operations Breakdown

1. **Create**: Initializes a BST with the given element.
2. **Insert**: Inserts an element into the BST.
3. **Delete**: Deletes an element from the BST.
4. **Find**: Searches for an element in the BST.
5. **FindMin**: Finds the minimum element in the BST.
6. **FindMax**: Finds the maximum element in the BST.
7. **Inorder**: Displays the BST elements in inorder traversal.
8. **Preorder**: Displays the BST elements in preorder traversal.
9. **Postorder**: Displays the BST elements in postorder traversal.
10. **Exit**: Exits the program.

VIVA QUESTIONS

1. Why is it said that in-order traversal of a BST produces a sorted sequence?
2. Can a BST have duplicate values? Why or why not?
3. Explain how the BST property ensures efficient search operations.
4. What is the worst-case time complexity for BST operations, and under what conditions does it occur?
5. What are some common applications of BSTs?

RESULT

Thus, the C program to implement the binary search trees was completed successfully.

Ex.No:10a IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS - DEPTH FIRST SEARCH

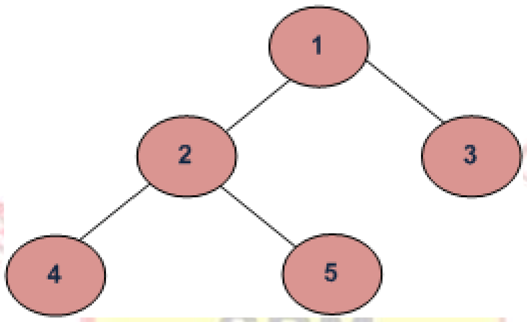
AIM

To write a C program for implementing the Tree traversal algorithm for Depth first traversal.

PRE LAB-DISCUSSION

DFS (Depth-first search) is a technique used for traversing trees or graphs. Here backtracking is used for traversal. In this traversal first, the deepest node is visited and then backtracks to its parent node if no sibling of that node exists.

Example:



Therefore, the Depth First Traversals of this Tree will be:
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1

ALGORITHM

1. Define a structure for a tree node with integer data, and left and right child pointers.
2. Function createNode to allocate memory for a new node, set its data, and initialize its child pointers to NULL.
3. Function insertNode to insert a new node into the binary search tree. If the tree is empty, create a new node. Otherwise, insert the node in the left subtree if the data is less than or equal to the current node's data, otherwise insert it in the right subtree.
4. Read the node values and insert them into the tree.
5. Perform and print preorder, inorder, and postorder traversals.
6. Free the allocated memory for the tree.

PROGRAM

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

```
// Definition of a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

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

// Preorder Traversal
void preorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    printf("%d ", node->data);
    preorderTraversal(node->left);
    preorderTraversal(node->right);
}
```

```
// Inorder Traversal
void inorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    inorderTraversal(node->left);
    printf("%d ", node->data);
    inorderTraversal(node->right);
}

// Postorder Traversal
void postorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    postorderTraversal(node->left);
    postorderTraversal(node->right);
    printf("%d ", node->data);
}

// Function to free memory allocated for the tree nodes
void freeTree(struct Node* root) {
    if (root == NULL)
        return;

    freeTree(root->left);
    freeTree(root->right);
    free(root);
}

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

    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);

    printf("Enter the node values:\n");
    for (int i = 0; i < numNodes; i++) {
        scanf("%d", &data);
        root = insertNode(root, data);
    }

    printf("\nPreorder traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");

    // Free allocated memory for the tree
    freeTree(root);

    return 0;
}
```

OUTPUT

```
Enter the number of nodes: 5
Enter the node values:
5 3 8 1 4

Preorder traversal: 5 3 1 4 8
Inorder traversal: 1 3 4 5 8
Postorder traversal: 1 4 3 8 5
```

EXPLANATION

Tree Structure:

```
       5
      / \
     3   8
    / \
   1   4
```

Node Insertion:
Insert 5:
Tree is empty. Create root node with data 5.
Insert 3:
3 <= 5, go left.
Left subtree is empty. Create node with data 3.
Insert 8:
8 > 5, go right.
Right subtree is empty. Create node with data 8.
Insert 1:
1 <= 5, go left to node 3.
1 <= 3, go left.
Left subtree of node 3 is empty. Create node with data 1.
Insert 4:
4 <= 5, go left to node 3.
4 > 3, go right.
Right subtree of node 3 is empty. Create node with data 4.

Traversals:

Preorder Traversal:	**Inorder Traversal:**	**Postorder Traversal:**
Visit node 5.	Visit node 1.	Visit node 1.
Visit node 3.	Visit node 3.	Visit node 4.
Visit node 1.	Visit node 4.	Visit node 3.
Visit node 4.	Visit node 5.	Visit node 8.
Visit node 8.	Visit node 8.	Visit node 5.

Memory Freeing: Recursively free memory for nodes 1, 4, 3, 8, and finally 5.

VIVA (PRE & POST LAB) QUESTIONS
1. Explain how DFS can be used to perform topological sorting in a directed acyclic graph (DAG).
2. How does DFS ensure that it goes as deep as possible along each branch before backtracking?
3. Can DFS be used on a binary search tree? How does its behavior differ from using DFS on a general tree?
4. Explain how in-order traversal of a binary search tree (BST) results in a sorted sequence of values.
5. How does DFS differ from Breadth-First Search (BFS)?

RESULT
Thus, the C program for implementing the Tree traversal algorithm for Depth first traversal was implemented successfully.

Ex.No: 10b

IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS- BREADTH FIRST SEARCH

AIM

To write a C program for implementing the Tree traversal algorithm for Breadth first traversal

PRE LAB-DISCUSSION

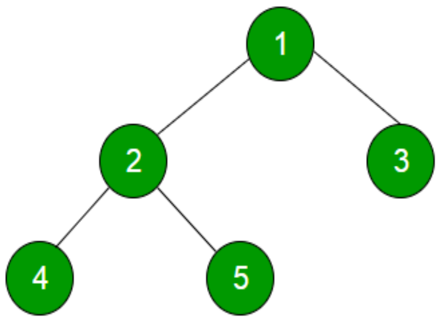
A breadth-first search is when you inspect every node on a level starting at the top of the tree and then move to the next level. It starts from the root node and visits all nodes of current depth before moving to the next depth in the tree.

Level Order Traversal (Breadth First Search or BFS) technique is defined as a method to traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level.

The main idea of level order traversal is to traverse all the nodes of a lower level before moving to any of the nodes of a higher level. This can be done in any of the following ways:

- **The naive one** (finding the height of the tree and traversing each level and printing the nodes of that level) : Find height of tree. Then for each level, run a recursive function by maintaining current height. Whenever the level of a node matches, print that node.
- **Efficiently using a queue**.: We need to visit the nodes in a lower level before any node in a higher level, this idea is quite similar to that of a queue. Push the nodes of a lower level in the queue. When any node is visited, pop that node from the queue and push the child of that node in the queue.This ensures that the node of a lower level are visited prior to any node of a higher level.

Example: Input:



Output:

```
1
2 3
4 5
```

PROGRAM

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

// Definition of a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

// Function to perform Breadth-First Search (BFS) traversal
void breadthFirstTraversal(struct Node* root) {
    if (root == NULL)
        return;

    // Queue for BFS implementation using an array
    struct Node* queue[100];  // Assuming a maximum of 100 nodes
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];
        printf("%d ", current->data);

        if (current->left != NULL)
            queue[rear++] = current->left;
        if (current->right != NULL)
            queue[rear++] = current->right;
    }
```

```
}

// Function to free memory allocated for the tree nodes
void freeTree(struct Node* root) {
    if (root == NULL)
        return;

    freeTree(root->left);
    freeTree(root->right);
    free(root);
}

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

    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);

    if (numNodes <= 0) {
        printf("Invalid number of nodes.\n");
        return 1;
    }

    printf("Enter the node values:\n");
    for (int i = 0; i < numNodes; i++) {
        scanf("%d", &data);
        if (root == NULL) {
            root = createNode(data);
        } else {
            struct Node* current = root;
            struct Node* parent = NULL;
            while (current != NULL) {
                parent = current;
                if (data <= current->data)
                    current = current->left;
                else
                    current = current->right;
            }
            if (data <= parent->data)
```

```
                parent->left = createNode(data);
            else
                parent->right = createNode(data);
        }
    }

    printf("\nBreadth-First Traversal: ");
    breadthFirstTraversal(root);
    printf("\n");

    // Free allocated memory for the tree
    freeTree(root);

    return 0;
}
```

OUTPUT

```
Enter the number of nodes: 5
Enter the node values:
4 5 2 3 7

Breadth-First Traversal: 4 2 5 3 7
```

EXPLANATION

Tree structure:

```
      4
    /   \
   2     5
    \     \
     3     7
```

1. **Node Insertion:**
   - Insert 4:
     - Tree is empty. Create root node with data 4.
   - Insert 5:
     - 5 > 4, go right.
     - Right subtree is empty. Create node with data 5.
   - Insert 2:
     - 2 <= 4, go left.
     - Left subtree is empty. Create node with data 2.
   - Insert 3:

        - 3 <= 4, go left to node 2.
        - 3 > 2, go right.
        - Right subtree of node 2 is empty. Create node with data 3.
    - Insert 7:
        - 7 > 4, go right to node 5.
        - 7 > 5, go right.
        - Right subtree of node 5 is empty. Create node with data 7.
2. **BFS Traversal:**
    - Initialize the queue with root node 4.
    - Process node 4: print 4, add 2 and 5 to the queue.
    - Process node 2: print 2, add 3 to the queue.
    - Process node 5: print 5, add 7 to the queue.
    - Process node 3: print 3.
    - Process node 7: print 7.
3. **Output:**
    - The BFS traversal prints nodes in the order 4, 2, 5, 3, 7.

VIVA QUESTIONS

1. Why is BFS called a level-order traversal?
2. How does BFS ensure that all nodes at the current level are processed before moving on to the next level?
3. Can BFS be used on a binary search tree? How does its behaviour differ from using BFS on a general tree?
4. Explain how BFS can be used to check if a binary tree is complete.
5. What is the significance of checking if a node is NULL before processing its children in the BFS algorithm?

RESULT

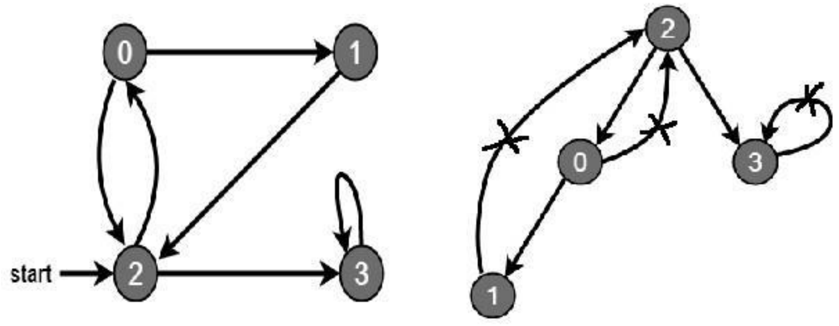
Thus, the C program for implementing the Tree traversal algorithm for Breadth first traversal was implemented successfully.

Ex.No:11a **IMPLEMENTATION OF GRAPH TRAVERSAL ALGORITHMS- DEPTH FIRST SEARCH**

AIM
To write a C program for implementing the graph traversal algorithm for Depth first traversal

PRE LAB-DISCUSSION

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.
Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree.
The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



ALGORITHM
1. Define an adjacency matrix adj[MAX][MAX] to represent the graph.
2. Define an array visited[MAX] to keep track of visited vertices.
3. Let n be the number of vertices in the graph.
4. Read the number of vertices n.
5. Read the number of edges.
6. For each edge, update the adjacency matrix:
   adj[v1][v2] = 1
   adj[v2][v1] = 1 (for undirected graph)
7. Define DFS(int v) where v is the starting vertex.

```
8. Mark v as visited: visited[v] = 1.
9. Print the vertex v.
10. For each vertex i from 0 to n-1:
      If adj[v][i] == 1 (there is an edge) and visited[i] == 0 (not visited):
   Call DFS(i) recursively.
11. Read the starting vertex.
12. Call DFS(start) to begin the DFS traversal from the starting vertex.
```

PROGRAM

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

#define MAX 100

int adj[MAX][MAX]; // Adjacency matrix to represent the graph
int visited[MAX];  // Array to track visited nodes
int n;             // Number of vertices in the graph

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

    for (int i = 0; i < n; i++) {
        if (adj[v][i] == 1 && !visited[i]) {
            DFS(i);
        }
    }
}

int main() {
    int edges, start, v1, v2;

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

    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    for (int i = 0; i < edges; i++) {
        printf("Enter edge (v1 v2): ");
        scanf("%d %d", &v1, &v2);
        adj[v1][v2] = 1;
        adj[v2][v1] = 1; // For an undirected graph
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    printf("Depth First Search starting from vertex %d:\n", start);
    DFS(start);

    return 0;
}
```

OUTPUT

```
Enter the number of vertices: 5
Enter the number of edges: 4
Enter edge (v1 v2): 0 1
Enter edge (v1 v2): 0 2
Enter edge (v1 v2): 1 3
Enter edge (v1 v2): 3 4

Enter the starting vertex: 0

Depth First Search starting from vertex 0:
0 1 3 4 2
```

EXPLANATION

Starting from vertex 0:

Visit 0, mark as visited.
From 0, visit 1 (next unvisited neighbor).
From 1, visit 3.
From 3, visit 4.
Backtrack to 1, then to 0.
From 0, visit 2 (next unvisited neighbor).

The output shows the order in which vertices are visited using DFS starting from vertex 0.

VIVA QUESTIONS

1. A person wants to visit some places. He starts from a vertex and then wants to visit every vertex till it finishes from one vertex, backtracks and then explore another vertex from same vertex. What algorithm he should use?
2. When the Depth First Search of a graph is unique?
3. In Depth First Search, how many times a node is visited?
4. Give the applications of DFS.
5. Depth First Search is equivalent to which of the traversal in the Binary Trees?

RESULT

Thus, the C program for implementing the graph traversal algorithm for Depth first traversal was implemented successfully

Ex.No:11b **IMPLEMENTATION OF GRAPH TRAVERSAL ALGORITHMS-BREADTH FIRST SEARCH**

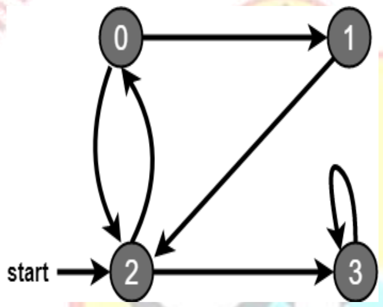
AIM

To write a C program for implementing the traversal algorithm for Breadth first traversal.

PRE LAB-DISCUSSION

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



ALGORITHM

1. Define an adjacency matrix adj[MAX][MAX] to represent the graph.
2. Define an array visited[MAX] to keep track of visited vertices.
3. Let n be the number of vertices in the graph.
4. Read the number of vertices n.
5. Read the number of edges.
6. For each edge, update the adjacency matrix:

   adj[v1][v2] = 1

   adj[v2][v1] = 1 (for undirected graph)
7. Define BFS(int start) where start is the starting vertex.
8. Initialize a queue.
9. Mark start as visited and enqueue it.

```
10. While the queue is not empty:
        Dequeue a vertex current.
        Print current.
        For each vertex i from 0 to n-1:
            If adj[current][i] == 1 (there is an edge) and visited[i] == 0 (not visited):
                Mark i as visited and enqueue it.
11. Read the starting vertex.
12. Call BFS(start) to begin the BFS traversal from the starting vertex.
```

PROGRAM

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

#define MAX 100

int adj[MAX][MAX]; // Adjacency matrix to represent the graph
int visited[MAX];  // Array to track visited nodes
int n;             // Number of vertices in the graph

void BFS(int start) {
    int queue[MAX], front = 0, rear = 0;

    printf("%d ", start);
    visited[start] = 1;
    queue[rear++] = start;

    while (front < rear) {
        int current = queue[front++];

        for (int i = 0; i < n; i++) {
            if (adj[current][i] == 1 && !visited[i]) {
                printf("%d ", i);
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
}

int main() {
    int edges, start, v1, v2;

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

    printf("Enter the number of edges: ");
    scanf("%d", &edges);

    for (int i = 0; i < edges; i++) {
        printf("Enter edge (v1 v2): ");
        scanf("%d %d", &v1, &v2);
        adj[v1][v2] = 1;
        adj[v2][v1] = 1; // For an undirected graph
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    printf("Breadth First Search starting from vertex %d:\n", start);
    BFS(start);

    return 0;
}
```

OUTPUT

```
Enter the number of vertices: 5
Enter the number of edges: 4
Enter edge (v1 v2): 0 1
Enter edge (v1 v2): 0 2
Enter edge (v1 v2): 1 3
Enter edge (v1 v2): 3 4

Enter the starting vertex: 0

Breadth First Search starting from vertex 0:
0 1 2 3 4
```

EXPLANATION

1. **Visit vertex 0**, mark it as visited, and add its neighbors (1 and 2) to the queue.
2. **Visit vertex 1** (next in the queue), mark it as visited, and add its unvisited neighbor (3) to the queue.
3. **Visit vertex 2**, mark it as visited. Vertex 2 has no unvisited neighbors.
4. **Visit vertex 3** (next in the queue), mark it as visited, and add its unvisited neighbor (4) to the queue.
5. **Visit vertex 4**, mark it as visited. Vertex 4 has no unvisited neighbors.
6. The queue is now empty, and the BFS traversal is complete.

The output shows the vertices in the order they are visited using BFS starting from vertex 0, ensuring that all vertices reachable from the starting vertex are visited level by level.

Starting from vertex 0:

1. **Visit 0, mark as visited.**
   - Queue: [0]
   - Visited: [1, 0, 0, 0, 0]
   - Output: 0
2. **From 0, visit 1 (next unvisited neighbor).**
   - Dequeue 0.
   - Enqueue 1.
   - Queue: [1]
   - Visited: [1, 1, 0, 0, 0]
   - Output: 0 1
3. **From 0, visit 2 (next unvisited neighbor).**
   - Enqueue 2.
   - Queue: [1, 2]
   - Visited: [1, 1, 1, 0, 0]
   - Output: 0 1 2
4. **From 1, visit 3 (next unvisited neighbor).**
   - Dequeue 1.
   - Enqueue 3.
   - Queue: [2, 3]
   - Visited: [1, 1, 1, 1, 0]
   - Output: 0 1 2 3
5. **From 3, visit 4 (next unvisited neighbor).**
   - Dequeue 2.
   - Dequeue 3.
   - Enqueue 4.
   - Queue: [4]
   - Visited: [1, 1, 1, 1, 1]
   - Output: 0 1 2 3 4
6. **No more neighbors to visit.**

- Dequeue 4.
- Queue is empty.

Final Output The BFS traversal from vertex 0 results in the following order: 0 1 2 3 4

VIVA QUESTIONS

1. Which data structure is used in standard implementation of Breadth First Search?
2. A person wants to visit some places. He starts from a vertex and then wants to visit every place connected to this vertex and so on. What algorithm he should use?
3. In BFS, how many times a node is visited?
4. Regarding implementation of Breadth First Search using queues, what is the maximum distance between two nodes present in the queue?
5. When the Breadth First Search of a graph is unique?

RESULT

Thus, the C program for implementing the graph traversal algorithm for Breadth first traversal was implemented successfully

Ex.No:12 IMPLEMENTATION OF DIJKSTRA’S ALGORITHM

AIM

To write a C program to implement the shortest path using Dijkstra’s algorithm.

PRE LAB-DISCUSSION

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Understanding the Working of Dijkstra's Algorithm:

A graph and source vertex are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.

Each Vertex in this Algorithm will have two properties defined for it:

1. Visited Property
2. Path Property

Let us understand these properties in brief.

1. **Visited Property:**

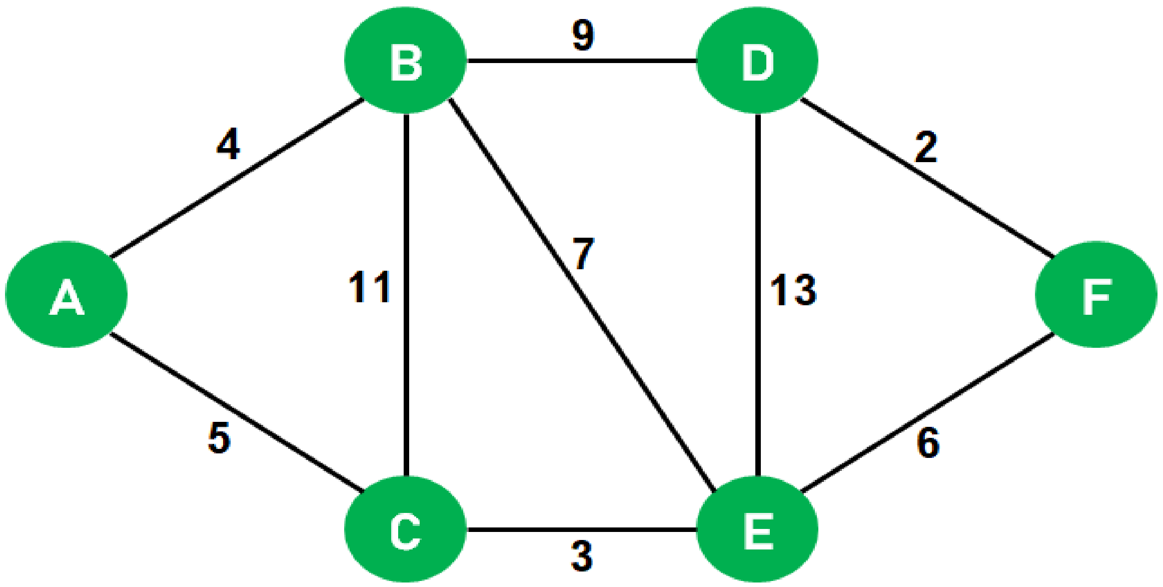
- The 'visited' property signifies whether or not the node has been visited.
- We are using this property so that we do not revisit any node.
- A node is marked visited only when the shortest path has been found.

2. **Path Property:**

- The 'path' property stores the value of the current minimum path to the node.
- The current minimum path implies the shortest way we have reached this node till now.

- This property is revised when any neighbor of the node is visited.
- This property is significant because it will store the final answer for each node.

Example:



1. A = 0
2. B = 4 (A -> B)
3. C = 5 (A -> C)
4. D = 4 + 9 = 13 (A -> B -> D)
5. E = 5 + 3 = 8 (A -> C -> E)
6. F = 5 + 3 + 6 = 14 (A -> C -> E -> F)

ALGORITHM

1. Read the number of vertices n and the number of edges edges.

2. Initialize the adjacency matrix graph with zeros.

3. Read each edge (u, v, w) and populate the adjacency matrix with weights.

4. Read the source vertex src.

5. Initialize dist[] array where dist[i] will hold the shortest distance from src to vertex i. Set all distances to infinity (INF) except the distance to the source itself, which is zero.

6. Initialize a visited[] array to keep track of vertices for which the minimum distance from the source is calculated. Set all entries to 0 (false).

7. Find the Vertex with Minimum Distance:

Create a helper function minDistance() that scans the dist[] array to find the vertex with the minimum distance that hasn't been visited yet.

8. Repeat the following steps n-1 times (for each vertex):

   Select the vertex u with the minimum distance from the dist[] array using minDistance(). Mark vertex u as visited. Update dist[] for each adjacent vertex v of u. For each vertex v: Check if v is not visited. Check if there is an edge from u to v.

   Check if the total weight of the path from the source to v through u is smaller than the current value of dist[v]. If so, update dist[v].

9. Print the Result:

10. After the main loop completes, print the shortest distances from the source to all vertices.

PROGRAM

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

#define MAX 100
#define INF INT_MAX

int n; // Number of vertices in the graph
int graph[MAX][MAX]; // Adjacency matrix representation of the graph

int minDistance(int dist[], int visited[]) {
    int min = INF, min_index;
    for (int v = 0; v < n; v++) {
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}

void dijkstra(int src) {
    int dist[MAX];
    int visited[MAX] = {0};

    for (int i = 0; i < n; i++) {
        dist[i] = INF;
    }
    dist[src] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, visited);
        visited[u] = 1;

        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printf("Vertex \t Distance from Source\n");
    for (int k = 0; k < n; k++) {
        printf("%d \t %d\n", k, dist[k]);
    }
}

int main() {
    int edges, u, v, w, src;

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

    printf("Enter the number of edges: ");
    scanf("%d", &edges);

    // Initialize the graph with 0s
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            graph[i][j] = 0;
        }
    }

    for (int m = 0; m < edges; m++) {
        printf("Enter edge (u v w): ");
        scanf("%d %d %d", &u, &v, &w);
        graph[u][v] = w;
        graph[v][u] = w; // If the graph is undirected
    }

    printf("Enter the source vertex: ");
    scanf("%d", &src);

    dijkstra(src);

    return 0;
}
```

OUTPUT

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter edge (u v w): 0 1 10
Enter edge (u v w): 0 4 20
Enter edge (u v w): 1 2 10
Enter edge (u v w): 1 3 50
Enter edge (u v w): 1 4 10
Enter edge (u v w): 2 3 10
Enter edge (u v w): 3 4 30

Enter the source vertex: 0

Vertex    Distance from Source
0         0
1         10
2         20
3         30
4         20
```

EXPLANATION

Initialization:

- Number of vertices: 5
- Number of edges: 7
- The adjacency matrix is initialized with all zeros.
- The edges are then populated into the adjacency matrix as input.

Graph Representation: The adjacency matrix after input:

```
0    10   0    0    20
10    0   10   50   10
0    10   0    10   0
0    50   10   0    30
20  10    0    30   0
```

1. **Dijkstra's Algorithm Execution:**
   - Start from vertex 0.
   - Set distance of the source vertex (0) to 0.
   - Initialize all other distances to infinity.
2. **Step-by-Step Execution:**
   - **Step 1:** Visit vertex 0 (distance 0).
     - Update distances of neighbors: 1 (10), 4 (20).
   - **Step 2:** Visit vertex 1 (distance 10).
     - Update distances of neighbors: 2 (20), 3 (60), 4 (20).
   - **Step 3:** Visit vertex 4 (distance 20).
     - Update distances of neighbors: 3 (50).
   - **Step 4:** Visit vertex 2 (distance 20).
     - Update distances of neighbors: 3 (30).
   - **Step 5:** Visit vertex 3 (distance 30).
     - No further updates as all neighbors have been visited or have shorter paths already.

Final Result:

The shortest distances from the source vertex (0) to all other vertices are:

- Vertex 0: Distance 0
- Vertex 1: Distance 10
- Vertex 2: Distance 20
- Vertex 3: Distance 30
- Vertex 4: Distance 20

The output shows the minimum distance from the source vertex to each of the other vertices using Dijkstra's algorithm.

VIVA QUESTIONS

1. How to find the adjacency matrix?
2. Which algorithm solves the single pair shortest path problem?
3. How to find shortest distance from source vertex to target vertex?
4. What is the maximum possible number of edges in a directed graph with no self-loops having 8vertices?
5. Does Dijkstra’s Algorithm work for both negative and positive weights?

RESULT

Thus, the C program to implement shortest path using Dijkstra’s algorithm was completed successfully.

Topic Beyond Syllabus

Ex.No:13 **HASHING WITH SEPARATE CHAINING**

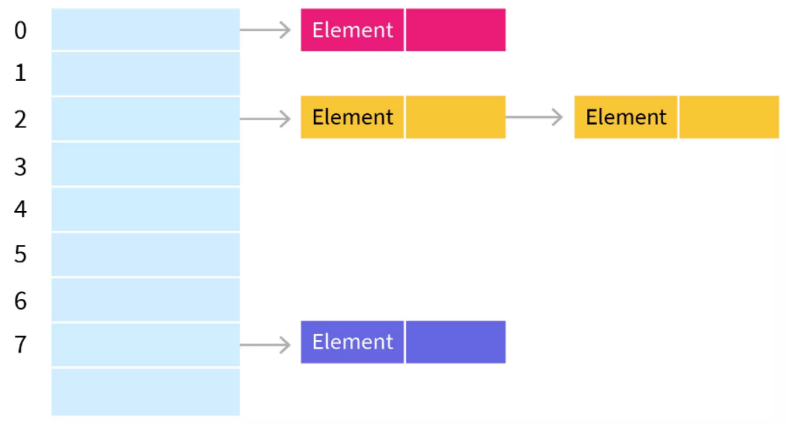
AIM

To write a C program to implement the concept of hashing using separate chaining.

PRE LAB-DISCUSSION

In hashing there is a hash function that maps keys to some values. But these hashing function may lead to collision that is two or more keys are mapped to same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash functionvalue.

Let's create a hash function, such that our hash table has 'N' number of buckets. To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hashfunction. **h(key) = key % table size**



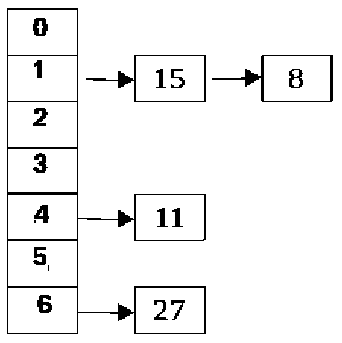
Example: hashIndex = key % no of Buckets

Insert: Move to the bucket corresponds to the above calculated hash index and insert the new node at the end of the list.

Here hash index = keys (15,11,27,8) % 7. Each key gets modulo with 7 to get the respected index value. Ex: 15%7=1,11%7=4,27%7=6 & 8%7= 1. % means, it takes the reminder value.

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



ALGORITHM

```
1: Start
2: Create Table size
3: Create hash function
4: To insert a node into the hash table, we need to find the hash index for the given key. And it
   could be calculated using the hash function.
5: Display hash entry.
6: Stop
```

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 3
struct node
{
    int data;
    struct node *next;
};
struct node *head[TABLE_SIZE] = {NULL}, *c, *p;
void insert(int i, int val)
{
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    newnode->data = val;
    newnode->next = NULL;
    if (head[i] == NULL)
        head[i] = newnode;
    else
    {
        c = head[i];
        while (c->next != NULL)
            c = c->next;
        c->next = newnode;
    }
}
void display(int i)
{
    if (head[i] == NULL)
    {
        if (i == 0)
            printf("No Hash Entry");
        return;
    }
    else
    {
        printf("%d ->", head[i]->data);
        for (c = head[i]->next; c != NULL; c = c->next)
            printf("%d ->", c->data);
    }
}
void main()
{
    int opt, val, i;
    printf("HASHING WITH SEPARATE CHAINING\n\n");
    while (1)
    {
        printf("********************\n");
        printf("*     MAIN MENU     *\n");
        printf("********************\n");
        printf("1. Insert\n");
        printf("2. Display\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &opt);
        switch (opt)
        {
            case 1:
                printf("\nEnter a value to insert into the hash table:\n");
                scanf("%d", &val);
                i = val % TABLE_SIZE;
                insert(i, val);
                printf("\n");
                break;
            case 2:
                for (i = 0; i < TABLE_SIZE; i++)
                {
                    printf("\nHash entries at index %d\n", i);
                    display(i);
                }
                printf("\n\n");
                break;
            case 3:
                printf("\nThank you for using the program. Exiting...\n");
                exit(0);
        }
    }
}
```

OUTPUT

```
HASHING WITH SEPARATE CHAINING

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
5

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
9

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
```

```
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
4

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
6

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
7

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
3. Exit
Enter your choice: 2

Hash entries at index 0
9 ->6 ->

Hash entries at index 1
```

```
4 ->7 ->

Hash entries at index 2
5 ->

*********************
*     MAIN MENU     *
*********************
1. Insert
2. Display
3. Exit
Enter your choice: 3

Thank you for using the program. Exiting...
```

EXPLANATION

Insert a value into the hash table:

- User chooses to insert a value.
- Enter a value, for example, 5.
- The index for insertion is calculated as 5 % 3 = 2.
- 5 is inserted at index 2.

Insert another value into the hash table:

- Enter a value, for example, 9.
- The index for insertion is calculated as 9 % 3 = 0.
- 9 is inserted at index 0.

Insert another value into the hash table:

- Enter a value, for example, 4.
- The index for insertion is calculated as 4 % 3 = 1.
- 4 is inserted at index 1.

Insert another value into the hash table:

- Enter a value, for example, 6.
- The index for insertion is calculated as 6 % 3 = 0.
- 6 is inserted at index 0, and since 9 is already present at index 0, 6 is appended after 9.

Insert another value into the hash table:

- Enter a value, for example, 7.
- The index for insertion is calculated as 7 % 3 = 1.
- 7 is inserted at index 1, and since 4 is already present at index 1, 7 is appended after 4.

Display the hash table:

- The program prints the contents of each index in the hash table.

Index 0: Contains values 9 and 6.
Index 1: Contains values 4 and 7.
Index 2: Contains value 5.

VIVA QUESTIONS

1. If several elements are competing for the same bucket in the hash table, what is itcalled?
2. How to insert a node in hashtable?
3. What is sizeof()function?
4. How to delete a node from hashtable.
5. What is Index value?

RESULT

Thus, the C program to implement hashing using separate chaining was completed successfully.