SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203.

DEPARTMENT OF INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

AND

DEPARTMENT OF CYBER SECURITY

ACADEMIC YEAR: 2025-2026 (ODD SEMESTER)

LAB MANUAL

(REGULATION - 2023)

IT3363 – PROGRAMMING AND DATA STRUCTURES LABORATORY

THIRD SEMESTER

Prepared by

Dr. S. Jeyalakshmi, Associate Professor / AI&DS

Dr. A. Vidhya, Assistant Professor / CSE

Ms. V. Prema, Assistant Professor / CSE

Ms. T. Sathya, Assistant Professor / CYS

Ms. M. Abinaya, Assistant Professor / AI&DS

Ms. M. Mohanapriya, Assistant Professor / CSE

Ms. N. J. Nithya Nandhini, Assistant Professor / IT

Ms. S. Priya, Assistant Professor / IT

TABLE OF CONTENTS

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.

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 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	PO												PSO			
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
1	3	-	-	-	2	-	-	-	-	-	-	-	3	2	-	-
2	2	2	-	-	-	-	-	-	-	-	-	-	3	3	-	-
3	2	2	-	-	-	-	-	-	-	-	-	-	3	-	-	-
4	3	-	-	-	-	-	-	-	-	-	-	-	3	2	-	-
5	3	3	3	-	2	-	-	-	-	-	-	-	3	2	-	-
Average	**2.6**	**2.3**	**3.0**	**-**	**2.0**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**3.0**	**2.3**	**-**	**-**

COURSE OUTCOMES:

Course Name: IT3363 - PROGRAMMING AND DATA STRUCTURES LABORATORY

Year of study: 2025 –2026

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	PO	Value	Justification
IT3363.1	PO1	3	Able to write, debug, and execute programs in a chosen programming language
	PO5	2	Develop a strong understanding of core data structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables.
IT3363.2	PO1	2	Implement and analyze fundamental algorithms for searching, sorting, and traversal of arrays.
	PO2	2	Enhance their problem-solving skills by applying appropriate data structures and algorithms to solve computational problems.
IT3363.3	PO1	2	Gain practical experience by working on projects that require the application of programming and data structure concepts.
	PO2	2	Develop collaboration skills by working in teams to complete lab assignments and projects.
IT3363.4	PO1	3	Learn effective debugging techniques and how to write test cases to ensure their code is robust and error-free.

Course Outcome	PO	Value	Justification
IT3363.5	PO1	3	Apply theoretical concepts learned in lectures to practical scenarios in the laboratory.
	PO2	3	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	3.0	-	2.0	-	-	-	-	-	-	-

CO-PSO Matrix:

CO	PSO1	PSO2	PSO3	PSO4
IT3363.1	3	2	-	-
IT3363.2	3	3	-	-
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	Understand the importance of writing efficient and optimized code.
IT3363.2	PSO1	3	Ability to apply appropriate data structures to solve practical problems.
	PSO2	3	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.

Course Outcome	PSO	Value	Justification
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	2.3	-	-

ASSESSMENT METHOD

i) EVALUATION PROCEDURE FOR EACH EXPERIMENTS

S. No.	Description	Marks
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	Marks
1	Conduction & Execution of Experiment	50
2	Record	17
3	Model Test	33
	Total	100

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:**
    - num1, num2 for user input
    - sum, difference, product, quotient, remainder for arithmetic results
    - quo for floating-point quotient
    - result, result1 for storing Boolean / logical results
3. **Input two integers:**
    - Read num1
    - Read num2
4. **Perform Arithmetic Operations:**
    - Compute sum = num1 + num2
    - Compute difference = num1 - num2
    - Compute product = num1 * num2
    - If num2 ≠ 0, compute:
        - quotient = num1 / num2
        - quo = (float)num1 / num2
        - remainder = num1 % num2
    - Else, print error message for division by zero
5. **Display Arithmetic Results**
6. **Perform Relational Operations:**
    - Compare num1 and num2 using:
        - >, >=, <, <=, ==, !=
    - Store and display results
7. **Perform Logical Operations:**
    - Evaluate expressions using:
        - Logical AND (&&)
        - Logical OR (||)
        - Logical NOT (!)
    - Store and display results
8. **Demonstrate Increment/Decrement:**
    - Print value of num1
    - Apply ++num1 and display
    - Apply num1++ and display
    - Print updated num1
9. **Perform Bitwise Operations:**
    - Apply &, |, ^, ~, <<, >>
    - Store and display results
10. **Use Conditional (Ternary) Operator:**
    - Check if num1 > num2, print TRUE or FALSE

11. **Use sizeof Operator:**
    - Display size of:
        - num1
        - int, float, double, char
12. **End**

PROGRAM

```
#include <stdio.h>
void main( )
{
        int num1, num2;
        int sum, difference, product, quotient, remainder;
        float quo;
        int result, result1;

        // Input: Read two integers from the user
        printf("Enter the first integer: ");
        scanf("%d", &num1);
        printf("Enter the second integer: ");
        scanf("%d", &num2);

        // Arithmetic Operators
        sum = num1 + num2;
        difference = num1 - num2;
        product = num1 * num2;
        quotient = num1 / num2;
        remainder = num1 % num2;
        quo = (float)num1 / num2;
        printf("\nEXPRESSIONS WITH ARITHMETIC OPERATORS\n");
        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);
        printf("Decimal Quotient : %d / %d = %f\n", num1, num2, quo);

        //Relational Operators
        printf("\nEXPRESSIONS WITH RELATIONAL OPERATORS\n");
        printf("%d Greater than %d ?: %d\n", num1, num2, num1 > num2);
        printf("%d Greater than or Equal %d ?: %d\n", num1, num2, num1 >= num2);
        result = num1 < num2;
        printf("%d Smaller than %d ?: %d\n", num1, num2, result);
        result1 = num1 <= num2;
        printf("%d Smaller than or Equal %d ?: %d\n", num1, num2, result1);
```

```
    printf("%d Equal to %d ?: %d\n", num1, num2, num1 == num2);
    result1 = num1 != num2;
    printf("%d Not Equal to %d ?: %d\n", num1, num2, result1);


    //Logical Operators
    printf("\nEXPRESSIONS WITH LOGICAL OPERATORS\n");
    printf("(%d > %d) Logical AND (%d > 10) ?: %d\n", num1, num2, num1, (num1 > num2) && (num1 >
                                                                                    10));
    printf("(%d > %d) Logical OR (%d > 10) ?: %d\n", num1, num2, num1, (num1 > num2) || (num1 > 10));;
    result = !(num1 < num2);
    printf("Negation(%d Smaller than %d) ?: %d\n", num1, num2, result);

    //Increment/Decrement Operators
    printf("\nEXPRESSIONS WITH INCREMENT / DECREMENT OPERATORS\n");
    printf("num1 value: %d\n", num1);
    printf("++num1 value: %d\n", ++num1);
    printf("num1++ value: %d\n", num1++);
    printf("num1 value: %d\n", num1);

    //Bitwise Operators
    printf("\nEXPRESSIONS WITH BITWISE OPERATORS\n");
    printf("%d Bitwise AND %d: %d\n", num1, num2, num1 & num2);
    printf("%d Bitwise OR %d: %d\n", num1, num2, num1 | num2);
    printf("%d Bitwise XOR %d: %d\n", num1, num2, num1 ^ num2);
    result = ~num1;
    printf("Complement(%d): %d\n", num1, result);
    printf("%d Left Shift 4: %d\n", num1, num1 << 4);
    printf("%d Right Shift 2: %d\n", num1, num1 >> 2);

    //Conditional Operator
    printf("\nEXPRESSIONS WITH CONDITIONAL OPERATOR\n");
    printf("(%d > %d ? TRUE: FALSE) :", num1, num2);
    (num1>num2?printf("TRUE"):printf("FALSE"));

    //Sizeof Operator
    printf("\nEXPRESSIONS WITH SIZEOF OPERATOR\n");
    printf("SIZE OF(%d): %d\n", num1, sizeof(num1));
    printf("SIZE OF(INTEGER): %d\n", sizeof(int));
    printf("SIZE OF(FLOAT): %d\n", sizeof(float));
    printf("SIZE OF(DOUBLE): %d\n", sizeof(double));
    printf("SIZE OF(CHARACTER): %d\n", sizeof(char));
}
```

OUTPUT:

```
Enter the first integer: 25
Enter the second integer: 10

EXPRESSIONS WITH ARITHMETIC OPERATORS
Sum: 25 + 10 = 35
Difference: 25 - 10 = 15
Product: 25 * 10 = 250
Quotient: 25 / 10 = 2
Remainder: 25 % 10 = 5
Decimal Quotient : 25 / 10 = 2.500000

EXPRESSIONS WITH RELATIONAL OPERATORS
25 Greater than 10 ?: 1
25 Greater than or Equal 10 ?: 1
25 Smaller than 10 ?: 0
25 Smaller than or Equal 10 ?: 0
25 Equal to 10 ?: 0
25 Not Equal to 10 ?: 1

EXPRESSIONS WITH LOGICAL OPERATORS
(25 > 10) Logical AND (25 > 10) ?: 1
(25 > 10) Logical OR (25 > 10) ?: 1
Negation(25 Smaller than 10) ?: 1

EXPRESSIONS WITH INCREMENT / DECREMENT OPERATORS
num1 value: 25
++num1 value: 26
num1++ value: 26
num1 value: 27

EXPRESSIONS WITH BITWISE OPERATORS
27 Bitwise AND 10: 10
27 Bitwise OR 10: 27
27 Bitwise XOR 10: 17
Complement (27): -28
27 Left Shift 4: 432
27 Right Shift 2: 6

EXPRESSIONS WITH CONDITIONAL OPERATOR
(27 > 10 ? TRUE: FALSE) :TRUE
EXPRESSIONS WITH SIZEOF OPERATOR
SIZE OF(27): 4
```

SIZE OF(INTEGER): 4
SIZE OF(FLOAT): 4
SIZE OF(DOUBLE): 8
SIZE OF(CHARACTER): 1

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.

i) Write a program to find the roots of a quadratic equation: $ax^2 + bx + c = 0$
root1 = (-b + sqrt (b^2-4ac)) / 2a root2 = (-b - sqrt (b^2-4ac)) / 2a
The program should request for the values of the constants a, b and c and print the values of 2 roots: root1 and root2. Use the following rules:

(a) No solution, if both a and b are zero.
(b) There is only one root, if a = 0 (root = - c / b).
(c) The roots are complex, if $b^2 – 4ac$ is negative.
(d) Otherwise there are 2 real roots

ALGORITHM:

1. **Start**
2. **Input** the coefficients a, b, and c.
3. **Check if both a and b are 0**
   → If true:
   → Display: "No solution"
   → **Stop**
4. **Check if a is 0 (i.e., linear equation):**
   → If true:
   → Compute root: root = -c / b
   → Display: "Only one root: root"
   → **Stop**
5. **Compute the discriminant:**
   → d = b² - 4ac
6. **Check the value of discriminant d:**
   - **If d = 0:**
     → Roots are **real and equal**
     → Compute: root = -b / (2a)
     → Display: "Roots are real and equal: root"
   - **If d > 0:**
     → Roots are **real and unequal**
     → Compute:
     r1 = (-b + √d) / (2a)
     r2 = (-b - √d) / (2a)
     → Display: "Roots are real and unequal: r1 and r2"
   - **If d < 0:**
     → Roots are **complex**
     → Compute:
     RealPart = -b / (2a)
     ImagPart = √(-d) / (2a)
     → Display:
     "Root1 = RealPart + i × ImagPart"
     "Root2 = RealPart - i × ImagPart"
7. **End**

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int a,b,c,d;
        float root,r1,r2,rp,ip;
        printf("\n Enter the values for a, b and c:");
        scanf("%d%d%d",&a, &b, &c);
        d = b*b - 4*a*c;
        if ((a==0) && (b==0))
        {
                printf("\na: %d\tb: %d\n There is no solution");
        }
        else if (a==0)
        {
                root = -1 * (float)c / b;
                printf("\n There is only one root: %.2f",root);
        }
        else if (d==0)
        {
                root = -1 * (float)b / (2*a);
                printf("\n The roots are real and equal:%.2f", root);
        }
        else if (d<0)
        {
                d = -1 * d;
                rp = -b / (2 * (float)a);
                ip = sqrt(d) / (2*a);
                printf("\n Roots are complex");
                printf("\n Root1:%.2f +i %.2f",rp,ip);
                printf("\n Root2:%.2f -i %.2f",rp,ip);
        }
        else
        {
                r1 = (-1 * b + sqrt(d)) / (2*a);
                r2 = (-1 * b - sqrt(d)) / (2*a);
                printf("\n Roots are real and unequal\nroot1: %.2f\nroot2: %.2f",r1,r2);
        }
}
```

OUTPUT:

Enter the values for a, b and c: 0 0 2
There is no solution

Enter the values for a, b and c: 1 4 1
Roots are real and unequal
root1: -0.27
root2: -3.73

Enter the values for a, b and c: 0 5 7
There is only one root: -1.40

Enter the values for a, b and c: 4 4 8
Roots are complex
Root1:-0.50 +i 1.32
Root2:-0.50 -i 1.32

Enter the values for a, b and c:4 4 1
The roots are real and equal:-0.50

ii) An electricity board charges the following rates for the use of electricity:

If the number of units consumed is less than or equal to 500 units, the charges are

0 – 100 units: Nil
101 – 200 units: Rs.2.35 / unit
201 – 400 units: Rs.4.70 / unit
401 – 500 units: Rs.6.30 / unit

If the number of units consumed is above 500 units, the charges are

0 – 100 units: Nil
101 – 400 units: Rs.4.70 / unit
401 – 500 units: Rs.6.30 / unit
501 – 600 units: Rs.8.40 / unit
601 – 800 units: Rs.9.45 / unit
801 – 1000 units: Rs.10.50 / unit
Above 1000 units: Rs.11.55 / unit

Write a program to read the names of users and number of units consumed and print the charges with names.

ALGORITHM:

1. **Start**
2. **Input** the consumer name and the number of units consumed.
3. **Initialize** a variable amount to store the total cost.
4. **Check if units ≤ 500**
   - If **units ≤ 100** → amount = 0

- Else if **units ≤ 200** →
  amount = (units - 100) × 2.35
- Else if **units ≤ 400** →
  amount = (100 × 2.35) + (units - 200) × 4.70
- Else (**units ≤ 500**) →
  amount = (100 × 2.35) + (200 × 4.70) + (units - 400) × 6.30

5. **Else if units > 500**
   - If **units ≤ 600** →
     amount = (300 × 4.70) + (100 × 6.30) + (units - 500) × 8.40
   - Else if **units ≤ 800** →
     amount = (300 × 4.70) + (100 × 6.30) + (100 × 8.40) + (units - 600) × 9.45
   - Else if **units ≤ 1000** →
     amount = (300 × 4.70) + (100 × 6.30) + (100 × 8.40) + (200 × 9.45) + (units - 800) × 10.50
   - Else →
     amount = (300 × 4.70) + (100 × 6.30) + (100 × 8.40) + (200 × 9.45) + (200 × 10.50) + (units - 1000) × 11.55
6. **Output**:
   - Consumer name
   - Units consumed
   - Total amount to be paid (formatted to two decimal places)
7. **End**

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int units;
        float amount;
        char name[20];
        printf("\nEnter the Consumer name:");
        scanf("%s",name);
        printf("\nEnter the Number of Units consumed:");
        scanf("%d",&units);

        if ( units <= 500)
        {
                if (units <= 100)
                        amount = 0.0;
                else if (units <= 200)
                        amount = (units-100) * 2.35;
                else if (units <= 400)
                        amount = (100 * 2.35) + ((units-200) * 4.70);
```

```
            else
                amount = (100 * 2.35) + (200 * 4.70) + ((units-400) * 6.30);
        }
        else
        {
            if (units <= 600)
                amount = (300 * 4.70) + (100 * 6.30) + ((units-500) * 8.40);
            else if (units <= 800)
                amount = (300 * 4.70) + (100 * 6.30) + (100 * 8.40) + ((units-600) * 9.45);
            else if (units <= 1000)
                amount = (300 * 4.70) + (100 * 6.30) + (100 * 8.40) + (200 * 9.45) + ((units-800)*10.50);
            else
                amount = (300 * 4.70) + (100 * 6.30) + (100 * 8.40) + (200 * 9.45) + (200 * 10.50) +
                        ((units-1000)*11.55);
        }
        printf("\nConsumed Name: %s", name);
        printf("\nUnits Consumed: %d", units);
        printf("\nAmount to be paid: %.2f", amount);
}
```

OUTPUT:

i) Enter the Consumer name:Aaaaaa
Enter the Number of Units consumed:90

Consumed Name: Aaaaaa
Units Consumed: 90
Amount to be paid: 0.00

ii) Enter the Consumer name:Bbbbbb
Enter the Number of Units consumed:165

Consumed Name: Bbbbbb
Units Consumed: 165
Amount to be paid: 152.75

iii) Enter the Consumer name:Ccccccc
Enter the Number of Units consumed:333

Consumed Name: Ccccccc
Units Consumed: 333
Amount to be paid: 860.10

iv) Enter the Consumer name:Dddddd
Enter the Number of Units consumed:478

Consumed Name: Dddddd
Units Consumed: 478
Amount to be paid: 1666.40

v) Enter the Consumer name:Ffffff
Enter the Number of Units consumed:585

Consumed Name: Ffffff
Units Consumed: 585
Amount to be paid: 2754.00

vi) Enter the Consumer name:Gggggg
Enter the Number of Units consumed:701

Consumed Name: Gggggg
Units Consumed: 701
Amount to be paid: 3834.45

vii) Enter the Consumer name:Jjjjjjj
Enter the Number of Units consumed:989

Consumed Name: Jjjjjjj
Units Consumed: 989
Amount to be paid: 6754.50

viii) Enter the Consumer name:Kkkkkk
Enter the Number of Units consumed:1154

Consumed Name: Kkkkkk
Units Consumed: 1154
Amount to be paid: 8648.70

iii) Write a program to implement simple calculator using switch case

ALGORITHM:

1. **Start**
2. Display a menu with the following choices:
    1. Addition
    2. Subtraction
    3. Multiplication
    4. Division
    5. Modulo Division

3. **Input** user's choice (choice)
4. **Input** two integers (a and b)
5. **Switch** on the value of choice:
    - **Case 1:**
      → Compute a + b
      → Display result
    - **Case 2:**
      → Compute a - b
      → Display result
    - **Case 3:**
      → Compute a * b
      → Display result
    - **Case 4:**
      → If b ≠ 0
      → Compute a / b as a float
      → Display result
      → Else
      → Display "Division by zero" error
    - **Case 5:**
      → If b ≠ 0
      → Compute a % b
      → Display result
      → Else
      → Display "Division by zero" error
    - **Default:**
      → Display "Invalid choice"
6. **End**

PROGRAM:

```
#include <stdio.h>
void main()
{
        int choice, a, b;
        printf("Menu:\n");
        printf("1. Addition\n2. Subtraction\n3. Multiplication\n4. Division\n5. Modulo Division\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        printf("Enter two numbers: ");
        scanf("%d %d", &a, &b);
        switch (choice)
        {
                case 1:
                        printf("Sum: %d\n", a + b);
                        break;
```

```
            case 2:
                printf("Difference: %d\n", a - b);
                break;
            case 3:
                printf("Product: %d\n", a * b);
                break;
            case 4:
                if (b != 0)
                    printf("Quotient: %.2f\n", (float)a / b);
                else
                    printf("Error: Division by zero.\n");
                break;
            case 5:
                if (b != 0)
                    printf("Remainder: %d\n", a % b);
                else
                    printf("Error: Division by zero.\n");
                break;
            default:
                printf("Invalid choice\n");
        }
}
```

OUTPUT:

```
Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 1
Enter two numbers: 56 74
Sum: 130

Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 2
Enter two numbers: 45 85
Difference: -40
```

Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 3
Enter two numbers: 15 15
Product: 225

Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 4
Enter two numbers: 35 75
Quotient: 0.466667

Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 5
Enter two numbers: 45 7
Remainder: 3

Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 4
Enter two numbers: 39 0
ERROR!

Error: Division by zero.

Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo Division

Enter your choice: 6
Enter two numbers: 23 67
Invalid choice

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, C Programs using decision-making constructs have been 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.

Write a program to find the sum of the digits, reverse, number of digits of a given number and also check whether it is a palindrome or not using while statement.

ALGORITHM:

1. **Start**
2. **Input** a number num
3. **Initialize**:
    - sum = 0 (for digit sum)
    - reverse = 0 (to store reversed number)
    - count = 0 (to count digits)
    - num1 = num (to keep original number for later comparison and display)
4. **Repeat** while num ≠ 0:
    - Get last digit: digit = num % 10
    - Add to sum: sum = sum + digit
    - Build reverse: reverse = reverse * 10 + digit
    - Remove last digit: num = num / 10
    - Increment digit count: count++
5. **Display**:
    - Sum of digits
    - Reversed number
    - Number of digits
6. **Check for palindrome**:
    - If num1 == reverse, print "Palindrome"
    - Else, print "Not a Palindrome"
7. **End**

PROGRAM:

```
#include<stdio.h>
void main()
{
        int num, digit, sum=0, reverse=0, num1, count=0;
        printf("Enter a number : ");
        scanf("%d", &num);
        num1 = num;
        while(num != 0)
        {
                digit = num % 10;
                sum += digit;
                reverse = reverse * 10 + digit;
                num = num / 10;
                count++;
        }
        printf("\nSum of the digits of %d: %d", num1, sum);
        printf("\nReverse of %d: %d", num1, reverse);
```

```
        printf("\nNumber of digits in %d: %d", num1, count);
        if (num1 == reverse)
                printf("\n%d is a Palindrome", num1);
        else
                printf("\n%d is not a Palindrome", num1);
}
```

OUTPUT:

i) Enter a number : 37658

Sum of the digits of 37658: 29
Reverse of 37658: 85673
Number of digits in 37658: 5
37658 is not a Palindrome

ii) Enter a number : 35453

Sum of the digits of 35453: 20
Reverse of 35453: 35453
Number of digits in 35453: 5
35453 is a Palindrome

Write a program to compute the sum, count, and average of a sequence of non-negative numbers entered by the user, stopping when a negative number is input using do-while statement.

ALGORITHM:

1. **Start**
2. **Initialize**:
   - sum ← 0
   - count ← 0
3. **Prompt** the user to:
   - "Enter a number (A negative number to stop):"
4. **Read input** into num
5. **If** num < 0:
   - Display: "No valid numbers are entered"
   - **Go to step 9**
6. **Repeat** the following steps **while num ≥ 0**:
   - Add num to sum
   - Increment count by 1
   - Prompt and read the next number:
     "Enter a number (A negative number to stop):"
   - Read into num
7. **After the loop ends**, calculate the average:
   - average ← sum / count

8. **Display Results**:
    - "Numbers entered: count"
    - "Sum of the entered numbers: sum"
    - "Average of the entered numbers: average (up to 3 decimal places)"
9. **End**

PROGRAM:

```
#include <stdio.h>
void main()
{
        int num, sum = 0, count = 0;
        float average;
        printf("Enter a number (A negative number to stop): ");
        scanf("%d", &num);
        if (num >= 0)
        {
                do
                {
                        sum += num;
                        count++;
                        printf("Enter a number (A negative number to stop): ");
                        scanf("%d", &num);
                } while (num >= 0);
                average = (float)sum / count;
                printf("\nNumbers entered: %d", count);
                printf("\nSum of the entered numbers: %d", sum);
                printf("\nAverage of the entered numbers: %.3f", average);
        }
        else
                printf("\nNo valid numbers are entered");
}
```

OUTPUT:

i)
```
Enter a number (A negative number to stop): 11
Enter a number (A negative number to stop): 22
Enter a number (A negative number to stop): 33
Enter a number (A negative number to stop): 44
Enter a number (A negative number to stop): 66
Enter a number (A negative number to stop): 77
Enter a number (A negative number to stop): 99
Enter a number (A negative number to stop): -5

Numbers entered: 7
Sum of the entered numbers: 352
Average of the entered numbers: 50.286
```

ii) Enter a number (A negative number to stop): -5

No valid numbers are entered

Write a program to read n numbers and compute the count and sum of even numbers in the sequence of numbers entered using for statement.

ALGORITHM:

1. **Start**
2. **Initialize**:
   - count ← 0 (to count even numbers)
   - sum ← 0 (to store sum of even numbers)
3. **Prompt the user**: "Enter the number of integers:"
4. **Read** the value of n (total numbers to be entered)
5. **Repeat** the following loop from i = 0 to i < n:
   - Prompt: "Enter integer:"
   - Read input into num
   - **If** num % 2 == 0 (i.e., number is even):
     i) Increment count by 1
     ii) Add num to sum
6. **After the loop ends:**
   - Display: "Count of even numbers: count"
   - Display: "Sum of even numbers: sum"
7. **End**

PROGRAM:

```
#include <stdio.h>
void main()
{
        int i, n, num, count = 0, sum = 0;
        printf("Enter the number of integers: ");
        scanf("%d", &n);
        printf("Enter the integers:\n");
        for (i = 0; i < n; i++)
        {
                scanf("%d", &num);
                if (num % 2 == 0)
                {
                        count++;
                        sum += num;
                }
        }
        printf("Count of even numbers: %d\n", count);
```

```
    printf("Sum of even numbers: %d\n", sum);
}
```

OUTPUT:

i) Enter the number of integers: 10
Enter the integers:
12
23
34
45
56
67
78
89
92
94
Count of even numbers: 6
Sum of even numbers: 366

ii) Enter the number of integers: 5
Enter the integers:
11
3
55
33
77
Count of even numbers: 0
Sum of even numbers: 0

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, C Programs using while, do while and for statements have been executed and the output was obtained.

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

AIM

To understand and implement programs using one dimensional 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).

Write a C program to calculate the sum and average of elements in a 1D array.

ALGORITHM:

1. **Start**
2. **Declare variables:**
   - n: an integer to store the number of elements in the array.
   - i: a loop counter.
   - sum: an integer initialized to 0, used to accumulate the sum of array elements.
   - arr[20]: an array to store up to 20 integers.
   - average: a float to store the calculated average of the array elements.
3. **Input the number of elements (n):**
   - Prompt the user to enter the number of elements (n).
   - Read the integer value of n from the user.
4. **Input the elements of the array:**
   - Prompt the user to enter n integer elements.
   - For each element (from 0 to n-1), read the integer and store it in the array arr[i].
5. **Calculate the sum of the array elements:**
   - Initialize sum to 0.
   - Loop through the array from index 0 to n-1:
     - Add the value of arr[i] to sum.
6. **Calculate the average:**
   - Compute the average using the formula:
     average = sum / n
7. **Output the sum and average:**
   - Print the sum of the array elements.
   - Print the average of the array elements, formatted to 2 decimal places.
8. **End**

PROGRAM

```
#include <stdio.h>
void main()
{
        int n, i, sum = 0, arr[20];
        float average;
        printf("Enter the number of elements: ");
        scanf("%d", &n);
        printf("Enter the elements:\n");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &arr[i]);
        }
        for(i = 0; i < n; i++)
        {
                sum += arr[i];
        }
        average = (float)sum / n;
        printf("Sum of the array elements: %d\n", sum);
        printf("Average of the array elements: %.2f", average);
}
```

OUTPUT

```
Enter the number of elements: 10
Enter the elements:
12 14 16 18 20 21 23 25 27 29
Sum of the array elements: 205
Average of the array elements: 20.50
```

Write a program to find the largest and the smallest elements and also their position in an array.

ALGORITHM:

1. **Start**
2. **Declare variables:**
    - n: Number of elements in the array
    - i: Loop counter
    - arr[20]: Array to hold up to 20 integers
    - max, min: Variables to hold the largest and smallest values
    - maxpos, minpos: To store the positions of the largest and smallest elements
3. **Input number of elements (n)**
    - Prompt user to enter number of elements
    - Read n
4. **Check if n is within valid range (1 to 20)**
    - If n is less than or equal to 0 or greater than 20
        - Print error message

        - Exit program
5. **Input array elements**
    - Loop from i = 0 to i < n:
        - Read arr[i]
6. **Initialize max, min, maxpos, minpos**
    - Set max = arr[0]
    - Set min = arr[0]
    - Set maxpos = 0
    - Set minpos = 0
7. **Find largest and smallest elements**
    - Loop from i = 1 to i < n
        - If arr[i] > max:
            - Set max = arr[i]
            - Set maxpos = i
        - If arr[i] < min:
            - Set min = arr[i]
            - Set minpos = i
8. **Print results**
    - Print value of max and its position (maxpos)
    - Print value of min and its position (minpos)
9. **End**

PROGRAM:

```
#include <stdio.h>
void main()
{
        int n, i, max, min, maxpos, minpos, arr[20];
        printf("Enter the number of elements: ");
        scanf("%d", &n);
        printf("Enter the elements:\n");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &arr[i]);
        }
        max = arr[0];
        maxpos = 0;
        min = arr[0];
        minpos = 0;
        for(i = 0; i < n; i++)
        {
                if (arr[i] > max)
                {
                        max = arr[i];
                        maxpos = i;
```

```
            }
            if (arr[i] < min)
            {
                min = arr[i];
                minpos = i;
            }
        }
        printf("\nLargest element in the array: %d", max);
        printf("\nPosition of the largest element in the array: %d\n", maxpos);
        printf("\nSmallest element in the array: %d", min);
        printf("\nPosition of the smallest element in the array: %d\n", minpos);
}
```

OUTPUT:

i) Enter the number of elements: 10
Enter the elements:
56 78 90 23 45 12 99 116 34 9

Largest element in the array: 116
Position of the largest element in the array: 7

Smallest element in the array: 9
Position of the smallest element in the array: 9

ii) Enter the number of elements: 10
Enter the elements:
11 89 56 23 45 34 98 76 87 110

Largest element in the array: 110
Position of the largest element in the array: 9

Smallest element in the array: 11
Position of the smallest element in the array: 0

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, Programs using One Dimensional Array have been executed and the output was obtained.

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

AIM

To understand and implement programs using two dimensional 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.

Write a program to add two matrices and print the result in matrix format.

ALGORITHM

1. **Start**
2. **Declare variables**:
    - r1, c1: rows and columns of Matrix 1
    - r2, c2: rows and columns of Matrix 2
    - mat1[7][7], mat2[7][7]: 2D arrays for input matrices
    - sum[7][7]: 2D array to store the result
    - i, j: loop counters
3. **Input dimensions of Matrix 1**
    - Prompt and read r1 and c1
4. **Input dimensions of Matrix 2**
    - Prompt and read r2 and c2
5. **Check if addition is possible**
    - If r1 == r2 **and** c1 == c2, proceed
    - Else:
        - Print "Order of the two matrices are not equal and cannot be added"
        - Exit program
6. **Input elements of Matrix 1**
   for i = 0 to r1 - 1:
        for j = 0 to c1 - 1:
            - Read mat1[i][j]
7. **Input elements of Matrix 2**
   for i = 0 to r2 - 1:
        for j = 0 to c2 - 1:
            - Read mat2[i][j]
8. **Compute the sum of matrices**
   for i = 0 to r1 - 1:

for j = 0 to c1 - 1:

- sum[i][j] = mat1[i][j] + mat2[i][j]

9. **Display the sum matrix**

   for i = 0 to r1 - 1:

   for j = 0 to c1 - 1:

   - Print sum[i][j] followed by a tab (\t)

   Print newline after each row

10. **End**

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
	int r1, c1, r2, c2, mat1[7][7], mat2[7][7], i, j, sum[7][7];
	printf("Enter the number of rows in Matrix 1: ");
	scanf("%d", &r1);
	printf("Enter the number of columns in Matrix 1: ");
	scanf("%d", &c1);
	printf("Enter the number of rows in Matrix 2: ");
	scanf("%d", &r2);
	printf("Enter the number of columns in Matrix 2: ");
	scanf("%d", &c2);
	if ((r1 == r2) && (c1 == c2))
	{
		printf("Enter the elements of Matrix 1:\n");
		for(i = 0; i < r1; i++)
		{
			for(j = 0; j < c1; j++)
			{
				scanf("%d", &mat1[i][j]);
			}
		}
		printf("Enter the elements of Matrix 2:\n");
		for(i = 0; i < r2; i++)
			for(j = 0; j < c2; j++)
				scanf("%d", &mat2[i][j]);
		for(i = 0; i < r1; i++)
		{
			for(j = 0; j < c1; j++)
				sum[i][j] = mat1[i][j] + mat2[i][j];
		}
```

```
            printf("\nSUM OF THE MATRICES:\n");
            for(i = 0; i < r1; i++)
            {
                  for(j = 0; j < c1; j++)
                  {
                        printf("%d\t", sum[i][j]);
                  }
                  printf("\n");
            }
      }
      else
            printf("\nOrder of the two matrices are not equal and cannot be added");
}
```

OUTPUT

i)
```
Enter the number of rows in Matrix 1: 4
Enter the number of columns in Matrix 1: 3
Enter the number of rows in Matrix 2: 4
Enter the number of columns in Matrix 2: 3
Enter the elements of Matrix 1:
2 4 6 8 10 12 14 16 18 20 22 24
Enter the elements of Matrix 2:
1 3 5 7 9 11 13 15 17 19 21 23

SUM OF THE MATRICES:
3       7       11
15      19      23
27      31      35
39      43      47
```

ii)
```
Enter the number of rows in Matrix 1: 3
Enter the number of columns in Matrix 1: 4
Enter the number of rows in Matrix 2: 3
Enter the number of columns in Matrix 2: 3

Order of the two matrices are not equal and cannot be added
```

Write a program to multiply two matrices and print the result in matrix format.

ALGORITHM:

1. **Start**
2. **Declare variables:**
   - r1, c1: Rows and columns of Matrix 1
   - r2, c2: Rows and columns of Matrix 2
   - mat1[7][7], mat2[7][7]: Input matrices
   - product[7][7]: Result matrix
   - i, j, k: Loop counters
3. **Input dimensions of Matrix 1**
   - Prompt and read r1, c1
4. **Input dimensions of Matrix 2**
   - Prompt and read r2, c2
5. **Check matrix multiplication rule**
   - If c1 != r2, display:
     - "No. of Columns in Matrix 1 is not equal to No. of Rows in Matrix 2 and cannot be multiplied"
     - **Exit**
6. **Input elements of Matrix 1**

   for i = 0 to r1 - 1:

   for j = 0 to c1 - 1:

   Read mat1[i][j]
7. **Input elements of Matrix 2**

   for i = 0 to r2 - 1:

   for j = 0 to c2 - 1:

   Read mat2[i][j]
8. **Initialize and compute product matrix**

   for i = 0 to r1 - 1:

   for j = 0 to c2 - 1:

   i. Set product[i][j] = 0

   ii. for k = 0 to c1 - 1:

   product[i][j] += mat1[i][k] * mat2[k][j]
9. **Display Matrix 1**
10. **Display Matrix 2**
11. **Display Product Matrix**
12. **End**

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
	int r1, c1, r2, c2, mat1[7][7], mat2[7][7], i, j;
	int k, product[7][7];
	printf("Enter the number of rows in Matrix 1: ");
	scanf("%d", &r1);
	printf("Enter the number of columns in Matrix 1: ");
	scanf("%d", &c1);
	printf("Enter the number of rows in Matrix 2: ");
	scanf("%d", &r2);
	printf("Enter the number of columns in Matrix 2: ");
	scanf("%d", &c2);
	if (c1 == r2)
	{
		printf("Enter the elements of Matrix 1:\n");
		for(i = 0; i < r1; i++)
		{
			for(j = 0; j < c1; j++)
			{
				scanf("%d", &mat1[i][j]);
			}
		}
		printf("Enter the elements of Matrix 2:\n");
		for(i = 0; i < r2; i++)
			for(j = 0; j < c2; j++)
				scanf("%d", &mat2[i][j]);
		for(i = 0; i < r1; i++)
		{
			for(j = 0; j < c2; j++)
			{
				product[i][j] = 0;
				for(k = 0; k < c1; k++)
					product[i][j] = product[i][j] + mat1[i][k] * mat2[k][j];
			}
		}
		printf("\nMATRIX 1:\n");
		for(i = 0; i < r1; i++)
		{
			for(j = 0; j < c1; j++)
			{
				printf("%d\t", mat1[i][j]);
```

```
                }
                printf("\n");
        }
        printf("\nMATRIX 2:\n");
        for(i = 0; i < r2; i++)
        {
                for(j = 0; j < c2; j++)
                {
                        printf("%d\t", mat2[i][j]);
                }
                printf("\n");
        }
        printf("\nPRODUCT OF THE MATRICES:\n");
        for(i = 0; i < r1; i++)
        {
                for(j = 0; j < c2; j++)
                {
                        printf("%d\t", product[i][j]);
                }
                printf("\n");
        }
    }
    else
        printf("\nNo. of Columns in Matrix 1 is not equal to No. of Rows in Matrix 2 and cannot be
        multiplied");
}
```

OUTPUT:

i) Enter the number of rows in Matrix 1: 3
Enter the number of columns in Matrix 1: 4
Enter the number of rows in Matrix 2: 3
Enter the number of columns in Matrix 2: 4

No. of Columns in Matrix 1 is not equal to No. of Rows in Matrix 2 and cannot be multiplied

ii) Enter the number of rows in Matrix 1: 3
Enter the number of columns in Matrix 1: 4
Enter the number of rows in Matrix 2: 4
Enter the number of columns in Matrix 2: 4
Enter the elements of Matrix 1:
2 4 6 8 1 3 5 7 1 2 3 4
Enter the elements of Matrix 2:
1 3 5 7 2 2 2 2 2 4 6 8 1 2 3 4

MATRIX 1:

2	4	6	8
1	3	5	7
1	2	3	4

MATRIX 2:

1	3	5	7
2	2	2	2
2	4	6	8
1	2	3	4

PRODUCT OF THE MATRICES:

30	54	78	102
24	43	62	81
15	27	39	51

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 programs using Two-Dimensional Array have been executed and the output was obtained.

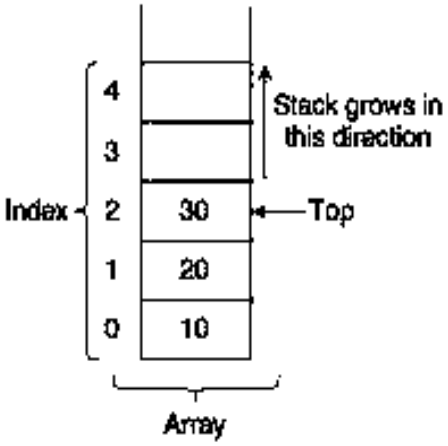
Ex.No:4a **ARRAY IMPLEMENTATION OF STACK ADT**

AIM

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

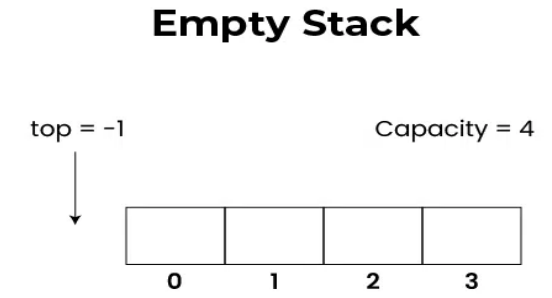
PRELAB DISCUSSION

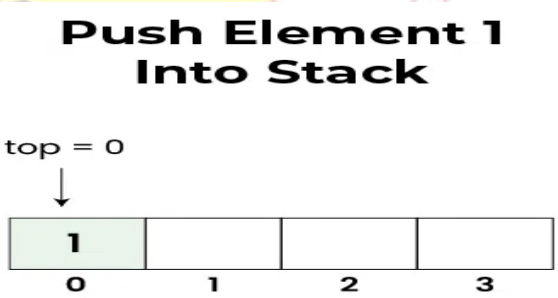
Stack Data Structure: A stack is a Last-In-First-Out (LIFO) data structure where elements are added and removed from one end called the top.

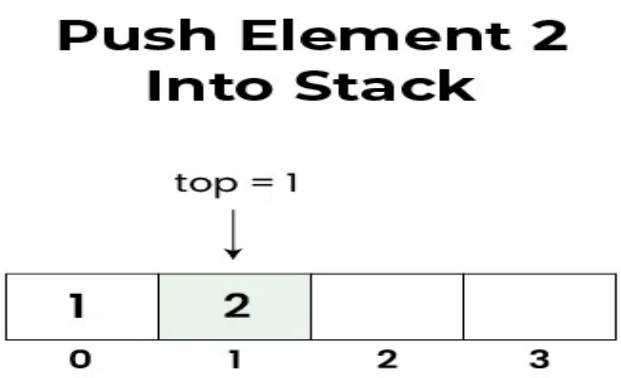


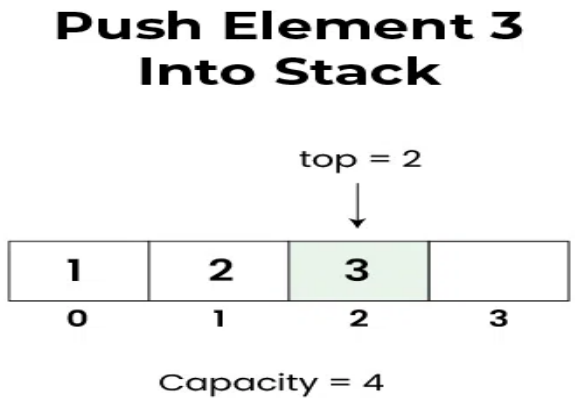
Operations:

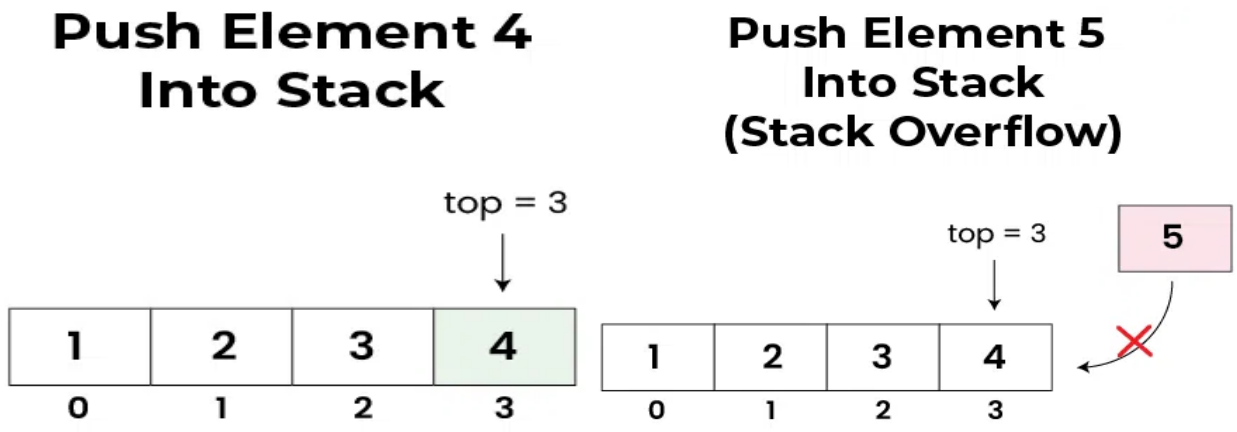
Push: Adds an element to the top of the stack.



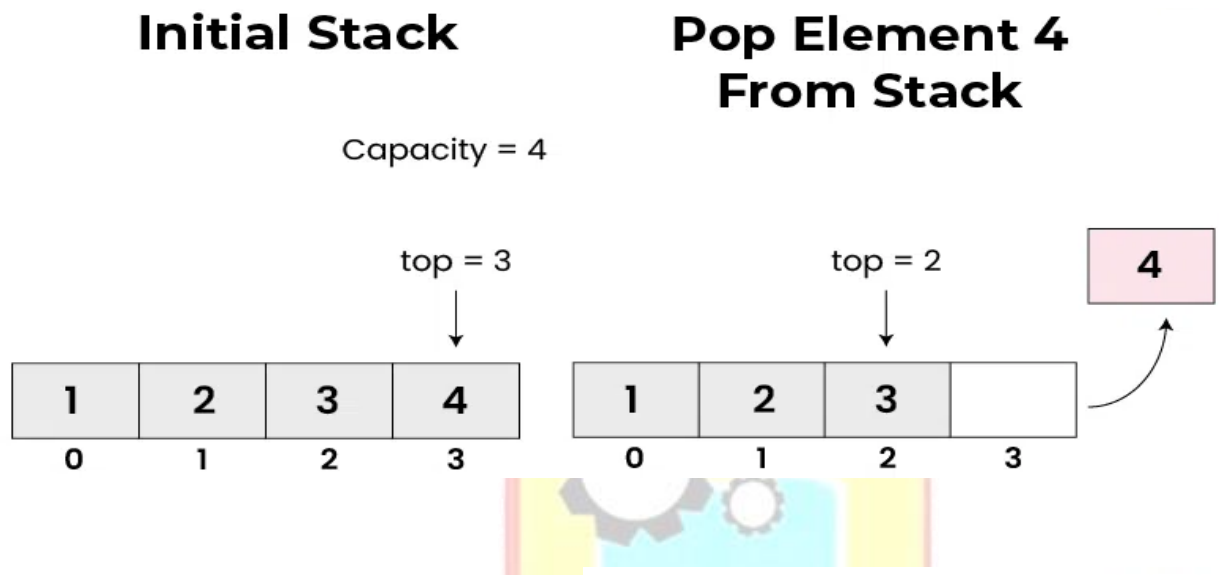


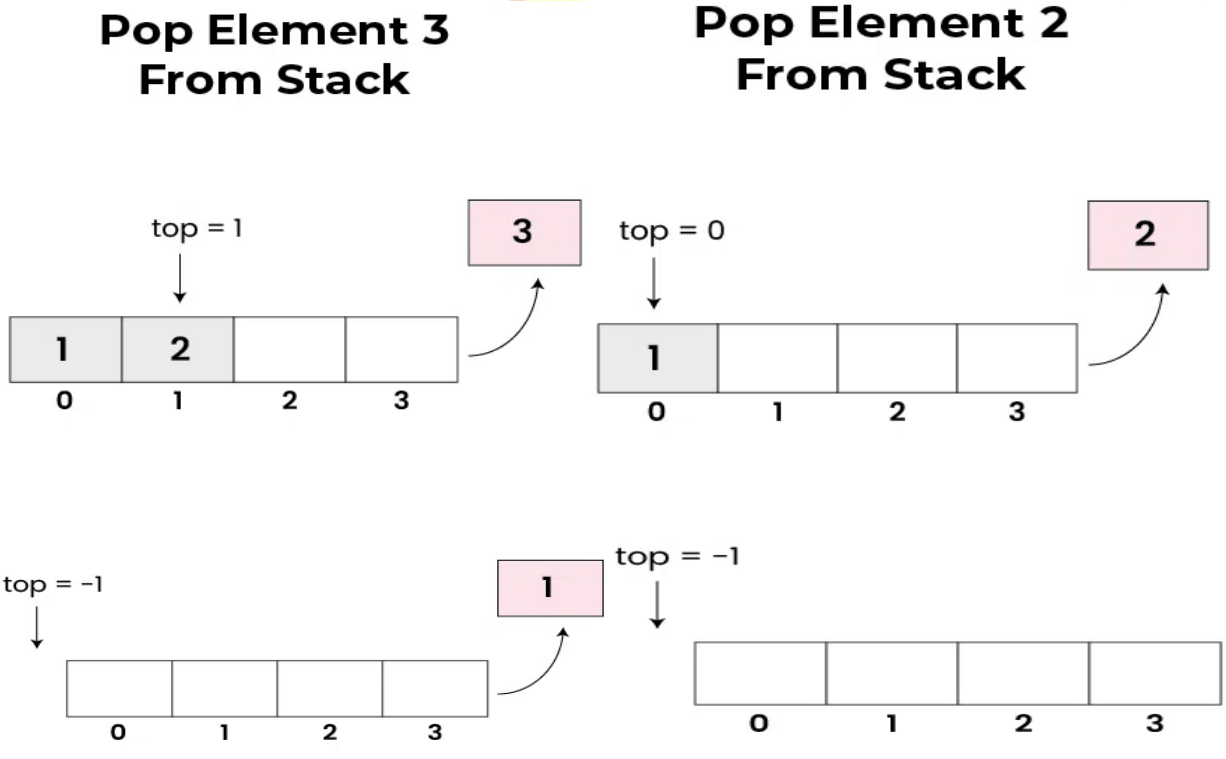






Pop: Removes and returns the element from the top of the stack.





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 stack and top pointer
   - Define an array stack[SIZE] to hold stack elements.
   - Set top = -1 to indicate the stack is empty.
3. Display menu repeatedly using a loop
   - Menu options:
     1. Push
     2. Pop
     3. Exit
4. Input user choice
   - Read the user's menu choice into variable ch.
5. Handle operations using switch-case:

   Case 1: Push Operation
   - If top == SIZE - 1
     - Print "Stack overflows."
     - Exit push operation
   - Else
     - Prompt user for item
     - Read the item
     - Increment top by 1
     - Set stack[top] = item
     - Call display() to print stack contents

   Case 2: Pop Operation
   - If top == -1
     - Print "The stack is empty."
     - Exit pop operation
   - Else
     - Print the element at stack[top] as the deleted item
     - Decrement top by 1
     - Call display() to print stack contents

   Case 3: Exit
   - Terminate the program using exit(0)
6. Loop continues until user exits

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 8
int item;
int stack[10], top = -1;
void display()
{
        int i;
        if (top == -1)
        {
                printf("\nThe stack is empty.");
                return;
        }
        printf("The content of the stack is:\n");
        for (i = 0; i <= top; i++)
        {
                printf("%d\t", stack[i]);
        }
}
void push()
{
        if (top == SIZE - 1)
        {
                printf("\nStack overflows.\n");
                return;
        }
        printf("\nEnter an item: ");
        scanf("%d", &item);
        top++;
        stack[top] = item;
        display();
}
void pop()
{
        if (top == -1)
        {
                printf("\nThe stack is empty.\n");
                return;
        }
        printf("\nThe deleted item is: %d", stack[top]);
        top--;
        display();
}
```

```
void main()
{
        int ch;
        printf("\nARRAY IMPLEMENTATION OF STACK ADT\n");
        do
        {
                printf("1. Push\n");
                printf("2. Pop\n");
                printf("3. Exit\n");
                printf("Enter your choice: ");
                scanf("%d", &ch);
                switch (ch)
                {
                        case 1:
                                push();
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                exit(0);
                }
        }while (1);
        getchar();
}
```

OUTPUT:

```
        ARRAY IMPLEMENTATION OF STACK ADT
        1. Push
        2. Pop
        3. Exit
        Enter your choice: 1
        Enter an item: 11
        The content of the stack is:
        11

        1. Push
        2. Pop
        3. Exit
        Enter your choice: 1
        Enter an item: 22
        The content of the stack is:
        11      22
```

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter an item: 33
The content of the stack is:
11 22 33

1. Push
2. Pop
3. Exit
Enter your choice: 2
The deleted item is: 33
The content of the stack is:
11 22

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter an item: 44
The content of the stack is:
11 22 44

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter an item: 55
The content of the stack is:
11 22 44 55

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter an item: 66
The content of the stack is:
11 22 44 55 66

1. Push
2. Pop
3. Exit
Enter your choice: 1

Enter an item: 77
The content of the stack is:
11 22 44 55 66 77

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter an item: 88
The content of the stack is:
11 22 44 55 66 77 88

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter an item: 99
The content of the stack is:
11 22 44 55 66 77 88 99

1. Push
2. Pop
3. Exit
Enter your choice: 1

Stack overflows.
1. Push
2. Pop
3. Exit
Enter your choice: 2
The deleted item is: 99
The content of the stack is:
11 22 44 55 66 77 88

1. Push
2. Pop
3. Exit
Enter your choice: 2
The deleted item is: 88
The content of the stack is:
11 22 44 55 66 77

1. Push
2. Pop

3. Exit
Enter your choice: 2
The deleted item is: 77
The content of the stack is:
11 22 44 55 66

1. Push
2. Pop
3. Exit
Enter your choice: 2
The deleted item is: 66
The content of the stack is:
11 22 44 55

1. Push
2. Pop
3. Exit
Enter your choice: 2
The deleted item is: 55
The content of the stack is:
11 22 44

1. Push
2. Pop
3. Exit
Enter your choice:
2
The deleted item is: 44
The content of the stack is:
11 22

1. Push
2. Pop
3. Exit
Enter your choice: 2
The deleted item is: 22
The content of the stack is:
11

1. Push
2. Pop
3. Exit
Enter your choice: 2

The deleted item is: 11
The stack is empty.

1. Push
2. Pop
3. Exit
Enter your choice: 3

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.

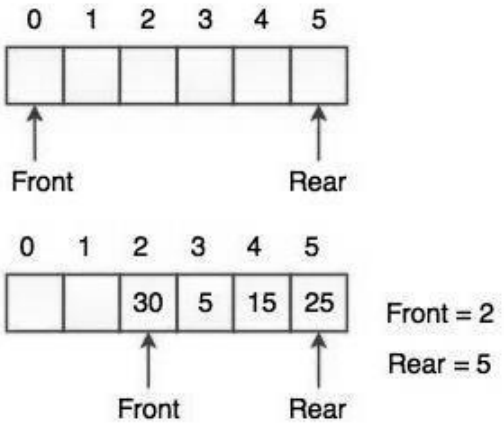
Ex.No:4b **ARRAY IMPLEMENTATION OF QUEUE ADT**

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 from 0). 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
    - Define queue[SIZE] to hold elements.
    - Set front = 0 and rear = -1.
3. Loop for Menu Options
    - Repeat:
        - Display menu:
            - Insert
            - Delete
            - Exit
        - Input user choice into option
    - Case 1: Enqueue (Insert)
        - Check if queue is full using isFull():
            - If **full**, print **QUEUE OVERFLOWS** and exit operation.
        - Else:
            - Increment rear
            - Assign queue[rear] = element
            - Print the updated queue using display()
    - Case 2: Dequeue (Delete)
        - Check if queue is empty using isEmpty():
            - If **empty**, print **QUEUE UNDERFLOWS** and exit operation.
        - Else:
            - Assign element = queue[front]
            - Increment front
            - Return and print the deleted element
            - Call display() to show the updated queue
    - Case 3: Exit
        - Terminate the program using exit(0)

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 8
int queue[SIZE], front = 0, rear = -1;
int isFull();
int isEmpty();
```

```
void enqueue(int element)
{
        if (isFull())
        printf("\nQUEUE OVERFLOWS\n");
        else
        {
                rear++;
                queue[rear] = element;
        }
        printf("\n");
}
int dequeue()
{
        int element;
        if (isEmpty())
        {
                printf("QUEUE UNDERFLOWS\n");
                return(-1);
        }
        else
        {
                element = queue[front];
                front = front + 1;
                return (element);
        }
}
int isFull()
{
        if (rear == SIZE - 1)
                return 1;
        return 0;
}
int isEmpty()
{
        if (front > rear)
                return 1;
        return 0;
}
void display()
{
        int i;
        if (isEmpty())
                printf("EMPTY QUEUE\n");
```

```
        else
        {
                printf("Front ->   ");
                for (i = front; i <= rear; i++)
                        printf("%d\t", queue[i]);
                printf("<- Rear\n");
        }
}
void main()
{
        int option, element;
        printf("ARRAY IMPLEMENTATION OF QUEUE ADT\n");
        do
        {
                printf("1. Insert\n");
                printf("2. Delete\n");
                printf("3. 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);
                                display();
                                break;
                        case 2:
                                element = dequeue();
                                if (element != -1)
                                {
                                        printf("Deleted Element is %d\n\n", element);
                                        display();
                                }
                                break;
                        case 3:
                                exit(0);
                                break;
                        default:
                                printf("Invalid Option. Try Again.");
                                break;
                }
        } while (1);
```

```
    getchar();
}
```

OUTPUT:

```
    ARRAY IMPLEMENTATION OF QUEUE ADT
    1. Insert
    2. Delete
    3. Exit
    Enter your choice: 1
    Enter the element to be inserted: 12
    Front ->   12   <- Rear

    1. Insert
    2. Delete
    3. Exit
    Enter your choice: 1
    Enter the element to be inserted: 23
    Front ->   12     23    <- Rear

    1. Insert
    2. Delete
    3. Exit
    Enter your choice: 1
    Enter the element to be inserted: 34
    Front ->   12       23       34     <- Rear

    1. Insert
    2. Delete
    3. Exit
    Enter your choice: 2
    Deleted Element is 12
    Front ->   23       34       <- Rear

    1. Insert
    2. Delete
    3. Exit
    Enter your choice: 2
    Deleted Element is 23
    Front ->   34   <- Rear

    1. Insert
    2. Delete
    3. Exit
    Enter your choice: 2
```

Deleted Element is 34
EMPTY QUEUE

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 45
Front -> 45 <- Rear

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 56
Front -> 45 56 <- Rear

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 67
Front -> 45 56 67 <- Rear

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 78
Front -> 45 56 67 78 <- Rear

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 89
Front -> 45 56 67 78 89 <- Rear

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 98

QUEUE OVERFLOWS

Front -> 45 56 67 78 89 <- Rear
1. Insert
2. Delete
3. Exit
Enter your choice: 3

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 queue called 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

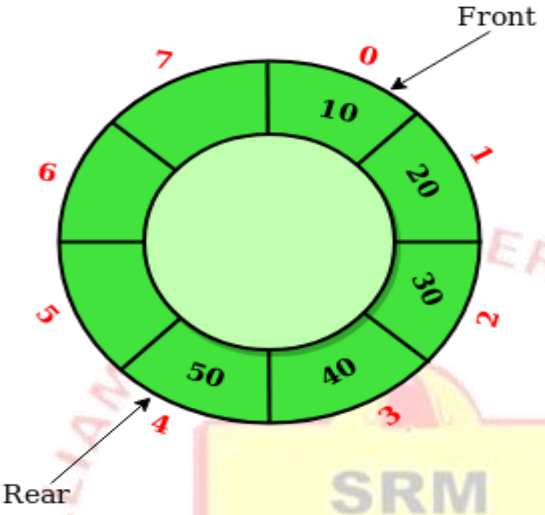
AIM

To write a C program to implement Circular 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'.



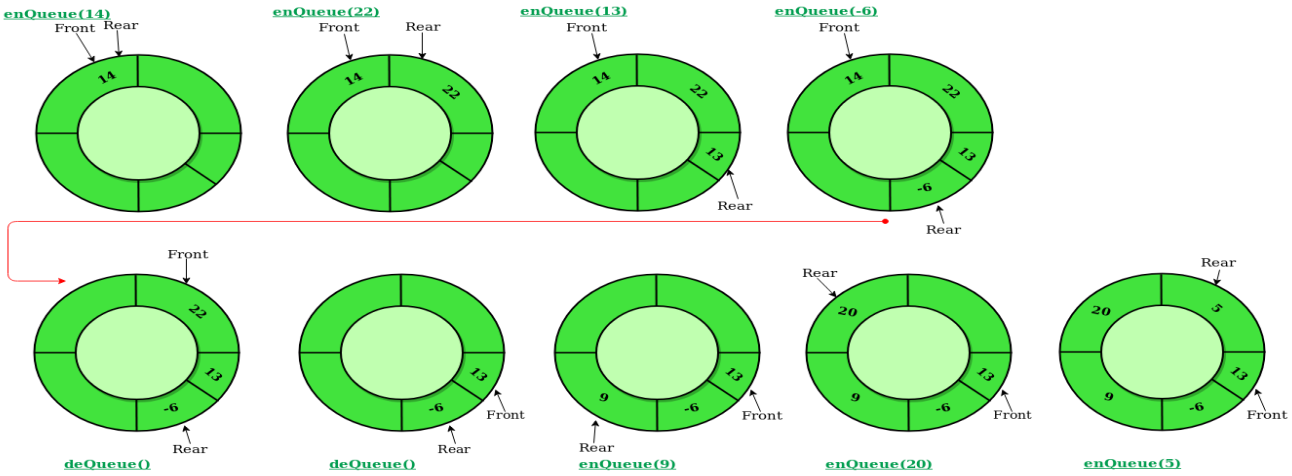
Operations on Circular Queue:

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
   - Define queue[SIZE] to hold elements.
   - Set front = -1 and rear = -1.
3. Loop for Menu Options
   - Repeat:
     - Display menu:
       - Insert
       - Delete
       - Exit
     - Input user choice into option
   - Case 1: Enqueue (Insert)
     - Check if queue is full using isFull():
       - If **full**, print **QUEUE OVERFLOWS** and exit operation.
     - Else check if queue is empty using isempty():
       - Front = rear = 0
     - Else
       - Increment rear = (rear + 1) % SIZE
     - Assign queue[rear] = element
     - Print the updated queue using display()
   - Case 2: Dequeue (Delete)
     - Check if queue is empty using isEmpty():
       - If **empty**, print **QUEUE UNDERFLOWS** and exit operation.
     - Else check if queue has only one element by front == rear:
       - Assign element = queue[front]
       - Front = rear = -1
     - Else
       - Assign element = queue[front]
       - Increment front = (front + 1) % SIZE
       - Return and print the deleted element
       - Call display() to show the updated queue
   - Case 3: Exit
     - Terminate the program using exit(0)

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 8
int queue[SIZE];
int front = -1, rear = -1;
int isEmpty()
{
        return (front == -1 && rear == -1);
```

```
}
int isFull()
{
        return ((rear + 1) % SIZE == front);
}
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);
}
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;
}
void display()
{
        int i;
        if (isEmpty())
        {
```

```
                printf("EMPTY QUEUE\n");
                return;
        }
        printf("Queue elements:\t ");
        i = front;
        do
        {
                printf("%d\t", queue[i]);
                i = (i + 1) % SIZE;
        } while (i != (rear + 1) % SIZE);
        printf("\n");
}
void main()
{
        int option, element;
        printf("CIRCULAR QUEUE IMPLEMENTATION\n");
        do
        {
                printf("1. Insert\n");
                printf("2. Delete\n");
                printf("3. 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);
                                display();
                                break;
                        case 2:
                                element = dequeue();
                                if (element != -1)
                                        printf("Deleted Element is %d\n", element);
                                display();
                                break;
                        case 3:
                                exit(0);
                        default:
                                printf("Invalid Option. Try Again.\n");
                                break;
                }
```

```
    } while (1);
}
```

OUTPUT

```
CIRCULAR QUEUE IMPLEMENTATION
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 11
Enqueued element: 11
Queue elements:         11

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 22
Enqueued element: 22
Queue elements:         11  22

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 33
Enqueued element: 33
Queue elements:         11  22  33

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 44
Enqueued element: 44
Queue elements:         11  22  33  44

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted Element is 11
Queue elements:         22  33  44
```

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 55
Enqueued element: 55
Queue elements: 22 33 44 55

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted Element is 22
Queue elements: 33 44 55

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted Element is 33
Queue elements: 44 55

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted Element is 44
Queue elements: 55

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted Element is 55
EMPTY QUEUE

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 66
Enqueued element: 66
Queue elements: 66

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 77
Enqueued element: 77
Queue elements: 66 77

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 88
Enqueued element: 88
Queue elements: 66 77 88

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 99
Enqueued element: 99
Queue elements: 66 77 88 99

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 12
Enqueued element: 12
Queue elements: 66 77 88 99 12

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 23
Enqueued element: 23
Queue elements: 66 77 88 99 12 23

1. Insert
2. Delete
3. Exit
Enter your choice: 1

Enter the element to be inserted: 34
Enqueued element: 34
Queue elements: 66 77 88 99 12 23 34

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 45
Enqueued element: 45
Queue elements: 66 77 88 99 12 23 34 45

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the element to be inserted: 56
QUEUE OVERFLOW
Queue elements: 66 77 88 99 12 23 34 45

1. Insert
2. Delete
3. Exit
Enter your choice: 3

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 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.

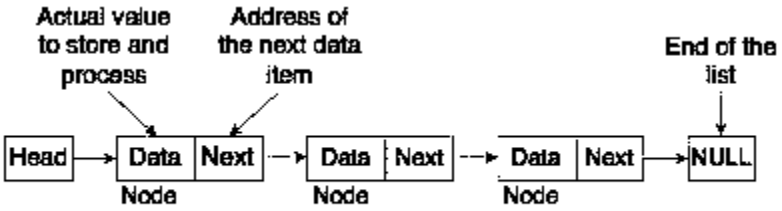
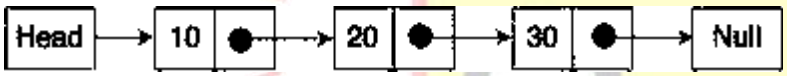


Fig. Linked List

Linked list contains a link element called first and each link carries a data item. Entry point into the 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 the list.

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 Program
   - Initialize head pointer to NULL.
2. Display Menu in a Loop
   - Repeat until user chooses to exit:
     1. Display options:
        - 1. Insert
        - 2. Delete
        - 3. Exit
     2. Read user choice into opt
     3. Perform the corresponding operation based on opt:
        - if opt == 1, Call insert() and then display()
        - if opt == 2, Call deletion() and then display()
        - if opt == 3, Exit program

Insert() Algorithm

A. If head == NULL (list is empty):
    - Prompt and read the first item.
    - Create a new node.
    - Assign it to head.
B. Else (list is not empty):
    - Prompt and read the item and position.
    - Create a new node.
    - If position is 1:
        - Set new node's next to current head.
        - Update head to new node.
    - Else:
        - Traverse to the node at position - 1:
            - If node does not exist (position invalid), print error and return.
        - Insert the new node:
            - Set new node's next to the next node.
            - Set previous node's next to the new node.

Deletion() Algorithm

A. If head == NULL:
    - Print "Linked list is empty" and return.
B. Prompt and read the item to delete.
C. Initialize ptr = head, prev = NULL.
D. Traverse the list:
    - While ptr != NULL and ptr->data != item:
        - Move prev = ptr and ptr = ptr->next
E. If ptr == NULL:
    - Print "Item not found" and return.
F. Else:
    - If prev == NULL (item is at head):
        - Set head = head->next
    - Else:
        - Set prev->next = ptr->next
    - Free the memory of ptr

Display() Algorithm

A. If head == NULL:
    - Print "Linked list is empty" and return.
B. Initialize ptr = head.
C. While ptr != NULL:
    - Print ptr->data
    - Move to ptr = ptr->next
D. Print newline.

3. End Program

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list
{
        int data;
        struct list *next;
} LIST;
LIST *p, *temp, *head = NULL, *prev, *ptr;
LIST* create(int element);
void insert();
void deletion();
void display();
int pos, opt, item;
void main()
{
        do
        {
                printf("1. Insert\n");
                printf("2. Delete\n");
                printf("3. Exit\n");
                printf("Enter your option: ");
                scanf("%d", &opt);
                switch (opt)
                {
                        case 1:
                                insert();
                                display();
                                break;
                        case 2:
                                deletion();
                                display();
                                break;
                        case 3:
                                exit(0);
                                break;
                }
                printf("\n");
        } while (1);
        getchar();
}
```

```
LIST* create(int element)
{
        LIST *newnode;
        newnode = (LIST *)malloc(sizeof(LIST));
        newnode->data = element;
        newnode->next = NULL;
        return newnode;
}
void insert()
{
        int j;
        if (head == NULL)
        {
                printf("Enter the first item to be inserted: ");
                scanf("%d", &item);
                p = create(item);
                head = p;
        }
        else
        {
                printf("Enter the item to be inserted: ");
                scanf("%d", &item);
                printf("Enter the position to insert: ");
                scanf("%d", &pos);
                p = create(item);
                temp = head;
                if (pos == 1)
                {
                        head = p;
                        head->next = temp;
                }
                else
                {
                        for (j = 1; j < (pos - 1); j++)
                                temp = temp->next;
                        if (temp == NULL)
                        {
                                printf("\nInvalid position");
                                return;
                        }
                        p->next = temp->next;
                        temp->next = p;
                }
        }}
```

```
void deletion()
{
        printf("Enter the item to be deleted: ");
        scanf("%d", &item);
        if (head == NULL)
        {
                printf("\nLINKED LIST IS EMPTY");
                return;
        }
        ptr = head;
        prev = NULL;
        while ((ptr != NULL) && (ptr->data != item))
        {
                prev = ptr;
                ptr = ptr->next;
        }
        if (ptr == NULL)
        {
                printf("\nITEM NOT FOUND");
                return;
        }
        if (prev == NULL)
                head = head->next;
        else
                prev->next = ptr->next;
        free(ptr);
}
void display()
{
        if (head == NULL)
        {
                printf("\nLINKED LIST IS EMPTY");
                return;
        }
        ptr = head;
        while (ptr != NULL)
        {
                printf("-> %d ", ptr->data);
                ptr = ptr->next;
        }
        printf("\n");
}
```

OUTPUT

1. Insert
2. Delete
3. Exit

Enter your option: 1
Enter the first item to be inserted: 11
-> 11

1. Insert
2. Delete
3. Exit

Enter your option: 1
Enter the item to be inserted: 22
Enter the position to insert: 3
Invalid position-> 11

1. Insert
2. Delete
3. Exit

Enter your option: 1
Enter the item to be inserted: 22
Enter the position to insert: 2
-> 11 -> 22

1. Insert
2. Delete
3. Exit

Enter your option: 1
Enter the item to be inserted: 44
Enter the position to insert: 3
-> 11 -> 22 -> 44

1. Insert
2. Delete
3. Exit

Enter your option: 1
Enter the item to be inserted: 55
Enter the position to insert: 3
-> 11 -> 22 -> 55 -> 44

1. Insert
2. Delete
3. Exit

Enter your option: 1

Enter the item to be inserted: 77
Enter the position to insert: 5
-> 11 -> 22 -> 55 -> 44 -> 77

1. Insert
2. Delete
3. Exit
Enter your option: 1
Enter the item to be inserted: 66
Enter the position to insert: 6
-> 11 -> 22 -> 55 -> 44 -> 77 -> 66

1. Insert
2. Delete
3. Exit
Enter your option: 2
Enter the item to be deleted: 55
-> 11 -> 22 -> 44 -> 77 -> 66

1. Insert
2. Delete
3. Exit
Enter your option: 2
Enter the item to be deleted: 11
-> 22 -> 44 -> 77 -> 66

1. Insert
2. Delete
3. Exit
Enter your option: 2
Enter the item to be deleted: 66
-> 22 -> 44 -> 77

1. Insert
2. Delete
3. Exit
Enter your option: 2
Enter the item to be deleted: 44
-> 22 -> 77

1. Insert
2. Delete
3. Exit
Enter your option: 2

Enter the item to be deleted: 22
-> 77

1. Insert
2. Delete
3. Exit
Enter your option: 2
Enter the item to be deleted: 77

LINKED LIST IS EMPTY
1. Insert
2. Delete
3. Exit
Enter your option: 3

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

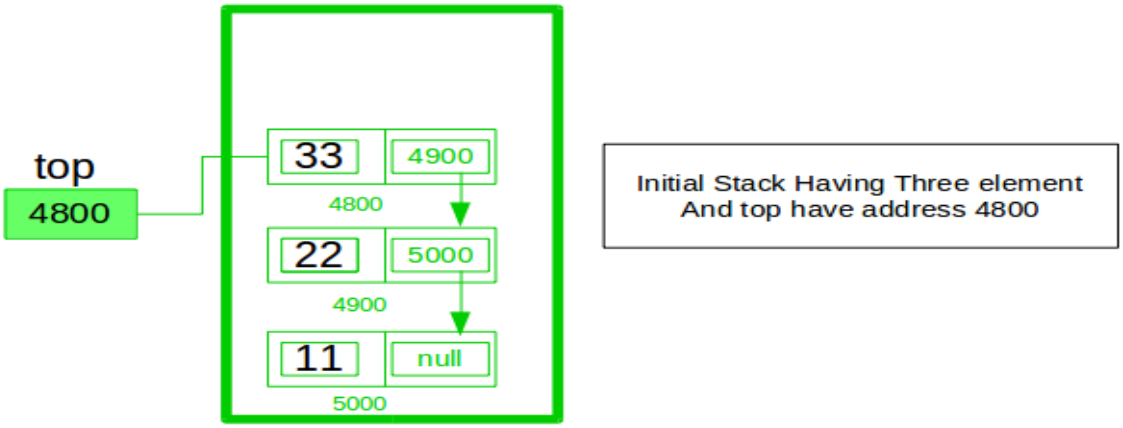
To write a C program to implement Stack operations such as push, pop and display using linked list.

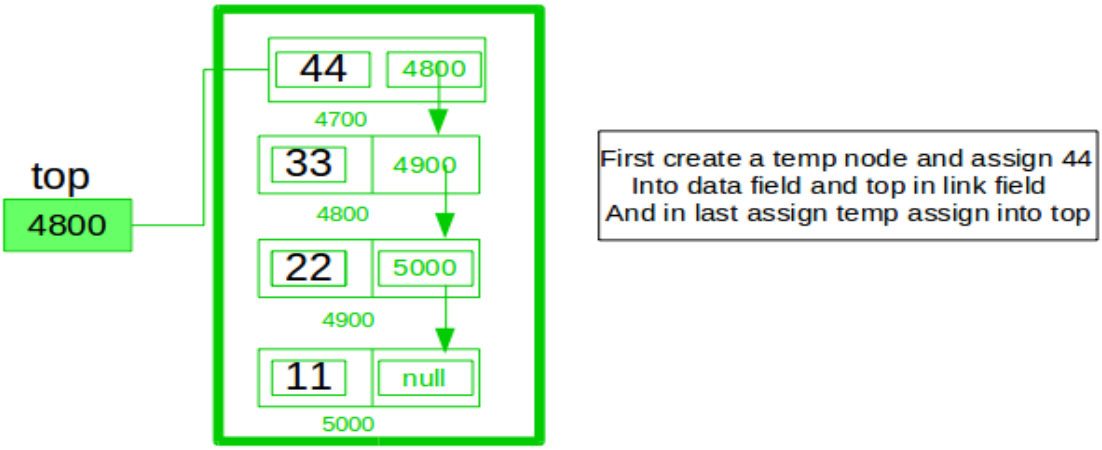
PRE LAB-DISCUSSION

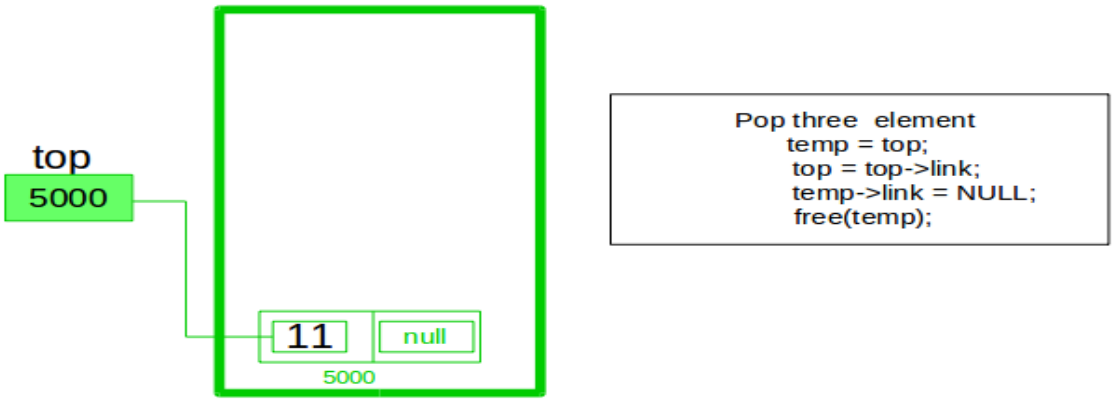
The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going 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:







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. Initialize
   - Set top = NULL (indicates an empty stack).
3. Display Menu in a Loop
   - Repeat until the user chooses to exit:
     - Display the options:
       1. Push
       2. Pop
       3. Exit
     - Read the user's choice.
4. Process User Choice

   If choice == 1 (Push):
   - Prompt user to enter a value.
   - Create a new node.
     - Allocate memory for the new node.
     - Set new_node->data = entered_value.
     - Set new_node->next = top.
     - Set top = new_node (update stack top).
   - Display the stack.

   If choice == 2 (Pop):
   - If top == NULL
     - Display "Stack Underflow".
   - Else
     - Store top in a temporary pointer.
     - Print the value of top->data (element being popped).
     - Update top = top->next.
     - Free the memory of the old top node.
   - Display the stack.

If choice == 3 (Exit):

- Terminate the program.

5. End

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
void pop();
void push(int value);
void display();
struct node
{
        int data;
        struct node *next;
};
struct node *top = NULL, *temp;
void main()
{
        int choice, element;
        printf("LINKED LIST IMPLEMENTATION OF STACK\n");
        while (1)
        {
                printf("1. Push\n");
                printf("2. Pop\n");
                printf("3. Exit\n");
                printf("Enter your choice: ");
                scanf("%d", &choice);
                switch (choice)
                {
                        case 1:
                                printf("Enter a new element: ");
                                scanf("%d", &element);
                                push(element);
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                exit(0);
                                break;
                }
        }
}
```

```
void display()
{
        temp = top;
        if (temp == NULL)
        {
                printf("\nStack is empty\n");
                return;
        }
        printf("The Contents of the Stack are...\n");
        while (temp != NULL)
        {
                printf(" -> %d  ", temp->data);
                temp = temp->next;
        }
        printf("\n");
}
void push(int data)
{
        temp = (struct node *)malloc(sizeof(struct node));
        temp->data = data;
        temp->next = top;
        top = temp;
        display();
}
void pop()
{
        if (top != NULL)
        {
                printf("The popped element is %d", top->data);
                top = top->next;
        }
        else
        {
                printf("Stack Underflow\n");
                return;
        }
        display();
}
```

OUTPUT:

LINKED LIST IMPLEMENTATION OF STACK
1. Push
2. Pop
3. Exit

Enter your choice: 2
Stack Underflow

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter a new element: 11
The Contents of the Stack are...
 -> 11

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter a new element: 22
The Contents of the Stack are...
 -> 22 -> 11

1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter a new element: 33
The Contents of the Stack are...
 -> 33 -> 22 -> 11

1. Push
2. Pop
3. Exit
Enter your choice: 2
The popped element is 33The Contents of the Stack are...
 -> 22 -> 11

1. Push
2. Pop
3. Exit
Enter your choice: 2
The popped element is 22The Contents of the Stack are...
 -> 11

1. Push
2. Pop
3. Exit

Enter your choice: 2
The popped element is 11
Stack is empty

1. Push
2. Pop
3. Exit
Enter your choice: 3

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.

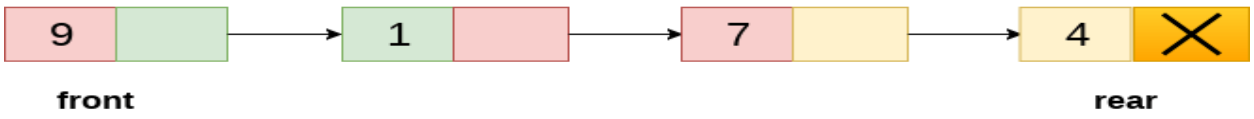
Ex.No:6b LINKED LIST IMPLEMENTATION OF LINEAR QUEUE ADT

AIM

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

PRE LAB-DISCUSSION

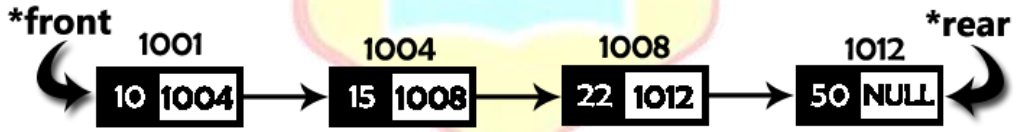
The major problem with the queue implemented using array is, It will work for only fixed number of data. That means the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can 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. Initialize
   - Set front = NULL, rear = NULL.
3. Menu Loop

   Repeat the following steps until the user selects "Exit":
   - **Display menu**:
     1. Insert
     2. Delete
     3. Exit
   - **Read user's choice**
4. Perform Operation Based on Choice

   If choice == 1 (Insert):
   - Prompt the user to enter a value.
   - Allocate memory for a new node.
   - Set newNode->data = value, newNode->next = NULL.
   - If front == NULL:
     - Set front = rear = newNode.
   - Else:
     - Set rear->next = newNode.
     - Set rear = newNode.
   - Print "Element inserted".
   - Display the queue.

   If choice == 2 (Delete):
   - If front == NULL:
     - Print "Queue is empty".
   - Else:
     - Store front in a temporary pointer.
     - Print temp->data as deleted.
     - Update front = front->next.
     - Free the memory of the deleted node.
   - Display the queue.

   If choice == 3 (Exit):
   - Terminate the program.

   Else:
   - Print "Invalid option. Try again".
5. End

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
        int data;
        struct Node *next;
};
struct Node *front = NULL, *rear = NULL, *ptr;
void insert(int element);
void delet();
void display();
void main()
{
        int choice, value;
        printf("LINKED LIST IMPLEMENTATION OF QUEUES\n");
        while (1)
        {
                printf("1. Insert\n");
                printf("2. Delete\n");
                printf("3. Exit\n");
                printf("Enter your choice: ");
                scanf("%d", &choice);
                switch (choice)
                {
                        case 1:
                                printf("Enter the value to be inserted: ");
                                scanf("%d", &value);
                                insert(value);
                                display();
                                break;
                        case 2:
                                delet();
                                display();
                                break;
                        case 3:
                                exit(0);
                        default:
                                printf("\nInvalid option. Try again.\n");
                }
        }
}
```

```
void insert(int element)
{
        struct Node *newNode;
        newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode -> data = element;
        newNode -> next = NULL;
        if (front == NULL)
                front = rear = newNode;
        else
        {
                rear -> next = newNode;
                rear = newNode;
        }
        printf("ELEMENT INSERTED");
}
void delet()
{
        if (front != NULL)
        {
                struct Node *temp = front;
                front = front->next;
                printf("Deleted element: %d", temp->data);
                free(temp);
        }
}
void display()
{
        if (front == NULL)
                printf("\nEMPTY QUEUE\n");
        else
        {
                printf("\nQUEUE ELEMENTS:  ");
                ptr = front;
                while (ptr != NULL)
                {
                        printf("%d -> ", ptr->data);
                        ptr = ptr->next;
                }
                printf("\n");
        }
}
```

OUTPUT:

```
LINKED LIST IMPLEMENTATION OF QUEUES
1. Insert
2. Delete
3. Exit
Enter your choice: 2
EMPTY QUEUE

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 11
ELEMENT INSERTED
QUEUE ELEMENTS:  11 ->

1. Insert
2. Delete
3. Exit
Enter your choice:
1
Enter the value to be inserted: 22
ELEMENT INSERTED
QUEUE ELEMENTS:  11 -> 22 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 33
ELEMENT INSERTED
QUEUE ELEMENTS:  11 -> 22 -> 33 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 44
ELEMENT INSERTED
QUEUE ELEMENTS:  11 -> 22 -> 33 -> 44 ->

1. Insert
2. Delete
3. Exit
```

Enter your choice: 2
Deleted element: 11
QUEUE ELEMENTS: 22 -> 33 -> 44 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted element: 22
QUEUE ELEMENTS: 33 -> 44 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 55
ELEMENT INSERTED
QUEUE ELEMENTS: 33 -> 44 -> 55 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted element: 33
QUEUE ELEMENTS: 44 -> 55 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted element: 44
QUEUE ELEMENTS: 55 ->

1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted element: 55
EMPTY QUEUE

1. Insert
2. Delete
3. Exit
Enter your choice: 3

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

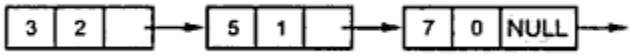
To write a 'C' program to represent a polynomial as a linked list and write functions for polynomial addition

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. Define the Node Structure
   - Each node contains:
     - coefficient: integer value
     - exponent: integer value
     - next: pointer to the next node
2. Create a Node
   - Function createNode(coefficient, exponent):
     - Allocate memory for a new node.
     - Set the coefficient and exponent.
     - Initialize next to NULL.
     - Return the new node.

3. Insert a Term into a Polynomial

- Function insertNode(pnum, terms):
  - For each term (loop terms times):
    - Input coefficient and exponent.
    - Create a node with the values.
    - Insert it into the appropriate polynomial (poly1 or poly2) in **descending order of exponent**:
      - If the list is empty, insert at the start.
      - Otherwise, find the correct position and insert.

4. Display the Polynomial

- Function displayPolynomial(poly):
  - Traverse the list.
  - For each node:
    - Print in the format coefficient x^exponent.
    - Add " + " if there are more terms.

5. Add Two Polynomials

- Function addPolynomials(poly1, poly2):
  - Initialize sum and sptr to NULL.
  - While both lists are not empty:
    - Compare the exponents of the current nodes:
      - If exp1 > exp2, copy term from poly1.
      - If exp1 < exp2, copy term from poly2.
      - If equal, add coefficients and create a new node.
    - Append the new node to the sum list.
  - If one list is not finished, append remaining terms.

6. Main Program Logic

- Prompt user for the number of terms in Polynomial 1.
- Call insertNode(1, terms) to build poly1.
- Display poly1.
- Repeat steps 1–3 for Polynomial 2.
- Call addPolynomials(poly1, poly2) to compute the sum.
- Display the resulting polynomial sum.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
        int coefficient;
        int exponent;
        struct Node* next;
};
struct Node *poly1=NULL, *poly2=NULL, *sum=NULL;
struct Node *ptr1, *ptr2, *sptr, *poly, *ptr;
```

```
struct Node* createNode(int coefficient, int exponent)
{
	struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
	newNode->coefficient = coefficient;
	newNode->exponent = exponent;
	newNode->next = NULL;
	return newNode;
}
void displayPolynomial(struct Node* poly)
{
	struct Node *temp;
	temp = poly;
	while (temp != NULL)
	{
		printf("%dx^%d", temp->coefficient, temp->exponent);
		temp = temp->next;
		if (temp)
			printf(" + ");
	}
	printf("\n");
}
void insertNode(int pnum, int terms)
{
	int i, coeff, expo;
	struct Node *temp, *prev;
	if (pnum == 1)
		poly = poly1;
	else
		poly = poly2;
	for(i=0; i<terms; i++)
	{
		printf("Enter the coefficient of term %d: ",i+1);
		scanf("%d", &coeff);
		printf("Enter the exponent of term %d: ",i+1);
		scanf("%d", &expo);
		ptr = createNode(coeff, expo);
		if (poly == NULL)
			poly = ptr;
		else
		{
			temp = poly;
			prev = NULL;
```

```
                while ((temp != NULL) && (temp->exponent > expo))
                {
                        prev = temp;
                        temp = temp->next;
                }
                if (prev == NULL)
                {
                        ptr->next = poly;
                        poly = ptr;
                }
                else
                {
                        ptr->next = temp;
                        prev->next = ptr;
                }
        }
    }
    if (pnum == 1)
        poly1 = poly;
    else
        poly2 = poly;
}
void addPolynomials(struct Node* poly1, struct Node* poly2)
{
    struct Node *temp;
    if (poly1 == NULL)
    {
        sum = poly2;
        return;
    }
    if (poly2 == NULL)
    {
        sum = poly1;
        return;
    }
    ptr1 = poly1;
    ptr2 = poly2;
    sptr = sum;
    while ((ptr1 != NULL) && (ptr2 != NULL))
    {
        if (ptr1->exponent > ptr2->exponent)
        {
            temp = createNode(ptr1->coefficient, ptr1->exponent);
```

```
            if (sptr == NULL)
            {
                    printf("\nTEMP: %d\tSUM: %d", temp, sum);
                    sptr = temp;
                    sum = temp;
                    printf("\nSUM: %d\tSPTR: %d", sum, sptr);
            }
            else
            {
                    sptr->next = temp;
                    sptr = sptr->next;
            }
            ptr1 = ptr1->next;
    }
    else if (ptr1->exponent < ptr2->exponent)
    {
            temp = createNode(ptr2->coefficient, ptr2->exponent);
            if (sptr == NULL)
            {
                    sptr = temp;
                    sum = sptr;
            }
            else
            {
                    sptr->next = temp;
                    sptr = sptr->next;
            }
            ptr2 = ptr2->next;
    }
    else
    {
            temp = createNode(ptr1->coefficient + ptr2->coefficient, ptr1->exponent);
            if (sptr == NULL)
            {
                    sptr = temp;
                    sum = sptr;
            }
            else
            {
                    sptr->next = temp;
                    sptr = sptr->next;
            }
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
```

```
            }
        }
        if (ptr1 == NULL)
        {
            while (ptr2 != NULL)
            {
                temp = createNode(ptr2->coefficient, ptr2->exponent);
                sptr->next = temp;
                ptr2 = ptr2->next;
                sptr = sptr->next;
            }
        }
        if (ptr2 == NULL)
        {
            while (ptr1 != NULL)
            {
                temp = createNode(ptr1->coefficient, ptr1->exponent);
                sptr->next = temp;
                ptr1 = ptr1->next;
                sptr = sptr->next;
            }
        }
}
void main()
{
        int terms;
        printf("Enter the number of terms in Ploynomial 1: ");
        scanf("%d", &terms);
        insertNode(1, terms);
        displayPolynomial(poly1);
        printf("Enter the number of terms in Ploynomial 2: ");
        scanf("%d", &terms);
        insertNode(2, terms);
        displayPolynomial(poly2);
        addPolynomials(poly1, poly2);
        printf("\nPOLYNOMIAL SUM:\n");
        displayPolynomial(sum);
}
```

OUTPUT:

Enter the number of terms in Ploynomial 1: 4
Enter the coefficient of term 1: 12
Enter the exponent of term 1: 4
Enter the coefficient of term 2: 10
Enter the exponent of term 2: 3
Enter the coefficient of term 3: 11
Enter the exponent of term 3: 1
Enter the coefficient of term 4: 15
Enter the exponent of term 4: 0
12x^4 + 10x^3 + 11x^1 + 15x^0

Enter the number of terms in Ploynomial 2: 4
Enter the coefficient of term 1: 10
Enter the exponent of term 1: 5
Enter the coefficient of term 2: 15
Enter the exponent of term 2: 3
Enter the coefficient of term 3: 20
Enter the exponent of term 3: 2
Enter the coefficient of term 4: 9
Enter the exponent of term 4: 1
10x^5 + 15x^3 + 20x^2 + 9x^1

POLYNOMIAL SUM:
10x^5 + 12x^4 + 25x^3 + 20x^2 + 20x^1 + 15x^0

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 the 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 addition using linked list was completed successfully.

Ex.No:8a IMPLEMENTATION OF EVALUATING POSTFIX EXPRESSIONS

AIM

To write a C program for evaluating Postfix Expression using stack

PRE LAB-DISCUSSION

Stack Operations:

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

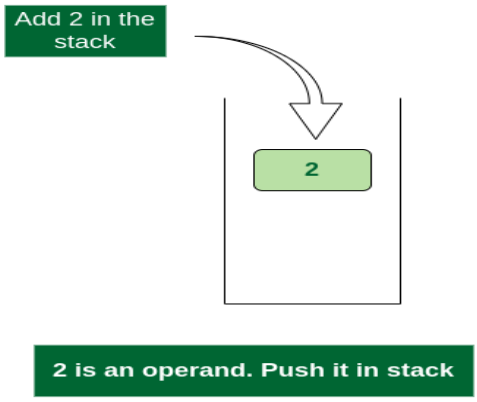
Pop: Removes and returns the top element of the stack.

Postfix Evaluation:

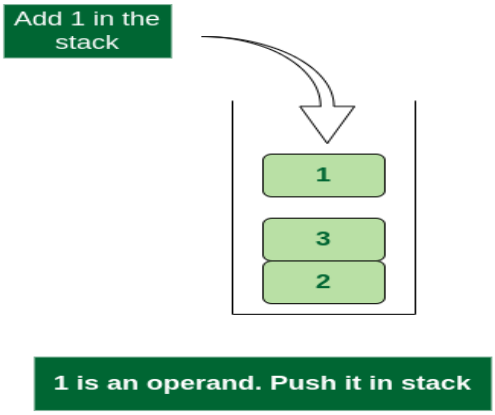
Evaluate Postfix: Iterates through the postfix expression. If a digit is encountered, it is pushed onto the stack. If an operator is encountered, two elements are popped from the stack, the operation is performed, and the result is pushed back onto the stack.

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

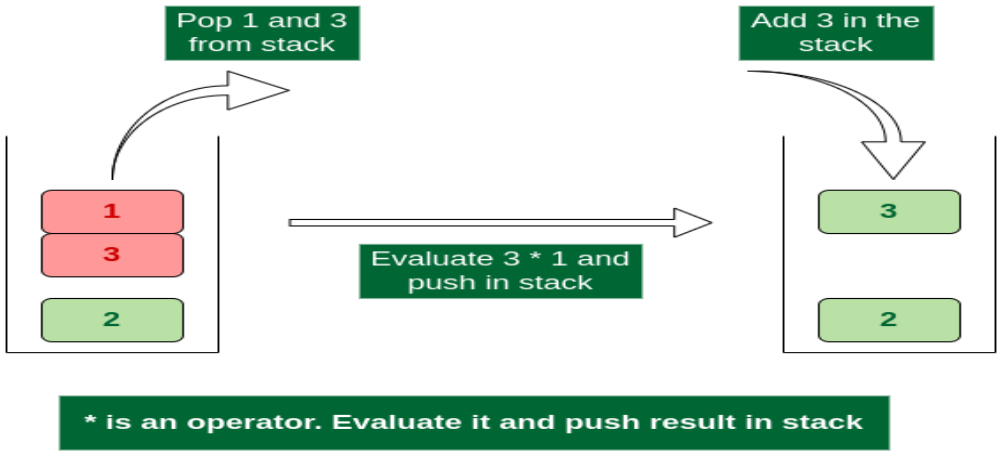


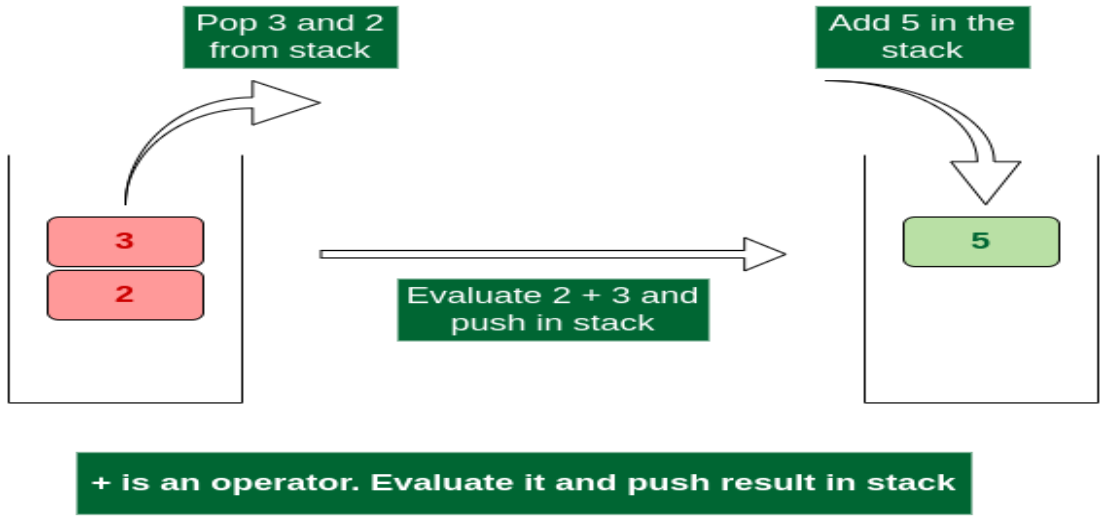
- Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top) Push 3 into stack
- Scan 1, again a number, push it to stack, stack now contains '2 3 1' Push 1 into stack



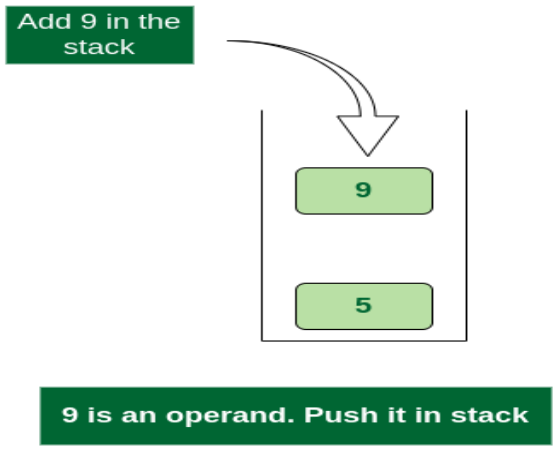
- Scan *, it's an operator. Pop two operands from stack, apply the * operator on operands. We get 3*1 which results in 3. We push the result 3 to stack. The stack now becomes '2 3'. Evaluate * operator and push result in stack

- 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.

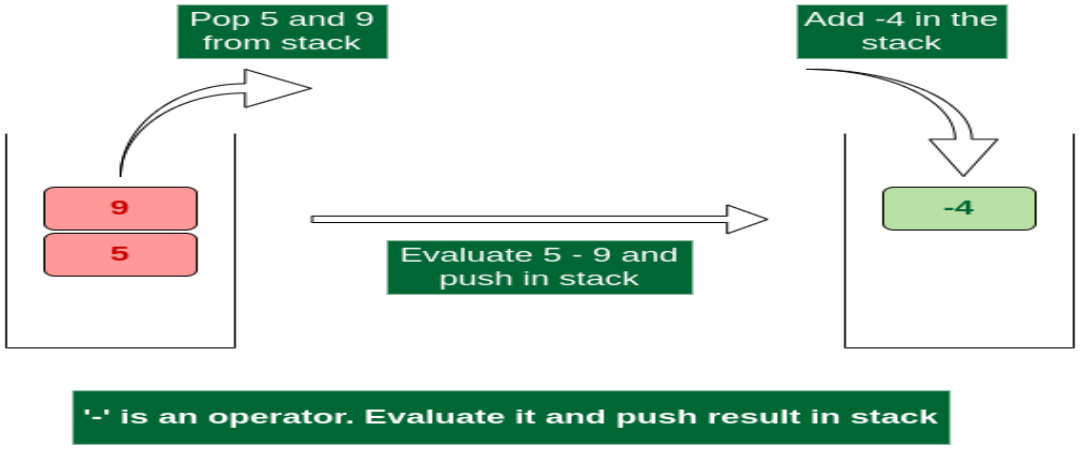




- Scan 9, it's a number. So we push it to the stack. The stack now becomes '5 9'. Push 9 into stack
- Scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9



which results in -4. We push the result -4 to the stack. The stack now becomes '-4'. Evaluate '-' operator and push result in stack



- 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. Start
2. Initialize an empty stack.
3. Read the postfix expression as a string.
4. Repeat for each character ch in the postfix expression until the end of the string:
    - If ch is a digit:
        - Convert ch to an integer (e.g., ch - '0')
        - Push the integer onto the stack.
    - Else if ch is an operator (+, -, *, /, ^):
        - Pop the top element from the stack and store it in variable B.
        - Pop the next top element from the stack and store it in variable A.
        - Perform the operation A (operator) B:
            - If ch is +, compute A + B
            - If ch is -, compute A - B
            - If ch is *, compute A * B
            - If ch is /, compute A / B
            - If ch is ^, compute A ^ B (A raised to the power B)
        - Push the result back onto the stack.
5. After the loop ends, pop the final result from the stack.
6. Print the result.
7. End

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<math.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
                {
                        B = pop();
                        A = pop();
```

```
                        switch (ch)
                        {
                                case '+':
                                        val = A + B;
                                        break;
                                case '-':
                                        val = A - B;
                                        break;
                                case '*':
                                        val = A * B;
                                        break;
                                case '/':
                                        val = A / B;
                                        break;
                                case '^':
                                        val = (int)pow(A, B);
                                        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:

```
Enter the postfix expression: 823^*23*51**-
The result of the postfix expression is: 34

Enter the postfix expression: 231*+9-52/83-**
The result of the postfix expression is: -40
```

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 for 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 of If]
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 of If]
7. END.

ALGORITHM

1. **Start**
2. **Initialize**:
    - An empty character stack.
    - Add a special symbol '#' to the bottom of the stack to mark its base.
    - An empty postfix expression string.
3. **Read the infix expression from left to right**, one symbol at a time.

4. **For each symbol in the infix expression**:
    - **If the symbol is an operand** (letter or digit):
        - Append it directly to the postfix expression.
    - **Else if the symbol is an opening parenthesis (**:
        - Push it onto the stack.
    - **Else if the symbol is a closing parenthesis)**:
        - Pop symbols from the stack and append them to the postfix string until an opening parenthesis (is encountered.
        - Discard the opening parenthesis.
    - **Else if the symbol is an operator** (+, -, *, /, ^, etc.):
        - While the stack is **not empty** and the precedence of the symbol is **less than or equal to** the precedence of the operator at the top of the stack:
            - Pop from the stack and append to the postfix expression.
        - Push the current operator onto the stack.
5. **After reading the entire infix expression**:
    - Pop all remaining operators from the stack and append them to the postfix expression, **until the base symbol # is found**.
6. **Terminate** the postfix expression with the null character \0.
7. **Print** the resulting postfix expression.
8. **End**

PROGRAM

```
#include<stdio.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[] = 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");
        scanf("%s", infix);
        convertInfixToPostfix(infix, postfix);
        printf("The corresponding postfix string is:\n");
        puts(postfix);
        getchar();
}

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-*+

Enter the valid infix string:
A*(B+C)/D
The corresponding postfix string is:
ABC+*D/

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**

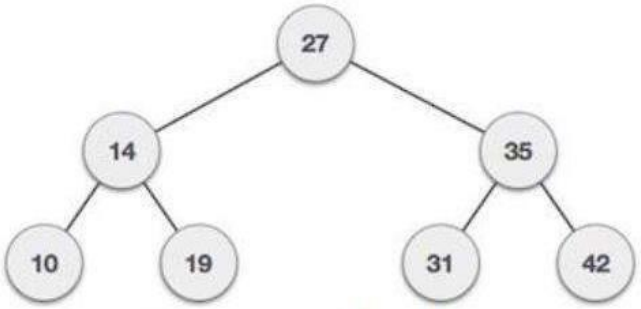
AIM

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!
```

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:10 IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS

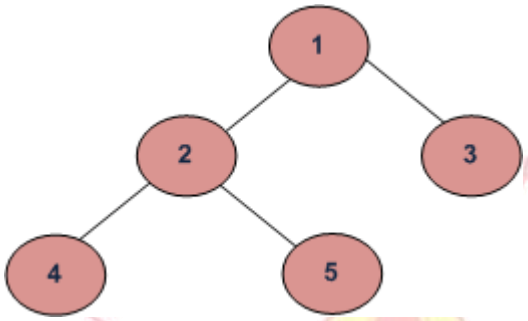
AIM

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
```

VIVA 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 was implemented successfully.

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

AIM

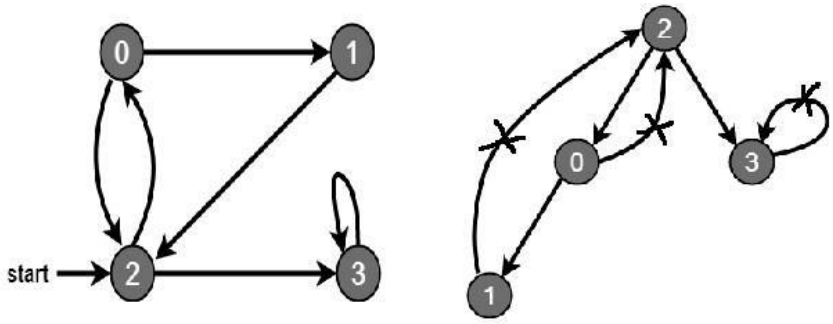
To write a C program for implementing the graph traversal algorithm for Depth first traversal

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

```
#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
```

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**

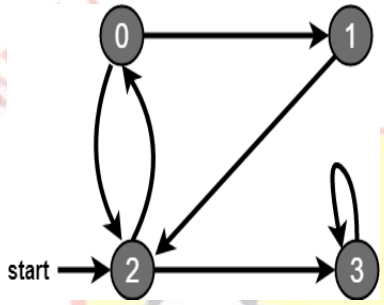
AIM

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

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
```

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 Breadth first graph traversal algorithm 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.

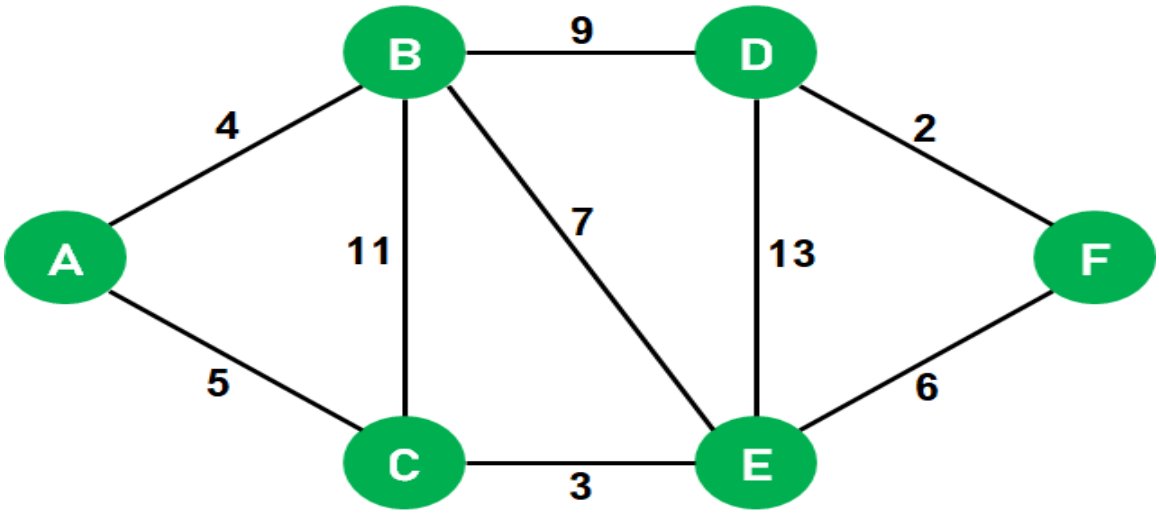
1. **Visited Property:**

- 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

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 **DOUBLY LINKED LIST**

AIM:

To write a C program to implement doubly linked list with Insert, Delete and Display operations.

PRE LAB DISCUSSION

Doubly linked list is a sequence of elements in which every node has link to its previous node and next node.

- Traversing can be done in both directions and displays the contents in the wholelist.

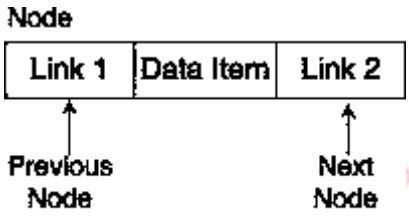


Fig. Doubly Linked List

Link1 field stores the address of the previous node and Link2 field stores the address of the next node. The Data Item field stores the actual value of that node. If we insert a data into the linked list, it will be look like asfollows:

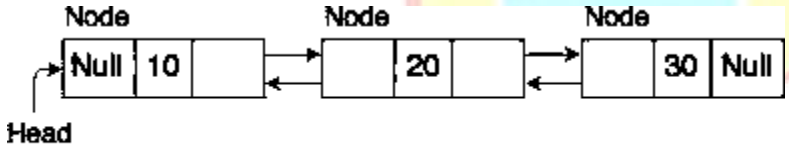


Fig. Doubly Linked List

In doubly linked list contains three fields. In this, link of two nodes allow traversal of the list in either direction. There is no need to traverse the list to find the previous node. We can traverse from head to tail as well as tail to head.

ALGORITHM

1. **Start**
2. Initialize head = NULL
3. Display a menu with options:
    - Insert at beginning
    - Insert at end
    - Delete from beginning
    - Delete from end

    - Display forward
    - Display backward
    - Exit
4. Accept user choice
5. Based on the choice, call the corresponding function
6. Repeat step 3 until the user chooses Exit
7. **End**

- Insert at Beginning
    - Create a new node with given data
    - Set newNode->prev = NULL
    - Set newNode->next = head
    - If head != NULL, then head->prev = newNode
    - Set head = newNode
- Insert at End
    - Create a new node with given data
    - Set newNode->next = NULL
    - If head == NULL, then:
        - Set newNode->prev = NULL
        - Set head = newNode
    - Else:
        - Traverse to the last node (temp)
        - Set temp->next = newNode
        - Set newNode->prev = temp
- Delete from Beginning
    - If head == NULL, print "List is empty", return
    - Store head in a temporary node (temp)
    - Set head = head->next
    - If head != NULL, set head->prev = NULL
    - Free temp
- Delete from End
    - If head == NULL, print "List is empty", return
    - If head->next == NULL, then:
        - Free head
        - Set head = NULL
    - Else:
        - Traverse to the last node (temp)
        - Set temp->prev->next = NULL
        - Free temp
- Display Forward
    - If head == NULL, print "List is empty", return
    - Initialize a pointer temp = head
    - While temp != NULL:
        - Print temp->data

        - Move to next node: temp = temp->next
- Display Backward
    - If head == NULL, print "List is empty", return
    - Traverse to the last node (temp)
    - While temp != NULL:
        - Print temp->data
        - Move to previous node: temp = temp->prev

PROGRAM

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

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

// Head pointer
struct Node* head = NULL;

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

// Insert at the beginning
void insertAtBeginning(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("Inserted %d at the beginning.\n", value);
}
```

```
// Insert at the end
void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Inserted %d at the end.\n", value);
}

// Delete from the beginning
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    if (head != NULL)
        head->prev = NULL;
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

// Delete from the end
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    if (temp->next == NULL) {
        head = NULL;
    } else {
        while (temp->next != NULL)
            temp = temp->next;
        temp->prev->next = NULL;
    }
```

```
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}

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

// Display list from end
void displayBackward() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;

    printf("List (backward): ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
}

// Main function
int main() {
    int choice, value;
    while (1) {
        printf("\n--- Doubly Linked List Menu ---\n");
        printf("1. Insert at Beginning\n2. Insert at End\n3. Delete from Beginning\n4. Delete from End\n");
        printf("5. Display Forward\n6. Display Backward\n7. Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;
            case 3:
                deleteFromBeginning();
                break;
            case 4:
                deleteFromEnd();
                break;
            case 5:
                displayForward();
                break;
            case 6:
                displayBackward();
                break;
            case 7:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    return 0;
}
```

OUTPUT

```
--- Doubly Linked List Menu ---
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display Forward
6. Display Backward
7. Exit
```

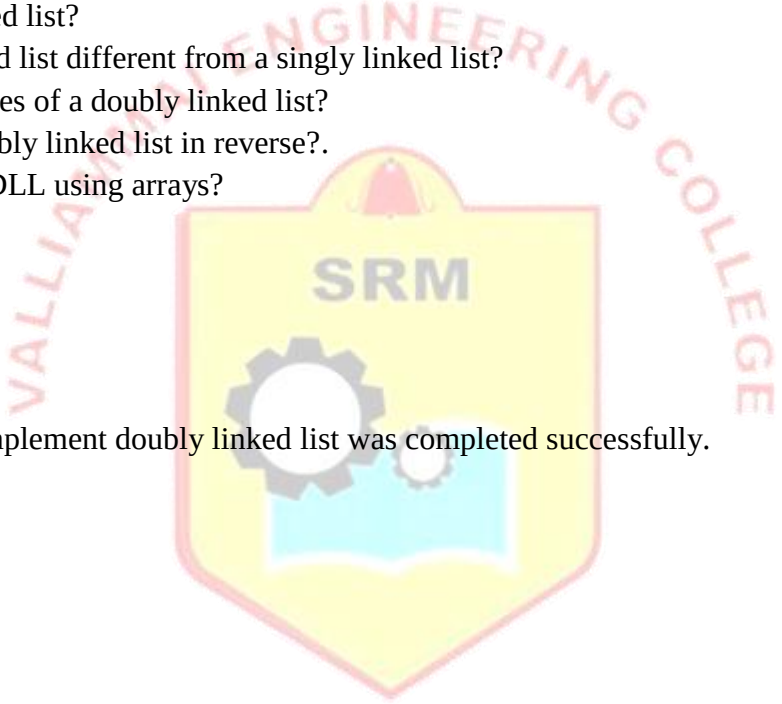
Enter your choice: 1
Enter value to insert: 10
Inserted 10 at the beginning.

Enter your choice: 2
Enter value to insert: 20
Inserted 20 at the end.

Enter your choice: 5
List (forward): 10 <-> 20 <-> NULL

VIVA QUESTIONS

1. What is a doubly linked list?
2. How is a doubly linked list different from a singly linked list?
3. What are the advantages of a doubly linked list?
4. Can we traverse a doubly linked list in reverse?.
5. Can we implement a DLL using arrays?



RESULT

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