LABORATORY MANUAL

B.Tech. Semester- II

DATA STRUCTURE LAB
Subject code: CSE-102P

Prepared by:
Prof. Sonika Lakra
Sign.: ……………………..

Checked by:
Prof. Megha Goel
Sign.: …………………..

Approved by:
Name : Prof. (Dr.) Isha Malhotra
Sign.: …………………..

DEPARTMENT OF APPLIED SCIENCE & HUMANITIES
DRONACHARYA COLLEGE OF ENGINEERING
KHENTAWAS, FARRUKH NAGAR, GURUGRAM (HARYANA)

Table of Contents

1. Vision and Mission of the Institute
2. Vision and Mission of the Department
3. Programme Educational Objectives (PEOs)
4. Programme Outcomes (POs)
5. Programme Specific Outcomes (PSOs)
6. University Syllabus
7. Course Outcomes (COs)
8. Course Overview
9. List of Experiments DOs and DON'Ts
10. General Safety Precautions
11. Guidelines for students for report preparation
12. Lab assessment criteria
13. Details of Conducted Experiments
14. Lab Experiments

Vision and Mission of the Institute

Vision:

"To impart Quality Education, to give an enviable growth to seekers of learning, to groom them as World Class Engineers and managers competent to match the expending expectations of the Corporate World has been ever enlarging vision extending to new horizons of Dronacharya College of Engineering."

Mission:

M1. To prepare students for full and ethical participation in a diverse society and encourage lifelong learning by following the principle of 'Shiksha evam Sahayata' i.e. Education & Help.
M2. To impart high-quality education, knowledge and technology through rigorous academic programs, cutting-edge research, & Industry collaborations, with a focus on producing engineers& managers who are socially responsible, globally aware, & equipped to address complex challenges.
M3. Educate students in the best practices of the field as well as integrate the latest research into the academics.
M4. Provide quality learning experiences through effective classroom practices, innovative teaching practices and opportunities for meaningful interactions between students and faculty.
M5. To devise and implement program of education in technology that are relevant to the changing needs of society, in terms of breadth of diversity and depth of specialization.

Vision and Mission of the Department

Vision:

To lay a strong foundation for the first-year students of the engineering discipline in the area of Applied Sciences and Humanities with a view to make them capable of innovating and inventing engineering solutions and also develop students as capable and responsible citizens of our nation.

Mission:

- To build strong fundamental knowledge and ability for application in students and make them capable to apply knowledge of mathematics and science to the solution of complex engineering problems.
- To impart knowledge, leading to understanding between engineering and other core areas of Applied Sciences and Humanities.
- To provide students the basic tools of analysis, as well as the knowledge of the principles on which engineering is based.
- To strive to inculcate the scientific temper and the spirit of enquiry in the students.
- To make students achieve a superior level in communication and presentation skills.
- To foster values and ethics and make students responsible citizens of India.
- To pursue inter-disciplinary research for the larger good of the society.

Program Educational Objectives (PEOs)

PEO1: To provide students with a sound knowledge of mathematical, scientific and engineering fundamentals required to solve real world problems.

PEO2: To develop research oriented analytical ability among students and to prepare them for making technical contribution to the society.

PEO3: To develop in students the ability to apply state-of-the–art tools and techniques for designing software products to meet the needs of Industry with dueconsideration for environment friendly and sustainable development.

PEO4: To prepare students with effective communication skills, professional ethics and managerial skills.

PEO5: To prepare students with the ability to upgrade their skills and knowledgefor life-long learning.

Program Outcomes (POs)

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

PO2: Problem analysis: Identify, formulate, review research literature, and analysis complex engineering problems reaching substantiated conclusions usingfirst principles of mathematics, natural sciences, and engineering sciences.

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

PO4: Conduct investigations of complex problems: Use research-basedknowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

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

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

PO7: Environment and sustainability: Understand the impact of the professionalengineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

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

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

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

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

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

University Syllabus

1. WAP to create a function to swap two numbers using call by value.
2. WAP to implement call by reference for swapping of two numbers.
3. WAP to create an array.
4. WAP to insert a new number in an array at any location.
5. WAP to delete a number in an array at any location.
6. WAP to search an element into an array using linear search.
7. WAP for finding the element in an array using binary search method using iteration and recursion.
8. WAP that implements the following sorting: -Insertion Sort, Selection Sort, Bubble Sort, Merge Sort and Quick Sort.
9. WAP that implement stack using array and linked list.
10. WAP that implement queue using array and linked list.
11. WAP to create a linked list & perform operations such as insert, delete, update and reverse in the linked list.
12. WAP to perform the following operations:-

    #Insert an element into a binary search tree.

    #Delete an element from a binary search tree.

    #Search for a key element in a binary search tree.

Course Outcomes (COs)

Upon successful completion of the course, the students will acquire:

C102.1: Basic understanding of C syntax. Learn the fundamental syntax and structure of the C programming language, including variables, data types, loops, conditionals and functions.

C102.2: Proficiency in writing C code. Gain the ability to write C programs to solve simple to moderately complex problems.

C102.3: Understanding of Algorithms. Student will learn how to create algorithms to write more modular and reusable code.

C102.4: Understanding of GUI concepts: Students will learn the basic concepts andprinciples of GUI design and development, including user interfaces,widgets, event handling, and layout management.

C102.2: Introduction to data analysis and visualization. Students will learn techniques to analyze and visualise data by plotting different graphs and trees.

CO-PO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C102.1		3			3			2	2	2	2	2
C102.2			2		3			2	2	2	2	2
C102.3				2	3			2	2	2	2	2
C102.4				2	3			2	2	2	2	2
C102.5				2	3			2	2	2	2	2

CO-PSO Mapping

	PSO1	PSO2	PSO3
C102.1	2	2	
C102.2		2	
C102.3		2	
C102.4		2	
C102.5		2	

Course Overview

Data Structure lab introduces a comprehensive overview of the C programming language, covering its syntax, data structures, control flow, object-oriented programming, file handling, modules and libraries, data analysis and visualization. Through practical exercises and projects, students gain proficiencyin C programming, learn to write clean and readable code, work with external libraries, handle files and databases, and analyze data. The course equipsstudents with the skills and knowledge to solve real-world problems usingC

List of Experiments mapped with COs

S No.	List of Experiments	Course Outcome	Page No.
1	WAP to create a function to swap two numbers using call by value.	CO102.1	1-3
2	WAP to implement call by reference for swapping of two numbers.	CO102.1	4-5
3	WAP to create an array.	CO102.1	6-8
4	WAP to insert a new number in an array at any location.	CO102.1	9-11
5	WAP to delete a number in an array at any location.	CO102.1	12-14
6	WAP to search an element into an array using linear search.	CO102.2	15-16
7	WAP for finding the element in an array using binary search method using iteration and recursion.	CO102.2	17-20
8	WAP that implements the following sorting: -Insertion Sort, Selection Sort, Bubble Sort, Merge Sort and Quick Sort.	CO102.3	21-30
9	WAP that implement stack using array and linked list.	CO102.3	31-38
10	WAP that implement queue using array and linked list.	CO102.3	39-45
11	WAP to create a linked list & perform operations such as insert, delete, update and reverse in the linked list.	CO102.4	46-55
12	WAP to perform the following operations: - a) Insert an element into a binary search tree. b) Delete an element from a binary search tree. c) Search for a key element in a binary search tree.	CO102.5	56-60

DOs and DON'Ts

DOs

1. Login-on with your username and password.
2. Log off the computer every time when you leave the Lab.
3. Arrange your chair properly when you are leaving the lab.
4. Put your bags in the designated area.
5. Ask permission to print.

DON'Ts

1. Do not share your username and password.
2. Do not remove or disconnect cables or hardware parts.
3. Do not personalize the computer setting.
4. Do not run programs that continue to execute after you log off.
5. Do not download or install any programs, games or music on computer in Lab.
6. Personal Internet use chat room for Instant Messaging (IM) and Sites is strictly prohibited.
7. No Internet gaming activities allowed.
8. Tea, Coffee, Water & Eatables are not allowed in the Computer Lab.

General Safety Precautions

Precautions (In case of Injury or Electric Shock)

1. To break the victim with live electric source, use an insulator such as fire wood or plastic tobreak the contact. Do not touch the victim with bare hands to avoid the risk of electrifying yourself.

2. Unplug the risk of faulty equipment. If main circuit breaker is accessible, turn the circuit off.

3. If the victim is unconscious, start resuscitation immediately, use your hands to press the chestin and out to continue breathing function. Use mouth-to-mouth resuscitation if necessary.

4. Immediately call medical emergency and security. Remember! Time is critical; be best.

Precautions (In case of Fire)

1. Turn the equipment off. If power switch is not immediately accessible, take plug off.

2. If fire continues, try to curb the fire, if possible, by using the fire extinguisher or by covering itwith a heavy cloth, if possible, isolate the burning equipment from the other surrounding equipment.

3. Sound the fire alarm by activating the nearest alarm switch located in the hallway.

4. Call security and emergency department immediately:

Emergency: Reception
Security: Main Gate

Guidelines to students for report preparation

All students are required to maintain a record of the experiments conducted by them. Guidelines for its preparation are as follows :-

1) All files must contain a title page followed by an index page. ***The files will not be signed bythe faculty without an entry in the index page.***

2) Student's Name, Roll number and date of conduction of experiment must be written on allpages.

3) For each experiment, the record must contain the following

(i) Aim/Objective of the experiment

(ii) Pre-experiment work (as given by the faculty)

(iii) Lab assignment questions and their solutions

(iv) Test Cases (if applicable to the course)

(v) Results/ output

Note:

1. Students must bring their lab record along with them whenever they come for the lab.
2. Students must ensure that their lab record is regularly evaluated.

Lab Assessment Criteria

An estimated 10 lab classes are conducted in a semester for each lab course. These lab classes are assessed continuously. Each lab experiment is evaluated based on 5 assessment criteria as shown in following table. Assessed performance in each experiment is used to compute CO attainment as well as internal marks in the lab course.

Grading Criteria	Exemplary (4)	Competent (3)	Needs Improvement (2)	Poor (1)
AC1: Pre-Lab written work (this may be assessed through viva)	Complete procedure with underlined concept is properly written	Underlined concept is written but procedure is incomplete	Not able to write concept and procedure	Underlined concept is not clearly understood
AC2: Program Writing/ Modeling	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied, Program/solution written is readable	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied	Assigned problem is properly analyzed & correct solution designed	Assigned problem is properly analyzed
AC3: Identification & Removal of errors/ bugs	Able to identify errors/ bugs and remove them	Able to identify errors/ bugs and remove them with little bit of guidance	Is dependent totally on someone for identification of errors/ bugs and their removal	Unable to understand the reason for errors/ bugs even after they are explicitly pointed out
AC4:Execution & Demonstration	All variants of input /output are tested, Solution is well demonstrated and implemented concept is clearly explained	All variants of input /output are not tested, However, solution is well demonstrated and implemented concept is clearly explained	Only few variants of input /output are tested, Solution is well demonstrated but implemented concept is not clearly explained	Solution is not well demonstrated and implemented concept is not clearly explained
AC5:Lab Record Assessment	All assigned problems are well recorded with objective, design constructs and solution along with Performance analysis using all variants of input and output	More than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output	Less than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output	

LAB EXPERIMENTS

LAB EXPERIMENT 1

OBJECTIVE:

WAP to create a function to swap two numbers using call by value.

PRE-EXPERIMENT QUESTIONS:

Q1. What are different data types in data structure?

Q2. What is the use of print function in data structure?

BRIEF DISCUSSION AND EXPLANATION:

To create a function to swap two numbers using call by value, follow these steps:

Step 1: Start the program.

Step 2: Set a ← 10 and b ← 20

Step 3: Call the function swap(a,b)

Step 3a: Start function.

Step 3b: Assign t ← x

Step 3c: Assign x ← y

Step 3d: Assign y ← t

Step 3e: Print x and y.

Step 3f: End function.

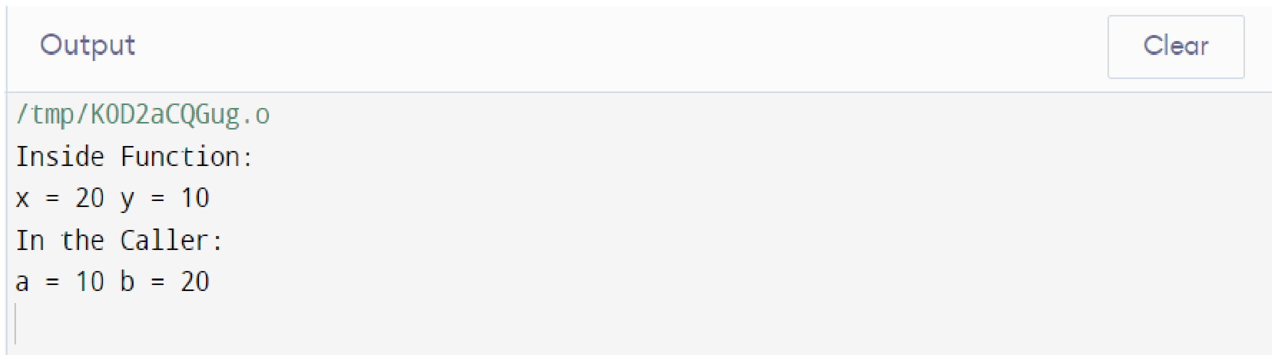
Step 4: Stop the program.

PROGRAM:

```
#include <stdio.h>
void swapx(int x, int y);
int main()
{
        int a = 10, b = 20;
        swapx(a, b);
        printf("In the Caller:\na = %d b = %d\n", a, b);
        return 0;
}
void swapx(int x, int y)
{
        int t;
        t = x;
```

```
        x = y;
        y = t;
        printf("Inside Function:\nx = %d y = %d\n", x, y);
}
```

OUTPUT



QUIZ WITH ANSWERS:

Q1. How do you define variables in C?
Ans: In C, variables are defined by specifying the variable's data type and name. Here's the general syntax for variable definition in C:
data_type variable_name;

Q2. How do you save and compile a C file?
Ans: Open a text editor such as Notepad, Notepad++, or Visual Studio Code.
Write your C code in the text editor and save the file with a .c extension. For example, you can save it as "myprogram.c".
Open the Command Prompt by pressing the Windows key + R, typing "cmd", and hitting Enter.
In the Command Prompt, navigate to the directory where you saved your C file using the cd command. For example, if your file is saved in the "Documents" folder, you can use the command cd Documents to navigate there.
Once you are in the correct directory, compile the C file using a C compiler like GCC (GNU Compiler Collection) by running the command gcc -o outputfilename sourcefilename.c. For example, if your source file is "myprogram.c" and you want the output file to be named "myprogram.exe", you can use the command gcc -o myprogram.exe myprogram.c.
If there are no syntax errors or other issues in your code, the compiler will generate an executable file in the same directory.
You can run the compiled program by typing its name in the Command Prompt and pressing Enter. For example, in our case, you can run the program by typing myprogram.exe.

Q3. What is a function?
Ans: In computer programming, a function is a named block of code that performs a specific task or a set of instructions. It is a fundamental building block of modular programming and helps in organizing and reusing code. Functions can be called or invoked from different parts of a program to execute the code within them.

Q4. How parameters are passed in a function?
Ans: Parameters can be passed to a function in programming languages using different methods. The two main methods are:
Pass by Value: In pass by value, a copy of the parameter's value is passed to the function. This means that any modifications made to the parameter within the function do not affect the original data outside the function. The function works with its own local copy of the parameter value.
When a parameter is passed by value, the function receives a copy of the value, and any changes made to the parameter within the function are confined to the function's scope.
Pass by Reference: In pass by reference, the memory address (reference) of the parameter is passed to the function. This allows the function to directly access and modify the original data outside the function. Any changes made to the parameter within the function are reflected in the original data.

Q5. What is the use of return statement of a function?
Ans: The return statement in a function is used to specify the value that the function should return when it is called or invoked. When a return statement is encountered in a function, it terminates the execution of the function and passes the specified value back to the caller.

LAB EXPERIMENT 2

OBJECTIVE:

WAP to implement call by reference for swapping of two numbers.

PRE-EXPERIMENT QUESTIONS:

1. What are the two methods you plan to use to swap the numbers?
2. Are the numbers to be swapped integers or floating-point numbers?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm that implements call by reference for swapping of two numbers:

Step 1: Start the program.

Step 2: Set a ← 10 and b ← 20

Step 3: Call the function swap(&a,&b)

Step 3a: Start function

Step 3b: Assign t ← *x

Step 3c: Assign *x ← *y

Step 3d: Assign *y ← t

Step 3e: End function

Step 4: Print x and y.

Step 5: Stop the program.

PROGRAM:

```
#include <stdio.h>
void swapx(int*, int*);
int main()
{
        int a = 10, b = 20;
        swapx(&a, &b);
        printf("Inside the Caller:\na = %d b = %d\n", a, b);
        return 0;
}
void swapx(int* x, int* y)
{
        int t;
        t = *x;
        *x = *y;
```

```
    *y = t;
    printf("Inside the Function:\nx = %d y = %d\n", *x, *y);
}
```

OUTPUT

```
Output                                                    Clear
/tmp/KOD2aCQGug.o
Inside the Function:
x = 20 y = 10
Inside the Caller:
a = 20 b = 10
```

QUIZ WITH ANSWERS:

Q1. What were the two methods you used to swap the numbers?
Ans: Call by Value and Call by reference

Q2. Did you allow user input for the numbers or were they predefined?
Ans: Predefined

Q3. What are pointers?
Ans: In computer programming, a pointer is a variable that stores the memory address of another variable or object. It "points" to the memory location where the data is stored rather than directly holding the data itself. By using pointers, you can indirectly access and manipulate data stored in memory.

Q4. What is the main difference between call by value and call by reference?
Ans: Call by Value: In call by value, a copy of the value of the actual parameter is passed to the function. Any modifications made to the parameter within the function do not affect the original data outside the function. The function works with its own local copy of the parameter value.
Call by Reference: In call by reference, the memory address of the actual parameter is passed to the function. This allows the function to directly access and modify the original data outside the function. Any changes made to the parameter within the function are reflected in the original data.

Q5. What is the use of & operator?
Ans: The & operator in programming languages is commonly used in conjunction with pointers to obtain the memory address of a variable. It is known as the "address-of" operator.

LAB EXPERIMENT 3

OBJECTIVE:

WAP to create an array.

PRE-EXPERIMENT QUESTIONS:

1. What do you mean by array?
2. Should the program handle any specific error cases, such as non-numericinput?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm that determines creation of an array:

STEP 1: START.
STEP 2: INITIALIZE arr[] = {1, 2, 3, 4, 5}.
STEP 3: length= sizeof(arr)/sizeof(arr[0])
STEP 4: PRINT "Elements of given array:"
STEP 5: i=0. REPEAT STEP 6 and STEP 7 UNTIL i<length.
STEP 6: PRINT arr[i]
STEP 7: i=i+1.
STEP 8: RETURN 0.

PROGRAM:

```
#include <stdio.h>
int main () {
  int n[ 10 ]; /* n is an array of 10 integers */
  int i,j;
  /* initialize elements of array n to 0 */
  for ( i = 0; i < 10; i++ ) {
    n[ i ] = i + 100; /* set element at location i to i + 100 */
  }
  /* output each array element's value */
  for (j = 0; j < 10; j++ ) {
    printf("Element[%d] = %d\n", j, n[j] );
  }
  return 0;
}
```

OUTPUT

Output Clear

```
/tmp/KOD2aCQGug.o
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

QUIZ WITH ANSWERS

Q1. How can you define the size of an array?
Ans: In C, the size of an array can be defined in a couple of ways:
Using a Constant Value:
You can define the size of an array by explicitly specifying a constant value within square brackets at the time of array declaration.
For example, to define an integer array with a size of 5, you can write:
int myArray[5];
Using a Preprocessor Constant:
Alternatively, you can use a preprocessor constant (a macro defined using the #define directive) to specify the array size.
This allows for more flexibility as the size can be easily changed in a single place.
For example, you can define an array with a size of 10 using a preprocessor constant as follows:
#define ARRAY_SIZE 10
int myArray[ARRAY_SIZE];

Q2. What is the syntax to initialize an array?
Ans: This is the most common way to initialize an array in C. // declare an array. int my_array[5];

Q3. What is an array?
Ans: An array is a data structure that stores a fixed-size sequence of elements of the same type. It provides a way to store multiple values under a single variable name. Each element in the array is identified by its index, which represents its position within the array.

Q4. What are the characteristics of array?
Ans: Some important characteristics of arrays include:

1. Size: Arrays have a fixed size defined during their declaration, and the size remains constant throughout their lifetime. This means that the number of elements in an array is predetermined and cannot be changed dynamically.

2. Homogeneity: Arrays store elements of the same data type. For example, an array of integers can only store integer values, and an array of strings can only store string values. This ensures data consistency within the array.
3. Contiguous Memory: The elements of an array are stored in contiguous memory locations, meaning they are stored one after another in memory. This allows for efficient memory access and arithmetic operations on indices.

Q5. How arrays are stored in memory?
Ans: Arrays are stored in contiguous memory locations, meaning that the elements of an array are stored one after another in memory. The exact memory layout and storage mechanism of arrays can vary slightly depending on the programming language and the underlying system architecture. However, the fundamental concept remains the same.

LAB EXPERIMENT 4

OBJECTIVE:

WAP to insert a new number in an array at any location.

PRE-EXPERIMENT QUESTIONS:

1. Should the program take user input for the number or should it be predefined?
2. How should the program handle non-numeric inputs or invalid inputs?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm to insert a new number in an array at any location:

1. [Initialize Counter] Set J: = N.
2. Repeat steps 3 and 4 while J ≥ K.
3. [Move Jth element downward] Set LA [J+1]: = LA [J]
4. [Decrease Counter] Set J: = J-1. [End of step 2 loop]
5. [Insert element] Set LA [K]: = ITEM.
6. [Reset N] Set N: = N+1.
7. Exit.

PROGRAM:

```
#include <stdio.h>

int main()

{

        int arr[100] = { 0 };

        int i, x, pos, n = 10;

        // initial array of size 10

        for (i = 0; i < 10; i++)

                arr[i] = i + 1;

        // print the original array
```

```
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    // element to be inserted
    x = 50;
    // position at which element is to be inserted
    pos = 5;
    // increase the size by 1
    n++;
    // shift elements forward
    for (i = n - 1; i >= pos; i--)
        arr[i] = arr[i - 1];
    // insert x at pos
    arr[pos - 1] = x;
    // print the updated array
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

OUTPUT

```
Output                                                        Clear
/tmp/KOD2aCQGug.o
1 2 3 4 5 6 7 8 9 10
1 2 3 4 50 5 6 7 8 9 10
```

QUIZ WITH ANSWERS:

Q1. Did you allow user input for the number or was it predefined?
Ans: Predefined

Q2. How did you handle cases where the input was not a valid number?
Ans: Printed error message

Q3. What is the size of array taken initially?
Ans: 100

Q4. How many inputs are given?
Ans: Two (Number and position)

Q5. How many outputs are generated?
Ans: After insert operation, one extra element is added in the input array.

LAB EXPERIMENT 5

OBJECTIVE:

WAP to delete a number in an array at any location.

PRE-EXPERIMENT QUESTIONS:

1. How can you insert a number at last position in an array?
2. How can you insert a number at first position in an array?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm to delete a number in an array at any location.

Step 01: Start.
Step 02: [Initialize counter variable.] ...
Step 03: Repeat Step 04 and 05 for i = pos - 1 to i < size.
Step 04: [Move i^{th} element backward (left).] ...
Step 05: [Increase counter.] ...
Step 06: [End of step 03 loop.]
Step 07: [Reset size of the array.] ...
Step 08: Stop.

PROGRAM:

```
#include<stdio.h>

void main() {

   int j, i, a[100], n, key, l;

   printf("Enter the number of elements:");

   scanf("%d", &n);

   printf("\nEnter the elements:\n");

   for (i = 0; i < n; i++)

      scanf("%d", &a[i]);

   printf("\nEnter the element to delete:");
```

```
  scanf("%d", &key);
  l = n;  //Length of the array
  for (i = 0; i < l; i++) {
    if (a[i] == key) {
      for (j = i; j < l; j++)
        a[j] = a[j + 1];
      l--;    //Decreasing the length of the array
    }
  }
  printf("\nThe new array is \n");
  for (i = 0; i < l; i++)
    printf("%d ", a[i]);
}
```

OUTPUT

```
Output                                                    Clear
/tmp/7Ndjb50YzZ.o
Enter the number of elements:5
Enter the elements:
1
2
3
4
5
Enter the element to delete:3
The new array is
1 2 4 5
```

QUIZ WITH ANSWERS:

Q1. Did you allow user input for the number or was it predefined?
Ans: User Defined

Q2. How did you handle cases where the input was not a valid number?
Ans: Printed error message

Q3. What is the size of array taken initially?
Ans: 100

Q4. How many inputs are given?
Ans: One (Position)

Q5. How many outputs are generated?
Ans: After deletion operation, one element is removed from the input array.

LAB EXPERIMENT 6

OBJECTIVE:

WAP to search an element into an array using linear search.

PRE-EXPERIMENT QUESTIONS:

1. What do you mean by searching?

2. What will be the output if number not found in list?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm to search an element into an array using linear search:

1. Set j to 1.
2. If j > n, jump to step 7.
3. If X[j] == i, jump to step 6.
4. Then, increment j by 1 i.e. j = j+1.
5. Go back to step 2.
6. Display the element i which is found at particular index i, then jump to step 8.
7. Display element not found in the set of input elements.
8. Exit/End.

PROGRAM:

```
#include <stdio.h>
int search(int arr[], int N, int x)
{
        int i;
        for (i = 0; i < N; i++)
                if (arr[i] == x)
                        return i;
        return -1;
}
int main(void)
{
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;
        int N = sizeof(arr) / sizeof(arr[0]);
```

```
        // Function call
        int result = search(arr, N, x);
        (result == -1)
            ? printf("Element is not present in array")
            : printf("Element is present at index %d", result);
        return 0;
    }
```

OUTPUT

```
Output                                   Clear
/tmp/7Ndjb50YzZ.o
Element is present at index 3
```

QUIZ WITH ANSWERS:

Q1. Did you consider any optimizations or improvements in your implementation?
Ans: Yes, Binary Search using sorted elements.

Q2. What do you mean by increment operator?
Ans: It adds the value by1

Q3. What is the time complexity of this program?
Ans: O(n)

Q4. What is linear search?
Ans: Linear search, also known as sequential search, is a simple searching algorithm used to find a particular element in a collection of elements. It sequentially checks each element in the collection until a match is found or the entire collection has been traversed.

Q5. Explain the working of Linear Search.
Ans: Start at the beginning of the collection.
Compare the target element with the current element.
If the target element matches the current element, the search is successful, and the position or index of the element is returned.
If the target element does not match the current element, move to the next element in the collection.
Repeat steps 2-4 until either a match is found or the end of the collection is reached.
If the end of the collection is reached without finding a match, the search is unsuccessful, and a special value (e.g., -1) or an appropriate indication is returned.

LAB EXPERIMENT 7

OBJECTIVE:

WAP for finding the element in an array using binary search method using iteration and recursion.

PRE-EXPERIMENT QUESTIONS:

1. Why do we use iteration call in a program?
2. How do we call recursion in a program?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm for finding the element in an array using binary search method using iteration and recursion:

Step 1: Find the middle element of array, using , middle = initial_value + end_ value / 2;
Step 2: If middle = element, return "element found" and index.
Step 3: if middle > element, call the function with end_value = middle - 1
Step 4: if middle < element, call the function with start_value = middle + 1
Step 5: exit

PROGRAM:

(A) Iteration

```
#include <stdio.h>
// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
while (l <= r)
{
int m = l + (r-l)/2;
// Check if x is present at mid
if (arr[m] == x)
return m;
// If x greater, ignore left half
if (arr[m] < x)
l = m + 1;
// If x is smaller, ignore right half
else
r = m - 1;
}
```

```
// if we reach here, then element was not present
return -1;
}
int main(void)
{
int arr[] = {2, 3, 4, 10, 40};
int n = sizeof(arr)/ sizeof(arr[0]);
int x = 10;
int result = binarySearch(arr, 0, n-1, x);
(result == -1)? printf("Element is not present in array")
: printf("Element is present at index %d", result);
return 0;
}
```

(B) Recursion

```
#include <stdio.h>
// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
if (r >= l)
{
int mid = l + (r - l)/2;
// If the element is present at the middle itself
if (arr[mid] == x) return mid;
// If element is smaller than mid, then it can only be present
// in left subarray
if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);
// Else the element can only be present in right subarray
return binarySearch(arr, mid+1, r, x);
}
// We reach here when element is not present in array
return -1;
}
int main(void)
{
int arr[] = {2, 3, 4, 10, 40};
int n = sizeof(arr)/ sizeof(arr[0]);
int x = 10;
int result = binarySearch(arr, 0, n-1, x);
(result == -1)? printf("Element is not present in array")
: printf("Element is present at index %d", result);
return 0;
}
```

OUTPUT

Output Clear

```
/tmp/7Ndjb50YzZ.o
Element is present at index 3
```

Output Clear

```
/tmp/7Ndjb50YzZ.o
Element is present at index 3
```

QUIZ WITH ANSWERS:

Q1. Which searching technique is best and why?
Ans: The best searching technique depends on several factors, including the size of the collection, the nature of the data, and the specific requirements of the problem at hand. There is no one-size-fits-all best searching technique for all scenarios. Different searching techniques have different strengths and weaknesses.

Q2. Is binary search applicable on unsorted array or not?
Ans: No

Q3. What is the time complexity of binary search?
Ans: O(logn)

Q4. What is the space complexity of binary search?
Ans: The space complexity of binary search is O(1), which means it uses a constant amount of additional space regardless of the size of the input.

Q5. What is the difference between linear search and binary search?
Ans: The main differences between linear search and binary search are as follows:

1. Search Space:
    - Linear Search: In linear search, the entire collection is sequentially traversed from the beginning to the end.
    - Binary Search: Binary search requires the collection to be sorted. It repeatedly divides the search space in half by comparing the target element with the middle element of the sorted collection.
2. Time Complexity:
    - Linear Search: Linear search has a time complexity of O(n), where n is the number of

elements in the collection. It searches through all the elements until a match is found or the end of the collection is reached.
  - Binary Search: Binary search has a time complexity of O(log n), where n is the number of elements in the collection. It halves the search space in each iteration, making it more efficient for larger collections.
3. Requirement of Sorted Data:
   - Linear Search: Linear search does not require the data to be sorted. It can be used on both sorted and unsorted collections.
   - Binary Search: Binary search requires the collection to be sorted in ascending or descending order. It relies on the sorted order to divide the search space and determine whether to continue searching in the left or right half.
4. Performance:
   - Linear Search: Linear search is suitable for small collections or unsorted data. It performs well when the target element is near the beginning of the collection or when there are only a few elements.
   - Binary Search: Binary search is efficient for larger collections. It quickly reduces the search space by half in each iteration, making it ideal for sorted collections with a large number of elements.

LAB EXPERIMENT 8

OBJECTIVE:

WAP that implements the following sorting: -

Insertion Sort, Selection Sort, Bubble Sort, Merge Sort and Quick Sort.

PRE-EXPERIMENT QUESTIONS:

1. What do you mean by sorting?
2. How do you define complexity of a program?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm for Insertion Sort:-

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

PROGRAM:

```
#include <math.h>
#include <stdio.h>
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++) {
		key = arr[i];
		j = i - 1;

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

```
            }
            arr[j + 1] = key;
        }
}
// A utility function to print an array of size n
void printArray(int arr[], int n)
{
        int i;
        for (i = 0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}
int main()
{
        int arr[] = { 12, 11, 13, 5, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);
        insertionSort(arr, n);
        printArray(arr, n);

        return 0;
}
```

OUTPUT

```
Output                                                  Clear
/tmp/nWAXzbzJWN.o
5 6 11 12 13
```

Algorithm for Selection Sort:-

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

PROGRAM:

```
#include <stdio.h>
void swap(int *xp, int *yp)
{
        int temp = *xp;
```

```
        *xp = *yp;
        *yp = temp;
}
void selectionSort(int arr[], int n)
{
        int i, j, min_idx;
        // One by one move boundary of unsorted subarray
        for (i = 0; i < n-1; i++)
        {
                // Find the minimum element in unsorted array
                min_idx = i;
                for (j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                        min_idx = j;
                // Swap the found minimum element with the first element
                swap(&arr[min_idx], &arr[i]);
        }
}
/* Function to print an array */
void printArray(int arr[], int size)
{
        int i;
        for (i=0; i < size; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

// Driver program to test above functions
int main()
{
        int arr[] = {64, 25, 12, 22, 11};
        int n = sizeof(arr)/sizeof(arr[0]);
        selectionSort(arr, n);
        printf("Sorted array: \n");
        printArray(arr, n);
        return 0;
}
```

OUTPUT

Output Clear

```
/tmp/nWAXzbzJWN.o
Sorted array:
11 12 22 25 64
```

Algorithm for Bubble Sort:-

1. begin BubbleSort(arr)
2. for all array elements
3. if arr[i] > arr[i+1]
4. swap(arr[i], arr[i+1])
5. end if
6. end for
7. return arr
8. end BubbleSort

PROGRAM:

```
#include <stdio.h>
void swap(int* xp, int* yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
        int i, j;
        for (i = 0; i < n - 1; i++)

                // Last i elements are already in place
                for (j = 0; j < n - i - 1; j++)
                        if (arr[j] > arr[j + 1])
                                swap(&arr[j], &arr[j + 1]);
}
/* Function to print an array */
void printArray(int arr[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

// Driver program to test above functions
int main()
{
        int arr[] = { 5, 1, 4, 2, 8 };
        int n = sizeof(arr) / sizeof(arr[0]);
        bubbleSort(arr, n);
        printf("Sorted array: \n");
```

```
    printArray(arr, n);
    return 0;
}
```

OUTPUT

```
Output                                   Clear
/tmp/nWAXzbzJWN.o
Sorted array:
1 2 4 5 8
```

Algorithm for Merge Sort:-

MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT

PROGRAM:

```
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high) {
  int l1, l2, i;
  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
    if(a[l1] <= a[l2])
      b[i] = a[l1++];
    else
      b[i] = a[l2++];
  }
  while(l1 <= mid)
    b[i++] = a[l1++];
  while(l2 <= high)
    b[i++] = a[l2++];
```

```
  for(i = low; i <= high; i++)
    a[i] = b[i];
}
void sort(int low, int high) {
  int mid;
  if(low < high) {
    mid = (low + high) / 2;
    sort(low, mid);
    sort(mid+1, high);
    merging(low, mid, high);
  } else {
    return;
  }
}
int main() {
  int i;
  printf("List before sorting\n");
  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);

  sort(0, max);
  printf("\nList after sorting\n");
  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);
}
```

OUTPUT

```
Output                                                    Clear
/tmp/nWAXzbzJWN.o
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44
```

Algorithm for Quick Sort:-

```
QUICKSORT (array A, start, end)
{
 if (start < end)
  {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
```

Quick sort Pivot algorithm:-

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

PROGRAM:

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b)
{
        int t = *a;
        *a = *b;
        *b = t;
}
// Partition the array using the last element as the pivot
int partition(int arr[], int low, int high)
{
        // Choosing the pivot
        int pivot = arr[high];
        // Index of smaller element and indicates
        // the right position of pivot found so far
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
                // If current element is smaller than the pivot
                if (arr[j] < pivot) {
                        // Increment index of smaller element
                        i++;
                        swap(&arr[i], &arr[j]);
                }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
}
// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
void quickSort(int arr[], int low, int high)
{
        if (low < high) {
```

```
        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);
        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Driver code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    quickSort(arr, 0, N - 1);
    printf("Sorted array: \n");
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT

```
Output                                                    Clear
/tmp/nWAXzbzJWN.o
Sorted array:
1 5 7 8 9 10
```

QUIZ WITH ANSWERS:

Q1. What are the applications of selection sort?

Ans: Selection of Small or Large Elements: Selection sort is useful when you need to find the smallest or largest elements in an array rather than sorting the entire array. By performing a partial sorting operation, you can easily select the desired elements.

Q2. What are the time complexities of best case of all the sorts?

Ans: The best-case time complexity of various sorting algorithms is as follows:

1. Insertion Sort: Best case time complexity is O(n), where 'n' is the number of elements in the array. This occurs when the input array is already sorted, and each element only needs to be compared to its adjacent element once.
2. Selection Sort: Best case time complexity is O(n^2), where 'n' is the number of elements in the array. This is because selection sort requires scanning the entire unsorted part of the array in each iteration, regardless of the initial order.

3. Bubble Sort: Best case time complexity is O(n), where 'n' is the number of elements in the array. This occurs when the input array is already sorted, and no swaps are needed during the passes.
4. Merge Sort: Best case time complexity is O(n log n), where 'n' is the number of elements in the array. Merge sort always divides the array into two halves and then merges them. The best case occurs when the input array is already sorted, and no merging is needed.
5. Quick Sort: Best case time complexity is O(n log n), where 'n' is the number of elements in the array. The best case occurs when the pivot chosen in each partitioning step always divides the array into two equal-sized parts. This leads to balanced partitions and faster sorting.

Q3. What are the time complexities of worst case of all the sorts?

Ans: The worst-case time complexity of various sorting algorithms is as follows:

1. Insertion Sort: Worst case time complexity is O(n^2), where 'n' is the number of elements in the array. This occurs when the input array is sorted in reverse order, and each element needs to be compared and shifted to its correct position in every pass.
2. Selection Sort: Worst case time complexity is O(n^2), where 'n' is the number of elements in the array. This happens when the input array is sorted in reverse order or contains a significant number of inversions, causing the algorithm to perform a large number of comparisons and swaps.
3. Bubble Sort: Worst case time complexity is O(n^2), where 'n' is the number of elements in the array. This occurs when the input array is sorted in reverse order, and each pass requires swapping adjacent elements until the largest element "bubbles" to the end.
4. Merge Sort: Worst case time complexity is O(n log n), where 'n' is the number of elements in the array. Merge sort consistently achieves this time complexity regardless of the input order. It divides the array into halves and performs merging operations until the entire array is sorted.
5. Quick Sort: Worst case time complexity is O(n^2), where 'n' is the number of elements in the array. This happens when the pivot chosen in each partitioning step consistently divides the array into highly imbalanced partitions, such as when the input array is already sorted or nearly sorted. However, with proper pivot selection techniques like choosing the median or random elements, the average and best-case time complexity of quicksort is O(n log n).

Q4. What is the space complexity of all the sorts?

Ans: The space complexity of various sorting algorithms is as follows:

1. Insertion Sort: Insertion sort has a space complexity of O(1) because it performs sorting in place, i.e., it does not require additional memory proportional to the input size. It swaps elements within the given array, without requiring any auxiliary data structures.
2. Selection Sort: Similar to insertion sort, selection sort also has a space complexity of O(1). It performs sorting in place by repeatedly selecting the minimum element and swapping it with the current position.
3. Bubble Sort: Bubble sort, like insertion sort and selection sort, has a space complexity of O(1). It operates directly on the input array and does not require extra memory.
4. Merge Sort: Merge sort has a space complexity of O(n) because it requires additional memory to store temporary arrays during the merging process. In each recursive call, merge sort creates temporary arrays to merge the sorted halves. The size of the temporary arrays is equal to the input array's size.
5. Quick Sort: The space complexity of quick sort varies depending on the implementation. The traditional recursive implementation has a space complexity of O(log n) due to the recursive call stack. However, this can be mitigated by using an optimized version of quicksort called "tail recursion elimination," which reduces the space complexity to O(log n) as well.

Q5. Which is the best sort?

Ans: The choice of the "best" sorting algorithm depends on several factors, including the characteristics of the data to be sorted, the size of the data set, the desired time complexity, and the available resources. Different sorting algorithms have different strengths and weaknesses. Here are a few commonly used sorting algorithms and their notable characteristics:

1. Quicksort: Quicksort is often considered one of the best general-purpose sorting algorithms. It has an average-case time complexity of O(n log n) and can achieve good performance in practice. Quicksort is efficient for large data sets, offers good average-case performance, and is often the algorithm of choice for sorting in-memory data.
2. Merge Sort: Merge sort also has an average-case time complexity of O(n log n). It is a stable sorting algorithm that performs well on large data sets. Merge sort is suitable for situations where the entire data set cannot fit into memory because it performs sorting using an external memory or disk.

LAB EXPERIMENT 9

OBJECTIVE:

WAP that implement stack using array and linked list.

PRE-EXPERIMENT QUESTIONS:

1. How to solve the problem of tower of Hanoi?
2. What do you mean by linked list?

BRIEF DISCUSSION AND EXPLANATION:

Push(value):-

This operation is used to insert values into the stack. It inserts elements on top of the stack.

Algorithm for Push operation using array:-

1. Check if the top is equal to the size of the array. If true, print "Stack is Full" and return.
2. If false, increment the top by one.
3. Assign an item to the top of the stack.

Algorithm for Push operation using linked list:-

Step 1 - Create a newNode with given value.
Step 2 - Check whether stack is Empty (top == NULL)
Step 3 - If it is Empty, then set newNode → next = NULL.
Step 4 - If it is Not Empty, then set newNode → next = top.
Step 5 - Finally, set top = newNode.

Pop(value):-

This operation is used to remove an element from the stack. It uses the LIFO property to remove elements from the stack, i.e., the element inserted last will be removed first from the stack. It returns the removed element from the stack.

Algorithm for pop operation using array:-

1. Check if the top is equal to -1. If true, then print "Stack is Empty" and return.
2. If false, then store the top item in a variable.
3. Decrement the top.
4. Return the stored item through the variable.

Algorithm for Push operation using linked list:-

Step 1 - Check whether stack is Empty (top == NULL).
Step 2 - If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
Step 3 - If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
Step 4 - Then set 'top = top → next'.
Step 5 - Finally, delete 'temp'. (free(temp)).

PROGRAM:

(A) Using Array

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
```

```
          {
             printf("\n\t EXIT POINT ");
             break;
          }
          default:
          {
             printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
          }

       }
    }
    while(choice!=4);
    return 0;
}
void push()
{
    if(top>=n-1)
    {
       printf("\n\tSTACK is over flow");

    }
    else
    {
       printf(" Enter a value to be pushed:");
       scanf("%d",&x);
       top++;
       stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
    {
       printf("\n\t Stack is under flow");
    }
    else
    {
       printf("\n\t The popped elements is %d",stack[top]);
       top--;
    }
}
void display()
{
    if(top>=0)
    {
       printf("\n The elements in STACK \n");
       for(i=top; i>=0; i--)
          printf("\n%d",stack[i]);
       printf("\n Press Next Choice");
```

```
    }
    else
    {
        printf("\n The STACK is empty");
    }

}
```

(B) Using Linked List

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

// Structure to create a node with data and the next pointer
struct node {
    int info;
    struct node *ptr;
}*top,*top1,*temp;

int count = 0;
// Push() operation on a  stack
void push(int data) {
     if (top == NULL)
    {
        top =(struct node *)malloc(1*sizeof(struct node));
        top->ptr = NULL;
        top->info = data;
    }
    else
    {
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
    }
    count++;
    printf("Node is Inserted\n\n");
}

int pop() {
     top1 = top;

    if (top1 == NULL)
    {
        printf("\nStack Underflow\n");
        return -1;
    }
    else
        top1 = top1->ptr;
    int popped = top->info;
```

```
    free(top);
    top = top1;
    count--;
    return popped;
}

void display() {
    // Display the elements of the stack
    top1 = top;

    if (top1 == NULL)
    {
        printf("\nStack Underflow\n");
        return;
    }

    printf("The stack is \n");
    while (top1 != NULL)
    {
        printf("%d--->", top1->info);
        top1 = top1->ptr;
    }
    printf("NULL\n\n");

}

int main() {
    int choice, value;
    printf("\nImplementation of Stack using Linked List\n");
    while (1) {
        printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("\nEnter your choice : ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf("\nEnter the value to insert: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            printf("Popped element is :%d\n", pop());
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
            break;
        default:
```

```
            printf("\nWrong Choice\n");
        }
    }
}
```

OUTPUT

```
Output                                                    Clear
/tmp/nWAXzbzJWN.o
Enter the size of STACK[MAX=100]:5
STACK OPERATIONS USING ARRAY
    --------------------------------
     1.PUSH
     2.POP
     3.DISPLAY
     4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:2
 Enter the Choice:1
 Enter a value to be pushed:3
 Enter the Choice:2
 The popped elements is 3
 Enter the Choice:
```

```
Output                                                              Clear
/tmp/KlSndYpG1r.o
Implementation of Stack using Linked List

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1
Enter the value to insert: 2
Node is Inserted


1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1
Enter the value to insert: 6
Node is Inserted


1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 2
Popped element is :6

1. Push
2. Pop
3. Display
4. Exit

Enter your choice :
```

QUIZ WITH ANSWERS:

Q1. Why stack is called recursive data structure?
Ans: A **stack** is a **recursive** data structure, so it's:

- a stack is either empty or
- it consists of a top and the rest which is a stack by itself.

Q2. Why stack is useful?
Ans: They're very useful because they afford you constant time $O(1)$ operations when *inserting* or *removing* from the front of a data structure. One common use of a stack is in compilers, where a stack can be used to make sure that the brackets and parentheses in a code file are all balanced, i.e., have an opening and closing counterpart. Stacks are also very useful in evaluating mathematical expressions.

Q3. What are the two operations of Stack?
Ans: Push and Pop

Q4. What is the principal of stack?
Ans: LIFO

Q5. Entries in a stack are "ordered". What is the meaning of this statement?
Ans: There is a Sequential entry that is one by one

LAB EXPERIMENT 10

OBJECTIVE:

WAP that implement queue using array and linked list.

PRE-EXPERIMENT QUESTIONS:

1. What do you understand by LIFO and FIFO?
2. Explain some examples of queue used in daily life ?

BRIEF DISCUSSION AND EXPLANATION:

Enqueue:-

Insert an element from the rear end into the queue. Element is inserted into the queue after checking the overflow condition n-1==REAR to check whether the queue is full or not.

Algorithm to insert an element into queue using array:-

1. If n-1==REAR then this means the queue is already full.
2. But if REAR<n means that we can store an element in an array.
3. So increment the REAR value by 1 and then insert an element at the REAR index.

Algorithm to perform Insertion on a linked queue:-

1. Create a new node pointer.
ptr = (struct node *) malloc (sizeof(struct node));

2. Now, two conditions arise, i.e., either the queue is empty, or the queue contains at least one element.

3. If the queue is empty, then the new node added will be both front and rear, and the next pointer of front and rear will point to NULL.

4. If the queue contains at least one element, then the condition front == NULL becomes false. So, make the next pointer of rear point to new node ptr and point the rear pointer to the newly created node ptr. Hence, a new node(element) is added to the queue.

Dequeue:-

Deleting an element from the FRONT end of the queue. Before deleting an element we need to

check the underflow conditionfront == - 1 or front > rear to check whether there is at least one element available for the deletion or not.

Algorithm to delete an element from queue using array:-

1. If front == - 1 or front > rear then no element is available to delete.
2. Else delete FRONT index element
3. If REAR==FRONT then we set -1 to both FRONT AND REAR
4. Else we increment FRONT.

Algorithm to perform Deletion on a linked queue:-

1. Check if the queue is empty or not.

2. If the queue is empty, i.e., front==NULL, so we just print 'underflow' on the screen and exit.

3. If the queue is not empty, delete the element at which the front pointer is pointing. For deleting a node, copy the node which is pointed by the front pointer into the pointer ptr and make the front pointer point to the front's next node and free the node pointed by the node ptr.

PROGRAM:

(A) Using Array

```
#include<stdio.h>
#define n 5
int main()
{
    int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
    printf("Queue using Array");
    printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
    while(ch)
    {
        printf("\nEnter the Choice:");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:
            if(rear==x)
                printf("\n Queue is Full");
            else
            {
                printf("\n Enter no %d:",j++);
                scanf("%d",&queue[rear++]);
            }
            break;
        case 2:
```

```
            if(front==rear)
            {
               printf("\n Queue is empty");
            }
            else
            {
               printf("\n Deleted Element is %d",queue[front++]);
               x++;
            }
            break;
         case 3:
            printf("\nQueue Elements are:\n ");
            if(front==rear)
               printf("\n Queue is Empty");
            else
            {
               for(i=front; i<rear; i++)
               {
                  printf("%d",queue[i]);
                  printf("\n");
               }
               break;
            case 4:
               exit(0);
            default:
               printf("Wrong Choice: please see the options");
            }
         }
      }
   return 0;
}
```

(B) Using Linked List

```
// C program for array implementation of queue
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
// A structure to represent a queue
struct Queue {
            int front, rear, size;
            unsigned capacity;
            int* array;
};
// function to create a queue
// of given capacity.
// It initializes size of queue as 0
struct Queue* createQueue(unsigned capacity)
{
            struct Queue* queue = (struct Queue*)malloc(
```

```
                sizeof(struct Queue));
        queue->capacity = capacity;
        queue->front = queue->size = 0;

        // This is important, see the enqueue
        queue->rear = capacity - 1;
        queue->array = (int*)malloc(
                queue->capacity * sizeof(int));
        return queue;
}

// Queue is full when size becomes
// equal to the capacity
int isFull(struct Queue* queue)
{
        return (queue->size == queue->capacity);
}

// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{
        return (queue->size == 0);
}

// Function to add an item to the queue.
// It changes rear and size
void enqueue(struct Queue* queue, int item)
{
        if (isFull(queue))
                return;
        queue->rear = (queue->rear + 1)
                                % queue->capacity;
        queue->array[queue->rear] = item;
        queue->size = queue->size + 1;
        printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue.
// It changes front and size
int dequeue(struct Queue* queue)
{
        if (isEmpty(queue))
                return INT_MIN;
        int item = queue->array[queue->front];
        queue->front = (queue->front + 1)
                                % queue->capacity;
        queue->size = queue->size - 1;
        return item;
}
```

```
// Function to get front of queue
int front(struct Queue* queue)
{
        if (isEmpty(queue))
                return INT_MIN;
        return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue)
{
        if (isEmpty(queue))
                return INT_MIN;
        return queue->array[queue->rear];
}

// Driver program to test above functions./
int main()
{
        struct Queue* queue = createQueue(1000);
        enqueue(queue, 10);
        enqueue(queue, 20);
        enqueue(queue, 30);
        enqueue(queue, 40);
        printf("%d dequeued from queue\n\n",
                dequeue(queue));
        printf("Front item is %d\n", front(queue));
        printf("Rear item is %d\n", rear(queue));
        return 0;
}
```

OUTPUT

```
Output                                                    Clear
/tmp/KlSndYpG1r.o
Queue using Array
1.Insertion
2.Deletion
3.Display
4.Exit
Enter the Choice:1
Enter no 1:1
Enter the Choice:1
Enter no 2:3
Enter the Choice:1
Enter no 3:5
Enter the Choice:2
Deleted Element is 1
Enter the Choice:
```

```
Output                                                    Clear
/tmp/KlSndYpG1r.o
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue

Front item is 20
Rear item is 40
```

QUIZ WITH ANSWERS:

Q1. What is the principle of Queue?
Ans: FIFO

Q2. What are the types of queue?
Ans: Simple queue (ordinary queue)
Circular queue

Double ended queue
Priority queue

Q3. What is a queue?
Ans: A Queue is an ordered collection of items from which items may be deleted at one end called the front of the queue and into which terms may be inserted at the other end called rear of the queue. Queue is called as First –in-First-Out(FIFO).

Q4. List out Applications of queue
Ans: Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
Computer systems must often provide a “h processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.

Q5. How do you test for an empty queue?
Ans: To test for an empty queue, we have to check whether READ=HEAD where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that pointer to the dummy header.
In the case of array implementation of queue, the condition to be checked for an empty queue is READ<FRONT.

LAB EXPERIMENT 11

OBJECTIVE:

WAP to create a linked list & perform operations such as insert, delete, update and reverse in the linked list.

PRE-EXPERIMENT QUESTIONS:

1. Differentiate array and linked list?
2. Explain different types of linked list?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm for creating a linked list:-

1. Create a temporary node(temp) and assign the head node's address.
2. Print the data which present in the temp node.
3. After printing the data, move the temp pointer to the next node.
4. Do the above process until we reach the end.

Algorithm for Insertion into a linked list:-

Step 1: IF PTR = NULL.
Step 2: SET NEW_NODE = PTR.
Step 3: SET PTR = PTR → NEXT.
Step 4: SET NEW_NODE → DATA = VAL.
Step 5: SET NEW_NODE → NEXT = HEAD.
Step 6: SET HEAD = NEW_NODE.
Step 7: EXIT.

Algorithm for Deletion into a linked list:-

1. Delete from beginning. Point head to the second node head = head->next;
2. Delete from end. Traverse to second last element. Change its next pointer to null struct node* temp = head; while(temp->next->next!=NULL){ temp = temp->next; } temp->next = NULL;
3. Delete from middle. Traverse to element before the element to be deleted. Change next pointers to exclude the node from the chain

Search an Element on a Linked List:-

You can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

1. Make head as the current node.
2. Run a loop until the current node is NULL because the last element points to NULL.
3. In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.

PROGRAM:

(A) Creation

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

// Creating a node
struct node {
  int value;
  struct node *next;
};

// print the linked list value
void printLinkedlist(struct node *p) {
  while (p != NULL) {
    printf("%d ", p->value);
    p = p->next;
  }
}

int main() {
  // Initialize nodes
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;
```

```
  // Allocate memory
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));
  // Assign value values
  one->value = 1;
  two->value = 2;
  three->value = 3;

  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = NULL;
  // printing node-value
  head = one;
  printLinkedlist(head);
}
```

(B) Insert

```
// A complete working C program to demonstrate all insertion methods
// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
int data;
struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and
an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
	/* 1. allocate node */
	struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

	/* 2. put in the data */
	new_node->data = new_data;

	/* 3. Make next of new node as head */
	new_node->next = (*head_ref);

	/* 4. move the head to point to the new node */
	(*head_ref) = new_node;
}

/* Given a node prev_node, insert a new node after the given
prev_node */
```

```
void insertAfter(struct Node* prev_node, int new_data)
{
        /*1. check if the given prev_node is NULL */
        if (prev_node == NULL)
        {
        printf("the given previous node cannot be NULL");
        return;
        }

        /* 2. allocate new node */
        struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

        /* 3. put in the data */
        new_node->data = new_data;

        /* 4. Make next of new node as next of prev_node */
        new_node->next = prev_node->next;

        /* 5. move the next of prev_node as new_node */
        prev_node->next = new_node;
}

/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
        /* 1. allocate node */
        struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

        struct Node *last = *head_ref; /* used in step 5*/

        /* 2. put in the data */
        new_node->data = new_data;

        /* 3. This new node is going to be the last node, so make next of
                it as NULL*/
        new_node->next = NULL;

        /* 4. If the Linked List is empty, then make the new node as head */
        if (*head_ref == NULL)
        {
        *head_ref = new_node;
        return;
        }

        /* 5. Else traverse till the last node */
        while (last->next != NULL)
                last = last->next;
```

```
        /* 6. Change the next of last node */
        last->next = new_node;
        return;
}

// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
while (node != NULL)
{
        printf(" %d ", node->data);
        node = node->next;
}
}

/* Driver program to test above functions*/
int main()
{
/* Start with the empty list */
struct Node* head = NULL;

// Insert 6. So linked list becomes 6->NULL
append(&head, 6);

// Insert 7 at the beginning. So linked list becomes 7->6->NULL
push(&head, 7);

// Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
push(&head, 1);

// Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
append(&head, 4);

// Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
insertAfter(head->next, 8);

printf("\n Created Linked list is: ");
printList(head);

return 0;
}
```

(C) Deletion

```
// C code to delete a node from linked list
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
        int number;
        struct Node* next;
```

```
} Node;

void Push(Node** head, int A)
{
        Node* n = malloc(sizeof(Node));
        n->number = A;
        n->next = *head;
        *head = n;
}

void deleteN(Node** head, int position)
{
        Node* temp;
        Node* prev;
        temp = *head;
        prev = *head;
        for (int i = 0; i < position; i++) {
                if (i == 0 && position == 1) {
                        *head = (*head)->next;
                        free(temp);
                }
                else {
                        if (i == position - 1 && temp) {
                                prev->next = temp->next;
                                free(temp);
                        }
                        else {
                                prev = temp;

                                // Position was greater than
                                // number of nodes in the list
                                if (prev == NULL)
                                        break;
                                temp = temp->next;
                        }
                }
        }
}

void printList(Node* head)
{
        while (head) {
                printf("[%i] [%p]->%p\n", head->number, head,
                        head->next);
                head = head->next;
        }
        printf("\n\n");
}
```

```
// Drivers code
int main()
{
        Node* list = malloc(sizeof(Node));
        list->next = NULL;
        Push(&list, 1);
        Push(&list, 2);
        Push(&list, 3);

        printList(list);

        // Delete any position from list
        deleteN(&list, 1);
        printList(list);
        return 0;
}
```

(D) Inversion

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

/* Link list node */
struct Node
{
        int data;
        struct Node* next;
};

/* Function to reverse the linked list */
static void reverse(struct Node** head_ref)
{
        struct Node* prev = NULL;
        struct Node* current = *head_ref;
        struct Node* next;
        while (current != NULL)
        {
                next = current->next;
                current->next = prev;
                prev = current;
                current = next;
        }
        *head_ref = prev;
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
        /* allocate node */
        struct Node* new_node =
                        (struct Node*) malloc(sizeof(struct Node));
```

```
        /* put in the data */
        new_node->data = new_data;

        /* link the old list of the new node */
        new_node->next = (*head_ref);

        /* move the head to point to the new node */
        (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct Node *head)
{
        struct Node *temp = head;
        while(temp != NULL)
        {
                printf("%d ", temp->data);
                temp = temp->next;
        }
}

/* Driver program to test above function*/
int main()
{
        /* Start with the empty list */
        struct Node* head = NULL;

        push(&head, 20);
        push(&head, 4);
        push(&head, 15);
        push(&head, 85);

        printf("Given linked list\n");
        printList(head);
        reverse(&head);
        printf("\nReversed Linked list \n");
        printList(head);
        getchar();
}
```

OUTPUT

Output Clear

```
/tmp/KlSndYpG1r.o
1 2 3
```

Output Clear

```
/tmp/KlSndYpG1r.o
Created Linked list is:  1  7  8  6  4
```

Output Clear

```
/tmp/KlSndYpG1r.o
[3] [0x1a00300]->0x1a002e0
[2] [0x1a002e0]->0x1a002c0
[1] [0x1a002c0]->0x1a002a0
[0] [0x1a002a0]->(nil)


[2] [0x1a002e0]->0x1a002c0
[1] [0x1a002c0]->0x1a002a0
[0] [0x1a002a0]->(nil)
```

Output Clear

```
/tmp/KlSndYpG1r.o
Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85
```

QUIZ WITH ANSWERS:

Q1. What does the dummy header in the linked list contain?
Ans: In a linked list, the dummy header consists of the first record of the actual data.

Q2. Define different types of linked list.
Ans: There are multiple types of Linked Lists available:
Singly Linked List
Doubly Linked List
Multiply Linked List
Circular Linked List

Q3. What is linked list?
Ans: A linked list may be defined as a linear data structure which can store a collection of items. In another way, the linked list can be utilized to store various objects of similar types. Each element or unit of the list is indicated as a node. Each node contains its data and the address of the next node. It is similar to a chain. Linked lists are used to create graphs and trees.

Q4. How can you represent a linked list node?
Ans: The simplest method to represent a linked list node is wrapping the data and the link using **typedef structure**. Then further assigning the structure as a Node pointer that points to the next node.

Q5. Which type of memory allocation is referred for Linked List?
Ans: Dynamic memory allocation is referred for Linked List.

LAB EXPERIMENT 12

OBJECTIVE:

WAP to perform the following operations:-

#Insert an element into a binary search tree.
#Delete an element from a binary search tree.
#Search for a key element in a binary search tree.

PRE-EXPERIMENT QUESTIONS:

1. What do you understand by tree and graph?
2. Explain complexity of binary search tree?

BRIEF DISCUSSION AND EXPLANATION:

Algorithm to insert an element into a binary search tree:-

Insert (TREE, ITEM)

Step 1: IF TREE = NULL. Allocate memory for TREE. SET TREE -> DATA = ITEM. SET TREE -> LEFT = TREE -> RIGHT = NULL. ELSE. IF ITEM < TREE -> DATA. Insert(TREE -> LEFT, ITEM) ELSE. Insert(TREE -> RIGHT, ITEM) [END OF IF] [END OF IF]

Step 2: END.

Algorithm to search an element in Binary search tree

- Search (root, item)
- Step 1 - if (item = root → data) or (root = NULL)
- return root.
- else if (item < root → data)
- return Search(root → left, item)
- else.
- return Search(root → right, item)
- END if.

Algorithm to delete an element from a binary search tree:-

Delete (TREE, ITEM)

```
Step 1: IF TREE = NULL
   Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
   Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
   Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
   SET TEMP = TREE
   IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
   SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
    SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
[END OF IF]
Step 2: END
```

PROGRAM:

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

struct node {
  int data; //node will store some data
  struct node *right_child; // right child
  struct node *left_child; // left child
};

//function to create a node
struct node* new_node(int x) {
  struct node *temp;
  temp = malloc(sizeof(struct node));
  temp -> data = x;
```

```
  temp -> left_child = NULL;
  temp -> right_child = NULL;

  return temp;
}

// searching operation
struct node* search(struct node * root, int x) {
  if (root == NULL || root -> data == x) //if root->data is x then the element is found
    return root;
  else if (x > root -> data) // x is greater, so we will search the right subtree
    return search(root -> right_child, x);
  else //x is smaller than the data, so we will search the left subtree
    return search(root -> left_child, x);
}
// insertion
struct node* insert(struct node * root, int x) {
  //searching for the place to insert
  if (root == NULL)
    return new_node(x);
  else if (x > root -> data) // x is greater. Should be inserted to the right
    root -> right_child = insert(root -> right_child, x);
  else // x is smaller and should be inserted to left
    root -> left_child = insert(root -> left_child, x);
  return root;
}

//function to find the minimum value in a node
struct node* find_minimum(struct node * root) {
  if (root == NULL)
    return NULL;
  else if (root -> left_child != NULL) // node with minimum value will have no left child
    return find_minimum(root -> left_child); // left most element will be minimum
  return root;
}

// deletion
struct node* delete(struct node * root, int x) {
  //searching for the item to be deleted
  if (root == NULL)
    return NULL;
  if (x > root -> data)
    root -> right_child = delete(root -> right_child, x);
  else if (x < root -> data)
    root -> left_child = delete(root -> left_child, x);
  else {
    //No Child node
    if (root -> left_child == NULL && root -> right_child == NULL) {
```

```
      free(root);
      return NULL;
    }

    //One Child node
    else if (root -> left_child == NULL || root -> right_child == NULL) {
      struct node *temp;
      if (root -> left_child == NULL)
        temp = root -> right_child;
      else
        temp = root -> left_child;
      free(root);
      return temp;
    }

    //Two Children
    else {
      struct node *temp = find_minimum(root -> right_child);
      root -> data = temp -> data;
      root -> right_child = delete(root -> right_child, temp -> data);
    }
  }
  return root;
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) // checking if the root is not null
  {
    inorder(root -> left_child); // traversing left child
    printf(" %d ", root -> data); // printing data at root
    inorder(root -> right_child); // traversing right child
  }
}
int main() {
  struct node *root;
  root = new_node(20);
  insert(root, 5);
  insert(root, 1);
  insert(root, 15);
  insert(root, 9);
  insert(root, 7);
  insert(root, 12);
  insert(root, 30);
  insert(root, 25);
  insert(root, 40);
  insert(root, 45);
  insert(root, 42);
```

```
  inorder(root);
  printf("\n");
  root = delete(root, 1);
  root = delete(root, 40);
  root = delete(root, 45);
  root = delete(root, 9);
  inorder(root);
  printf("\n");
  return 0;
}
```

OUTPUT

```
Output                                                    Clear

/tmp/KlSndYpG1r.o
1   5   7   9   12   15   20   25   30   40   42   45
 5   7   12   15   20   25   30   42
```

QUIZ WITH ANSWERS:

Q1. What is a leaf node?
Ans: Any node in a binary tree or a tree that does not have any children is called a leaf node.

Q2. What is a root node?
Ans: The first node or the top node in a tree is called the root node.

Q3. What is a binary search tree?
Ans: A binary search tree (BST) is a special type of binary tree in which each internal node contains a key. For a binary search tree, the rule is:
a) A node can have a key that is greater than all the keys in the node's left subtree.
b) A node can have a key that is smaller than all the keys in the node's right subtree.

Q4. How trees are different from graph?
Ans: Trees and graphs are both data structures used to represent relationships between objects or entities. While they share some similarities, they also have distinct characteristics that set them apart. Here are some key differences between trees and graphs:
Structure: A tree is a special type of graph that is acyclic (does not contain cycles or loops), connected (every pair of nodes is connected by exactly one path), and has a unique root node. In other words, a tree has a hierarchical structure with a single starting point. On the other hand, a graph can be cyclic (may contain cycles) and may not have a unique root node. Graphs can have multiple disconnected components or subgraphs.
Directionality: Trees are typically rooted, meaning they have a defined directionality. Each edge in a tree points from a parent node to its child node, forming a directed acyclic graph (DAG). In contrast, graphs can have both directed edges (where the relationship between nodes has a specific direction) and undirected edges (where the relationship is bidirectional).

Q5. Which data structure is used to implement tree?
Ans: Linked List

This lab manual has been updated by

Prof. Sonika Lakra
(sonika.lakra@ggnindia.dronacharya.info)

Crosschecked By
HOD Applied Sciences and Humanities

Please spare some time to provide your valuable feedback.