



**ANNASAHEB DANGE COLLEGE OF ENGINEERING AND
TECHNOLOGY ASHTA**

Artificial Intelligence and Data Science

DATA STRUCTURES LAB MANUAL

Subject Code: 1ADPC202

A.Y. 2024-25

Course Coordinator

Dr. Asma Shaikh

Professor, AI & DS

Artificial Intelligence and Data Science Department

Vision

To produce exclusive software professionals who shall effectively contribute to the leveraging field of Artificial Intelligence and Data Science.

Mission

We will achieve our Vision by:

- Providing Excellent Infrastructure facilitating the students and faculty members with recent trends and technologies.
- Imparting High-Quality Education to the students also instigating them with ethical and moral values.
- Enabling students to enhance their research abilities to address various society-oriented issues through Innovative projects
- Collaborating with various Industries to make students industry ready

PO Programme Outcomes

Learners / Students of Artificial Intelligence and Data Science Engineering Programme
Graduates are expected to have attained & will be able to:

- 01. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- 02. Problem analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences
- 03. 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.
- 04. 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.
- 05. 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.
- 06. The engineer and society:** Demonstrate understanding of contemporary knowledge of engineering to assess societal, health, safety, legal and cultural issues and the consequent responsibilities.
- 07. 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.
- 08. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- 09. 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, write effective reports, 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 manage projects and in multidisciplinary environments.
- 12. Life-long learning:** Recognize the need for, and have the ability to engage in independent and life-long learning in the broadest context of technological change.

Sant Dnyaneshwer Shikshan Sanstha's
ANNASAHEB DANGE COLLEGE OF ENGINEERING AND TECHNOLOGY, ASHTA
(An Autonomous Institute affiliated to Shivaji University, Kolhapur)

Artificial Intelligence and Data Science Department

PEO Program Educational Objectives

PEO 1: Ability to understand, apply, analyse, design models and applications for all the real-world scenarios related to Artificial Intelligence and Data Science. (PO 1,2,3)

PEO 2: Practice engineering in a broader aspect and exhibit professional leadership qualities in their field. (PO 4,5,6)

PEO 3: Enhancing technological competence to withstand the challenges in the volatile IT industry. (PO 7,8,9)

PEO 4: To be committed in Life-long research and Learning activities that supports societal development. (PO 10,11,12)

PSO Program Specific Outcomes:

PSO 1: - Practically Applying the skills & knowledge acquired to various Inter/Multi/Trans disciplinary problem areas. (PEO 1,2)

PSO 2: - Enrich Leading abilities in the field of Artificial Intelligence and Data Science to qualify for employability. (PEO 3,4)

LAB INSTRUCTIONS

Do

1. Join the lab session few minutes before the start time.
2. Maintain proper environment of the lab.
3. Go through the theory behind the experiments before attending each lab.
4. Complete your assignment within the given period of time.

Don'ts

1. Don't use internet for solving assignment problems.
2. Don't share your codes with other students. You will get zero marks if your codes are found copied from any online resource.
3. Don't use electronic gadgets (e.g., Mobile phone, Tab, etc.) during the lab hours. You should only use one system for solving assignment problems.

LAB EQUIPMENTS

Following hardware and software are necessary to perform the experiments in data structure lab.

Programming Languages:

1. C
2. C++

Programming Editor:

1. Dev C++
2. Eclipse

Hardware:

1. A computer that can execute C/C++ programs.

Operating system:

1. Windows / UNIX Operating System.
-

LABORATORY-DATA STRUCTURE

Content

Lab	Topic of Experiment	Course Outcome
Lab I	Array Data Structure	1ADPC202-2
Lab II	Function	1ADPC202-1
Lab III	Structure	1ADPC202-1
Lab IV	Pointers	1ADPC202-1
Lab V	Singly Linked List	1ADPC202-2
Lab VI	Doubly Linked List	1ADPC202-2
Lab VII	Circular Linked List	1ADPC202-2
Lab VIII	Stack ADT – Static and Dynamic	1ADPC202-3
Lab IX	Queue ADT- Static and Dynamic	1ADPC202-3
Lab X	Stack applications , circular and doubly linked list	1ADPC202-3
Lab XI	Searching – Linear, Binary and Hashing	1ADPC202-6
Lab XII	Sorting – Bubble, Selection, Insertion	1ADPC202-6
Lab XIII	Sorting – Merge and Quick	1ADPC202-6
Lab XIV	Binary Search Tree, Traversal of trees	1ADPC202-4
Lab XV	Graph using adjacency list and traversal	1ADPC202-5

LABORATORY-I (ARRAY)

Objective:

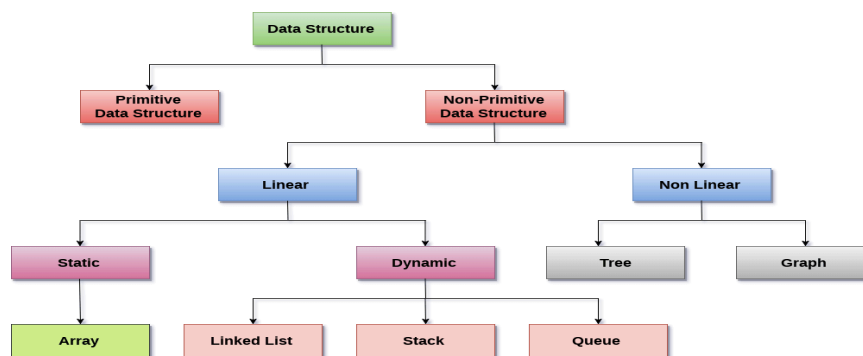
To understand one of the most basic data structures called array. This will help the students as a pre-requisite for most of the complex data structures.

Brief Theory:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e., Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Based on how they are stored in the memory data structures are classified as:



The main characteristics of data structures are as follows:

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Arrays are defined as the collection of similar types of data items, data items in array are stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

Properties of array:

- o Each element in an array is of the same data type and carries the same size.
- o Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- o Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Arrays are useful because:

- o Sorting and searching a value in an array are easier.
- o Arrays are best to process multiple values quickly and easily.
- o Arrays are good for storing multiple values in a single variable - In computer programming, most cases require storing a large numbers of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Problem statements :

Q. No	Question	CO	BL
1	Write a C program to perform the following operations on a single-dimensional array: 1. Insert an element at a specific position. 2. Delete an element from a specific index. 3. Display all elements of the array.	1ADPC202_2	K3
2	Develop a C program to create and perform the following operations on a two-dimensional array (matrix): 1. Insert an element at a specific position. 2. Delete an element from a specific index. 3. Display all elements of the array.	1ADPC202_2	K3

Observations

1. **Memory Allocation:**
 - o Arrays allocate a contiguous block of memory, which allows for efficient access to elements.
 - o Understanding how memory is managed and the implications of fixed-size arrays is crucial.
2. **Indexing:**
 - o Arrays are zero-indexed, meaning the first element is accessed with index 0.
 - o Accessing elements by index is fast and efficient ($O(1)$ time complexity).
3. **Homogeneous Elements:**
 - o All elements in an array are of the same data type, ensuring consistency.
 - o This homogeneity makes arrays simpler but less flexible than other data structures like lists or linked lists.
4. **Static vs Dynamic Arrays:**
 - o Static arrays have a fixed size, while dynamic arrays (like Python's lists) can grow or shrink.
5. **Common Operations:**
 - o Insertion, deletion, traversal, and searching are basic operations that can be performed on arrays.
 - o The time complexity of these operations can vary, with insertion and deletion often requiring shifting of elements.

Outcomes

1. The students will learn the basic idea of array data structure and how it is stored in computer memory.
2. The students will be able to use arrays for programming problems.

Conclusion

Arrays are a foundational data structure that provides a simple way to store and access elements in a sequential manner. They offer efficient indexing and are easy to implement, but their fixed size and homogeneity can be limiting. Understanding arrays is crucial for learning more advanced data structures and algorithms.

Textbooks

1. **"Data Structures and Algorithms in C" by Robert Lafore**
 1. Chapter 2: Arrays
2. **"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein**
 1. Chapter 10: Elementary Data Structures
3. **"Data Structures Using C" by Aaron M. Tenenbaum**
 1. Chapter 1: Arrays

Online Reference Websites

📖 **GeeksforGeeks - Arrays**

<https://www.geeksforgeeks.org/array-data-structure/>

📖 **TutorialsPoint - Arrays**

https://www.tutorialspoint.com/data_structures_algorithms/array_data_structure.htm

📖 **Programiz - Arrays**
<https://www.programiz.com/dsa/array>

📖 **Khan Academy - Arrays**
<https://www.khanacademy.org/computing/computer-science/algorithms#arrays>

Expected Oral Questions

1. What is an array, and how does it differ from other data structures?
2. How are elements accessed in an array?
3. Explain the concept of contiguous memory allocation in arrays.
4. What are the advantages and limitations of using arrays?
5. How would you implement a dynamic array in a language that only supports static arrays?
6. What is the time complexity of accessing an element in an array? Why?
7. Can arrays store elements of different data types? Why or why not?

FAQs in Interviews

Q: What is an array?

A: An array is a data structure that stores elements of the same data type in a contiguous block of memory, accessed by a single index.

Q: How is memory allocated for an array?

A: Memory for an array is allocated in a contiguous block, with each element occupying a fixed amount of memory.

Q: What are the time complexities for different operations in an array?

A: Access: $O(1)$, Insertion/Deletion: $O(n)$ (in the worst case, when elements need to be shifted).

Q: Can the size of an array be changed after its declaration?

A: In static arrays, no. In dynamic arrays (like vectors in C++ or ArrayLists in Java), the size can be adjusted.

Q: How would you handle array index out-of-bound errors?

A: By ensuring that any index used is within the valid range (0 to size-1) and by using error handling mechanisms where applicable.

GATE Exam Questions on Arrays

1. Question 1:

Consider the following C code:

```
#include<stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr = arr;
    printf("%d", *(ptr + 3));
    return 0;
}
```

What will be the output of the above code?

- o (A) 10
- o (B) 30
- o (C) 40
- o (D) 50

Answer: (C) 40

2. **Question 2:**

Consider the following C code snippet:

```
int arr[] = {1, 2, 3, 4, 5};  
int i, sum = 0;  
for(i = 0; i < 5; i++) {  
    sum += arr[i];  
}  
printf("%d", sum);
```

What is the output of the above code?

- ☐ (A) 10
- ☐ (B) 15
- ☐ (C) 20
- ☐ (D) 25

Answer: (B) 15

3. **Question 3:**

Consider an array A of size n. The maximum number of swaps required to sort this array using Bubble Sort algorithm in the worst case is:

- ☐ (A) n
- ☐ (B) n-1
- ☐ (C) $n*(n-1)/2$
- ☐ (D) n^2

Answer: (C) $n*(n-1)/2$

4. **Question 4:**

In a sorted array of integers, a binary search is to be performed to find an element. If the array has 1024 elements, what is the maximum number of comparisons required to find an element?

- ☐ (A) 10
- ☐ (B) 11
- ☐ (C) 512
- ☐ (D) 1024

Answer: (B) 11

5. **Question 5:**

Consider the following C code:

```
int arr[5] = {5, 4, 3, 2, 1};  
int i;  
for (i = 0; i < 5; i++) {  
    arr[i] = arr[i] + arr[4-i];  
}
```

What will be the value of arr[2] after the execution of the above code?

- ☐ (A) 3
- ☐ (B) 5
- ☐ (C) 6
- ☐ (D) 10

Answer: (C) 6

6. **Question 6:**

In an array A of n elements, if an element at position k ($0 \leq k < n$) is to be deleted, which of the following is the time complexity of the deletion operation in the worst case?

- ☐ (A) $O(1)$

- o (B) $O(k)$
- o (C) $O(n)$
- o (D) $O(\log n)$

Answer: (C) $O(n)$

LABORATORY-II (FUNCTION)

Objective:

To understand implementation the concept of functions.

Brief Theory:

A **function in C** is a set of statements that when called perform some specific task. It is the basic building block of a C program that provides modularity and code reusability. The programming statements of a function are enclosed within **{ }** **braces**, having certain meanings and performing certain operations. They are also called subroutines or procedures in other languages.

Syntax of Functions in C

The syntax of function can be divided into 3 aspects:

1. **Function Declaration**
2. **Function Definition**
3. **Function Calls**

Function Declarations

In a function declaration, we must provide the function name, its return type, and the number and type of its parameters. A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program.

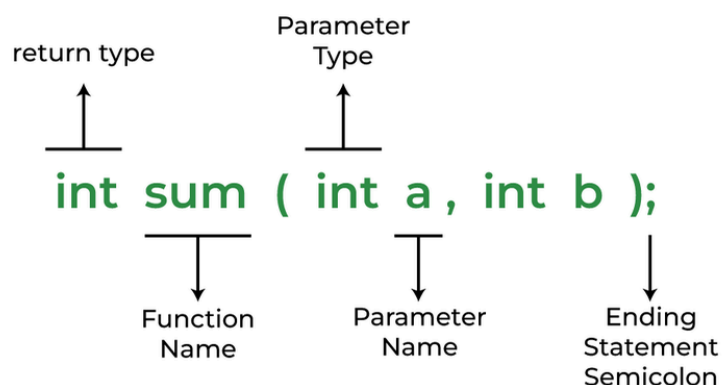
Syntax

```
return_type name_of_the_function (parameter_1, parameter_2);
```

The parameter name is not mandatory while declaring functions. We can also declare the function without using the name of the data variables.

Example

```
int sum(int a, int b); // Function declaration with parameter names  
int sum(int , int);    // Function declaration without parameter names
```



Function Declaration

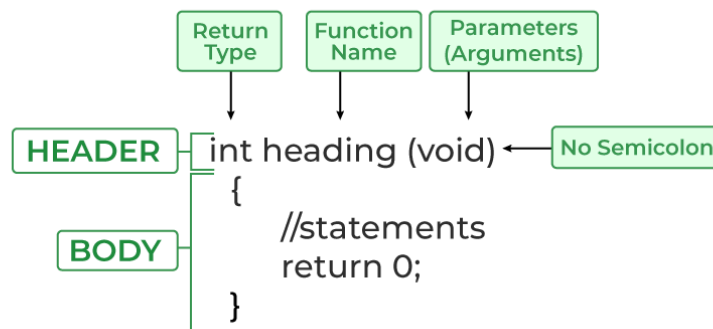
Function Definition

The function definition consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).

A C function is generally defined and declared in a single step because the function definition always starts with the function declaration so we do not need to declare it explicitly. The below example serves as both a function definition and a declaration.

```
return_type function_name (para1_type para1_name, para2_type para2_name)
{
    // body of the function
}
```

Function Definition

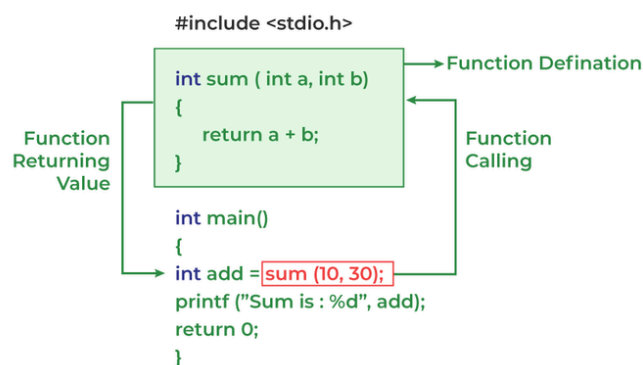


Function Definition in C

Function Call

A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call. In the below example, the first sum function is called and 10,30 are passed to the sum function. After the function call sum of a and b is returned and control is also returned back to the main function of the program.

Working of Function in C



Working of function in C

Note: Function call is neccessary to bring the program control to the function definition. If not called, the function statements will not be executed.

Questions

Q. No	Question	CO	BL
1	Basic Function Demonstration: Write a C program to create and use a function that: 1. Accepts two integers as parameters. 2. Returns their sum to the main function. 3. Displays the result in the main function.	1ADPC202_1	K3
2	Swapping Numbers Using Functions: Write a C program that swaps the values of two variables using a function. The program should: 1. Accept two numbers from the user. 2. Use a function to swap their values by passing them as pointers. 3. Display the values before and after the swap in the main function.	1ADPC202_1	K3
3	Recursive Function for Factorial: Develop a C program that calculates the factorial of a number using a recursive function. The program should: 1. Accept a number from the user. 2. Use recursion to compute the factorial. 3. Display the result in the main function.	1ADPC202_1	K3
4	Function with Array as Parameter: Write a C program that uses a function to find the largest element in an array. The program should: 1. Pass the array and its size as parameters to the function. 2. Find and return the largest element to the main function. 3. Display the result in the main function.	1ADPC202_1	K3

Observations

1. Modularity:

- o Functions allow for the modularization of code, enabling the separation of concerns and making the code easier to read, maintain, and debug.
- o Each function performs a specific task, and larger problems can be solved by combining multiple functions.

2. Reusability:

- o Once defined, a function can be reused multiple times throughout a program, reducing code duplication.
- o This also makes updates and changes easier, as the function only needs to be modified in one place.

3. Parameter Passing:

- o Functions can accept input parameters and return output values, allowing for dynamic and flexible operations.
- o Understanding the difference between pass-by-value and pass-by-reference is crucial in determining how data is manipulated within functions.

4. Scope and Lifetime:

- o Variables declared within a function have local scope, meaning they are only accessible within that function.
 - o Understanding the scope and lifetime of variables helps prevent errors and conflicts in larger programs.
5. **Recursion:**
- o Functions can call themselves, a concept known as recursion, which is particularly useful for solving problems that can be broken down into similar sub-problems.

Outcome:

1. Develop the understanding of functions and its types
2. Ability to use various function for solving problems.

Conclusion:

Functions are a fundamental building block of programming that enable modularity, reusability, and clarity in code. By breaking down complex tasks into smaller functions, students can write more organized and efficient programs. Understanding functions is essential for mastering more advanced programming concepts and techniques.

Textbooks :

1. **"The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie**
 - o Chapter 4: Functions and Program Structure
2. **"Introduction to Programming in Python" by Robert Sedgewick, Kevin Wayne, and Robert Dondero**
 - o Chapter 3: Functions and Modules
3. **"Data Structures and Algorithm Analysis in C" by Mark Allen Weiss**
 - o Chapter 2: Functions and Recursion


Online Reference Websites

 **GeeksforGeeks - Functions in C**

<https://www.geeksforgeeks.org/functions-in-c/>

 **Programiz - Functions (Python)**

<https://www.programiz.com/python-programming/function>

 **W3Schools - Python Functions**

https://www.w3schools.com/python/python_functions.asp

 **TutorialsPoint - Functions in C**

https://www.tutorialspoint.com/cprogramming/c_functions.htm

Expected Oral Questions

1. What is a function, and why is it used in programming?
2. Explain the difference between a function definition and a function call.
3. What are the different types of functions (e.g., user-defined, built-in) in the language you are using?
4. How does parameter passing work in functions? What is the difference between passing by value and passing by reference?

5. What is recursion, and can you give an example of a problem that can be solved using recursion?
6. How does the scope of variables work within functions?
7. What is the significance of the return type in a function?

FAQs in Interviews

1. **Q: What is a function in programming?**
A: A function is a block of code designed to perform a specific task, and it can be executed by calling its name. Functions help in organizing code, improving readability, and reusability.
2. **Q: How do you pass arguments to a function?**
A: Arguments are passed to a function through its parameters, which are defined in the function signature. They can be passed by value or by reference, depending on the programming language.
3. **Q: What is recursion?**
A: Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem until a base condition is met.
4. **Q: What is the difference between local and global variables?**
A: Local variables are declared within a function and are only accessible within that function, whereas global variables are declared outside any function and are accessible throughout the program.
5. **Q: Can you explain the concept of a pure function?**
A: A pure function is a function that, given the same input, always produces the same output and has no side effects (i.e., it does not modify any external state).

GATE Exam Questions on Functions

1. Question 1:

Consider the following C code:

```
#include<stdio.h>
void func(int a, int b) {
    a = a + b;
    b = a - b;
    a = a - b;
}
int main() {
    int x = 10, y = 20;
    func(x, y);
    printf("%d %d", x, y);
    return 0;
}
```

What will be the output of the above code?

- o (A) 10 20
- o (B) 20 10
- o (C) 30 10
- o (D) Undefined

Answer: (A) 10 20

2. Question 2:

What is the output of the following C program?

```
#include<stdio.h>
int fun(int x) {
    if (x == 1)
        return 1;
    else
        return x + fun(x - 1);
}
```

```

}
int main() {
    int result;
    result = fun(5);
    printf("%d", result);
    return 0;
}

```

- ☐ (A) 15
- ☐ (B) 10
- ☐ (C) 5
- ☐ (D) 20

Answer: (A) 15

3. **Question 3:**

Consider the following recursive C function:

```

int foo(int n) {
    if (n <= 0) return 1;
    if (n % 2 == 0)
        return foo(n/2);
    else
        return foo(n/2) + foo(n/2 + 1);
}

```

What is the output of foo(4)?

- ☐ (A) 1
- ☐ (B) 2
- ☐ (C) 3
- ☐ (D) 4

Answer: (B) 2

4. **Question 4:**

Which of the following statements is TRUE about functions in C?

- ☐ (A) A function can return multiple values.
- ☐ (B) A function name can be used as a variable name in the same scope.
- ☐ (C) A function can be defined inside another function.
- ☐ (D) A function can have default parameter values.

Answer: (D) A function can have default parameter values.

5. **Question 5:**

In C, what is the output of the following code?

```

#include <stdio.h>
void foo(int *ptr) {
    *ptr = *ptr + 10;
}
int main() {
    int num = 10;
    foo(&num);
    printf("%d", num);
    return 0;
}

```

- (A) 10
- ☐ (B) 20
 - ☐ (C) 30
 - ☐ (D) Compilation Error

Answer: (B) 20

LABORATORY-III (STRUCTURE)

Objective:

To create a program that defines a structure for a student class, allowing storage and manipulation of student-related data such as name, roll number, marks, and grade.

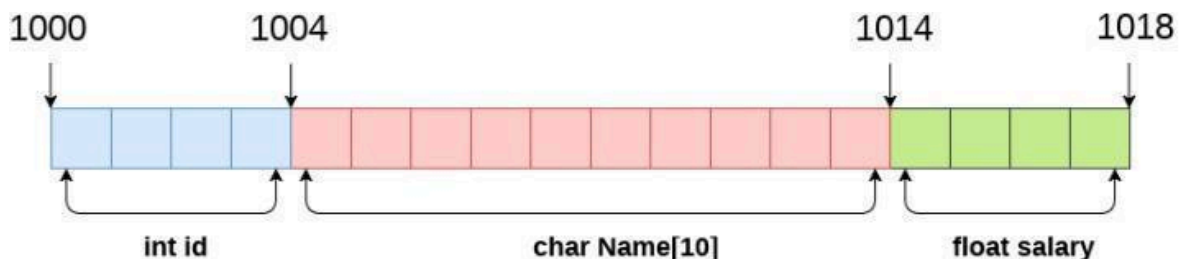
Brief Theory:

Structures: A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

Structure key word is used to create a structure.

Syntax of structure:

```
struct structure_name
{
    data_type member1;
    data_type member2;
    ...
    data_type memberN;
};
```



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte



A **structure** (often referred to as a struct in programming) is a user-defined data type available in C, C++, and some other programming languages. It allows the grouping of variables under a single name, making it easier to manage related data as one logical unit. Unlike arrays, which can hold only elements of the same data type, a structure can contain variables of different data types.

Key Concepts of Structures:

Definition:

- A structure is defined using the struct keyword. The definition includes the structure's name and a block of code that declares the variables (called members or fields) that belong to the structure.

```
struct Student {
    int rollNumber;
    char name[50];
    float marks;
};
```

In the above example, Student is a structure with three members: rollNumber (an integer), name (a character array), and marks (a floating-point number).

Declaring Structure Variables:

- o Once a structure is defined, you can declare variables of that structure type, just like you would with any other data type.

```
struct Student student1, student2;
```

Here, student1 and student2 are variables of the Student structure type.

Accessing Members:

- o Members of a structure are accessed using the dot (.) operator.

```
student1.rollNumber = 101;
strcpy(student1.name, "John Doe");
student1.marks = 85.5;
```

This example assigns values to the members of the student1 structure.

Nested Structures:

- o Structures can also contain other structures as members, allowing for complex data types.

```
struct Address {
    char street[50];
    char city[50];
    int zipCode;
};
```

```
struct Employee {
    int empID;
    char empName[50];
    struct Address empAddress;
};
```

In this example, Address is a structure that is used as a member of the Employee structure.

Pointers to Structures:

- o You can also create pointers to structures and access structure members using the arrow (->) operator.

```
struct Student *ptr;
ptr = &student1;
ptr->marks = 90.0;
```

Here, ptr is a pointer to the student1 structure, and the -> operator is used to access the marks member.

Structure as Function Arguments:

- o Structures can be passed to functions by value or by reference. Passing by reference (using pointers) is more efficient, especially for large structures.

```
void displayStudent(struct Student s) {
    printf("Roll Number: %d\n", s.rollNumber);
    printf("Name: %s\n", s.name);
    printf("Marks: %.2f\n", s.marks);
}
```

In this example, the displayStudent function takes a Student structure as an argument and prints its members.

Advantages of Using Structures:

- **Organization:** Structures help organize related data in a logical way, making the code more readable and maintainable.

- **Efficiency:** By grouping related data, structures reduce the need for multiple variables and can be more efficient in terms of memory usage.
- **Modularity:** Structures make it easier to reuse code by encapsulating related data and operations.

Common Use Cases:

- **Modeling Real-World Entities:** Structures are commonly used to model real-world entities in programs, such as a Student, Employee, or Book.
- **Linked Data Structures:** Structures form the basis of more complex data structures like linked lists, trees, and graphs.
- **Data Management:** They are used to manage data in applications like databases, where each record can be represented by a structure.

Outcome:

1. Develop the understanding of data structure and its types.
2. Ability to use various data structures for solving unseen problems.
3. Understand the use of appropriate data structure to solve real world problems

Problems on:.

Q. No	Question	CO	BL
1	Basic Structure Usage: Write a C program to create a structure Student with the following fields: roll_no, name, and marks. The program should: 1. Accept details of 5 students from the user. 2. Display the details of all students.	1ADPC202_1	K3
2	Structure and Function Integration: Write a C program that uses a structure Employee with fields id, name, and salary. The program should: 1. Accept details of employees from the user. 2. Pass the structure to a function to calculate and display the total salary of all employees.	1ADPC202_1	K3
3	Array of Structures: Develop a C program to create an array of structures for Books, where each structure contains fields title, author, and price. The program should: 1. Accept details of N books from the user. 2. Search for a book by title and display its details if found.	1ADPC202_1	K3
4	Nested Structures: Write a C program that defines a nested structure for a Company, where the Employee structure (with id and name) is a part of the Company structure. The program should: 1. Accept and display details of employees in a company. 2. Demonstrate accessing nested structure elements.	1ADPC202_1	K3

Observations

1. **Structures Group Data:**
 - o Structures allow for the grouping of different data types (e.g., integers, floats, and characters) into a single unit, making data management easier and more logical.
2. **Memory Allocation:**
 - o Each member of a structure is allocated separate memory, and the total memory occupied by the structure is the sum of the memory occupied by each member.
3. **Accessing Members:**
 - o Members of a structure are accessed using the dot (.) operator, allowing direct access to each component of the structure.
4. **Use in Real-world Applications:**
 - o Structures are used to represent complex data in real-world applications, such as student records, employee databases, and more.

Conclusion

Understanding the implementation of structures in C is crucial for efficiently handling and managing related data. This knowledge is applicable in various fields, such as database management and software development, where handling multiple attributes of an entity is required.

Textbooks

1. "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie
 - o Chapter 6: Structures
2. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss
 - o Chapter 1: Basic Data Structures
3. "Let Us C" by Yashavant Kanetkar
 - o Chapter 11: Structures

Online Reference Websites

📺 **GeeksforGeeks - Structures in C**
<https://www.geeksforgeeks.org/structures-c/>

📺 **TutorialsPoint - Structures in C**
https://www.tutorialspoint.com/cprogramming/c_structures.htm

📺 **Programiz - C Structures**
<https://www.programiz.com/c-programming/c-structures>

📺 **W3Schools - C Structures**
https://www.w3schools.com/c/c_structs.php

Expected Oral Questions

1. What is a structure in C, and why is it used?
2. How are members of a structure accessed in a C program?

3. What is the difference between an array and a structure?
4. Explain the memory allocation process for structures in C.
5. Can you nest one structure within another? If yes, how?
6. What are the limitations of using structures in C?
7. How do structures differ from classes in object-oriented programming languages?

FAQs in Interviews

1. **Q: What is a structure in C?**
A: A structure is a user-defined data type in C that allows the grouping of variables of different data types under a single name.
2. **Q: How do you initialize a structure in C?**
A: A structure can be initialized by assigning values to its members individually or using an initializer list at the time of declaration.
3. **Q: Can structures contain functions in C?**
A: No, structures in C cannot contain functions, but they can contain function pointers.
4. **Q: How is memory managed for a structure in C?**
A: Memory for a structure is allocated based on the size of its members, and each member is stored in contiguous memory locations.
5. **Q: Can you have an array of structures?**
A: Yes, you can have an array of structures, where each element of the array is a structure.

GATE Exam Questions on Structures

1. **Question 1:**
 Consider the following C code:

```
#include<stdio.h>

struct student {
    int roll_no;
    float marks;
};

int main() {
    struct student s1 = {1, 85.5};
    struct student *ptr = &s1;

    printf("%d %.1f\n", s1.roll_no, ptr->marks);
    return 0;
}
```

 What will be the output of the above code?
 - o (A) 1 85.5
 - o (B) 1 0.0
 - o (C) 0 85.5
 - o (D) Compilation Error

Answer: (A) 1 85.5

2. **Question 2:**
 Which of the following statements is TRUE regarding structures in C?
 - o (A) Structures can only contain data members of the same type.
 - o (B) Structures cannot contain pointers.
 - o (C) Structures can be passed to functions by value.
 - o (D) Structures cannot be nested within other structures.

Answer: (C) Structures can be passed to functions by value.

3. **Question 3:**

What is the size of the following structure in C?

```
struct Test {  
    int x;  
    char y;  
    double z;  
};
```

- ☐ (A) 13 bytes
- ☐ (B) 16 bytes
- ☐ (C) 24 bytes
- ☐ (D) 8 bytes

Answer: (B) 16 bytes (depending on padding and alignment in the system architecture).

4. **Question 4:**

Consider the following structure in C:

```
c  
Copy code  
struct Point {  
    int x;  
    int y;  
};
```

How do you access the y value of the structure p which is a pointer to the Point structure?

- ☐ (A) p.y
- ☐ (B) *p.y
- ☐ (C) p->y
- ☐ (D) &p->y

Answer: (C) p->y

5. **Question 5:**

Which of the following correctly defines a structure that contains an array of integers?

- ☐ (A) struct Array { int arr[10]; };
- ☐ (B) struct Array { int arr[]; };
- ☐ (C) struct Array { arr[10] int; };
- ☐ (D) struct Array { arr[10]; };

Answer: (A) struct Array { int arr[10]; };

LABORATORY-IV (Pointers)

Objectives:

- To understand the concept of memory addresses and their manipulation in C/C++.
- To learn how to use pointers for dynamic memory allocation.

Brief Theory:

Pointers are variables that store the memory address of another variable. Instead of holding a direct value, they hold the address where the value is stored. This allows for powerful programming techniques like dynamic memory allocation, passing arguments by reference, and efficient manipulation of arrays and structures.

Key Concepts:

- **Pointer Declaration:** `int *ptr;` declares a pointer to an integer.
- **Dereferencing:** `*ptr` gives the value stored at the address held by the pointer.
- **Pointer Arithmetic:** Incrementing a pointer moves it to the next memory location of the type it points to.
- **Function Pointers:** Pointers can be used to store addresses of functions and call them dynamically.
- **Null Pointers:** A special pointer that points to nothing, used for error handling.

Syntax

```
int *ptr;           // Declares a pointer to an integer
ptr = &var;         // Stores the address of variable 'var' in pointer 'ptr'
*ptr = 10;          // Assigns the value 10 to the location pointed by 'ptr'
```

Program Code Example

```
#include <stdio.h>

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;
    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap: a = %d, b = %d\n", a, b);
    return 0;
}
```

Expected Outcomes

- Ability to manipulate memory directly.
- Understanding of dynamic memory allocation using `malloc`, `calloc`, and `free`.
- Proficiency in passing data between functions by reference.
- Enhanced ability to implement complex data structures like linked lists, stacks, and queues.

Problems on Pointers.

Q. No	Question	CO	BL
1	Pointer Basics: Write a C program to demonstrate pointer operations: 1. Declare a pointer and assign it the address of an integer variable. 2. Perform read and write operations using the pointer. 3. Print the address and value of the variable using the pointer.`	1ADPC202_1	K3
2	Swapping Numbers Using Pointers: Write a C program that swaps the values of two variables using pointers. The program should: 1. Accept two numbers from the user. 2. Use a function to swap their values by passing pointers as arguments. 3. Display the values before and after the swap.	1ADPC202_1	K3
3	Pointer Arithmetic: Develop a C program that demonstrates pointer arithmetic. Perform the following: 1. Create an array and initialize it with values. 2. Modify array elements using pointer arithmetic and display the updated array.	1ADPC202_1	K3

Observations

- Pointers allow for efficient memory management and manipulation.
- Incorrect use of pointers can lead to issues like segmentation faults, memory leaks, and undefined behavior.
- Pointers add flexibility to function arguments, allowing them to modify variables in the caller's scope.

Conclusion

Pointers are a fundamental concept in C/C++ programming, providing a way to directly interact with memory. Mastery of pointers leads to a deeper understanding of how programs manage memory, offering control and efficiency in coding.

Textbooks

1. **"The C Programming Language"** by Brian W. Kernighan and Dennis M. Ritchie - This is the classic book for learning C, covering pointers in detail.
2. **"C Programming: A Modern Approach"** by K. N. King - Offers clear explanations and examples of pointers.
3. **"Data Structures Using C"** by Reema Thareja - Provides insight into data structures and their implementation using pointers.

Online Reference Websites

GeeksforGeeks - Pointers in C

<https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/>

TutorialsPoint - C Pointers

https://www.tutorialspoint.com/cprogramming/c_pointers.htm

Programiz - C Pointers

<https://www.programiz.com/c-programming/c-pointers>

Cprogramming.com - Pointers in C/C++

<http://www.cprogramming.com/tutorial/c/lesson6.html>

Guru99 - Pointers in C

<https://www.guru99.com/c-pointers.html>

Expected Oral Questions

1. What is a pointer?
2. Explain pointer arithmetic.
3. How do you pass a pointer to a function?
4. What is a null pointer?
5. What are the applications of pointers in real-world scenarios?

FAQ in Interviews

1. **Why are pointers used in C?** Pointers provide a way to access and manipulate memory directly, allowing for efficient data handling and manipulation, especially in dynamic data structures.
2. **What is the difference between ++ptr and *ptr++?** ++ptr increments the pointer itself, moving it to the next memory location, while *ptr++ increments the pointer after accessing the value it currently points to.
3. **What are dangling pointers?** Dangling pointers refer to pointers that point to a memory location that has already been freed or deallocated.

GATE Questions

1. **What will be the output of the following code?**

```
int a = 10;  
int *p = &a;  
*p = 20;  
printf("%d", a);  
(Answer: 20)
```

2. **Which of the following statements is true about pointers?**

- o A) They store the value of a variable.
- o B) They store the address of a variable.
- o C) They can be dereferenced to access the value at the stored address.
- o D) Both B and C. (Answer: D)

3. **Consider the following C code:**

```
int arr[] = {1, 2, 3, 4};  
int *ptr = arr;  
printf("%d", *(ptr + 2));
```

What is the output? (Answer: 3)

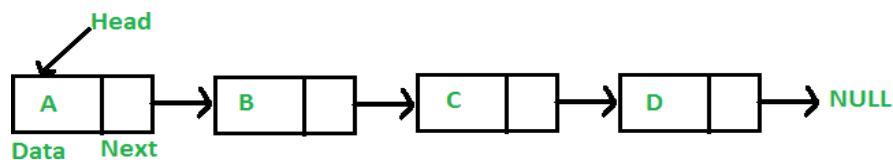
LABORATORY-V (SINGLY LINKED LIST-I)

Objectives:

To get familiar with the linked list data structure and gain the ability to utilize linked list data structure to develop other advanced data structure.

Brief Theory:

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data field and the address field. Data field is used to store the data and the address field is used to store the address of



its adjacent nodes. First node of a linked list is called head, and only the address of the first node is stored to store a linked list. All other node can be accessed from the first node itself by sequentially exploring the adjacent node. Basically, linked list forms a chain like structure. called head Linked Lists are used to create trees and graphs. Structure is used to create a node in linked list.

Types of Linked Lists:

- **Singly Linked List:** Each node points to the next node in the sequence.
- **Doubly Linked List:** Each node points to both the next and the previous node, allowing bidirectional traversal.
- **Circular Linked List:** The last node points back to the first node, forming a circle.

Key Concepts:

- **Node Structure:** Comprises data and a pointer.
- **Head Pointer:** Points to the first node of the linked list
- **Traversal:** Visiting each node sequentially to perform operations like searching or modifying data

Advantage of Linked List:

- Unlike array In Linked Lists, the size of the list need not be fixed in advance.
- Linked list is dynamic in nature, which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked lists let you insert elements at the beginning and end of the list.

Disadvantage:

- One disadvantage of linked list is random access is not possible in case of linked list, to access any element we have to search it from the head, which lead to $O(N)$ complexity of searching.
- Another disadvantage of linked list is memory wastage, each nodes waist some amount of memory to store the address of its adjacent nodes.

1. **Singly Linked List:** Singly linked lists contain nodes which have a data part as well as an address part i.e., next, which points to the next node in the sequence. In singly linked list from any particular node there is no way to reach its previous node. The operations we can perform on singly linked lists are insertion, deletion and traversal.

Observations/Outcomes:

1. To get an idea of Linked List and its types.
2. The students will be able to develop Linked List and its type from scratch.

Problems on:

1. Problem on implementation of the various operations on singly *Linked List*, like **create, display, delete**.
2. Program to create doubly linked list, and search for a particular key.
3. Inserting, deleting elements in linked list at various position.

Syntax

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

// Creating a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->next = NULL;
    return node;
}
```

Program Code Example

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

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

```

void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head;

    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    while (last->next != NULL) {
        last = last->next;
    }

    last->next = newNode;
}

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

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 4);
    printList(head);
    return 0;
}

```

Outcomes

- Ability to implement and manipulate linked lists.
- Understanding of dynamic memory management and pointers.
- Proficiency in performing basic operations on linked lists like insertion, deletion, and traversal.
- Enhanced problem-solving skills through linked list-based algorithms.

Problems on Linked Lists

Q. No	Question	CO	BL
1	Creating a Singly Linked List: Write a C program to create a singly linked list. The program should: 1. Accept values from the user to insert as nodes. 2. Display the linked list.	1ADPC202_ 2	K3
2	Insertion Operations: Write a C program to perform insertion operations in a singly linked list. The program should: 1. Insert a node at the beginning. 2. Insert a node at the end. 3. Display the updated linked list.	1ADPC202_ 2	K3
3	Deletion Operations: Write a C program to perform deletion operations in a singly linked list. The program should: 1. Delete a node from the beginning. 2. Delete a node from the end. 3. Display the updated linked list.	1ADPC202_ 2	K3
4	Searching in a Linked List: Write a C program to search for a given value in a singly linked list. The program should: 1. Accept a value from the user. 2. Traverse the linked list to find the value. 3. Display whether the value was found and its position.	1ADPC202_ 4	K3

Observations

- Linked lists provide flexibility in memory usage compared to arrays.
- They are efficient for insertion and deletion operations, especially at the beginning or middle of the list.
- Traversing linked lists can be slower compared to arrays due to the need to follow pointers.

Conclusion

Linked lists are an essential data structure in computer science, offering dynamic memory allocation and efficient manipulation of data. Understanding linked lists is crucial for implementing more complex data structures and algorithms.

Textbooks

1. **Data Structures and Algorithm Analysis in C"** by Mark Allen Weiss - Comprehensive coverage of data structures including linked lists.
2. **Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein - Covers linked lists and their applications.
3. **Data Structures Using C"** by Reema Thareja - Provides detailed explanations and examples of linked lists in C.

Online Reference Websites

1. **GeeksforGeeks - Linked List Data Structure**
<https://www.geeksforgeeks.org/data-structures/linked-list/>
2. **Tutorialspoint - C Linked Lists**
https://www.tutorialspoint.com/data_structures_algorithms/linked_list_program_in_c.htm
3. **Programiz - Linked List in C**
<https://www.programiz.com/dsa/linked-list>
4. **Studytonight - Linked Lists in Data Structure**
<https://www.studytonight.com/data-structures/linked-list>
5. **JavaTpoint - Linked List**
<https://www.javatpoint.com/data-structure-linked-list>

Expected Oral Question

- What is a linked list, and how does it differ from an array?
- Explain the different types of linked lists.
- How do you insert a new node at the beginning of a linked list?
- What are the advantages of using a doubly linked list?
- Can you explain how to detect a cycle in a linked list?

FAQ in Interviews

1. **What are the real-world applications of linked lists?** Linked lists are used in various applications like dynamic memory management, implementation of stacks and queues, and in scenarios where memory allocation is not known in advance.
2. **How do you reverse a linked list?** Reversing a linked list involves re-pointing the next pointers of each node to the previous node until the end of the list is reached.
3. **What is the time complexity for searching an element in a linked list?** The time complexity for searching an element in a linked list is $O(n)$, where n is the number of nodes in the list.
4. **Why might you choose a linked list over an array?** Linked lists are preferred over arrays when dynamic memory allocation is required, and when frequent insertion and deletion of elements are necessary.

GATE Questions

1. **What is the time complexity of inserting a new node at the end of a singly linked list?**
 1. A) $O(1)$
 2. B) $O(n)$
 3. C) $O(\log n)$
 4. D) $O(n^2)$ (Answer: B)
2. **Which of the following statements is true for a doubly linked list?**
 1. A) It requires more memory per node compared to a singly linked list.
 2. B) It allows traversal in both directions.
 3. C) It requires extra pointers to maintain.
 4. D) All of the above. (Answer: D)
3. **Consider the following C code:**

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

If you want to insert a node at the beginning, what would be the time complexity? (Answer: $O(1)$)

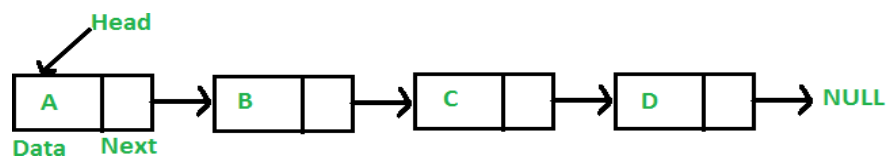
LABORATORY-6(DOUBLY LINKED LIST)

Objectives:

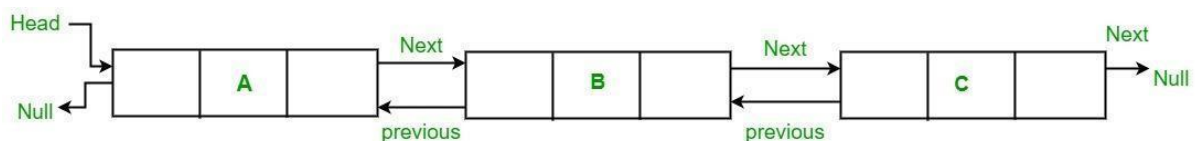
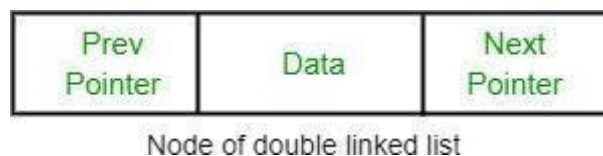
- Understand the concept and structure of doubly linked lists.
- Implement basic operations on doubly linked lists (insertion, deletion, traversal).
- Compare the advantages and disadvantages of doubly linked lists with singly linked lists.
- Solve problems using doubly linked lists.

Brief Theory:

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs



Doubly Linked List: In a doubly linked list, each node maintains two pointers namely *Prev* and *Next* which are used to store the previous and next adjacent element respectively. The first link points to the previous node and the next link points to the next node in the sequence.



Key Concepts:

- **Node Structure:** Understanding how each node in a doubly linked list is composed of data, a next pointer, and a previous pointer.
- **Head and Tail Pointers:** Head points to the first node, and the tail points to the last node in the list.
- **Traversal:** Ability to traverse the list both forward and backward.

Advantages of Doubly Linked List:

- **Bidirectional Traversal:** Can be traversed in both forward and backward directions.
- **Efficient Deletions:** Easier to delete a node when a pointer to it is provided.

- **Insertion Flexibility:** Easier to insert a node before a given node, unlike in a singly linked list.

Disadvantages of Doubly Linked List:

- **Increased Memory Usage:** Each node requires an extra pointer (to the previous node).
- **Complexity in Implementation:** More complex to implement compared to a singly linked list.
- **Slower Operations:** Operations may take slightly longer due to the need to update two pointers during insertion or deletion.

Outcomes:

- Students will be able to implement a doubly linked list and perform basic operations like insertion, deletion, and traversal.
- Understand the practical applications and limitations of doubly linked lists.

Problems on Doubly Linked Lists:

Q. No	Question	CO	BL
1	Creating a Doubly Linked List: Write a C program to create a doubly linked list. The program should: <ol style="list-style-type: none"> 1. Accept values from the user to insert as nodes. 2. Display the doubly linked list (both forward and backward). 	1ADPC202_2	K3
2	Insertion Operations in Doubly Linked List: Write a C program to perform insertion operations in a doubly linked list. The program should: <ol style="list-style-type: none"> 1. Insert a node at the beginning. 2. Insert a node at the end. 3. Insert a node after a given node. 	1ADPC202_2	K3
3	Deletion Operations in Doubly Linked List: Write a C program to perform deletion operations in a doubly linked list. The program should: <ol style="list-style-type: none"> 1. Delete a node from the beginning. 2. Delete a node from the end. 3. Delete a given node from the list. 	1ADPC202_2	K3

4	Traversing Doubly Linked List: Write a C program to traverse a doubly linked list in both forward and backward directions. The program should: <ol style="list-style-type: none"> 1. Accept values to create a doubly linked list. 2. Display the list forward and backward. 	1ADPC202_2	K3
---	---	------------	----

Syntax:

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

- **Insertion:** Example code for inserting a node at the beginning, end, or before a given node.
- **Deletion:** Example code for deleting a node from the beginning, end, or a specific position.

Expected Outcomes:

- Efficient manipulation of doubly linked lists for various use cases.
- Clear understanding of when and why to use doubly linked lists over singly linked lists.

Observations:

- Observe how the additional pointer in each node influences the memory usage.
- Note the differences in time complexity when performing various operations compared to singly linked lists.

Conclusion:

Doubly linked lists are a powerful data structure that offers flexibility in traversal and efficient manipulation of nodes at the cost of increased memory usage and implementation complexity. They are particularly useful in scenarios where bidirectional traversal is necessary.

Textbooks:

- "Data Structures Using C" by Reema Thareja
- "Fundamentals of Data Structures in C" by Horowitz, Sahni, and Anderson-Freed

Online Reference Websites:

- GeeksforGeeks: Doubly Linked List - <https://www.geeksforgeeks.org/doubly-linked-list/>
- Tutorialspoint: Doubly Linked List - https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list.htm

Expected Oral Questions:

- Explain the difference between a singly linked list and a doubly linked list.
- What are the advantages of using a doubly linked list over a singly linked list?

- How would you implement a function to delete a node in a doubly linked list?

FAQs in Interviews (with Answers):

1. **Q:** What is a doubly linked list?
 - **A:** A doubly linked list is a type of linked list where each node contains two pointers: one to the next node and one to the previous node, allowing traversal in both directions.
2. **Q:** How do you reverse a doubly linked list?
 - **A:** To reverse a doubly linked list, you need to swap the next and previous pointers for each node and then update the head and tail pointers accordingly.
3. **Q:** What are the main disadvantages of using a doubly linked list?
 - **A:** The main disadvantages are increased memory usage due to the extra pointer and added complexity in implementing the data structure.
4. **Q:** When would you prefer a doubly linked list over a singly linked list?
 - **A:** You would prefer a doubly linked list when you need efficient bidirectional traversal or need to delete nodes efficiently when a pointer to the node is provided.

Gate Questions:

1. Conceptual Questions

- **Question:** Which of the following is true about a doubly linked list?
 - A) It allows traversal in both forward and backward directions.
 - B) It requires more space per node compared to a singly linked list.
 - C) Deletion of a node is easier compared to a singly linked list.
 - D) All of the above.
- **Answer:** D) All of the above.

2. Time Complexity Questions

- **Question:** What is the time complexity of inserting a node at the end of a doubly linked list of 'n' nodes?
 - A) $O(1)$
 - B) $O(n)$
 - C) $O(\log n)$
 - D) $O(n \log n)$
- **Answer:** A) $O(1)$ (if the tail pointer is maintained).

3. Coding-Based Questions

- **Question:** Which of the following is the correct way to delete a node 'ptr' in a doubly linked list?
 - A) `ptr->prev->next = ptr->next; ptr->next->prev = ptr->prev;`
 - B) `ptr->next->prev = ptr->next; ptr->prev->next = ptr->prev;`
 - C) `ptr->prev->next = ptr; ptr->next->prev = ptr;`
 - D) `ptr->next->prev = ptr->next; ptr->prev->next = ptr;`
- **Answer:** A) `ptr->prev->next = ptr->next; ptr->next->prev = ptr->prev;`

4. Memory Allocation Questions

- **Question:** If each node of a doubly linked list contains 3 pointers (one for data and two for links), how many bytes will a list with 10 nodes occupy, assuming each pointer occupies 4 bytes?
 - o A) 120 bytes
 - o B) 160 bytes
 - o C) 200 bytes
 - o D) 240 bytes
- **Answer:** C) 200 bytes (Each node occupies $3 * 4 \text{ bytes} = 12 \text{ bytes}$, so 10 nodes will occupy $10 * 12 = 120 \text{ bytes}$ for pointers + additional bytes for data).

5. Traversal Questions

- **Question:** How many pointers need to be updated to insert a new node after a given node in a doubly linked list?
 - o A) 1
 - o B) 2
 - o C) 3
 - o D) 4
- **Answer:** B) 2 (The pointers to the next and previous nodes need to be updated).

LABORATORY-7(CIRCULAR LINKED LIST)

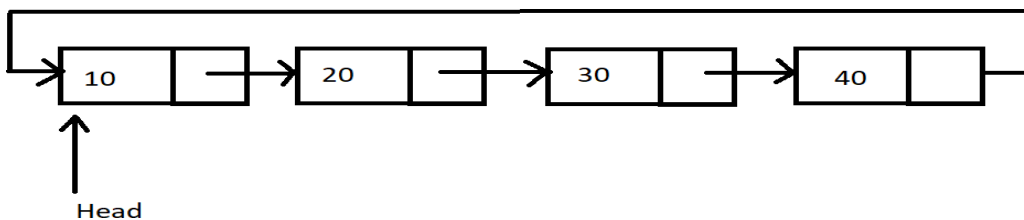
Objectives:

- Understand the concept and structure of circular linked lists.
- Implement basic operations on circular linked lists (insertion, deletion, traversal).
- Compare the advantages and disadvantages of circular linked lists with other linked lists.
- Solve problems using circular linked lists.

Brief Theory:

A **circular linked list** is a type of linked list in which the last node points to the first node, forming a circular loop. This can be implemented using singly or doubly linked lists. The primary characteristic is that there is no NULL at the end of the list; instead, the last node points back to the head of the list.

Circular Linked List: In the circular linked list the last node of the list contains the address of the first node and forms a circular chain



Key Concepts:

- **Circular Structure:** The last node points to the first node, creating a loop.
- **Head Pointer:** Points to the first node, and traversal starts from here.
- **Traversal:** Can continue indefinitely since there is no end to the list; special conditions are needed to stop traversal.

Advantages of Circular Linked List:

- **Efficient Memory Utilization:** No need for a NULL pointer at the end of the list.
- **Easier Implementation of Circular Data:** Ideal for circular buffering and applications like round-robin scheduling.
- **Uniform Traversal:** Easier to traverse the entire list from any point since it loops back.

Disadvantages of Circular Linked List:

- **Complex Implementation:** More complex to implement than a singly linked list due to the circular nature.
- **Difficult to Detect End of List:** Special care is needed to detect the end of traversal or to avoid infinite loops.

Outcomes:

- Students will be able to implement a circular linked list and perform basic operations like insertion, deletion, and traversal.

- Understand the practical applications and limitations of circular linked lists.

Problems on Circular Linked Lists:

Q. No	Question	CO	BL
1	Creating a Circular Linked List: Write a C program to create a circular singly linked list. The program should: 1. Accept values from the user to insert as nodes. 2. Display the circular linked list.	1ADPC20 2_2	K3
2	Insertion Operations in Circular Linked List: Write a C program to perform insertion operations in a circular singly linked list. The program should: 1. Insert a node at the beginning. 2. Insert a node at the end. 3. Insert a node after a given node.	1ADPC20 2_2	K3
3	Deletion Operations in Circular Linked List: Write a C program to perform deletion operations in a circular singly linked list. The program should: 1. Delete a node from the beginning. 2. Delete a node from the end. 3. Delete a given node from the list.	1ADPC20 2_2	K3
4	Traversing a Circular Linked List: Write a C program to traverse a circular singly linked list. The program should: 1. Accept values to create a circular linked list. 2. Display the list once around the circle (traverse the list).	1ADPC20 2_2	K3

Syntax:

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

```
// Example of creating a circular linked list with a single node
struct Node* head = (struct Node*)malloc(sizeof(struct Node));
head->data = 10;
head->next = head; // Points to itself, forming a circle
```

- **Insertion:** Example code for inserting a node at the beginning, end, or after a given node.
- **Deletion:** Example code for deleting a node from the beginning, end, or a specific position.

Expected Outcomes:

- Efficient manipulation of circular linked lists for various use cases.
- Clear understanding of when and why to use circular linked lists over other linked list types.

Observations:

- Observe how circular structures allow continuous traversal and are beneficial in scenarios requiring circular navigation.
- Note the differences in time complexity when performing various operations compared to other linked lists.

Conclusion:

Circular linked lists are a specialized form of linked lists that offer unique advantages in scenarios requiring continuous traversal or circular navigation. They provide efficient memory usage and simplify circular data handling but come with added complexity in implementation.

Textbooks:

- "Data Structures Using C" by Reema Thareja
- "Fundamentals of Data Structures in C" by Horowitz, Sahni, and Anderson-Freed

Online Reference Websites:

📖 **GeeksforGeeks: Circular Linked List:** <https://www.geeksforgeeks.org/circular-linked-list/>
Tutorialspoint: Circular Linked List
https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list.htm

Expected Oral Questions:

- How does a circular linked list differ from a singly linked list?
- What are the primary use cases for circular linked lists?
- How would you detect the end of traversal in a circular linked list?

FAQs in Interviews (with Answers):

1. **Q:** What is a circular linked list?
 - **A:** A circular linked list is a linked list where the last node points back to the first node, creating a loop or circle.
2. **Q:** What are the primary use cases for circular linked lists?
 - **A:** Circular linked lists are often used in applications like round-robin scheduling, circular buffering, and scenarios where a circular data structure is needed.
3. **Q:** How do you detect the end of a circular linked list?
 - **A:** In a circular linked list, there is no traditional "end." Traversal stops when you return to the starting node (head), or you use a condition like a counter to limit traversal steps.
4. **Q:** What are the main disadvantages of using a circular linked list?
 - **A:** The main disadvantages include increased complexity in implementation and potential difficulties in ensuring proper traversal without infinite loops.
5. **Q:** When would you prefer a circular linked list over a regular linked list?
 - **A:** You would prefer a circular linked list when you need continuous traversal or have a requirement for cyclic data management, like in scheduling or buffer management.

Gate questions:

1. Basic Understanding Questions

- **Question:** Which of the following statements is true about a circular linked list?
 - A) The last node points to the first node.
 - B) There is no NULL in the circular linked list.

- o C) It can be either singly or doubly linked.
- o D) All of the above.
- **Answer:** D) All of the above.

2. Time Complexity Questions

- **Question:** What is the time complexity to insert a node at the end of a circular singly linked list with 'n' nodes?
 - o A) $O(1)$
 - o B) $O(n)$
 - o C) $O(\log n)$
 - o D) $O(n \log n)$
- **Answer:** B) $O(n)$ (if no tail pointer is maintained).

3. Coding-Based Questions

- **Question:** In a circular singly linked list, which of the following is the correct way to insert a node after a given node 'ptr'?
 - o A) `newNode->next = ptr->next; ptr->next = newNode;`
 - o B) `ptr->next = newNode; newNode->next = ptr;`
 - o C) `newNode->next = ptr; ptr->next = newNode;`
 - o D) `ptr->next = newNode; newNode->next = ptr->next;`
- **Answer:** A) `newNode->next = ptr->next; ptr->next = newNode;`

4. Traversal Questions

- **Question:** How many times do you need to traverse the circular singly linked list to count the total number of nodes?
 - o A) Until NULL is reached
 - o B) Until the list loops back to the first node
 - o C) n times
 - o D) Twice the number of nodes
- **Answer:** B) Until the list loops back to the first node.

5. Memory Management Questions

- **Question:** In a circular singly linked list, which of the following operations requires more memory than in a normal singly linked list?
 - o A) Traversal
 - o B) Insertion at the beginning
 - o C) Deletion of the last node
 - o D) None of the above
- **Answer:** D) None of the above (Memory requirements are similar; operations are affected by the circular nature).

6. Application Questions

- **Question:** Which of the following applications can be efficiently implemented using a circular linked list?
 - o A) Round Robin scheduling
 - o B) Undo functionality in text editors
 - o C) Polynomial arithmetic
 - o D) Stack implementation
- **Answer:** A) Round Robin scheduling.

LABORATORY-8 (STACK)

Objectives:

- Understand the concept and structure of stacks.
- Implement stack operations (push, pop, peek) using arrays and linked lists.
- Compare the advantages and disadvantages of implementing stacks with arrays versus linked lists.

Brief Theory:

A stack is an abstract data type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack. A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

A stack can be implemented by means of Array, Structure, Pointer and Linked-List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays which makes it a fixed size stack implementation.

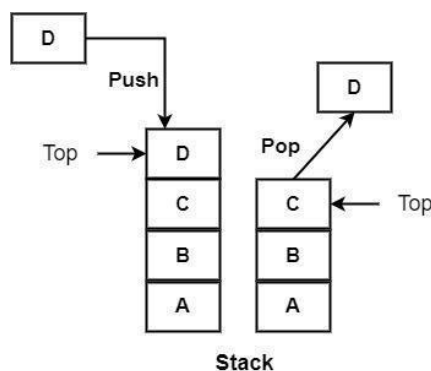
Basic Operations:

- *push()*: Pushing (storing) an element on the stack.
- *()*: Removing (accessing) an element from the stack.

To use a stack efficiently we need to check status of stack as well. For the same purpose, the following functionality is added to stacks;

- *p()*: Get the top data element of the stack, without removing it.
- *isF()*: Check if stack is full.
- *isEm()*: Check whether the stack is empty, and return true or false.

Below given diagram tries to depict push and pop operation of stack:



A **stack** is a linear data structure that follows the Last In, First Out (LIFO) principle. The operations of a stack are mainly:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove the element from the top of the stack.
- **Peek/Top:** Retrieve the top element without removing it.

Stacks can be implemented using arrays or linked lists:

- **Array-Based Stack:** Uses a static array to store elements, which has a fixed size.
- **Linked List-Based Stack:** Uses a dynamic linked list where each node represents an element, allowing for a stack of flexible size.

Key Concepts:

- **LIFO Principle:** Understanding the Last In, First Out behavior of stacks.
- **Overflow and Underflow:** Conditions when a stack is full (overflow) or empty (underflow).
- **Top Pointer:** A pointer (or index) that keeps track of the top element of the stack.

Advantages of Stack Using Array:

- **Simple Implementation:** Easy to implement using arrays.
- **Memory Efficiency:** Requires less memory overhead compared to linked lists since there's no need for pointers.

Disadvantages of Stack Using Array:

- **Fixed Size:** The size of the stack is fixed, which can lead to overflow if the array is full.
- **Wasted Space:** If the stack doesn't use all the allocated space, memory is wasted.

Advantages of Stack Using Linked List:

- **Dynamic Size:** The stack can grow or shrink as needed, which is memory efficient.
- **No Wasted Space:** Only the required memory is allocated, with no predefined limit.

Disadvantages of Stack Using Linked List:

- **Extra Memory:** Each node requires additional memory for storing pointers.
- **Complex Implementation:** Slightly more complex than arrays due to pointer management.

Outcomes:

- Students will be able to implement stacks using arrays and linked lists.
- Understand the practical applications and differences between stack implementations.

Problems on Stacks Using Arrays and Linked Lists:

Q. No	Question	CO	BL
1	Implementing Stack Using Array: Write a C program to implement a stack using an array. The program should: 1. Push elements onto the stack. 2. Pop elements from the stack. 3. Display the stack content.	1ADPC202_3	K3
2	Stack Operations: Write a C program to implement the basic stack operations (push, pop, peek) using an array. The program should: 1. Push an element onto the stack. 2. Pop an element from the stack. 3. Display the top element without popping.	1ADPC202_3	K3
3	Stack Using Linked List: Write a C program to implement a stack using a singly linked list. The program should: 1. Push elements onto the stack. 2. Pop elements from the stack. 3. Display the stack content.	1ADPC202_3	K3

Syntax:

Stack Using Array:

```
#define MAX 100
int stack[MAX];
int top = -1;

// Push operation
void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = value;
    }
}

// Pop operation
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}
```

Stack Using Linked List:

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

```

};

struct Node* top = NULL;

// Push operation
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack Overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}

// Pop operation
int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        struct Node* temp = top;
        int data = temp->data;
        top = temp->next;
        free(temp);
        return data;
    }
}

```

Expected Outcomes:

- Efficient manipulation of stacks for various use cases using arrays and linked lists.
- Clear understanding of the trade-offs between using arrays and linked lists for stack implementation.

Observations:

- Observe the behavior of stack operations and how they manage memory.
- Note the differences in handling overflow and underflow conditions in both implementations.

Conclusion:

Stacks are a fundamental data structure with many applications, such as expression evaluation, backtracking, and memory management. Choosing between array-based and linked list-based stacks depends on the specific requirements, such as memory constraints and the need for dynamic sizing.

Textbooks:

- "Data Structures Using C" by Reema Thareja
- "Fundamentals of Data Structures in C" by Horowitz, Sahni, and Anderson-Freed

Online Reference Websites:

📖 **GeeksforGeeks: Stack Using Array:**

<https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

📖 **GeeksforGeeks: Stack Using Linked List:**

<https://www.geeksforgeeks.org/stack-data-structure-using-linked-list/>

Expected Oral Questions:

- Explain the difference between a stack implemented using an array and one using a linked list.
- What are the main applications of stacks in computer science?
- How would you handle stack overflow in an array-based stack?

FAQs in Interviews (with Answers):

1. **Q:** What is a stack?
 - **A:** A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where the last element added is the first one to be removed.
2. **Q:** How would you implement a stack using a linked list?
 - **A:** In a linked list-based stack, each node contains data and a pointer to the next node. The top pointer always points to the top node, where push operations add nodes at the start, and pop operations remove nodes from the start.
3. **Q:** What is the main disadvantage of using an array-based stack?
 - **A:** The main disadvantage is its fixed size, which can lead to overflow if the stack exceeds its allocated array size.
4. **Q:** When would you prefer a linked list-based stack over an array-based stack?
 - **A:** A linked list-based stack is preferred when the stack size is unknown or can vary significantly, as it allows for dynamic memory allocation without a fixed limit.
5. **Q:** Can you implement a stack using other data structures?
 - **A:** Yes, stacks can also be implemented using other data structures like dynamic arrays or even other stacks.

Gate Questions:

1. Basic Understanding Questions

- **Question:** What is the time complexity of inserting an element in a stack implemented using an array?
 - A) $O(1)$
 - B) $O(n)$
 - C) $O(\log n)$
 - D) $O(n \log n)$
- **Answer:** A) $O(1)$

2. Operations Questions

- **Question:** Which of the following operations is not applicable to a stack?
 - A) Push
 - B) Pop
 - C) Peek
 - D) Enqueue
- **Answer:** D) Enqueue (Enqueue is an operation specific to queues).

3. Algorithm-Based Questions

- **Question:** What is the output of the following sequence of stack operations?
 - Push(1), Push(2), Pop(), Push(3), Push(4), Pop(), Pop()
 - A) 1, 4, 3
 - B) 2, 3, 1
 - C) 2, 4, 3
 - D) 1, 3, 4

- **Answer:** B) 2, 3, 1 (The stack would be manipulated in this order).

4. Infix to Postfix Conversion Questions

- **Question:** Convert the infix expression $(A + B) * (C - D)$ to postfix using a stack.
 - o A) $AB + CD - *$
 - o B) $AB + * CD -$
 - o C) $ABCD + - *$
 - o D) $AB + C - D *$
- **Answer:** A) $AB + CD - *$

5. Memory Management Questions

- **Question:** If a stack is implemented using a linked list, which of the following statements is true?
 - o A) Stack size is fixed.
 - o B) Push operation can fail due to lack of memory.
 - o C) Pop operation is $O(n)$.
 - o D) Stack requires continuous memory allocation.
- **Answer:** B) Push operation can fail due to lack of memory (since linked list implementation depends on dynamic memory allocation).

6. Applications Questions

- **Question:** Which of the following problems can be efficiently solved using a stack?
 - o A) Evaluating arithmetic expressions.
 - o B) Managing function calls.
 - o C) Undo mechanisms in text editors.
 - o D) All of the above.
- **Answer:** D) All of the above.

7. Postfix Evaluation Questions

- **Question:** What is the result of evaluating the postfix expression $5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$?
 - o A) 14
 - o B) 15
 - o C) 16
 - o D) 17
- **Answer:** B) 14
 - o **Solution:**
 1. Push 5
 2. Push 1
 3. Push 2
 4. Pop 2 and 1, evaluate $1 + 2 = 3$, push 3
 5. Push 4
 6. Pop 4 and 3, evaluate $3 * 4 = 12$, push 12
 7. Pop 12 and 5, evaluate $5 + 12 = 17$, push 17
 8. Push 3
 9. Pop 3 and 17, evaluate $17 - 3 = 14$.

8. Stack Overflow/Underflow Questions

- **Question:** Which condition indicates stack overflow when using an array-based stack?
 - o A) Top pointer is at -1.
 - o B) Top pointer is equal to the maximum size - 1.
 - o C) Stack is empty.
 - o D) Push operation fails even though there is space.
- **Answer:** B) Top pointer is equal to the maximum size - 1.

LABORATORY-9 (QUEUE)

Objectives:

- Understand the concept and structure of queues.
- Implement queue operations (enqueue, dequeue, front, rear) using arrays and linked lists.
- Compare the advantages and disadvantages of implementing queues with arrays versus linked lists.

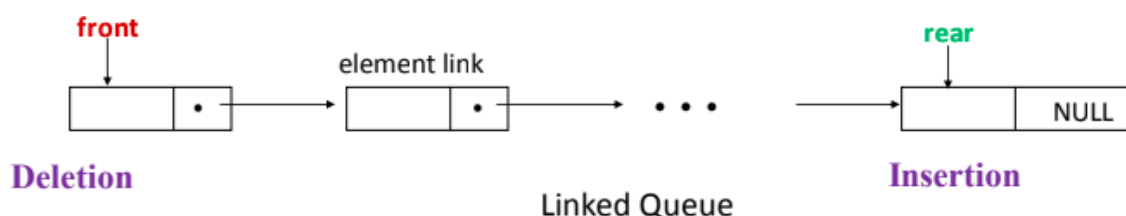
Brief Theory:

A queue is a linear data structure. It's an ordered list that follows the FIFO principle (First In - First Out). A queue is a structure that has some insertion and deletion constraints. In the case of Queue, insertion is done from one end, which is referred to as the rear end. The deletion is carried out by another end, which is referred to as a front end. The technical terminology for insertion and deletion in Queue are *Enqu()* and *Dequeue()* respectively. It has two pointers in its structure: a *Front* pointer and a *rear* pointer, element will be added using rear pointer and will be deleted using front pointer.

Operations in Queue:

1. *Create Q*: It creates an empty queue, Q.
2. *Enqueue Q, item*: Adds the element item to the rear of a queue and returns the new queue. The complexity of enqueue operation is (1).
3. *Dequeue Q*: Returns the older element of the queue.

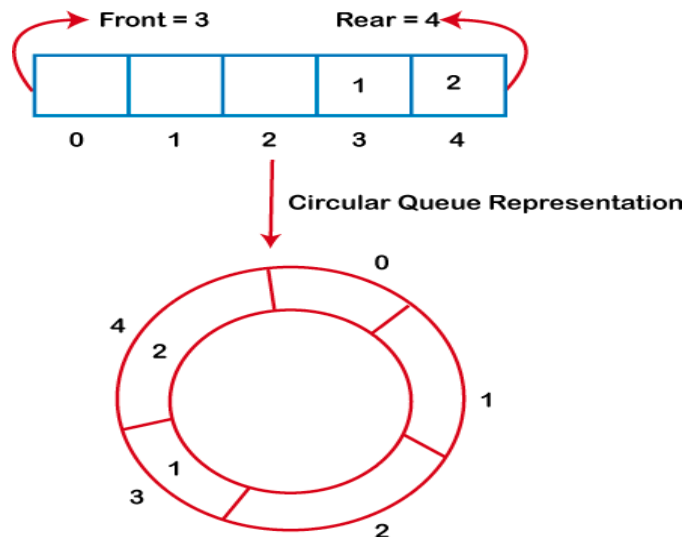
Queues can be implemented in two ways; 1) using array data structure, 2) Using linked list data structure.



Circular Queue: There is a limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue, then there might be

possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome those limitations, the concept of the circular queue was introduced.

A circular queue is similar to a linear queue as it is also based on the FIFO (First in First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a **Ring Buffer**. A modular function is used to implement *Enqueue* and *Deque* operation in circular queue.



Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority. The order in which elements are processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

These priority queue can be implemented using arrays and linked lists.

Queues can be implemented using arrays or linked lists:

- **Array-Based Queue:** Uses a static array with fixed size, where front and rear pointers track the queue ends.
- **Linked List-Based Queue:** Uses a dynamic linked list where each node represents an element, and nodes are added or removed from the front or rear dynamically.

Key Concepts:

- **FIFO Principle:** Understanding the First In, First Out behavior of queues.
- **Front and Rear Pointers:** Indices or pointers used to keep track of the first and last elements in the queue.
- **Circular Queue:** A variation of the array-based queue where the last position is connected back to the first, making it circular to efficiently utilize space.

Advantages of Queue Using Array:

- **Simple Implementation:** Easy to implement with basic array operations.
- **Access Time:** Faster access to elements through indexing.

Disadvantages of Queue Using Array:

- **Fixed Size:** The queue size is fixed, which can lead to overflow if the array is full.
- **Inefficient Memory Use:** Requires shifts for dequeue operations if not implemented circularly, causing potential wasted space.

Advantages of Queue Using Linked List:

- **Dynamic Size:** The queue can grow or shrink dynamically, allowing for efficient memory usage.
- **No Wasted Space:** Only the required memory is allocated, and the size can adjust to the number of elements.

Disadvantages of Queue Using Linked List:

- **Extra Memory:** Requires additional memory for pointers in each node.
- **Complexity:** More complex to implement due to the need for pointer management.

Outcomes:

- Students will be able to implement queues using arrays and linked lists.
- Understand the practical applications and differences between queue implementations.

Problems on Queues Using Arrays and Linked Lists:

Q. No	Question	CO	BL
1	Implementing Queue Using Array: Write a C program to implement a queue using an array. The program should: 1. Enqueue elements into the queue. 2. Dequeue elements from the queue. 3. Display the queue content.	1ADPC202_3	K3
2	Queue Operations: Write a C program to implement basic queue operations (enqueue, dequeue, front) using an array. The program should: 1. Enqueue an element. 2. Dequeue an element. 3. Display the front element.	1ADPC202_3	K3
3	Queue Using Linked List: Write a C program to implement a queue using a singly linked list. The program should: 1. Enqueue elements into the queue. 2. Dequeue elements from the queue. 3. Display the queue content.	1ADPC202_3	K3

4	Circular Queue: Write a C program to implement a circular queue using an array. The program should: 1. Enqueue elements into the circular queue. 2. Dequeue elements from the circular queue. 3. Display the queue content.	1ADPC202_3	K3
---	---	------------	----

Syntax:

Queue Using Array:

```
#define MAX 100
int queue[MAX];
int front = -1, rear = -1;

// Enqueue operation
void enqueue(int value) {
    if (rear == MAX - 1) {
        printf("Queue Overflow\n");
    } else {
        if (front == -1) front = 0;
        queue[++rear] = value;
    }
}

// Dequeue operation
int dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return -1;
    } else {
        return queue[front++];
    }
}
```

Queue Using Linked List:

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

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

// Enqueue operation
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Queue Overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

// Dequeue operation
```

```

int dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        return -1;
    } else {
        struct Node* temp = front;
        int data = temp->data;
        front = front->next;
        if (front == NULL) rear = NULL;
        free(temp);
        return data;
    }
}

```

Expected Outcomes:

- Efficient manipulation of queues for various use cases using arrays and linked lists.
- Clear understanding of the trade-offs between using arrays and linked lists for queue implementation.

Observations:

- Observe how queue operations affect memory management and element access times.
- Note the differences in handling overflow and underflow conditions in both implementations.


Conclusion:

Queues are fundamental data structures used extensively in various applications such as scheduling, buffering, and managing shared resources. Choosing between array-based and linked list-based queues depends on factors like the need for dynamic sizing, memory availability, and ease of implementation.

Textbooks:

- "Data Structures Using C" by Reema Thareja
- "Fundamentals of Data Structures in C" by Horowitz, Sahni, and Anderson-Freed

Online Reference Websites:

 **GeeksforGeeks: Queue Using Array:**

<https://www.geeksforgeeks.org/queue-set-1-introduction-and-array-implementation/>

 **GeeksforGeeks: Queue Using Linked List:** <https://www.geeksforgeeks.org/queue-linked-list-implementation/>

Expected Oral Questions:

- How does a queue differ from a stack in terms of operations and use cases?
- What are the main applications of queues in computer science?
- Explain the advantages of implementing a circular queue

FAQs in Interviews (with Answers):

1. **Q:** What is a queue?
 - **A:** A queue is a linear data structure that follows the First In, First Out (FIFO) principle, where the first element added is the first one to be removed.
2. **Q:** How would you implement a circular queue using an array?

- **A:** In a circular queue, the last position is connected back to the first to make the queue circular, which is implemented by adjusting the front and rear indices using modulo arithmetic.
- 3. **Q:** What is the main disadvantage of using an array-based queue?
 - **A:** The main disadvantage is its fixed size, leading to overflow if the array is full, and inefficient memory use if the queue does not utilize all allocated space.
- 4. **Q:** When would you prefer a linked list-based queue over an array-based queue?
 - **A:** A linked list-based queue is preferred when the size of the queue is unknown or can vary, as it allows for dynamic memory allocation without a predefined limit.
- 5. **Q:** How can queues be used in operating systems?
 - **A:** Queues are used in operating systems for process scheduling, managing tasks in a buffer, and handling requests in various services, ensuring tasks are processed in the order they arrive.

Gate Questions

1. Basic Understanding Questions

- **Question:** What is the time complexity of enqueue and dequeue operations in a queue implemented using a linked list?
 - A) $O(1)$
 - B) $O(n)$
 - C) $O(\log n)$
 - D) $O(n \log n)$
- **Answer:** A) $O(1)$

2. Circular Queue Questions

- **Question:** In a circular queue, how do you determine if the queue is full?
 - A) If the rear pointer is equal to the front pointer.
 - B) If the next position of the rear pointer is equal to the front pointer.
 - C) If the front pointer is equal to the rear pointer.
 - D) None of the above.
- **Answer:** B) If the next position of the rear pointer is equal to the front pointer.

3. Time Complexity Questions

- **Question:** What is the time complexity of inserting an element into a queue implemented using an array?
 - A) $O(1)$
 - B) $O(n)$
 - C) $O(\log n)$
 - D) $O(n \log n)$
- **Answer:** A) $O(1)$

4. Circular Queue Overflow Questions

- **Question:** Consider a circular queue with a capacity of 5. If the queue has 3 elements, which of the following conditions indicates that the queue is full?
 - A) The number of elements in the queue is equal to the capacity.
 - B) The rear pointer has wrapped around and is pointing to the front.
 - C) The front pointer is equal to the rear pointer.
 - D) None of the above.
- **Answer:** B) The rear pointer has wrapped around and is pointing to the front.

5. Queue Implementation Questions

- **Question:** Which of the following is a correct way to implement a queue using two stacks?
 - o A) Use one stack for enqueue and one stack for dequeue operations.
 - o B) Use two stacks for enqueue operations and one stack for dequeue operations.
 - o C) Use two stacks, one for enqueue operations and the other for dequeue operations.
 - o D) Use two stacks to keep track of the front and rear pointers.
- **Answer:** A) Use one stack for enqueue and one stack for dequeue operations.

6. Applications Questions

- **Question:** Which of the following scenarios can be efficiently managed using a queue?
 - o A) Task scheduling
 - o B) Undo functionality in text editors
 - o C) Parentheses matching
 - o D) None of the above
- **Answer:** A) Task scheduling (Queues are used for managing tasks in a scheduling system).

7. Queue with Maximum Size Questions

- **Question:** What will be the effect of increasing the maximum size of a queue implemented using an array?
 - o A) It will decrease the time complexity of enqueue and dequeue operations.
 - o B) It will increase the space complexity but will not affect the time complexity.
 - o C) It will increase the time complexity of enqueue and dequeue operations.
 - o D) It will not have any effect on the space or time complexity.
- **Answer:** B) It will increase the space complexity but will not affect the time complexity.

8. Queue with Linked List Questions

- **Question:** What is the major advantage of implementing a queue using a linked list over an array?
 - o A) Constant time complexity for enqueue and dequeue operations.
 - o B) No need to manage the queue size.
 - o C) Better memory utilization.
 - o D) Easier implementation of circular queue.
- **Answer:** B) No need to manage the queue size (Linked lists can grow dynamically, whereas arrays have a fixed size).

Laboratory 10: Stack Applications

Objectives:

1. Understand the concept of a **stack** and its core operations.
 2. Implement **stack operations** (push, pop, peek) using both **arrays** and **linked lists**.
 3. Explore **real-world applications** of stacks in computer science.
 4. Compare the advantages and disadvantages of **array-based** and **linked list-based** stacks.
-

Brief Theory:

A **stack** is a **linear data structure** that follows the **LIFO (Last In, First Out)** principle. The element that is added last is the first one to be removed.

- **Push()**: Adds an element to the top of the stack.
- **Pop()**: Removes the topmost element from the stack.
- **Peek()**: Returns the top element without removing it.
- **isEmpty()**: Checks if the stack is empty.

Stacks are widely used in various applications, including **expression evaluation**, **function calls**, **undo/redo operations**, and **syntax parsing**.

Applications of Stacks:

1. **Expression Evaluation:**
 - **Infix to Postfix Conversion:** Stacks are used to evaluate arithmetic expressions by converting infix expressions to postfix expressions (Reverse Polish Notation). This is useful in compiler design and mathematical evaluation.
 - **Parentheses Matching:** A stack can be used to check if parentheses in an expression are balanced.
2. **Undo/Redo Functionality:**
 - **Undo Operations:** When performing operations on software (e.g., text editing, drawing), stacks keep track of the changes made. When the user presses **undo**, the stack pops the last operation, reverting the software to its previous state.
 - **Redo Operations:** Similarly, a stack can help implement redo functionality by pushing the undone operation onto another stack.
3. **Function Call Management (Call Stack):**
 - **Function calls** are managed using stacks in the operating system. When a function is called, the stack stores the return address and local variables. When the function execution is complete, it is popped off the stack, and the program control is returned to the previous function.
4. **Backtracking Algorithms:**
 - In algorithms like **Depth First Search (DFS)** or **maze solving**, stacks are used to backtrack and explore different paths.
5. **Evaluating Postfix Expressions:**
 - In postfix notation, operators follow operands (e.g., **AB+**). Stacks are used to evaluate such expressions by pushing operands onto the stack and applying operators to the operands.
6. **Memory Management (Stack Allocation):**
 - Stacks are used to manage memory in many systems, including **automatic memory allocation** for function calls and local variables.

Infix to Postfix Conversion Using Stack

Infix notation is the standard arithmetic notation we use, where operators are placed between operands (e.g., $A + B$). Postfix notation (Reverse Polish Notation) is a form where operators come after their operands (e.g., $A B +$).

To convert an infix expression to postfix, we can use a **stack** to temporarily hold operators and manage the precedence of operators. Here's how the conversion process works:

Steps for Conversion:

1. **Operand** (i.e., numbers or variables): If the character is an operand, add it directly to the result.
2. **Left Parenthesis (**: Push it onto the stack to indicate that operators inside the parentheses should be handled first.
3. **Right Parenthesis)**: Pop from the stack and add to the result until a left parenthesis (is encountered. Pop the (but don't add it to the result.
4. **Operator**: Pop operators from the stack to the result if they have higher or equal precedence than the current operator. Push the current operator onto the stack.

Operator Precedence:

- Operators $+$ and $-$ have lower precedence than $*$ and $/$.
- Operators $*$ and $/$ have higher precedence than $+$ and $-$.
- Parentheses (and) are used to change the precedence order.

Infix to Postfix Algorithm:

1. Initialize an empty stack for operators and an empty list for the output.
2. Read the infix expression from left to right:
 - If the character is an **operand**, append it to the result.
 - If the character is a **left parenthesis (**, push it onto the stack.
 - If the character is a **right parenthesis)**, pop from the stack to the result until a left parenthesis is encountered.
 - If the character is an **operator**, pop operators from the stack to the result if they have higher or equal precedence. Push the current operator onto the stack.
3. After the entire expression has been read, pop all remaining operators from the stack to the result.

Code Implementation:

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

#define MAX 100

// Define precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    } else if (op == '*' || op == '/') {
        return 2;
    }
    return 0;
}
```

```

// Function to perform the conversion from infix to postfix
void infixToPostfix(char* infix, char* postfix) {
    char stack[MAX];
    int top = -1; // Stack pointer
    int j = 0; // Index for postfix array

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

        // If the current character is an operand (letter or digit), add it to the result
        if (isalnum(current)) {
            postfix[j++] = current;
        }
        // If the current character is '(', push it to the stack
        else if (current == '(') {
            stack[++top] = current;
        }
        // If the current character is ')', pop from the stack to the result until '(' is encountered
        else if (current == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = stack[top--];
            }
            top--; // Discard the '('
        }
        // If the current character is an operator
        else if (current == '+' || current == '-' || current == '*' || current == '/') {
            while (top != -1 && precedence(stack[top]) >= precedence(current)) {
                postfix[j++] = stack[top--];
            }
            stack[++top] = current; // Push the current operator to the stack
        }
    }

    // Pop any remaining operators in the stack
    while (top != -1) {
        postfix[j++] = stack[top--];
    }

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

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

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

    infixToPostfix(infix, postfix);

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

    return 0;
}

```

Applications of Stack:

Here are some common applications of stacks with brief explanations and examples.

1. **Expression Evaluation (Postfix/Infix/Prefix):**
 - Stacks are used to convert an **infix expression** to **postfix** (or **prefix**) and to evaluate postfix expressions. For example, for the expression $3 + 5 * 2$, we can convert it to postfix as $3\ 5\ 2\ * +$ and then evaluate it.
2. **Algorithm for Postfix Evaluation:**
 - Traverse the postfix expression:
 - If the character is a number, push it to the stack.
 - If the character is an operator, pop the top two numbers, apply the operator, and push the result back onto the stack.
 - At the end, the stack will contain the final result.
3. **Undo/Redo Functionality:**
 - Stacks are often used to implement undo and redo functionality in text editors or applications. Each operation (like text insertion or deletion) is pushed onto a stack. When the user presses undo, the last operation is popped from the stack and reversed.
4. **Balancing Parentheses:**
 - Stacks can be used to check whether parentheses in an expression are balanced. Every opening parenthesis is pushed onto the stack, and every closing parenthesis causes a pop operation. If at any point a closing parenthesis is encountered without a matching opening one, the expression is unbalanced.
5. **Function Call Management (Recursion):**
 - Stacks are used to manage function calls in recursive algorithms. Each recursive call is pushed onto the stack with its local variables, and when the base condition is met, the stack is popped to return control to the previous function call.
6. **Backtracking Algorithms:**
 - Stacks are used in **backtracking algorithms**, such as solving mazes, puzzles, or finding paths in graphs. A stack stores the current state, and backtracking involves popping states off the stack until the desired condition is met.

Practical Applications of Stack:

1. Expression Evaluation:

- Stacks are used to evaluate **mathematical expressions** in postfix or infix notation.
- Example: Convert infix expressions like $(A+B)*(C+D)$ to postfix form and evaluate them.

2. Parentheses Matching (Balanced Parentheses):

- A stack can be used to check if the parentheses in an expression are balanced.
- Example: For an expression like $((A+B) * (C+D))$, we push (and pop it when encountering). If the stack is empty at the end, the parentheses are balanced.

3. Undo/Redo Operations:

- Implement undo and redo features by maintaining separate stacks for operations.
- Example: In a text editor, every action (e.g., type, delete) is pushed onto an undo stack. When the user presses **undo**, the action is popped from the undo stack and pushed to the redo stack.

4. Function Call Stack (Call Stack):

- The function call stack is used to store the return address and local variables of functions. When a function call occurs, a new **stack frame** is created. After the function finishes execution, the frame is removed, and control is returned to the calling function.

5. Backtracking Algorithms (DFS):

- In depth-first search (DFS) algorithms, stacks are used to backtrack through paths in a tree or graph.
- Example: In maze solving, a stack can be used to store the positions in the maze to backtrack and find the solution.

Advantages and Disadvantages of Stack Implementations:

Array-Based Stack:

- **Advantages:**
 - **Simple Implementation:** Easy to implement using an array with fixed size.
 - **Faster Access:** Direct access to elements through array indexing.
- **Disadvantages:**
 - **Fixed Size:** The size is limited, which can cause overflow if the stack grows beyond its capacity.
 - **Wasted Space:** In some cases, there may be unused space at the end of the array.

Linked List-Based Stack:

- **Advantages:**
 - **Dynamic Size:** The size of the stack can grow or shrink dynamically, avoiding overflow problems.
 - **Efficient Memory Use:** Memory is allocated only when needed.
- **Disadvantages:**
 - **Extra Memory:** Each node requires additional memory for pointers.
 - **Complexity:** More complex to implement and manage due to pointer operations.

Questions :

Q. No	Question	CO	BL
1	Infix to Postfix Conversion: Write a C program to convert an infix expression to a postfix expression using a stack. The program should: 1. Accept an infix expression with operators and operands. 2. Convert the expression to postfix notation. 3. Display the resulting postfix expression.	1ADPC202_3	K3
2	Postfix Evaluation: Write a C program to evaluate a postfix expression using a stack. The program should: 1. Accept a postfix expression with operands and operators. 2. Evaluate the expression using a stack. 3. Display the result of the evaluation.	1ADPC202_3	K3

Conclusion:

Stacks are fundamental data structures that are used in many practical applications such as **expression evaluation**, **backtracking**, **function calls**, and **undo/redo functionality**. The choice between using an **array-based** or **linked list-based stack** depends on the problem requirements such as **fixed size** or **dynamic resizing**.

By implementing stacks, students gain insights into **LIFO** operations and learn to apply this concept to real-world scenarios in computer science.

Textbooks

1. **"Data Structures and Algorithms in C"** by Robert Lafore
 - Chapter 4: Stacks and Queues
2. **"Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
 - Chapter 10: Stacks and Queues
3. **"Data Structures Using C"** by Aaron M. Tenenbaum
 - Chapter 3: Stacks

Online Reference Websites

- **GeeksforGeeks - Stack Data Structure**
<https://www.geeksforgeeks.org/stack-data-structure/>
- **TutorialsPoint - Stacks**
https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
- **Programiz - Stack Data Structure**
<https://www.programiz.com/dsa/stack>
- **Khan Academy - Stacks**
<https://www.khanacademy.org/computing/computer-science/algorithms#stacks>

Expected Oral Questions

1. How do you detect a stack overflow or stack underflow condition?
2. How does the stack data structure help in balancing parentheses in an expression?
3. How can stacks be used in evaluating mathematical expressions (like postfix or infix)?
4. Can you explain how a stack is used in function call management in programming?
5. What is the time complexity of push, pop, and peek operations in a stack?

FAQs in Interviews:

1. **Q: What is a stack?**
 - **A:** A stack is a linear data structure that follows the **LIFO** (Last In, First Out) principle, where elements are added (pushed) and removed (popped) from the top of the stack.

2. **Q: How would you implement a stack using an array?**
 - A: A stack can be implemented using a static array where the **top** pointer keeps track of the last element added. The **push()** operation adds an element at the top, and the **pop()** operation removes the top element.
 3. **Q: What are some applications of stacks in real-world programming?**
 - A: Stacks are used in applications such as **expression evaluation, function calls and recursion, undo-redo operations, parsing syntax in compilers, and balancing parentheses** in expressions.
 4. **Q: How can a stack be used in depth-first search (DFS)?**
 - A: In **DFS**, a stack is used to explore nodes in a graph. The algorithm pushes each node onto the stack and explores its neighbors before backtracking. The stack helps in managing the state of each node during the traversal.
-

Gate Questions (with Answers):

1. **Basic Understanding Questions Q: What is the time complexity of push and pop operations in a stack implemented using an array?**
 - A) $O(1)$
 - B) $O(n)$
 - C) $O(\log n)$
 - D) $O(n \log n)$
2. **Answer: A) $O(1)$**
3. **Expression Evaluation Questions Q: Which data structure is used for converting infix expressions to postfix expressions?**
 - A) Queue
 - B) Stack
 - C) Array
 - D) Linked List
4. **Answer: B) Stack**
5. **Backtracking Questions Q: In a backtracking algorithm, how does a stack help in exploring all possible solutions?**
 - A) It stores the previous states and allows for backtracking to the last valid state.
 - B) It stores the solution set at each step and updates it when necessary.
 - C) It is used to store the best solution found so far.
 - D) None of the above.
6. **Answer: A) It stores the previous states and allows for backtracking to the last valid state.**
7. **Time Complexity of Stack Operations Q: What is the time complexity of popping an element from a stack implemented using an array?**
 - A) $O(1)$
 - B) $O(n)$
 - C) $O(\log n)$
 - D) $O(n \log n)$
8. **Answer: A) $O(1)$**

Laboratory 11: Searching Algorithms

Objectives

- Understand and implement different searching algorithms in C.
- Analyze the efficiency of each algorithm.
- Recognize use cases where each searching technique is most appropriate.

Brief Theory

1. Linear Search

Description:

Linear search is the simplest searching algorithm, where each element in the list is checked sequentially until the target element is found or the end of the list is reached. It works well for small datasets or when a single search operation is needed on unsorted data.

Diagram:

Consider a list of integers: [3, 5, 8, 2, 9, 1, 6] and a target element 9.

[3] → [5] → [8] → [2] → [9] → [1] → [6]

- **Step 1:** Start at the first element 3 and check if it matches 9 (it doesn't).
- **Step 2:** Move to the next element 5 and check if it matches 9 (it doesn't).
- **Step 3:** Continue until reaching the element 9, where a match is found.
- **Result:** Element 9 is found at index 4.

Time Complexity: $O(n)$

Space Complexity: $O(1)$

Program Code (Linear Search):

```
#include <stdio.h>

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

int main() {
    int arr[] = {3, 5, 8, 2, 9, 1, 6};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 9;
    int result = linearSearch(arr, size, target);

    if (result != -1) {
```

```

        printf("Element found at index: %d\n", result);
    } else {
        printf("Element not found.\n");
    }
    return 0;
}

```

2. Binary Search

Description:

Binary search is an efficient searching algorithm that requires the array to be sorted. It works by repeatedly dividing the search interval in half. If the target value is less than the middle element, the search continues in the left half; otherwise, it continues in the right half.

Diagram:

Consider a sorted list of integers: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] and a target element 9.

1. **Step 1:** Start by finding the middle element (9 at index 4).
2. **Step 2:** Since the middle element 9 matches the target, the search stops.

Binary search halves the search space with each iteration, making it significantly faster than linear search for large, sorted datasets.

Time Complexity: $O(\log n)$

Space Complexity: $O(1)$ for iterative and $O(\log n)$ for recursive implementations.

Program Code (Binary Search):

```

#include <stdio.h>

int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if the target is present at mid
        if (arr[mid] == target) {
            return mid;
        }

        // If target is greater, ignore left half
        if (arr[mid] < target) {
            left = mid + 1;
        }

        // If target is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }
}

```



```

    }
    }
    return -1; // Return -1 if the element is not present
}

int main() {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 9;
    int result = binarySearch(arr, size, target);

    if (result != -1) {
        printf("Element found at index: %d\n", result);
    } else {
        printf("Element not found.\n");
    }
    return 0;
}

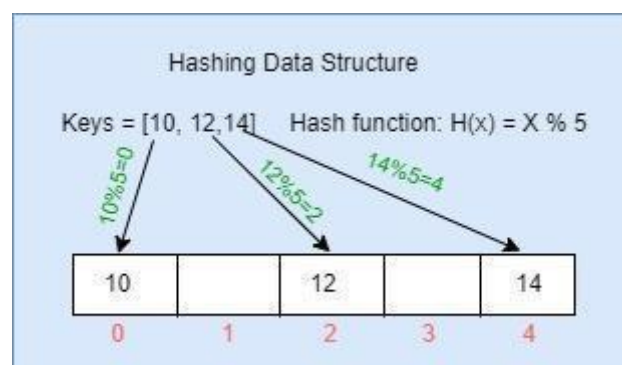
```

3. Hashing

Description:

Hashing is a technique used to map data to a fixed-size table using a hash function. In the context of searching, hashing allows us to access elements in constant time, $O(1)$, by using a key to compute an index in the hash table. Collisions occur when two keys map to the same index, which can be managed by techniques such as chaining or open addressing.

Hashing is done for the faster access of elements. In hashing a given key is converted to a smaller value I by the help of hash function, and then the given key is stored at the index I . Basically, the goal of the hash function is determining the hash value of a particular key, and then that hash value will be used as index to store that particular key. Generally, the hash function are simple modular functions, which can determine the hash value at (1), hence the element can be stored and access in $O(1)$ time using hashing. Below picture is depicting a hashing scheme.



Collision: If the hash value calculated by hash function is same for

more than one key then those keys are indexed at same slot, this is known as collision. That collision needs to be handled properly; few collision handling techniques are:

1. Linear Probing.
2. Double Hashing.
3. Quadratic Hashing.

In the hashing scheme shown in the above picture if key value 15 is need to inserted now, then that will lead to collision because 15 will also map to index 0.

Diagram:

Consider a hash table of size 5, where elements are mapped based on element \% 5 .

Key	Hash Value	Index
12	2	2
25	0	0
35	0	Collision managed by chaining or open addressing

Time Complexity: $O(1)$ $O(1)$ $O(1)$ for average cases, $O(n)$ $O(n)$ $O(n)$ in the worst case when many collisions occur.

Space Complexity: $O(n)$ $O(n)$ $O(n)$

Program Code (Hashing with Chaining):

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

#define TABLE_SIZE 5

// Define a node for the chaining linked list
struct Node {
    int data;
    struct Node* next;
```

```

};

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

// Insert function using chaining
void insert(struct Node* hashTable[], int key) {
    int hashIndex = hashFunction(key);
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = hashTable[hashIndex];
    hashTable[hashIndex] = newNode;
}

// Search function
int search(struct Node* hashTable[], int key) {
    int hashIndex = hashFunction(key);
    struct Node* temp = hashTable[hashIndex];
    while (temp) {
        if (temp->data == key) {
            return 1; // Element found
        }
        temp = temp->next;
    }
    return 0; // Element not found
}

int main() {
    struct Node* hashTable[TABLE_SIZE] = {NULL};
    int keys[] = {12, 25, 35, 26, 15};

    for (int i = 0; i < 5; i++) {
        insert(hashTable, keys[i]);
    }

    int target = 26;
    if (search(hashTable, target)) {
        printf("Element found in the hash table.\n");
    } else {
        printf("Element not found in the hash table.\n");
    }
    return 0;
}

```

Questions :

Q. No	Question	CO	BL
1	Linear Search: Write a C program to implement linear search. The program should: 1. Accept a list of integers from the user. 2. Search for a target value in the list. 3. Display the index of the target if found or a message if not found.	1ADPC202_6	K3
2	Binary Search: Write a C program to implement binary search on a sorted array. The program should: 1. Accept a sorted list of integers. 2. Search for a target value in the list using binary search. 3. Display the index of the target if found or a message if not found.	1ADPC202_6	K3
3	Hashing (Chaining): Write a C program to implement hashing using chaining. The program should: 1. Create a hash table using an array of linked lists. 2. Implement the insert and search operations using hashing. 3. Display the hash table after insertion and search operations.	1ADPC202_6	K3

Observations

- **Linear Search** is simple but inefficient for large datasets.
- **Binary Search** is very efficient for sorted datasets.
- **Hashing** provides nearly constant-time access but requires collision management.

Outcomes

- Develop a clear understanding of linear search, binary search, and hashing techniques.
- Ability to apply appropriate searching algorithms based on data properties.
- Enhanced problem-solving skills through hands-on coding of search algorithms.

Conclusion

Each searching algorithm has its own advantages and limitations. Linear search is best for unsorted or small datasets, binary search is ideal for large sorted arrays, and hashing is efficient for constant-time lookups with appropriate collision handling.

References

1. "Data Structures Using C" by Reema Thareja.
2. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Expected Oral Questions

1. **What is the difference between Linear Search and Binary Search?**
 2. **Why is Binary Search faster than Linear Search?**
 3. **Explain how Hashing works. What is a hash collision?**
 4. **In what scenarios would you use Linear Search over Binary Search?**
 5. **Describe a situation where Hashing might be inefficient.**
 6. **Explain open addressing and chaining in hashing.**
 7. **What is the time complexity of Linear Search, Binary Search, and Hashing?**
-

FAQ in Interviews with Answers

1. **Q: When would you use Linear Search instead of Binary Search?**
A: Linear Search is used when the data is unsorted or small. Binary Search requires a sorted dataset to be efficient, making it unsuitable for unsorted data.
2. **Q: What is a hash collision, and how is it resolved?**
A: A hash collision occurs when two keys hash to the same index. It can be resolved through techniques like chaining (storing multiple elements at the same index using a linked list) or open addressing (finding another index within the hash table).
3. **Q: Why is Binary Search time complexity $O(\log n)$?**
A: Binary Search divides the search space in half each time, making the time complexity logarithmic, $O(\log n)$. Each division reduces the dataset size, achieving efficient search in large datasets.
4. **Q: Describe the difference between open addressing and chaining in Hashing.**
A: In chaining, each index in the hash table points to a list where all colliding elements are stored. In open addressing, if a collision occurs, the algorithm probes for the next available index to store the item.
5. **Q: How does the load factor affect the performance of a hash table?**
A: The load factor (ratio of elements to hash table size) impacts collision frequency. A higher load factor increases collision likelihood, reducing efficiency. Expanding the table size when the load factor is high can mitigate this.
6. **Q: Can Binary Search be applied to linked lists?**
A: Binary Search is generally inefficient for linked lists, as accessing the middle element requires linear traversal, defeating the purpose of a logarithmic search.

Laboratory 12: Sorting Algorithms - 1

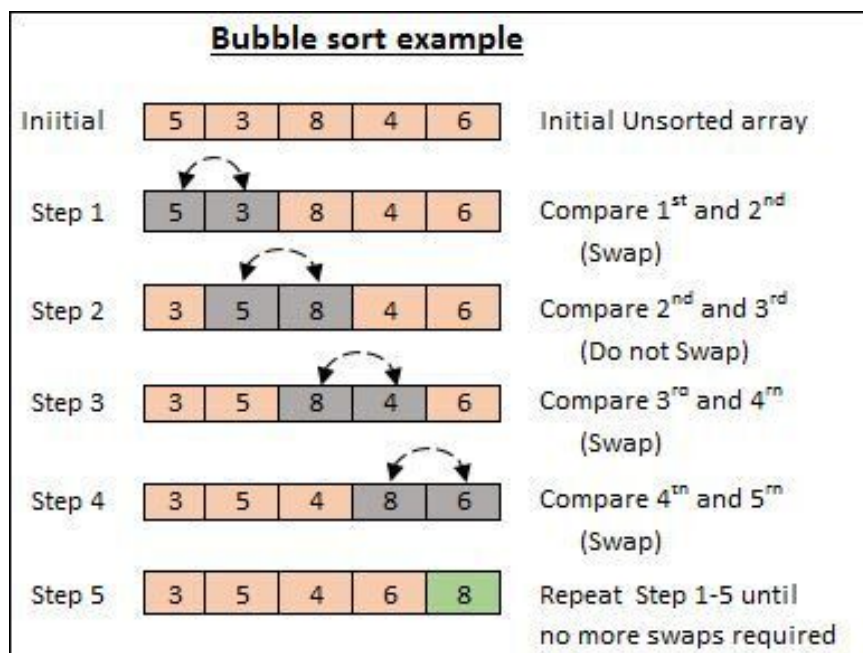
Objective

- Understand and implement the basic sorting algorithms: Bubble Sort, Selection Sort, and Insertion Sort.
- Analyze the time and space complexity of these algorithms.
- Compare the performance of these sorting algorithms.

Brief Theory

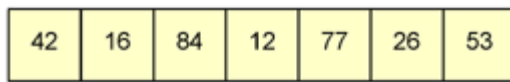
Bubble Sort:

- Bubble Sort is a simple comparison-based sorting algorithm where each element is compared with the next one and swapped if they are in the wrong order. The process is repeated until the array is sorted.
- **Time Complexity:** Best case: $O(n)$, Average case: $O(n^2)$, Worst case: $O(n^2)$.
- **Space Complexity:** $O(1)$

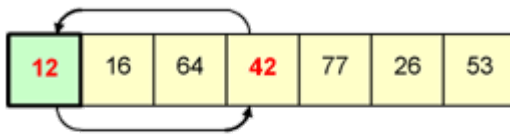


Selection Sort:

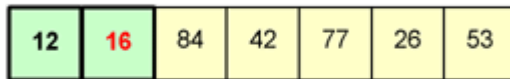
- In Selection Sort, the list is divided into two parts: the sorted part and the unsorted part. The algorithm repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted part and swaps it with the first unsorted element.
- **Time Complexity:** Best, Worst, and Average case: $O(n^2)$
- **Space Complexity:** $O(1)$



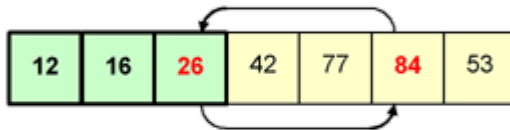
The array, before the selection sort operation begins.



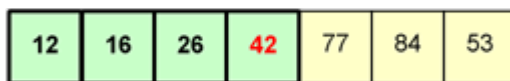
The smallest number (12) is swapped into the first element in the structure.



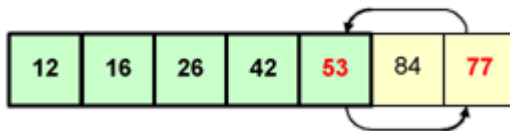
In the data that remains, 16 is the smallest; and it does not need to be moved.



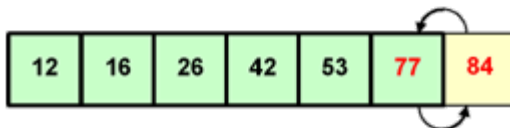
26 is the next smallest number, and it is swapped into the third position.



42 is the next smallest number; it is already in the correct position.



53 is the smallest number in the data that remains; and it is swapped to the appropriate position.

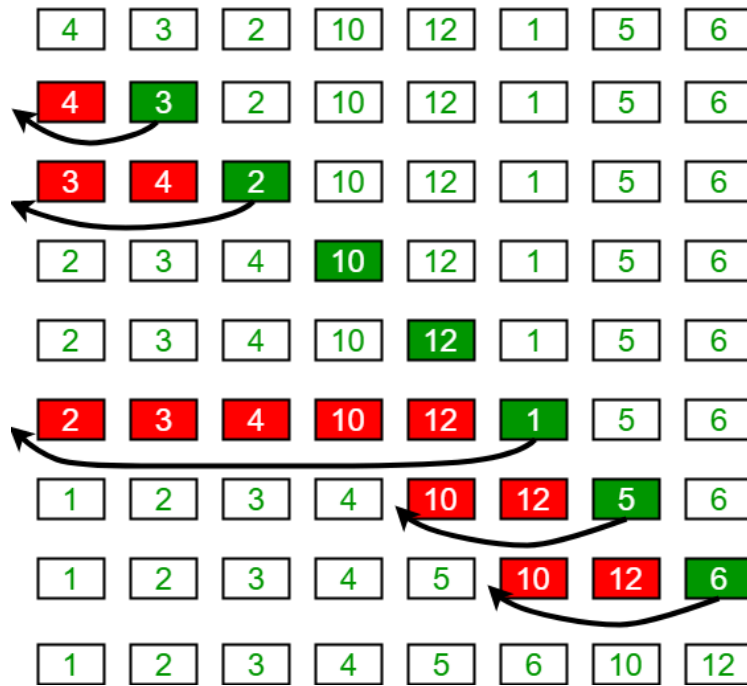


Of the two remaining data items, 77 is the smaller; the items are swapped. *The selection sort is now complete.*

Insertion Sort:

- In Insertion Sort, the list is divided into a sorted part and an unsorted part. The algorithm picks elements from the unsorted part and inserts them into the correct position in the sorted part.
- **Time Complexity:** Best case: $O(n)$, Worst and Average case: $O(n^2)$
- **Space Complexity:** $O(1)$

Insertion Sort Execution Example



Problems on Sorting Algorithms

Q. No	Question	CO	BL
1	Bubble Sort: Write a C program to implement the bubble sort algorithm. The program should: <ol style="list-style-type: none"> 1. Accept a list of integers from the user. 2. Sort the list using the bubble sort algorithm. 3. Display the sorted list. 	1ADPC20 2_6	K3
2	Selection Sort: Write a C program to implement the selection sort algorithm. The program should: <ol style="list-style-type: none"> 1. Accept a list of integers from the user. 2. Sort the list using the selection sort algorithm. 3. Display the sorted list. 	1ADPC20 2_6	K3
3	Insertion Sort: Write a C program to implement the insertion sort algorithm. The program should: <ol style="list-style-type: none"> 1. Accept a list of integers from the user. 2. Sort the list using the insertion sort algorithm. 3. Display the sorted list. 	1ADPC20 2_6	K3

Observation

- **Bubble Sort** is inefficient for large datasets due to its $O(n^2)$ time complexity, but it is stable and simple.
 - **Selection Sort** also has $O(n^2)$ time complexity in all cases, but it performs fewer swaps than Bubble Sort.
 - **Insertion Sort** is more efficient than Bubble and Selection Sort for small datasets or nearly sorted arrays.
-

Outcome

- After completing this lab, students will be able to:
 - Implement sorting algorithms in C.
 - Analyze and compare the time complexity of each algorithm.
 - Understand when to use each algorithm based on data conditions.
-

Conclusion

- The sorting algorithms have their specific use cases:
 - **Bubble Sort** is easy to implement but inefficient for large datasets.
 - **Selection Sort** has fewer swaps but is also inefficient for large arrays.
 - **Insertion Sort** is efficient for small or nearly sorted arrays.
 - Despite their differences, all three sorting algorithms have a worst-case time complexity of $O(n^2)$ and are not suitable for sorting large datasets compared to more efficient algorithms like Quick Sort and Merge Sort.
-

Textbooks

1. **Data Structures and Algorithms in C** by Adam Drozdek
 2. **Introduction to Algorithms** by Cormen, Leiserson, Rivest, and Stein
 3. **Data Structures and Algorithm Analysis in C** by Mark Allen Weiss
-

Online Reference Websites

- [GeeksforGeeks - Sorting Algorithms](#)
 - [TutorialsPoint - Sorting Algorithms](#)
 - [Programiz - Sorting Algorithms](#)
-

Expected Oral Questions

1. What are the time complexities of Bubble Sort, Selection Sort, and Insertion Sort?
2. Why is Bubble Sort considered inefficient for large datasets?
3. How does Insertion Sort differ from Selection Sort in terms of operation?
4. What is the best case scenario for Insertion Sort?
5. Can you modify any of these algorithms to sort in descending order?

6. Why do we say that Bubble Sort is a stable sorting algorithm?
 7. What is the space complexity of these sorting algorithms?
 8. In which cases would you prefer to use Insertion Sort over other algorithms?
-

FAQs in Interviews with Answers

1. **Q:** Which sorting algorithm is best for small datasets?
A: Insertion Sort is best for small datasets because it has low overhead and performs well on nearly sorted arrays.
2. **Q:** What is the worst-case scenario for Bubble Sort?
A: The worst-case scenario for Bubble Sort occurs when the array is in reverse order, and it has a time complexity of $O(n^2)$.
3. **Q:** How do you optimize Bubble Sort to improve its performance?
A: One optimization is to add a flag to track whether any swaps were made in a pass. If no swaps were made, the algorithm terminates early, reducing unnecessary passes.
4. **Q:** Why is Selection Sort not considered a stable sorting algorithm?
A: Selection Sort is not stable because it can change the relative order of equal elements during the swapping process.
5. **Q:** Is there a situation where you would prefer Selection Sort over Insertion Sort?
A: Selection Sort may be preferable if memory usage is limited, as it performs fewer swaps than Insertion Sort and does not require additional memory for temporary storage.

GATE MCQs with Answers

1. Which of the following sorting algorithms is the most efficient for large datasets?

- A) Bubble Sort
- B) Selection Sort
- C) Merge Sort
- D) Insertion Sort

Answer: C) Merge Sort

2. What is the time complexity of the Insertion Sort algorithm in the worst case?

- A) $O(n)$
- B) $O(n^2)$
- C) $O(\log n)$
- D) $O(n \log n)$

Answer: B) $O(n^2)$

3. The time complexity of Bubble Sort in the worst case is:

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n^2)$
- D) $O(n^3)$

Answer: C) $O(n^2)$ $O(n^2)$ $O(n^2)$

4. Which of the following is a characteristic of Selection Sort?

- A) Stable sorting
- B) Always performs $O(n^2)$ $O(n^2)$ $O(n^2)$ comparisons
- C) Requires $O(1)$ $O(1)$ $O(1)$ space
- D) More efficient than Insertion Sort for small arrays

Answer: B) Always performs $O(n^2)$ $O(n^2)$ $O(n^2)$ comparisons

5. In the worst case, how many swaps does Bubble Sort make for an array of size n ?

- A) n
- B) n^2
- C) $n \log n$
- D) n^3

Answer: B) n^2

6. Which sorting algorithm works by finding the smallest element from the unsorted part and swapping it with the first unsorted element?

- A) Insertion Sort
- B) Selection Sort
- C) Merge Sort
- D) Bubble Sort

Answer: B) Selection Sort

Laboratory 13: Sorting Algorithms - 2

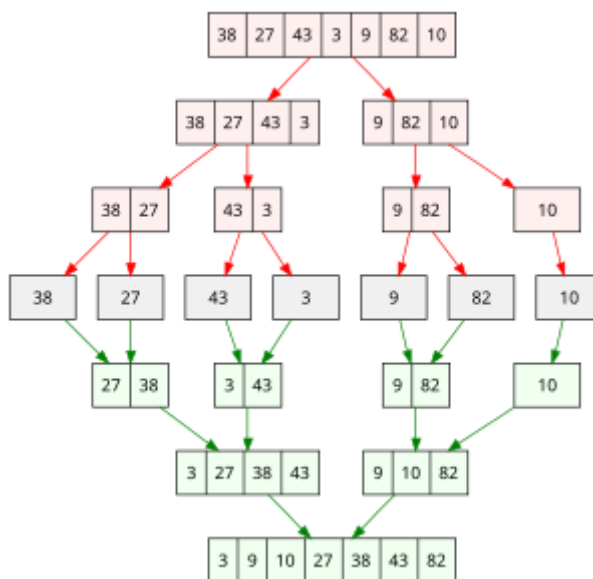
Objective

- Understand and implement the **Merge Sort** and **Quick Sort** algorithms.
- Analyze and compare the time and space complexity of **Merge Sort** and **Quick Sort**.
- Explore the divide-and-conquer strategy in sorting algorithms.

Brief Theory

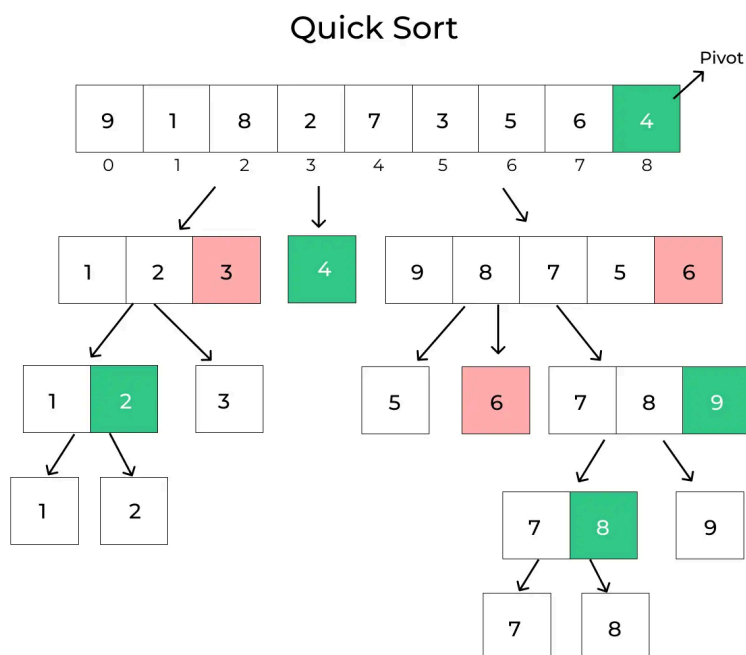
Merge Sort:

- **Merge Sort** is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and merges the two sorted halves to produce the final sorted array.
- It works by recursively dividing the array until each sub-array has only one element. Then, it merges these sub-arrays to form a sorted array.
- **Time Complexity:**
 - Worst, Best, and Average Case: $O(n \log n)$
- **Space Complexity:**
 - $O(n)$, as it requires extra space for the merging process.



Quick Sort:

- **Quick Sort** is another divide-and-conquer algorithm, but unlike Merge Sort, it works by selecting a "pivot" element, partitioning the array into two sub-arrays (elements less than the pivot and elements greater than the pivot), and then recursively sorting the sub-arrays.
- Quick Sort is generally faster than Merge Sort on average, but its worst-case time complexity is higher if a poor pivot is chosen.
- **Time Complexity:**
 - Worst Case: $O(n^2)$
 - Best and Average Case: $O(n \log n)$
- **Space Complexity:**
 - $O(\log n)$, for the recursive stack in the algorithm.



Problems on Sorting Algorithms

Q. No	Question	CO	BL
1	Merge Sort: Write a C program to implement the merge sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the merge sort algorithm. 3. Display the sorted list.	1ADPC202_6	K3
2	Quick Sort: Write a C program to implement the quick sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the quick sort algorithm. 3. Display the sorted list.	1ADPC202_6	K3
3	Heap Sort: Write a C program to implement the heap sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the heap sort algorithm. 3. Display the sorted list.	1ADPC202_6	K3

Observation

- **Merge Sort** guarantees $O(n \log n)$ time complexity, which makes it more predictable than Quick Sort in terms of performance.
 - **Quick Sort** typically performs better than **Merge Sort** on average, but its worst-case time complexity is $O(n^2)$, which can be avoided with proper pivot selection.
 - The space complexity of **Merge Sort** is higher than **Quick Sort** due to the need for temporary arrays for merging.
-

Outcome

- After completing this lab, students will be able to:
 - Implement **Merge Sort** and **Quick Sort** in C.
 - Analyze and compare the time complexities of both algorithms.
 - Understand and implement divide-and-conquer techniques.
-

Conclusion

- **Merge Sort** is an efficient, stable sorting algorithm with a guaranteed time complexity of $O(n \log n)$, but it requires extra space for merging.
 - **Quick Sort** is an efficient, in-place sorting algorithm with an average time complexity of $O(n \log n)$, but its worst-case performance can degrade to $O(n^2)$ if a poor pivot is chosen. Randomizing the pivot or using median-of-three selection can mitigate this issue.
 - The choice of sorting algorithm depends on the dataset size, memory constraints, and the need for stability.
-

Textbooks

1. **Introduction to Algorithms** by Cormen, Leiserson, Rivest, and Stein
 2. **Algorithms in C** by Robert Sedgewick
 3. **Data Structures and Algorithms** by Aho, Hopcroft, and Ullman
-

Online Reference Websites

- GeeksforGeeks - Merge Sort
 - GeeksforGeeks - Quick Sort
 - TutorialsPoint - Sorting Algorithms
 - Programiz - Merge Sort and Quick Sort
-

Expected Oral Questions

1. What is the time complexity of Merge Sort and Quick Sort in the worst case?
2. How does **Merge Sort** differ from **Quick Sort** in terms of recursive calls?
3. What is the pivot in **Quick Sort**, and how is it selected?
4. Why is **Merge Sort** considered stable, but **Quick Sort** is not?
5. How can you optimize **Quick Sort** to avoid the worst-case time complexity?

6. Explain the merging process in **Merge Sort**.
 7. What is the best-case scenario for **Quick Sort**?
 8. How does **Merge Sort** handle duplicate elements in the array?
 9. Why does **Quick Sort** perform better on average than **Merge Sort**?
 10. What would be the space complexity of **Merge Sort**?
-

FAQs in Interviews with Answers

1. **Q:** Which sorting algorithm is faster, Merge Sort or Quick Sort?
A: **Quick Sort** is generally faster because it sorts in-place and has better cache performance. However, **Merge Sort** has guaranteed $O(n \log n)$ time complexity, while **Quick Sort** can degrade to $O(n^2)$ in the worst case.
 2. **Q:** Can **Quick Sort** perform poorly?
A: Yes, if a bad pivot is chosen repeatedly (e.g., the smallest or largest element), **Quick Sort** can degrade to $O(n^2)$. Randomized pivot selection or the median-of-three method can help mitigate this issue.
 3. **Q:** What is the difference between stable and unstable sorting algorithms?
A: A **stable sorting algorithm** preserves the relative order of equal elements (i.e., the order in which they appeared in the input). **Merge Sort** is stable, while **Quick Sort** is unstable.
 4. **Q:** How does **Merge Sort** handle large datasets?
A: **Merge Sort** handles large datasets efficiently with $O(n \log n)$ time complexity, but it requires additional space proportional to the size of the array being sorted.
 5. **Q:** What is the key difference in space complexity between **Merge Sort** and **Quick Sort**?
A: **Merge Sort** has $O(n)$ space complexity due to the need for auxiliary arrays for merging, while **Quick Sort** has $O(\log n)$ space complexity for the recursive stack.
-

GATE MCQs with Answers

1. Which of the following sorting algorithms works on the principle of divide-and-conquer?

- A) Bubble Sort
- B) Insertion Sort
- C) Merge Sort
- D) Heap Sort

Answer: C) Merge Sort

2. What is the worst-case time complexity of Quick Sort?

- A) $O(n \log n)$
- B) $O(n^2)$
- C) $O(n)$
- D) $O(\log n)$

Answer: B) $O(n^2)$

3. In the worst case, the time complexity of Merge Sort is:

- A) $O(n \log n)O(n \log n)O(n \log n)$
- B) $O(n^2)O(n^2)O(n^2)$
- C) $O(n)O(n)O(n)$
- D) $O(\log n)O(\log n)O(\log n)$

Answer: A) $O(n \log n)O(n \log n)O(n \log n)$

4. What is the space complexity of Merge Sort?

- A) $O(1)O(1)O(1)$
- B) $O(n)O(n)O(n)$
- C) $O(n \log n)O(n \log n)O(n \log n)$
- D) $O(\log n)O(\log n)O(\log n)$

Answer: B) $O(n)O(n)O(n)$

5. The time complexity of Quick Sort in the average case is:

- A) $O(n \log n)O(n \log n)O(n \log n)$
- B) $O(n^2)O(n^2)O(n^2)$
- C) $O(\log n)O(\log n)O(\log n)$
- D) $O(n)O(n)O(n)$

Answer: A) $O(n \log n)O(n \log n)O(n \log n)$

6. Which of the following is true for Quick Sort?

- A) Quick Sort is a stable sorting algorithm.
- B) Quick Sort always requires $O(n \log n)O(n \log n)O(n \log n)$ time.
- C) Quick Sort is faster than Merge Sort for large datasets.
- D) Quick Sort requires additional space proportional to the size of the array.

Answer: C) Quick Sort is faster than Merge Sort for large datasets.

Laboratory 14: TREE

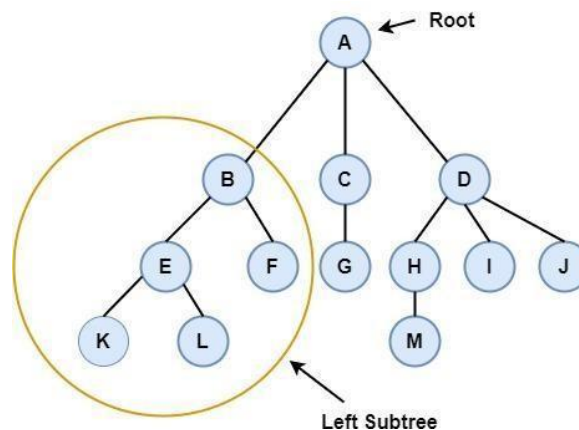
Objectives:

- Understand the basic concepts and properties of trees.
- Learn about different types of trees (binary trees, binary search trees, AVL trees, etc.).
- Implement tree operations such as insertion, deletion, traversal, and searching

Brief Theory:

Tree is a non-linear data structure in which data are stored in hierarchical manner. Tree can be defined as a finite set of one or more nodes such that:

1. There is a specially designated node called the root.
2. The remaining nodes are portioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree. T_1, \dots, T_n are called the subtrees of the root. For example, in the below tree A is root node, and node within the orange circle is a subtree of A. Similarly, {C, G} and {D, H, I, J, M} are two other subtree of root A.



Tree Terminologies:

Edge: In a tree data structure, the connecting link between any two nodes is called as edge. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

Parent Node: If N is immediate predecessor of a node X, then N is called the parent of X. In the above tree A is the parent node of B and C.

Sibling: Nodes which belong to same Parent are called as siblings. E and F are siblings in the above tree.

Internal Node: In tree data structure node which has at least one child is called as internal Node. A, B, C, D, E, F, G, H, I, J are the internal node in the above tree.

Level: The number of edges present in a path from a node to the root

is called the level of that node. Level of root is zero, if a node is at level l , its children are at level $l + 1$. For example, level of node H is 2.

Height: In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as height of that particular Node. Height of H is 1.

Tree Traversal: Visiting every node of the tree exactly once is known as traversal. Any hierarchical data structure like a tree can be traversed in different ways. Tree can be traversed in three ways; **1) In order traversal** in which for every node first left subtree will be visited then root and then right subtree will be visited, **2) Pre order traversal** in which for every node first root then left subtree then and then right subtree will be visited, **3) Post Order traversal** in which first left subtree, then right subtree and then root will be visited.

Binary Tree: Binary tree is a special kind of tree in which every node can have at most two children. Binary tree can be represented either using array or linked list. For array representation if a root node is stored at array index I , then its left child will be stored at index $2 \times I$ and the right child will be stored at index $2 \times I + 1$. For linked list representation along with data fields every node will contain two pointers, one will be linked with left child and another will be linked with right child.

Key types of trees include:

- **Binary Tree:** A tree where each node has at most two children.
- **Binary Search Tree (BST):** A binary tree where the left child contains nodes with values less than the parent node, and the right child contains nodes with values greater than the parent node.
- **AVL Tree:** A self-balancing binary search tree where the difference in heights of the left and right subtrees is no more than one.
- **B-Tree:** A self-balancing tree data structure that maintains sorted data and allows searches, insertions, deletions, and sequential access.

Binary search tree: Binary search tree is a binary tree with some restriction. In binary search tree value of root node will be larger than any node in the left subtree, and will be smaller than any node in the right subtree. Because of this restriction searching time of an element will be reduced to $(\log N)$.

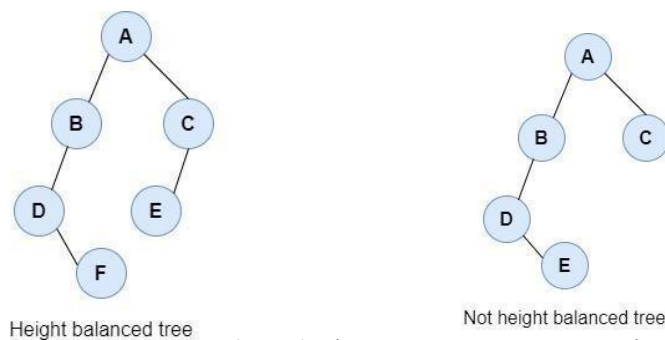
Almost Complete Binary Tree: A almost complete binary tree is actually a binary tree with some restriction in which all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

Heap: A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Heap is of two types, i.e., Max-Heap and Min-Heap.

Max-Heap: In a Max-Heap the key present at the root node must be maximum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Height Balancing Tree: Height balancing tree means the difference of the height of left subtree and the height of right subtree of any node is not more than 1. While inserting element in a height balancing tree for any node if the height difference of left and right subtree exceed 1 then appropriate action will immediately be taken to balance the height. The advantage of height balancing tree is that in worst case also searching for an element will take



balancing tree.

Advantages of Trees:

- **Hierarchical Data Representation:** Ideal for representing data with inherent hierarchy, like file systems or organizational structures.
- **Efficient Searching and Sorting:** Trees like BST allow efficient search, insertion, and deletion operations.
- **Dynamic Memory Allocation:** Nodes can be added or removed dynamically without significant reorganization.

Disadvantages of Trees:

- **Complexity:** Tree operations can become complex, especially for balancing trees like AVL or B-trees.
- **Memory Overhead:** Each node requires additional memory for pointers (especially in linked representations).

Outcomes:

- Students will be able to implement various types of trees and perform operations such as insertion, deletion, and traversal.
- Understand the applications of trees in solving complex problems.

Problems on Trees:

- Implement basic tree operations (insertion, deletion, searching).
- Perform tree traversals (inorder, preorder, postorder).

- Implement self-balancing trees like AVL trees.
- Solve problems using trees, such as expression tree evaluations, Huffman coding trees, and decision trees.

Syntax:

Binary Search Tree (BST):

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

// Insert a node
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    return root;
}

// Inorder Traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

Questions :

Q. No	Question	CO	BL
1	Binary Search Tree Formation: Write a C program to create a binary search tree (BST). The program should: <ol style="list-style-type: none"> 1. Accept a list of integers from the user. 2. Insert elements into the BST following BST properties. 3. Display the in-order traversal of the BST. 	1ADPC202_4	K3

2	Binary Search Tree Traversal: Write a C program to implement different types of binary search tree (BST) traversal. The program should: 1. Perform In-order, Pre-order, and Post-order traversal. 2. Display the results of all three traversals.	1ADPC202_4	K3
3	Search in Binary Search Tree: Write a C program to search for an element in a binary search tree (BST). The program should: 1. Insert elements into the BST. 2. Search for a specific element in the tree. 3. Display whether the element is found or not.	1ADPC202_4	K3

Expected Outcomes:

- Efficient manipulation of tree data structures for hierarchical data and complex problem-solving.
- Ability to handle various tree operations with proficiency.

Observations:

- Observe the performance differences between balanced and unbalanced trees.
- Analyze how different tree structures affect the efficiency of operations like search, insert, and delete.

Conclusion:

Trees are versatile data structures that offer efficient ways to organize and manipulate data, especially when hierarchical relationships exist. Understanding various tree types and their properties is crucial for effectively using them in applications ranging from database indexing to machine learning algorithms.

Textbooks:

- **"Data Structures and Algorithm Analysis in C"** by Mark Allen Weiss
- **"Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Online Reference Websites:

- GeeksforGeeks: Binary Trees
- GeeksforGeeks: Binary Search Trees

Expected Oral Questions:

- What are the differences between a binary tree and a binary search tree?
- How does an AVL tree maintain balance?
- Explain the time complexities of insertion, deletion, and search operations in a binary search tree.

FAQs in Interviews (with Answers):

1. **Q:** What is a binary search tree (BST)?
 - **A:** A BST is a binary tree in which each node has at most two children, with the left child containing values less than the node and the right child containing values

greater than the node. This property allows for efficient searching, insertion, and deletion operations.

2. **Q:** How do you perform an inorder traversal of a binary tree?
 - **A:** Inorder traversal involves visiting the left subtree, the root node, and then the right subtree. This traversal yields the nodes of a binary search tree in ascending order.
3. **Q:** What is the purpose of balancing in trees, and how do AVL trees achieve it?
 - **A:** Balancing ensures that the tree remains approximately balanced, preventing operations from degrading to $O(n)$ time. AVL trees maintain balance by ensuring that the height difference between the left and right subtrees of any node is no more than one, using rotations to maintain this property after insertions and deletions.
4. **Q:** What are some real-world applications of trees?
 - **A:** Trees are used in various applications such as database indexing (B-Trees), file systems (directory structure), expression evaluation (expression trees), and decision-making processes (decision trees).
5. **Q:** What is the difference between a tree and a graph?
 - **A:** A tree is a type of graph with no cycles and a hierarchical structure where each node has exactly one parent (except the root). In contrast, a graph is a more general structure that can have cycles and multiple connections between nodes without hierarchical constraints.

Gate Questions (with Answers):

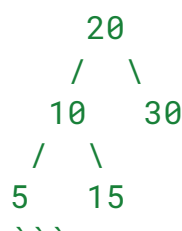
Binary Trees

Q1: The maximum number of nodes in a binary tree of height h is:

- (A) $2^{h+1} - 1$
 - (B) $2^{h+1} - 2^0$
 - (C) $2^h - 2^{h-1}$
 - (D) $2^{h+1} + 2^0$
- Answer:** (B) $2^{h+1} - 2^0$

Binary Search Trees

Q2: What is the in-order traversal of the following binary search tree?



- (A) 20, 10, 5, 15, 30
 - (B) 5, 10, 15, 20, 30
 - (C) 30, 20, 15, 10, 5
 - (D) 10, 5, 15, 20, 30
- Answer:** (B) 5, 10, 15, 20, 30

3. Complete Binary Tree

Q3: A complete binary tree with n nodes has how many levels (height h)?

- (A) $\lfloor \log_2 n \rfloor$

- (B) $\lceil \log_2 n \rceil \lceil \log_2 n \rceil$
 - (C) $\lfloor \log_2 n \rfloor + 1 \lfloor \log_2 n \rfloor + 1$
 - (D) $\lceil \log_2 n \rceil + 1 \lceil \log_2 n \rceil + 1$
- Answer:** (C) $\lfloor \log_2 n \rfloor + 1 \lfloor \log_2 n \rfloor + 1$
-

4. Tree Traversals

Q4: The pre-order traversal of a binary tree is: 1, 2, 4, 5, 3, 6, 7. What is its post-order traversal?

- (A) 4, 5, 2, 6, 7, 3, 1
- (B) 4, 2, 5, 6, 3, 7, 1
- (C) 4, 5, 3, 6, 7, 2, 1
- (D) 5, 4, 2, 7, 6, 3, 1

Answer: (A) 4, 5, 2, 6, 7, 3, 1

5. Heap Trees

Q5: A min-heap is represented as an array: [1, 3, 6, 5, 9, 8]. What is the height of this heap?

- (A) 2
- (B) 3
- (C) 4
- (D) 5

Answer: (A) 2

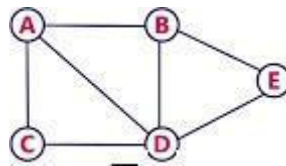
Laboratory 12: GRAPHS

Objectives:

To understand the representation, implementation and traversal of graph data structure.

Brief Theory:

A Graph (V, E) can be represented by set of vertices denoted by V and edges E . A graph is a collection of set of vertices connected by edges. Vertices of graph is called node, and these nodes are having data field which stores the data. Difference between tree and graph is tree is acyclic whereas graph may contain cycle. Many real-world problems can be represented using graph, and those problem can be solved using various graph algorithms. For example, let's say we want to find the shortest distance between two cities, we can represent every city as a node of a graph and the edge as the distance between two cities. After this representation we can apply shortest path finding algorithm of graph to determine the shortest path between two cities.



$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (B, E), (A, D), (A, C), (C, D), (D, E), (B, D)\}$$

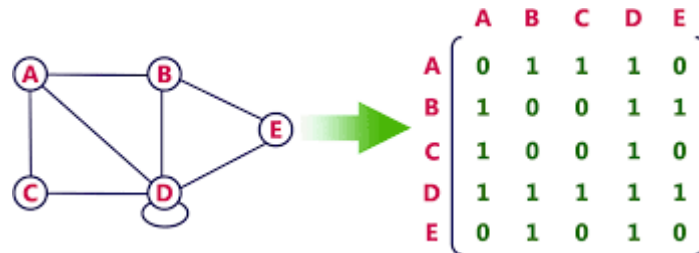
Types of Graphs:

1. **Directed graph:** In directed graph, edges have a specific direction. Edge (a, b) and edge (b, a) have different meaning in case of directed graph.
2. **Undirected:** In undirected graph, edges don't represent a specific direction. Edge (a, b) and edge (b, a) is equivalent in case of undirected graph.
3. **Weighted Graphs:** Weights are associated with the edges connecting two vertices. For example, if the two vertices are two cities, then edge between them may represent the distance between those two cities. If the nodes represent the two workstations in a network, then edge between two nodes may represent the cost to reach a workstation from other one.
4. **Unweighted Graphs:** Weights are not associated with the edges connecting two vertices. If there one undirected edge is present between two nodes that simply means that these two nodes are related.

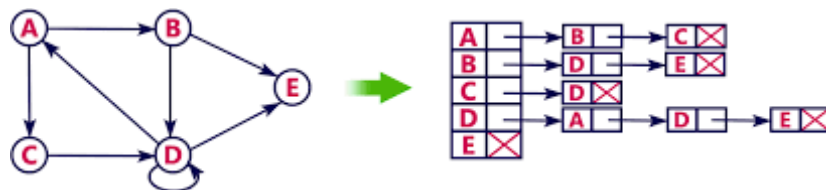
Representation of Graphs: These graphs can be stored in two ways:

1. Sequential or Adjacency matrix Representation
2. Linked list or Adjacency List Representation

Adjacency matrix Representation: The graph G can be represented in the form of matrix representation and the weights associated with the edge from one node (node i) to the other (node j) forms the entries (A_{ij}) of the matrix A.



Adjacency list Representation: An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.



Graph Traversal: The graphs can be traversed either by Breadth first search (BFS) or Depth first search (DFS). In BFS we traverse from the root node to the other nodes in a breadth first manner and uses queue to go to its next node. Depth first search (DFS) traverses the whole graph depth-wise and uses stack to get to the next nodes.

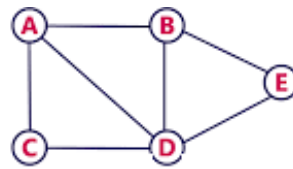
Depth First Search: A, B, E, D, C

Breadth First Search: A, B, C, D, E

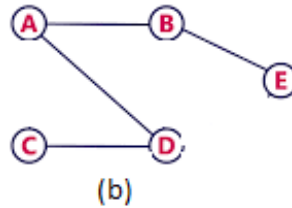
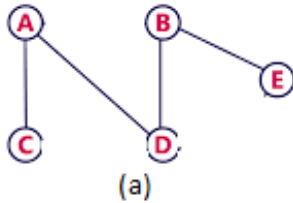
Spanning Tree:

A spanning tree of a graph is the subset of a graph which covers all the vertices with minimum number of possible edges. Spanning tree is used to find the minimum path connecting all nodes in a graph.

A spanning tree does not have cycles and it cannot be disconnected.



Graph G



Spanning trees of graph G

Observations/Outcomes:

1. Understand the properties of graph data structures.
2. Learning the different types of graph representations.
3. Implementation of DFS and BFS traversal.

Problems on:

Q. No	Question	CO	BL
1	Graph Representation (Adjacency Matrix): Write a C program to represent a graph using an adjacency matrix. Your program should: <ol style="list-style-type: none"> 1. Accept the number of vertices. 2. Accept edges and weights. 3. Display the adjacency matrix. 	1ADPC202_4	K3
2	Graph Representation (Adjacency List): Write a C program to represent a graph using an adjacency list. The program should: <ol style="list-style-type: none"> 1. Accept the number of vertices. 2. Accept edges. 3. Display the adjacency list representation of the graph. 	1ADPC202_4	K3

3	Graph Traversal (BFS): Write a C program to perform Breadth-First Search (BFS) on a graph. Your program should: 1. Accept a graph using adjacency list. 2. Perform BFS starting from a given vertex. 3. Display the traversal sequence.	1ADPC202_4	
4	Graph Traversal (DFS): Write a C program to perform Depth-First Search (DFS) on a graph. Your program should: 1. Accept a graph using adjacency list. 2. Perform DFS starting from a given vertex. 3. Display the traversal sequence.	1ADPC202_4	K3

Expected Outcomes

1. Efficient representation and traversal of graph data structures for solving network-related problems.
2. Ability to handle operations like searching, traversal, and shortest path computation with proficiency.

Observations

1. Observe the performance differences between adjacency matrix and adjacency list representations in terms of memory and speed.
2. Analyze how the choice of traversal algorithm (BFS or DFS) affects the efficiency of solving different problems like connected components or pathfinding.

Conclusion

Graphs are powerful data structures used to represent relationships between entities. Understanding graph representation and traversal techniques is essential for solving problems in areas like social networks, routing algorithms, and artificial intelligence.

Textbooks

1. **"Data Structures and Algorithm Analysis in C"** by Mark Allen Weiss
2. **"Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Online Reference Websites

1. **GeeksforGeeks: Graph Representations**
 2. **GeeksforGeeks: Graph Traversal (BFS and DFS)**
-

Expected Oral Questions

1. What is the difference between BFS and DFS?
 2. How is a graph represented using an adjacency matrix versus an adjacency list?
 3. Explain the time complexity of BFS and DFS.
 4. What are the real-world applications of graphs?
 5. How can you detect a cycle in a graph?
-

FAQs in Interviews (with Answers)

Q: What is a graph in data structures?

A: A graph is a collection of vertices (nodes) connected by edges. It is used to represent relationships between entities, such as a network of roads or friendships.

Q: What is the difference between BFS and DFS?

A: BFS explores all neighbors of a node before moving to the next level, making it suitable for finding shortest paths in unweighted graphs. DFS dives deep into one branch before backtracking, often used for connectivity checks and cycle detection.

Q: How do you represent a graph using an adjacency matrix?

A: An adjacency matrix is a 2D array where the cell at row i and column j is 1 if there is an edge between vertex i and vertex j , otherwise 0.

Q: What are some real-world applications of graphs?

A: Graphs are used in social networks (friend connections), web crawling (hyperlinks), transportation networks (routes), and AI (state-space search).

Q: How can you check if a graph is connected?

A: Perform a traversal (BFS or DFS) starting from any node. If all nodes are visited, the graph is connected; otherwise, it is not.