DATA STRUCTURES

(SE- SEM-III)

Compiled, reviewed and edited by:

Mrs. Foram Shah

Ms. Vinitta Sunish

Mr. Ashish Dwivedi

Ms.Soumyamol P.S

TCET, Mumbai

STUDY MATERIAL FOR FE SEM-III
Under the guidance of Dr. B. K. Mishra, Principal

4th Edition of revised syllabus, July 2024

Published by:
Thakur College of Engineering and Technology

PREFACE

This Resource book is a structured and guided learning material to be used effectively by students to learn the contents of syllabus based on the TCET Autonomy scheme. Top priority is given to the students' needs and contents are presented lecture-wise in order to help students keep track of their learning after every lecture. The resource book emphasizes on the fundamental definitions with a continuity of details easy to follow finally rendering to the practical applications of the topic. The module is written sequentially starting from prerequisite to theoretical background eventually highlighting the concepts and covering all the topics in detail. Large number of short answer questions, long answer questions and practice questions are included which will be helpful in preparing for examinations.

This is a comprehensive (structured and guided) resource book covering the Data Structures syllabus as per TCET Autonomy scheme. The details of content is presented module wise below:

Module 1: Introduction to Java: Data Types, Operators, Control structure, Array, String & vectors, classes & objects, Methods. Introduction to Data Structure :Types of data Structures, Abstract data type, Importance of ADT, Operations on data structures.

Module 2: Stack: Operations on stack, array implementation of stack, applications of stack.

Queue: Operations on queue, array implementation of queue, Types of queues: circular queue, priority queue, double ended queue.

Module 3: Linked lists

Linked list: Operations on linked list, Types of linked lists: Single linked list, Double Linked list, Circular linked list,

Implementation of linked list, stack implementation using linked list, queue implementation using linked list, Applications of linked list.

Module 4: Trees: Terminologies, Binary tree and its types, Binary tree operations and implementation, Tree traversing techniques, Expression tree, AVL tree, B tree, B+ tree, Representing Lists as binary tree.

Module 5: Graph: Terminologies, Cycles in graph, Graph representation: Matrix and Adjacency list, Graph traversing techniques: BFS, DFS, Applications of graph, Graph Application: Topological sort.

Module 6: Program Specific Applications, Domain Specific Applications.

General Guidelines for Learners:

1. Resource book is for a structured and guided teaching learning process and therefore learners are recommended to come with the same in every lecture.
2. While conducting the teaching learning process this resource book will be extensively used.
3. Resource book is framed to improve the understanding of subject matter at depth and therefore the learners are recommended to take up all the module contents, home assignments and exercise seriously.
4. A separate notebook should be maintained for every subject.
5. Lectures should be attended regularly. In case of absence, topics done in the class should be referred from the module before attending the next lecture.
6. Motivation, weightage and pre-requisite in every chapter have been included in order to maintain continuity and improve the understanding of the content to clarify topic requirements from exam point of view.
7. For any other additional point related to the topic instructions will be given by the subject teacher from time to time.

Subject Related Guidelines:

1. The subject is pure as well as applied in nature and it requires thorough understanding of subject
2. Questions are expected from all modules and learners are instructed not to leave any module in option.
3. Theory paper will be of 60 marks, Internal assessments of 20 marks, practical examination will be of 25 marks and continuous and systematic work during the term (Term work) will be of 25 Marks. Importance should be given to term work and all internal assessment/continuous assessment for improving overall percentage in examination.
4. Practice questions should be solved sincerely in order to enhance confidence level and to excel in the end semester examination.

Exam Specific Guidelines:

1. All modules are equally important from an examination point of view.
2. Wherever applicable chemical reaction, mechanism and neat labeled diagram must be provided for writing effective answers.
3. Proper time management is required for completing the question paper within a time frame.
4. Read the question paper thoroughly first then choose the questions. Attempt the one that you know the best first but do not change the internal sequence of the sub questions.
5. For further subject clarification/ doubt in the subject, learners can contact the subject teacher.

Guidelines for Writing Quality Answer:

1. Write content as per marks distribution.
2. Highlight the main points.
3. Write necessary content related to the point.
4. Draw neat and labeled diagrams wherever necessary.
5. While writing distinguishing points, write double the number of points as per the marks given, excluding the example.

S.E. Semester –III

Choice Based Credit Grading Scheme with Holistic and Multidisciplinary Education (CBCGS-HME 2023)
Proposed TCET Autonomy Scheme (w.e.f. A.Y. 2024-25)

<table>
<tr><td colspan="8">B.E. (Computer Engineering)</td><td colspan="3">S.E. SEM : III</td></tr>
<tr><td colspan="8">Course Name : Data Structures Using Java</td><td colspan="3">Course Code : PCC-CS302</td></tr>
<tr><td colspan="5">Teaching Scheme (Program Specific)</td><td colspan="6">Examination Scheme (Academic)</td></tr>
<tr><td colspan="5">Modes of Teaching / Learning / Weightage</td><td colspan="6">Modes of Continuous Assessment / Evaluation</td></tr>
<tr><td colspan="5" rowspan="2">Hours Per Week</td><td colspan="3">Theory (100)</td><td rowspan="2">Practical/Oral/ Presentation (25)</td><td rowspan="2">Term Work (25)</td><td rowspan="2">Total</td></tr>
<tr><td colspan="2">40/20</td><td>60/30</td></tr>
<tr><td>Theory</td><td>Tutorial</td><td>Practical</td><td>Contact Hours</td><td>Credits</td><td colspan="2">IA</td><td>ESE</td><td>PR/OR</td><td>TW</td><td rowspan="2">150</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>ISE</td><td>IE</td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>1</td><td>2</td><td>6</td><td>5</td><td>20</td><td>20</td><td>60</td><td>25</td><td>25</td><td></td></tr>
<tr><td colspan="11">IA: In-Semester Assessment - Paper Duration – 1 Hour ESE: End Semester Examination - Paper Duration – 2/1 Hours
The weightage of marks for continuous evaluation of Term work/Report: Formative (40%), Timely completion of practical (40%) and Attendance / Learning Attitude (20%)</td></tr>
<tr><td colspan="11">Prerequisite: Computer Basics, Object Oriented Programming Languages</td></tr>
</table>

<u>Course Objective:</u> The course intends to deliver the fundamentals of data structures by providing a platform to learn, compare and apply them in real world scenario.

<u>Course Outcomes:</u> Upon completion of the course students will be able to:

Sr. No.	Course Outcomes	Cognitive levels of attainment as per Bloom's Taxonomy	PO Mapping	PSO Mapping	PI	Module wise % weightage in exam
1	Understand and Apply OOPM to linear and non-linear data structures.	L1, L2,L3	1,2,3,4,5,6,9	1,2,3	1.1.1,1.3.1, 2.1.3	8
2	Understand and Apply operations like insertion, deletion, searching and traversing on stack and queue data structure.	L1, L2, L3,L4	1,2,3,4,5,9	1,2,3	1.1.1,1.3.1, 2.1.3,3.1.1	10
3	Understand and Apply operations like insertion, deletion, searching and traversing on linked list data structure.	L1, L2, L3,L4	1,2,3,4,5,9	1,2,3	1.1.1,1.3.1, 2.1.3,3.1.1	12
4	Understand and Apply operations like insertion, deletion, searching and traversing on tree data structure.	L1, L2, L3,L4	1,2,3,4,5,9	1,2,3	1.1.1,1.3.1, 2.1.3,3.1.1	12
5	Understand and Apply operations like insertion, deletion, searching and traversing on graph data structure.	L1, L2, L3.L4	1,2,3,4,5,9	1,2,3	1.1.1,1.3.1, 2.1.3,3.1.1	12

6	Analyze usage of data structure in different domains	L1, L2, L3, L4	1,2,3,4,5,11,12	1,2,3	1.1.1,1.3.1, 2.1.3,3.1.1	6

Detailed Syllabus:

Module No.	Topics	Hrs.	Cognitive levels of attainment as per Bloom's Taxonomy
1	**Introduction to Data Structure** **Introduction to Java:** Data Types, Operators, Control structure, Array, String & vectors, classes & objects, Methods. **Introduction to Data Structure :** Types of data Structures, Abstract data type, Importance of ADT, Operations on data structures.	10	L1, L2,L3
2	**Stacks and Queues** **Stack**: Operations on stack, array implementation of stack, applications of stack. **Queue:** Operations on queue, array implementation of queue, Types of queues: circular queue, priority queue, double ended queue.	8	L1, L2, L3,L4
3	**Linked lists** **Linked list**: Operations on linked list, Types of linked lists: Single linked list, Double Linked list, Circular linked list, Implementation of linked list, stack implementation using linked list, queue implementation using linked list, Applications of linked list.	8	L1, L2, L3,L4
4	**Tree** **Trees:** Terminologies, Binary tree and its types, Binary tree operations and implementation, Tree traversing techniques, Expression tree, AVL tree, B tree, B+ tree, Representing Lists as binary tree.	9	L1, L2, L3,L4
5	**Graphs** **Graph:** Terminologies, Cycles in graph, Graph representation: Matrix and Adjacency list, Graph traversing techniques: BFS, DFS, Applications of graph, Graph Application: Topological sort.	8	L1, L2, L3,L4
6	**Application of Data Structure** Program Specific Applications, Domain Specific Applications	2	L1, L2, L3, ,L4
	Total Hours	**45**	

Books and References:

Sr. No.	Title	Authors	Publisher	Edition	Year
1	Data Structures with Java	Johan R Hubbard	Tata McGraw-Hill	Second Edition	2012
2	Data Structures: A Pseudocode Approach with Java	Richard F. Gilberg & Behrouz A., Forouzan	CENGAGE Learning	Second Edition	2011
3	Data Structures Using Java	Aaron M Tenenbaum, Yedidyah Langsam, Moshe J Augenstein	Pearson	Second Edition	2006

Online References:

Sr. No.	Website Name	URL	Modules Covered
1	www.w3schools.com	https://www.w3schools.com/java/	M1
2	www.geeksforgeeks.org	https://www.geeksforgeeks.org/stack-data-structure/	M2
3	www.studytonight.com	https://www.studytonight.com/data-structures/introduction-to-data-structures	M1- M6
4	www.w3schools.in	https://www.w3schools.in/category/data-structures-tutorial/	M1- M6

List of Practical/ Experiments:

Practical Number	Type of Experiment	Practical/ Experiment Topic	Hrs.	Cognitive levels of attainment as per Bloom's Taxonomy
1 A	**Basic Experiments**	Build a java program to demonstrate data types and operators in Java.	2	L1, L2, L3,L4
1 B		Build a java program to demonstrate control statements in Java.	2	L1, L2, L3,L4
2		Build a Program for stack using an array (Menu driven program)	2	L1, L2, L3,L4
3		Build a Program for Queue using an array. (Menu driven program)	2	L1, L2, L3,L4
4	**Design Experiments**	Develop a code for circular queue. (Menu driven)	2	L1, L2, L3,L4
5		Develop a code for Single Linked List. (Menu driven program)	2	L1, L2, L3,L4
6		Develop a code for Doubly linked list. (Menu driven program)	2	L1, L2, L3,L4
7		Develop a code for Binary Search Tree (Menu driven program)	2	L1, L2, L3,L4
8		Develop a code for AVL tree (Menu driven program)	2	L1, L2, L3,L4
9		Develop a code for BFS. (Menu driven program)	2	L1, L2, L3,L4
10		Develop a code for DFS. (Menu driven program)	2	L1, L2, L3,L4
11	**Advanced Experiments**	Develop a code for circularly linked doubly linked list.	2	L1, L2, L3,L4
12	**Mini/Minor Projects/ Seminar/ Case Studies**	**Case study:** 1.Selection of Data Structure 2. Complete binary tree from its link list representation 3. Android multiple tasks/ Multiple activity using Stack 4. Priority queue in bandwidth management **Mini Project:** 1. Attendance Management System 2. Tic-Tac-Toe Game 3. Brick Breaker Game 4. Currency Converter 5. Word Counter 6. Grading System in Java	6	L1,L2,L3,L4, L5, L6
		Total Hours	**30**	

List of Tutorials:

Sr. No.	Topic	Hrs.	Cognitive levels of attainment as per Bloom's Taxonomy
1	Various ways to accept data through keyboard for 1D array & 2D array	1	L1, L2, L3
2	Class creation including members and methods, accepting and displaying details for single object.	1	L1, L2, L3, L4
3	Constructor and constructor overloading	1	L1, L2, L3, L4
4	String and String Buffer	1	L1, L2, L3, L4
5	Inheritance use super keyword.	1	L1, L2, L3, L4
6	Interface implementation.	1	L1, L2, L3, L4
7	try, catch, throw, throws and finally	1	L1, L2, L3, L4
8	Creating user defined package	1	L1, L2, L3, L4
9	Multithreading using Thread class and Runnable interface	1	L1, L2, L3, L4
10	Java Frameworks for different operations	1	L1, L2, L3, L4

INDEX

Module-01
Introduction to Data Structure

Motivation:

To store/organize a given data in the memory of the computer so that each subsequent operation (query/update) can be performed quickly. Also to know about the object oriented programming concepts for Class and Object use. To understand the Data Types, Variables, Operators, Control Structures and many different concepts of Java Programming to make programming easier.

Syllabus:

Lecture	Content	Duration	Self-Study
1	Introduction to Java	1 Lecture	1 hour
2	Data Types & Operators	1 Lecture	1 hour
3	Control Structure	1 Lecture	1 hour
4	Array	1 Lecture	1 hour
5	String & vectors	1 Lecture	1 hour
6	classes & objects, Methods	1 Lecture	1 hour
7	Types of data Structures	1 Lecture	1 hour
8	Abstract data type	1 Lecture	1 hour
9	Importance of ADT	1 Lecture	1 hour
10	Operations on data structures.	1 Lecture	1 hour

Learning Objectives:

- Students shall know about Data structure basics.
- Students shall know ADT.
- Students shall be able to recall operations on DS.

Theoretical Background:

A good command on Programming in C is required. Where in it involves arrays, Recursion, Linked lists

Course Content:

Lecture 1

Introduction to Java:

Java was created by Sun Microsystems in 1991 and was later acquired by Oracle Corporation. The language was developed by James Gosling and Patrick Naughton. Java is known for its simplicity, which facilitates easy writing, compiling, and debugging of programs. Additionally, Java promotes modular programming and code reuse, making it a versatile tool for developers.

Why Use Java?

- Java is compatible with various platforms, including Windows, Mac, Linux, and Raspberry Pi.
- It ranks among the most widely-used programming languages globally and has a significant presence in today's job market.
- Its ease of learning and simplicity make it an accessible choice for many developers.
- Java is open-source and free to use, offering security, speed, and robustness.
- With extensive community support from millions of developers, Java benefits from a collaborative ecosystem.
- As an object-oriented language, Java provides a well-organized structure for programs and supports code reuse, which helps reduce development costs.
- Its similarities to C++ and C# make transitioning between these languages relatively straightforward for programmers.

Java is categorized as both a compiled and interpreted language. Initially, Java source code is compiled into bytecode, which is then interpreted by the Java Virtual Machine (JVM). The JVM translates this bytecode into platform-specific machine code, making Java known for its platform independence. The name "Java" itself is inspired by the Indonesian island of Java. The term also traces its roots to the coffee known as Java coffee. James Gosling, one of Java's creators, chose the name while enjoying a cup of coffee near his office. Initially, the language was named "Oak," but due to the existence of a company called Oak Technologies, the development team decided to rename it. They considered several names, including Silk, Revolutionary, Dynamic, DNA, and Jolt. Ultimately, they chose "Java" for its uniqueness and positive reception.

Garbage Collector

In Java, programmers do not have the ability to manually delete objects. Instead, the Java Virtual Machine (JVM) employs a Garbage Collector to manage memory. This Garbage Collector automatically reclaims memory from objects that are no longer referenced, simplifying memory management for developers. However, programmers need to be mindful of their code and the objects they use, as the Garbage Collector cannot free memory for objects that are still being referenced. Garbage collection is thus an automated process, but awareness of object references remains important.

To illustrate, consider a "Car" class in Java.

This class can have various attributes such as model, color, engine power, top speed, and year of manufacture, which are collectively known as the properties of the Car class.

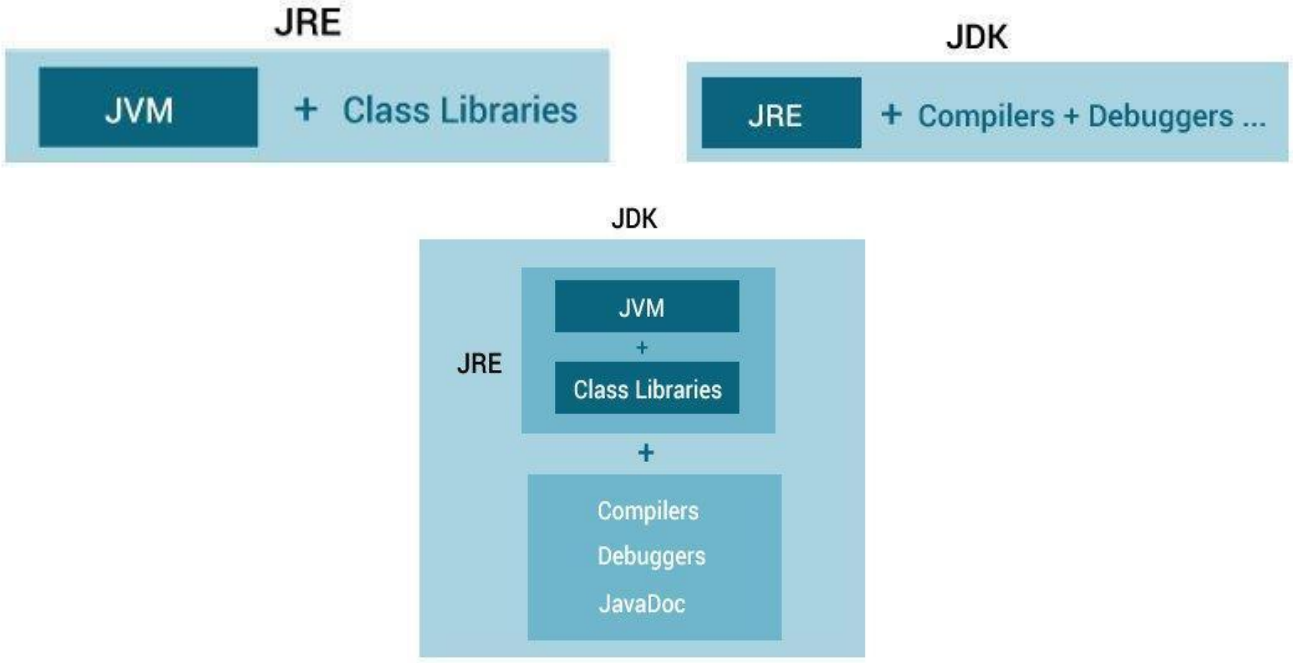
Additionally, methods like start, stop, and move define the functionalities of the Car class.

Memory is not allocated when the class is defined but is allocated when an object of that class is created, such as a new car object.

Java Virtual Machine (JVM): The JVM is an abstract computing machine designed to provide a runtime environment where Java bytecode can be executed. It serves as a specification that can be implemented on various hardware and software platforms, enabling Java programs to run consistently across different systems.

Java Development Kit (JDK): The JDK is a comprehensive suite tailored for software developers. It includes essential tools such as the Java compiler, Javadoc, Jar, and a debugger, which are necessary for developing Java applications.

Java Runtime Environment (JRE): The JRE provides the core Java libraries and components needed to run Java applications. It is designed for end-users and can be considered a subset of the JDK, focusing specifically on the execution of Java programs rather than development tools.



The Four Fundamental Principles of Object-Oriented Programming:

1. Inheritance: Inheritance allows one class to inherit the properties and behaviors of another class, referred to as the parent or superclass. This promotes code reuse and can facilitate runtime polymorphism, where a subclass can override methods of its superclass.

2. Polymorphism: Polymorphism refers to the ability to perform a single action in different ways. In Java, this is achieved through method overloading (where multiple methods in the same class have the same name but different parameters) and method overriding (where a subclass provides a specific implementation of a method already defined in its superclass).

3. Abstraction: Abstraction involves hiding the internal details of an implementation and exposing only the necessary functionalities to users. In Java, abstraction is achieved using interfaces and abstract classes. For instance, when using an ATM, we interact with the machine's interface without needing to understand its internal workings, which exemplifies abstraction in everyday life.

4. Encapsulation: Encapsulation is the practice of bundling data and methods that operate on that data within a single unit, typically a class. This concept is central to Java, where classes encapsulate data and methods. For example, a JavaBean demonstrates encapsulation by keeping its data members private and providing access through public methods. This is similar to how a capsule encloses medicine, keeping its contents protected and contained.

Applications of Java Programming Language:

As Java evolved, it gained immense popularity due to its ability to support a variety of platforms and configurations. One of Java's most notable features is its "write once, run anywhere" capability. Over time, Java has introduced several versions, now known as Java SE (Standard Edition), Java ME (Micro Edition), and Java EE (Enterprise Edition). Here are some key applications and features of the Java programming language:

Multithreading: Java simplifies the creation of applications that can perform multiple tasks simultaneously through its multithreading capabilities. This is particularly useful for developing interactive applications that require smooth, concurrent operations.

High Performance: Java's architecture is optimized to minimize overhead during application execution. The Just-In-Time (JIT) compiler further enhances performance by compiling methods on demand, only when they are called, which accelerates the execution of applications.

Distributed Computing: Java supports the development of distributed applications through technologies like Remote Method Invocation (RMI) and Enterprise Java Beans (EJB). These features enable Java applications to run across multiple systems connected via the internet, allowing objects on one Java Virtual Machine (JVM) to interact with those on a remote JVM.

Dynamic: Compared to C and C++, Java is more dynamic. It can adapt to evolving environments by carrying a significant amount of runtime information. This flexibility allows for runtime verification of objects, making Java more adaptable and responsive to changes during execution.

Hello World Program:

```
Import java.io.*;
public class Main
{
	public static void main(String[] args)

	{
		System.out.println("Hello World"); // to print the output
	}

}
```

Explanation:

Comments: In Java, comments are used to explain and annotate the code. They are not executed by the compiler. There are two types of comments:

- **Single-line comments:** Begin with // and extend to the end of the line.
- **Multi-line comments:** Enclosed between /* and */ and can span multiple lines.

import java.io.*;: The import keyword is used to include classes from the io package. This package provides classes for input and output operations, such as reading from and writing to files.

public class Main: Defines a class named Main. In Java, all data and methods are encapsulated within classes. The class serves as a blueprint for creating objects and defining their behavior.

public static void main(String[] args): This is the main method where the execution of the program begins:

- **public**: An access modifier that makes the method accessible from outside the class.
- **static**: Indicates that the method can be called without creating an instance of the class.
- **void**: Specifies that the method does not return any value.
- **main**: The name of the method which serves as the entry point of the program.
- **String[] args**: An array of String arguments that can be passed to the method from the command line.

System.out.println("Hello World");: This command prints the string "Hello World" to the console. The println method outputs text followed by a newline character, moving the cursor to the next line.

Summary:

- Java is a class-based, object-oriented programming language designed to reduce implementation dependencies to the minimum. It is a high-level language with a syntax that is similar to C and C++, making it relatively easy to understand.

- One of Java's key features is its platform independence, which supports the "write once, run anywhere" principle. This characteristic makes Java highly versatile and distinguishes it from many other programming languages. Java is known for being portable, simple, secure, and robust, which makes it well-suited for handling large codebases.

- The Java Development Kit (JDK) provides the tools needed to develop applications, applets, and components using Java. It includes a range of utilities for building, testing, and running Java programs on the Java platform.

- The Java Runtime Environment (JRE) encompasses several components, including the Java Virtual Machine (JVM), Java class libraries, and the Java class loader. Both the JDK and JRE depend on the JVM, which is responsible for translating bytecode into machine-specific code. While the JVM is platform-dependent, it handles various tasks such as memory management and can execute programs converted to Java bytecode from other languages.

- Java's garbage collector automatically reclaims memory from objects that are no longer referenced, aiding in efficient memory management. Additionally, the classpath is a designated file path where the Java runtime and compiler search for `.class` files.

- Java is utilized in various applications, including multithreaded, distributed, dynamic, and high-performance environments.

Lecture 2

Data Types:

For example:

```
int a = 5;              // Integer (whole number)
float b = 5.99f;    // Floating point number
char c = 'D';         // Character
boolean d = true;       // Boolean
String e = "Hello";     // String
```

Data types are divided into two groups:

- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter)

Data Type	Size	Description
byte	**1 byte**	**Stores whole numbers from -128 to 127**
short	**2 bytes**	**Stores whole numbers from -32,768 to 32,767**
int	**4 bytes**	**Stores whole numbers from -2,147,483,648 to 2,147,483,647**
long	**8 bytes**	**Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**
float	**4 bytes**	**Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits**
double	**8 bytes**	**Stores fractional numbers. Sufficient for storing 15 decimal digits**
boolean	**1 bit**	**Stores true or false values**
char	**2 bytes**	**Stores a single character/letter or ASCII values**

Operators :

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc. There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr++ expr--*
	prefix	*++expr --expr +expr -expr ~ !*
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Example 1:

```
public class Unary
{
    public static void main(String args[])
            {
                    int x=10;
                    System.out.println(x++);//10 (11)
                    System.out.println(++x);//12
                    System.out.println(x--);//12 (11)
                    System.out.println(--x);//10
            }
}
```

Output:

```
10
12
12
10
```

Example 2:

```
public class Unary2
{
    public static void main(String args[])
    {
        int a=10;
        int b=-10;
        boolean c=true;
        boolean d=false;
        System.out.println(~a);
        //-11 (minus of total positive value which starts from 0)
        System.out.println(~b);//9 (positive of total minus, positive starts from 0)
        System.out.println(!c);//false (opposite of boolean value)
        System.out.println(!d);//true
    }
}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Example 3:

```
public class Arithmetic
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        System.out.println(a+b);//15
        System.out.println(a-b);//5
```

```
            System.out.println(a*b);//50
            System.out.println(a/b);//2
            System.out.println(a%b);//0
            }
    }
```

Output:

```
15
5
50
2
0
```

Java Left & Right Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times whereas the Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Example 4:

```
public class LeftShift
{
    public static void main(String args[])
    {
            System.out.println(10<<2);//10*2^2=10*4=40
            System.out.println(10<<3);//10*2^3=10*8=80
            System.out.println(20<<2);//20*2^2=20*4=80
            System.out.println(15<<4);//15*2^4=15*16=240
    }
}
```

Output:

```
40
80
80
240
```

Example 5:

```
public class RightShift
{
    public static void main(String args[])
    {
```

```
            System.out.println(10>>2);//10/2^2=10/4=2
            System.out.println(20>>2);//20/2^2=20/4=5
            System.out.println(20>>3);//20/2^3=20/8=2
    }
}
```

Output:

```
2
5
2
```

Example 6:

```
public class Shift
{
    public static void main(String args[])
    {
                //For positive number, >> and >>> works same
        System.out.println(20>>2);
        System.out.println(20>>>2);
                //For negative number, >>> changes parity bit (MSB) to 0
        System.out.println(-20>>2);
        System.out.println(-20>>>2);
    }
}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example:
Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true whereas the bitwise & operator always checks both conditions whether first condition is true or false.

Example 7:

```
public class Logical_Bitwise_AND
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
```

```
        int c=20;
        System.out.println(a<b&&a<c);//false && true = false
        System.out.println(a<b&a<c);//false & true = false
    }
}
```

Output:

```
false
false
```

Java OR Operator Example:

Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false whereas the bitwise | operator always checks both conditions whether the first condition is true or false.

Example 8:

```
public class Logical_Bitwise_OR
{
    public static void main(String args[])
    {
            int a=10;
            int b=5;
            int c=20;
            System.out.println(a>b||a<c);//true || true = true
            System.out.println(a>b|a<c);//true | true = true
            //|| vs |
        System.out.println(a>b||a++<c);//true || true = true
        System.out.println(a);//10 because second condition is not checked
        System.out.println(a>b|a++<c);//true | true = true
        System.out.println(a);//11 because second condition is checked
    }
}
```

Output:

```
true
true
true
10
true
11
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Example 9:

```
public class Ternary
{
    public static void main(String args[])
    {
            int a=2;
            int b=5;
            int min=(a<b)?a:b;
            System.out.println(min);
    }
}
```

Output:

2

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Example 10:

```
public class Assignment
{
    public static void main(String args[])
    {
            int a=10;
            int b=20;
            int x=10;
            a+=4;//a=a+4 (a=10+4)
            b-=4;//b=b-4 (b=20-4)
            System.out.println(a);
            System.out.println(b);
            x+=3;//10+3
            System.out.println(x);
            x-=4;//13-4
            System.out.println(x);
            x*=2;//9*2
            System.out.println(x);
            x/=2;//18/2
            System.out.println(x);
    }
}
```

Output:

14
16
13
9
18
9

Lecture 3

Control Structure:

Java Control Statements:

In Java, the compiler executes code sequentially from top to bottom, following the order in which statements appear. However, Java includes control flow statements that allow for more flexible execution paths, which are essential for directing the flow of a program. These control flow statements are fundamental to programming in Java and help manage the execution flow efficiently.

Java provides three main types of control flow statements:

1. **Decision-Making Statements**
    - **if Statements**
    - **switch Statements**
2. **Loop Statements**
    - **do-while Loop**
    - **while Loop**
    - **for Loop**
    - **for-each Loop**
3. **Jump Statements**
    - **break Statement**
    - **continue Statement**
    - **return Statement**

Decision-Making Statements

Decision-making statements determine which code block should be executed based on certain conditions. These statements evaluate Boolean expressions and control the program flow based on whether the conditions are true or false. In Java, there are two primary decision-making statements: the if statement and the switch statement.

If Statement

The if statement is used to evaluate a condition and direct the flow of execution based on the result. It checks a Boolean expression, and depending on whether it evaluates to true or false, the program either enters a specific block of code or skips it. Java supports several variations of the if statement:

- **Simple if Statement:** The basic form of the if statement that executes a block of code if the condition is true.
- **if-else Statement:** Provides an alternative block of code to execute if the condition is false.
- **if-else-if Ladder:** Allows for multiple conditions to be checked sequentially, with different blocks of code executed based on which condition is true.
- **Nested if Statement:** An if statement placed inside another if statement, allowing for more complex decision-making scenarios.

Here is a basic example of a simple if statement in Java:

```
int number = 10;
if (number > 0)
{
        System.out.println("The number is positive.");
 }
```

In this example, the if statement checks if the variable number is greater than 0. If the condition is true, the message "The number is positive." is printed.

Example 11:

```
public class Student
{
    public static void main(String[] args)
    {
            int x = 10;
            int y = 12;
            if(x+y > 20)
            {
                    System.out.println("x + y is greater than 20");
            }
    }
}
```

Output:

x + y is greater than 20

if-else statement:

The if-else statement extends the basic if statement by introducing an additional block of code known as the else block. The else block is executed if the condition in the if block evaluates to false, providing an alternative path of execution.

Consider the following example.

Example 12:

```
public class Student
{
    public static void main(String[] args)
    {
            int x = 10;
            int y = 12;
            if(x+y < 10)
            {
                    System.out.println("x + y is less than     10");
            }
            else
```

```
            {
                System.out.println("x + y is greater than 20");
            }
        }
}
```

Output:

x + y is greater than 20

if-else-if ladder:

The if-else-if statement is a control flow structure that allows for multiple conditions to be evaluated in sequence. It consists of an initial if statement followed by one or more else-if statements. Each else-if statement checks an additional condition, and only the block of code associated with the first true condition is executed. An optional else block can be included at the end to handle cases where none of the previous conditions are met.

This construct effectively creates a decision tree where the program enters the block of code corresponding to the first condition that evaluates to true. If none of the if or else-if conditions are true, the else block (if present) will be executed.

Consider the following example.

Example 13:

```
public class Student
{
    public static void main(String[] args)
    {
        String city = "Delhi";
        if(city == "Meerut")
        {
            System.out.println("city is meerut");
        }
        else if (city == "Noida")
        {
            System.out.println("city is noida");
        }
        else if(city == "Agra")
        {
            System.out.println("city is agra");
        }
        else
        {
            System.out.println(city);
```

```
            }
        }
    }
```

Output:

Delhi

Nested if-statement:

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Example 14:

```
public class Student
{
    public static void main(String[] args)
    {
            String address = "Delhi, India";
            if(address.endsWith("India"))
            {
                    if(address.contains("Meerut"))
                    {
                            System.out.println("Your city is Meerut");
                    }
                    else if(address.contains("Noida"))
                    {
                            System.out.println("Your city is Noida");
                    }
                    else
                    {
                            System.out.println(address.split(",")[0]);
                    }
            }
            else
            {
                    System.out.println("You are not living in India");
            }
    }
}
```

Output:

Delhi

Switch Statement:

The switch statement in Java is a control flow mechanism that provides an alternative to the if-else-if chain. It executes a block of code based on the value of a variable, which is compared against several possible case values. Using switch can simplify code and enhance readability, especially when dealing with multiple conditions that depend on the same variable.

Key Points about the switch Statement:

- **Supported Types:** The variable in the switch statement can be of types such as int, short, byte, char, or enum. Starting with Java 7, String type variables are also supported.
- **Unique Cases:** Each case label must be unique within the switch block.
- **Default Case:** The default case is optional and is executed if none of the specified case values match the variable's value.
- **Break Statement:** The break statement is used to terminate the switch block. If break is omitted, the execution will "fall through" to subsequent cases, which might not always be desired.
- **Type Consistency:** The case expressions must match the type of the switch variable and must be constant values.

Here is an example illustrating how a switch statement works:

Example 15:

```
public class Student
{
    public static void main(String[] args)
    {
            int num = 2;
            switch (num)
            {
                    case 0:
                            System.out.println("number is 0");
                            break;
                    case 1:
                            System.out.println("number is 1");
                            break;
                    default:
                            System.out.println(num);
            }
    }
}
```

Output:

2

Loop Statements:

In programming, loops are used to execute a block of code repeatedly as long as a specified condition remains true. Java provides three primary types of loops, each with its own syntax and condition-checking behavior:

1. **For Loop**
2. **While Loop**
3. **Do-While Loop**

Java For Loop

The for loop in Java is similar to the for loop in C and C++. It allows you to initialize the loop variable, check the loop condition, and update the loop variable all within a single line. This type of loop is ideal when the number of iterations is known beforehand.

Here's a basic example of a for loop in Java:

Example 16:

```
public class Calculation
{
    public static void main(String[] args)
    {
            //      TODO Auto-generated method stub
            int sum = 0;
            for(int j = 1; j<=10; j++)
            {
                    sum = sum + j;
            }
    System.out.println("The sum of first 10 natural numbers is " + sum);
    }
}
```

Output:

The sum of first 10 natural numbers is 55

Java for-each loop:

Java offers an enhanced for loop, often referred to as the "for-each" loop, designed to simplify the iteration over arrays and collections. This loop eliminates the need to manually update the loop variable and provides a cleaner and more readable way to traverse through elements.

Example 17:

```
public class Calculation
{
    public static void main(String[] args)
    {
            // TODO Auto-generated method stub
            String[] names = {"Java","C","C++","Python","JavaScript"};
            System.out.println("Printing the content of the array names:\n");
            for(String name:names)
```

```
            {
                System.out.println(name);
            }
    }
}
```

Output:

Printing the content of the array names:

Java
C
C++
Python
JavaScript

Java while loop:

The while loop in Java is used to execute a block of code repeatedly as long as a specified condition remains true. It is particularly useful when the number of iterations is not known beforehand. Unlike the for loop, the while loop does not handle initialization, condition checking, and incrementing/decrementing within a single line of code. Instead, these actions are managed separately. The while loop is also known as an entry-controlled loop because it evaluates the condition before executing the loop body.

Here is an example demonstrating the use of a while loop:

Example 18:

```
public class Calculation
{
    public static void main(String[] args)
    {
            // TODO Auto-generated method stub
            int i = 0;
            System.out.println("Printing the list of first 10 even numbers \n");
            while(i<=10)
            {
                System.out.println(i);
                i = i + 2;
            }
    }
}
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

Java do-while loop:

The do-while loop in Java is a control flow statement that guarantees the loop body will execute at least once, regardless of whether the condition is true or false. This is because the condition is evaluated after the loop body has executed. It is also known as an exit-controlled loop because the loop's condition is checked only after the loop statements are executed.

Example 19:

```
public class Calculation
{
    public static void main(String[] args)
    {
            // TODO Auto-generated method stub
            int i = 0;
            System.out.println("Printing the list of first 10 even numbers \n");
            do
            {
                    System.out.println(i);
                    i = i + 2;
            }while(i<=10);
    }
}
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

The break statement in Java is utilized to exit from the current loop or switch statement and transfer control to the statement immediately following the loop or switch. When used within nested loops, the break statement only exits the innermost loop that encloses it. It cannot be used independently outside of loops or switch statements.

Example of Using the break Statement in a For Loop

Here's an example that demonstrates the use of the break statement within a for loop:

Example 20:

```
public class BreakEg
{
    public static void main(String[] args)
    {
            // TODO Auto-generated method stub
            for(int i = 0; i<= 10; i++)
            {
                    System.out.println(i);
                    if(i==6)
                    {
                            break;
}       }       }       }
```

Output:

```
0
1
2
3
4
5
6
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately. Consider the following example to understand the functioning of the continue statement in Java.

Example 21:

```
public class ContinueEg
{
    public static void main(String[] args)
    {
            // TODO Auto-generated method stub
            for(int i = 0; i<= 2; i++)
            {
                    for (int j = i; j<=5; j++)
                    {
                            if(j == 4)
                            {
                                    continue;
                            }
                    System.out.println(j);
                    }
            }
    }

}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

Return Statement

In Java programming, the return statement serves the purpose of returning a value from a method once its execution is completed. When used within a method, the return statement not only exits the method but also provides a value back to the caller. Additionally, if a return statement is used within a loop, it will break out of the loop and terminate the method, skipping any remaining code in that method.

Here are some key points to remember about the return statement:

- Return Type Consistency: The return type declared in the method signature must match the type of value returned. For instance, if a method is defined with a float return type, the value returned should also be of type float.
- Void Methods: Methods declared with a void return type do not require a return statement to return a value. If a return statement is used in a void method, it should not include a value, or it will result in a compilation error.
- Variable Type Matching: The variable used to store the returned value from a method call must be of the same type as the method's return type to avoid type mismatch errors.
- Parameter Sequence: When calling a method that requires parameters, the sequence and types of the arguments must match the order and types specified in the method's declaration.

Example of the Return Statement

Here's an example to illustrate the use of the return statement in a Java method:

Example 22:

```
public class ReturnEg
{
   /* Method with an integer return type and no arguments */
   public int CompareNum()
   {
      int x = 3;
      int y = 8;
      System.out.println("x = " + x + "\ny = " + y);
      if(x>y)
         return x;
      else
         return y;
   }
   /* Driver Code */
   public static void main(String ar[])
   {
      SampleReturn1 obj = new SampleReturn1();
      int result = obj.CompareNum();
      System.out.println("The greater number among x and y is: " + result);
   }
}
```

Output:

```
x = 3
y = 8
The greater number among x and y is: 8
```

Lecture 4

Array:

Normally, an array is a collection of similar type of elements which has contiguous memory location. Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on. Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the size of operator. In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Clone able interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java. Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

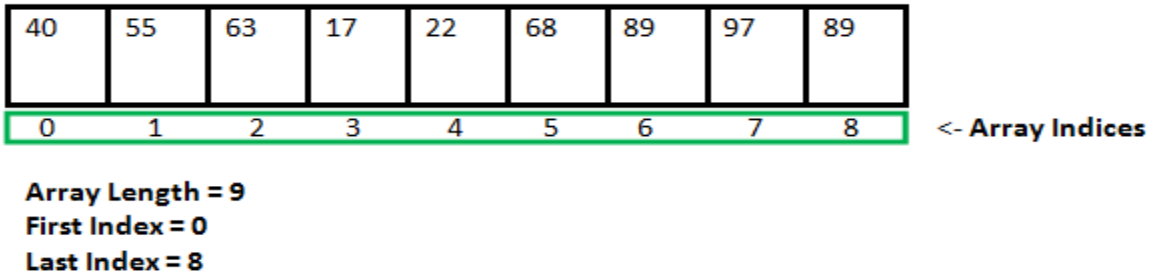
Advantages:

- Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
- Random access: We can get any data located at an index position.

Disadvantages:

- Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values are stored in contiguous memory locations.



The general form of a **one-dimensional array** declaration is

type var-name[]; OR type[] var-name;

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc., or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example 23:

```
class Testarray
{
    public static void main(String args[])
    {
            int a[]=new int[5];//declaration and instantiation
            a[0]=10;//initialization
            a[1]=20;
            a[2]=70;
            a[3]=40;
            a[4]=50;
                    //traversing array
            for(int i=0;i<a.length;i++)//length is the property of array
                    System.out.println(a[i]);
    }
}
```

Output:

```
10
20
70
40
50
```

Instantiation of a Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example 24:

```
class Testarray3
{
    public static void main(String args[])
    {
            //declaring and initializing 2D array
            int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
            //printing 2D array
```

```
            for(int i=0;i<3;i++)
            {
                for(int j=0;j<3;j++)
                {
                    System.out.print(arr[i][j]+" ");
                }
             System.out.println();
            }
    }
}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Addition of 2 Matrices in Java

Example 25:

```
class Matrices
{
    public static void main (String args[])
    {
            int a[][]={{1,3,4},{3,4,5}};
            int b[][]={{1,3,4},{3,4,5}};
            int c[][]=new int[2][3];
            for(int i=0;i<2;i++)
            {
                for(int j=0;j<3;j++)
                {
                    c[i][j]=a[i][j]+b[i][j];
                    System.out.print(c[i][j]+" ");
                }
            System.out.println();//new line
            }
    }
}
```

Output:

```
2 6 8
6 8 10
```

Lecture 5

String :

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

Example 26:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

In Java, the String class is used to represent a sequence of characters. Strings in Java are immutable, meaning once a String object is created, it cannot be altered. Any modification to a string results in the creation of a new String object. To handle mutable sequences of characters, Java provides the StringBuffer and StringBuilder classes. There are two ways to create String object:

1. By string literal
2. By new keyword

Java String literal is created by using double quotes. For Example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

Java String literal is created by using new keyword. For Example:

```
String s=new String("Welcome");//creates two objects and one reference variable
```

While String objects in Java are immutable, you can use StringBuffer or StringBuilder for mutable sequences. Strings can be created either through literals or by using the new keyword, each method having different implications for memory and performance.

Example 27:

```
public class StringEg
{
    public static void main(String args[])
    {
            String s1="java";//creating string by Java string literal
            char ch[]={'s','t','r','i','n','g','s'};
            String s2=new String(ch);//converting char array to string
```

```
        String s3=new String("example");//creating Java string by new keyword
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```

Output:

```
java
strings
example
```

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.

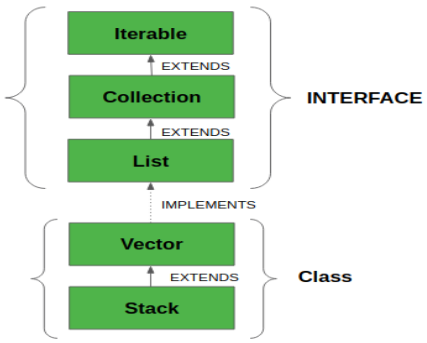
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

Vectors

The Vector class in Java provides a resizable array implementation of the List interface. Although it is part of the legacy classes introduced in earlier versions of Java, it is fully compatible with the modern Collections Framework. The Vector class is found in the java.util package and supports dynamic array resizing, meaning it can grow or shrink as needed.

Key Characteristics of Vector:

1. **Resizable Array:** Vector can automatically adjust its size as elements are added or removed.
2. **Thread Safety:** Vector is synchronized, which means it is thread-safe and can be used safely in multi-threaded environments. However, this synchronization can lead to performance overhead compared to non-synchronized collections like ArrayList.
3. **Implements List Interface:** Since Vector implements the List interface, it provides all the methods defined in List, such as add(), remove(), get(), size(), etc.



The Vector class in Java represents a dynamic array that can automatically adjust its size as needed. It shares several similarities with ArrayList, such as maintaining the order of elements and allowing access via an integer index. However, Vector has some distinct characteristics:

Key Characteristics of Vector:

1. **Dynamic Sizing:** Like an array, a Vector can grow or shrink in size as elements are added or removed. This flexibility makes it a dynamic data structure.
2. **Synchronized:** Unlike ArrayList, Vector is synchronized. This means it is thread-safe and can be used in concurrent environments. However, this synchronization introduces additional overhead, which can lead to decreased performance compared to ArrayList when used in single-threaded contexts.
3. **Legacy Methods:** Vector includes some legacy methods that are not part of the modern Collection framework. These methods include addElement(), removeElement(), and elementAt(), which are not present in ArrayList.
4. **Insertion Order:** Vector maintains the order of elements in which they are added, similar to ArrayList.
5. **Performance Considerations:** Due to its synchronization, Vector is generally less efficient than ArrayList for operations like adding, searching, deleting, and updating elements when thread safety is not a concern.
6. **Fail-Fast Iterators:** Iterators returned by the Vector class are fail-fast. If the Vector is modified while an iterator is in use (except through the iterator's own remove method), the iterator will throw a ConcurrentModificationException to indicate concurrent modification.

Syntax:

public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

Here, E is the type of element.

- It extends AbstractList and implements List interfaces.
- It implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess interfaces.
- The directly known subclass is Stack.

Important points regarding the Increment of vector capacity are as follows:

If the increment is specified, Vector will expand according to it in each allocation cycle. Still, if the increment is not specified, then the vector's capacity gets doubled in each allocation cycle. Vector defines three protected data members:

- int capacityIncreament: Contains the increment value.
- int elementCount: Number of elements currently in vector stored in it.
- Object elementData[]: Array that holds the vector is stored in it.

Common Errors in the declaration of Vectors are as follows:

- Vector throws an IllegalArgumentException if the InitialSize of the vector defined is negative.
- If the specified collection is null, It throws NullPointerException.

Constructors

1. Vector(): Creates a default vector of the initial capacity is 10.
Vector<E> v = new Vector<E>();

2. Vector(int size): Creates a vector whose initial capacity is specified by size.
Vector<E> v = new Vector<E>(int size);
3. Vector(int size, int incr): Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time a vector is resized upward.
Vector<E> v = new Vector<E>(int size, int incr);

4. Vector(Collection c): Creates a vector that contains the elements of collection c.
Vect.
Vector<E> v = new Vector<E>(Collection c);

Methods in Vector Class

METHOD	DESCRIPTION
add(E e)	Appends the specified element to the end of this Vector.
add(int index, E element)	Inserts the specified element at the specified position in this Vector.
addAll(Collection<? extends E> c)	Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
addAll(int index, Collection<? extends E> c)	Insert all of the elements in the specified Collection into this Vector at the specified position.
addElement(E obj)	Adds the specified component to the end of this vector, increasing its size by one.
capacity()	Returns the current capacity of this vector.
clear()	Removes all of the elements from this Vector.
clone()	Returns a clone of this vector.

METHOD	DESCRIPTION
contains(Object o)	Returns true if this vector contains the specified element.
containsAll(Collection<?> c)	Returns true if this Vector contains all of the elements in the specified Collection.
copyInto(Object[] anArray)	Copies the components of this vector into the specified array.
elementAt(int index)	Returns the component at the specified index.
elements()	Returns an enumeration of the components of this vector.
ensureCapacity(int minCapacity)	Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
equals(Object o)	Compares the specified Object with this Vector for equality.
firstElement()	Returns the first component (the item at index 0) of this vector.
forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
get(int index)	Returns the element at the specified position in this Vector.
hashCode()	Returns the hash code value for this Vector.
indexOf(Object o)	Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
indexOf(Object o, int index)	Returns the index of the first occurrence of the specified element in this vector, searching forwards from the index, or returns -1 if the element is not found.
insertElementAt(E obj, int index)	Inserts the specified object as a component in this vector at the specified index.
isEmpty()	Tests if this vector has no components.

METHOD	DESCRIPTION
iterator()	Returns an iterator over the elements in this list in a proper sequence.
lastElement()	Returns the last component of the vector.
lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
lastIndexOf(Object o, int index)	Returns the index of the last occurrence of the specified element in this vector, searching backward from the index, or returns -1 if the element is not found.
listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
remove(int index)	Removes the element at the specified position in this Vector.
remove(Object o)	Removes the first occurrence of the specified element in this Vector. If the Vector does not contain the element, it is unchanged.
removeAll(Collection<?> c)	Removes from this Vector all of its elements contained in the specified Collection.
removeAllElements()	Removes all components from this vector and sets its size to zero.
removeElement(Object obj)	Removes the first (lowest-indexed) occurrence of the argument from this vector.
removeElementAt(int index)	Deletes the component at the specified index.
removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
removeRange(int fromIndex, int toIndex)	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.

METHOD	DESCRIPTION
replaceAll(UnaryOperator<E> operator)	Replaces each element of this list with the result of applying the operator to that element.
retainAll(Collection<?> c)	Retains only the elements in this Vector contained in the specified Collection.
set(int index, E element)	Replaces the element at the specified position in this Vector with the specified element.
setElementAt(E obj, int index)	Sets the component at the specified index of this vector to be the specified object.
setSize(int newSize)	Sets the size of this vector.
size()	Returns the number of components in this vector.
sort(Comparator<? super E> c)	Sorts this list according to the order induced by the specified Comparator.
spliterator()	Creates a late-binding and fail-fast Spliterator over the elements in this list.
subList(int fromIndex, int toIndex)	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
toArray()	Returns an array containing all of the elements in this Vector in the correct order.
toArray(T[] a)	Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
toString()	Returns a string representation of this Vector, containing the String representation of each element.
trimToSize()	Trims the capacity of this vector to be the vector's current size.

Example 28:

```
import java.io.*;
import java.util.*;

class TCET {

   // Main driver method
   public static void main(String[] args)
   {
      // Size of the Vector
      int n = 5;

      // Declaring the Vector with
      // initial size n
      Vector<Integer> v = new Vector<Integer>(n);

      // Appending new elements at
      // the end of the vector
      for (int i = 1; i <= n; i++)
         v.add(i);

      // Printing elements
      System.out.println(v);

      // Remove element at index 3
      v.remove(3);

      // Displaying the vector
      // after deletion
      System.out.println(v);

      // iterating over vector elements
      // using for loop
      for (int i = 0; i < v.size(); i++)

         // Printing elements one by one
         System.out.print(v.get(i) + " ");
   }
}
```

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Note:

- *If the vector increment is not specified then it's capacity will be doubled in every increment cycle.*
- *The capacity of a vector cannot be below the size, it may equal to it.*

Performing Various Operations on Vector class in Java

Let us discuss various operations on Vector class that are listed as follows:

1. Adding elements
2. Updating elements
3. Removing elements
4. Iterating over elements

Operation 1: Adding Elements

In order to add the elements to the Vector, we use the add() method. This method is overloaded to perform multiple operations based on different parameters. They are listed below as follows:

- add(Object): This method is used to add an element at the end of the Vector.
- add(int index, Object): This method is used to add an element at a specific index in the Vector.

Example 29:

```
// Java Program to Add Elements in Vector Class

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
// AddElementsToVector
class TCET {

    // Main driver method
    public static void main(String[] arg)
    {

        // Case 1
        // Creating a default vector
        Vector v1 = new Vector();

        // Adding custom elements
        // using add() method
        v1.add(1);
```

```
        v1.add(2);
        v1.add("TCET");
        v1.add("MUMBAI");
        v1.add(3);

        // Printing the vector elements to the console
        System.out.println("Vector v1 is " + v1);

        // Case 2
        // Creating generic vector
        Vector<Integer> v2 = new Vector<Integer>();

        // Adding custom elements
        // using add() method
        v2.add(1);
        v2.add(2);
        v2.add(3);

        // Printing the vector elements to the console
        System.out.println("Vector v2 is " + v2);
    }
}
```

Output:

```
Vector v1 is [1, 2, TCET, MUMBAI, 3]
Vector v2 is [1, 2, 3]
```

Operation 2: Updating Elements

After adding the elements, if we wish to change the element, it can be done using the set() method. Since a Vector is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element to be inserted at that index.

Example 30:

```
// Java code to change the
// elements in vector class

import java.util.*;

public class UpdatingVector {

    public static void main(String args[])
    {
```

```
        // Creating an empty Vector
        Vector<Integer> vec_tor = new Vector<Integer>();

        // Use add() method to add elements in the vector
        vec_tor.add(12);
        vec_tor.add(23);
        vec_tor.add(22);
        vec_tor.add(10);
        vec_tor.add(20);

        // Displaying the Vector
        System.out.println("Vector: " + vec_tor);

        // Using set() method to replace 12 with 21
        System.out.println("The Object that is replaced is: "
                                    + vec_tor.set(0, 21));

        // Using set() method to replace 20 with 50
        System.out.println("The Object that is replaced is: "
                                    + vec_tor.set(4, 50));

        // Displaying the modified vector
        System.out.println("The new Vector is:" + vec_tor);
    }
}
```

Output:

```
Vector: [12, 23, 22, 10, 20]
The Object that is replaced is: 12
The Object that is replaced is: 20
The new Vector is:[21, 23, 22, 10, 50]
```

Operation 3: Removing Elements

In order to remove an element from a Vector, we can use the remove() method. This method is overloaded to perform multiple operations based on different parameters. They are:

- remove(Object): This method is used to remove an object from the Vector. If there are multiple such objects, then the first occurrence of the object is removed.
- remove(int index): Since a Vector is indexed, this method takes an integer value which simply removes the element present at that specific index in the Vector. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.

Example 31:

```
// Java code illustrating the removal
```

```
// of elements from vector

import java.util.*;
import java.io.*;

class RemovingElementsFromVector {

	public static void main(String[] arg)
	{

		// create default vector of capacity 10
		Vector v = new Vector();

		// Add elements using add() method
		v.add(1);
		v.add(2);
		v.add("TCET");
		v.add("MUMBAI");
		v.add(4);

		// removing first occurrence element at 1
		v.remove(1);

		// checking vector
		System.out.println("after removal: " + v);
	}
}
```

Output:

after removal: [1, TCET, MUMBAI, 4]

Operation 4: Iterating the Vector

There are multiple ways to iterate through the Vector. The most famous ways are by using the basic for loop in combination with a get() method to get the element at a specific index and the advanced for a loop.

Example 32:

```
// Java program to iterate the elements
// in a Vector

import java.util.*;

public class IteratingVector {

	public static void main(String args[])
	{
```

```
                // create an instance of vector
                Vector<String> v = new Vector<>();

                // Add elements using add() method
                v.add("TCET");
                v.add("ENGINEERS");
                v.add(1, "For");

                // Using the Get method and the
                // for loop
                for (int i = 0; i < v.size(); i++) {

                        System.out.print(v.get(i) + " ");
                }

                System.out.println();

                // Using the for each loop
                for (String str : v)
                        System.out.print(str + " ");
        }
}
```

Output:

```
TCET For ENGINEERS
TCET For ENGINEERS
```

Here is a simple example that demonstrates how to use a Vector in Java:

Example 33:

```
import java.util.Vector;

public class VectorExample {
        public static void main(String[] args) {
                // Create a new vector
                Vector<Integer> v = new Vector<>(3, 2);

                // Add elements to the vector
                v.addElement(1);
                v.addElement(2);
                v.addElement(3);

                // Insert an element at index 1
                v.insertElementAt(0, 1);

                // Remove the element at index 2
```

```
        v.removeElementAt(2);

        // Print the elements of the vector
        for (int i : v) {
            System.out.println(i);
        }
    }
}
```

Output:

```
1
0
3
```

Advantages of using Vector in Java:

- Synchronization: As mentioned before, Vector is synchronized, making it safe to use in a multi-threaded environment.
- Dynamic Size: The size of a Vector can grow or shrink dynamically as elements are added or removed, so you don't have to worry about setting an initial size that will accommodate all elements.
- Legacy support: Vector has been part of Java since its inception and is still supported, so it's a good option if you need to work with older Java code that uses Vector.

Disadvantages of using Vector in Java:

- Performance: The synchronization in Vector can lead to slower performance compared to other collection classes, such as ArrayList.
- Legacy Code: While Vector is still supported, newer Java code is often written using the more modern collection classes, so it may be harder to find examples and support for Vector.
- Unnecessary overhead: If you don't need the synchronization features of Vector, using it will add unnecessary overhead to your code.

Lecture 6

Classes :

Java Classes/Objects

Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake. A Class is like an object constructor, or a "blueprint" for creating objects.

Properties of Java Classes

Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created. Class does not occupy memory. Class is a group of variables of different data types and a group of methods. A Class in Java can contain:

- Data member
- Method
- Constructor
- Nested Class
- Interface

Components of Java Classes

In general, class declarations can include these components, in order:

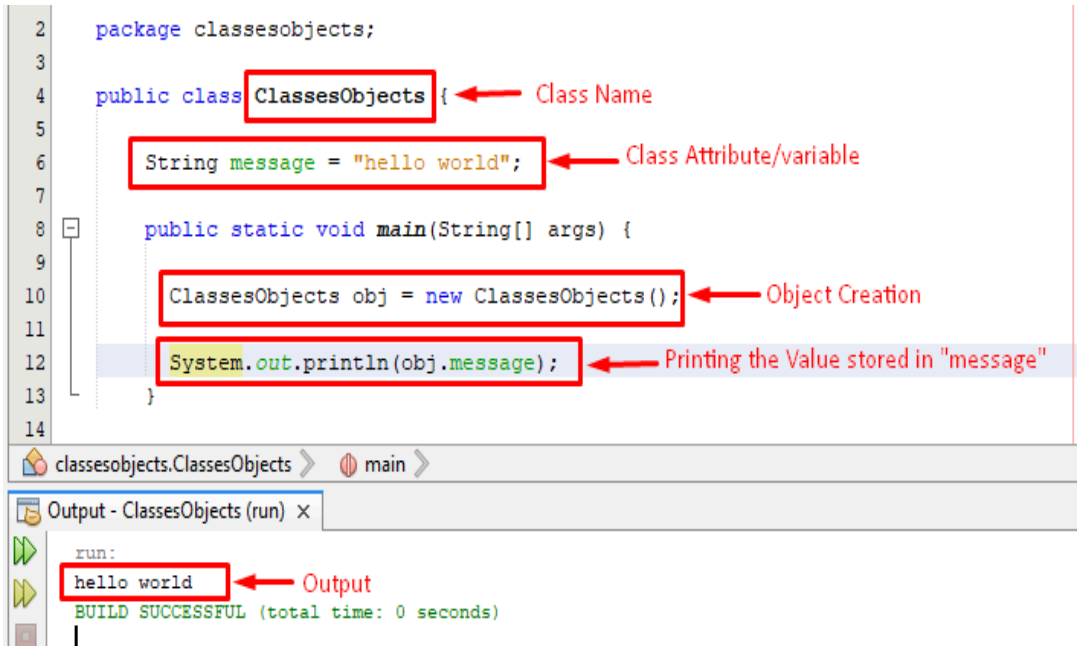
- Modifiers: A class can be public or has default access (Refer this for details).
- Class keyword: class keyword is used to create a class.
- Class name: The name should begin with an initial letter (capitalized by convention).
- Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- Body: The class body is surrounded by braces, { }.

Java Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.

2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.



Example 34:

```
// Class Declaration

public class Dog {
        // Instance Variables
        String name;
        String breed;
        int age;
        String color;

        // Constructor Declaration of Class
        public Dog(String name, String breed, int age,
                        String color)
        {
                this.name = name;
                this.breed = breed;
                this.age = age;
                this.color = color;
        }

        // method 1
        public String getName() { return name; }
```

```
        // method 2
        public String getBreed() { return breed; }

        // method 3
        public int getAge() { return age; }

        // method 4
        public String getColor() { return color; }

        @Override public String toString()
        {
                return ("Hi my name is " + this.getName()
                                + ".\nMy breed,age and color are "
                                + this.getBreed() + "," + this.getAge()
                                + "," + this.getColor());
        }

        public static void main(String[] args)
        {
                Dog tuffy
                        = new Dog("tuffy", "papillon", 5, "white");
                System.out.println(tuffy.toString());
        }
}
```

Output:

```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

Java Methods

A method is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as functions. Why use methods? To reuse code: define the code once, and use it many times.

Here is a simple example that demonstrates how to use a Method in Java:

Initialize by using method/function:

Example 35:

```
public class TCET {
        // sw=software
```

```
    static String sw_name;
    static float sw_price;

    static void set(String n, float p)
    {
        sw_name = n;
        sw_price = p;
    }

    static void get()
    {
        System.out.println("Software name is: " + sw_name);
        System.out.println("Software price is: "
                                    + sw_price);
    }

    public static void main(String args[])
    {
        TCET.set("Visual studio", 0.0f);
        TCET.get();
    }
}
```

Output:

```
Software name is: Visual studio
Software price is: 0.0
```

Difference Between Class And Object:

Class	Object
Class is used as a template for declaring and creating the objects.	An object is an instance of a class.
When a class is created, no memory is allocated.	Objects are allocated memory space whenever they are created.
The class has to be declared first and only once.	An object is created many times as per requirement.
A class can not be manipulated as they are not available in the memory.	Objects can be manipulated.

Class	Object
A class is a logical entity.	An object is a physical entity.
It is declared with the class keyword	It is created with a class name in C++ and with the new keywords in Java.
Class does not contain any values which can be associated with the field.	Each object has its own values, which are associated with it.
A class is used to bind data as well as methods together as a single unit.	Objects are like a variable of the class.

Lecture 7

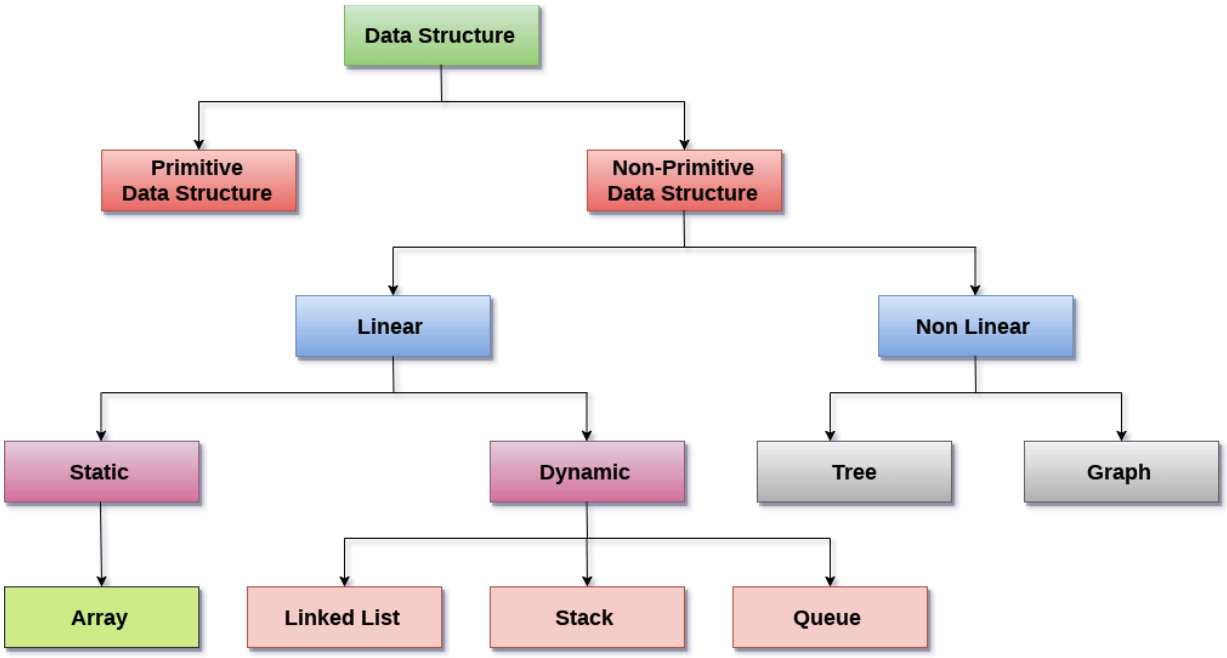
Introduction to Data Structures:

A data structure refers to a collection of data elements that are organized in a manner allowing for efficient storage and access to data on a computer. Examples of data structures include arrays, linked lists, stacks, queues, and more. These structures are fundamental to various fields in computer science, such as operating systems, compiler design, artificial intelligence, computer graphics, and others.

Data structures are essential components of many algorithms, providing programmers with the tools to manage data efficiently. Their proper use can significantly improve the performance of software or applications, as the primary goal is often to store and retrieve data quickly and efficiently.

Types of data structure:

We can classify data structure as follows:



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in a non-hierarchical way where each element has the successors and predecessors except the first and last element.
Types of Linear Data Structures are given below:
Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.
The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.
The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
A stack is an abstract data type (ADT), which can be implemented in most programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.
It is an abstract data structure, similar to stack. Queue is opened at both ends therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.
Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.
Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from a tree in the sense that a graph can have cycles while the tree cannot have one.

Learning Objective: Students will be able to understand what Data structure is.

- What are the different operations performed on data structures
- Enlist Types of data structure

Exercise	
1.	What are the operations performed on DS?
2.	Explain Data structures and its types.

Learning from the lecture: Data Structures its types and operations performed.

Lecture 8
Abstract data Type

Learning Objectives:

- Students shall know what ADT is.

Course Content

The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

The abstract datatype is special kind of datatype, whose behaviour is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

Let us see some operations of those mentioned ADT −

- Stack −
    - isFull(), This is used to check whether stack is full or not
    - isEmpry(), This is used to check whether stack is empty or not
    - push(x), This is used to push x into the stack
    - pop(), This is used to delete one element from top of the stack
    - peek(), This is used to get the top most element of the stack
    - size(), this function is used to get number of elements present into the stack
- Queue −
    - isFull(), This is used to check whether queue is full or not
    - isEmpry(), This is used to check whether queue is empty or not
    - insert(x), This is used to add x into the queue at the rear end
    - delete(), This is used to delete one element from the front end of the queue
    - size(), this function is used to get number of elements present into the queue
- List −
    - size(), this function is used to get number of elements present into the list
    - insert(x), this function is used to insert one element into the list
    - remove(x), this function is used to remove given element from the list
    - get(i), this function is used to get element at position i
    - replace(x, y), this function is used to replace x with y value

Exercise	
1.	Explain the ADT in Data structure.
2.	List the operations of ADT used in DS.

Learning from the lecture: ADT and its operations.

Lecture 9
Importance of ADT

Learning Objectives:

- Student shall understand ADT of DS and its use.

Course Content:

Earlier if a programmer wanted to read a file, the whole code was written to read the physical file device. So that is how Abstract Data Type (ADT) came into existence. The code to read a file was written and placed in a library and made available for everyone's use. This concept of ADT is being used in the modern languages nowadays. An abstract data type is also known as ADT. It means providing only necessary details by hiding internal details. In short abstract data, types have only data not how to use or implement it. Also, it can be created at the level.
An object like list, set graphs with operations like

- Insert
- Locate
- Retrieve
- Delete
- MAKENULL()
- Print()

Advantage :

- Used to create a fast algorithm in less time.
- Help to manage and organize the data.
- It makes code clean and easy to understand.
- It makes the execution program fast.
- Consume less time.

Abstract data type model is considered as a combination of abstraction and encapsulation.
abstract data type model at level first encapsulation is performed then at second level abstraction is performed on the data structure.

Lecture 10
Operations on Data Structure

Learning Objectives:

- Students shall understand different operations related to DS.

Course Content:

- **Operations on data structure:**

Traversing: Every data structure consists of multiple data elements. Traversing involves visiting each element in the data structure to perform specific tasks, such as searching or sorting. For instance, if we need to calculate the average marks of a student in 6 subjects, we must go through the entire array containing the marks, compute the total, and divide the sum by 6 to get the average.
Insertion: Insertion refers to the process of adding new elements into a data structure at any specified position. If the data structure can hold `n` elements, we can insert up to `n-1` elements into it.
Deletion: Deletion is the removal of an element from a data structure. It can happen at any random position. If an attempt is made to delete an element from an already empty data structure, an underflow condition arises.
Searching: Searching is the method of locating a specific element within a data structure. Two common algorithms used for searching are Linear Search and Binary Search. Both approaches will be covered in detail later.
Sorting: Sorting involves arranging the elements of a data structure in a particular order, such as ascending or descending. Various algorithms, such as bubble sort, selection sort, and insertion sort, are used for this purpose.
Merging: Merging refers to the combination of two lists, List A with `M` elements and List B with `N` elements, into a new list, List C, which will contain `M+N` elements. This operation combines two sets of similar data types into one larger set.

Let's check the take away from this lecture

- Which of the following is not operation of DS

1. Inserting
2. Deleting
3. Traversing
4. Merging
5. None of above

Exercise	
1.	Explain operation on DS.

Learning from the lecture is operations performed on DS.

Learning Outcomes:

1. **Know:**
   a) Student should be able to understand Data structure.
   b) Student should be able to apply concept and its operation to solve problems.

2. **Comprehend:**
   a) Student should be able to write ADT.
   b) Explain which operation is used for given problem.

3. **Apply, analyze and synthesize:**
   Student should be able to analyze application of Data Structures.

Objective Questions:

Q1) which of the following is not ADT

A. Stack
B. Queue
C. Both
D. None

Ans) None

Q2) Data Structure is?

A. Grouping of data
B. Merging of data
C. Sorting of data
D. All of above

Ans) All of above

Subjective Questions (Practice questions)

Q1) Explain operations of DS. (5marks)
Q2)List types of DS. (2marks)
Q3) Explain Data Structure. (5 marks)
Q4) Enlist applications of DS (5 marks)

Self-Evaluation:

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		**Tick Your choice**
1.	Do you understand the term Data Structure?	o Yes o No

2.	Do you understand the operations of DS?	o Yes o No
3.	Do you know the ADT?	o Yes o No
4.	Do you understand module ?	o Yes, Completely. o Partialy. o No, Not at all.

Module-02
Stacks and Queues

Motivation:

An array is a data structure with random access that allows each element to be accessed directly and in real time. A book is a good example of random access because each page can

be opened independently of the others. Many algorithms, such as binary search, require random access. A stack is an abstract data type with a predefined (bounded) capacity. It's a straightforward data structure that everyone understands.

Syllabus:

Lecture	Content	Duration	Self-Study
1	Operations on stack	1 Lecture	1 hours
2	Array implementation of stack	1 Lecture	1 hours
3	Applications of stack	1 Lecture	1 hours
4	Operations on queue	1 Lecture	1 hours
5	Array implementation of queue	1 Lecture	1 hours
6	Types of queues: circular queue	1 Lecture	1 hours
7	Priority queue	1 Lecture	1 hours
8	Double ended queue	1 Lecture	1 hours

Learning Objectives:

- Student shall know abstract data type.
- Student shall know stack.
- Student shall know abstract data type for Queue.
- Student shall understand operations on Queue.
- Student shall be able to recall implementation of array for stack.
- Student will be able to understand Stack and its Operations.

Theoretical Background:

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array.

Course Content:

Lecture 1

ADT of Stack, Operation on Stack

A stack is a data structure that follows the last-in, first-out (LIFO) principle, meaning that the most recently added item is the first one to be removed. In a stack, only two operations are typically performed: pushing, which adds an item to the top, and popping, which removes the top item. This structure allows access only to the top element, limiting how items are inserted and removed. A useful way to visualize this is by imagining a stack of books, where you can only remove or place a book at the very top.

Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO(Last in First out) structure or we can say FILO(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.
4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

A stack, sometimes referred to as a "push-down stack," is a structured collection of items where additions and removals occur exclusively at one end, known as the "top." The opposite end is called the "base." The items closer to the base have been in the stack the longest, while the most recent addition is at the top, ready to be removed next. This structure follows the LIFO (last-in, first-out) principle, where newer items are positioned near the top, and older items are situated closer to the base.

We encounter stacks in everyday life. For instance, in cafeterias, trays or plates are often stacked, and the top item is taken first, revealing the next one underneath. A similar example is a stack of books, where only the top book is immediately accessible. To reach other books, you must remove the ones above them. Stacks can also be used to store various data objects, as shown in programming, where they organize data in a way that adheres to the LIFO principle.

The Stack Abstract Data Type

The stack abstract data type (ADT) is characterized by a specific structure and set of operations. It is an ordered collection of elements where additions and removals occur only at one end, referred to as the "top." Stacks operate under the LIFO (last-in, first-out) principle. The primary operations for a stack are outlined below:

- **Stack()**: This method creates a new, empty stack. It does not take any parameters and simply returns an empty stack.
- **push(item)**: This method adds a new item to the top of the stack. It requires the item to be added, but does not return a value.
- **pop()**: This operation removes and returns the top item from the stack. It takes no parameters but modifies the stack.
- **peek()**: This method returns the top item of the stack without removing it. It requires no parameters.
- **isEmpty()**: This method checks if the stack is empty, returning a boolean value. No parameters are required.
- **size()**: This method returns the number of items currently in the stack, providing the size as an integer without requiring parameters.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations

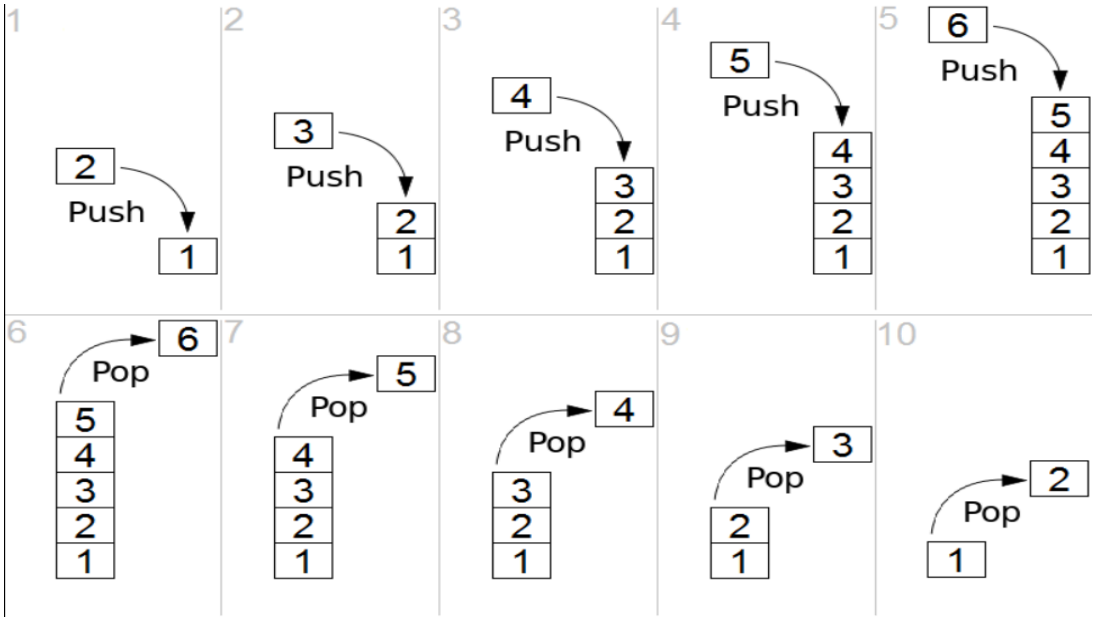
- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

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

- peek() − get the top data element of the stack, without removing it.
- isFull() − check if stack is full.
- isEmpty() − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.



Methods of the Stack Class:

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.

push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Let's check the take away from this lecture.

1. A single array A[1..MAXSIZE] is used to implement two stacks. The two stacks grow from opposite ends of the array. Variables top1 and top2 (topl< top 2) point to the location of the topmost element in each of the stacks. If the space is to be used efficiently, the condition for "stack full" is _____ .

2. The seven elements A, B, C, D, E, F and G are pushed onto a stack in reverse order, i.e., starting from G. The stack is popped five times and each element is inserted into a queue.Two elements are deleted from the queue and pushed back onto the stack. Now, one element is popped from the stack. The popped item is _____.

Exercise	
1.	What is stack? Give example.
2.	Explain PUSH and POP operation with example.

Learning from the lecture: Abstract data type stack and its basic operations.

Lecture 2

Implementation of stack

PUSH operation algorithm

A Push Operation is the act of adding a new data piece to the stack. A multitude of processes are involved in the push operation.

- Step 1 determines whether the stack is full.
- Step 2: If the stack is full, an error is generated and the programme exits.
- Step 3: If the stack isn't full, raise the top to indicate to the next empty slot.
- Step 4: Adds a data element to the top of the stack.

Program :

```
// Java Code to illustrate push() Method

import java.util.*;

// Main class
public class StackDemo {

	// Main driver method
	public static void main(String args[])
	{
		// Creating an empty Stack
		Stack& lt;
		String& gt;
		STACK = new Stack& lt;
		String& gt;
		();

		// Adding elements into the stack
		// using push() method
		STACK.push(" Welcome & quot;);
		STACK.push(" To & quot;);
		STACK.push(" Geeks & quot;);
		STACK.push(" For & quot;);
		STACK.push(" Geeks & quot;);

		// Displaying the Stack
		System.out.println(" Initial Stack: "+ STACK);

		// Pushing elements into the stack
		STACK.push(" Hello & quot;);
		STACK.push(" World & quot;);

		// Displaying the final Stack
		System.out.println(" Final Stack: "+ STACK);
	}
}
```

Algorithm for POP operation

A Pop Operation is when you access stuff while removing it from the stack. The data element is not actually destroyed in an array implementation of pop(); instead, top is decremented to

a lower place in the stack to point to the next value. Pop(), on the other hand, removes data elements and dealslocates memory space in a linked-list implementation.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.

Program :

```
// Java code to illustrate pop()
import java.util.*;

public class StackDemo {
	public static void main(String args[])
	{
		// Creating an empty Stack
		Stack<String> STACK = new Stack<String>();

		// Use add() method to add elements
		STACK.push("Welcome");
		STACK.push("To");
		STACK.push("Geeks");
		STACK.push("For");
		STACK.push("Geeks");

		// Displaying the Stack
		System.out.println("Initial Stack: " + STACK);

		// Removing elements using pop() method
		System.out.println("Popped element: " + STACK.pop());

	System.out.println("Popped element: " + STACK.pop());
		// Displaying the Stack after pop operation
		System.out.println("Stack after pop operation "+ STACK);

	}
}
```

peek():Algorithm of peek() function −

```
begin procedure peek
  return stack[top]
end procedure
```

Algorithm of isfull() function –

```
begin procedure isfull

  if top equals to MAXSIZE
    return true
  else
    return false
  endif
```

Algorithm of isempty() function –

```
begin procedure isempty

  if top less than 1
    return true
  else
    return false
  endif
 end procedure
end procedure
```

Program :

```
// Java code to illustrate peek() function

import java.util.*;

public class StackDemo {
  public static void main(String args[])
  {
            // Creating an empty Stack
            Stack<String> STACK = new Stack<String>();

            // Use push() to add elements into the Stack
            STACK.push("Welcome");
            STACK.push("To");
            STACK.push("Geeks");
            STACK.push("For");
            STACK.push("Geeks");

            // Displaying the Stack
            System.out.println("Initial Stack: " + STACK);

            // Fetching the element at the head of the Stack
            System.out.println("The element at the top of the"
                                          + " stack is: " + STACK.peek());
```

```
        // Displaying the Stack after the Operation
        System.out.println("Final Stack: " + STACK);
    }
}
```

Algorithm :

1 − Checks if the stack is full.
2 − If the stack is full, produces an error and exit.
3 − If the stack is not full, increments top to point next empty space.
4 − Adds data element to the stack location, where top is pointing.
5 − Returns success.

Program :

```
// Java code for stack implementation

import java.io.*;
import java.util.*;

class Test
{
        // Pushing element on the top of the stack
        static void stack_push(Stack<Integer> stack)
        {
                for(int i = 0; i < 5; i++)
                {
                        stack.push(i);
                }
        }

        // Popping element from the top of the stack
        static void stack_pop(Stack<Integer> stack)
        {
                System.out.println("Pop Operation:");

                for(int i = 0; i < 5; i++)
                {
                        Integer y = (Integer) stack.pop();
                        System.out.println(y);
                }
        }

        // Displaying element on the top of the stack
        static void stack_peek(Stack<Integer> stack)
        {
                Integer element = (Integer) stack.peek();
                System.out.println("Element on stack top: " + element);
        }
```

```
        // Searching element in the stack
        static void stack_search(Stack<Integer> stack, int element)
        {
                Integer pos = (Integer) stack.search(element);

                if(pos == -1)
                        System.out.println("Element not found");
                else
                        System.out.println("Element is found at position: " + pos);
        }


        public static void main (String[] args)
        {
                Stack<Integer> stack = new Stack<Integer>();

                stack_push(stack);
                stack_pop(stack);
                stack_push(stack);
                stack_peek(stack);
                stack_search(stack, 2);
                stack_search(stack, 6);
        }
}
```

Output :

```
Pop Operation:
4
3
2
1
0
Element on stack top: 4
Element is found at position: 3
Element not found
```

Let's check the take away from this lecture

1. If the sequence of operations - push (1), push (2), pop, push (1), push (2), pop, pop, pop, push (2), pop are performed on a stack, the sequence of popped out values ____.

2. Consider the following operations performed on a stack of size 5 : Push (a); Pop() ; Push(b); Push(c); Pop(); Push(d); Pop();Pop(); Push (e) Which of the following statements is correct?
    - Underflow occurs
    - Stack operations are performed smoothly
    - Overflow occurs

- None of the above

Exercise	
1.	Write a code for implementing basic operations of stack.
2.	Explain isfull and isempty methods with example.

Learning from the lecture is implementation of stack with the help of Algorithm.

Lecture 3

Stack Applications

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

- Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

- Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

Infix to Postfix:

Infix expression:The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression:The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.
Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Let's check the take away from this lecture

1. Which of the following is not an inherent application of stack?
   Implementation of recursion
   Evaluation of a postfix expression
   Job scheduling
   Reverse a string
2. Stack A has the entries a, b, c (with a on top). Stack B is empty. An entry popped out of stack A can be printed immediately or pushed to stack B. An entry popped out of the stack B can be only be printed. In this arrangement, which permutations of a, b, c are not possible?

Exercise	
1.	Convert the following infix expression into its equivalent post fix expression (A + B^ D) / (E – F) + G
2.	Write a program to reverse a sentence using stack.

Learning from the lecture is the knowledge of application where stack can be applied.

Lecture 4

ADT of Queue, Operation on Queue

A queue is an abstract data structure that operates differently from a stack. Unlike a stack, which is accessible only from one end, a queue has two open ends: one for inserting data (enqueue) and one for removing data (dequeue). This structure follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed. Real-world examples of queues include lines at ticket counters, bus stops, and single-lane roads where vehicles enter and exit in the order they arrived.

Queue Representation

In a queue, both ends are accessed for different operations. To illustrate this concept, a diagram of the queue data structure is often used.

Similar to stacks, queues can be implemented using various methods such as arrays, linked lists, pointers, and structures. For simplicity, we will focus on implementing queues using a one-dimensional array.

Fundamental Operations

Queue operations typically involve initializing or defining the queue, using it, and eventually deallocating it from memory. The fundamental operations associated with a queue are:

- **enqueue()**: Adds (or stores) an item to the queue.
- **dequeue()**: Removes (or accesses) an item from the queue.

To make these operations more efficient, additional functions are often implemented:

- **peek()**: Retrieves the element at the front of the queue without removing it.
- **isfull()**: Checks whether the queue is full.
- **isempty()**: Checks whether the queue is empty.

In a queue, data is typically dequeued (or accessed) using the front pointer, while data is enqueued (or stored) with the help of the back pointer.

Queue Class Methods

Method	Method Prototype	Description
add	boolean add(E e)	Adds element e to the queue at the end (tail) of the queue without violating the restrictions on the capacity. Returns true if success or IllegalStateException if the capacity is exhausted.
peek	E peek()	Returns the head (front) of the queue without removing it.
element	E element()	Performs the same operation as the peek () method. Throws NoSuchElementException when the queue is empty.

remove	E remove()	Removes the head of the queue and returns it. Throws NoSuchElementException if queue is empty.
poll	E poll()	Removes the head of the queue and returns it. If the queue is empty, it returns null.
Offer	boolean offer(E e)	Insert the new element e into the queue without violating capacity restrictions.
size	int size()	Returns the size or number of elements in the queue.

Let's check the take away from this lecture

1. How many stacks are needed to implement a queue? Consider the situation where no other data structure like arrays, linked list is available to you.
2. A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. The statement "Both operations can be performed in O(1) time" is CORRECT (n refers to the number of items in the queue)

Exercise	
1.	Explain concept of Queue using its operations.
2.	Discuss Dequeue and Enqueue with example.

Learning from the lecture is basic ADT Queue and its operations.

Lecture 5

Array implemantation of Queue:

peek()
This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

Algorithm

```
begin procedure peek
  return queue[front]
```

```
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
  return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull
  if rear equals to MAXSIZE
    return true
  else
    return false
  endif
end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
  if(rear == MAXSIZE - 1)
    return true;
  else
    return false;
}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty
  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif
  end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

Example

```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
```

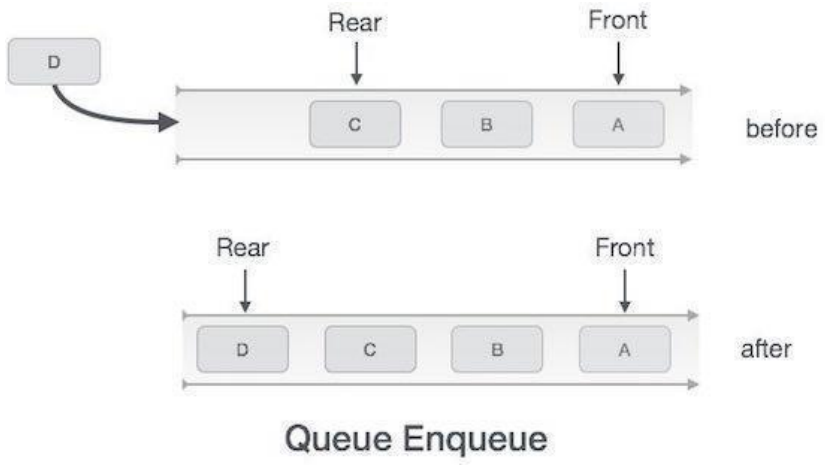
```
    return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
    if queue is full
    return overflow
  endif
     rear ← rear + 1
  queue[rear] ← data
  return true
end procedure
```

Implementation of enqueue() in C programming language –

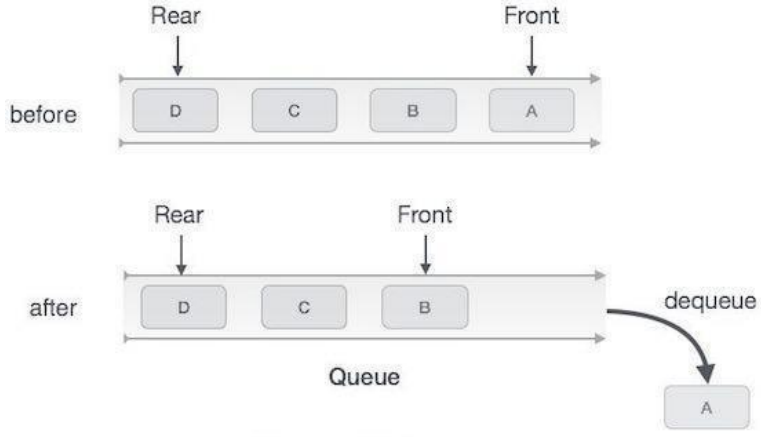
Example

```
int enqueue(int data)
  if(isfull())
    return 0;
    rear = rear + 1;
  queue[rear] = data;
  return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
    if queue is empty
    return underflow
  end if
  data = queue[front]
  front ← front + 1
  return true
```

end procedure

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {
   if(isempty())
      return 0;
   int data = queue[front];
   front = front + 1;
   return data;
}
```

Algorithm :

1 – START

2 – Check if the queue is full.

3 – If the queue is full, produce overflow error and exit.

4 – If the queue is not full, increment rear pointer to point the next empty space.

5 – Add data element to the queue location, where the rear is pointing.

6 – return success.

7 – END

Program :

```
class Queue {

  private static int front, rear, capacity;
  private static int queue[];

  Queue(int size) {
    front = rear = 0;
    capacity = size;
    queue = new int[capacity];
  }

  // insert an element into the queue
  static void queueEnqueue(int item)  {
    // check if the queue is full
    if (capacity == rear) {
      System.out.printf("\nQueue is full\n");
      return;
    }

    // insert element at the rear
    else {
      queue[rear] = item;
```

```
      rear++;
   }
   return;
}

//remove an element from the queue
static void queueDequeue()  {
   // check if queue is empty
   if (front == rear) {
      System.out.printf("\nQueue is empty\n");
      return;
   }

   // shift elements to the right by one place uptil rear
   else {
      for (int i = 0; i < rear - 1; i++) {
         queue[i] = queue[i + 1];
      }


  // set queue[rear] to 0
      if (rear < capacity)
         queue[rear] = 0;

      // decrement rear
      rear--;
   }
   return;
}

// print queue elements
static void queueDisplay()
{
   int i;
   if (front == rear) {
      System.out.printf("Queue is Empty\n");
      return;
   }

   // traverse front to rear and print elements
   for (i = front; i < rear; i++) {
      System.out.printf(" %d , ", queue[i]);
   }
   return;
}
```

```
    // print front of queue
    static void queueFront()
    {
        if (front == rear) {
            System.out.printf("Queue is Empty\n");
            return;
        }
        System.out.printf("\nFront Element of the queue: %d", queue[front]);
        return;
    }
}

public class QueueArrayImplementation {
    public static void main(String[] args) {
        // Create a queue of capacity 4
        Queue q = new Queue(4);

        System.out.println("Initial Queue:");
       // print Queue elements
        q.queueDisplay();

        // inserting elements in the queue
        q.queueEnqueue(10);
        q.queueEnqueue(30);
        q.queueEnqueue(50);
        q.queueEnqueue(70);

        // print Queue elements
        System.out.println("Queue after Enqueue Operation:");
        q.queueDisplay();

        // print front of the queue
        q.queueFront();

        // insert element in the queue
        q.queueEnqueue(90);

        // print Queue elements
        q.queueDisplay();

        q.queueDequeue();
        q.queueDequeue();
        System.out.printf("\nQueue after two dequeue operations:");

        // print Queue elements
        q.queueDisplay();
```

```
    // print front of the queue
    q.queueFront();
  }
}
```

Output :

```
Initial Queue:
Queue is Empty
Queue after Enqueue Operation:
 10 ,  30 ,  50 ,  70 ,
Front Element of the queue: 10
Queue is full
 10 ,  30 ,  50 ,  70 ,
Queue after two dequeue operations: 50 ,  70 ,
Front Element of the queue: 50
```

Let's check the take away from this lecture

1. Let Q denote a queue containing sixteen numbers and S be an empty stack. Head(Q) returns the element at the head of the queue Q without removing it from Q. Similarly Top(S) returns the element at the top of S without removing it from S. Consider the algorithm given below.

```
while Q is not Empty do
    if S is Empty OR Top(S) ≤ Head(Q) then
        x := Dequeue(Q);
        Push(S,x);
    else
        x := Pop(S);
        Enqueue(Q,x);
    end
end
```

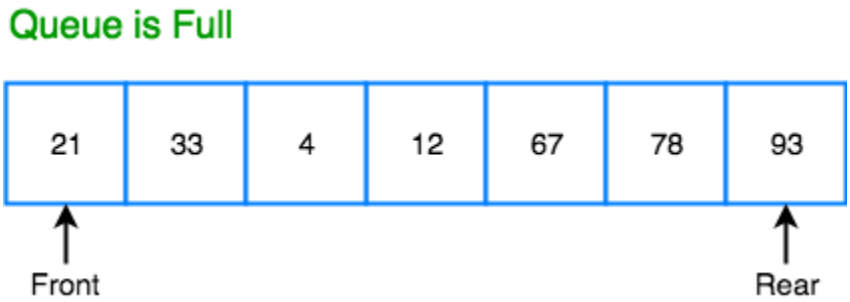
The maximum possible number of iterations of the while loop in the algorithm is_____ .

	Exercise
1.	Write a code for implementing Queue
2.	Give difference between queue and stack.

Learning from the lecture is implemenation of Queue.

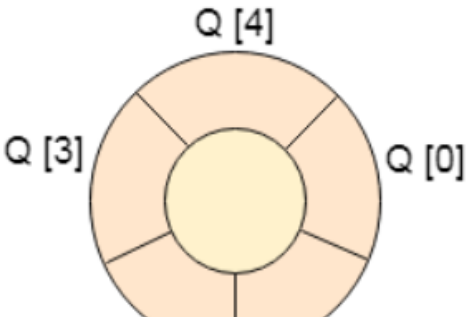
Lecture 6

Circular Queue



When we dequeue an element from the queue, we effectively move the front of the queue forward, which reduces the visible size of the queue. However, since the rear pointer remains at the end of the queue, it may seem like we cannot insert new elements even though space has been freed up at the front. This can be addressed by implementing a circular queue or by resizing the queue to accommodate new elements as needed.

To manage a queue efficiently and avoid issues with space utilization, a circular queue can be employed. This data structure operates under the FIFO (First-In-First-Out) principle but differs from a linear queue by reusing the space at the beginning of the queue after reaching the end, effectively creating a circular structure.



Here's how a circular queue functions:

- **Pointers**: Two pointers, FRONT and REAR, are used to track the positions of the first and last elements in the queue.
- **Initialization**: When initializing the queue, both FRONT and REAR are set to -1.
- **Enqueuing**: To add an element, the REAR pointer is incremented circularly (i.e., it wraps around to the start of the queue if it reaches the end). The new element is placed at the position indicated by REAR.
- **Dequeuing**: To remove an element, the value at the FRONT pointer is returned, and the FRONT pointer is incremented circularly.
- **Full Queue Check**: Before adding a new element, the queue's fullness is checked.
- **Empty Queue Check**: Before removing an element, the queue's emptiness is verified.
- **First Element Enqueue**: When the first element is enqueued, FRONT is set to 0.
- **Last Element Dequeue**: When the last element is dequeued, both FRONT and REAR are reset to -1.

This circular approach ensures efficient utilization of space by reusing positions in the queue, thereby overcoming the limitations of a linear queue.

Insertion in Circular queue

There are three scenario of inserting an element in a queue.

1. **If (rear + 1)%maxsize = front**, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
2. **If rear != max - 1**, then rear will be incremented to the **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
3. **If front != 0 and rear = max - 1**, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

    - *Algorithm to insert an element in circular queue*

- **Step 1:** IF (REAR+1)%MAX = FRONT
  Write " OVERFLOW "
  Goto step 4
  [End OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE IF REAR = MAX - 1 and FRONT ! = 0
  SET REAR = 0
  ELSE
  SET REAR = (REAR + 1) % MAX
  [END OF IF]
- **Step 3:** SET QUEUE[REAR] = VAL
- **Step 4:** EXIT

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.

2. CPU scheduling and Memory management.

Let's check the take away from this lecture

1. Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are:
   i. isEmpty (Q) — returns true if the queue is empty, false otherwise.
   ii. delete (Q) — deletes the element at the front of the queue and returns its value.
   iii. insert (Q, i) — inserts the integer i at the rear of the queue.
   Consider the following function:

```
voidf (queue Q) {
inti ;
if(!isEmpty(Q)) {
  i = delete(Q);
  f(Q);
  insert(Q, i);
 }
}
```

Which operation is performed here?

Exercise	
1.	Explain why circular queue was needed.
2.	How the conditions of isfull anf isempty differs in Queue and circular queue?

Learning from the lecture is working of circular queue and its application

Lecture 7

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.

insertRear(): Adds an item at the rear of Deque.

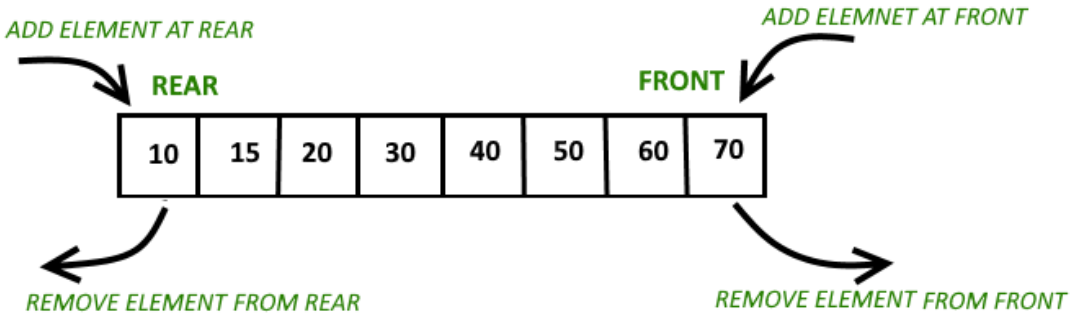
deleteFront(): Deletes an item from front of Deque.

deleteRear(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

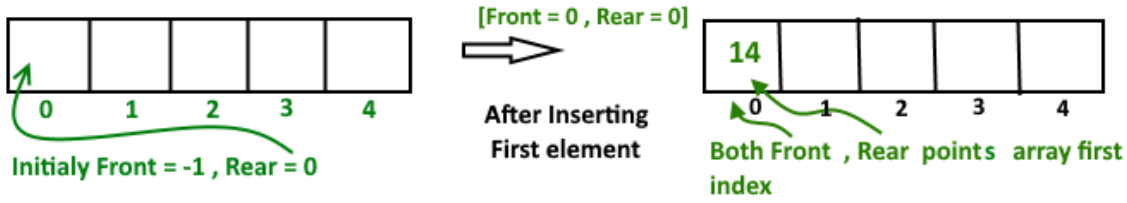
getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.



Circular array implementation deque
For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.

Working
1. Create an empty array 'arr' of size 'n'
initialize **front = -1** , **rear = 0**
Inserting First element in deque, at either front or rear will lead to the same result.

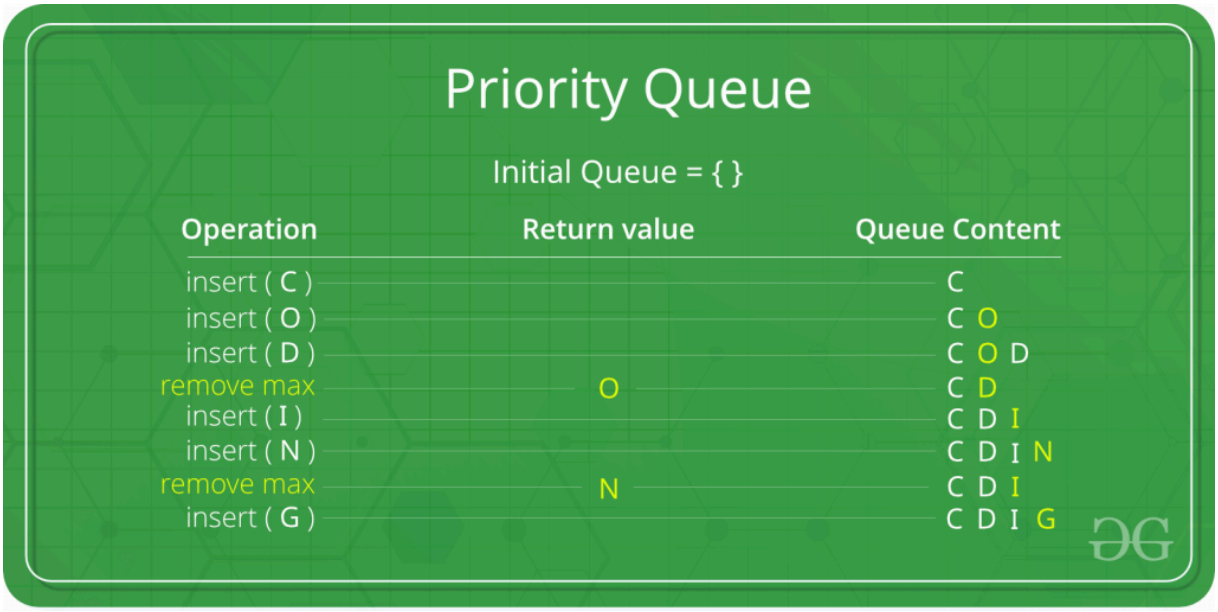


A priority queue is an advanced variant of a standard queue with additional features:

- **Priority Assignment**: Each item in the queue has an associated priority.
- **Priority-Based Removal**: Elements are dequeued based on their priority, with higher-priority items being removed before lower-priority ones.
- **Order Preservation**: If two elements share the same priority, they are dequeued in the order they were enqueued.

This structure allows for more complex management of elements compared to a basic queue, ensuring that the most critical items are processed first.

In the below priority queue, element with maximum ASCII value will have the highest priority.



A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.
How to implement priority queue?
Using Array: A simple implementation is to use array of following structure.

```
struct item {
  int item;
  int priority;
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.
deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.
We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items.

Let's check the take away from this lecture

1. What are the applications of dequeue?
2. How to delete element in double ended queue?

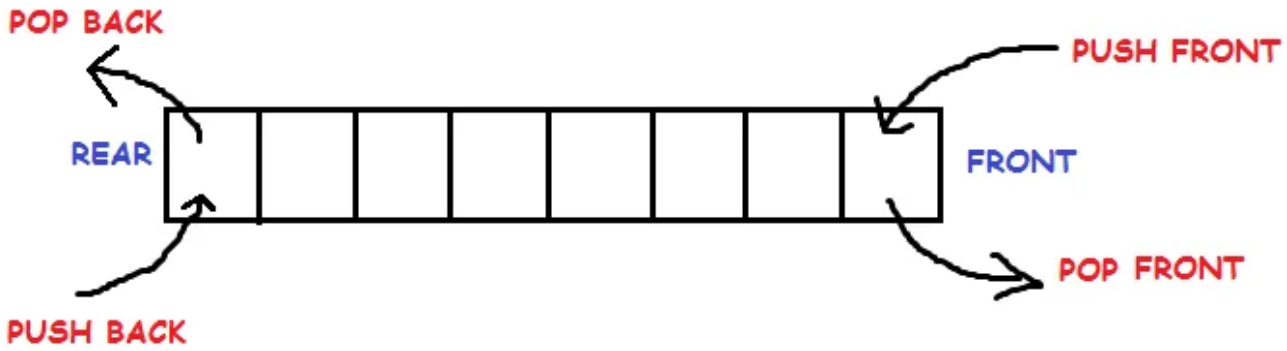
Exercise	
1.	Write a code for double ended queue.
2.	Give the applications of Priority queue.

Learning from the lecture is priority queue and double ended queue an its implementation

Lecture 8

Double Ended Queue:

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e , front and back.



Here we will implement a double ended queue using a circular array. It will have the following methods:

- push_back : inserts element at back
- push_front : inserts element at front
- pop_back : removes last element
- pop_front : removes first element
- get_back : returns last element
- get_front : returns first element
- empty : returns true if queue is empty
- full : returns true if queue is full

Algorithm for rear end :

Step -1: [Check for overflow]

if(rear==MAX)

Print("Queue is Overflow");

return;

Step-2: [Insert element]

else

rear=rear+1;

q[rear]=no;

[Set rear and front pointer]

if rear=0

rear=1;

if front=0

front=1;

Step-3: return

Algorithm for front end :

Step-1 : [Check for the front position]

if(front<=1)

Print (“Cannot add item at front end”);

return;

Step-2 : [Insert at front]

else

front=front-1;

q[front]=no;

Step-3 : Return

Content beyond Syllabus

Parsing Expressions

Parsing infix notations directly can be inefficient. Instead, infix expressions are often converted to postfix or prefix notations before evaluation. This conversion simplifies the computation process.

When parsing arithmetic expressions, it's essential to consider both operator precedence and associativity:

- **Precedence**: This determines which operator should be applied first when an operand is situated between two operators. Operators with higher precedence are executed before those with lower precedence.

By managing these aspects, you can effectively evaluate arithmetic expressions and ensure correct results.

For example –

In arithmetic expressions, operator precedence and associativity dictate the order of operations:

- **Precedence**: This determines the priority of operators. For instance, multiplication has higher precedence than addition, so in the expression `b * c`, the multiplication operation is performed before any addition.
- **Associativity**: This specifies how operators with the same precedence are handled. For example, in the expression `a + b - c`, where both `+` and `-` have the same precedence, associativity determines the evaluation order. Since both `+` and `-` are left-associative, the expression is evaluated as `(a + b) - c`.

Understanding precedence and associativity is crucial for correctly evaluating arithmetic expressions and ensuring the intended result.

Following is an operator precedence and associativity table (highest to lowest) –

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example –

In a + b*c, the expression part b*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)*c.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation –

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

Learning Outcomes:

1. **Know:**
   a) Student should be able to understand control statements.
   b) Student should be able to apply two way and multiway decision making .

2. **Comprehend:**
   a) Student should be able to write algorithm with control structures.
   b) Explain which loop works better for given problem.

3. **Apply, analyze and synthesize:**
   Student should be able to analyze the loop and its usage.

Objective Questions:

Q1) the term push and pop is related to what ?
   A. Stack
   B. Queue
   C. Both
   D. None

Ans) Stack

Q2) Stack can be implemented using?
   A. Array and binary tree
   B. Linked list and graph
   C. Array and Linked list
   D. Queue and Linked list

Ans) Array and Linked list

Q3) Insertion and Deletion operation in Queue is known as?
   A. Push and pop
   B. Enqueue and Dequeue
   C. Insert and delete
   D. None

   Ans) Enqueue and Dequeue

Q4) Stack data structure cannot be used for
   A. Implementation of recursive function
   B. Allocation resources and scheduling
   C. Reversing string
   D. Evaluation of string in postfix form

   Ans) Allocation resources and scheduling

Q5) When function calls another function then the details of previous function are stored in Stack?
   A. True
   B. false

   Ans) True

Q6) In which data structure element is inserted at one end called Rear and deleted at other end called Front.
   A. Stack

B. Queue
C. Binary Tree
D. All of above
Ans) Queue

Subjective Questions (Practice questions)

Q1) Explain operations of stack. (5marks)
Q2)List types of Queue. (2marks)
Q3) Explain priority queue with example. (5 marks)
Q4) Enlist applications of stack (5 marks)
Q5) With example give all the operations of queue (10 marks)
Q6) Describe double ended queue
Q7) Give details about circular queue with example.

References:

Sr. No.	Title	Authors	Publisher	Edition	Year
1	Data Structures using C	ReemaThareja	Oxford	Second Edition	2014
2	Data Structures: A Pseudocode Approach with C	Richard F. Gilberg&Behrouz A., Forouzan	CENGAGE Learning	Second Edition	2011
3	Data Structures Using C	Aaron M Tenenbaum, YedidyahLangsam, Moshe J Augenstein	Pearson	Second Edition	2006
4	Data Structures with C	SeymoreLipschutz	Tata McGraw-Hill	India Special Edition	2011

https://www.geeksforgeeks.org/introduction-to-arrays/
https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html
https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Self-Evaluation:

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		**Tick Your choice**
1.	Do you understand the term Stack?	o Yes o No

2.	Do you understand the Applications of stack?	o Yes o No
3.	Do you know the different types of queues?	o Yes o No
4.	Are you able to implement Stack and Queue with its operation?	o Yes o No
5.	Do you understand module ?	o Yes, Completely. o Partialy. o No, Not at all.

Module-3
Linked List

Motivation: The aim of this module is to equip students with a foundational understanding of linked lists, a fundamental data structure. It will cover the concept of linked lists as an Abstract Data Type (ADT) and explore their various applications.

Syllabus:

Lecture	Content	Duration	Self-Study

1	ADT of Linked lists: Operations on linked list	1 Lecture	1 hours
2	Types of linked lists: Single linked list	1 Lecture	1 hours
3	Double Linked list	1 Lecture	1 hours
4	Ciruclar Linked List	1 Lecture	1 hours
5	Implementation of linked list	1 Lecture	1 hours
6	Stack implementation using linked list	1 Lecture	1 hours
7	Queue implementation using linked list	1 Lecture	1 hours
8	Applications of linked list	1 Lecture	1 hours

Learning Objective:

- Learner should know about basic of Linked List and ADT
- Learner should know the concepts and types of List.
- Learner shall be able to describe applications of List.

Theoretical Background:

In Compute Science Linked List is widely used abstract data type for indexing and hashing purpose. Linked List is a ADT with Data field and the pointer to next data field in the list. It can be used where large data needs to be stored in memory blocks with keeping addressing of data as a main concern.

Key Definitions:

1. Linked list It is a data structure that consists of a sequence of nodes each of which contains a reference to the next node in the sequence.
2. Node It is a record used to create linked data structures such as trees, linked lists, and computer-based representations of graphs. Each node contains a data field and one or more links to other nodes.
3. References are the link and data fields are often implemented by pointers or references although it is also quite common for the data to be embedded directly in the node.
4. Singly list. Singly linked lists contain nodes which have a data field as well as a next field, which points to the next node in the linked list.

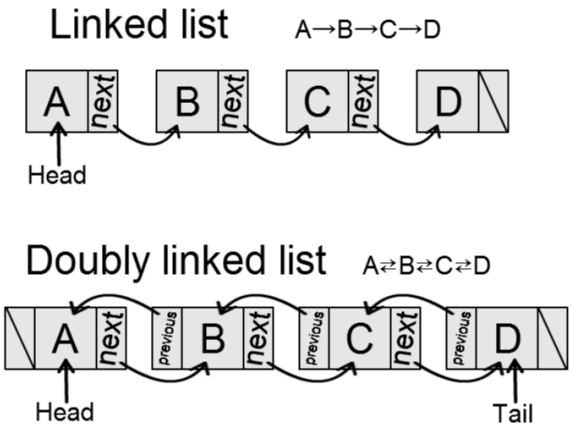
Course Content

Lecture: 1

ADT of Linked lists:

An Abstract Data Type (ADT) is a conceptual model for data structures that focuses on their usage and behavior rather than their specific implementation details. It defines a minimal interface and set of operations that the data structure must support, but does not dictate how these operations are carried out. In contrast, a data structure refers to the detailed implementation and operational procedures for managing data. Analyzing the time and space complexity is applicable to data structures but not directly to data types. Data structures can be implemented in various ways, and their implementation can vary across different programming languages.

A linked list, as an Abstract Data Type (ADT), consists of a sequence of nodes. Each node typically contains a reference to the next node, and potentially the previous one, forming a chain-like structure. A linked list does not support random access to nodes. When nodes are linked only through a reference to the next node, the list is known as a singly linked list. When nodes are linked through both next and previous references, it is referred to as a doubly linked list.



Linked List Stores Nodes from primitive types such as integers to more complex types like instances of classes.

Operations on linked list:
Basic Operations on Linked List **are as follows:**

- Traversal: To traverse all the nodes one after another.
- Insertion: To add a node at the given position.
- Deletion: To delete a node.
- Searching: To search an element(s) by value.
- Updating: To update a node.
- Sorting: To arrange nodes in a linked list in a specific order.
- Merging: To merge two linked lists into one.

- **Linked List Traversal**

  The idea here is to step through the list from beginning to end. ***For example,*** we may want to print the list or search for a specific node in the list.

The algorithm for traversing a list

- Start with the head of the list. Access the content of the head node if it is not null.
- Then go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the null node)

Algorithm

STEP 1: SET PTR = HEAD
STEP 2: IF PTR = NULL
  WRITE "EMPTY LIST"
 GOTO STEP 7
 END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
STEP 5: PRINT PTR→ DATA
STEP 6: PTR = PTR → NEXT
[END OF LOOP]
STEP 7: EXIT

Program :

```
ptr = head;
      while (ptr!=NULL)
         {
            ptr = ptr -> next;
         }
```

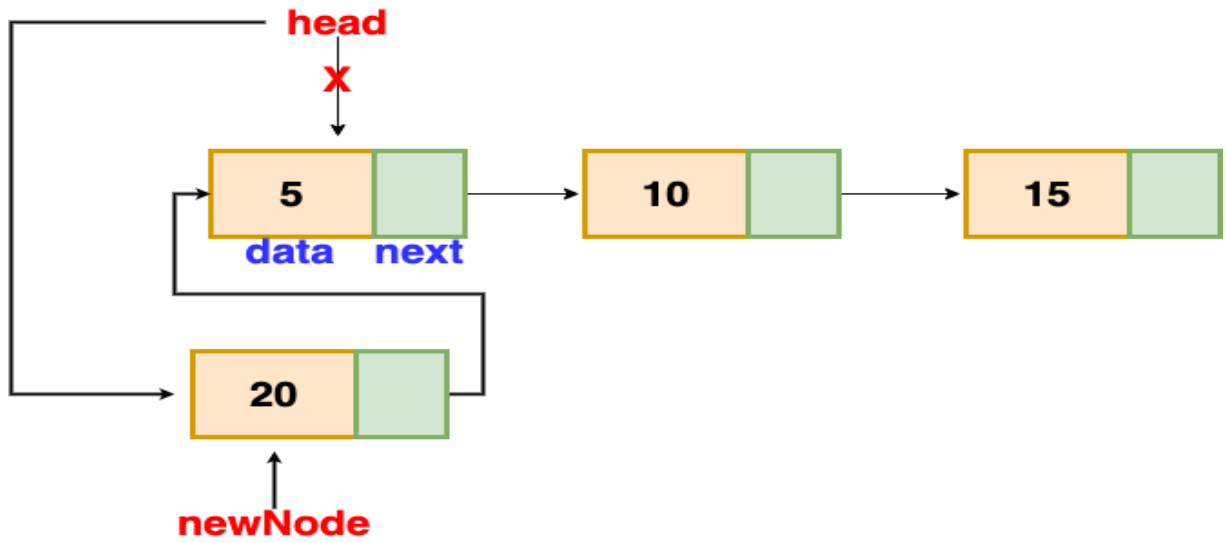
Linked List node Insertion

when we are inserting a node in a linked list following cases can occurred.

- Insertion at the beginning
- Insertion at the end. (Append)
- Insertion after a given node

Insertion at the beginning

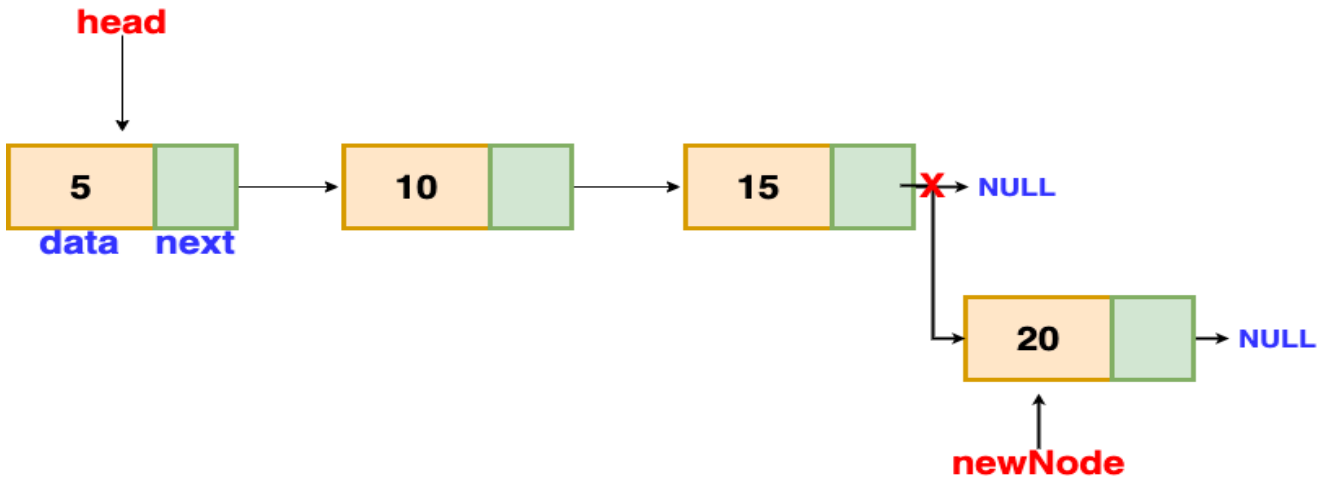
If the list is empty, the new node becomes the head of the list without needing to locate the end of the list. However, if the list is not empty, the new node is inserted at the beginning and becomes the new head, with the new node pointing to the previous head of the list.



Insertion at the beginning

Insertion at end

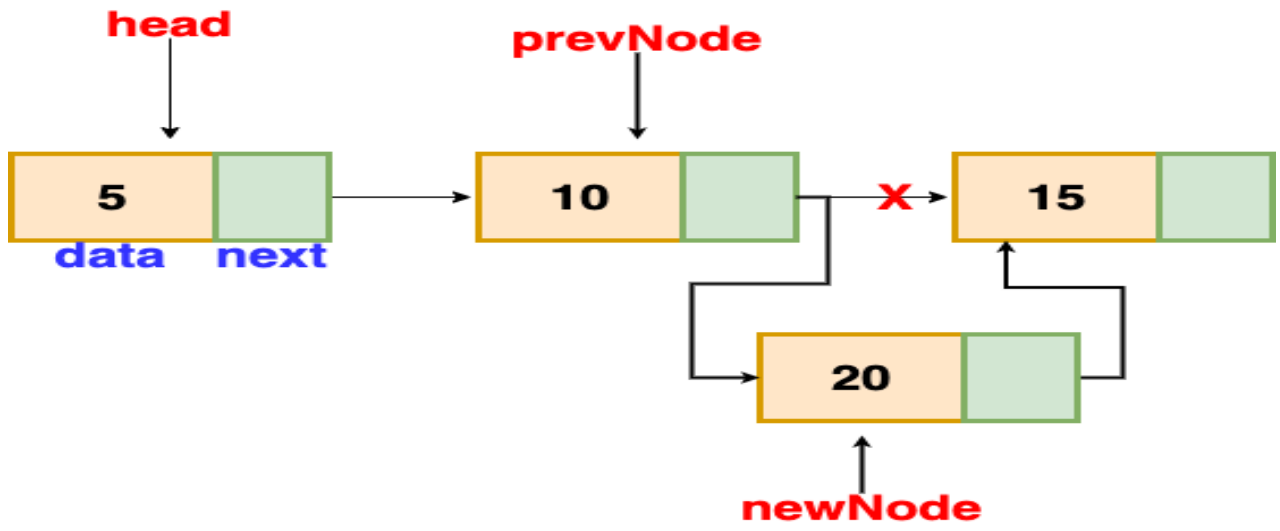
To insert a new node at the end of a linked list, we first traverse the list until we reach the last node. We then append the new node to the end of the list. Special cases must be handled, such as when the list is initially empty. In this scenario, the new node will be both the first and the last node, and thus, the head of the linked list will be updated to point to this new node.



Insertion at the end

Insertion after a given node

We are given the reference to a node, and the new node is inserted after the given node.



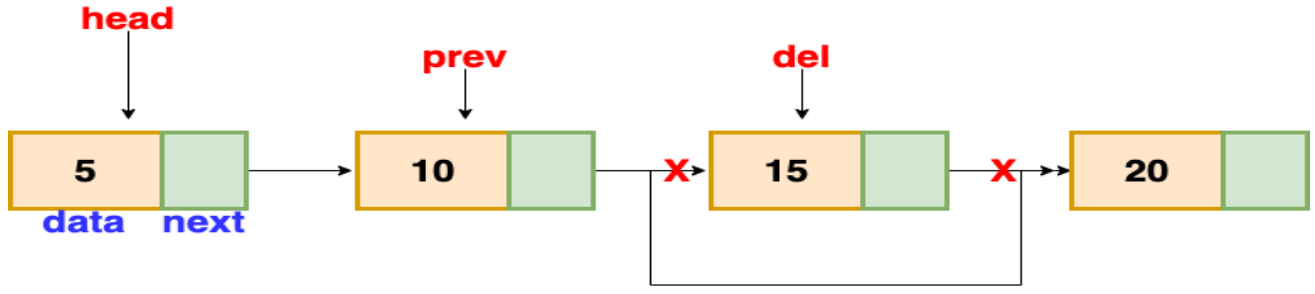
Insertion after a given node

Linked List node Deletion

To delete a node from a linked list, we need to do these steps

- Find the previous node of the node to be deleted.
- Change the next pointer of the previous node
- Free the memory of the deleted node.

In the deletion, there is a special case in which the first node is deleted. In this, we need to update the head of the linked list.



Deleting a Node in Linked List

Linked List node Searching

To search any value in the linked list, we can traverse the linked list and compares the value present in the node.

Linked List node Updation

To update the value of the node, we just need to set the data part to the new value.
Below is the implementation in which we had to update the value of the first node in which data is equal to **val** and we have to set it to **newVal**.

Let's check the takeaway from this lecture

1) Full form of ADT is
   a) Advanced Data Type
   b) Advanced Data Time
   c) Abstract Data Type
   d) Abstract Data Time
2) We can access Linked List Randomly?
   a) True
   b) False
3) A linear collection of data elements where the linear node is given by means of pointer is called?

   a) Linked list
   b) Node list
   c). Primitive list
   d) None

4) What is the functionality of the following piece of code?

```
public int function(int data)
{
Node temp = head;
int var = 0;
while(temp != null)
{
if(temp.getData() == data)
{
return var;
}
var = var+1;
temp = temp.getNext();
}
return Integer.MIN_VALUE;
}
```

   a) Find and delete a given element in the list
   b) Find and return the given element in the list
   c) Find and return the position of the given element in the list
   d) Find and insert a new element in the list

Exercise

Q.1 What is ADT?
Q.2 Explain Linked List as ADT.
Q. 3 Explain insertion operation of linked list in detail.
Q.4 With example explain deletion operation of Linked list.
Q.5 Describe node updating in linked list

Learners will be able ADT and explain the different operations on linked list.

Lecture: 2

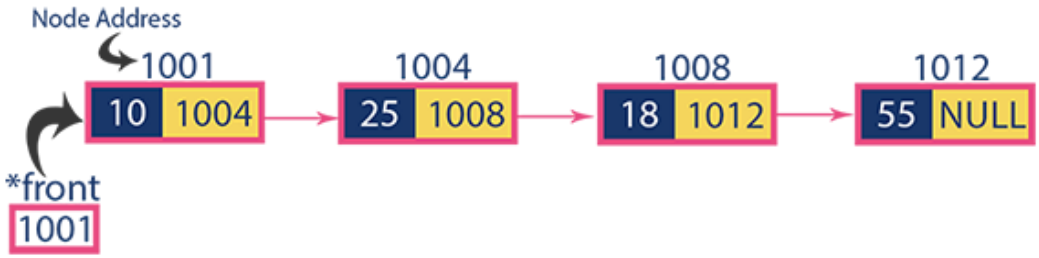
Types of linked lists: Single linked list:

A singly linked list is a collection of elements where each element, known as a "Node," contains a reference to the next element in the sequence. Each node has two components: the data field, which holds the actual value of the node, and the next field, which stores the address of the subsequent node in the list. The graphical representation of a node in a singly linked list typically shows these two fields and their connection to the next node in the sequence.

In a single linked list the address of the first node is always stored in reference node and it is know as front or head

Next part of last node must me NULL

Example



- Operations on Single Linked List

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

- Step 1 - Include all the header files which are used in the program.
- Step 2 - Declare all the user defined functions.
- Step 3 - Define a Node structure with two members data and next
- Step 4 - Define a Node pointer 'head' and set it to NULL.
- Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

- Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

- Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

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

- Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- Step 1 - Create a newNode with given value and newNode → next as NULL.
- Step 2 - Check whether list is Empty (head == NULL).
- Step 3 - If it is Empty then, set head = newNode.
- Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
- Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
- Step 6 - Set temp → next = newNode.

- Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list..

- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.
- Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
- Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
- Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

- Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

- Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Check whether list is having only one node (temp → next == NULL)
- Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)
- Step 6 - If it is FALSE then set head = temp → next, and delete temp.

- Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Check whether list has only one Node (temp1 → next == NULL)
- Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
- Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
- Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

- Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
- Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
- Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
- Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
- Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).
- Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
- Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

o Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
- Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Program :

```
import java.io.*;

// Java program to implement
// a Singly Linked List
public class LinkedList {

    Node head; // head of list

    // Linked list Node.
    // Node is a static nested class
    // so main() can access it
    static class Node {

        int data;
        Node next;

        // Constructor
        Node(int d)
        {
```

```
                data = d;
                next = null;
        }
}

// **************INSERTION**************

// Method to insert a new node
public static LinkedList insert(LinkedList list,
                                                        int data)
{
        // Create a new node with given data
        Node new_node = new Node(data);
        new_node.next = null;

        // If the Linked List is empty,
        // then make the new node as head
        if (list.head == null) {
                list.head = new_node;
        }
        else {
                // Else traverse till the last node
                // and insert the new_node there
                Node last = list.head;
                while (last.next != null) {
                        last = last.next;
                }

                // Insert the new_node at last node
                last.next = new_node;
        }

        // Return the list by head
        return list;
}

// **************TRAVERSAL**************

// Method to print the LinkedList.
public static void printList(LinkedList list)
{
        Node currNode = list.head;

        System.out.print("\nLinkedList: ");
```

```
        // Traverse through the LinkedList
        while (currNode != null) {
                // Print the data at current node
                System.out.print(currNode.data + " ");

                // Go to next node
                currNode = currNode.next;
        }
        System.out.println("\n");
}

// **************DELETION BY KEY**************

// Method to delete a node in the LinkedList by KEY
public static LinkedList deleteByKey(LinkedList list,
                                                                        int key)
{
        // Store head node
        Node currNode = list.head, prev = null;

        //
        // CASE 1:
        // If head node itself holds the key to be deleted

        if (currNode != null && currNode.data == key) {
                list.head = currNode.next; // Changed head

                // Display the message
                System.out.println(key + " found and deleted");

                // Return the updated List
                return list;
        }

        //
        // CASE 2:
        // If the key is somewhere other than at head
        //

        // Search for the key to be deleted,
        // keep track of the previous node
        // as it is needed to change currNode.next
        while (currNode != null && currNode.data != key) {
```

```
                // If currNode does not hold key
                // continue to next node
                prev = currNode;
                currNode = currNode.next;
        }

        // If the key was present, it should be at currNode
        // Therefore the currNode shall not be null
        if (currNode != null) {
                // Since the key is at currNode
                // Unlink currNode from linked list
                prev.next = currNode.next;

                // Display the message
                System.out.println(key + " found and deleted");
        }

        //
        // CASE 3: The key is not present
        //

        // If key was not present in linked list
        // currNode should be null
        if (currNode == null) {
                // Display the message
                System.out.println(key + " not found");
        }

        // return the List
        return list;
}

// **************DELETION AT A POSITION**************

// Method to delete a node in the LinkedList by POSITION
public static LinkedList
deleteAtPosition(LinkedList list, int index)
{
        // Store head node
        Node currNode = list.head, prev = null;

        //
        // CASE 1:
        // If index is 0, then head node itself is to be
```

```
// deleted

if (index == 0 && currNode != null) {
        list.head = currNode.next; // Changed head

        // Display the message
        System.out.println(
                index + " position element deleted");

        // Return the updated List
        return list;
}

//
// CASE 2:
// If the index is greater than 0 but less than the
// size of LinkedList
//
// The counter
int counter = 0;

// Count for the index to be deleted,
// keep track of the previous node
// as it is needed to change currNode.next
while (currNode != null) {

        if (counter == index) {
                // Since the currNode is the required
                // position Unlink currNode from linked list
                prev.next = currNode.next;

                // Display the message
                System.out.println(
                        index + " position element deleted");
                break;
        }
        else {
                // If current position is not the index
                // continue to next node
                prev = currNode;
                currNode = currNode.next;
                counter++;
        }
}
```

```
		// If the position element was found, it should be
		// at currNode Therefore the currNode shall not be
		// null
		//
		// CASE 3: The index is greater than the size of the
		// LinkedList
		//
		// In this case, the currNode should be null
		if (currNode == null) {
			// Display the message
			System.out.println(
				index + " position element not found");
		}

		// return the List
		return list;
}

// **************MAIN METHOD**************

// method to create a Singly linked list with n nodes
public static void main(String[] args)
{
		/* Start with the empty list. */
		LinkedList list = new LinkedList();

		// ******INSERTION******
		// Insert the values
		list = insert(list, 1);
		list = insert(list, 2);
		list = insert(list, 3);
		list = insert(list, 4);
		list = insert(list, 5);
		list = insert(list, 6);
		list = insert(list, 7);
		list = insert(list, 8);

		// Print the LinkedList
		printList(list);

		//
		// ******DELETION BY KEY******
		//
```

```
// Delete node with value 1
// In this case the key is ***at head***
deleteByKey(list, 1);

// Print the LinkedList
printList(list);

// Delete node with value 4
// In this case the key is present ***in the
// middle***
deleteByKey(list, 4);

// Print the LinkedList
printList(list);

// Delete node with value 10
// In this case the key is ***not present***
deleteByKey(list, 10);

// Print the LinkedList
printList(list);

// ******DELETION AT POSITION******
// Delete node at position 0
// In this case the key is ***at head***
deleteAtPosition(list, 0);

// Print the LinkedList
printList(list);

// Delete node at position 2
// In this case the key is present ***in the
// middle***
deleteAtPosition(list, 2);

// Print the LinkedList
printList(list);

// Delete node at position 10
// In this case the key is ***not present***
deleteAtPosition(list, 10);

// Print the LinkedList
```

```
                printList(list);
        }
}
```

Output :

LinkedList: 1 2 3 4 5 6 7 8

1 found and deleted

LinkedList: 2 3 4 5 6 7 8

4 found and deleted

LinkedList: 2 3 5 6 7 8

10 not found

LinkedList: 2 3 5 6 7 8

0 position element deleted

LinkedList: 3 5 6 7 8

2 position element deleted

LinkedList: 3 5 7 8

10 position element not found

LinkedList: 3 5 7 8

Let's check the takeaway from this lecture

1) A linear collection of data elements where the linear node is given by means of pointer is called?

   A. Linked list
   B. Node list
   C. Primitive list
   D. None

2) What is the functionality of the following piece of code?

```
public int function(int data)
{
Node temp = head;
int var = 0;
while(temp != null)
{
if(temp.getData() == data)
```

```
{
return var;
}
var = var+1;
temp = temp.getNext();
}
return Integer.MIN_VALUE;
}
```

A. Find and delete a given element in the list
B. Find and return the given element in the list
C. Find and return the position of the given element in the list
D. Find and insert a new element in the list

Exercise

Q.1 Explain single link list it in detail.

Q.2 Explain with example operations on single linked list

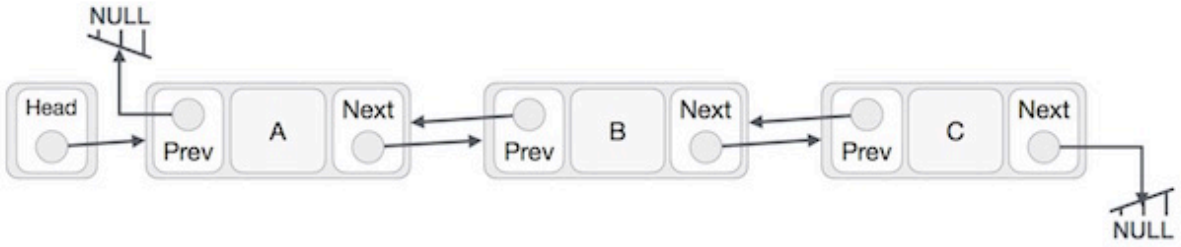
Learning from this lecture: Learners will be able understand and apply the concept of single linked list.

Lecture: 3

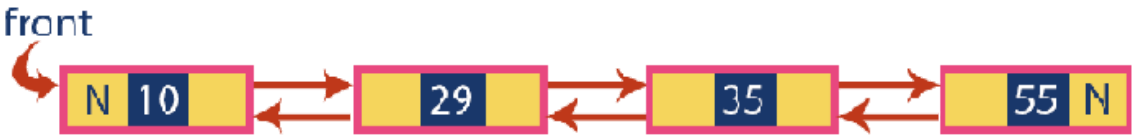
Double Linked list

A doubly linked list is an extension of the standard linked list that allows for traversal in both directions—forward and backward—making navigation more flexible compared to a singly linked list. In a doubly linked list, each node contains three fields: a data field and two link fields, referred to as `next` and `prev`. The `next` field points to the subsequent node in the list, while the `prev` field points to the preceding node. The first and last nodes of the list are marked with pointers to signify the start and end of the list, respectively. The last node's `next` field and the first node's `prev` field are set to `null` to indicate the boundaries of the list.

Doubly Linked List Representation



Example



- Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

- Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

- Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- Step 1 - Create a newNode with given value and newNode → previous as NULL.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, assign NULL to newNode → next and newNode to head.
- Step 4 - If it is not Empty then, assign head to newNode → next and newNode to head.

- Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- Step 1 - Create a newNode with given value and newNode → next as NULL.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty, then assign NULL to newNode → previous and newNode to head.
- Step 4 - If it is not Empty, then, define a node pointer temp and initialize with head.

- Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
- Step 6 - Assign newNode to temp → next and temp to newNode → previous.

o Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, assign NULL to both newNode → previous & newNode → next and set newNode to head.
- Step 4 - If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.
- Step 5 - Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.
- Step 7 - Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

o Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

o Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Check whether list is having only one node (temp → previous is equal to temp → next)
- Step 5 - If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)
- Step 6 - If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

o Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- Step 1 - Check whether list is Empty (head == NULL)

- Step 2 - If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Check whether list has only one Node (temp → previous and temp → next both are NULL)
- Step 5 - If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)
- Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)
- Step 7 - Assign NULL to temp → previous → next and delete temp.

o Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.
- Step 4 - Keep moving the temp until it reaches to the exact node to be deleted or to the last node.
- Step 5 - If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).
- Step 8 - If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).
- Step 9 - If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.
- Step 10 - If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).
- Step 11 - If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).
- Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

o Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty, then display 'List is Empty!!!' and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.
- Step 4 - Display 'NULL <--- '.
- Step 5 - Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node

- Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Let's check the takeaway from this lecture

1) Which of the following is false about a doubly linked list?
   a) We can navigate in both the directions
   b) It requires more space than a singly linked list
   c) The insertion and deletion of a node take a bit longer
   d) Implementing a doubly linked list is easier than singly linked list
2) What is a memory efficient double linked list?
   a) Each node has only one pointer to traverse the list back and forth
   b) The list has breakpoints for faster traversal
   c) An auxiliary singly linked list acts as a helper list to traverse through the doubly linked list
   d) A doubly linked list that uses bitwise AND operator for storing addresses
3) How do you calculate the pointer difference in a memory efficient double linked list?
   a) head xor tail
   b) pointer to previous node xor pointer to next node
   c) pointer to previous node – pointer to next node
   d) pointer to next node – pointer to previous node

Exercise

Q.1 Explain Doubly link list it in detail.

Q.2 Explain with example operations on double linked list

Learning from this lecture: Learners will be able understand and apply the concept of double linked list.

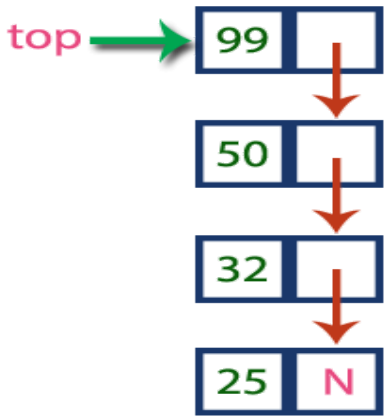
Lecture: 4 & 5

Implementation of linked list: Stack implementation using linked list:

A significant limitation of implementing a stack using an array is that it requires specifying the maximum number of elements at the outset. This fixed size can be problematic if the number of elements to be stored is unknown or variable. In contrast, a stack can be implemented using a linked list, which offers dynamic sizing and can accommodate an indefinite number of elements.

In a linked list implementation of a stack, each new element is added to the top of the stack, with the `top` pointer referencing the most recently added node. When an element needs to be removed, the node pointed to by `top` is deleted, and the `top` pointer is updated to the previous node in the list. Additionally, the `next` field of the node at the bottom of the stack should be set to `NULL`, indicating the end of the list. This approach eliminates the need to predefine the stack size and allows for flexible and efficient data management.

- Example



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

- Stack Operations using Linked List

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

- Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.
- Step 2 - Define a 'Node' structure with two members data and next.
- Step 3 - Define a Node pointer 'top' and set it to NULL.
- Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

- push(value) - Inserting an element into the Stack

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

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

- pop() - Deleting an Element from a Stack

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

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

- display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- Step 1 - Check whether stack is Empty (top == NULL).
- Step 2 - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
- Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).
- Step 5 - Finally! Display 'temp → data ---> NULL'.

Let's check the takeaway from this lecture

1) What does the following function do?

```
public Object some_func()throws emptyStackException
{
     if(isEmpty())
            throw new emptyStackException("underflow");
     return first.getEle();
}
```

a) pop
b) delete the top-of-the-stack element
c) retrieve the top-of-the-stack element
d) push operation

2) What does 'stack overflow' refer to?
a) accessing item from an undefined stack
b) adding items to a full stack
c) removing items from an empty stack
d) index out of bounds exception

Exercise

Q.1 Explain Stack implementation using link list with example.

Learning from this lecture: Learners will be able understand and apply the concept of stack using linked list.

Lecture: 7

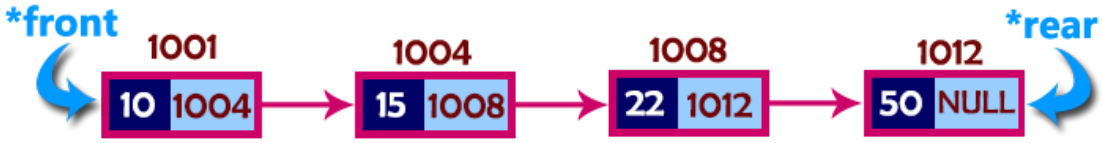
Queue implementation using linked list

A key limitation of implementing a queue using an array is that it requires specifying a fixed size at the outset, which can be problematic if the number of elements is unknown or varies. In contrast, a queue can be implemented using a linked list, which allows for dynamic sizing and can accommodate an unlimited number of elements without predefining the size.

In a linked list implementation of a queue, the `front` pointer refers to the first node in the queue, while the `rear` pointer refers to the last node. This setup enables efficient insertion of new elements

at the rear and removal of elements from the front, allowing the queue to handle a variable number of data values seamlessly.

- Example



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

- Operations

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

- Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.
- Step 2 - Define a 'Node' structure with two members data and next.
- Step 3 - Define two Node pointers 'front' and 'rear' and set both to NULL.
- Step 4 - Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

- enQueue(value) - Inserting an element into the Queue

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

- Step 1 - Create a newNode with given value and set 'newNode → next' to NULL.
- Step 2 - Check whether queue is Empty (rear == NULL)
- Step 3 - If it is Empty then, set front = newNode and rear = newNode.
- Step 4 - If it is Not Empty then, set rear → next = newNode and rear = newNode.

- deQueue() - Deleting an Element from Queue

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

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

- display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- Step 1 - Check whether queue is Empty (front == NULL).
- Step 2 - If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
- Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
- Step 5 - Finally! Display 'temp → data ---> NULL'.

Let's check the takeaway from this lecture

1) A linear collection of data elements where the linear node is given by means of pointer is called?

   A. Linked list
   B. Node list
   C. Primitive list
   D. None

2) What is the functionality of the following piece of code?

```
public int function(int data)
{
Node temp = head;
int var = 0;
while(temp != null)
{
if(temp.getData() == data)
{
return var;
}
var = var+1;
temp = temp.getNext();
}
return Integer.MIN_VALUE;
}
```

   A. Find and delete a given element in the list
   B. Find and return the given element in the list
   C. Find and return the position of the given element in the list
   D. Find and insert a new element in the list

Exercise

Q.1 Explain Queue implementation using link list with example.

Learning from this lecture: Learners will be able understand and apply the concept of queue using linked list

Lecture: 8

Applications of linked list:

- **Applications of Singly Linked List are as following:**

1. Singly linked lists are fundamental structures in computer science, used to implement essential data structures such as stacks and queues. They help manage data efficiently and are widely applicable in various contexts.

2. For instance, in hash maps, singly linked lists are employed to handle collisions, ensuring that multiple elements can be stored at the same hash index. Similarly, in everyday applications like a text editor, singly linked lists facilitate undo and redo operations by maintaining a sequence of actions.

3. In a photo viewer, singly linked lists can be used to manage a continuous slideshow, allowing seamless navigation through a series of images. Additionally, the concept of a singly linked list is analogous to train systems, where each train carriage (bogie) can be added to the end of the train or inserted between existing carriages, mimicking the linked list's dynamic structure.

- **Applications of Circular Linked List are as following:**

Circular doubly linked lists offer versatile applications due to their unique properties. They maintain references to both the next and previous nodes, enabling efficient navigation and updates.

- **Queues**: Circular doubly linked lists can be used to implement queues by keeping a pointer to the last inserted node. The front of the queue is easily accessed via the next pointer of the last node.
- **Advanced Data Structures**: They are utilized in the implementation of advanced data structures like Fibonacci heaps, where efficient access to both the next and previous nodes is crucial.
- **Operating Systems**: Circular lists are often employed in round-robin scheduling mechanisms within operating systems. This approach ensures that each process or user receives an equal time slice, reflecting the principle of fairness and rotation.
- **Multiplayer Games**: In gaming, circular lists are used to cyclically switch between players or game elements, maintaining a continuous loop of actions.
- **Software Applications**: Applications like Photoshop, Microsoft Word, or paint programs use circular lists to manage undo operations, allowing users to revert to previous states in a circular manner.

- **Applications of Doubly Linked List are as following:**

1. Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.
2. In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.
3. Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.
4. It is used in music playing system where you can easily play the previous one or next one song as many times one person wants to. Basically it provides full flexibility to perform functions and make the system user-friendly.
5. In many operating systems, the thread scheduler (the thing that chooses what processes need to run at which times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).
6. It is used in a famous game concept which is a deck of cards.

Let's check the takeaway from this lecture

1) In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?
   a) Insertion
   b) Deletion
   c) To empty a queue
   d) Both Insertion and To empty a queue
2) In linked list implementation of a queue, where does a new element be inserted?
   a) At the head of link list
   b) At the centre position in the link list

c) At the tail of the link list
d) At any position in the linked list

3) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?
   a) Only front pointer
   b) Only rear pointer
   c) Both front and rear pointer
   d) No pointer will be changed
4) In case of insertion into a linked queue, a node borrowed from the _____ list is inserted in the queue.
   a) AVAIL
   b) FRONT
   c) REAR
   d) NULL
5) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?
   a) Only front pointer
   b) Only rear pointer
   c) Both front and rear pointer
   d) No pointer will be changed

Exercise

Q.1 Give application of single link list

Q.2 Give application of double link list

Learning from this lecture: Learners will be able understand the different applications of linked list.

Objective Questions

1. The Linked List is an ----------

a. Linear data structures

b. Non Linear data structures

c. Not A and B

d. None of above

2. Which of the following name is relate to List?

a. Linked Stack

b. push

c. Piles

d. Push-down lists

3. Suppose cursor refers to a node in a linked list What Boolean expression will be true when cursor refers to the tail node of the list?

a. (cursor == null)

b. (cursor.link == null)

c. (cursor.data == null)

d. (cursor.data == 0.0)

6. The term "push" and "pop" is not related to the

a. array

b. lists

c. Linked List

d. all of above

Subjective Questions

1. What is Linked List? Explain its operations.
2. Explain array implementation of linked list
3. Write a program for creating singly list
4. Explain linked stack and queue
5. Explain Circular linked list and doubly linked list

University Questions

1. Write a program to create linked list. (Dec 10)(10 marks)
2. Explain Linked stack and queue (May 05)(10 marks)
3. Explain doubly and circular list (May 09) (10 marks)
4. Write Short note on array implementation of linked list (May 05)(06 marks)

References:

Sr. No.	Title	Authors	Publisher	Edition	Year
1	Data Structures using C	ReemaThareja	Oxford	Second Edition	2014
2	Data Structures: A Pseudocode Approach with C	Richard F. Gilberg&Behrouz A., Forouzan	CENGAGE Learning	Second Edition	2011
3	Data Structures Using C	Aaron M Tenenbaum, YedidyahLangsam, Moshe J Augenstein	Pearson	Second Edition	2006
4	Data Structures with C	SeymoreLipschutz	Tata McGraw-Hill	India Special Edition	2011

https://www.geeksforgeeks.org/introduction-to-arrays/
https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html
https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Self-Evaluation

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		**Tick Your choice**
1.	Do you understand the term Linked List?	o Yes o No
2.	Do you understand the Applications of Lined list?	o Yes o No
3.	Do you know the different operations of Linked List?	o Yes o No
4.	Are you able to implement Linked List with its operation?	o Yes o No
5.	Do you understand module ?	o Yes, Completely. o Partialy. o No, Not at all.

Module:04

Trees

Lecture: 1

Motivation:

Trees are well known as a non-linear Data Structure. It doesn't store data in a linear way. It organizes data in a hierarchical way. A tree is an important data structure of computer science which is useful for storing hierarchically ordered data. Unlike stack, queue and linear lists (arrays and linked lists) which are used to store linear data like marks of students in a class, list of tasks to be done, etc. Trees are used to store non-linear data structures in a hierarchical order. A family tree is the most common example of hierarchical data. Directory structure, corporate structure, etc. are also common examples of hierarchical data.

Syllabus:

Lecture no	Content	Duration (Hr)	Self-Study (Hrs)
1	Tree terminologies	1	1
2	Binary tree and its types.	1	1

3	Binary tree operations and implementation	1	2
4	Tree traversing techniques.	1	2
5	Expression Tree	1	2
6	AVL tree- Height balanced binary tree	1	2
7	B tree	1	2
8	B+ Tree	1	2
9	Representing list as binary tree	1	2

Learning Objective:

- Learner should know the key concept of Tree.
- Learners shall be able to illustrate types of trees and their properties.
- Learners shall be able to write tree algorithms.
- Learners shall be able to apply traversal techniques on given data.

Theoretical Background:

In computer science, a tree is a widely used abstract data type that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes. A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

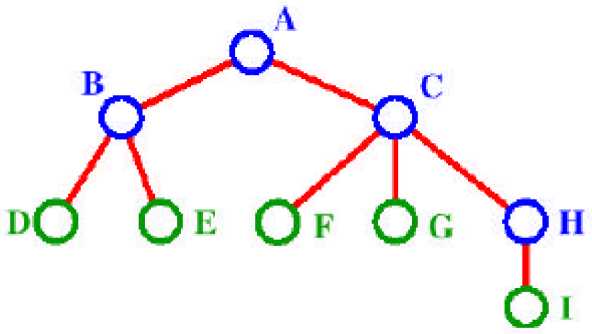
Key Definitions:

1. Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.
2. Binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes
3. Full Binary Tree: A Binary Tree is a full binary tree if every node has 0 or 2 children.
4. Complete Binary Tree: A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.
5. Perfect Binary Tree A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.
6. Binary Search Tree is a node-based binary tree data structure which has the following properties:
    - The left subtree of a node contains only nodes with keys lesser than the node's key.
    - The right subtree of a node contains only nodes with keys greater than the node's key.
    - The left and right subtree each must also be a binary search tree.
7. AVL tree satisfies the property of the *binary tree* as well as of the *binary search tree*. It is a self-balancing binary search tree that was invented by *Adelson Velsky Lindas* (AVL).
8. B-tree is a balanced m-way tree where m defines the order of the tree.

Course Content

Tree terminologies

The important terms related to tree data structure are-



- Root node: Top element of tree. The root node has no parents.
  A is the root node.
- Parent node: Immediate node joined by a single line segment when traversing up the tree. B is the parent of D and E.
- Children: All immediate nodes joined by a line segment when traversing one level down the tree. D and E are children of B.
- Sibling: Children of the same parent. C is a sibling of B.
- Ancestors: All nodes along the path to the root. H, C, and A are ancestors of I.
- External node: Node without children. Also called a leaf. D, E, F, G, and I are external nodes or leaves.
- Internal node: Node with one or more children. A, B, C and H are internal nodes.
- Depth (level): Number of line segments from the root to a node (or the number of ancestors). The depth of B is 1. The depth of I is 3.
- Leaf Nodes: Nodes that do not have any children are called leaf nodes. They are also referred to as terminal nodes. (Nodes D,E,F,G,I are leaf nodes)
- Height: The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree.
- Subtree: A subtree of a tree T is a tree consisting of a node in T and all of its descendants in T. (This is different from the formal definition of subtree used in graph theory.) The subtree corresponding to the root node is the entire tree;
- Proper Subtree: The subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).
- Directed Edge: A directed edge refers to the link from the parent to the child.
- In-Degree: In-degree of a node is the number of edges arriving at that node.
- Out Degree: Out-degree of a node is the number of edges leaving that node.

A node can be instantiated using a user-defined data structure called "Node," exemplified below:

```
class Node {
   int data;
   Node left;
   Node right;
```

}

The structure above defines a node consisting of three attributes: a data field, a left pointer of the Node type, and a right pointer of the Node type.

Let's check the take away from this lecture

1) The number of edges from the node to the deepest leaf is called _____ of the tree.
   a) Height
   b) Depth
   c) Length
   d) Width
2) What is a full binary tree?
   a) Each node has exactly zero or two children
   b) Each node has exactly two children
   c) All the leaves are at the same level
   d) Each node has exactly one or two children

Exercise

Q.1 What is tree?

Q.2 Explain depth of tree with example.

Questions/Problems for practice:

Q.3 What is the average case time complexity for finding the height of the binary tree?

Learning from this lecture: Learners will be able to understand tree terminologies.

Lecture: 2

Binary tree and its types

Learning objective:

In this lecture learners will able to understand Binary tree and its types.

Binary tree

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a binary tree has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

A parent node has two child nodes: the left child and right child.

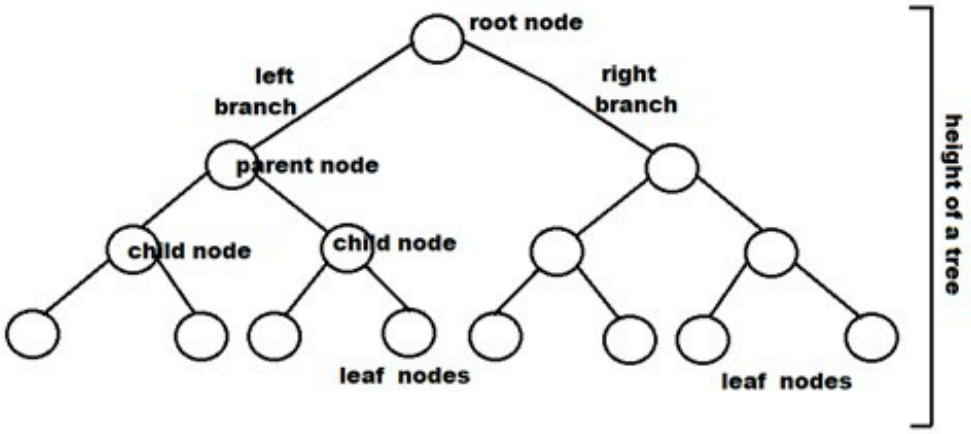
Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.

Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. Minimum number of nodes in a binary tree of height H = H + 1
2. Maximum number of nodes in a binary tree of height H = 2H+1 – 1
3. Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1

4. Maximum number of nodes at any level ‘L’ in a binary tree =2L

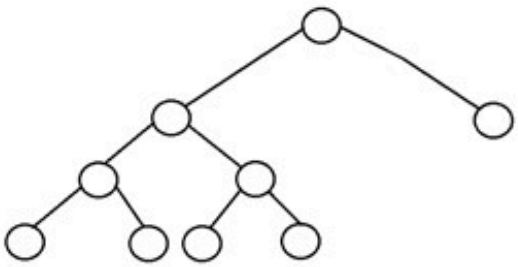


Types of Binary Trees

- **1. Full Binary Tree**
- **2. Complete Binary Tree**
- **3. Perfect Binary Tree**
- **4. Balanced Binary Tree**
- **5. Degenerate Binary Tree**

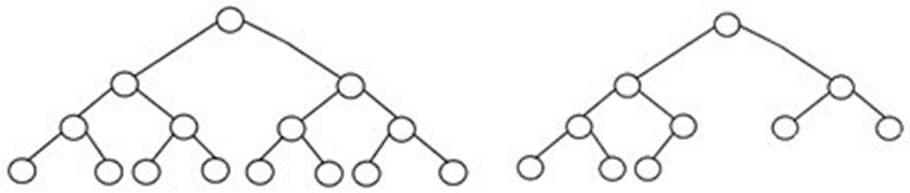
Full Binary Tree

- It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node**.**



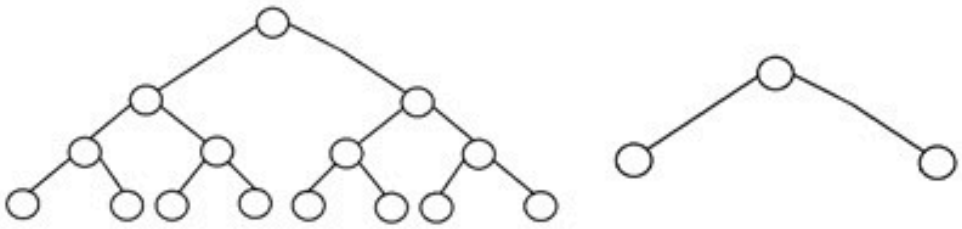
Complete Binary Tree

A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side.



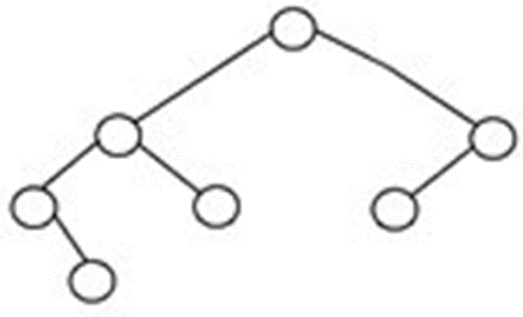
Perfect Binary Tree:

- A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height 'h' has 2h – 1 node.



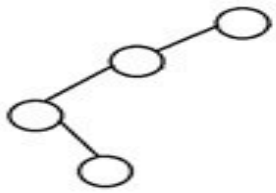
Balanced Binary Tree

A binary tree is said to be 'balanced' if the tree height is O(logN), where 'N' is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree.



Degenerate Binary Tree

A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child. Such trees are similar to a linked list performance-wise.



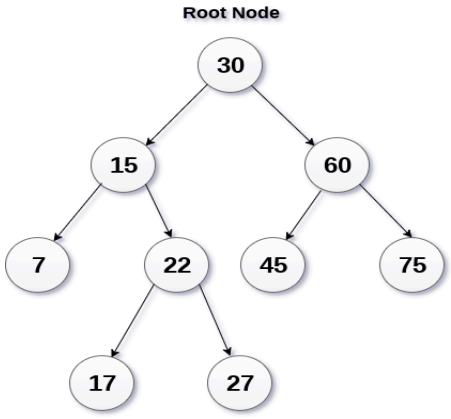
Lecture: 3

Binary search tree operations and implementation

Learning objective: In this lecture learners will able to understand BST properties and different operations that we can perform on BST.

Binary search tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Binary search tree operations

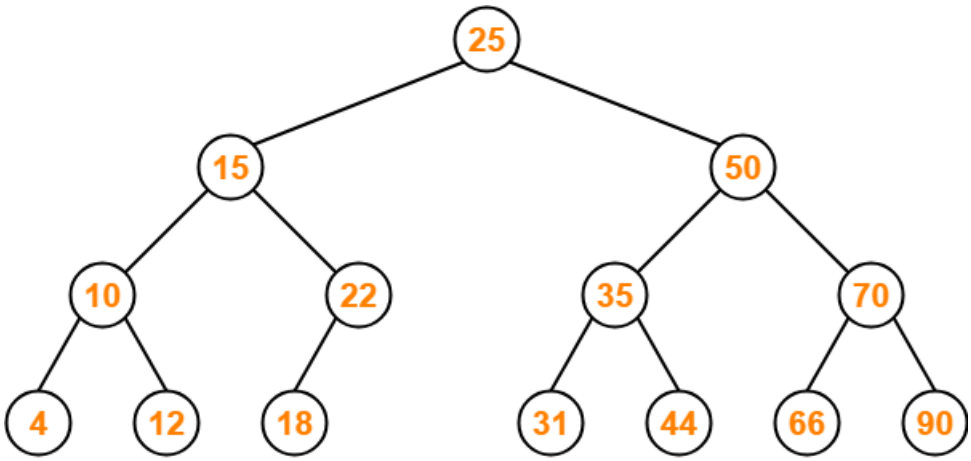
1. Search Operation
2. Insertion Operation
3. Deletion Operation

1. Search Operation

- Search Operation is performed to search a particular element in the Binary Search Tree.

For searching a given key in the BST,

- Compare the key with the value of root node.
- If the key is present at the root node, then return the root node.
- If the key is greater than the root node value, then recur for the root node's right subtree.
- If the key is smaller than the root node value, then recur for the root node's left subtree.



Binary Search Tree

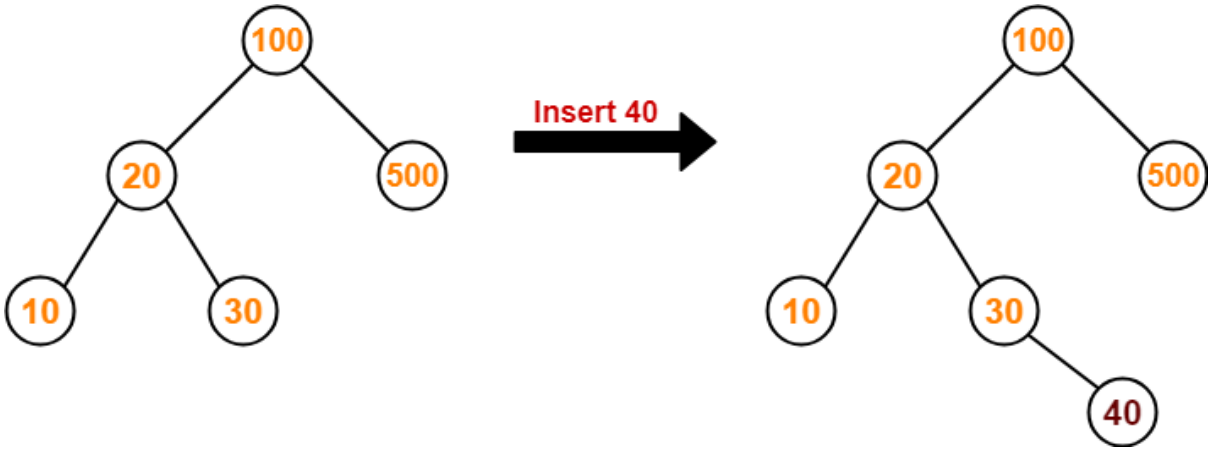
Example: Consider key = 45 has to be searched in the given BST-

- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we can conclude that 45 is not present in the above BST.

2. Insertion Operation

- Insertion Operation is performed to insert an element in the Binary Search Tree.
- The insertion of a new key always takes place as the child of some leaf node.
- For finding out the suitable leaf node,
    - Search the key to be inserted from the root node till some leaf node is reached.
    - Once a leaf node is reached, insert the key as child of that leaf node.

Example: Consider the following example where key = 40 is inserted in the given BST-



- We start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

Let's check the take away from this lecture

1)Which of the following is false about a binary search tree?
a) The left child is always lesser than its parent
b) The right child is always greater than its parent
c) The left and right sub-trees should also be binary search trees
d) In order sequence gives decreasing order of elements

2) The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)? (GATE CS 2004)

a) 2
b) 3
c) 4
d) 5

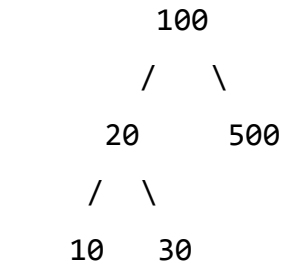
Exercise

Q.1 What is a binary search tree**?** State its properties.

Q.2 Explain on insertion is performed on BST.

Questions/Problems for practice:

Q.3 Insert node 40 in following tree.



Learning from this lecture: Learners will be able to state various properties of BST and also can apply searching, insertion, deletion operation on BST.

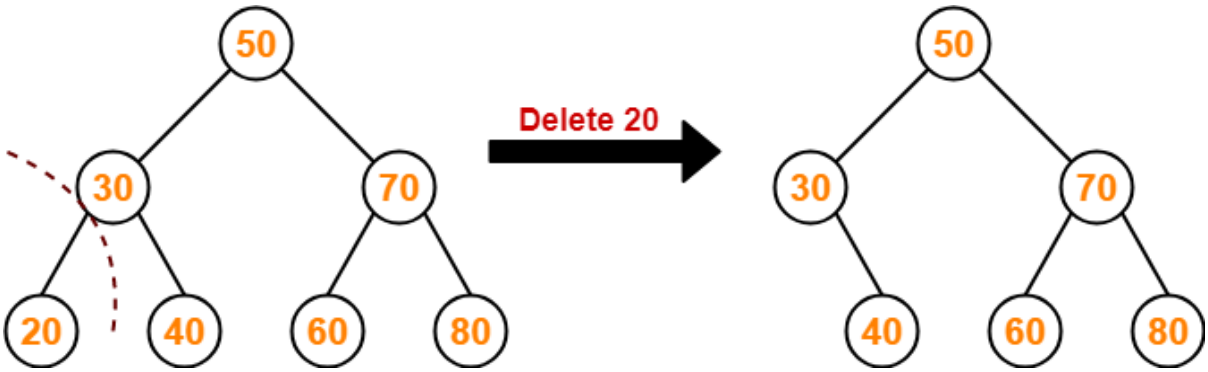
Binary search tree operations

Learning objective: In this lecture learners will able to apply delete operation on Binary search tree.

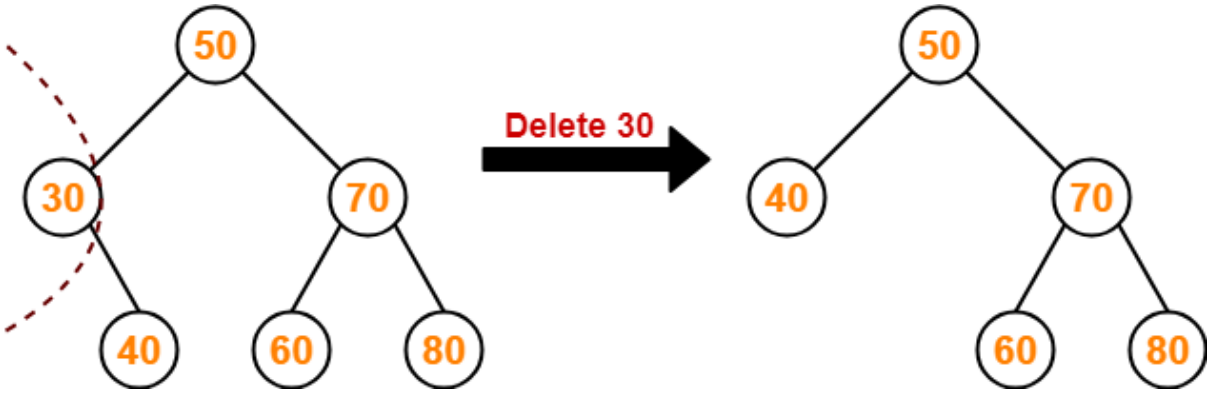
Deletion Operation

- Deletion Operation is performed to delete a particular element from the Binary Search Tree.
- When it comes to deleting a node from the binary search tree, following three cases are possible-

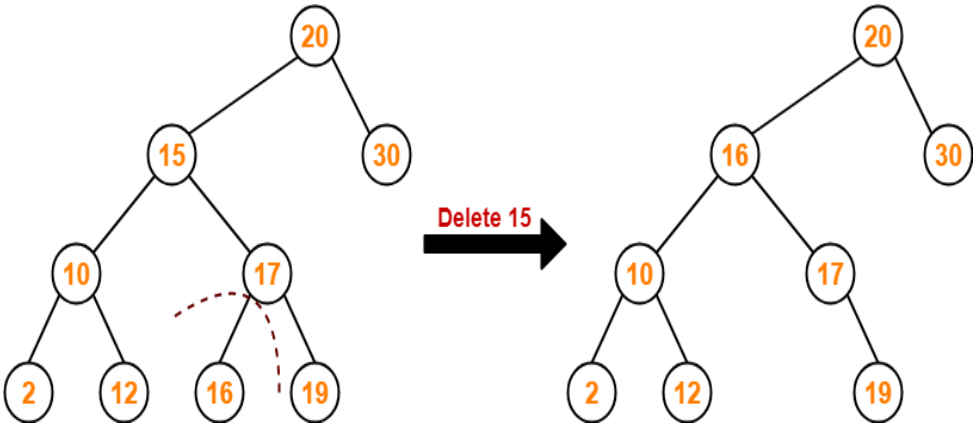
1. Deletion of A Node Having No Child (Leaf Node)
   - Just remove / disconnect the leaf node that is to deleted from the tree.
   - Consider the following example where node with value = 20 is deleted from the BST:



2. Deletion of A Node Having Only One Child:
   - Just make the child of the deleting node, the child of its grandparent.
   - Consider the following example where node with value = 30 is deleted from the BST-

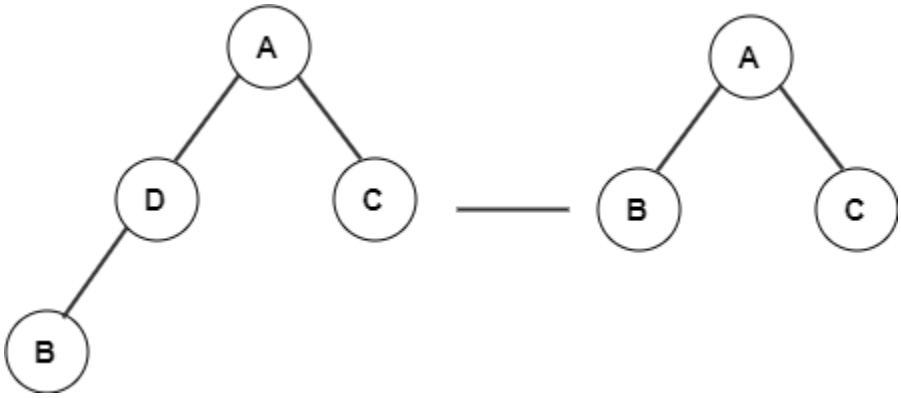


3. Deletion of A Node Having Two Children
   - Visit to the right subtree of the deleting node.
   - Pluck the least value element called as in-order successor.
   - Replace the deleting element with its in-order successor.
   - Consider the following example where node with value = 15 is deleted from the BST-



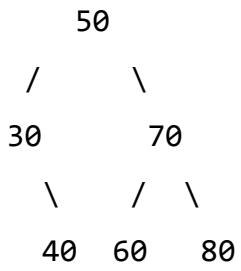
Let's check the take away from this lecture

1) **What operation does the following diagram depict?**



a) inserting a leaf node
b) inserting an internal node
c) **deleting a node with 0 or 1 child**
d) deleting a node with 2 children

Exercise:
Q.1 Explain deletion process of BST in detail
Questions/problems for practice:
Q. 2 Delete 30 and then 50 from following BST.



Learning from this lecture: Learners will be able to state various properties of BST and also can apply deletion operation on BST.

Lecture: 4

Learning objective:

In this lecture learners will able to understand and apply traversal technique on tree data structure.

In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and the right subtree.

Properties of Binary Trees:
Some of the important properties of a binary tree are as follows:

1. Minimum number of nodes in a binary tree of height H = H + 1
2. Maximum number of nodes in a binary tree of height H = 2H+1 – 1
3. Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1

4. Maximum number of nodes at any level ‘L’ in a binary tree =2L

Traversing a tree means visiting all the nodes of a tree in order.

There are three different methods of traversing a binary tree:

- pre-order traversal
- in-order traversal
- post-order traversal

In each case, the algorithms for traversal are recursive - they call themselves.

1. **Pre-order traversal**
   - Start at the root node
   - Traverse the left subtree
   - Traverse the right subtree

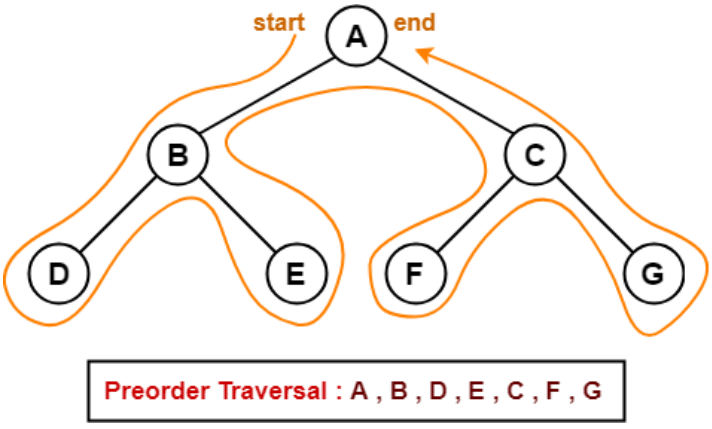


Fig. Pre-order traversal

Applications-

- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.

2. **In-order traversal**
   - Traverse the left sub tree
   - Visit the root
   - Traverse the right sub tree

Application-

- Inorder traversal is used to get infix expression of an expression tree.

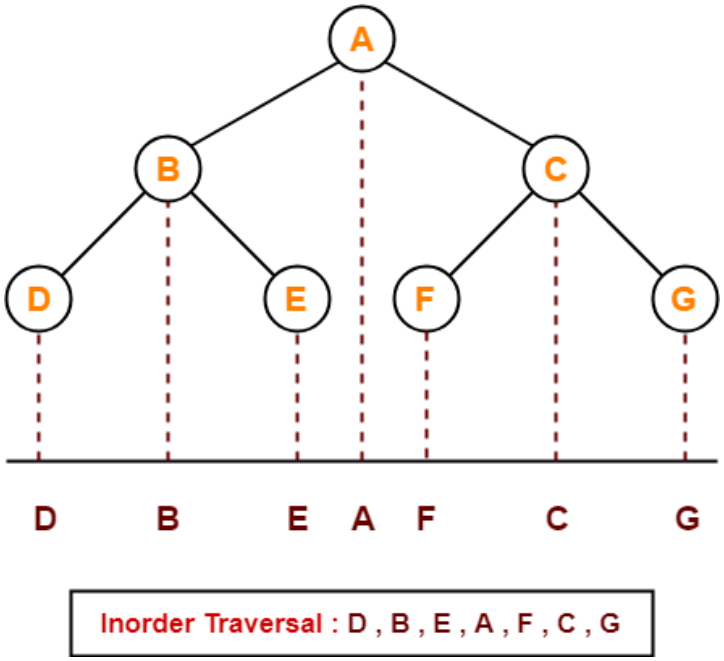


Fig. In-order traversal

3.Post-order traversal

- Traverse the left sub tree i.e. call Postorder (left sub tree)
- Traverse the right sub tree i.e. call Postorder (right sub tree)
- Visit the root

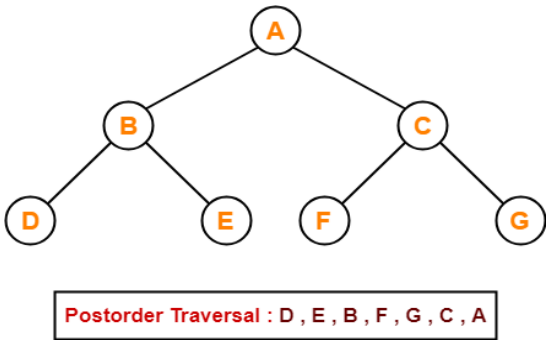


Fig. Post-order traversal

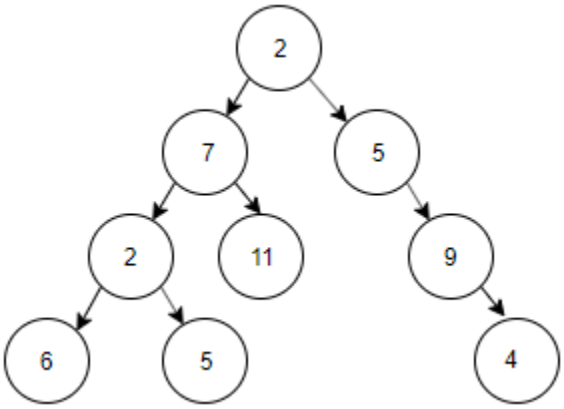
Application-

- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree.
- This is because it deletes the children first and then it deletes the parent.

Given a binary tree, find its preorder,inorder and postorder traversal.

Let's check the take away from this lecture

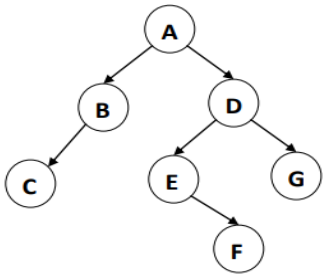
1) For the tree below, write the pre-order traversal.



a) **2, 7, 2, 6, 5, 11, 5, 9, 4**
b) 2, 7, 5, 2, 6, 9, 5, 11, 4
c) 2, 5, 11, 6, 7, 4, 9, 5, 2
d) 2, 7, 5, 6, 11, 2, 5, 4, 9

2) What is the space complexity of the in-order traversal in the recursive fashion? (d is the tree depth and n is the number of nodes)
a) O(1)
b) O(nlogd)
c) O(logd)
d**) O(d)**

Exercise
Q.1 Illustrate tree traversal techniques.
Q.2 Explain applications of each type of traversal.
Questions/Problems for practice:
Q.3 Given a binary tree, find its preorder,inorder and postorder traversal.



Learning from this lecture: Learners will be able to apply traversal concept on tree data structure.

Lecture: 5

Expression tree

Learning objective: In this lecture, learners will able to understand the Expression tree.

What is an Expression Tree in Data Structure?

A mathematical expression can be expressed as a binary tree using expression trees. Expression trees are binary trees with each leaf node serving as an operand and each internal (non-leaf) node serving as an operator.

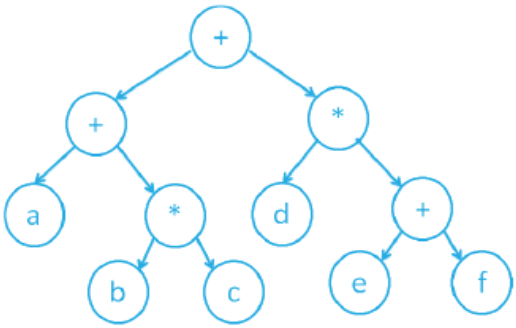
Properties of Expression Tree in Data Structure:

Let's go through some expression tree properties.

The operands are always represented by the leaf nodes. These operands are always used in the operations.
The operator at the root of the tree is always given top priority.
When compared to the operators at the bottom of the tree, the operator at the bottom is always given the lowest priority.
Because the operand is always present at a depth of the tree, it is given the highest priority of all operators.
The expression tree can be traversed to evaluate prefix expressions, postfix expressions, and infix expressions.
In summary, the value present at the depth of the tree has the highest priority when compared to the other operators located at the top of the tree. The expression tree is immutable, and once built, we cannot change or modify it further, so to make any changes, we must completely construct the new expression tree.

The given expression can be evaluated using the expression tree in data structure.

a + (b * c) + d * (e + f)



Uses of an Expression Tree in Data Structure
The following is the primary application of these expression trees in data structure:

It evaluates, analyses, and modifies diverse phrases. It can also be used to determine the associativity of each operator in an expression.
As an example:
The + operator is left-associative, while the / operator is right-associative. The expression trees helped to solve the conundrum of this associativity.

A context-free grammar is used to create these expression trees.

It is a key component in compiler design and is part of the semantic analysis step.

The expression trees are mostly used to create complex expressions that can be quickly evaluated.

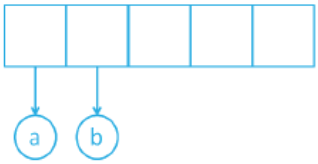
Construction of an Expression Tree in Data Structure

A stack is used to build an expression tree. For each character, we cycle through the input expressions and do the following.

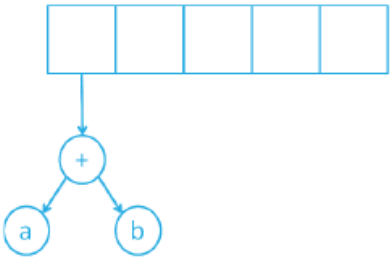
If a character is an operand, add it to the stack.
If a character is an operator, pop both values from the stack and make both its children and push the current node again.
Finally, the stack's lone element will be the root of an expression tree.

Let us understand this above process using a postfix expression. The implementation of the expression tree is described for the below expression**.**

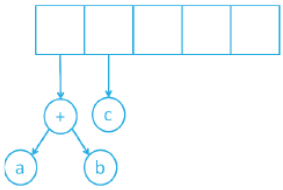
a b + c *
Step1 : The first two symbols are operands, for which we generate a one-node tree and push a reference to the stack**.**



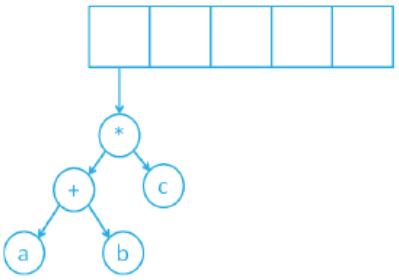
Step2 : Next, read a'+' symbol to pop two pointers to the tree, form a new tree, and push a pointer to it onto the stack.



Step3 : In the next stage 'c' is read, we build a single node tree and store a pointer to it on the stack



Step4 : Finally, we read the last symbol' ', pop two tree pointers, and build a new tree with a,''as root, and a pointer to the final tree remains on the stack.



Let's check the take away from this lecture

1) What is the main purpose of an Expression Tree in Data Structure?

A) To store a sequence of characters.

B) To represent linear data structures.

C) To evaluate, analyze, and modify mathematical expressions.

D) To perform sorting operations on arrays.

2) **What data structure is commonly used to construct an Expression Tree from an expression?**

A) Linked List

B) Queue

C) Stack

D) Binary Search Tree

Exercise:

Q.1 What is the primary application of Expression Trees in Data Structure?

Q.2 How are operands and operators represented in an Expression Tree?

Learning from the lecture: Learners will able to understand Threaded binary tree.

Lecture: 6

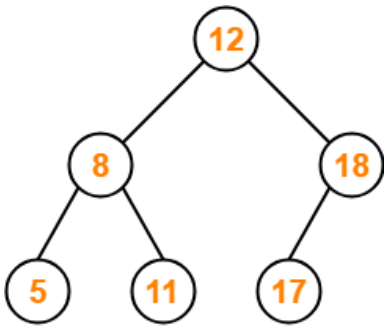
Height balanced binary tree-AVL

Learning objective: In this lecture learners will able to know about AVL trees. learners will able to apply insertion, deletion on AVL tree

- AVL trees are special kind of binary search trees.
- In AVL trees, height of left subtree and right subtree of every node differs by at most one.
- AVL trees are also called as self-balancing binary search trees.

In AVL tree,

- Balance factor is defined for every node.
- Balance factor of a node = Height of its left subtree – Height of its right subtree
- In AVL tree, Balance factor of every node is either 0 or 1 or -1.



AVL Tree Example

This tree is an AVL tree because-

- It is a binary search tree.
- The difference between height of left subtree and right subtree of every node is at most one.

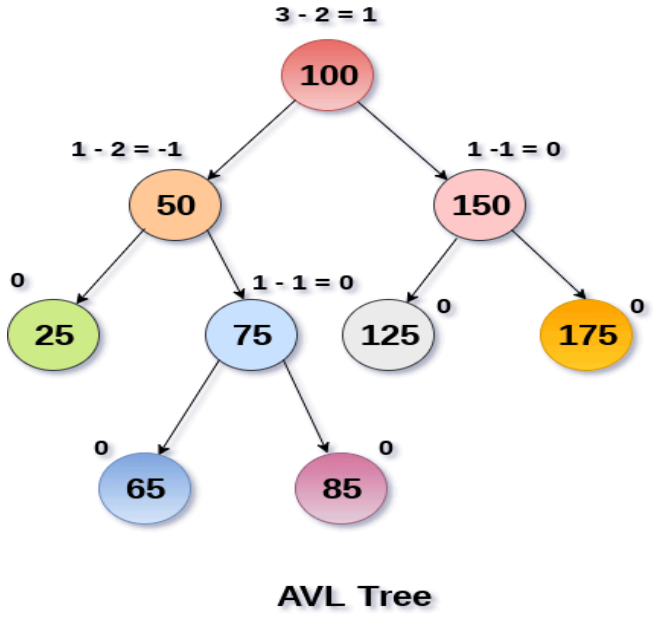


Fig. Balance factor of a node in AVL

- After performing any operation on AVL tree, the balance factor of each node is checked.
- There are following two cases possible-

Case 1:

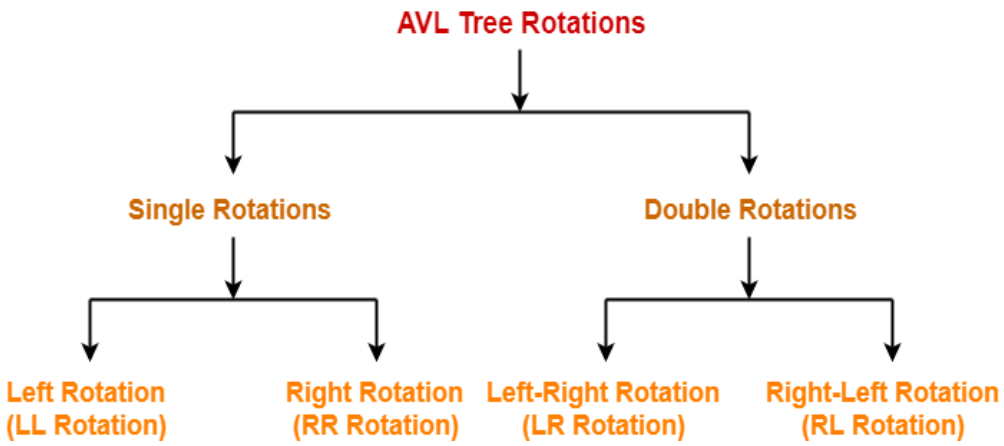
- After the operation, the balance factor of each node is either 0 or 1 or -1.
- In this case, the AVL tree is considered to be balanced.
- The operation is concluded.

Case 2:

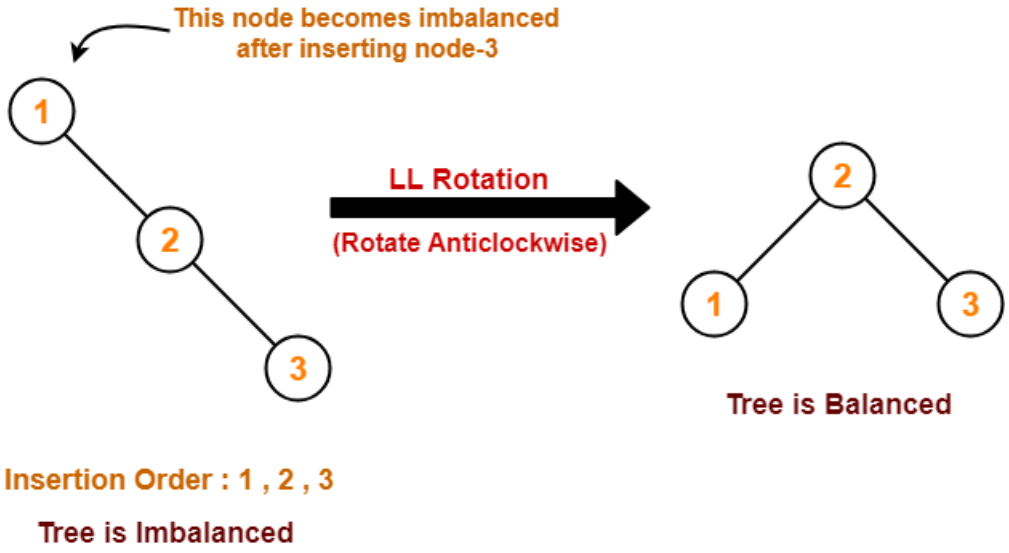
- After the operation, the balance factor of at least one node is not 0 or 1 or -1.
- In this case, the AVL tree is considered to be imbalanced.
- Rotations are then performed to balance the tree.

AVL Tree Rotations-

There are 4 kinds of rotations possible in AVL Trees-



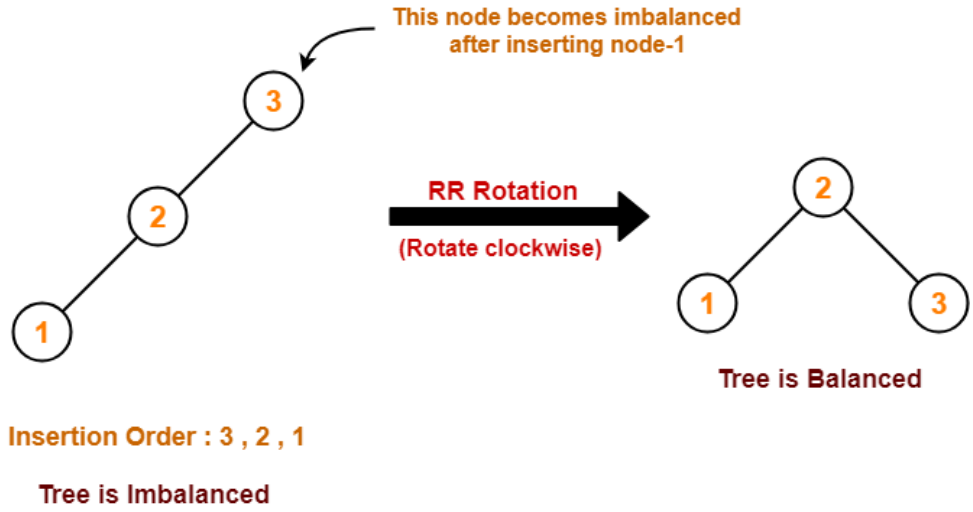
Case-1:



Insertion Order : 1 , 2 , 3

Tree is Imbalanced

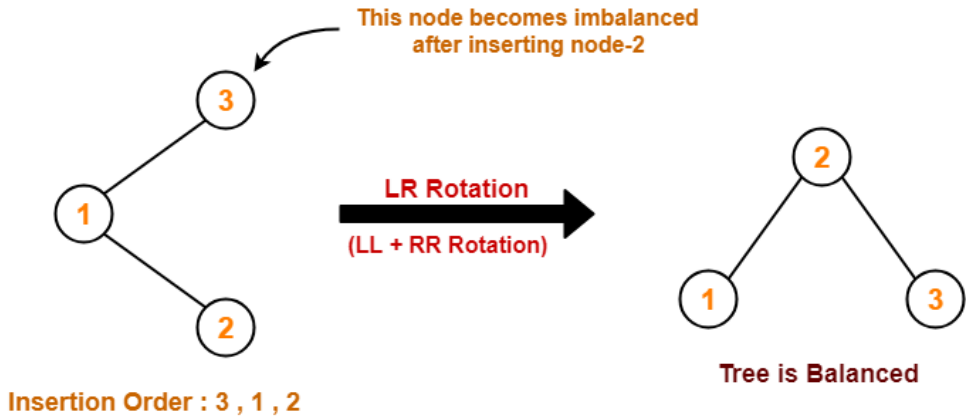
Case-2:



Insertion Order : 3 , 2 , 1

Tree is Imbalanced

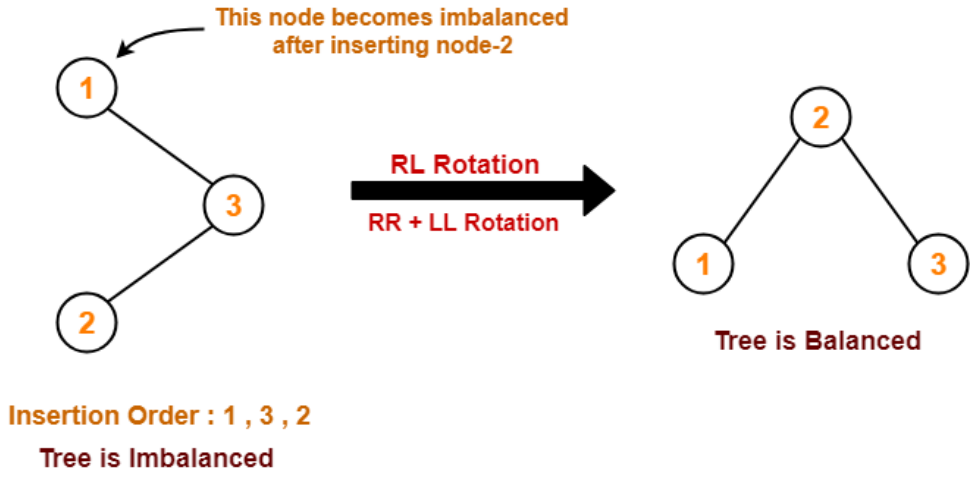
Case-3:



Insertion Order : 3 , 1 , 2

Tree is Imbalanced

Case-4:



Let's check the take away from this lecture

1) Why we need to a binary tree which is height balanced?

a) to avoid formation of skew trees
b) to save memory
c) to attain faster memory access
d) to simplify storing

2) What is an AVL tree?

a) a tree which is balanced and is a height balanced tree
b) a tree which is unbalanced and is a height balanced tree
c) a tree with three children
d) a tree with atmost 3 children

Exercise:
Q.1 Explain rotations of AVL tree with example.
Questions/problems for practice:
Q. 2 Explain balance factor of AVL tree with the help of suitable example.

Learning from the lecture: In this lecture learners will be able to understand AVL tree concepts.

Insertion in AVL Tree

To insert an element in the AVL tree, follow the following steps-

- Insert the element in the AVL tree in the same way the insertion is performed in BST.
- After insertion, check the balance factor of each node of the resulting tree.
- Balance factor of only those nodes will be affected that lies on the path from the newly inserted node to the root node.
- To check whether the AVL tree is still balanced or not after the insertion,
  - There is no need to check the balance factor of every node.

- Check the balance factor of only those nodes that lies on the path from the newly inserted node to the root node.

After inserting an element in the AVL tree,

- If tree becomes imbalanced, then there exists one particular node in the tree by balancing which the entire tree becomes balanced automatically.
- To re balance the tree, balance that particular node.

To find that particular node,

- Traverse the path from the newly inserted node to the root node.
- Check the balance factor of each node that is encountered while traversing the path.
- The first encountered imbalanced node will be the node that needs to be balanced.

To balance that node,

- Count three nodes in the direction of leaf node.
- Then, use the concept of AVL tree rotations to re balance the tree.

Example: Construct AVL Tree for the following sequence of numbers-

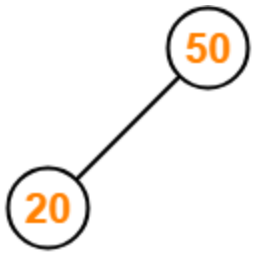
50 , 20 , 60 , 10 , 8

Step-01: Insert 50

Tree is Balanced

Step-02: Insert 20

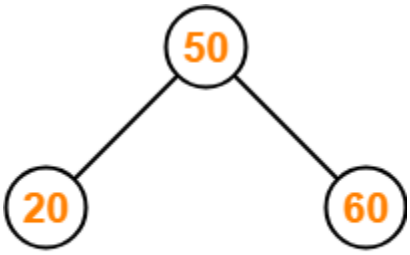
- As 20 < 50, so insert 20 in 50's left sub tree.



Tree is Balanced

Step-03: Insert 60

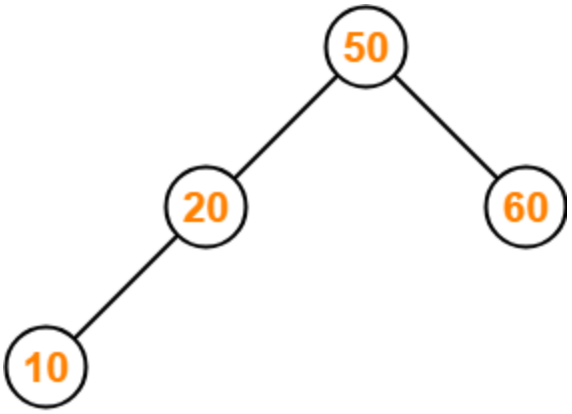
- As 60 > 50, so insert 60 in 50's right sub tree.



Tree is Balanced

Step-04: Insert 10

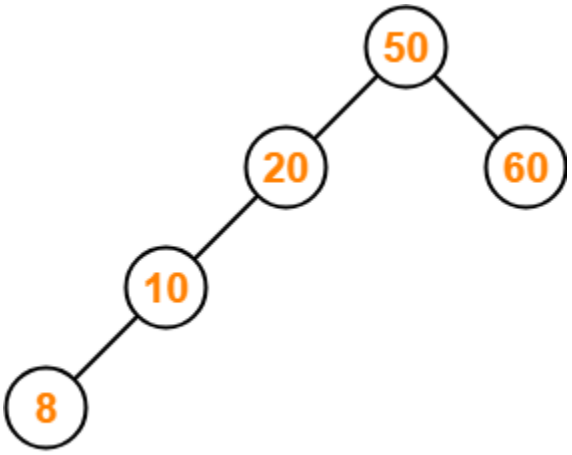
- As 10 < 50, so insert 10 in 50's left sub tree.
- As 10 < 20, so insert 10 in 20's left sub tree.



Tree is Balanced

Step-05: Insert 8

- As 8 < 50, so insert 8 in 50's left sub tree.
- As 8 < 20, so insert 8 in 20's left sub tree.
- As 8 < 10, so insert 8 in 10's left sub tree.

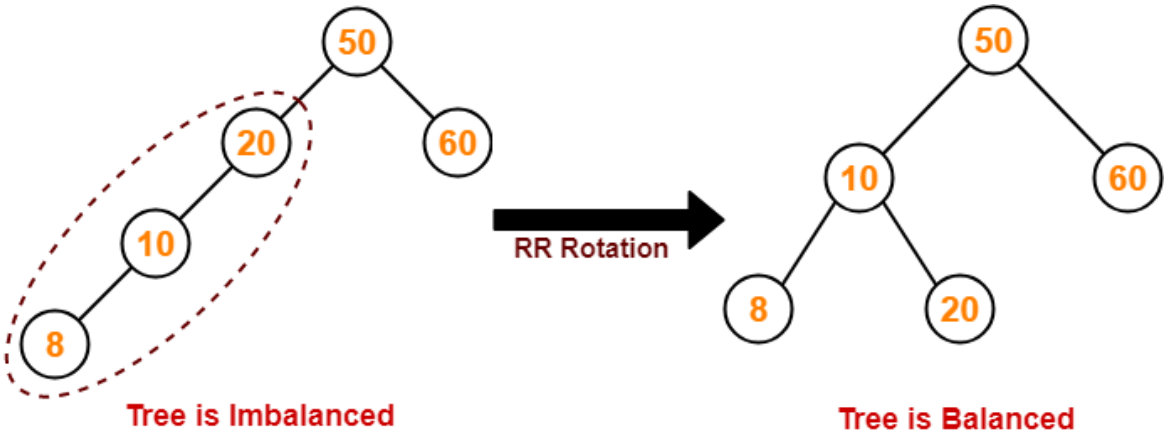


Tree is Imbalanced

To balance the tree,

- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Let's check the take away from this lecture

1) **Given an empty AVL tree, how would you construct AVL tree when a set of numbers are given without performing any rotations?**
   a) just build the tree with the given input
   b) find the median of the set of elements given, make it as root and construct the tree
   c) use trial and error
   d) use dynamic programming to build the tree

Exercise:

Q.1 Construct AVL Tree for the following sequence of numbers-
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

Q.2 Construct an AVL tree having the following elements
H, I, J, B, A, E, C, F, D, G, K, L

Learning from the above lecture: Learners will able to apply insertion, deletion on AVL tree

Lecture: 7

B Tree

Learning objective: In this lecture learners will able to understand B Tree mechanisms.

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reasons of using B tree is its capability to

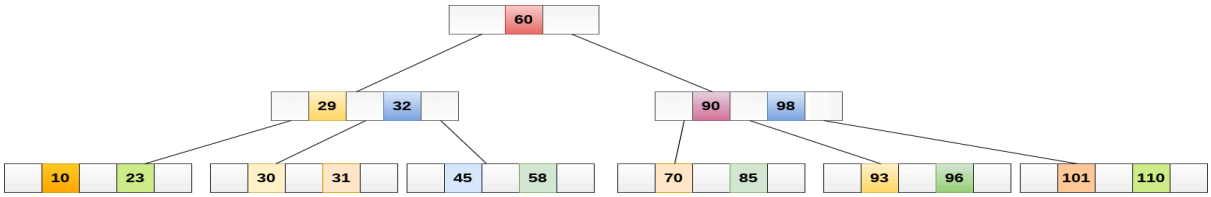
store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following:
Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.

1. Since, 40<49<56, traverse right sub-tree of 40.
2. 49>45, move to right. Compare 49.
3. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.

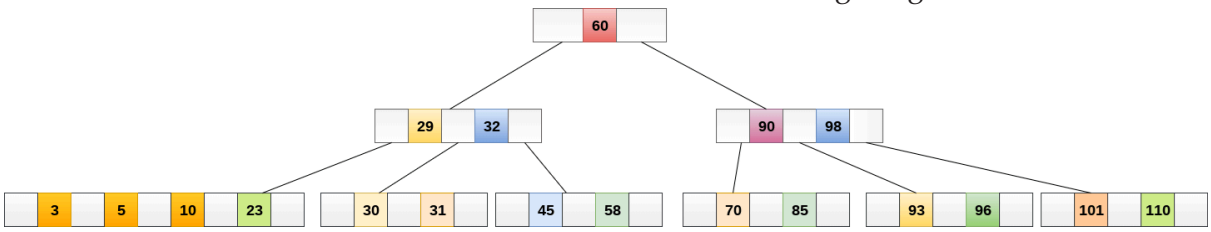
Inserting:

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

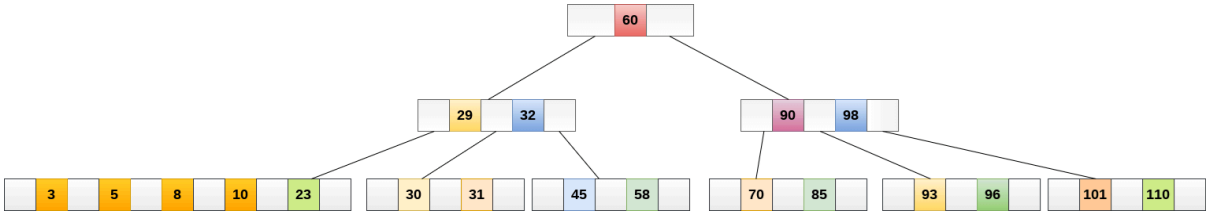
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.

3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

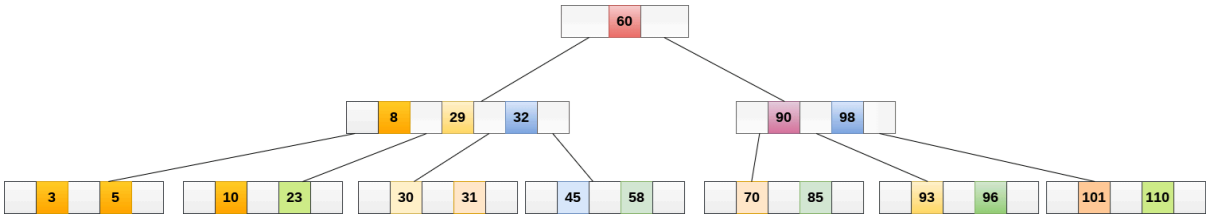
Example:
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node now contains 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.

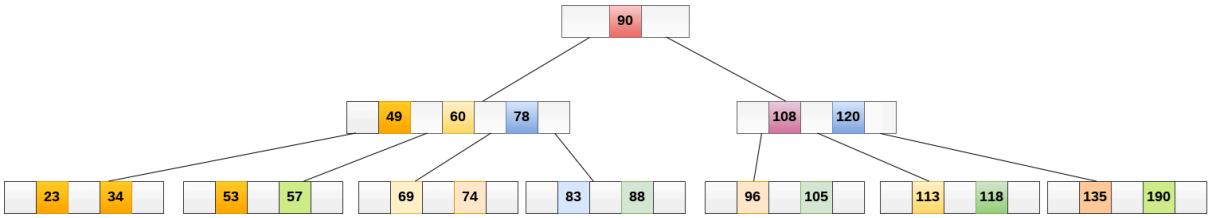
- If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

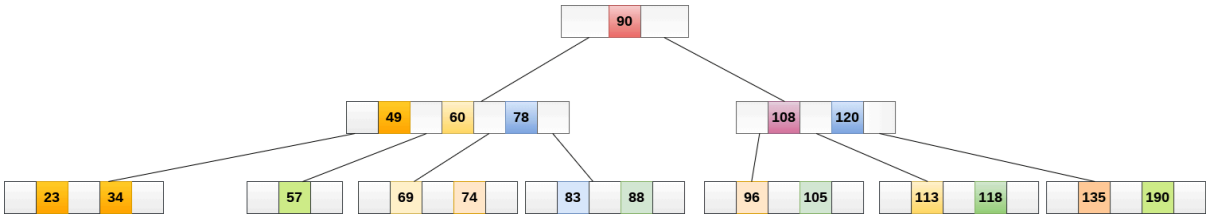
If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, the successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example:

Delete the node 53 from the B Tree of order 5 shown in the following figure.

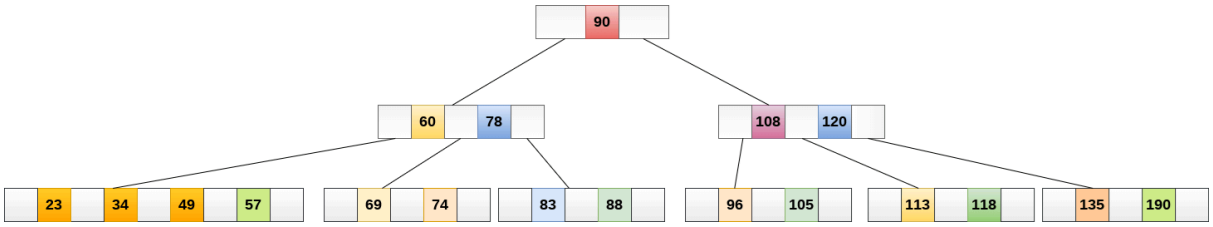


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Let's check the take away from this lecture

1)**Which of the following is true?**

a**) larger the order of B-tree, less frequently the split occurs**

b) larger the order of B-tree, more frequently the split occurs

c) smaller the order of B-tree, more frequently the split occurs

d) smaller the order of B-tree, less frequently the split occurs

Exercise:

Q.1 Explain deletion operation on B tree.

Learning from the lecture:

Learners will be able to understand B Tree and can apply operations on B Tree.

Lecture : 8

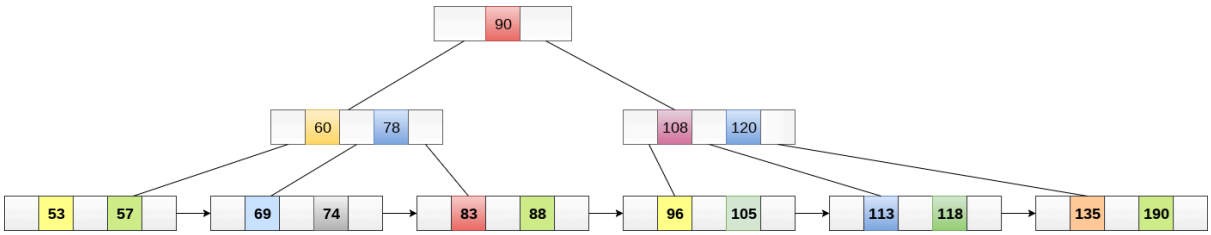
B+ Tree

Learning objective: In this lecture learners will be able to understand B+ tree concepts.

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
The leaf nodes of a B+ tree are linked together in the form of singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which cannot be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree is often called index nodes. A B+ tree of order 3 is shown in the following figure.

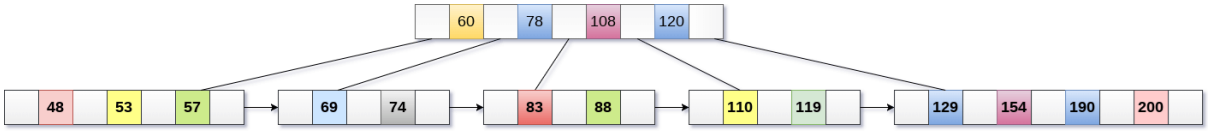


Insertion in B+ Tree

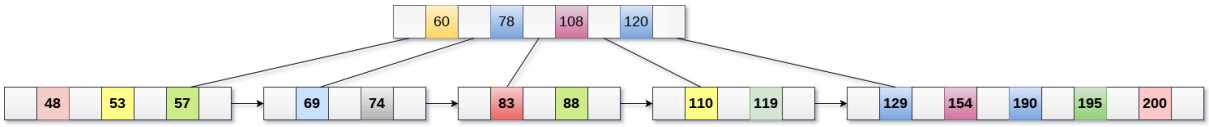
Step 1: Insert the new node as a leaf node
Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

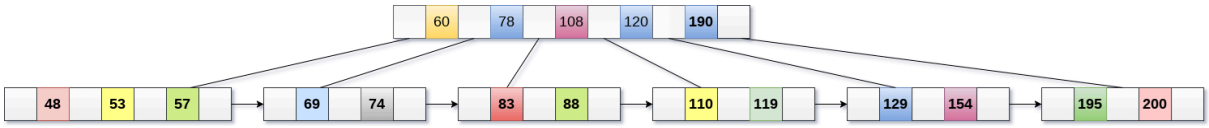
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



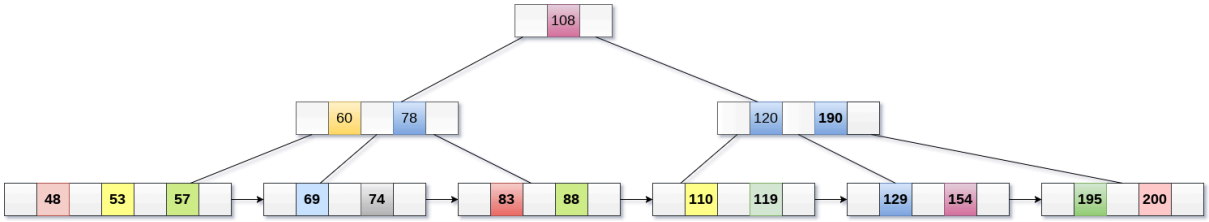
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.

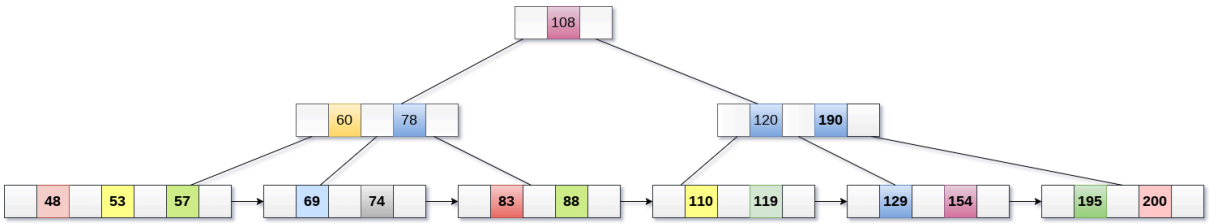


Deletion in B+ Tree

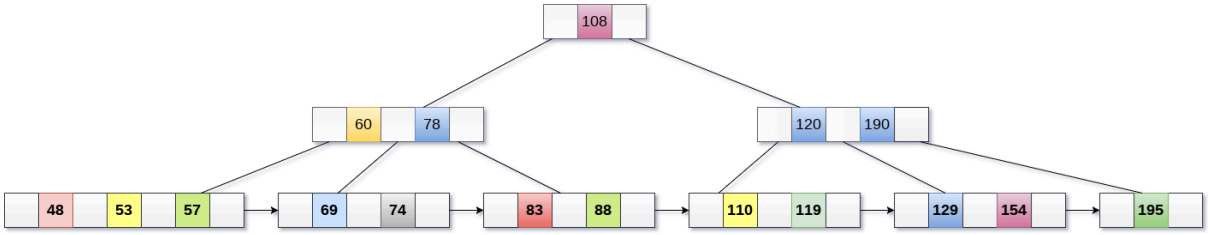
Step 1: Delete the key and data from the leaves.
Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

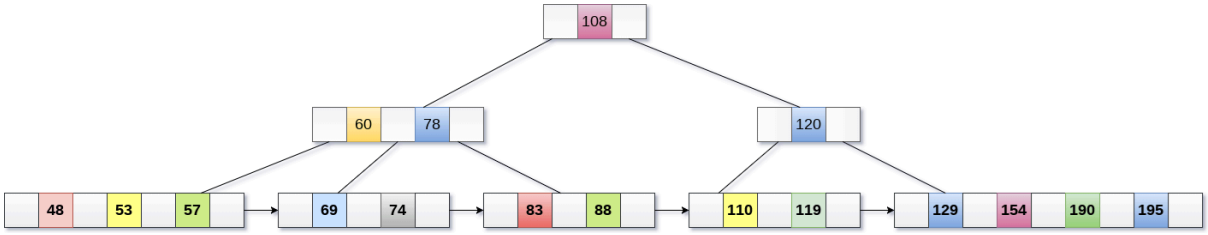
Delete the key 200 from the B+ Tree shown in the following figure.



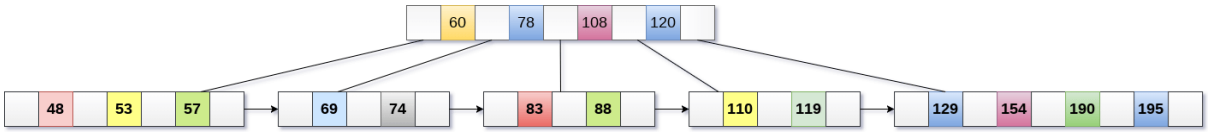
200 is present in the right sub-tree of 190, after 195. delete it.



Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.
Now, the height of B+ tree will be decreased by 1.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

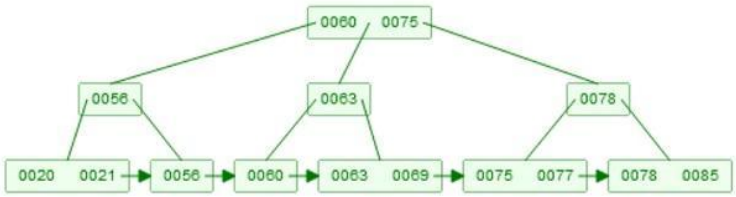
Let's check the take away from this lecture

1)In a B+ tree, both the internal nodes and the leaves have keys.
a) True b) False

2) Which of the following is true?
a) B + tree allows only the rapid random access
b) B + tree allows only the rapid sequential access
c) B + tree allows rapid random access as well as rapid sequential access
d) B + tree allows rapid random access and slower sequential access

Exercise:

Q.1 Explain how to insert 65 into the below B+ tree.



Learning from this lecture: Learners will be able to understand B+ Tree and can apply operations on B+ Tree.

Conclusion

The study of nonlinear data structure gives idea of how data can be stored in hierarchical manner. It shows how we can perform basic operations on them.

Lecture: 9

Representing list as binary tree

Learning objective: In this lecture learners will able to understand how to represent list as binary tree concepts.

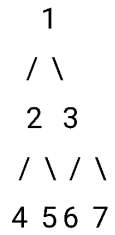
Representing a list as a binary tree involves structuring the list's elements in a way that resembles the structure of a binary tree. In a binary tree, each node can have at most two child nodes: a left child and a right child. By arranging the elements of a list in this hierarchical structure, we can visualize the relationships between the elements more intuitively.

[1, 2, 3, 4, 5, 6, 7]

To represent this list as a binary tree, we'll start by creating nodes for each element of the list and organizing them based on their positions. The root of the tree will be the first element of the list (1), and subsequent elements will be added as children following a left-to-right order.

1. The first element, 1, becomes the root of the tree.
2. The second element, 2, becomes the left child of 1.
3. The third element, 3, becomes the right child of 1.
4. The fourth element, 4, becomes the left child of 2.
5. The fifth element, 5, becomes the right child of 2.
6. The sixth element, 6, becomes the left child of 3.
7. The seventh element, 7, becomes the right child of 3.

Here's how the binary tree representation looks:



In this representation

The root node (1) contains the first element of the list.
The left child of the root (2) contains the second element of the list.
The right child of the root (3) contains the third element of the list.
The left child of node 2 (4) contains the fourth element of the list.
The right child of node 2 (5) contains the fifth element of the list.
The left child of node 3 (6) contains the sixth element of the list.
The right child of node 3 (7) contains the seventh element of the list.
Note that in this binary tree representation, the order of elements in the list is preserved, and the hierarchy of the binary tree reflects the sequential structure of the list. This representation can be useful for certain operations and algorithms that are more naturally applied to trees, such as binary tree traversal algorithms.

Let's check the take away from this lecture

1) **What is the role of the root node in the binary tree representation of a list?**
A) It contains the sum of all elements in the list.
B) It is always the largest element in the list.
C) It contains the first element of the list.
D) It contains the average of all elements in the list.

Exercise:

Q.1 Describe the process of representing a list as a binary tree.

Learning from the lecture:

Learners will be able to understand how to represent lists as a binary tree.

Short Answer Questions:

1. What are the different tree traversing techniques?

Ans) In-order, Pre-Order, Post-order

2. What are Binary search tree operations?

Ans) Search, Insertion, Deletion

3. What is use of threaded binary tree?

Ans) We can reuse empty links by making some threads.

4. Name the self-balancing binary search tree.

Ans) AVL tree

5. Define Leaf Node.

Ans) Anode with no children is called leaf node.

Long Answer Questions:

1. Explain Binary Tree operations.

Ans) Data Structures with C by Seymore Lipschutz

2. What are the different tree traversing techniques?

Ans) Data Structures with C by Seymore Lipschutz

3. Explain threaded binary tree in detail.

Ans) Data Structures with C by Seymore Lipschutz

4. Consider the following list of numbers: 18 25 16 36 8 29 45 12 32 19

   Create a binary search tree using these numbers and display them in non-decreasing order.

Ans) Data Structures with C by Seymore Lipschutz

5. Insert the following elements in AVL tree. 44 17 32 78 50 88 48 62 54

   Explain the rotations used for the same.

Ans) Data Structures with C by Seymore Lipschutz

6. Explain the concept of threaded binary tree. Write a function for inorder traversal of the threaded binary search tree.

Ans) Data Structures with C by Seymore Lipschutz

Set of Questions for FA/CE/IA/ESE

Q. 1) How do you define terms:

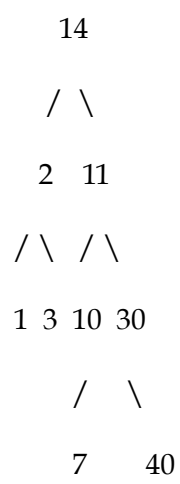
a. Leaf node
b. Subtree
c. Children of a node

d. Degree of a node
e. Edge

Q. 2) What are steps for preorder traversing a binary tree? Explain with example.

Q. 3) Draw a full binary tree with at least 6 nodes.

Q. 4) Here is a small binary tree:



Write the order of the nodes visited in:
A. An in-order traversal:
B. A pre-order traversal:
C. A post-order traversal:

Q. 5) Consider this binary search tree:

Suppose we remove the root, replacing it with something from the left subtree. What will be the new root?

Q. 6) Differentiate between B tree and B+ tree.

Q. 7) Explain B tree with example.

Q. 8) Explain B+ tree with example.

Q. 9) Construct AVL search tree by inserting following elements in order of their occurrences:
6, 5, 4, 3, 2, 1

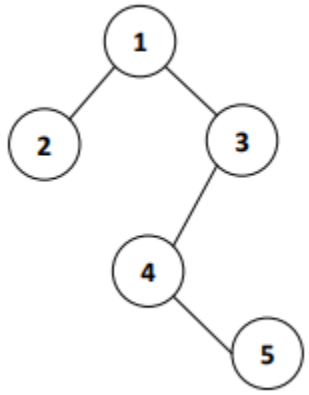
Q.10) Create binary search tree for following:
50, 25, 75, 22, 40, 60, 80, 90, 15, 30

Q. 11) Construct binary tree for the given inorder and preorder traversals.

inorder traversals: Q B K C F A G P E D H R

preorder traversals: G B Q A C K F P D E R H

Q.12) Give traversal order of following tree:



References:

1) Data Structures and Algorithms in Java" by Robert Lafore

Practice for Module-03

Q.1) a) Explain Binary Tree operations. (10 Marks)
b) Define binary tree? List the various properties of binary tree.

Q.2) a) Differentiate between B tree and B+ tree. (10 Marks)

Q.3) a) Explain Binary Tree traversal techniques. (10Marks)

Q. 4) Explain threaded binary tree in detail. (10Marks)

Q. 5) Explain AVL tree in detail. (10Marks)

Q. 6) Explain rotations in binary tree. (10Marks)

Self-assessment

Q.1) Define binary tree. Explain insertion operation on binary tree.

Q. 2) Differentiate between B tree and B+ Tree.

Q. 3) Explain traversal techniques of tree.

Self-evaluation

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		Tick Your choice
1.	Do you understand the various terminologies of tree?	o Yes o No
2.	Do you understand the B tree?	o Yes o No
3.	Do you know the various factors while performing operations on tree?	o Yes o No
4.	Do you understand height balanced tree?	o Yes o No
5.	Do you understand module ?	o Yes, Completely. o Partialy. o No, Not at all.

Module: 05

Graphs

Lecture: 1

Motivation:

Graphs are used to represent, find, analyze, and optimize connections between elements (houses, airports, locations, users, articles, etc.). Graphs are awesome data structures that you use every day through Google Search, Google Maps, GPS, and social media. They are used to represent elements that share connections. The elements in the graph are called Nodes and the connections between them are called Edges.

Syllabus:

Lecture no	Content	Duration (Hr)	Self-Study (Hrs)
1	Graph terminologies	1	1
2	Cycles in graph	1	1
3	Graph representation in memory	1	1
4	Matrix and Adjacency list	1	1
5	Graph traversing techniques: Breadth First Search	1	1
6	Depth First Search	1	1
7	Applications of graph	1	1
8	Graph application: Topological sort	1	1

Learning Objective:

- Learner should know the key concept of Graph.
- Learners shall be able to illustrate different terminologies of graph.
- Learners shall be able to write graph traversal algorithms.

Theoretical Background:

Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.). For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes. Each node can be a structure that contains information like user's id, name, gender, etc.

Key Definitions:

9. A graph is a way of representing relationships that exist between pairs of objects.
10. A graph G is simply a set V of vertices and a collection E of pairs of vertices from V, called edges.
11. A subgraph of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G.

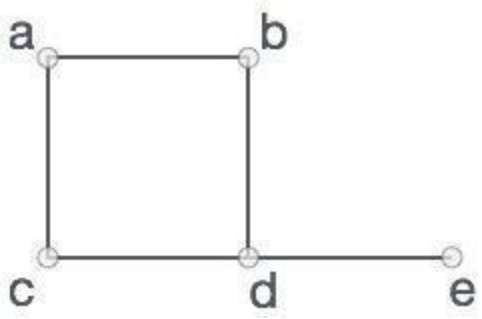
12. A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges.
13. A directed graph (digraph), is a graph whose edges are all directed
14. A weighted graph is a graph that has a numeric label w(e) associated with each edge e, called the weight of edge e

Course Content

Graph terminologies

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph −

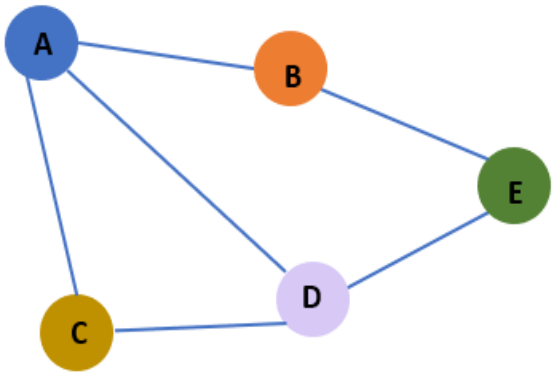


In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Terminologies:



1. Graph Representation: Generally, a graph is represented as a pair of sets (V, E). V is the set of

vertices or nodes. E is the set of Edges. In the above example,

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

E = { AB, AC, AD, BE, CD, DE }

2. Node or Vertex: The elements of a graph are connected through edges.

3. Edges: A path or a line between two vertices in a graph.

4. Adjacent Nodes: Two nodes are called adjacent if they are connected through an edge. In the above example, node A is adjacent to nodes B, C, and D, but not to node E.

5. Path: Path is a sequence of edges between two nodes. It is essentially a traversal starting at one node and ending at another. In the example above, there are multiple paths from node A to node E.

Path(A, E) = { AB, BE }

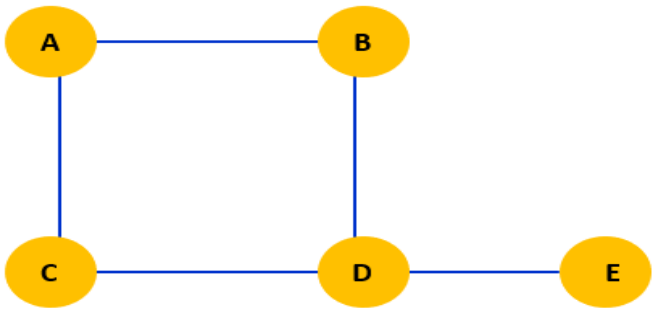
OR

Path(A, E) = { AC, CD, DE }

OR

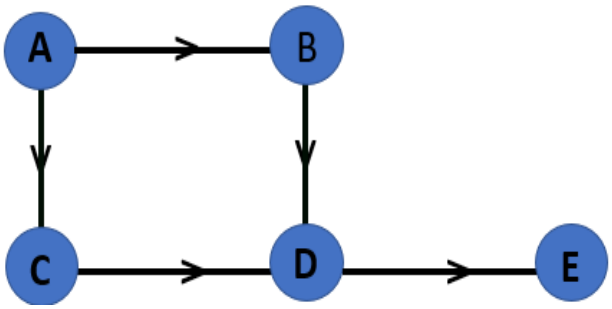
Path(A, E) = { AD, DE }

6. Undirected Graph: An undirected graph is one where the edges do not specify a particular direction. The edges are bi-directional.



Thus, in this example, the edge AC can be traversed from both A to C and C to A. Similar to all the edges. A path from node B to node C would be either { BA, AC } or { BD, DC }.

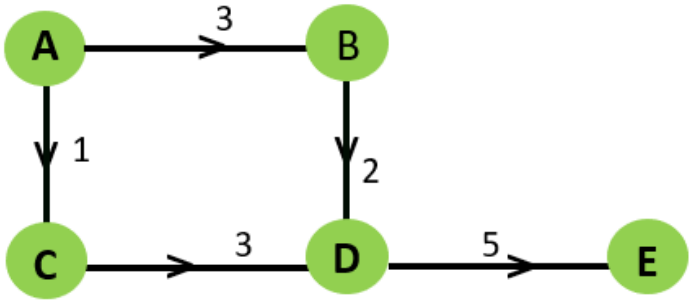
7. Directed Graph: A directed graph is one where the edges can be traversed in a specified direction only.



Thus, in the same example, now the edges are directional. You can only traverse the edge along its direction. There is no path from node B to node C now.

8. Weighted Graph: A weighted graph is one where the edges are associated with a weight. This is generally the cost to traverse the edge.

Example



Thus, in the same example, now the edges have a certain weight associated with them. There are two possible paths between node A to node E.

Path1 = { AB, BD, DE }, Weight1 = 3+2+5 = 10

Path2 = { AC, CD, DE }, Weight2 = 1+3+5 = 9

Clearly, one would prefer Path2 if the goal is to reach node E from node A with minimum cost.

Let's check the take away from this lecture

1) **Which of the following properties does a simple graph not hold?**
   a) Must be connected
   b) Must be unweighted
   c) Must have no loops or multiple edges
   d) Must have no multiple edges
2) **A simple graph in which there exists an edge between every pair of vertices is called**
   a. Complete graph
   b. Euler graph
   c. Planar graph
   d. Regular graph

Exercise
Q.1 What is graph?
Q.2 Explain terminologies of graph with example.

Learning from this lecture: Learners will be able to understand graph terminologies.

Lecture: 2

Lecture: 3

Graph representation in memory

Learning objective:

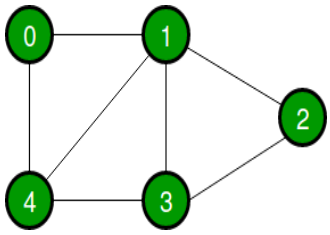
In this lecture learners will able to understand how Graph representation in memory is done.

The following two are the most commonly used representations of a graph.
1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Advantages:

- Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

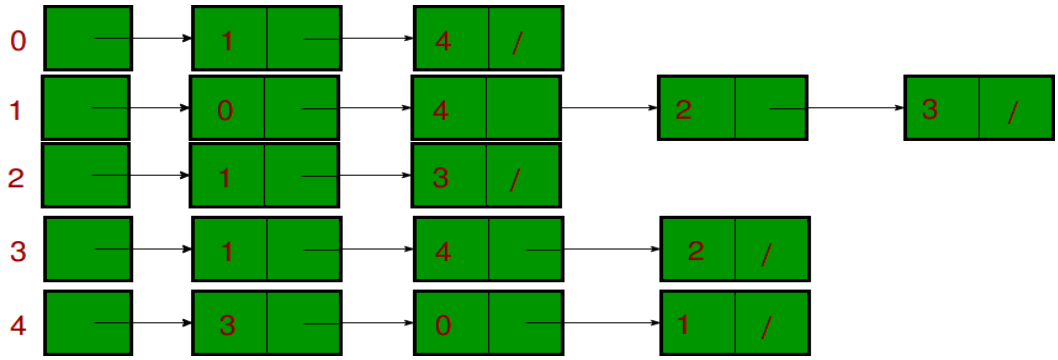
Disadvantages:

- Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Lecture

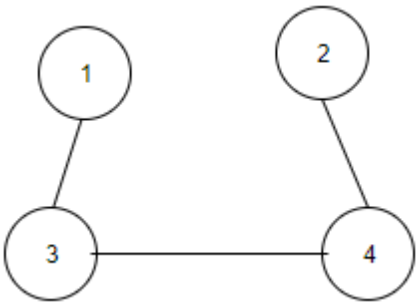
Adjacency List:

- An array of lists is used.
- The size of the array is equal to the number of vertices.
- Let the array be an array [].
- An entry array[i] represents the list of vertices adjacent to the ith vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be represented as lists of pairs.
- Following is the adjacency list representation of the above graph.



Let's check the take away from this lecture

1) **What would be the number of zeros in the adjacency matrix of the given graph?**



a) 10
b) 6
c) 16
d) 0

2) The number of elements in the adjacency matrix of a graph having 7 vertices is _____

a) 7
b) 14
c) 36
d) 49

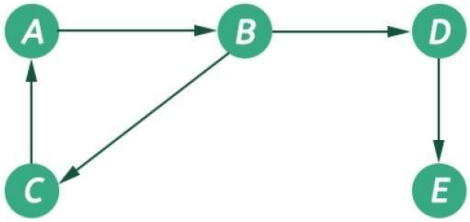
Exercise

Q.1 Illustrate graph representation techniques.

Q.2 Explain Adjacency List representation with suitable example.

Questions/Problems for practice:

Q.3 Given a graph, find its Adjacency List and Adjacency matrix representation.



Learning from this lecture: Learners will be able to apply representation techniques on graph data structure.

Lecture: 4

Graph traversing techniques: Depth First Search

Learning objective: In this lecture learners will able to understand Depth First Search traversing technique in detail.

A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges. **Depth-first search** is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

Traversing mechanism:

- Depth-first search in an undirected graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost.
- We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "visited."
- The vertex s is now our "current" vertex—call our current vertex u.
- We then traverse G by considering an (arbitrary) edge (u,v) incident to the current vertex u
- If the edge (u,v) leads us to an already visited (that is, painted) vertex v, we immediately return to vertex u.
- If, on the other hand, (u, v) leads to an unvisited vertex v, then we unroll our string, and go to v.
- We then paint v as "visited," and make it the current vertex, repeating the computation.
- Eventually, we will get to a "dead-end," that is, a current vertex u such that all the edges incident on u lead to vertices already visited.
- Thus, taking any edge incident on u will cause us to return to u.
- To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to u, going back to a previously visited vertex v.
- We then make v our current vertex and repeat the computation above for any edges incident upon v that we have not looked at before.
- If all of v's incident edges lead to visited vertices, then we again roll up our string and backtrack to the vertex we came from to get to v, and repeat the procedure at that vertex.
- Thus, we continue to backtrack along the path that we have traced so far until we find a

vertex that has yet unexplored edges, take one such edge, and continue the traversal.

- The process terminates when our backtracking leads us back to the start vertex s, and there are no more unexplored edges incident on s.

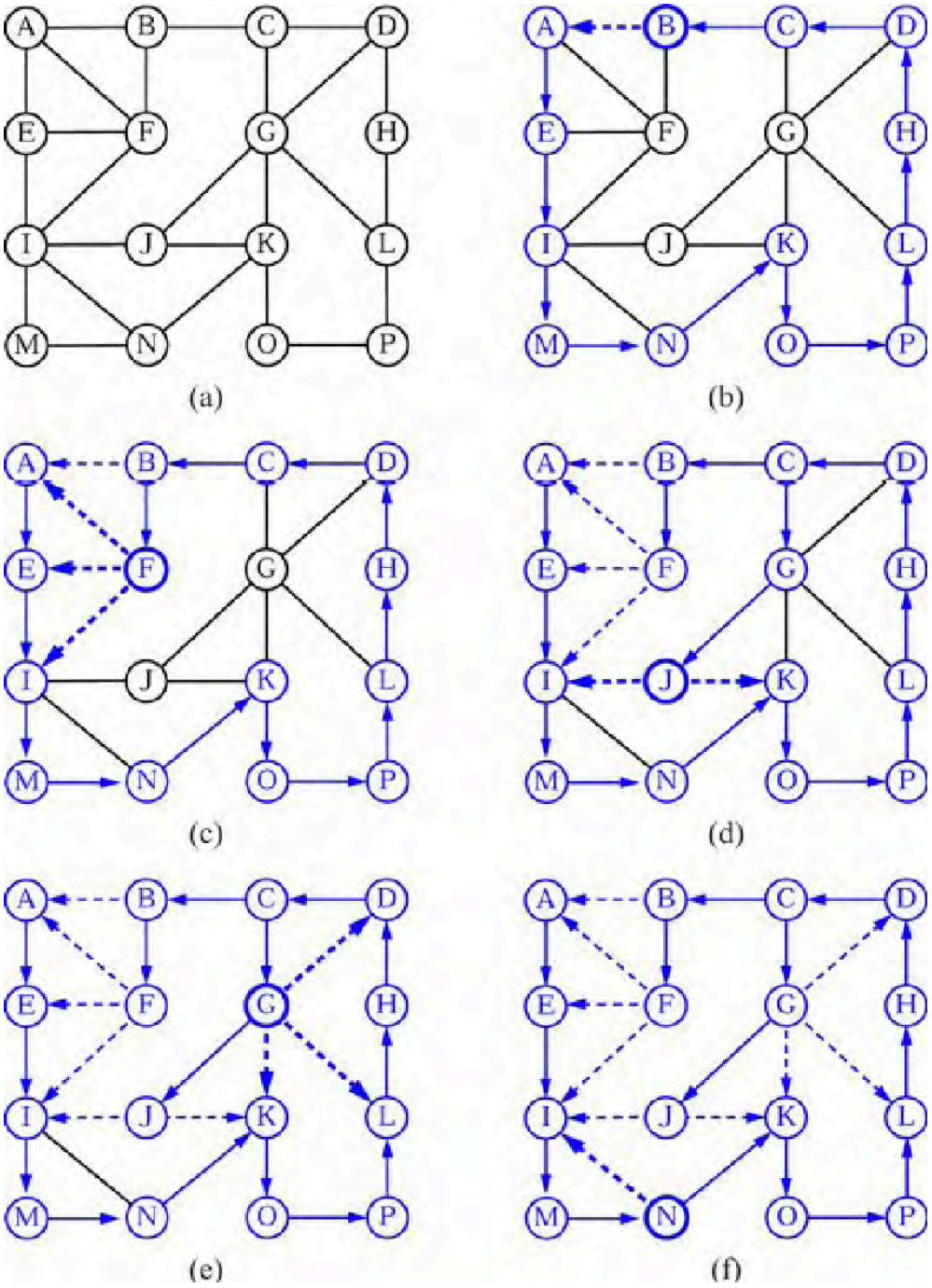


Fig: depth-first search traversal on a graph starting at vertex *A*

Let's check the take away from this lecture

Exercise Q.1 What is a DFS?

Learning from this lecture: Learners will be able to understand basics of DFS.

Lecture: 5

Graph traversing techniques: Depth First Search

Learning objective: In this lecture learners will be able to understand Depth First Search traversing technique in detail.

Discovery Edges and Back Edges

- We can visualize a DFS traversal by orienting the edges along the direction in which they are explored during the traversal, distinguishing the edges used to discover new vertices, called discovery edges, or tree edges, from those that lead to already visited vertices, called back edges.
- In the analogy above, discovery edges are the edges where we unroll our string when we traverse them, and back edges are the edges where we immediately return without unrolling any string.
- As we will see, the discovery edges form a spanning tree of the connected component of the starting vertex s.
- We call the edges not in this tree "back edges" because, assuming that the tree is rooted at the start vertex, each such edge leads back from a vertex in this tree to one of its ancestors in the tree.
- The pseudo-code for a DFS traversal starting at a vertex v follows our analogy with string and paint.
- We use recursion to implement the string analogy, and we assume that we have a mechanism (the paint analogy) to determine if a vertex or edge has been explored or not, and to label the edges as discovery edges or back edges.
- This mechanism will require additional space and may affect the running time of the algorithm.

Code for DFS algorithm

```
Algorithm DFS(G, v):
    Input: A graph G and a vertex v of G
    Output: A labeling of the edges in the connected component of v as discovery
      edges and back edges
  label v as visited
  for all edge e in G.incidentEdges(v) do
    if edge e is unvisited then
      w ← G.opposite(v, e)
      if vertex w is unexplored then
        label e as a discovery edge
        recursively call DFS(G, w)
      else
        label e as a back edge
```

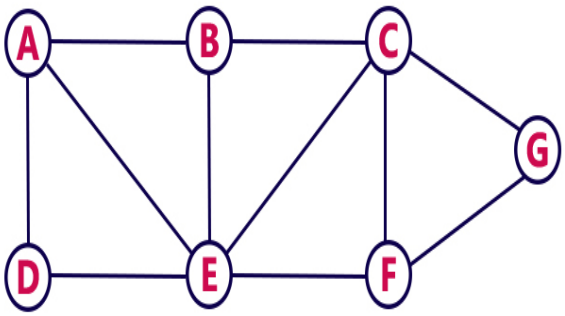
- In terms of its running time, depth-first search is an efficient method for traversing a graph.
- Note that DFS is called exactly once on each vertex, and that every edge is examined exactly twice, once from each of its end vertices.
- Thus, if ns vertices and ms edges are in the connected component of vertex s, a DFS starting at s runs in O(ns + ms) time, provided the following conditions are satisfied:

Let's check the take away from this lecture

1) **Depth First Search is equivalent to which of the traversal in the Binary Trees?**
   a) Pre-order Traversal
   b) Post-order Traversal
   c) Level-order Traversal
   d) In-order Traversal

2) **The Data structure used in standard implementation of Breadth First Search is?**
   a) Stack
   b) Queue
   c) Linked List
   d) Tree

Exercise:
Q.1 Write algorithm of DFS.
Questions/problems for practice:
Q. 2 Perform DFS traversal on following graph.



Learning from this lecture: Learners will be able to apply DFS on graph.

Lecture: 6

Breadth First Search

Learning objective: In this lecture, learners will able to understand BFS.

- Like DFS, BFS traverses a connected component of a graph, and in so doing defines a useful spanning tree.
- BFS is less "adventurous" than DFS, however. Instead of wandering the graph, BFS proceeds in rounds and subdivides the vertices into levels.
- BFS can also be thought of as a traversal using a string and paint, with BFS unrolling the string in a more conservative manner.

Traversing Mechanism:

- BFS starts at vertex s, which is at level 0 and defines the "anchor" for our string.
- In the first round, we let out the string the length of one edge and we visit all the vertices we can reach without unrolling the string any farther.
- In this case, we visit, and paint as "visited," the vertices adjacent to the start vertex s—these vertices are placed into level 1.
- In the second round, we unroll the string the length of two edges and we visit all the new vertices we can reach without unrolling our string any farther.
- These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. T
- he BFS traversal terminates when every vertex has been visited.

Algorithm for a BFS starting at a vertex s:

We use auxiliary space to label edges, mark visited vertices, and store collections associated with levels. That is, the collections L0, L1, L2, and so on, store the vertices that are in level 0, level 1, level 2, and so on. These collections could, for example, be implemented as queues. They also allow BFS to be nonrecursive.

Algorithm BFS(s):

```
initialize collection L_0 to contain vertex s
i ← 0
while L_i is not empty do
  create collection L_{i+1} to initially be empty
  for all vertex v in L_i do
    for all edge e in G.incidentEdges(v) do
      if edge e is unexplored then
        w ← G.opposite(v,e)
        if vertex w is unexplored then
          label e as a discovery edge
          insert w into L_{i+1}
        else
          label e as a cross edge
  i ← i + 1
```

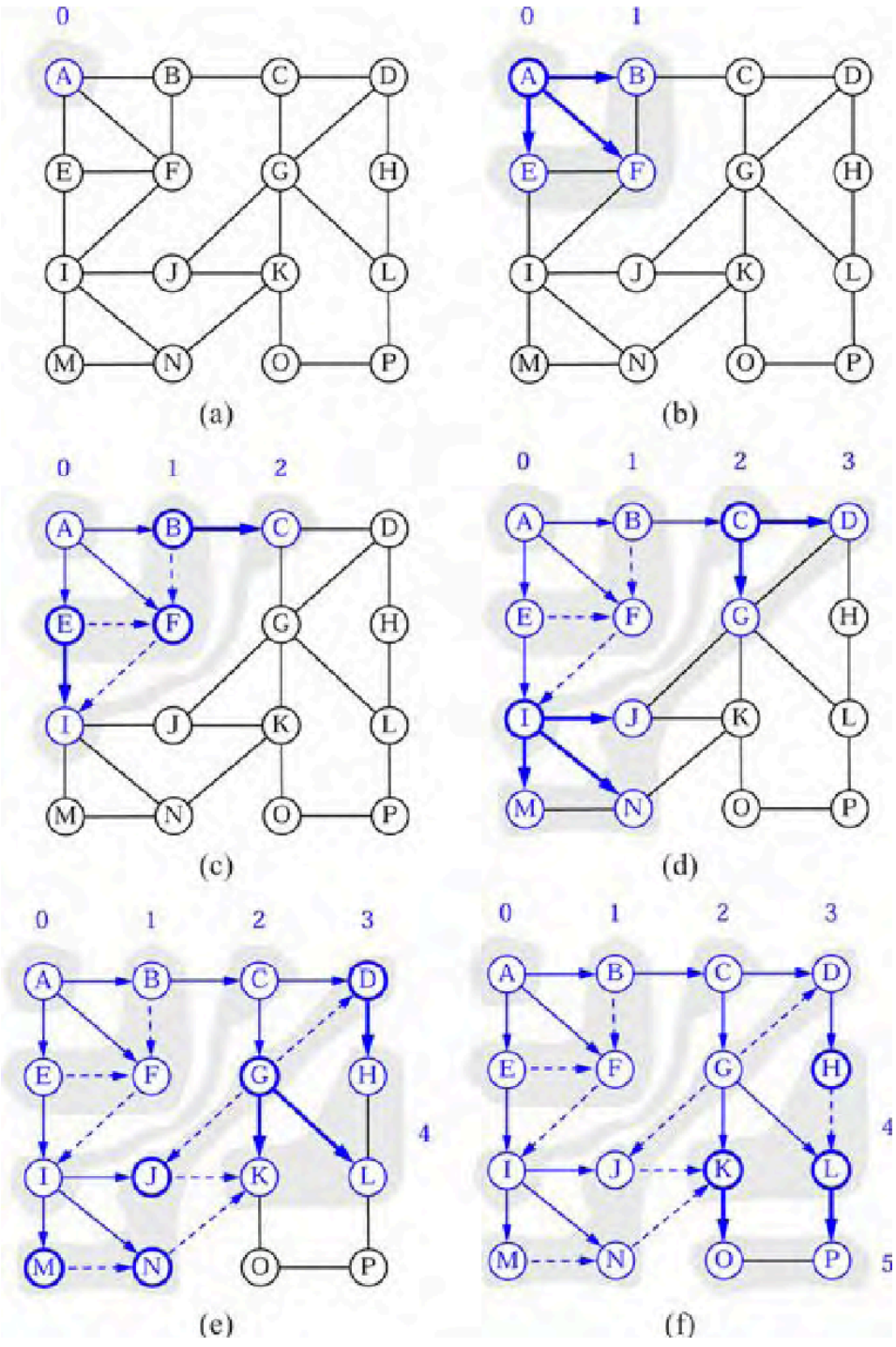


Fig. Breadth-first search traversal

One of the nice properties of the BFS approach is that, in performing the BFS traversal, we can label each vertex by the length of a shortest path (in terms of the number of edges) from the start vertex s. In particular, if vertex v is placed into level i by a BFS starting at vertex s, then the length of a shortest path from s to v is i.

Let's check the take away from this lecture

1) The Data structure used in standard implementation of Breadth First Search is?

a) Stack
b) Queue
c) Linked List
d) Tree

2) The Breadth First Search traversal of a graph will result into?

a) Linked List
b) Tree
c) Graph with back edges
d) Arrays

Exercise:

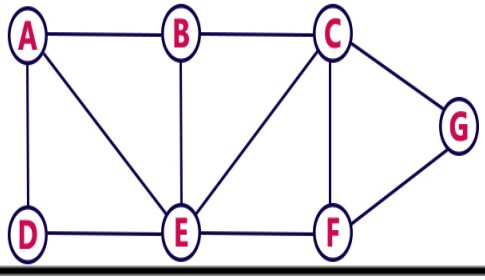
Q.1 What is BFS?

Q.2 Write algorithm of BFS.

Q.3 What is difference between DFS and BFS.

Questions/problems for practice:

Q.3 Perform BFS traversal on following graph.



Learning from the lecture: Learners will able to apply BFS on graph.

Lecture: 7

Applications of graph

Learning objective: In this lecture learners will able to know different applications of graph.

A graph is a non-linear data structure, which consists of vertices(or nodes) connected by edges(or arcs) where edges may be directed or undirected.

- In **Computer science** graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.

- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to another page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
- Graphs are used to represent the routes between the cities. With the help of tree that is a type of graph, we can create hierarchical ordered information such as **family tree**.
- Graph theory is also used in sociology. For example, to explore rumor spreading, or to measure actors' prestige notably through the use of social network analysis software.
- In computer network, the relationships among interconnected computers within the network, follow the principles of graph theory.
- In computer science graph theory is used for the study of algorithms like:
    - Dijkstra's Algorithm
    - Prims's Algorithm
    - Kruskal's Algorithm

Let's check the take away from this lecture

Exercise: Explain different applications of graph.

Learning from the lecture: In this lecture learners will be able to applications of graph.

Conclusion

The study of nonlinear data structure graph gives idea of how data can be stored in hierarchical manner. It shows how we can perform basic operations on them.

Lecture: 8

Applications of graph

Learning objective: In this lecture learners will able to know graph application Topological sort.

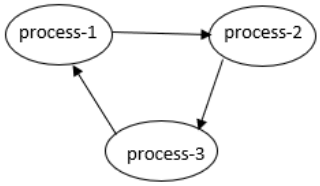
Topological Sort of a directed graph is a linear ordering of its vertices such that for every directed edge u->v, vertex u comes before v in the ordering. In the Topological sort, a process can start when it has 0 prerequisites. In this lecture , we will cover various Applications of Topological Sort in depth.

The Applications of Topological Sort are:

Finding cycle in a graph
Operation System deadlock detection
Critical Path Analysis
Course Schedule problem
Other applications like manufacturing workflows, data serialization and context-free grammar.

1. Finding cycle in a graph

A topological ordering is possible only for directed acyclic graph(DAG). For a cyclic graph topological ordering is not possible.



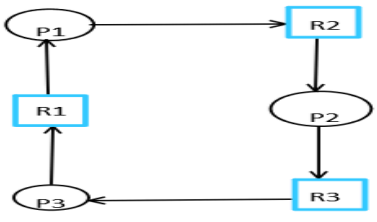
iq.opengenus.org

In the figure above (process-1 → process-2), process-2 can be started only when process-1 is already finished. We can say that process-2 depends on process-1, process-3 on process-2, and process-1 on process-3.

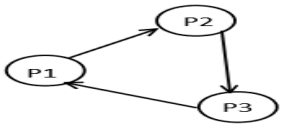
If the given graph contains a cycle, then there is at least one vertex will break topological order. If topological sort isn't defined then we can say that the graph is cyclic.

2. Operation System deadlock detection

Deadlock is a state in which a process in a waiting state and another waiting process is holding the demanded resource.



Resource allocation graph



Corresponding wait for graph

iq.opengenus.org

Here, process P1 holds resource R1, process P2 holds resource R2, process P3 holds resource R3, and process P1 is waiting for R2, process P2 is waiting for R3 and process P3 waiting for R1, then process P1, process P2 and process P3 will be in a deadlock. Topological sorting is used here to identify a cycle. If the wait-for graph has a cycle, then there is deadlock.

3. Critical Path Analysis: Critical path analysis is a project management technique. It is used to find the minimum time a project can take and the dependencies of each activity on others. The completion of the project requires the completion of some activities. An activity may have some predecessor activities. It is mandatory to finish the predecessor activities to start a new activity. The critical path calculates the longest path of the activity graph. Activities represent vertices, and edges represent the preceding relationship between them. Activities are listed in topological order.

The critical path algorithm is also used to find the Hamiltonian path in a DAG. The Hamiltonian path is a path in a graph(undirected or directed graph) that visits each vertex exactly once. If a Hamiltonian path exists, the topological order is unique.

Let's check the take away from this lecture

Exercise: Explain different topological sort applications of graph.

Learning from the lecture: In this lecture learners will be able to know graph application Topological sort.

Short Answer Questions:

6. What is graph?

Ans) A graph is a way of representing relationships that exist between pairs of objects.

7. What is directed graph?

Ans) It is a graph whose edges are all directed

8. What is weighted graph?

Ans) It is the one where the edges are associated with a weight.

Long Answer Questions:

7. Illustrate graph representation techniques.

Ans) Data Structures with C by Seymore Lipschutz

8. Write algorithm of DFS.

Ans) Data Structures with C by Seymore Lipschutz

9. Explain with example how DFS works.

Ans) Data Structures with C by Seymore Lipschutz

10. Write algorithm of BFS.

Ans) Data Structures with C by Seymore Lipschutz

11. Explain with example how DFS works.

Ans) Data Structures with C by Seymore Lipschutz

12. Explain difference between DFS and BFS.

Ans) Data Structures with C by Seymore Lipschutz

13. Explain different applications of graph.

Ans) Data Structures with C by Seymore Lipschutz

Set of Questions for FA/CE/IA/ESE

Q. 1) Explain depth first search algorithm in detail.

Q. 2) Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights.

Q. 3) Write an algorithm for (a) Depth first search (b) breadth first search

Q. 4) Explain graph representation with example.

Q. 5) Write a function for DFS traversal of graph. Explain its working with an example.

References:

1) Data Structures with C by Seymore Lipschutz

Practice for Module-05

Q.1) Explain graph and its terminologies. (10 Marks)
Q. 2) Explain representation techniques of graph. (10 Marks)
Q.3) Write algorithm of BFS and DFS. (10 Marks)
Q.4) Explain difference between DFS and BFS. (10 Marks)

Self-assessment

Q.1) Draw the directed graph that corresponds to this adjacency matrix:

	0	1	2	3
0	true	false	true	false
1	true	false	false	false
2	false	false	false	true
3	true	false	true	false

Q. 2) Draw a directed graph with five vertices and seven edges. Exactly one of the edges should be a loop, and do not have any multiple edges.

Self-evaluation

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		Tick Your choice
6.	Do you understand the various graph terminologies?	o Yes o No
7.	Do you understand the Graph representation in memory?	o Yes o No
8.	Do you know the BFS traversal?	o Yes o No
9.	Do you know the DFS traversal?	o Yes o No
10.	Do you understand module ?	o Yes, Completely. o Partialy.

		o No, Not at all.

Module:06

Application of Data Structure

Lecture: 1

Motivation:

Data structures help manage and analyze large datasets in fields such as data science, bioinformatics, and finance. They enable efficient data storage, indexing, and querying, supporting complex operations like data filtering, aggregation, and statistical analysis

In program-specific applications, data structures optimize memory usage and processing time, improving program performance through efficient storage, retrieval, and manipulation of data.

In domain-specific applications, data structures tailor the organization and representation of data to specific problem domains, facilitating effective problem-solving, modeling, and analysis.

Syllabus:

Lecture no	Content	Duration (Hr)	Self-Study (Hrs)
1	Program Specific Applications	1	1
2	Domain Specific Applications	1	1

Learning Objective:

- Learner should know the key concept of Application of Data Structure.
- Learners Should analyze program specific usage of data structure.
- Learners Should analyze domain specific usage of data structure.

Theoretical Background:

Data structures focuses on optimizing memory usage and processing time, enabling efficient storage, retrieval, and manipulation of data. In domain-specific applications, the theoretical background emphasizes tailoring data structures to specific problem domains, facilitating effective problem-solving, modeling, and analysis within those domains.

Key Definitions:

In program-specific applications, the application of data structures involves selecting and implementing efficient data organization schemes to optimize memory usage, improve data access and manipulation, and enhance overall program performance.

In domain-specific applications, the application of data structures involves designing and utilizing data structures that align with the specific problem domain, enabling effective representation, processing, and analysis of domain-specific data and facilitating problem-solving within that domain.

Program-specific applications focus on the efficient management of data within a particular software system or application, leveraging data structures to enable faster searching, sorting, filtering, and other data operations.

Domain-specific applications utilize data structures tailored to the unique characteristics and requirements of a specific problem domain, ensuring that data is organized and represented in a manner that aligns with the specific needs and constraints of that domain.

Course Content

Program Specific application of Data structure

Program-specific applications of data structures involve using data structures to optimize the management and manipulation of data within a particular software system or application. Here are a few examples:

Database Management Systems: Data structures such as B-trees, hash indexes, and heaps are utilized to efficiently store and retrieve data in database management systems. These data structures help optimize data access, searching, and sorting operations, ensuring fast and scalable data processing.

Text Editors: Data structures like linked lists or ropes are used to represent and manipulate text in text editor applications. These data structures allow for efficient insertion, deletion, and editing of text, ensuring responsive and user-friendly text editing experiences.

Compiler Design: Data structures such as symbol tables, syntax trees, and control flow graphs are essential in compiler design. They enable efficient storage, analysis, and transformation of source code during the compilation process, facilitating the generation of optimized machine code.

Image Processing: Data structures like matrices or multidimensional arrays are employed in image processing applications. These data structures enable the representation and manipulation of image data, facilitating operations such as filtering, transformation, and compression.

Graph Algorithms: Data structures like adjacency lists or adjacency matrices are utilized in graph algorithms, such as shortest path algorithms or network analysis. These data structures allow efficient representation and traversal of graph data, enabling the analysis and optimization of relationships and connections in various applications, including social networks and transportation systems.

Networking Applications: Data structures like graphs or adjacency matrices are utilized in networking applications for tasks such as routing and network analysis. These data structures allow efficient representation and traversal of network data, enabling the analysis and optimization of connections and paths in various network topologies.

File Systems: Data structures such as file allocation tables (FAT), linked lists, or B-trees are used in file systems to organize and manage file data efficiently. These data structures help in locating and accessing files quickly, supporting operations like file creation, deletion, and retrieval.

Artificial Intelligence and Machine Learning: Data structures like matrices, vectors, or tensors are extensively used in AI and machine learning applications. These data structures facilitate the storage and manipulation of large datasets, enabling efficient computations for tasks such as training and inference in machine learning models.

In each program-specific application, data structures are chosen and implemented based on the specific requirements of the software system or application. They are utilized to optimize data storage, retrieval, and manipulation, leading to improved performance and efficiency in handling program-specific data.

Overall, in program-specific applications, data structures are employed to enhance data management, processing, and analysis, leading to improved efficiency and performance in specific software systems or applications.

Let's check the take away from this lecture

1)Which data structure is commonly used to efficiently store and retrieve data in database management systems?
A) Linked Lists
B) Heaps
C) B-trees
D) Hash Tables
(Correct Answer: C)

2)What is the primary purpose of using data structures like matrices, vectors, or tensors in artificial intelligence and machine learning applications?
A) To store and manage variables and identifiers
B) To represent hierarchical structures of source code
C) To efficiently manipulate and store large datasets
D) To facilitate image processing operations
(Correct Answer: C)

Exercise

Q.1 Explain how data structures like symbol tables, syntax trees, and control flow graphs are utilized in compiler design.

Q.2 Describe the role of data structures in image processing applications.

Learning from this lecture: Learners will be able to analyze domain specific application of data structure.

Lecture: 2

Domain Specific application of Data structure

Learning objective:

In this lecture learners will be able to analyze domain specific application of data structure.

Domain specific application of data structure:

Domain-specific applications of data structures involve using specialized data structures that are tailored to the unique characteristics and requirements of a specific problem domain. Here are a few examples of domain-specific applications and the corresponding data structures used:

Social Networks: Social networks require efficient representation and analysis of relationships between individuals or entities. Graph data structures, such as adjacency lists or adjacency matrices, are commonly used to model and analyze social network connections, enabling tasks such as finding friends, identifying communities, or recommending connections.

Geographic Information Systems (GIS): GIS applications deal with spatial data and require efficient storage and retrieval of geographic information. Spatial data structures, including spatial indexes (e.g., R-trees, quadtrees), help organize and query spatial data efficiently, supporting tasks like spatial queries, routing, and map visualization.

Bioinformatics: Bioinformatics involves analyzing and managing biological data, such as DNA sequences or protein structures. Data structures like suffix trees, which efficiently store and search for patterns in strings, are used in DNA sequence analysis. Additionally, graph-based data structures, like de Bruijn graphs, are employed for genome assembly and analysis.

Financial Systems: Financial systems handle large volumes of data related to transactions, market data, and customer information. Data structures like balanced binary search trees (e.g., AVL trees, red-black trees) or hash maps are commonly used to store and retrieve financial data efficiently. These structures enable fast searching, sorting, and aggregation of data for tasks such as portfolio management, risk analysis, or algorithmic trading.

Compiler Optimizations: Compiler optimizations involve transforming source code to improve the efficiency and performance of compiled programs. Data structures like control flow graphs, symbol tables, and abstract syntax trees are employed to represent and analyze program structures. These structures enable optimizations such as dead code elimination, loop unrolling, or register allocation.

Natural Language Processing (NLP): NLP applications involve processing and analyzing human language data. Data structures like trie, suffix tree, or n-gram models are employed to efficiently represent and search for words, phrases, or patterns in text. These structures enable tasks such as text search, language modeling, sentiment analysis, or information retrieval.

Supply Chain Management: Supply chain management involves tracking and optimizing the flow of goods and services. Data structures like queues, stacks, or linked lists are used to manage orders, shipments, or inventory efficiently. These structures facilitate tasks such as order processing, inventory management, or scheduling in supply chain systems.

Genetic Algorithms and Evolutionary Computing: Genetic algorithms and evolutionary computing involve solving optimization problems inspired by natural evolution. Data structures like population pools, chromosomes, or genetic trees are employed to represent and manipulate candidate solutions. These structures enable tasks such as selection, crossover, mutation, and fitness evaluation in evolutionary algorithms.

In each domain-specific application, the choice of data structure is driven by the specific requirements and characteristics of the problem domain. Domain-specific data structures enable efficient representation, storage, and analysis of data, facilitating effective problem-solving and specialized operations within their respective domains.

Let's check the take away from this lecture

1) Which data structure is commonly used in supply chain management systems to efficiently manage orders, shipments, or inventory?
A) Linked Lists
B) Stacks

C) Queues
D) Tree
(Correct Answer: C)

Exercise

Q.1 Describe the role of suffix trees in natural language processing (NLP) applications.

Learning from this lecture: Learners will be able to apply traversal concept on tree data structure.