Introduction to JAVA

Objectives

1. Important Features of Java
2. JVM
3. JDK
4. Object-oriented programming
5. Variables
6. Data types
7. Types of variables
8. Type casting
9. Operators
10. Summary
11. List of references
12. Bibliography
13. Questions

1.1 Important Features of Java

1. Simple

Java language is a **simple** programming language because there is no need to remove unreferenced objects because there is an Automatic Garbage Collection in **Java**. Java code is easy to read and write.

2. Platform Independent

Platform independent language means once compiled you can execute the program on any **platform** (**OS**). **Java is platform independent**

3. Architectural Neutral

Java compiler produces an architecture-neutral object file format, which makes the compiled code executable on many processors.

4. Portable

Java is called portable because you can compile a java code which will send out a byte-code, and then you run that code with Java Virtual Machine.

5. Multithreaded

Multithreading is a Java feature that allows synchronised execution of two or more parts of a program for maximum utilization of CPU.

6. Distributed

Java is distributed because it facilitates users to create spread applications in Java. Like EJB and RMI is used to create applications.

7. Networked

It is mainly designed for web based applications, J2EE is used for developing network based applications.

8. Robust

It uses strong memory management. There is a lack of pointers that avoids security problems and there are exception handling and the type checking mechanism in Java.

9. Dynamic

The process of allocating the memory space to the input of the program at a run-time is known as dynamic memory allocation

10. Secure

It is a more secure language compared to other language. In this language, all code is covered in byte code after compilation which is not readable by human.

11. High performance

It have high performance because of following reasons;

- This language **uses Bytecode**
- **Garbage collector**, collect the unused memory space
- It has **no pointers** so it makes java language simple to understand
- It **support multithreading**,

12. Interpreted

A Java interpreter is a software that translates the Java virtual machine and runs Java applications

13. Object Oriented

It has all OOP features such as abstraction, encapsulation, inheritance and polymorphism.

1.2 JVM (Java Virtual Machine) Architecture

What is JVM??

Java Virtual Machine that provides IDE for java. It converts Java byte code into machines language.

What it does

Java code is compiled into byte code. This byte code gets interpreted on different machines Between host system and Java source, Byte code is an intermediary language JVM is responsible for allocating memory space.

JVM Architecture

1) ClassLoader

The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization

2) Method Area

JVM Method Area stores class structures like metadata.

3) Heap

All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

4) JVM language Stacks

A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

5) PC Registers

PC register store the address of the Java virtual machine instruction which is currently executing.

6) Native Method Stacks

Native method stacks hold the instruction of native code depends on the native library.

7) Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:**
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc

1.3 JDK

JDK: Java Development Kit

The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets.

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.

1.4 Object Oriented Programming (OOPs) Concept in Java

Object-oriented programming: . Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Class
- Object
- Method

1.5 Variables

A variable is used to hold data within your Java program. A variable represents a location in computer's memory. You can store data into this location and retrieve data out of it. Every variable has two parts, a name and a data type. In short a variable is a named memory location that is used to store value.

Variables are the basic units of storage in a Java program. They are declared to store values. They must be declared before use in the program. Their declaration must begin with data type and is followed by variable name and a semicolon. The data type can be a primitive data type or a class.

Variable Names

A variable name should be chosen in a meaningful way so as to reflect its purpose in the program. Variable name should be short. A variable can also be called as an identifier.

A Few important rules to be followed while naming variables are

- A variable is any combination of letters, number, underscore and $ sign.
- A variable name may not be a java Keyword, or reserved word in Java.
- A variable name is case sensitive.
- A variable name can not contain any blank spaces in between.

A few valid java variables are:

Total

total

cName

student_name

_min

A few invalid java variables are:

6^{th}_grade

area&volume

max-val

The syntax to declare a variable as follows:

data-type variable1, variable2, variable3,….;

here variable1, variable2, variable3 are the variable names, and these name must be separated by by commas. The declaration statement must be terminated by a semi-colon.

Some valid variables declaration are:

int count;

char name[20];

float total;

Initializing a variable:

int sum=0; // initization at the time of declaration

or

int sum;

…

…

…

sum=0;

one thing must be remembered is a variable must be declared, before it is initialized.

Defalut Initial Value for Primitive Data Types

Variable

Data Type	**Defalut Initial Value**
Char	Null

Byte	0
Short	0
Int	0
Long	0L
Float	0.0f
Double	0.0d
Boolean	False

If the variable name consists of more than one word, the first letter of first word should be small and first letter of each subsequent word should be capital.

Ex: studentName, totalMarks etc

Assigning Value to a Variable

Values can be assigned to variables by using the assignment operator (=). There are two ways to assign value to variables. These are as follows:

Declares three integer variables a, b, and c

int a,b,c;

Declares three integer variables and initializing a and c

int a = 10, b, c = 100;

Declares a byte variable flag and initialize its value to 50

byte b = 50;

Declares the character variable c with value 'c'

char c = 'c';

Stores value 10 in num1 and num2
int num1 = num2 = 10;

1.6 Data Types

When you define a variable in Java, you must inform the compiler what kind of a variable it is. That is, whether it will be expected to store an integer, a character, or some other kind of data. This information tells the compiler how much space to allocate in the memory depending on the data type of a variable.

Primitive Data Types

The Java programming language provides eight primitive data types to store data in Java programs. A primitive data type, also called built-in data types, stores a single value at a time, such as a number or a character.

The primitive data types are predefined in the Java language and are identified as reserved words.
The primitive data types that are broadly grouped into four groups.

Integer,Float,Character,Boolean

Type	Size in bytes	Min Value	Max Value
byte	1	-128	127
Short	2	- 32,768	32,767
Int	4	-2,147,483,648	2,147,483,647
Long	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Floating-point Types

Type	Size in bytes	Min Value	Max Value
float	4	3.4e-038	1.7e+038
double	8	3.4e-038	1.7e+038

Java has two floating point data are float and double. These are also called real numbers, as they represent numbers with fractional precision.

Character Type : To store character in memory, java has char data type.

Boolean Type
In some senarios we need to store 0 or 1.

Example program

```
public class EmployeeData {

  public static void main(String[] args) {

        int empNumber;
        float sal;
```

```
        double shareBal = 564790.975;
        char gender = 'f';
        Boolean ownVehicle = false;
        empNumber = 2012;
        sal = 65729.50f;
        System.out.println("Employee Number: " +empNumber);
        System.out.println("Salary: " + sal);
        System.out.println("Gender: " + gender);
        System.out.println("Share Balance: " +shareBal);
        System.out.println("Owns vehicle: " +ownVehicle);
    }
}
```

Here, variables of type int, float, char, double, and boolean are declared. A float value needs to have the letter f appended at its end. Otherwise, by default, all the decimal values are treated as double in Java.

1.7 Types of variables

In Java, there are three types of variables:

1. Local Variables
2. Instance Variables
3. Static Variables

1) Local Variables

Local Variables are a variable that are declared inside the body of a method.

2) Instance Variables

Instance variables are defined without the STATIC keyword .They are defined Outside a method declaration. They are Object specific and are known as instance variables.

3) Static Variables

Static variables are initialized only once, at the start of the program execution. These variables should be initialized first, before the initialization of any instance variables.

```
class Abc {
        int i = 125; //instance variable
        static int sVar = 1; //static variable
        void method() {
                int b = 225; //local variable
        }
}
```

1.8 Type Conversion & Type Casting

A variable of one type can receive the value of another type. Here there are 2 cases -

Type conversion : Variable of smaller capacity is be assigned to another variable of bigger capacity.

```
double d;
int i= 200
d =i;
```

Here we are assigning int value into double type variable. This known as automatic conversion.

Type Casting : Variable of larger capacity is be assigned to another variable of smaller capacity

```
double d = 10;

int i;

i = (int) d;
```

In this case, we have to explicitly specify the type cast operator. This is also known as *Type Casting*.

In case, we do not specify a type cast operator; the compiler gives an error. Since this rule is enforced by the compiler.

Example:

```
import java.util.Date;

class CoversionEx {

        public static void main(String args[]) {

                byte b;

                int i = 120;

                double d = 278.278;

                System.out.println("int converted to byte");

                b = (byte) i;

                System.out.println("i and b " + i + " " + b);

                System.out.println("double converted to int");

                i = (int) d;

                System.out.println("d and i " + d + " " + i);

                System.out.println("\ndouble converted to byte");

                b = (byte)b;

                System.out.println("d and b " + d + " " + b);
```

```
    }

}
```

Output:

```
int converted to byte
i and b 120 120
double converted to int
d and i 278.278 278

double converted to byte
d and b 278.278 120
```

1.9 **OPERATORS IN JAVA**

Java has several operators. We have already used several operators such as =,+,-and *

The types of operators in java.

1. Arithmetic Operator,
2. Relational Operator,
3. Logical Operator,
4. Bitwise Operator
5. Ternary Operator
6. Assignment Operator.
7. Unary Operator,

1) Java Arithmetic Operators

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

Java Arithmetic Operator Example

```
class ArOpEx{
    public static void main(String args[]){
        int a=20;
        int b=10;
        System.out.println(a+b);
        System.out.println(a-b);
        System.out.println(a*b);
        System.out.println(a/b);
```

```
            System.out.println(a%b);
        }
}
```

Output:

```
30
10
200
2
0
```

Java Arithmetic Operator Example: Expression

```
class OpExpEx{
        public static void main(String args[]){
            System.out.println(20*10/6+4-1*4/3);
        }
}
```

Output:

```
36
```

2) Relational Operator

```
    class RelationalOpEx{

        public static void main(String args[])    {

         int a=10,b=20;

         System.out.println(a>b);    //true

         System.out.println(a<=b);   // true

        System.out.println(a>=b);   // true

       System.out.println(a==b);   // false
```

```
        System.out.println(a!=b);   // true
        System.out.println('E'==69); // true    ascii value of'E'is  69
        }
    }
```

Output:

```
false
true
false
false
true
true
```

3) Logical Operator

```
class LogicalOperatorEx {
  public static void main(String args[])   {
    int a,b;
    a=10;
    b=20;
    System.out.println(true  && true); // true
    System.out.println(true  && false);   // false
    System.out.println(false && true);     //false
    System.out.println(false && false);    // false
    System.out.println(true  || true);
    System.out.println(true  || false);   // true
    System.out.println(false || true);    // true
```

```
    System.out.println(false || false);    // false
    System.out.println(!true);             // false
    System.out.println(!false);            // true
    System.out.println(a>b || a<b);     // true  (true || false)
    System.out.println(a>b && a<b);     // false  (true && false)
    //System.out.println(!a>b);          //error
    System.out.println( !(a==b) );       // true   !(false)
    }
}
```

Output:

```
true
false
false
false
true
true
true
false
false
true
true
false
true
```

4) Bitwise Operator

The bitwise & operator always checks both conditions whether first condition is true or false.

```
class logOpEx{
```

```
        public static void main(String args[]){
                int a=20;
                int b=10;
                int c=30;
                System.out.println(a<b&&a<c);//false && true = false
                System.out.println(a<b&a<c);//false & true = false
        }
}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample{
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);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
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
```

5) Ternary Operator

```
class ConditionalOpEx {

  public static void main(String args[])    {

    int i = 10,j = 20;

    System.out.println((i==j)?true:false);   //false

    System.out.println((i<j)?true:false);   //true

    System.out.println(true?1:2);           //1

    System.out.println(false?1:2);          //2

  }

}
```

Output:

false
true
1
2

6) Assignment Operator

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

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

Java Assignment Operator Example

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

Output:

```
13
9
18
9
```

7) Unary Operator,

These are used to increment/decrement a value by 1. Generally to increment a value in the variable i by 1 we use the statement like i=i+1;
The same statement can be written like x+=1;

Example:

- i++ is valid
- 10++ is not valid
- i++++ is not valid
- i++ + i++ is valid

1.10 SUMMARY :

In this unit, we learn Introduction of Java, Features of Java, Java virtual Machine and Java Environment, concept of dtata types, variable and constants with example, operators.

1.11 LIST OF REFERENCES

1. Java 2: The Complete Reference, Fifth Edition, Herbert Schildt, Tata McGraw Hill.

2. Programming in JAVA2,Dr. K. Somasundaram, JAICO Publishing House

1.12 BIBLIOGRAPHY

http://www.javabeginner.com/

https://www.tutorialspoint.com/

https://www.geeksforgeeks.org/

https://www.javatpoint.com/java-tutorial

https://guru99.com

1.12 QUESTIONS:

1. Give Advantages and Disadvantages of Java.
2. Explain features of java.
3. What is JVM and JDK
4. Explain types of Operators in java with suitable example.
5. Explain types of Data types in java with example.
6. Give Advantages and Disadvantages of switch.

Control Structures

Objectives

1. Introduction
2. Decession making and branching
3. If statement
4. If…else statement
5. Else if ladder
6. Switch .. case statement
7. The ? : operator
8. Decision making and looping
   - The while statement
   - Do … while
   - For
   - For each
9. Jump Statements
   - Break
   - Continue
10. Summary
11. List of references
12. Bibliography
13. Questions

2.1 Introduction

Up to this we have discussed about computer science and JAVA, You have created programs which used sequential execution. A program is a set of statements, which normally executed line by line in the order they appear. This approach is fine when conditional evaluation or repetition of certain statements are not necessary.

However, in real time there are number of scenarios where we may have to write code based on a certain criteria or repeat a set of statements until a specific condition is met.

For example you may need to write a program for assigning grades to the students according to the following conditions

Percentage and the respective grades given below

>90% - Grade O,
>=74% and <90% - Grade A
>=61% and <74% - Grade B
>=51% and <61% - Grade C
>=35% and <51% - Grade D
<35% - Grade F

2.2 Decision making and branching

The selection statement of java are used for branching and decision making within a program. As we have seen, this required a kind of decision making to check whether a particular condition is met or not and then execute certain set of statements accordingly.

When a program jumps to another portion of a program based on the condition is known as conditional branching. If branching takes place without condition is called as unconditional branching. JAVA supports both types of branching or decision making statements.

These statements are also known as decision making statements.

- if
- switch .. case
- conditional operator

2.3 If Conditional statement

A few examples of decision making using if statement are

```
if(you are bored)
        Go for picnic

if(error in program)
        Debug the program


if (number is not divisible by 2)
        number is odd
else
        number is even
```

the if statement can have other forms for different conditions depending upon the complexity of the conditions to be tested

a. if statements
b. if-else statements
c. Nested if … else
d. else-if Ladder

Simple if Statement

Syntax :

```
if(condition or test expression)
{
        statement True Block;
}
Statement - false
```

If the test condition or expression is true, the statements in the true block will get executed, and the program control is passed to statement –false. if the test condition or expression is false, the statements in the statement-true block will be ignored and the program control is passed to the statement – false.

Let us consider a program for simple if

```
import java.util.Scanner;
public class SimpleIf {
        public static void main(String[] args) {
                char ch;
                int posnum;
                Scanner sc = new Scanner(System.in);
                System.out.println("Please enter an alphabet :");
                ch=sc.next().charAt(0);
                if (ch >='A'&& ch<='Z'||ch>='a'&&ch<='z' ) {
                        System.out.println("Alphabet"+ch);
                }
                System.out.println("Program completes");
        }
}
```

Output:

```
Please enter an alphabet : v
Alphabetv
Program completes
```

In the above program the if checks whether a given character is an alphabet or not.

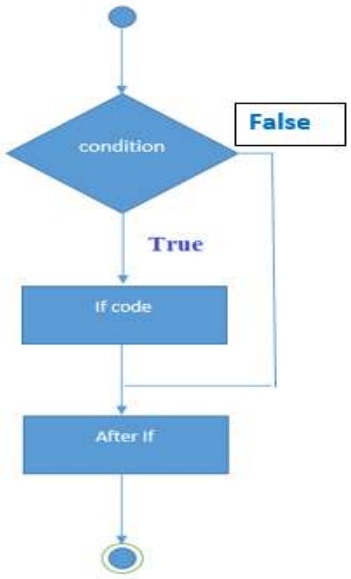
2.4 If.. else Statement

The if..else statement, if the specified condition in the if the statement is false, then the statement after the else keyword (that can be a block) will execute.

The general form of if else is as follows :

```
If(condition or test expression)
{
      True Block statements
}
else
{
      flase block statements
}
Statement- n
```

If the condition or test expression is true, then the true block statements will be executed and flase block will not be executed. If the condition or test expression is false, then the false block statements will be executed. In either case either true block or false block will get executed, but not both.Once , one of these blocks completes their execution the program control is transferred to statement-n.



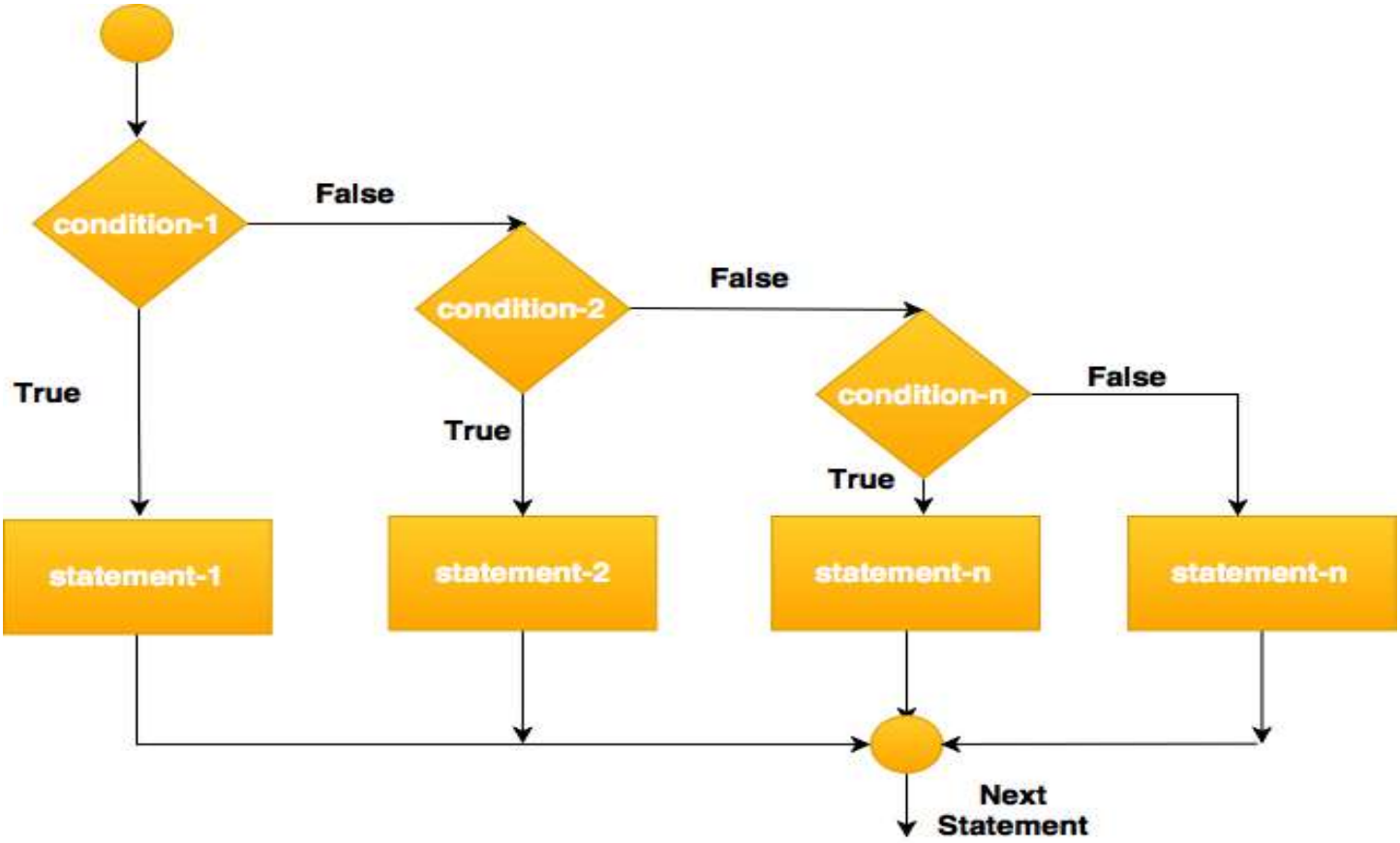
```
import java.util.Scanner;
public class ExIfElse {
	public static void main(String[] args) {
		int uAge;
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter Your Age: ");
		uAge = sc.nextInt();
		if (uAge >= 18)  {
			System.out.println("You are Major ");
		}
		else  {
			System.out.println("You are Minor");
		}
	}
  }
```

This program accepts User age from user and checks whether major or minor. If the test expression result is true ie., uAge>18, then if block will get executed. If the test expression result is flase ie., uAge>18, then else block will get executed.

Output:

```
Enter Your Age: 25
You are Major
```

Nested if .. else statements:



When a series of decisions are involved, we can use more than one if … else statement in a nested form: The logic of the above nested if .. else is as follows: if test expression_1 results into true, then program control is passed to inner if statement and evaluates test expression_2, depending upon the truth value of the test expression _1 returns false, the program control is passed to else part of the outer if and statement _3 will get executed. Once, the if block (outer if or inner if) completes the execution, the program control will be transferred to statement_x

An example program using nested if … else

```
import java.util.Scanner;
public class NestedIfElseEx {
        public static void main(String[] args) {
                int age;
                Scanner inputDevice = new Scanner(System.in);
                System.out.print("Please enter Age: ");
                age = inputDevice.nextInt();
                if (age >= 18 && age <= 35)  {
                System.out.println("between 18-35 ");
                }
                else if (age > 35 && age <= 60)  {
                System.out.println("between 36-60");
                }
                else  {
```

```
                        System.out.println("not matched");
                    }
                }
            }
```

2.5 else if ladder

Another way of putting ifs together is using a chain of ifs, in which the statement associated with each else is again an if. This logical structure is ifs is commonly referred as else if ladder and is used when multi-path decisions are involved. Else if ladder takes the following form

If (test expression-1)
Statement-1;
Else if (test expression-2)
Statement-2;
Else if(test expression-3)
Statement-3;
Else
Default statement;
Statement-x;

In else if ladder, the conditions are evaluated from the top of the ladder to downwards. As soon as the true condition is found, the statement associated with it is executed and the program control is passed on to the statement-x. when all the conditions becomes false, then the final else containing the default statement will get executed, and then the program control is transferred to statement-x.

An example program using else if ladder

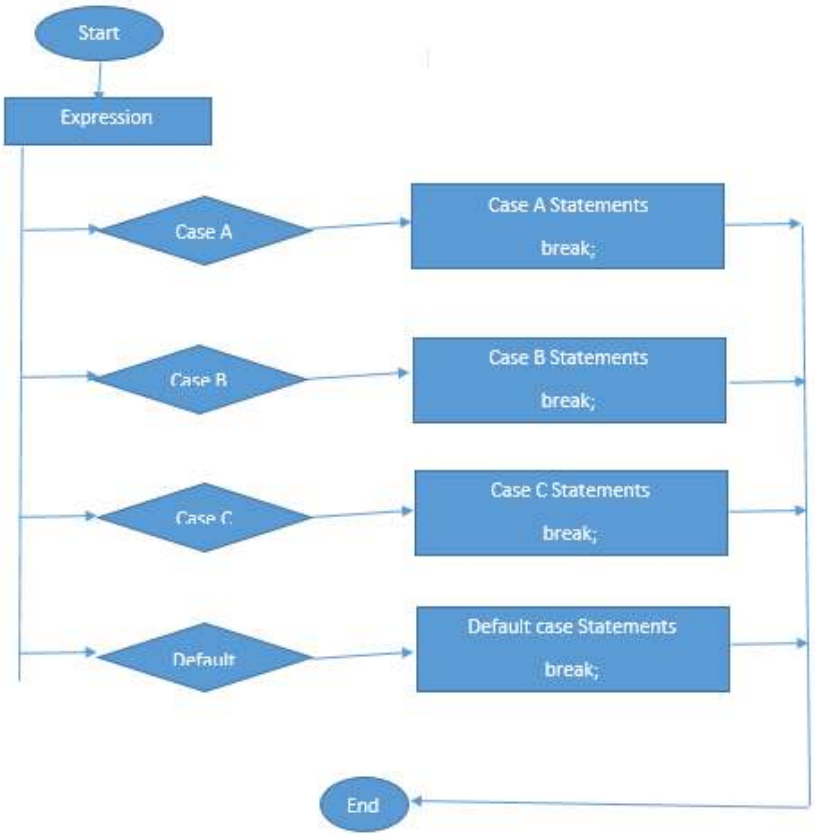
```
public class ElseIfLadderEx {
   public static void main(String args[]) {
      int per = 80;

      if( per >= 75 && per<=100 ) {
         System.out.print("Grade O");
      }else if( per >= 60 ) {
         System.out.print("Grade A");
      }else if( per >= 50 ) {
         System.out.print("Grade B");
      }else if( per >= 35 ) {
         System.out.print("Grade C");
      }else{
         System.out.print("Fail");
      }
   }
}
```

Output:

2.6 switch … case statement:

C supports a multi-way decision statement known as switch … case. The switch statement tests the value of a given variable (or expression) against a set of case values and when a match is found, the block of statements associated with that case are executed. The general form of switch… case statement is as follows:

```
Switch (expression)
{
        Case val-1:
                Stsement-block-1;
                Break;
        Case val-2:
                Stsement-block-2;
                Break;
        Case val-3:
                Stsement-block-3;
                Break;
        ……
        default :
                default -block;
                break;
}
Statement-x;
```



Here the expression associated with switch is an integer or character expression. The values (value-1, value-2 …)associated with case are known as case labels. Each of these labels must be unique within a switch statement. Each case label must end with a colon (:).

An example program using switch.. case

```
import java.util.Scanner;
public class ExSwitch {
        public static void main(String[] args) {
                int day;
                Scanner sc = new Scanner(System.in);
                System.out.print("Please enter number(1 for  weekday 2 for weekend): ");
                day = sc.nextInt();
                switch (day) {
                        case 1:
                                System.out.println("Weekday");
                                break;
                        case 2:
                                System.out.println(“weekend");
                                break;
```

```
                default:
                        System.out.println("not matched");
                        break;
        }
       }
      }
```

2.7 the ? operator

Java language has a special operator, useful for making two-way decisions. This operator is a combination of ? and : and takes three operands. This operator is popularly known as conditional operator or ternary operator. The general form of conditional operator is as follows:

Conditional expression ? expression-1 : expression-2

The conditional expression is evaluated first. If the result is true, expression-1 is evaluated and returned as the value of the conditional expression. Otherwise expression-2 is evaluated and its value is returned. In general conditional operator is used as substitute for simple for simple if … else statement.

Consider the following code

```
int a =10;
int b =20;
int c;
if (a>b)
        C=a;
else
        C=b;
```

This if else this code segment can be written as follows using ternary operator

```
C=(a>b) ? a : b
```

```
public class CondEx {
  public static void main(String args[]) {
    int a, b, c;
    a = 30;
    b = 55;
    c = (a > b) ? a: b;
    System.out.println("Biggest Value is: " + c);
  }
```

}

Output:

```
Biggest Value is: 55
```

2.8 Loop Statements: -

In some cases we need to execute some set of statements, until condition is satisfied. This is called iteration. loop statements will be executed the same set of instructions, until a termination condition is met.

Following are the java Loops

a. while
b. for
c. do-while
d. for-each

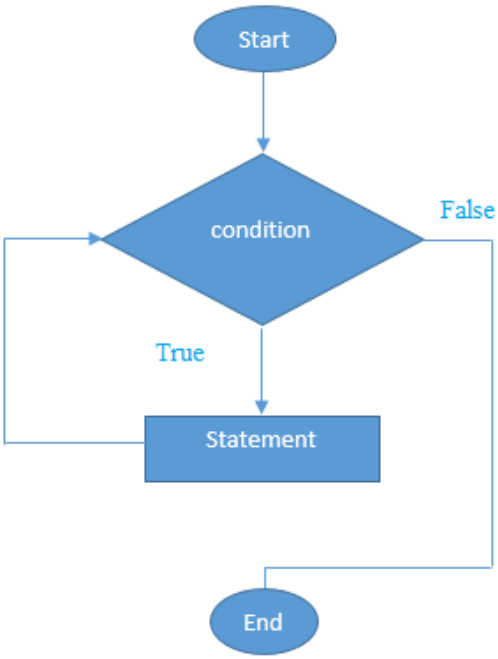
2.8.1 while : -

The simplest of all the looping controls is in java is while statement. The general form of while statement is as follows:

Syntax:

```
Initialization of loop control variable;
while (condition)
{
    Body of the loop

}
```

Flowchart –



If the condition is true the set of statements in the block executes. The condition returns a Boolean value. When the condition is 0 or false, control goes out of the while loop

An example program using while statement

```
public class LoopWhile {
    public static void main(String[] args) {
        int i = 0;
        while (i < 5) {
            System.out.println("Value :: " + i);
            i++;
        }
    }
}
```

Output: -

```
Value :: 0
Value :: 1
Value :: 2
Value :: 3
Value :: 4
```

2.8.2 The do-while loop: -

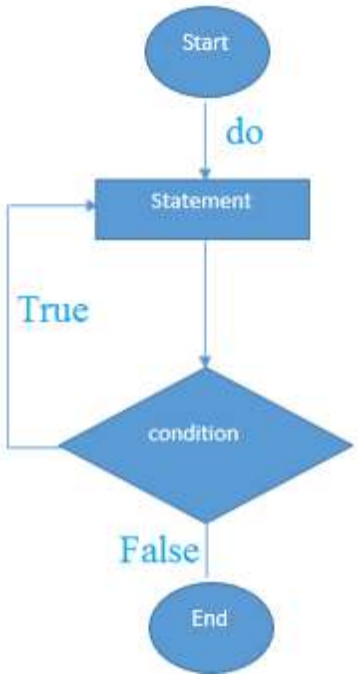
This loop also resembles while loop, but the difference is that do-while evaluates its condition at the end of the loop, instead of the first. The do-while loop executes at least once, then it will check the condition prior to the next iteration.

Do … while is an exit-controlled looping structure, i.e., the loop body will be executed first before evaluating loop control statement. Do … whitle takes the follwong form:

Syntax:

```
Initialization of loop control variable;
do
{
    Body of the loop;

}while(condition);
```

- Flowchart



Example: -

```
public class ExDoWhile {

    public static void main(String[] args) {

        int i = 0;
        do {
            System.out.println("value :: " + i);
            i++;
        } while (i < 5);
    }
}
```

Output: -

```
Value :: 0
Value :: 1
Value :: 2
Value :: 3
Value :: 4
```

2.8.3 The for loop: -

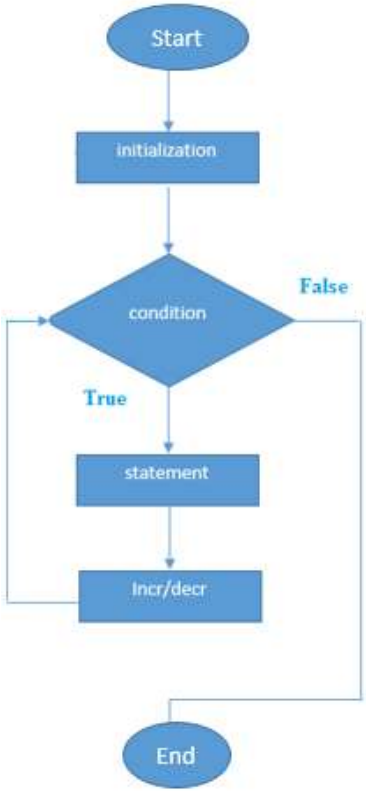
For loop works similar to other loops, but it is simpler one. Initialization, condition, decrement or increment or loop runner will be written in one line.

Syntax:

```
for ( initialization ;  condition;  increment or decrement statement)

{

    Loop body

}
```

The for loop starts by initializing the loop variable. The condition will be evaluated for every iteration before the block statements executes. If the condition is true then only the statement (that is usually a block) will execute. The increment or decrement statement executes every time, after every iteration.

- Flowchart-



The execution of the for statement is as follows:
Initialization of the loop control variable is done first, using assignment statements such as i = 1 or count =10 etc. here I and count are referred as loop control variables.

Example: -

```
public class ExWhile {
    public static void main(String[] args) {
            int i;
            for (i=0;i < 5;i++) {
                    System.out.println("Value :: " + i);
            }
       }
}
```

Output: -

```
Value :: 0
Value :: 1
Value :: 2
Value :: 3
Value :: 4
```

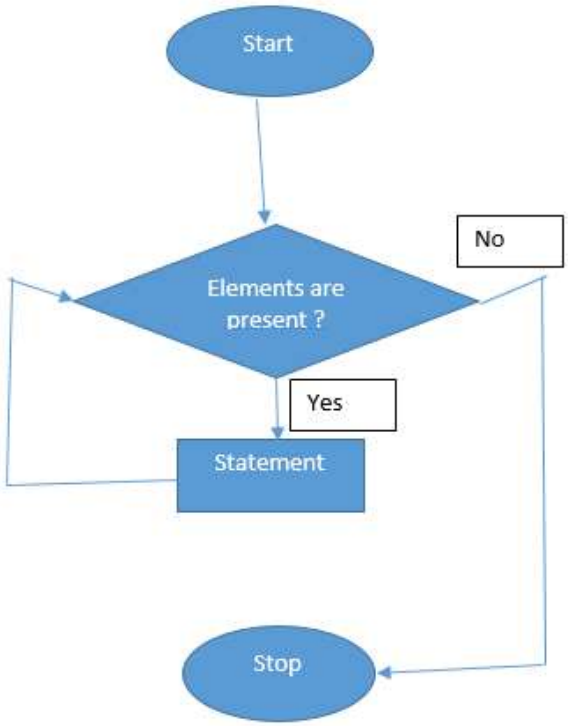
2.8.4 For each loop: - (Enhanced for loop)

This was introduced in Java 5. This loop is basically used to traverse the array or collection elements. In the most recent version, it is also called as lambda expression.

Syntax:

```
for( Type Identifier : expression)
{
//statements;
}
```

- Flowchart –



Example: -

```
    public class ForEachEx {
        public static void main(String[] args) {
            int[] i = {  1, 2, 3, 4, 5 };
            for (int j: i) {
                System.out.println("value :: " + j);
            }
     }

}
```

Output: -

value : : 1

value : : 2

value : : 3

value : : 4

value : : 5

2.9 Jump Statements: -

Jump statements are used to unconditionally transfer the program control to another part of the program.

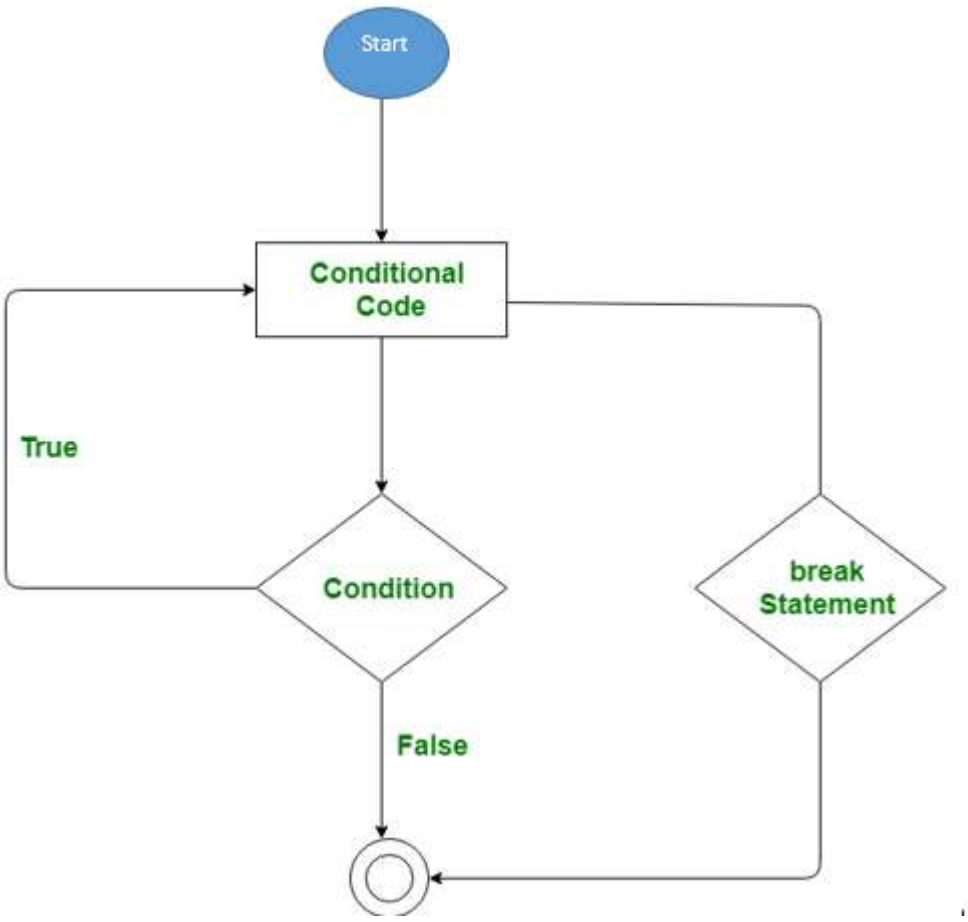
Java provides the following jump statements:

a. break statement
b. continue statement

2.9.1 Break Statement: -

The break statement immediately quits the current iteration and goes to the first statement, following the loop. Another form of break is used in the switch statement.

- Flowchart –



Example: -

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

```
for (int i=1; i<=10; i++) {
if(i==5) {
        break;
   }
     System.out.println(i);
  }
}
}
```

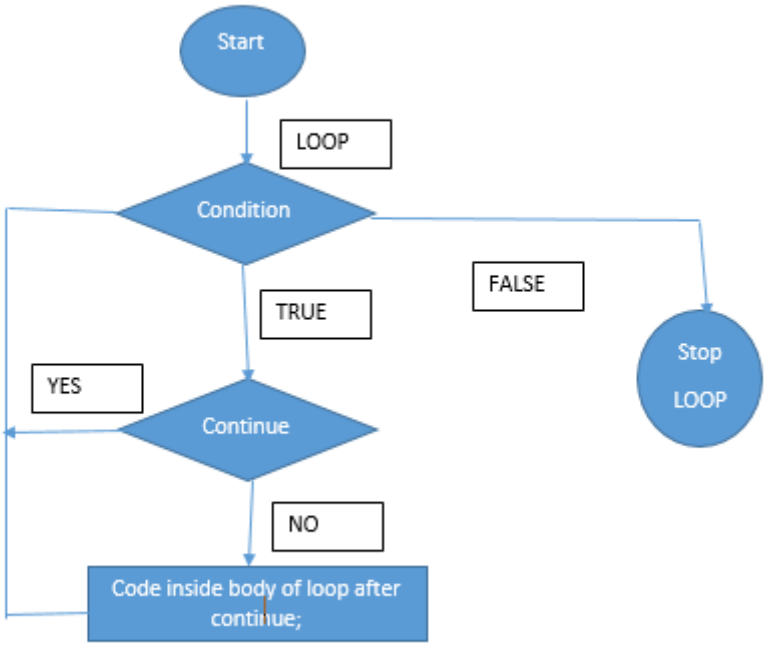
Output: -

```
1
2
3
4
```

2.9.2 Continue Statement: -

The continue statement is used when you want to continue running the loop, with the next iteration, and want to skip the rest of the statements of the body, for the current iteration. It is written by writing continue keyword.

- Flowchart-



Example: -

```
public class ContinueExample {
public static void main (String [] args) {
for (int i=1; i<=10; i++) {
if(i==5) {
      //using continue statement
       continue;//it will skip the rest statement
  }
      System.out.println(i);
  }
}
}
```

Output: -

```
1
2
3
4
```

2.10 SUMMARY:

In this unit, we learned Selection Statement, Iteration Statement and Jump in Statement.In all if statement and switch statement, for loop, while loop, do-while loop with examples. In Jump Statements, we have covered break, continue and label with example

2.11 LIST OF REFERENCES

1. Java 2: The Complete Reference, Fifth Edition, Herbert Schildt, Tata McGraw Hill.

2. Programming in JAVA2,Dr. K. Somasundaram, JAICO Publishing House

2.12 BIBLIOGRAPHY

http://www.javabeginner.com/

https://www.tutorialspoint.com/

https://www.geeksforgeeks.org/

https://www.javatpoint.com/java-tutorial

https://guru99.com

2.13 QUESTIONS:

1. Give Advantages and Disadvantages of do while loop.
2. Explain types of control structures with example.
3. Explain types of loops in java with example.
4. Give Advantages and Disadvantages of switch.

Classes

Objectives

1. Introduction
2. Constructors
3. Method overloading
4. Finalizer methods
5. Summary
6. List of references
7. Bibliography
8. Questions

3.1 Introduction

Classes are building blocks for Java programming. A class is a collection of objects of similar type. Once a class is defined, You can create any number of objects of that class.

Defining class

A class defines the data types used and the methods that manipulate them. The variables declared in a class are called instance variables. A class can contain any number of methods, which are used to manipulate the data and carry out a specific task. The general form of declaring a class is:

```
Class  classname extends superclassname {

Type -variable-1;

Type –variable-2;

Type –variable-N;

Return-type methodname1(para_list1) {

Body1;

}

Return_type methodname2(para_list2) {

Body2;

}

Return_type methodnameN (para_listN) {
```

Body N;

}

}

Class name is a user-defined class name. the name can be coined using the same rules to form a variable. Extends is a java key reserved word and is optional. Superclassname is the name of the parent class from which this class is derived. The instance-variable1 and the other two are instance variables of the type type. Method name 1 is the name of the method, which, when executed, will return a data of return-type. Body1, body2, bodyN are java statements. Parameterlist1, 2, N are the list of parameters passed to the methods methodname1, methodname2 and methodnameN as arguments.

An example of a simple class is given below:

```
class Emp {
  String eName;
  int eAge;
}
```

In the above example, Emp is the name of the class. By convention, all class names start with uppercase followed by lowercase and numbers. The subsequent words will start with uppercase letters. eName and eAge are instance variables.

The new operator and objects

A java object is one instance of a class. An object can be created from a class using the new operator. The new operator creates an object and returns an object reference. The object reference can be stored in a variable. Variables that store object references are called object reference variable. The type of the object reference variable is of the class type. Consider the following statement:

Emp oe = new Emp();

The variable oe is the object reference variable whose type is Emp. The new operator creates an object of type rectangle and returns the object reference, which is stored in oe.

A class, except in a few cases, cannot be used as such. Only instances of the class can be used. The object reference variable rec contains a reference pointing to the object rec and does not contain any data pertaining to the object. When you copy one object reference variable to another object reference variable, no object is created, but another copy of the reference variable to

another object reference variable, no object is created, but another copy of the reference is created, which again points to the same object. The following statement creates another copy of the object reference variable rec:

Emp oe1 = oe;

When oe is printed, only the object reference 3B75 is printed and not the content of the object, which again is in contrast to the procedure-oriented languages.

Instance of a class.

The dot (.) operator

The dot operator (.) is used to access the instance variables and methods defined in that object. The general form of using the dot operator is:

The following program illustrates the use of dot operator:

```
class Emp {
        String eName;
        int eAge;
}
public class Two {
   public static void main(String[] args) {
        Emp oe= new Emp();
        oe.eName="Yashwanth";
        oe.eAge=20;
        System.out.println("Name is :" + oe.eName );
        System.out.println("age is :" + oe.eAge );


   }
}
```

Method declaration and calling

Method is declared inside a class. Each method performs a specific task. The data associated with the instance variable is manipulated in the methods. The methods may directly output the result or return a value to the calling section.

The general form of declaring a method is:

Return_type methodname (parameterlist){

Body;

}

Here methodname is the name of the method and is coined by using the rules to form a variable name, return-type is the type returned by the method and parameterlist is the list of formal parameters passed a s arguments to the method and body is the java statements.

A method can be called only on its object. However, methods declared as static can be called directly independent of an object. The general form of calling a method is:

Object.methodname (parameter-list);

Program

```
class Emp {
  String eName;
  int eAge;
  void details() {
            System.out.println("Name is :" + eName );
            System.out.println("age is :" + eAge );
  }
}
public class Two {
  public static void main(String[] args) {

    Emp oe= new Emp();
    oe.details();
```

```
  }
}
```

Output:

```
Name is :null
age is :0
```

On initializing the instance variable we can get values.

```
class Emp {
  String eName;
  int eAge;
  void details() {
            System.out.println("Name is :" + eName );
            System.out.println("age is :" + eAge );
  }
}
public class Two {
  public static void main(String[] args) {

    Emp oe= new Emp();
    oe.eName="Yashwanth";
    oe.eAge=20;
    oe.details();
  }
}
```

Output:

```
Name is :Yashwanth
age is :20
```

3.2 Constructors

A constructor can initial values to an object when it is created. In many cases, instance variables needs to be initialized. In the previous program, we have seen that we have initialized instance variables. Using constructors are special methods. The name of the constructor shall be the same as the name of the class itself. There can be any number of constructors in a class. The general form of declaring a constructor is:

```
Class calssname {
Type instance variable;
Classname(parameter-list1) {//constructor
Body1;
}
Classname(parameter-list2) { //constructor
Body2;
}
Type methodname(parameter-list3) {
Body;
        }
}
```

Default constructor

```
class Simple{

  //properties
 int no1,no2;
 // default constructor
 Simple(){
          no1 = 10;
   no2 = 20;
 }
```

```
        //behaviour - methods
  int mul() {
    int i=no1*no2;
    return i;

   }
}
public class DefaultConstructor {
    public static void main(String[] args) throws Exception {
        // Your code here!

      Simple s1= new Simple();
      int k = s1.mul();
      System.out.println("Multiplication "+k);
    }
}
```

The above default constructor program will create objects with same initialized values. In some cases we need to create objects with different initialized values in each object. For this we have parameter constructor in java.

In parameter constructor, when creating object we need to give initialized values.

Let us understand the program on Parameter constructor

```
class Simple{

   //properties
  int no1,no2;
  // default constructor
  Simple(int a, int b){
```

```
            no1 = a;
    no2 = b;
  }

          //behaviour - methods
  int mul() {
    int i=no1*no2;
    return i;

   }
}
public class ParameterConstructor {
    public static void main(String[] args) throws Exception {

      Simple s1= new Simple(10,20);
      int k = s1.mul();
      System.out.println("Multiplication "+k);
    }
}
```

This

In declaring methods and constructors, parameters are used as arguments. The name of parameters are generally different from that of instance variables. However, java allows parameters and instance variables to have the same name. in order to identify instance variables from formal parameters, the key word this is used. this refers to the current object. The following program illustrates the use of this in a class:

```
class Simple{

  //properties
 int no1,no2;
 // default constructor
 Simple(int no1, int no2){
            this.no1 = no1;
   this.no2 = no2;
 }

          //behaviour - methods
 int mul() {
   int i=no1*no2;
   return i;

  }
}

public class ThisEx {
   public static void main(String[] args) throws Exception {

     Simple s1= new Simple(10,20);
     int k = s1.mul();
     System.out.println("Multiplication "+k);
   }
}
```

Output:

```
Multiplication 200
```

Constructors having the same name with different parameter list is called constructor overloading.

this in constructor

One constructor of a class can refer to another constructor of the same class through the keyword this. The following program illustrates the use of this to refer another constructor of the same class:

```
class Simple{


  //properties
        int no1,no2;
         // default constructor
         Simple(int no1, int no2){
                this.no1 = no1;
                this.no2 = no2;
        }
        Simple(Simple os){
                this(os.no1,os.no2);
        }
}
public class ThisEx {
   public static void main(String[] args) throws Exception {


        Simple s1= new Simple(10,20);
        Simple s2= new Simple(s1);
        System.out.println("s1.no1 :"+s1.no1);
        System.out.println("s1.no2 :"+s1.no2);


        System.out.println("s2.no1 :"+s2.no1);
        System.out.println("s2.no2 :"+s2.no2);
        }
}
```

Output:

```
s1.no1 :10
s1.no2 :20
s2.no1 :10
s2.no2 :20
```

Having seen simple methods and constructors, we will now consider a method that does some computing and returns a value to the calling section. The following program calculates the area of the triangle through the area method:

```
class Triangle{


  //properties
 double b,h;
 // default constructor
 Triangle(){
   b=10;
   h=20;
 }
 //behaviour - methods


 double area(double i, double j){
   b = i;
   h = j;
          return(b*h/2);
 }
}



public class ExMethodinClass {
   public static void main(String[] args)  {
      // Your code here!
```

```
    double area;
    Triangle ot= new Triangle();
    area = ot.area(10,12);
    System.out.println("Height of Triangle is : "+ot.h);
    System.out.println("Base of Triangle is : "+ot.b);
    System.out.println("Area of Triangle is : "+area);
   }
}
```

Output:

```
Height of Triangle is : 12.0
Base of Triangle is : 10.0
Area of Triangle is : 60.0
```

One can also print the required values within the method itself instead of calling the method to compute the required values and print them out. The following program does that kind of job:

```
class Triangle{

  //properties
 double b,h;
 // default constructor
 Triangle(double i, double j){
   b = i;
   h = j;
 }
 //behaviour - methods

 void area(){
   System.out.println("Area of Triangle is : "+(b*h/2));
 }
}
```

```
public class ExPrintInMethod {
   public static void main(String[] args)  {
      // Your code here!

     double area;
     Triangle ot= new Triangle(10,12);

     System.out.println("Height of Triangle is : "+ot.h);
     System.out.println("Base of Triangle is : "+ot.b);
     ot.area();
   }
}
```

Output:

```
Height of Triangle is : 12.0
Base of Triangle is : 10.0
Area of Triangle is : 60.0
```

3.3 Method overloading

A class can contain any number of methods. We can pass input values to Methods through parameters. These parameters can be any number in a method. Java permits many methods to have the same name, but they vary in types of parameters or no of parameters. This way of methods having the same name, but with a different type of signature is known as method overloading.

The following program illustrates the use of method overloading. The following program contains a class with two methods with the same name area with a different type of signature. When two parameters are given, one method finds out the area of a rectangle and another one finds out the area of a square when a single parameter is passed as its argument.

Program on method over loading

```
class Triangle{
```

```
  //properties
  double b,h;
  // default constructor
  Triangle(){
    b=10;
    h=20;
  }
  //behaviour - methods
  double area(){
    return(b*h/2);
  }
  double area(double i){
    b = i;
    h = 20;
            return(b*h/2);
  }
  double area(double i, double j){
    b = i;
    h = j;
            return(b*h/2);
  }
}

public class OverMethod {
    public static void main(String[] args) throws Exception {
        // Your code here!

      double area;
      Triangle ot= new Triangle();
```

```
    area = ot.area();
    System.out.println("Area of Triangle is : "+area);
    area = ot.area(10.5);
    System.out.println("Area of Triangle is : "+area);
    area = ot.area(10.5,5.5);
    System.out.println("Area of Triangle is : "+area);
  }
}
```

Output:

```
Area of Triangle is : 100.0
Area of Triangle is : 105.0
Area of Triangle is : 28.875
```

Passing objects as parameter to methods

We have seen that methods in a class can take parameters as inputs and process them. Objects can also be passed as parameter to the methods. The following program contains methods that take object as parameter:

```
class Triangle{

  //properties
 double b,h;
 // default constructor
 Triangle(double i, double j){
   b = i;
   h = j;
 }
 //behaviour - methods

 double area(Triangle t){
```

```
    return(t.b*t.h/2);
  }
}

public class ObjParmtoMethod {
    public static void main(String[] args)  {
        // Your code here!

      double area;
      Triangle ot= new Triangle(10.5,12.5);
      area = ot.area(ot);//passing object of Triangle
      System.out.println("Area of Triangle is : "+area);

    }
}
```

Output:

```
Area of Triangle is : 65.625
```

In the above program, we have seen that an object is passed as reference to access the values of the parameters. This method of passing parameters is known as passing by reference, as the object reference variables contain only reference to the object. When arguments are passed as values, it is known as passing by value. Passing of the parameters in the earlier programs was done through passing by value.

3.4 Finalize method:

- In Java we have finalize() method, which is going to called before object being garbage collected. It ensures an object is cleaned before garbage collector destruct it.
- For example, you might use finalize() to make sure that an open file owned by that object is closed.
- To add a finalizer to a class, you need to define the finalize( ) method. The Java runtime calls the method whenever it is about to recycle an object of that class.
- In finalize() method, You can mention all cleaning procedures before the object is destoryed

The finalize() method has this general form −

```
protected void finalize() {
// finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize() by code defined outside its class.

For Example:

```
public class Example
{
    public static void main(String[] args)
    {
        Example m = new Example();
        m.finalize();
        m = null;
        System.gc();
        System.out.println("We are at the end of main");
    }

    public void finalize()
    {
        System.out.println("we are in finalize ");
    }
}
```

Output:

we are in finalize

We are at the end of main

we are in finalize

3.5 LIST OF REFERENCES

1. Java 2: The Complete Reference, Fifth Edition, Herbert Schildt, Tata McGraw Hill.

2. Programming in JAVA2,Dr. K. Somasundaram, JAICO Publishing House

3.6 BIBLIOGRAPHY

http://www.javabeginner.com/

https://www.tutorialspoint.com/

https://www.geeksforgeeks.org/

https://www.javatpoint.com/java-tutorial

https://guru99.com

3.7 QUESTIONS:

1. Give Advantages and Disadvantages of classes in java.
2. Explain types of constructors in java with example.
3. Explain method overloading in java with example.
4. Explain finalizer method.

Inheritance, Packages and Interface

Objectives

1. Introduction
2. Single Inheritance
3. Multilevel Inheritance
4. Hierarchical inheritance
5. Using super
6. Method overloading
7. Abstract classes and Methods
8. Packages
9. Declaring packages
10. Access Protection
11. Importing packages
12. Interfaces
13. Implementing Interfaces
14. Summary
15. List of references
16. Bibliography
17. Questions

4.1 Introduction

The mechanism of inheriting a new class from an existing one is called inheritance.

The existing class is referred as supper class and new class is referred as sub class.

Java strongly supports the concept of reusability. Once a class has been designed, created and tested, it can be adopted by other programmers in their programs.

This is basically done by creating new classes using existing classes ie., reusing the properties and methods of the existing ones.

Functions and variables of a class can be used by object of another class.

Inheritance may take following forms:-

1) Single inheritance:- Only one super class

2) Multiple inheritance:- java not support multiple inheritance

3) Multilevel inheritance:- Derived from a derived class

4) Hierarchal inheritance:- One superclass, many subclasses

Syntax:

```
class A{
……..
}
class B extends A{
   ……..
}
```

4.2 Single Inheritance

A sub class with only one super class is called as single inheritance.

Here A is a super class and B is a sub class.

Here is an example program on single inheritance

```
class A{
   void number(){
     System.out.println("Super class A");
   }
}
class B extends A{
   void numb(){
     System.out.println("Sub class to A");
   }
}
public class SingleInhe{
   public static void main(String args[]){
     B ob=new B();
     ob.number();
     ob.numb();
```

```
  }
}
```

Output:

```
Super class A
Sub class to A
```

4.3 Multilevel Inheritance

The mechanism of inheriting one class from another super class, that super type is inheriting one more super class is known as multilevel inheritance.

The class A serves as a super class for sub class B which in turn acts as a super class for the sub class C.

The class B is known as intermediate super class since it provides link for the inheritance between A and C. The chain ABC is known as Inheritance path.

Multilevel inheritance can be declared as follows

```
Class A
{
……..
}

Class B extends A{
…….
}

Class C extends B{
……..
```

```
}
```

Here is an example program on single inheritance

```
class A{
  void number(){
    System.out.println("Super class A");
  }
}
class B extends A{
  void numb(){
    System.out.println("Sub class to A");
  }
}
class C extends B{
  void num(){
    System.out.println("Sub class to B");
  }
}

public class ExMultiLevel{
  public static void main(String args[]){
    C oc=new C();
    oc.number();
    oc.numb();
    oc.num();

    }
}
```

Output:

```
Super class A
Sub class to A
Sub class to B
```

4.4 Hierarchical inheritance

Two or more classes are inheriting from same super class is known as Hierarchical inheritance.

```
class A{
  void number(){
    System.out.println("Super class A");
  }
}

class B extends A{
  void numb(){
    System.out.println(" B Sub class to A");
  }
}

class C extends A{
  void num(){
    System.out.println("C Sub class to A");
  }
}

public class ExhirInheritance{
    public static void main(String args[]){
        C oc=new C();
        oc.number();
        //oc.numb(); it gives an error
        oc.num();
```

```
   }
}
```

Output:

```
Super class A
C Sub class to A
```

4.5 Using super

When a Super class needs to be referred using the keyword ‘super’.

Super has two general forms.

1) Super class constructor can be called using ‘super’ keyword.

super(*parameter-list*);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass.

2) A member of Super class hidden by the subclass can be referred using the ‘super’ keyword.

super.*member*

Here, *member* can be either a method or an instance variable.

Note: a subclass constructor must have super() as the first statement in definition.

Example:-

```
class A{
  int ai,aj;
  A(int x,int y){
        ai=x;
    aj=y;
  }
  void suma(){
  System.out.println("\nai+aj : "+(ai+aj));
   }
```

```
}
class B extends A{
  int bi;
  void sumb(){
    System.out.println("ai+aj+bi : "+(ai+aj+bi));
  }
  B(int a, int b, int c){
    super(a,b);
    bi=c;
  }
}

public class ExSuper{
  public static void main(String args[]){
    B ob1=new B(10,20,30);
    ob1.suma();
    ob1.sumb();
  }
}
```

Output:

```
ai+aj : 30
ai+aj+bi : 60
```

4.6 Method Overriding

When a method in subclass and superclass has same name and type signature, then the method of superclass is override by the subclass which is known as method overriding.

When an overridden method is called, it will refer to the method of the subclass.

When a method in subclass and superclass has same name, same arguments, and same return type, then the method of superclass is override by the subclass which is known as method overriding.

Example:-

```
class A{
  int ai,aj;
  A(int x,int y){
            ai=x;
    aj=y;
  }
  void sum(){
  System.out.println("\nai+aj : "+(ai+aj));
   }
}
class B extends A{
  int bi;
  void sum(){
    System.out.println("ai+aj+bi : "+(ai+aj+bi));
  }
  B(int a, int b, int c){
    super(a,b);
    bi=c;
  }
}

public class ExMethodOverride{
  public static void main(String args[]){
    B ob1=new B(10,20,30);
    ob1.sum();// this calls sum of B class
  }
}
```

Output:

```
ai+aj+bi : 60
```

4.7 Abstract Classes and Methods

Abstract class is used for design Convenience. It can have n number of methods. The methods are only declared, they don't have any definition i.e. no method body or implementation.

Syntax of abstract method, use this general form:

abstract *type name(parameter-list)*;

Any class that contains one or more abstract methods must also be declared

abstract.

To create an abstract class use the 'abstract' keyword before the class name during its class declarations.

No objects of an abstract class can be created.

No 'new' keyword can be used for abstract class.

Constructors and static methods cannot be created in abstract class.

All Abstract methods of an Abstract class must be implemented in its subclass. If any of the abstract methods of an abstract class are not being implemented in the sub class than sub class will become an abstract class. So, the keyword 'abstract' should be added before the subclass name.

//Program to be reviewed.

simple example of a class with an abstract method, followed by a class

which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callAM();
  // concrete methods are still allowed in abstract classes
  void callA() {
    System.out.println("Concrete method callA().");
  }
}
class B extends A {
  void callAM() {
```

```
    System.out.println("B's implementation of callAM.");
  }
}

class AbstractEx {
  public static void main(String args[]) {
    B b = new B();
    b.callAM();
    b.callA();
  }
}
```

Output:

```
B's implementation of callAM.
Concrete method callA().
```

No objects of class **A** are declared in the program. It is not possible to instantiate an abstract class. class **A** implements a concrete method called **callAM( )**. This is perfectly acceptable.

4.8 Packages

Grouping of related classes and interfaces together in single unit is called as package.

Packages are used for managing large group of classes and interfaces to avoid naming conflicts.

Java API itself is implemented as a group of packages.

Package are classified as two categories and they are as follows –

User defined packages

Built in packages

4.9 Declaring Packages

The syntax for the package statement follows:

package <package name>;

First statement of java program, should be ‘package’ statement [If packages are used].

Must be placed before any class declarations.

A class using package statement is considered as the part of that package.

Creating hierarchy of packages is supported by Java.

Syntax:

package *pkg1*[.*pkg2*[.*pkg3*]];

Example:

```
package thread;

public class ExUserDefinedPackage {
   public static void main(String args[]){
      System.out.print("Hello");
   }
   void runner(){
      System.out.print("This method is running");
   }
}
```

The above class can be re-used by the following way

```
package thread;
import thread.ExUserDefinedPackage;

public class Thread1 {
   public static void main(String args[]){
     ExUserDefinedPackage e=new ExUserDefinedPackage();
     e.runner();
   }

}
```

Built-in package Examples

Ex: 1) Import java.util.Scanner;

```
package thread;

import java.util.Scanner;

public class ExUtilScanner{
```

```
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        String name = sc.nextLine();
        System.out.println("Name: "+name);
    }
}
```

4.10 Access Protection

Java provides multi-level protection through the visibility of variables and methods within classes, subclasses, and packages

- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.

Java addresses four categories of visibility for class members:

1. Subclasses of the same package
2. Non-subclasses of the same package
3. Subclasses of different packages
4. Classes not belonging to the same package nor subclasses

Various Access specifiers are:

1. Default
2. Private
3. Public
4. Protected

The Default Access Modifier

It specifies that only classes of the same package can have access to its class variables and methods. Its visibility is limited to other classes within the same package.

The public Access Modifier

It specifies that class variables and methods are accessible to anyone i.e. both inside and outside the class. It has global visibility and can be accessed by any other objects.

The protected Access Modifier

It specifies that class members are accessible only to methods of that class and subclasses of that class. It has visibility limited to subclasses.

The private Access Modifier

It specifies that class members are accessible only by the class in which they are defined. It is the most restricted access modifier.

// to be review

```
class A{
        int i; //public by default
        private int j; //private to A
        void setij(int x, int y){
                i = x;
                j = y;
        }
}
//A's j is not accessible here.
class B extends A{
        int total;
        void sum(){
                //total = i + j; //Error, j is not accessible here
        }
}
class Access{
        public static void main(String args[]){
                B subOb = new B();

                subOb.setij(10,12);
                subOb.sum();
                System.out.println("Total is"+subOb.total);
        }
}
```

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**.This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

This is file **Protection.java**:

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **Derived.java**:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
```

```
}
```

This is file **SamePackage.java**:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);

System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions which are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**.s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
```

```
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **OtherPackage.java**:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
 public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
```

```
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

4.11 Importing Packages

If you want to use class outside the package, than you must import the package which is having that class. In other words, it enables you to import classes from other packages.

Individual classes or entire package can be imported.

Syntax of **import** statement:

import *pkg1*[.*pkg2*].(*classname*|*);

pkg1 is the name of a top-level package, and *pkg2* is the name of a subordinate.

The packages are separated by a dot.

If you are specifying star (*) after package it will import all classes within the package.

4.12 Interfaces

An interface contains only abstract methods and final variables. Interface cannot be inherited, it can only be implemented.

Interface defines protocols for the class without its implementation detail.

Defining an Interface

An interface is defined much like a class.

Syntax:

access interface *name* {

return-type method-name1(*parameter-list*)*;*

return-type method-name2(*parameter-list*)*;*

type final-varname1 = value;

type final-varname2 = value;

// ...

return-type method-nameN(*parameter-list*)*;*

type final-varnameN = value;

}

Access modifier can be public or not specified.

Interface declaration can consist of variables. These variables must be static or final, they cannot be modified. A constant value must be assigned to variable.

Syntax:

interface Interface_name {

//variables and methods

}

4.13 Implementing Interfaces

Interface can be implemented by one or more classes. To implement the interface use the keyword ‘implement’ in the class definition and then implement the methods defined by the interface.

Syntax:

access class *classname* [extends *superclass*]

[implements *interface* [,*interface*...]] {

// class-body

}

An Interface Example

Interface Area

```
{
Final static float pi=3.14f;
float compute(float x, float y);
}
Class Rectangle implements Area {
Public float compute(float x, float y) {
return(x*y);
}
}
Class circle implements Area {
Public float compute(float x, float y) {
return(pi*x*x);
}
}
Class InterfaceTest
{
public static void main(String args[]) {
Rectangle rect = new Rectangle();
circle cir = new circle();
Area area;
area=rect;
System.out.println(.Area of rectangle=.+area.compute(10,20));
Ares=cir;
System.out.println(.Area of circle=.+cir.compute(10,10));
}
}
```

To implement the methods, we need to refer to the class objects as type of the interface rather than types of their respective classes. Note that if a class that implements an interface does not implement all the methods of the interface, than the class became an abstract class and cannot be instantiated.

4.14 Summary

4.15 LIST OF REFERENCES

1. Java 2: The Complete Reference, Fifth Edition, Herbert Schildt, Tata McGraw Hill.
2. Programming in JAVA2,Dr. K. Somasundaram, JAICO Publishing House

4.16 BIBLIOGRAPHY

http://www.javabeginner.com/

https://www.tutorialspoint.com/

https://www.geeksforgeeks.org/

https://www.javatpoint.com/java-tutorial

https://guru99.com

4.17 Questions

1. What is inheritance explain with suitable example
2. What is package and how to create a package?
3. Explain Interface ith suitable example

Exceptions, Strings and Multithreding

Objectives

1. Introduction
2. Catching exceptions
3. Try catch
4. Multiple catch
5. Nested try
6. throw
7. throws
8. Finally
9. String, constructor and String functions
10. String Buffer
11. Multi Thread
12. Creating thread
13. Implementing Runnable
14. Extending Thread
15. Using aLive() and join()
16. Thread Priorities
17. Summary
18. List of references
19. Bibliography
20. Questions

5.1 Introduction

When we are running java programs we may come across errors like syntactical, logical, runtime errors in our programs.

Type of errors

1. Compile time errors
2. Runtime errors

```
class ExError  {
        public static void main(String arg[]) {
                int i=200;
                int j=100  // semi-colon(;) missing
                int k=i-20;
                System.out.println("difference is "+k);
        }
```

}

This program will not compiled because there is a semi-colon missing when declaring the variable j. This is syntax error in Java language. If program is violating the syntax, the compiler will give an error. This kind of error is known as syntactical error or compilation error. The compiler catches the error and its location.

Runtime errors

```
class ExException      {
        public static void main(String arg[])  {
                int i=100;
                System.out.println("First Line");
                System.out.println(i/0);
                System.out.println("Last Line");
        }
}
```

Output:

First Line

Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExException.main(ExException.java:5)

while we are executing program, first statement will be executed and it prints “First Line” on the console. Then control moves to the second statement,
As the program is trying to divide i by 0 ie., 100/0 an exception is raised and the program will be terminated. There are 3 statements in the program. The first line gets executed normally and output will be “First Line”. The next statement System.out.println(i/0); will raise an exception and terminates the program. So the control will not go to the third statement.

Exception

Exception is caused by a runtime error in the program. Even if an exception raised the program will not be terminated abruptly, and to take alternative path to overcome the exception and execute the remaining task.

Example

```
class ExException {
        public static void main(String arg[])  {
                int i=100;
                System.out.println("First Line");
```

```
                System.out.println(i/0);
                System.out.println("Last Line");
        }
}
```

Output:
First Line

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExException.main(ExException.java:5)
```

Types of Exceptions

Class throwable in java is super class of all exceptions. It has two sub classes called Exception and Error. Exception class has two subclasses namely IOException and RuntimeException.

Error :
Errors which are outside the ability to control of the software engineers are managed in this class. disk full and memory insufficient are of this sort. Java doesn't give any system to deal with them and ought not be caught.

Exception

In this scenario, a user can make one's own exception and use it in java program. All special cases under this exceptions are to be caught and taken care of.

IOException

When we come across the scenarios like, File not found, end of file encountered comes under this type. Exceptions due to i/o devices and i/o related are handled in this class.

RuntimeException :

Sometimes a program may compile successfully creating the class file but may not run. Exception occurring in the program code at runtime are handled in this class. Divide by zero error, array index out of bound, wrong cast and null pointer access are of this type. These types of errors could have been avoided, if the programmer had taken care to write the program. These exceptions can be caught and handled by java.

5.2 Catching exceptions

Java developers have identified commonly occurring exceptions and they are specified in the exception. When such exceptions appear in a program at runtime, they are to be caught and

handled. The exception is caught by try…catch mechanism. The general form of the try…catch block is:

```
try     { …

        Statements that may give

        Exceptions

        } catch (exception type e1) {

        Statements to handle

        Exception type1

        } catch (exception type2 e2) {

        Statements to handle exception type2

        } finally {

        Statements to be executed

        Whether an exception occurs or not

        }
```

The terms try, catch and finally are java keywords. e1 and e2 are errors. In the try block, the statements that are suspected to cause exceptions are placed. Exception type1 and exception type2 are the different exception types. The catch block contains statements that are to be executed in the event of occurring of an exception.

Multiple catch blocks for single try block can be set up, each catch block dealing one specific type of exception out of several occurring inside the try block. The try block may be constructed to trap any number of exceptions. If an exception occurs in the try block and if there is no matching catch block, the program is aborted. In case no exception occurs inside the try block, all catch block statements will be skipped and only statements in the finally block will be executed. This finally block will be executed irrespective of whether an exception occurs or not. The catch block statements are operative only for the preceding try block. The finally block is optional.

Example on array index

```
class ArrayIndex{
  public static void main(String args[]){
    int m[]=new int[12];
     m[12]=100;
       System.out.println("End of try block");
```

```
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 12 out of bounds for length 12
        at ArrayIndex.main(ArrayIndex.java:4)
```

5.3 Try Catch

we can place the problem creating a statement in a try block. Generally, the statements that may raise an exception are placed in the 'try' block. If an exception is raised then control goes to the 'catch' block.

If no exception is raised, then the catch block is skipped.

If we write the above program in the following way, then the program will not be terminated abnormally.

Example on array index

```
class ArrayIndex{
  public static void main(String args[]){
    int m[]=new int[12];
    try{
      m[12]=100;
      System.out.println("End of try block");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Array Index out of bounds ");
      m[0]=20;

    }
    System.out.println("Next statement after try-catch block");
    System.out.println("First elt in array is :"+m[0]);
  }
}
```

Output:

```
Array Index out of bounds
Next statement after try-catch block
First elt in array is :20
```

5.4 Multiple catch blocks

Multiple catch blocks can have a try Block.
Each catch block must have different exception handler.
The sub class exceptions are caught first and then the super class exceptions.

If you have try block may raise different exceptions, then we can have java multi-catch block.

```
try {

// method calls go here

}

catch( ExceptionClass1 e1 ) {

// handle ExceptionClass1 exceptions here

}

catch(ExceptionClass2 e2 ) {

// handle ExceptionClass2 exceptions here

}
```

Example program on Multi catch.

```
class ExMulticatch{
  public static void main(String args[]){
    try{
      int arr[]=new int[12];
      arr[12]=10;
      arr[4]=100/0;
      System.out.println("End of try block");
    }
    catch(ArithmeticException e){
      System.out.println("Divide a number by zero");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("We are accessing array elements outside of the limit");
    }
    catch(Exception e){
      System.out.println("Some Other Exception");
    }
    System.out.println("Out of the try-catch block");
  }
}
```

Output:

We are accessing array elements outside of the limit
Out of the try-catch block

When writing multiple catch blocks for try block, sub class exceptions handlers should be placed first and then super class exceptions handlers should be placed latter.

If super class exceptions handlers are placed on the top and then sub class exceptions handlers placed latter.
Java compiler will give an error.

5.5 Nested try blocks

In java try can hold one more try block.
For every try we can have a associated catch block.
The following program demonstrates the nested try.

```
class ExNestedTry {
  public static void main(String args[])
  {
    try {
      int a[] = { 5,6,7,8,9 };
      System.out.println(a[5]);
      try {
        int x = a[2] / 0;
      }
      catch (ArithmeticException e2) {
        System.out.println("division by zero is not possible");
      }
    }
    catch (ArrayIndexOutOfBoundsException e1) {
      System.out.println("ArrayIndexOutOfBoundsException");
      System.out.println("Element at such index does not exists");
    }
  }

}
```

Output:

ArrayIndexOutOfBoundsException
Element at such index does not exists

5.6 throw

It is used to throw an exception forcefully by a method.

Syntax:

throw *ThrowableObject*;

Throwableobject must be an object of type **Throwable** or a subclass of **Throwable**.

The throw statement causes termination of the normal flow of control of the java code and prevents the execution of the subsequent statements.The throw clause convey the control to the nearest catch block handling the type of exception object throws. If no such catch block exists, the program terminates.The throw statement accepts a single argument, which is an object of the Exception class.

```
// Demonstrate throw.

class ThrowDemo {

static void demoproc() {

try {

throw new NullPointerException("demo");

} catch(NullPointerException e) {

System.out.println("Caught inside demoproc.");

throw e; // rethrow the exception

}

}

public static void main(String args[]) {

try {

demoproc();

} catch(NullPointerException e) {

System.out.println("Recaught: " + e);

}

}

}
```

Here is the resulting output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

5.7 Throws

Throws specifies that method may throw an exception i.e. throws is like a warning.

If the user does not want to handle an exception inside a method than he can specify the exception in throws clause of method.

Syntax:

type method-name(parameter-list) throws *exception-list*

{

// body of method

}

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

The example is shown here:

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
```

}

Here is the output generated by running this example program:

inside throwOne

caught java.lang.IllegalAccessException: demo

5.8 Finally

After the execution of try/catch block the finally block will be executed. It will be executed irrespective of an exception i.e. whether an exception is thrown or not.

If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

It can be used for closing files or freeing up the resources and memory space.

example Demonstrate finally.

```
import java.util.Scanner;

class ExFinally

{

  public static void main(String args[]) {

    try{

                int numarator=321;

                Scanner sc=new Scanner(System.in);

      System.out.println("Enter dr : ");

      int denominator=sc.nextInt();

      int k= numarator/denominator;

                System.out.println(“The result is : ”+k);

    }

    catch(ArithmeticException e){

      System.out.println("in catch divided by zero");

    }
```

```
    finally{
            System.out.println("In finally block");
    }
    System.out.println("After try-catch-finally");
   }
}
```

Output:

First time execution

```
Enter dr:10
The result is : 32
In finally block
After try-catch-finally
```

2nd time execution

```
Enter dr : 0
in catch divided by zero
In finally block
After try-catch-finally
```

If this example executed, and in try block no exception will occur as input is 10. Then control goes to the finally block.

Same program was executed for second time to get an exception. In try block exception was occurred as input is 0. The control goes to the catch block and it will be executed. Then control goes to the finally block.

In both the cases the finally block is still executed.

5.9 String, string constructors and string functions

The String class represents A string of characters. A String object is created by the Java compiler whenever it encounters a string in doublequotes. When you create a String object,

you are creating a string that cannot be changed. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string; a new String object is created that contains the modifications. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created.

String Constructors

The String class supports several constructors.

String():- To create an empty String, you call the default constructor.

String(char chars[ ]):- To create a String initialized by an array of characters

String(char chars[ ], int startIndex, int numChars):- You can specify a

subrange of a character array as an initializer

String(String strObj):- You can construct a String object that contains the same

character sequence as another String object.

For example

String str1 = new String();

will create an instance of String with no characters in it.

char chars[] = { 'R', 'A', 'M','A','I','A','H' };

String str2 = new String(chars);// This constructor initializes str2 with the string RAMAIAH..

String str3 = new String(chars, 0, 4);// This initializes s with the characters RAMA

String str4 = new String(str2);

String Length : The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method, shown here:

char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

System.out.println(s.length());

String Operations

Java has added special support for several string operations within the syntax of the

language. These operations are:-

String Concatenation

In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result

For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string .He is 9 years old..

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them.

String Conversion and toString( )

The toString() method returns a string representing the value of an object.

Every class implements toString( ) because it is defined by Object. For most important classes that

you create, you will want to override toString( ) and provide your own string representations. General form:

```
String toString( )
```

The followingprogram demonstrates this by overriding toString( ) for the Box class:

```
// Override toString() for Box class.
class Box {
double width;
double height;
double depth;
```

```
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +
depth + " by " + height + ".";
}
}
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
System.out.println(s);
}
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

As you can see, Box.s toString( ) method is automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object.

charAt( ):-To extract a single character from a String. It has this general form:

char charAt(int where)

Here, where is the index of the character that you want to obtain. charAt( ) returns the character at the specified location. For example,

char ch;

ch = "abc".charAt(1);

assigns the value .b. to ch.

getChars( ):- If you need to extract more than one character at a time, you can use the getChars( ) method. It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.

The following program demonstrates getChars( ):

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

Here is the output of this program:

demo

getBytes( ):- This stores the characters in an array of bytes, and it uses the default character-to-byte conversions Here is its simplest form:

byte[ ] getBytes( ).

getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

toCharArray( ):-If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

String Comparison

equals( ):-To compare two strings for equality, use equals( ). General form:

boolean equals(Object str)

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

equalsIgnoreCase( ):-To perform a comparison that ignores case differences, call equalsIgnoreCase( ).When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:

boolean equalsIgnoreCase(String str)

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

Here is an example that demonstrates equals( ) and equalsIgnoreCase( ):

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
```

```
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " +
s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " +
s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4));
}
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

regionMatches( ):-The regionMatches( ) method compares a specific region inside a string with another specific region in another string. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2, int str2StartIndex, int
numChars)
boolean regionMatches(boolean ignoreCase, int startIndex, String str2,
int str2StartIndex, int numChars)
```

For both versions, startIndex specifies the index at which the region begins within the invoking String object.

startsWith( ):- The startsWith( ) method determines whether a given String begins with a specified string. general forms:

```
boolean startsWith(String str)
```

boolean startsWith(String str, int startIndex)

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,

"Foobar".startsWith("bar", 3)

returns true.

endsWith( ):- endsWith( ) determines whether the String in question ends with a specified string. general forms:

boolean endsWith(String str)

For example,

"Foobar".endsWith("bar")

and

"Foobar".startsWith("Foo")

are both true.

equals( ) Versus ==

the **equals( )** method compares the characters inside a **String** object. The == operator compares two object references to see whether they refer to the same instance. Example:

```
// equals() vs ==
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

Output of the above example:

Hello equals Hello -> true

Hello == Hello -> false

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The **String** method **compareTo( )** serves this purpose. It has this general form:

int compareTo(String *str*)

Here, str is the String being compared with the invoking String.

If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, shown here:

int compareToIgnoreCase(String str)

This method returns the same results as compareTo( ), except that case differences are ignored

Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

indexOf( ) Searches for the first occurrence of a character or substring.

lastIndexOf( ) Searches for the last occurrence of a character or substring.

To search for the first occurrence of a character, use

int indexOf(int ch)

To search for the last occurrence of a character, use

int lastIndexOf(int ch)

Here, ch is the character being sought.

To search for the first or last occurrence of a substring, use

int indexOf(String str)

int lastIndexOf(String str)

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int ch, int startIndex)

int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)

int lastIndexOf(String str, int startIndex)

Here, startIndex specifies the index at which point the search begins. For indexOf( ),the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero.

Example :-

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
"to come to the aid of their country.";
System.out.println(s);
System.out.println("indexOf(t) = " +
s.indexOf('t'));
System.out.println("lastIndexOf(t) = " +
s.lastIndexOf('t'));
System.out.println("indexOf(the) = " +
s.indexOf("the"));
System.out.println("lastIndexOf(the) = " +
s.lastIndexOf("the"));
```

```
System.out.println("indexOf(t, 10) = " +
s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " +
s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " +
s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60));
}
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

substring( )

You can extract a substring using substring( ). It has two forms. The first is

```
String substring(int startIndex)
```

Here, startIndex specifies the index at which the substring will begin.

The second form of substring( ) allows you to specify both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex)

Here, startIndex specifies the beginning index, and endIndex specifies the stopping point.

The following program uses substring( ) to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace
{
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i;
do { // replace all matching substrings
System.out.println(org);
i = org.indexOf(search);
if(i != -1) {
result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
org = result;
}
} while(i != -1);
}
}
```

The output from this program is shown here:

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

concat( ):-You can concatenate two strings using concat( ), shown here:

String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end.

For example,

String s1 = "one";

String s2 = s1.concat("two");

puts the string .onetwo. into s2

replace( ):- The replace( ) method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement)

Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,

String s = "Hello".replace('l', 'w');

puts the string .Hewwo. into s.

trim( ):-The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Here is an example:

String s = " Hello World ".trim();

This puts the string .Hello World. into s.

Data Conversion Using valueOf( )

The valueOf( ) method converts data from its internal format into a human-readable form. It is a static method that is overloaded within String for all of Java.s built-in types, so that each type can be converted properly into a string.Here are a few of its forms:

static String valueOf(double num)

static String valueOf(long num)

static String valueOf(Object ob)

static String valueOf(char chars[])

valueOf( ) is called when a string representation of some other type of data is needed. for example, during concatenation operations. Any object that you pass to valueOf( ) will return the result of a call to the object.s toString( ) method.

5.10 StringBuffer

□□StringBuffer is a peer class of String that provides much of the functionality of strings.

□□String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.

□□StringBuffer may have characters and substrings inserted in the middle or appended to the end.

StringBuffer Constructors

StringBuffer defines these three constructors:

StringBuffer( ):- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.

StringBuffer(int size):- This version accepts an integer argument that explicitly sets the size of the buffer.

StringBuffer(String str):- This version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

StringBuffer methods:-

□□**length( ) and capacity( ):-** The current length of a StringBuffer can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method general forms:

int length( )

int capacity( )

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out.println("capacity = " + sb.capacity());
}
}
```

Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

□□**ensureCapacity( ):-** If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity() to set the size of the buffer. General form:

void ensureCapacity(int capacity)

Here, capacity specifies the size of the buffer.

setLength( ):-To set the length of the buffer within a StringBuffer object, use setLength( ). General form:-

void setLength(int len)

Here, len specifies the length of the buffer.

charAt( ) and setCharAt( ):-The value of a single character can be obtained from a StringBuffer via the charAt( ) method. You can set the value of a character within a StringBuffer using setCharAt( ).General form:-

char charAt(int where)

void setCharAt(int where, char ch)

For charAt( ), where specifies the index of the character being obtained. For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character.

append( ):-The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has overloaded versions for all the built-in types and for Object. Here are a few of its forms:

StringBuffer append(String str)

StringBuffer append(int num)

StringBuffer append(Object obj)

```
// Demonstrate append().
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString();
```

```
System.out.println(s);
}
}
```

The output of this example is shown here:

```
a = 42!
```

The append( ) method is most often called when the + operator is used on String objects.

insert( ):-The insert( ) method inserts one string into another. It calls String.valueOf( ) to obtain the string representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. These are a few of its forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

The following sample program inserts .like. between .I. and .Java.:

```
// Demonstrate insert().
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);
}
}
```

The output of this example is shown here:

I like Java!

reverse( ):- You can reverse the characters within a StringBuffer object using reverse( ), shown here:

StringBuffer reverse( )

The following program demonstrates reverse( ):

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

Here is the output produced by the program:

abcdef

fedcba

delete( ) and deleteCharAt( ):- The delete( ) method deletes a sequence of characters from the invoking object. The general form is:-

StringBuffer delete(int startIndex, int endIndex)

Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove.

The deleteCharAt ( ) method deletes the character at the index specified by loc.

StringBuffer deleteCharAt (int loc)

This method deletes the character at the index specified by loc.

Here is a program that demonstrates the delete( ) and deleteCharAt( ) methods:

```
// Demonstrate delete () and deleteCharAt ()
class deleteDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.delete(4, 7);
System.out.println("After delete: " + sb);
sb.deleteCharAt(0);
System.out.println("After deleteCharAt: " + sb);
}
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( ):-It replaces one set of characters with another set inside a StringBuffer object. Its general form is:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex.1 is replaced.

The following program demonstrates replace( ):

```
// Demonstrate replace()
class replaceDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
}
}
```

Here is the output:

After replace: This was a test.

substring( ):-This method returns a portion of a StringBuffer. It has the following

two forms:

String substring(int startIndex)

String substring(int startIndex, int endIndex)

The first form returns the substring that starts at startIndex and runs to the end of

The invoking StringBuffer object. The second form returns the substring that starts at

startIndex and runs through endIndex.1.

5.11 Multithreading

Multithreaded programming is one of the features supported by Java languages.

What Is a Thread?

In the early days of computing, computers were single tasking--that is, they ran a single job at a time. Multitasking refers to a computer's capability to perform multiple jobs concurrently. Multithreading is an extension of the multitasking paradigm. But rather than multiple programs, multithreading involves multiple threads of control within a single program. For example, using a Web browser, you can print one Web page, download another, and fill out a form in a third--all at the same time.

A thread is a single sequence of execution within a program.

Java threads allow you to write programs that do many things at once. Each thread represents an independently executing sequence of control. One thread can write a file out to disk while a different thread responds to user keystroke events.

Threads exist in several states. A thread can be *running.* It can be *ready to run* as

soon as it gets CPU time. A running thread can be *suspended,* which temporarily suspends its activity. A suspended thread can then be *resumed,* allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface

Java.s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads.

Method Meaning

getName Obtain a thread.s name.

getPriority Obtain a thread.s priority.

isAlive Determine if a thread is still running.

join Wait for a thread to terminate.

run Entry point for the thread.

sleep Suspend a thread for a period of time.

Start Start a thread by calling its run method.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:

It is the thread from which other .child. threads will be spawned.

Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object.

Let.s begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try {
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted");
}
}
}
```

Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

5.12 Creating a Thread

you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

You can implement the **Runnable** interface.

You can extend the **Thread** class, itself.

5.13 Implementing Runnable

Runnable *abstracts a unit of executable code. To implement* ***Runnable****, a class need only implement a single method called* ***run( )****, which is declared like this:*

public void run( )

Inside ***run( )****, you will define the code that constitutes the new thread.* ***run( )*** *can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that* ***run( )*** *establishes the entry point for another.*

After you create a class that implements ***Runnable****, you will instantiate an object of type* ***Thread*** *from within that class.* ***Thread*** *defines several constructors. The one that we will use is shown here:*

Thread(Runnable threadOb, String threadName)

In this constructor, threadOb is an instance of a class that implements the ***Runnable*** *interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.*

After the new thread is created, it will use a ***start( )*** *method to executes a call to* ***run( )****.*

The ***start( )*** *method is shown here:*

void start( )

Here is an example that creates a new thread and starts it running:

```
public class RunnableExample implements Runnable{
```

```
	public void run(){
		for(int i=0; i < 5; i++){
			System.out.println("Child Thread : " + i);
		}
		System.out.println("Child thread finished!");
	}
	public static void main(String[] args) {
		Thread t = new Thread(new RunnableExample(), "My Thread");
		t.start();
		for(int i=0; i < 5; i++){
			System.out.println("Main thread : " + i);
		}
		System.out.println("Main thread finished!");
	}
}
```

Output :

```
Main thread : 0
Main thread : 1
Main thread : 2
Main thread : 3
Main thread : 4
Main thread finished!
Child Thread : 0
Child Thread : 1
Child Thread : 2
Child Thread : 3
Child Thread : 4
Child thread finished!
```

5.14 Extending Thread

*The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, and **start( )** to begin execution of the new thread.*

Example program on thread

// Create a thread by extending Thread

```
public class CThread extends Thread{
  public void run(){
     System.out.println("Thread is Running.");
  }
  public static void main(String[] args) {
     CThread c=new CThread();
     c.start();
  }

}
```

Output:

```
Thread is Running.
```

This program generates the same output as the preceding version. The child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**. Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)

Here, *threadName* specifies the name of the thread.

5.15 Using isAlive( ) and join( )

Two ways exist to determine whether a thread has finished:-

isAlive( ) :-This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running.

It returns **false** otherwise.

join( ):- This method is used to wait for a thread to finish. and its general form:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates.

```
// Using join() to wait for threads to finish.
class ExJoinThread extends Thread {
  public void run() {
    try {
      for(int n=1;n<=5;n++)    {
        Thread.sleep(500);
        System.out.println(n);
      }
    }
    catch(InterruptedException e)  {
      System.out.println(e);
    }
  }
  public static void main(String args[])  {
    JoiningThread T1=new JoiningThread();
    JoiningThread T2=new JoiningThread();
    JoiningThread T3=new JoiningThread();
    T1.start();
    try    {
      T1.join();
    }
    catch(InterruptedException e)    {
```

```
      System.out.println(e);
    }
    T2.start();
    T3.start();
  }
}
```

Output:

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

5.16Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another.

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower priority threads.

A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

To set a thread.s priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread.

The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**.

Currently, these values are 1 and 10, respectively. To return a thread to default priority,

specify **NORM_PRIORITY**, which is currently 5.

You can obtain the current priority setting by calling the **getPriority( )** method of

Thread, shown here:

final int getPriority( )

The following example demonstrates two threads at different priorities, which do not run

on a preemptive platform in the same way as they run on a nonpreemptive platform.

```
// Demonstrate thread priorities.
class clicker implements Runnable {
int click = 0;
Thread t;
private volatile boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
}
}
public void stop() {
```

```
running = false;
}
public void start() {
t.start();
}
}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
```

```
System.out.println("Low-priority thread: " + lo.click);

System.out.println("High-priority thread: " + hi.click);

}

}
```

The output of this program:-

Low-priority thread: 4408112

High-priority thread: 589626904

5.17 SUMMARY :

After reading this chapter, we learn Exception handling in Java, Features of strings in Java, Multithreaded programming in Java with example.

5.18 LIST OF REFERENCES

1. Java 2: The Complete Reference, Fifth Edition, Herbert Schildt, Tata McGraw Hill.

2. Programming in JAVA2,Dr. K. Somasundaram, JAICO Publishing House

BIBLIOGRAPHY

http://www.javabeginner.com/

https://www.tutorialspoint.com/

https://www.geeksforgeeks.org/

https://www.javatpoint.com/java-tutorial

https://guru99.com

http://java.sun.com/docs/books/tutorial/essential/exceptions/

Questions

1. What is exception? Explain various exceptions in detail.
2. Write various constructors available for string class.
3. Explain multithreading with suitable programs.

Chap. No. 6: File

6.0 OBJECTIVES:

- ✓ To understand what is a file
- ✓ learn how to save data in a file
- ✓ learn different methods of a file
- ✓ To understand Basic Usage of File and how to create directory

6.1 Introduction

This chapter introduces the reader to the File concept in java. Java has several methods for creating, reading, updating, and deleting files. A file is an object on a computer that stores data, information.

6.2 File:

File class is a part of **java.io** package. File class represents the files and directory pathnames in an abstract manner.

If you are looking for class to handle file and directory operations, the File class suits this requirement. Let's take a look at a more detailed functionality of File class.

Java File class represents the files and directory pathnames in an abstract manner.

6.2.1 File Class Syntax

We can create an instance of File class using below constructors.

File(String pathname): Using this constructor we can create an instance of File class from specified string pathname by converting it into an abstract pathname

File(File parent, String child): Using this constructor we can create an instance of File class from specified parent abstract pathname and a child pathname string.

File(String parent, String child): Using this file constructor we can create an instance of File class from specified parent and child pathname.

Write a Program to show different way to create java.io.File instance.

package filedef;

```
import java.io.File;

public class FileDef

{

   public static void main(String[] args)

        {

                //First Constructor

                File file = new File("D:/Rahul/data/file.txt");

                //Second Constructor

                File parent = new File("D:/Rahul/");

                File file2 = new File(parent, "data2/file2.txt");

                //Third Constructor

                File file3 = new File("D:/Rahul/", "data3/file3.txt");

                System.out.println("First : "+file.getAbsolutePath());

                System.out.println("Second : "+file2.getAbsolutePath());

                System.out.println("Third : "+file3.getAbsolutePath());

         }

}
```

Output:

```
First : D:\Rahul\data\file.txt

Second : D:\Rahul\data2\file2.txt

Third : D:\Rahul\data3\file3.txt
```

6.2.2 File Methods

Let’s have a look at the below methods of File class

Create new file: Java File class provides a **createNewFile()** method which automatically creates new empty file named by this abstract pathname if and only if a file with this name does not exist.

Write a Program to create a new file.

```
package filedef;
import java.io.File;
import java.io.IOException;
public class CreateFile
{
   public static void main(String[] args)
        {
              //initialize File constructor
              File file = new File("D:/Rahul/file.txt");
              try
              {
                    boolean createFile = file.createNewFile();
                    if (createFile)
                    {
                          System.out.println("New File is created.");
                    }
                    else
                    {
                          System.out.println("File already exists.");
                    }
              }
              catch (IOException e)
                    {
                    e.printStackTrace();
```

```
                    }

        }
}
```

Output:

New File is created

Check file permission: File class provides several methods to check file permissions (Execute/Read/Write) – **canExecute(), canWrite(),canRead()** .

Write a Program to implement file permission

```
package filedef;
import java.io.File;
public class filePermission
{
   public static void main(String[] args)
   {
                //initialize File constructor
                File file = new File("D:/Rahul/file.txt");
                System.out.println("File is readable? "+file.canRead());
               System.out.println("File is writable? "+file.canWrite());
               System.out.println("File is executable? "+file.canExecute());
   }

}
```

Output:

File is readable? true

File is writable? true

File is executable? true

Check file already exist or not: File class provides **exist()** method that checks whether the file or directory exists or not. This method returns true if file or directory exists or else returns false.

Write a Program to Check file already exist or not

```
package filedef;
import java.io.File;
public class FileExist
{
   public static void main(String[] args)
   {
                // initialize File constructor
                File file = new File("D:/Rahul/file.txt");
                System.out.println("File Exists : "+file.exists());

                File nonExistfile = new File("D:/Rahul/nofile.txt");
                System.out.println("File Exists : "+nonExistfile.exists());
        }
 }
```

Output:

```
File Exists : true
File Exists : false
```

Delete files in java: File class provides **delete( )** method that deletes the file denoted by this abstract pathname. This method returns true if file gets deleted or else returns false.

Write a Program Delete files in java

```
package filedef;
import java.io.File;
```

```
public class DeleteFile {
  public static void main(String[] args)
   {
                // initialize File constructor
                File file = new File("D:/Rahul/file.txt");

                boolean delete = file.delete();
                if (delete)
            {
                        System.out.println("File Deleted");
                } else
            {
                        System.out.println("File not Deleted");
                }
}
```

Output:

File not Deleted

File Absolute & Canonical Path- Java File class provides two methods to fetch absolute and canonical path-**getAbsolutePath() and getCanonicalPath().** Below example code will clear any confusion around the difference between them.

Write a Program to fetch File Absolute & Canonical Path

```
package filedef;
import java.io.File;
import java.io.IOException;
public class absoluteAndCanonicalPath
{

   public static void main(String[] args) throws IOException
```

```
{
        File file = new File("/Users/Siom/source.txt");
        File file1 = new File("/Users/Siom/temp/../source.txt");
        System.out.println("Absolute Path : " + file.getAbsolutePath());
        System.out.println("Canonical Path : " + file.getCanonicalPath());
        System.out.println("Absolute Path : " + file1.getAbsolutePath());
        System.out.println("Canonical Path : " + file1.getCanonicalPath());
    }
}
```

Output:

Absolute Path : C:\Users\Siom\source.txt
Canonical Path : C:\Users\Siom\source.txt
Absolute Path : C:\Users\Siom\temp\..\source.txt
Canonical Path : C:\Users\Siom\source.txt

6.3 Directories:

A directory is defined as an organizational unit, or container, used to organize folders and files into a hierarchical structure. You can think of a directory as a file cabinet that contains folders that contain files.

6.3.1 Java File class provides two methods to create directories which are given below.

mkdir(): This method creates the directory named by this abstract pathname.

It creates only single directory and returns true if the directory gets created or else returns false.

mkdirs(): This method creates the directory named by this abstract pathname including subdirectories and returns true if directories gets created or else returns false.

Write a Program to create directories

```
package filedef;
import java.io.File;
public class createDirectory
{
   public static void main(String[] args)
   {
            // initialize File constructor
            File file = new File("D:/Rahul/");
            boolean created = file.mkdir();
            if (created)
          {
                  System.out.println("Directory created");
            }
          else
          {
                  System.out.println("Directory is not created");
            }
            //create directories including sub directories
            File file2 = new File("D:/Rahul/data/java");
            boolean creatSub = file2.mkdirs();
            if (creatSub)
          {
                  System.out.println("Directory including sub directories
                  created");
            }
```

```
            else
        {
                System.out.println("Directory including sub directories are
                not created");
            }
    }
}
```

Output:

Directory is not created

Directory including sub directories created

6.3.2 List of Files in Directory :

We can use **listFiles()** method, which returns an array of abstract pathnames denoting the files in the directory.

Write a Program to catch List of Files in Directory

```
package filedef;
import java.io.File;
public class fileListInDirectory
{
   public static void main(String[] args)
   {
            // initialize File constructor
            File dir = new File("D:/Rahul");
            File[] files = dir.listFiles();
            System.out.println("Files in Directory:");
            for (File file : files)
             {
                System.out.println(file.getAbsolutePath());
            }
   }
```

```
}
```

Output:

Files in Directory:

D:\Rahul\data

D:\Rahul\file.txt

6.4 Using FilenameFilter:

FilenameFilter interface can be implemented to filter file names when File class and listFiles() method is used .

Write a Program to implement FilenameFilter Example:

```
package filedef;
import java.io.File;
import java.io.FilenameFilter;

public class FileNameFilter
{
   public static void main(String[] args)
   {
                String dir = "/Users/Siom/Downloads";
                String extension = ".doc";
                findFiles(dir, extension);
        }
        private static void findFiles(String dir, String extension)
      {
                File file = new File(dir);
                if (!file.exists())
                        System.out.println(dir + " Directory doesn't exists");
                File[] listFiles = file.listFiles(new MyFileNameFilter(extension));
```

```
        if (listFiles.length == 0)
    {
            System.out.println(dir + "doesn't have any file with
            extension " + extension);
        } else
    {
            for (File f: listFiles)
            System.out.println("File: " + dir + File.separator +
                                        f.getName());
        }
  }

  // FileNameFilter implementation
  public static class MyFileNameFilter implements FilenameFilter
{
        private String extension;

        public MyFileNameFilter(String extension)
    {
            this.extension = extension.toLowerCase();
        }

        @Override
        public boolean accept(File dir, String name)
    {
            return name.toLowerCase().endsWith(extension);
        }
  }
```

}

Output:

File: /Users/Siom/Downloads\!qhlogs.doc

File: /Users/Siom/Downloads\~$MS End Term paper.doc

6.5 The ListFiles( ) Alternative :

There is an alternative for list method discussed in Using FilenameFilter Interface In Java. list method it returns the names of the String array. But what if we want File objects. So in order to get File objects we use a method listFiles which returns File objects that match the specified criteria.

The signatures for listFiles() are shown below:

These methods return the file list as an array of File objects instead of strings.

listFiles( ) -this method works just like the list method only difference it returns the File objects.

listFiles(FilenameFilter FFObj)- it returns the File objects that ends with the specified extension.

listFiles(FileFilter FObj)- it returns the File objects as we specify. Here we can present a filter so that the File objects which passes the filter are accepted

Program:

```
package filedef;
import java.io.File;
import java.io.FilenameFilter;
import java.io.FileFilter;
public class ListFileAlternative {
   public static void main(String arg[])
   {
      File f = new File("D:\\"); // LINE A
      FilenameFilter fnf = new Extension(".mp4");
      File[] farray = f.listFiles(fnf); // LINE B
```

```
        for(int i =0; i < farray.length; i++)
        {
            System.out.println(farray[i]);  // LINE C
        }
        System.out.println("-----------------------");

        FileFilter ff = new CheckFilter();
        File ffarray[] = f.listFiles(ff); // LINE D

        for(int i =0; i < ffarray.length; i++)
        {
            System.out.println(ffarray[i]);
        }
    }
}
class Extension implements FilenameFilter
{
    String name;
    public Extension(String name)
    {
        this.name = name;
    }
    public boolean accept(File dir, String name)
    {
        return name.endsWith(this.name);
    }
}
class CheckFilter implements FileFilter
{
```

```
    public boolean accept(File pathname)
    {
        return pathname.toString().endsWith("jpg");
    }
}
```

Output:

D:\ilovepdf_pages-to-jpg

6.6 Creating Directories:

We can also create directories by using static method **createDirectory()** and **createDirectories()** to create directories by using **java.nio.Files** class.

Program:

```
package filedef;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class createDirecoryExample
{
    public static void main(String[] args)
    {
                Path path = Paths.get("D:/java7");
                Path subPath = Paths.get("D:/java7/createdir/data");
                try
            {
                        Path dirPath = Files.createDirectory(path);
                        Path subdirPath = Files.createDirectories(subPath);
                        System.out.println("Directory Path : "+dirPath);
```

```
                    System.out.println("Directory Including sub Directories
                                        Path   : "+subdirPath);
            }
        catch (IOException e)
        {
                    e.printStackTrace();
            }
    }
}
```

Output:

Directory Path : D:\java7

Directory Including sub Directories Path : D:\java7\createdir\data

6.7 Summary:

We study the various classes of java.io package that your programs can use to read and write data.

We study various constructor of file class

We create new file by using createNewFile() method.

Check file permissions by using canExecute(),canRead() and canWrite() method.

We check file already exist by using exist() method and delete file by using delete() method.

We study how to fetch absolute and canonical path by using getAbsolutePath() and getCanonicalPath() method

We create directories by using mkdir() and mkdirs() method

We can list out all files in directory by using listFiles() , filename filter and the listFiles alternative .

We can directories by using static method createDirctory() and createDirectories()

6.8 Exercise:

1) What are the constructors in File class?
2) What are the basic methods in File class?
3) How do you handle directories in Java?
4) How to create a File Object?
5) What is filename filter ?
6) What is listfiles alternatives?
7) Write a Program to check if a file or directory physically exist or not.
8) Write a Program to check file permission.
9) Write a Program to fetch absolute and canonical path of file.
10) Write a Program to create directories.

Chapter 7 Byte stream

7.0 Objectives

- ✓ learn how use the classes ObjectOutputStream and ObjectInputStream to read and write class objects with binary files
- ✓ To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way

7.1 Introduction

Stream: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)

it acts as a buffer between the data source and destination

7.2 The Stream Classes-

In Java, a stream is a path along which the data flows. Every stream has a source and a destination. We can build a complex file processing sequence using a series of simple stream operations.

Two fundamental types of streams are **Writing streams and Reading streams**. While an Writing streams writes data into a source(file) , an Reading streams is used to read data from a source(file).

Types of Streams

The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

- **Byte stream classes**
- **Character stream classes**

Text and Binary Formats of Data

There are two fundamentally different ways to store data. They are **text format** and **binary format**.

If data items are available in **text format**, we have to use the **Character stream classes** to process the input and output.

If the data items are made available in **binary format**, we have to use the **Byte stream classes**.

While the text format is convenient for humans, storage of data in binary format is more compact and more efficient.

Byte Stream Classes

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of byte stream classes: **input stream classes** and **output stream classes**.

7.3 Input Stream Classes

Input stream classes that are used to read bytes from file. Input stream classes are derived from super class known as **Inputstream** and a number of subclasses for supporting various input-related functions.

The super class InputStream is an **abstract** class, and, therefore, we cannot create instances of this class.

Rather, we must use the subclasses that inherit from this class.

7.4 Output Stream Classes

Output stream classes that are used to write bytes in to file. Output stream classes are derived from the base class **Outputstream** .

The OutputStream is an **abstract** class and therefore we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations.

7.5 FileInputStream

FileInputStream is byte streams that read data in binary format, exactly 8-bit bytes. They are descended from the abstract classes InputStream . **FileInputStream** class possesses the functionality of reading **one byte at a time** from a file.

FileInputStream Constructor:

FileInputStream(String name)- Creates a FileInputStream by opening a connection to an actual file, the file named by the path name **name** in the file system.

FileInputStream(File file)- Creates a FileInputStream by opening a connection to an actual file, the file named by the File object **file** in the file system.

FileInputStream Method:

int read()-Reads a byte of data from this input stream.

int available()-Returns the number of remaining bytes that can be read from the input stream.

void close()-Closes the stream and releases any system resources associated with it.

Program to demonstrate the FileInputStream class

```
package bytestreamclass;
import java.io.FileInputStream;
import java.io.IOException;
public class ByteStreamClass
{
   public static void main(String[] args)
   {
     FileInputStream fi = null;
      try
      {
         fi = new FileInputStream("data.da");
         int getSize = fi.available();
```

```
            System.out.println("File Size in Bytes : "+getSize);
            int i = 0;
            while(i<getSize)
            {
               System.out.println(fi.read());
               i++;
            }
         }
         catch(IOException io)
         {
            System.out.println("Exception : "+io);
          }
         finally
         {
            try
            {
               fi.close();
             }
            catch(IOException io)
            {
               System.out.println("Exception : "+io);
            }
         }
      }
}
```

Output:

File Size in Bytes : 6

40

60

80

100

120

7.6 FileOutputStream

FileOutputStream is byte streams that write data in binary format, exactly 8-bit bytes. They are descended from the abstract classes OutputStream

FileOutputStream class possesses the functionality of writing one byte at a time into a file.

FileOutputStream Constructors

FileOutputStream(String name)- Creates a file output stream to write to the file with the specified name.

FileOutputStream(File file)- Creates a file output stream to write to the file represented by the specified File object.

FileOutputStream Method

void write(int b)- This method used to write the specified byte to this file output stream.

void close()-This method used to close the stream.

Program to demonstrate fileouputstream class

```
package bytestreamclass;

import java.io.FileOutputStream;

import java.io.IOException;

public class fileOutputstream
```

```
{
    public static void main(String[] args)
    {
       FileOutputStream fo = null;
       byte by[] = {20,40,60,80,100,120};
       try
       {
          fo = new FileOutputStream("data.da");
          fo.write(by);
          System.out.println("write to file succeesful ");
       }
       catch(IOException io)
        {
          System.out.println("Exception : "+io);
        }
       finally
       {
         try
```

```
            {
              fo.close();
            }
            catch(IOException io)
              {
                  System.out.println("Exception : "+io);
              }
       }
    }
}
```

Output:

write to file succeesful

7.8 ByteArrayInputStream

ByteArrayInputStream class contains an internal buffer which is used to read byte array as stream. In this stream, the data is read from a byte array.

ByteArrayInputStream Constructors:

ByteArrayInputStream(byte[] buf)- Creates a new byte array input stream which uses **'buf'** as its buffer array.

ByteArrayInputStream Methods

int read()-Reads the next byte of data from the InputStream. Returns an int as the next byte of data. If it is the end of the file, then it returns -1.

int available()-Return the number of remaining bytes that can be read from the input stream.

long skip(long n)- Skip the **'n'** bytes of input from the input stream.

void reset()-Resets the stream to the most recent mark, or to the beginning of the string if it has never been marked.

Program Source

```
package bytestreamclass;
import java.io.ByteArrayInputStream;
import java.io.IOException;
public class bytearrayInputstream
{
   public static void main(String[] args) throws IOException
   {
      byte bya[] = {60,70,80,90,100,10,20,30,40,50,120,70,80,};
      ByteArrayInputStream bis = new ByteArrayInputStream(bya);
      System.out.println("avb : "+bis.available());
      bis.skip(5);
      System.out.println("avb : "+bis.available());

      byte byte1[] = new byte[5];
      bis.read(byte1);
      for(byte by : byte1)
      {
         System.out.print("["+by+"] ");
      }
      System.out.println();
      bis.reset();
      System.out.println("avb : "+bis.available());

      byte byte2[] = new byte[5];
      bis.read(byte2);
```

```
        for(byte by : byte2)
        {
            System.out.print("["+by+"] ");
        }
    }
}
```

Output:

```
avb : 13
avb : 8
[10] [20] [30] [40] [50]
avb : 13
[60] [70] [80] [90] [100]
```

7.9 ByteArrayOutputStream

The ByteArrayOutputStream class stream creates a buffer in memory and all the data sent to one or more stream is stored in the buffer. The buffer of ByteArrayOutputStream automatically grows according to data. In this stream, the data is written into a byte array.

You write your data to the ByteArrayOutputStream and when you are done you call the ByteArrayOutputStream's method toByteArray( ) to obtain all the written data in a byte array.

ByteArrayOutputStream Constructors

ByteArrayOutputStream()-Creates a new byte array output stream.

ByteArrayOutputStream(int size)- Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

ByteArrayOutputStream Methods

void write(int b)- Writes the specified byte(int to byte) to this byte array output stream.

void write(byte[] b)- Writes the specified byte array to this byte array output stream.

void write(byte[] b, int off, int len)- Writes '**len**' bytes from the specified byte array starting at offset '**off**' to this byte array output stream.

void writeTo(OutputStream out)- Writes the entire content of this stream to the specified stream argument.

byte[] toByteArray()-Creates a newly allocated Byte array. Its size would be the current size of the output stream and the contents of the buffer will be copied into it. Returns the current contents of the output stream as a byte array.

int size()-Returns the current size of the buffer.

void reset()-Resets the number of valid bytes of the byte array output stream to zero, so all the accumulated output in the stream will be discarded. The output stream can be used again, reusing the already allocated buffer space.

Program Source

```
package bytestreamclass;
import java.io.ByteArrayOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class bytearrayOutputstream
{
   public static void main(String[] args) throws IOException
   {
      ByteArrayOutputStream bos = new ByteArrayOutputStream(12);
      System.out.println("bos Size : "+bos.size());

      byte by1[] = {10,20,30,40,50};
      bos.write(by1);
      System.out.println("bos Size : "+bos.size());
```

```
        byte by2[] = {40,50,60,70,80,90,100,110,120};
        bos.write(by2, 2, 5);
        System.out.println("bos Size : "+bos.size());

        for(byte b : bos.toByteArray())
        {
            System.out.print("["+b+"] ");
        }
        System.out.println();
        try(FileOutputStream fo =new FileOutputStream("data.da"))
        {
            bos.writeTo(fo);
        }
        bos.reset();
        System.out.println("bos Size : "+bos.size());
        byte by3[] = {11,22,33,44,55};
        bos.write(by3);
        for(byte b : bos.toByteArray())
        {
            System.out.print("["+b+"] ");
        }
    }
}
```

Output:

```
bos Size : 0
bos Size : 5
bos Size : 10
[10] [20] [30] [40] [50] [60] [70] [80] [90] [100]
bos Size : 0
```

[11] [22] [33] [44] [55]

7.10 Filtered Byte Streams –

These are the wrapper classes for the InputStream and OutputStreams.

These streams add special features and improves the performance.

The filtered byte streams are FilterInputStream and FilterOutputStream.

FilterInputStream-

protected FilterInputStream(InputStream in)

Creates FilterInputStream object for the passed Inputstream

Important Methods :

int available()-Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.

abstract int read()-Reads the next byte of data from the input stream.

boolean markSupported()-Tests if this input stream supports the mark and reset methods.

Program:

```
package bytestreamclass;
import java.io.*;
public class filterInputstream
{
   public static void main(String[] args) throws IOException
   {
      File f = new File("data.da");
      if(f.exists())
         System.out.println("File exists.");
      else
         System.out.println("No file found.");
      FilterInputStream fis = new BufferedInputStream(new FileInputStream(f));
```

```
        int i;
        System.out.println("Available bytes : "+ fis.available());
        System.out.println("Mark Supported : " + fis.markSupported());
        while ( (i = fis.read()) != -1)
        {
           System.out.print((char)i);
        }
        fis.close();
    }
}
```

Output:

```
File exists.
Available bytes : 10
Mark Supported : true
  (2<FPZd
```

FilterOutputStream.

Important Methods :

void write(int b)- Writes the specified byte to this output stream.

void flush()-Flushes the output stream and forces any buffered output bytes to be written out to the stream.

Program:

```
package bytestreamclass;
import java.io.*;
public class filteroutputStream
{
    public static void main(String[] args) throws IOException
    {
```

```
        File data = new File("D:\\data.txt");
        FileOutputStream file = new FileOutputStream(data);
        FilterOutputStream filter = new FilterOutputStream(file);
        String s="Welcome to java Programming.";
        byte b[]=s.getBytes();
        filter.write(b);
        filter.flush();
        filter.close();
        file.close();
        System.out.println("Success...Welcome to Programming");
    }
}
```

OUTPUT:

Success... Welcome to Programming

7.11 Bufferred byte stream:

All the InputStreams and OutputStreams we have seen so for use unbuffered Streams. This means each read or write request is directly handled by the underlying Operating System. This will reduce the program efficiency. To reduce this kind of overhead, Java implements buffered I/O streams. Buffered input streams read data from a memory area known as buffer. We send request to OS only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and request are send to OS only when the buffer is full and there is no space further to perform write operation.

BufferedInputStream:

Commonly used constructors of BufferedInputStream

BufferedInputStream(InputStream in)-Creates a BufferedInputStream and saves its argument, the input stream in, for later use.

BufferedInputStream(InputStream in, int size)- Creates a BufferedInputStream with the specified buffer size, and saves its argument, the input stream in, for later use.

1. **BufferedInputStream(InputStream in)**

Creates a BufferedInputStream and saves its argument, the input stream in, for later use.

2**. BufferedInputStream(InputStream in, int size)**

Creates a BufferedInputStream with the specified buffer size, and saves its argument, the input stream in, for later use.

```
package bytestreamclass;
import java.io.*;
class IOTest
{
        public void readFile()
      {
              try
            {
               //creating FileInputStream object.
               FileInputStream fis =
                   new FileInputStream("D:\\data.txt");
                //Creating BufferedInputStream object.
               BufferedInputStream bis =
                           new BufferedInputStream(fis);
               int i;
               //read file.
               while((i=bis.read())!=-1)
                 {
                       System.out.print((char)i);
```

```
                }
            }
        catch (Exception e)
        {
            e.printStackTrace();
            }
    }
}
public class bufferedInputstream
{
   public static void main(String args[])
   {
            //creating IOTest object.
            IOTest obj = new IOTest();
            //method call.
            obj.readFile();
   }
}
```

Output:

Welcome to java Programming.

BufferedOutputStream:

Commonly used constructors of BufferedOutputStream:

BufferedOutputStream(OutputStream out)- Creates a new buffered output stream to write data to the specified underlying output stream.

BufferedOutputStream(OutputStream out, int size)- Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

```
package bytestreamclass;
import java.io.*;
class Test
{
        String str = "Hello Welcome to IO Class Demonstration";
        public void writeFile()
      {
              try
           {
                    //Creating FileOutputStream object.
                    //It will create a new file
                //before writing if not exist.
                   FileOutputStream fos =
                  new FileOutputStream("D:\\data.txt");
                    //Creating BufferedOutputStream object.
                    BufferedOutputStream bos =
                       new BufferedOutputStream(fos);
                    byte b[]=str.getBytes();
                    bos.write(b);
                    bos.flush();
                  //Close file after write operation.
                    bos.close();
                 System.out.println(" write successfully.");
              }
           catch (Exception e)
           {
                    e.printStackTrace();
              }
```

```
        }
}
public class bufferredoutputstream
{
   public static void main(String args[])
   {
                //Creating IOTest object.
                Test obj = new Test();
                //method call
                obj.writeFile();
        }
}
```

Output:

write successfully.

7.12 SequenceInputStream class

The SequenceInputStream class allows you to concatenate multiple InputStreams.

It reads data of streams one by one.

It starts out with an ordered collection of input streams .

Constructor :

SequenceInputStream(Enumeration e) : Initializes a newly created SequenceInputStream .

SequenceInputStream(InputStream s1, InputStream s2) : Initializes a newly created SequenceInputStream by remembering the two arguments, which will be read in order, first s1 and then s2.

Methods:

read() :Reads the next byte of data from this input stream.

read(byte[] b, int off, int len) : Reads up to len bytes of data from this input stream into an array of

available() : Returns an estimate of the number of bytes that can be read

The following is an example of SequenceInputStream class that implements some of the important methods.

Program:

```
//Java program to demonstrate SequenceInputStream
package bytestreamclass;
import java.io.*;
public class sequenceinputstream
{
  public static void main(String args[])throws Exception
  {
    FileInputStream input1=new FileInputStream("D:\\data.txt");
    FileInputStream input2=new FileInputStream("data.da");
    SequenceInputStream inst=new SequenceInputStream(input1, input2);
    int j;
    while((j=inst.read())!=-1)
    {
        System.out.print((char)j);
    }
  inst.close();
  input1.close();
  input2.close();
 }
}
```

Output:

Hello Welcome to IO Class Demonstration

-(2<FPZd

7.13 PrintStream

We can use print streams to output data into a file or to the console. **System.out** and **System.err** are instances of the **PrintStream** class.

The PrintStream class automatically flushes the data so there is no need to call flush() method.

Moreover, its methods don't throw IOException.

Print and Println Methods

PrintStream supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the PrintStream methods will call the object's **toString( )** method and then display the result.

Any preexisting file by the same name is destroyed. Once created, the PrintStream object directs all output to the specified file.

Program Source

```
package bytestreamclass;
import java.io.*;
public class printstream
{
   public static void main(String[] args) throws IOException
   {
      try(PrintStream ps = new PrintStream("D:\\data1.txt"))
      {
         ps.print("Rajiv Gandhi National Park ");
         ps.println("India");
         int km = 194;
         ps.println("It covers an area of "+km+" square kilometres");
         short el = 11;
```

```
            ps.println("Elephant : "+el);
            ps.printf("Lion     : %d"+System.lineSeparator(),new Short((short)28));
            ps.write(new byte[]{76,101,111,112,97,114,100});
            ps.print("  : 20");
            System.out.println("data write successfully");
        }
    }
}
```

Output:

data write successfully

within file data.txt data written in following fashion:

Rajiv Gandhi National Park India

It covers an area of 194 square kilometres

Elephant : 11

Lion   : 28

Leopard : 20

7.14 Summary:

We study how Programs use *byte streams* to perform input and output of 8-bit bytes.

We learn All byte stream classes are descended from InputStream and OutputStream.

There are many byte stream classes.

We focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

We also study the sequence input stream to concanate data from multiple input steam.

We can use print streams to output data into a file or to the console.

7.15 Exercise:

1) What are byte streams?

2) What is byte stream class in Java?

3) What are the super most classes of all streams?

4) What are FileInputStream and FileOutputStream? Demonstrate with program

5) What is the functionality of SequenceInputStream? Demonstrate with program

6) Write a program to implement printstream class?

7) What System.out.println()?

8) What is a IO stream?

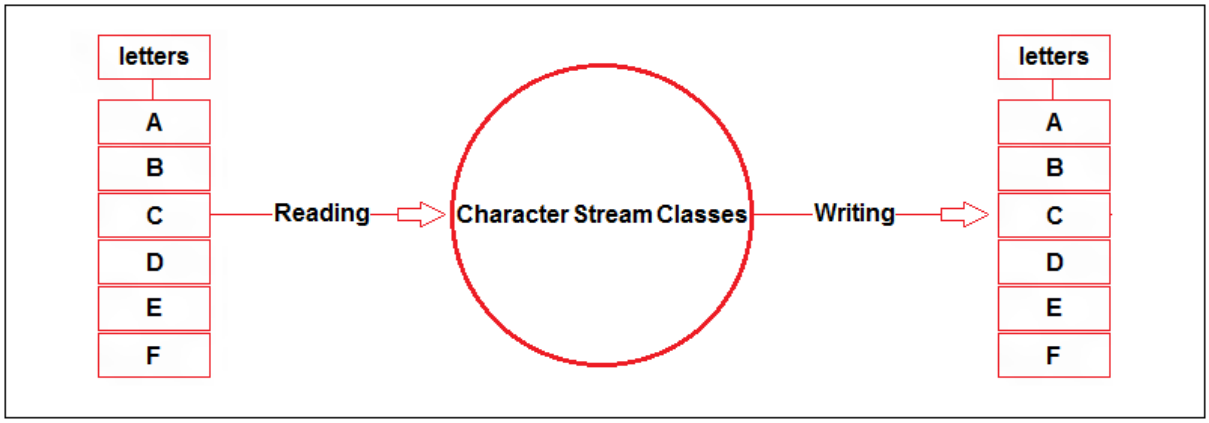
Chap. No. 8: Character Stream classes

8.0 Objectives:

- ✓ To become familiar with the concept of an Character stream
- ✓ To understand the difference between binary files and text files
- ✓ learn how to read and write data using fileReader and fileWriter class
- ✓ learn how to use different classes of character stream classes

8.1 Character Stream Classes

Character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes, namely, **reader stream classes** and **writer stream classes**.



8.2 Reader Stream Classes

Reader stream classes that are used to read characters include a super class known as **Reader** and a number of subclasses for supporting various input-related functions.

Reader stream classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters.

The Reader class contains methods that are identical to those available in the Input Stream class, except Reader is designed to handle characters.

Therefore, reader classes can perform all the functions implemented by the input stream classes.

8.3 Writer Stream Classes

Like output stream classes, the writer stream classes are designed to perform all output operations on files.

Only difference is that while output stream classes are designed to write bytes, the writer stream are designed to write character.

The **Writer** class is an **abstract** class which acts as a base class for all the other writer stream classes.

This base class provides support for all output operations by defining methods that are identical to those in Output stream class.

8.4 FileReader

FileReader class provides the functionality of reading character from a file.

FileReader class reads a single character at a time.

FileReader Constructors

FileReader(File file)- Creates a new FileReader, given the File to read from.

FileReader(String fileName)- Creates a new FileReader, given the name of the file to read from.

FileReader Methods

int read()-Reads a single character.

int read(char[] buf)- Reads characters into an array.

int read(char[] buf, int off, int len)- Reads characters into a portion of an array.

boolean ready()-Tells whether this stream is ready to be read.

long skip(long n)- Skips characters.

Program:

```
package characterstream;
import java.io.*;
public class CharacterStream
{
   public static void main(String[] args)
   {
      FileReader fr = null;
      try
      {
         fr = new FileReader("D://data.txt");
         int getc;
         while((getc=fr.read())!=-1)
         {
            System.out.print((char)getc+"");
         }
      }
      catch(IOException io)
      {
         System.out.println("Exception : "+io);
      }
      finally
      {
        try
         {
            fr.close();
         }catch(IOException io)
         {
            System.out.println("Exception : "+io);
         }
```

```
        }
    }
}
```

Output:

"Welcome to Character stream"

8.5 FileWriter

FileWriter class is an important subclass of the Writer class.

FileWriter class provides the functionality of writing characters into a file.

FileWriter Constructors:

FileWriter(String fileName)- Creates a FileWriter object given a file name.

FileWriter Method:

void write(int c)- This method used to write the single character.

void write(String str)- This method used to write the string.

Program Source

```
package characterstream;
import java.io.*;
public class filewriter
{
        public static void main(String[] args)  {
        FileWriter fw = null;
        try
        {
            fw = new FileWriter("D:\\data.txt");
            fw.write("Filewriter_class");
            System.out.println("Data written to file ");
        }
```

```
        catch(IOException io)
        {
            System.out.println("Exception : "+io);
        }
        finally
        {
          try
          {
              fw.close();
           }catch(IOException io)
           {
              System.out.println("Exception : "+io);
           }
        }
    }
}
```

Output:

Data written to file

8.6 Buffered Input/Output

Java also supports creation of buffers to store temporarily data that is read from or written to a stream.

The process is known as buffered I/O operation.

A buffer sits between the program and the source or destination and functions like a filter. Buffers can be created using

the BufferedWriter and BufferedReader classes.

Previous section we've seen so far use unbuffered

Input/Output(FileWriter, FileReader).

This means each read or write request is handled directly by the underlying OS.

This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. The BufferedReader and BufferedWriter classes provide internal character buffers.

Text that's written to a buffered writer is stored in the internal buffer and only written to the underlying writer when the buffer fills up or is flushed.

Likewise, reading text from a buffered reader may cause more characters to be read than were requested; the extra characters are stored in an internal buffer. Future reads first access characters from the internal buffer and only access the underlying reader when the buffer is emptied.

BufferedWriter Constructors

BufferedWriter(Writer out)- Creates a new buffered character-output stream that uses a default-sized(8192 character) output buffer.

BufferedWriter Methods

void write(int c)- This method used to write the single character.

void write(String str)- This method used to write the string.

void write([],buf)- This method used to write the string of array.

Program Source:

```
package characterstream;
import java.io.*;
public class bufferwriter
{
   public static void main(String[] args) {
      FileWriter fw = null;
```

```
    BufferedWriter bw = null;
    String nationalParks[] = {"1) Kaziranga National Park",
                              "2) Rajiv Gandhi National Park",
                              "3) Table Mountain National Park",
                              "4) Kgalagadi Transfrontier Park"};
    try{
        fw = new FileWriter("D:\\data2.txt");
        bw = new BufferedWriter(fw);
        int i = 0;
        while(i<nationalParks.length)
        {
            bw.write(nationalParks[i]);
            bw.newLine();
            i++;
        }
        System.out.println("write buffered data successfully");
    }catch(IOException io)
    {
        System.out.println("Exception : "+io);
    }finally
    {
        try{
            bw.close();
        }catch(IOException io)
        {
            System.out.println("Exception : "+io);
        }
    }
}
```

}

Output:

write buffered data successfully

8.7 BufferedReader:

The **BufferedReader** class possesses the functionality of reading **buffers of characters** from a file.

BufferedReader Constructors

BufferedReader(Reader in)- Creates a buffering character-input stream that uses a default-sized(8192 character) input buffer

BufferedReader Methods

int read()-Reads a single character.

int read(char[] buf)- Reads a single character.

int read(char[] buf)- Reads characters into an array.

Program Source

```
package characterstream;
import java.io.*;
public class bufferedreader
{
   public static void main(String[] args)
   {
      FileReader fr = null;
      BufferedReader br = null;
      try{
         fr = new FileReader("D:\\data2.txt");
         br = new BufferedReader(fr);
         String getline = null;
         while((getline=br.readLine())!=null)
         {
```

```
            System.out.println(getline);
          }
        }catch(IOException io)
        {
          System.out.println("Exception : "+io);
         }finally{
          try{
            br.close();
          }catch(IOException io)
          {
            System.out.println("Exception : "+io);
          }
        }
    }
}
```

Output:

1) Kaziranga National Park
2) Rajiv Gandhi National Park
3) Table Mountain National Park
4) Kgalagadi Transfrontier Park

8.8 PushbackReader

The PushbackReader can be used to push a single or multiple characters(that was previously read) back into the input stream so that it can be reread.

This is commonly used with parsers.

When a character indicating a new input token is read, it is pushed back into the input stream until the current input token is processed.

It is then reread when processing of the next input token is initiated.

PushbackReader Constructors

PushbackReader(Reader in)- Creates a pushback reader with a one-character pushback buffer.

PushbackReader Methods

int read()-Reads a single character underlying input stream.

int read(char[ ] cb)- Reads up to **cb.length** characters from the underlying input stream into an array of characters.

Program Source

```
package characterstream;
import java.io.*;
public class pushbackreader
{
   public static void main(String[] args) throws IOException
   {

      String str = "XDXXRXGGFXXU";
      try(CharArrayReader chr = new CharArrayReader(str.toCharArray());
         PushbackReader pr = new PushbackReader(chr)){
         int getc = 0;
         while((getc=pr.read())!=-1)
         {
            if(getc=='X')
            {
               if((getc=pr.read())=='X')
               {
                  System.out.print("(XX)");
               }else{
                  System.out.print("(X)");
                  pr.unread(getc);
```

```
                }
            }else
                System.out.print((char)getc);
        }
    }
  }
}
```

Output:

(X)D(XX)R(X)GGF(XX)U

8.9 PrintWriter

The **java.io.PrintWriter** class is a subclass of **java.io.Writer**.

PrintWriter is essentially a character-oriented version of **PrintStream**, that contains the familiar **print( )** and **println( )** methods from PrintStream.

The PrintWriter class implements the abstract write() methods from java.io.Writer.

The PrintWriter can send text either to an output stream or to another writer.

Difference Between PrintStream and PrintWriter

The main difference between PrintStream and PrintWriter is that PrintWriter does not contain methods for writing raw bytes, for which a program should use unencoded byte streams. The other, more minor difference is that automatic flushing is performed only when println( ) is invoked, not every time a newline character is seen.

PrintWriter Constructors

PrintWriter(File file)- Creates a new PrintWriter, without automatic line flushing, with the specified file.

Program Source

```
package characterstream;
import java.io.*;
```

```
public class printwriter
{
   public static void main(String[] args) throws IOException
   {
     try(PrintWriter ps = new PrintWriter("data3.txt"))
       {
          ps.print("Rolls Royce Motor Cars Limited established in ");
          ps.println(new Integer(1998));
          ps.println("Rolls Royce Top Models");
          ps.printf("%s : %d Crore"+System.lineSeparator(),"Rolls Royce
Phantom",10);
          ps.format("%s : %d Crore","Rolls Royce Ghost  ",6);
       }
       try(PrintWriter ps = new PrintWriter(System.out))
       {
          ps.print("Rolls Royce Motor Cars Limited established in ");
          ps.println(new Integer(1998));
          ps.println("Rolls Royce Top Models");
          ps.printf("%s : %d Crore"+System.lineSeparator(),"Rolls Royce
Phantom",10);
          ps.format("%s : %d Crore","Rolls Royce Ghost  ",6);
       }
   }
}
```

Output:

```
Rolls Royce Motor Cars Limited established in 1998
Rolls Royce Top Models
Rolls Royce Phantom : 10 Crore
Rolls Royce Ghost   : 6 Crore
```

8.10 CharArrayReader

CharArrayReader class contains an internal buffer that can be used as a character-input stream. It inherits Reader class.

CharArrayReader Constructors

CharArrayReader(char[] buf)- Creates a CharArrayReader from a specified character array.

CharArrayReader Methods

int read()-Reads a single character.

int read(char[ ] cb)- Reads up to **cb.length** characters from this stream into an array of characters.

int read(char[ ] cb, int off, int len)- Reads up to **len** characters from this stream into an array of characters.

void reset()-Resets the stream to the most recent mark, or to the beginning if it has never been marked.

long skip(long n)- Skips characters.

Program Source

```
package characterstream;
import java.io.*;
public class chararrayreader
{
   public static void main(String[] args) {
      String str = "ProgramPythonRprogrammingDatascience";
      try(CharArrayReader chr = new CharArrayReader(str.toCharArray())){
         chr.skip(14);
         char ca1[] = new char[4];
         chr.read(ca1, 0, 4);
         for(char c : ca1)
         {
            System.out.print(c);
```

```
            }
            chr.reset();
            char ca2[] = new char[7];
            chr.read(ca2, 0, 7);
            for(char c : ca2)
            {
               System.out.print(c);
            }
        }catch(IOException ex){

            System.out.println("An I/O Error Occurred");
        }
    }
}
```

Output:

progProgram

8.11 CharArrayWriter

The CharArrayWriter class creates a character buffer in memory and all the character sent to one or more file is stored in the buffer.

The buffer of CharArrayWriter automatically grows according to data.

In this stream, the data is written into a character array.

CharArrayWriter Constructors

CharArrayWriter()-Creates a new CharArrayWriter.

CharArrayWriter(int initialSize)- Creates a new CharArrayWriter with the specified initial size.

CharArrayWriter Methods

void write(int c)- Writes a character(int to char) to the buffer.

void write(char[] c)- Writes an array of characters to the buffer.

void write(char[] c, int off, int len)- Writes a portion of an array of characters to the buffer.

void write(String str)- Write a string to the buffer.

void writeTo(Writer out)- Writes the contents of the buffer to another character stream.

int size()-Returns the current size of the buffer.

void reset()-Resets the buffer so that you can use it again without throwing away the already allocated buffer

Program Source

```
package characterstream;
import java.io.*;
public class characterwriter
{
   public static void main(String[] args) throws IOException {
      CharArrayWriter cw = new CharArrayWriter(8);
      cw.write("Rah");
      cw.write(new char[]{'V','E','D'});
      System.out.println("Valid chars : "+cw.size());
      System.out.println(cw.toString());
      cw.write("distance");
     try(FileWriter fw = new FileWriter("char.txt")){
                cw.writeTo(fw);
      }
      System.out.println("Valid chars : "+cw.size());
      System.out.println(cw.toString());
      cw.reset();
      cw.write("52745878");
      System.out.println(cw.toString());
```

```
    }

}
```

Output:

```
Valid chars : 6
RahVED
Valid chars : 14
RahVEDdistance
52745878
```

8.12 RandomAccessFile

RandomAccessFile provides the facility to read and write data to a file.

RandomAccessFile class is part of Java IO.

While creating the instance of RandomAccessFile in java, we need to provide the mode to open the file.

For example, to open the file for read only mode we have to use “r” and for read-write operations we have to use “rw”.

we can read or write data from random access file at any position.

To get the current file pointer, you can call **getFilePointer()** method and to set the file pointer index, you can call **seek(int i)** method.

Program:

```
package bytestreamclass;
import java.io.*;
import java.io.IOException;
public class randomaccessfile
{
   public static void main(String[] args)
   {
```

```
            try
        {
                // file content is "ABCDEFGH"
                String filePath = "D:\\data.txt";
            System.out.println(new String(readCharsFromFile(filePath, 1, 5)));
                writeData(filePath, "Data", 5);
                //now file content is "ABCDData"
        }
        catch (IOException e)
        {
                e.printStackTrace();
            }
    }
private static void writeData(String filePath, String data, int seek) throws IOException
  {
            RandomAccessFile file = new RandomAccessFile(filePath, "rw");
            file.seek(seek);
            file.write(data.getBytes());
            file.close();
    }
    private static byte[] readCharsFromFile(String filePath, int seek, int chars) throws IOException
    {
            RandomAccessFile file = new RandomAccessFile(filePath, "r");
            file.seek(seek);
            byte[] bytes = new byte[chars];
            file.read(bytes);
            file.close();
```

```
            return bytes;
        }
}
```

Output:

ABCDE

8.13 Summary:

All character stream classes are descended from Reader and Writer.

There are character stream classes that specialize in file

I/O: FileReader and FileWriter.

We implement filereader and filewriter class to read and write data to file

We study bufferedreader and bufferedwriter class

We learn pushback and printwriter class of character stream

We focuses on characterArrayReader and characterArrayWriter class.

8.14 Exercise:

1) What is character stream?

2) Which is used for writing characters to a stream?

3) Which stream contains the classes which can work on character stream?

4) What is FileReader and FileWriter in Java?

5) Which class is used to read streams of characters from a file?

6) Write a program to implement filereader and filewriter class

7)Write a program to implement bufferedReader and bufferedWriter class

8) What is Random access file and how to implement it?

9) What is pushback reader and printwriter?

Chapter no 9 Algorithm Analysis and Array

9.0 Objectives:

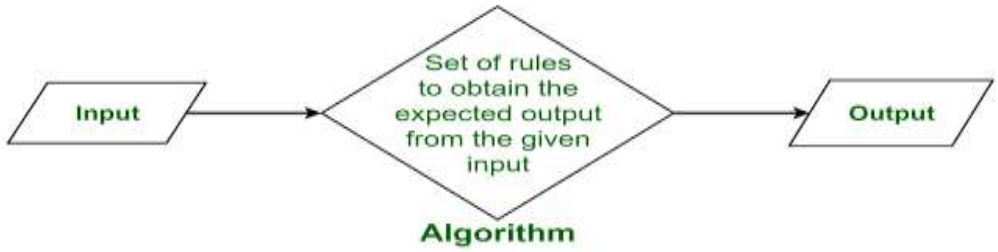
- ✓ To describe why an array is necessary in programming
- ✓ To learn the steps involved in using arrays: declaring array reference variables and creating arrays .
- ✓ To initialize the values in an array
- ✓ To copy contents from one array to another .
- ✓ To understand Sequential Search Algorithm.
- ✓ To understand how to make measure the performance of algorithm .

9.1 Complexity and analysis of algorithms:

9.1.1 Algorithm :

An **algorithm** is unambiguous finite step-by-step procedure which is given input to produce output.

Typically algorithms are used to solve problems such that for every input instance it produces the solution.



Describe: in natural language / pseudo-code / diagrams / etc.

Criteria to follow:

Input: Zero or more quantities (externally produced)

Output: One or more quantities

Definiteness: Clarity, precision of each instruction

Finiteness: The algorithm has to stop after a finite (may be very large) number of steps

Effectiveness: Each instruction has to be basic enough and feasible

9.1.2 Time and Space Complexity:

Time complexity of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the input.

Space complexity of an algorithm quantifies the amount of **space** or memory taken by an algorithm to run as a function of the length of the input.

Sometimes , We need to learn how to compare the performance of different algorithms and choose the best one to solve a particular problem.

While analyzing an algorithm, we mostly consider time complexity and space complexity Time and space complexity depends on lots of things like hardware, operating system, processors, etc.

However, we don't consider any of these factors while analyzing the algorithm.

We will only consider the execution time of an algorithm.

Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables
2. Program Instruction
3. Execution

S(P) = c + S(instance characteristics)

C is constant or fixed part and S is variable part of memory

how to compute space complexity

Example 1:

```
{
   int z = a + b + c;
   return(z);
```

```
}
```

In the above expression, variables a, b, c and z are all integer types,

hence they will take up 4 bytes each, so total memory requirement will be (4(4) + 4) = 20 bytes, additional 4 bytes is for **return value**.

And because this space requirement is fixed for the above example,

hence it is called **Constant Space Complexity**.

Example: 2

```
// n is the length of array a[]
int sum(int a[], int n)
{
        int x = 0;                    // 4 bytes for x
        for(int i = 0; i < n; i++)  // 4 bytes for i
        {
           x  = x + a[i];
        }
        return(x);
}
```

- In the above code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.

Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as **Linear Space Complexity.**

How is time complexity calculated?

So we can multiply or divide by a constant factor to get to the simplest expression. The most common metric for **calculating time complexity** is Big O notation.

9.1.3 Asymptotic Notations

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

1) Θ Notation: The theta notation bounds functions from above and below, so it defines exact asymptotic behavior.

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.

Example: consider the following expression.

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than $\Theta n^2)$ irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0).

The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

Big-O Notation

Definition: f(n) = O(g(n)) iff there are two positive constants c and n0 such that

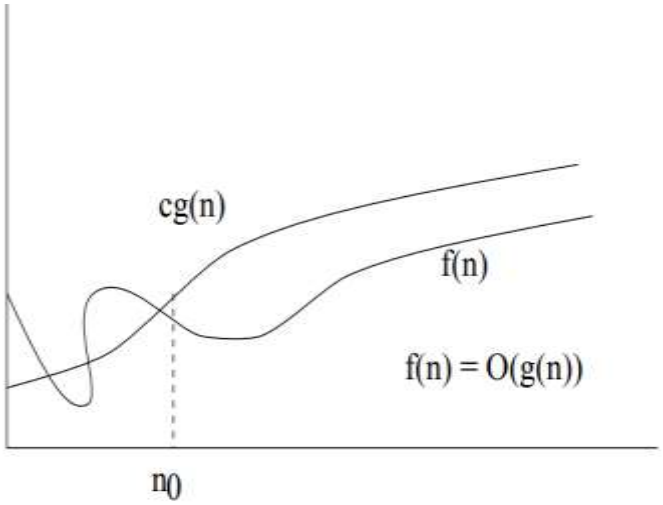
$|f(n)| \leq c\ |g(n)|$ for all $n \geq n0$

If f(n) is nonnegative, we can simplify the last condition to $0 \leq f(n) \leq c\ g(n)$ for all $n \geq n0$

We say that "f(n) is big-O of g(n)."

As n increases, f(n) grows no faster than g(n).

In other words, g(n) is an asymptotic upper bound on f(n).



Example: $n^2 + n = O(n^3)$

Proof:

Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$

Notice that if $n \geq 1$, $n \leq n^3$ is clear.

Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.

In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$.

This fact is used often in these types of proofs.

Therefore, $n^2 + n \leq n^3 + n^3 = 2n^3$

We have just shown that $n^2 + n \leq 2n^3$ for all $n \geq 1$

Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-O, with n0 = 1, and c = 2.)

Big-Ω notation

Definition: $f(n) = \Omega(g(n))$ iff there are two positive constants c and n0 such that

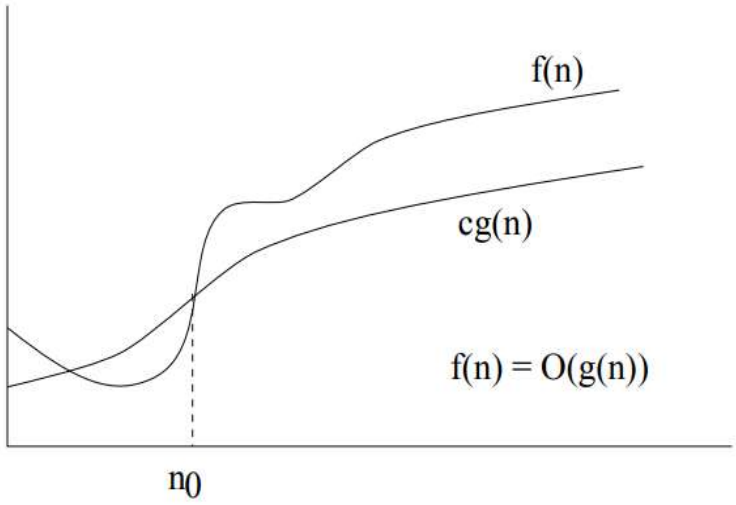
$$|f(n)| \geq c\,|g(n)| \text{ for all } n \geq n0$$

If f(n) is nonnegative, we can simplify the last condition to $0 \leq c\,g(n) \leq f(n)$ for all $n \geq n0$

We say that "f(n) is omega of g(n)."

As n increases, f(n) grows no slower than g(n).

In other words, g(n) is an asymptotic lower bound on f(n).



Example: $n^3 + 4n^2 = \Omega(n^2)$

Proof:

we have $f(n) = n^3 + 4n^2$, and $g(n) = n\,2$

It is not too hard to see that if $n \geq 0$, $n^3 \leq n^3 + 4n^2$

We have already seen that if $n \geq 1$, $n^2 \leq n^3$

Thus when $n \geq 1$, $n^2 \leq n^3 \leq n^3 + 4n^2$

Therefore, $1n^2 \leq n^3 + 4n^2$ for all $n \geq 1$

Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with n0 = 1, and c = 1.)

Big-Θ notation

Definition: f(n) = Θ(g(n)) iff there are three positive constants c1, c2 and n0 such that

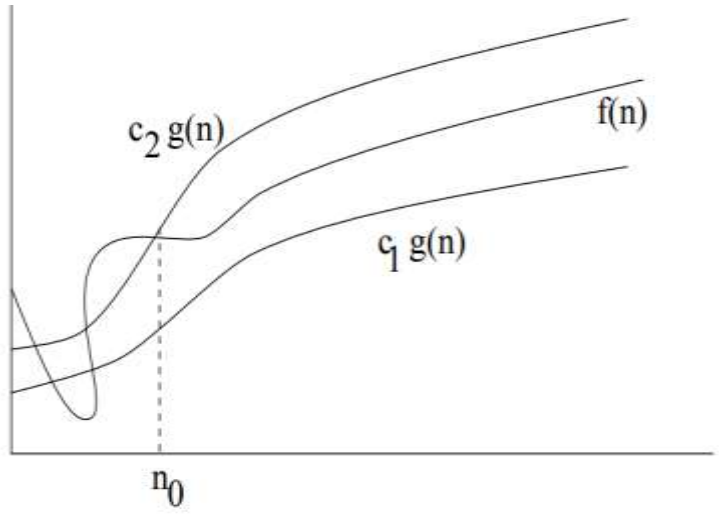
$c1|g(n)| \leq |f(n)| \leq c2|g(n)|$ for all $n \geq n0$

If f(n) is nonnegative, we can simplify the last condition to $0 \leq c1\ g(n) \leq f(n) \leq c2\ g(n)$ for all $n \geq n0$

We say that "f(n) is theta of g(n)."

As n increases, f(n) grows at the same rate as g(n).

In other words, g(n) is an asymptotically tight bound on f(n).



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof:

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$

Thus, when $n \geq 1$ $1n^2 \leq n^2 + 5n + 7 \leq 13n^2$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with n0 = 1, c1 = 1, and c2 = 13.)

9.1.4 Types of data structures:

Data structure is a way of storing and organizing data. Data structure provide a way to process and store data efficiently.

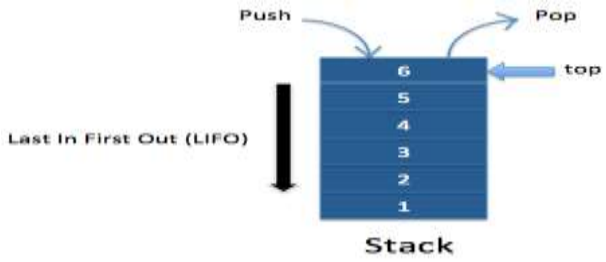
Array

Array is data structure which stores fixed number of similar elements.

Array can store primitive data types as well as object but it should be of same kind.

Stack

Stack is abstract data type which depicts Last in first out (LIFO) behavior.



Queue

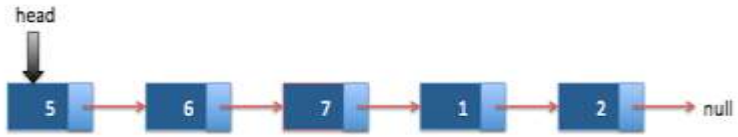
Queue is abstract data type which depicts First in first out (FIFO) behavior.

LinkedList

Singly **LinkedList**

In singly linked list, Node has data and pointer to next node.

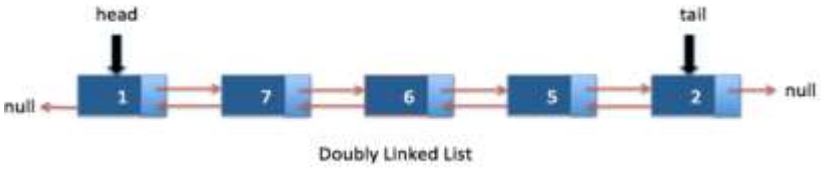
It does not have pointer to the previous node. Last node ‘s next points to null, so you can iterate over linked list by using this condition.



Doubly **LinkedList**

In doubly linked list, Node has data and pointers to next node and previous node.

You can iterate over linkedlist either in forward or backward direction as it has pointers to prev node and next node. This is one of most used data structures in java.



9.2 Arrays

Definition: An array is group of like-typed variables that are referred to by a common name.

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).

9.2.1 Array properties:

1) Unlike C and C++ array is an object in Java.

2) The length attribute of the array gives the length of an array, this is different from the length() method of String.

3) The length of an array is fixed and cannot be changed once created. The only way to increase or decrease length is to create a new array and copy contents of an old array to a new array.

4) You can access elements of an array using the index, which is non-negative integer value e.g. a[1] will give you second element.

5) Array index starts at zero and ranges till length -1.

6) Unlike C and C++ Java performs bound check while accessing array elements. Trying to access an invalid index in the array will throw java.lang.ArrayIndexOutOfBoundsException.

7) An array is typed in Java, you cannot store Integer in an array of String in Java. It will throw ArrayStoreException at runtime.

8) You can create an array of both primitive and reference types in Java.

9) Array elements are stored in the contiguous memory location.

10) The array is the backbone of many useful collection classes in Java e.g. ArrayList and HashMap both are backed by an array.

9.2.2 Duplicating of array:

Array in java can be copied to another array using the following ways.

1) Using variable assignment.

2) Create a new array of the same length and copy each element.
3) Use the clone method of the array. Clone methods create a new array of the same size.
4) Use System.arraycopy() method. arraycopy can be used to copy a subset of an array.

```
package arrays;
public class arraycopy
{
    public static void main(String args[])
    {
        //Scenario 1: Copy array using assignment
        int[] a = {10, 20, 30};
        int[] b = a;
        System.out.println("Scenario 1: ");
        System.out.print("Array a: ");
        printArray(a);
        System.out.print("Array b: ");
        printArray(b);
        //Scenario 2: Copy array by iterating
        int[] c = {11, 22, 33};       int[] d = new int[c.length];
        for (int i = 0; i < d.length; i++)
        {
           d[i] = c[i];
        }
        System.out.println("Scenario 2: ");
        System.out.print("Array c: ");
        printArray(c);
        System.out.print("Array d: ");
        printArray(d);
        //Scenario 3: Copy array using clone
        int[] e = {14, 24, 34};
        int[] f = e.clone();
```

```
        System.out.println("Scenario 3: ");
        System.out.print("Array e: ");
        printArray(e);
        System.out.print("Array f: ");
        printArray(f);
        //Scenario 4: Copy array using arraycopy
         int[] g = {15, 25, 35};
         int[] h = new int[g.length];
         System.arraycopy(g, 0, h, 0, h.length);
         System.out.println("Scenario 4: ");
         System.out.print("Array g: ");
         printArray(g);
         System.out.print("Array h: ");
         printArray(h);
    }
    public static void printArray(int[] a)
    {
        System.out.print("[ ");
        for (int i = 0; i < a.length; i++)
        {
         System.out.print(a[i] + " ");
        }
        System.out.println("]");
     }
}
```

Output

Scenario 1:

Array a: [10 20 30]

Array b: [10 20 30]

Scenario 2:

Array c: [11 22 33]

Array d: [11 22 33]

Scenario 3:

Array e: [14 24 34]

Array f: [14 24 34]

Scenario 4:

Array g: [15 25 35]

Array h: [15 25 35]

9.2.3 Linear or Sequential Search Algorithm

Linear or sequential search algorithm is a method for finding a target value within a list.

It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed.

The worst case is when the value is not in the list

Program:

```
package arrays;
public class linearsearch
{
```

```
 public static int linearSearch(int[] inputArray,int key)
        {
        for(int i=0;i<inputArray.length;i++)
        {
                if(inputArray[i] == key)
                {
                   return i;
                }
            }
            return -1;
        }
        public static void main(String args[])
        {
                int[] arr1 = {5,9,10,2,90,4};
                int key = 10;
                int result = linearSearch(arr1,key);
                System.out.println( (result != -1) ? "key:- "+key+" found at index:-
"+result : "Key "+key+ " not found");
                //try with another key
                key = 15;
                result = linearSearch(arr1,key);
                System.out.println( (result != -1) ? "key:- "+key+" found at index:-
"+result :       "Key "+key+ " not found");
        }
}
```

Output

key:- 10 found at index:- 2

Key 15 not found

9.2.4 Binary Search Algorithm

Binary search algorithm requires already sorted collection.

Binary search, is a search algorithm that finds the position of a target value within a sorted array.

Binary search compares the target value to the middle element of the array. If they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful or the remaining half is empty.

Once the target element found or collection is empty, then the search is over.

Program:

```
package arrays;
public class binarySearch
{
   public static int binarySearch(int[] inputArray,int key)
   {
        int start = 0;
      int end = inputArray.length - 1;
      while (start <= end)
      {
         int mid = (start + end) / 2;
         if (key == inputArray[mid])
         {
              return mid;
```

```
            }
            if (key < inputArray[mid])
            {
                end = mid - 1;
            }
          else
          {
              start = mid + 1;
            }
         }
       return -1;
   }
public static void main(String args[])
{
          int[] arr1 = {1,5,20,35,50,65,70};
          int key = 200;
          int result = binarySearch(arr1,key);
          System.out.println( (result != -1) ? "key:- "+key+" found at index:- "+result
: "Key "+key+ " not found");
          //try with another key
          key = 70;
          result = binarySearch(arr1,key);
          System.out.println( (result != -1) ? "key:- "+key+" found at index:- "+result
: "Key "+key+ " not found");

   }
```

}

Output

Key 200 not found

key:- 70 found at index:- 6

9.3 Summary:

We study basics of algorithm like definition, analysis of algorithm by using time and space complexity.

There are three asymptotic notations which is used to calculate time complexity of algorithm

Array is a collection of similar elements; we can make copy of array creating another array.

We study sequential search algorithm and binary search algorithms

9.4 Exercise:

1) What is algorithm?

2) Define the term Time complexity and Space complexity with examples

3) How to create array?

4) Write a program to duplicate the elements of array.

5) Explain different types of Asymptotic notations with examples

6) Write a program to implement sequential search algorithm

7) Write a program to implement binary search algorithm

Chapter no 10 Stack and Queue

10.0 Objectives:

- ✓ To understand the abstract data types stack, queue.
- ✓ To understand the implementations of stack using indexed and linked representation.
- ✓ To learn the applications of a stack
- ✓ To understand the indexed implementation of queue.
- ✓ To learn applications of queue

10.1 Stack

Definition:

A stack is a linear data structure which follows the LIFO (last-in first-out) principle. That means the objects can be inserted or removed only at one end of it also called as top.

Stack is a recursive data structure having pointer to its top element.

Examples of stack: Stack of Books

5.1.1 Stack Operations:

push- pushes an item onto the top of the stack i.e. you can add element by using push operation.

pop- removes the object at the top of the stack and returns that object from the function.

The stack size will be decremented by one.

peek : Look all the elements of stack without removing them.

isEmpty- Tests if the stack is empty or not.

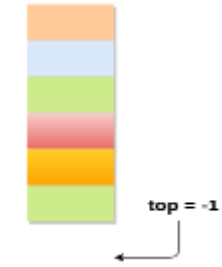
isFull - Tests if the stack is full or not.

Size- Returns the number of elements present in the stack.

How the stack grows?

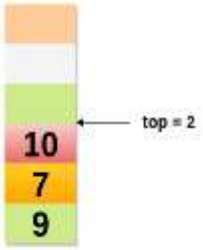
Case 1 : Stack is empty-

The stack is called empty if it doesn't contain any element inside it. At this stage, the value of variable top is -1.



Case 2 : Stack is not empty

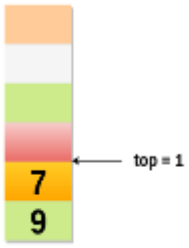
Value of top will get increased by 1 every time when we add any element to the stack. In the following stack, After adding first element, top = 2.



Case 3 : Deletion of an element

Value of top will get decreased by 1 whenever an element is deleted from the stack.

In the following stack, after deleting 10 from the stack, top = 1.



Top and its value :

Top position	**stack**
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

10.1.2 Indexed Implementation of stack- Stack can easily be implemented as array. As follows:

Programs

```
package arrays;
public class Stack
{
   private int arr[];
   private int top;
```

```
private int capacity;
// Constructor to initialize stack
Stack(int size)
{
    arr = new int[size];
    capacity = size;
    top = -1;
}
// Utility function to add an element x in the stack
public void push(int x)
{
    if (isFull())
    {
     System.out.println("OverFlow\nProgram Terminated\n");
     System.exit(1);
    }
    System.out.println("Inserted element " + x);
    arr[++top] = x;
}
// Utility function to pop top element from the stack
public int pop()
{
    // check for stack underflow
    if (isEmpty())
    {
     System.out.println("UnderFlow\nProgram Terminated");
     System.exit(1);
```

```
        }
        System.out.println("Removed element " + peek());
        // decrease stack size by 1 and (optionally) return the popped element
        return arr[top--];
    }
       // Utility function to return top element in a stack
    public int peek()
    {
        if (!isEmpty())
          return arr[top];
        else
          System.exit(1);
        return -1;
    }
    // Utility function to return the size of the stack
    public int size()
    {
        return top + 1;
    }
// Utility function to check if the stack is empty or not
    public Boolean isEmpty()
    {
        return top == -1;    // or return size() == 0;
    }
// Utility function to check if the stack is full or not
    public Boolean isFull()
    {
```

```
        return top == capacity - 1;         // or return size() == capacity;
    }
    public static void main (String[] args)
    {
        Stack stack = new Stack(3);
        stack.push(10);             // Inserting 10 in the stack
        stack.push(20);             // Inserting 20 in the stack
        stack.pop();          // removing the top 2
        stack.pop();          // removing the top 1
        stack.push(30);             // Inserting 30 in the stack
        System.out.println("Top element is: " + stack.peek());
        System.out.println("Stack size is " + stack.size());
        stack.pop();          // removing the top 3
        // check if stack is empty
        if (stack.isEmpty())
                System.out.println("Stack Is Empty");
        else
                System.out.println("Stack Is Not Empty");
        }
}
```

Output:

```
Removed element 10
Inserted element 30
Top element is: 30
Stack size is 1
Removed element 30
Stack Is Empty
```

10.1.3 Stack implementation using linked list-

A stack can be easily implemented through a linked list. In linked list implementation, a stack is then a pointer to the "head" of the list where pushing and popping items happens at the head of the list, with perhaps a counter to keep track of the size of the list.

The advantage of using linked list over arrays is that it is possible to implement a stack that can grow or shrink as much as needed.

Using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated, so overflow is not possible unless memory is exhausted.

Program:

```
class Node
{
        int data;       // integer data
        Node next;   // pointer to the next node
};
class Stack
{
        private Node top;
        public Stack()
      {
                this.top = null;
        }
        // Utility function to add an element x in the stack
        public void push(int x) // insert at the beginning
        {
```

```
        // Allocate the new node in the heap
        Node node = new Node();
        // check if stack (heap) is full. Then inserting an element would
        // lead to stack overflow
        if (node == null)
        {
                System.out.print("\nHeap Overflow");
                return;
        }
        System.out.println("Inserted element " + x);
        // set the data in allocated node
        node.data = x;
        // Set the .next pointer of the new node to point to the current
        // top node of the list
        node.next = top;
        // update top pointer
        top = node;
}

// Utility function to check if the stack is empty or not
public boolean isEmpty()
{
        return top == null;
}

// Utility function to return top element in a stack
public int peek()
```

```
    {
        // check for empty stack
        if (!isEmpty())
        {
            return top.data;
        }
        else
        {
            System.out.println("Stack is empty");
            return -1;
        }
    }
    // Utility function to pop top element from the stack
    public void pop() // remove at the beginning
    {
        // check for stack underflow
        if (top == null)
        {
            System.out.print("\nStack Underflow");
            return;
        }
        System.out.println("Removed element " + peek());
        // update the top pointer to point to the next node
        top = (top).next;
    }
}
public class StackImpl
```

```
{
   public static void main(String[] args)
        {
              Stack stack = new Stack();
              stack.push(11);
              stack.push(22);
              stack.push(33);
              System.out.println("Top element is " +  stack.peek());
              stack.pop();
              stack.pop();
              stack.pop();
              if (stack.isEmpty())
            {
                    System.out.print("Stack is empty");
              }
            else
            {
                    System.out.print("Stack is not empty");
              }
        }

}
```

Output:

Inserted element 22

Inserted element 33

Top element is 33

Removed element 33

Removed element 22

Removed element 11

Stack is empty

10.1.4 Applications of stack - Reverse a stack using recursion

Program:

```
import java.util.Stack;
class StackRecursion
{
   // using Stack class for
   // stack implementation
   static Stack<Character> st = new Stack<>();
   // Below is a recursive function
   // that inserts an element
   // at the bottom of a stack.
   static void insert_at_bottom(char x)
   {
        if(st.isEmpty())
          st.push(x);
       else
       {
          // All items are held in Function
          // Call Stack until we reach end
          // of the stack. When the stack becomes
          // empty, the st.size() becomes 0, the
          // above if part is executed and
```

```
        // the item is inserted at the bottom
        char a = st.peek();
        st.pop();
        insert_at_bottom(x);
        // push allthe items held
        // in Function Call Stack
        // once the item is inserted
        // at the bottom
        st.push(a);
    }
}
 // Below is the function that
// reverses the given stack using
// insert_at_bottom()
static void reverse()
{
    if(st.size() > 0)
    {
       // Hold all items in Function
        // Call Stack until we
        // reach end of the stack
        char x = st.peek();
        st.pop();
        reverse();
        // Insert all the items held
        // in Function Call Stack
        // one by one from the bottom
```

```
            // to top. Every item is
            // inserted at the bottom
            insert_at_bottom(x);
        }
    }
     public static void main(String[] args)
    {
        // push elements into
        // the stack
        st.push('5');
        st.push('7');
        st.push('9');
        st.push('3');
        System.out.println("Original Stack");
        System.out.println(st);
       // function to reverse
        // the stack
        reverse();
        System.out.println("Reversing Stack");
        System.out.println(st);
    }
}
```

Output:

Original Stack

[5, 7, 9, 3]

Reversing Stack

[3, 9, 7, 5]

10.2 Queue:

A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at another end (front).

The first element to be inserted is the first one to be deleted. Hence, it is called **First in First out (FIFO) or Last in Last out (LILO)** list.

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In a queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Examples:

A real-world example of a queue can be a single-lane one-way road, where the vehicle enters first, exits first.

More real-world examples can be seen as queues at the ticket windows and bus-stops.

10.2.1 Queue Operations-

enqueue() − When an element is inserted in a queue, the concept is called **enqueue.**

EnQueuing an element in a full queue is called **overflow** .

Queues maintain two data pointers, front, and rear.

enqueue 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 a 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.

dequeue()- When an element is removed from the queue, the concept is called **dequeue.**

DeQueueing an empty queue is called **underflow** .

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 the front is pointing.

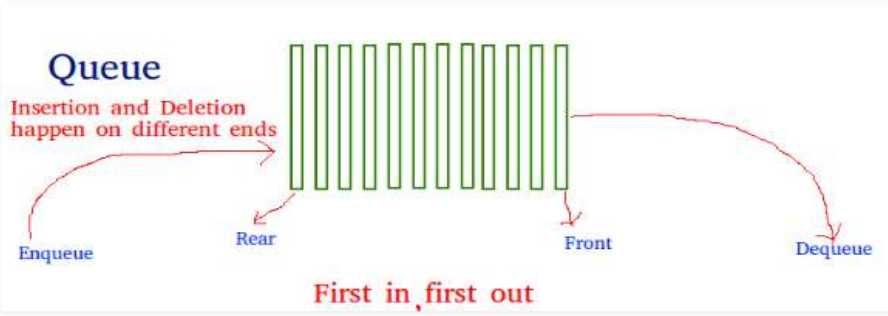
Step 4 − Increment front pointer to point to the next available data element.

Step 5 − Return success.

peek() − Gets the element at the front of the queue without removing it.

isFull() − Checks if the queue is full.

isEmpty() − Checks if the queue is empty.



10.2.2 Indexed Implementation of queue:

For implementing queue, we need to keep track of two indices, front and rear.

We enqueue an item at the rear and dequeue an item from front.

If we simply increment front and rear indices, then there may be problems, front may reach end of the array.

//program for indexed implementation of queue

```
class Queue
{
```

```
  int front, rear, size;
  int capacity;
  int array[];
  public Queue(int capacity)
{
        this.capacity = capacity;
        front = this.size = 0;
        rear = capacity - 1;
        array = new int[this.capacity];
  }
  // Queue is full when size becomes equal to
  // the capacity
  boolean isFull(Queue queue)
  { return (queue.size == queue.capacity);
  }
  // Queue is empty when size is 0
  boolean isEmpty(Queue queue)
  { return (queue.size == 0); }
  // Method to add an item to the queue.
  // It changes rear and size
  void enqueue( int item)
  {
        if (isFull(this))
              return;
        this.rear = (this.rear + 1)%this.capacity;
        this.array[this.rear] = item;
        this.size = this.size + 1;
```

```
        System.out.println(item+ " item enqueued to queue");
}
// Method to remove an item from queue.
// It changes front and size
int dequeue()
{
        if (isEmpty(this))
                return Integer.MIN_VALUE;
        int item = this.array[this.front];
        this.front = (this.front + 1)%this.capacity;
        this.size = this.size - 1;
        return item;
}
// Method to get front of queue
int front()
{
        if (isEmpty(this))
                return Integer.MIN_VALUE;

        return this.array[this.front];
}
// Method to get rear of queue
int rear()
{
        if (isEmpty(this))
                return Integer.MIN_VALUE;
```

```
            return this.array[this.rear];
        }
}
 public class QueueImpl
{
        public static void main(String[] args)
        {
            Queue queue = new Queue(1000);
                queue.enqueue(100);
            queue.enqueue(200);
            queue.enqueue(300);
            queue.enqueue(400);
            System.out.println(queue.dequeue() +
                                " dequeued from queue\n");
            System.out.println("Front end item is " +
                                                queue.front());
            System.out.println("Rear end item is " +
                                                    queue.rear());
        }
}
```

Output:

Front end item is 200

Rear end item is 400

10.2.3 Applications Queue:

Multiprogramming.

Asynchronous data transfer (file IO, pipes, sockets).

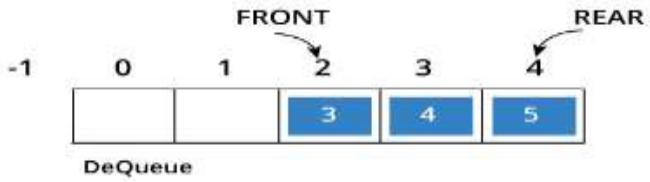
Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).

Determining the number of cashiers to have at a supermarket.

Simulation of real-world queues such as lines at a ticket counter, or any other first come the first-served scenario requires a queue.

10.3 Circular Queue-

Circular queue avoids the wastage of space in a regular queue implementation using arrays.

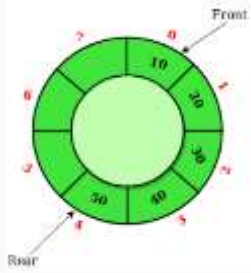


As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.

The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

How Circular Queue Works?

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.



10.3.1 Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue.

In a circular queue, the new element is always inserted at rear position.

The enQueue() function takes one integer value as parameter and inserts that value into the circular queue.

We can use the following steps to insert an element into the circular queue...

Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.

Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0

5.3.2 Deleting a value from Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue.

In a circular queue, the element is always deleted from front position.

The deQueue() function doesn't take any value as parameter.

We can use the following steps to delete an element from the circular queue...

Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function. •

Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

10.4 Summary:

Stack is linear data structure , elements of stack accessed in first in last out (FILO) fashion

Stack provide push and pop operations

Queue is linear data structure , elements of stack accessed in first in first out (FIFO) fashion

Queue provide enqueue and dequeue operations

Stack implemented by using indexed and linked representation

Queue implemented by using indexed representation

We also study the circular queue.

10.5 Exercise

1) What is Stack?

2) Explain various operations of stack

3) Write a program show indexed implementation of stack

4) Write a program to show linked implementation of stack

5) What is Queue?

6) Explain various operations of Queue

7) Write a program show indexed implementation of Queue

8) What is circular Queue? Explain it’s operations.

Chapter 11 Linked List

Objectives:-

- ✓ **Linked Lists - representation of linked list**
- ✓ **Traversing**
- ✓ **Searching**
- ✓ **Insertion**
- ✓ **Deletion**
- ✓ **Doubly linked list**

11.1 **Linked List: Introduction**

One of the disadvantages of static data structure like array, stack and queue is that it has limited and fixed size which might result in wastage of memory and may have reduced efficiency. Linked List has a great scope in building data structure, as it doesn't have a size limit and we can go on adding new nodes and increasing the size of the list to any degree.

Definition: -

Linked List is a dynamic data structure which can be defined as collection of objects called nodes that are randomly stored in the memory.

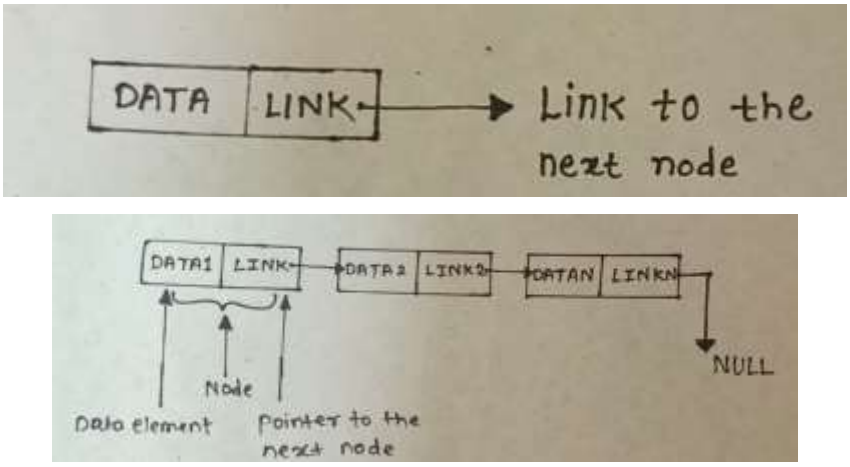


Fig: -11.1 Linked List Node

A node contains two fields i.e. data stored at that particular address and the pointer (reference) which contains the address of the next node in the memory .The last node of the list contains pointer to the null.

Example:-

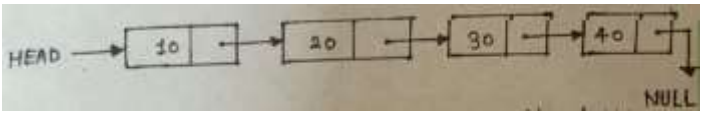


Fig 11.2 Linked list with 4 values 10,20,30,40

In above example there is a start address (HEAD) i.e node 1 which points to 10 node2 which points to 20 node 3 points to 30 and node 4 in turn point to nothing i.e NULL.

- In linked lists a linear collection of data items is called nodes, where the linear order is given by means of links or pointers or references in java .
- A link or reference actually is the address (memory location) of the successive element Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be preserved.
- An element in a linked list is specially termed s node, which can be viewed as shown in Fig 11.2 A node consists of two field named DATA (to store the actual information or data value) and LINK (to point to the next node)
- Linked list data structure is used for sorting data in the term of list . A linked list consist of a sequence of data record such that in each there is a record there is a field that contains a reference (i.e. ,a link) to the next record in the sequence.
- A node in a linked list is usually a class in Java and can be declared as follows:

```
public class Linklist {          // create an outer class  for
                                 // singly Linked list
    class Node{                   //create the inner class
        int data;
        Node link;
          public Node (int data) {     // initialize the data
and link
            this.data = data;
            this.link = null;
        }
    } //end of inner class
} // end of outer class
```

- It Represents the head and tail of the singly linked list
- Each node has two parts DATA (contains data element) and LINK (contains the address of the next node).
- The LINK part of the last node is of the list contains a NULL value indicating the end of the list. The beginning of the list is indicated with the help of a special pointer called FIRST. Similarly, the end of the list can be indicated by a pointer called LAST.

Advantages of Linked List:

1. Dynamic Data Structure: Unlike array, linked list can grow or shrink during the execution of a program.
2. Efficient Memory Utilization: Here memory is not defined (Pre-allocated). Memory is allocated whenever it is required. And it is de-allocated (removed) when it is no longer needed.

3. Insertion and Deletions are Easier and Efficient: Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Complex Application: Many complex applications can be easily carried out with linked lists.

Disadvantages of Linked List:

1. More Memory Required: A linked list element requires more memory space because it also has to store address of next element in the list.
2. Time Consuming: Access to an arbitrary data item is little bit cumbersome and also time consuming.
3. Difficult to Access Elements: Accessing an element is little difficult in linked list than array as there is no index associated with each element. Thus, to access a particular element it is mandatory to traverse all elements before it.

Terminology

- Basic terms used in linked list data structure are:
- Node: Each item in the linked list is called a node and contains two fields and information field (INFO\DATA) and a next address (LINK\NEXT) field.
- Information: The information field holds the actual element or data on the list.
- Address: A link or pointer actually is an address (memory location) of the subsequent element.
- Link: The field contains the address of next node in the list.
- Null pointer: The linked field of the last node contains Null rather than a valid address. Null pointer is a pointer containing o address. p=NULL; The statements will set the pointer to NULL or 0 address.
- Empty list: If the nodes are not present in a linked list, then it is called an empty linked list or simply empty list. It is also called the null list .The value of the external pointer will be zero for an empty list.

TYPES OF LINKED LIST

- A list is an order list, which consist of different data items connected by means of a link or pointer this type of list is also called a linked list.
- A linked list is defined as an ordered collection of finite homogeneous data elements called as nodes where the linear order is maintained by means of pointers or links or references."

- Types of linked list are given below:

I. Singly Linked List:

- The way to represent a linear list is to expand each node to contain a link or pointer to the next node .This representation is called a one-way sequence or singly linked list.
- In this type of linked list two nodes are linked with each other in sequential linear manner .In the singly link list each node have two fields one for data and other is for link reference of the next node . The last node's link field points to NULL.

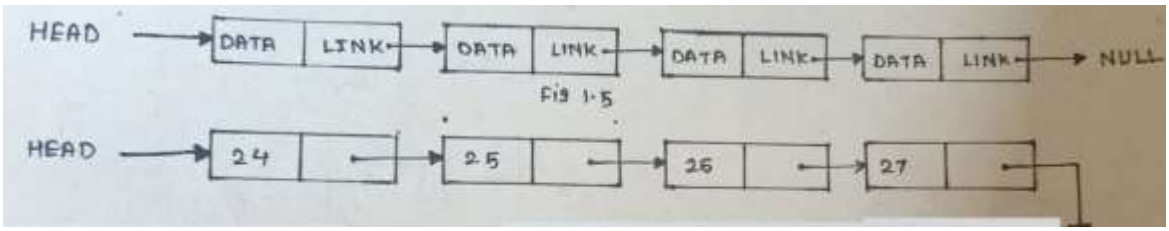


Fig 11.3 Singly Linked list with 4 values

Advantages of singly linked list:

1 .Accessibility of a node in the forward direction is easier.
2 Insertion and deletion of nodes are easier.

Disadvantages of singly linked list:

1 Accessing the preceding node of a current node is not possible as there is no backwater traversal
2 Accessing a node is time consuming.

11.2 Operations on singly linked list:

1. Create: - creating a linked list with data or element into it.
2. Traversing: - moving from the first node (HEAD/START) to last (TAIL/END) and displaying the elements.
3. Insertion: It has 3 possibilities either to insert from beginning, middle or end.
4. Searching: - traversing through the list and searching the key element in the list.
5. Deletion:-removing the element from the linked list either from beginning, middle and end.

1. Algorithm: Create and Insert
Input:- node (start, data, link)
Output:-nodes with data and links

- Create a class Node which has two attributes: data and next.
- Create another class which has two attributes: head and tail.
  addNode() will add a new node to the list:
  Create a new node.
  It first checks, whether the head is equal to null which means the list is empty.

If the list is empty, both head and tail will point to the newly added node.
If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

Program:- To create and display the node.
Considering the above program of link list we will create a link list with the first operation.

```
public void createNode(int data) {
        //Create a new node
        Node newNode = new Node(data);

        //Checks if the list is empty
        if(head == null) {
            //If list is empty, both head and tail will refer to
new node
            head = newNode;
            tail = newNode;
        }
        else {
            //newNode will be added after tail such that tail's
next will refer to newNode
            tail.link = newNode;
            //newNode will become new tail of the list
            tail = newNode;
        }
    }
```

Now create a traverse or display operation

2. Algorithm: Traverse/Display
Input:- node (start)
Output:-nodes with data and links

- Initialize a node current which initially points to the head of the list.
- Traverse through the list till current points to null.
- Display each node by making current to point to node next to it in each iteration.

```
//display() will display all the nodes present in the list
   public void display() {
        //Node current will point to head
        Node current = head;

        if(head == null) {
            System.out.println("List is empty");
            return;
        }
        System.out.println("Nodes of singly linked list: ");
```

```
        while(current != null) {
            //Prints each node by incrementing pointer
            System.out.print(current.data + " ");
            current = current.link;
        }
        System.out.println();
    }
```

The main class to call the operations create and display

```
public class Data_structure_with_java { //main class


       public static void main(String[] args) {

        Linklist list = new Linklist();    //create object

        //Add nodes to the list
        list.createNode(10); //create a node with elements
        list.createNode(20);
        list.createNode(30);
        list.createNode(40);
        list.createNode(50);


        //Displays the nodes present in the list
        list.display();//display the nodes
    }
    }
```

```
    Output:- run:(netbeans 8.2)
Nodes of singly linked list:
10 20 30 40 50
BUILD SUCCESSFUL (total time: 0 seconds)
```

3. Algorithm :Insertion

Input:-position (from middle)

Output: - element inserted in middle.

Program:- To insert the node in middle

- **Considering link list already created with few nodes into it.**

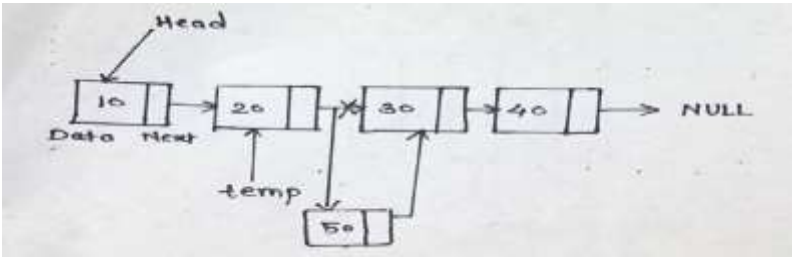


Fig 11.4 Insertion in middle

insertMiddle() will add a new node at the middle of the list:

- First checks, whether the list is empty.
- If the list is empty, both head and tail will point to a newly added node.
- If the list is not empty, then calculate the size of the list and divide it by 2 to get mid-point of the list.
- node current that will iterate through the list until current will point to the mid node.
- another node temp which will point to node next to current.
- The new node will be inserted after current and before temp such that current will point to the new node and the new node will point to temp.

```
/* this code inserts a new node after the given previous node. */
public void insertAfter(Node prevnode, int data)
{
      /* 1. Check if the given Node is null */
      if (prevnode == null)
      {
            System.out.println("The given previous node cannot be
null");
            return;
      }

      /* 2. Allocate the Node &
      3. Put in the data*/
      Node newnode = new Node(data);

      /* 4. Make next of new Node as next of prev_node */
      newnode.next = prevnode.next;

      /* 5. make next of prev_node as new_node */
      prevnode.next = newnode;
        }
```

Algorithm :Insertion

Input:-position (from beginning)

Output: - element inserted at beginning

Program:- To insert the node at beginning

- **Considering link list already created with few nodes into it.**

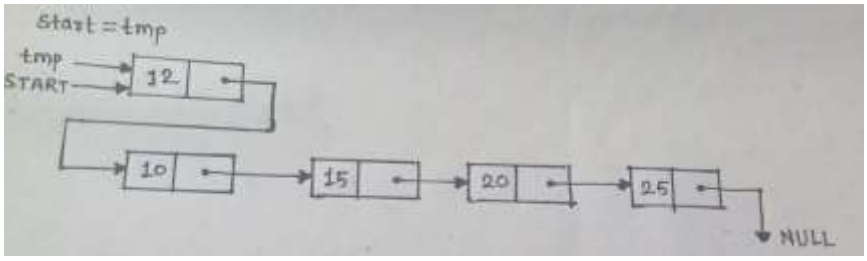


Fig 11.5 Insertion at beginning

```
/* this code inserts a new Node at front of the list. This method
is
defined inside LinkedList class shown above */
public void push(int data)
{
      /* 1 & 2: Allocate the Node & Put in the data*/
      Node newnode = new Node(data);

      /* 3. Make next of new Node as head */
      newnode.next = head;

      /* 4. Move the head to point to new Node */
      head = newnode;
}
```

Algorithm: Insertion

Input:-position (from last)

Output: - element inserted at the end.

Program:- To insert the node at end

- **Considering link list already created with few nodes into it.**

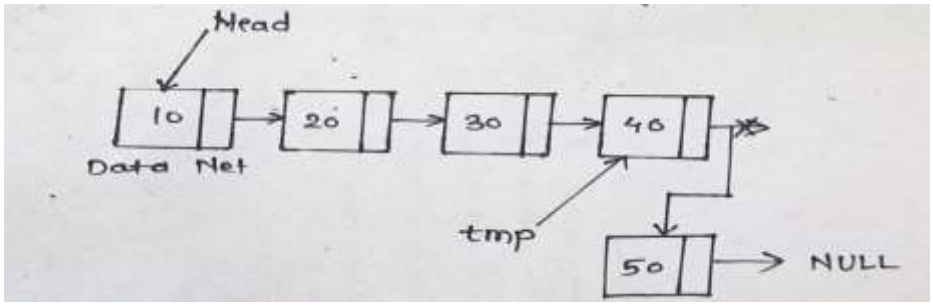


Fig 11.6 Insertion at end

```
/* inserts new node at the end.  */
public void append(int data)
{
      /* 1. Allocate the Node &
      2. Put in the data
      3. Set next as null */
      Node newnode = new Node(data);
```

```
        /* 4. If the Linked List is empty, then make the
              new node as head */
        if (head == null)
        {
              head = new Node(data);
              return;
        }

        /* 4. This new node is going to be the last node, so
              make next of it as null */
        newnode.next = null;

        /* 5. Else traverse till the last node */
        Node last = head;
        while (last.next != null)
              last = last.next;

        /* 6. Change the next of last node */
        last.next = newnode;
        return;
    }
```

4. **Searching**
   - **Algorithm :searching**

Input:- element to search

Output: - element found or not found

1) Initialize a node pointer, current = head.

2) while current is not NULL

a) current->key is equal to the key being searched return true else return false

b) current = current->next

Program:- To search a element in linked list

- **Considering link list already created with few nodes into it.**

// linked list java program to search an element

```
//Node class
class Nodesearch
{
      int data;
      Nodesearch next;
      Nodesearch(int d)
      {
            data = d;
            next = null;
      }
}
```

```
//Linked list class
class SearchLinkList
{
      Nodesearch head; //Head of list

      //Inserts a new node at the front of the list
      public void push(int new_data)
      {
            //Allocate new node and putting data
            Nodesearch new_node = new Nodesearch(new_data);

            //Make next of new node as head
            new_node.next = head;

            //Move the head to point to new Node
            head = new_node;
      }

      //Checks whether the value x is present in linked list
      public boolean search(Nodesearch head, int x)
      {
            Nodesearch current = head; //Initialize current
            while (current != null)
            {
                  if (current.data == x)
                        return true; //data found
                  current = current.next;
            }
            return false; //data not found
      }

      //Driver function to test the above functions
      public static void main(String args[])
      {

            //Start with the empty list
            SearchLinkList llist = new SearchLinkList();

            /*Use push() to construct below list
            14->21->11->30->10 */
            llist.push(01);
            llist.push(5);
            llist.push(87);
            llist.push(33);
            llist.push(56);

            if (llist.search(llist.head, 1))
                  System.out.println("Element found........");
            else
                  System.out.println("Element Not found");
      }
}
```

5. Deleting

Algorithm : Deleting a node from given position

Input:-position

Output: - element deleted from position.

Program: - To delete the node using position.

- **Considering link list already created with few nodes into it.**

```
import java.util.Scanner;

// A complete working Java program to delete a node in a linked
list
// at a given position
class LinkedListDel
{
      Node head; // head of list

      /* Linked list Node*/
      class Node
      {
            int data;
            Node next;
            Node(int d)
            {
                  data = d;
                  next = null;
            }
      }

      /* Inserts a new Node at front of the list. */
      public void push(int new_data)
      {
            /* 1 & 2: Allocate the Node &
                        Put in the data*/
            Node new_node = new Node(new_data);

            /* 3. Make next of new Node as head */
            new_node.next = head;

            /* 4. Move the head to point to new Node */
            head = new_node;
      }

      /* Given a reference (pointer to pointer) to the head of a
list
      and a position, deletes the node at the given position */
      void deleteNode(int position)
      {
            // If linked list is empty
            if (head == null)
                  return;
```

```
            // Store head node
            Node temp = head;

            // If head needs to be removed
            if (position == 0)
            {
                  head = temp.next; // Change head
                  return;
            }

            // Find previous node of the node to be deleted
            for (int i=0; temp!=null && i<position-1; i++)
                  temp = temp.next;

            // If position is more than number of ndoes
            if (temp == null || temp.next == null)
                  return;

            // Node temp->next is the node to be deleted
            // Store pointer to the next of node to be deleted
            Node next = temp.next.next;

            temp.next = next; // Unlink the deleted node from list
      }

      /* This function prints contents of linked list starting
from
            the given node */
      public void printList()
      {
            Node tnode = head;
            while (tnode != null)
            {
                  System.out.print(tnode.data+" ");
                  tnode = tnode.next;
            }
      }

      /* program to test above functions. Ideally this function
      should be in a separate user class. It is kept here to keep
      code compact */
      public static void main(String[] args)
      {
            /* Start with the empty list */
            LinkedListDel llist = new LinkedListDel();

            llist.push(7);
            llist.push(1);
            llist.push(3);
            llist.push(2);
            llist.push(8);
```

```
            System.out.println("\nCreated Linked list is: ");
            llist.printList();
                  System.out.println("enter position to delete
from");
                  Scanner s=new Scanner(System.in);
                  int pos=s.nextInt();
            llist.deleteNode(pos); // Delete node at position

            System.out.println("\nLinked List after Deletion at
position"+pos);
            llist.printList();
      }
}
```

11.3 Doubly linked list

- One of the disadvantages of singly linked list is that it traverses in only one direction. There are several applications and requirements in which the list should traverse in both the directions.
- So to overcome above problem we now move doubly linked list which is also called as two-way linked list.
- A linked list which can be traversed both in backward as well as forward direction is called doubly link list.
- In this type of linked list each node holds two-pointer field. Pointers exist between adjacent nodes in both directions.
- In the doubly linked list each node holds three field two field for link which is the reference to next and previous and one for data record.
- The node has null only to the first node of previous and last node at the next. The list node can be traversed either forward or backward.

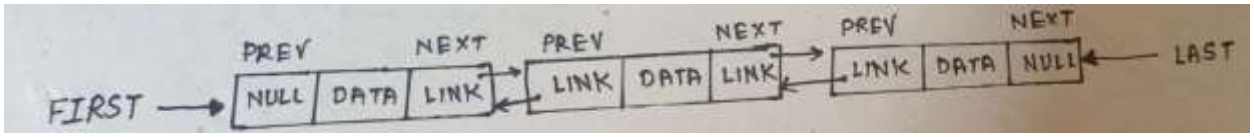


Fig:- 11.7 Doubly Linked List

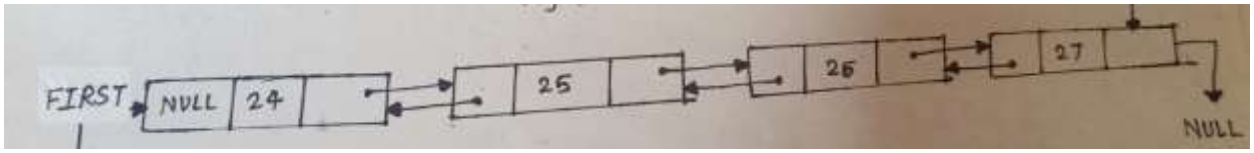


Fig:- 11.8 Doubly Linked List

```
// Class for Doubly Linked List

public class DoublyLinkedList {
```

```
        Node head; // head of list

        //Doubly Linked list Node
        class Node {
              int data;
              Node prev;
              Node next;

              // Constructor to create a new node
              // next and prev is by default initialized as null
              Node(int d) { data = d; }
        }
    }
```

Advantages of doubly Linked List:

1. Doubly linked list with header node, most of the problems can be solved very easily and effectively.
2. Insertion and deletion are simple as compared to other lists.
3. Efficient utilization of memory, no wastage of memory as in sequential representation.
4. Bidirectional (both forward and backward) traversal helps in efficient and very easily modes can be accessible.
5. Doubly linked list are extensively used in trees because hierarchical structure of the tree can be easily represented.

Disadvantages of doubly linked list:

1. A nodes in a linked requires more memory than a corresponding element in a array
2. Requires two references forward and backward.

Difference between Singly and doubly linked lists:

Sr.No	***Singly linked list***	***Doubly linked list***
1)	Traverse in only one direction.	Traverse in both the directions.
2)	Each node holds one reference.	Each node holds two references.
3)	It consists of data and one forward reference.	It consists of data and one forward reference and one backward reference.
4)	Applications:- Stack can be implemented using singly linked list.	Applications:- Stack and hash table can be implemented using doubly linked list.

5)	Ex:-
	Class Node{ //create the inner class
	int data;
	Node link;
	public Node (int data) { // initialize the data and link
	this.data = data;
	this.link = null;
	}
	} //end of inner class
	} // end of outer class
		Ex:-
		Class Node{ //create the inner class
		int data;
		Node link,prev;
		public Node (int data) { // initialize the data and link
		this.data = data;
		this.link = null;
		this.prev=null;
		}
		} //end of inner class
		} // end of outer class

Program to create doubly linked list.

```
public class DoublyLinkedList {

    //Represent a node of the doubly linked list

    class Node{
        int data;
        Node previous;
        Node next;

        public Node(int data) {
            this.data = data;
        }
    }

    //Represent the head and tail of the doubly linked list
    Node head, tail = null;

    //addNode() will add a node to the list
    public void addNode(int data) {
        //Create a new node
        Node newNode = new Node(data);

        //If list is empty
        if(head == null) {
            //Both head and tail will point to newNode
            head = tail = newNode;
            //head's previous will point to null
            head.previous = null;
            //tail's next will point to null, as it is the last
node of the list
            tail.next = null;
        }
        else {
            //newNode will be added after tail such that tail's
next will point to newNode
            tail.next = newNode;
            //newNode's previous will point to tail
            newNode.previous = tail;
            //newNode will become new tail
            tail = newNode;
            //As it is last node, tail's next will point to null
            tail.next = null;
        }
    }

    //display() will print out the nodes of the list
    public void display() {
        //Node current will point to head
        Node current = head;
```

```
            if(head == null) {
                System.out.println("List is empty");
                return;
            }
            System.out.println("Nodes of doubly linked list: ");
            while(current != null) {
                //Prints each node by incrementing the pointer.

                System.out.print(current.data + " ");
                current = current.next;
            }
        }

        public static void main(String[] args) {

            DoublyLinkedList dList = new DoublyLinkedList();
            //Add nodes to the list
            dList.addNode(100);
            dList.addNode(20);
            dList.addNode(300);
            dList.addNode(004);
            dList.addNode(-5);

            //Displays the nodes present in the list
            dList.display();
        }
    }
```

****

Summary:

- A linked list is a data structure that is basically a chain of nodes in which each node is connected to each other by means of pointers or references.
- Linked list is dynamic in nature that means that the size of the linked list can vary depending on the requirements of the users.
- The basic building block of linked list is a node.
- There are a few different types of linked lists. A single linked list, a doubly linked list, a multilinked list, and a circular linked list.
- Operations of linked list are Insertion, Deletion, traversal, Searching and Deletion.

Questions:

1. What is linked list? Explain the need of linked list?
2. Give Advantages and Disadvantages of Linked list.
3. Explain types of linked lists.
4. Write a program for search an element in linked list.
5. Write a program for insertion of an element in linked list.
6. Write a program for deletion of an element from linked list.
7. Give node structure for doubly linked list. Write advantages of doubly linked list over singly list.
8. Explain operations on singly linked list.
9. Write an algorithm to insert a new node as the last of a singly linked list. Give example.
10. Describe working of doubly linked list. Write syntax used for double linked list in program.
11. Write an algorithm to traverse a singly linked list.
12. Differentiate between the following:
    a. Linear linked list, doubly linked list and circular linked list.
    b. Linked List and Array
    c. Linked List, Queue and stack

Chapter 12: Hashing in Data Structures

Objectives:

- ✓ Hash table methods - hashing functions
- ✓ collision-resolution techniques

12.1 Hash table methods:

One of the important applications of Data structure is to have searched an element but as data increases it becomes very time consuming to search an element from the data store. So in computing, a hash table (called sometimes as hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values.

A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

Hashing

Before understanding a hash table first, we will understand what we mean by hashing, which is applied on hash table. Hashing is a procedure to convert a range of key values into a range of indexes of an array. We will use modulo operator to get a range of key values. Consider an example of hash table of size 15 and the following items are to be stored. Item are in the (key, value) format.

index = hash % array size

In java we have the following fields for designing a data structure for hash table.

```
class Hashtable
{
    int[] list;
    int capacity;
}
```

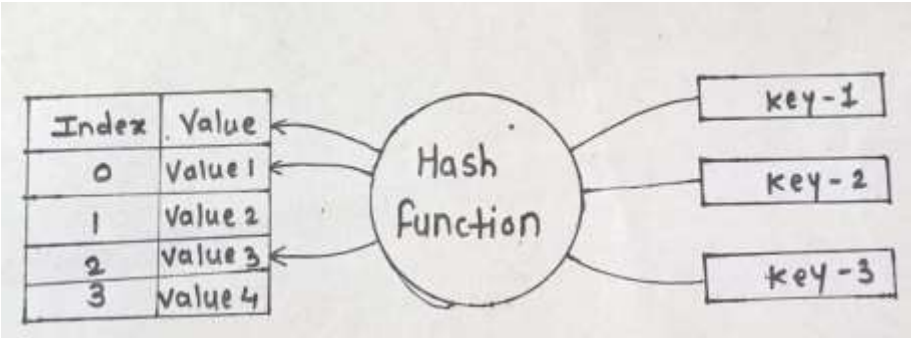


Fig : -12.1 Hashing

- (1,10)
- (15,50)
- (25,80)
- (0,15)
- (12,34)
- (14,52)
- (17,51)
- (13,68)
- (37,80)
- (20,60)

Sr.No.	Key	Hash	Array Index
1	1	1 % 15 = 1	1
2	15	15 % 15 = 0	0
3	25	25 % 15 = 10	10
4	0	0 % 15 = 0	0
5	12	12 % 15 = 12	12
6	14	14 % 15 = 14	14
7	17	17 % 51 = 17	17
8	13	13 % 68 = 13	13
9	37	37 % 80 = 37	37
10	20	20 % 60= 20	20

Table : 12.1

In hashing, large keys are transformed into small key. The values are then stored in a data structure called hash table. The idea of hashing is to allocate entries (key/value pairs) uniformly across an array. Each element is assigned a key. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two stages:

- An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
- The element is stored in the hash table where it can be quickly retrieved using hashed key.

hash = hashfunc(key)
index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

A good hash function requirement:

1. Easy Computation: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Need for a good hash function

Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

Hash Table Data Structure: -

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

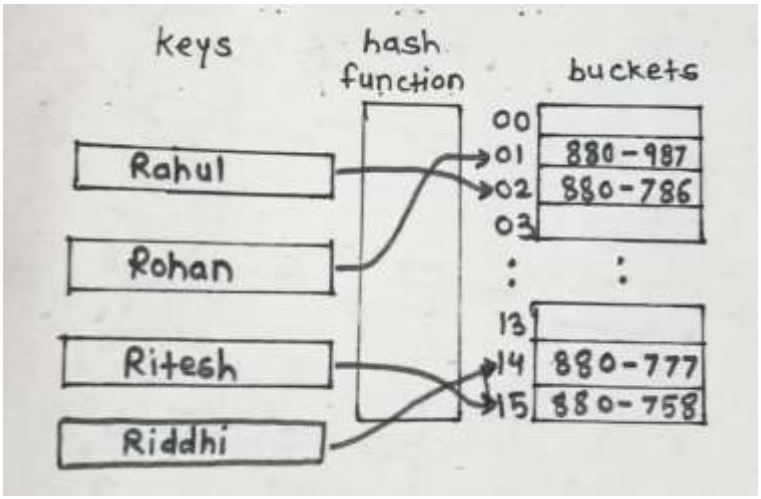


Fig: - 12.2 A Hash table

Hash Table is a data structure which stores data in an associative manner. So the above figure has the hash table data structure divided into keys hash function and buckets.

In a hash table (bucket), data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
KEY----appiled on hash function------→HASH_FUNCTION( )--------→HASH_TABLE

Example:-
hash = hashfunc(key)
HASH_FUCNTION(KEY)=VALUE

John smith = 521-1234

Basic Operations

Following are the basic primary operations of a hash table.

- Contains − Searches an element in a hash table.
- Insert − inserts an element in a hash table.
- Delete − Deletes an element from a hash table.

Algorithm for creating hash table data structure: -

Input: - capacity, values in the hash table and key to search
Output: - value contained in table or not there.

1. Create a hash table and allocate the values using following :
   hash = hashfunc(key)
   index = hash % array_size
   Calculate prime numbers to allocate the values.

2. Input the values
3. Input the key value to search.
4. If value found return true else return false.

Program to implement hash table and its operations:

```
import java.util.Scanner;
 class Hashtable
{
    int[] list;
    int capacity;

    /** constructor **/
    public Hashtable(int capacity)
    {
        this.capacity = nextPrime(capacity);
        list = new int[this.capacity];
    }

    /** function to insert **/
    public void insert(int ele)
    {
        list[ele % capacity] = ele;
    }

    /** function to clear **/
    public void clear()
    {
        list = new int[capacity];
    }

    /** function contains **/
    public boolean contains(int ele)
    {
        return list[ele % capacity] == ele;
    }

    /** function to delete **/
    public void delete(int ele)
    {
        if (list[ele % capacity] == ele)
            list[ele % capacity] = 0;
        else
            System.out.println("\nError : Element not found\n");
    }

    /** Function to generate next prime number >= n **/
    private static int nextPrime( int n )
    {
        if (n % 2 == 0)
            n++;
        for (; !isPrime(n); n += 2);
```

```
        return n;
    }

    /** Function to check if given number is prime **/
    private static boolean isPrime(int n)
    {
        if (n == 2 || n == 3)
            return true;
        if (n == 1 || n % 2 == 0)
            return false;
        for (int i = 3; i * i <= n; i += 2)
            if (n % i == 0)
                return false;
        return true;
    }

    /** function to print hash table **/
    public void printTable()
    {
        System.out.print("\nHash Table = ");
        for (int i = 0; i < capacity; i++)
            System.out.print(list[i] +" ");
        System.out.println();
    }
}

/** Class Hasht **/
public class Hasht
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");
        /** Make object of HashTable **/
        Hashtable ht = new Hashtable(scan.nextInt() );

        char ch;
        /**  Perform HashTable operations  **/
        do
        {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
            System.out.println("3. contains");
            System.out.println("4. clear");

            int choice = scan.nextInt();
            switch (choice)
            {
            case 1 :
```

```
                System.out.println("Enter integer element to insert");
                ht.insert( scan.nextInt() );
                break;
            case 2 :
                System.out.println("Enter integer element to delete");
                ht.delete( scan.nextInt() );
                break;
            case 3 :
                System.out.println("Enter integer element to check if
present");
                System.out.println("Contains           :           "+
ht.contains(scan.nextInt() ));
                break;
            case 4 :
                ht.clear();
                System.out.println("Hash Table Cleared\n");
                break;
            default :
                System.out.println("Wrong Entry \n ");
                break;
            }
            /** Display hash table **/
            ht.printTable();

            System.out.println("\nDo you want to continue (Type y or
n) \n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y'|| ch == 'y');
    }
}
```

Output:-

Run (netbeans 8.2)

run:

Hash Table Test

Enter size

2

Hash Table Operations

1. insert

2. remove

3. contains

4. clear

1

Enter integer element to insert

7

Hash Table = 0 7 0

Do you want to continue (Type y or n)

y
Hash Table Operations
1. insert
2. remove
3. contains
4. clear
1
Enter integer element to insert
9
Hash Table = 9 7 0
Do you want to continue (Type y or n)
y
Hash Table Operations
1. insert
2. remove
3. contains
4. clear
3
Enter integer element to check if present
9
Contains : true
Hash Table = 9 7 0
Do you want to continue (Type y or n)

Subsequently, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table practices an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be found from.

Hash function assigns each key to a unique bucket, but most hash table designs employ an defective hash function, which might cause hash collisions where the hash function generates the same index for more than one key. Such collisions are always accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table.

In many situations, hash tables are on average more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets

Applications of hash table data structures: -

Associative arrays: Hash tables are used to implement many types of in-memory tables. They are used to implement associative arrays whose indices are random strings or other complicated objects.

Database indexing: Hash tables may also be used as disk-based data structures and database indices.

Caches: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.

Object representation: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.

Hash Functions are used in various algorithms to make their computing faster.

12.2 Collision Resolution Techniques:

Separate chaining

Separate chaining also known as open hashing is one of the most commonly used collision resolution techniques. It is commonly implemented using linked lists. In separate chaining, each element of the hash table is a linked list data structure. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

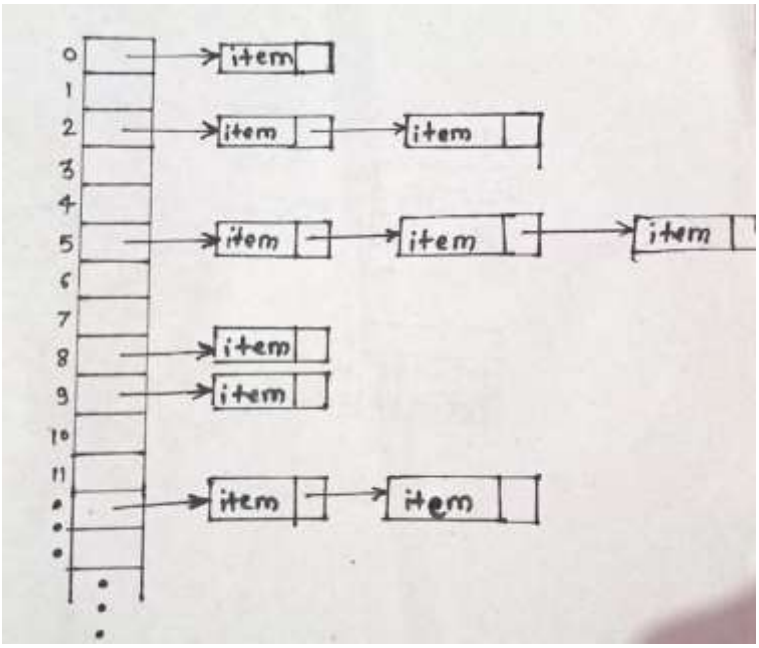


Fig:- 12.3 Chaining

The price of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is constant, then the average cost of a lookup depends only on the average number of keys per linked list. Because of which the chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

The worst-case scenario is when all the entries are inserted into the same.The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

In the following image, **CodeMonk** and **Hashing** both hash to the value **2**. The linked list at the index **2** can hold only one entry, therefore, the next entry (in this case **Hashing**) is linked (attached) to the entry of **CodeMonk**.

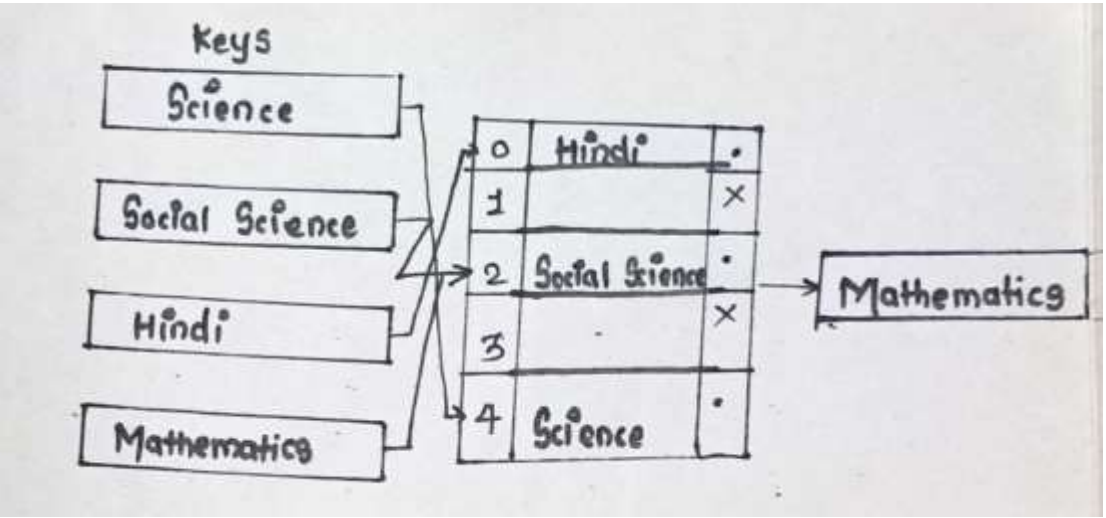


Fig:- 12.4

Implementation of hash tables with separate chaining (open hashing)

Assumption

Hash function will return an integer from 0 to 19.

Linear probing (open addressing or closed hashing)

In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is **index**. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

and so on…

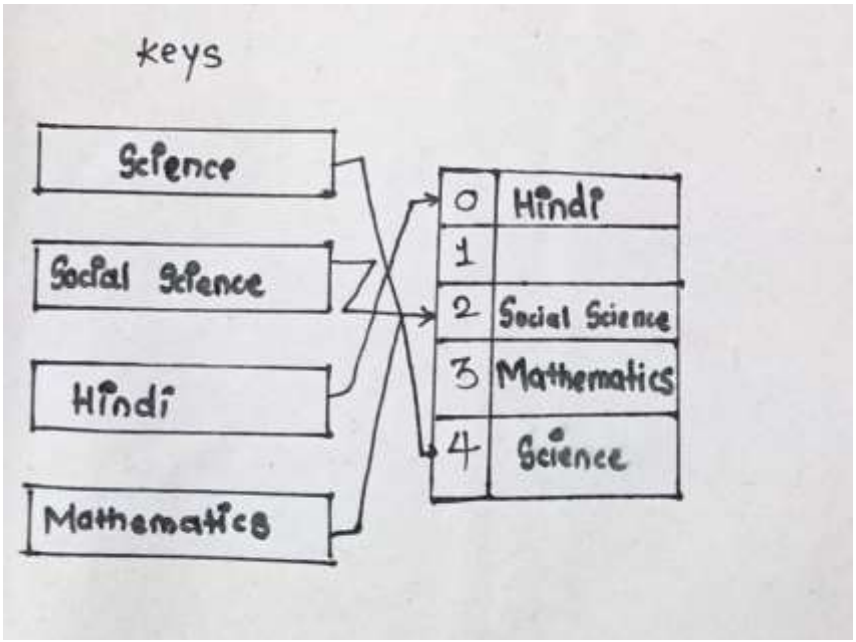


Fig:- 12.5 A Hash table

Hash collision is resolved by open addressing with linear probing. Since **CodeMonk** and **Hashing** are hashed to the same index i.e. **2**, store **Hashing** at **3** as the interval between successive probes is **1**.

Implementation of hash table with linear probing

Assumption

- There are no more than 20 elements in the data set.
- Hash function will return an integer from 0 to 19.
- Data set must have unique elements.

Quadratic Probing

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is **index** and at **index** there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize and so on

Implementation of hash table with quadratic probing

Assumption

- There are no more than 20 elements in the data set.
- Hash function will return an integer from 0 to 19.

Data set must have unique elements.

Double hashing

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;

and so on…

Here, **indexH** is the hash value that is computed by another hash function.

****

Summary:-

- Hashing is a technique which uses less key comparisons and searches the element in O(n) time in the worst case and in an average case it will be done in O(1) time.
- Hashing method generally used the hash functions to map the keys into a table, which is called a hash table.
- Three methods in open addressing are linear probing, quadratic probing, and double hashing.
- These methods are of the division hashing method because the hash function is f(k) = k % M.
- Some other hashing methods are middle-square hashing method, multiplication hashing method, and Fibonacci hashing method, and so on.

Questions:-

1. Define Hashing.
2. What is need of hashing?
3. Explain Hash table methods.
4. Discuss hashing functions.
5. Explain different collision-resolution techniques in detail.
6. Write Short notes on :
    a. Linear probing
    b. Quadratic probing
    c. Double hashing.

Chapter 13 Trees

Objectives:

- ✓ **Trees- Binary Trees**
- ✓ **Traversing binary tree**
- ✓ **Traversing algorithm using stacks**
- ✓ **Header nodes**
- ✓ **Threads**
- ✓ **Binary search trees (insertion and deletion)**
- ✓ **AVL trees**
- ✓ **B trees**

Introduction to Tree:

In all the previous topics we have seen data structures like stack, queue and linked list which follows a specific order and are called as linear data structures. Now we are going to understand non-linear data structures do not form a sequence and thus called as non-linear data structures. This hierarchical structure which has relationship between data elements is called as tree. Tree Data structures not only stores data in hierarchical manner but have number of applications, ease and efficiency of various data structure operations like searching, sorting, etc. compare to linear data structure.

Tree Data Structure:

Tree is a collection of finite set "T" of one or more nodes such that there is a node assigned as the root of the tree and other nodes are divided into n>=0 disjoint sets T1, T2,…., Tn and each of this node is tree in turn. Nodes in root are called as sub-tree or child of the root.

Definition of Tree:

A tree is a non-linear data structure used to represent the hierarchical structure of one or more elements known as nodes.

Tree holds values and has either zero or more than zero references which are referring to other nodes known as child nodes. A tree can have zero or more child nodes, which is at one level below it and each child node can have only one parent node which above it. The node at the top of the tree is known as root of the tree and the node at the lowest level is known as leaf node.

Example: - a tree of an organization

Tree with one or more child nodes are called as internal nodes and the node without child is called as external nodes.

Applications of Trees:

- Hierarchical Management: - a tree like structure helps us to understand the protocol easily. Example: -Board of management.
- Effortlessness searching: - as data is in sorted manner so searching is easier compare to other linear data structures. Example: - searching a file in linux .
- Easy Manipulations: as data is two linear so can be used to process. Example: - sorting according to date_of _joining.

13.1 Trees- Binary Trees:

A binary tree has a special requirement that each node can have a maximum of two children.

One of the advantages of using binary tree over both static linear data structure i.e. ordered array and a dynamic linear data structure i.e. linked list will have searching faster.

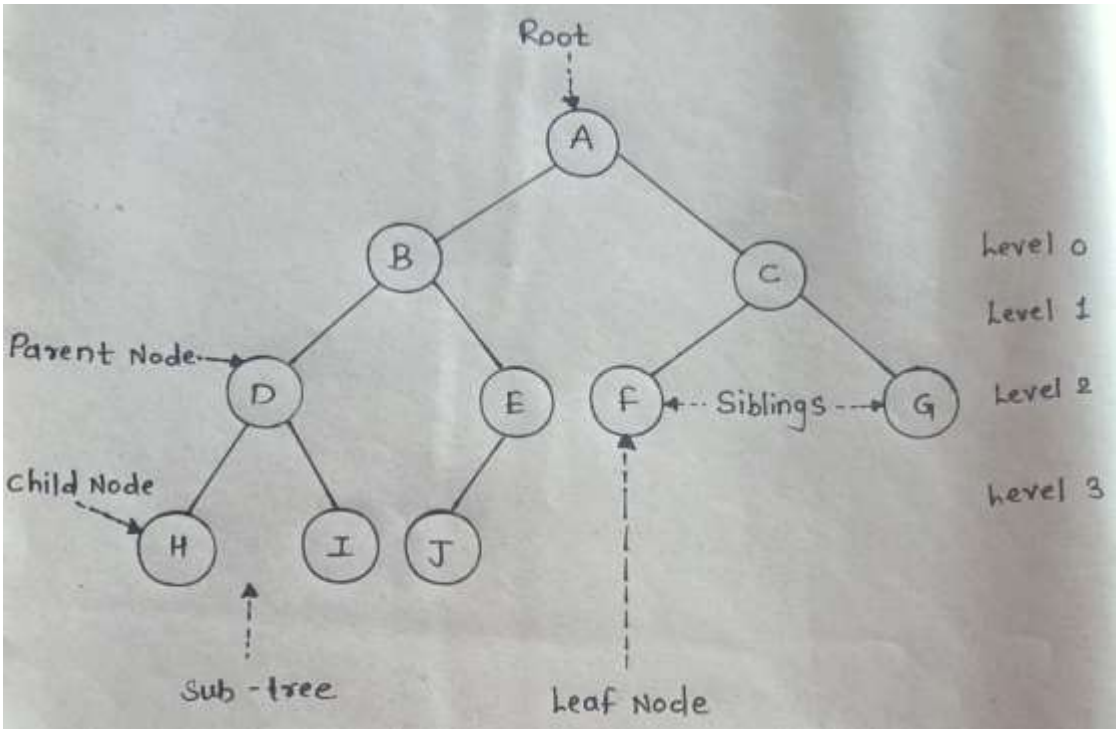


Fig: -13.1

Terminologies used TREE

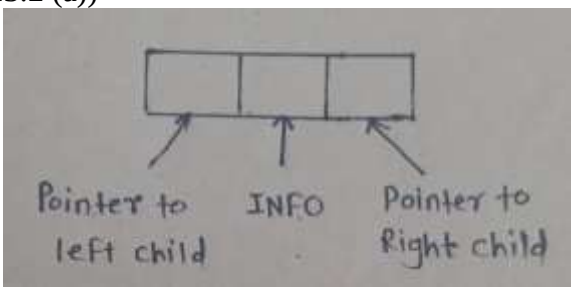
Following are the important terms with respect to tree.

1. Root – The mother node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
2. Path − Path refers to the sequence of nodes along the edges of a tree.
3. Node − every individual element in tree is called as node. Every information is stored in the node with the value and link to other nodes.
4. Parent node− any node excluding the root node has one edge upward to a node called parent node.

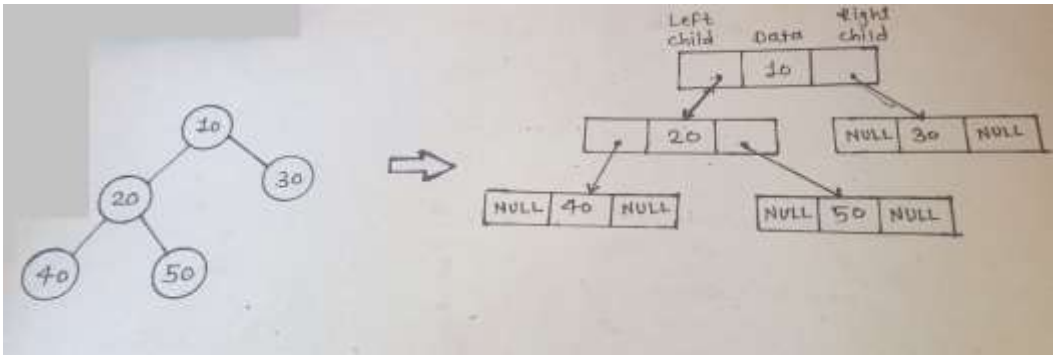
5. Child node − the node below a given node connected by its edge downward is called its child node.
6. Siblings – Nodes which belongs to same node is called as siblings of each other.
7. Leaf − the node which does not have any child node is called the leaf node.
8. Sub tree − Sub tree denotes the descendants of a node.
9. Visiting − Visiting means checking the value of a node when control is on the node.
10. Traversing − Traversing means passing through nodes in a specific order.
11. Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
12. Keys − Key represents a value of a node based on which a search operation is to be carried out for a node.
13. Predecessor: Every node in binary tree, except the root has unique parent called predecessor.
14. Descendants: The descendants of a node are all those nodes which are reachable from that node.
15. Ancestors: The ancestors of a nodes are all the node along the path from the root to that node.
16. In-degree: The in-degree of a node is the total number of edges coming to the node.
17. Out degree: The out degree of a node is the total number of edges going outside from the node
18. Level: Level is a rank of a tree order. The whole tree structure is leveled. The level of root node is always at 0.the immediate children of root are at level 1 and their immediate children are at level 2 and so on.
19. Depth of Tree: the depth of a tree is the longest path from the root to leaf node. In a tree data structure, the total number of edges from node to a particular node is called as depth of that node.
20. Height: The highest number of nodes that is possible in a way starting from the first node {root} to a leaf node is called the height of a tree.

Linked Implementation of Trees

- In this representation we use singly linked list. In this representation each node requires three fields, (see, Fig.13.2 (a))



(a)



(b)

Fig: -13.2(a) and (b)

1) One for the link of the left child,
2) Second field for representing the information associated with the node, and information associated with the node, and third is used to represent the link of the right child.

When a node has no child then the corresponding pointer fields are null. The left and right field of a node is pointer to left and the right child of that node.

- Example: Fig 13.2(b) shows an example of linked representation of tree.
- Following is the code to declarant a tree data structure in java. Each node of binary tree,(as the root of some sub-tree)has both left and right sub tree, which we can access through pointer by declaring as follows:

```
Class Node
{
    int   data;
    Node left, right
 }
```

13.2 Traversing algorithm using stacks:

Algorithm for in order traversal:

1) Create stack S.
2) Set current node as root
3) Push the current node to S and set current = current->left until current = NULL
4) If current = NULL and stack is not empty then
   a) Pop the top item.
   b) Print the popped item, set current = popped-item->right
   c) Go to step 3.
5) If current is NULL and stack is empty then done.

```
//java program for in order traversal using stack
import java.util.Stack;   //import the stack class from util
```

```
/* Class containing left and right child of
current node and key value*/
class Nodes
{
	int data;
	Nodes left, right;

	public Nodes(int item)
	{
		data = item;
		left = right = null;
	}
}

/* Class to print the inorder traversal */
class BinaryTreeStackTrav
{
	Nodes root;
	void inorder()
	{
		if (root == null)
			return;


		Stack<Nodes> s = new Stack<Nodes>();	//using
collection object stack
		Nodes curr = root;

		// traverse the tree
		while (curr != null || s.size() > 0)
		{

			/* Reach the left most Node of the
			curr Node */
			while (curr != null)
			{
				/* place pointer to a tree node on
				the stack before traversing
				the node's left subtree */
				s.push(curr);
				curr = curr.left;
			}

			/* Current must be NULL at this point */
			curr = s.pop();
```

```
                        System.out.print(curr.data + " ");

                        /* we have visited the node and its
                        left subtree. Now, it's right
                        subtree's turn */
                        curr = curr.right;
                  }
            }

            public static void main(String args[])
            {

                  /* creating a binary tree and entering
                  the nodes */
                  BinaryTreeStackTrav       tree       =       new
BinaryTreeStackTrav();
                  tree.root = new Nodes(1);
                  tree.root.left = new Nodes(2);
                  tree.root.right = new Nodes(3);
                  tree.root.left.left = new Nodes(4);
                  tree.root.left.right = new Nodes(5);
                  tree.inorder();
            }
}
```

Algorithm for preorder traversal:

1) Create an stack nodeStack and push root node to stack.
2) Do following while nodeStack is not empty.
   I. Pop an item from stack and print it.
   II. Push right child of popped item to stack
   III. Push left child of popped item to stack

```
// Java program to implement iterative preorder traversal
import java.util.Stack;
// A binary tree node
class Nodess {

            int data;
            Nodess left, right;

            Nodess(int item) {
                  data = item;
                  left = right = null;
            }
}
```

```
class BinaryTreePre {

	Nodess root;

	void iterativePreorder()
	{
		iterativePreorder(root);
	}

	// An iterative process to print preorder traversal of
Binary tree
	void iterativePreorder(Nodess node) {

		// Base Case
		if (node == null) {
			return;
		}

		// Create an empty stack and push root to it
		Stack<Nodess> nodeStack = new Stack<Nodess>();
		nodeStack.push(root);

		/* Pop all items one by one. Do following for
every popped item
		a) print it
		b) push its right child
		c) push its left child
		Note that right child is pushed first so that
left is processed first */
		while (nodeStack.empty() == false) {

			// Pop the top item from stack and print it
			Nodess mynode = nodeStack.peek();
			System.out.print(mynode.data + " ");
			nodeStack.pop();

			// Push right and left children of the
popped node to stack
			if (mynode.right != null) {
				nodeStack.push(mynode.right);
			}
			if (mynode.left != null) {
				nodeStack.push(mynode.left);
			}
		}
```

```
            }

            // driver program to test above functions
            public static void main(String args[]) {
                  BinaryTreePre tree = new BinaryTreePre();
                  tree.root = new Nodess(20);
                  tree.root.left = new Nodess(18);
                  tree.root.right = new Nodess(22);
                  tree.root.left.left = new Nodess(-3);
                  tree.root.left.right = new Nodess(35);
                  tree.root.right.left = new Nodess(26);
                  tree.iterativePreorder();

            }
}
```

Algorithm for post order traversal

1. Create a stack
2. Do following while root! = NULL
   a) Push root's right child and then root to stack.
   b) Set root as root's left child.
3. Pop an item from stack and set it as root.
   a) If the popped item has a right child and the right child
      is at top of stack, then remove the right child from stack,
      push the root back and set root as root's right child.
   b) Else print root's data and set root as NULL.
4. Repeat steps 2 and 3 while stack is not empty.

```
// A java program for  postorder traversal using stack

import java.util.ArrayList; // import collection classes
import java.util.Stack;

// A binary tree node
class Nodesss
{
            int data;
            Nodesss left, right;

            Nodesss(int item)
            {
                  data = item;
                  left = right;
            }
}
```

```
class BinaryTreePo
{
	Nodesss root;
	ArrayList<Integer> list = new ArrayList<Integer>();

	// An iterative function to do postorder traversal
	// of a given binary tree
	ArrayList<Integer> postOrderIterative(Nodesss node)
	{
		Stack<Nodesss> S = new Stack<Nodesss>();

		// Check for empty tree
		if (node == null)
			return list;
		S.push(node);
		Nodesss prev = null;
		while (!S.isEmpty())
		{
			Nodesss current = S.peek();

			/* go down the tree in search of a leaf an
if so process it
			and pop stack otherwise move down */
			if (prev == null || prev.left == current ||

	prev.right == current)
			{
				if (current.left != null)
					S.push(current.left);
				else if (current.right != null)
					S.push(current.right);
				else
				{
					S.pop();
					list.add(current.data);
				}

				/* go up the tree from left node, if
the child is right
				push it onto stack otherwise process
parent and pop
				stack */
			}
			else if (current.left == prev)
			{
```

```
                        if (current.right != null)
                              S.push(current.right);
                        else
                        {
                              S.pop();
                              list.add(current.data);
                        }

                        /* go up the tree from right node and
after coming back
                        from right node process parent and pop
stack */
                  }
                  else if (current.right == prev)
                  {
                        S.pop();
                        list.add(current.data);
                  }

                  prev = current;
            }

            return list;
      }

      // Driver program to test above functions
      public static void main(String args[])
      {
      BinaryTreePo tree = new BinaryTreePo();

            // Let us create trees shown in above diagram
            tree.root = new Nodesss(11);
            tree.root.left = new Nodesss(20);
            tree.root.right = new Nodesss(34);
            tree.root.left.left = new Nodesss(34);
            tree.root.left.right = new Nodesss(57);
            tree.root.right.left = new Nodesss(61);
            tree.root.right.right = new Nodesss(17);

            ArrayList<Integer>          mylist          =
tree.postOrderIterative(tree.root);

            System.out.println("Post   order   traversal   of
binary tree is :");
            System.out.println(mylist);
      }
```

}

13.3 Binary search trees (insertion and deletion)

- A Binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (data/info) and satisfies the following condition;
    1. The key in the left child of a node, is less than the key in its parent node.
    2. The key in the right child of a node, is greater than the key in its parent node.
    3. The left and right sub-trees of node are again binary search trees.
- The definition ensures that no two entries in a binary search tree can have equal keys, following is the example of binary search trees where all keys in left subtree is less than root and all keys in right subtree are greater than root.
- In order to create a binary search tree on given data use following steps:
  Step1: Read a data in item variable.
  Step2: Allocate memory for a new node and store the address in pointer root.
  Step3: Store the data x in the node root.
  Step4: If (data<root.data).
  Step5: Recursively create the left subtree of root and make it the left child of root.
  Step6: If (data>root.data).
  Step7: Recursively create the right subtree of root and make it the right child of root.
- Below is a recursive algorithm to perform above mentioned steps.
  Algorithm: create (root, data)
  Root is initially NULL; data is key which you want to insert
    1. If root is NULL
    2. create a node with data
    3. return
    4. else
    5. If data is greater than root.data
    6. root. left=create (root. left , data)
    7. else
    8. Root, right=create (root .left ,data)

  Example:

- Let create a binary search tree for sequence:20,17,29,22,45,9,19
- Step1: The first item is 20 and this is the root node, so begin the tree.
- Step2: Sequence 20,17,29,22,45,9,19.
  This is a binary search tree, so there are two child nodes available, the left and the right. The next number is 17, the rule is applied (left is less than parent node) and so it has to be the left node, like this,
- Step3: Sequence 20,17,29,22,45,9,19.
  This next number is 29, this is higher than the root node so it goes to the RIGHT sub-tree which happens to be empty at this stage, so the tree now looks like this,

- Step4: Sequence 20,17,29,22,45,9,19.
  The next number is 22. This is more than the root and so need so be on the RIGHT sub-tree. The first node is already occupied.so the rule is applied again to that node,22 comes before 29 and so it needs to be on the LEFT sub-tree of that node like this,
- Step5: Sequence 20,17,29,22,45,9,19.
  The next number is 45, this is more than the root and more than the first right node,so it is placed on the right side of the tree like this,
- Step6: 4Sequence 20,17,29,22,45,9,19.
  The next number is 9 which is less than the root, the first left node is occupied and 9 is less than that node too, so it is placed on the left sub-tree, like this,
- Step7: Sequence 20,17,29,22,45,9,19.
  The next number is 19, which is less than the root, so it will need to be in the left sub-tree. It is greater than the occupied 17 node and so it is placed in the right sub-tree, like this.

13.3 Header nodes and threads

A binary tree is threaded by making all right child pointers that would normally be null point to the in-order successor of the node and all left child pointers that would normally be null point to the in-order predecessor of the node.

Consider a recursive traversal for a binary search tree, there is a problem with this algorithm is that, because of its recursion, it uses stack space proportional to the height of a tree. If the tree is equally balanced, this volumes to O (log n) space for a tree containing n elements. In the worst case, when the tree takes the form of a chain, the height of the tree is n so the algorithm takes O(n) space.

A second problem is that all traversals must begin at the root when nodes have pointers only to their children. It is common to have a pointer to a specific node, but that is not appropriate to get back to the rest of the tree unless extra information is added, such as thread pointers.

In this method, it may not be possible to tell whether the left and or right pointers in a given node actually point to children, or are a result of threading. If the distinction is necessary, adding a single bit to each node is enough to record it.

Threads are reference to the predecessors and successors of the node according to an inorder traversal.

In-order traversal of the threaded tree is A,B,C,D,E,F,G,H,I, the predecessor of E is D, the successor of E is F.

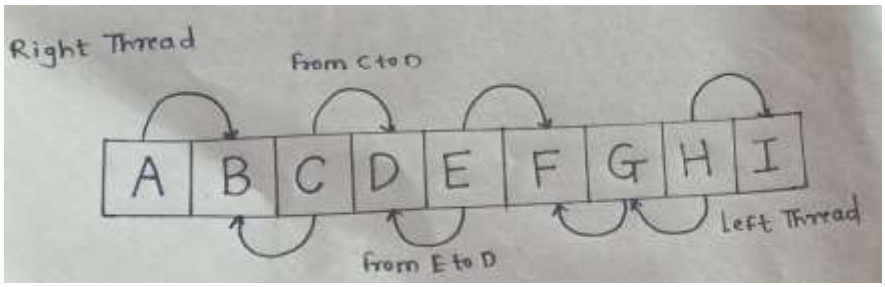


Fig:-13.3

Example

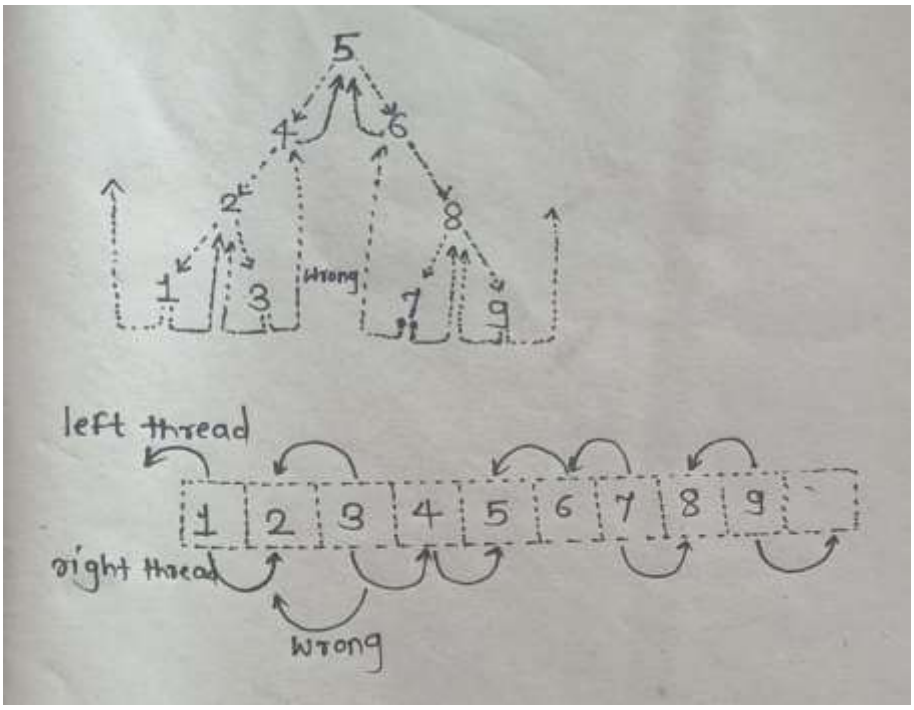


Fig: -13.4

Let's make the Threaded Binary tree out of a normal binary tree:

The in-order traversal for the above tree is — D B A E C. So, the respective Threaded Binary tree will be

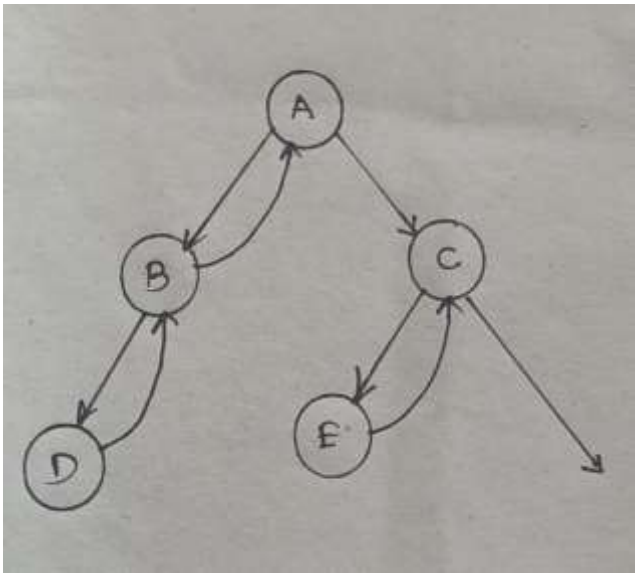


Fig: -13.5

13.4 Binary Search Tree

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

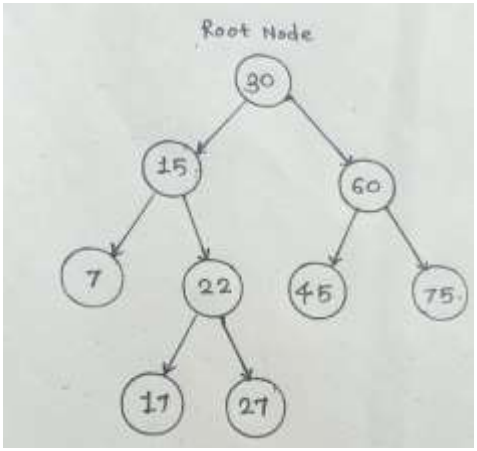


Fig: -13.6

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.

Benefits of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the preferred element.
2. The binary search tree is considered as well-organized data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2$n) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

- Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown below

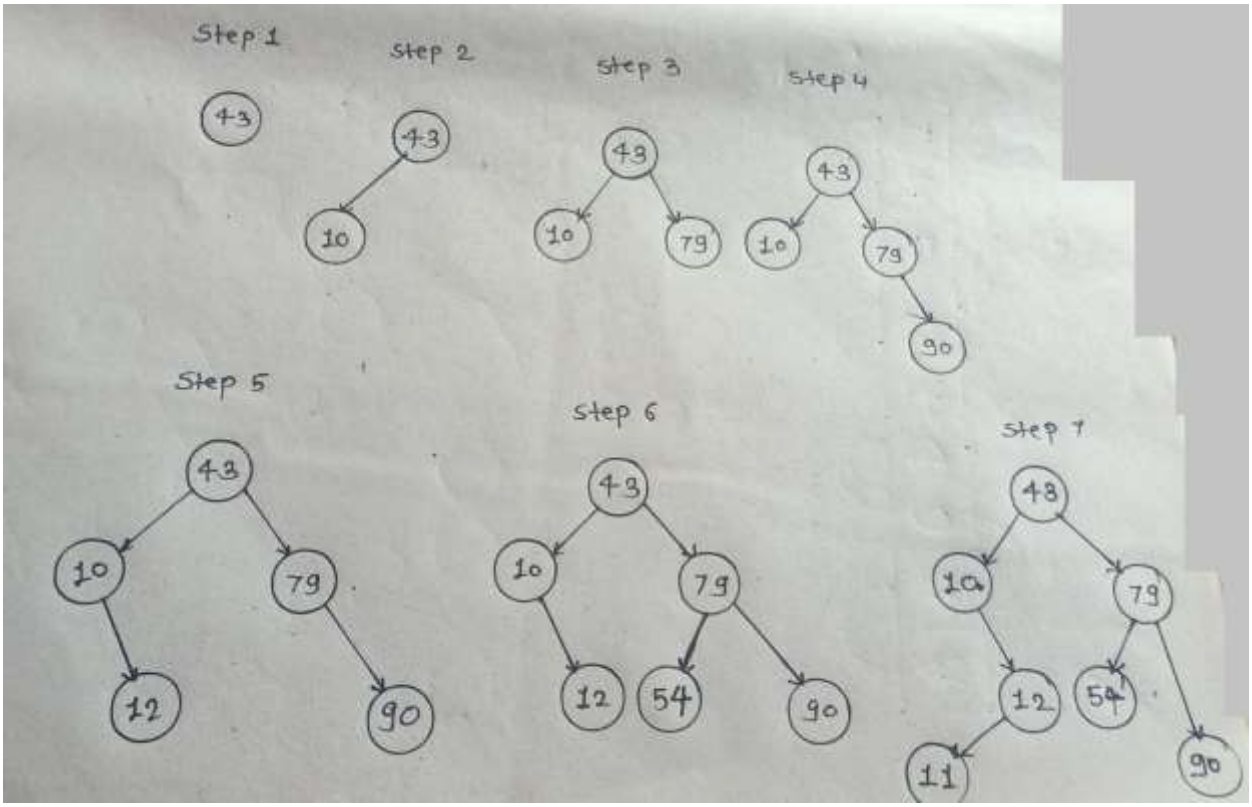


Fig: -13.7 (a)

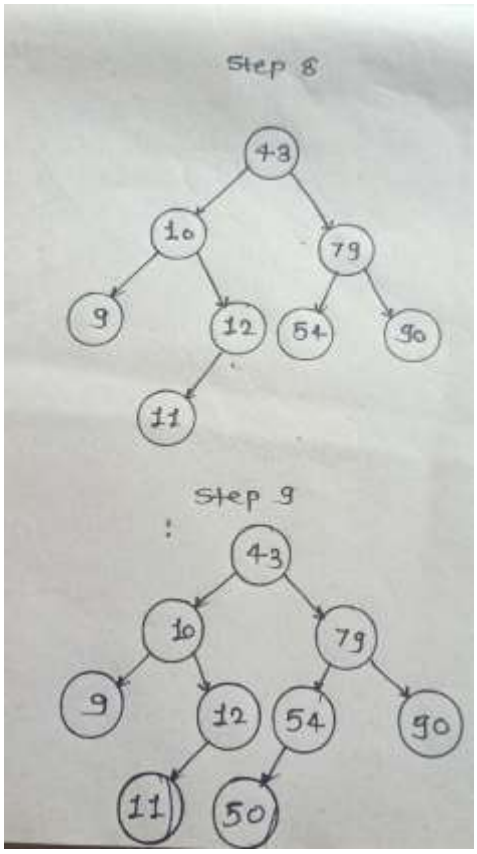


Fig: -13.7 (b)

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree but few are the following: -

1. Insertion Operation
2. Deletion Operation

1. Insertion Operation-

The insertion of a new key always considers 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-

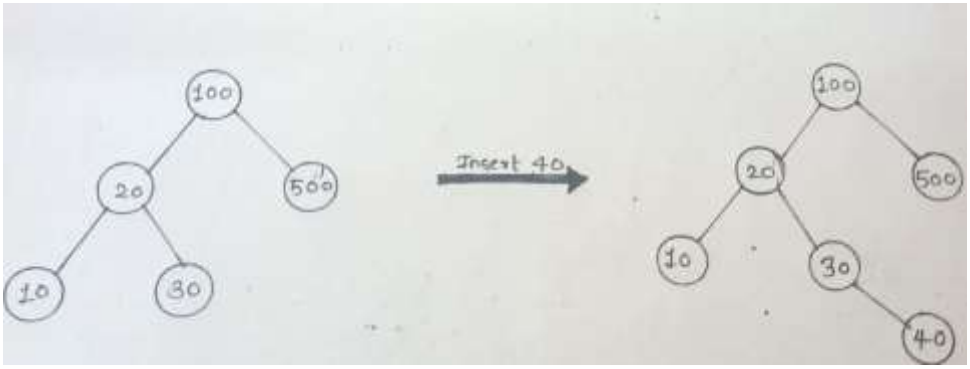


Fig: -13.8

We from 40 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.

2. Deletion Operation-

When it comes to deleting a node from the binary search tree, following three cases are possible-

Case-01: Deletion Of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to deleted from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-

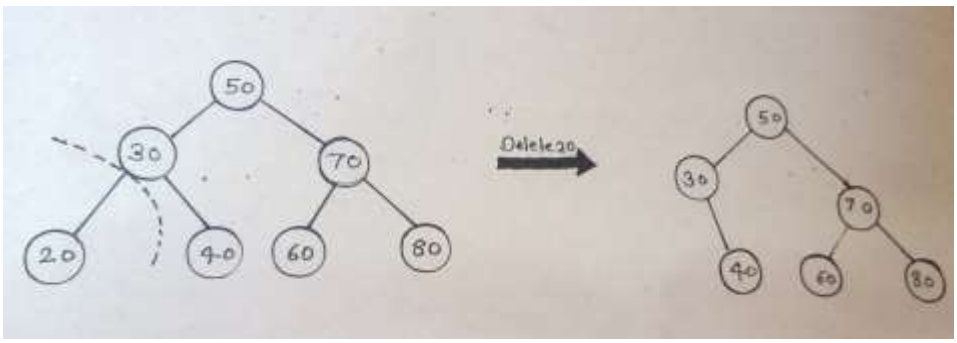


Fig: -13.8

Case-02: Deletion Of A Node Having Only One Child-

Just make the child of the deleting node, the child of its grandparent.

Example-

Consider the following example where node with value = 30 is deleted from the BST-

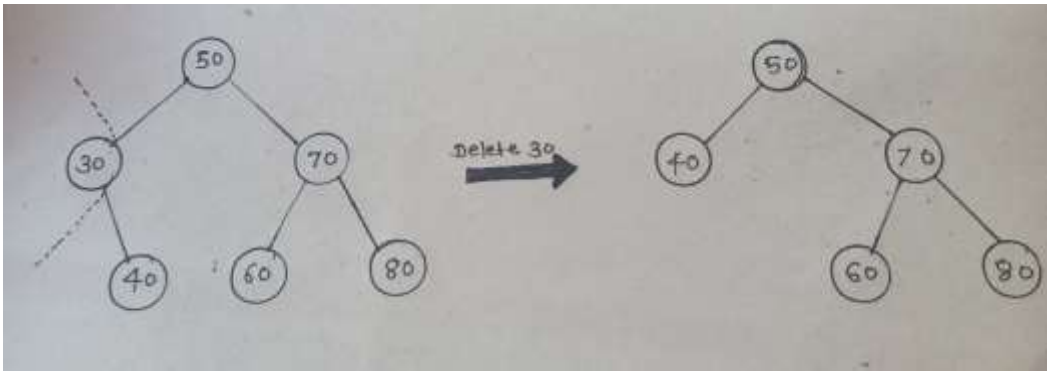


Fig: -13.9

Case-02: Deletion Of A Node Having Two Children-

A node with two children may be deleted from the BST in the following two ways-

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-

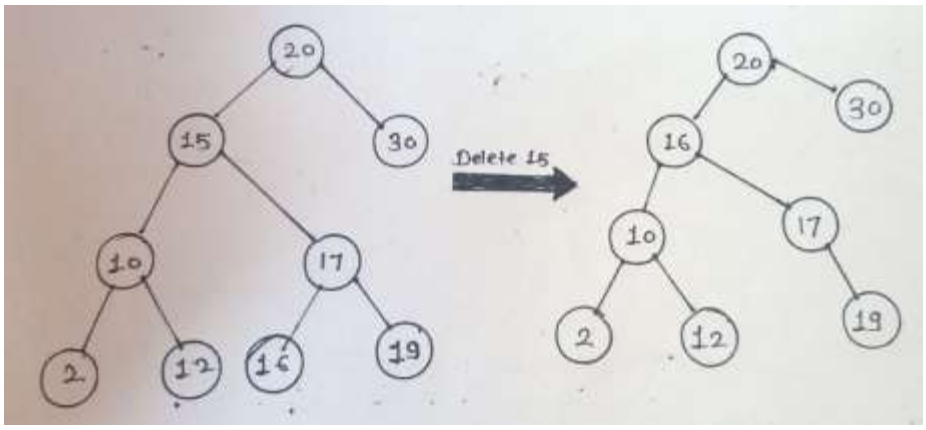


Fig: -13.10

13.5 AVL trees

An **AVL tree** is another balanced binary search tree, named after their originators, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. They are not perfectly balanced like red-black trees, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(*log*n)*** search time. Addition and deletion operations also take ***O(*log*n)*** time.

Definition of an AVL tree

An AVL tree is a binary search tree which has the following properties:

- The sub-trees of every node differ in height by at most one.
- Every sub-tree is an AVL tree.
- B Tree

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and it need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.

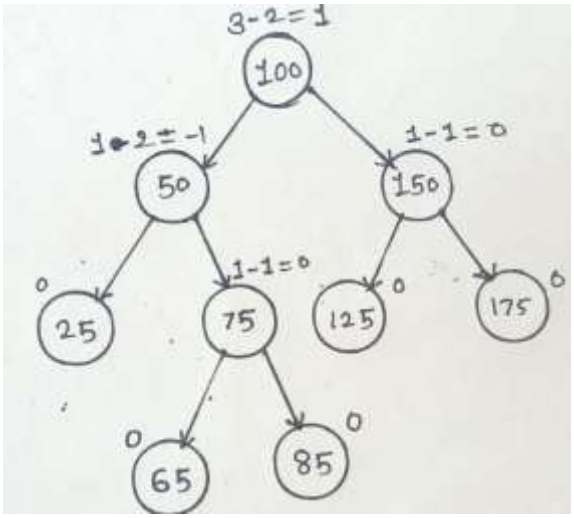


Fig: -13.11

13.6 B Tree

B Tree is a particular m-way tree that can be widely applied for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason 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.

Every node in a B-Tree contains at most m children. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children. The root nodes must have at least 2 nodes. All leaf nodes must be at the same level. A B tree of order 4 is shown in the following image.

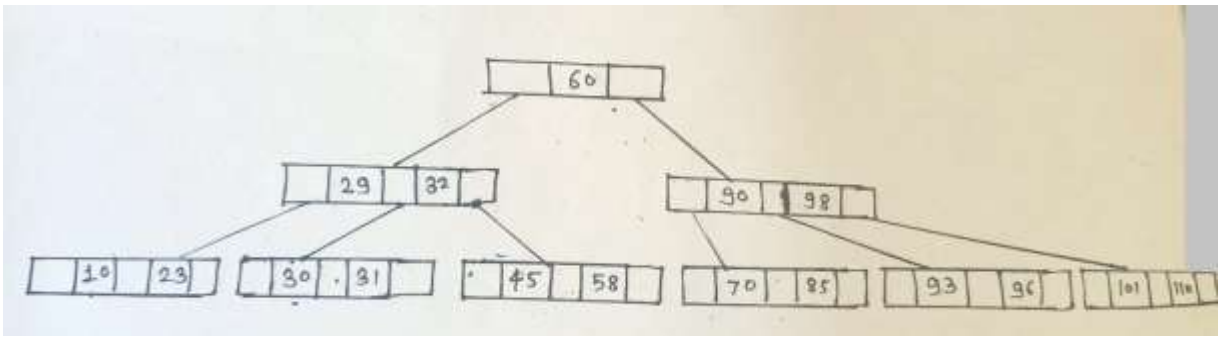


Fig: -13.12

****

Summary:

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes.
- Binary traversals are pre-order, inorder and post order.
- Stack is the approach to traverse tree without recursion
- A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
- Binary search tree (BST) is a special kind of binary tree where each node contains only larger values in its right subtree and Only smaller values in its left subtree.
- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.

Questions:

1. Discuss following with reference to trees. (i) Height of the tree (ii) Complete Binary Tree (iii) Expression tree (iv) Sibling (v) Full Binary Tree
2. What is Binary Tree? Explain Representation of Binary tree. Also explain different operation that can be performed on Binary tree.
3. Explain inorder, preorder and postorder Traversal operation on Binary tree with example.
4. List the types of Binary Search Tree. Explain Insertion and Deletion Operation on Binary Search Tree with Example.
5. What is the meaning of height balanced tree? How rebalancing is done in height balanced tree.
6. Construct a tree for the given in order and post order traversals. In order: DGBAHEICF Post order: GDBHIEFCA.
7. Create a Binary Search Tree for the following data and do in-order, Preorder and Post-order traversal of the tree. 50, 60, 25, 40, 30, 70, 35, 10, 55, 65, 5.
8. What is B-tree?
9. Write short notes on any FOUR: -
    a. Threaded Binary Tree.
    b. Depth First Traversal
    c. AVL tree

Chapter 14 Heap data structure and Sorting

Objectives: -

- ✓ **Heap data structure**
- ✓ **Sorting: - selection, bubble, merge, tree, radix, insertion**

A binary heap is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of applying priority queues. The binary heap was announced by J. W. J. Williams in 1964.

A binary heap is defined as a binary tree with two constraints: -

Form property: a binary heap is a complete binary tree and all levels of the tree, except possibly the last one is fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

Heap property: the key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children, according to some total order.

Max-heaps are the parent key which are greater than or equal to (≥) the child keys and those where it is less than or equal to are called min-heaps. Effective algorithms are known for the two operations needed to implement a priority queue on a binary heap: inserting an element, and removing the smallest or largest element from a min-heap or max-heap, respectively. Binary heaps are also commonly employed in the heapsort sorting algorithm, which is an in-place algorithm because binary heaps can be implemented as an implicit data structure, storing keys in an array and using their relative positions within that array to represent child-parent relationships.

14.1 Heap operations

Both the insert and remove operations modify the heap to adapt to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations have complexity of an O (log n) time.

Insert

To add an element to a heap we must perform an up-heap operation by following this algorithm:

- Add the element to the bottom level of the heap at the most left.
- Compare the added element with its parent; if they are in the correct order, stop.
- If not, swap the element with its parent and return to the previous step.

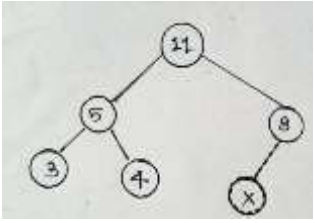


Fig:-14.1

The number of operations required depends only on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a worst-case time complexity of O(log n) but an average-case complexity of O(1).

As an example of binary heap insertion, say we have a max-heap

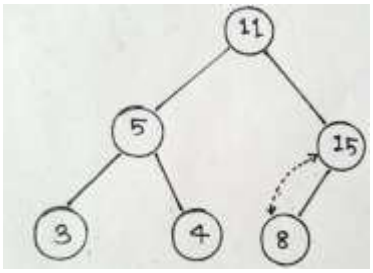


Fig:- 14.2

and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:

However the heap property is still violated since 15 > 11, so we need to swap again:

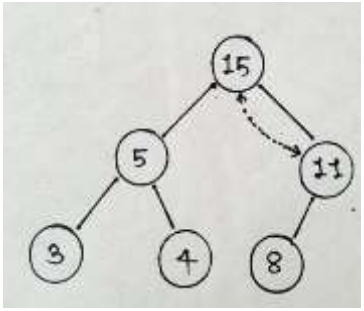


Fig: -14.3

which is a valid max-heap. There is no need to check the left child after this final step: at the start, the max-heap was valid, meaning 11 > 5; if 15 > 11, and 11 > 5, then 15 > 5, because of the transitive relation.

Delete

The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called down-heap.

Replace the root of the heap with the last element on the last level.
Compare the new root with its children; if they are in the correct order, stop.
If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)
So, if we have the same max-heap as before

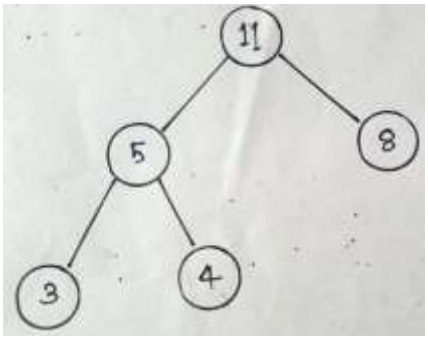


Fig: -14.4

We remove the 11 and replace it with the 4.

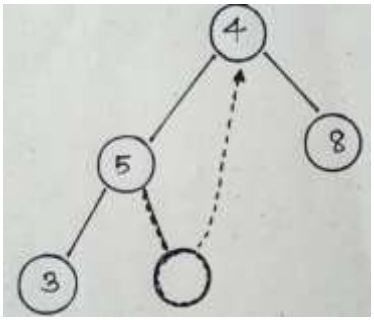


Fig: -14.5

Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:

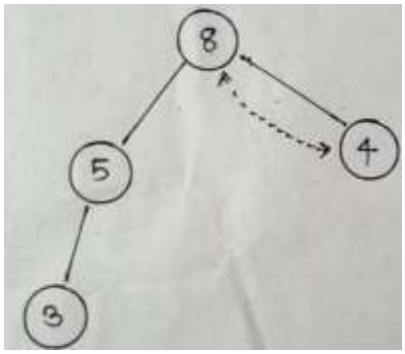


Fig: -14.6

The downward-moving node is swapped with the larger of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position.

14.2 SORTING

Sorting is a process, in which we can perform on data structure in order to arrange data elements in ascending and descending order. There are different methods that can be used to arrange data in ascending or descending order. There are two types of sorting.

Internal sorting:

Internal sorting is used when the data to be sort, all stored in main memory. Bubble sort, selection sort, Insertion sort, Quick sort, Radix sort are internal sorting techniques.

External sorting:

If all the data that is to be sorted do not fit entirely in the main memory, external sorting is required. An external sort requires the use of external memory such as disks or tapes during sorting. In external sorting, some part of the data is loaded into the main memory, sorted using any internal sorting technique and written back to the disk in some intermediate file. This process continues until all the data is sorted. Merge sort is an example of external sorting.

Bubble sort

- The simplest sorting algorithm is bubble sort. A bubble sort is an internal exchange sort. This sorting technique is named bubble because of the logic is similar to the bubble in water. When a bubble are a formed it is small at the bottom and when it moves up it becomes bigger and bigger i.e. bubbles are in ascending order of their size from the bottom to the top.
- The key idea of the bubble sort is to make pairwise comparisons and exchange the position of the pair if they are out of order.
- This sorting method proceeds by scanning through the elements one pair at a time, and swapping any adjacent pairs it finds to be out of order.
- Bubble sort starts by comparing first element with second element; If first element is greater than second element then it will swap both the elements and the move on to the compare second element with third element and so on.
- Let Array[Max] is integer array, N is size of array. Bubble sort is simplest sorting algorithm where after every pass the largest elements of array bubbles up toward the last place/index. If we have total n elements in array then to sort all these elements, we need to repeat this process for n-1 times. In each pass, sorting starts from starting index and end on (N-pass-1) element.

Bubble Sort Algorithm

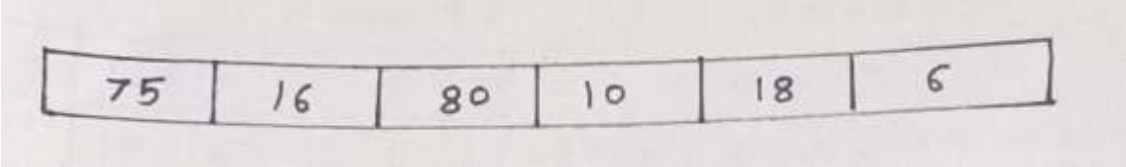
- Let Array[Max] is array, N is size of array.
    1. Repeat for pass=1 to N-1.
    2. Repeat for j=0 to N-pass-1.
    3. If array[j]>array[j+1].
    4. Interchange a[j] with a[j+1].
    5. Print sorted Array.
- Time complexity of bubble sort in worst case is O(N2), in Best case is O(N) and in Average case is O(N2).

  Program:- Bubble sort

  Input:-array of numbers

  Output:- numbers arranged in ascending order.

Bubble sort example: - consider the following numbers



Pass 1: compare the elements j and j+1 and do swap / no swap operations

Pass	j	Elements						operation
1	0	75	16	80	10	18	6	swap 75 and 16.
	1	16	75	80	10	18	6	No swap for 75 and 80.
	2	16	75	80	10	18	6	swap 80 and 10.
	3	16	75	10	80	18	6	swap 80 and 18.
	4	16	75	10	18	80	6	swap 80 and 6.
		16	75	10	18	6	80	Pass 1 finish.

Table 14.1

At the end of pass first largest element is filtered and placed at the end

Pass 2:

Compare the elements j and j+1 and do swap / no swap operations

Pass	j	Elements						operation
2	0	16	75	10	18	6	80	No swap for 16 and 75.
	1	16	75	10	18	6	80	swap 75 and 10.
	2	16	10	75	18	6	80	swap 75 and 18.
	3	16	10	18	75	6	80	swap 75 and 6.
		16	10	18	6	75	80	Pass 2 finish.

Table 14.2

At the end of pass second largest element is filtered and placed at its proper place

Pass 3:

Pass	j	Elements						operation
3	0	16	10	18	6	75	80	swap 16 and 10.
	1	10	16	18	6	75	80	No swap for 16 and 18.
	2	10	16	18	6	75	80	swap 18 and 6.
		10	16	6	18	75	80	pass 3 Finish.

Table 14.3

At the end of pass third largest element is filtered and placed at the end

Pass 4:

Pass	j	Elements						operation
4	0	10	16	6	18	75	80	No swap for 10 and 16.
	1	10	16	6	18	75	80	swap 16 and 6.
		10	6	16	18	75	80	pass 4 finish.

Table 14.4

At the end of pass fourth largest element is filtered and placed at the end

Pass 5:

Pass	j	Elements						operation
5	0	10	6	16	18	75	80	swap 10 and 6.
		6	10	16	18	75	80	pass 5 finish.

Table 14.5

At the end of pass fifth largest element is filtered and placed at the end and finally sorted

Advantages of Bubble sort

1) Bubble sort is easy and simple to implement.
2) Elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

Disadvantages of Bubble sort

1) Bubble sort is slow as it takes O(n!) comparison to complete sorting.
2) It is not sufficient for large lists.

Bubble sort program: -

// Java program for implementation of Bubble Sort

```
class BubbleSort
```

```
{
     void bubbleSort(int arr[])
     {
          int n = arr.length;
          for (int i = 0; i < n-1; i++)
               for (int j = 0; j < n-i-1; j++)
                    if (arr[j] < arr[j+1])
                    {
                         // swap temp and arr[i]
                         int temp = arr[j];
                         arr[j] = arr[j+1];
                         arr[j+1] = temp;
                    }
     }

     /* Prints the array */
     void printArray(int arr[])
     {
          int n = arr.length;
          for (int i=0; i<n; ++i)
               System.out.print(arr[i] + " ");
          System.out.println();
     }

     // Driver method to test above
     public static void main(String args[])
     {
          BubbleSort ob = new BubbleSort();
          int arr[] = {4, 3, 5, 1, 2, 1, 9};
          ob.bubbleSort(arr);
          System.out.println("Sorted array");
          ob.printArray(arr);
     }
}
```

Output:- run:
Sorted array
9 5 4 3 2 1 1
BUILD SUCCESSFUL (total time: 0 seconds)

Selection sort

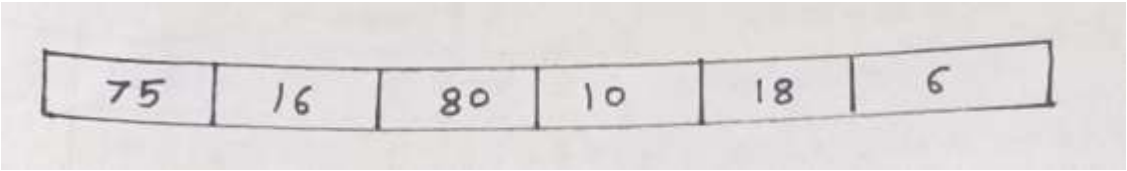
- Selection sort performs in-place comparison which means that the array is divided into two parts, the sorted part at the left end and the unsorted part at the right end Initially, the sorted part is empty and the unsorted part is the entire list.

- During Iteration 1, smallest element in the array (index 0 to last index) is selected and that selected element is replaced with first position element.
- During Iteration 2 smallest element from sub array (starting from index 1 to last index) is selected and that selected element is replaced with second position element and so on.
- In other words , the idea of the selection sort is to search the smallest element in the list and exchange it with the element in the first position, Then, find the second smallest element and exchange it with the element in the second position, and so on until the entire array is sorted.
- In selection sort, iteration 1 looks for smallest element in the array and replace first position with that smallest element. After that iteration 2 look for smallest element present in the sub array, starting from index 1, till the last index and replace second position with that smallest element. This is repeated, until the array is completely sorted.

Selection sort algorithm

- Let array {Max} integer array, N is size of array.
  1. Repeat for i=0 to N-2.
  2. Set min=Array[i] and loc=i.
  3. Repeat for j=i+1 toN-1.
  4. IF MIN>Array[j] then.
  5. Set MIN=[j] and loc=j;
  6. Interchange Array [j] with Array [loc].
- Time complexity of selection sort in worst case is O (N2), in best case is O(N2) and in average case is O(N2).

Selection sort example:- consider the following numbers



Following procedure is used:

Pass	I/o	Elements						operation
Pass 1	Input	75	16	80	10	18	6	MIN=6 swap 75 with 6.
	output	6	16	80	10	18	75	After pass 1 smallest element i.e. 6 is at its proper position.
Pass 2	Input	6	16	80	10	18	75	MIN = 10 swap 16 with 10.
	output	6	10	80	16	18	75	After pass 2 second smallest element i.e. 10 is at its proper position.
Pass 3	Input	6	10	80	16	18	75	MIN =16 swap 80 with 16.
	output	6	10	16	80	18	75	After pass 316 is at its proper position.
Pass4	Input	6	10	16	80	18	75	MIN=18 swap 80 with 18.
	output	6	10	16	18	80	75	After pass 418 is at its proper position.
Pass 5	Input	6	10	16	18	80	75	MIN=75 swap 80 with 75.
	output	6	10	16	18	75	80	After pass 5 75 is at its proper position

Table 14.6

Selection sort program:-

```
// Java program for implementation of Selection Sort
class SelectionSort
{
    void sort(int arr[])
    {
        int n = arr.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] > arr[min_idx])
                    min_idx = j;

            // Swap the found minimum element with the first
            // element
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
```

```
    // Prints the array
    void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver code to test above
    public static void main(String args[])
    {
        SelectionSort ob = new SelectionSort();
        int arr[] = {4,5,1,2,1,9};
        ob.sort(arr);
        System.out.println("Sorted array");
        ob.printArray(arr);
    }
}
```

run:
Sorted array
9 5 4 2 1 1
BUILD SUCCESSFUL (total time: 0 seconds)

Advantages of selection sort.

1. Selection sort is simple and easy to implement.
2. It gives 60% performance improvement over bubble sort.

Disadvantages of selection sort.

Selection sort performs poor when dealing with large number of elements.

Insertion Sort

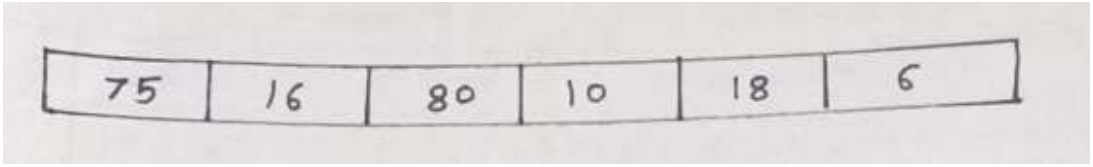
- Insertion sort is different from other sorting algorithms as sorting starts from the index 1(not0), i.e. the element at index1compared with index0 first. If element at index 1 is smaller is smaller than element at index 0 then we insert the element at index 1 to index 0.

Insertion sort algorithm

- Let array [Max] is integer array , N is size of array

1. Repeat for i=to N-1.
2. Set temp=**A**rray[i]and j=i-1.
3. While j >=0 and array [j] >temp do.
4. Array [j+1] = Array[j].
5. J=j-1.
6. Array [j+1]=temp.

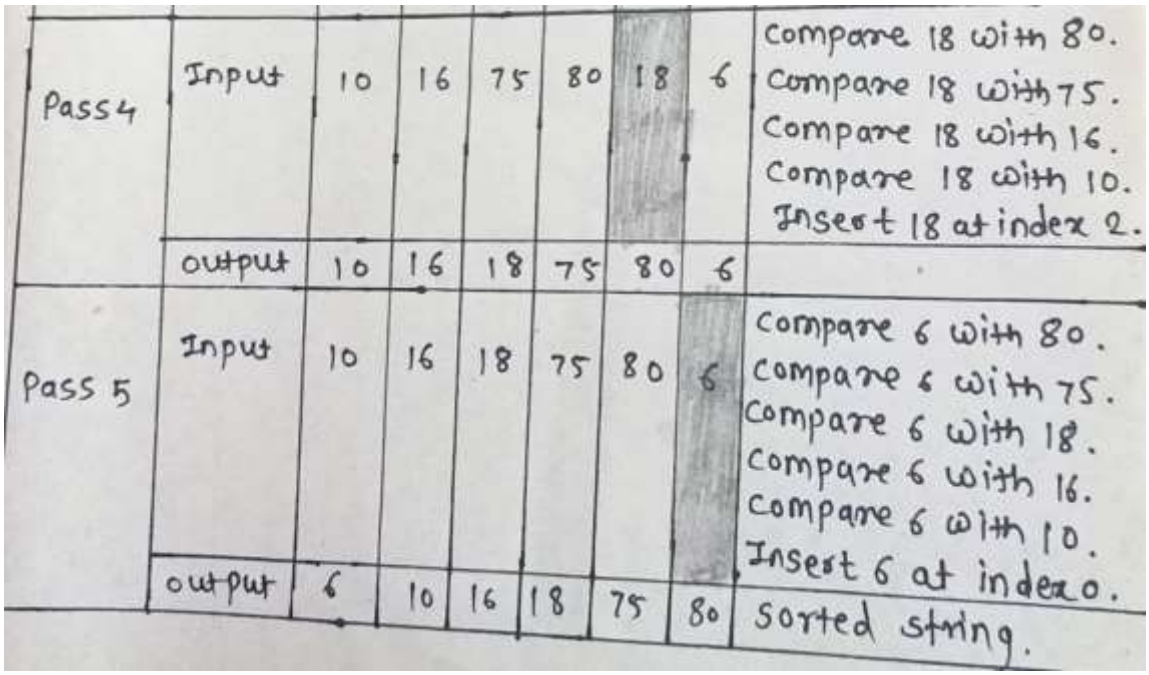
Insertion sort example:- consider the following numbers



Following procedure is used:

	I/O	Elements						operation
Pass 1	Input	75	16	80	10	18	6	Compare 16 with 75 and insert 16 at index 0.
	output	16	75	80	10	18	6	
Pass 2	INPUT	16	75	80	10	18	6	compare 80 with 75, no change. compare 80 with 16 no change.
	output	16	75	80	10	18	6	
Pass 3	Input	16	75	80	10	18	6	Compare 10 with 80. Compare 10 with 75. Compare 10 with 16. Insert 10 at index 0.
	output	10	16	75	80	18	6	

Table 14.7



Pass 4	Input	10	16	75	80	18	6	Compare 18 with 80. Compare 18 with 75. Compare 18 with 16. Compare 18 with 10. Insert 18 at index 2.
	output	10	16	18	75	80	6	
Pass 5	Input	10	16	18	75	80	6	Compare 6 with 80. Compare 6 with 75. Compare 6 with 18. Compare 6 with 16. Compare 6 with 10. Insert 6 at index 0.
	output	6	10	16	18	75	80	Sorted string.

Table 14.8

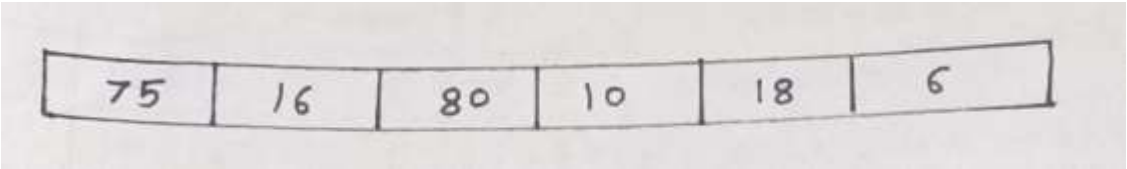
Insertion sort program:

```
// Java program for implementation of Insertion Sort
class InsertionSort {
     /*Function to sort array using insertion sort*/
     void sort(int arr[])
     {
          int n = arr.length;
          for (int i = 1; i < n; ++i) {
               int key = arr[i];
               int j = i - 1;

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

     /* A utility function to print array of size n*/
```

```
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6 };

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);

        printArray(arr);
    }
}
```

run:
5 6 11 12 13
BUILD SUCCESSFUL (total time: 0 seconds)

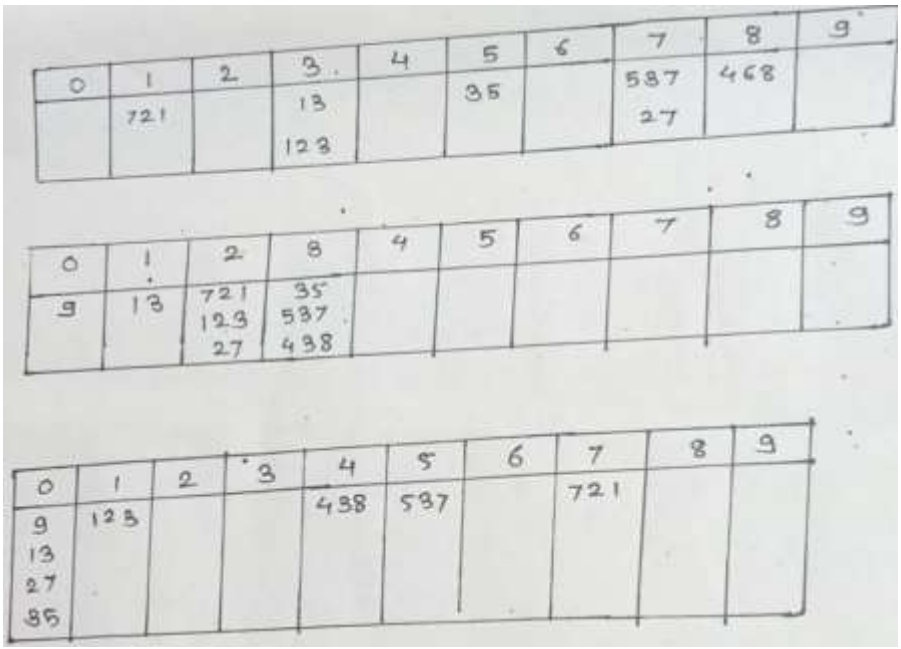
Radix sort example:- consider the following numbers



Pass 1: Bucket sort the numbers while scanning the input from left to right one's digit (LSD).
At the end of this pass 1 remove the elements from the buckets from 0 to 9 in first out manner.
The elements will be 721, 13, 123, 35, 537, 27, 438, and 9

Pass 2: now it will scan as per pass 1, but the ten's place (i.e. next LSD)
At the end of this pass 2 remove the elements from the buckets from 0 to 9 in first out manner.
The elements will be 9,13,7,21,123,27,35,537,438.

Pass 3: now it will scan as per pass 1, but the hundred's place (i.e. next MSD)
At the end of this pass 3 remove the elements from the buckets from 0 to 9 in first out manner.
The elements will be 9,13,27,35,123,438,537,721. And the numbers are sorted.



0	1	2	3	4	5	6	7	8	9
	721		13 123		35		587 27	468	

0	1	2	8	4	5	6	7	8	9
9	13	721 123 27	35 587 438						

0	1	2	3	4	5	6	7	8	9
9 13 27 35	123			438	587		721		

Table 14.9

Radix program:

```
// Radix sort Java implementation
import java.io.*;
import java.util.*;

class Radix {

     // A utility function to get maximum value in arr[]
     static int getMax(int arr[], int n)
     {
          int mx = arr[0];
          for (int i = 1; i < n; i++)
               if (arr[i] > mx)
                    mx = arr[i];
          return mx;
     }

     // A method to do counting sort of arr[] according to
     // the digit represented by exp.
     static void countSort(int arr[], int n, int exp)
     {
          int output[] = new int[n]; // output array
          int i;
          int count[] = new int[10];
```

```
        Arrays.fill(count,0);

        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
            count[ (arr[i]/exp)%10 ]++;

        // Change count[i] so that count[i] now contains
        // actual position of this digit in output[]
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];

        // Build the output array
        for (i = n - 1; i >= 0; i--)
        {
            output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
            count[ (arr[i]/exp)%10 ]--;
        }

        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to current digit
        for (i = 0; i < n; i++)
            arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
static void radixsort(int arr[], int n)
{
        // Find the maximum number to know number of digits
        int m = getMax(arr, n);

        // Do counting sort for every digit. Note that instead
        // of passing digit number, exp is passed. exp is 10^i
        // where i is current digit number
        for (int exp = 1; m/exp > 0; exp *= 10)
            countSort(arr, n, exp);
}

// A utility function to print an array
static void print(int arr[], int n)
{
```

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


    /*Driver function to check for above function*/
    public static void main (String[] args)
    {
        int arr[] = {770, 425, 175, 90, 802, 214, 12, 166};
        int n = arr.length;
        radixsort(arr, n);
        print(arr, n);
    }
}
```

run:

12 90 166 175 214 425 770 802 BUILD SUCCESSFUL (total time: 0 seconds)

Tree Sort

Algorithm: Input: -array of numbers

Output: -sorted array of numbers

1. Input the numbers
2. Transform the unsorted list into a binary search tree
3. Traverse the resultant binary search tree in order

Example: consider the numbers 27,48,13,50,39,77,82,91,65,19,70,66

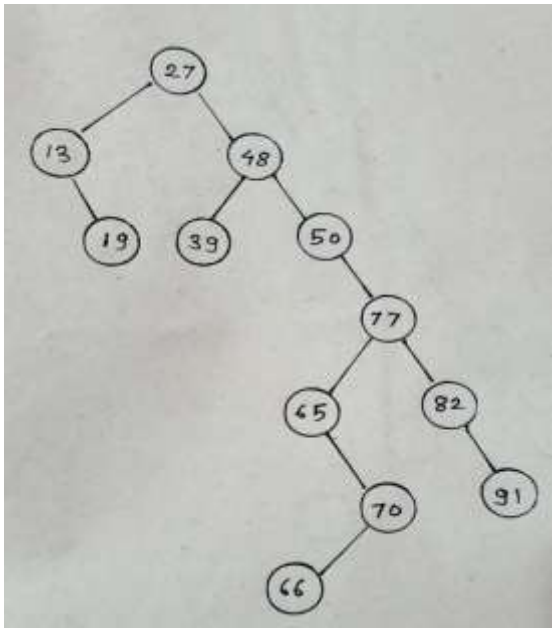


Fig: -14.7

Program:-

```
// Java program to
// implement Tree Sort
class TreeSort
{

      // Class containing left and
      // right child of current
      // node and key value
      class Node
      {
            int key;
            Node left, right;

            public Node(int item)
            {
                  key = item;
                  left = right = null;
            }
      }

      // Root of BST
      Node root;

      // Constructor
      TreeSort()
      {
            root = null;
      }

      // This method mainly
      // calls insertRec()
      void insert(int key)
      {
            root = insertRec(root, key);
      }

      /* A recursive function to
      insert a new key in BST */
      Node insertRec(Node root, int key)
```

```
{

     /* If the tree is empty,
     return a new node */
     if (root == null)
     {
          root = new Node(key);
          return root;
     }

     /* Otherwise, recur
     down the tree */
     if (key < root.key)
          root.left = insertRec(root.left, key);
     else if (key > root.key)
          root.right = insertRec(root.right, key);

     /* return the root */
     return root;
}

// A method to do
// inorder traversal of BST
void inorderRec(Node root)
{
     if (root != null)
     {
          inorderRec(root.left);
          System.out.print(root.key + " ");
          inorderRec(root.right);
     }
}
void treeins(int arr[])
{
     for(int i = 0; i < arr.length; i++)
     {
          insert(arr[i]);
     }

}
```

```
	// Driver Code
	public static void main(String[] args)
	{
		TreeSort tree = new TreeSort();
		int arr[] = {5, 4, 7, 2, 11};
		tree.treeins(arr);
		tree.inorderRec(tree.root);
	}
}
```

run:

2 4 5 7 11 BUILD SUCCESSFUL (total time: 0 seconds)

Advantages - Tree Sort

1. The main advantage of tree sort algorithm is that we can make changes very easily as in a linked list.
2. Sorting in Tree sort algorithm is as fast as quick sort algorithm.

Disadvantages - Tree Sort

1. The worst case occur when the elements in an array is already sorted.
2. In worst case, the running time of tree sort algorithm is 0 (n2).

MergeSort:

Algorithm:Input:- array of numbers

Output:-numbers arranged in ascending order

1. MergeSort(array A, int p, int r) {
2. if (p < r) { // we have at least 2 items
   q = (p + r)/2
3. MergeSort(A, p, q) // sort A[p..q]
4. MergeSort(A, q+1, r) // sort A[q+1..r]
5. Merge(A, p, q, r) // merge everything together
   }
   }

Example: consider 38, 27,43,3,9,82 and 10

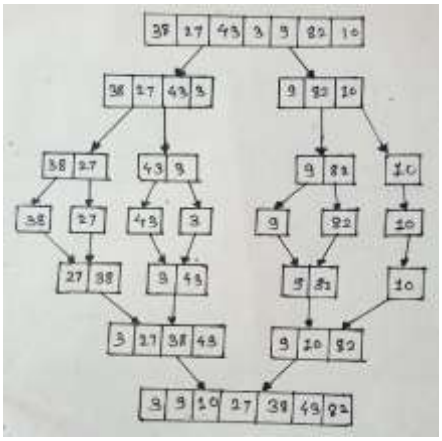


Fig: -14.8

```
/* Java program for Merge Sort */
class MergeSort
{
     // Merges two subarrays of arr[].
     // First subarray is arr[l..m]
     // Second subarray is arr[m+1..r]
     void merge(int arr[], int l, int m, int r)
     {
          // Find sizes of two subarrays to be merged
          int n1 = m - l + 1;
          int n2 = r - m;

          /* Create temp arrays */
          int L[] = new int [n1];
          int R[] = new int [n2];

          /*Copy data to temp arrays*/
          for (int i=0; i<n1; ++i)
               L[i] = arr[l + i];
          for (int j=0; j<n2; ++j)
               R[j] = arr[m + 1+ j];


          /* Merge the temp arrays */

          // Initial indexes of first and second subarrays
          int i = 0, j = 0;

          // Initial index of merged subarry array
          int k = l;
          while (i < n1 && j < n2)
```

```
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr , m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
```

```
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};

        System.out.println("Given Array");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length-1);

        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

Advantages - Merge Sort

1. Merge sort algorithm is best case for sorting slow-access data e.g) tape drive.
2. Merge sort algorithm is better at handling sequential - accessed lists.

Disadvantages - Merge Sort

1. The running time of merge sort algorithm is 0(n log n) which turns out to be the worse case.
2. Merge sort algorithm requires additional memory space of 0(n) for the temporary array TEMP.

****

Summary:

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then key(α) ≥ key(β).
- The sorting algorithms are used to arrange data in order and we use array as a data storage structure.
- All the algorithms in this chapter execute in O (N^2) time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient, but the simplest, sort.
- The insertion sort is the most commonly used of the O (N^2) sorts described in this chapter.
- None of the sorts in this chapter require more than a single temporary variable, in addition to the original array.

Questions: -

1. What is Heap data structure? Explain operations on it.
2. Arrange the following numbers in ascending as well as descending order with the help of the specified algorithms and show each pass in detail.
   Numbers: - 123,456,200,23,56,12
   Algorithms: -
   I. Bubble sort
   II. Selection
   III. Merge
   IV. Tree
   V. Radix
   VI. Insertion
3. Write the short note on the above (ques 2) sorting algorithms with example.
4. Write down the complexity for any 5 algorithms from above (ques 2).
5. Write the advantages and disadvantages of any 3 algorithms from above (ques 2).
6. Write down the algorithm for any 3 algorithms from above (ques 2).

Unit 6 Chapter 2 Graphs

Objectives:-

- ✓ **Graphs - graph theory**
- ✓ **Sequential representation**
- ✓ **Adjacency matrix**
- ✓ **Path matrix**
- ✓ **Warshall's algorithm**
- ✓ **Linked representations**
- ✓ **Operations**
- ✓ **Traversing**

15.1 Graph theory

- Graph is one of the most important non-linear data structure. It is used to describe a wide variety of relationships between object and in practice can be related to almost everything.
- In last section, we have studied tree data structure, which is a special class of a graph. In tree structure there exist a hierarchical relationship between parent and children and there exist a unique path between every pair of vertices.
- In a graph there may exist more than one path between any two vertices. Graphs have many applications in subjects as diverse as sociology, chemistry, geography, mathematical electrician, Computer engineering.
- A graph is a pictorial representation of the relationship between entities. A graph G is collection of nodes which are called as Vertices V, connected in pairs by line segment, called as Edges E. Sets of vertices are represented as V(G)and sets of Edges are represented as E(G). So, we can represent a graph as G (V, E).
- Fig.15.1 shows an example of graph.

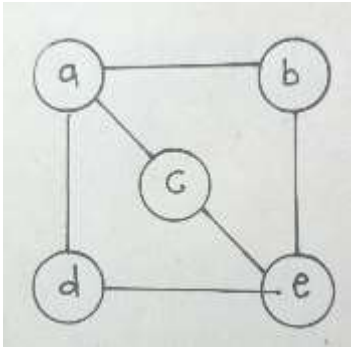


Fig: -15.1

- Fig 15.2 shows real word example of graph. This one represents the courses and the order in which they must be taken to complete a major in IT at University College.

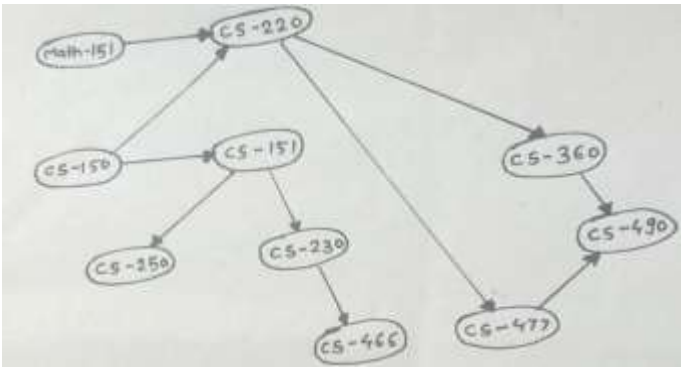


Fig:-15.2

- A graph is non-linear data structure. A graph consists of a set of non-empty vertices, together with a set of edges, and each edge joins two different vertices. A path is sequence of distinct vertices each adjacent to the next, except possibly the first vertex and last vertex is different.
- There are two types of graphs namely Directed graph and Undirected graph as explained below:

1. Directed Graph:

- If an edge between any two nodes is directionally oriented, a graph is referred as a directed graph or digraph.
- We can define directed graph as "a graph whose pairs are ordered is called a directed graph" or "a graph in which each edge is directed is called directed graph".
- Fig.15.3 shows an example or directed graph.

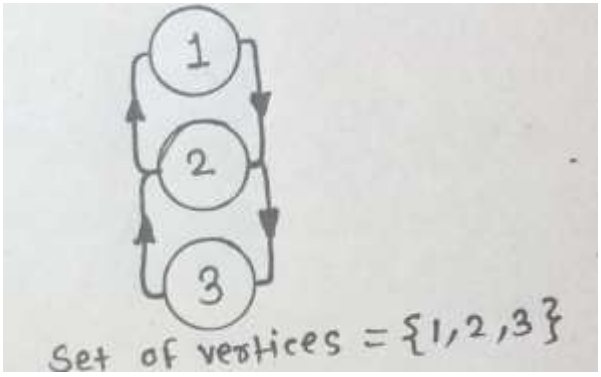


Fig.15.3

2. Undirected Graph:

- If an edge between any two nodes is not directionally oriented a graph is referred as an undirected graph or unqualified graph.
- We can define undirected graph as "If a graph is not ordered, it is called an unordered graph, or just a graph." OR "A graph in which each edge is undirected is called an undirected graph."
- Fig.15.4 shows an example of undirected graph.

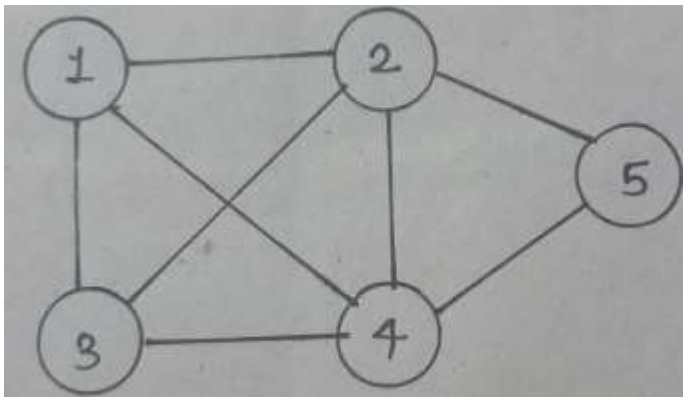


Fig.15.4

- In the following Fig.15.5 the graphs G1 and G2 are undirected graphs and G3 is a directed graph. The graph G2 is also a tree while G1 and G2 are not. Trees are the special case of graphs.

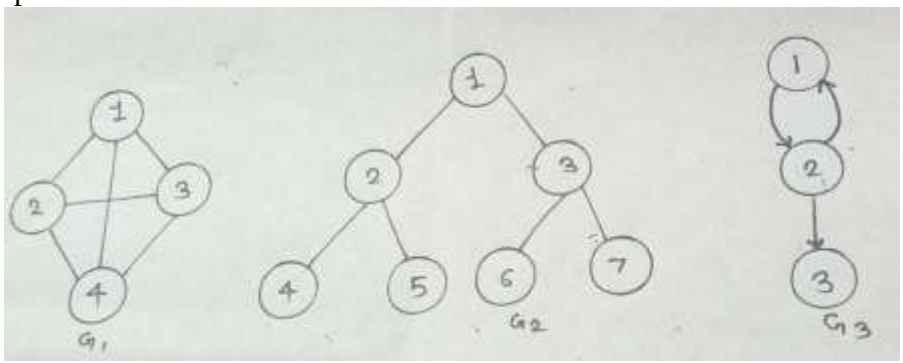


Fig.15.5

V (G1) = {1,2,3,4};
V (G2) = {1,2,3,4,5,6,7};
V(G3) = {1,2,3};
E(G1) = {(1,2), (1,3), (1,4) ,(2,3),(2,4),(3,4)}
E(G2) = {(1,2), (1,3), (2,4) ,(2,5),(3,6),(3,7)}
E(G3) = {<1,2>, <2,1>, <2,3>}

Graph operations and Terminology

- Terminology of graph is given below:

1. Adjacent Node: When there is an edge from one node to another then these nodes are called adjacent node. EXAMPLE: Node 1 is called adjacent to node 2 as there exists an edge from node 1 and node 2 as shown in Fig.6.5.6. Node 2 is called successor of node 1 and node 1 is predecessor of 2.The set of nodes adjacent to node 1 are called neighbors of node 1.Node 2 and node 3 are neighbors of node 1.

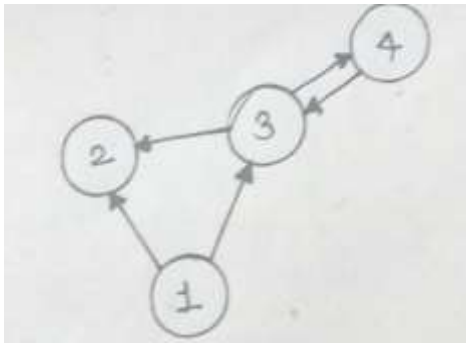


Fig.15.6

2. Incidence: An edge is incident with a vertex if the edge is joined to the vertex.
3. Length of path: The number of edges appearing in the sequence of the path is called length of path. The path length is equal to the number of edges present In a path.
4. Degree of Node: The number of edges connected directly to the node is called as degree of node. In other words, a degree of a node is the number of edges containing that node.In Fig.15.7 degree of node 10 is 2 and node 20 is 2, node 30 is 4 and node 40 is 2.

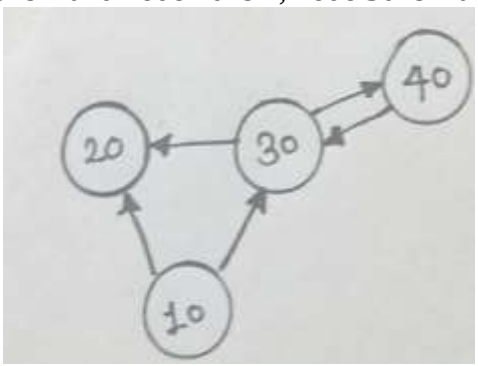


Fig.15.7

5. Indegree: The number edges pointing towards the node are called in-degree/in-order. In Fig. 15.7, the In degree of node 1 is 0 and in degree of node 2 is 2.
6. Outdegree: The number edges pointing away from the node are called out-degree/out-order. In Fig.15.7, out degree of node 1 is 2 and out degree of node 2 is 0.
7. Source: A node which has only outgoing edges and no incoming edges is called source. In Fig.15.7, node 1 is source node.
8. Sink: A node having only incoming edges and no outgoing edges is called sink node. In Fig.15.7node 2 is a sink node.
9. Pendant Node: When in-degree of node is one and out-degree is zero then such a node is called pendant node.
10. Reachable: If a path exists between two nodes, it will be called reachable from one node to another node.
11. Articulation point: If on removing the node the graph gets disconnected, then that node is called the articulation point. In Fig,15.7 node 3 is an articulation point.
12. Biconnected Graph: The biconnected graph is the graph which does not contain any articulation point.
13. Sling or Loop: An edge of a graph, which joins a node to itself, is called a sling or loop. Fig.15.8 shows an example of loop.

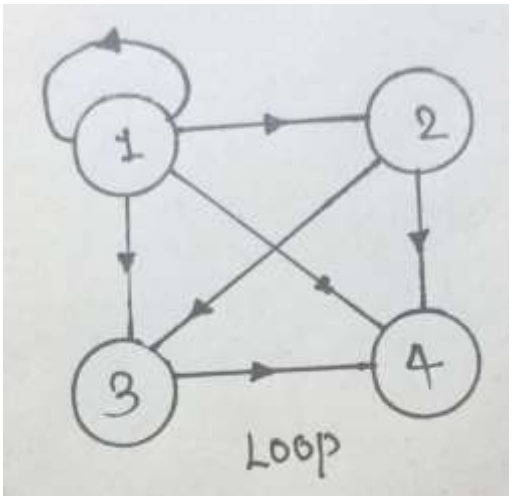


Fig.15.8

14. Parallel Edges: The two distinct edges between a pair to nods which are opposite in direction are called as parallel edges. Fig. 15.9 shows parallel edges in a graph.

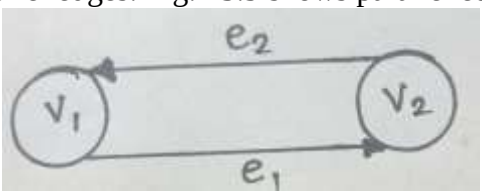


Fig.15.9

15. Weighted Edges: An edge which has a numerical value assigned to it is called weighted edge.Fig.15.10 weighted edges in a graph.

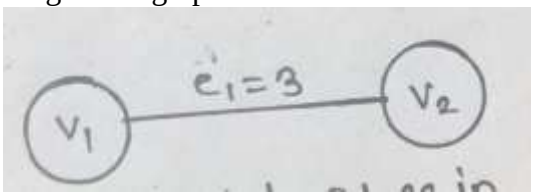


Fig.15.10

REPRESENTATIONS OF GRAPH

- Representation of graph is process to store the graph data into the computer memory. Graphs can be implemented in following two ways:
    1. Using arrays as a adjacency matrix, (Sequential Representation), and
    2. Linked list as adjacency list (Linked Representation).

15.2 Sequential Representation of a Graph

- Graph can be represented through matrix (array) in computer systems memory. This is sequential in nature. This type of representation is called sequential representation of graphs.

- A graph can be comfortably represented using an adjency matrix implemented through a two-dimensional array. This representation of graph is called as static representation (static graph).
- Graphs can be represented using matrices. Adjacency matrix is n x n matrix A. Each element of 'A' is either zero or one.

Advantages of Array representation of Graph:

1. Simple and easy to understand.
2. Graphs can be constructed at run-time.
3. Efficient for dense (lots of edges) graphs.
4. Simple and easy to program.
5. Most of the implicit graph problems can be easily represented as Adjacency matrix with little or no pre-processing.
6. Adapts easily to different kinds of graph.

Disadvantages of Array representation of Graph:

1. Adjacency matrix consumes huge amount of memory for storing big or large graphs.
2. Adjacency matrix requires huge efforts for adding/removing a vertex.
3. The matrix representation of graph does not keep track of the information related to the nodes.

4. Requires that graph access be a command rather than a computation.
5. The excessive number of empty cells also reduces the efficiency of some of the Traversal algorithms.

15.3 Adjacency List Implementation of a Graph

- There are many ways of representing the linked structure of graph. The information and edges must be stored in the node structure. This is true, irrespective of the structure we use. Consider the graph shown in fig.
- Where a, b, c, d, e represents the key values. If we wish to store the values of the nodes, then we can simply use the node-list, as a linked structure as shown in fig.
- Now to have the edge record is by maintaining the edge list. The edge is to be formed for each node. An edge list is a linked list of all the nodes adjacent to given node.
- Thus, for node 'd', the adjacent nodes are 'c' and 'e'. Hence there will be two elements on the edge list of 'd'. fig represents the adjacency list for the graph shown in Fig.
  Example: consider the undirected graph in Fig and provide adjacency list.
  Solution: The adjacency list is given in Fig.

Advantages of Linked List representation of Graph:

1. Less storage for sparse (few edges) graphs.

2. Easy to store additional information in the data structure like vertex degree, edge weight etc.
3. Better space usage.
4. Better graph traversal times.
5. Generally better for most algorithms.

Disadvantages of Linked List representation of Graph:

1. Generally, takes some pre-processing to create Adjacency list.
2. Algorithms involving edge creation, deletion and querying edge between two vertices are better way with matrix representation than the list representation.
3. Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix.

15.4 Adjacency matrix representation

The size of the matrix is V X V where V is the number of vertices in the graph and the value of an entry A_{ij} is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.

Adjacency Matrix Example

The image below shows a graph and its equivalent adjacency matrix.

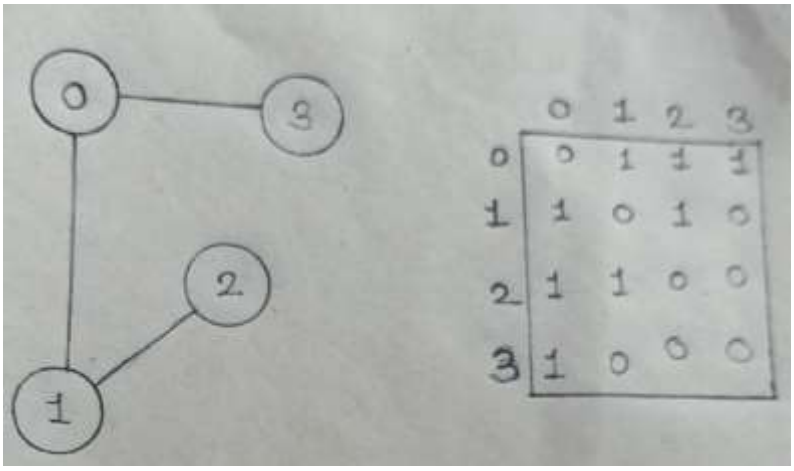


Fig.15.11

In case of undirected graph, the matrix is symmetric about the diagonal because of every edge (i,j), there is also an edge (j,i)

15.5 Path Matrix

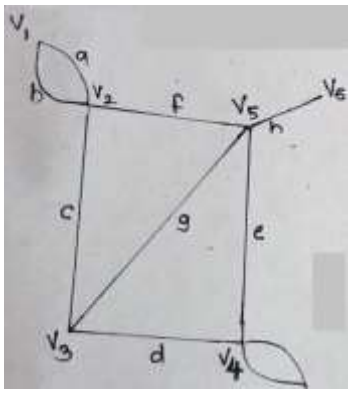
It can be defined as an open walk in which no vertex/edge can appear twice.

If edge of graph is a part of given path then put 1 else put 0.

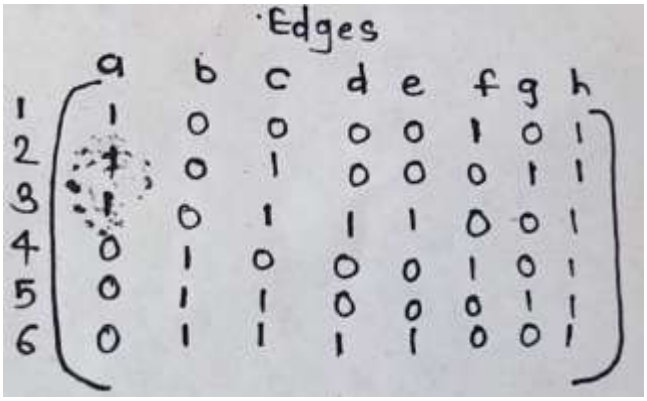
P(Vj,Vi)=1, if edge is on path

=0, otherwise

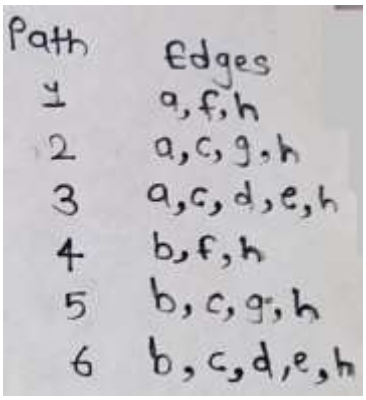
Example: - from v1 to v6



(a)



(b)



(c)

Fig.15.12 a, b & c

15.6 Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. It works for both the directed and undirected weighted graphs. But it not applicable for the graphs with negative cycles.

Let the given graph be:

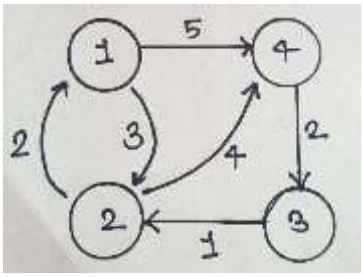


Fig.15.13

Steps to find the shortest path between all the pairs of vertices:

1. Create a matrix A^1 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.
   Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

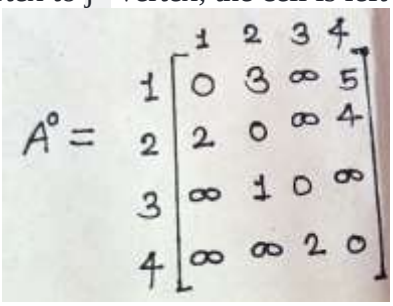


Fig.15.14

2. Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

   Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

   That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

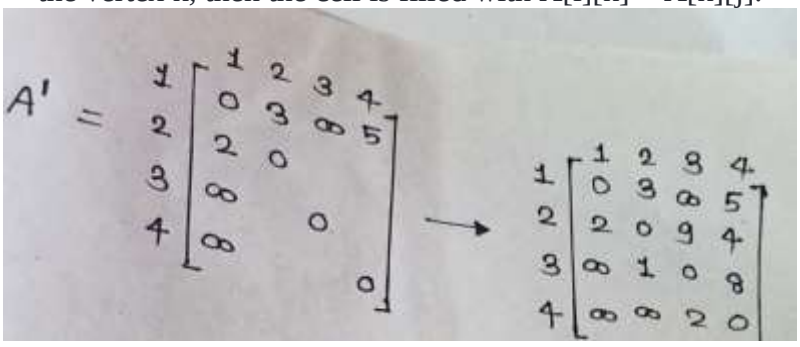


Fig.15.15

For example: For A^1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (i.e. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A^0[2, 4] is filled with 4.

3. In a similar way, A^2 is created using A^3. The elements in the second column and the second row are left as they are.
   In this step, k is the second vertex. The remaining steps are the same as in **step 2**.

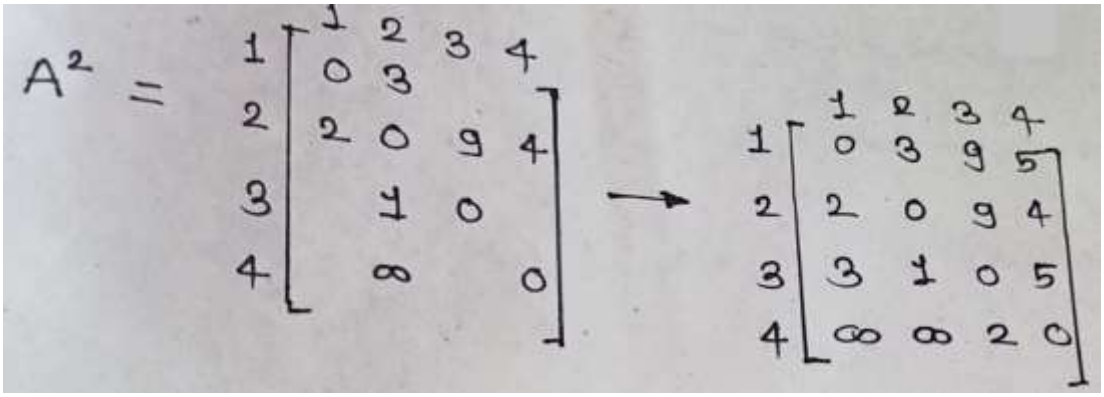


Fig.15.16

Similarly, A^3 and A^4 is also created.

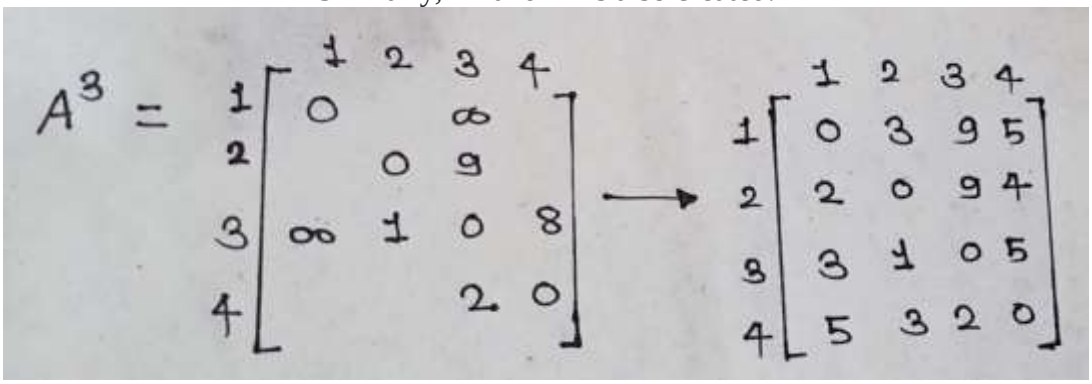


Fig.15.17

4.

5. A^4 gives the shortest path between each pair of vertices.

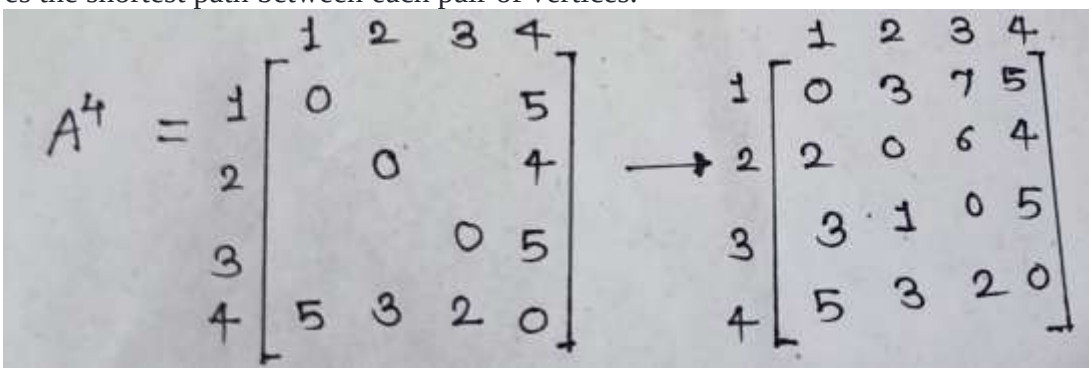


Fig.15.18

Algorithm: Input:- graph with distances
Output:-shortest path

1. n = no of vertices
2. A = matrix of dimension n*n
3. for k = 1 to n
   for i = 1 to n
   for j = 1 to n
   Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
4. return A

Program:-

```
import java.io.*;
class FloydWarshall {
  final static int INF = 999, nV = 4;
  void floydWarshall(int graph[][]) {
    int A[][] = new int[nV][nV];
    int i, j, k;
    for (i = 0; i < nV; i++)
      for (j = 0; j < nV; j++)
        A[i][j] = graph[i][j];
    for (k = 0; k < nV; k++) {
      for (i = 0; i < nV; i++) {
        for (j = 0; j < nV; j++) {
          if (A[i][k] + A[k][j] < A[i][j])
            A[i][j] = A[i][k] + A[k][j];
        }
      }
    }
    printMatrix(A);
  }
  void printMatrix(int A[][]) {
    for (int i = 0; i < nV; ++i) {
      for (int j = 0; j < nV; ++j) {
        if (A[i][j] == INF)
          System.out.print("INF ");
        else
          System.out.print(A[i][j] + "  ");
      }
      System.out.println();
    }
  }
  public static void main(String[] args) {
    int graph[][] = { { 0, 3, INF, 5 },
             { 2, 0, INF, 4 },
             { INF, 1, 0, INF },
             { INF, INF, 2, 0 } };
    FloydWarshall a = new FloydWarshall();
    a.floydWarshall(graph);
  }
}
```

Output:-

run:

0 3 7 5

2 0 6 4

3 1 0 5

5 3 2 0

BUILD SUCCESSFUL (total time: 0 seconds)

15.7 Linked list representations

- Following graph consists of the linked list representation of 4 nodes 1,2,3,& 4 with 6 edges as shown in the following fig:-15.19.

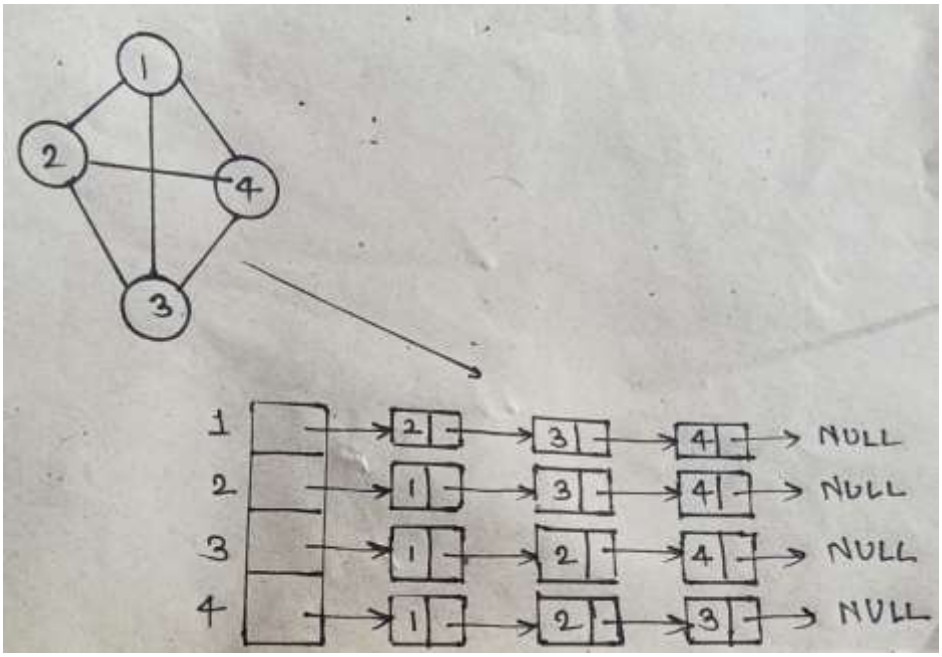


Fig.15.19

15.8 Graph Traversal:

- Traversing a graph means visiting all the vertices in a graph exactly one. In other words exploring each vertex of the graph is known as traversing a graph. The two common graph traversal methods are Breadth First Search (BFS) and Depth First Search (DFS).

BFS (Breadth First Search)

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
Step 5 - Repeat steps 3 and 4 until queue becomes empty.
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

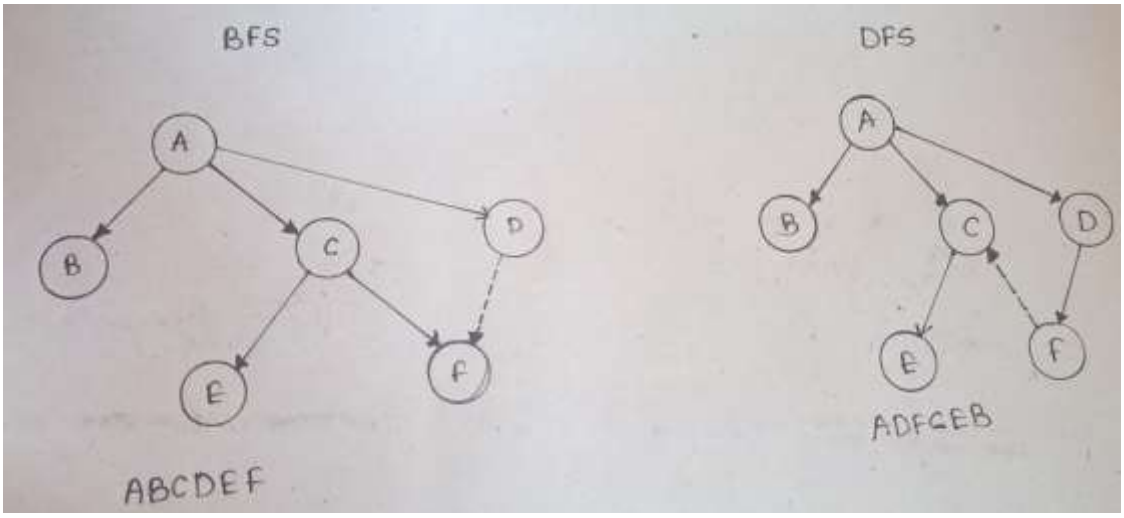


Fig.15.20

DFS (Depth First Search)
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.
We use the following steps to implement DFS traversal.

- Step 1 - Define a Stack of size total number of vertices in the graph.
- Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
- Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.
- Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

****

Summary: -

- 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.
- In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges.
- In adjacency matrix, the rows and columns are represented by the graph vertices.
- An adjacency matrix is a square matrix used to represent a finite graph.
- A path matrix $P=(p_{ij})$ of a simple directed graph (V,E) with n vertices (v1), (v2),….(vn), is a Boolean matrix, i.e. one with entries as 0 or 1 only, where $p_{ij}=1$ if there exists a path and $p_{ij}=0$, if there is no such path.
- The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem.
- An adjacency list is maintained for each node present in the graph which supplies the node value and a pointer to the next adjacent node to the respective node.
- The two common graph traversal methods are Breadth First Search (BFS) and Depth First Search (DFS).

Questions: -

1) What is Graph? Explain matrix and linked list representation of a graph. Also give the application of Graph.
2) Draw the complete undirected graphs on one, two, three, four and five vertices. Prove that the number of edges in an n vertex complete graph is n(n-1)/2.
3) What are the different ways of representing a graph? Represent the following graph using those ways.
4) Write an algorithm which does depth first search through an un-weighted connected graph. In an un-weighted graph, would breadth first search or depth first search or neither find a shortest path tree from some node? Why?
5) Which are the two standard ways of traversing a graph? Explain them with an example of each.
6) Write the steps of Warshall's algorithm.
7) Discuss following with reference to graphs. (i) Directed graph (ii) Undirected graph (iii) Degree of vertex (iv)Null graph (v) Acyclic Graph.
8) Explain various graph traversal schemes and write their merits and demerits.