

Flowcharts

INTRODUCTION

Here are the steps that may be followed to solve an algorithmic problem:

- Analyzing the problem statement means making the objective of the program clear in our minds like what the input is and what is the required output.
- Sometimes the problems are of complex nature, to make them easier to understand, we can break-down the problem into smaller sub-parts.
- In order to save our time in debugging our code, we should first-of-all write down the solution on a paper with basic steps that would help us get a clear intuition of what we are going to do with the problem statement.
- In order to make the solution error-free, the next step is to verify the solution by checking it with a bunch of test cases.
- Now, we clearly know what we are going to do in the code. In this step we will start coding our solution on the compiler.

Basically, in order to structure our solution, we use flowcharts. A flowchart would be a diagrammatic representation of our algorithm - a step-by-step approach to solve our problem.

Flowcharts

Uses of Flowcharts

- Used in documentation.
- Used to communicate one's solution with others, basically used for group projects.
- To check out at any step what we are going to do and get a clear explanation of the flow of statements.

Flowchart components

- **Terminators**
-



Start

Mainly used to denote the start point of the program.



End

Used to denote the end point of the program.

- **Input/Output**



Read var

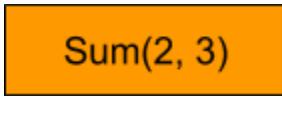
Used for taking input from the user and store it in variable 'var'.



Print var

Used to output value stored in variable 'var'.

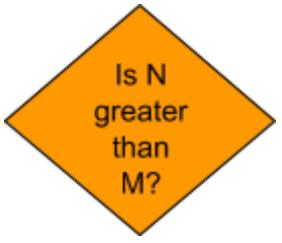
- **Process**



Sum(2, 3)

Used to perform the operation(s) in the program. For example: Sum(2, 3) just performs arithmetic summation of the numbers 2 and 3.

- **Decision**



**Is N
greater
than
M?**

Used to make decision(s) in the program means it depends on some condition and answers in the form of TRUE(for yes) and FALSE(for no).

- **Arrows**



Generally, used to show the flow of the program from one step to another. The head of the arrow shows the next step and the tail shows the previous one.

- **Connector**

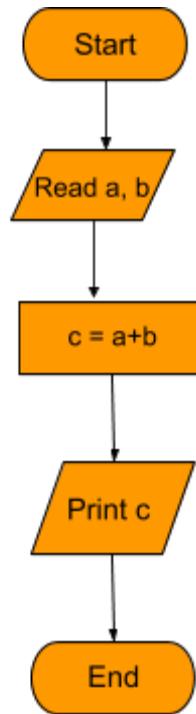


Used to connect different parts of the program and are used in case of break-through. Generally, used for functions(which we will study in our further sections).

Example 1:

Suppose we have to make a flowchart for adding 2 numbers a and b.

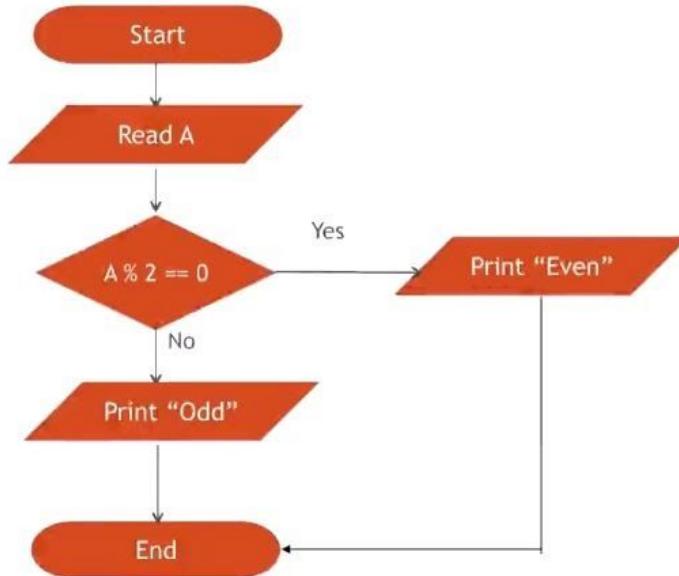
Solution:



Example 2:

Suppose we have to check if a number is even or odd.

Solution:



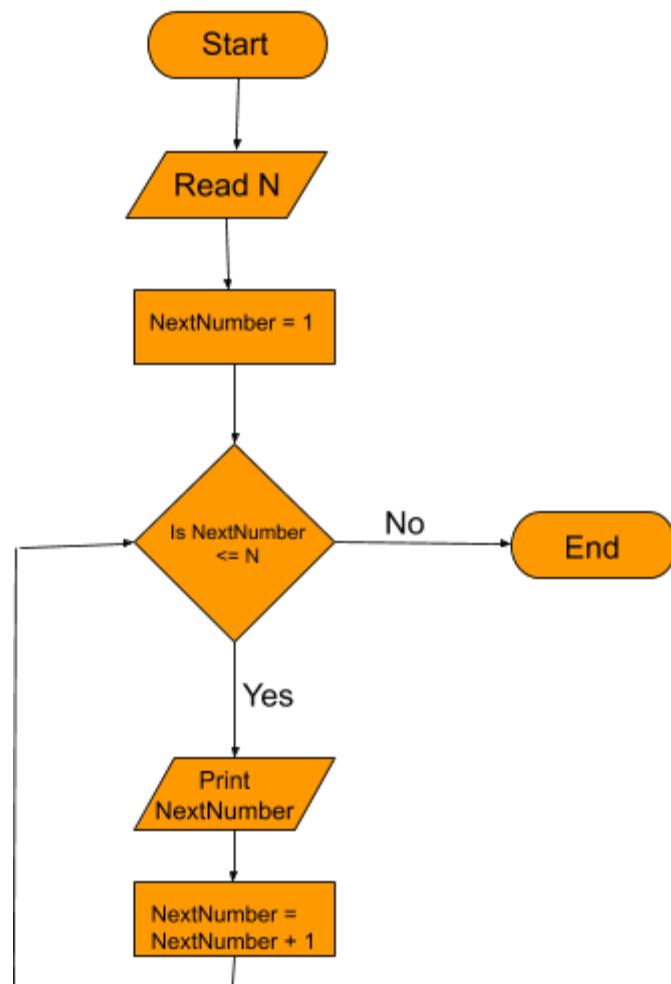
Note: Operator % is used to find the remainder of a number with respect to another number. For example:

- $4 \% 3 = 1$
- $10 \% 5 = 0$
- $4 \% 2 = 0$
- $4 \% 4 = 0$
- $0 \% 1 = 0$
- $1 \% 0 = \text{undefined}$ (as it leads to division by 0)

Example 3:

To print the numbers less than or equal to n where n is given by the user.

Solution:



Summary

- Flowcharts are the building-block of any program written in any language.
- Different shapes used to have different meanings.
- Every problem can be represented in the form of a flow chart.

- Sometimes, it becomes a bulky process to represent any program using flowchart.
In those cases, try to find out the optimal solution to the given problem.

Practice Problems

Till now in examples we learnt how to draw decision making blocks and how to manage looping the same part again and again until the condition is not satisfied.

Using the concepts shown above, here are few tricky flowchart problems for your practice :

- **Check if a given number, N is positive or negative.**
- **Find the average of 3 given numbers.**
- **Given 3 numbers, check whether a valid triangle can be formed using these numbers or not. Print YES or NO.**
- **Given a number N, print N! (N factorial).**
- **Given a number N, print even numbers upto N.**
- **Given a number N, check if it is prime or not. Print YES or NO.**

Java Foundation with Data Structures

Lecture 2 : Getting Started

a) About Eclipse

Eclipse is an integrated development environment (IDE) for developing applications using the Java programming language and many other programming languages. The Java Development Tools (JDT) project provides a plug-in that allows Eclipse to be used as a Java IDE.

A new Java class can be created using the New Java Class wizard. The Java Class wizard can be invoked in different ways –

1. By clicking on the File menu and selecting New → Class, or
2. By right clicking in the package explorer and selecting New → Class, or
3. By clicking on the class drop down button and selecting class.

Note : We will understand what classes are when we will study Object Oriented Programming. For now you can assume them as a file. Also name of class and .java file inside which we have this class should be same.

b) About Main

Consider the following line of code:

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

1. This is the line at which the program will begin executing. This statement is similar to start block in flowcharts. All Java programs begin execution by calling main()
2. We will understand what public, static, void mean in subsequent lectures. For now we should assume that we have to write main as it is.
3. The curly braces {} indicate start and end of main.

c) print / println

In order to print things to console we have to write - System.out.println("Hello World"). Again for now we should leave System.out.print mean, and should write it as it is.

The built-in method `print()` is used to display the string which is passed to it. This output string is not followed by a newline, i.e., the next output will start on the same line. The built-in method `println()` is similar to `print()`, except that `println()` outputs a newline after each call.

Example Code:

```
public static void main(String[] args) {  
    System.out.println("Hello World");  
    System.out.println("Programming is fun");  
}
```

Output:

```
Hello World  
Programming is fun
```

Variables

a) Add two numbers

Consider the following code for adding two numbers

```
public static void main(String[] args) {  
    int num1 = 10;  
    int num2 = 5;  
    int ans = num1 + num2;  
    System.out.println("Sum =" +ans);  
}
```

Output:

```
15
```

Here, we used variables to store values of two integers and their sum. Thus, a variable is a basic unit of storage in a Java program.

Syntax for Declaring a Variable:

```
type variable_name [ = value];
```

Here, `type` is one of Java's primitive datatypes. The `variable_name` is the name of a variable. We can initialize the variable by specifying an equal sign and a value (Initialization is optional). However, the compiler never assigns a default value to an uninitialized local variable in Java.

While writing variable names you should be careful and follow the rules for naming them. Following are the rules for writing variable names -

1. All variable names may contain uppercase and lowercase letters (a-z, A-Z), underscore (_), dollar sign (\$) and the digits 0 to 9. The dollar sign character is not intended for general use. No spaces and no other special characters are allowed.
2. The variable names must not begin with a number.
3. Java is case-sensitive. Uppercase characters are distinct from lowercase characters.
4. A Java keyword (reserved word) cannot be used as a variable name.

b) Data types of variables

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, we can store integers, decimals, or characters in these variables.

There are eight primitive data types in Java:

DATA TYPE	DEFAULT VALUE	DEFAULT SIZE
char	'\0' (null character)	2 bytes
byte	0	1 byte
short	0	2 bytes
int	0	4 bytes
long	0L	8 bytes
Float	0.0f	4 bytes
Double	0.0d	8 bytes
Boolean	false	Not specified

c) Code for calculating Simple Interest

Example Code:

```
public class SimpleInterest {
    public static void main(String[] args) {
        double principal = 2500.0, rate = 6.0, time = 5.0;
        double si = (principal * rate * time) / 100;
        System.out.println("Simple Interest = " + si);
    }
}
```

Output:
Simple Interest = 750.0

Taking Input

a) Scanner

The Java Scanner class breaks the input into tokens using a delimiter that is whitespace by default. It provides many ways to read and parse various primitive values.

In order to use scanner you have to write this import statement at the top –

```
import java.util.Scanner;
```

Example Code:

```
//Code for adding two integers entered by the user
import java.util.Scanner;
class AddTwoNumbers
{
    public static void main(String args[])
    {
        int a, b, c;
        System.out.println("Enter two integers to calculate their sum: ");

        // Create a Scanner
        Scanner s = new Scanner(System.in);
        a = s.nextInt();
        b = s.nextInt();
        c = a + b;
        System.out.println("Sum of entered integers = "+c);
    }
}
```

Sample Input:

10 5

Output:

15

Here, `s.nextInt()` scans and returns the next token as int. A token is part of entered line that is separated from other tokens by space, tab or newline. So when input line is: “10 5” then `s.nextInt()` returns the first token i.e. “10” as int and `s.nextInt()` again returns the next token i.e. “5” as int.

b) Code for calculating simple interest taking input from user

Example Code:

```
import java.util.Scanner;

public class SimpleInterest {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        double si, principal, rate, time;
        principal = input.nextDouble();
        rate = input.nextDouble();
        time = input.nextDouble();
        si = (principal * rate * time) / 100;
        System.out.println("Simple Interest= " + si);
    }
}
```

Sample Input:

2500.0 6.0 5.0

Output:

750.0

c) Taking character input

To read a character as input, we use `next().charAt(0)`. The `next()` function returns the next token in the input as a string and `charAt(0)` function returns the first character in that string.

Example code to read a character as input:

```
import java.util.Scanner;
public class ScannerDemo1 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        char ch = s.next().charAt(0);    // character input
        System.out.println("input character = " +ch);
    }
}
```

Sample Input:

k

Output:

input character = k

Example code to take a string as input:

```
public static void main(String[] args) {  
    Scanner s = new Scanner(System.in);  
    String str;  
    str = s.next();  
    System.out.print(str);  
}
```

Sample Input:

Coding Ninjas

Output:

Coding

Here, s.next() returns the next token as String. A token is part of entered line that is separated from other tokens by space, tab or newline. So when input line is - “Coding Ninjas” then s.next() returns the first token i.e. “Coding”.

d) Other scanner options

Some commonly used Scanner class methods are as follows:

METHOD	DESCRIPTION
public String next()	It returns the next token from the Scanner.
public String nextLine()	It moves the Scanner position to the next line and returns the value as a string.
public byte nextByte()	It scans the next token as a byte.
public short nextShort()	It scans the next token as a short value.
public int nextInt()	It scans the next token as an int value.
public long nextLong()	It scans the next token as a long value.
public float nextFloat()	It scans the next token as a float value.
public double nextDouble()	It scans the next token as a double value.

Example code:

```
public static void main(String[] args) {  
    Scanner s = new Scanner(System.in);  
    int a = s.nextInt();
```

```
        String str = s.nextLine();
        System.out.println(a);
        System.out.println(str);
    }
```

Sample Input:

100 Hello World

Output:

100

Hello World

Here, `s.nextInt()` scans and returns the next token as int. A token is part of entered line that is separated from other tokens by space, tab or newline. So when input line is - “100 Hello World” then `s.nextInt()` returns the first token as int i.e. “100” and `s.nextLine()` returns remaining part of line i.e “(space)Hello World”

How is Data Stored ?

a) How are integers stored ?

The most commonly used integer type is `int` which is a signed 32-bit type. When you store an integer, its corresponding binary value is stored. The way integers are stored differs for negative and positive numbers. For positive numbers the integral value is simple converted into binary value and for negative numbers their 2's compliment form is stored.

Let's discuss How are Negative Numbers Stored?

Computers use 2's complement in representing signed integers because:

1. There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.
2. Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the "addition logic".

Example:

```
int i = -4;
```

Steps to calculate Two's Complement of -4 are as follows:

Step 1: Take Binary Equivalent of the positive value (4 in this case)

0000 0000 0000 0000 0000 0000 0100

Step 2: Write 1's complement of the binary representation by inverting the bits

1111 1111 1111 1111 1111 1111 1111 1011

Step 3: Find 2's complement by adding 1 to the corresponding 1's complement

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011 \\ +0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 \\ \hline 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100 \end{array}$$

Thus, integer -4 is represented by the binary sequence (1111 1111 1111 1111 1111 1111 1111 1100) in Java.

b) Float and Double values

In Java, any value declared with decimal point is by default of type double (which is of 8 bytes). If we want to assign a float value (which is of 4 bytes), then we must use 'f' or 'F' literal to specify that current value is "float".

Example:

```
float float_val = 10.4f;           //float value
double val = 10.4;                //double value
```

c) How are characters stored

Java uses Unicode to represent characters. As we know system only understands binary language and thus everything has to be stored in the form binaries. So for every character there is corresponding code – Unicode/ASCII code and binary equivalent of this code is actually stored in memory when we try to store a char.

Unicode defines a fully international character set that can represent all the characters found in all human languages. In Java, char is a 16-bit type. The range of a char is 0 to 65,536.

Example code:

```
public static void main(String[] args) {
    char ch1, ch2;
    ch1 = 88;    //ASCII value for 'X'
    ch2 = 'Y';
    System.out.println(ch1 +" " +ch2);
}
```

Output:

X Y

Adding int to char

When we add int to char, we are basically adding two numbers i.e. one corresponding to the integer and other is corresponding code for the char.

Example code:

```
public static void main(String[] args) {  
    System.out.println('a' + 1);  
}
```

Output:

98

Here, we added a character and an int, so it added the ASCII value of char 'a' i.e 97 and int 1. So, answer will be 98.

Similar logic applies to adding two chars as well, when two chars are added their codes are actually added i.e. 'a' + 'b' will give 195.

Typecasting

1. Widening or Automatic type conversion:

In Java, automatic type conversion takes place when the two types are compatible and size of destination type is larger than source type.

2. Narrowing or Explicit type conversion:

When we are assigning a larger type value to a variable of smaller type, then we need to perform explicit type casting.

Example code:

```
public static void main(String[] args) {  
    int i = 100;  
    long l1 = i;           //automatic type casting  
  
    double d = 100.04;  
    long l2 = (long)d;    //explicit type casting  
    System.out.println(i);  
    System.out.println(l1);
```

```

        System.out.println(d);
        System.out.println(l2);
    }
}

```

Output:

```

100
100
100.04
100

```

Operators

a) Arithmetic operators

Arithmetic operators are used in mathematical expression in the same way that are used in algebra.

OPERATOR	DESCRIPTION
+	Adds two operands
-	Subtracts second operand from first
*	Multiplies two operands
/	Divides numerator by denominator
%	Calculates Remainder of division

b) Relational operators

Relational Operators are the operators that used to test some kind of relation between two entities. The following table lists the relation operators supported by Java.

OPERATOR	DESCRIPTION
==	Check if two operands are equal
!=	Check if two operands are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check if operand on the left is smaller than right operand
>=	Check if left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

c) Logical operators

Java supports following 3 logical operators. The result of logical operators is a Boolean i.e. true or false.

OPERATOR	DESCRIPTION
&&	Logical AND
	Logical OR
!	Logical NOT

Example:

Suppose $a = \text{true}$ and $b = \text{false}$, then:

$(a \&\& b)$ is false

$(a || b)$ is true

$(!a)$ is false



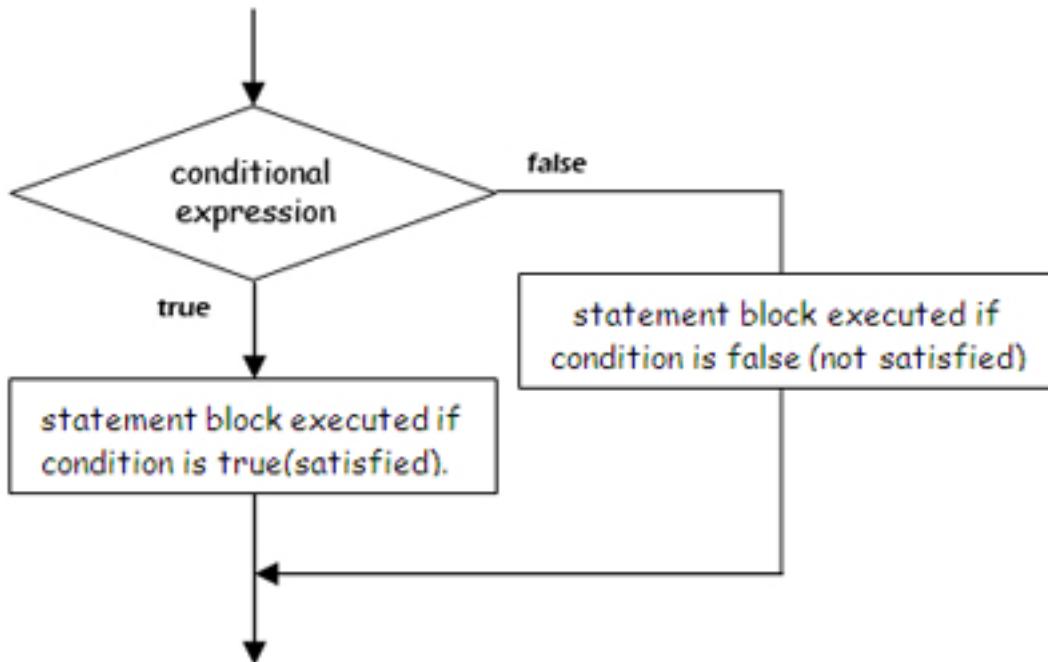
Java Foundation with Data Structures

Lecture 3 : Conditionals and Loops

Conditional Statements (if else)

Description

Conditionals are used to execute a certain section of code only if some specific condition is fulfilled, and optionally execute other statements if the given condition is false. The result of given conditional expression must be either true or false.



Different variations of this conditional statement are –

- **if statement**

if statement evaluates the given test expression. If it is evaluated to true, then statements inside the if block will be executed. Otherwise, statements inside if block is skipped.

Syntax

```
if(test_expression) {  
    // Statements to be executed only when  
    test_expression is true  
}
```

Example Code

```
public static void main(String args[]) {  
    int n = 5;
```

```
if( n < 10 ) {
    System.out.print("Inside if statement");
}
System.out.println("Outside if statement");
}
```

Output

Inside if statement
Outside if statement

So if the condition given inside if parenthesis is true, then statements inside if block are executed first and then rest of the code. And if the condition evaluates to false, then statements inside if block will be skipped.

- **If – else statement**

if statement evaluates the given test expression. If it is evaluated to true, then statements inside the if block will be executed. Otherwise, statements inside else block will be executed. After that, rest of the statements will be executed normally.

Syntax

```
if(test_expression) {
    // Statements to be executed when test_expression
    is true
}

else {
    // Statements to be executed when test_expression
    is false
}
```

Example Code:

```
public static void main(String[] args) {
    int a = 10, b = 20;
    if(a > b) {
        System.out.println("a is bigger");
    }
    else {
        System.out.println("b is bigger");
    }
}
```

```
}
```

Output

b is bigger

- **if – else – if**

Using this we can execute statements based on multiple conditions.

Syntax

```
if(test_expression_1) {  
    // Statements to be executed only when  
    test_expression_1 is true  
}  
else if(test_expression_2) {  
    // Statements to be executed only when  
    test_expression_1 is false and test_expression_2 is true  
}  
else if(test_expression_2) {  
    // Statements to be executed only when  
    test_expression_1 & test_expression_2 are false and  
    test_expression_3 is true  
}  
....  
....  
else {  
    // Statements to be executed only when all the above  
    test expressions are false  
}
```

Out of all block of statements, only one will be executed based on the given test expression, all others will be skipped. As soon as any expression evaluates to true, that block of statement will be executed and rest will be skipped. If none of the expression evaluates to true, then the statements inside else will be executed.

Example Code:

```
public static void main(String[] args) {  
    int a = 5;  
    if(a < 3) {
```

```

        System.out.println("one");
    }
else if(a < 10) {
    System.out.println("two");
}
else if(a < 20) {
    System.out.println("three");
}
else {
    System.out.println("four");
}
}

Output :
two

```

- **Nested if statement**

We can put another if – else statement inside an if.

Syntax

```

if(test_expression_1) {
    // Statements to be executed when
    test_expression_1 is true
    if(test_expression_2) {
        // Statements to be executed when
        test_expression_2 is true
    }
    else {
        // Statements to be executed when
        test_expression_2 is false
    }
}

```

Example Code:

```

public static void main(String[] args) {
    int a = 15;
    if(a > 10) {
        if(a > 20) {
            System.out.println("Hello");
        }
        else {

```

```
        System.out.println("Hi");
    }
}
```

Output :

Hi

return keyword

return is a special keyword, when encountered ends the main. That means, no statement will be executed after return statement. We'll study in more detail when we'll study functions.

Example Code:

```
public static void main(String[] args) {
    int a = 10;
    if(a > 5) {
        System.out.println("Hello");
        return;
    }
    System.out.println("Hi");
}
```

Output :

Hello

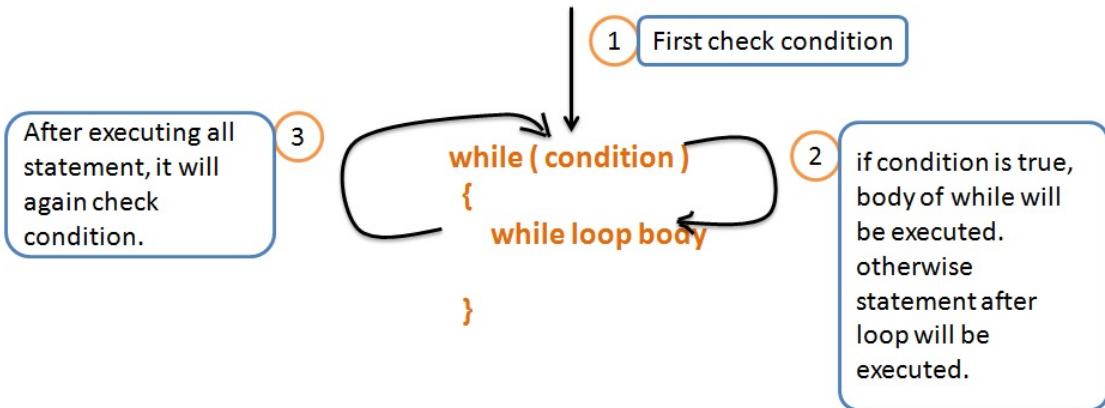
while loop

Loop statements allows us to execute a block of statamenets several number of times depending on certain condition. **while** is one kind of loop that we can use.

When executing, if the *test_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Syntax

```
while(test_expression) {
    // Statements to be executed till test_expression is true
}
```



Example Code:

```
public static void main(String[] args) {
    int i = 1;
    while(i <= 5) {
        System.out.println(i);
        i++;
    }
}
```

Output :

1
2
3
4
5

In while loop, first given test expression will be checked. If that evaluates to be true, then the statements inside while will be executed. After that, the condition will be checked again and the process continues till the given condition becomes false.

Patterns

Introduction

Patterns are a handy application of loops and will provide you with better clarity and understanding of the implementation of loops.

Before printing any pattern, you must consider the following three things:

- The first step in printing any pattern is to figure out the number of rows that the pattern requires.
- Next, you should know how many columns are there in the i^{th} row.
- Once, you have figured out the number of rows and columns, then focus on the pattern to print.

For eg. We want to print the following pattern for N rows: (**Pattern 1.1**)

```
// For N=4:  
****  
****  
****  
****
```

Approach:

From the above pattern, we can observe:

- **Number of Rows:** The pattern has 4 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 4 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- **What to print:** We have to print * 4 times in all the 4 rows. Thus, in a pattern of N rows, we will have to print * N times in all the rows.

Now, let us discuss how to implement such patterns using Java.

Java Implementation for Patterns

We generally need two loops to print patterns. The outer loop iterates over the rows, while the inner nested loop is responsible for traversing the columns. The **algorithm** to print any pattern can be described as follows:

- Accept the number of rows or size of the pattern from a user using the `.nextInt()` function.
- Iterate the rows using the outer loop.
- Use the nested inner loop to handle the column contents. The internal loop iteration depends on the values of the outer loop.
- Print the required pattern contents using the `print` function.
- Add a new line after each row.

The implementation of **Pattern 1.1** in Java will be:

Step 1: Let us first use a loop to traverse the rows. This loop will start at the first row and go on till the N^{th} row. Below is the implementation of this loop:

```
public static void main(String[] args) {  
    Scanner s = new Scanner(System.in);  
    int N = s.nextInt(); // Take user input, N= Number of Rows  
    int row = 1; // The Loop starts with the 1st row  
    while (row <= N) { // Loop will run for N row  
        // <Here goes the Nested Loop>  
        row = row+1; // Increment the current row (Outer Loop)  
        System.out.println(); // Add a new Line after each row  
    }  
}
```

Printing a New Line: Since we need to print the pattern in multiple lines, we will have to add a new line after each row. Thus for this purpose, we use an empty print statement. The print function in Java, can be written as
System.out.println(). 'ln' after print indicates a new line.

Step 2: Now, we need another loop to traverse the row during each iteration and print the pattern; this can be done as follows:

```
public static void main(String[] args) {  
    Scanner s = new Scanner(System.in);  
    int N = s.nextInt(); // Take user input, N= Number of Rows  
    int row = 1; // The loop starts with the 1st row  
    while (row <= N) { // Loop will run for N rows  
        int col = 1; // Loop starts with the first column in the  
                    //current row  
        while (col <= N) { //Loop will run for N columns  
            System.out.print("*") // printing (*) in each column  
            col = col+1 //Increment the current column (Inner Loop)  
        }  
        row = row+1; // Increment the current row (Outer Loop)  
        System.out.println(); // Add a new Line after each row  
    }  
}
```

There are two popular types of patterns-related questions that are usually posed:

- Square Pattern - **Pattern 1.1** is square.
- Triangular Pattern

Let us now look at the implementation of some common patterns.

Square Patterns

Pattern 1.2

```
// N = 5
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- **What to print:** All the entries in any row are the same as the corresponding row numbers. Thus in a pattern of N rows, all the entries of the i^{th} row are i (1^{st} row has all 1's, 2^{nd} row has all 2's, and so on).

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will run for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= N) { //Loop will run for N columns
            System.out.print("*"); // printing (*) in each column
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}
    
```

Pattern 1.3

```

// N = 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
    
```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.

→ **What to print:** All the entries in any row are the same as the corresponding column numbers. Thus in a pattern of N rows, all the entries of the i^{th} column are i (1^{st} column has all 1's, 2^{nd} column has all 2's, and so on).

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= N) { //Loop will on for N columns
            System.out.print(col); // printing (*) in each column
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}
  
```

Pattern 1.4

```

// N = 5
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
  
```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- **What to print:** All the entries in any row are $N - \text{columnNumber} + 1$. Thus in a pattern of N rows, all the entries of the i^{th} column are $N - i + 1$ (1^{st} column has all 5's ($5 - 1 + 1$), 2^{nd} column has all 4's ($5 - 2 + 1$), and so on).

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= N) { //Loop will on for N columns
            System.out.print(N-col+1); // printing (*) in each column
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}

```

This way there can be several other square patterns and you can easily print them using this approach- **By finding the number of Rows, Columns and What to print.**

Pattern 1.5

```
// N = 5
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- **What to print:** The first entry in the 1st row is 1, the first entry in the 2nd row is 2, and so on. Further, these values are incremented continuously by 1 in the remaining entries of any particular row. Thus in a pattern of N rows, the first entry of the ith row is i. The remaining entries in the ith row are i+1, i+2, and so on. It can be observed that any entry in this pattern can be written as row+col-1.

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= N) { //Loop will on for N columns
            System.out.print(col+row-1); // printing in each column
            col = col+1; //Increment the current column (Inner Loop)
        }
    }
}
```

```

        }

    row = row+1; // Increment the current row (Outer Loop)
    System.out.println(); // Add a new Line after each row
}

}

```

Triangular Patterns

Pattern 1.6

```

// N = 5
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5

```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- **Number of Columns:** The number of columns in any row is the same as the corresponding row number. 1st row has 1 column, 2nd row has 2 columns, and so on. Thus, in a pattern of N rows, the ith row will have i columns.
- **What to print:** All the entries in any row are the same as the corresponding row numbers. Thus in a pattern of N rows, all the entries of the ith row are i (1st row has all 1's, 2nd row has all 2's, and so on).

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will run for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= row) { //Loop will run for row times
            System.out.print(row); // printing in each column
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}
  
```

Pattern 1.7

```

// N = 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
  
```

Approach:

From the above pattern **we can observe:**

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.

- **Number of Columns:** The number of columns in any row is the same as the corresponding row number. 1st row has 1 column, 2nd row has 2 columns, and so on. Thus, in a pattern of N rows, the ith row will have i columns.
- **What to print:** All the entries in any row are the same as the corresponding column numbers. Thus in a pattern of N rows, all the entries of the ith column are i (1st column has all 1's, 2nd column has all 2's, and so on).

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= row) { //Loop will on for row times
            System.out.print(col); // printing in each column
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}
  
```

Pattern 1.8

```

// N = 5
1
2 3
4 5 6
7 8 9 10
  
```

[11](#) [12](#) [13](#) [14](#) [15](#)

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- **Number of Columns:** The number of columns in any row is the same as the corresponding row number. 1st row has 1 column, 2nd row has 2 columns, and so on. Thus, in a pattern of N rows, the ith row will have i columns.
- **What to print:** The pattern starts with 1 and then each column entry is incremented by 1. Thus, we will initialize a variable temp=1. We will keep printing the value of temp in the successive columns and upon printing, we will increment the value of temp by 1.

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    int temp = 1;
    while (row <= N) { // Loop will on for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= row) { //Loop will on for row times
            System.out.print(temp); // printing in each column
            temp = temp+1;
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}
  
```

```

    }
}
```

Character Patterns

Pattern 1.9

```
// N = 4
ABCD
ABCD
ABCD
ABCD
```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 4 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 4 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- **What to print:** The 1st column has all A's, 2nd column has all B's, and so on. The **ASCII** value of A is **65**. In the 1st column, the character corresponds to the **ASCII** value 65 (**64+1**). In the 2nd column, the character corresponds to the **ASCII** value 66 (**64+2**). Thus, all the entries in the ith column are equal to the character corresponding to the **ASCII** value **64+i**. The `char()` function gives the character associated with the integral ASCII value within the parentheses.

Java Implementation:

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows
```

```

int col = 1; // Loop starts with the first column in the
             //current row
while (col <= N) { //Loop will run for N columns
    System.out.print((char)(64+col)); //print in each column
    col = col+1; //Increment the current column (Inner Loop)
}
row = row+1; // Increment the current row (Outer Loop)
System.out.println(); // Add a new Line after each row
}
}

```

Pattern 1.10

```

// N = 4
ABCD
BCDE
CDEF
DEFG

```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 4 rows. We have to print the pattern for N rows.
- **Number of Columns:** All the rows have 4 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- **What to print:** This pattern is very similar to **Pattern 1.5**. We can implement this using a similar code with a minor change. Instead of integers, we need capital letters of the same order. Instead of 1, we need A, instead of 2, we need B and so on. **ASCII** value of A is 65. Thus if we add 64 to all the entries in **Pattern 1.5** and find their **ASCII** values, we will get our result. The `char()` function gives the character associated with the integral **ASCII** value within the parentheses.

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= N) { //Loop will on for N columns
            System.out.print((char)(64+col+row-1)); //print character
            col = col+1; //Increment the current column (Inner Loop)
        }
        row = row+1; // Increment the current row (Outer Loop)
        System.out.println(); // Add a new Line after each row
    }
}

```

Practice Problems

Here are a few similar patterns problems for your practice. All the patterns have been drawn for N=4.

A
AB
ABC
ABCD

12344321
123***321
12****21

1*****1

ABCD
ABC
AB
A

4555
3455
2345
1234

1
11
202
3003

A
BB
CCC
DDDD

Patterns

Some Advanced Patterns

Pattern 2.1 - Inverted Triangle

```
// N = 3
* * *
* *
*
```

Approach:

From the above pattern, **we can observe**:

- **Number of Rows:** The pattern has 3 rows. We have to print the pattern for N rows.
- **Number of Columns:** The number of columns in any row is equal to $N - \text{rowNumber} + 1$. 1st row has 3 columns ($3 - 1 + 1$), 2nd row has 2 columns ($3 - 2 + 1$), and so on. Thus, in a pattern of N rows, the i^{th} row will have $N - i + 1$ columns.
- **What to print:** All the entries in any row are `"*"`.

Java Implementation:

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will run for N rows
        int col = 1; // Loop starts with the first column in the
                      //current row
        while (col <= N - row + 1) { //Number of columns = N - rowNumber + 1
            System.out.print("*"); // printing in each column
        }
        row++;
    }
}
```

```

        col = col+1; //Increment the current column (Inner Loop)
    }

    row = row+1; // Increment the current row (Outer Loop)
    System.out.println(); // Add a new Line after each row
}

}

```

Pattern 2.2 - Reversed Pattern

```

// N = 3
*
* *
* * *

```

Approach:

From the above pattern, **we can observe**:

- **Number of Rows:** The pattern has 3 rows. We have to print the pattern for N rows.
- **Number of Columns:** The number of columns in any row is equal to N.
- **What to print:** In the 1st row, while `columnNumber <= 2(3-1)`, we print a " " in every column. Beyond the 2nd column, we print a "*". Similarly, in the 2nd row, we print a " " till `columnNumber <=1(3-2)` and beyond the 1st column, we print a "*". We can easily notice that if `col <= N-rowNumber`, we are printing a " " (**Space**). And if `col > N-rowNumber`, we are printing a "*".

Java Implementation:

```

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int N = s.nextInt(); // Take user input, N= Number of Rows
    int row = 1; // The Loop starts with the 1st row
    while (row <= N) { // Loop will on for N rows

```

```

int col = 1; // Loop starts with the first column in the
             //current row
while (col <= N) { //Loop will run for N rows
    if(col<=N-row)
        System.out.print(" "); // printing " "
    else
        System.out.print("*"); // printing "*"
    col = col+1; //Increment the current column
}
row = row+1; // Increment the current row (Outer Loop)
System.out.println(); // Add a new Line after each row
}
}
  
```

Pattern 2.3 - Isosceles Pattern

```

// N = 4
  1
 121
12321
1234321
  
```

Approach:

From the above pattern **we can observe**:

- **Number of Rows:** The pattern has 3 rows. We have to print the pattern for N rows.
- **Number of Columns:** Similar to Pattern 2.2, we first have N-rowNumber columns of spaces. Following this, we have 2*rowNumber-1 columns of numbers.
- **What to print:** We can notice that if col <= N-rowNumber, we are printing a " " (Space). Further, the pattern has two parts. First is the increasing part and second is the decreasing part. For the increasing part, we will initialise a

variable `num=1`. In each row we will keep printing `num` till its value becomes equal to the `rowNumber`. We will increment `num` by 1 after printing it; ;this will account for the first part of the pattern. We have `num = rowNumber` at this stage. The decreasing part starts with `rowNumber - 1`. Hence, we will initialise `num` with `rowNumber - 1`. Now, for the decreasing part, we will again start printing `num` till `num>=1`. After printing `num` we will decrement it by 1.

Java Implementation:

```
public static void main(String[] args) {  
    Scanner s = new Scanner(System.in);  
    int N = s.nextInt(); // Take user input, N= Number of Rows  
    int row = 1; // The loop starts with the 1st row  
    while (row <= N) { // Loop will run for N rows  
        int spaces = 1; // Printing spaces  
        while (spaces <= N-row) {  
            System.out.print(" ");  
            spaces=spaces+1;  
        }  
        int num=1; // Variable to print the numbers  
        while (num <= row) { // Increasing Pattern  
            System.out.print(num);  
            num=num+1;  
        }  
  
        num=row-1; // We have to start printing the decreasing part  
        // from one less than the rowNumber  
        while (num >= 1) { // Decreasing Pattern  
            System.out.print(num);  
        }  
    }  
}
```

```

        num=num-1;
    }

    row = row+1; // Increment the current row (Outer Loop)
    System.out.println(); // Add a new Line after each row
}

}

```

Practice Problems

Here are a few similar patterns problems for your practice. All the patterns have been drawn for N=4.

```

*
 ***
 *****
 ******

```

```

1
121
12321
1234321
12321
121
1

```

```

1   1
2   2
3  3
4
3  3
2   2

```

1 1

```
*
```

```
***
```

```
*****
```

```
*****
```

```
***
```

```
*
```



Java Foundation with Data Structures

Lecture 4 : Loops, Keywords, Associativity and Precedence

for loop

Loop statements allows us to execute a block of statements several number of times depending on certain condition. **for** loop is kind of loop in which we give initialization statement, test expression and update statement can be written in one line.

Inside for, three statements are written –

- Initialization – used to initialize your loop control variables. This statement is executed first and only once.
- Test condition – this condition is checked everytime we enter the loop. Statements inside the loop are executed till this condition evaluates to true. As soon as condition evaluates to false, loop terminates and then first statement after for loop will be executed next.
- Updation – this statement updates the loop control variable after every execution of statements inside loop. After updation, again test conditon is checked. If that comes true, the loop executes and process repeats. And if condition is false, the loop terminates.

```
for (initializationStatement; test_expression; updateStatement) {  
    // Statements to be executed till test_expression is true  
}
```

Example Code :

```
public static void main(String[] args) {  
    for(int i = 0; i < 3; i++) {  
        System.out.print("Inside for loop : ");  
        System.out.println(i);  
    }  
    System.out.println("Done");  
}
```

Output:

```
Inside for Loop : 0  
Inside for Loop : 1  
Inside for Loop : 2  
Done
```

In for loop its not compulsory to write all three statements i.e. initializationStatement, test_expression and updateStatement. We can skip one or more of them (even all three)

Above code can be written as:

```
public static void main(String[] args) {  
    int i = 1;      //initialization is done outside the for loop  
    for(; i <=5; i++) {  
        System.out.println(i);  
    }  
}
```

OR

```
public static void main(String[] args) {  
    int i = 1;      //initialization is done outside the for loop  
    for(; i <=5; ) {  
        System.out.println(i);  
        i++;      // update Statement written here  
    }  
}
```

We can also skip the test_expression. See the example below :

Variations of for loop

- The three expressions inside for loop are optional. That means, they can be omitted as per requirement.

Example code 1: Initialization part removed –

```
public static void main(String[] args) {  
    int i = 0;  
    for( ; i < 3; i++) {  
        System.out.println(i);  
    }  
}
```

Output:

0
1
2

Example code 2: Updation part removed

```
public static void main(String[] args) {  
    for(int i = 0; i < 3; ) {  
        System.out.println(i);  
        i++;  
    }  
}
```

Output:

0
1
2

Example code 3: Condition expression removed , thus making our loop infinite –

```
public static void main(String[] args) {  
    for(int i = 0; ; i++) {  
        System.out.println(i);  
    }  
}
```

Example code 4:

We can remove all the three expression, thus forming an infinite loop-

```
public static void main(String[] args) {  
    for( ; ; ) {  
        System.out.print("Inside for loop");  
    }  
}
```

- **Multiple statements inside for loop**

We can initialize multiple variables, have multiple conditions and multiple update statements inside a *for loop*. We can separate multiple statements using comma, but not for conditions. They need to be combined using logical operators.

Example code:

```
public static void main(String[] args) {  
    for(int i = 0, j = 4; i < 5 && j >= 0; i++, j--) {  
        System.out.println(i + " " + j);  
    }  
}
```

Output:

```
0 4  
1 3  
2 2  
3 1  
4 0
```

break and continue

1. **break statement:** The break statement terminates the loop (for, while and do..while loop) immediately when it is encountered. As soon as break is encountered inside a loop, the loop terminates immediately. Hence the statement after loop will be executed next.
2. **continue statement:** The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else. (caution always update the counter in case of while loop else loop will never end)

```
while(test_expression) {  
    // codes  
    if (condition for break) {  
        break;  
    }  
    //codes
```

```
}
```



```
for (initializationStatement; test_expression; updateStatement) {  
    // codes  
    if (condition for break) {  
        break;  
    }  
    //codes  
}
```

❖ break

- Example: (using break inside for loop)

```
public static void main(String[] args) {  
    for(int i = 1; i < 10; i++) {  
        System.out.println(i);  
        if(i == 5) {  
            break;  
        }  
    }  
}
```

Output:

```
1  
2  
3  
4  
5
```

- Example: (using break inside while loop)

```
public static void main(String[] args) {  
    int i = 1;  
    while (i <= 10) {  
        System.out.println(i);  
        if(i==5)  
        {  
            break;  
        }  
        i++;  
    }  
}
```

```
    }  
}
```

Output:

```
1  
2  
3  
4  
5
```

- Inner loop break:

When there are two more loops inside one another. Break from innermost loop will just exit that loop.

Example Code 1:

```
public static void main(String[] args) {  
    for (int i=1; i <=3; i++) {  
        System.out.println(i);  
        for (int j=1; j<= 5; j++)  
        {  
            System.out.println("in");  
            if(j==1)  
            {  
                break;  
            }  
        }  
    }  
}
```

Output:

```
1  
in...  
2  
in...  
3  
in...
```

Example Code 2:

```
public static void main(String[] args) {  
    int i=1;  
    while (i <=3) {  
        System.out.println(i);  
        int j=1;  
        while (j <= 5)  
        {  
            System.out.println("in");  
            if(j==1)  
            {  
                break;  
            }  
            j++;  
        }  
        i++;  
    }  
}
```

Output:

```
1  
in...  
2  
in...  
3  
in...
```

❖ Continue

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- Example: (using for loop)

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

```
for (int i=1; i <= 5; i++) {  
    if(i==3)  
    {  
        continue;  
    }  
    System.out.println(i);  
}
```

Output:

```
1  
2  
4  
5
```

- Example: (using while loop)

```
public static void main(String[] args){  
    int i=1;  
    while (i <= 5) {  
        if(i==3)  
        {  
            i++;  
            // if increment isn't done here then loop will run  
            // infinite time for i=3  
            continue;  
        }  
        System.out.println(i);  
        i++;  
    }  
}
```

Output:

```
1  
2  
4  
5
```

Scope of variables

Scope of variables is the curly brackets {} inside which they are defined. Outside which they aren't known to the compiler. Same is for all loops and conditional statement (if).

❖ Scope of variable - for loop

```
for (initializationStatement; test_expression; updateStatement) {  
    // Scope of variable defined in loop  
}
```

Example:

```
public static void main(String[] args) {  
    for (int i=0; i<5; i++) {  
        int j=2;      // Scope of i and j are both inside the loop they  
        can't be used outside  
    }
```

❖ Scope of variable for while loop

```
while(test_expression) {  
    // Scope of variable defined in loop  
}
```

```
public static void main(String[] args) {  
    int i=0;  
    while(i<5)  
    {  
        int j=2; // Scope of i is main and scope of j is only the loop  
        i++;  
    }  
}
```

❖ Scope of variable for conditional statements

```

if(test_expression) {
    // Scope of variable defined in the conditional statement
}

public static void main(String[] args) {
    int i=0;
    if (i<5)
    {
        int j=5;      // Scope of j is only in this block
    }
    // cout<<j; → This statement if written will give an error because
    // scope of j is inside if and is not accessible outside if.
}

```

Increment Decrement operator

Explanation

Pre-increment and *pre-decrement* operators' increments or decrements the value of the object and returns a reference to the result.

Post-increment and *post-decrement* creates a copy of the object, increments or decrements the value of the object and returns the copy from before the increment or decrement.

Post-increment(a++):

This increases value by 1, but uses old value of a in any statement.

Pre-increment(++a):

This increases value by 1, and uses increased value of a in any statement.

Post-decrement(a--):

This decreases value by 1, but uses old value of a in any statement.

Pre-decrement(-- a):

This decreases value by 1, and uses decreased value of a in any statement.

```

public static void main(String[] args) {

```

```

int I=1, J=1, K=1, L=1;

cout<<I++<<' '<<J-- <<' '<<++K<<' '<< --L<<endl;

cout<<I<<' '<<J<<' '<<K<<' '<<L<<endl;

}

```

Output:

```

1 1 2 0
2 0 2 0

```

Bitwise Operators

Bitwise operators are used to perform operations at bit level. Following is the summary of various bitwise operations:

Operator	Name	Example	Result	Description
$a \& b$	and	4 & 6	4	1 if both bits are 1.
$a b$	or	4 6	6	1 if either bit is 1.
$a ^ b$	xor	4 ^ 6	2	1 if both bits are different.
$\sim a$	not	~ 4	-5	Inverts the bits. (Unary bitwise compliment)
$n << p$	left shift	$3 << 2$	12	Shifts the bits of n left p positions. Zero bits are shifted into the low-order positions.
$n >> p$	right shift	$5 >> 2$	1	Shifts the bits of n right p positions. If n is a 2's complement signed number, the sign bit is shifted into the high-order positions.
$n >>> p$	right shift	$-4 >>> 28$	15	Shifts the bits of n right p positions. Zeros are shifted into the high-order positions.

Example Code:

```

public static void main(String args[]) {
    int a = 19; // 19 = 10011
    int b = 28; // 28 = 11100
    int c = 0;

```

```

c = a & b;      // 16 = 10000
System.out.println("a & b = " + c );

c = a | b;      // 31 = 11111
System.out.println("a | b = " + c );

c = a ^ b;      // 15 = 01111
System.out.println("a ^ b = " + c );

c = ~a;         // -20 = 01100
System.out.println("~a = " + c );

c = a << 2;    // 76 = 1001100
System.out.println("a << 2 = " + c );

c = a >> 2;   // 4 = 00100
System.out.println("a >> 2 = " + c );
}

```

Output

a & b = 16
 a | b = 31
 a ^ b = 15
 ~a = -20
 a << 2 = 76
 a >> 2 = 4
 a >>> 2 = 4

Precedence and Associativity

- **Operator precedence** determines which operator is performed first in an expression with more than one operators with different precedence.
 For example, $10 + 20 * 30$ is calculated as $10 + (20 * 30)$ and not as $(10 + 20) * 30$.

- **Associativity** is used when two operators of same precedence appear in an expression. Associativity can be either **Left to Right** or **Right to Left**. For example, '*' and '/' have same precedence and their associativity is **Left to Right**, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Precedence and Associativity are two characteristics of operators that determine the evaluation order of subexpressions in absence of brackets.

Note : We should generally use add proper brackets in expressions to avoid confusion and bring clarity.

1) Associativity is only used when there are two or more operators of same precedence.

The point to note is associativity doesn't define the order in which operands of a single operator are evaluated. For example, consider the following program, associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2(). The output of following program is in-fact compiler dependent.

```
// Associativity is not used in the below program. Output is compiler dependent.  
static int x = 0;  
public static int F1() {  
    x = 5;  
    return x;  
}  
public static int F2() {  
    x = 10;  
    return x;  
}  
public static void main(String[] args) {  
    int p = F1() + F2();  
    System.out.println(x);  
}
```

2) All operators with same precedence have same associativity

This is necessary, otherwise there won't be any way for compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity. For example, + and – have same associativity.

3) There is no chaining of comparison operators in Java

Trying to execute the statement a>b>c will give an error and the code will not compile

Following is the Precedence table along with associativity for different operators.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
() [] . ++ —	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Postfix increment/decrement (see Note 2)	left-to-right
++ — + — ! ~ (<i>type</i>)	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of <i>type</i>)	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ — << >>	Addition/subtraction Bitwise shift left, Bitwise shift right	left-to-right
< <= > >= == !=	Relational less than/less than or equal to Relational greater than/greater than or equal to Relational is equal to/is not equal to	left-to-right
& ^	Bitwise AND Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left

=	Assignment	
$+=$ $-=$	Addition/subtraction assignment	
$*=$ $/=$	Multiplication/division assignment	
$%=$ $\&=$	Modulus/bitwise AND assignment	
$\^=$ $ =$	Bitwise exclusive/inclusive OR assignment	
$<<=$ $>>=$	Bitwise shift left/right assignment	right-to-left

Coding Ninjas

Java Foundation with Data Structures

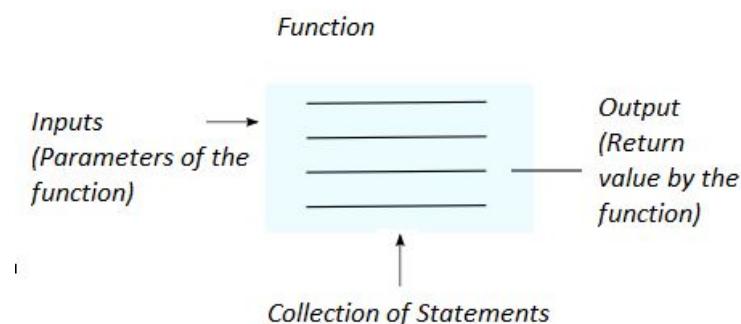
Lecture 5 : Functions, Variables and Scope

sFunctions

A Function is a collection of statements designed to perform a specific task. Function is like a black box that can take certain input(s) as its parameters and can output a value which is the return value. A function is created so that one can use it as many time as needed just by using the name of the function, you do not need to type the statements in the function every time required.

Defining Function

```
return_type function_name(parameter 1, parameter 2, .....){  
    statements;  
}
```



- **return type:** A function may return a value. The *return type* of the function is the data type of the value that function returns. Sometimes function is not required to return any value and still performs the desired task. The return type of such functions is **void**.

Example:

Following is the example of a function that sum of two numbers. Here input to the function are the numbers and output is their sum.

```
1. public static int findSum( int a, int b){  
2.     int sum = a + b;  
3.     return sum;  
4. }
```

Function Calling

Now that we have read about how to create a function lets see how to call the function. To call the function you need to know the name of the function and number of parameters required and their data types and to collect the returned value by the function you need to know the return type of the function.

Example

```
1. public static int findSum( int a, int b){  
2.     int sum = a + b;  
3.     return sum;  
4. }  
5. public static void main () {  
6.     int a = 10, b = 20;  
7.     int c=findSum (a, b); //function findSum () is called using its name and  
//by knowing  
8.     System.out.print(c);// the number of parameters and their data type.  
9. } // integer c is used to collect the returned value by  
//the function
```

Output:

30

IMPORTANT POINTS:

- **Number of parameter** and their **data type** while calling must match with function signature. Consider the above example, while calling function **findSum ()** the number of parameters are two and both the parameter are of integer type.
- It is okay not to collect the return value of function. For example, in the above code to find the sum of two numbers it is right to print the return value directly.

“System.out.print(c);”

- **void return type functions:** These are the functions that do not return any value to calling function. These functions are created and used to perform specific task just like the normal function except they do not return a value after function executes.

Following are some more examples of functions and their use to give you a better idea.

Function to find area of circle

```
1. public static double findArea(double radius){  
2.         double area = radius*radius*3.14; //return type is double  
3.         return area;  
4.     }  
5.  
6. public static void main(String[] args) {  
7.         double radius = 5.8;  
8.         double c = findArea(radius);  
9.         System.out.print(c);  
10.    }  
11.
```

Function to print average

```
1. public static void printAverage(int a, int b ){ //return type of the  
function is void  
2.     int avg = (a + b) / 2;  
3.     System.out.print(avg);  
4. } // This function does not return any value  
5.  
6. public static void main () {  
7.     int a = 15, b = 25;  
8.     printAverage (a, b);  
9. }
```

Why do we need function?

- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many time as it is needed, which saves work. Consider that you are required to find out the area of the circle, now either you can apply the formula every time to get the area of circle or you can make a function for finding area of the circle and invoke the function whenever it is needed.
- **Neat code:** A code created with function is easy to read and dry run. You don't need to type the same statements over and over again, instead you can invoke the function whenever needed.
- **Modularisation –** Functions help in modularising code. Modularisation means to divides the code in small modules each performing specific task. Functions helps in doing so as they are the small fragments of the programme designed to perform the specified task.
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.

How does function calling works?

Consider the following code where there is a function called **findsum** which calculates and returns sum of two numbers.

//Find Sum of two integer numbers

```
1. public static int findSum( int a, int b){  
2.     int sum = a + b;  
3.     return sum;  
4. }  
5. public static void main () {  
6.     int a = 10, b = 20;  
7.     int c=findSum (a, b);  
8.     System.out.print(c);  
9. }
```

```
#include <iostream>
```

The function being called is called **callee**(here it is **findsum** function) and the function which calls the callee is called

```
void function_name() { ←
```

```
    ... ... ...
```

```
    ... ... ...
```

```
int main() {
```

```
    ... ... ...
```

```
    function_name(); ←
```

```
    ... ... ...
```

```
}
```

the **caller** (here main function is the caller) .

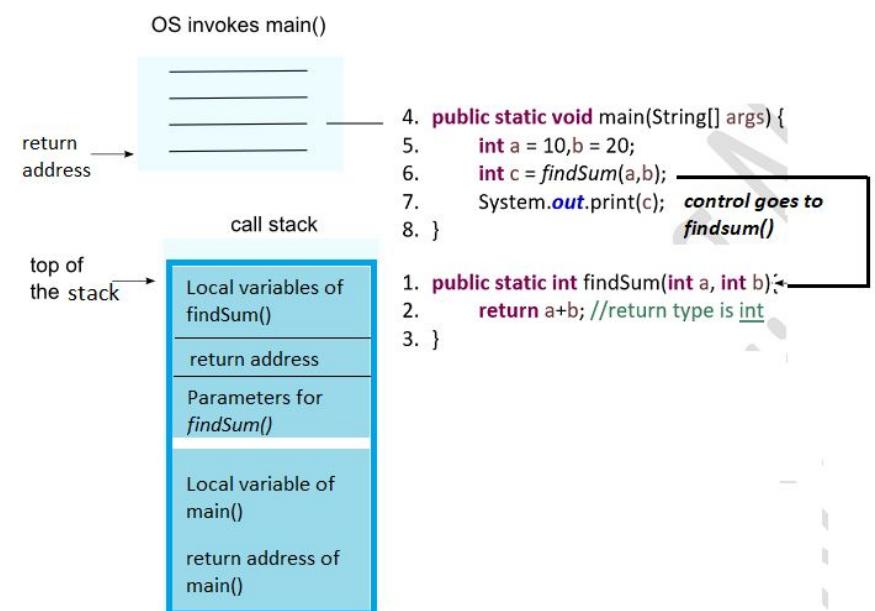
When a function is called, programme control goes to the entry point of the function. Entry point is where the function is defined. So focus now shifts to callee and the caller function goes in paused state .

For Example: In above code entry point of the function **findSum ()** is at line number 3. So when at line number 9 the function call occurs the control goes to line number 3, then after the statements in the function **findSum ()** are executed the programme control comes back to line number 9.

Role of stack in function calling (call stack)

A call stack is a storage area that store information about the *active* function and paused functions. It stores parameters of the function, return address of the function and variables of the function that are created statically.

Once the function statements are terminated or the function has returned a value, the call stack removes all the information about that function from the stack.



Benefits of functions

- **Modularisation**
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.
- **Neat code:** A code created with function is easy to read and dry run.

Variables and Scopes

Local Variables

Local variable is a variable that is given a local scope. Local variable belonging to a function or a block has its scope only within the function or block inside which it is declared. Scope of a variable is part of a programme for which this variable is accessible.

Example:

```
1. #include<iostream>
2. using namespace std;
3. public static void main(){}
4.     int a = 10;
5.     System.out.print(a);
6.
7. }
```

Output

5

In the above code the variable **a** declared inside the block after if statement is a local variable for this block.

Lifetime of a Variable

The lifetime of a variable is the time period for which the declared variable has a valid memory. Scope and lifetime of a variable are two different concepts, scope of a variable is part of a programme for which this variable is accessible whereas lifetime is duration for which this variable has a valid memory.

Loop variable

Loop variable is a variable which defines the loop index for each iteration.

Example

```
"for (int i = 0; i < 3; i++) { // variable i is the loop variable
    ....;
    ....;
    statements;
}
```

For this example, variable **i** is the loop variable.

Variables in the same scope

Scope is part of programme where the declared variable is accessible. In the same scope, no two variables can have name. However, it is possible for two variables to have same name if they are declared in different scope.

Example:

```
1. public static void main(String[] args) {  
2.     int a = 10;  
3.     double a = 5; // two variables with same name, the code will not  
    compile  
4.     System.out.println(a);  
5. }
```

For the above code, there are two variables with same name **a** in the same scope of main () function. Hence the above code will not compile.

Pass by value:

When the parameters are passed to a function by pass by value method, then the formal parameters are allocated to a new memory. These parameters have same value as that of actual parameters. Since the formal parameters are allocated to new memory any changes in these parameters will not reflect to actual parameters.

Example:

```
//Function to increase the parameters value  
1. public static void increase(int x, int y){  
2.     x++;  
3.     y = y + 2;  
4.     System.out.println(x + ":" + y); //x and y are formal  
    parameters  
5. }  
6. public static void main(String[] args) {  
7.     int a = 10;  
8.     int b = 20;  
9.     increase(a,b);  
10.    System.out.println(a + ":" + b); //a and b are actual  
    parameters  
11.  
12.}
```

Output:

11: 22

10: 20

For the above code, changes in the values of **x** and **y** are not reflected to **a** and **b** because x and y are formal parameters and are local to function increment so any changes in their values here won't affect variables a and b inside main.

Arrays

What you will learn in this lecture?

- What are arrays
- How are they stored?
- Array Indices

Introduction

In cases where there is a need to use several variables of the same type, for storing, example, names or marks of 'n' students we use a data structure called arrays.

Arrays are basically collections of fixed numbers of elements of a single type. Using arrays saves us from the time and effort required to declare each of the elements of the array individually.

The length of an array is established when the array is created. After creation, its length is fixed. For example: {1,2,3,4,5} is an array of integers.

Similarly, an array can be a collection of characters, Boolean, Double as well.

Declaring Array Variables

To use an array in a program, you must declare a variable to refer to the array, and you must specify the type (which once specified can't be changed) of the array the variable can reference. Here is the syntax for declaring an array variable –

Syntax

```
datatype [] arrayRefVar;           // preferred way. OR  
datatype arrayRefVar [];
```

Example:

```
int [] arr;          OR  
int arr [];
```

Creating Array

Declaring an array variable does not create an array (i.e. no space is reserved for array). Here is the syntax for creating an array –

```
arrayRefVar = new datatype [array Size];
```

Example:

```
arr=new int [20];
```

The above statement does two things –

- It creates an array using the new keyword [array Size].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Combining declaration of array variable, creating array and and assigning the reference of the array to the variable can be combined in one statement, as shown below –

```
datatype [] arrayRefVar = new datatype [array Size];
```

Array Indexes

In order to access different elements in an array -- all elements in the array are indexed and indexing starts from 0. So if there are 5 elements in the array then the first index will be 0 and last one will be 4. Similarly, if we have to store n values in an array, then indexes will range from 0 to n - 1.

```
int [] arr= { 1 , 2 , 3 , 4 , 5 };  
  
arr[0] = 1   arr[1] = 2   arr[2] = 3   arr[3] = 4   arr[4] = 5
```

Trying to retrieve an element from an invalid index will give an

ArrayIndexOutOfBoundsException.

Initialising an Array

In Single Line

Syntax of creating and initializing an array in single line ... dataType [] arrayRefVar = {value0, value1, ..., value};

```
int [] arr= {1,2,3,4,5,6,7};
```

Using Loop

```
public static void main(String[] args) {  
  
    int [] arr = new int [20];  
  
    Scanner Scan = new Scanner(System.in);  
  
    for (int i = 0 ; i < arr.length; i++){
```

```
    arr[i]=Scan.nextInt();  
}  
}
```

For Each Loop

This is a special type of loop to access array elements of array. But this loop can be used only to traverse an array, nothing can be changed in the array using this loop.

```
public class Solutions {  
    public static void main (String [] args) {  
        int [] arr= {10,20,30,40,50};  
        for (int i:arr)  
        {  
            System.out.print(i+" ");  
        }  
    }  
}
```

How are Arrays Stored?

Arrays in Java store one of two things: either primitive values (int, char,) or references (a.k.a pointers).

When an object is created by using “new”, memory is allocated on the heap and a reference is returned. This is also true for arrays, since arrays are objects.

int arr [] = new int [10]; **//here arr is a reference to the array and not the name of the array.**

Reassigning references and Garbage collector

All the reference variables (not final) can be reassigned again and again but their data type to whom they will refer is fixed at the time of their declaration.

```
public class Solutions {  
  
    public static void main (String [] args) {  
  
        int [] arr = new int [20]; // HERE arr is a reference  
        not array name...  
  
        int [] arr1 = new int [10];  
  
        arr = arr1; // We can re-assign arr to the arrays  
        which referred by arr1. Both arr and arr1 refer to the same arrays  
        now.  
  
    }  
  
}
```

In Above example, we create two reference variables arr and arr1. So now there are also 2 objects in the garbage collection heap.

If you assign arr1 reference variable to arr, then no reference will be present for the 20 integer space created earlier, so this block of memory can now be freed by Garbage Collector.

Garbage Collector

Live objects(We can think of them as blocks of memory for now) are tracked and everything else designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.

1. When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation.
2. This means there is no explicit deletion and no memory is given back to the operating system.

New objects are simply allocated at the end of the used heap

Passing Arrays to Functions?

Passing Array as function parameter

In Java programming language the parameter passing is always, ALWAYS, made by value. Whenever we create a variable of a type, we pass a copy of its value to the method.

Passing Reference Type // In case of array reference of array is passed

As we studied in lecture, we store references of Non--Primitive data types and access them via references, So in such cases the references of Non--Primitives are passed to function.

```
public class Solutions {  
    public static void print(int [] arr)  
    {  
        for (int i=0;i<5;i++)  
        {  
            System.out.print(arr[i]+" ");  
        }  
    }  
}
```

```
}
```

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

```
    int [] arr= {1,2,3,4,5};
```

```
    print(arr);      //Reference to array is passed
```

```
}
```

```
}
```

Similarly, when we pass an array to the increment function shown below then the reference(address) to the array is passed and not the array itself.

```
public class Solutions {
```

```
    public static void increment (int [] arr)
```

```
    {
```

```
        for (int i=0;i<5;i++)
```

```
        {
```

```
            arr[i]++;
        }
```

```
}
```

```
    }
```

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

```
        int [] arr= {1,2,3,4,5};
```

```
        increment(arr);
```

```
        for (int i=0;i<5;i++)
```

```
        {
```

```
        System.out.print(arr[i]+" ");
    }
}

}
```

Output:

2 3 4 5 6

Here reference to the array was passed. Thus inside increment function arr refers to the same array which was created in main. Hence the changes by increment function are performed on the same array and they will reflect in main.

Now, lets change code for increment function a little and make arr point to another array as shown in example given below.

```
public class Solutions {
    public static void increment (int [] arr)
    {
        int [] arr1= {1,2,3,4,5};
        arr=arr1;
        for (int i=0;i<5;i++)
        {
            arr[i]++;
        }
    }

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

```
int [] arr= {1,2,3,4,5};  
  
increment(arr);  
  
for (int i=0;i<5;i++)  
  
{  
  
    System.out.print(arr[i]+" ");  
  
}  
  
}
```

Output:

1 2 3 4 5

Here the changes done in main didn't reflect. Although here as well the reference to the array was passed, but in the first line inside the function we created another arr of size 5 and made arr refer to that array (without affecting the array created in main). Thus the changes this time won't reflect.

Returning Array from a Method

Similarly, as we pass reference as a function parameter we will return reference too in case of array.

```
class ArrayUse{  
  
    public static void main(String[] args){  
  
        int[] A = numbers();  
  
    }  
}
```

```
public static int[] numbers(){
    int[] A = new int[3];
    A[0] = 2;
    A[1] = 3;
    A[2] = 4;
    return A;
}
```

BufferedReader Class

So far we have learnt to take input with the help of Scanner class. There is another way to take input: using BufferedReader class. Let us dive deep into it.

BufferedReader:

It can be used to read input from a file as well as a keyboard. Since, throughout our course, since our source of input will be keyboard only, therefore, we will limit our discussions to taking input from keyboard.

Using this method, we read characters from the input stream. As we know, we have three streams:

1. System.in
2. System.out
3. System.err

In this method, we will use a class called InputStreamReader, which takes input byte by byte and decodes it into a character stream. After reading data from the source keyboard, the decoded data (character stream) is stored in a buffer (storage meant for temporary usage) and then the object of class BufferedReader reads from this buffer. You will get to know about this in detail in the lecture of Object Oriented Programming (OOP). It is explained in the OOP lecture that non-static functions or methods cannot be accessed by class. For accessing and invoking non-static methods, we use objects of class.

Note:-

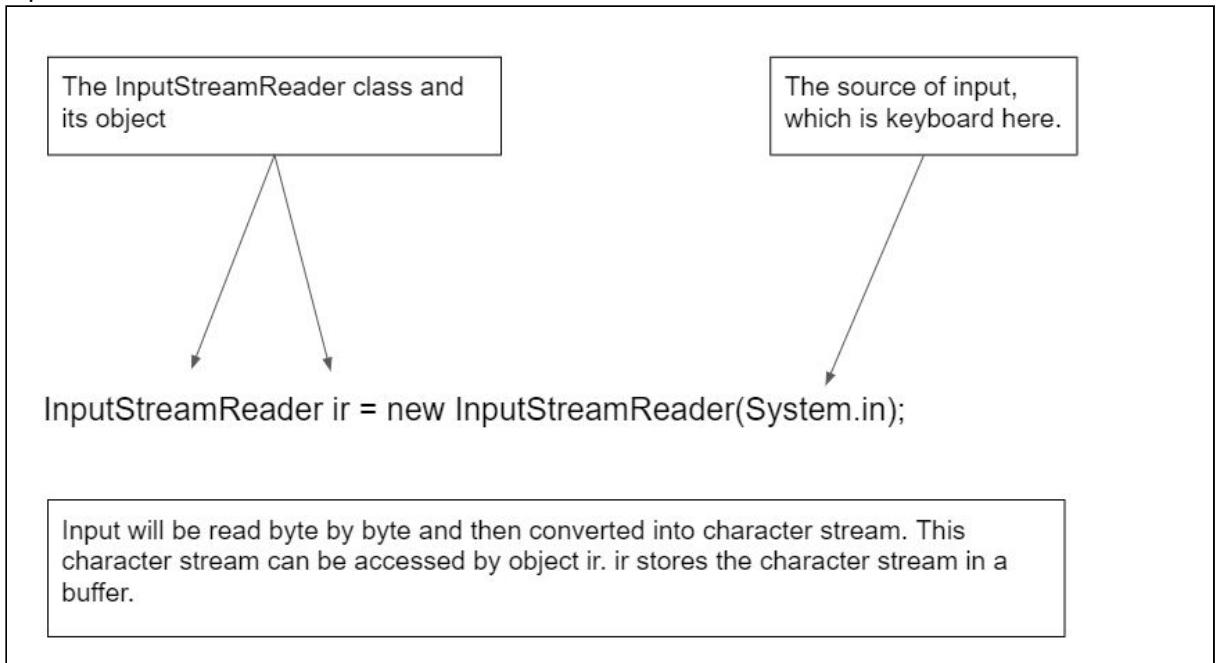
1. Since, methods of BufferedReader class and InputStreamReader deal in input and output operations, therefore, these methods may lead to errors in reading input or writing output. Therefore, the function must throw IOException.
2. BufferedReader and InputStreamReader are in the "io" package. Therefore, following statements must be included at the top of the code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
Or alternatively, we can include: import java.io.*;
```

Process

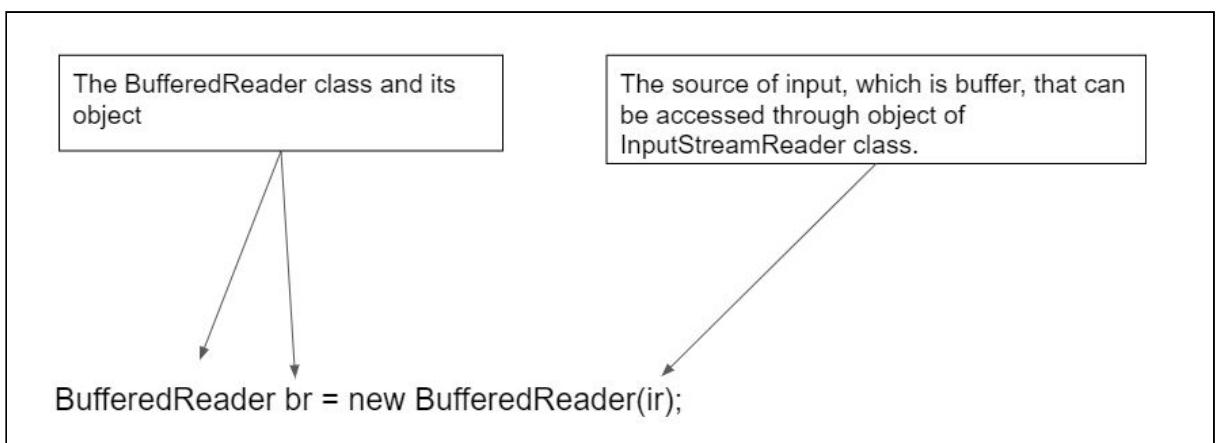
The complete can be divided into two steps:

1. In first step, we have to read the input or access the input using InputStreamReader



2. Now, in the second step, we have to read data from the buffer. This can be done using the object of the BufferedReader class. BufferedReader can read only characters or string. It does so using the following two methods:
 - a. `read()`: reads only single character
 - b. `readLine()`: reads multiple character or a string

The following syntax is used to read character using BufferedReader class:



How to take input of integer and floating point numbers

As BufferedReader method can only be used to find reading characters or string, therefore, for reading integers or floating-point values, we have to first read the input in the form of characters and then typecast it into integer or floating-point values.

We will static function parseInt for type casting character into integers and similarly, parseFloat for floating point values.

Examples:

```
int a = Integer.parseInt(br.readLine());
float b = Float.parseFloat(br.readLine());
String str = br.readLine();
```

Before getting our hands dirty in a more complex input format, we have to discuss split function. This function splits the given string on a certain delimiter. For example, if the delimiter is space, then it will divide the string into smaller substrings, which are separated by space. It will return an array of those substrings.

Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {

    public static void main(String[] args) throws NumberFormatException,
    IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

        String str = br.readLine();
        String[] strNums = str.split(" ");

        for (int i = 0; i < strNums.length; i++) {
            System.out.print(strNums[i]++);
        }
    }
}
```

Example Input:

11 12 13 14

Example Output:

11 12 13 14

More Examples

Let us suppose that we have to read input with the following input format:

"The first line contains an Integer 't' which denotes the number of test cases or queries to be run. Then the test cases follow.

First line of each test case or query contains an integer 'N' representing the size of the array/list.

Second line contains 'N' single space separated integers representing the elements in the array/list."

One Example of this input format is:

```
2
5
9 3 6 2 0
4
2 3 1 2
```

For reading this input, following code will be used:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {
    /*
        The object of class BufferedReader is made static because it
        is being used by multiple functions.
    */
    static BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

    public static int[] takeInput() throws IOException {
        int size = Integer.parseInt(br.readLine());
        int[] input = new int[size];

        if (size == 0) {
            return input;
        }

        String[] strNums;
        strNums = br.readLine().split("\\s");

        for (int i = 0; i < size; ++i) {
            input[i] = Integer.parseInt(strNums[i]);
        }

        return input;
    }

    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
```

```
        System.out.print(arr[i] + " ");
    }

    System.out.println();
}

public static void main(String[] args) throws NumberFormatException,
IOException {
    int t = Integer.parseInt(br.readLine());

    while (t > 0) {

        int[] input = takeInput();
        printArray(input);

        t -= 1;
    }
}
```

Searching And Sorting

Searching

Searching means to find out whether a particular element is present in a given sequence or not. There are commonly two types of searching techniques:

- Linear search (We have studied about this in **Arrays**)
- Binary search

In this module, we will be discussing binary search.

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In a nutshell, this search algorithm takes advantage of a collection of elements being already sorted by ignoring half of the elements after just one comparison.

Prerequisite: Binary search has one pre-requisite; unlike the linear search where elements could be any order, the array in **binary search** must be sorted,

The algorithm works as follows:

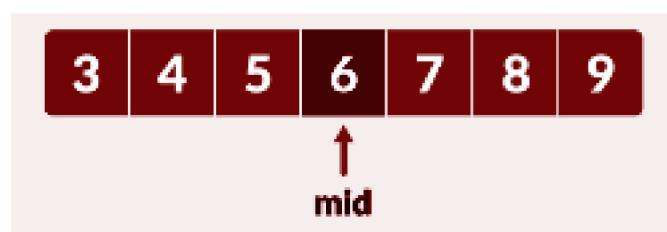
1. Let the element we are searching for, in the given array/list is X.
2. Compare X with the middle element in the array.
3. If X matches with the middle element, we return the middle index.
4. If X is greater than the middle element, then X can only lie in the right (greater) half subarray after the middle element, then we apply the algorithm again for the right half.
5. If X is smaller than the middle element, then X must lie in the left (lower) half, this is because the array is sorted. So we apply the algorithm for the left half.

Example Run

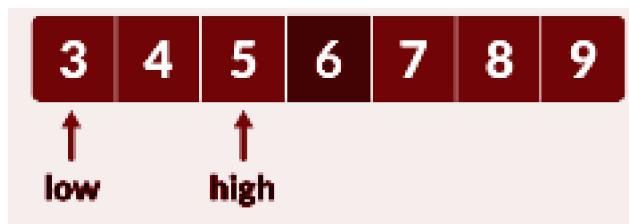
- Let us consider the array to be:



- Let $x = 4$ be the element to be searched.
- Set two pointers **low** and **high** at the first and the last element respectively.
- Find the middle element **mid** of the array ie. $\text{arr}[(\text{low}+\text{high})/2] = 6$.



- If $x == \text{mid}$, then return **mid**. Else, compare the element to be searched with **m**.
- If $x > \text{mid}$, compare x with the middle element of the elements on the right side of **mid**. This is done by setting **low** to $\text{low} = \text{mid} + 1$.
- Else, compare x with the middle element of the elements on the left side of **mid**. This is done by setting **high** to $\text{high} = \text{mid} - 1$.



- Repeat these steps until low meets high. We found 4:



Java Code

```
// Function to implement Binary Search Algorithm
public static int binarySearch(int arr[], int n, int x) {
    int start = 0, end = n - 1;
    // Repeat until the pointers start and end meet each other
    while(start <= end) {
        int mid = (start + end) / 2; // Middle Index
        if(arr[mid] == x) { // element found
            return mid;
        }
        else if(x < arr[mid]) { // x is on the left side
            end = mid - 1;
        }
        else { // x is on the right side
            start = mid + 1;
        }
    }

    return -1; // Element is not found
}

public static void main(String[] args) {
    int[] input = {3, 4, 5, 6, 7, 8, 9};
    int x = 4;
    System.out.print(binarySearch(input, n, x)); // print index
}
```

We will get the **output** of the above code as:

```
1 // Element found at index 1
```

Advantages of Binary search:

- This searching technique is faster and easier to implement.
- Requires no extra space.
- Reduces the time complexity of the program to a greater extent. (The term **time complexity** might be new to you, you will get to understand this when you will be studying algorithmic analysis. For now, just consider it as the time taken by a particular algorithm in its execution, and time complexity is determined by the number of operations that are performed by that algorithm i.e. time complexity is directly proportional to the number of operations in the program).

Sorting

Sorting is a permutation of a list of elements such that the elements are either in increasing (**ascending**) order or decreasing (**descending**) order.

There are many different sorting techniques. The major difference is the amount of **space** and **time** they consume while being performed in the program.

For now, we will be discussing the following sorting techniques:

- Selection sort
- Bubble sort
- Insertion sort

Let us now discuss these sorting techniques in detail.

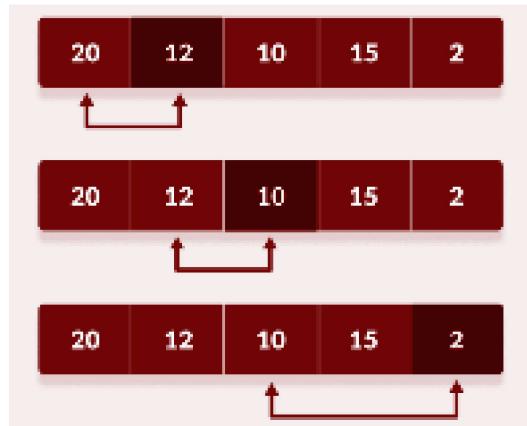
Selection Sort

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. The detailed algorithm is given below.

- Consider the given unsorted array to be:



- Set the first element as `minimum`.
- Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`.
- Compare `minimum` with the third element. Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.

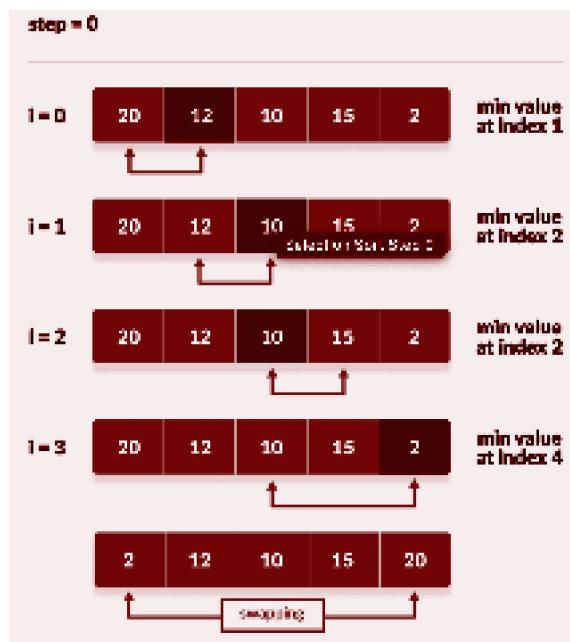


- After each iteration, `minimum` is placed in the front of the unsorted list.

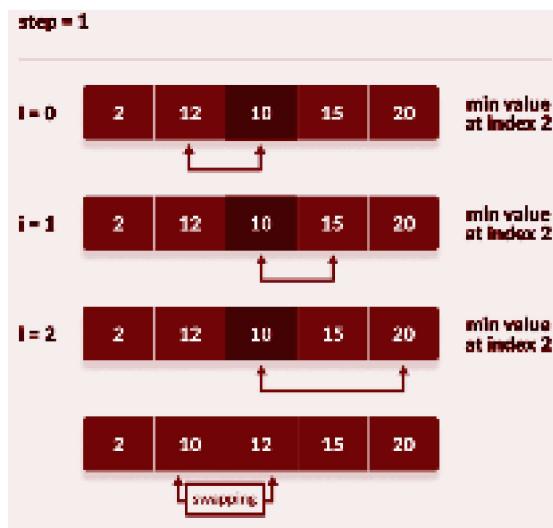


- For each iteration, indexing starts from the first unsorted element. These steps are repeated until all the elements are placed at their correct positions.

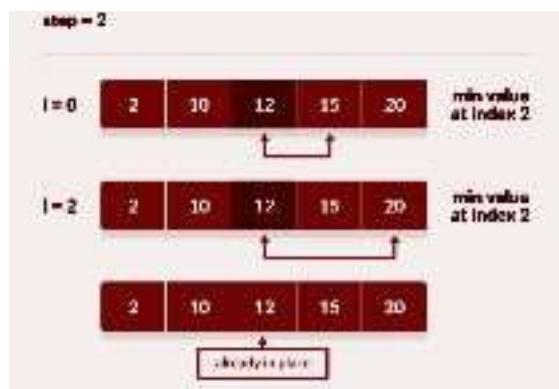
First Iteration



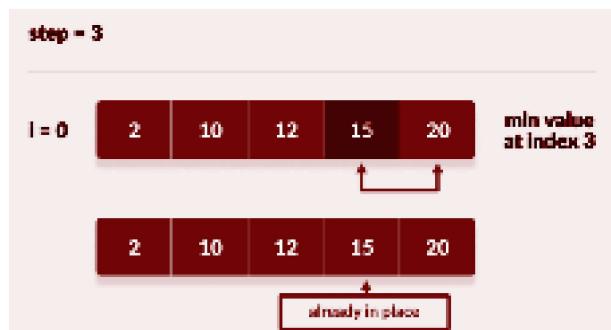
Second Iteration:



Third Iteration



Fourth Iteration



Java Code

```

public static void selectionSort(int input[], int n) {
    for(int i = 0; i < n-1; i++ ) {
        // Find min element in the array
        int min = input[i], minIndex = i;
        for(int j = i+1; j < n; j++) {
            // to sort in descending order, change < to > in this
            // line select the minimum element in each loop
            if(input[j] < min) {
                min = input[j];
                minIndex = j;
            }
        }
        // Swap
        int temp = input[i];
        input[i] = input[minIndex];
    }
}

```

```

        input[minIndex] = temp;
    }

public static void main(String[] args) {
    int input[] = {20, 12, 10, 15, 2};
    selectionSort(input, 6);
    for(int i = 0; i < 6; i++) {
        System.out.print(input[i] + " ");
    }
}

```

We will get the **output** of the above code as:

```
2 10 12 15 20 // sorted array
```

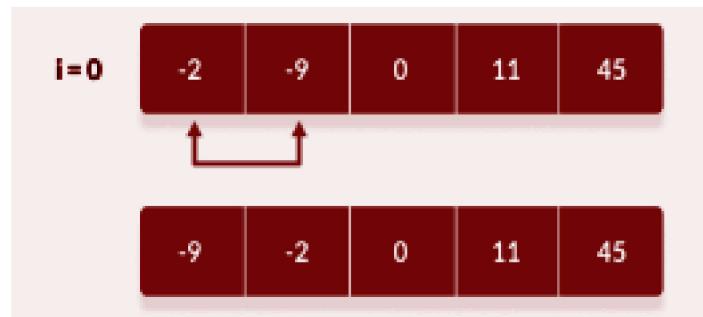
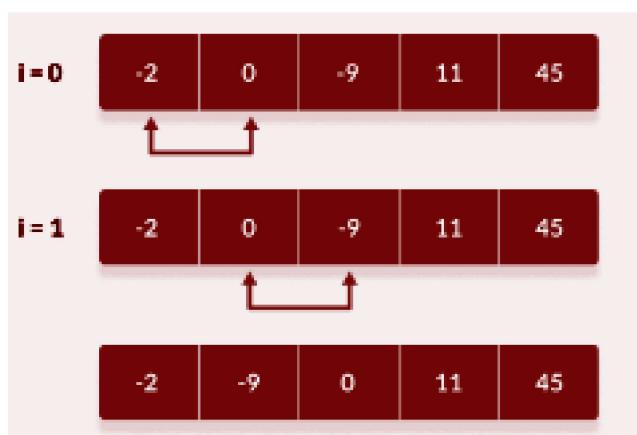
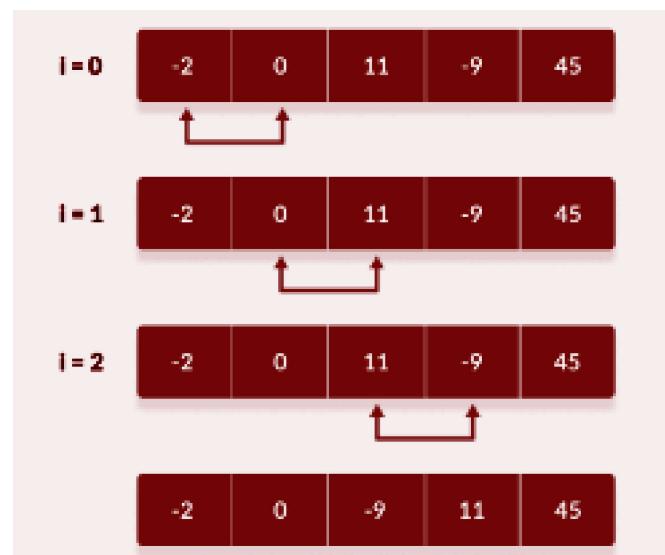
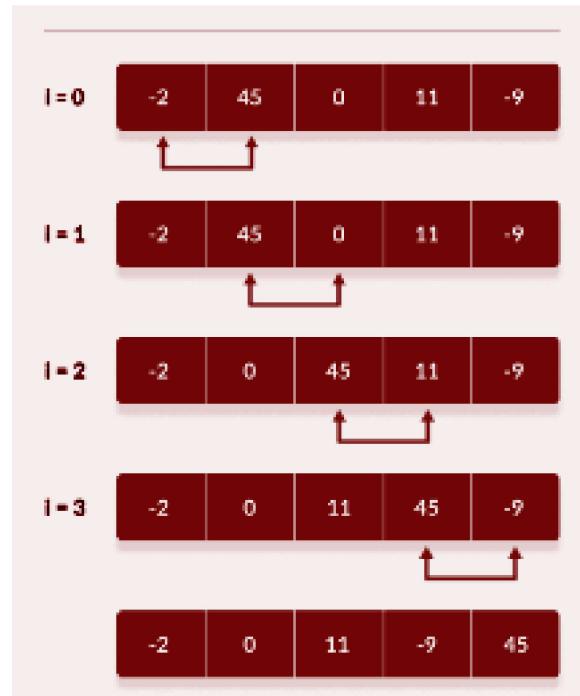
Bubble Sort

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

How does Bubble Sort work?

- Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.
- Now, compare the second and third elements. Swap them if they are not in order.
- The above process goes on until the last element.
- The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
- In each iteration, the comparison takes place up to the last unsorted element.
- The array is sorted when all the unsorted elements are placed at their correct positions.

Let the array be [-2, 45, 0, 11, -9].



Java Code

```
public static void bubbleSort(int array[], int size) {  
  
    // run Loops two times: one for walking through the array  
    // and the other for comparison  
    for (int step = 0; step < size - 1; ++step) {  
        for (int i = 0; i < size - step - 1; ++i) {  
  
            // To sort in descending order, change > to < in this line.  
            if (array[i] > array[i + 1]) {  
  
                // swap if greater is at the rear position  
                int temp = array[i];  
                array[i] = array[i + 1];  
                array[i + 1] = temp;  
            }  
        }  
    }  
  
    // driver code  
    public static void main(String[] args) {  
        int input[] = {-2, 45, 0, 11, -9};  
        bubbleSort(input, 6);  
        for(int i = 0; i < 6; i++) {  
            System.out.print(input[i] + " ");  
        }  
    }  
}
```

We will get the **output** of the above code as:

```
-9 -2 0 11 45 // sorted array
```

Insertion Sort

- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted
- Then, we select an unsorted card.
- If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.
- In the same way, other unsorted cards are taken and put in the right place. A similar approach is used by insertion sort.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration

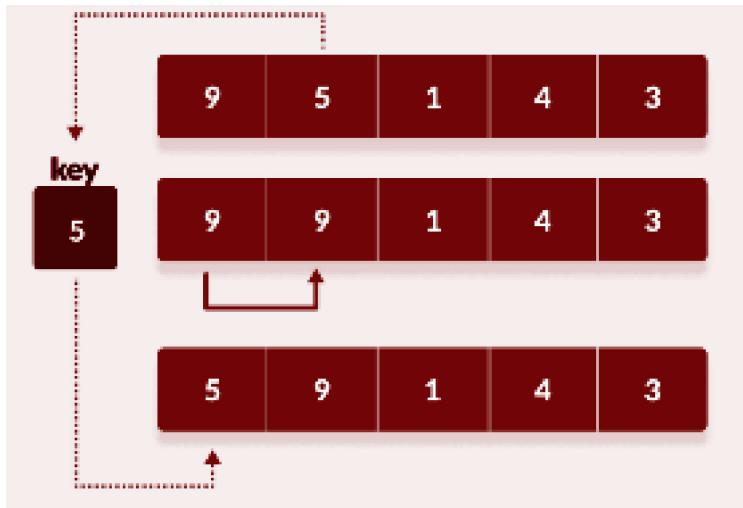
Algorithm

- Suppose we need to sort the following array.

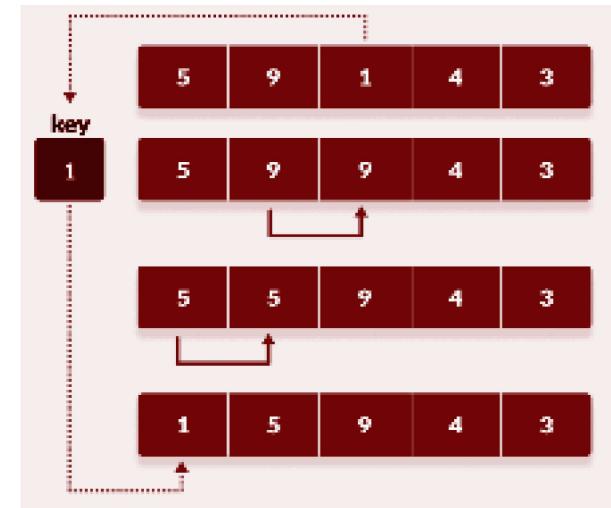
9	5	1	4	3
---	---	---	---	---

- The first element in the array is assumed to be sorted. Take the second element and store it separately in **key**.
- Compare the key with the first element. If the first element is greater than **key**, then **key** is placed in front of the first element.
- If the first element is greater than **key**, then **key** is placed in front of the first element.
- Now, the first two elements are sorted.
- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
- Similarly, place every unsorted element at its correct position.

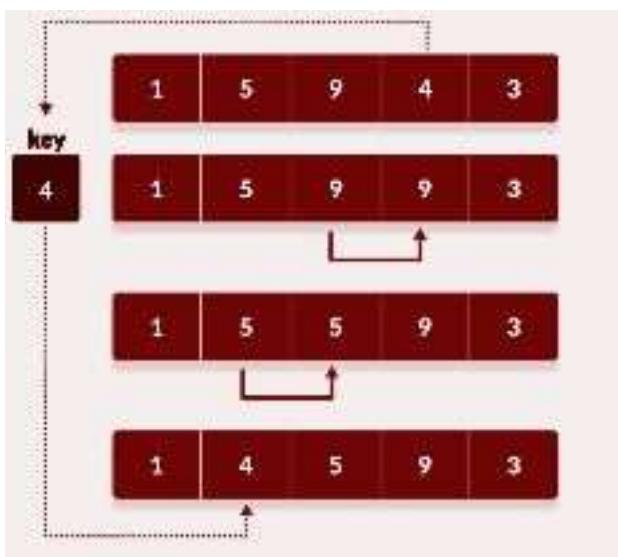
The various iterations are depicted below:



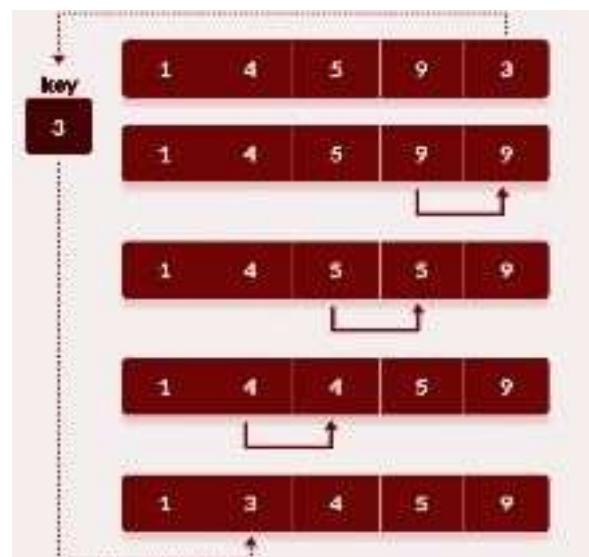
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Java Code

```

public static void insertionSort(int array[], int size) {
    int key, j;
    for(int i = 1; i<size; i++) {
        key = array[i];                      //take value
        j = i;
        // Compare key with each element on the left of it until an
        // element smaller than it is found
        while(j > 0 && array[j-1]>key) {
            array[j] = array[j-1];
            j--;
        }
        array[j] = key;                      //insert in right place
    }
}

// driver code
public static void main(String[] args) {
    int input[] = {9, 5, 1, 4, 3};
    insertionSort(input, 6);
    for(int i = 0; i < 6; i++) {
        System.out.print(input[i] + " ");
    }
}

```

We will get the **output** as:

```

1 3 4 5 9                         // sorted array

```

Now, practice different questions to get more familiar with the concepts. In the advanced course, you will study more types of sorting techniques.

Problem Statement - Merge 2 Sorted Arrays

In this problem, you are given 2 sorted arrays, you need to return a new array including all the elements of both given arrays in a sorted manner.

Algorithm

1. Create an array arr3[] of size n1 + n2.
2. Simultaneously traverse arr1[] and arr2[].
 - Pick a smaller of current elements in arr1[] and arr2[], copy this smaller element to the next position in arr3[] and move ahead in arr3[] as well as in the array whose element is picked.
 - Keep reiterating the above step, till either one of arr1[] and arr2[] are completely traversed.
3. Copy the remaining elements of the array which is left untraversed if there exists into the arr3, as the above loop breaks if any of the pointers exceeds their respective size.

Java Code

```

public static void mergeArrays(int arr1[], int[] arr2) {
    int n1 = arr1.length;
    int n2 = arr2.length;

    int[] arr3 = new int[n1+n2];
    int i = 0, j = 0, k = 0;
    while (i < n1 && j < n2){
        if (arr1[i] < arr2[j])
            arr3[k++] = arr1[i++];
        else
            arr3[k++] = arr2[j++];
    }

    // Store remaining elements of first array
    while (i < n1)
        arr3[k++] = arr1[i++];

    // Store remaining elements of second array
    while (j < n2)
        arr3[k++] = arr2[j++];

    return arr3;
}

```


Strings

What you will learn in this lecture?

- Strings in Java
- String Objects
- StringBuilder and StringBuffer
- Important Functions of Java

Strings in Java

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects(we study about objects in detail in OOPS lecture). The Java platform provides the String class to create and manipulate strings. The most direct and easiest way to create a string is to write:

```
String str = "Hello world";
```

In the above example, "Hello world!" is a string literal—a series of characters in code that is enclosed in double quotes. Whenever it encounters a string literal in code, the compiler creates a String object with its value—in this case, Hello world!.

Note: Strings in Java are immutable, thus we cannot modify its value. If we want to create a mutable string we can use StringBuffer and StringBuilder classes.

A String can be constructed by either:

directly assigning a string literal to a String reference -- just like a primitive, or

via the "new" operator and constructor, similar to any other classes(like arrays and scanner). However, this is not commonly--used and is not recommended.

For example,

```
String str1 = "Java is Amazing!";      // Implicit construction  
via string literal  
  
String str2 = new String("Java is Cool!"); // Explicit  
construction via new
```

In the first statement, str1 is declared as a String reference and initialized with a string literal "Java is Amazing". In the second statement, str2 is declared as a String reference and initialized via the new operator to contain "Java is Cool".

String literals are stored in a common pool called String pool. This facilitates sharing of storage for strings with the same contents to conserve storage. String objects allocated via new operator are stored in the heap memory(all non primitives created via new operator are stored in heap memory), and there is no sharing of storage for the same contents.

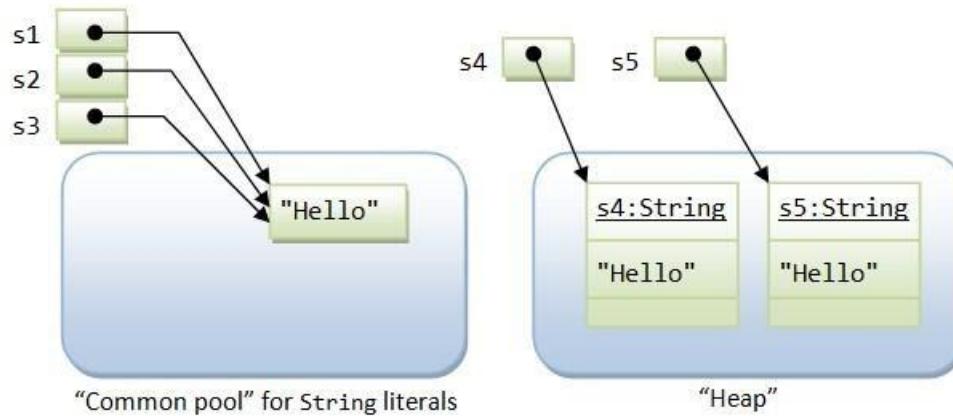
1. String Literal v/s String Object

As mentioned, there are two ways to construct a string: implicit construction by assigning a string literal or explicitly creating a String object via the new operator and constructor.

For example,

```

String s1 = "Hello";      // String literal
String s2 = "Hello";      // String literal
String s3 = s1;          // same reference
String s4 = new String("Hello"); // String object
String s5 = new String("Hello"); // String object
    
```



Java has provided a special mechanism for keeping the String literals -- in a so-called string common pool. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to conserve storage for frequently-used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap memory. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in heap even if two String objects have the same contents.

You can use the method equals() of the String class to compare the contents of two Strings. You can use the relational equality operator '==' to compare the references

(or pointers) of two objects. Study the following codes for s1 and s2 defined in code above:

```
s1 == s1;      // true, same pointer  
  
s1 == s2;      // true, s1 and s1 share storage in common pool  
  
s1 == s3;      // true, s3 is assigned same pointer as s1  
  
s1.equals(s3); // true, same contents  
  
s1 == s4;      // false, different pointers  
  
s1.equals(s4); // true, same contents  
  
s4 == s5;      // false, different pointers in heap  
  
s4.equals(s5); // true, same contents
```

2. String is Immutable

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and therefore is undesirable. Methods such as `toUpperCase()` might appear to modify the contents of a String object. In fact, a completely new String object is created and returned to the caller. The original String object will be deallocated, once there are no more references, and subsequently garbage-collected.

Because String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas).

For example,

```
// inefficient code

String str = "Hello";

for (int i = 1; i < 1000; ++i)

{
    str = str + i;

}
```

3. StringBuilder & StringBuffer

As explained earlier, Strings are immutable because String literals with the same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side--effects to other Strings sharing the same storage.

JDK provides two classes to support mutable strings: StringBuffer and StringBuilder (in core package `java.lang`) . A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without causing adverse side--effect to other objects.

The StringBuilder class was introduced in JDK 1.5. It is the same as the StringBuffer class, except that StringBuilder is not synchronized for multi--thread operations(you can read more about multi threading). However, for a single--thread program, StringBuilder, without the synchronization overhead, is more efficient.

Important Java Methods

1. String "Length" Method

String class provides an inbuilt method to determine the length of the Java String.

For example:

```
String str1 = "test string";
//Length of a String
System.out.println("Length of String: " + str.length());
```

2. String "indexOf" Method

String class provides an inbuilt method to get the index of a character in Java String.

For example:

```
String str1 = "the string";
System.out.println("Index of character 's': " +
str_Sample.indexOf('s')); // returns 5
```

3. String "charAt" Method

Similar to the above question, given the index, how do I know the character at that location? Simple one again!! Use the “charAt” method and provide the index whose character you need to find.

```
String str1 = "test string";
System.out.println("char at index 3 : " + str.charAt());
// output - 't'
```

4. String "CompareTo" Method

This method is used to compare two strings. Use the method “**compareTo**” and specify the String that you would like to compare.

Use “**compareToIgnoreCase**” in case you don’t want the result to be case sensitive. The result will have the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

```
String str = "test";
System.out.println("Compare To "test": " + str.compareTo("test"));
//Compare to -- Ignore case
System.out.println("Compare To    "test":      --    Case Ignored:    "
+ str.compareToIgnoreCase("Test"));
```

5. String "Contain" Method

Use the method “contains” to check if a string contains another string and specify the characters you need to check.

Returns true if and only if this string contains the specified sequence of char values.

```
String str = "test string";
System.out.println("Contains sequence ing: " + str.contains("ing"));
```

6. String "endsWith" Method

This method is used to find whether a string ends with a particular prefix or not.

Returns true if the character sequence represented by the argument is a suffix of the character sequence represented by this object.

```
String str = "star";
System.out.println("EndsWith character 'r': " + str.endsWith("r"));
```

7. String "replaceAll" & "replaceFirst" Method

Java String Replace, replaceAll and replaceFirst methods. You can specify the part of the String you want to replace and the replacement String in the arguments.

```
String str = "sample string";
System.out.println("Replace sample with test: " +
str.replace("sample", "test"));
```

8. String Java "toLowerCase" & Java "toUpperCase"

Use the "toLowerCase()" or "ToUpperCase()" methods against the Strings that need to be converted.

```
String str = "TEST string";
System.out.println("Convert to LowerCase: " + str.toLowerCase());
```

```
System.out.println("Convert to UpperCase: " + str.toUpperCase());
```

Other Important Java Strings Methods:

No.	Method	Description
1	String substring(int beginIndex)	returns substring for given begin index
2	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
3	boolean isEmpty()	checks if string is empty
4	String concat(String str)	concatenates specified string
5	String replace(char old, char new)	replaces all occurrences of specified char value
6	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
7	String[] split(String regex)	returns splitted string matching regex
8	String[] split(String regex, int limit)	returns splitted string matching regex and limit
9	int indexOf(int ch)	returns specified char value index
10	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index

11	int indexOf(String substring)	Returns specified substring index
12	int indexOf(String substring, int fromIndex)	Returns specified substring index starting with given index
13	String trim()	removes beginning and ending spaces of this string.
14	static String valueOf(int value)	converts given type into string. It is overloaded.

Some Important Key Points about Java Strings

1. Strings are not NULL terminated in Java: Unlike C and C++, String in Java doesn't terminate with null character. Instead String is an Object in Java and backed by a character array. You can get the character array used to represent String in Java by calling **toCharArray()** method of **java.lang.String** class of JDK.
2. Internally, String is stored as a character array only.
3. String is a Immutable and Final class, that is, once created the value cannot be altered. Thus String objects are called immutable.
4. The Java Virtual Machine(JVM) creates a memory location especially for Strings called String Constant Pool. That's why String can be initialized without the '**new**' keyword.

Object-Oriented Programming (OOPS-1)

Introduction to OOPS

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of “*objects*” and “*classes*” that contain data and methods. The primary purpose of OOP is to increase the flexibility and maintainability of programs. It is used to structure a software program into simple, reusable pieces of code **blueprints** (called *classes*) which are used to create individual instances of *objects*.

What is an Object?

The object is an entity that has a state and a behavior associated with it. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Objects have states and behaviors. Arrays are objects. You've been using objects all along and may not even realize it. Apart from primitive data types, objects are all around in java.

What is a Class?

A class is a **blueprint** that defines the variables and the methods (Characteristics) common to all objects of a certain kind.

Example: If **Car** is a class, then **Maruti 800** is an object of the **Car** class. All cars share similar features like 4 wheels, 1 steering wheel, windows, breaks etc. Maruti 800 (The **Car** object) has all these features.

Classes vs Objects (Or Instances)

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.

A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for defining a **Car**, but it doesn't contain the name or top-speed of any specific **Car**.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.

Put another way, a class is like a form or questionnaire. An **instance** is like a form that has been filled out with information. Just like many people can fill out the same form with their unique information, many instances can be created from a single class.

Defining a Class in Java

All class definitions start with the **class** keyword, which is followed by the name of the class.

Here is an example of a **Car** class:

```
class Car{  
}
```

Note: Java class names are written in *CapitalizedWords* notation by convention. **For example**, a class for a specific model of Car like the Bugatti Veyron would be written as **BugattiVeyron**. The first letter is capitalized. This is just a good programming practice.

The **Car** class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Car objects should have. There are several properties that we can choose from, including **color**, **brand**, and **top-speed**. To keep things simple, we'll just use **color** and **top-speed**.

Constructor

- Constructors are generally used for instantiating an object.
- The task of a constructor is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Java, constructor for a class must be of the same name as of class.
- In Java, constructors don't have a return type.

Syntax of Constructor Declaration

```
public Car(){  
    // body of the constructor  
}
```

Types of constructors

- **Default Constructor:** The default constructor is a simple constructor that doesn't accept any arguments.
- **Parameterized Constructor:** A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its arguments provided by the programmer.

Class Attributes or Data Members

Class attributes are attributes that may or may not have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of **constructor**.

For example, the following **Car** class has a class attribute called **name** and **topSpeed** with the value "Black":

```
class Car{  
    String name;  
    int topSpeed;  
  
    public Car(String carName, int speed){  
        name = carName;  
        topSpeed = speed;  
    }  
}
```

- Class attributes are defined directly beneath the first line of the class name.
- When an instance of the class is created, the class attributes are automatically created.

The **this** Parameter

- The **this** parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
- We can use **this** every time but the main use of **this** comes in picture when the attributes and data members share the same name.

Let's update the Car class with the **Car** method that creates **name** and **topSpeed** attributes:

```
class Car{
```

```
String name;  
int topSpeed;  
  
public Car(String name, int topSpeed){  
    this.name = name;  
    this.topSpeed = topSpeed;  
}  
}
```

In the body of **constructor**, two statements are using the self variable:

1. `this.name = name` assigns the value of the `name` parameter to `name` attribute.
2. `this.topSpeed= topSpeed` assigns the value of the `topSpeed` parameter to `topSpeed` attribute.

All **Car** objects have a `name` and a `topSpeed`, but the values for the `name` and `topSpeed` attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

Now that we have a **Car** class, let's create some cars!

Instantiating an Object in Java

Creating a new object from a class is called instantiating an object. Consider the previous simpler version of our **Car** class:

```
Car c1 = new Car("Creta", 200);
```

You can instantiate a new **Car** object by typing the name of the class, followed by opening and closing parentheses:

Now, instantiate a second **Car** object:

```
Car c2 = new Car("i 10", 190);
```

The new **Car** instance is located at a different memory address. That's because it's an entirely new instance and is completely different from the first **Car** object that you instantiated.

Even though **c1** and **c2** are both instances of the **Car** class, they represent two distinct objects in memory. So they are not equal.

After you create the **Car** instances, you can access their instance attributes using dot notation:

```
System.out.println(c1.name);  
System.out.println(c2.topSpeed);
```

Output will be:

Creta

190

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. The values of these attributes **can** be changed dynamically:

```
c1.topSpeed= 250  
c2.name = "Jeep"
```

In this example, you change the **topSpeed** attribute of the **c1** object to **250**. Then you change the **name** attribute of the **c2** object to **"Jeep"**.

Note:

- The key takeaway here is such custom objects are **mutable** by default i.e. their states can be modified.

Access Modifiers

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package

We write the type of modifier before every method or data member.

Final Keyword

If you make any variable final, you cannot change the value of the final variable (It will be constant).

```
class Pen{  
    final int price = 15;  
}  
  
public class MCQs {  
    public static void main(String[] args) {  
        Pen p = new Pen();  
        p.price = 20;  
        System.out.println(p.price);  
    }  
}
```

There is a final variable price, we are going to change the value of this variable, but it can't be changed because the final variable once assigned a value can never be changed. Therefore this will give a **compile time error**.

Static Keyword

The static variable gets memory only once in the class area at the time of class loading.

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class rather than an instance of the class.

```
class Car{
    static int year;
    String company_name;

}

class NewCar{
    public static void main (String[] args) {
        Car c=new Car();
        Car.year=2018;
        c.company_name="KIA";
        Car d=new Car();
        System.out.print(d.year);
    }
}
```

Now in this code, when we look carefully, even when the new instance of Car is created, the year is defined by the first instance of the Car and it tends to remain

the same for all instances of the object. But here's the catch, we can change the value of this static variable from any instance. Here the output will be 2018 for every instance as long as it is not changed.

Static Functions

If you apply a static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data members and can change the value of it.

```
class Test{  
    static int a = 10;  
    static int b;  
    static void fun(){  
        b = a * 4;  
    }  
}  
  
class NewCar{  
    public static void main(String[] args) {  
        Test t=new Test();  
        t.fun();  
        System.out.print(t.a + t.b);  
    }  
}
```

When t.fun() is called, it will simply change the value of b to 40.

Therefore the output of this code will be 50.

Object-Oriented Programming (OOPS-2)

What you will learn in this lecture?

- Components of OOPs.
- Access modifiers with inheritance and protected modifiers.
- All about exception handling.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of class as private and the class is exposed to the end user or the world without providing any details behind implementation using the abstraction concept, so it is also known as combination of data-hiding and abstraction..

- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Inheritance

- Inheritance is a powerful feature in Object-Oriented Programming.
- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.
- The class which inherits the properties of the other is known as **subclass** (*derived class or child class*) and the class whose properties are inherited is known as **superclass** (*base class, parent class*).

Super Keyword:

The super keyword in Java is a reference variable which is used to refer to an immediate parent class object.

Whenever you create an instance of a subclass, an instance of the parent class is created implicitly which is referred to by a super reference variable.

Let us take a real-life example to understand inheritance. Let's assume that **Human** is a class that has properties such as **height, weight, age**, etc and functionalities (or methods) such as **eating(), sleeping(), dreaming(), working()**, etc.

Now we want to create **Male** and **Female** classes. Both males and females are humans and they share some common properties (like **height**, **weight**, **age**, etc) and behaviors (or functionalities like **eating()**, **sleeping()**, etc), so they can inherit these properties and functionalities from the **Human** class. Both males and females also have some characteristics specific to them (like men have short hair and females have long hair). Such properties can be added to the **Male** and **Female** classes separately.

This approach makes us write less code as both the classes inherited several properties and functions from the superclass, thus we didn't have to re-write them. Also, this makes it easier to read the code.

Java Inheritance Syntax

```
class SuperClass{  
    // Body of parent class  
}  
  
class SubClass extends SuperClass{  
    // Body of derived class  
}
```

To inherit properties of the parent class, **extends** keyword is used followed by the name of the parent class.

Example of Inheritance in Java

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called **Polygon** defined as follows.

```
class Polygon{  
    int n;  
    int[] sides;
```

```

public Polygon(int no_of_sides){ //Constructor
    this.n = no_of_sides;
    this.sides = new int[no_of_sides];
}

void inputSides(){ //Take user input for side lengths
    Scanner s = new Scanner(System.in);
    for (int i=0; i<this.n; i++){
        System.out.println("Enter side: ");
        this.sides[i] = s.nextInt();
    }
}

void displaySides(): //Print the sides of the polygon
    for (int i=0; i<this.n; i++){
        System.out.println("Side " + i+1 + " is" + this.sides[i]);
    }
}

```

This class has **data attributes** to store the number of sides **n** and magnitude of each side as a list called **sides**.

The **inputSides()** method takes in the magnitude of each side and **dispSides()** displays these side lengths.

Now, a triangle is a polygon with 3 sides. So, we can create a class called **Triangle** which inherits from **Polygon**. In other words, we can say that every triangle is a polygon. This makes all the attributes of the **Polygon** class available to the **Triangle** class.

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass using super keyword:

```
super.Polygon(<Parameter1>, <Parameter2>, ...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' constructor/ function, otherwise it will throw an error.

The **Triangle** class can be defined as follows.

```
class Triangle extends Polygon{
    public Triangle(){
        super.Polygon(3); //Calling constructor of superclass
    }

    void findArea(){
        int a = super.sides[0];
        int b = super.sides[1];
        int c = super.sides[2];
        // calculate the semi-perimeter
        int s = (a + b + c) / 2;
        int area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        print('The area of the triangle is ' + area);
    }
}
```

However, the class **Triangle** has a new method **findArea()** to find and print the area of the triangle. This method is only specific to the **Triangle** class and not **Polygon** class.

Here, even though we did not define methods like **inputSides()** or **displaySides()** for class **Triangle** separately, we will be able to use them. If an attribute is not found in the subclass itself, the search continues to the superclass.

Access Modifiers

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. There are four types of access modifiers available in java, which are **Public**, **Private**, and **Protected** in a class, then there is a **default** case (we don't write any keyword in this case), which lies somewhere in between public and private.

Public Modifier

The public access modifier is specified using the keyword `public`.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Consider the given example:

```
// Package 1
public class Student{
    public String name; // public member
    public int age; // public member

    // constructor
    public void Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}

-----
// Package 2
```

```
class Test{  
    public static void main(String[] args) {  
        Student obj = Student("Boy", 15)  
        System.out.println(obj.age); //calling public member of class  
        System.out.println(obj.name); //calling public member  
    }  
}
```

We will get the output as:

```
10  
Boy
```

We will be able to access both **name** and **age** of the object from outside the class and package as they are **public**. However, this is not a good practice due to *security concerns*.

Private Modifier

The members of a class that are declared **private** are accessible within the class only. A private access modifier is the most secure access modifier. Data members of a class are declared private by adding a private keyword before the data member of that class. Consider the given example:

```
// Package 1  
public class Student{  
    private String name; // private member  
    public int age; // public member  
  
    // constructor  
    public void Student(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
// Package 2
class Test{
    public static void main(String[] args) {
        Student obj = Student("Boy", 15)
        System.out.println(obj.age); //calling public member of class
        System.out.println(obj.name); //calling private member
    }
}
```

We will get the output as:

```
10
```

```
AttributeError: 'Student' object has no attribute 'name'
```

We will get an **AttributeError** when we try to access the **name** attribute. This is because **name** is a **private** attribute and hence it cannot be accessed from outside the class.

Note: We can even have **public** and **private** methods.

Private and Public modifiers with Inheritance

- The subclass will be able to access any **public** method or instance attribute of the superclass.
- The subclass will not be able to access any **private** method or instance attribute of the superclass.

Protected Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared **protected** by adding a **protected** keyword before the data member of that class.

The given example will help you get a better understanding:

```
// superclass
public class Student{
    protected String name; // private member

    // constructor
    public void Student(String name){
        this.name = name;
    }
}
```

This is the parent class **Student** with a **protected** instance attribute **name**. Now consider a subclass of this class:

```
class Display extends Student{
    // constructor
    public Display(String name){
        super.Student(name);
    }
    public void displayDetails(){
        // accessing protected data members of the superclass
        System.out.println("Name: ", super.name);
    }
}

class Test{
    public static void main(String[] args) {
        Display obj = Student("Boy");    // creating objects of the
                                         // derived class
        obj.displayDetails();          // calling public member functions
                                         // of the class
        System.out.println(obj.name);   // trying to access
                                         // protected attribute
    }
}
```

This class **Display** inherits the **Student** class. The method `displayDetails()` accesses the **protected** attribute `_name`. Further, we try to access it again outside this class.

Output:

```
Name: Boy
AttributeError: 'Display' object has no attribute 'name'
```

You can observe that we were able to access the **protected** attribute `_name` from inside the `displayDetails()` method in the subclass. However, we were not able to access it outside the subclass and we got an **AttributeError**. This justifies the definition of the **protected** modifier.

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
 - Runtime Polymorphism
1. **Compile-time polymorphism:** It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java supports the Operator Overloading with only the '+' symbol.** '+' symbol in java works for adding two integer numbers and it can also be used for string concatenation.

Function/ Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

Example 1: Polymorphism in addition(+) operator

We know that the + operator is used extensively in Java programs. But, it does not have a single usage. For integer data types, the + operator is used to perform arithmetic addition operation.

```
int num1 = 1;  
int num2 = 2;  
System.out.println(num1+num2);
```

Hence, the above program outputs **3**.

Similarly, for string data types, the + operator is used to perform concatenation.

```
String str1 = "Java"  
String str2 = "Programming"  
print(str1+" "+str2)
```

As a result, the above program outputs **"Java Programming"**.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of **polymorphism** in Python.

Example 2: Polymorphism with methods/ functions in Java

Let's look at an example.

```
// Java program for Method overloading
```

```

class MultiplyFun {
    // Method with 2 parameter
    static int Multiply(int a, int b){
        return a * b;
    }
    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c){
        return a * b * c;
    }
}

class Test{
    public static void main(String[] args) {
        System.out.println(MultiplyFun.Multiply(2, 4));
        System.out.println(MultiplyFun.Multiply(2, 7, 3));
    }
}
  
```

Output

8
42

2. Runtime Polymorphism: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

It occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Let us see this in code:

```

// Java program for Method overriding

class Parent {
    void Print() {
        System.out.println("parent class");
    }
}
  
```

```
}
```

```
class subclass1 extends Parent {
    void Print() {
        System.out.println("subclass1");
    }
}

class subclass2 extends Parent {
    void Print() {
        System.out.println("subclass2");
    }
}

class TestPolymorphism3 {
    public static void main(String[] args) {
        Parent a;

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Output

```
subclass1
subclass2
```

Exception Handling

Error in Java can be of two types i.e. normal unavoidable errors and Exceptions.

- Errors are the problems in a program due to which the program will stop the execution.
- On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Difference between Syntax Errors and Exceptions

Error: An Error “indicates serious problems that a reasonable application should not try to catch.”

Both Errors and Exceptions are the subclasses of `java.lang.Throwable` class. Errors are the conditions which cannot get recovered by any handling techniques. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error. Also, there are syntax errors that are caused by the wrong syntax in the code. It leads to the termination of the program in compile time itself.

Example:

When you are using recursion to solve any problem, you must have seen errors which say “Stack overflow”. In your case, this might have arised due to the incorrect or absence of base case. But this has a deeper explanation. This stack overflow error may also arise when the input is huge and to solve the problem you need too many recursive calls one above the other, this will lead to overflow of the main stack space provided. So there comes the need to solve this problem iteratively.

You will practically experience these errors in Dynamic Programming lecture.

For a 64 bits Java 8 program with minimal stack usage, the maximum number of nested method calls is about 7 000. Generally, we don't need more, except in very specific cases. You can

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
int marks = 10000;  
int a = marks / 0;  
System.out.println(a);
```

Output:

```
ZeroDivisionError: division by zero
```

The above example raised the **ZeroDivisionError** exception, as we are trying to divide a number by 0 which is not defined.

Exceptions in Java

- Java has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Java interpreter stops the current process and passes it to the calling process until it is handled.
- If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and the program comes to a sudden unexpected halt.

Some Common Exceptions

A list of common exceptions that can be thrown from a standard Java program is given below.

- **ArithmaticException**

It is thrown when an exceptional condition has occurred in an arithmetic operation.

- **ArrayIndexOutOfBoundsException**

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

- **ClassNotFoundException**

This Exception is raised when we try to access a class whose definition is not found

- **FileNotFoundException**

This Exception is raised when a file is not accessible or does not open.

- **IOException**

It is thrown when an input-output operation failed or interrupted

- **InterruptedException**

It is thrown when a thread is waiting , sleeping , or doing some processing, and it is interrupted.

- **NoSuchFieldException**

It is thrown when a class does not contain the field (or variable) specified

- **NoSuchMethodException**

It is thrown when accessing a method which is not found.

- **NullPointerException**

This exception is raised when referring to the members of a null object. Null represents nothing

- **NumberFormatException**

This exception is raised when a method could not convert a string into a numeric format.

- **RuntimeException**

This represents any exception which occurs during runtime.

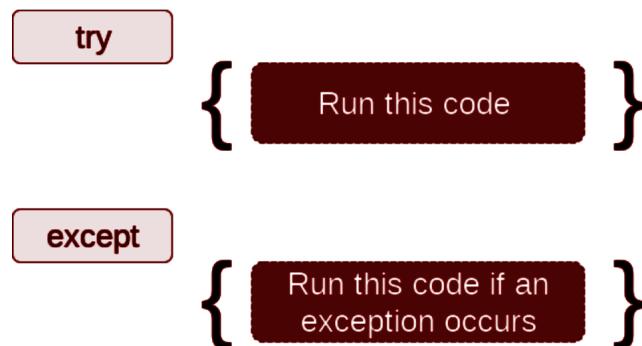
- **StringIndexOutOfBoundsException**

It is thrown by String class methods to indicate that an index is either negative than the size of the string

Catching Exceptions

In Java, exceptions can be handled using **try-catch** blocks.

- If the Java program contains suspicious code that may throw the exception, we must place that code in the **try** block.
- The **try** block must be followed by the **catch** statement, which contains a block of code that will be executed in case there is some exception in the **try** block.
- We can thus choose what operations to perform once we have caught the exception.



- Here is a simple example:

```

int[] arr = {1, 0, 2};
for (int ele : arr){
    try{ //This block might raise an exception while executing
        System.out.println("The entry is" + ele);
        int r = 1/int(ele);
    }
    catch(Exception e) { //This block executes in case of an
                          // exception in "try"
        System.out.println("Oops! An error occurred: "+e.toString());
    }
    System.out.println();
}

```

We get the output to this code as:

```

The entry is 1

The entry is 0
Oops! An error occurred: java.lang.ArithmetricException: / by zero

The entry is 2

```

- In this program, we loop through the values of an array arr.
- As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the catch block is skipped and normal flow continues.
- But if any exception occurs, it is caught by the catch block (second value of the array).
- Here, we print the name of the exception using the `e.toString()` function.
- We can see that element 0 causes ZeroDivisionError.

Every exception in Java inherits from the base **Exception** class.

Catching Specific Exceptions in Java

- In the above example, we did not mention any specific exception in the `catch` clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions a `catch` clause should catch.
- A try clause can have any number of `catch` clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- You can use multiple `catch` blocks for different types of exceptions.

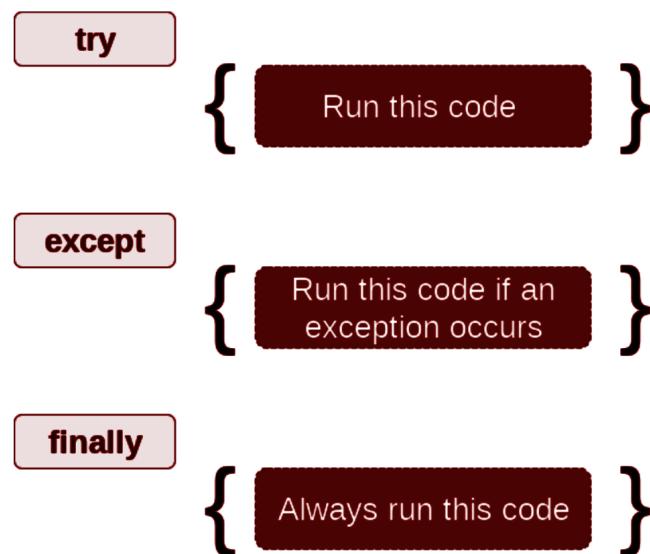
Here is an example to understand this better:

```
try{
    a=10/0;
}
catch(ArithmetricError e){
    System.out.println("Arithmetric Exception");
}
catch(IOException e){
    System.out.println("input output Exception");
}
```

Output:

```
Arithmetric Exception
```

finally Statement



The **try** statement in Java can have an optional **finally** clause. This clause is executed no matter what and is generally used to release external resources. Here is an example of file read and close to illustrate this:

```

FileReader f = null;
try{
    f = new FileReader(file);
    BufferedReader br = new BufferedReader(f);
    String line = null;
}
catch (FileNotFoundException fnf) {
    fnf.printStackTrace();
}
finally {
    if( f != null)
        f.close();
}

```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.



Java Foundation with Data Structures

Topic: Recursion

Recursion

a. What is Recursion?

In previous lectures, we used iteration to solve problems. Now, we'll learn about recursion for solving problems which contain smaller sub-problems of the same kind.

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. By same nature it actually means that the approach that we use to solve the original problem can be used to solve smaller problems as well.

So in other words in recursion a function calls itself to solve smaller problems. Recursion is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts and the code is also shorter and easier to understand.

b. How Does Recursion Work?

We can define the steps of a recursive solution as follows:

1. Base Case:

A recursive function must have a terminating condition at which the function will stop calling itself. Such a condition is known as a base case.

2. Recursive Call:

The recursive function will recursively invoke itself on the smaller version of problem. We need to be careful while writing this step as it is important to correctly figure out what your smaller problem is on whose solution the original problem's solution depends.

3. Small Calculation:

Generally we perform a some calculation step in each recursive call. We can perform this calculation step before or after the recursive call depending upon the nature of the problem.

It is important to note here that recursion uses stack to store the recursive calls. So, to avoid memory overflow problem, we should define a recursive solution with minimum possible number of recursive calls such that the base condition is achieved before the recursion stack starts overflowing on getting completely filled.

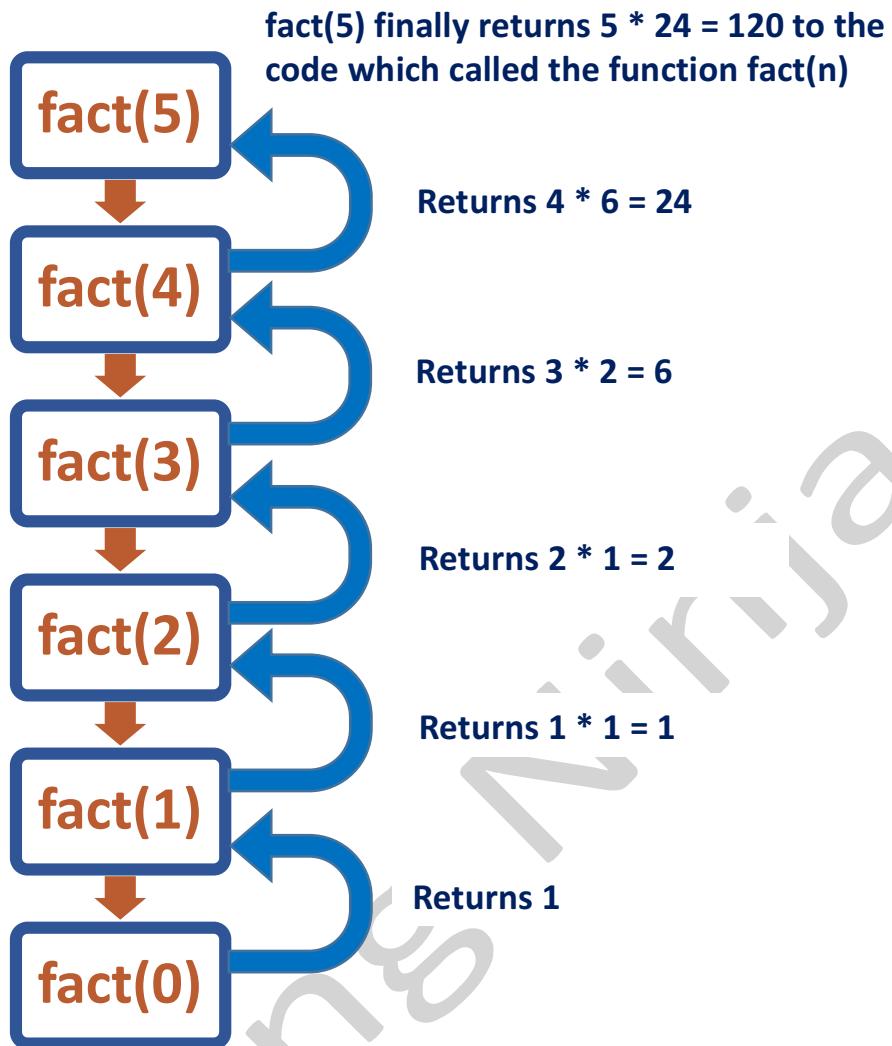
Now, let us look at an example to calculate factorial of a number using recursion.

Example Code 1:

```
public class Solution{  
  
    public static int fact(int n)  
    {  
        if(n==0) //Base Case  
        {  
            return 1;  
        }  
        return n * fact(n-1); //Recursive call with small  
        calculation  
    }  
  
    public static void main()  
    {  
        int num;  
        Scanner s = new Scanner(Syatem.in);  
        num = s.nextInt();  
        System.out.println(fact(num));  
        return 0;  
    }  
}  
  
Output:  
120 //For num=5
```

Explanation:

Here, we called factorial function recursively till number became 0. Then, the statements below the recursive call statement were executed. We can visualize the recursion tree for this function, where let n=5, as follows:



We are calculating the factorial of $n=5$ here. We can infer that the function recursively calls $\text{fact}(n)$ till n becomes 0, which is the base case here. In the base case, we returned the value 1. Then, the statements after the recursive calls were executed which returned $n * \text{fact}(n-1)$ for each call. Finally, $\text{fact}(5)$ returned the answer 120 to $\text{main}()$ from where we had invoked the $\text{fact}()$ function.

Now, let us look at another example to find n^{th} Fibonacci number . In Fibonacci series to calculate n^{th} Fibonacci number we can use the formula $F(n) = F(n - 1) + F(n - 2)$ i.e. n^{th} Fibonacci term is equal to sum of $n-1$ and $n-2$ Fibonacci terms. So let's use this to write recursive code for n^{th} Fibonacci number.

Example Code 2:

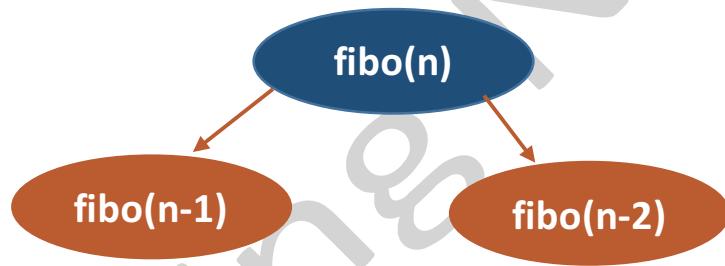
// Recursive function:

```
int fibo(int n) {  
    if(n==0 || n==1) { //Base Case  
        return n;  
    }  
    int a = fibo(n-1); //Recursive call  
    int b = fibo(n-2); //Recursive call  
    return a+b; //Small Calculation and return statement  
}
```

Explanation:

As we are aware of the Fibonacci Series (0, 1, 1, 2, 3, 5, 8,... and so on), let us assume that the index starts from 0, so, 5th Fibonacci number will correspond to 5; 6th Fibonacci number will correspond to 8; and so on.

Here, in recursive Fibonacci function, we have made two recursive calls which are depicted as follows:



Note: One thing that we should be clear about is that both recursive calls don't happen simultaneously. First fibo(n-1) is called, and only after we have its result and store it in "a" we move to next statement to calculate fibo(n - 2).

It is interesting to note here that the concept of recursion is based on the mathematical concept of **PMI** (Principle of Mathematical Induction). When we use PMI to prove a theorem, we have to show that the base case (usually for x=0 or x=1) is true and, the induction hypothesis for case x=k is true must imply that case x=k+1 is also true. We can now understand how the steps which we followed in recursion are based on the induction steps, as in recursion also, we have a base case while the assumption corresponds to the recursive call.

Recursion-1

Introduction

The process in which a function calls itself is called **recursion** and the corresponding function is called a **recursive function**.

Since computer programming is a fundamental application of mathematics, so let us first try to understand the mathematical reasoning behind recursion.

In general, we all are aware of the concept of functions. In a nutshell, functions are mathematical equations that produce an output on providing input. **For example:** Suppose the function **F(x)** is a function defined by:

$$F(x) = x^2 + 4$$

We can write the **Java Code** for this function as:

```
public static int F(int x){  
    return (x * x + 4);  
}
```

Now, we can pass different values of x to this function and receive our output accordingly.

Before moving onto the recursion, let's try to understand another mathematical concept known as the **Principle of Mathematical Induction (PMI)**.

Principle of Mathematical Induction (PMI) is a technique for proving a statement, a formula, or a theorem that is asserted about a set of natural numbers. It has the following three steps:

1. **Step of the trivial case:** In this step, we will prove the desired statement for a base case like $n = 0$ or $n = 1$.
2. **Step of assumption:** In this step, we will assume that the desired statement is valid for $n = k$.
3. **To prove step:** From the results of the assumption step, we will prove that, $n = k + 1$ is also true for the desired equation whenever $n = k$ is true.

For Example: Let's prove using the **Principle of Mathematical Induction** that:

$$S(N): 1 + 2 + 3 + \dots + N = (N * (N + 1))/2$$

(The sum of first N natural numbers)

Proof:

Step 1: For $N = 1$, $S(1) = 1$ is true.

Step 2: Assume, the given statement is true for $N = k$, i.e.,

$$1 + 2 + 3 + \dots + k = (k * (k + 1))/2$$

Step 3: Let's prove the statement for $N = k + 1$ using step 2.

To Prove: $1 + 2 + 3 + \dots + (k+1) = ((k+1)*(k+2))/2$

Proof:

Adding $(k+1)$ to both LHS and RHS in the result obtained on step 2:

$$1 + 2 + 3 + \dots + (k+1) = (k*(k+1))/2 + (k+1)$$

Now, taking $(k+1)$ common from RHS side:

$$1 + 2 + 3 + \dots + (k+1) = (k+1)*((k + 2)/2)$$

According to the statement that we are trying to prove:

$$1 + 2 + 3 + \dots + (k+1) = ((k+1)*(k+2))/2$$

Hence proved.

One can think, why are we discussing these over here. To answer this question, we need to know that these three steps of PMI are related to the three steps of recursion, which are as follows:

1. **Induction Step and Induction Hypothesis:** Here, the Induction Step is the main problem which we are trying to solve using recursion, whereas the Induction Hypothesis is the sub-problem, using which we'll solve the induction step. Let's define the Induction Step and Induction Hypothesis for our running example:

Induction Step: Sum of first n natural numbers - F(n)

Induction Hypothesis: This gives us the sum of the first n-1 natural numbers - F(n-1)

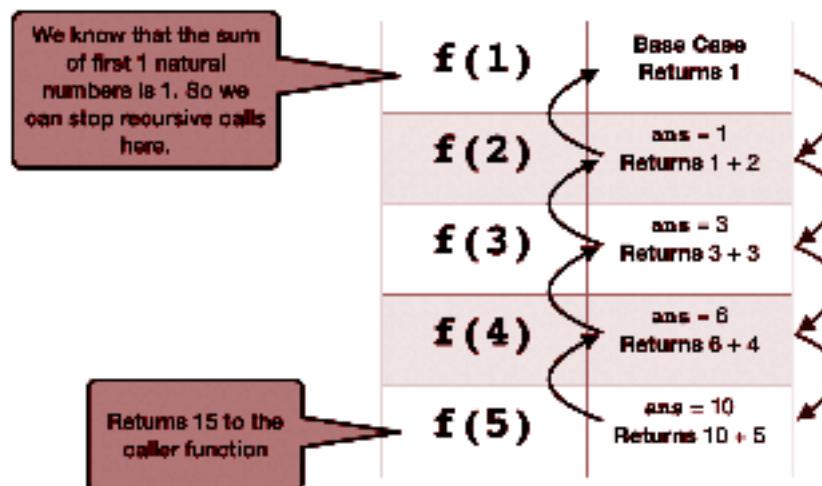
2. Express F(n) in terms of F(n-1) and write code:

$$F(N) = F(N-1) + N$$

Thus, we can write the Java code as:

```
public static int f(int N){
    int ans = f(N-1); //Induction Hypothesis step
    return ans + N; //Solving problem from result in previous step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop.



4. After the dry run, we can conclude that for N equals 1, the answer is 1, which we already know. So we'll use this as our base case. Hence the final code becomes:

```
public static int f(int N){  
    if(N == 1)    // Base Case  
        return 1;  
    int ans = f(N-1);  
    return ans + N;  
}
```

This is the main idea to solve recursive problems. To summarize, we will always focus on finding the solution to our starting problem and tell the function to compute the rest for us using the particular hypothesis. This idea will be studied in detail in further sections with more examples.

Now, we'll learn more about recursion by solving problems which contain smaller subproblems of the same kind. Recursion in computer science is a method where the solution to the question depends on solutions to smaller instances of the same problem. By the exact nature, it means that the approach that we use to solve the original problem can be used to solve smaller problems as well. So, in other words, in recursion, a function calls itself to solve smaller problems. **Recursion** is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts, and the code is also shorter and easier to understand.

Working of recursion

We can define the steps of the recursive approach by summarizing the above three steps:

- **Base case:** A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the **recursion depth*** will be exceeded and it will throw an error.
- **Recursive call:** The recursive function will invoke itself on a smaller version of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is.
- **Small calculation:** Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

Note*: Recursion uses an in-built stack which stores recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow. If the number of recursion calls exceeded the maximum permissible amount, the **recursion depth*** will be exceeded.

Now, let us see how to solve a few common problems using Recursion.

Problem Statement - Find Factorial of a Number

We want to find out the factorial of a natural number.

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

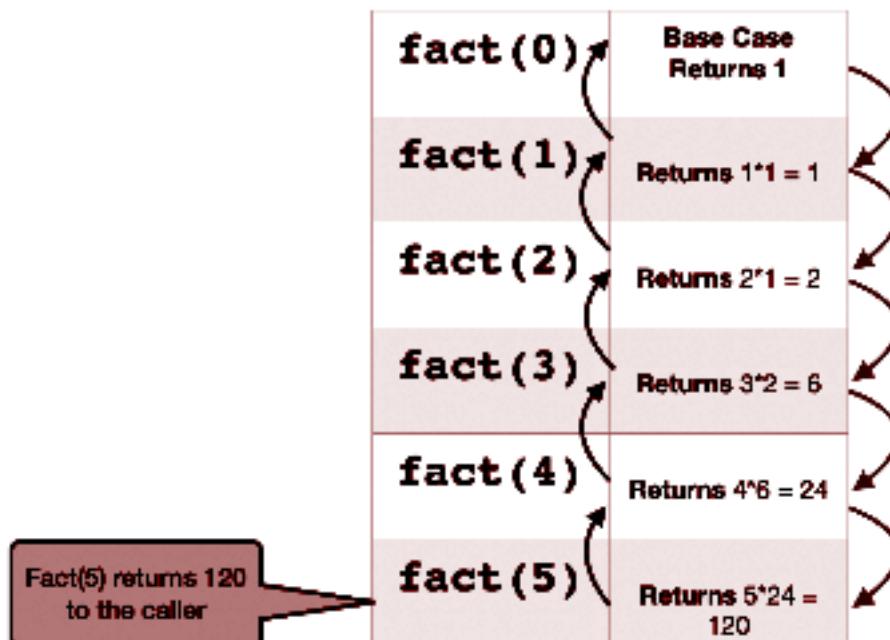
1. **Induction Step:** Calculating the factorial of a number n - $F(n)$

Induction Hypothesis: We have already obtained the factorial of n-1 - $F(n-1)$

- Expressing $F(n)$ in terms of $F(n-1)$: $F(n) = n * F(n-1)$. Thus we get:

```
public static int fact(int n){
    int ans = fact(n-1); #Assumption step
    return ans * n; #Solving problem from assumption step
}
```

- The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop. Consider $n = 5$:



As we can see above, we already know the answer of $n = 0$, which is 1. So we will keep this as our base case. Hence, the code now becomes:

```
public static int factorial(int n){
    if (n == 0) // base case
        return 1;
    else
        return n * factorial(n-1); // recursive case
}
```

Problem Statement - Fibonacci Number

Write a function `int fib(int n)` that returns nth fibonacci number. For example, if $n = 0$, then $fib(int n)$ should return 0. If $n = 1$, then it should return 1. For $n > 1$, it should return $F(n-1) + F(n-2)$, i.e., fibonacci of $n-1$ + fibonacci of $n-2$.

Function for Fibonacci series:

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0 \text{ and } F(1) = 1$$

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

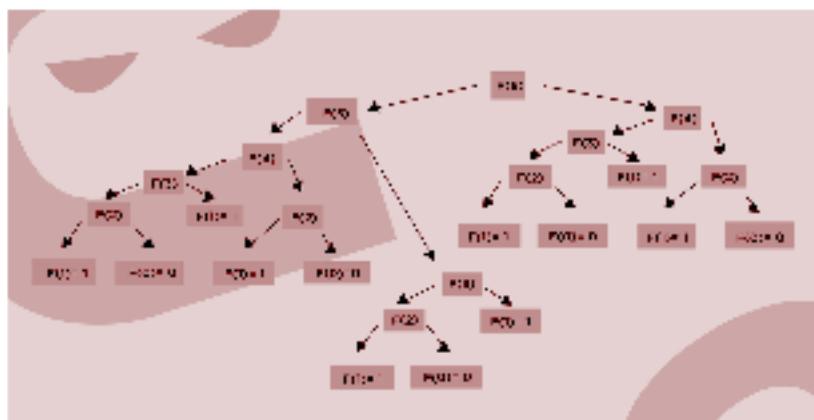
1. **Induction Step:** Calculating the n^{th} Fibonacci number n .

Induction Hypothesis: We have already obtained the $(n-1)^{\text{th}}$ and $(n-2)^{\text{th}}$ Fibonacci numbers.

2. Expressing $F(n)$ in terms of $F(n-1)$ and $F(n-2)$: $F_n = F_{n-1} + F_{n-2}$.

```
public static int f(int n){
    int ans = f(n-1) + f(n-2); //Assumption step
    return ans; //Solving problem from assumption step
}
```

3. Let's dry run the code for achieving the base case: (Consider $n= 6$)



From here we can see that every recursive call either ends at 0 or 1 for which we already know the answer: **F(0) = 0 and F(1) = 1**. Hence using this as our base case in the code below:

```
public static int fib(int n){  
    if (n <= 1)  
        return n;  
    else  
        return (fib(n-1) + fib(n-2));  
}
```

Recursion and array

Let us take an example to understand recursion on arrays.

Problem Statement - Check If Array Is Sorted.

We have to tell whether the given array is sorted or not using recursion.

For example:

- If the array is {2, 4, 8, 9, 9, 15}, then the output should be **YES**.
- If the array is {5, 8, 2, 9, 3}, then the output should be **NO**.

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Induction hypothesis or Assumption step:** We assume that we have already obtained the answer to the array starting from index 1. In other words, we assume that we know whether the array (starting from the first index) is sorted.
2. **Solving the problem from the results of the “Assumption step”:** Before going to the assumption step, we must check the relation between the first

two elements. Find if the first two elements are sorted or not. If the elements are not in sorted order, then we can directly return false. If the first two elements are in sorted order, then we will check for the remaining array through recursion.

```
public static int isSorted(int[][] a, int size){
    if (a[0] > a[1]) // Small Calculation
        return false
    int isSmallerSorted = isSorted(a+1, size-1); //Assumption step
    return isSmallerSorted;
}
```

3. We can see that in the case when there is only a single element left or no element left in our array, the array is always sorted. Let's check the final code now:

```
public static int isSorted(int[][] a, int size){
    if (size == 0 or size == 1) // Base case
        return true;

    if (a[0] > a[1]) // Small calculation
        return false;

    int isSmallerSorted = isSorted(a+1, size-1); //Recursive call
    return isSmallerSorted;
}

// driver code
public static void main(String[] args) {
    int arr[] = {2, 3, 6, 10, 11};
    if(isSorted(arr, 5))
        System.out.println("Yes");
    else
        System.out.println("No");
}
```

Problem Statement - First Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The first index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes first in the array.

To get a better understanding of the problem statement, consider the given cases:

Case 1: Array = {1,4,5,7,2}, Integer = 4

Output: 1

Explanation: 4 is present at 1st position in the array.

Case 2: Array = {1,3,5,7,2}, Integer = 4

Output: -1

Explanation: 4 is not present in the array

Case 3: Array = {1,3,4,4,4}, Integer = 4

Output: 2

Explanation: 4 is present at 3 positions in the array; i.e., [2, 3, 4]. But as the question says, we have to find out the first occurrence of the target value, so the answer should be 2.

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Small calculation part:

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the startIndex which we will increment in each recursive call.

```
if(arr[sI] == x)
    return sI;
```

Recursive Call step:

- Since, in the running example, the startIndex element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5], x=6. Though we will pass the same array but startIndex will be incremented.
- The recursive call will look like this:

```
f(arr, sI+1, x);
```

- In the recursive call, we are incrementing the startIndex pointer.
- We have to assume that the answer will come from the recursive call. The answer will come in the form of an integer.
- If the answer is -1, this denotes that the element is not present in the remaining array.
- If the answer is any other integer (other than -1), then this denotes that the element is present in the remaining array.

Base case step:

- The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array.
- **For example:** Consider the array [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the startIndex exceeds size of the array.
- Therefore , then we will return -1. This is because if the base case is reached, then this means that the element is not present in the entire array.
- We can write the base case as:

```
if(sI == arr.length) // Base Case
    return -1;
```

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement - Last Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The last index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes last in the array.

Case 1: Array = {1,4,5,7,2}, Integer = 4

Output: 1 (**Explanation:** 4 is present at 1st position in the array, which is the last and the only place where 4 is present in the given array.)

Case 2: Array = {1,3,5,7,2}, Integer = 4

Output: -1 (**Explanation:** 4 is not present in the array.)

Case 3: Array = {1,3,4,4,4}, Integer = 4

Output: 4 (**Explanation:** 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the last occurrence of the target value, so the answer should be 4.)

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution.

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[sI] == x)
    return sI;
```

Since, in the running example, the 0th index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**. We will start with startIndex from the last index of the array.

The recursive call will look like this:

```
f(arr, sI-1, x);
```

- In the recursive call, we are decrementing the startIndex pointer..
- We have to assume that the answer will come for a recursive call. The answer will come in the form of an integer.

Base Case:

- The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array.
- **For example:** [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the startIndex passes beyond left of index 0 as we started from the rightmost index.

- When startIndex becomes -1, then we will return -1.
- This is because if the startIndex reaches -1, then this means that we have traversed the entire array and we were not able to find the target element.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

All Indices of A Number

Here, given an array of length N and an integer x, you need to find all the indexes where x is present in the input array. Save all the indexes in a new array (in increasing order) and return that array.

Case 1: Array = {1, 4, 5, 7, 2}, Integer = 4

Output: [1], the size of the array will be 1 (as 4 is present at 1st position in the array, which is the only position where 4 is present in the given array).

Case 2: Array = {1, 3, 5, 7, 2}, Integer = 4

Output: [], the size of the array will be 0 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, Integer = 4

Output: [2, 3, 4], the size of the array will be 3 (as 4 is present at three positions in the array; i.e., [2, 3, 4]).

Now, let's think about solving this problem...

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let us assume the given array is: **[5, 6, 5, 5, 6]** and the target element is **5**, then the output array should be **[0, 2, 3]** and for the same array, let's suppose the target element is **6**, then the output array should be **[1, 4]**.

To solve this question, the base case should be the case when the startIndex reaches the size of the array. In this case, we should simply return an empty array, i.e., an array of size 0, since there are no elements left.

The next two components of the solution are Recursive call and Small calculation. Let us try to figure them out using the following images:

So, the following are the recursive call and small calculation components of the solution:

Recursive Call

```
int[][][] output = fun(arr, startIndex+1, size, x);
```

Small Calculation:

1. If the element at startIndex of array is equal to the x, then create a new array of size of output+1. Now copy paste all the elements of the output array to the new array starting from 1st index and at the 0th index add the element of startIndex in original array. Finally return this new output.
2. Else is the case when element at startIndex do not match with x. So simply return the output.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Using the same concept, other problems can be solved using recursion, just remember to apply PMI and three steps of recursion intelligently.

Recursion 2

In this module, we are going to understand how to solve different kinds of problems using recursion.

Recursion With Strings

Recursion in strings is not a very different logic, it is the same as we apply in arrays, in fact it becomes more easy to pass a complete new string using substring method.

Problem Statement - Replace pi

You are given a string of size n containing characters a-z. The task is to write a recursive function to replace all occurrences of pi with 3.14 in the given string and print the modified string.

Approach

- Step of the trivial case:** In this step, we will prove the desired statement for a base case like **size of string = 0** or **1**.
- Small calculation and recursive part interlinked:** In this step, you will first check character at 0th index and character at 1st index of the string.
 - If it comes out to be 'p' and 'i' then we make a recursive call passing the string from index 2. And thereafter "3.14" needs to be concatenated with the recursive answer and return this new result.

- Else we will just make a recursive call passing the string from index 1. Thereafter the character at 0th index needs to be concatenated with a recursive answer and return the same.

Binary Search Using Recursion

In a nutshell, this search algorithm takes advantage of a collection of elements that are already sorted by ignoring half of the elements after just one comparison.

You are given a target element X and a sorted array. You need to check if X is present in the array. Consider the algorithm given below:

- Compare X with the middle element in the array.
- **If** X is the same as the middle element, we return the index of the middle element.
- **Else if** X is greater than the mid element, then X can only lie in the right (greater) half subarray after the mid element. Thus, we apply the algorithm, recursively, for the right half. *#Condition1*
- **Else if** X is smaller, the target X must lie in the left (lower) half. So we apply the algorithm, recursively, for the left half. *#Condition2*

```
// Returns the index of x in arr if present, else -1
public static int binary_search(int[] arr, int low, int high, int x){
    if (high >= low){ // Check base case
        int mid = (high + low) / 2;
        if (arr[mid] == x) //If element is at the middle itself
            return mid;
        else if (arr[mid] > x) //Condition 2
            return binary_search(arr, low, mid - 1, x);
        else //Condition 1
            return binary_search(arr, mid + 1, high, x);
    }
    else
        return -1; // Element is not present in the array
}
```

}

Sorting Techniques Using Recursion - Merge Sort

Merge sort requires dividing a given list into equal halves until it can no longer be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

- **Step 1** – If it is only one element in the list it is already sorted, return.
- **Step 2** – Divide the list recursively into two halves until it can't be divided further.
- **Step 3** – Merge the smaller lists into a new list in sorted order.

It has just one disadvantage and that is it's **not an in-place sorting** technique i.e. it creates a copy of the array and then works on that copy.

Pseudo-Code

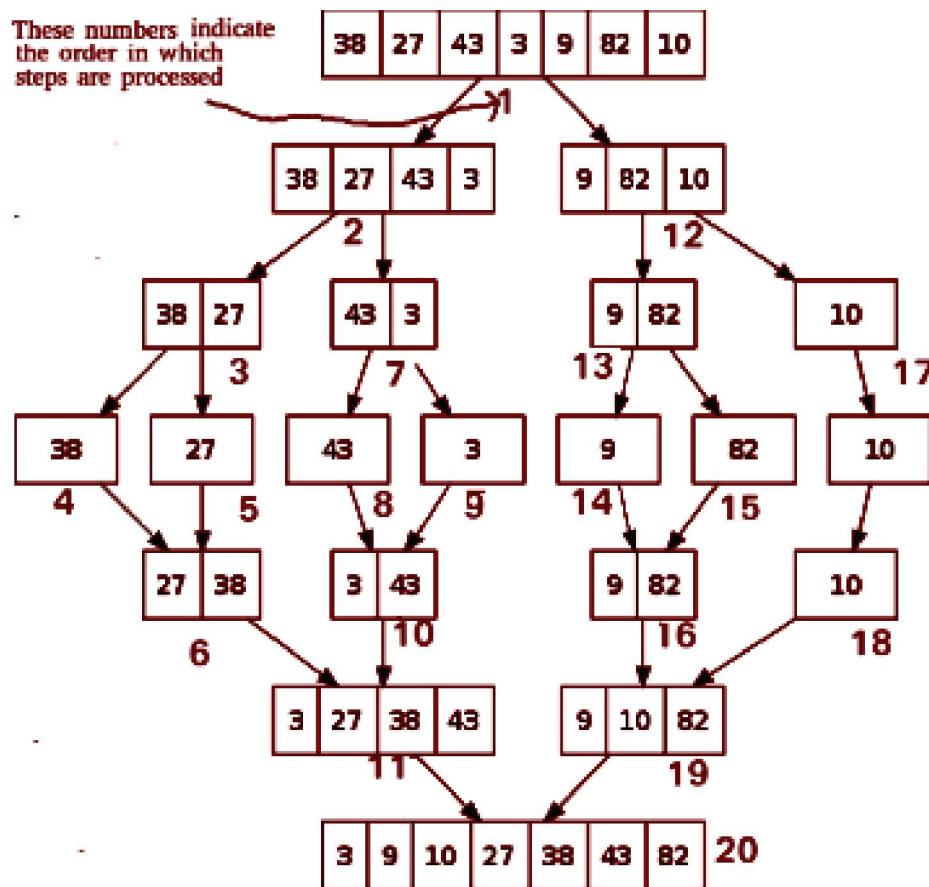
```

public static void mergeSort(int arr[], int l, int r){
    if (r > l){
        1. Find the middle point to divide the array into two halves:
            middle m = (l+r)/2
        2. Call mergeSort for the first half:
            Call mergeSort(arr, l, m)
        3. Call mergeSort for the second half:
            Call mergeSort(arr, m+1, r)
        4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m, r)
    }
}

```

The following diagram shows the complete merge sort process for an example array [38, 27, 43, 3, 9, 82, 10]. If we take a closer look at the diagram, we can see

that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Quick Sort

Quick-sort is based on the **divide-and-conquer approach**. It works along the lines of choosing one element as a pivot element and partitioning the array around it such that:

- The left side of the pivot contains all the elements that are less than the pivot element

- The right side contains all elements greater than the pivot.

Algorithm for Quick Sort:

Based on the **Divide-and-Conquer** approach, the quicksort algorithm can be explained as:

- **Divide:** The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.
- **Conquer:** The left and right sub-parts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
- **Combine:** This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

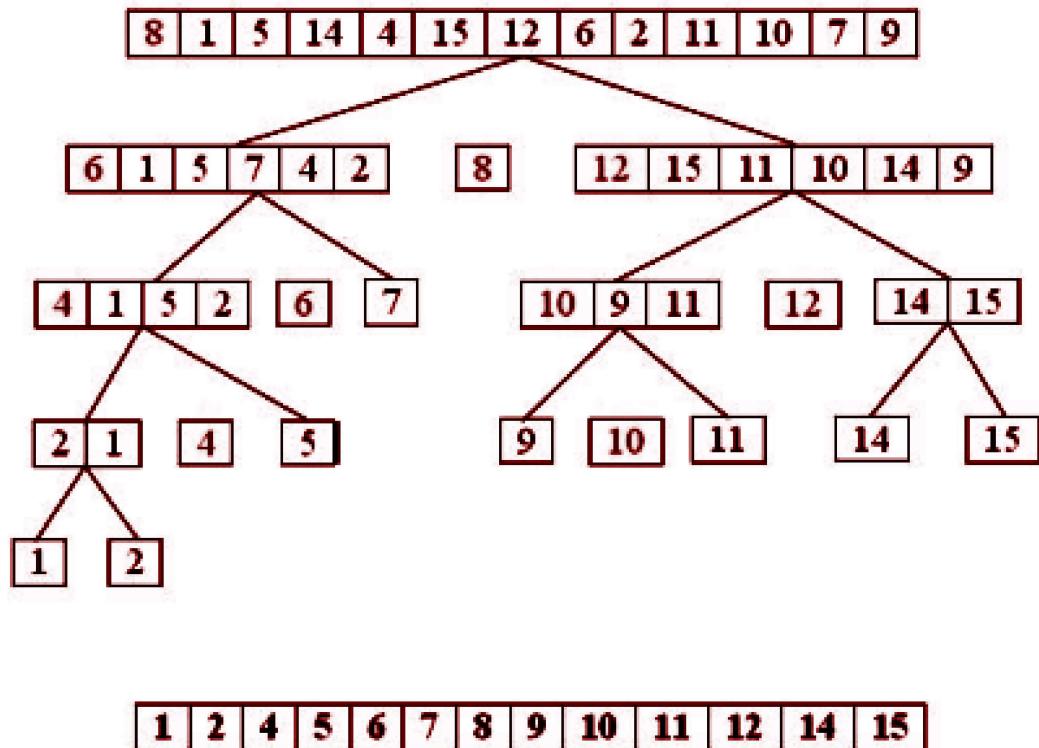
The advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, that it is an in-place sorting technique.

There are many ways to pick a pivot element:

1. Always pick the first element as the pivot.
2. Always pick the last element as the pivot.
3. Pick a random element as the pivot.
4. Pick the middle element as the pivot.

Given below is a pictorial representation of how this algorithm sorts the given array:

```
[8, 1, 5, 14, 4, 15, 12, 6, 2, 11, 10, 7, 9]
```



- In **step 1**, 8 is taken as the pivot.
- In **step 2**, 6 and 12 are taken as pivots.
- In **step 3**, 4, and 10 are taken as pivots.
- We keep dividing the list about pivots till there are only 2 elements left in a sublist.

Problem Statement - Tower Of Hanoi

Tower of Hanoi is a **mathematical puzzle** where we have **3** rods and **N** disks. The objective of the puzzle is to move all disks from **source rod** to **destination rod** using a **third rod (say auxiliary)**. The rules are :

- Only one disk can be moved at a time.
- A disk can be moved only if it is on the top of a rod.
- No disk can be placed on the top of a smaller disk.

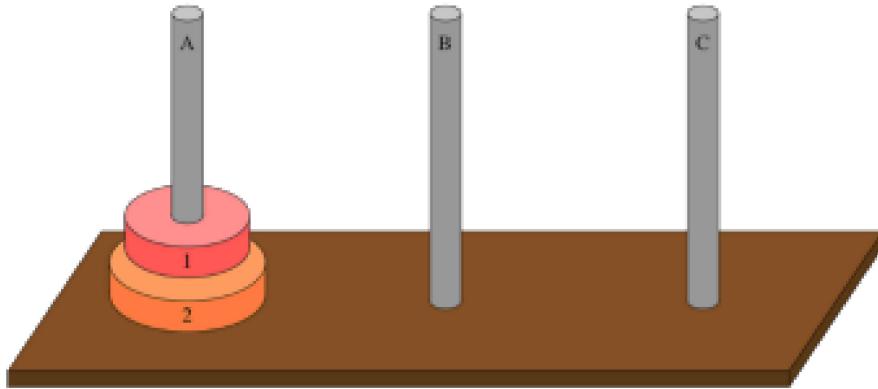
Print the steps required to move **N** disks from source rod to destination rod.

Source Rod is named as '**A**', the destination rod as '**B**', and the auxiliary rod as '**C**'.

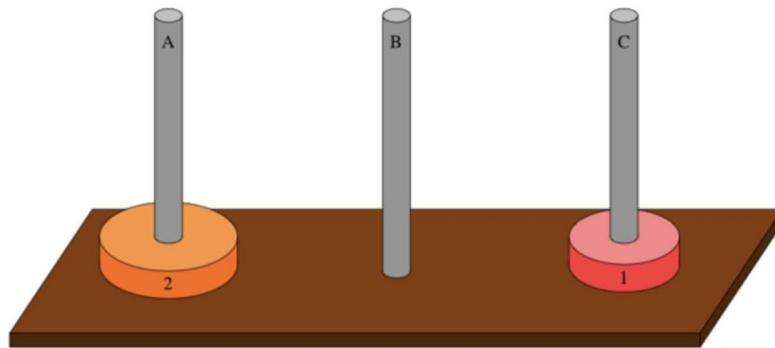
Let's see how to solve the problem recursively. We'll start with a really easy case **N=1**. We just need to move one disk from source to destination.

- You can always move **disk 1** from peg **A** to peg **B** because you know that any disks below it must be larger.
- There's nothing special about pegs **A** and **B**. You can move disk 1 from peg **B** to peg **C** if you like, or from peg **C** to peg **A**, or from any peg to any peg.
- Solving the Towers of Hanoi problem with one disk is trivial as it requires moving only the one disk one time.

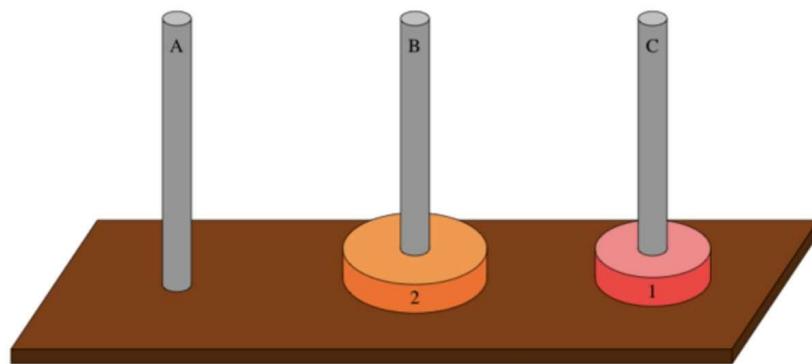
Now consider the case **N=2**. Here's what it looks like at the start:



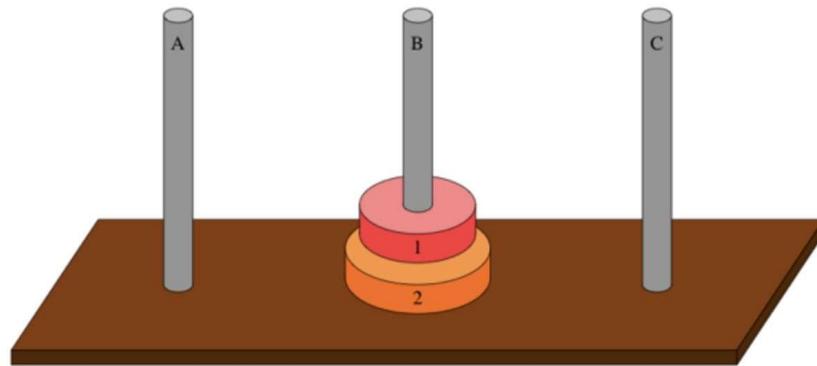
First, move **disk 1** from peg **A** to peg **C**:



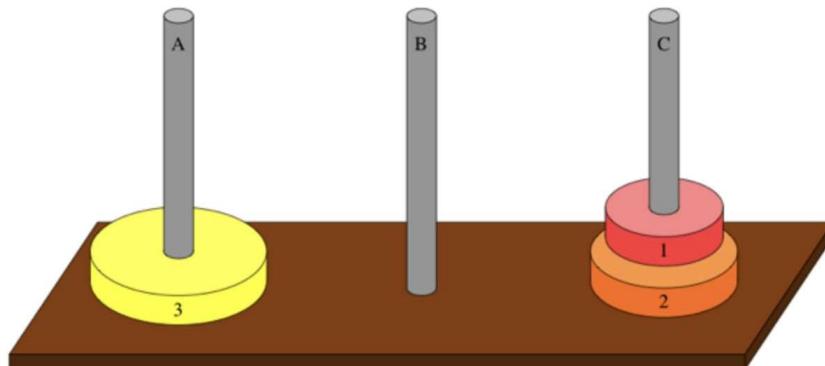
Notice that we're using peg **C** as a spare peg, a place to put **disk 1** so that we can get at **disk 2**. Now that **disk 2**—the bottommost disk—is exposed, move it to peg **B**:



Finally, move **disk 1** from peg **C** to peg **B**:

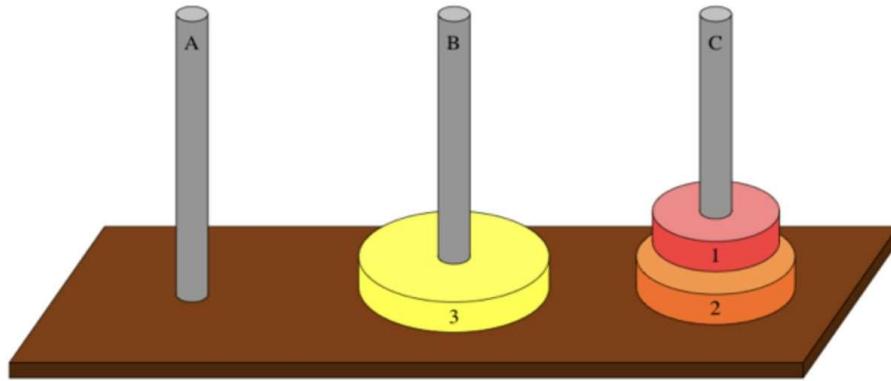


Now let us solve this problem for **3** disks. You need to expose the bottom disk (**disk 3**) so that you could move it from peg **A** to peg **B**. To expose **disk 3**, you needed to move disks 1 and 2 from peg **A** to the spare peg, which is peg **C**:



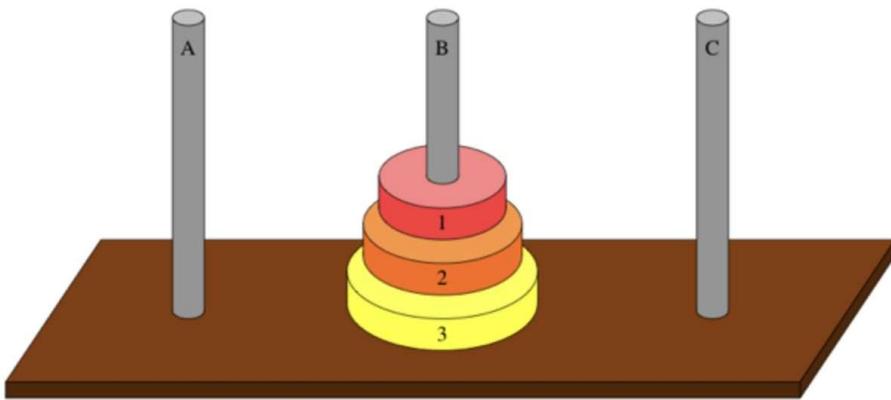
Wait a minute—it looks like two disks moved in one step, violating the first rule. But they did not move in one step. You agreed that you can move disks 1 and 2 from any peg to any peg, **using three steps**. The situation above represents what you have after three steps. (Move **disk 1** from peg **A** to peg **B**; move **disk 2** from peg **A** to peg **C**; move **disk 1** from peg **B** to peg **C**.)

More to the point, by moving disks 1 and 2 from peg **A** to peg **C**, you have recursively solved a subproblem: move disk **1 through n-1** (remember that $n = 3$)



from peg **A** to peg **C**. Once you've solved this subproblem, you can move **disk 3** from peg **A** to peg **B**:

Now, to finish up, you need to recursively solve the subproblem of moving disks **1 through n-1**, from peg **C** to peg **B**. Again, you agreed that you can do so in three steps. (Move **disk 1** from peg **C** to peg **A**; move **disk 2** from peg **C** to peg **B**; move **disk 1** from peg **A** to peg **B**.) And you're done:



At this point, you might have picked up the pattern. The **algorithm** can be summarised as:

If **n == 1**, just move **disk 1**. Otherwise, when $n \geq 2$, solve the problem in three steps:

- Recursively solve the subproblem of moving disks **1 through n-1** from whichever peg they start on, to the spare peg.
- Move disk **N** from the peg it starts on, to the peg it's supposed to end up on.
- Recursively solve the subproblem of moving disks **1 through n-1**, from the spare peg to the peg they're supposed to end up on.

Practice Problems

- Return all subsequences of a given string
- Print all subsequences of a given string
- Return all keypad combinations
- Print all keypad combinations

Refer to the course lecture for detailed explanation of these problems.

Time Complexity

What you will learn in this lecture?

- Algorithm Analysis
- Type of Analysis
- Big O Notation
- Determining Time Complexities Theoretically
- Time complexity of some common algorithms

Introduction

An important question while programming is: How efficient is an algorithm or piece of code?

Efficiency covers lots of resources, including:

1. CPU (time) usage
2. memory usage
3. disk usage
4. network usage

All are important but we are mostly concerned about CPU time. Be careful to differentiate between:

1. **Performance:** how much time/memory/disk/etc. is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
2. **Complexity:** how do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger. Complexity affects performance but not vice-versa. The time required by a function/method is proportional to the number of "basic operations" that it performs.

Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Analysis

To analyze a given algorithm, we need to know, with which inputs the algorithm takes less time (i.e. the algorithm performs well) and with which inputs the algorithm takes a long time.

Three types of analysis are generally performed:

- **Worst-Case Analysis:** The worst-case consists of the input for which the algorithm takes the longest time to complete its execution.
- **Best Case Analysis:** The best case consists of the input for which the algorithm takes the least time to complete its execution.
- **Average case:** The average case gives an idea about the average running time of the given algorithm.

There are two main complexity measures of the efficiency of an algorithm:

- **Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Big-O notation

We can express algorithmic complexity using the **big-O** notation. For a problem of size N:

- A constant-time function/method is "order 1": **O(1)**
- A linear-time function/method is "order N": **O(N)**
- A quadratic-time function/method is "order N squared": **O(N²)**

Definition: Let g and f be functions from the set of natural numbers to itself. The function f is said to be **$O(g)$** (read big-oh of g), if there is a constant c and a natural n_0 such that $f(n) \leq cg(n)$ for all $n > n_0$.

Note: $O(g)$ is a set!

Abuse of notation: $f = O(g)$ does not mean $f \in O(g)$.

Examples:

- $5n^2 + 15 = O(n^2)$, since $5n^2 + 15 \leq 6n^2$, for all $n > 4$.
- $5n^2 + 15 = O(n^3)$, since $5n^2 + 15 \leq n^3$, for all $n > 6$.
- **$O(1)$** denotes a constant.

Although we can include constants within the big-O notation, there is no reason to do that. Thus, we can write **$O(5n + 4) = O(n)$** .

Note: The **big-O** expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time function/method, which will be faster than a quadratic-time function/method).

Determining Time Complexities Theoretically

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;  
statement 2;  
...  
statement k;
```

The total time is found by adding the times for all statements:

```
totalTime = time(statement1) + time(statement2) +...+  
time(statementk)
```

2. if-else statements

```
if (condition):  
    #sequence of statements 1  
else:  
    #sequence of statements 2
```

Here, either **sequence 1** will execute, or **sequence 2** will execute. Therefore, the worst-case time is the slowest of the two possibilities:

```
max(time(sequence 1), time(sequence 2))
```

For example, if sequence 1 is $O(N)$ and sequence 2 is $O(1)$ the worst-case time for the whole if-then-else statement would be **$O(N)$** .

3. for loops

```
for i in range N:  
    #sequence of  
statements
```

Here, the loop executes **N** times, so the sequence of statements also executes **N** times. Now, assume that all the statements are of the order of **$O(1)$** , then the total time for the **for** loop is **$N * O(1)$** , which is **$O(N)$** overall.

4. Nested loops

```

for i in range N:
    for i in range
M:
    #statements

```

The outer loop executes **N** times. Every time the outer loop executes, the inner loop executes **M** times. As a result, the statements in the inner loop execute a total of **N * M** times. Assuming the complexity of the statement inside the inner loop to be **O(1)**, the overall complexity will be **O(N * M)**.

Sample Problem:

What will be the Time Complexity of following while loop in terms of 'N' ?

```

while N>0:
    N =
    N//8

```

We can write the iterations as:

Iteration Number	Value of N
1	N
2	N//8
3	N//64
...	...
k	N//8 ^k

We know, that in the last i.e. the k^{th} iteration, the value of N would become 1 , thus, we can write:

```

N//8k = 1
=> N = 8k
=> log(N) = log(8k)
=> k*log(8) =
log(N)
=> k =
log(N)/log(8)
=> k = log8(N)
    
```

Now, clearly the number of iterations in this example is coming out to be of the order of $\log_8(N)$. Thus, the time complexity of the above while loop will be $O(\log_8(N))$.

Qualitatively, we can say that after every iteration, we divide the given number by 8, and we go on dividing like that, till the number remains greater than 0. This gives the number of iterations as $O(\log_8(N))$.

Time Complexity Analysis of Some Common Algorithms

Linear Search

Linear Search time complexity analysis is done below-

Best case- In the best possible case:

- The element being searched will be found in the first position.

- In this case, the search terminates in success with just one comparison.
- Thus in the best case, the linear search algorithm takes **O(1)** operations.

Worst Case- In the worst possible case:

- The element being searched may be present in the last position or may not present in the array at all.
- In the former case, the search terminates in success with **N** comparisons.
- In the latter case, the search terminates in failure with **N** comparisons.
- Thus in the worst case, the linear search algorithm takes **O(N)** operations.

Binary Search

Binary Search time complexity analysis is done below-

- In each iteration or each recursive call, the search gets reduced to half of the array.
- So for **N** elements in the array, there are **log₂N** iterations or recursive calls.

Thus, we have-

- Time Complexity of the Binary Search Algorithm is **O(log₂N)**.
- Here, **N** is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Big-O Notation Practice Examples

Example-1 Find upper bound for **f(n) = 3n + 8**

Solution: $3n + 8 \leq 4n$, for all $n \geq 8$

$\therefore 3n + 8 = O(n)$ with $c = 4$ and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$

$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Space Complexity Analysis

Introduction

- The space complexity of an algorithm represents the amount of extra memory space needed by the algorithm in its life cycle.
- Space needed by an algorithm is equal to the sum of the following two components:
 - A fixed part is a space required to store certain data and variables (i.e. simple variables and constants, program size, etc.), that are not dependent on the size of the problem.
 - A variable part is a space required by variables, whose size is dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation, etc.
- Space complexity $S(p)$ of any algorithm p is $S(p) = A + Sp(I)$ Where A is treated as the fixed part and $S(I)$ is treated as the variable part of the algorithm which depends on instance characteristic I .

Note: It's necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.

To get warmed up, let's consider a simple operation that sums two integers (numbers without a fractional part):

```
public static int difference(int a, int b){  
    return a + b;  
}
```

In this particular method, three variables are used and allocated in memory:

The first integer argument, a; the second integer argument, b; and the returned sum which is also an integer.

In Java, these three variables point to three different memory locations. We can see that the space complexity is constant, so it can be expressed in big-O notation as **O(1)**.

Next, let's determine the space complexity of a program that sums all integer elements in an array:

```
public static int sumArray(int[] array){  
    int sum = 0;  
    for(int i=0; i<array.length; i++)  
        sum += array[i];  
    return sum;  
}
```

Again, let's list all variables present in the above code:

- **array**
- **size**
- **sum**
- **iterator**

The space complexity of this code snippet is **O(n)**, which comes from the reference to the array that was passed to the function as an argument.

Let us now analyze the space complexity for a few common sorting algorithms. This will give you deeper insight into complexity analysis.

Quick-Sort Space Complexity Analysis

Let us consider the various scenarios possible :

Best case scenario: The best-case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either equal or have a size difference of 1 of each other.

- **Case 1:** The case when the sizes of the sublist on either side of the pivot become equal occurs when the subarray has an odd number of elements and the pivot is right in the middle after partitioning. Each partition will have $(n-1)/2$ elements.
- **Case 2:** The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even number, n , of elements. One partition will have $n/2$ elements with the other having $(n/2)-1$.
- In either of these cases, each partition will have at most $n/2$ elements, and the tree representation of the subproblem sizes will be as below:

Worst case scenario:

This happens when we encounter the most unbalanced partitions possible, then the original call takes place n times, the recursive call on $n-1$ elements will take place $(n-1)$ times, the recursive call on $(n-2)$ elements will take place $(n-2)$ times, and so on.

Based on the above-mentioned cases we can conclude that:

- The space complexity is calculated based on the space used in the recursion stack. The worst-case space used will be $O(n)$.
- The average case space used will be of the order $O(\log n)$.
- The worst-case space complexity becomes $O(n)$ when the algorithm encounters its worst-case when we need to make n recursive calls for getting a sorted list.

Practice Problems

Problem 1: What is the time & space complexity of the following code:

```
int a = 0
int b = 0
for(int i=0; i<n; i++){
    a = a + i;
}

for(int j=0; j<m; j++){
    b = b + j;
}
```

Problem 2: What is the time & space complexity of the following code:

```
int a = 0;
int b = 0;
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        a = a + j;
    }
}

for(int k=0; k<n; k++){
    b = b + k
}
```

Problem 3: What is the time and space complexity of the following code:

```
int a = 0;
for(int i=0; i<n; i++){
    int j = n;
    while (j>i){
        a = a + i + j;
        j = j-1;
    }
}
```

Object-Oriented Programming (OOPS-3)

What you will learn in this lecture?

- Important keywords and their use.
- Abstraction.
- Interfaces.

Final Keyword

- When a variable is declared with a final keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable.
- If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final list.
- Final keywords can be used to initialise constants.

Initializing a final variable:

```
final int {name_of_variable} = {value};
```

Example:

```
final int pi = 3.14;
```

Refer to the course videos to see the use case and more about the final keyword.

Abstract Classes

An abstract class can be considered as a blueprint for other classes. Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that has a declaration but does not have an implementation. This set of methods must be created within any child classes which inherit from the abstract class. *A class that contains one or more abstract methods is called an **abstract class**.*

Creating Abstract Classes in Java

- By default, Java does not provide abstract classes.
- A method becomes abstract when decorated with the keyword **abstract**.
- An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.
- However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.
- While declaring abstract methods in the class, it is not mandatory to use the **abstract** decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Java code uses the **ABC** class and defines an abstract base class:

```
abstract class ABC{  
    int value;  
    Abstract int do_something(){ //Our abstract method declaration  
        // TO_DO  
    }  
}
```

We will do it in the following example, in which we define two classes inheriting from our abstract class:

```
class add extends ABC{
    int do_something(){
        return value + 42;
    }
}

class mul extends ABC{
    int do_something(){
        return value * 42;
    }
}

class Test{
    public static void main(String[] args) {
        add x = new add(10);
        mul y = new mul(10);

        System.out.println(x.do_something());
        System.out.println(y.do_something());
    }
}
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

- An abstract method can have an implementation in the abstract class.

- However, even if they are implemented, this implementation shall be overridden in the subclasses.
- If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with `super()` call mechanism. (*Similar to cases of "normal" inheritance*).
- Similarly, we can even have concrete methods in the abstract class that can be invoked using `super()` call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.
- Consider the given example:

```

abstract class ABC{

    abstract int do_something(){ //Abstract Method
        System.out.println("Abstract Class AbstractMethod");
    }

    int do_something2(){ //Concrete Method
        System.out.println("Abstract Class ConcreteMethod");
    }
}

class AnotherSubclass extends ABC{
    int do_something(){
        //Invoking the Abstract method from super class
        super().do_something();
    }

    //No concrete method implementation in subclass
}

class Test{
    public static void main(String[] args) {
        AnotherSubclass x = new AnotherSubclass()
        x.do_something() //calling abstract method
        x.do_something2() //Calling concrete method
    }
}

```

```
    }  
}
```

We will get the output as:

```
Abstract Class AbstractMethod  
Abstract Class ConcreteMethod
```

Another Example

The given code shows another implementation of an abstract class.

```
// Java program showing how an abstract class works  
abstract class Animal{ //Abstract Class  
    abstract move();  
}  
  
class Human extends Animal{ //Subclass 1  
    void move(){  
        System.out.println("I can walk and run");  
    }  
}  
  
class Snake extends Animal{ //Subclass 2  
    void move(){  
        System.out.println("I can crawl")  
    }  
}  
  
class Dog extends Animal{ //SubClass 3  
    void move(){  
        System.out.println("I can bark")  
    }  
}  
  
// Driver code
```

```
class Test{  
    public static void main(String[] args) {  
        Animal R = new Human();  
        R.move();  
        Animal K = Snake();  
        K.move();  
        R = Dog();  
        R.move();  
    }  
}
```

We will get the output as:

```
I can walk and run  
I can crawl  
I can bark
```

Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

An interface is different from a class in several ways:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interface

```
public interface Name_of_interface {  
    // body  
}
```

Example:

```
public interface VehicleInterface {  
  
    public final static double PI = 3.14;  
  
    public int getMaxSpeed();  
    public void print();  
}
```

Now we need to implement this interface using a different class. A class uses the `implements` keyword to implement an interface. The **implements** keyword appears in the class declaration following the extends portion of the declaration.

```
public class Vehicle implements CarInterface{  
  
    @Override  
    public void print() {  
        // TODO Auto-generated method stub  
        // We can implement this function further.  
    }  
  
    @Override  
    public int getMaxSpeed() {
```

```
// TODO Auto-generated method stub
return 0;
}

@Override
public String getCompany() {
    // TODO Auto-generated method stub
    return null;
}
}
```

@Override annotation informs the compiler that the element is meant to **override** an element declared in an interface.

We can implement the given overridden functions and instantiate an object of Vehicle class.

Linked-List 1

Data Structures

Data structures are just a way to store and organize our data so that it can be used and retrieved as per our requirements. Using these can affect the efficiency of our algorithms to a greater extent. There are many data structures that we will be going through throughout the course, linked-lists are a part of them.

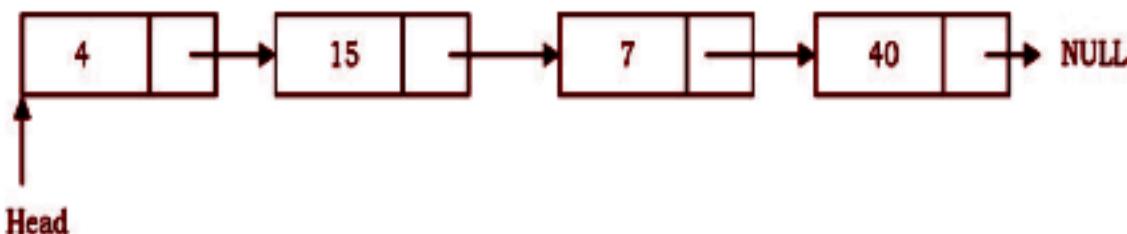
Introduction

A **Linked List** is a data structure used for storing collections of data. A linked list has the following properties:

- Successive elements are connected by pointers.
- Can grow or shrink in size during the execution of a program.
- Can be made just as long as required (until systems memory exhausts).
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as the list grows.

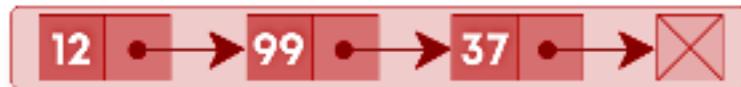
Basic Properties:

- Each element or node of a list is comprising of two items:
 - Data
 - Pointer(reference) to the next node.
- In a Linked List, the elements are not stored at contiguous memory locations.
- The first node of a linked list is known as **Head**.
- The last node of a linked list is known as **Tail**.
- The last node has a reference to null.



Types of A Linked List

- **Singly-Linked List:** Generally “linked list” means a singly linked list. Each node contains only one link which points to the subsequent node in the list.



- **Doubly-Linked List:** It's a two-way linked list as each node points not only to the next pointer but also to the previous pointer.



- **Circular-Linked List:** There is no tail node i.e., the next field is never **null** and the next field for the last node points to the head node.



- **Circular Doubly-Linked List:** Combination of both Doubly linked list and circular linked list.



Node Class (Singly Linked List)

```
// Node class
class Node{
    int data;
    Node next;
    // Function to initialize the node object
    Node(int data){
        this.data = data; // Data that the node contains
        this.next = null; // Next node that this node points to
    }
}
```

Note: The first node in the linked list is known as **Head** pointer and the last node is referenced as **Tail** pointer. We must never lose the address of the head pointer as it references the starting address of the linked list and if lost, would lead to loss of the list.

Traversing the Linked List

Let us assume that the head points to the first node of the list. To traverse the list we do the following:

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to **null**.

Printing the Linked List

To print the linked list, we will start traversing the list from the beginning of the list(head) until we reach the **null** pointer which will always be the tail pointer. Let us add a new function **printList()** to our **LinkedList** class.

```
// This function prints contents of Linked List starting from head
public static void printList(Node headNode){
    Node temp = headNode; //Start from the head of the List
    while (temp != null){ //Till we reach the last node
        System.out.print(temp.data + " ");
        temp = temp.next; //Update temp to point to the next Node
    }
}
```

Insertion of A Node in a Singly Linked List

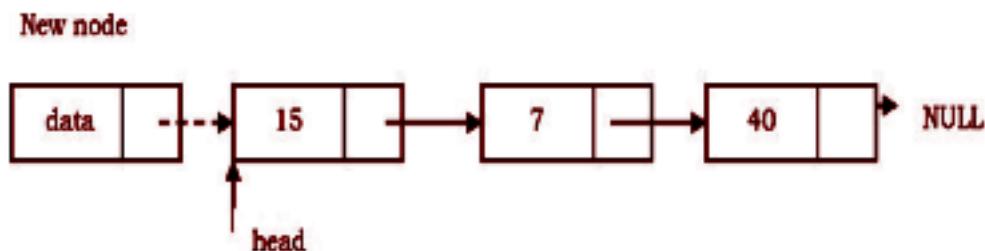
There are 3 possible cases:

- Inserting a new node before the head (at the beginning).
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node in the middle of the list (random location).

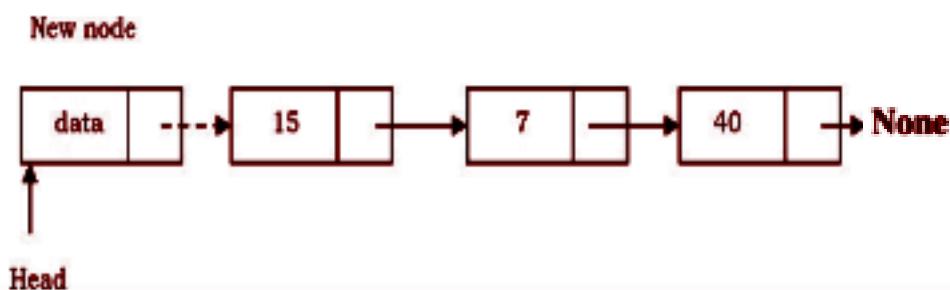
Case 1: Insert node at the beginning:

In this case, a new node is inserted before the current head node. Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps:

- Create a new node. Update the **next** pointer of the new node, to point to the current head.



- Update **head** pointer to point to the new node.



Java Code:

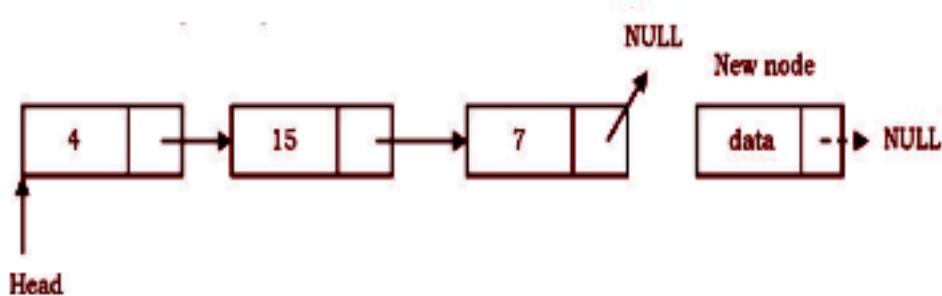
```

public static Node insertAtStart(Node head, int data){
    Node newNode = new Node(data); //Create a new node
    newNode.next = head; //Set next node of new node to current head
    head = newNode; //Update the head pointer to the new node
    return head;
}
  
```

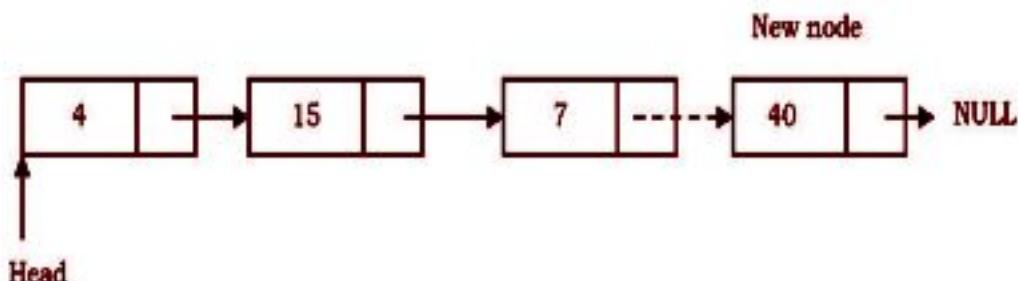
Case 2: Insert node at the ending:

In this case, we need to modify two next pointers (last nodes next pointer and new nodes next pointer).

- New node's **next** pointer points to **null**.



- Last node's **next** pointer points to the [new node](#).



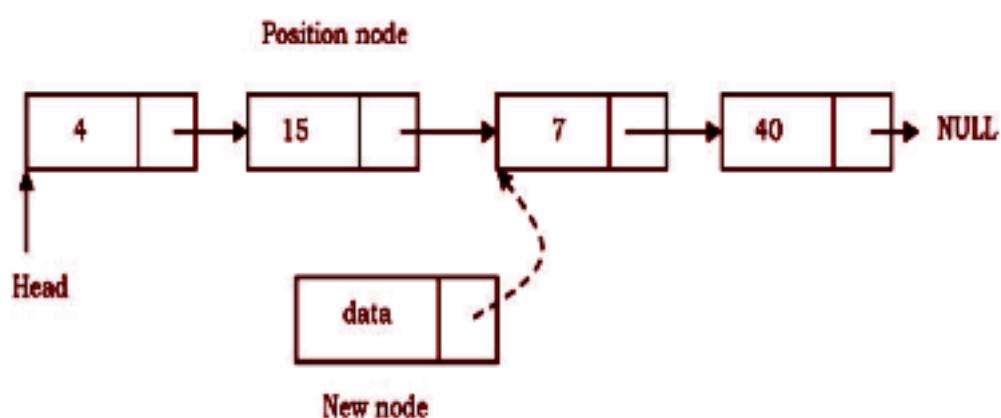
Java Code:

```
public static Node insertAtEnd(Node head, int data){
    Node newNode = new Node(data); //Create a new node
    if (head == null){ //Incase of empty LL
        head = newNode;
        return;
    }
    Node n = head;
    while (n.next != null) //If not empty traverse till last node
        n = n.next;
    n.next = newNode; //Set next = new node
    return head;
}
```

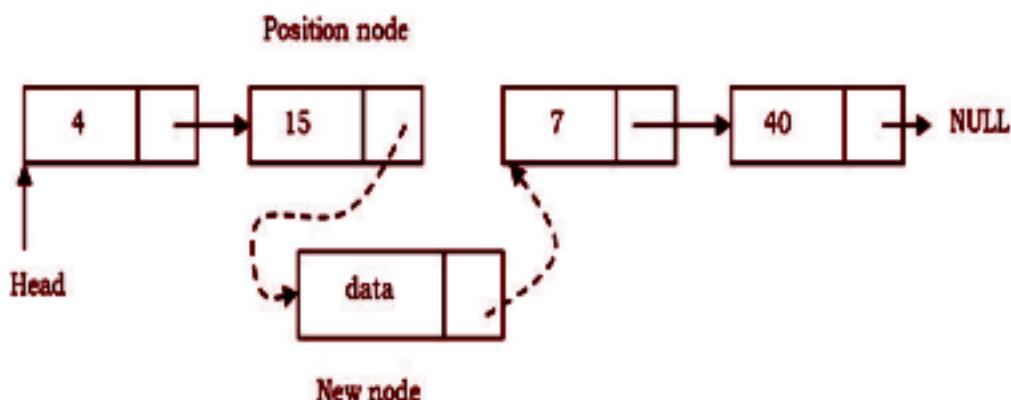
Case 3: Insert node anywhere in the middle: (At any specified Index)

Let us assume that we are given a position where we want to insert the new node. In this case, also, we need to modify two next pointers.

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node.
- For simplicity let us assume that the second node is called the **position node**. The new node points to the next node of the position where we want to add this node.



- Position node's **next** pointer now points to the new node.



Java Code:

```

public static Node insertAtIndex (int head, int index, int data){
    if (index == 1) // Insert at beginning
        insertAtStart(head, data);
    int i = 1;
    Node n = head;
    while (i < index-1 && n != null){
        n = n.next;
        i = i+1;
    }
    if (n == null)
        print("Index out of bound");
    else{
        Node newNode = new Node(data);
        newNode.next = n.next;
        n.next = newNode;
    }
}

```

Deletion of A Node in a Singly Linked List

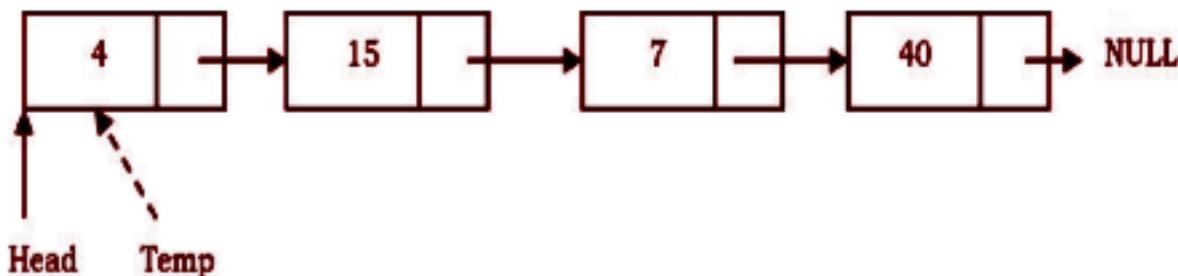
Similar to insertion, here we also have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

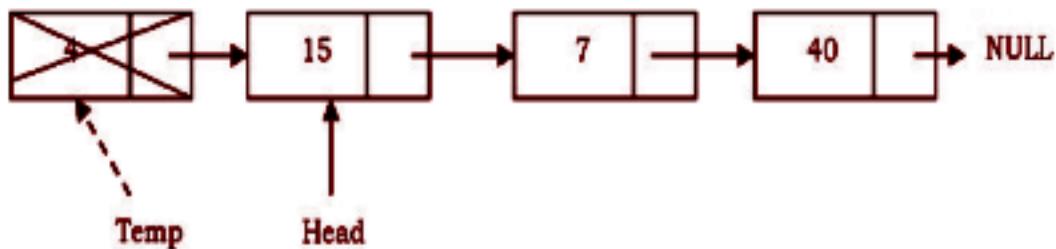
Deleting the First Node in Singly Linked List

It can be done in two steps:

- Create a temporary node which will point to the same node as that of the head.



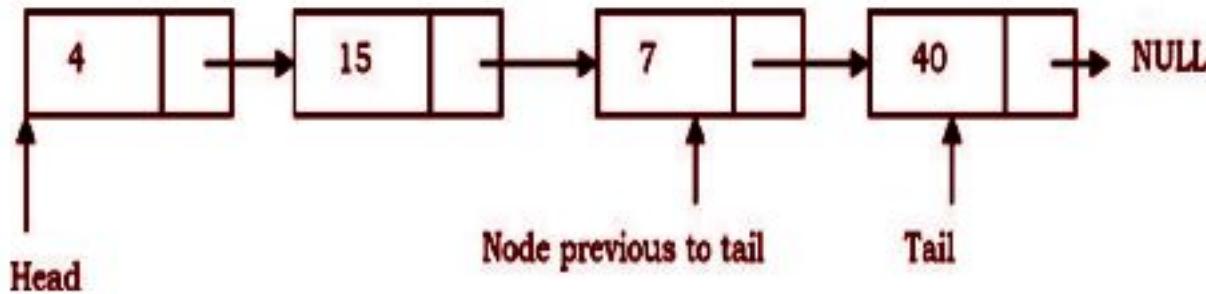
- Now, move the head nodes pointer to the next node and dispose of the temporary node.



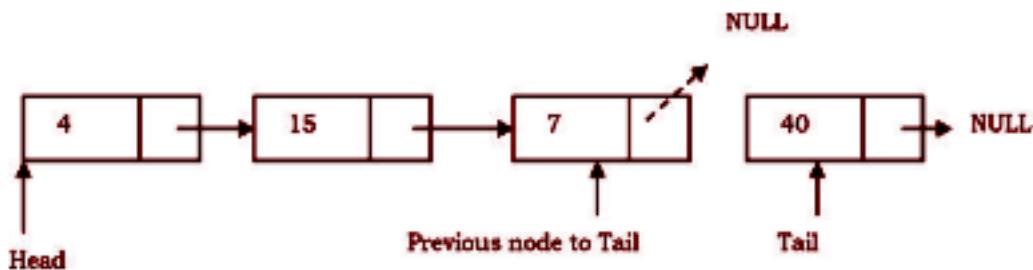
Deleting the Last Node in Singly Linked List

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

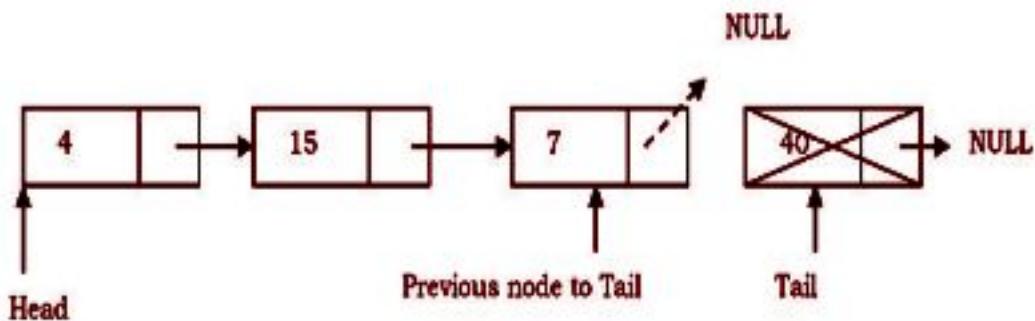
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail node and the other pointing to the node before the tail node.



- Update the previous node's next pointer with **null**.



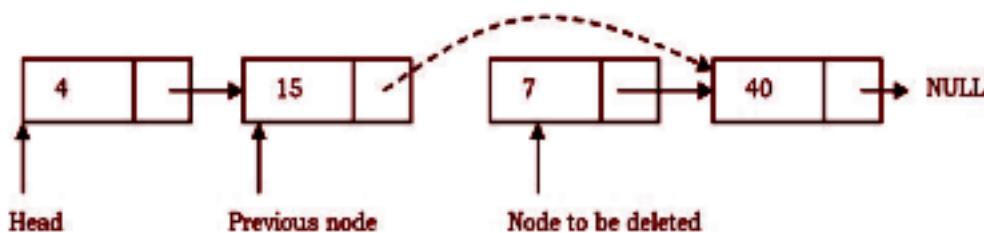
- Dispose of the tail node.



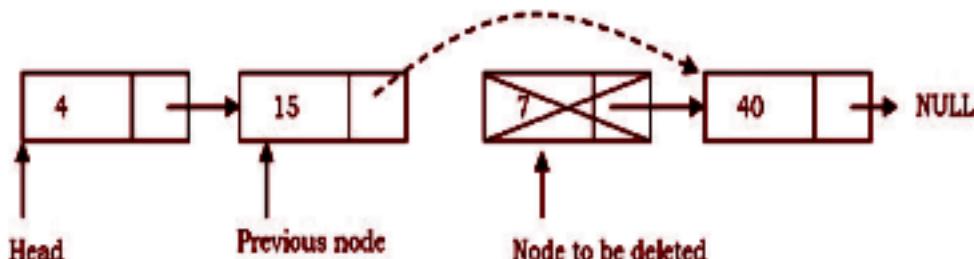
Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such removal can be done in two steps:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



- Dispose of the current node to be deleted.



Insert node recursively

Follow the steps below and try to implement it yourselves:

- If **Head** is **null** and **position** is not 0. Then exit it.
- If **Head** is **null** and **position** is 0. Then insert a new Node to the **Head** and exit it.
- If **Head** is not **null** and **position** is 0. Then the **Head** reference set to the new Node.
Finally, a new Node set to the Head and exit it.
- If not, iterate until finding the Nth position or **end**.

For the code, refer to the Solution section of the problem.

Delete node recursively

Follow the steps below and try to implement it yourselves:

- If the node to be deleted is the **root**, simply delete it.
- To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if the position is not zero, we run a loop **position-1** times and get a pointer to the previous node.
- Now, simply point the previous node's next to the current node's next and delete the current node.

For the code, refer to the Solution section of the problem.

Linked-List 2

Now moving further with the topic, let's try to solve some problems now...

The Midpoint of A Linked List

The Trivial Approach- Two Passes

- This approach requires us to traverse through the linked list twice i.e. 2 passes.
- In the first pass, we will calculate the **length** of the linked list. After every iteration, update the **length** variable.
- In the second pass, we will find the element of the linked list at the **(length-1)/2th** position. This element shall be the middle element in the linked list.
- However, we wish to traverse the linked list only once, therefore let us see another approach.

The Optimal Approach- One Pass

- The midpoint of a linked list can be found out very easily by taking two pointers, one named **slow** and the other named **fast**.
- As their names suggest, they will move in the same way respectively.
- The **fast** pointer will move ahead **two pointers at a time**, while the **slow** pointer one will move at a speed of **a pointer at a time**.
- In this way, when the fast pointer will reach the end, by that time the slow pointer will be at the middle position of the array.
- These pointers will be updated like this:
 - **slow = slow.next**
 - **fast = fast.next.next**

Java Code

```
public static Node returnMiddle(Node headNode){  
    if (headNode == null || headNode.next == null)  
        return head;  
    Node slow = headNode; //Slow pointer  
    Node fast = headNode.next; //Fast Pointer  
    while (fast != null && fast.next != null){  
        fast = fast.next.next;  
        slow = slow.next;  
    }  
    return slow; // Slow pointer shall point to our middle element  
}
```

Note:

- For odd length there will be only one middle element, but for the even length there will be two middle elements.
- In case of an even length LL, both these approaches will return the first middle element and the other one will be the direct **next** of the first middle element.

Merge Two sorted linked lists

- We will be merging the linked list, similar to the way we performed merge over two sorted arrays.
- We will be using the two **head** pointers, compare their data and the one found smaller will be directed to the new linked list, and increase the **head** pointer of the corresponding linked list.
- Just remember to maintain the **head** pointer separately for the new sorted list.
- And also if one of the linked list's length ends and the other one's not, then the remaining linked list will directly be appended to the final list.
- Try to implement this approach on your own.

Mergesort over a linked list

- Like the merge sort algorithm is applied over the arrays, the same way we will be applying it over the linked list.
- Just the difference is that in the case of arrays, the middle element could be easily figured out, but here you have to find the middle element, each time you send the linked list to split into two halves using the above approach.
- The merging part of the divided lists can also be done using the [merge sorted linked lists code](#) as discussed above.
- The functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.
- Try to implement this approach on your own.

Reverse the linked list

Recursive approach:

- In this approach, we will store the last element of the list in the small answer, and then update that by adding the [next last node](#) and so on.
- Finally, when we will be reaching the first element, we will assign the **next** to **null**.
- Follow the Java code below, for better understanding.

```
public static Node reverseLinkedList(Node head){  
    if (headNode == null || headNode.next == null)  
        return head;  
  
    Node smallHead = reverseLinkedList(head.next);  
    Node tail = smallHead;  
    while(tail.next != null){  
        tail = tail.next;  
    }  
    tail.next = head;
```

```

    head.next = null;
    return smallHead;
}

```

After calculation, you can see that this code has a time complexity of **O(n²)**. Now let's think about how to improve it.

Recursive approach (Optimal):

- There is another recursive approach to the order of **O(n)**.
- What we will be doing is that head but also the tail pointer, which can save our time in searching over the list to figure out the tail pointer for appending or removing.
- Check out the given code for your reference:

```

class Pair{
    Node head;
    Node tail;
    public Pair(Node head, Node tail){
        this.head = head;
        this.tail = tail;
    }
}

class main{
    private static Pair reverse2Helper(Node head){
        if (head == null || head.next == null)
            return new Pair(head, head);

        Pair p = reverse2Helper(head.next);
        p.tail.next= head;
        head.next = null;
        return new Pair(p.head,head);
    }

    private static Node reverse2(Node head){
        return reverse2Helper(head).head;
    }
}

```

```
// Main driver function can be written by yourself
}
```

Now let us try to improve this code further.

A simple observation is that the **tail** is always **head.next**. By making the recursive call we can directly use this as our **tail** pointer and reverse the linked list by **tail.next = head**. Refer to the code below, for better understanding.

```
public static Node reverse3(Node head){
    if (head == null || head.next == null)
        return head;

    smallHead = reverse3(head.next);
    tail = head.next;
    tail.next = head;
    head.next = null;
    return smallHead;
}
```

Iterative approach:

- We will be using three-pointers in this approach: **previous**, **current**, and **next**.
- Initially, the **previous** pointer would be **null** as in the reversed linked list, we want the original head to be the last element pointing to **null** .
- The **current** pointer will point to the current node whose **next** will be pointing to the previous element but before pointing it to the previous element, we need to store the next element's address somewhere otherwise we will lose that element.
- Similarly, iteratively, we will keep updating the pointers as **current** to the **next**, **previous** to the **current**, and **next** to **current's next**.

Refer to the given Java code for better understanding:

```
public static Node reverse(Node head){  
    if (head == null || head.next == null)  
        return head;  
  
    Node prev = null;  
    Node curr = head.next;  
    Node next = curr.next;  
  
    while (next != null){  
        curr.next = prev;  
        prev = curr;  
        curr = next;  
        next = next.next;  
    }  
    curr.next = prev;  
    return curr;  
}
```

Stacks

What you will learn in this lecture?

- Operations on stack.
- Implementation of stack.
- Use of inbuilt stack.

Introduction

- Stacks are simple data structures that allow us to store and retrieve data sequentially.
- A stack is a linear data structure like arrays and linked lists.
- It is an abstract data type(**ADT**).
- In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out**.
- Consider the example of a pile of books:

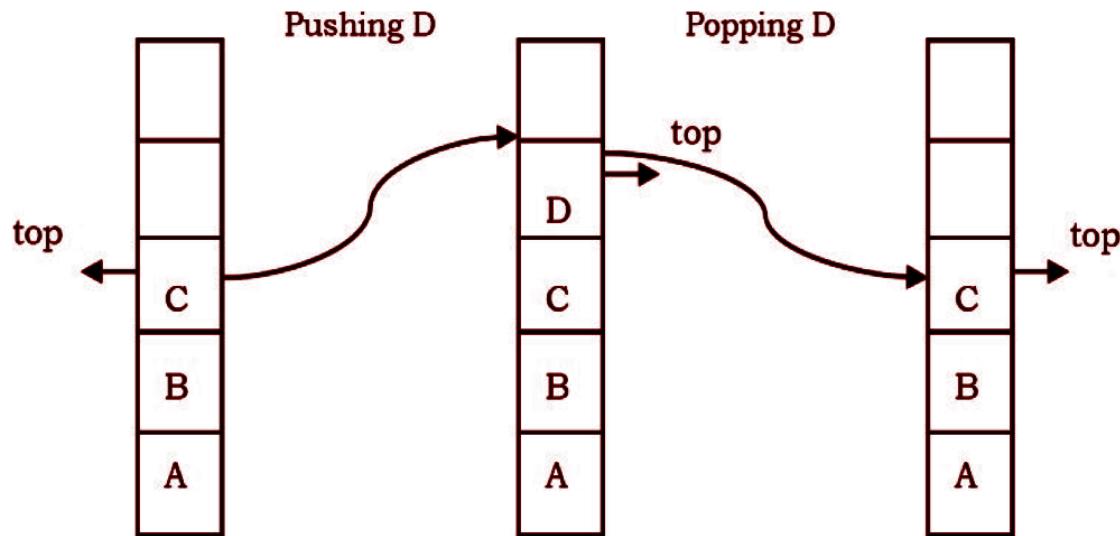


Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO**.

Operations on the stack:

- In a stack, insertion and deletion are done at one end, called **top**.
- **Insertion:** This is known as a **push** operation.
- **Deletion:** This is known as a **pop** operation.



Main stack operations

- **push (int data):** Insert data onto the stack.
- **int pop():** Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- `int top()`: Returns the last inserted element without removing it.
- `int size()`: Returns the number of elements stored in the stack.
- `boolean isEmpty()`: Indicates whether any elements are stored in the stack or not.

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of Push()	$O(1)$
Time Complexity of Pop()	$O(1)$
Time Complexity of Size()	$O(1)$
Time Complexity of IsEmptyStack()	$O(1)$
Time Complexity of IsFullStack()	$O(1)$
Time Complexity of DeleteStackQ	$O(1)$

Exceptions

- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- Attempting the execution of `pop()` on an empty stack throws an exception called **Stack Underflow**.
- Trying to push an element in a full-stack throws an exception called **Stack Overflow**.

Implementing stack- Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the **top** element.



Consider the given implementation in Java for more understanding:

```
class StackUsingArray{
    int[] data;                      // Dynamic array created serving as stack
    int nextIndex;                   // To keep the track of current top index
    int capacity;                   // To keep the track of total size of stack

    public StackUsingArray(int totalSize) {      //Constructor
        data = new int[totalSize];
        nextIndex = 0;
        capacity = totalSize;
    }

    // return the number of elements present in my stack
    public int size() {
        return nextIndex;
    }

    public boolean isEmpty() {
        /*
        if(nextIndex == 0) {
            return true;
        }
        else {
            return false;
        */
        return nextIndex == 0;      //Above program written in short-hand
    }

    // insert element
}
```

```

public void push(int element) {
    if(nextIndex == capacity) {
        System.out.println("Stack full");
        return;
    }
    data[nextIndex] = element;
    nextIndex++;                                //Size incremented
}

// delete element
public int pop() {
    //Before deletion check for empty to prevent underflow
    if(isEmpty()) {
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    nextIndex--;                                //Conditioned satisfied so deleted
    return data[nextIndex];
}

//to return the top element of the stack
public int top() {
    if(isEmpty()) {                            // checked for empty stack
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    return data[nextIndex - 1];
}
}
  
```

Limitations of Simple Array Implementation

In programming languages like C++, Java, etc, the maximum size of an array must first be defined i.e. it is fixed and it cannot be changed.

Dynamic Stack

There is one limitation to the above approach, which is the size of the stack is fixed. In order to overcome this limitation, whenever the size of the stack reaches its limit

we will simply double its size. To get the better understanding of this approach, look at the code below...

```

class StackUsingArray{

    int[] data;                      // Dynamic array created serving as stack
    int nextIndex;                   // To keep the track of current top index
    int capacity;                   // To keep the track of total size of stack

    public StackUsingArray() {      //Constructor
        data = new int[4];
        nextIndex = 0;
        capacity = 4;
    }

    // return the number of elements present in my stack
    public int size() {
        return nextIndex;
    }

    public boolean isEmpty() {
        /*
        if(nextIndex == 0) {
            return true;
        }
        else {
            return false;
        }
        */
    }

    return nextIndex == 0;          //Above program written in short-hand
}

// insert element
public void push(int element) {
    if(nextIndex == capacity) {
        int newData[] = new int[2 * capacity]; //Capacity doubled
        for(int i = 0; i < capacity; i++) {
            newData[i] = data[i];           //Elements copied
        }
        capacity *= 2;
        data = newData;
    }
    data[nextIndex] = element;
    nextIndex++;                      //Size incremented
}

```

```

// delete element
public int pop() {
    //Before deletion check for empty to prevent underflow
    if(isEmpty()) {
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    nextIndex--;           //Conditioned satisfied so deleted
    return data[nextIndex];
}

//to return the top element of the stack
public int top() {
    if(isEmpty()) {           // checked for empty stack
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    return data[nextIndex - 1];
}
}

```

Stack using templates for Generic Data type Stack

While implementing the dynamic stack, we kept ourselves limited to the data of type integer only, but what if we want a generic stack(something that works for every other data type as well). For this we will be using templates. Refer the code below(based on the similar approach as used while creating dynamic stack):

```

class StackUsingArray <T>{

    T[] data;           // Dynamic array created serving as stack
    int nextIndex;     // To keep the track of current top index
    int capacity;      // To keep the track of total size of stack

    public StackUsingArray() {   //Constructor
        data = new T[4];
        nextIndex = 0;
        capacity = 4;
    }

    // return the number of elements present in my stack
}

```

```

public int size() {
    return nextIndex;
}

public boolean isEmpty() {
    /*
        if(nextIndex == 0) {
            return true;
        }
        else {
            return false;
        }
    */
}

return nextIndex == 0;      //Above program written in short-hand
}

// insert element
public void push(T element) {
    if(nextIndex == capacity) {
        T newData[] = new T[2 * capacity]; //Capacity doubled
        for(int i = 0; i < capacity; i++) {
            newData[i] = data[i];           //Elements copied
        }
        capacity *= 2;
        data = newData;
    }
    data[nextIndex] = element;
    nextIndex++;                      //Size incremented
}

// delete element
public T pop() {
    //Before deletion check for empty to prevent underflow
    if(isEmpty()) {
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    nextIndex--;                     //Conditioned satisfied so deleted
    return data[nextIndex];
}

//to return the top element of the stack
public T top() {
    if(isEmpty()) {                // checked for empty stack
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
}

```

```

        }
        return data[nextIndex - 1];
    }
}

```

You can see that every function whose return type was int initially now returns T type (i.e., template-type).

Generally, the template approach of stack is preferred as it can be used for any data type irrespective of it being int, char, float, etc.

Stack using Generic Linked Lists

Till now we have learned how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists. All the five functions that stacks can perform could be made using linked lists:

```

class Node<T> {                                //Node class for Linked List
    T data;
    Node<T> next;

    Node(T data) {
        this.data = data;
        next = NULL;
    }
    Node() {
        next = null;
    }
}

class Stack {
    Node<T> head;
    Node<T> tail;
    int size;           // number of elements present in stack

    public Stack() {      // Constructor to initialize the head and
                          // tail to NULL and size to zero
    }
}

```

```

public int getSize() { // traverse the LL and return its length
}

public boolean isEmpty() { // check if the head pointer is NULL or not
}

public void push(T element) { // insert the newNode at the end
                           // update the tail node
}

public T pop() { // remove the tail node and then update the tail
                   // pointer to the previous position
}

public T top() { //return the value at the tail node.
}
}
  
```

Inbuilt Stack in Java

Java provides the in-built stack in its **library** which can be used instead of creating/writing a stack class each time. To use this stack, we need to use the import following file:

```
import java.util.Stacks;
```

To declare a stack use the following syntax:

```
Stack <datatype_that_will_be_stored> Name_of_stack = new Stack<>();
```

There are various functions available in this module:

- **st.push(value_to_be_inserted)** : To insert a value in the stack
- **st.top()** : Returns the value at the top of the stack
- **st.pop()** : Deletes the value at the top from the stack.
- **st.size()** : Returns the total number of elements in the stack.

- **st.isEmpty()** : Returns a boolean value (True for empty stack and vice versa).

Problem Statement- Balanced Parenthesis

For a given string expression containing only round brackets or parentheses, check if they are balanced or not. Brackets are said to be balanced if the bracket which opens last, closes first. You need to return a boolean value indicating whether the expression is balanced or not.

Approach:

- We will use stacks.
- Each time, when an open parenthesis is encountered, push it in the stack, and when closed parenthesis is encountered, match it with the top of the stack and pop it.
- If the stack is empty at the end, return Balanced otherwise, Unbalanced.

Java Code:

```

public String checkBalanced(String inputStr){//Function to check parentheses
    Stack<Character> s = new Stack<>(); //The stack
    for(char i : inputStr.toCharArray()){
        if (i=='[' || i=='{' || i=='(')
            s.push(i);
        else if (i==']' || i=='}' || i==')')
            if (s.size()>0 && s.top()==i)
                s.pop();
            else
                return "Unbalanced";
    }
    if (s.size() == 0)
        return "Balanced";
    else
        return "Unbalanced";
}
  
```

Queues

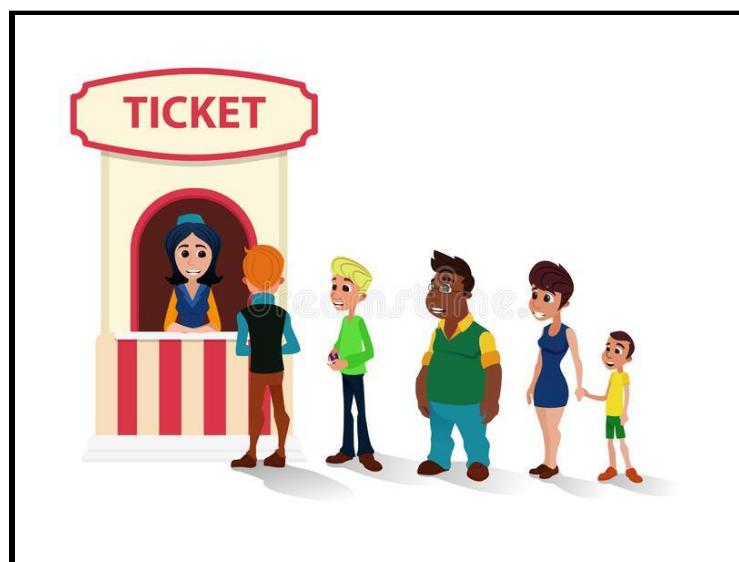
What you will learn in this lecture?

- Operations on queue.
- Implementation of queue.
- Use of inbuilt queue.

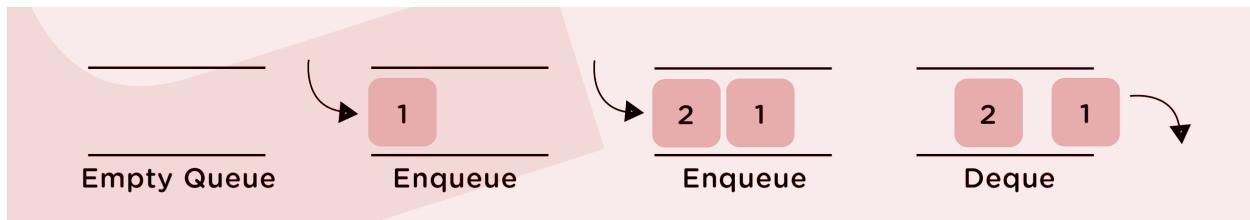
Introduction

- Like stack, the queue is also an abstract data type.
- As the name suggests, in queue elements are inserted at one end while deletion takes place at the other end.
- Queues are open at both ends, unlike stacks that are open at only one end(the top).

Let us consider a queue at a movie ticket counter:



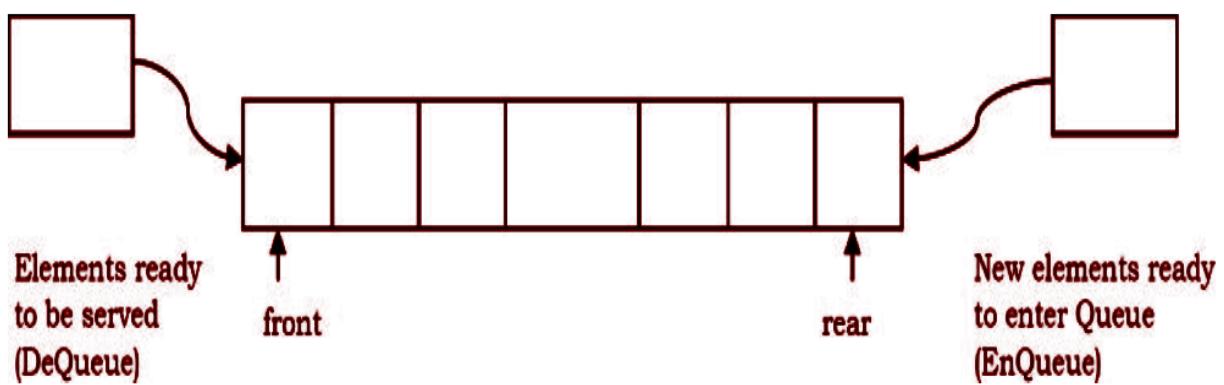
- Here, the person who comes first in the queue is served first with the ticket while the new seekers of tickets are added back in the line.
- This order is known as **First In First Out (FIFO)**.
- In programming terminology, the operation to add an item to the queue is called "enqueue", whereas removing an item from the queue is known as "dequeue".

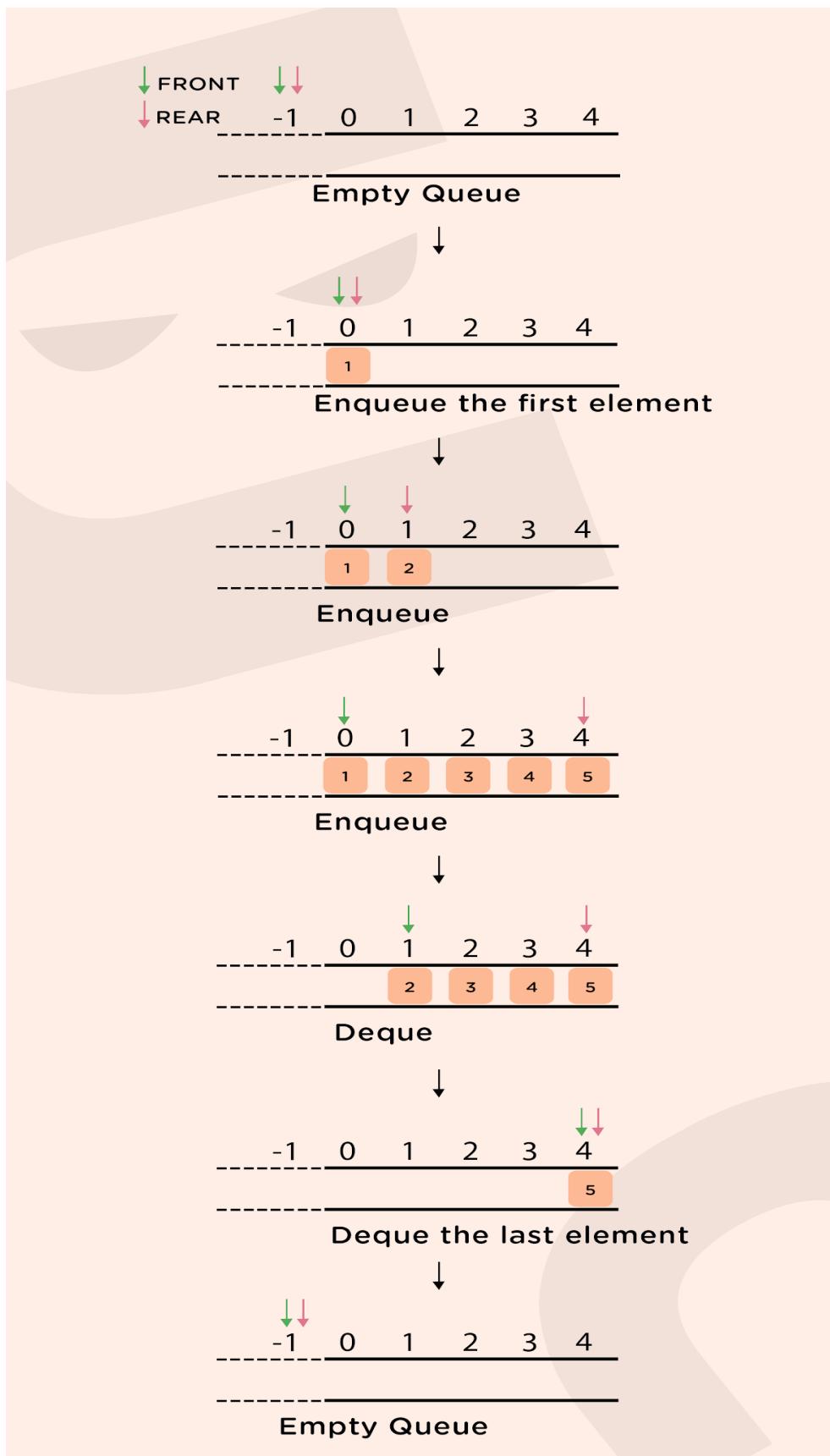


Working of A Queue

Queue operations work as follows:

1. Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On **enqueueing** an element, we increase the value of the REAR index and place the new element in the position pointed to by REAR.
4. On **dequeuing** an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueueing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueueing the first element, we set the value of FRONT to 0.
8. When dequeuing the last element, we reset the values of FRONT and REAR to -1.





Applications of queue

- CPU Scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processes Queue is used for synchronization. eg: IO Buffers, pipes, file IO, etc.
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people in order of their calling.

Implementation of A Queue Using Array

Queue contains majorly these five functions that we will be implementing:

- **enqueue()**: Insertion of element
- **dequeue()**: Deletion of element
- **front()**: returns the element present in the front position
- **getSize()**: returns the total number of elements present at current stage
- **isEmpty()**: returns boolean value, TRUE for empty and FALSE for non-empty.

Now, let's implement these functions in our program.

NOTE: We will be using templates in the implementation, so that it can be generalised.

```
class QueueUsingArray <T> {
    T data;                                // to store data
    int nextIndex;                          // to store next index
    int firstIndex;                         // to store the first index
    int size;                               // to store the size
    int capacity;                           // to store the capacity it can hold

    public QueueUsingArray(int s) {          // Constructor to initialize values
        data = new T[s];
        nextIndex = 0;
        firstIndex = -1;
    }
}
```

```

        size = 0;
        capacity = s;
    }

public int getSize() {           // Returns number of elements present
    return size;
}

public boolean isEmpty() {        // To check if queue is empty or not
    return size == 0;
}

public void enqueue(T element) {      // Function for insertion
    if(size == capacity) { // To check if the queue is already full
        System.out.println("Queue Full!");
        return;
    }
    data[nextIndex] = element;      // Otherwise added a new element
    nextIndex = (nextIndex + 1) % capacity ;    // in cyclic way
    if(firstIndex == -1) {          // Suppose if queue was empty
        firstIndex = 0;
    }
    size++;                      // Finally, incremented the size
}

public T front() {           // To return the element at front position
    if(isEmpty()) {      // To check if the queue was initially empty
        System.out.println("Queue is Empty!");
        return 0;
    }
    return data[firstIndex];     // otherwise returned the element
}

public T dequeue() {           // Function for deletion
    if(isEmpty()) {      // To check if the queue was empty
        System.out.println("Queue is Empty!");
        return 0;
    }
    T ans = data[firstIndex];
    firstIndex = (firstIndex + 1) % capacity;
    size--;                  // Decrementing the size by 1
    if(size == 0) { // If queue becomes empty after deletion, then
        firstIndex = -1;    // resetting the original parameters
        nextIndex = 0;
    }
}

```

```

        }
        return ans;
    }
}

```

Dynamic queue

In the dynamic queue, we will be preventing the condition where the queue becomes full and we were not able to insert any further elements in that.

As we all know that when the queue is full it means the internal array that we are using in the form of queue has become full, we can resolve this problem by creating a new array of double the size of previous one and copy pasting the elements of previous array to the new one. Now this new array which has the double size will be considered as our queue. We will do this in insert function when we check for queue full (`size==capacity`), when this happens we will discard the previous array and create a new array of double size, copy pasting all the elements so that we don't lose the data. Let's now check the implementation of the same.

Implementation is pretty similar to the static approach discussed above. A few minor changes are there which could be followed with the help of comments in the code below.

```

class QueueUsingArray <T> {
    T data;                                // to store data
    int nextIndex;                          // to store next index
    int firstIndex;                         // to store the first index
    int size;                               // to store the size
    int capacity;                           // to store the capacity it can hold

    public QueueUsingArray() {   // Constructor to initialize values
        data = new T[4];
        nextIndex = 0;
        firstIndex = -1;
        size = 0;
        capacity = 4;
    }
}

```

```

}

public int getSize() {           // Returns number of elements present
    return size;
}

public boolean isEmpty() {      // To check if queue is empty or not
    return size == 0;
}

public void enqueue(T element) {        // Function for insertion
    if(size == capacity) { // To check if the queue is already full
        T[] newData = new T[2 * capacity]; // we simply doubled
                                            // the capacity
        int j = 0;
        for(int i=firstIndex; i<capacity; i++) { // Now copied the
                                            //Elements to new one
            newData[j] = data[i];
            j++;
        }
        for(int i=0; i<firstIndex; i++) { //Overcoming the initial
                                            // cyclic insertion by copying
                                            // the elements linearly
            newData[j] = data[i];
            j++;
        }
        data = newData;
        firstIndex = 0;
        nextIndex = capacity;
        capacity *= 2;           // Updated here as well
    }
    data[nextIndex] = element;     // Otherwise added a new element
    nextIndex = (nextIndex + 1) % capacity ; // in cyclic way
    if(firstIndex == -1) {         // Suppose if queue was empty
        firstIndex = 0;
    }
    size++;                      // Finally, incremented the size
}

public T front() {           // To return the element at front position
    if(isEmpty()) { // To check if the queue was initially empty
        System.out.println("Queue is Empty!");
        return 0;
    }
}

```

```

        return data[firstIndex];      // otherwise returned the element
    }

public T dequeue() {           // Function for deletion
    if(isEmpty()) {           // To check if the queue was empty
        System.out.println("Queue is Empty!");
        return 0;
    }
    T ans = data[firstIndex];
    firstIndex = (firstIndex + 1) % capacity;
    size--;                // Decrementing the size by 1
    if(size == 0) { // If queue becomes empty after deletion, then
        firstIndex = -1; // resetting the original parameters
        nextIndex = 0;
    }
    return ans;
}
}

```

Queues using Generic LL

Given below is an implementation of Queue using Linked List. This is similar to the way we wrote the LL Implementation for a Stack:

```

class Node <T> {           // Node class for Linked List, no change needed
    T data;
    Node<T> next;
    Node(T data) {
        this->data = data;
        next = NULL;
    }
}

class Queue <T> {
    Node<T> head;           // for storing front of queue
    Node<T> tail;           // for storing tail of queue
    int size;                // number of elements in queue

    public Queue() {         // Constructor to initialise head, tail to NULL
        // and size to 0
    }

    public int getSize() {    // just return the size of Linked List

```

```

    }

public boolean isEmpty() { // just check if head is NULL or not

}

public void enqueue(T element) { // Simply insert the new node
//at the tail of LL

}

public T front() { // Returns the head pointer of LL.
// Be careful for the case when size is 0
}

public T dequeue() { // moves the head pointer one position ahead
// and deletes the head pointer.
// Also decrease the size by 1
}
}

```

In-built Queue in Java

Java provides the in-built queue in its **library** which can be used instead of creating/writing a queue class each time. To use this queue, we need to use the import following file:

```

import java.util.Queues;
import java.util.LinkedList;

```

Key functions of this in-built queue:

- **.push(element_value)** : Used to insert the element in the queue
- **.pop()** : Used to delete the element from the queue
- **.front()** : Returns the element at front of the queue

- **.size()** : Returns the total number of elements present in the queue
- **.isEmpty()** : Returns TRUE if the queue is empty and vice versa

Let us now consider an example to implement queue using inbuilt library:

Problem Statement: Implement the following parts using queue:

1. Declare a queue of integers and insert the following elements in the same order as mentioned: 10, 20, 30, 40, 50, 60.
2. Now tell the element that is present at the front position of the queue
3. Now delete an element from the front side of the queue and again tell the element present at the front position of the queue.
4. Print the size of the queue and also tell if the queue is empty or not.
5. Now, print all the elements that are present in the queue.

```

import java.util.Queues;
import java.util.LinkedList;

Class QueueTesting{
    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<>();
        q.push(10);           // part 1
        q.push(20);
        q.push(30);
        q.push(40);
        q.push(50);
        q.push(60);
        System.out.println(q.front());      // Part 2
        q.pop();                // Part 3
        System.out.println(q.front());      // Part 3
        System.out.println(q.size());       // Part 4
        System.out.println(q.isEmpty());    // prints 1 for TRUE and 0 for
                                         // FALSE(Part 4)

        while(!q.isEmpty()) { // prints all the elements until the queue
                             // is empty (Part 5)
            System.out.println(q.front());
            q.pop();
        }
    }
}

```

```
        }  
    }  
}
```

We get the following output:

```
10  
20  
5  
0  
20  
30  
40  
50  
60
```

Queues

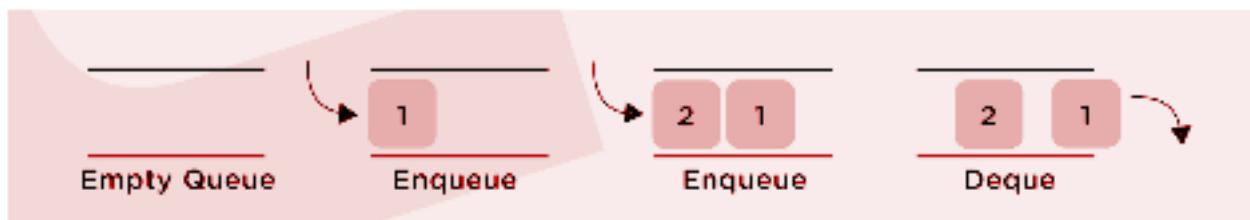
Introduction

- Like stack, the queue is also an abstract data type.
- As the name suggests, in queue elements are inserted at one end while deletion takes place at the other end.
- Queues are open at both ends, unlike stacks that are open at only one end(the top).

Let us consider a queue at a movie ticket counter:



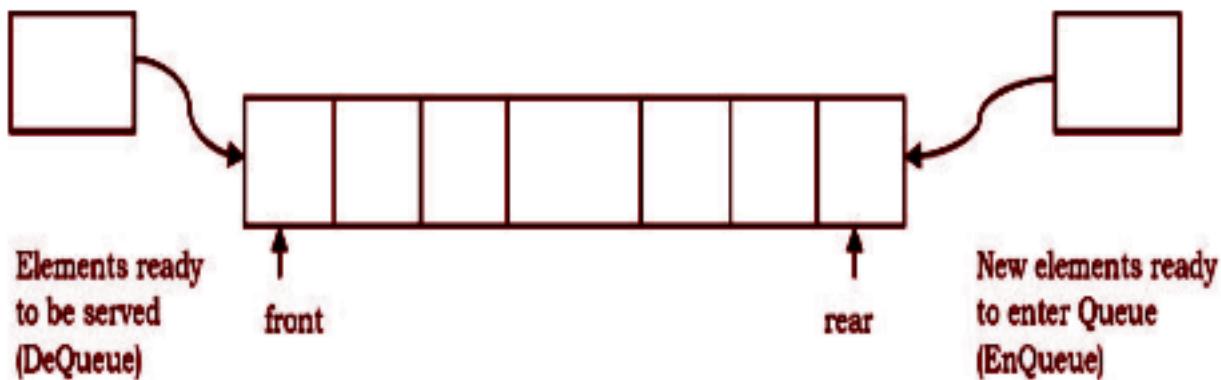
- Here, the person who comes first in the queue is served first with the ticket while the new seekers of tickets are added back in the line.
- This order is known as **First In First Out (FIFO)**.
- In programming terminology, the operation to add an item to the queue is called "enqueue", whereas removing an item from the queue is known as "dequeue".

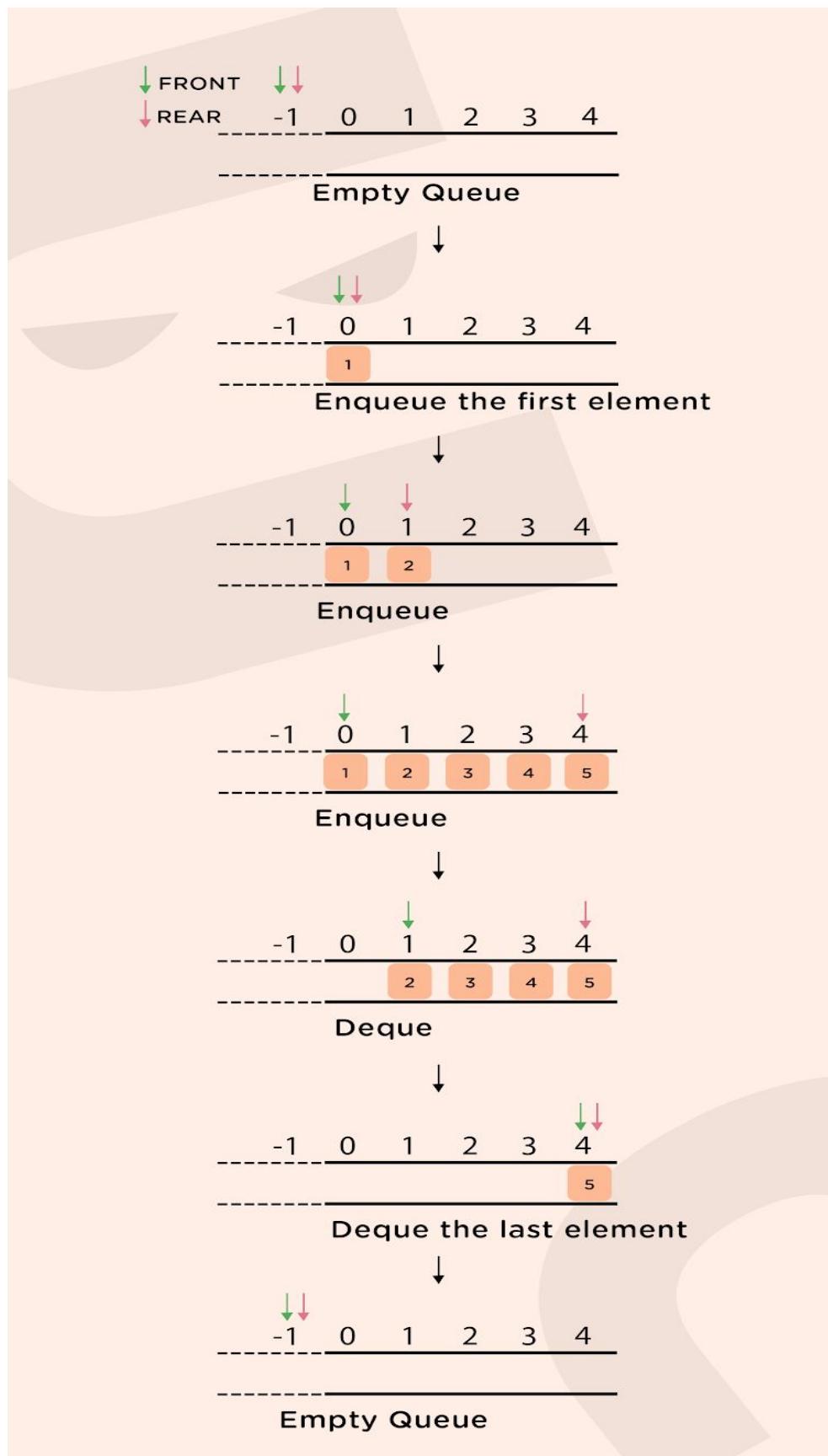


Working of A Queue

Queue operations work as follows:

1. Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On **enqueueing** an element, we increase the value of the REAR index and place the new element in the position pointed to by REAR.
4. On **dequeuing** an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueueing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueueing the first element, we set the value of FRONT to 0.
8. When dequeuing the last element, we reset the values of FRONT and REAR to -1.





Applications of queue

- CPU Scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processes Queue is used for synchronization. eg: IO Buffers, pipes, file IO, etc.
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people in order of their calling.

Implementation of A Queue Using Array

Queue contains majorly these five functions that we will be implementing:

- **enqueue()**: Insertion of element
- **dequeue()**: Deletion of element
- **front()**: returns the element present in the front position
- **getSize()**: returns the total number of elements present at current stage
- **isEmpty()**: returns boolean value, TRUE for empty and FALSE for non-empty.

Now, let's implement these functions in our program.

NOTE: We will be using templates in the implementation, so that it can be generalised.

```
class QueueUsingArray <T> {
    T data;                                // to store data
    int nextIndex;                          // to store next index
    int firstIndex;                         // to store the first index
    int size;                               // to store the size
    int capacity;                           // to store the capacity it can hold

    public QueueUsingArray(int s) {          // Constructor to initialize values
        data = new T[s];
        nextIndex = 0;
        firstIndex = -1;
    }
}
```

```

        size = 0;
        capacity = s;
    }

public int getSize() {           // Returns number of elements present
    return size;
}

public boolean isEmpty() {        // To check if queue is empty or not
    return size == 0;
}

public void enqueue(T element) {   // Function for insertion
    if(size == capacity) { // To check if the queue is already full
        System.out.println("Queue Full!");
        return;
    }
    data[nextIndex] = element;      // Otherwise added a new element
    nextIndex = (nextIndex + 1) % capacity ; // in cyclic way
    if(firstIndex == -1) {         // Suppose if queue was empty
        firstIndex = 0;
    }
    size++;                         // Finally, incremented the size
}

public T front() {               // To return the element at front position
    if(isEmpty()) {             // To check if the queue was initially empty
        System.out.println("Queue is Empty!");
        return 0;
    }
    return data[firstIndex];     // otherwise returned the element
}

public T dequeue() {             // Function for deletion
    if(isEmpty()) {             // To check if the queue was empty
        System.out.println("Queue is Empty!");
        return 0;
    }
    T ans = data[firstIndex];
    firstIndex = (firstIndex + 1) % capacity;
    size--;                      // Decrementing the size by 1
    if(size == 0) {             // If queue becomes empty after deletion, then
        firstIndex = -1;          // resetting the original parameters
        nextIndex = 0;
    }
}

```

```

        }
        return ans;
    }
}

```

Dynamic queue

In the dynamic queue, we will be preventing the condition where the queue becomes full and we were not able to insert any further elements in that.

As we all know that when the queue is full it means the internal array that we are using in the form of a queue has become full, we can resolve this problem by creating a new array of double the size of the previous one and copy pasting the elements of the previous array to the new one. Now this new array which has the double size will be considered as our queue. We will do this in insert function when we check for queue full (`size==capacity`), when this happens we will discard the previous array and create a new array of double size, copy pasting all the elements so that we don't lose the data. Let's now check the implementation of the same.

Implementation is pretty similar to the static approach discussed above. A few minor changes are there which could be followed with the help of comments in the code below.

```

class QueueUsingArray <T> {
    T data;                                // to store data
    int nextIndex;                          // to store next index
    int firstIndex;                         // to store the first index
    int size;                               // to store the size
    int capacity;                           // to store the capacity it can hold

    public QueueUsingArray() {   // Constructor to initialize values
        data = new T[4];
        nextIndex = 0;
        firstIndex = -1;
        size = 0;
        capacity = 4;
    }
}

```

```

}

public int getSize() {           // Returns number of elements present
    return size;
}

public boolean isEmpty() { // To check if queue is empty or not
    return size == 0;
}

public void enqueue(T element) {      // Function for insertion
    if(size == capacity) { // To check if the queue is already full
        T *newData = new T[2 * capacity];// we simply doubled the
                                         // capacity
        int j = 0;
        for(int i=firstIndex; i<capacity; i++) { // Now copied the
                                         //Elements to new one
            newData[j] = data[i];
            j++;
        }
        for(int i=0; i<firstIndex; i++) { //Overcoming the initial
                                         // cyclic insertion by copying
                                         // the elements linearly
            newData[j] = data[i];
            j++;
        }
        data = newData;
        firstIndex = 0;
        nextIndex = capacity;
        capacity *= 2;           // Updated here as well
    }
    data[nextIndex] = element;     // Otherwise added a new element
    nextIndex = (nextIndex + 1) % capacity ; // in cyclic way
    if(firstIndex == -1) {         // Suppose if queue was empty
        firstIndex = 0;
    }
    size++;                      // Finally, incremented the size
}

public T front() {           // To return the element at front position
    if(isEmpty()) { // To check if the queue was initially empty
        System.out.println("Queue is Empty!");
        return 0;
    }
}

```

```

        return data[firstIndex];      // otherwise returned the element
    }

public T dequeue() {           // Function for deletion
    if(isEmpty()) {           // To check if the queue was empty
        System.out.println("Queue is Empty!");
        return 0;
    }
    T ans = data[firstIndex];
    firstIndex = (firstIndex + 1) % capacity;
    size--;                // Decrementing the size by 1
    if(size == 0) { // If queue becomes empty after deletion, then
        firstIndex = -1; // resetting the original parameters
        nextIndex = 0;
    }
    return ans;
}
}

```

Queues using Generic LL

Given below is an implementation of Queue using Linked List. This is similar to the way we wrote the LL Implementation for a Stack:

```

class Node <T> {           // Node class for Linked List, no change needed
    T data;
    Node<T> next;
    Node(T data) {
        this->data = data;
        next = NULL;
    }
}

class Queue <T> {
    Node<T> head;           // for storing front of queue
    Node<T> tail;           // for storing tail of queue
    int size;                // number of elements in queue

    public Queue() {         // Constructor to initialise head, tail to NULL
        // and size to 0
    }

    public int getSize() {    // just return the size of Linked List
}

```

```
}

public boolean isEmpty() { // just check if head is NULL or not

}

public void enqueue(T element) { // Simply insert the new node
//at the tail of LL

}

public T front() { // Returns the head pointer of LL.
// Be careful for the case when size is 0
}

public T dequeue() { // moves the head pointer one position ahead
// and deletes the head pointer.
// Also decrease the size by 1
}
```

In-built Queue in Java

Java provides the in-built queue in its **library** which can be used instead of creating/writing a queue class each time. To use this queue, we need to use the import following file:

```
import java.util.Queues;
import java.util.LinkedList;
```

Key functions of this in-built queue:

- **.push(element_value)** : Used to insert the element in the queue
- **.pop()** : Used to delete the element from the queue
- **.front()** : Returns the element at front of the queue

- **.size()** : Returns the total number of elements present in the queue
- **.isEmpty()** : Returns TRUE if the queue is empty and vice versa

Let us now consider an example to implement queue using inbuilt library:

Problem Statement: Implement the following parts using queue:

1. Declare a queue of integers and insert the following elements in the same order as mentioned: 10, 20, 30, 40, 50, 60.
2. Now tell the element that is present at the front position of the queue
3. Now delete an element from the front side of the queue and again tell the element present at the front position of the queue.
4. Print the size of the queue and also tell if the queue is empty or not.
5. Now, print all the elements that are present in the queue.

```

import java.util.Queues;
import java.util.LinkedList;

Class QueueTesting{
    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<>();
        q.push(10);          // part 1
        q.push(20);
        q.push(30);
        q.push(40);
        q.push(50);
        q.push(60);
        System.out.println(q.front());      // Part 2
        q.pop();                // Part 3
        System.out.println(q.front());      // Part 3
        System.out.println(q.size());       // Part 4
        System.out.println(q.isEmpty());    // prints 1 for TRUE and 0 for
                                         // FALSE(Part 4)

        while(!q.isEmpty()) { // prints all the elements until the queue
                             // is empty (Part 5)
            System.out.println(q.front());
            q.pop();
        }
    }
}

```

```
        }  
    }  
}
```

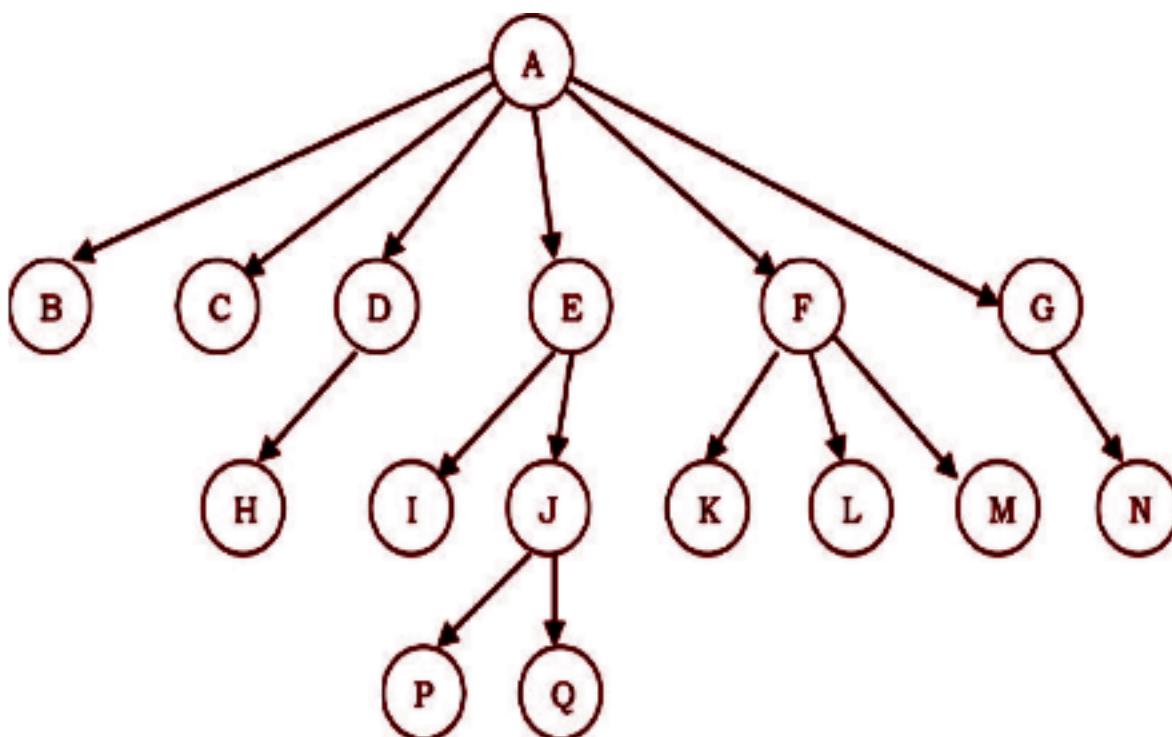
We get the following output:

```
10  
20  
5  
0  
20  
30  
40  
50  
60
```

Trees

Introduction

- In the previous modules, we discussed binary trees where each node can have a maximum of two children and these can be represented easily with two pointers i.e right child and left child.
- But suppose, we have a tree with many children for each node.
- If we do not know how many children a node can have, how do we represent such a tree?
- **For example**, consider the tree shown below.



Generic Tree Node

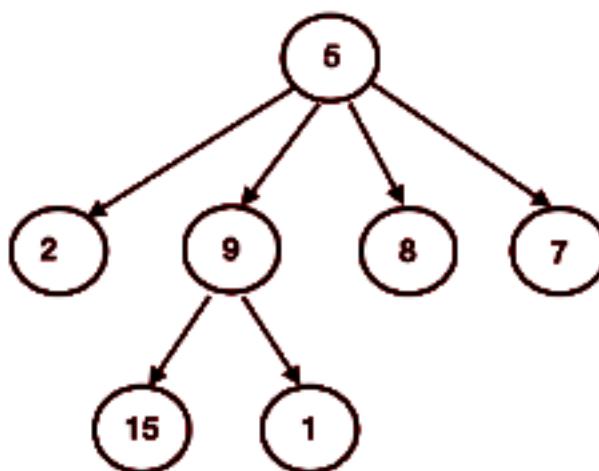
- Implementation of a generic tree node in Java is quite simple.
- The node class will contain two attributes:
 - The node **data**
 - Since there can be a variable number of children nodes, thus the second attribute will be a list of its children nodes. Each child is an instance of the same node class. This is a general **n-nary tree**.
- Consider the given implementation of the **Tree Node** class:

```
class TreeNode <T>{
    T data;
    ArrayList<TreeNode<T>> children;
    public GenericTreeNode(data){
        this.data = data //Node data
        children = new ArrayList<>() //List of children nodes
    }
}
```

Adding Nodes to a Generic Tree

We can add nodes to a generic tree by simply using the **list.add()** function, to add children nodes to parent nodes. This is shown using the given example:

Suppose we have to construct the given Generic Tree:



Consider the given **Java code**:

```
// Create nodes
TreeNode<Integer> n1= TreeNode<>(5);
TreeNode<Integer> n2 =TreeNode<>(2);
TreeNode<Integer> n3 =TreeNode<>(9);
TreeNode<Integer> n4 =TreeNode<>(8);
TreeNode<Integer> n5 =TreeNode<>(7);
TreeNode<Integer> n6 =TreeNode<>(15);
TreeNode<Integer> n7 =TreeNode<>(1);

// Add children for node 5 (n1)
n1.children.add(n2);
n1.children.add(n3);
n1.children.add(n4);
n1.children.add(n5);

// Add children for node 9 (n3)
n3.children.add(n6);
n3.children.add(n7);
```

Print a Generic Tree (Recursively)

In order to print a given generic tree, we will recursively traverse the entire tree.

The steps shall be as follows:

- If the root is **null**, i.e. the tree is an empty tree, then return **null**.
- For every child node at a given node, we call the function recursively.

Go through the given Python code for better understanding:

```
public void printTree(TreeNode root){
    //Not a base case but an edge case
    if (root == null)
        return;

    System.out.println(root.data); //Print current node's data
    for (TreeNode child : root.children)
        printTree(child); //Recursively call the function for children
}
```

Take Generic Tree Input (Recursively)

Go through the given Java code for better understanding:

```

public TreeNode takeTreeInput(){
    System.out.println("Enter root Data");
    int rootData = s.nextInt(); //TAKE USER INPUT
    if (rootData == -1) //Stop taking inputs
        return null;

    TreeNode<Integer> root = TreeNode<>(rootData);

    System.out.println("Enter number of children for " + rootData);
    childrenCount = s.nextInt(); //Get input for no. of child nodes
    while(childrenCount > 0){
        TreeNode child = takeTreeInput(); //Input for all childs
        root.children.add(child); //Add child
        childrenCount--;
    }
    return root;
}

```

Take input level-wise

For taking input level-wise, we will use **queue data structure**. Follow the comments in the code below:

```

public TreeNode<Integer> takeTreeInputLevelwise(){
    System.out.println("Enter root Data");
    int rootData = s.nextInt(); //TAKE USER INPUT
    TreeNode<Integer> root = new TreeNode<int>(rootData);

    Queue<TreeNode<Integer>> pendingNodes = new Queue<>();
    pendingNodes.push(root); // Root data pushed into queue at first

    while(pendingNodes.size() != 0){ //Runs until the queue is not empty
        TreeNode<Integer> front = pendingNodes.front(); //stores front
        pendingNodes.pop(); // deleted that front node stored previously
        System.out.println("Enter num of children of "+front.data);
    }
}

```

```

    int numChild = s.nextInt(); // get the number of child nodes
    for (int i=0; i<numChild; i++) { // iterated over each
        //child node to input it
        System.out.println("Enter "+i+"th child of "+front.data);
        int childData = s.nextInt();
        TreeNode<Integer> child = new TreeNode<>(childData);
        front.children.add(child); //Each child node is pushed
        //into the queue as well as the List of child
        //nodes as it is taken input so that next
        // time we can take its children as input while
        //we kept moving in the level-wise fashion
        pendingNodes.push(child);
    }
}
return root; // Finally returns the root node
}
  
```

Similarly, we can also print the child nodes using a queue itself. Now, try doing the same yourselves and for solution refer to the solution tab of the respective question.

Count total nodes in a tree

To count the total number of nodes in the tree, we will just traverse the tree recursively starting from the root node until we reach the leaf node by iterating over the vector of child nodes. As the size of the child nodes vector becomes 0, we will simply return. Kindly check the code below:

```

public void numNodes(TreeNode<Integer> root){
    if(root == null) { // Edge case
        return 0;
    }
    int ans = 1; // To store total count
    for (int i = 0; i < root.children.size(); i++) {
        ans += numNodes(root.children[i]); // recursively storing count
        // of children's children nodes.
    }
}
  
```

```
    return ans;      // ultimately returning the final answer
}
```

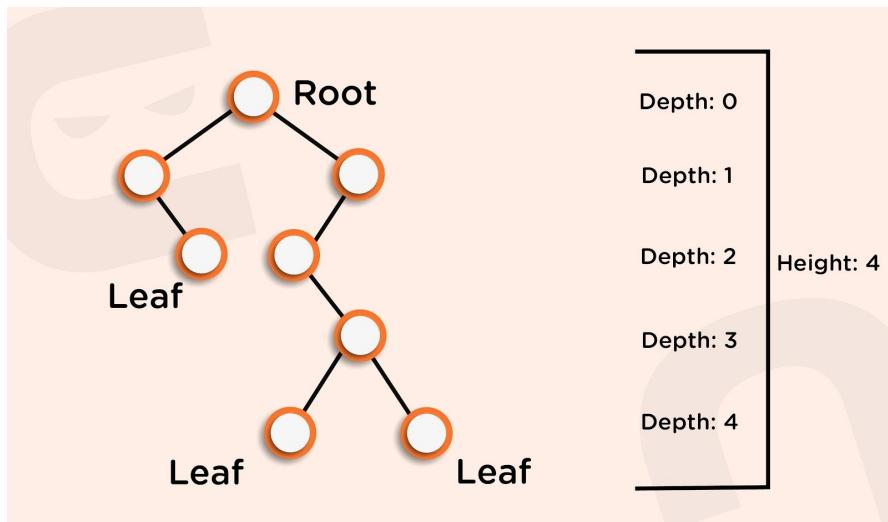
Height of the tree

Height of a tree is defined as the length of the path from the tree's root node to any of its leaf nodes. Just think what should be the height of a tree with just one node? Well, there are a couple of conventions; we can define the height of a tree with just one node to be either 1 or zero. We will be following the convention where the height of a null tree is zero and that with only one node is one. This has been left as an exercise for you, if need be you may follow the code provided in the solution tab of the topic corresponding to the same question.

Approach: Consider the height of the root node as 1 instead of 0. Now, traverse each child of the root node and recursively traverse over each one of them also and the one with the maximum height is added to the final answer along by adding 1 (this 1 is for the current node itself).

Depth of a node

Depth of a node is defined as it's distance from the root node. For example, the depth of the root node is 0, depth of a node directly connected to root node is 1 and so on. Now we will write the code to find the same... (Below is the pictorial representation of the depth of a node)



If you observe carefully, then the depth of the node is just equal to the level in which it resides. We have already figured out how to calculate the level of any node, using a similar approach we will find the depth of the node as well. Suppose, we want to find all the nodes at level 3, then from the root node we will tell its children to find the node that is at level $3 - 1 = 2$, and similarly keep this up recursively until we reach the depth = 0. Look at the code below for better understanding...

```

public void printAtLevelK(TreeNode<Integer> root, int k){
    if(root == null) {                                // Edge case
        return;
    }

    if(k == 0) {                                     // Base case: when the depth is 0
        System.out.println(root.data);
        return;
    }

    for(int i=0; i<root.children.size(); i++) { // Iterating over each
                                                // child and
        printAtLevelK(root.children[i], k - 1); // recursively calling
                                                // with 1 depth less
    }
}

```

Count Leaf nodes

To count the number of leaves, we can simply traverse the nodes recursively until we reach the leaf nodes (the size of the children vector becomes zero). Following recursion, this is very similar to finding the height of the tree. Try to code it yourself and for the solution refer to the solution tab of the same.

Traversals

Traversing the tree is the manner in which we move on the tree in order to access all its nodes. There are generally 4 types of traversals in a tree:

- Level order traversal
- Preorder traversal
- Inorder traversal
- Postorder traversal

We have already discussed level order traversal. Now let's discuss the other traversals.

In Preorder traversal, we visit the current node first(starting with root) and then traverse the left sub-tree. After covering all nodes there, we will move towards the right subtree and visit in a similar manner. Refer the code below:

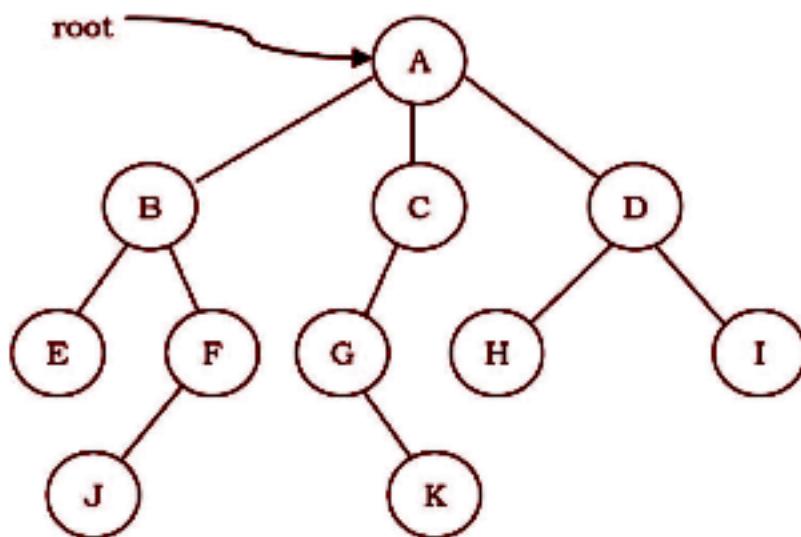
```
public void preorder(TreeNode<Integer> root){  
    if(root == null) {  
        return;  
    }  
    System.out.println(root.data);  
    for(int i = 0; i < root.children.size(); i++) {  
        preorder(root.children[i]);  
    }  
}
```

Binary Trees

What is A Tree?

- A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to several nodes.
- A tree is an example of a non-linear data structure.
- A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

Terminology Of Trees

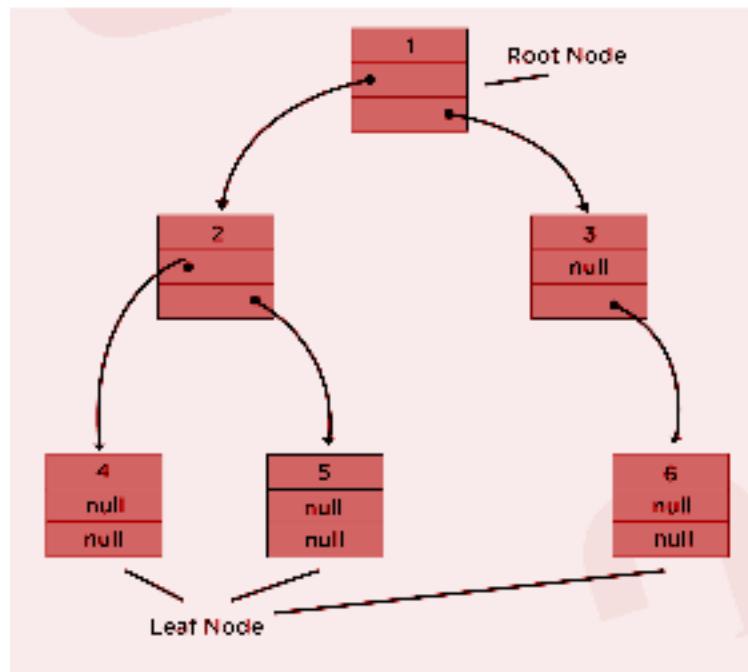


- The root of a tree is the node with no parents. There can be at most one root node in a tree (**node A in the above example**).
- An **edge** refers to the link from a parent to a child (**all links in the figure**).
- A node with no children is called a **leaf node** (**E, J, K, H, and I**).
- The children nodes of the same parent are called **siblings** (**B, C, D are siblings of parent A, and E, F are siblings of parent B**).

- The set of all nodes at a given depth is called the **level** of the tree (**B, C, and D are the same level**). The root node is at level zero.
- The **depth** of a node is the length of the path from the root to the node (**depth of G is 2, A -> C -> G**).
- The **height** of a node is the length of the path from that node to the deepest node.
- The **height** of a tree is the length of the path from the root to the deepest node in the tree.
- A (rooted) tree with only one node (the root) has a height of zero.

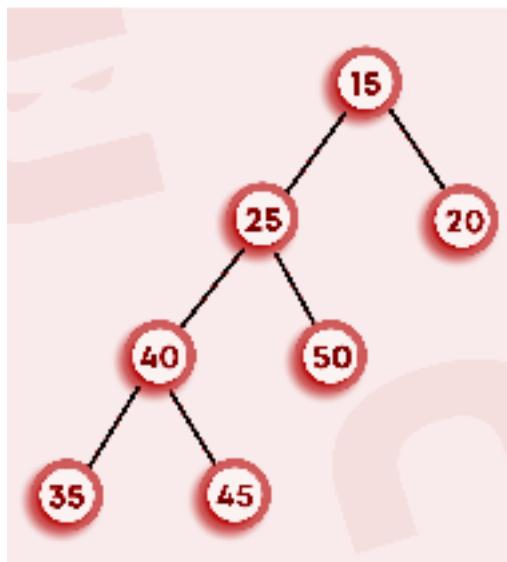
Binary Trees

- A generic tree with at most two child nodes for each parent node is known as a binary tree.
- A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a data element. The **root** pointer is the topmost node in the tree.
- The left and right pointers recursively point to smaller **subtrees** on either side.
- An empty tree is also a valid binary tree.
- *A formal definition is:* A **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.



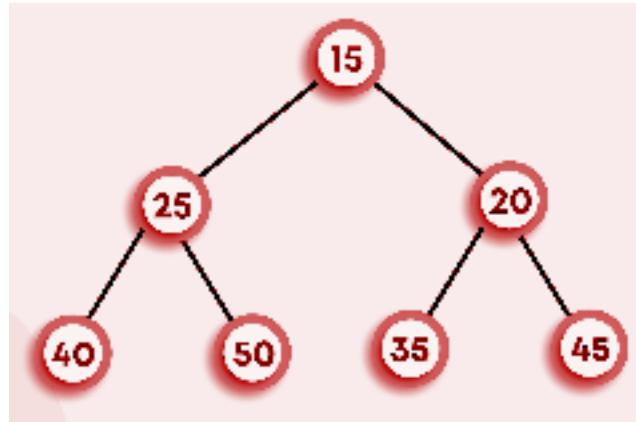
Types of binary trees:

Full binary trees: A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

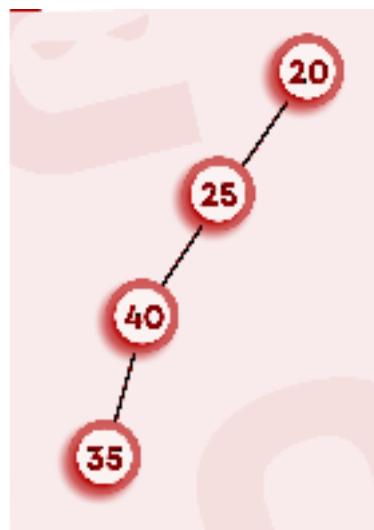


Complete binary tree: A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

Perfect binary tree: A binary tree is termed perfect when all its internal nodes have two children along with the leaf nodes that are at the same level.

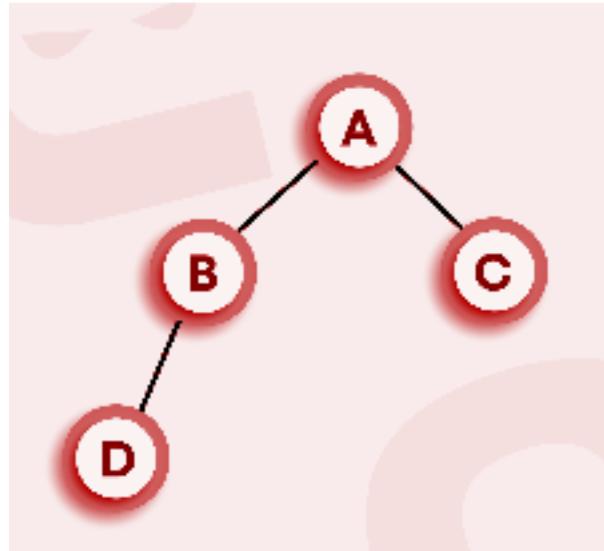


A degenerate tree: In a degenerate tree, each internal node has only one child.



The tree shown above is degenerate. These trees are very similar to linked-lists.

Balanced binary tree: A binary tree in which the difference between the depth of the two subtrees of every node is at most one is called a balanced binary tree.



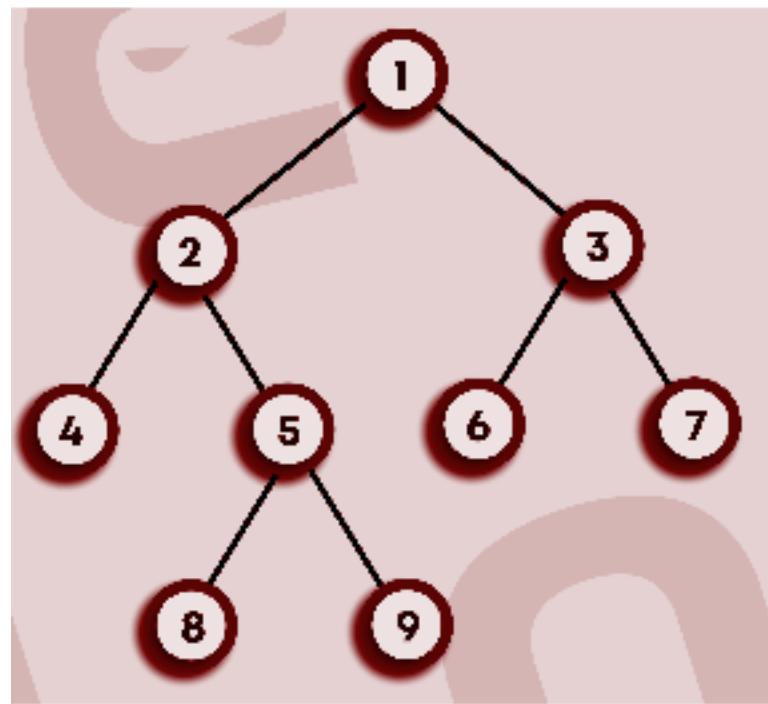
Binary tree representation:

Binary trees can be represented in two ways:

Sequential representation

- This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes.
- The number of nodes in a tree defines the size of the array.
- The root node of the tree is held at the first index in the array.
- In general, if a node is stored at the i^{th} location, then its **left** and **right** child are kept at $(2i)^{\text{th}}$ and $(2i+1)^{\text{th}}$ locations in the array, respectively.

Consider the following binary tree:



The array representation of the above binary tree is as follows:

1	2	3	4	5	6	7			8	9
1	2	3	4	5	6	7	8	9	10	11

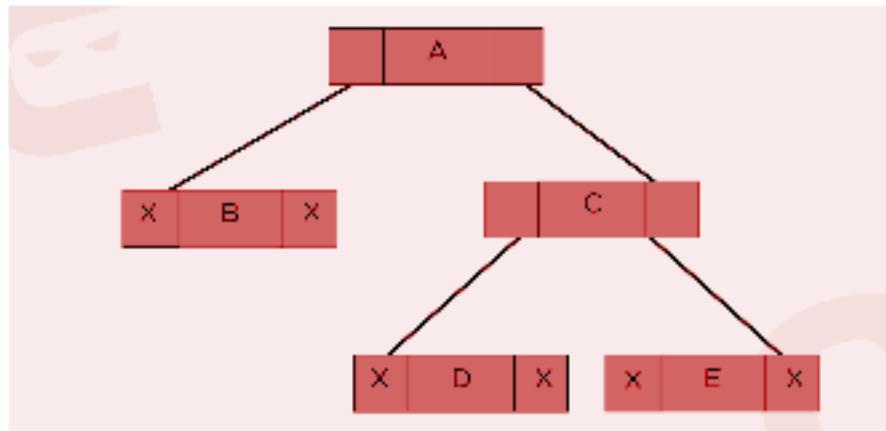
As discussed above, we see that the left and right child of each node is stored at locations **2*(nodePosition)** and **2*(nodePosition)+1**, respectively.

For Example, The location of node 3 in the array is 3. So its left child will be placed at **2*3 = 6**. Its right child will be at the location **2*3 +1 = 7**. As we can see in the array, children of 3, which are 6 and 7, are placed at locations 6 and 7 in the array.

Note: The sequential representation of the tree is not preferred due to the massive amount of memory consumption by the array.

Linked list representation:

In this type of model, a linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree. The following diagram shows a linked list representation for a tree.

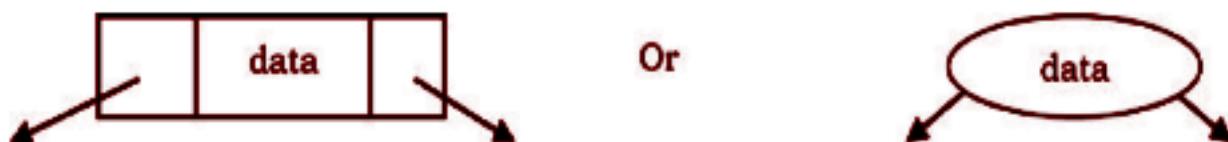


As shown in the above representation, each linked list node has three components:

- Pointer to the left child
- Data
- Pointer to the right child

Note: If there are no children for a given node (leaf node), then the left and right pointers for that node are set to **null**.

Let's now check the implementation of the **Binary tree class**.



```
class BinaryTreeNode<T> {
    T data;           // To store data
    BinaryTreeNode left; // for storing the reference to left pointer
    BinaryTreeNode right; // for storing the reference to right pointer
    // Constructor
    BinaryTreeNode(T data) {
        this.data = data; // Initializes data of the node
    }
}
```

```

        this.left = null; // initializes left and right pointers to null
        this.right = null;
    }
}

```

Operations on Binary Trees

Basic Operations

- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree
- Finding the level which has the maximum sum and many more...

Print Tree Recursively

Let's first write a program to print a binary tree recursively. Follow the comments in the code below:

```

public void printTree(BinaryTreeNode<Integer> root) {
    if (root == null) {           // Base case
        return;
    }
    System.out.print(root.data + ":"); //printing the data at root node
    if (root.left != null) {      // checking if left not null
        System.out.print("L" + root.left.data);
    }

    if (root.right != null) { // checking if right not null
        System.out.print("R" + root.right.data);
    }
    System.out.println();
    printTree(root.left); //Now recursively, call left and right subtrees
    printTree(root.right);
}

```

Input Binary Tree

We will be following the level-wise order for taking input and -1 denotes the **null** pointer.

```

public BinaryTreeNode<Integer> takeInput() {
    System.out.print("Enter data:");
    int rootData = s.nextInt();           // taking data as input
    if (rootData == -1) {                // if the data is -1, means null pointer
        return null;
    }
    // Dynamically create root Node which calls constructor of the same class
    BinaryTreeNode<Integer> root = new BinaryTreeNode<>(rootData);
    // Recursively calling over left subtree
    BinaryTreeNode<Integer> leftChild = takeInput();
    // Recursively calling over right subtree
    BinaryTreeNode<Integer> rightChild = takeInput();
    root.left = leftChild; // now allotting left and right childs to root
    root.right = rightChild;
    return root;
}

```

Count nodes

- Unlike the Generic trees, where we need to traverse the children vector of each node, in binary trees, we just have at most left and right children for each node.
- Here, we just need to recursively call on the right and left subtrees independently with the condition that the node pointer is not null.
- Follow the comments in the upcoming code for better understanding:

```

public int numNodes(BinaryTreeNode<Integer> root) {
    if (root == null) { // Condition to check if the node is not null
        return 0;       // counted as zero if so
    }
    return 1 + numNodes(root.left) + numNodes(root.right);
    //recursive calls on left and right subtrees with addition of 1(for
    //counting current node)
}

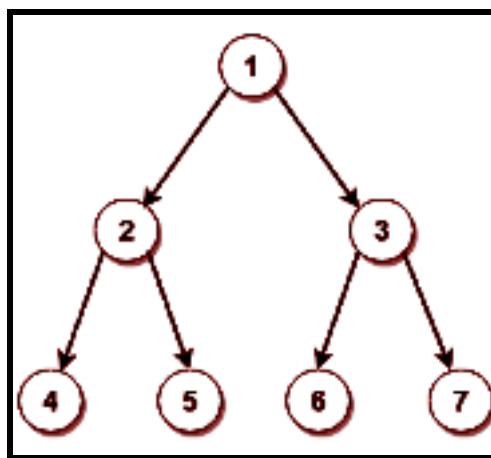
```

Binary tree traversal

Following are the ways to traverse a binary tree and their orders of traversal:

- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- ❖ **Preorder traversal:** 1, 2, 4, 5, 3, 6, 7
- ❖ **Postorder traversal:** 4, 5, 2, 6, 7, 3, 1
- ❖ **Inorder traversal:** 4, 2, 5, 1, 6, 3, 7

Let's look at the code for inorder traversal, below:

```

public void inorder(BinaryTreeNode<Integer> root) {
    if (root == null) { // Base case when node's value is null
        return;
    }
    inorder(root.left); //Recursive call over left part as it needs
                        // to be printed first
    System.out.print(root.data); // Now printed root's data
    inorder(root.right); //Finally recursive call made over right subtree
}
  
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. If you get stuck, refer to the solution tab for the same.

Node with the Largest Data

In a Binary Tree, we must visit every node to figure out the maximum. So the idea is to traverse the given tree and for every node return the maximum of 3 values:

- Node's data.
- Maximum in node's left subtree.
- Maximum in node's right subtree.

Below is the implementation of the above approach.

```
public static int findMaximum(root){  
    # Base case  
    if (root == null):  
        return Integer.MAX_VALUE;  
  
    // Return maximum of 3 values:  
    // 1) Root's data  
    // 2) Max in Left Subtree  
    // 3) Max in right subtree  
    int max = root.data;  
    int lmax = findMaximum(root.left); //Maximum of left subtree  
    int rmax = findMaximum(root.right); //Maximum of right subtree  
    if (lmax > max)  
        max = lmax;  
    if (rmax > max)  
        max = rmax;  
    return max;  
}
```

Construct a binary tree from preorder and inorder traversal

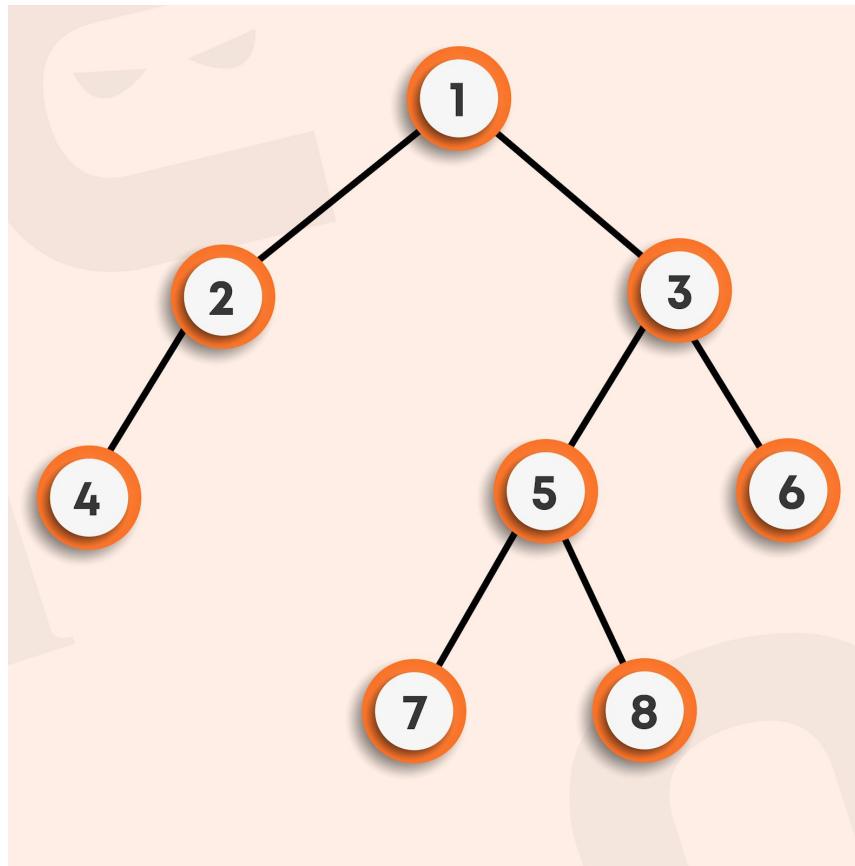
Consider the following example to understand this better.

Input:

Inorder traversal : {4, 2, 1, 7, 5, 8, 3, 6}

Preorder traversal : {1, 2, 4, 3, 5, 7, 8, 6}

Output: Below binary tree...



The idea is to start with the root node, which would be the first item in the preorder sequence and find the boundary of its left and right subtree in the inorder array.

Now all keys before the root node in the inorder array become part of the left

subtree, and all the indices after the root node become part of the right subtree. We repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider below inorder and preorder sequence-

Inorder: {4, 2, 1, 7, 5, 8, 3, 6}

Preorder: {1, 2, 4, 3, 5, 7, 8, 6}

The root will be the first element in the preorder sequence, i.e. 1. Next, we locate the index of the root node in the inorder sequence. Since 1 is the root node, all nodes before 1 must be included in the left subtree, i.e., {4, 2}, and all the nodes after one must be included in the right subtree, i.e. {7, 5, 8, 3, 6}. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

<u>Left subtree:</u>	<u>Right subtree:</u>
Inorder : {4, 2}	Inorder : {7, 5, 8, 3, 6}
Preorder : {2, 4}	Preorder : {3, 5, 7, 8, 6}

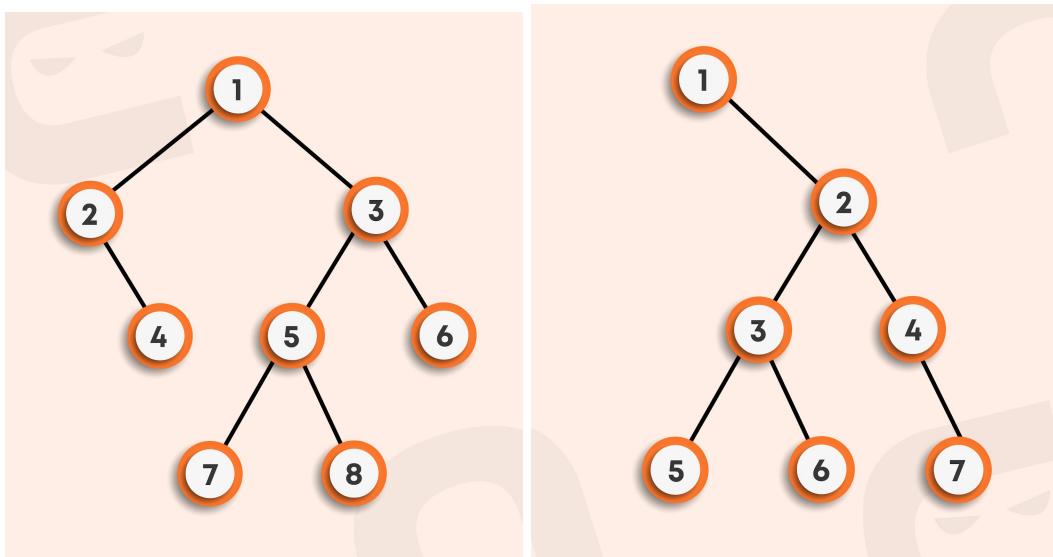
Using the above explanation you can easily create logic and code this. Refer solution tab for solution code for this problem.

Now, try to construct the binary tree when inorder and postorder traversals are given...

The diameter of a binary tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two child nodes. The diameter of the binary tree may pass through the root (not necessary).

For example, the Below figure shows two binary trees having diameters 6 and 5, respectively (nodes highlighted in blue color). The diameter of the binary tree shown on the left side passes through the root node while on the right side, it doesn't.



There are three possible paths of the diameter:

1. The diameter could be the sum of the left height and the right height.
2. It could be the left subtree's diameter.
3. It could be the right subtree's diameter.

We will pick the one with the maximum value.

Now let's check the code for this...

```

public int height(BinaryTreeNode<Integer> root) { //Func for height of tree
    if (root == null) {
        return 0;
    }
    return 1 + Math.max(height(root.left), height(root.right));
}

public int diameter(BinaryTreeNode<Integer> root) { //calculates diameter
    if (root == null) { // Base case
        return 0;
    }

    int option1 = height(root.left) + height(root.right); // Option 1
    int option2 = diameter(root.left); // Option 2
    int option3 = diameter(root->right); // Option 3
    return Math.max(option1, Math.max(option2, option3)); //returns max
}

```

The time complexity for the above approach:

- Height function traverses each node once; hence time complexity will be $O(n)$.
- Option2 and Option3 also traverse on each node, but for each node, we are calculating the height of the tree considering that node as the root node, which makes time complexity equal to $O(n*h)$. (worst case with skewed trees, i.e., a type of binary tree in which all the nodes have only either one child or no child.) Here, h is the height of the tree, which could be $O(n^2)$.

This could be reduced if the height and diameter are obtained simultaneously, which could prevent extra n traversals for each node. To achieve this, move towards the other sections...

The Diameter of a Binary tree: Better Approach

In the previous approach, for each node, we were finding the height and diameter independently, which was increasing the time complexity. In this approach, we will find height and diameter for each node at the same time, i.e., we will store the height and diameter using a pair class where the **first** pointer will be storing the height of that node and the **second** pointer will be storing the diameter. Here, also we will be using recursion.

Let's focus on the **base case**: For a null tree, height and diameter both are equal to 0. Hence, pair class will store both of its values as zero.

Now, moving to **Hypothesis**: We will get the height and diameter for both left and right subtrees, which could be directly used.

Finally, the **induction step**: Using the result of the Hypothesis, we will find the height and diameter of the current node:

Height = max(leftHeight, rightHeight)

Diameter = max(leftHeight + rightHeight, leftDiameter, rightDiameter)

Now we will create a Pair class which will help us do this problem in better complexity.

```
class Pair {
    int first;
    int second;
    public Pair(int first, int second){
        this.first = first;
        this.second = second;
    }
}
```

To access this pair class, we will use **.first** and **.second** pointers.

Follow the code below along with the comments to get a better grip on it...

```

public static Pair heightDiameter(BinaryTreeNode<Integer> root) {
    // pair class return-type function
    if (root == null) {                                // Base case
        Pair p = new Pair(0, 0);
        // p.first = 0;
        // p.second = 0;
        return p;
    }
    // Recursive calls over left and right subtree
    Pair leftAns = heightDiameter(root.left);
    Pair rightAns = heightDiameter(root.right);
    // Hypothesis step
    // Left diameter, Left height
    int ld = leftAns.second;
    int lh = leftAns.first;
    // Right diameter, Right height
    int rd = rightAns.second;
    int rh = rightAns.first;

    // Induction step
    int height = 1 + Math.max(lh, rh);      // height of current root node
    int diameter = Math.max(lh + rh, Math.max(ld, rd)); //diameter of
                                                        // current root node
    Pair p;                                     // Pair class for current root node
    p.first = height;
    p.second = diameter;
    return p;
}

```

Now, talking about the time complexity of this method, it can be observed that we are just traversing each node once while making recursive calls and rest all other operations are performed in constant time, hence the time complexity of this program is $O(n)$, where n is the number of nodes.

Binary Search Trees

Introduction

- These are the specific types of binary trees.
- These are inspired by the binary search algorithm.
- Time complexity on insertion, deletion, and searching reduces significantly as it works on the principle of binary search rather than linear search, as in the case of normal binary trees (will discuss it further).

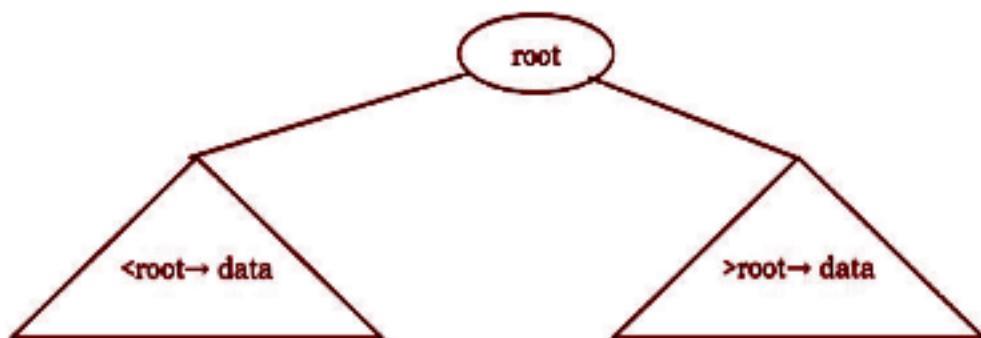
Binary Search Tree Property

In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called

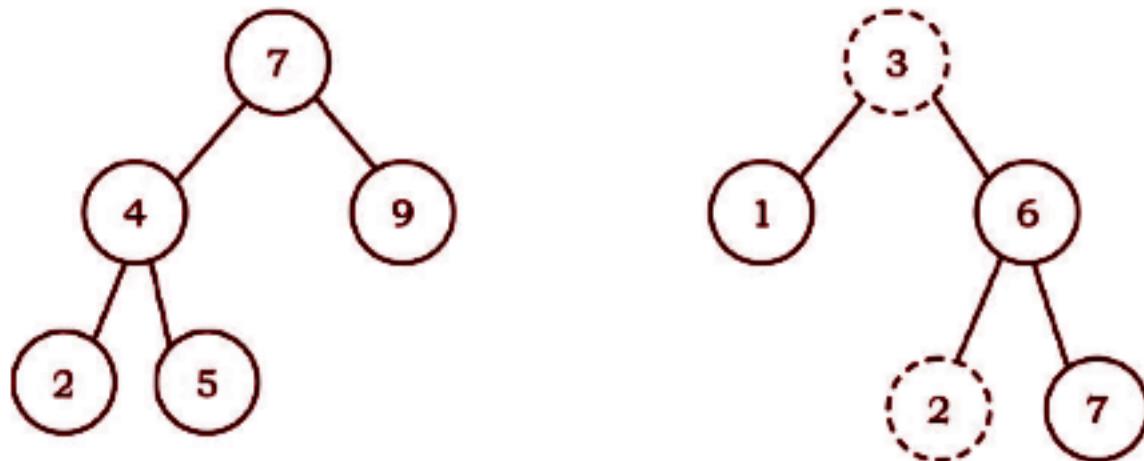
Binary Search Tree property.

- The left subtree of a node ONLY contains nodes with keys less than the node's key.
- The right subtree of a node ONLY contains nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Note: The **BST** property should be satisfied at every node in the tree.



Example: The left tree is a binary search tree and the right tree is not a binary search tree (*This because the BST property is not satisfied at node 6. Its child with key 2, is less than its parent with key 3, which is a violation, as all the nodes on the right subtree of root node 3, must have keys greater than or equal to 3.*)



Store Data in BST

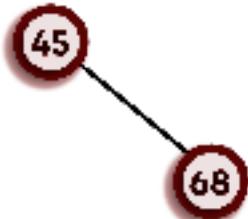
Example: Insert {45, 68, 35, 42, 15, 64, 78} in a BST in the order they are given.

Solution: Follow the steps below:

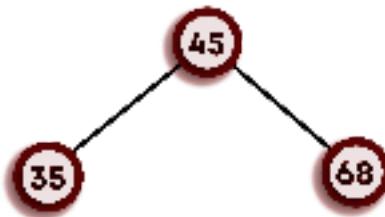
1. Since the tree is empty, so the first node will automatically be the root node.



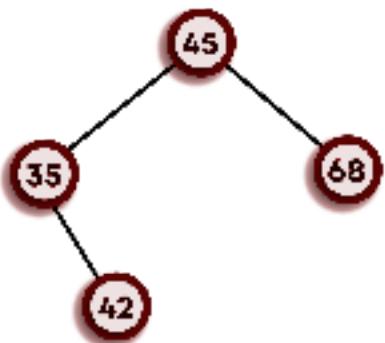
2. Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



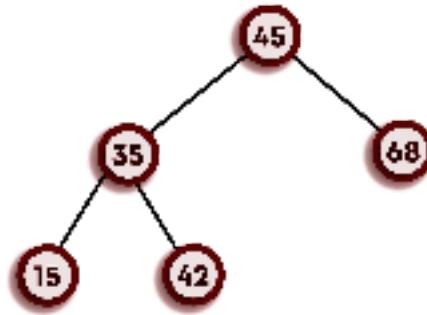
3. To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



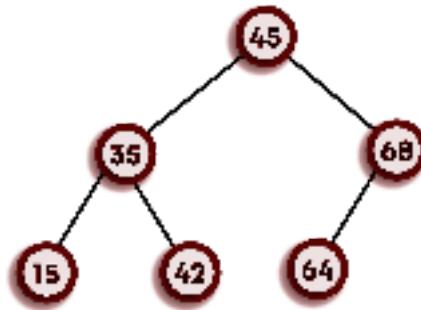
4. Moving on to inserting 42. We can see that $42 < 45$, so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that $42 > 35$ means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



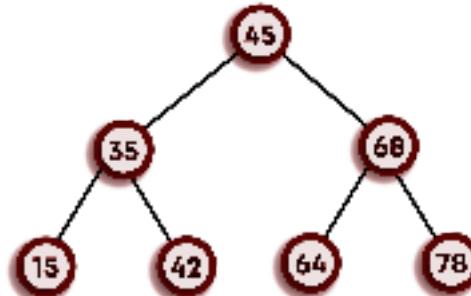
5. Now, to insert 15, we will follow the same approach starting from the root node. Here, $15 < 45$, which means 15 will be a part of the left subtree. As $15 < 35$, we will continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



6. Continuing further, to insert 64, we know that $64 >$ root node's data, but less than 68, hence 64 will be the left child of 68.



7. Finally, we have to insert 78. We can see that $78 > 45$ and $78 > 68$, so 78 will be the right child of 68.



In this way, the data is stored in a BST.

Note:

- If we follow the **inorder traversal** of the final BST, we will get the sorted array.
- As seen above, to insert an element in the BST, we will be traversing till either the left subtree's leaf node or right subtree's leaf node, in the worst-case scenario.
- Hence, the **time complexity of insertion** for each node is **O(log(H))** (where **H** is the height of the tree).
- For inserting **N** nodes, complexity will be **O(N*log(H))**.

Problem Statement: Search in BST

Given a BST and a target value(x), we have to return the binary tree node with data x if it is present in the BST; otherwise, return NULL.

Approach: As the given tree is BST, we can use the **binary search algorithm**. Using recursion will make it easier.

Base Case:

- If the tree is empty, it means that the root node is NULL, then we will simply return NULL as the node is not present.
- Suppose if root's data is equal to **x**, we don't need to traverse forward in this tree as the target value has been found out, so we will simply return the **root** from here.

Small Calculation:

- In the case of BST, we'll only check for the condition of binary search, i.e., if **x** is greater than the root's data, then we will make a recursive call over the right subtree; otherwise, the recursive call will be made on the left subtree.

- This way, we are entirely discarding half the tree to be searched as done in case of a binary search. Therefore, the time complexity of searching is $O(\log(H))$ (where H is the height of BST).

Recursive call: After figuring out which way to move, we can make recursive calls on either left or right subtree. This way, we will be able to search the given element in a BST.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement: Print elements in a range

Given a BST and a range (L, R), we need to figure out all the elements of BST that are present in the given range inclusive of L and R.

Approach: We will be using recursion and binary searching for the same.

Base case: If the root is NULL, it means we don't have any tree to check upon, and we can simply return.

Small Calculation: There are three conditions to be checked upon:

- If the root's data lies in the given range, then we can print it.
- We will compare the root's data with the given range's maximum. If root's data is smaller than R, then we will have to traverse only the right subtree.
- Now, we will compare the root's data with the given range's minimum. If the root's data is greater than L, then we will traverse only the left subtree.

Recursive call: Recursive call will be made as per the small calculation part onto the left and right subtrees. In this way, we will be able to figure out all the elements in the range.

Note: Try to code this yourself, and refer to the solution tab in case of any doubts.

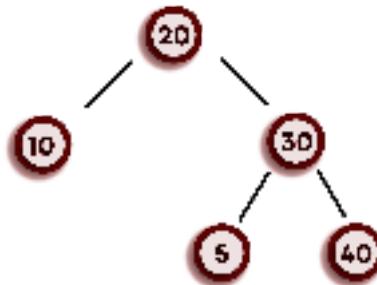
Problem Statement: Check BST

Given a binary tree, we have to check if it is a BST or not.

Approach: We will simply traverse the binary tree and check if the nodes satisfy the BST Property. Thus we will check the following cases:

- If the node's value is greater than the value of the node on its left.
- If the node's value is smaller than the value of the node on its right.

Important Case: Don't just compare the direct left and right children of the node; instead, we need to compare every node in the left and right subtree with the node's value. Consider the following case:



- Here, it can be seen that for root, the left subtree is a BST, and the right subtree is also a BST (individually).
- But the complete tree is not a BST. This is because a node with value 5 lies on the right side of the root node with value 20, whereas it should be on the left side of the root node.
- Hence, even though the individual subtrees are BSTs, it is also possible that the complete binary tree is not a BST. Hence, this third condition must also be checked.

To check over this condition, we will keep track of the minimum and maximum values of the right and left subtrees correspondingly, and at last, we will simply compare them with root.

- The left subtree's maximum value should be less than the root's data.
- The right subtree's minimum value should be greater than the root's data.

Now, let's look at the Python code for this approach using this approach.

```

public int maximum(BinaryTreeNode<Integer> root) {
    if (root == null) {           // If root is NULL, then we simply return
        return Integer.MIN_VALUE;           // -∞ (negative infinity)
    }
    // Otherwise returning maximum of left/right subtree and root's data
    int left = maximum(root.left);
    int right= maximum(root.right);
    return Math.max(root.data, Math.max(left, right));
}

public int minimum(BinaryTreeNode<Integer> root) {
    if (root == null) {           // If root is NULL, then we simply return
        return Integer.MAX_VALUE;           // +∞ (positive infinity)
    }
    // Otherwise returning minimum of left/right subtree and root's data
    int left = maximum(root.left);
    int right= maximum(root.right);
    return Math.min(root.data, Math.min(left, right));
}

public boolean isBST(BinaryTreeNode<Integer> root) {
    if (root == null) {           // Base case
        return true;
    }

    int leftMax = maximum(root.left);   // Figuring out left's maximum
    int rightMin = minimum(root.right); // Figuring out right's minimum
    boolean output = (root.data > leftMax) && (root.data <= rightMin) &&
                    isBST(root.left) && isBST(root.right);
    //Checked the conditions discussed above
    return output;
}

```

Time Complexity: In the `isBST()` function, we are traversing each node, and for each node, we are then calculating the minimum and maximum value by again

traversing that complete subtree's height. Hence, if there are **N** nodes in total and the height of the tree is **H**, then the time complexity will be **O(N*H)**.

Improved Solution for Check BST

- To improve our solution, observe that for each node, the minimum and maximum values are being calculated separately.
- We now wish to calculate these values, while checking the **isBST** condition itself, to get rid of another cycle of iterations.
- We will follow a similar approach as that of the diameter calculation of binary trees.
- At each stage, we will return the maximum value, minimum value, and the BST status (True/False) for each node of the tree, in the form of a tuple.

Let's look at its implementation now:

```

class IsBSTReturn {      // Class to store data for each node of tree
    boolean isBST;
    int minimum;
    int maximum;
}

-----
public IsBSTReturn isBST2(BinaryTreeNode<Integer> root) {
    if (root == null) {           // Base Case
        IsBSTReturn output = new IsBSTReturn(); //Object created
        output.isBST = true;             // Empty tree is a BST
        output.minimum = INT_MAX;
        output.maximum = INT_MIN;
        return output;
    }
    IsBSTReturn leftOutput = isBST2(root.left); // Left subtree Recursion
    IsBSTReturn rightOutput = isBST2(root.right); //Right subtree
                                                //Recursive call

    // Small Calculation
    // Minimum and maximum values figured out side-by-side preventing
    // extra traversals
    int minimum = Math.min(root.data, Math.min(leftOutput.minimum,
                                                rightOutput.minimum));
    int maximum = max(root.data, max(leftOutput.maximum,
                                    rightOutput.maximum));
}

```

```

// Checking out for the subtree if it's a BST or not
bool isBSTFinal = (root.data > leftOutput.maximum) && (root.data <=
rightOutput.minimum) && leftOutput.isBST &&
rightOutput.isBST;

// Assigning values to the output class object
IsBSTReturn output;
output.minimum = minimum;
output.maximum = maximum;
output.isBST = isBSTFinal;
return output;
}
  
```

Time Complexity: Here, we are going to each node and doing a constant amount of work. Hence, the time complexity for **N** nodes will be of **O(N)**.

Another Improved Solution for Check BST

The time complexity for this problem can't be improved further, but there is a better approach to this problem, which makes our code look more robust. Let's discuss that approach now.

Approach: We will be checking on the left subtree, right subtree, and combined tree without returning a tuple of maximum and minimum values. We will be using the concept of default arguments over here. Check the code below:

```

// This time function is using default arguments for storing minimum and
// maximum value for each node
private bool isBST3Helper(BinaryTreeNode<Integer> root, int min, int max) {
    if (root == null) {           // Base case: Empty tree
        return true;
    }
    // checking out the special condition first and returning false if not
    // satisfied
    if (root->data < min || root->data > max) {
        return false;
    }
    // Checking out left and right subtrees
    bool isLeftOk = isBST3(root.left, min, root.data - 1);
    bool isRightOk = isBST3(root.right, root.data, max);
    // Returning true if both are BST and false otherwise.
    return isLeftOk && isRightOk;
}
  
```

```
public bool isBST3(BinaryTreeNode<Integer> root) {  
    return isBST3Helper(root, Integer.MIN_VALUE, Integer.MAX_VALUE);  
}
```

Time Complexity: Here also, we are just traversing each node and doing constant work on each of them; hence time complexity remains the same, i.e. **O(n)**.

Problem Statement: Construct BST from sorted array

Given a sorted array, we have to construct a BST out of it.

Approach:

- Suppose we take the first element as the root node, then the tree will be skewed as the array is sorted.
- To get a balanced tree (so that searching and other operations can be performed in **O(log(n))** time), we will be using the binary search technique.
- Figure out the middle element and mark it as the root.
- This is done so that the tree can be divided into almost 2 equal parts, as half the elements which will be greater than the root, will form the right subtree (These elements are present to its right).
- The elements in the other half, which are less than the root, will form the left subtree (These elements are present to its left).
- Just put root's left child to be the recursive call made on the left portion of the array and root's right child to be the recursive call made on the right portion of the array.
- Try it yourself and refer to the solution tab for code.

BST to Sorted LL

Problem statement: Given a BST, we have to construct a sorted linked list out of it.

Approach: As discussed earlier, the inorder traversal of the BST, provides elements in a sorted fashion, so while creating the linked list, we will be traversing the tree in inorder style.

- **Base case:** If the root is NULL, it means the head of the linked list is NULL; hence we return NULL.
- **Small calculation:** Left subtree will provide us the head of LL, and the right subtree will provide the tail of LL; hence root node will be placed after the LL obtained from the left subtree and right LL will be connected after the root.

Left subtree's LL	Root	Right subtree's LL
-------------------	------	--------------------

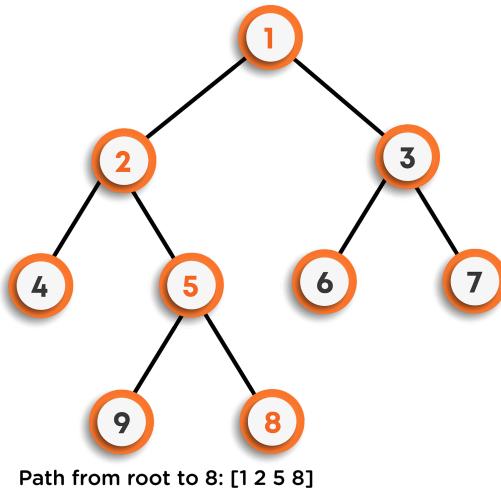
- **Recursive call:** Simply call the left subtree, connect the root node to the end of it, and then connect the right subtree's recursive call after root.

Try it yourselves, and for code, refer to the solution tab of the corresponding question.

Root to node path in a Binary Tree

Problem statement: Given a Binary tree, we have to return the path of the root node to the given node.

For Example: Refer to the image below...



Approach:

1. Start from the root node and compare it with the target value. If matched, then simply return, otherwise, recursively call over left and right subtrees.
2. Continue this process until the target node is found and return if found.
3. While returning, you need to store all the nodes that you have traversed on the path from the root to the target node in a vector.
4. Now, in the end, you will be having your solution vector.

Now try to create logic and code the same problem with BST instead of a binary tree.

BST Class

Here, we will be creating our own BST class to perform operations like insertion, deletion, etc...

Kindly, follow the code below:

```

class BST {
    BinaryTreeNode<Integer> root;      // root node

    BST() {                         // Constructor to initialize root to NULL
        root = null;
    }

    public boolean hasData(int data, BinaryTreeNode<Integer> node) {
        if (node == null) {           // the presence of a
            return false;             // node in BST
        }

        if (node->data == data) {
            return true;
        } else if (data < node->data) {
            return hasData(data, node->left);
        } else {
            return hasData(data, node->right);
        }
    }

    public boolean hasData(int data) {
        return hasData(data, root); // from here the value is returned
    }
}

```

You can observe that `hasData()` function has been declared under the private section to prevent it from being manipulated, presenting the concept of data abstraction.

Try insertion and deletion on your own, and in case of any difficulty, refer to the hints and solution code below...

Insertion in BST:

We are given the root of the tree and the data to be inserted. Follow the same approach to insert the data as discussed above using Binary search algorithm. Check the code below for insertion:

```

public BinaryTreeNode<Integer> insert(int data, BinaryTreeNode<Integer> node){
    // Using Binary Search algorithm
    if (node == null) {
        BinaryTreeNode<Integer> newNode = new BinaryTreeNode<>(data);
        return newNode;
    }

    if (data < node.data) {
        node.left = insert(data, node.left);
    } else {
        node.right = insert(data, node.right);
    }
    return node;
}

public void insert(int data) {      // Insertion function
    this.root = insert(data, this.root);
}

```

Deletion in BST:

Recursively, find the node to be deleted.

- **Case 1:** If the node to be deleted is the leaf node, then simply delete that node with no further changes and return NULL.
- **Case 2:** If the node to be deleted has only one child, then delete that node and return the child node.
- **Case 3:** If the node to be deleted has both the child nodes, then we have to delete the node such that the properties of BST remain unchanged. For this, we will replace the node's data with either the left child's largest node or right child's smallest node and then simply delete the replaced node.

Now, let's look at the code below:

```

BinaryTreeNode<Integer> deleteData(int data, BinaryTreeNode<Integer> node){
    if (node == NULL) {      // Base case
        return NULL;
    }
    // Finding that node by traversing the tree

```

```

        if (data > node.data) {
            node.right = deleteData(data, node.right);
            return node;
        } else if (data < node.data) {
            node.left = deleteData(data, node.left);
            return node;
        } else {          //found the node
            if (node.left == NULL && node.right == NULL) { //Leaf node
                delete node;
                return NULL;
            } else if (node.left == NULL) { //node having only left child
                BinaryTreeNode<Integer> temp = node.right;
                node.right = NULL;
                delete node;
                return temp;
            } else if (node.right == NULL) { //node having only right child
                BinaryTreeNode<Integer> temp = node.left;
                node.left = NULL;
                delete node;
                return temp;
            } else {      //node having both the childs
                BinaryTreeNode<Integer> minNode = node.right;
                while (minNode.left != NULL) { // replacing node with
                    minNode = minNode.left; // right subtree's min
                }
                int rightMin = minNode.data;
                node.data = rightMin;
                // now simply deleting that replaced node using recursion
                node.right = deleteData(rightMin, node.right);
                return node;
            }
        }
    }

public void deleteData(int data) {          // Function to delete
    root = deleteData(data, root);
}

```

Hashmaps

Introduction to Hashmaps

Suppose we are given a string or a character array and asked to find the maximum occurring character. It could be quickly done using arrays. We can simply create an array of size 256, initialize this array to zero, and then, simply traverse the array to increase the count of each character against its ASCII value in the frequency array. In this way, we will be able to figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values. For this, we will be using a different data structure called hashmaps.

In hashmaps, the data is stored in the form of keys against which some value is assigned. Keys and values don't need to be of the same data type.

If we consider the above example in which we were trying to find the maximum occurring string, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example: The given string array is:

```
str[] = {"abc", "def", "ab", "abc", "def", "abc"}
```

Hashmap will look like as follows:

Key (datatype = string)	Value (datatype = int)
"abc"	3
"def"	2
"ab"	1

From here, we can directly check for the frequency of each string and hence figure out the most frequent string among them all.

Note: One more limitation of arrays is that the indices could only be whole numbers but this limitation does not hold for hashmaps.

The values are stored corresponding to their respective keys and can be invoked using these keys. To insert, we can do the following:

- `hashmap.put(key, value)`

The functions that are required for the hashmaps are(using templates):

- **insert(k key, v value):** To insert the value of type v against the key of type k.
- **get(k key):** To get the value stored against the key of type k.
- **remove(k key):** To delete the key of type k, and hence the value corresponding to it.

To implement the hashmaps, we can use the following data structures:

1. **Linked Lists:** To perform insertion, deletion, and search operations in the linked list, the time complexity will be $O(n)$ for each as:
 - For insertion, we will first have to check if the key already exists or not, if it exists, then we just have to update the value stored corresponding to that key.
 - For search and deletion, we will be traversing the length of the linked list.

2. **BST:** We will be using some kind of a balanced BST so that the height remains of the order $O(\log N)$. For using a BST, we will need some sort of comparison between the keys. In the case of strings, we can do the same. Hence, insertion, search, and deletion operations are directly proportional to the height of the BST. Thus the time complexity reduces to $O(\log N)$ for each.
3. **Hash table:** Using a hash table, the time complexity of insertion, deletion, and search operations, could be improved to $O(1)$ (same as that of arrays). We will study this in further sections.

Inbuilt Hashmap

In Java, we have two types of hashmaps:

- **Treemap** (uses BST implementation)
- **HashMap** (uses hash table implementation)

Note: Both have similar functions and similar ways to use, differing only in the time complexities. The time complexity of each of the operations(insertion, deletion, and searching) in the **Treemap** is $O(\log N)$, while in the case of **HashMap**, they are $O(1)$.

HashMap:

- Import hashmap library

```
import java.util.HashMap;
```

- Syntax to declare:

```
HashMap<datatype_for_keys, datatype_for_values> name;
```

- Operations performed:

1. **Insertion:** Suppose, we want to insert the string “abc” with the value 1 in the hashmap named *ourmap*, there are two ways to do so:

- Simply create a pair of both and insert in the map using **.insert** function. Syntax:

```
HashMap<String, Integer> ourmap = new HashMap<>();
```

```
ourmap.put("abc", 1);
```

2. Searching: Suppose we want to find the value stored in the hashmap against key "abc", there are two ways to do so:

- As did in insertion like arrays, the same way we can figure out the value stored against the corresponding key. Syntax:

```
int value = ourmap.get("abc");
```

Note: If we try to access a key that is not present in the unordered_map, then there are two different outcomes:

- If we are accessing the value using .get() function, then we will get an error specifying that we are trying to access the value that is not present in the map.

But what if we want to check if the key is present or not on the map? For that, we will be using the .count() function, which tells if the key is present in the map or not. It returns false, if not present, and true, if present

Syntax:

```
boolean isPresent = ourmap.containsKey("ghi");
```

Note: We can also check the size of the map by using .size() function, which returns the number of key-value pairs present on the map.

Syntax:

```
int size_of_map = ourmap.size();
```

3. Deletion: Suppose we want to delete the key "abc" from the map, we will be using .remove() function.

Syntax:

```
ourmap.remove("abc");
```

Remove Duplicates

Problem statement: Given an array of integers, we need to remove the duplicate values from that array, and the values should be in the same order as present in the array.

For example: Suppose the given array is arr = {1, 3, 6, 2, 4, 1, 4, 2, 3, 2, 4, 6}, answer should be {1, 3, 6, 2, 4}.

Approach: We will add unique values to the vector and then return it. To check for unique values, start traversing the array, and for each array element, check if the value is already present in the map or not. If not, then we will insert that value in the vector and update the map; otherwise, we will proceed to the next index of the array without making any changes.

Let's look at the code for better understanding.

```
public ArrayList<Integer> removeDuplicates(ArrayList<Integer> a, int size){
    // to store the unique elements.
    ArrayList<Integer> output = new ArrayList<>();

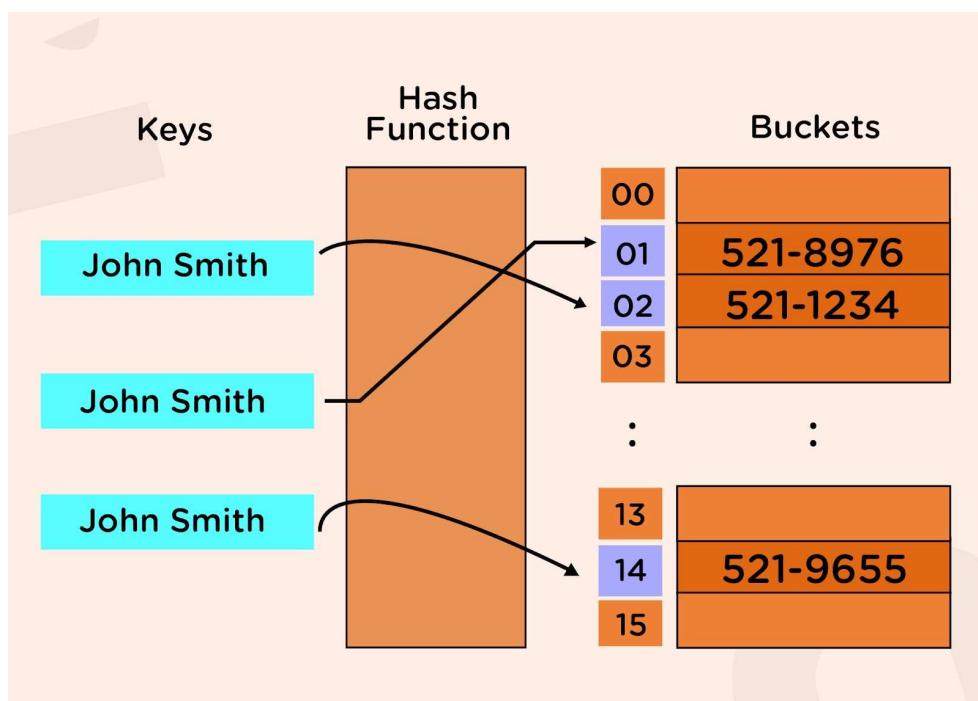
    HashMap<Integer, Boolean> seen = new HashMap<>();
    for (int i=0; i<size; i++) {    // traversing the array
        if (seen.containsKey(a[i])) // using .containsKey() function to
            continue;           //check if the value has already occurred.
        else {
            seen.put(a[i], true); // If not, then updating the map
            output.add(a[i]);   // and inserting in map.
        }
    }
    return output;
}
```

Bucket Array and Hash function

Now, let's see how to perform insertion, deletion, and search operations using hash tables. Till now, we have seen that arrays are the fastest way to extract data as compared to other data structures as the time complexity of accessing the data in the array is $O(1)$. So we will try to use them in implementing the hashmap.

Now, we want to store the key-value pairs in an array, named as a **bucket array**. We need an integer corresponding to the key so that we can keep it in the bucket array. To do so, we use a **hash function**. A hash function converts the key into an integer, which acts as the index for storing the key in the array.

For example: Suppose, we want to store some names from the contact list in the hash table, check out the following the image:



Suppose we want to store a string in a hash table, and after passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as 10593, as this index does not exist in the array of size 20.

To overcome this problem, we will divide the hashmap into two parts:

- Hash code
- Compression function

The first step to store a value into the bucket array is to convert the key into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using hashCode. For different types of keys, we will be having different kinds of hash codes. Now we will pass this value through the compression function, which will convert that value within the range of our bucket array's size.

Now, we can directly store that key against the index obtained after passing through the compression function.

The compression function can be used as (% bucket_size).

One example of a hash code could be: (Example input: "abcd")

"abcd" = ('a' * p³) + ('b' * p²) + ('c' * p¹) + ('d' * p⁰)
Where p is generally taken as a prime number so that they are well distributed.

But, there is still a possibility that after passing the key through from hash code, when we give the same through the compression function, we can get the same values of indices. For example, let s1 = "ab" and s2 = "cd". Now using the above hash function for p = 2, h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes through the compression function, we will get:

Compression_function1 = 292 % 2 = 0

Compression_function2 = 298 % 2 = 0

This means they both lead to the same index 0.

This is known as a **collision**.

Collision Handling

We can handle collisions in two ways:

- Closed hashing (or closed addressing)
- Open addressing

In closed hashing, each entry of the array will be a linked list. This means it should be able to store every value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add the new element at the end of that linked list. This is also known as **separate chaining**.

On the other hand, in open addressing, we will check for the index in the bucket array if it is empty or not. If it is empty, then we will directly insert the key-value pair over that index. If not, then we will find an alternate position for the same. To find the alternate position, we can use the following:

$$h_i(a) = hf(a) + f(i)$$

Where $hf(a)$ is the original hash function, and $f(i)$ is the i -th try over the hash function to obtain the final position $h_i(a)$.

To figure out this $f(i)$, following are some of the techniques:

1. **Linear probing:** In this method, we will linearly probe to the next slot until we find the empty index. Here, $f(i) = i$.

2. **Quadratic probing:** As the name suggests, we will look for alternate i^2 positions ahead of the filled ones, i.e., $f(i) = i^2$.
3. **Double hashing:** According to this method, $f(i) = i * H(a)$, where $H(a)$ is some other hash function.

In practice, we generally prefer to use separate chaining over open addressing, as it is easier to implement and is also more efficient.

Let's now implement the hashmap of our own.

Hashmap implementation - Insert

As discussed earlier, we will be implementing separate chaining. We will be using value as a template and key as a string as we are required to find the hash code for the key. Taking key as a template will make it difficult to convert it using hash code.

Let's look at the code for the same.

```
public class MapNode<K, V> {
    K key;
    V value;
    MapNode<K, V> next;

    public MapNode(K key, V value) {
        this.key = key;
        this.value = value;
    }
}

-----
class Map <K, V> {
    ArrayList<MapNode<K, V>> buckets;
    int size;
    int numBuckets;
    public Map() {
        numBuckets = 5;
        buckets = new ArrayList<>();
        for (int i = 0; i < numBuckets; i++) {
            buckets.add(null);
        }
    }
}
```

```

private int getBucketIndex(K key) {
    int hashCode = key.hashCode();
    return hashCode % numBuckets;
}

public void insert(K key, V value) {
    int bucketIndex = getBucketIndex(key);
    MapNode<K, V> head = buckets.get(bucketIndex);
    while (head != null) {
        if (head.key.equals(key)) {
            head.value = value;
            return;
        }
        head = head.next;
    }
    head = buckets.get(bucketIndex);
    MapNode<K, V> newElementNode = new MapNode<K, V>(key, value);
    size++;
    newElementNode.next = head;
    buckets.set(bucketIndex, newElementNode);
    double loadFactor = (1.0 * size) / numBuckets;
    if (loadFactor > 0.7) {
        rehash();
    }
}
}

```

Hashmap implementation - Delete and search

Refer to the code below and follow the comments in it.

```

public V removeKey(K key) {
    int bucketIndex = getBucketIndex(key);
    MapNode<K, V> head = buckets.get(bucketIndex);
    MapNode<K, V> prev = null;
    while (head != null) {
        if (head.key.equals(key)) {
            size--;
            if (prev == null) {
                buckets.set(bucketIndex, head.next);
            } else {
                prev.next = head.next;
            }
            return head.value;
        }
        prev = head;
    }
}

```

```

        head = head.next;
    }
    return null;
}

public V getValue(K key) {
    int bucketIndex = getBucketIndex(key);
    MapNode<K, V> head = buckets.get(bucketIndex);
    while (head != null) {
        if (head.key.equals(key)) {
            return head.value;
        }
        head = head.next;
    }
    return null;
}

```

Time Complexity and Load Factor

Let's define specific terms before moving forward:

1. n = Number of entries in our map.
2. l = length of the word (in case of strings)
3. b = number of buckets. On average, each box contains (n/b) entries. This is known as **load factor** means b boxes contain n entries. We also need to ensure that the load factor is always less than 0.7, i.e.,

$(n/b) < 0.7$, this will ensure that each bucket does not contain too many entries in it.

4. To make sure that load factor < 0.7 , we can't reduce the number of entries, but we can increase the bucket size comparatively to maintain the ratio. This process is known as **Rehashing**.

This ensures that time complexity is on an average O(1) for insertion, deletion, and search operations each.

Rehashing

Now, we will try to implement the rehashing in our map. After inserting each element into the map, we will check the load factor. If the load factor's value is greater than 0.7, then we will rehash.

Refer to the code below for better understanding.

```
public int size() {
    return size;
}

public double loadFactor() {
    return (1.0 * size)/numBuckets;
}

private void rehash() {
    System.out.println("Rehashing: buckets"+ numBuckets+" size " + size);
    ArrayList<MapNode<K, V>> temp = buckets;
    buckets = new ArrayList<>();
    for (int i = 0; i < 2*numBuckets; i++) {
        buckets.add(null);
    }
    size = 0;
    numBuckets *= 2;
    for (int i = 0; i < temp.size(); i++) {
        MapNode<K, V> head = temp.get(i);
        while (head != null) {
            K key = head.key;
            V value = head.value;
            insert(key, value);
            head = head.next;
        }
    }
}
```

Note: While solving the problems, use the in-built hashmap only.

Priority Queues

Introduction

- Priority Queues are abstract data structures where each data/value in the queue has a certain priority.
- A priority queue is a special type of queue in which each element is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.
- **For example**, the element with the highest value is considered as the highest priority element. However, in some cases, we may assume the element with the lowest value to be the highest priority element. In other cases, we can set priorities according to our needs.

Difference between Priority Queue and Normal Queue

In a queue, the **First-In-First-Out(FIFO)** rule is implemented whereas, in a priority queue, the values are removed based on priority. The element with the highest priority is removed first.

Main Priority Queues Operations

- **Insert (key, data):** Inserts data with a key to the priority queue. Elements are ordered based on key.
- **DeleteMin/DeleteMax:** Remove and return the element with the smallest/largest key.
- **GetMinimum/GetMaximum:** Return the element with the smallest/largest key without deleting it.

Auxiliary Priority Queues Operations

- **kth - Smallest/kth – Largest:** Returns the kth -Smallest/kth –Largest key in the priority queue.
- **Size:** Returns the number of elements in the priority queue.
- **Heap Sort:** Sorts the elements in the priority queue based on priority (key).

Priority Queue Applications

Priority queues have many applications - a few of them are listed below:

- **Data compression:** Huffman Coding algorithm
- **Shortest path algorithms:** Dijkstra's algorithm
- **Minimum spanning tree algorithms:** Prim's algorithm
- **Event-driven simulation:** Customers in a line
- **Selection problem:** Finding the kth- smallest element

Priority Queue Implementations

Before discussing the actual implementation, let us enumerate the possible options.

Unordered Array Implementation

- Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are performed by searching the key and then deleting.
- Insertions complexity: **O(1).**
- DeleteMin complexity: **O(n)**

Unordered List Implementation

- It is very similar to array implementation, but instead of using arrays, linked lists are used.
- Insertions complexity: **O(1).**
- DeleteMin complexity: **O(n).**

Ordered Array Implementation

- Elements are inserted into the array in sorted order based on the key field.
Deletions are performed at only one end.
- Insertions complexity: **O(n)**; DeleteMin complexity: **O(1)**.

Ordered List Implementation

- Elements are inserted into the list in sorted order based on the key field.
Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.
- Insertions complexity: **O(n)**; DeleteMin complexity: **O(1)**.

Binary Search Trees Implementation

- Both insertions and deletions take **O(log(n))** on average if insertions are random (refer to Trees chapter).

Balanced Binary Search Trees Implementation

- Both insertions and deletion take **O(log(n))** in the worst case (refer to Trees chapter).

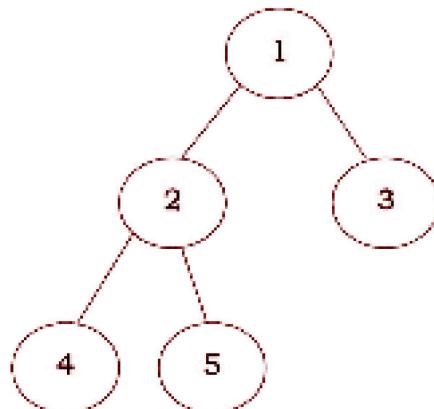
Binary Heap Implementation

In subsequent sections, we will discuss this in full detail.

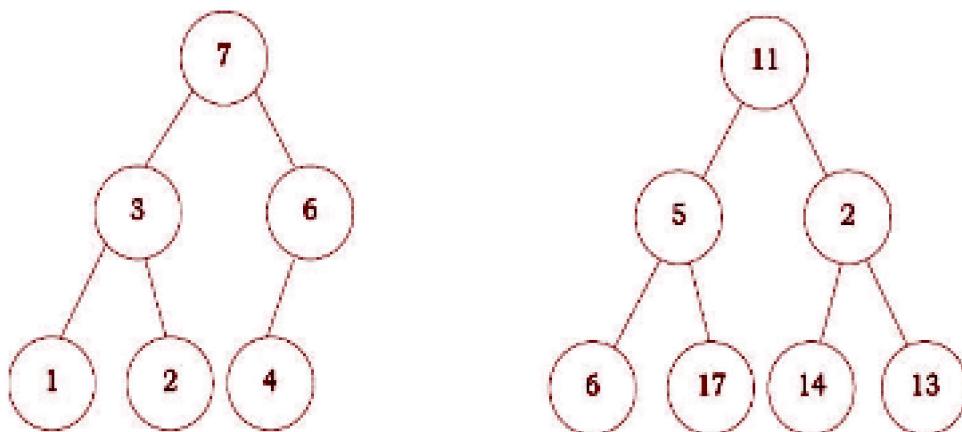
Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	logn (average)	logn (average)	logn (average)
Balanced Binary Search Trees	logn	logn	logn
Binary Heaps	logn	logn	1

Heaps

- A heap is a tree with some special properties.
- The basic requirement of a heap is that the value of a node must be \geq (or \leq) than the values of its children. This is called the **heap property**.
- A heap also has the additional property that all leaf nodes should be at **h** or **$h - 1$** level (where h is the height of the tree) for some $h > 0$ (complete binary trees).
- That means the heap should form a complete binary tree (as shown below).



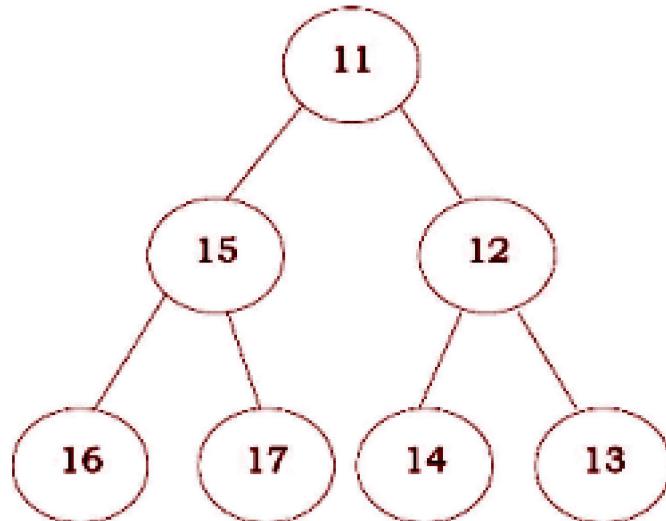
In the examples below, the left tree is a heap (each element is greater than its children) and the right tree is not a heap (since 11 is greater than 2).



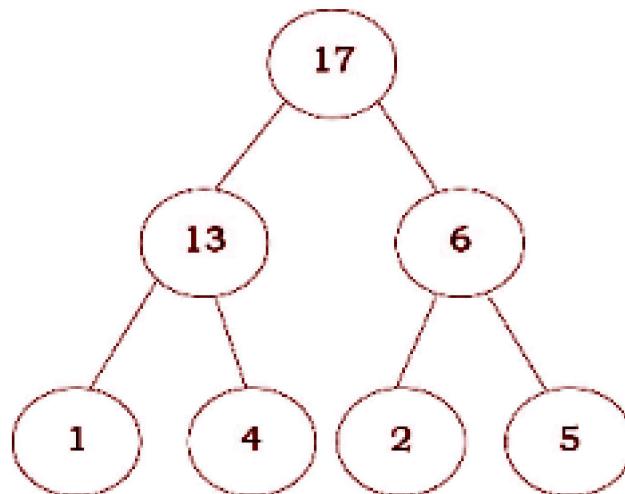
Types of Heaps

Based on the property of a heap we can classify heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children.



- **Max heap:** The value of a node must be greater than or equal to the values of its children.



Binary Heaps

- In a binary heap, each node may have up to two children.
- In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for the remaining discussion.

Representing Heaps: Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the discussion below let us assume that elements are stored in arrays, which starts at index 0. The previous max heap can be represented as:

17	13	6	1	4	2	5
0	1	2	3	4	5	6

Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

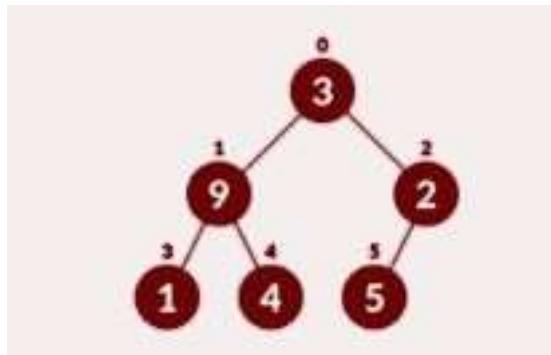
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

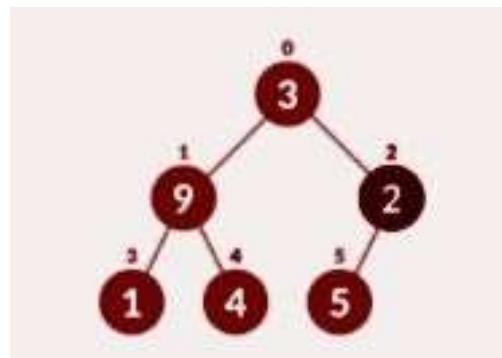
- Let the input array be

3	9	2	1	4	5
0	1	2	3	4	5

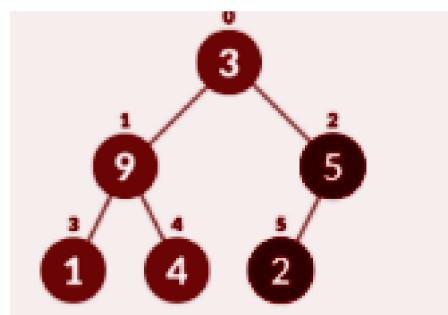
- Create a complete binary tree from the array



- Start from first index of the non-leaf node whose index is given by $n/2 - 1$.



- Set current element **i** as **largest**.
- The index of the left child is given by $2i + 1$ and the right child is given by $2i + 2$.
- If **leftChild** is greater than **currentElement** (i.e. element at the **i**th index), set **leftChildIndex** as largest. #Condition1



- If **rightChild** is greater than element in **largest**, set **rightChildIndex** as **largest**. #Condition2

- Swap **largest** with **currentElement**. #Condition3
- Repeat steps 3-7 until the subtrees are also heapified.
- For Min-Heap, both **leftChild** and **rightChild** must be smaller than the parent for all nodes.

Java Code

```

public class Priority_Queue {

    private ArrayList<Integer> heap;

    public Priority_Queue() {
        heap = new ArrayList<>();
    }

    boolean isEmpty(){
        return heap.size() == 0;
    }

    int size(){
        return heap.size();
    }

    int getMax() throws PriorityQueueException{
        if(isEmpty()){
            // Throw an exception
            throw new PriorityQueueException();
        }
        return heap.get(0);
    }

    public void heapify(int i) {
        int largest = i;
        int l = 2 * i + 1; //Index of Left Child
        int r = 2 * i + 2; //Index of Right Child

        if (l < n && arr[i] < arr[l]) #Condition1
    }
}
  
```

```

        largest = 1;
if (r < n && arr[largest] < arr[r]) #Condition2
    largest = r;

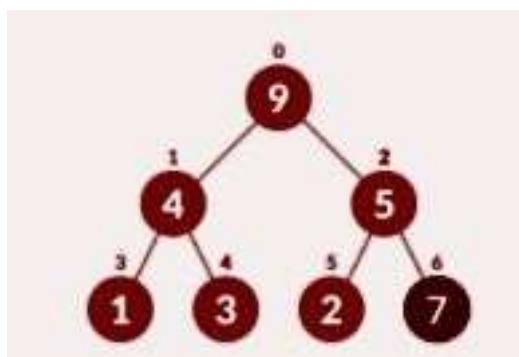
if (largest!=i) {
    int temp = heap.get(i);
    heap.set(i, heap.get(largest));
    heap.set(largest, temp);
    heapify(largest);
}
}
}

```

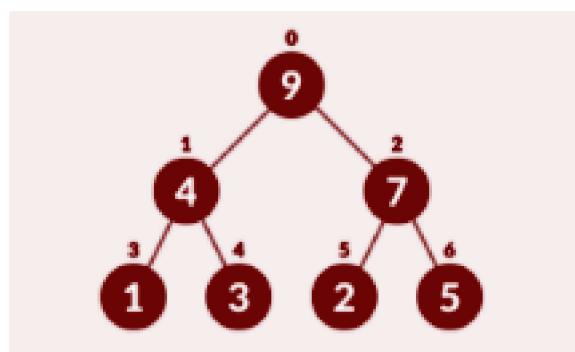
Insert Element into Heap

Insertion into a heap can be done in two steps:

- Insert the new element at the end of the tree. **#Step1**



- Move that element to its correct position in the heap.



Java Code

```
public void insert(int element) {  
  
    heap.add(element);  
    int childIndex = heap.size()-1;  
    int parentIndex = (childIndex-1)/2;  
    while(childIndex > 0) {  
        if(heap.get(parentIndex) < heap.get(childIndex)) {  
            int temp = heap.get(parentIndex);  
            heap.set(parentIndex, heap.get(childIndex));  
            heap.set(childIndex, temp);  
            childIndex = parentIndex;  
            parentIndex = (childIndex - 1)/2;  
        }  
        else {  
            break;  
        }  
    }  
}
```

Delete Max Element from Heap

There are three easy steps to remove max from the heap, we know that 0th element would be the max.

- Set the 0th element with the last element.
- Remove the last element from the heap.
- Heapify the 0th element.

If you want to return the max element then store it before replacing it with the last index, and return it in the end. Below shown is the java code for the same.

Java Code

```
public int removeMax() throws PriorityQueueException {  
  
    if(isEmpty()) {  
        throw new PriorityQueueException();  
    }  
    int retVal = heap.get(0);  
    heap.set(0, heap.get(heap.size()-1));  
    heap.remove(heap.size()-1);  
    if(heap.size() > 1) {  
        heapify(0);  
    }  
    return retVal;  
}
```

Implementation of Priority Queue

- Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.
- Among these data structures, heap data structure provides an efficient implementation of priority queues.
- Hence, we will be using the heap data structure to implement the priority queue in this tutorial.

Insertion and Deletion in a Priority Queue

The first step would be to represent the priority queue in the form of a max/min-heap. Once it is heapified, the insertion and deletion operations can be performed similar to that in a Heap. Refer to the codes discussed above for more clarity.

Heap Sort

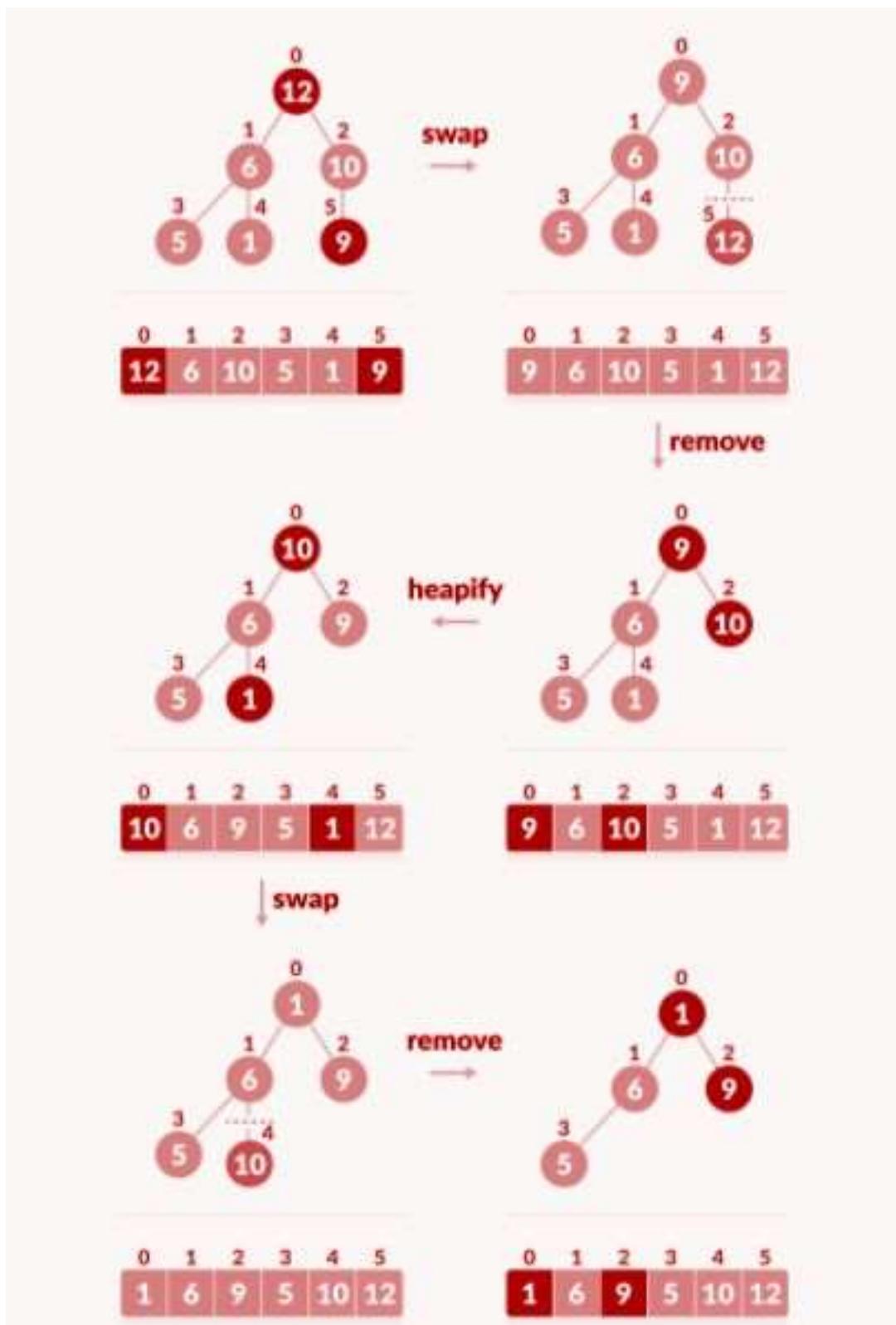
- Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a great worst-case runtime of **O(nlog(n))** regardless of the input data.
- As the name suggests, Heap Sort relies heavily on the heap data structure - a common implementation of a Priority Queue.
- Without a doubt, Heap Sort is one of the simplest sorting algorithms to implement, and coupled with the fact that it's a fairly efficient algorithm compared to other simple implementations, it's a common one to encounter.

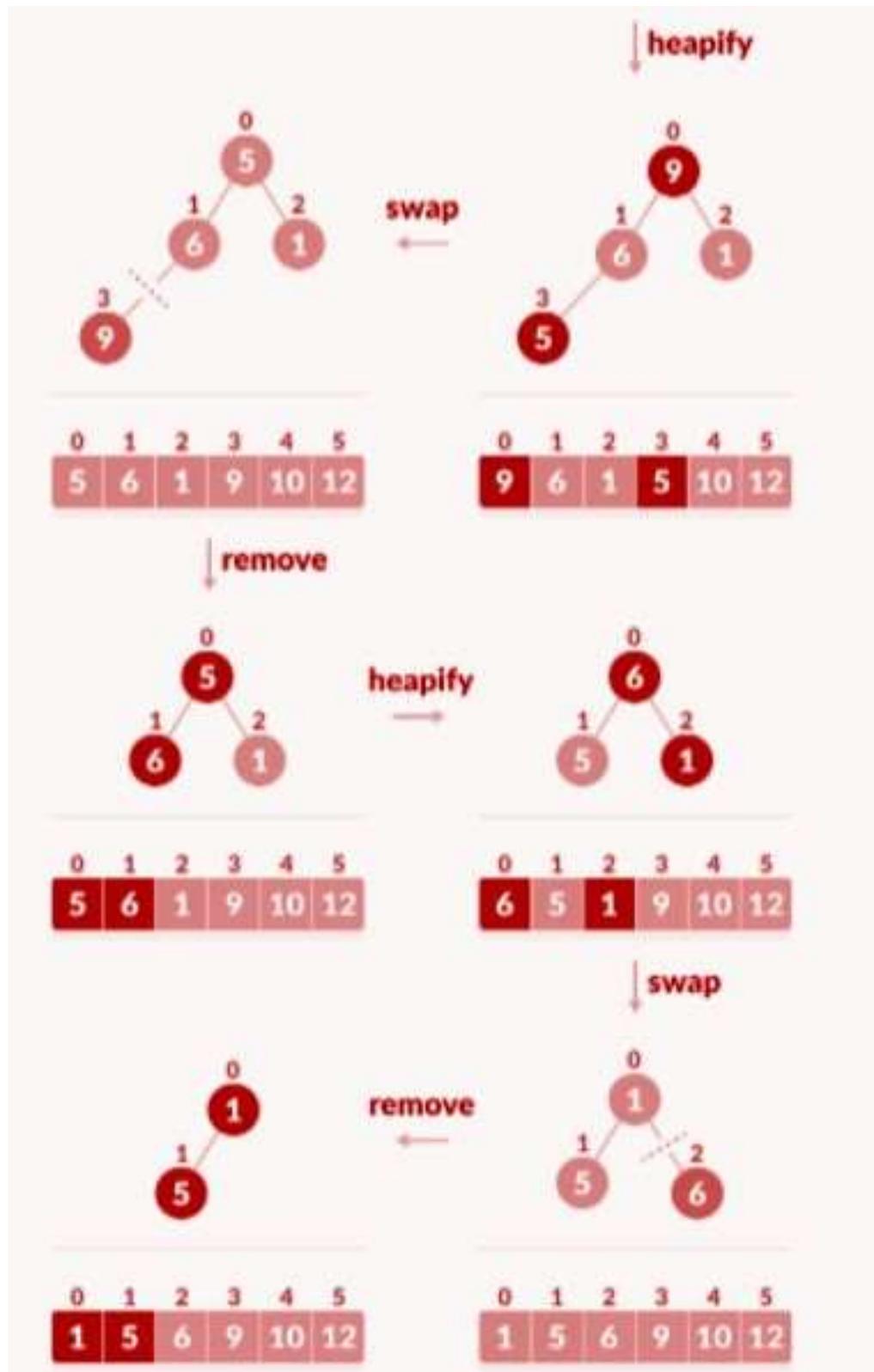
Algorithm

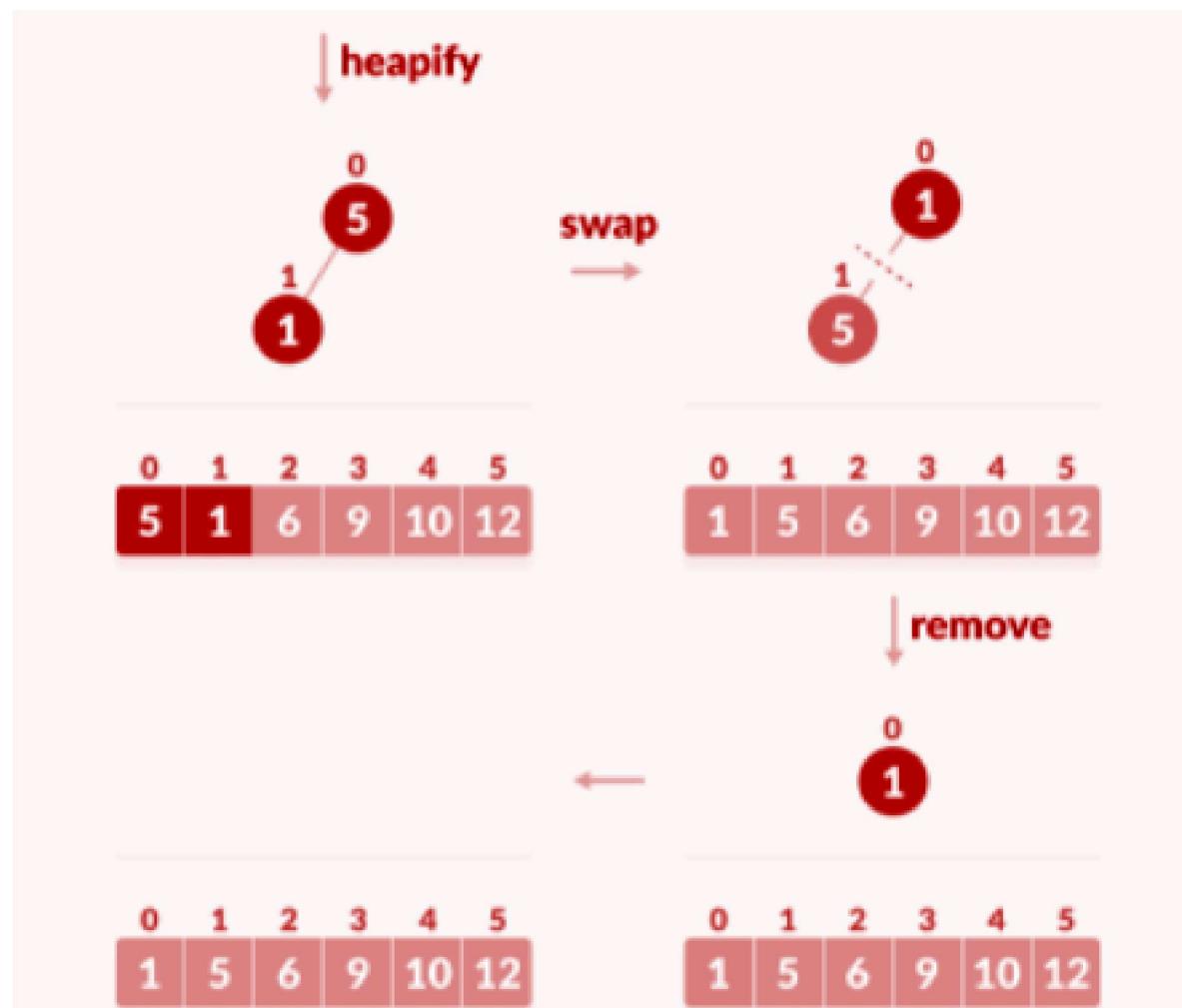
- The heap-sort algorithm inserts all elements (from an unsorted array) into a maxheap.
- Note that heap sort can be done **in-place** with the array to be sorted.
- Since the tree satisfies the Max-Heap property, then the largest item is stored at the root node.
- **Swap:** Remove the root element and put at the end of the array (nth position)
- Put the last item of the tree (heap) at the vacant place.
- **Remove:** Reduce the size of the heap by 1.
- **Heapify:** Heapify the root element again so that we have the highest element at root.
- The process is repeated until all the items in the list are sorted.

Consider the given illustrated example:

-> Applying heapsort to the unsorted array **[12, 6, 10, 5, 1, 9]**







Go through the given **Java Code** for better understanding:

```
public static void heapSort(int input[], int startIndex, int endIndex) {

    for(int i=(input.length/2)-1; i>=0; i--) {
        heapify(input, i, input.length);
    }

    int n = input.length;
    for (int i=n-1; i>=0; i--) {
        // Move current root to end
        int temp = input[0];
        input[0] = input[i];
        input[i] = temp;

        // call heapify on the reduced heap
    }
}
```

```

        heapify(input, 0, i);
    }

}

public static void heapify(int input[], int index, int arrLength) {
    int largest = index;
    int left = 2*index+1;
    int right = 2*index+2;
    if(left<arrLength && input[left]>input[largest])
        largest = left;
    }
    if(right<arrLength && input[right]>input[largest]){
        largest = right;
    }

    if(largest!=index){
        int k = input[index];
        input[index] = input[largest];
        input[largest] = k;
        heapify(input, largest, arrLength);
    }
}

```

In-built Min-Heap in Java

A heap is created by using java's inbuilt library named **PriorityQueue**. This library has the relevant functions to carry out various operations on a **min-heap** data structure. Below is a list of these functions.

- **add**- This function adds an element to the heap without altering the current heap.
- **remove**- This function returns the smallest data element from the heap.

Creating, inserting and removing from a Min-Heap

In the below example we supply a list of elements and the heapify function rearranges the elements bringing the smallest element to the first position.

```
import java.util.PriorityQueue;
```

```

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

        PriorityQueue<Integer> pq = new PriorityQueue<>();
        int arr[] = {9,1,0,4,7,3};
        for(int i = 0; i < arr.length; i++){
            pq.add(arr[i]);
        }
        for(int i = 0; i < arr.length; i++){
            pq.remove();
        }
    }
}

```

When the above code is executed, it produces the following result –

[0, 1, 3, 4, 7, 9]

In-built Max-Heap in Python

For max heap you just need to change the initialisation with reverse order as shown below:

```
PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
```

K-Smallest Elements in a List

This is a good example of problem-solving via a heap data structure. The basic idea here is to create a min-heap of all n elements and then extract the minimum element K times (We know that the root element in a min-heap is the smallest element).

Approach

- Build a min-heap of size n of all elements.
- Extract the minimum elements K times, i.e. delete the root and perform heapify operation K times.
- Store all these K smallest elements.

Note: The code written using these insights can be found in the solution tab of the problem itself.

Tries and Huffman Coding

Introduction to Tries

Suppose we want to implement a word-dictionary using a Java program and perform the following functions:

- Addition of the word(s)
- Searching a word
- Removal of the word(s)

To do the same, hashmaps can be thought of as an efficient data structure as the **average case time complexity** of insertion, deletion, and retrieval is $O(1)$ for integer, character, float, and decimal values.

Let us discuss the time complexity of the same in case of strings.

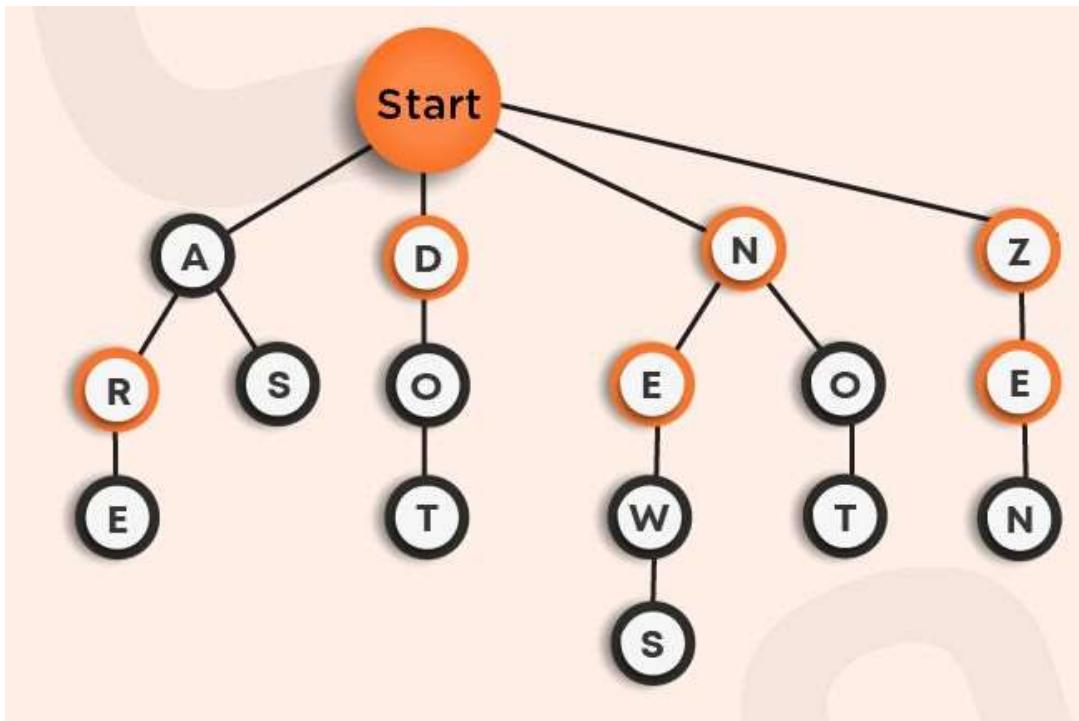
Suppose we want to insert string ***abc*** in our hashmap. To do so, first, we would need to calculate the hashCode for it, which would require the traversal of the whole string *abc*. Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any of the above operations. Thus, we can conclude that the insertion of a string will be $O(\text{string_length})$.

To search a word in the hashmap, we again have to calculate the hashCode of the string to be searched, and for that also, it would require $O(\text{string_length})$ time.

Similarly, for removing a word from the hashmap, we would have to calculate the hashCode for that string. It would again require $O(\text{string_length})$ time.

For the same purpose, we can use another data structure known as **tries**.

Below is the visual representation of the trie:



Here, the node at the top named as the **start** is the root node of our **n-ary tree**.

Suppose we want to insert the word **ARE** in the trie. We will begin from the root, search for the first word **A** and then insert **R** as its child and similarly insert the letter **E**. You can see it as the left-most path in the above trie.

Now, we want to insert the word **AS** in the trie. We will follow the same procedure as above. First-of-all, we find the letter **A** as the child of the root node, and then we will search for **S**. If **S** was already present as the child of **A**, then we will do nothing as the given word is already present otherwise, we will insert **S**. You can see this in the above trie. Similarly, we added other words in the trie.

This approach also takes $O(\text{word_length})$ time for insertion.

Moving on to searching a word in the trie, for that also, we would have to traverse the complete length of the string, which would take $O(\text{word_length})$ time.

Let us take an example. Suppose we want to search for the word **DOT** in the above trie, we will first-of-all search for the letter **D** as the direct child of the start node and then search for **O** and **T** and, then we will return true as the word is present in the trie.

Consider another example **AR**, which we want to search for in the given trie.

Following the above approach, we would return true as the given word is present in it. However ideally, we should return false as the actual word was **ARE** and not **AR**. To overcome this, it can be observed that in the above trie, some of the letters are marked with a bolder circle, and others are not. This boldness represents the termination of a word starting from the root node. Hence, while searching for a word, we will always be careful about the last letter that it should be bolded to represent the termination.

Note: While insertion in a trie, the last letter of the word should be bolded as a mark of termination of a word.

In the above trie, the following are all possibilities that can occur:

- ARE, AS
- DO, DOT
- NEW, NEWS, NOT
- ZEN

Here, to remove the word, we will simply unbold the terminating letter as it will make that word invalid. For example: If you want to remove the string **DO** from the above trie by preserving the occurrence of string DOT, then we will reach **O** and then unbolden it. This way the word **DO** is removed but at the same time, another

word **DOT** which was on the same path as that of word **DO** was still preserved in the trie structure.

For removal of a word from trie, the time complexity is still $O(\text{word_length})$ as we are traversing the complete length of the word to reach the last letter to unbold it.

When to use tries over hashmaps?

It can be observed that using tries, we can improve the space complexity.

For example: We have 1000 words starting from character **A** that we want to store. Now, if you try to hold these words using hashmap, then for each word, we have to store character **A** differently. But in case of tries, we only need to store the character **A** once. In this way, we can save a lot of space, and hence space optimization leads us to prefer tries over hashmaps in such scenarios.

In the beginning, we thought of implementing a dictionary. Let's recall a feature of a dictionary, namely **Auto-Search**. While browsing the dictionary, we start by typing a character. All the words beginning from that character appear in the search list. But this functionality can't be achieved using hashmaps as in the hashmap, the data stored is independent of each other, whereas, in case of tries, the data is stored in the form of a tree-like structure. Hence, here also, tries prove to be efficient over hashmaps.

TrieNode Class Implementation

Follow the below-mentioned code (with comments)...

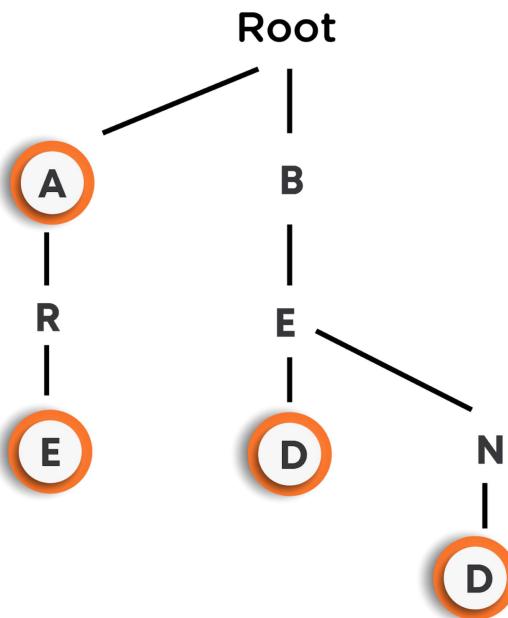
```
class TrieNode {  
    char data;          // To store data (character value: 'A' - 'Z')  
    TrieNode[] children; // To store the address of each child  
    boolean isTerminal; // it will be TRUE if the word terminates  
                        // at this character
```

```

public TrieNode(char data) { // Constructor for values initialization
    this.data = data;
    children = new TrieNode[26];
    for(int i = 0; i < 26; i++) {
        children[i] = null;
    }
    isTerminal = false;
}
  
```

Insert Function

To insert a word in a trie, we will use recursion. Suppose we have the following trie:



Now we want to insert the word **BET** in our trie.

Recursion says that we need to work on a smaller problem, and the rest it will handle itself. So, we will do it on the root node.

Note: Practically, the functionality of bolding the terminal character is achieved using the boolean **isTerminal** variable for that particular node. If this value is true means that the node is the terminal value of the string, otherwise not.

Approach:

We will first search for letter **B** and check if it is present as the children of the root node or not and then call recursion on it. If **B** is present as a child of the root node as in our case it is, then we will simply recurse over it by shifting the length of the string by 1. In case, character **B** was not the direct child of the root node, then we have to create one and then call recursion on it. After the recursive call, we will see that **B** is now a root node, and the character **E** is now the word we are searching for. We will follow the same procedure as done in searching for character **B** against the root node and then move forward to the next character of the string, i.e., **T**, which happens to be the last character of our string. Now we will check character **T** as the child of character **E**. In our case, it is not a child of character **E**, so we'll create it. As **T** is the last character of the string, so we will mark its **isTerminal** value to **True**.

Following will be the three steps of recursion:

- **Small Calculation:** We will check if the root node has the first character of the string as one of its children or not. If not, then we will create one.
- **Recursive call:** We will tell the recursion to insert the remaining string in the subtree of our trie.
- **Base Case:** As the length of the string becomes zero, or in other words, we reach the NULL character, then we need to mark the **isTerminal** for the last character as True.

Follow the code below, along with the comments...

```

class Trie {
    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insertWord(TrieNode root, String word) {
        // Base case
        if(word.length() == 0) {
            root.isTerminal = true;
            return;
        }
        // Small Calculation
        int index = word.charAt(0) - 'a'; // As for 'a' refers to
        // index 0, 'b' refers to index 2 and so on,
        // so to reach the correct index we will do so
        TrieNode child;
        if(root.children[index] != null) { //If the first character of
        // string is already present as the child node of the root node
            child = root.children[index];
        }
        else { // If not present as the child then creating one.
            child = new TrieNode(word.charAt(0));
            root.children[index] = child;
        }

        // Recursive call
        insertWord(child, word.substring(1));
    }
    // For user
    public void insertWord(String word) {
        insertWord(root, word);
    }
}

```

Search in Tries

Objective: Create a search function which will get a string as its argument to be searched in the trie and returns a boolean value. **True** if the string is present in the trie and **False** if not.

Approach: We will be using the same process as that used during insertion. We will call recursion over the root node after searching for the first character as its child. If the first character is not present as one of the children of the root node, then we will simply return false; otherwise, we will send the remaining string to the recursion. When we reach the empty string, then check for the last character's **isTerminal** value; if it is **true**, means that word exists in our trie, and we will return true from there otherwise, return false.

Try to code it yourselves, and for the code, refer to the solution tab of the same.

Tries Implementation: Remove

Objective: To delete the given word from our trie.

Approach: First-of-all we need to search for the word in the trie, and if found, then we simply need to mark the **isTerminal** value of the last character of the word to **false** as that will simply denote that the word does not exist in our trie.

Check out the code below: (Nearly same as that of insertion, just a minor changes which are explained along side)

```
void removeWord(TrieNode root, String word) {
    // Base case
    if(word.length() == 0) {
        root.isTerminal = false;
        return;
    }
    // Small calculation
    TrieNode child;
    int index = word.charAt(0) - 'a';
    if(root.children[index] != null) {
        child = root.children[index];
    }
    else {
        // Word not found
        return;
    }
    removeWord(child, word.substring(1));
    if(child.isTerminal && child.children.length == 0) {
        root.children[index] = null;
    }
}
```

```

}

removeWord(child, word.substring(1));
// Suppose if the character of the string doesn't have any child and is a
part of the word to be deleted, then we can simply delete that node also as
it is not referencing to any other word in the trie

// Removing child Node if it is useless
if(child.isTerminal == false) {
    for(int i = 0; i < 26; i++) {
        if(child.children[i] != null) {
            return;
        }
    }
    delete child;
    root.children[index] = null;
}
}

// For user
void removeWord(string word) {
    removeWord(root, word);
}

```

Types of Tries

There are two types of tries:

- **Compressed tries:**
 - Majorly, used for space optimization.
 - We generally club the characters if they have at most one child.
 - **General rule:** Every node has at least two child nodes.

Refer to the figure below:

Suppose our regular trie looks like this-

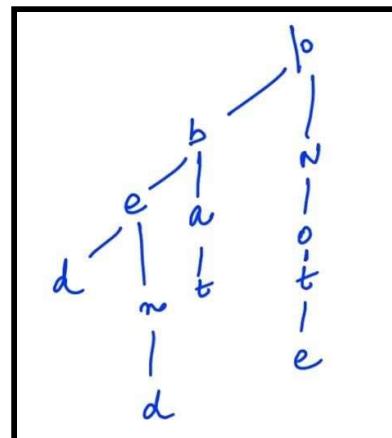
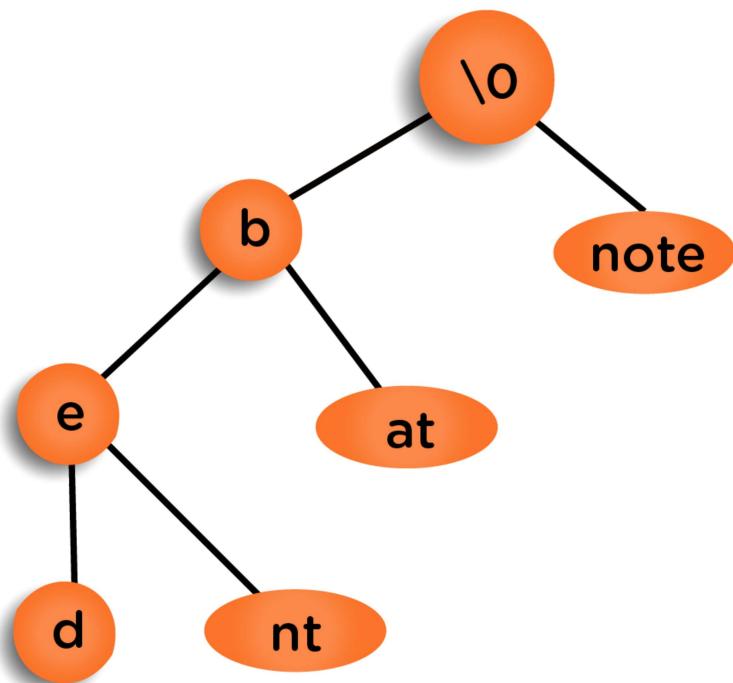


Figure - 1

Its compressed trie version will look as follows:



- **Pattern matching:**

- Used to match patterns in the trie.

- Example: In the figure-1 (shown above), if we want to search for pattern **ben** in the trie, but the word **bend** was present instead, using the normal search function, we would return false, as the last character **n**'s **isTerminal** property was false, but in this trie, we would return true.
 - To overcome this problem of the last character's identification, just remove the **isTerminal** property of the node.
 - In the figure-1, instead of searching for the pattern **ben**, we now want to search for the pattern **en**. Our trie would return false if **en** is not directly connected to the root. But as the pattern is present in the word **ben**, it should return true. To overcome this problem, we need to attach each of the prefix strings to the root node so that every pattern is encountered.
- **For example:** for the string **ben**, we would store **ben**, **en**, **n** in the trie as the direct children of the root node.

Huffman Coding

Introduction

Huffman Coding is one approach followed for **Text Compression**. Text compression means reducing the space requirement for saving a particular text.

Huffman Coding is a lossless data compression algorithm, ie. it is a way of compressing data without the data losing any information in the process. It is useful in cases where there is a series of frequently occurring characters.

Working of Huffman Algorithm:

Suppose, the given string is:



A horizontal row of 15 red rectangular boxes, each containing a character from the string: B, C, A, A, D, D, D, C, C, A, C, A, C, A, C. This represents the initial string for Huffman coding.

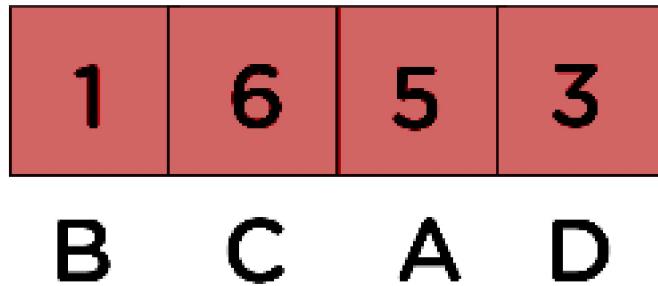
Initial String

Here, each of the characters of the string takes 8 bits of memory. Since there are a total of 15 characters in the string so the total memory consumption will be $15*8 = 120$ bits. Let's try to compress its size using the Huffman Algorithm.

First-of-all, Huffman Coding creates a tree by calculating the frequencies of each character of the string and then assigns them some unique code so that we can retrieve the data back using these codes.

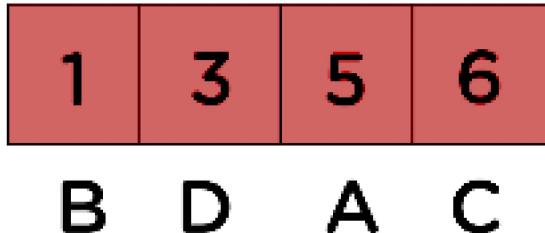
Follow the steps below:

1. Begin with calculating the frequency of each character value in the given string.



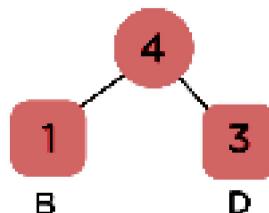
Frequency of String

2. Sort the characters in ascending order concerning their frequency and store them in a priority queue, say **Q**.
3. Each character should be considered as a different leaf node.



Characters sorted according to the frequency

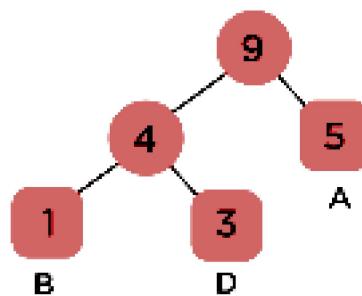
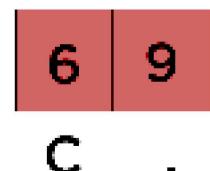
4. Make an empty node, say **z**. The left child of z is marked as the minimum frequency and the right child, the second minimum frequency. The value of z is calculated by summing up the first two frequencies.



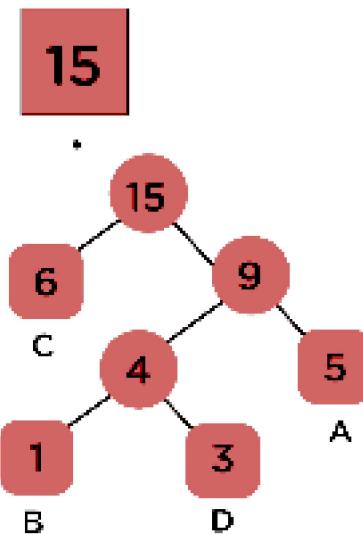
Getting the sum of the least numbers

Here, “.” denote the internal nodes.

5. Now, remove the two characters with the lowest frequencies from the priority queue **Q** and append their sum to the same.
6. Simply insert the above node **z** to the tree.
7. For every character in the string, repeat steps 3 to 5.

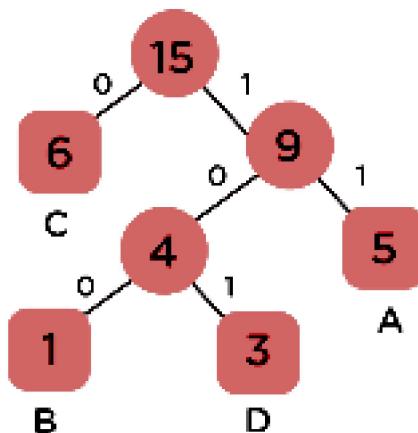


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. Assign 0 to the left side and 1 to the right side except for the leaf nodes.



Assign 0 to the left edge and 1 to the right edge

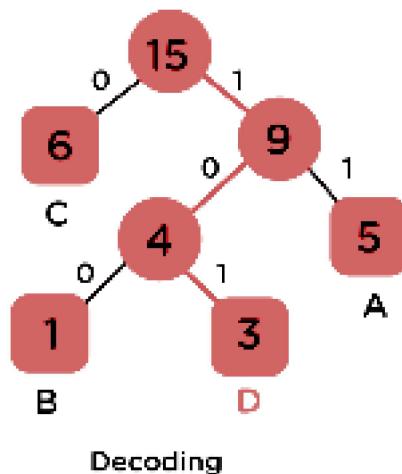
The size table is given below:

Character	Frequency	Code	Size
A	5	11	$5*2 = 10$
B	1	100	$1*3 = 3$
C	6	0	$6*1 = 6$
D	3	101	$3*3 = 9$
$4*8 = 32$ bits	15 bits		28 bits

Size before encoding: 120 bits

Size after encoding: $32 + 15 + 28 = 75$ bits

To decode the code, simply traverse through the tree (starting from the root) to find the character. Suppose we want to decode 101, then:



Time Complexity:

In the case of encoding, inserting each character into the priority queue takes **O(log n)** time. Therefore, for the complete array, the time complexity becomes **O(nlog(n))**.

Similarly, extraction of the element from the priority queue takes **O(log n)** time. Hence, for the complete array, the achieved time complexity is **O(nlog n)**.

Applications of Huffman Coding:

- They are used for transmitting fax and text.
- They are used by conventional compression formats like PKZIP, GZIP, etc.

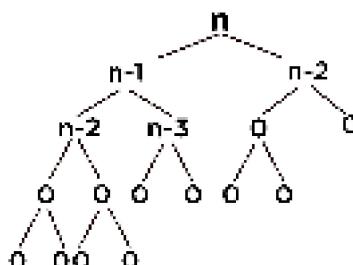
Dynamic Programming - 1

Introduction

Suppose we need to find the n^{th} Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
public static int fibo(int n){
    if(n <= 1)
        return n;
    return fibo(n-1) + fibo(n-2);
}
```

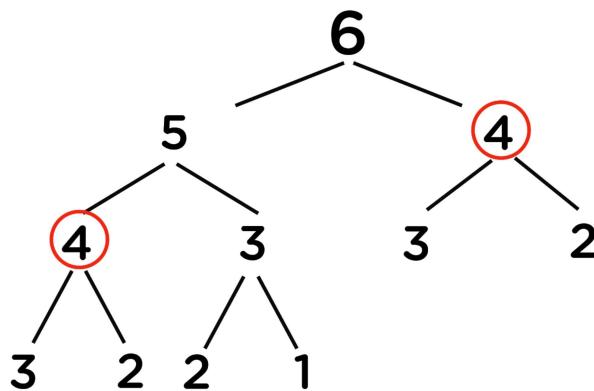
- Here, for every n , we need to make a recursive call to $f(n-1)$ and $f(n-2)$.
- For $f(n-1)$, we will again make the recursive call to $f(n-2)$ and $f(n-3)$.
- Similarly, for $f(n-2)$, recursive calls are made on $f(n-3)$ and $f(n-4)$ until we reach the base case.
- The recursive call diagram will look something like shown below:



- At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one).
- At every level, we are doing $2^n K$ work (where $n = 0, 1, 2, \dots$).
- Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^{n-1}k$ work.
- Total work can be calculated as:

$$(2^0 + 2^1 + 2^2 + \dots + 2^{n-1}) * k \approx 2^n k$$

- Hence, it means time complexity will be $O(2^n)$.
- We need to improve this complexity. Let's look at the example below for finding the 6th Fibonacci number.



Important Observation:

- We can observe that there are repeating recursive calls made over the entire program.
- As in the above figure, for calculating $f(5)$, we need the value of $f(4)$ (first recursive call over $f(4)$), and for calculating $f(6)$, we again need the value of $f(4)$ (second similar recursive call over $f(4)$).
- Both of these recursive calls are shown above in the outlining circle.
- Similarly, there are many other values for which we are repeating the recursive calls.
- Generally, while recursing, there are repeated recursion calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered values (preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will

directly consider their already stored outputs and then use these in our calculations instead of calculating them over again.

This way, we can improve the running time of our code. This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called **Memoization**.

- To achieve this in our example we will simply take an answer array, initialized to -1.
- Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not.
- If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value.
- After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most **(n+1)** only.

Let's look at the memoization code for Fibonacci numbers below:

```
private static int fibo_helper(int n, int[] ans){
    if (n==0 || n==1)          //Base case
        return n;

    //check if output already exists
    if (ans[n] != -1){
        return ans[n];
    }

    // calculate output
    int a = fibo_helper(n-1, ans);
    int b = fibo_helper(n-2, ans);
```

```

        // save the output for future use
        ans[n] = a + b;

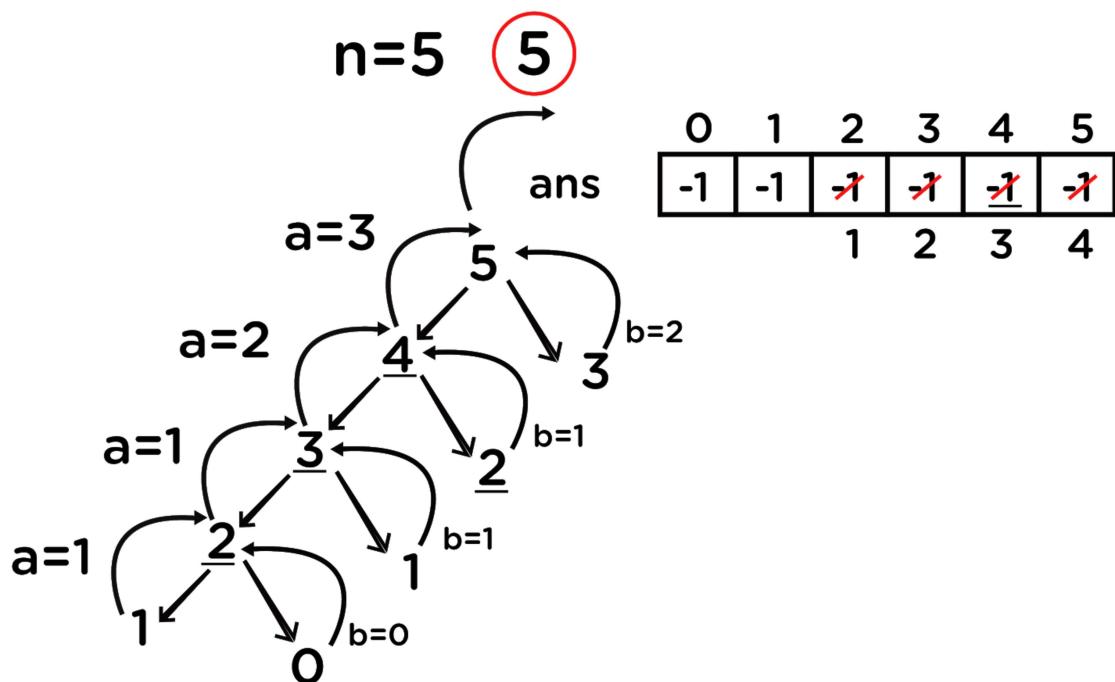
        // return the final output
        return ans[n];
    }

public static int fibo_2(int n){
    int[] ans = new int[n+1];

    for(int i=0; i<=n; i++){
        ans[i] = -1;           // -1 represents that fibb for that
                            // index does not exist
    }

    return fibo_helper(n, ans);
}
    
```

Let's dry run for $n = 5$, to get a better understanding:



Again, if we observe carefully, we can see that for any number, we are not able to make a recursive call on the right side of it. This means that we can make at most $5+1 = 6$ ($n+1$) unique recursive calls which reduce the time complexity to $O(n)$ which is highly optimized as compared to simple recursion.

Summary

- Memoization is a **top-down approach**, where we save the previous answers so that they can be used to calculate future answers and improve the time complexity to a greater extent.
- Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored.
- Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes.
- In cases of Fibonacci numbers, these indexes are 0 and 1 as $f(0) = 0$ and $f(1) = 1$. So we can directly allot these two values to our answer array and then use these to calculate $f(2)$, which is $f(1) + f(0)$, and so on for every other index.
- This can be simply done iteratively by running a loop from $i = (2 \text{ to } n)$.
- Finally, we will get our answer at the 5th index of the answer array as we already know that the i -th index contains the answer to the i -th value.

We are first trying to figure out the dependency of the current value on the previous values and then using them calculating our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, we will follow a **bottom-up approach** to reach the desired index. This approach of converting recursion into iteration is known as **Dynamic programming(DP)**.

Let us now look at the DP code for calculating the n^{th} Fibonacci number:

```

public static int fibo_3(int n){
    int[] ans = new int[n+1];

    ans[0] = 0;      // storing independent values in solution array
    ans[1] = 1;

    // following bottom-up approach to reach n
    for(int i=2; i<=n; i++){
        ans[i] = ans[i-1] + ans[i-2];
    }

    return ans[n];    // final answer
}
  
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach.

For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimize the recursive approach by storing the previous answers using memoization.
3. Finally, replace recursion by iteration using dynamic programming. (It is preferred to be done in this manner because recursion generally has an increased space complexity as compared to iteration methods.)

Problem Statement: Min steps to 1

Given a positive integer n , find the minimum number of steps s , that takes n to 1. You can perform any one of the following three steps:

1. Subtract 1 from it. ($n = n-1$).
2. If its divisible by 2, divide by 2. (if $n \% 2 == 0$, then $n = n/2$).

3. If its divisible by 3, divide by 3. (if $n \% 3 == 0$, then $n = n / 3$).

Example 1: For $n = 4$:

STEP-1: $n = 4 / 2 = 2$

STEP-2: $n = 2 / 2 = 1$

Hence, the answer is **2**.

Example 2: For $n = 7$:

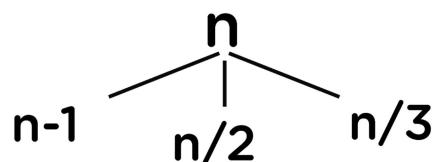
STEP-1: $n = 7 - 1 = 6$

STEP-2: $n = 6 / 3 = 2$

STEP-3: $n = 2 / 2 = 1$

Hence, the answer is **3**.

Approach: We are only allowed to perform the above three mentioned ways to reduce any number to 1.



Let's start thinking about the brute-force approach first, i.e., recursion.

We will make a recursive call to each of the three steps keeping in mind that for dividing by 2, the number should be divisible by 2 and similarly for 3 as given in the question statement. After that take the minimum value out of the three obtained and simply add 1 to the answer for the current step itself. Thinking about the base case, we can see that on reaching 1, simply we have to return 0 as it is our destination value. Let's now look at the code:

```

public static int minSteps(int n){
    // base case
    if (n <= 1)
        return 0;

    int x = minSteps(n-1);           // Recursive call 1

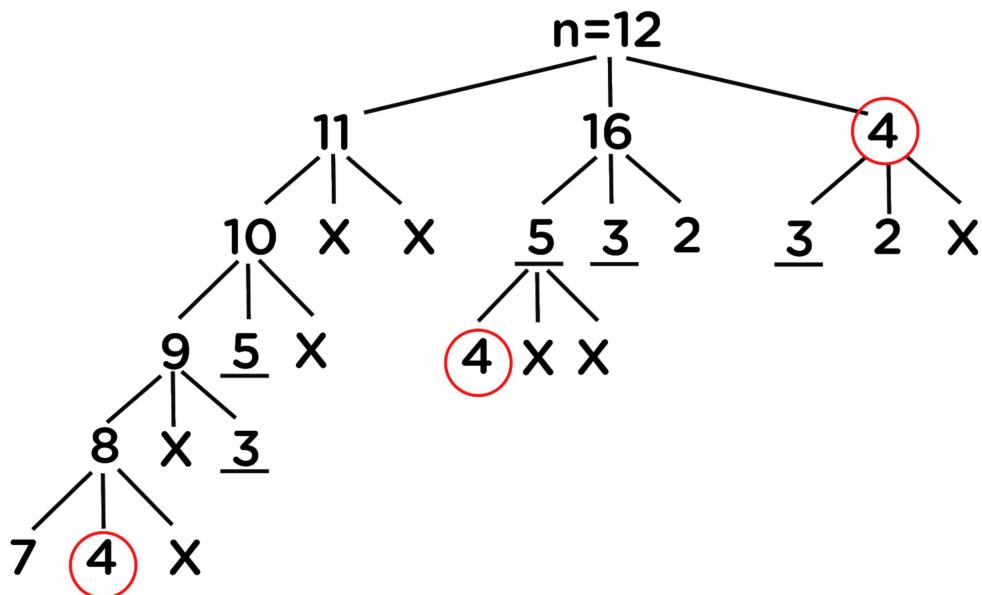
    int y = Integer.MAX_VALUE;      // Initialise to infinity to check
    int z = Integer.MAX_VALUE;      // divisibility by 2 or 3

    if (n%2 == 0)
        y = minSteps(n/2);         //Recursive call 2
    if (n%3 == 0)
        z = minSteps(n/3);         //Recursive call 3

    // Calculating answer
    int ans = Math.min(x, Math.min(y, z)) + 1;
    return ans;
}
  
```

Now, we have to check if we can optimize the code that we have written. It can be done using memoization. But, for memoization to apply, we need to check if there are any overlapping sub-problems so that we can store the previous values to obtain the new ones. To check this let's dry run the problem for $n = 12$:

(Here X represents that the calls are not feasible as the number is not divisible by either of 2 or 3)



Here, if we blindly make three recursive calls at each step, then the time complexity will approximately be **$O(3^n)$** .

From the above, it is visible that there are repeating sub-problems. Hence, this problem can be optimized using memoization.

Now, we need to figure out the number of unique calls, i.e., how many answers we are required to save. It is clear that we need at most $n+1$ responses to be saved, starting from $n = 0$, and the final answer will be present at index n .

The code will be nearly the same as the recursive approach; just we will not be making recursive calls for already stored outputs. Follow the code and comments below:

```
private static int minStepsHelper(n, int[] memo){
    // base case
    if (n == 1)
        return 0;
```

```

if (memo[n] != -1)
    return memo[n];

int res = minStepsHelper(n-1, memo);

if (n%2 == 0)
    res = Math.min(res, minStepsHelper(n/2, memo));
if (n%3 == 0)
    res = Math.min(res, minStepsHelper(n/3, memo));

// store memo[n] and return
memo[n] = 1 + res;
return memo[n];
}

public static int minSteps_2(int n){

    int[] memo = new int[n+1];

    // initialize memoized array with -1
    for (int i=0; i<=n; i++)
        memo[i] = -1;

    return minStepsHelper(n, memo);
}

```

Time complexity has been reduced significantly to **O(n)** as there are only **(n+1)** unique iterations. Now, try to code the DP approach by yourself, and for the code, refer to the solution tab.

Problem Statement: Minimum Number of Squares

Given an integer N, find and return the count of minimum numbers, the sum of whose squares is equal to N.

That is, if N is 4, then we can represent it as : $\{1^2 + 1^2 + 1^2 + 1^2\}$ and $\{2^2\}$.

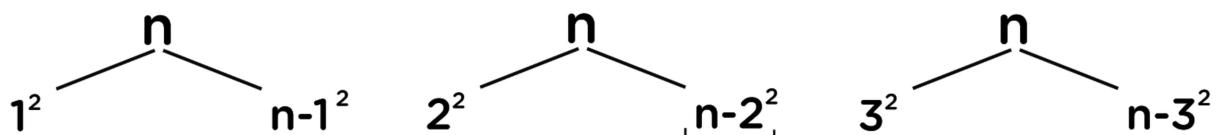
The output will be 1, as 1 is the minimum count of numbers required. (x^y represents x raised to the power y.)

Example: For $n = 12$, we have the following ways:

- $1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1$
- $1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 2^2$
- $1^1 + 1^1 + 1^1 + 1^1 + 2^2 + 2^2$
- $2^2 + 2^2 + 2^2$

Hence, the minimum count is obtained from the 4-th option. Therefore, the answer is equal to 3.

Approach: First-of-all, we need to think about breaking the problems into two parts, one of which will be handled by recursion and the other one will be handled by us(smaller sub-problem). We can break the problem as follows:



And so on...

- In the above figure, it is clear that in the left subtree we are making ourselves try over a variety of values that can be included as a part of our solution.
- The right subtree's calculation will be done by recursion.
- Hence, we will just handle the i^2 part, and $(n-i^2)$ will be handled by recursion.
- By now, we have got ourselves an idea of solving this problem, the only thinking left is the loop's range on which we will be iterating, i.e., the values of i for which we will be deciding to consider while solving or not.

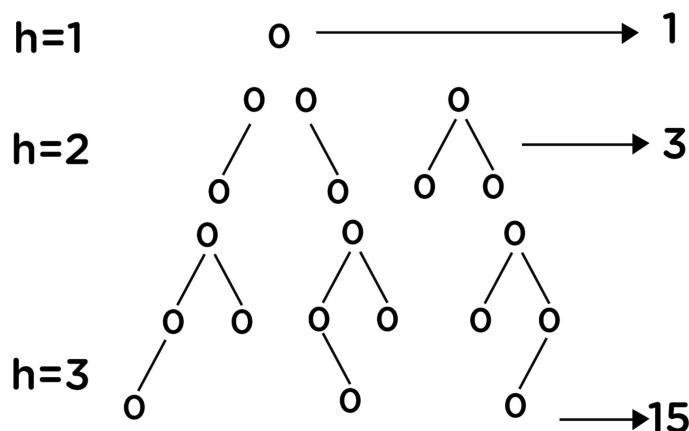
- As the maximum value up to which i can be pushed, to reach n is \sqrt{n} as $(\sqrt{n} * \sqrt{n} = n)$. Hence, we will be iterating over the range **(1 to \sqrt{n})** and do consider each possible way by sending **($n-i^2$)** over the recursion.
- This way we will get different subsequences and as per the question, we will simply return the minimum out of it.

This problem is left for you to try out using all the three approaches and for code, refer to the solution tab of the same.

No. of balanced BTs

Problem Statement: Given an integer h , find the possible number of balanced binary trees of height h . You just need to return the count of possible binary trees which are balanced. This number can be huge, so return output modulo $10^9 + 7$.

For Example: In the figure below, the left side represents the height h , and the right side represents the possible binary trees along with the count.

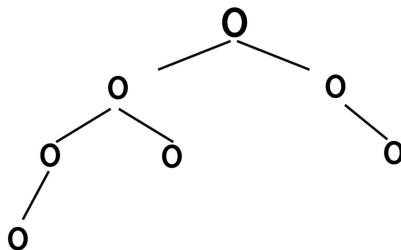


Here for $h = 1$, the answer is 1. For $h = 2$, the answer is 3. For $h = 3$, the answer is 15.

Approach: Suppose we have $h = 3$, so at level 3 there are four nodes and each node has two options, either it can be included or excluded from the binary tree,

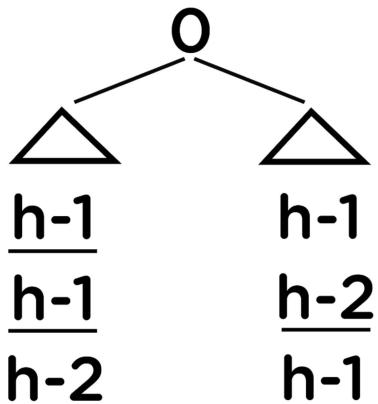
hence in total, we have $2^4 = 16$ possibilities. Here, we need to exclude the possibility of the case when none out of 4 is present. Hence, the remaining options are $16-1 = 15$. We can think that this approach could be efficient as we just need to know the number of nodes at the last level, and then we can simply apply the above formula.

Now, consider for $h = 4$, where the last level has got eight nodes, so according to the above approach, the answer could be $2^8 - 1 = 255$ ways, but the solution for $h = 4$ is 315. Let's look at the cases which we missed. One of the examples could be:



Till now, we were only working over the last level, but in the above example, if we remove the nodes from upper levels, then also the binary tree could remain balanced.

Let's now think about implementing it using recursion on trees. If the height of the complete binary tree is h , that means the height of its left and right subtrees individually is at most $h-1$. So if the height of the left subtree is $h-1$, then the height of the right subtree could be any among $\{h-1, h-2\}$ and vice versa.



Initially, we were given the problem of finding the output for height h . Now we have reduced the same to tell the output of height $h-1$ and $h-2$. Finally, we just need to figure out these counts, add them, and return.

Let's now look at the code below:

```

public static int balancedBTs(int h) {
    if(h <= 1) {                                // Base case
        return 1;
    }

    int mod = (int) (Math.pow(10, 9)) + 7;
    int x = balancedBTs(h - 1);                  // Answer for h-1
    int y = balancedBTs(h - 2);                  // Answer for h-2

    /* Since, we need to find the total number of combinations, so will
     multiply the left height's output and the right height's output as they are
     independent of each other (Using law of multiplication in combinations)

    Possible Cases:
    • Both h-1           =      x*x
    • h-1 and h-2       =      x*y
    • h-2 and h-1       =      y*x

    Now, we will add all these together.
*/
  
```

```
int temp1 = (int)((long)(x)*x) % mod;
int temp2 = (int)((2* (long)(x) * y) % mod);
int ans = (temp1 + temp2) % mod;

return ans;
}
```

Time Complexity: If we observe this function, then we can find it very similar to the pattern of the Fibonacci number. Hence, the time complexity is $O(2^h)$.

Now, try to reduce the time complexity of the code using memoization and DP by yourselves and for solution refer to the solution tab of the problem.

Dynamic Programming- 2

Let us now move to some advanced-level DP questions, which deal with 2D arrays.

Problem Statement: Min Cost Path

Given an integer matrix of size $m \times n$, you need to find out the value of minimum cost to reach from the cell $(0, 0)$ to $(m-1, n-1)$. From a cell (i, j) , you can move in three directions : $(i+1, j)$, $(i, j+1)$ and $(i+1, j+1)$. The cost of a path is defined as the sum of values of each cell through which the path passes.

For example, The given input is as follows-

```
3 4
3 4 1 2
2 1 8 9
4 7 8 1
```

The path that should be followed is **3 -> 1 -> 8 -> 1**. Hence the output is **13**.

Approach:

- Thinking about the **recursive approach** to reach from the cell $(0, 0)$ to $(m-1, n-1)$, we need to decide for every cell about the direction to proceed out of three.
- We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum cost and add the current cell's value to it.
- Let's now look at the recursive code for this problem:

```

public int minCostPath(int[][] input, int m, int n, int i, int j) {
    // Base case: reaching out to the destination cell
    if(i == m- 1 && j == n- 1) {
        return input[i][j];
    }

    if(i >= m || j >= n) { // checking for within the constraints or not
        return Integer.MAX_VALUE; //if not, returning +infinity so that
    } // it will not be considered as the answer

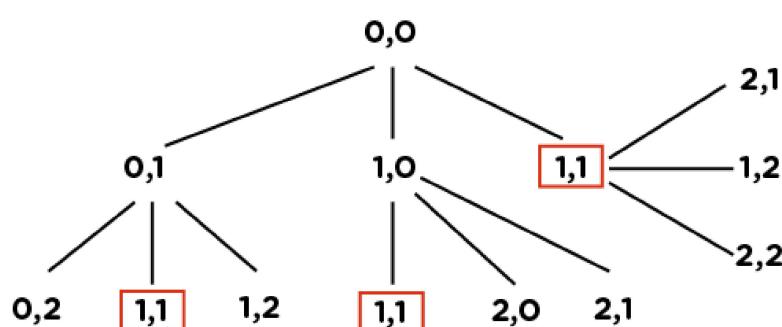
    // Recursive calls
    int x = minCostPath(input, m, n, i, j+1); // Towards right direction
    int y = minCostPath(input, m, n, i+1, j+1); // Towards diagonal
    int z = minCostPath(input, m, n, i+1, j); // Towards down direction

    // Small Calculation: figuring out the minimum value and then adding
    // current cells value to it
    int ans = Math.min(x, Math.min(y, z)) + input[i][j];
    return ans;
}

public int minCostPath(int[][] input, int m, int n) {
    // we will be using a helper function
    return minCostPath(input, m, n, 0, 0); // as we need to keep the
} //track of current row and column
  
```

Let's dry run the approach to see the code flow. Suppose, **m = 4 and n = 5**; then the recursive call flow looks something like below:

m=4, n=5,



Here, we can see that there are many repeated/overlapping recursive calls(for example: **(1,1)** is one of them), leading to exponential time complexity, i.e., **O(3ⁿ)**. If we store the output for each recursive call after their first occurrence, we can easily avoid the repetition. It means that we can improve this using memoization.

Now, let's move on to the **Memoization approach**.

In memoization, we avoid repeated overlapping calls by storing the output of each recursive call in an array. In this case, we will be using a 2D array instead of 1D, as we already discussed in our previous lectures that the storage used for the memoization is generally the same as the one that recursive calls use to their maximum.

Refer to the memoization code (along with the comments) below for better understanding:

```

public int helper(int[][] input, int m, int n, int i, int j, int[][] output)
{
    if(i == m- 1 && j == n- 1) {      // Base case
        return input[i][j];
    }

    if(i >= m || j >= n) {
        return Integer.MAX_VALUE;
    }

    // Check if ans already exists
    if(output[i][j] != -1) {
        return output[i][j];      // as each cell stores its own ans
    }

    // Recursive calls
    int x = helper(input, m, n, i, j+1, output);
    int y = helper(input, m, n, i+1, j+1, output);
    int z = helper(input, m, n, i+1, j, output);

    // Small Calculation
    int a = Math.min(x, Math.min(y, z)) + input[i][j];

    // Save the answer for future use
    output[i][j] = a;
}
  
```

```

    return a;
}

public int minCostPath_Mem(int[][][] input, int m, int n, int i, int j) {
    int[][] output = new int[m][];
    for(int i = 0; i < m; i++) {
        output[i] = new int[n];
        for(int j = 0; j < n; j++) {
            output[i][j] = -1; // Initialising the output array by -1.
                                // Here, -1 denotes that the value of the
                                // current cell is unknown and could be
                                // replaced only after we find the same
        }
    }
    return helper(input, m, n, i, j, output);
}
  
```

Here, we can observe that as we move from the cell **(0,0) to (m-1, n-1)**, in general, the i-th row varies from 0 to m-1, and the j-th column runs from 0 to n-1. Hence, the unique recursive calls will be a maximum of **(m-1) * (n-1)**, which leads to the time complexity of **O(m*n)**.

To get rid of the recursion, we will now proceed towards the **DP approach**.

The DP approach is simple. We just need to create a solution array (lets name that as **ans**), where:

ans[i][j] = minimum cost to reach from (i, j) to (m-1, n-1)

Now, initialize the last row and last column of the matrix with the sum of their values and the value, just after it. This is because, in the last row or column, we can reach there from their forward cell only (You can manually check it), except the cell **(m-1, n-1)**, which is the value itself.

```

ans[m-1][n-1] = cost[m-1][n-1]
ans[m-1][j] = ans[m-1][j+1] + cost[m-1][j] (for 0 < j < n)
ans[i][n-1] = ans[i+1][n-1] + cost[i][m-1] (for 0 < i < m)
  
```

Next, we will simply fill the rest of our answer matrix by checking out the minimum among values from where we could reach them. For this, we will use the same formula as used in the recursive approach:

```
ans[i][j] = Minimum(ans[i+1][j], ans[i+1][j+1], ans[i][j+1]) + cost[i][j]
```

Finally, we will get our answer at the cell (0, 0), which we will return.

The code looks as follows:

```
public int minCost_DP(int[][] input, int m, int n) {
    int[][] ans = new int[m][n];

    ans[m-1][n-1] = input[m-1][n-1];

    // Last row
    for(int j = n - 2; j >= 0; j--) {
        ans[m-1][j] = input[m-1][j] + ans[m-1][j+1];
    }

    // Last col
    for(int i = m-2; i >= 0; i--) {
        ans[i][n-1] = input[i][n-1] + ans[i+1][n-1];
    }

    // Calculation using formula
    for(int i = m-2; i >= 0; i--) {
        for(int j = n-2; j >= 0; j--) {
            ans[i][j] = input[i][j] + Math.min(ans[i][j+1],
                                              Math.min(ans[i+1][j+1], ans[i+1][j]));
        }
    }
    return ans[0][0];      // Our Final answer as discussed above
}
```

Note: This is the bottom-up approach to solve the question using DP.

Problem Statement: LCS (Longest Common Subsequence)

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

Note: Subsequence is a part of the string which can be made by omitting none or some of the characters from that string while maintaining the order of the characters.

If s_1 and s_2 are two given strings then z is the common subsequence of s_1 and s_2 , if z is a subsequence of both of them.

Example 1:

```
s1 = "abcdef"  
s2 = "xyczef"
```

Here, the longest common subsequence is "cef"; hence the answer is 3 (the length of LCS).

Example 2:

```
s1 = "ahkolp"  
s2 = "ehyozp"
```

Here, the longest common subsequence is "hop"; hence the answer is 3.

Approach: Let's first think of a brute-force approach using **recursion**. For LCS, we have to match the starting characters of both strings. If they match, then simply we can break the problem as shown below:

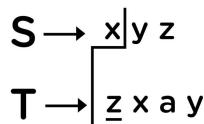
```
s1 = "x|yzar"  
s2 = "x|qwea"
```

The rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can't be directly predicted by just looking at them, so we will be traversing over both of them one-by-one and check for the maximum value of LCS obtained among them to be considered for our answer.

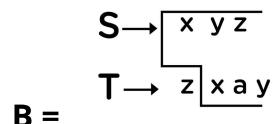
For example:

Suppose, string $s = "xyz"$ and string $t = "zxay"$.

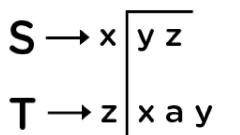
We can see that their first characters do not match so that we can call recursion over it in either of the following ways:



A =



C =



Finally, our answer will be:

```
LCS = max(A, B, C)
```

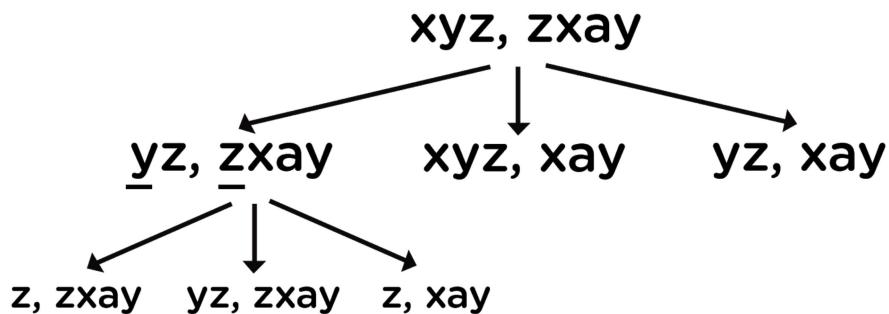
Check the code below and follow the comments for a better understanding.

```

public int lcs(string s, string t) {
    // Base case
    if(s.size() == 0 || t.size() == 0) {
        return 0;
    }

    // Recursive calls
    if(s[0] == t[0]) {
        return 1 + lcs(s.substr(1), t.substr(1));
    }
    else {
        int a = lcs(s.substring(1), t); // discarding the first
                                         // character of string s
        int b = lcs(s, t.substring(1)); // discarding the first
                                         // character of string t
        int c = lcs(s.substring(1), t.substring(1)); // discarding the
                                         // first character of both
        return Math.max(a, Math.max(b, c)); // Small calculation
    }
}
  
```

If we dry run this over the example: $s = "xyz"$ and $t = "zxay"$, it will look something like below:

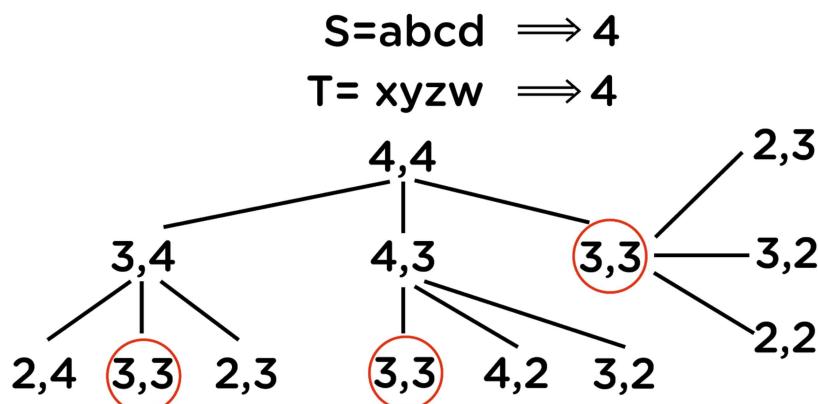


Here, as for each node, we will be making three recursive calls, so the time complexity will be exponential and is represented as $O(2^{m+n})$, where m and n are

the lengths of both strings. This is because, if we carefully observe the above code, then we can skip the third recursive call as it will be covered by the two others.

Now, thinking over improving this time complexity...

Consider the diagram below, where we are representing the dry run in terms of its length taken at each recursive call:



As we can see there are multiple overlapping recursive calls, the solution can be optimized using **memoization** followed by DP. So, beginning with the memoization approach, as we want to match all the subsequences of the given two strings, we have to figure out the number of unique recursive calls. For string s, we can make at most **length(s)** recursive calls, and similarly, for string t, we can make at most **length(t)** recursive calls, which are also dependent on each other's solution. Hence, our result can be directly stored in the form of a 2-dimensional array of size **(length(s)+1) * (length(t) + 1)** as for string s, we have **0 to length(s)** possible combinations, and the same goes for string t.

So for every index 'i' in string s and 'j' in string t, we will choose one of the following two options:

1. If the character **s[i]** matches **t[j]**, the length of the common subsequence would be one plus the length of the common subsequence till the **i-1** and **j-1** indexes in the two respective strings.
2. If the character **s[i]** does not match **t[j]**, we will take the longest subsequence by either skipping **i-th or j-th character** from the respective strings.

Hence, the answer stored in the matrix will be the LCS of both strings when the length of string s will be 'i' and the length of string t will be 'j'.

Hence, we will get the final answer at the position `matrix[length(s)][length(t)]`.
Moving to the code:

```

public int lcs_mem(String s, String t, int[][][] output) {
    int m = s.length();
    int n = t.length();

    // Base case
    if(m == 0 || n == 0) {
        return 0;
    }

    // Check if ans already exists
    if(output[m][n] != -1) {
        return output[m][n];
    }

    int ans;
    // Recursive calls
    if(s[0] == t[0]) {
        ans = 1 + lcs_mem(s.substring(1), t.substring(1), output);
    }
    else {
        int a = lcs_mem(s.substring(1), t, output);
        int b = lcs_mem(s, t.substring(1), output);
        int c = lcs_mem(s.substring(1), t.substring(1), output);
        ans = Math.max(a, Math.max(b, c));
    }

    // Save your calculation
    output[m][n] = ans;

    // Return ans
  
```

```

        return ans;
    }

public int lcs_mem(String s, String t) {
    int m = s.length();
    int n = t.length();
    int[][] output = new int[m+1][n+1];
    for(int i = 0; i <= m; i++) {
        for(int j = 0; j <= n; j++) {
            output[i][j] = -1; // Initializing the 2D array with -1
        }
    }
    return lcs_mem(s, t, output);
}

```

Now, converting this approach into the **DP** code:

```

public int lcs_DP(String s, String t) {
    int m = s.length();
    int n = t.length();
    // declaring a 2D array of size m*n
    int[][] output = new int[m+1][n+1];

    // Fill 1st row
    for(int j = 0; j <= n; j++) { // as if string t is empty, then the
        output[0][j] = 0;           // lcs(s, t) = 0
    }

    // Fill 1st col
    for(int i = 1; i <= m; i++) { // as if string s is empty, then the
        output[i][0] = 0;           // lcs(s, t) = 0
    }

    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            // Check if 1st char matches
            if(s[m-i] == t[n-j]) {
                output[i][j] = 1 + output[i-1][j-1];
            }
            else {
                int a = output[i-1][j];
                int b = output[i][j-1];

```

```

        int c = output[i-1][j-1];
        output[i][j] = Math.max(a, Math.max(b, c));
    }
}
return output[m][n];      // final answer
}

```

Time Complexity: We can see that the time complexity of the DP and memoization approach is reduced to **O(m*n)** where **m** and **n** are the lengths of the given strings.

Edit Distance

Problem statement: Given two strings s and t of lengths m and n respectively, find the Edit Distance between the strings. Edit Distance of two strings is the minimum number of steps required to make one string equal to another. To do so, you can perform the following three operations only :

- Delete a character
- Replace a character with another one
- Insert a character

Example 1:

s1 = "but"
s2 = "bat"

Answer: 1

Explanation: We just need to replace 'a' with 'u' to transform s2 to s1.

Example 2:

s1 = "cbda"
s2 = "abdca"
Answer: 2

Explanation: We just need to replace the first 'a' with 'c' and delete the second 'c'.

Example 3:

s1 = "ppspqr"

s2 = "passpot"

Answer: 3

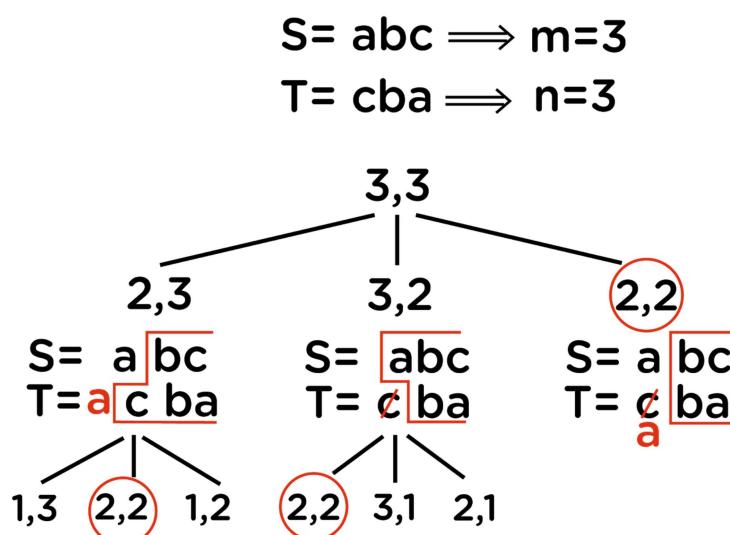
Explanation: We just need to replace first 'a' with 'p', 'o' with 'q', and insert 'r'.

Approach: Let's think about this problem using **recursion** first. We need to apply each of the three operations on each character of s2 to make it similar to s1 and then find the minimum among them.

Let's assume index1 and index2 point to the current indexes of s1 and s2 respectively, so we have two options at every step:

1. If the strings have the same character, we can recursively match for the remaining lengths of the strings.
2. If the strings do not match, we start three new recursive calls representing the three edit operations, as mentioned in the problem statement. Consider the minimum count of operations among the three recursive calls.

Let's dry run the code:



From here, it is clear that the time complexity is again exponential, which is $O(3^{m+n})$, where m and n are the lengths of the given strings.

Also, we can see the overlapping/repeated recursive calls(for example: (2,2) is one of them), which means this problem can be further solved using **memoization**.

This problem is somehow similar to the LCS problem. We will be using a similar approach to solve this problem too. The answer to each recursive call will be stored in a 2-Dimensional array of size $(m+1)*(n+1)$, and the final solution will be obtained at index (m,n) as each cell will be storing the answer for the given m length of s1 and n length of s2. Try to code it yourself.

Time Complexity: As there are $(m+1)*(n+1)$ number of unique calls, hence the time complexity becomes $O(m*n)$, which is better than the recursive approach.

Let's move on to the DP approach...

We have already discussed the basic requirements like output array size, final output's position, and the value stored at each position of the output array in the memoization approach. We have already figured out that this problem is similar to the LCS question. So try to code it yourself. Refer to the solution tab for the same.

Problem Statement: Knapsack

Given the weights and values of 'N' items, we are asked to put these items in a knapsack, which has a capacity 'C'. The goal is to get the maximum value from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

For example:

Items: {Apple, Orange, Banana, Melon}

Weights: {2, 3, 1, 4}

Values: {4, 5, 3, 7}

Knapsack capacity: 5

Possible combinations that satisfy the given conditions are:

Apple + Orange (total weight 5) => 9 value

Apple + Banana (total weight 3) => 7 value

Orange + Banana (total weight 4) => 8 value

Banana + Melon (total weight 5) => 10 value

This shows that **Banana + Melon** is the best combination, as it gives us the maximum value, and the total weight does not exceed the capacity.

Approach: First-of-all, let's discuss the brute-force-approach, i.e., the **recursive approach**. There are two possible cases for every item, either to put that item into the knapsack or not. If we consider that item, then its value will be contributed towards the total value, otherwise not. To figure out the maximum value obtained by maintaining the capacity of the knapsack, we will call recursion over these two cases simultaneously, and then will consider the maximum value obtained out of the two.

If we consider a particular weight 'w' from the array of weights with value 'v' and the total capacity was 'C' with initial value 'Val', then the remaining capacity of the knapsack becomes 'C-w', and the value becomes 'Val + v'.

Let's look at the recursive code for the same:

```

public int knapsack(int[] weight, int[] values, int i, int n, int maxWeight) {
    // Base case : if the size of array is 0 or we are not able to add
    // any more weight to the knapsack
    if(n == i || maxWeight == 0) {
        return 0;
    }

    // If the particular weight's value extends the limit of knapsack's
    // remaining capacity, then we have to simply skip it
    if(weight[i] > maxWeight) {
        return knapsack(weight, values, i+1, n, maxWeight);
    }

    // Recursive calls
    //1. Considering the weight
    int x = knapsack(weight, values, i+1, n, maxWeight - weight[i]) +
           values[i];
    // 2. Skipping the weight and moving forward
    int y = knapsack(weight, values, i+1, n, maxWeight) + values[i];

    // finally returning the maximum answer among the two
    return Math.max(x, y);
}
  
```

Now, the memoization and DP approach is left for you to solve. For the code, refer to the solution tab of the same. Also, figure out the time complexity for the same by running the code over some examples and by dry running it.

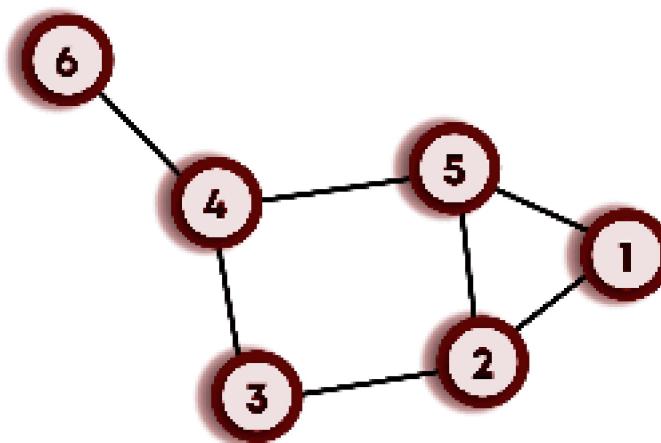
Graphs- 1

Introduction

A **graph** is a pair $G = (V, E)$, where V is a set whose elements are called **vertices**, and E is a set of two sets of vertices, whose elements are called **edges**.

The vertices x and y of an edge $\{x, y\}$ are called the **endpoints** of the edge. The edge is said to **join** x and y and to be **incident** on x and y . A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use trees to store it. In such cases, we will be using graphs. Refer to the figure for better understanding.



Relationship between trees and graphs:

- A tree is a special type of graph in which we can reach any node to any other node using some path, unlike the graphs where this condition may or may not hold.
- A tree does not have any cycles in it whereas a graph may or may not contain a cycle.
- A **cyclic graph** is a graph containing at least one cycle in a graph. If in a graph any combination of edges form a closed or rounded circuit then we say that there is a cycle in the graph.

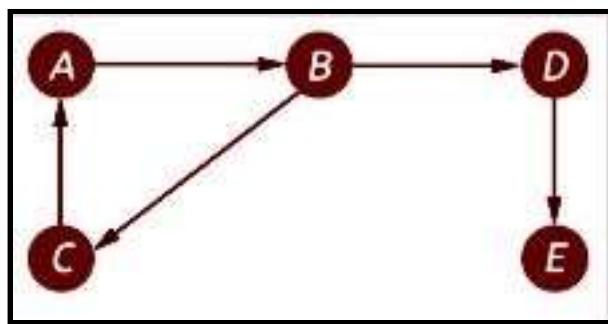
Graphs Terminology

- Nodes are named **vertices**, and the connections between them are called **edges**.
- Two vertices are said to be **adjacent** if there exists a direct edge connecting them.
- The **degree** of a node is defined as the number of edges that are incident to it.
- A **path** is a collection of edges through which we can reach from one node to the other node in a graph.
- A graph is said to be **connected** if there is a path between every pair of vertices.
- If the graph is not connected, then all the connected subsets of the graphs are called **connected components**. Each component is connected within the self, but two different components of a graph are never connected.
- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be **(N-1)**, where **N** is the number of nodes.

- In a complete graph (where each node is connected to every other node by a direct edge), there are nC_2 number of edges means $(N * (N-1)) / 2$ edges, where n is the number of nodes.
- This is the maximum number of edges that a graph can have.
- Hence, if an algorithm works on the terms of edges, let's say $O(E)$, where E is the number of edges, then in the worst case, the algorithm will take $O(N^2)$ time, where N is the number of nodes.

Graphs Implementation

Suppose the graph is as follows:



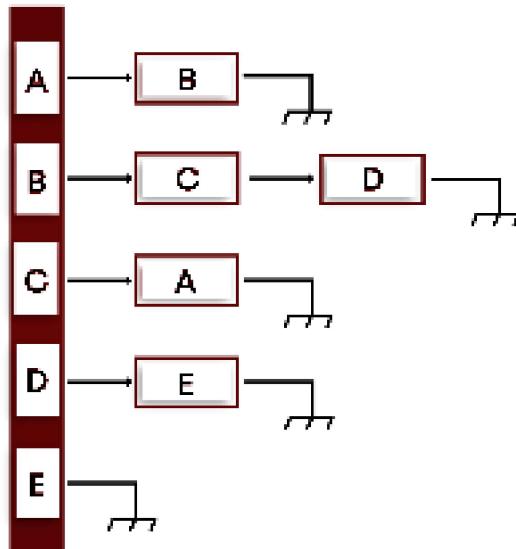
There are the following ways to implement a graph:

- 1. Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred to check for a particular edge connecting two nodes; we have to traverse the complete array leading to $O(n^2)$ time complexity in the worst case. Pictorial representation for the above graph using the edge list is given below:



- 2. Adjacency list:** We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. Now to check for a particular edge, we can take any one of the nodes and then check

in its list if the target node is present or not. This will take $O(n)$ work to figure out a particular edge.



3. Adjacency matrix: Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j . It is the most reliable method to implement a graph in terms of ease of implementation. We will be using the same throughout the session. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each node stores only those nodes that are directly connected to them. For the above graph, the adjacency matrix looks as follows:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

DFS - Adjacency matrix

Here, if we have n vertices(labelled: 0 to n-1). Then we will be asking the user for the number of edges. We will run the loop from 0 to the number of edges, and at each iteration, we will take input for the two connected nodes and correspondingly update the adjacency matrix. Let's look at the code for better understanding.

```

public static void print(int[][] edges, int n, int sv, boolean[] visited){
    System.out.println(sv);
    visited[sv] = true;      // marked the starting vertex true
    for(int i=0; i<n; i++){// Running the loop over all n nodes and checking
                           // if there is an edge between sv and i
        if(i==sv){
            continue;
        }
        if(edges[sv][i]==1){ // As the edge is found, we then checked if the
                           // node i was visited or not
            if(visited[i]){
                continue;
            }
            print(edges, n, i, visited); // Otherwise, recursively called over
                           // node i taking it as starting vertex
        }
    }
}

public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    int n = s.nextInt();                                // Number of nodes
    int e = s.nextInt();                                // Number of edges

    int[][] edges = new int[n][n];                      // adjacency matrix of size n*n
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            edges[i][j]=0;   // 0 indicates that there is no edge between i and j
        }
    }

    for(int i=0; i<e; i++){
        int f = s.nextInt();
    }
}

```

```

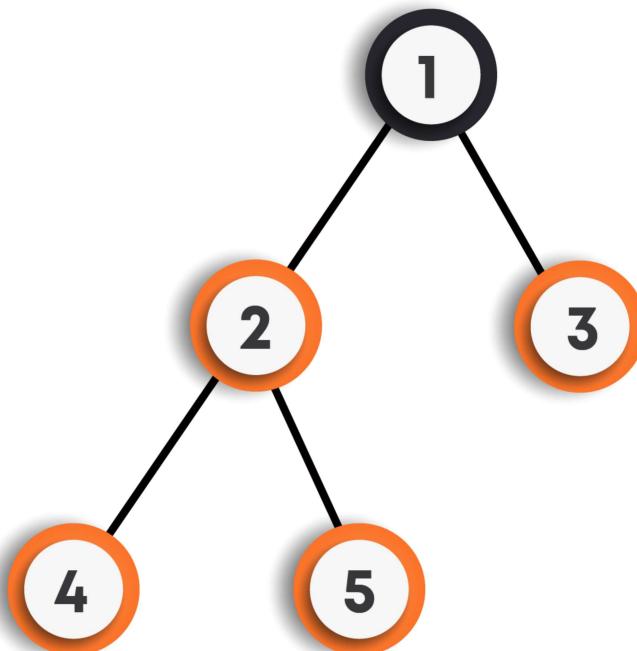
int s = s.nextInt();      // Nodes f and s having edges between them
edges[f][s]=1;           // marking f to s as 1
edges[s][f]=1;           // also, marking s to f as 1
}

boolean[] visited = new boolean[n]; // this is used to keep the track
                                   // of nodes if we have visited them or not.
for(int i=0; i<n; i++){
    visited[i]=false; // Marking all nodes as false which means not visited
}

print(edges, n, 0, visited);     // starting vertex is taken as 0
}

```

Let's take an example graph:

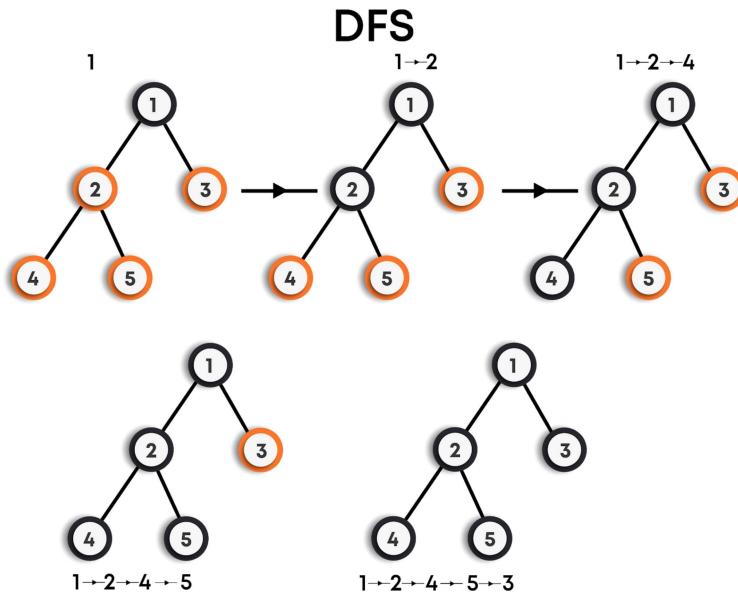


On dry running the above code, the output will be 1 2 4 5 3.

Here, we are starting from a node, going in one direction as far as we can, and then we return and do the same on the previous nodes. This method of graph traversal is known as the **depth-first search (DFS)**. As the name suggests, this algorithm

goes into the depth first and then recursively does the same in other directions.

Follow the figure below, for step-by-step traversal using DFS.

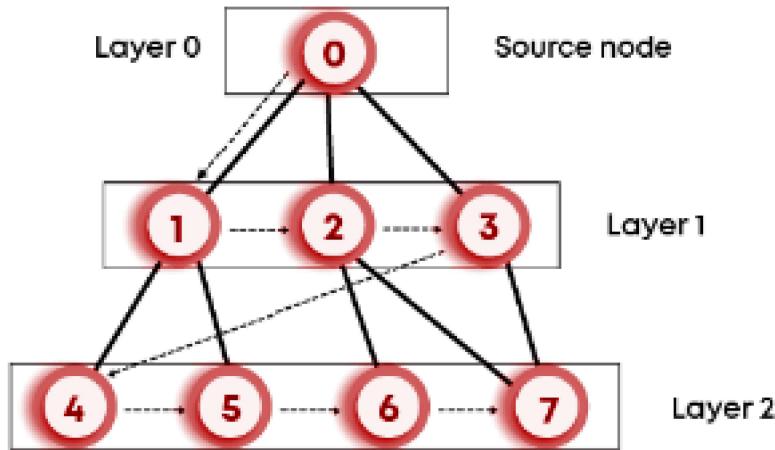


BFS Traversal

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

As the name suggests:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.



This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

Let's look at the code below:

```

public void printBFS(int[][] edges, int n, int sv, boolean[] visited) {
    Queue<Integer> pendingVertices = new Queue<>();           // queue
    pendingVertices.push(sv);                                // starting vertex directly pushed
    visited[sv] = true;
    while (!pendingVertices.empty()) {          // until the size of queue is not 0
        int currentVertex = pendingVertices.top(); // stored the top of queue
        pendingVertices.pop(); // deleted that top element
        System.out.print(currentVertex + " ");
        for (int i = 0; i < n; i++) {      // now checked for its vertices
            if (i == currentVertex) {
                continue;
            }
            if (edges[currentVertex][i] == 1 && !visited[i]) {
                pendingVertices.push(i); // if found, then inserted in queue
                visited[i] = true;
            }
        }
    }
}

```

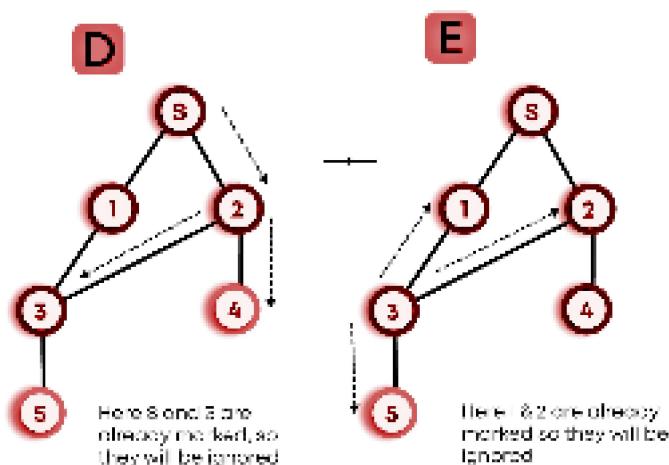
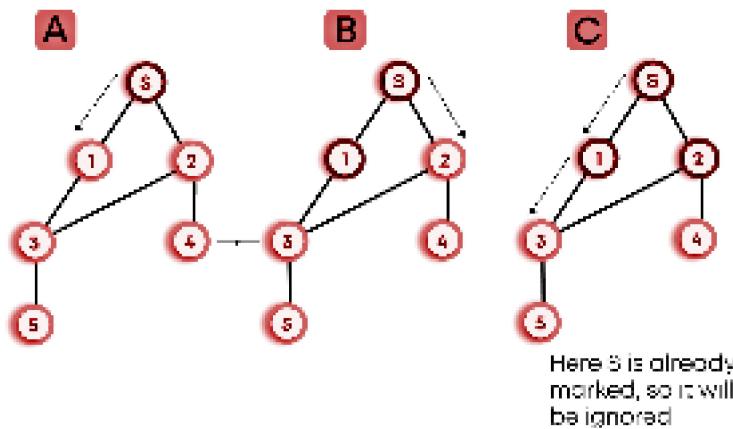
```

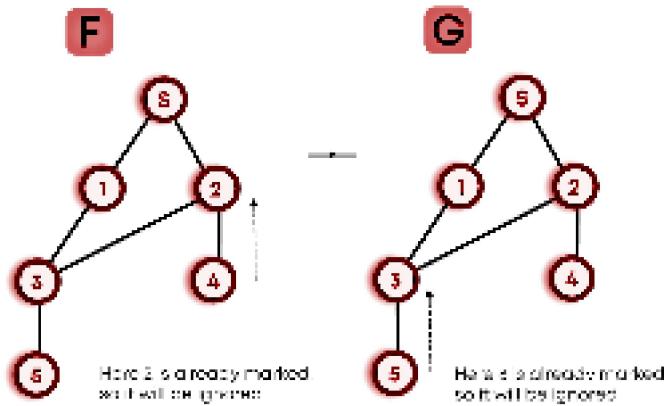
public void BFS(int[][] edges, int n) {
    boolean[] visited = new boolean[n]; // visited array
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    printBFS(edges, n, 0, visited);      // starting vertex = 0
}

```

Consider the dry run over the example graph below for a better understanding of the same:





BFS & DFS for Disconnected Graph

Till now, we have assumed that the graph is connected. For the disconnected graph, there will be a minor change in the above codes. Just before calling out the print functions, we will run a loop over each node and check if that node is visited or not. If not visited, then we will call a print function over that node, considering it as the starting vertex. In this way, we will be able to cover up all the nodes of the graph.

Consider the same for the BFS function. Just replace this function in the above code to make it work for the disconnected graph too.

```
public void BFS(int[][] edges, int n) {
    boolean[] visited = new boolean[n]; // visited array
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    for (int i = 0; i < n; i++) { // run a loop over each node
        if (!visited[i]) { // if a node is not visited, then called print()
            printBFS(edges, n, i, visited); //on it taking it as starting vtx
        }
    }
}
```

Has Path

Problem statement: Given an undirected graph $G(V, E)$ and two vertices v_1 and v_2 (as integers), check if there exists any path between them or not. Print true or false. V is the number of vertices present in graph G , and vertices are numbered from 0 to $V-1$. E is the number of edges present in graph G .

Approach: This can be simply solved by considering the vertex v_1 as the starting vertex and then run either BFS or DFS as per your choice, and while traversing if we reach the vertex v_2 , then we will simply return true, otherwise return false.

This problem has been left for you to try yourself. For code, refer to the solution tab of the same.

Get Path - DFS

Problem statement: Given an undirected graph $G(V, E)$ and two vertices v_1 and v_2 (as integers), find and print the path from v_1 to v_2 (if exists). Print nothing if there is no path between v_1 and v_2 . Find the path using DFS and print the first path that you encountered irrespective of the length of the path. V is the number of vertices present in graph G , and vertices are numbered from 0 to $V-1$. E is the number of edges present in graph G . Print the path in reverse order. That is, print v_2 first, then intermediate vertices, and v_1 at last.

Example: Suppose the given input is:

```
4 4
0 1
0 3
1 2
2 3
1 3
```

The output should be:

```
3 0 1
```

Explanation: Here, $v1 = 1$ and $v2 = 3$. The connected vertex pairs are $(0, 1)$, $(0, 3)$, $(1, 2)$ and $(2, 3)$. So, according to the question, we have to print the path from vertex $v1$ to $v2$ in reverse order using DFS only; hence the path comes out to be $\{3, 0, 1\}$.

Approach: We have to solve this problem by using DFS. Suppose, if the start and end vertex are the same, then we simply need to put the start in the solution array and return the solution array. If this is not the case, then from the start vertex, we will call DFS on the direct connections of the same. If none of the paths leads to the end vertex, then we do not need to push the start vertex as it is neither directly nor indirectly connected to the end vertex, hence we will simply return NULL. In case any of the neighbors return a non-null entry, it means that we have a path from that neighbor to the end vertex, hence we can now insert the start vertex into the solution array.

Try to code it yourself, and for the answer, refer to the solution tab of the same.

Get Path - BFS

Problem: It is the same problem as the above, just we have to code the same using BFS.

Approach: Using BFS will provide us the shortest path between the two vertices. We will use the queue over here and do the same until the end vertex gets inserted into the queue. Here, the problem is how to figure out the node, which led us to the end vertex. To overcome this, we will be using a map. In the map, we will store the resultant node as the index, and its key will be the node that led it into the queue.

For example: If the graph was such that 0 was connected to 1 and 0 was connected to 2, and currently, we are on node 0 such that node 1 and node 2 are not visited. So our map will look as follows:

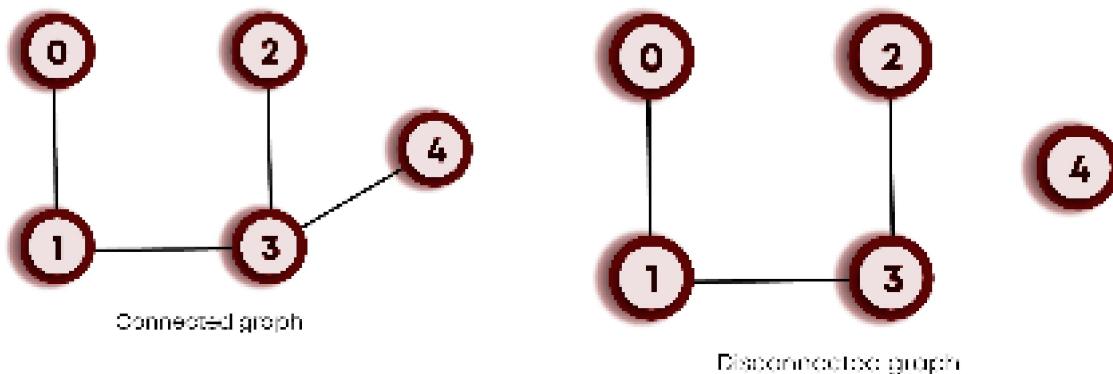
1	0
2	0

This way, as soon as we reach the end vertex, we can figure out the nodes by running the loop until we reach the start vertex as the key value of any node.

Try to code it yourselves, and for the solution, refer to the specific tab of the same.

Is connected?

Problem statement: Given an undirected graph $G(V, E)$, check if the graph G is a connected graph or not. V is the number of vertices present in graph G , and vertices are numbered from 0 to $V-1$. E is the number of edges present in graph G .



Example 1: Suppose the given input is:

```
4 4
0 1
0 3
1 2
2 3
1 3
```

The output should be: **true**

Explanation: As the graph is connected, so according to the question, the answer will be true.

Example 2: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be: **false**

Explanation: The graph is not connected, even though vertices 0,1, and 3 are connected, but there isn't any path from vertices 0,1,3 to vertex 2. Hence, according to the question, the answer will be false.

Approach: This is very start-forward. Take any vertex as the starting vertex as traverse the graph using either DFS or BFS. In the end, check if all the vertices are visited or not. If not, it means that the node was not connected to the starting vertex, which means it is a disconnected graph. Otherwise, it is a connected graph. Try to code it yourselves, and for the code, refer to the solution tab of the same.

Problem statement: Return all Connected Components

Given an undirected graph $G(V, E)$, find and print all the connected components of the given graph G . V is the number of vertices present in graph G , and vertices are numbered from 0 to $V-1$. E is the number of edges present in graph G .

You need to take input in the main and create a function that should return all the connected components. And then print them in the main, not inside a function.

Print different components in a new line. And each component should be printed in increasing order (separated by space). The order of different components doesn't matter.

Example: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be:

```
0 1 3
2
```

Explanation: As we can see that $\{0, 1, 3\}$ is one connected component, and $\{2\}$ is the other one. So, according to the question, we just have to print the same.

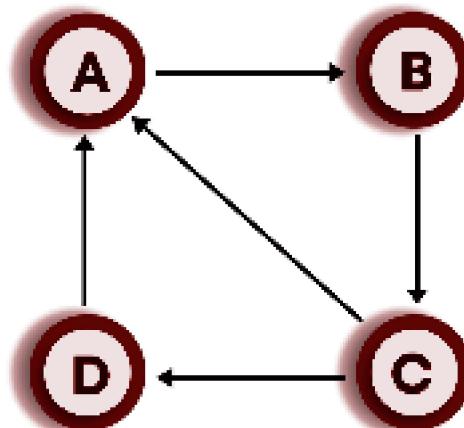
Approach: For this problem, start from vertex 0 and traverse until vertex $n-1$. If the vertex is not visited, then run DFS/BFS on it and keep track of all the connected vertices through that node. This way, we will get all the distinct connected components, and we can print them at last.

This problem is left for you to solve. For the code, refer to the solution tab of the same.

Weighted and Directed Graphs

There are two more variations of the graphs:

Directed graphs: These are generally required when we have one-way routes. Suppose you can go from node A to node B, but you cannot go from node B to node A. Another example could be social media (like Twitter) if you are following someone, it does not mean that they are following you too.



To implement these, there is a small change in the implementation of indirect graphs. In indirect graphs, if there was an edge between node i and j, then we did:

```

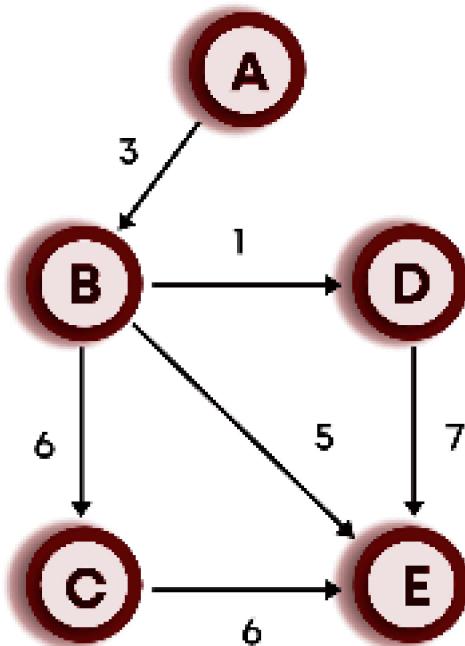
edges[i][j] = 1;
edges[j][i] = 1;
  
```

But, in the case of a directed graph, we will just do the following:

```

edges[i][j] = 1;
  
```

Weighted graphs: These generally mean that all the edges are not equal, which means somehow, each edge has some weight assigned to it. This weight can be the length of the road connecting the cities or many more.



To implement this, in the edges matrix, we will assign a weight to connected nodes instead of putting it 1 at that position. For example: If node i and j are connected, and the weight of the edge connecting them is 5, then `edges[i][j] = 5`.

Graphs- 2

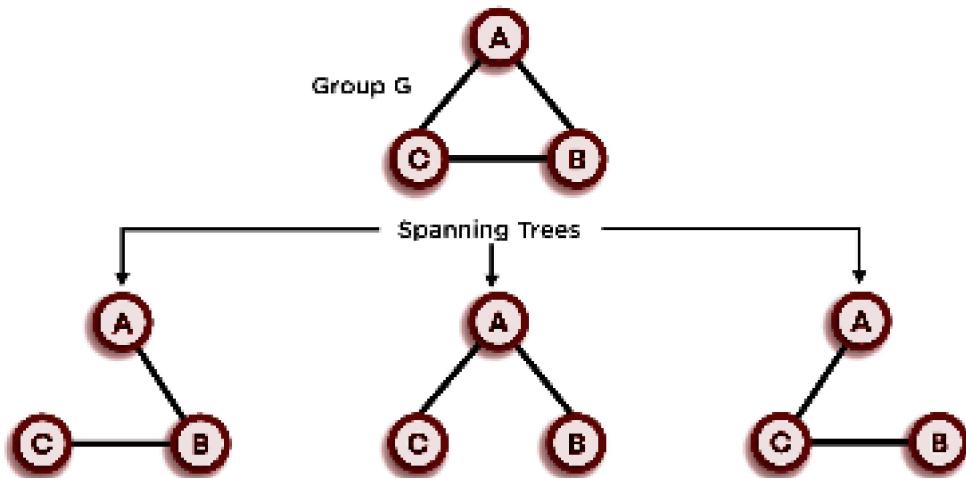
MST(Minimum Spanning Tree) & Kruskal's Algorithm

As discussed earlier, a tree is a graph, which:

- Is always connected.
- Contains no cycle.

If we are given an undirected and connected graph, a **spanning tree** means a tree that contains all the vertices of the same. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



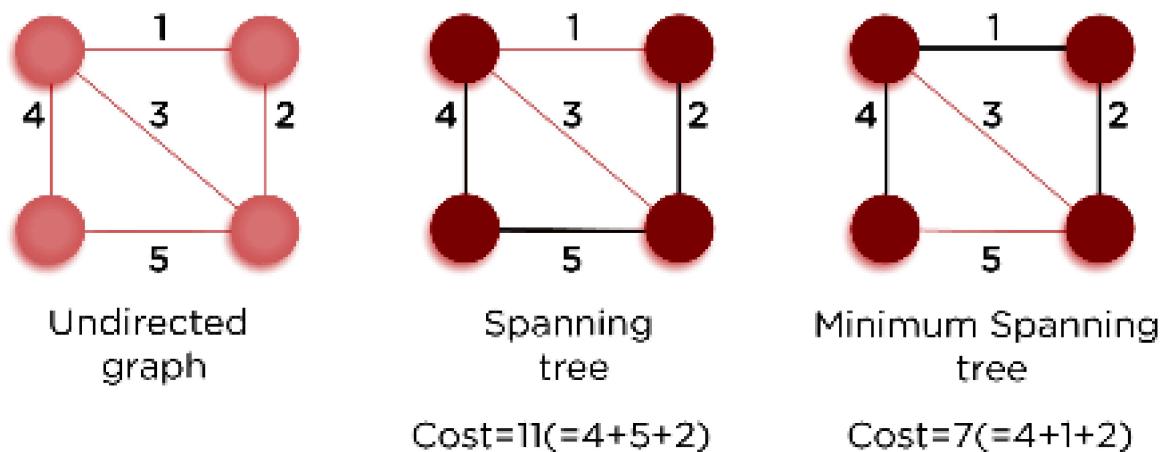
If there are n vertices and e edges in the graph, then any spanning tree corresponding to that graph contains n vertices and $n-1$ edges.

Properties of spanning trees:

- A connected and undirected graph can have more than one spanning tree.
- The spanning tree is free of loops, i.e., it is acyclic.
- Removing any one of the edges will make the graph disconnected.
- Adding an extra edge to the spanning tree will create a loop in the graph.

Minimum Spanning Tree(MST) is a spanning tree with weighted edges.

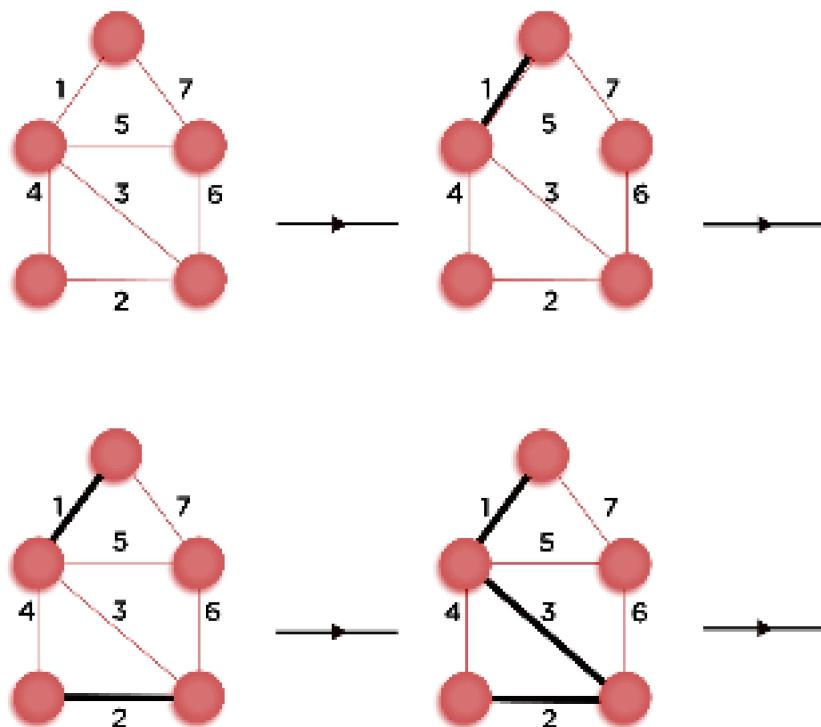
In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding.

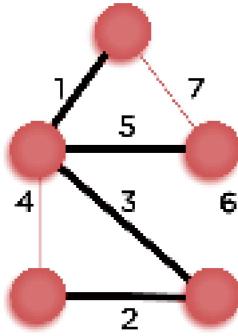


Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the same. This process will continue until the count of edges in the MST reaches $n-1$. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the nodes A and B, if both nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of two vertices, if any one of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.

Let's think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two vertices v1 and v2, otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is $O(E+V)$, where E is the number of edges in the graph and, V is the number of vertices. So, for $(n-1)$ edges, this function will run $(n-1)$ times, leading to bad time complexity, as in the worst case, $E = V^2$.

Now, moving on to a better approach for cycle detection in the graph.

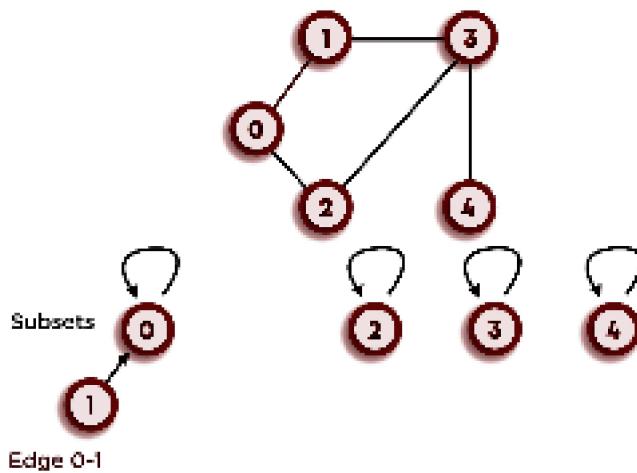
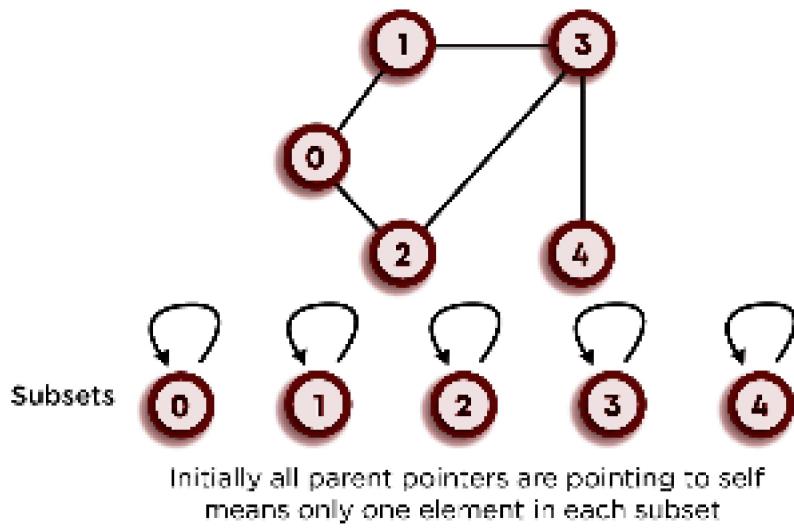
Union-Find Algorithm:

Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not. If not, then it is safe to add that edge to the MST.

Following the steps of the algorithm:

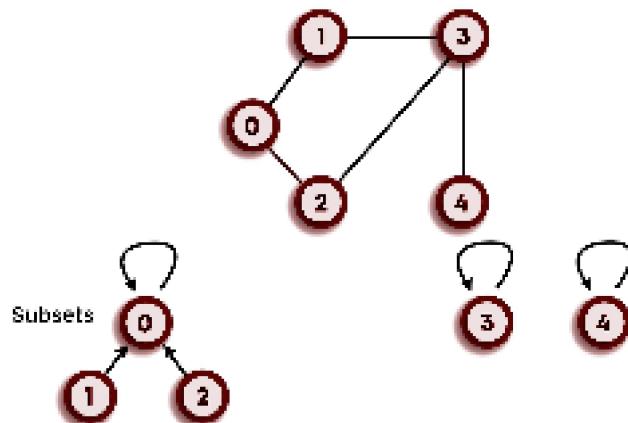
- We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to $n-1$.
- We will maintain a parent array specifying the parent vertex of each of the vertex of the graph. Initially, as each vertex belongs to a different disjoint set (connected component), hence each vertex will be its parent.
- Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge.
- Otherwise, we can add that edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).

Look at the following example, for better understanding:



Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

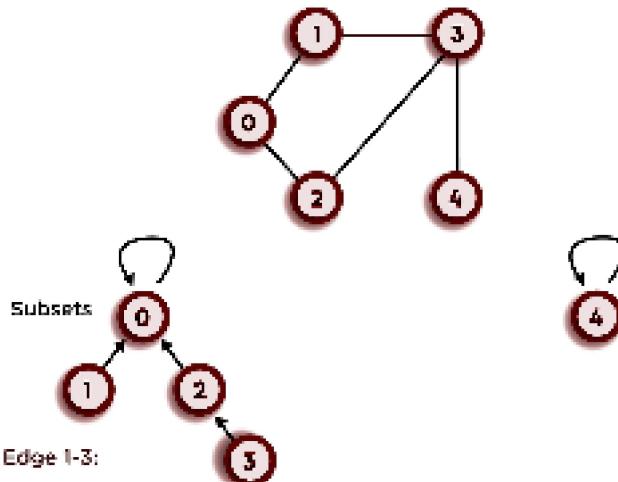
Union: Make 0 as the parent of 1. Updated set is {01}. 0 is the set representative since 0 is parent for itself.



Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

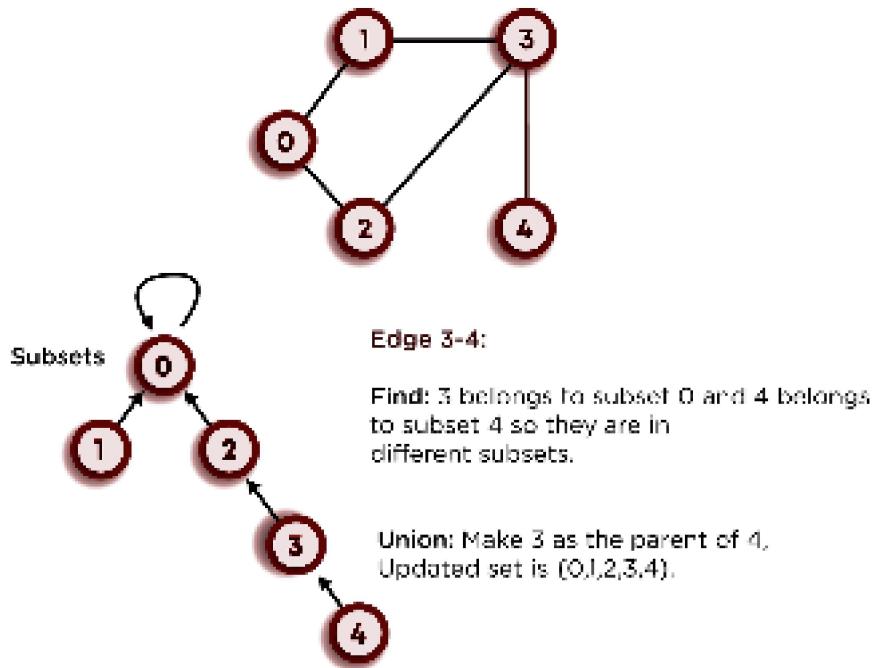
Union: Make 0 as the parent of 2. Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.



Edge 1-3:

Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3. Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.



Note: While finding the parent of the vertex, we will be finding the topmost parent (Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes $O(V)$ for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Kruskal's Algorithm: Implementation

Till now, we have studied the logic behind Kruskal's algorithm for finding MST. Now, let's discuss how to implement it in code.

Consider the code below and follow the comments for a better understanding.

```

class Edge {      // Class that store values for each vertex
    int source;
    int dest;
    int weight;
}

class Sortbyweight implements Comparator<Edge> {
    public int compare(Edge a, Edge b){
        return a.weight - b.weight;
    }
}

class Main{

    public static int findParent(int v, int[] parent) {
        // Function to find the parent of a vertex
        if (parent[v] == v) { // Base case, when a vertex is parent of itself
            return v;
        }
        // Recursively called to find the topmost parent of the vertex.
        return findParent(parent[v], parent);
    }

    public static void kruskals(Edge[] input, int n, int E) {
        sort(input, new Sortbyweight());      // In-built sort function:
        // Sorts the edges in increasing order of their weights

        Edge[] output = new Edge[n-1]; // Array to store final edges of MST
        int[] parent = new int[n];

        // Parent arr initialized with their indexes
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }

        int count = 0; // To maintain the count of number of edges in the MST
        int i = 0;           // Index to traverse over the input array
        while (count != n - 1) { // As the MST contains n-1 edges.
            Edge currentEdge = input[i];
    }
}

```

```

        // Figuring out the parent of each edge's vertices
        int sourceParent = findParent(currentEdge.source, parent);
        int destParent = findParent(currentEdge.dest, parent);
        // If their parents not equal, then we added that edge to output
        if(sourceParent != destParent) {
            output[count] = currentEdge;
            count++;           // Increased the count
            parent[sourceParent] = destParent; // Updated parent array
        }
        i++;
    }
    // Finally, printing the MST obtained.
    for (int i = 0; i < n-1; i++) {
        if(output[i].source < output[i].dest) {
            System.out.println(output[i].source + " " +
                               output[i].dest + " " + output[i].weight);
        }
        else {
            System.out.println(output[i].dest + " " +
                               output[i].source + " " + output[i].weight);
        }
    }
}

public static void main (String[] args){
    Scanner s = new Scanner(System.in);
    int n = s.nextInt();
    int E = s.nextInt();

    Edge[] input = new Edge[E];

    for (int i = 0; i < E; i++) {
        int s = s.nextInt();
        int d = s.nextInt();
        int w = s.nextInt();
        input[i].source = s;
        input[i].dest = d;
        input[i].weight = w;
    }

    kruskals(input, n, E);
}

```

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n , and the total number of edges is E)

- Take input in the array of size E .
- Sort the input array based on edge-weight. This step has the time complexity of $O(E \log(E))$.
- Pick $(n-1)$ edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle detection, in the worst-case time complexity of E edges will be $O(E.n)$, as discussed earlier.

Hence, the total time complexity of Kruskal's algorithm becomes $O(E \log(E) + n.E)$.

This time complexity is bad and needs to be improved.

We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named **Union by Rank and Path**

Compression. You need to explore this on yourselves. The basic idea in these algorithms is that we will be avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to $O(\log(E))$.

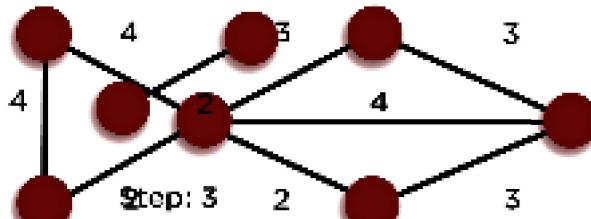
Prim's Algorithm

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the

unselected set of vertices. This process is repeated until we have inserted a total of $(n-1)$ edges in the MST.

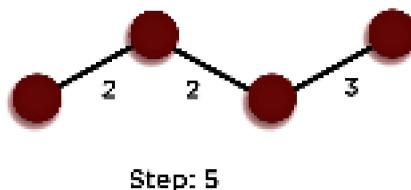
Consider the following example for a better understanding.



Choose the shortest edge from this vertex add it

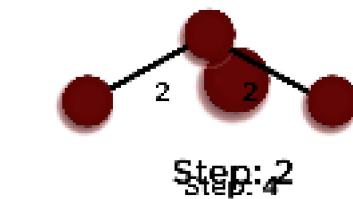
Step: 1

Start with a weighted graph



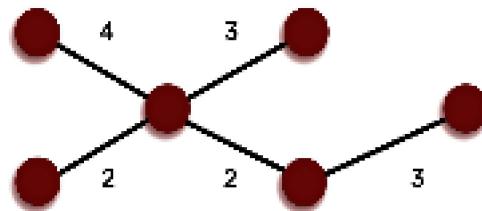
Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 2

Choose the nearest vertex in the solution



Step: 6

Repeat until you have a spanning tree

Implementation:

- We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).
- The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.
- We will mark the vertex 0 as visited and the rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.

- Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Let's look at the code now:

```

public static int findMinVertex(int[] weights, boolean[] visited, int n) {

    int minVertex = -1; // -1 means there is no vertex till now
    for (int i = 0; i < n; i++) {
        // Conditions: the vertex must be unvisited and either minVertex value
        // is -1 or if minVertex has some vertex to it, then weight of
        // currentvertex should be less than the weight of the minVertex.
        if (!visited[i] && (minVertex == -1 || weights[i] <
                               weights[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

public static void prims(int[][] edges, int n) {

    int[] parent = new int[n];
    int[] weights = new int[n];
    boolean[] visited = new boolean[n];
    // Initially, the visited array is assigned to false and weights
    // array to infinity.
    for(int i = 0; i < n; i++) {
        visited[i] = false;
        weights[i] = Integer.MAX_VALUE;
    }
    // Values assigned to vertex 0. (the selected starting vertex to begin
    // with)
    parent[0] = -1;
    weights[0] = 0;

    for (int i = 0; i < n-1; i++) {
        // Find min vertex
        int minVertex = findMinVertex(weights, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++) {
            if (edges[minVertex][j] != null && !visited[j] &&
                weights[j] > edges[minVertex][j]) {
                parent[j] = minVertex;
                weights[j] = edges[minVertex][j];
            }
        }
    }
}

```

```

        if(edges[minVertex][j] != 0 && !visited[j]) {
            if(edges[minVertex][j] < weights[j]) {
                // updating weight array and parent array
                weights[j] = edges[minVertex][j];
                parent[j] = minVertex;
            }
        }
    }
}

// Final MST printed
for (int i = 0; i < n; i++) {
    if (parent[i] < i) {
        System.out.println(parent[i] + " " + i + " " + weights[i]);
    }
    else {
        System.out.println(i + " " + parent[i] + " " + weights[i]);
    }
}
}

public static void main (String[] args){
    Scanner s = new Scanner(System.in);
    int n = s.nextInt();
    int e = s.nextInt();

    int[][] edges = new int[n][n]; //Adjacency matrix to store the graph

    for (int i = 0; i < e; i++) {
        int f = s.nextInt();
        int s = s.nextInt();
        int weight = s.nextInt();
        edges[f][s] = weight;
        edges[s][f] = weight;
    }

    prims(edges, n);
}
}

```

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- The time complexity for finding the minimum weighted vertex is $O(n)$ for each iteration. So for $(n-1)$ edges, it becomes $O(n^2)$.
- Similarly, for exploring the neighbor vertices, the time taken is $O(n^2)$.

It means the time complexity of Prim's algorithm is $O(n^2)$. We can improve this in the following ways:

- For exploring neighbors, we are required to visit every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.
- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of $O(n^2)$. Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take $O(\log(n))$ time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of $O((n+E)\log(n))$, which is much better than the earlier one. Try to write the optimized code by yourself.

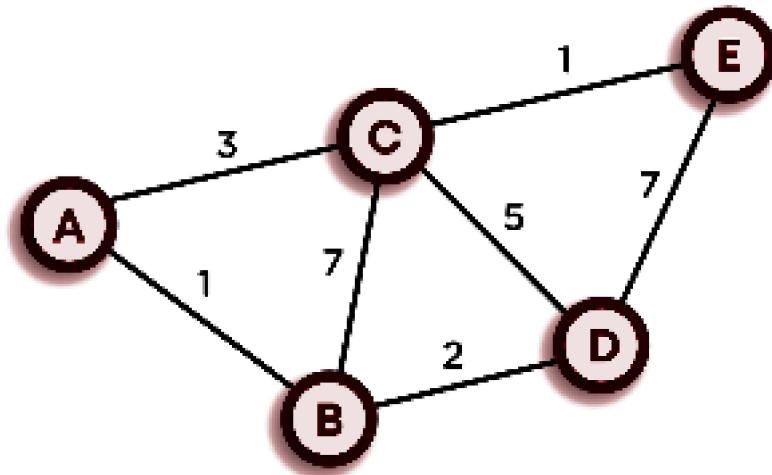
Dijkstra's Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

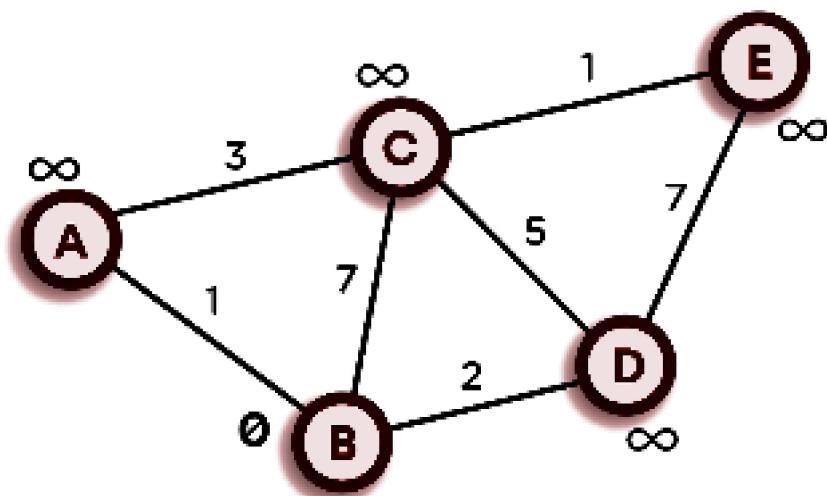
Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

Let's consider the algorithm with an example:

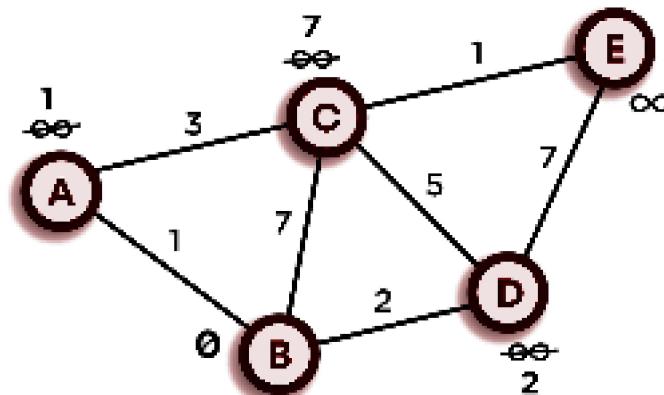
1. We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.



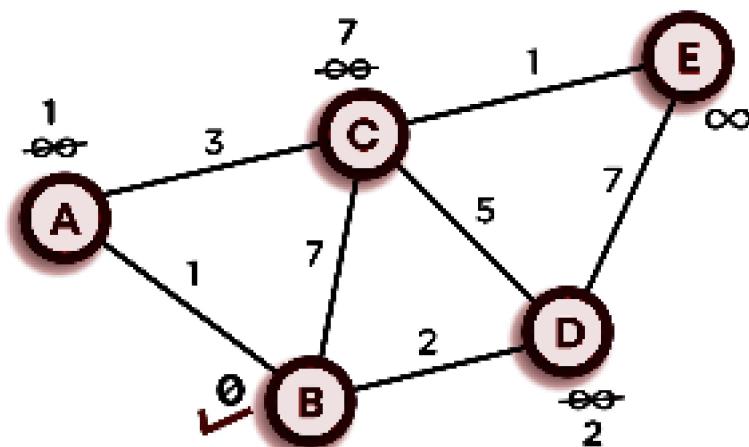
2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.



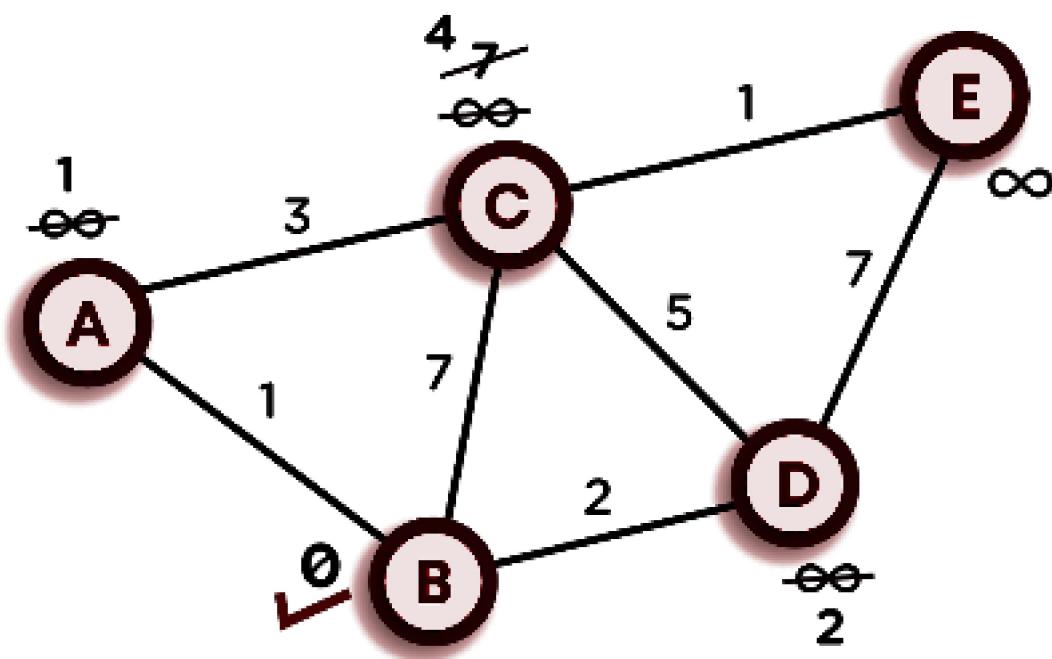
3. Now, we will check for the neighbors of the current node, which are A, B, and D. Now, we will add the minimum cost of the current node to the weight of the edge connecting the current node and the particular neighbor node. For example, for node B, its weight will become $\min(\infty, 0+7) = 7$. This same process is repeated for other neighbor nodes.

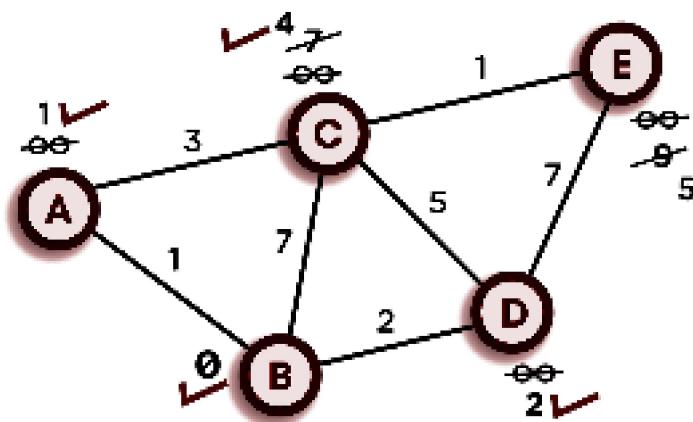
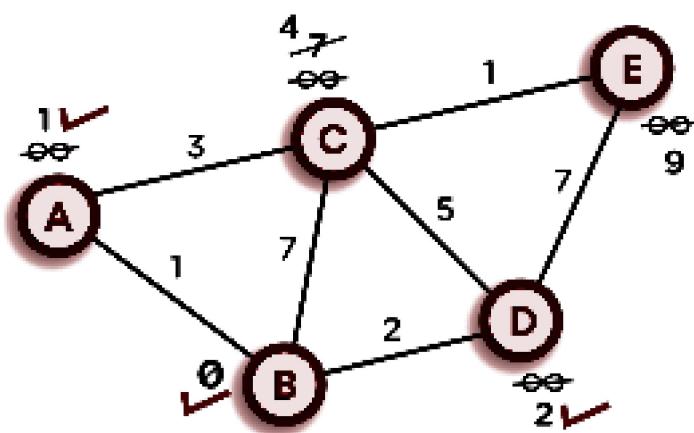
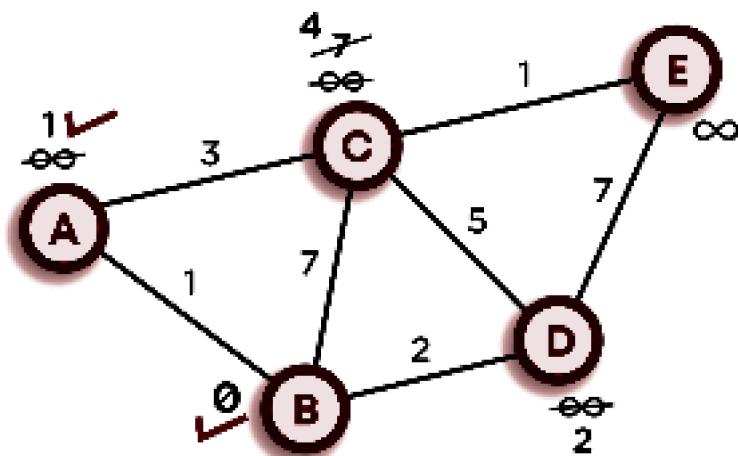


4. Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.

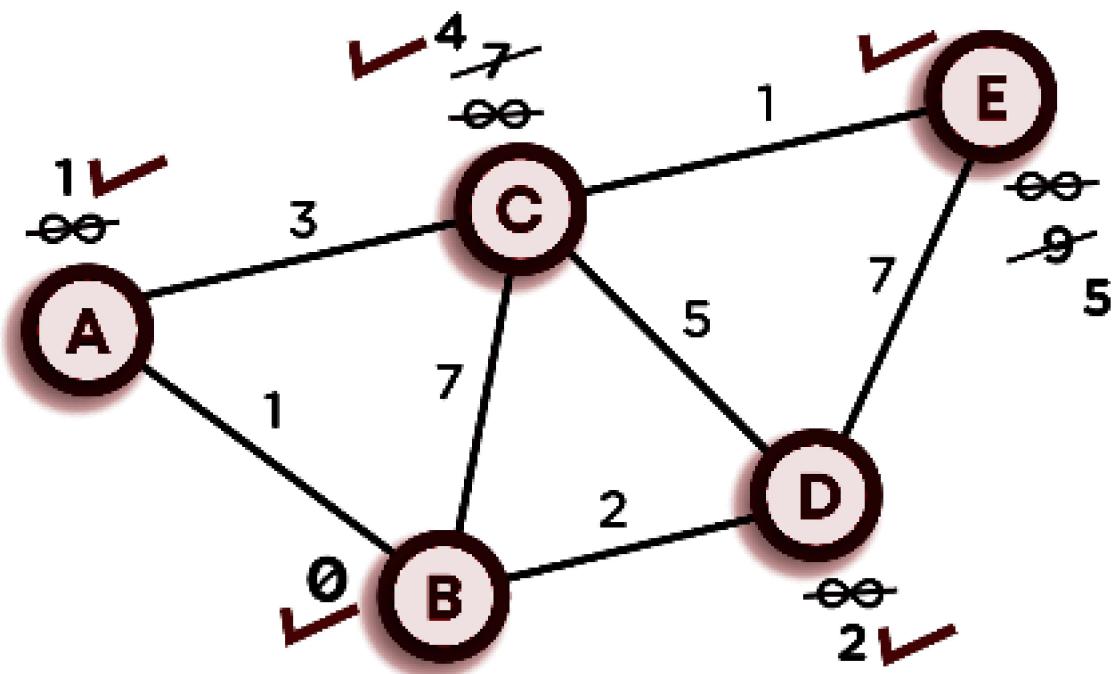


5. After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.
6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:





7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from node C.

Implementation:

Let's look at the code below for a better explanation:(Code is nearly same as that of Prim's algorithm, just a change while updating the distance)

```

public static int findMinVertex(int[] distance, boolean[] visited, int n) {

    int minVertex = -1;
    for (int i = 0; i < n; i++) {
        if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

public static void dijkstra(int[][] edges, int n) {

    int[] distance = new int[n];
    boolean[] visited = new boolean[n];

    for (int i = 0; i < n; i++) {
        visited[i] = false;
        distance[i] = Integer.MAX_VALUE;
    }

    distance[0] = 0; // 0 is considered as the starting node.

    for (int i = 0; i < n-1; i++) {
        // Find min vertex
        int minVertex = findMinVertex(distance, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++) {
            if (edges[minVertex][j] != 0 && !visited[j]) {
                // distance of any node will be the current node's distance + the
                // weight of the edge between them
                int dist = distance[minVertex] + edges[minVertex][j];
                if (dist < distance[j]) { // If required, then updated.
                    distance[j] = dist;
                }
            }
        }
    }
    // Final output of distance of each node with respect to 0
    for (int i = 0; i < n; i++) {
}

```

```
        System.out.println(i + " " + distance[i]);
    }
}

public static void main (String[] args){
    Scanner s = new Scanner(System.in);
    int n = s.nextInt();
    int e = s.nextInt();

    int[][] edges = new int[n][n]; //Adjacency matrix to store the graph

    for (int i = 0; i < e; i++) {
        int f = s.nextInt();
        int s = s.nextInt();
        int weight = s.nextInt();
        edges[f][s] = weight;
        edges[s][f] = weight;
    }

    dijkstra(edges, n);
}
```

Time Complexity of Dijkstra's algorithm:

The time complexity is also the same as that of Prim's algorithm, i.e., $O(n^2)$. This can be reduced by using the same approaches as discussed in Prim's algorithm's content.

OOPS - 4

Introduction

In this lecture we will try to create game logics by playing around different classes. We will create a Tic-Tac-Toe game and we will discuss Othello game. It will be a java console application. You can refer to course videos to understand the game rules.

Tic-Tac-Toe

In this game, two players will be played and you have one print board on the screen where from 1 to 9 numbers will be displayed or you can say it box number. Now, you have to choose X or O for the specific box number. For example, if you have to select any number then for X or O will be shown on the print board, and turn for next will be there. The task is to create a Java program to implement a 3×3 Tic-Tac-Toe game for two players.

How to Play the Game :

- Both the players choose either **X** or **O** to mark their cells.
- There will be a 3×3 grid with numbers assigned to each of the 9 cells.
- The player who chose **X** begins to play first.
- He enters the cell number where he wishes to place **X**.
- Now, both **O** and **X** play alternatively until any one of the two wins.
- **Winning criteria:** Whenever any of the two players has fully filled one row/ column/ diagonal with his symbol (X/ O), he wins and the game ends.

- If neither of the two players wins, the game is said to have ended in a **draw**.

Let us now code this game

Board class

```

public class Board {
    private char board[][];
    private int boardSize = 3;
    private char p1Symbol, p2Symbol;
    private int count;
    public final static int PLAYER_1_WINS = 1;
    public final static int PLAYER_2_WINS = 2;
    public final static int DRAW = 3;
    public final static int INCOMPLETE = 4;
    public final static int INVALID = 5;

    public Board(char p1Symbol, char p2Symbol){
        board = new char[boardSize][boardSize];
        for(int i =0; i < boardSize; i++){
            for(int j =0; j < boardSize; j++){
                board[i][j] = ' ';
            }
        }
        this.p1Symbol = p1Symbol;
        this.p2Symbol = p2Symbol;
    }
    public int move(char symbol, int x, int y) {

        if(x < 0 || x >= boardSize || y < 0 || y >= boardSize ||
           board[x][y] != ' '){
            return INVALID;
        }

        board[x][y] = symbol;
        count++;
        // Check Row
        if(board[x][0] == board[x][1] && board[x][0] == board[x][2]) {

```

```
        return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
    }
    // Check Col
    if(board[0][y] == board[1][y] && board[0][y] == board[2][y]){
        return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
    }
    // First Diagonal
    if(board[0][0] != ' ' && board[0][0] == board[1][1] &&
       board[0][0] == board[2][2]){
        return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
    }
    // Second Diagonal
    if(board[0][2] != ' ' && board[0][2] == board[1][1] &&
       board[0][2] == board[2][0]){
        return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
    }
    if(count == boardSize * boardSize){
        return DRAW;
    }
    return INCOMPLETE;
}

public void print() {
    System.out.println("-----");
    for(int i =0; i < boardSize; i++){
        for(int j =0; j < boardSize; j++){
            System.out.print("| " + board[i][j] + " |");
        }
        System.out.println();
    }
    System.out.println();
    System.out.println("-----");
}
}
```

Player class

```
public class Player {  
  
    private String name;  
    private char symbol;  
  
    public Player(String name, char symbol){  
        setName(name);  
        setSymbol(symbol);  
    }  
  
    public void setName(String name) {  
  
        if(!name.isEmpty()) {  
            this.name = name;  
        }  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public void setSymbol(char symbol) {  
        if(symbol != '\0') {  
            this.symbol = symbol;  
        }  
    }  
  
    public char getSymbol() {  
        return this.symbol;  
    }  
}
```

TicTacToe class (Main class)

```

import java.util.Scanner;

public class TicTacToe {
    private Player player1, player2;
    private Board board;

    public static void main(String args[]){
        TicTacToe t = new TicTacToe();
        t.startGame();
    }

    public void startGame(){
        Scanner s = new Scanner(System.in);
        // Players input
        player1 = takePlayerInput(1);
        player2 = takePlayerInput(2);
        while(player1.getSymbol() == player2.getSymbol()){
            System.out.println("Symbol Already taken !! Pick another
                                symbol !!");
            char symbol = s.next().charAt(0);
            player2.setSymbol(symbol);
        }
        // Create Board
        board = new Board(player1.getSymbol(), player2.getSymbol());
        // Conduct the Game
        boolean player1Turn = true;
        int status = Board.INCOMPLETE;
        while(status == Board.INCOMPLETE || status == Board.INVALID){
            if(player1Turn){
                System.out.println("Player 1 - " +
                                   player1.getName() + "'s turn");
                System.out.println("Enter x: ");
                int x = s.nextInt();
                System.out.println("Enter y: ");
                int y = s.nextInt();
                status = board.move(player1.getSymbol(), x, y);
                if(status != Board.INVALID){
                    player1Turn = false;
                    board.print();
                }else{
                    System.out.println("Invalid Move! Try Again");
                }
            }
        }
    }

    private Player takePlayerInput(int playerNumber){
        System.out.println("Player " + playerNumber + ", Enter your name");
        Scanner s = new Scanner(System.in);
        String name = s.nextLine();
        return new Player(name);
    }
}

```

```

        }
    }else{
        System.out.println("Player 2 - " +
                           player2.getName() + "'s turn");
        System.out.println("Enter x: ");
        int x = s.nextInt();
        System.out.println("Enter y: ");
        int y = s.nextInt();
        status = board.move(player2.getSymbol(), x, y);
        if(status != Board.INVALID){
            player1Turn = true;
            board.print();
        }else{
            System.out.println("Invalid Move! Try Again");
        }
    }

    if(status == Board.PLAYER_1_WINS){
        System.out.println("Player 1 - " + player1.getName() +"
                           wins !!");
    }else if(status == Board.PLAYER_2_WINS){
        System.out.println("Player 2 - " + player2.getName() +"
                           wins !!");
    }else{
        System.out.println("Draw !!");
    }
}

private Player takePlayerInput(int num){
    Scanner s = new Scanner(System.in);
    System.out.println("Enter Player " + num + " name: ");
    String name = s.nextLine();
    System.out.println("Enter Player " + num + " symbol: ");
    char symbol = s.next().charAt(0);
    Player p = new Player(name, symbol);
    return p;
}
}

```

Othello

Refer to course videos for better understanding of the game.

Othello is a board game and you are expected to implement the move function for this game.

Approach: We have eight different directions to explore before we make a move (before we make changes to the board) to ensure which all boxes will toggle. We will move in every direction one by one and check for the player's value who just made his turn. We will then toggle the desired boxes, which the current player secured by playing in that position. To explore all eight directions we can create arrays for different directions and looping in the board we can increment or decrement in respective value to move in the particular direction, like we explore ways in backtracking lecture for rat in a maze game.

Now this code can be written on your own. Refer to the solution tab for the solution.

Backtracking

Introduction

- A **backtracking** algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.
- The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.
- The term **backtracking** suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- This approach is used to solve problems that have multiple solutions.
- Backtracking is thus a form of recursion.
- We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specified option does not give the required solution.
- We repeat these steps by going across each available option until we get the desired solution.

There are three types of problems in backtracking:

- **Decision Problem:** In this, we search for a feasible solution
- **Optimization Problem:** In this, we search for the best solution
- **Enumeration Problem:** In this, we find all feasible solutions

Difference between Recursion and Backtracking

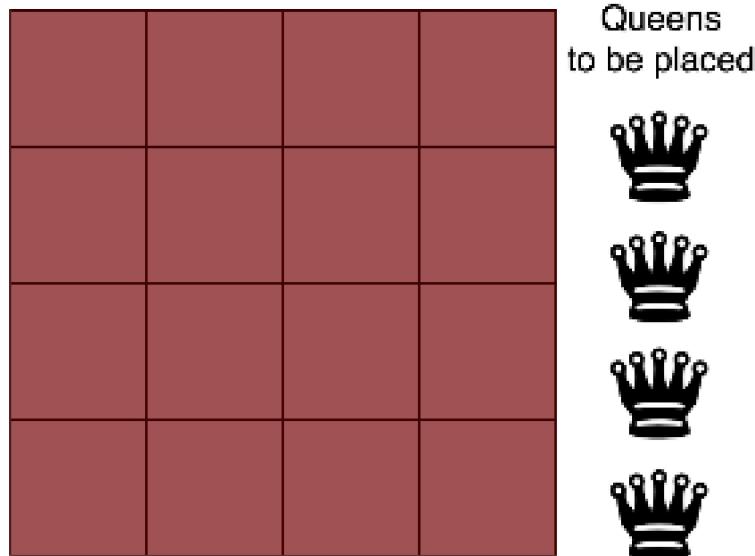
In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Problem Statement: N-Queen

One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally.

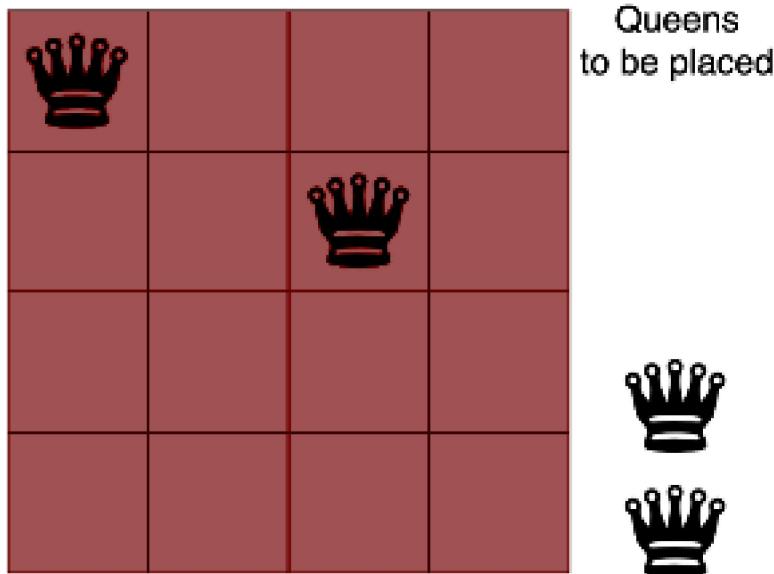
The solution to this problem is also attempted using Backtracking.

- We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places.
- We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left.
- If no safe place is left, then we change the position of the previously placed queen.

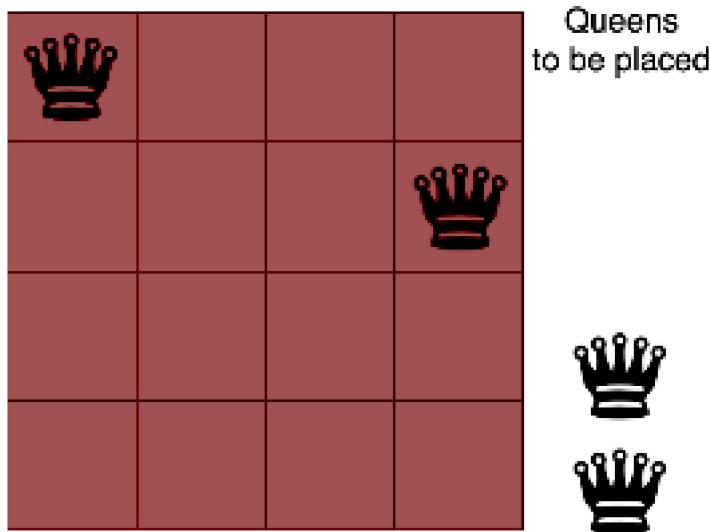


- The above picture shows an NxN chessboard and we have to place N queens on it. So, we will start by placing the first queen.

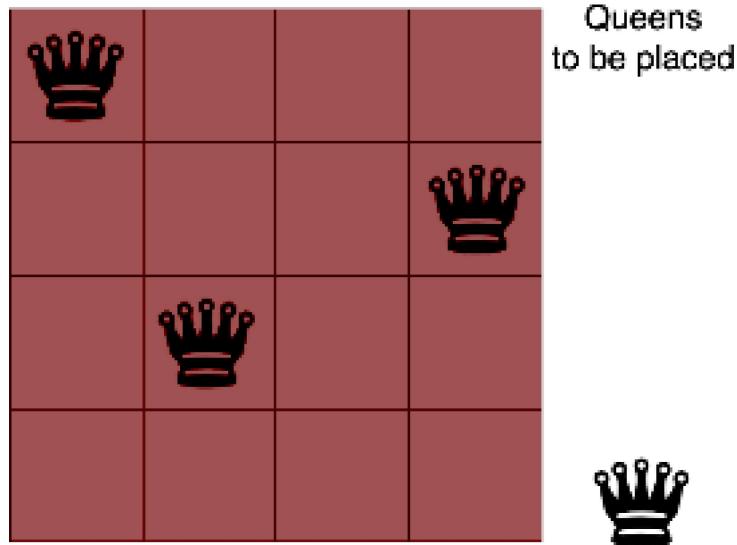
- Now, the second step is to place the second queen in a safe position and then the third queen.



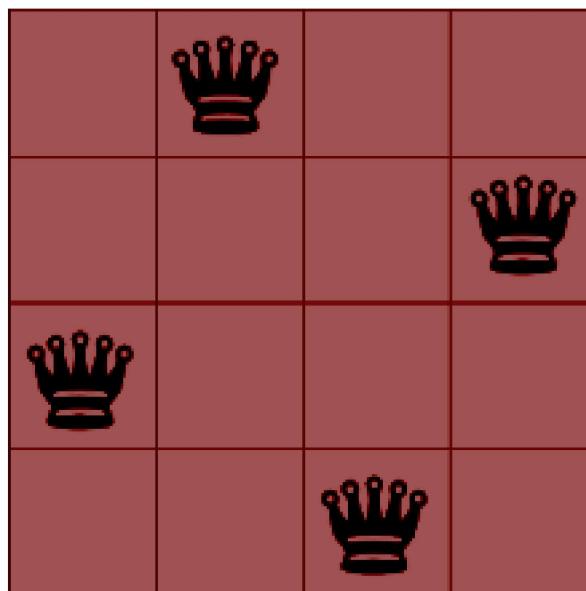
- Now, you can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking.
- Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.



- And now we will place the third queen again in a safe position until we find a solution.



- We will continue this process and finally, we will get the solution as shown below.



As now you have understood backtracking, let us now code the above problem of placing N queens on an NxN chessboard using the backtracking method.

```

public static boolean check_possible(int i, int j, int n){
    //checking if there is a queen in row or column
    for(int k=0; k<n; k++){
        if (board[i][k]==1 || board[k][j]==1)
            return true;
    }
    //checking diagonals
    for(int k=0; k<n; k++){
        for(int l=0; l<n; l++){
            if((k+l==i+j) || (k-l==i-j)){
                if(board[k][l]==1)
                    return true;
            }
        }
    }
    return false;
}

public static boolean N_queen(int n){
    //if n is 0, solution found
    if (n==0)
        return true;
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            /*checking if we can place a queen here or not
            queen will not be placed if the place is being attacked
            or already occupied*/
            if (!(check_possible(i,j, n))) && (board[i][j]!=1){
                board[i][j] = 1;
                //recursion
                //check if we can put a queen in this arrangement
                if (N_queen(n-1)==true)
                    return true;
                board[i][j] = 0;
            }
        }
    }
    return false;
}

```

```

        }
    return false;
}

```

Explanation of the code

- `check_possible(int i,int j)` → This is a function to check if the cell (i,j) is under attack by any other queen or not. We are just checking if there is any other queen in the row ‘i’ or column ‘j’. Then we are checking if there is any queen on the diagonal cells of the cell (i,j) or not. Any cell (k,l) will be diagonal to the cell (i,j) if k+l is equal to i+j or k-l is equal to i-j.
- `N_queen` → This is the function where we are implementing the backtracking algorithm.
- `if(n==0)` → If there is no queen left, it means all queens are placed and we have got a solution.
- `if((!check_possible(i,j)) && (board[i][j]!=1))` → We are just checking if the cell is available to place a queen or not. `check_possible` function will check if the cell is under attack by any other queen and `board[i][j]!=1` is making sure that the cell is vacant. If these conditions are met then we can put a queen in the cell – `board[i][j] = 1`.
- `if(N_queen(n-1)==1)` → Now, we are calling the function again to place the remaining queens and this is where we are doing backtracking. If this function (for placing the remaining queen) is not true, then we are just changing our current move – `board[i][j] = 0` and the loop will place the queen in some other position this time.

Another Example: Rat in A Maze

Go through the given blog to get a deeper understanding of the **Rat in A Maze** Problem:

<https://www.codingninjas.com/blog/2020/09/02/backtracking-rat-in-a-maze/>