Java Advantages over C++

1. Fully OOPS (Purely OOPS)
2. Array.length() → DS's in C++ also
3. Dynamic array (ArrayList<>) 1D
4. Strings, StringBuilder, Integers and all their properties
5. Self build constructors.
6. Type casting

DS's in C++ also

(7) "varargs" (Variable Arguments) three dots [func(Strings...)] ↳ see Page 46

int N = Integer.parseInt(inp[0]);

↳ ParseInt in Java generally used to convert String to Integer

2.9 String S = "50";
int n = Integer.parseInt(S);
sysO(n); // 50
sysO(s); // 50

import java.io.*; ↳ This statement will import everything to the package
import java.util.*;

same as ... but its little fast

type casting implicit: ⟺ [float a;
double b = (float) a;

CRUX

Data Structure & Algorithm in Java

Everything Headon ... do leftovers and data structure and recursion based only.

Table of Contents

VKP

done & done

Compiled By :

Apoorvaa Gupta • Bhavya Aggarwal • Rishab Kapoor • Sumeet Malik

Vector<T> vector = new Vector<T>(); → Vectors plays same role as Array in C++

1 Introduction to Java Programming

Introduction :

Java is a high level programming language which was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' java platform. The language derives much of its syntax from C and object oriented features from C++.

To write and run programs in any language we need a text editor (to write the code), a compiler (to convert the code into machine readable form) and a debugger (for finding and removing the bugs). For writing code any text editor like notepad can be used. For compiling, we can use the compiler from the terminal. Debugging needs be done by looking through the code.

Now instead of doing all these separately we can use an environment where we can do all the things at one place. Such an environment is called an IDE which stands for **Integrated Development Environment.**

Here, we can write, compile, run and debug our code. We will use eclipse IDE in this course.

When we compile our code in languages like C/C++, we get an executable file but that executable file can be run only on that platform (Operating System) on which it is compiled. For instance a C++ program compiled on windows can only be run on windows and not on Linux. This implies that a program compiled on a particular platform cannot be run on any other platform. This creates the problem of platform dependency. Java solves this problem and thus became very popular.

Platform Independency in Java:

Instead of converting the code directly to the executable file, java converts it into a middle form which can then be converted to the executable file depending on the platform on which it is to be run.

The code we write in the editor is compiled by javac (java compiler) which converts it into a middle form known as the bytecode (intermediate). This bytecode is then converted to the executable form by **JVM (Java Virtual Machine)** according to the platform on which it is being run. This achieves platform independency.

3 Functions

We know that all the code that we want to execute has to be written within the 'main' body. Now assume that we have to calculate the factorial of a number at two different points in our program. For that we would need to write the code for factorial twice in the same program. A better option would have been to write the code for factorial once and use that whenever required! This will achieve the **reusability of code**. Also, if we need to change our code a little bit, we would need to change it in different places. But if all the code is written in a single place, changing the code in only that part would reflect the changes in all the other places where that same code is used. This makes our program **maintainable**. All this can be done by using functions.

A function is a set of code which is referred to by a name and can be called (invoked) at any point in the program by using the function's name. It is like a subprogram where we can pass parameters to it and it can even return a value. All functions are written outside of the main body and can be called from the main body or even other functions. Thus our main function doesn't get cluttered with all the code, improving readability. In fact 'main' is also a function, a special function which is called by the JVM at the runtime.

Basic Syntax for a Function :

```
// public static returnType nameOfFunction(Parameters)
public static int func(){
      //do your work here
      //return is compulsory
      return 0;
}

public static void func(){
      //do your work here
      //return is not compulsory
      return;
}
```

For now, begin all your functions with the keywords 'public static' till we understand their significance in detail. After that the **return type** of the function is specified.

By return type we mean that we specify the data type of the return value. It may also happen sometimes that the function does not return anything. In that case the return type of the function is **'void'** and it is not compulsory to write the 'return' keyword in this case. After the return type we write the name of the function followed by two parentheses where we give the **parameters** to the function. We enclose the entire body of the function within two curly braces. Inside that we can write the function body. Generally the last statement in the function is the return statement.

Now just making the function would not execute it when we run the program. To execute any function we need to invoke it. This is called **calling the function**. Functions can be called from the main function and also from other functions. If there are any parameters to be given to the function, they are also passed within the parenthesis. If the function returns any value it can be stored in a variable. Now let us write two functions and see how they can be called!

```
public static void main(String[] args) {
      System.out.println("this is main");
      // Calling functions from the main
      func1();
      func2();
}

public static void func1() {
      System.out.println("this is func1");
}

public static void func2() {
      System.out.println("this is func2");
      // Calling another function inside a function
      func1();
}
```

OUTPUT:

```
this is main
this is func1
this is func2
this is func1
```

We can also pass parameters to the functions. When we are defining the function, we specify the data type of the parameter followed by the name of the parameter within the parenthesis. This parameter would be referred by that name within the function body. We can pass as many parameters as we want. When the function is called, we need to pass the parameters of the same data type as specified in its definition. Note that while calling the function we only pass in the names of the variables or values and don't specify its data type!

```
public static void main(String[] args) {
      int one = 3, two = 5, sum;
      sum = add(one, two);
      System.out.println(sum + " is the sum");
      // OUTPUT: 8 is the sum
}
```

These conditions can also be nested which means that we can have the 'if-else' statements inside an 'if' block.

```
public static void main(String[] args) {
      // To calculate largest of three numbers
      // Using nested if-else statement

      int a = 5, b = 10, c = 15;

      if (a >= c) {
            if (a >= b) {
                  System.out.println("a is largest");
            } else {
                  System.out.println("b is largest");
            }
      } else {
            if (c >= b) {
                  System.out.println("c is largest");
            } else {
                  System.out.println("b is largest");
            }
      }
}
```

Looping : Most often while writing code, we need to perform a task over and over again till our work is done! This can be achieved by the concept of loops. In loops, we check a condition of 'Is the work done?' every time before doing the work. Thus, eventually when the work gets finished we break out of that loop. Note that it is very important to have a breaking condition in the loop else the loop would continue forever and it will become an infinite loop! There are mainly three types of loops - the **'for' loop**, the **'while' loop** and the **'do-while' loop**. The syntax for each of them is given below.

```
// To calculate sum of first 5 natural numbers.
int sum = 0; // Initialization before loop

// Using for loop
// for(declaration,condition,increment)
for (int i = 1; i <= 5; i++) {
      sum = sum + i;
}

// Using while loop
int i = 1; // Declaration
while (i <= 5) {
      sum = sum + i;
      i = i + 1; // increment
}
```



```
// Using do-while loop
int i = 1; // Declaration
do {
      sum = sum + i;
      i = i + 1; // increment
} while (i<=5);
```

conditions ← [if else → condition
if else → nested conditions

loops ← [For loop
while loop
do while loop

For showing output we use "System.out.print("")". Whatever is inserted in the quotes is printed on console. This statement prints out the text in continuation on the console. If we want to print each text in a separate line then we must use "System.out.println("")".

Taking Input:

```
import java.util.Scanner;
public static void main(String[] args) {

      // Creating a Scanner scn
      Scanner scn = new Scanner(System.in);

      // Taking input as an integer and
      // Storing it in a variable 'n'

      int n = scn.nextInt();

      // Taking input as a String and
      // Storing it in a variable 's'

      String s = scn.next();
}
```

For taking input from the user we first need to import the 'java.util.Scanner' which contains the code for the Scanner. Then we make a new Scanner and use it to take input from the console screen.

But after taking the input we need to store the input in some place so as to retrieve it later when required. The data is stored in some memory location which is identified by a name. This name is called a variable. Also each variable can't store anything in it but a particular type of data which is limited by its data type!

Data Types in Java:

There are two types of data types:

- **Primitive data types** : These are basic data types which are used to represent single values. Primitive data types are defined by the java programming language.
- **Non Primitive data types** : These are not defined by the java programming language, but are created by the user like arrays and strings. We will learn about them later!

Primitive Data Types : There are 8 types of primitive data types.

1. **Byte:** The byte data type is an 8-bit (1 byte) integer having values from -2^7 to + ($2^7 - 1$).
2. **Short:** The short data type is a 16-bit (2 bytes) integer having values from -2^{15} to + ($2^{15} - 1$).
3. **Int:** The int data type is a 32-bit (4 bytes) integer having values from -2^{31} to + ($2^{31} - 1$).
4. **Long:** The long data type is a 64-bit (8 bytes) integer having values from -2^{63} to + ($2^{63} - 1$).
5. **Char:** The char data type is a single 16-bit Unicode character having values from 0 to 2^{16}-1.
6. **Float:** The float data type is a single-precision 32-bit (4 bytes) floating point.

7. **Double:** The double data type is a double-precision 64-bit (4 bytes) floating point.
8. **Boolean:** The boolean data type can only contain two values true and false. Its size is not precisely defined.

```
public static void main(String[] args) {
      byte b = 10;
      short s = -10;
      int n = 45;
      long l = 10000000000L; // L is used to show the long data type
      float f = 1.5F; // F is used to show the float data type
      double d = 1.000000000001;
      char c = 'a'; // Includes alphabets, digits, special characters
      boolean bool = false; // boolean can have values true or false
}
```

We will learn about the non primitive data types later!

Decision Making : Decision making is an integral part of most of the algorithms. This is achieved by the use of the 'if-else' statements!

```
public static void main(String[] args) {
      // To calculate largest of three numbers
      // Using if-elseif-else statement

      int a = 5, b = 10, c = 15;

      if (a > b && a > c) {
            System.out.println("a is largest");
      } else if (b > a && b > c) {
            System.out.println("b is largest");
      } else {
            System.out.println("c is largest");
      }
}
```

Here the condition in the parenthesis of 'if' is evaluated. If the condition comes out to be true then the code in the **'if'** block is executed but if it comes out as false then the code in **'else'** block is executed. If more than two conditions are there which need to be checked then the **'else if'** keyword is used and hence multiple conditions can be evaluated.

JVM is the Java Runtime which converts the bytecode into the form understandable by hardware. A lot of functionalities in java have been already written in the core libraries of the java language to help the successful running of the program we need to have the JVM as well as the core libraries on our RAM. Thus to run any java program on our system we need to have the JRE installed. JRE together with the java's compiler 'javac' make the JDK which is called the **Java Development Kit**. This is the superset of JRE and is needed by the developers of java. Thus we would also need to have JDKs installed on our machines to begin developing!

Now every time when we run the java source code, two steps are performed. First the source code is converted to the byte code and then to the binary code. Due to this, Java is SLOWER than languages like C/C++.

Another major issue is that to run the java programs the user also needs to have the JRE installed which is not very desirable.

Linker and Loader :

The core libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs the compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Loader is a program that loads machine codes of a program into the system memory. It places programs into the memory and prepares them for execution.

After having learnt about all these it's time to begin coding!

2 Getting Started with Programming

Introduction:

Now let us start our java programming by writing a simple hello world program and understand its syntax.

```
package Demo;
import java.util.Scanner;
public class HelloWorld {
	public static void main(String[] args) {
		// This is the body of main
		// Here we write all our code
		System.out.println("Hello World");
	}
}
```

The first line specifies the package name 'demo' and below that we have a class 'HelloWorld'. For time being you can assume that package is like a folder and classes are like the files stored in those packages (folders). The **'import'** keyword is used to include the core libraries which already have lot of java functionality written and which can be used in our program. In the 'HelloWorld' class we have a main method where we write down all our code. Methods in java is similar to functions in C or C++. For now use all the keywords and the basic syntax as it is till we learn about them in more detail!

Note that every statement in Java is terminated by a semicolon ';'.

Now to write any code, majorly three things are needed. These basic things are: method to interact with the user that is to take input and display the output, decision making based on conditions and repeating a particular task again and again. Let us learn how to execute each of these!

Printing on Screen:

```
public static void main(String[] args) {
	// This will print Hello and World together
	// without any spaces or newline in between.

	// OUTPUT: HelloWorld
	System.out.print("Hello");
	System.out.print("World");

	//This will print Hello and World in two different lines.
	//OUTPUT: Hello
	//        World
	System.out.println("Hello");
	System.out.println("World");

	//Prints an empty new line.
	System.out.println();
}
```

```
public static int add(int one, int two) {
        int sum;
        sum = one + two;
        return sum;
}
```

In the previous code, we have written a function for adding two numbers. An important point to note here is that the variables present in the main function are different from those that are present in the add function, though they have the same names. To understand this better let us learn about the concept of variables and their scope.

Variables and their Scope :

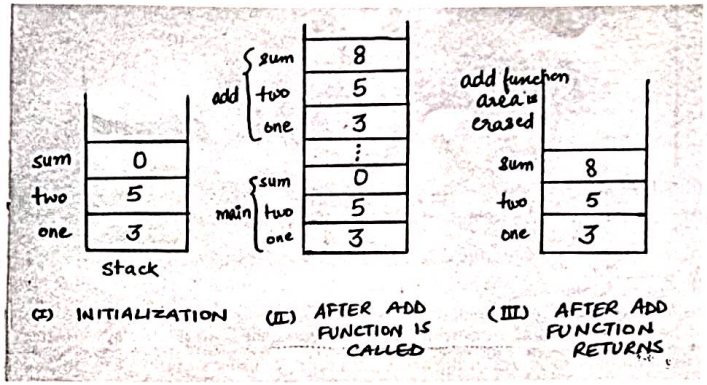
During runtime a memory pool is created by the JVM to store all the variables, codes for functions etc. In this memory pool there are majorly two types of memory – the **stack memory** and the **heap memory**. Note that the values of all the primitive data types are stored on the stack whereas the non primitive data types are made on the heap and their reference is stored on the stack! All the functions that are called occupy their area on the stack where all their variables are stored. The functions keep stacking one above the other in the order of their calls. When the function returns, it is erased from the stack and all its local variables are destroyed.

The variables that are present in one function are local to only that function. It cannot be accessed outside that function. For instance a variable made in the main function is not available to the add function unless we pass it as a parameter to it. Even then only the value would be passed (if the variable is of primitive data type) and that would be stored in another variable which is local to the add function. Thus in the last code, the 'one' and 'two' variables in the main function are different from the 'one' and 'two' variables of the add function. So any changes to the variables in the add function would not alter the values of the variables in the main function.

But this is always not the case. When the data type is non primitive, the reference to the actual variable is stored on heap. Now when this variable is passed onto a function, the reference stored on the stack would be passed. Thus the variable in the function would have the same reference as that in the main function and both would point to the same variable present on heap. Thus if the function alters the value of the variable on the heap, it will be reflected in the variable of the main function also.

Now let us talk about another type of variable which no matter what is always made on the heap. It is the global variable. **Global variable** is the variable which is available to all the functions of the program. It is not local to any function and thus its value cannot be stored on the stack. If its value had to be stored on the stack then it would have been local to that function in whose area it is stored. As it local to none, thus it is stored on the heap memory and can be accessed by all the functions.

To understand this more clearly, let us draw the memory maps of different situations.



In java during the function calls only the values stored on the stack are passed. Unlike C/C++, there is nothing like 'pass by reference' in java but only 'pass by value'.

4 Arrays

Let us consider a case in which we have to store the names of 50 students of a class.

One way would be to create 50 variables and store each name in one variable. But it would be very difficult to remember the names of 50 variables.

This problem could be solved by using **non-primitive data type** called 'arrays'. Arrays are the container objects that hold a fixed number of values of the same data type. By using arrays we can store names of all the students under a single variable name. An array is a **continuous block of memory** made in the heap and its reference is stored in the stack. Now let us learn how to make an array and store different values in it.

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

    Scanner scn = new Scanner(System.in);

    // Declaring an array
    // Data type[] name;
    int[] arr;

    // Initializing
    arr = new int[3];

    // Setting the values; array is [0,1,2]
    for (int i = 0; i < 3; i++)
        arr[i] = i;

    // Taking input and setting the values
    for (int i = 0; i < 3; i++) {
        arr[i] = scn.nextInt();
    }
}
```

Firstly we specify the data type of values to be stored in that array followed by the name of the array. This only creates a variable named 'arr' on the stack and it contains the value null. Right now it does not point to any memory on the heap.

The new keyword creates a new array of the size specified in the square brackets on heap and the variable 'arr' on the stack contains the reference of that new array. Note that after the creation of the array, its **length is fixed** and cannot be changed as it is static memory. Array is created on heap such that all the elements occupy contiguous memory.

The indices start from 0 and go up to n-1 for an n-sized array.

Another method to create an array is to enter the elements in the curly braces.

```
public static void main(String[] args) {
    // Setting the values directly makes an array
    // of length with elements as specified
    int[] arr = { 1, 2, 3, 4, 5 };
}
```

To access the elements of an array we use square brackets notation. Within the square brackets we specify the index number of the value we want to get or set.

For instance if we need to store three numbers in an array, the first number would be stored at the 0th index, the second number would be stored at the 1st index and the third number would be stored at the 2nd index. Thus using the indices we can get and change the value at any position in the array.

If we want to know the size of the array or the total number of elements in the array we can use the **'array.length' property** and use it on the name of the array.

```
public static void main(String[] args) {
    int[] arr = new int[3];

    // Setting the values at each index
    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;

    // Updating the value at 1st index
    arr[1] = 4;

    // To find the length of the array
    int l = arr.length;
    // l = 3;
}
```

Displaying an Array : To display an array we can use a 'for' loop. We iterate over every index and display each value.

```
public static void displayArray() {
    int[] arr = { 1, 2, 3, 4, 5, 6 };

    // Using for loop on the elements of array
    for (int i = 0; i < arr.length; i++) {
        System.out.print(arr[i] + " ");
    }

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

We can also have a ***staggered array*** *where each row is of different size. For this we have to new each row separately.*

```
// Making a staggered array with 5 rows
int[][] arr = new int[5][];

// Each row can have different columns
for (int i = 0; i < arr.length; i++) {
    arr[i] = new int[i + 1];
}
```

Searching Algorithms: Suppose we have an array and we want to search for a particular element in it. If the element exists we return its index else we return -1 to signify its absence. This can be done by two algorithms:

- Linear search
- Binary search

Linear Search: In linear search, we start from the 0th index and compare the data stored in the array with the data to be searched. If they match we return the index and if not we move to the next index. If we reach the end of the array and still are not able to find the data we return -1. This algorithm is illustrated below.

An array with 10 elements, search for "9" :

56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9

Binary Search: The binary search algorithm is based on the '**DIVIDE and CONQUER**' paradigm. In this we try and divide our problem into smaller sub problems, solve these smaller problems and finally merge them to get the result.

Let us look at the binary search algorithm and try to formulate the problem into the above steps!

First of all there is an assumption in this algorithm that the array in which we have to search an element has to be sorted. Then we find out the mid element of that sorted array. If that matches with the data to be found we just return the index and return. If the element is greater than the data that means the data must be present in the left half of the array as it is sorted whereas if the element is smaller than the data it means it is present in the right part of the array. Then we search for the data only in one of the parts and discard the rest of the elements. Thus in each turn we are dividing the problem into smaller problem. In the end if all the elements are finished and still we are not able to find the data, it means that the data is not present and thus we return -1. There is no merge step in this algorithm.

If searching for 23 in the 10-element array :

	2	5	8	12	16	23	38	56	72	91
	L									H
23 > 16, Take 2nd half	2	5	8	12	16	23	38	56	72	91
						L				H
23 > 56, Take 1st half	2	5	8	12	16	23	38	56	72	91
						L	H			
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

Sorting Algorithms: Sorting means to arrange the elements of an array into a particular order (generally ascending). There are various algorithms for carrying out this but we will focus on three for now.

- Bubble sort
- Selection sort
- Insertion sort

Bubble Sort:

In this algorithm we start with the 0th index and traverse through the array. We compare the adjacent elements and swap them if they are in incorrect order. Thus at the end of the first traversal the biggest element in the array reaches to the end and attains its correct position. We repeat this same process but in the second traversal we won't go till the end as the last element is already in its correct position. We will stop one index before the last element and in the end the second highest element would attain its correct position. We repeat this process n-1 times placing n-1 elements at their correct positions. The last element would automatically be placed properly thus sorting the whole array. The illustration for this is given below.

ArrayLists: Now suppose that 2 more students join our class and we need to store their data as well. As the size of our array is fixed, we would need to make another array of greater size and copy all the data from the previous array. But this would be a tedious task. Instead we can use 'ArrayLists'. ArrayList is a dynamic array that is, its size is not fixed. For using ArrayList we have to import the 'java.util.ArrayList' class. For getting, setting, adding and removing elements in an arraylist we have to use methods provided inside 'ArrayList' Class. Let us create our own arraylist and learn more about these functions.

While creating an arraylist, we need to specify the data type of the data that has to be stored in the arraylist within the angular brackets. An important point to note here is that arraylist can only be made up of the non primitive data types and not of primitive data types like 'int', 'boolean', 'float', etc. Thus for creating an arraylist of integers we need to use the 'Integer' keyword within the angular brackets. The syntax for the add, set, get, remove and size method has been discussed below.

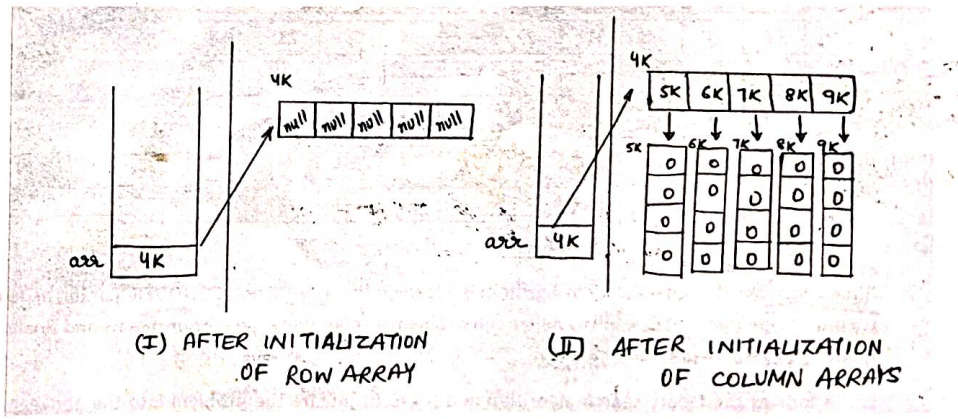
```
import java.util.ArrayList;

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

            // ArrayList<Datatype> name = new ArrayList<>();
            ArrayList<Integer> al = new ArrayList<>();

            // Adds 5 to the arraylist
            al.add(5); // al: [5]

            // Adds 4 at the 0th index
            al.add(0, 4); // al: [4,5]

            // Get the value at 1st index
            al.get(1); // returns 5

            // Set the value at any index
            al.set(1, 100); // al: [4,100]

            // Remove val from any index
            al.remove(0); // al: [100]

            // To get the size of arraylist
            al.size(); // returns 1
```

Multi-Dimensional Array: Till now we have read about arrays and arraylists which are linear structures. We can also have 2-d structures that have rows and columns. We can also have 3-d structures that also have heights. These are all called multi-dimensional arrays. We will only focus on the **2-d array** which is also called **matrix**. In general we can say that a 2-d array is an array of arrays.

Declaration of 2-d arrays is similar to that of normal array with the difference of an extra square bracket. The first square bracket specifies the number of rows and the second specifies the number of columns. So every time we want to access any element we have to specify both its row and column index. If there are m rows and n columns then the indices would go from 0 to m-1 for rows and 0 to n-1 for columns.

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

    // Makes a variable with value null
    int[][] arr;

    // Creates a 1D row array
    arr = new int[5][];

    // Creates all the columns
    for (int i = 0; i < arr.length; i++) {
          arr[i] = new int[4];
    }
    // Finally creates a 5*4 2D array

    // Direct method
    int[][] arr1 = new int[5][4];

    // Setting any element
    arr[0][3] = 5;

    // Getting any element
    int temp = arr[0][3];

    // temp=5
```

Let us make a memory map and understand how a 2-d array is stored in memory.



There is another way to write the 'for' loop and iterate over the elements of the array. It is known as the **'enhanced for loop'**. Its syntax is given below. In this rather than accessing the values in the array by indices, they are directly iterated by an iterator which is 'val' in this case.

```
public static void displayArray2() {
    int[] arr = { 1, 2, 3, 4, 5, 6 };

    // Making an iterator val of type int
    // and looping it over the array
    for (int val : arr) {
        System.out.print(val + " ");
    }

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

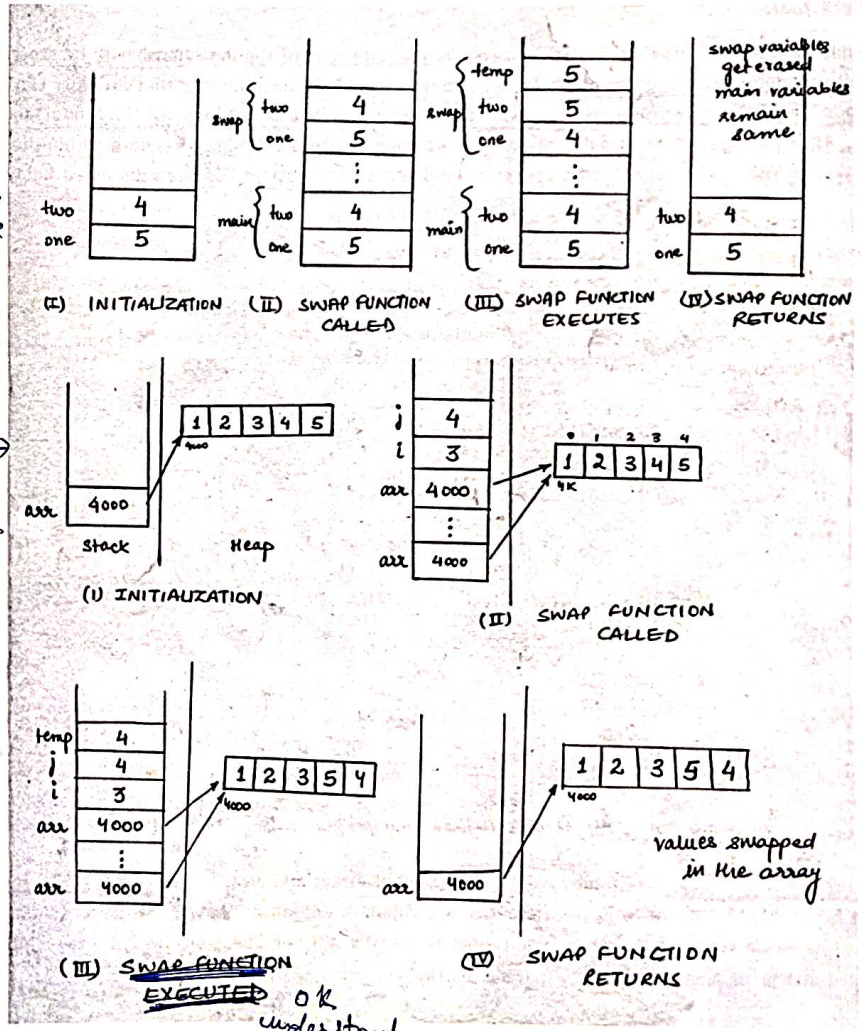
Function and Arrays: In the function definition we specify the type of the array along with its name whereas in the function call we just pass the name of the array. As we know that array is a non primitive data type so, only its reference is stored in the stack and when passed in a function call, the **reference is passed**. Thus if the function makes any changes in the array, they would be reflected in the original array also as both the functions (the calling function and the called function) share the reference to the same array.

Let us verify this thing by writing a simple program of swapping two values.

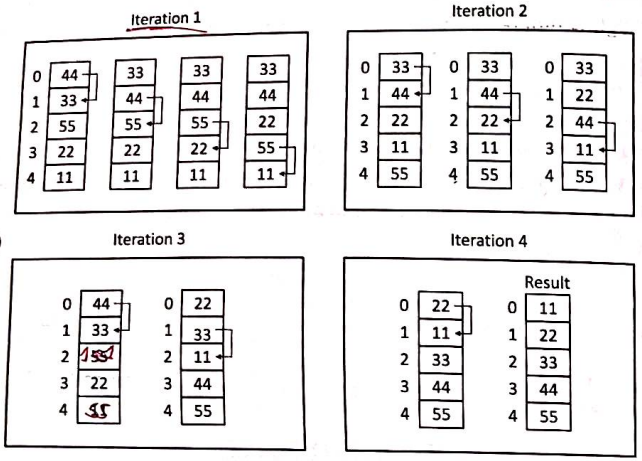
```
public static void main(String[] args) {
    int one = 5, two = 4;
    int[] arr = { 1, 2, 3, 4, 5 };
    System.out.println(one + " " + two); // 5 4
    swapIntegers(one, two);
    System.out.println(one + " " + two); // 5 4
    // Numbers not reversed

    System.out.println(arr[3] + " " + arr[4]);// 4 5
    swapArrayNums(arr, 3, 4);
    System.out.println(arr[3] + " " + arr[4]);// 5 4
    // Numbers reversed
}

public static void swapIntegers(int one, int two) {
    int temp = one;
    one = two;
    two = temp;
}

public static void swapArrayNums(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```


In the first case, where the variables were primitive, only the copies of the values were passed. Thus after the returning of the swap function, the changed variables are destroyed and hence no changes are retained. But in the second case, changes are made in the same array as both the functions' array variable pointed to the same array. Thus even after the function returns, the changes are retained. Remember that there is no such thing as pointers in java!

Memory Maps:



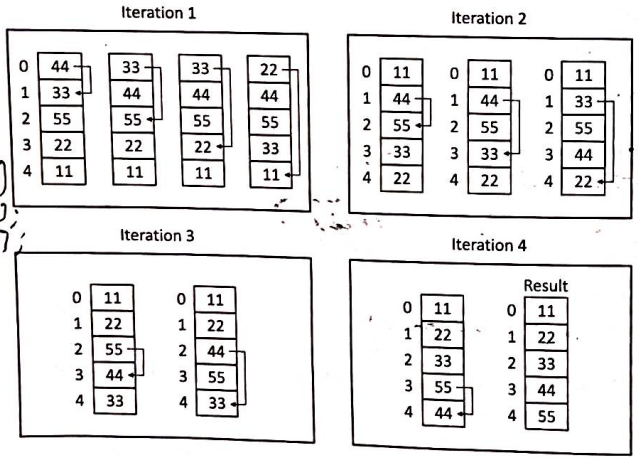
Bubble Sort



Selection Sort:

This algorithm is similar to bubble sort in a way that we traverse the array n-1 times. In the first traversal we compare the first element with all the other subsequent elements and swap if they are in incorrect order. Thus after the first traversal the smallest element comes to the 0th index. Similarly in the n-1 total traversals the n-1 smallest elements are placed at their correct positions. This sorts the array.

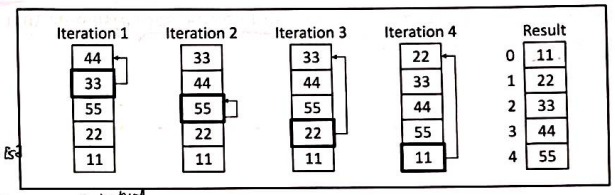
Selection Sort



Insertion Sort:

This is a slightly different algorithm. In this we try to insert the elements of the array into an already sorted array. When we have inserted all the elements the array is said to be sorted. Let us understand this in a little detail. Consider that there are 5 elements in an array which are to be sorted. First we consider only the 0th index as the sorted part and the rest is unsorted. Then we try to insert the element at the 1st index into the sorted array and after this the sorted array contains 0th and the 1st index. Thus each time we insert an element the sorted array becomes lager and the unsorted becomes smaller. We repeat this process till all the elements in the array become the part of the sorted array. At this point the array becomes sorted!

Insertion Sort

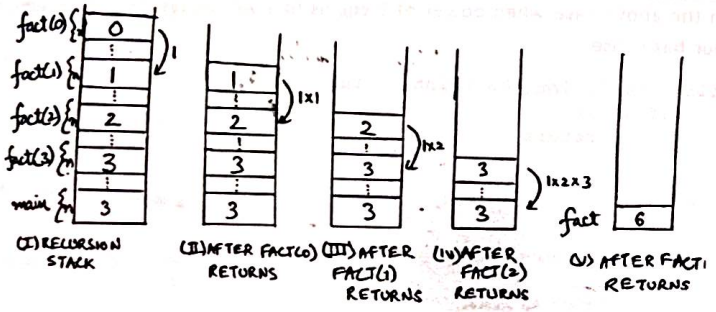


An important point to note here is that in the base case we need to have a return statement compulsorily even if the return type of the function is void. This is because even in the base case if we don't have the return statement, the code after the base case would begin to execute and the recursion would execute infinitely. Thus to terminate recursion, we need to have a return statement in the base case.

For factorial of 5:

```
factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120
```

Let us understand the recursive function calls through memory maps:



Now we know that every recursive function includes a base case and a call to itself. So every time the function calls itself we have two opportunities to do our work. First is before calling itself and second is after the call. First opportunity helps to do work on the input whereas second opportunity lets us do work on the result for the smaller problem. To understand this better let us write code for printing a series of increasing and decreasing number


```
public static void printDI(int n) {
    // Base case
    if (n == 0) {
        return;
    }

    // Work before the recursive call
    System.out.println(n);

    // Recursive call
    printDI(n - 1);
    // Work after the recursive call
    System.out.println(n);
}
```

OUTPUT for n=3

```
3
2
1
1
2
3
```

Practise for recursion as much as you can and also see from lecture videos and from youtubes.

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

    StringBuilder sb = new StringBuilder("hello ");
    sb.append("world"); // sb = "hello world"
    int len = sb.length(); // len = 11
    sb.setCharAt(5, '-'); // sb = "hello-world"
}
```

Assignment:

Q.1 Print all the subsequence of the given string.
[e.g. string: "abc", output: {a,ab,abc,b,bc,c}]

Q.2 Print all the permutations of the given string.
[e.g. string: "abc", output: {abc,acb,bac,bca,cab,cba}]

6 Recursion

Recursion is a method where the solution to a problem depends on the solution to smaller instances of that problem. In cases where we need to solve a bigger problem, we try to reduce that bigger problem in a number of smaller problems and then solve them. For instance we need to calculate 2^8. We can do this by first calculating 2^4 and then multiplying 2^4 with 2^4. Now to calculate 2^4, we can calculate 2^2 which in turn depend on just 2^1. So for calculating 2^8, we just need to calculate 2^1 and then keep on multiplying the obtained numbers until we reach the result.

So by dividing the problem in smaller problems we can solve the original program easily. But we can't go on making our problem smaller infinitely. We need to stop somewhere and terminate. This point is called a **base case**. The base case tells us that the problem doesn't have to be divided further. For instance, in the above case when power of 2 equals to 1 we can return 2. So power being equal to 1 becomes our base case.

```
public static int power(int x, int n) {
    if (n == 1) {
        return x;
    }

    int smallPow = power(x, n / 2);
    if (n % 2 == 0) {
        return smallPow * smallPow;
    } else {
        return smallPow * smallPow * x;
    }
}
```

In the next program we are calculating the factorial of a number.

For calculating the factorial of n, all we need is the factorial for n-1 and then multiply it with the number itself. As we can see that factorial of a bigger number depends on the factorial of a smaller number, thus this problem can be solved by recursion.

So for calculating factorial of a bigger number, we first calculate factorial of n-1 and then multiply it by n. If the number is 1, we don't need to calculate anything and just return 1. Thus this is our base case.

Notice that in the above code the factorial function calls itself on a smaller input and then use that result to calculate the final answer. Thus in recursion, a function calls itself.

5 Strings

If we have to save a student's name, one way is to create an array of type char and store each letter at one index. But it would be very difficult to use that as every time we will need to process the whole array. Instead we can use a **non primitive datatype** called 'String'. String is basically an object that represents a sequence of characters. One important point about strings is that they are **immutable**. This means that once a string is created its data or state can't be changed! Also note that strings can be concatenated by using the '+' operator between them.

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

    String s1 = "hello";
    String s2 = s1;
    String s3 = "hello";
    // All three have same reference

    // Using new keyword, makes a new object
    String s4 = new String("hello");

    // String concatenation
    String s5 = s1 + " world";

    // OUTPUT: hello world
    System.out.println(s5);

}
```

In the first case a string "hello" is created on the heap and its address is stored in the variable s1. Now the second variable s2 is assigned the same reference as s1 and both point to the same string "hello". Now the third case is a little interesting. Even though we explicitly assign the value "hello" to s3, it still has the same reference as that of s1 and s2. This is due to the interning action of JVM. JVM has an intern pool in its heap memory. It checks if it already has a string of the same value as that specified then instead of making a new string object it makes the variable point to the old object only. But if we explicitly tell java to make a new string then it doesn't perform the interning action and makes a new string object on the heap.

Different Functions: Now let us learn about different function that can be used to process strings.

→ **'charAt(index)'** function returns the character that is present at the specified index. If the index is not valid it will give a run time error.

'substring(si,ei)' function returns a substring of the original string from si (start index) to ei (end index), including si and excluding ei. If we just give one parameter that is si, then this function returns the string from si till the end of the string. Again if any of the parameters are invalid, it will give a run time error.

'concat(str)' function joins the string passed as a parameter to string on which the function is invoked.

'indexOf(ch)' function returns the first index at which the character is found. It returns -1 if the character is not found.

'startsWith(str)' function checks if the string on which the function is invoked starts with the string passed as a parameter or not. It returns a Boolean value - true or false

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

    String s = "hello";

    char ch = s.charAt(1); // ch = 'e'
    String ss = s.substring(1, 4); // ss = "ell"
    String ss1 = s.substring(2); // ss1 = "llo"
    String sc = s.concat("world"); // sc = "helloworld"
    int idx = s.indexOf('l'); // idx = 2
    boolean bool = s.startsWith("he"); // bool = true

    String s1 = new String("hello");

    boolean b1 = s == s1; // b1 = false
    boolean b2 = s.equals(s1); // b2 = true

}
```

For comparison of two values we have always used '==' operator. But in the case of string it doesn't always gives the correct answers. This is because this operator just compares the value present on the stack. As string is a non primitive, the reference to the string object is stored on stack. So if the reference is same it returns true otherwise false. But there might be a case when the reference in the stack is different but the values present on the actual object is same, but even then it would return false. To overcome this problem, we use the **'equals' method** on the string. This method compares character by character and even if the reference is different it returns true if the values are the same. Therefore we should always use s1.equals(s2) for checking if string s1 is equal to s2 or not.

String Builder: We already know that strings are immutable. So if we want to change any string or add a character to a string, first we will need to copy the contents of the original string into a new string object and then add the necessary changes. So this is a very costly operation in terms of time. To save time, we can instead use String Builder which is not immutable and hence is **very efficient**. It is similar to string but just with the difference that its value is not fixed and can be altered, thus providing better processing. Let us look at some of the string builder functions.

7 Time & Space Complexity

In this world, we always want our program to be fast and use less space. In short we want our program to be space and time efficient.

To understand this better, let us learn a new type of sorting algorithm called the merge sort and learn how to analyse time complexity.

Merge sort is based on the **divide and conquer** paradigm. The technique is based on divide, conquer and merge operations. Marge sort is a recursive algorithm. In this, we divide our array into two halves. This is the divide step. Then we recursively sort these two sub arrays which is the conquer step. Then finally we merge the two sorted sub arrays to get the final sorted array. Sorting two smaller arrays is easier than sorting a very big array. In this the base case is hit when we have divided our array so much that it contains just one element. In that case we don't need to do any work as one element is already sorted. So we do nothing and just return. Then while returning we keep on merging the smaller arrays until we get the whole array back.

Now to test which algorithm- the merge sort or the bubble sort is better, we have calculated the times that each one takes to see which one is more time efficient.

INPUT SIZE	TIME TAKEN BY BUBBLE SORT	TIME TAKEN BY MERGE SORT
1000	13 ms	3 ms
10000	488 ms	5 ms
100000	37179 ms	39 ms
1000000	>1 hr	300 ms

From the results it can be seen clearly that merge sort is the winner. This can be seen from the experimental results but it's not possible all the time to verify experimentally. So let us learn a technique which can help us to calculate the time complexity of any given algorithm.

We represent the amount of time taken by the algorithm to run as a function of its input size. For instance, the time taken by an algorithm is: $T(n) = 6n^3 + 2n^2 + 3$. While computing time complexity, we generally ignore the lower order term and just keep the highest degree term. We also ignore all the coefficients of the terms.

Thus the time would be represented as $T(n) = O(n^3)$ where Big-O represents the upper bound for the equation. We generally use the Big-O notation to represent time complexity. We have neglected $2n^2$ and 3 in the above example.

We consider addition, subtraction, multiplication, division etc to be of constant time so we represent them by O(1) as they take constant time. Constant time is always considered as O(1). Now let us try to analyse the time complexities of some programs.

```
// Loop runs for n times
for (int i = 0; i < n; i++) {
      // Doing constant work
      System.out.println("Hello");

}
```

Here the loop runs for n times and every time we perform work which takes some constant time. Therefore, **T(n) = O(n)**.

```
// Outer loop runs for n times
for (int i = 0; i < n; i++) {
      // Inner loop runs for m times
      for (int j = 0; j < m; j++) {
            // Doing constant work
            System.out.println("Hello");
      }
}
```

Here the outer loop runs for n times whereas inner loop runs for m times. So every time the outer loop runs, the inner loop runs m times. So therefore, **T(n) = O(m*n)**.

```
// Loop runs for only log(n) times
for (int i = n; i > 0; i = i / 2) {
      // Doing constant work
      System.out.println("Hello");
}
```

Every time loop runs, n gets halved. So, the loop will run only $\log_2 n$ number of times.
Therefore, **T(n) = O($\log_2 n$)**.

Now let's examine the algorithms that we have studied before.

Bubble Sort, Selection Sort, Insertion Sort:
The loops run for (n*(n+1))/2 times. So, **T(n) = O(n^2)**.

Linear Search: The loop runs for n times. Therefore **T(n) = O(n)**.

Binary Search: Every time the loop runs, we neglect half the terms, so the loop runs for $\log_2 n$ times. Therefore, **T(n) = O($\log_2 n$)**.

Fibonacci Number: For calculating the nth Fibonacci, we need to calculate the (n-1)th Fibonacci and (n-2)th Fibonacci. So for calculating a Fibonacci we need to calculate two other Fibonacci numbers. Therefore **T(n) = O(2^n)**.

Now let's make a time stack where the complexities are written in increasing order. Constant time is the best complexity whereas the exponential proves to be the worst.

Objects are the basic units of Object-oriented programming. Objects are the real world entities about which we code. Objects have properties and behavior. For instance take an example of a car which has properties like its model and has four tyres and behavior of changing speed and applying brakes.

State of the objects is represented by data members and their behavior is represented by methods.

Now many objects share common properties and behavior. So a group of such properties and behavior is made that can generate many instances of itself that have similar characteristics. This group is called a class.

Classes are the blueprints from which objects are created. Like there are many cars which share common properties and behavior. The class provides these properties and behavior to its objects which are the instances of that class.

For example a vehicle class might look like this:

```
public class Vehicle {

    public String brand;
    protected String model;
    private double price;
    int numWheels;
    int yearOfManufacture;
    String color;

    public double getPrice() {
        return price;
    }

    public void printDescription() {
        System.out.println(brand + " " + model + " " + price + ""
+ numWheels);
    }
}
```

Now each vehicle will be a specific copy of this template.

A class in java contains:

- Data members
- Methods
- Constructor

Now let us talk about each of these in detail:

Data Members: Data members are the properties that are present in a class. The type of these properties can be modified by using special keywords called modifiers. Let us build our own student class and learn about them.

Static and Non Static Properties: Static properties are those that are common to all objects and belong to the class rather each specific object. So each object that we create doesn't have their copy. They are shared by all the objects of the class. We need to write the static keyword before it in order to make it static.

For e.g.:

```
static int numStudents;
```

Here the number of students in a batch is a property that isn't specific to each student and hence is static.

But the properties like name, Roll Number etc can have different values for each student and are object specific and thus are non static.

An important point to note is that whenever we create a new object only the non static data member copies are created and the static properties are stored within the class only! This could be considered a very memory efficient practice as static members of a class are made only once.

A general student class might look like this:

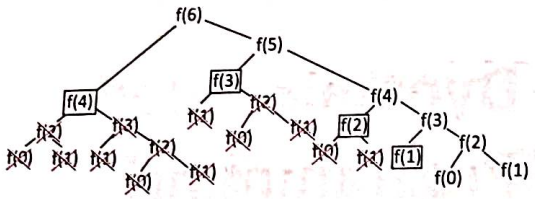
```
public class Student {
    static int numStudents;
    String name;
    int rollNo;
}
```

Access Modifiers:

Private: If we make any data member as private it is visible only within the class i.e. it can be accessed by and through the methods of the same class. So we can provide setters and getters function through which they can be accessed outside the class.

For instance in our student class we would like to keep the roll numbers for each student as private as we may not want any other person to modify those roll numbers by making them public. So we will make these private and provide the getter method to access the roll numbers of the students outside the student class.

```
public class Student {
    private int rollNo;

    public int getRollNo() {
        return this.rollNo;
    }
}
```



So in this way, we have reduced our time complexity from $T(n) = O(2^n)$ to $T(n) = O(n)$.
Though we have used an extra space of $O(n)$ in memoization method!

II. **Optimal Substructure:** A problem is said to have the optimal substructure property if the optimal solution of that problem can be obtained from the optimal solutions of the sub problems. These problems can also be solved using dynamic programming!

Assignment:

1. Two strings are given. Find out the longest subsequence of the two strings.
2. Two strings are given. If the cost of each operation of insert, delete and replace is 1, find out the minimum cost to make the second string same as the first string. This problem is famously known as the EDIT DISTANCE PROBLEM.

9 Object Oriented Programming I

Programming languages help a programmer to translate his ideas into a form that the machine can understand. The method of designing and implementing the programs using the features of a language is a programming paradigm. One such method is the Procedural programming which focuses more on procedure than on data. It separates the functions and the data that uses those functions which is not a very useful thing as it makes our code less maintainable. This leads us to another technique called the object oriented programming.

What is OOPs?

Object means any real world entity like pen, car, chair, etc. Object oriented programming is the programming model that focuses more on the objects than on the procedure for doing it. Thus we can say that oops focuses more on data than on logic or action. In this method the program is made in a way as real world works. In short it is a method to design a program using objects and classes.

Advantages of OOPs:

1. It is more appropriate for real world problems
2. OOP provides better modularity :
   Data hiding
   Abstraction
3. OOPs provides better reusability :
   Inheritance
4. OOP makes our code easy to maintain and modify :
   Polymorphism

Concepts in OOPs: One of the major shortcomings of the procedural method is that the data and the functions that use those data are separated. This is not preferred as if a data member is changed then all the functions related to it have to be changed again and again. This leads to mistakes and maintaining the code becomes tough.

How to solve?

The best way to solve this problem would be to wrap the data and all the functions related to them in one unit. Each of this unit is an object.

EXPONENTIAL TIME – $O(2^n)$ Fibonacci series
CUBIC TIME – $O(n^3)$ Multiplication of Matrices
SQUARED TIME – $O(n^2)$ Bubble, Selection, Insertion Sort
O(nlog(n)) Merge Sort
LINEAR TIME – O(N) Linear search
LOG TIME – O(log(n)) Binary Search
CONSTANT TIME – O(1) Printing, Algebraic operations

Space Complexity: Total amount of computer memory required for the execution of an algorithm is its space complexity. Auxiliary space is the extra space or temporary space used by an algorithm. Space complexity includes both auxiliary space and the space required by the input.

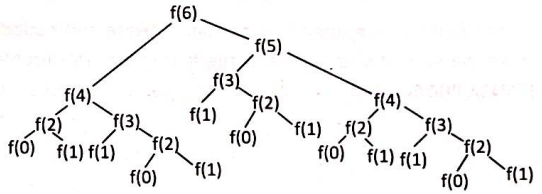
It is always better to use minimum of auxiliary space. Like if we have to calculate the transpose of a 2-D matrix, one way would be to make a new 2-D array and copy the elements from one matrix to another. But this wastes a lot of memory. So a better way would be to convert the original array into its transpose saving a lot of memory.

Well there is always a trade-off between improving time and space complexity. So improving one may lead to bad performance with respect to the other! Thus we need to wisely choose between the two.

8 Dynamic Programming

Dynamic programming is a technique to solve complex problems in more efficient manner. Many times we end up solving some of the sub-problems repeatedly. To avoid repetitive computation, we store the results of the sub problems and use them whenever we require them. This though uses extra space but provides time efficiency. Dynamic programming can be applied to problems, which have the following attributes :

I. **Overlapping Sub-Problems :**



Consider the case of calculation of a Fibonacci number. We can see that to calculate Fibonacci of 6 we need to calculate Fibonacci of 4 and 5, but to calculate Fibonacci of 5 we need to calculate the Fibonacci of 4 again. So this makes us calculate Fibonacci of some numbers multiple times. Thus this problem is said to have overlapping sub problems. We use the technique of Dynamic Programming for solving this.

To solve our problem of Fibonacci efficiently we can save the Fibonacci of a number once it is calculated so that if it is needed again we can simply use the saved value.

There are two ways to save the values so that they can be reused:

Memoization (Top-Down): One way would be to save the values in an array and pass this array as a parameter in the recursive function. This is the top-down approach which is used in recursive problems

Tabulation (Bottom-Up): Another method would be to iteratively calculate the Fibonacci number In this we start from the first entry and move to fill the table. The last entry of the table generall gives us the answer.

Default: When we explicitly don't write any modifier it is default. This modifier is package friendly i.e. it can be accessed anywhere within the same package.

Protected: It is accessible only within the same package but can be accessed outside the package using inheritance.

Public: It is accessible everywhere.

An important point to note here is that it is better to make a variable private and then provide getters and setters in case we wish others to view and change it than making the variable public. Because by providing setter we can actually add constraints to the function and update value only if they are satisfied (say if we make the marks of a student public someone can even set them to incorrect values like negative numbers. So it would be wise to provide a setter method for marks of the student so that these conditions are checked and correct marks are updated).

Final Keyword : Final keyword can be applied before a variable, method and a class. A final variable is one whose value can't be changed. So we can either initialize a final variable at the time of declaration or in a constructor. The value of a final variable can be assigned only once. A final method is one that can't be overridden. Making a class final means it can't be inherited. (like the String class in java is final). For example if we want the number of students in a class to be equal to 40 and we don't wish to change this number in any circumstances then this data member would be called final.

```
public class Student {
    final int noOfStudents = 40;
}
```

An important point to note here is that if a static variable is made final then they must be assigned a value with their declaration. This behavior is obvious as static variables are shared among all the objects of a class; creating a new object would change the same static variable which is not allowed if the static variable is final.

Methods: After having discussed about the data members of a class let us move onto the methods contained in the class relating to the data members of the class. We made many members of the class as private or protected. Now to set, modify or get the values of those data members, public getter and setter methods can be made and called on the objects of that class. The methods are called on the object name by using the dot operator.

Now let us look at various modifiers that can be used to modify the types of methods and the differences between them.

Static v/s Non Static Methods: Like data members, methods of a class can also be static which means those methods belong to the class rather than the objects for the class. These methods are directly called by the class name.

As the static methods belong to a class we don't need any instance of a class to access them. An important implication of this point is that the non static properties thus can't be accessed by the static methods as there is no specific instance of the class associated with them (the non static properties are specific to each object). So, non static members and the 'this' keyword can't be used with the static functions. Thus these methods are generally used for the static properties of the class only!

The non static methods on the other hand are called on an instance of a class or an object and can thus access both static and non static properties present in the object.

The access modifiers work the same with the methods as they do with the data members. The public methods can be accessed anywhere whereas the private methods are available only within the same class. Thus private methods can be used to work with the data members that we don't wish to expose to the clients.

Now let us add some methods to our student class.

```
public class Student {
    final static int noOfStudents = 40;
    String name;
    private int rollNo;

    public String getName() {
        return this.name;
    }

    public int getRollNo() {
        return this.rollNo;
    }

    public static int getNumStudents() {
        return Student.noOfStudents;
    }
}
```

Constructor: As we have our student class ready, we can now create its objects. Each student will be a specific copy of this template. The syntax to create an object is:

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

    Student s = new Student();
    // s.name - will give access to this student's name
}
```

Here the person class is the base class or the super class where as the student class is the subclass and inherits the properties name, age and methods displayInfo from the person class. The student class also has its specific property of marks and a method displayMarks. Also note the use of the 'extends' keyword which specifies that the subclass extends or inherits the properties of the base class.

Thus, we can use the data members and the methods of the parent class and also have our own data members and methods.

Now suppose we wish that the displayInfo method of the person class should not be used as such for the student class and must be changed a little to display the marks as well. This could be achieved by overriding the method of the person class in the student class. This is done as follows by using '@Override' above the method to be overridden. '@Override' just tells the compiler that the function is being updated and it is not compulsory to write it.

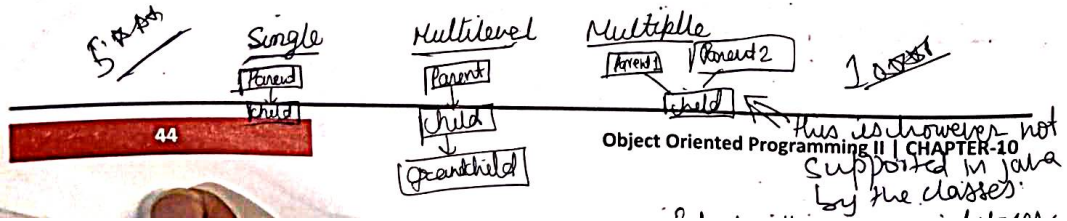
```
public class student extends person {

    int marks;

    @Override
    public void displayInfo() {
        System.out.println(this.name + " " + this.age + " " + this.marks);
    }

    public static void main(String[] args) {
        student obj = new student();
        obj.name = "Rahul";
        obj.age = 15;
        obj.marks = 80;
        obj.displayInfo();

        // OUTPUT
        // Rahul 15 80
    }
}
```

Now when we call the 'displayInfo' method on any student object the overridden method in the student class would be called.

Types of Inheritance: There are various types of inheritance like:

Single inheritance: here a single subclass inherits from a single base class. Like a student class inherits from a person class.

Multilevel inheritance: there is chain of classes inheriting from one another. Like a person class inherits from a mammal class which in turn inherits from the animal class!



Multiple inheritance: here a single subclass inherits from more than one parent classes. This type of inheritance is not supported in java by the use of classes. This can only be supported with the help of interfaces about which we will learn later. The reason for which it is not supported is that if there is a method by the same name present in both the parent classes and that is called on the object the JVM would not know which method to call for. So to avoid this confusion this type of inheritance is not supported in java.

We know that to refer to any instance of a class or any method of the current class, the 'this' keyword is used. Now if that class extends from another class, an instance of the parent class is created implicitly whenever an instance of the subclass is created. The instance of the parent class is referred by the **'super'** keyword. Let us look at the uses of the super keyword:

1. The 'super' keyword can be used to refer immediate parent class instance variable.
2. The 'super' keyword can be used to invoke immediate parent class method.
3. 'super()' can be used to invoke immediate parent class constructor.

Polymorphism: Now let us consider a situation a little different from the present context. Suppose we want to write functions to add integers, decimal numbers and strings. For that we will need to write three different functions with different names. This means that for each new data type we will need to choose a different name for the function whose purpose is the same. This would quite mess up the things. Thus, to make our code maintainable, we will now learn about the third pillar of the object oriented programming – POLYMORPHISM. Thus polymorphism is a way in which something behaves differently depending on its call. In other words, it is same name but different forms! In fact the ability to override the methods of the base class in the child class is also a form of polymorphism. Let us learn about polymorphism and its types in more detail.

Polymorphism in java is a concept by which we can perform a single action by different ways. There are two types of polymorphism in java:

1. Compile time polymorphism
2. Run time polymorphism

Compile Time Polymorphism: The compile time polymorphism is implemented by **method overloading** in java.

It means that there is more than one function with the same name in a class but with different signatures. Thus the functions with the same name can have different parameter types, number and even different order. Note that the return type is not the part of the function signature. Thus method cannot be overloaded by changing the return type of the function as then the compiler would not know which method to call for as the parameters are the same. Now let us look at an example to demonstrate the function overloading where the parameters can be of different type and can vary in number and order.

Q) What is an instance of a class?
⇒ In class-based programming, objects are created from classes by subroutines called constructors, and destroyed by destructors; an object is an instance of a class, and may be called a class instance or class object; instantiation is then also known as construction.

The signature for the queue class:

```
public class Queue {

    protected int[] data;
    protected int front;
    protected int size;

    public int size();

    public boolean isEmpty();

    public void enqueue(int item) throws Exception;

    public int dequeue() throws Exception;

    public int front() throws Exception;

    public void display();
}
```


10 Object Oriented Programming II

The second pillar of the object oriented programming is Inheritance. Consider that we have a person class and now wish to make classes for students and teachers as well. All the students and teachers have properties like name and age which are not specific to them instead are general properties of the person class. If we include these properties in the student and teacher class as well, this would just be the repetition of code. Instead it would be better if we reuse the code of the person class and include more specific properties like marks in student class and subject to be taught in the teacher class. This reusability of code is achieved through inheritance.

Inheritance: Inheritance is a mechanism in which one object acquires all the properties and behaviors of the parent object. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Now let us make a person class and then use it to build our student class

```
public class person {

    String name;
    int age;

    public void displayInfo() {
        System.out.println(this.name + " " + this.age);
    }
}

public class student extends person {

    int marks;

    public void displayMarks(){
        System.out.println(this.marks);
    }
}
```

A class classifies its members into three types: private, protected and public. Thus **Data Hiding** is implemented through private and protected members. These private and protected members can be accessed or set using public functions.

Also, the outside world needs to know only the essential details through public members. The rest of the implementation details remain hidden from the outside world which is nothing but **Abstraction**. Like for instance if we wish to know the aggregate marks of a student, we need not know how the aggregate is calculated. We are just interested in the marks and for that we only need to know that we need to call a public function called 'totalMarks' on any student object.

We have an additional benefit that comes bounded with encapsulation which is **Modularity**!

Modularity is nothing but partitioning our code into small individual components called modules. It makes our code less complex and easy to understand. Like for example, our code represents a school. A school has many individual components like students, teachers, other helping staff, etc. now each of them are complete units in themselves yet they are part of the school. This is called modularity.

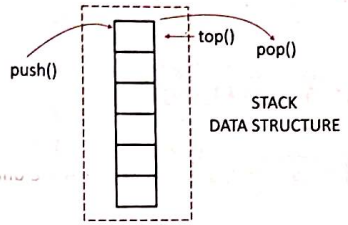
After having the overview of benefits of encapsulation let us dive into the details of data hiding and abstraction.

Let us suppose that we make all the data members of our student class as public. Now accidently if any of our client changes the roll numbers of the student objects, all are data would be ruined. Thus public data is not safe. So, to make or data safe, we need to hide our data by making it either private or protected! This technique is called information hiding. The private data can be accessed only via a proper channel that is through their access methods which are made public.

To use a class, we only need to know the public API of the class. We only need to know the signature of the public methods that is their input and output forms to access a class. Thus, only the essential details are sown to the end user and all the complex implementation details are kept hidden. This is Abstraction. After all sometimes ignorance is bliss!

Data Structures: Algorithms are just not enough! We also need to store our data. Normally we can store our data in variables but when we have huge amount of data, they just cannot be stored in primitive variables as it is not possible to even remember their names. Thus we need to build some structures where we can store the related data and process them easily. Let us learn about the stack and queue data structures whose meanings are implied by their names!

Stack: A stack is a linear data structure similar to a plate stack in the kitchen. In a stack we can add an element only to the top of it and also remove an element only from the top of it. Thus stack is based on the **Last In First Out (LIFO)** principle that is the element which is inserted first is the last one to be removed and similarly the last one to be inserted is the first one to be removed. The last element is always the top element in the stack. Stack has two important functions of **push** and **pop** which are used for insertion and deletion respectively. Stack allows operation at only one end! Stack can be implemented with the help of arrays or even linked lists about which we will learn later.



Queue: A queue is a linear data structure similar to a queue in front of the ticket counter. In a queue we can add an element only to the end of it and also remove an element only from the front of it. Thus queue is based on the **FIRST In First Out (FIFO)** principle that is the element which is inserted first is the first one to be removed and similarly the last one to be inserted is the last one to be removed. The first element is always the front element in the queue. Queue has two important functions of **enqueue** and **dequeue** which are used for insertion and deletion respectively. Queue allows operation at both the ends! Queue can be implemented with the help of arrays or even linked lists about which we will learn later.

Assignment: Now that we have learnt so much about classes and objects, it's time to put our knowledge on test. Make your own Stack and Queue. The signature of the class – its data members and methods has been given below for reference.

The signature for the stack class:

```
public class Stack {

    protected int[] arr;
    private int tos;

    public int size();

    public boolean isEmpty();

    public void push(int item) throws Exception;

    public int pop() throws Exception;

    public int top() throws Exception;

    public void display();
}
```

do yourself the for body's - i.e. define signature or methods yourself.

Notice that there is a 'new' keyword and a method is called with the same name as that of the class. The new keyword creates a java object and occupies the memory for it on the heap. The method called constructor, is a special method used to initialize a new object and its name is same as that of the class name.

Even though in our student class we haven't created an explicit constructor there is a default constructor implicitly there. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class. It initializes member data variable to default values (numeric values are initialized as 0, boolean's are initialized as *false* and references are initialized as *null*).

We can also create our own constructors. One important point to note here is that as soon as we create our constructor the default constructor goes off. We can also make multiple constructors each varying in the number of arguments being passed (i.e. constructor overloading). The constructor that will be called will be decided on runtime depending on the type and number of arguments specified while creating the object.

Below is our own custom constructor for our student class.

```
public class Student {
      String name;
      int rollNo;

      public Student(String name) {
            this.name = name;
      }

      public Student(String name, int rollNo) {
            this.name = name;
            this.rollNo = rollNo;
      }
}
```

The 'this' keyword: Here this is a keyword that refers to current object. So, **this.name** refers to the data member (i.e. name) of this object and not the argument variable name.

In general, there can be many uses of 'this' keyword.

1. The 'this' keyword can be used to refer current class instance variable.
2. The this() can be used to invoke current class constructor.
3. The 'this' keyword can be used to invoke current class method (implicitly)
4. The 'this' can be passed as an argument in the method call.
5. The 'this' can be passed as argument in the constructor call.
6. The 'this' keyword can also be used to return the current class instance.

There is also a special type of constructor called the **Copy Constructor**. Java doesn't have a default copy constructor but we can create one of our own. Given below is an example of a copy constructor.

```
public class Student {
      String name;
      int rollNo;

      public Student(String name, int rollNo) {
            this.name = name;
            this.rollNo = rollNo;
      }

      public Student(Student s) {
            this.rollNo = s.rollNo;
            this.name = s.name;
      }
}
```

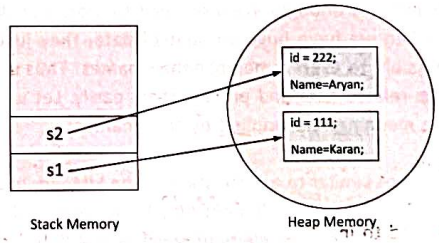
After leaning so much about the objects and constructors let us summarize the initialization of an object in simple steps.

As any new object is created, following steps take place:

1. Memory is allocated on heap and its reference is stored in stack.
2. The data members are initialized to their default values. (Only the non static properties are copied!)
3. The 'this' keyword is set to the current object.
4. The instance initialize is run and the fields are initialized to their respected values.

The constructor code is executed.

The final memory map of the instance of our student class is given below:



Encapsulation-The First Pillar of OOPs: We learnt that one of the major disadvantages of the procedural paradigm was that the functions and the data were separated which made the maintainability of the code poor! To overcome this we made classes which contained both the data members and their methods. This wrapping up of data and its functions into a single unit (called class) is known as **Encapsulation**.

```
public void add(int a, int b) {
    return a + b;
}

// Change in the number of parameters
public void add(int a, int b, int c) {
    return a + b;
}

// Change in the type of parameters
public void add(double a, double b) {
    return a + b;
}
// Change in the order of parameters
public void add(double a, int b) {
    return a + b;
}
```

Here the name for all the functions is the same but the compiler chooses which function to call depending on the parameters passed.

Now suppose that we do not know how many parameters would be passed in the function call. Thus it would not be possible for us to write functions with all the possible parameters. Thus to solve this problem we can use 'varargs' (variable arguments). In the function parameters we put three dots after the data type in the function declaration. This specifies the use of varargs. Note that there can just be one vararg in a function and also this has to be the last parameter! Below we have an example to demonstrate the same.

Note that compile time polymorphism can also be implemented by the use of generics which will be discussed later. Also operator overloading is not allowed in java.

```
public void display(String... values) {
    for (String s : values) {
        System.out.print(s + " ");
    }
}

public static void main(String[] args){
    display(); //No output
    display("hello"); //hello
    display("hello","world"); //hello world
}
```

Runtime Polymorphism: Runtime polymorphism is the process in which the call to an overridden method is resolved at the run time rather than at the compile time. When an overridden method is called by a reference, java determines which version of that method to execute depending on the type of the object it refers to. Now let us first understand method-overriding in more detail.

Method Overriding: When the derived class has a method same as that declared in the parent class, it is known as method overriding. Thus for method overriding the function in both the base and the derived class should have the same signature. Also note that the function in the derived class cannot be more restricted than that of its parent class. This means that if we are overriding a public function from the parent class, it can't be private or protected in the derived class as this will throw an exception.

In general, all the virtual functions can be overridden in java. By default, all non-static functions are virtual functions in java. Only the ones marked with final keyword which can't be overridden and the private functions which can't be inherited are non-virtual.

Now after having learnt about method overriding let us learn how runtime polymorphism is implemented.

Let us consider a context where we have a parent class P and a child class derived from P called C. Now while making an object there can be four combinations depending on the type of the object and its reference. Let us understand each in detail:

'P obj = new P()' : here the object is of P type and its reference is also of type P. Thus we can call any method of the class P on 'obj'.

'P obj = new C()' : here the object is of type P but its reference is of type C. Now let C class override a function 'func()' of class P. Now if we call the method func() on the object 'obj' of class P, since it refers to the object of the subclass C and the subclass overrides the parent class method, the subclass method is invoked at the run time. Now as the method invocation is decided by the JVM at the run time, this is known as runtime polymorphism.

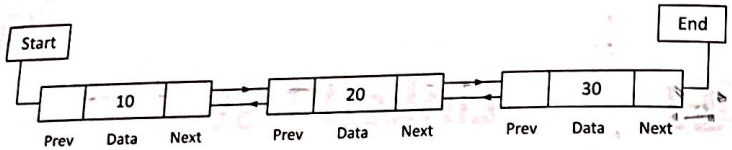
'C obj = new P()' here we are making the object of C class which is the subclass of P and making it point to an object of type P. This is not allowed in java and it throws a compile time error. This is because there maybe some data members and methods in the subclass which are absent in the parent class. Thus if we make the object of class C refer to object of class P, we won't be able to access those data members and methods. Thus a compile time error is raised whenever we try to do so.

'C obj = new C()' : here the object is of C type and its reference is also of type C. Thus we can call any method of the class C on 'obj'.

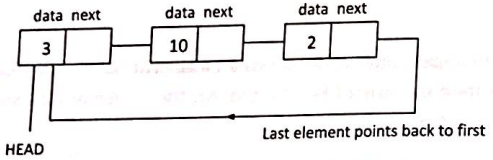
Thus we can summarize the above points with a rule of thumb:

The Compiler has its eyes on the LHS or the reference and it will compile code congruent to the LHS. Whereas JVM has its eyes on the RHS or the object that got created and it will invoke functionality congruent to the RHS.

Now there are two exceptions to this rule because run time polymorphism cannot be achieved by data members and static functions. This is because data members and static functions are not overridden in the subclass. Instead they are the properties of the class and not the object! Thus whatever the case be data members and static functions that are accessed will always be that of the parent class.



Circular Linked List: This is a type of singly linked list in which the last node points to the starting node.



Advantages of Linked Lists: They are dynamic in nature as their size is not fixed and can be changed during the runtime.

→ Insertion and deletion operations can be easily implemented at any point in the linked list.

Disadvantages of Linked Lists: Some amount of memory is wasted in storing the references for the next nodes.

Unlike array, no element can be accessed randomly in a linked list. Thus the search takes constant time. Also reverse traversing is difficult in a singly linked list.

12 Object Oriented Programming III

In the last module we have learnt about the various types of inheritance. Among them one was the multiple inheritance. Multiple inheritance was where the subclass had more than one parent classes. But this is not allowed in java as if there was a common method that existed in both the base classes, the compiler would not know which method to call. Thus this gives a compile time error and hence is not allowed. However multiple inheritance can be achieved by the use of interfaces in java. Let us learn about them in detail.

Interfaces: An interface is a reference type in java. It is similar to classes but is purely abstract! Interface is a collection of abstract methods. In an interface, all the **methods are public and abstract** and all the data members are public static and final. Well this is because an interface cannot be instantiated that is an object cannot be made. Thus an interface doesn't even have a constructor. All the methods don't have their body as all of them are abstract.

For declaring an interface we need to use the keyword 'interface' followed by the name of the interface. Interface can contain as many abstract methods that we need. An interface is by default abstract so we do not need to specify it explicitly. Also, all the methods in the interface are also abstract and public, thus we don't need to specify the 'abstract' keyword!

Let us write our person interface to understand clearly.

```
interface person {

    // public abstract methods
    public void name();
    public void age();
    public void gender();

}
```

Here the person interface has three abstract methods that don't have their body. Let us now learn how to use an interface.

Assignment: In the last module we learnt about the data structures stack and queue. A major problem with these data structures is that they are fixed in their size. Thus we can solve this problem by making stack and queue such that their size can be changed as per the insertion and deletion of data.

⇒Using the concept of inheritance, make a **dynamic stack and a dynamic queue** by extending the stack and queue that we made in the previous module.

11 Linked List

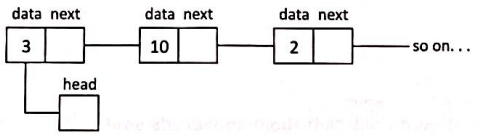
We have used arrays to store multiple values under a single name. But using arrays has its disadvantages that they are static and hence their size cannot be changed. Another problem with arrays is that it needs contiguous memory to store its values.

To overcome these problems, we use another data structure called linked lists. Linked list is a linear data structure that contains sequence of elements such that each element has a reference to its next element in the sequence. Each element in a linked list is called **"Node"**. Each Node contains a data element and a Node next which points to the next node in the linked list. In a linked list we can create as many nodes as we want according to our requirement. This can even be done at the runtime. The memory allocation done at the run time by the JVM is known as **DYNAMIC MEMORY ALLOCATION.** Thus, linked list uses dynamic memory allocation to allocate memory at the run time.

General Implementation: The linked list class contains a private class Node and a Node head. The Node head points to the starting of the linked list. Every Node contains a data element and a Node next which points to the next node in the linked list. The last node points to null.

Different Types Of Linked Lists:

Singly Linked List: These are normal linked lists in which every node points to the next node and head node points to the starting node.



Doubly Linked List: These are linked lists in which every node points to the next node as well as the previous node. In this case, Node class has a data element, a Node next pointing to the next node and a Node previous pointing to the previous Node. In this case, Node head points to the starting of the node whereas a Node tail points to the last node of the linked list. The previous of first node and the next of the last node points to null.

In the last module we talked about abstraction in java. **Abstraction** is the process of hiding the implementation details from the end user and only showing the necessary functionality. There are two ways to achieve abstraction. It can be achieved by the use of Abstract Class or Interfaces. We will talk about in Interfaces in the later modules.

Abstract Methods & Classes: The literal meaning of the word abstract is that something exists in idea but not physically defined. Similar are the abstract methods and classes in java.

A method that doesn't have any implementation and is marked as abstract is an abstract method. Similarly a class marked abstract, which needs to be extended and all its methods implemented, is known as an abstract class.

An abstract class cannot be instantiated that is its object can't be made. An abstract class can have data members, abstract and non abstract methods and also constructors. An important point to note here is that any non-abstract class cannot have abstract methods. Thus if any class has abstract methods, it has to marked abstract. Also, while extending an abstract class all the abstract methods have to be provided with the implementations or the subclass should also be marked abstract. Let us understand this clearly by making an abstract person class and extend it to make our student class.

```
abstract class person {
      abstract void display();
}

class student extends person {

      // Display method of the person class is implemented
      void display() {
            System.out.println("this is a student");
      }

      public static void main(String[] args) {
            student obj = new student();
            obj.display();
            // Output: this is a student
      }
}
```

Now having learnt thoroughly about the three pillars of OOPs, let us discuss various modifiers that can be used with the data members, methods and classes. There are two types of modifiers: Access modifiers and Non-Access modifiers. The access modifiers change the scope of the properties, methods and that of the class itself whereas the non-access modifiers achieve other functionality. We have already discussed about the use of modifiers with data members, methods and classes. Let us now summarize it all together.

Modifiers & Data Members:

Access Modifiers:

Public: they are accessible everywhere.

Protected: they are available only within the package and outside the package using inheritance.

Private: they are available only within the class.

Default: they are available only within the package.

Non-Access Modifiers:

Static: it creates data members that are common to all the objects of the class. They belong to the class and can be accessed by the class name followed by a dot and their name.

Final: it creates data members whose value cannot be changed. Thus their value is defined either at the time of declaration or in the constructor.

Modifiers & Functions:

Access Modifiers:

Public: the method is available anywhere, even outside the package.

Protected: the method is available within the package and outside the package using inheritance.

Private: the method is available only in the class.

Default: the method is available only within the package.

Non-Access Modifiers:

Static: it creates methods that are independent of the instances of the class. They cannot use the non-static data members of the class and can be accessed by the class name followed by a dot and their name.

Final: it creates methods which cannot be overridden and hence can't be changed in the subclasses.

Abstract: it creates a method declared without any implementation which has to be provided in the subclass. Thus an abstract method can never be final.

Modifiers & Classes:

Access Modifiers:

Public: this class can be accessed anywhere.

Private: we can have a private class only within a public or a default class and its access is limited to only those classes.

Default: it can be accessed anywhere within the package but not outside of it.

Non-Access Modifiers:

Final: a final class is the one that cannot be inherited.

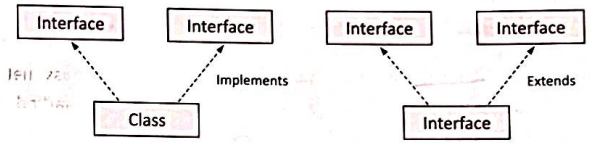
Abstract: a class which is marked abstract and has to be extended and its method implemented is known as an abstract class.

As a class can extend another class, similarly interfaces can also be 'implemented' by the classes. This is done by using the keyword 'implements' followed by the name of the interface. The class which implements an interface has to implement all the methods of that interface. Thus an interface can be thought of as a contract signed by the class to implement all the functions of that interface, if the class that implements an interface doesn't implement all the abstract methods then that class must be marked abstract. While overriding the methods of the interface, the signature and the return type of the method cannot be changed. Another point to note here is that while a class can extend only one class at a time, it can implement more than one interface at a time. Now let us make a student class to implement our person interface that we just made.

```
class student1 implements person {

    // Providing the implementation of all methods
    public void name() {
        System.out.println("the name method");
    }

    public void age() {
        System.out.println("the age method");
    }

    public void gender() {
        System.out.println("the gender method");
    }

    public static void main(String[] args) {
        student1 s = new student1();
        s.name(); // The name method
        s.age(); // The age method
        s.gender(); // The gender method
    }
}

abstract class student2 implements person {

    // Abstract class as all methods are not implemented
    // Providing the implementation of name method only
    public void name() {
        System.out.println("the name method");
    }

    public static void main(String[] args) {
        student2 s = new student2(); // error
        // abstract class can't be instantiated
    }
}
```

Note here that the student1 class that implements the person interface implements all the methods of the interface whereas the student2 class doesn't implement all the methods of the interface and hence is marked abstract.

Multiple Inheritance:



Multiple Inheritance in Java

We know that multiple inheritance cannot be achieved by classes in java but interfaces can implement multiple inheritance. We just saw that a class can implement multiple interfaces. This can be thought of as multiple inheritance as the class is implementing methods of multiple interfaces. Also as a class can extend another class, interfaces can also extend other interfaces by using the 'extends' keyword. Also, as interfaces showcase multiple inheritance, an interface can extend multiple interfaces. Thus when a class implements an interface that has extended another interface, it will have to implement the methods of both the interfaces failing which it would be marked as abstract. Now let us write an interface that extends from another interface and our student class implements these interfaces.

```
interface male {
    public void gender();
}

interface female {
    public void gender();
}

// Interface extending 2 interfaces at a time
interface person extends male, female {
    public void name();
}

class student implements person {
    public void name() {
        System.out.println("the name method");
    }

    public void gender() {
        System.out.println("the gender method");
    }

    public static void main(String[] args) {
        student s = new student();
        s.name(); // The name method
        s.gender(); // The gender method
    }
}
```

In a class, multiple inheritance is not supported as there is ambiguity but in the case of interfaces, there is no ambiguity as the implementation is provided within the implementation class only.

Try block - The code which can cause an exception is enclosed within try block.

Catch block - The action to be taken when an exception has occurred is done in catch block. It must be used after the try block only.

Finally block - Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not.

Here is a sample code to explain the same.

```
Scanner s = new Scanner(System.in);

System.out.println("Enter dividend");
int dividend = s.nextInt();

System.out.println("Enter divisor");
int divisor = s.nextInt();

try {
      int data = dividend / divisor;
      System.out.println(data);
} catch (ArithmeticException e) {
      System.out.println("Divide by zero error");
} finally {
      System.out.println("Finally block is always executed");
}
System.out.println("Rest of the code...");
```

Note:

1. Whenever an exception occurs statements in the try block after the statement in which exception occurred are not executed
2. For each try block there can be zero or more catch blocks, but only one finally block.

Creating an Exception/User Defined Exceptions: A user defined exception is a sub class of the exception class. For creating an exception you simply need to extend Exception class as shown below:

```
public class InvalidInputException extends Exception {
      private static final long serialVersionUID = 1L;
}
```

Throwing an Exception: Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example if input to the factorial method is a negative number, then it makes more sense for the factorial to throw an exception and the method that has called factorial method to handle the exception.

Here is the code for the factorial method:

```
public static int fact(int n) throws InvalidInputException {
      if (n < 0) {
            InvalidInputException e = new InvalidInputException();
            throw e;
      }
      if (n == 0) {
            return 1;
      }
      return n * fact(n - 1)
}
```

The fact method throws an InvalidInputException that we created above and we will handle the exception in main.

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

      Scanner scn = new Scanner(System.in);
      System.out.println("Enter number");
      int n = scn.nextInt();
      int a = 10;
      try {
            System.out.println(fact(n));
            a++;
      } catch (InvalidInputException e) {
            System.out.println("Invalid input!! Try again");
            return;
      }
}
```

done & done

2. **Type casting is not required:** There is no need to typecast the object.
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Multiple Type Parameters: We can have multiple type parameters as well i.e. we can create a pair class where "first" and "second" can be of different types unlike the Pair class defined above where both have to be of same type.

```
public class Pair<T, S> {
    T first;
    S second;
}
```

Its instance can be created as follows:

```
Pair<Integer, String> pair = new Pair<Integer, String>();
```

So here pair.first is a Integer and pair.second is a String.

Multilayer Generic Parameters: The generic parameters can be multiplayered.

```
Pair<Pair<Integer>> pLayered = new Pair<>();
```

⇒ Here pLayered.first and pLayered.second are themselves pair of Integers.

⇒ Similarly we can add multiple layers to the parameters.

Bounded Type Parameters: Many a times when you might want to restrict the kinds of types that are allowed to be passed to a type parameter. Say we want to create a generic sort function. In order to sort elements we will have to compare them. The "<" or ">" are not defined for non–primitives. So instead we will have to use compareTo() method (in Comparable interface) which compares two objects and returns an int based on result.

Now in our sort function we should allow only those non-primitives who have compareTo() method defined for them or in other terms who have implemented the Comparable interface (as interface serves as a contract, so if a non-abstract class has implemeted Comparable method then we can be sure that it has compareTo() method). We can do this as shown below:

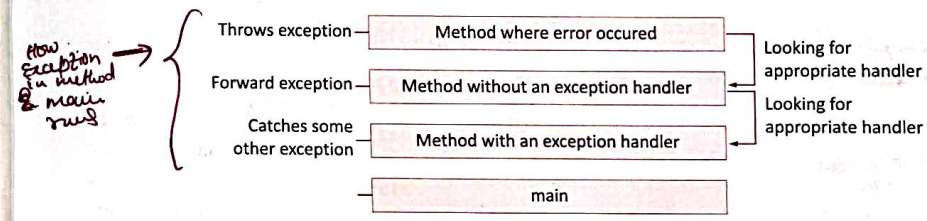
```
public static <T extends Comparable<T>> void sort(T[] input) {
    T temp;
    for (int i = 0; i < input.length; i++) {
        for (int j = 0; j < input.length - i - 1; j++) {
            if (input[j].compareTo(input[j + 1]) == 1) {
                temp = input[j + 1];
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }
}
```


When we write <T extends Comparable<T>> this means that only those parameters are allowed those who have implemented Comparable Interface.

Exception Handling: An exception is an event, which occurs during the execution of a program, and that disrupts the normal flow of the program's instructions. The **exception handling in java** is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The block of code that handles an exception is called exception handler. When an exception occurs the run time system first tries to find an exception handler in the method where the exception occurred and then searches the methods in the reverse order in which they were called for the exception handler. The list of methods is known as the call stack (shown below). If no method handles the exception then exception appears on the console (like we see ArrayIndexOutOfBoundsException etc)



Types of Exception:

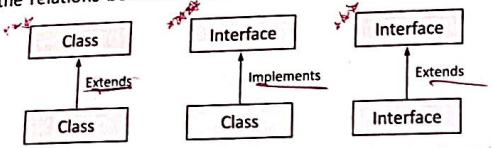
Checked Exceptions: These are exceptional conditions that we can anticipate when user makes mistake. For example computing factorial of a negative number. A well-written program should catch this exception and notify the user of the mistake, possibly prompting for a correct input. Checked exceptions *are subject* to the Catch or Specify Requirement i.e. either the function where exception can occur should handle or specify that it can throw an exception (We will look into it in detail later).

Error: These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction.

Unchecked Exception: These are exceptional conditions that might occur at runtime but we don't expect them to occur while writing code. These usually indicate programming bugs, such as logic errors or improper use of an API. For example StackOverflowException.

⇒ **Exception Handling:** Exception handling is achieved by using try catch and/or finally block.

All the relations between interfaces and class are summarized below.



Packages: A java package is a group of similar type of classes, interfaces and sub-packages.
Packages can be of two forms:

Built-in Packages:

User-defined Packages:

There are two main uses for creating packages. Firstly packages help us to organize our code better as similar classes and interfaces can be enclosed in one package which helps us to maintain them properly. The second usefulness of packages is that it **prevents name collisions**. Thus is there were no packages, no two classes would be able to have the same name. Thus, if we have packages we can have two classes by the same name in two different packages.

A package within a package is called a **sub-package.** It is created to categorize a package further.

There are already many built-in packages like java, lang, swing, util, io, etc we can also make our own user defined packages and further make classes and interfaces in them.

Now if we want to use a class or all the classes of a particular package in another package, we need to import that package in our package. Let us learn how we can do that.

There are three ways to access the package from outside of it.

Using the **'packagename.*'**: by using this, all the classes and the interfaces can be accessed outside of the package. However, the sub-packages cannot be accessed.

The **import** keyword is used for the same.

Using **'packagename.classname'**: by using this, only the specified class will be made available.

Using **full name**: we write the packagename followed by a dot and the class/interface name wherever we want to access.

```
package Demo;
public class person {
      public void display() {
            System.out.println("hello");
      }
}
//For accessing outside package using package name
import Demo.*;
//Using package name and class name
import Demo.person;
//Using full name
public static void main(String args[]){
      Demo.person obj = new Demo.person();
}
```


13 Object Oriented Programming IV

In the previous module we learnt about the linked list data structure. But in that our node class could store only integer values and thus our linked list was restricted to only integers. Thus if we wanted to make a character or a string linked list, we would need to make another linked list. Thus for each new data type or object we will need to make separate linked lists. To solve this problem we use Generics.

```
public static <T> T add(T a, T b) {
      return a;
}
```

Here <T> represents the type parameter and 'T' is an identifier that specifies a generic type name, it could have been any other letter like K, S etc.

Creating a Generic Class: Suppose we make a pair class to store two integers. Now if we want to have a pair of two chars, strings or double then we will have to create separate pair classes for each of them. Generics allow us to create a single Pair class that will work for different types.

```
public class Pair<T> {
      T first;
      T second;
}

// Pair of two Integers
Pair<Integer> pInts = new Pair<Integer>();
// Pair of two Strings
Pair<String> pStrings = new Pair<String>();
```

So this class creates a pair which has two variables first and second of types specified in angle brackets (<>). The type parameters can represent only reference types, not primitive types. So, for the primitive data types like int, char, etc java has corresponding Wrapper classes Integer, Character etc.
A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.

There are mainly 3 advantages of generics. They are as follows:

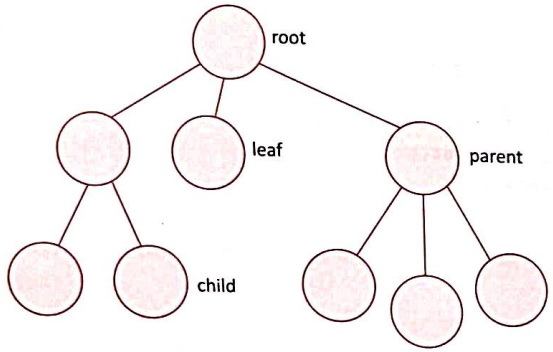
1. **Type-Safety:** We can hold only a single type of objects in generics. It doesn't allow storing other objects.

14 Generic Trees

Till now we have just learnt about the linear data structures like arrays, linked lists, stacks and queues. But how would we store the data like that of a file structure or maybe a family tree. In these, data is not related linearly. Thus we would need to use a **non linear data structure** where one element could be related to more than one element. One such data structure is a tree.

One of the most common examples of a tree is an XML document. The top-level document element is the root node, and each tag found within that is a child. Each of those tags may have children, and so on. At each node, the type of tag, and any attributes, constitutes the data for that node. In such a tree, the hierarchy and order of the nodes is well defined, which is an important part of the data.

A tree has nodes which are connected to their children via edges.



Basic terminology in a Generic Tree:

Node: It is the structure that contains the data about each element in the tree. Each element in a tree constitutes a node.

Root: The node at the top of the tree is known as the root node. That is, the starting of the tree is known as root.

Children: Each node can be connected to 0 or more nodes. These nodes are the children of this node.

Parent: A node which has a child is that child's parent node.

Siblings: A group of nodes with same parent.

Ancestor: A node reachable by repeated proceeding from child to parent.

Descendant: A node reachable by repeated proceeding from parent to child.

Leaves: These are the nodes that don't have any children.

Path: The path refers to the sequence of nodes along the edges of the tree.

Height of Tree: The height of the tree is the number of edges on the longest downward path from root to a leaf.

Depth: The depth of a node is the number of edges from the node to the root node.

Degree: The degree of a node is the number of children that node has.

General Implementation: Inside the tree class, we make a private class Node which contains a data element and an arraylist of nodes, children. The tree class contains the root node initially null.

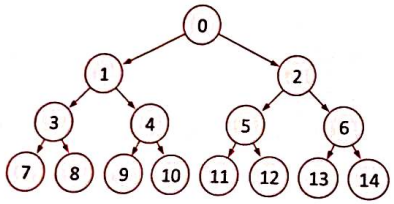
Tree Traversals: A tree traversal is a way of visiting all the nodes in the tree in a particular order.

1. **Pre-order Traversal**: Each node is processed before any nodes in its subtrees.
2. **Post-order Traversal**: Each node is processed after all nodes in its subtrees.
3. **Level-order Traversal**: Each node is processed level by level from left to right before any nodes in their subtrees.

15 Binary Trees

A binary tree is a special type of tree in which every node has at most two children. The first one is known as **left child** and second child is known as **right child**. A node in a binary tree may have 0, 1 or 2 children.

In the case of a Binary tree, we do not need an arraylist of nodes, children. Rather we only need two more nodes inside the node class as left and right.



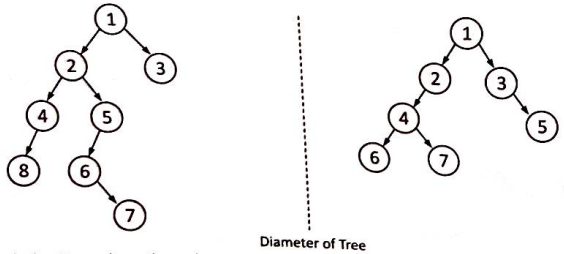
Traversals in Binary Tree: Apart from the traversals in generic trees, there is also a traversal called the in order traversal.

In-order Tree Traversal: In this each node is processed after the left subtree is processed but before the right subtree is processed.

So for a binary tree:

1. Pre-order Traversal: Node -> Left -> Right
2. In-order Traversal: Left -> Node -> Right
3. Post-order Traversal: Left -> Right -> Node

Diameter of Binary Tree: The diameter of a binary tree is defined as the number of nodes on the longest path between two leaves in the tree. It is not compulsory that the longest path must include the root node. Sometimes diameter is also known as the **width** of the binary tree.



Path that Doesn't go through root

Path goes through root.

Assignment:

1. Build a binary tree using its inorder and preorder traversals.
2. Build a binary tree using its inorder and postorder traversals.
3. Build a binary tree using its preorder and postorder traversals.

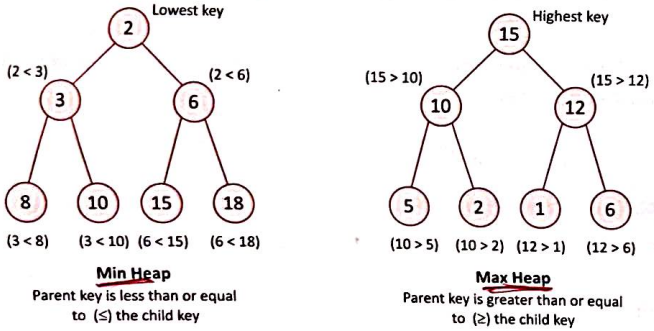
18 Heaps

Assume that we have a system which serves the request on the basis of the time duration for that request. That is, a request with minimum time duration is served first. In this case we need a priority queue which gives us the minimum element in a set of elements in O(1). This is achieved by Heap Data Structure. The maximum or minimum element is calculated in O(1).

Heap is binary tree with two additional properties:

1. Shape property: A heap is a **complete binary tree**. That is all the levels of the tree are fully filled except possibly the last one. If the last level is not fully filled then that level is filled from left to right. The height of the complete binary tree is O(log(n)).
2. **Heap Order property**: The key stored in each node is either greater than or equal to (Max-Heap) or less than or equal to (Min-Heap) the keys in the node's children.

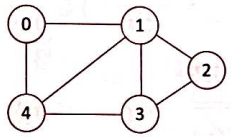
So in case of a Max-Heap, the topmost element is the maximum in that heap while in case of a Min-Heap, the topmost element is the minimum in that heap.



19 Graphs

Consider a city. City has a lot of places which can be visited by a tourist. There are multiple roads connecting one place to another directly or via another place. To represent this kind of data we use graphs.

Graph is a non linear data structure consisting of vertices and edges between them.



Vertex: An individual data element of a graph is known as vertex. It can be considered similar to a Node in a Tree.

Edge: An edge is the connecting link between two vertices. An edge from A to B is represented as (A, B).

There are two types of edges:

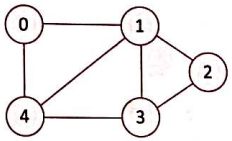
1. **Undirected edge**: This is bidirectional edge. If there is a bi directed edge from A to B then edge (A, B) is equal to the edge (B, A).
2. **Directed edge**: This is unidirectional edge. If there is a uni directed edge from A to B then edge (A, B) is not equal to the edge (B, A).

Degree: The number of edges connected to a vertex is called as the degree of that vertex.

Adjacency: Two vertices are adjacent if they are connected to each other through an edge.

Path: Path represents a sequence of edges between two vertices.

Different Representations :



17 Hash Tables

Consider that we want to make a population map of different countries. One way would be to use a linked list in which every node has two data elements, country name and population. But then get, set and remove in that list would be O(n). So to solve this problem in constant time we have hashtables and hashmaps.

Hashmap is a collection of **key-value pairs**. That is all the keys are mapped to a particular value. All the keys in a hashmap are unique. If we try to insert the same key again, the previous value is updated. Internally, the hashmap is an **array of linked lists**. Each list is known as a bucket.

Whenever a new key value pair is added, the **hash function** is called on the key to identify the bucket for this pair. The hash function includes a **Hash Code** function and a **compression function**.

What if two keys have the same bucket?

This is known as **collision**. We add both of the keys in the same list then. This increases the time to find an element in that list. Here comes the concept of load factor and rehashing.

The **load factor** is a measure of how full the hashmap is allowed to get before its capacity is increased. Generally if load factor is greater than 0.5, we increase the size of bucket array and rehash all the key value pairs again.

In case of **rehashing**, the hash code function remains the same whereas the compression function changes.

HashTable is also same as HashMap with some differences.

1. HashTable is synchronised whereas HashMap is not.
2. HashTable does not allow any null key or value whereas HashMap allows one null key and multiple null values.
3. HashMap is faster than HashTable.

HashMap - syntax and common functions.

```
// K, V can be any Data Type
HashMap<K,V> map = new HashMap<>();
```

```
// To get the value of the key
map.get(key);
```

```
// To insert a key-value pair
map.put(key, value);
```

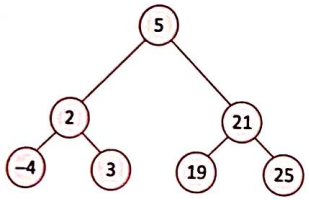


```
// To remove the key-value pair
map.remove(key);

// To check if it contains a key or not
boolean c = map.containsKey(key);

// Returns a set of all the keys
Set<K> keys = map.keySet();

// Returns a collection of all the values
Collection<V> values = map.values();

// Returns a Set of key-value pairs
Set<Map.Entry<K,V>> entries = map.entrySet();

// To iterate over all the pairs
for (Map.Entry<K,V> entry : entries) {
    // To get the key
    entry.getKey();
    // To get the value
    entry.getValue();
}
```

16 Binary Search Trees

A binary search tree is a special type of binary tree in which:

1. Every node in the left subtree of a node has a value less than or equal to the value of that node.
2. Every node in the right subtree of a node has a value greater than or equal to the value of that node.



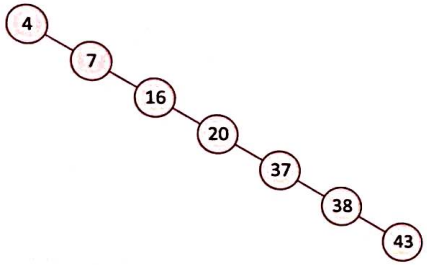
An important point to note in the binary search tree is that its inorder traversal is in sorted order. Like in the above case the inorder traversal would be -4, 2, 3, 5, 19, 21, 25. The numbers are in sorted order!

Searching in a Binary Search Tree: The main application of a Binary search tree is efficient searching. We compare the value to be found with the value of root node. If they are equal we return the root node. If the value to be found is less than the value of the root node, we search only in the left subtree and discard the right subtree. If the value to be found is greater than the value of the root node, we search only in the right subtree and discard the left subtree. Thus in each turn we are neglecting some of the elements without even traversing them. So using this data structure we do not have to linearly traverse the list. This type of searching algorithm is similar to that of the binary search.

Time complexity for searching in a Binary Search Tree:

Though the search algorithm is similar to that of a binary search, the time complexity is not log(n) every time. Instead it depends on the height of the tree.

Consider a binary search tree:



In this case the Time Complexity to search an element would be O(n). This means that the Time Complexity for searching an element in a Binary Search Tree is O(Height of the BST).

Only in the case of a complete/balanced binary search tree, the height of the tree is log(n) and hence only in that case Time Complexity is O(log(n)).

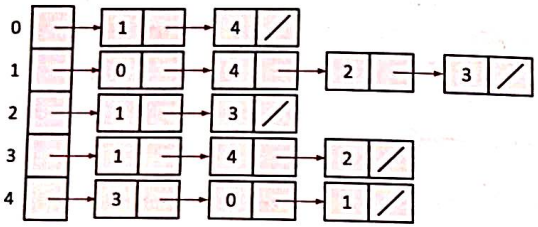
Assignment:

1. Given a binary tree, check whether it is BST.
2. Build a BST using a sorted array.

1. **Adjacency Matrix**: Adjacency Matrix is a 2D array of size A x A where A is the number of vertices in the graph. Let the 2D array be adjacencyMatrix[][] and adjacencyMatrix[i][j] = 1 implies that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric.

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

2. **Adjacency List**: An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be arr[]. An entry arr[i] represents the linked list of vertices adjacent to the i^{th} vertex.

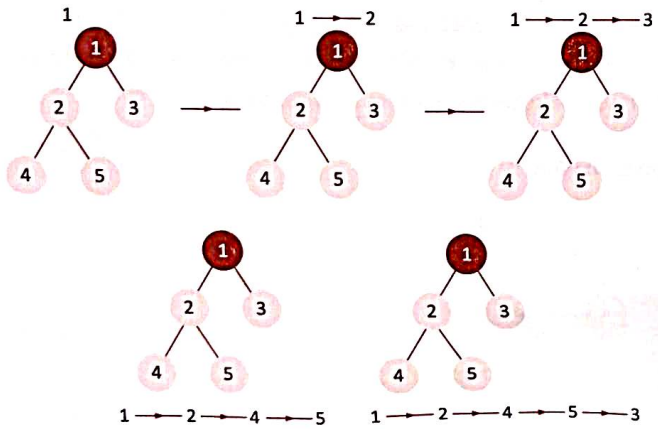


Graph Traversals: Graph traversal is a technique for searching a vertex or travelling in the graph. The technique decides the order in which each vertex is visited.

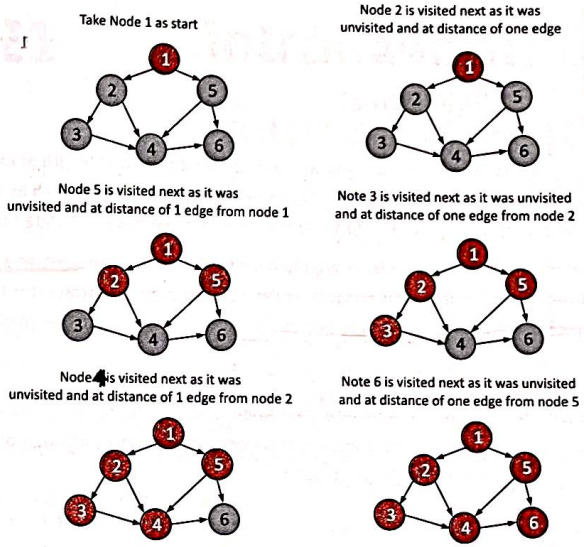
1. **Depth First Search (DFS)**: In this algorithm, we try to reach as far from the starting vertex as possible. That is we start at the starting vertex and explore as far as possible along each branch before backtracking.

2. **Breadth First Search (BFS)**: In this algorithm, we try to work closer to the starting vertex. That is we start at the starting vertex and explore all the neighbour vertices before moving to the next level neighbours. We move in a level wise order.

DFS



BFS :

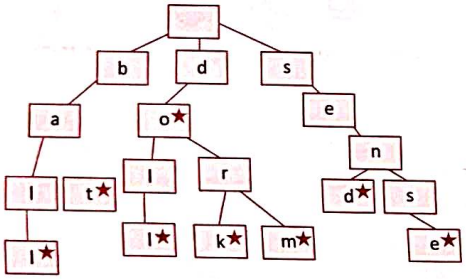


20 Tries
[Prefix Tree]

Consider that we have a large paragraph of size n and we have to find if a word of length m exists in it or not. If we use linear search that would take O(n*m) time. This is very bad for programs such as Microsoft word in which they have to find a word very quickly. To solve this problem, we have Trie Data Structure.

In this Data Structure, the keys of the normal trees are changed to characters. We used an arraylist to store children of a node but here we cannot use an arraylist as the search time in an arraylist is linear. Thus, we have to use a HashMap of Characters and Nodes to get the Node corresponding to the given character key in constant time.

General Implementation: The trie class has a private class Node. The class Node contains a Character data, a boolean isTerminal which determines if a word ends at this node or not and a HashMap of Character and Node, children. The trie class also contains the root Node.



Initially we add '\n' at the root node and all the words are its children. The node marked with star shows that the boolean isTerminal for them is true. Some of the words in the trie are bat, ball, do, doll etc.

Time Complexity for a trie: The formation of trie has a time complexity of O(n) where n is length of the paragraph but once the trie is created, the time for searching a word is only **O(m)** where m is the length of the word. So searching a word has been reduced from O(n*m) to O(m). This makes finding a word very fast because generally the size of word is small and we get nearly **constant searching time.**

21 Introduction to Game Programming

Till now we have learned how to write programs in java, about different algorithms and data structures. The output that we got for these programs was always on the console. But when we are using programming for development, we don't really run the programs through consoles rather we use applications like desktop applications, android applications, web applications, etc.

Java provides functionality for creating desktop applications like browser, a simple calculator or a game. Java has different classes for these purposes. Like we have ArrayList class for dynamic array, similarly we have different classes in java to create a **GUI (Graphical User Interface)**. To learn this type of programming we have to study **Java Swing**. Java Swing is the set of components for Java programmers that provide the ability to create GUI components. Swing components are used with Java classes.

We will study a few Java classes.

JFrame: JFrame is a Swing's top level container that renders a window on screen. A frame is a base window on which other components are added.

```
public class SwingJFrameDemo extends javax.swing.JFrame {
    public SwingJFrameDemo() {
        super("Demo program for JFrame");
    }
}
```

A java class which extends javax.swing.JFrame or simply extends JFrame can create a frame by calling the constructor of the super ie JFrame class.

```
JFrame frame = new JFrame("Demo Program for JFrame");
```

A new frame is created by creating an object of JFrame class.

```
// To set different Layout
frame.setLayout(new GridLayout());

// To add a child component
```

```
frame.add(new JButton());

// To change title for the window
frame.setTitle("Title");

// To change the size of the window
// X pixels by Y pixels
frame.setSize(100, 100);
```

JButton : To create a button we first create an object of class JButton.

```
JButton button = new JButton("Edit");
```

Now we simply add it to the frame we created using JFrame.

To change the behaviour of a button, we add an **ActionListener** on it using anonymous class.

```
//Creating an object of ActionListener class anonymously
button.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent e) {
        // Do everything here...

    }
});
```

Similarly we can use different classes to create a Desktop Application. Different classes are **JLabel, JTextField, JCheckBox** etc. VKP

make command on such questions

More Focus on Questions like

1. Patterns all types
2. Recursive functions
3. More use of strings and Arraylist.
4. Dynamic Programming

* Data structures & OOPS (always)

A see for try, catch & finally

NOTES