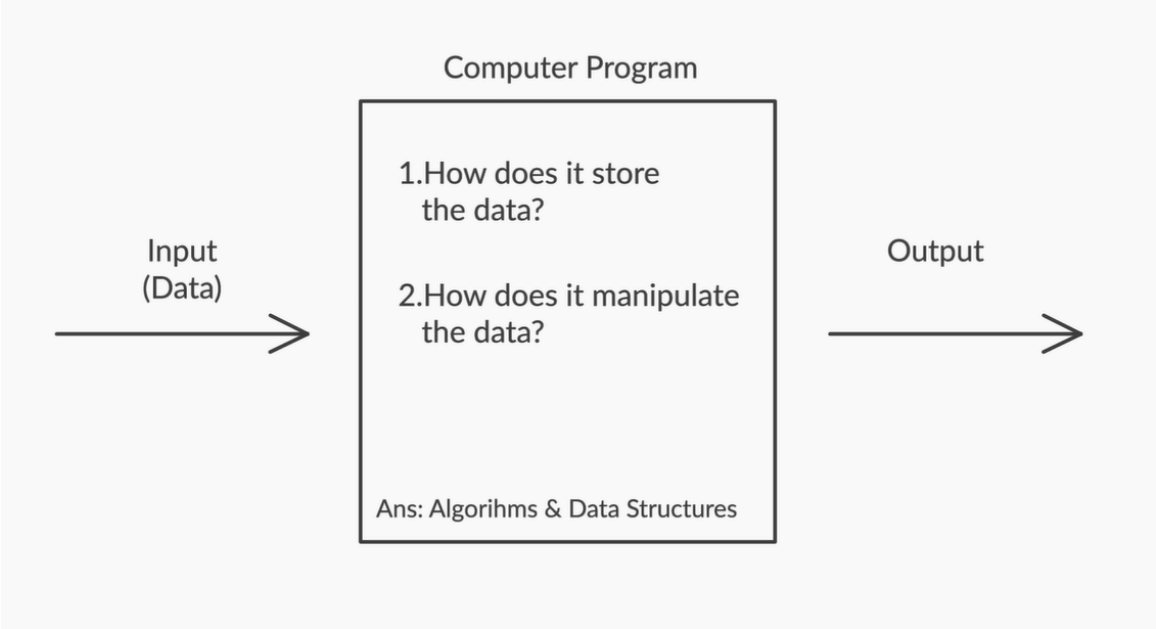
Lecture Notes

Algorithm Analysis

Session 1: Algorithms

Introduction to algorithms

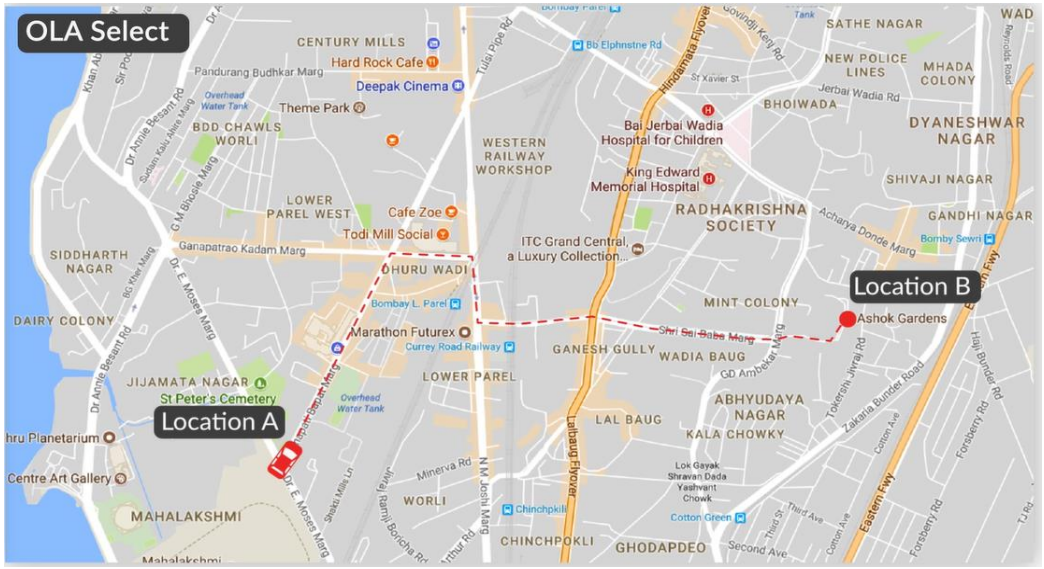
In this session, you have seen the importance of algorithms in a computer program. Specifically,
Computer Program = Algorithms + Data structures



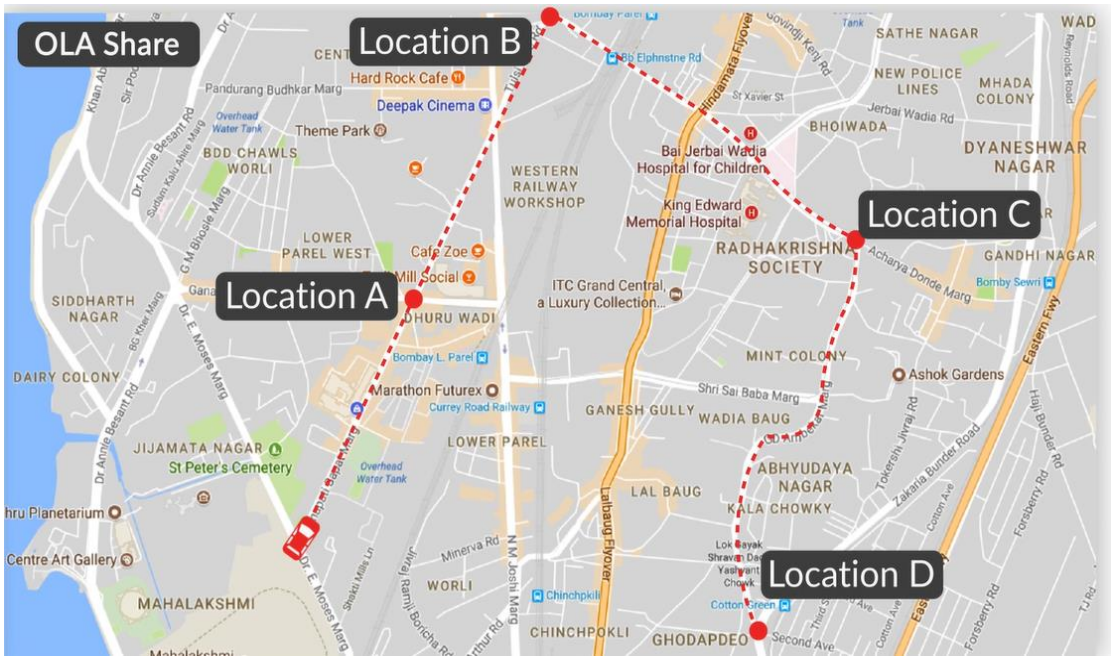
What is an algorithm?

An algorithm is a method of solving a problem through a set of sequential instructions.

Example 1: Ola app, on your request to pick from location A and drop at location B, an algorithm running behind the screen helps to locate a cab nearby and find the shortest route to destination.



Example 2: In a similar way, Ola share has another algorithm which helps to identify the multiple requests from different people en route to the destination and find out the shortest distance to pick and drop in their respective locations.



Algorithm 1

We discussed a practical scenario where students can register for a course both online mode & offline at academic office and certain students registered for the same course in both the modes.

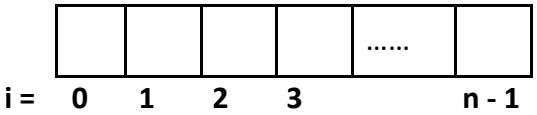
You have learnt an algorithm to find out the student ID's registered twice for the same course.

Algorithm 1

```
public void findDuplicates(int[] id) {
    System.out.println("Duplicate student id : ");
    for (int i = 0; i < id.length; i++) {
        for (int j = i+1; j < id.length; j++) {
            if (id[i] == id[j]) {
                System.out.print(id[i] + " ");
                break;
            }
        }
    }
}
```

We have assumed that the university consists of not more than 10000 students and student ID's are a set of integers between 1 – 10000.

The combined data of student ids are stored in an array variable **id [ ]**

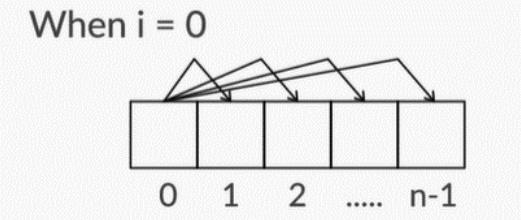


id [i] : i^{th} student id registered

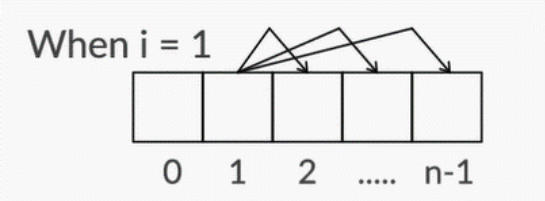
n : length of the array

The nested for loops in algorithm 1 helps to iterate across the data and compare each student id with other already registered ids to find duplicate student IDs.

For the first iteration of outer loop(i) i.e., when i = 0, the inner loop(j) iterates from 1 (i+1) to n - 1 and compares the first student id (id [0]) with other student IDs on right-hand side till it finds the duplicate student ID in id [0].



For the second iteration of outer loop(i) i.e., when i = 1, the inner loop(j) iterates from 2 (i + 1) to n – 1 and compares the second student id (id [1]) with other student IDs on the right-hand side till it finds the duplicate student ID of id [1].



For each iteration of inner loop(j), instruction set will compare id[i] & id[j], if it is true then prints the duplicate ID and breaks out of inner loop(j).

In this way the outer loop(i) is iterated from 0 to n – 1 and for each iteration of outer loop(i), inner loop(j) instruction set is executed for n – (i + 1) and prints all the duplicate IDs found.

Algorithm 2

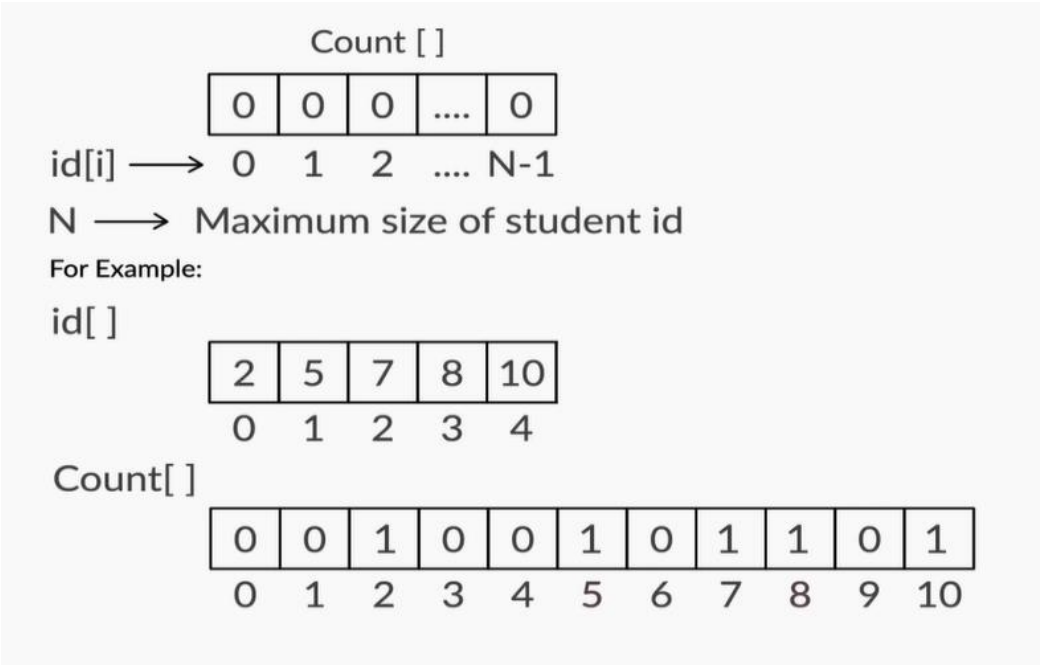
As you know, typically, there is more than one way to solve the same problem and you have seen another approach in finding the duplicate student IDs from given data.

Algorithm 2

```
public void findDuplicates(int[] id) {
        System.out.println("Duplicate data : ");
        int count[] = new int[10000];
        for (int i = 0; i < id.length; i++) {
            count[id[i]]++;
            if (count[id[i]] == 2)
                System.out.print(id[i] + " ");
        }
        System.out.println();
}
```

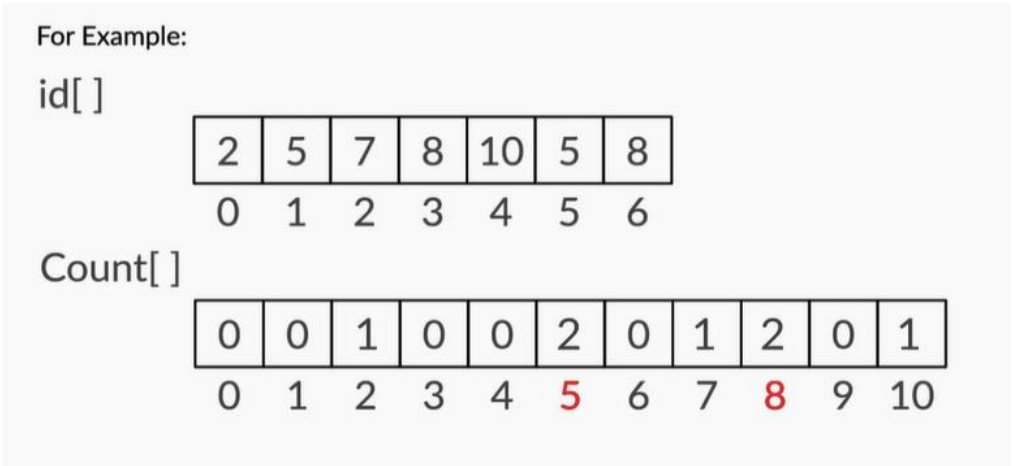
In algorithm 2, declare an extra array variable count [ ] with the maximum size of student id i.e. 10000 besides an array id [ ] for student registrations data.

Count array index values refers to corresponding student IDs. For id[i] (i^{th} student), count[id[i]] increments to 1, if same id appears again then the same cell is incremented to 2.



As you can observe, there are no duplicates in the above given data and so the corresponding cells of student ID in the count array are incremented to 1.

Whereas if there is a duplicate student ID in the given data,



Here, Student ID – 5 & 8 are repeated in the given data, so you can see that the arrays cells corresponding to the duplicate student ID (5 & 8) are incremented to 2.

When count [id[i]] is equal to 2 then prints the duplicate student ID.

Parameters

Now, you need to analyse the above two algorithms and find the efficient algorithm. For which we discussed that the parameters based on which an algorithm is analysed are

- How long does an algorithm takes to process output?
- How much memory space is required to execute an algorithm?

Both execution time and memory space are calculated as a function of input size(n).

In general, to analyse an algorithm you need to calculate the no of times certain instruction set is executed rather than the exact time values as it depends on various external factors such as processor speed, the compiler etc. Also, while analysing an algorithm, we consider worst case possible.

To understand worst case, you saw an analogy of unlocking a lock when you have ten different keys, of which you are not aware of the right key. When you try to unlock on a trial and error method, the worst case is that the lock is unlocked on your tenth attempt and the best case is that it unlocked on your first attempt.

Time complexity

The worst case to be considered in algorithm 1 is when there is no duplicate student IDs and for every i^{th} iteration of outer loop(i), inner loop instruction set is executed n – (i + 1) times.

Therefore, in total the no of times inner loop instruction set is executed = $(n-1) + (n-2) + \ldots\ldots + 1$

$= \frac{n(n-1)}{2}$

In a similar way, algorithm 2 executes certain instruction set to find the duplicate student IDs n times

On assuming constant times:

1. C_1 for rest of the instruction set like declaring variables, passing data to functions, etc.,
2. C_2 for the instruction set inside the loop to find duplicate student IDs.

Therefore, the total time taken,

Algorithm 1 – $T(n) = C_1 + \frac{n(n-1)}{2} * C_2$

Algorithm 2 – $T(n) = C_1 + n * C_2$

The total time taken to execute an algorithm as a function of input size(n) is called time complexity of an algorithm and is represented by T(n).

Space complexity

The other parameter considered to analyse an algorithm is memory space required to execute an algorithm.

The total memory space required to execute an algorithm as a function of input size(n) is called space complexity of an algorithm and is represented by S(n). In general, you only calculate the extra memory required, not including the memory needed to store the input.

As you have learnt, algorithm 1 uses a constant memory space besides an array variable id [ ] to store student IDs and algorithm 2 uses an extra array variable count [ ], the size of count array variable increases linearly with the increase in the student strength of the university i.e., when student ID exceeds 10000, then the maximum student ID is considered as the size of count array.

Therefore,

Algorithm 1 – S(n) - Constant space, memory space does not depend on the input size

Algorithm 2 – S(n) is linearly proportional to the number of possible students in the university

Asymptotic notations

After obtaining the complexity functions of both algorithms, you have learnt mathematical notations like Big O, Big omega (Ω), Big theta (Θ) called as asymptotic notations which help us to compare the functions of two different algorithms.

Big O

Big O indicates the upper bound (worst case) of the running time or space complexity of an algorithm.

To calculate the Big O of any function, we discussed certain simplification rules as,

- Drop the constant multiplier in a function as it depends on hardware like processor speed, on which the program is run.

  For example,

  i) $T(n) = 2n \Rightarrow T(n) \in O(n)$

  ii) $T(n) = 5n^2 \Rightarrow T(n) \in O(n^2)$

- Drop less significant terms in a polynomial function. Except for higher order terms, rest of the terms relatively contribute very less in explaining the growth of a function.
  For example,
  $T(n) = 10n^3 + n^2 + 4n + 800. \Rightarrow T(n) \in O(n^3)$

  If n = 1000, then T(n)=10,001,040,800.

  If you drop all the less significant terms except for $10n^3$, then the error rate is 0.01%, which is very minimal.

Definition: Big O is "bounded above by" (upper bound), there exists constants $c > 0$ and $N > 0$
$T(n) \leq c\, f(n)$ for all $n > N$, $T(n) \in O(f(n))$

If an algorithm takes the time complexity function as T(n) = $2n^2 - 3n + 6$

When calculating Big O, the time complexity function of an algorithm is **less than or equal** to upper bound and input size(n) is a positive value, as n value increases $-3n + 6 < 0$.

So, $T(n) \leq 2n^2$, $n > 2$ $\Rightarrow -3n + 6 < 0$
$\Rightarrow 3n > 6$
$\Rightarrow n > 2$

Therefore, $T(n) \leq 2n^2 \Rightarrow T(n) \in O(n^2)$

Big Omega

Big Omega indicates the lower bound of running time or space complexity of an algorithm. In the scenario of opening a lock, if you dealt with the best case, i.e. on your first attempt itself, you could unlock.

Definition: Big Omega (Ω) is "bounded below by" (lower bound), there exists constants $c > 0$ and $N > 0$
$T(n) \geq c\, f(n)$ for all $n > N$, $T(n) \in \Omega(f(n))$

If an algorithm takes the time complexity function as $T(n) = 2n^2 - 3n + 6$,

When calculating Big Omega, the time complexity function of an algorithm is **greater than or equal** to lower bound.

$T(n) = 2n^2 - 3n + 6$
$\geq 2n^2 - 3n$ (because 6 > 0)
$\geq n^2, n \geq 3 \Rightarrow 2n^2 - 3n \geq n^2$
$\Rightarrow 2n^2 - n^2 \geq 3n$
$\Rightarrow n^2 \geq 3n$
$\Rightarrow n \geq 3$

So, $T(n) \geq n^2$, for all $n \geq 3$

Therefore, $T(n) \in \Omega\,(n^2)$

Big Theta

Big Theta represents an in-between case, bounded both above and below the running time or space complexity of an algorithm.

Definition: Big Theta (Θ) is "bounded above and below", there exists constants $c_1 > 0$, $c_2 > 0$ and $N > 0$,
$c_1 . g(n) \leq T(n) \leq c_2 . f(n)$, for all $n > N$, $T(n) \in \theta(f(n))$

If an algorithm takes the time complexity function as $T(n) = 2n^2 - 3n + 6$,
As already discussed,

Upper Bound: $2n^2 - 3n + 6 \leq 2n^2$
Lower Bound: $n^2 \leq 2n^2 - 3n + 6$

So, $n^2 \leq 2n^2 - 3n + 6 \leq 2n^2, n \geq 3$ (N)

Among all the three asymptotic notations, the most used notation to compare algorithms is Big O. Therefore, you always tend to find the worst case of an algorithm with respect to the input size(n).

Rule of Sums and Rule of Products

You have learnt two rules which are used in general while analysing for time complexity of algorithms,

Rule of sums

In an algorithm, when two for loops are one after another as follows

```
for(int i = 0;i < n;i++){
    //instruction set
}
for(int j = 0;j < m;j++){
    //instruction set
}
```

Then the first for loop(i) instruction set is executed n times and second for loop(j) instruction set is executed m times.
So, in total n + m steps taken in executing this kind of an algorithm. If you consider the time complexity of this algorithm, $T(n) = n + m$ (neglecting the constant time taken by instruction set)
If $n > m \Rightarrow$ then you can say that $T(n) = n + m$

$$\leq n + n$$
$$\leq 2n$$

On dropping the constant multiplier 2, $T(n) \in O(n)$

Rule of products

In an algorithm, when two for loops are nested as follows

```
for(int i = 0;i < n;i++){
    for(int j = 0;j < m;j++){
        //instruction set
        }
}
```

Then the outer loop(i) is iterated n times and for each iteration of outer loop(i), inner loop(j) instruction set is executed for m times. In total, the instruction set inside the inner loop is executed n * m times, if you consider the time complexity function of algorithm, $T(n) = n * m$(neglecting the constant time taken by instruction set).

if $n > m \Rightarrow$ then you can say that $T(n) = n * m$

$$\leq n * n$$

$$\leq n^2$$

So, $T(n) \in O(n^2)$

Big-O and Growth Rates

You have learnt different functions encountered when analysing algorithms as follows,

Constant function	- $T(n) = C$ (C is a constant) $T(n) \in O(1)$
Linear function	- $T(n) = C_1 n + C_2$ (C_1 , C_2 are Constants) $T(n) \in O(n)$
Quadratic function	- $T(n) = C_1 n^2 + C_2 n + C_3$ (C_1 , C_2 , C_3 are Constants) $T(n) \in O(n^2)$
Cubic function	- $T(n) = C_1 n^3 + C_2 n^2 + C_3 n + C_4$ (C_1 , C_2 , C_3 are Constants) $T(n) \in O(n^3)$

To find out the relative efficiency among the different functions, the following table has been discussed in detail on how it helps you to give an idea on the different time taken for different input sizes.

To understand the below table, we have assumed that a computer performs 1 billion (10^9) operations per second.

Input size(n)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	Running time does not depend on input size(n)	3 ns	0.01 µs	0.03 µs	0.1 µs	1 µs	1 µs
20		4 ns	0.02 µs	0.09 µs	0.4 µs	8 µs	1 ms
30		5 ns	0.03 µs	0.15 µs	0.9 µs	27 µs	1 s
40		5 ns	0.04 µs	0.21 µs	1.6 µs	64 µs	18 min
50		5 ns	0.05 µs	0.28 µs	2.5 µs	125 µs	13 days
100		6 ns	0.1 µs	0.66 µs	10 µs	1 ms	4×10^3 years
10^3		9 ns	1 µs	9.96 µs	1 ms	1 s	32×10^{283} years
10^4		13 ns	10 µs	130 µs	100 ms	16.67 min	
10^5		16 ns	100µs	1.66 ms	10 s	11.57 days	
10^6		19 ns	1ms	19.92 ms	16.67 min	31.17 years	

Hence, the efficiency order of different functions

$$O(1) > O(log\ n) > O(n) > O(n\ log\ n) > O(n^2) > O\ (n^3)$$

Using this, you can compare different complexity functions of algorithms and find the efficient algorithm.

Duplicates

After finding the relative efficiency of different functions, you have learnt how to compare algorithms to find duplicate student IDs and find the efficient algorithm

Time complexity:

Algorithm 1 – T(n) = $C_1 + \frac{n(n-1)}{2} * C_2 \Rightarrow$ T(n) ∈ O(n^2) – Quadratic function

Algorithm 2 – T(n) = C_1 + n* C_2 ⇒ T(n) ∈ O(n) – Linear function

you can observe that algorithm 2: O(n) is more efficient when compared to algorithm 1: O(n^2)

Space complexity:

Algorithm 1 – S(n) ∈ O(1) – constant function

Algorithm 2 – S(n) ∈ O(n) – linear function

You can observe that algorithm 1: O(1) is more efficient when compared to algorithm 2: O(n)

Time vs Space Complexity Trade-off:

As you've seen with the algorithms for identifying duplicated student ids, the two algorithms have made different trade-offs in terms of time and space.

Specifically:

- Algorithm 1 runs slower, but uses less memory
- Algorithm 2 runs faster, but uses more memory

As a software developer, you'll often face this kind of dilemma while designing programs and software. Do you write programs that runs fast but uses lots of memory space? Or do you write program that runs slower but uses less memory space?

The answer is it depends.

For example, if you are writing software to do high-frequency stock trading where every microsecond can be the difference between earning or losing hundreds of thousands of dollars, you will likely want to design programs can execute very quickly at the expense of using lots of memory space.

On the other hand, you may be writing software that runs on smartphones where the memory available to the software is limited. In this situation, you may want to write software that uses less memory but runs a bit more slowly.

Therefore, use your best judgement when it comes to Time vs Space Complexity trade-off. Identify your business needs or constraints, and then decide if you should trade space for time or vice versa.

Session 2: Run-time Analysis

Fibonacci sequence

In this session, you have been introduced to a mathematical function called Fibonacci sequence, which is defined as:

F(0) = 0

F(1) = 1

F(n) = F(n - 1) + F(n - 2)

We made a small change in the above function as F(n) = [F(n - 1) + F(n - 2)]%10, i.e.by dividing the function with modulo 10 in order to avoid integer overflow error for higher input(n) values. We then discussed on calculating n^{th} number of function i.e., n = 0, 1, 2, 3, 4, 5 …

F(0) = 0
F(1) = 1
F(n) = [F(n-1) + F(n-2)]%10

If n = 0, then F(0) = 0

If n = 1, then F(1) = 1

If n = 2, then F(2) = [F(2 - 1) + F(2 - 2)]%10
= [F(1) + F(0)]%10
= [0 +1]%10 = 1

If n = 3, then F(3) = [F(2) + F(1)]%10
= [1 + 1]%10
= 2

If n = 4, then F(4) = [F(3) + F(2)]%10
= [2 + 1]%10
= 3

If n = 5, then F(5) = [F(4) + F(3)]%10
= [3 + 2]% 10
= 5

If n = 6, then F(6) = [F(5) + F(4)]%10
= [5 + 3]%10
= 8

If n = 7, then F(7) = [F(6) + F(5)]%10
= [8 + 5]%10
= 13%10
= 3

You have learnt algorithm 1 to generate n^{th} number of the function F(n) = [F(n - 1) + F(n - 2)]%10.

Algorithm 1

```
public int fibonacci(int n){
    if (n < 2)
        return n;
    else
        return (fibonacci(n - 1) + fibonacci(n - 2))%10;
}
```

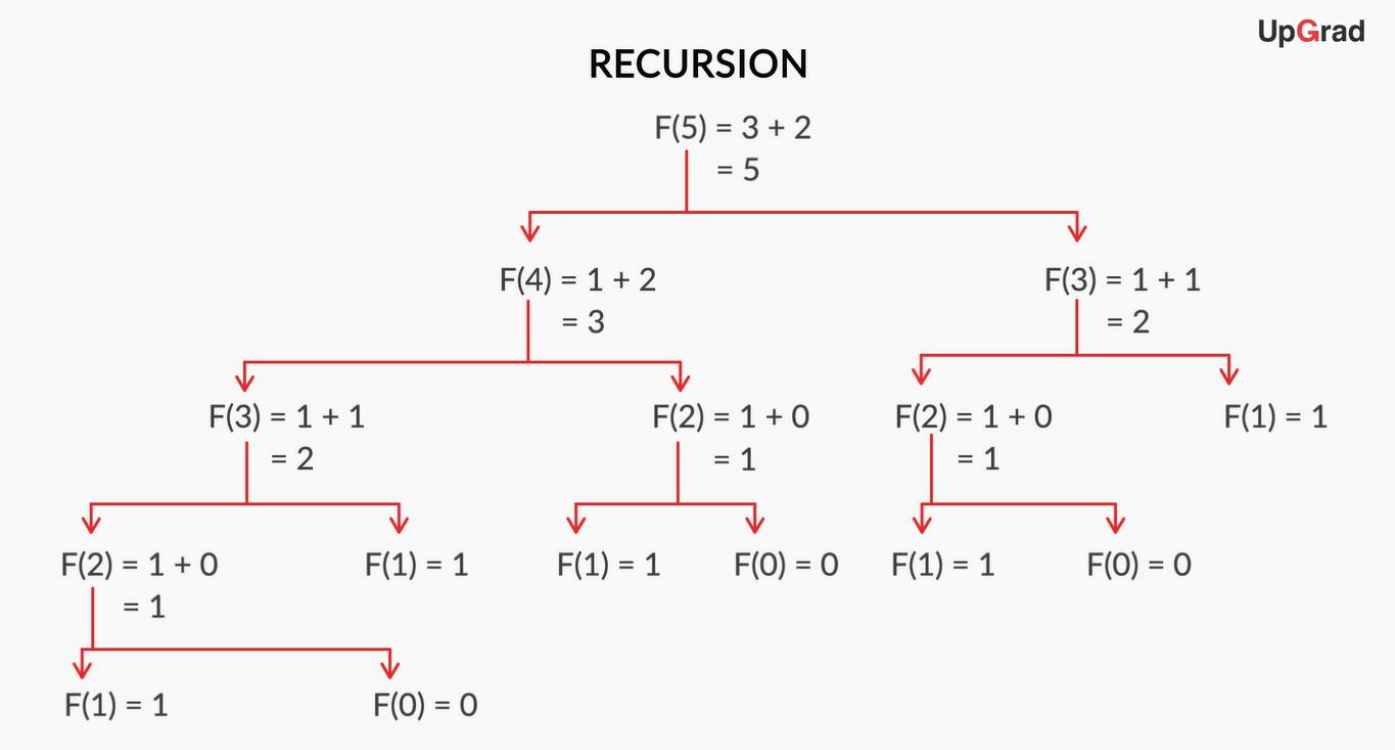
As you have learnt, the function fibonacci() is called inside the same function which is called as recursive function

Recursive function

Recursive function is a function which calls itself during its execution, and a recursive function typically needs to define two cases:

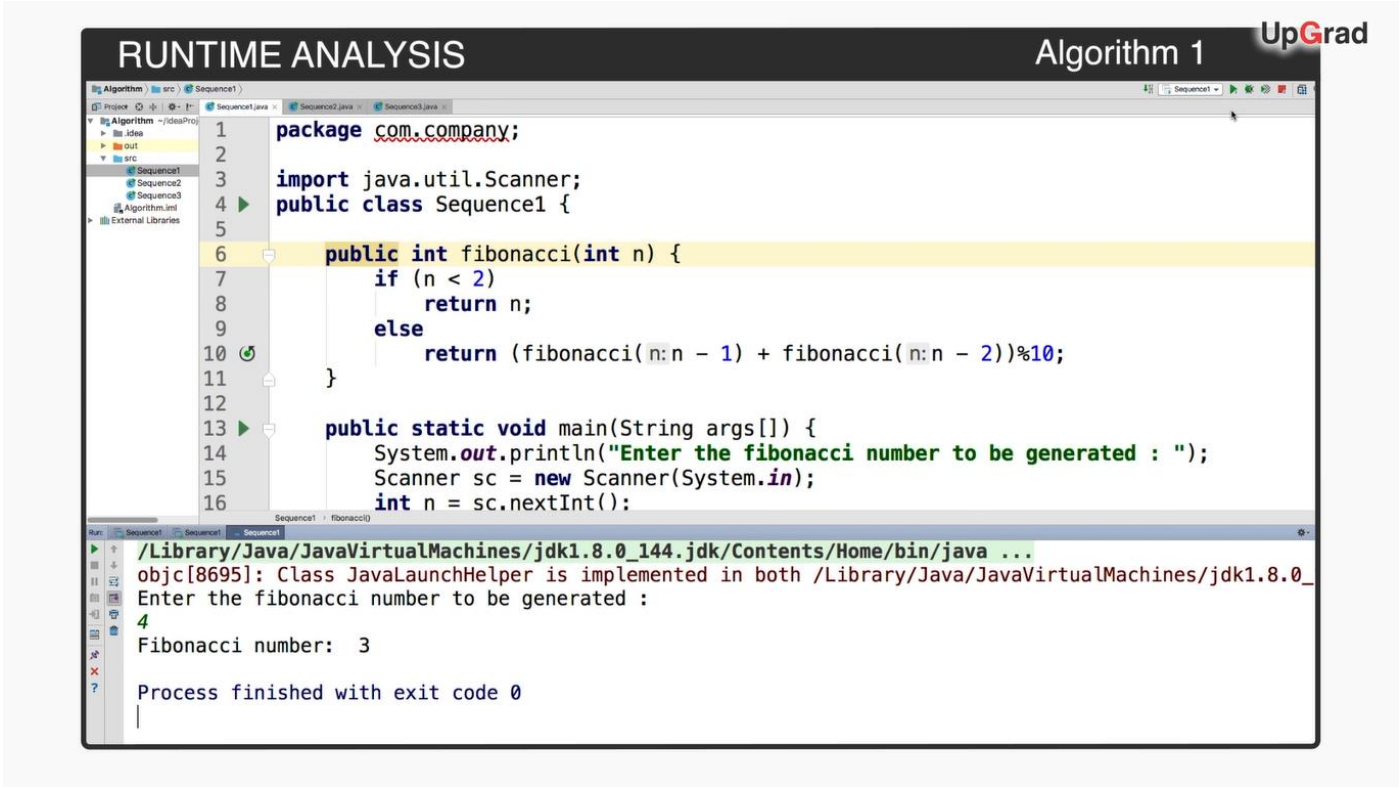
1. the base case that returns a definitive value and
2. the recursive case where the recursive function calls itself and tries to solve smaller parts of the problem at hand

In algorithm 1, the if condition acts as a terminating or base condition which returns the definite values to end the recursive calls of the function fibonacci() and else condition acts as a recursive case, which calls the same function fibonacci() again till the passing argument satisfies the base condition. Now to understand how exactly the recursion function generates the Fibonacci number for a given input(n), we discussed a recursion tree to generate 5th Fibonacci number.

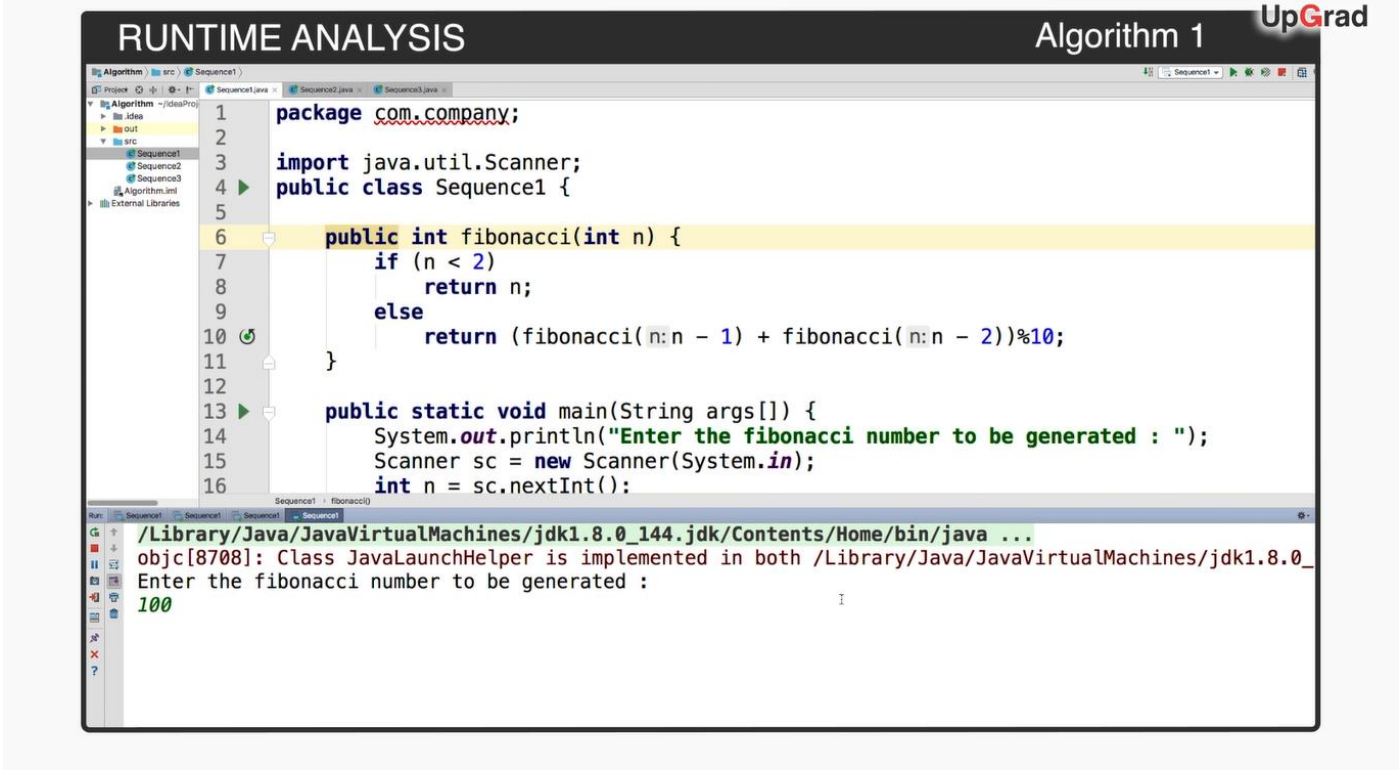


In which F(0) & F(1) are the terminating conditions and helps to find rest of the values and print final output.

Then, we have demonstrated the code of algorithm 1 for different input size like n = 4, n =42



But when the input given is n = 100, then the compiler took time to process and you never saw the output. As suggested, If you have run this code in your system for the input n = 100 even after waiting for hours the compiler must not have given any output and still processing.

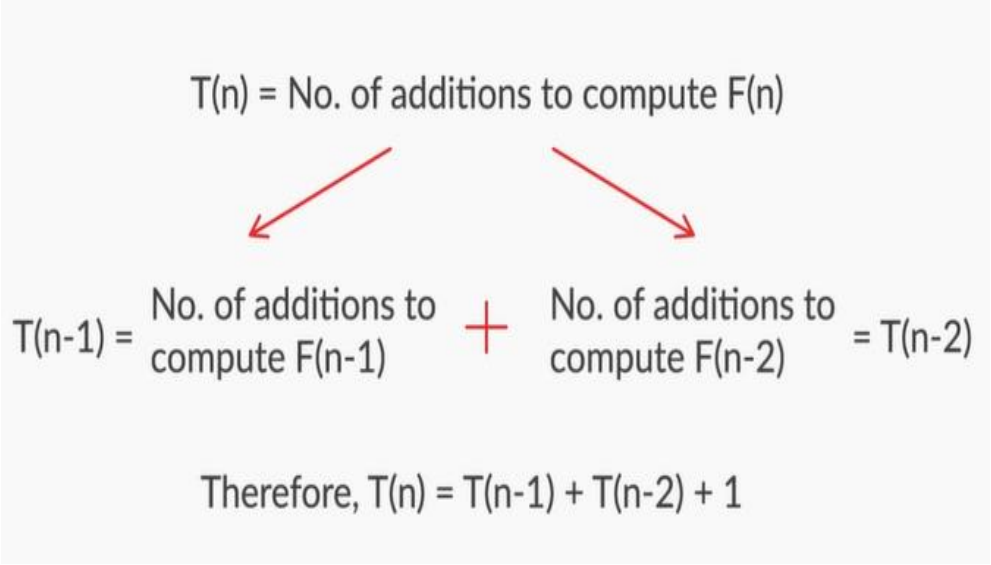


Time complexity of algorithm 1

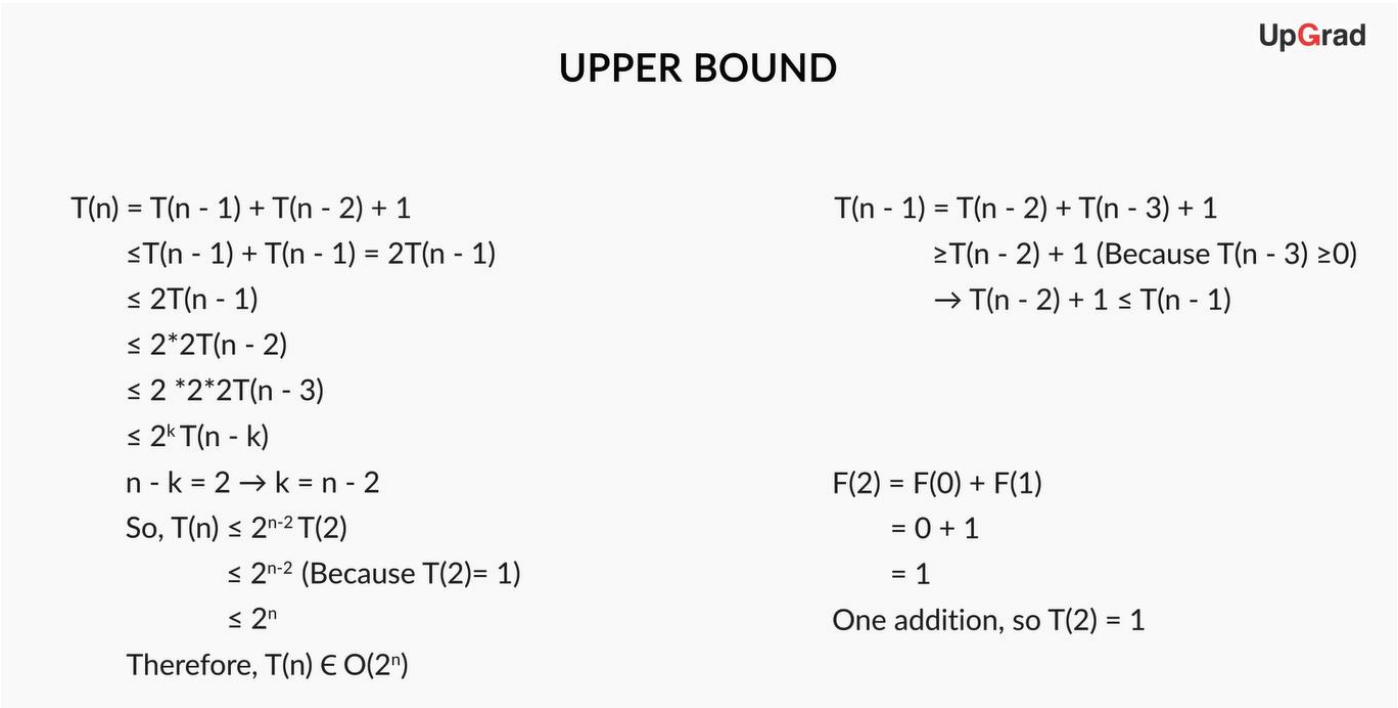
We considered that,

T(n) = No of additions required to compute F(n)

No of additions required to calculate F(0)& F(1) = 0, as the if condition returns the same value without any calculations(additions) required.



Then, we have calculated the upper bound for the time complexity function as follows,

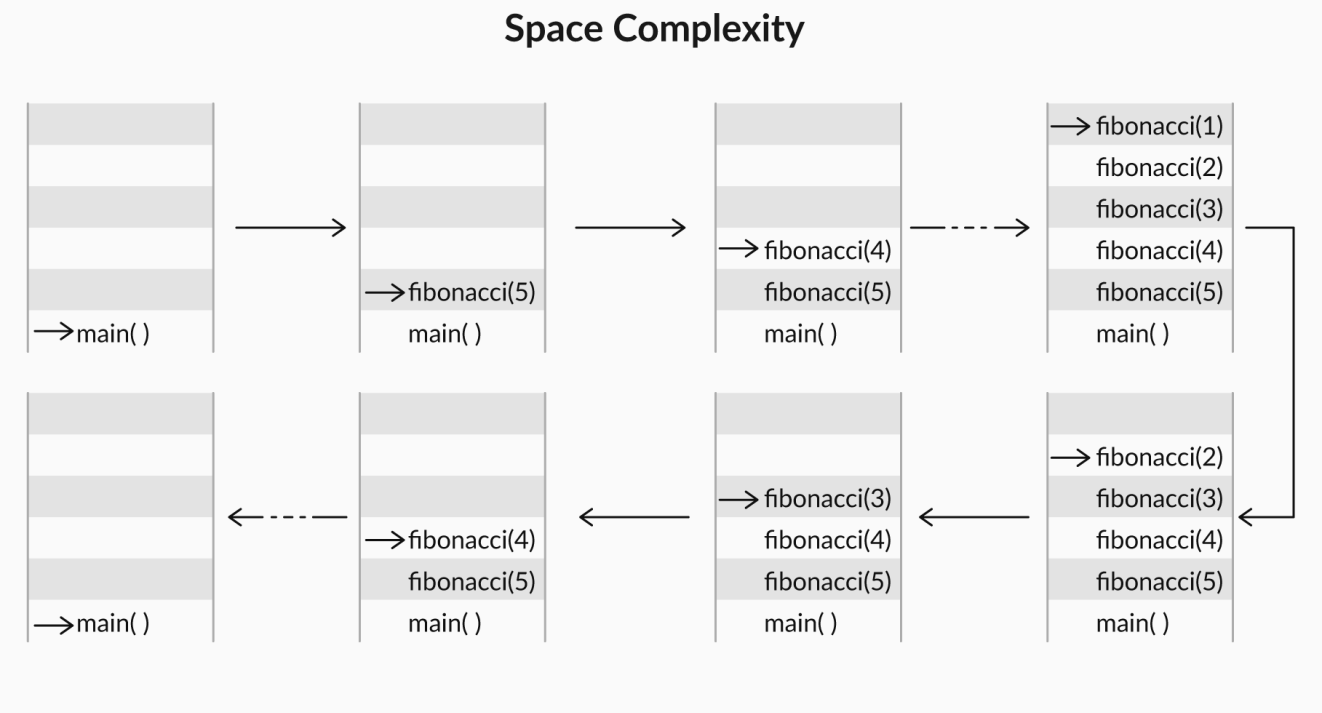


So, the time complexity of algorithm 1 is an exponential function with O(2^n), which is really slow, and this is why algorithm 1 is unable to produce an output when n = 100.

Space complexity of algorithm 1

With respect to the memory space i.e., space complexity required for algorithm 1, we have again discussed with an example of generating 5th Fibonacci number and how the memory space is occupied on each recursive call?

Think of memory space as partitions as shown and each partition is occupied as the function is called and the same function pops out of memory space when the process is done and returns the value.



As you can observe, the maximum memory space required is proportional to the Fibonacci number generated. So, space complexity S(n) ∈ O(n) – linear function

Recursion

We have discussed a real-world problem “file search” in a laptop using recursion,

The file search process as follows,

Consider each folder is a directory and it consists of many sub directories(folders), in order to search a specific filename, pass in the filename that you are looking for and the file directory path(folder path) that you want to start the search to the recursive function.

Then follow certain steps inside the recursive function as

1. List all the contents inside the file directory passed to the function
2. Loop through the contents inside the file directory,
   - If a specific content is another directory, recursively call the function and pass in the file directory and its path.

- If the content is a file, check and see if the file matches with the filename you are searching for. If it matches, then return “file is found” with the file path.

This has been demonstrated using java code as follows,

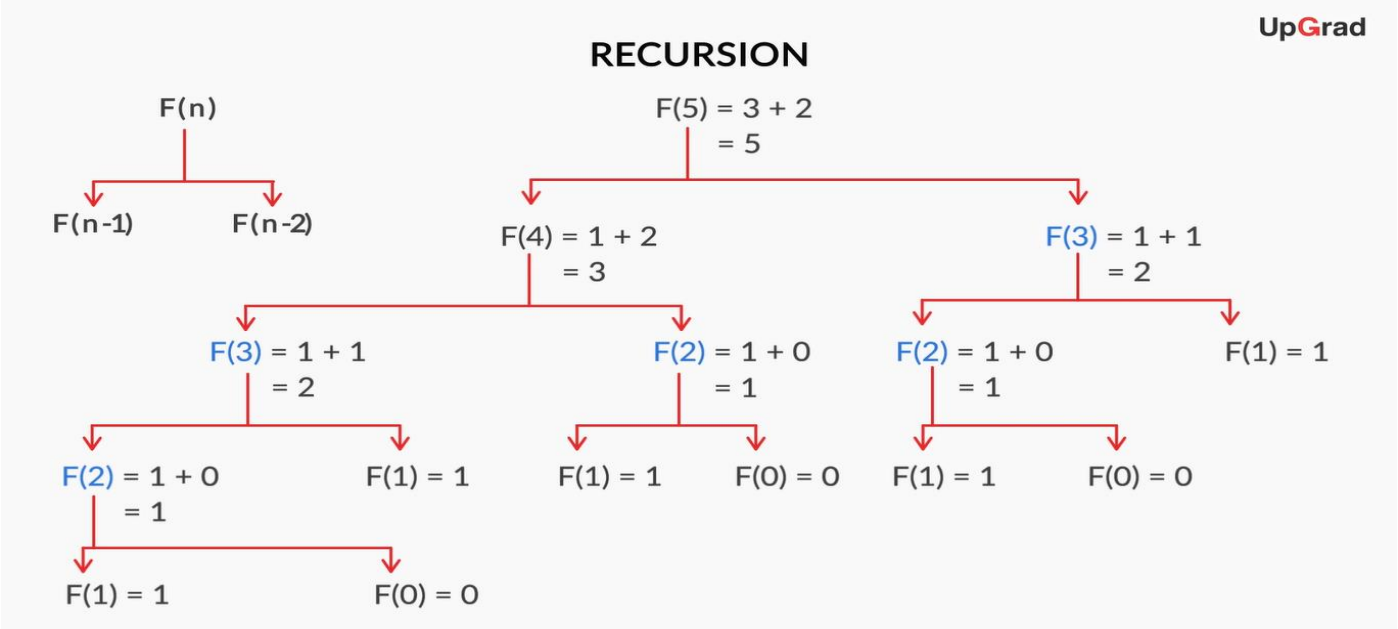
RUNTIME ANALYSIS File Finder UpGrad

```
 * @param fileName name of the file to be searched
 * @param inputPath path of the directory in which the file is to be searched
 */
public void findFile(String fileName, String inputPath)
{
    File directory = new File(inputPath);
    if (directory.isDirectory()){
        File[] list = directory.listFiles();
        for (File file : list){
            if (file.isDirectory()){
                findFile(fileName, file.getAbsolutePath());
            }else if (fileName.equals(file.getName())){
                System.out.println(inputPath + "/" + fileName + " found");
            }
        }
    }
}

public static void main(String args[]) {
    String projectPath =  System.getProperty("user.dir");
    FileFinder fileFinder = new FileFinder();
    fileFinder.findFile( fileName: "Algorithms.iml", projectPath);
}
}
```

Algorithm 2

Now, coming back the function F(n) = [F(n - 1) + F(n - 2)]%10,

In algorithm 1, we have not stored the values F(2) & F(3) and had to calculate each time whenever required



We try and overcome the redundant calculations in algorithm 2 by storing all the values calculated in an array variable f [ ]. Therefore, if we need to recall a Fibonacci value that has been previously calculated, we can simply refer back to the values stored in f[ ] rather than recalculating the value again.

Algorithm 2

```
public void fibonacci(int n){
    int[] f = new int[n];
    f[0] = 0;
    f[1] = 1;
    for(int i = 2;i <= n;i++)
        f[i] = (f[i - 1] + f[i - 2])%10;
    System.out.println("Fibonacci number : "+f[n]);
}
```

As you can observe, there is only one for loop iterating from 2 to n i.e., n – 1 times executes certain instructions to generate n^{th} number of the function F(n) = [F(n - 1) + F(n - 2)]%10

Therefore,

Time Complexity:

T(n) = No. of additions to compute F(n)

So, T(n) = n – 1

Therefore, T(n) ∈ O(n), linear time

With respect to the memory space, besides declaring variable n for input, you need to create an array variable f[ ] of size n in order to store all the calculated values of the function F(n) = [F(n - 1) + F(n - 2)]%10

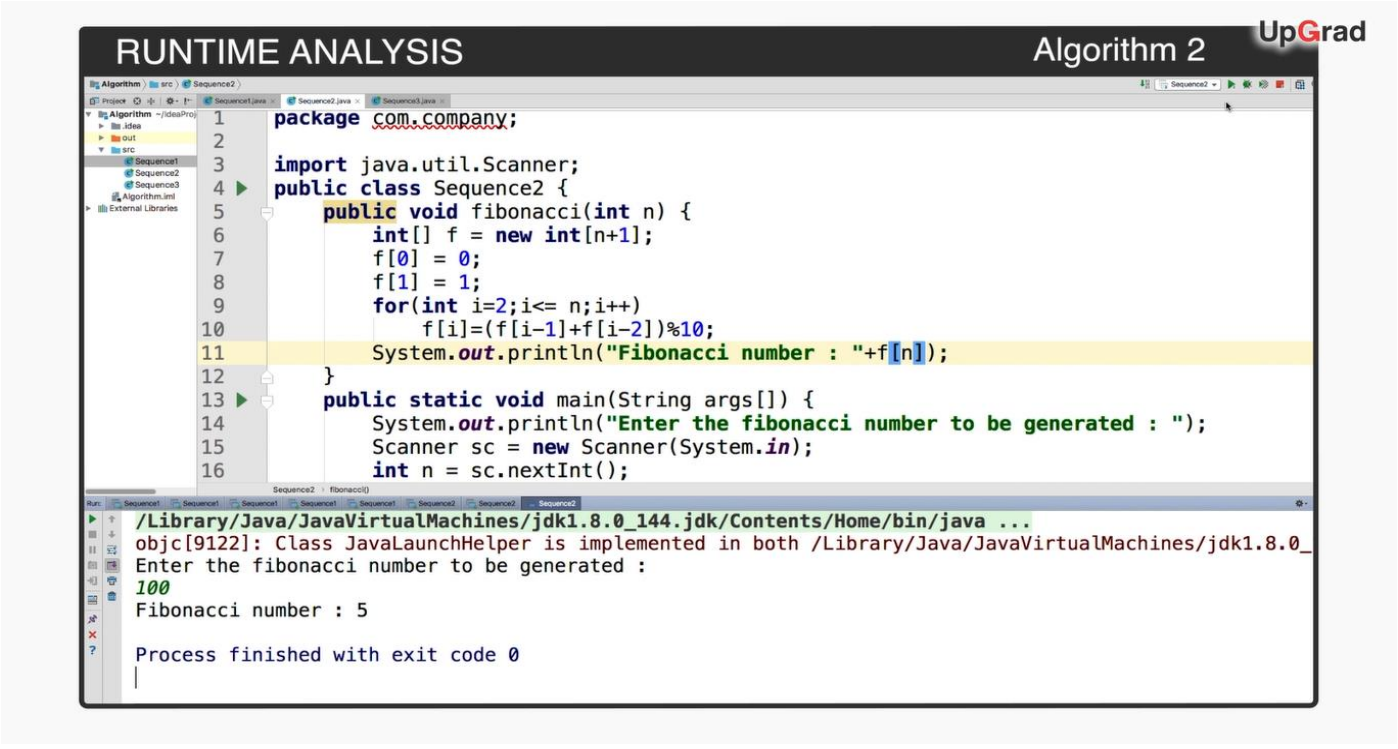
Therefore,

Space Complexity:

An extra array variable f[] is defined, whose size is dependent on input variable n

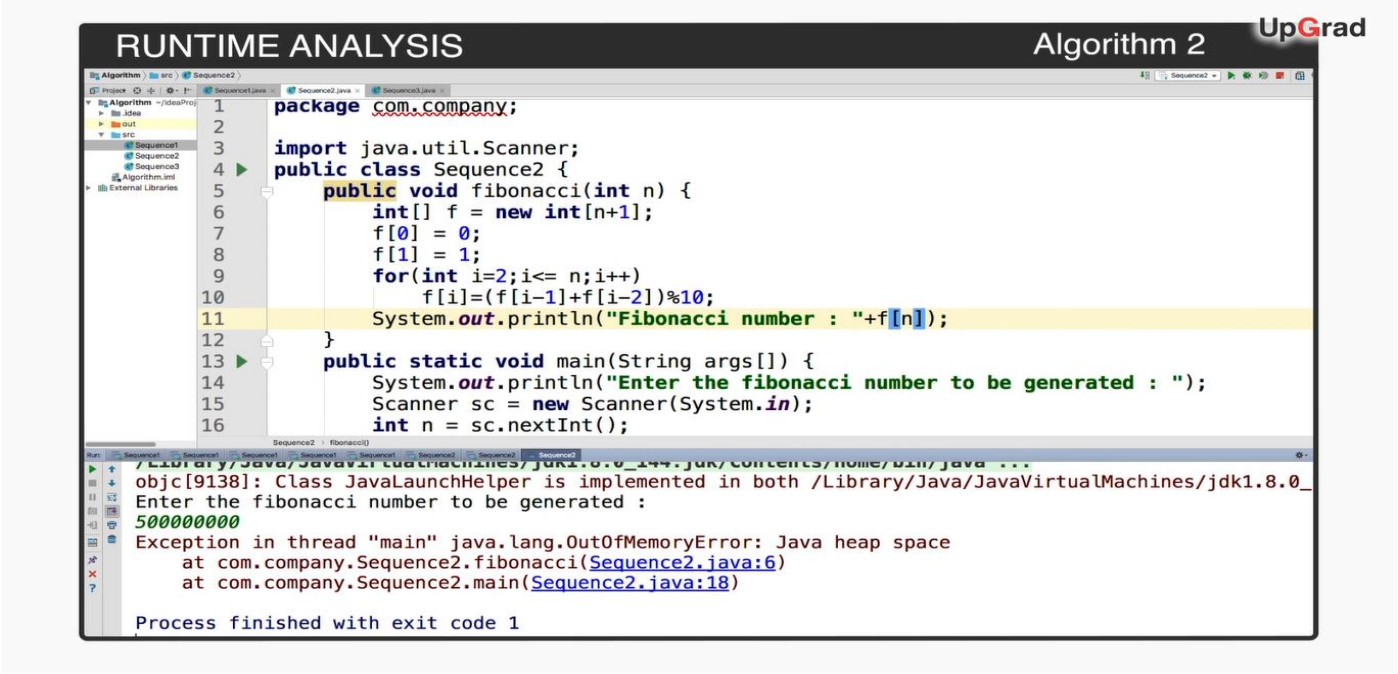
So, S(n) ∈ O(n), linear in memory space

After analysing algorithm 2 with respect to time taken and memory space required, we have run the java code for input values like n = 4, n = 100 and using algorithm 2 we are able to generate the output for n = 100.



So, algorithm 2 overcomes the constraint of algorithm 1 and process the n^{th} number of the function when n =100.

But, for an input n = 5 x 10^8, the compiler gives an error as follows,



The compiler displays an out of memory error as the memory needed to create an array of size 5 x 10^8 is much larger than the total memory available to our entire program. Therefore, algorithm 2 cannot process the output when $n = 5 \times 10^8$.

If you compare algorithm 2 with algorithm 1,
Algorithm 1 – T(n) ∈ $O(2^n)$ – Exponential time
S(n) ∈ $O(n)$ – Linear space

Algorithm 2 – T(n) ∈ $O(n)$ – Linear time
S(n) ∈ $O(n)$ – Linear space

With respect to the execution time, algorithm 2 with O(n) is better than algorithm 1 with $O(2^n)$ and so is able to process the Fibonacci number for n = 100 in no time.

Whereas algorithm 2 must be improved with respect to the memory space required. Otherwise it can't store and process Fibonacci numbers for large input values such as $n = 5 \times 10^8$.

Algorithm 3

To overcome the memory space constraint in algorithm 2, you have learnt algorithm 3, a clever technique that calculates the Fibonacci sequence by using three different variables a, b & c.

Algorithm 3

```
public int fibonacci(int n){
    int a = 0, b = 1, c = n;
    for (int i = 2;i <= n;i++){
        c = (a + b)%10;
        a = b;
        b = c;
    }
    return c;
}
```

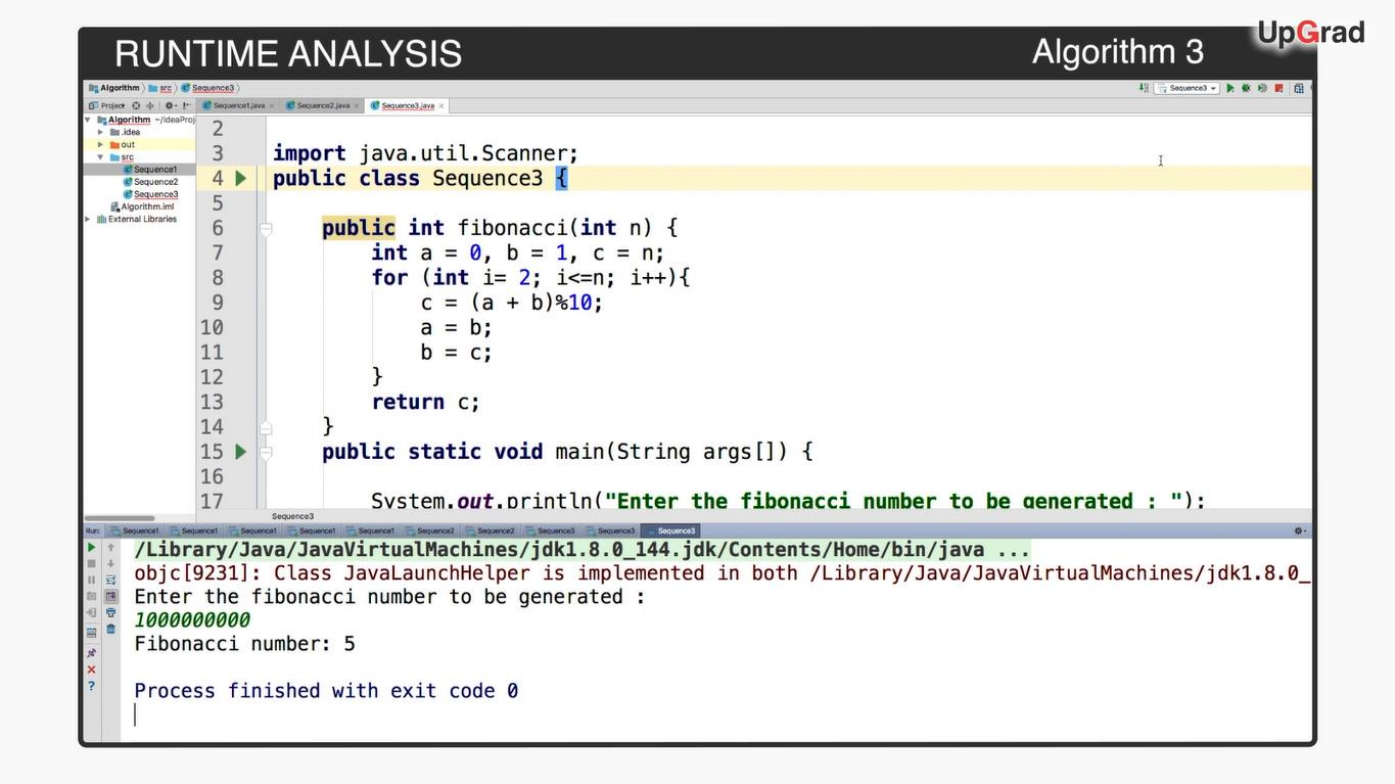
The variables are initialized as
a = 0, b = 1, c = n

and then for each iteration of for loop(i), variables are assigned as follows
c = (a + b)%10;
a = b;
b = c;

The values are listed below for first two iterations and for n^{th} iteration, variable c stores n^{th} number of the function F(n) = [F(n - 1) + F(n - 2)]%10

Variable \ i	2	3	.....	n
a	1	1		F(n-2)
b	1	2		F(n-1)
c	1	2		F(n)

The java code of algorithm 3 is executed for different values of n and in specific when n = 10^9, output is processed and displayed as follows,



So, with this we are able to confirm that algorithm 3 overcomes the memory space constraint of algorithm 2. More specifically, unlike algorithm 2, the memory required for algorithm 3 is constant and independent of the input n. On analysing algorithm 3 with respect to time taken and memory space required,

The instruction set which help in generating n^{th} number of the function is executed n – 1 times,
So, the time complexity of algorithm 3, T(n) = n – 1 ⇒ T(n) ∈ O(n) – linear time

Memory space required is constant, as only three variables are used to process the output irrespective of the input size(n), so the space complexity of algorithm 3, S(n) ∈ O(1) – constant space

Summary

Algorithm 1 – $T(n) \in O(2^n)$ – Exponential time
$S(n) \in O(n)$ – Linear space

Algorithm 2 – $T(n) \in O(n)$ – Linear time
$S(n) \in O(n)$ – Linear space

Algorithm 3 – $T(n) \in O(n)$ – Linear time
$S(n) \in O(1)$ – Constant space

The runtime and space complexity of algorithm 3 is O(n) and O(1) respectively, which is much more efficient than the other two algorithms. Therefore, algorithm 3 can process the n^{th} number in the Fibonacci sequence for large values of n = 10^9, a value that would otherwise be too large for algorithm 1 and algorithm 2.

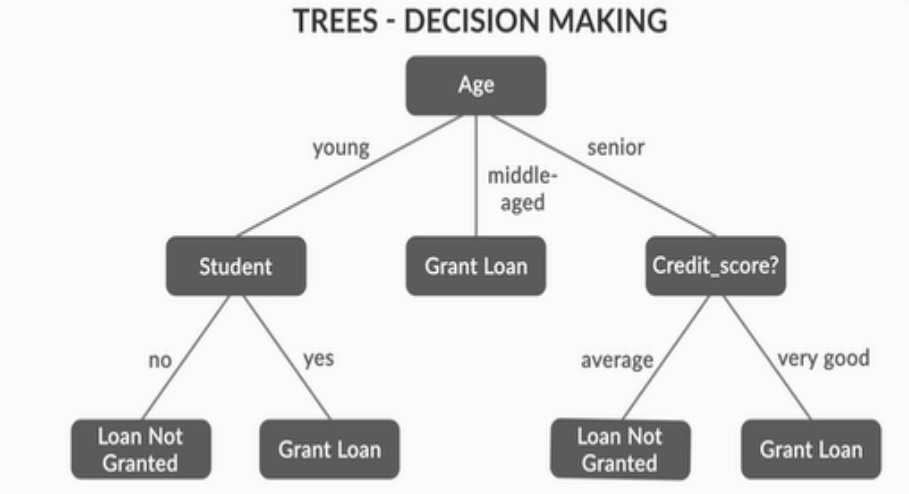
Lecture Notes

Binary Trees and BSTs

So, in this module you learnt about a non- linear data structure called Trees. The first session focused on a special variant of trees called binary trees. The second session dealt majorly with a special variant of Binary Trees called Binary Search Trees.

Introduction to Trees

In this segment you learnt about the structure of trees which can store non-linear data with various data points having specific relation with each other. You learnt that trees naturally convey the information concealed within a data set. All this was discussed with the help of few examples. One of the examples that was discussed was a decision-making tree representing the criteria used to grant loan to a bank customer.



Components of a Tree

Here you learnt the following

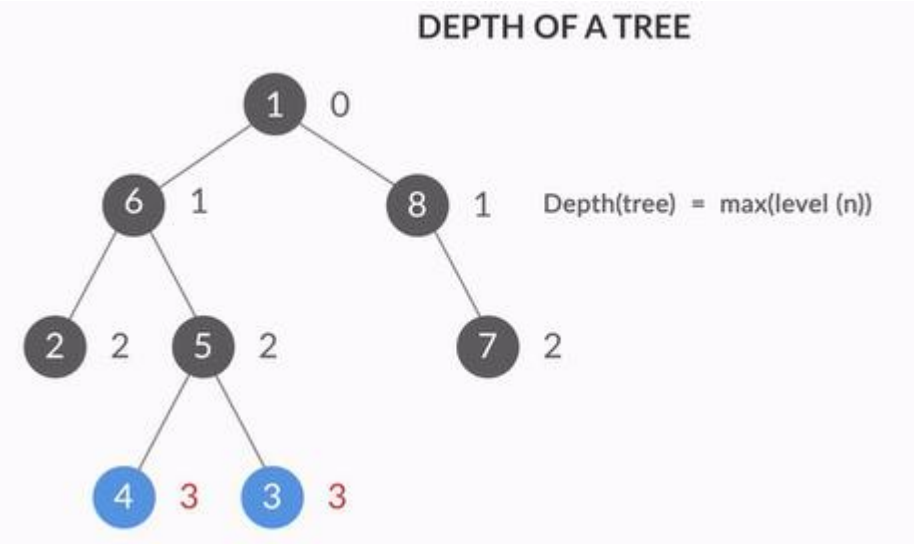
- Nodes of a Tree - Each data point in the tree with some value is the node of a tree
- Root Node – A node which does not have any parent is called the root node of the tree
- Leaf Node – A node without any child/children is the leaf node
- Internal Node – Any node which is not the leaf node is internal node
- Edges – The connection between 2 nodes. An edge connects one node to another node
- Definition of a tree –
    - It should have only one root node
    - A node should have exactly one parent
    - No cycles are allowed in a tree

Binary Trees

In this segment, you learnt about a special variant of trees called Binary Trees. You learnt that a tree with a degree 2 is called a Binary Tree. Degree is defined as the maximum number of child nodes a node can have. So, a tree where a node can have maximum of 2 child nodes is called a binary tree. We also discussed that unlike lists, trees do not have any natural ordering within its nodes. So we had to define certain traversal techniques based on which we could allot some ordering within the nodes of a tree

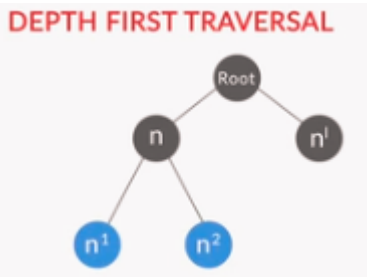
Depth First Traversal

Next, we moved onto a traversal technique called Depth First Traversal. Before learning what DFS is, we defined Depth of a Tree as the maximum level a node can have.

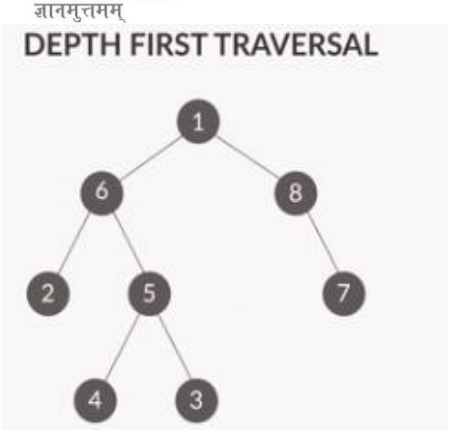


In the image above, nodes 3 and 4 have the maximum level in the tree which is 3. So the depth of this three is 3.

The basic ideology of Depth first traversal (DFS) was explained with the help of the following image



It was discussed that if the node n is visited and its sibling node n' and its child nodes n^1 and n^2 are not visited, then before visiting n' n^1 and n^2 are visited. We move along the depth first and then the breadth is traversed.
Following example was used in the video to explain the same



The DFS for the tree comes out to be 1,6,2,5,4,3,8.

Following was the pseudocode that was discussed for DFS

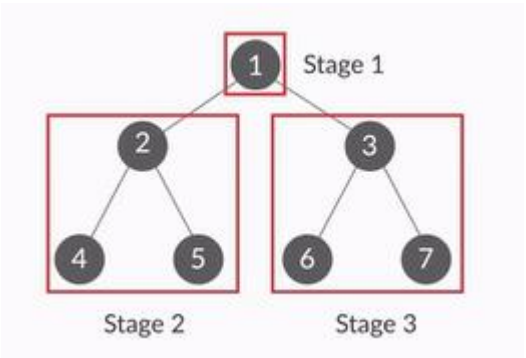
```
dfs (tree)
dfs_node(start(tree))
dfs_node(n)
  if (left (n) exists)
    dfs (left (n))
  if (right (n) exists)
    dfs (right (n))
```

Variants of DFS

You learnt about 3 variants of DFS namely

- Post Order Traversal
- Pre Order Traversal
- In Order Traversal

We discussed that any visit to a node in a tree has 3 stages as depicted in the figure given below.



Stage 1: When none of the child nodes of a node is visited
Stage 2: When the Depth First Traversal of the left child is taken care of
Stage 3: When the Depth First Traversal of the right child is taken care of.

Depending upon at which of these stages the action is performed on a node, we defined the particular variant of DFS.

When the action is performed during the stage 1 we call it Pre Order Traversal

When the action is performed during the stage 2 we call it In Order Traversal

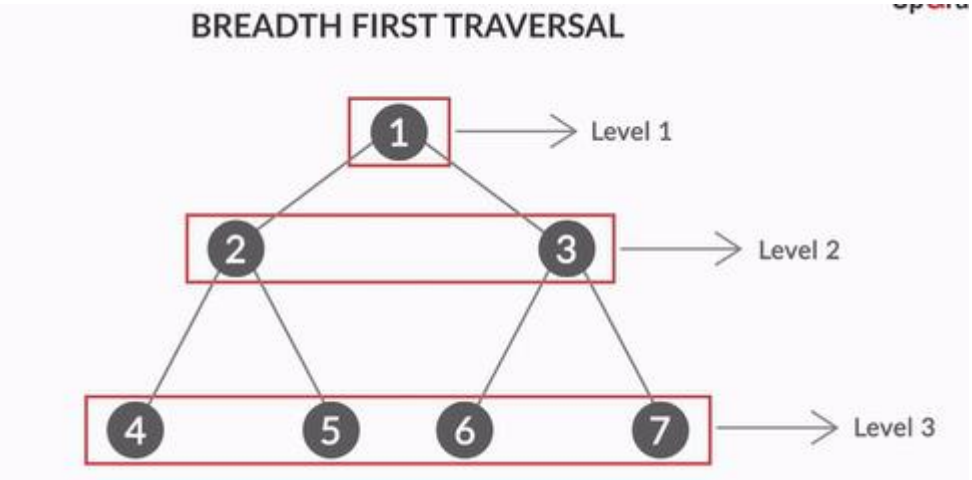
When the action is performed during the stage 3 i.e. once the right child has been taken care of we call it Post Order Traversal.

Following pseudocode was discussed for the same.

```
dfs (tree)
dfs_node(start(tree))

dfs_node(n)
  if (left (n) exists)          ---> Preorder
    dfs (left (n))              ---> Inorder
  if (right (n) exists)
    dfs (right (n))             ---> Postorder
```

Breadth First Traversal

The next traversal technique that was discussed was Breadth First Traversal (BFS). You learnt that in BFS, also known as Level Order Traversal, all the nodes of a level are visited first before we move onto the next level.



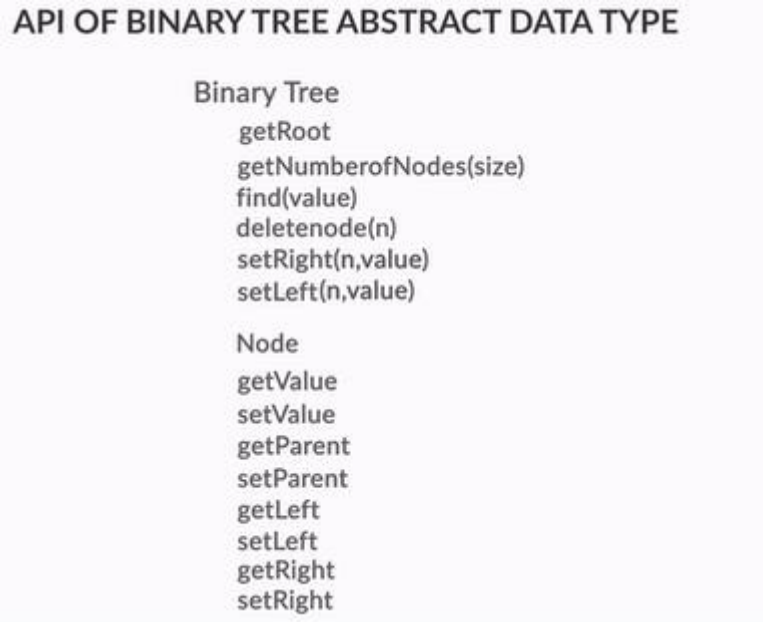
In the figure above, all the nodes of Level 1 are visited first. Then we move onto level 2 where all the nodes i.e node 2 and node 3 are visited before moving onto Level 3 where nodes 4,5,6 and 7 are visited.

The Java implementation of BFS was discussed using Queues. Following was the pseudocode discussed.

```
bfs(t)
 Q = new queue
  enqueue(Qroot(t))
  while(Q is not empty)
    n = dequeue(Q)
    action n
If(n has a left child)
 enqueue(left child)
If(n has a right child)
 enqueue(right child)
End while
end while
```

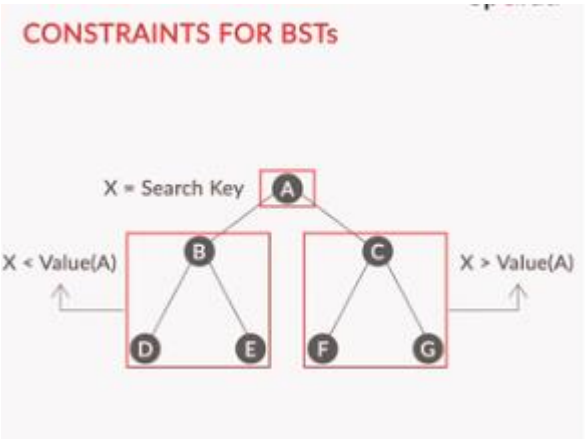
API Of Binary Abstract Data Type

To conclude the session, we discussed the API of Binary Abstract Data Type as can be seen in the image below.



Session 2 – Binary Search Trees

In this session, you learnt about a special variant of Binary Trees called Binary Search Trees or BSTs. The following constraints make Binary Tree a Binary Search Tree



- The values of all the nodes in the left subtree are less than the root node.

- The values of all the nodes in the right subtree are greater than the root node.
- Again, each subtree behaves like a binary search tree

Search in a BST

You learnt that searching in a BST is a $\log_2 N$ operation. This is because every time we move onto the next node, we conclusively ignore the remaining half of the tree from our search space. Following was discussed

- We compare the search key with the root.
- If the search key is greater than the root, we move onto the right subtree of the root. Else we move onto the left subtree of the root, thus ignoring one half of the tree.
- In case search key = root, we return the root itself.
- When we move onto the left or right subtree, we again compare the search key with the node we are at.
- Depending upon the search key is greater or less than the current node, we move onto the left or right of this node and so on.
- So at every step the remaining half of the tree is ignored thus giving it a logN efficiency.

Following pseudocode was discussed for its Java implementation.

```
find(n,v)
If value(n) = v then
 Return n
If n is a leaf node
 Return null
If v<value(n) then
 Return find(left(n),v)
Else
 Return find(Right(n),v)

find(tree,v)
find(root(tree),v)
```

Adding a node to a BST

Adding a node to a BST in principle is very similar to the search operation. You learnt that while adding a node to a BST, you first must figure out the position at which the node can be added. It was discussed that a new node can only be added as a leaf node. So, adding a node with a given value is like finding the node with this value in the binary tree. Since this node does not exists in the tree, you will end up at a leaf node where you can add this node. This gives a fair intuition as to why adding a node is also a logN operation.

Following pseudocode was discussed for tis Java implementation.

```
Add Node(parent, node)
    If parent=node then
        return
    If parent > node then
            If (left(parent)exists) then
                add Node(left(parent), node)
             else,   /* no left child */
                make node the left child of parent
    If parent < node then
            If (right(parent)exists) then
                add Node(right(parent), node)
            else
                make node the right child of parent
```

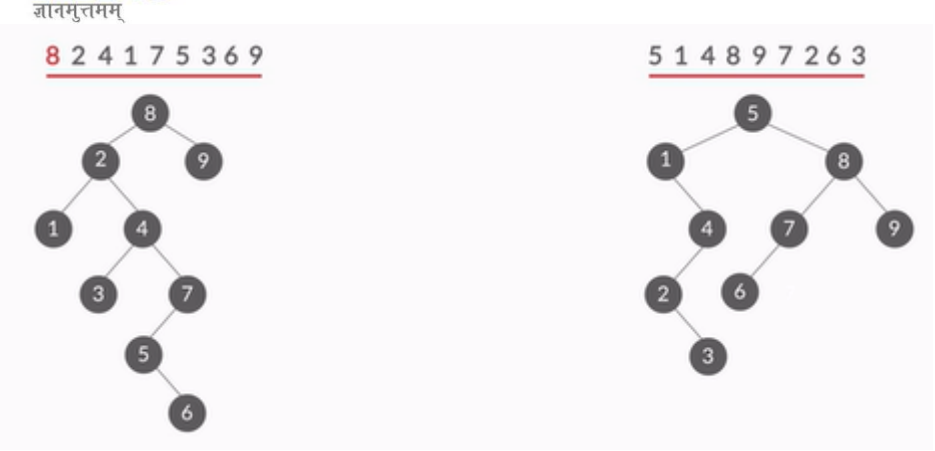
Deleting a Node

Next, you learnt about deleting a node from a BST. You learnt that the deletion of data from binary search trees can be carried out in three ways:

1. If the node to be deleted is a leaf node, it is directly removed from the tree.
2. If the node has one child, the node itself is deleted, and its child node is connected to its parent node.
3. If the node has two children, you find its successor to replace it. The successor node is the minimum node in the right subtree or the maximum node in the left subtree.

Balanced BSTs

Here you learnt that given a data set different BSTs can be created from it. It all depends on the sequence in which the elements are inserted. One such example can be seen in the image below.

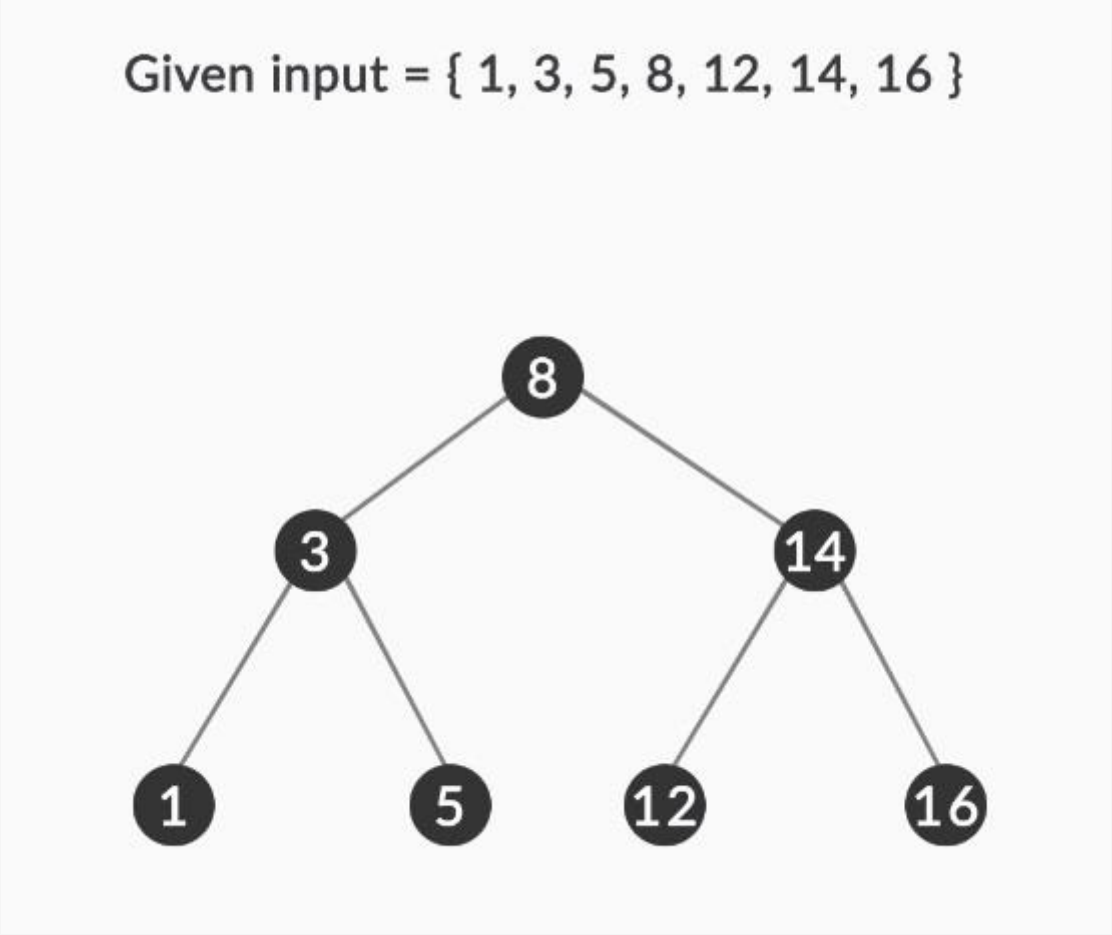


If the elements are inserted in the increasing order of their values, the tree essentially becomes a list and all the operations viz. adding a node, deleting a node or searching for an element become O(N) efficient.

Thus, to make sure that the operations are efficient the tree should be balanced i.e. the number of nodes in the left subtree should be comparable to the number of nodes in the right subtree. Ideally this number should be equal for both left and right subtrees.

So, pre-processing of data can improve the search efficiency of binary search trees.

Hence, the sorted data can be stored by choosing the median of all the elements as the root of the tree and iterating the same for all the subtrees, as shown in the figure below.



Here, 8 is the middle element. So, 8 is selected as the root of the BST. The middle element of the left part of the array is 3. So, 3 is the root of the left subtree. The elements 1 and 5 will become its left and right children, respectively. Similarly, 14 will be the middle element of the right part of the array, or the root node of the right subtree. Similarly, 12 and 16 will become its left and right children, respectively.

Storing the data in this way will always produce balanced binary search trees and, in turn, will enhance the search process efficiency as it reduces the number of comparisons required in the trees. You can read more about creating balanced BSTs from the data given in this link.

Additionally, you can read about self-balancing BSTs that will attempt to balance itself when new nodes are added to the tree. Find more information about self-balancing trees in the links provided below.

- Red-black trees
- AVL trees

Lecture Notes

Lecture Notes – Classes & Objects

Object-oriented programming, or OOP in short, gives you a particular methodology for implementing large and complex programming projects in a very simple manner. The methodology uses the structure of classes and objects, and four OOP principles, namely abstraction, encapsulation, inheritance and polymorphism, to simplify the large programming projects.

All programming languages are not based on this OOP model of classes and objects, and some are known as procedural programming languages. But most of the modern general-purpose programming languages, such as C++, C#, Java, and Python, are object-oriented programming languages.

Benefits of Object Oriented Programming:

1. The most important benefit of an object-oriented approach is that you tend to build any project in a more organised manner, making the development process very efficient.
2. The second benefit is that an object-oriented approach helps you implement real-world scenarios very naturally.
3. The third benefit is that the modification and updation of each independent module is easier in this case.

All of these benefits are the requirements of a good software development methodology; and through this module, you will see how all these requirements are met well by an object-oriented programming approach.

Classes

What is a class?
A class is a blueprint of an application, where you implement methods (to perform certain actions/functionalities) and declare variables (that describe the properties of the application). For example, Payments could be an entire class in itself in the Ola App.

In Java, every program has at least one class, whose structure looks like this:

A class can have 'variables' that have certain information in them, 'methods' that have functionalities linked with different tasks of that class, and it can also have subclasses. For now, let's focus only on the variables and methods.

Let's look at an example of a university's information management system. Now suppose a particular department needs to manage all the student information. So there can be a class named Student{ }. The pseudocode of this class will be

```
class Student{
   public int rollno;
   public String name;
   public double cgpa;
}
```

Apart from the variables, the classes mainly contain methods; these basically represent the functionalities of different activities related to each class. Now the activities that students may need to perform on the

information management system could be editing the profile, registering for a new course, submitting assignments, or checking their exam results. So in the case of a student class, the methods can be editProfile( ), displayProfile( ), registerCourse( ), submitAssignment( ), checkResult( ), etc. These methods can have their own functionalities. For example, you can have a code for printing a student's name, roll number and cgpa on the screen, implemented inside the displayProfile method, as shown in the sample code. This is more like a pseudocode. You can see that the return-type for this method is set to void as nothing is being returned from this method. Some of these methods, such as editProfile() or submitAssignment(), may have some arguments in them; but right now, you need to go into the actual implementations of these codes. This is a pseudocode representing the structure of a class:

```
public class Student{
   public int rollno;
   public String name;
   public double cgpa;
}
```

You can see the nomenclature pattern of the classes, which is always a single word with the first letter in upper case. So just like you had Student, there can be other classes such as Faculty{ }, Staff { }, etc.

```
public class Student{
      ...
}
public class Faculty{
      ...
}
public class Staff{
      ...
}
```

The **whole idea behind classes is to have independent modules in the program, each class representing one particular entity**. That entity can have various data members, variables, and methods linked to it. For example, Circle & Square can be examples of different classes in a drawing software. The data members for Circle and Square would be variables such as radius and length, respectively.

And as a good practice, keep the **public static void main** method in a separate class that was named **Main** in our example. Also, you can have a different name for the class that contains the main method. However, the name of the Java file has to be the same as the name of that particular class.

Let's say you need to build a program to calculate the area of a circle. The most basic structure for doing this would be to create a Circle class that contains a main( ) method and a findArea( ) method. Now you would start with what you already know about writing a code with a single method only and then convert it into the ideal object-oriented programming structure.

Let's look at this code in detail. You will define your findArea method in the same way as you defined the main method, except for one thing: the return type for this method will be double since the method is returning an area that can be a decimal value. You can declare your function as `public static double findArea(double radius);` and inside this, you can write the formula for calculating the radius, and assign this to a new variable area that is declared as double. Here it is declared in one line, but you can also manage this process in two lines: declaring, assigning, and then returning the area.

Now you can call this function in main by assigning a value, let's say, 2, to the radius variable, and then calling the findArea( ) method and storing it in a variable called area. This process is called method

invoking. And then you can write the command to print this variable. So if you run this program, it will print the result on the console: the area of the circle is 12.56.

Now you need to modify this code to convert it into a better design as per the object-oriented programming style. As we discussed, you need to have two classes: a main class containing the main method and a Circle class containing the findArea( ) method.

```
public class Main {
  public static void main(String[] args) {
      double area = Circle.findArea(2);
      System.out.println("Area of circle is "+area);
  }
}
public class Circle {
  public static double findArea(double radius) {
      double area = 3.14 * radius * radius;
      return area;
  }
}
```

If you are declaring an independent class, Circle{ }, the best practice is to declare the variable 'double radius' in the class. Declare it in the same way as the method, that is, `public static double radius`. You can modify this method by passing no argument in the findArea( ) method, as your radius is already declared above, and if now, you write the radius inside your method, it will refer to the declared one.

Now you need to modify the main class accordingly; since you declared a variable radius in the Circle class, you can set any value for that radius from the main method by writing Circle.radius = let's say, 2. So you can see that even to access the variable, you need to write the same, 'Circle + dot operator', before writing the name of the variable; it is not in the current class. And now, since your variable radius is assigned a value of 2, you can invoke the findArea( ) method by passing no argument. This will give you the same output as before.

```
public class Main {
  public static void main(String[] args) {
      Circle.radius = 2.0;
      double area = Circle.findArea();
      System.out.println("Area of circle is "+area);
  }
}
public class Circle{
  public static double radius;
  public static double findArea() {
      double area = 3.14 * radius * radius;
      return area;
  }
}
```

So the program can be considered as having a good design, as per an object-oriented style, because it has all the information and functionality related to the circles in the Circle{ } class.

Objects

What are objects?

Class is a blueprint and objects are the blocks built upon this blueprint with certain state and behaviour.

What is state and behaviour of an object?

The state of an object refers to the specific values of the variables of the class the object belongs to. The behaviour of an object refers to the action performed by the object when called by a particular method from the class.

Example

Class	Circle		
Object	c1 circle of radius 5f	c2 circle of radius 10f	c3 circle of radius 20f
State of object	radius 5f	radius 10f	radius 20f
Behaviour of object	c1.area( ), which returns 78.525	c1.area( ), which returns 314.1	c1.area( ), which returns 1256.4

Relation between a class and its object

Let's take another example of a house class to understand this better.

Suppose that an architect designs a blueprint for a house. This same blueprint can then be used to build multiple houses. A single blueprint is useful to build several houses that have the same architecture. The concept of classes and objects is similar to this. Here, the **house blueprint acts as a class** and **a particular house represents the object**.

For a Student class, your objects would be different students such as Prateek, Ankit, Ajay, etc. There can be thousands of different objects, with each object having the characteristics of the Student class.

Let's say Ankit is an object of the Student class. This means that Ankit has a roll number, a name and a CGPA. These variables, name, CGPA and roll number, are the instance variables in this case. And the values of these instance variables represent the state of the object, Ankit. So if Ankit displays his profile and registers for a course, these actions represent the behaviour of the object, Ankit. Similarly, other objects of this Student class will also have a roll number, name, CGPA, etc., but the values for these instance variables will be different from Ankit's.

Now that you know what objects are, it's time to show you how you can create objects for a particular class, and how you can use these objects in the main class. So let's go back to the code you built earlier, in which you had two different classes.

You can use the object 'circle' in this example. This process is very similar to how earlier, you used the dot operator, with the name of the class, to access variables and invoke methods when you didn't have objects. Now, you will use the same dot operator, but this time, with a name for the object, which, in this ie, is circle with a small c. So write circle, the dot operator, and the name of the variable from the class you want to set. Now assigning the value 2, here, would mean that the radius of this particular circle is set to 2. So you defined the state of this circle. You can now check the behaviour of this object by invoking the findArea( ) method. To do this, you have a similar command. You need to write the name of the object, 'circle', with the dot operator, and the name of the method you want to invoke, which is findArea( ), in this case. You can run this particular program and see the output, which is very similar to your earlier programs. The only difference is that you used an object to run this program.

```
public class Main {
  public static void main(String[] args) {
      Circle circle = new Circle();
      circle.radius = 2.0;
      double area = circle.findArea();
      System.out.println("Area of circle is "+area);
  }
}
public class Circle{
  public double radius;
  public double findArea() {
      double area = 3.14 * radius * radius;
      return area;
  }
}
```

Multiple Objects

For the **Student class**, your **objects would be different students — s1, s2, s3, s4 —** with names such as Prateek, Ankit, and Ajay. There can be thousands of objects, where each object has all the characteristics of the Student class.

Instances: An object of a class is also referred to as an instance. The word "object" or "instance" can be interchangeably used.

For example, s1, s2, s3, etc. are all instances or objects of the Student class.

Instance variables: Instance variables are variables from the class, which become the characteristics (or states) of the objects/instances. Every object has its own copies of instance variables.

For example, roll number, name, CGPA are all instance variables of the Student class.

State and behaviour of instances: Specific values assigned to these variables for a given instance is represented as the state of that object/instance, calling the methods would be represented as the behaviour of that instance.

So now you can create as many circles as you want, and you can name them c1, c2, c3, and so on. You can also use these circles for any operations you want to perform. This is the real benefit of objects.

```
public class Main {
  public static void main(String[] args) {
      Circle c1 = new Circle();
      c1.radius = 2.0;

      Circle c2 = new Circle();
      circle2.radius = 4.0;

      // Adding radii of both the circles
       System.out.println(c1.radius + c2.radius);

      // Adding areas of both the circles
       System.out.println(c1.findArea() + c2.findArea());
  }
}

public class Circle{
  public double radius;
  public double findArea() {
      double area = 3.14 * radius * radius;
      return area;
  }
}
```

Modelling Real-life Scenarios

Object-oriented programming allows you to implement real-world scenarios very naturally. So let's see how classes and objects will help you do this. Consider a scenario in the information management system of a university, where a student registers for a course that is taught by a particular faculty. Now for this example, if you consider these three entities as three different classes, then what could the objects for each of these classes be? The instances for the Student class can be Ankit, Prateek, Ajay, etc. Similarly, the instances for the Course class can be C, C++, Java, Python, etc. Finally, the instances for the Faculty class can be Prof. Sujit, Prof. Tricha, Prof. Murli, etc. Now these names would represent the state of the objects for the Faculty class.

Constructors

Constructor is a special type of method used to initialize an object of a class and define values of the instance variables of the object.

You can call a Java constructor while creating a new object. It constructs the values for the instance variables, i.e. provides data for the object. This is why it is known as a constructor.

let's see why customised constructors are required in the first place. Let's take the same example as before, and let's say in this code, instead of defining the radius of the circle as 2, in the next line, the user may forget to assign a value to the instance variable radius, in the case of some objects. Now if you want to make a new circle and assign the radius as 2 in the same line, you can do this by passing the parameter through a constructor, and the parameter will be the radius you want to set, which is 2. Then, you have to create the customised constructor in the Circle class in which, this time, you have to pass an argument. This is because you passed 2 directly into the constructor in the main class. You can create this method by passing an argument, double r, and inside the method, just mention that this r is nothing but the value of the public variable radius defined in this class. So if you look carefully, you'll see that whatever is passed in this method, the argument is stored in the variable radius. In this example, 2 is the argument, so 2 gets stored in the radius for this particular object. So you basically made a customised constructor method so

that you can assign the value of the variables at the same time as you make an object. This will ensure that the values are assigned to the instance variables as soon as a new object is made, so that you don't forget to assign the values to instance variables while creating a new object.

```
public class Main {
  public static void main(String[] args) {
      Circle circle = new Circle(2);
      circle.radius = 2.0;
      double area = circle.findArea();
      System.out.println("Area of circle is "+area);
  }
}
public class Circle{
  public double radius;

   public Circle(double r) {
      radius = r;
   }


  public double findArea() {
      double area = 3.14 * radius * radius;
      return area;
  }
}
```

A constructor is used to initialize the state of an object.

Parameterized constructor: A constructor that has parameters is known as a parameterized constructor. A parameterized constructor is used to assign different values to the instance variables of distinct objects.

Default constructor: A constructor that has no parameter and does nothing apart from creating a new object is known as a default constructor. It is a method that need not be declared if there is no parameterized constructor, or you can declare it and leave it empty in case of a declaration of any other parameterized constructor. Default constructors assign default values to the instance variables of the objects depending on the type, for example, 0, null, etc.

To help better your understanding of how these constructors help you make your codes shorter, and how they take care to ensure that you don't forget to assign values to the instance variables when you make new objects

The 'this' keyword

The **this** keyword is used to **refer to the current class instance variable**. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem.

For example, if there is an instance variable named name declared inside a class, it can be referred to as this.name in all the methods within that class.

The 'static' keyword

The **static keyword** is used for memory management, i.e. it helps in making your program memory-efficient. The static keyword belongs to the class, rather than any particular object.

Static Variable: Variables declared as static are known as static variables. Static variable is used while referring to the common property of all the objects of the class. For example, university name for students in the Student class for an information management system of a university. Static variables are useful if you need to share the same set of information across all objects of a class, such as the University name in our example.

The static variable is allocated memory only once, at the time of loading a class.

Static Method: Methods declared as static are known as static methods. A static method belongs to the class, rather than object of a class. It can be called without creating the object of the class, using ClassName.staticMethod( ) notation.

Static methods cannot access non-static variables, or call non-static methods. But static methods can access any static variable and change its value.

The 'final' keyword

The **final keyword** is used to restrict the user. If any method is declared as final, you cannot override that particular method.

Final variables: If any variable is declared as final, you cannot change the value of that variable. It can be assigned any value only once, and then it remains constant throughout the program. Final variables are useful if you have global constants in your programs that should never be changed.

Final methods: If you declare any method as final, you cannot override that.

Abstraction

What is Abstraction?
Abstraction is a principle based on **hiding the details of implementations of classes**, and access only certain features/functionalities given to the users/other parts of the program. This is done by writing your program using the framework of classes and objects.

Advantages of abstraction:

1. Abstraction allows you to use the functionality (behaviour) of the objects from other parts of the code without showing the internal implementation of that functionality (methods).

2. Rights to change the implementation of a class can be given only to a certain set of users. Others cannot go and arbitrarily change anything within the class, apart from using the behaviors defined by that class by creating and using its objects.

While programming projects also, abstraction is sometimes very necessary in cases where you don't have to show the implementation to certain users, but you just need to give them access to use the functionality. You can create multiple objects for a class, assign values to them, invoke methods, perform operations, and do a lot more from the main class. But you can't go and edit the class and the methods inside. That implementation is limited to specific people only.

Public & Private Access Modifiers

Access modifiers are used to restrict the accessibility of methods or variables of a class. There are four types of access modifiers—public, private, protected, and default—in Java. Let's learn more about them.

Public Access Modifier

When you declare a variable or a method as public in a class, it signifies that it can be accessed throughout the class. You can access these variables and methods from other classes as well. When a method of a class is declared as public, it can be accessed by creating an object of its class in other classes. If it is a static method, then you can directly access it using className.method() without creating an instance.

This creates potential issues because other classes can now access these public variables and methods and write code that depends on them. This will reduce your flexibility to change the implementation of these public variables and methods at a later point. This is because it will affect other classes that depends on these public variables and methods.

Private Access Modifier

When you declare a variable as private in a class, it signifies that it can be accessed throughout the class but not outside of it. You can't access it from another class. When a method is declared private, it cannot be accessed by creating an object outside the class. It is restricted only to the class that contains it.

Getter Methods

Getter methods are used to access private variables from outside the class in which they are declared. The standard way in which a private variable can be declared along with its getter method is as follows:

```
class Demo {
  private int var = 5;      //assigning a value to the private int variable

  public int getVar() {
      return var;
  }
}
```

After the previous code, you can access the value assigned to this variable from outside the class. You can do this by calling the getVar( ) method from outside the class in the following manner.

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

        Demo d = new Demo();

        d.getVar();        //calling getVar method to access its value
  }
```

Setter Methods

Setter methods are used to set a new value to private variables, or modify their values from outside the class in which they are declared. The standard way in which a private variable can be declared along with its setter method is as follows:

```
class Demo {
  private int var;      //just declaring a private int variable

  public void setVar(int var) {
      this.var = var;
  }
}
```

After the previous code, you can assign a new value to this variable from outside the class by calling the setVar( ) method from outside the class in the following manner.

```
public class ClassesAndObjects {
  public static void main(String[] args) {
        Demo d = new Demo();

        d.setVar(5);        //calling setVar method to modify its value
  }
}
```

Difference between Constructors & Setter Methods

1. **Constructors are automatically called as soon as the object** is being created, whereas **setter methods can be called after the object is created**, for example, when you want to set/modify the value of any instance variable.
2. Name of a constructor is always the same as the name of the class.
3. No return type (void, int, String, etc) is mentioned in case of constructors.
4. Different constraints for the value of variable can be added to the setter methods. You can also call the setter method from the constructor, if you want to place constraints on the values of the instance variables of the class while creating a new object. This can be done in the following way:

```
class Demo {
  private int var;         //just declaring a private int variable

public Demo(int var) {       //declaring parameterised setter method
      setVar(var);
  }

  public void setVar(int var) {     //declaring setter method
      this.var = var;
      if (this.var < 0){                //adding constraint
         this.var = 0;
      }
  }
}
```

Encapsulation

What is encapsulation?
Encapsulation is a principle based on **hiding the state of objects and restricting their access** to various parts of your program. Encapsulation is achieved with the use of private access modifiers and the getter and setter methods.

How do you create a fully encapsulated class?

1. Make all the instance variables of a class private.
2. Only use getter and setter methods to read or write values of the instance variables.

Advantages of encapsulation:

1. You can make your class read-only or write-only by declaring only getter or setter methods. This prevents other code or malicious code from accessing instance variables in your class that they should not read or modify.
2. You can add variable logic/constraints in the setter methods, so you have full control of the data that can be assigned to the instance variables.

Both of the above points contribute to the **objective of code-safety**, which we discussed at the start of this session.

Classes whose variables are declared as private and can only be accessed through methods are known as encapsulated classes. Encapsulation is also referred to as the process of wrapping all data and code acting on the data i.e. the methods together, into a single unit, which is your class in this case. In your encapsulated class, the information or data of that class is private or hidden from the user. Let's look at some of the benefits of encapsulation. The first one is,you have some control on what values can be stored in the variables. No one can come in and save any junk values in these variables. The second benefit is you can make these variables either read-only or write-only depending on your program. What this means is if you want to make the instance variable write-only, then you can create a setter method for that without creating any getter method. And in case of the read-only variable, you can just create a getter method without creating a setter method.

Module Summary

So far, you learnt about two important principles of object-oriented programming. The first one is abstraction - i.e. implementing the structure of your classes and objects in your program appropriately, to hide implementation of the classes from the user and by solely giving access only to the instance variables and the ability to invoke the methods for different objects. The second design principle is encapsulation - this will keep all the instance variables private and gives its access to the user through getter and setter functions only. This is done to hide the information or variables of that class and give access to that information, only through certain methods. This also restricts the storage of junk values in the variables. You have seen the examples of encapsulation through the Circle class.

1. **Classes & Objects:** 'Classes and Objects' is the framework of Object Oriented Programming which revolve around the real-life entities.

Class: A class is a blueprint from which objects are created. It represents a set of attributes and methods that represents a group of entities, and common to all the objects of one type.

Object: An object represents a real life entity, which belongs to a class with same properties as that of a class. The object can perform all functionalities declared in a class, by invoking the methods declared there.

2. **Constructors:** A constructor is used to initialize the state of an object.

.

Default constructor: A constructor that doesn't need to be declared and is without any parameter is known as a default constructor. A default constructor can assign user defined ors the default values to the instance variables of the object depending on the type. For example, 0, null, etc..

Parameterized constructor: A constructor that takes in some parameter for the initialisation of instances variables while creating an object is known as a parameterized constructor. A parameterized constructor is used to assign different values to the instance variables when creating of distinct objects.

3. **The this, static & final keywords:**

.

The this keyword: The this keyword can be used to refer to the current class instance variables. If there is ambiguity between the instance variables and parameters, the this keyword resolves the problem.

The static keyword: The static keyword is used for memory management and helps in making your program memory-efficient. Anything which is declared static belongs to the entire class, rather than only the instance of the class, i.e. a static member is the common property of all the objects of the class.

The final keyword: The final keyword is used to restrict the user from updating the value of instance variables. If any variable is declared as final, you cannot change the value of that variable. If any method is declared as final, you cannot override that particular method.

4. **Abstraction:** Abstraction is a principle based on **hiding the details of implementations of classes**, and access only certain features/functionalities given to the users/other parts of the program. This is done by writing your program using the framework of classes and objects.

5. **Private access modifier:** A type which is declared while declaring any member of a class (instance variables, methods, etc.). Private access modifiers restrict the access of members within their own classes only.

6. **Declaring instance variables of a class as private:** All instance variables of a class should be declared as private in order to restrict the access to the variable's data from outside the class, which makes the program safer.

7. **Getter methods to read the value of private variables:** A public getter method is declared in the class to give read-only access of the private variable from outside the class.

8. **Setter methods to write the value of private variables:** A public setter method is declared in the class to gives write-only access to the private variable from outside the class. You can add constraints/logic of values that can be assigned to these variables.

9. **Declaring private methods and constructors as private:** Methods declared as private cannot be called from outside that class. Constructors declared as private don't allow the creation of objects of that particular class outside that class..

10. **Making a fully encapsulated class:** We can boost the safety of a program by declaring all the data members/instance variables of that class as private. These classes can be made read-only, or write-only by declaring only getter or setter methods.

Lecture Notes

Divide and Conquer

So, in this session we dealt with various algorithms for a very important technique called searching. Before getting into the algorithms, we stated the importance of having a time efficient algorithm for searching where we discussed the Facebook example. You learnt that Facebook despite having a database of billions of users, takes little time to search for an email id the user enters. An efficient algorithm makes it possible. Then we dived into the following search algorithms

Linear Search

In Linear Search, you learnt that the algorithm compares the key element with each and every element in the array, starting from the first index, till the comparison results in a match. So, if the key element lies at the end of the array having N elements, the algorithm takes N steps to search for the key. The following code was used as Java implementation of Linear Search.

```
public class LinearSearch {

   public static int linearSearch(int[] arr, int key) {

      int size = arr.length;
      for (int i = 0; i < size; i++) {
         if (arr[i] == key) {
            return i;
         }
      }
      //This is the default value if the key is not found in the array.
      return -1;
   }

   public static void main(String a[]) {

      int[] arr1 = {23, 45, 21, 55, 234, 1, 34, 90};
      int searchKey = 34;
      System.out.println("Key " + searchKey + " found at index: " + linearSearch(arr1, searchKey));
      int[] arr2 = {123, 445, 421, 595, 2134, 41, 304, 190};
      searchKey = 421;
      System.out.println("Key " + searchKey + " found at index: " + linearSearch(arr2, searchKey));
   }
}
```

As per the code, if the element is found at the ith index, it returned the index i. In case the element is not found in the array, it returned -1.

Analysis of Linear Search

You learnt that Linear Search takes N steps in worst case to search for an element in the given array of N elements. This is because if the element lies at the end of the array or it does not exist in the array, the algorithm has to go through all the elements of the array to find the match or to ensure that the element does not exist in the array. Hence the Big O for Linear Search came out to be O(N).

Divide and Conquer

In this segment, you learnt that dividing a problem into smaller sub problems and then solving these subproblems to come up with a solution greatly improves the efficiency of the solution. This is something similar to what you learnt in Computational thinking in decomposition segment. In the Divide and Conquer segment, we discussed a phonebook example. There we were supposed to search for a name Cheng in the phonebook. We started at the middle (dividing the sample space into 2 halves) where the names were starting with L. Since C comes before L, so we ignored all the names that were to the right of middle and started searching to the left of L. In case we were searching for a name that starts with T, we would have ignored all the names to the left of L and would have started searching to the right of L. This straight away reduces the sample search space to one half of the original and we do not need to waste time searching there. On the ideas of Divide and Conquer, we moved on to another search algorithm called Binary Search.

Binary Search

As discussed in the videos, the first and foremost requirement for Binary Search is a sorted array. Binary search can only be applied to a sorted array.

In Binary Search, you learnt that the algorithm first compares the key to the middle element. If the key< middle element, the algorithm moves to the left of the middle and if key>middle element the algorithm moves to the right of the middle discarding the other half. In case key = middle element the algorithm returns the index of the middle element itself. Once the algorithm moves to the left or the right of the middle, it again computes the middle index of the corresponding half of the array. It again compares the key with the new middle element and moves to the left or right of the middle depending upon the comparison of the key and the middle element. The process continues till the match is found or the algorithm reaches the end of the array. Eg. If we were to search for the element 9 in the following array using Binary Search,

1	3	5	7	9	11

the algorithm first compares 9 with the middle element 5. Since 9>5, so it moves to the right and we are left with the following array.

7	9	11

It now compares 9 with the middle element of this array. This time the middle element is 9 itself, so the match is found and the algorithm returns the index of this element.

Code for Binary Search

The following code was used for the Java implementation of Binary Search.

```
public class BinarySearch {


    public int binarySearch(int[] inputArr, int key) {
```

```
        int start = 0;
        int end = inputArr.length - 1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (key == inputArr[mid]) {
                return mid;
            }
            if (key < inputArr[mid]) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {

        BinarySearch mbs = new BinarySearch();
        int[] arr = {2, 4, 6, 8, 10, 12, 14, 16};
        System.out.println("Key 14's position: " + mbs.binarySearch(arr, 14));
        int[] arr1 = {6, 34, 78, 123, 432, 900};
        System.out.println("Key 432's position: " + mbs.binarySearch(arr1, 432));
    }
}
```

The first part of the code defines the binary search function. It takes in 2 arguments, the integer array and the key the user is looking to search for.

As evident from the code itself, it first computes the middle element by dividing the sum of first and last index by 2.

```
int mid = (start + end) / 2;
```

The code then compares the key with the element at middle index. And if this results in a match, the index of the middle element is returned.

```
if (key == inputArr[mid]) {
            return mid;
        }
```

If the element at the middle index is greater than the key, the algorithms moves to the left of the middle. This is equivalent to setting the end index to mid-1.

```
if (key < inputArr[mid]) {
            end = mid - 1;
        }
```

If the key is greater than the element at the middle, the algorithm moves to the right of the middle. This is equivalent to setting start index as mid+1.

```
else {
            start = mid + 1;
        }
```

In case the element is not found, it returns -1.

```
    return -1;
```

At last, in the main function two integer arrays are initialized. Also, the object of the class Binary Search is created to access the binarySearch function to which the integer array and the key are passed as arguments.

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

    BinarySearch mbs = new BinarySearch();
    int[] arr = {2, 4, 6, 8, 10, 12, 14, 16};
    System.out.println("Key 14's position: " + mbs.binarySearch(arr, 14));
    int[] arr1 = {6, 34, 78, 123, 432, 900};
    System.out.println("Key 432's position: " + mbs.binarySearch(arr1, 432));
  }
}
```

Analysis of Binary Search

They you learnt about the efficiency of Binary Search. The recursion equation came out to be T(n) = T(n/2) + 2 where T(n) represents the number of comparisons done in an array of n elements using binary search. The rationale behind T(n/2) in R.HS. is that in every next iteration, half of the elements are discarded. So, if the array has n elements initially, in the next iteration binary search will be applied on n/2 elements and so on. 2 is added to it because of 2 additional comparisons done in every iteration.

1) The comparison to check if the middle element is equal to key or not.
2) If not equal, we need a comparison to check if the middle element is less than or greater than the key.

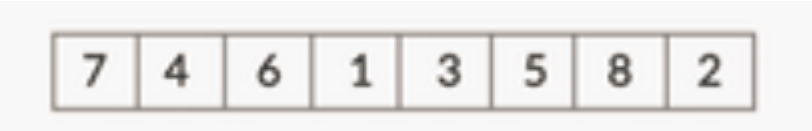
On solving this recursion equation, the Big O came out to be O(logn).

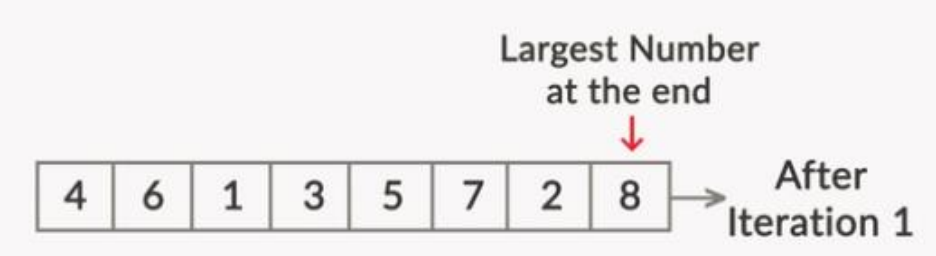
Session 2 – Sorting Algorithms -1

In this session you learnt about the sorting algorithms. You learnt that there are many real-life scenarios where sorting the data in certain order makes our lives a lot easier. Specifically, we discussed the Facebook example where you can sort your news feed based on the popularity. Also, on a shopping website the items can be sorted based on price, brands etc. In session 2 we discussed the sorting algorithms with a Big O efficiency of $O(N^2)$. The first algorithm discussed in this session was bubble sort.

Bubble Sort

You started with the first two numbers in each iteration and did a comparison between them. When the numbers were not in order, you swapped them. The same steps were repeated for the second and third number, and so on. At the end of the first iteration, the largest number was pushed to the end of the array; at the end of the second iteration, the second largest number was pushed to the second last position, and so on. For the array used in the videos, it looked something like this.





The following pseudo code was discussed for its java implementation.

```
FOR i = 0 to n-1
  FOR j = 1 to n-i
    IF(aj-1 > aj)
      SWAP(aj-1 , aj)
```

As per the pseudo code, for an array which is supposed to be sorted in an increasing order, if the number at (j-1) th index is greater than the number at the jth index, we swap them. The inner loop with j counter takes care of all the comparisons in each iteration and the outer loop corresponds to n-1 iteration required to sort the array.

One the pseudo code was discussed we moved on to the Java implementation of Bubble Sort. Following code was used by the Professor for this.

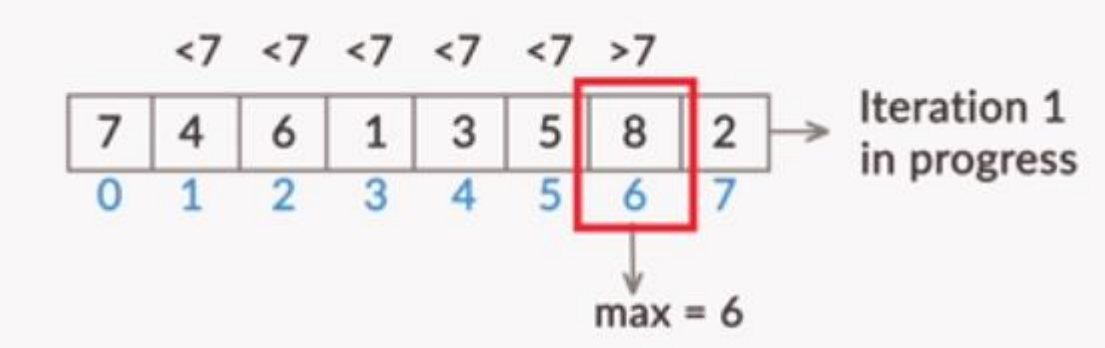
```
import java.util.Arrays;

public class BubbleSort {

   public static int[] sort(int[] numbers) {
       for (int i = 0; i < numbers.length; i++) {
           for (int j = 1; j < (numbers.length - i); j++) {
               if (numbers[j - 1] > numbers[j]) {
                   //swap elements
                   swap(j - 1, j, numbers);
               }
           }
       }
       return numbers; // returning the final sorted array
   }

   public static void swap(int i, int j, int[] array) {
       int temp = array[i];
       array[i] = array[j];
```

```
        array[j] = temp;
    }

    public static void main(String args[]) {
        int[] randomNumbers = {13, 3242, 23, 2351, 352, 3915, 123, 32, 1, 5, 0};
        int[] sortedNumbers;

        sortedNumbers = sort(randomNumbers); // bubble sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }
}
```

Once you learnt the Java implementation of Bubble Sort, we moved on to the analysis of this algorithm. As discussed by the Professor, bubble sort has a time complexity of $O(N^2)$, where N is the number of elements in the array. In first iteration, it does (N-1) comparisons; in the second iteration it does (N-2) comparisons; in third, it does (N-3) comparisons. This continues for a total of N iterations. So, the total number of comparisons effectively become the sum of first (N-1) natural numbers.

Selection Sort

The next algorithm that you learnt was Selection Sort. You learnt that selection sort also, at the end of each iteration, pushes the next highest number to the end. However, this time it was done with fewer number of swaps. You just picked the highest number in each iteration and swapped it with the last, unsorted number. So, each iteration in the worst case needs only one swap. This is unlike bubble sort where you must compare and swap every two numbers every time they are out of order. It locates the highest number and puts it at the end of the sequence. In the next iteration, it locates the next highest number and puts it at the second last position of the array and so on.



The following pseudo code was discussed for its java implementation.

```
FOR i = 0 to n
  FOR j = 1 to n-i
    IF(aj > amax)
      max = j
      SWAP(max,n-1-i)
```

As per the pseudo code if the number at jth index is greater than the number at the current max index, we update the value of max with j. At the end of the iteration, we swap the number at the max index with the number at the (n-1-i)th position. The following code was used for its Java implementation.

```
import java.util.Arrays;

public class SelectionSort {

    public static int[] sort(int[] numbers) {
        for (int i = 0; i < numbers.length - 1; i++) {

            int highestIndex = 0;

            // looping through the rest of the array to find a bigger element
            for (int j = 1; j < numbers.length - i; j++) {
                if (numbers[j] > numbers[highestIndex]) {
                    highestIndex = j; //bigger element found
                }
            }

            // swap the smallest element found with the ith element
            swap(highestIndex, numbers.length -1 - i, numbers);
        }
        return numbers;
    }

    public static void swap(int i, int j, int[] array) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String args[]) {
        int[] randomNumbers = {7,4,6,1,3,5,8,2};
        int [] sortedNumbers = sort(randomNumbers); // selection sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }
}
```

Post that, you learnt about the efficiency of selection sort. So, as far as Big O goes, selection sort also turned out to have a time complexity of $O(N^2)$. Remember that you learnt that it performs fewer swaps than bubble sort. So ideally, it should run a bit more efficiently than bubble sort. We then used the cards demonstration to solve this. There you learnt that the actual number of steps a selection sort takes is $\mathbf{N^2/2}$. But $\mathbf{N^2/2}$ also falls into the time complexity of $O(N^2)$. Therefore, despite selection sort being more efficient than bubble sort, it falls into the same Big O category as does bubble sort.

Insertion Sort

The next algorithm that you learnt was Insertion sort. As discussed by our professor, in insertion sort, you compare an element with the element to its left. If the element to its left is greater, you should shift the greater element to the right by one position and the smaller one to the left. In the next iteration, you need to compare this smaller element with the one to its left, and shift it if the element to the left is greater. You stop when you find that the element to the left is smaller than the element you are comparing with. Following pseudo code was discussed for its java implementation.

```
FOR i = 1 to n
  j = i - 1;
  t = a[i];
 WHILE(j>=0 && a[j]>t)
   a[j + 1] = a[j-- ];
   a[j + 1] = t;
```

As per the pseudo code, if a[i-1] > a[i], we swap them. The two lines written inside the while loop takes care of this swap. Please refer to the In-Video question asked in the video for the explanation of this pseudo code. The following code was used for the Java implementation of Insertion Sort.

```
import java.util.Arrays;

public class InsertionSort {

   public static int[] sort(int[] numbers) {
       // going through each number in int[] numbers
       for (int i = 1; i < numbers.length; i++) {
           int j = i - 1;
           int t = numbers[i];
           while (j >= 0 && numbers[j] > t) {
               numbers[j + 1] = numbers[j--];
           }
```

```
            numbers[j + 1] = t;

        }
        return numbers; // returning the final sorted array
    }


    public static void main(String args[]) {
        int[] randomNumbers = {10, 5, 2, 3, 4, 98, 67};
        int[] sortedNumbers;



        sortedNumbers = sort(randomNumbers); // insertion sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }
}
```

As far as analysis of Insertion sort is concerned, you learnt that it also has an efficiency of $O(N^2)$. However, it follows O(N) in the best case. So, if the array is almost sorted, i.e. it is nearer to the best case than the worst case, the insertion sort will have a time complexity somewhere between $O(N^2)$ and O(N). It will be nearer to O(N) if the array is almost sorted, and nearer to $O(N^2)$ if the array is arranged in the opposite order of our preference. For example, consider an array where all elements are in descending order, and you want to sort it in ascending order.

Insertion Sort vs Selection Sort

Next, we compared insertion sort and selection sort. In this segment you learnt that when two algorithms fall into the same Big O category, we need to think beyond Big O to choose more efficient of the two. The following example was discussed to communicate the idea.

Merge Sort

Then you moved on to the next session where you learnt about sorting algorithms following the efficiency of O(N logN). The first one that was discussed was Merge Sort. There you learnt that you keep on divind the array into halves until you reach one single element. After that these arrays, having one element each, are merged back in the required order. Following was the code that was discussed for the algorithm.

```
import java.util.Arrays;


public class MergeSort {
    public static int[] sort(int[] randomNumbers) {
        return mergeSort(randomNumbers, 0, randomNumbers.length - 1);
    }
    public static int[] mergeSort(int[] numbers, int first, int last) { if (first <
        last) {
            int mid = (first + last) / 2;
            mergeSort(numbers, first, mid);
            mergeSort(numbers, mid + 1, last);
            merge(numbers, first, mid, last);
        }
        return numbers;
    }
    private static int[] merge(int[] numbers, int i, int m, int j) { int l = i;
        //inital index of first subarray
        int r = m + 1; // initial index of second subarray int k = 0;
        // initial index of merged array int[] t = new int
        [numbers.length];
        while (l <= m && r <= j) {
            if (numbers[l] <= numbers[r]) {
                t[k] = numbers[l];
                k++;
                l++;
            } else {
                t[k] = numbers[r];
                k++;
                r++;
```

```
            }
        }


        // Copy the remaining elements on left half , if there are any while (l
        <= m) {
            t[k] = numbers[l];
            k++;
            l++;

        }


        // Copy the remaining elements on right half , if there are any while (r
        <= j) {
            t[k] = numbers[r];
            k++;
            r++;
        }
        // Copy the remaining elements from array t back the numbers array l = i;
        k = 0;
        while (l <= j) {
            numbers[l] = t[k];
            l++;
            k++;
        }
        return numbers;
    }
    public static void main(String args[]) {
        int[] randomNumbers = {13, 3242, 23, 2351, 352, 3915, 123, 32, 67,5, 9};
        int[] sortedNumbers;
        sortedNumbers = sort(randomNumbers); // Merge Sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }

}
```

Then you learnt about the efficiency of the merge sort algorithm. On analysis it was found that merge sort follows O(NlogN).

Master's Theorem

Next you learnt about Master's theorem. You learn that The master theorem helps in calculating the time complexity of algorithms that use recursion.

So, we discussed the following cases in the master theorem:

Case 1: $T(n) = (n_d)$ if $a<b_d$

Case 2: $T(n) = (n_d \log n)$ if $a = b_d$

Case 3 $T(n) = (n_{\log_b a})$ if $a> b_d$

Here, a, b and d are defined as follows:

$T(n) = aT(n/b) + f(n)$

$T(1) = c$

Here, $a > 1$, $b > 2$, $c > 0$

$(n) = n_d$

Quick Sort

The last sorting algorithm that was discussed was Quick Sort. You learnt that quicksort uses a pivot element, which is chosen randomly from the array, to partition the array such that all the elements smaller than or equal to the pivot are shifted to its left-hand side, and all the elements greater than the pivot are shifted to its right-hand side. Now, to do this partition, you needed a partition function. Therefore, you are using 'l' (the left pointer) and 'r' (the right pointer) to scan through the elements on the left and the right sides of the pivot, identify elements that are out of place, and swap those elements with the ultimate goal of arranging all the elements smaller than the pivot to its left and all the elements that are larger than the pivot to its right. You used two pointers, namely, 'l' and 'r'. The first pointer 'l' was used for the indices located on the left side of the pivot, and the second pointer 'r' was used for indices located on the right side of the pivot. We compared the element at 'l' with the pivot and kept incrementing the value of 'l' by 1 until an element greater than the pivot was found. If it was greater than the pivot, you stopped and started comparing the element at r. Similar to 'l', you kept decrementing the value of 'r' until an element smaller than the pivot was found. If such an element is found, you stopped, and at this point, the elements at 'l' and 'r' were swapped. Following code was discussed for Quick Sort.

```
import java.util.Arrays;
import java.util.Random;



public class QuickSort {
```

```
static Random random = new Random();


public static int[] sort(int[] numbers) { // let's sort
    numbers using quick sort quickSort(numbers, 0,
    numbers.length - 1); return numbers;

}



public static void quickSort(int[] numbers, int first, int last) { if (first <
    last) {
        // select a pivot point

        int pivotIndex = first + random.nextInt(last - first + 1); int pivot
        = numbers[pivotIndex];

        int k = partition(numbers, first, last, pivot);


        // recursively sort the elements to the left of the pivot
        quickSort(numbers, first, k);

        // recursively sort the elements to the right of the pivot
        quickSort(numbers, k + 1, last);

    }


}



public static int partition(int[] numbers, int first, int last, int pivot) {

    int l = first;

    int r = last;


    while (l <= r) {

        // In each iteration, we will identify a number

        // from left side which is greater than the pivot // value,
        and also we will identify a number from // right side which
        is less then the pivot value.

        // Once the search is done, then we exchange both numbers.


        while (l <= r && numbers[l] <= pivot) { l++;


        }



        while (l <= r && numbers[r] > pivot) {

            r--;


        }
```

```
            if (l <= r) {
                exchangeNumbers(numbers, l, r);
                //move index to next position on both sides
                l++;

                r--;

            }

        }


        return l - 1;
    }


    public static void exchangeNumbers(int[] numbers, int i, int j) { int temp =
        numbers[i];
        numbers[i] = numbers[j];

        numbers[j] = temp;


            // exchange numbers using XOR, which doesn't require a temp
variable

    }


    public static void main(String args[]) {
        int[] randomNumbers = {13, 3242, 23, 2351, 352, 3915, 123, 32, 67,
5, 9};
        int[] sortedNumbers;


        sortedNumbers = sort(randomNumbers); // Quicksort


        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));

    }

}
```

Then we discussed the analysis of this algorithm. On analysis it was found to follow the time complexity of O(NlogN) for an average case scenario. However, you learnt that in worst case Quicksort follows $O(N^2)$.

Comparison of Sorting Algorithms

In this segment we compared all the sorting algorithms you learnt so far. You learnt that bubble sort, selection sort, and insertion sort have a time complexity of $O(N^2)$ in average and worst cases. So, generally they are not used unless we are very sure that the dataset is very close to the best case (where insertion sort has an O(N) time complexity). Both merge sort and quicksort has an O(N log N) time complexity. But quicksort has $O(N^2)$ in the worst case. Also, merge sort is a stable sort, which quicksort is not. So, in case you need a stable sorting algorithm, the only sorting algorithm you can opt for is merge sort.

Lecture Notes

EL Armaments

Arrays and ArrayLists

Arrays in Java

Arrays: An array is a collection of elements of similar data types that are stored in contiguous memory locations. The length of an array is fixed, and only a fixed set of elements can be stored in a Java array.

The first index of an array is always 0.

Advantages of a Java Array

1. It is simple to create.
2. It is easy to access the elements within a Java array.

Disadvantages of a Java Array

1. Only a fixed number of elements can be stored in a Java array, and the length of the array cannot be increased or decreased.

Every time you create a new array (for example, 'random'), you also **create a new array object**. Depending on the data type of that particular array object, you can store either an int, double, string or any primitive data type/object in the array. You can also declare any variable as an array in the following ways:

```
int random[]
int[] random
```

Click on this link to learn how to declare an array of different data types.

ArrayList

A Java ArrayList is quite similar to an array and has the additional capability of adding or removing elements dynamically at runtime. This is why **ArrayList is also referred to as a dynamic array**. It is used to store a group of elements and allows you to store duplicates.

```
ArrayList studentList = new ArrayList();
```

In ArrayList, **all the elements are of the data type Object**, which needs to be typecast in order to be accessible.

```
public static void printStudentList(ArrayList students) {
  for(Object o : students) {
    Student s = (Student) o;
    System.out.println(s.getDetails());
  }
}
```

Type Safety

Type safety is a property of programming languages. Type safety means that a well-written program with no syntax errors detected at the compile-time should not throw a runtime error. You can learn more about type safety by referring to this additional reading material.

Now, **how could a single ArrayList store both 'String' and 'Student' data types?**

Let's revise the concept of inheritance to understand this better.

An object of a child class can be stored into a variable of a parent class. For example, if there are two classes, 'Cat' and 'Dog', which inherit the parent class 'Pet', then an object of 'Cat' class can be stored in a variable of type 'Pet', as shown below.

```
Pet ob1 = new Cat();
Pet ob2 = new Dog();
```

An ArrayList stores all elements of the data type 'Object', which is the parent class of all Java classes by default.

As 'Object' class is the parent class of both 'String' class and 'Student' class, you are able to store these data types in a single ArrayList.

Generally, **you should create an ArrayList of the same data type.** However, you may encounter runtime errors if you accidentally store an element that cannot be cast.

To deal with this error, you can define the ArrayList using generics.

ArrayList Using Generics

Defining ArrayLists using generics ensures that only the data of a particular data type can be stored in the ArrayList. For example, consider the following two lines of code:

```
ArrayList<String> names = new ArrayList<String>();
ArrayList<Student> studentList = new ArrayList<Student>();
```

In the 'names' ArrayList, only the data of data type 'String' can be stored. Similarly, in the 'studentList' ArrayList, only the data of data type 'Student' can be stored. This special ArrayList, in which you **cannot add elements** of different data types, is referred to as an **ArrayList using Generics**. If you do try to add elements of any other data type in this ArrayList, it will throw a compile-time error. Some of the advantages of using Generics are as follows.

1. The ArrayList can only hold a specific data type, and the elements of other data types are not allowed.
2. Typecasting is not required.
3. There is a conversion of potential runtime errors into compile-time errors.

Format to Declare ArrayList Using Generics

The class ArrayList can be declared using generics in the following method:

```
ArrayList<datatype> listName = new ArrayList<datatype>();
```

Here, the data type to be mentioned is always non-primitive (reference). For example, Student and String are declared as classes. Primitive data types such as int, char and double cannot be used here.

If you want to store primitive data types in ArrayList classes, you will need to use their Object cousins, which are Integer, Double, Float and Boolean.

You can create the Generics ArrayList with elements of certain data types using the following formats:

1. ArrayList of int-type values

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

2. ArrayList of double-type values

```
ArrayList<Double> list = new ArrayList<Double>();
```

3. ArrayList of float-type values

```
ArrayList<Float> list = new ArrayList<Float>();
```

Operations on ArrayList

The following basic operations can be performed on an ArrayList:

1. Adding an element at any arbitrary position
2. Removing an element
3. Searching for an element

The following methods can be used to **add elements** to the ArrayList:

1. **add(Object o):** This method appends the specified object ‘o’ to the end of ArrayList. Its return type is Boolean, which returns TRUE when the element is added to the list.
2. **add(int index, Object o):** This method inserts a specified object into a specified position in the ArrayList.

The following methods can be used to **remove elements** from the ArrayList:

remove(int index): This method is used to remove an element from the ArrayList at the specified index.

clear(): This method is used to delete/remove all the elements from the list, as shown in the example of the syntax of this method below.

```
studentList.clear();
```

The following method can be used to **search for an element** in the ArrayList:

contains(Object o): This method searches for the element in the ArrayList and returns ‘true’ if the element is present in the list.

Additional Reading

You can go through the Method Summary and Method Detail tables available on this page to learn more about other methods in the ArrayList class. You can use these methods to perform different operations.

LinkedLists (Optional)

A LinkedList is another data structure in Java, which works like an ArrayList and has similar methods and functionalities.

```
LinkedList<Student> studentList = new LinkedList<Student>();
```

However, **in a LinkedList the elements are not stored in contiguous memory locations.** This affects programs in terms of performance, which means even though the outputs of all the operations are the same, the speed at which these operations are conducted is different.

All the three methods, add, remove and contain, work in the same way in a linked list as they do in the ArrayList.

A LinkedList has another method that is used to access an element from a list through its index, which is as follows

get(int index): This method returns the element at the specified position in the list.

You can learn how this method works from this link.

Lists and Polymorphism

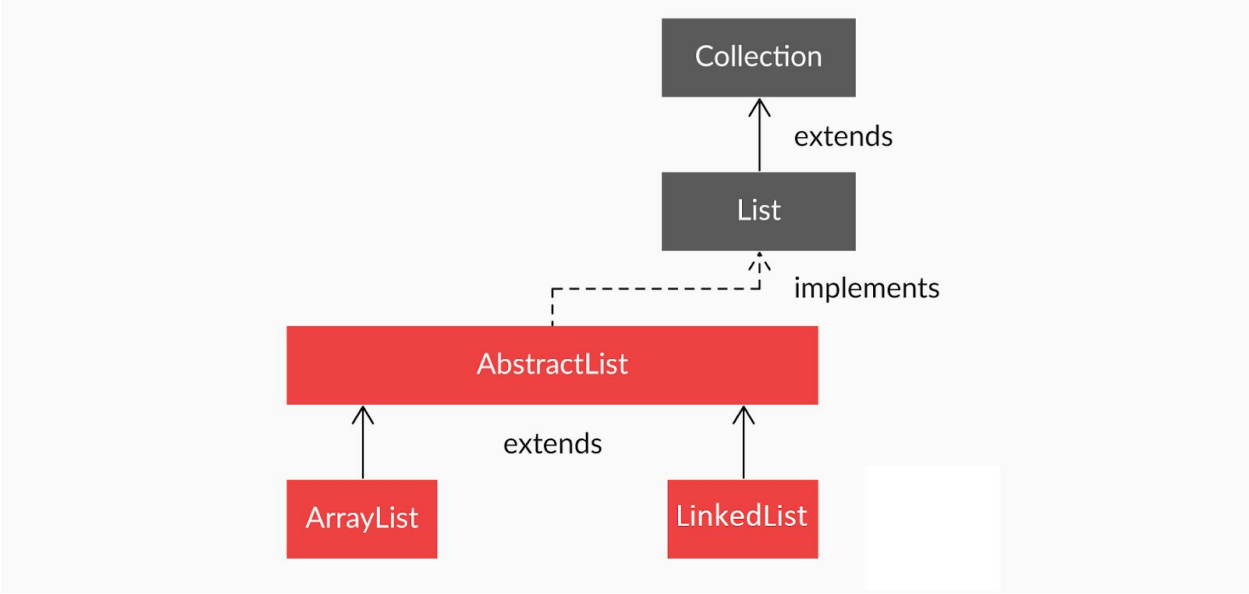
A **List** is an interface that is implemented by the **ArrayList** and **LinkedList** classes. This is why you can instantiate both 'ArrayList' and 'LinkedList' by declaring the type of the variable as List.

```
List<Student> studentList1 = new ArrayList<Student>();
List<Student> studentList2 = new LinkedList<Student>();
```

Given below is the printStudentList as a polymorphic function. It uses List instead of ArrayList or LinkedList, thus inter-operates smoothly with both types.

```
public static void printStudentList(List<Student> students) {
  for(Student s : students) {
    System.out.println(s.getDetails());
  }
}
```

You can refer to the diagram given below to understand how these classes and interfaces are linked to a larger interface called **Collection**.



1. The List interface extends the Collection interface, or in other words, List is the child interface of Collection.
2. AbstractList implements the List interface, which is further extended by the ArrayList and LinkedList classes. In other words, the ArrayList and LinkedList classes are implementations of the List interface.
3. The AbstractList class is extended by the ArrayList and LinkedList classes. In other words, ArrayList and LinkedList are the subclasses of the Abstract class.

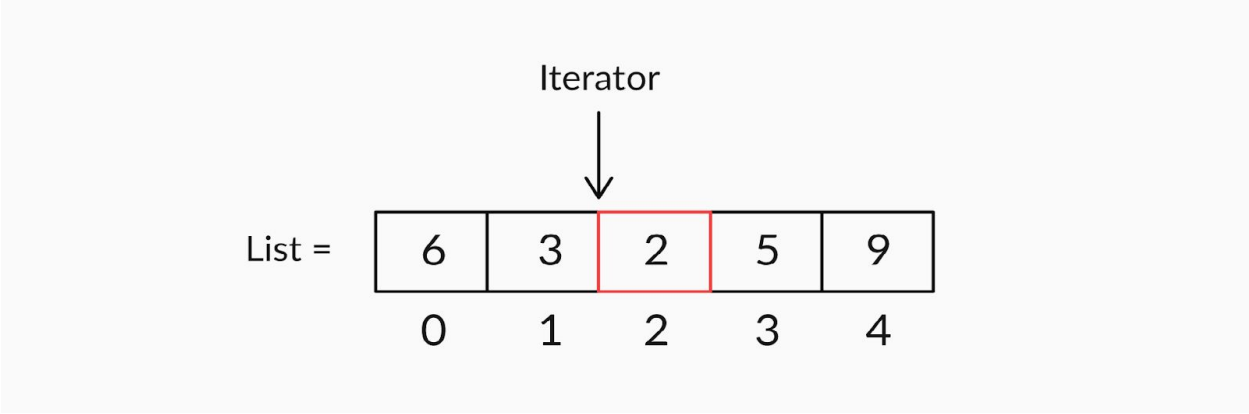
ListIterator

An iterator is an interface in Java, which is used to iterate over a collection of objects. In simple terms, an 'iterator' acts as a cursor to an element in a collection. You can also use the 'iterator' (or the cursor) to move to the next element in the collection.

```
ListIterator<Student> it = students.listIterator();
```

ListIterator is a subinterface of the Iterator interface, which is used to iterate over a list. It has a lot more features than the Iterator interface. Some of these features are as follows:

1. ListIterator is used to traverse a list in any direction, i.e., forward or backward, quite easily.
2. It does not point to any current element; its cursor position always lies between the previous and next elements (See Figure given below).
3. It has methods that can be used to determine whether a next or previous element is present in a list. You can also use the methods to find the value of the next or previous element, as well as its indices.



Iterating Forward Using ListIterator:

```
private static void iterateFwd(List<Student> students) {
  ListIterator<Student> it = students.listIterator();
  while(it.hasNext()) {
    System.out.println(it.next().getDetails());
  }
}
```

Iterating Backward Using ListIterator:

```
private static void iterateBkwd(List<Student> students) {
  ListIterator<Student> it = students.listIterator(students.size());
  while(it.hasPrevious()) {
    System.out.println(it.previous().getDetails());
  }
}
```

Additional Reading

Read this to know more about the additional methods of the listIterator interface.

Annotations

Annotations can be used to add some extra information about a piece of code. Such information can be used by other tools to treat that piece of code differently. You will be using a lot of annotations when you develop your own application.

All annotations in Java always start with '@'. For example, @Override.

@Override Annotation

What is the use of the @Override annotation?

When a method is marked with the @Override annotation, the compiler searches for that particular method in the parent class. If the method is not found in the parent class, then the compiler throws an error.

Advantages of using the @Override annotation

- Fewer chances of bugs (because of an incorrect method name)
- Better readability

@Deprecated Annotation

What is the use of the @Deprecated annotation?

The @Deprecated annotation is used to mark a piece of code that should not be used to write new code. If your code includes the deprecated code, the compiler will show a warning.

Advantages of using the @Deprecated annotation

- Warns the team/programmer about deprecated elements
- Helps in maintaining the hygiene of code

Additional Reading

Refer to this link to learn more about Custom Annotations.

Introduction to Version Control and Git

Version Control

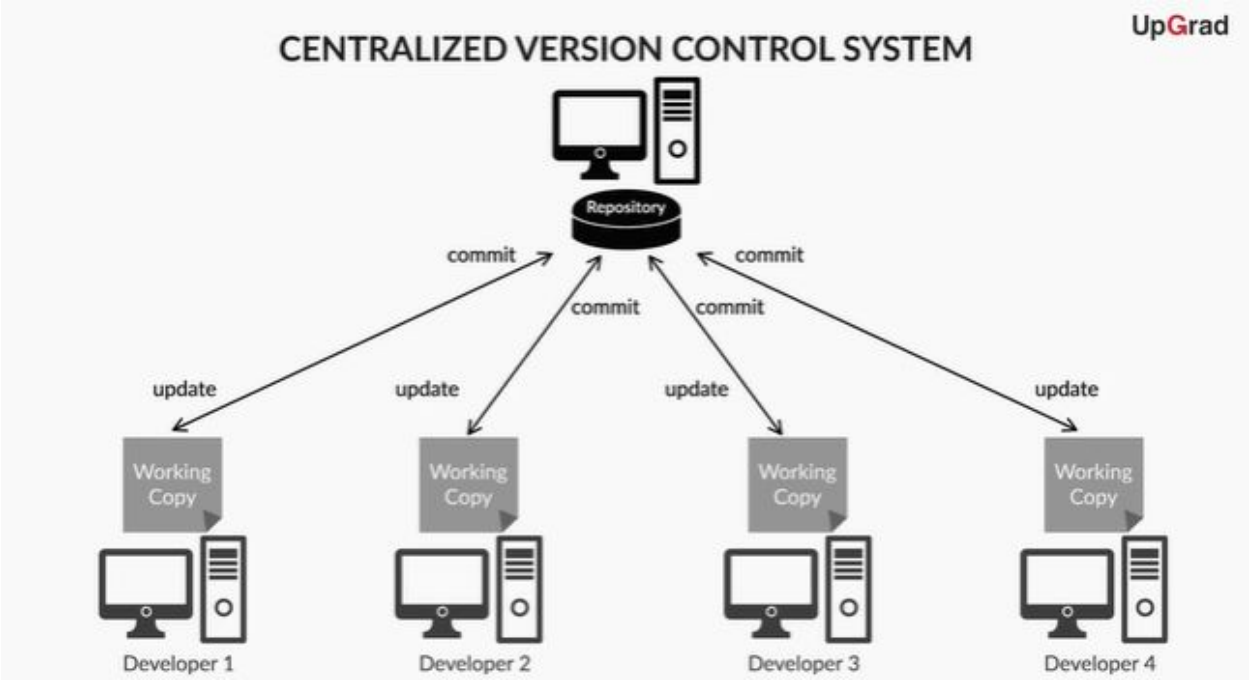
Version Control is a system that records or keeps track of the changes made to a file or a set of files over time so that you can recall or revert to specific versions later.

Types of Version Control Systems

Version control systems are of the following two types:

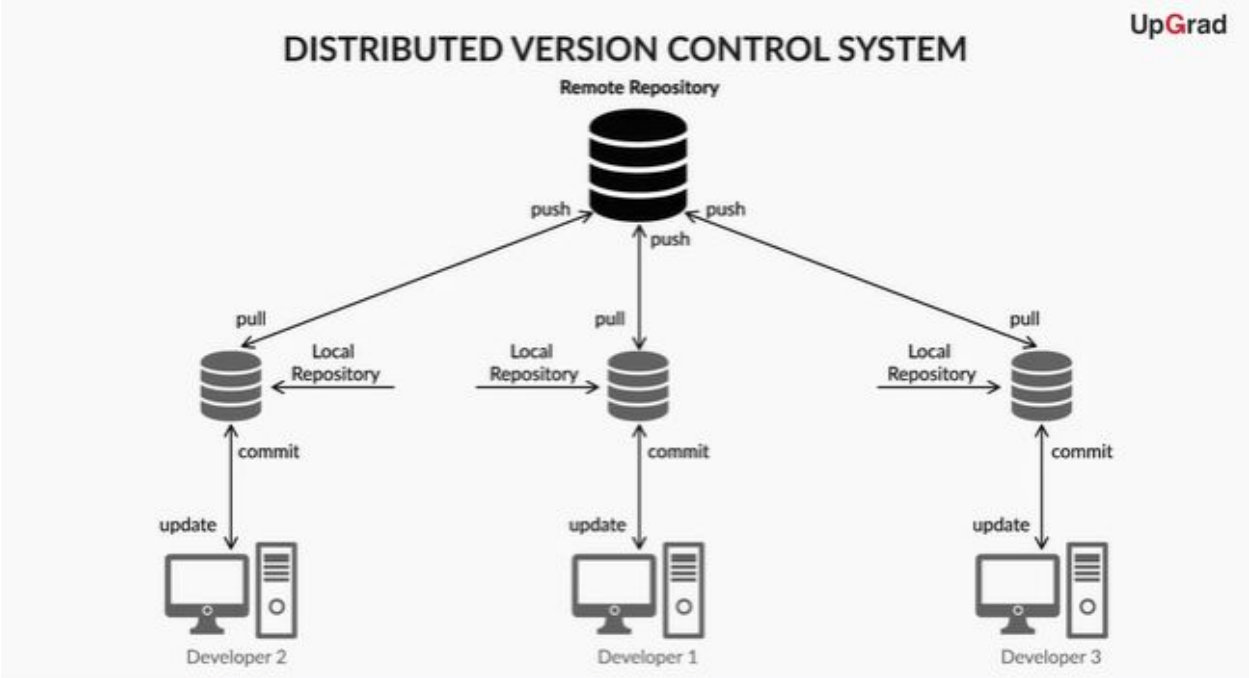
Centralized Version Control System

Centralized version control system works as a client-server model. This system has one centralized server and a localized repository file system that is accessible by many clients.



Distributed Version Control System

In the distributed version control system model, all developers have their own local file system, and changes between different file systems are implemented locally on their machines.



Git and GitHub

Git and GitHub are different.

Git is a **distributed version control system**. It is a tool that is used to manage your project source code history.

GitHub is a **web-based git file hosting service** that enables you to showcase or share your projects and files with others.

Repository: A repository is a directory that contains your project work. All the files in the repository can be uploaded to GitHub and shared with other people, either publicly or privately.

The three steps that your files may go through internally are as follows:

1. Modified
2. Staged
3. Committed

Basic git commands: Click here to read about some of the basic git commands.

- First-time git set-up
  - For the first-time git configuration, you need to use the following commands:
    - *git config --global user.name "random"*
      - This command will be used to enter your GitHub username.
    - *git config --global user.email "random@example.com"*
      - This command will be used to enter your GitHub username.

- Making a commit and pushing your changes to GitHub
  - Use the following commands to do this:
    - git init
    - git add filename
    - git commit -m "commit message"
    - git remote add origin <url of the remote repository>
    - git push -u origin master

- The following git commands that are used quite frequently:
  - **git status**: This command will display the state of the working directory and the staging area. In other words, it allows you to view the changes that have been staged and the changes that have not been added to the staging area.
  - **git log**: This command shows you the commit details. It lists out the commits made in the repository in reverse-chronological order, i.e., the most recent commits show up first followed by other commits. It shows commits along with the following details:
    - The commit ID or SHA

- Author's name (who made the commit)
- Date and time (when the commit was made)
- Commit message

The 'git remote add origin url' command

- You can use this command to add a new remote repository to your local repository. To do so, you should use the '**git remote add'** command on the terminal, in the directory where your repository is stored. The **git remote add** command takes the following two arguments:
  - A remote name, for example, origin (it can be any name)
  - A remote URL, for example, https://github.com/user/repo.git (the address of the repository on your GitHub account to which you want to link your local repository.)

Pushing changes after the first commit

When you are pushing your changes after the first commit, i.e., after:

1. Initialising your git repository, and
2. Linking it with the remote repository on git

you only need to run the following git commands to commit any changes into your local git repository and then push those changes to the remote repository on GitHub:

1. **'git status'**: This command is used to check which files are changed and not yet moved to the staging area.
2. **'git add <filename>' or 'git add .'**: This command is used to add modified files to the staging area. You can add a specific file using the command 'git add <filename>'. If you want to add all the modified and unstaged files present in the workspace to the staging area, you can use the 'git add .' command.
3. **git commit -m** "**New commit message**": This command gives a new commit message and commits all the files present in the staging area.
4. **git push -u origin master or git push**: This command is used to upload all the files and changes that were included in the most recent commit, to your remote repository on GitHub.

Downloading Repository

git clone

This command is used to clone or copy another user's repository. When you start at a new job, you will most likely work on existing projects. In order to get the code of that project in your system, you will need to clone it from an existing GitHub repository.

The syntax for the command is as follows:

git clone <url of the remote repository>

Example:

git clone https://github.com/user/repo.git

You can go through this link to learn more about this command.

Lecture Notes

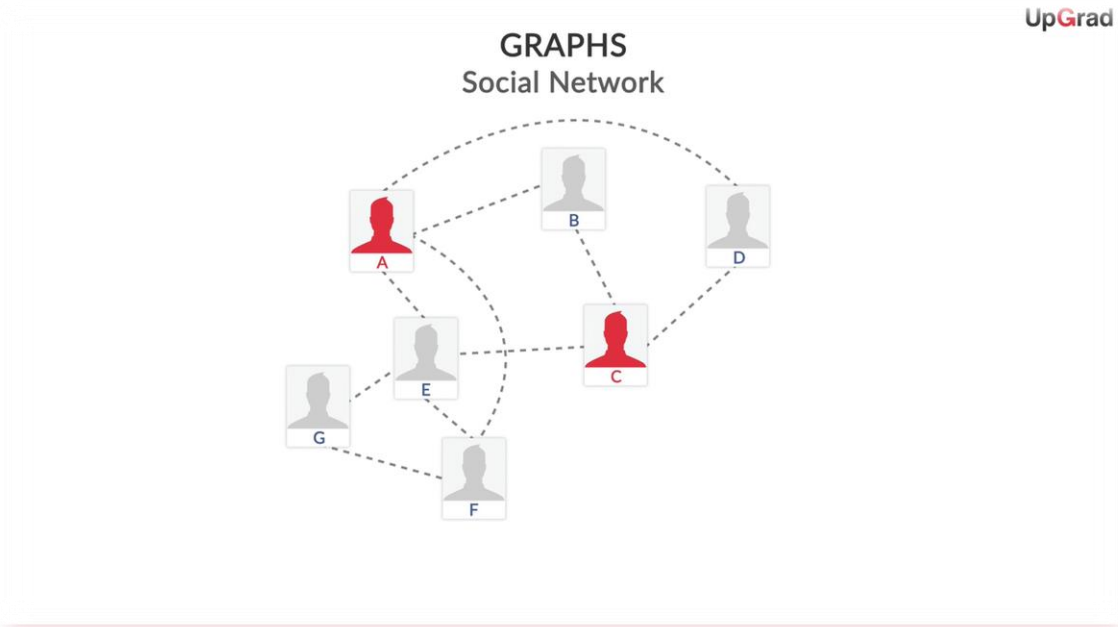
Graphs & Graph Algorithm

Session 1: Graphs

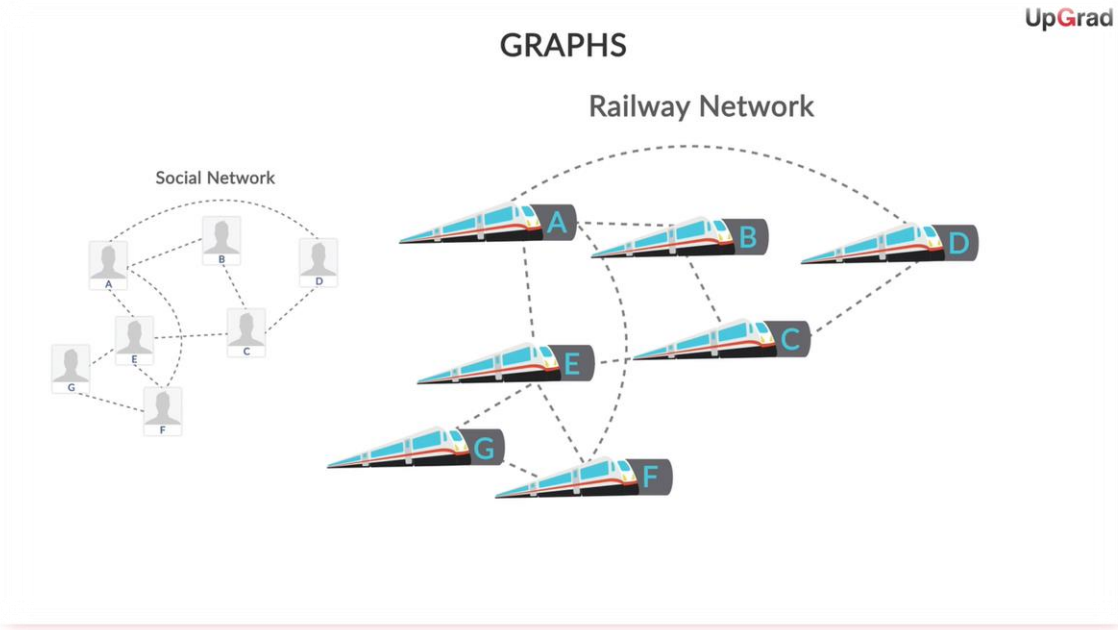
Introduction to graphs

In this session, you have learnt how graph data structures can represent the relationship between different entities. Here, the entities are called as nodes(vertices) and the relationship is called as edges(arcs).

Example 1: You have seen the example of social network, in which different people are represented as nodes and the edges connecting between them represent the friendship relation. The absence of an edge between A and C means that they are not friends

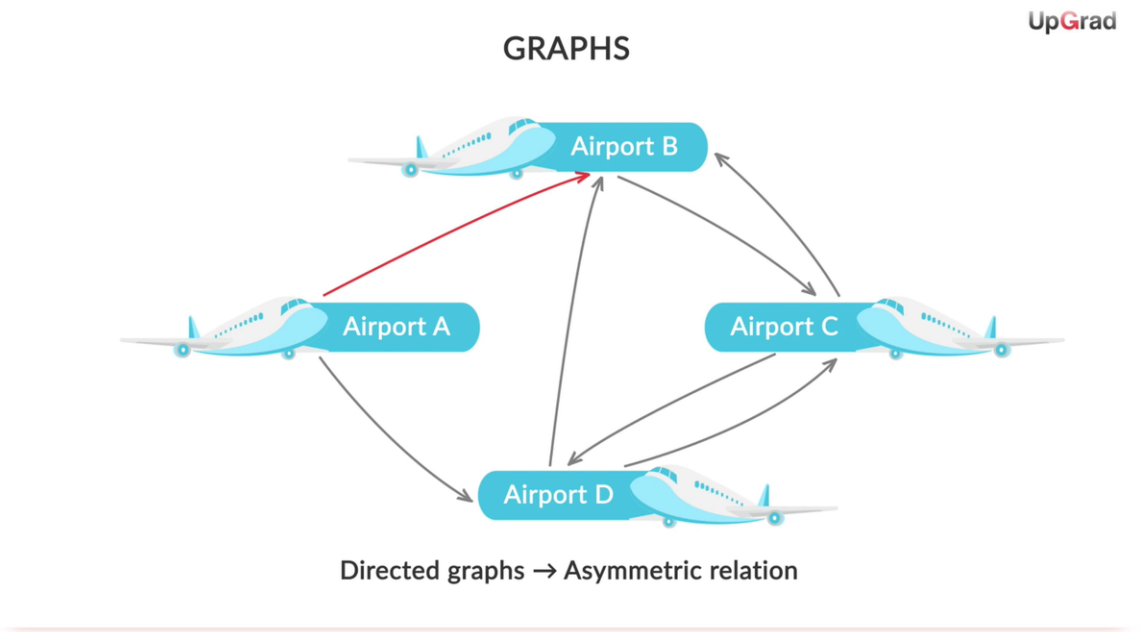


Example 2: Then, you have seen a railway network represented as a graph structure, where the railway stations are nodes and the edges connecting between different stations represent the railway track.



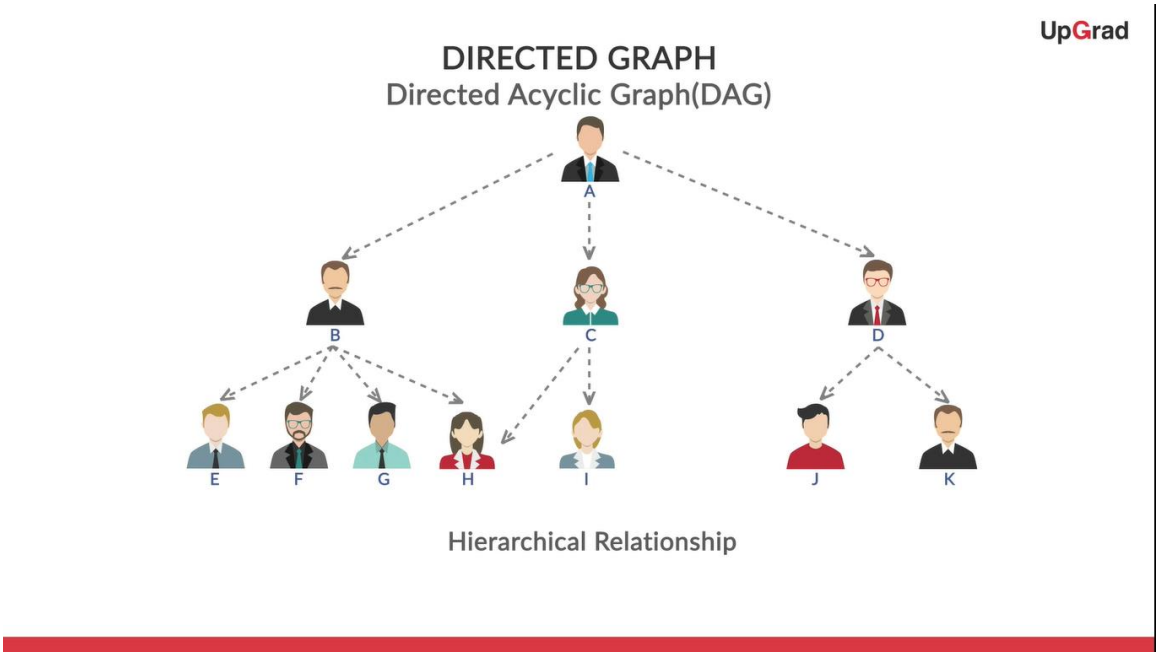
Based on the relationship between two nodes, graphs can be broadly classified as follows-

- Undirected graphs – symmetric relationship
  Example: The friendship relation among different friends in the social network represents undirected graphs. The edge connecting between A and B means A is a friend of B and vice-versa.
- Directed graphs – asymmetric relationship
  Example:
  The flight network among different airports can be represented using directed graph. In the below image, the directed arrow between Airport A and Airport B indicates a flight running from A to B but not from B to A.



In the above directed graph image, you can observe that if you start from Airport B and traverse through Airport C, Airport D and return back to Airport B. This is called as a cycle in the graph where you reach the same node after traversing across different nodes along the connected edges.

In the absence of such cycle in the directed graph can further be classified as directed acyclic graph.



You have also learnt that the directed acyclic graph is different from trees, in the above image node H has more than one parent node i.e., node B & C whereas in a tree data structure, each child node can have only one parent node.

Terminology

You have also learnt certain common terms while working on graphs as follows,

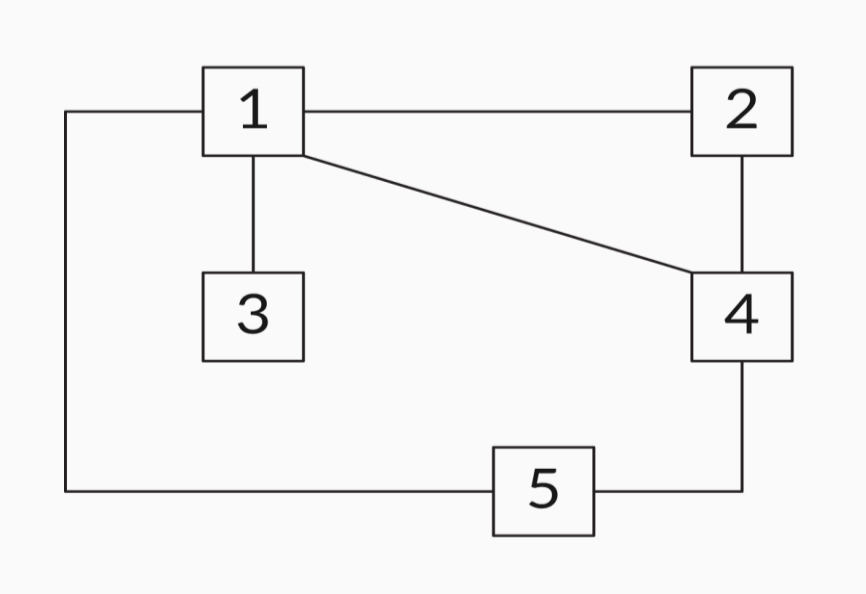
Neighbours: If two nodes are adjacent to each other and connected by an edge, then those nodes are called neighbours.

Degree: The number of edges that are connected to a node is called the degree of the node.

In case of directed graphs, this can be classified into:

- In-degree: The number of incoming edges to a node
- Out-degree: The number of outgoing edges from a node

  In the following undirected graph, degree of each node



Nodes	Neighbours	Degree
Node 1	{2, 3, 4, 5}	4
Node 2	{1, 4}	2
Node 3	{1}	1
Node 4	{1, 2, 5}	3
Node 5	{1, 4}	2

Path: When a series of vertices are connected by a sequence of edges between two specific nodes in a graph, the sequence is called a path. For example, in the above graph, {2, 1, 4, 5} indicates the path between the nodes 2 and 5, and the intermediate nodes are 1 and 4.

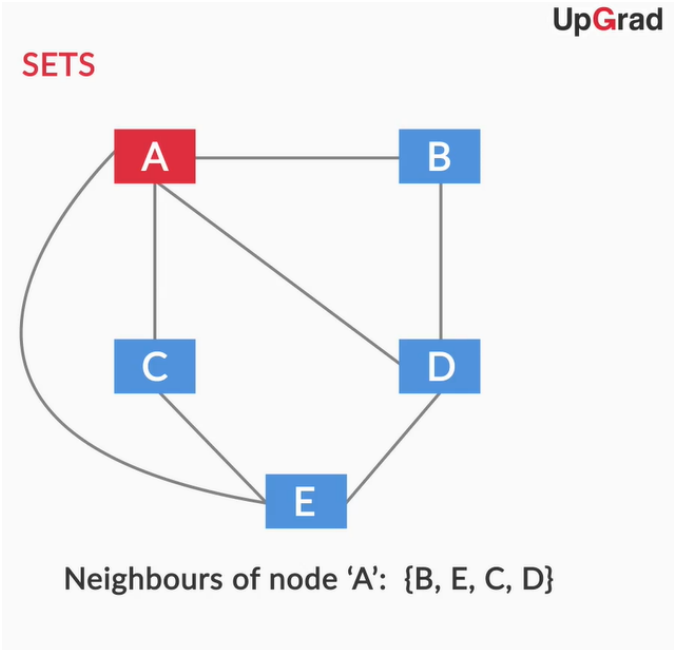
Sets and Maps

Before learning the traversal techniques of graph data structure, you have learnt the collection types sets and maps which are deliberately used to implement traversal techniques in Java.

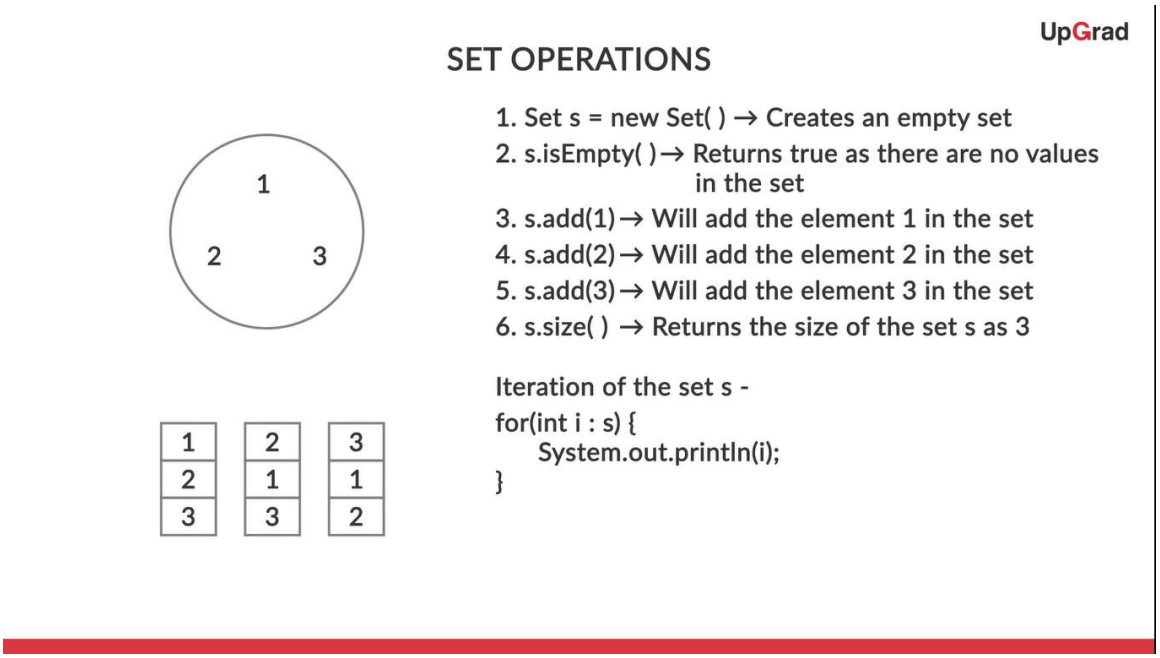
Sets

A set is a collection of unordered elements without any duplicates. It models from mathematical abstraction.

In the context of graph data structure, sets can be used to store the neighbours of a given node



You have also seen different operations performed on a given set which will be helpful in the implementation of traversal techniques of graph data structure.



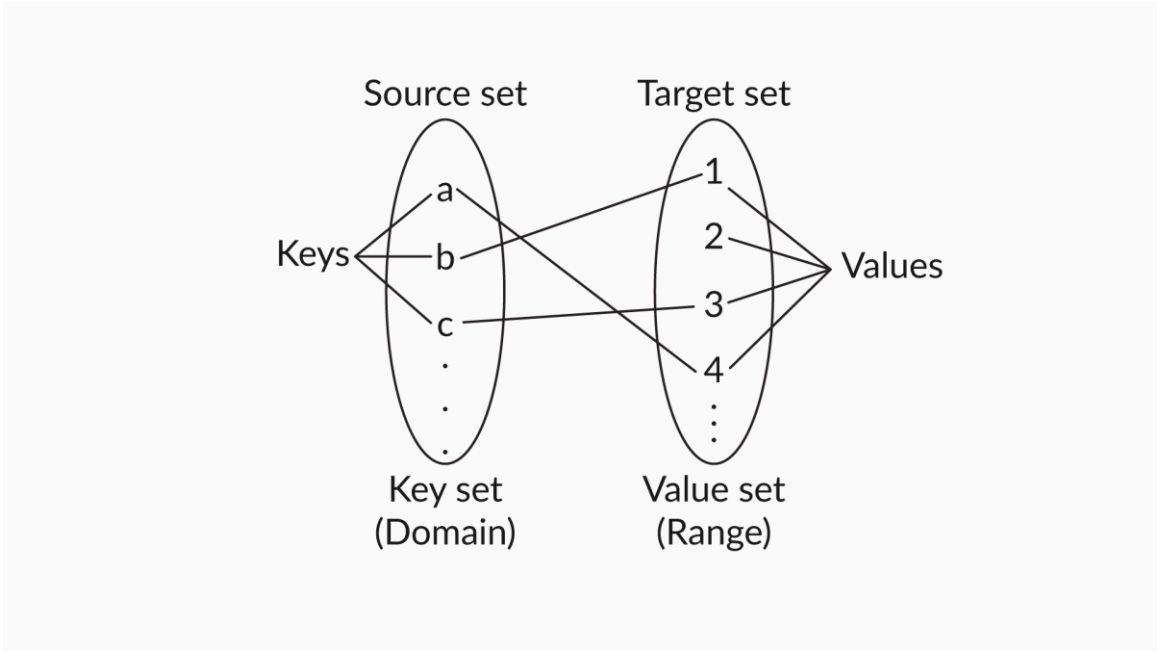
Properties:

1. Unlike lists and stacks, the elements present in a set do not follow any particular order. They are randomly present in the set.
2. The elements are not repeated in a given set.

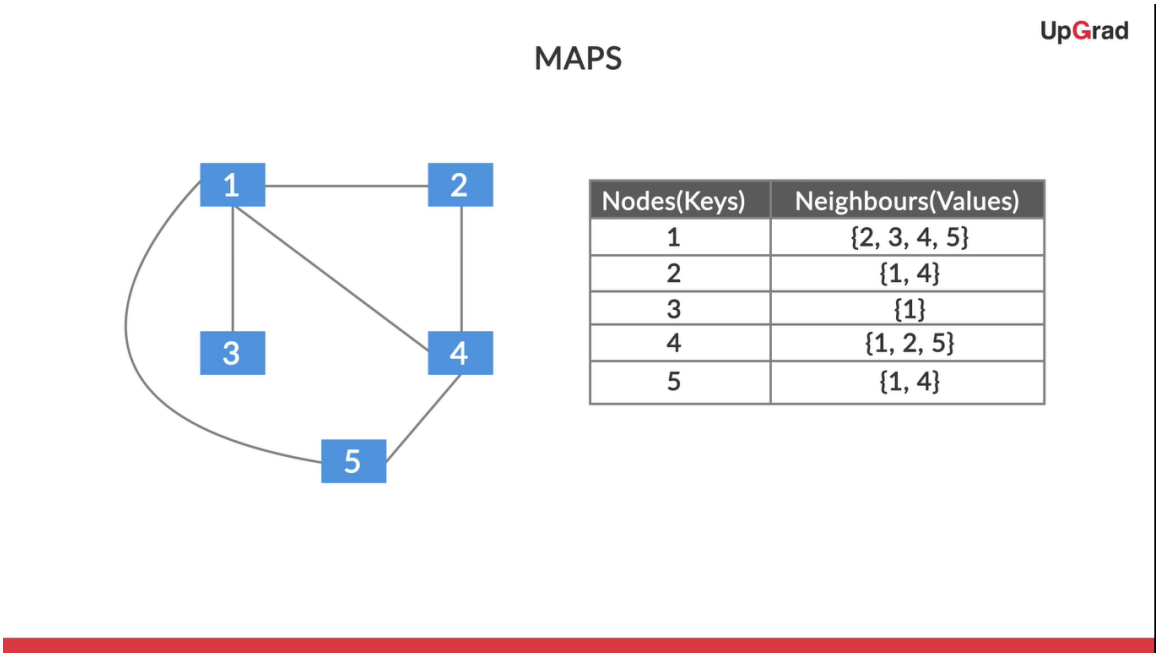
Methods	Description
add(ele)	Adds an element to the set
clear()	Removes all elements from the set
contains(ele)	Returns true if a specified element is in the set
isEmpty()	Returns true if the set is empty
remove(ele)	Removes a specific element from the set
size()	Returns the number of elements in the set

Maps

Maps is a collection type where it provides connection or mapping between the elements of source set(domain) and target set(range).



In the context of graph data structure, maps can be used to relate all the neighbours of a given node as follows, where nodes are keys and neighbours are values.



Nodes(Keys)	Neighbours(Values)
1	{2, 3, 4, 5}
2	{1, 4}
3	{1}
4	{1, 2, 5}
5	{1, 4}

You have also seen different operations performed on maps which are deliberately used in the traversal techniques of graph data structure

UpGrad

MAP OPERATIONS

Price List

Commodity	Price (in Rupees/kg)
Potato	30
Onion	25

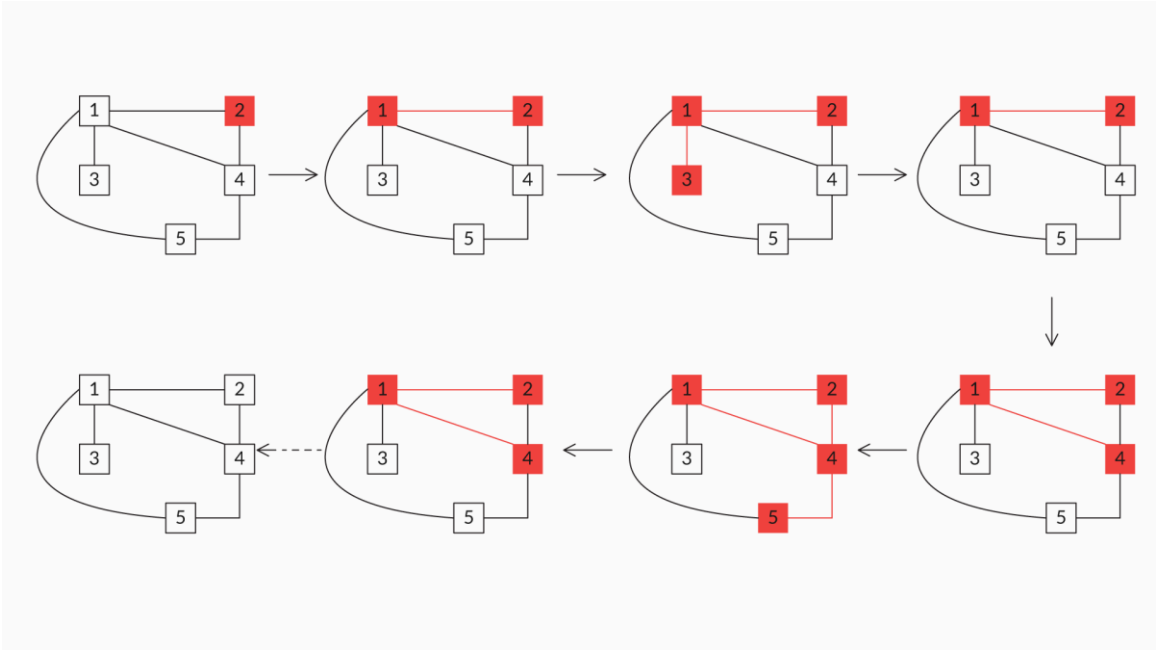
pl.get(potato) = returns the price of potatoes as 30
pl.put(onion, 25) = create a new entry
pl.put(potato, 35) = updates the corresponding entry in the price list

Depth-First Search - I

Depth-first search (DFS) is a traversal algorithm. From the start node, it traverses through any one of its neighbours and explores the farthest possible node in each branch before backtracking.

Backtracking happens when the search algorithm reaches a node where there are no neighbours to visit, or all the neighbours have already been visited. Then, the DFS algorithm traces back to the previous node and traverses any neighbour if it is left unvisited. Thus, backtracking helps to traverse through all the connected nodes in the graph and trace back to the start node.

The depth first search can be visualized in the following image, where the DFS starts from node 2 in the graph.



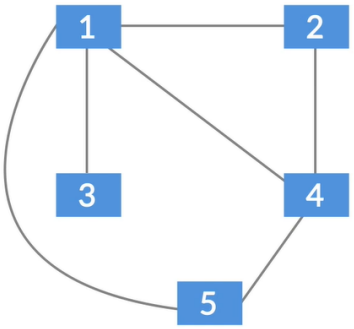
Now, you have the learnt the pseudo code for depth first search algorithm,

UpGrad

DEPTH FIRST SEARCH (DFS)

Pseudo code

```
Visited ← {}
Procedure dfs(n)
   add n to visited set
   for all n` ∈ neighbours(n) do
         if (n` ∉ visited) then
             dfs(n`)
         end if
   end for
end procedure
```



Then, professor has explained the pseudo code in detail using the same graph example.

The pseudocode for the depth-first search algorithm is as follows:

```
Procedure bfs(n)
  Q ← new Queue
  Visited ← { }
  enqueue (Q, n)
  Add n to visited set
  While Q is not empty
     n  ← dequeue(Q)
     for all n` ∈ neighbours(n)
        if (n` ∉ visited) then
           enqueue(Q, n`)
           add n` to visited set
        end if
     end for
  end while
end procedure
```

Step 1: The start node of the DFS algorithm is added to the visited set.

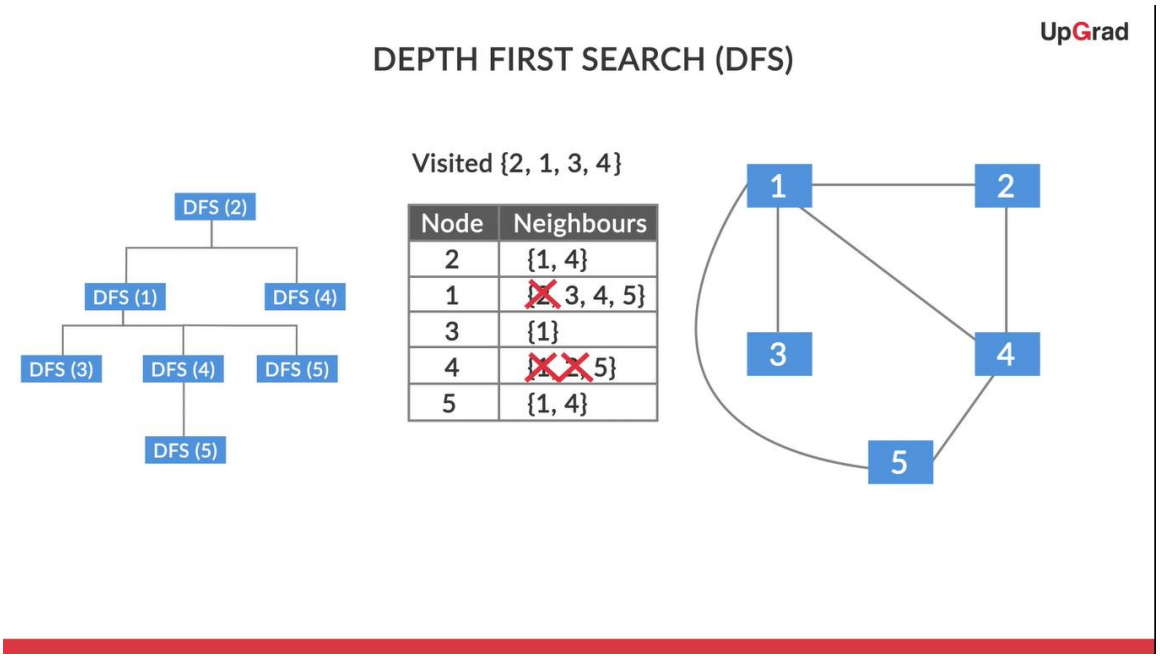
Step 2: The ‘for’ loop instruction set is executed for all the neighbours of the start node.

Step 3: Then if the neighbour node is unvisited, only then it recursively calls the dfs() method.

Step 4: Now, the unvisited neighbour node becomes the start node and repeats the above steps.

Step 5: The DFS traversal reaches a node from where there are no unvisited neighbour nodes; here, the recursive algorithm backtracks to the earlier traversed nodes and visits the remaining nodes.

Thus, the depth-first search recursively visits all the nodes along one branch and then backtracks to the unvisited neighbour nodes. Once all the nodes connected from the start node are visited, the algorithm ends.



In graphs, the visited nodes record must be maintained in order to avoid going in an infinite loop along the cycles present in the graph. This is the major difference between the DFS algorithm of graphs and trees, where you need not maintain the record of visited nodes.

Depth-First Search - II

Now, you have learnt the Java implementation of depth-first search on an application to record the order in which nodes are visited during depth-first search given the start node.

The professor has explained how DFS is implemented in the following code,

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class DFS1 {

    public static void main(String[] args) {
        MyGraph<Integer> graph = new AdjacencyList<>();

        Node<Integer> n1 = graph.addNode(1);
        Node<Integer> n2 = graph.addNode(2);
        Node<Integer> n3 = graph.addNode(3);
        Node<Integer> n4 = graph.addNode(4);
        Node<Integer> n5 = graph.addNode(5);

        try {
            graph.addEdge(n1, n2);
            graph.addEdge(n1, n3);
            graph.addEdge(n1, n4);
            graph.addEdge(n1, n5);
            graph.addEdge(n2, n4);
```

```
            graph.addEdge(n4, n5);

            Map<Node<Integer>, Integer> dfs_nums = DFS1.dfs(graph, n2);
            for (Node<Integer> n : dfs_nums.keySet()) {
                System.out.println(n + " : " + dfs_nums.get(n));
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static Map<Node<Integer>, Integer> dfs(MyGraph<Integer> graph, Node<Integer> node)
throws Exception {
        Map<Node<Integer>, Integer> dfs_nums = new HashMap<Node<Integer>, Integer>();
        DFS1.dfs_rec(graph, node, dfs_nums);
        return dfs_nums;
    }

    private static void dfs_rec(MyGraph<Integer> graph, Node<Integer> node, Map<Node<Integer>,
Integer> dfs_nums) throws Exception {
        if (dfs_nums.containsKey(node)) {
            return;
        }
        dfs_nums.put(node, dfs_nums.size());
        for (Node<Integer> neighbour : graph.getAllNeighbours(node)) {
            DFS1.dfs_rec(graph, neighbour, dfs_nums);
        }
    }
}
```

MyGraph - It is the interface which represents graph abstract data type

AdjacencyList - This is the data structure which implements graph ADT

AddNode () - This method helps to add nodes to the variable 'graph'

AddEdge (n1, n2) - This method helps to add edge between the nodes n1, n2

dfs_nums – This variable maps each node to the order in which they have been visited during depth-first search.

dfs () method takes two parameters:

- graph, on which traversal to be done
- node, from which traversal to begin

dfs_rec() method takes three parameters:

- graph, on which traversal to be done
- node, from which traversal to begin
- dfs_nums, maps the relation between order of visit and nodes

So far, you have understood the depth-first search algorithm and its implementation but in order to perform the traversal technique, you need to implement graph structure in Java. For which you have learnt three different classes to implement graph data structure.

- Node class
- Edge class
- MyGraph interface

Node class is a generic class that represents all the nodes present in a graph

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class Node<T> {
    private T value;

    public Node(T value) {
        this.value = value;
    }

    public String toString() {
        return this.value.toString();
    }

}
```

Edge class has two variables representing the nodes that are connected through an edge and all the edges in a simple undirected graph

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class Edge<T> {
    public final Node<T> n1;
    public final Node<T> n2;

    public Edge(Node<T> n1, Node<T> n2) {
        this.n1 = n1;
        this.n2 = n2;
    }

    public boolean equals(Edge<T> e) {
        return (this.n1.equals(e.n1) && this.n2.equals(n2)) ||
                (this.n2.equals(e.n1) && this.n1.equals(n2));
    }
}
```

MyGraph is an interface that contains the various methods such as

- getAllNodes() – This method helps to retrieve all the nodes present in graph
- addNode() – This method adds nodes to the graph
- addEdge() – This method adds an edge between two nodes
- getAllNeighbours() – This method helps to find all the neighbours of a given node

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public interface MyGraph<T> {

    Set<Node<T>> getAllNodes();

    Node<T> addNode(T e);

    /*
      throws exception when either of the nodes is not a member of the graph.
    */
    Edge<T> addEdge(Node<T> n1, Node<T> n2) throws Exception;

    /*
```

```
      throws exception when the node is not a member of the graph.
    */
    Set<Node<T>> getAllNeighbours(Node<T> node) throws Exception;
}
```

Depth-First Search - III

You have seen another application of depth- first search which is to find all the reachable nodes from a given node. In this application, professor has demonstrated how depth first search reaches to all the connected nodes in a graph.

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;

public class DFS2 {

    public static void main(String[] args) {
        MyGraph<Integer> graph = new AdjacencyList<Integer>();

        Node<Integer> n1 = graph.addNode(1);
        Node<Integer> n2 = graph.addNode(2);
        Node<Integer> n3 = graph.addNode(3);
        Node<Integer> n4 = graph.addNode(4);
        Node<Integer> n5 = graph.addNode(5);
        Node<Integer> n6 = graph.addNode(6);

        try {
            graph.addEdge(n1, n2);
            graph.addEdge(n1, n3);
            graph.addEdge(n1, n4);
            graph.addEdge(n1, n5);
            graph.addEdge(n2, n4);
            graph.addEdge(n3, n4);
            graph.addEdge(n3, n5);
            graph.addEdge(n4, n5);
            graph.addEdge(n6, n5);
            Set<Node<Integer>> reachable = DFS2.dfs(graph, n2);
            for (Node<Integer> n : reachable) {
                System.out.println(n);
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static Set<Node<Integer>> dfs(MyGraph<Integer> graph, Node<Integer> n) throws
Exception {
        Set<Node<Integer>> reachable = new HashSet<Node<Integer>>();
        DFS2.dfs_rec(graph, n, reachable);
        return reachable;
    }

    private static void dfs_rec(MyGraph<Integer> graph, Node<Integer> node, Set<Node<Integer>>
reachable) throws Exception {
        if (reachable.contains(node)) {
            return;
        }
        reachable.add(node);
        for (Node<Integer> neighbour : graph.getAllNeighbours(node)) {
            DFS2.dfs_rec(graph, neighbour, reachable);
        }
    }
}
```

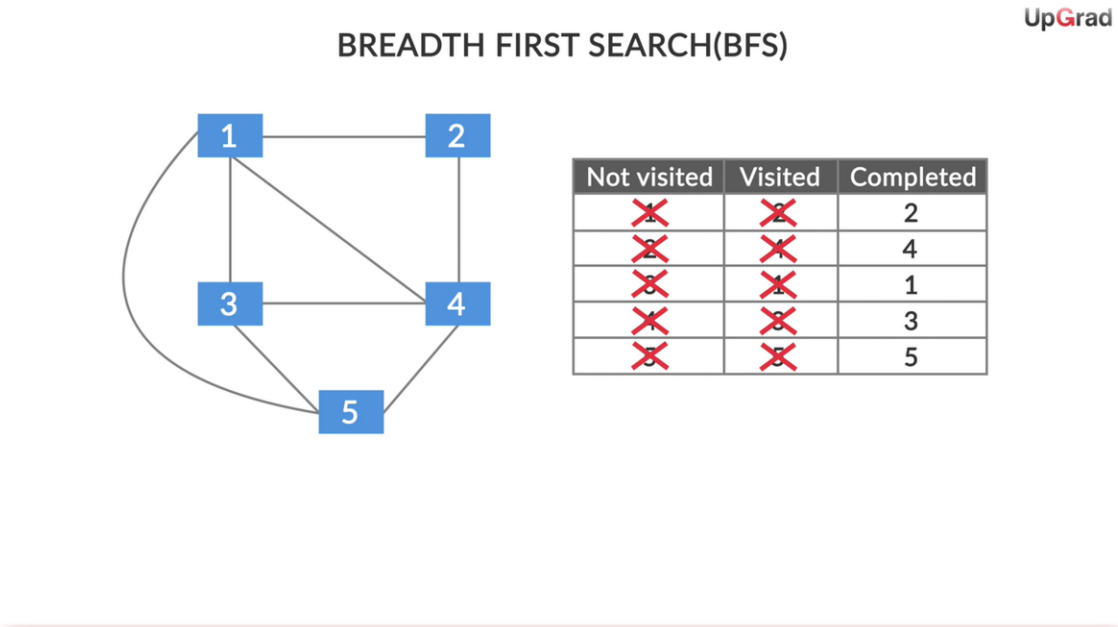
Breadth-First Search - I

Breadth-first search is a traversing algorithm where traversing starts from the start node and then explores the immediate neighbours of the start node; then the traversing moves towards the next-level neighbours of the graph structure. As the name suggests, the traversal across the graph happens breadthwise.

To implement the breadth-first search, you need to consider the stage of each node. Nodes, in general, are considered to be in three different stages as,

- Not visited
- Visited
- Completed

In BFS algorithm, nodes are marked as visited during the traversal, in order to avoid the infinite loops caused because of the possibilities of cycles in a graph structure.



Breadth-First Search - II

Having understood the breadth-first search traversal technique, professor has explained the pseudo code of breadth-first search algorithm.The pseudocode is as follows:

```
Procedure bfs(n)
  Q ← new Queue
  Visited ← { }
  enqueue (Q, n)
  Add n to visited set
  While Q is not empty
     n  ← dequeue(Q)
     for all n` ∈ neighbours(n)
        if (n` ∉ visited) then
           enqueue(Q, n`)
           add n` to visited set
        end if
     end for
  end while
end procedure
```

Step 1: The start node is enqueued and also marked as visited in the following set of instructions,

- enqueue (Q, n)
- Add n to visited set

Step 2: 'While' loop instruction set is executed when the queue is not empty

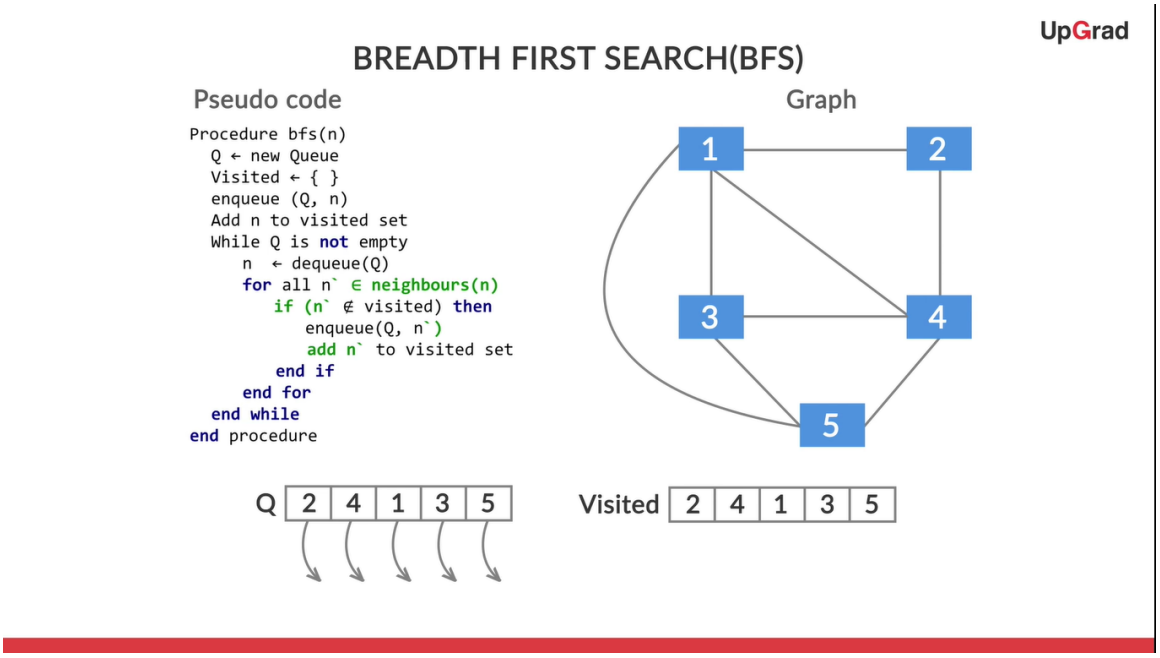
Step 3: For each iteration of while loop, a node gets dequeued

Step 4: Now 'for' loop runs till all the unvisited neighbours of the dequeued node(n) are enqueued and also marked as visited.

Step 5: For the first iteration of the while loop, all the neighbour nodes of the start node are enqueued and on the second iteration, all the next level unvisited neighbour nodes of one of the neighbour node are enqueued.

This way, all the neighbour nodes are enqueued and visited level wise from the start node and after a certain number of iterations, all the nodes are dequeued and the algorithm ends.

Then professor has explained the pseudo code of breadth-first search algorithm on the graph example as follows,



Now, breadth-first search algorithm is implemented in Java to find the different levels of nodes based on breadth-first traversal of graph from a given start node.

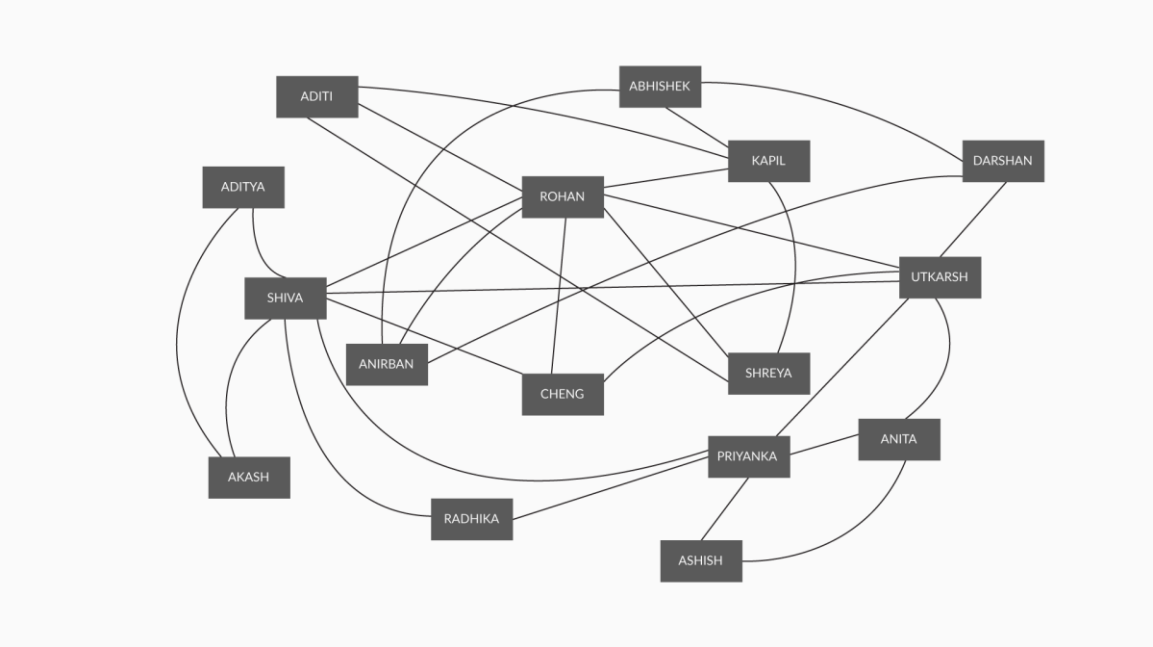
Professor has explained the implementation of BFS in the following code,

```
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;
import java.util.Queue;
import java.util.LinkedList;

public class BFS {

    public static void main(String[] args) {
        MyGraph<Integer> graph = new AdjacencyList<Integer>();
```

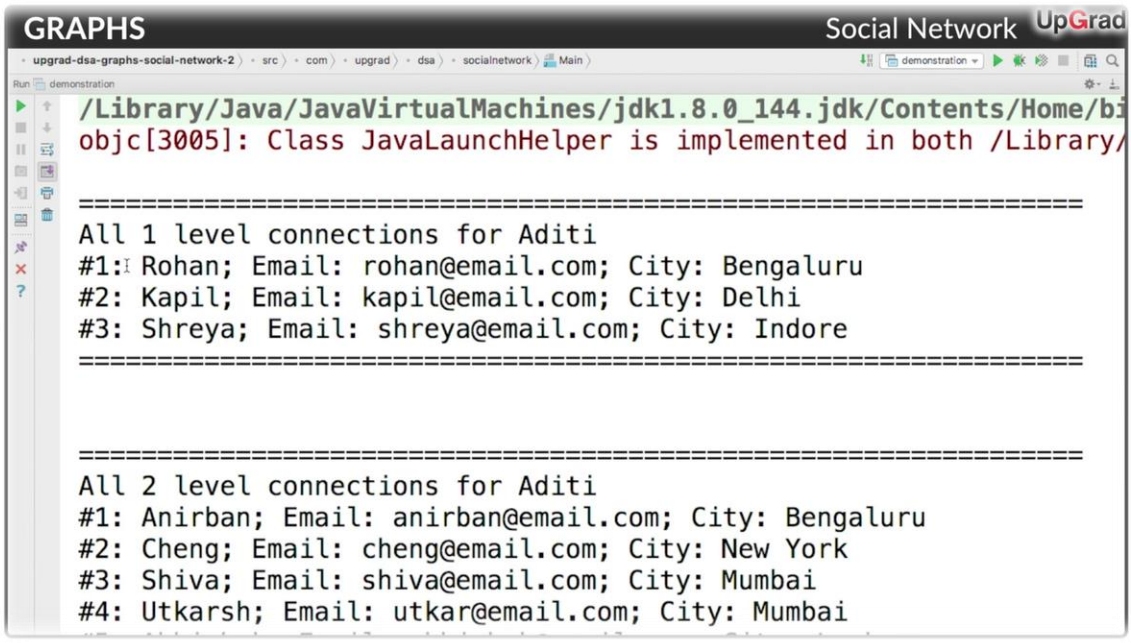
```
        Node<Integer> n1 = graph.addNode(1);
        Node<Integer> n2 = graph.addNode(2);
        Node<Integer> n3 = graph.addNode(3);
        Node<Integer> n4 = graph.addNode(4);
        Node<Integer> n5 = graph.addNode(5);

        try {
            graph.addEdge(n1, n2);
            graph.addEdge(n1, n3);
            graph.addEdge(n1, n4);
            graph.addEdge(n1, n5);
            graph.addEdge(n2, n4);
            graph.addEdge(n3, n4);
            graph.addEdge(n3, n5);
            graph.addEdge(n4, n5);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        Map<Node<Integer>, Integer> levels;
        try {
            levels = BFS.bfs(graph, n2);
            for (Node<Integer> n : levels.keySet()) {
                System.out.println(n + " : " + levels.get(n));
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }

    }

    public static Map<Node<Integer>, Integer> bfs(MyGraph<Integer> graph, Node<Integer> n)
throws Exception {
        Queue<Node<Integer>> queue = new LinkedList<Node<Integer>>();
        Map<Node<Integer>, Integer> levels = new HashMap<Node<Integer>, Integer>();
        queue.add(n);
        levels.put(n, 0);
        while (queue.isEmpty() == false) {
            Node<Integer> nextNode = queue.remove();
            Set<Node<Integer>> neighbours = graph.getAllNeighbours(nextNode);
            for (Node<Integer> neighbour : neighbours) {
                if (levels.containsKey(neighbour) == false) {
                    queue.add(neighbour);
                    levels.put(neighbour, levels.get(nextNode) + 1);
                }
            }
        }
        return levels;
    }
}
```

Industry Demonstration - I

The industry relevance of the two traversal techniques with respect to the graphs has been discussed using a real-world example on social networks such as Facebook, LinkedIn. In order to perform the applications of traversal techniques, we have considered a small set of people and connections between them, as shown in the following graph

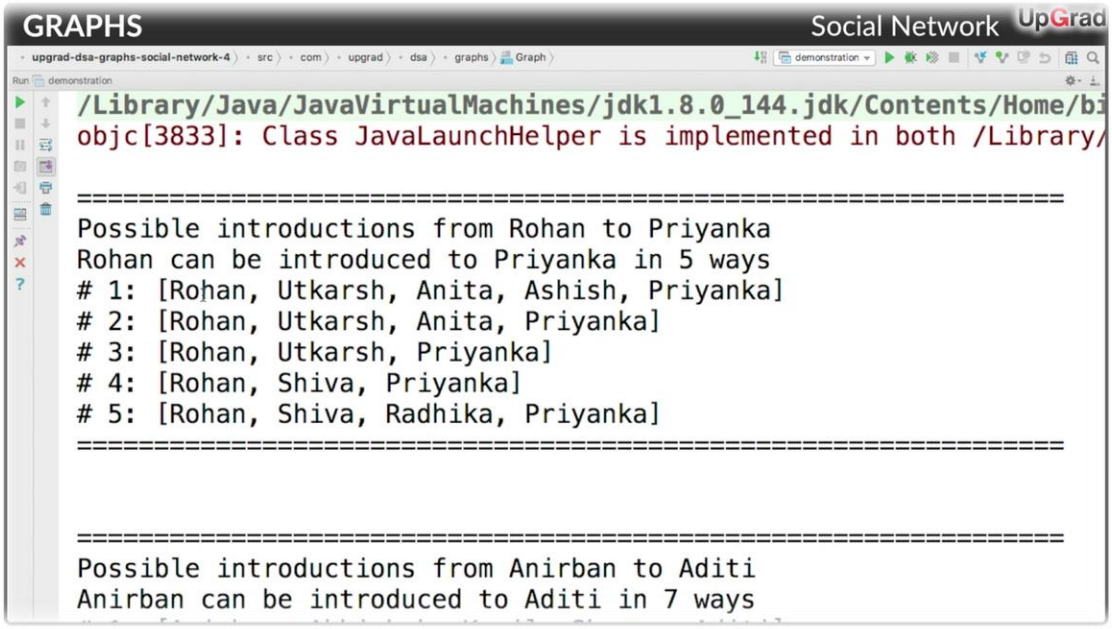


The first application discussed on the social network is to find the different level of connections of a person in the social network, you have learnt to implement the application in Java using breadth-first search algorithm to find different level of connections of a person in the social network

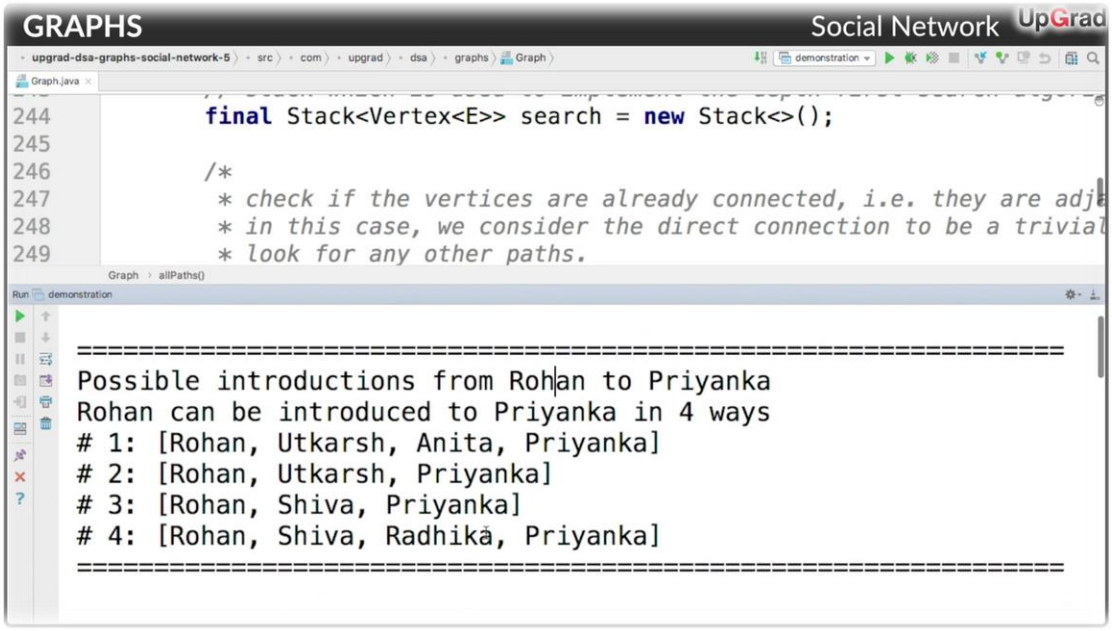


Industry Demonstration - II

In the second real-world application, depth first search algorithm is implemented to find how a person can be introduced to another person along the path connected with other people in social network. You have also learnt the java implementation of the application and the output is as follows,



On the same implementation of DFS, industry expert has introduced depth limit such that only 1st level, 2nd level, 3rd level connections can only be introduced to a person. In which case the output is as follows,



Summary

In this session, you have learnt

- What is a graph ADT?
- Different types of graphs
  - Undirected graph
  - Directed graph
    - Directed acyclic graph

- Differences between graphs and trees
- Depth-first search
  - Pseudocode
  - Applications
    - Compute the order of visit of all nodes in DFS traversal
    - Compute the set of reachable nodes from a given start node
    - Compute the different ways of introducing one person to another in a social network

- Breadth-First search
  - Pseudocode
  - Applications
    - Compute the level of each node in BFS traversal
    - Compute the nth level connections in a social network

Session 2: Graph Algorithms

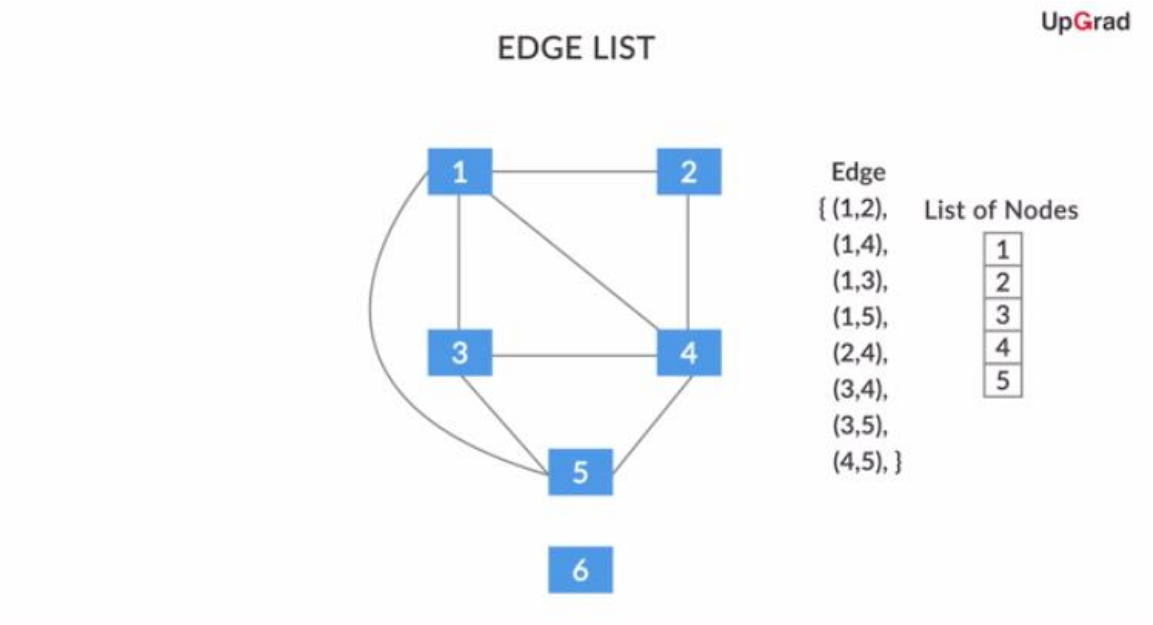
Let's revisit the topics learnt in Welcome to the session on Graph Algorithms.

In this session we gave you a brief introduction on what are the things you will be learning in this session on Graph Algorithm.

You were told that you will be learning about the following-

- Edge list
- Adjacency matrix
- Adjacency list
- Dijkstra's algorithm
- How to apply all of the above in real life

Introduction to Edge Lists



In this segment you learnt about the first graph ADT implementation i.e. an edge list and how it is implemented using the MyGraph interface.

This is the graph which we have been working with. There are these edges between -

- 1 and 2
- 1 and 4
- 3 and 4 and so on
- So all we did was to maintain a list of these edges. So the edge list class would have an attribute named edges, and it's just a list of all the edges, and in this case it suffices to represent the edges as pairs of the end point nodes.

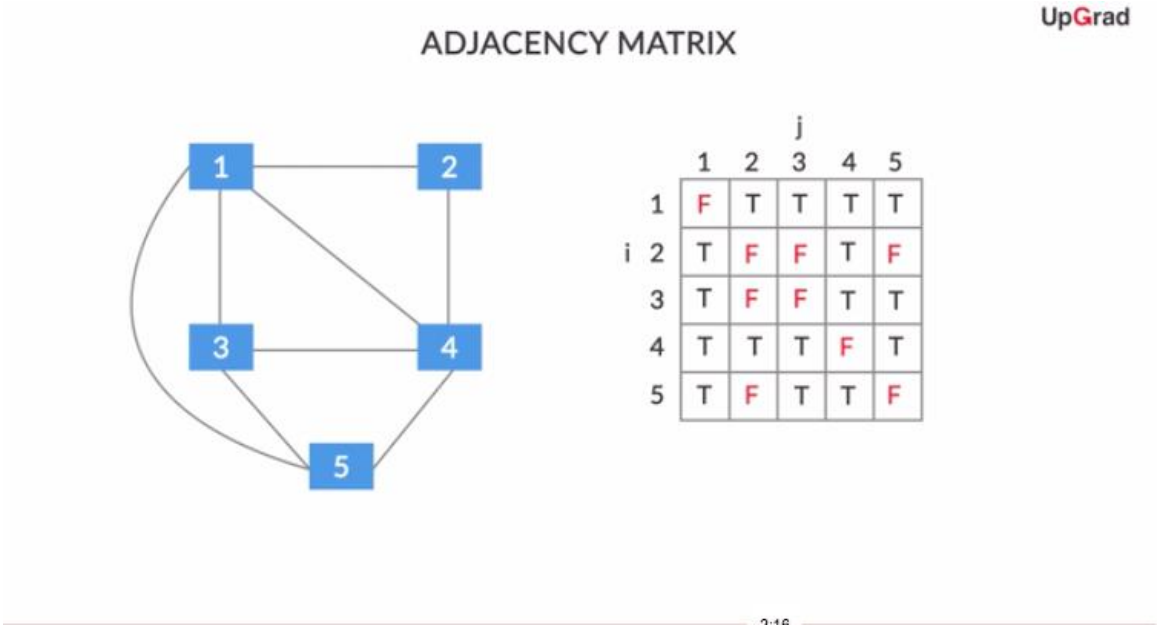
For example, the edge between 1 and 2 is merely a pair of the two nodes.
You also learnt that an edge list fails in a scenario when you have an isolated edge in your graph.

Introduction to Adjacency Matrix

In this segment you were introduced to another graph implementation: the 'adjacency matrix'. As the name suggests, an adjacency matrix is a two-dimensional boolean matrix that represents a connection between nodes.

The two-dimensional matrix is a Boolean matrix. Here, the number of rows and the number of columns are the same. It's a square matrix and that is equal to the number of nodes in the graph. Each row and each column corresponds to a particular node.

If you want to represent that there exists an edge between the node I and node J then you would place a true value in the cell corresponding to the Ith row and Jth column.



In this segment you saw how you can represent an adjacency matrix using a two-dimensional matrix of boolean values.

Adjacency Matrix Implementation

In this segment you learnt about the MyGraph interface, which has four functions, namely

- getAllNodes
- addNode
- addEdge
- getAllNeighbors

Let's recap what these functions do, in detail-

- **getAllNodes**: This function converts a list of nodes into a set of nodes.
- **add Node** This function returns a newly created node.
- **addEdge** method returns a newly created edge between node n1 and n2. So in order to find the row and the column corresponding to the nodes n1 and n2, we have to find out what is their position in the node list. And that we do by using the Index Of operation.
- **getallNeighbors** method returns the set of nodes which are neighbors to the node known, and its implementation also happens to be fairly simple. All you have to do is to find out the row which corresponds to the node that we are interested in.

To recap the code for the four methods were as follows: -

- **getAllNodes**

```
public Set<Node<T>>getAllNodes(){
    Set<Node<T>>set=new HashSet<Node<T>>();
    for(Node n:this.nodes){
        set.add(n);
    }
    return set;
}
```

- **addNode**

```
public Node<T> addNode(T e){
    Node<T> newNode=new Node<T>(e);
    this.nodes.add(newNode);
    boolean[][]m=new boolean[this.nodes.size()][this.nodes.size()];
    for(int i=0;i<m.length;i++){
        for(int j=0;j<m.length;j++){
            m[i][j]=false;
        }
    }
    for(int i=0;i< this.adjacencyMatrix.length;i++){
        for(int j=0;j< this.adjacencyMatrix.length;j++){
            m[i][j]=this.adjacencyMatrix[i][j];
        }
    }
    this.adjacencyMatrix=m;
    return newNode;
}
```

- **addEdge**

```
public Set<Node<T>> getAllNeighbours(Node<T> node) throws Exception {
    if(this.nodes.contains(node) == false) {
        throw new Exception("node not contained in this graph.");
    }
    Set<Node<T>> neighbours = new HashSet<Node<T>>();
    int row = this.nodes.indexOf(node);
    for(int i = 0; i < this.adjacencyMatrix.length; i++) {
        if(this.adjacencyMatrix[row][i] == true) {
            neighbours.add(this.nodes.get(i));
        }
    }
    return neighbours;
}
```

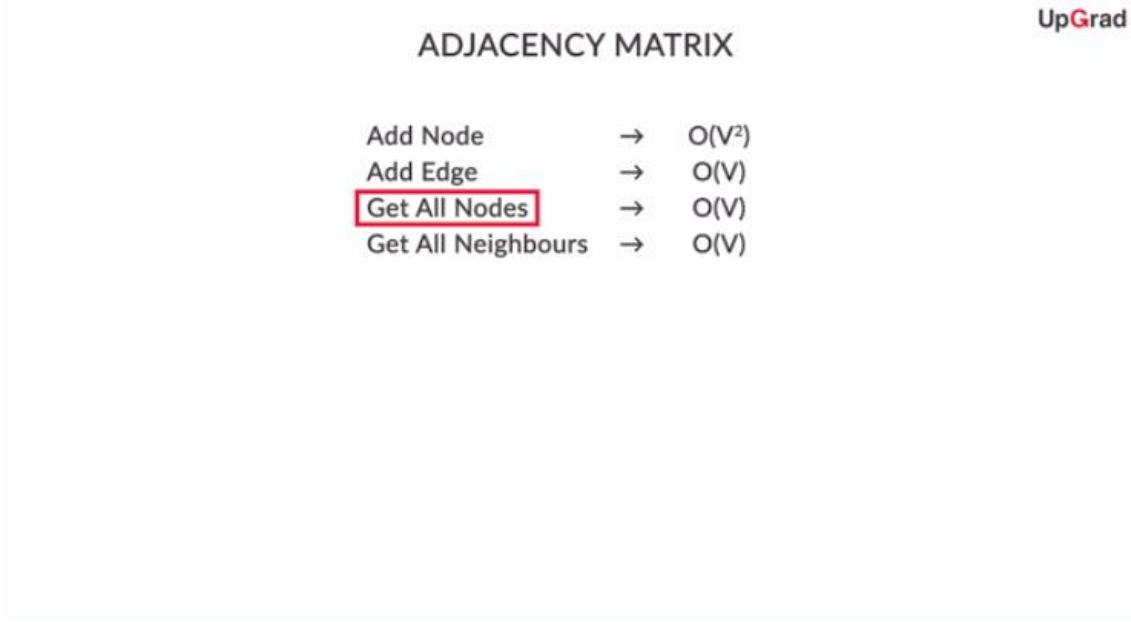
- **getAllNeighbors**

```
public Set<Node<T>>getAllNeighbours(Node<T> node)throws Exception{
    if(this.nodes.contains(node)==false){
        throw new Exception("node not contained in this graph.");
    }
    Set<Node<T>>neighbours=new HashSet<Node<T>>();
    int row=this.nodes.indexOf(node);
    for(int i=0;i< this.adjacencyMatrix.length;i++){
        if(this.adjacencyMatrix[row][i]==true){
            neighbours.add(this.nodes.get(i));
        }
    }
    return neighbours;
}
```

Performance Characteristics of Adjacency Matrix

The most important aspect of any algorithm is its performance characteristics, which determine how it will perform in a given situation or circumstance. Therefore, you need to take into account an algorithm's performance when you make your choices.

In this segment you calculated the time complexity for an adjacency matrix implementation, specifically the time complexity for the following four methods: addNode, addEdge, getAllNodes, and getAllNeighbours. You also explored the performance characteristics of these methods.

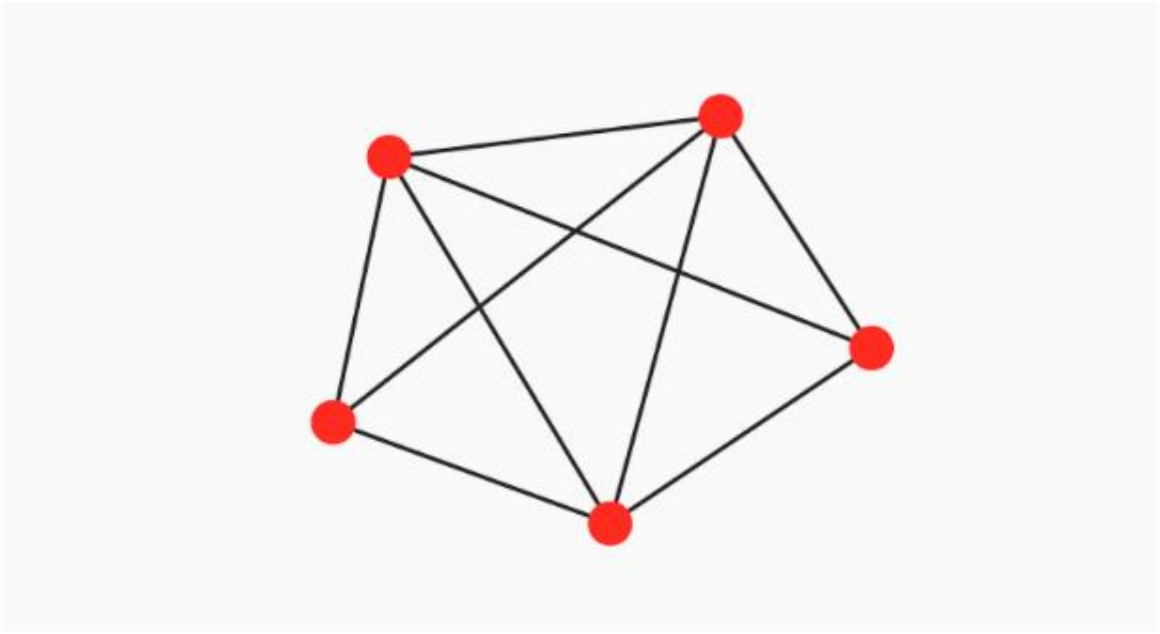


In this segment you learnt

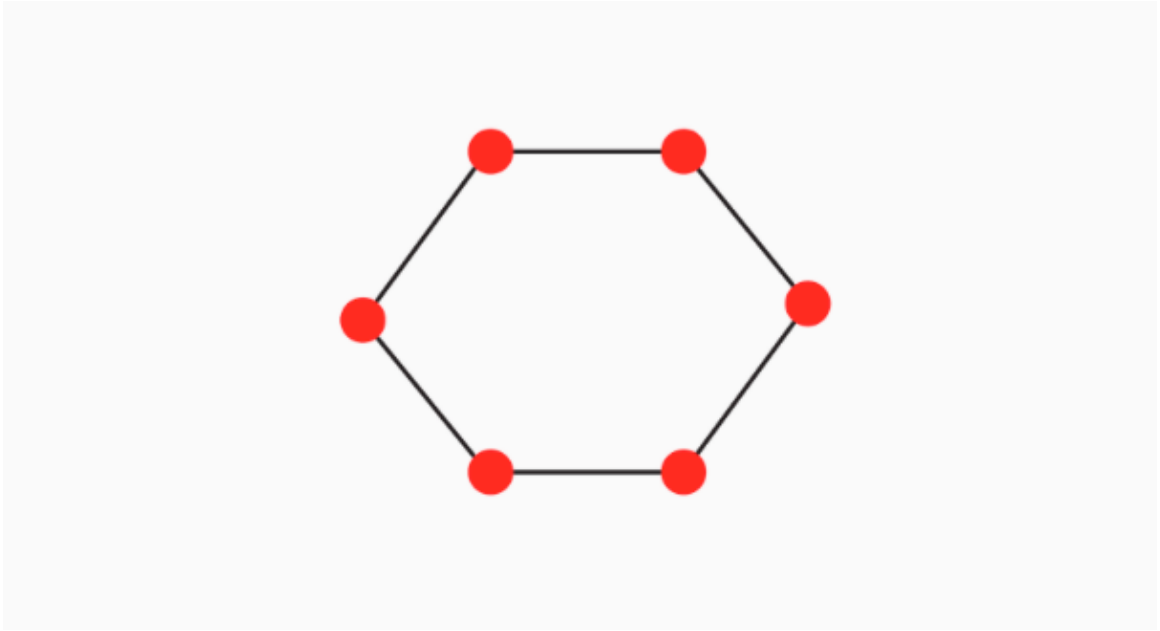
- How to calculate the time complexity of each of the methods in an adjacency matrix implementation
- The time complexities for the methods turn out to be
- For addNode method - O(V * V)
- For addEdge method - O(V)
- For getAllNodes method - O(V)
- For getAllNeighbors - O(V)
- The practices you can follow to improve the time complexities of the following methods: addEdge and getAllNeighbors

You also learnt about 'dense graphs' and 'sparse graphs' in detail.

Dense graphs: Dense graphs are densely connected, which means they have the maximum number of edges between nodes, i.e. there is an edge from each node to every other node. Given below is an example of a dense graph.



Sparse graphs: Sparse graphs are connected graphs with the minimum or a small number of edges connecting nodes. In sparse graphs, there may or may not be an edge between two nodes. Here, usually, the number of edges is n, which is also the number of vertices.



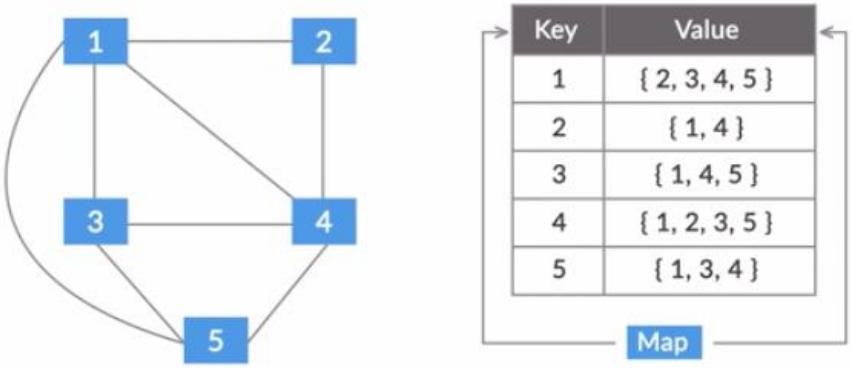
Introduction to and Implementation of Adjacency Lists

In this segment, you learnt about our third and last graph ADT implementation: the 'Adjacency list'. You learnt the following:

- The purpose behind why we came up with this third method
- The data structure that you need to use for an adjacency list implementation (you will also see the Java code for the same)
- How to distinguish when you should choose an adjacency list over the other two implementations

UpGrad

ADJACENCY LIST



- At the end of this segment, you know how this list is implemented and how it is different from an edge list and an adjacency matrix. You also saw the Java implementation of an adjacency list by using the MyGraph interface. After attempting the questions below, you are sure to get an even clearer idea of all of this.

Performance Characteristics

So far you learnt about, edge lists, adjacency matrix, and adjacency list. You explored the data structures required for each of these implementations. You even worked out the performance characteristics of an adjacency matrix in great detail. Now, it's time to compare these three graph ADT representations in terms of their performance characteristics.

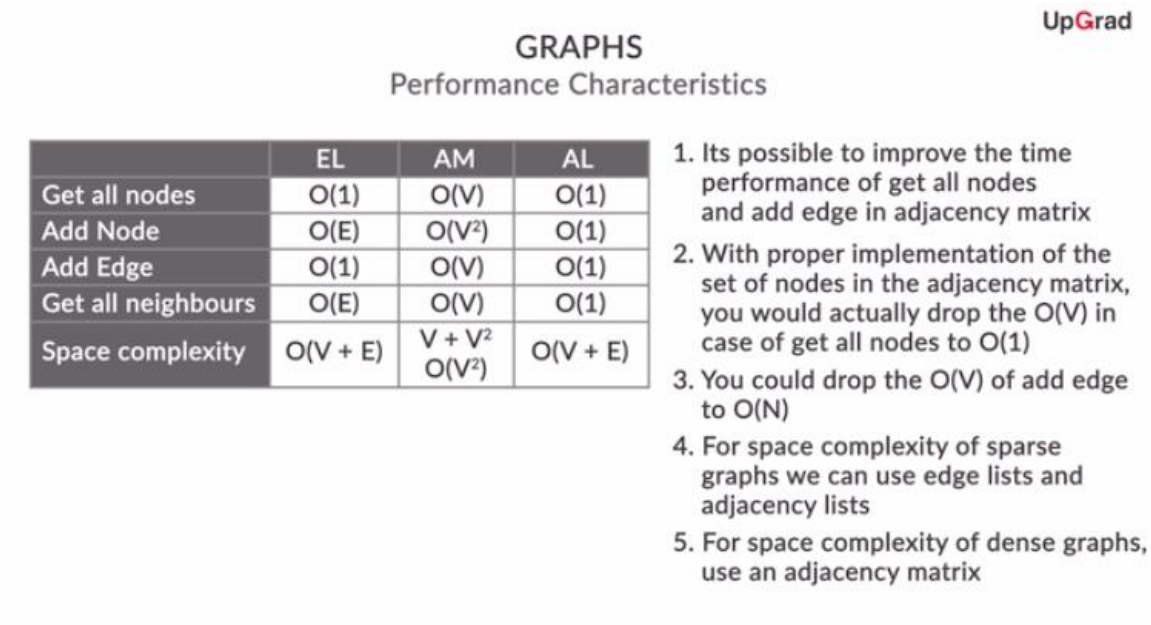
UpGrad

GRAPHS

Performance Characteristics

	EL	AM	AL
Get all nodes	O(1)	O(V)	O(1)
Add Node	O(E)	$O(V^2)$	O(1)
Add Edge	O(1)	O(V)	O(1)
Get all neighbours	O(E)	O(V)	O(1)
Space complexity	O(V + E)	$V + V^2$ $O(V^2)$	O(V + E)

In this segment you saw situations in which particular algorithms do not perform as well as others; you will also had a look at the improvements (if any) that can be made to the performance characteristics of the graph ADT implementations.



To sum up everything from this segment, you made a detailed comparison between the three graph implementations. So, the video in this segment was basically about the core of all the implementations you learnt about, where you explored the performance characteristics of each implementation.

At the end adjacency list was concluded as behaving the most optimally, but, again, this depends on your requirements.

Introduction to Dijkstra's Algorithm

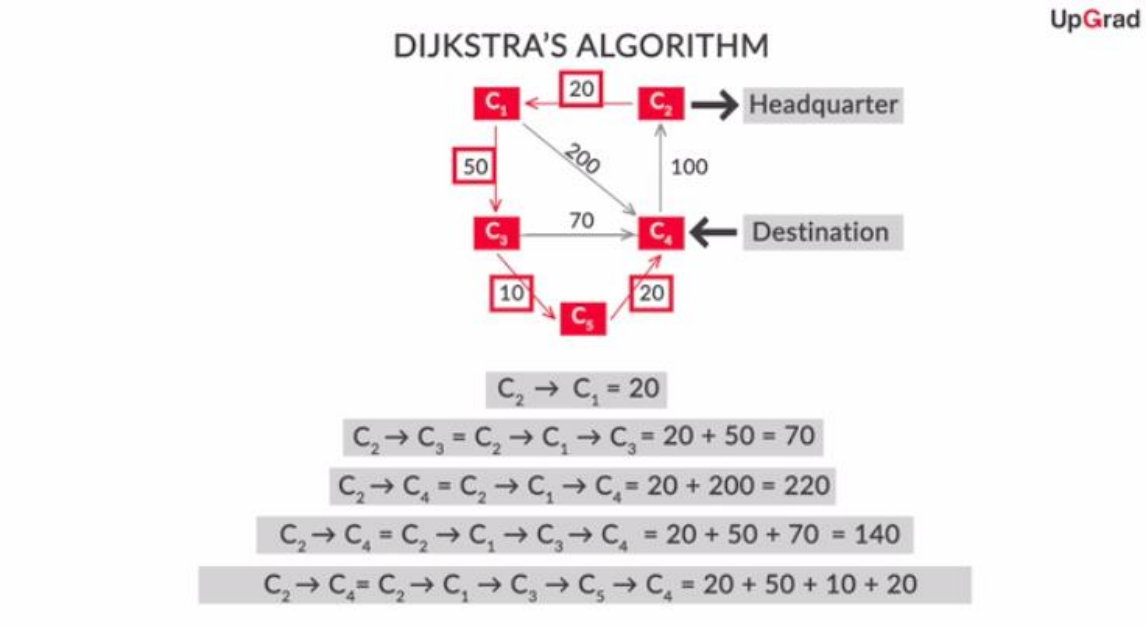
In this segment you were introduced to the famous Graph Algorithm i.e. Dijkstra's Algorithm .

This algorithm was named after Edsker Dijkstra who was the inventor of this algorithm.

- What does Dijkstra's shortest path algorithm do?
    - Given a graph g and a node in that graph n, it computes the shortest distances from n to all the other nodes of that graph.
- Dijkstra's algorithm falls in the broad category of greedy algorithms.
- Dijkstra's algorithm will work only on graphs with positive or non-negative edge weights

Dijkstra's shortest path algorithm has a wide variety of applications.

- We introduced you to one of them, considering that you are a logistic company which supplies various products or various parcels to a number of other places or other ports.
- you would be interested to send your parcels to various ports through the path which cost you the least and therefore you would be interested to find out the paths which-have the least cost for each of the other ports to which you want to send your parcels.
- Assuming C2 to be the headquarter we wanted to send the parcel from C2 to all the other ports such that it would cost us the least to reach each destination from C2.



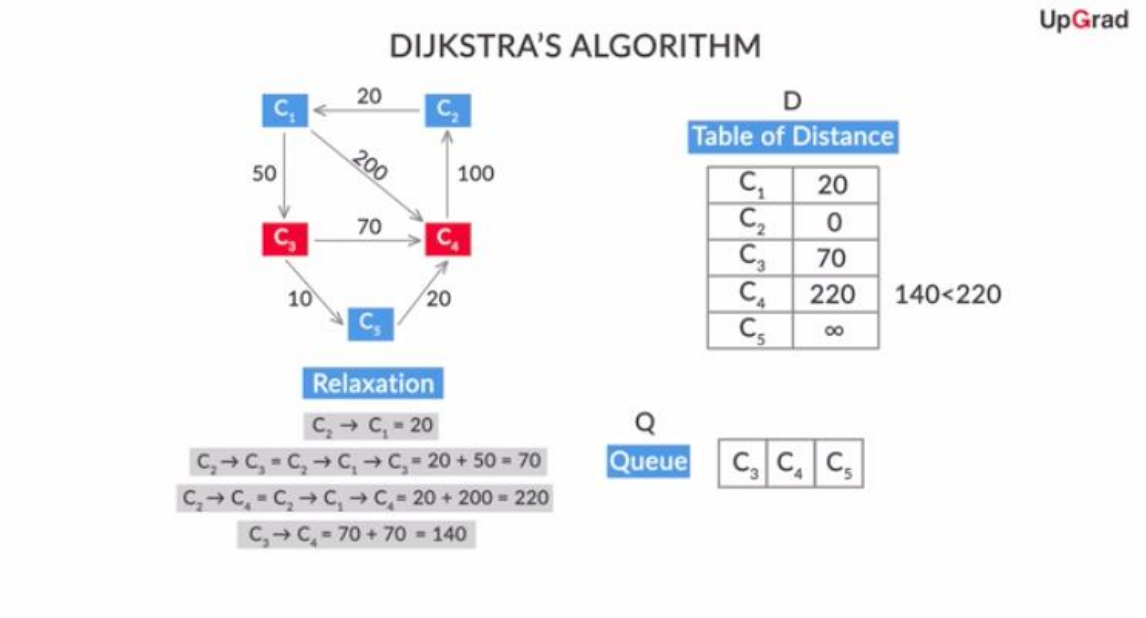
So, to sum up in this segment—

- You learnt about a new algorithm, Dijkstra's algorithm, which calculates the shortest path from a source to a destination.

Dijkstra's Algorithm

In this segment, you saw the following:

- Maintaining two data structures
  - Table of distance which was an array
  - Priority Queue
- How to make updates in the table of distance, including conditional updates
- When the priority queue comes into the picture and when you use the dequeue and enqueue operations on this priority queue



So, to sum up in this segment, you learnt the following:

- The concept of edge relaxation or relaxation
- How to make conditional updates in the table of distances, i.e. how relaxation works
- How you reach your final answer at the end of the while loop

You were also given the pseudocode of the Dijkstra's algorithm is as follows:

```
Procedure Dijkstra(graph, node)
  While Q is not empty
    Dequeue the first element from priority queue
    nextNode ← front element of queue after previous deque
    if nextNode is not null
      Relax the edges if necessary
    end if
  end while
end procedure
```

Step 1: 'While' loop instruction set is executed when the queue is not empty

Step 2: For each iteration of while loop, the node in the front gets dequeued

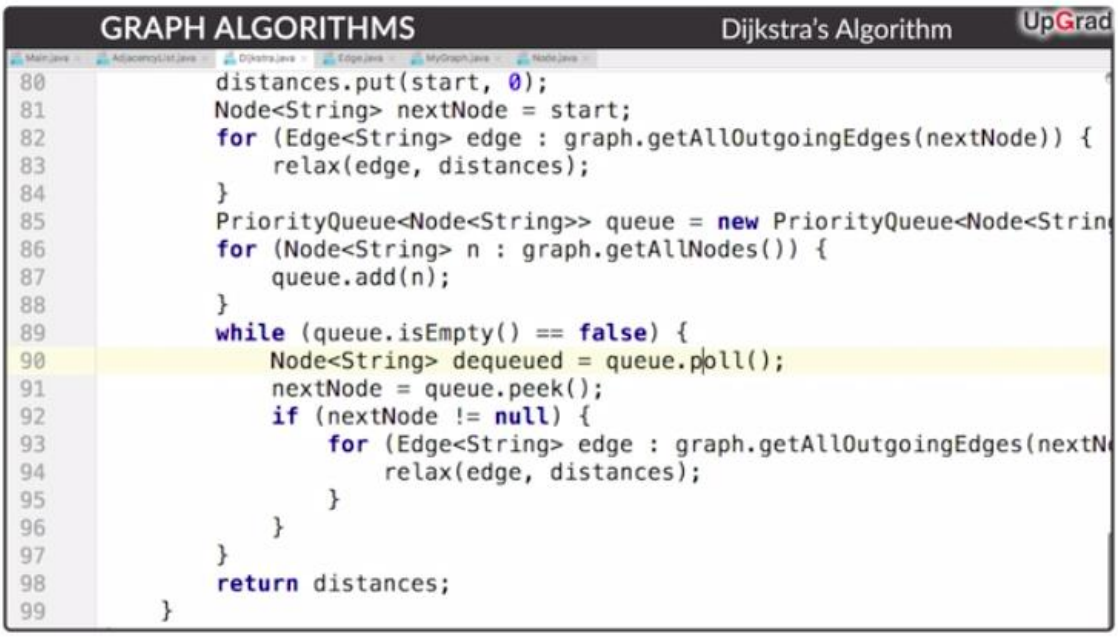
Step 3: nextNode will store the node in front of priority queue after the last deque

Step 4: We will check if the value of nextNode is null or not, if It is not null we will proceed and do edge relaxation on the nodes where require.

This way, we will end up getting the shortest distances from the given node to all the other nodes in our Table of Distance.

Dijkstra's Implementation

This segment we saw how Dijkstra's algorithm helps us in making updates to the distances in the Table of distance and the priority queue using the concept of Edge relaxation. The pseudocode that we saw earlier we saw it working here through actual Java code.




```
80          distances.put(start, 0);
81          Node<String> nextNode = start;
82          for (Edge<String> edge : graph.getAllOutgoingEdges(nextNode)) {
83              relax(edge, distances);
84          }
85          PriorityQueue<Node<String>> queue = new PriorityQueue<Node<Strin
86          for (Node<String> n : graph.getAllNodes()) {
87              queue.add(n);
88          }
89          while (queue.isEmpty() == false) {
90              Node<String> dequeued = queue.poll();
91              nextNode = queue.peek();
92              if (nextNode != null) {
93                  for (Edge<String> edge : graph.getAllOutgoingEdges(nextN
94                      relax(edge, distances);
95                  }
96              }
97          }
98          return distances;
99      }
```

The main logic for dijkstra's goes inside the While loop which looked like this-

```
while (queue.isEmpty() == false) {
    Node<String> dequeued = queue.poll();
    nextNode = queue.peek();
    if (nextNode != null) {
        for (Edge<String> edge : graph.getAllOutgoingEdges(nextNode)) {
            relax(edge, distances);
        }
    }
}
return distances;
}
```

In the end of this segment we saw that our java code was working fine and the values obtained in our Table of Distances was exactly the same as we had seen before.

Lecture Notes

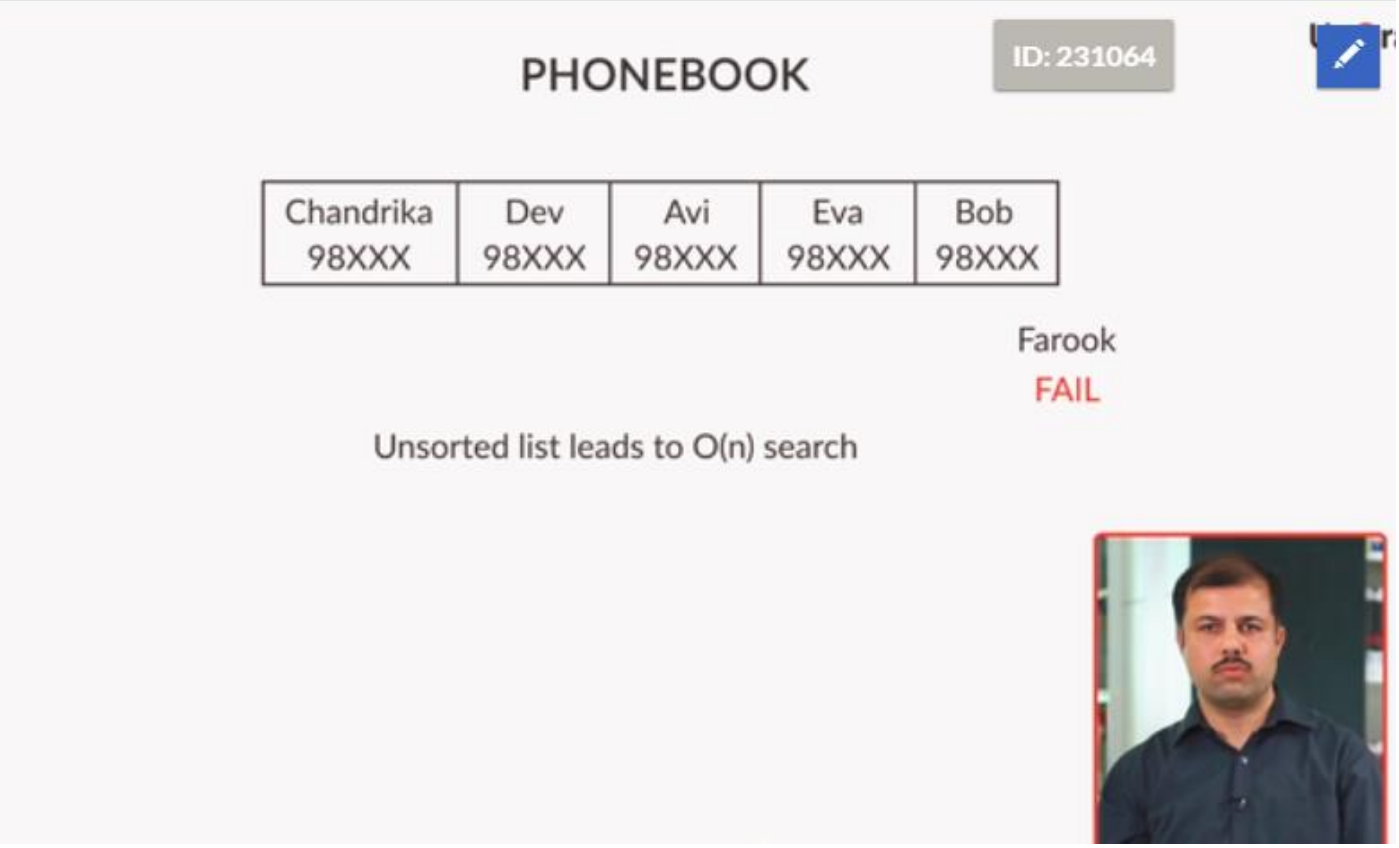
Hash Tables

Let's revisit the topics learnt in the session on Hash Tables.

In the Searching and Sorting module, you implemented a phone book using an array of names, which, in the worst case, took linear time to search for a name in the phone book when the names were unsorted. However, it would be a better move if you store the names in a sorted manner, thus reducing the overall time complexity to O(log n) by applying binary search functionalities to the phone book. But is this what you desire to achieve in the case of a phone book? No, you want fast retrieval, something in the order of O(1). In this summary, you will get to grips with the data structure known as Hashtable, which retrieves results in O(1).

Array Implementation of a Phone book

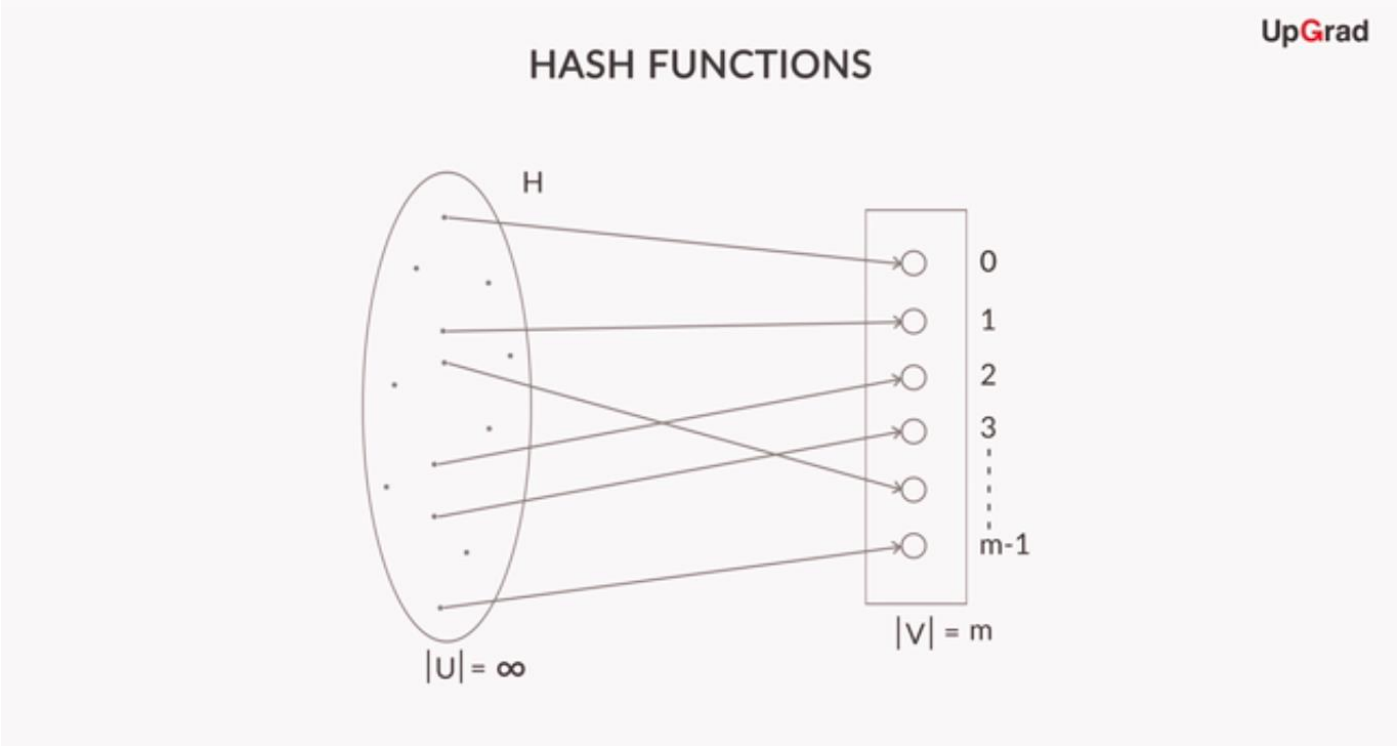
You saw in the video that searching for a name in a phone book implemented using an array required you to traverse the entire list in the worst case. This leads to a time complexity of O(n).



However, if you are cautious to sort the names in the phone book, you can apply the method of binary search, which will lead to an overall decline in the time complexity to O(log n).

Hashing and Hash Functions

The idea of hashing is based on the central concept of the **hash function**. A hash function is any function that can be used for mapping arbitrarily sized data to fixed size data. As you can see in the figure below, the cardinality of the input set is infinite, whereas the output is a range with a fixed size M, where 0<M<=N (N is the set of natural numbers).



The basic idea behind a hash function is to map the input domain, which could be any generic data type, into the output range, which in most cases is a natural number that can be realised in the program memory.

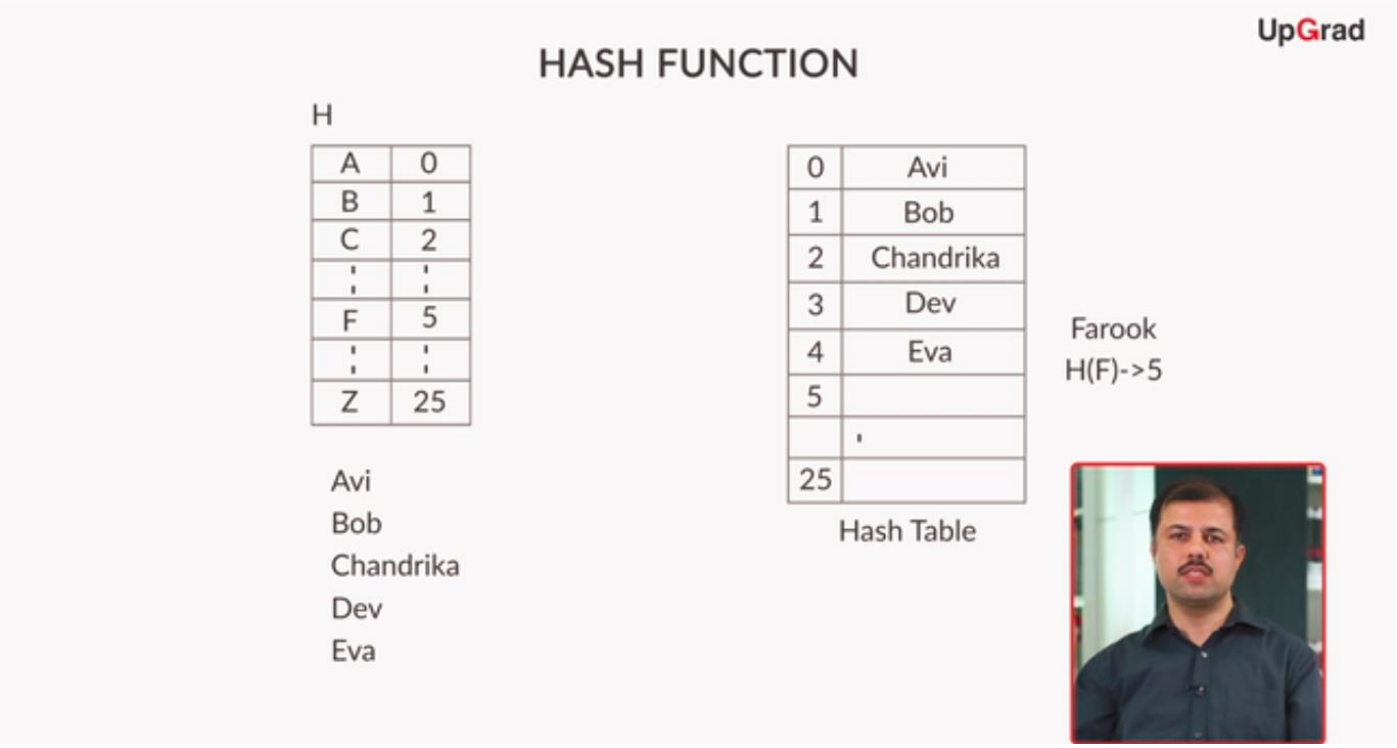
An important point to consider here is that the hash function should be fast to compute, or else, the whole objective of having a constant time to search or add won't come to pass if you have a hash function that itself is computationally expensive.

So, let's take a look at a hash function that takes up the first alphabet of an individual's name, and then based on the position of the letter in the English alphabet, it maps the name to the corresponding index in the hash table.

In the example below, you can see that the name 'Avi' starts with an 'A' and since A is the first letter of the English alphabet, it gets mapped to the first index of the hash table, i.e. 0. Similarly, 'Bob' gets mapped to the hash index 1, and so on.

Now, when a new entry, let's say 'Farook', comes in for search, you can directly apply the hash function to 'Farook'. Here, see that the first letter of the name is 'F', which means H(F) -> 5, so we can directly go to index 5 of the hash table and see whether the entry 'Farook' is there in the hash table or not. This search for the name 'Farook' by applying the hash function and directly checking the hash index 5 would take a total time of O(1), i.e. constant time.

This way, the use of hashing and hash functions can help you realise the constant time complexity, which is desired in the case of the phone book.



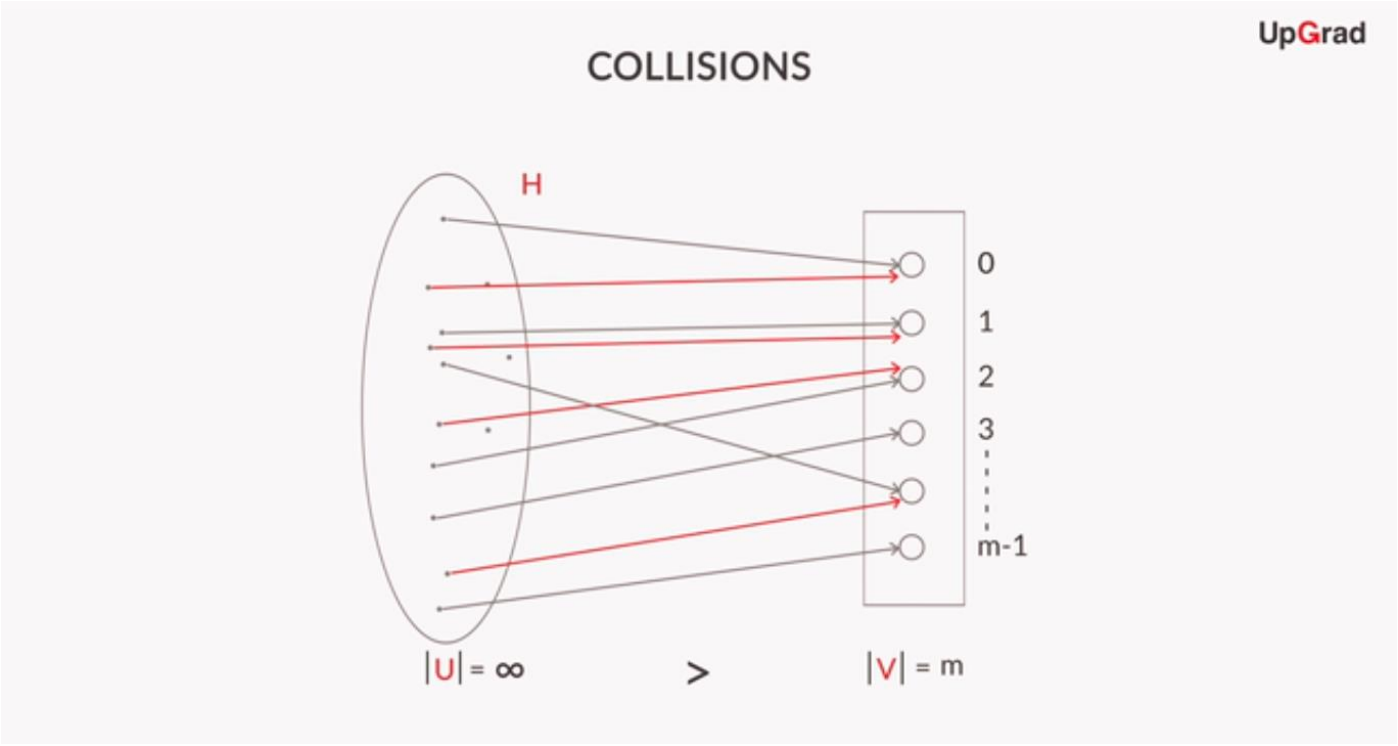
Let's say that you're working with some mathematical operations such as division, and modulus, which play a great role in many hash functions. For instance, the hash function H = i%10 would map any integer in the range of 0-9 depending on the remainder you get on dividing 'i' by 10. If i = 23, then H = 3, since 23 divided by 10 would give you a remainder of 3.

Collisions in Hash Tables

Let's understand a major limitation of hash tables that was overlooked in the previous segment, collision.

Let's say that you maintained a phone book with a hash function that takes the first character of each name to decide the hash key. What would happen if there were two names with the same first letter?

As discussed, this hash function maps an input domain which is infinitely long to an output range that lies in the range of natural numbers. So, applying the basic laws of mathematics, if the input from a large dataset is mapped to the output of a small dataset, collisions are inevitable. The following diagram illustrates an example of a collision:



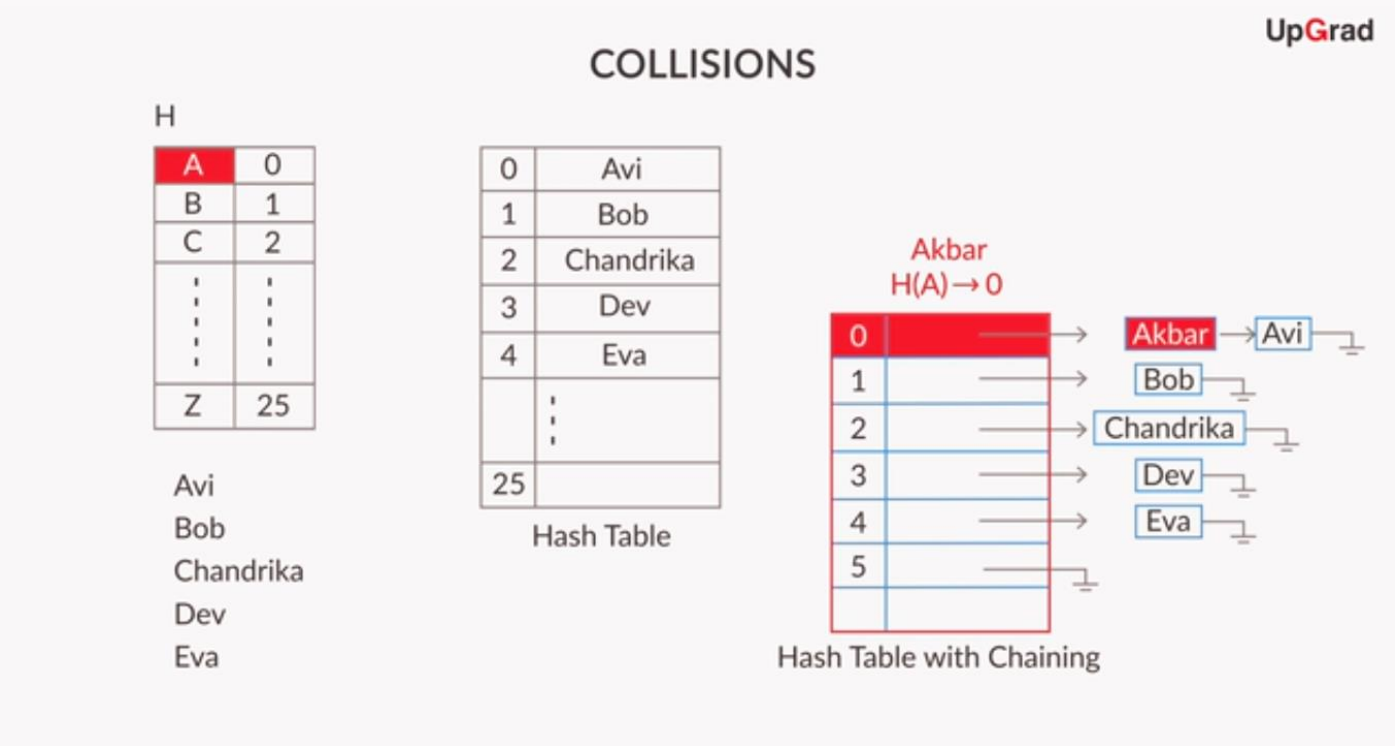
So, the main challenge that lies here is how we resolve this conflict. In other words, how do we come up with a solution to resolve this issue?

Let's say that you already have 'Avi' in the hash table and now another entry is made with the name 'Akash'. Since the first letter of both names is 'A', which means both get hashed to the same index, i.e. 0, now either we can store 'Akash' and delete 'Avi' from the table or the other way around.

But both of these ways are not what we desire because we want the data to be complete in every sense.

Thus, we come up with an appropriate way of handling collisions, i.e. Chaining. In this method, rather than storing a single entry at each index, there is a linked list of entries maintained at each index. So, every time a new entry with the same hash index comes into the picture, it gets added as the head of the linked list at that particular index.

In the figure below, we can see 'Akbar' getting added to the linked list at the 0^{th} index where 'Avi' is already present, thus increasing the size of the list and also giving the opportunity to store multiple keys with the same hash index at the same position. The following illustration shows how the linked list:

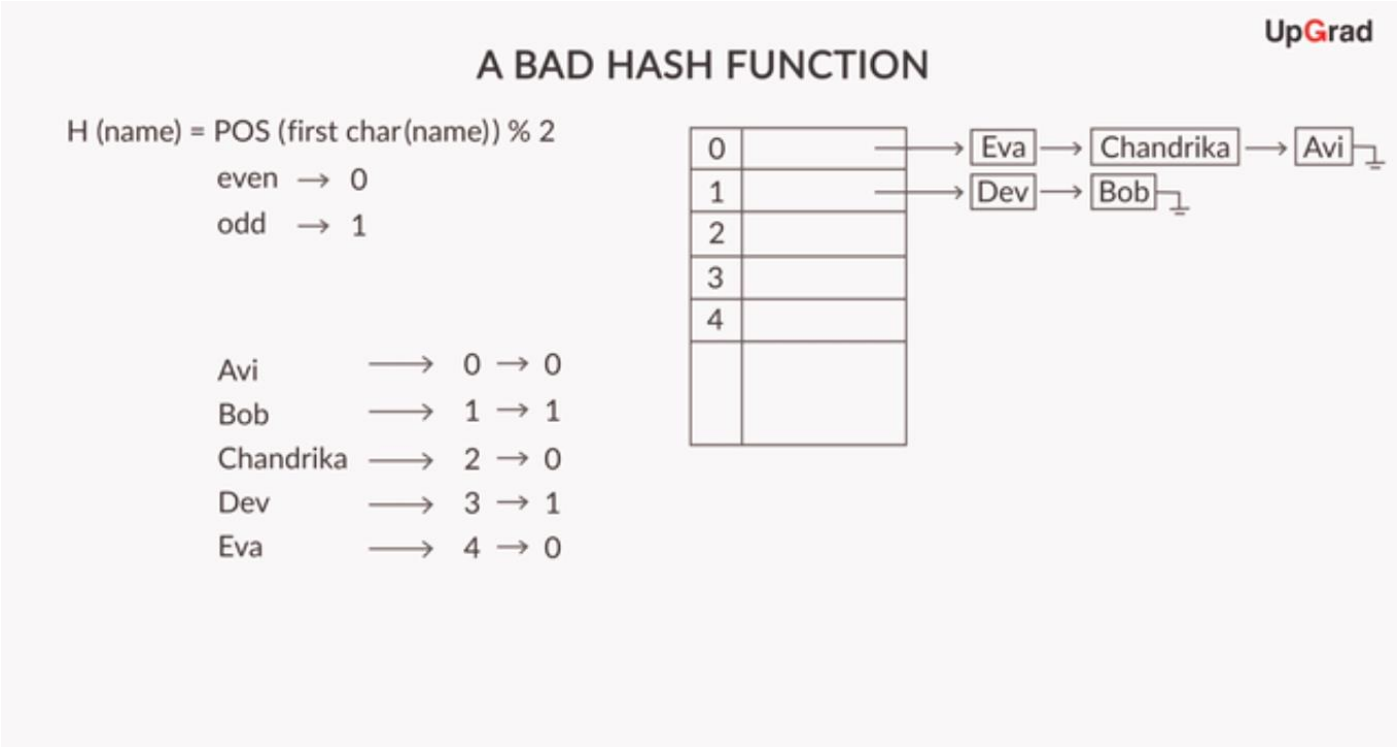


While chaining, the addition of a new entry into the hash table still is an O(1) operation because you are adding the element to the head of the list, whereas searching can take O(n) time when the list at an index grows proportional to the length of the hash table. Regardless of this, maintaining a hash table with chaining is a good way to eliminate collisions.

How the Choice of a Hash Function Affects Hashing

Now that you know that the overall idea of hashing is based on the central concept of hash functions, you will now learn how the choice of a hash function affects the overall process of hashing.

So, for example, let's choose a hash function that is proven to be a bad choice, as shown in the illustration below:



As you can see, the hash function considers the position of the first letter of each name in the English alphabet and then considers whether it lies in an even position or odd position in the alphabet. If it's located in an even position, like 'A' comes at 0 (considering a 0-based index), then the names starting with A, C, E, etc. will get mapped to the hash index 0, whereas names starting with B, D, F, etc. will get mapped to index 1.

In this scenario, only two of the indices of the hash table get occupied and that too with both indices getting chained with long linked lists. This way of storing prevents us from calculating the constant run-time complexity, since we would have to traverse the entire list at a single index in the worst case.

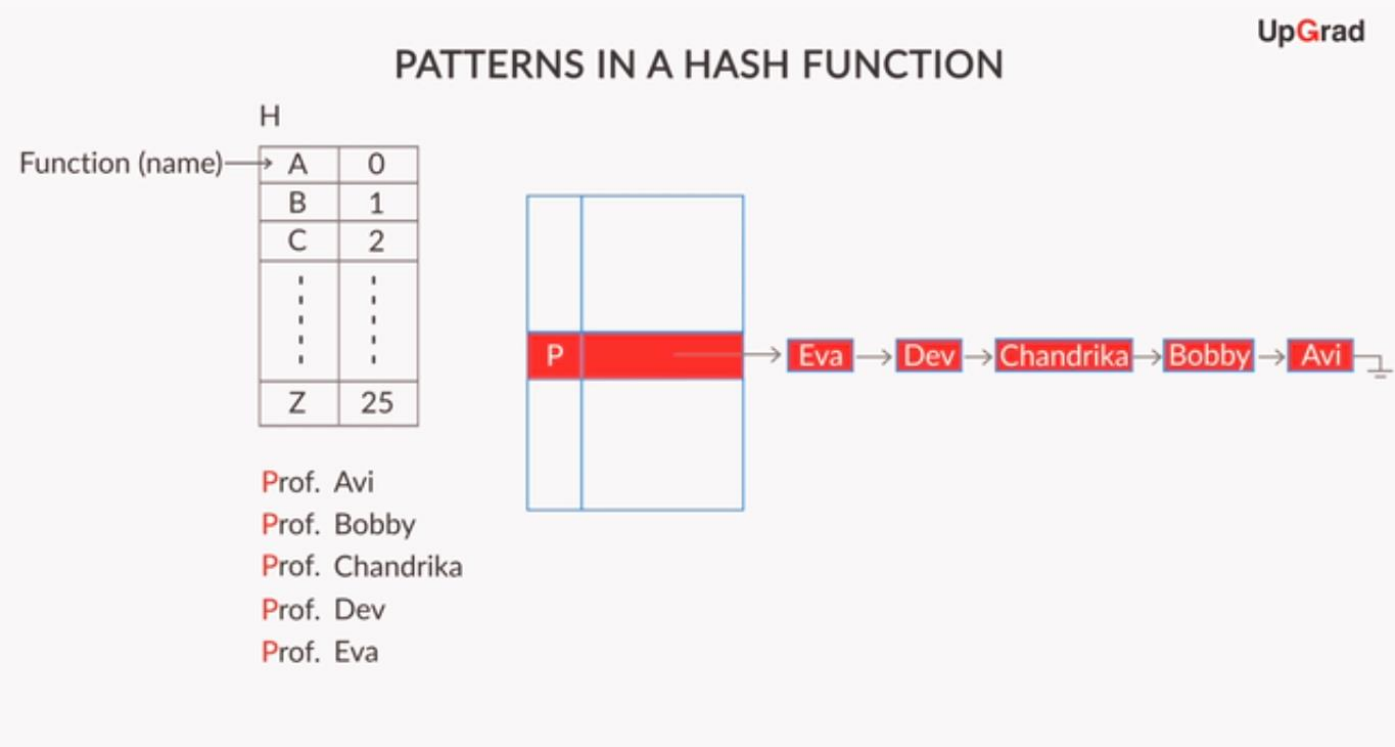
Besides, since only two of the indices are have taken up the load of the entire hash table, a lot of useful space is being wasted.

However, if we were careful in choosing the hash function, the scenario would have been quite different. For instance, we wouldn't be wasting so much space and could have avoided creating the long chains at only two of the indices of the hash table.

Next thing that comes up in hashing is patterns, which you need to avoid in order to ensure better performance of your hash functions.

Let's consider the case where all the names that appear in the phone book unexpectedly have the same first letter. Let's say that we are considering the case of a university, so the names are all saved with the title 'Prof.' ahead of them, e.g. 'Prof. Avi', 'Prof. Bob', and so on.

Now, all of the names get mapped to the same index since they all start with 'P'. In this scenario, a well-defined hash function, which worked well in some other cases, would fail as soon as it encounters a pattern in the input domain. Consider the image below:



What we see above is an example of a hash function that happens to work well in certain cases but doesn't produce desirable results when the input values have a pattern in them.

There is no single hash function that is universally applicable. For a given application, a good hash function should be designed with the following characteristics in mind:

1. It should use all the keys.
2. It should distribute the keys uniformly across the array indices.
3. It should output different hash values for similar, yet unequal, keys.

Phonebook Implementation Using Hash Tables in Java

Now, you will see the hash table API provided by Java and implement a phone book using the functionalities provided by the hash table API.

You basically require two main functions called **put** and **get**, where put does the work of adding values to your hash table and get retrieves the results from the hash table using the key.

First, you must import the hash table to your class, which is done by simply writing 'import java.util.Hashtable;' where all the import statements go in the code. After that, you have to declare the hash table by writing —

Hashtable<int, string> name = new Hashtable<int, String>;

In the preceding code snippet, int is the key of the hash table, string is the value that is intended to be stored at that key, and name is the general name that would be assigned to your hash table.

As for the put function, you have to write 'name.put(key, value)', which hashes the value to the particular key specified in the function itself. And as for the get function, you simply write 'name.get(key)', which returns the value stored at the specified key. The code that we used in the demonstration is shown below:

```
package com.company;

import java.util.*;
import java.util.Hashtable;         //import the Hashtable API.
class Main{
    public static void main(String args[]){
        Hashtable<String,Integer> contacts=new Hashtable<String,Integer>();  //Hashtable API
provided by Java (initialize).

        contacts.put("Ross",24434);  //the put function adds value to the hash table.
        contacts.put("Rachel",24244);
        contacts.put("Chandler",12444);
        contacts.put("Monica",13144);


        //to check if any key is contained in the hash table.
        System.out.println(contacts.containsKey("Chandler"));    //returns a bool value.

        //Let's search for some names in this phone book.
        System.out.println(contacts.get("Chandler")); //the get function gets the value of the
key sent in the function.


        Set<String> keys = contacts.keySet();  //to get all the keys present in the hash table.
        for(String key: keys){
            System.out.println("Number of "+key+" is: "+contacts.get(key));
        }

        // to remove an entry from the hash table.
        contacts.remove("Chandler");

        System.out.println(contacts.containsKey("Chandler")); //to check if the key has been
removed.

        contacts.clear();   //to clear the hashtable completely.
    }
}
```

There are other functions as well provided by the hash table API, such as containsKey(key), which checks whether a particular key is contained in a hash table or not.

The difference between containsKey() and get() is that containsKey() returns a Boolean value whether the key remains in the hash table or not, whereas get() returns the value at that key.

Moreover, Hashtable.keySet() returns the set of keys that are contained in the hash table, and you can traverse the set of keys to get a list of all the keys contained in the hash table.

Also, Hashtable.remove(key) deletes the value entered with the key as given in the function from the hash table, thus clearing out the memory assigned to that key.

Lastly, Hashtable.clear() completely clears the hash table and leaves it empty.

Summary

In this module, you learnt about Hashtable, a data structure used for implementing phone book systems. The main objective of a hash table is that it provides constant time for additions or retrievals, which is the main reason why it's so prevalent. Also, you learnt how choosing the wrong hash function can affect the performance of hashing. Next, you got insights on collisions, an integral part of hashing, and learnt how to resolve them using the method of chaining. Finally, you went through the hash table API provided by Java and its various functions.

Lecture Notes

Lecture Notes – Inheritance and Polymorphism

With the use of inheritance and polymorphism, programmers can reuse lot of existing code, thereby avoiding code duplication and saving on development time and effort.

Inheritance

Let’s look at an example of classes by implementing a Rectangle class.

```
class Rectangle {
  protected final float length ;
  private final float breadth ;
  public Rectangle ( float l, float b) {
      this . length = l;
      this . breadth = b;
  }
  public float area () {
      return this . length * this . breadth ;
  }
}
```

Figure 1: A rectangle class

The code in figure 1 can be tested as follows:

```
public class Geometry2 {
  public static void main(String[] a) {
      Rectangle r = new Rectangle(10, 20);
      System.out.println(" area = " + r.area());
  }
}
```

Next, let's also add a Square class to the code.

```
class Square {
  private final float length;

  public Square(float l) {
      this.length = l;
  }

  public float area() {
      return this.length * this.length;
  }
}
```

Figure 2: A square class

The code in figure 2 can be tested as follows:

```
public class Geometry2 {
  public static void main(String[] a) {
      Rectangle r = new Rectangle(10, 20);
      System.out.println(" area = " + r.area());
      Square s = new Square(200);
      System.out.println(" area = " + s.area());
  }
}
```

The Rectangle and Square classes _g. 1 and _g. 2 aren't similar by co-incidence. The fact is, a square is a rectangle with its length equal to its breadth. Unfortunately, the code here doesn't capture this fact. It would be nice if we could make this knowledge an explicit part of our code. Does Java allow us to do that? Yes, it does!

```
class Square extends Rectangle {
  private final float length;

  public Square(float l) {
      this.length = l;
  }

  public float area() {
      return this.length * this.length;
  }
}
```

Figure 3: Square class declared a sub-class of Rectangle

The code in figure 3 modifies that in figure 2 by declaring Square as a sub-class of Rectangle, by using the extends keyword as shown. This is good. But it doesn't do much functionally. However, following version of the Square class does the real magic!

```
class Square extends Rectangle {
  public Square(float l) {
      super(l, l);
  }
}
```

Figure 4: Square class declared a sub-class of Rectangle

Several lines from the Square have been reduced. Does this code even compile? Sure enough, it does! In particular, the call to s.area() from the main method compiles in spite of there being no area method anymore in the Square class. And if you run the code, it seems to work just as _ne as before, giving the same result. How could this happen?

The reason for this is: Square being the child of the Rectangle class, inherits all its properties. This is called inheritance, the most important feature of all object oriented programming languages. We say that:

- Square inherits from Rectangle.

- Square is the sub-class/child-class/sub-type/child-type/derived-class of Rectangle.
- Rectangle is the super-class/parent-class/super-type/parent-type of Square.

Thus, area method in Rectangle also becomes a property of Square. But this area method needs the length and breadth attributes of Rectangle to be set to the correct values (in this case, equal to the length attribute (now deleted) of the Square class). Where and how does this happen?

It happens in the constructor of Square class, when super(l, l) executes. To understand what this does observe figure 5

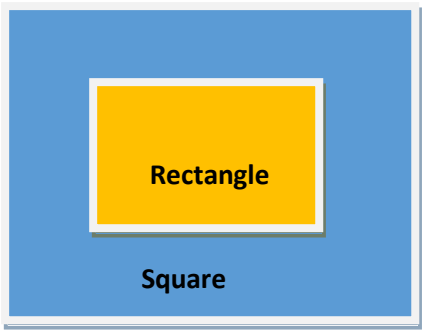


Figure 5: Square object with an object of Rectangle embedded within itself

Each object of a sub-class can be visualised as embedding within it an object of its super-class. Thus, an object of Square class has within it an object of Rectangle as shown in figure 5. During the construction of a sub-class object, the first step to complete is the construction of the embedded super-class object. The call to super method does precisely this. It calls the constructor of the super-class with the given arguments, in this case l and l (l being the parameter to Square's constructor).

This initialises the embedded Rectangle to have its length and breadth attributes both set to the argument passed to the constructor to Square. Thus, a subsequent call to s.area() in the main method calls the area method of the embedded Rectangle object. This, in turns returns the product of length and breadth attributes (which, remember, are equal to each other) thus giving us the correct area of the Square.

So, super is a new keyword we have learned, it used in the context of a sub-class, is essentially a reference to the embedded instance of the superclass.

Can we derive further classes from Square? Yes, and we present an example in figure 6.

```
class Point extends Square {
  public Point() {
      super(0);
  }
```

Figure 6: Point class declared as sub-class of Square

Again, a very rudimentary class, with hardly any code! It just creates the new type Point as a sub-type of Square, making point a special type of square with zero length, which is indeed a reasonable way to look at things.

The code in figure 6 can be tested with the following lines added to the main method:

```
Square p = new Point(
 System.out.println(" area = "+p.area());
```

This works just smoothly, giving us the expected results:

```
area = 0.0
```

Indeed, a point is a square (and hence a rectangle) with zero area. Pictorially, the scenario can be depicted as in figure 7.

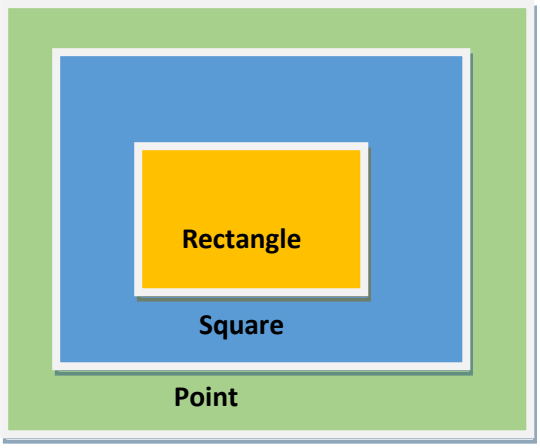


Figure 7: Point object with an object of Square embedded within itself

Protected Access Specifier

Any method from within Square class won't be able to access Rectangle's length attribute directly. This may be OK sometimes, but sometimes this may be too restrictive. For example, the following piece of code

```
public float circumference(){
   return 4.0 f*this.length;
   }
```

if added to the Square class would lead to a compilation error:

```
error : length has private access in Rectangle
return 4.0 f * this . length ;
```

In other words, the designer of the Rectangle class may want the implementers of its sub-classes to have direct access to length. One option would be to turn length into a public attribute. This would work, but this is an overkill, and too permissive. We would like to tune the visibility of length to just the level where the sub-classes have direct access to it, but it remains invisible to any other class in the program. For this we use the protected access specifier:

protected float length;

With this, it is possible to write code within Square class that directly refers to length. For instance, the line of code above, when added to Square class, will not give any error.

INHERITANCE AND POLYMORPHISM- I

To summarise:

- Inheritance is a powerful feature to avoid code duplication
- Whenever there is a common code, group them into super and subclasses.
- Use the extends keyword for inheriting features of a superclass into a subclass
- In order to restrict access of class members to only subclasses, use the protected modifier.

Polymorphism

Consider the modified main method shown in figure 8. The notable point here is that the variable r, which is of the type Rectangle is first initialised to a Rectangle, which is familiar. However, subsequently, we assign to it an instance of a Square, and then a Point. We print the area in each case

```
public static void main(String[]a){
   Rectangle r=new Rectangle(10,20);
   System.out.println(" area = "+r.area());
   r=new Square(10);
   System.out.println(" area = "+r.area());
   r=new Point();
   System.out.println(" area = "+r.area());
      }
```

Figure 8: Square class declared a sub-class of Rectangle

The output of running the code in figure 8 is as shown below:

area = 200.0

area = 100.0

area = 0.0

And look, the area gets printed correctly for all the three shapes.

At this point, a slight refinement of terminology. Here, rather than thinking of r as a variable of type Rectangle, it's more proper to think of it as a reference of the type Rectangle. This means that it can point to an object of the type Rectangle. What we observe further in the code in figure 8 is that it is allowed for r to point to any object whose type is a sub-type of Rectangle.

In fact, references in Java are called polymorphic. And this property of a language which implements polymorphic references in the above sense is called polymorphism, more precisely, dynamic polymorphism. We discuss the meaning of this term a little later. But let's try to appreciate what this feature can do for us. Consider the modified driver code shown in figure9.

The main calls another method printRectangles. As argument, it passes an array of Rectangles constructed out of three Rectangles: r (indeed a Rectangle), s (actually, a Square) and p (which is in fact a Point).

Firstly, note that Java allows us to construct an array of Rectangles, wherein the elements can be objects of any sub-class of Rectangle. Secondly, passing this array to printRectangles gives us just the expected output: the areas of all the Rectangles in the array getting printed.

What does this mean? This means that printRectangles method couldn't care less what the precise type of the objects in the rarray array are. The Java type system assures that they all are instances of Rectangle or one of its sub-classes. In fact, there's no need for Square and Point classes to even exist at the time of implementing printRectangles. Even if these classes are implemented afterwards, much after the time printRectangles is implemented, everything here is guaranteed to work perfectly.

```
public static void main(String[]a){
   Rectangle r=new Rectangle(10,20);
   Square s=new Square(200);
   Square p=new Point();
   Rectangle[]rarray={r,s,p};
   printRectangles(rarray);
    }
public static void printRectangles(Rectangle[]rarray){
    for(Rectangle rec:rarray){
        System.out.println(" area = "+rec.area());
      }
     }
```

Figure 9: printRectangles method prints an array of Rectangles

The above idea is not new, but has existed for a long time in engineering. Wherever there is a system with components that interact and interoperate, engineers go about designing them by defining what we call interfaces. Consider the USB port, the VGA, power audio ports of your computer. As long as a VGA cord following the specifications of VGA is inserted into your computer's VGA port, it is kind of guaranteed to work. It doesn't matter who manufactured the VGA cord. Similarly, the Android OS can be installed on any Android compatible device. It could be any of hundreds of phone brands, it could be a tablet, a PC, a TV or anything else. Are these devices identical? No. But they follow the interfaces specified by the creators of Android OS. Internally, each one of them may have many variations, but Android doesn't concern itself with them.

Similarly, the super-class (here, Rectangle) is kind of an interface which the printRectangles method accepts. The inheritance rules of Java guarantee that all sub-classes of Rectangle adhere to its interface, i.e. if area method is called on them, it will be available. And therefore, printRectangle is able to work with any array of Rectangles, even when it may actually contain objects of other types. All that's needed is those other types must be sub-types of Rectangle. Java's type-system makes sure that requirement is fulfilled: an attempt to populate a Rectangle[] array with an object of a type which isn't a sub-type of Rectangle will fail at compile-time.

In a short while, we will have a bit more to say about interfaces, which are a very important concept in Java and OOP in general.

Method Overriding

What we have learned so far about inheritance is good to create sub-classes which are specialisations of their super-classes. In other words, they are the same as their super-classes, but for some additional constraints. This is useful, but not useful enough. Often there are situations when we wish to modify our super-classes as per need. I will present here a simple example.

Let's add a method printName in the rectangle class:

```
public void printName(){
    System.out.println(" I 'm a rectangle . ");
      }
```

Now, let's call this function from the main for all the Rectangles we have created there.

```
Rectangle[]rectangles={r,s,p};
   for(Rectangle rec:rectangles){
       rec.printName();
      }
```

This will produce the following output:

I'm a rectangle .

I'm a rectangle .

I'm a rectangle .

This output is technically correct, but not interesting. It would be nice if we could print the correct name as per the sub-class. For example, for a Square, the message should be "I'm a square.". Is it possible to have this output?

Given the fact that in the context of main, each shape is being accessed through a Rectangle type reference, this looks unlikely. Nevertheless, let's go ahead and implement the methods that we would have liked to be called to print the correct shape names.

In Square class, we add:

```
public void printName(){
   System.out.println(" I 'm a square . ");
      }
```

In Point class, we add:

```
public void printName(){
      System.out.println("I'm a point .");
      }
```

Let us compile the code and run it:

I'm a rectangle .

I'm a square .

I'm a point .

As you can see, the program works as expected.For each object, the version of printName as

defined in the sub-class was called. Indeed, that's what happened. Even though printDetails is called from the context of main, with a reference to the Rectangle class, the implementations in the sub-classes are called. In fact, it's quite allowed to implement the main method in a separate source file, and compile it even before the sub-classes like Square and Point are written. These can be written and added to the program later, and yet, everything would work seamlessly. This feature is realised with a mechanism called **dynamic dispatch.**

To make things further interesting, let's implement a method named printDetails in the Rectangle class along with the main method as shown in figre 10.

```
public void printDetails(){
      this.printName();
      System.out.println("... and my area is "+this.area());
      }
```

Figure 10: printDetails method prints the details of the object.

Note that printDetails has a call to this.printName. If this method is called on a sub-class of Rectangle using a Rectangle reference, which version of printName would be called? Rectangle's or the sub-class's? To test, we also modify the driver code as follows:

```
Rectangle r=new Rectangle(10,20);
      Square s=new Square(200);
      Square p=new Point();
      Rectangle[]rectangles={r,s,p};
      for(Rectangle rec:rectangles){
      rec.printDetails();
      }
      }
```

When we compile and run the modified code, we get the following output:

```
I'm a rectangle .

... and my area is 200.0

I'm a square .

... and my area is 40000.0

I'm a point .

... and my area is 0.0
```

As you can observe, the correct versions of printName gets called from within Rectangle.printDetails.

This is one of the most powerful features of object-oriented programming and should be mastered well by an object-oriented programmer. Let's say, in a class C1 a method m1 calls another m2 in the same class, which has implementations in sub-class C2. Now, if a reference r points to an instance of C1 and a call r:m1 is called. Internally, r:m1 will call C2:m2. Because of this, we say that C2:m2 overrides C1:m2.

Method Overloading

Another type of polymorphism is known as static or compile time polymorphism. This is implemented with the help of method overloading. Whenever methods with similar name are defined in the same class or an immediate subclass, they can be made to perform differently.

Each version of a method in this case, must have a different argument list. The argument list can be made different either by changing the number of arguments or the type of arguments. In the code given below, the area() method based on the argument passed to it during run time, will invoke the corresponding version of the method. Thus, using the same method, you can calculate area for both rectangle and a square

```
class Shape {
  void area(double length) {
      System.out.println("Area of Square is: " + Math.pow(length, 2));
  }

  void area(double length, double breadth) {
      System.out.println("Area of Rectangle is: " + (length * breadth));
  }
```

Figure 11: area( ) method overloaded for square and rectangle

Abstract Classes

Let's add another class Circle into our family of classes as shown in figure 12.

```
class Circle {
  public static final float PI = 3.141
  private float radius;
  f;

  public Circle(float r) {
      radius = r;
  }
}
```

Figure 12: Circle class.

If we try to instantiate a Circle in the main and try to refer to it using an Rectangle reference, this will lead to a compilation error: the types simply don't match.

The first solution is very simple. Define a class Shape as the superclass of both Circle and Rectangle. Let's do it.

```
class Shape {
  public printName() {
      "I'm a shape .";
  }

  public float area() {
      return 0;
  }

  public void printDetails() {
      this.printName();
      System.out.println("... and my area is " + this.area());
  }
}

class Circle extends Shape {
...

  class Rectangle extends Shape {
```

Figure 13: Shape class

In Shape, we provide a default definition of the area method and printName methods. Note that these implementations don't make much sense. The printName method doesn't provide adequate information about the Shape, and area method would simply give a wrong result for all but Shapes with zero area. Nevertheless, they are needed. Otherwise, the code will not compile.

The main methods gets modified as follows to accommodate the above

```
public static void main(String[] a) {
          Rectangle r = new Rectangle(10, 20);
          Square s = new Square(200);
          Square p = new Point();
          Circle c = new Circle(10);
          Shape[] shapes = {r, s, p, c};
          for (Shape sh : shapes) {
              sh.printDetails();
          }
      }
```

When we compile and run the above code, we get the following output:

I'm a rectangle .

... and my area is 200.0

I'm a square .

... and my area is 40000.0

I'm a point .

... and my area is 0.0

I'm a shape .

... and my area is 0.0

Note that the outputs corresponding to the Circle c are wrong. In case of the Circle, it was the Shape's implementation of printName and area that gets called. This is not merely undesirable, but completely wrong.

On looking at the problem a little more closely, we realise the following:

- We have forgotten to implement printName and area methods in Circle class.
- The default implementation of printName and area methods provided in Shape class are really unnecessary, and are doing more harm than good. They have been put there just to satisfy the compiler (which is a very bad reason to implement a method). Not surprisingly, they are becoming the source of a bug which could be quite hard to detect.

What's the nature of this bug? This bug happens whenever there are default implementation of methods provided in the super-class which are necessarily supposed to implemented in the sub-classes, and somehow the implementer of the sub-class forgets to provide one. What we really want is:

1. No enforcement to provide useless dummy implementations of methods in the super-classes just to appease the compiler.
2. If we forget to implement such methods in the sub-classes, the compiler should alert us of our mistake.

Can we design our code to get the above? Yes, we can. By using abstract classes. To full condition 1 above, we make the following change to the Shape class:

```
abstract class Shape {
    public abstract void printName();
    public abstract float area();
    public void printDetails() {
         this.printName();
         System.out.println("... and my area is " + this.area());
       }
    }
```

Figure 14: Abstract Shape class

The modified code in figure 14 declares Shape as an abstract class: a class with one or more abstract methods. Abstract methods are methods which have a declaration in the class, but have not been implemented. In figure 14,printName and area methods are abstract methods of Shape.

A very important characteristic of abstract classes is that they can't be instantiated directly. That is, an attempt to have something like **Shape s = new Shape()** in the program would not be accepted by the compiler. The only way to instantiate abstract classes is through their concrete (which are not abstract themselves) sub-classes.

Let us try to compile the above code. Given below is the output:

error : Circle is not abstract and does not override

abstract method area () in Shape

class Circle extends Shape {

^

This says that Circle, which is not declared abstract doesn't provide necessary implementation for the abstract methods of its super-class Shape. This takes care of condition 2 above: the compiler has pointed us out our mistake of having forgotten to provide implementations for printName and area methods in Circle class.

To correct this, we add the implementations of printName and area methods in the Circle class as shown in figure 15.

```
class Circle extends Shape {
   private float radius;
   public static final float PI = 3.141
      public Circle(float r) {
         radius = r;
          }

      public void printName() {
         System.out.println("I'm a circle .");
          }

       public float area() {
          return PI * radius * radius;
          }
}
```

Figure 15: Circle class with printName and area methods implemented

Now the code compiles and on running it, we get the following output:

I'm a rectangle .

... and my area is 200.0

I'm a square .

... and my area is 40000.0

I'm a point .

... and my area is 0.0

I'm a circle .

... and my area is 314.1

Note that the errors in the last two lines of the output have now been corrected.

Interfaces

A possible design of the Shape class would to make it completely abstract, i.e. when all its methods are abstract, as shown in figure 16.

```
abstract class Shape {
   public void printName();
   public float area();
      }
```

Figure 16: A completely abstract Shape class.

Java presents another syntactic way to define such completely abstract classes. They are called interfaces, as shown in figure 17.

```
interface Shape {
  void printName();
  float area();
      }
class Circle implements Shape {
...
      }

class Rectangle implements Shape {
...
      }
```

Figure 17: Shape as an interface.

Interfaces are more significant than just being able to build completely abstract classes. An interface is not extended, but implemented, by its sub-classes, as shown in the modified code for Circle and Rectangle.

INHERITANCE AND POLYMORPHISM- II

To summarise:

- Polymorphism is an important feature of object oriented programming, implemented with the help of method overriding and method overloading
- Static Polymorphism is implemented with the help of method overloading
- Dynamic Polymorphism is implemented with the help of method overriding
- Abstract classes and Interfaces are helpful in cases, when we want to leave the implementation details for individual classes that inherit or implement them.

Lecture Notes

Priority Queues and Heaps

Welcome to session on Priority Queues and Heaps.

In the module of Stacks and Queues you learnt that the element entering first will be the element to come out first and that is called queues while, if the element going in last comes out first then it's called stack.
But this is always not the case, sometimes it may happen that the element going in first may have to come out first or the element going in first may have to come out first.
Consider the example of airline checkin queue- If you are in the checkin queue to board a flight but your flight is about to leave and many people are there infront of you for their checkin. So what will you do in that scenario.

You will certainly not wait for your turn because you might loose your flight, you will try to jump the queue and checkin early. This type of situation represents a priority queue.

Check-In Queue In Airport

The man who came in last has to leave early here, so his checkin will be done first.

Priority Queues

Consider a sports club who wants to sell its ticket for a match to the waitlisted customer it has. There are 4 types of tickets available club ticket, season ticket, online ticket and counter ticket and each of them are of different priority or are of different cost i.e club ticket being the highest priority cost the most while counter ticket cost the least and is of lowest priority. So what will be the order in which their tickets will be confirmed? Will it be first come first serve or of FIFO kind? Not necessarily, the club would certainly want to maximise its profit so it will sell the ticket which cost the most first.

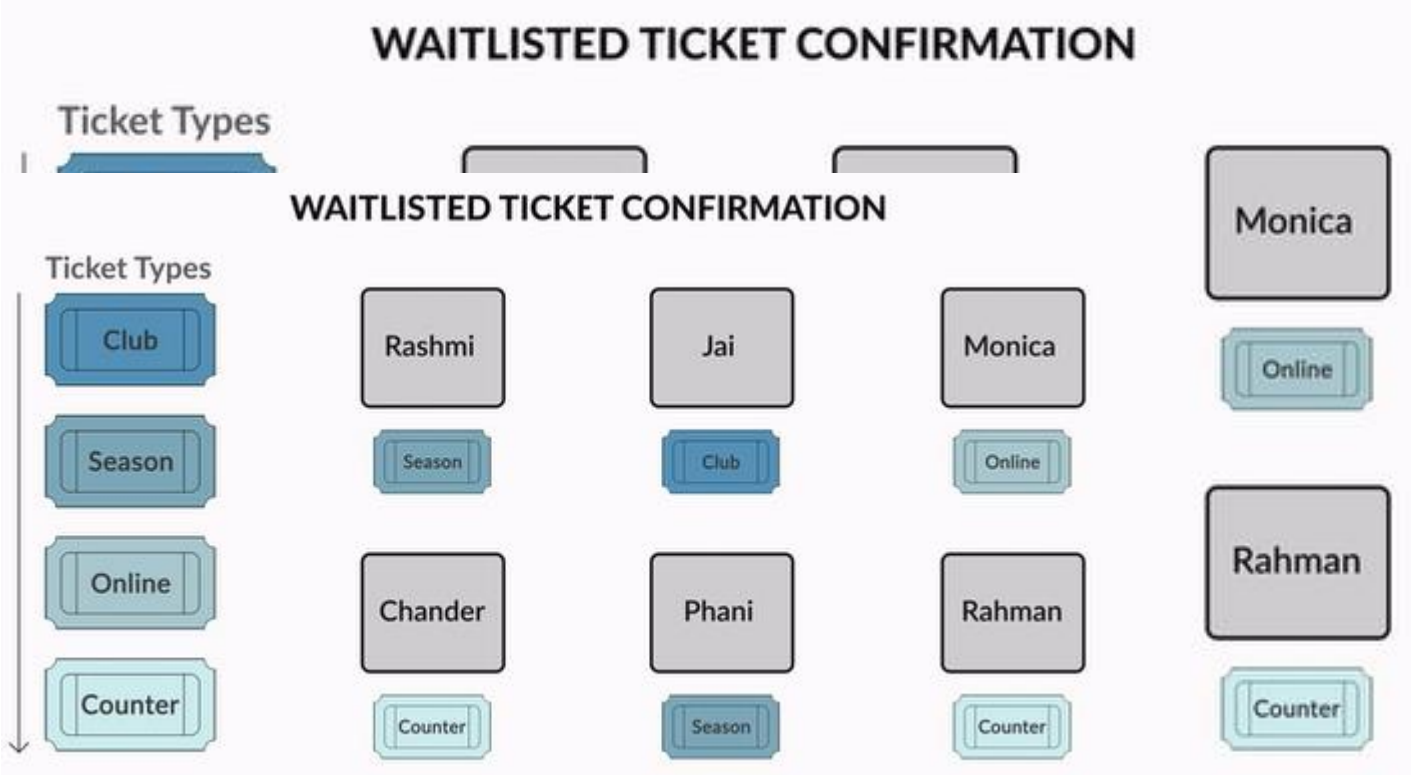
This structure of selling tickets on the basis of their priority and not in the order they came in is analogous to an abstract data structure called "**Priority Queue**". In a Priority queue, the element with the highest priority will go out first and the one with least priority will go in last.
In computers the highest priority element is shown with least integral value of priority In case of tickets club ticket will get a priority of 1 and counter ticket will get a priority of 4.
A Priority queue is therefore anything:

1. into which you remove (removeMin) things from the front of the queue.

2. that removes the highest priority element first from the queue.

Priority Queue Operations

Considering the waitlisted ticket example –



In the above example club has the highest priority and counter the lowest. So, the order in which ticket will be confirmed will be starting with Jai then Rashmi, Phani, Monica, Chander and lastly Rahman.
There are two types of priority queue min and max but we will learn only about min-priority queue and java ADT class also implements min-priority queue.
Priority Queue ADT has a class in java for it through which you can directly implement some functions like:
poll() : It returns the minimum element from the priority queue as it will have the highest priority and then removes that element.
peek() : It returns the smallest element from the priority queue as it will have the highest priority .
add(T element) : It adds an element in the priority queue of type T.
isEmpty(), which returns true if the priority queue is empty and false otherwise.
size() which returns the size of the priority queue.

To implement priority queue you need to implement comparator function if the object type is generic.
If you change the comparator accordingly you can implement max-priority queue also.

Implementing a Priority Queue

You could either implement a priority queue using a custom class made using lists (array lists or linked lists) such as the one below, where you used a linked list to implement a Priority Queue.

In the following program, you have created the basic functions of priority queu.

```
import java.util.Comparator;
import java.util.List;

public class ListPQ<T> {

  protected final Comparator<T> comparator;

  public ListPQ(Comparator<T> comparator) {
    this.comparator = comparator;
  }

  public void add(T element) {
    if(!this.isEmpty()) {
      for(int i = 0; i < this.list.size(); i++) {
        T e = this.list.get(i);
        if(this.comparator.compare(e, element) >= 0) {
          this.list.add(i, element);
          return;
        }
      }
    }
    this.list.add(element);
  }

  public T removeMinimum() {
      return this.list.remove(0);
  }

  public T minimum() {
```

```
    return this.list.get(0);
  }
   public boolean isEmpty() {
       return this.list.isEmpty();
  }

   public int size() {
       return this.list.size();
  }

}
```

You could also use the internal Priority queue library provided by JAVA, which you could call using -

```
import java.util.PriorityQueue;
```

The features poll, peek, isEmpty and size can all be called using this library.

So the Priority Queue code implemented using this library would have looked like the following:

```
import java.util.PriorityQueue;
import java.util.List;

public class PQImplement<T> {

public static void main(String[] arg) {
      PriorityQueue <Integer> PQ = new PriorityQueue <Integer> ();

         for(int i=2; i<=20; i=i+2) {
             PQ.add(new Integer (i));
         }
      int x= PQ.peek();
      int y= PQ.poll();
      int z= PQ.size();

    }
}
```

There might be elements, such as custom objects, that don't have a predefined order. In such cases, you can utilise a comparator interface to create a custom order of the elements. Here is the code for comparator function.

```
Comparator

compare(a,b){
    if a<b
        return -1

    else if a>b
        return 1

    else
        return 0
        }
```

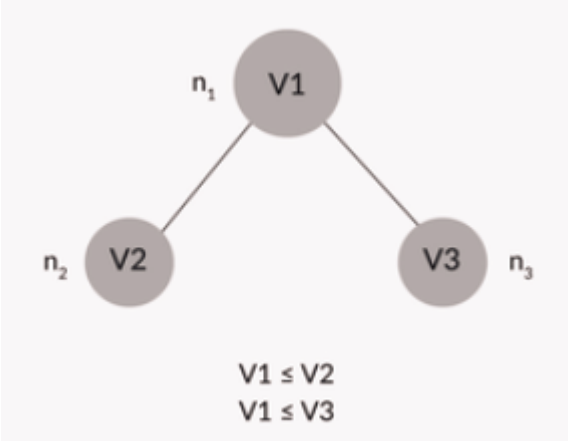
Heaps

	Insert	FindMin	RemoveMin
Unordered ArrayList	O(1)	O(n)	O(n)
Unordered LinkedList	O(1)	O(n)	O(n)
Ordered ArrayList	O(n)	O(1)	O(1)
Ordered LinkedList	O(n)	O(1)	O(1)

Lists are not the most efficient way to implement Priority queue, as you can see in the above table. Overall Big O for all operations is O(n). So, we have an alternative implementation for that known as Binary Heaps. Heaps can implement all of the operations of Priority Queue in maximum O(log n) time.

Binary Heap has two basic properties:

- They are complete binary trees
- Order property: They maintain a hierarchical (level-wise) order among the nodes of their trees.

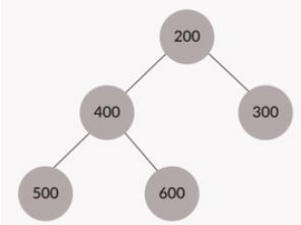


The order among the nodes at the same level does not matter and duplicate nodes can exist in a heap. Also, there are two types of heaps:

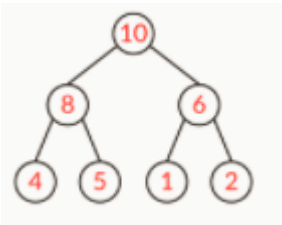
- Min-heap
- Max-heap

In min-heap the minimum element is always at the root node. While in max-heap maximum element is always at the root node.

Min-Heap:



Max-Heap:



Addition or removal from a heap is done always from the last node. The minimum value, which is at the root node, cannot be removed directly. First, it is swapped with the last node and then it is deleted from the last node.

Any modifier operation(add or removeMin) performed on heap will lead to the disturbance of heap property. So, we have to perform Heapify operation to restore the heap property:

There are two heapify operations:

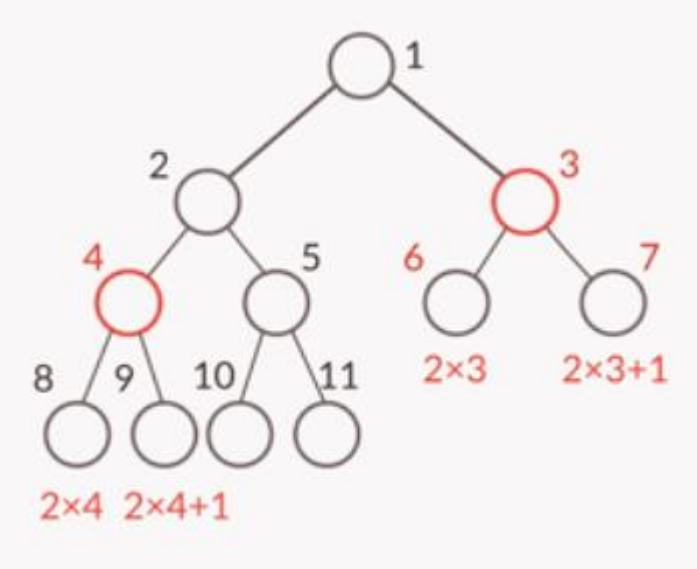
- HeapifyUp is used during insertions
- HeapifyDown is used during deletions

Both of these operations are recursive as the property violations can bubble up to the root or bubble down to the leaf node.

Here is the pseudo code for HeapifyDown operation:

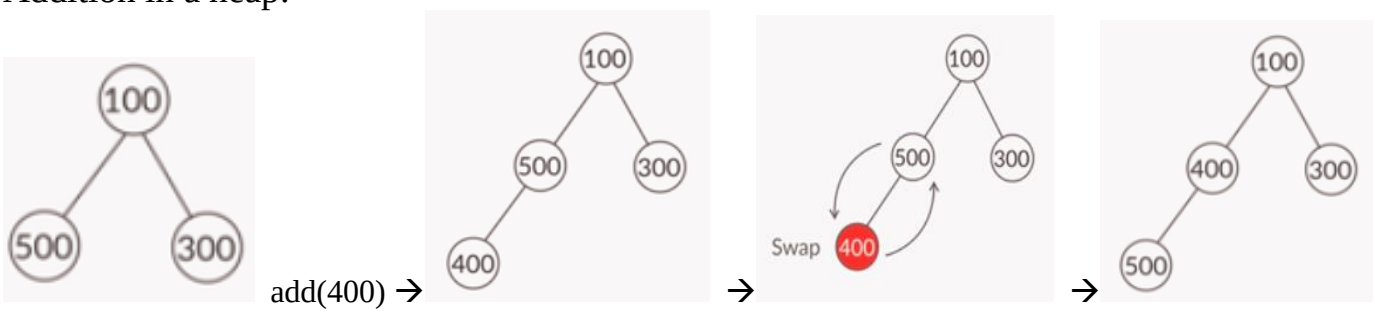
```
HeapifyDown(A,Index,n)
   min =Index
   l = 2*i
   r = 2*i +1

   If l<n and A[l]<A[min]
         min=l

   If r<n and A[r]<A[min]
         min=r

   If min != Index
         swap(A[min], A[Index])
         HeapifyDown(A,min,n)
```

Heap in Java is implemented as an array. In the array, the root node is the initial element of the array and any child node's index is 2*i or 2*i+1, if the parent's index is i. In this implementation the array starts with index 1.

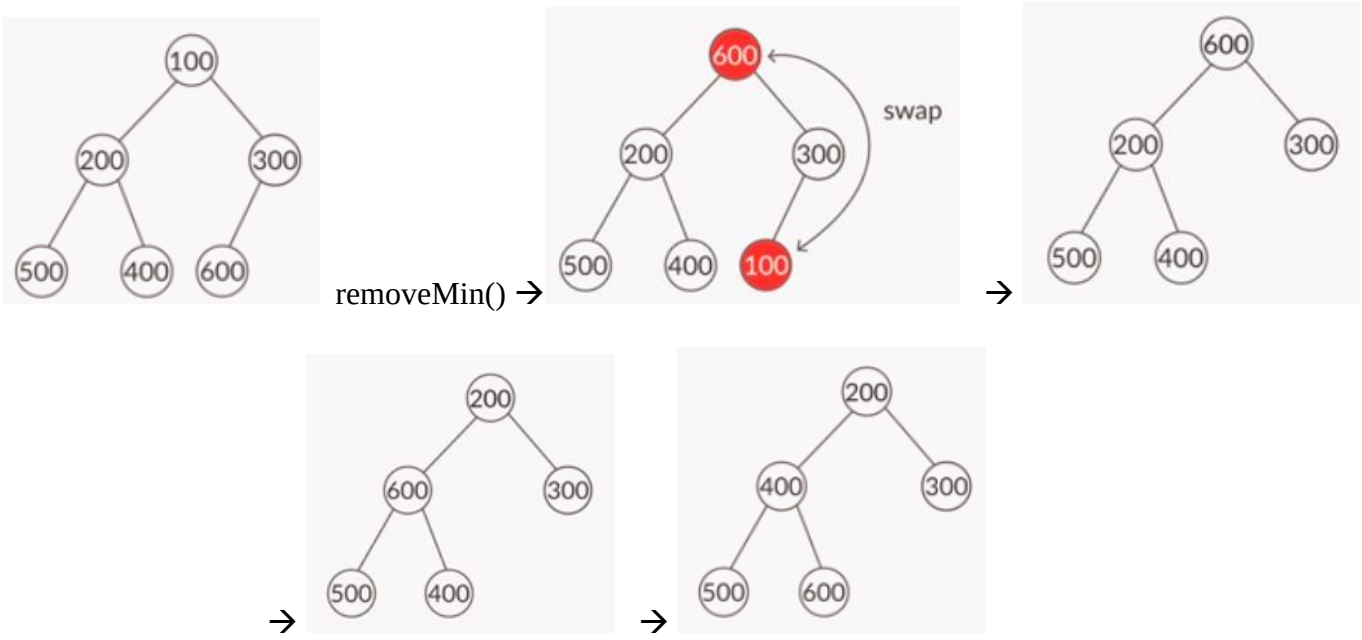


Addition and Removal in a Heap

Addition in a heap:



Remove Minimum in a heap:



Both addition and deletion in a heap takes O(log n) time.

HeapSort

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array. So, Priority Queues can be implemented using a heap to better its performance. Heap sort is not a Stable sort, and requires a constant space for sorting a list.

Application of Priority Queues and Heaps

There are many uses of priority queues and heaps. Some of them are:

- Event-driven simulation: customers in a line
- Collision detection: "next time of contact" for colliding bodies
- Data compression: Huffman coding
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- AI Path Planning: A* search
- Statistics: maintain largest M values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
- Spam filtering: Bayesian spam filter

Summary

Here's a quick summary of the topics you learnt about in this session:

1. Priority queues: This is an abstract data type (ADT) that resembles a regular queue or a stack data structure, but this data structure has the functionality to add a 'priority' to each of its elements to make organising data more efficient. In a priority queue, a high-priority element is served before a low-priority element. If a priority queue has two elements with the same priority, the order of the elements in the queue determines which one gets served first.

   In its most basic form, a priority queue ADT supports the following operations:

   - poll(): Removes an instance of the minimum element from a priority queue and returns the element.
   - peek(): Returns the minimum element of a priority queue but does not make any changes to the state of the priority queue.
   - add(T element): Adds an element to a priority queue.
   - isEmpty(): Checks whether a priority queue is empty or not and returns 'True' if the priority queue is empty and 'False' otherwise.
   - size(): Returns the number of elements in a priority queue.

   You learnt that a normal queue data structure could not implement a priority queue efficiently because searching for the highest-priority element will take O(n) time. A list, whether sorted or not, will also require O(n) time for either insertion or removal.
   Instead, you found a data structure that is guaranteed to give good performance in this special application.

   This session also took you through the heap data structure, which inserts and removes elements in O(log n) time. A heap has two basic properties. First, it is a complete binary tree; and second, it is partially ordered. This means that there is always a relationship between a child node and its parent's node. There are two variants of the heap data structure, depending on the parent-child relationship.

   - Max-heap: It is a heap data structure in which all the child nodes are less than or equal to their parents. As the root stores a value that's greater than or equal to its children's values, the root holds the maximum of all values in the tree.

   - Min-heap: It is a heap data structure in which all the child nodes are greater than or equal to their parents. As the root stores a value that's less than or equal to its children's values, the root holds the minimum of all values in the tree.

   You can sort a sequence of elements efficiently by using a heap-based priority queue. This algorithm is called **heap sort**.

Lecture Notes

Stacks and Queues

Welcome to session on Stacks and Queues.

In the OOPS course, you learnt about two data structures: array lists and linked lists. You saw how both are implementations of the same abstract data type list. You learnt about the various capabilities and properties of lists, and how they could be used in solving practical software development problems (recall Institute Management System). You learnt how despite being functionally nearly identical to each other, array lists and linked lists were quite different from each other. In particular, when you take into account, their performance aspects, you observe that one of them fairs better than the other in certain scenarios, and worse in certain others. For example, getting a value by its index is typically much faster in an array list. However, adding/removing an element at an arbitrary location may be faster in a linked list.

Data Structures

Algorithms are computational procedures which deal with data. They create them, modify them and refer to them time to time. In order to deliver good results, your algorithms need the program data to be arranged in a way suitable for the purpose. A well arranged data will make it easy to implement algorithms (and software systems in general) that not only are fast and efficient, but are also flexible while being robust and secure. Hence, how the data is arranged within a program is a crucial question that must be answered at a very early stage of design of a program.

The subject of data structures is the study of some of these 'arrangements' of data which have been found useful by software developers in a variety of scenarios. While studying a data structure, our aim should be to understand:

1. What are their capabilities?

2. In what scenarios can they be used and how?

3. What are various implementations and how do they compare in terms of performance (space/time)?

You have till now seen the use of terms - *abstract data types* and *data structures* apparently synonymously, but there is a difference between the two. Abstract data type is the definition of the interface (i.e. the properties and capabilities) of a data arrangement. Data structure is a specific implementation. For example, list is an ADT (where the elements are linearly sequence, each associated with an index or position represented by a nonnegative integer); linked list is an implementation. In other words, there may be multiple data structures which implement the same ADT. In most places, you'll find these two terms used as near synonyms.

Stacks

Consider a stack of books - or plates or some such things. If you were to pick one book from the stack, from which portion of it would you pick it? From the top? From the bottom? Or from somewhere in between? Of course, any sane person would pick it up from the top. Trying to pull out a book from anywhere else in the stack would be possible, but would lead to difficulties, and untoward results. In the same manner, if you were to add a book to this stack, where would you prefer to add it? Again, at the top, of course.

This structure of books where you add things at the top and remove things again from the top is analogous to a data structure called "**Stacks**". In a stack, what goes in last is the first to come out. This property of the above structure is called last in first out (or LIFO) order.

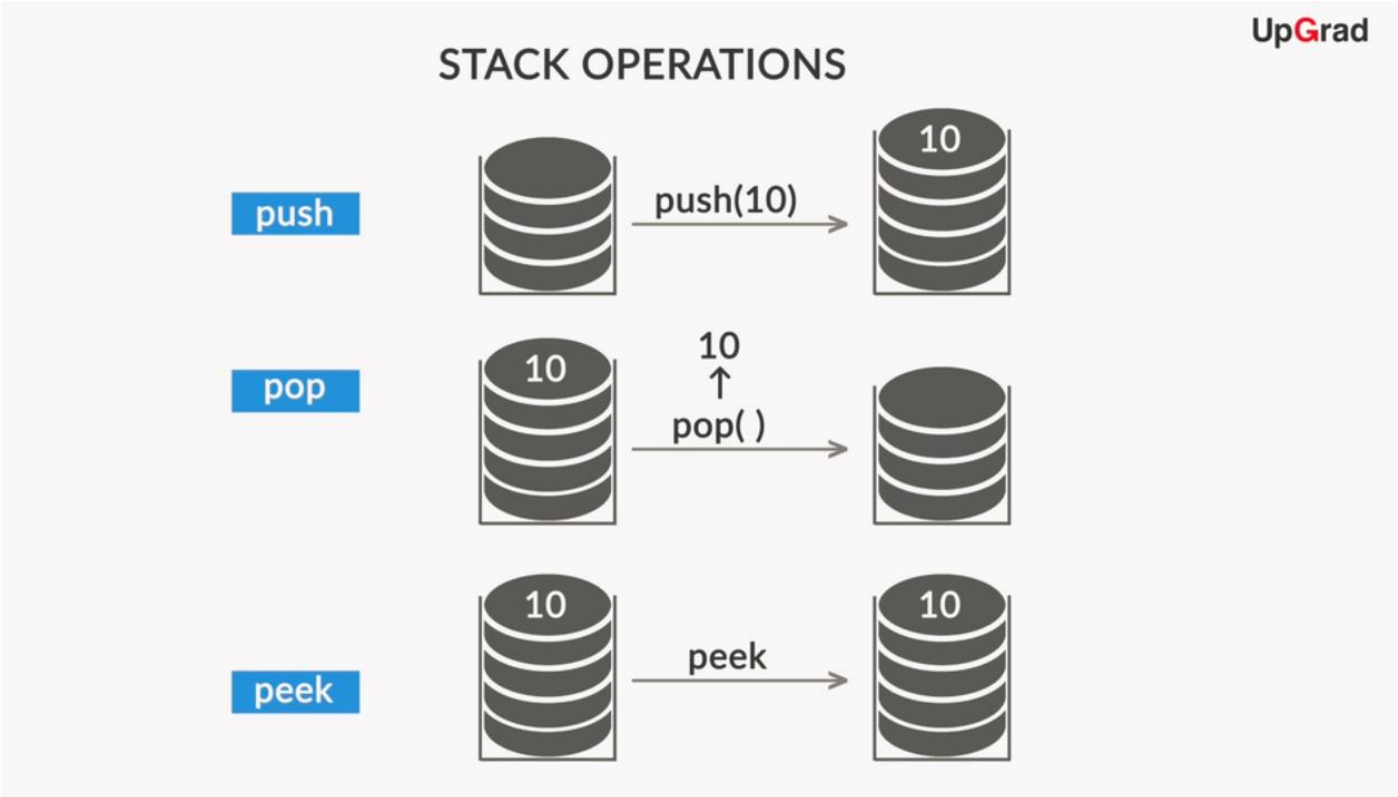
In real life, and in computing, there happen umpteen instances wherein things have to inserted into something and removed from it in a LIFO order. These somethings at least in computer science are called stacks. A stack is therefore anything:

1. into which you insert (push) and remove (pop) things from one end of it.

2. that follows LIFO rule.

Stack Operations

There are four general operations you do on a stack –



Apart from the 3 operations in the image above, there is another operation called – isEmpty(), which returns true if the stack is empty and false otherwise.

There are two kinds of exceptions you could encounter in stacks:

- Underflow: Trying to pop/peek an element from an empty stack
- Overflow: Trying to push an element into a stack that is already at its max capacity

You try to avoid these exceptions by writing checks in your code or by handling exceptions. For the underflow, you saw that a condition was framed, where a pop is not allowed from an empty stack. On the other hand, you learnt that a stack overflow is caused when you've used up more memory for a stack than your program was supposed to use.

Implementing a Stack

You could either implement a stack using a custom class made using lists (array lists or linked lists) such as the one below, where you used a linked list to implement a stack.

In the following program, you added the courses you have done to a stack and then popped them one by one.

```
import java.util.List;
import java.util.LinkedList;
import java.util.EmptyStackException;

public class MyStack<T> {

    public static void main(String[] arg) {
        MyStack<String> stack = new MyStack<String>();

        stack.push("OOP");
        stack.push("Algorithms");
        stack.push("Data Structures");

        try {
            while (true) {
                System.out.println("Popped " + stack.pop());
            }
        } catch (EmptyStackException e) {
            System.out.println("Done!");
        }
    }

    private LinkedList<T> list = new LinkedList<T>();

    public void push(T e) {
        this.list.add(e);
    }
```

```
    public T pop() {
        if (this.list.size() > 0) {
            T e = list.get(list.size() - 1);
            list.remove(list.size() - 1);
            return e;
        }
        throw new EmptyStackException();
    }

    public Boolean isEmpty() {
        return this.list.size() == 0;
    }
}
```

The output for this code was –

```
Popped Data Structures
Popped Algorithms
Popped OOP
```

You can see that the element which was pushed in first was the last one to be popped out, thus demonstrating the Last in, First out order followed in a stack.

You could also use the internal Stack library provided by JAVA, which you could call using -

```
import java.util.Stack;
```

The features push, pop, isEmpty and peek can all be called using this library.

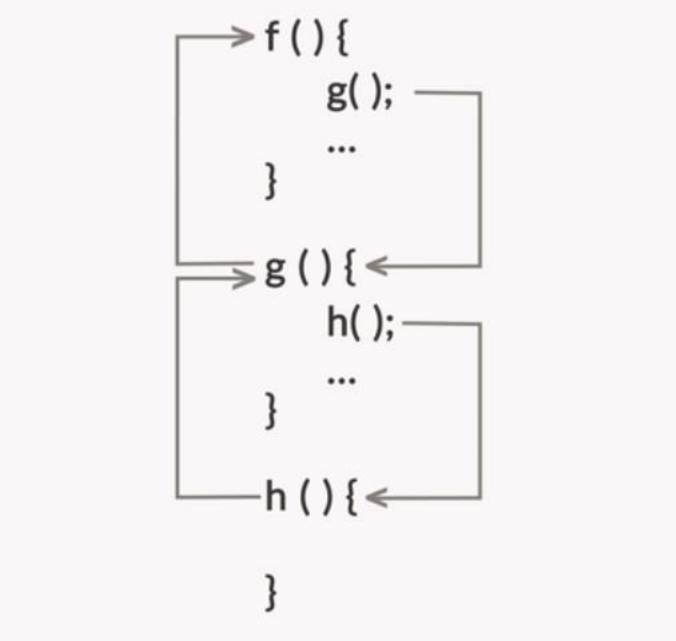
So the above code implemented using this library would have looked like the following:

```
import java.util.Stack;
import java.util.EmptyStackException;

public class MyStack<T> {

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

        stack.push("OOP");
        stack.push("Algorithms");
        stack.push("Data Structures");

        try {
            while (true) {
                System.out.println("Popped " + stack.pop());
            }
        } catch (EmptyStackException e) {
            System.out.println("Done!");
        }
    }
}
```

Industry Applications

You saw the example of how an internet browser implements this stack functionality. Whenever you visit a website, it is added on top of a stack. When you push the back button, the website at the top of the stack is popped off, and you come at the site which was the previous visited website.

You also saw the example of a program stack. If you are given a function which makes call to different functions, it is done with the help of a stack data structure. Lets say you have a function – f() which calls a function g() which in turn calls a function h(). You could visualise using the diagram below:



So, the program stack first finds a reference to the function f(), pushes it into the stack. Then finds the reference to function g(), then pushes it into the stack. Then, it finds a reference to function h(), again pushes it into the stack. When, h() is solved, it is popped from stack. Then g() is solved and popped and finally f() is popped.

Compilers (programs which take your program and turns it into a program in some other language, like machine language or byte code) do a very important thing called parsing. Parsing, in simple language, means reading. A compiler really uses parsing to figure out if a program it's compiling is really well-formed or not. For a program to successfully compile, it's necessary but not sufficient that it should be well-formed. If a program is not well-formed, i.e. if it's ill-formed, the compiler will typically detect that, and will flag a syntax error. Parsing is a complex process. Although, it was beyond the scope of discussion for this session, it is important to note that it uses Stacks for some of its functions. One of the functions is matching Parentheses.

Matching Parentheses Problem

One of the things that must happen for a program to be well-formed is that all its parentheses should match, i.e. if there's an open parenthesis anywhere in the program, there must also be a corresponding closing parenthesis. Obviously, the closing parenthesis must follow, and not precede, its corresponding open paren. For example, "(())" and "()()" are well-formed strings, but ")(" is not. So, having an equal number of open and close parentheses is necessary but not sufficient for a string to well-formed. You saw the following approaches to solve this problem.

Algorithm - Matching Parentheses using a *count* variable

1. Initialise variable *count* to 0.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever the algorithm scans an opening parenthesis, increase *count* by 1.
4. If the algorithm scans a closing parenthesis, decrease *count* by 1. However, if *count* is already 0, it means error; so return false.
5. If the algorithm reaches the end of string, and *count* = 0, it means that the string is well-formed; so return true. Otherwise, return false.

Algorithm - Matching Parentheses using Stack

1. Initialise stack *S* to be empty.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open parenthesis, push '(' into *S*.
4. If you meet a closing parenthesis, pop a '(' from *S*. However, if *S* is already empty, it means error; so return false.
5. If you reach the end of string, and *S* is empty now, it means that the string is well-formed; so return true. Otherwise, return false.

The counter based approach fails when we consider multiple parenthesis. So, you modify the counter based approach to have two counters, one tracking parenthesis and the other tracking braces. The problem with this algorithm is that it will correctly accept all well-formed strings and will correctly reject some of the ill-formed strings, however, it will fail to detect strings where both the parentheses and braces are individually matched, but their relative positions are messed up, e.g. "({)}".

The stack algorithm when you consider multiple types of parentheses would be -

1. Initialise stack *S* to be empty.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open parenthesis, push '(' into *S*. If see an open brace push '{' into *S*.
4. If you meet a close parenthesis, pop a symbol from *S*. If it's not an open parenthesis (e.g. if it's an open brace), return false. Also, if *S* is already empty, it means error; so return false.
5. If you meet a close brace, pop a symbol from *S*. If it's not an open brace (e.g. if it's an open parenthesis), return false. Also, if *S* is already empty, it means error; so return false.
6. If you reach the end of string, and *S is empty*, it means that the string is well-formed; so return true. Otherwise, return false.

File Versioning System

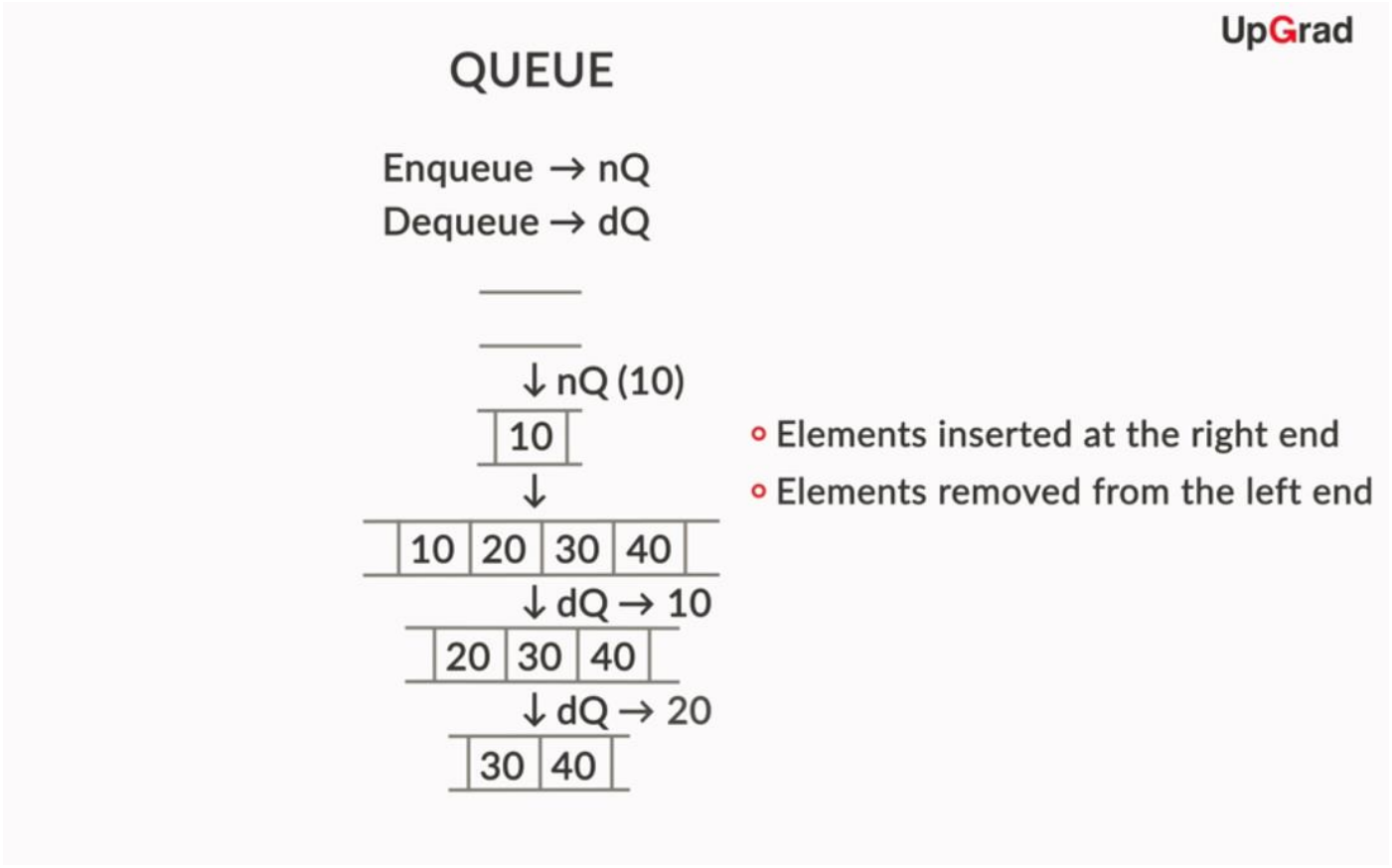
You then saw how to implement a file versioning system, like the one you see in MS Word or Google Docs. Stacks are best suited for this purpose. Whenever you make changes to a file lets say – documentVersion1, and you save the file, the new file gets stored as documentVersion2 and is pushed on top of the stack. Now when you have to undo your changes, you call, pop() which reverts to the previous version. You could go back multiple versions to restore an older version, or could see the no. of versions etc..

Queues

Stacks are data structures where elements follow **Last in, first out** order. You could add and remove objects from the same end, called the top of the stack.

Objects in Queues follow **First in, first out** principle. You could relate these with queues you see in movie theatres, where, the person who gets in first in the line is the first to get the ticket. Similarly, you see examples of queues in other booking systems as well.

Analogous to a push in stack, there is an enqueue in Queue. Similarly, analogous to pop in stack, there is a dequeue in Queue. You could visualise these operations through the image below.



Queues find use in various computing processes, such as scheduling printing jobs, scheduling system processes such as maintenance etc..

Implementing a Queue

Unlike Stack, which is a class in Java, Queue is an interface that often needs to be implemented as a linked list. You learnt about interfaces in polymorphism. They are classes that contain only abstract methods and cannot be instantiated. In Java, unlike Stack, Queue is an interface, which needs to be implemented by a class (usually the Linked List class) before you can instantiate and use it.

You can see the following implementation of the Queue. Where you initially add some tasks to the queue and then remove them.

```
import java.util.Queue;
import java.util.LinkedList;

public class ToDoList {
    public static void main(String[] args) {
        Queue<String> todolist = new LinkedList();
        makeToDoList(todolist);
        doAllTasks(todolist);
    }

    public static void makeToDoList(Queue<String> todolist) {
        todolist.add("task 1");
        todolist.add("task 2");
        todolist.add("task 3");
        todolist.add("task 4");
    }

    public static void doAllTasks(Queue<String> todolist) {
        while (todolist.size() != 0) {
            System.out.println(todolist.remove());
        }
    }
}
```

The output for the code above was –

```
task 1
task 2
task 3
task 4
```

You can see that the tasks were printed(removed) from the queue in the same order in which they were added.

Booking System using Queues

Ticket booking systems heavily use queues for their processes. You saw the implementation of such a system using Queues. When an individual would make a booking request, it would be stored in queues, so that the customer who makes the request first, gets tickets first. Another, important feature in the system was that a request could be accepted only if there were sufficient tickets in the system. So, if a customer is making a request for 4 tickets, but there are only three tickets in the system, this request would have to be denied. Thus, each time a request is made, the tickets available are checked. If the tickets are available, then the request is dequeued and the next request in the queue is processed.

More Applications

You would see the implementations of stacks and queues in Graphs as well, where you would be performing the breadth-first and the depth-first searches.

Lecture Notes

Unit Testing, TDD and Refactoring

Welcome to session on Unit Testing.

As you have been through the life cycle of Software Development, this module is about the essential part of the Software Life Cycle which comes after the development, that is testing.

Unit Testing is a level of software testing where individual units/ components of a software are tested.

In this Session, the following fundamentals of Software Unit Testing had been covered:

- What is a Software Unit
- What are Unit Test Cases
- Characteristics of Good Unit Test Cases
- Testing in JUnit
- Tags and Assertions in JUnit

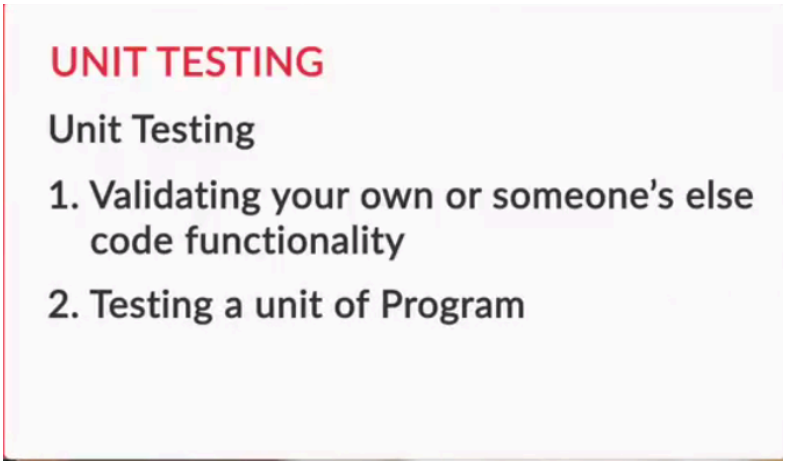
What is Unit Testing and What is a Software Unit

The term Unit Testing comprises of two words: Unit and Testing. Testing as you know is validating your own or someone else's code to check for the functionality.

A unit in Software Testing can be:

- A method
- A sequence of methods
- A class
- A sequence of classes

Any method that is public facing or deals with the outside world data can be thought of as a Software Unit.

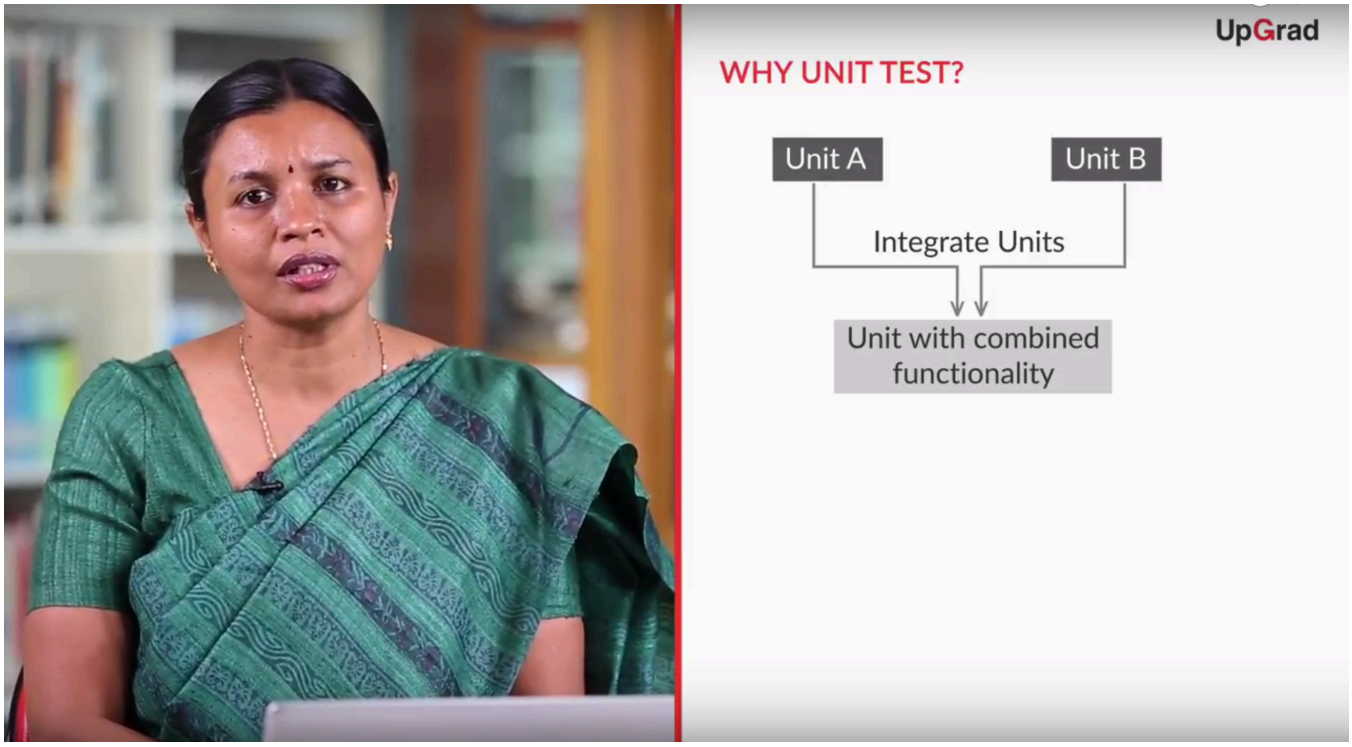


A UNIT OF PROGRAM

1. Method
2. Class
3. Sequence of Classes

Why Unit Testing

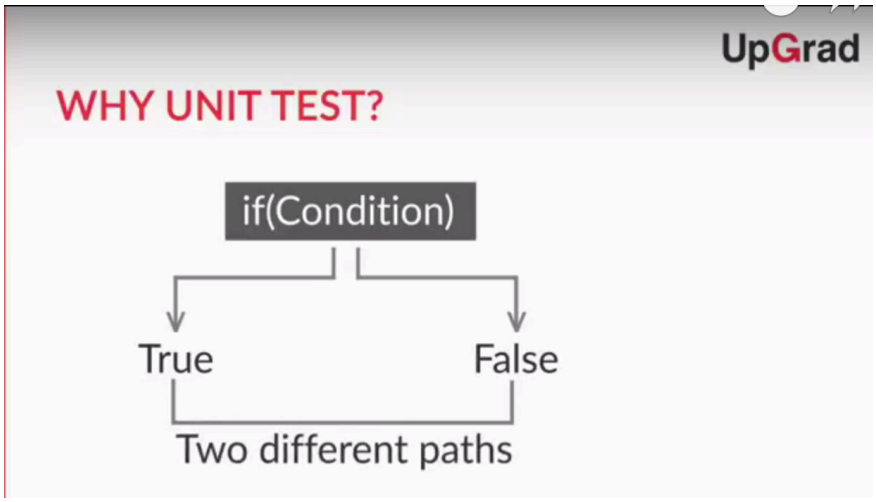
The certain question to ask here is why do we unit test, what is the benefit that it provides us with.



Let's say two teammates are working on a collaboration project and each of them is working on their individual units that needs to be combined together in the end to provide the desired functionality, then it is obviously going to save the overhead if the two individuals have been independently tested so as to simplify the process of integration of both the units. So, here Unit Testing plays a key role.

In Unit Testing, the important thing to check is that we consider each and every possible path a program might take, so as to give a certain assurance that the software would perform absolutely fine even under boundary conditions.

Let's say in case of an if condition, the tester needs to check for both if the condition is true and if the condition is false, so as to be certain whether the code is ready to be deployed or not.

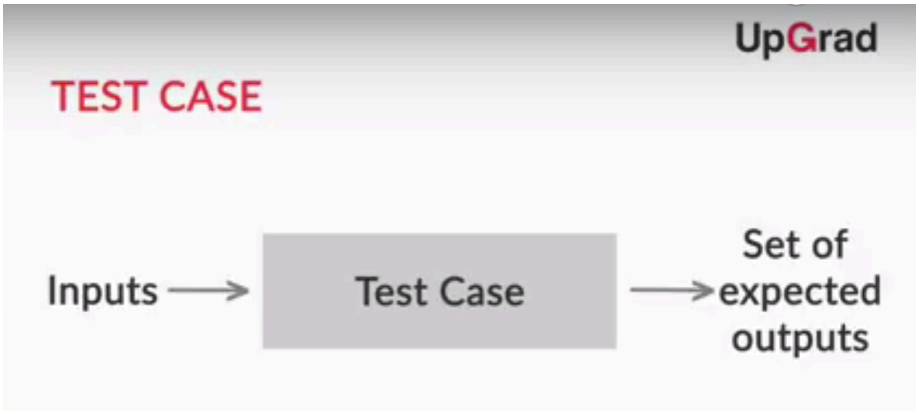


And in case of a for statement, we need to consider the end points of the loop during testing because this is where the unexpected behaviour happens.

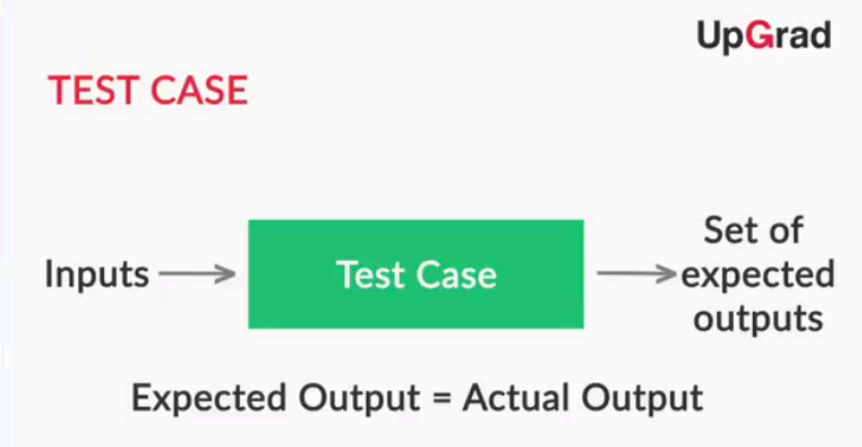
Now, it might be clear that to check if the desired functionality of the software is being fulfiled or not, we employ unit testing to test for the various units of the program.

Unit Test Cases

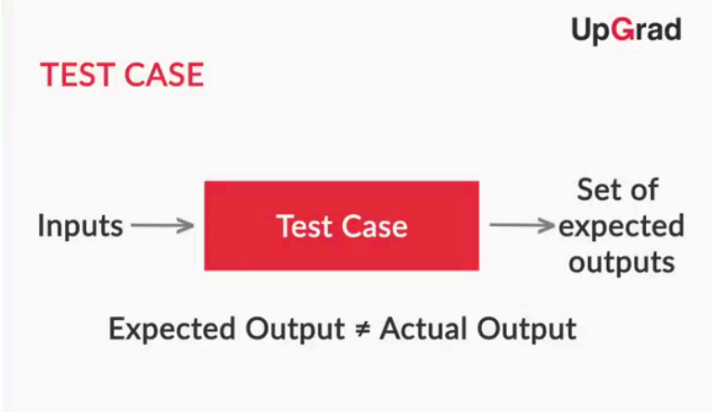
A test case can be thought of as a box which takes in inputs and gives a set of expected outputs which is then matched with the actual output of the code under test.



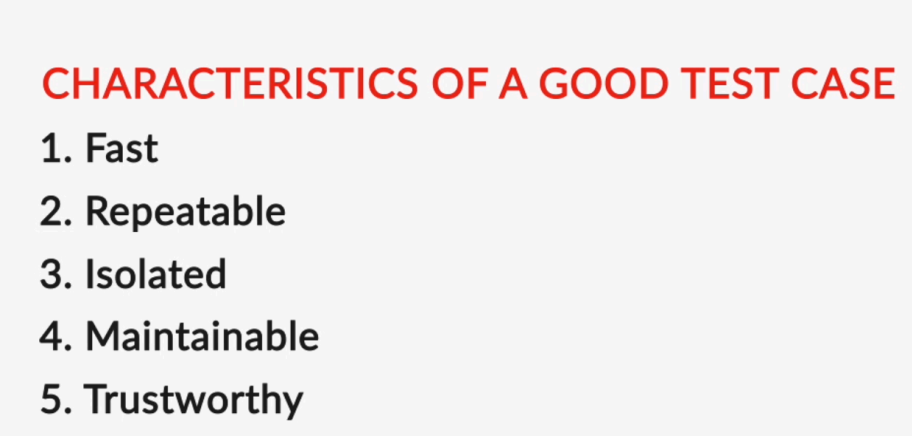
If the expected output matches the actual output, the test case is said to have passed.



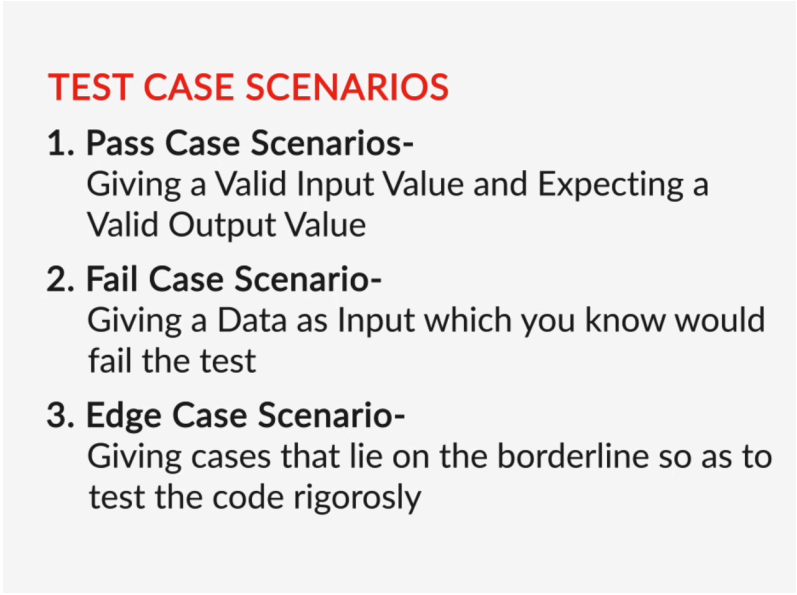
But if the expected output doesn't match the actual output, the test case is said to have failed.



Test Cases are the backbone of Unit Testing and creating good test cases is a really healthy practice in the process of Unit Testing. So, to create Good Unit Test cases, following factors should be taken into consideration:



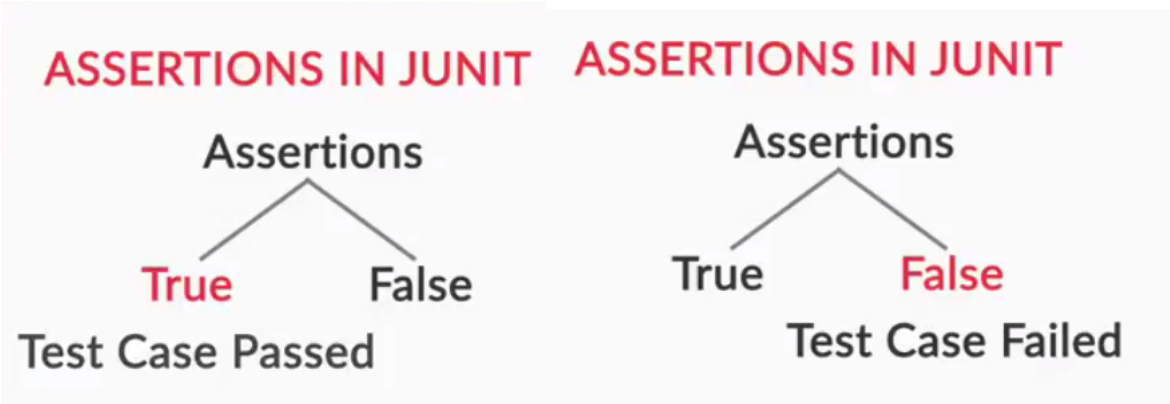
While testing a software, the test cases checks for certain scenarios or it can be said that the test cases are written to check for the following scenarios:



Assertions in Unit Testing

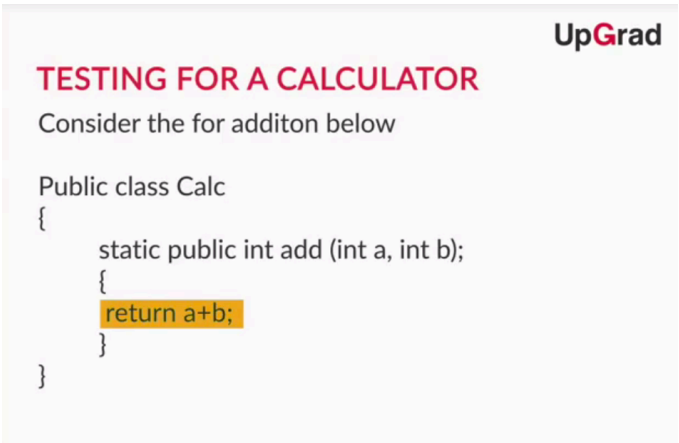
Now, you know about unit test cases, the obvious thing to ask is how to write test cases and determine whether they pass or fail, so we make use of methods known as Assertions.

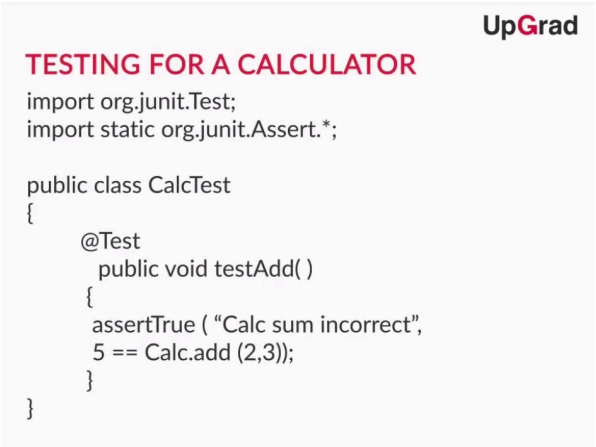
Assertions in Unit Testing are the methods which are used to determine the pass or fail status of a unit test case.



Assertions are the methods which are used to check for test case passing/failing behavior. If the result of assertion comes out to be true, then the test case is believed to be passed and if the result of assertion is false, then the test case is believed to be failed.

Underneath you see a sample code for a calculator which does addition of two numbers and there is also a test method written to check if the calculator program is working as it should or not.





The assertion assertTrue() checks if the boolean condition that has been passed to it is true or not, if it is true, then the assertion returns true, else the assertion returns false.

There are mainly four types of assertions that we have used.

- **AssertEquals():** Takes in two parameters and passes if the objects passed to it are equal.
- **AssertNull():** Takes in a single paramter and passes if the object passed to it is null.
- **AssertNotNull():** Takes in a single paramter and passes if the object passed to it is not null or something is contained by the object.
- **AssertTrue():** Takes in a boolean object and passes if the object value is true, else it fails.

Assertions are one of the A's of Unit Testing. Unit Testing is a process which consists of three A's

The three A's or AAA of unit testing stand for:

Arrange: This is the first step of a unit test application. Here we will arrange the test, in other words we will do the necessary setup of the test. For example, to perform the test we need to create an object of the targeted class.

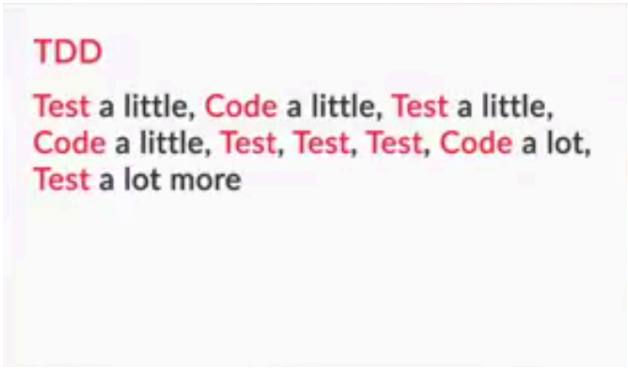
Act: This is the middle step of a unit step application. In this step we will execute the test. In other words we will do the actual unit testing and the result will be obtained from the test application. Basically we will call the targeted function in this step using the object that we created in the previous step.

Assert: This is the last step of a unit test application. In this step we will check and verify the returned result with expected results. Use of Assertions is in the Assert part of Unit Testing.

```
[TestClass]
public class UnitTest
{
        [TestMethod]
        public void TestMethod()
        {
        //Arrange test
                testClass objtest = new testClass();
                Boolean result;

        //Act test
                result = objtest.testFunction();

        //Assert test
                Assertions.assertEqual(true, result);

        }
}
```

Test Driven Development

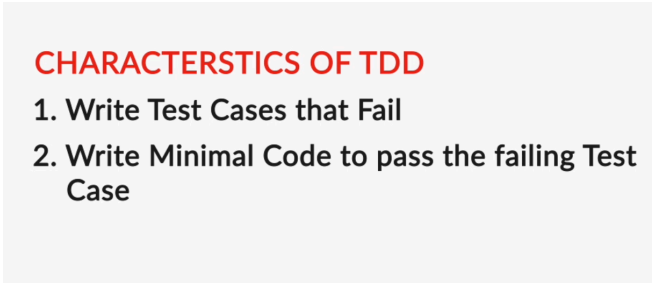
Test Driven Development or TDD as we call it is a process of software development in which testing and coding go hand in hand or in other words, we could say that in TDD, there is equal importance given to testing as is given to coding.



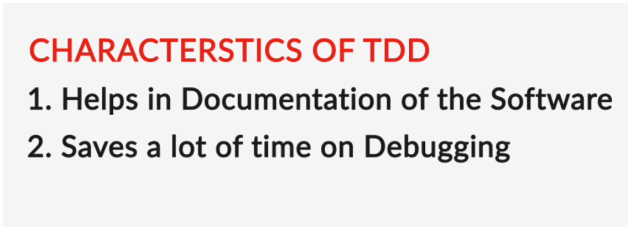
The above figure depicts TDD in a crisp manner, “We test a little, then code a little, test,code,test,code and the process goes on until we have a final deployable software”.

In TDD, development follows the order:

1. Write Test Cases that fail.
2. Write minimal code to pass that failing test case.
3. Refactor the code.



Benefits of TDD:

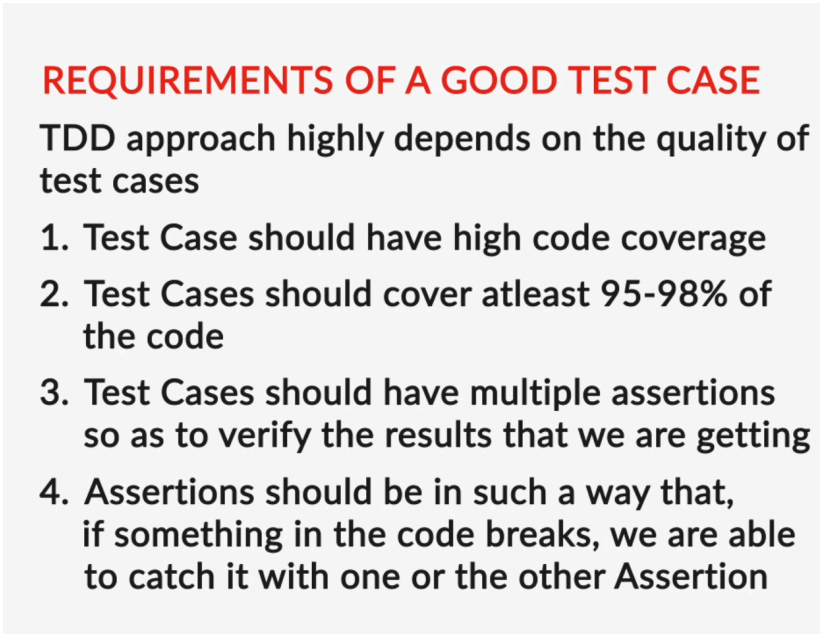


As we follow TDD, we sort of create a pseudo Documentation of the Software since we keep on writing test cases even before writing the code, so it helps in the latter stages of development since it already creates a set of test cases for the code. In this manner, it also saves the time on debugging later since the code that we write is already tested.

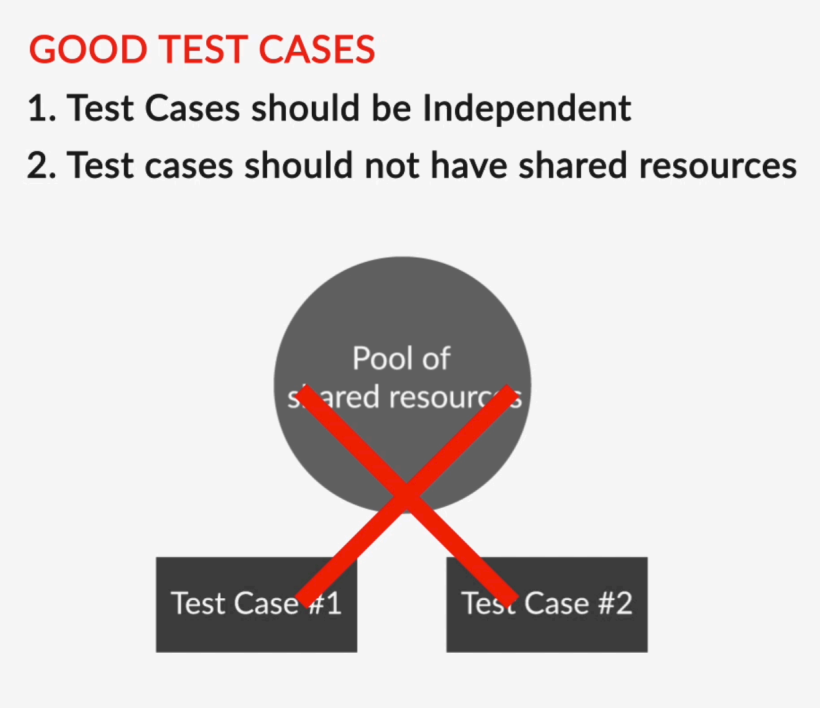
Requirements of a Good Test Case

Test cases are the integral part of the whole TDD process and a lot depends on the quality of test cases written.

So, the test cases should be written keeping these things in mind:



There are also certain things that we ought to keep in mind while writing the test cases that they shouldn't be part of the test case that we write so as to improve the quality of the test case.

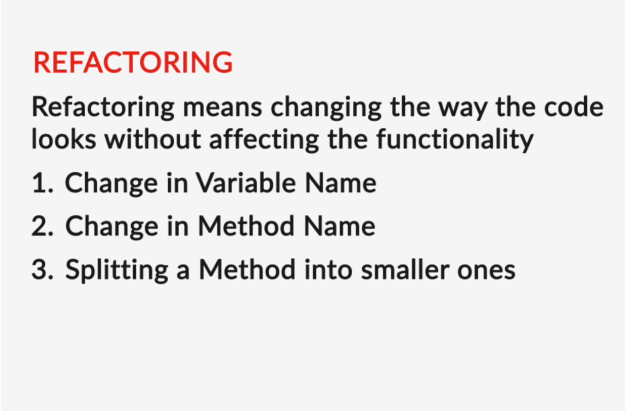


Since TDD is all about writing test cases and then coding to pass that test case, so the overall quality of test case needs to be maintained and the above points are what need to be considered while writing test cases.

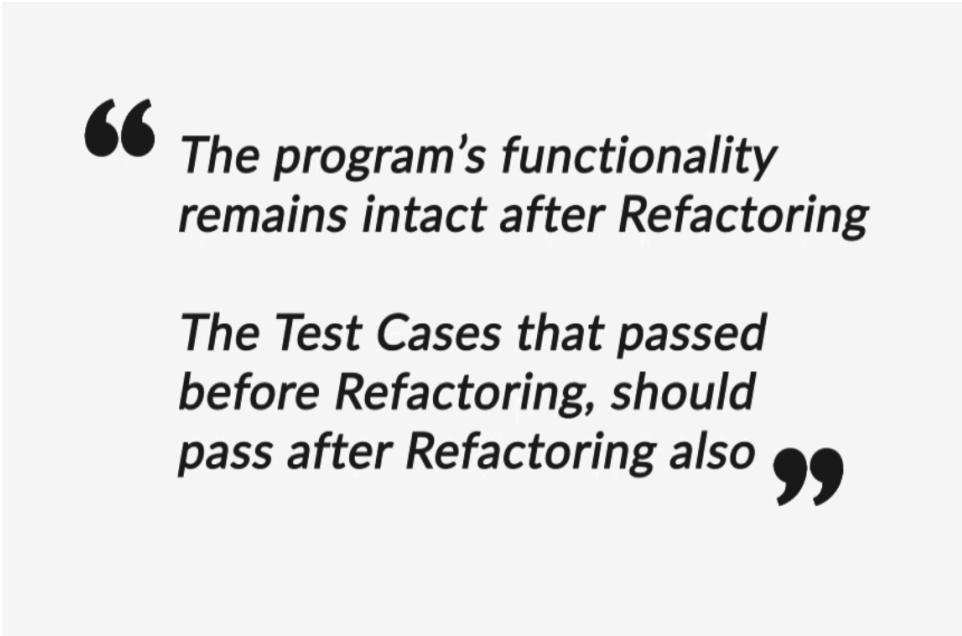
Refactoring

Refactoring is a term that you saw as being a step in the process of TDD. In this TDD, we actually end up writing bad code or code that smells. So to improve the quality of code that we write, we do Refactoring.

Refactoring in simple terms means changing the way the code looks without affecting the functionality of the code.

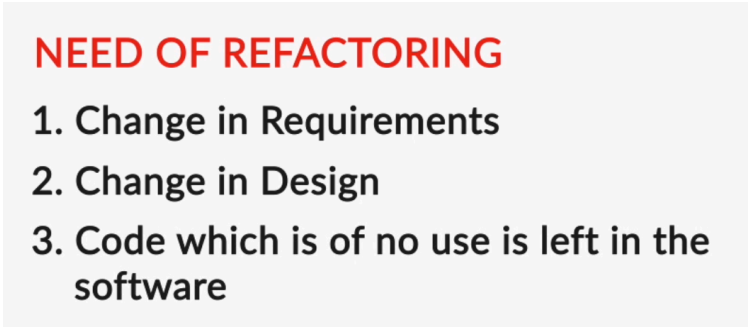


An important thing to keep in mind while employing Refactoring is that:



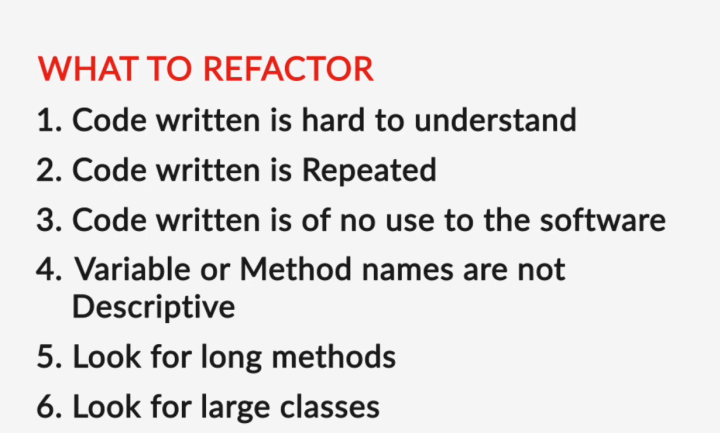
Need of Refactoring and What to Refactor

An interesting thing to ask here is what is the need of Refactoring or why do need Refactoring at all. The following figure depicts where exactly do we find the need of Refactoring:

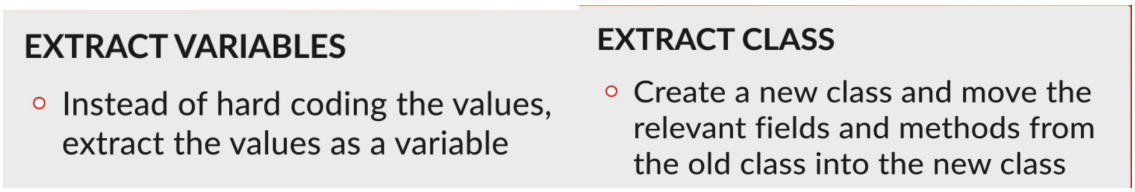


There are certain cases in software companies when there is a change in requirements of the client or there is change in the design of the software, there we need Refactoring. Sometimes, there is some code which is of no use left in the software, there also we need to remove that extra piece of code which is yet another way of Refactoring.

Now, the next thing is what to Refactor which is shown in the below figure:



You saw two particular ways to Refactor namely: Extract Variable and Extract Class.



When a certain value let's say 100 is being used at various places in the code to depict a maximum value, so rather than putting 100 everywhere, we can replace 100 with a variable let's say temp_Max=100 and use that temp_Max everywhere in the code, so this practice of replacing the value with a variable is what is known as Extract Variable.

Assume a class let's say calculator which consists of methods such as add, subtract, multiply and divide. Apart from these basic operations, the class also consists of certain scientific operations such as log, sqrt,etc. which can be realised as part of a scientific calculator, so we can move out these methods from the calculator class since they would be irrelevant to a basic calculator, this process is known as Extract Class Refactoring.

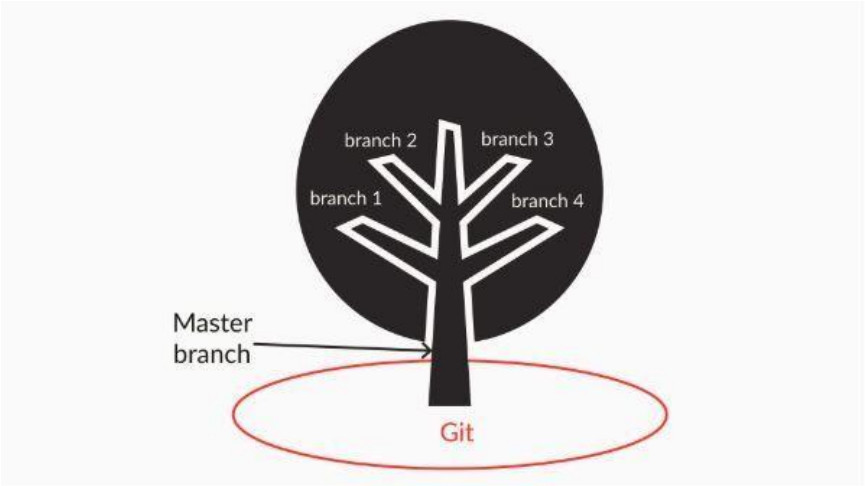
This brings an end to the module of 'Unit Testing,TDD and Refactoring'.

Lecture Notes for Version Control Part - ||

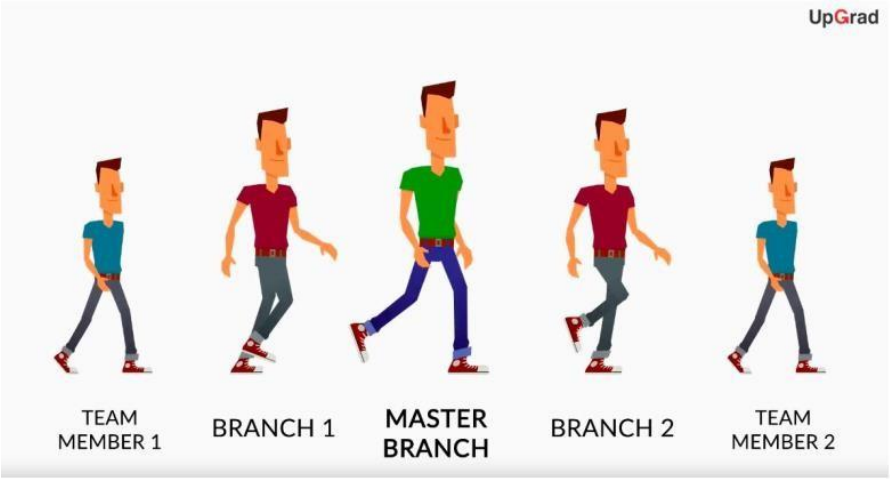
Branching :

When you come across the term branching, you might correlate it to branches of a tree.

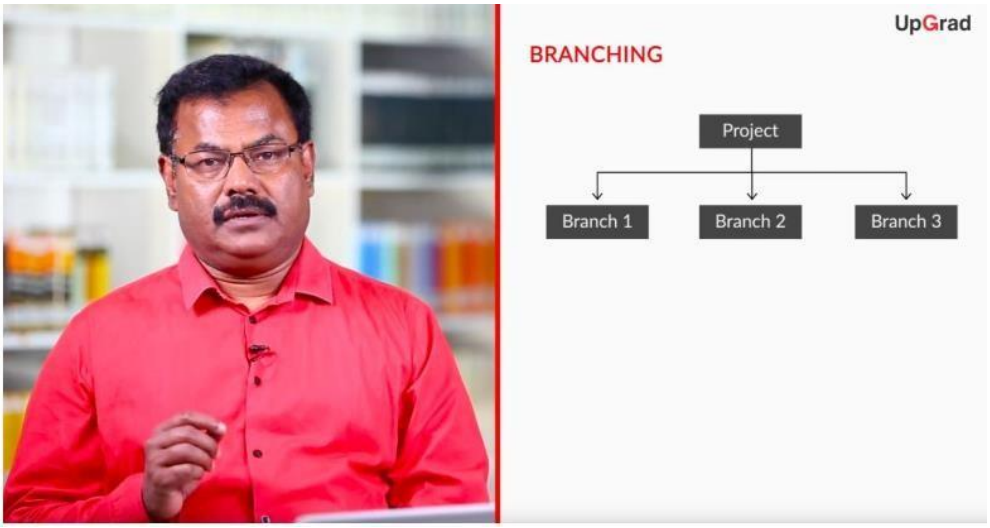
Well, yes! Branching means exactly the same. Imagine the branches growing out of the trunk of a tree.



The trunk here plays the role of a master branch, and the branches coming out of the trunk represents the branches in git.



You can have your own copy and work on that, make modifications to it, and if the changes look fine you can merge them back to you master branch



Steps for branching and merging :

Working With Branches :

In this video, you learnt about the following steps:

- Creating branches
- Viewing the created branches
- Working with different branches concurrently

In this video, you learnt about the following commands:

- **git branch <branchname>**: This command will create a branch with the given branch name
- **git branch**: This command will show you all the branches along with the HEAD pointing to the branch you are currently working on
- **git checkout <branchname>**: If you want to move from one branch to another, you can run this command

In-Depth Study of the Concept of Branching

A branch in git is simply a lightweight, movable pointer that points to one of the commits. The default branch name in git is master, which points to the last commit. Every time you make a commit, the master moves forward to that commit.

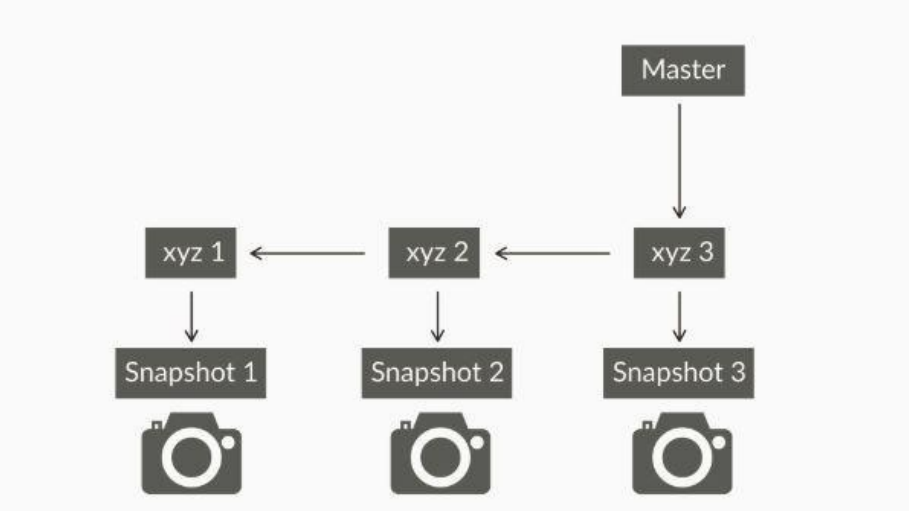


Figure 1. Here the master (branch) is pointing to our last commit

Now if you create another branch, what do you think will happen? When you create a new branch, git will create a new pointer at the same commit you're currently on.

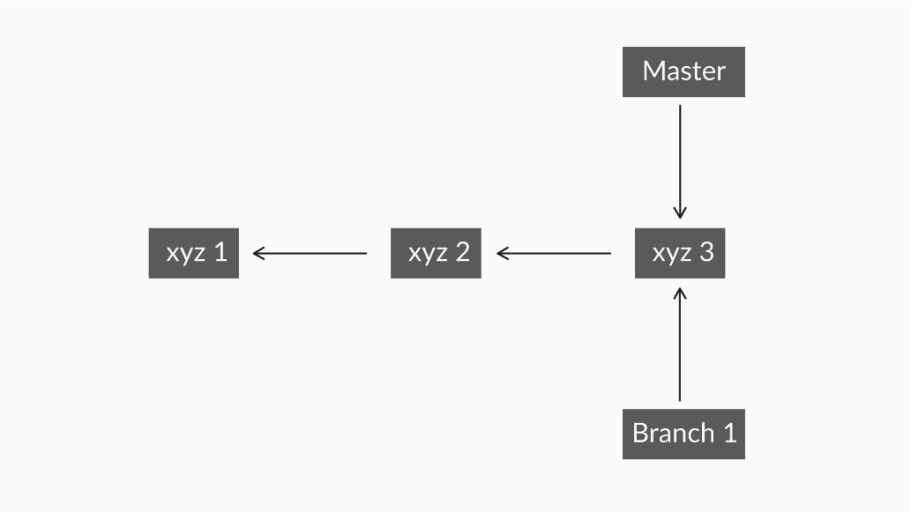


Figure 2: A new pointer at the same commit is created

Now, how do you think Git knows what branch you're currently working on?

Git keeps a special pointer called the HEAD. It points to the branch you are currently working on. To make the HEAD point to the new branch, i.e. Branch1, you will have to switch to that branch — if it didn't switch to that branch automatically.

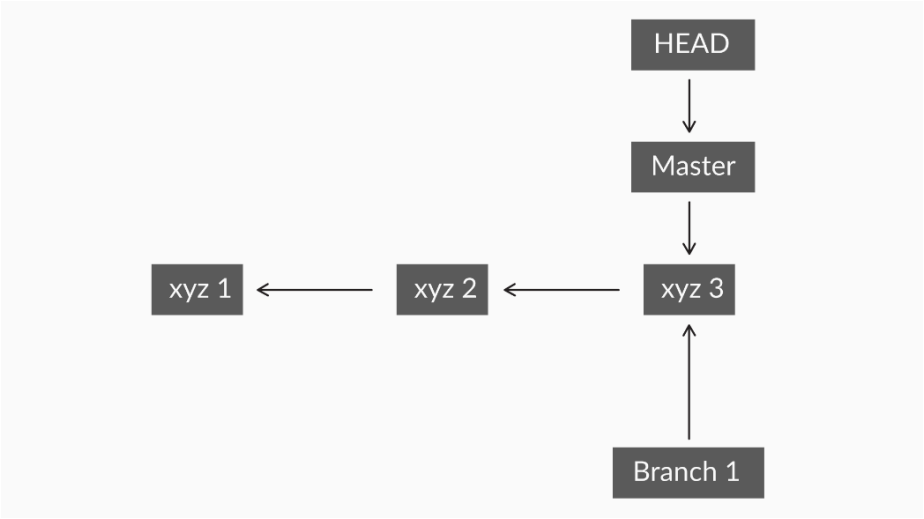


Figure 3: The HEAD points to the branch you're working on

To switch to Branch1, you need to run the command:**git checkout Branch1**
This will now move the HEAD to point to your newly created branch, i.e Branch1

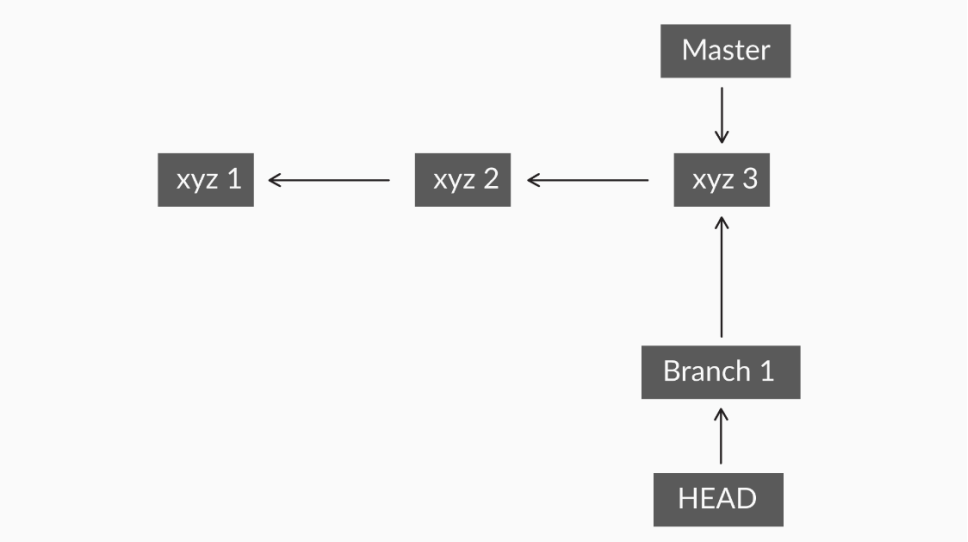


Figure 4: The HEAD points to branch1 when you switch the branches

Working with branches – II

In this video, you learnt about —

- **Merging branches**: For merging, you can use the command **git merge <branchname>**
    - Note: This command will merge the changes in the branch <branchname> with the branch that you are currently working on. Merging can happen between all the branches. Imagine that you have three branches, namely —
        - Master
        - Branch1
        - Branch2

      You can merge any one of the branches above with another one.

- **Deleting branches**: You can use the command **git branch -d <branchname>**

More about the **git diff** command.

The **git diff** command: This command is used to show the changes performed between commits. The main objective of version control is to enable you to work with different versions of the same file. Hence, git provides the command 'diff' to allow you to compare between the different versions of your files. The most common scenario where 'diff' is used would be when you need to see what changes you had made after your last commit. Ways in which we can use the 'git diff' command:

- **git diff commitid1 commitid2**: To see the difference between two commits using their commit IDs
- **git diff branch_name1..branch_name2**: To see the difference between two different branches. Here, 'branch_name1' represents the branch you are currently working on

- **git diff**: This will show you the difference between Git data sources(data sources be commits,branches,files etc).

In the following image, notice that there are two branches connecting multiple boxes:

- The red line represents the master branch
- The grey line represents another branch named 'branch1'
- The rectangular boxes represent the commits and the different commit IDs

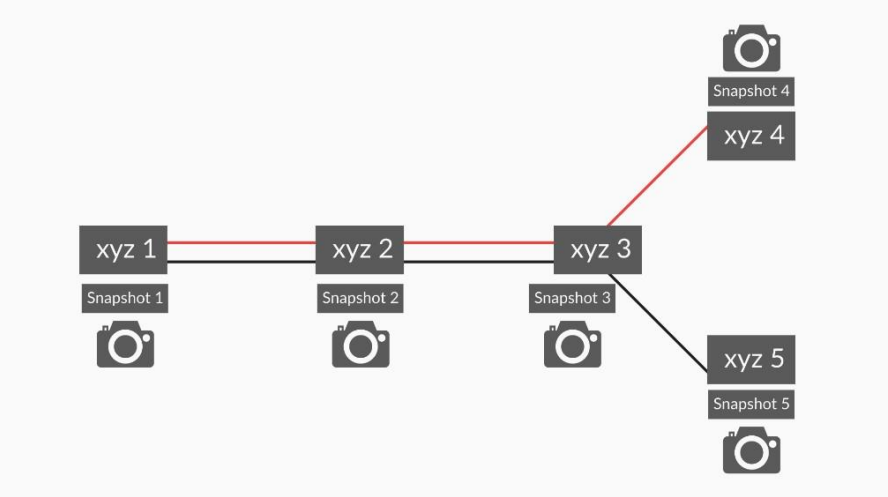


Figure 1: A depiction of git branches and commits

Now, suppose that you need to merge the commit 'xyz 5' on branch1 with the master branch. On merging of the two branches, it creates a new commit. At this point, git will take a snapshot of the changes made. After merging, the new commit would be named 'xyz 6', as shown in the following diagram:

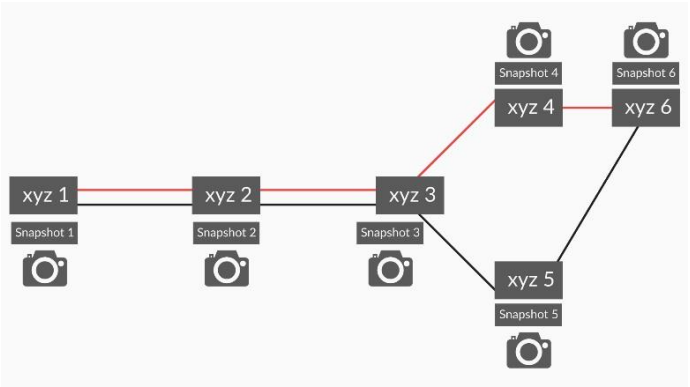


Figure 2: The git branches after merging

Now, once the commit 'xyz 5' is merged, if you want to work on 'branch1', you can keep making commits to it, e.g. commit 'xyz 7'.

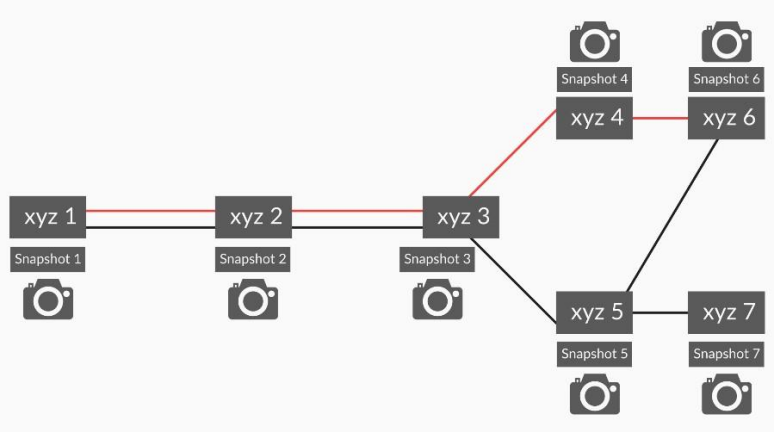
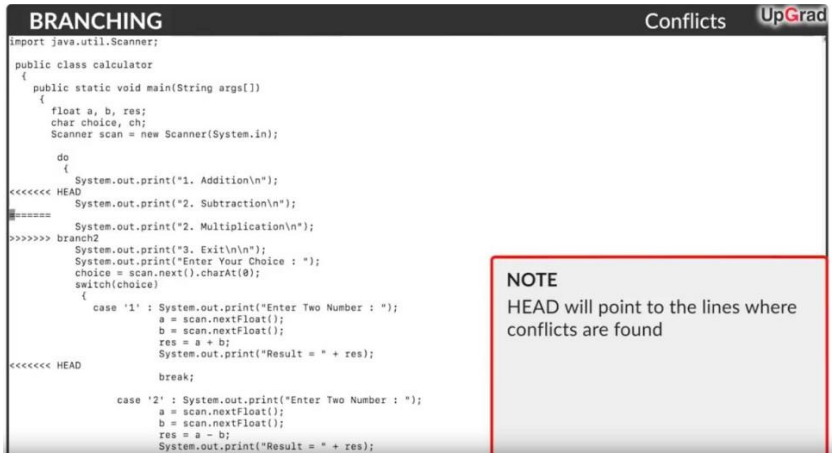


Figure 3: The git branches after the commit 'xyz 7'

Managing Conflicts :

Merging branches in git isn't always as easy as it may look. If two people change the same line of a code at the same time, don't you think git will get confused?

Definitely! Let's look at how you can help git in such a situation.

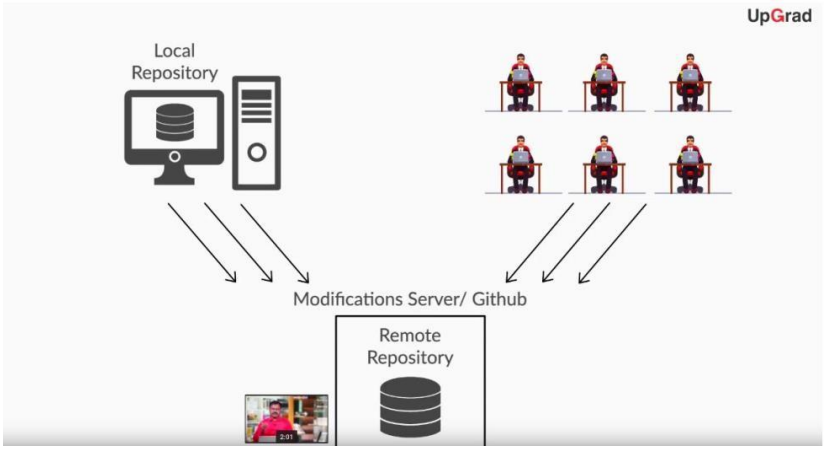


The best workaround to deal with a merge conflict is to use your best decision, that is —

- You can keep all the changes by making necessary modifications to the files where the merge conflict happens
- Or, simply discard some changes that cause the merge conflict

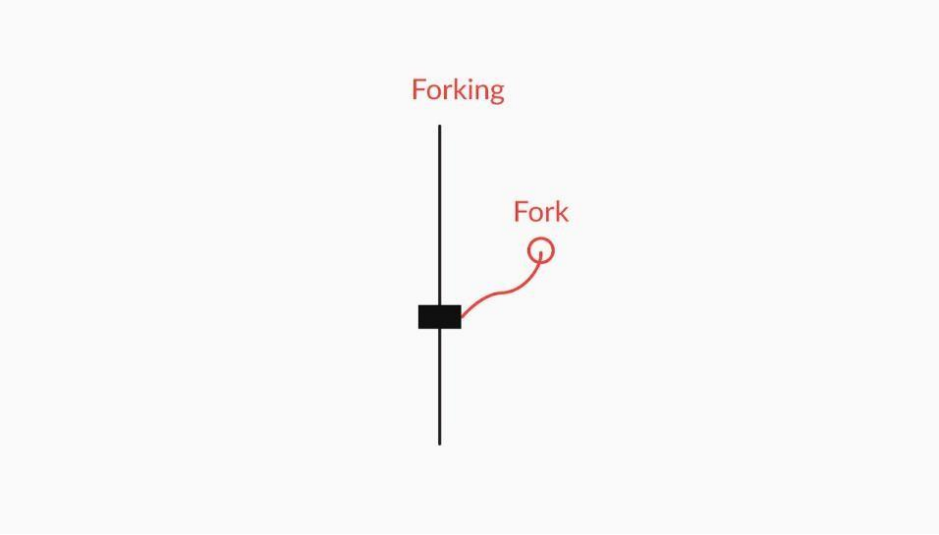
Collaboration :

Collaboration helps you work with the people around you and produce something even better than before. You can add features or improvise some of the features in someone else's project who is sitting miles away from you.

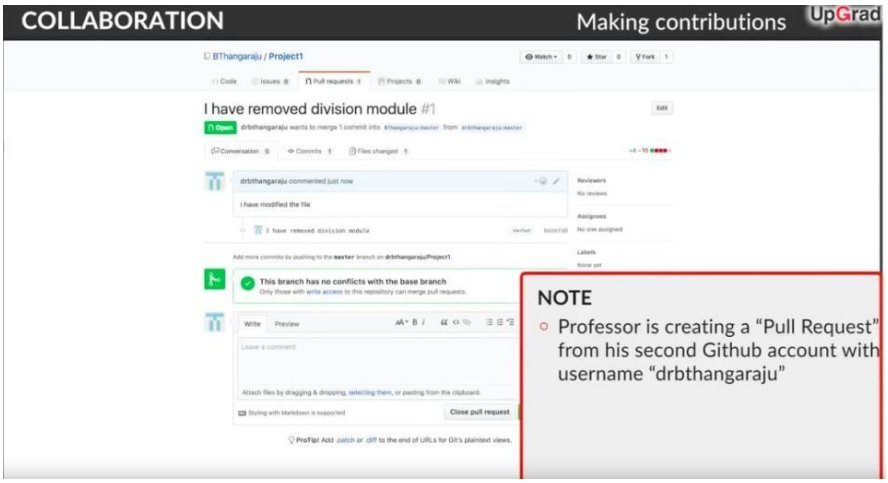


To contribute to someone's project we need to follow the steps below-

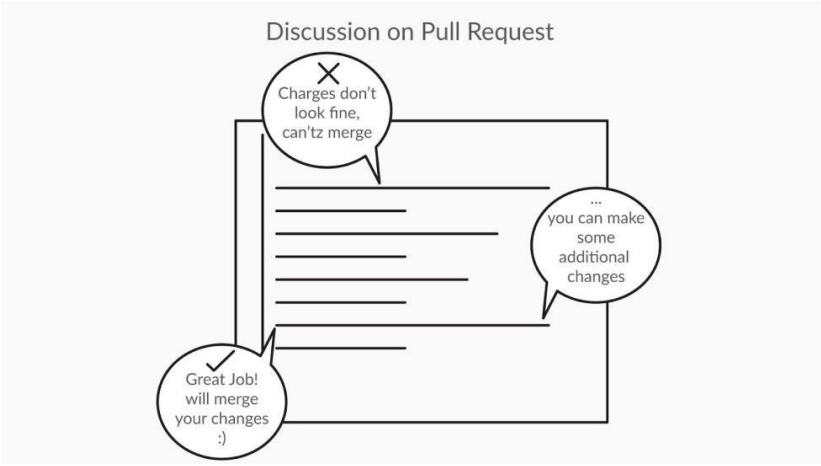
- To contribute to someone's project we first need to **fork** it. Forking will get you that project on your github account.

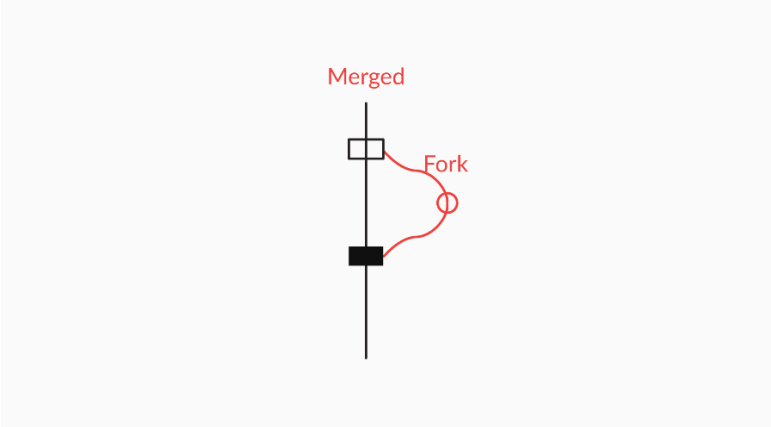


- Now to bring that project to your local system you need to **clone** it.
- Next you will make modifications and push the changes back to your Github account.
- Now to inform the owner of the project about new changes that you made you will create a **pull request**



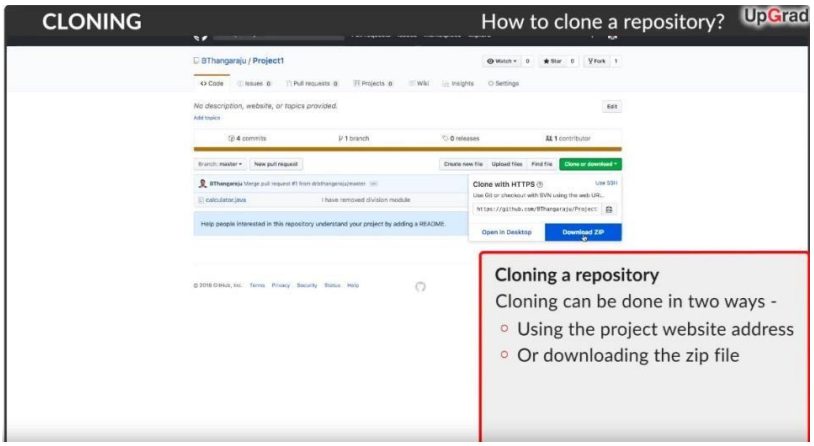
- There will be discussion on your changes





Cloning :

Cloning means making an identical copy. This video will help you clone a github project and bring it to you local machine



The two ways of cloning are:

- Cloning from your GitHub account
- Cloning from the command line using commands such as —

'git clone url' (of the git repository)

Lecture Notes: Object Oriented Analysis And Design

1.1 Introduction to OOAD and UML

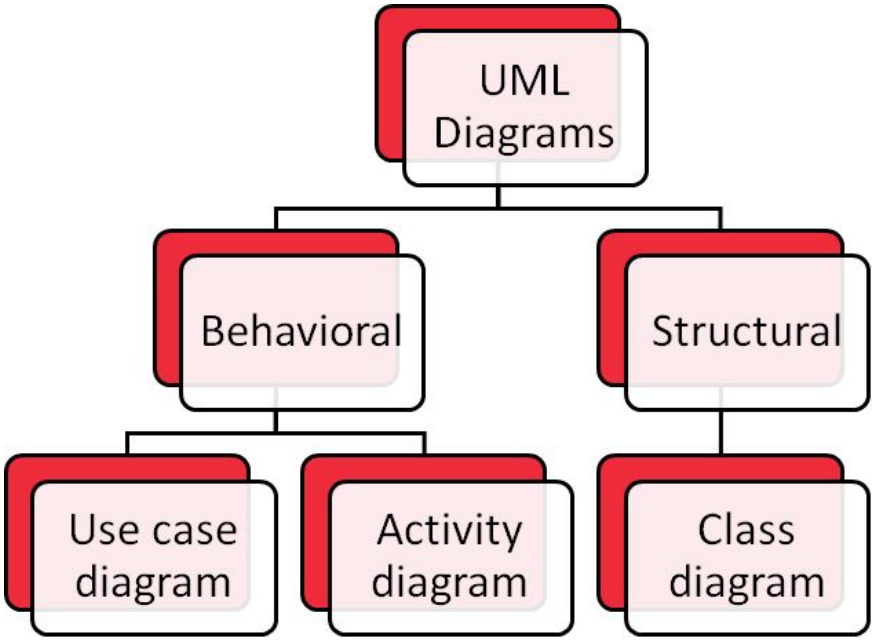
UML: Unified modeling language is a common language for all software developers to visualize the system and communicate with their peers for developmental and business purposes.

OOAD: Object oriented analysis and design is a methodology which has two parts: The analysis and the design. The analysis is done during the requirement gathering part, where you work with the users of the system or the software to define the functionality of the system. Post analysis, you design the system by refining the analysis models created during the analysis phase.

Why OOAD:

1. To keep all stakeholders on the same page
2. Help the product evolve
3. Transfer knowledge (documentation)

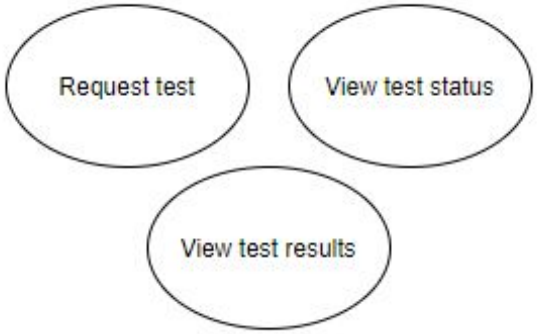
Object oriented programming: Attributes are properties and methods define the actions of an object.



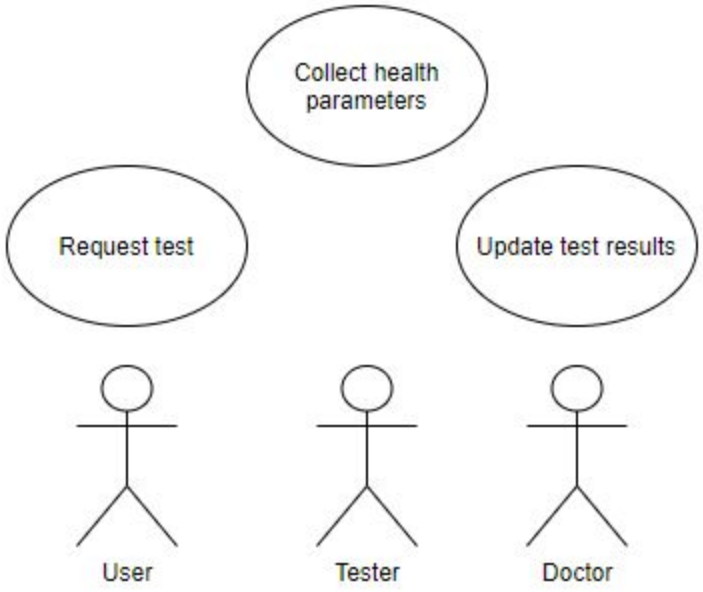
1.2 Use case diagram

Use case diagram is a type of behavioral diagram and is used for requirement gathering.
The following are the different elements of the use case diagram:

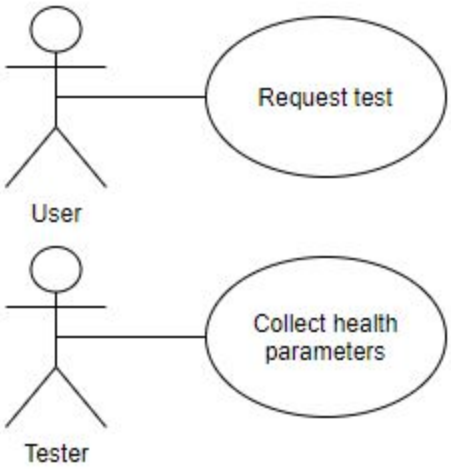
1. **Use cases**: Use cases are functionalities or features that a software will provide. It can be thought of as the answer to the question: *What will my software be used for?*
   Use cases have no particular order.



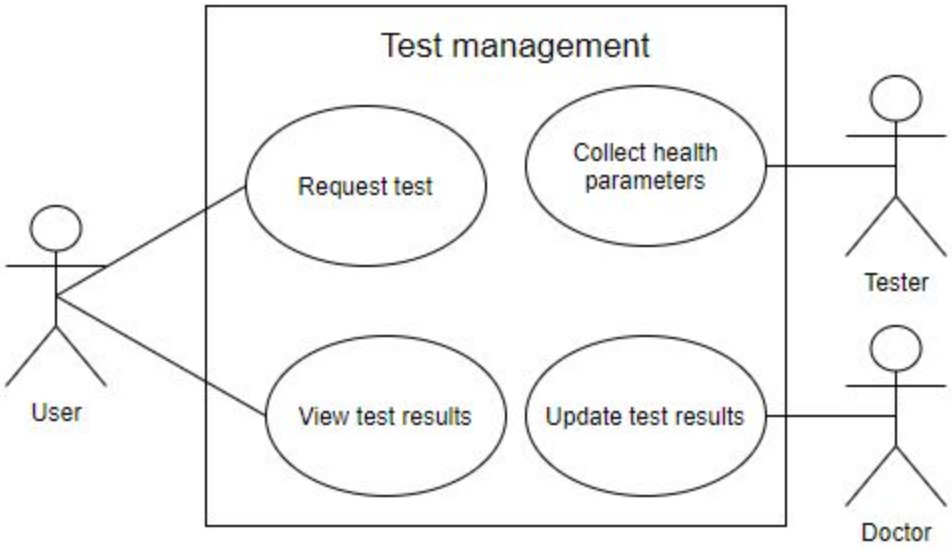
2. **Actor:** Actors are the external entities which invoke the use cases. Actors may or may not be humans. They can also be a 3rd party service (eg. Notification system). Actors are represented by stick figures.



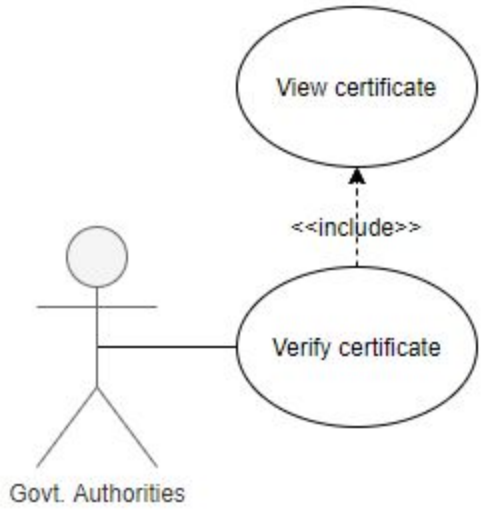
3. **Associations:** Actors are related to use cases by using associations. The associations are depicted by drawing lines between actors and use cases. One use case can be associated with multiple actors and one actor can be associated with more than one use case.



4. **System boundary:** A system boundary is drawn around the use cases, to separate the use cases from the actors. This indicates that use cases are internal to the system and actors are external to the system.

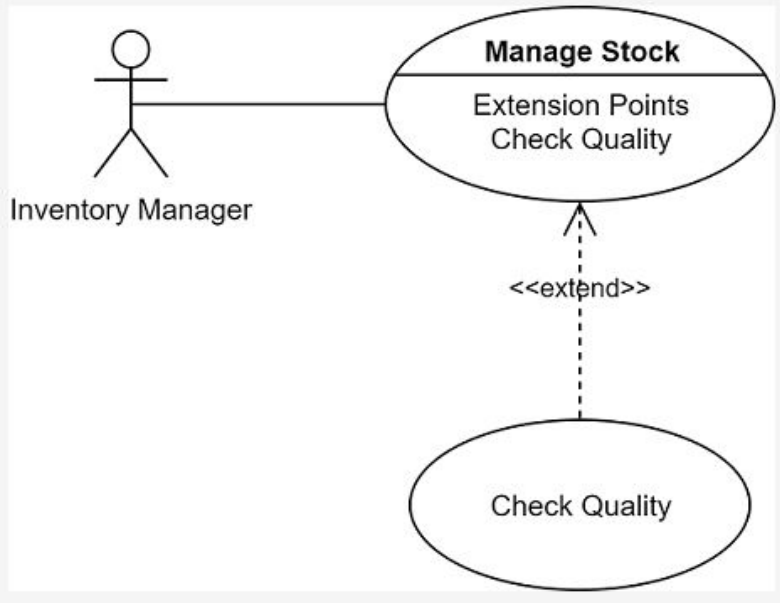


5. **Include:** Include relationships can be used to represent use cases which are connected with each other. In an include relationship, the arrow starts from the including use case and points towards the included use case. Before an actor can execute the including use case, they have to first execute the included use case.



In the above example, the Government authority has to view the certificate before he/she can verify it. So before executing *verify certificate*, which is the including use case, the government authority has to execute *view certificate*, which is the included use case.

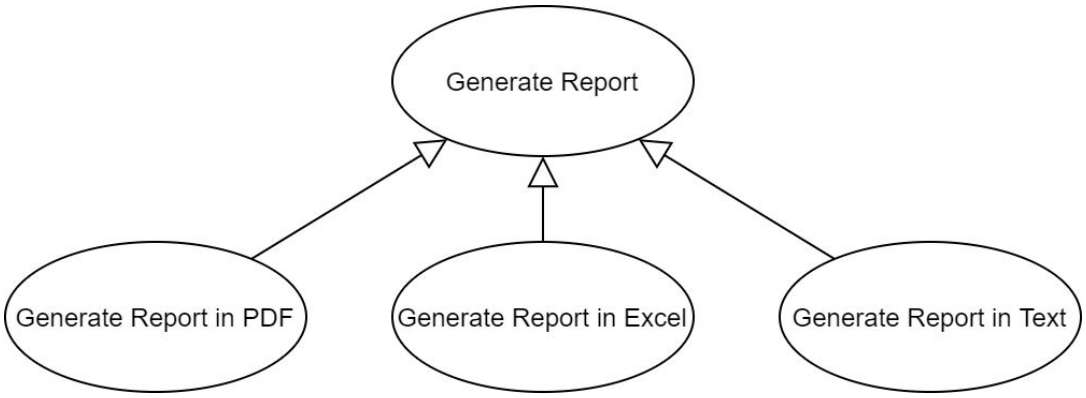
6. **Extend relationship:** In an extend relationship, the arrow starts from the extending use case and points towards the extended use case. Before an actor can execute the extended use case, they may or may not first execute the extending use case.



In the above example, whenever the inventory manager is managing the stock using the *Manage Stock* use case, they can first check the quality using the *Check Quality* use case, but it is not mandatory.

7. **Generalisation relationship:** In a generalisation relationship, the common functionalities of different use cases can be clubbed under a parent use case.

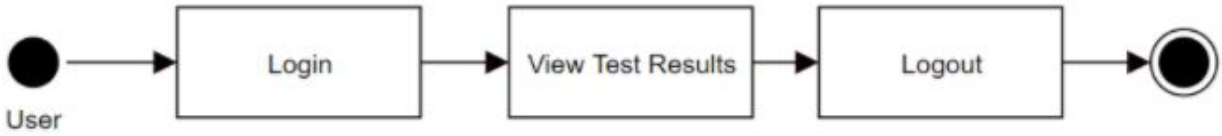
Other use cases that modify the behaviour from the parent use case are called children use cases.



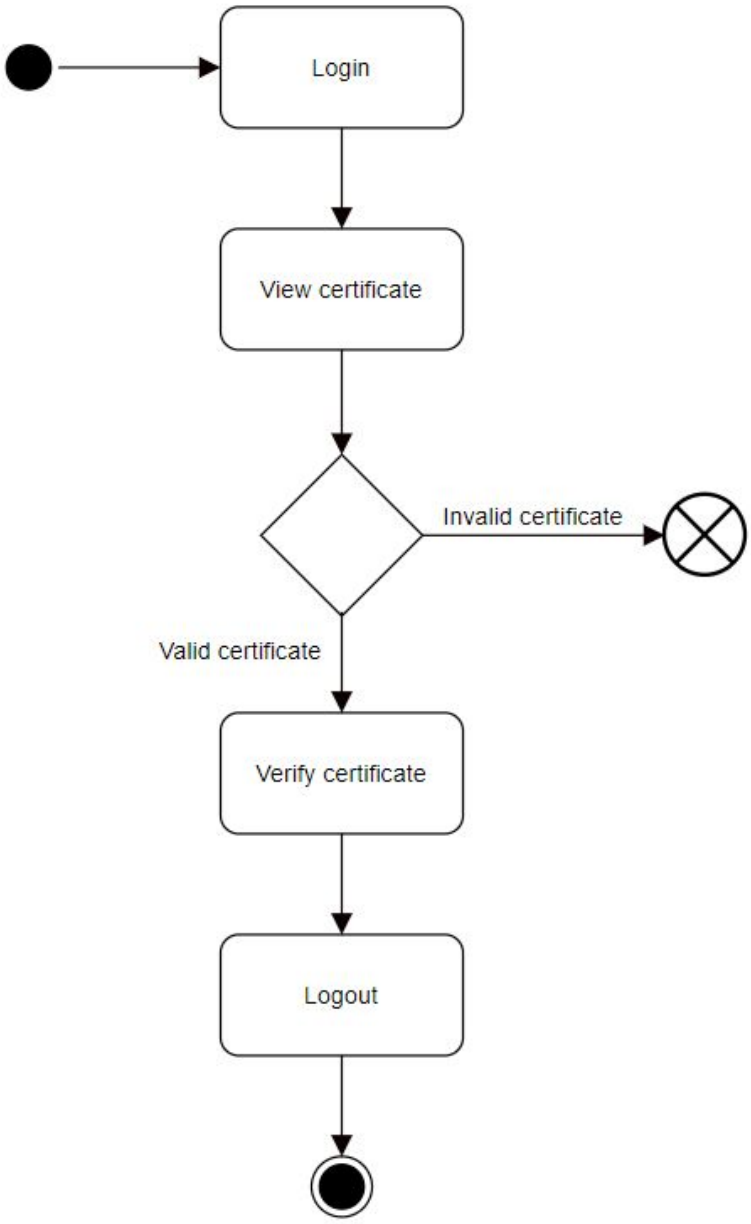
1.3 Activity diagram

Activity diagrams are pictorial representations of use case diagrams. They contain additional information, like the order of execution of use cases and also certain constraints on these use cases. The following are the elements of the use case diagram:

1. **Initial node, actions and activity final node**: The initial node is where the flow begins. It is represented by a solid circle. Actions are single units of behaviour performed by the system. The final node is where the flow ends and is represented by a solid circle inside a hollow circle.

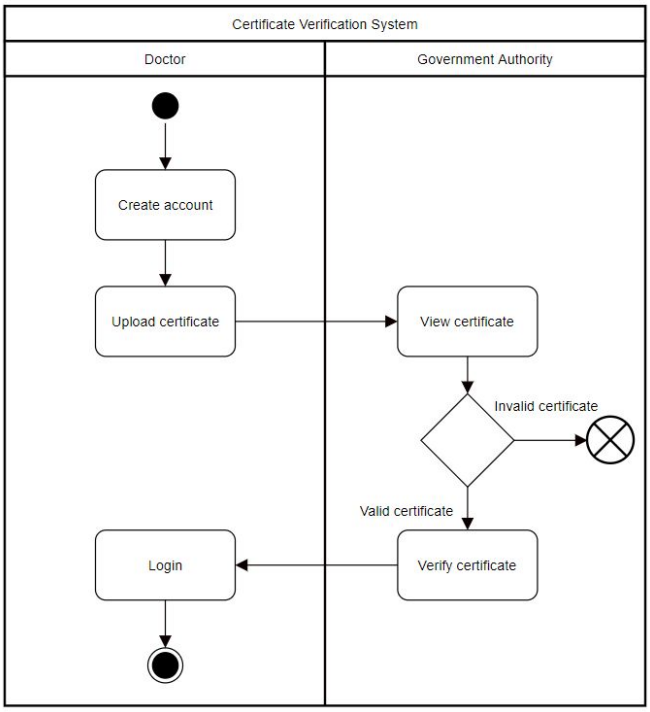


2. **Decision and merge node:** This is similar to an if-else condition, where the activity flow takes different routes for different conditions. The decision node is represented by a diamond, which has a single incoming flow and multiple outgoing flows. The direction of an outgoing flow is decided on the basis of the guard conditions.

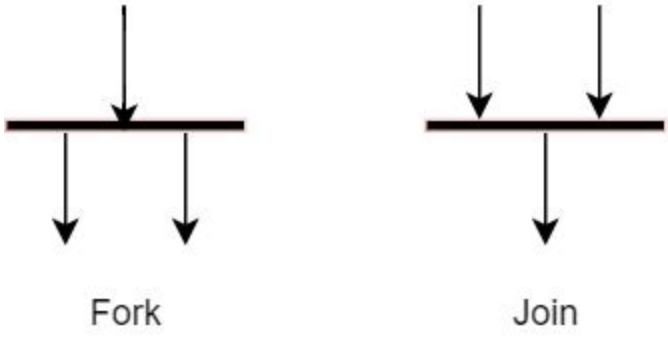


In the above example, the guard conditions are *Valid certificate & Invalid certificate*. When *Invalid certificate* is true, the flow terminates. If *Valid certificate* is true, the flow continues.

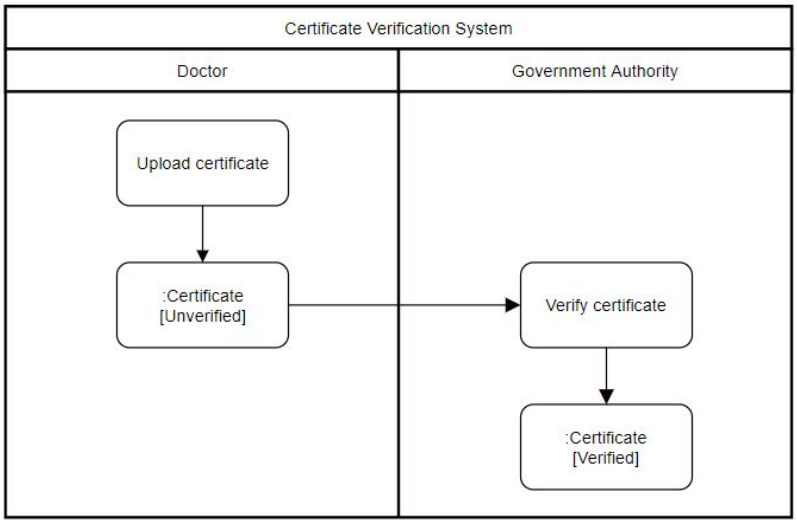
3. **Partitions:** Partitions are used to distinguish between various responsible parties and group the actions performed by the same responsible party.



4. **Fork and Join:** Fork is similar to a decision node with one input flow and multiple output flows. The difference is that, in the case of Fork all the outflows happen concurrently. Joins are the inverse of forks, with multiple input flows and a single output flow.



5. **Object flow:** Object flow is a representation of how a particular object gets updated after an activity. They are



In the above example, the object *Certificate* takes two states: *Unverified* and *Verified*. It takes the *Unverified* state before it is verified by the Government authority and takes the *Verified* state after it is verified by the Government authority.

1.4 Class diagram

A class diagram is used to show what classes will be present in the system and how they will be related to each other. Class diagrams represent the attributes and methods present inside classes and the relationships between the classes themselves.
The following are the different elements in a class diagram:

1. **Class, attributes and methods:** The class name is written on top. The box contains two partitions with attributes on the top partition and the methods on the bottom partition. The attributes are mentioned with the type. The methods may contain parameters and have a return type. The following template is an example of how classes are represented in class diagrams.

ClassName
attribute1: type attribute2: type
method1(parameter1 : type) : returnType method2(parameter1 : type) : returnType

2. **Visibility:** Visibility (also called access specifier) specifies how attributes and methods of a class will be accessed by different classes.Visibility is mainly of two types:
   a. Public (+) = Attributes and methods that are marked 'public' can be accessed from within or outside that class.
   b. Private (-) = Attributes and methods that are marked 'private' can be accessed only from within that class.

Order
+ orderId : int + orderName : String + databaseAddress : String
+ getTotalCost(productName : String, quantity : int) : int + checkStatus() : String - checkDatabase() : String - checkWithSupplier() : String

3. **Multiplicity:** Multiplicity is used to represent an array of objects.The syntax for multiplicity is *ClassName [minValue..maxValue]*

Order
+ name : String + userId : String + address : String - inventoryManagers : InventoryManager [5..20]
+ addInventoryManager() : void + updateInventoryManager() : void + removeInventoryManager() : void

4. **Default values:** Some of these attributes can have a default value, which can be represented as *attributeName : type = defaultValue*.

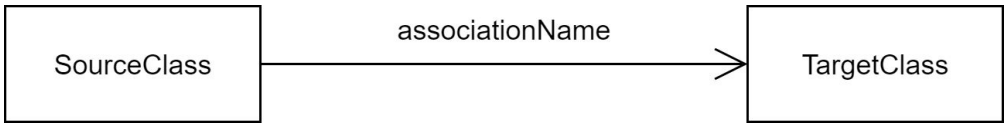
Order
+ name : String + userId : String + address : String = "Inventory Address" - inventoryManagers : InventoryManager [5..20]
+ addInventoryManager() : void + updateInventoryManager() : void + removeInventoryManager() : void

5. **Property strings:** Attributes can also have properties. This can be represented as *attributeName : type = {PropertyString}*.

Order
+ name : String + userId : String = {unique} + address : String = "Inventory Address" - inventoryManagers : InventoryManager [5..20]
+ addInventoryManager() : void + updateInventoryManager() : void + removeInventoryManager() : void

Apart from the elements of the class, class diagrams also represent the relationship between classes. There are 4 types of relationships:

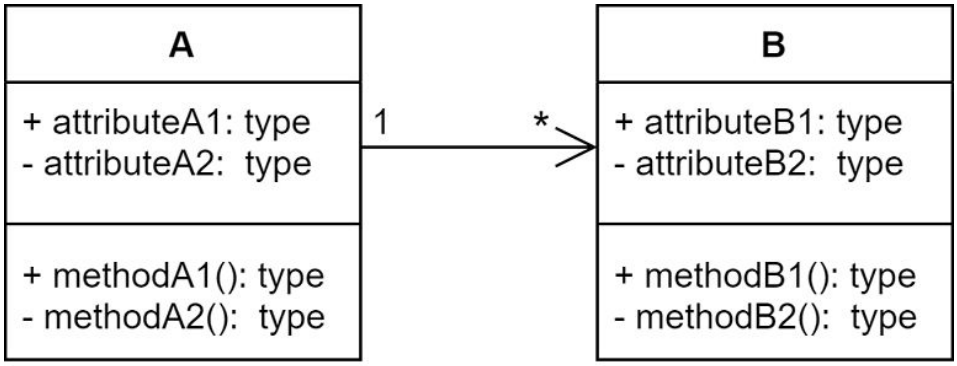
1. **Association relationship:** Association relationships are of two types: Unidirectional (as shown below) and bi-directional.



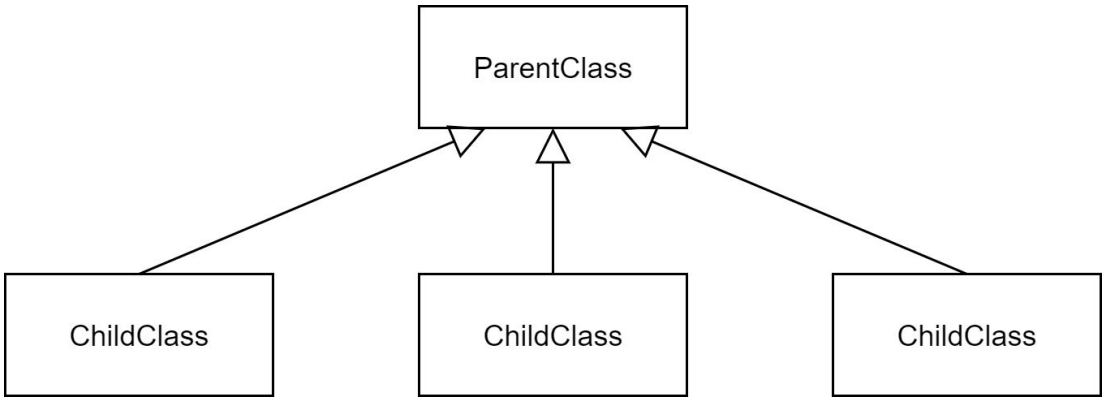
Association relationships can also have multiplicity associated with it, which is of 3 types:

- **a.** One-to-one
- **b.** One-to-many
- **c.** Many-to-many

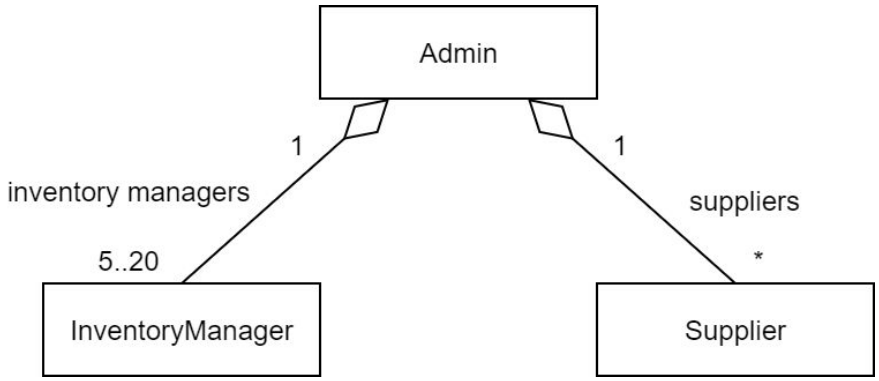
The following is an illustration association relationship between classes which is one-to-many.



2. **Generalisation Relationship:** A generalisation relationship has a parent class and multiple children classes. Using a generalisation relationship, you can put common attributes and methods of several classes into one general class, which saves you from repeating the same code and also allows code reusability.

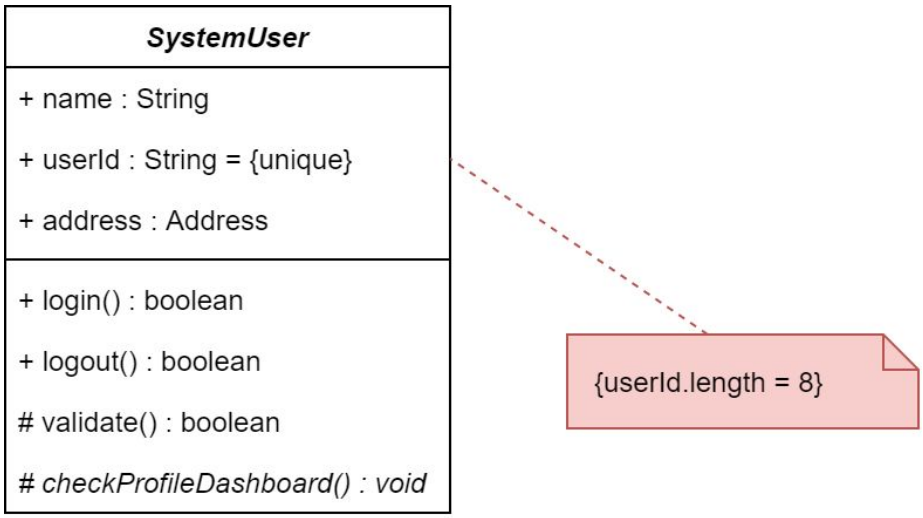


3. **Aggregation:** In an aggregation relationship, an object of one class (whole object) will contain objects of other classes (part objects). But when the whole object is deleted, part objects will not be deleted. An aggregation relationship has an association name and multiplicity associated with it.

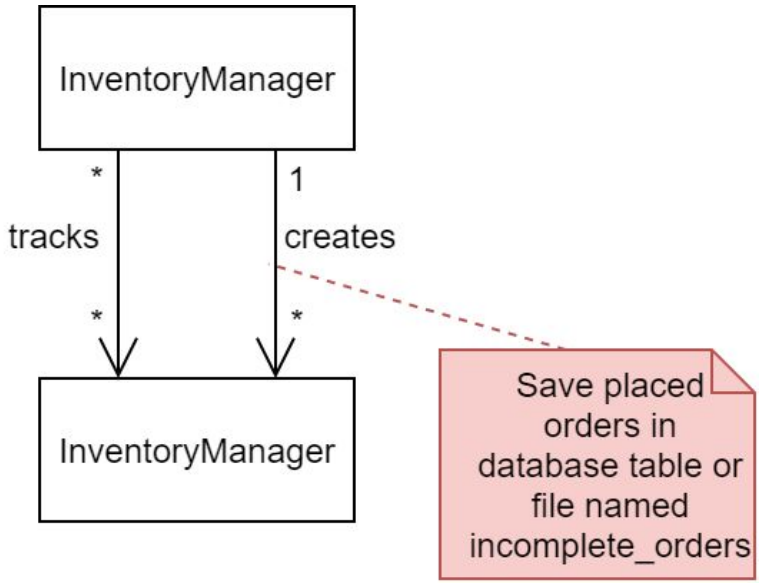


4. **Composition:** Similar to aggregation relationship, in a composition relationship, an object of one class (whole object) will contain objects of other classes (part objects). But in a composition relationship, when the whole object is deleted, part objects will also be deleted. A composition relationship also has an association name and multiplicity associated with it.

Constraints and notes can also be represented in class diagrams. Constraints refer to the conditions that must be preserved while writing code for that class. For example, if you want that userId for every user must be 8 characters long, then such condition you can specify a class diagram as shown below.



Notes are used to write decisions or assumptions made while designing your system. For example, while designing your system you decided that you will save placed orders in a database table called "incomplete_orders". Such decisions can be noted down in the class diagram as shown below.



Association class: Association class is used to capture the properties of the relationship between different classes. For example, when an Admin pays an invoice raised by a supplier, a transaction takes place. This transaction will have several properties which can be captured in the class diagram as shown below.