10 Object Oriented Programming I

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

What is OOPs?

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

Advantages of OOPs:

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

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

How to solve?

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

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

State of the objects is represented by data members and their behavior is represented by methods. Now many objects share common properties and behavior. So, a group of such properties and behavior is made that can generate many instances of itself that have similar characteristics. This group is called a class.

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

For example a vehicle class might look like this:

```
public class Vehicle {

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

  public double getPrice() {
       return price;
  }

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

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

A class in java contains:

- Data members
- Methods
- Constructor

Now let us talk about each of these in detail:

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

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

For e.g.:

```java
static int numStudents;
```

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

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

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

A general student class might look like this:

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

Access Modifiers:

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

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

```java
public class Student {
    private int rollNo;
    public int getRollNo() {
            return this.rollNo;
    }
}
```

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

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

Public: It is accessible everywhere.

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

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

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

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

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

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

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

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

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

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

Now let us add some methods to our student class.

```
public class Student {
final static int noOfStudents = 40;
String name;
private int rollNo;
public String getName() {
        return this.name;
}
public int getRollNo() {
        return this.rollNo;
}
public int getNumStudents() {
        return Student.noOfStudents;
}
}
```

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

```
public static void main(String[] args) {
        Student s = new Student();
        // s.name - will give access to this student's name
}
```

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

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

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

Below is our own custom constructor for our student class.

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

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

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

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

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

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

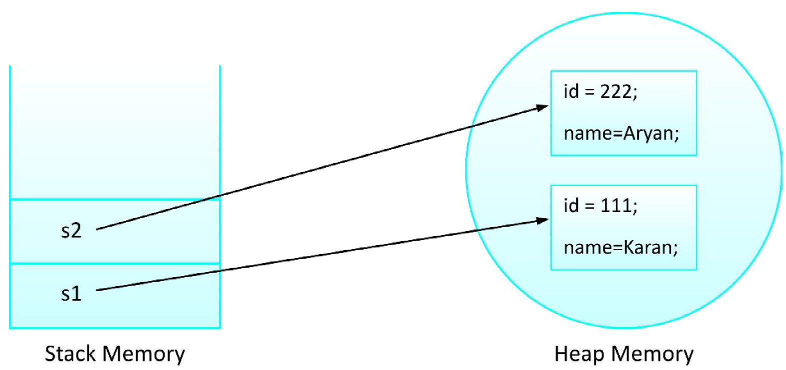
```
    public Student(Student s) {
            this.rollNo = s.rollNo;
            this.name = s.name;
    }
}
```

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

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

1. Memory is allocated on heap and its reference is stored in stack.
2. The non staticdata members are initialized to their default values.
3. The 'this' keyword is set to the current object.
4. The instance initialize is run and the fields are initialized to their respected values.
5. The constructor code is executed.

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



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

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

Also, the outside world needs to know only the essential details through public members. The rest of the implementation details remain hidden from the outside world which is nothing but **Abstraction**.

Like for instance if we wish to know the aggregate marks of a student, we need not know how the aggregate is calculated. We are just interested in the marks and for that we only need to know that we need to call a public function called 'totalMarks' on any student object.

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

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

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

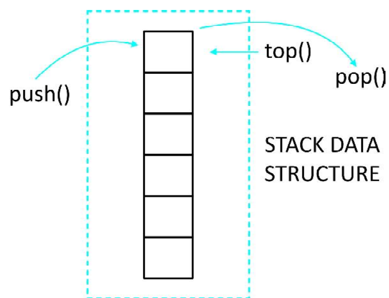
To use a class, we only need to know the public API of the class. We only need to know the signature of the public methods that is their input and output forms to access a class. Thus, only the essential details are shown to the end user and all the complex implementation details are kept hidden. This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

11 Stack and Queue

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

Stack:

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



Implementation:

```
public class Stack {

protected int[] data;
protected int tos;

public Stack(){
      this(5) ;
}

public Stack(int capacity) {
      this.data = new int[capacity];
      this.tos= -1;
}

public void push(int item) throws Exception {
```

```
        if (this.size() == this.data.length) {
              throw new Exception("Stack is Full");
        }
        this.tos++;
        this.data[this.tos] = item;
  }

  public int pop() throws Exception {

        if (this.size() == 0) {
              throw new Exception("Stack is Empty");
        }

        int element = this.data[this.tos];
        this.data[this.tos] = 0;
        this.tos--;
        return element;
  }

  public int peek() throws Exception {

        if (this.size() == 0) {
              throw new Exception("Stack is Empty");
        }

        int element = this.data[this.tos];
        return element;
  }

  public int size() {
        return this.tos + 1;
  }

  public boolean isEmpty(){
        return this.size() == 0;
  }

  public void display() {


        System.out.println("----------");
        for (int i = this.tos; i>= 0; i--) {
              System.out.print(this.data[i] + " ");
        }
        System.out.println(".");
        System.out.println("----------");
  }
}
```

Functions used in Stack:

- push: Insert an item in stack
- pop: Delete the element which is at top of stack, i.e. last inserted item will be removed from stack
- peek: Return the top most element of stack without removing it from stack
- size: Return the no. of elements present in stack
- isEmpty: Return true if entire stack is empty else return false
- display: Display the elements of entire stack

Data Members of Stack Class:

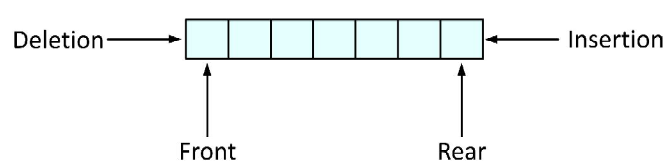
- data: Array which is supposed to store the elements of stack.
- tos: tos stands for *Top Of Stack*, it is a variable used to keep track of index from where the element will be removed.

For pop operation element will be removed from *tos* index. For push operation, *tos* will be incremented and then a new item will be added.

Queue:

A queue is a linear data structure similar to a queue in front of the ticket counter. In a queue we can add an element only to the end of it and also remove an element only from the front of it.Thus, queue is based on the **First-In First-Out (FIFO)** principle that is the element which is inserted first is the first one to be removed and similarly the last one to be inserted is the last one to be removed.

The first element is always the front element in the queue. Queue has two important functions of **enqueue** and **dequeue** which are used for insertion and deletion respectively. Queue allows operation at both the ends. Queue can be implemented with the help of arrays or with the help of linked list.



Implementation:

```
public class Queue {

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

public Queue() {
       this(5);
}

public Queue(int cap) {
       this.data = new int[cap];
       this.front = 0;
       this.size = 0;
}
```

```
    public void enqueue(int item) throws Exception {

        if (this.size() == this.data.length) {
            throw new Exception("Queue is full.");
        }
        int rear = (this.front + this.size) % this.data.length;
        this.data[rear] = item;
        this.size++;
    }

    public int dequeue() throws Exception {

        if (this.size() == 0) {
            throw new Exception("Queue is empty.");
        }

        int element = this.data[this.front];
        this.data[this.front] = 0;

        this.front = (this.front + 1) % this.data.length;
        this.size—;

        return element;
    }

    public int getFront() throws Exception {

        if (this.size() == 0) {
            throw new Exception("Queue is empty.");
        }

        int element = this.data[this.front];
        return element;
    }
    public int size() {
        return this.size;
    }

    public boolean isEmpty() {
        return this.size() == 0;
    }
    public void display() {
        System.out.println("————————————");
        for (int i = 0; i<this.size(); i++) {
            int idx = (this.front + i) % this.data.length;
            System.out.print(this.data[idx] + " ");
        }
        System.out.println();
        System.out.println("————————————");
    }
}
```

Functions used in Queue:

- enqueue: Insert an item in queue
- dequeue: Delete the element from front, i.e. element inserted first will be removed first from the queue
- getFront: Return the element which is at front, without removing it from queue
- size: Return the no. of elements present in queue
- isEmpty: Return true if entire queue is empty else return false
- display: Display the elements of entire queue

Data Members of Stack Class:

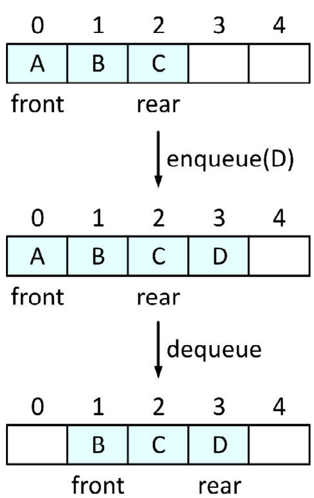
- data: Array which is supposed to store the elements of queue
- front: Variable used to keep track of index from where the element will be removed
- size: Number of elements present in queue

For dequeue operation element will be removed from *front* index. For enqueue operation, a new item will be added at *(front + size)* index.

How the front is updated in the queue?

Enqueue Operation:

Let's take an example,



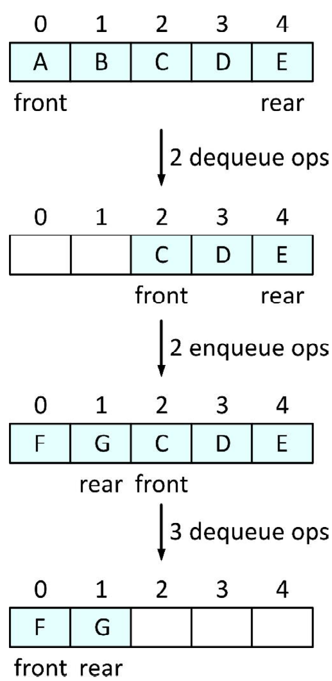
In the given example, we have a queue in which A, B and C is already present.We perform an enqueue with Dso the element is inserted in queue. Then we perform a dequeue so the first element viz. A is removed from the queue. Now if we perform2 enqueue operations with elements Eand F, the element E gets inserted as usual at the back of the queue but F should be added at the first index as it is empty because of the dequeue operation. So to maintain this continuity we firstly calculate the index where the new element should be inserted and then modulo it with the length of the queue.

So, a new element is inserted at (front + size) % data.length. Because of doing modulus, the queue basically becomes a **circular queue.**

Dequeue Operation:

Let's take an example,

Suppose we are having a queue with 5 elements, A, B, C, D and E. As of now *front* is at 0 and *size* is 5. Now we want to dequeue two elements, so elements get removed from *front* and *size* is decreased by 2. If we want to perform two enqueue operation, then a new element is added at 0 and 1 index. *front* is at 2 and *size* is 5. Now, we need to perform 3 dequeue operations, elements are removed from 2, 3 and 4 index and *front* should now point to 0 index. We need to make *front* point to 0, so while dequeuing the elements, for updating front we will use the formula front = (front + 1) %data.length;



12 Object Oriented Programming II

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

Inheritance:

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

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

```
   public class Person {
   String name;
   int age;

   public void displayInfo() {
         System.out.println(this.name + " " + this.age);
   }
}
   public class Student extends Person {
   int marks;

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

Here the Person class is the base class or the super class whereas the student class is the subclass and inherits the properties name, age and methods *displayInfo* from the Person class. The Student class also has its specific property of marks and a method *displayMarks*. Also note the use of the 'extends' keyword which specifies that the subclass extends or inherits the properties of the base class. Thus, we can use the data members and the methods of the Parent class and also have our own data members and methods.

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

```
public class Student extends Person {
int marks;

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

public class Client {

public static void main(String[] args) {

        Student s = new Student();
        s.name = "Rahul";
        s.age = 15;
        s.marks = 80;
        s.displayInfo();
        // OUTPUT: Rahul 15 80
}
}
```

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

Types of Inheritance:

There are various types of inheritance like:

Single inheritance: Here a single subclass inherits from a single base class. Like a Student class inherits from a Person class.

Multilevel inheritance: There is chain of classes inheriting from one another. Like a Person class inherits from a Mammal class which in turn inherits from the Animal class.

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

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

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

Polymorphism:

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

Let us learn about polymorphism and its types in more detail.Polymorphism in java is a concept by which we can perform a single action by different ways. There are two types of polymorphism in java:

1. Compile time polymorphism
2. Run time polymorphism

Compile Time Polymorphism:

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

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

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

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

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

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

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

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

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

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

Runtime Polymorphism:

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

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

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

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

Let us consider a context where we have a parent class **P** and a child class derived from P called **C**.

Now while making an object there can be four combinations depending on the type of the instance and its reference. Let us understand each in detail:

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

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

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

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

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

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

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

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

Abstract Methods & Classes : The literal meaning of the word abstract is that something exists in idea but not physically defined. Similar are the abstract methods and classes in java.A method that doesn't have any implementation and is marked as abstract is an abstract method.Similarly, a class marked abstract, which needs to be extended and all its abstract methods to be implemented, is known as an abstract class.

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

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

class Student extends Person {

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

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

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

Modifiers & Data Members:

Access Modifiers:

Public: they are accessible everywhere.

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

Private: they are available only within the class.

Default: they are available only within the package.

Non-Access Modifiers:

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

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

Modifiers & Functions:

Access Modifiers:

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

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

Private: the method is available only in the class.

Default: the method is available only within the package.

Non-Access Modifiers:

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

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

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

Modifiers & Classes:

Access Modifiers:

Public: this class can be accessed anywhere.

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

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

Non-Access Modifiers:

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

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

13 Linked List

What are the disadvantages of using Arrays?

1. **The size of the arrays is fixed:** We must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.
2. **Inserting a new element in an array of elements is expensive**, because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, …..].

And if we want to insert a new ID 1005 such that the array is in sorted order, we have to shift all the elements after 1000 to their right.

Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], every element post 1010 has to be moved to its left.

So, in this chapter we consider another data structure called Linked list which provides the following two advantages over arrays

1. Dynamic size
2. Ease of insertion/deletion

Definition :

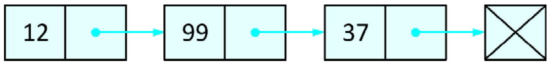
Linked List are linear data structures where every element is a separate object with a data part and address part. Each object is called a Node. The nodes are not stored in contiguous locations. They are linked using addresses.In a linkedlist we can create as many nodes as we want according to our requirement.This can be done at the runtime. The memory allocation done by the JVM at runtime is known as DYNAMIC MEMORY ALLOCATION**.** Thus, linked list uses dynamic memory allocation to allocate memory to the nodes at the runtime.

Java has an in-built class for Linked List. To use the in-built LinkedList class, we need to import java.util library. Here’s how you can use this class:

```
import java.util.LinkedList;
LinkedList<Integer> LL = new LinkedList<>();
```

We can also create our own implementation of linked list.

Implementation



Each element enclosed in a rectangle is called a Node. Nodes are comprised of two parts, a data part and an address part. Each node will be stored at some address in memory. The address part of the node stores the address of the next node in sequence and the last node stores **null** in its address part as there is no next node. This is how the nodes are linked with each other to form a linked list.

Thus, the first node with data 12 stores the address of the next node with data 99. This node subsequently stores the address of its next node which has data 37. Since this node is the last node of the linked list, its address part stores null.

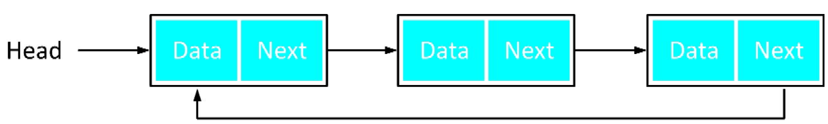
Different Types of Linked List

1. **Singly Linked List :** These are unidirectional linked lists in which every node points to the next node and the head node points to the starting node. The example that we saw above was a singly linked list.

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

3. **Circular Linked List :** A circular linked list can be either singly linked or doubly linked. It is circular in the sense that from the last node, you can directly move to the first node.

   - In singly linked list, next pointer of last node points to the first node.



   - In doubly linked list, next pointer of last node points to the first node and previous pointer of first node points to last node as well.

General Implementation

The linked list class contains a class Node,a Node head and a Node tail as the property of its class. Head will point to the first node of the linked list and tail will point to the last node of the linked list. Every Node contains a data element and a Node next reference which will point to the next node in the Linked List. Initially, both the head and tail will store null (as the linked list is currently empty).

```
private class Node {
        int data;
        Node next;
}
```

1. **Add First:** AddFirst means to add the element in the list at the starting of the linked list. In this case, the head of the linked list is updated.

```
public voidaddFirst(int item) {

            // create
            Node nn = new Node();

            // values update
            nn.data = item;
            nn.next = null;

            // attach
            nn.next = this.head;

            // summary update
            if (size == 0) {
                  this.head = nn;
                  this.tail = nn;
                  this.size++;
            } else {
                  this.head = nn;
                  this.size++;
            }

      }
```

2. **Add Last :** AddLast means to add the element at the end of the linked list. In this case, the tail pointer is updated.

```
public void addLast(int item) {

            // create a new node
            Node nn = new Node();

            // update
```

```
            nn.data = item;
            nn.next = null;

            // attach
            if (this.size> 0)
                  this.tail.next = nn;
            // summary update
            if (this.size == 0) {
                  this.head = nn;
                  this.tail = nn;
                  this.size++;
            } else {
                  this.tail = nn;
                  this.size++;
            }

      }
```

3. **Remove First :** RemoveFirst means to remove the element from the starting of the linked list. In this case, the head will be updated.

```
public int removeFirst() throws Exception {

            if (this.size == 0) {
                  throw new Exception("LL is empty.");
            }
            int rv = this.head.data;

            if (this.size == 1) {
                  this.head = null;
                  this.tail = null;
            } else {
                  this.head = this.head.next;
            }

            this.size–;
            return rv;
      }
```

4. **RemoveLast :** RemoveLast means to remove the element from the end of the linked list. In this case, tail of linked list is updated.

```
public int removeLast() throws Exception {

            if (this.size == 0) {
                  throw new Exception("LL is empty.");
            }

            int rv = this.tail.data;
```

```
        if (this.size == 1) {
            this.head = null;
            this.tail = null;
        } else {
            Node nsm2 = this.getNodeAt(this.size - 2);
            nsm2.next = null;
            this.tail = nsm2;
        }

        this.size–;
        return rv;

    }

    private Node getNodeAt(int idx) throws Exception {

        if (this.size == 0) {
            throw new Exception("LL is empty.");
        }

    if (idx< 0 || idx>= this.size) {
            throw new Exception("Invalid index.");
        }
        Node temp = this.head;
        for (int i = 0; i<idx; i++) {
            temp = temp.next;
        }
        return temp;
    }
```

Methods provided by the inbuilt Linked List class:

1. **boolean add(Object item) :** It adds the item at the end of the list.
2. **void add(int index, Object item):** It adds an item at the given index of the the list.
3. **void addFirst(Object item):** It adds the item (or element) at the first position in the list.
4. **void addLast(Object item):** It inserts the specified item at the end of the list.
5. **void clear():** It removes all the elements of a list.
6. **boolean contains(Object item):** It checks whether the given item is present in the list or not. If the item is present then it returns true else false.
7. **Object get(int index):** It returns the item of the specified index from the list.
8. **Object getFirst():** It returns the item of the specified index from the list.
9. **Object getLast():** It fetches the last item from the list.
10. **int indexOf(Object item):** It returns the index of the specified item.
11. **int lastIndexOf(Object item):** It returns the index of last occurrence of the specified element.
12. **Object remove():** It removes the first element of the list.

13. **Object remove(int index):** It removes the item from the list which is present at the specified index.
14. **Object remove(Object obj):** It removes the specified object from the list.
15. **Object removeFirst():** It removes the first item from the list.
16. **Object removeLast():** It removes the last item of the list.
17. **Object set(int index, Object item):** It updates the item of specified index with the give value.
18. **int size():** It returns the number of elements of the list.

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

        // Using in-built linked list
        LinkedList<Integer> LL = new LinkedList<>();

        // Adding 10 at the end of linked list
        LL.add(10);         // [10]
        LL.add(20);         // [10, 20]
        LL.add(30);         // [10, 20, 30]
        LL.add(40);         // [10, 20, 30, 40]
        LL.add(50);         // [10, 20, 30, 40, 50]

        // Adding 70 at 2nd index
        LL.add(2, 70);              // [10, 20, 70, 30, 40, 50]
        LL.add(30);         // [10, 20, 70, 30, 40, 50, 30]

        // Adding 90 at first position
        LL.addFirst(90);    // [90, 10, 20, 70, 30, 40, 50, 30]

        // Adding 66 at last position
        LL.addLast(66);             // [90, 10, 20, 70, 30, 40, 50, 30, 66]
        LL.add(30);                 // [90, 10, 20, 70, 30, 40, 50, 30, 66, 30]

        // Returns true if list contains 90 otherwise false
        System.out.println(LL.contains(90));          // true

        // Returns element at 4th index
        System.out.println(LL.get(4));                // 30

        // Returns element at first index
        System.out.println(LL.getFirst());            // 90
        // Returns element at last index
        System.out.println(LL.getLast());             // 30

        // Returns index of first occurrence of 40
        System.out.println(LL.indexOf(40));           // 5

        // Return index of last occurrence of 30
        System.out.println(LL.lastIndexOf(30)); // 9

        // Returns and remove the First element
        System.out.println(LL.remove());              // 90
```

```
        // Return and remove the element at 2nd index
        System.out.println(LL.remove(2));           // 70

        // Returns and removes the first element
        System.out.println(LL.removeFirst());       // 10

        // Returns and removes the last element
        System.out.println(LL.removeLast());        // 30

        // Update element to 55 at 2nd index
        System.out.println(LL.set(2, 55));          // [20, 30, 55, 50, 30, 66]

        // Print the size of the linked list
        System.out.println(LL.size());              // 6

        // Clears the whole Linked List
        LL.clear();

        // Printing Linked List
        System.out.println(LL);                           // []

}
```

Disadvantages of Linked List :

1. **Memory Usage :** More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.
2. **Traversal :** We cannot randomly access any element as we do in array by index. For example, if we want to access a node at position n then we must traverse all the nodes before it. So, time required to access a node is large.
3. **Reverse Traversing :** In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

ArrayList v/s Linked List

ArrayList	LinkedList
ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

Example :

1. Write a program to reverse a linked list.

   Input: 1->2->3->4->5->null **Output:** 5->4->3->2->1->null

```
public void reverse() {

        Node prev = this.head;
        Node curr = prev.next;

        while (curr != null) {

                Node temp = curr.next;
                curr.next = prev;

                prev = curr;
                curr = temp;

        }

        Node t = this.head;
        this.head = this.tail;
        this.tail = t;

        this.tail.next = null;
    }
```

2. Find the middle node of the linked list.

   Input: 1->2->3->4->5->null **Output:** 3

   Input: 1->2->3->4->5->6->null **Output:** 3

```
public int mid() {

        Node slow = this.head;
        Node fast = this.head;

        while (fast.next != null && fast.next.next != null) {
                slow = slow.next;
                fast = fast.next.next;
        }

        return slow.data;
    }
```

14 Objective Oriented Programming III

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

Interfaces:

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

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

Let us write our person interface to understand clearly.

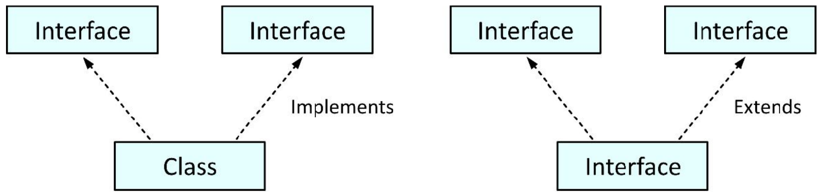
```
interface Person {

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

Here the person interface has three abstract methods that don't have their body. Let us now learn how to use an interface. As a class can extend another class, similarly interfaces can also be 'implemented' by the classes. This is done by using the keyword ***implements***followed by the name of the interface. The class which implements an interface has to implement all the methods of that interface. Thus, an interface can be thought of as a contract signed by the class to implement all the functions of that interface, if the class that implements an interface doesn't implement all the abstract methods then that class must be marked abstract.

While overriding the methods of the interface, the signature and the return type of the method cannot be changed. Another point to note here is that while a class can extend only one class at a time, it can implement more than one interface at a time. Now let us make a Student class to implement our Person interface that we just made.

```
class Student1 implements Person {

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

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

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

      public stati cvoid main(String[] args) {
            Student1 s = new Student1();
            s.name(); // The name method
            s.age(); // The age method
            s.gender(); // The gender method
      }
}

abstract class Student2 implements Person {

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

      public static void main(String[] args) {
            Student2 s = new Student2(); // error
            // abstract class can't be instantiated
      }
}
```

Note here that the Student1 class that implements the person interface implements all the methods of the interface whereas the Student2 class doesn't implement all the methods of the interface and hence is marked abstract.



Multiple Inheritance in Java

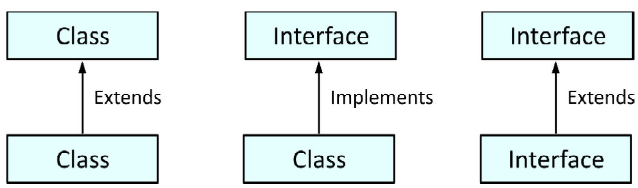
We know that multiple inheritance cannot be achieved by classes in java but interfaces can implement multiple inheritance. We just saw that a class can implement multiple interfaces. This can be thought of as multiple inheritance as the class is implementing methods of multiple interfaces. Also as a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

```
interface male {
	public void gender();
}
interface female {
	public void gender();
}
	// Interface extending 2 interfaces
interface person extends male, female {
	public void name();
}
class student implements person {
	public void name() {
		System.out.println("the name method");
	}
	public void gender() {
		System.out.println("the gender method");
	}
	public static void main(String[] args) {
		student s = new student();
		s.name(); // The name method
		s.gender(); // The gender method
	}
}
```

```
package Demo;

public class person {
       public void display() {
              System.out.println("hello");
       }

}

// For accessing outside package using package name
import Demo.*;

// Using package name and class name
import Demo.person;

// Using full name
public static void main(String args[]){
       Demo.person obj = new Demo.person();
}
```

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

All the relations between interfaces and class are summarized below.



Packages:

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

1. Built-in Packages   2. User-defined Packages

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

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

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

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

- Using the ***packagename.****By using this, all the classes and the interfaces can be accessed outside of the package. However, the sub-packages cannot be accessed.The **import** keyword is used for the same.
- Using***packagename.classname***By using this, only the specified class will be made available.
- Using ***full name***We write the packagename followed by a dot and the class/interface name wherever we want to access.

```
package Demo;

public class person {
       public void display() {
              System.out.println("hello");
       }

}

// For accessing outside package using package name
import Demo.*;

// Using package name and class name
import Demo.person;

// Using full name
public static void main(String args[]){
       Demo.person obj = new Demo.person();
}
```

15 Objective Oriented Programming IV

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

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

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

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

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

// Pair of two Integers
Pair<Integer> pInts = new Pair<Integer>();

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

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

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

1. **Type-Safety:** We can hold only a single type of objects in generics. It doesn't allow storing other objects.
2. **Type casting is not required:** There is no need to typecast the object.
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

The good programming strategy says it is far better to handle the problem at compile time than at runtime.

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

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

Its instance can be created as follows:

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

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

Multilayer Generic Parameters: The generic parameters can be multilayered.

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

Here pLayered.first and pLayered.second are themselves pair of Integers. Similarly, we can add multiple layers to the parameters.

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

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

We can do this as shown below:

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

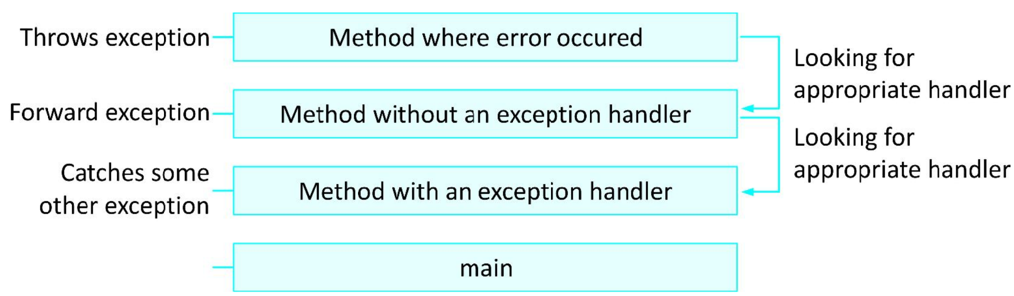
When we write <T extends Comparable<T>> this means that only those parameters are allowed those who have implemented Comparable Interface.

Exception Handling:

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

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

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



Types of Exception:

Checked Exceptions: These are exceptional conditions that we can anticipate when user makes mis-take.

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

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

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

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

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

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

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

Here is a sample code to explain the same.

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

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

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

	try {
		int data = dividend / divisor;
		System.out.println(data);
	} catch (ArithmeticExceptione) {
		System.out.println("Divide by zero error");
	} finally {
		System.out.println("Finally block is always executed");
	}

	System.out.println("Rest of the code...");
```

Note :

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

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

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

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

Here is the code for the factorial method:

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

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

```
public static void main(String[] args) {
        Scanner scn = new Scanner(System.in);
        System.out.println("Enter number");
        int n = scn.nextInt();
        int a = 10;
        try {
            System.out.println(fact(n));
            a++;
        } catch (InvalidInputException e) {
            System.out.println("Invalid input. Try again.");
            return;
        }
    }
```

16 Trees

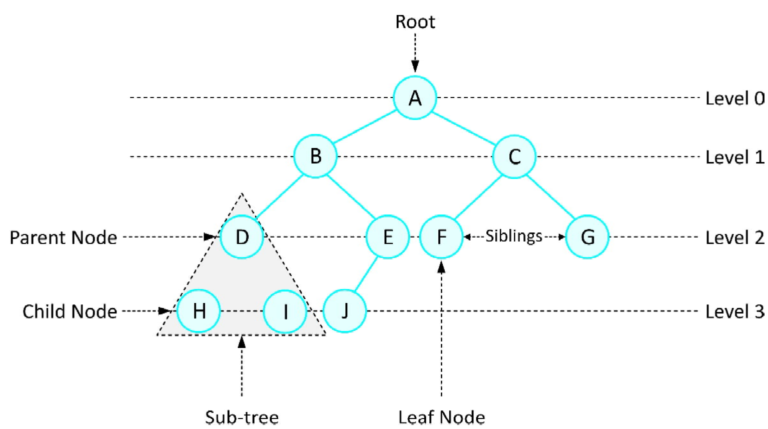
It is a non-linear data structure in which each node may be associated with more than one node.Eachnode in the tree points to its children. A tree that is not empty consists of a root node and potentiallymany levels of additional nodes that form a hierarchy.

- The root of a tree is the node with no parent. There can be only one root node.
- A node with no children is known as a leaf.
- A node p is ancestor of node q if the node p lies on the path from q to the root of the tree.
- If each node of a tree has only one child then it is called skew.

The most basic examples of a Tree are:

- File system inside the computer
- HTML DOM, where there is a hierarchy of HTML tags
- Organisational structure of employees inside an organization.

The Tree data structure looks reverse of the actual physical treeand has root on top.



Basic Terminology in Trees:

- **Node :** It is the structure that contains the data about each element in the tree.
- **Root :** The top node in a tree.
- **Children :** A node directly connected to another node when moving away from the Root.
- **Parent :** The immediate node to which a node is connected on the path to root.
- **Siblings :** A group of nodes with same parent.
- **Ancestor:** A node reachable by repeated proceeding from child to parent.

- **Descendant:** A node reachable by repeated proceeding from parent to child.
- **Leaves :** The nodes that don't have any children.
- **Path :** The sequence of nodes along the edges of the tree.
- **Height of Tree :** The number of edges on the longest downward path from root to a leaf.
- **Depth :** The number of edges from the node to the root node.
- **Degree :** The number of edges that are being connected to the node, can be further subdividedin to in-degree and out-degree.

Binary Tree :

A tree is called Binary tree if each node has two or fewer children.

Each node of a binary tree can be represented as a class.This class contains a data and two nodes to keep track of left and right children and the tree class contains a property as root node.

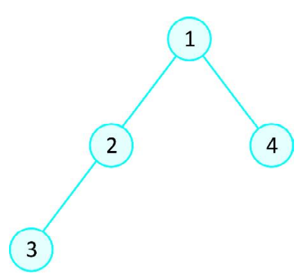
```
private class Node {
        int data;
        Node left;
        Node right;
    }
```

Construction of Tree:

If a node has no child then user will give false for left and right child.
Input : 1 true 2 true 3 true false falsefalse true 4 false false

1!left of 1 (2)!left of 2 (3)!left of 3 is false !backtrackto 3 !rightof 3 is false !backtrack to 3 ! backtrack to 2 !right of 2 is false!backtrack to 1 !right of 1(4) !leftof 4 is false !backtrack to 4 !rightof 4 is false !backtrack to 4 ! backtrack to 1 ! finish

The tree would look like,



Code for the above tree construction can be written like:

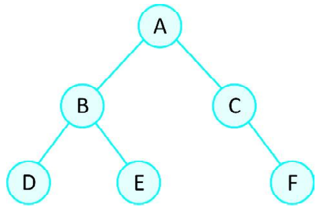
```
public class BinaryTree {

    Scanner scn = new Scanner(System.in);

    private class Node {
        int data;
```

```
            Node left;
            Node right;
        }
        private Node root;
        public BinaryTree() {
            this.root = takeInput(null, false);
        }
        private Node takeInput(Node parent, booleanilc) {
            if (parent == null) {
                System.out.println("Root data ?");
            } else {
                if (ilc) {
            System.out.println("Enter the data for left child of " + parent.data);
                } else {
            System.out.println("Enter the data for right child of " + parent.data);
                }
            }
            int item = scn.nextInt ();
            Node nn = new Node();
            nn.data = item;
            System.out.println("Left child for " + nn.data + " ?");
            boolean hlc = scn.nextBoolean();
            if (hlc) {
                nn.left = takeInput(nn, true);
            }
            System.out.println("Right child for " + nn.data + " ?");
            boolean hrc = scn.nextBoolean();
            if (hrc) {
                nn.right = takeInput(nn, false);
            }
            return nn;
        }
    }
```

Applications :

- File hierarchy structure in the computer is tree based. Assuming folder as a node can have sub-directories and files as a leaf node.
- For Android Developers out there, findViewById searches the view using search in tree type layout structure.

- For Web Developers out there, Document Object Model is tree type of structure and all traversal in it are tree search.



Pre-order Traversal:

Each node is processed before(pre) any nodes in its subtrees: **root left right**

Output :A (B D E) (C F)

```
private void preorder(Node node) {
            if (node == null) {
                  return;
            }
            // Node
            System.out.println(node.data);
            // Left
            preorder(node.left);

            // Right
            preorder(node.right);
      }
```

Post-order Traversal:

Each node is processed after(post) all nodes in its subtrees: **left right root**

Output : (D E B) (F C) A

```
private void postorder(Node node) {
      if (node == null) {
            return;
      }
      // Left
      postorder(node.left);
      // Right
      postorder(node.right);
      // Node
      System.out.println(node.data);
}
```

Inorder Traversal:

Each node is processed after all nodesofleft subtreeand before all nodes of right subtree: left root right

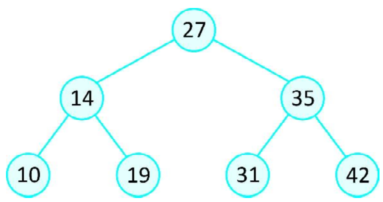
Output : (D B E) A (C F)

```
private void inorder(Node node) {
        if (node == null) {
                return;
        }
        // Left
        inorder(node.left);
        // Node
        System.out.println(node.data);
        // Right
        inorder(node.right);
}
public void levelorder() {

        // Queue DS is used for achieving level order
        Queue<Node>queue = new LinkedList<>();
        queue.add(this.root);

        while(!queue.isEmpty()) {

                // remove a node from queue
                Node rn = queue.remove();

                // print the data of removed node
                System.out.println(rn.data);

                // if left child is not null then insert in queue
                if (rn.left != null) {
                        queue.add(rn.left);
                }

                // if right child is not null then insert in queue
                if (rn.right != null) {
                        queue.add(rn.right);
                }

        }
}
```

17 Binary Search Trees

It's a binary tree with an additional property that should be satisfied by each node, which is:

- Each node in the left subtree is less than the value of the root node.
- Each node in the right subtree is greater than the root node's value.



Inorder traversal (left root right) of above tree: 10, 14, 19, 27, 31, 42, 35

The inorder traversal of any Binary Search Tree will give elements in sorted order.

Creating a BST

```
private Node add(Node node, int item) {

        if (node == null) {
            Node nn = new Node();
            nn.data = item;
            return nn;
        }

        if (item<= node.data) {
            node.left = add(node.left, item);
        } else {
            node.right = add(node.right, item);
        }

        return node;

    }
```

Searching in a BST

For searching, we compare the node's value with the item to be searched.

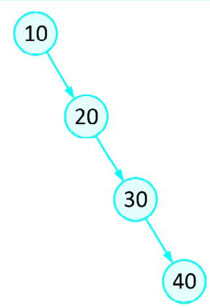
- If they are equal we return the true.
- If the item to be found is less than the value of the root node, we search only in the left subtree and discard the right subtree.
- If the item to be found is greater than the value of the root node, we search only in the right subtree and discard the left subtree.

Thus, in each turn we are neglecting some of the elements without even traversing them. While searching, at every level we will be visiting exactly one node. So, the complexity of find is actually equal to height of tree.

```
private boolean find(Node node, int item) {

		if (node == null) {
			return false;
		}

		if (item<node.data) {
			return find(node.left, item);
		} else if (item>node.data) {
			return find(node.right, item);
		} else {
			return true;
		}

	}
```

Time complexity for Searching in a BST

While searching for an item, at every level exactly one node is visited. So, the complexity of searching in BST is O(h).



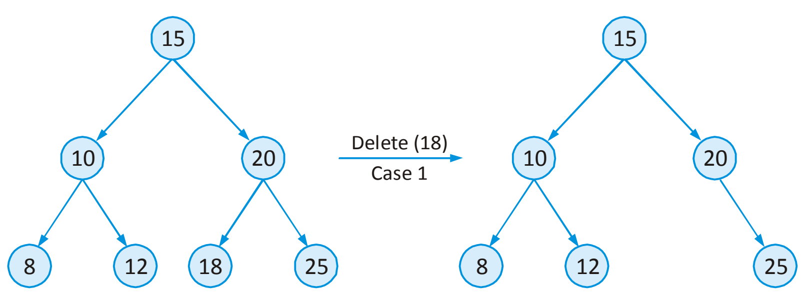
In the above figure, the tree shown is Right Skewed. It might happen that the BST is either left skewed or right skewed, then height of tree = no. of nodes in tree. In such case, time complexity of searching will be O(n).

The Time Complexity for searching an element in a Binary Search Tree is O(height of BST).

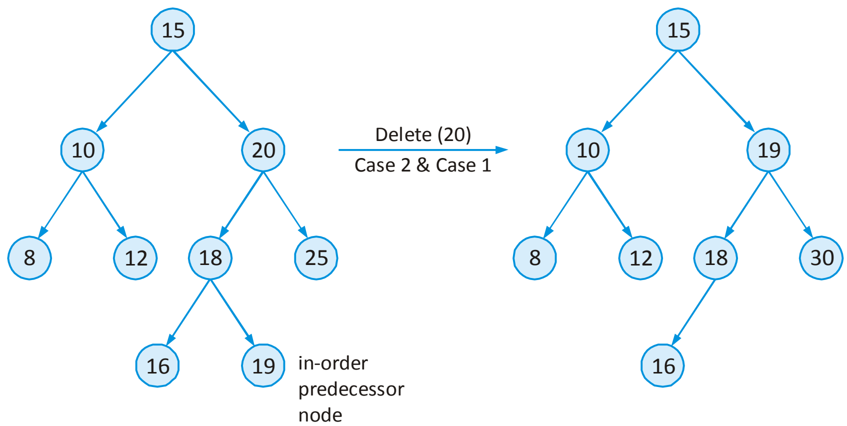
Deletion in a BST

Given a BST, write an efficient function to delete a given item in it.To delete a node from BST, there are three possible cases to consider:

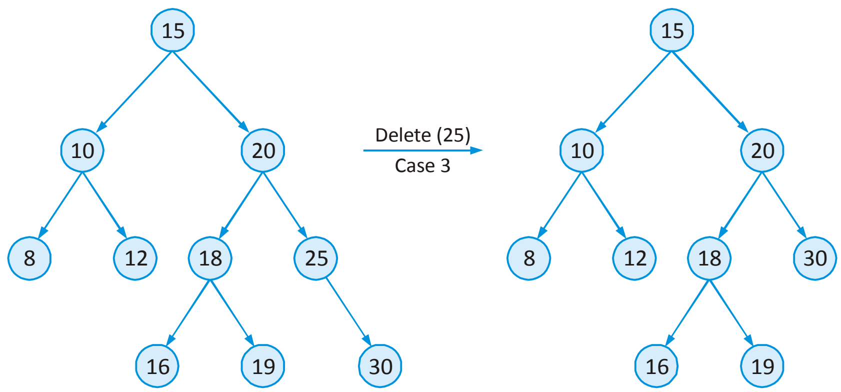
Case 1: Deleting a node with no children: simply remove the node from the tree.



Case 2: Deleting a node with two children: call the node to be deleted N. Here, we are not deleting the node N, instead replacing the content of the node N. Best node to replace the data of N will be the maximum of left subtree or minimum of right subtree. As of now, we will be using maximum of left subtree, let's call the maximum node of left subtree as R.Store the data of this R node in *temp* variable, recursively call deleteon left subtree with *temp* then replace the content of N node with the value stored in *temp*.



Case 3: Deleting a node with one child: remove the node and replace it with its child.




```
private int max(Node node) {

        if (node.right == null) {
                return node.data;
        }

        return max(node.right);

    }

    private void remove(Node node, Node parent, int item) {

        if (item < node.data) {
                remove(node.left, node, item);
    } else if (item > node.data) {
        remove(node.right, node, item);
```

```
        } else {

            if (node.left == null && node.right == null) {
                if (node.data<= parent.data) {
                    parent.left = null;
                } else {
                    parent.right = null;
                }
            }

            else if (node.left != null && node.right == null) {

                if (node.data<= parent.data) {
                    parent.left = node.left;
                } else {
                    parent.right = node.left;
                }
            }

            else if (node.right != null && node.left == null) {
                if (node.data<= parent.data) {
                    parent.left = node.right;
                } else {
                    parent.right = node.right;
                }
            }

            else {

                int temp = max(node.left);
                remove(node.left, node, temp);
                node.data = temp;
            }
        }
    }
```

18 Heap

Consider an array in which we repeatedly have to find maximum or minimum element. Naive method would be searching for maximum/minimum in each query, that would be take O(qn) time if q queries were there. Another way could to be keep a sorted array and give min/ max element in constant time but in that case, insertion and deletion would cost us linear time.

To solve this problem we needed a data structure called Heap.In heap, time complexity of operationsare:

- Insertion : log(n)
- Deletion : log(n)
- Minimum Element (for Min-Heap) : O(1)
- Maximum Element (for Max-Heap) : O(1)

Binary Heap is a treebased data structure, each node having at most 2 children. The constraint on abinary heap are:

- **The priority of parent node is greater than both children.**

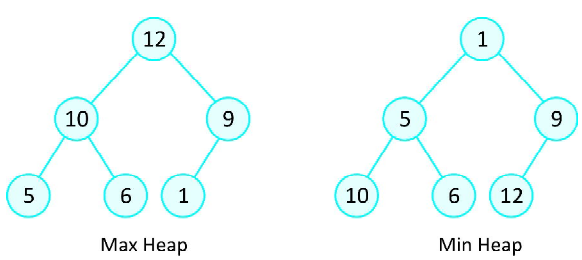
  Now, priority can have two meaning: one is marks and other is rank. Higher marks means higher priority and lower rank means higher priority. If we are talking about something like rank then parent node will have less value as compared to its children and such a heap will be called as **min heap**. If we talk about marks then value of parent node will be larger as compared to children node value, such a heap is called as **max heap.**

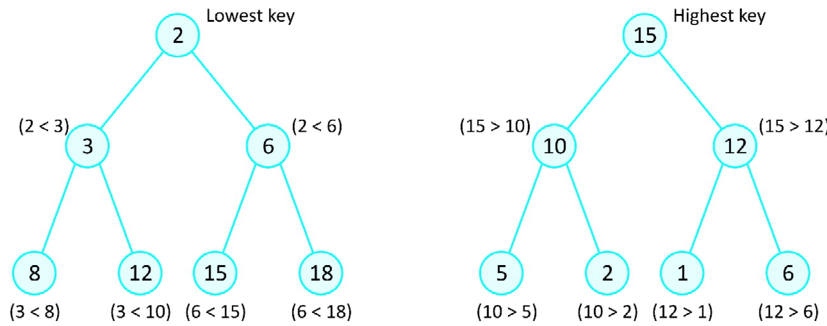
- It is a complete binary tree that is new level will begin only when all the previous levels arecompletely full.

Two types of heap are possible:

Max-heap Property: the value of parent node is greater than or equal to the value of its children, with the maximum-value element at the root and it must be a complete binary tree that is new level will begin only when all the previous levels are completely full.

Min-heap Property: the value of parent node is smaller than or equal to the value of its children, with the minimum-value element at the root and it must be a complete binary tree that is new level will begin only when all the previous levels are completely full.



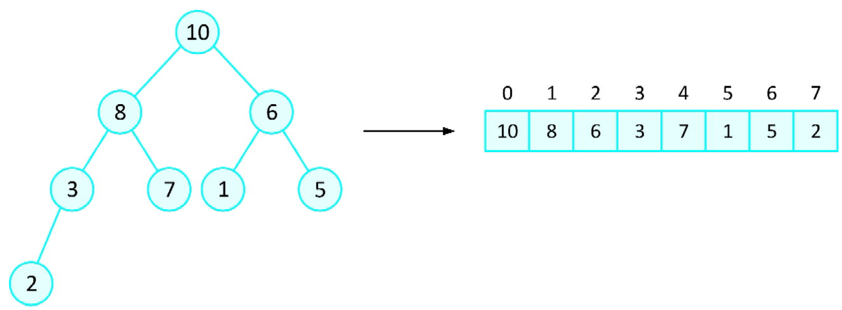


Min Heap
(Parent key is less than or equal
to (≤) the child key)

Max Heap
(Parent key is less than or equal
to (≥) the child key)

Reason behind Using Complete Binary Tree

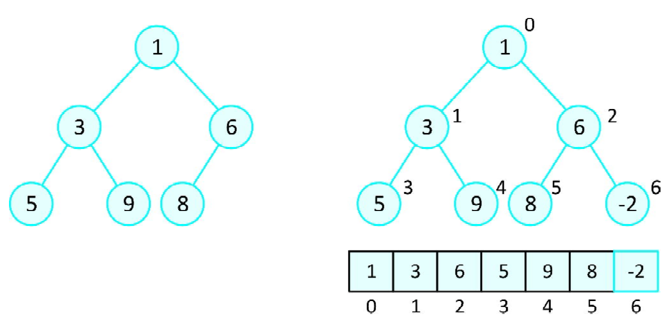
Being a complete binary tree, heap can be stored in the form of the array, making access to each value easier. Also, as we can see in image below, children of each node are available at location 2i+1 or 2i +2, so complete binary tree can be stored in random access linear DS in effective manner.

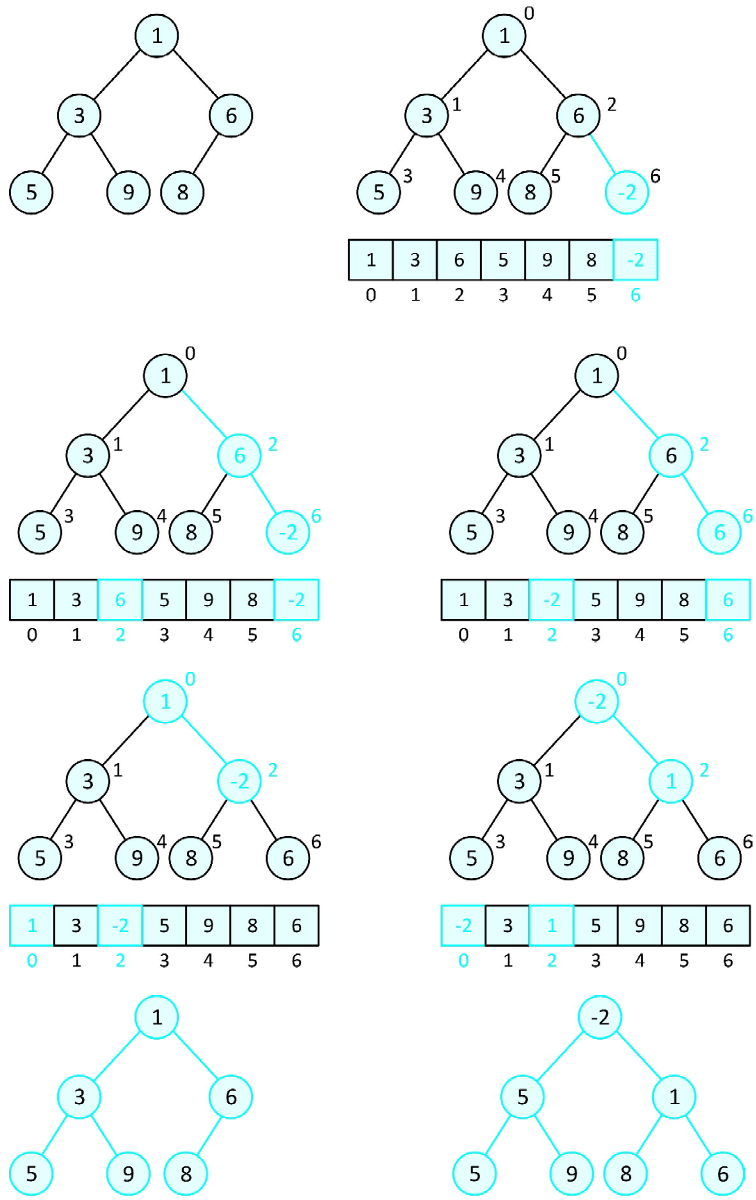


Insertion in Heap

Now, let us phrase general algorithm to insert a new element into a heap.

1. Add a new element to the end of an array.
2. Shift up the new element, while heap property is broken. Shifting is done as following: compare node's value with parent's value. If they are in wrong order, swap them, viz known as Upheapify.





Source Heap After -2 insertion

Complexity of the insertion operation is O(h), where h is heap's height. Taking into account completeness of the tree, O(h) = O(log n), where n is number of elements in a heap.

```
public class Heap {

    private ArrayList<Integer> data = new ArrayList<>();

    public void add(int item) {

        data.add(item);
        upheapify(data.size() - 1);
    }
```

```
	private void upheapify(int ci) {

		int pi = (ci - 1) / 2;

		if (data.get(ci) <data.get(pi)) {
			swap(ci, pi);
			upheapify(pi);
		}
	}
	private void swap(int i, int j) {

		int ith = data.get(i);
		int jth = data.get(j);

		data.set(i, jth);
		data.set(j, ith);

	}
	public int get() {
		return this.data.get(0);
	}

	public int remove() {

		swap(0, data.size() - 1);
		int rv = data.remove(data.size() - 1);
		downheapify(0);
		return rv;
	}
	private void downheapify(int pi) {

		int lci = (2 * pi) + 1;
		int rci = (2 * pi) + 2;
		int mini = pi;
		if (lci<data.size() && data.get(lci) <data.get(mini)) {
			mini = lci;
		}

		if (rci<data.size() && data.get(rci) <data.get(mini)) {
			mini = rci;
		}
```

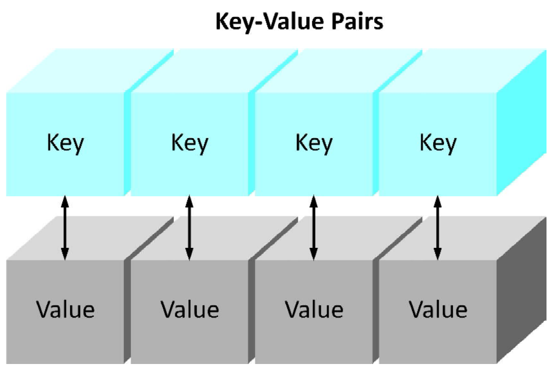
```
                if (mini != pi) {
                        swap(mini, pi);
                        downheapify(mini);
                }
        }

        public void display() {
                System.out.println(this.data);
        }

        public int size() {
                return this.data.size();
        }

        public boolean isEmpty() {
                return this.size() == 0;
        }
}
```

19 Hashmap

Suppose, we want to store the data of students and each student is represented by an object, we can use data structures like array, array-list, linked-list etc. but if we want to get a specific student object then we need to traverse over the entire list, hence it will give complexity of O(n). With HashMap, we can give a unique identity to each such object and access, update and remove any such object in constant time.

Introduction

Hashmap is a collection of key-value pairs i.e. all the keys are mapped to a particular value. All the keys in the hashmap are unique. If we try to insert the same key again, the previous value is updated.



Methods available in Java HashMap:

public V put(K key, V value): Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

public V get(Object key): Returns the value mapped to the specified key, or null if there is no mapping for the key.

public boolean containsKey(Object key): This method returns ‘true’ if the key exists otherwise it will return ‘false’.

public boolean containsValue(Object value): This HashMap method returns true if the value exists otherwise false.

public V remove(Object key): Removes the mapping for the specified key from this map if present.

public boolean isEmpty(): A utility method returning true if no key-value mappings are present.

public Set<K>keySet(): Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

public int size(): Returns the number of key-value mappings in this map.

public void clear(): This HashMap method will remove all the mappings and HashMap will become empty.

Complexity of put, get,containsKey, remove, isEmpty, size function is O(1) and of containsValue, keySet and clear function is O(n)

```
HashMap<String, Integer> map = new HashMap<>();

// put ->O(1)
map.put("India", 10);
map.put("US", 20);
map.put("UK", 30);
System.out.println(map); // {UK=30, US=20, India=10}

// value of US updated to 50
map.put("US", 50);
System.out.println(map); // {UK=30, US=50, India=10}

// get ->O(1)
System.out.println(map.get("India")); // 10
System.out.println(map.get("Aus")); // null

// containsKey ->O(1)
System.out.println(map.containsKey("Aus")); // false
System.out.println(map.containsKey("India")); // true

// containsValue -> O(n)
System.out.println(map.containsValue(10)); // true
System.out.println(map.containsValue(100)); // false

// remove ->O(1)
System.out.println(map.remove("India")); // 10
System.out.println(map.remove("SL")); // null

// isEmpty ->O(1)
System.out.println(map.isEmpty()); // false

// keySet -> O(n)
Set<String>set = map.keySet();

for (String key :set) {
       System.out.println(key + "->" + map.get(key));
}

// size ->O(1)
System.out.println(map.size()); // 2
// clear -> O(n)
map.clear();

System.out.println(map); // {}
```

20 Dynamic Programming

Dynamic Programming:

It is a ***technique*** of solving a complex problem by breaking it down into a collection of simpler sub-problems, solving each of those sub-problems just once, and storing their solutions.

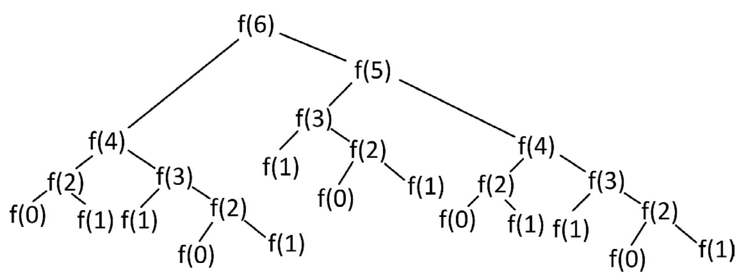
Dynamic Programming avoids repeated computations and thus is an optimization over standard solutions.

Dynamic Programming can be applied to any problem if that satisfies these two criteria.

1. Overlapping Sub-problem:

One or more task is computed multiple times

Consider recursion tree of calls for fibonacci(6)



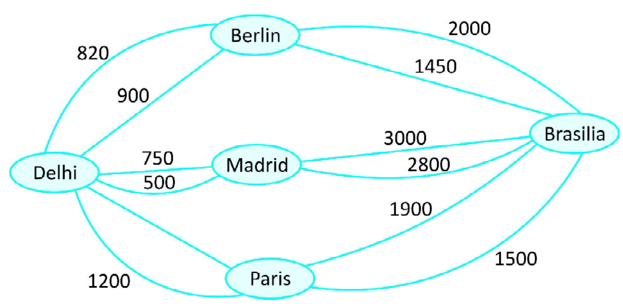
f(4) ! 2 times

f(3) ! 1 time independently, and 2 times inside f(4)

2. Optimal Substructure:

The optimal solution of the smaller problem helps building the optimal solution to the larger problem.

That means, when building your solution for a problem of size n, you split the problem to smaller problems, one of them of size n'. Now, you need only to consider the optimal solution to n', and not all possible solutions to it, based on the optimal substructure property.



There exists multiple paths from one city to another. But to choose the optimal (smallest) route from Delhi to Brasilia, you consider only the optimal (smallest) paths from Delhi to intermediate cities and from intermediate cities to Brasilia.

Presence of alternate routes is not taken into consideration to find the optimal path.

To avoid re-computation of overlapping sub-problems, either of two approaches can be used.

1. **Memorization (Top-Down):** Calculate the solution for a problem and all its sub-problems as and when it is required. The next time they are required just use the value. Here all the sub-problems are not computed, only those that are required are computed.

   This is the top-down approach which is used in recursive problems

2. **Tabulation (Bottom-up):** The solution of all smaller sub-problems is computed before hand and then the solution of actual problem is solved.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of **Fibonacci numbers.**

Fibonacci (n) = 0; if n = 0

Fibonacci (n) = 1; if n = 1

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

So, the first few numbers in this series will be: 0, 1, 1, 2, 3, 5, 8, 13, 21, 35...

A code for it using pure recursion:

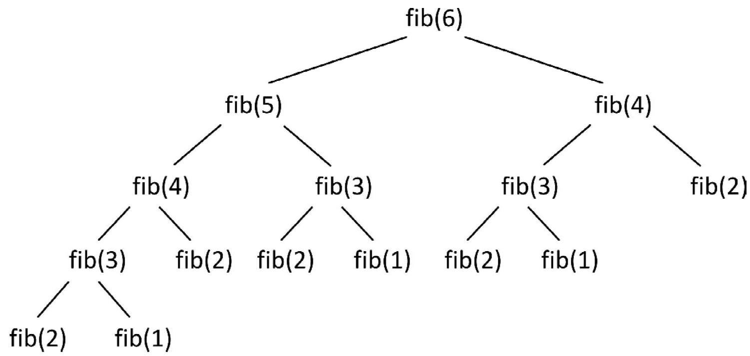
Using Dynamic Programming approach with bottom-up approach:

```
public static void fib(int[] strg, intn) {
            strg[0] = 1;
            strg[1] = 1;

            for (inti = 2; i<n; i++)
                    strg[i] = strg[i - 1] + strg[i - 2];
        }
```

Q. Are we using a different recurrence relation in the two codes? **No**

Q. Are we doing anything different in the two codes? **Yes**

Let's Visualize:



Here we are running fibonacci function multiple times for the same value of n, which can be prevented using memorization.

Optimization Problems:

Dynamic Programming is typically applied to optimization problems. In such problems there can be any possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem.

Longest Common Subsequence :

LCS Problem Statement: Given two sequences, find the length of longest subsequence present inboth of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

For example, "abc", "abg", "bdf", "aeg", '"acefg", ... etc are subsequences of "abcdefg". So a string oflength n has 2^n different possible subsequences.

Example **:**

LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4

Recurrence Relation:

LCS(str1, str2, m, n) = 0, if m = 0 or n = 0 //Base Case
LCS(str1, str2, m, n) = 1 + LCS(str1, str2, m-1, n-1), if str1[m] = str2[n]
LCS(str1, str2, m, n) = max{LCS(str1, str2, m-1, n), LCS(str1, str2, m, n-1)}, otherwise
LCS can take value between 0 and min(m,n).

Code for LCS using bottom-up approach:

```
public static intLCSBU(String s1, String s2) {
        int[][] strg = new int[s1.length() + 1][s2.length() + 1];

        for (inti = 0; i<= s1.length(); i++)
                strg[i][s2.length()] = 0;
```

```
        for (inti = 0; i<= s2.length(); i++)
              strg[s1.length()][i] = 0;

        for (introw = s1.length() - 1; row>= 0; row–) {
              for (intcol = s2.length() - 1; col>= 0; col–) {
                    if (s1.charAt(row) == s2.charAt(col))
                    strg[row][col] = strg[row + 1][col + 1] + 1;
                    else
                    strg[row][col] = Math.max(strg[row][col + 1], strg[row + 1][col]);
              }
        }
        return strg[0][0];
  }
```

T = O(mn)where m is the length of first string and n is the length of second string.

Edit Distance

Problem Statement: Given two strings str1 and str2 and below operations can be performed on str1. Find min number of edits (operations) required to convert str1 to str2.

- Insert
- Remove
- Replace

All the above operations are of equal cost.
http://www.spoj.com/problems/EDIST/

Example:
str1 = “cat”
str2 = “cut”
Replace ‘a’ with ‘u’, min number of edits = 1

str1 = “sunday”
str2 = “saturday”
Last 3 characters are same, we only need to replace “un” with “atur”.
Replace n ->r and insert ‘a’ and ‘t’ before ‘u’, min number of edits = 3

Recurrence Relation :

```
if str1[m] = str2[n]
      editDist(str1, str2, m, n) = editDist(str1, str2, m-1, n-1)
else
      editDist(str1, str2, m, n) = 1 + min{editDist(str1, str2, m-1, n) //Remove
      editDist(str1, str2, m, n-1) //Insert
      editDist(str1, str2, m-1, n-1) //Replace
```

Transform “Sunday” to “Saturday”:

Last 3 are same, so ignore. We will transform Sun ->Satur

```
(Sun, Satur) -> (Su, Satu)//Replace ‘n’ with ‘r’, cost= 1
(Su, Satu) -> (S, Sat) //Ignore ‘u’, cost = 0
(S, Sat) ->(S,Sa) //Insert ‘t’, cost = 1
(S,Sa) -> (S,S) //Insert ‘a’, cost = 1
(S,S) -> (‘’,’’) //Ignore ‘S’, cost = 0
(‘’, ‘’) ->return 0
```

```
public static intEditDistanceBU(String s1, String s2) {
        int[][]strg= new int[s1.length() + 1][s2.length() + 1];
        for (introw = s1.length(); row>= 0; row–) {
                for (intcol = s2.length(); col>= 0; col–) {
                        if (row == s1.length()) {
                                strg[row][col] = s2.length() - col;
                        } else if (col == s2.length()) {
                                strg[row][col] = s1.length() - row;
                        } else {
                                if (s1.charAt(row) == s2.charAt(col)) {
                                        strg[row][col] = strg[row + 1][col + 1];
                                } else {
                                        int insert = strg[row + 1][col];
                                        int delete = strg[row][col + 1];
                                        int replace = strg[row + 1][col + 1];
                                        strg[row][col] = Math.min(replace,
Math.min(insert, delete)) + 1;
                                }
                        }
                }
        }
        return strg[0][0];
}
```

0-1 Knapsack Problem

Problem Statement: For each item you are given its weight and its price. You want to maximize the total price of all the items you are going to put in the knapsack such that the total weight of items is less than knapsack’s capacity. What is this maximum total price?

http://www.spoj.com/problems/KNAPSACK/

To consider all subsets of items, there can be two cases for every item:

1. the item is included in the optimal subset
2. not included in the optimal set.

Therefore, the maximum price that can be obtained from n items is max of following two prices.

1. Maximum price obtained by n-1 items and W weight (excludingnth item).
2. Price of nth item plus maximum price obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

Recurrence Relation:

// Base case : If we have explored all the items or we have reached the maximum capacity of Knapsack

```
if (n == 0 || W == 0)
return 0

// If the weight of nth item is greater than the capacity of knapsack, we
cannotinclude this item
if (weight[n] >W) {
return solve(n - 1, W)
} else {
exclude = solve(n - 1, W)  // We have not included the item
       include = solve(n - 1, W - weight[n])+ price[n]     // We have included the
item in the knapsack
       return max(exclude, include)
}
```

If we build the recursion tree for the above relation, we can clearly see that the **property of overlapping sub-problems** is satisfied. So, we will try to solve it using dynamic programming.

Let us define the dp solution with states i and j as strg[i,j] ->max value that can be obtained with objects upto index i and knapsack capacity of j.

The most optimal solution to the problem will be **strg[N][W]** i.e. max value that can be obtained upto index N with max capacity of W.

Code:

```
public static int KnapsackBU(int[] weight, int[] price, int W) {
int nr = weight.length + 1;
int nc = W + 1;
int[][] strg = new int[nr][nc];
```

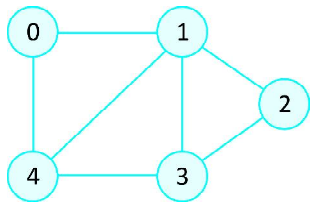
```
for (int row = 0; row<nr; row++) {

     for (int col = 0; col<nc; col++) {

          if (row == 0 || col == 0) {
                strg[row][col] = 0;
            } else {
                int include = 0;
                if (col>= weight[row - 1]) {
                include = price[row - 1] + strg[row - 1][col - weight[row - 1]];
                     }

                     int exclude = strg[row - 1][col];
                     int ans = Math.max(include, exclude);

                     strg[row][col] = ans;

              }

       }
}

return strg[nr - 1][nc - 1];
 }
```

Time Complexity: O(N*W)
Space Complexity: O(N*W)

21 Graphs

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

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



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

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

There are two types of edges:

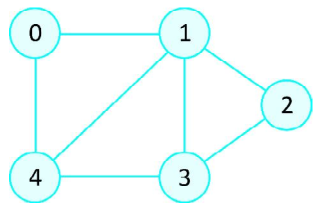
1. **Undirected Edge :** This is bidirectional edge. If there is a bi-directed edge from A to B then edge (A, B) is equal to the edge (B, A).
2. **Directed Edge :** This is unidirectional edge. If there is a uni-directed edge from A to B then edge (A, B) is not equal to the edge (B, A).

Some Real Life Applications:

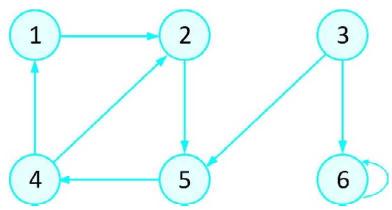
1. **Google Maps:** To find a route based on shortest route/Time.
2. **Social Networks:** Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
3. **Web Search:** Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks, each page is a vertex and the link between two pages is an edge.
4. **Recommendation System:** On E-Commerce websites relationship graphs are used to show recommendations.

Types of Graphs:

1. **Undirected:** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



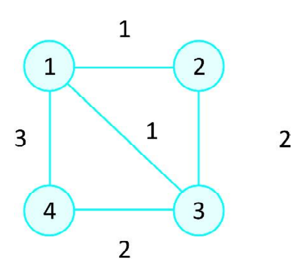
2. **Directed:** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



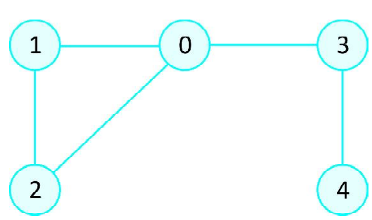
3. **Weighted:** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:

   - 1 -> 2 -> 3
   - 1 -> 3
   - 1 -> 4 -> 3

   Therefore, the total cost of each path will be as follows: The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e.3 units. The total cost of 1 -> 3 will be 1 unit. The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units.



4. **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

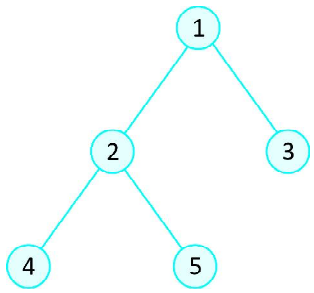


Cycle presents in the above graph (0->1->2).

A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

Note: A root node has no parent.

A tree cannot contain any cycles or self-loops, however, the same does not apply to graphs.



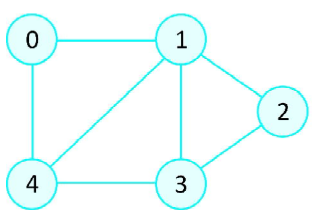
(Tree: There is no any cycle or self-loop in this graph)

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

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

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

Graph Representations :



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

1. Adjacency Matrix:

An adjacency matrix is a V”V binary matrix A. Element Ai, j is 1 if there is an edge from vertex i to vertex j else A_i, j is 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in A_i, j the weight or cost of the edge will be stored.

In an undirected graph, if $A_{i,\,j}$ = 1, then $A_{j,\,i}$= 1.

In a directed graph, if Ai, j = 1, then may or may not be 1.

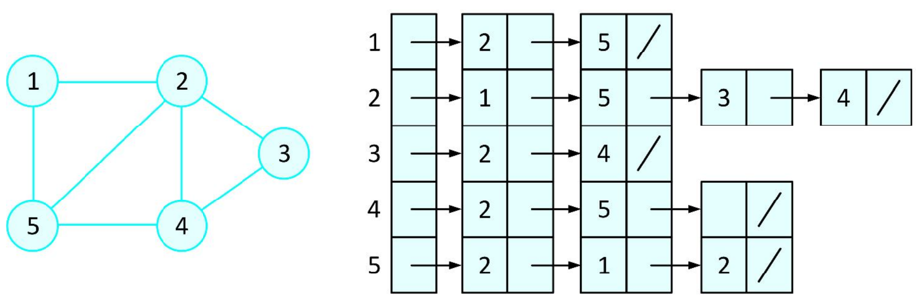
Adjacency matrix provides constant time access (O(1)) to determine if there is an edge betweentwo nodes. Space complexity of the adjacency matrix is O(V2).

2. Adjacency List:

The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array Ai is a list, which contains all the vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list Ai then vertex i will be in list Aj.

The space complexity of adjacency list is O(V + E)because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.



(a) (b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

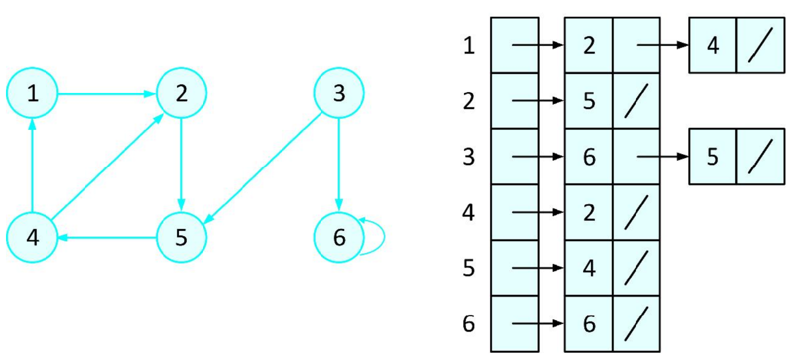
(c)

Two representations of an undirected graph.

(a) An undirected graph G having five vertices and seven edges

(b) An adjacency-list representation of G

(c) The adjacency-matrix representation of G



(a) (b)

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

(c)

Graph Traversals :

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While usingcertain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The orderin which the vertices are visited are important and may depend upon the algorithm or question thatyou are solving.

During a traversal, it is important that you track which vertices have been visited. The most common wayof tracking vertices is to mark them.

[I] Depth First Search (DFS):

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

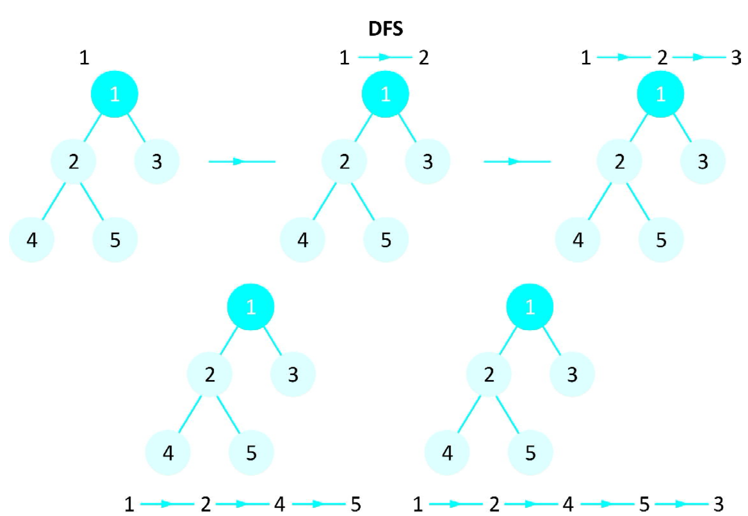
Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

As in breadth-first search, vertices are colored during the search to indicate their state.

Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.



Time Complexity:

Time complexity of above algorithm is O(V + E).

Application of DFS:

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning treeand all pair shortest path tree.
2. **Detecting cycle in a graph:** A graph has cycle if and only if we see a back edge during DFS. So we canrun DFS for the graph and check for back edge.
3. **Path Finding:** We can specialize the DFS algorithm to find a path between two given vertices u and z.
   (a) Call DFS (G, u) with u as the start vertex.
   (b) Use a stack S to keep track of the path between the start vertex and the current vertex.
   (c) As soon as destination vertex z is encountered, return the path as the contents of the stack.
4. **Topological Sorting:** In the field of computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering.
5. **To test if a graph is bipartite:** We can augment either BFS or DFS when we first discover a new vertex, color it opposite of its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.
6. **Finding Strongly Connected Components of a graph:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
7. **Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set).
8. **For finding Connected Components :** In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super-graph.

[II] Breadth First Search (BFS):

There are many ways to traverse graphs. BFS is the most commonly used approach.

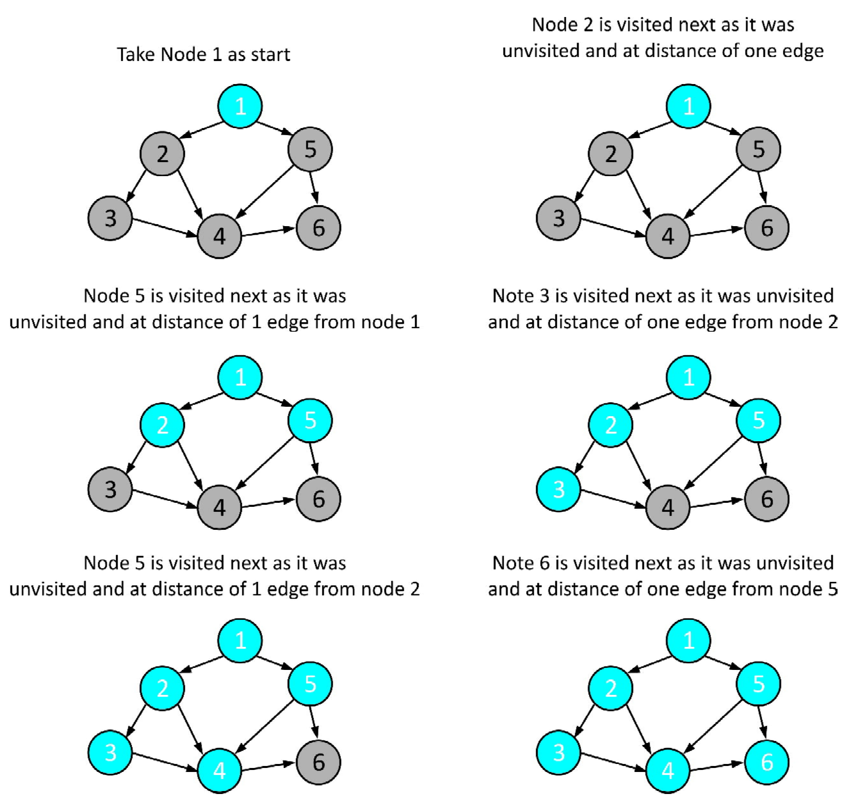
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer-wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is ***discovered***the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If (*u*, v) is an edge in *E* and vertex *u* is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex *u*, the vertex v and the edge (*u*, v) are added to the tree. We say that *u* is the ***predecessor***or ***parent***of v in the breadth-first tree.



Time Complexity:

Time complexity and space complexity of above algorithm is O(V + E) and O(V).

Application of BFS

1. **Shortest Path and Minimum Spanning Tree for unweighted graph :** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2. **Peer to Peer Networks:** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4. **Social Networking Websites:** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection:** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8. **Cycle detection in undirected graph :** In undirected graphs, either Breadth First Search or Depth.
9. First Search can be used to detect cycle. In directed graph, only depth first search can be used.
10. **Ford-Fulkerson algorithm:** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worstcase time complexity to O(VE2).
11. **To test if a graph is Bipartite:** We can either use Breadth First or Depth First Traversal.
12. **Path Finding:** We can either use Breadth First or Depth First Traversal to find if there is a pathbetween two vertices.
13. **Finding all nodes within one connected component:** We can either use Breadth First or Depth FirstTraversal to find all nodes reachable from a given node.

Minimum Spanning Tree

What is a Spanning Tree?

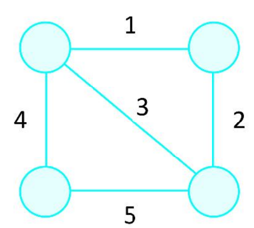
Given an undirected and connected graph G = (V,E) , a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G).

What is a Minimum Spanning Tree?

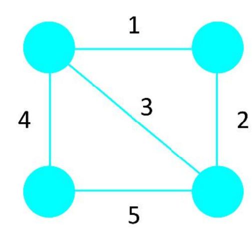
The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

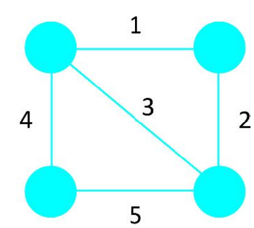


Undirected Graph



Spanning Tree

Cost = 11(= 4 + 5 + 2)



Minimum Spanning Tree

Cost = 7(= 4 + 1 + 2)

There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm :

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps :

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle, edges which connect only disconnected components.

In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first.

***Note*:** Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Time Complexity:

The total running time complexity of kruskal algorithm is O(V log E).

Prim’s Algorithm:

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal’s, we add vertex to the growing spanning tree in Prim’s.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues.

Insert the vertices that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

In Prim’s Algorithm, we will start with an arbitrary node (it doesn’t matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim’s algorithm will select the cheapest edge and mark the vertex.

Shortest Path Algorithms :

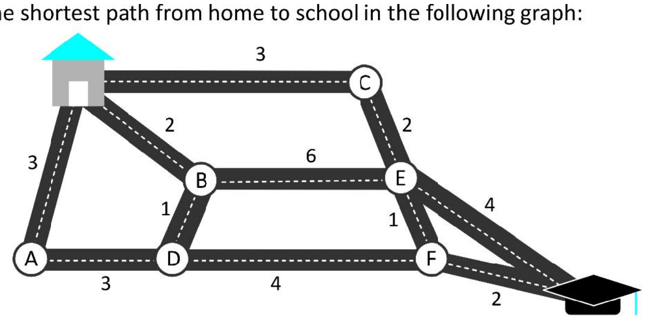
The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value. There are some algorithm discussed below which work to find shortest path between two vertices.

1. **Single Source Shortest Path Algorithm :** In this kind of problem, we need to find the shortest path of single vertices to all other vertices.

   Example:

   Find the shortest path from home to school in the following graph:



A weighted graph representing roads from home to school

The shortest path, which could be fund using Dijkstra's algorithm, is Home → B → D → F → School

The shortest path, which can be found using Dijkstra's algorithm, is

Home ->B->D->F->School

There are two algorithm works to find Single source shortest path from a given graph.

A. Dijkstra's Algorithm :

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph *G* = (*V*, *E*) for the case in which all edge weights are non-negative.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance, vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Time Complexity: Time Complexity of Dijkstra's Algorithm is O(V2) but with min- priority queue it drops down to O(V + E logV).

B. Bellman Ford's Algorithm : Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n – 1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle?
There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from 0 to n – 1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices isgradually replaced by more accurate values until eventually reaching the optimum solution. In thebeginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0,then update all the connected vertices with the new distances (source vertex distance + edgeweights), then apply the same concept for the new vertices with new distances and so on.

2. **All-Pairs Shortest Paths :** The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using n applications of Dijkstra's algorithm at each vertex if graph doesn't contains negative weight. We use two algorithms for finding All-pair shortest path of a graph.
   1. Floyd-Warshall's Algorithm
   2. Johnson's Algorithm

22 Bit Manipulation

Working on bytes, or data types comprising of bytes like int, float, double or even data structures which stores large amount of bytes is normal for a programmer.

Operations with bits are used in Data compression (data is compressed by converting it from one representation to another, to reduce the space),Exclusive-Or Encryption (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at bit level. Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

We all know that 1 byte comprises of 8 bits and any integer or character can be represented using bits in computers, which we call its binary form(contains only 0 or 1) or in its base 2 form.

Example **:**

1. $14 = \{1110\}_2$
   $= 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$
   $= 14.$

2. $20 = \{10100\}_2$
   $= 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$
   $= 20.$

Bitwise Operators:

There are different bitwise operations used in the bit manipulation. These bit operations operate on the individual bits of the bit patterns. Bit operations are fast and can be used in optimizing time complexity. Some common bit operators are:

1. **Bitwise And (&)** Bitwise AND is a binary operator that operates on two equal-length bit patterns. If both bits in the compared position of the bit patterns are 1, the bit in the resulting bit pattern is 1, otherwise 0.

AND

x	*y*	*xy*
0	0	0
0	1	0
1	0	0
1	1	1

Eg: A = 5 = (101)

B = 3 = (011)

A & B = (101)& (011) = (001) = 1

2. **Bitwise Or (|)** Bitwise OR is also a binary operator that operates on two equal-length bit patterns. If both bits in the compared position of the bit patterns are 0, the bit in the resulting bit pattern is 0, otherwise 1.

OR

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

Eg: A = 5 = $(101)_2$

B = 3 = $(011)_2$

A | B = $(101)_2$ | $(011)_2$ = $(111)_2$ = 7

3. **Bitwise XOR (^) :** Bitwise XOR also takes two equal-length bit patterns. If both bits in the compared position of the bit patterns are 0 or 1, the bit in the resulting bit pattern is 0, otherwise 1.

XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Eg: A = 5 = $(101)_2$

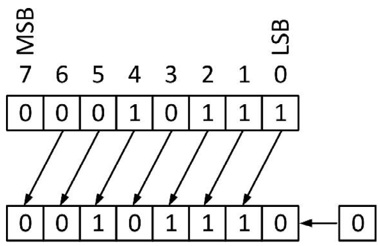
B = 3 = $(011)_2$

A ^ B = $(101)_2$ ^ $(011)_2$ = $(110)_2$ = 6

Important:

1. a ^ 1 = a' (Complement of a)
2. a ^ 0 = a (Remains Same)

4. **Left Shift (<<) :** Left shift operator is a binary operator which shift thebits towards left in the given bit pattern and appends 0 at the end. Left shift by 1 is equivalent to multiplying by 2. Left shift by k is equivalent to multiplying the bit pattern with 2^k (if we are shifting k bits).



Eg:

Number before shift was 23 (as given in the diagram) and after left shift by 1 it became 46 (or we can say 23 $*2^1$)

So, we can say that,

$1 << 1 = 2 = 2^1$

$1 << 2 = 4 = 2^2$

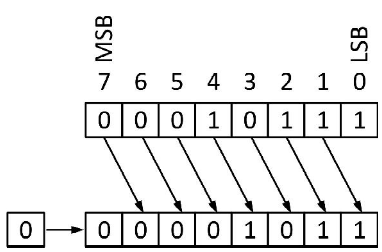
$1 << 3 = 8 = 2^3$

$1 << 4 = 16 = 2^4$

…

a<< n = a * 2^n (Generalized)

5. **Right Shift (>>) :** Right shift operator is a binary operator which shift thebits towards right in the given bit pattern and appends 0 or 1 in MSB (depending on the previous MSB).Right shift by 1 is equivalent to dividing by 2.Right shift by k is equivalent to dividing the bit pattern by 2^k (if we are shifting k bits).



Eg:

Number before shift was 23 (as given in the diagram) and after right shift by 1 it became 11 (or we can say 23 / 2^1).

So, we can say that,

$4 >> 1 = 2$

$6 >> 1 = 3$

$5 >> 1 = 2$

$16 >> 4 = 1$

…

a>> n = a / 2^n (Generalized)

6. **Not(~) :** Bitwise NOT is a unary operator that flips the bits of the number i.e. if the i^{th} bit is 0, it will change it to 1 and vice versa. Bitwise NOT is nothing but simply the one's complement of a number. Let's take an example.

NOT

x	x'
0	1
1	0

Eg: $N = 5 = (101)_2$

$\sim N = \sim 5 = \sim(101)_2 = (010)_2 = 2$

Questions:

1. **Check whether a given number is even or odd :** The idea is to divide the number with 2, if the remainder is 0 then it is even otherwise odd (Using modulus operator).

   A better solution is to use bitwise operators. We need to check whether last bit is 1 or not. If we do n & 1 then we can obtain the last bit. If last bit is 1 then the number is odd, otherwise its even.

12 ←	In decimal	→ 15
0000 1100 ←	In binary	→ 0000 1111
0000 0001 ←	Binary of 1	→ 0000 0001
0000 0000 ←	Bitwise AND operation with 1	→ 0000 0001

```
public static void isEvenOdd(intn) {
    if ((n& 1) == 0)
           System.out.println("Even");
    else
           System.out.println("Odd");
}
```

2. **Extract the i^{th} bit from a number :** To extract an.y i^{th} bit we will use mask here.

   A mask defines which bits you want to keep, and which bits you want to clear. Masking is the act of applying a mask to a value.

   In our case we want to extract the i^{th} bit, and we don't want the rest of the bits, so build a mask that has value 1 at i^{th} bit and 0 at rest of the positions.Then perform bitwise and operation that will give us answer, if the ans is non- zero then we can conclude that the value at the i^{th} bit was 1 but if the answer is 0 then we can conclude that the ith bit was 0.

   Eg: Extract 5^{th} bit from 171

   Binary representation of 171 is 0101011, 5^{th} bit will be 0

   Extract 4^{th} bit from 72

   Binary representation of 72 id 1001000, 4^{th} bit will be 1

```
public static int extractBit(intn, inti) {
int mask = 1;
mask = 1 << (i - 1);
int ans = n&mask;
return (ans == 0 ?0 : 1);
}
```

3. **Set the i^{th} bit of a number :** To set the i^{th} bit of any number again we will use mask that will set the bit at i^{th}position.

Eg: Set 4^{th} bit in 7

Binary Representation of 7 is 00000111

After setting the bit at 4^{th} position, binary representation will be 0001111

Set 5^{th} bit in 7

Binary Representation of 7 is 00000111

After setting the bit at 5^{th} position, binary representation will be 00010111

```
public static int setBit(intn, inti) {

int mask = 1;
mask = 1 << (i - 1);

n = n | mask;

return n;
}
```

4. **Given a number find the rightmost set bit :** To find the rightmost set bit we need to check every bit from starting till end.
   1. Checking every bit means to build a mask having 1 at 1^{st} position and zero at the rest of the positions initially.
   2. Perform bitwise-and operation.
   3. If the ans obtained is non-zero that means that the ith bit is one.
      3.1 Break the loop and print the position.
   4. But if ans is zero then the ith bit is zero.
      4.1 Now upgrade your mask to the 2^{nd} position and so on till u find the ith set bit.
      4.2 Print the ans.

```
public static int rightmostSetBit(intn) {

    int pos = 1;
    int mask = 1;
```

```
        while ((n&mask) == 0) {
                pos++;
                mask = mask<< 1;
        }

        return pos;
}
```

5. **Adding one to a given number using bit manipulation :** To add 1 to a number x (say 0011000111), flip all the bits after the rightmost 0 bit (we get 0011000000). Finally, flip the rightmost 0 bit also (we get 0011001000) to get the answer.
   1. To find the rightmost 0 bit in any given number we need to build a mask.
   2. Perform the bitwise-and(&) operation on every bit starting from right.
   3. If the ans is zero that means the bit is zero.
   4. But if the ans is non-zero that means the bit is 1.
      4.1 Then perform xor on that bit with the mask. As we have studied earlier performingxor on any number with 1, results into it's complement(or flipping of the bit).
      4.2 Update the mask to next bit.
   5. After the end of the loop, flip the current bit containing zero to one.
   6. Print the answer.

```
public static int addOne(intn) {
        int mask = 1;
        while ((n&mask) != 0) {
                n = n ^ mask;
                mask = (mask<< 1);
        }
        n = n ^ mask;
        return n;
}
```

6. **Find whether a no is power of 2 or not :** All power of 2 numbers have only onebit set. If we subtract a power of 2 number by 1 then all unset bits after the only set bit become set and the set bit become unset.

   Eg: n = 4 Binary of 4 = 0100
   n-1 = 3 Binary of 3 = 0011
   n & n-1 = 0
   n = 16 Binaryof 16 = 10000
   n-1 = 15 Binary of 15 = 01111
   n & n-1 = 0

So, if a number n is a power of 2 then bitwise & of n and n-1 will be zero. We can say n is a power of 2 or not based on value of n&(n-1). The expression n&(n-1) will not work when n is 0.

```
public static boolean powerOfTwo(intn) {

	if (n != 0) {
		if ((n& (n - 1)) == 0) {
			return true;
		}
	}

	return false;
}
```