Assignment No:03

Title: - Use of divide and conquer strategies to exploit distributed/parallel/concurrent processing of the above to identify objects, morphisms, overloading in functions (if any), and functional relations and any other dependencies (as per requirements).

Divide and conquer strategies

In computer science, **divide and conquer** (**D&C**) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (**divide**), until these become simple enough to be solved directly (**conquer**). The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).Understanding and designing D&C algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved.

As when proving a theorem by induction, it is often necessary to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding the proper generalization. These D&C complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion. The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

OOPS Concepts:

1. Classes
2. Objects
3. Morphism
4. Overloading in Functions
5. Functional Relations & others.

Classes

In object-oriented programming, a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (a subroutine that creates objects), and as the type of objects generated by instantiating the class; these distinct concepts are easily conflated.

When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

In some languages, classes are only a compile-time feature (new classes cannot be declared at runtime), while in other languages classes are first-class citizens, and are generally themselves objects (typically of type Class or similar). In these languages, a class that creates classes is called a metaclass.

Object

In computer science, an **object** is a location in memory having a value and possibly referenced by an identifier. An object can be a variable, a data structure, or a function. In the class-based object-oriented programming paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures. In relational database management, an object can be a table or column, or an association between data and a database entity (such as relating a person's age to a specific person).

Morphism

In many fields of mathematics, **Morphism** refers to a structure-preserving map from one mathematical structure to another. The notion of Morphism recurs in much of contemporary mathematics. In set theory, morphisms are functions; in linear algebra, linear transformations; in group theory, group homomorphisms; in topology, continuous functions, and so on. In category theory, Morphism is a broadly similar idea, but somewhat more abstract: the mathematical objects involved need not be sets, and the relationship between them may be something more general than a map.

The study of morphisms and of the structures (called objects) over which they are defined is central to category theory. Much of the terminology of morphisms, as well as the intuition underlying them, come from concrete categories, where the objects are simply sets with some additional structure, and morphisms are structure-preserving functions. In category theory, morphisms are sometimes also called **arrows**.

A category C consists of two classes, one of objects and the other of morphisms.There are two objects that are associated to every morphisms, the source and the target. For many common categories, objects are sets (usually with more structure) and morphisms are functions from an object to another object. Therefore the source and the target of morphisms are often called respectively domain and codomain. A morphisms f with source X and target Y is written
$f : X \to Y$. Thus Morphism is represented by an arrow from its source to its target.

Morphisms are equipped with a partial binary operation, called composition. The composition of two Morphism f and g is defined if and only if the target of g is the source of f, and is denoted $f \circ g$. The source of $f \circ g$ is the source of g, and the target of $f \circ g$ is the target of f. The composition satisfies two axioms:

Identity: for every object X, there exists a morphism $id_X : X \to X$ called the **identity**

Morphism: on X, such that for every morphism $f : A \to B$ we have $id_B \circ f = f = f \circ id_A$.

Associativity: $h \circ (g \circ f) = (h \circ g) \circ f$ whenever the operations are defined, that is when the target of f is the source of g, and the target of g is the source of h.

For a concrete category (that is the objects are sets with additional structure, and of the morphisms as structure-preserving functions), the identity morphisms is just the identity function, and composition is just the ordinary composition of functions. Associativity then follows, because the composition of functions is associative.

The composition of morphisms is often represented by a commutative diagram. For example,

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\
 & Z
\end{tikzcd}$$

The collection of all morphisms from X to Y is denoted $hom_C(X,Y)$ or simply hom(X, Y) and called the **hom-set** between X and Y. Some authors write $Mor_C(X,Y)$, Mor(X, Y) or C(X, Y). Note that the term hom-set is a bit of a misnomer as the collection of morphisms is not required to be a set. A category where hom(X, Y) is a set for all objects X and Y is called locally small.

Monomorphism

$f: X \to Y$ is called a monomorphism if $f \circ g_1 = f \circ g_2$ implies $g_1 = g_2$ for all morphisms g_1, $g_2: Z \to X$. It is also called a mono or a monic.

- The morphism f has a **left inverse** if there is a morphism, $g: Y \to X$ such that $g \circ f = id_X$. The left inverse g is also called a **retraction** of f. Morphisms with left inverses are always Monomorphism, but the converse is not always true in every category; a monomorphism may fail to have a left-inverse.

- A **split monomorphism** $h: X \to Y$ is a monomorphism having a left inverse $g: Y \to X$, so that $g \circ h = id_X$. Thus $h \circ g: Y \to Y$ is idempotent; that is, $(h \circ g)^2 = h \circ (g \circ h) \circ g = h \circ g$.

- In concrete categories, a function that has a left inverse is injective. Thus in concrete categories, monomorphisms are often, but not always, injective. The condition of being an injection is stronger than that of being a monomorphism, but weaker than that of being a split monomorphism.

Overloading in functions

In some programming languages, **function overloading** or **method overloading** is the ability to create multiple methods of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context.
For example, doTask() and doTask(object O) are overloaded methods. To call the latter, an object must be passed as a parameter, whereas the former does not require a parameter, and is called with an empty parameter field.

A common error would be to assign a default value to the object in the second method, which would result in an ambiguous call error, as the compiler wouldn't know which of the two methods to use. Another appropriate example would be a Print(object O) method. In this case one might like the method to be different when printing, for example, text or pictures. The two different methods may be overloaded as Print(text_object T); Print(image_object P). If we write the overloaded print methods for all objects our program will "print", we never have to worry about the type of the object, and the correct function call again and the call is always: Print (something).

Function overloading

Function overloading or method overloading is the ability to create multiple methods of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context. For example, doTask() and doTask(object O) are overloaded methods. To call the latter, an object must be passed as a parameter, whereas the former does not require a parameter, and is called with an empty parameter field. A common error would be to assign a default value to the object in the second method, which would result in an ambiguous call error, as the compiler wouldn't know which of the two methods to use.

Rules in function overloading

- The overloaded function must differ either by the data types.
- The same function name is used for various instances of function call.

It is a classification of static polymorphism in which a function call is resolved using the '**best match technique'**, i.e., the function is resolved depending upon the argument list. Method overloading is usually associated with statically-typed programming languages which enforce type checking in function calls. When overloading a method, you are really just making a number of different methods that happen to have the same name. It is resolved at compile time on the basis of which methods are used.

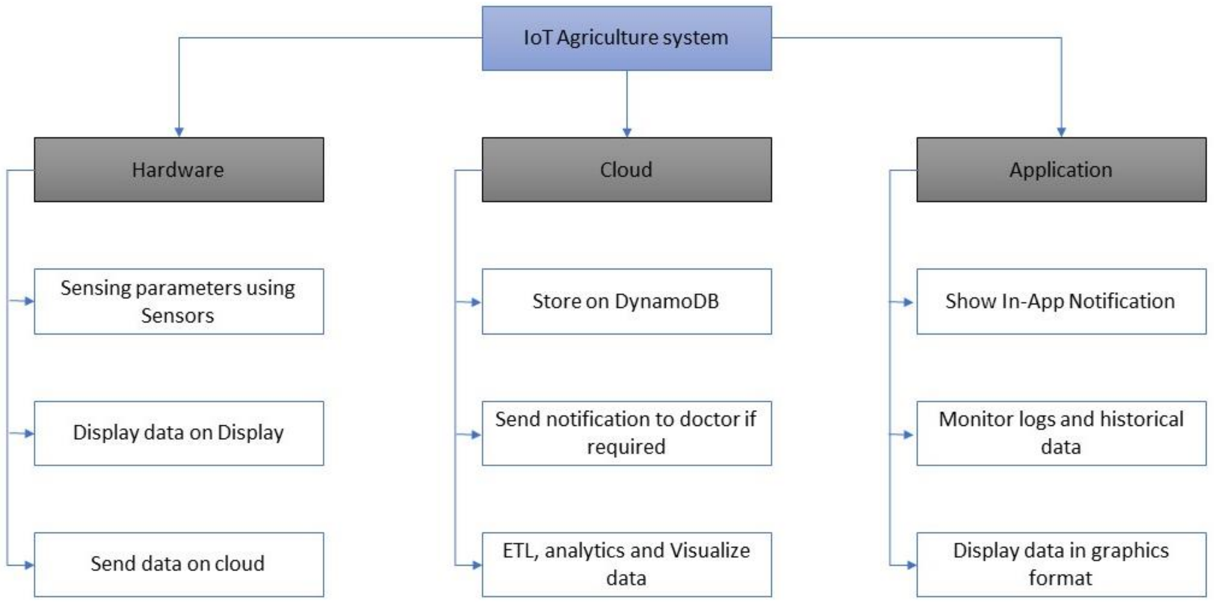
Method overloading should not be confused with forms of polymorphism where the correct method is chosen at runtime, e.g. through virtual functions, instead of statically. Polymorphism is the ability for a message or data to be processed in more than one form.

Divide and Conquer strategy:

A common Problem-Solving Heuristic in which the problem is split into two or more smaller problems which are solved separately. Often, these sub-problems can further be divided until solving each one is trivial. The sub-solutions are then combined to generate the solution to the original problem.

Divide and Conquer is more general than just algorithms. There’s no need to restrict it to using the same recursive algorithm. Divide and Conquer applies to such things as war and capitalism, for instance, where the notion of ’algorithm’ is pretty much moot.

System Modules


IoT Agriculture system
Hardware
Cloud
Application
Sensing parameters using Sensors
Display data on Display
Send data on cloud
Store on DynamoDB
Send notification to doctor if required
ETL, analytics and Visualize data
Show In-App Notification
Monitor logs and historical data
Display data in graphics format

Modules:

1. **Hardware:** In this part we are interfacing the ultrasonic sensor and actuators like display, buzzer with ESP32 Microcontroller and read values from sensor and display it on display.
2. **Cloud:** In this module, we are setting up and configure the various services of cloud (Amazon AWS). Create and Setup Amazon AWS Services – AWS DynamoDB, API Gateway, IoT Core.
3. **Data Visualization:** Amazon QuickSight is considered as one of the most powerful, fast, cloud-based easy to use business intelligence service provided by Amazon that allows user to that makes it easy to create and produce visualizations to build interactive dashboards and deliver useful insights from the data. Amazon QuickSight provides cost-effective, fast and extremely interactive business intelligence for any enterprise.
4. **Mobile Application:** In this we are developing a mobile application using Flutter framework and Dart Programming language. We are using a Flutter because we can create both android and iOS application using single code base.