Chapter – 1

INTRODUCTION

1. Philosophy of Data Structures

- Data structures are methods for organising and storing data in a computer, facilitating efficient manipulation.
- Data structure provides a way of organising, managing, and storing data efficiently.
- With the help of data structure, the data items can be traversed easily.
- Data structure provides efficiency, reusability and abstraction.
- **Data structures** are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organising, processing, accessing, and storing data efficiently

General philosophy

1. **Hardware and Program Efficiency:**
   - People might think that as computers get better, we don't need to worry about making programs efficient.
   - But in reality, as computers become more powerful, we start dealing with more complex problems, and we still need efficient programs.
2. **Dealing with Complexity in Programming:**
   - Complex tasks in programming are not like everyday experiences.
   - Programmers need to know how to design efficient programs, especially as problems get more complicated.
3. **Understanding Data Structures:**
   - Data structures are just different ways of organising and storing information in a computer.
   - Even simple things like numbers are considered basic data structures.
4. **Abstract Data Types (ADTs):**
   - Abstract Data Types are about how we organise and work with data.
   - It's all about how we represent and handle information.
5. **Choosing the Right Data Structure:**
   - Picking the right way to organise data makes programs work better.

- For example, searching for something in a well-organised list is faster than in a messy one.

6. **Efficient Solutions and Limits:**
   - Efficient solutions are ones that work well without using too much computer space or time.
   - Computers have limits on how much space and time they can use, and efficient solutions stay within those limits.

7. **Balancing Cost and Resource Use:**
   - A good solution uses fewer resources (like time) compared to other ways, while still meeting the requirements.
   - Finding the right balance means making a program that works well without using up too many computer resources.

Operations that can be performed in a data structure [2014 Fall]

➲ Next, we will look at some of the most common data structure operations.

➲ Data structure operations are the methods used to manipulate the data in a data structure. The most common data structure operations are:

1. **Traversal:** Traversal operations are used to visit each node in a data structure in a specific order. This technique is typically employed for printing, searching, displaying, and reading the data stored in a data structure.
2. **Insertion:** Insertion operations add new data elements to a data structure. You can do this at the data structure's beginning, middle, or end.
3. **Deletion:** Deletion operations remove data elements from a data structure. These operations are typically performed on nodes that are no longer needed.
4. **Searching:** Search operations are used to find a specific data element in a data structure. These operations typically employ a compare function to determine if two data elements are equal.
5. **Sorting:** Sort operations are used to arrange the data elements in a data structure in a specific order. This can be done using various sorting algorithms, such as insertion sort, bubble sort, merge sort, and quick sort.
6. **Merge:** Merge operations are used to combine two data structures into one. This operation is typically used when two data structures need to be combined into a single structure.
7. **Copy:** Copy operations are used to create a duplicate of a data structure. This can be done by copying each element in the original data structure to the new one.

Need of Data structure :

The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.

Data structures provide an easy way of organising, retrieving, managing, and storing data.

Here is a list of the needs for data.

1. Data structure modification is easy.
2. It requires less time.
3. Save storage memory space.
4. Data representation is easy.
5. Easy access to the large database.
6. **Structured Storage:** Data structures provide a systematic way to store and organise information, making it easier to manage and retrieve.
7. **Speedy Retrieval:** They enable swift data retrieval, ensuring that accessing specific pieces of information doesn't become a time-consuming process.
8. **Algorithmic Enhancement:** Properly chosen data structures enhance the efficiency of algorithms, contributing to faster and more optimised computational processes.
9. **Memory Efficiency:** Data structures contribute to efficient memory management, preventing unnecessary use of resources and ensuring effective utilisation.
10. **Task-Specific Solutions:** Different data structures are designed for specific tasks, allowing programmers to choose the most suitable structure for a particular job, increasing flexibility.
11. **Resource Utilisation Precision:** They assist in the precise utilisation of computational resources, helping to minimise wastage of time and space in program execution.
12. **Improved Problem Solving:** Data structures provide a toolkit for solving complex problems, enabling programmers to devise elegant and effective solutions.
13. **Enhanced Program Adaptability:** The right data structures contribute to the adaptability of programs, allowing them to handle diverse datasets and computational scenarios with ease.
14. **Streamlined Operations:** They streamline operations like searching, sorting, and updating data, making these fundamental tasks more manageable and less resource-intensive.

15. **Foundation of Software Design:** Serving as the foundation of software design, data structures are pivotal in creating robust, scalable, and high-performance applications.

1.2 Characteristics of Data Structures

- **Organisation:** Data structures provide a systematic way to organise and store data, promoting efficient management.
- **Accessibility:** They facilitate quick and easy access to data, ensuring that information can be retrieved promptly.
- **Efficiency:** Data structures are designed to enhance the efficiency of algorithms, making operations like searching and sorting more streamlined.
- **Versatility:** Different data structures suit various tasks, allowing programmers to choose the most appropriate structure for a specific job.
- **Memory Management:** They contribute to effective memory management, optimising the utilisation of computer memory resources.
- **Flexibility:** Data structures provide flexibility in handling diverse datasets and adapting to different computational scenarios.
- **Task-Specific Design:** Tailored to specific tasks, data structures are crafted to address the unique requirements of different types of information processing.
- **Resource Utilisation:** They assist in the efficient use of computational resources, minimising wastage of time and space.
- **Ease of Operations:** Data structures simplify fundamental operations like searching, sorting, and updating data, making them more manageable.
- **Foundation of Algorithms:** Serving as the foundation of algorithmic design, data structures are fundamental to the creation of effective and optimised computational solutions.

1.2.1 Advantages of Data Structure

- Data structures allow storing the information on hard disks.
- An appropriate choice of ADT (Abstract Data Type) makes the program more efficient.
- Data Structures are necessary for designing efficient algorithms.
- It provides reusability and abstraction.
- Using appropriate data structures can help programmers save a good amount of time while performing operations such as storage, retrieval, or processing of data.
- Manipulation of large amounts of data is easier.

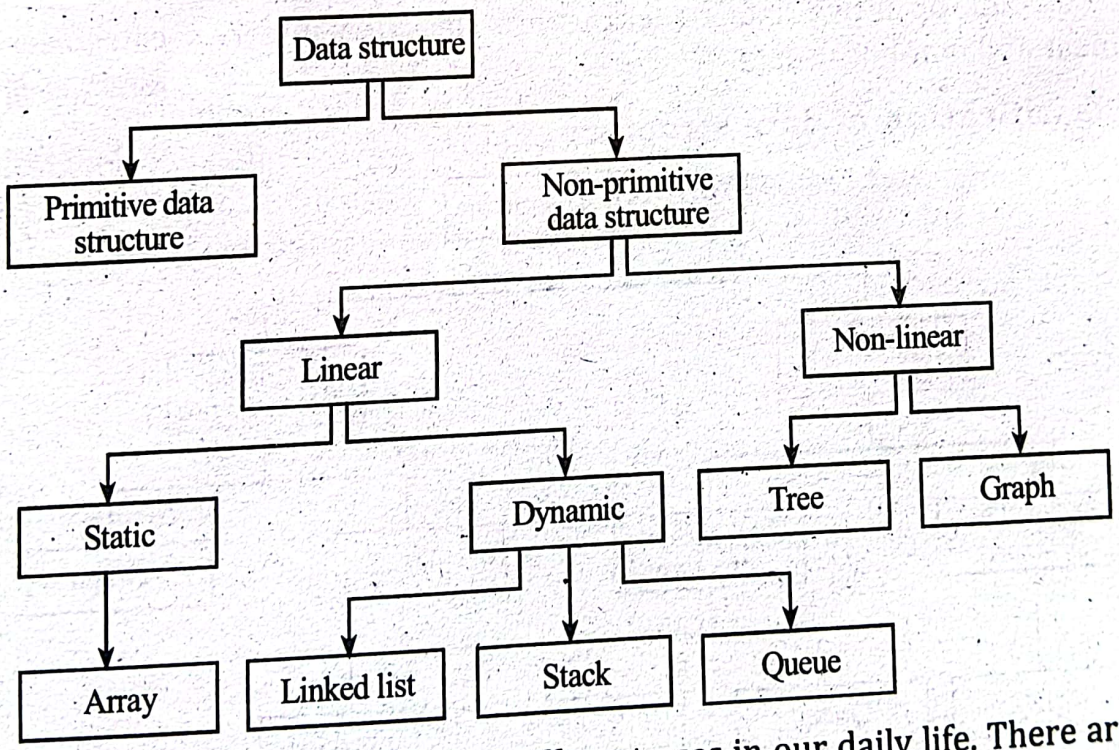
1.2.2 Data Structure Applications

1. Organisation of data in a computer's memory
2. Representation of information in databases
3. Algorithms that search through data (such as a search engine)
4. Algorithms that manipulate data (such as a word processor)
5. Algorithms that analyse data (such as a data miner)
6. Algorithms that generate data (such as a random number generator)
7. Algorithms that compress and decompress data (such as a zip utility)
8. Algorithms that encrypt and decrypt data (such as a security system)
9. Software that manages files and directories (such as a file manager)
10. Software that renders graphics (such as a web browser or 3D rendering software)

1.2.3 Types of Data Structure

[2013 Spring]

Classification of Data Structure:



Data structure has many different uses in our daily life. There are many different data structures that are used to solve different mathematical and logical problems. By using data structure, one can organise and process a very large amount of data in a relatively short period. Let's look at different data structures that are used in different situations.

Primitive data structure

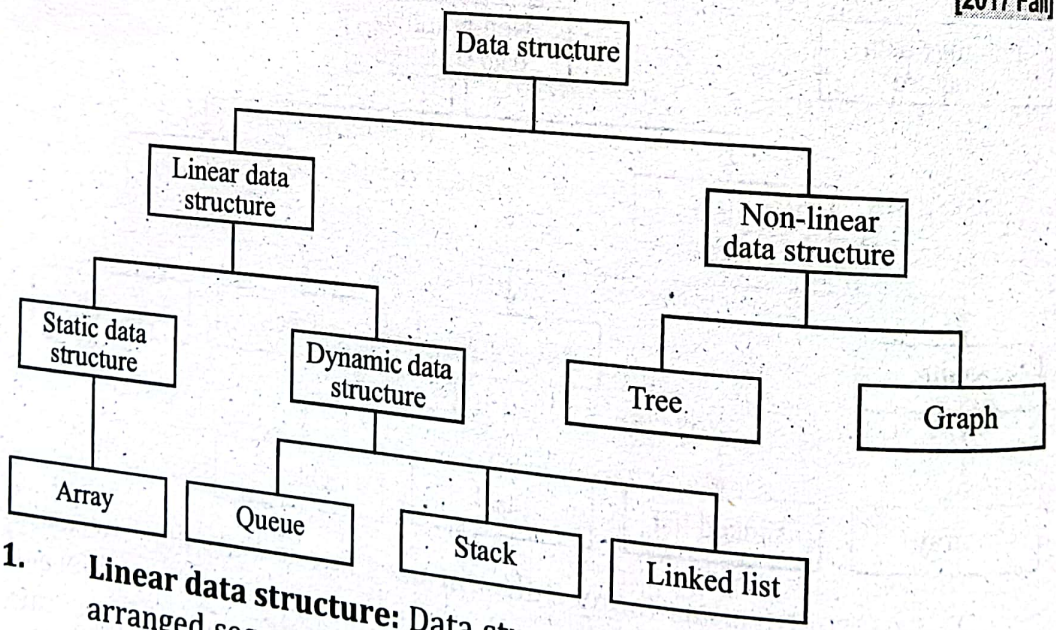
- Primitive data structures are the **fundamental data structures.**
- It can be operated directly on the data and machine instructions.
- It is also known as basic data structure.
- Primitive data structures are defined by the programming languages, or we can say that it is **built-in.**
- Some of the Primitive data types are integer, real, character, floating point number, and **pointer.**
- Basically, 'data-type', 'data structure' are often used interchangeably.

Non-primitive data structures

- Non-primitive data structures are the data structures that are created using the primitive data structures.
- It is a little bit complicated as it is derived from primitive data structures.
- Some Non-primitive data structures are linked lists, stacks, trees, and graphs.
- Also we can say that is a grouping of the same or different data items.

Non-primitive data structures are classified into two categories linear and non-linear data structure [2019 Spring]

Classification of data structure [2017 Fall]



1. **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.

   Examples of linear data structures are array, stack, queue, linked list, etc.

- **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

  An example of this data structure is an array.

- **Dynamic data structure:** In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.

  Examples of this data structure are queue, stack, etc.

2. **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.

   Examples of non-linear data structures are trees and graphs.

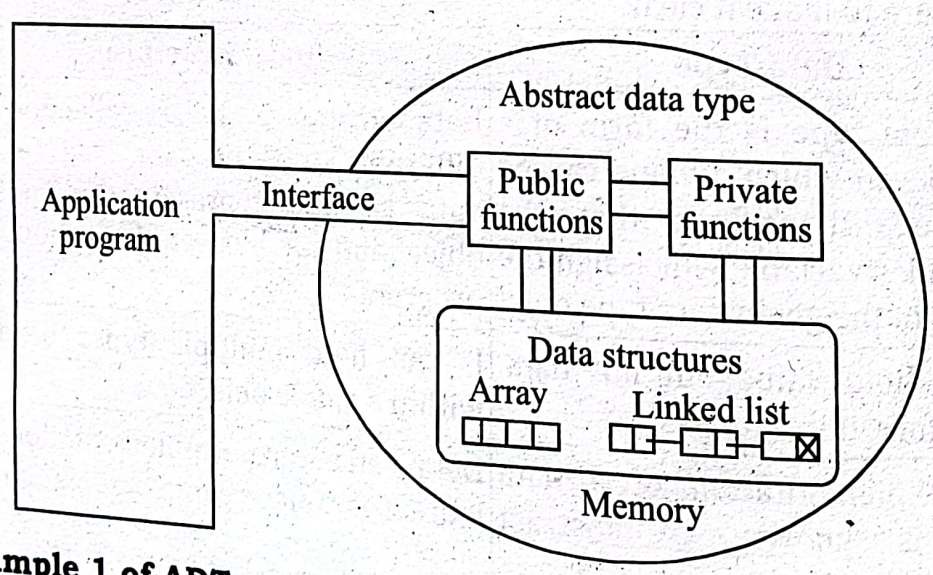
Q. How Data Structure varies from Data Type: **[2016 Spring]**

We already have learned about data structure. Many times, what happens is that people get confused between data type and data structure. So let's see a few differences between data type and data structure to make it clear.

Data Type	Data Structure
The data type is the form of a variable to which a value can be assigned. It defines that the particular variable will assign the values of the given data type only.	Data structure is a collection of different kinds of data. That entire data can be represented using an object and can be used throughout the program.
It can hold value but not data. Therefore, it is dataless.	It can hold multiple types of data within a single object.
The implementation of a data type is known as abstract implementation.	Data structure implementation is known as concrete implementation.
There is no time complexity in the case of data types.	In data structure objects, time complexity plays an important role.
In the case of data types, the value of data is not stored because it only represents the type of data that can be stored.	While in the case of data structures, the data and its value acquire the space in the computer's main memory. Also, a data structure can hold different kinds and types of data within one single object.
Data type examples are int, float, double, etc.	Data structure examples are stack, queue, tree, etc.

2. Abstract Data type (ADT) and Data Structures [2013/018/ 023 Fall]

- Abstract Data Type (ADT) is a concept in programming where a user can define a new type of data along with the operations that can be performed on it.
- It only defines how, not how the operations are performed.
- Unlike built-in data types such as integers or floats, ADTs are user-created, encapsulating both data and the functions that manipulate it.
- **The key feature of ADTs is their abstraction:** they specify what operations can be done without revealing how those operations are implemented.
- This abstraction allows for a clear separation between the functionality a user wants and the internal details of how it's achieved, promoting modular and organised software design.
- In essence, ADTs provide a way to structure and organise data and operations in a manner that enhances code clarity and problem-solving capabilities.
- The process of providing only the essentials and hiding the details is known as abstraction.



Example 1 of ADT

Let's understand the abstract data type with a real-world example.

If we consider the smartphone. We look at the high specifications of the smartphone, such as:

- 4 GB RAM
- Snapdragon 2.2ghz processor
- 5 inch LCD screen
- Dual camera
- Android 8.0

The above specifications of the smartphone are the data, and we can also perform the following operations on the smartphone:

- **call():** We can call through the smartphone.
- **text():** We can text a message.
- **photo():** We can click a photo.
- **video():** We can also make a video.

Here, we have all functions that are so abstract we don't know how to make a call. How text is sent? How is the photo taken? And how video is generated inside the machine and we only know about what is done. So they are called ADT. The smartphone is an entity whose data or specifications and operations are given above. The abstract/logical view and operations are the abstract or logical views of a smartphone.

Example 2 of ADT

Example: Bank Account

- **Data (Specifications):**
  - Balance
  - Account Holder's Name
  - Account Number
  - Account Type (Savings/Checking)
- **Operations:**
  - **Deposit():** Add money to the account.
  - **Withdraw():** Remove money from the account.
  - **Check_Balance():** View the current balance.
  - **Transfer():** Move money between accounts.
  - **Get_Account_Information():** Retrieve account details.

Explanation:

In this example, a bank account can be viewed as an abstract data type. The data includes attributes such as balance, account holder's name, account number, and type. The operations define actions that can be performed on the bank account, like depositing money, withdrawing money, checking the balance, transferring funds, and obtaining account information.

Abstract/Logical View:

The abstract view focuses on what can be done with the bank account (operations) and what information it contains (data), without delving into how these operations are implemented internally by the bank.

This abstraction allows users to interact with their bank accounts without needing to understand the intricate details of the banking

system's implementation. It provides a clear separation between what the account is and what can be done with it, which aligns with the concept of Abstract Data Types.

Example 3: Representing Natural Number as ADT [2015 Fall]

Mathematical Properties and Operations

Set of Natural Numbers

Let N represent the set of natural numbers.

N = {1, 2, 3, 4, ...}

Operations

1. **Addition**
   - For any two natural numbers a, b in N, their sum is denoted as a + b and belongs to N.
   - a + b = b + a (Commutative Property)
   - (a + b) + c = a + (b + c) (Associative Property)
2. **Multiplication**
   - For any two natural numbers a, b in N, their product is denoted as a * b and belongs to N.
   - a * b = b * a (Commutative Property)
   - (a * b) * c = a * (b * c) (Associative Property)
3. **Subtraction**
   - Subtraction is defined as a partial operation for a, b in N such that a - b is defined if b <= a, and the result belongs to N.
4. **Division**
   - Division is defined as a partial operation for a, b in N such that a / b is defined if b != 0 and b divides a without leaving a remainder.
5. **Ordering**
   - There exists a total order on N, denoted as <, such that for any a, b in N: a < b means a is less than b. a > b means a is greater than b. a = b means a is equal to b.
6. **Mathematical Properties:** Associative and Commutative
   - **Associative Property of Addition**
     - For any three natural numbers a, b, and c in N:
       (a + b) + c = a + (b + c)
   - **Commutative Property of Addition**
     - For any two natural numbers a and b in N:
       a + b = b + a

- **Associative Property of Multiplication**
  - For any three natural numbers a, b, and c in N:
    $(a * b) * c = a * (b * c)$
- **Commutative Property of Multiplication**
  - For any two natural numbers a and b in N:
    $a * b = b * a$

Other properties and operations may be defined based on specific needs.

Example 3: Representing Rational number as ADT [2013/016 Fall]

- **Fractional Representation:** A rational number is represented as a fraction, where the numerator and denominator are integers, and the denominator is not zero. **Example:** 3/4
- **Decimal Representation:** A rational number is represented as a decimal, which can be a finite or repeating decimal. **Example:** 0.25
- **Mixed Number Representation:** A rational number is represented as a whole number combined with a proper fraction. **Example:** 2 1/3
- **Ratio of Integers Representation:** A rational number is represented as the ratio of two integers. **Example:** -6/9
- **Floating-Point Representation:** A rational number is represented using a floating-point number, which may introduce precision limitations. **Example:** –0.75
- **Percentage Representation:** A rational number is represented as a percentage, where the numerator represents a percentage of the denominator. **Example:** 25% (Equivalent to 1/4)
- **Scientific Notation Representation:** A rational number is represented in scientific notation, expressing it as a product of a coefficient and a power of 10. **Example:** 3.0 x 10^2 (Equivalent to 300)

Example 4: Matrix as ADT [2019 Fall]

Let consider two matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

1. **Matrix addition:** $C = A + B = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$
2. **Matrix subtraction:** $D = A - B = \begin{bmatrix} 1-5 & 2-6 \\ 3-7 & 4-8 \end{bmatrix} = \begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$

3. **Matrix multiplication:**

$$E = A \times B = \begin{bmatrix} 1(5)+2(7) & 1(6)+2(8) \\ 3(5)+4(7) & 3(6)+4(8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

4. **Scalar multiplication:** $F = 2 \times A = \begin{bmatrix} 2(1) & 2(2) \\ 2(3) & 2(4) \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$

5. **Matrix transposition:** $G = \text{transpose}(A) = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

6. **Determinant calculation:** $dct(A) = (1 \times 4) - (2 \times 3) = -2$

7. **Inverse calculation:** $A^{-1} = \frac{1}{dct(A)} \times adj(A)$

   where, adj(A) is the adjugate of matrix A.

8. **Matrix equality:**

   $A = A$

   These examples demonstrate basic operations of matrix addition, subtraction, multiplication, scalar multiplication, transposition, determinant calculation, and matrix equality within the context of the matrix ADT. Specific calculations for inverse and adjugate matrices are typically more involved and might require additional steps.

Sample product function to find the product of two matrices. [2019 Fall]

For two matrices A and B where A is an m × p matrix and B is a p × n matrix, their product C = A × B is given by:

$$C_{ij} = \sum_{k=1}^{p} A_{ik} \times B_{kj}$$

Here,

- C_{ij} represents the element in the i-th and j-th column of the resulting matrix C.
- A_{ik} represents the element in the i-th row and k-th column of matrix A.
- B_{kj} represents the element in the k-th row and j-th column of matrix B.
- The sum is taken over values of k from 1 to p.

This formula captures the essence of matrix multiplication. The resulting matrix C will have dimensions m × n. Each element in C is obtained by taking the dot product of the corresponding row from matrix A and column from matrix B.

Features of ADT: [2017 Spring]

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure; only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation:** ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction:** ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence:** ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorised users and operations. This helps prevent errors and misuse of the data.
- **Modularity:** ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organising and manipulating data in a structured and efficient manner.

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

Advantages: [2017 Spring]

- **Encapsulation:** ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction:** ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.

- **Data Structure Independence:** ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- **Information Hiding:** ADTs can protect the integrity of data by controlling access and preventing unauthorised modifications.
- **Modularity:** ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

- **Overhead:** Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity:** ADTs can be complex to implement, especially for large and complex data structures.
- **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- **Cost:** Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Class vs Structure

Structure	Class
It is a value type.	It is a reference type.
Its object is created on the stack memory.	Its object is created on the heap memory.
It does not support inheritance.	It supports inheritance.
The member variable of structure cannot be initialized directly.	The member variable of class can be initialize directly.
It can have only parameterized constructor.	It can have all the types of constructor and destructor.

Class	Structure
1. Members of a class are private by default.	1. Members of a structure are public by default.
2. An instance of a class is called an 'object'.	2. An instance of structure is called the 'structure variable'.

Class	Structure
3. Member classes/structures of a class are private by default but not all programming languages have this default behaviour eg Java etc.	3. Member classes/ structures of a structure are public by default.
4. It is declared using the class keyword.	4. It is declared using the struct keyword.
5. It is normally used for data abstraction and further inheritance.	5. It is normally used for the grouping of data
6. NULL values are possible in Class.	6. NULL values are not possible.

7. **Syntax:**
class class_name {
data_member;
member_function;
};
	7. **Syntax:**
	struct structure_name {
	type structure_member1;
	type structure_member2;
	};

Data Type vs Abstract Data Type

Aspect	Datatype	Abstract Data Type (ADT)
Definition	Predefined by the language.	User-defined, encapsulating data and operations.
Operations	Built-in, standard operations.	User-defined, specific to the structure.
Implementation Details	Transparent and language-specific.	Hidden, providing a black-box view.
Flexibility	Limited for custom operations.	High, allows defining custom structures.
Example (Datatype)	int, float, char.	N/A (ADT is a concept). Stack (push, pop), Queue (enqueue, dequeue).

3. Algorithm Design Technique

3.1 Divide and Conquer

➲ A divide and conquer algorithm breaks down the complexity of its problem so it can solve smaller and easier sub-problems. It involves three major steps:

- **Divide:** Divide the problem into multiple sub-problems of the same nature
- **Solve:** Solve each resulting sub-problem
- **Combine:** Combine the solutions to the sub-problems to get the solution to the starting problem

A divide and conquer algorithm handles each sub-problem separately. Such algorithms give the most optimal solution for problems like efficiently sorting a collection of elements.

Example

Thanks to their simple approach, it isn't hard to understand divide and conquer algorithms. There are many divide and conquer algorithm examples in the real world. For example, take the common problem of looking for a lost item in a huge space. It is easier to divide the space into smaller sections and search in each separately.

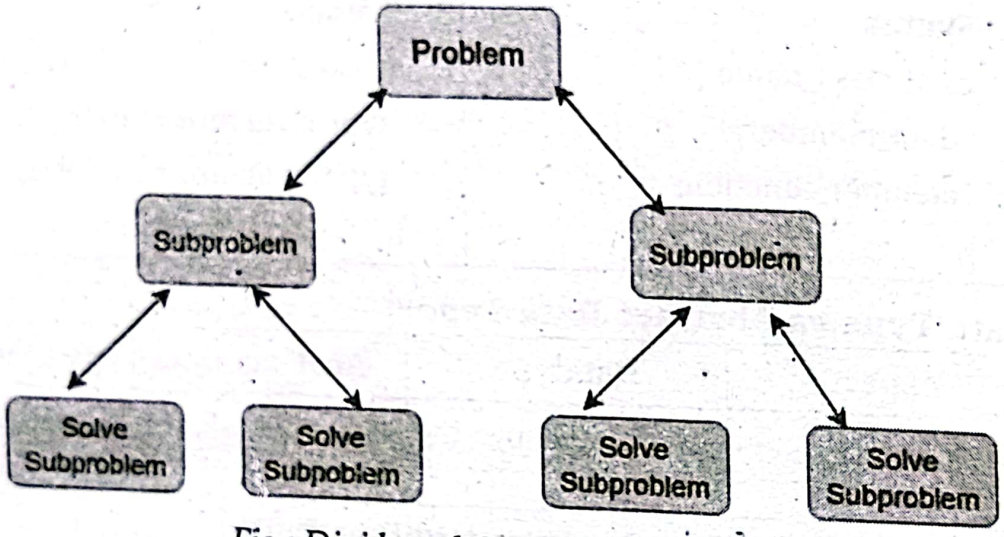


Fig.: Divide and Conquer Algorithm

3.2 Greedy Algorithm

- Greedy algorithms craft a solution piece by piece, and their selection criteria when selecting the next piece is that it should be instantly fruitful.
- Hence, the algorithm evaluates all the options at each step and chooses the best one at the moment. However, they aren't beneficial in all situations.
- A greedy algorithm solution isn't necessarily an overall optimal solution since it only goes from one best solution to the next.
- Additionally, there is no backtracking involved if it chooses the wrong option or step.

Example

- Greedy algorithms are the best option for certain problems.

- A popular example of a greedy algorithm is sending some information to the closest node in a network.
- Some other graph-based greedy algorithm examples are: Dijkstra's Algorithm Prim and Kruskal's Algorithm Huffman Coding Tree.

When attempting to find the largest sum of numbers in a node, a greedy algorithm (blue) lacks the foresight to pick a suboptimal (green) in order to eventual find the optimal solution

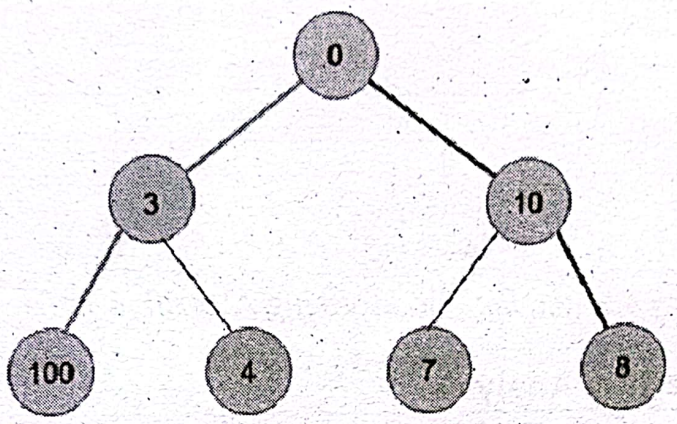


Fig.: Greedy Algorithm

3.3 Backtracking

- A backtracking algorithm finds all the possible combinations of a solution and evaluates if it isn't optimal.
- If it isn't, the algorithm backtracks and starts evaluating other solutions. Backtracking algorithms share a common approach with the brute force algorithm design technique.
- However, they are much faster than brute-force algorithms.

There are different kinds of backtracking algorithms based on the kind of problems they solve:

- **Decision Problem:** Find a feasible solution
- **Optimization Problem**: Find the most optimal solution
- **Enumeration Problem:** Find all feasible solutions

Example

Backtracking algorithms are the most optimal for problems where we may need to go back a few steps and make different decisions. For example, one of the most famous backtracking algorithm examples is the one for solving crossword puzzles. Similarly, the eight queens puzzle also requires going back if the current solution isn't the right one.

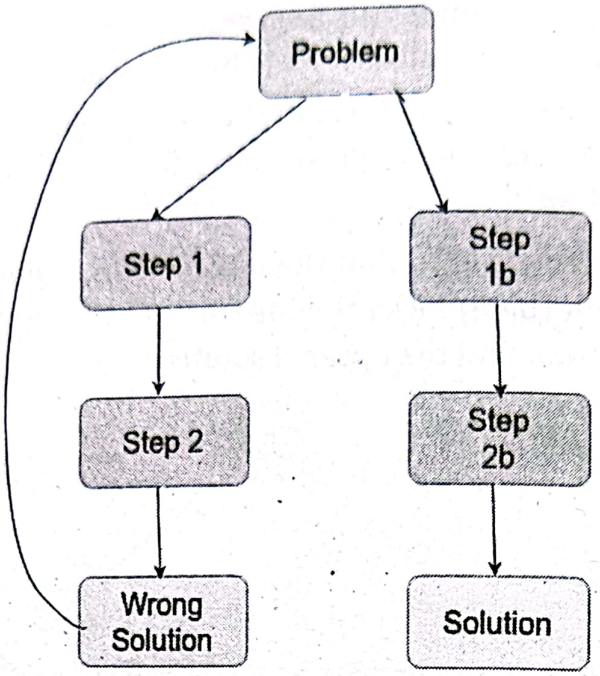


Fig.: Backtracking Algorithm

4. Algorithm Analysis

Why is algorithm analysis important?

- To predict the behaviour of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behaviour of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analysing different algorithms, we can compare them to determine the best one for our purpose.

Performance: How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

Complexity: How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Note: Complexity affects performance but not vice-versa.

4.1 Best Case, Worst Case and Average Case Analysis

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case, calculate the lower bound of an algorithm.

Example: In the linear search when search data is present at the first location of large data then the best case occurs.

- **Worst Case:** Define the input for which algorithm takes a long time or maximum time. In the worst case, calculate the upper bound of an algorithm.

  Example: In the linear search when search data is not present at all then the worst case occurs.

- **Average case:** In the average case take all random inputs and calculate the computation time for all inputs.

  And then we divide it by the total number of inputs.

$$\text{Average case} = \frac{\text{All random case time}}{\text{Total no of case}}$$

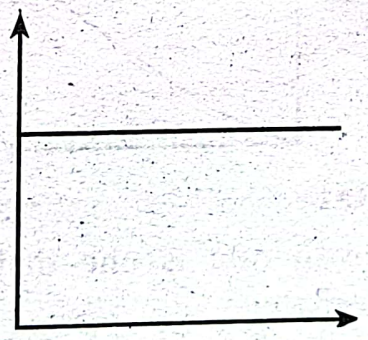
4.2 Rate of Growth

Rate of growth is defined as the rate at which the running time of the algorithm is increased when the input size is increased.

There are many growth rate function they are:

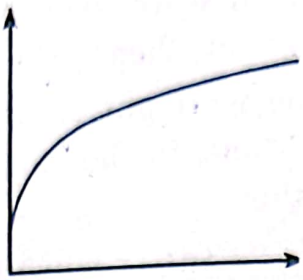
Constant Growth Rate

- A constant resource need is one where the resource need does not grow.
- That is, processing 1 piece of data takes the same amount of resources as processing 1 million pieces of data.
- The graph of such a growth rate looks like a horizontal line



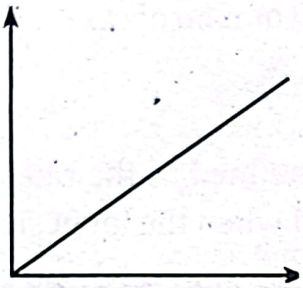
Logarithmic Growth Rate

- A logarithmic growth rate is a growth rate where the resource needs grows by one unit each time the data is doubled.
- This effectively means that as the amount of data gets bigger, the curve describing the growth rate gets flatter (closer to horizontal but never reaching it).
- The following graph shows what a curve of this nature would look like.



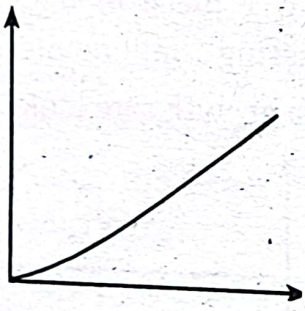
Linear Growth Rate

- A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other.
- That is the growth rate can be described as a straight line that is not horizontal.



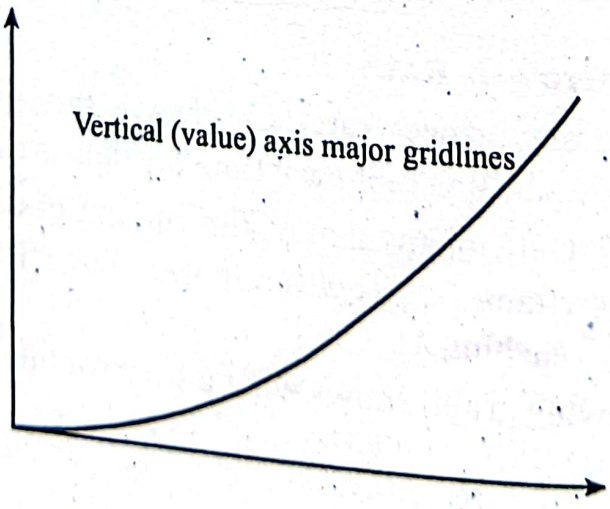
Log Linear

- A log linear growth rate is a slightly curved line. the curve is more pronounced
- for lower values than higher ones



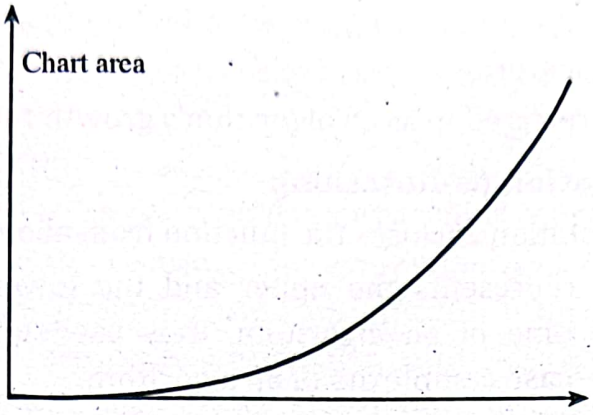
Quadratic Growth Rate

- A quadratic growth rate is one that can be described by a parabola.



Cubic Growth Rate

- While this may look very similar to the quadratic curve, it grows significantly faster



Exponential Growth Rate

- An exponential growth rate is one where each extra unit of data requires a doubling of resources. As you can see the growth rate starts off looking like it is flat but quickly shoots up to near vertical (note that it can't actually be vertical)

Growth-rate functions:

- O(1) – constant time, the time is independent of n, e.g. array loop-up
- O(log n) – logarithmic time, usually the log is base 2, e.g. binary search
- O(n) – linear time, e.g. linear search.
- O(n*log n) – e.g. efficient sorting algorithms
- $O(n^2)$ – quadratic time, e.g. selection sort
- $O(n^k)$ – polynomial (where k is some constant)
- $O(2^n)$ – exponential time, very slow!
- Order of growth of some common functions

  $O(1) < O(\log n) < O(n) < O(n*\log n) < O(n^2) < O(n^3) < O(2^n)$

4.3 Asymptotic Notations: Big Oh, Big Omega and Big Theta

- Asymptotic Notations are mathematical tools that allow you to analyse an algorithm's running time by identifying its behaviour as its input size grows.
- This is also referred to as an algorithm's growth rate.

1. Theta Notation (Θ-Notation):

- Theta notation encloses the function from above and below.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the **average-case** complexity of an algorithm.
- Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n \geq n0$

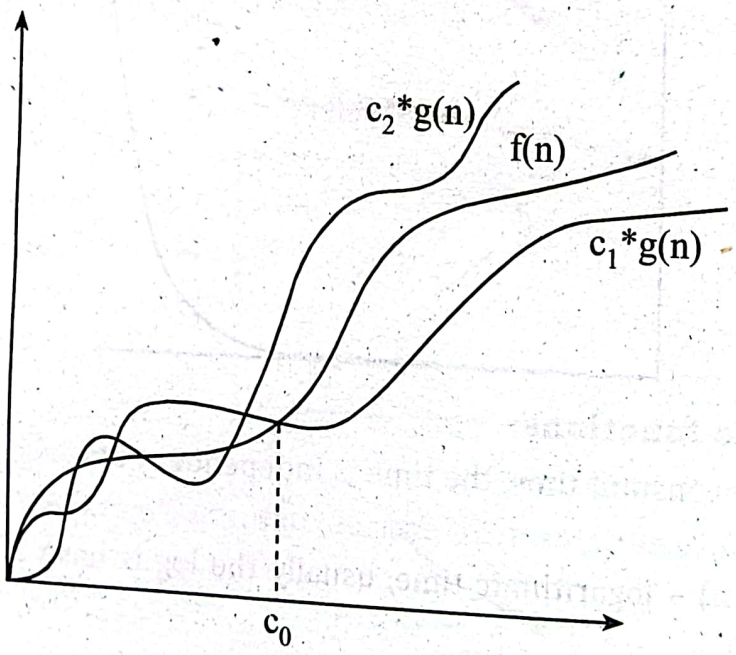


Fig.: Theta notation

Mathematical Representation of Theta notation:

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that $0 \leq c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n \geq n0$}

Note: Θ(g) is a set

- The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between $c1 * g(n)$ and $c2 * g(n)$ for large values of n ($n \geq n0$). The definition of theta also

requires that f(n) must be non-negative for values of n greater than n0.

- The execution time serves as both a lower and upper bound on the algorithm's time complexity.
- It exists as both, most, and least boundaries for a given input value.
- A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants.
- For example, Consider the expression **$3n^3 + 6n^2 + 6000 = \Theta(n^3)$**, the dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved. For a given function g(n), we denote $\Theta(g(n))$ is following set of functions.

Examples:

{ 100, log (2000), 10^4 } belongs to Θ(1)

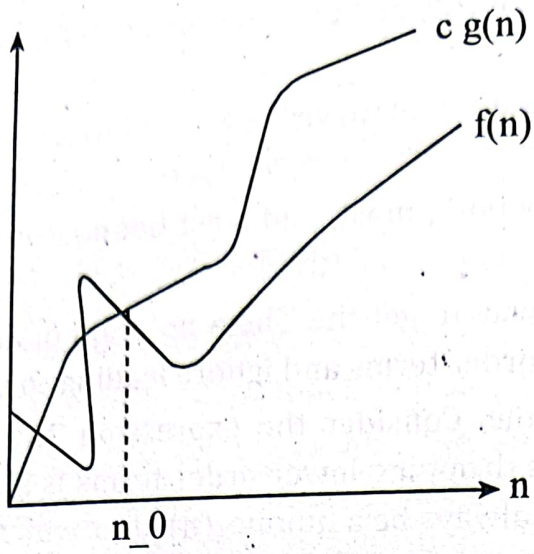
{ (n/4), (2n+3), (n/100 + log(n)) } belongs to Θ(n)

{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to ***Θ(n2)***

Note: *Θ provides exact bounds.*

2. Big-O Notation (O-notation):

- Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.
- It is the most widely used notation for Asymptotic analysis.
- It specifies the upper bound of a function.
- The maximum time required by an algorithm or the worst-case time complexity.
- It returns the highest possible output value(big-O) for a given input.
- Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.
- If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$
- It returns the highest possible output value (big-O)for a given input.
- The execution time serves as an upper bound on the algorithm's time complexity.



f(n) = O(g(n))

Mathematical Representation of Big-O Notation:

O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

- For example, Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion sort is O(n2).

  Note: O(n2) also covers linear time.

If we use Θ notation to represent the time complexity of Insertion sort, we have to use two statements for best and worst cases:

- The worst-case time complexity of Insertion Sort is Θ(n2).
- The best case time complexity of Insertion Sort is Θ(n).

The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

Examples:

{ 100 , log (2000) , 10^4 } belongs to ***O(1)***

U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to ***O(n)***

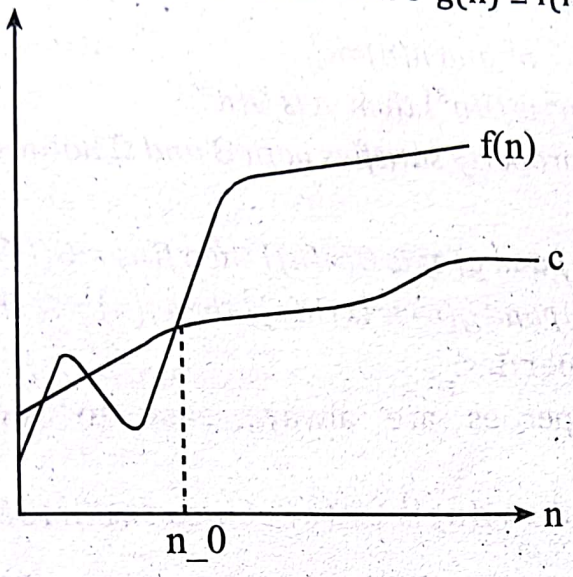
U { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to ***O(n^2)***

Note: Here, U represents union, we can write it in this manner because O provides exact or upper bounds.

3. **Omega Notation (Ω-Notation):**

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.
- The execution time serves as a lower bound on the algorithm's time complexity.
- It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n0 such that c*g(n) ≤ f(n) for all n ≥ n0



f(n) = Omega(g(n))

Mathematical Representation of Omega notation:

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in the worst-case and sometimes in the average case.

Examples:

{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to ***Ω(n^2)***

U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to ***Ω(n)***

U { 100 , log (2000) , 10^4 } belongs to ***Ω(1)***

Note: Here, U represents union, we can write it in this manner because Ω provides exact or lower bounds.

Properties of Asymptotic Notations:

1. **General Properties:**

If **f(n)** is **O(g(n))** then **a*f(n)** is also **O(g(n))**, where **a** is a constant.

Example:

$f(n) = 2n^2+5$ is $O(n^2)$

then, $7*f(n) = 7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

We can say,

*If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)), where a is a constant.*

*If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)), where a is a constant.*

2. **Transitive Properties:**

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)).

Example:

If $f(n) = n$, $g(n) = n^2$ and $h(n)=n^3$

n is $O(n^2)$ and n^2 is $O(n^3)$ then, n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n))$ then $f(n) = \Theta(h(n))$.

If $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n))$ then $f(n) = \Omega(h(n))$

3. **Reflexive Properties:**

Reflexive properties are always easy to understand after transitive.

If f(n) is given then f(n) is O(f(n)). Since *MAXIMUM VALUE OF f(n) will be f(n) ITSELF!*

Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.

Example:

$f(n) = n^2$; $O(n^2)$ i.e $O(f(n))$

Similarly, this property satisfies both Θ and Ω notation.

We can say that,

If f(n) is given then f(n) is $\Theta(f(n))$.

If f(n) is given then f(n) is $\Omega(f(n))$.

4. **Symmetric Properties:**

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)).

Example:

If $f(n) = n^2$ and $g(n) = n^2$

then, $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

5. **Transpose Symmetric Properties:**

If f(n) is O(g(n)) then g(n) is Ω (f(n)).

Example:

If $f(n) = n$, $g(n) = n^2$

then n is $O(n^2)$ and n^2 is $\Omega(n)$

This property only satisfies O and Ω notations.

6. **Some More Properties:**

1. If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))
2. If f(n) = O(g(n)) and d(n)=O(e(n)) then f(n) + d(n) = O(max(g(n), e(n)))

Example:

f(n) = n i.e O(n)

$d(n) = n^2$ *i.e* $O(n^2)$

then $f(n) + d(n) = n + n^2$ *i.e* $O(n^2)$

3. If f(n) = O(g(n)) and d(n) = O(e(n)) then f(n)*d(n) = O(g(n)*e(n))

Example:

$f(n) = n$ *i.e* $O(n)$

$d(n) = n^2$ *i.e* $O(n^2)$

then $f(n) * d(n) = n * n^2 = n^3$ *i.e* $O(n^3)$

Note: *If* $f(n) = O(g(n))$ *then* $g(n) = \Omega(f(n))$

Q. What are the worst-case and the best-case time complexities of the following function? **[2023 Fall]**

```
int add(int A[], int n)
{
int sum = 0;
for(int i=0; i<n; i++)
sumsum + A[i];
return sum;
}
```

Solution:

To analyse the time complexity of the given function, let's break it down:

Initialization:

Initialising the sum takes constant time, denoted as O(1).

Loop:

The loop iterates over each element in the array A[] exactly once, for a total of 'n' iterations where 'n' is the size of the array.

Inside the loop, there is a constant-time operation of accessing A[i] and adding it to sum.

So, the loop contributes O(n) time complexity.

Return:

Returning the final sum takes constant time, O(1).

Putting it all together, the overall time complexity of the function is the sum of the complexities of its constituent parts:

Best-case time complexity: O(n)

The best-case occurs when the array is empty, but it still requires traversing through the loop once, resulting in a linear time complexity.

Worst-case time complexity: O(n)

The worst-case also occurs when the array is not empty and requires traversing through the loop once, again resulting in a linear time complexity.

So, both the best-case and worst-case time complexities are O(n), where 'n' is the size of the input array.

1. **Define Abstract Data type with an example. Explain which data structures are suitable for the following problems with proper reasons:**
   i. **Evaluating the arithmetic expression**
   ii. **Process scheduling by operating systems**
   iii. **Developing the social networks** **[2021 Fall]**

⇒ **Abstract Data Type (ADT):** An Abstract Data Type (ADT) is a type (or class) for objects whose behavior is defined by a set of operations and a set of values. The implementation details of these operations are hidden from the user. The ADT defines what operations are available and what they do, but not how they are implemented.

Example: Consider a Stack ADT, which allows the following operations:

- **push(x):** Add element x to the top of the stack.
- **pop():** Remove and return the element at the top of the stack.
- **peek():** Return the element at the top without removing it.
- **isEmpty():** Check if the stack is empty.

The underlying implementation of these operations could use an array or a linked list, but the user of the Stack ADT does not need to know this.

Data Structures for Specific Problems:

i. **Evaluating the arithmetic expression: Suitable Data Structure: Stack**

Reason: Stacks are suitable for evaluating arithmetic expressions because they allow easy management of operators and operands following the Last-In-First-Out (LIFO) principle, which is essential for handling nested operations and precedence in expressions.

ii. **Process scheduling by operating systems: Suitable Data Structure: Queue**

Reason: Queues are used in process scheduling because they follow the First-In-First-Out (FIFO) principle, ensuring fair and orderly management of processes where the first process to arrive is the first to be executed.

iii. **Developing the social networks: Suitable Data Structure: Graph**

Reason: Graphs are ideal for modeling social networks as they naturally represent entities (nodes) and their relationships

(edges). They allow efficient traversal and connectivity operations necessary for functionalities like finding friends or suggesting connections.

2. Why do you use Abstract Data Type (ADT)? Which data structure would you prefer to implement the following features? Justify with your reasons: [2023 Spring]

⇒ **Abstract Data Type (ADT):**

Abstract Data Types (ADTs) are used to provide a clear specification of the data and operations on the data, without concern for implementation details. They allow for modular programming, easy maintenance, and abstraction in software development.

i. **CPU Scheduling: Preferred Data Structure: Priority Queue**

Reason: CPU scheduling requires prioritizing tasks based on their importance or urgency. A priority queue allows efficient insertion and deletion of elements based on priority, ensuring that high-priority tasks are processed first.

ii. **History on a Browser: Preferred Data Structure: Stack**

Reason: Browser history uses a stack to enable the back functionality, following the Last-In-First-Out (LIFO) principle. The most recently visited page is the first to be accessed when navigating back.

iii. **File Management System: Preferred Data Structure: Tree (Hierarchical Tree)**

Reason: File management systems use hierarchical trees to represent directories and files, allowing efficient organization, traversal, and searching within the file system.

iv. **Display route on GPS: Preferred Data Structure: Graph**

Reason: GPS routing involves finding the shortest path between points on a map. A graph data structure efficiently represents the locations (nodes) and the paths (edges) between them, allowing algorithms like Dijkstra's to find optimal routes.

v. **Representation of Polynomial Equations: Preferred Data Structure: Linked List**

Reason: Polynomials can be efficiently represented using linked lists, where each node contains a term's coefficient and exponent. This allows dynamic storage and easy manipulation of terms.

3. **What are the differences between data types, data structures, and abstract data types (ADTs)? Explain their significance in programming with examples.** [2023 Fall]

⇒ **a. Definitions and Significance**

Data Types:

- **Definition:** Data types specify the kind of data that a variable can hold in a programming language, such as integers, floats, characters, and booleans.
- **Significance:** They define the operations that can be performed on data and how the data is stored.
- **Example:** In C++, int, float, char are primitive data types.

Data Structures:

- **Definition:** Data structures are ways of organizing and storing data so that it can be accessed and modified efficiently.
- **Significance:** They are crucial for designing efficient algorithms and managing data in software applications.
- **Example:** Arrays, linked lists, stacks, queues, trees, and graphs.

Abstract Data Types (ADTs):

- **Definition:** ADTs are mathematical models for data types where the data type is defined by its behavior (semantics) from the point of view of a user, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.
- **Significance:** They provide a clear specification of how data structures should behave without dictating their implementation.
- **Example:** A stack ADT could be implemented using an array or a linked list, but its operations (push, pop, peek) remain consistent.

4. **What is the role of data structures for algorithms? Create an ADT for Stack using C or C++ code.** [2022 Fall]

⇒ **a. Role of Data Structures for Algorithms**

- **Efficiency:** Proper data structures can significantly improve the efficiency of algorithms by optimizing time and space complexity.
- **Ease of Implementation:** Algorithms are easier to implement when using the appropriate data structures.

- **Scalability:** Efficient data structures ensure that algorithms can handle large amounts of data without significant performance degradation.

ADT for Stack in C++

```
#include <iostream>
#define MAX 1000
class Stack {
  int top;
public:
  int a[MAX];   // Maximum size of Stack
  Stack() { top = -1; }
  bool push(int x);
  int pop();
  int peek();
  bool isEmpty();
};

bool Stack::push(int x) {
  if (top >= (MAX - 1)) {
    std::cout << "Stack Overflow\n";
    return false;
  } else {
    a[++top] = x;
    std::cout << x << " pushed into stack\n";
    return true;
  }
}

int Stack::pop() {
  if (top < 0) {
    std::cout << "Stack Underflow\n";
    return 0;
  } else {
    int x = a[top--];
    return x;
  }
}
```

```
int Stack::peek() {
  if (top < 0) {
    std::cout << "Stack is Empty\n";
    return 0;
  } else {
    int x = a[top];
    return x;
  }
}

bool Stack::isEmpty() {
  return (top < 0);
}

// Test the Stack
int main() {
  Stack s;
  s.push(10);
  s.push(20);
  s.push(30);
  std::cout << s.pop() << " Popped from stack\n";
  return 0;
}
```

5. What are the data structures used in the following areas print jobs in a computer, network data model, hierarchical data structure? [2018 Fall]

⇒ print jobs in a computer=queue
network data model=graph
hierarchical data structure=tree