Unit 1

Introduction

- **1 Introduction to data structure (8)**
- 1.1 Linear & Non linear
- 1.2 Algorithm Basic Concepts
- 1.3 Time and Space complexity of algorithms, Big O Notation and theta
- notations
- 1.4 Definition, implementation and notation of Array
- 1.5 Basic operation such as addition, deletion

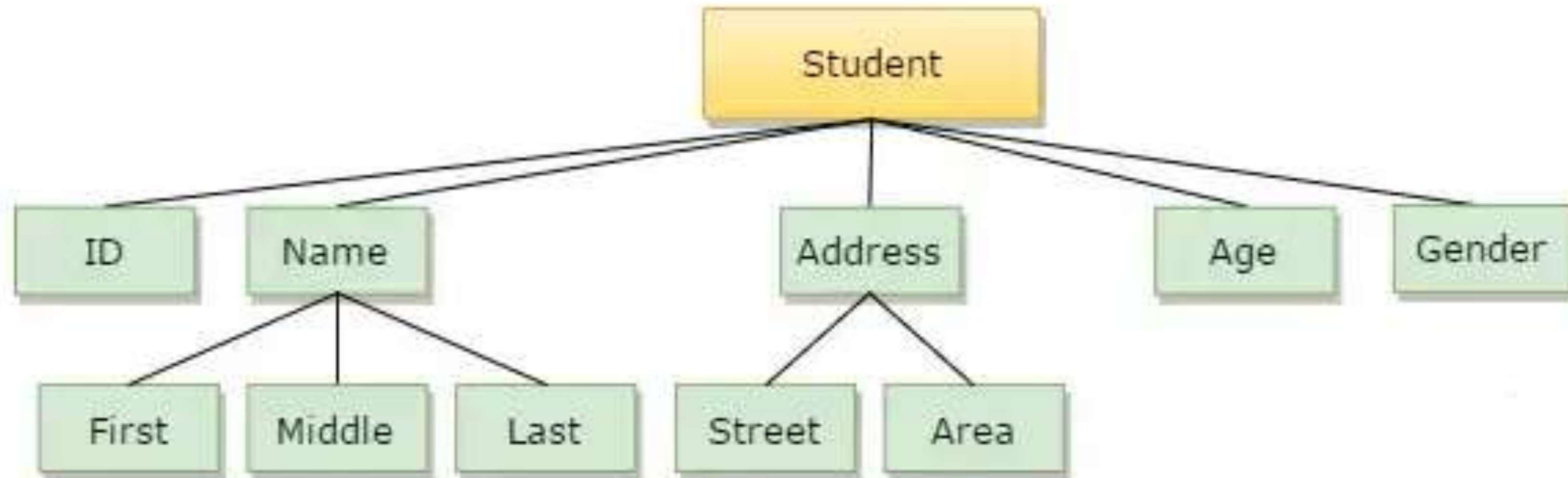
What is Data Structure?

In the context of computers, the data structure is a specific way of storing and organizing data in the computer's memory so that these data can be easily retrieved and efficiently used when needed later. The data can be managed in many different ways, such as a logical or mathematical model for a particular organization of data is called a data structure.

The variety of a specific data model depends on the two factors:

- First, it must be loaded enough into the structure to reflect the actual relationship of the data with a real-world object.
- Second, the formation should be so simple that one can efficiently process the data whenever necessary.

Let's take an example where a student's name can be broken down into three sub-items: first, middle, and last. But an ID assigned to a student will usually be considered a single item.



The example mentioned above, such as ID, Age, Gender, First, Middle, Last, Street, Area, etc., are elementary data items, whereas the Name and the Address are group data items.

Categories of Data Structure

Data structures can be subdivided into two major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure

A data structure is said to be linear if its elements combine to form any specific order. There are two techniques for representing such linear structure within memory.

- The first way is to provide a linear relationship between all the elements represented using a linear memory location. These linear structures are called arrays.
- The second technique provides a linear relationship between all the elements represented using the concept of pointers or links. These linear structures are called linked lists.

The typical examples of the linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non-linear Data Structure

This structure mainly represents data with a hierarchical relationship between different elements.

Examples of Non-Linear Data Structures are listed below:

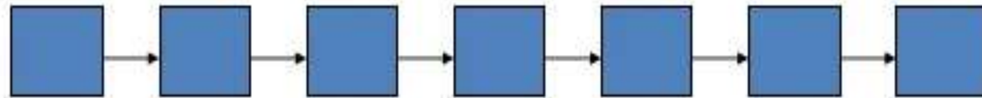
- Graphs
- Family of trees and
- Table of contents

Tree: In this case, the data often has a hierarchical relationship between the different elements. The data structure that represents this relationship is called a rooted tree graph or tree.

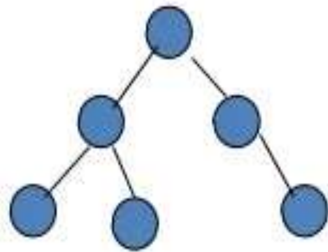
Graph: In this case, the data sometimes has relationships between pairs of elements, which do not necessarily follow a hierarchical structure. Such a data structure is called a graph.

Types of data structures

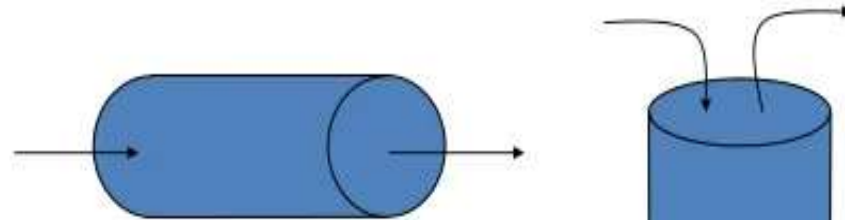
Array



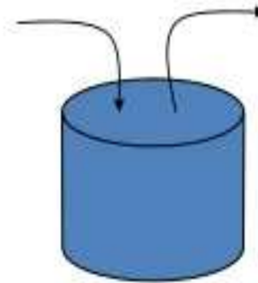
Linked List



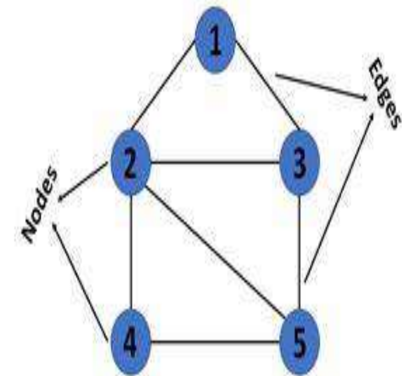
Tree



Queue



Stack



There are many, but we named a few. We'll learn these data structures in great detail!

	Linear Data structure	Non-linear data structure
1	The data elements are stored in a linear order in the case of linear data structure. Each and every element is connected to the first and the next element in the sequence.	The data elements in the case of a non-linear data structure are arranged in a non-linear way and attached hierarchically. The data elements are attached to multiple elements.
2	The structure of the data consists of a single level. There is no hierarchy in the linear data structure.	In this structure, there are multiple levels involved in the structure. Therefore the elements are arranged hierarchically.
3	The implementation of the linear structure of data is easy as the elements are stored in a linear way.	The implementation of the structure is a complex process compared to the linear structure.
4	Traversal of the elements in a linear data structure can be carried out in a single execution because the data is present in a single level	Traversal of the elements cannot be carried out in a single execution only. Multiple runs are required for traversing the data in a non-linear data structure.
5	There is no efficient utilization of memory in a linear data structure.	There is efficient utilization of memory in a non-linear data structure.
6	Examples of linear data structures include array, stack, queues, and linked list.	Examples of non-linear data include trees and graphs
7	The linear structure of data is applied mainly in software development.	The non-linear structure of data is mostly applied in Artificial intelligence and image processing.
8	With the increase in the size of the input, the time complexity increases.	Even if there is an increase in the size of the input, the time complexity remains the same.
9	Only one type of relationship might be present between the data elements	A one-to-one or one-to-many type of relationship can exist between the elements in a non-linear type of data structure.

Unit1

Algorithm

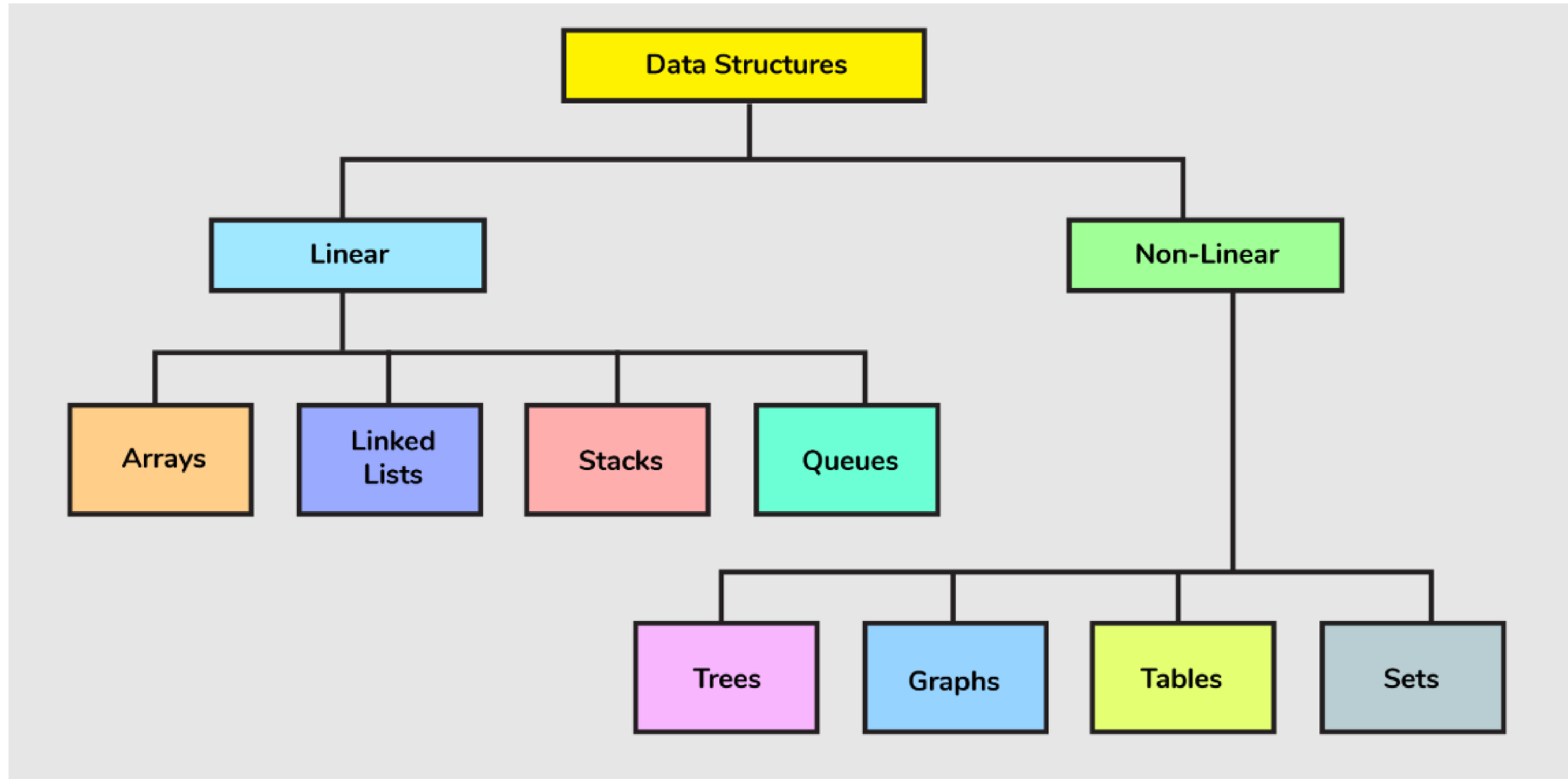
7. COURSE CONTENTS

UNIT NO.	Topics/Sub-Topics
I	**Introduction to data structure** 1.1 Linear & Non linear 1.2 Algorithm Basic Concepts 1.3 Introduction to Time and Space complexity of algorithms, Big O Notation and theta notations 1.4 Definition, implementation and notation of Array- Numerical and character 1.5 Basic operation such as addition, deletion , String operations

- What is Data Structure?
- A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

- A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Classification of Data Structure



- 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.
- Example: 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.
- Example: array.
- Dynamic Data Structure: In 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.
- Example: Queue, Stack, etc.

- 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: Trees and Graphs.

Algorithm

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

- From the data structure point of view, following are some important categories of algorithms −
- Search − Algorithm to search an item in a data structure.
- Sort − Algorithm to sort items in a certain order.
- Insert − Algorithm to insert item in a data structure.
- Update − Algorithm to update an existing item in a data structure.
- Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Why do we need Algorithms?

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Algorithm example

Example

Let's try to learn algorithm-writing by using an example.

Problem – Design an algorithm to add two numbers and display the result.

```
Step 1 – START
Step 2 – declare three integers a, b & c
Step 3 – define values of a & b
Step 4 – add values of a & b
Step 5 – store output of step 4 to c
Step 6 – print c
Step 7 – STOP
```

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

```
Step 1 – START ADD
Step 2 – get values of a & b
Step 3 – c ← a + b
Step 4 – display c
Step 5 – STOP
```

- Two common approaches used in designing algorithms are the bottom-up and top-down methods.
- Top-Down Approach
- The top-down approach, also known as “divide and conquer,” begins by viewing the problem in its entirety.
- It starts with the main problem and breaks it down into smaller, more manageable sub problems.
- This approach is generally associated with the use of recursion and is prevalent in design techniques like dynamic programming and divide-and-conquer algorithms.

- For example, in the case of a sorting problem like Merge Sort, a top-down approach would start by splitting the entire array into smaller pieces, sorting these smaller pieces, and then merging them back together to create a sorted array.

- The bottom-up approach, on the other hand, starts with the simplest and most basic subproblems and gradually builds up solutions to larger subproblems.
- This technique is widely used in dynamic programming, where solutions to subproblems are typically stored in a table for easy access and to avoid recomputation.

- Take, for instance, the Fibonacci sequence, where each number is the sum of the two preceding ones. A bottom-up algorithm would start by computing the smallest values (base cases) of the sequence

Example 1: Write an algorithm to find the maximum of all the elements present in the array.

Follow the algorithm approach as below:

Step 1: Start

Step 2: Declare a variable max with the value of the first element of the array.

Step 3: Compare max with other elements using loop.

Step 4: If max < array element value, change max to new max.

Step 5: If no element is left, return or print max otherwise goto step 3.

Step 6: End

Example 2: Write an algorithm to find the average of 3 subjects.

- Step 1: Start
- Step 2: Declare and Read 3 Subject, let's say S1, S2, S3
- Step 3: Calculate the sum of all the 3 Subject values and store result in Sum variable (Sum = S1+S2+S3)
- Step 4: Divide Sum by 3 and assign it to Average variable.
- (Average = Sum/3)
- Step 5: Print the value of Average of 3 Subjects
- Step 6: End

Unit 1

Algorithm Complexity

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
// Suppose we have to calculate the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum holds the sum of the n numbers
return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

- Space complexity: An algorithm's space complexity is the amount of space required to solve a problem and produce an output.
- Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

- To store program instructions
- To store constant values
- To store variable values
- To track the function calls, jumping statements, etc.

- Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space.
- The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

- So, Space complexity = Auxiliary space + Input size.

An algorithm's space complexity quantifies how much space or memory it takes to run as a function of the length of the input while an algorithm's time complexity measures how long it takes an algorithm to run as a function of the length of the input.

That's why it's so important to learn about the time and space complexity analysis of various algorithms.

1. O(1)

Where an algorithm's execution time is not based on the input size n, it is said to have constant time complexity with order O (1).

Whatever be the input size **n**, the runtime doesn't change. Here's an example:

- #include <stdio.h>
- int main()
- {
- printf("Hello World");
- return 0;
- }
- Output
- Hello World

- In the above code “Hello World” is printed only once on the screen.
- So, the time complexity is constant: O(1) i.e. every time a constant amount of time is required to execute code, no matter which operating system or which machine configurations you are using.

2. O(n)

When the running time of an algorithm increases linearly with the length of the input, it is assumed to have linear time complexity, i.e. when a function checks all of the values in an input data set (or needs to iterate once through every value in the input), it is said to have a Time complexity of order O (n). Consider the following scenario:

```
• #include <stdio.h>
• void main()
• {
•         int i, n = 8;
•         for (i = 1; i <= n; i++) {
•                 printf("Hello World !!!\n");
•         }
• }
```

- Output
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!

- In the above code “Hello World !!!” is printed only n times on the screen, as the value of n can change.
- So, the time complexity is linear: O(n) i.e. every time, a linear amount of time is required to execute code.

What does it mean to state best-case, worst-case and average time complexity of algorithms?

- Let's take the example of searching for an item sequentially within a list of unsorted items.
- If we're lucky, the item may occur at the start of the list. If we're unlucky, it may be the last item in the list.
- The former is called *best-case complexity* and the latter is called *worst-case complexity*.

- If the searched item is always the first one, then complexity is $O(1)$.
- if it's always the last one, then complexity is $O(n)$.
- We can also calculate the *average complexity*, which will turn out to be $O(n)$.
- The term "complexity" normally refers to worst-case complexity.

Mathematically, different notations are defined

- Worst-case or upper bound: Big-O: $O(n)$
- Best-case or lower bound: Big-Omega: $\Omega(n)$
- Average-case: Big-Theta: $\Theta(n)$

- 1. Big 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.

Omega 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.

Theta 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 analyzing the average-case complexity of an algorithm.
- The execution time serves as both a lower and upper bound on the algorithm’s time complexity.
- It exists as both, the most, and least boundaries for a given input value.

Unit 1

Time and Space complexity

- Analyzing an algorithm means determining the amount of resources (such as time and memory) needed to execute it.
- Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.
- The time complexity of an algorithm is basically the running time of a program as a function of the input size.
- Similarly, the space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.
- In other words, the number of machine instructions which a program executes is called its time complexity.
- This number is primarily dependent on the size of the program's input and the algorithm used.

- Generally, the space needed by a program depends on the following two parts:
- Fixed part: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- Variable part: It varies from program to program.
- It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.
- However, running time requirements are more critical than memory requirements.

Worst-case, Average-case, Best-case, and Amortized Time Complexity

Worst-case running time This denotes the behaviour of an algorithm with respect to the worst-possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time The term 'best-case performance' is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

Amortized running time Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

- Algorithm Efficiency
- If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm

Linear Loops

To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for(i=0;i<100;i++)
        statement block;
```

Here, 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

```
f(n) = n
```

```
for(i=0;i<100;i+=2)
        statement block;
```

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

```
f(n) = n/2
```

Logarithmic Loops

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, in logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)                for(i=1000;i>=1;i/=2)
    statement block;                        statement block;
```

Consider the first `for` loop in which the loop-controlling variable `i` is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of `i` doubles. Now, consider the second loop in which the loop-controlling variable `i` is divided by 2. In this case also, the loop will be executed 10 times. Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied. In the examples discussed, it is 2. That is, when `n = 1000`, the number of iterations can be given by `log 1000` which is approximately equal to 10.

Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as

```
f(n) = log n
```

- Nested Loops
- Loops that contain loops are known as nested loops.
- In order to analyze nested loops, we need to determine the number of iterations each loop completes.
- The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

- we analyze the efficiency of the algorithm based on whether it is a
- linear logarithmic, quadratic, or dependent quadratic nested loop.

quadratic, or dependent quadratic nested loop.

Linear logarithmic loop Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is `log 10`. This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according to the formula, the number of iterations for this code can be given as `10 log 10`.

```
for(i=0;i<10;i++)
    for(j=1; j<10;j*=2)
        statement block;
```

In more general terms, the efficiency of such loops can be given as $f(n) = n \log n$.

Quadratic loop In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
    for(j=0; j<10;j++)
        statement block;
```

The generalized formula for quadratic loop can be given as $f(n) = n^2$.

Dependent quadratic loop In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
    for(j=0; j<=i;j++)
        statement block;
```

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

$$1 + 2 + 3 + \cdots + 9 + 10 = 55$$

If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates $(n + 1)/2$ times. Therefore, the efficiency of such a code can be given as

$$f(n) = n\,(n + 1)/2$$

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as $O(1)$
- Linear time algorithm: running time complexity given as $O(n)$
- Logarithmic time algorithm: running time complexity given as $O(\log n)$
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$
- Exponential time algorithm: running time complexity given as $O(2^n)$

Table 2.2 shows the number of operations that would be performd for various values of n.

Table 2.2 Number of operations for different functions of n

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

- Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case.
- For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case.
- For example, when sorting an array the worst case is when the array is sorted in reverse order.

Questions

Multiple-choice Questions

1. Which data structure is defined as a collection of similar data elements?
   (a) Arrays (b) Linked lists
   (c) Trees (d) Graphs
2. The data structure used in hierarchical data model is
   (a) Array (b) Linked list
   (c) Tree (d) Graph
3. In a stack, insertion is done at
   (a) Top (b) Front
   (c) Rear (d) Mid
4. The position in a queue from which an element is deleted is called as
   (a) Top (b) Front
   (c) Rear (d) Mid
5. Which data structure has fixed size?
   (a) Arrays (b) Linked lists
   (c) Trees (d) Graphs
6. If `TOP = MAX-1`, then that the stack is
   (a) Empty (b) Full
   (c) Contains some data (d) None of these
7. Which among the following is a LIFO data structure?
   (a) Stacks (b) Linked lists
   (c) Queues (d) Graphs
8. Which data structure is used to represent complex relationships between the nodes?
   (a) Arrays (b) Linked lists
   (c) Trees (d) Graphs
9. Examples of linear data structures include
   (a) Arrays (b) Stacks
   (c) Queues (d) All of these
10. The running time complexity of a linear time algorithm is given as
    (a) $O(1)$ (b) $O(n)$
    (c) $O(n \log n)$ (d) $O(n^2)$

3. Define data structures. Give some examples.
4. In how many ways can you categorize data structures? Explain each of them.
5. Discuss the applications of data structures.
6. Write a short note on different operations that can be performed on data structures.
7. Compare a linked list with an array.
8. Write a short note on abstract data type.
9. Explain the different types of data structures. Also discuss their merits and demerits.
10. Define an algorithm. Explain its features with the help of suitable examples.
11. Explain and compare the approaches for designing an algorithm.
12. What is modularization? Give its advantages.
13. Write a brief note on trees as a data structure.
14. What do you understand by a graph?
15. Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.
16. What do you understand by time–space trade-off?
17. What do you understand by the efficiency of an algorithm?
18. How will you express the time complexity of a given algorithm?

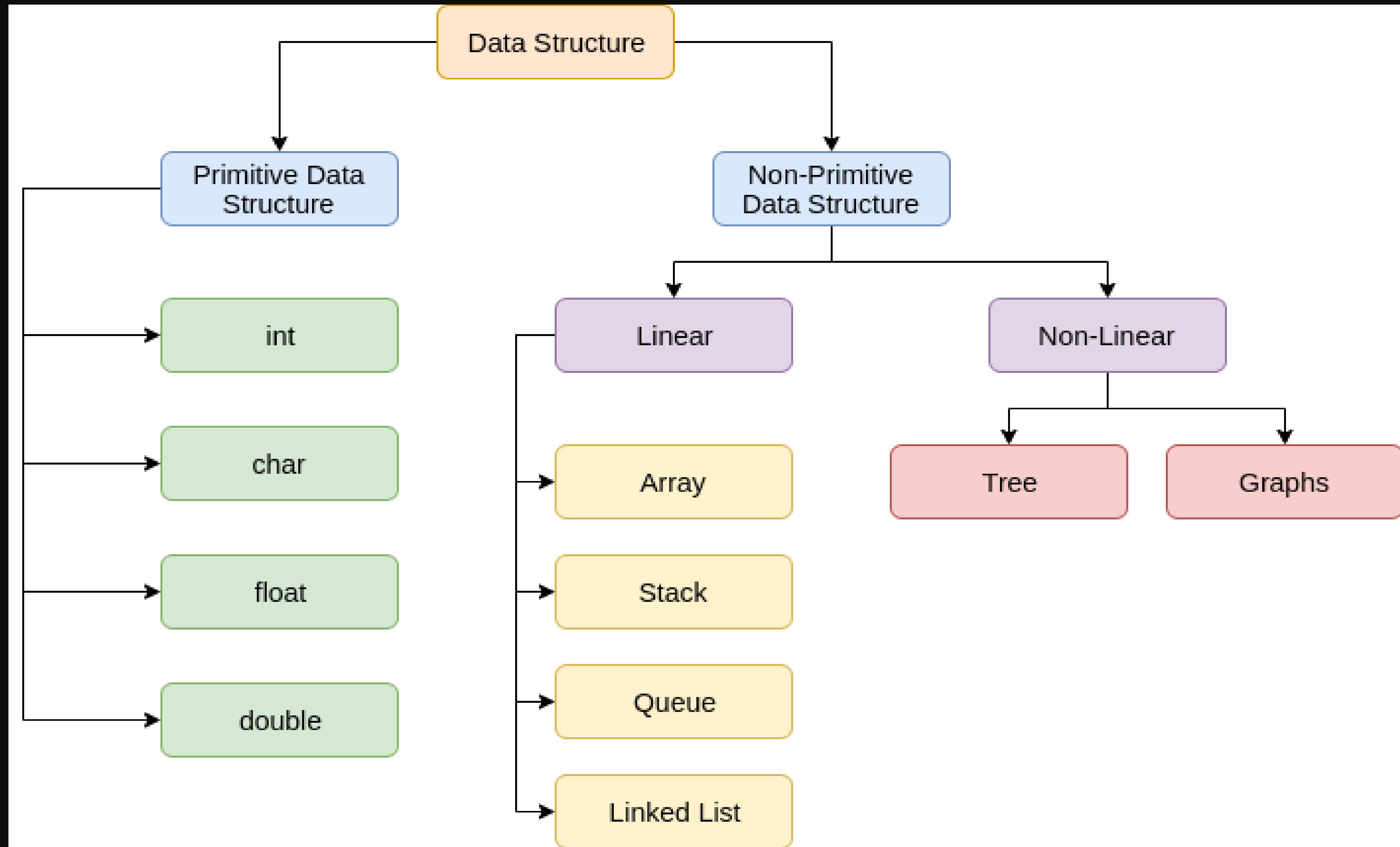
19. Discuss the significance and limitations of the Big O notation.
20. Discuss the best case, worst case, average case, and amortized time complexity of an algorithm.
21. Categorize algorithms based on their running time complexity.
22. Give examples of functions that are in Big O notation as well as functions that are not in Big O notation.
23. Explain the little o notation.
24. Give examples of functions that are in little o notation as well as functions that are not in little o notation.
25. Differentiate between Big O and little o notations.
26. Explain the Ω notation.
27. Give examples of functions that are in Ω notation as well as functions that are not in Ω notation.
28. Explain the Θ notation.
29. Give examples of functions that are in Θ notation as well as functions that are not in Θ notation.
30. Explain the ω notation.
31. Give examples of functions that are in ω notation as well as functions that are in ω notation.

Unit 1

Basics of array

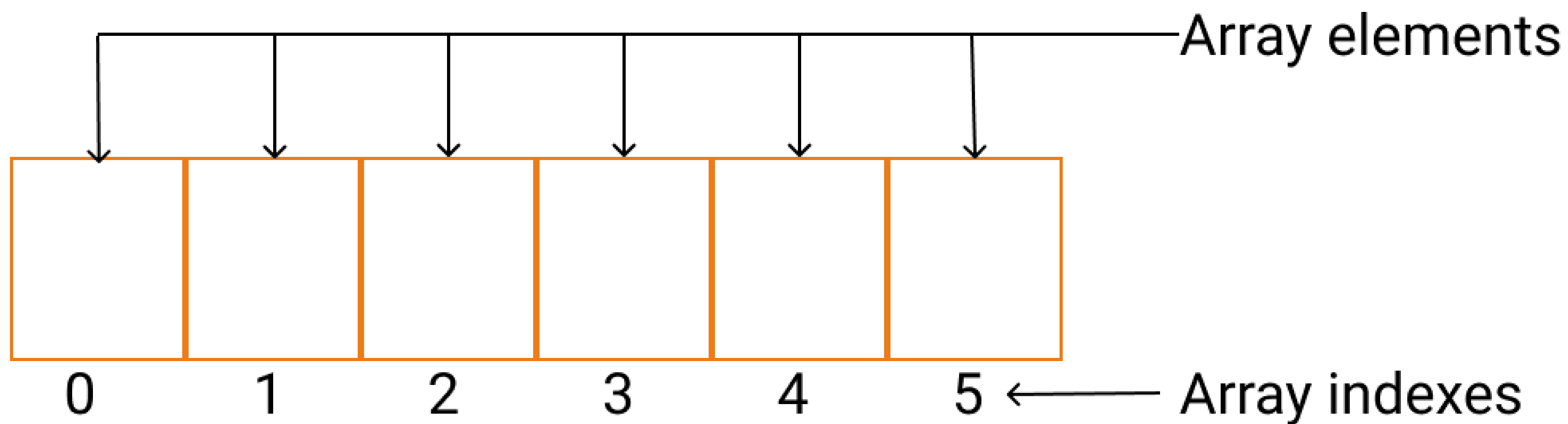
Array in Data Structure

- Arrays are defined as the collection of similar types of data items stored at contiguous memory locations.
- It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

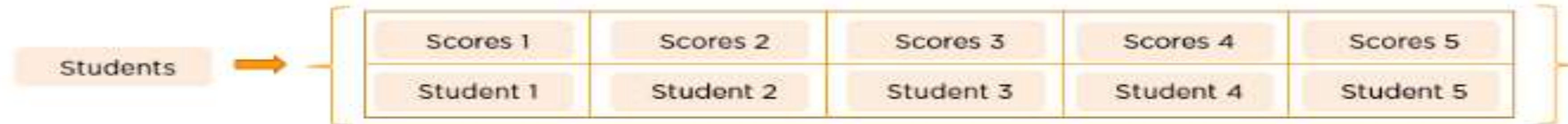


Data Structure
Primitive Data Structure
Non-Primitive Data Structure
int
char
float
double
Linear
Non-Linear
Array
Stack
Queue
Linked List
Tree
Graphs

- In C programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects.
- Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.


Array elements
0
1
2
3
4
5
Array indexes

Why Do You Need an Array in Data Structures?



Let's suppose a class consists of ten students, and the class has to publish their results. If you had declared all ten variables individually, it would be challenging to manipulate and maintain the data.

If more students were to join, it would become more difficult to declare all the variables and keep track of it. To overcome this problem, arrays came into the picture.

What Are the Types of Arrays?

There are majorly two types of arrays, they are:

One-Dimensional Arrays:

You can imagine a 1d array as a row, where elements are stored one after another.

Multi-Dimensional Arrays:

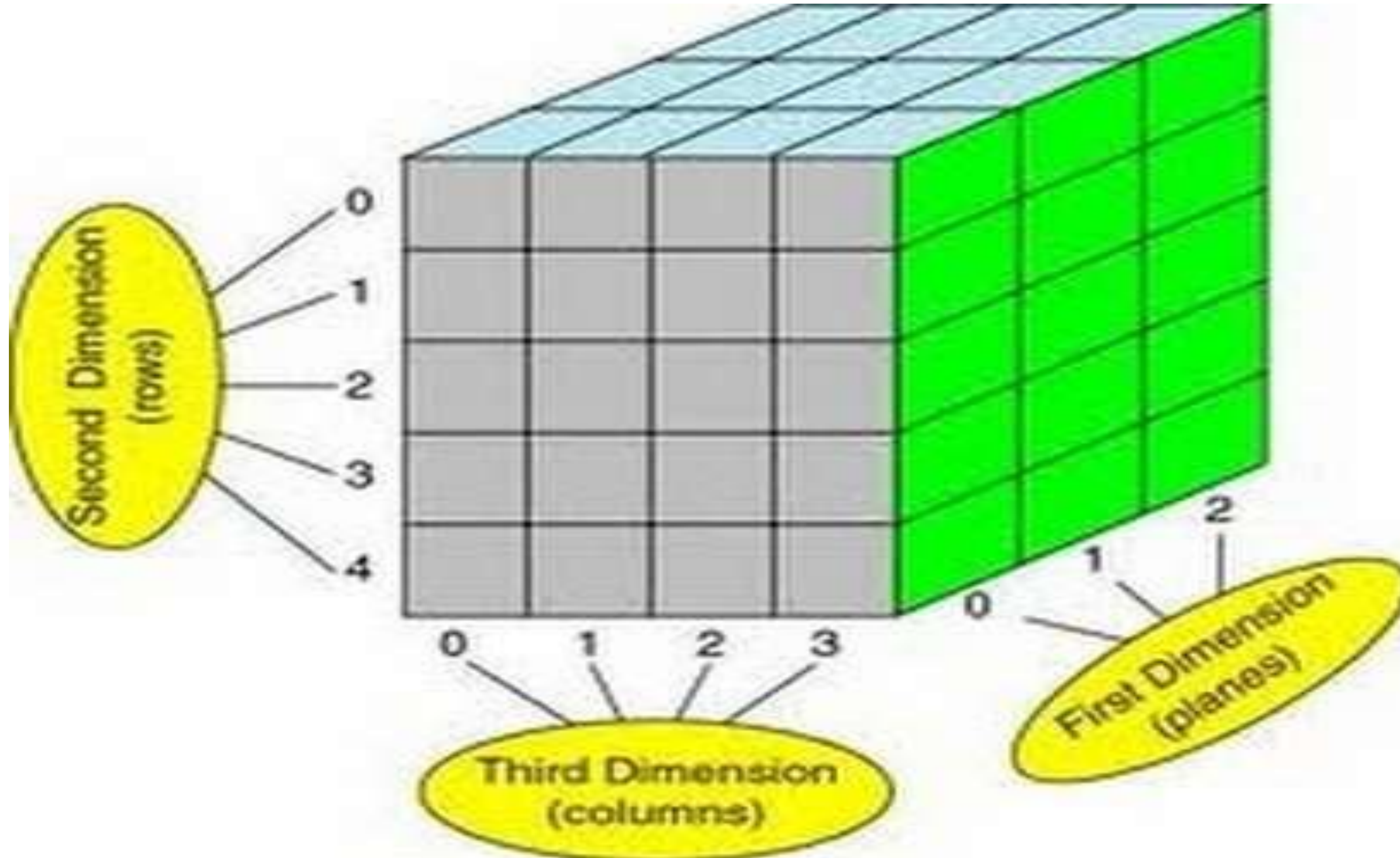
These multi-dimensional arrays are again of two types. They are:

Two-Dimensional Arrays:

Row ↓ / Col →	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

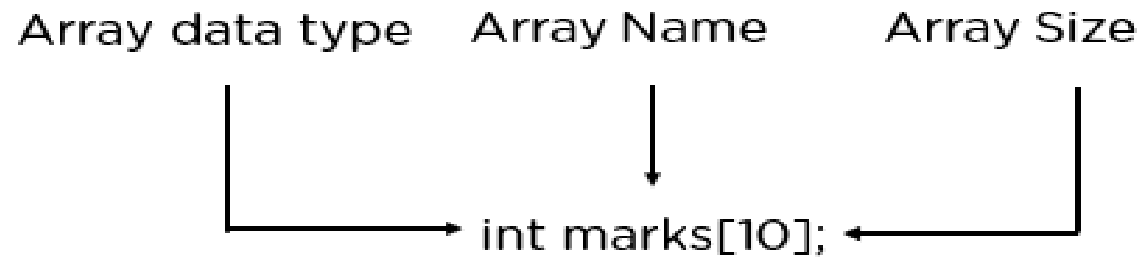
You can imagine it like a table where each cell contains elements.

3 dimensional array



OLAP Data Hypercube (No. of Dimensions = 3)

How Do You Declare an Array?



Arrays are typically defined with square brackets with the size of the arrays as its argument.

Here is the syntax for arrays:

1D Arrays: int arr[n];

2D Arrays: int arr[m][n];

3D Arrays: int arr[m][n][o];

How Do You Initialize an Array?

You can initialize an array in four different ways:

- ### Method 1:

```
int a[6] = {2, 3, 5, 7, 11, 13};
```

- ### Method 2:

```
int arr[]= {2, 3, 5, 7, 11};
```

- ### Method 3:

```
int n;

scanf("%d",&n);

int arr[n];

for(int i=0;i<5;i++)

{

scanf("%d",&arr[i]);

}
```

- ## Method 4:

```
int arr[5];

arr[0]=1;

arr[1]=2;

arr[2]=3;

arr[3]=4;

arr[4]=5;
```

How Can You Access Elements of Arrays in Data Structures?

5	10	25	30	50
0	1	2	3	4

You can access elements with the help of the index at which you stored them. Let's discuss it with a code:

```
#include<stdio.h>

int main()

{

int a[5] = {2, 3, 5, 7, 11};

printf(“%d\n”,a[0]); // we are accessing

printf(“%d\n”,a[1]);

printf(“%d\n”,a[2]);

printf(“%d\n”,a[3]);

printf(“%d”,a[4]);

return 0;

}
```

output

```
2
3
5
7
11
--------------------------------
Process exited after 0.1476 seconds with return value 0
Press any key to continue . . .
```