Introduction to Data Structure

Data Structures

- Data is just collection of facts or set of values that are in particular format.
- Information is the processed data.
- If the **data** is not organized effectively, it is very difficult to perform any task on **large** amount of **data.**
- A data structure is a particular way of organizing a large amount of data more efficiently in a computer so that any operation on that data becomes easy.
- Data structures is about rendering data elements in terms of some relationship, for better **performance**, **organization** and **storage**.

Types of Data Structures

Data structures are divided into two types:

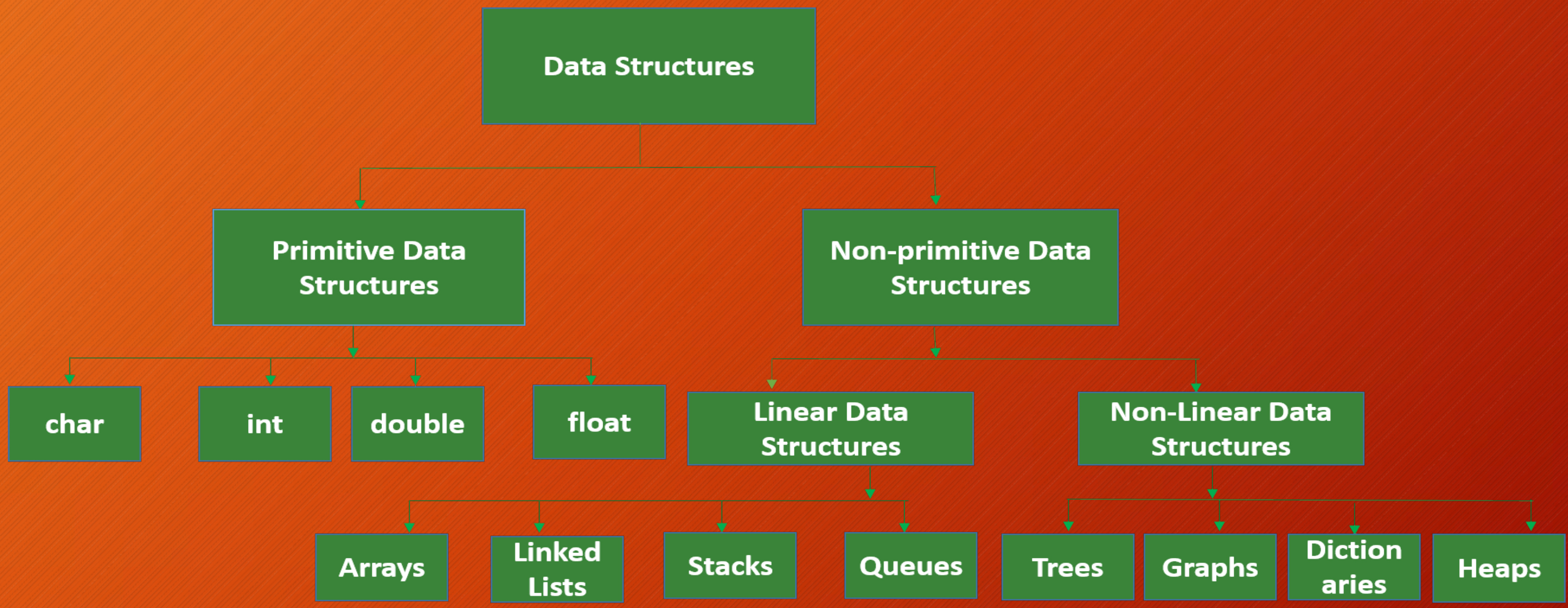
1. Primitive Data Structures (Built-In Data Structures)
   - **Primitive data structures** are char, int, double and float.

2.Non-primitive Data Structures (User-defined Data Structures)
   - **Non-primitive data structures** are used to store large and connected data. Some example of non-primitive data structures are: **Linked List**, **Tree**, **Graph**, **Stack** and **Queue**.

Types of Data Structures

- The **non-primitive data structures** are subcategorized into two ways: **Linear data structures** and **Non-linear data structures.**
- If a data structure is organizing the data in **sequential order** then that data structure is called a linear data structure.
  - Some of the examples are Arrays, Linked Lists, Stacks and Queues.
- If a data structure is organizing the data in **random order** or **hierarchical order**, not in sequential order, then that data structure is called as non-linear data structure.
  - Some of the examples are Trees, Graphs, Dictionaries and Heaps.

Types of Data Structures



Introduction to Algorithm

- An Algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- An **Algorithm** is independent of the programming language. An Algorithm is the core logic to solve a given problem.
- An **Algorithm** is expressed generally as **flow chart** or as an informal high level description called as **pseudocode Algorithm** can be defined as “a sequence of steps to be performed for getting the desired output for a given input.”
- There may be more than one way to solve a problem, hence there may be more than one **algorithms** for a given problem.

Characteristics of an Algorithm

1. Input: Algorithm should be accepting 0 or more inputs supplied externally.
2. Output: Algorithm should be generating at least one output.
3. Definiteness: Each step of an algorithm must be precisely defined. Meaning the step should perform a clearly defined task without much complication.
4. Finiteness: An algorithm must always terminate after a finite number of steps.
5. Effectiveness: The efficiency of the steps and the accuracy of the output determine the effectiveness of the algorithm.
6. Correctness: Each step of the algorithm must generate a correct output.

Efficiency of an Algorithm

- An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space.
- The performance of an algorithm is measured on the basis of following properties:
  - Time complexity
  - Space complexity

Time Complexity

- Time complexity is a way to represent the amount of time required by the program to run till its completion.
- The amount of time taken by the program depends on lot of things like hardware, operating system, number of processors, processor architecture etc.
- We do not consider all these factors when analysing the algorithms, we only consider the number of operations to be executed for the completion of the algorithm.
- The reason is very simple because we want the algorithm analysis to be system independent.
- **Time complexity** is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Time Complexity

- Let us consider the problem statement of the algorithm is "**Find the sum of first n natural numbers**".
- Algorithm - 1 :
- // Calculating the sum of first n natural numbers

```
int sum(int n)
 {
  int i, total = 0;
  for (i = 1; i <= n; i++)
   {
     total = total + i;
   }
 return total;
 }
```

Time Complexity

Code	No. of Elementary Operations	No of Times Executed	Total Operations
int i, total = 0;	1 (One Assignment Operation)	1	1 * 1 = 1
for (i = 1; i <= n; i++) { ... }	1 (One Assignment Operation)	1	1 * 1 = 1
for (i = 1; i <= n; i++) { ... }	1 (One Comparision Operation)	n + 1 (**n** times it evaluates to true, **1** time it evaluates to false)	1 * (n + 1) = n + 1
for (i = 1; i <= n; i++) { ... }	1 (One Increment Operation)	n times (**i** is incremented **n** times)	1 * n = n
for (i = 1; i <= n; i++) { total = total + i; }	2 (One Addition operation, One Assignment operation)	n (**n** times as loop executes **n** times)	2 * n = 2 * n
return total;	1 (One return operation)	1	1 * 1 = 1
Total Number of Operations			
Time complexity of the algorithm T(n) = 4 * n + 4 [Assuming each operation completes in unit time]			

Time Complexity

- The total number of elementary operation executed in the above algorithm is 4 * n + 4.
- As the value of n increases the time taken will also increase linearly. Thus this algorithm has a **linear time complexity.**
- Algorithm - 2 :
  // Calculating the sum of first n natural numbers using formula

```
int sum(int n)
{
 return n * (n + 1) / 2;
}
```

Time Complexity

Code	No. of Elementary Operations	No of Times Executed	Total Operations
return n * (n + 1) / 2	4 (One multiplication, One Addition, One division, One retrun operation)	1	4
Total Number of Operations			4
Time complexity of the algorithm T(n) = 4 [Assuming each operation completes in unit time]			

For the above code, the time complexity is **constant**, because it will never be dependent on the value of n, it will always give the result in 4 elementary operations.

Thus the time complexity of the **Algorithm - 2** is **constant**. In other words, the running time of the algorithm does not change with the input n.

In the above two simple algorithms, you saw how a single problem can have many solutions

Space Complexity

- Space complexity is the amount of memory space that is required by the algorithm for its execution.
- An algorithm generally requires space for following:
  - Instruction space: Space required to store the executable version (Also known as Binary Code) of the program. This space is fixed and depends on the number of instructions in the algorithm.
  - Data space: Space required to store all the program variables (Constants, Variables, Temporary Variables etc.).
  - Environment space: Space or Memory that is to be dynamically allocated. Memory that is required for storing data between functions
  - **One heap and one stack is maintained during the program execution.**
    - Heap is the segment where dynamic memory allocation usually takes place.
    - Stack is a segment where automatic variables and function call stack is stored.

Space Complexity

- When we are measuring the space complexity analysis of an algorithm, we consider only the **data space** of the algorithm and ignore the **instruction space** and **environmental space**
- For calculating the **space complexity**, we need to know the amount of memory used by variables of different data types, which generally varies for different operating systems.
- The method for calculating the **space complexity** remains the same and independent of the operating system.

Space Complexity

Data Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Calculating Constant Space Complexity

- Let us see the procedure for calculating the space complexity of a code with an example:

```
int sum(int x, int y)
{
 int z=x+y;
return z;
}
```

- The above code taken two inputs x and y of type int as formal parameters.
- In the code, another local variable z of type int is used for storing the sum of x and y.
- The int data type takes 2 **bytes** of memory, so the total space complexity is **3 (number of variables) * 2 (Size of each variable) = 6 bytes.**
- The space requirement of this algorithm is fixed for any input given to the algorithm, hence it is called as **constant space complexity**

Calculating Linear Space Complexity

- //Function to calculate a sum of n elements in the array //n is the number of elements in the array.

```
int sum(int a[ ] , int n)
{
   int x = 0, i = 0;
   for (i = 0; i < n; i++)
      {
         x= x + a[i];
      }
 return x;
 }
```

Calculating Linear Space Complexity

- In the above code, **2 * n** bytes of memory (size of int data type is **2**) is required by the array a[ ] and **2** bytes of memory for each variable of x, n and i.
- Hence the total space requirement for the above code would be **(2 * n + 6)**.
- The space complexity of the program is increasing linearly with the size of the array (input) n then it is called as **Linear Space Complexity**.
- Similarly, when the memory requirement of the algorithm increases quadratic to the given input then it is called as a **"Quadratic Space Complexity"**.
- Similarly, when the memory requirement of the algorithm increases cubic to the given input then it is called as a **"Cubic Space Complexity"** and so on.