Concept of Data Structure

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.

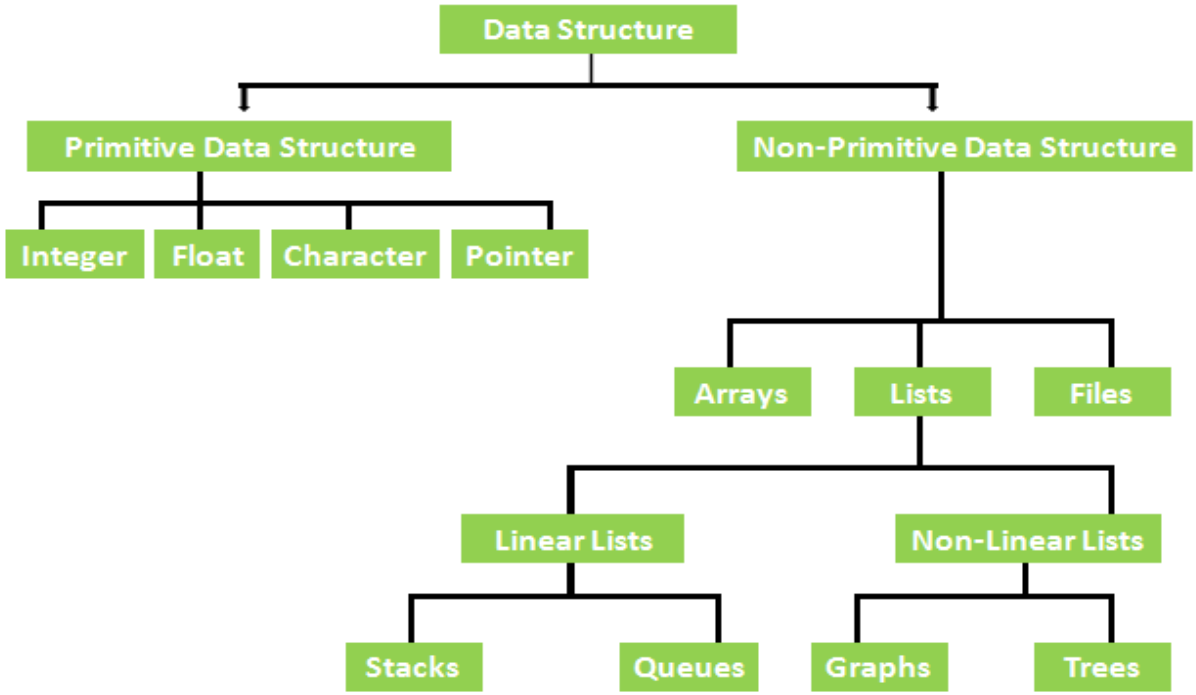
In computer science, a data structure is a data organization, management and storage format that enable efficient access and modification - Wikipedia

Data Structure is not a physical entity or data type (like integer, float) or actual memory, it is logical specifications or set of rules so that data can be stored in a efficient way

Why do we need Data Structure?

- Each Data Structure allows data to be stored in **specific manner**
- Data Structure allows **efficient** data **search** and **retrieval**
- **Specific** Data Structures are decided to work for **specific problems**
- It allows to **manage large amount** of data such as large database and **indexing** services such as **hash** table.

Classification of Data Structure:



Primitive data structure:- They are basic structure which are directly operated upon by the machine instruction *integer, floating, string, character* fall in this categories. These are represented in computer memory.

Integer - It represents some range of mathematical integers.

Float - Float stores double precision floating point number.

Character - The character represents a sequence of character data.

Pointer - Pointer holds the memory address of another variable.

Non-primitive data structure:- They are sophisticated data structure. These are derived from the primitive data structure. It emphasized a group of homogeneous or heterogeneous data items. Under non-primitive data structure *array, list, and files.*

- ## Linear Data Structure

  Linear data structures have all their elements arranged in a linear or sequential fashion. Each element in a linear data structure has a predecessor (previous element) and a successor (next element). Linear data structure further can divide into static and dynamic. Array is a linear static data type whereas Stack, Queue and Linked List are linear dynamic data type.

- ## Non-linear Data Structure

  In non-linear data structures, data is not arranged sequentially, instead, it is arranged in a non-linear fashion. Elements are connected to each other in a non-linear arrangement. Non-linear data structures are Trees and Graphs.

User-defined data type:

i) Enum:

enum (short for enumeration) is a data type that represents a set of named values. It is typically used to define a list of constants that have a specific meaning or purpose within a program. Enumerations provide a way to organize related values and improve code readability.

```
#include<stdio.h>
enum colour{red, blue, green, yellow};

int main()
{
        enum colour x;
```

```
        x= yellow;
        printf("Value = %d",x);
        return 0;
}
```

ii) Structure:

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

```
struct address {
   char name[50];
   char street[100];
   char city[50];
   char state[20];
   int pin;
};
```

iii) Union:

Union is a composite data type that allows two or more data types to be accessed via the same memory location.

```
union Data {
   int intValue;
   float floatValue;
   char stringValue[20];
};
```

iv) Typedef:

typedef is a keyword or construct that allows defining a new name for an existing data type. It is often used to create aliases or alternative names for existing types, which can enhance code readability and provide a level of abstraction.

```
    struct student
    {
        char name[20];
        int age;
    };
    typedef struct student stud;
stud s1, s2;
```

Operations on Data Structure:

The basic operations that are performed on data structures are as follows:

- **Insertion:** Insertion means addition of a new data element in a data structure.
- **Deletion:** Deletion means removal of a data element from a data structure if it is found.
- **Searching:** Searching involves searching for the specified data element in a data structure.
- **Traversal:** Traversal of a data structure means processing all the data elements present in it.
- **Sorting:** Arranging data elements of a data structure in a specified order is called sorting.
- **Merging:** Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

Data Structure Advantages

- **Efficient Memory use**: With efficient use of data structure memory usage can be optimized, for e.g we can use linked list vs arrays when we are not sure about the size of data. When there is no more use of memory, it can be released.
- **Reusability**: Data structures can be reused, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
- **Abstraction**: Data structure serves as the basis of abstract data types, the data structure defines the physical form of ADT(Abstract Data Type). ADT is theoretical and Data structure gives physical form to them.

Abstract Data Type

Abstraction is wrapping of functionality in a block and ignoring the internal part. It means to consider separated from the detail specification. It only focuses on the view and hide the information on how the data will be organized in the memory and which algorithm will be used to process the data.

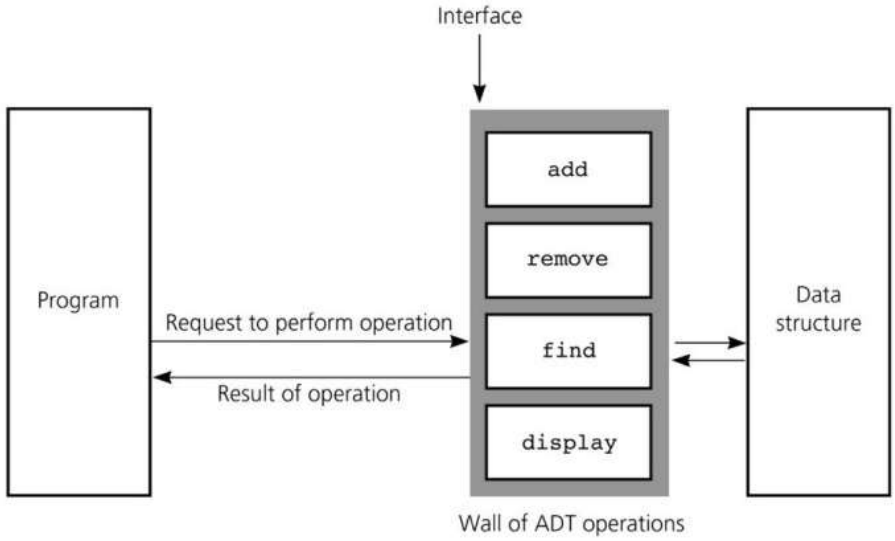
In the programming world, the code becomes so large that unlikely we will have to understand the code. So we divide the programming code into small chunk and makes the coding pattern easier to understand. It makes easier of the updating or new implementation in the code.

This is the most important concept of data science. Abstraction Data Types (ADT) implementation are not visible to the user. It has an inside and outside. The outside is called Interface and inside is called Implementation. The separation between the inside and outside is called Abstraction barrier.

Abstract data type is totally theoretical part. It specifies the description of abstract algorithm, evaluate the data structure and describe the type system of programming languages.

Example of ADT:

Integers are an abstract data type, which is used in addition, multiplication, division and other mathematical calculation. The computer displays the result, according to the given mathematical instructions without disclosing how the integers are evaluated by the computer.



How ADT Works

For more details:
https://www.youtube.com/watch?v=ZniDyolzrBw&t=22s

The areas in which data structure are applied:

- Compiler Design,
- Operating System,
- Database Management System,
- Statistical analysis and design,
- Numerical Analysis,
- Graphics,
- Artificial Intelligence,
- Simulation

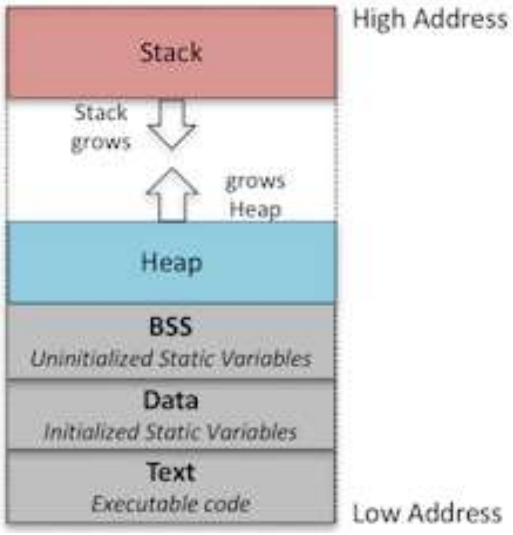
Dynamic Memory Allocation in C:

- An array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
- Static memory allocated during compile time and can not be increased and decreased during run time.

Example:

```
int main()
{
        int arr[5] = {1, 2, 3, 4, 5 };
}
```

- Sometimes the size of the array you declared may be insufficient.
- To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
- The process of allocating memory at the time of execution is called **dynamic memory allocation**.
- **Pointers** play very important role in dynamic memory allocation, without pointer allocated memory cannot be accessed.



- To allocate memory dynamically, library functions are:
    1. **malloc()**
    2. **calloc()**
    3. **realloc()**
    4. **free()**
- This functions are defined in the <stdlib.h>

malloc():

malloc is a built-in function declared in the header file <stdlib.h>

malloc is the short name for "memory allocation: and is used to dynamically allocate a single large block of contiguous memory according to the size specified.

Syntax:

ptr = (castType*) malloc(size);

Example:

ptr = (float*) malloc(100* sizeof(float));

callof():

calloc() function is used to dynamically allocate multiple blocks of memory.

calloc() stands for clear allocation.

Calloc() is difference in two ways:

- calloc() needs two arguments instead of just one.
  Syntax: void *calloc(size_t n, size_t size); // here n is number of blocks and size is size of each block

Example:

int *ptr = (int *)calloc(10, sizeof(int));

an equivalent malloc call:

int *ptr = (int *)malloc(10*sizeof(int));

- memory allocated by calloc is initialized to zero
  it is not the case with malloc. Memory allocated by malloc is initialized with some garbage value.

realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.
Syntax:

ptr = realloc(ptr, X);
Here, ptr is reallocated with a new size X.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You much explicitly use free() to release the space.

Syntax:

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

Algorithm:

An algorithm is a finite set of instructions, written in order to accomplish a certain predefined task.

Algorithm is not the complete code or program, it is just the core logic of a problem, which can be expressed either as an informal high-level description as pseudocode or using a flowchart.

Properties of algorithm:

- Input: There should be 0 or more inputs supplied externally to the algorithm.
- Output: There should be at least 1 output obtained.
- Definiteness: Every step of the algorithm should be clear and well defined.
- Finiteness: The algorithm should have finite number of steps.
- Correctness: Every step of the algorithm mush generate a correct output.
- Effectiveness: Each step must be carried out in finite time.

Types of Algorithms:

i) Deterministic and Non-deterministic Algorithm:

Deterministic and non-deterministic algorithms are two different types of algorithms based on the predictability of their execution and the output they produce.

Deterministic Algorithms:

- Deterministic algorithms are those in which the execution steps are precisely defined and will produce the same output for a given input every time they are run.
- These algorithms follow a well-defined sequence of steps, where each step is deterministic and has a unique outcome.

- Examples of deterministic algorithms include sorting algorithms like bubble sort, insertion sort, and merge sort, as well as mathematical algorithms like Euclid's algorithm for finding the greatest common divisor.

Non-deterministic Algorithms:

- Non-deterministic algorithms are those in which the execution steps are not precisely defined, and they may produce different outputs for the same input on different runs.
- Non-deterministic algorithms are often used in optimization problems, where they explore various possibilities to find an optimal solution.
- Examples of non-deterministic algorithms include randomized algorithms like quicksort with random pivot selection, genetic algorithms, and simulated annealing.

Non-deterministic algorithms can be used to solve problems that deterministic algorithms may struggle with or take a long time to solve. However, the non-deterministic nature of these algorithms means that they may not always produce the optimal solution, and their performance can vary across different runs.

ii) **Divide and Conquer Algorithm:**

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Broadly, we can understand **divide-and-conquer** approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples

The following computer algorithms are based on **divide-and-conquer** programming approach

- Merge Sort
- Quick Sort

iii) Sequential and Parallel Algorithm:

An **algorithm** is a sequence of instructions followed to solve a problem. While designing an algorithm, we should consider the architecture of computer on which the algorithm will be executed. As per the architecture, there are two types of computers −

- Sequential Computer
- Parallel Computer

Depending on the architecture of computers, we have two types of algorithms −

- **Sequential Algorithm** − An algorithm in which some consecutive steps of instructions are executed in a chronological order to solve a problem.
- **Parallel Algorithm** − The problem is divided into sub-problems and are executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output. A **parallel algorithm** is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

iv) Heuristic Algorithm:

A heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive.

It's important to note that while heuristic algorithms can provide efficient and practical solutions, they do not guarantee optimality and may not always produce the best possible solution for a given problem.

Algorithm Complexity

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

1. Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

2. Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

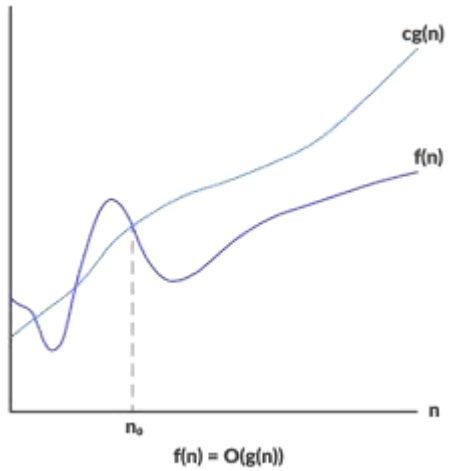
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



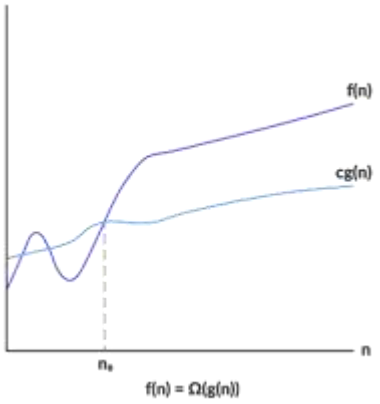
Big-O gives the upper bound of a function

O(g(n)) = { f(n): there exist positive constants C and N0 such that 0<=f(n) <= C*G(n) for all N>= N0}

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)). Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

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.



Omega gives the lower bound of a function

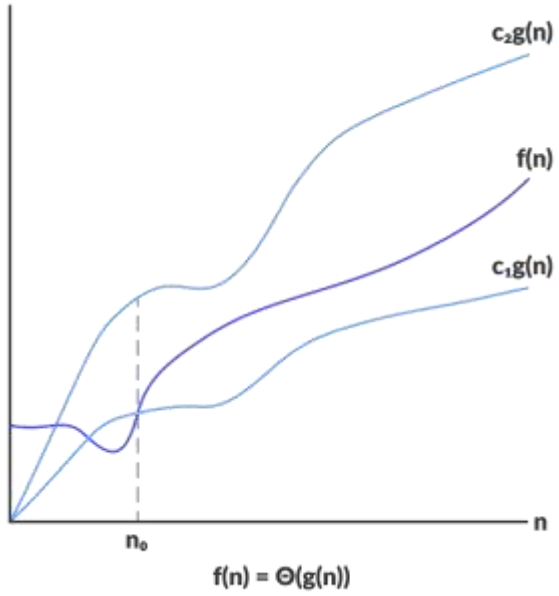
Ω (g(n)) = { f(n): there exist positive constants C and N0 such that 0<=C*g(n) <= f(n) for all N>= N0}

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

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 analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

Θ (g(n)) = { f(n): there exist positive constants C1, C2 and N0 such that 0<=C1*g(n) <= f(n) <= C2*g(n) for all N>= N0}

For a function g(n), Θ(g(n)) is given by the relation:

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

Big O Notation

- It looks for upper bound in the worst case in the given expression
- It always ignores the lower value
- It always ignores the constant value as well

Examples:

$f(n) = 3n^2 + 5n$ -----→ $O(n^2)$

$f(n) = n + 100 \log n$ ----→ $O(n)$

$f(n) = 3n^3 + 4n^5$ ------→ $O(n^5)$

$f(n) = 1000$ -------→ $O(1)$

Logarithms

In short form we called log,

It is nothing but inverse operation of Exponents.

Example - 1

a. $2^x = 8$
   Here 2 * 2 * 2 = 8 , so 3 times of two exponent equals to 8, the value of x = 3
   $2^3 = 8$
   In above expression,
   2 is Base
   3 is Exponent, and
   8 is Number

   Is equals to
b. $Log_2 8 = 3$
   Here, 2 is base, 8 is number and 3 is exponent
   Log of the number to the base is equals to exponent

Example – 2

a. $3^x = 81$

Here 3 * 3 * 3 * 3 = 81, so value of x = 4

$3^4 = 81$

Here, 3 is base, 4 is exponent and 81 is number

$\text{Log}_3 81 = 4$

So log of the number to the base is equals to exponent

Note:

Exponent of base is Number is always equals to log of the number of base is exponent i.e.

$a^{\text{Exponent}} = N$ is equals to

$\log_a N = \text{Exponent}$

Q1: What is logarithm of 8 to the base 2 ?

$\text{Log}_2 8 = \text{Exponent}$

So 2 power exponent 3 equals to 8.