# Unit 1: Introduction to Data Structures & Algorithms

# Syllabus

| **Unit 1** | **Introduction to Data Structures & Algorithms** | **Teaching Hours (4)** |
|---|---|---|
| Data types, Data structure and Abstract date type | Concept of data type. Basic and user defined data types. Concept of data structure and its uses. Definition and use of ADT. Benefits of using ADTs. | 1 hr |
| Dynamic memory allocation in C | Concept of dynamic memory allocation. | 1 hr |
| Introduction to Algorithms | Definition of algorithm. What is a good algorithm? Different structures used in algorithms. | 1 hr |
| Asymptotic notations and common functions | Time and space complexity. Big Oh (O) notation. | 1 hr |

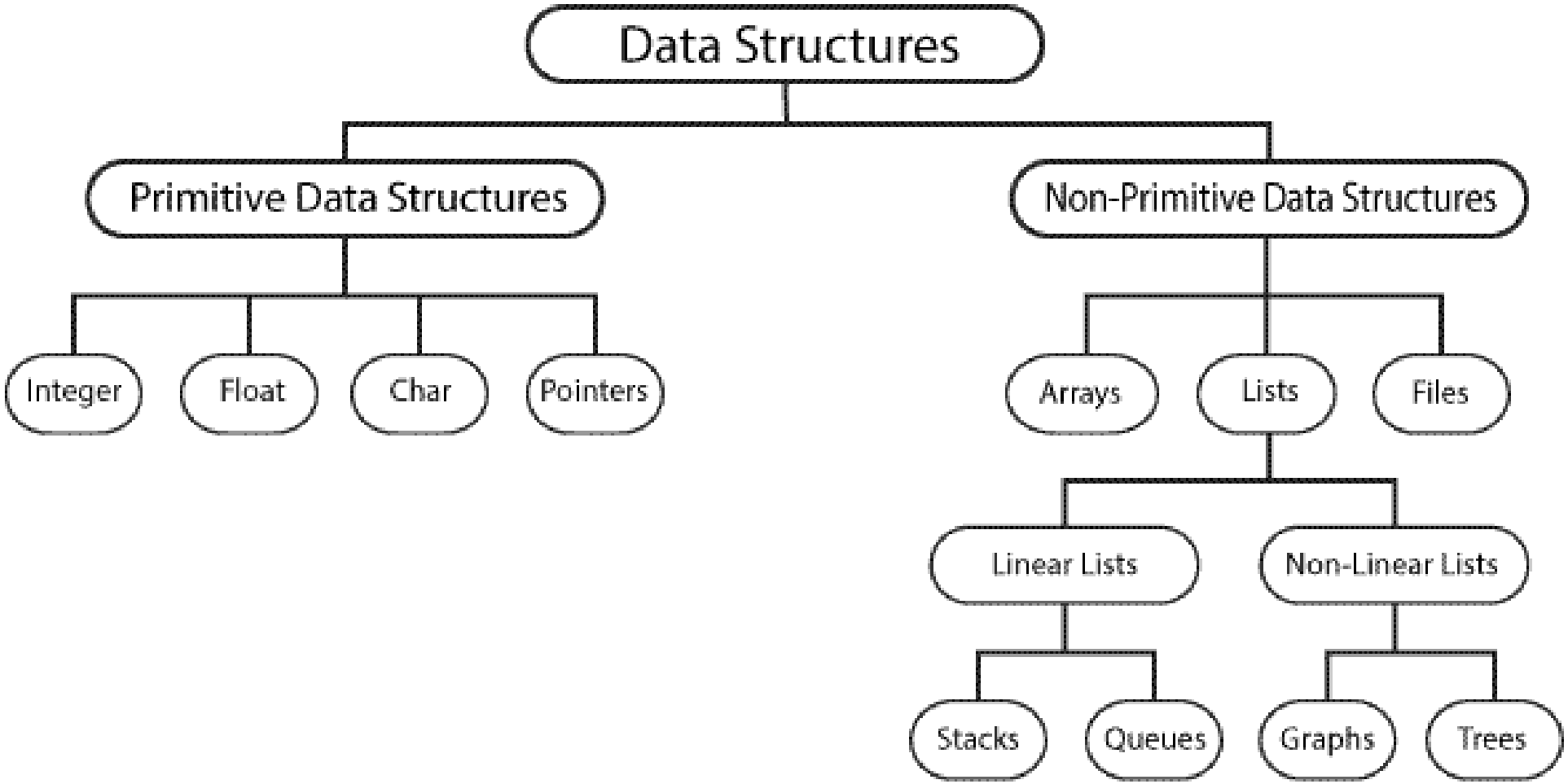
# Data Structure

1. Organization of data
2. Accessing methods
3. Degree of Associativity
4. Processing alternatives for information

Program = Algorithm + Data Structure

Data Structure:

- Pimitives: int, float, char, pointers
- non Primitives:
  - Linear data structure: Array, stack, Queue, List etc
  - Non Linear Data Structure: Tree, Graph




# Types Of Data Structure

# Abstract Data Type(ADT):

Consists of **data types** together with a **set of operation** which define how the type  may be manipulated. This specification is stated in an implementation independent manner.

Is a specification of only the behavior of instances of that type.

**Advantages:**

1. **Modularity**
2. **Precise Specifications**
3. **Information hiding**
4. **Simplicity**
5. **Integrity**
6. **Implementation Independence**

| malloc () | calloc () |
| --- | --- |
| Malloc function allocates single block of memory. | Calloc function allocates multiple block of memory |
| Int *pointer;<br>Pointer = malloc(10*sizeof (int));<br>10*4 = 40bytes of memory is allocated in one block.<br>Total allocated memory is 40bytes. | Int *pointer;<br>Pointer = calloc(10,10*sizeof(int));<br>10 blocks are created and 10*4 = 40 byte of memory is allocated to each block.<br>Total allocated memory is 10*10*4 = 400bytes. |
| It contains garbage value when malloc () function is used to initialize allocated memory. | It contains zero value when calloc () function is used to initialize allocated memory. |
| Type-cast must be done, since this function returns void pointer<br>Int *pointer;<br>(int*)malloc(10*sizeof(int)); | Type-cast must be done, since this function returns void pointer<br>Int *pointer;<br>(int*)calloc(10,10*sizeof(int)); |

# The Function realloc( )

- The function realloc( ) modifies the amount of memory previously reserved by a call to either malloc or calloc.

**realloc** (*pointer*, *number_of_bytes*);

  - The amount of memory reserved is equal to *number_of_bytes*.
  - The location of the memory reserved is indicated by *pointer*.
  - The argument pointer must have been returned by a previous call to either calloc or malloc.
    - realloc(bb[1], yy);
  - Reserve different block of memory
    - bb[1] = (char *) realloc(bb[1], yy);

# Concept and Definition: Algorithm

- An algorithm is a precise specification of a sequence of instructions to be carried out in order to solve a given problem.

- Each instruction tells what task is to be done.
-
- There should be a finite number of instructions in an algorithm and

- Each instruction should be executed in a finite amount of time.

# Characteristics of Algorithms

- Input:

A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are inputs which are processed by the algorithm.

- Definiteness:

Each step must be clear and unambiguous that leads to a specific action.

- Effectiveness:

Each step must be carried out in finite time.

- Finiteness:

Algorithm must terminate after finite time or steps

- Output:

An algorithm must have one or more output.

- Correctness:

Correct set of output values must be produced from the each set of inputs. For the same input data, it must always produce the same output.

# Writing Algorithm

- An algorithm can be written in a number of ways such as natural language, pseudo-code, or flowcharts. Pseudo-code is a popular way to express algorithm.
- Pseudo-code is a way of expressing algorithms that uses a mixture of *English phrases* and *indention* to make the steps in the solution explicit
- Pseudo-code cannot be compiled nor executed, and there are no real formatting or syntax rules.
- The benefit of pseudo-code is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language and easy to convert it into programs in any language.
- There is not any standard rules for writing algorithm.

Example:

**Write an algorithm to find the largest among three different numbers entered by user.**

Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Read variables a, b and c.
Step 4: If a>b
    If a>c
        Display a is the largest number.
    Else
        Display c is the largest number.
    Else If b>c
        Display b is the largest number.
    Else
        Display c is the greatest number.
Step 5: Stop

# Time Complexity

- *Time complexity* is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- That is, it is a function that refers the rate at which consumed time increases as input size increases.
- To simplify the analysis, the time for the operations that is constant independent of input size is ignored. Simplified analysis can be based on:
  - Number of arithmetic operation performed
  - Number of comparisons made
  - Number of times through a loop
  - Number of array elements are accessed etc.
- Although the running time of an implementation of the algorithm would depend upon the speed of the computer, programming language, compiler etc, these technical issues are ignored. Only how running time is increasing with increasing input size is considered in analysis.
- When analyzing algorithm it depends upon the input data, there are three cases:
  - Best case:
  - Average case:
  - Worst case:
  - **best**, **worst,** and **average cases** of a given algorithm express what the resource usage is *at least*, *at most* and *on average*, respectively. Usually the resource being considered is running time, i.e. time complexity, but it could also be memory or other resources.

## Let us consider the following implementation of Linear Search.

```
#include <stdio.h>

// Linearly search x in arr[].
// If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
       if (arr[i] == x)
         return i;
    }
    return -1;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

### For Linear Search

- The worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one.

- The best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n).

# Space Complexity

Analysis of space complexity of an algorithm is the amount of memory it needs to run to completion.
It is the rate at which required storage space grows as a function of input size.

The space needed by a program consists of following components:

- Instruction Space: space for code
- Data Space: space for variables, constants
- Environment stack space: return address, local variables in called function.

Example:

```
Algorithm Sum(a,n)
{
        s:=0.0;
        for i:=1 to n do
                s:=s+a[i];
         return s;
}
```

The space needed by n is one word, since it is of type integer. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, $S_{sum(n)} >= (n+3)$, n for a[], 1 for each n, I and s).

# Asymptotic Analysis

- The exact time of an algorithm will depend on the implementation of algorithm, amount input data, programming language, CPU speed etc.
- There is a proportionality approach to measure the time complexity in terms of relationship between input size and number of key operations of computational process. This type of analysis is known as asymptotic analysis.
- For asymptotic analysis we use different notations:

## Big Oh Notation

- Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements by eliminating constant factors.
- Big Oh notation is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.
- With big O notation we express the runtime in terms of—*how quickly it grows relative to the input, as the input gets arbitrarily large*.
- It's how we compare the efficiency of different algorithms to a problem.
- Properties of Big O:
  - Measuring the amount of work as it varies with the size of the data affected
  - Predicting the expected overall performance of an algorithm
  - Generally measured relative to other algorithms to handle the same problem

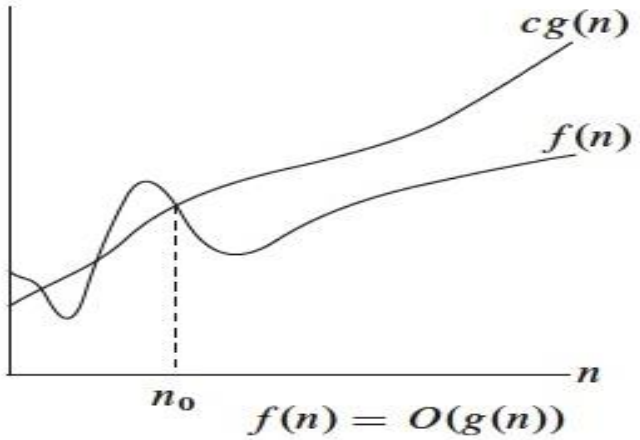
- Big O has following limitations:
  - It ignores potential of constant factors in the algorithm.
  - It does not try to improve algorithm, only gives the complexity.

## Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation.

*Formal definition*

*Let f and g be any two functions defined over set of positive integers. A function f(n) is big oh of g(n) ), denoted as f(n)=O(g(n)), if there exists two positive constants c and n0 such that for all n >= n0, f(n) <= c*g(n).*




- O(1) is used to denote constants.
- Example:1

  f(n)=$5n^3+3n^2+4$ and g(n) =$12n^3$ Then, is f(n) = $O(n^3)$?

| n | F(n) | g(n) | |
|---|---|---|---|
| 1 | 12 | 12 | True |
| 2 | 56 | 96 | True |
| 3 | 166 | 324 | True |
| 4 | 372 | 768 | True |

Therefore, for all n >=n0=1, c = 12, f(n) <= c*g(n) => f(n) = O(g(n)).

- Example: 2 If f(x)= $5x^2+4x + 2$ find big oh(O) of f(x).

  Here, f(x)= $5x^2+4x + 3 <= 5x^2+4x^2+ 2x^2 = 11x^2$ , for all x>=1

  f(x) <= c * g(x), where c =11, g(x) = $x^2$ and x0 = 1.

  Therefore, f(x) = $O(x^2)$.

# Big Omega ( Ω ) notation

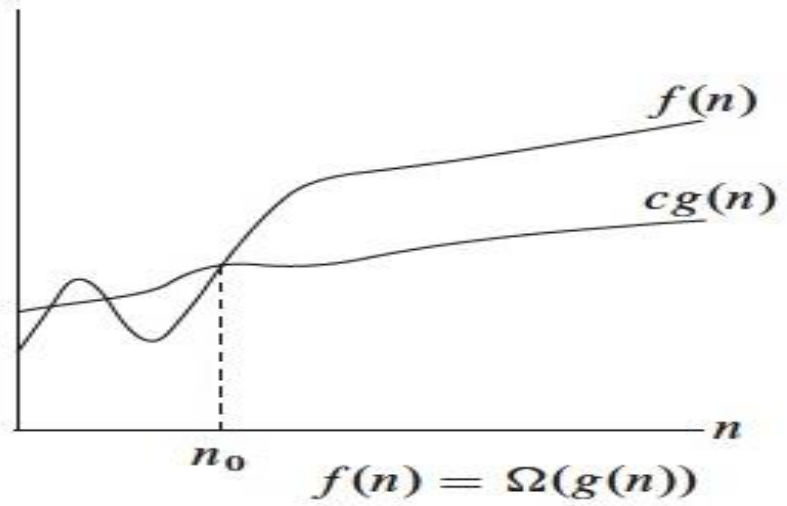
Big omega notation gives asymptotic lower bound.

*Formal definition*

*Let f and g be any two functions defined over set of positive integers. A function f(n) is big Omega of g(n) ), denoted as f(n)= Ω (g(n)), if there exists two positive constants c and n0 such that for all n >= n0, f(n) >= c*g(n).*

*The above relation says that g(x) is a lower bound of f(x).*

Example:
If f(n)= $3n^2$ +4n + 7, find big omega of f(n).
Here,
f(n)= $3n^2$ +4n + 7 >= $3n^2$ for all n>= 1.
i.e. f(n) >= c * g(n), where c = 3, n0=1
and g(n) =$n^2$ .
Therefore, f(n) = Ω (g(n).



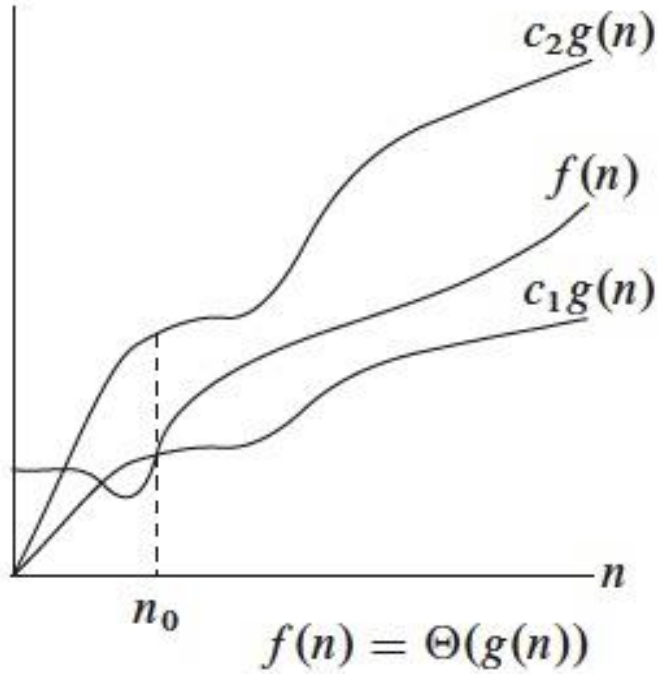
# Big Theta ( Θ ) notation

When we need asymptotically tight bound then we use notation.

*Formal definition*

*Let f and g be any two functions defined over set of positive integers. A function f(n) is big Theta of g(n) ), denoted as f(n)= Θ (g(n)), if there exists three positive constants c1, c2 and n0 such that for all n >= n0, c1*g(n) <= f(n) <= c2*g(n).*

The above relation says that f(x) is order of g(x)




Example: f(n) = $3n^2 + 4n + 7$ g(n) = $n^2$ , then prove that f(n) = (g(n)).

Proof:

let us choose c1, c2 and n0 values as 3, 14 and 1 respectively then we can have,

f(n) <= c2*g(n), n>=n0 as $3n^2 + 4n + 7 <= 14*n^2$ , and

f(n) >= c1*g(n), n>=n0 as $3n^2 + 4n + 7 >= 3*n^2$

for all n >= 1(in both cases).

So c1*g(n) <= f(n) <= c2*g(n) is trivial.

Hence f(n) = Θ(g(n)).