Chapter 1

Introduction to Data Structures

TODAY'S TOPIC

- Data Structure
- Need of Data Structure
- Different Types of Data Structure
- Representation of Data Structure

VARIABLES

- Lets relate to old mathematical equations:

$$x^2 + 2y - 2 = 1$$

- Equation has *names (x and y)* which holds values (data)
- The *names (x and y) are placeholders for representing data.*
- *Similarly in programming we need something for holding data and variables is the way to do that.*

DATA TYPES

- A data type is a collection of values and a set of operations on those values
- Most programming languages have data types such as *integers, floating point numbers, characters etc.*
- All these data types define set of values and operations on those values

DATA TYPES

- For example, the ***int*** *data type in C program* takes only integer values and we can apply addition, subtraction, multiplication, modulus etc. operations with them
- The ***float*** *data type in C* takes floating point values and addition, subtraction, multiplication operations are allowed whereas modulus operation is not allowed.

DATA STRUCTURES

- Once we have in variables , we need some mechanism for manipulating that data to solve problems.
- *Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.*
- A data structure is a special format for organizing and storing data.
- General data structure types *include arrays, files, linked lists, stacks, queues, trees, graphs and so on.*

DATA STRUCTURES

- A data structure is a *technique of organizing and storing of different types of data items in computer memory.*
- It is considered as not only the storing of data elements but also the maintaining of the logical relationship existing between individual data elements.
- A data structure *is the portion of memory allotted for a model, in which the required data can be arranged in proper fashion.*
- To develop a program of an algorithm, appropriate data structure for that algorithm.

Algorithm + Data Structure = Program

NEED OF DATA STRUCTURES

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

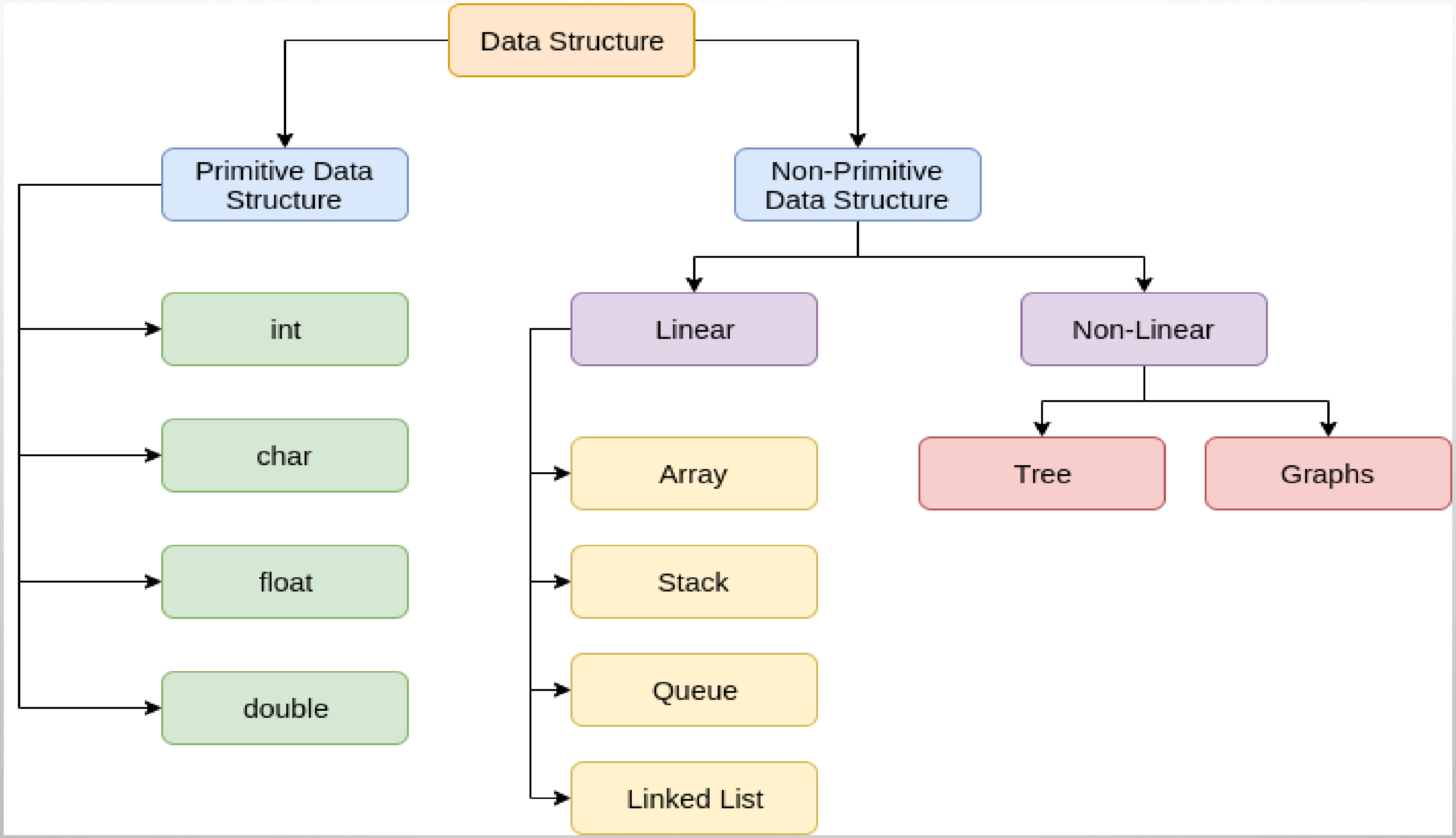
- *Processor speed:* To handle very large amount of data , high speed processing is required, but as growing data, processor may fail to deal with that much amount of data.
- *Data search:* Consider an inventory size of 500 items in a store, for searching an item, it needs to traverse 500 items every time, results in slowing down the search process.
- *Multiple requests:* If thousands of users are searching the data simultaneously, a very large server can also fail.

NEED OF DATA STRUCTURES

- In order to solve the mentioned problems, *data structures are used.*
- Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be accessed efficiently.

DATA STRUCTURES

Classification:



PRIMITIVE DATA STRUCTURE

- These are the basic data structures.
- The data structure that are directly operated upon by machine level instructions i.e fundamental types such as *int, float , double etc in case of C programming are known as primitive data structures.*

NON- PRIMITIVE DATA STRUCTURE

- Are highly developed complex data structures.
- Developed from the primitive data structure.
- Examples :
- Arrays, Lists, stack, queue, trees
- Two Types of non-primitive data structures:
  i. Linear Data Structure
  ii. Non linear data structure

LINEAR DATA STRUCTURE

- Data structure where the data elements are processed in some sequence or linear fashion.
- The various operations on a data structure are possible only in a sequence.
- Examples : array, stacks, queue and linked list
- Operations:
    - Add an element
    - Delete an element
    - Traverse
    - Sort the list of elements
    - Search for a data element

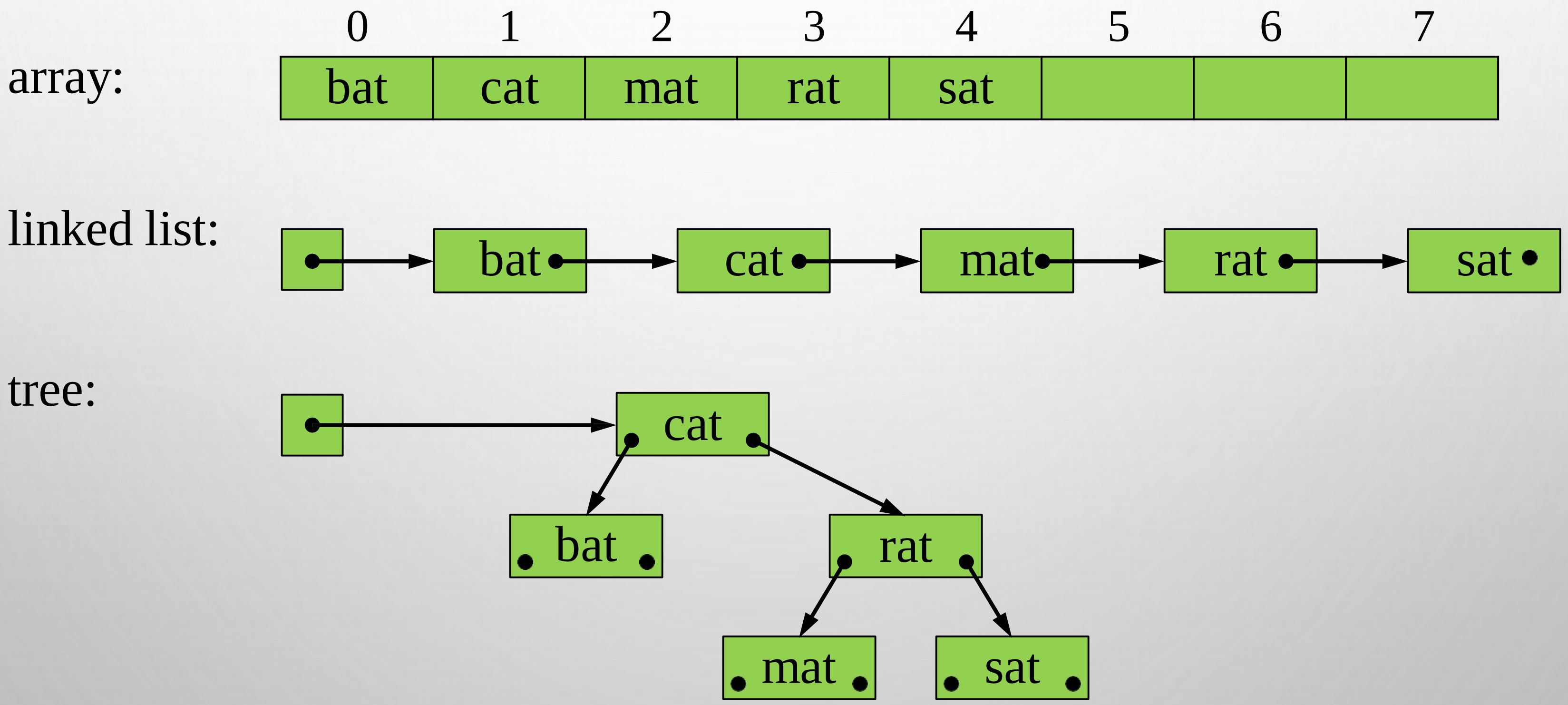
NON - LINEAR DATA STRUCTURE

- Data structure where the data elements are processed in some arbitrary fashion without any sequence.

- Examples : tree, graphs
- Operations:
    - Add elements
    - Delete elements
    - Sort the list of elements
    - Search for a data element

DATA STRUCTURES

- Possible data structures to represent the set of words {bat, cat, mat, rat, sat}:



STATIC AND DYNAMIC

- Data structures can be classified as **static** and **dynamic**;
- A **static data structure** is one whose capacity is fixed at creation (for example, array);
- A **dynamic data structure** is one whose capacity is variable, so it can expand or contract at any time (for example, linked list and tree)
- For each data structure we need algorithms for insertion, deletion, searching, sorting etc.

REPRESENTATION OF DATA STRUCTURES

- Sequential Representation
- Linked Representation

REPRESENTATION OF DATA STRUCTURES

Sequential Representation:

- A sequential representation maintains the data in continuous memory locations, such as arrays.
- *It takes less time to retrieve the data but lead to time complexity to insertion and deletion operations.*
- To insert new data at a particular position, the element of the list must be freed.
- To acquire free position shifting operations should be done.
- Wastage of CPU time occurs.

REPRESENTATION OF DATA STRUCTURES

Linked Representation:

- This representation maintains the *list by means of a link* between the adjacent elements *which need not to be stored in continuous memory locations*, such as linked list, tree.
- During insertion and deletion operations, links will be created or remove between which takes less time compared to the sequential representation.

TODAY'S TOPIC

- Abstract Data Types
- Introduction to Algorithm
- Analysis and Complexity
- Asymptotic Notations

ABSTRACT DATA TYPE (ADT)

- Abstract Data Type (ADT) is special kind of datatype that hides inner structure and design of the data type from the user
- *Most commonly used ADTs are **linked list, stack, queues, trees, hash tables, graphs***
- If someone wants to use ADT in a program, he/she can simply use operations without knowing its implementation detail

ABSTRACT DATA TYPE (ADT)

- We know what the *int* can do i.e add, subtract, multiply etc. *But we need not even know how an int is actually stored.*
- As user, *we use the int operations without having to know how they are implemented.*
- To simplify the process of solving problems, *we combine the data structures with their operations* and we call this Abstract Data Types.

- The ADTs do not worry about the implementation details.
- They come into the picture only when we want to use them.

INTRODUCTION TO ALGORITHMS

- An algorithm is *the step by step finite sequence of instructions for solving a stated problem*.
- An algorithm is thus a sequence of computational steps that transform the input into the output
- We can also view an *algorithm as a tool for solving a well-specified computational problem*
- Examples: sorting algorithms, searching algorithms etc.

CHARACTERISTICS TO ALGORITHMS

- Input : An algorithm should have inputs
- Definiteness: The processing rules specified in the algorithm must be unambiguous and lead to specific action
- Effectiveness: Each instruction should be sufficient basic and easy to understand.
- Finiteness: Every step in an algorithm should end in finite amount of time.
- Output: An algorithm should have output
- Correctness: Correct set of output must be produced from the output

WHY ANALYSIS OF ALGORITHM?

Scenario:

- To go from city "A" to city "B", there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle.
- Depending on the availability and convenience, we choose the one that suits us.
- *Similarly, in computer science, multiple algorithms are available for solving the same problem* (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more).
- Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

EFFICIENCY OF ALGORITHMS

- **Time efficiency:** How much **time** does the algorithm require? **[significant operations]**
- **Space efficiency:** How much **space** (memory) does the algorithm require? **[Extra space]**
- In general, both time and space requirements depend on the algorithm's input (typically the "size" of the input)

COMPUTATIONAL AND ASYMPTOTIC COMPLEXITY

- Whenever we want to perform analysis of an algorithm *we need to calculate the complexity of that algorithm*. But *when we calculate complexity of an algorithm it does not provide exact amount of resource required*

- So instead of taking exact amount of resource we present that complexity in general form which produces the basic nature of that algorithm

- We analyze the algorithm in terms of bound the it would be easier. For this purpose, we need the concept of asymptotic notations

ASYMPTOTIC NOTATIONS

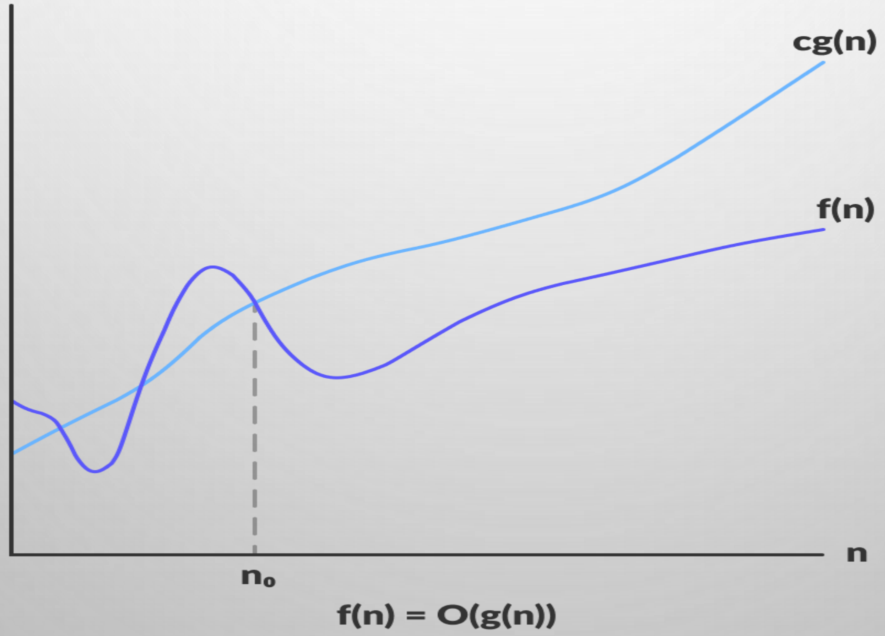
- The efficiency of algorithm is measured with the help of asymptotic notations.
- The *study of change in performance of the algorithm with the change in the order of the input size* is defined as asymptotic analysis.
- Three most common asymptotic notations: Big-oh (O), omega (Ω), and theta (θ).
- **Big-oh (O):**
  - It is the formal method of expressing the upper bound of an algorithm's running time.
  - It is the measure of the longest amount of time.

ASYMPTOTIC NOTATIONS

- The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if there exist positive constant c and n_0 such that

$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0$$

- For example, 3n + 2 = O(n) as 3n+2 ≤ 4n **for** all n ≥ 2
- Hence, the complexity of **f(n)** can be represented as O (g (n))



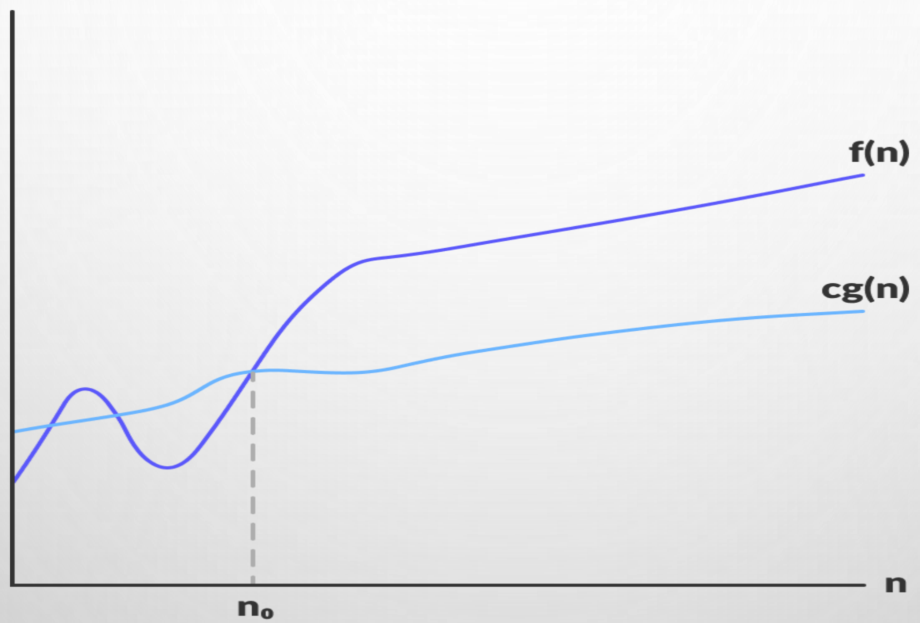
ASYMPTOTIC NOTATIONS

- **Omega (Ω):**
  - The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

    $$0 \le cg(n) \le f(n) \text{ for all } n \ge n_0$$

  - It gives lower bound of an algorithms running time
  - For example, $8n^2 + 2n - 3 = \Omega(n^2)$ as $8n^2 + 2n - 3 \ge 7n^2$ for all $n \ge 1$
  - Hence, the complexity of **f(n)** can be represented as Ω (g (n))

ASYMPTOTIC NOTATIONS



$f(n) = \Omega(g(n))$

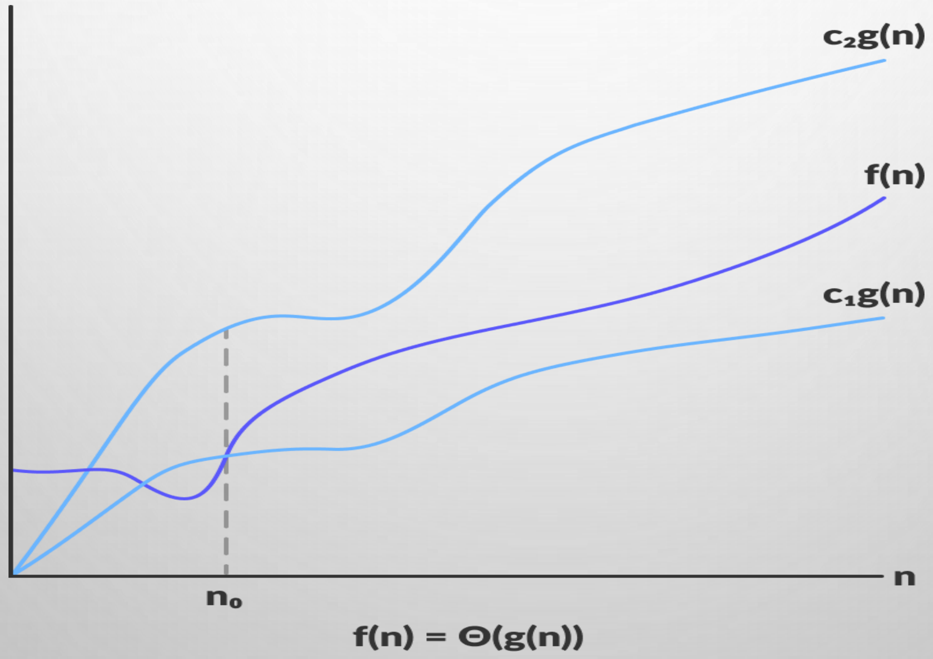
ASYMPTOTIC NOTATIONS

- **Theta (θ):**
  - The function f (n) = θ (g (n)) [read as "f of n is theta of g of n"] if and only if there exists positive constant c and n_0 such that

    $$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$
  - Gives both upper and lower bound of algorithms running time
  - The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.
  - For example, 3n + 2 = θ(n) as 3n + 2 ≥ 3n and 3n+2 ≤ 4n, for c1 = 3, c2 = 4, and n_0 = 2

ASYMPTOTIC NOTATIONS

- The function f (n) = θ (g (n)) [read as "f of n is theta of g of n"] if and only if there exists positive constant c and n_0 such that



FINDING BIG OH (O) NOTATION

- For many interesting algorithms, *the exact number of operations is too difficult to analyze mathematically.*
- To simplify the analysis:
  - *identify the fastest-growing term*
  - *neglect slower-growing terms*
  - *neglect the constant factor in the fastest-growing term.*
- The resulting formula is the algorithm's **time complexity**. It focuses on the **growth rate** of the algorithm's time requirement.
- Similarly for **space complexity**.

FINDING BIG OH (O) NOTATION

- For example, suppose maximum number of operations of an algorithm is

  $f(n) = 5n^2 – 3n + 2$

  simplify to $5n^2$

  then to n^2

  Time complexity is **of order n^2**. This is written ***$O(n^2)$***.

FINDING BIG OH (O) NOTATION

- Common time complexities:

$O(1)$	**constant** time	(feasible)
$O(\log n)$	**logarithmic** time	(feasible)
$O(n)$	**linear** time	(feasible)
$O(n \log n)$	**log linear** time	(feasible)
$O(n^2)$	**quadratic** time	(sometimes feasible)
$O(n^3)$	**cubic** time	(sometimes feasible)
$O(2^n)$	**exponential** time	(rarely feasible)

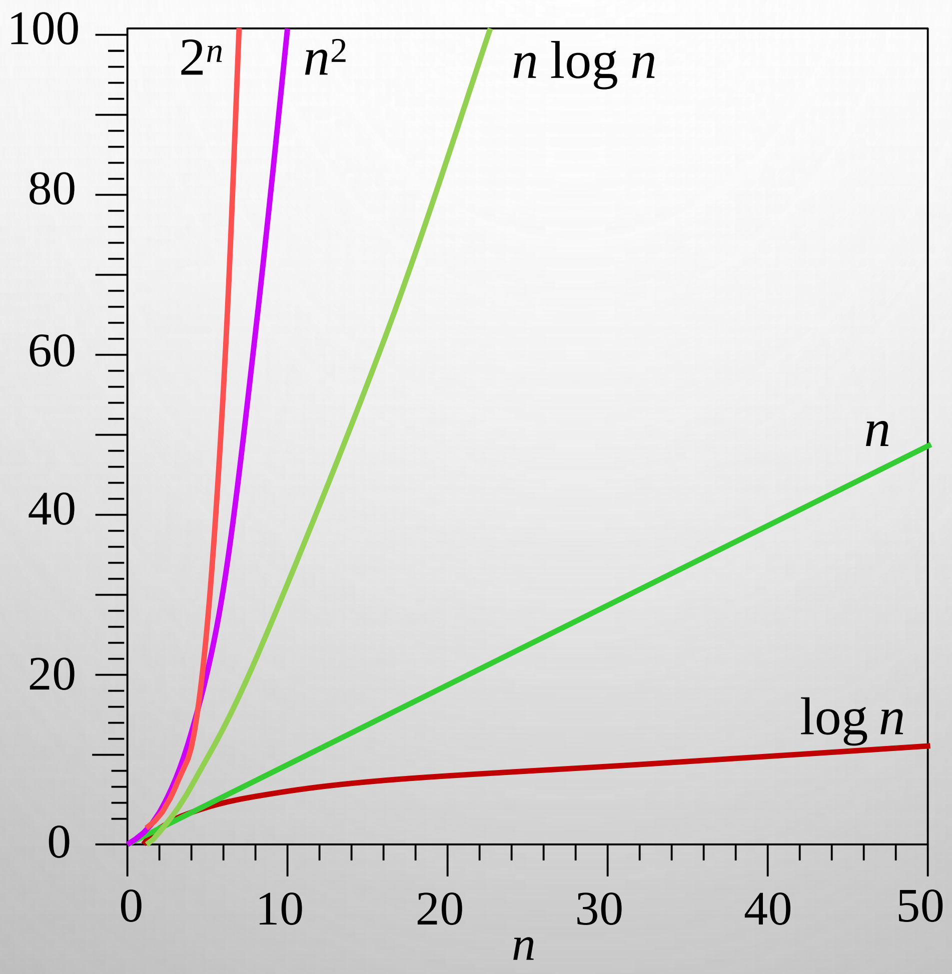
FINDING BIG OH (O) NOTATION

- Comparison of growth rate:

1	1	1	1	1
log n	3.3	4.3	4.9	5.3
n	10	20	30	40
n log n	33	86	147	213
n^2	100	400	900	1,600
n^3	1,000	8,000	27,000	64,000
2^n	1,024	1.0 million	1.1 billion	1.1 trillion

FINDING BIG OH (O) NOTATION

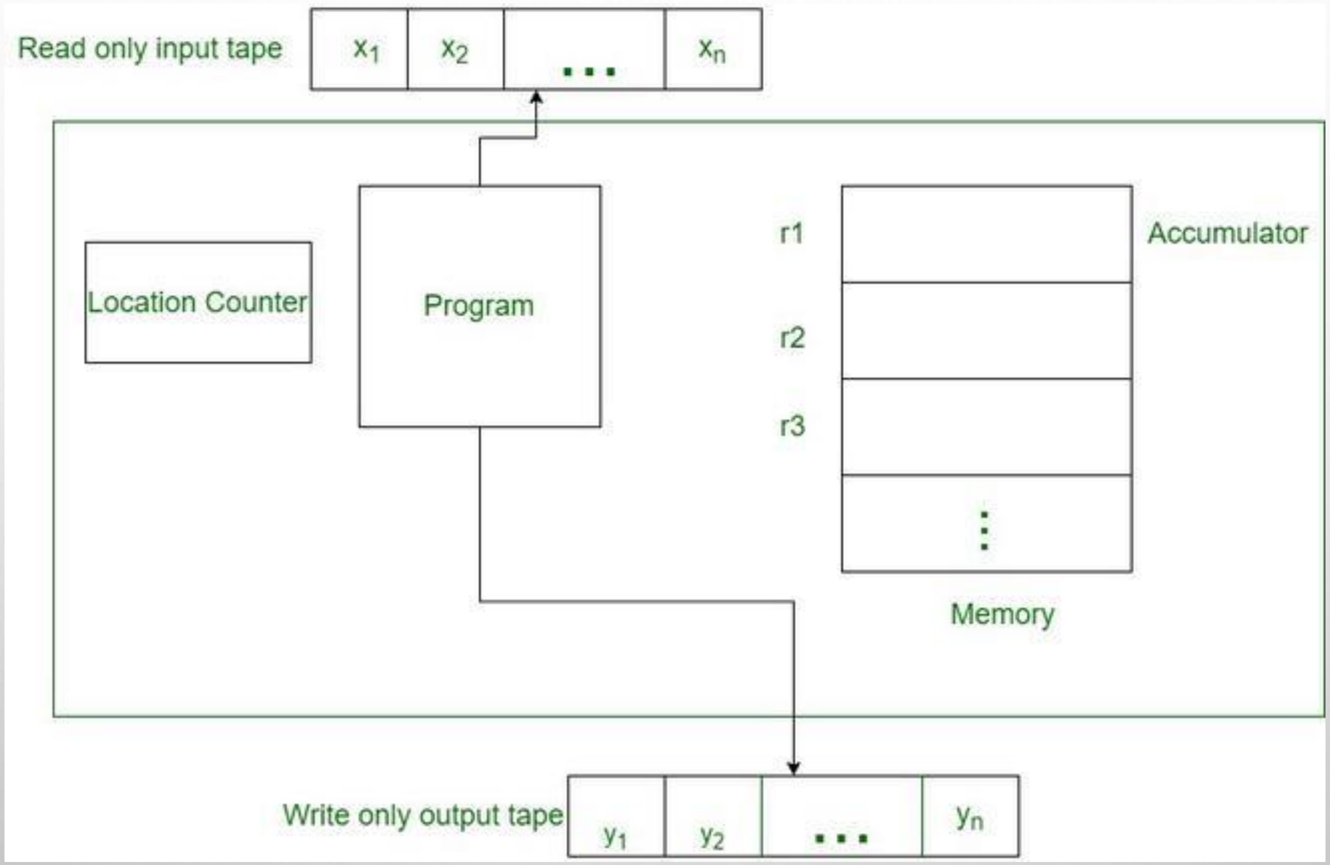
- Graphically:



RANDOM ACCESS MODEL (RAM)

- *The RAM model is used for analyzing algorithms* without running them on a physical machine.
- The RAM model has the following properties:
  - A basic or primitive operation (+, -, /, *, =, if) takes one time step.
  - Loops and subroutines are not basic operation
  - In computing time complexity, one good approach is to count primitive operations
  - We measure run time of algorithm by counting the steps
  - Each memory reference takes one step.

RANDOM ACCESS MODEL (RAM)



- Fig : RAM Model

RANDOM ACCESS MODEL (RAM)

- *Input tape/Output tape:* Input tape consists of a sequence of squares, each of which can store integer. Whenever one square is read from the tape head moves one square to the right. Output is written on the square under the tape head and the after the writing, the tape head moves one square to the right over writing on the same square is not allowed.

- *Program:* Program for RAM contains of labeled instructions resembling those found in assembly language programs. All computations take place in the first register called accumulator.

RANDOM ACCESS MODEL (RAM)

- *Memory:* The memory consists of a sequence of registers, each of which is capable of holding an integer.

- *Program Counter:* The program counter determines next operation to be executed. Some examples of operations are:
  - Assigning a value to a variable
  - Performing an arithmetic operation
  - Indexing to an array
  - Returning from method
  - Calling a method etc.

DETAILED ANALYSIS

- Example 1: Algorithm for sequential search

```
sequential _search(A, n, key)
{
          int i, flag = 0;
          for(i=0;i<n;i++)
          {
               if(A[i] == key)
               {
                    flag = 1;
               }
          }
          if(flag ==1 )
               printf("Search successful")
          else
               printf("Search un-successful")   }
```

DETAILED ANALYSIS

- *Time complexity:* By counting number of statements takes the total time complexity

1. Declaration statement takes 1 step time
2. In for loop:
    - i = 0 takes 1 step time
    - i<n takes (n+1) step time
    - i++ takes 'n' step time
    - Within for loop if condition takes n step time
    - Within if statement flag = 0 takes at most n step time
3. If statement takes 1 step time
4. Print statement takes 1 step time

$T(n) = 1 + 1 + (n+1) + n + n + n +1 + 1 = 5 + 4n$

$T(n) = O(n)$

DETAILED ANALYSIS

- *Space complexity:* By counting number of memory references gives the space complexity

1. The variable ‘i’ takes 1 unit space
2. The variable ‘n’ takes 1 unit space
3. The variable ‘key’ takes 1 unit space
4. The variable ‘flag’ takes 1 unit space
5. The variable ‘A’ takes n unit space

Total space complexity:

$T(n) = 1 + 1 + 1 + 1 + n$

$= 4 + n$

$T(n) = O(n)$

THANK YOU!