UNIT-I

1.1 INTRODUCTION TO DATA STRUCTURES

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.

Data structure provides a way of organizing, managing, and storing data efficiently. With the help of data structure, the data items can be traversed easily. Data structure provides efficiency, reusability and abstraction. It plays an important role in enhancing the performance of a program because the main function of the program is to store and retrieve the user's data as fast as possible.

There are different data-structures used for the storage of data.

For Examples: Array, Stack, Queue, Tree, Graph, etc.

Why Learn Data Structure?

Data structure and algorithms are two of the most important aspects of computer science. Data structures allow us to organize and store data, while algorithms allow us to process that data in a meaningful way. Learning data structure and algorithms will help you become a better programmer. You will be able to write code that is more efficient and more reliable. You will also be able to solve problems more quickly and more effectively.

1.2 CLASSIFICATION OF DATA STRUCTURES:

There are 2 types of Data Structures:

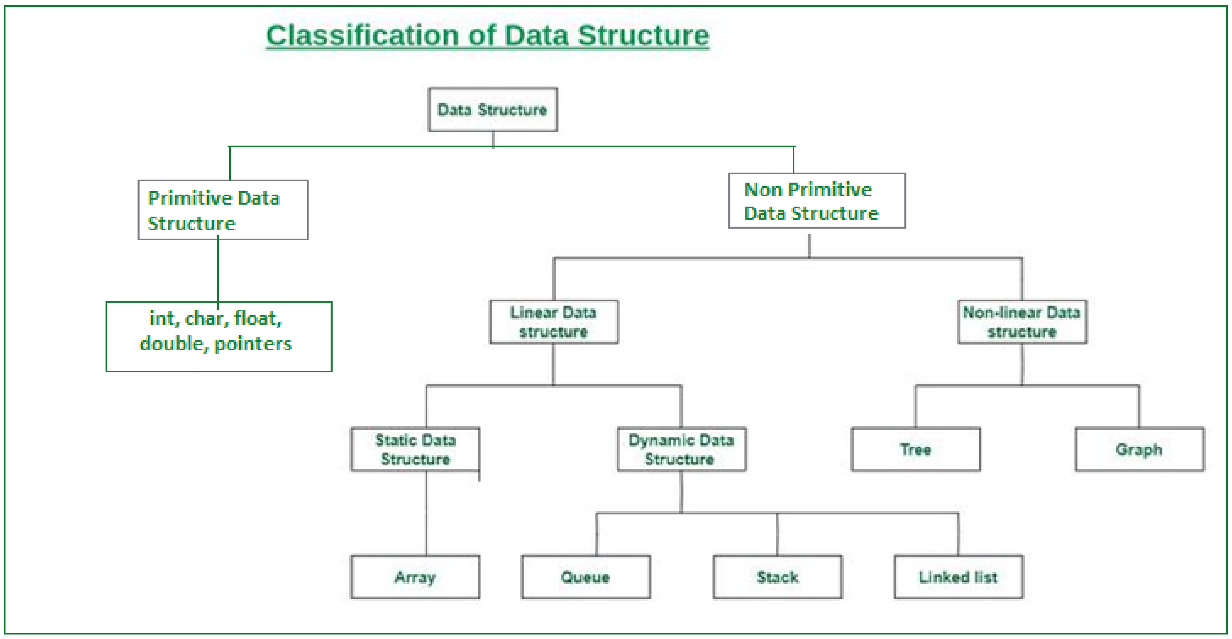
- Primitive Data Structure
- Non – Primitive Data Structure

Primitive Data Structure –

Primitive Data Structures directly operate according to the machine instructions. These are the primitive data types. Data types like int, char, float, double, and pointer are primitive data structures that can hold a single value.

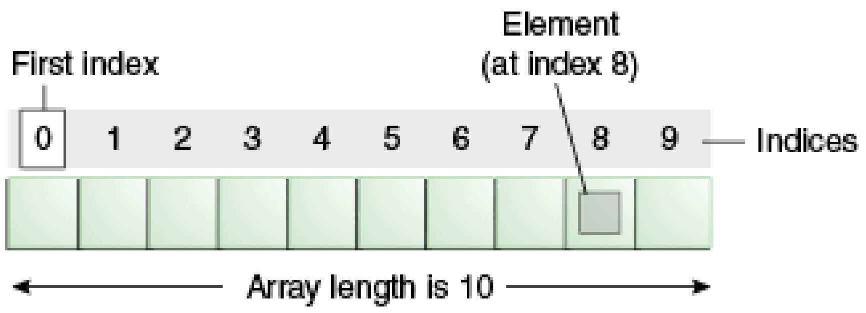
Non – Primitive Data Structure –

Non-primitive data structures are complex data structures that are derived from primitive data structures. Non – Primitive data types are further divided into two categories.



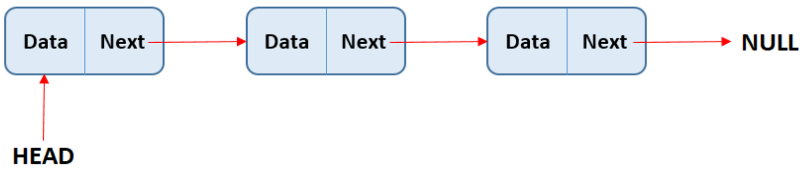
a) Arrays:

- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
- Array elements can be denoted by A(1),A(2),A(3),.....A(N).



b) Linked List:

- A linked list is the most sought-after data structure when it comes to handling dynamic data elements.
- A linked list consists of a data element known as a node.
- And each node consists of two fields: one field has data, and in the second field, the node has an address that keeps a reference to the next node.



c) Trees:

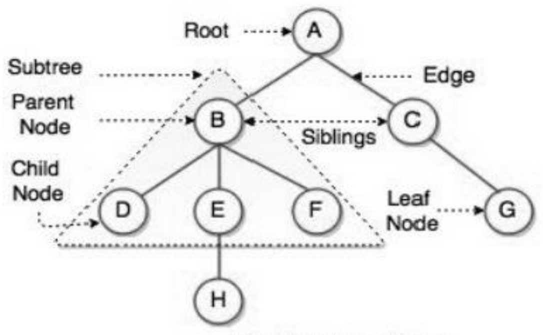
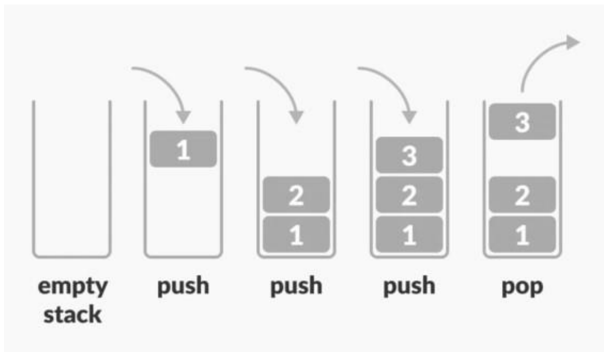


Fig. Structure of Tree

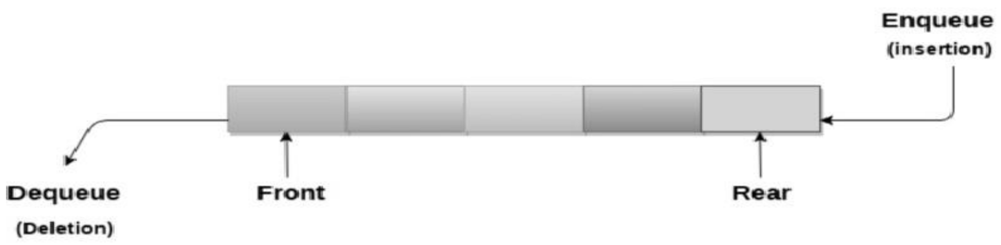
- Data frequently contain a hierarchical relationship between various elements.
- The data structure which reflects this relationship is called rooted tree graph or simply a tree.

d) Stack:



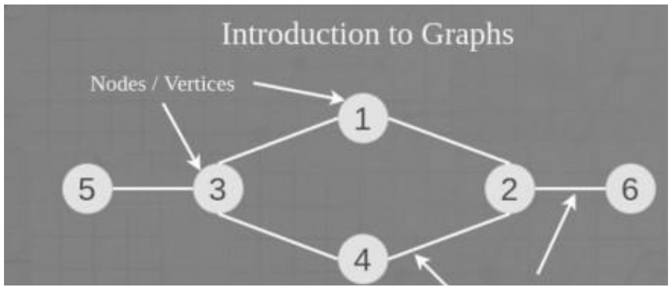
- Stack is also called a last in first out (LIFO) system.
- It is a linear list in which insertion and deletion can take place only at one end, called the top.

e) Queue:



- A queue is also called a first in first out (FIFO) system.
- It is a linear list in which deletion can take place at one end of the list which is 'front'.
- Insertion takes place at the end of the list which is 'rear'.

f) Graph:



- Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature.
- The data structure which reflects this type of relationship is called graph.

1.3 DATA STRUCTURE OPERATIONS

The common operations that can be performed on the data structures are as follows

- **Insertion** – We can insert new data elements in the data structure.
- **Deletion** – We can delete the data elements from the data structure.
- **Updation** – We can update or replace the existing elements from the data structure.
- **Searching** – Finding the location of the record with a given key value or finding the locations of all records which satisfy one or more condition.
- **Sorting** – We can sort the elements either in ascending or descending order.

Advantages of Data Structure –

1. Data structures allow storing the information on hard disks.
2. An appropriate choice of ADT (Abstract Data Type) makes the program more efficient.
3. Data Structures are necessary for designing efficient algorithms.

4. It provides reusability and abstraction.
5. Using appropriate data structures can help programmers save a good amount of time while performing operations such as storage, retrieval, or processing of data.
6. Manipulation of large amounts of data is easier.

1.4 INTRODUCTION TO ALGORITHMS

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as input, processes data according to the program instructions and finally produces a result.

Specifications of Algorithms

Every algorithm must satisfy the following specifications...

1. **Input -** Every algorithm must take zero or more number of input values from external.
2. **Output -** Every algorithm must produce an output as result.
3. **Definiteness -** Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).
4. **Finiteness -** For all different cases, the algorithm must produce result within a finite number of steps.
5. **Effectiveness -** Every instruction must be basic enough to be carried out and it also must be feasible.

a) What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

b) <u>What is Space complexity?</u>

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like funcion calls, jumping statements etc,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

Example1 :

```
int square(int a)
{
        return a*a;
}
```

In the above piece of code, it requires 2 bytes of memory to store variable **'a'** and another 2 bytes of memory is used for **return value**.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be *Constant Space Complexity.*

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code...

Example 2 :

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

In the above piece of code it requires
'n*2' bytes of memory to store array variable **'a[ ]'**
2 bytes of memory for integer parameter **'n'**
4 bytes of memory for local integer variables **'sum'** and **'i'** (2 bytes each)
2 bytes of memory for **return value**.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be ***Linear Space Complexity.***

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

c) What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

The time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine.
3. Read and Write speed of the machine.
4. The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
5. Input data.

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine

2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above-defined model machine...

Consider the following piece of code...

Example 1 :

```
int sum(int a, int b)
{
  return a+b;
}
```

In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires the same amount of time i.e. 2 units.

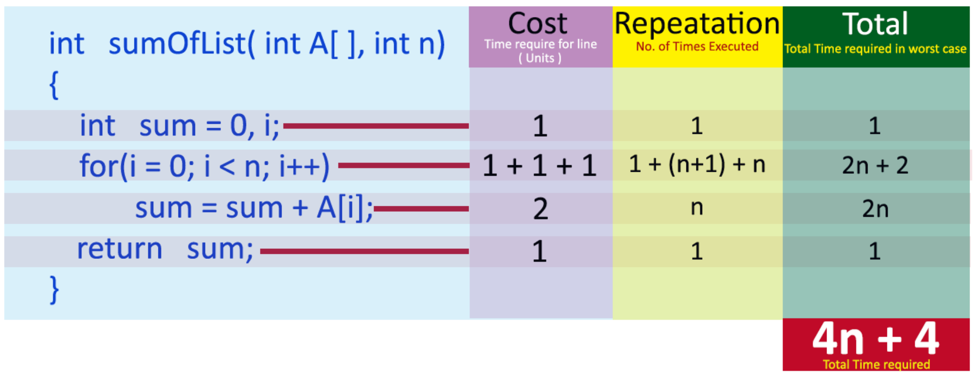
If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

Example 2 :

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

For the above code, time complexity can be calculated as follows...



	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
`int sumOfList( int A[ ], int n)`			
`{`			
`int sum = 0, i;`	1	1	1
`for(i = 0; i < n; i++)`	1 + 1 + 1	1 + (n+1) + n	2n + 2
`sum = sum + A[i];`	2	n	2n
`return sum;`	1	1	1
`}`			
			4n + 4 Total Time required

In above calculation,

Cost is the amount of computer time required for a single operation in each line.

Repetition is the amount of computer time required by each operation for all its repetitions.

Total is the amount of computer time required by each operation to execute.
So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is ***Linear Time Complexity.***

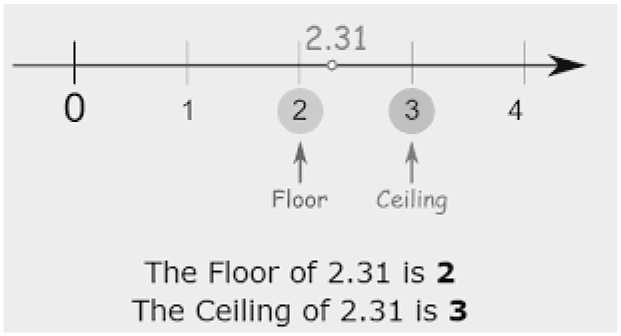
If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

1.5 MATHEMATICAL NOTATIONS AND FUNCTIONS:

a) Floor and Ceiling Functions:

- The ceiling function returns the smallest nearest integer which is greater than or equal to the specified number.
- The floor function returns the largest nearest integer which is less than or equal to a specified value.
- Let x be any real number.
- Then x lies between two integers called the floor and ceiling of x.

- X, called the floor of x, denotes the greatest integer that does not exceed x.
- X, called the ceiling of x, and denotes the least integer that is not less than x.
- If x is itself an integer, then [x]=[x]; otherwise [x]+1=[x].



b)Remainder Function: Modular Arithmetic:

- Modular arithmetic is a system of arithmetic for integers, which considers the remainder.
- In modular arithmetic, numbers "wrap around" upon reaching a given fixed quantity (this given quantity is known as the modulus) to leave a remainder.
- **Modulo** is a math operation that finds the remainder when one integer is divided by another. In writing, it is frequently abbreviated as **mod**, or represented by the symbol **%**.

c) Integer and Absolute value functions

- The absolute value (or modulus) | x | of a real number x is the non-negative value of x without regard to its sign.
- For example, **the absolute value of 5 is 5, and the absolute value of -5 is also 5**.
- The absolute value of a number may be thought of as its distance from zero along real number line.

d) Summation Symbol; Sums:

- The Greek capital letter, Σ, is used to represent the sum.
- The series 4+8+12+16+20+24 can be expressed as **6Σn=14n**.
- The expression is read as the sum of 4n as n goes from 1 to 6.
- The variable n is called the index of summation.

e) Factorial Function:

- The product of the positive integers from 1 to n, inclusive is denoted by n!

  N! = 1.2.3.... (n-2)(n-1)n

f) Permutations:

- Any arrangement of a set of n objects in a given order is called Permutation of Object.
- Any arrangement of any $r \leq n$ of these objects in a given order is called an r-permutation or a permutation of n object taken r at a time.

g) Exponents and Logarithms

- A logarithm is an exponent which indicates to what power a base must be raised to produce a given number.
- $y = b^x$ exponential form. $x = \log_b y$ logarithmic form. x is the logarithm of y to the base b.

1.6 WHAT IS ASYMPTOTIC NOTATION?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required. So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**.

Here, for larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm.

In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

a) Big - Oh Notation (O):

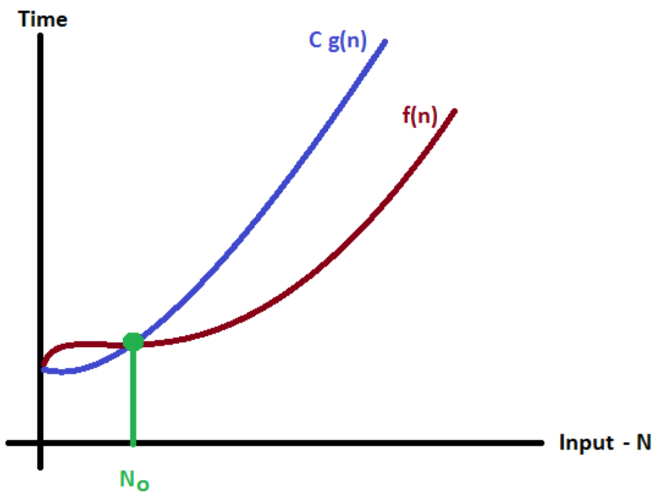
Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity. Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If $f(n) <= C\, g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent f(n) as O(g(n)).

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) <= C g(n)
⇒3n + 2 <= C n
Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

b) Big - Omega Notation (Ω):

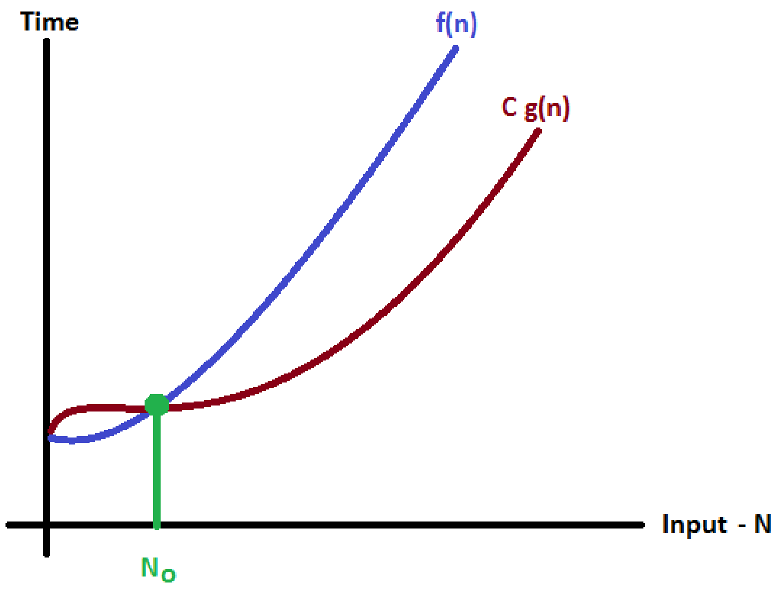
Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) >= C g(n)
⇒3n + 2 >= C n
Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

c) Big - Theta Notation (Θ):

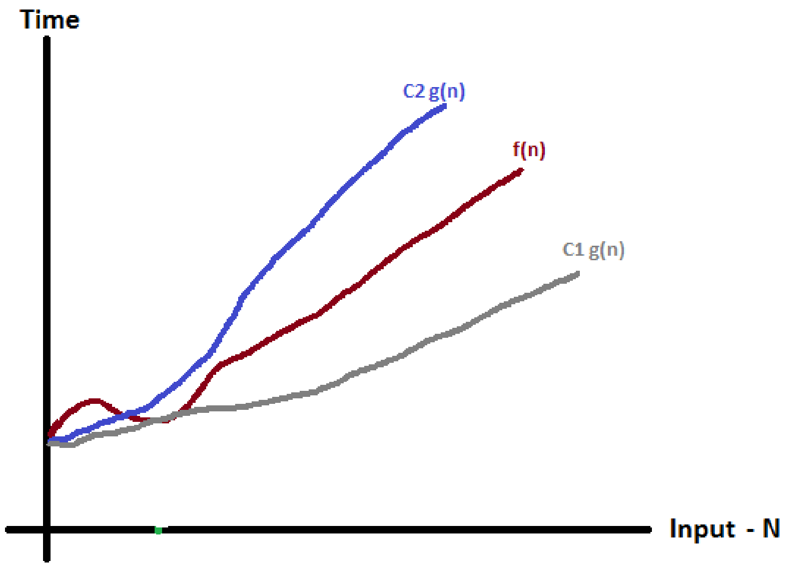
Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity. Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2

$g(n) = n$
If we want to represent **$f(n)$** as **$\Theta(g(n))$** then it must satisfy **$C_1 g(n) <= f(n) <= C_2 g(n)$** for all values of **$C_1 > 0, C_2 > 0$** and **$n_0 >= 1$**
$C_1 g(n) <= f(n) <= C_2 g(n)$
$\Rightarrow C_1 n <= 3n + 2 <= C_2 n$

Above condition is always TRUE for all values of **$C_1 = 1, C_2 = 4$** and **$n >= 2$**. By using Big - Theta notation we can represent the time complexity as follows...
$3n + 2 = \Theta(n)$

d) <u>Little oh notation(o):</u>

Little-o notation is used to denote an upper-bound that is not asymptotically tight. It is formally defined as: $o(g(n))=\{f(n)$ for any positive constant c, there exists positive constant n_0 such that **$f(n) < c.g(n)$** for all $n>= n_0\}$.

Note that in this definition**,** the set of functions **$f(n)$** are strictly smaller than **$c.g(n)$,** meaning that little-o notation is a stronger upper bound than big-O notation**.** In other words, the little-o notation does not allow the function f(n) to have the same growth rate as g(n).

The mathematical representation of little oh is as follow:

$f(n)= o(g(n))$; or $f(n) < c.g(n)$.

e) <u>Little omega notation(ω):</u>

Another asymptotic notation is little omega notation. it is denoted by(ω). Little omega (ω) notation is used to describe a loose lower bound of f(n).

f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions. In the case of Big Omega $f(n)=\Omega(g(n))$ and the bound is $0<=cg(n)<=f(n)$, but in case of little omega, it is true for $0<=c*g(n)<f(n)$.

The mathematical representation of little omega is as follow:

$f(n)= \omega (g(n))$; or $f(n) > c.g(n)$.

1.7 LINEAR SEARCH ALGORITHM (SEQUENTIAL SEARCH ALGORITHM)

This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with the next element in the list.

Repeat the same until search element is compared with the last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

In our worst-case scenario, this is not very efficient. We have to check every single number in the list until we get to our answer. This method is called Linear Search. **The Big O notation for Linear Search is O(N). The complexity is directly related to the size of the inputs.**

Linear search is implemented using following steps...

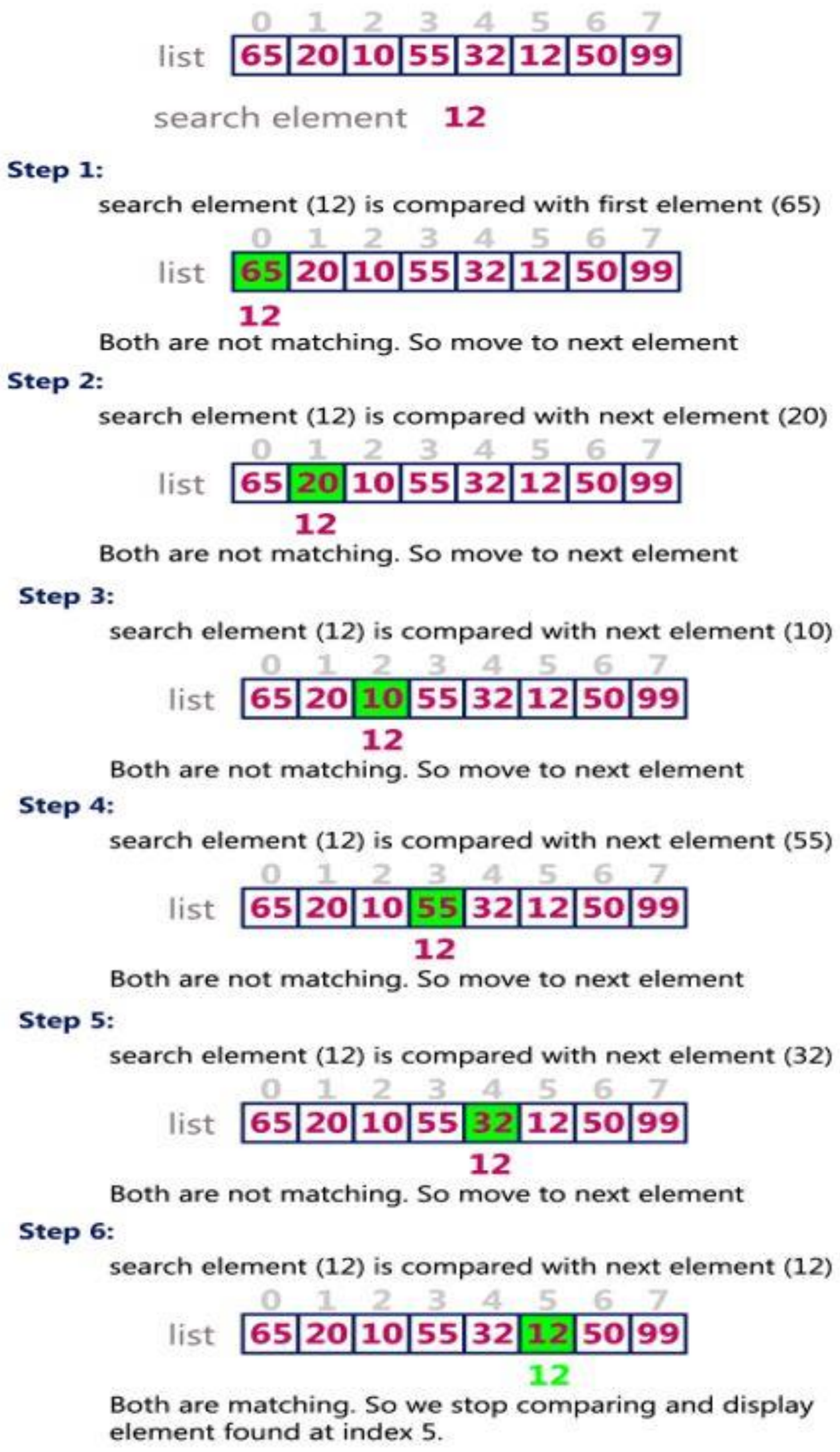
- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

SPACE AND TIME COMPLEXITY OF LINEAR SEARCH

Time Complexity (best)	O(1)
Time Complexity (average)	O(n)
Time Complexity (worst)	O(n)
Space Complexity	O(1)

Example

Consider the following list of elements and the element to be searched...



0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
search element 12

Step 1:

search element (12) is compared with first element (65)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

1.8 BINARY SEARCH ALGORITHM

The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order.

The binary search cannot be used for a list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list.

If the search element is smaller, then we repeat the same process for the left sublist of the middle element. If the search element is larger, then we repeat the same process for the right sublist of the middle element.

We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

This strategy is called Binary Search. It is efficient because it eliminates half of the list in each pass. **Binary search algorithm finds a given element in a list of elements with O(log n) time complexity where n is total number of elements in the list.**

O(log N) means that the algorithm takes an additional step each time the data doubles.

Binary search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

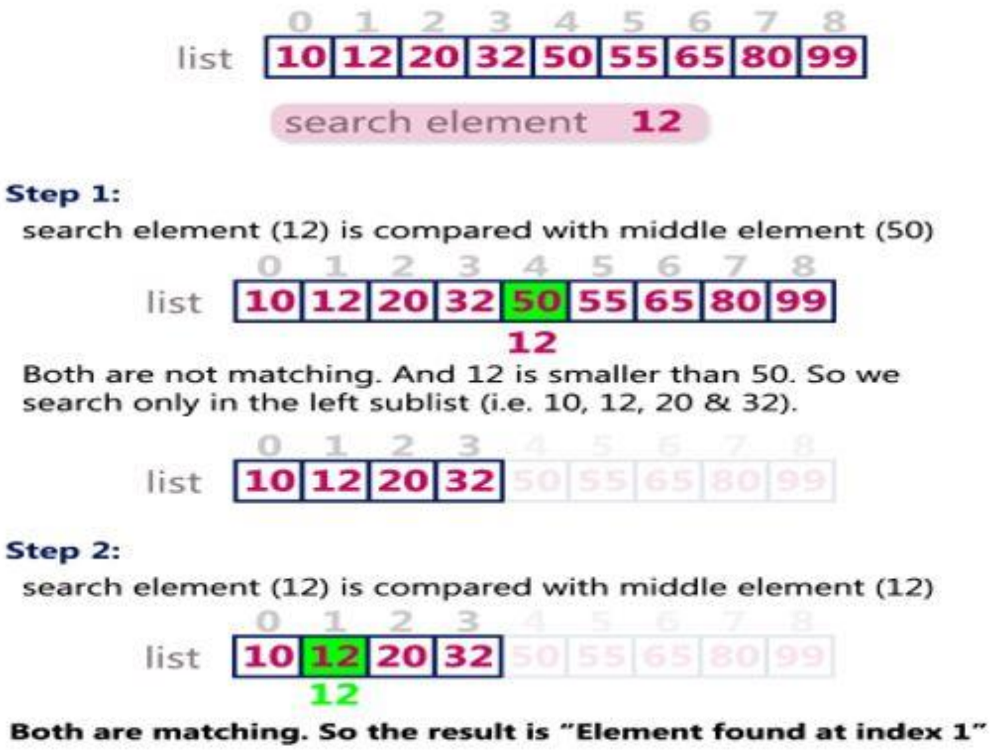
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

SPACE AND TIME COMPLEXITY OF BINARY SEARCH

Time complexity (best)	O(1)
Time complexity (average)	O(log(n))
Time complexity (worst)	O(log(n))
Space complexity	O(1)

Example

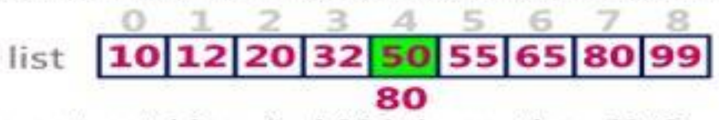
Consider the following list of elements and the element to be searched...


0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
search element 12
Step 1:
search element (12) is compared with middle element (50)
12
Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).
Step 2:
search element (12) is compared with middle element (12)
12
Both are matching. So the result is "Element found at index 1"

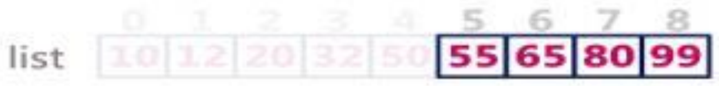
search element **80**

Step 1:

search element (80) is compared with middle element (50)

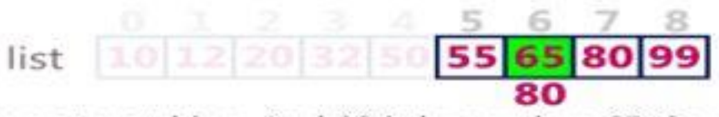


Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

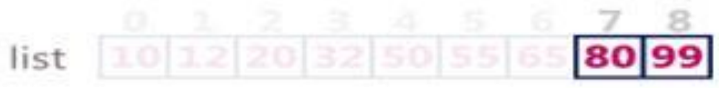


Step 2:

search element (80) is compared with middle element (65)

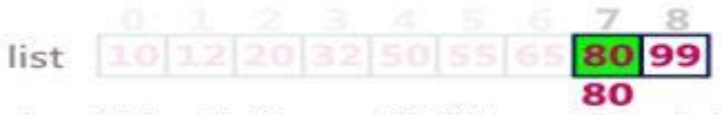


Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).



Step 3:

search element (80) is compared with middle element (80)



Both are not matching. So the result is "Element found at index 7"