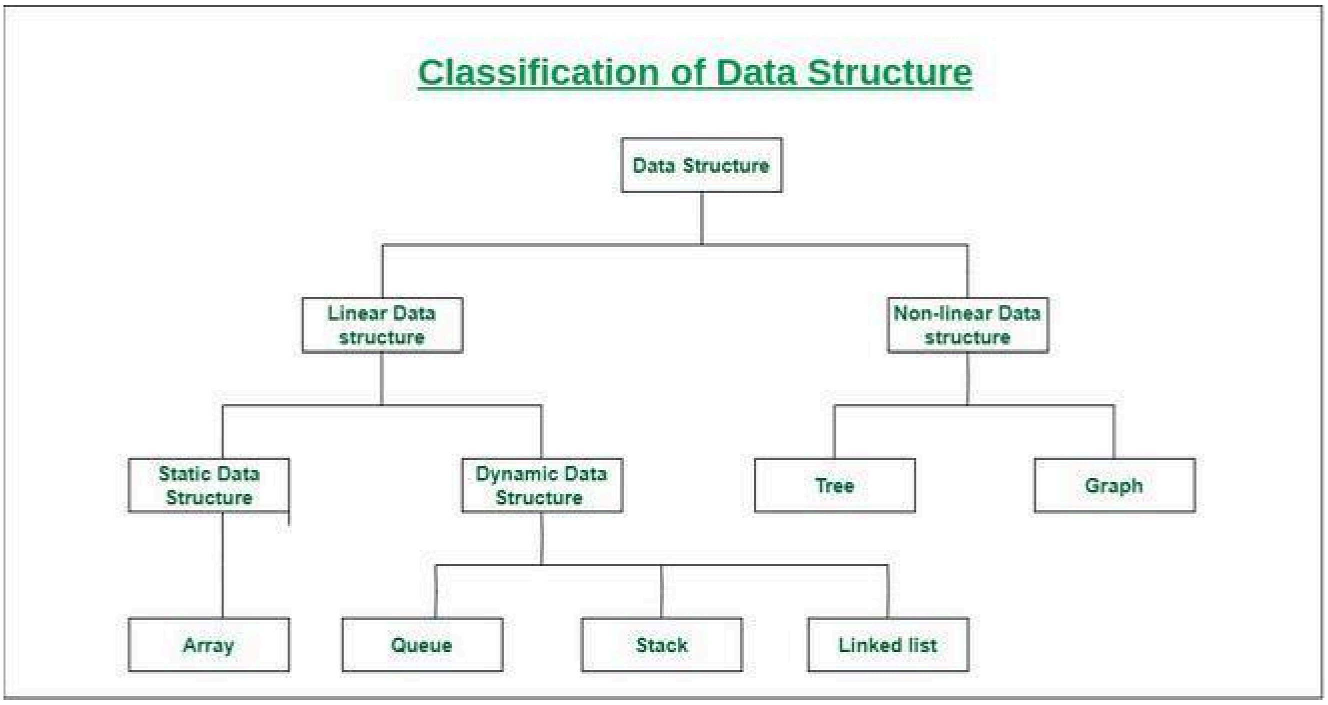
Subject : DSA **UNIT 1**

What is DSA?

Data Structures and Algorithms (DSA) refer to the study of methods for organizing and storing data and the design of procedures (algorithms) for solving problems, which operate on these data structures.

Data structures are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organizing, processing, accessing, and storing data efficiently

- Data structure is a collection of different kinds of data. That entire data can be represented using an object and can be used throughout the program.
- It can hold multiple types of data within a single object
- Data structure implementation is known as concrete implementation
- In data structure objects, time complexity plays an important role.
- While in the case of data structures, the data and its value acquire the space in the computer's main memory. Also, a data structure can hold different kinds and types of data within one single object



1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
    1. Linear Data Structures
    2. Non-Linear Data Structures

- **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.

Examples of linear data structures are array, stack, queue, linked list, etc.

- **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

  An example of this data structure is an array.

- **Dynamic data structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.

  Examples of this data structure are queue, stack, etc.

- **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.

  Examples of non-linear data structures are trees and graphs.

For example, we can store a list of items having the same data-type using the array data structure.

Need Of Data structure :

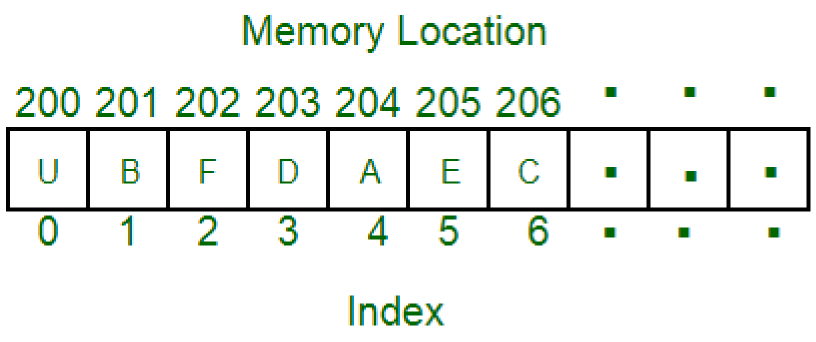
The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.

Data structures provide an easy way of organizing, retrieving, managing, and storing data. Here is a list of the needs for data.

1. Data structure modification is easy.
2. It requires less time.
3. Save storage memory space.
4. Data representation is easy.
5. Easy access to the large database

Arrays:

An array is a linear data structure and it is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together in one place. It allows the processing of a large amount of data in a relatively short period. **The first element of the array is indexed by a subscript of 0**. There are different operations possible in an array, like Searching, Sorting, Inserting, Traversing, Reversing, and Deleting.



Characteristics of an Array:

Arrays use an index-based data structure which helps to identify each of the elements in an array easily using the index.

If a user wants to store multiple values of the same data type, then the array can be utilized efficiently.

An array can also handle complex data structures by storing data in a two-dimensional array.
An array is also used to implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.

The search process in an array can be done very easily

Applications of Array:

Different applications of an array are as follows:

- An array is used in solving matrix problems.
- Database records are also implemented by an array.
- It helps in implementing a sorting algorithm.
- It is also used to implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.
- Can be applied as a lookup table in computers.
- Arrays can be used in speech processing where every speech signal is an array.
- The screen of the computer is also displayed by an array. Here we use a multidimensional array.

The array is used in many management systems like a library, students, parliament, etc.

- The array is used in the online ticket booking system. Contacts on a cell phone are displayed by this array.
- In games like online chess, where the player can store his past moves as well as current moves. It indicates a hint of position.
- To save images in a specific dimension in the android Like 360*1200

Real-Life Applications of Array:

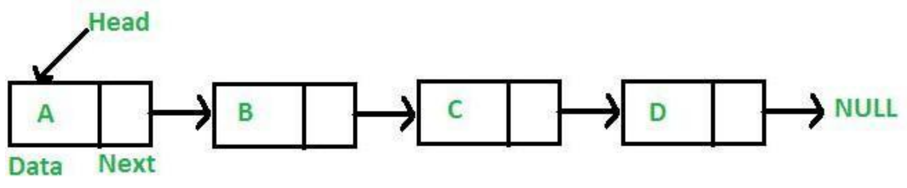
- An array is frequently used to store data for mathematical computations.
- It is used in image processing.
- It is also used in record management.
- Book pages are also real-life examples of an array.
- It is used in ordering boxes as well.

Linked list:

A linked list is a linear data structure in which elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

Types of linked lists:

- Singly-linked list
- Doubly linked list
- Circular linked list
- Doubly circular linked list



Characteristics of a Linked list:

A linked list has various characteristics which are as follows:

- A linked list uses extra memory to store links.
- During the initialization of the linked list, there is no need to know the size of the elements.
- Linked lists are used to implement stacks, queues, graphs, etc.
- The first node of the linked list is called the Head.
- The next pointer of the last node always points to NULL.
- In a linked list, insertion and deletion are possible easily.
- Each node of the linked list consists of a pointer/link which is the address of the next node.
- Linked lists can shrink or grow at any point in time easily.

Operations performed on Linked list:

A linked list is a linear data structure where each node contains a value and a reference to the next node. Here are some common operations performed on linked lists:

- **Initialization:** A linked list can be initialized by creating a head node with a reference to the first node. Each subsequent node contains a value and a reference to the next node.

- **Inserting elements:** Elements can be inserted at the head, tail, or at a specific position in the linked list.
- **Deleting elements**: Elements can be deleted from the linked list by updating the reference of the previous node to point to the next node, effectively removing the current node from the list.
- **Searching for elements**: Linked lists can be searched for a specific element by starting from the head node and following the references to the next nodes until the desired element is found.
- **Updating elements**: Elements in a linked list can be updated by modifying the value of a specific node.
- **Traversing elements:** The elements in a linked list can be traversed by starting from the head node and following the references to the next nodes until the end of the list is reached.
- **Reversing a linked list**: The linked list can be reversed by updating the references of each node so that they point to the previous node instead of the next node.

These are some of the most common operations performed on linked lists. The specific operations and algorithms used may vary based on the requirements of the problem and the programming language used.

Applications of the Linked list:

Different applications of linked lists are as follows:

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists are used to perform arithmetic operations on long integers.
- It is used for the representation of sparse matrices.
- It is used in the linked allocation of files.
- It helps in memory management.
- It is used in the representation of Polynomial Manipulation where each polynomial term represents a node in the linked list.
- Linked lists are used to display image containers. Users can visit past, current, and next images.
- They are used to store the history of the visited page.
- They are used to perform undo operations.

- Linked are used in software development where they indicate the correct syntax of a tag.
- Linked lists are used to display social media feeds.

Real-Life Applications of a Linked list:

- A linked list is used in Round-Robin scheduling to keep track of the turn in multiplayer games.
- It is used in image viewer. The previous and next images are linked, and hence can be accessed by the previous and next buttons.
- In a music playlist, songs are linked to the previous and next songs.

Stack:

Stack is a linear data structure that follows a particular order in which the operations are performed. The order is LIFO(Last in first out). Entering and retrieving data is possible from only one end. The entering and retrieving of data is also called push and pop operation in a stack. There are different operations possible in a stack like reversing a stack using recursion, Sorting, Deleting the middle element of a stack, etc.

Characteristics of a Stack:

Stack has various different characteristics which are as follows:

- Stack is used in many different algorithms like Tower of Hanoi, tree traversal, recursion, etc.
- Stack is implemented through an array or linked list.
- It follows the Last In First Out operation i.e., an element that is inserted first will pop in last and vice versa.
- The insertion and deletion are performed at one end i.e. from the top of the stack.
- In stack, if the allocated space for the stack is full, and still anyone attempts to add more elements, it will lead to stack overflow.

Applications of Stack:

Different applications of Stack are as follows:

- The stack data structure is used in the evaluation and conversion of arithmetic expressions.

- Stack is used in Recursion.

- It is used for parenthesis checking.
- While reversing a string, the stack is used as well.
- Stack is used in memory management.
- It is also used for processing function calls.
- The stack is used to convert expressions from infix to postfix.
- The stack is used to perform undo as well as redo operations in word processors.
- The stack is used in virtual machines like JVM.
- The stack is used in the media players. Useful to play the next and previous song.
- The stack is used in recursion operations.

Operation performed on stack ;

A stack is a linear data structure that implements the Last-In-First-Out (LIFO) principle. Here are some common operations performed on stacks:

- **Push**: Elements can be pushed onto the top of the stack, adding a new element to the top of the stack.
- **Pop**: The top element can be removed from the stack by performing a pop operation, effectively removing the last element that was pushed onto the stack.
- **Peek:** The top element can be inspected without removing it from the stack using a peek operation.
- **IsEmpty**: A check can be made to determine if the stack is empty.
- **Size**: The number of elements in the stack can be determined using a size operation.

Real-Life Applications of Stack:

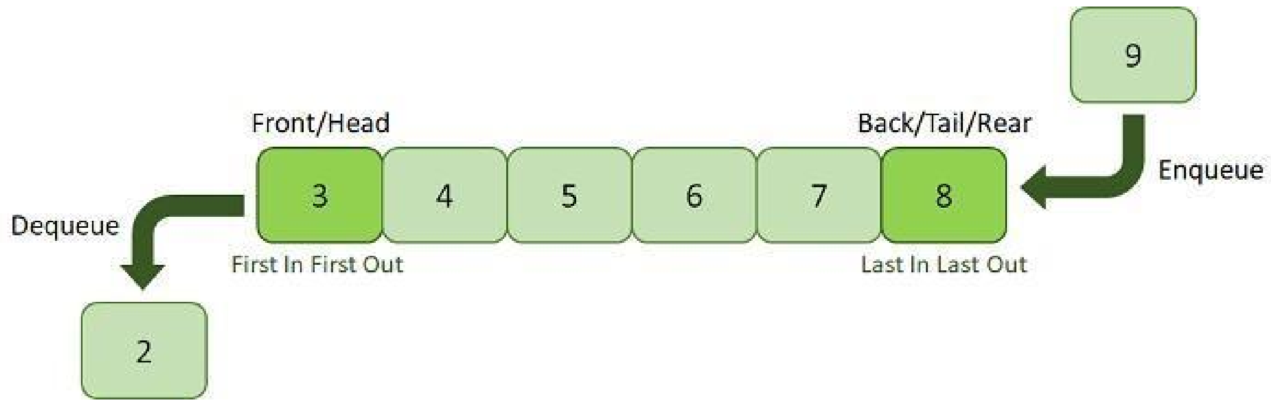
- Real life example of a stack is the layer of eating plates arranged one above the other. When you remove a plate from the pile, you can take the plate to the top of the pile. But this is exactly the plate that was added most recently to the pile. If you want the plate at the bottom of the pile, you must remove all the plates on top of it to reach it.
- Browsers use stack data structures to keep track of previously visited sites.
- Call log in mobile also uses stack data structure.

Queue:

Queue is a linear data structure that follows a particular order in which the operations are performed. The order is First In First Out(FIFO) i.e. the data item stored first will be accessed first. In this, entering and retrieving data is not done from only one end. An example of a queue is any queue of consumers for a resource where the consumer that came first is served first. Different operations are performed on a Queue like Reversing a Queue (with or without using recursion), Reversing the first K elements of a Queue, etc. A few basic operations performed In Queue are enqueue, dequeue, front, rear, etc

The following are the primary operations of the Queue:

a. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.
b. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.
c. **Front:** Get the front item from the queue.
d. **Rear:** Get the last item from the queue.



Some Applications of Queues:

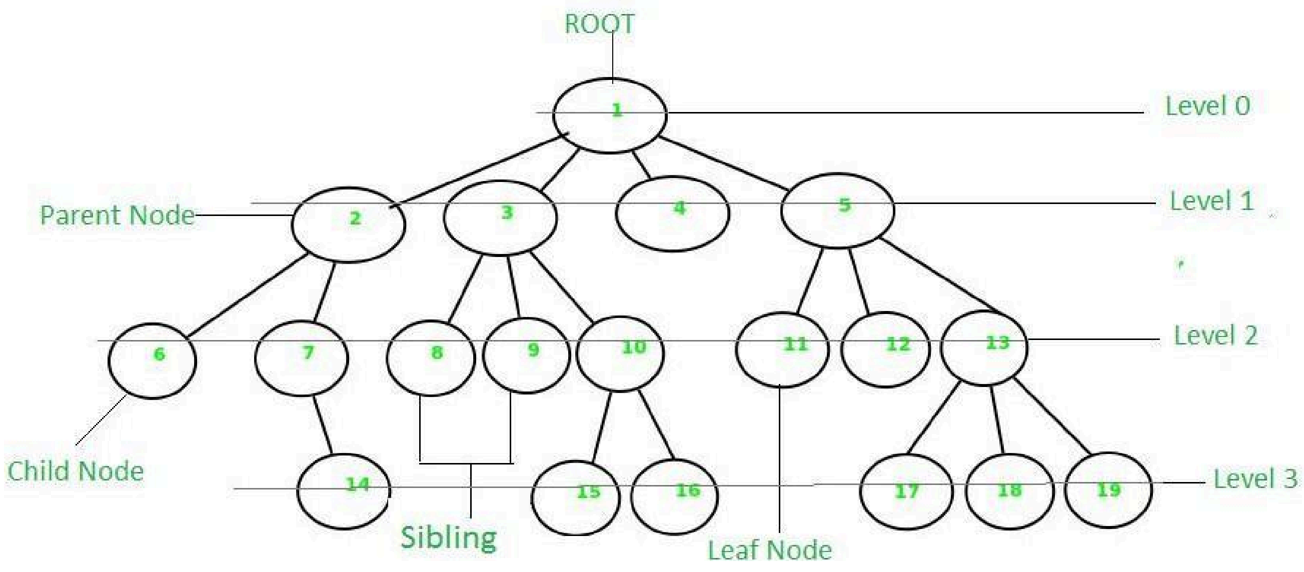
a. Queues are generally used in the breadth search operation in Graphs.
b. Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.
c. Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.
d. Priority Queues are utilized in file-downloading operations in a browser.
e. Queues are also used to transfer data between peripheral devices and the CPU.
f. Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

Tree:
A tree is a non-linear and hierarchical data structure where the elements are arranged in a tree-like structure. In a tree, the topmost node is called the root node. Each node contains some data, and data can be of any type. It consists of a central node, structural nodes, and

sub-nodes which are connected via edges. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure. A tree has various terminologies like Node, Root, Edge, Height of a tree, Degree of a tree, etc.

There are different types of Tree-like

a. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.
b. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.
c. **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.
d. **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.



Applications of Tree:

Different applications of Tree are as follows:

- Heap is a tree data structure that is implemented using arrays and used to implement priority queues.
- B-Tree and B+ Tree are used to implement indexing in databases.

- Syntax Tree helps in scanning, parsing, generation of code, and evaluation of arithmetic expressions in Compiler design.
- K-D Tree is a space partitioning tree used to organize points in K-dimensional space.
- Spanning trees are used in routers in computer networks

Real-Life Applications of Tree:

- In real life, tree data structure helps in Game Development.
- It also helps in indexing in databases.
- A Decision Tree is an efficient machine-learning tool, commonly used in decision analysis. It has a flowchart-like structure that helps to understand data.
- Domain Name Server also uses a tree data structure.
- The most common use case of a tree is any social networking site.

Binary Tree**:** Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures. A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

A Binary Tree is represented by a pointer to the topmost node in the tree. If the tree is empty, then the value of root is NULL. A Binary Tree node contains the following parts.

1. Data
2. Pointer to left child
3. Pointer to the right child

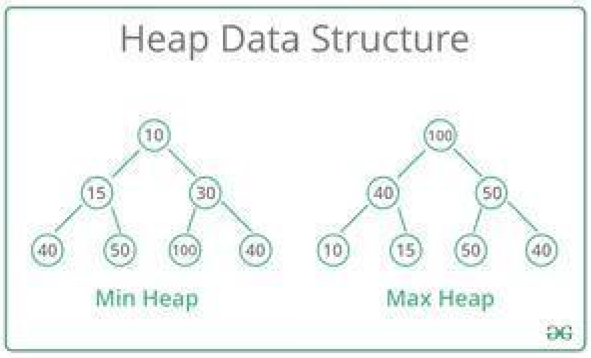
Binary Search Tree: A Binary Search Tree is a Binary Tree following the additional properties:
The left part of the root node contains keys less than the root node key.

- The right part of the root node contains keys greater than the root node key.
- There is no duplicate key present in the binary tree.

A Binary tree having the following properties is known as Binary search tree (BST).

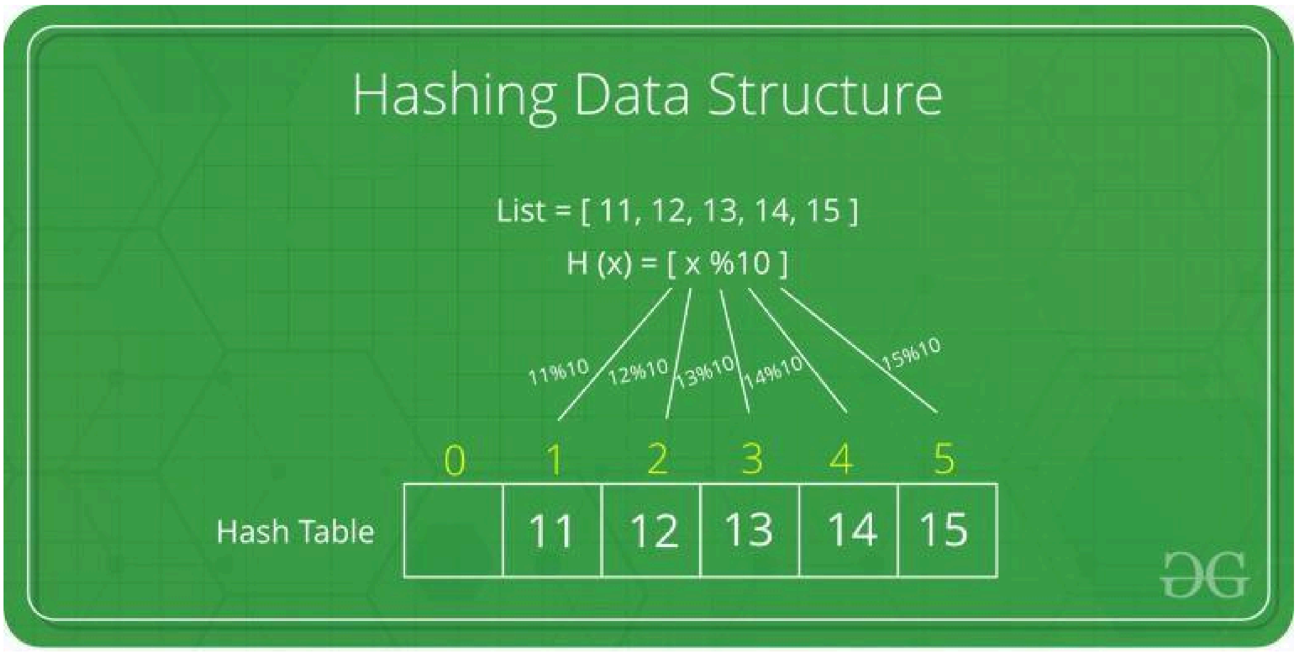
7. Heap**:** A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

- **Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- **Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.



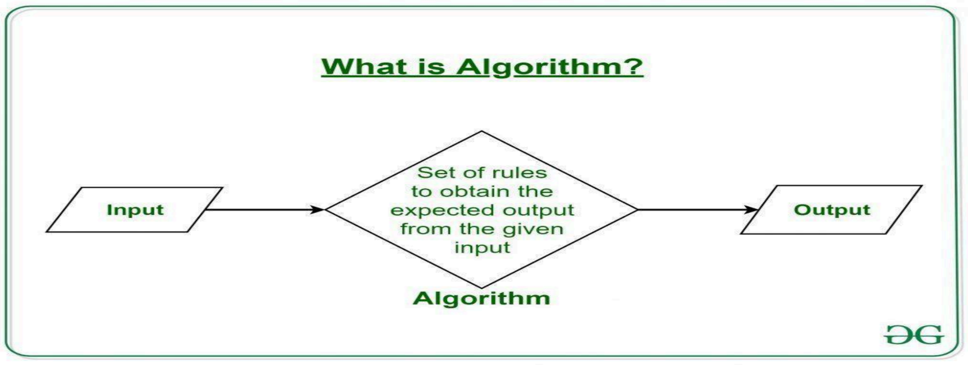
Hashing Data Structure: Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a **particular key for faster access of elements.** The efficiency of mapping depends on the efficiency of the hash function used.

Let **a hash function H(x) maps the value x at the index x%10 in an Array**. For example, if the list of values is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash table respectively.



<u>What is an Algorithm?</u>

The word Algorithm means "*A set of rules to be followed in calculations or other problem-solving operations*" Or "*A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations* ".


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Algorithm
Output

Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

1. Brute Force Algorithm: It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. Recursive Algorithm**:** A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Backtracking Algorithm**:** The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

4. Searching Algorithm**:** Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. Sorting Algorithm**:** Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

6. Hashing Algorithm**:** Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. Divide and Conquer Algorithm**:** This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

- Divide
- Solve
- Combine

8. Greedy Algorithm**:** In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

9. Dynamic Programming Algorithm**:** This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

10. Randomized Algorithm**:** In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
// Suppose we have to calculate the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum holds the sum of the n numbers
return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

- **Space complexity:** An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So,

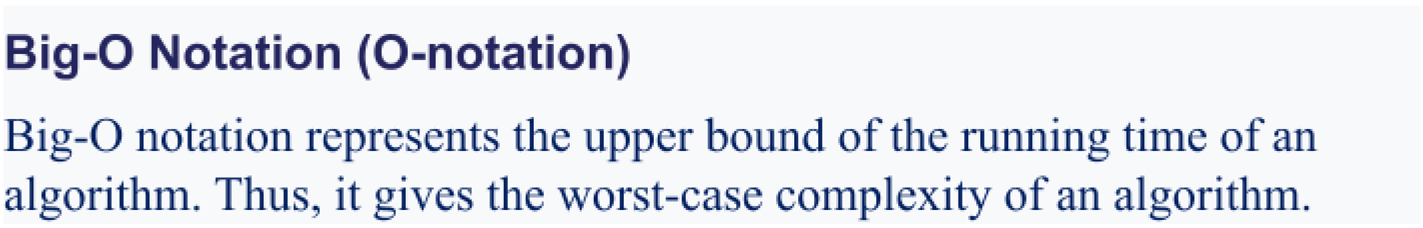
Space complexity = Auxiliary space + Input size

Asymptotic Notation and Analysis (Based on input size) in Complexity Analysis of Algorithms

Asymptotic Analysis is defined as the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, ***we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.***

Asymptotic notation is a way to describe the running time or space complexity of an algorithm based on the input size. It is commonly used in complexity analysis to describe how an algorithm performs as the size of the input grows.

The three most commonly used notations are Big O, Omega, and Theta.



Big-O Notation (O-notation)

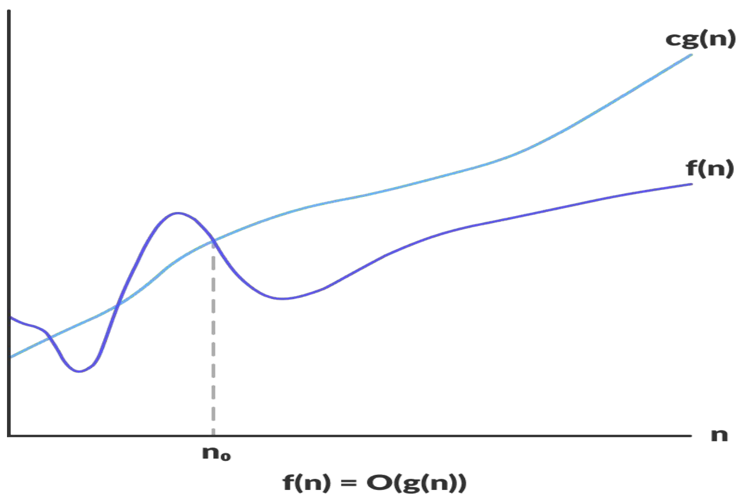
Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

It specifies the upper bound of a function.

The maximum time required by an algorithm or the worst-case time complexity.

It returns the highest possible output value(big-O) for a given input.

Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.



O(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

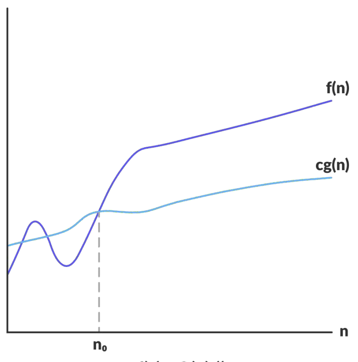
The above expression can be described as a function `f(n)` belongs to the set `O(g(n))` if there exists a positive `c` h that it lies betwee`0` and `cg(n)`, for sufficiently constant large n.

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.



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

Ω(g(n)) = { f(n): there exist positive constants c

and n0 such that 0 ≤ cg(n) ≤ f(n) for

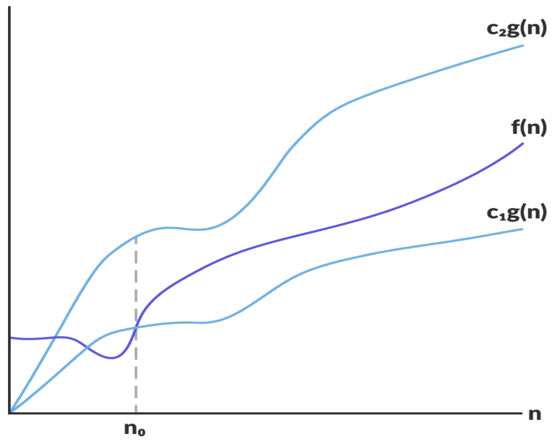
all n ≥ n0 }

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.



f(n) = Θ(g(n))

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all $n \ge n0$ }

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 c1 g(n) and c2 g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1 g(n) and c2 g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

For example, let us consider the search problem (searching a given item) in a sorted array.

The solution to above search problem includes:

- **Linear Search** (order of growth is linear)
- **Binary Search** (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,

- let us say:

- we run the Linear Search on a fast computer A and
- Binary Search on a slow computer B and
- pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.
- Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.
- For small values of input array size n, the fast computer may take less time.
- **But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine**.

Input Size	**Running time on A**	**Running time on B**
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

- The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.
- **So the machine-dependent constants can always be ignored after a certain value of input size.**

Running times for this example:

- Linear Search running time in seconds on A: 0.2 * n
- Binary Search running time in seconds on B: 1000*log(n)

Does Asymptotic Analysis always work?

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically the same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

Advantages:

1. Asymptotic analysis provides a high-level understanding of how an algorithm performs with respect to input size.
2. It is a useful tool for comparing the efficiency of different algorithms and selecting the best one for a specific problem.
3. It helps in predicting how an algorithm will perform on larger input sizes, which is essential for real-world applications.
4. Asymptotic analysis is relatively easy to perform and requires only basic mathematical skills.

Disadvantages:

1. Asymptotic analysis does not provide an accurate running time or space usage of an algorithm.
2. It assumes that the input size is the only factor that affects an algorithm's performance, which is not always the case in practice.
3. Asymptotic analysis can sometimes be misleading, as two algorithms with the same asymptotic complexity may have different actual running times or space usage.
4. It is not always straightforward to determine the best asymptotic complexity for an algorithm, as there may be trade-offs between time and space complexity

What is Step Count Method?

The step count method is one of the methods to analyze the Time complexity of an algorithm. In this method, we count the number of times each instruction is executed. Based on that we will calculate the Time Complexity.

The step Count method is also called as **Frequency Count method**. Let us discuss step count for different statements:

1. <u>Comments:</u>

- Comments are used for giving extra meaning to the program. They are not executed during the execution. Comments are ignored during execution.

- Therefore the number of times that a comment executes is **0**.

2. Conditional statements:

Conditional statements check the condition and if the condition is correct then the conditional subpart will be executed. So the execution of conditional statements happens only once. The compiler will execute the conditional statements to check whether the condition is correct or not so it will be executed one time.

- In **if-else** statements the if statement is executed one time but the else statement will be executed **zero** or **one** time because if the "if" statement is executed then the else statement will not execute.
- In **switch** case statements the starting switch(condition) statement will be executed one time but the inner case statements will execute if none of the previous case statements are executed.
- In **nested if and if else** ladder statements also the initial if statement is executed at least once but inner statements will be executed based on the previous statements' execution.

3. Loop statements:

Loop statements are iterative statements. They are executed one or more times based on a given condition.

- A typical **for(i = 0; i ≤ n; i++)** statement will be executed "**n+1**" times for the first n times the condition is satisfied and the inner loop will be executed and for the (n+1)th time the condition is failed and the loop terminates.
- **While:** The statement is executed until the given condition is satisfied.
- **Do while:** The statement will repeat until the given condition is satisfied. The do-while statement will execute at least once because for the first time it will not check the condition.

4. Functions:

Functions are executed based on the number of times they get called. If they get called n times they will be executed **n** times. If they are not called at least once then they will not be executed. Other statements like **BEGIN**, **END** and **goto** statements will be executed one time.

Illustration of Step Count Method:

Analysis of Linear Search algorithm

```
Linearsearch(arr, n, key)
{
  i = 0;
  for(i = 0; i < n; i++)
  {
    if(arr[i] == key)
    {
        printf("Found");
    }
}
```

Where,

- **i = 0**, is an initialization statement and takes O(1) times.
- **for(i = 0;i < n ; i++)**, is a loop and it takes O(n+1) times .
- **if(arr[i] == key)**, is a conditional statement and takes O(1) times.
- **printf("Found")**, is a function and that takes O(0)/O(1) times.

Therefore Total Number of times it is executed is **n + 4** times. As we ignore lower exponents in time complexity total time became **O(n)**.

Time complexity: O(n).
Auxiliary Space: O(1)

2]Linear Search in Matrix

Searching for an element in a matrix

```
AlgoMatrixsearch(mat[][],key)
{
    //numberofrow
    s;    r:=len(mat)
    //numberofcolu
    mns;
    c:=len(mat[0])
    for(i=0;i<r;i++)
    {
        for(j=0; j<c; j++)
        {
            if(mat[i][j]==key)
            {
                printf("Elementfound");
            }
        }
    }
}
```

Where,

- **r = len(mat),** takes O(1) times.
- **c = len(mat[0]),** takes O(1) times

- **for(i = 0; i < r; i++),** takes O(r + 1) times

- **for(j = 0; j < c; j++),** takes O((c + 1)) for each time the outer loop is satisfied. So total r times the loop is executed.
- **if(mat[i][j] == key),** takes O(1) times

- **printf("Element found"),** takes O(0)/O(1) times.

Therefore Total Number of times it is executed is **(1 + 1 + (r + 1) + (r) * (c + 1) + 1)** times. As we ignore the lower exponents, total complexity became **O(r * (c + 1)).**

the mat is an array so it takes n*n words , k, c, r, i, j will take 1 word. Time Complexity: O(n2).

Auxiliary Space: O(n2)

In this way, we calculate the time complexity by counting the number of times each line executes.