CHAPTER ONE

Introduction and Elementary Data Structures

Introduction

❖ What is Algorithm?

- It is a clearly specified set of simple instructions to be followed to solve a problem.
  - Takes a set of values, as input and
  - produces a value, or set of values, as output
- May be specified
  - In English
  - As a computer program
  - As a pseudo-code
- It is the best way to represent the solution of a particular problem in a very simple and efficient way.

Introduction

- **Algorithm:** **the sequence of computational steps to solve a problem.**
- **Data Structure:** **the way data are organized in a computer's memory.**

- **Program = Data structures + Algorithm.**

- **A program is a set of instruction which is written to solve a problem.**
- **A program is the expression of an algorithm in a programming language.**

Algorithm Design

- The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.
- To solve a problem, different approaches can be followed.
  - Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient.
  - However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously.
  - If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Dataflow of an Algorithm

- **Problem:** **A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions.**
- **Algorithm:** **An algorithm will be designed for a problem which is a step by step procedure.**
- **Input:** **After designing an algorithm, the required and the desired inputs are provided to the algorithm.**
- **Processing unit:** **The input will be given to the processing unit, and the processing unit will produce the desired output.**
- **Output:** **The output is the outcome or the result of the program.**

Why do we need Algorithms?

- **We need algorithms because of the following reasons:**
  - **Scalability:**
    - **It helps us to understand the scalability.**
    - **When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.**
  - **Performance:**
    - **The real-world is not easily broken down into smaller steps.**
    - **If the problem can be easily broken into smaller steps means that the problem is feasible.**

Characteristics of an Algorithm

1. Finiteness:

- **Algorithm must complete after a finite number of steps.**
- **Algorithm should have a finite number of steps.**

Example of Finite Steps:

```
int i=0;
while(i<10)
{
    cout<<“Hello!”;
    i++;
}
```

Example of Infinite Steps

```
while(true)
{
    cout<<“Hello!”;
}
```

Characteristics of an Algorithm

2. Definiteness (Absence of ambiguity):

- **An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.**
- **Each step must be clearly defined, having one and only one interpretation.**

3. Sequential:

- **Each step must have a uniquely defined preceding and succeeding step.**
- **The first step and last step must be clearly noted.**

Characteristics of an Algorithm

4. Feasibility:

- **It should be feasible with the available resources.**
- **It must be possible to perform each instruction.**
- **Each instruction should have a possibility to be executed.**

Example:

```
a)  for(int i=0; i<0; i++)
     {
       cout<< i;  // there is no possibility that this statement to be executed.
      }
b)    if(5>7)
       {
         cout<<"Hello!"; // not executed.
       }
```

Characteristics of an Algorithm

5. Correctness:

- It must compute correct answer for all possible legal inputs.
- The output should be as expected and required, and correct.

6. Language Independence

- It must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

7. Completeness

- It must solve the problem completely.

Characteristics of an Algorithm

8. Effectiveness:

- **It should be effective as each instruction in an algorithm affects the overall process.**
  - **i.e. doing the right thing.**
- **It should return the correct result all the time for all of the possible cases.**

9. Efficiency:

- **It must solve with the least amount of computational resources such as time and space.**
- **Producing an output as per the requirement within the given resources (constraints).**

Characteristics of an Algorithm

- **Example:** **Write a program that takes two numbers and displays the sum of the two.**

Program A

```
cin>>a;
cin>>b;
sum= a+b;
cout<<sum;
```

Program B

```
cin>>a;
cin>>b;
a= a+b;
cout<<a;
```

Program C

```
cin>>a;
cin>>b;
cout<<a+b;
```

(the most efficient)

❖ **All are effective but with different efficiencies.**

Characteristics of an Algorithm

10. Input/output

- There must be a specified number of input values, and one or more result values.
- Zero or more inputs and one or more outputs.

11. Precision

- The result should always be the same if the algorithm is given identical input.

12. Simplicity

- A good general rule is that each step should carry out one logical step.
- What is simple to one processor may not be simple to another.

Algorithm Analysis

- **Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.**
  - **It is the determination of the amount of time and space resources required to execute it.**
  - **It is a process of predicting the resource requirement of algorithms in a given environment.**
- **In order to solve a problem, there are many possible algorithms.**
- **One must be able to choose the best algorithm for the problem at hand using some scientific method.**

Algorithm Analysis

- **Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:**
- **A Priori Analysis: This is a theoretical analysis of an algorithm.**
  - **Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.**
- **A Posterior Analysis: This is an empirical analysis of an algorithm.**
  - **The selected algorithm is implemented using programming language.**
  - **This is then executed on target computer machine.**
  - **In this analysis, actual statistics like running time and space required, are collected.**

Algorithm Analysis

- **Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation).**
- **However, the main concern of analysis of algorithms is the required time or performance.**
- **Generally, we perform the following types of analysis:**
  - **Worst-case: Maximum time required for program execution.**
    - **the maximum number of steps taken on any instance of size n.**
  - **Best-case: Minimum time required for program execution.**
    - **the minimum number of steps taken on any instance of size n.**
  - **Average case: Average time required for program execution.**
    - **an average number of steps taken on any instance of size n.**

Complexity Analysis

- In designing of Algorithm, complexity analysis of an algorithm is an essential aspect.
- Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.
- The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.
- Complexity of an algorithm is analyzed in two perspectives: Time and Space.

Complexity Analysis

Time complexity

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

- For example: addition of two n-bit integers takes n steps.
  - the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits.
  - here, we observe that T(n) grows linearly as the input size increases.

Complexity Analysis

Space complexity

- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.
- The space required by an algorithm is equal to the sum of the following two components:
  - A fixed part is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
  - A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.
- Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Asymptotic Notations

- **Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes.**
- **Asymptotic Notations are used to write fastest and slowest possible running time for an algorithm.**
  - **These are also referred to as 'best case' and 'worst case' scenarios respectively.**
- **In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n).**
- **These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms.**

Asymptotic Notations

- **Asymptotic Analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound!.**
  - **i.e. concerned with how the running time of an algorithm increases when the size of the input increases.**
- **Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.**
- **Time function of an algorithm is represented by T(n), where n is the input size.**

Asymptotic Notations

- **Why is Asymptotic Notations Important?**
  1. **They give simple characteristics of an algorithm's efficiency.**
  2. **They allow the comparisons of the performances of various algorithms.**

- **Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.**
  - **O − Big Oh**
  - **Ω − Big Omega**
  - **θ − Big Theta**

Asymptotic Notations: O – Big Oh

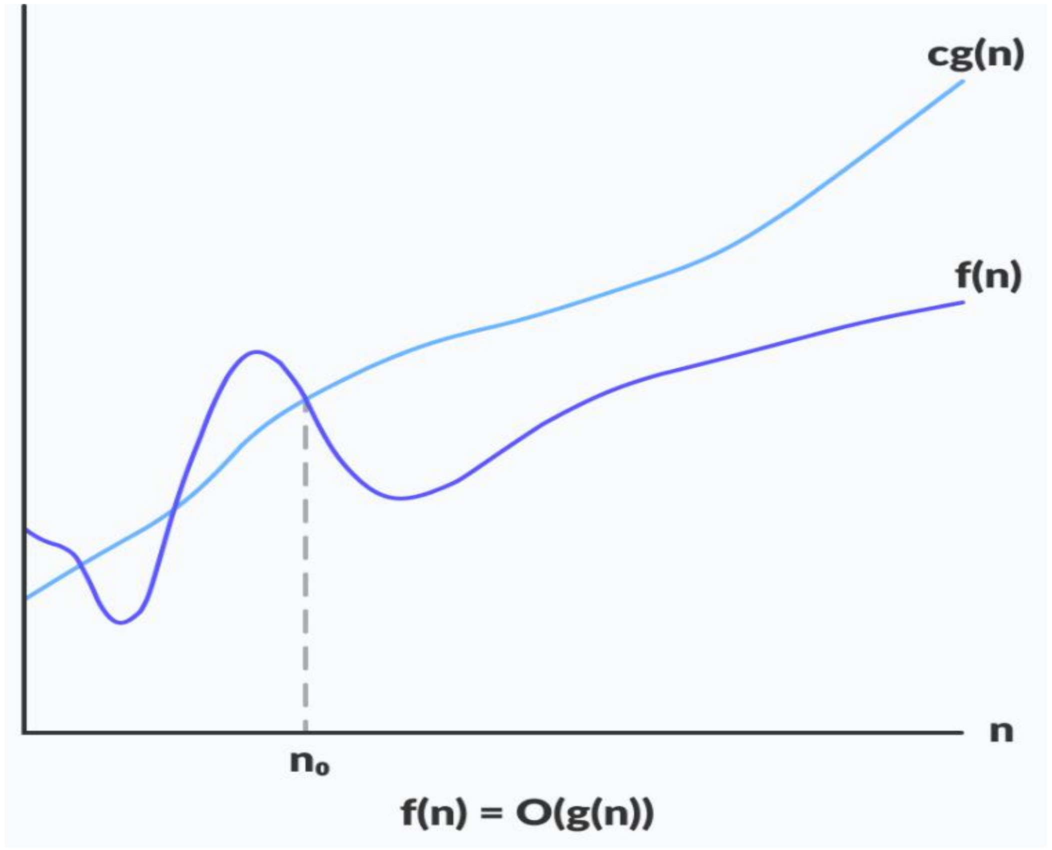
- **The notation O(n) is the formal way to express the upper bound of an algorithm's running time.**
- **It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.**

  - **If f(n) and g(n) are the two functions defined for positive integers,**
  - **then f(n) = O(g(n)) as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and n_o such that:**

    $f(n) \leq c.g(n)$ for all $n \geq n_o$
  - **The growth rate of f(n) is less than or equal to the growth rate of g(n).**

Asymptotic Notations: O – Big Oh

- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n).



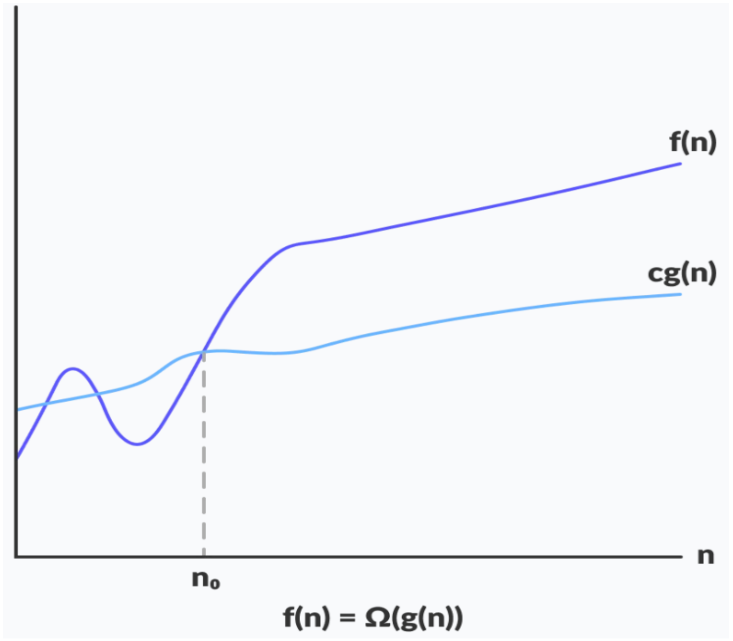
Asymptotic Notations: Ω – Big Omega

- **The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.**
- **It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.**
  - **If f(n) and g(n) are the two functions defined for positive integers,**
  - **then $f(n) = \Omega(g(n))$ as f(n) is Omega of g(n) or f(n) is on the order of g(n)) if there exists constants c and n_0 such that:**

    $f(n) >= c.g(n)$ for all $n \geq n_0$ and $c>0$
  - **The growth rate of f(n) is greater than or equal to the growth rate of g(n).**

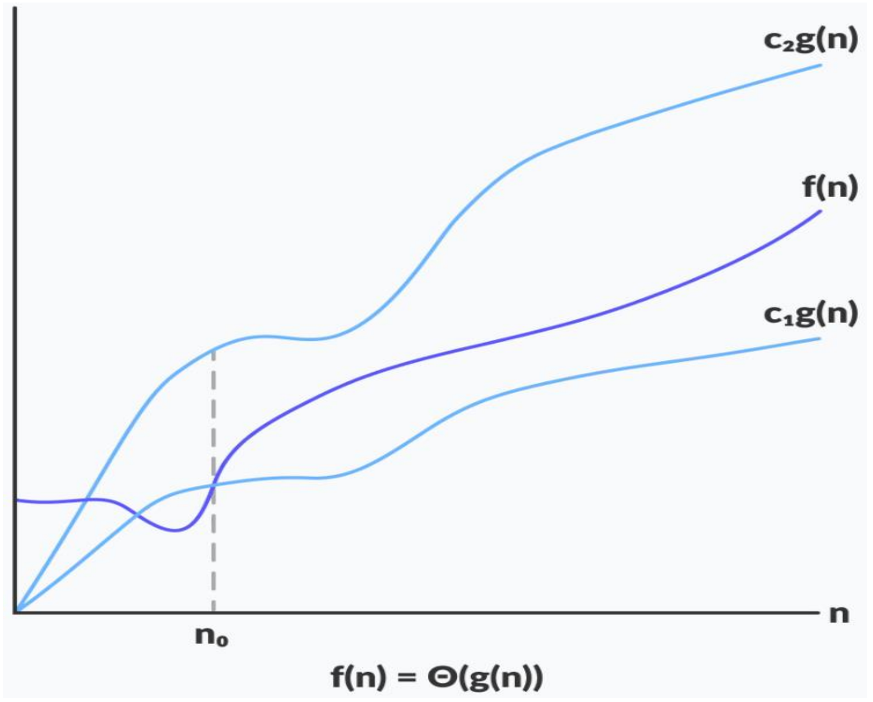
Asymptotic Notations: Ω – Big Omega

- **Therefore, this notation gives the fastest running time.**
  - **But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.**



Asymptotic Notations: θ – Big Theta

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
  - The theta notation mainly describes the average case scenarios.
- For some algorithms (but not all), the lower and upper bounds on the rate of growth will be the same.
  - f(n) = θ(g(n))
- The growth rate of f(n) is the same as the growth rate of g(n).



Asymptotic Notations: θ – Big Theta

- **Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:**

  f(n)= θg(n)

- **The above condition is satisfied only if when**

  c1.g(n)<=f(n)<=c2.g(n)

- **where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between.**
- **The condition f(n)= θg(n) will be true if and only *if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n)*.**

Class Activity

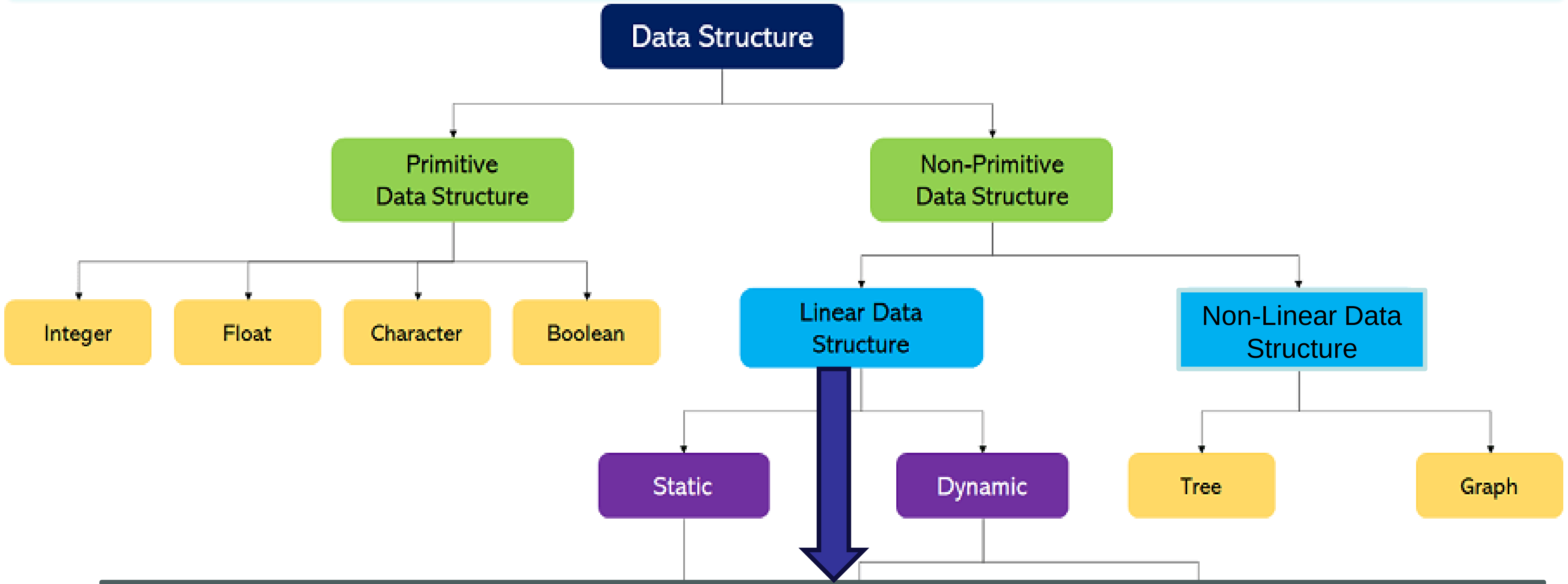
- **If $f(n) = 2n+3$ and $g(n) = n$ then show that**
  1. **$f(n) = O(g(n))$**
  2. **$f(n) = \Omega(g(n))$**
  3. **$f(n) = \theta(g(n))$**

Elementary Data Structures

Elementary Data Structures

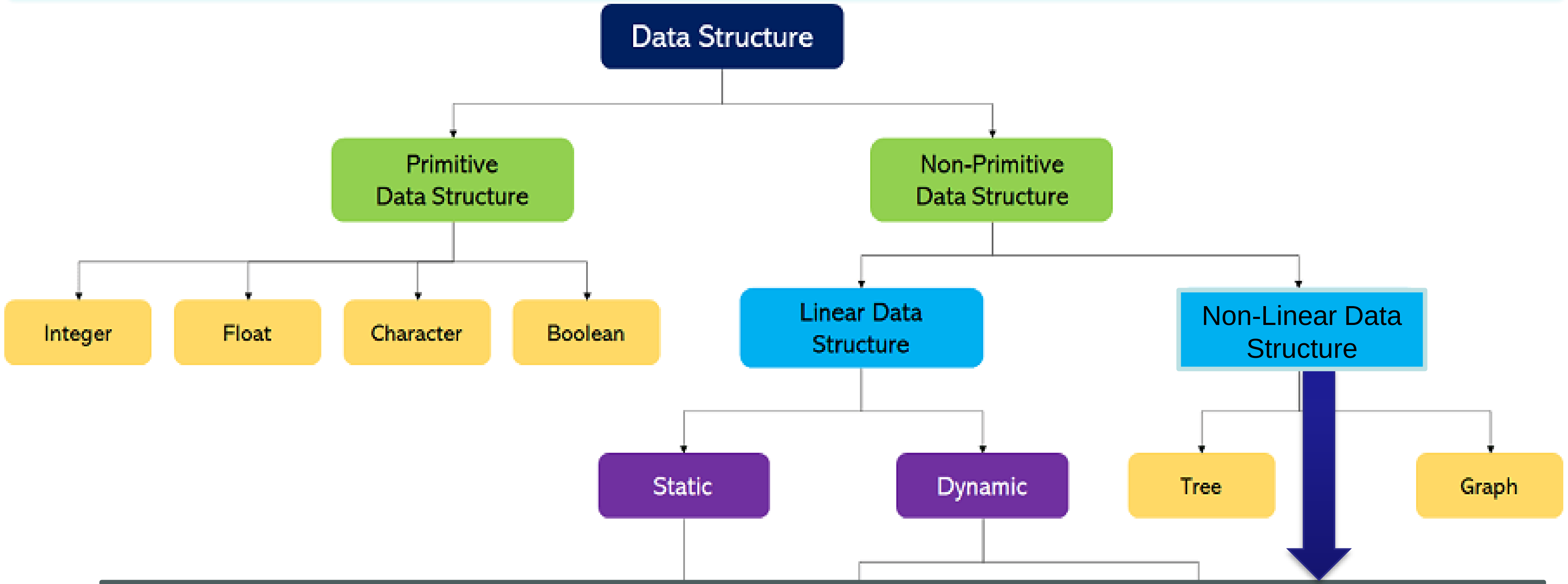
- **A data structure is a way of organizing and storing data in a computer program so that it can be accessed and used efficiently.**
- **Data structures can be categorized into two types:**
  1. **Primitive data structures: are the most basic data structures available in a programming language, such as integers, floating-point numbers, characters, and Booleans.**
  2. **Non-primitive data structures: are complex data structures that are built using primitive data types, such as arrays, linked lists, stacks, queues, trees, and graphs.**

Classification of Data Structure



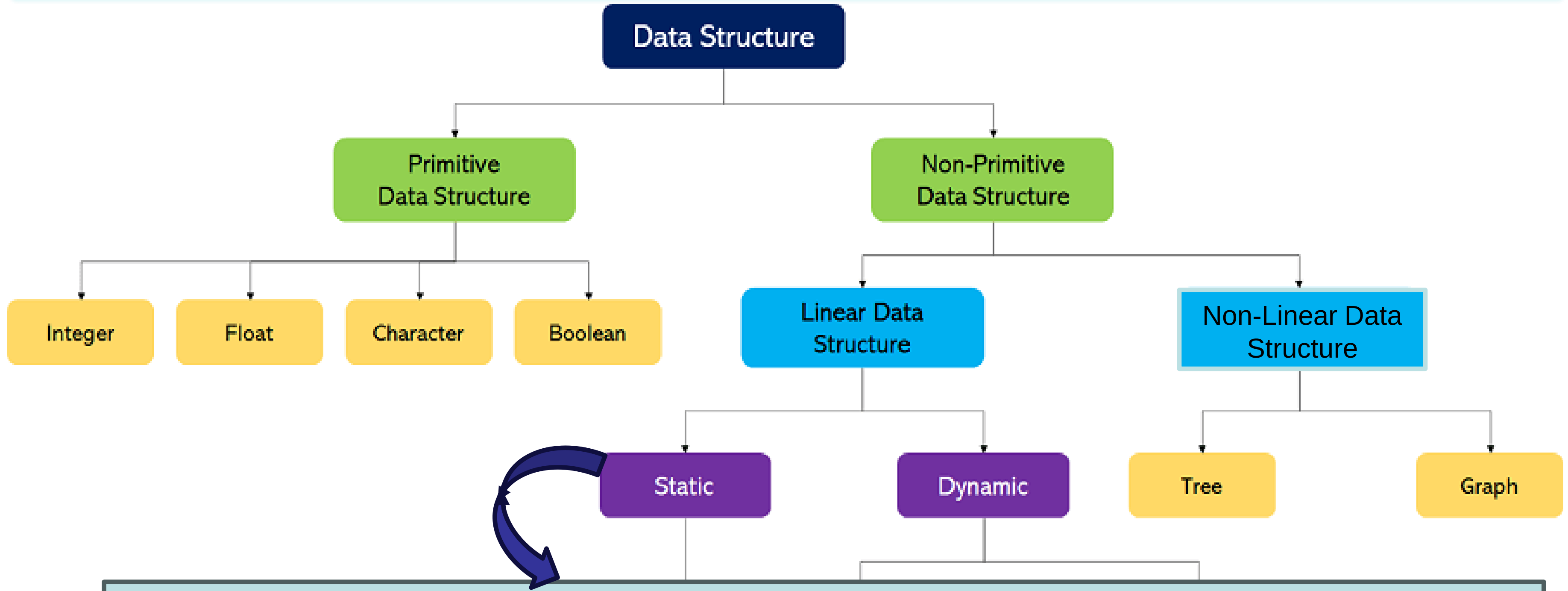
- The data items are arranged and stored in sequential order, one after the other.
- Applications: mainly in application software development

Classification of Data Structure



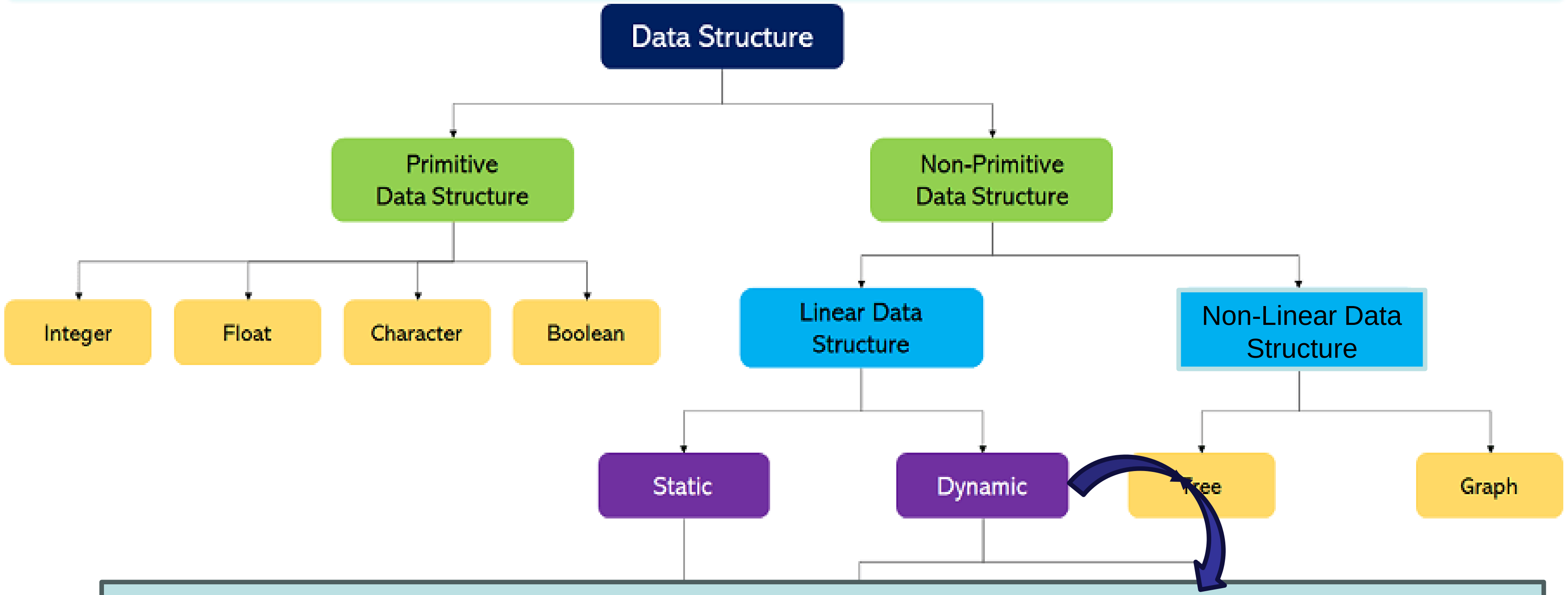
- The data items are arranged and stored in the form of hierarchy, where one element will be connected to one or more elements.
- Application: in AI and Image Processing

Classification of Data Structure



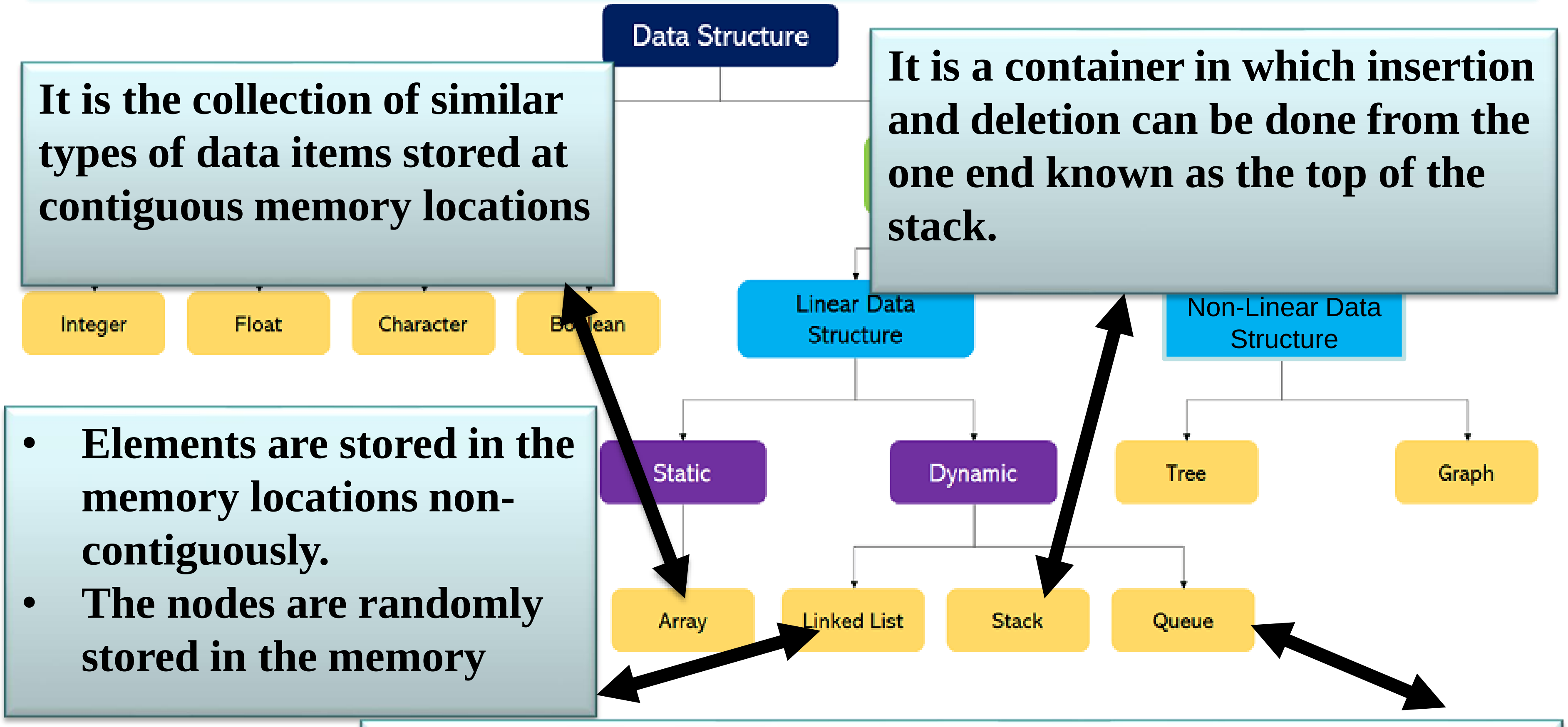
- Static data structure has a fixed memory size and the memory allocation is not scalable.
- It is easier to access the elements in a static data structure.

Classification of Data Structure



- The size is **not fixed** and the memory allocation can be done **dynamically when required**.
- It can be **randomly updated during the runtime** which may be considered efficient concerning the memory (space) complexity of the code.

Classification of Data Structure



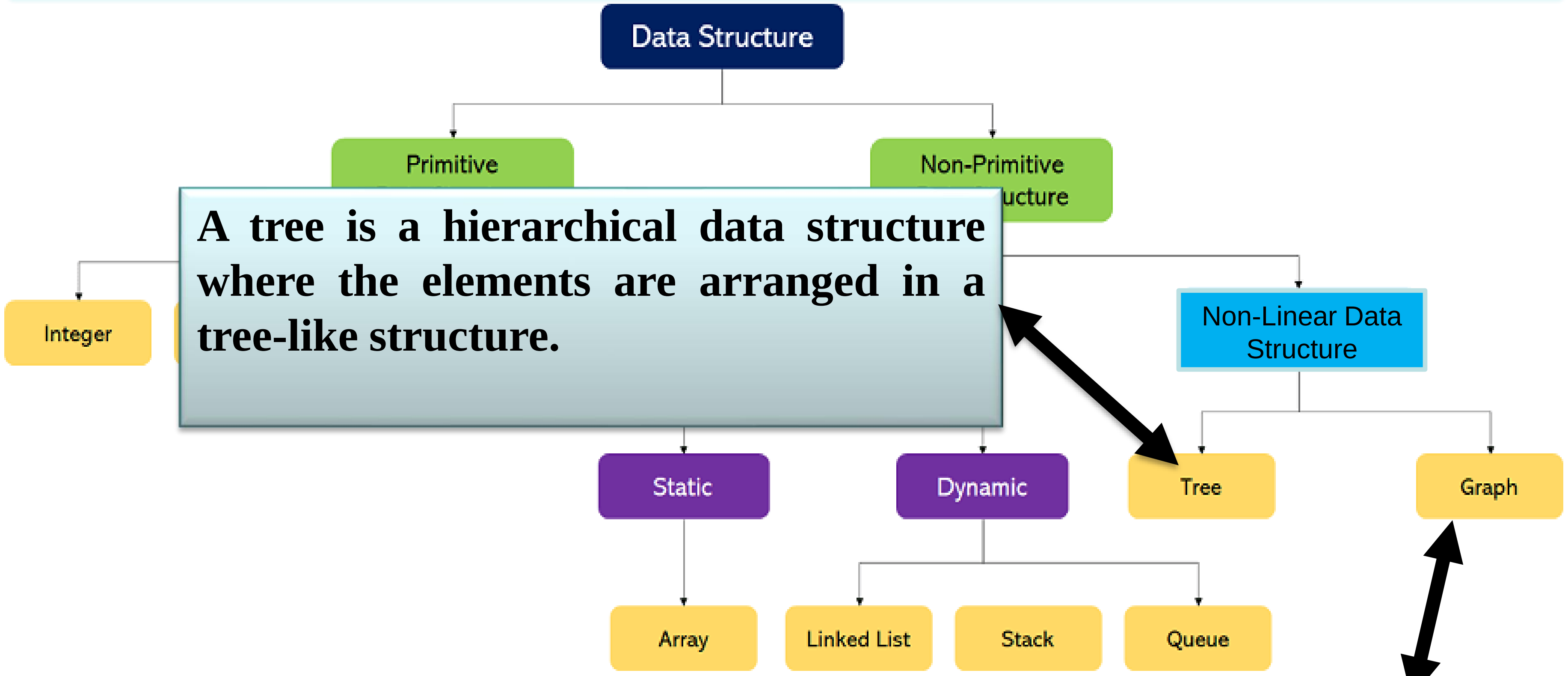
It is the collection of similar types of data items stored at contiguous memory locations

It is a container in which insertion and deletion can be done from the one end known as the top of the stack.

- Elements are stored in the memory locations non-contiguously.
- The nodes are randomly stored in the memory

It is defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

Classification of Data Structure



A tree is a hierarchical data structure where the elements are arranged in a tree-like structure.

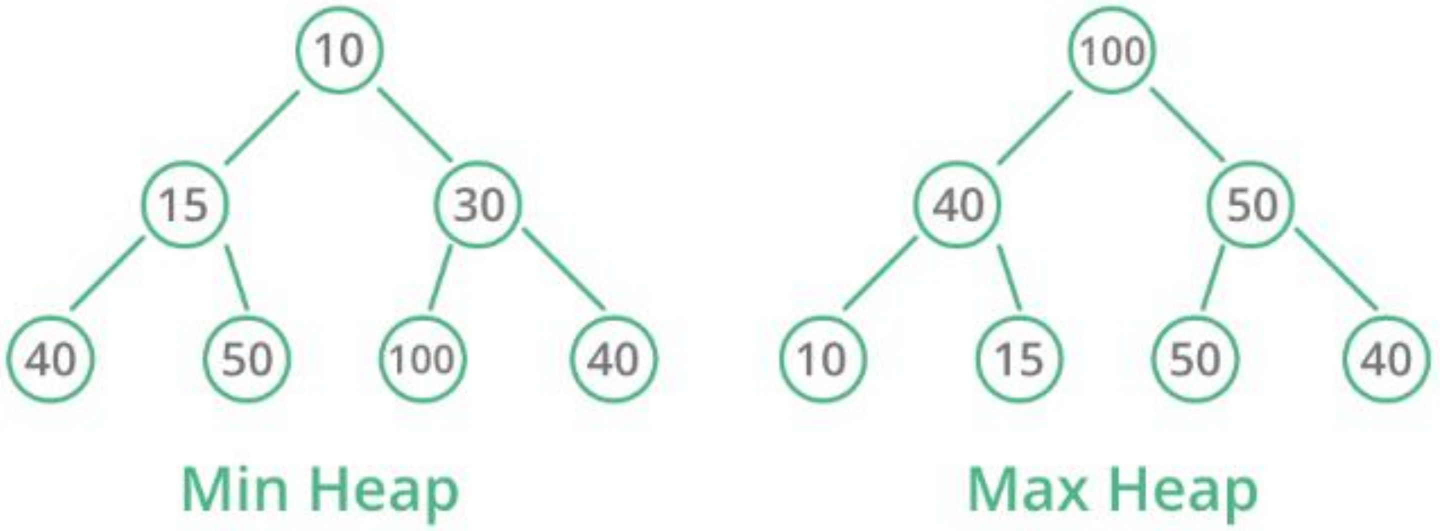
A graph consists of vertices (or nodes) and edges. It consists of a finite set of vertices and set of edges that connect a pair of nodes.

Elementary Data Structures

- **The choice of data structure for a particular task depends on:**
  - **the type and amount of data to be processed,**
  - **the operations that need to be performed on the data, and**
  - **the efficiency requirements of the program.**
- **Efficient use of data structures can greatly improve the performance of a program, making it faster and more memory-efficient.**
- **The idea is to reduce the space and time complexities of different tasks.**

Heap Data Structure

- A Heap is a complete binary tree data structure that satisfies the heap property.
  - *Heap property ensures that the minimum (or maximum) element is always at the root of the tree according to the heap type.*



Heap Data Structure

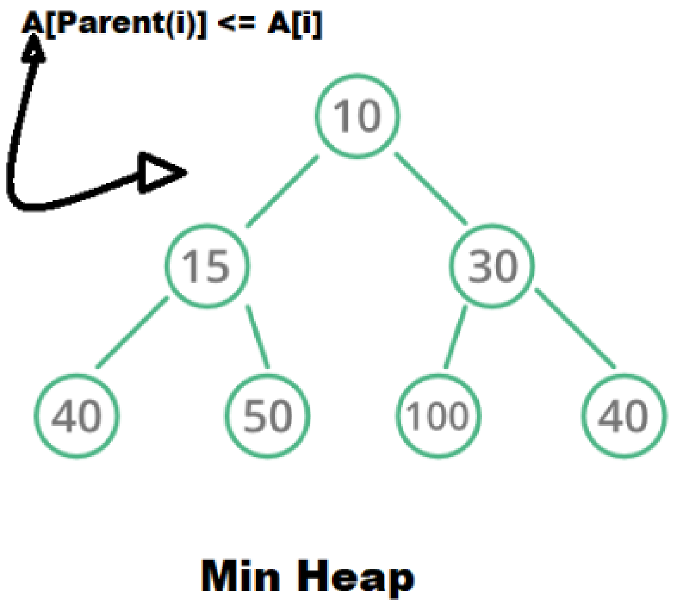
- A binary tree is a tree in which the node can have at most two children.

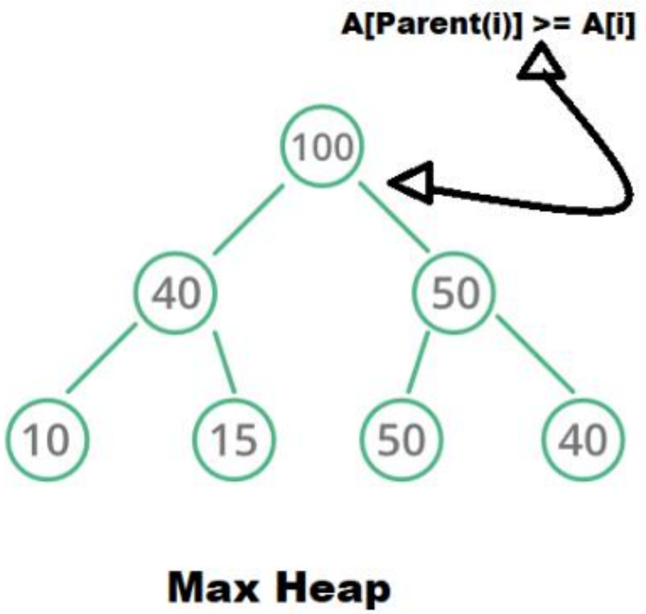
- A complete binary tree is a binary tree in which all the nodes except the leaf node should be completely filled, and all the nodes should be left-justified.

- A heap is defined by two properties.
  - First, it is a complete binary tree, so heaps are nearly always implemented using the array representation.
  - Second, the values stored in a heap are partially ordered.
    - This means that there is a relationship between the value stored at any node and the values of its children.

Heap Data Structure

- There are two variants of the heap, depending on the definition of this relationship
  1. Min-Heap
  2. Max-Heap





Min Heap

- In this heap, the value of the root node must be the smallest among all its child nodes and the same thing must be done for its left and right sub-tree also.
  - The value of the parent node should be less than or equal to either of its children.
  - Mathematically, it can be defined as: A[Parent(i)] <= A[i]
- The total number of comparisons required in the min heap is according to the height of the tree.
- The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Max Heap

- In this heap, the value of the root node must be the greatest among all its child nodes and the same thing must be done for its left and right sub-tree also.
  - The value of the parent node is greater than or equal to its children.
  - Mathematically, it can be defined as: A[Parent(i)] >= A[i]
- The total number of comparisons required in the max heap is according to the height of the tree.
- The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Heap Operation

- **Some of the important operations performed on a heap are:**
  - **Heapify: a process of creating a heap from an array.**
  - **Insertion: process to insert an element in existing heap.**
  - **Deletion: deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element.**
  - **Peek: to check or find the most prior element in the heap, (max or min element for max and min heap respectively).**

Heap Operation: Heapify

- It is the process of creating a heap data structure from a binary tree.
- It is used to create a Min-Heap or a Max-Heap.
- It is the process to rearrange the elements to maintain the property of heap data structure.
- It is done when a certain node creates an imbalance in the heap due to some operations on that node.
- It takes O(logN) to balance the tree.

Heap Operation: Heapify (Algorithm)

```
Heapify(array, size, i)
  set i as largest
  leftChild = 2i + 1
  rightChild = 2i + 2

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild > array[largest]
    set rightChildIndex as largest

  swap array[i] and array[largest]
```

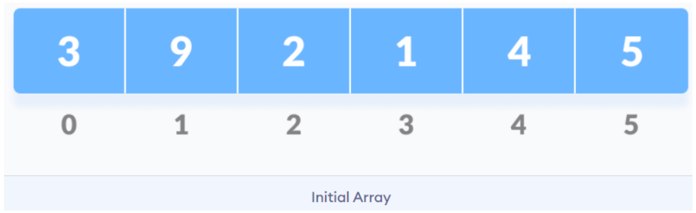
```
MaxHeap(array, size)
  loop from the first index of non-leaf
  node down to zero
  call heapify ()
```

For Min-Heap, both leftChild and rightChild must be larger than the parent for all nodes.

Heap Operation: Heapify (Example)

- **Heapify for Max-Heap**

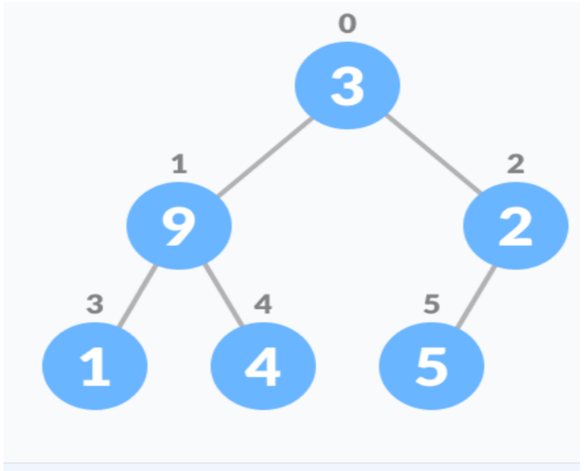
1. **Let the input array be:**



3	9	2	1	4	5
0	1	2	3	4	5

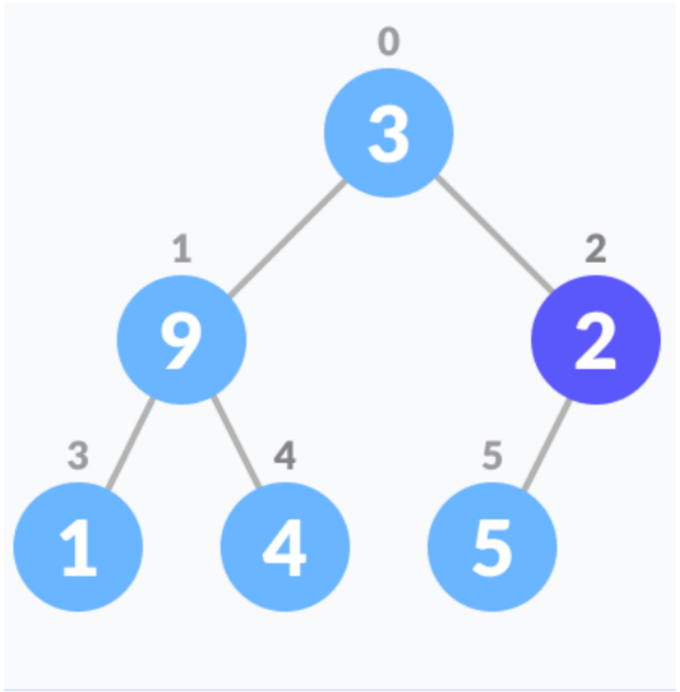
Initial Array

2. **Create a complete binary tree from the array.**



Complete binary tree

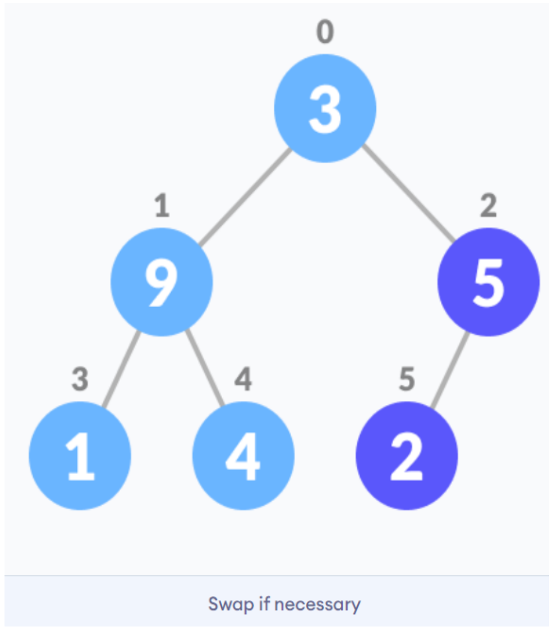
3. **Start from the first index of non-leaf node whose index is given by n/2 – 1.**



Start from the first on leaf node

Heap Operation: Heapify (Example)

4. Set current element i as largest.
5. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.
   - If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.
   - If rightChild is greater than element in largest, set rightChildIndex as largest.
6. Swap largest with currentElement
7. Repeat steps 3-7 until the subtrees are also heapified.



Heap Operation: Insertion

- Insert a new element to the existing heap.
- This operation takes O(logN) time.

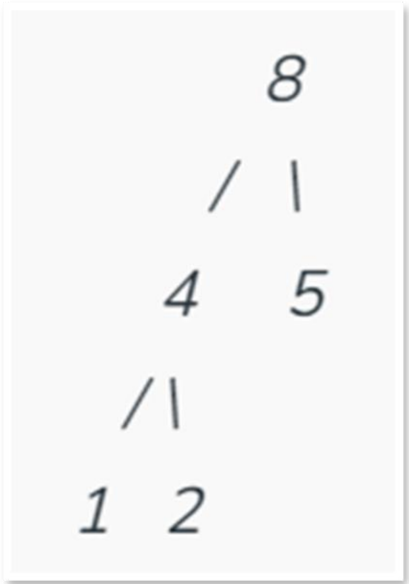
Algorithm

```
If there is no node
  create a newNode
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

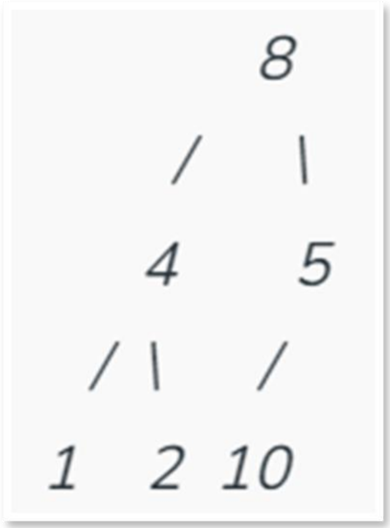
Heap Operation: Insertion (Example)

1. Assume initially heap (taking max-heap) is as follows:



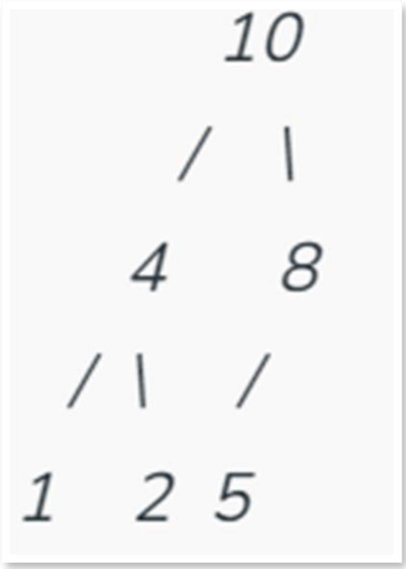
```
      8
     / \
    4   5
   / \
  1   2
```

2. Now if we insert 10 into the heap:



```
        8
       / \
      4   5
     / \  /
    1  2 10
```

3. After heapify operation final heap will be look like this:



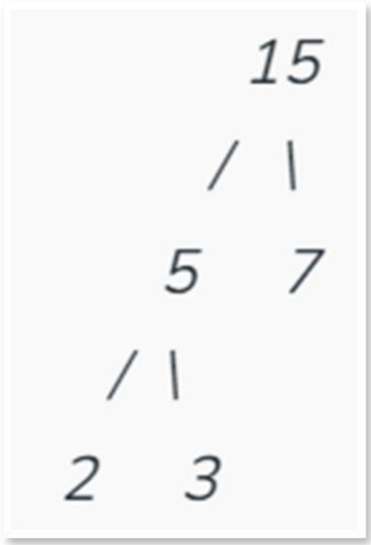
```
        10
       /  \
      4    8
     / \  /
    1  2 5
```

Heap Operation: Deletion

- Deleting the top element of the heap or the highest priority element.
- If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.
- The time complexity for this operation is O(logN).

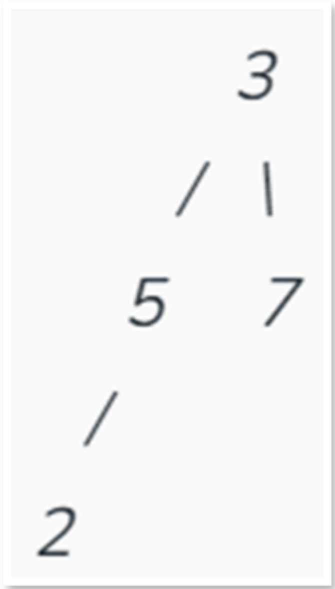
Heap Operation: Deletion (Example)

1. **Assume initially heap(taking max-heap) is as follows:**



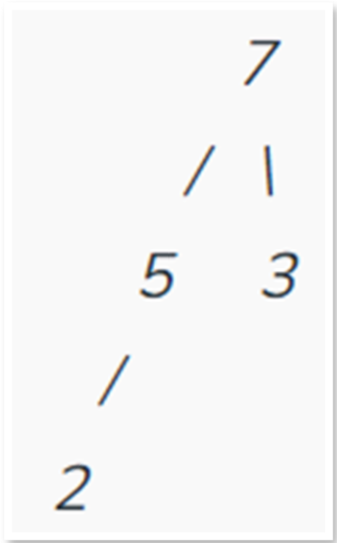
```
        15
        / \
      5    7
     / \
    2   3
```

2. **Now if we delete 15 into the heap it will be replaced by leaf node of the tree for temporary.**



```
        3
       / \
      5    7
     /
    2
```

3. **After heapify operation final heap will be look like this:**



```
        7
       / \
      5    3
     /
    2
```

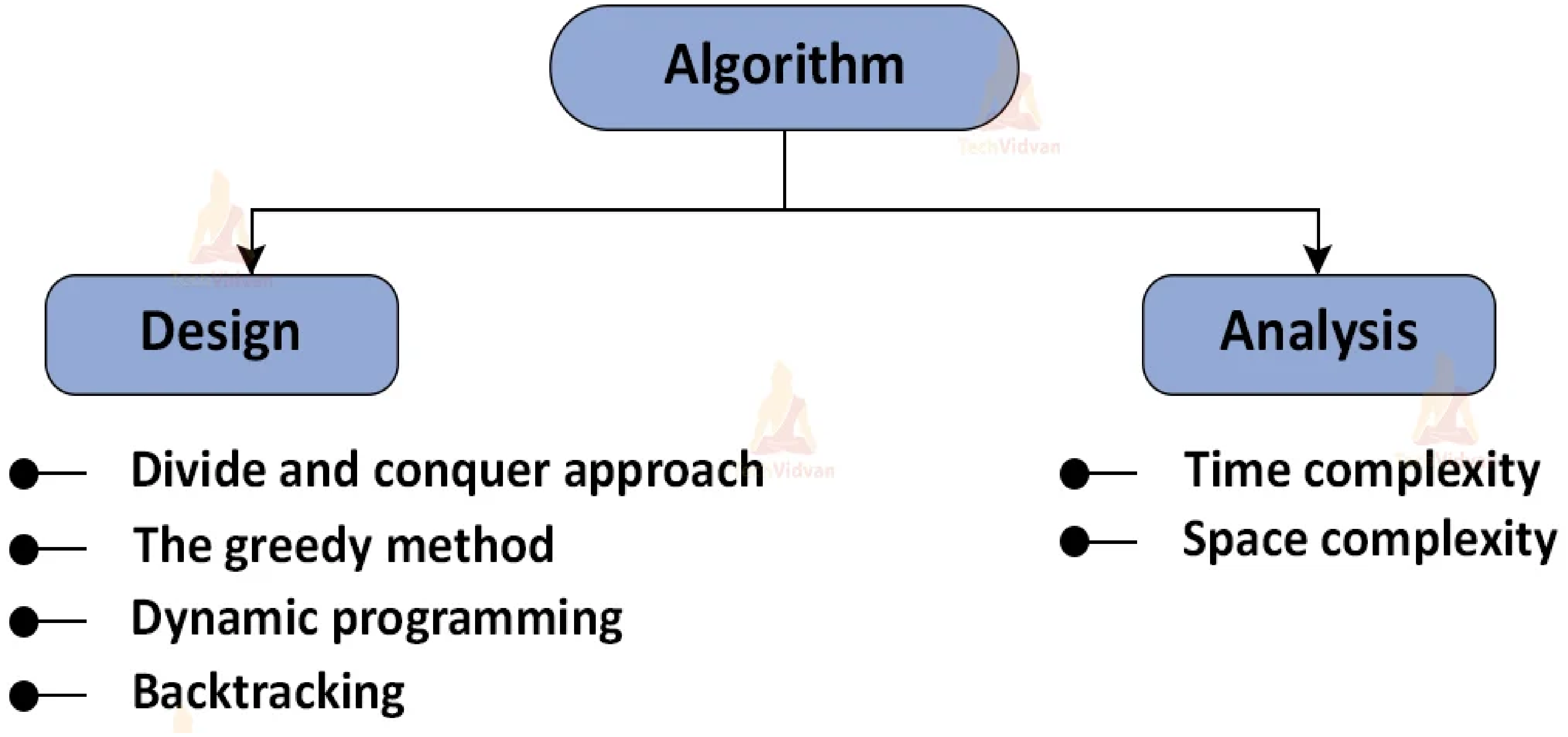
Heap Data structure: Operation

- **Peek (getMax or getMin):**
  - Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.
  - It takes O(1) time.

- **Extract-Max/Min:**
  - Extract-Max returns the node with maximum value after removing it from a Max Heap.
  - Extract-Min returns the node with minimum after removing it from Min Heap.

Heap Data structure: Application

- Priority queues
  - It is commonly used to implement priority queues, where elements are stored in a heap and ordered based on their priority.
- Heapsort algorithm
  - It is the basis for the heap sort algorithm, which is an efficient sorting algorithm with the worst-case time complexity of O(nlogn).
- Memory management
  - It is used in memory management systems to allocate and deallocate memory dynamically.
- Graph algorithms
  - It is used in various graph algorithms, including Dijkstra’s algorithm.
- Job scheduling
  - It is used in job scheduling algorithms, where tasks are scheduled based on their priority or deadline.


Algorithm
Design
Analysis
Divide and conquer approach
The greedy method
Dynamic programming
Backtracking
Time complexity
Space complexity

Chapter 2

Divide and Conquer Approach