

**PRECAT FAST TRACK BATCH  
OM22**

# Data Structure

**SunBeam Institute of  
Information & Technology,  
Hinjawadi, Pune & Karad.**

**SACHIN PAWAR**



# Data Structures: Introduction

---

## # PreCAT Scope : Data Structure

- In Section B, 7 Questions are reserved for this subject and mostly all questions are pseudocode oriented (i.e. theory based/concepts).
- In this course the main focus is on **basic data structure's algorithms** and **introduction to an advanced data structures** to build a base which is required to learn and implement advanced data structures and algorithms in CDAC courses.



# Data Structures: Introduction

---

## + Introduction

- Data structure
- Algorithm and analysis of an algorithm

## + Array

- Concept & definition
- **Searching Algorithms:**
  1. Linear Search
  2. Binary Search
- **Sorting Algorithms:**
  1. Selection Sort
  2. Bubble Sort
  3. Insertion Sort
  4. Quick Sort
  5. Merge sort



# Data Structures: Introduction

---

## + **Linked List**

- Concept & definition
- Types of Linked List
- Operations on Linked List
- Difference between an array and linked list

## + **Stack**

- Concept & definition
- Implementation of stack data structure
- Stack applications algorithms:
  1. Conversion of infix expression into its equivalent prefix
  2. Conversion of infix expression into its equivalent postfix
  3. Conversion of prefix expression into its equivalent postfix
  4. Postfix expression evaluation



# Data Structures: Introduction

---

## + Queue

- Concept & definition
- Types of queue
- Implementation of queue data structure
- Applications of queue

## + Introduction to an advanced data structure

- Tree
- Graph



# Data Structures: Introduction

---

## Q. What is Data Structure?

Data Structure is **a way to store data elements into the memory (i.e. into the main memory) in an organized manner** so that operations like addition, deletion, traversal, searching, sorting etc... can be performed on it efficiently.



# Data Structures: Introduction

---

Two types of **Data Structures** are there:

**1. Linear/Basic Data Structures:** data elements gets stored into the memory in a linear manner (e.g. sequentially ) and hence can be accessed linearly/sequentially.

- Array
- Structure & Union
- Linked List
- Stack
- Queue

**2. Non-linear/Advanced Data Structures:** data elements gets stored into the memory in a non-linear manner (e.g. hierarchical) and hence can be accessed non-linearly.

- Tree (Hierarchical)
- Graph
- Hash Table
- Binary Heap



# Data Structures: Introduction

---

**Array:** It is a **basic/linear data structure** which is a collection/list of **logically related similar type of elements** in which data elements gets stored into the memory at **contiguous locations**.

**Structure:** It is a **basic/linear data structure** which is a collection/list of **logically related similar and dissimilar type of elements** gets stored into the memory **collectively (as a single entity/record)**.

Sizeof of the structure = sum of size of all its members.

**Union:** Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).





# Data Structures: Introduction

## Q. What is a Program?

- A program is a finite set of instructions written in any programming language (like C, C++, Java, Python, Assembly etc...) given to the machine to do specific task.

## Q. What is an Algorithm?

- An algorithm is a finite set of instructions written in human understandable language (like english), if followed, accomplish a given task.

- An algorithm is a finite set of instructions written in human understandable language (like english) **with some programming constraints**, if followed, accomplish a given task, such an algorithm also called as **pseudocode**.

- **An algorithm is a template whereas a program is an implementation of an algorithm.**



# Data Structures: Introduction

**Example: An algorithm to do sum of all array elements**

**Algorithm ArraySum(A, n)//whereas A is an array of size n**

```
{  
    sum=0;//initially sum is 0  
    for( index = 1 ; index <= size ; index++ ) {  
        sum += A[ index ];//add each array element into the sum  
    }  
    return sum;  
}
```

- In this algorithm, **traversal/scanning** operation is applied on an array. Initially sum is 0, each array element gets added into to the sum by traversing array sequentially from the first element till last element and final result is returned as an output.



# Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two measures of an analysis of an algorithms:

- 1. Time Complexity** of an algorithm is the amount of time i.e. computer time required for it to run to completion.

- 2. Space Complexity** of an algorithm is the amount of space i.e. computer memory required for an algorithm to run to completion.

**Asymptotic Analysis:** It is a **mathematical** way to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language**.

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms comparison is the basic operation and hence analysis gets done on the basis of no. of comparisons, in addition of matrices algorithms addition is the basic operation and hence on the basis of addition operation.



# Data Structures: Introduction

**"Best case time complexity":** if an algo takes min amount of time to run to completion then it is referred as best case time complexity.

**"Worst case time complexity":** if an algo takes max amount of time to run to completion then it is referred as worst case time complexity.

**"Average case time complexity":** if an algo takes neither min nor max amount of time to run to completion then it is referred as an average case time complexity.

**"Asympotic Notations":**

**1. Big Omega ( $\Omega$ ):** this notation is used to denote best case time complexity – also called as **asymptotic lower bound**

**2. Big Oh ( $O$ ):** this notation is used to denote worst case time complexity – also called as **asymptotic upper bound**

**3. Big Theta ( $\Theta$ ):** this notation is used to denote an average case time complexity – also called as **asymptotic tight bound**



# Data Structures: Searching Algorithms

## 1. Linear Search/Sequential Search:

**Step-1:** accept key from the user

**Step-2:** compare the value of key with each array element sequentially by traversing it from the first element till either key is found or maximum till the last element. If key is found then return true otherwise return false.

```
Algorithm LinearSearch( A, size, key){  
    for( index = 1 ; index <= size ; index++ ){  
        if( key == A[ index ] )  
            return true;  
    }  
    return false;  
}
```



# Data Structures: Searching Algorithms

**Best Case:** If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is  $O(1) \Rightarrow$  and hence time complexity =  $\Omega(1)$

**Worst Case:** If either key is found at last position or key does not exist, maximum  $n$  no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is  $O(n) \Rightarrow$  and hence time complexity =  $O(n)$

**Average Case:** If key is found at any in between position it is considered as an average case and running time of an algorithm in this case is  $O(n/2) \Rightarrow$  and hence time complexity =  $\theta(n)$



# Data Structures: Searching Algorithms

## 2. Binary Search/Logarithmic Search:

- This algorithm follows **divide-and-conquer** approach.
- To apply binary search on an array prerequisite is that **array elements must be in a sorted manner.**

**Step-1** : accept key from the user

**Step-2** : calculate mid position of an array by the formula,  **$\text{mid} = (\text{left} + \text{right}) / 2$**  ( by means of calculating mid position big size array has been divided logically into two subarrays, from **left to mid-1 = left subarray** & **mid+1 to right = right subarray** )

**Step-3** : compare the value of key with element which is at mid position. if key matches with element at mid position means key is found and return true.

**Step-4** : if key do not matches then check, is the value of key is less than element which is at mid position, if yes then goto search key only into the left subarray by skipping whole right subarray otherwise (means of the value of key is greater than element which is at mid position ) goto search key only into the right subarray by skipping whole left subarray.

**Step-5** : repeat Step-2, Step-3 & Step-4 till either key is found or max till the subarray is valid, if subarray becomes invalid means key is not found and hence return false in this case.



# Data Structures: Searching Algorithms

```
Algorithm BinarySearch(A, n, key) //A is an array of size "n", and key to be search
{
    left = 1;
    right = n;

    while( left <= right )
    {
        //calculate mid position
        mid = (left+right)/2;
        //compare key with an ele which is at mid position
        if( key == A[ mid ] )//if found return true
            return true;

        //if key is less than mid position element
        if( key < A[ mid ] )
        {
            right = mid-1; //search key only in a left subarray
        }
        else //if key is greater than mid position element
        {
            left = mid+1; //search key only in a right subarray
        }
    } //repeat the above steps either key is not found or max any subarray is valid
    return false;
}
```





# Data Structures: Searching Algorithms

- as in each iteration one comparison takes place and search space is getting reduced by half.

$n \Rightarrow n/2 \Rightarrow n/4 \Rightarrow n/8 \dots\dots$

after iteration-1  $\Rightarrow n/2 + 1 \Rightarrow T(n) = (n/2^1) + 1$

after iteration-2  $\Rightarrow n/4 + 2 \Rightarrow T(n) = (n/2^2) + 2$

after iteration-3  $\Rightarrow n/8 + 3 \Rightarrow T(n) = (n/2^3) + 3$

after k iterations  $\Rightarrow \mathbf{T(n) = (n/2^k) + k \dots\dots \text{(equation-I)}}$

let us assume,

$\Rightarrow \mathbf{n = 2^k}$

$\Rightarrow \log n = \log 2^k$  (by taking log on both sides)

$\Rightarrow \log n = k \log 2$

$\Rightarrow \log n = k$

$\Rightarrow \mathbf{k = \log n}$

**By substituting value of n & k in equation-I, we get**

$\Rightarrow T(n) = (n / 2^k) + k$

$\Rightarrow T(n) = (2^k / 2^k) + \log n$

$\Rightarrow T(n) = 1 + \log n \Rightarrow T(n) = \log n \Rightarrow \mathbf{T(n) = O(\log n)}.$



# Data Structures: Searching Algorithms

---

**Best Case:** if the key is found in very first iteration at mid position in only 1 no. of comparison it is considered as a best case and running time of an algorithm in this case is  $O(1) = \Omega(1)$ .

**Worst Case:** if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is  $O(\log n) = \mathbf{O(\log n)}$ .

**Average Case:** if key is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is  $O(\log n) = \mathbf{\theta(\log n)}$ .



# Data Structures: Sorting Algorithms

## 1. Selection Sort:

- In this algorithm, in first iteration, first position gets selected and element which is at selected position gets compared with all its next position elements, if selected position element found greater than any other position element then swapping takes place and in first iteration smallest element gets settled at first position.
- In the second iteration, second position gets selected and element which is at selected position gets compared with all its next position elements, if selected position element found greater than any other position element then swapping takes place and in second iteration second smallest element gets settled at second position, and so on in maximum **(n-1)** no. of iterations all array elements get arranged in a sorted manner.



# Data Structures: Sorting Algorithms

Iteration-1	Iteration-2	Iteration-3	Iteration-4	Iteration-5
<div><div>302060501040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>103060502040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102060503040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030605040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030406050</div><div>012345</div><div>sel_pospos</div></div>
<div><div>203060501040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>103060502040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102050603040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030506040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030405060</div><div>012345</div><div></div></div>
<div><div>203060501040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>103060502040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030605040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030406050</div><div>012345</div><div></div></div>	
<div><div>203060501040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102060503040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102030605040</div><div>012345</div><div></div></div>		
<div><div>103060502040</div><div>012345</div><div>sel_pospos</div></div>	<div><div>102060503040</div><div>012345</div><div></div></div>			
<div><div>103060502040</div><div>012345</div><div></div></div>				



# Data Structures: Sorting Algorithms

**Best Case** :  $\Omega(n^2)$

**Worst Case** :  $O(n^2)$

**Average Case** :  $\theta(n^2)$

## 2. Bubble Sort:

- In this algorithm, in every iteration elements which are at two consecutive positions get compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if prev position element is greater than its next position element then swapping takes place, and by this logic in first iteration largest element gets settled at last position, in second iteration second largest element gets settled at second last position and so on, in max **(n-1)** no. of iterations all elements get arranged in a sorted manner.



# Data Structures: Sorting Algorithms

Iteration-1	Iteration-2	Iteration-3	Iteration-4	Iteration-5
<div><div>302060501040</div><div>012345</div><div>pospos+1</div></div>	<div><div>203050104060</div><div>012345</div><div>pospos+1</div></div>	<div><div>203010405060</div><div>012345</div><div>pospos+1</div></div>	<div><div>201030405060</div><div>012345</div><div>pospos+1</div></div>	<div><div>102030405060</div><div>012345</div><div>pospos+1</div></div>
<div><div>203060501040</div><div>012345</div><div>pospos+1</div></div>	<div><div>203050104060</div><div>012345</div><div>pospos+1</div></div>	<div><div>203010405060</div><div>012345</div><div>pospos+1</div></div>	<div><div>102030405060</div><div>012345</div><div>pospos+1</div></div>	<div><div>102030405060</div><div>012345</div><div></div></div>
<div><div>203060501040</div><div>012345</div><div>pospos+1</div></div>	<div><div>203050104060</div><div>012345</div><div>pospos+1</div></div>	<div><div>201030405060</div><div>012345</div><div>pospos+1</div></div>	<div><div>102030405060</div><div>012345</div><div></div></div>	
<div><div>203050601040</div><div>012345</div><div>pospos+1</div></div>	<div><div>203010504060</div><div>012345</div><div>pospos+1</div></div>	<div><div>201030405060</div><div>012345</div><div></div></div>		
<div><div>203050106040</div><div>012345</div><div>pospos+1</div></div>	<div><div>203010405060</div><div>012345</div><div></div></div>			
<div><div>203050104060</div><div>012345</div><div>pospos+1</div></div>				
<div><div>203050104060</div><div>012345</div><div></div></div>				



# Data Structures: Sorting Algorithms

**Best Case** :  $\Omega(n)$  - if array elements are already arranged in a sorted manner.

**Worst Case** :  $O(n^2)$

**Average Case** :  $\theta(n^2)$

## 3. Insertion Sort:

- In this algorithm, in every iteration one element gets selected as a **key element** and key element gets inserted into an array at its appropriate position towards its left hand side elements in a such a way that elements which are at left side are arranged in a sorted manner, and so on, in max **(n-1)** no. of iterations all array elements gets arranged in a sorted manner.
- This algorithm works efficiently for already sorted input sequence by design and hence running time of an algorithm is  $O(n)$  and it is considered as a best case.



# Data Structures: Sorting Algorithms

**Best Case** :  $\Omega(n)$  - if array elements are already arranged in a sorted manner.

**Worst Case** :  $O(n^2)$

**Average Case:**  $\theta(n^2)$

- Insertion sort algorithm is an efficient algorithm for smaller input size array.

## 4. Merge Sort:

- This algorithm follows **divide-and-conquer** approach.

- In this algorithm, big size array is divided logically into smallest size (i.e. having size 1) subarrays, as if size of subarray is 1 it is sorted, after dividing array into sorted smallest size subarray's, subarrays gets merged into one array step by step in a sorted manner and finally all array elements gets arranged in a sorted manner.

- This algorithm works fine for **even** as well **odd** input size array.

- This algorithm takes extra space to sort array elements, and hence its space complexity is more.



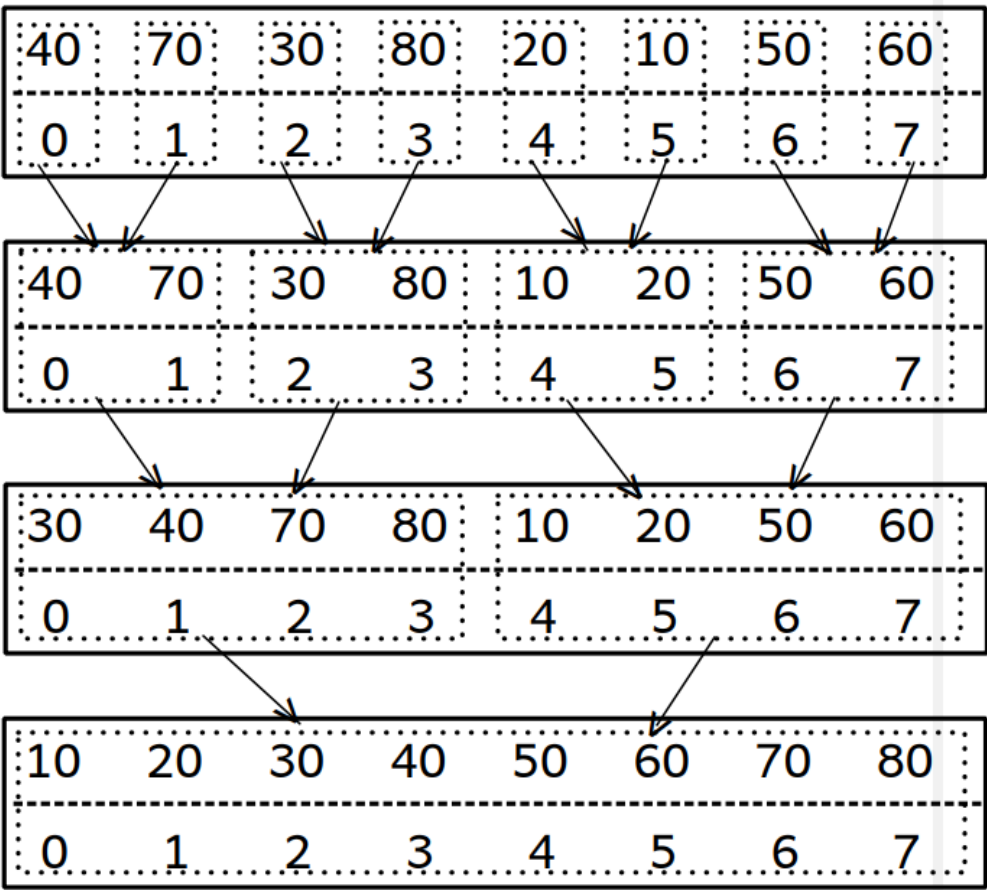
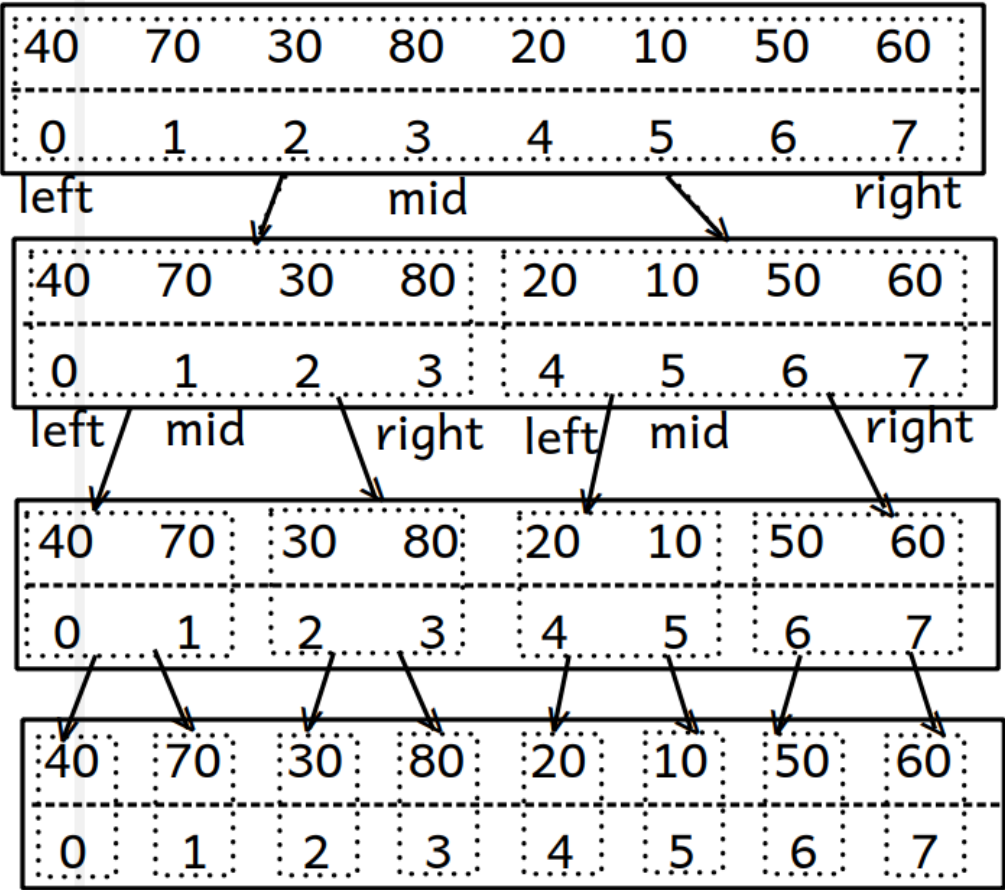


# Data Structures: Sorting Algorithms

## ## Merge Sort ##

Dividing big size array into smallest size subarrays

Merge already sorted arrays



# Data Structures: Sorting Algorithms

---

**Best Case** :  $\Omega(n \log n)$

**Worst Case** :  $O(n \log n)$

**Average Case** :  $\theta(n \log n)$

