

# PCCST303 DATA STRUCTURES AND ALGORITHMS

---

PREPARED BY SHARIKA T R, ASSISTANT PROFESSOR, ASIET

# Course Outcome

---

At the end of the course students should be able to:

Course Outcome		Bloom's Knowledge Level (KL)
CO1	Identify appropriate data structures for solving real world problems.	K3
CO2	Describe and implement linear data structures such as arrays, linked lists, stacks, and queues.	K3
CO3	Describe and Implement non linear data structures such as trees and graphs.	K3
CO4	Select appropriate searching and sorting algorithms to be used in specific circumstances.	K3

*Note: K1- Remember, K2- Understand, K3- Apply, K4- Analyse, K5- Evaluate, K6- Create*

# Module 4- Sorting and Hashing

---

Sorting and Searching Sorting Techniques :- Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort. Searching Techniques :- Linear Search, Binary Search, Hashing - Hashing functions : Mid square, Division, Folding, Digit Analysis; Collision Resolution : Linear probing, Quadratic Probing, Double hashing, Open hashing.

# SORTING

---

Arranging data in a particular order - either ascending or descending

Ascending – arranged from smallest to largest value

Descending – arranged from largest to smallest value

Classification of sorting algorithms:

1. Internal sorting:

- If the elements to be sorted are in main memory
- Used when the list of elements to sort is small so that the sorting can be carried out in main memory

2. External sorting:

- If the elements to be sorted are in the secondary memory (eg. In Disks or tapes)
- Used to sort a large file of records which cannot be carried out in main memory

External sorting depends on:

- Type of device ( eg. Disks, tapes etc)
- Number of such device that can be used at a time

# Internal sorting:

---

Bubble sort

Selection sort

Insertion sort

Merge sort

Quick sort

Heap sort

# PERFORMANCE OF SORTING ALGORITHM

---

Depends on

1. **Space complexity**

- Space required for the number of elements to be sorted

2. **Time complexity**

- Run time for execution of an algorithm
- Sorting time or time complexity is denoted by the Big Oh(O) notation

- Eg: Time complexity =  $O(n)$

→ Time complexity is proportional to the number of elements in the list to be sorted

# EFFICIENCY OF SORTING ALGORITHM

---

## Criteria:

1. **Best case:**
  - If the elements in the **input are already sorted**
2. **Worst case:**
  - If the elements are **in the reverse order** than the one required
  - Eg: if we need to sort elements in ascending order, the input available is in descending order
3. **Average case:**
  - Elements are **scattered** in the list

# SELECTION SORT

---

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part.

This process is repeated for the remaining unsorted portion until the entire list is sorted.



1. Set  $k=0$

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

2. While  $k < n-1$

1. Set  $\text{min}=A[k]$  and  $\text{loc} = k$

2. Set  $j=k+1$

3. While  $j < n$

1. If  $(\text{min} > A[j])$

1.  $\text{min}=A[j]$  and  $\text{loc}=j$

2.  $j++$

4. End while

5. Swap  $A[k]$  and  $A[\text{loc}]$

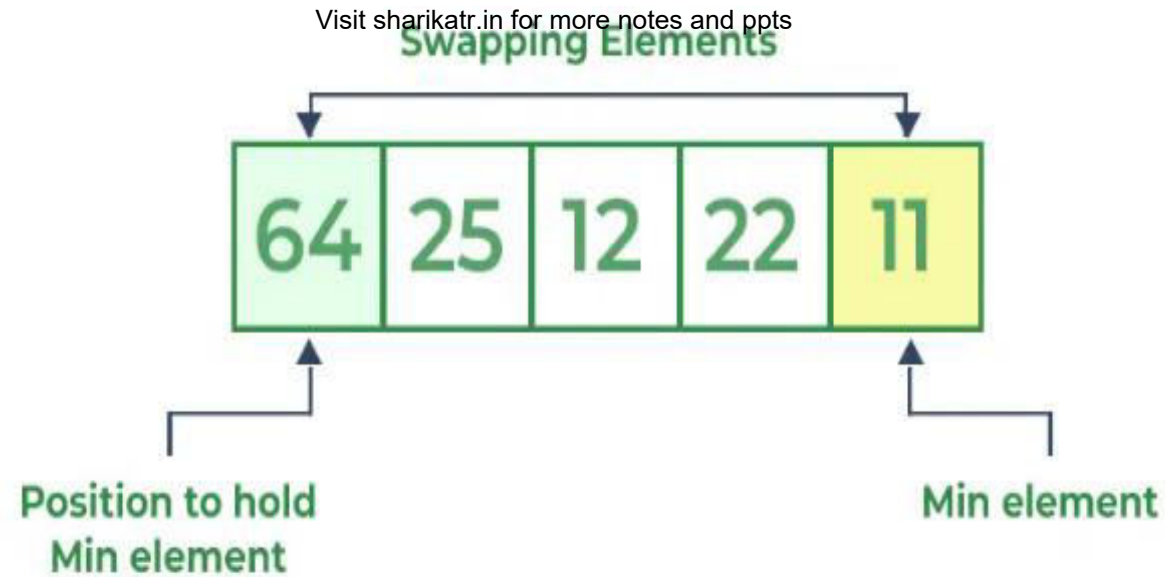
6.  $k++$

3. End while

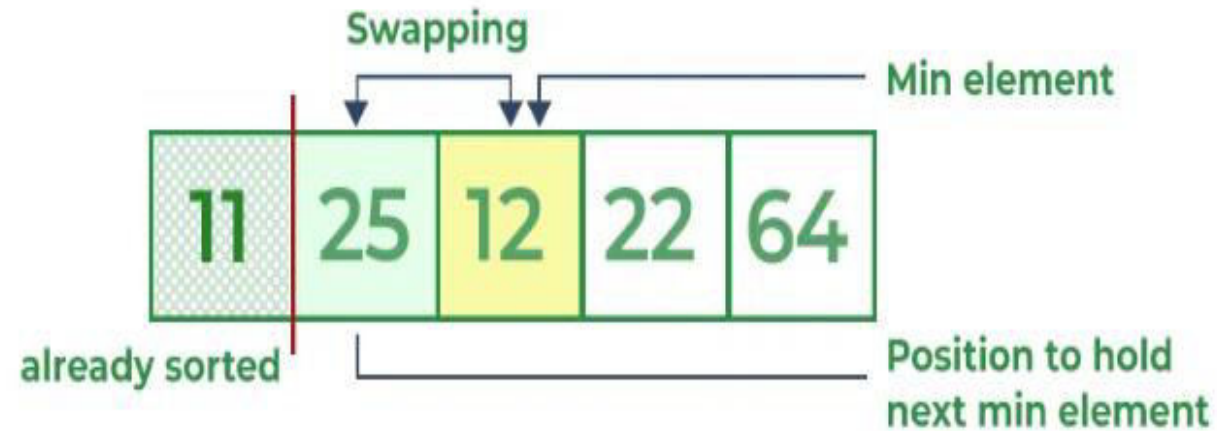
4. stop

Algorithm : Selection sort

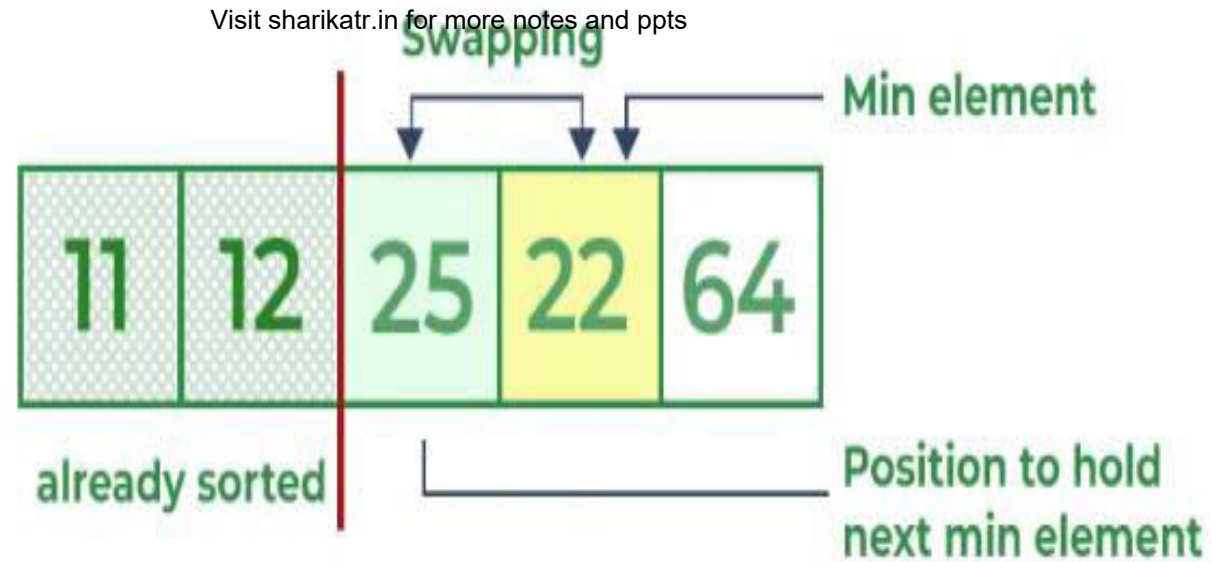
## First pass



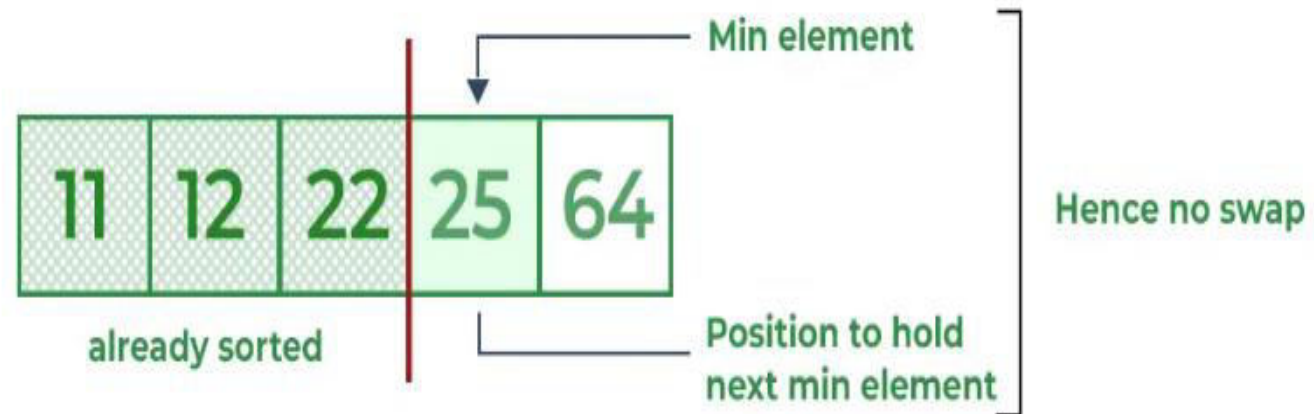
## Second Pass



## Third pass



## Fourth Pass



## ***Fifth pass***

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

11	12	22	25	64
----	----	----	----	----

Sorted array

### **Algorithm**

**Step 1** – Set MIN to location 0

**Step 2** – Search the minimum element in the list

**Step 3** – Swap with value at location MIN

**Step 4** – Increment MIN to point to next element

**Step 5** – Repeat until list is sorted

# Time complexity of selection sort

---

<b>Best case</b>	<b><math>O(N)</math></b>
<b>Average Case</b>	<b><math>O(N^2)</math></b>
<b>Worst case</b>	<b><math>O(N^2)</math></b>

# INSERTION SORT

---

Element is **inserted at its proper position in the sorted list**

**Initially**, the list is divided into 2 parts

**One part** containing the **first element – sorted list**

**Second part** containing the **remaining elements – unsorted list**

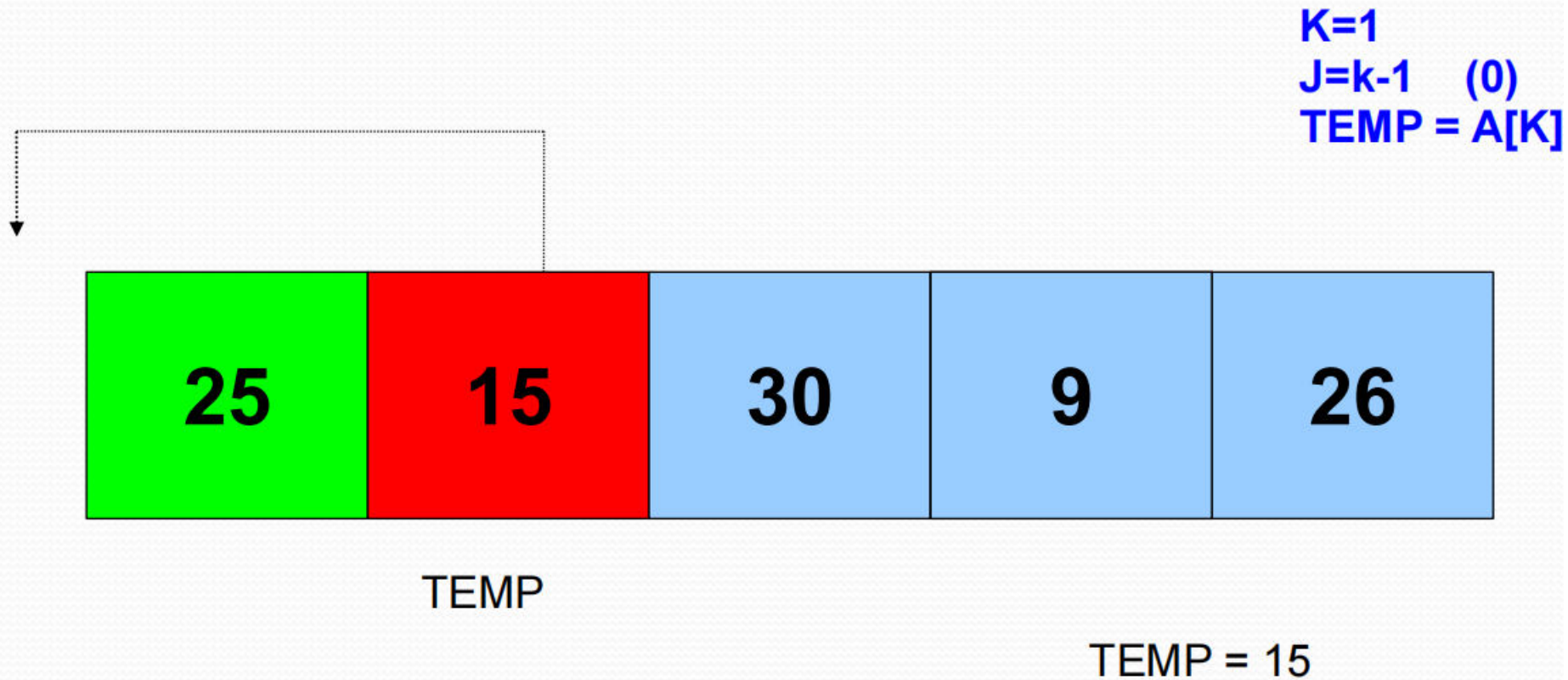
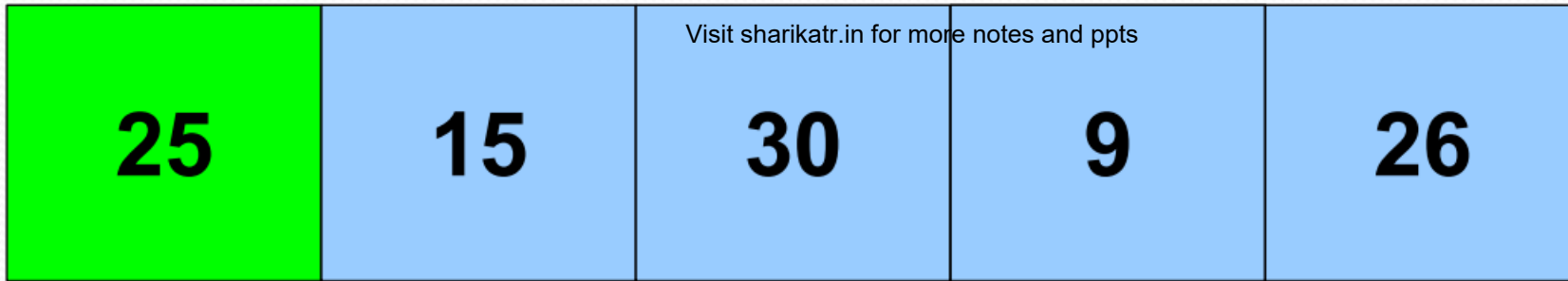
We start with the **first element of the unsorted list** and **insert it at its proper position** in the sorted list

This process continues until all the elements are sorted

1. Set  $k=1$
2. While  $k < n$ 
  1. Temp =  $A[k]$
  2. Set  $j=k-1$
  3. While ( temp  $< A[j]$  and  $j \geq 0$ )
    1.  $A[j+1] = A[j]$
    2.  $j--$
  4. End while
  5.  $A[j+1] = \text{temp}$
  6.  $k++$
3. end while
4. stop

Algorithm : Insertion Sort







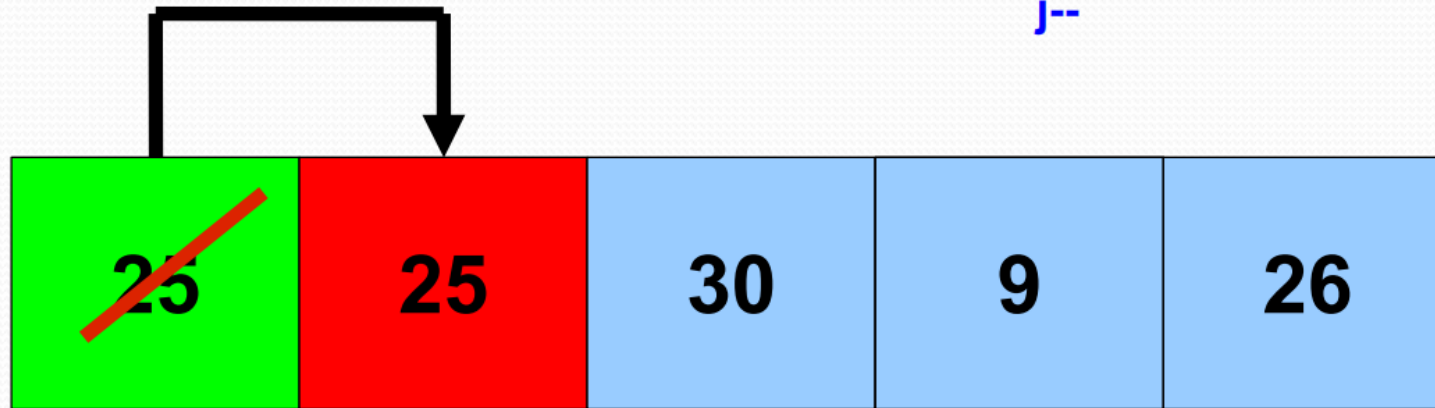


TEMP

TEMP = 15

While (temp < A[j] && j > 0)  
A[j+1] = A[j]

j--



TEMP

**A[j] = temp**

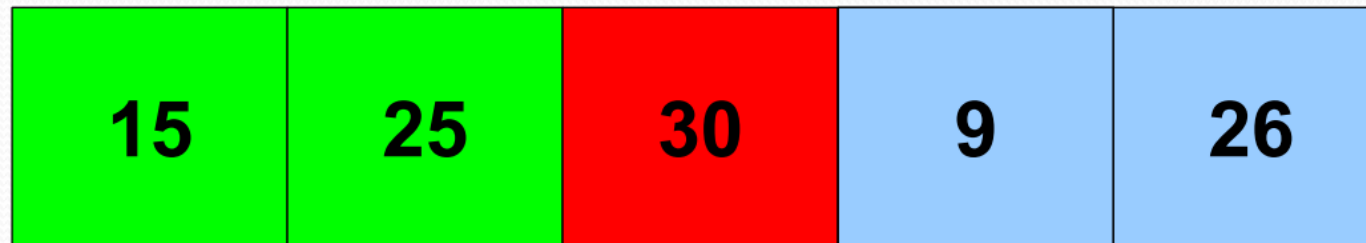
Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts



TEMP

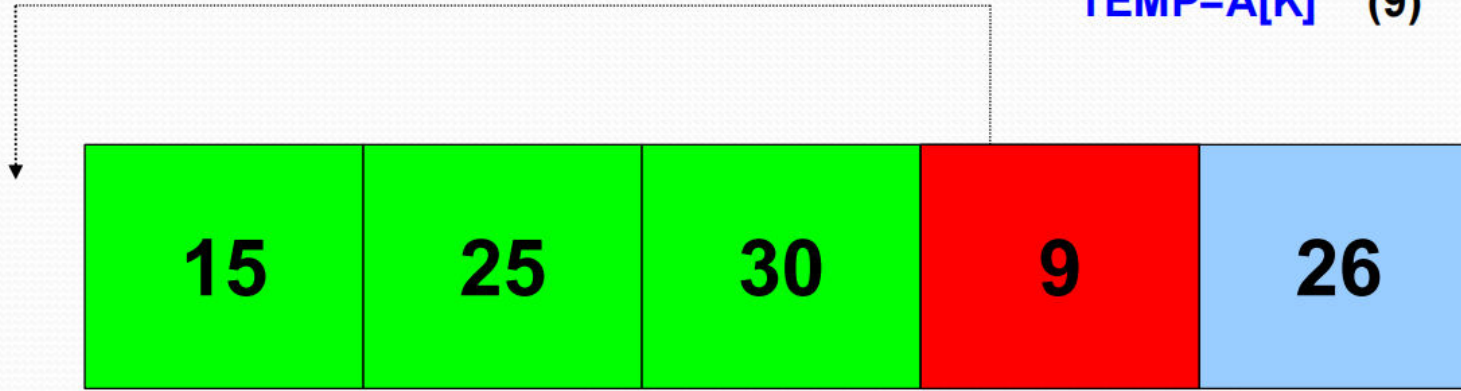
**TEMP = 15**

**K=2**  
**J=K-1 (1)**  
**TEMP=A[K] (30)**



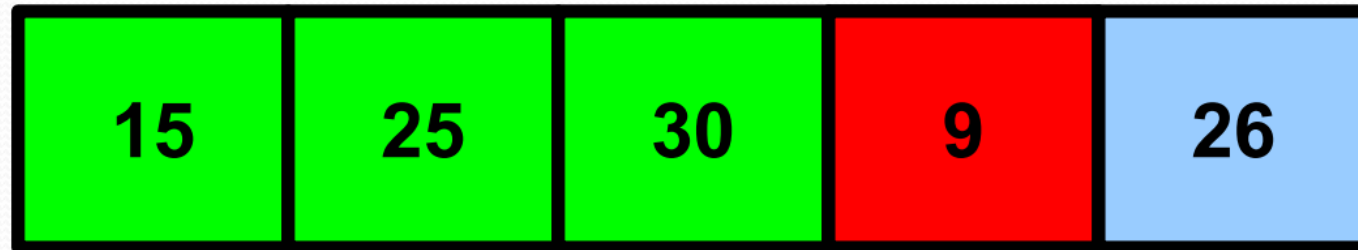
Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

$K=3$   
 $J=K-1$  (2)  
 $TEMP=A[K]$  (9)



TEMP

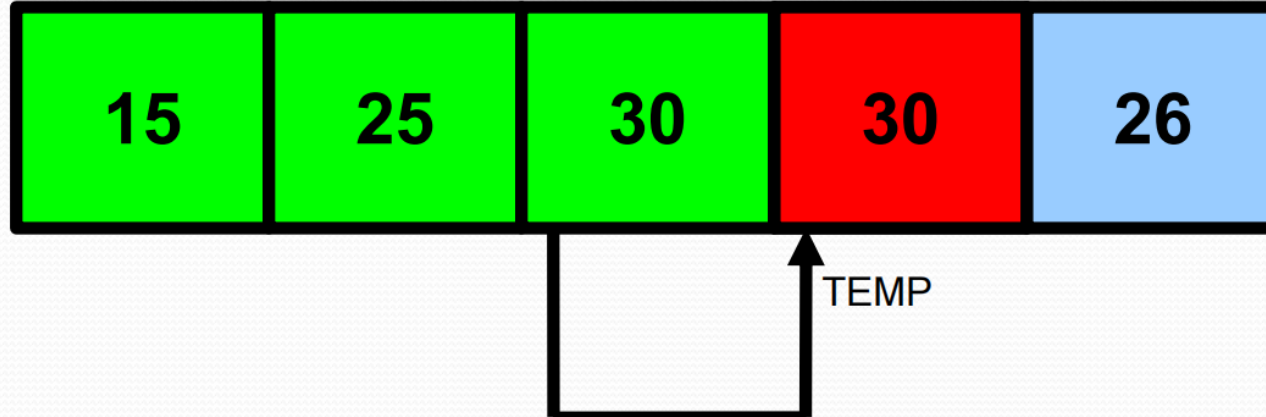
$J=K-1$  (2)  
While ( $temp < A[j]$  &&  $j > 0$ )  
     $A[j+1] = A[j]$   
     $j--$



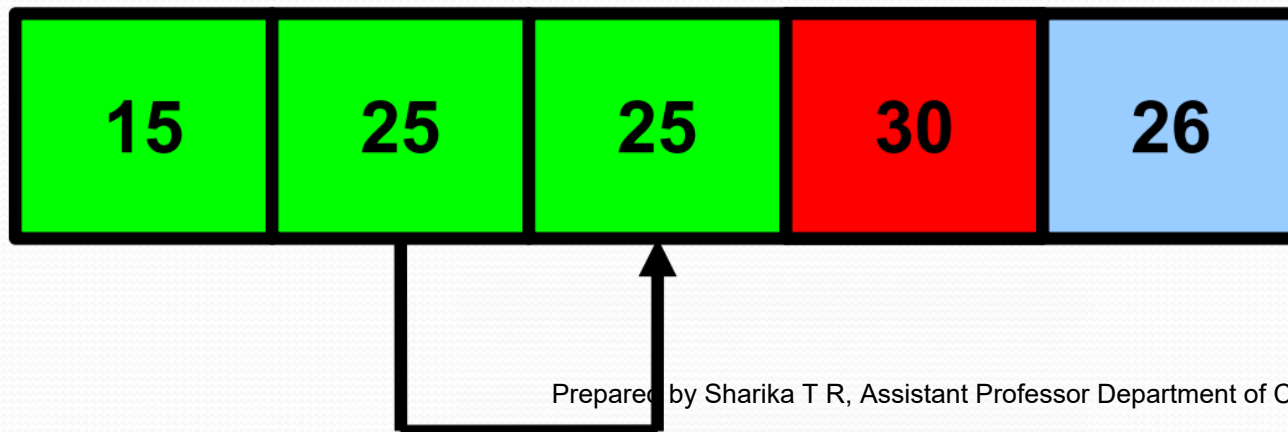
TEMP

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppt's

```
J=K-1      (2)
While (temp<A[j] && j>0)
  A[j+1] = A[j]
  j--
```

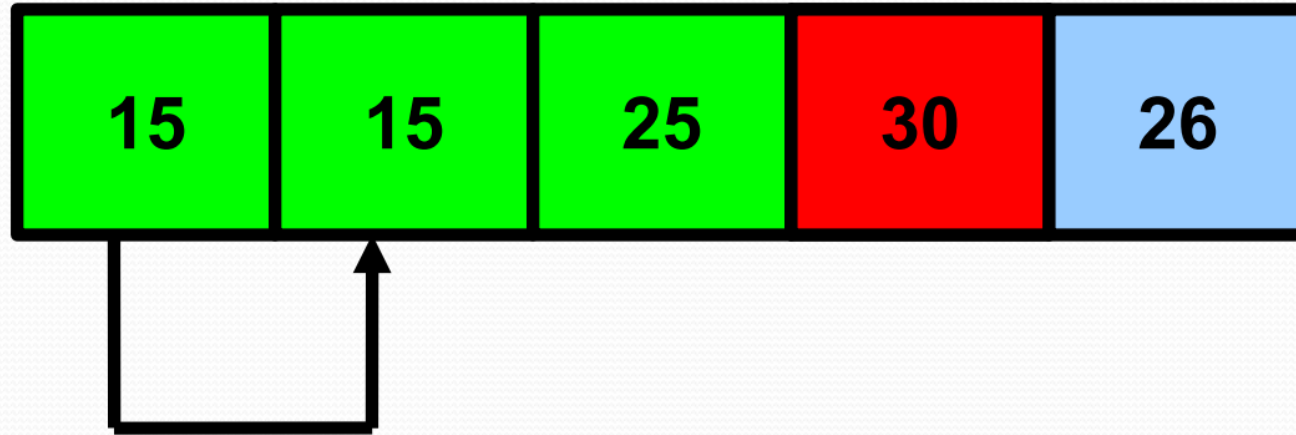


```
J=K-1      (2)
While (temp<A[j] && j>0)
  A[j+1] = A[j]
  j--
```



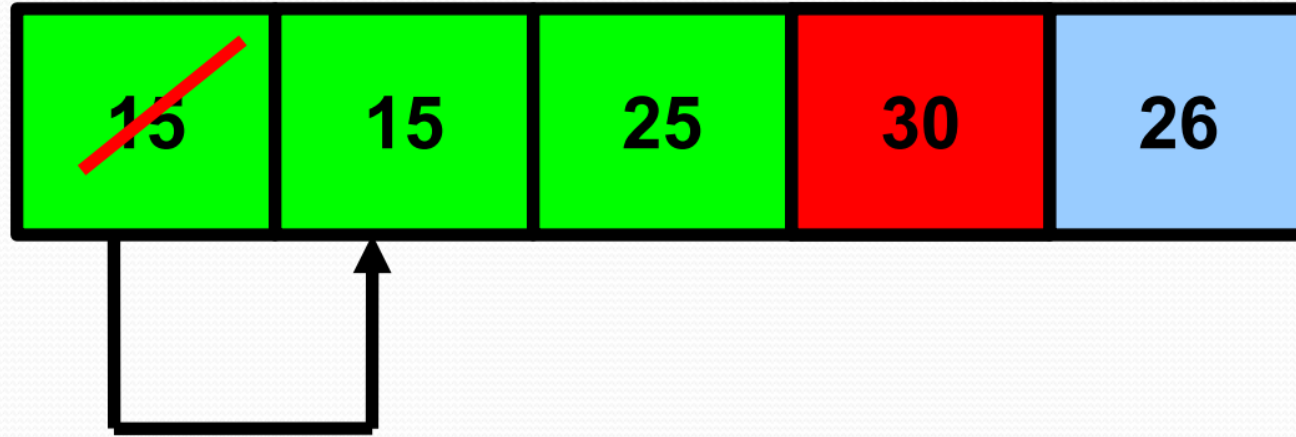
Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

**J=K-1 (2)**  
**While (temp<A[j] && j>0)**  
**A[j+1] = A[j]**  
**j--**

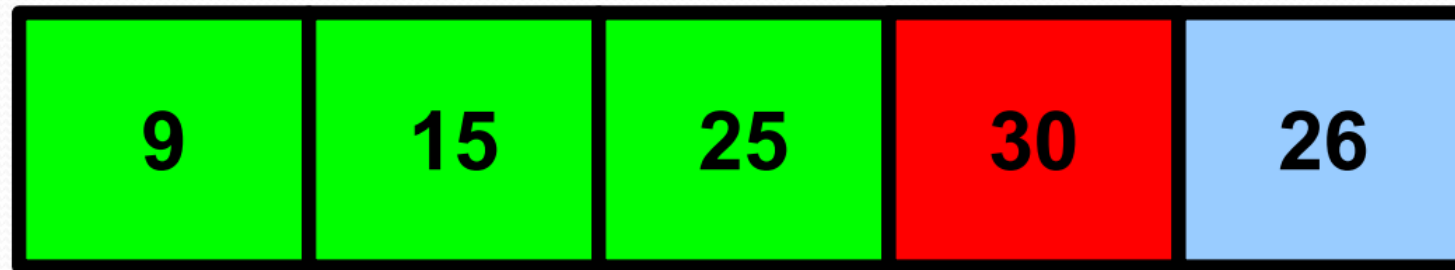


Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

```
J=K-1      (2)
While (temp<A[j] && j>0)
A[j+1] = A[j]
j--
```

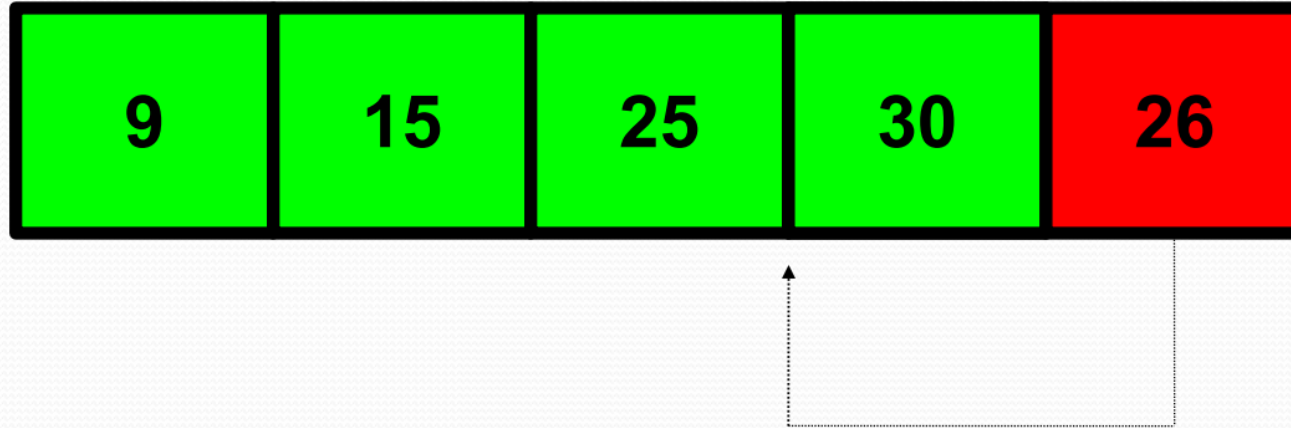


```
A[J] = TEMP      (9)
```



Visit [sharikatr.in](http://sharikatr.in) for more notes and apt's

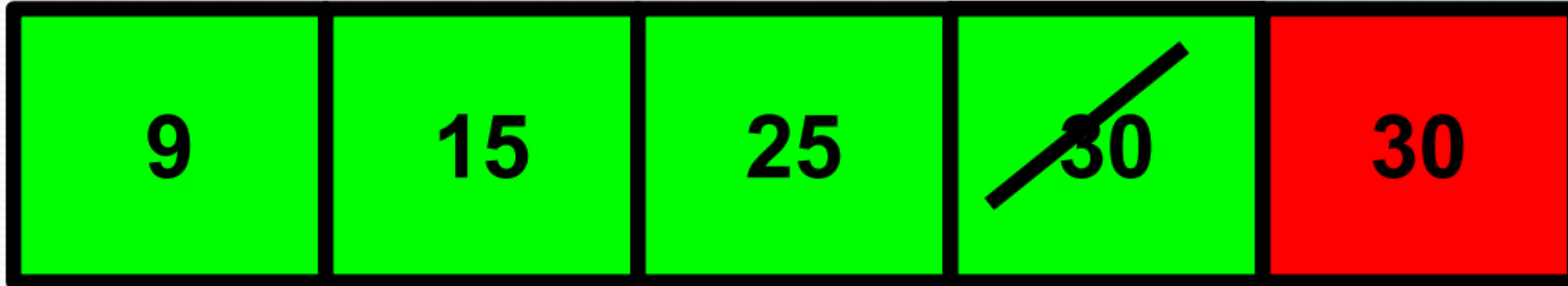
$K=4$   
 $J=K-1$  (3)  
 $TEMP=A[K]$  (26)



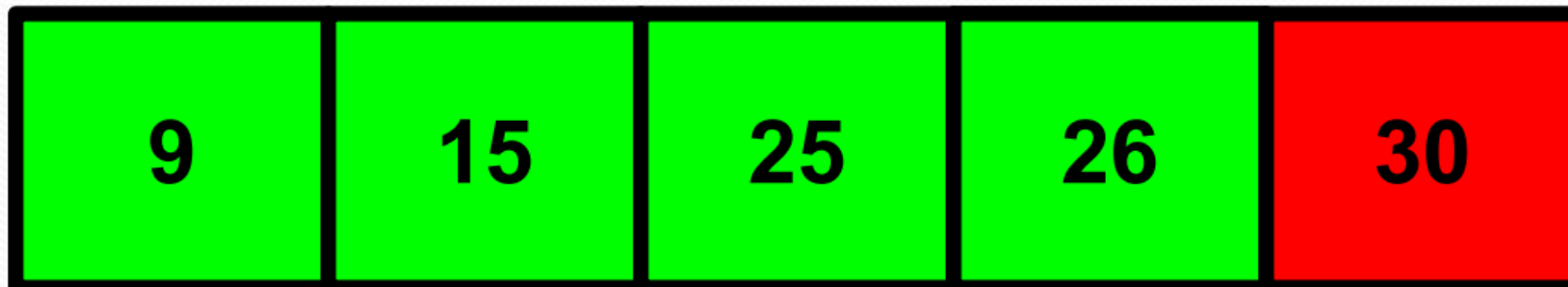
$J=K-1$  (3)  
While ( $temp < A[j] \ \&\& \ j > 0$ )  
     $A[j+1] = A[j]$   
     $j--$



$K=4$   
 $J=K-1$  (3)  
 $TEMP=A[K]$  (26)

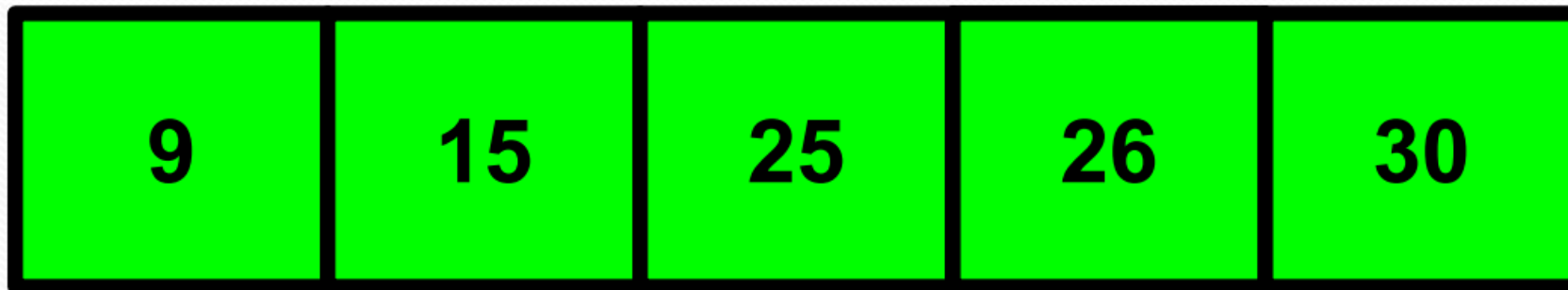


$TEMP=A[K]$  (26)





**SORTED**



# Time complexity of insertion sort

---

<b>Best case</b>	<b><math>O(N)</math></b>
<b>Average Case</b>	<b><math>O(N^2)</math></b>
<b>Worst case</b>	<b><math>O(N^2)</math></b>

## Merge Sort

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

**Merge sort** is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

### **The concept of Divide and Conquer involves three steps:**

- Divide the problem into multiple sub-problems.
- Solve the Sub Problems. The idea is to break down the problem into atomic sub-problems, where they are actually solved.
- Combine the solutions of the sub-problems to find the solution of the actual problem.

### **So, the merge sort working rule involves the following steps:**

- Divide the unsorted array into sub-array, each containing a single element.
- Take adjacent pairs of two single-element array and merge them to form an array of 2 elements.
- Repeat the process till a single sorted array is obtained.

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

divide

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

divide

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

divide

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

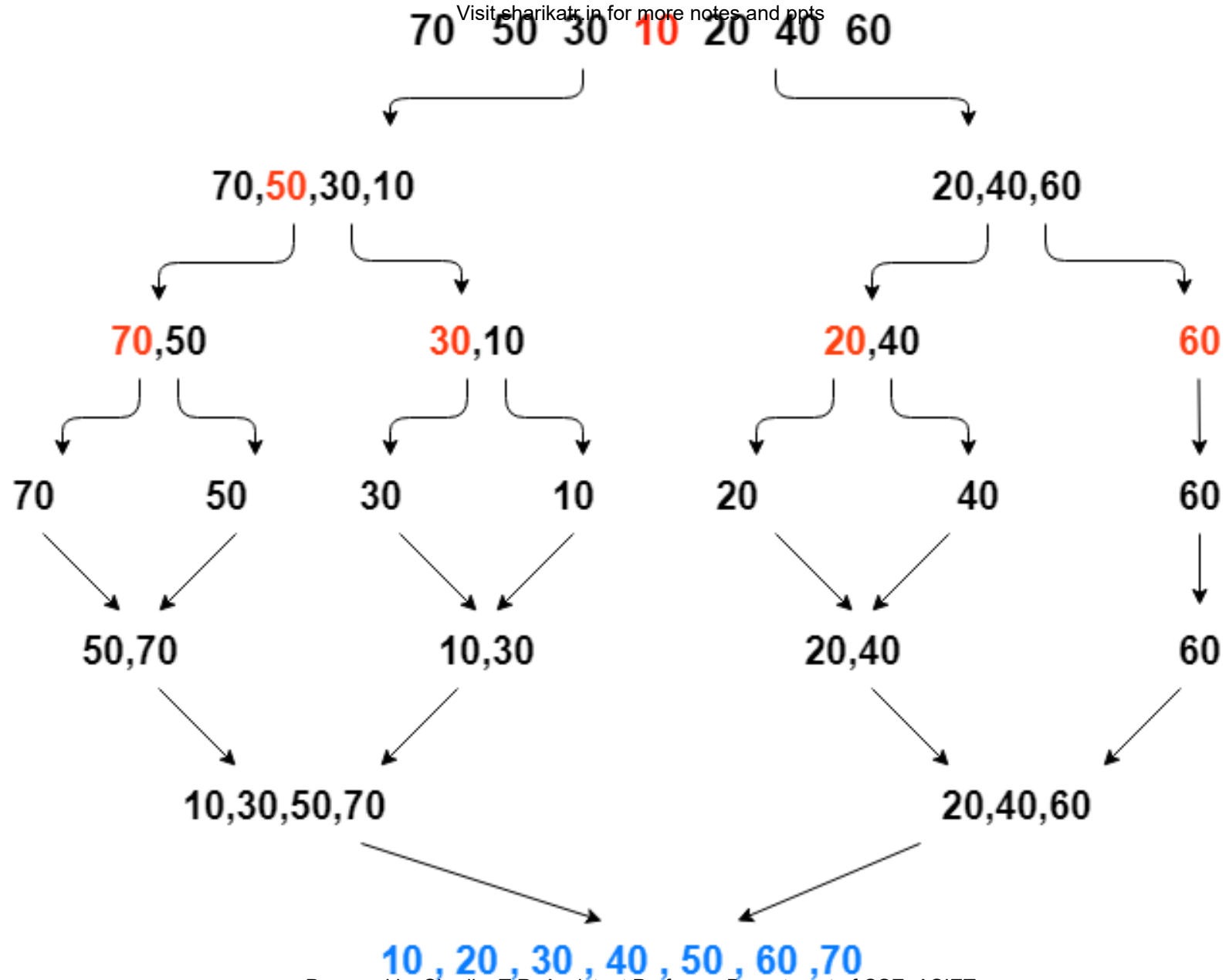
merge

12	31	8	25	17	32	40	42
----	----	---	----	----	----	----	----

merge

8	12	25	31	17	32	40	42
---	----	----	----	----	----	----	----

8	12	17	25	31	32	40	42
---	----	----	----	----	----	----	----



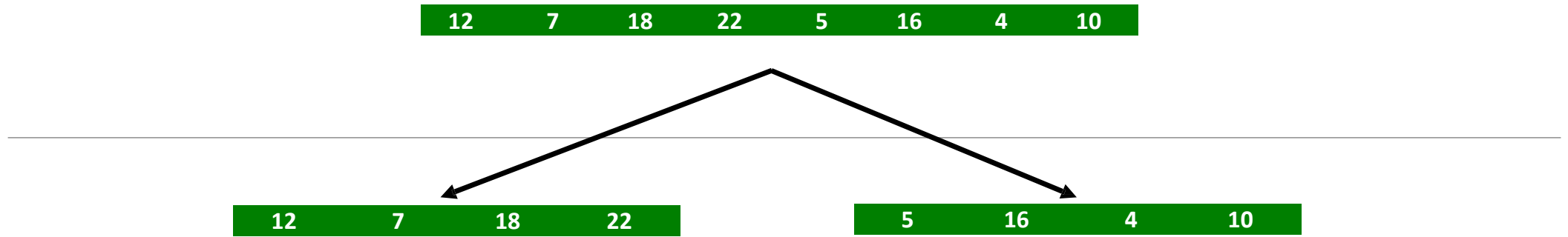
**Eg: DIVIDE**  
Visit sharika.in for more notes and ppts

**12 7 18 22 5 16 4 10**

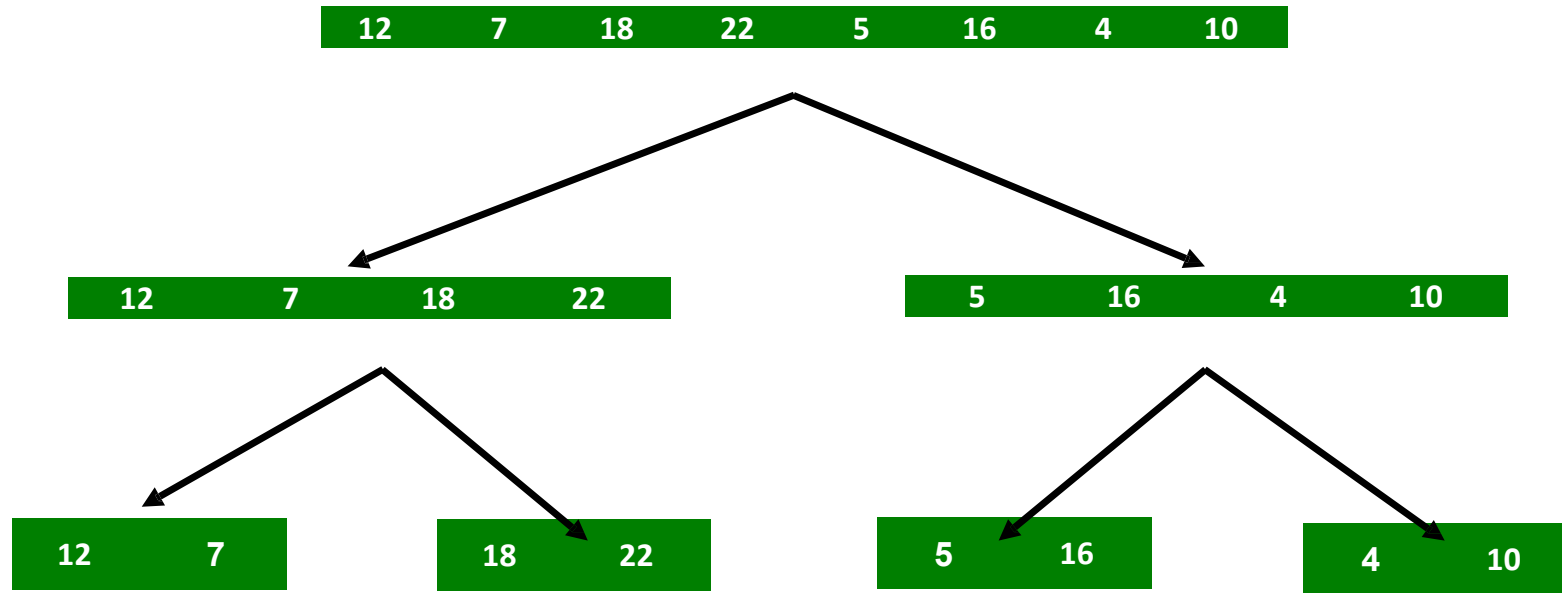
---

Eg: **DIVIDE**

Visit [sharikatr.in](#) for more notes and ppts

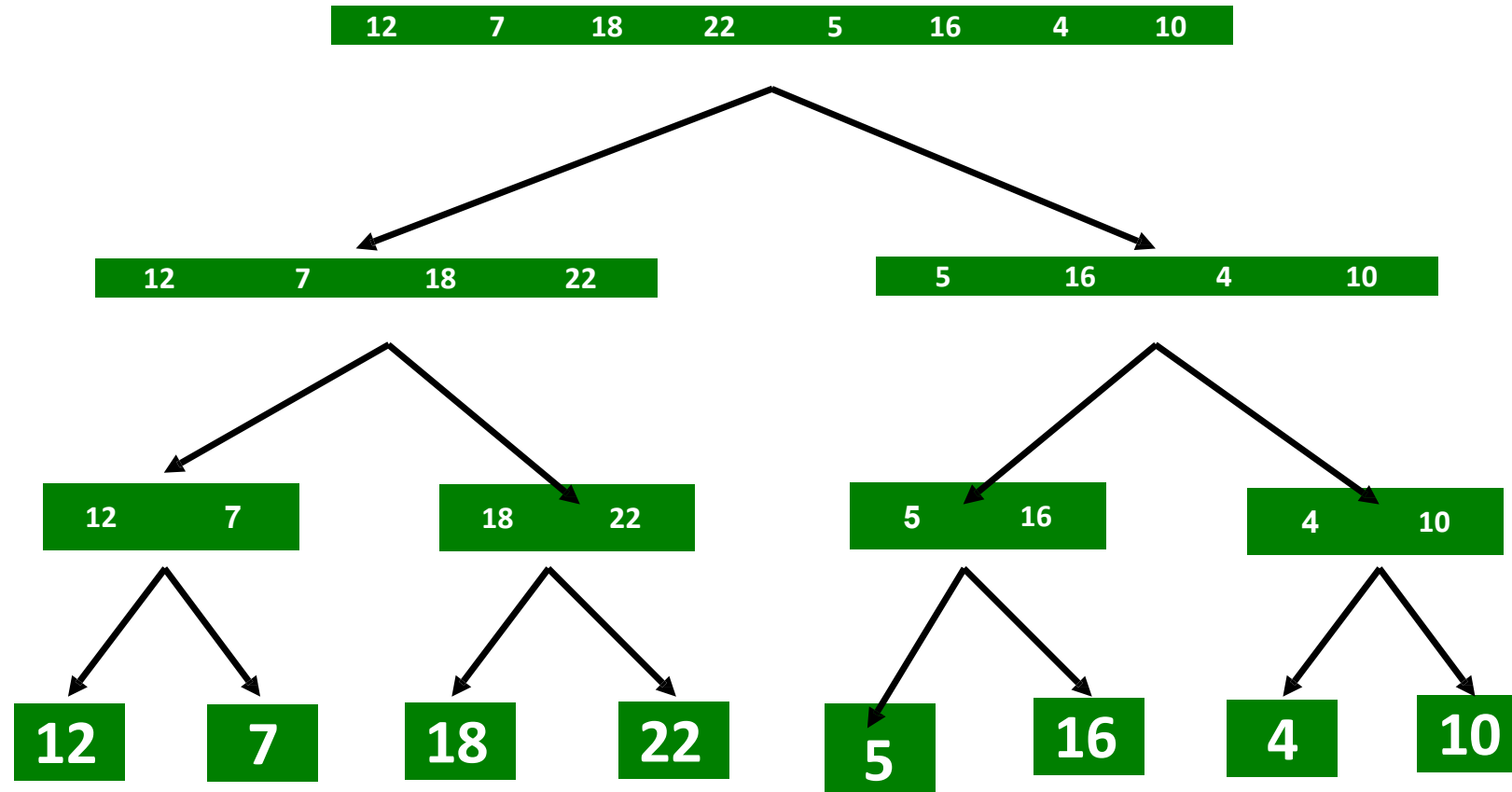


**Eg: DIVIDE**  
Visit sharika.in for more notes and ppts





Eg: **DIVIDE**  
Visit sharika.in for more notes and ppts



Eg: CONQUER

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

12

7

18

22

5

16

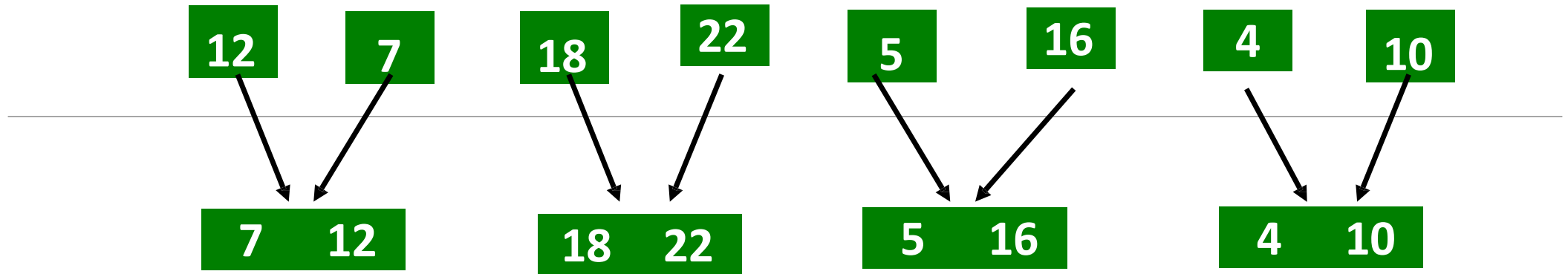
4

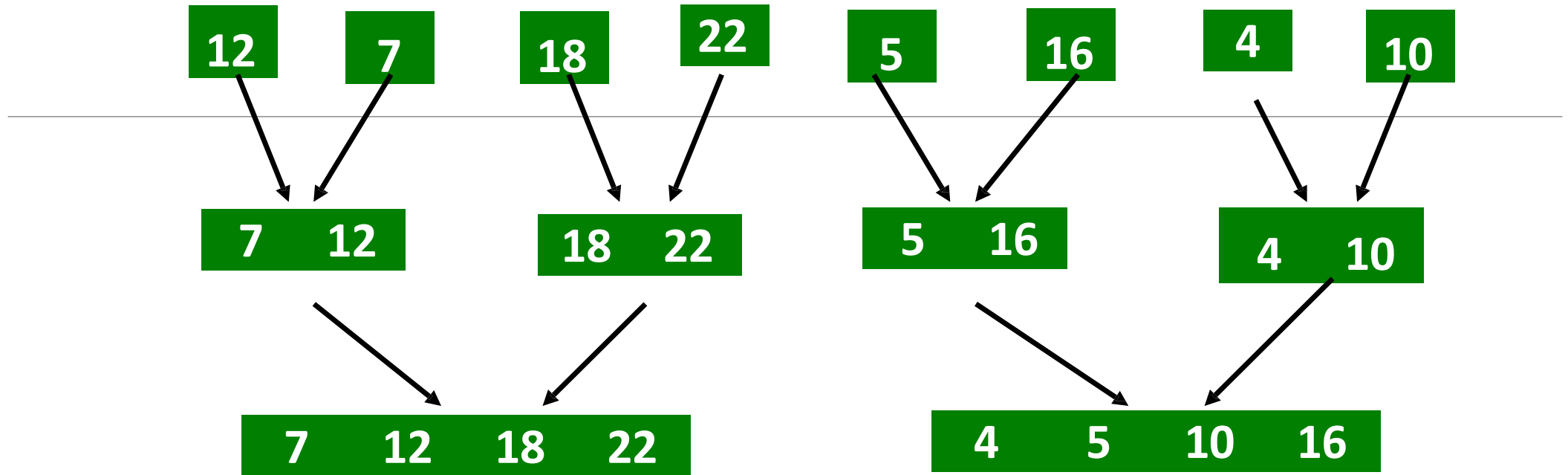
10

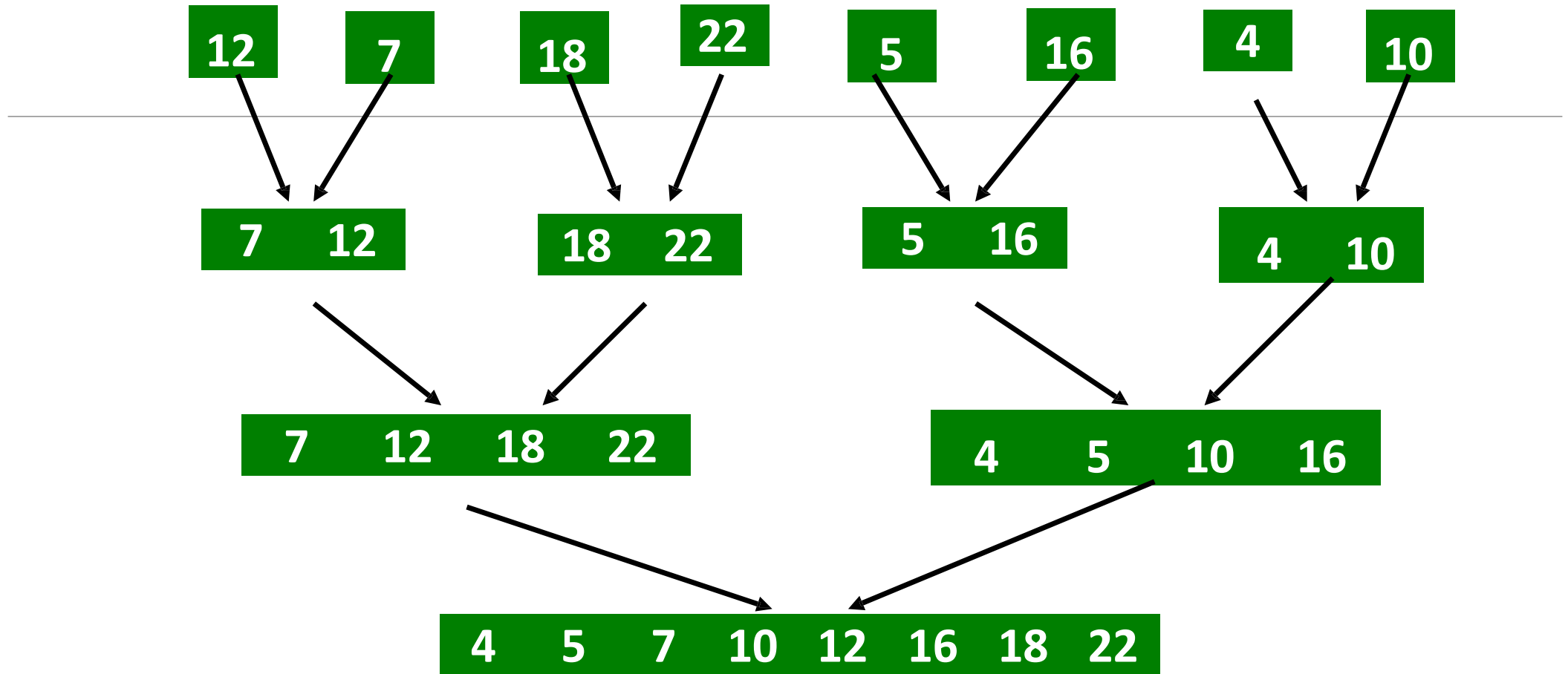
---

Eg: CONQUER

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts







- **Divide part:** We calculate the mid-index, i.e.,  $\text{mid} = l + (r - l)/2$ .
- **Conquer part 1:** We recursively sort the left part of size  $n/2$  by calling the same function with mid as the right end, i.e., `mergeSortAlgorithm(A, l, mid)`.
- **Conquer part 2:** We recursively sort the right part of size  $n/2$  by calling the same function with mid + 1 as the left end, i.e., `mergeSortAlgorithm(A, mid + 1, r)`.
- **Combine part:** Now we use the `merge(A, l, mid, r)` function to merge both sorted halves into a final sorted array.
- **Base case:** If we find  $l == r$  during recursive calls, then the sub-array has one element left, which is trivially sorted. So recursion will not go further and return from there. In other words, the sub-array of size 1 is the smallest version of the sorting problem for which recursion directly returns the solution.

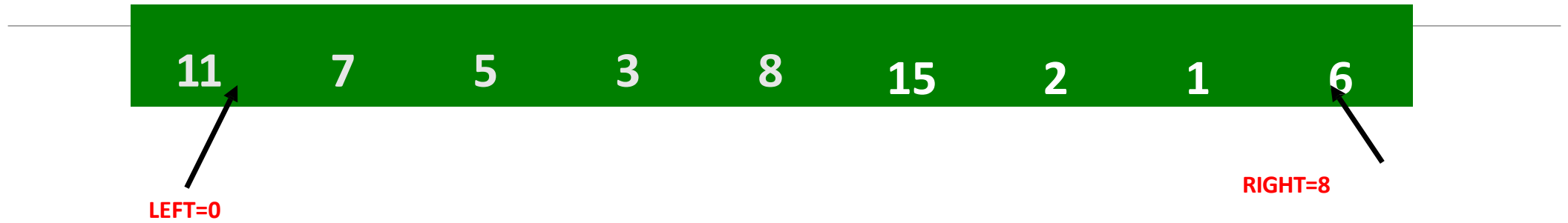
**void mergeSort(int a[], int beg, int end)**  
**{**  
    **if (beg < end)**  
    **{**  
        **int mid = (beg + end) / 2;**  
        **mergeSort(a, beg, mid);**  
        **mergeSort(a, mid + 1, end);**  
        **merge(a, beg, mid, end);**  
    **}**  
**}**

# HOW IS IT DONE ??



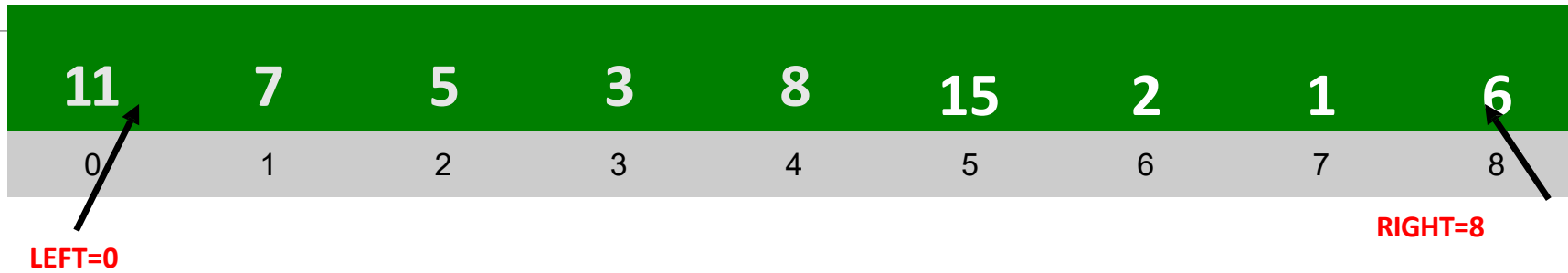
Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

# INPUT ARRAY



Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

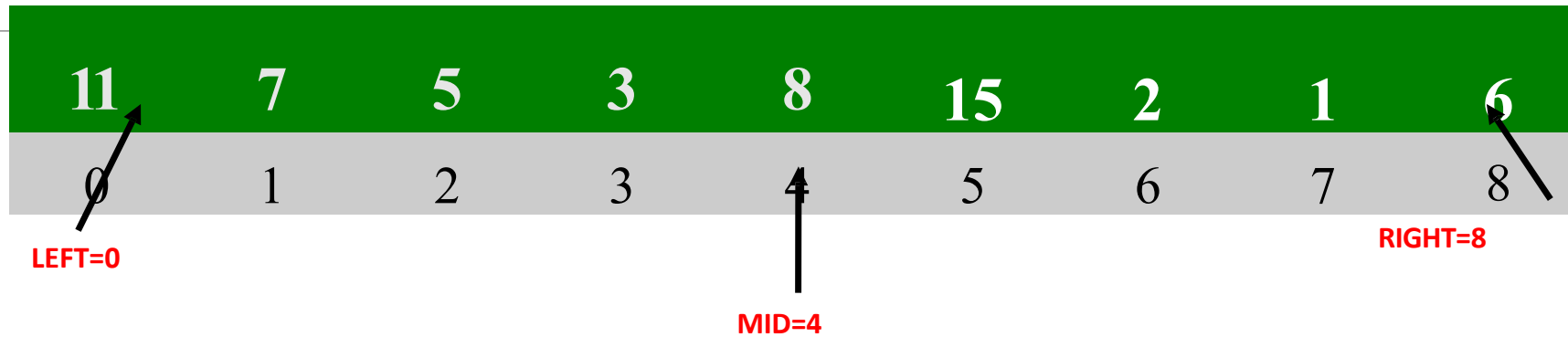
## FIND MID POSITION



$$\begin{aligned}\text{MID} &= (\text{LEFT} + \text{RIGHT}) / 2 \\ &= (0 + 8) / 2 \\ &= 4\end{aligned}$$

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

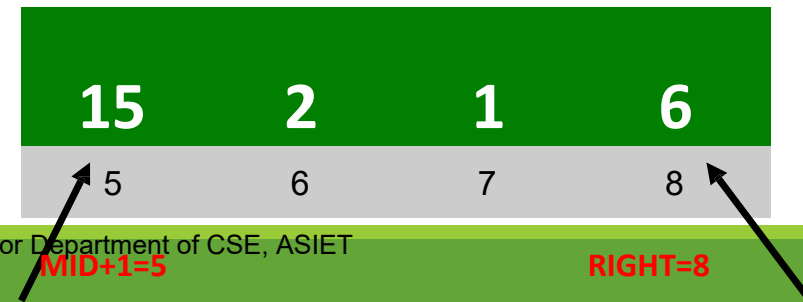
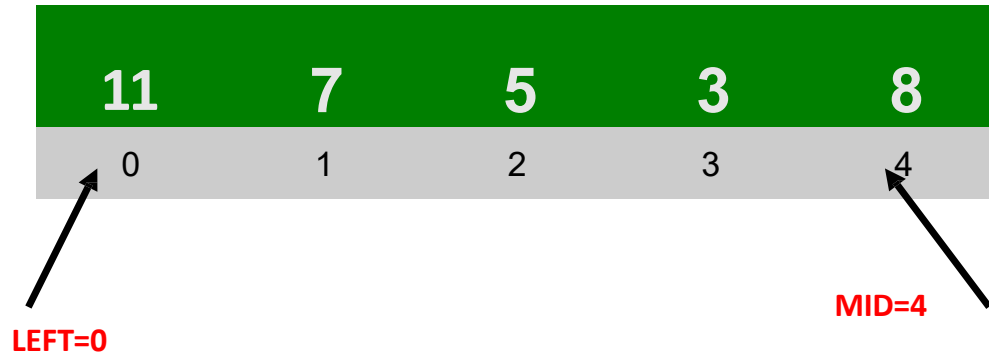
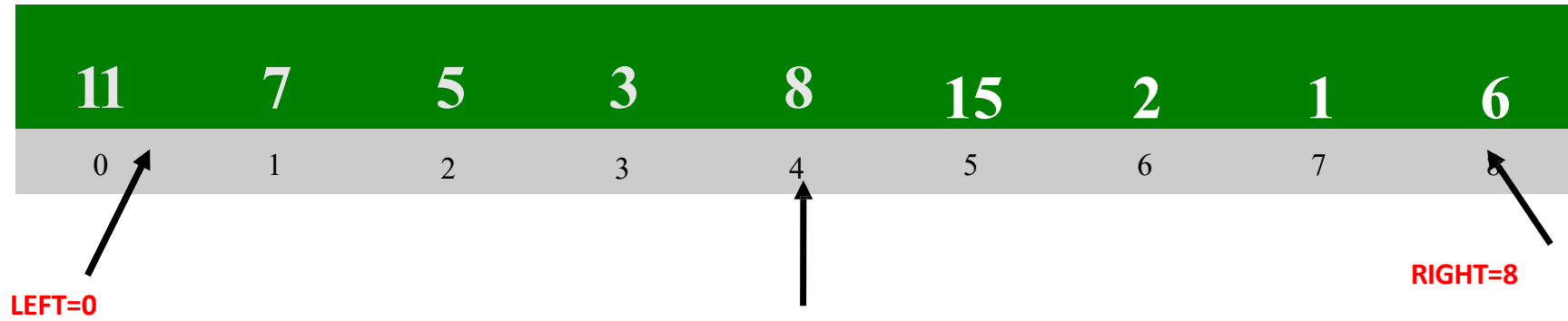
## FIND MID POSITION



$$\begin{aligned} \text{MID} &= (\text{LEFT} + \text{RIGHT}) / 2 \\ &= (0 + 8) / 2 \\ &= 4 \end{aligned}$$

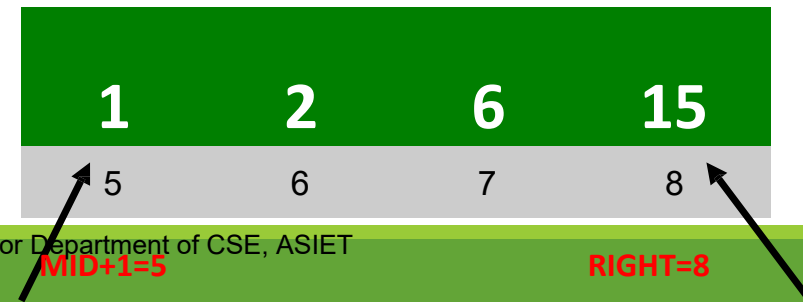
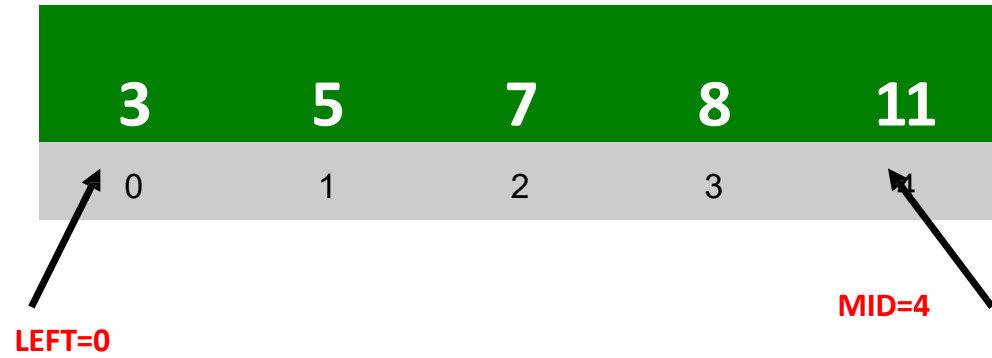
# SPLIT THE ARRAY INTO TWO

Visit [sharkatr.in](http://sharkatr.in) for more notes and ppts

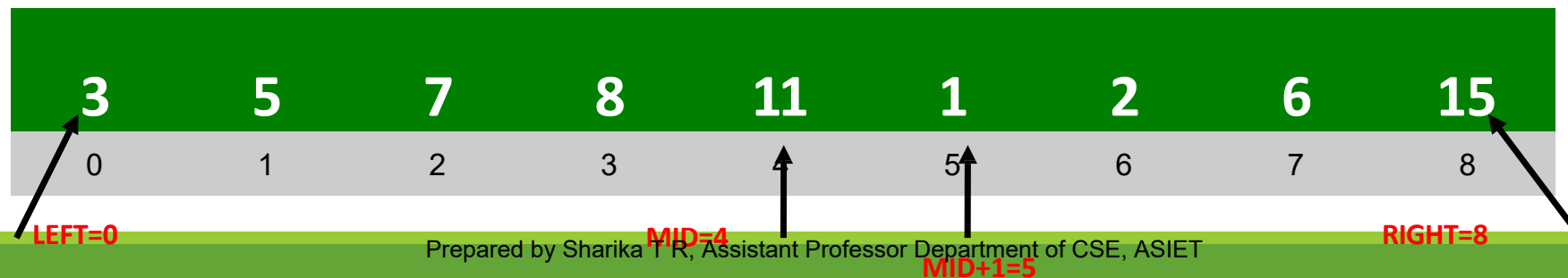
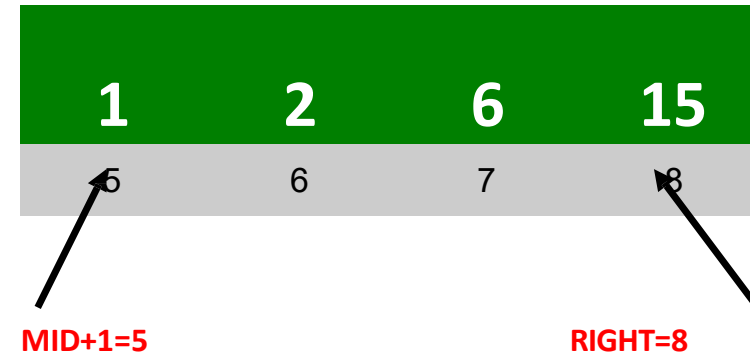
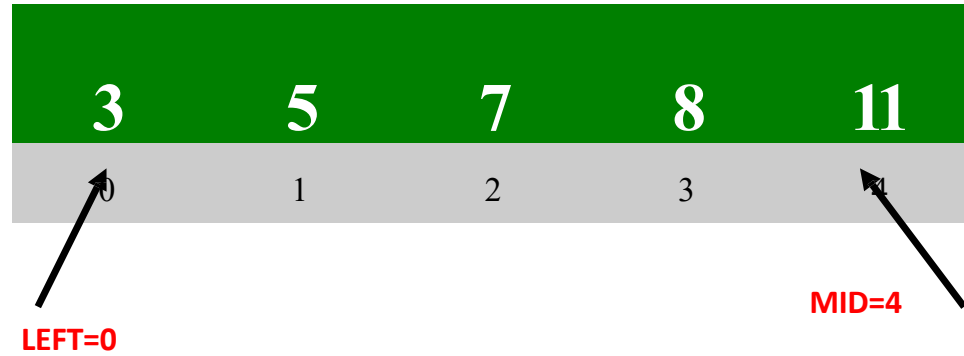


Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

# **SORT INDEPENDENTLY**



## MERGE



## Algorithm Mergesort(left,right)

---

1. if (left<right) *// more than 1 element*
  1. mid = (left+right)/2
  2. mergesort(left,mid) *// Divide*
  3. mergesort (mid+1,right)
  4. arraymerge (left,mid,right) *// Conquer*
2. endif

## HOW TO MERGE 2 SORTED ARRAYS?

---



**LEFT**



Temporary Array



## Process of Merging 2 Sorted Arrays

---



**LEFT**



## Process of Merging 2 Sorted Arrays

---



## Process of Merging 2 Sorted Arrays

---



**LEFT**



## Process of Merging 2 Sorted Arrays

---

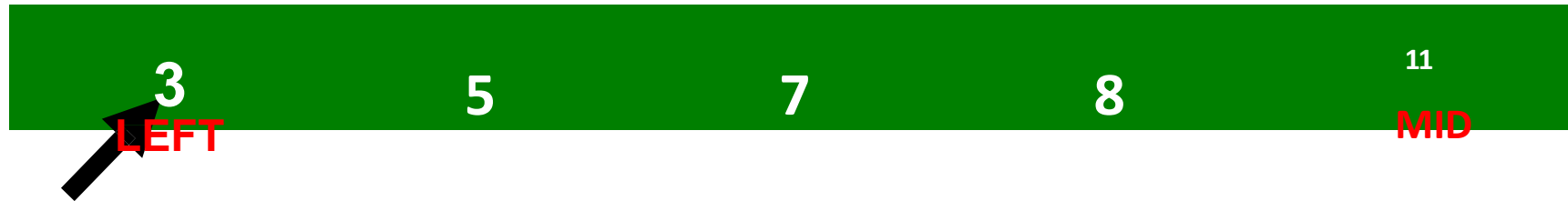


**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays



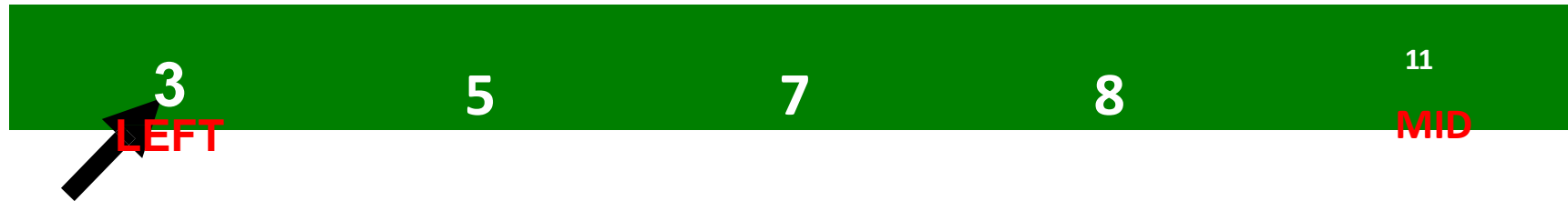
**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays

---

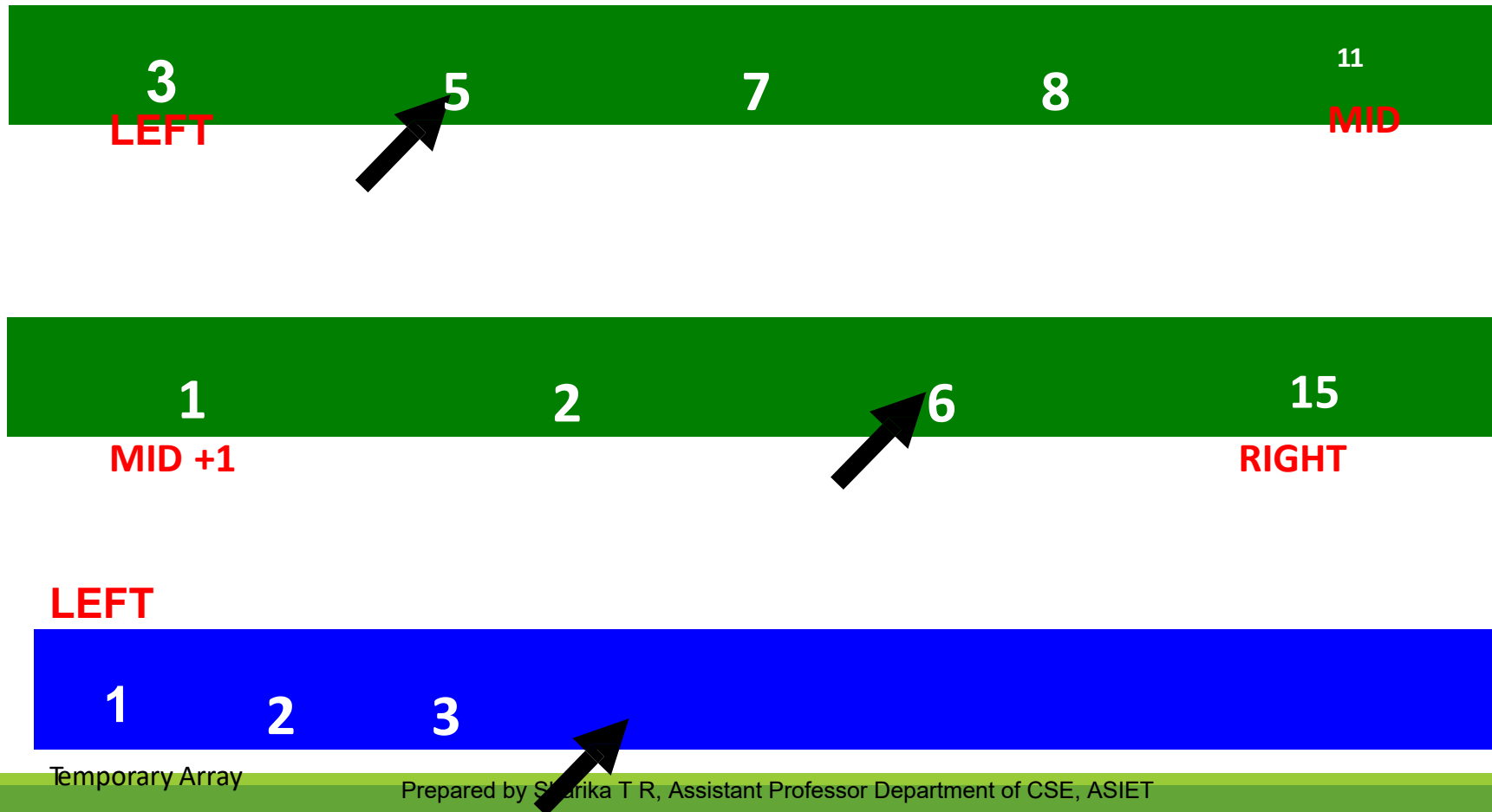


**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays



## Process of Merging 2 Sorted Arrays



**LEFT**

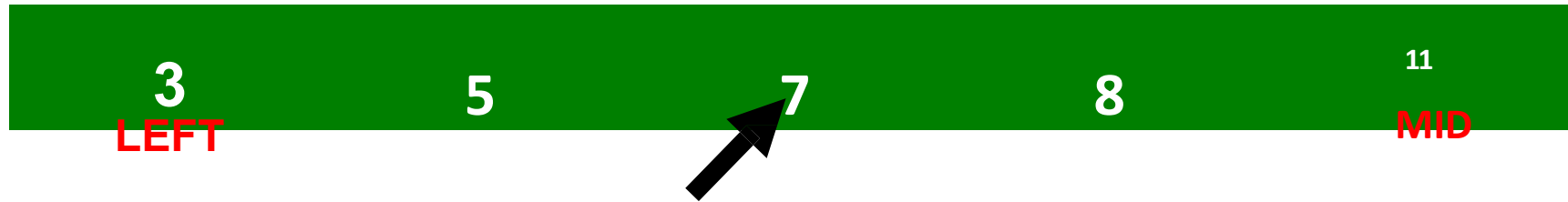


Temporary Array



## Process of Merging 2 Sorted Arrays

---



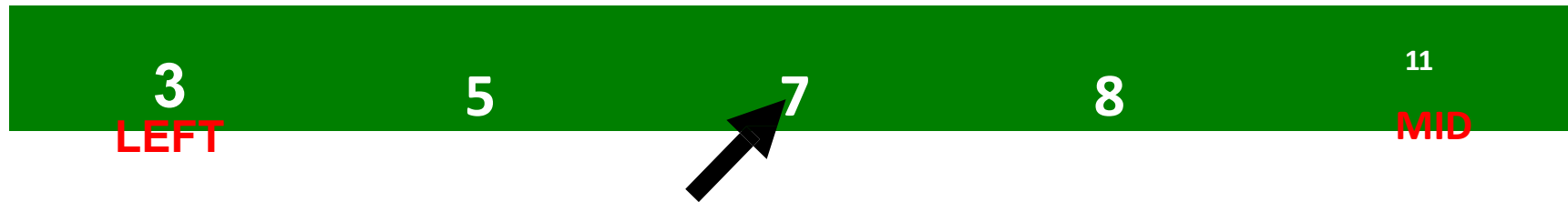
**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays

---



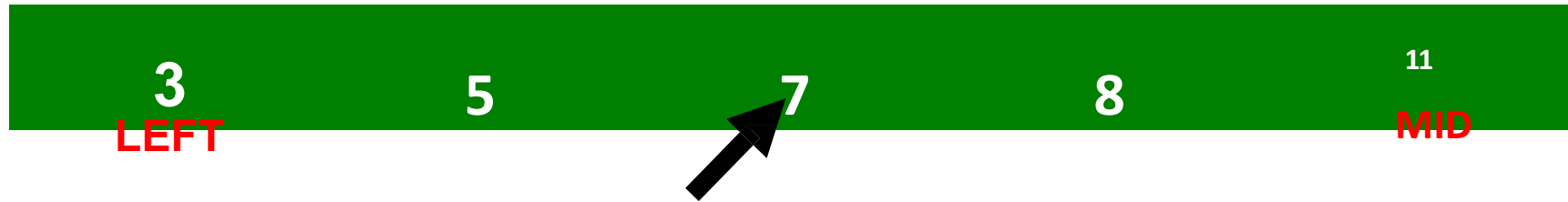
**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays

---

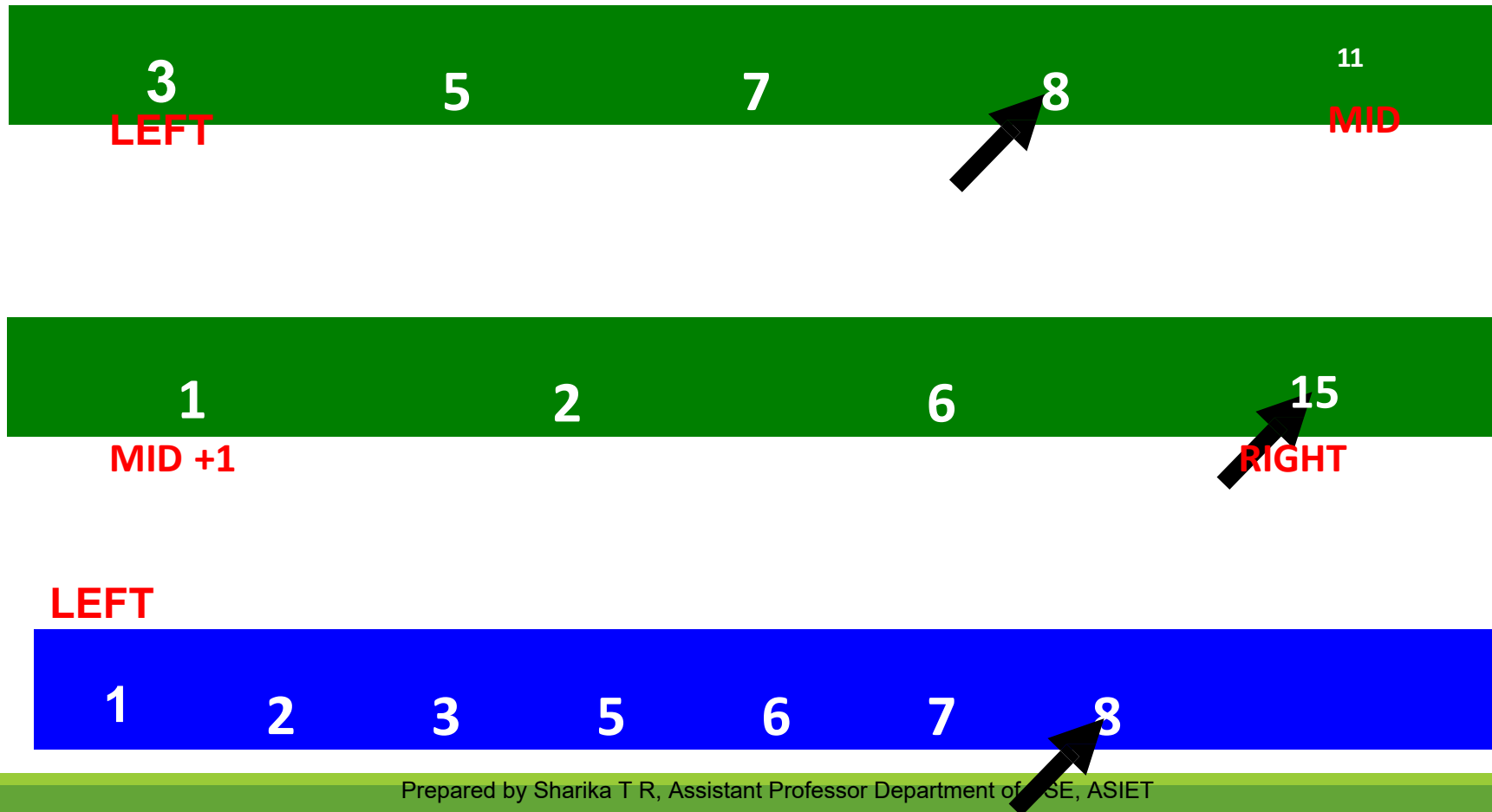


**LEFT**



## Process of Merging 2 Sorted Arrays

---



## Process of Merging 2 Sorted Arrays

---



**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays



**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays

---



**LEFT**



Temporary Array

## Process of Merging 2 Sorted Arrays



**LEFT**



Temporary Array

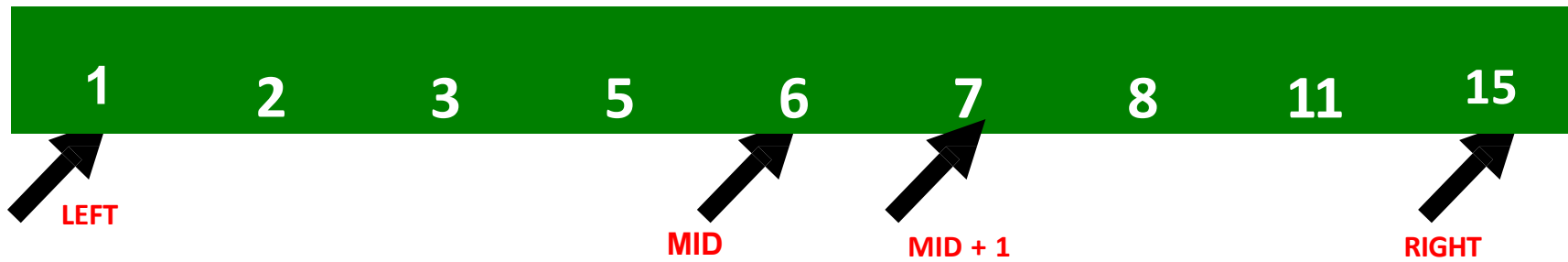


## Copy from temporary Array to original Array

---



Temporary Array



### Algorithm: ArrayMerge(left,mid,right)

```
1. set i=left, j=mid+1, k=left
2. while (i<=mid and j<=right)
    1. if (array[i]<array[j])
        1. temp[k]=array[i]
        2. i++
    2. else 1.
        temp[k]=array[j]
        2. j++
    3. endif
    4. k++
3. endwhile
4. while(i<=mid)
    1.
        temp[k]=array[i]
    2. i++
    3. k++
5. endwhile
6. while(j<=right)
    1.
        temp[k]=array2[j]
    2. j++
    3. k++
7. endwhile
8. i=left
9. while(i<=right)
    1.
        array[i]=temp[i]
    2. i++
10. endwhile
```

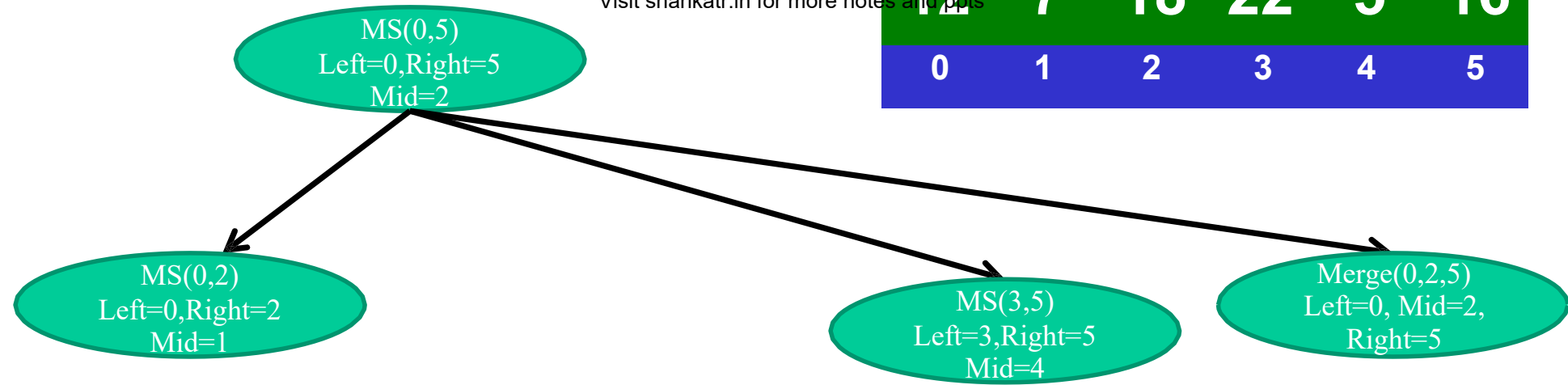
**Why set  $i=\text{left}$ ,  $j=\text{mid}+1$ ,  $k=\text{left}$**

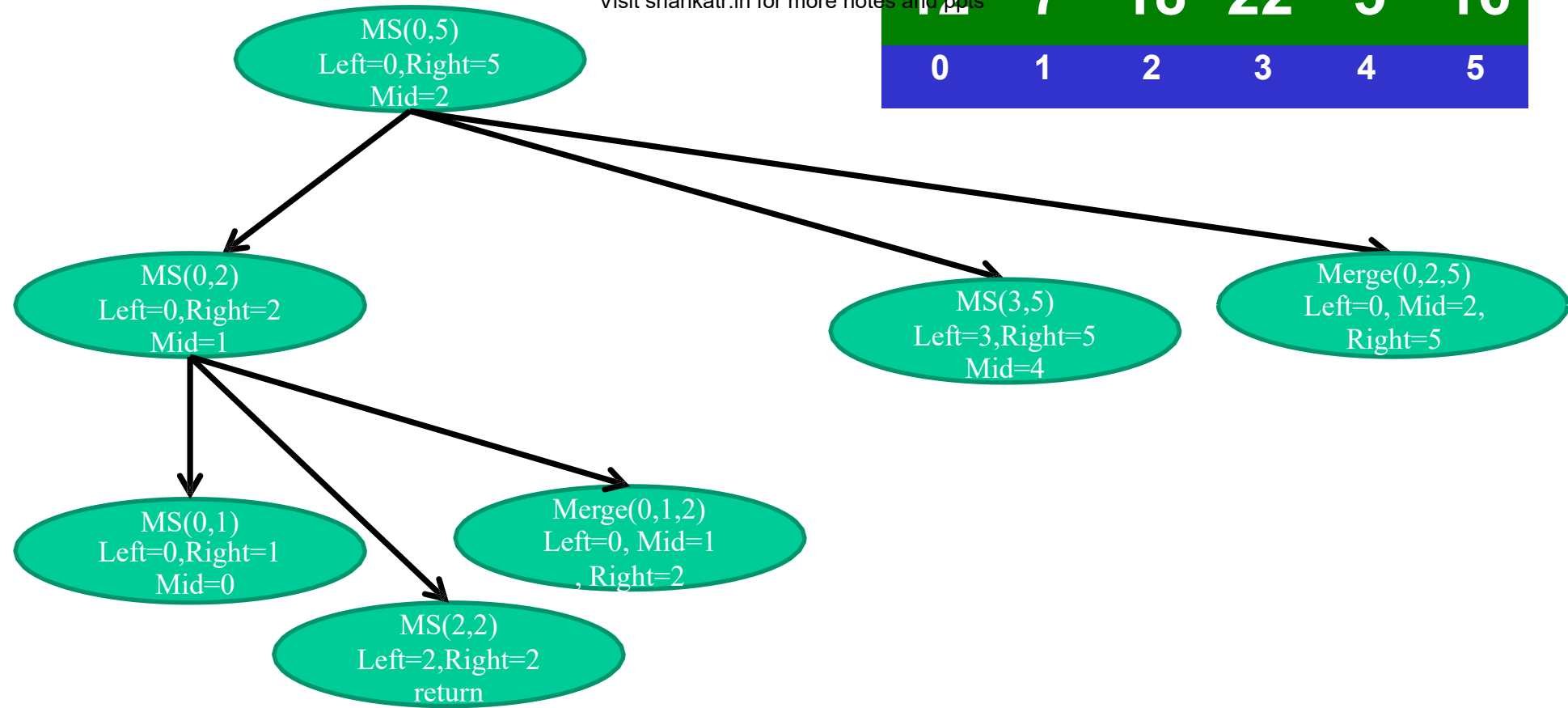
MS(0,5)  
Left=0,Right=5  
Mid=2

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppt's

12	7	18	22	5	16
0	1	2	3	4	5

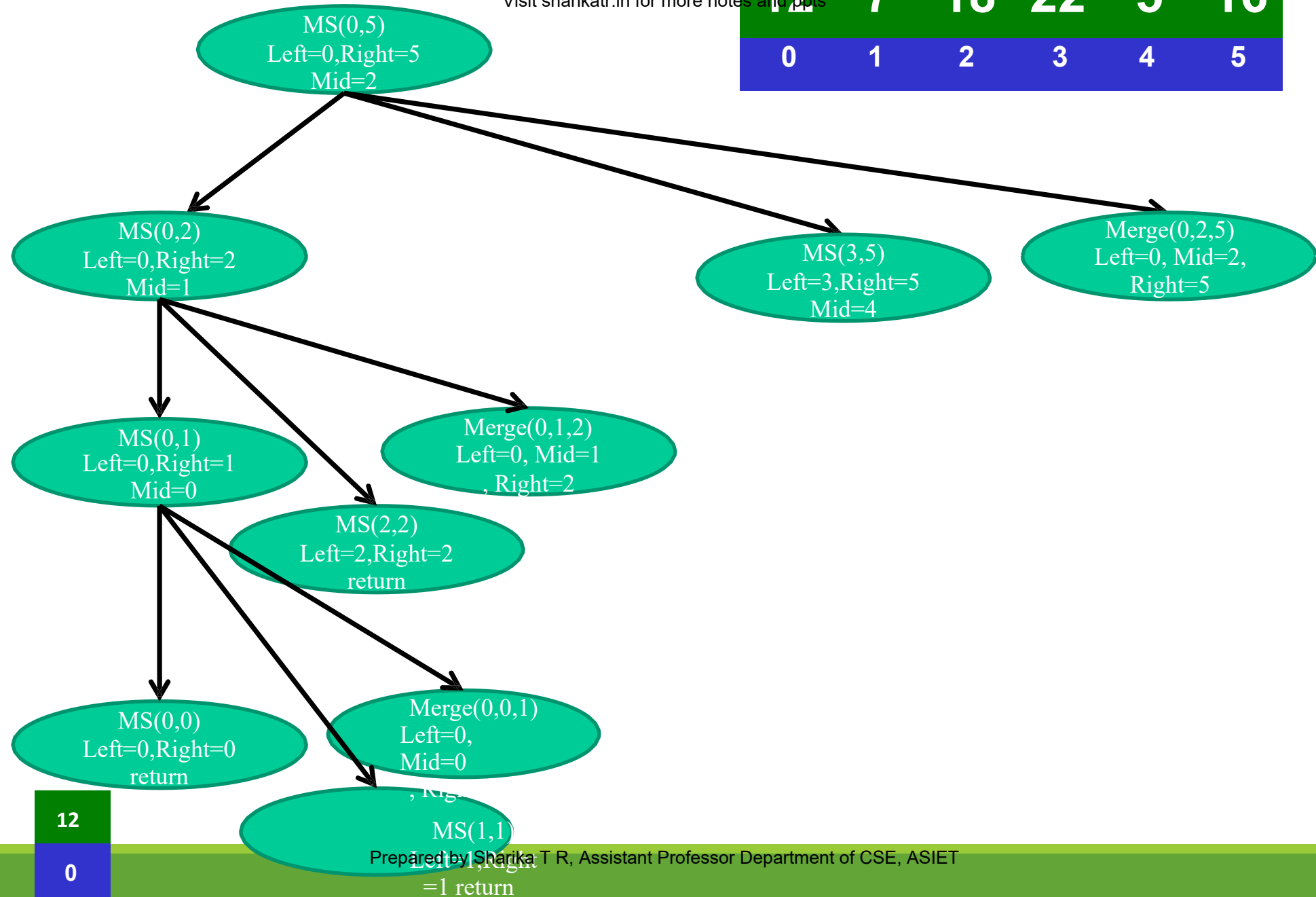
12	7	18	22	5	16
0	1	2	3	4	5





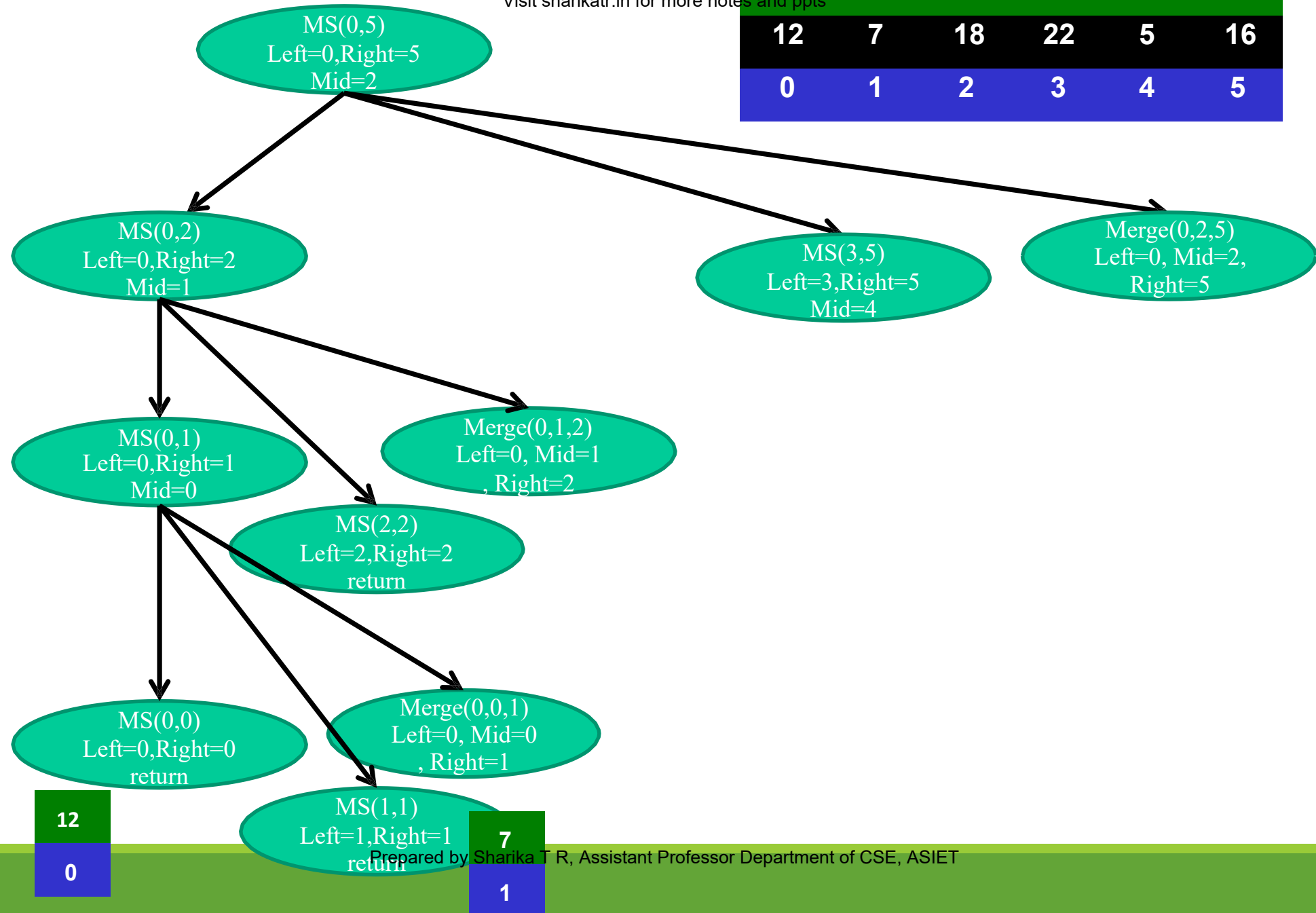


12	7	18	22	5	16
0	1	2	3	4	5

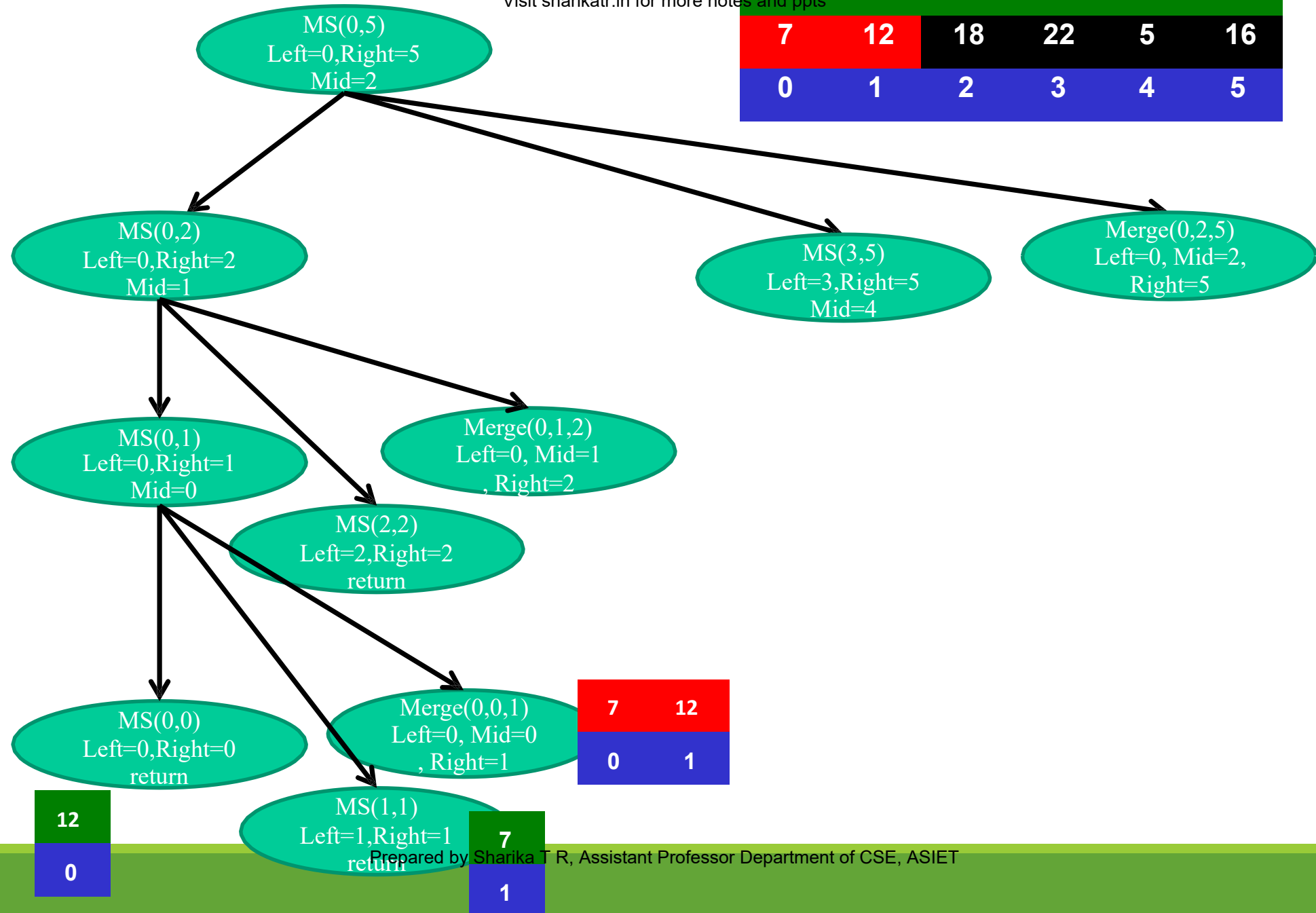




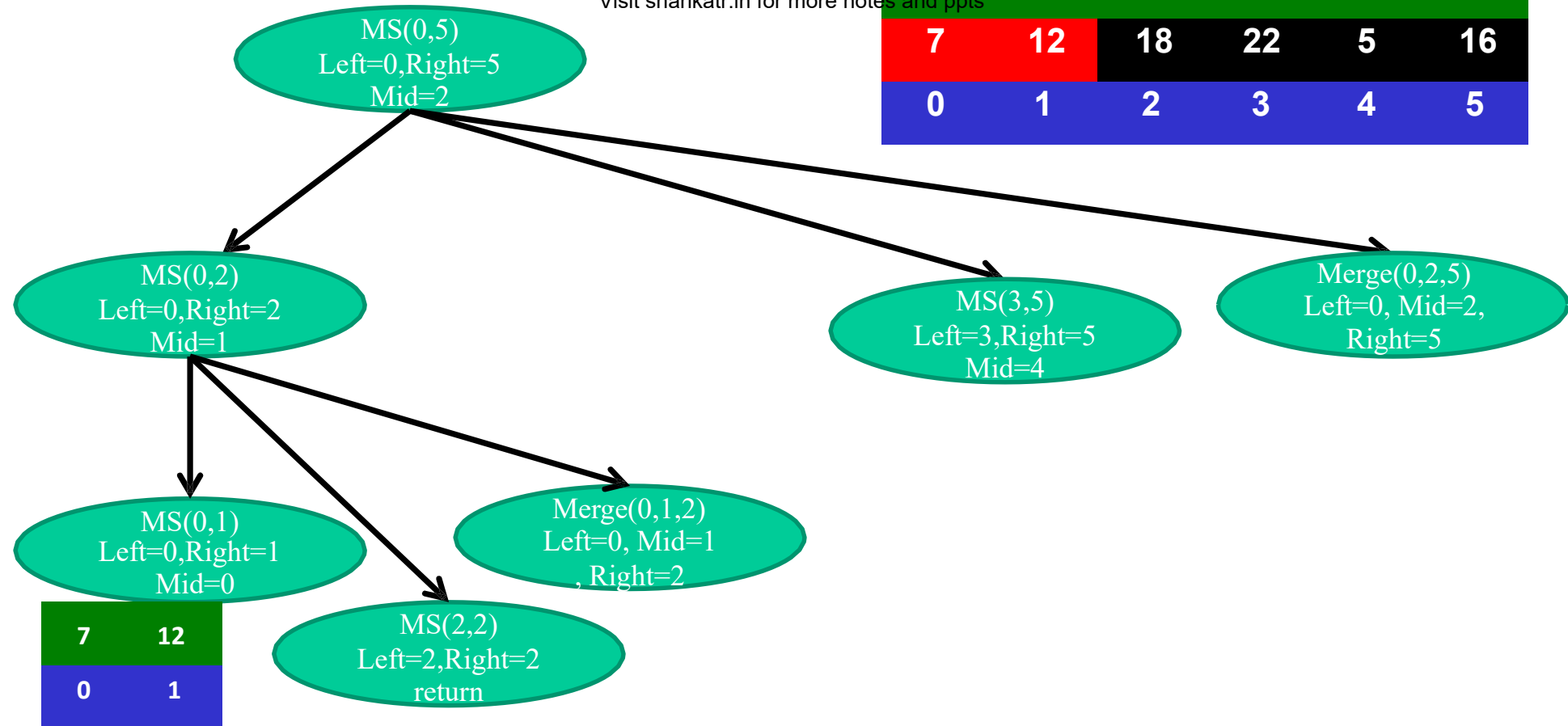
12	7	18	22	5	16
12	7	18	22	5	16
0	1	2	3	4	5



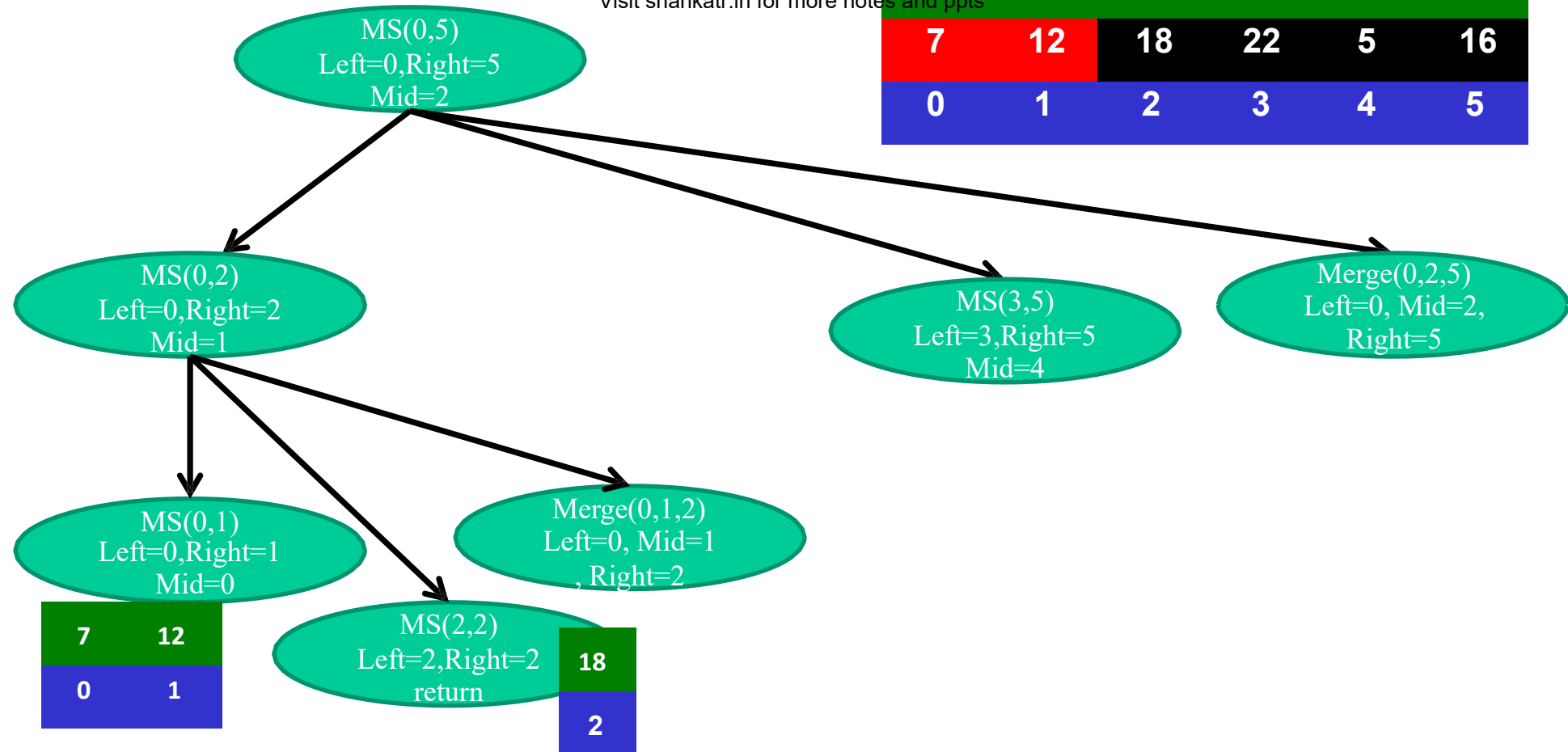
12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

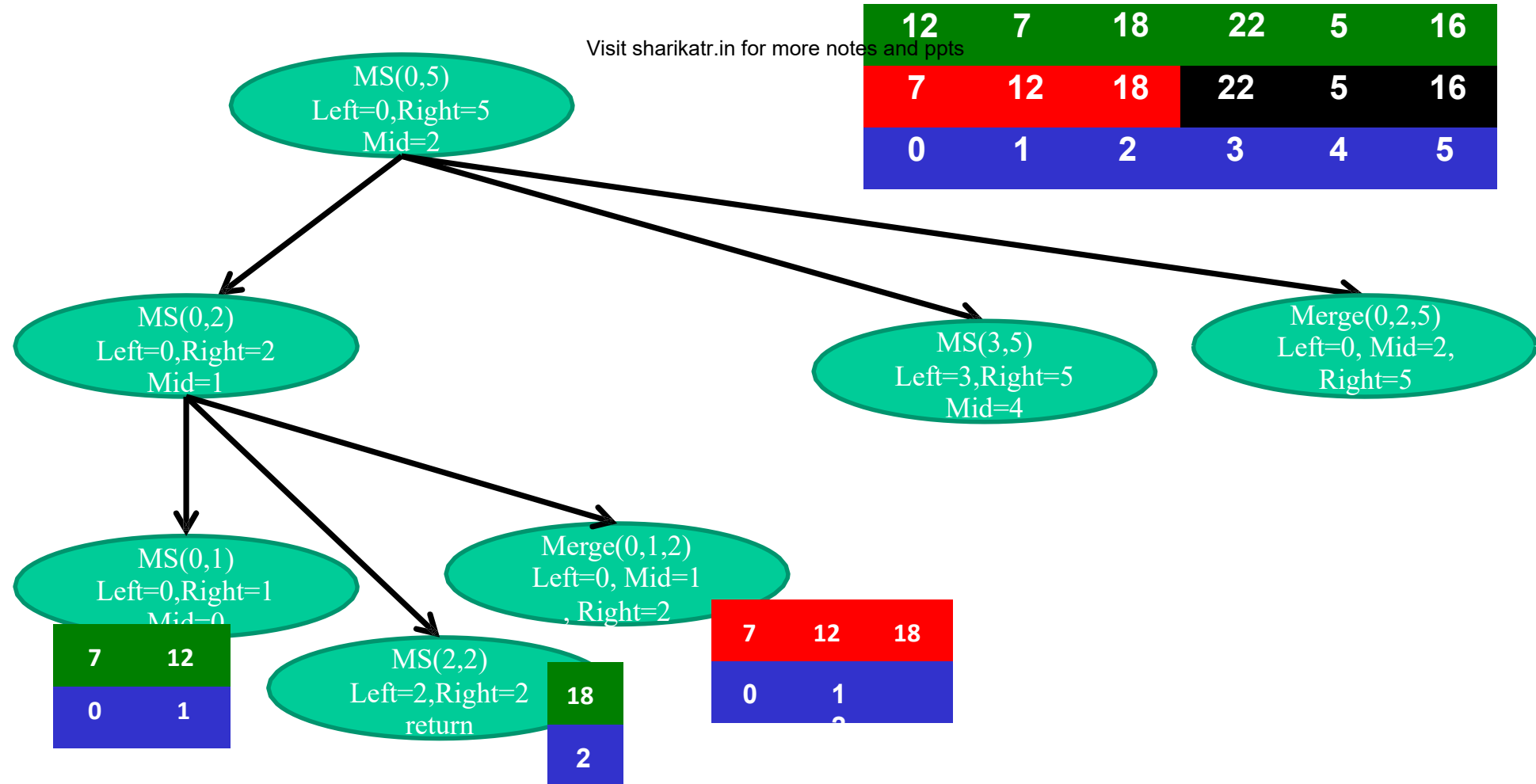


12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

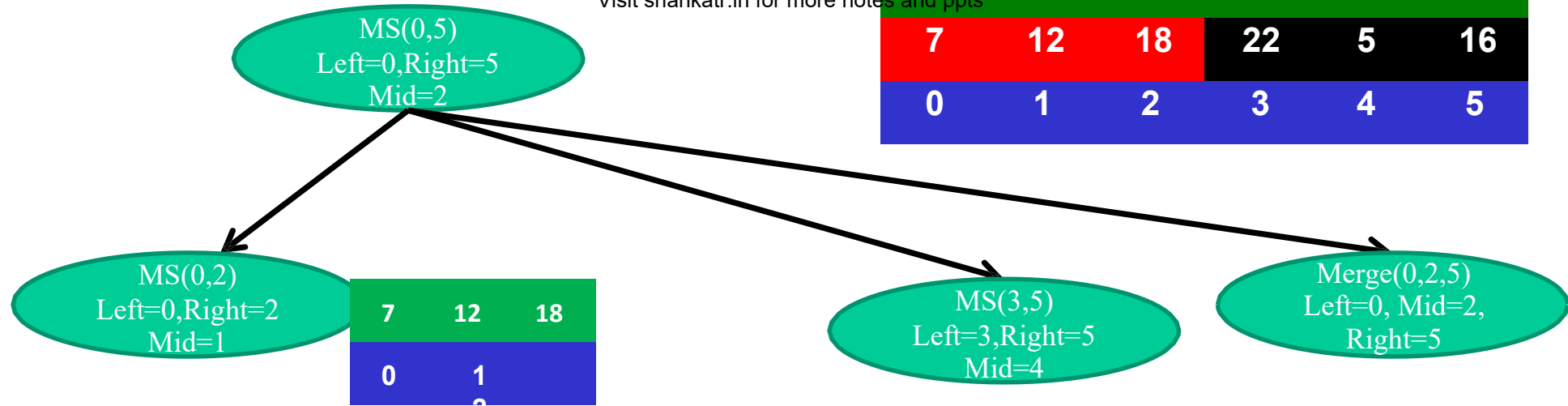


12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

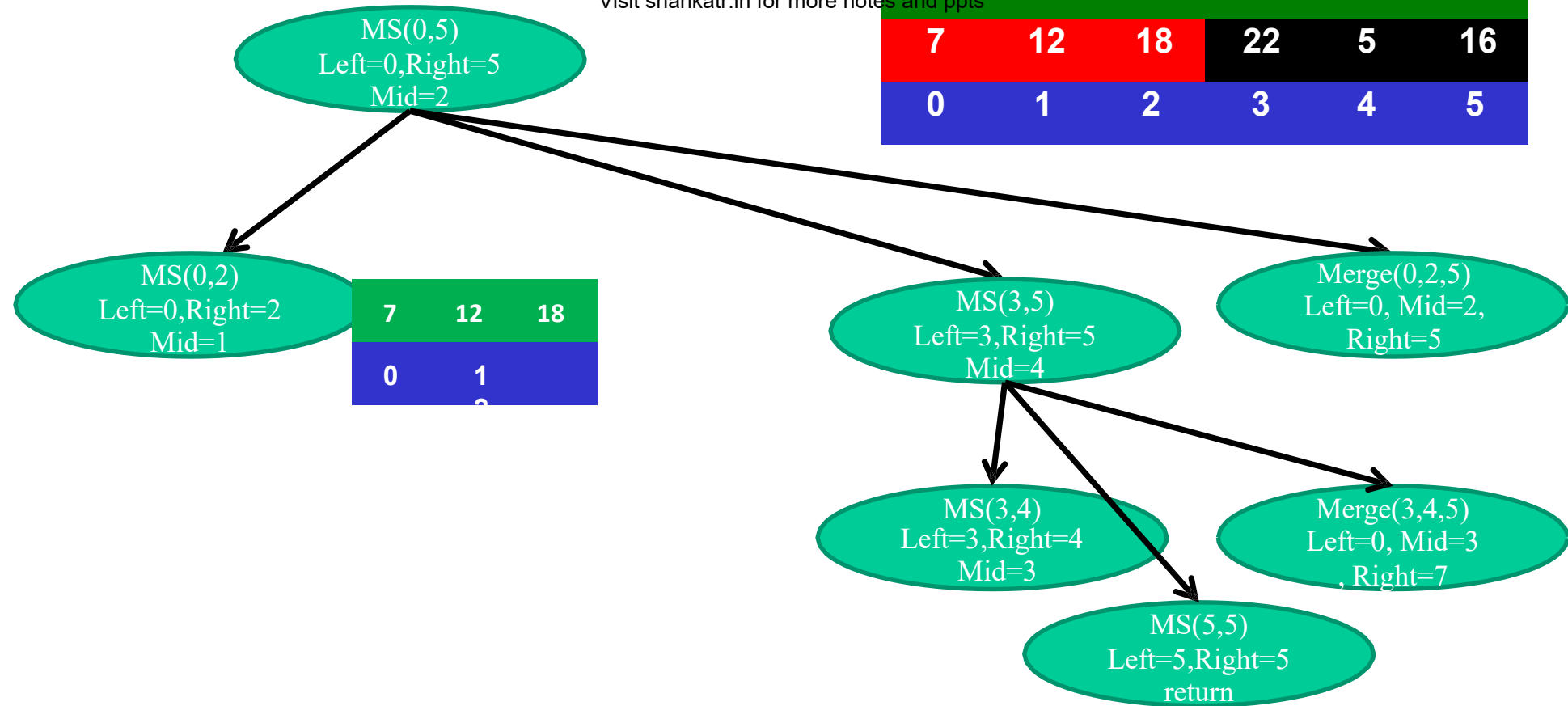




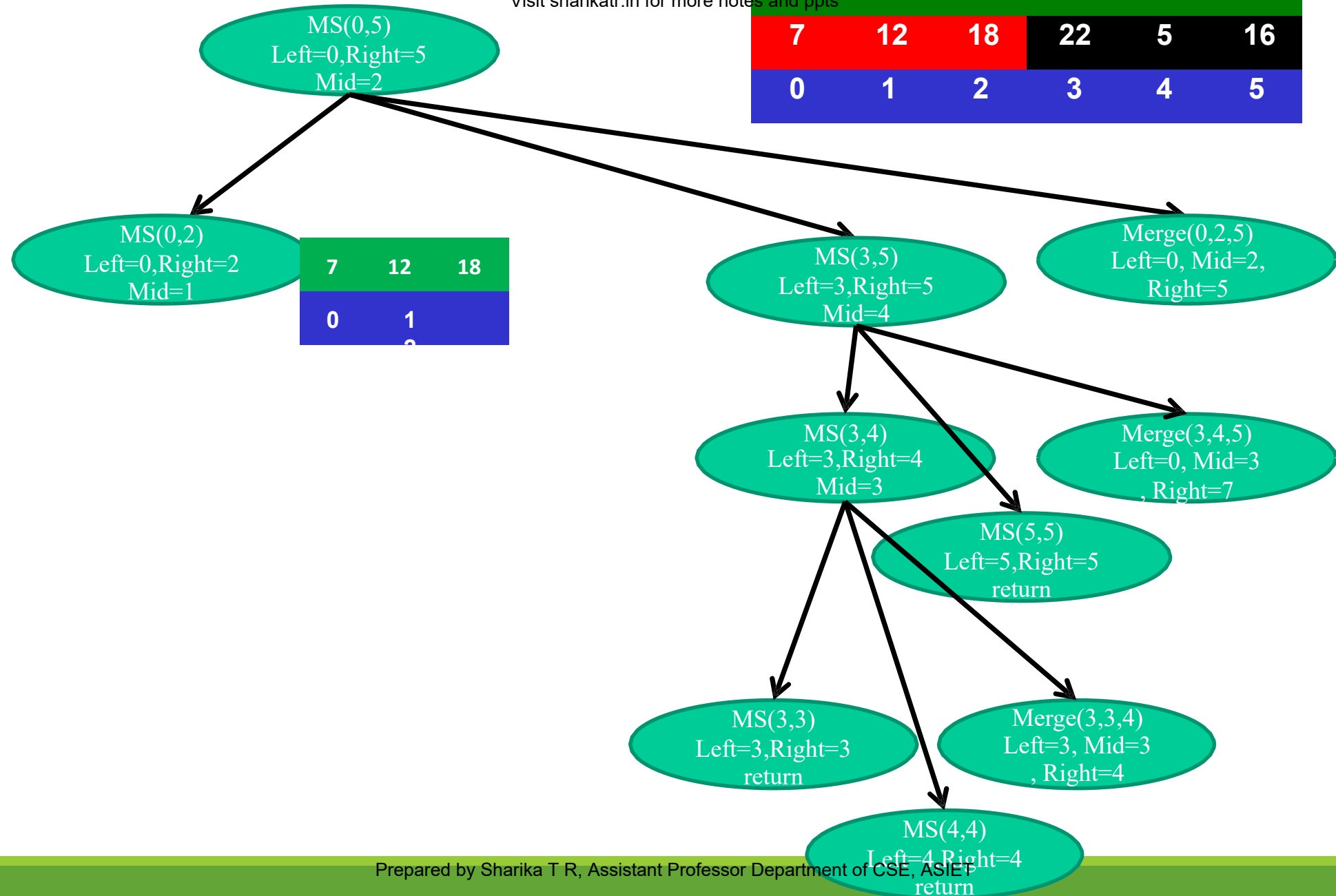
12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5



12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

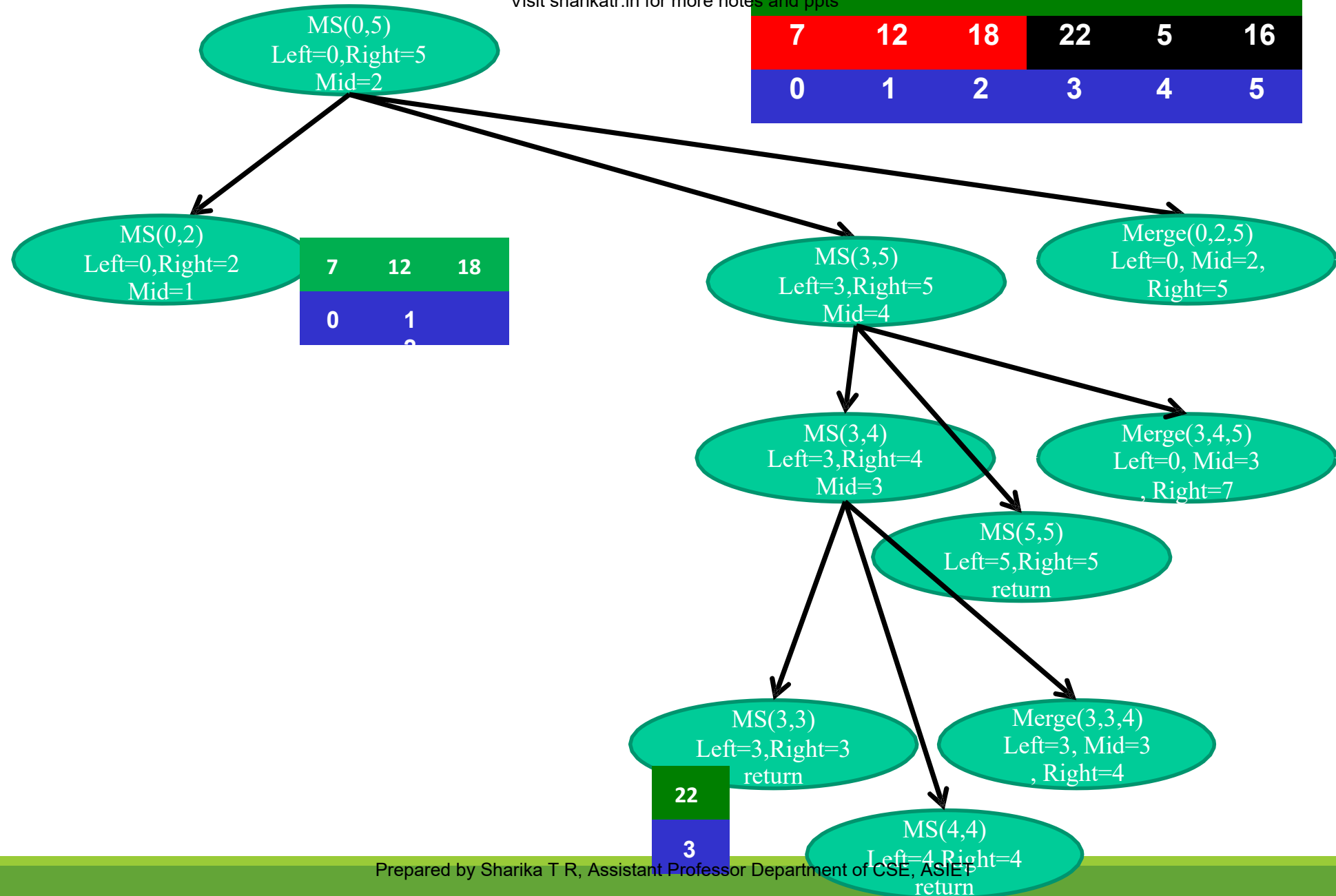


12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

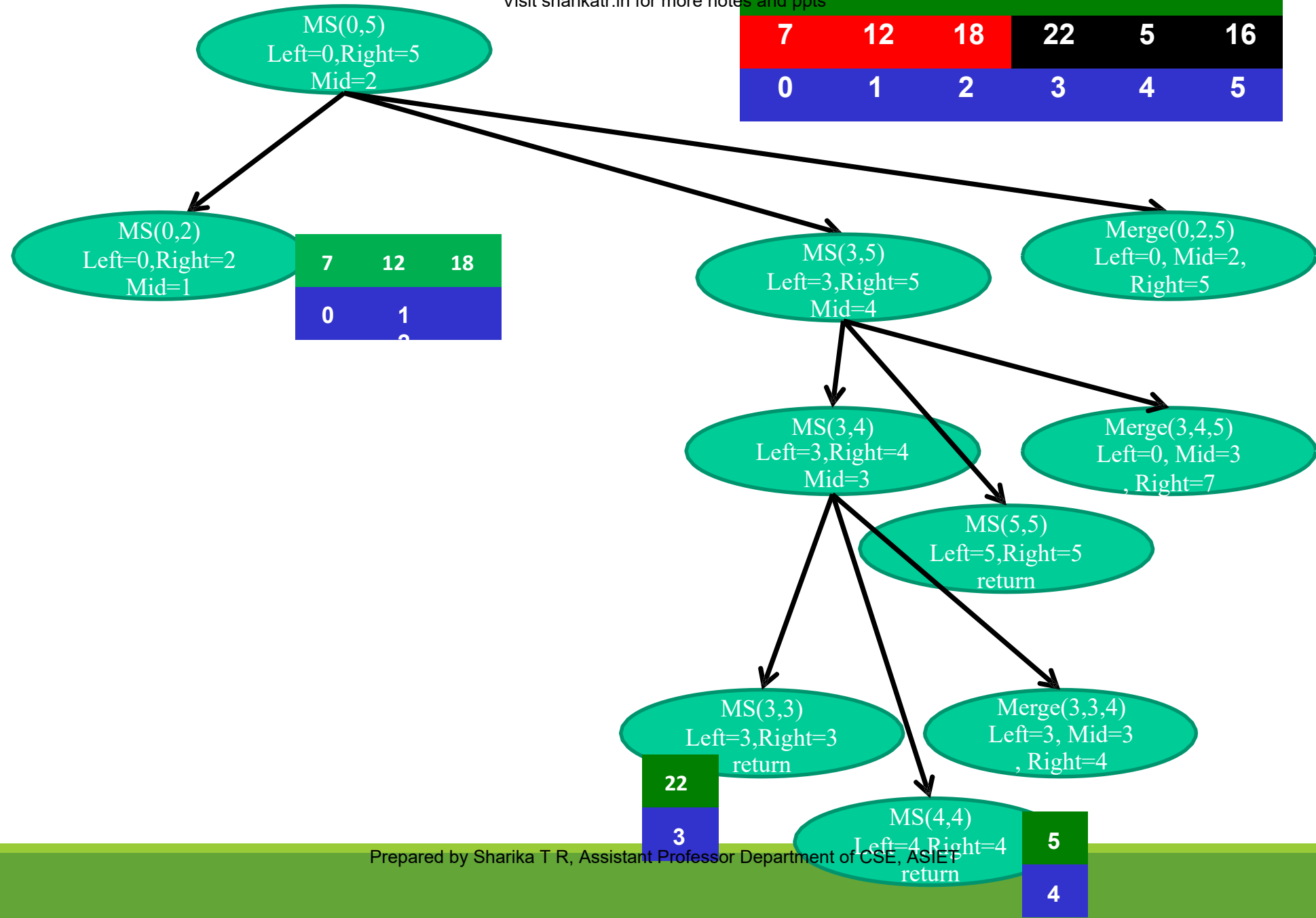




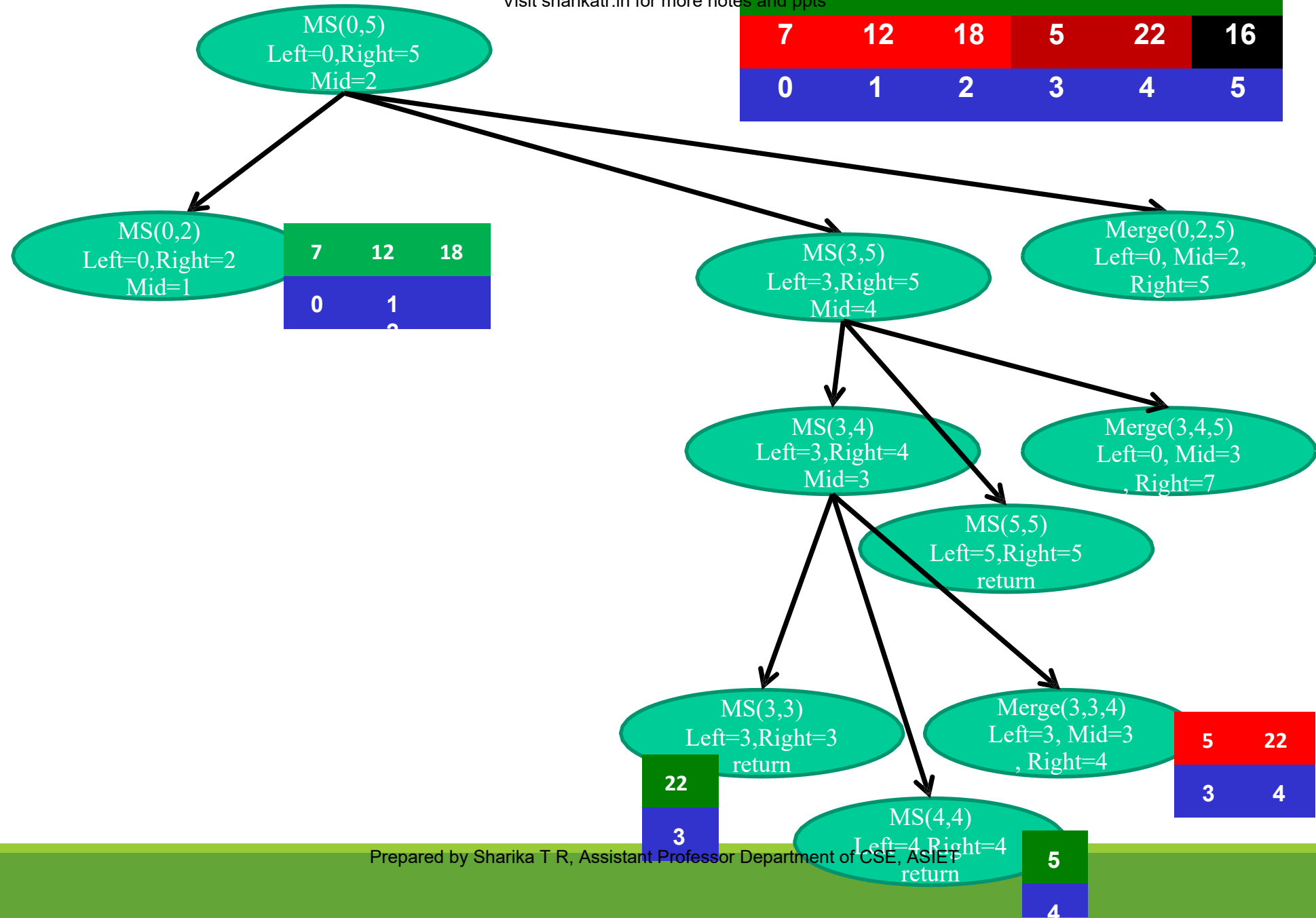
12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

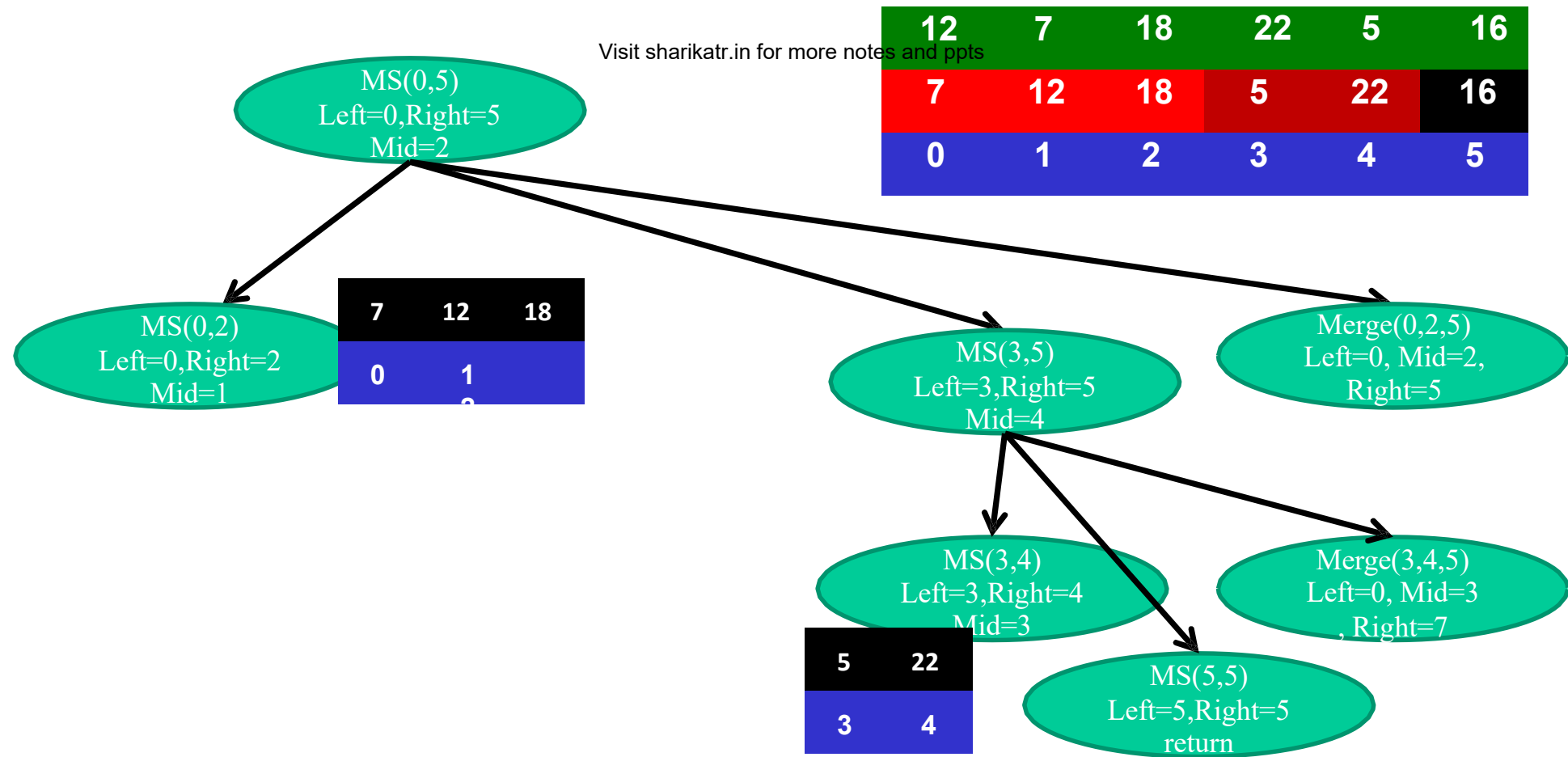


12	7	18	22	5	16
7	12	18	22	5	16
0	1	2	3	4	5

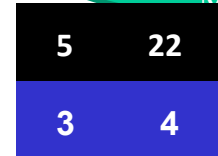


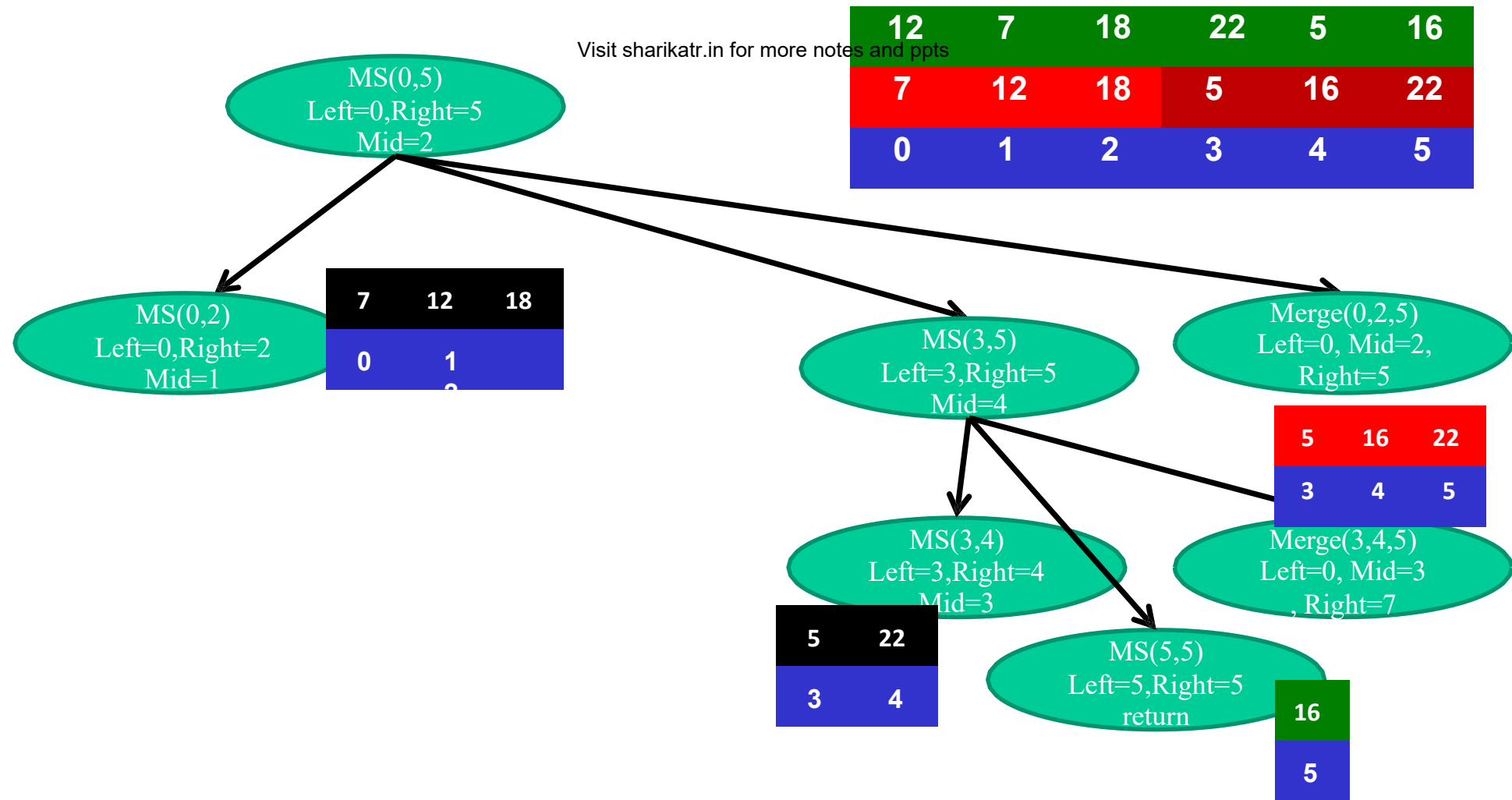
12	7	18	22	5	16
7	12	18	5	22	16
0	1	2	3	4	5

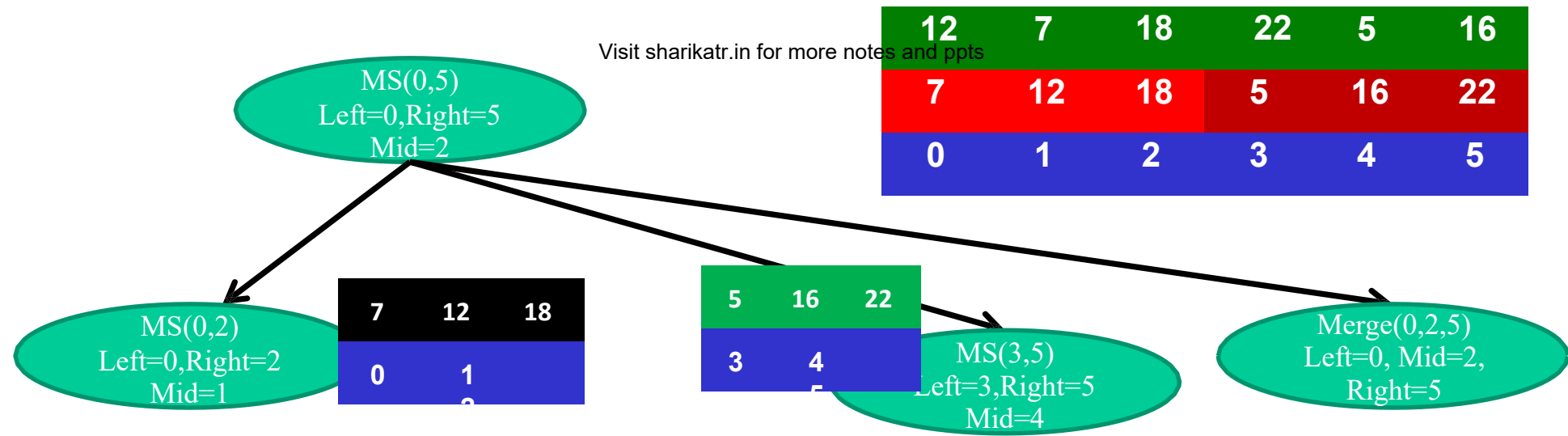


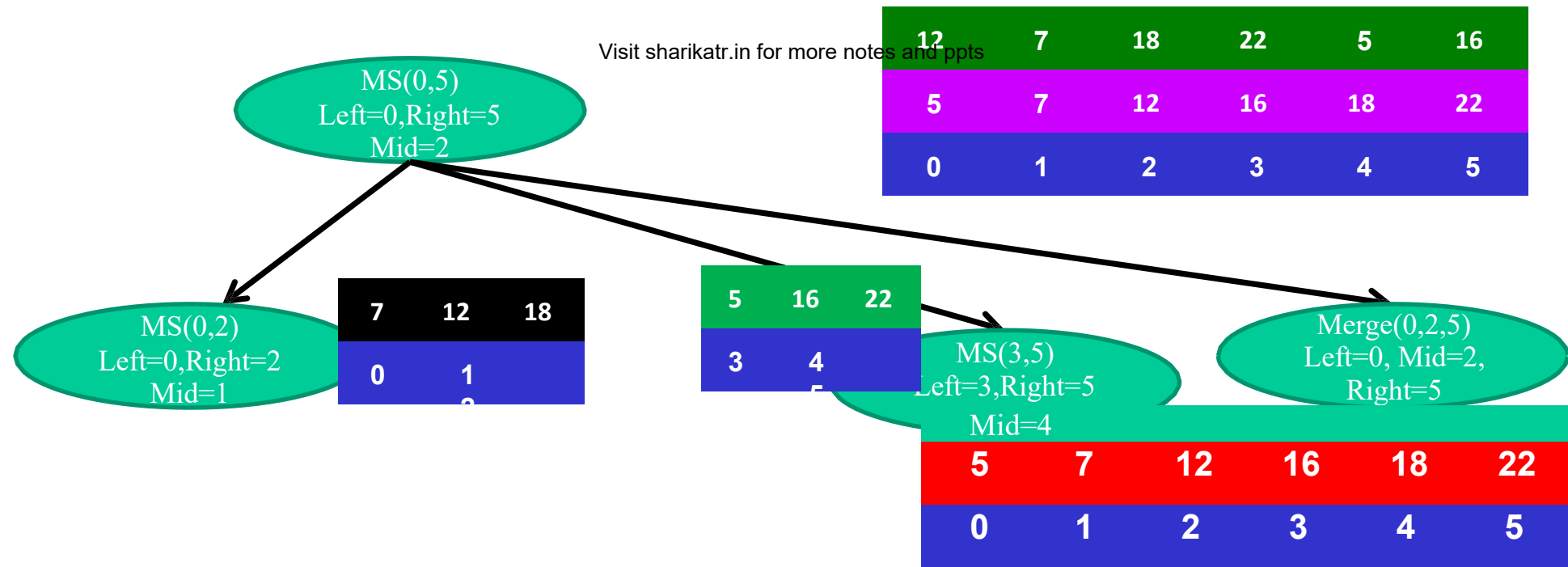


12	7	18	22	5	16
7	12	18	5	22	16
0	1	2	3	4	5

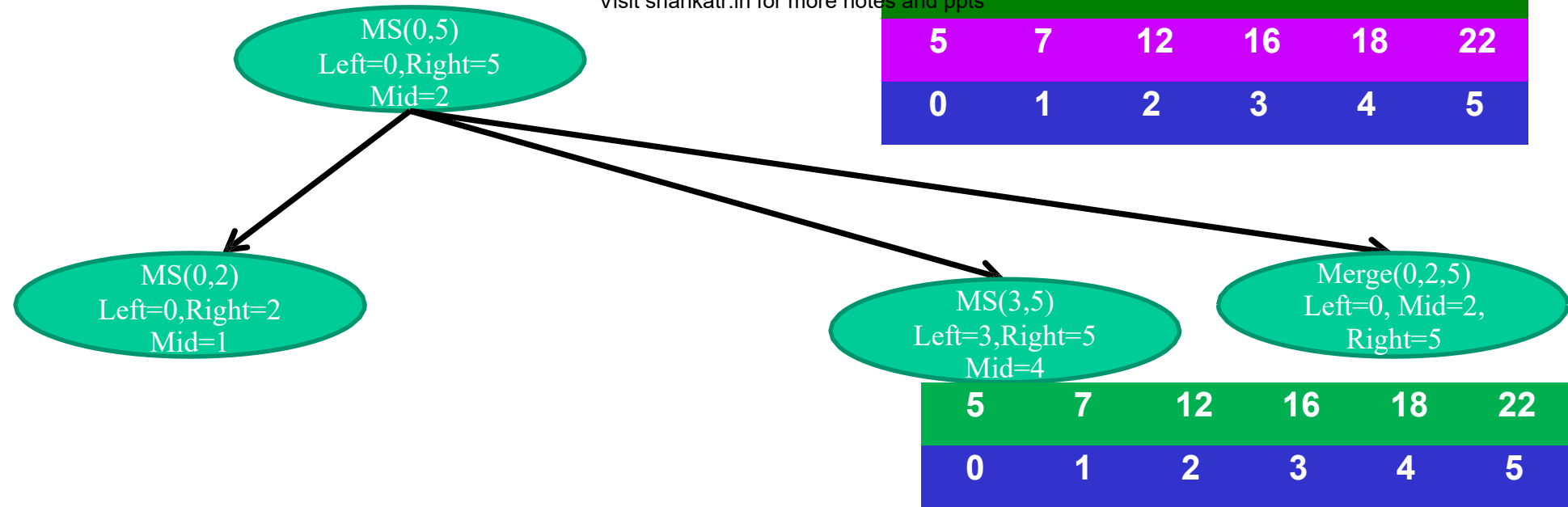


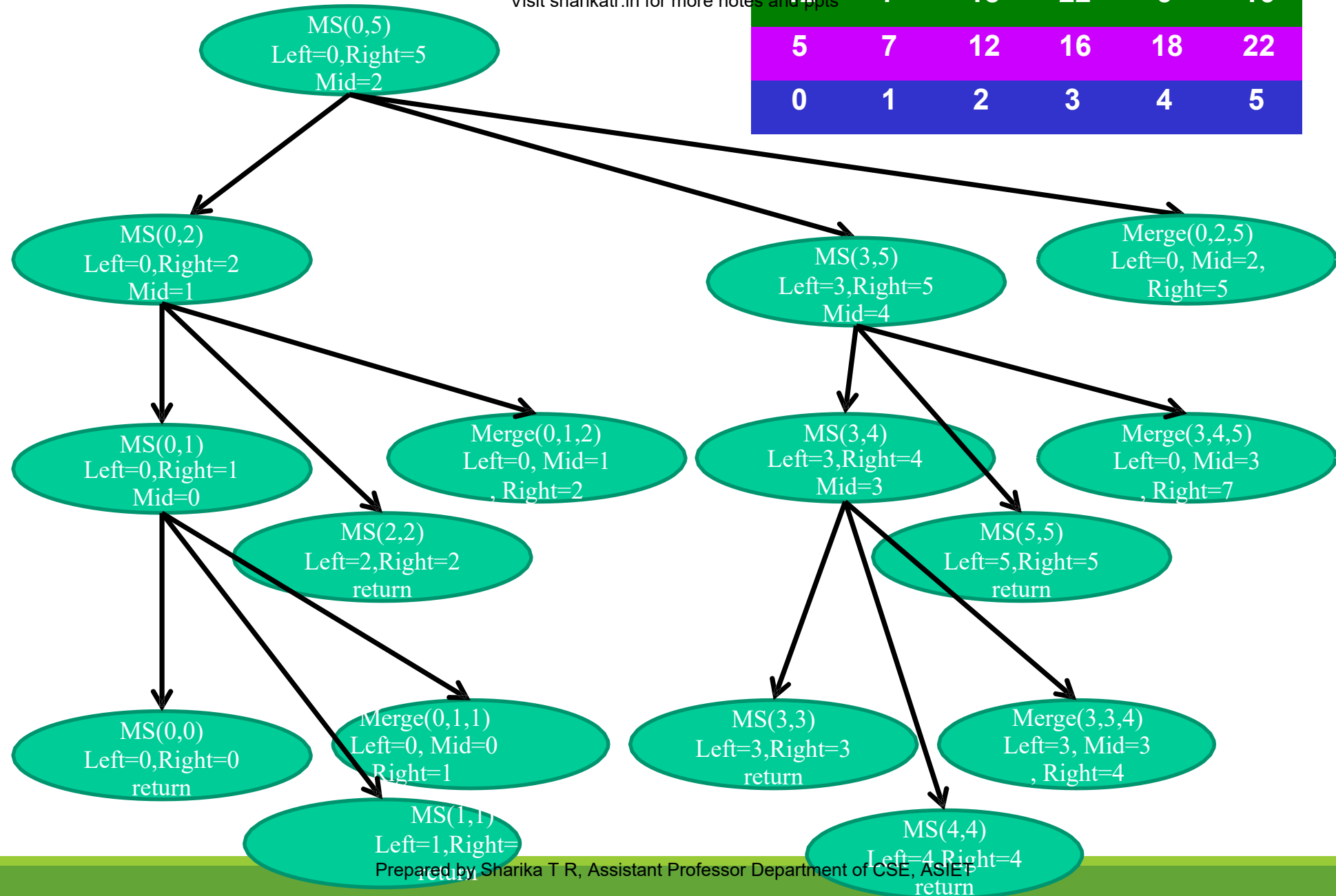












**SORT 9, 8, 7, 6, 5, 4, 3, 2, 1**

# HEAP SORT

---

Heap sort is a sorting algorithm that organizes elements in an array into a binary heap, and then sorts that heap by moving the largest element in the array.

# Algorithm for Heap Sort

---

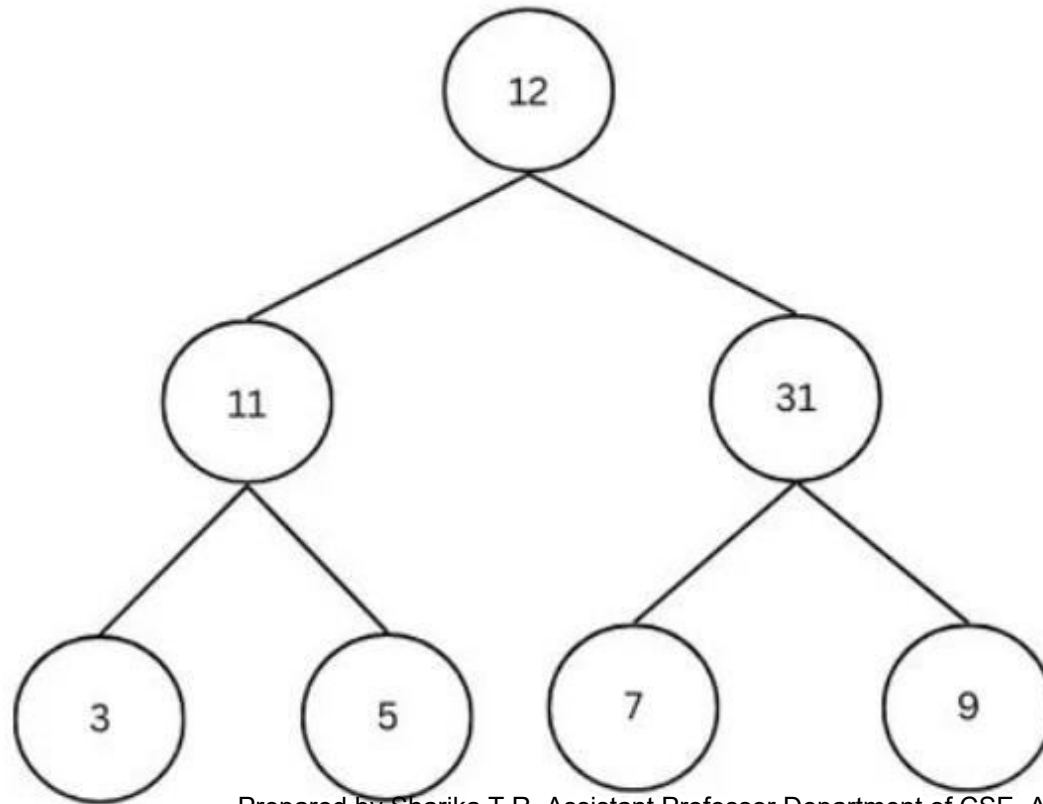
1. Transform the array into a binary tree by inserting each element as a node in a breadth-first manner.
2. Convert the binary tree into a max heap, ensuring that all parent nodes are greater than or equal to their child nodes.
3. Swap the root node — the largest element — with the last element in the heap.
4. Call the `heapify()` function to restore the max heap property.
5. Repeat steps 3 and 4 until the heap is sorted, and exclude the last element from the heap on each iteration.
6. After each swap and `heapify()` call, ensure that the max heap property is satisfied.

[12, 11, 31, 3, 5, 7, 9]

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

we can start by inserting the first element of the array as a node into the tree and continuing this process in a breadth-first manner until all elements of the array have been added.

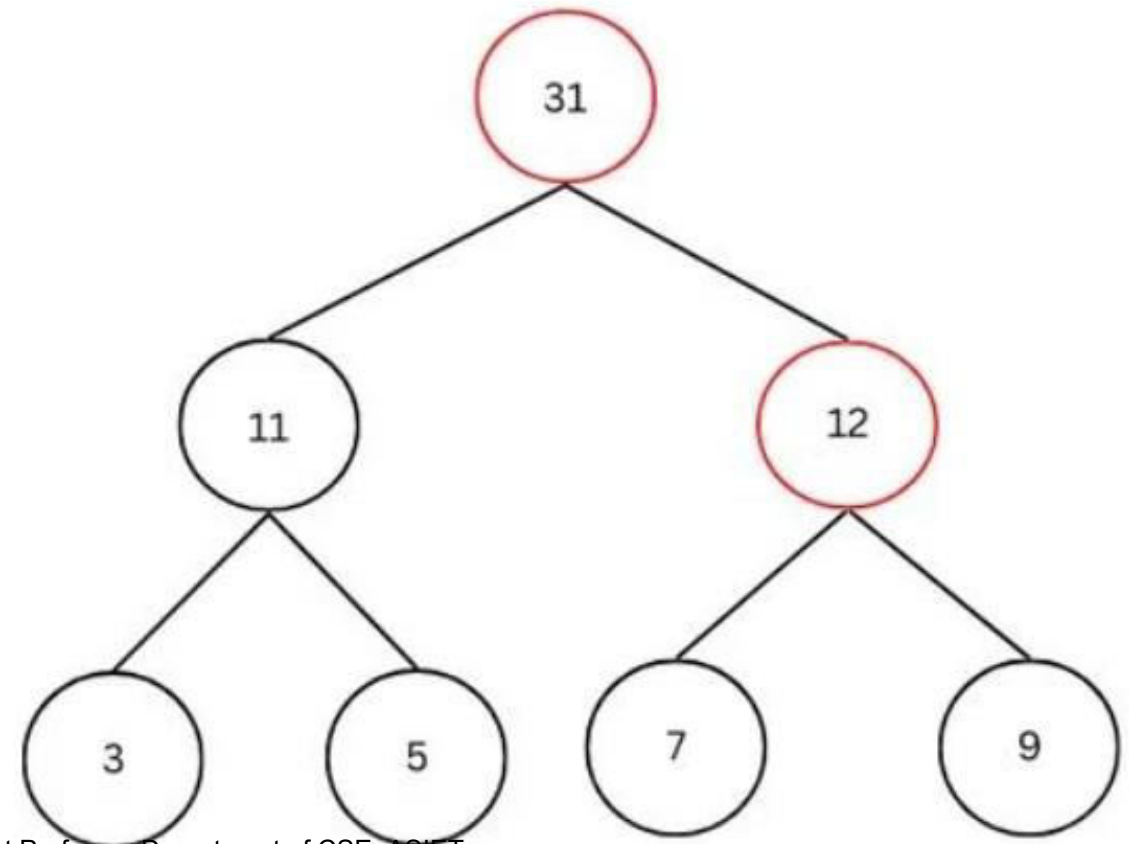
---



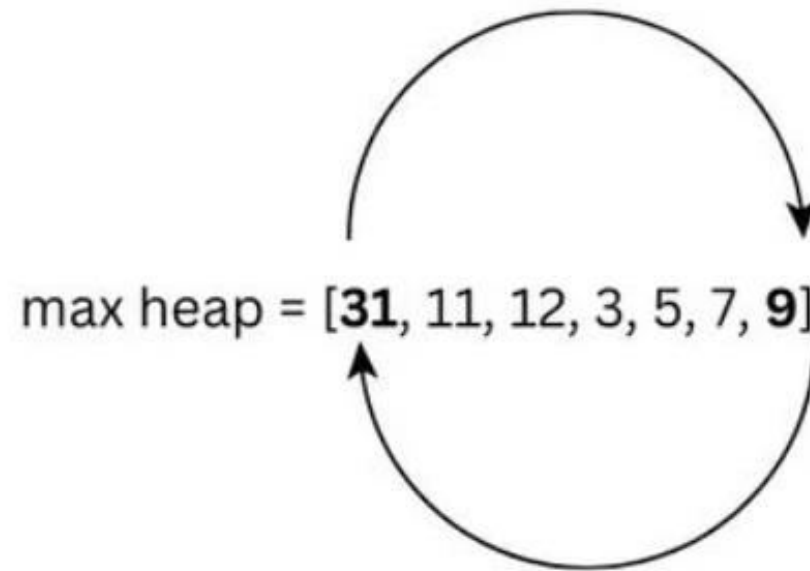
**2. Convert the Binary Tree Into a Max Heap:** all parent nodes must have values that are greater than or equal to the values of their children. this means swapping node 12 and node 31 positions in the tree to satisfy the requirements for a max-heap tree

---

max heap = [31, 11, 12, 3, 5, 7, 9]



**3. Swap the Root Node With the Last Element in the Heap:** The next step in the sorting process is to swap the root node, which contains the largest element, with the last node in a heap. In other words, you're swapping the element in the first position of the max-heap array with the element in the last position of the max-heap array. Whatever method you choose to understand this step, 31 ends at the end of the array, and nine ends at the first position of the array



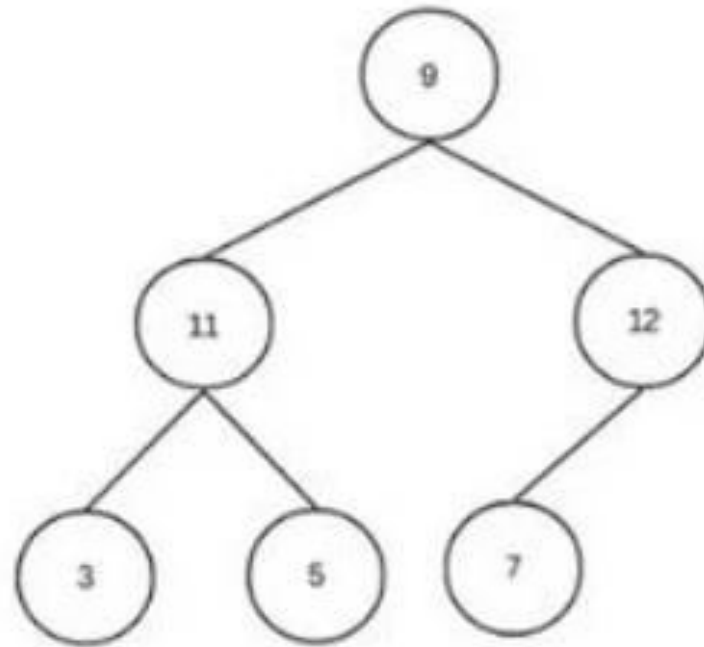
`[9, 11, 12, 3, 5, 7, 31]`



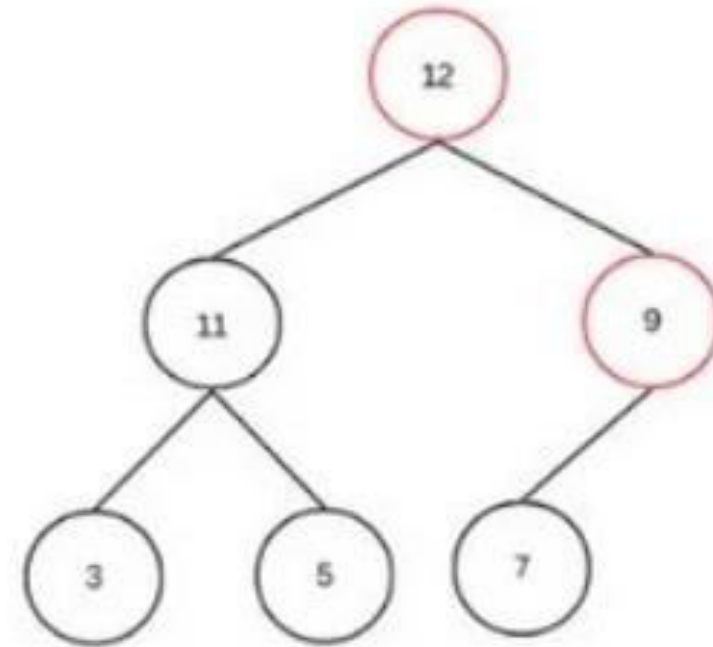
in future iterations, we will be omitting the last value because it's in a sorted position. Therefore, we move forward with the following array into the next step:

[9, 11, 12, 3, 5, 7]

Now, we will transform the array into a tree, then the tree into a max heap.



Binary Tree



Max-Heap

---

After swapping the root node with the last element in the heap, the visualized heap will now have one less element.

This is because the largest element has been placed at the end of the array and will be excluded from future iterations.

If it were not excluded, the sorting algorithm would never finish.

Therefore, every time the root node is swapped, the next iteration should exclude the last element in the heap.

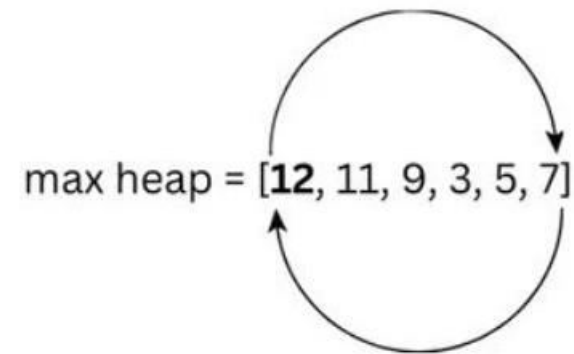
## 4. Call the heapify() Function

---

Let's now refer to the process of converting the tree or array into a max heap as heapify

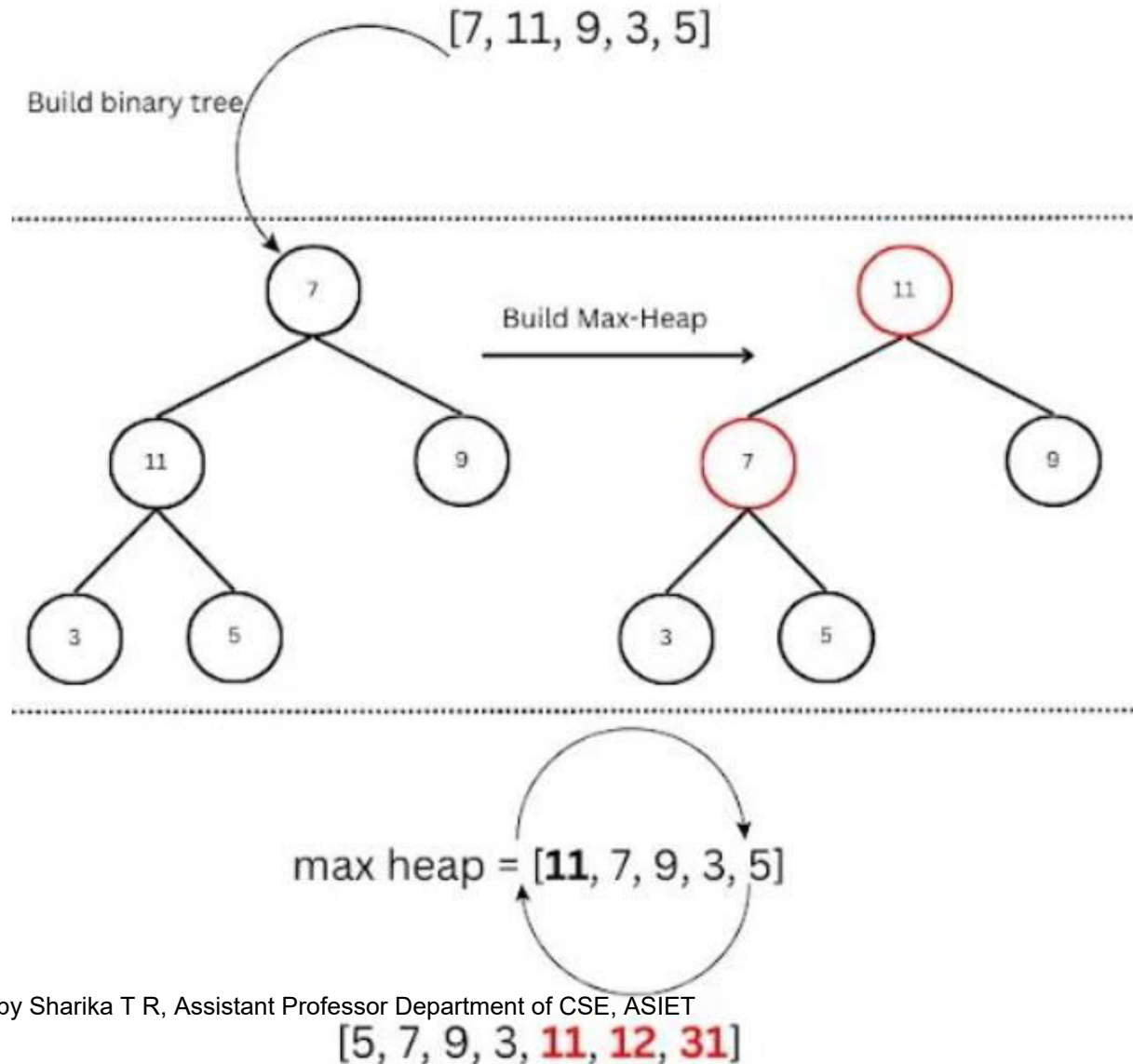
The heapify() function should be called again to restore the max heap property. This will result in the heap being rearranged as shown:

[12, 11, 9, 3, 5, 7]



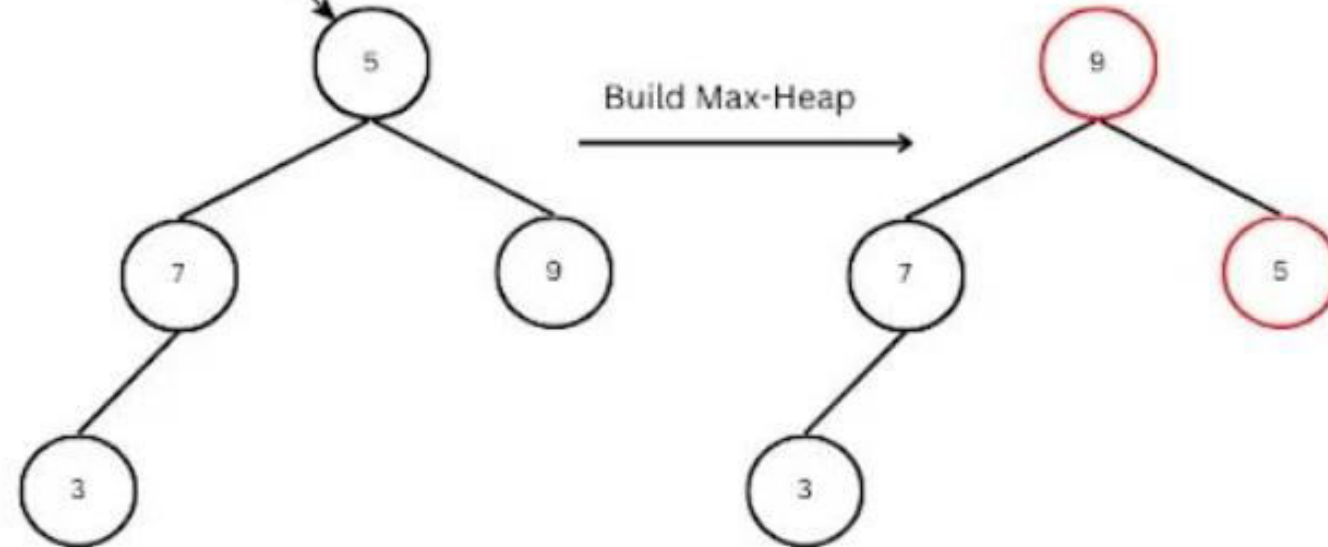
[7, 11, 9, 3, 5, 12, 31]

# 5. Repeat Steps 3 and 4 Until the Heap Is Sorted



Build binary tree

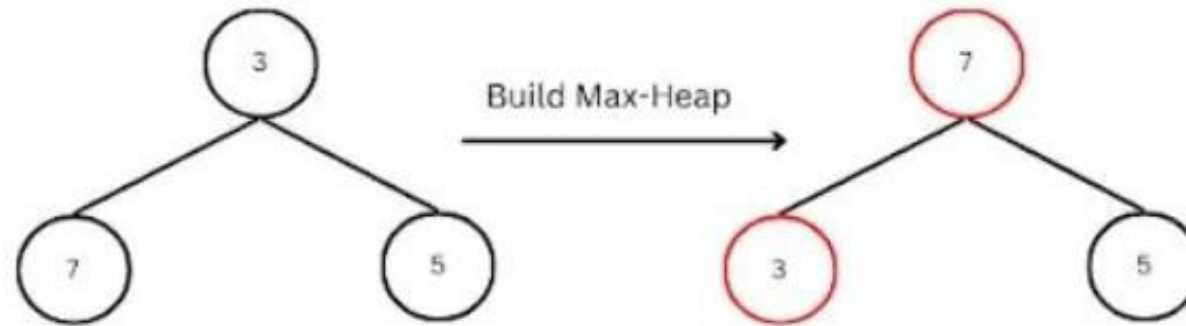
[5, 7, 9, 3]



max heap = [9, 7, 5, 3]

Build binary tree

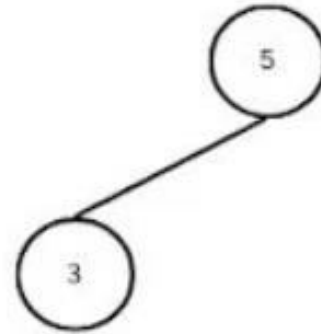
[3, 7, 5]



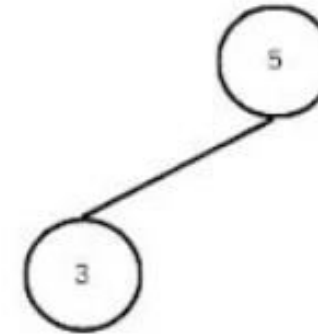
max heap = [7, 3, 5]

Build binary tree

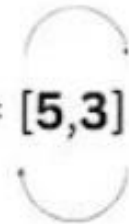
[5, 3]



Build Max-Heap



max heap = [5, 3]



The final sorted array:  
[3, 5, 7, 9, 11, 12, 31]

---

The heap sort algorithm's best, worst and average **time complexities** are all the same —  **$O(n \cdot \log(n))$** .



---

Heap sort is called **an in-place algorithm** because it does not require extra memory space to sort. It uses the same array for both the elements' storage and the sorting process. This is done by rearranging the elements of the array in place to satisfy the max-heap property, which is used to sort the array.

# RADIX SORT

---

**Radix Sort** is a linear sorting algorithm (for fixed length digit counts) that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

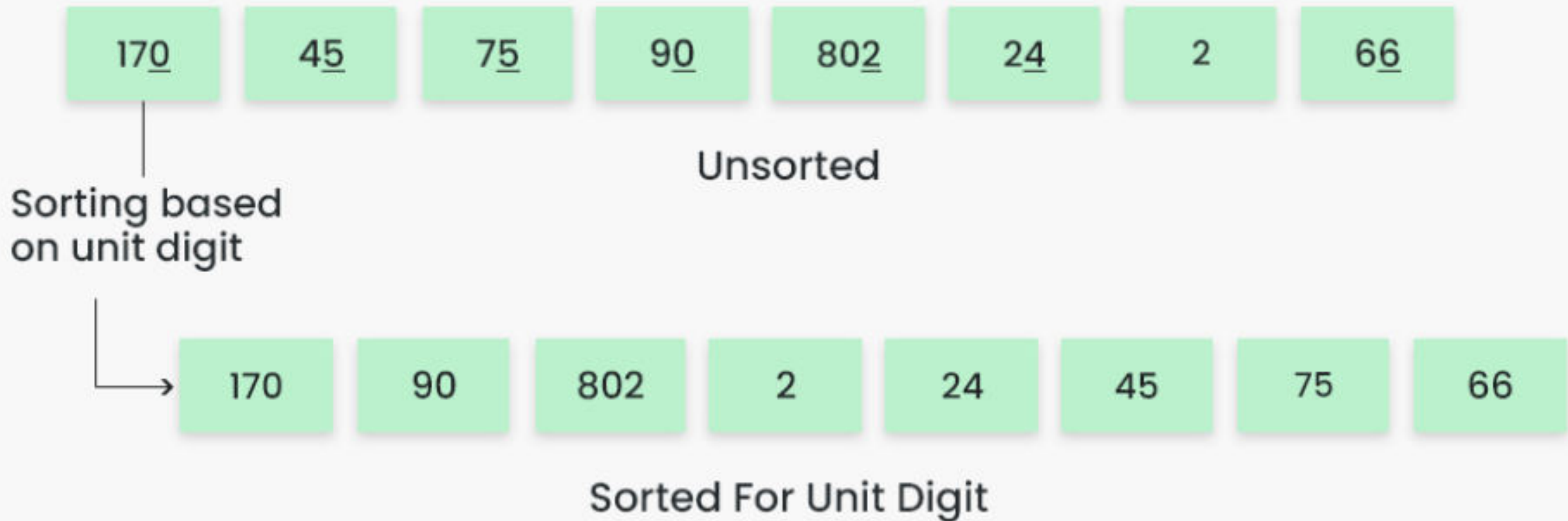
- It repeatedly distributes the elements into buckets based on each digit's value. This is different from other algorithms like Merge Sort or Quick Sort where we compare elements directly.
- By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, it achieves the final sorted order.
- We use a stable algorithm like Counting Sort to sort the individual digits so that the overall algorithm remains stable.

To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:

[Visit sharikatr.in](http://sharikatr.in) for more notes and ppts

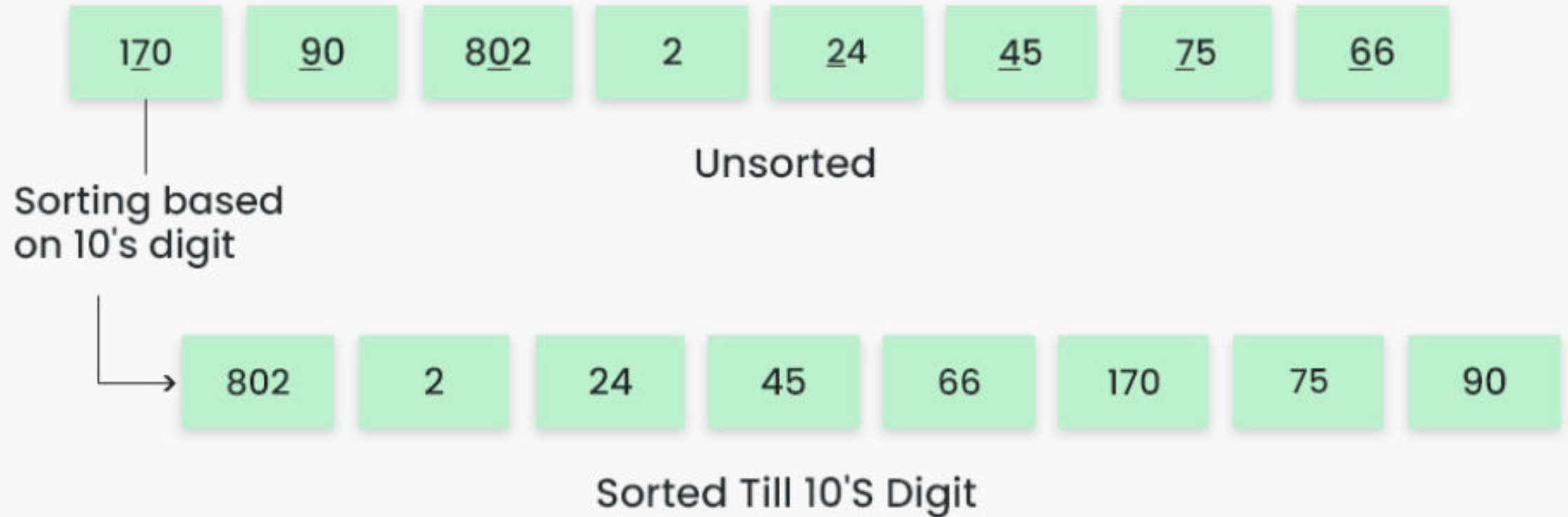
**Step 1:** Find the largest element, which is 802. It has three digits, so we will iterate three times.

**Step 2:** Sort the elements based on the **unit** place digits ( $X=0$ ).

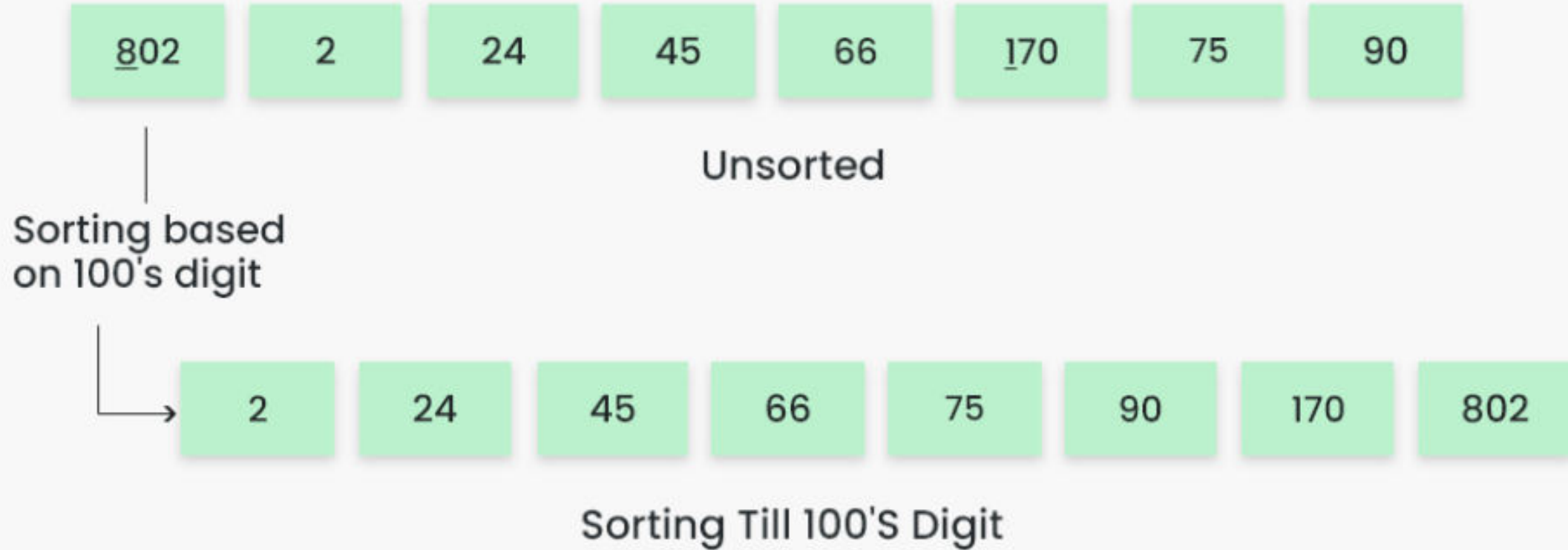


*How does Radix Sort Algorithm work | Step 2*

**Step 3:** Sort the elements based on the **tens** place digits.



**Step 4:** Sort the elements based on the hundreds place digits.



How does Radix Sort Algorithm work | Step 4

**Step 5:** The array is now sorted in ascending order.

Array after performing **Radix Sort** for all digits

2

24

45

66

75

90

170

802

---

## Time Complexity:

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value.

It has a time complexity of  $O(d * (n + b))$ , where  $d$  is the number of digits,  $n$  is the number of elements, and  $b$  is the base of the number system being used.

In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

# Hashing

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

- Hashing in the data structure is used to quickly identify a specific value within a given array.
- It creates a unique hash code for each element in the array and then stores the hash code instead of the actual element.
- This allows for quick lookup when searching for a specific value, as well as easy identification of any duplicates. Hashing in the data structure is a technique that is used to quickly identify a specific value within a given array.
- It works by creating a unique hash code for each element in the array and then stores the hash code in lieu of the actual element.
- This allows for quick look-up when searching for a specific value, as well as easy identification of any duplicates.



# Components of Hashing

Visit [sharkatt.in](http://sharkatt.in) for more notes and apps

There are majorly three components of hashing:

- **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
- **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
- **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

## How does Hashing work?

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

Eg 1:

Suppose we have a set of strings {"ab", "cd", "efg"} and we would like to store it in a table.

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
  - "a" = 1,
  - "b" = 2, .. etc, to all alphabetical characters.

•**Step 3:** Therefore, the numerical value by summation of all characters of the string:

•  $"ab" = 1 + 2 = 3,$

•  $"cd" = 3 + 4 = 7,$

•  $"efg" = 5 + 6 + 7 = 18$

•**Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.

•**Step 5:** So we will then store

•  $"ab" \text{ in } 3 \bmod 7 = 3,$

•  $"cd" \text{ in } 7 \bmod 7 = 0, \text{ and}$

•  $"efg" \text{ in } 18 \bmod 7 = 4.$

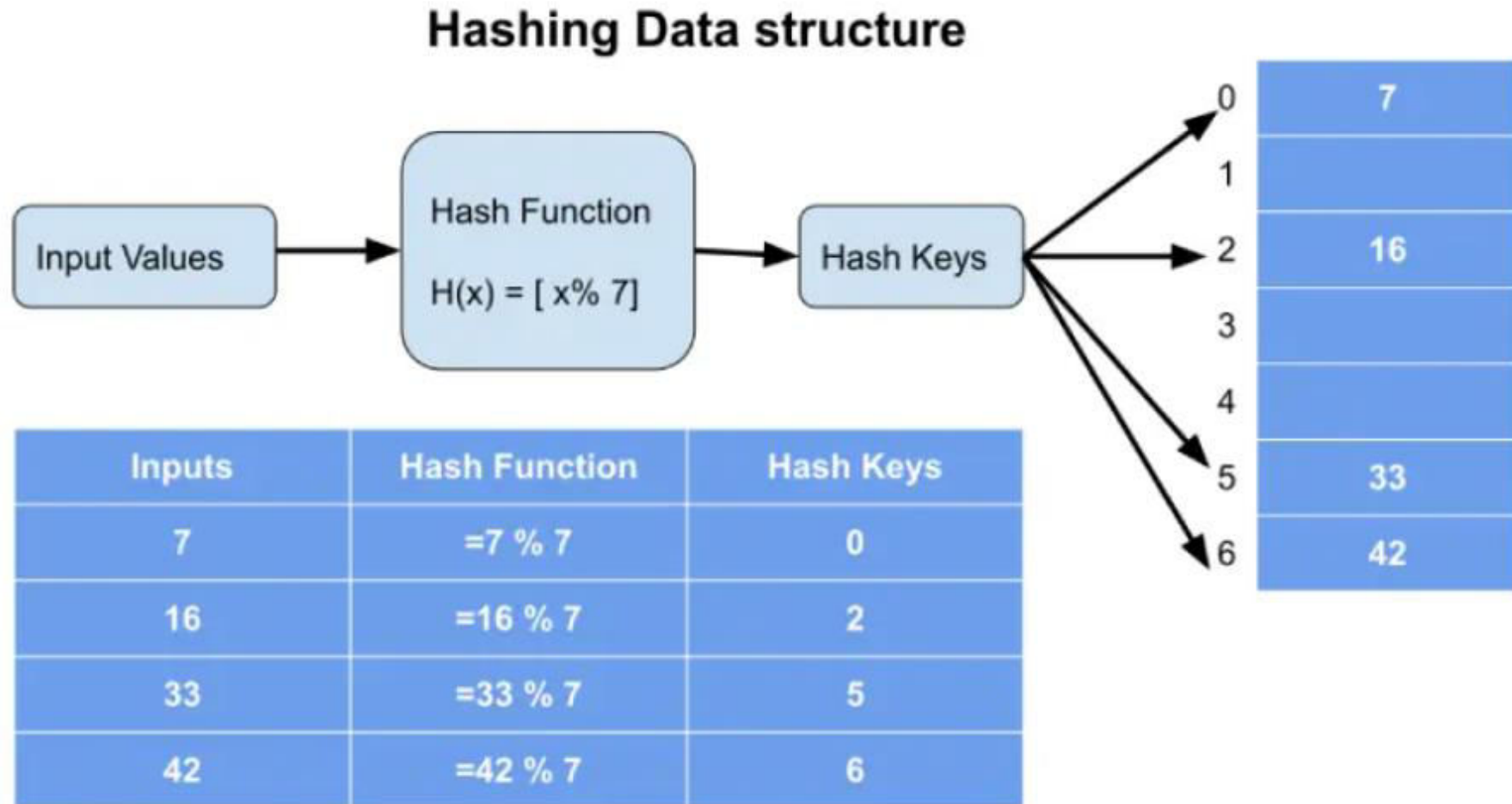
0	1	2	3	4	5	6
cd			ab	efg		

Prepared by Sharika T R, Assistant Professor Department of CSE, ASIET

*Mapping key with indices of array*

# Eg 2:

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts



# Hashing

---

- Consider a hash table of size 10 whose indices are 0,1,...8,9.
- Let the set of key values :10,19,35,43,62,59,31,49,77,33
- The hash function:
  - Add two digits in the key
  - Take the digit at unit place of the result as index. Ignore the digit at decimal place, if any.

# Hashing

Visit [sharikatrin.in](http://sharikatrin.in) for more notes and ppts

- key values :10,19,35,43,62,59,31,49,77,33

Hash table

key	Hash	Index
10	1+0	1
19	1+9	0
35	3+5	8
43	4+3	7
62	6+2	8
59	5+9	4
31	3+1	4
49	4+9	3
77	7+7	4
33	3+3	6

Index
0
19
1
10
2
3
49
4
59,31,77
5
6
33
7
43
8
35,62
9

# What is a Hash function?

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

## Types of Hash functions:

1. Division Method
2. Mid Square Method
3. Folding Method
4. Multiplication Method

# 1.Division Method

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

The division method is the simplest and easiest method used to generate a hash value. In this hash function, the value of k is divided by M and uses the remainder as obtained.

$$\text{Formula - } h(K) = k \bmod M$$

(where k = key value and M = the size of the hash table)

**Example –**

$$k = 1320$$

$$M = 11$$

$$\begin{aligned} h(1320) &= 1320 \bmod 11 \\ &= 0 \end{aligned}$$



Assume a table with 8 slots:

Hash key = key % table size

$$4 = 36 \% 8$$

$$2 = 18 \% 8$$

$$0 = 72 \% 8$$

$$3 = 43 \% 8$$

$$6 = 6 \% 8$$

[0]

72

[1]

[2]

18

[3]

43

[4]

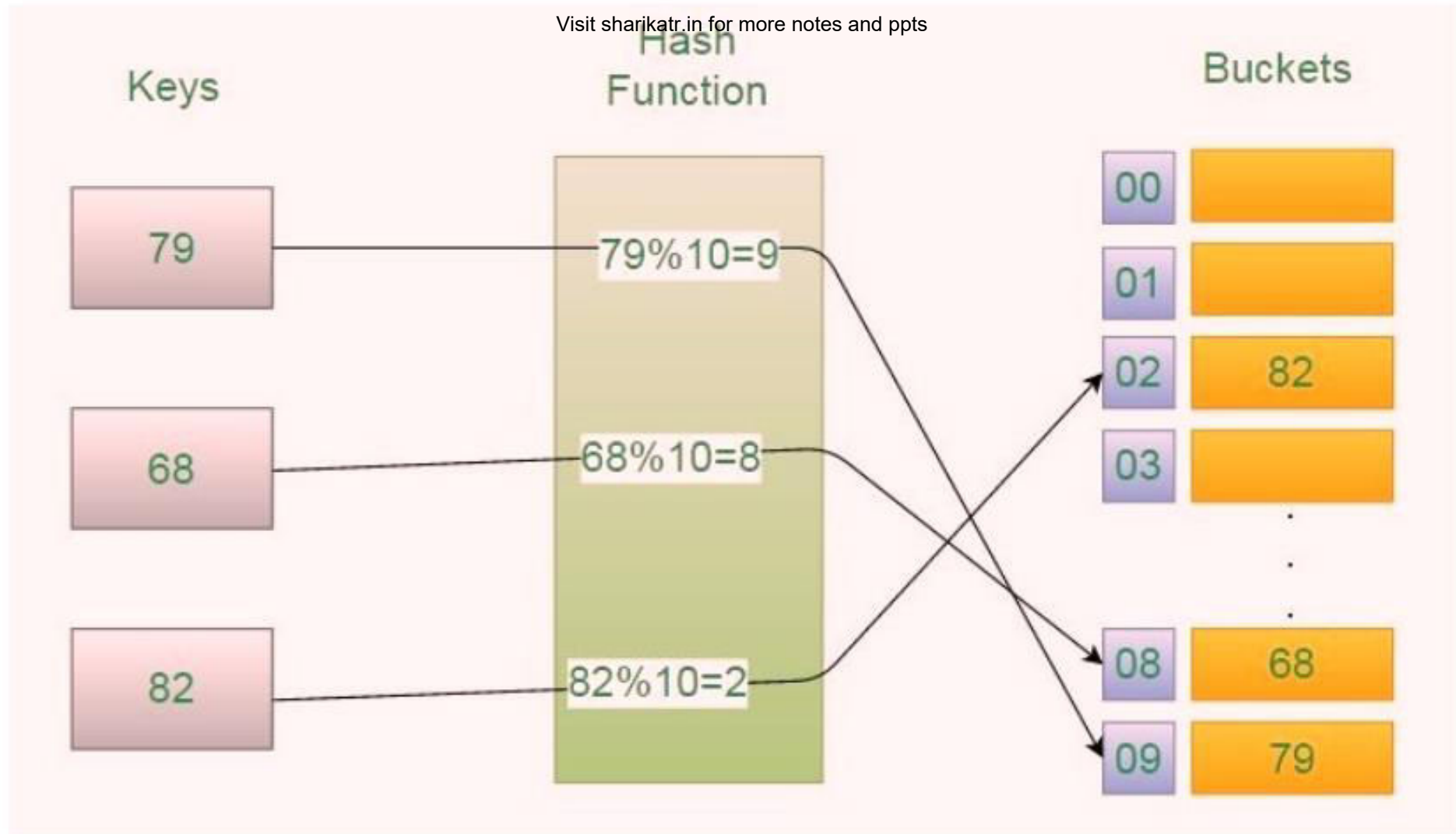
36

[5]

[6]

6

[7]



## 2. Mid Square Method

Visit [www.sharikatr.in](http://www.sharikatr.in) for more notes and ppts

The steps involved in computing this hash method include the following -

Squaring the value of  $k$  ( like  $k*k$  )

Extract the hash value from the middle  $r$  digits.

***Formula -  $h(K) = h(k \times k)$***

(where  $k$  = key value )

Suppose the size of the Hash Table  $(m) = 10$  (0 - 9) *maximum digits required for the index is 1*

$$\text{Element } (x) = 12 \Rightarrow x^2 = 144$$

Mid 1 digit of 144 is 4, so the element  $x=12$  will be stored at the  $\text{index}=4$  in the hash table with the size of 10 slots.

### **Another Example:**

Suppose the size of the Hash Table  $(m) = 1000$  (0 - 999) *maximum digits required for the index is 3*

$$\text{Element } (x) = 87431 \Rightarrow x^2 = 7644179761$$

The possible 3 digit mids of 7644179761 are 417 or 179, we can pick any of those mids. If we pick 419 then the element  $x=87431$  will be stored at the  $\text{index}=419$  in the hash table with the size of 1000 slots.

## Example:

Calculate the hash values of key 31, 15, 22

assume hash table has 10 memory locations whose indices vary from 0 to 9

$$h(31) = 31^2 = 961$$

$$h(15) = 15^2 = 225$$

$$h(22) = 22^2 = 484$$

Index	Value
5	15
6	31
7	
8	22

### 3. Folding Method

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

- There are two steps in this method -
- The key-value  $k$  should be divided into a specific number of parts, such as  $k_1, k_2, k_3, \dots, k_n$ , each having the very same number of digits aside from the final component, which may have fewer digits than the remaining parts.
- Add each component separately. The last carry, if any, is disregarded to determine the hash value.

***Formula -  $k = k_1, k_2, k_3, k_4, \dots, k_n$***

***$s = k_1 + k_2 + k_3 + k_4 + \dots + k_n$***

***$h(K) = s$***

(Where,  $s$  = addition of the parts of key  $k$ )

Example -

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

$$k = 54321$$

$$k1 = 54 ; k2 = 32 ; k3 = 1$$

Therefore,

$$s = k1 + k2 + k3$$

$$= 54 + 32 + 1$$

$$= 87$$

Thus,

$$h(k) = 87$$

- Here we are dealing with a hash table with index from 00 to 99, i.e., two-digit hash table
- So we divide the K numbers of two digits

K	2103	7148	12345
$k_1 k_2 k_3$	21, 03	71, 46	12, 34, 5
$H(k)$ $= k_1 + k_2 + k_3$	$H(2103)$ $= 21+03 = 24$	$H(7148)$ $= 71+4\textcolor{red}{6} = 19$	$H(12345)$ $= 12+\textcolor{red}{34}+\textcolor{red}{5} = 51$



Ex. (SSN) 123-45-6789

1. Divide into 3 parts:  
123, 456 and 789.

2. Add them.  
 $123 + 456 + 789 = 1368$

3.  $h(k) = k \bmod M$   
where  $M = 1000$   
 $h(1368) = 1368 \bmod 1000$   
 $= 368$

1. Divide into five parts: 12, 34, 56, 78 and 9.

2. Add them.  
 $12 + 34 + 56 + 78 + 9 = 189$

3.  $h(k) = k \bmod M$   
where  $M = 1000$   
 $h(189) = 189 \bmod 1000$   
 $= 189$

$P_1$	$P_2$	$P_3$	$P_4$	$P_5$
123	203	241	112	20

$$\begin{array}{r}
 P_1 \quad 123 \\
 P_2 \quad 203 \\
 P_3 \quad 241 \\
 P_4 \quad 112 \\
 P_5 \quad 20 \\
 \hline
 699
 \end{array}$$

Shift Folding

$$\begin{array}{r}
 P_1 \quad 123 \\
 P_2^r \quad 302 \\
 P_3 \quad 241 \\
 P_4^r \quad 211 \\
 P_5 \quad 20 \\
 \hline
 897
 \end{array}$$

Folding at boundaries

## 4. Multiplication Method

Visit [www.sharika.com](http://www.sharika.com) for more notes and ppts

Steps to follow -

- Pick up a constant value  $A$  (where  $0 < A < 1$ )
- Multiply  $A$  with the key value
- Take the fractional part of  $kA$
- Take the result of the previous step and multiply it by the size of the hash table,  $M$ .

***Formula -  $h(K) = \text{floor}(M (kA \bmod 1))$***

(Where,  $M$  = size of the hash table,  $k$  = key value and  $A$  = constant value)

## Example -

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

$$k = 1234$$

$$A = 0.35784$$

$$M = 100$$

So,

$$h(1234) = \text{floor} [ 100(1234 \times 0.35784 \bmod 1)]$$

$$= \text{floor} [ 100 ( 441.57456 \bmod 1)]$$

$$= \text{floor} [100 ( 0. 57456)]$$

$$= \text{floor} [ 57.456]$$

$$= 57$$

Thus,

$$h(1234) = 57$$


# What is a Hash Collision?

visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

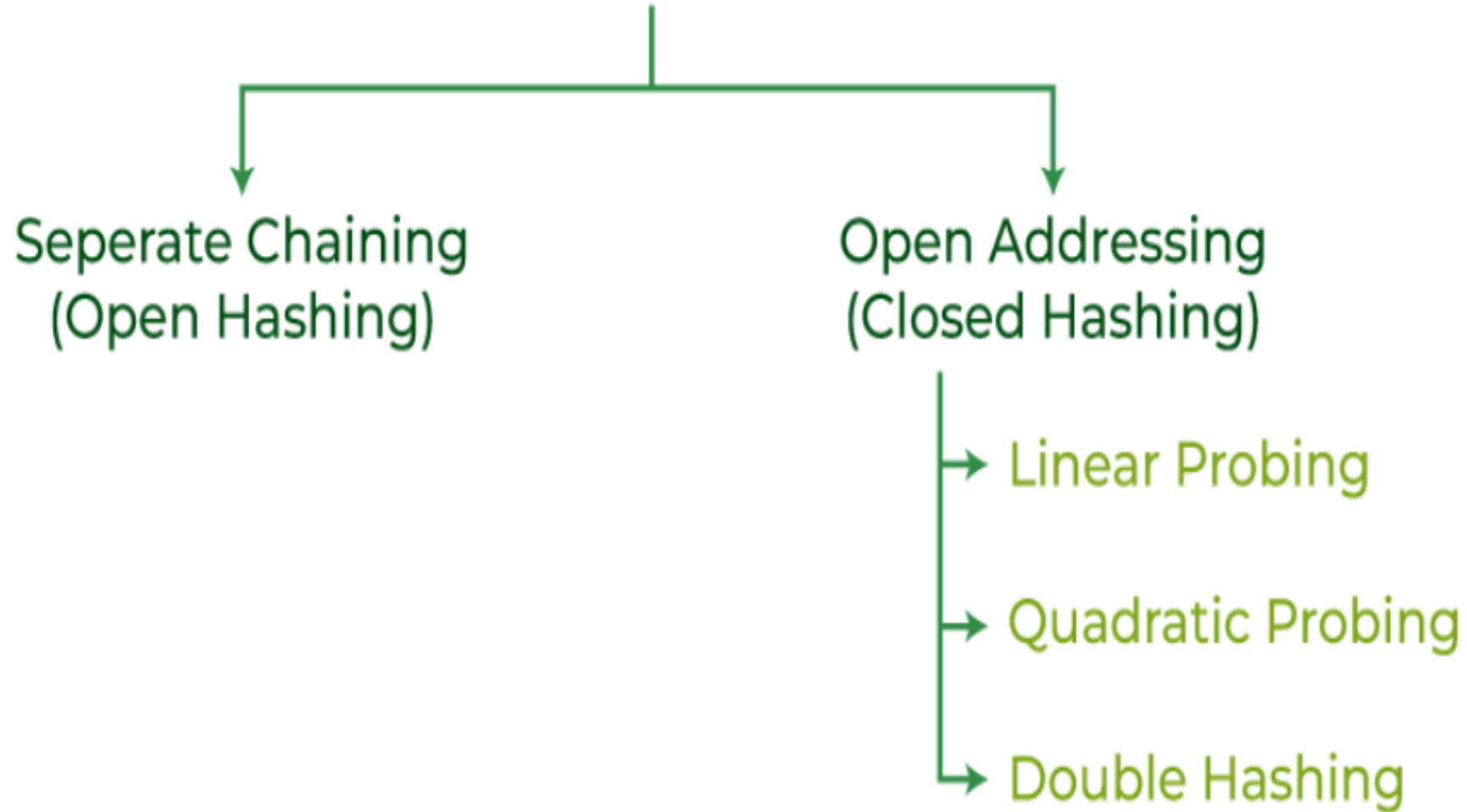
- A hash collision happens when the same hash value is produced for two different input values by a hash algorithm.
- Collisions occur because different hashing techniques in data structure convert every input into a fixed-length code, regardless of its length.
- Since there are an endless number of inputs and a limited number of outputs, the hashing algorithms will eventually produce repeating hashes.

- Let's take the exact same hash function from before: take the value to be hashed mod 10, and place it in that slot in the hash table.
- Numbers to hash: 22, 9, 14, 17, 42
- As before, the hash table is shown to the right.
- As before, we hash each value as it appears in the string of values to hash, starting with the first value. The first four values can be entered into the hash table without any issues. It is the last value, 42, however, that causes a problem.  $42 \bmod 10 = 2$ , but there is *already* a value in slot 2 of the hash table, namely 22. This is a collision.

0	
1	
2	22
3	
4	14
5	
6	
7	17
8	
9	9



# Collision Resolution Technique



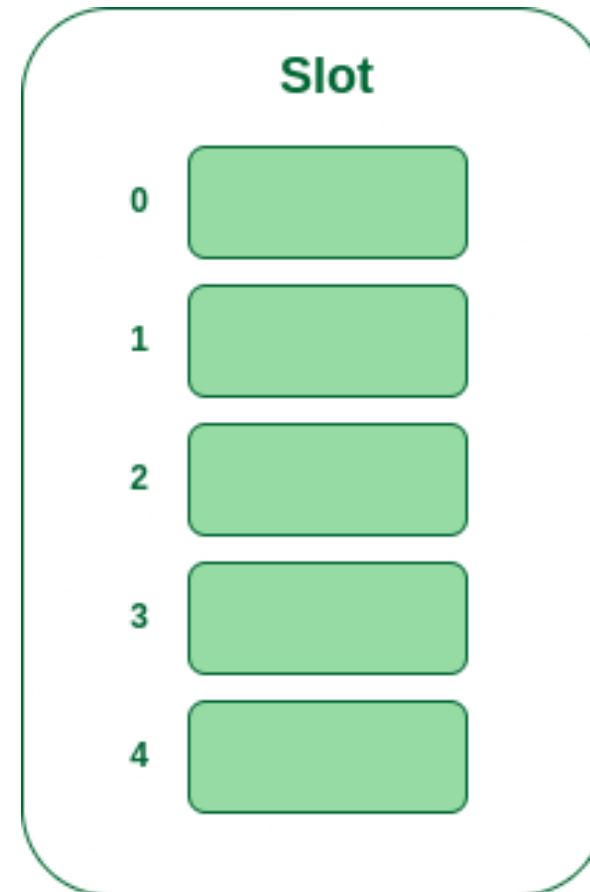
## Separate Chaining

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

**Hash function =  $\text{key} \% 5$ ,**  
**Elements = 12, 15, 22, 25 and 37.**

- **Step 1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.

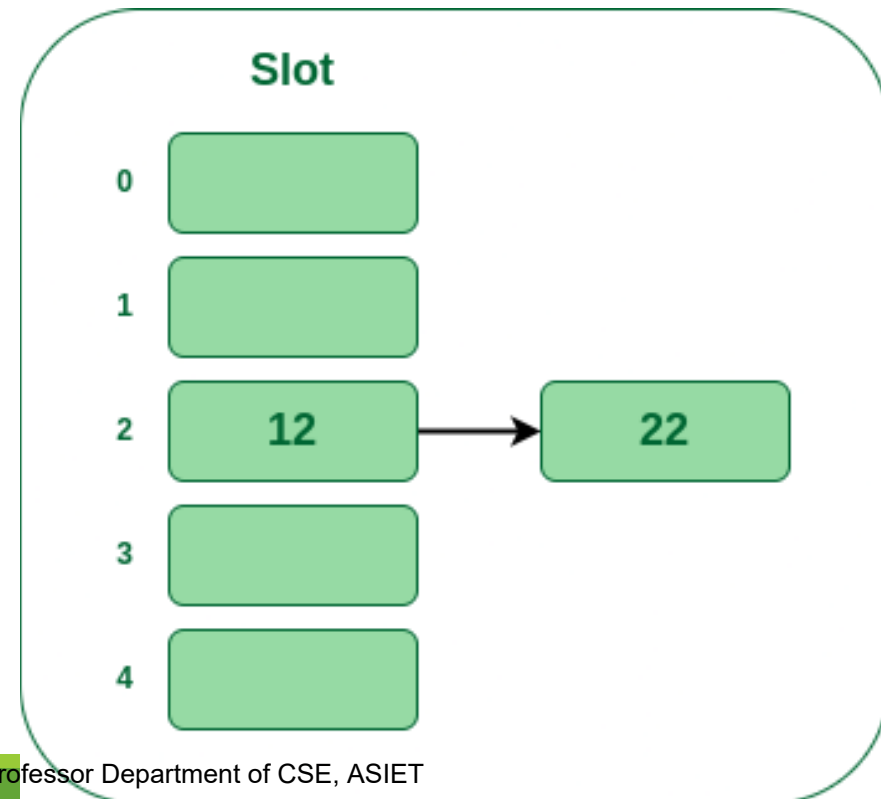
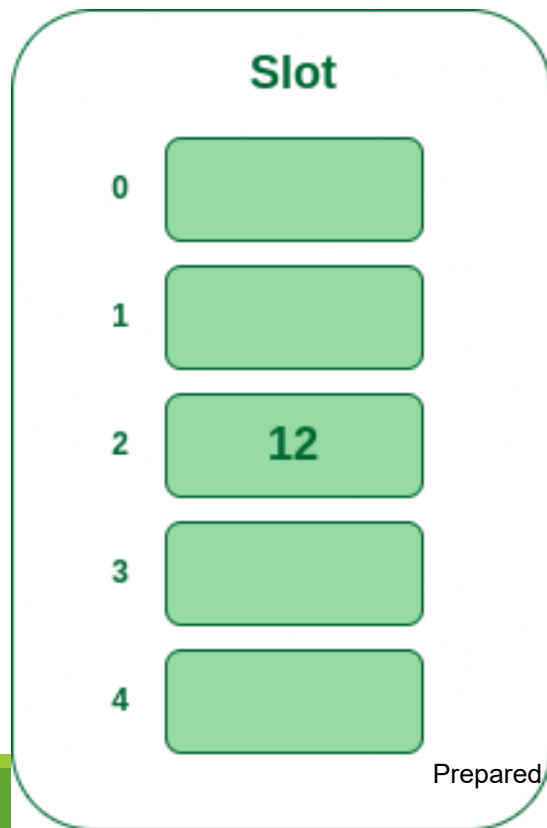




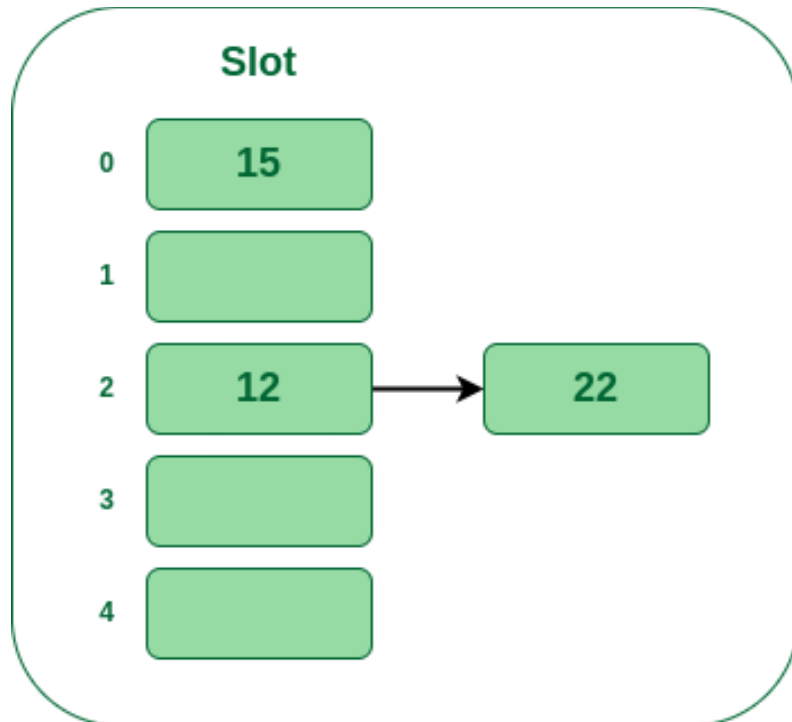
•**Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function  $12\%5=2$ .

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

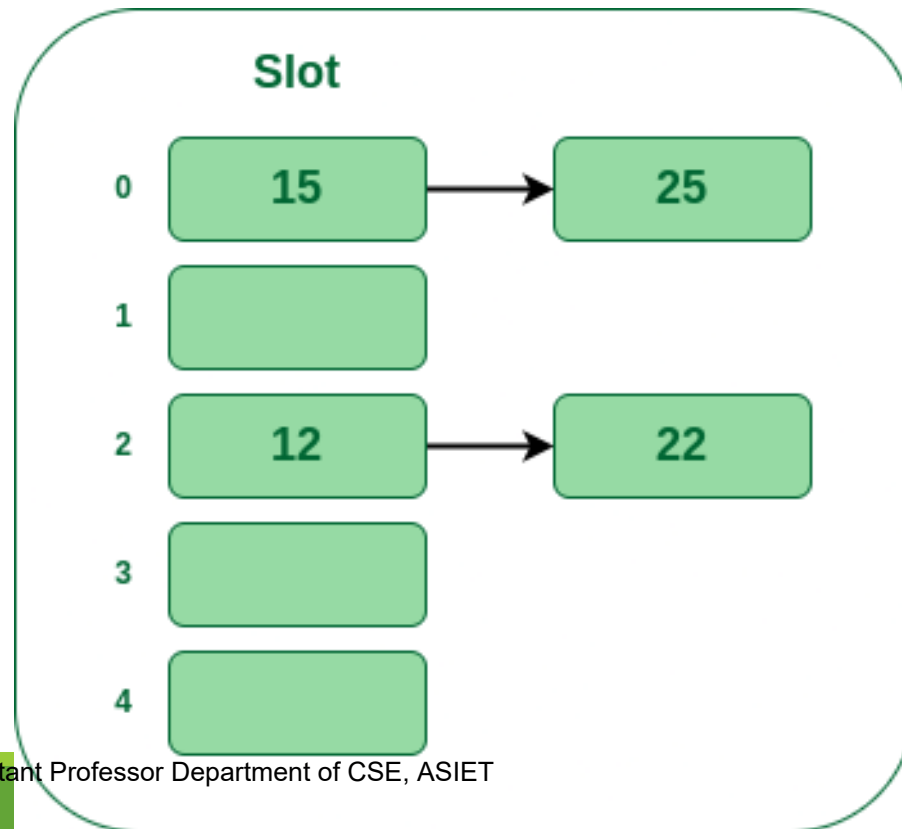
•**Step 3:** Now the next key is 22. It will map to bucket number 2 because  $22\%5=2$ . But bucket 2 is already occupied by key 12.



- **Step 4:** The next key is 15. It will map to slot number 0 because  $15\%5=0$ .



- **Step 5:** Now the next key is 25. Its bucket number will be  $25\%5=0$ . But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.



In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

### **2.a) Linear Probing**

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

### **2.b) Quadratic Probing**

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

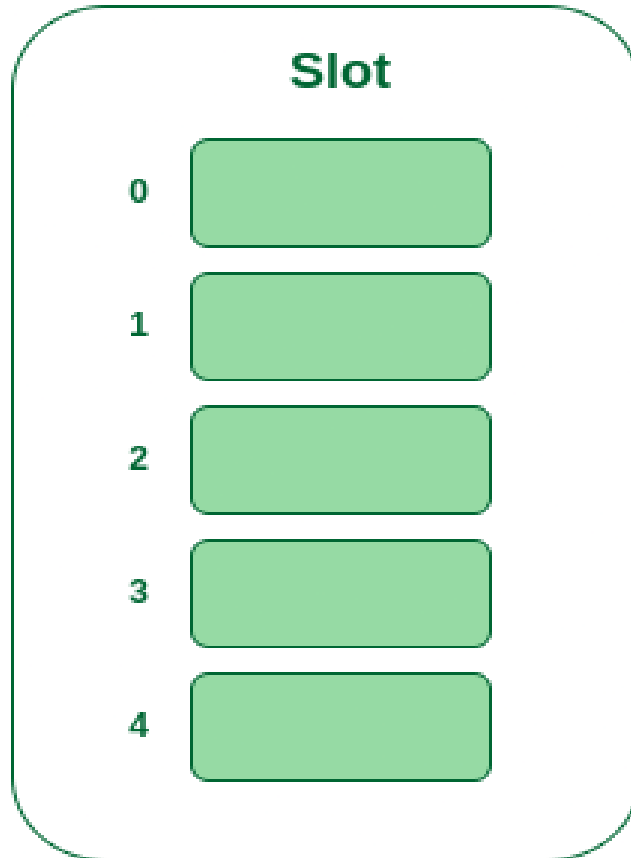
### **2.c) Double Hashing**

The double hashing technique uses two hash functions. The second hash function comes into use when the first function causes a collision. It provides an offset index to store the value.

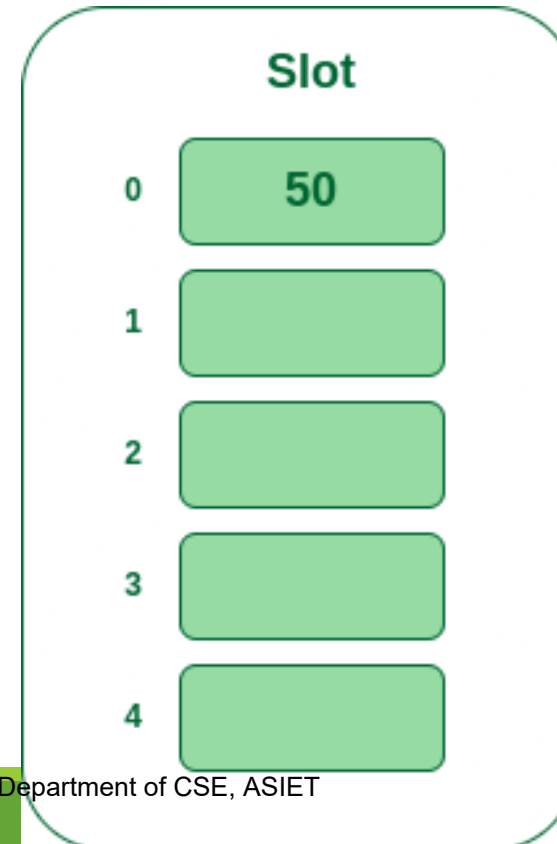
## 2.a) Linear Probing

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

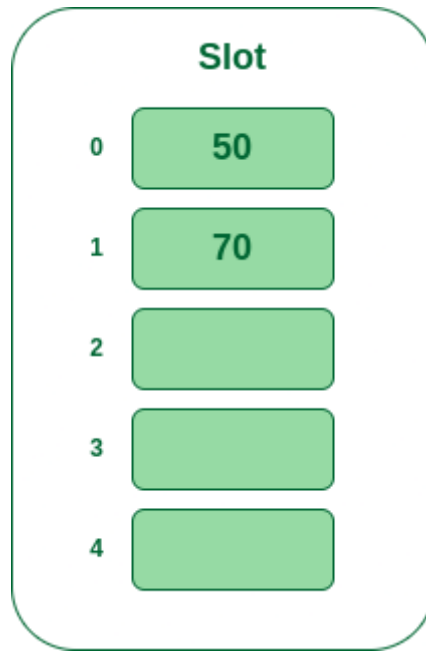
**Example:** Let us consider a simple hash function as “*key mod 5*” and a sequence of keys that are to be inserted are **50, 70, 76, 85, 93**.



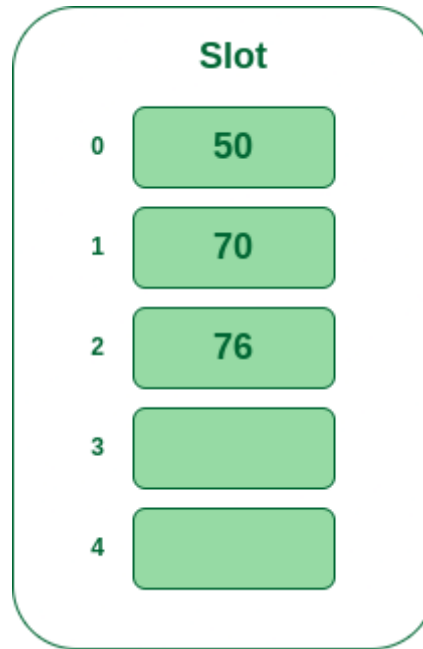
- insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because  $50\%5=0$ . So insert it into slot number 0.



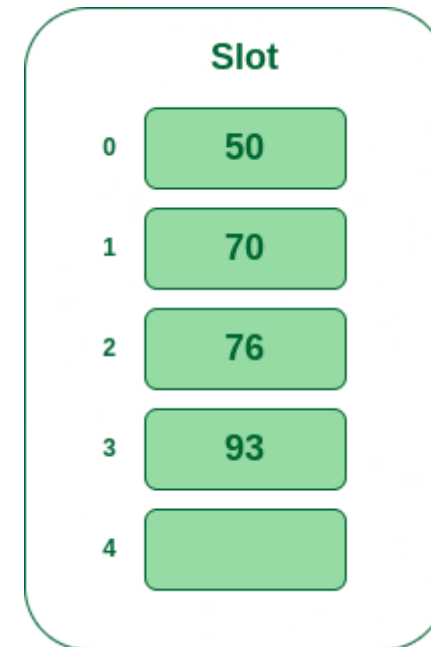
- The next key is 70. It will map to slot number 0 because  $70\%5=0$  but 50 is already at slot number 0 so, search for the next empty slot and insert it.



- The next key is 76. It will map to slot number 1 because  $76\%5=1$  but 70 is already at slot number 1 so, search for the next empty slot and insert it.



- The next key is 93. It will map to slot number 3 because  $93\%5=3$ , So insert it into slot number 3.



## Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \dots, H + k^2$$

Insert  
18, 89, 21

0	
1	21
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert  
58

	21
	58
	18
	89

For **58**:

- $H = \text{hash}(58, 10) = 8$
- Probe sequence:
  - $i = 0, (8+0) \% 10 = 8$
  - $i = 1, (8+1) \% 10 = 9$
  - $i = 2, (8+4) \% 10 = 2$

Insert  
68

	21
	58
	68
	18
	89

For **68**:

- $H = \text{hash}(68, 10) = 8$
- Probe sequence:
  - $i = 0, (8+0) \% 10 = 8$
  - $i = 1, (8+1) \% 10 = 9$
  - $i = 2, (8+4) \% 10 = 2$
  - $i = 3, (8+9) \% 10 = 7$

## 2.c) Double Hashing

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

The double hashing technique uses two hash functions. The second hash function comes into use when the first function causes a collision. It provides an offset index to store the value.

The formula for the double hashing technique is as follows:

$$(\text{firstHash}(\text{key}) + i * \text{secondHash}(\text{key})) \% \text{sizeOfTable}$$

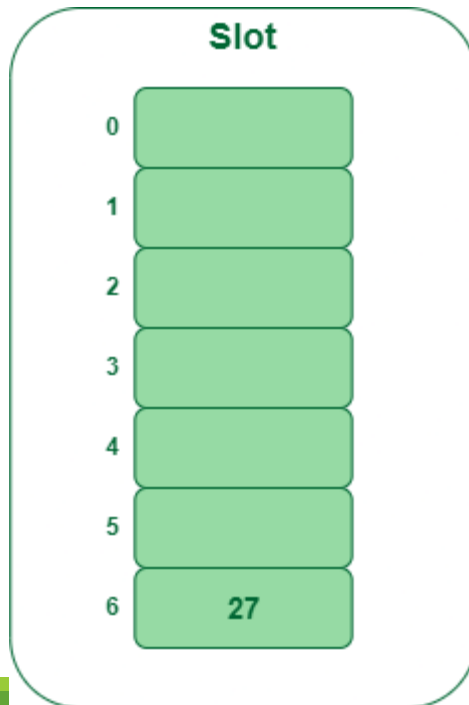
Where  $i$  is the offset value. This offset value keeps incremented until it finds an empty slot.



**Example:** Insert the keys 27, 43, 692, 72 into the Hash Table of size 7.  
where first hash-function is  $h1(k) = k \bmod 7$  and second hash-function is  $h2(k) = 1 + (k \bmod 5)$

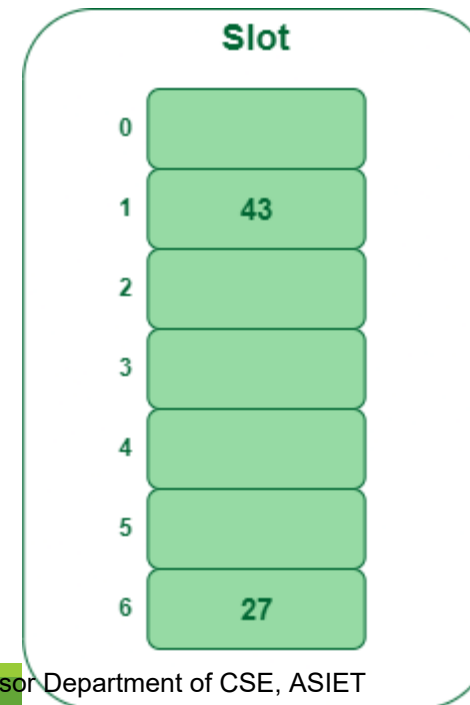
•**Step 1:** Insert 27

- $27 \% 7 = 6$ , location 6 is empty so insert 27 into 6 slot.



•**Step 2:** Insert 43

- $43 \% 7 = 1$ , location 1 is empty so insert 43 into 1 slot.



- **Step 3:** Insert 692

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

- $692 \% 7 = 6$ , but location 6 is already being occupied and this is a collision

- So we need to resolve this collision using double hashing.

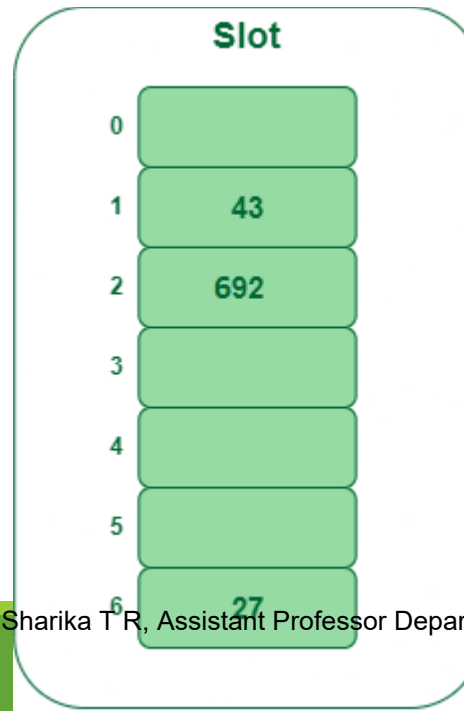
$$h_{\text{new}} = [h_1(692) + i * (h_2(692))] \% 7$$

$$= [6 + 1 * (1 + 692 \% 5)] \% 7$$

$$= 9 \% 7$$

$$= 2$$

Now, as 2 is an empty slot, so we can insert 692 into 2nd slot.



#### •Step 4: Insert 72

Visit [sharikatr.in](http://sharikatr.in) for more notes and ppts

- $72 \% 7 = 2$ , but location 2 is already being occupied and this is a collision.

- So we need to resolve this collision using double hashing.

$$h_{\text{new}} = [h_1(72) + i * (h_2(72))] \% 7$$

$$= [2 + 1 * (1 + 72 \% 5)] \% 7$$

$$= 5 \% 7$$

$$= 5,$$

***(suppose 5 is not an empty slot we do double hashing again with  $i=2$ )***

Now, as 5 is an empty slot, so we can insert 72 into 5th slot.

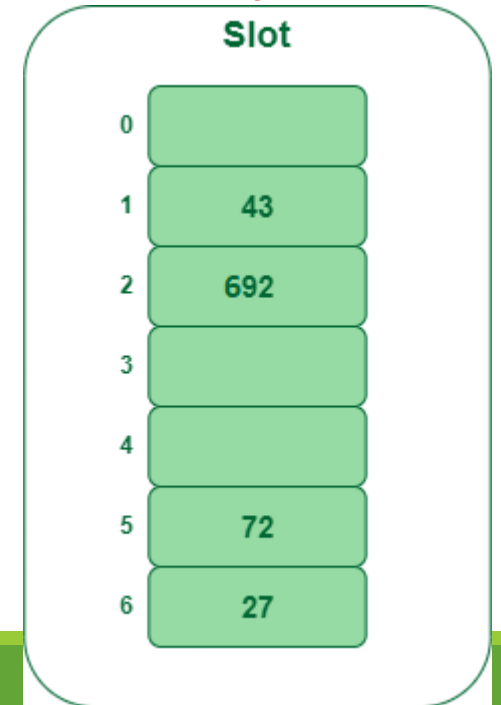


Table Size = 10 elements

Hash<sub>1</sub>(key) = key % 10

Hash<sub>2</sub>(key) = 7 - (k % 7)

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !  
= 7 - (49 % 7)  
= 7 positions from [9]

Hash(58) = 58 % 10 = 8  
= 7 - (58 % 7)  
= 5 positions from [8]

Hash(69) = 69 % 10 = 9  
= 7 - (69 % 7)  
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

# LINEAR SEARCH

*Search 20*

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Step 1:

*number to search == arr[0]      20 == 12?      No, check next element.*

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Step 2:

*number to search == arr[1]      20 == 5?      No, check next element.*

12	5	10	15	31	20	25	2	40
----	---	----	----	----	----	----	---	----

Step 3:

*number to search == arr[2]      20 == 10?      No, check next element.*

12	5	10	15	31	20	25	2	40
----	---	----	----	----	----	----	---	----

Step 4:

*number to search == arr[3]      20 == 15?      No, check next element.*

12	5	10	15	31	20	25	2	40
----	---	----	----	----	----	----	---	----

Step 5:

*number to search == arr[4]      20 == 31?      No, check next element.*

12	5	10	15	31	20	25	2	40
----	---	----	----	----	----	----	---	----

Step 6:

*number to search == arr[5]      20 == 20?      YES, return true*

12	5	10	15	31	20	25	2	40
----	---	----	----	----	----	----	---	----

# Algorithm

Linear Search ( Array A, Value x)

Step 1: Set i to 0

Step 2: if  $i > n-1$  then go to step 7

Step 3: if  $A[i] = x$  then go to step 6

Step 4: Set i to  $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

# Complexity

---

Best case complexity- $O(1)$

Average case complexity- $O(n)$

Worst case complexity- $O(n)$

# BINARY SEARCH

## Binary Search for 50 in 7 elements Array

Given Array

1	5	20	35	50	65	70
---	---	----	----	----	----	----

0 1 2 3 4 5 6  
start end

$$\text{mid} = \frac{0+6}{2} = 3$$

1	5	20	35	50	65	70
---	---	----	----	----	----	----

0 1 2 3 4 5 6  
start end

35 < 50  
Take 2<sup>nd</sup> Half

$$\text{mid} = \frac{4+6}{2} = 5$$

1	5	20	35	50	65	70
---	---	----	----	----	----	----

0 1 2 3 4 5 6  
start end

65 > 50  
Take 1<sup>st</sup> Half

$$\text{mid} = \frac{4+4}{2} = 4$$

1	5	20	35	50	65	70
---	---	----	----	----	----	----

0 1 2 3 4 5 6  
start end

50 Found  
Return 50



## Binary Search Algorithm

1. Def. binary Search (A, x):
2.  $n = \text{len}(A)$
3.  $\text{beg} = 0$
4.  $\text{end} = n - 1$
5.  $\text{result} = -1$
6. While ( $\text{beg} \leq \text{end}$ ):
7.      $\text{mid} = (\text{beg} + \text{end}) / 2$
8.     If ( $A[\text{mid}] \leq x$ ):
9.          $\text{beg} = \text{mid} + 1$
10.         $\text{result} = \text{mid}$
11.     Else:
12.         $\text{end} = \text{mid} - 1$
13. Return result

# Complexity

---

Best case- $O(1)$

Average and worst case-  $O(\log n)$

Worst case-  $O(n)$

# MODULE 5 ENDS

---