Evaluation Scheme

NOIDA INSTITUTE OF ENGINEERING & TECHNOLOGY, GREATER NOIDA
(An Autonomous Institute)

B. TECH (CSE)
EVALUATION SCHEME
SEMESTER-III

Sl. No.	Subject Codes	Subject Name	Periods			Evaluation Scheme				End Semester		Total	Credit
			L	T	P	CT	TA	TOTAL	PS	TE	PE		
WEEKS COMPULSORY INDUCTION PROGRAM													
1	AAS0301A	Engineering Mathematics-III	3	1	0	30	20	50		100		150	4
2	ACSE0306	Discrete Structures	3	0	0	30	20	50		100		150	3
3	ACSE0304	Digital Logic & Circuit Design	3	0	0	30	20	50		100		150	3
4	ACSE0301	Data Structures	3	1	0	30	20	50		100		150	4
5	ACSE0302	Object Oriented Techniques using Java	3	0	0	30	20	50		100		150	3
6	ACSE0305	Computer Organization & Architecture	3	0	0	30	20	50		100		150	3
7	ACSE0354	Digital Logic & Circuit Design Lab	0	0	2				25		25	50	1
8	ACSE0351	Data Structures Lab	0	0	2				25		25	50	1
9	ACSE0352	Object Oriented Techniques using Java Lab	0	0	2				25		25	50	1
10	ACSE0359	Internship Assessment-I	0	0	2				50			50	1
11	ANC0301/ ANC0302	Cyber Security*/ Environmental Science*(Non Credit)	2	0	0	30	20	50		50		100	0
12		MOOCs** (For B.Tech. Hons. Degree)											
		GRAND TOTAL										1100	24

****List of MOOCs (Coursera) Based Recommended Courses for Second Year (Semester-III) B. Tech Students**

S. No.	Subject Code	Course Name	University / Industry Partner Name	No of Hours	Credits
1	AMC0023	Java Programming: Arrays, Lists, and Structured Data	Duke University	14	1
2	AMC0032	Object Oriented Programming in Java	Duke University	40	3

PLEASE NOTE:-

- **Internship (3-4 weeks) shall be conducted during summer break after semester-II and will be assessed during semester-III**
- ***Non Credit Course**
  *All Non Credit Courses (a qualifying exam) are awarded zero (0) credit.
  *Total and obtained marks are not added in the Grand Total.

Abbreviation Used: -
L: Lecture, T: Tutorial, P: Practical, CT: Class Test, TA: Teacher Assessment, PS: Practical Sessional, TE: Theory End Semester Exam., PE: Practical End Semester Exam.

Unit I Syllabus

Introduction to data structure, Arrays, Searching, Sorting and Hashing

Data types: Primitive and non-primitive, Types of Data Structures- Linear & Non-Linear Data Structures. Time and Space Complexity of an algorithm, Asymptotic notations (Big Oh, Big Theta and Big Omega), Abstract Data Types (ADT)

Arrays: Definition, Single and Multidimensional Arrays, Representation of Arrays: Row Major Order, and Column Major Order, Derivation of Index Formulae for 1-D,2-D,3-D and n-D Array Application of Arrays, Sparse Matrices and their Representations.

Searching: Linear search, Binary search. **Sorting:** Bubble sort, Insertion sort, Selection sort, Radix Sort, Merge sort, Quick sort.

Branch wise Application

- A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- For example, we can store a list of items having the same data-type using the *array* data structure.
- 2D Arrays, commonly known as, matrix, are used in image processing.
- It is also used in speech processing, in which each speech signal is an array.

Unit Content

- Introduction
- Basic Terminology
- Data Structure, operations and classification
- Algorithm
  - Efficiency
  - Complexity
  - Asymptotic Notations
  - Time-space tradeoff
  - Abstract data types

Unit Content (contd..)

- Arrays
  - Declaration
  - Initialization
  - Multidimensional arrays
  - Representation of arrays
  - Application of arrays
  - Sparse matrix
- Linked lists
  - Representation
  - Types of link list
  - Basic operations on link list
  - Insertion and deletion
  - Polynomial representation

Unit Objective

- Understanding basic data structure.
- To learn about the basic properties of different data structures.
- Understand algorithms and their efficiency
- To learn about arrays and linked lists.
- To understand different types of arrays and Linked lists.
- Basic operations of arrays and linked list..

Course Objective

- Introduction to basic data structures.
- To know about the basic properties of different data structures.
- Classification and operations on data structure
- Understand algorithms and their efficiency
- Study logical and mathematical description of array and link list.
- Implementation of array and link list on computer.
- Differentiate the usage of array and link list in different scenarios.

Course Outcome

CO	CO Description	Bloom's Knowledge Level (KL)
CO 1	Describe the need of data structure and algorithms in problem solving and analyze Time space trade-off.	K2, K4
CO 2	Describe how arrays are represented in memory and how to use them for implementation of matrix operations, searching and sorting along with their computational efficiency.	K2, K6
CO 3	Design, implement and evaluate the real-world applications using stacks, queues and non-linear data structures.	K5, K6
CO 4	Compare and contrast the advantages and disadvantages of linked lists over arrays and implement operations on different types of linked list.	K4, K6
CO 5	Identify and develop the alternative implementations of data structures with respect to its performance to solve a real-world problem.	K1, K3, K5, K6

Program Outcomes (POs)

1. Engineering knowledge
2. Problem analysis
3. Design/development of solutions
4. Conduct investigations of complex problems
5. Modern tool usage
6. The engineer and society
7. Environment and sustainability
8. Ethics
9. Individual and team work
10. Communication
11. Project management and finance
12. Life-long learning

CO-PO Mapping

CO-PO correlation matrix of Data Structure (KCS 301)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
ACSE0301.1	3	3	3	2	-	1	-	1	2	2	2	2
ACSE0301.2	3	3	2	2	-	1	-	1	2	2	1	2
ACSE0301.3	3	3	2	2	-	1	-	1	2	2	2	2
ACSE0301.4	3	3	2	2	-	1	-	1	2	2	2	2
ACSE0301.5	3	3	3	3	2	2	2	2	3	3	3	3
Average	3	3	2.4	2.2	0.4	1.2	0.4	1.2	2.2	2.2	2	2.2

Program Specific Outcomes (PSOs)

On successful completion of graduation degree the Engineering graduates will be able to:

PSO1: The ability to design and develop the hardware sensor device and related interfacing software system for solving complex engineering problem.

PSO2: The ability to understanding of Inter disciplinary computing techniques and to apply them in the design of advanced computing .

PSO 3: The ability to conduct investigation of complex problem with the help of technical, managerial, leadership qualities, and modern engineering tools provided by industry sponsored laboratories.

PSO 4: The ability to identify, analyze real world problem and design their solution using artificial intelligence ,robotics, virtual. Augmented reality ,data analytics, block chain technology and cloud computing.

CO-PSO Mapping

Mapping of Program Specific Outcomes and Course Outcomes

	PSO1	PSO2	PSO3	PSO4
ACSE0301.1	3	3	2	2
ACSE0301.2	3	3	2	3
ACSE0301.3	3	3	2	2
ACSE0301.4	3	3	3	3
ACSE0301.5	3	3	3	3
Average	3	3	2.4	2.6

CO-PO and PSO Mapping

CO-PO correlation matrix of Data Structure (KCS 301)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
ACSE0301.1	3	3	3	2	-	1	-	1	2	2	2	2
ACSE0301.2	3	3	2	2	-	1	-	1	2	2	1	2
ACSE0301.3	3	3	2	2	-	1	-	1	2	2	2	2
ACSE0301.4	3	3	2	2	-	1	-	1	2	2	2	2
ACSE0301.5	3	3	3	3	2	2	2	2	3	3	3	3
Average	3	3	2.4	2.2	0.4	1.2	0.4	1.2	2.2	2.2	2	2.2

Mapping of Program Specific Outcomes and Course Outcomes

	PSO1	PSO2	PSO3	PSO4
ACSE0301.1	3	3	2	2
ACSE0301.2	3	3	2	3
ACSE0301.3	3	3	2	2
ACSE0301.4	3	3	3	3
ACSE0301.5	3	3	3	3
Average	3	3	2.4	2.6

Video Lecture

- https://www.youtube.com/watch?v=_t2GVaQasRY

- https://www.youtube.com/watch?v=pkYVOmU3MgA

Prerequisite and Recap

- Interest
- Get Familiar with python programming language.
- Start learn Data Structure and Algorithm daily.
- Practice ! Because practice makes you perfect.

Basic Terminology(CO1)

- **Data-** values or set of values
- Data item (Group items and Elementary items)
- **Entity** –something with **attributes** (properties)
- Entity set (Entities with similar attributes)
- **Information-** meaningful or processed data
- Field, records and files

Topic Objective

- To learn basic terminologies used in Data Structure.
- Understand elementary data organization.
- To study built-in data types.
- Understand efficiency of algorithms.
- Learn time- space complexity.
- Understand asymptotic notations.

Definition

Data Structure ..

A data structure is a systematic way of organizing data in a computer so that it can be used efficiently.

Ex:- Instead of storing values in multiple variables just create an array to store all those values.

- storing strings as sequence of characters in an array.

Why Data Structure

Why Data Structure

- Human requirement with computer are going to complex day by day. To solve the complex requirements in efficient way we need this study.

- Provide fastest solution of human requirements.

- Provide efficient solution of complex problem.

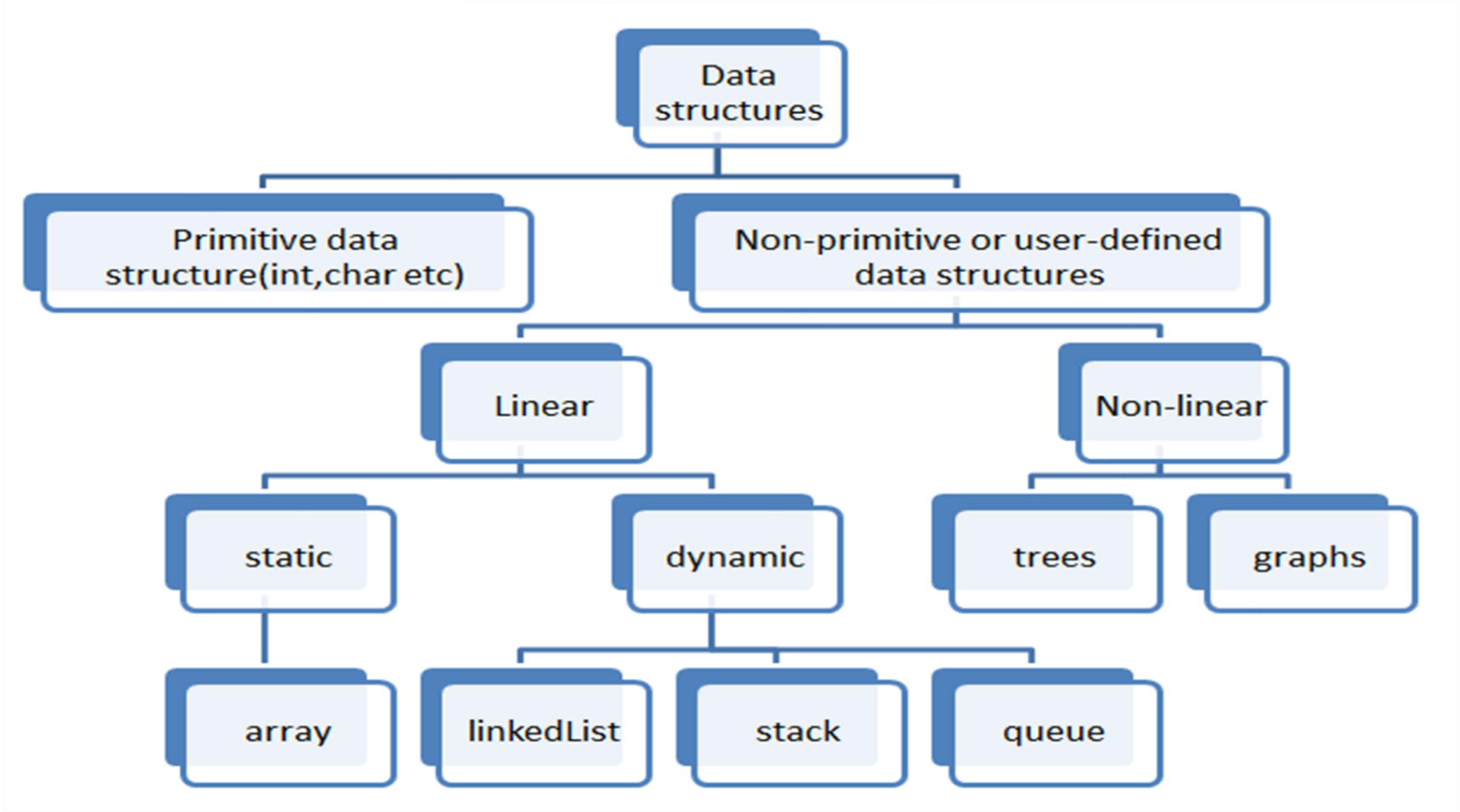
  >Space

  >Time

Real life Examples

- **REDO/UNDO -** STACKS
- Storing **BITMAP IMAGES -** ARRAYS
- Storing **FRIENDSHIP INFORMATION** on Social Media - GRAPHS

Classification of Data Structures



Introduction (CO1)

Classification of Data Structure ...

- **Simple Data Structure /Primitive data structure:**
  - used to represent the standard data types of any one of the computer languages.
  - Defines certain domain of values and operation allowed on those values (integer, Character, float etc.)
- **Compound Data Structure / Non Primitive Data Structure / User Defined Data Structure:**
  - can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as Linear and Non-Linear Data Structure.
  - Operations and values are not specified in the language but is specified by the user. (structure, union)

Classification of User Defined Data Structures ...

- **Linear Data Structures**: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists, Queue.
  - **Static data structure:** memory is allocated at compile time. Therefore, Fixed size. (fast access but slower insertion and deletion)
  - **Dynamic data structures:** memory is allocated at run time. Therefore, memory size is flexible. (fast insertion and deletion but slower access)
- **Non-Linear Data Structures**: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs.

Operation on Linear/Non-Linear Data Structure

- Add an element
- Delete an element
- Traverse / Display
- Sort the list of elements
- Search for a data element

Types of Linear Data Structure

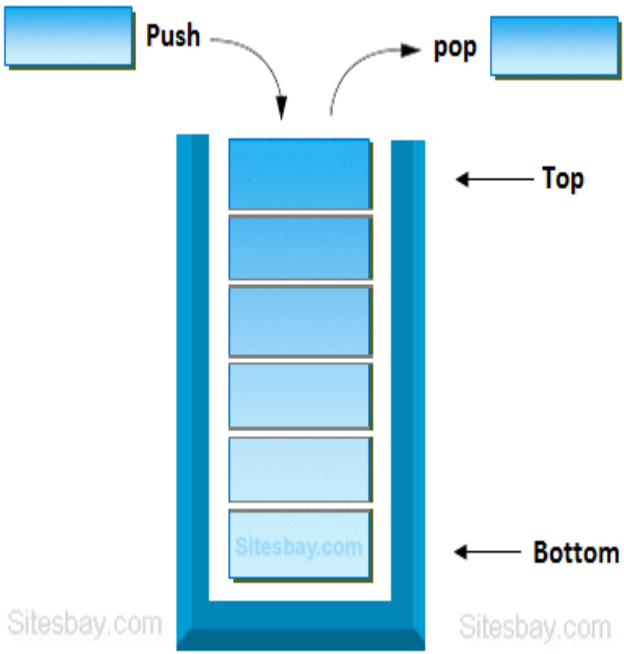
- Array : An array is the collection of the variables of the same data type that are referenced by the common name.
- int A[10], char B[10]

Array of Integers									
0	1	2	3	4	5	6	7	8	9
5	6	4	3	7	8	9	2	1	2

Types of Linear Data Structure....

- **Stack**

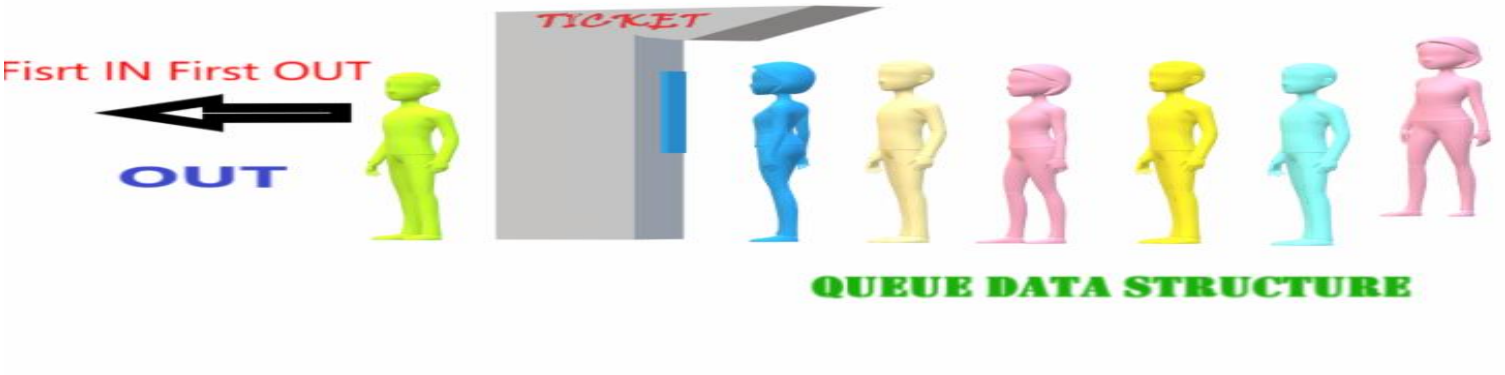
Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO**(Last In First Out) principle.



Types of Linear Data Structure…

Queue

- it is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'.
- In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.

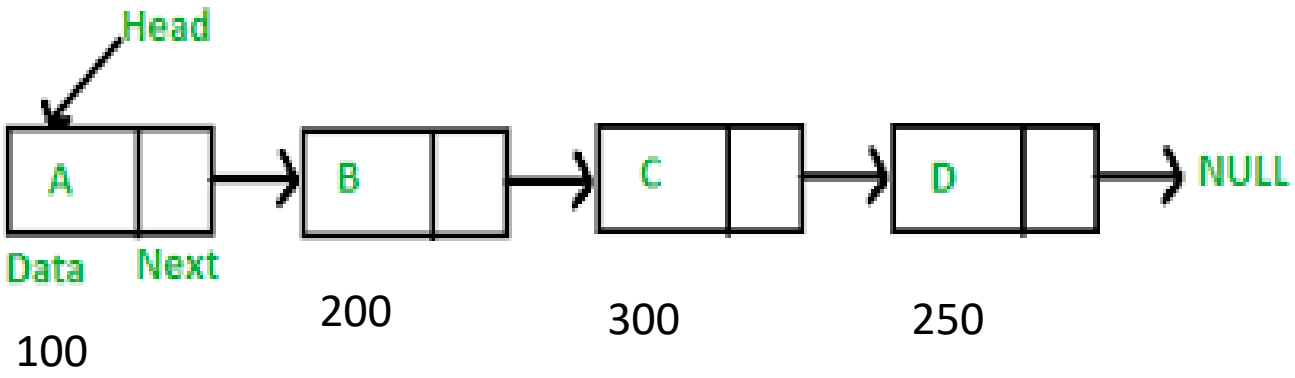


Types of Linear Data Structure…

- **LINKED LIST**

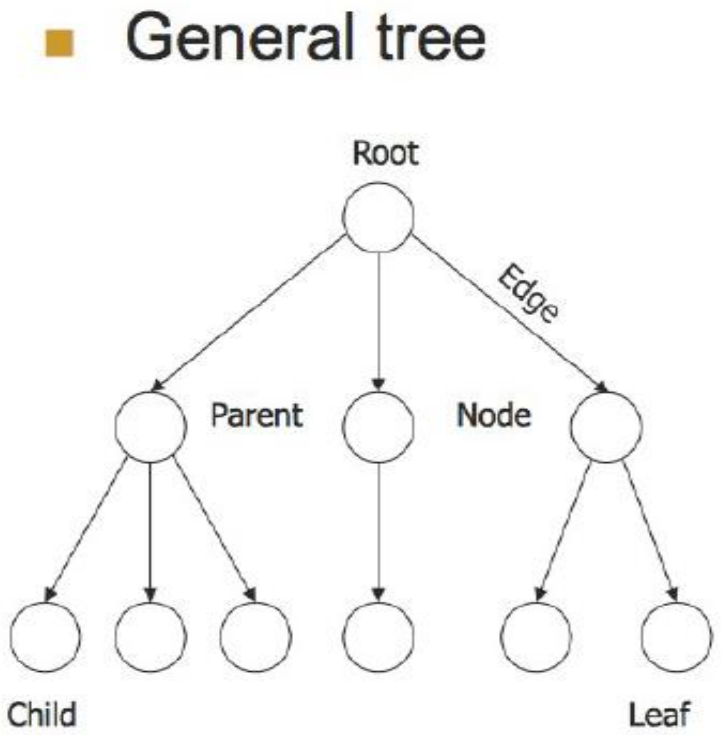
When we want to work with an unknown number of data values, we use a linked list data structure to organize that data.

The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".



Types of NON Linear Data Structure

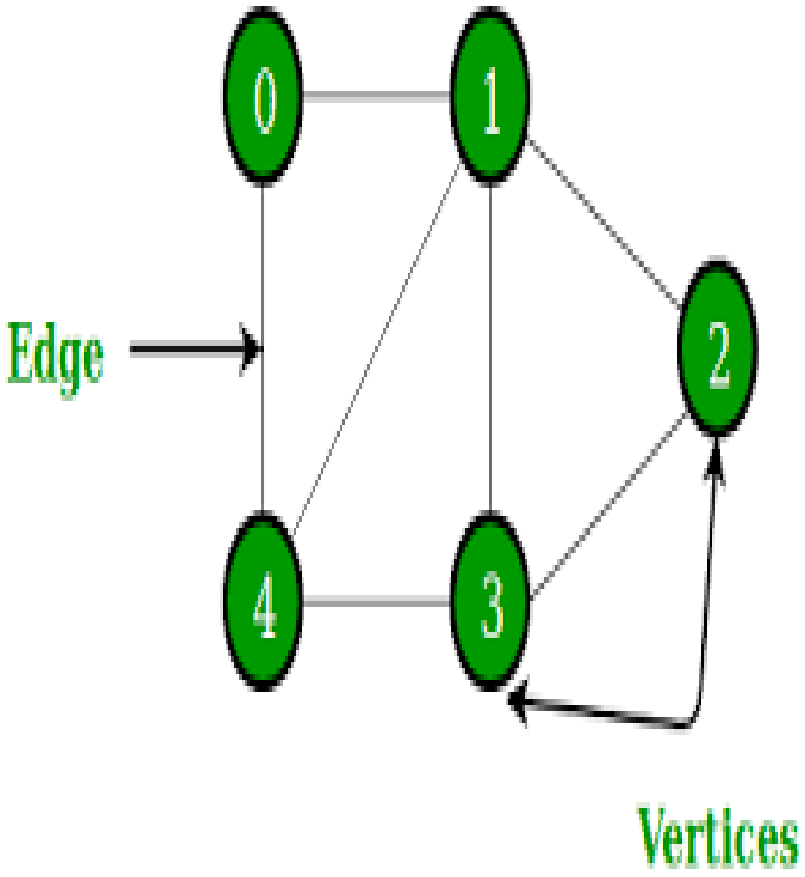
- **Tree** is a non-linear data structure which organizes data in hierarchical structure.



Types of NON Linear Data Structure..

- **Graph** is a non-linear data structu
  It contains a set of points known a
  nodes (or vertices) and a set of lin
  known as edges (or Arcs). Here ed
  are used to connect the vertices.

- Generally, a graph G is represente
  as G = (V , E), where V is set of
  vertices and E is set of edges.

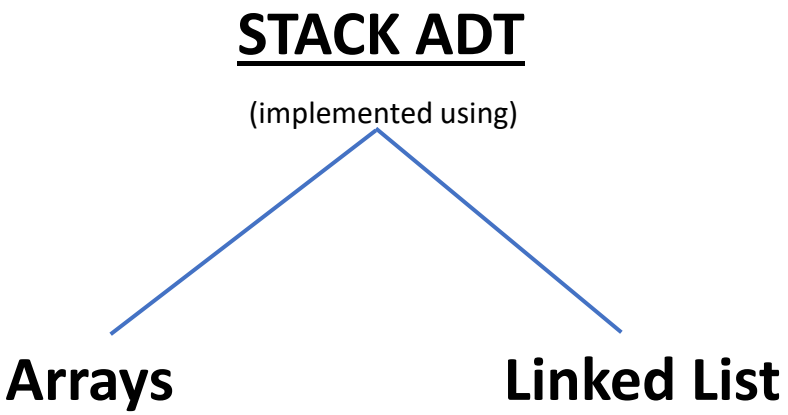


Data Structures(CO1)

- A **data structure** is the organization of the data in a way so that it can be used **efficiently**.
- The choice of a particular data model depends on two considerations.
  - It must be rich enough in structure to mirror the actual relationships of the data in the real world.
  - The structure should be simple enough that one can effectively process the data when necessary.
- In other words, data structure is used to implement an ADT.
  - Ex: stack ADT can be implemented using array data structure or linked list data structure.
- ADT tells us What is to be done and data structures tells us How to do it.

Which Data Structures? (CO1)

- In reality, different implementations of ADT are compared for time and space efficiency. The one best suited according to the current requirement of the user will be selected.



Advantages of Data Structures (CO1)

- **Efficiency-** proper choice of DS make program efficient in terms of space and time.
- **Reusability-** one implementation can be used by multiple client programs.
- **Abstraction-** data structure is specified by an ADT which provides a level of abstraction. The client program doesn't have to worry about the implementation details.

Data Structure Operations (CO1)

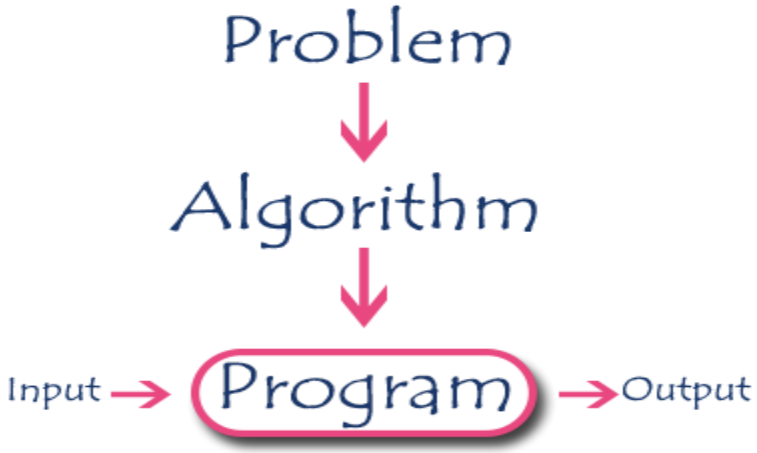
- **Traversing** : Accessing each record exactly once
- **Searching**: Finding the location of the record with a given key value
- **Inserting**: Adding a new record to the structure.
- **Deleting**: Removing a record from the structure.
- **Sorting**: Arranging the records in some logical order
- **Merging**: Combining the records in two different sorted files into a single sorted file

Algorithm (CO1)

- An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Every Algorithm must satisfy the following properties:
  - **Input**- There should be 0 or more inputs supplied externally to the algorithm.
  - **Output**- There should be at least 1 output obtained.
  - **Definiteness**- Every step of the algorithm should be clear and well defined.
  - **Finiteness**- The algorithm should have finite number of steps.
  - **Correctness**- Every step of the algorithm must generate a correct output.

Algorithms

- Algorithm is a **step-by-step** procedure, which defines a set of instructions to be executed in a **certain order** to get the **desired output.**
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.



Characteristics of an Algorithm

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code

Example:

An algorithm to sum N natural numbers.

```
Sum(A[],n)
{
        s=0;
        for(i=1 to n) do
                s=s+A[i]
        return s

}
```

An algorithm to add to matrix

```
Matrixadd(A[][],B[][],C[][], m, n)

{
        for(i=1 to n) do
                for(j=1 to n) do

c[i][j]=a[i][j]+b[i][j]

}
```

Example

Sum of n natural number

1. Take input an array from the user.
2. Initialize sum = 0
3. Repeat step 4 from 1 to size of the array
4. Add all the elements in the sum one by one
   sum = sum+ a[i]
5. Print sum

Efficiency of Algorithm(CO1)

- An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space
- In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms
- The performance of an algorithm is measured on the basis of following properties :
  - Time Complexity
  - Space Complexity

Time and Space Complexity(CO1..)

- **Time complexity**
  - Time Complexity is a way to represent the amount of time required by the program to run till its completion.
  - It's generally a good practice to try to keep the time required minimum, so that our algorithm completes its execution in the minimum time possible.
  - We can compare the time complexity of the data structures on the basis of operations performed on them
    - Ex: Inserting an element at the beginning of the list is way faster in Linked list than Arrays.
- **Space complexity**
  - It is the amount of memory space required by the algorithm, during the course of its execution.
  - An algorithm generally requires space for following components
    - **Instruction Space**
    - **Data Space**
    - **Environment Space**

Space Complexity

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

1.To store program instructions.

2.To store constant values.

3.To store variable values.

4.And for few other things like function calls, jumping statements etc,.

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

Time Complexity

- Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.
  The time complexity of an algorithm can be defined as follows...

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

Example 1

```
def sum(a, b)

   c = a+b;
    return c;
```

Space Complexity –

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

That means, totally it requires 6 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be *Constant Space Complexity*.

Time Complexity –

It requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution

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

Example 2

```
def sum(A[ ], n)
  sum = 0
  for i in range(n):
        sum = sum + A[i]
  return sum;
```

In the above piece of code it requires

'n*2' bytes of memory to store array variable 'a[ ]'

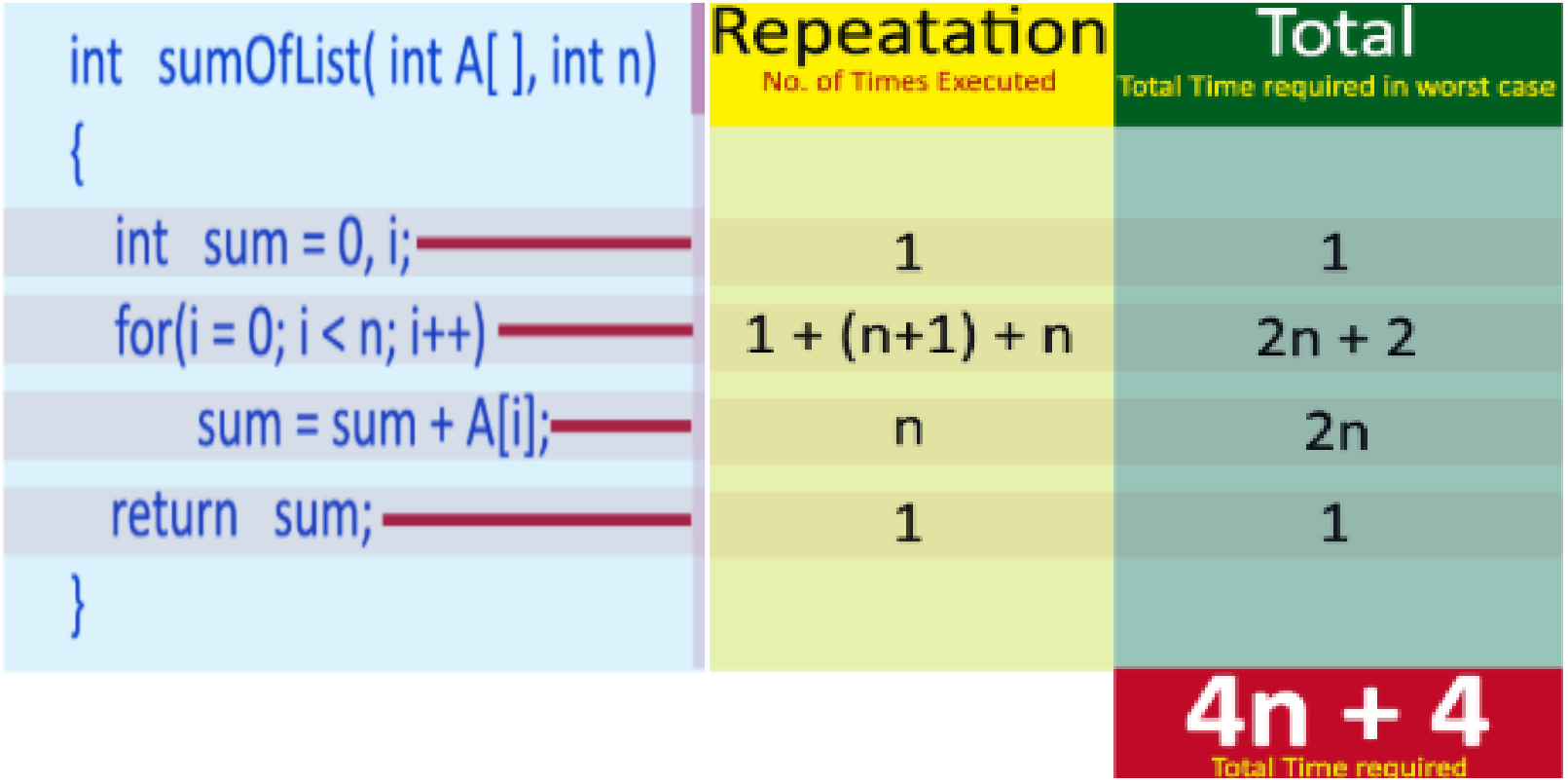
2 bytes of memory for integer parameter 'n'

4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)

2 bytes of memory for return value.

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

Introduction to Algorithms(CO1)



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

Asymptotic Notations(CO1)

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Usually, the time required by an algorithm falls under three types −
  - **Best Case** − Minimum time required for program execution.
  - **Average Case** − Average time required for program execution.
  - **Worst Case** − Maximum time required for program execution.

Q: HOW TO FIND THE TIME COMPLEXITY OR RUNNING TIME OF AN OPERATION PERFORMED ON DATA STRUCTURE?

Asymptotic Notations(CO1)

- **METHOD #1: Examine the exact running time**

Turn the timer on and Run the operation for different inputs on the data structures you want to compare one by one on different machines.

see how much time a particular operation will take on these data structures. The one who takes less time is the best.

Problem:

It might be possible that-

performance depends upon the input size and machines on which the data structures are performing operations.

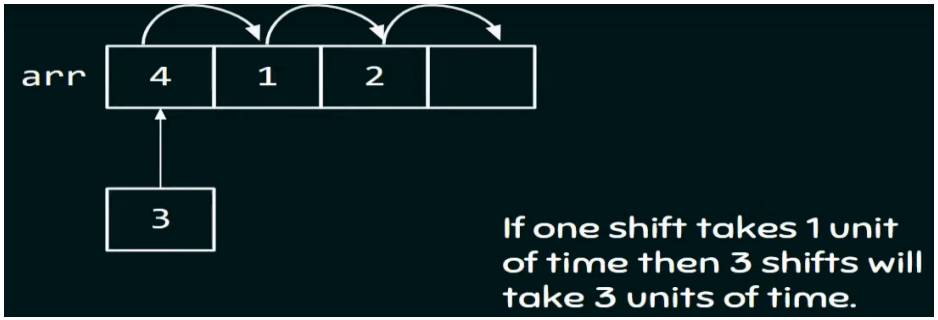
Therefore

This is not the best solution to calculate the time complexity.

Asymptotic Notations(CO1)

Running time depend on the input size:

Example: if we have to insert “3” at the beginning of this array. We have to shift all the elements towards the end.



After shift:

3	4	1	2

But what if we have 10000 elements in an array?

it requires 10000 shifts which will take 10000 units of time.

Asymptotic Notations(CO1)

Running time depend on the input size:

Therefore, if the size of the input is n, then f(n) is a function of n denotes the time complexity.

F(n) = number of instructions executed for the input value n.

Purpose of Asymptotic Notations

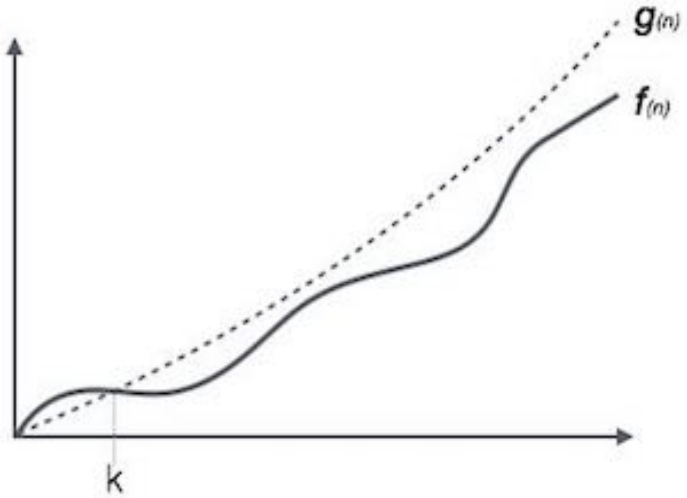
1. To simplify the running time of the function
2. To describe the behavior of the function.
3. To provide asymptotic bound.
4. To describe how the running time of an algorithm increases with increase in input size

Asymptotic Notations(contd..)

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

Big Oh Notation, O

- 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.

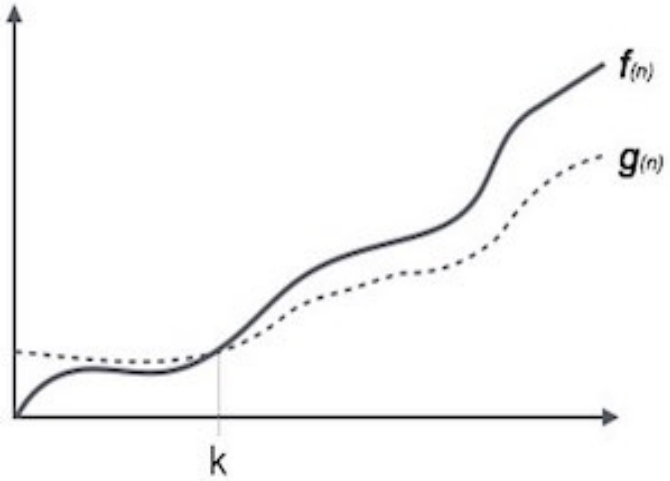


- $k=n_0$
- For example, for a function ***f*(n)**

  O(*f*(n)) = { *g*(n) : there exists c > 0 and n_0 such that *f*(n) ≤ c.*g*(n) for all n > n_0. }

Omega Notation, Ω

- 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.

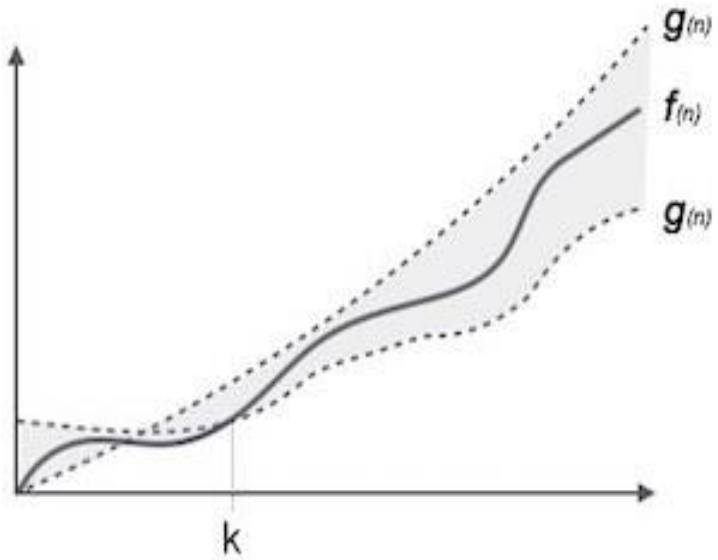


- $k=n_0$
- For example, for a function $f(n)$

  $\Omega(f(n)) \geq \{ g(n)$: there exists $c > 0$ and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$

Theta Notation, θ

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.



- $k=n_0$
- For example, for a function ***f*(n)**

  θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }

Small o Notation(o)

- Small-o, commonly written as **o**, is an Asymptotic Notation to denote the upper bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

$o(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$0 <= f(n) < c*g(n)$ for all $n >= n_0\}$

- Another definition for o notation:

  $f(n) = o(g(n))$ if and only if

  $\lim_{n\to\infty} f(n)/g(n)=0$

Small omega Notation(w)

- Small-omega, commonly written as **w**, is an Asymptotic Notation to denote the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

  w(g(n)) = { f(n): there exist positive constants c and n_0 such that

  f(n) > c*g(n)>=0 for all n >= n_0}

- Another definition for o notation:

  f(n) = o(g(n)) if and only if

  $\lim_{n\to\infty} g(n)/f(n)=0$

Common Asymptotic Notations

constant	O(1)
logarithmic	O(log n)
linear	O(n)
n log n	O(n log n)
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$n^{O(1)}$
exponential	$2^{O(n)}$

Time-space trade-off (CO1)

- Most computers have a large amount of space, but not infinite space.
- Also, most people are willing to wait a little while for a big calculation, but not forever.
- So if your problem is taking a long time but not much memory, a space-time trade-off would let you use more memory and solve the problem more quickly.
- Or, if it could be solved very quickly but requires more memory than you have, you can try to spend more time solving the problem in the limited memory.

Abstract Data Types (ADT) (CO1)

- The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations.
- The keyword "**Abstract**" is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user.
- The ADT is made of with primitive data types, but operation logics are hidden.
- ADT can be defined as:
  - **A user defined data types which defines operations on values using functions without specifying what is there inside the function and how the operations are performed.**
- Some examples of ADT are **Stack, Queue**, **List** etc.

Abstract Data Types (contd..)

- Let us see some operations of those mentioned ADT –
  - Stack –
    - **isFull(),** This is used to check whether stack is full or not
    - **isEmpty(),** This is used to check whether stack is empty or not
    - **push(x),** This is used to push x into the stack
    - **pop(),** This is used to delete one element from top of the stack
  - Queue –
    - **isFull(),** This is used to check whether queue is full or not
    - **isEmpty(),** This is used to check whether queue is empty or not
    - **insert(x)**, This is used to add x into the queue at the rear end
    - **delete(),** This is used to delete one element from the front end of the queue
- Think of ADT as a black box which hides the inner structure and design of the data types from user.
- There are multiple ways to implement an ADT. (using array, linked list)

WHY ADT?

- The program which uses data structure is called a client program.
- It has access to the ADT i.e. interface.
- The program which implements the data structure is known as the implementation.
  - If someone wants to use stack in the program, then he simply use PUSH and POP operations without knowing its implementation.
  - Also, if in future, the implementation of stack is changed from array to linked list, then the client program will work in the same way without being affected.

Topic Objective

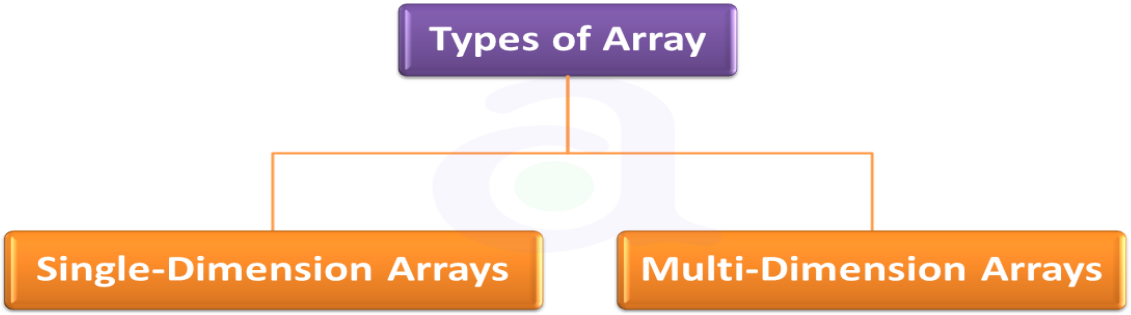
- To learn about arrays and its types: single dimensional array and multi-dimensional array.
- Understand basic representation of array.
- To study applications of arrays.
- To learn sparse matrices and their representations.

Array V/s List

Sr. No	List	Array
1.	The list can store the value of different types.	It can only consist of value of same type.
2.	The list cannot handle the direct arithmetic operations.	It can directly handle arithmetic operations.
3.	We need to import the array before work with the array.	The lists are the build-in data structure so we don't need to import it.
4.	The lists are less compatible than the array to store the data.	An array are much compatible than the list.
5.	It consumes a large memory.	It is a more compact in memory size comparatively list.
6.	It is suitable for storing the longer sequence of the data item.	It is suitable for storing shorter sequence of data items.
7.	We can print the entire list using explicit looping.	We can print the entire list without using explicit looping.
8.	It can be nested to contain different types of elements.	It must contain either all nested elements of same size.

Arrays (CO1)

- An array is a collection of variables of the same type that are referred to through a common name.
- A specific element in an array is accessed by an index.
- An array is the derived data type.
- Consist of contiguous memory locations.
- Lowest address corresponds to first element while highest address corresponds to last element.
- Can have data item of type int, float, char, double etc.
- In Python, all arrays consist of contiguous memory locations.



Need for an Array

- To store large number of array of variables of same type under a single variable.
- Eg.

  To store Marks of 50 Students

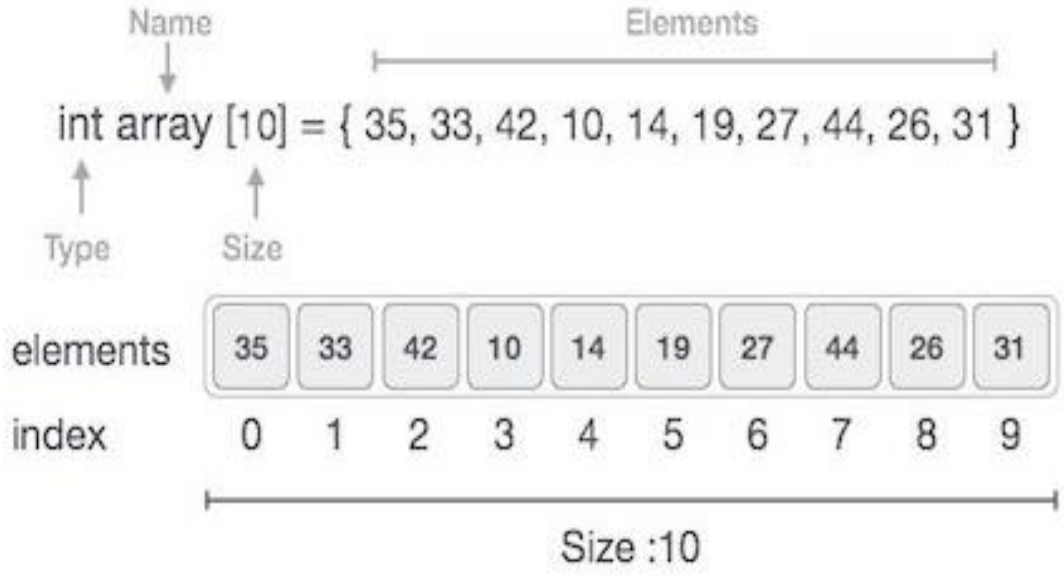
  Record of sales of 100 salesman

Array Declaration

- Like other variables, arrays must be explicitly declared so that the compiler can allocate space for them in memory.
- The general form for declaring a single-dimension array is:
  - datatype var_name[size];
    - Here, **type** declares the base type of the array, which is the type of each element in the array
    - **size** defines how many elements the array will hold.

Array Declaration (Example)

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 27.



Array Creation in Python

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *
arrayName = array(typecode, [initializers])
```

Array Creation in Python (Cont.)

Typecode are the codes that are used to define the type of value the array will hold. Some common type codes used are:

TYPE CODE	VALUE
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Array Creation in Python (Example)

The below code creates an array named array1.

```
from array import *

array1 = array('i', [10,20,30,40,50])

for x in array1:
  print(x)
```

When we compile and execute the above program, it produces the following result –

Output

```
10
20
30
40
50
```

Accessing elements of an Array

We can access each element of an array using the index of the element. The below code shows how

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1[0])

print (array1[2])
```

When we compile and execute the above program, it produces the following result

Output

```
10
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.insert(1,60)
for x in array1:
  print(x)
```

Insertion Operation (Cont.)

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position

Output

10

60

20

30

40

50

Insertion Operation(CO1)

Insert element in array

- Example

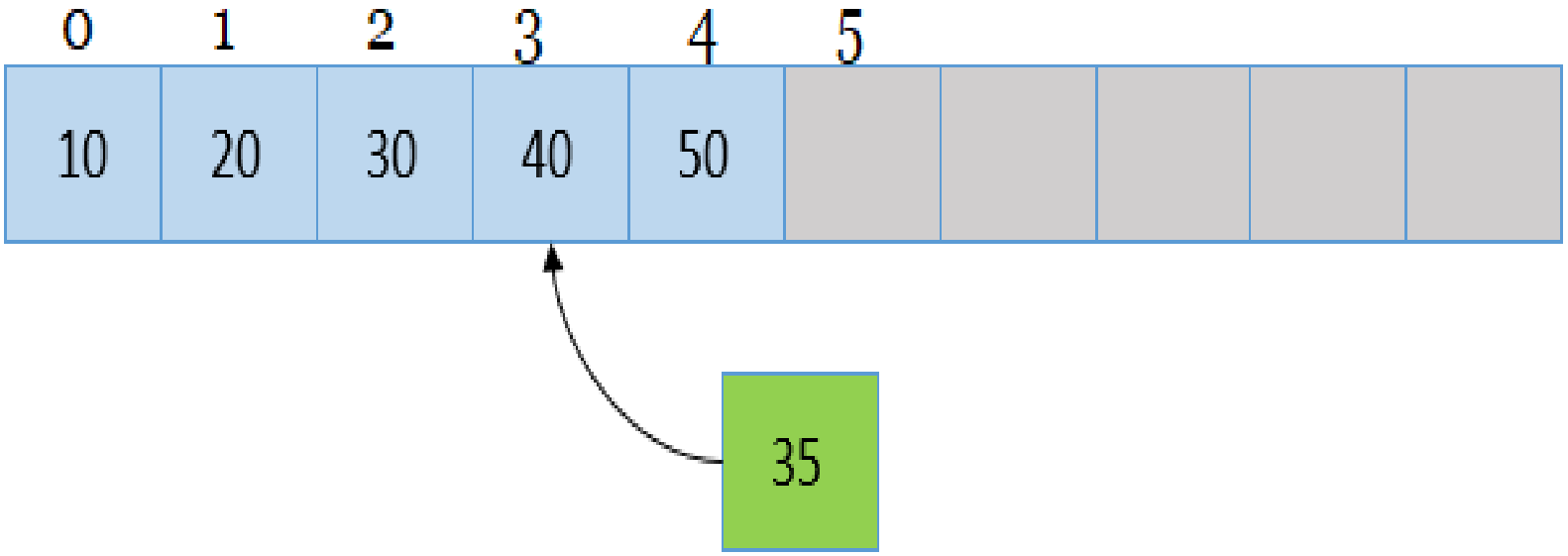
- Input

Input array elements: 10, 20, 30, 40, 50
Input element to insert: 35
Input position where to insert: 4
Output

Elements of array are: 10, 20, 30, 35, 40, 50

Insertion Operation(CO1)

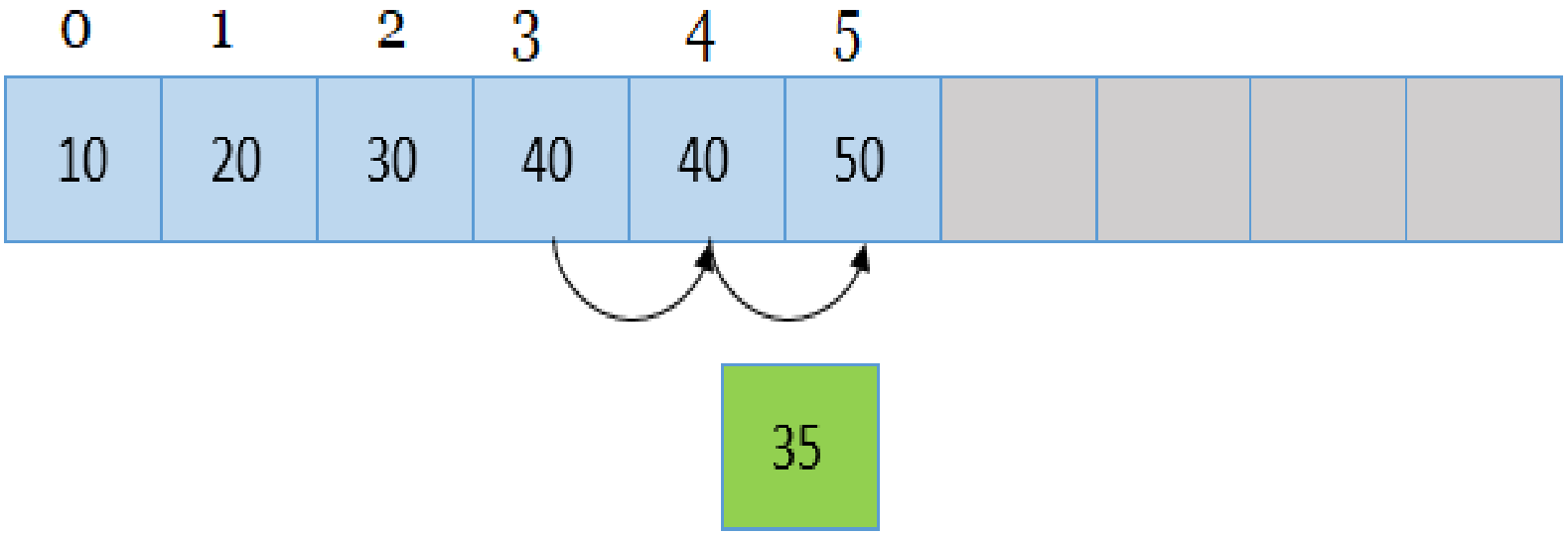
Logic to insert element in array



Insert an element at position 4.

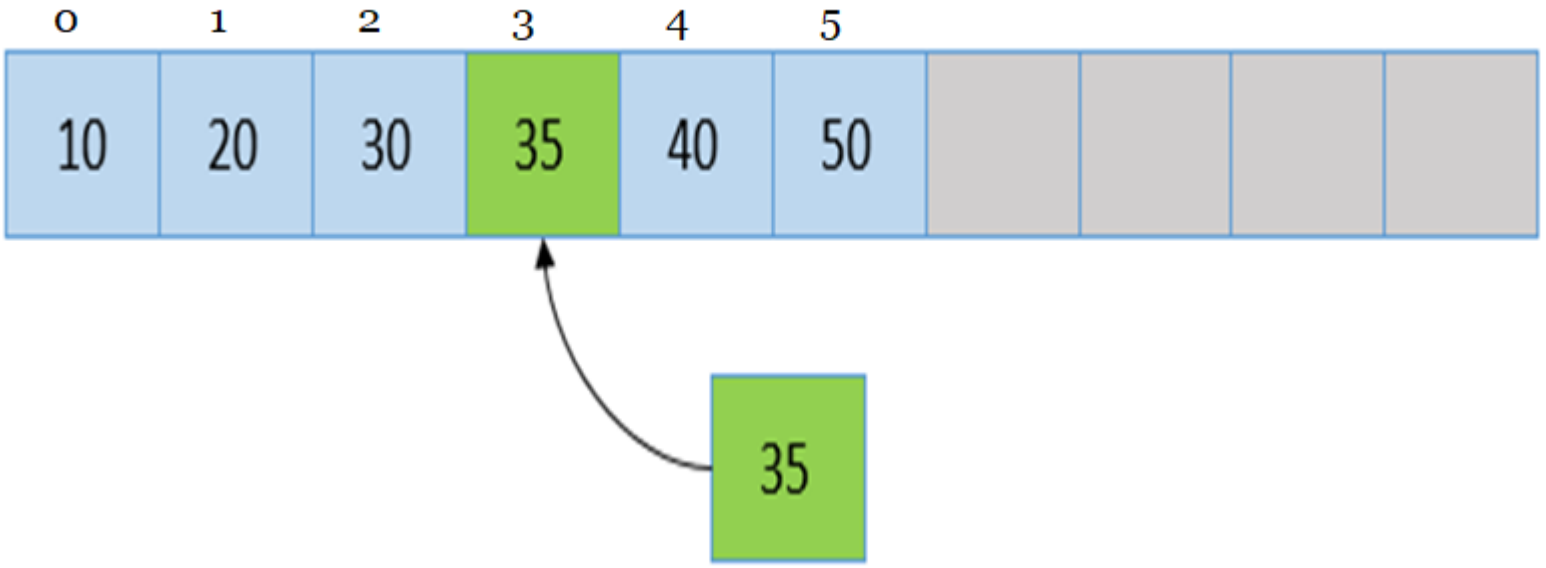
Insertion Operation (CO1)

Logic to insert element in array




```
for(i=size; i>=pos; i--)
    arr[i] = arr[i - 1];
```

Insertion Operation (CO1)



arr[pos - 1] = num;

Algorithm for Insertion at k_{th} Position

INSERT (A, N, K, ITEM).

Here, A is a linear array with N elements and K is a positive integer such that K <= N. This algorithm inserts an element ITEM into the Kth position in A.

Step 1: Set J:= N.	[Initialize counter.]
Step 2: Repeat steps 3 and 4 while J >= K:	
Step 3: Set A[J+1] := A[J].	[Move element downward.]
Step 4: J:= J-1.	[Decrease counter.]
	[End of step 2 loop.]
Step 5: Set A[K] := ITEM.	[Insert element.]
Step 6: Set N:= N+1.	[Reset N.]
Step 7: Exit.	

Array Insertion (CO1)

Python Program to insert element in array

```
n=int(input("Enter how many elements you want:"))
print("Enter numbers in array:")
l=[]
for i in range(n):
        a=int(input("num:"))
         l.append(a)
        print("ARRAY:",l)
x=int(input("Enter position you want to enter element:"))
y=int(input("Enter the element you want to enter:"))
l.insert(x,y)
print(l)
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.remove(40)
for x in array1:
 print(x)
```

Deletion Operation (Cont.)

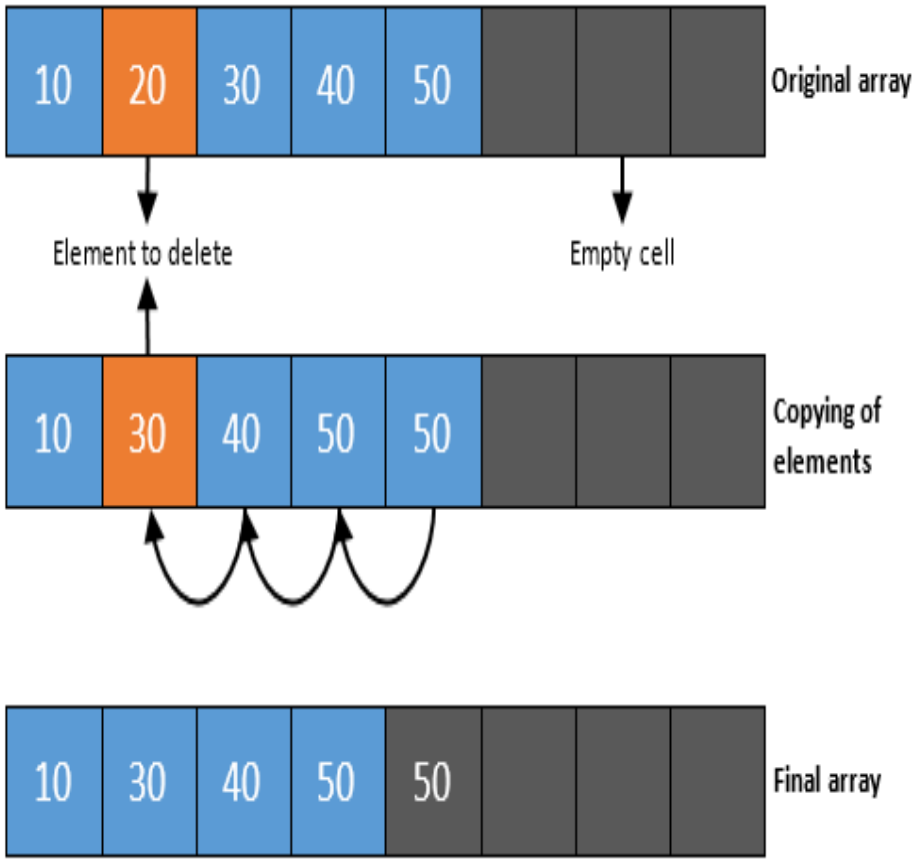
When we compile and execute the above program, it produces the following result which shows the element is removed from the array.

Output

10
20
30
50

Deletion Operation (CO1)

Logic to remove element from array



Step by step descriptive logic to remove element from array.

Move to the specified location which you want to remove in given array.

Copy the next element to the current element of array. Which is you need to perform array[i] = array[i + 1].

Repeat above steps till last element of array.

Finally decrement the size of array by one.

Algorithm for Deletion at k_{th} Position

- DELETE (A, K, N, ITEM).
- Here, A is a linear array with N elements and K is a positive integer such that K<= N. This algorithm deletes the Kth element from A.

Step I: Set ITEM:= A[K] and J:= K. [Initialize counter.]
Step II: Repeat steps 3 and 4 while N>J:
Step III: A[J] := A[J+1]. [Move element upward.]
Step IV: Set J:= J+1. [Increase counter.]
[End of step 2 loop.]

Step V: Set N:= N-1. [Reset N.]
Step VI: Exit.

Introduction to Array(CO1)

Program to delete an element from an array

```
n=int(input("Enter how many elements you want:"))
print("Enter numbers in array:")
l=[]
for i in range(n):
        a=int(input("num:"))        l.append(a)

print("ARRAY:",l)
b=int(input("Enter position you want to delete element:"))
l.pop(b)

print(l)
```

Search Operation

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1.index(40))
```

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

Output

3

Algorithm for searching in linear array

Linear Search for array with n elements (Array A, Value x)

Step 1: Set i
Step 2: if i > n 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 3 until i less than equal to n
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Time Complexity: O(n)

Search Operation

Linear Search

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1[2] = 80

for x in array1:
 print(x)
```

Update Operation (Cont.)

When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

Output

```
10
20
80
40
50
```

Algorithm for min-max

Algorithm: Max-Min-Element (numbers[])

```
max := numbers[1]
min := numbers[1]
for i = 2 to n do
    if numbers[i] > max then
          max := numbers[i]
    if numbers[i] < min then
          min := numbers[i]
return (max, min)
```

Time Complexity: O(n)

Introduction to Array (CO1)

Address Calculation in single (one) Dimension Array:

Actual Address of the **1st** element of the array is known as **Base Address (B)**

Here it is 1100

Memory space acquired by every element in the Array is called **Width (W)**

Here it is 4 bytes

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	**15**	**7**	**11**	**44**	**93**	**20**
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Lower Limit/Bound of Subscript **(LB)**

Introduction to Array (CO1)

- Array of an element of an array say “A[I]” is calculated using the following formula:
- **Address of A [I] = B + W * (I – LB)**
- Where,
  B = Base address
  W = Storage Size of one element stored in the array (in byte)
  I = Subscript of element whose address is to be found
  LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

Introduction to Array (CO1)

Example:

- Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.
- Solution:

The given values are: B = 1020, LB = 1300, W = 2, I = 1700

Address of A [I] = B + W * (I – LB)

= 1020 + 2 * (1700 – 1300)

= 1020 + 2 * 400

= 1020 + 800

= 1820 [Ans]

Introduction to Array (CO1)

- Find the base address of an array **A[-15:65]**. The size of each element is 2 bytes in the memory and the location of A[37] is 4279.

Solution:

The given values are: LB = -15, UB = 65, W = 2, I = 37, A[37] = 4279

Address of A [I] = B + W * (I – LB)

4279= B + 2 * (37 – (-15))

4279= B + 2 * 52

4279= B + 104

B= 4175 [Ans]

Multidimensional Array (CO1)

- We can create an array of arrays.
- These arrays are known as multidimensional arrays.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

- For example,
  - float x[3][4];
    - Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns as shown in figure.

Multidimensional Array (CO1)

Initializing Two-Dimensional Arrays

- Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

  A = [[0, 1, 2, 3] , [4, 5, 6, 7] , [8, 9, 10, 11]]

Multidimensional Array (CO1)

Accessing Two-Dimensional Array Elements

- An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.

For example :

```
A = [[0, 1, 2, 3] , [4, 5, 6, 7] , [8, 9, 10, 11]]
for i in range(0,len(A)):
    for j in range(0,len(A[0])):
                print(A[i][j],end=" ")
    print()
```

Multidimensional Array (CO1)

To store the elements entered by user in 2D Array

```
r1 = int(input("Number of rows, m = "))
c1 = int(input("Number of columns, n = "))
// create matrix of all zeros
a = [[0 for i in range(c1)]for j in range(r1)]

for i in range(r1):
        for j in range(c1):
                print("Entry in row:", (i+1),"column:" ,(j+1))
        a[i][j]=int(input())
```

Introduction to Array (CO1)

To store the elements entered by user in 2D Array

```
r = int(input("Enter num of rows"))
c = int(input("Enter num of col"))
arr = []
for i in range(r):
    l = []
    for j in range(c):
        print("Enter data in row ",i+1,"col ",j+1)
        v = int(input())
        l.append(v)
    arr.append(l)
print(arr)
```

Multidimensional Array (CO1)

Python Program to Transpose given matrix

```
r1 = int(input("Number of rows, m = "))
c1 = int(input("Number of columns, n = "))
a = [[0 for i in range(c1)]for j in range(r1)]
b = [[0 for i in range(r1)]for j in range(c1)]
for i in range(r1):
    for j in range(c1):
        print("Entry in row:", (i+1),"column:", (j+1))
        a[i][j]=int(input())

for i in range(r1):
    for j in range(c1):
        b[j][i] = a[i][j]
```

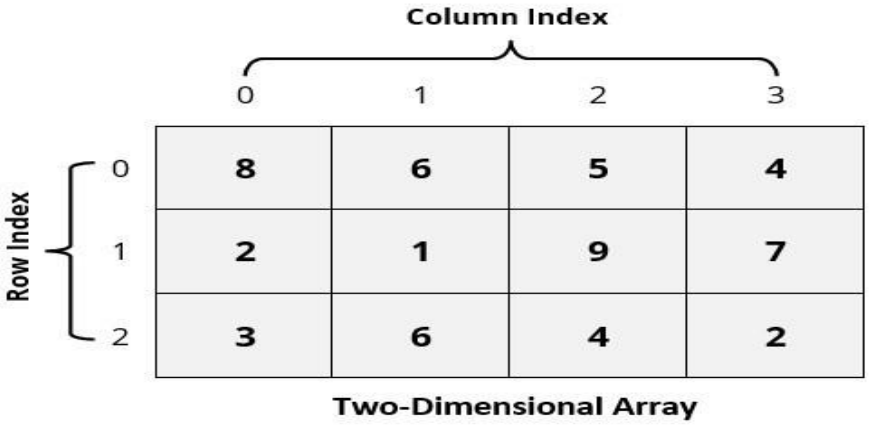
```
for i in range(r1):
    for j in range(c1):
        print(a[i][j], end = " ")
    print()
print("Second Matrix")
for i in range(c1):
    for j in range(r1):
        print(b[i][j], end = " ")
    print()
```

Multidimensional Array

- Similarly, you can declare a three-dimensional (3d) array.
- For example,
  - float y[2][4][3];
    - Here, the array y can hold 24 elements.

2D Array (CO1)

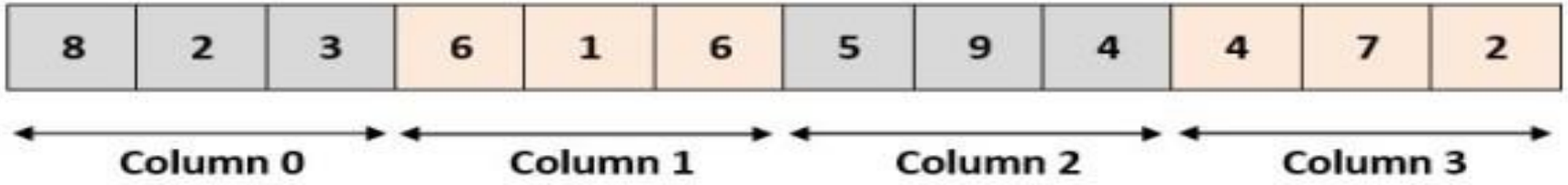
Address Calculation in Two Dimensional Array:



2D Array (CO1)

Row-Major (Row Wise Arrangement)

Column-Major (Column Wise Arrangement)



2D Array (CO1)

Address of an element of any array say "A[I][J]" is calculated in two forms as given:

Row Major System:

Address of $A[I_1][I_2] = B + W * [(I_1 - L_1) * N_2 + (I_2 - L_2)]$

Column Major System:

Address of $A[I_1][I_2]$ Column Major Wise $= B + W * [(I_1 - L_1) + N_1 * (I_2 - L_2)]$

Where,
B = Base address
$\mathbf{I_1}$ = Row subscript of element whose address is to be found
$\mathbf{I_2}$ = Column subscript of element whose address is to be found
W = Storage Size of one element stored in the array (in byte)
$\mathbf{L_1}$ = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
$\mathbf{L_2}$ = Lower limit of column/start column index of matrix, if not given assume 0 (zero)
$\mathbf{N_1}$ = Number of row of the given matrix
$\mathbf{N_2}$ = Number of column of the given matrix

Row-Major Order

- Row Major Order is a method of representing multi dimension array in sequential memory.
- In this method elements of an array are arranged sequentially row by row.
- Thus elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.
- Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory row by row as shown below:

Row 0	A[0,0]	A[0,1]	.....	A[0,M-1]
Row 1	A[1,0]	A[1,1]	.....	A[1,M-1]
	.....			
Row N-1	A[N-1,0]	A[N-1,1]	.....	A[N-1,M-1]

Row-Major Order

- Example:
  Consider following example in which a two dimensional array consist of two rows and four columns is stored sequentially in row major order as:

2000	A[0][0]	Row 0
2002	A[0][1]	
2004	A[0][2]	
2006	A[0][3]	
2008	A[1][0]	Row 1
2010	A[1][1]	
2012	A[1][2]	
2014	A[1][3]	

Row-Major Order

- The Location of element A[i, j] can be obtained by evaluating expression:
  - **LOC (A [i, j]) = Base_Address + W [M (i-L1) + (j-L2)]**
    - **Here,**
      - **Base_Address** is the address of first element in the array.
      - **W** is the word size. It means number of bytes occupied by each element.
      - **N** is number of rows in array.
      - **M** is number of columns in array.
      - **L1** is lower bound of row.
      - **L2** is lower bound of column.

Row-Major Order

- Suppose we want to calculate the address of element A [1, 2].
  - It can be calculated as follow:
    - **Here,**

      Base_Address = 2000, W= 2, M=4, N=2, i=1, j=2

      LOC (A [i, j])=**Base_Address + W [M (i-L1) + (j-L2)]**

      LOC (A[1, 2])=2000 + 2 *[4*(1) + 2]

      =2000 + 2 * [4 + 2]

      =2000 + 2 * 6

      =2000 + 12

      =2012

Column Major Order

- Column Major Order is a method of representing multidimensional array in sequential memory.
  In this method elements of an array are arranged sequentially column by column. Thus elements of first column occupies first set of memory locations reserved for the array, elements of second column occupies the next set of memory and so on.
  Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory column by column as shown below:

Column 0	**A[0,0]**	**A[1,0]**		**A[N-1,0]**
Column 1	A[0,1]	A[1,1]		A[N-1,1]
Column N-1	A[0,M-1]	A[1,M-1]		A[N-1,M-1]

Column Major Order

- Example:
  Consider following example in which a two dimensional array consist of two rows and four columns is stored sequentially in Column Major Order as:

2000	A[0][0]	Column 0
2002	A[1][0]	
2004	A[0][1]	Column 1
2006	A[1][1]	
2008	A[0][2]	Column 2
2010	A[1][2]	
2012	A[0][3]	Column 3
2014	A[1][3]	

Column Major Order

- The Location of element A[i, j] can be obtained by evaluating expression:
  - **LOC (A [i, j]) = Base_Address + W [N (j-L2) + (i-L1)]**
    - **Here,**
      - **Base_Address** is the address of first element in the array.
      - **W** is the word size. It means number of bytes occupied by each element.
      - **N** is number of rows in array.
      - **M** is number of columns in array.
      - **L1** is lower bound of row.
      - **L2** is lower bound of column.

Column Major Order

- Suppose we want to calculate the address of element A [1, 2].
  - It can be calculated as follow:
    - **Here,**

      Base_Address = 2000, W= 2, M=4, N=2, i=1, j=2

      LOC (A [i, j])=Base_Address + W [N (j-L2) + (i-L1)]

      LOC (A[1, 2])=2000 + 2 *[2*(2) + 1]

      =2000 + 2 * [4 + 1]

      =2000 + 2 * 5

      =2000 + 10

      =2010

Examples

Q 1. An array X [-15…….10, 15……40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

As you see here the number of rows and columns are not given in the question. So they are calculated as:

Number or rows say $N_1 = (U_1 – L_1) + 1 = [10 – (- 15)] +1 = 26$

Number or columns say $N_2 = (U_2 – L_2) + 1 = [40 – 15)] +1 = 26$

(i) **Column Major Wise Calculation of above equation**

The given values are: B = 1500, W = 1 byte, $I_1 = 15$, $I_2 = 20$, $L_1 = -15$, $L_2 = 15$, $N_1 = 26$

Address of A [I_1][I_2] =B + W * [($I_1 – L_1$) + N_1 * ($I_2 – L_2$)]

= 1500 + 1 * [(15 – (-15)) + 26 * (20 – 15)] = 1500 + 1 * [30 + 26 * 5] = 1500 + 1 * [160] = 1660 [Ans]

Examples

Q 1. An array X [-15…….10, 15……40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

(ii) Row Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, $I_1 = 15$, $I_2 = 20$, $L_1 = -15$, $L_2 = 15$, $N_2 = 26$

Address of A [I_1][I_2] =B + W * [($I_1 – L_1$)* N_2 + ($I_2 – L_2$)]

= 1500 + 1* [26 * (15 – (-15))) + (20 – 15)] = 1500 + 1 * [26 * 30 + 5] = 1500 + 1 * [780 + 5] = 1500 + 785

= 2285 [Ans]

Memory Location Formula (CO1)

Memory Location Formula for N- Dimensional Array

Formula for 1 D Array –

$$A [I_1] = B + W * (I_1 – L_1)$$

Where I_1 _ Index whose location we want to find.

B – Base Address

W – Size of Data Type

L_1 – Lower Bound of the array

Let $(I_1 – L_1) = E_1$ (Effective index of I_1)

Where E_i – Effective index of I_i

$$A [I_1] = B + W * [E_1]$$

Memory Location Formula (CO1)

Memory Location Formula for N- Dimensional Array

Formula for 2 D Array –

$$A[I_1][I_2] = B + W * [(I_1 - L_1)*N_2 + (I_2 - L_2)]$$

Where I_1, I_2 _ Index whose location we want to find.

B – Base Address

W – Size of Data Type

L_1, L_2 – Lower Bound of the array

Let $(I_1 - L_1) = E_1$ (Effective index of I_1); $(I_2 - L_2) = E_2$

Where E_i – Effective index of I_i

$$A[I_1][I_2] = B + W * [E_1*N_2 + E2]$$

Memory Location Formula (CO1)

Memory Location Formula for N- Dimensional Array

Formula for 2 D Array –

$$A[I_1][I_2] = B + W * [E_1 * N_2 + E_2]$$

Formula for 3 D Array –

$$A[I_1][I_2][I_3] = B + W * [(E_1 * N_2 + E2) * N_3 + E_3]$$

Formula for 4 D Array –

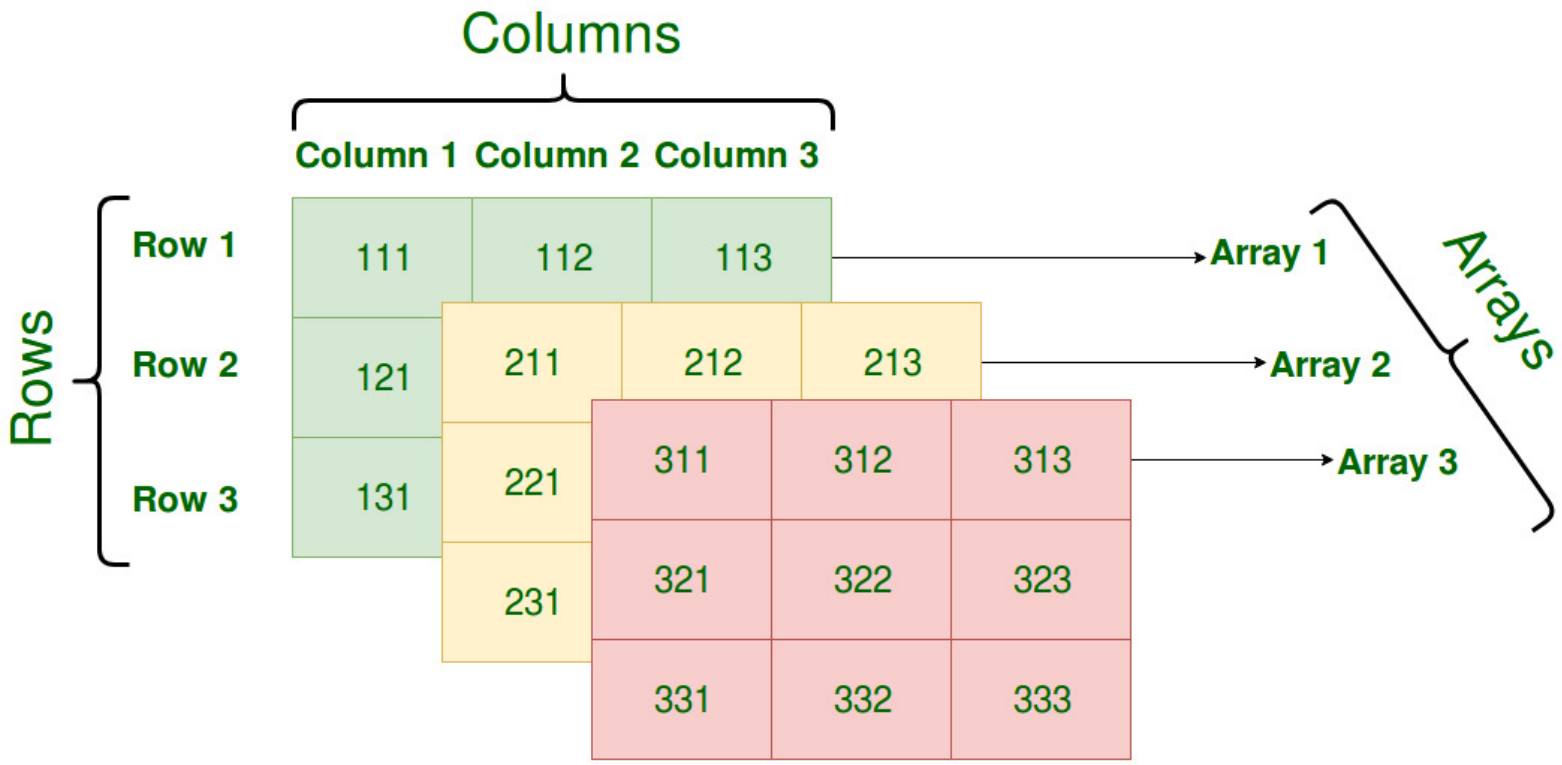
$$A[I_1][I_2][I_3][I_4] = B + W * [((E_1 * N_2 + E_2) * N_3 + E_3) * N_4 + E_4]$$

So on for N- Dimensional array ..

$$A[I_1][I_2][I_3][I_4]....[I_n] = B + W * [(((E_1 * N_2 + E_2) * N_3 + E_3) * N_4 + E_4....E_{n-1}) * Nn + E_n]$$

Multidimensional Array (CO1)

Three-Dimensional Array



3D Array (CO1)

Three-Dimensional Array

Given an array [1..8, 1..5, 1..7] of integers. Calculate address of element A[5,3,6], by using rows wise methods, if Base Address=900?

Solution:- The dimensions of A are :

$N_1=8$, $N_2=5$, $N_3=7$, $I_1=5$, $I_2=3$, $I_3=6$, W=2,L1=1,L2=1,L3=1

Rows - wise :

Location $(A[I_1, I_2, I_3]) = B + W*[(E_1N_2+E_2)*N_3+E_3]$

$= B + W*[((I_1-L_1)*N_2+(I_2-L_2))*N_3+(I_3-L_3)]$

Location(A[5,3,6])= 900 + 2*[((5-1)*5+(3-1))*7+(6-1)]
= 900 + 2*[(4*5+2)*7+5]
= 900 +2*159
= 1218

Matrix Multiplication Algorithm

- Step 1: Start
  Step 2: Declare matrix A[m][n], B[p][q], and C[m][q]
  Step 3: Read m, n, p, q
  Step 4: Check if matrix can be multiplied or not.
  (If n is not equal to p, generate error msg)
  Step 5: Read A[][] and B[][]
  Step 6: Declare variable i=0, k=0, j=0 and sum=0
  Step 7: Repeat until i<m
  Step 7.1: Repeat until j<q
  Step 7.1.1: Repeat until k<p
  set sum = sum+ A[i][k] * B[k][j]
  set C[i][j]= sum
  set sum = 0 and k=k+1
  Step 7.1.2: Set j=j+1
  Step 7.2: Set i=i+1
  Step 8: C is the required matrix
  Step 9: Stop

Time Complexity: $O(n^3)$

Application of arrays (CO1)

- Arrays are used to Store List of values
- Arrays are used to Perform Matrix Operations
- Arrays are used to implement Search Algorithms
  - Linear search
  - Binary search
- Arrays are used to implement Sorting Algorithms
  - Insertion sort
  - Selection sort
  - Quick sort
- Arrays are used to implement Data Structures
  - Stack using array
  - Queue using array
- Arrays are also used to implement CPU Scheduling Algorithms

Sparse Matrix (CO1)

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.
- **Why to use Sparse Matrix instead of simple matrix ?**
  - **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
  - **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..
  - Example

    0 0 3 0 4
    0 0 5 7 0
    0 0 0 0 0
    0 2 6 0 0

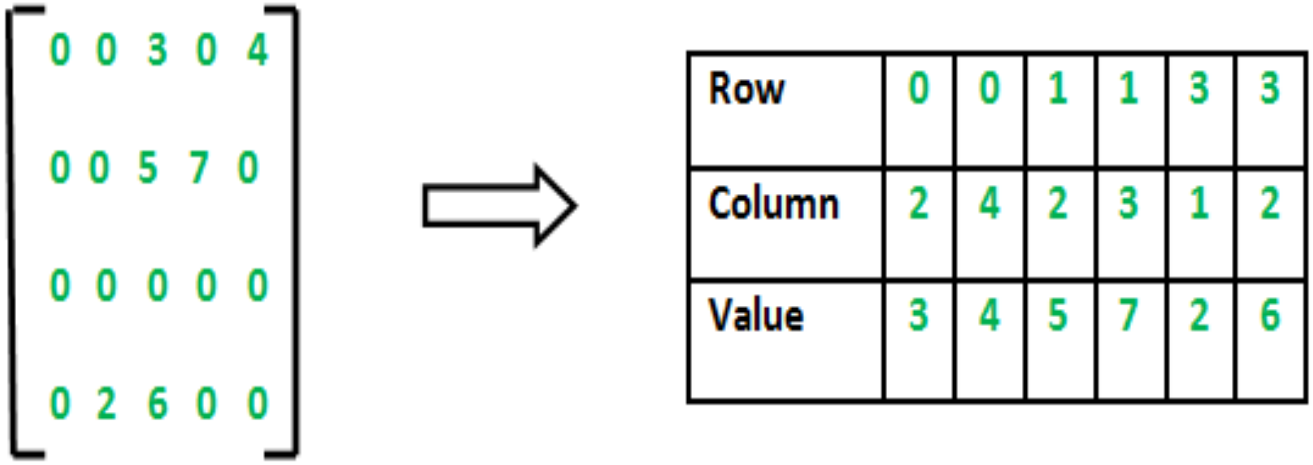
Sparse Matrix Representation

- Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

- Sparse Matrix Representations can be done in many ways following are two common representations:
  - Array representation
  - Linked list representation

Sparse Matrix Representation

- **Method 1: Using Arrays**
  - 2D array is used to represent a sparse matrix in which there are three rows named as
  - **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row, column)



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Sparse Matrix Representation

- **Method 1: Using Arrays**

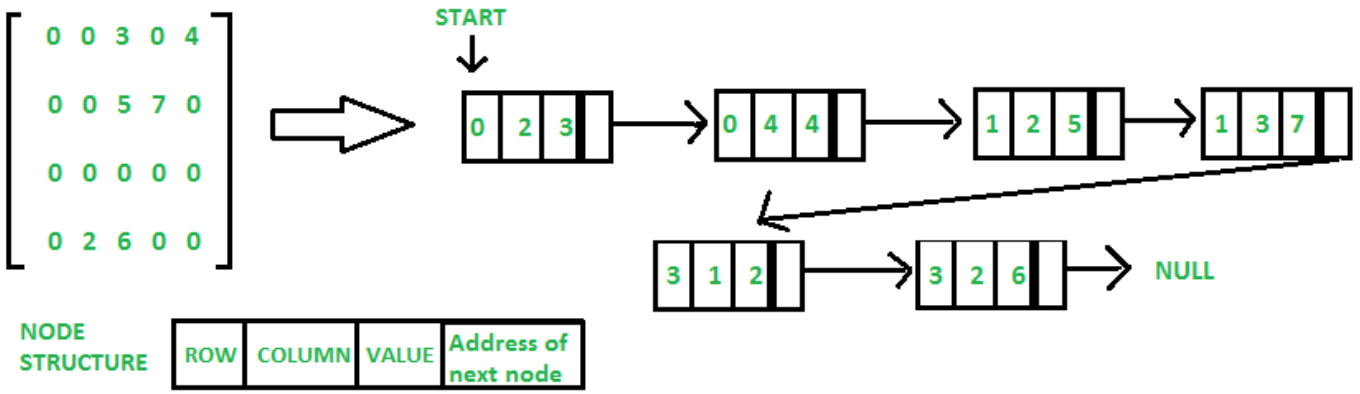
```
# Python program for Sparse Matrix Representation
# using arrays

# assume a sparse matrix of order 4*5
# let assume another matrix compactMatrix
# now store the value,row,column of arr1 in sparse matrix compactMatrix

sparseMatrix = [[0,0,3,0,4],[0,0,5,7,0],[0,0,0,0,0],[0,2,6,0,0]]

# initialize size as 0
size = 0

for i in range(4):
    for j in range(5):
        if (sparseMatrix[i][j] != 0):
            size += 1

# number of columns in compactMatrix(size) should
# be equal to number of non-zero elements in sparseMatrix
rows, cols = (3, size)
compactMatrix = [[0 for i in range(cols)] for j in range(rows)]

k = 0
for i in range(4):
    for j in range(5):
        if (sparseMatrix[i][j] != 0):
            compactMatrix[0][k] = i
            compactMatrix[1][k] = j
            compactMatrix[2][k] = sparseMatrix[i][j]
            k += 1

for i in compactMatrix:
    print(i)
```

Output

```
0 0 1 1 3 3
2 4 2 3 1 2
3 4 5 7 2 6
```

Sparse Matrix Representation

- **Method 2: Using Linked Lists**
  - In linked list, each node has four fields. These four fields are defined as:
  - **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row,column)
  - **Next node:** Address of the next node



Sparse Matrix (CO1)

Python Program that will convert 2D-representation to Sparse representation.

```
print("Enter values for Matrix ")
m=int(input("Number of rows, m = "))
n=int(input("Number of columns, n = "))
arr=[]
for i in range(m):
        l=[]
        for j in range(n):
        print("Entry in row:",i+1,"column:",j+1)

        x=int(input())
        l.append(x)
        arr.append(l)
```

```
print("Matrix  =",arr)
print("Sparse Matrix: ")
for row in range(len(arr)):
for i in range(len(arr[row])):
if arr[row][i]==0:
                        pass
else:
print(row,i,arr[row][i],"")
```

Topic Objective

- To learn Searching
  - Linear search
  - Binary search

Searching

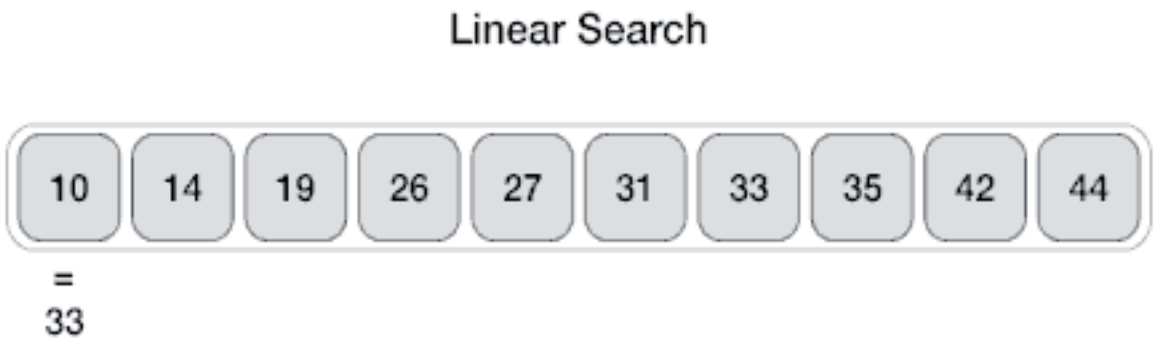
- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms

- Searching Algorithms are a family of algorithms used for the purpose of searching.
- The searching of an element in the given array may be carried out in the following two ways-
  - Linear Search
  - Binary Search

Linear Search

- Linear search is the simplest search algorithm and often called sequential search.
- In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- If the match found then location of the item is returned otherwise the algorithm return NULL.



Linear Search Algorithm

- LINEAR_SEARCH(A, N, VAL)
  - **Step 1:** [INITIALIZE] SET POS = -1
  - **Step 2:** [INITIALIZE] SET I = 1
  - **Step 3:** Repeat Step 4 while I<=N
  - **Step 4:** IF A[I] = VAL
    SET POS = I
    PRINT POS
    Go to Step 6
    [END OF IF]
    SET I = I + 1
    [END OF LOOP]
  - **Step 5:** IF POS = -1
    PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
    [END OF IF]
  - **Step 6:** EXIT

Linear Search Analysis

To write a python program linear search.

```
arr=list(map(int,input("Enter the list of numbers: ").split()))
x=int(input("The number to search for: "))
for i in range(len(arr)):
        if arr[i]==x:
                print(x,"was found at index {}.".format(i))
                break
        else:
                print(x,"was not found.")
```

Linear Search Analysis

Analysis of Linear Search

How long will our search take?

- In the best case, the target value is in the first element of the array.

So the search takes some tiny, and constant, amount of time.

- In the worst case, the target value is in the last element of the array.

So the search takes an amount of time proportional to the length of the array. O(n)

Linear Search Analysis

Analysis of Linear Search

- In the **average case**, the target value is somewhere in the array.
- In fact, since the target value can be anywhere in the array, any element of the array is equally likely.
- So on average, the target value will be in the middle of the array.
- So the search takes an amount of time proportional to half the length of the array.

Linear Search Algorithm Complexity

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

Linear Search Explained

- Linear Search
  - https://www.youtube.com/watch?v=C46QfTjVCNU&feature=youtu.be

Binary Search

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.
- Binary Search Algorithm can be applied only on **Sorted arrays**.
- So, the elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.

- Mid = floor value of (Beg+End)/2
- One of the following case must be true in every comparison.
  If A is the sorted array then,
  - Case 1: data == A[mid]
  - Case 2: data < A[mid]
  - Case 3: data > A[mid]

Binary Search Algorithm

- BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
  - **Step 1:** [INITIALIZE] SET BEG = lower_bound
    END = upper_bound, POS = - 1
  - **Step 2:** Repeat Steps 3 and 4 while BEG <=END
  - **Step 3:** SET MID = (BEG + END)/2
  - **Step 4:** IF A[MID] = VAL
    SET POS = MID
    PRINT POS
    Go to Step 6
    ELSE IF A[MID] > VAL
    SET END = MID - 1
    ELSE
    SET BEG = MID + 1
    [END OF IF]
    [END OF LOOP]

Binary Search Algorithm (contd..)

- **Step 5:** IF POS = -1
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
  [END OF IF]
- **Step 6:** EXIT

Binary Search Algorithm (contd..)

Algorithm for Binary Search Program:

Step 1 - Read the search element from the user.
Step 2 - Find the middle element in the sorted list.
Step 3 - Compare the search element with the middle element in the sorted list.
Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Binary Search Complexity

Time complexity of binary search

At each iteration, the array is divided by half. So let's say the length of array at any iteration is n

At Iteration 1,

Length of array = n 10

At Iteration 2,

Length of array = n⁄2 5

At Iteration 3,

Length of array = (n⁄2)⁄2 = n⁄2^2 ^{2}

Therefore, after Iteration k,

Length of array = n⁄2^k ^{1}

Also, we know that after

After k divisions, the length of array becomes 1

Therefore

Length of array = $n/2^k = 1$

$=> n = 2^k$

Applying log function on both sides:

$=> \log_2 (n) = \log_2 (2^k)$

$=> \log_2 (n) = k \log_2 (2)$

$=> k = \log_2 (n)$

Hence the time complexity of Binary Search is

$\log_2 (n)$

Binary Search Complexity

SN	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)

Binary Search Example

Item to be searched = 23

Step 1 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6

Step 2 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7

Step 3 →

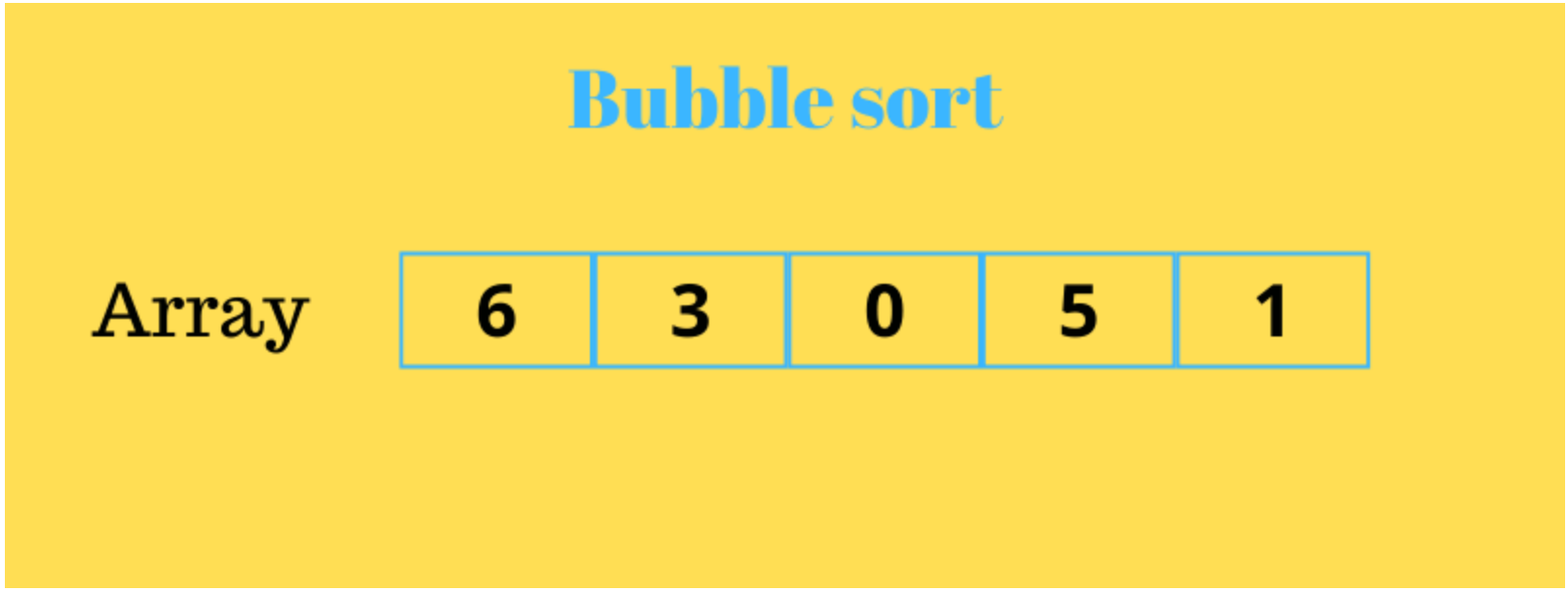
1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

a [mid] = 23
23 = 23
loc = mid

Return location 7

Sorting using Array(CO1)

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.



Sorting Using Array(CO1)

Bubble sort Illustration

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Sorting Using Array CO1)

Algorithm for Bubble Sort

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Sorting Using Array CO1)

Program for Bubble Sort

```
def bubbleSort(arr):
        n = len(arr)
        for i in range(n-1):
                for j in range(0, n-i-1):
                        if arr[j] > arr[j + 1]:
                                arr[j], arr[j + 1] = arr[j + 1], arr[j]
# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print("Sorted array is:")
for i in range(len(arr)):
        print("% d" % arr[i], end=" ")
```

Sorting using Array(CO1)

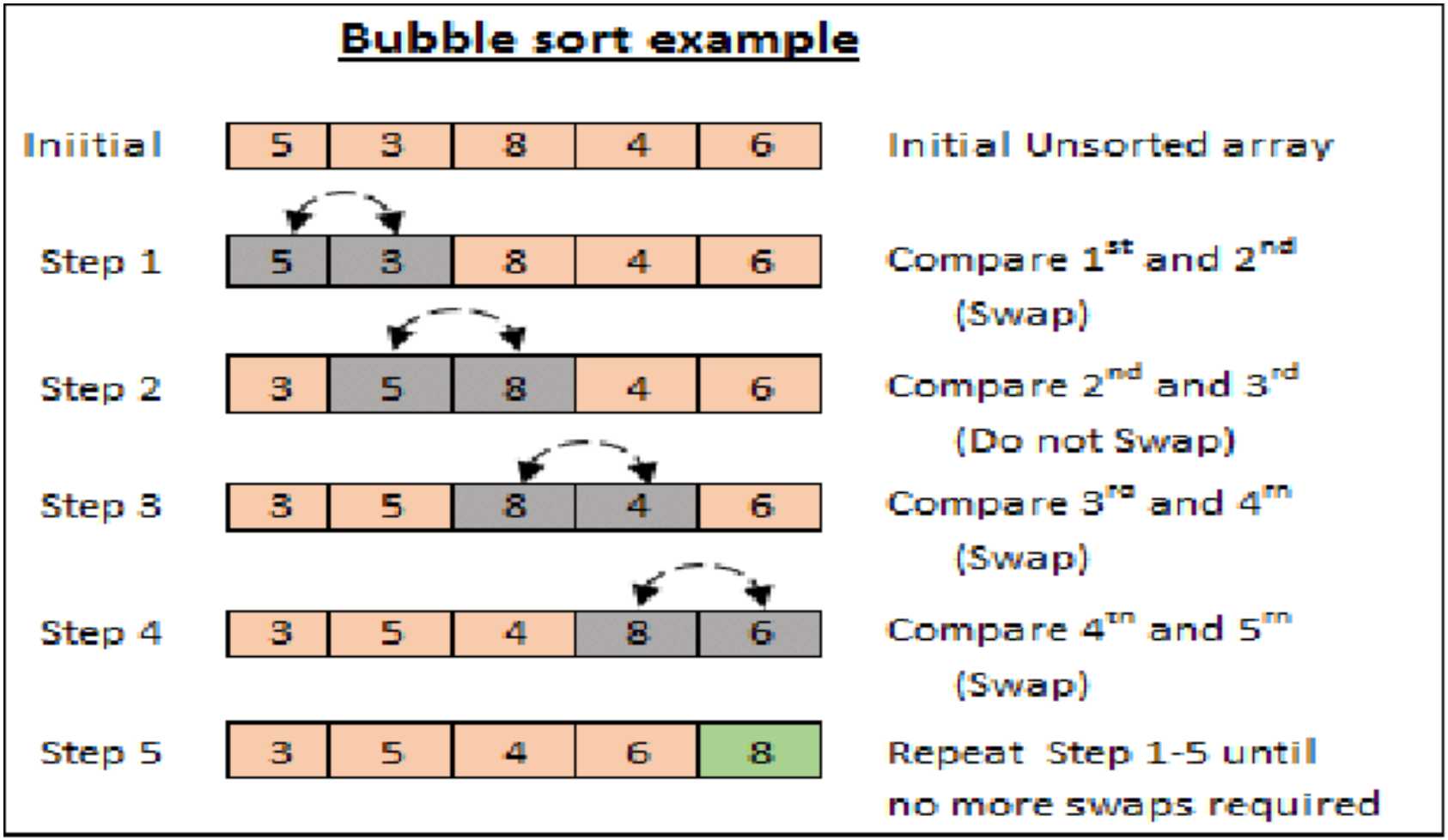
Bubble Sort

- Python Code

```
def BubbleSort(arr):
    for i in range(len(arr)-1):
        for j in range(len(arr)-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

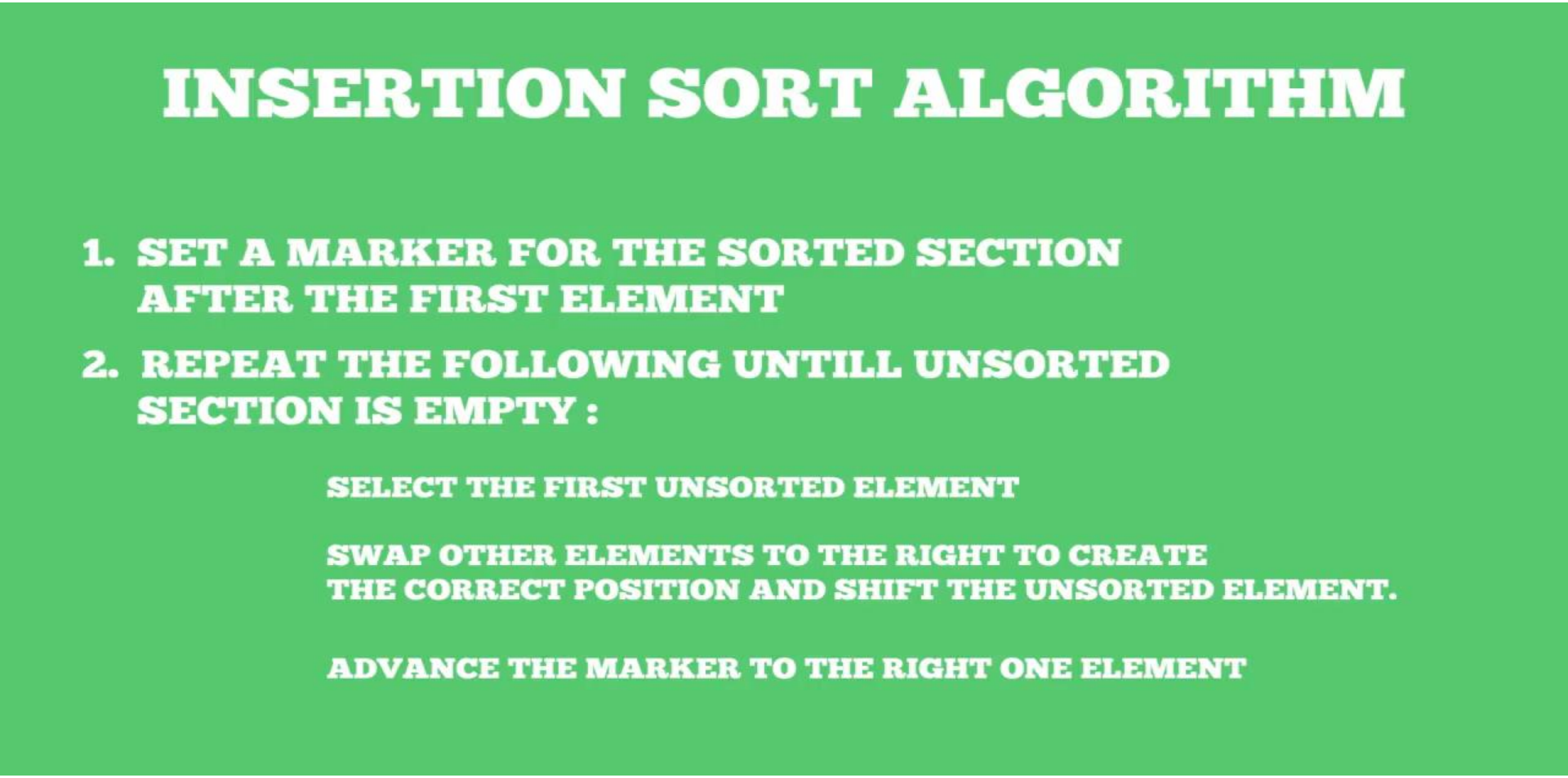
Sorting using Array(CO1)

Bubble sort Illustration



Sorting using Array(CO1)

Insertion Sort



Insertion Sort Algorithm

1. Set a marker for the sorted section, after the first element.

2. Repeat the following until unsorted section is empty:
   a) Select the first unsorted element
   b) Swap other elements to the right to create the correct position and shift the unsorted elements
   c) Advance the marker to the right one element.

- Insertion sort is Stable
- Insertion sort is in place (working on same array, no extra space taken for sorting the array)

Sorting using Array(CO1)

Insertion Sort Algorithm

40	20	60	10	50	30

Sorting using Array(CO1)

Insertion Sort

- Python Code

```
def InsertionSort(arr):
    for i in range(1, len(arr)):
        store = arr[i]
        j = i-1
        while j >=0 and store < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = store
    return arr
```

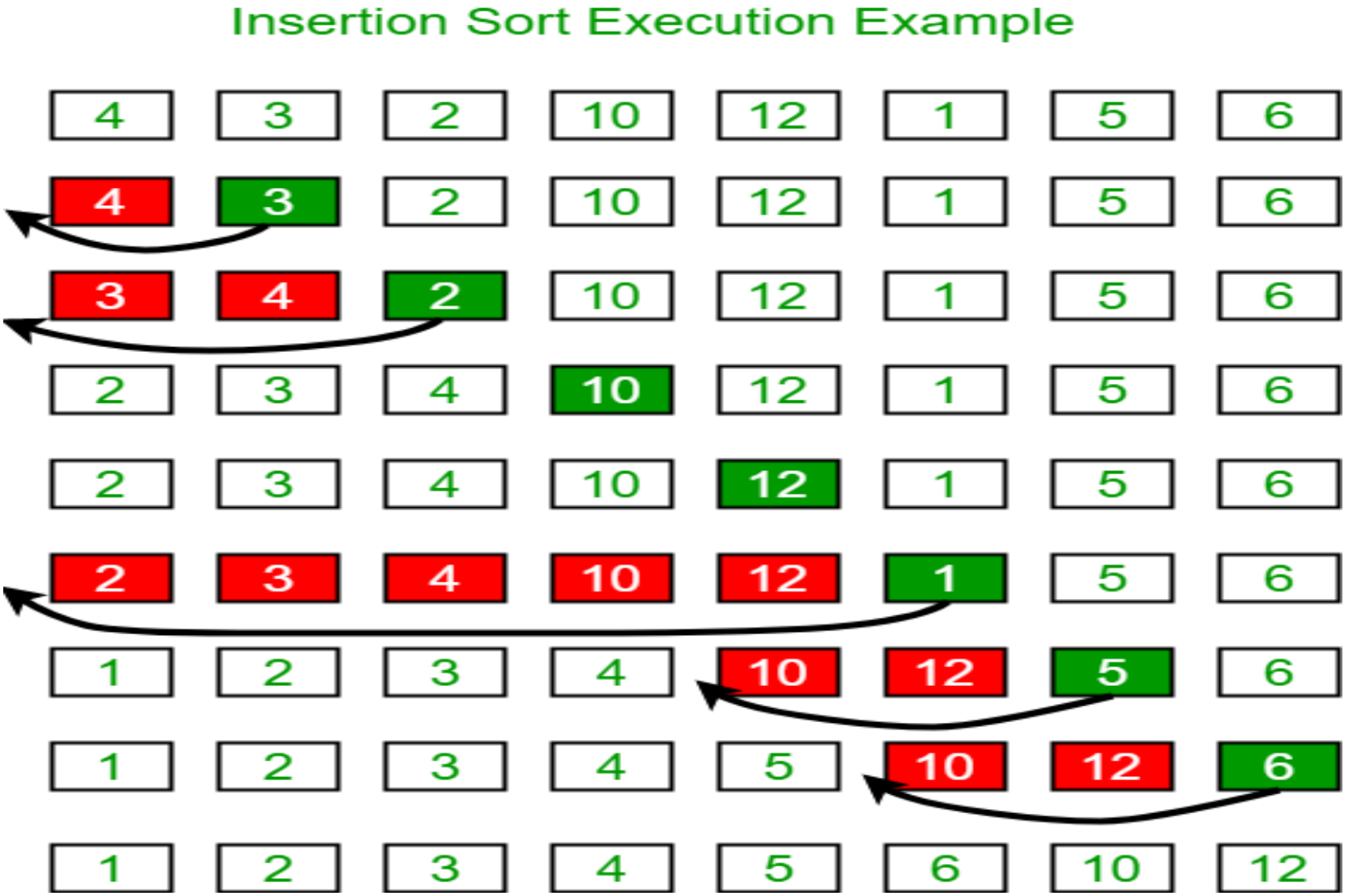
Sorting using Array(CO1)

Python Program for Insertion Sort

```
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]

        # Move elements of arr[0..i-1], that are
greater than key, to one position ahead of their
current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
```

```
#sorting the array [12, 11, 13, 5, 6]
using insertionSort
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
lst = [] #empty list to store sorted
elements
print("Sorted array is : ")
for i range(len(arr)):
  lst.append(arr[i])      #appending
the elements in sorted order
print(lst)
```

Sorting using Array(CO1)



Sorting using Array(CO1)

Selection sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

1. Select the lowest element in the remaining array
2. Bring it to the starting position
3. Change the counter for unsorted array by one.

Sorting using Array(CO1)

Selection Sort

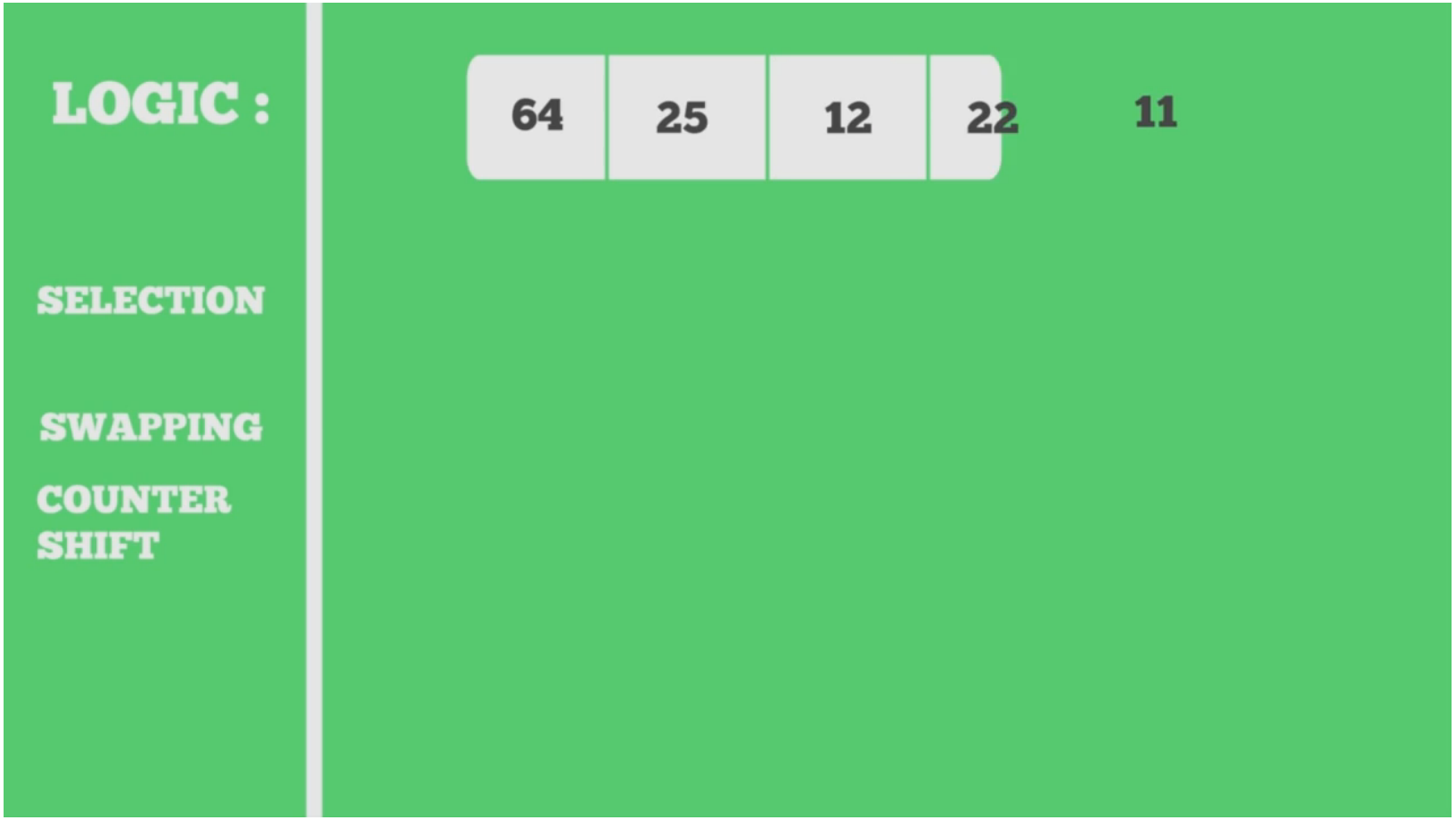
❑ Python Code

```
def SelectionSort(A):
    for i in range(len(A)):
        minind = i
        for j in range(i+1, len(A)):
            if A[minind] > A[j]:
                minind = j
        A[i], A[minind] = A[minind], A[i]
    return A
```

Time Complexity: $O(n^2)$.

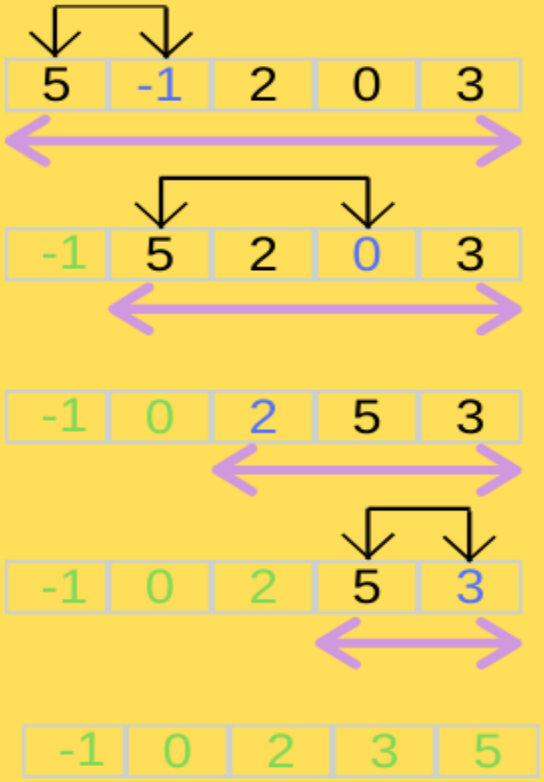
Sorting using Array(CO1)

Selection sort



Sorting using Array(CO1)

Selection Sort



Green = Sorted

Blue = Current minimum

Find minimum elements in unsorted array and swap if required (element not at correct location already).

Sorting Using Array CO1)

Algorithm for Selection Sort

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

Sorting using Array(CO1)

Python Program for Selection Sort

```
def selectionSort(array, size):

    for ind in range(size):
        min_index = ind

        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
         # swapping the elements to sort the array
        (array[ind], array[min_index]) = (array[min_index], array[ind])

arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
```

Sorting using Array(CO1)

Merge sort

- Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Sorting using Array(CO1)

Divide And Conquer

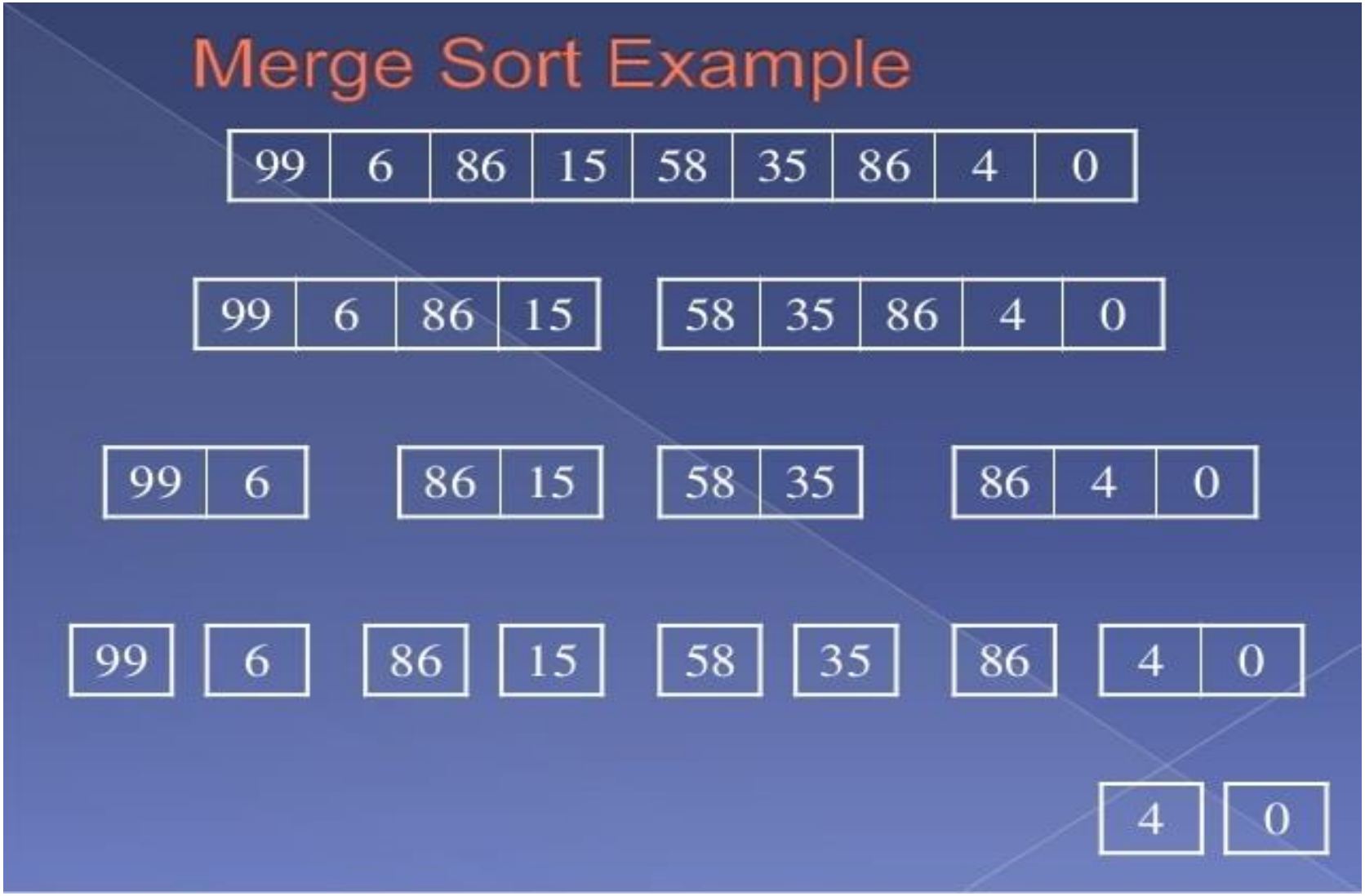
1. Divide: Divide the unsorted list into two sub lists of about half the size.
2. Conquer: Sort each of the two sub lists recursively until we have list sizes of length 1,in which case the list itself is returned.
3. Combine: Merge the two-sorted sub lists back into one sorted list.

Sorting using Array(CO1)

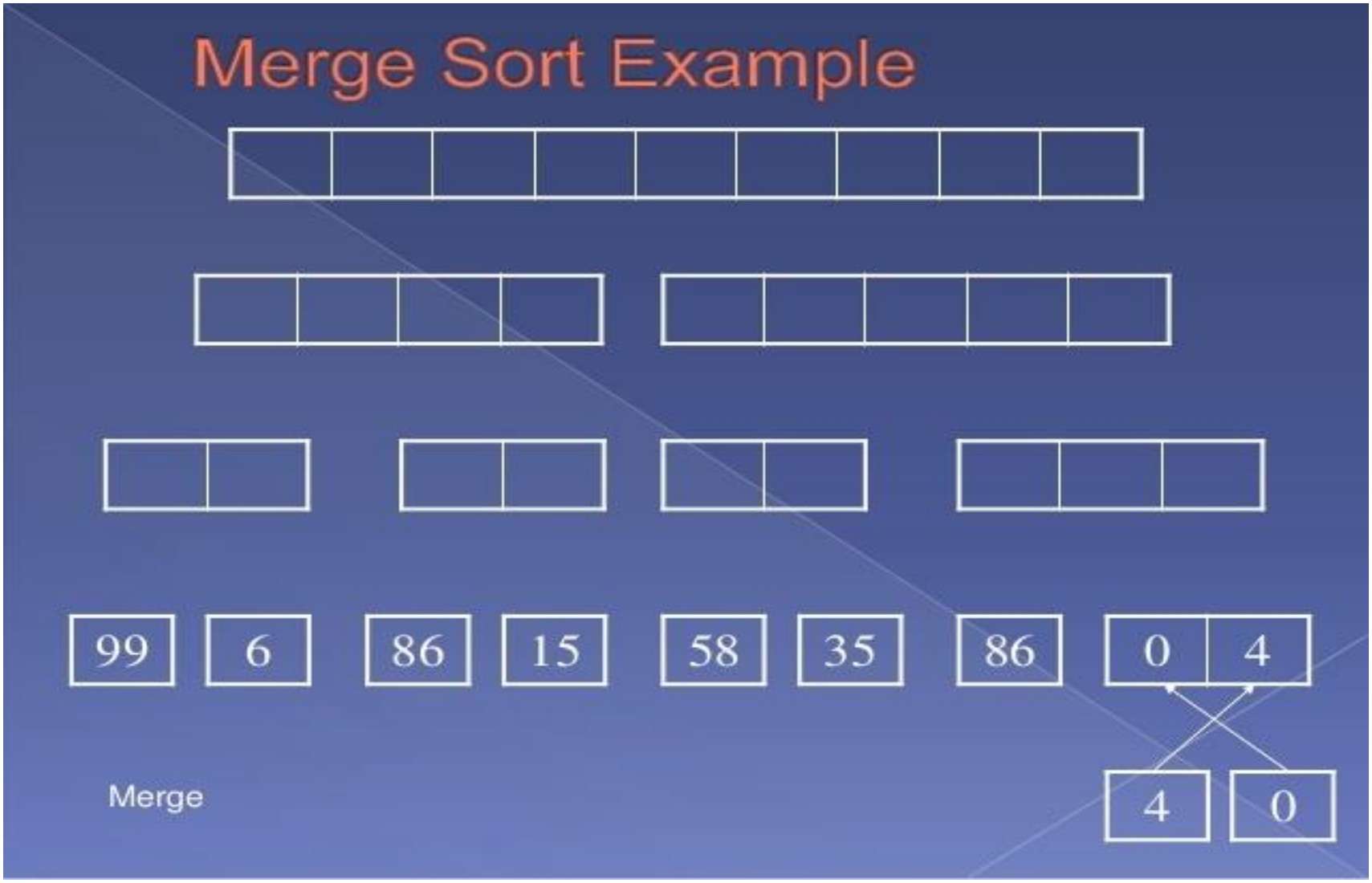
MERGE SORT ALGO

- MERGE SORT (*A,p,r*) //divide
  if p < r
  then *q*= [(p + r) / 2]
  MERGE SORT(*A,p,q*)
  MERGER SORT(*A,q* + 1,*r*)
  MERGE(*A,p,q,r*)

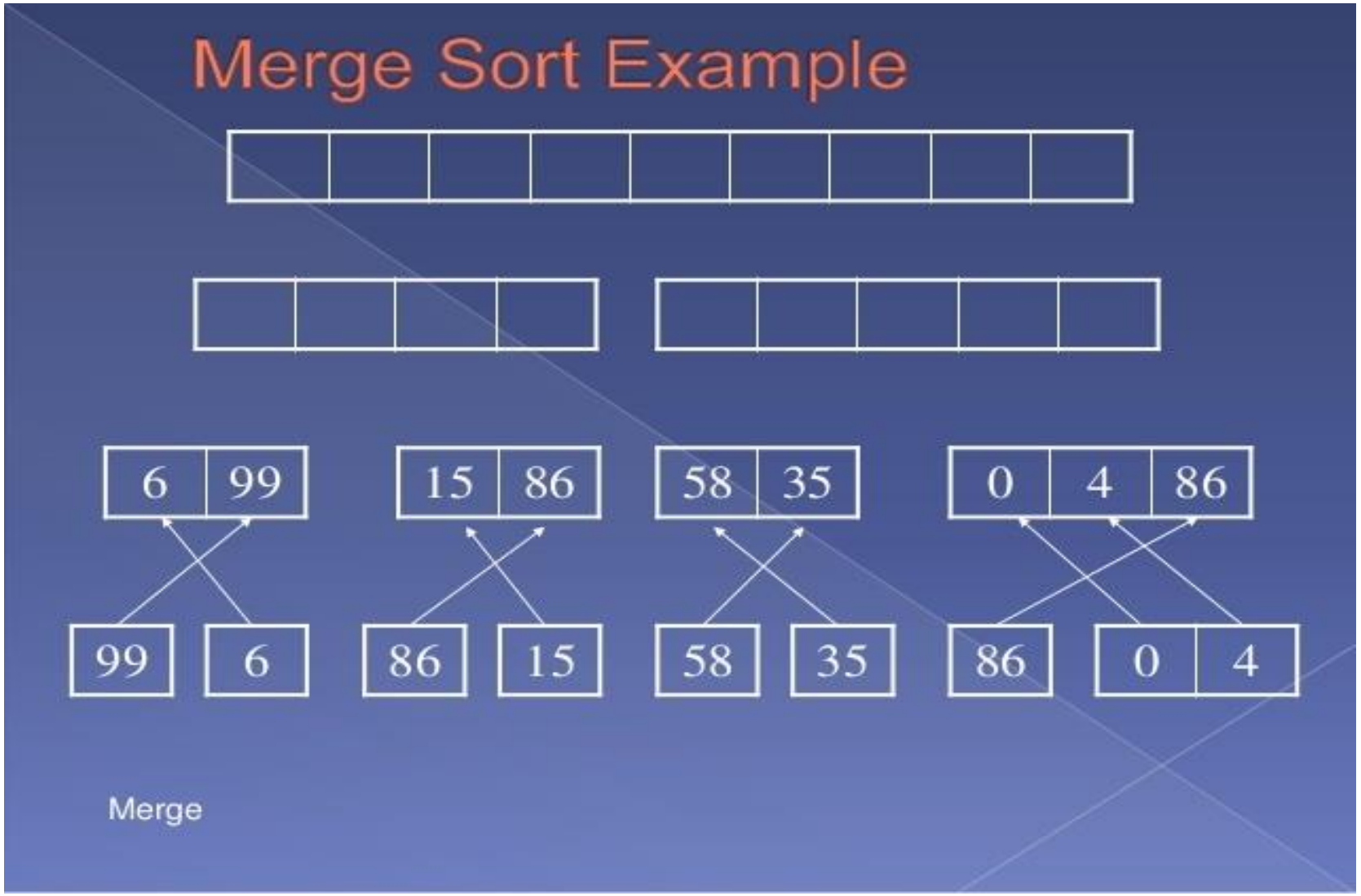
Sorting using Array(CO1)



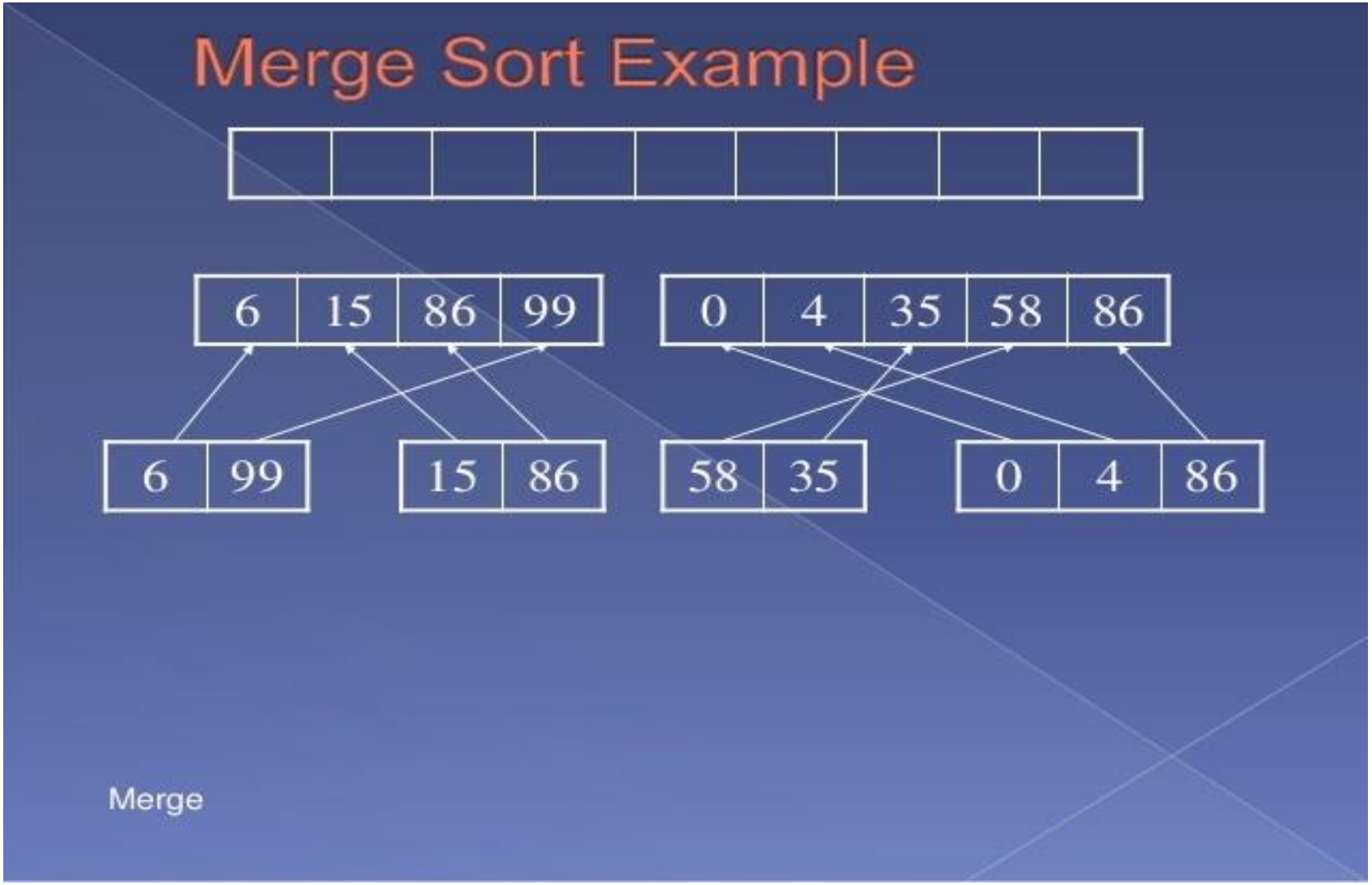
Sorting using Array(CO1)



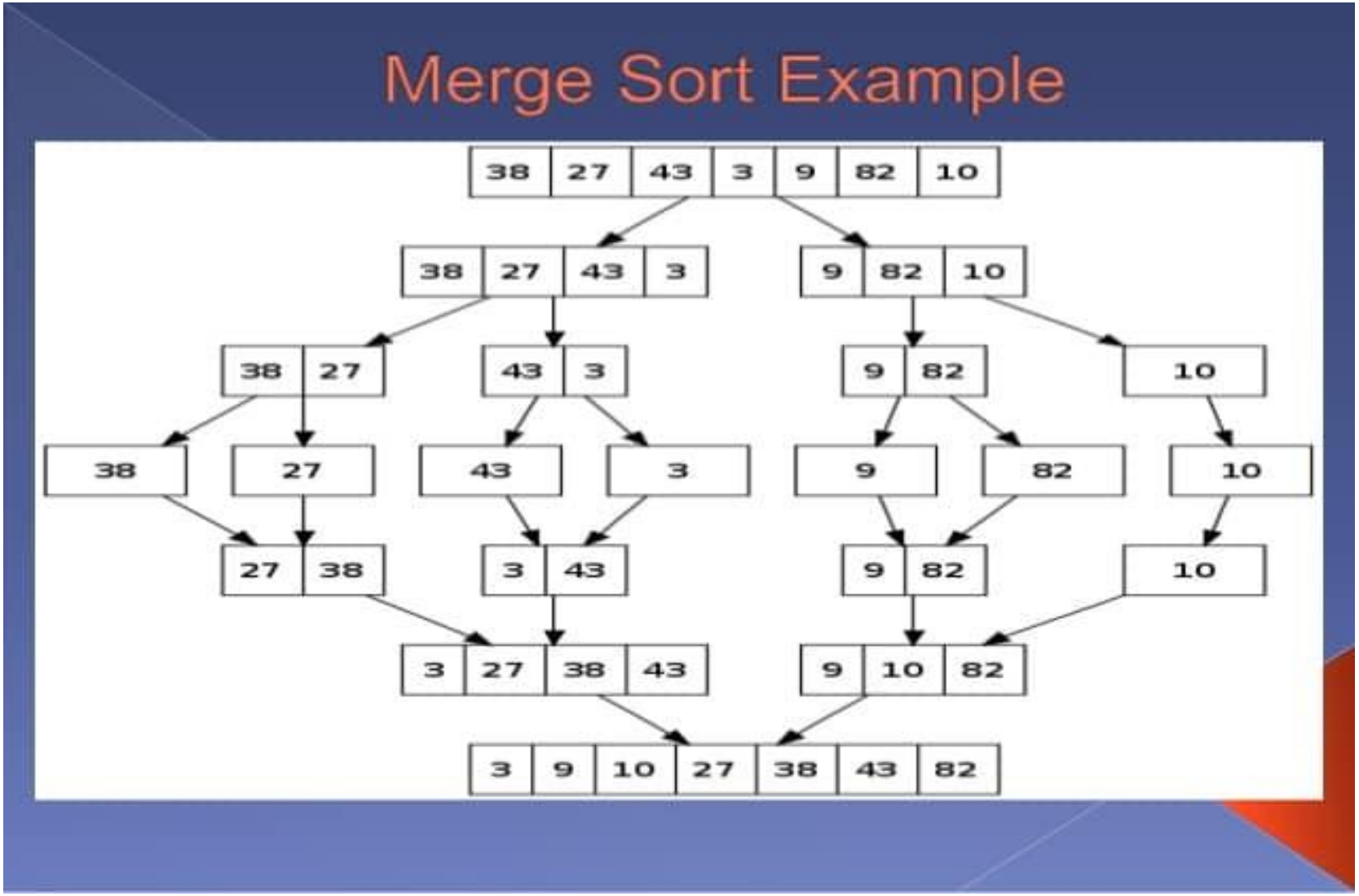
Sorting using Array(CO1)



Sorting using Array(CO1)



Sorting using Array(CO1)



Sorting using Array(CO1)

Merge Sort Algorithm

step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
if left > right
return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
step 4: Stop

Sorting using Array(CO1)

Merge Sort Algorithm

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5       L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7       R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13       if L[i] ≤ R[j]
14            A[k] = L[i]
15            i = i + 1
16       else A[k] = R[j]
17            j = j + 1
```

Sorting using Array(CO1)

QUICK SORT

This sorting algorithm uses the idea of divide and conquer.

It finds the element called **pivot** which divides the array into two halves in such a way that elements in the left half are smaller than pivot and elements in the right half are greater than pivot.

Sorting using Array(CO1)

QUICK SORT

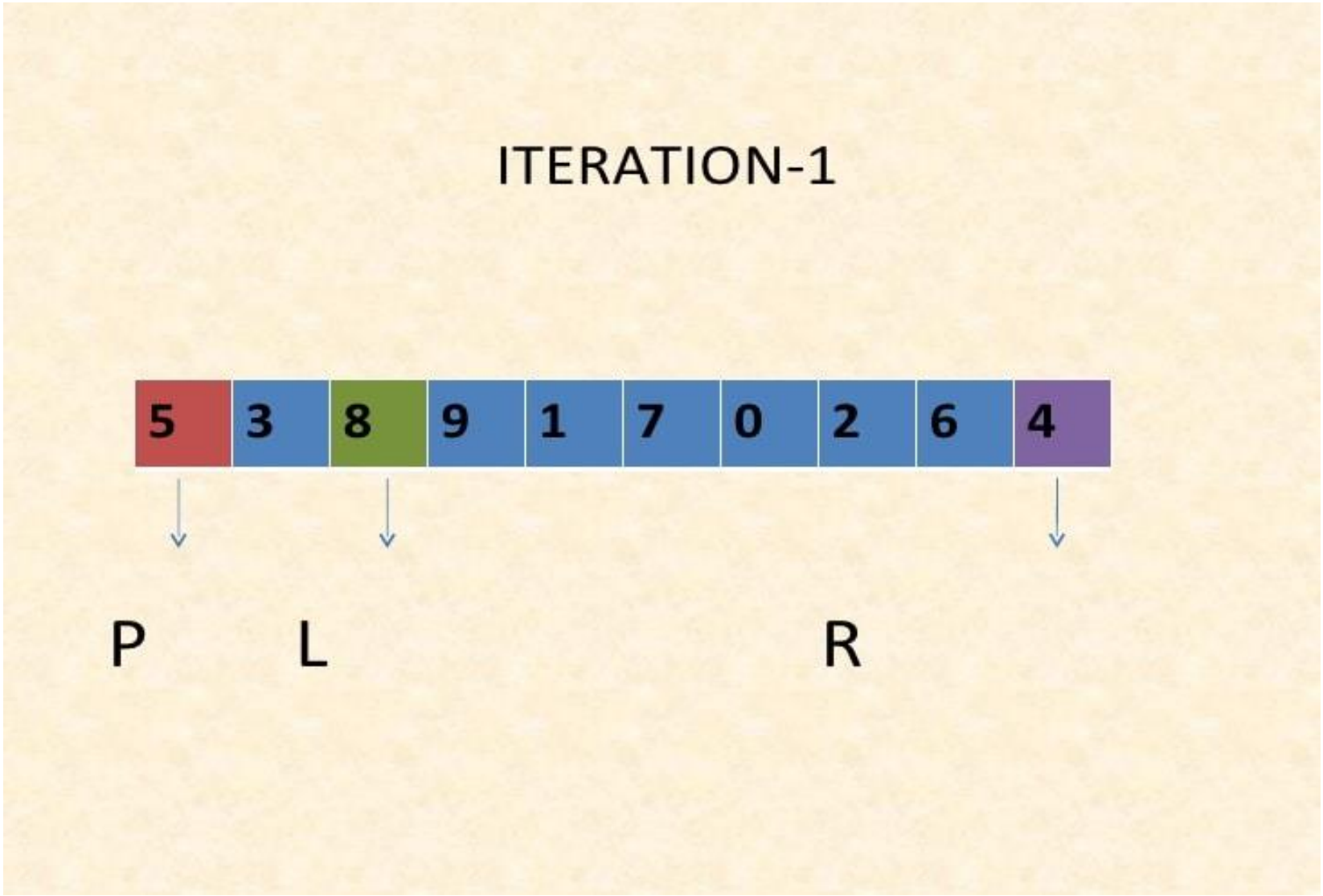
Three steps

- Find pivot that divides the array into two halves.
- Quick sort the left half.
- Quick sort the right half.

Sorting using Array(CO1)

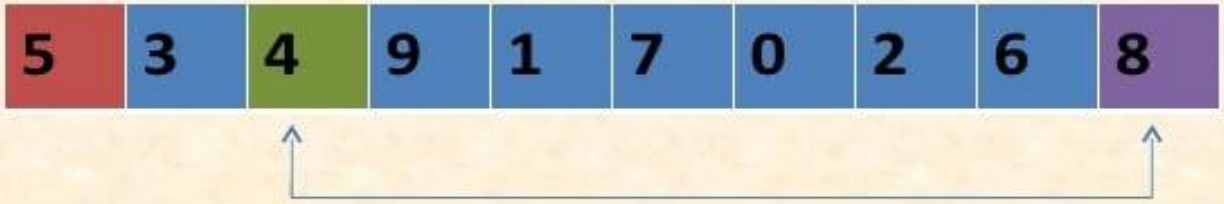
GIVEN SERIES

Sorting using Array(CO1)

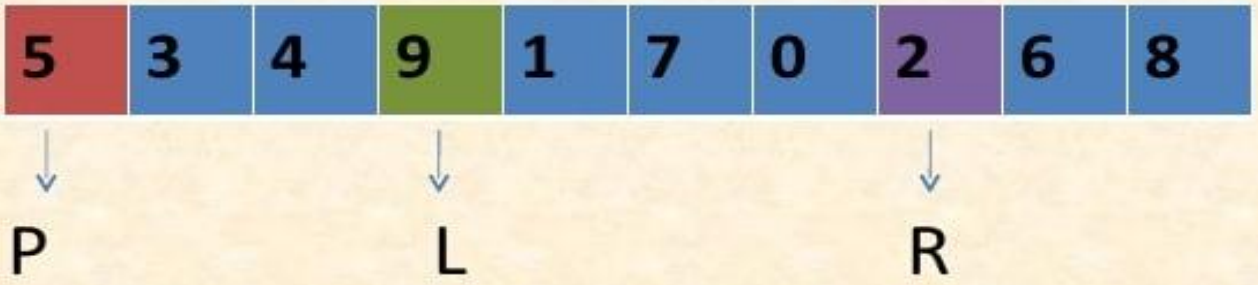


Sorting using Array(CO1)

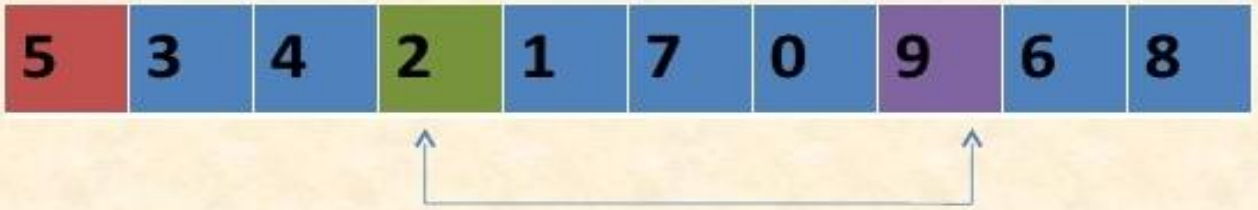
SWAPE THE ELEMENTS



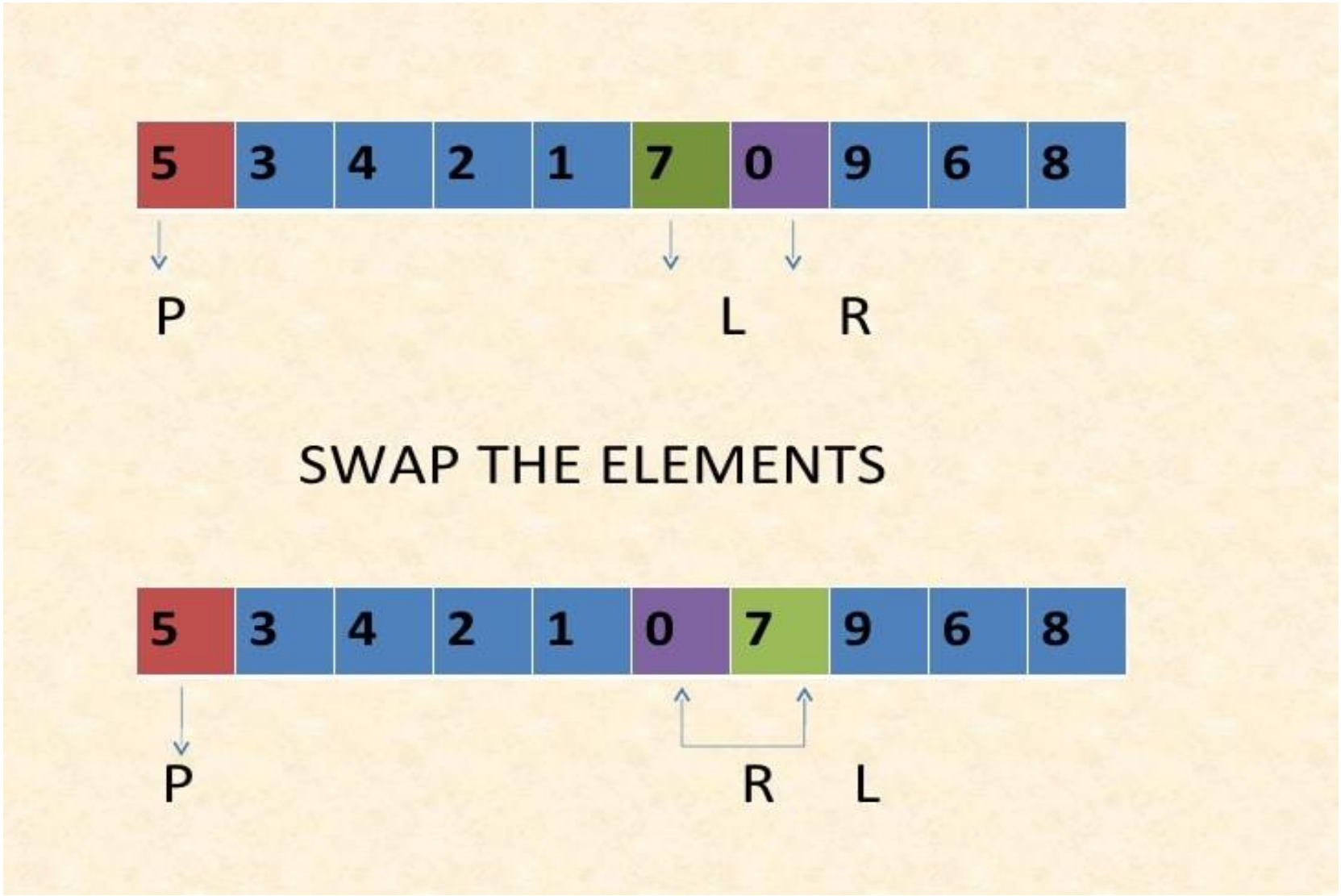
Sorting using Array(CO1)



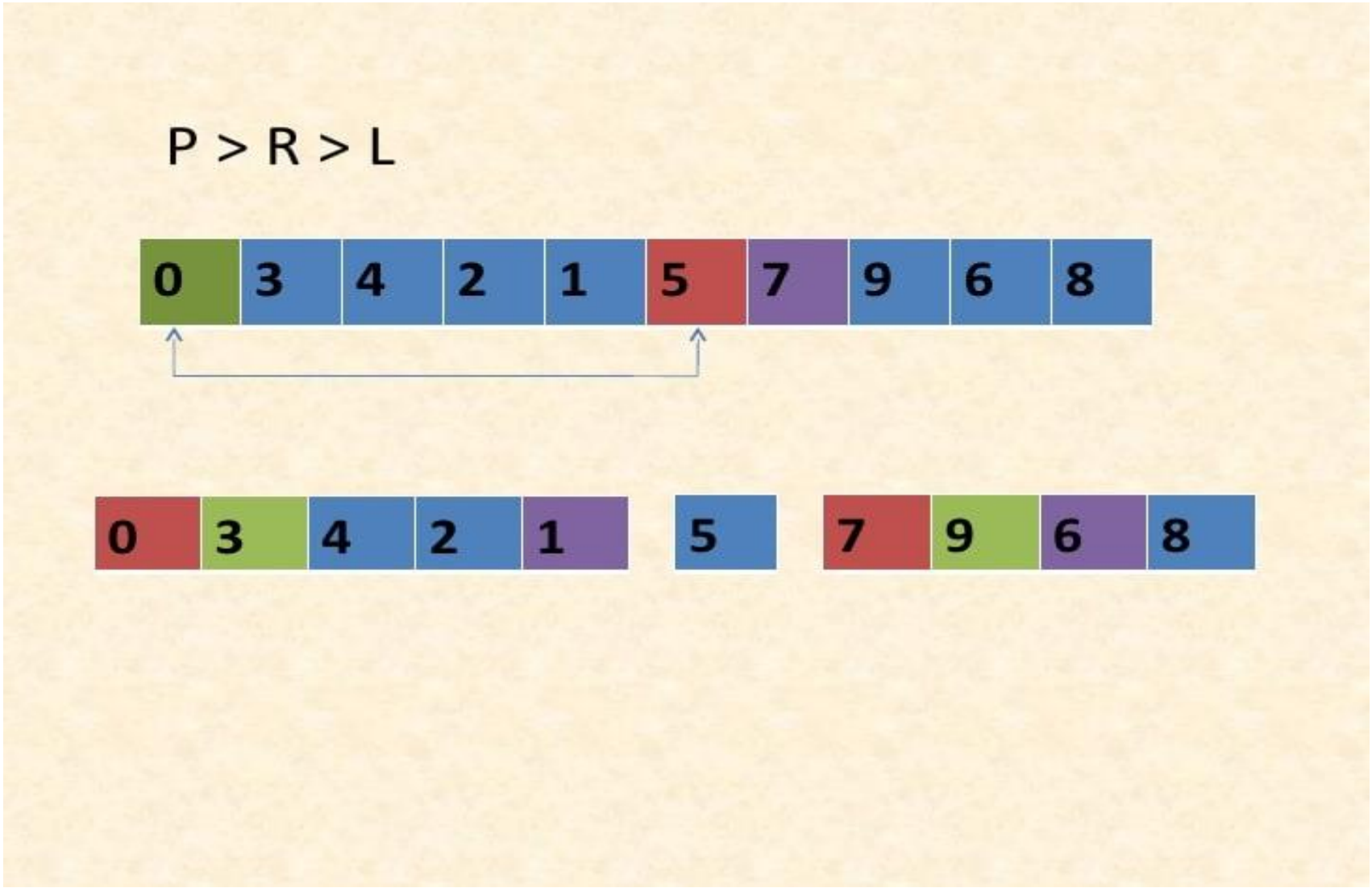
SWAP THE ELEMENTS



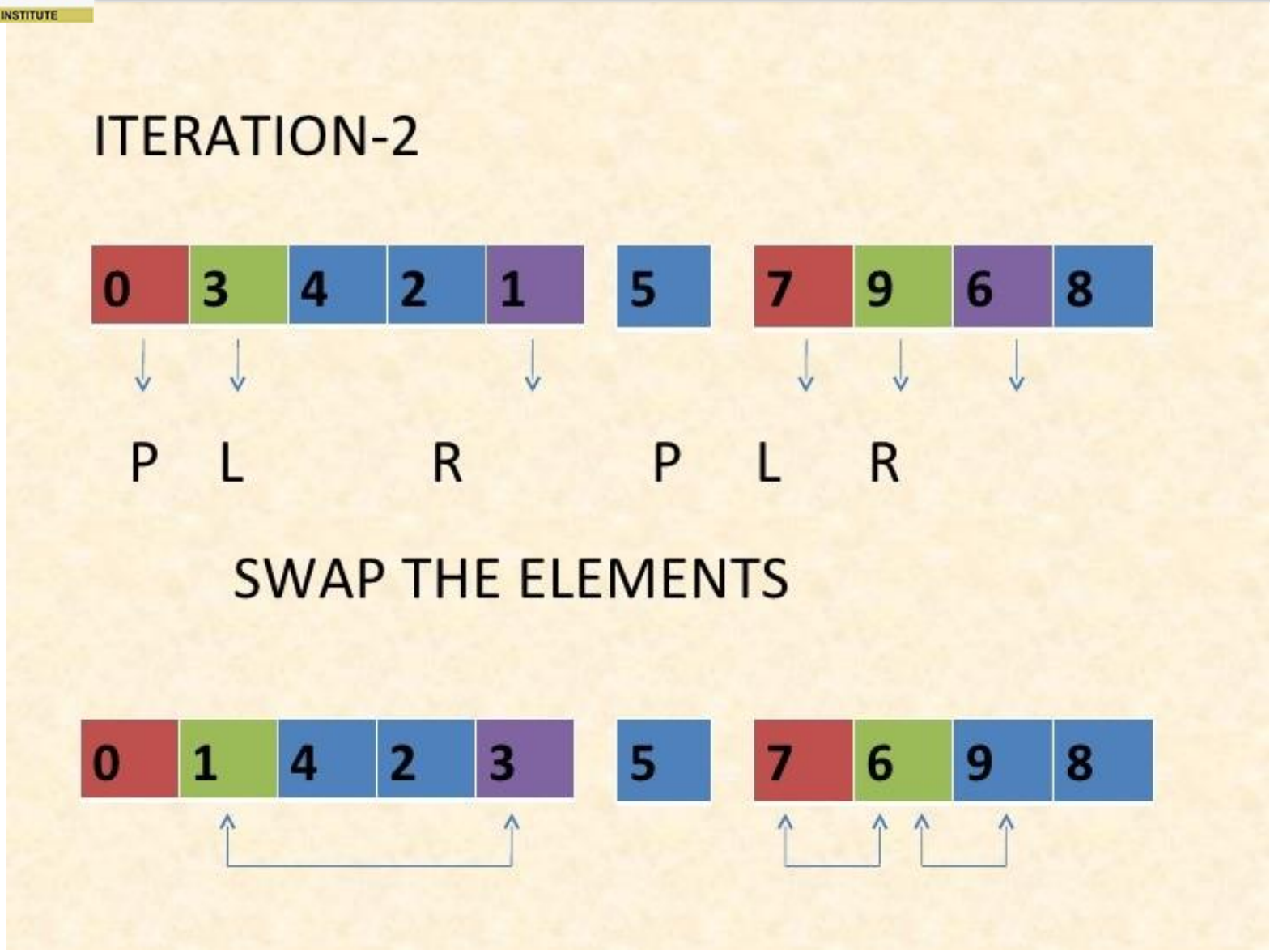
Sorting using Array(CO1)



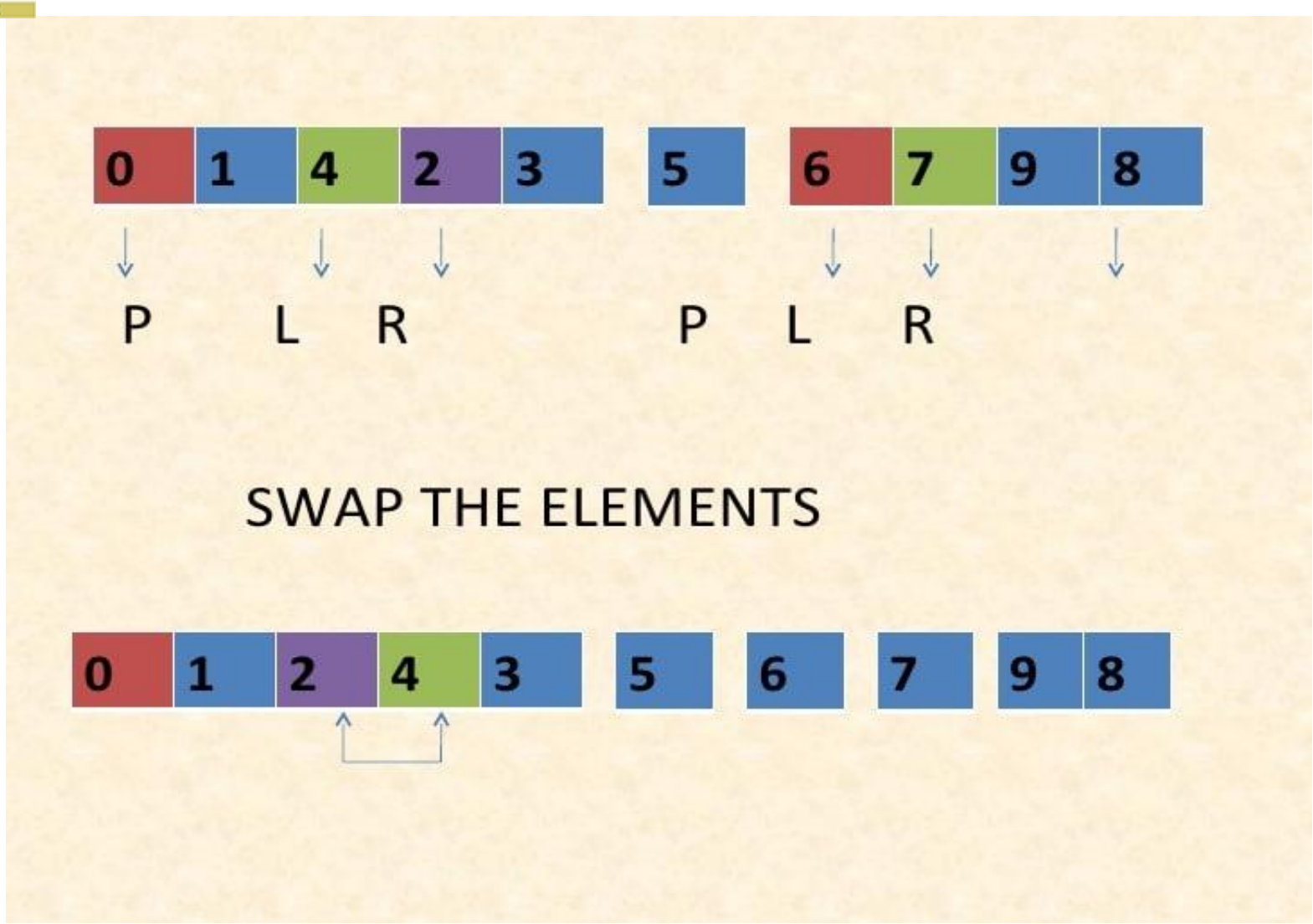
Sorting using Array(CO1)



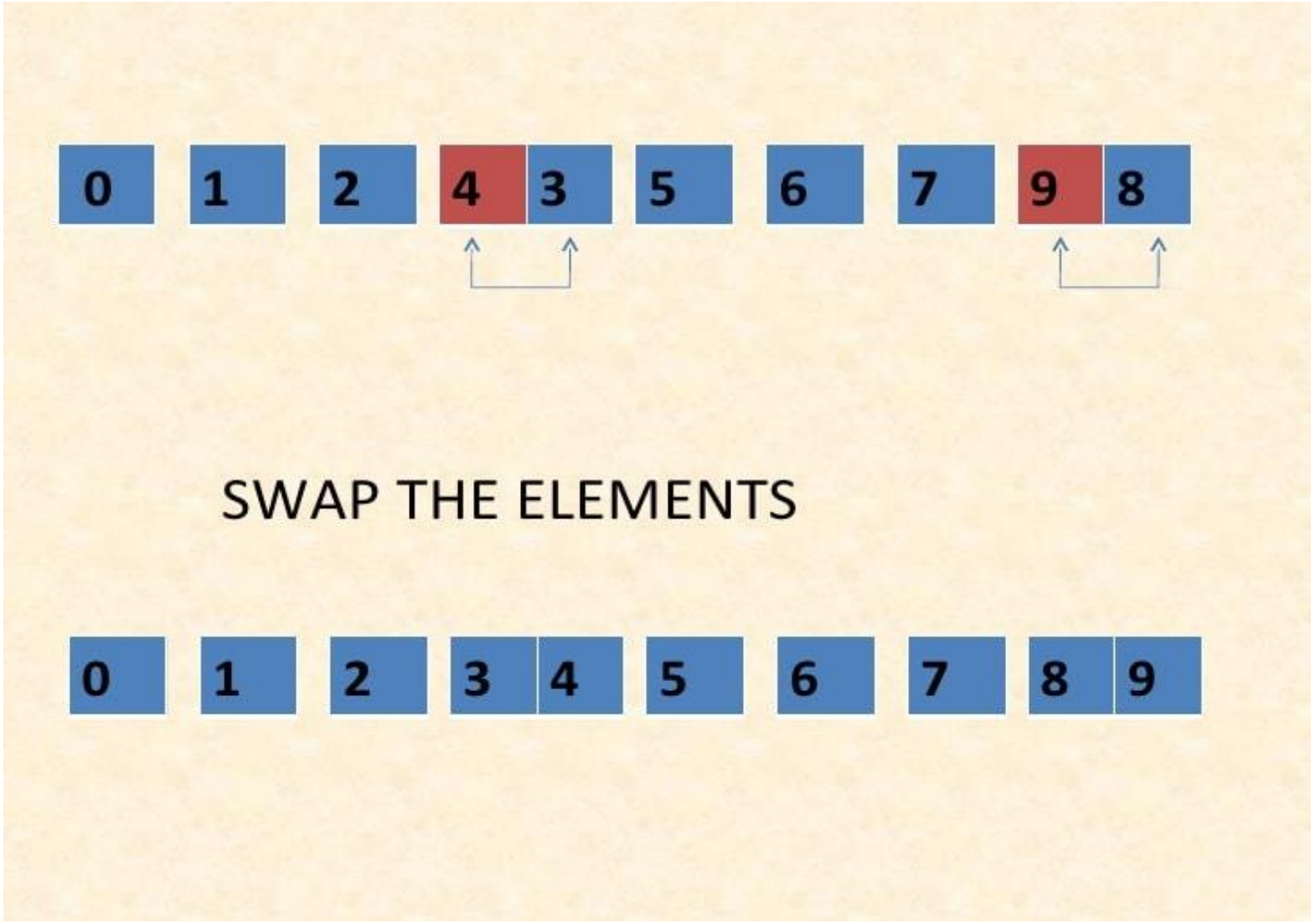
Sorting using Array(CO1)



Sorting using Array(CO1)



Sorting using Array(CO1)



Sorting using Array(CO1)

Algorithm

```
QUICKSORT (array A, lb, ub)
    if (lb < ub)
        p = partition(A, lb, ub)
       QUICKSORT (A, lb, p - 1)
       QUICKSORT (A, p + 1, ub)
```

Partition algo

```
PARTITION (array A, lb, ub)
  pivot = A[lb]
  start = lb
  end = ub
  While (start < end)
      While (A[start] <= pivot)
            start ++
      While (A[end]> pivot)
            end --
 if (start < end)
          swap(A[start], [end])
 else
 swap(A[lb], A[end])
 return end
```

Hashing (CO1)

Hashing

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Some examples of how hashing is used in our lives include:

•In universities, each student is assigned a unique roll number that can be used to retrieve information about them.

•In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

Hashing (CO1)

Hash Function

A hash function maps a key/value to a small integer that can be used as the index in the hash table.

Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Hashing (CO1)

The **Objective** of hashing is to access the key from hash table with O(1) time.

If distinct key values map to distinct key location then that hash function is called **good hash function.** In that case we can access the key in O(1) time

If distinct key values map to same key location then that hash function is called **bad hash function.** In that case we can access the key in O(n) times and collision occurs.

Types of Hash Function

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

Hashing (CO1)

Division Method

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

$h(K) = k \bmod M$

Here,
k is the key value, and
M is the size of the hash table.

It is best suited that M is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Hashing (CO1)

Examples of division Method:

k = 12345
M = 95
h(12345) = 12345 mod 95
= 90
k = 1276
M = 11
h(1276) = 1276 mod 11
= 0

Hashing (CO1)

Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

Square the value of the key k i.e. k2
Extract the middle r digits as the hash value.
Formula:

h(K) = h(k x k)

Here,
k is the key value.

The value of r can be decided based on the size of the table.

Hashing (CO1)

Example of mid square method:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60
k x k = 60 x 60
= 3600
h(60) = 60

The hash value obtained is 60

Hashing (CO1)

Digit Folding Method:

This method involves two steps:

Divide the key-value k into a number of parts i.e. k1, k2, k3,….,kn, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
Add the individual parts. The hash value is obtained by ignoring the last carry if any.
Formula:

k = k1, k2, k3, k4, ….., kn
s = k1+ k2 + k3 + k4 +….+ kn
h(K)= s

Hashing (CO1)

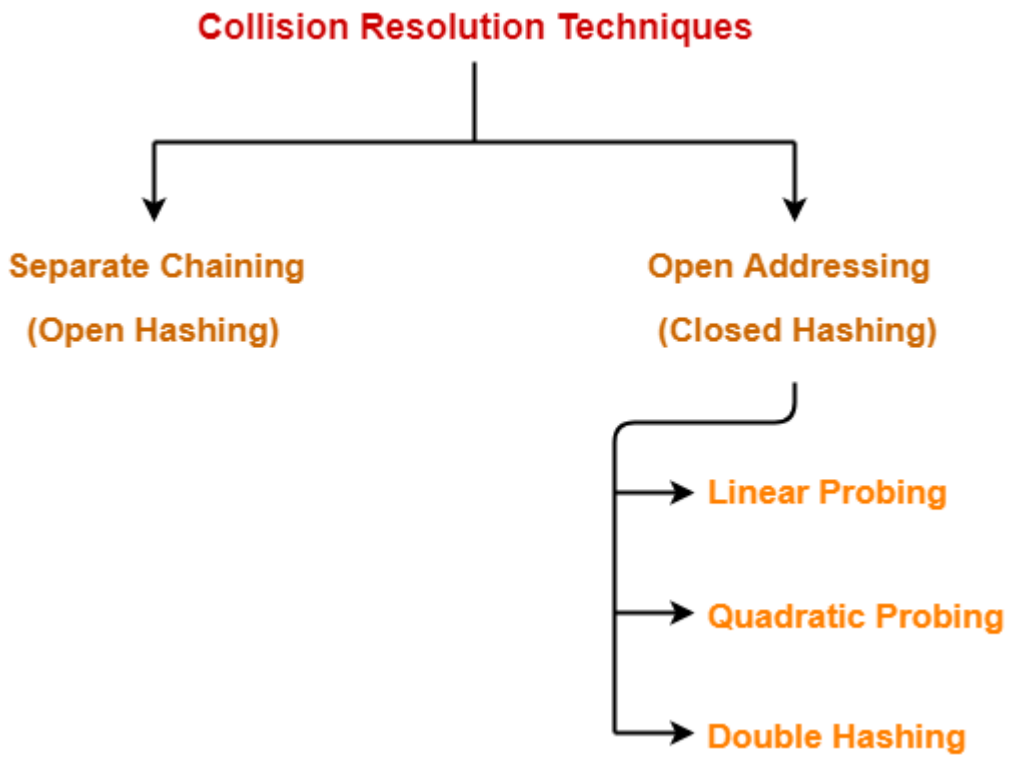
Example of Digit folding method:

k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3
  = 12 + 34 + 5
  = 51
h(K) = 51

Note:
The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

Hashing (CO1)

Collision Handling Techniques in Hashing

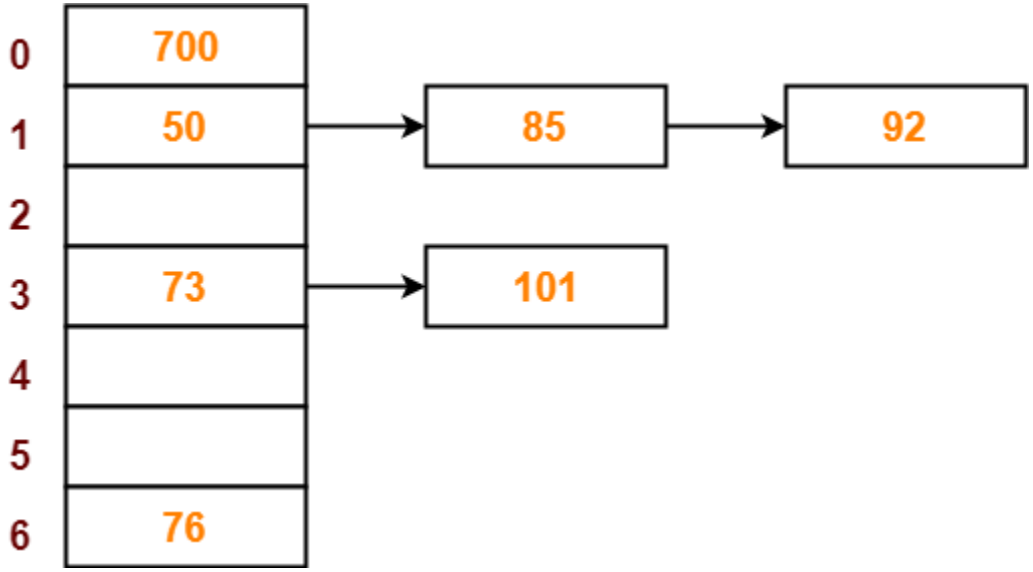


Hashing (CO1)

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

Use **separate chaining technique** for collision resolution.



Hashing (CO1)

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

Use **linear probing technique** for collision resolution.

Collision Resolution function → Hi(x) = (H(x) + f(i))mod m
Where H(x) = x mod 7, f(i) = i, m = size of has table

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Hashing (CO1)

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

Use **quadratic probing technique** for collision resolution.

Collision Resolution function → Hi(x) = (H(x) + f(i))mod m
Where H(x) = x mod 7, f(i) = i*i, m = size of has table

0	700
1	50
2	85
3	73
4	101
5	92
6	76

Hashing (CO1)

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

Use **double hashing technique** for collision resolution.

Collision Resolution function → Hi(x) = (H(x) + f(i))mod m
Where H(x) = x mod 7, m = size of has table, , f(i) = i(p-x mod p), p is the largest prime num less then m

0	700
1	50
2	101
3	92
4	85
5	73
6	76

Daily Quiz

1. Where is linear searching used?
a) When the list has only a few elements
b) When performing a single search in an unordered list
c) Used all the time
d) When the list has only a few elements and When performing a single search in an unordered list

2. What is the time complexity for linear search?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(1)

Daily Quiz

3. What is the best case and worst case complexity of ordered linear search?

a) O(nlogn), O(logn)

b) O(logn), O(nlogn)

c) O(n), O(1)

d) O(1), O(n)

MCQ s

- Which of these best describes an array?
  a) A data structure that shows a hierarchical behaviour
  b) Container of objects of similar types
  c) Arrays are immutable once initialised
  d) Array is not a data structure

- How do you initialize an array in C?
  a) int arr[3] = (1,2,3);
  b) int arr(3) = {1,2,3};
  c) int arr[3] = {1,2,3};
  d) int arr(3) = (1,2,3);

MCQ s

- What are the advantages of arrays?
  a) Objects of mixed data types can be stored
  b) Elements in an array cannot be sorted
  c) Index of first element of an array is 1
  d) Easier to store elements of same data type

- Elements in an array are accessed _____
  a) randomly
  b) sequentially
  c) exponentially
  d) logarithmically

MCQ s

- In linked list each node contain minimum of two fields. One field is data field to store the data second field is?
  - A. Pointer to character
  - B. Pointer to integer
  - **C. Pointer to node**
  - D. Node

- Linked list data structure offers considerable saving in
  - A. Computational Time
  - B. Space Utilization
  - **C. Space Utilization and Computational Time**
  - D. None of the mentioned

Glossary Question

i. Algorithm ii. Abstract Data type iii.Asymptotic Analysis iv. Binary search

Answer the questions.

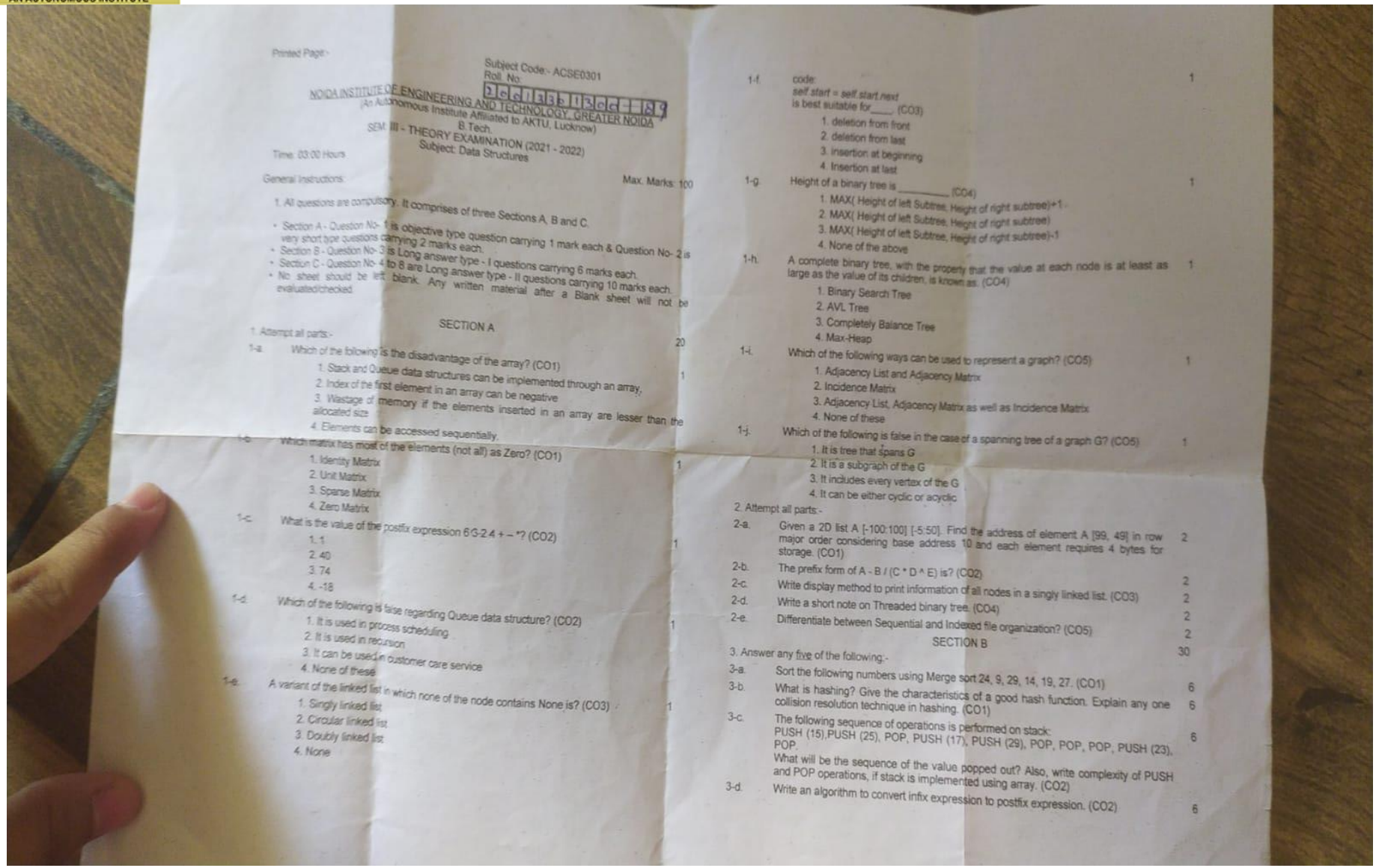
a. A standard recursive algorithm for finding the record with a given search key value within a sorted list.

b. A process followed to solve a problem.

c. The specification of a data type with some language.

d.Computer Program by identifying its growth rate.

Last year Question Paper



Printed Page:-

Subject Code:- ACSE0301
Roll. No.

NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY, GREATER NOIDA
(An Autonomous Institute Affiliated to AKTU, Lucknow)
B.Tech.
SEM: III - THEORY EXAMINATION (2021 - 2022)
Subject: Data Structures

Time: 03:00 Hours

Max. Marks: 100

General Instructions:

1. All questions are compulsory. It comprises of three Sections A, B and C.
- Section A - Question No- 1 is objective type question carrying 1 mark each & Question No- 2 is very short type questions carrying 2 marks each.
- Section B - Question No- 3 is Long answer type - I questions carrying 6 marks each.
- Section C - Question No- 4 to 8 are Long answer type - II questions carrying 10 marks each.
- No sheet should be left blank. Any written material after a Blank sheet will not be evaluated/checked.

SECTION A

1. Attempt all parts:- 20

1-a. Which of the following is the disadvantage of the array? (CO1) 1
1. Stack and Queue data structures can be implemented through an array.
2. Index of the first element in an array can be negative
3. Wastage of memory if the elements inserted in an array are lesser than the allocated size
4. Elements can be accessed sequentially.

1-b. Which matrix has most of the elements (not all) as Zero? (CO1) 1
1. Identity Matrix
2. Unit Matrix
3. Sparse Matrix
4. Zero Matrix

1-c. What is the value of the postfix expression 6 3 2 4 + – *? (CO2) 1
1. 1
2. 40
3. 74
4. -18

1-d. Which of the following is false regarding Queue data structure? (CO2) 1
1. It is used in process scheduling
2. It is used in recursion
3. It can be used in customer care service
4. None of these

1-e. A variant of the linked list in which none of the node contains None is? (CO3) 1
1. Singly linked list
2. Circular linked list
3. Doubly linked list
4. None

1-f. code: 1
self.start = self.start.next
is best suitable for____. (CO3)
1. deletion from front
2. deletion from last
3. insertion at beginning
4. Insertion at last

1-g. Height of a binary tree is _____ (CO4) 1
1. MAX(Height of left Subtree, Height of right subtree)+1
2. MAX(Height of left Subtree, Height of right subtree)
3. MAX(Height of left Subtree, Height of right subtree)-1
4. None of the above

1-h. A complete binary tree, with the property that the value at each node is at least as large as the value of its children, is known as. (CO4) 1
1. Binary Search Tree
2. AVL Tree
3. Completely Balance Tree
4. Max-Heap

1-i. Which of the following ways can be used to represent a graph? (CO5) 1
1. Adjacency List and Adjacency Matrix
2. Incidence Matrix
3. Adjacency List, Adjacency Matrix as well as Incidence Matrix
4. None of these

1-j. Which of the following is false in the case of a spanning tree of a graph G? (CO5) 1
1. It is tree that spans G
2. It is a subgraph of the G
3. It includes every vertex of the G
4. It can be either cyclic or acyclic

2. Attempt all parts:-

2-a. Given a 2D list A [-100:100] [-5:50]. Find the address of element A [99, 49] in row major order considering base address 10 and each element requires 4 bytes for storage. (CO1) 2

2-b. The prefix form of A - B / (C * D ^ E) is? (CO2) 2

2-c. Write display method to print information of all nodes in a singly linked list. (CO3) 2

2-d. Write a short note on Threaded binary tree. (CO4) 2

2-e. Differentiate between Sequential and Indexed file organization? (CO5) 2

SECTION B 30

3. Answer any five of the following:-

3-a. Sort the following numbers using Merge sort 24, 9, 29, 14, 19, 27. (CO1) 6

3-b. What is hashing? Give the characteristics of a good hash function. Explain any one collision resolution technique in hashing. (CO1) 6

3-c. The following sequence of operations is performed on stack: 6
PUSH (15),PUSH (25), POP, PUSH (17), PUSH (29), POP, POP, POP, PUSH (23), POP.
What will be the sequence of the value popped out? Also, write complexity of PUSH and POP operations, if stack is implemented using array. (CO2)

3-d. Write an algorithm to convert infix expression to postfix expression. (CO2) 6

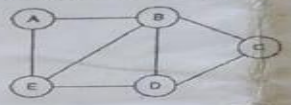
Last year Question Paper

NIET Greater Noida
GET FUTURE READY
Estd.2001
AN AUTONOMOUS INSTITUTE

3-e. Write a function in Python to reverse a singly linked list. (CO3) 6

3-f. Can you find a unique tree when any two traversals are given? Using the following traversal construct the corresponding binary tree: 6
INORDER: H K D B I L E A F C M J G
PREORDER: A B D H K E I L C F G J M
Also find the Post Order traversal of obtained tree. (CO4)

3-g. Give (i) DFS and (ii) BFS traversal of the following graph. (CO5) 6



SECTION C 50

4. Answer any one of the following:-

4-a. Write a program to implement Quick sort. Trace the working of the algorithm on the following input: 44, 14, 6, 34, 51, -7, 95, 72, 48. (CO1) 10

4-b. Write a program in Python for multiplication of two matrices. Order of the matrices must be entered by the user at run time. (CO1) 10

5. Answer any one of the following:-

5-a. What is recursion? Write a recursive function in Python to solve Tower of Hanoi problem. Also, remove tail recursion, if any, from the created function. (CO2) 10

5-b. Convert the following infix expression into its equivalent (i) prefix and (ii) postfix expression using stack implementation: 10
(a + b) / d ^ ((e - f) + g) (CO2)

6. Answer any one of the following:-

6-a. How can we represent a polynomial using a linked list? Write a function in Python to add two polynomials represented by linked list. (CO3) 10

6-b. Write functions in Python to insert a node (i) at beginning, (ii) at the end in a doubly linked list. Illustrate with an example. (CO3) 10

7. Answer any one of the following:-

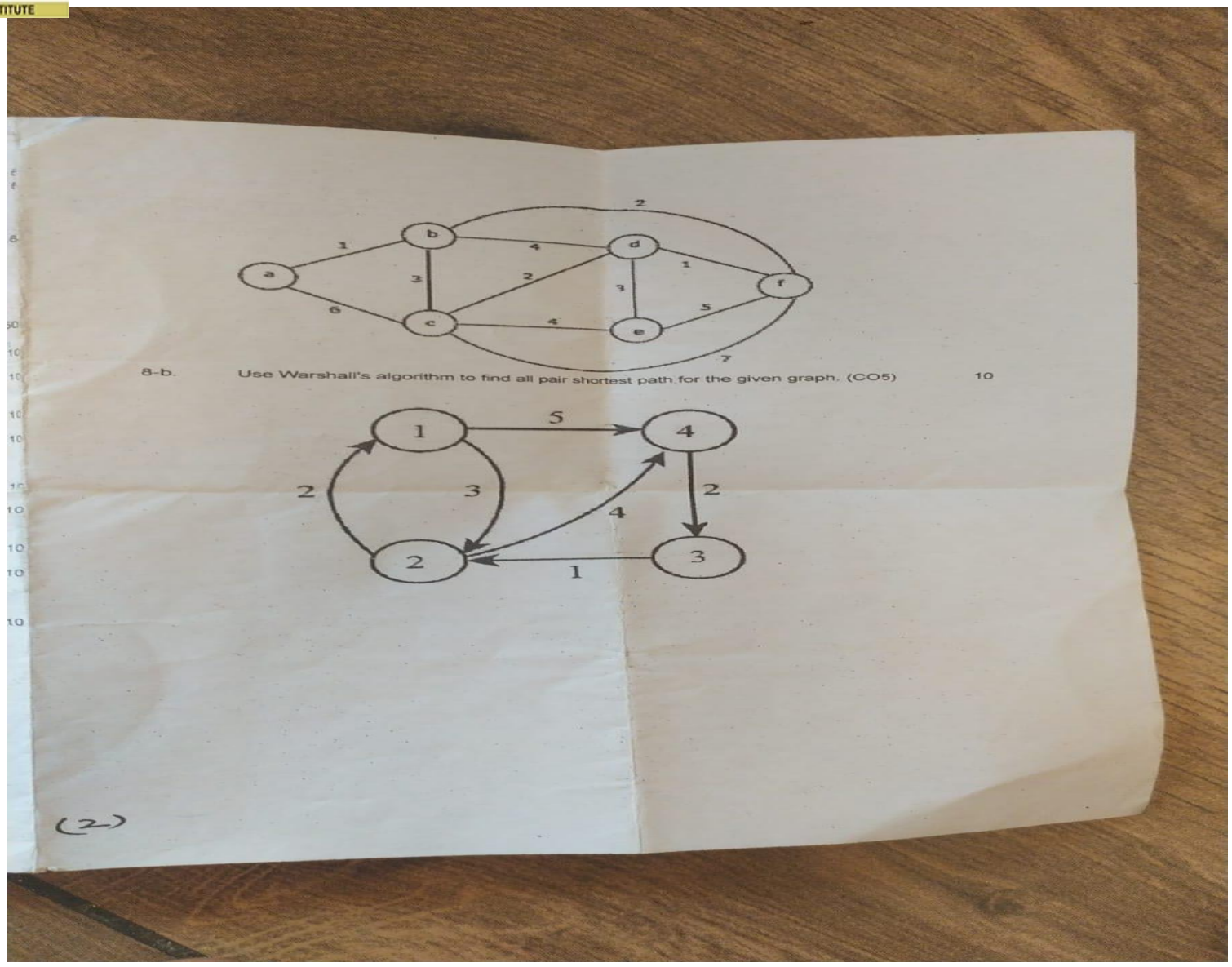
7-a. What is AVL tree. Explain the term balance factor in AVL tree? Describe various rotations performed on AVL tree with the help of neat diagram. (CO4) 10

7-b. Write the characteristics of a B-Tree of order m. Create B-Tree of order 5 from the following lists of data items : 10
20, 30, 35, 85, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 80, 45. (CO4)

8. Answer any one of the following:-

8-a. What is Spanning Tree ? Describe Kruskal and Prim's algorithm to find the minimum cost spanning tree. Determine the minimum cost spanning tree for the graph given below using (i) Kruskal and (ii) Prim's algorithm: (CO5) 10

Last year Question Paper



Old Question Papers

- http://www.aktuonline.com/papers/btech-cs-3-sem-data-structures-kcs301-2020.html
- http://www.aktuonline.com/papers/btech-cs-3-sem-data-structures-rcs-305-2018-19.html
- http://www.aktuonline.com/papers/btech-cs-3-sem-data-structures-rcs-305-2017-18.html
- http://www.aktuonline.com/papers/btech-cs-3-sem-data-structures-using-c-ncs-301-2016-17.html
- https://www.aktuonline.com/papers/btech-cs-3-sem-data-structures-kcs301-2020.html
- https://firstranker.com/fr/frdA290120A1345373/download-aktu-b-tech-3rd-sem-2018-2019-data-structures-rcs-305-question-paper

Expected Questions for University Exam

- What is doubly linked list? What are its applications? Explain how an element can be deleted from doubly linked list using python program.
- What are the merits and demerits of array? Given two arrays of integers in ascending order, develop an algorithm to merge these arrays to form a third array sorted in ascending order.
- How can you represent a sparse matrix in memory?
- List the various operations on linked list.
- What do you understand by time and space complexity?
- Define the various asymptotic notations. Derive the O-notation for linear search.
- Write a program in c to delete a specific element in single linked list. Double linked list takes more space than single linked list for storing one extra address. Under what condition, could a double linked list more beneficial than single linked list.

Summary

- Knowledge of basic data structures is important for understanding organization of data.
- Understanding of array and link list is compulsory to implement other data structures.

References

- Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, “Data Structures Using C and C++”, PHI Learning Private Limited, Delhi India
- Horowitz and Sahani, “Fundamentals of Data Structures”, Galgotia Publications Pvt Ltd Delhi India.
- Lipschutz, “Data Structures” Schaum’s Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.
- Thareja, “Data Structure Using C” Oxford Higher Education.
- AK Sharma, “Data Structure Using C”, Pearson Education India.
- Michael T. Goodrich, Roberto Tamassia, David M. Mount “Data Structures and Algorithms in C++”, Wiley India.
- . P. S. Deshpandey, “C and Data structure”, Wiley Dreamtech Publication.

End

Thank You