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

Unit: 1

DATA STRUCTURES

Course Details
(B Tech AI - 3th Sem)

Dr Kumud Saxena

Associate Professor

Computer Science and Engineering(AI)

Course Outcome

- **CO 1** Describe the need of data structure and algorithms in problem solving and analyze Time space trade-off.

- **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.

- **CO 3** Design, implement and evaluate the real-world applications using stacks, queues and non-linear data structures.

- **CO 4** Compare and contrast the advantages and disadvantages of linked lists over arrays and implement operations on different types of linked list.

- **CO 5** Identify and develop the alternative implementations of data structures with respect to its performance to solve a real-world problem.

Syllabus

UNIT-1

- **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.

- **Hashing:** The symbol table, Hashing Functions, Collision-Resolution Techniques.

UNIT-2

Linked List Self-referential structure, Singly Linked List, Doubly Linked List, Circular Linked List.

Operations on a Linked List: Insertion, Deletion, Traversal, Reversal, Searching Polynomial Representation and Addition Subtraction & Multiplications of Polynomials.

UNIT-3

Stacks: Primitive Stack operations: Push & Pop, Array and Linked Implementation of Stack, Application of Stack: Infix, Prefix, Postfix Expressions and their Mutual Conversion, Evaluation of Postfix Expression.

Recursion: Principles of Recursion, Tail Recursion, Removal of Recursion, Problem Solving using Iteration and Recursion with examples such as Binary Search, Fibonacci Series, and Tower of Hanoi, Trade-offs between Iteration and Recursion.

Queues: Array and linked Implementation of Queues, Operations on Queue: Create, Insert, Delete, Full and Empty, Circular Queues, Dequeue and Priority Queue.

Syllabus

UNIT-4

- **Tree Basic terminology** used with Tree, Binary Trees, Binary Tree Representation: Array Representation and Pointer (Linked List) Representation, Binary Search Tree, Strictly Binary Tree, Complete Binary Tree, An Extended Binary Trees.

- **Tree Traversal algorithms**: In-order, Pre-order and Post-order, Constructing Binary Tree from given Tree Traversal, Operation of Insertion, Deletion, Searching & Modification of Data in Binary Search Tree, Binary Heaps, Heap Sort, Threaded Binary Trees, Traversing Threaded Binary trees, Huffman Coding using Binary Tree, Concept & Basic Operations for AVL Tree, B-Tree & Binary Heaps, Heap Sort.

Syllabus

UNIT-5

- **Graphs:** Terminology used with Graph, Data Structure for Graph Representations: Adjacency matrices, Adjacency List.

- **Graph Traversal**: Depth First Search and Breadth First Search. Connected Component, Spanning Trees, Minimum Cost Spanning Trees: Prim' s and Kruskal's algorithm. Transitive Closure and Shortest Path algorithms: Dijkstra Algorithm.

- **File Structure**: Concepts of files, records and files, Sequential, Indexed and Random File Organization, Indexing structure for index files, hashing for direct files, Multi-Key file organization and Access Methods.

Contents

- Primitive and non-primitive data types

- Types of Data Structures- Linear & Non-Linear Data Structures

- Asymptotic notations (Big Oh, Big Theta and Big Omega),

- Abstract Data Types (ADT).

- Arrays

- Searching and Sorting

- Hashing

Objective of the course is to make students able to:

1. Learn the basic types for data structure, implementation and application.

2. Know the strength and weakness of different data structures.

3. Use the appropriate data structure in context of solution of given problem.

4. Develop programming skills which require to solve given problem.

CO1:

Describe the need of data structure and algorithms in problem solving and analyze Time space trade-off.

Topic mapping with Course Outcome

Topic	CO1	CO2	CO3	CO4	CO5
Basic Data Structure	3	-	-	-	-
Array	2	-	-	-	-

1.Data types

2.Variables

3.Operators

4.Input/ output

5.Conditional and Control statements

6.Arrays

7.Strings

Basic Terminologies

- Data: are simply a value or a set of values of different type which is called data types like string, integer, char etc.

- Structure: Way of organizing information, so that it is easier to use.

- In simple words we can define **DATA STRUCTURES** as its a way organizing data in such a way so that data can be easier to use.
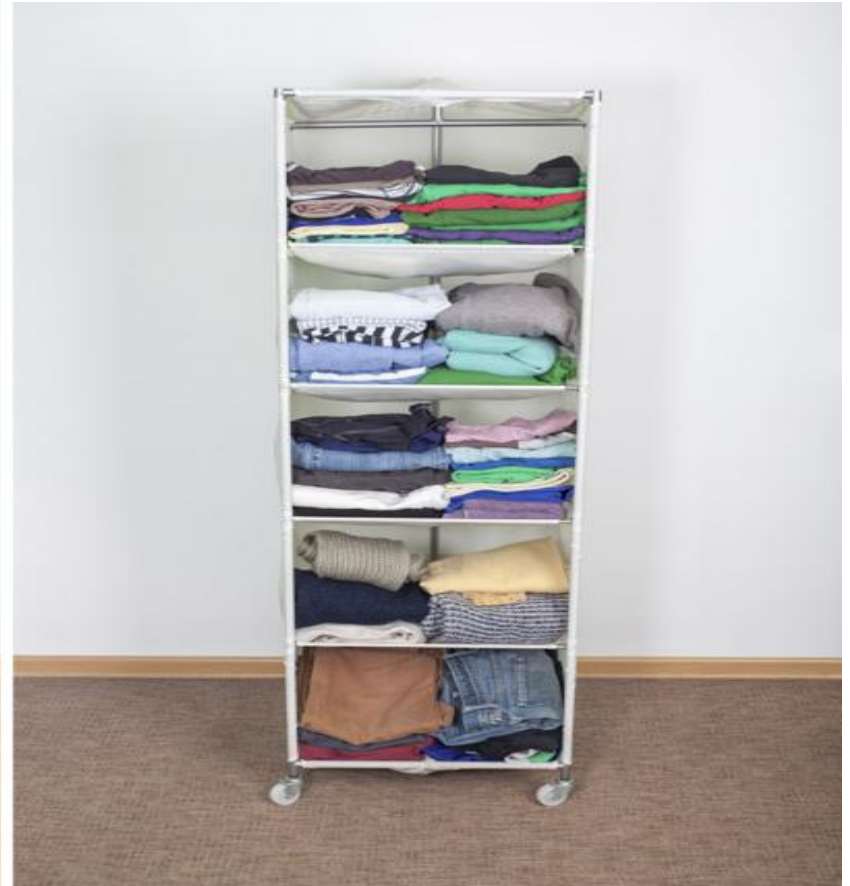
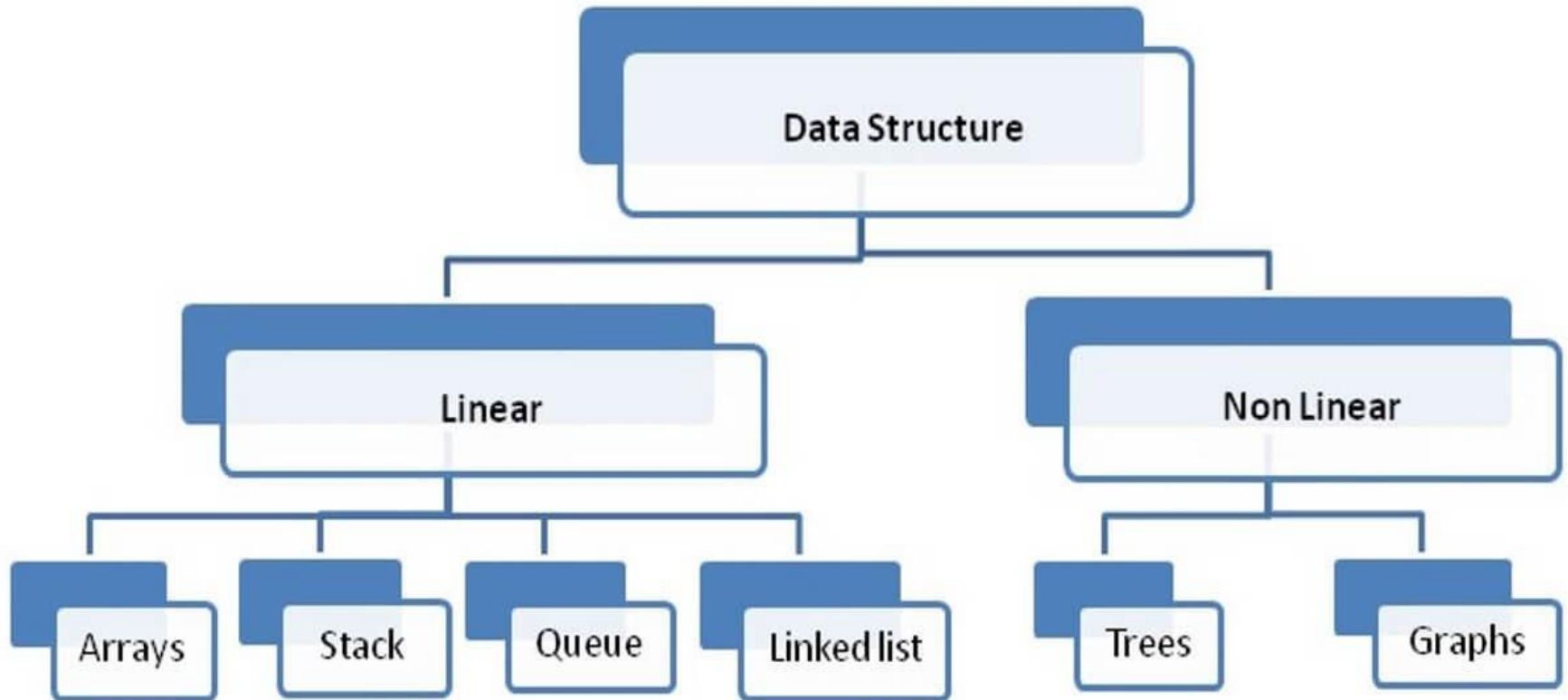**Data Type:-**
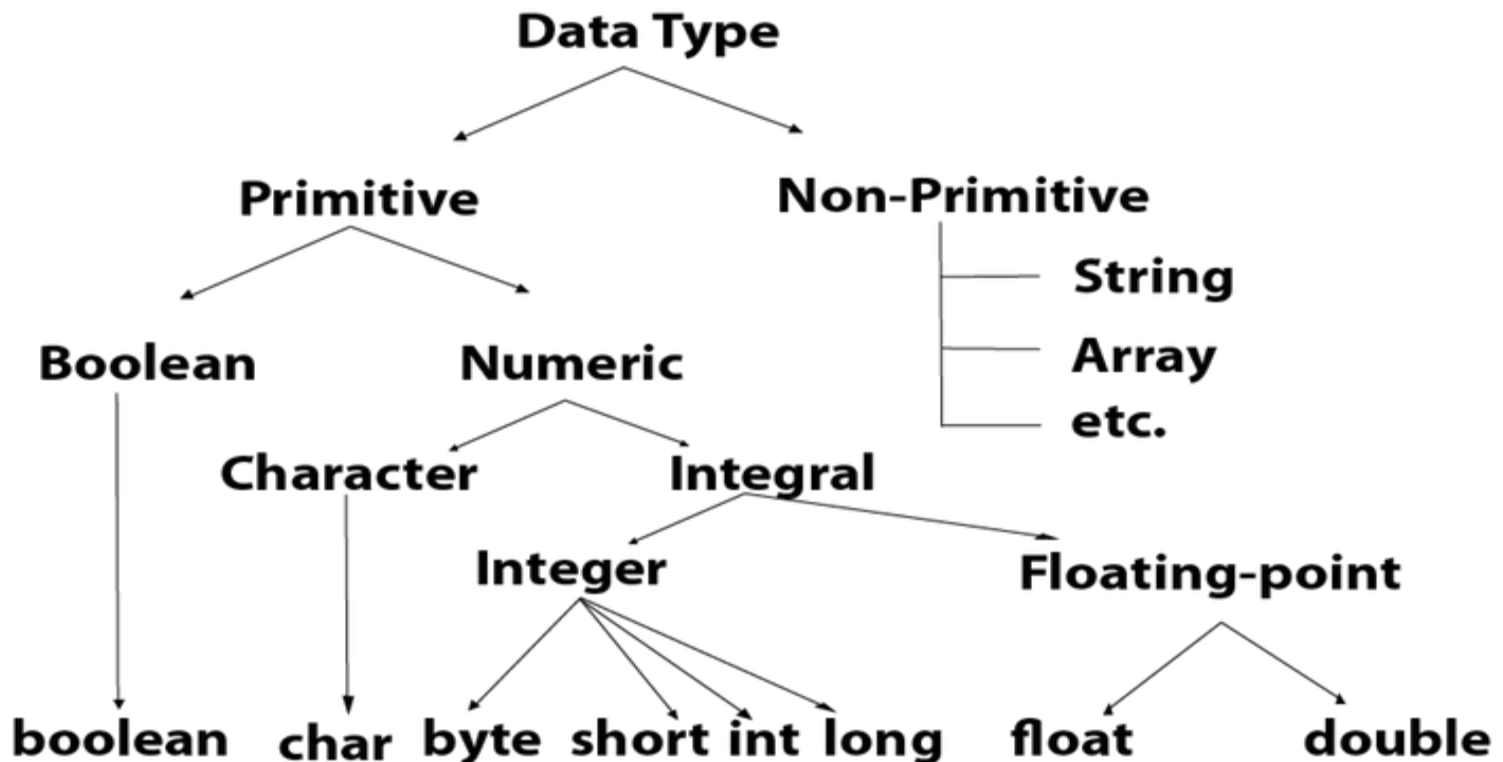
Data Structure

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
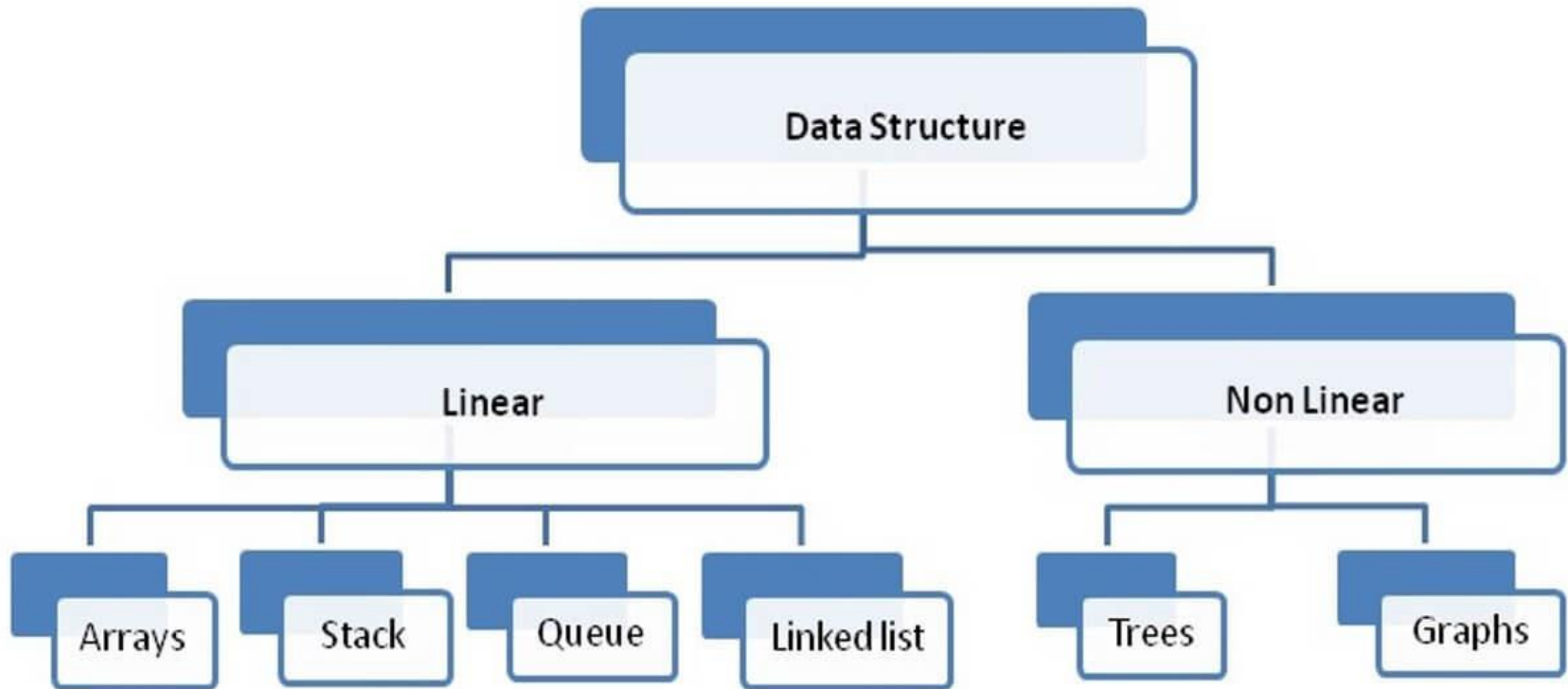
Why Data Structure

Types of Data Structure

Data Structure is classified into two categories :

Linear Data Structure

Non Linear Data Structure

Classification of Data Structure ...

- **Linear Data Structure:**

  Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a **linear data structure.** Examples are:- array, stack, queue, linked list, etc.

- **Non-linear Data Structure:**

  Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**.

Difference between Linear and Non-linear Data Structures:

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next item.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data can not be traversed in a single run.
Examples: Array, Stack, Queue, Linked List.	Examples: Tree, Graph.
Implementation is Easy.	Implementation is Difficult.

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.

- **A stack** is a LIFO (Last In First Out) data structure where element that added last will be deleted first. All operations on stack are performed from on end called TOP. A TOP is a pointer which points to the top element- the element which entered last in the stack.

- **A queue** is a FIFO (First In First Out) data structure where element that added first will be deleted first. In queue, insertion is performed from one end called REAR and deletion is performed from another end called FRONT.

- **A linked list** is a collection of nodes, where each node is made up of a data element and a reference to the next node in the sequence.

Types of NON Linear Data Structure

- **A tree** is collection of nodes where these nodes are arranged hierarchically and form a parent child relationships. A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non empty sets each of which is a sub tree of the root.

- **A Graph** is a collection of a finite number of vertices and an edges that connect these vertices. Edges represent relationships among vertices that stores data elements.

- 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.
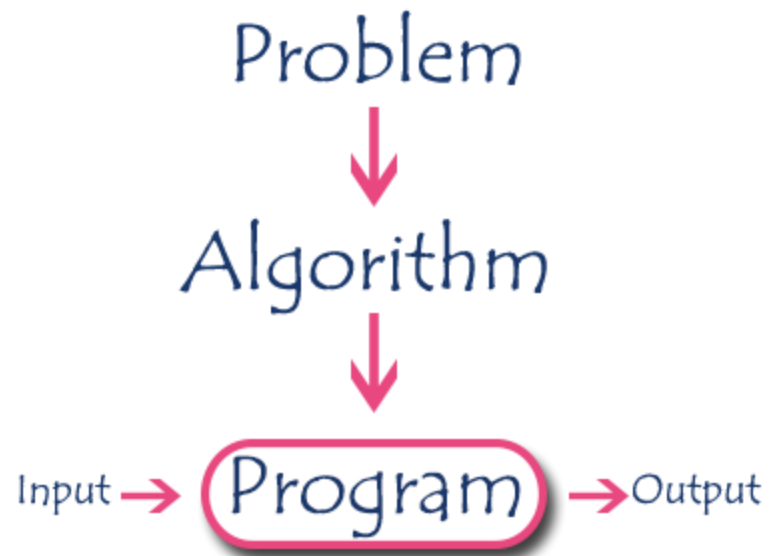
Performance analysis of an algorithm

- Space required to complete the task of that algorithm (**Space Complexity**).

- Time required to complete the task of that algorithm (**Time Complexity**).

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

Dr Kumud Saxena   DS Unit 1

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
sum(a, b)
c = a+b;
return c;

Space Complexity –

In the above piece of code, it requires memory to store variable **'a'**, 'b' and 'c'

That means, totally it requires 3 memory location to complete its execution. And this memory is dependent on the type of data stored in variable.

If any program requires a fixed amount of space then it is called Constant Space complexity

Time Complexity –

It requires c unit of time to add 1 bit so for n bits integer c*n . That means, totally it takes c*n 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

```
int sum(int A[ ], int n)
{
   int sum = 0, i;
   for(i = 0; i < n; i++)
      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.

int sumOfList(int A[ ], int n)	Repeatation No. of Times Executed	Total Total Time required in worst case
{		
int sum = 0, i;	1	1
for(i = 0; i < n; i++)	1 + (n+1) + n	2n + 2
sum = sum + A[i];	n	2n
return sum;	1	1
}		
		4n + 4 Total Time required

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.

The asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers. The following are the types of asymptotic notation

- Big Theta Notation
- Big Oh Notation
- Big Omega Notation
- Small Oh Notation
- Small Omega Notation

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

The function f(n) = Θ(g(n)) (read as " f of n is theta of g of n") if there exist positive constants c1 and c2 and n0 such that

c1g(n)≤f(n)≤c2g(n)   for all n≥ n0
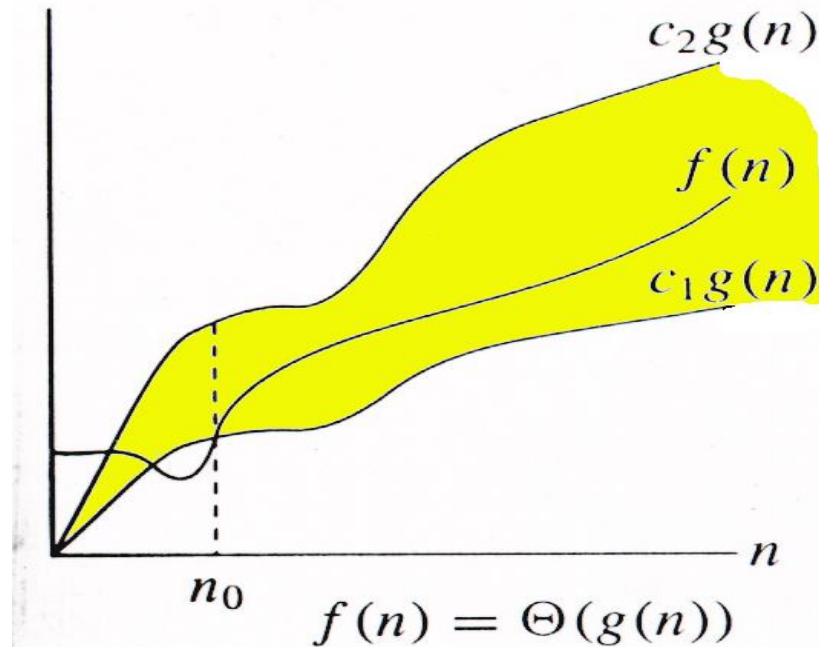
Example

Let us consider a given function,

$$f(n)=4.n^3+10.n^2+5.n+1$$

Considering $g(n)=n^3$,

$4.g(n) \leqslant f(n) \leqslant 5.g(n)$ for all the large values of **n.**

Hence, the complexity of ***f(n)*** can be represented as
$\theta(g(n))$, i.e. $\theta(n^3)$

The function f(n) = O(g(n)) (read as " f of n is big oh of g of n") if there exist positive constants c and n0 such that

$$f(n) \leq cg(n) \text{ for all } n \geq n0$$
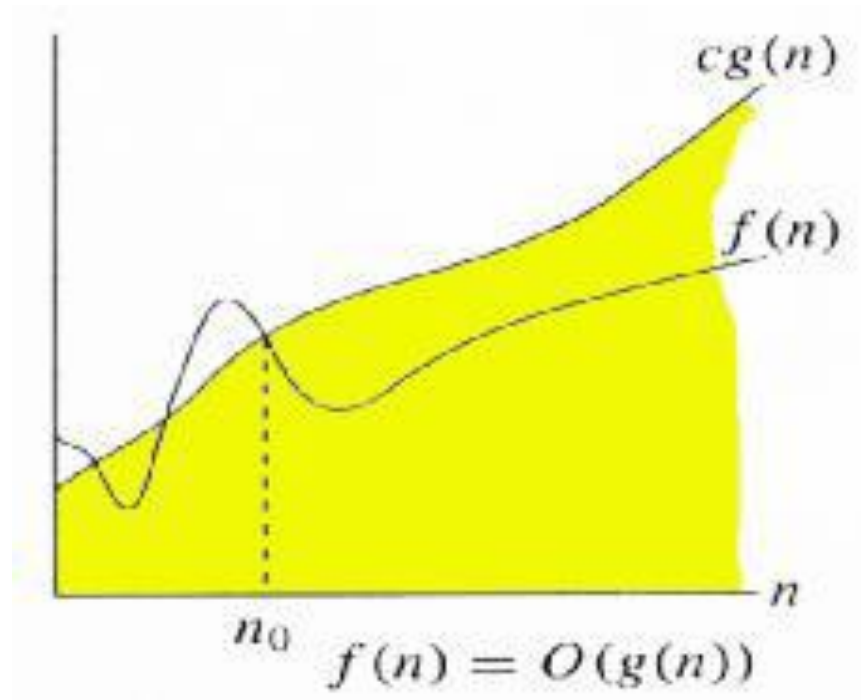
$$f(n) = O(g(n))$$

Example

Let us consider a given function,

$$f(n)=4.n^3+10.n^2+5.n+1$$

Considering $g(n)=n^3$,

$$f(n) \leqslant 5.g(n) \text{ for all the values of } n>2.$$

Hence, the complexity of *f(n)* can be represented as
O(g(n)), i.e. O(n^3)

Q1 f(n) = 3n + 4, g(n) = n,   Prove f(n) = O(g(n))

Sol.

To prove f(n) = O(g(n))   i.e. f(n) <= c.g(n)

It means 3n + 4 <=c.n

Let c= 5

n	f(n)		c. g(n)
	3n+4		5n
1	7	>	5
2	10	=	10
3	13	<	15
4	16	<	20

So, for $n_0 = 2$
$f(n) < = 5.g(n)$

Hence $f(n) = O(g(n))$

The function f(n) = Ω(g(n)) (read as " f of n is Ω of g of n") if there exist positive constants c and n0 such that

$$cg(n) \leq f(n) \text{ for all } n \geq n0$$

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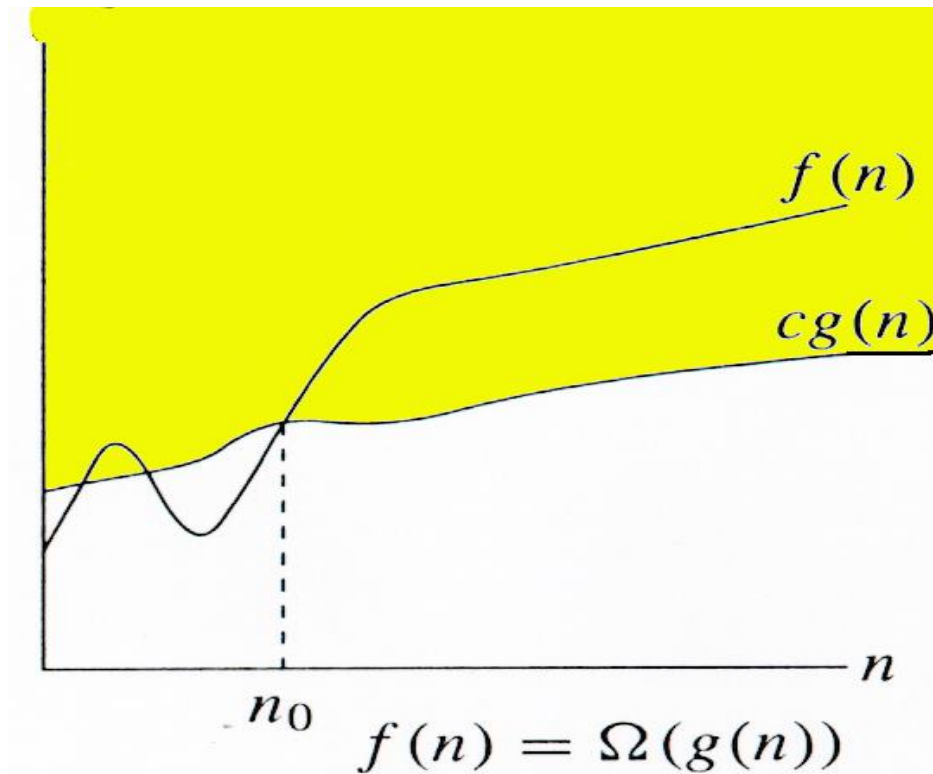
Example

Let us consider a given function,

$$f(n)=4.n^3+10.n^2+5.n+1$$

Considering $g(n)=n^3$,

$$f(n) \geq 4.g(n) \text{ for all the values of } n>0.$$

Hence, the complexity of *f(n)* can be represented as
$$\Omega (g(n)), \text{ i.e. } \Omega(n^3)$$

Small Oh Notation

- Upper bound that is not asymptotically tight

    $f(n) = o(g(n))$ if $f(n) < c(g(n))$ for all $n \geq n0$

- $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

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

- $g(n)$ is an **upper bound** for $f(n)$ that is not asymptotically tight.

Example

Let us consider a given function,

$$f(n)=4.n^3+10.n^2+5.n+1$$

Considering $g(n)=n^4$,

$$\lim_{n\to\infty} (4.n^3+10.n^2+5.n+1/ n^4)=0$$

Hence, the complexity of *f(n)* can be represented as
o(g(n)), i.e. o(n^4)

Small Omega Notation

- lower bound that is not asymptotically tight
  $$f(n) = \omega(g(n)) \text{ if } c(g(n)) < f(n) \text{ for all } n \geq n_0$$

- $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:
  $$\lim_{n \to \infty} [f(n) / g(n)] = \infty.$$

- $g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Example

Let us consider a given function,

$$f(n)=4.n^3+10.n^2+5.n+1$$

Considering $g(n)=n^2$,

$$\lim_{n\to\infty} (4.n^3+10.n^2+5.n+1/ n^2)=0$$

Hence, the complexity of *f(n)* can be represented as
$\omega (g(n))$, i.e. $\omega(n^4)$

Abstract Data Types

➢ Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.

➢ The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.

➢ It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

➢ It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Abstract Data Types

An ADT is a mathematical model of a data structure that specifies the type of **data** stored, the **operations** supported on them, and the types of **parameters of the operations**. An ADT specifies what each operation does, but not how it does it. Typically, an ADT can be implemented using one of many different data structures. A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.
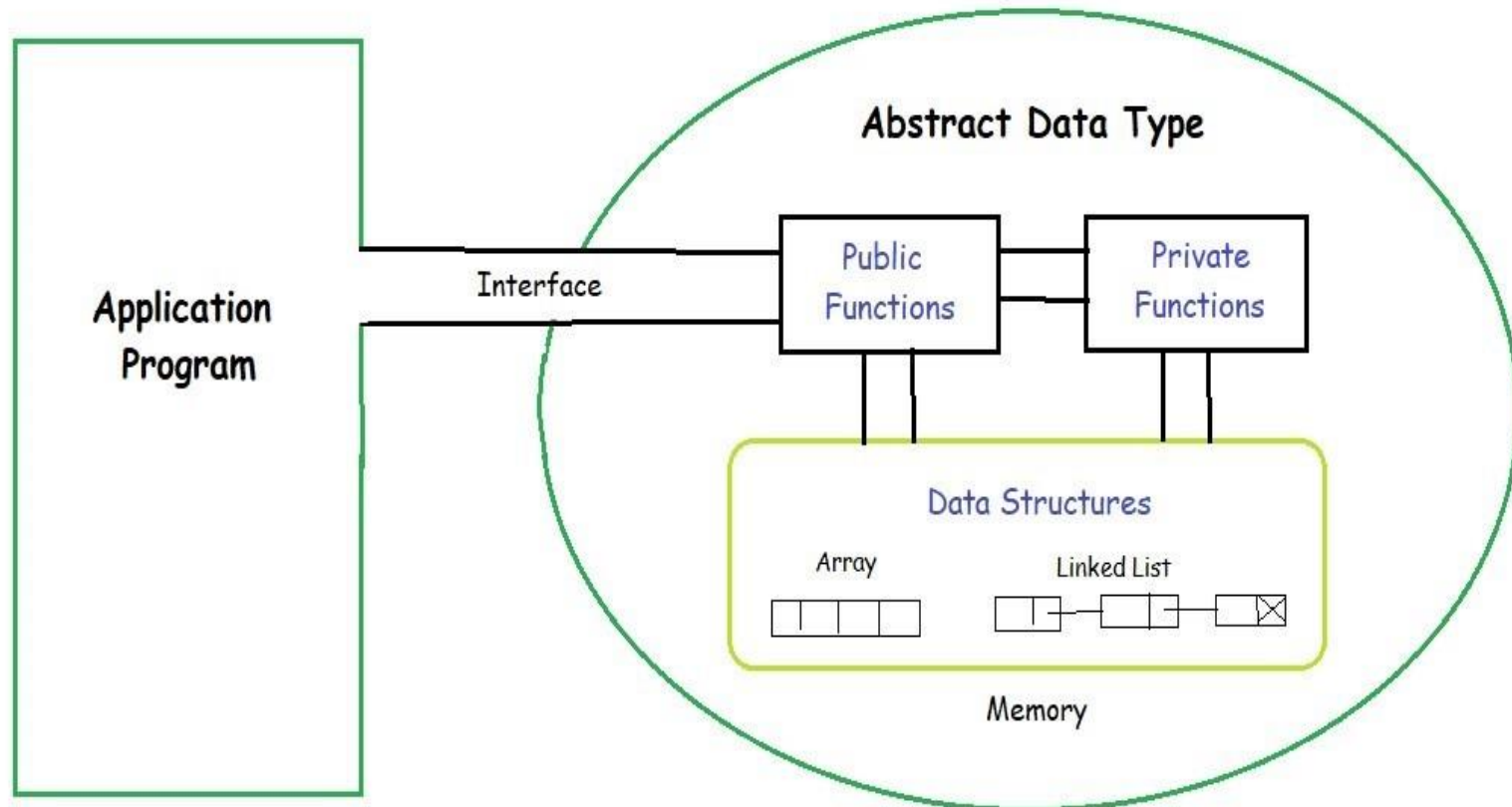
Abstract Data Types

In general, the steps of building ADT to data structures are:

1.Understand and clarify the nature of the target information unit.

2.Identify and determine which data objects and operations to include in the models.

3.Express this property somewhat formally so that it can be understood and communicate well.

4.Translate this formal specification into proper language Upon finalized specification, write necessary implementation. This includes storage scheme and operational detail. Operational detail is expressed as separate functions (methods).

Array

- Array is defined as the collection of similar type of data items stored at contiguous memory locations.

- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
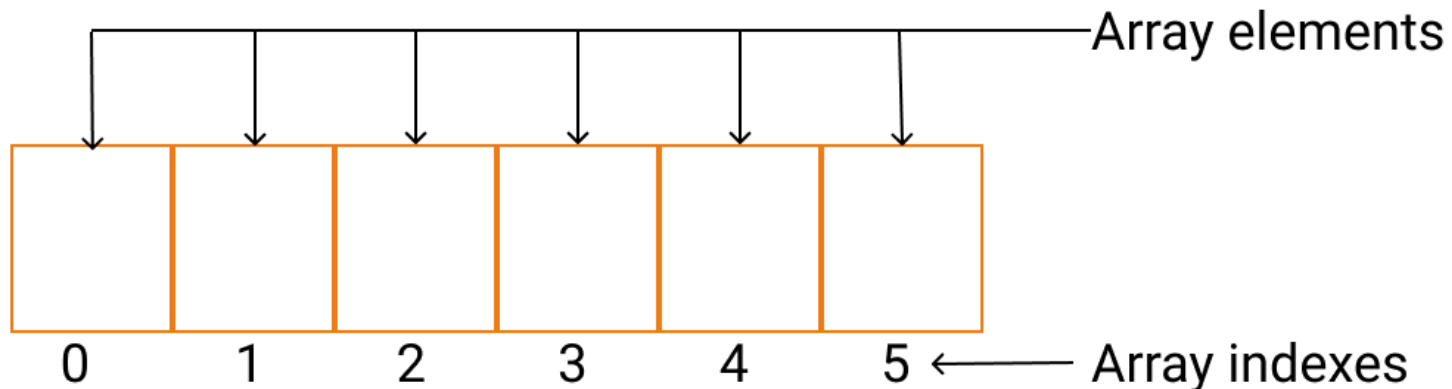
Array

- **Array Index:** The location of an element in an array has an index, which identifies the element. Array index starts from 0.

- **Array element:** Items stored in an array is called an element. The elements can be accessed via its index.

- **Array Length:** The length of an array is defined based on the number of elements an array can store. In the above example, array length is 6 which means that it can store 6 elements.

Need for an Array

- Arrays are best for storing multiple values in a single variable.

- Arrays are better at processing many values easily and quickly.

- Sorting and searching the values is easier in arrays.

There is some specific operation that can be performed or those that are supported by the array. These are:

Traversing: It prints all the array elements one after another.

Inserting: It adds an element at given index.

Deleting: It is used to delete an element at given index.

Searching: It searches for an element(s) using given index or by value.

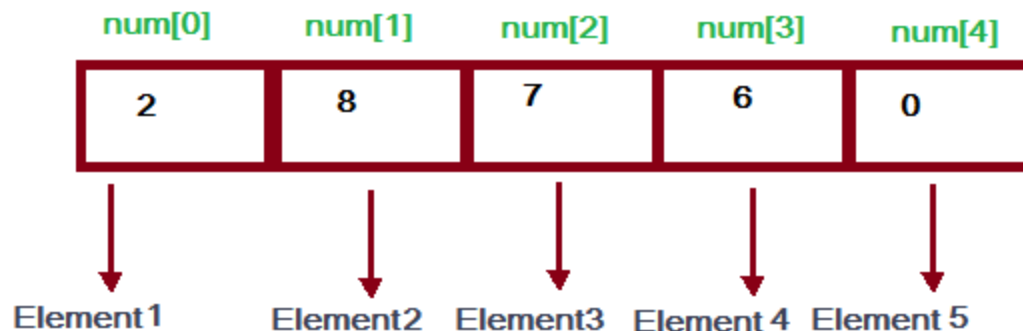**Updating:** It is used to update an element at given index.

Types of Array:- The various types of arrays are as follows.

➢ One dimensional array or Linear Array

➢ Multi-dimensional array

- **One dimensional (1-D) arrays or Linear arrays:**

  In it each element is represented by a single subscript.

  The elements are stored in consecutive memory

  locations. E.g. A [0], A [1], ….., A [N-1].

Calculating the address Of any element In the 1-D array:

A 1-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript that can either represent a row or column index.

$$Address\ of\ A[I] = B + W * (I - LB)$$

I = Subset of element whose address to be found,

B = Base address,

W = Storage size of one element store in any array(in byte),

LB = Lower Limit/Lower Bound of subscript(If not specified assume zero).

Example:

Given the base address of an array **A[1300…………1900]** as **1020** and the size

of each element is 2 bytes in the memory, find the address of **A[1700]**?

Solution:

Address of A[I] = B + W * (I – LB)

Address of A[1700] = 1020 + 2 * (1700 – 1300)

= 1020 + 2 * (400)

= 1020 + 800

Address of A[1700] = 1820

2. Multi dimensional arrays:

(a) **Two dimensional (2-D) arrays or Matrix arrays:**

In it each element is represented by two subscripts. Thus a two dimensional m x n array A has m rows and n columns and contains m*n elements. It is also called matrix array because in it the elements form a matrix. E.g. A [2] [3] has 2 rows and 3 columns and 2*3 = 6 elements.

Two dimensional (2-D) arrays or Matrix arrays:

Columns

	0	1	2	3
0	[0] [0]	[0] [1]	[0] [2]	[0] [3]
1	[1] [0]	[1] [1]	[1] [2]	[1] [3]
2	[2] [0]	[2] [1]	[2] [2]	[2] [3]

Rows

1st Subscript
indicating the rows

2nd Subscript
indicating the columns

There are two main techniques of storing 2D array elements into memory.

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously.

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

first, the 1st row of the array is stored into the memory

completely, then the 2nd row of the array is stored into the

memory completely and so on till the last row.

Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$
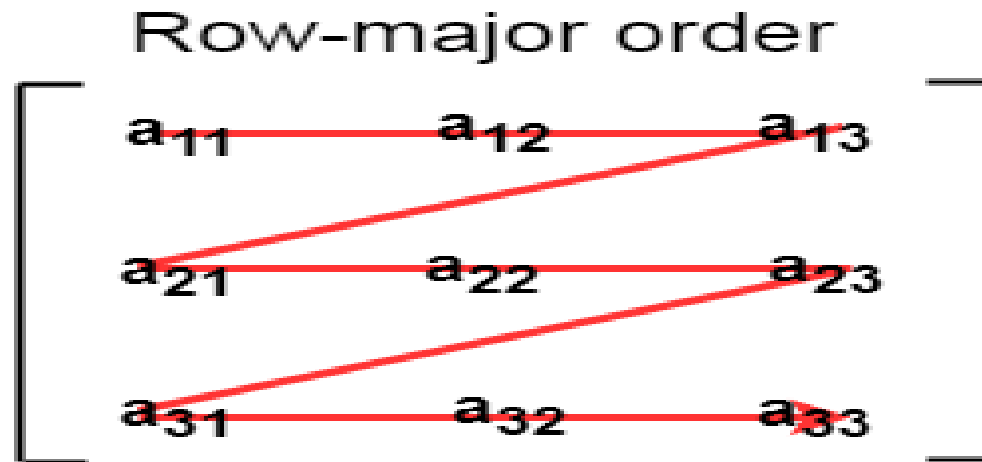
Calculate the address of any element in the 2-D array: Row Major Order: assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are being stored in a Row-Wise fashion. To find the address of the element using row-major order use the following formula:

$$\text{Address of A[I][J]} = B + W * ((I - LR) * N + (J - LC))$$

I = Row Subset of an element whose address to be found,

J = Column Subset of an element whose address to be found,

B = Base address,

W = Storage size of one element store in an array(in byte),

LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),

LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),

N = Number of column given in the matrix.

Example: Given an array, **arr[1………10][1………15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order?
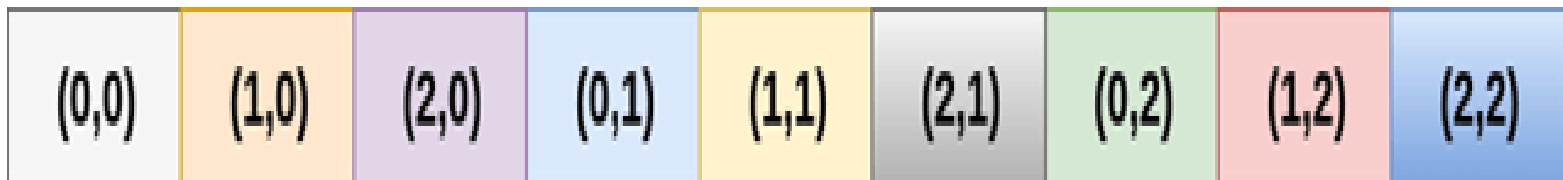
Solution:

*Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))*

$$= 100 + 1 * ((7) * 15 + (5))$$

$$= 100 + 1 * (110)$$

Address of A[I][J] = 210

2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.

(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$
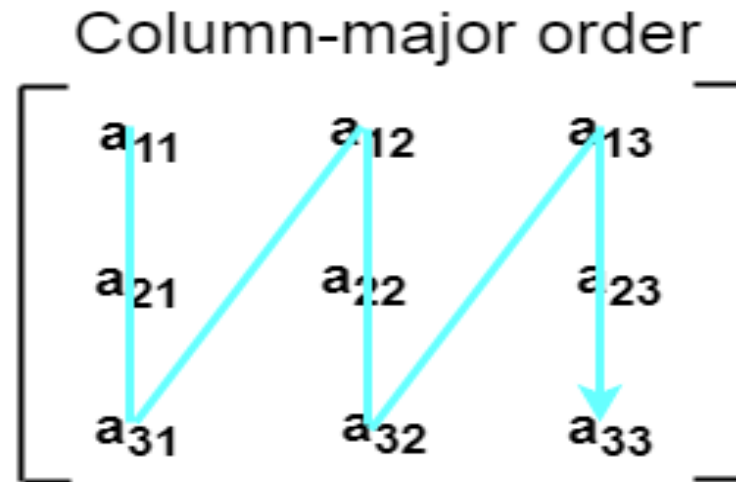
Column Major Order: If elements of an array are stored in column-major fashion means moving across the column and then to the next column then it's in column-major order. To find the address of the element using column-major order uses the following formula:

$$\text{Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))}$$

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
M = Number of rows given in the matrix.

Example: Given an array **arr[1………10][1………15]** with base value **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.

Formula:

*Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*

*Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*
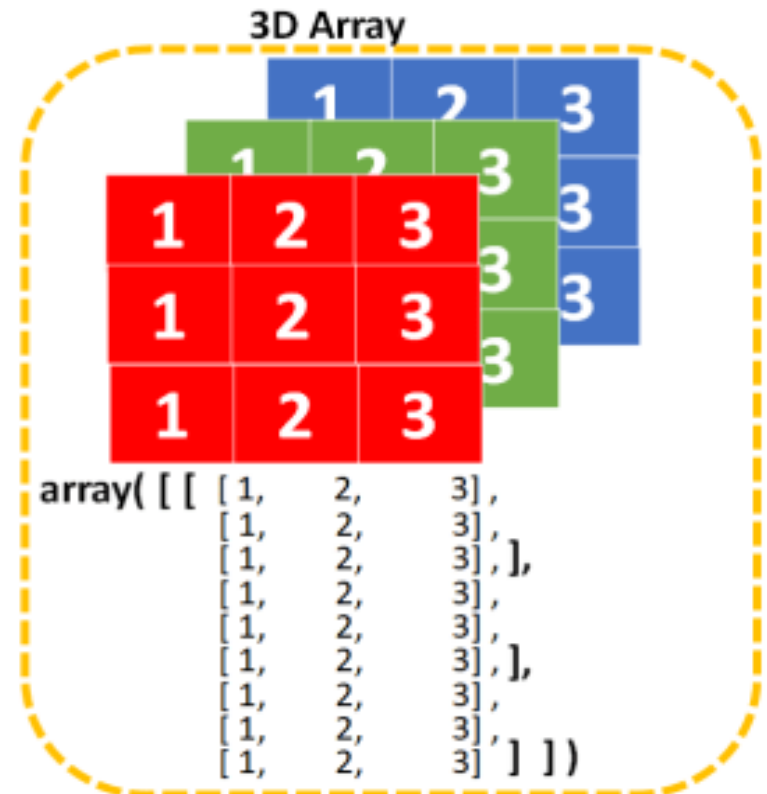
$$= 100 + 1 * ((5) * 10 + (7))$$

$$= 100 + 1 * (57)$$

Address of A[I][J] = 157

(b) Three dimensional arrays:

In it each element is represented by three subscripts. Thus a

three dimensional m x n x l array A contains m*n*l elements. E.g.

A [2] [3] [2] has 2*3*2 = 12 elements.

Applications of Arrays

● 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   Bubble Sort

      Selection Sort   Quick Sort

      Merge Sort, etc.,

● Arrays are used to implement Data structures.

      Stack Using Arrays   Queue Using Arrays

Syntax to Create an Array in Python

We can declare an array in Python while initializing it using the following syntax.

arrayName = array.array(data type, [array,items])
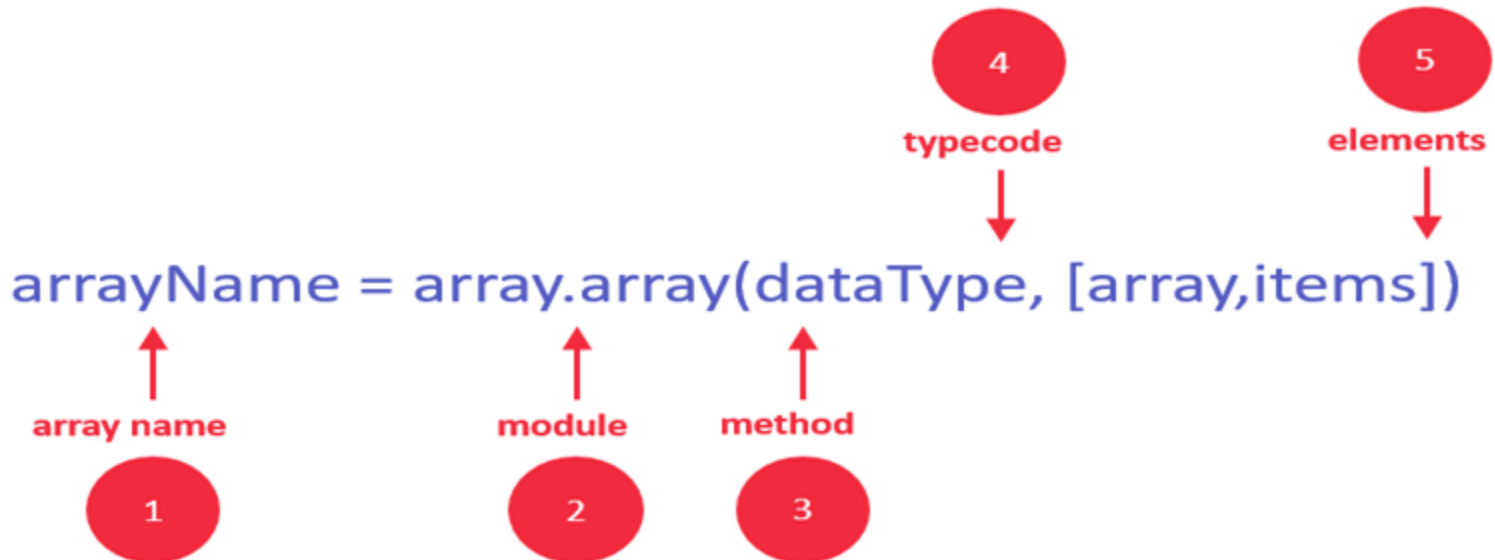
1. **Identifier**: specify a name like usually, you do for variables

2. **Module**: Python has a special module for creating array in Python, called "array" – you must import it before using it

3. **Method**: the array module has a method for initializing the array. It takes two arguments, type code, and elements.

4. **Type Code**: specify the data type using the type codes available

5. **Elements**: specify the array elements within the square brackets, for example [130,450,103]

Example

In Python, we use following syntax to create arrays:

array.array(type code,[initializer])

For Example

import array as myarray

abc = myarray.array('d', [2.5, 4.9, 6.7])

Traverse an Array

We can traverse a Python array by using loops, like this one:

```
import array

 balance = array.array('i',  [300,200,100])

for x in (balance):

     print(x)
```

Output:

200 300 100

Access array elements:-

We can access any array item by using its index.

The syntax is

      arrayName[indexNum]

For **example 1,**

    import array

    balance = array.array('i', [300,200,100])

    print(balance[1])

Output:

    200

Example 2:

import array as myarray

abc = myarray.array('d', [2.5, 4.9, 6.7])

print("Array first element is:",abc[0])

print("Array last element is:",abc[-1])

Output:

Array first element is: 2.5

Array last element is: 6.7

Insert element into an array

Python array insert operation enables you to insert one or more items into an array at the beginning, end, or any given index of the array. This method expects two arguments index and value.

The syntax is

arrayName.insert(index, value)

In order to insert the new value right "after" index 1, we need to

reference index 2 in your insert method, as shown in the below

Python array example:

```
import array

balance = array.array('i',  [300,200,100])

balance.insert(2, 150)

print(balance)
```

Output:

```
array('d',[300 ,200, 150 ,100])
```

Delete element from an array

With remove operation, we can delete one item from an array

by value. This method accepts only one argument, value.

After running this method, the array items are re-arranged,

and indices are re-assigned.

The syntax is

arrayName.remove(value)

Example to Delete element from an array

Let's remove the value of "3" from the array

```
import array as myarray

first = myarray.array('b', [2, 3, 4])

first.remove(3)

print(first)
```

Output:

```
array('b', [2, 4])
```

Dr Kumud Saxena   DS Unit 1

Sparse Matrix

A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m*n matrix. Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Need to use a sparse matrix :-

We can also use the simple matrix to store the elements in the memory; then why do we need to use the sparse matrix. The following are the advantages of using a sparse matrix:

1. **Storage:** a sparse matrix that contains lesser non-zero elements than zero so less memory can be used to store elements. It evaluates only the non-zero elements.

2. **Computing time:** In the case of searching n sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Sparse Matrix Representations

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)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \implies$$
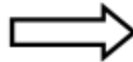
Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Dr Kumud Saxena   DS Unit 1

Program for Sparse Matrix representation.

```
Can be created by using list
1. Create empty list
2. Find non zero elements
3. If non zero element create a temporary list
4. Append row, col and data in list
5. Append temporary list in sparse matrix

By using SciPy
import numpy as np
from scipy.sparse import csr_matrix/csc_matrix

# create a 2-D representation of the matrix

# convert to sparse matrix representation
S = csr_matrix(A)

# convert back to 2-D representation of the matrix
D = S.todense()
```

- *Searching* is the technique of selecting specific data from a collection of data based on some condition.

- The process of finding an element from a list of elements is known as searching.

- Searching is the process of locating given value position in a list of values.

Unsuccessful Search: search(40)   Not Found

Successful Search: search(15)   Found at **3** position

0	1	2	3	4	5	6
25	**10**	**20**	**15**	**17**	**27**	**10**

linear search:- which simply checks the values in sequence until the desired value is found

*binary search***:-** which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are definitely either all larger or all smaller than the desired value

- **linear search *Steps:-***

- Start from the leftmost element of the list and one by one compare value with each number of the list.

- If value matches with an element, return `True`.

- else, return `False`

Find '20'

0	1	2	3	4	5	6	7	8
10	50	30	70	80	60	20	90	40
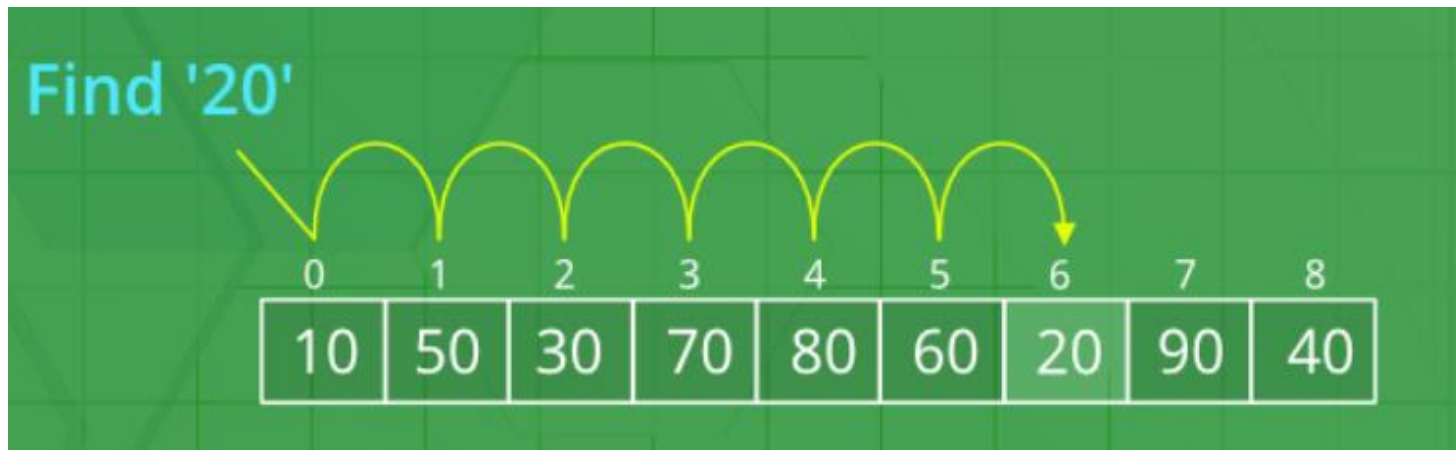
- Search the array from first to the last position in linear progression.

Linear Search

| 10 | 14 | 19 | 26 | 27 | 31 | 33 | 35 | 42 | 44 |

=
33

```
def search(l, n, x):
    for i in range(0, n):
        if (l[i] == x):
            return i
    return -1
l = [2, 3, 4, 10, 40]
x = 10
n = len(l)

result = search(l, n, x)
if(result == -1):
    print("Element is not present in list")
else:
    print("Element is present at list", result)
```

***Binary search*:-** which requires a sorted input list, and checks for

the value in the middle of the list, repeatedly discarding the half

of the list which contains values which are definitely either all

larger or all smaller than the desired value

Binary search:- searching for a value in a sorted list, the algorithm compares value to the middle element of the list.

•If mid is the element we are looking for, we return its index.

•If not, we identify which side of mid value is more likely to be on based on whether value is smaller or greater than mid, and discard the other side of the array.

•We then recursively or iteratively follow the same steps, choosing a new value for mid, comparing it with value and discarding half of the possible matches in each iteration of the algorithm.

Low High Mid

#1 0 8 4

Search (45)

$$mid = \left[\frac{low + high}{2}\right]$$

0	1	2	3	4	5	6	7	8
6	12	17	23	38	45	77	84	90

Low Mid High

38 < 45 ⟶ Low = Mid + 1 = 5

© w3resource.com
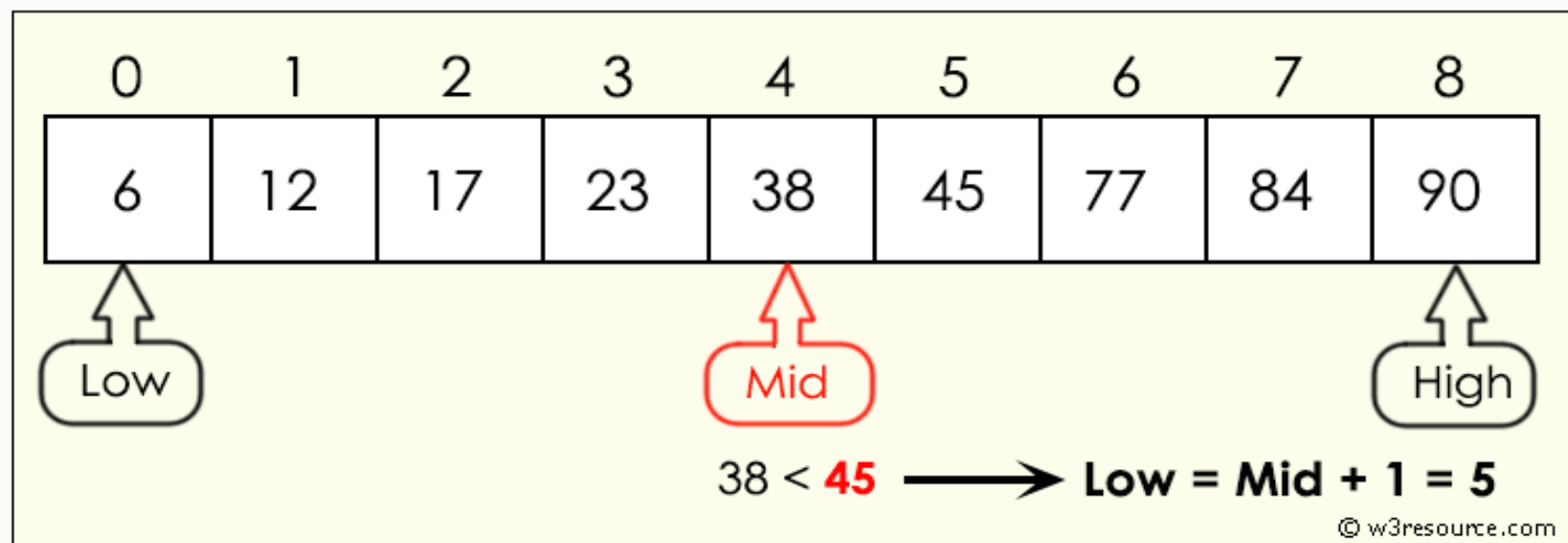
	Low	High	Mid
#1	0	8	4
#2	5	8	6

Search (45)

$$mid = \left[\frac{low + high}{2} \right]$$

0	1	2	3	4	5	6	7	8
					45	77	84	90

Low — 45   Mid — 77   High — 90

High = Mid - 1 = 5 ⟵ **45** < 77

© w3resource.com
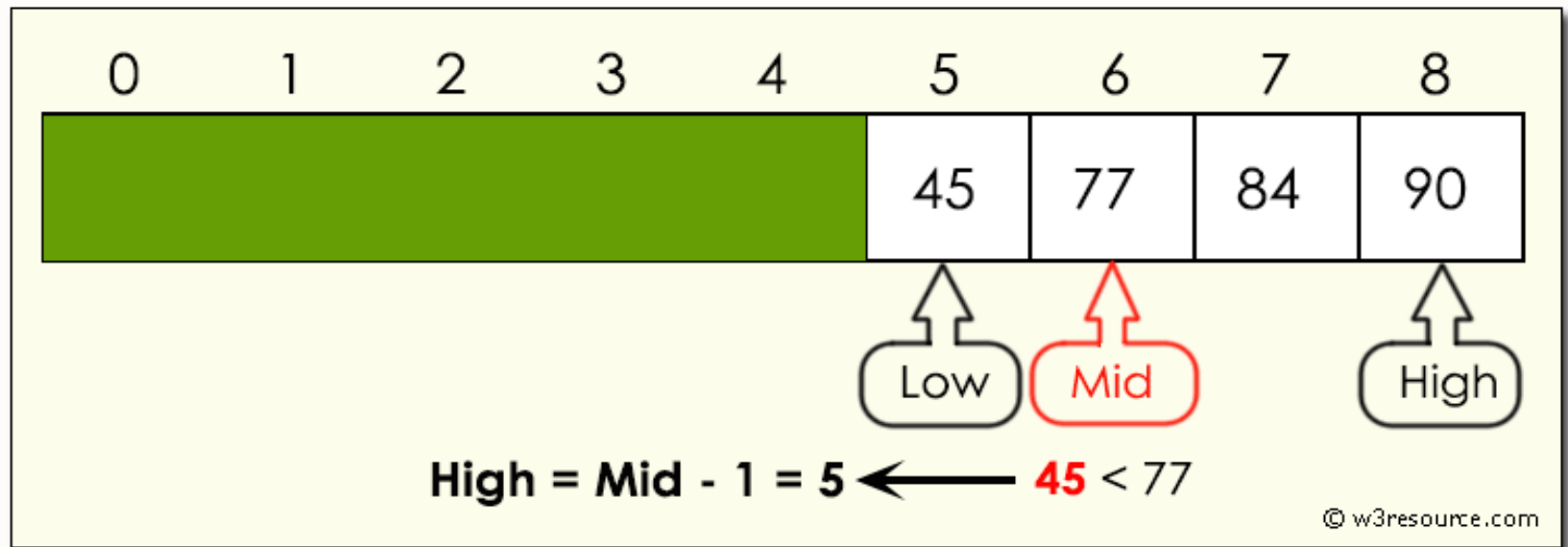
	Low	High	Mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

Search (45)

$$mid = \left[\frac{low + high}{2}\right]$$

0	1	2	3	4	5	6	7	8
					45			

Successful Search !!

Low   High

Mid

45 = = 45

© w3resource.com
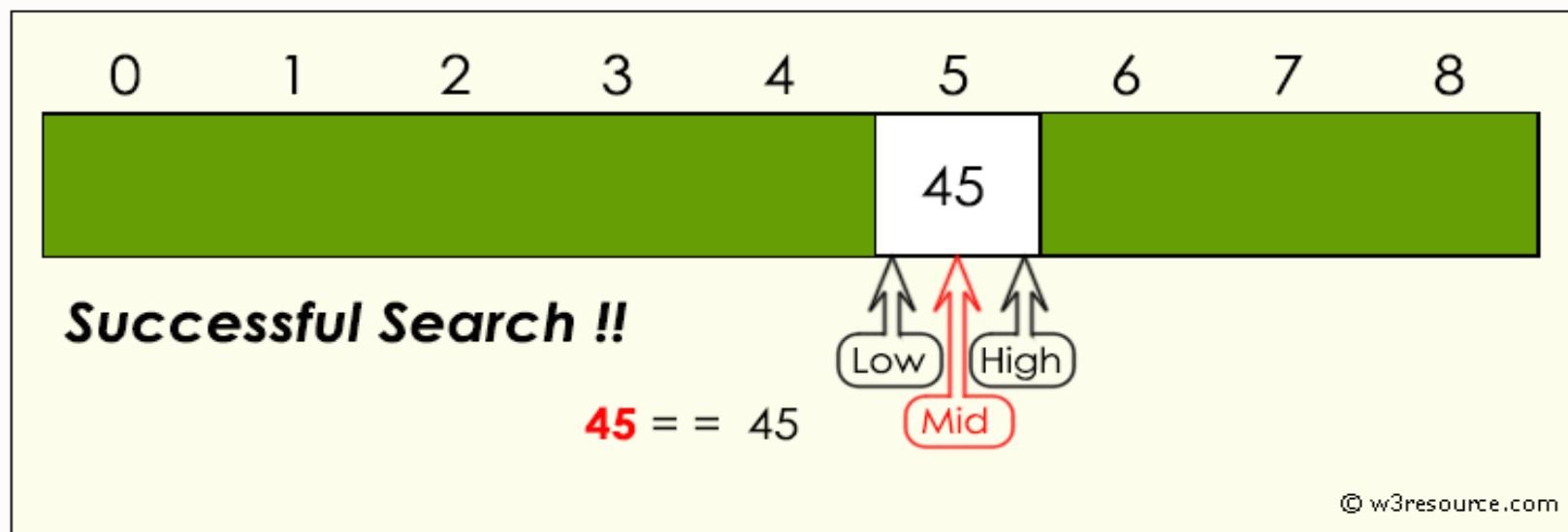
search element **12**

Step 1:

search element (12) is compared with middle element (50)

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Search for 47

0	4	7	10	14	23	45	47	53

Binary Search(CO1)

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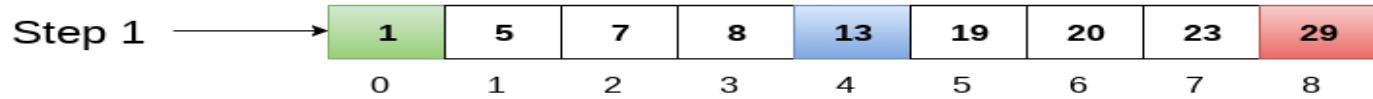
a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6

Step 2	1	5	7	8	13	**19**	**20**	23	**29**
	0	1	2	3	4	5	6	7	8
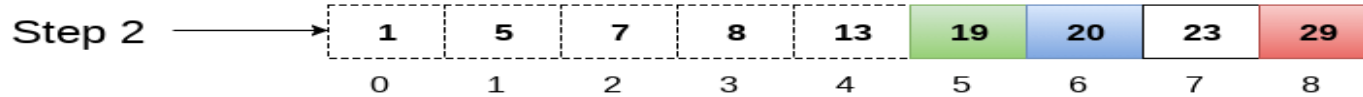
a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7

Step 3	1	5	7	8	13	19	20	**23**	**29**
	0	1	2	3	4	5	6	7	8
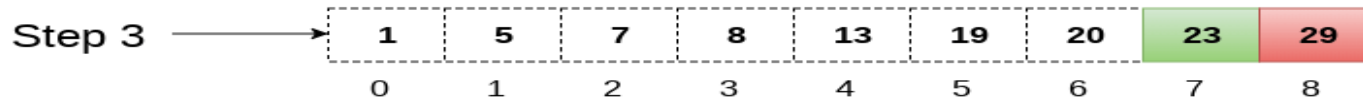
a [mid] = 23
23 = 23
loc = mid

Return location 7

Problem statement – We will be given a sorted list and we need to find an element with the help of a binary search.

Algorithm

1. Compare x with the middle element.

2. If x matches with the middle element, we return the mid index.

3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.

4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

```python
def binarysearch(a, val):
    first = 0
    last = len(a)-1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first+last)//2
        if a[mid] == val:
            index = mid
        else:
            if val<a[mid]:
                last = mid -1
            else:
                first = mid +1
    return index
```

➢ Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order.

➢ Sorting can be done in ascending and descending order.

➢ It arranges the data in a sequence which makes searching easier.

Sorting can be performed using several techniques or methods, as follows:

1. Bubble Sort

2. Insertion Sort

3. Selection Sort

4. Radix Sort

5. Merge Sort

6. Quick Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1.Starting with the first element(index = 0), compare the current element with the next element of the array.

2.If the current element is greater than the next element of the array, swap them.

3.If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Bubble sort

Array 6 3 0 5 1

Bubble sort Illustration

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

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 Execution Example

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.

Selection Sort
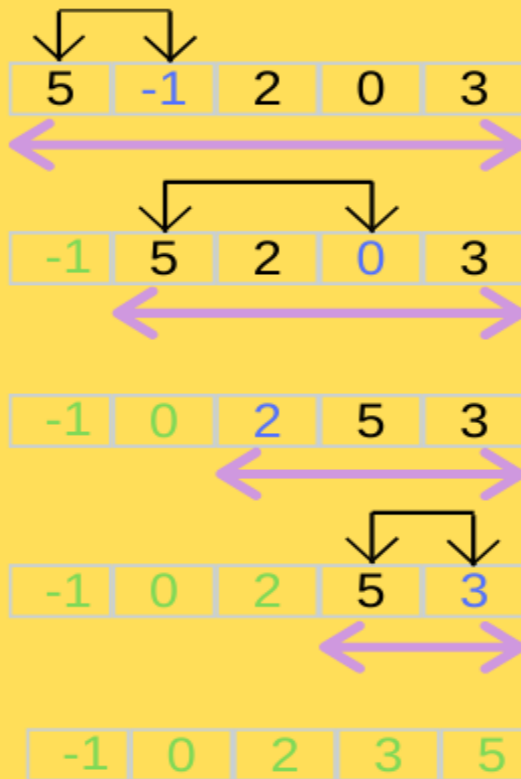
Green = Sorted
Blue = Current minimum

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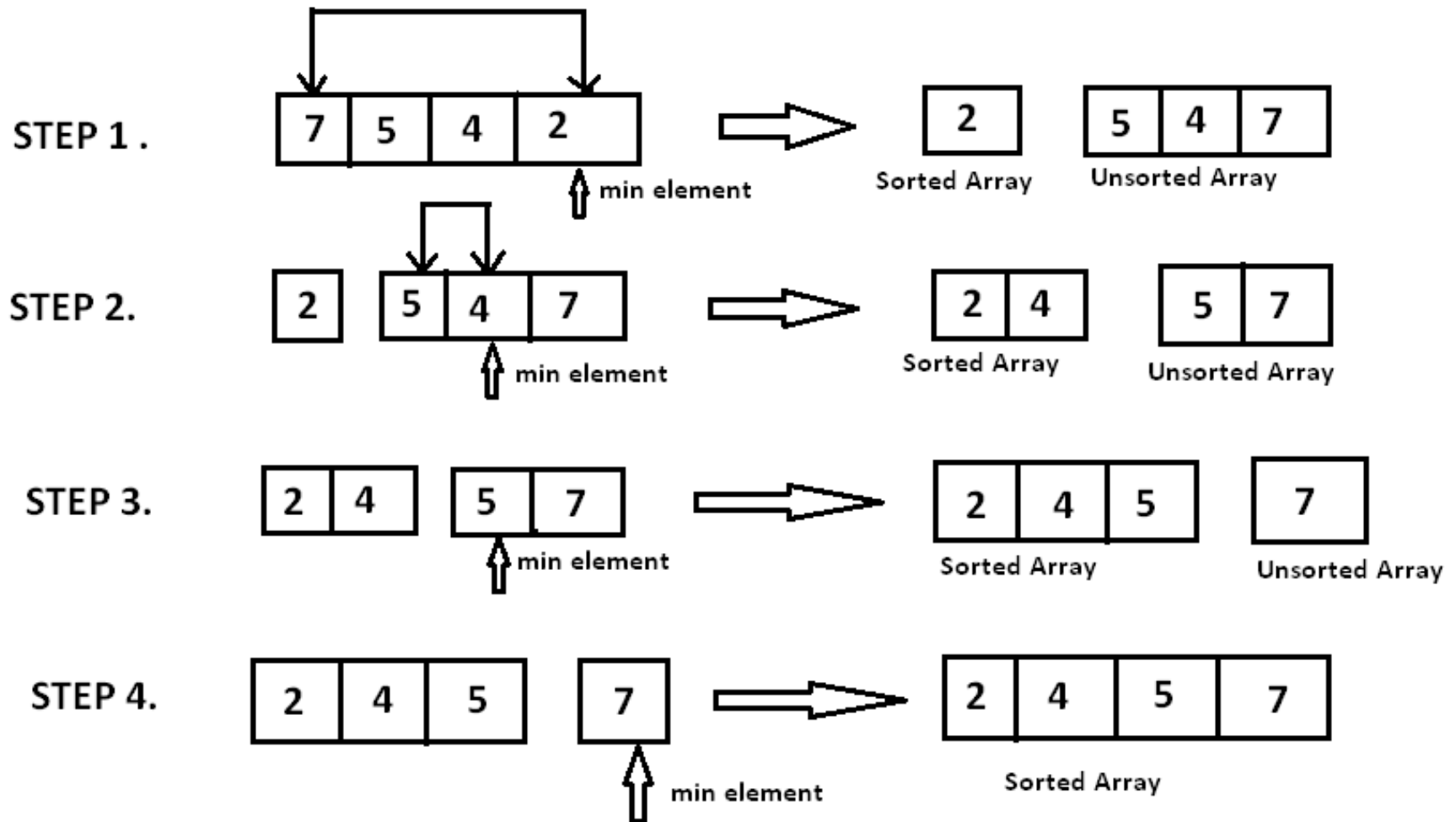
Steps:-

Step 1) Get the value of n which is the total size of the array

Step 2) Partition the list into sorted and unsorted sections. The sorted section is initially empty while the unsorted section contains the entire list

Step 3) Pick the minimum value from the unpartitioned section and placed it into the sorted section.

Step 4) Repeat the process (n – 1) times until all of the elements in the list have been sorted.

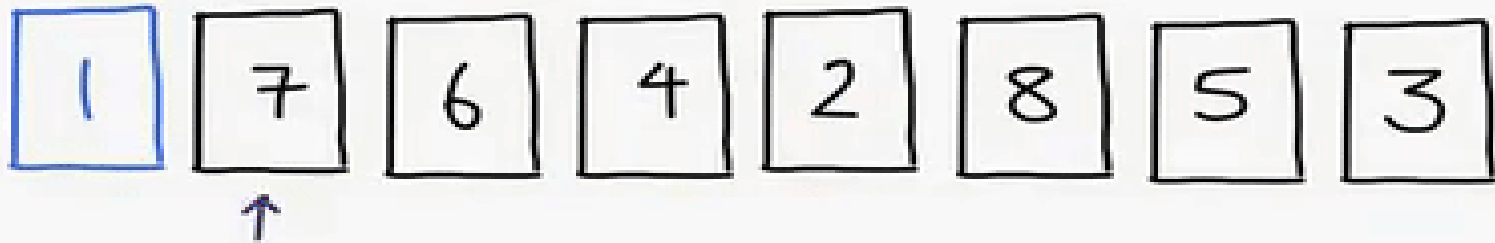STEP 1.   7 5 4 2   ⟹ Sorted Array: 2   Unsorted Array: 5 4 7

min element

STEP 2.   2   5 4 7   ⟹ Sorted Array: 2 4   Unsorted Array: 5 7

min element

STEP 3.   2 4   5 7   ⟹ Sorted Array: 2 4 5   Unsorted Array: 7

min element

STEP 4.   2 4 5   7   ⟹ Sorted Array: 2 4 5 7

min element

Boxes containing: 1, 7, 6, 4, 2, 8, 5, 3

Minimum : 7

```python
def selection_sort(a):
    for i in range(0, len(a) - 1):
        smallest = i
        for j in range(i + 1, len(a)):
            if a[j] < a[smallest]:
                smallest = j
                temp=a[i]
                a[i]=a[smallest]
                a[smallest]=a[i]


a= [40,90,100,2, 4, 7, 10,105,107]
selection_sort(a)
print('Sorted list: ', end='')
print(a)
```

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers.

Step 1:Find the largest number in ARR as LARGE

Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE

Step 3: SET PASS =0

Step 4: Repeat Step 5 while PASS <= NOP-1

Step 5: SET I = 0 and INITIALIZE buckets

Step 6:Repeat Steps 7 to 9 while I<n-1< li=""></n-1<>

Step 7: SET DIGIT = digit at PASSth place in A[I]

Step 8: Add A[I] to the bucket numbered DIGIT

Step 9: INCREMENT bucket count for bucket numbered DIGIT [END OF LOOP]

Step 10: Collect the numbers in the bucket [END OF LOOP]

Step 11: END

A = {10, 2, 901, 803, 1024}

Pass 1: (Sort the list according to the digits at 0's place)

10, 901, 2, 803, 1024.

Pass 2: (Sort the list according to the digits at 10's place)
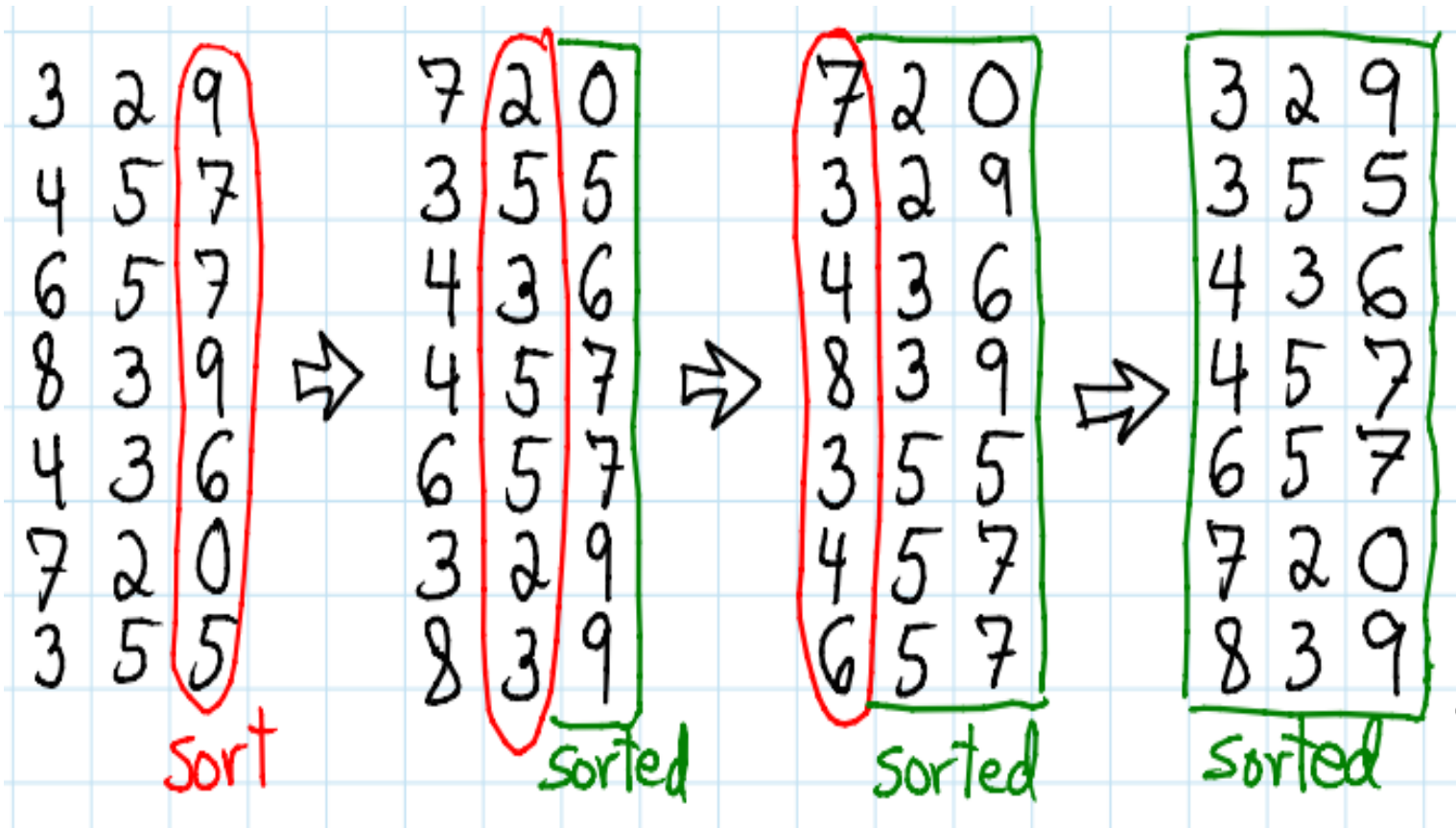
02, 10, 901, 803, 1024

Pass 3: (Sort the list according to the digits at 100's place)
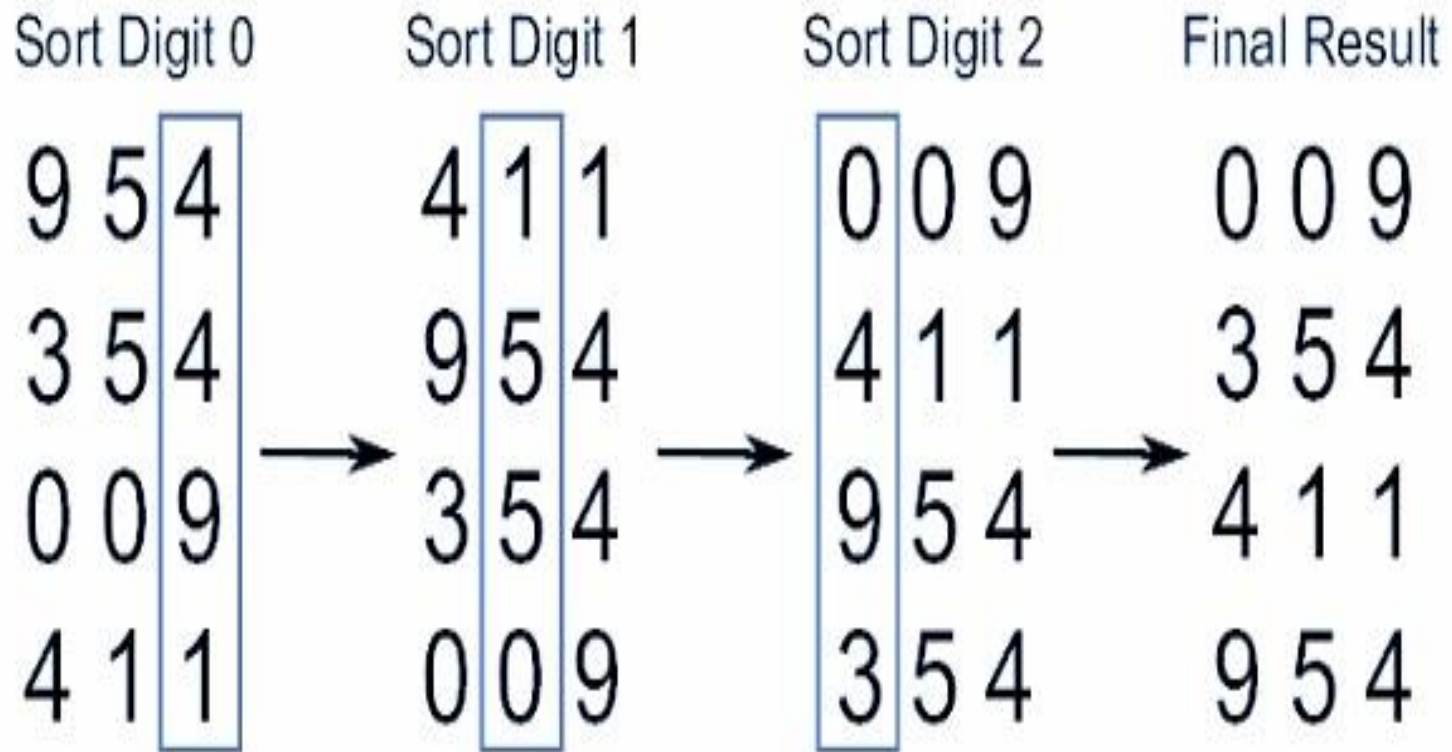
02, 10, 1024, 803, 901.

Pass 4: (Sort the list according to the digits at 1000's place)

02, 10, 803, 901, 1024

•Merge Sort is based on the divide and conquer algorithm where the input array is divided into two halves, then sorted separately and merged back to reach the solution. Merge sort works as follows :

- ■ Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- ■ Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.
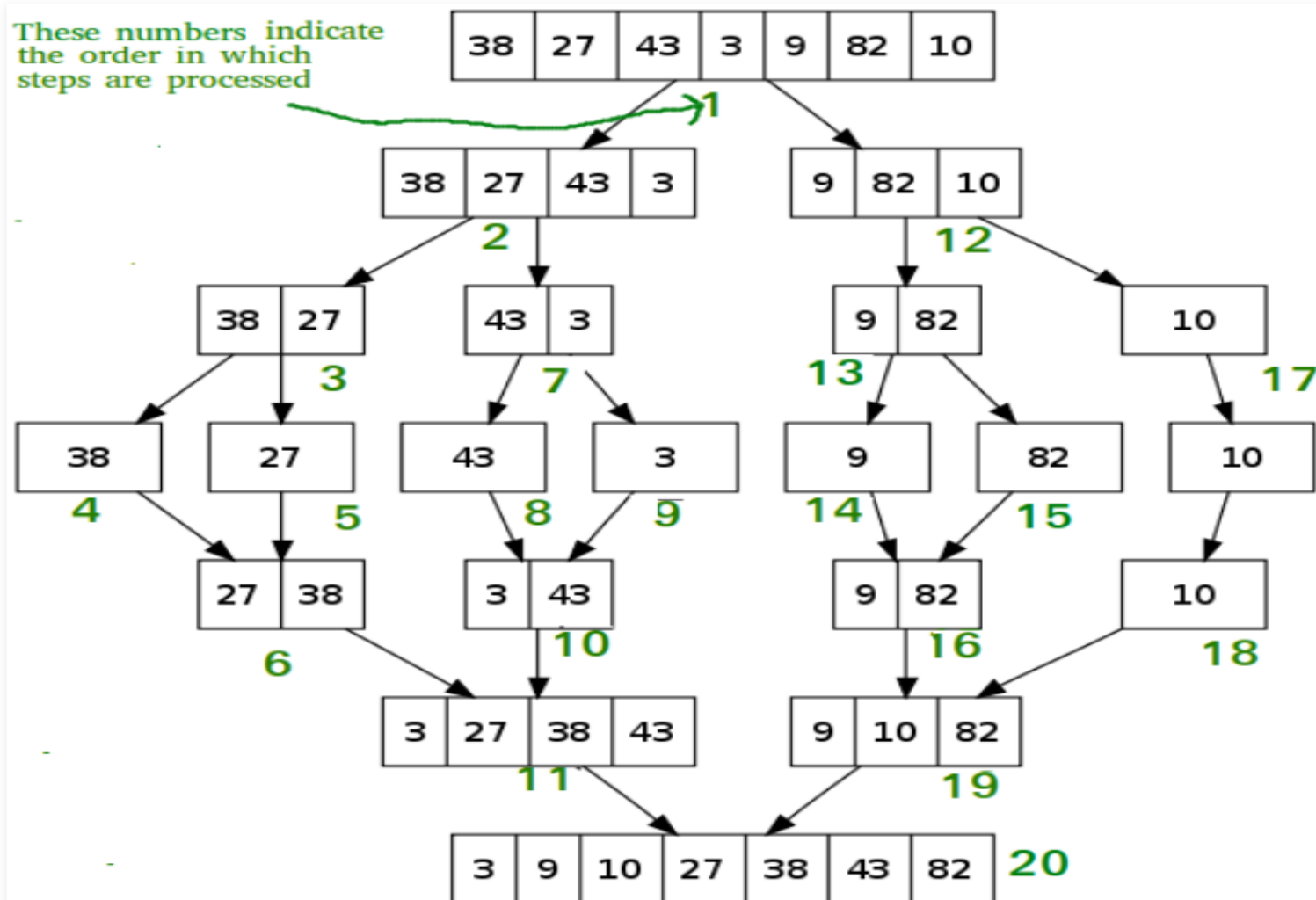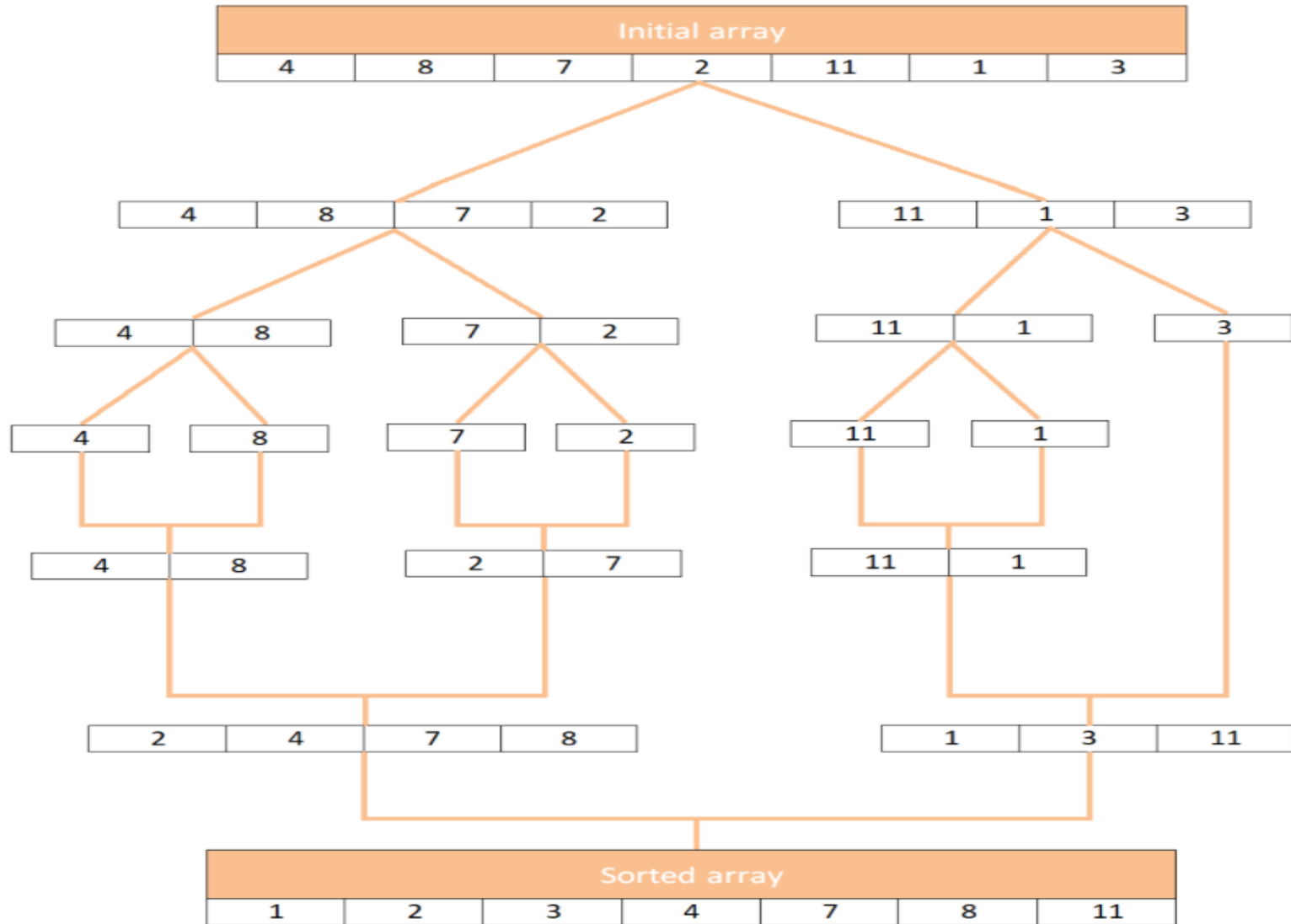
Merge sort Steps (CO1)

➢ The list is divided into left and right in each recursive call until two adjacent elements are obtained.

➢ Now begins the sorting process. The i and j iterators traverse the two halves in each call. The k iterator traverses the whole lists and makes changes along the way.

➢ If the value at i is smaller than the value at j, left[i] is assigned to the myList[k] slot and i is incremented. If not, then right[j] is chosen.

➢ This way, the values being assigned through k are all sorted.

➢ At the end of this loop, one of the halves may not have been traversed completely. Its values are simply assigned to the remaining slots in the list.

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).
   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

In Quick sort algorithm, partitioning of the list is performed using following steps…

Step 1 - Consider the first element of the list as **pivot** (i.e., Element at first position in the list).

Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3 - Increment i until list[i] > pivot then stop.

Step 4 - Decrement j until list[j] < pivot then stop.

Step 5 - If i < j then exchange list[i] and list[j].

Step 6 - Repeat steps 3,4 & 5 until i > j.

Step 7 - Exchange the pivot element with list[j] element.

Quick sort Example (CO1)

Unsorted Array

| 35 | 33 | 42 | 10 | 14 | 19 | 27 | 44 | 26 | 31 |

- Hashing is a technique or process of mapping keys, 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.

- Hashing is one of the searching techniques that uses a constant time.

- **Hash Function** is a function which applied to the key, produced an integer which can be used as an address in a hash table.

- The Hash table data structure stores elements in key-value pairs where

  - **Key**- unique integer that is used for indexing the values

  - **Value** - data that are associated with keys.
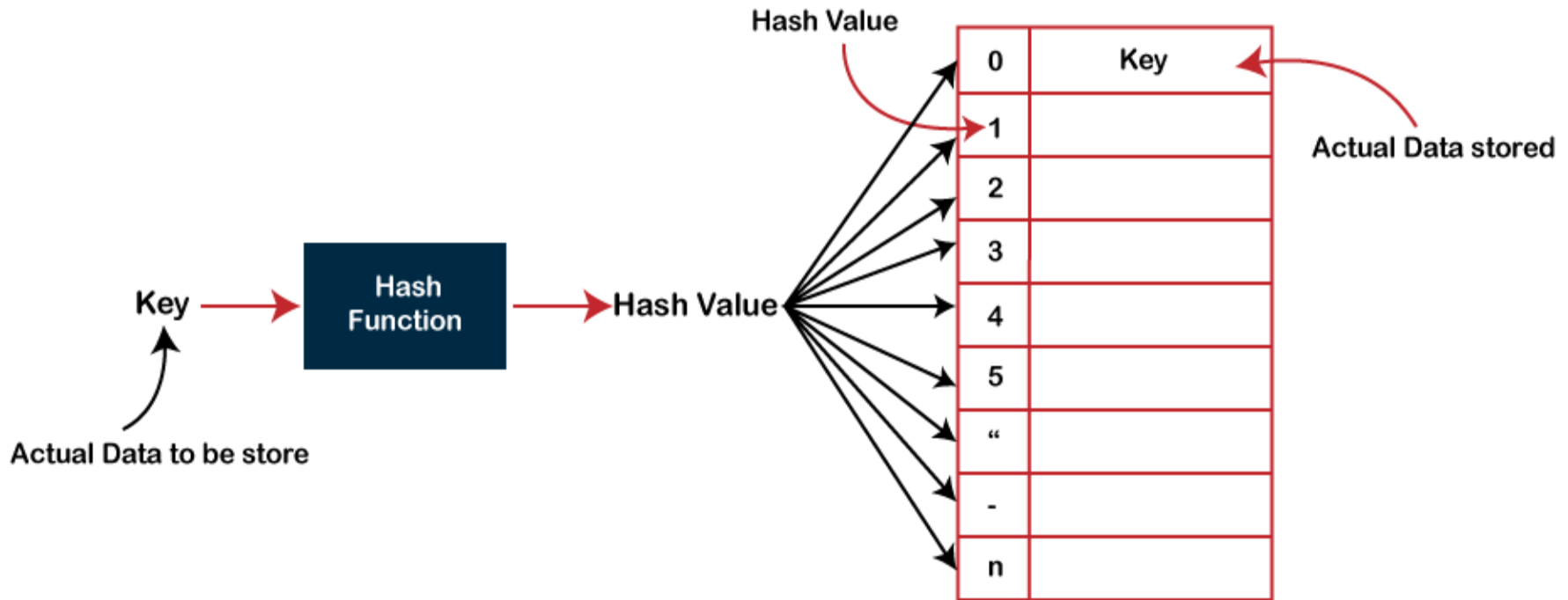
A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

Hash Function

Hash function is a function that maps any big number or string to a small integer value.

•Hash function takes the data item as an input and returns a small integer value as an output.

•The small integer value is called as a hash value.

•Hash value of the data item is then used as an index for storing it into the hash table.

Hash Functions

Types of Hash Functions-

There are various types of hash functions available such as-

1. Mid Square Hash Function

2. Division Hash Function

3. Folding Hash Function

Hashing

1. Division method

In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.

Then:

h(key)=record% table size.

2=52%10

8=68%10

9=99%10

4=84%10

DIVISION METHOD

0	
1	
2	52
3	
4	84
5	
6	
7	
8	68
9	99

2. Mid square method

In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. h (3101) = 162 (middle 3 digit)

3. Digit folding method

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12

=791

Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision.

Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.
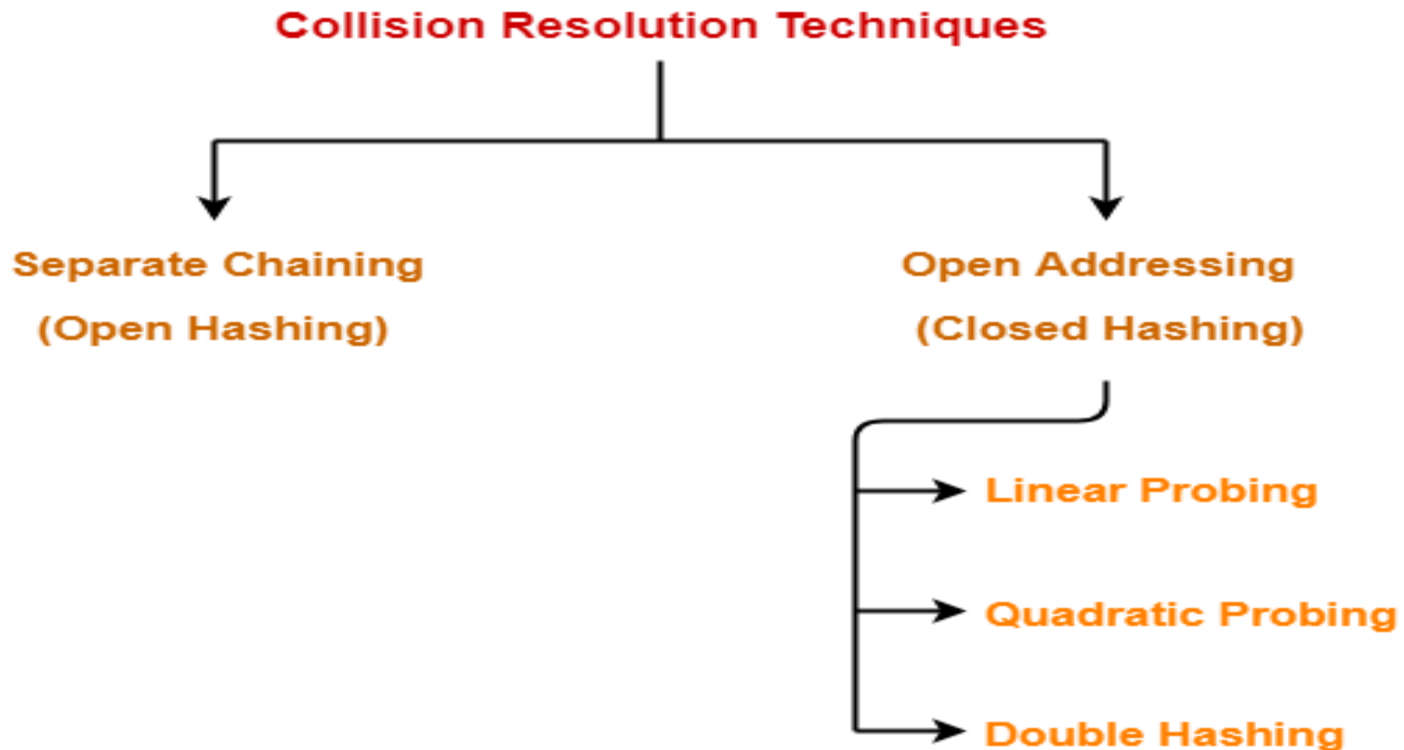
The following are the collision techniques:

•Open Hashing: It is also known as closed addressing.

•Closed Hashing: It is also known as open addressing.

Collision resolution technique

If there is a problem of collision occurs then it can be handled by applying some techniques. These techniques are called as collision resolution techniques.

Collision Resolution Techniques

Separate Chaining
(Open Hashing)

Open Addressing
(Closed Hashing)

- Linear Probing
- Quadratic Probing
- Double Hashing

Separate Chaining

In this technique, a linked list is created from the slot in which collision has occurred, after which the new key is inserted into the linked list. This linked list of slots looks like a chain, so it is called **separate chaining**. It is used more when we do not know how many keys to insert or delete.
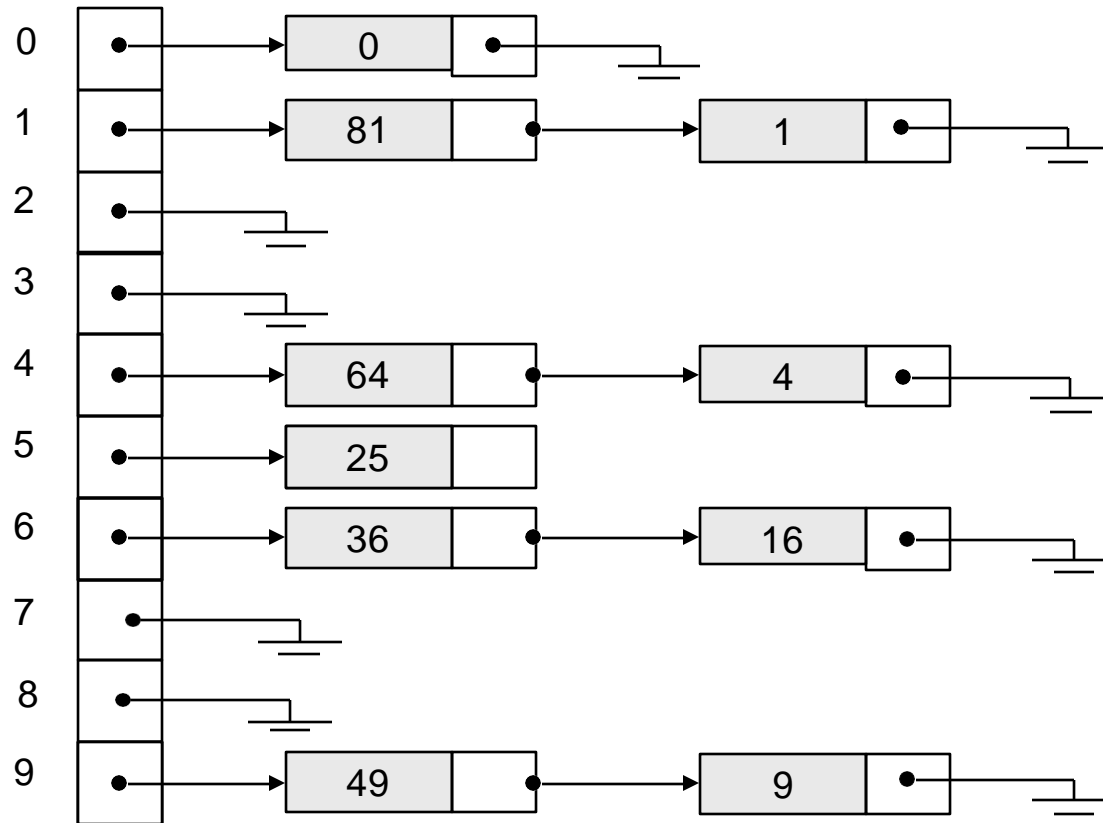
Time complexity

1.Its worst-case complexity for searching is o(n).

2.Its worst-case complexity for deletion is o(n).

Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.

Separate Chaining

Advantages of separate chaining

1.It is easy to implement.

2.The hash table never fills full, so we can add more elements to the chain.

3.It is less sensitive to the function of the hashing.

Disadvantages of separate chaining

1.In this, cache performance of chaining is not good.

2.The memory wastage is too much in this method.

3.It requires more space for element links.

Collision Resolution with Open Addressing

Open addressing is collision-resolution method that is used to control the collision in the hashing table. There is no key stored outside of the hash table. Therefore, the size of the hash table is always greater than or equal to the number of keys. It is also called **closed hashing**.

The following techniques are used in open addressing:

1. Linear probing

2. Quadratic probing

3. Double hashing

Hashing

Linear Probing

In this, when the collision occurs, we perform a linear probe for the next slot, and this probing is performed until an empty slot is found. In linear probing, the worst time to search for an element is O(table size).

$$h´(x) = x \; mod \; m$$

$$h(x, i) = (h´(x) + i) mod \; m$$

Hashing

Linear Probing Steps

1. Calculate the hash key. key = data % size;

   If hashTable[key] is empty, store the value directly.

   hashTable[key] = data.

2. If the hash index already has some value, check for next index.

   key = (key+1) % size;

   If the next index is available hashTable[key], store the

   value. Otherwise try for next index.

3. Do the above process till we find the space.

Hashing

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	50
2	
3	
4	
5	
6	

Insert 50

0	700
1	50
2	
3	
4	
5	
6	76

Insert 700 and 76

0	700
1	50
2	85
3	
4	
5	
6	76

Insert 85: Collision Occurs, insert 85 at next free slot.

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

Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot

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

Insert 73 and 101

Linear Probing Example
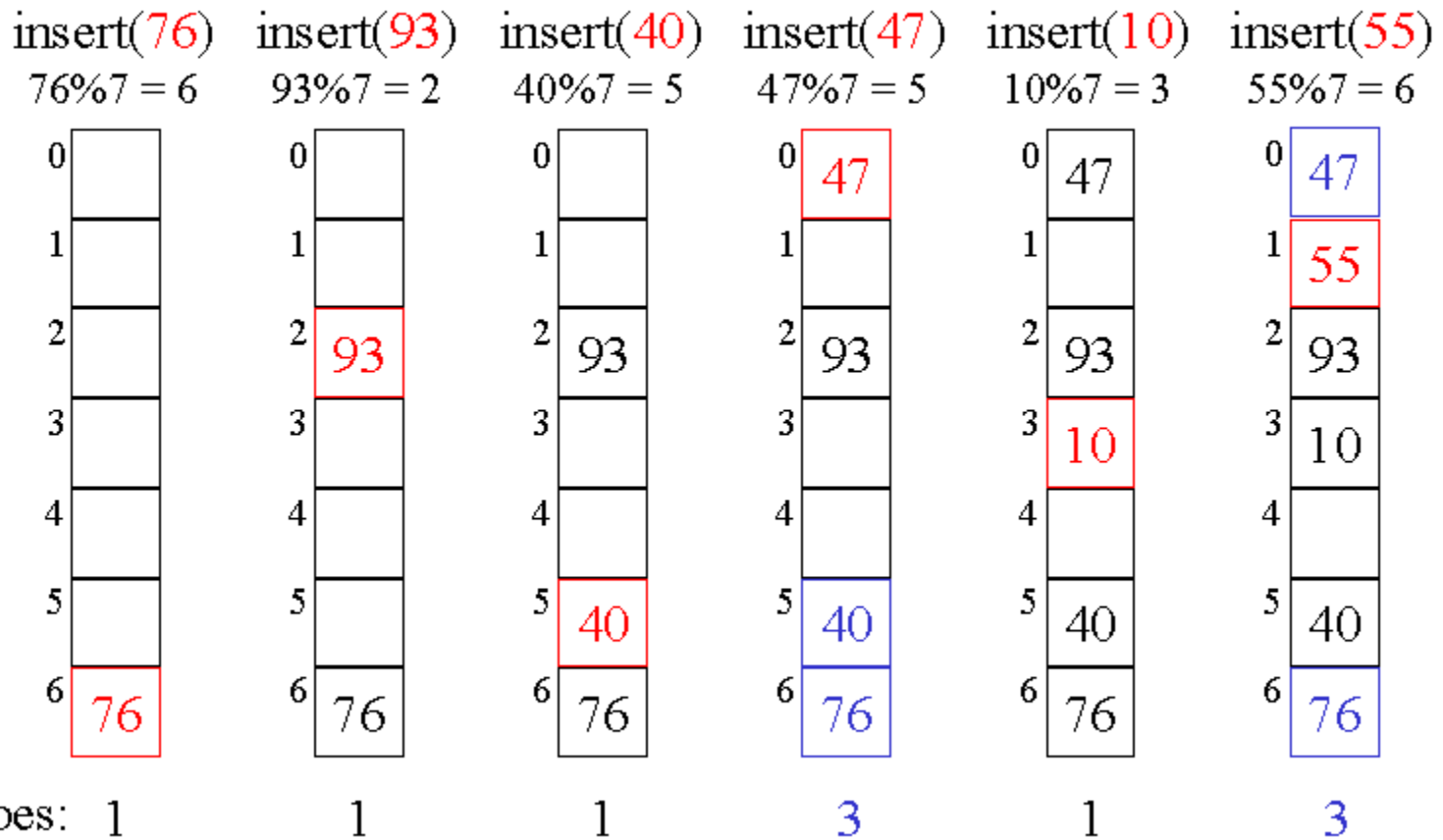
Linear Probing

Advantages of linear probing

1. The linear probing gives the best performance of the

   cache .

2. It can be easily calculated.

Disadvantages of linear probing

1.The main problem is clustering.

2.It takes too much time to find an empty slot.

Quadratic Probing

In quadratic probing,

When collision occurs, we probe for i^2'th bucket in i^{th} iteration.

We keep probing until an empty bucket is found.

$$h'(x) = x \bmod m$$

$$h(x, i) = (h'(x) + i^2) \bmod m$$

Hashing

Steps for Quadratic Probing

Let hash(x) be the slot index computed using the hash function.

- If the slot hash(x) % S is full, then we try

    (hash(x) + 1*1) % S

- If (hash(x) + 1*1) % S is also full, then we try

    (hash(x) + 2*2) % S.

- If (hash(x) + 2*2) % S is also full, then we try

    (hash(x) + 3*3) % S.

- This process is repeated for all the values of i until an empty slot is found.

Quadratic Probing Example

we have a list of size 20 (m = 20). We want to put some elements in linear

probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}

x	h(x, i) = (h'(x) + i²) mod 20
96	i = 0, h(x, 0) = 16
48	i = 0, h(x, 0) = 8
63	i = 0, h(x, 0) = 3
29	i = 0, h(x, 0) = 9
87	i = 0, h(x, 0) = 7
77	i = 0, h(x, 0) = 17
48	i = 0, h(x, 0) = 8 i = 1, h(x, 1) = 9 i = 2, h(x, 2) = 12
65	i = 0, h(x, 0) = 5
69	i = 0, h(x, 0) = 9 i = 1, h(x, 1) = 10
94	i = 0, h(x, 0) = 14
61	i = 0, h(x, 0) = 1

Double Hashing

In double hashing,

• We use another hash function hash2(x) and look for i *

hash2(x) bucket in i^{th} iteration.

• It requires more computation time as two hash functions need

to be computed.

Double hashing can be done using :

(hash1(key) + i * hash2(key)) % TABLE_SIZE

Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table.(We repeat by increasing i when collision occurs)

Double Hashing Example

Example:

Load the keys 18, 26, 35, 9, 64, 47, 96, 36, and 70 in this order,

in an empty hash table of size 13 using double hashing with the

first hash function:

h(key) = key % 13 and

the second hash function:

h2 (key) = PRIME –(key % PRIME)

where PRIME <table size

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 − (key % 7)

Hash1(18)=18%13=5

Hash1(26)=26%13=0

Hash1(35)=35%13=9

Hash1(9)=9%13=9 ——————— Collision

Apply Hash2()

Hash2(9)=7-(9%7)=5

Find new index, repeat until empty

(Hash1(9)+1*Hash2(9))%13=1

Comparison of Open Addressing Techniques-

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Youtube/other Video Links

- https://www.youtube.com/watch?v=zWg7U0OEAoE&list=PLBF3763AF2E1C572F&index=1
- https://www.youtube.com/watch?v=aGjL7YXl31Q&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0
- https://www.youtube.com/watch?v=FEnwM-iDb2g&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0&index=2
- https://www.youtube.com/watch?v=HEjmH9wKiMo&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0&index=6
- https://www.youtube.com/watch?v=sr_bR1WwcLY
- https://www.youtube.com/watch?v=puMz5Jt96sg
- https://www.youtube.com/watch?v=d_XvFOkQz5k&list=PLhb7SOmGNUc5AZurO-im4t_RDr-ymjz0d&index=1
- https://www.youtube.com/watch?v=KELqVT7hjeE&list=PLhb7SOmGNUc5AZurO-im4t_RDr-ymjz0d&index=2
- https://www.youtube.com/watch?v=CZYR2v8rYLA&list=PLhb7SOmGNUc5AZurO-im4t_RDr-ymjz0d&index=3
- https://www.youtube.com/watch?v=-lY4_THb2wM&list=PLhb7SOmGNUc5AZurO-im4t_RDr-ymjz0d&index=4

1. 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

2. What are the disadvantages of arrays?

a) Data structure like queue or stack cannot be implemented

b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size

c) Index value of an array can be negative

d) Elements are sequentially accessed

3. In general, the index of the first element in an array is _____

a) 0

b) -1

c) 2

d) 1

4. If the number of records to be sorted is small, then …… sorting can be efficient.

A. Merge

B. Insertion

C. Selection

D. Bubble

5.Which of the following is not a limitation of binary search algorithm?

A. must use a sorted array

B. requirement of sorted array is expensive when a lot of insertion and deletions are needed

C. there must be a mechanism to access middle element directly

D. binary search algorithm is not efficient when the data elements more than 1500.

6. What is a hash table?

a) A structure that maps values to keys

b) A structure that maps keys to values

c) A structure used for storage

d) A structure used to implement stack and queue

7. If several elements are competing for the same bucket in the hash table, what is it called?

a) Diffusion

b) Replication

c) Collision

d) Duplication

8. What is a hash function?

a) A function has allocated memory to keys

b) A function that computes the location of the key in the array

c) A function that creates an array

d) A function that computes the location of the values in the array

9. ………. is putting an element in the appropriate place in a sorted list yields a larger sorted order list.

A. Insertion

B. Extraction

C. Selection

D. Distribution

10. Which of the following sorting algorithm is of divide and conquer type?

A. Bubble sort

B. Insertion sort

C. Merge sort

D. Selection sort

1.   In ...., search start at the beginning of the list and check every element in the list.
a.   Binary search
b.   Hash Search
c.   Linear search
d.   Binary Tree search.

2. To represent hierarchical relationship between elements, which data structure is suitable?
a.   Graph
b.   Tree
c.   Dequeue
d.   Priority Queue

3. Which of the following data structure is linear type?
a.   Stack
b.   Graph
c.   Trees
d.   Binary tree

4.Which of the following data structure can't store the nonhomogeneous data elements?
a.   Arrays
b.   Stacks
c.   Records
d.   None of the above

5. Two main measures for the efficiency of an algorithm are
a.   Processor and memory
b.   Complexity and capacity
c.   Time and space
d.   Data and space

6. The Worst case occur in linear search algorithm when
a.   Item is somewhere in the middle of the array
b.   Item is not in the array at all
c.   Item is the last element in the array
d.   Item is the last element in the array or is not there at all

7. The complexity of Binary search algorithm is
a. $O(n)$
b. $O(\log_2 n)$
c. $O(n^2)$
d. $O(n \log_2 n)$

8. The complexity of linear search algorithm is
a. $O(n)$
b. $O(\log_2 n)$
c. $O(n^2)$
d. $O(n \log_2 n)$

9. Arrays are best data structures
a. for relatively permanent collections of data
b. for the size of the structure and the data in the structure are constantly changing
c. for both of above situation
d. for none of above situation

10. Linked lists are best suited
a. for relatively permanent collections of data
b. for the size of the structure and the data in the structure are constantly changing
c. for both of above situation
d. for none of above situation

Q1. Distinguish between linear and nonlinear data structures.

Q2. Write a program to insert a new element in the given unsorted array at kth position.

Q3. Define data structure with example.

Q4 . How 2-D arrays are represented in memory? Also obtain the formula for calculating the address of any element stored in 2-D array in case of column-major order.

Q5 Explain Binary search and write a program to perform binary search in python .

Q6 Write the difference between link list and array.

https://drive.google.com/open?id=15fAkaWQ5ccZRZPzwlP4PBh1LxcPp4VAd

References

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser «Data Structures and Algorithms in Python (An Indian Adaptation)" - WILEY Publication.

[2] Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, "Data Structures Using C and C++", PHI Learning Private Limited, Delhi India

[3] Horowitz and Sahani, "Fundamentals of Data Structures", Galgotia Publications Pvt Ltd Delhi India.

[4] Lipschutz, "Data Structures" Schaum's Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.

Thank you