SCS3102 :Data Structures and Algorithms

Prepared by Golooba Ronald

Tel: 0706335306

Email:rgolooba@kyu.ac.ug

What are Data Structures

- A **data structure** is a named location that can be used to store and organize data.
- A data structure is **a specialized format for organizing, processing, retrieving and storing data**.
- There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose.
- Data structures make it easy for users to access and work with the data they need in appropriate ways.
- **An array** is the simplest and most widely used data structure. Other data structures like stacks and queues are derived from arrays.
- **An array is a linear data structure** that collects elements of the same data type and stores them in contiguous and adjacent memory locations.

Terms Used in Data Structures

- I**nterface** An interface, sometimes also called an abstract data type, defines the set of operations supported by a data structure and the semantics, or meaning, of those operations.
- An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations but not how it does the operations.
- **Implementation** − Implementation provides the internal representation of a data structure.
- Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible

Why do we study data structure?

No single data structure works well for all purposes and so it is important to know the strength and limitation of several of them, to help software designer and/or programmer to choose the most appropriate data structure for his or her task.

Problems faced by Software Applications as they get complex and data rich

- **Data Search** − Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 10^6 items every time, slowing down the search. As data grows, search will become slower.
- **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.
- So Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

- There are three cases which are usually used to compare various data structure's execution time in a relative manner.
- **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(n) represents function of n.
- **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then m operations will take mƒ(n) time.
- **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then the actual operation may take time as the random number which would be minimum as ƒ(n)

Basic Terminology

- **Data** − Data are values or set of values.
- **Data Item** − Data item refers to single unit of values.
- **Group Items** − Data items that are divided into sub items are called as Group Items.
- **Elementary Items** − Data items that cannot be divided are called as Elementary Items.
- **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values.
- **Entity Set** − Entities of similar attributes form an entity set.
- **Field** − Field is a single elementary unit of information representing an attribute of an entity.
- **Record** − Record is a collection of field values of a given entity.
- **File** − File is a collection of records of the entities in a given entity set.

What is an algorithm?

- An **algorithm** is a collection of steps to solve a particular problem.
- In otherward an algorithm is **a set of instructions for solving a problem or accomplishing a task**.
- One common example of an algorithm is a recipe, which consists of specific instructions for preparing a dish or meal.
- **Data structures are used to hold data while algorithms are used to solve the problem using that data**.
- Data structures and algorithms (DSA) goes through solutions to standard problems in detail and gives you an insight into how efficient it is to use each one of them.

Algorithms Basics

- **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.
- It is a well defined computational procedure that takes some values as input and produces some value or a set of values as output.
- It is thus a sequence of computational steps that transform the input into output.
- The algorithm describes specific computational procedures for achieving the input/output relationship

Categories of Algorithms.

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

An algorithm should have the following characteristics;

- **Unambiguous** − Algorithm should be clear and instantly recognizable. 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 determinate number of steps.
- **Feasibility** − Should be achievable with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

Is Study of Algorithm worthy while?

- Suppose computer were infinitely fast and memory was free, would you have any reason to study data structures and algorithm?
- It is True because we would like to see our solutions terminating, and with a correct answer. Computers are not infinitely fast and memory is not free as much as it may be cheap. Therefore we need to study Algorithms because;
  1. We need algorithms that use computer resources (memory) wisely
  2. We need to write programs that executes a task with in a shortest time possible.
- In development of software, it is important that both the designer and the programmer know the kind of the problem the computer can solve and how can do it efficiently

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent.
- Algorithms are never written to support a particular programming code. As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc.
- These common constructs can be used to write an algorithm.
- We write algorithms in a step-by-step manner, but it is not always the case.
- Algorithm writing is a process and is executed after the problem domain is well-defined i.e, we should know the problem domain, for which we are designing a solution.

Example

- Problem − Design an algorithm to add two numbers and display the result.
- Step 1 − START
- Step 2 − declare three integers a, b & c
- Step 3 − define values of a & b
- Step 4 − add values of a & b
- Step 5 − store output of step 4 to c
- Step 6 − print c
- Step 7 – STOP
- Algorithms tell the programmers how to code the program.

Algorithm Analysis

- It deals with the execution or running time of various operations involved.
- The running time of an operation can be defined as the number of computer instructions executed per operation.
- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation.
- They are the following −
- **A Prior Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- **A Posterior Analysis** − This is an empirical (experimental) analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

- Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.
  - **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
  - **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.
- The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

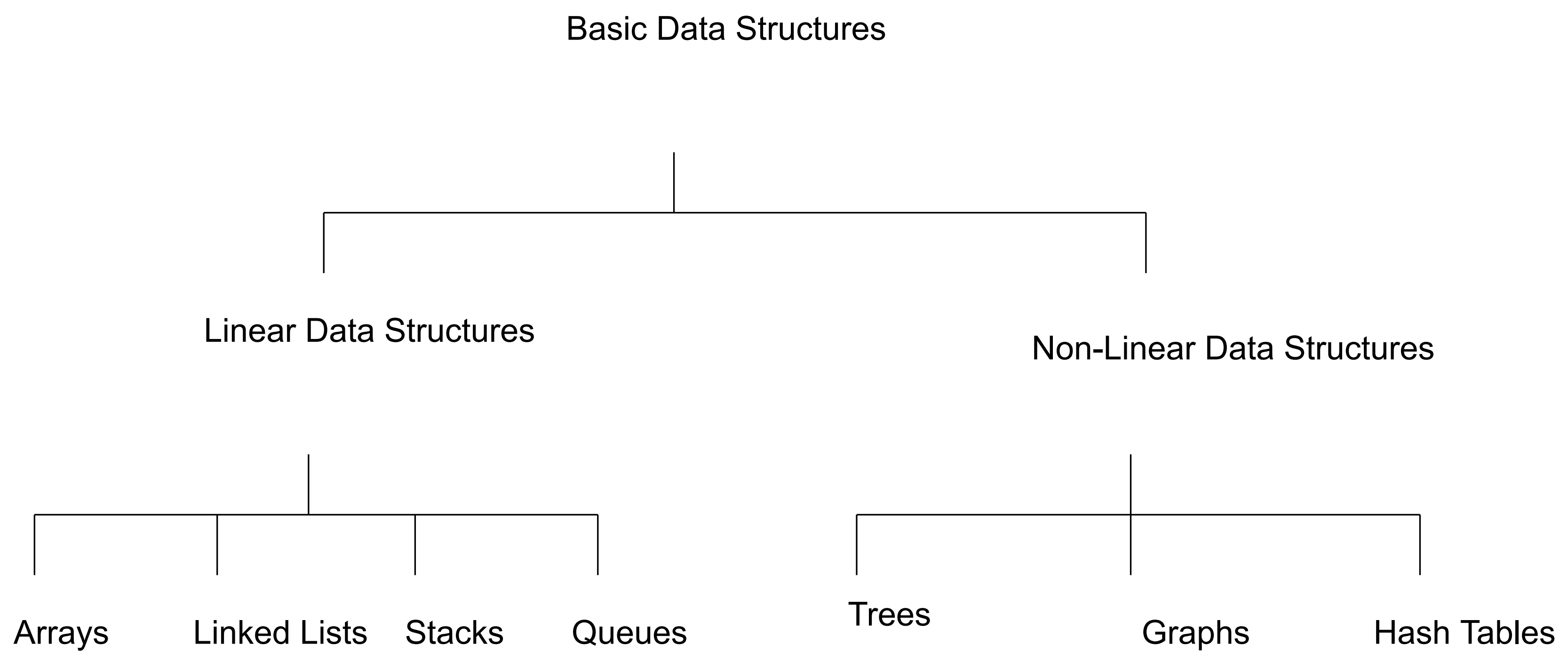
Space Complexity

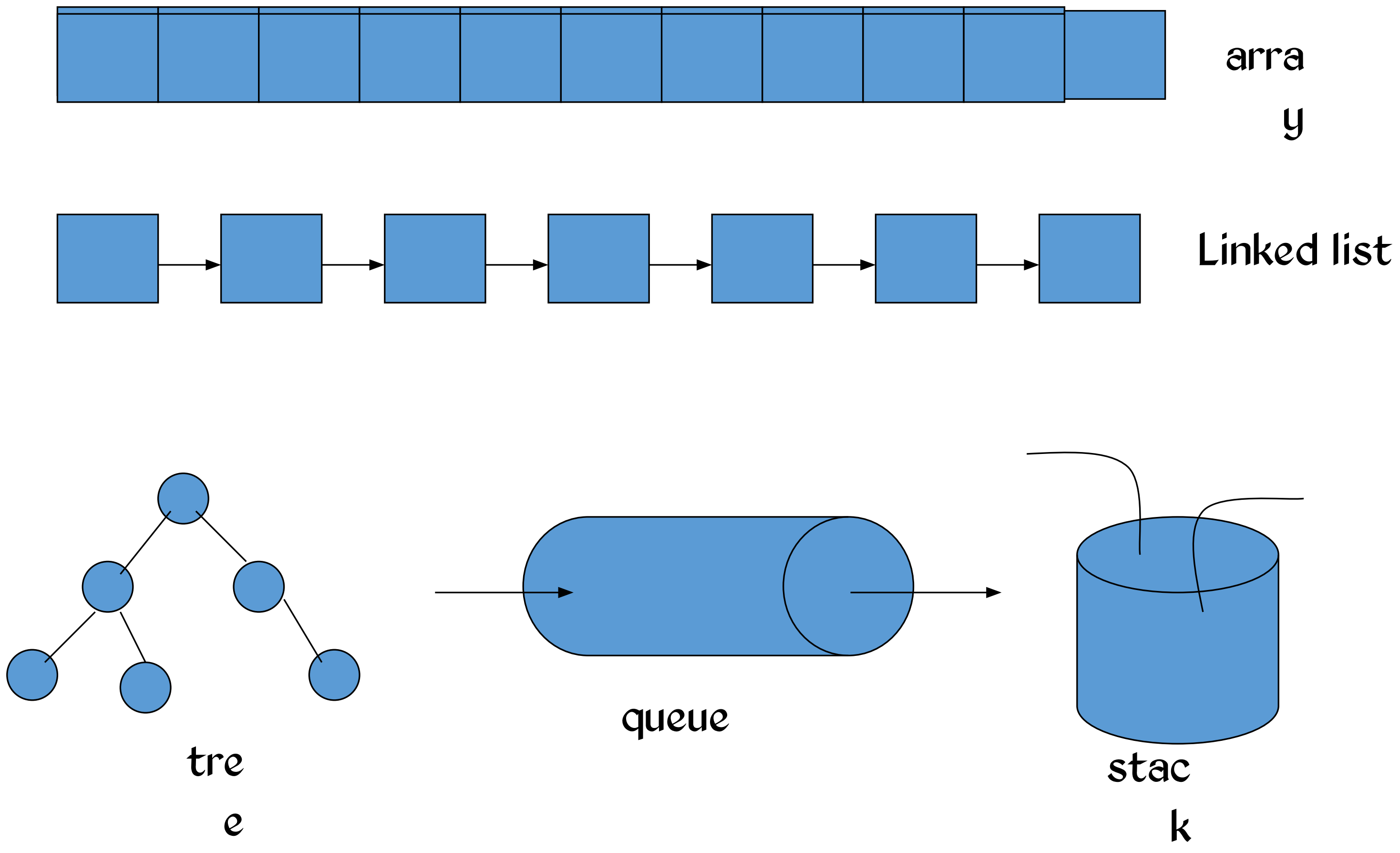
- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.
- The space required by an algorithm is equal to the sum of the following two components;
- **A fixed part** that is a space required to store certain data and variables, that are independent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

Time Complexity

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

Basic Data Structure




arra
y
Linked list
tre
e
queue
stac
k

Selection of Data Structure

- The choice of particular data model depends on two consideration:
    - It must be rich enough in structure to represent the relationship between data elements
    - The structure should be simple enough that one can effectively process the data when necessary.

Types of Data Structure

- Linear: In Linear data structure, values are arrange in linear fashion.
- Linear collection of items are a data structure that displays adjacency relationship in its elements.
- The linear structured data types include batches, records, and arrays while linear abstract data types include linked lists, stacks and queues

Types of Data Structure

- Non-Linear: The data values in this structure are not arranged in order. They are those that do not show adjacency relationship between its elements.
  - Non-linear abstract data types include:
  - THash tables: Unordered lists which use a ‘hash function’ to insert and search
  - Tree: Data is organized in branches.
  - Graph: A more general branching structure, with less strict connection conditions than for a tree
- Elementary and non-elementary data structures include both linear structure data types and linear abstract data types.
- Non elementary data types include non-linear structure data types.

Type of Data Structures

- Homogenous: In this type of data structures, values of the same types of data are stored.
  - Array
- Non-Homogenous: In this type of data structures, data values of different types are grouped and stored.
  - Structures
  - Classes

PRIMITIVE (BUILT-IN/BASIC) DATA TYPES/ STRUCTUER

- These are data types that are typically directed upon by machines level instruction within the computer at their primitive level.
- These include integers, real numbers, character, data Boolean/logical data types

STRUCTURE DATA TYPES (SDTs)

- A SDT is a data type that is constructed by the user from the primitive (built-in/basic) data types.
- These are user-defined data types (typedef, enum, etc) since it is user to determine what structure the data is to have.
- Once the structure has been defined any variable can be defined to be the user defined data type.
- SDTs include: Arrays, record, sets, Batch(es).
- Batches are similar to queued tasks where tasks flow one after another.

Abstract Data Type and Data Structure

- Definition:-
  - *Abstract Data Types (ADTs)* An ADT data type is one in which the axioms (theorems) do not imply the form of the representation (i.e. we know what it does but we don't know how it does whatever it does).
  - An ADT stores data and allow various operations on the data to access and change it.
  - It is a mathematical model, together with various operations defined on the model
  - An ADT is a collection of data and associated operations for manipulating that data
- Data Structures
  - Physical implementation of an ADT
  - data structures used in implementations are provided in a language *(primitive* or *built-in)* or are built from the language constructs *(user-defined)*
  - Each operation associated with the ADT is implemented by one or more subroutines in the implementation.

Abstract Data Type

- ADTs support *abstraction*, *encapsulation*, and *information hiding*.
- *Abstraction* is the structuring of a problem into well-defined entities by defining their data and operations. Abstraction **“displays” only the relevant attributes of objects and “hides” the unnecessary details from the user**. For example, when we are driving a car, we are only concerned about driving the car like start/stop the car, accelerate/ break, etc.
- The principle of hiding the used data structure and to only provide a well-defined interface is known as *encapsulation.*

The Core Operations of ADT

- Every Collection ADT should provide a way to:
  - add an item
  - remove an item
  - find, retrieve, or access an item
- Many, many more possibilities
  - is the collection empty
  - make the collection empty
  - give me a sub set of the collection

  - NB. No single data structure works well for all purposes, and so it is important to know the   strengths and limitations of several of them.

Data Definition

- Data Definition defines a particular data with the following characteristics.
- **Atomic** − Definition should define a single concept.
- **Traceable** − Definition should be able to be mapped to some data element.
- **Accurate** − Definition should be unambiguous.
- **Clear and Concise** − Definition should be understandable.

Data Type

- Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data.
- There are two data types :
  - Built-in Data Type
  - Derived Data Type

Built-in Data Type

- Those data types for which a language has built-in support are known as Built-in Data types.
- For example, most of the languages provide the following built-in data types.
- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

- Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types.
- These data types are normally built by the combination of primary or built-in data types and associated operations on them.
- For example;
  - List
  - Array
  - Stack
  - Queue

Basic Operations

- The data in the data structures are processed by certain operations.
- The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.
- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Linear data structures and their sequential storage representation.
Arrays, Lists and Files

- An array is a data structure that stores/holds data of the same type.
- It is either integer, Boolean, double, float but cannot contain mixed data types.
- It is an ordered set which consists of a fixed number of objects.
- The deletions and insertions are too expensive to carry out for arrays.
- Most of the data structures make use of arrays to implement their algorithms.
- An array is a static data structure, meaning, the size of the array is fixed and cannot be modified as and when needed.
- An array stores it‘s elements in a contiguous/adjacent/connected manner.

List

- List is a collection data type. It allows multiple values to be stored within the same field.
- **A list** is an order set consisting of a variable number of elements to which insertions and deletions can be made.
- Note that operations performed on arrays can also be performed on lists.
- Such operations include combining two or more lists to form another list (i. e. Concatenate); splitting a list into two other lists; copying a list; determining a number of elements in a list; sorting elements in a list; and searching the list for an element which contains a field having a certain value.

i. Each element in a list is composed of one or more fields

ii. A field can be referred to as the smallest piece of information that can be referenced in a programming language.

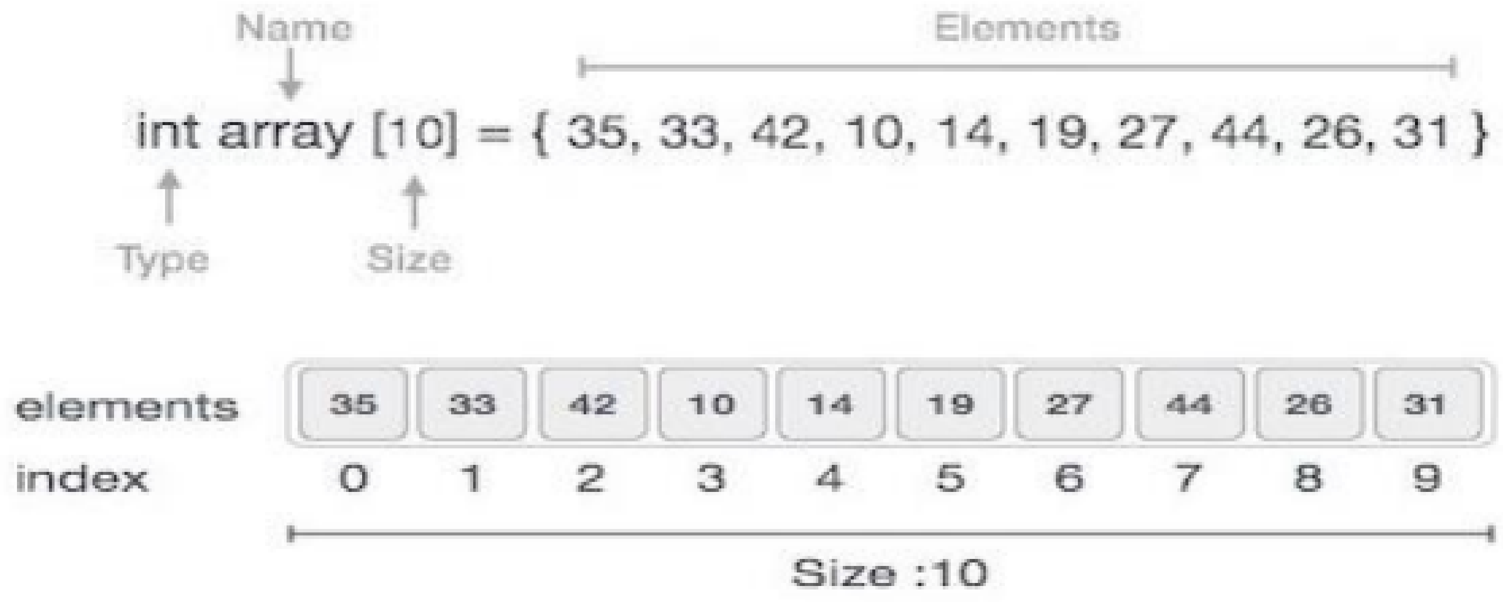
iii. A collection of fields is referred to as a record.

File and Array

- **A file** is typically the largest list that is stored in memory of the computer. It is a collection of records.
- In C implementation, **an array** of size n runs its index from 0, 1, 2, ------, (n-1).
- The following are the important terms to understand the concept of Array.
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

- **Arrays** can be declared in various ways in different languages.
- For illustration purposes, let's take C array declaration and initialization;



Explaining the Array above

- 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.
- Other examples include: - Int A[10] = {1, 2, 3, 4, 5, 6, 8, 9, 12, 13};
- char letter[5] = {‘A’, ‘B’, ‘O’, ’U’ ,’T’}; here an array called letter of size 5 has been created, and initialized to letter[0] to A, letter [1] to B, letter [2] to O, letter [3] to U, letter [4] to T.
- In C, when an array is initialized with size, then it assigns defaults values to its elements in the following order. Boolean = false (f), char = 0, int = 0, float = 0.0 and double = 0.0f

Basic Operations on Arrays

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Sort** - Compares two elements in the array and insert it in the appropriate place.
- **Update** − Updates an element at the given index

Insertion Operation

- It is not always necessary that an element is inserted at the end of an array. The following are the possible operations;
- Insertion at the beginning of an array
- Insertion at the given index of an array
- Insertion after the given index of an array
- Insertion before the given index of an array

Insertion at the Beginning of an Array

- When the insertion happens at the beginning, it causes all the existing data items to shift one step downward.
- Algorithm Assuming A is an array with N elements. The maximum numbers of elements it can store is defined by MAX. We shall first check if an array has any empty space to store any element and then we proceed with the insertion process.
- begin
- IF N = MAX, return //Increasing the size of the array from original N to N+1
- ELSE
- N = N + 1 //Increasing the size of the array from original N to N+1
- ELSE
- For All Elements in A
- Move to next adjacent location
- A[FIRST] = New Element
- end
- Implement this algorithm in python.

How do you insert an element in an array at a given position

4	7	8	9	2	

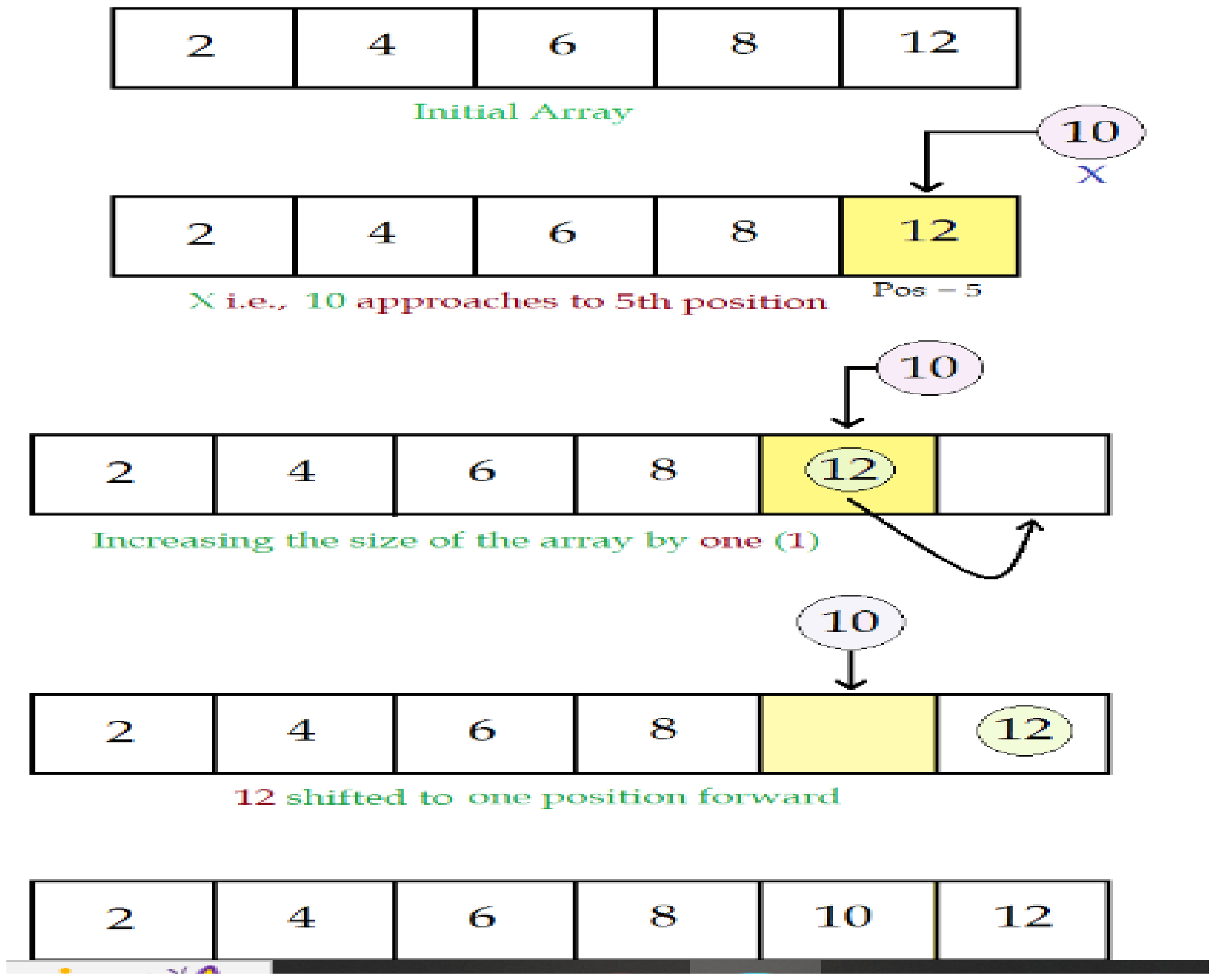
1. arr: Name of the Array.
2. size: size of the array(Total number of elements in the array)
3. i: Loop counter or counter variable for the loop
4. x: the data element to be inserted
5. pos: The position where we wish to insert the element

- The following algorithm inserts a data element x into a given position pos (position specified by the programmer in) in a linear array arr.

```
1.  Start
2.  Size = size +1   //increasing the size of the array by one
3.  i = size -1  //initializing counter variable
4.  i = size -1 to i> pos-1 // repeating steps 5 and 6
5.  arr[i+1] = arr[i]  // moving the $i^{th}$ element forward.
6.  i = i-1 //decrase the counter
7.  End of step 4 loop
8.   arr[pos -1] = x  inserting the element x at position pos -1
9.  End
```

- In the above Algorithm, Step 2 to step 6 creates an empty space in the array by moving forward one location each element from the position on.

Visual Representation


2
4
6
8
12
Initial Array
10
X
Pos – 5
X i.e., 10 approaches to 5th position
Increasing the size of the array by one (1)
12 shifted to one position forward
10

Array with X inserted at posion pos.

```
#include<stdio.h>
#define size 5
int main()  {
   int arr[size] = {1, 20, 5, 78, 30};      int element, pos, i;
   printf("Enter position and element\n");
   scanf("%d%d",&pos,&element);
   if(pos <= size && pos >= 0)   {
 //shift all the elements from the last index to pos by 1 position to right
      for(i = size; i > pos; i--)
         arr[i] = arr[i-1];    //insert element at the given position
      arr[pos] = element;
      /*  print the new array, the new array size will be size+1(actual size+new element)
       * so, use i <= size in for loop  */
      for(i = 0; i <= size; i++)          printf("%d ", arr[i]);      }
   else
      printf("Invalid Position\n");
   return 0;   }
```

Time Complexity Analysis.

- Insert an element at a particular index in an array
- **Worst Case - O(N)**
- If we want to insert an element to index 0, then we need to shift all the elements to right.
- For example, if we have 5 elements in the array and need to insert an element in arr[0], we need to shift all those 5 elements one position to the right.
- In general, if we have **n** elements we need to shift all **n** elements.
- So, worst case time complexity will be **O(n)**. where n = number of elements in the array.

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

Best Case

- If the position value equals to size, then the below for loop will not work as the pos == size.
- for(i=size; i>pos;i--)
- Arr[i] = arr[i-1];
- We can directly place the element in arr[size].
- So, best case time complexity will be **O(1)**.

NB. O(1) indicates that the insertion operation is not depended on the size of the array. Whatever the array size it will always take constant time.

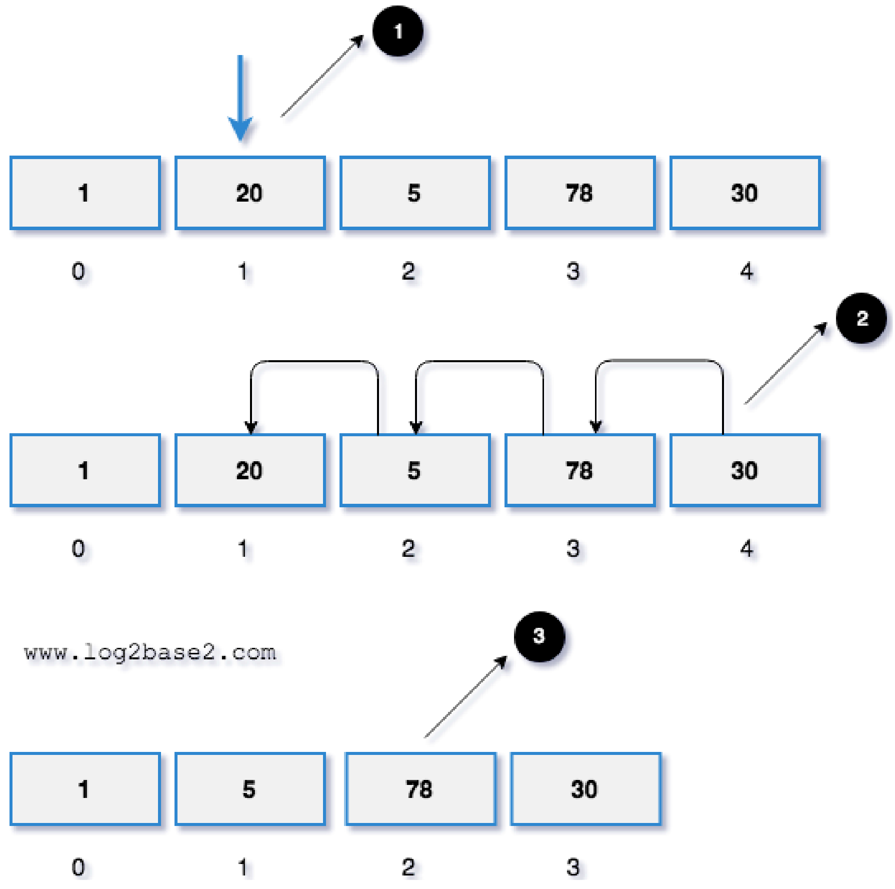
Remove a specific element from array

- **Algorithm**

1. Find the given element in the given array and note the index.
2. If the element is found,
3. Shift all the elements from inde x + 1 by 1 position to the left.
4. Reduce the array size by 1.
5. Otherwise, print "Element Not Found"

Visual Representation

- Let's take an array of 5 elements. {1, 20, 5, 78, 30}. If we remove element 20 from the array, the execution will be,



1. We need to remove the element 20 (index 1) from the array.
2. Shift all the elements from index + 1 (from index 2 to 4) by 1 position to the left.
3. arr[2] (value 5) will be placed in arr[1].
4. arr[3] (value 78) will be placed in arr[2].
5. arr[4] (value 30) will be placed in arr[3].
6. 3. Finally, the new array.

Implementation Algorithm

```
1.  Set **index** value as -1 initially. i.e. **index = -1**
2.  Get **key** value from the user which needs to be deleted.
3.  Search and store the index of a given key
4.  [index will be -1, if the given key is not present in the array]
5.  if index not equal to -1
6.  Shift all the elements from index + 1 by 1 position to the left.
7.  else
8.      print "Element Not Found"
```