VIVEKANANDA COLLEGE
THAKURPUKUR
KOLKATA-700063
NAAC ACCREDITED 'A' GRADE

Topic: Introduction to Data Structure & Algorithm

Course Title: Algorithm and Data Structure

Paper: CMS-A-CC-2-3-TH

Unit: Introduction

Semester: 2nd Semester

Name of the Teacher: Amitava Biswas

Name of the Department: Computer Science

Basic Concepts: Introduction to Data Structures:

A data structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used. The choice of the data structure begins from the choice of an abstract data type (ADT). A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

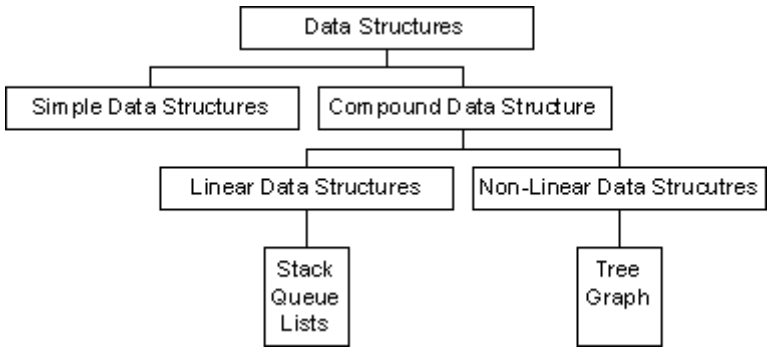
1. First, how the data will be stored, and
2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.

Classification of Data Structures:

Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure



[Fig 1.1 Classification of Data Structures]

Simple Data Structure:

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

Compound Data structure:

Compound 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 data structure
- Non-linear data structure

Linear Data Structure:

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

Operations applied on linear data structure:

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

For example Stack, Queue, Tables, List, and Linked Lists.

Non-linear Data Structure:

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

Operations applied on non-linear data structures:

The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

For example Tree, Decision tree, Graph and Forest

Algorithms:

Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

An algorithm is endowed with the following properties:

1. **Finiteness:** An algorithm must terminate after a finite number of steps.

2. **Definiteness:** The steps of the algorithm must be precisely defined or unambiguously specified.
3. **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
4. **Effectiveness:** the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.
5. **Input-Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

Different Approaches to Design an Algorithm:

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

Practical Algorithm Design Issues:

1. **To save time (Time Complexity):** A program that runs faster is a better program.
2. **To save space (Space Complexity):** A program that saves space over a competing program is considerable and desirable.

Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Analyzing Algorithms

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) with some standard functions. The most common computing times are

$O(1), O(\log_2 n), O(n), O(n \log_2 n), O(n^2), O(n^3), O(2^n)$

Asymptotic Notations:

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size n for large n.

Big oh(O): Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. Here g(n) is the upper bound of the function f(n).

Reasons for analyzing algorithms:

1. To predict the resources that the algorithm requires
   - Computational Time(CPU consumption).
   - Memory Space(RAM consumption).
   - Communication bandwidth consumption.
2. To predict the running time of an algorithm
   - Total number of primitive operations executed.

Recursive Algorithms:

GCD Design: Given two integers a and b, the greatest common divisor is recursively found using the formula

gcd(a,b) = a if b=0 — Base

b if a=0

gcd(b, a mod b) } — General

Fibonacci Design: To start a fibonacci series, we need to know the first two numbers.

Fibonacci(n) = 0 if n=0 — Base

1 if n=1

Fibonacci(n-1) + fibonacci(n-2) } — General case

Difference between Recursion and Iteration:

1. A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
2. Reursion uses stack whereas iteration does not use stack.
3. Recursion uses more memory than iteration as its concept is based on stacks.
4. Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
5. Recursion makes code smaller and iteration makes code longer.
6. Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
7. While using recursion multiple activation records are created on stack for each

call where as in iteration everything is done in one activation record.

Infinite recursion can crash the system where infinite loopi uses CPU repeatedly.