Introduction

A program is said to be efficient when it executes in minimum time and with minimum memory space. In order to write efficient programs we need to apply certain data management concepts.

The concept of data management is a complex task that includes activities like data collection, organization of data into appropriate structures, and developing and maintaining routines for quality assurance.

Data structure is a crucial part of data management. ***A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.***

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

When you will study DBMS as a subject, you will realize that the major data structures used in the Network data model are graphs, Hierarchical data model is trees, and RDBMS is arrays.

Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently. Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design. This is because representing information is fundamental to computer science. The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally categorized into two classes: primitive and non-primitive data structures.

Primitive and Non-primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

Linear and Non-linear Structures

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Arrays

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).

In C, arrays are declared using the following syntax:

type name[size];

For example, int marks[10];

Linked Lists

A linked list is a very flexible, dynamic data structure in which elements (called nodes) form a sequential list. In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance. In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list

The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list. Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available

Stacks

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack. In the computer's memory, stacks can be implemented using arrays or linked lists.

Queues

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front. Like stacks, queues can be implemented by using either arrays or linked lists

Trees

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root. The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree. The root element is the topmost node which is pointed by a 'root' pointer. If root = NULL then the tree is empty.

Graphs

A graph is a non-linear data structure which is a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist. In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions.

OPERATIONS ON DATA STRUCTURES

This section discusses the different operations that can be performed on the various data structures previously mentioned.

- Traversing It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

- Searching It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

- Inserting It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

- Deleting It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

- Sorting Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

- Merging Lists of two sorted data items can be combined to form a single list of sorted data items.

ABSTRACT DATA TYPE

An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job. For example, stacks and queues are perfect examples of an ADT. We can implement both these ADT using an array or a linked list.

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values

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.

Some examples of ADT are Stack, Queue, List etc.

Let us see some operations of those mentioned ADT −

- Stack −
    - isFull(), This is used to check whether stack is full or not
    - isEmpty(), This is used to check whether stack is empty or not
    - push(x), This is used to push x into the stack
    - pop(), This is used to delete one element from top of the stack
    - peek(), This is used to get the top most element of the stack
    - size(), this function is used to get number of elements present into the stack
- Queue −
    - isFull(), This is used to check whether queue is full or not
    - isEmpty(), This is used to check whether queue is empty or not
    - insert(x), This is used to add x into the queue at the rear end
    - delete(), This is used to delete one element from the front end of the queue
    - size(), this function is used to get number of elements present into the queue
- List −
    - size(), this function is used to get number of elements present into the list
    - insert(x), this function is used to insert one element into the list
    - remove(x), this function is used to remove given element from the list
    - get(i), this function is used to get element at position i
    - replace(x, y), this function is used to replace x with y value

ALGORITHMS

The typical definition of algorithm is ‘a formally defined procedure for performing some calculation’. If a procedure is formally defined, then it can be implemented using a formal language, and such a language is known as a programming language. In general terms, an algorithm provides a blueprint to write a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

Algorithms are mainly used to achieve software reuse. Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++, or Java. An algorithm is basically a set of instructions that solve a problem. It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

DIFFERENT APPROACHES TO DESIGNING AN ALGORITHM

Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.

A complex algorithm is often divided into smaller units called modules. This process of dividing an algorithm into modules is called modularization. The key advantages of modularization are as Follows:

- It makes the complex algorithm simpler to design and implement.
- Each module can be designed independently. While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.

There are two main approaches to design an algorithm—top-down approach and bottom-up Approach:

Top-down approach A top-down design approach starts by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved. Top-down design method is a form of stepwise refinement where we begin with the topmost module and incrementally add modules that it calls.

Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

Bottom-up approach A bottom-up approach is just the reverse of top-down approach. In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules. The higher level modules are implemented by using the operations performed by lower level modules. Thus, in this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algorithm is obtained.

CONTROL STRUCTURES USED IN ALGORITHMS

An algorithm has a finite number of steps. Some steps may involve decision-making and repetition. Broadly speaking, an algorithm may employ one of the following control structures: (a) sequence, (b) decision, and (c) repetition.

Sequence

By sequence, we mean that each step of an algorithm is executed in a specified order.

Decision

Decision statements are used when the execution of a process depends on the outcome of some condition.

Repetition

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as while , do–while , and for loops.

TIME AND SPACE COMPLEXITY

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity. The time complexity of an algorithm is basically the running time of a program as a function of the input size. Similarly, the space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size. In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

Generally, the space needed by a program depends on the following two parts:

- Fixed part: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- Variable part: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

Worst-case, Average-case, Best-case

Worst-case running time This denotes the behaviour of an algorithm with respect to the worst-possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time The term 'best-case performance' is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

Time–Space Trade-off

The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution. But practically, designing such an ideal algorithm is not a trivial task. There can be more than one algorithm to solve a particular problem. One may require less memory space, while the other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for the other. Hence, there exists a time–space trade-off among algorithms.

So, if space is a big constraint, then one might choose a program that takes less space at the cost of \ more CPU time. On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

Expressing Time and Space Complexity

The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved. Expressing the complexity is required when:

- We want to predict the rate of growth of complexity as the input size of the problem increases.
- There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function f(n) is the Big O notation. It provides the upper bound for the complexity.

Algorithm Efficiency

If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains. However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

Let us consider different cases in which loops determine the efficiency of an algorithm.

Linear Loops

To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for(i=0;i<100;i++)
statement block;
```

Here, 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

$f(n) = n$

However calculating efficiency is not as simple as is shown in the above example. Consider the loop given below:

```
for(i=0;i<100;i+=2)
statement block;
```

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

$f(n) = n/2$

Logarithmic Loops

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, in logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)
statement block;
```

Or,

```
for(i=1000;i>=1;i/=2)
statement block;
```

Consider the first for loop in which the loop-controlling variable i is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of i doubles. Now, consider the second loop in which the loop-controlling variable i is divided by 2. In this case also, the loop will be executed 10 times. Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied. In the examples discussed, it is 2. That is, when n = 1000 , the number of iterations can be given by log 1000 which is approximately equal to 10. Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as:

f(n) = log n

Nested Loops

Loops that contain loops are known as nested loops. In order to analyse nested loops, we need to determine the number of iterations each loop completes. The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.
In this case, we analyse the efficiency of the algorithm based on whether it is a linear logarithmic, quadratic, or dependent quadratic nested loop.

Linear logarithmic loop Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is log 10 . This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according to the formula, the number of iterations for this code can be given as 10 log 10 .

```
for(i=0;i<10;i++)
for(j=1; j<10;j*=2)

statement block;
```

In more general terms, the efficiency of such loops can be given as f(n) = n log n .

Quadratic loop In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times.
Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
for(j=0; j<10;j++)

statement block;
```

The generalized formula for quadratic loop can be given as f(n) = n^2 .

Dependent quadratic loop In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
for(j=0; j<=i;j++)

statement block;
```

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as:

$1 + 2 + 3 + ... + 9 + 10 = 55$

If we calculate the average of this loop ($55/10 = 5.5$), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates $(n + 1)/2$ times. Therefore, the efficiency of such a code can be given as:

$f(n) = n (n + 1)/2$