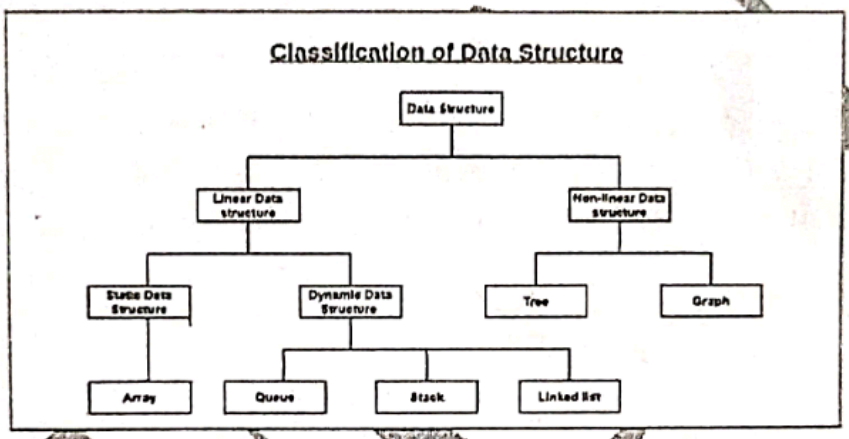
Module – I

Introduction

Definition and importance of linear data structures, Abstract Data Types (ADTs) and their implementation), Overview of Time and Space complexity, Analysis for linear data structures.

Data Structures:

A 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. A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data.



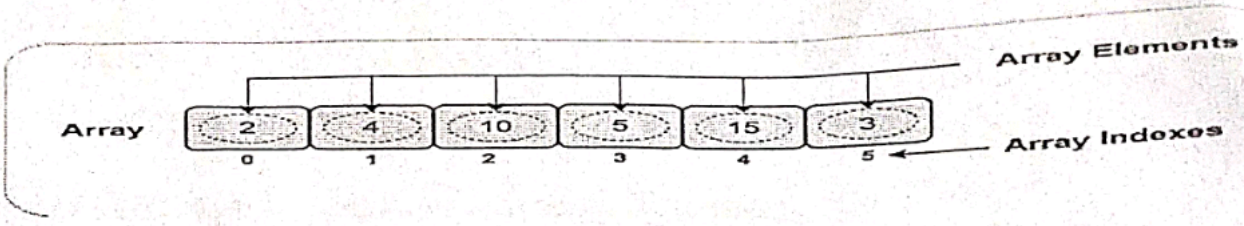
Linear data structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
Examples of linear data structures are array, stack, queue, linked list, etc.

Static data structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
An example of this data structure is an array.

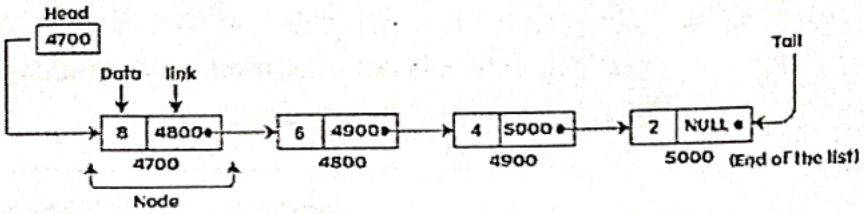
Dynamic data structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
Examples of this data structure are queue, stack, etc.

Non-linear data structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
Examples of non-linear data structures are trees and graphs.

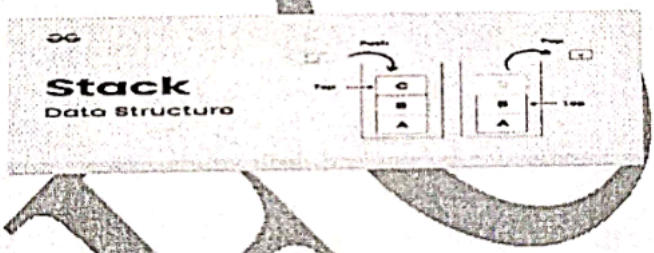
Array: The representation of an array can be defined by its declaration. A declaration means allocating memory for an array of a given size.



Linked list: Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null.

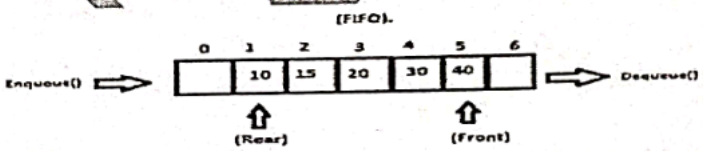


Stack: Stacks in Data Structures is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure, that is top.



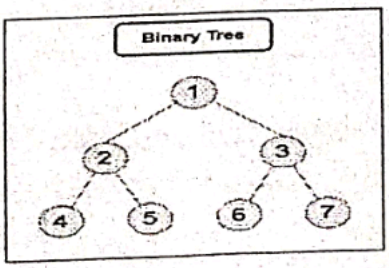
Queue:

A queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

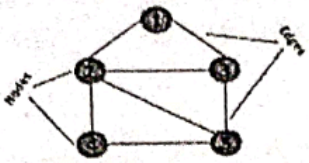


Tree:

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges", both directed and undirected.



Graph: A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).



Abstract Data Type (ADT): An abstract data type (ADT) is a mathematical model for data types, defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. Common examples include lists, stacks, sets, etc.

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- Abstraction: The user does not need to know the implementation of the data structure only essentials are provided.
- Better Conceptualization: ADT gives us a better conceptualization of the real world.
- Robust: The program is robust and has the ability to catch errors.
- Encapsulation: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- Data Abstraction: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- Data Structure Independence: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- Information Hiding: ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- Modularity: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Advantages:

- Encapsulation: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- Abstraction: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- Data Structure Independence: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- Information Hiding: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
- Modularity: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

- Overhead: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- Complexity: ADTs can be complex to implement, especially for large and complex data structures.
- Learning Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- Limited Flexibility: Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- Cost: Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Time and Space Complexity:

Time Complexity

The time required by the algorithm to solve given problem is called *time complexity* of the algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Space Complexity:

The amount of memory required by the algorithm to solve given problem is called space complexity of the algorithm.

To estimate the memory requirement we need to focus on two parts:

(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

Ex1:

```
#include <stdio.h>
int main()
{
        printf("Hello World");
        return 0;
}
```

Time Complexity: In the above code "Hello World" is printed only once on the screen.
So, the time complexity is constant: O(1).

Ex2:

```
#include <stdio.h>
void main()
{
        int i, n = 8;
        for (i = 1; i <= n; i++) {
                printf("Hello World !!!\n");
        }
}
```

Time Complexity: In the above code "Hello World !!!" is printed only n times on the screen, as the value of n can change. So, the time complexity is linear: O(n).

Example : Addition of two scalar variables

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <— A+B
return C
```

The addition of two scalar numbers requires one extra memory location to hold the result. Thus the space complexity of this algorithm is constant, hence S(n) = O(1).