
RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Data Structure**

Subject Code: **IT-303**

Semester: **3rd**

**Subject Notes**
**IT 302- Data Structure**

**Unit-1**

**1.1 Introduction Data**

**1.1.1 Data and Data Item**

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of value. Data items that are divided into sub items are group items; those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.

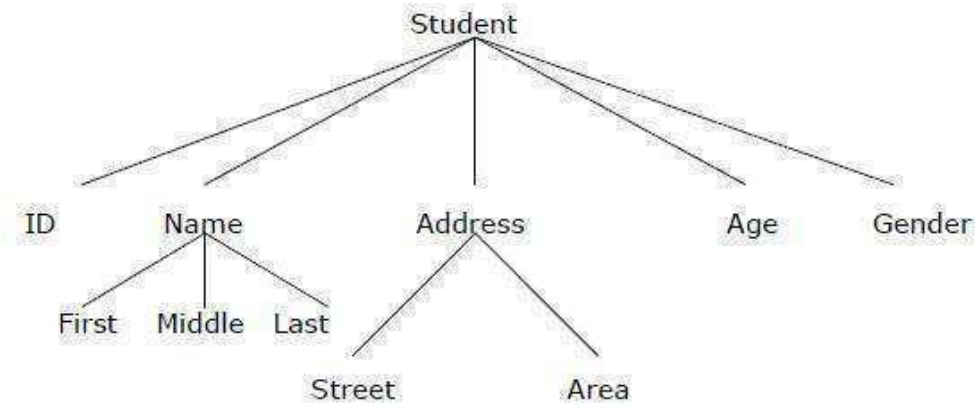


**Fig 1.1 Example of Data item**

In the above example ( ID, Age, Gender, First, Middle, Last, Street, Area ) are elementary data items, whereas (Name, Address ) are group data items.

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.

**1.2 Data Type**

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored.

**1.3 Data Object**

A data object is a region of storage that contains a value or group of values. Each value can be accessed using its identifier or a more complex expression that refers to the object. In addition, each object has a unique data type. The data type of an object determines the storage allocation for that object and the interpretation of the values during subsequent access. It is also used in any type checking operations. Both the identifier and data type of an object are established in the object declaration.
An instance of a class type is commonly called a class object. The individual class members are also called objects.

**1.4 Types of Data Structures:**

A data structure can be broadly classified into 2 types:(i) Primitive data structure
(ii) Non-primitive data structure

**1.4.1 Primitive data structure**

The data structures that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double ,char are known as primitive data structures. It is the basic data type that is provided by the programming language with built -in support. This data type is native to the language and is supported by machine directly.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

**1.4.2 Non-primitive data structure**

The data structures, which are not primitive i.e which can be derived from primitive are called non-primitive data structures. There are two types of non-primitive data structures.

1.Linear Data Structures
2.Non Linear Data Structures

**1.4.3 Linear Data Structures**:- In linear data structures, elements are arranged in linear fashion. The linear ordering is maintained either through consecutive memory locations or by means of pointers .Arrays, linked lists, stacks and queues are examples of linear data structures in which values are stored in a sequence. Eg: Stacks , Queues, Lists, Arrays.

There are basically two ways of representing such linear structure in memory.

**a)** One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays**.**

**b)** The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists**.**

The common examples of linear data structure are arrays, queues, stacks and linked lists.

**Array:** Array is a collection of data of same data type stored in consecutive memory location and is referred by common name**.**

**Linked list:** Linked list is a collection of data of same data type but the data items need not be stored in consecutive memory locations.

**Stack:** A stack is a Last-In-First-Out linear data structure in which insertion and deletion takes place at only one end called the top of the stack.

**Queue:** A Queue is a First in First-Out Linear data structure in which insertions takes place one end called the rear and the deletions takes place at one end called the Front.

**1.4.4 Non-linear Data Structure**:- Elements are stored based on the hierarchical relationship among the data. The data values in this structure are not arranged in order. Tree, graph, table and sets are examples of non-linear data structures.

The following are some of the Non-Linear data structure:

**Trees:** Trees are used to represent data that has some hierarchical relationship among the data elements.

**Graph:** Graph is used to represent data that has relationship between pair of elements not necessarily hierarchical in nature. For example electrical and communication networks, airline routes, flow chart, graphs for planning projects

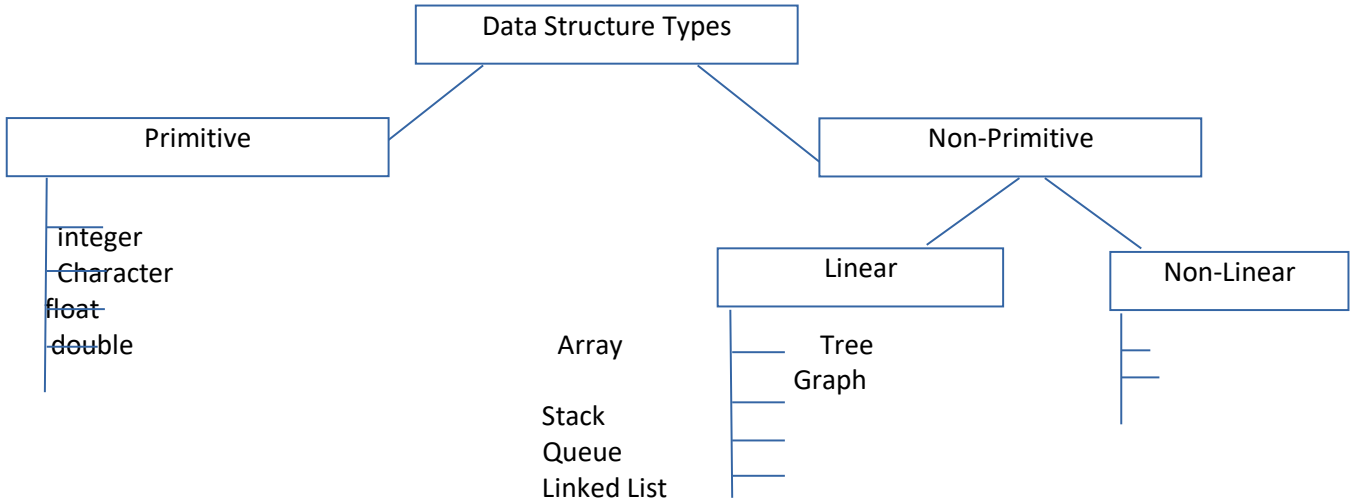


**Fig 1.2 Classification of Data Structure**

**The data structures can also be classified on the basis of the following characteristics:**

| Characteristic | Description |
|---|---|
| Linear | In Linear data structures,the data items are arranged in a linear sequence. Example: **Array** |
| Non-Linear | In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph** |
| Homogeneous | In homogeneous data structures,all the elements are of same type. Example: **Array** |
| Non-Homogeneous | In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures** |
| Static | Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array** |
| Dynamic | Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers** |

**1.5 Operation On Data Structures: -**

The four major operations performed on data structures are:

(i) Insertion: - Insertion means adding new details or new node into the data structure.

(ii)Deletion: - Deletion means removing a node from the data structure.

(iii)Traversal:- Traversing means accessing each node exactly once so that the nodes of a data structure can be processed. Traversing is also called as visiting.

(iv)Searching: -Searching means finding the location of node for a given key value.

Apart from the four operations mentioned above, there are two more operations occasionally performed on data structures. They are:

(v) Sorting: - Sorting means arranging the data in a particular order.

(vi)Merging: -Merging means joining two lists.

**Algorithm**

A well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output. It can also be defined as sequence of computational steps that transform the input into the output.

An algorithm can be expressed in three ways:-

1. in any natural language such as English.
2. in pseudo code or
3. in the form of a flowchart.

**1.6 Complexity analysis**

Algorithmic efficiency are the properties of an algorithm which relate to the amount of resources used by the algorithm. An algorithm must be analyzed to determine its resource usage. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process. For maximum efficiency we wish to minimize resource usage. However, the various resources (e.g. time, space) can not be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is being considered as the most important, e.g. is the requirement for high speed, or for minimum memory usage, or for some other measure. It can be of various types:

- Worst case efficiency: It is the maximum number of steps that an algorithm can take for any collection of data values.

- Best case efficiency: It is the minimum number of steps that an algorithm can take any collection of data values.

- Average case efficiency: It can be defined as the efficiency averaged on all possible inputs.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.
Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

**1.7 Time Space Trade-off**

The best algorithm to solve a given problem is one that requires less memory space and less time to run to completion. But in practice, it is not always possible to obtain both of these objectives. One algorithm may require less memory space but may take more time to complete its execution. On the other hand, the other algorithm may require more memory space but may take less time to run to completion. Thus, we have to sacrifice one at the cost of other. In other words, there is Space-Time trade-off between algorithms.
If we need an algorithm that requires less memory space, then we choose the first algorithm at the cost of more execution time. On the other hand if we need an algorithm that requires less time for execution, then we choose the second algorithm at the cost of more memory space.

**1.8 Algorithm Efficiency**

**Time Complexity:** Time complexity of an algorithm is the amount of time it needs in order to run to completion. It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.
**Space Complexity:** Space Complexity of an algorithm is the amount of space it needs in

order to run to completion. It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this. Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

**1.9 Asymptotic Notations**

**1.9.1 Asymptotic**

It means a line that continually approaches a given curve but does not meet it at any finite distance.

Example: x is asymptotic with x + 1 as shown in graph.
Asymptotic may also be defined as a way to describe the behavior of functions in the limit or without bounds.
Let f(x) and g(x) be functions from the set of real numbers to the set of real numbers. We say that f and g are asymptotic and write f(x) ≈ g(x) if

$\lim_{x \to \infty}$ f(x) / g(x) = c (constant)

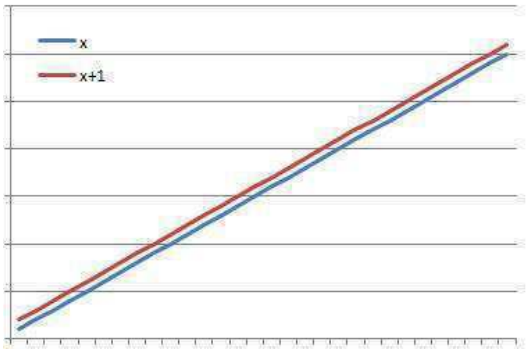


**Fig 1.3 asymptotic bound**

**1.9.2 Big-Oh Notation (O)**

The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

Definition:A function f(n) is said to be in O(g(n)), denoted f(n) ∈ O(g(n)), if f(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that f(n) ≤ cg(n) for all n ≥ n0

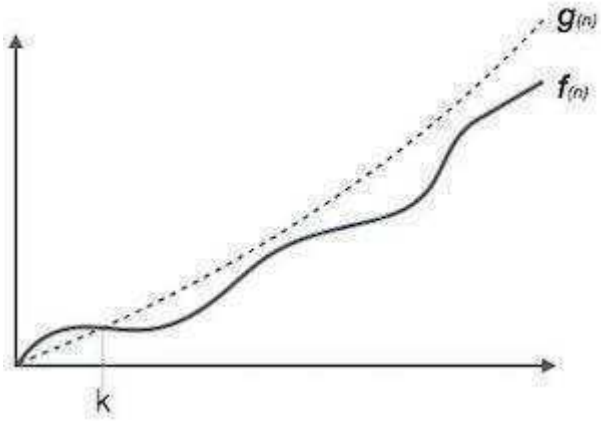


**Fig 1.4 Big-Oh Notation**

For example, for a function *f*(n) O(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

**1.9.3 Omega Notation, Ω**

The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

Definition:A function f (n) is said to be in Ω (g(n)), denoted f(n) ∈ Ω (g (n)), if t (n) is bounded below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that f(n) ≥ cg(n) for all n ≥ n0

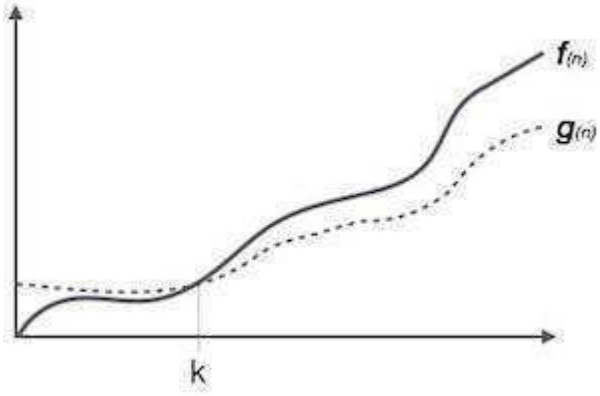


**Fig 1.5 Big-Omega Notation**

For example, for a function *f*(n)Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

**1.9.4 Theta Notation, θ**

The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time.

Definition:A function f(n) is said to be in Θ(g (n)), denoted t(n) ∈ Θ (g (n)), if f(n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such thatc2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0. It is represented as following θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

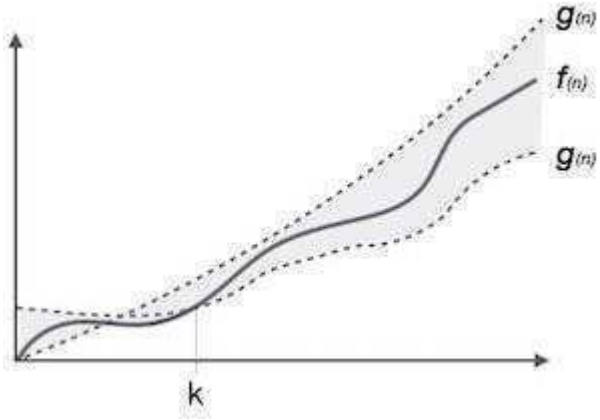


**Fig 1.5 Big-Theta Notation**


RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in