Data Structures [R1UC308B]

Module-I: Introduction

Dr. Subhash Chandra Gupta

School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

July 31, 2025

Contents

Basic Terminology

What is Data Structure?
- A data structure is a particular way of organising data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.
- The choice of a good data structure makes it possible to perform a variety of critical operations effectively.
- An efficient data structure also uses minimum memory space and execution time to process the structure.
- A data structure is not only used for organising the data. It is also used for processing, retrieving, and storing data.

- **Data**: Data are simply values or sets of values.
- **Data items**: Data items refers to a single unit of values.

- **Group items**: Data items that are divided into sub-items are called Group items. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name.
- **Elementary items**: Data items that are not able to divide into sub-items are called Elementary items. Ex: EnRollNo.
- **Entity**: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.
- Entities with similar attributes form an **entity set**.
- Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.
- The term **information** is sometimes used for data with given attributes, in other words meaningful or processed data.

- **Field** is a single elementary unit of information representing an attribute of an entity.
- **Record** is the collection of field values of a given entity.
- **File** is the collection of records of the entities in a given entity set.

Need of Data Structure:
The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand to the developer, as well as the user, can make an efficient implementation of the operation.
Data structures provide an easy way of organising, retrieving, managing, and storing data. Here is a list of the needs for data structure.

- Data structure modification is easy.
- It requires less time.

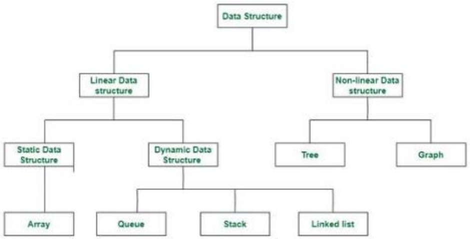
- Save storage memory space.
- Data representation is easy.
- Easy access to the large database

Classification/Types of Data Structures

1. Linear Data Structure
2. Non-Linear Data Structure

Linear Data Structure:
Linear data structures: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). Examples: Linked Lists, Stacks and Queues.
Non-Linear Data Structure:
Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and graphs.

Classification of Data Structure



Applications of Data Structures

Data structures are used in various fields such as:

- Operating system
- Graphics
- Computer Design
- Blockchain
- Genetics
- Image Processing
- Simulation
- ...

Algorithm

- What is an algorithm?
  - An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
  - An algorithm is a sequence of computational steps that transform the input into output.
  - An algorithm is a sequence of operations performed on data that have to be organized in data structures.
  - A finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems.
  - An algorithm is an abstraction of a program to be executed on a physical machine.

- *An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.*

- Why do we study algorithms?
  - To make solution more faster.
  - To compare performance as a function of input size.

Two main property of algorithm is:

1. **Correctness**: Does the algorithm give solution to the problem in a finite number of steps?
2. **Efficiency**: How much resources in terms of memory and time, does it take to execute the algorithm.

Efficiency of an algorithm

- To go from city “A” to city “B”, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle.
- Depending on the availability, convenience, and affordability etc., we choose the one that suits us.
- Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more).
- Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.
- The goal of the analysis of algorithms is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors e.g., memory, developer effort, etc.

Time-space trade-off and complexity

A tradeoff is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either in less time and by using more space
- In very little space by spending a long amount of time.

The best Algorithm is that which helps to solve a problem that requires less space in memory and also takes less time to generate the output.
- But in general, it is not always possible to achieve both of these conditions at the same time.
- The most common condition is an algorithm using a lookup table.
- This means that the answers to some questions for every possible value can be written down.

- One way of solving this problem is to write down the entire lookup table, which will let you find answers very quickly but will use a lot of space.
- Another way is to calculate the answers without writing down anything, which uses very little space, but might take a long time.
- Therefore, the more time-efficient algorithms you have, that would be less space-efficient.

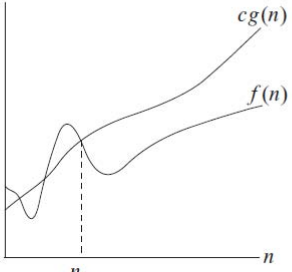
Types of Space-Time Trade-off:

- Compressed or Uncompressed data: A space-time trade-off can be applied to the problem of data storage. If data stored is uncompressed, it takes more space but less time. But if the data is stored compressed, it takes less space but more time to run the decompression algorithm. There are many instances where it is possible to directly work with compressed data. In that case of compressed bitmap indices, where it is faster to work with compression than without compression.
- Re-Rendering or Stored images: In this case, storing only the source and rendering it as an image would take more space but less time i.e., storing an image in the cache is faster than re-rendering but requires more space in memory.

- Smaller code or loop unrolling: Smaller code occupies less space in memory but it requires high computation time that is required for jumping back to the beginning of the loop at the end of each iteration. Loop unrolling can optimize execution speed at the cost of increased binary size. It occupies more space in memory but requires less computation time.
- Lookup tables or Recalculation: In a lookup table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed. It can recalculate i.e., compute table entries as needed, increasing computing time but reducing memory requirements.

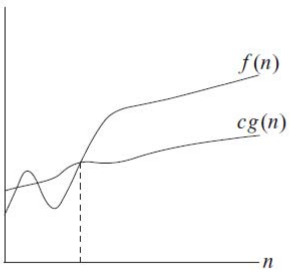
Asymptotic notations

1. O - notation "Big O" : Asymptotic upper bound,
   $O(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
   $f(n) \in O(g(n))$
   $f(n) = O(g(n))$



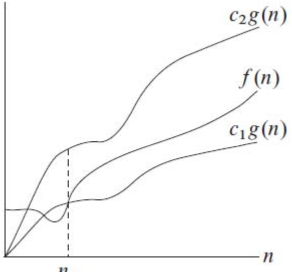
$f(n) = O(g(n))$

2. Ω - notation "Big omega" : Asymptotic lower bound,
   $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$
   $f(n) \in \Omega(g(n))$
   $f(n) = \Omega(g(n))$



$f(n) = \Omega(g(n))$

3. Θ - notation : Asymptotic tight bound,
   $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
   $f(n) \in \Theta(g(n))$
   $f(n) = \Theta(g(n))$



$f(n) = \Theta(g(n))$

4. o - notation "small o": Asymptotic loose upper bound, $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$
5. ω - notation "small omega": Asymptotic loose lower bound, $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

Benefits of Asymptotic Notations:

- Simple representation of algorithm efficiency.
- Easy comparison of performance of algorithms.

Complexity analysis

Analyzing an algorithm means predicting the resources that the algorithm requires. Resources may be memory, communication bandwidth, computer hardware or CPU time. Our primary concern is to measures the computational time required for the algorithm.
Running time:-The running time of an algorithm is the number of primitive operations or steps executed on a particular input.
Why do we normally concentrate on finding only the worst-case running time?

1. The worst-case running time of an algorithm gives us an upper bound on the running time for any input. So it guarantees that the algorithm will never slower than this. In real applications, worst case normally occurs for example searching a non existing data.

2. Best case is like an ideal case which guarantees that the algorithm will never faster than stated. Based upon this we can't allocate the resources.
3. Average case normally perform as worst case because normally we take average case as average of best and worst or best for half size input and worst for other half size.

Complexity analysis: Insertion sort

Insertion-Sort(A,N)	Cost	Times
1. for $j = 2$ to N	c_1	n
2. $key = A[j]$	c_2	n-1
Insert $A[j]$ in sorted $A[1]$ to $A[j-1]$		
3. $i = j - 1$	c_3	n-1
4. while $i > 0$ and $A[i] > key$	c_4	$\sum_{j=2}^{n} t_j$
5. $A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
6. $i = i - 1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
while-end		
7. $A[i+1] = key$	c_7	n-1
for-end		

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1)$$
$$+ c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

$$\Rightarrow T(n) = an + b + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1)$$

Now consider different cases:

1. **Best Case:** The algorithm performs best if $key \leq A[i]$ for every value of j in step 4.
   Then it executes only once for each value of j and total of n-1 times.
   Step 5 and 6 will not be execute at all.
   This is the case when array is already sorted

$$T(n) = an + b = O(n)$$

2. **Worst Case:** The algorithm performs worst if $key > A[i]$ for each value of j and stops only when $i < 1$ in step 4.
   Then it will execute always j times for each value of
   $j = 2, 3, \dots, n$
   so

   $$\sum_{j=2}^{n} j = \frac{(n-1)(2+n)}{2}$$

   and step 5 and 6 will execute

   $$\sum_{j=2}^{n} (j-1) = \frac{(n-1)n}{2}$$

   This is the case when array is already sorted in reverse order

   $$T(n) = an^2 + bn + c = O(n^2)$$

Thank you

Please send your feedback or any queries to subhash.chadra@galgotiasuniversity.edu.in