Last Moment Tuitions

Introduction to Data Structures & Algorithm

Prepare for CDAC CCAT Exam with LMT Study Materials

Get Complete Study Notes, Practice Problems and Previous Year Question Papers, Syllabus and More

Section [A]:-

https://lastmomenttuitions.com/course/ccat-study-material-section-a/

Section [A+B]:-

http://lastmomenttuitions.com/course/ccat-study-material-section-asection-b/

Telegram Group: https://t.me/LMTCDAC

Data structures & Algorithm Notes :
https://lastmomenttuitions.com/course/data-structures-and-algorithms-notes/

Everything you need for CCAT Preparation

http://lastmomenttuitions.com/cdac

1.Data Structures & Algorithms

★Data Structures

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

Why are data structures important?

Typical base data types, such as integers or floating-point values, that are available in most computer programming languages are generally insufficient to capture the logical intent for data processing and use. Yet applications that ingest, manipulate and produce information must understand how data should be organized to simplify processing. Data structures bring together the data elements in a logical way and facilitate the effective use, persistence and sharing of data. They provide a formal model that describes the way the data elements are organized.

Some examples of how data structures are used include the following:

- **Storing data.** Data structures are used for efficient data persistence, such as specifying the collection of attributes and corresponding structures used to store records in a database management system.
- **Managing resources and services.** Core operating system (OS) resources and services are enabled through the use of data structures such as linked lists for memory allocation, file directory management and file structure trees, as well as process scheduling queues.
- **Data exchange.** Data structures define the organization of information shared between applications, such as TCP/IP packets.
- **Ordering and sorting.** Data structures such as binary search trees -- also known as an ordered or sorted binary tree -- provide efficient methods of sorting objects, such as character strings used as tags. With data structures such as priority queues, programmers can manage items organized according to a specific priority.

- **Indexing**. Even more sophisticated data structures such as B-trees are used to index objects, such as those stored in a database.
- **Searching.** Indexes created using binary search trees, B-trees or hash tables speed the ability to find a specific sought-after item.
- **Scalability.** Big data applications use data structures for allocating and managing data storage across distributed storage locations, ensuring scalability and performance.

Types of Data Structure

Basically, data structures are divided into two categories:

- Linear data structure
- Non-linear data structure

Let's learn about each type in detail.

Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

Popular linear data structures are:

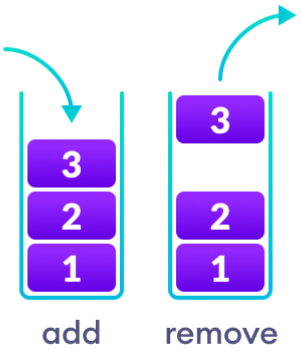
1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

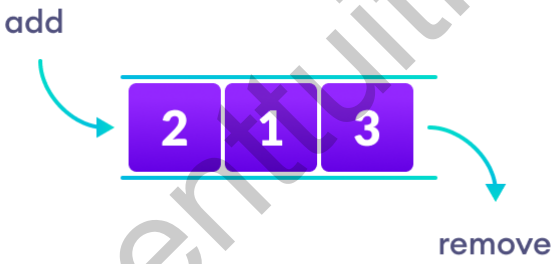
It works just like a pile of plates where the last plate kept on the pile will be removed first.



3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.



4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

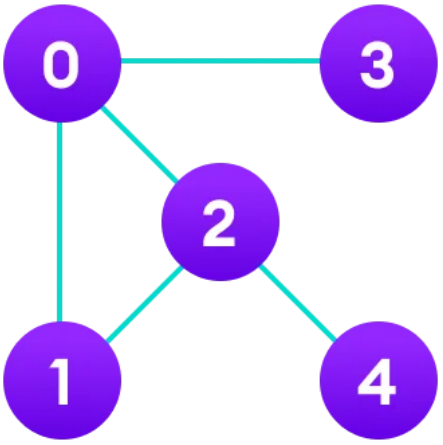
Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.

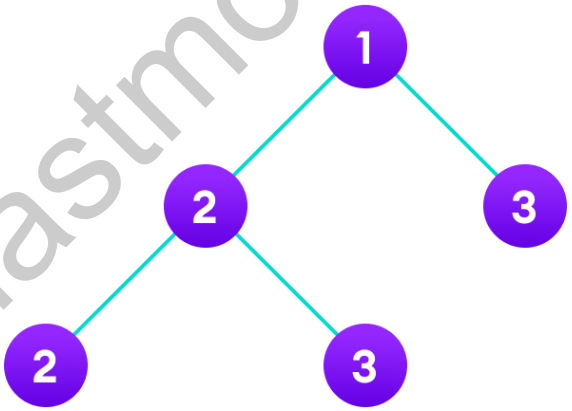


Popular Graph Based Data Structures:

- Spanning Tree and Minimum Spanning Tree
- Strongly Connected Components
- Adjacency Matrix
- Adjacency List

2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.



Popular Tree based Data Structure

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- B+ Tree
- Red-Black Tree

Linear Vs Non-linear Data Structures

Now that we know about linear and non-linear data structures, let's see the major differences between them.

Linear Data Structures	Non Linear Data Structures
The data items are arranged in sequential order, one after the other.	The data items are arranged in non-sequential order (hierarchical manner).
All the items are present on the single layer.	The data items are present at different layers.
It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.	It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.
The memory utilization is not efficient.	Different structures utilize memory in different efficient ways depending on the need.
The time complexity increases with the data size.	Time complexity remains the same.
Example: Arrays, Stack, Queue	Example: Tree, Graph, Map

Data structure operations

- **Traversing:** Accessing each element exactly once, is known as traversing. It is also known as visiting.
- **Insertion:** Adding a new element to data structure.
- **Deletion:** Removing an element from the data structure.
- **Searching:** Finding the location of an item in data structure.
- **Sorting:** Arranging the items in ascending or descending order is known as sorting.

★Algorithms

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces a desired output. For example,
An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

Qualities of Good Algorithms

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Algorithm 1: Add two numbers entered by the user

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

Algorithm 2: Find the largest number among three numbers

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

If a > c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b > c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

Algorithm 3: Find Root of the quadratic equation $ax^2 + bx + c = 0$

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D ← b2-4ac

Step 4: If D ≥ 0

r1 ← (-b+√D)/2a

r2 ← (-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp ← -b/2a

ip ← √(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less

memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Space Complexity

Space complexity of an algorithm is the amount of space it uses for execution in relation to the size of the input.

```
n = int(input())

nums = []
for i in range(1, n+1):
    nums.append(i*i)
```

In this example, the length of the list we create depends on the input value we provide for n. Let's say adding a single integer to the list takes c space and other initial operations, including creating a new list, takes d space. Then, we can create an equation for the space taken by the above algorithm like this.

when n -> c*n + d

when n = 10 -> c*10 + d

when n = 100 -> c*100 + d

The value calculated by this equation is the space the algorithm needs to complete execution. The values of the constants c and d are outside of the control of the algorithm and depend on factors such as programming language, hardware specifications, etc.

However, we don't need the exact value this equation calculates to talk about the space complexity of an algorithm. Instead, we use the highest order of the variable n as a representative of the space complexity.

For example, the above algorithm has a space complexity in the order of n. If another algorithm has the equation $c*n^2 + d*n + e$ for space it needs, we say it has an order of n^2 space complexity.

Time Complexity

Time complexity is the number of elementary operations an algorithm performs in relation to the input size. Here, we count the number of operations, instead of time itself, based on the assumption that each operation takes a fixed amount of time to complete.

If we look at the previous algorithm again, it performs n number of operations (n iterations of the loop) to complete its execution.

If we construct a similar equation for time complexity as we did before, it also takes the shape of c*n + d, with c as the fixed time taken for each loop iteration and d as the fixed time taken for other initial operations.

Therefore, the time complexity of this algorithm is also in the order of n.

Asymptotic Analysis

As you saw in these examples, we can't compare one algorithm to another using exact values because they depend on the tools we use and underlying hardware. In fact, if we calculated time and space values for two instances of running the same algorithm on the same system, there would be differences in the values we get due to subtle changes in the system environment.

Therefore, we use Asymptotic Analysis to compare the space and time complexity of two algorithms. It analyzes the performance of an algorithm against the input size. It evaluates how the performance changes as the input size increases. This type of analysis doesn't need actual values for space or time taken by the algorithm for comparison.

Best, Worst, and Average Cases

Usually, in asymptotic analysis, we consider three cases when analyzing an algorithm: best, worst, and average.

To understand each case, let's take an example of a linear search algorithm. We use a simple for loop to search if a given integer k is present in a list named nums of size n.

```
def linear_search(nums, n, k):
    for i in range(n):
        if k == nums[i]:
            return i
    return -1
```

Let's consider what are the best, worst, and average case scenarios for this algorithm in terms of time complexity (We can talk about these three scenarios in terms of space complexity too).

Best Case

We consider the combination of inputs that allows the algorithm to complete its execution in the minimum amount of time as the best-case scenario in terms of time complexity. The execution time in this case acts as a lower bound to the time complexity of the algorithm.
In linear search, the best-case scenario occurs when k is stored at the 0th index of the list. In this case, the algorithm can complete execution after only one iteration of the for loop.
nums = [1, 2, 3, 4, 5, 6]
n = 6
k = 1

Worst Case

Worst case scenario occurs when the combination of inputs that takes the maximum amount of time for completion is passed to the algorithm. The execution time of the worst case acts as an upper bound to the time complexity of the algorithm.
In linear search, the worst case occurs when k is not present in the list. This takes the algorithm n+1 iterations to figure out that the number is not in the list.
nums = [1, 2, 3, 4, 5, 6]
n = 6
k = 7

Average Case

To find the average case, we get the sum of running times of the algorithm for every possible input combination and take their average.

In linear search, the number of iterations the algorithm takes to complete execution follows this pattern.

When k is stored at the 0th index -> 1 iteration

When k is stored at the 1st index -> 2 iterations

When k is stored at the 2nd index -> 3 iterations

When k is stored at the 3rd index -> 4 iterations

: :

When k is stored at the nth index -> n iterations

When k is not in the list -> n+1 iterations

So, we can calculate the average running time of the algorithm this way.

$$\frac{\sum_{i=1}^{n+1} i}{n+1} = \frac{(n+1)*(n+2)/2}{n+1} = n/2 + 1$$

★Asymptotic Notation

Asymptotic notation is a mathematical notation used to represent the time and space complexity of algorithms in asymptotic analysis. We mainly use three asymptotic notations to represent the best, worst, and average cases of algorithms.

Ω (Big-Omega) Notation

Ω notation denotes an asymptotic lower bound of a function. In other words, it says that the function should output at least the respective big-omega value for a given input.

For a function g(n), we can define the set of functions denoted by Ω(g(n)) as follows.

Ω(g(n)) = {

f(n): there exist positive constants c and n_0 such that

0 <= c*g(n) <= f(n) for all n >= n_0

}

It's a mouthful. But let's break down this definition with an example and try to understand what it means.

First, let's take the function $g(n) = n^2$.

Now, the big-omega of g(n) represents the set of functions that satisfies the condition $0 <= c*g(n) <= f(n)$ for all $n >= n_0$ when c and n_0 are positive constants.

Let's consider the function $f(n) = 2n^2 + 4$

For $c = 1$ and $n_0 = 1$, $0 <= c*g(n) <= f(n)$ for all $n >= n_0$

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

Now, if we consider $f(n) = 3n + 5$, we can't find values for constants c and n0 that satisfy the above conditions. Therefore, $f(n) = 3n + 5$ doesn't belong to big-omega of g(n).

In time and space complexity, Ω notation is used to represent the best-case scenario of an algorithm. It can provide lower bounds to time and space complexity.

O (Big-O) Notation

O notation denotes an asymptotic upper bound of a function. In other words, the function should output at most the respective big-O value for a given input.

For a function g(n), the definition of the set O(g(n)) is as follows.

```
O(g(n)) = {
  f(n): there exist positive constants c and n0 such that
    0 <= f(n) <= c*g(n) for all n >= n0
}
```

Again, let's use an example to understand this definition.

$g(n) = n^2$

$f(n) = 2n^2 + 4$

For $c = 5$ and $n_0 = 1$, $0 <= f(n) <= c*g(n)$ for all $n >= n_0$

Therefore, f(n) = O(g(n)

And if we consider $f(n) = n^3+2$, it doesn't belong to O(g(n)) because no combinations of values for c and n0 satisfies the required condition.

We use O notation to represent the worst-case time and space complexity of an algorithm.

Θ (Big-Theta) Notation

Θ notation denotes an upper and a lower bound of a function. Therefore, it defines both at most and at least boundaries for the values the function can take for a given input.

The standard definition of the Θ notation is as follows.

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

$f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 <= c_1*g(n) <= f(n) <= c_2*g(n)$ for all $n >= n_0$

$\}$

Let's use an example to understand this definition with the g(n) and f(n) functions we used so far.

$g(n) = n^2$

$f(n) = 2n^2 + 4$

For $n_0 = 1$, $c_0 = 1$, and $c_1 = 5$, $0 <= c_0*g(n) <= f(n) <= c_1*g(n)$ for all $n >= n_0$

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

Big-theta notation is used to define the average case time and space complexity of an algorithm.