
PNS SCHOOL OF ENGINEERING & TECHNOLOGY

Nishamani Vihar, Marshaghai, Kendrapara

LECTURE NOTES

# DATA STRUCTURE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

3RD SEMESTER

PREPARED BY

***MRS. JAYASHREE BISHOI***

LECTURER IN COMPUTER SCIENCE & ENGINEERING

# CHAPTER-1

**Introduction to Data Structures**

1.1 Basic Terminology
1.2 Classification of Data Structure
1.3 Operations on Data Structure
1.4 Asymptotic and worst-case analysis of algorithms.

**Introduction**

**Data**

The term ***Data*** is defined as a raw and unstructured fact that needs to be processed to make it Meaningful. Data can be simple and unstructured at the same time until it is structured. Usually data contains facts, numbers, symbols, image, observations, perceptions, characters, etc.

**Information**

The term ***Information*** is defined as a set of data that is processed according to the given requirement in a meaningful way. To make the information useful and meaningful, it must be processed, presented and structured in a given context. Information is processed from data and possess context, purpose and relevance.

**Data Type:**

Data types are used within type systems, which offer different ways of defining, implementing and using the data. Different languages may use different terminology. Common data types are :
Integers,
Booleans,
Characters,
Floating-point numbers,
Alphanumeric strings.

**What is a Data Type?**

A **data type** is a classification that tells a computer or programming language:

- **What kind of data is being stored**, and
- **What operations can be performed** on that data.

For example:

- Is the data a **number**?
- Is it **text**?
- Is it a **true/false** value?
- Or is it a **more complex structure**, like an object or list?

So, **data types** help a program manage memory efficiently and perform correct operations.

**Classes of Data Types**

Data types are usually grouped into **two main classes** (sometimes three if you separate Abstract/Derived types):

---

**1 Primitive (Basic) Data Types**

These are the simplest, predefined by the programming language.

| Class | Description | Examples |
|---|---|---|
| **Integer** | Whole numbers without decimals | int in C/C++, Java |
| **Floating Point** | Numbers with decimals | float, double |
| **Character** | Single alphabet or symbol | char |
| **Boolean** | Logical true or false | bool |
| **Void** | Represents absence of value | void (used for functions that return nothing) |

---

**2Derived Data Types**

These are built from primitive data types.

| Class | Description | Examples |
|---|---|---|
| **Array** | Collection of elements of the same type | int arr[10]; |
| **Pointer** | Stores memory address of another variable | int *ptr; |
| **Function** | Functions can be treated as data types (in C/C++) | void func(); |
| **Reference** | Alias for another variable (C++) | int &ref = var; |

---

**3User-Defined or Abstract Data Types**

Created by programmers to model real-world entities.

| Class | Description | Examples |
|---|---|---|
| **Structure (struct)** | Groups variables of different types | struct Employee { int id; char name[20]; }; |
| **Union** | Like struct, but shares memory | union Data { int i; float f; }; |
| **Class** | Used in OOP; defines objects with attributes & methods | class Car { ... }; |
| **Enumeration (enum)** | User-defined constants | enum Color { RED, GREEN, BLUE }; |

---

**Summary**

| Class of Data Type | Includes |
|---|---|
| **Primitive** | int, float, char, bool |
| **Derived** | arrays, pointers, functions, references |
| **User-defined** | structs, unions, classes, enums |

**Introduction to 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. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge of data structures.

**Definition**

In Computer Science, 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. Data structures determine the way in which information can be stored in computer and used. Finding the

best data structure when solving a problem is an important part of programming. Programs that use
the right data structure are easier to write, and work better.

***Data Structure Operations:***

The various operations that can be performed on different data structures are as follows:

1. Create– A data structure created from data.
2. Traverse – Processing each element in the list
3. Searching – Finding the location of given element.
4. Insertion – Adding a new element to the list.
5. Deletion – Removing an element from the list.
6. Sorting – Arranging the records either in ascending or descending order.
7. Merging – Combining two lists into a single list.
8. Modifying – the values of DS can be modified by replacing old values with new ones.
9. Copying – records of one file can be copied to another file.
10. Concatenating – Records of a file are appended at the end of another file.
11. Splitting – Records of big file can be splitting into smaller files.

**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 so 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.

- 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:**

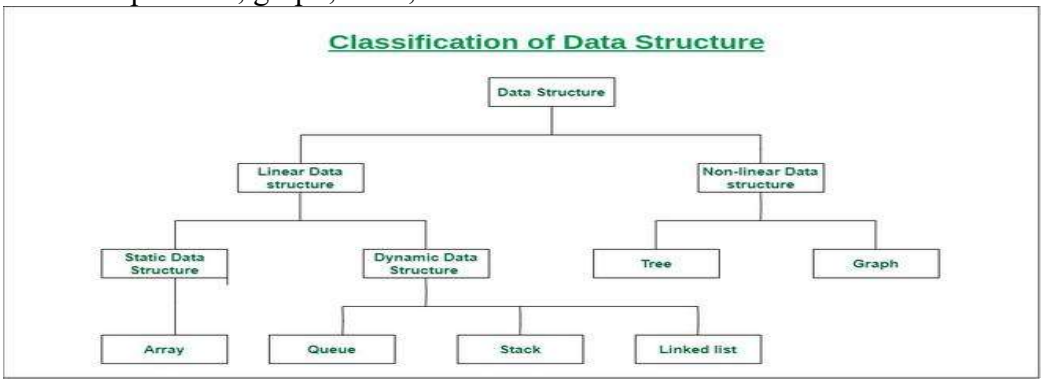
1. Linear Data Structure
2. Non-Linear Data Structure.

**Linear Data Structure:**

- Elements are arranged in one dimension ,also known as linear dimension.
- Example: lists, stack, queue, etc.

**Non-Linear Data Structure**

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.



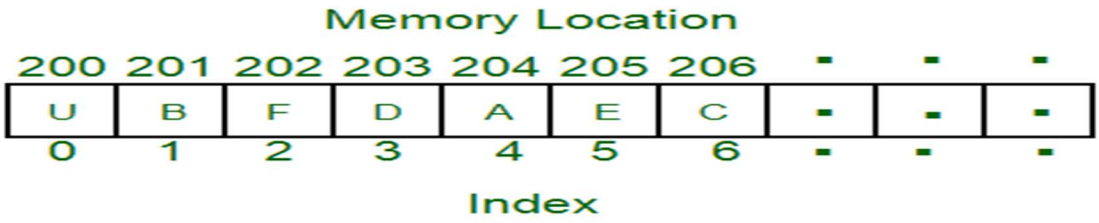
- **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
  *Examples: array, stack, queue, linked list, etc.*
- **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
  *Example: array data structure.*
- **Dynamic data structure:** In the 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: stack and queue data structures.*
- **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.
  *Examples: tree and graph data structures.*

**Arrays Data Structure**

An array is a **linear data structure** and it is a collection of element of same **data type** stored at **contiguous memory locations**.

It offers mainly the following advantages.

- **Random Access**: i-th elements can be accessed in O(1) Time as we have the base address and every element is of same size.
- **Cache Friendliness**: Since elements are stored at contiguous locations, we get the advantage of locality of reference.




# Array

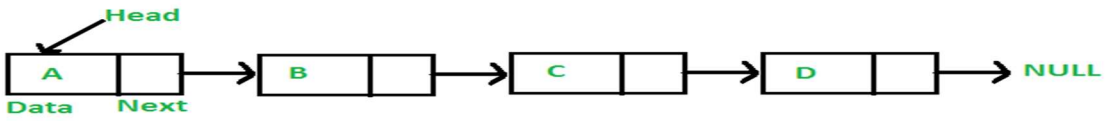
**Different applications of an array are as follows:**

Arrays efficiently manage and store database records.

- It helps in implementing sorting algorithm.
- It is also used to implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.
- An array can be used for CPU scheduling.

**Linked list Data Structure**

A linked list is a linear data structure in which elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image.



## Linked List

### Applications of the Linked list

- Linked lists are used to implement other data structures like stacks, queues, etc.
- It is used for the representation of sparse matrices.
- It is used in the linked allocation of files.
- Linked lists are used to display image containers. Users can visit past, current, and next images.
- They are used to perform undo operations.

### Stack Data Structure

Stack is a **linear data structure** that follows LIFO(Last in first out) principle i.e., entering and retrieving data is possible from only one end. The entering and retrieving of data is also called push and pop operation in a stack.


Stack Data Structure

## Stack

### Applications of Stack

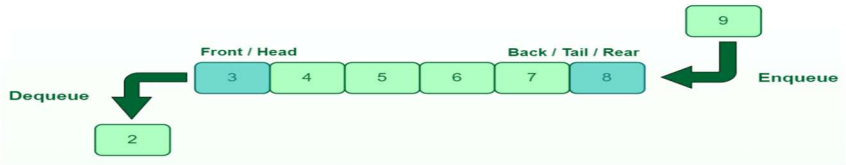
Different applications of Stack are as follows:

- The stack data structure is used in the evaluation and conversion of arithmetic expressions.
- It is used for parenthesis checking and string reversal.
- A memory stack is also used for processing function calls.
- The stack is used in virtual machines like JVM.

### Queue Data Structure

Queue is a **linear data structure** that follows First In First Out(FIFO) principle i.e. the data item stored first will be accessed first. In this, entering is done from one end and retrieving data is done from other end. An example of a queue is any queue of consumers for a resource where the consumer that came first is served first.

Queue




Queue Data Structure

### Applications of Queue:

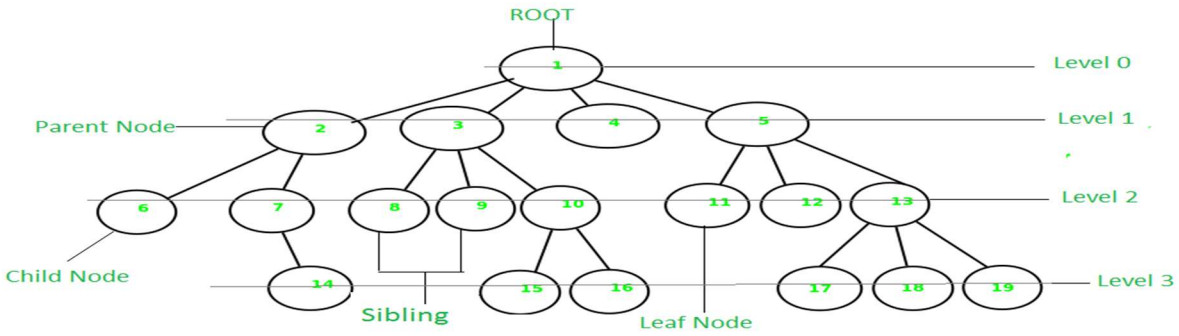
Different applications of Queue are as follows:

- Queue is used for handling website traffic.
- It helps to maintain the playlist in media players.
- It helps in serving requests on a single shared resource, like a printer, CPU task scheduling, etc.

- Queues are used for job scheduling in the operating system.

**Tree Data Structure**

A tree is a non-linear and hierarchical data structure where the elements are arranged in a tree-like structure. In a tree, the topmost node is called the root node. Each node contains some data, and data can be of any type. It consists of a central node, structural nodes, and sub-nodes which are connected via edges. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.




**Applications of Tree:**

- Heap is a tree data structure that is implemented using arrays and used to implement priority queues.
- B-Tree and B+ Tree are used to implement indexing in databases.
- Syntax Tree helps in scanning, parsing, generation of code, and evaluation of arithmetic expressions in Compiler design.
- Spanning trees are used in routers in computer networks.
- Domain Name Server also uses a tree data structure.

**Binary Search Tree Data Structure**

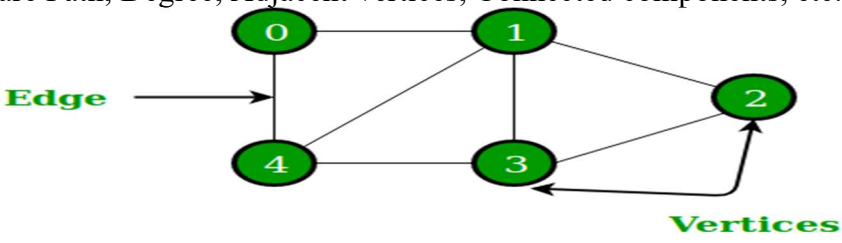
A **Binary Search Tree (or BST)** is a data structure used for organizing and storing data in a sorted manner. Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child, with the **left** child containing values less than the parent node and the **right** child containing values greater than the parent node. This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.

**Applications of Binary Search Tree:**

- A Self-Balancing BST maintains a sorted stream of data in RAM, useful for tracking online orders by price and querying item counts above or below a given cost.
- It enables a doubly-ended priority queue, supporting both **extractMin()** and **extractMax()** in **O(log n)** time, unlike a Binary Heap.
- Many algorithmic problems, like counting smaller elements on the right or finding the smallest greater element, benefit from a Self-Balancing BST.
- **Tree Map** and **Tree Set** in Java, and **set** and **map** in C++, are implemented using Red-Black Trees, a type of Self-Balancing BST.

**Graph Data Structure**

A graph is a non-linear data structure that consists of vertices (or nodes) and edges. It consists of a finite set of vertices and set of edges that connect a pair of nodes. The graph is used to solve the most challenging and complex programming problems. It has different terminologies which are Path, Degree, Adjacent vertices, Connected components, etc.



Graph

**Applications of Graph:**

- The operating system uses Resource Allocation Graph.
- Also used in the World Wide Web where the web pages represent the nodes.
- One of the most common real-world examples of a graph is Google Maps where cities are located as vertices and paths connecting those vertices are located as edges of the graph.
- A social network is also one real-world example of a graph where every person on the network is a node, and all of their friendships on the network are the edges of the graph.

**Operations on Data Structure**

Operations on data structures refer to the fundamental actions that can be performed to manipulate and interact with the data stored within them. These operations vary depending on the specific data structure but generally fall into common categories.

Common Operations on Data Structures:

- **Insertion:**

  The process of adding a new data element into the data structure. The placement of the new element depends on the structure's rules (e.g., at the end of a list, at the top of a stack, or based on a key in a hash table).

- **Deletion:**

  The process of removing an existing data element from the data structure. This may involve removing a specific element, the first element, or the last element, depending on the data structure and the deletion criteria.

- **Traversal:**

The operation of visiting and processing each element within the data structure, often in a specific order (e.g., sequentially in an array, in-order in a binary search tree).

- **Searching:**

  The process of locating a specific data element within the data structure based on a given key or value.

- **Sorting:**

  The operation of arranging the elements within the data structure in a specific order, such as ascending or descending.

- **Updating/Modification:**

  The process of changing the value of an existing data element within the data structure.

- **Access:**

  Retrieving the value of an element at a specific position or based on a key.

- **Merging:**

  Combining two or more data structures into a single data structure, often while maintaining a specific order if the original structures were sorted.

- **Reversing:**

  Changing the order of elements within a data structure to the inverse of their original arrangement.

The specific set of operations and their implementation details will vary significantly across different data structures like arrays, linked lists, stacks, queues, trees, graphs, and hash tables.

**Asymptotic Analysis**

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T(n)**, where **n** is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

**Types of Asymptotic Notations**

Asymptotic notation is a mathematical tool used in computer science to describe the efficiency of algorithms, particularly how their performance (time or space) scales with the size of the input. There are five main types of asymptotic notations:

- Big O Notation (O): Represents the worst-case complexity of an algorithm, providing an upper bound on its growth rate. If an algorithm's running time is O(f(n)), it means that the running time will not grow faster than f(n) (multiplied by a constant) for large inputs. This is the most commonly used notation for comparing algorithms because it helps predict performance under the most challenging scenarios.
  - Examples: O(1) (constant time, like accessing an array element), O(log n) (logarithmic time, like binary search), O(n) (linear time, like linear search), $O(n^2)$ (quadratic time, like bubble sort).
- Big Omega Notation (Ω): Represents the best-case complexity, providing a lower bound on an algorithm's growth rate. If an algorithm's running time is Ω(f(n)), it means that the running time will grow at least as fast as f(n) (multiplied by a constant) for large inputs.
  - Examples: Ω(1) (constant time, best case for many operations), Ω(log n) (best case for binary search).
- Big Theta Notation (Θ): Represents the average-case complexity, providing a tight bound on the growth rate. If an algorithm's running time is Θ(f(n)), it means it is bounded both from above and below by constant multiples of f(n). This provides a more precise description of the algorithm's typical performance.
  - Examples: Θ(1) (constant time, like array access in best case), Θ(n) (linear time, traversing a linked list).
- Little o Notation (o): Provides a strict upper bound on an algorithm's growth rate, indicating that it is strictly less than the specified function for large inputs. If an algorithm's running time is o(f(n)), it means the running time becomes insignificant compared to f(n) as the input size approaches infinity.

- Example: If an algorithm is $o(n^2)$, its growth is faster than n, but never reaches $n^2$.

- Little Omega Notation (ω): Provides a strict lower bound on an algorithm's growth rate, indicating that it is strictly greater than the specified function for large inputs. If an algorithm's running time is ω(f(n)), it means the running time grows faster than f(n) for large inputs.

- Example: If an algorithm is ω(n), its runtime grows faster than n, potentially quadratically or exponentially.

In essence, Big-O focuses on the upper limit (worst-case), Big-Omega on the lower limit (best-case), and Big-Theta provides a more exact "tight" bound, representing both upper and lower limits. Little-o and Little-omega offer strict bounds that are not asymptotically tight, meaning the functions do not grow at the same rate but one is strictly slower or faster than the other.

## Worst-case Analysis of Algorithm

**1. Worst Case Analysis (Mostly used)**

- In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
- For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one.
- This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

**2. Best Case Analysis (Very Rarely used)**

- In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
- For linear search, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

**3. Average Case Analysis (Rarely used)**

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
- We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). We take (n+1) to consider the case when the element is not present.

**Why is Worst Case Analysis Mostly Used?**

**Average Case :** The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we need to consider every input, its frequency and time taken by it which may not be possible in many scenarios

**Best Case :** The Best Case analysis is considered bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

**Worst Case:** This is easier than average case and gives an upper bound which is useful information to analyze software products.

# Chapter-2
# Linear Data Structures