A **data structure** is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.

Some common types of data structures include:

1. **Arrays:** An array is a collection of elements of the same type, stored in contiguous memory locations. Elements can be accessed by their index in the array.

2. **Linked Lists**: A linked list is a collection of elements, called nodes, where each node contains an element and a reference to the next node in the list. Linked lists can be used to implement dynamic data structures, such as stacks and queues.

3. **Stacks:** A stack is a last-in, first-out (LIFO) data structure. Elements are added to and removed from the top of the stack.

4. **Queues:** A queue is a first-in, first-out (FIFO) data structure. Elements are added to the back of the queue and removed from the front.

5. **Trees**: A tree is a hierarchical data structure composed of nodes, where each node can have several children, but only one parent. Trees can be used to represent hierarchical relationships, such as the organization of a file system.

6. **Graphs**: A graph is a non-linear data structure that is composed of a set of vertices (or nodes) and edges that connect them. Graphs can be used to represent relationships between elements, such as the connections between web pages.

7. **Hash tables**: A hash table is a data structure that is used to implement an associative array, which is a collection of key-value pairs. Keys are used to access the associated values.

8. **Heaps:** A heap is a specialized tree-based data structure that satisfies the heap property, which states that the key of each node is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the keys of its children.

9. **Trie:** A Trie is a tree-based data structure which is used to store an associative array where the keys are sequences (usually strings). It is used to store a dynamic set or associative array where the keys are usually strings.

Need for Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, if our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Data structures provide an easy way of organizing, 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.

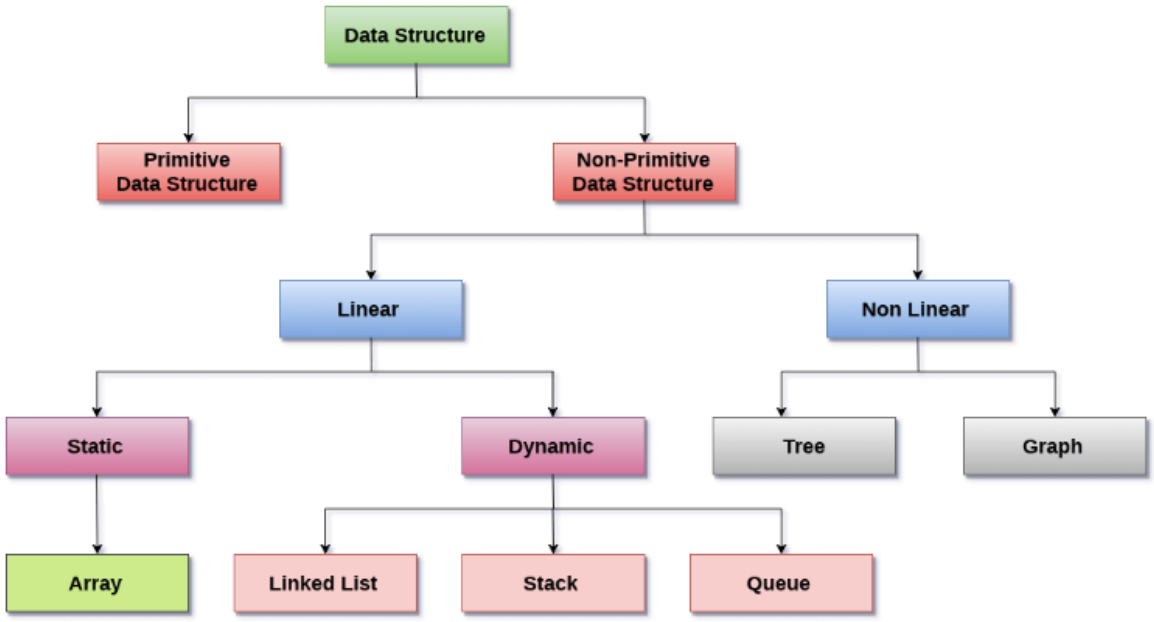
Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e., once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Types of Data Structures



1) Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

2) Non-Linear Data Structures

This data structure does not form a sequence i.e., each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non-Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Data Structure Operations

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

An Abstract Data Type (ADT) is a high-level description of a collection of data and the operations that can be performed on that data. It specifies what the data represents and what operations are allowed on the data, but it does not specify how the data is stored or how the operations are implemented. This allows different implementations of the same ADT to be used in different situations, depending on the specific requirements of the application.

Some examples of ADT's include:

1. **Stack**: An ADT that represents a last-in, first-out (LIFO) collection of elements. It provides operations such as push, pop, and peek.

2. **Queue**: An ADT that represents a first-in, first-out (FIFO) collection of elements. It provides operations such as enqueue, dequeue, and peek.

3. **List**: An ADT that represents an ordered collection of elements. It provides operations such as insert, delete, and get.

4. **Set**: An ADT that represents a collection of unique elements. It provides operations such as add, remove, and contains.

5. **Map**: An ADT that represents a collection of key-value pairs. It provides operations such as put, get, and remove.

6. **Graph**: An ADT that represents a collection of vertices (or nodes) and edges that connect them. It provides operations such as add vertex, add edge, and find shortest path.

7. **Tree**: An ADT that represents a hierarchical collection of elements. It provides operations such as add child, remove child, and get parent.

An Abstract Data Type (ADT) consists of two main parts:

The interface: This defines the operations that can be performed on the data structure, such as inserting, deleting, and searching for data. The interface specifies the input and output of each operation, but not the implementation details. This allows the data structure to be used in a variety of contexts and to be easily replaced with other data structures if needed.

The implementation: This is the specific way in which the data structure is implemented, such as using an array, a linked list, or a tree. The implementation details are hidden from the user, and only the interface is exposed. This allows for different implementations of the same ADT to be used interchangeably, depending on the specific needs of the application.

For example, a stack is an ADT that consists of an interface that defines the push, pop, peek, is_empty and size operations, while its implementation can be done using array or linked list, but the user only need to know how to use the stack interface.

Another way to think of an Abstract Data Type (ADT) is as a combination of value definition and operator definition.

Value definition: This defines the type of data that the ADT will hold and manipulate, such as integers, strings, or custom objects. It specifies the domain of the data and the operations that can be performed on it.

Operator definition: This defines the set of operations that can be performed on the data, such as inserting, deleting, and searching for data. It specifies the input and output of each operation, but not the implementation details.

Together, the value definition and operator definition provide a complete and consistent view of the data and the operations that can be performed on it. The ADT serves as a contract between the user and the implementer, specifying what the user can expect the data structure to do and how to use it, while hiding the implementation details.

For example, a stack is an ADT that defines the value of integers, and the operators like push, pop, peek, is_empty and size. This ADT is consistent and clear, allowing a user to understand how to use a stack without worrying about how it is implemented.

In this way, the ADT provides a clear and consistent abstraction of the data and the operations, making it easy to use and understand, and allowing for flexibility and modularity in software design.

Justify, data structure is the backbone of software programming. Explain.

Data structures are the backbone of software programming because they provide the foundation for organizing and manipulating data in a way that is efficient and effective for the specific needs of the software.

Data structures provide the means for the software to store, retrieve, and manipulate data in an organized and efficient manner. They allow for the efficient use of memory and processing power by providing optimized algorithms for common operations such as inserting, deleting, and searching for data.

In addition, data structures provide a way to model complex relationships and structures in the data, such as hierarchical relationships, networks, and graphs. This allows the software to effectively manipulate and analyze the data, and make decisions based on the data.

Also, the choice of data structure affects the performance of the algorithm implemented on it, which can be significant in large-scale systems, where performance is critical.

In summary, data structures provide the foundation for efficient and effective data manipulation and analysis in software programming, and their proper use is essential for the overall performance and functionality of the software.