UNIT-III
NON LINERAR DATA STRUCTURES

Arrays and its representations – Stacks and Queues – Linked lists – Linked list-based implementation of Stacks and Queues – Evaluation of Expressions – Linked list based polynomial addition.

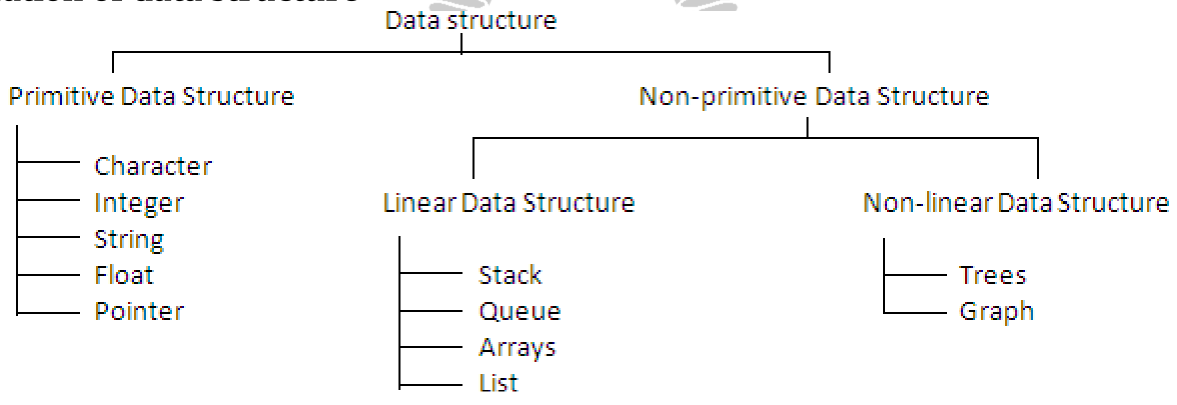
Definition:

- Data structure is a particular way of organizing, storing and retrieving data, so that it can be used efficiently. It is the structural representation of logical relationships between elements of data.

Where data structures are used?

- Data structures are used in almost every program or software system. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
- **Applications** in which data structures are applied extensively
  - Compiler design (Hash tables),
  - Operating system,
  - Database management system (B+Trees),
  - Statistical analysis package,
  - Numerical analysis (Graphs),
  - Graphics,
  - Artificial intelligence,
  - Simulation

Classification of data structure



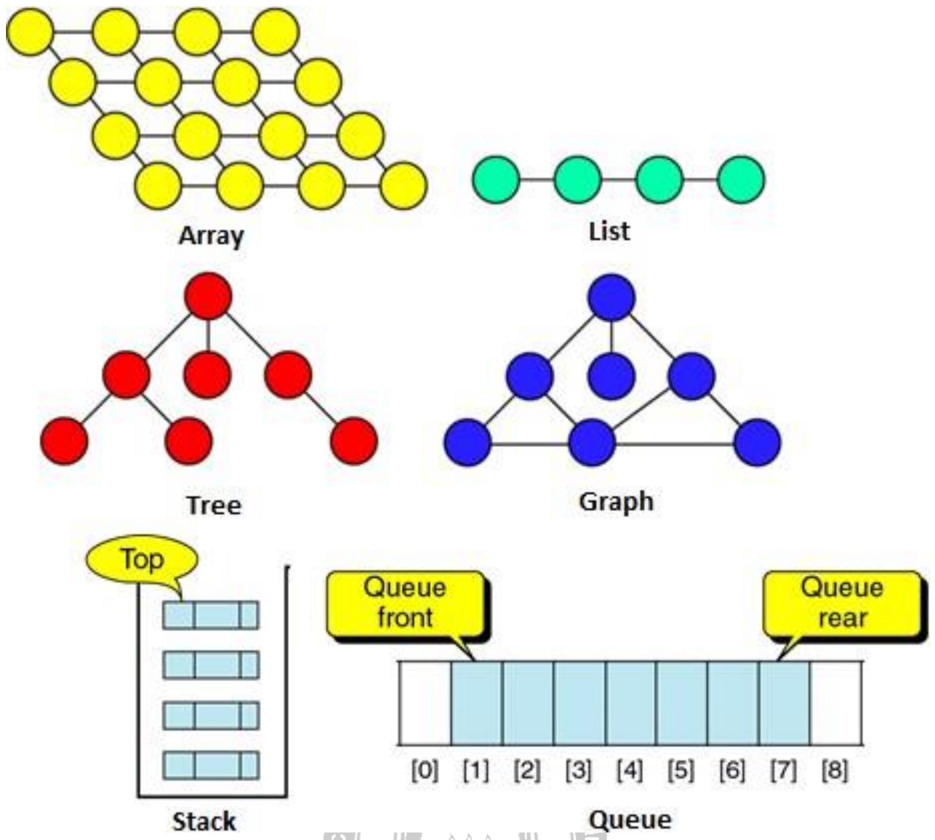
- **Primitive Data Structure -** Primitive data structures are predefined types of data, which are supported by the programming language. These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level.
- **Non-Primitive Data Structure -** Non-primitive data structures are not defined by the programming language, but are instead created by the programmer. It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items**.**
- **Linear data structure-** only two elements are adjacent to each other. (Each node/element has a

single successor)

  - Restricted list (Addition and deletion of data are restricted to the ends of the list)
    - ✓ Stack (addition and deletion at **top** end)
    - ✓ Queue (addition at **rear** end and deletion from **front** end)
  - General list (Data can be inserted or deleted anywhere in the list: at the beginning, in the middle or at the end)

- **Non-linear data structure-** One element can be connected to more than two adjacent elements.(Each node/element can have more than one successor)
  - Tree (Each node could have multiple successors but just one predecessor)
  - Graph (Each node may have multiple successors as well as multiple predecessors)

Note - Array and Linked list are the two basic structures for implementing all other ADTs.



MODULARITY

- **Module-** A module is a self-contained component of a larger software system.Each module is a logical unit and does a specific job. Its size kept small by calling other modules.
- **Modularity** is the degree to which a system's components may be separated and recombined. Modularity refers to breaking down software into different parts called modules.
- **Advantages** of modularity
  - It is easier to debug small routines than large routines.
  - Modules are easy to modify and to maintain.
  - Modules can be tested independently.
  - Modularity provides reusability.
  - It is easier for several people to work on a modular program simultaneously.

ABSTRACT DATA TYPE

What is Abstract Data Type (ADT)?

- ADT is a mathematical specification of the data, a list of operations that can be carried out on that data. It **includes** the specification of what it does, but **excludes** the specification of how it does. Operations on **set ADT**: Union, Intersection, Size and Complement.
- The **primary objective** is to separate the implementation of the abstract data types from their function. The program must know what the operations do, but it is actually better off not knowing how it is done. Three most common used abstract data types are Lists, Stacks, and Queues.
- ADT is an **extension of modular design**. The **basic idea** is that the implementation of these operations is **written once in the program**, and any other part of the program that needs to

perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

- Examples of ADT: Stack, Queue, List, Trees, Heap, Graph, etc.

- **Benefits of using ADTs or Why ADTs**

- Code is easier to understand. Provides modularity and reusability.
- Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs.

LIST ADT

- List is a linear collection of ordered elements. The general form of the list of size N is: **A_0, A_1, …, A_{N-1}**
  - Where A_1 - First element
    A_N - Last Element
    N - Size of the list
  - If the element at position 'i' is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.
- Various operations performed on a List ADT

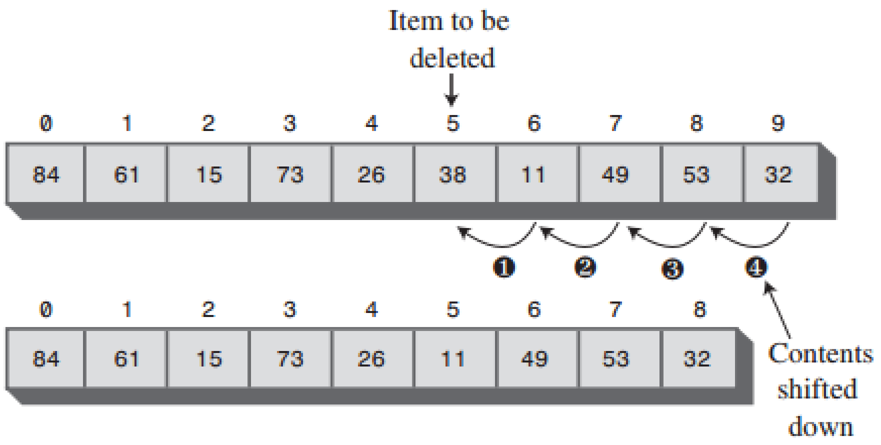
Insert (X,5)	- Insert the element X after the position 5.
Delete (X)	- The element X is deleted.
Find (X)	- Returns the position of X
Next (i)	- Returns the position of its successor element i+1.
Previous (i)	- Returns the position of its Predecessor element i-1.
PrintList	- Displays the List contents.
MakeEmpty	- Makes the List empty.

- **Implementation of List ADT**
  - Array implementation
  - Linked List implementation
  - Cursor implementation

ARRAY IMPLEMENTATION OF LIST ADT

- An array is a collection of homogeneous data elements described by a single name. Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis. In array implementation, elements of list are stored in contiguous cells of an array. Find K^{th} operation takes constant time. **Print List**, **Find** operations take linear time.
- Advantages - **Searching** an array for an **individual** element can be **very efficient -** Fast, random access of elements.
- Limitations - Array implementation has some limitations such as
  1. Maximum size must be known in advance, even if it is dynamically allocated.
  2. The size of array can't be changed after its declaration (static data structure). i.e., the size is fixed.
  3. Data are stored in continuous memory blocks.
  4. The running time for Insertion and deletion of elements is so slow. Inserting and deletion requires shifting other data in the array. For example, inserting at position 0 requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is O(n). On average, half the list needs to be moved for either operation, so linear time is still required.

5. Memory is wasted, as the memory remains allocated to the array throughout the program



execution even few nodes are stored.

Deleting an item

Type Declarations#define Max 10

```
int A[Max],N;
```

Routine to insert an Element in the specified position

```
void insert(int x, int p, int A[], int N)
{
int i; If(p==N)
printf("Array Overflow");
else
{
for(i=N-1;i>=p-1;i--) A[i+1]=A[i];
A[p-1]=x; N=N+1;
}
}
```

Routine to delete an Element in the specified

```
int deletion(int p, int A[],int N)
{
int Temp; If(p==N)
Temp=A[p-1]; else
{
Temp=A[p-1];          For(i=p-1;i<=N-1;i++)
A[i]=A[i+1];
}
N=N-1;
return Temp;
}
```

Find Routine

```
void Find (int X)
{
int i,f=0; for(i=0;i<N;i++)if(a[i]==x)
{ f=1;
break;
}
if (f==1)
printf("Element found");
else
printf("Element not found");
}
```