Introduction to Data Structures and Algorithms

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's **name** "Virat" and **age** 26. Here "Virat" is of **String** data type and 26 is of **integer**data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

If you are aware of Object Oriented programming concepts, then a `class` also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

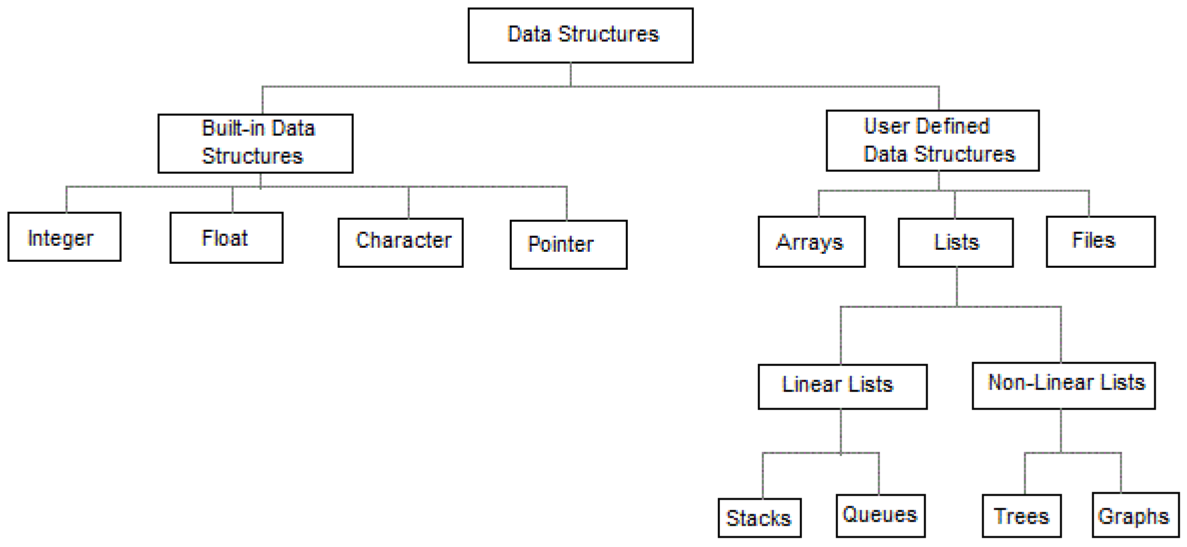
Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

Characterstic	**Description**
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**

Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

What is an Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

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

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space:** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** Its the space required to store all the constants and variables(including temporary variables) value.
- **Environment Space:** Its the space required to store the environment information needed to resume the suspended function.

To learn about Space Complexity in detail, jump to the Space Complexity tutorial.

Time Complexity

Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible. We will study about Time Complexityin details in later sections.

NOTE: Before going deep into data structure, you should have a good knowledge of programming either in C or in C++ or Java or Python etc.

Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

***Space complexity** is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = **Auxiliary Space + Input space**

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

   It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

   Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

   For example, If a function `A()` calls function `B()` inside it, then all th variables of the function `A()`will get stored on the system stack temporarily, while the function `B()` is called and executed inside the funciton `A()`.

3. **Data Space**

   Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space**and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	**Size**
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Now let's learn how to compute space complexity by taking a few examples:

```
{
    int z = a + b + c;
    return(z);
}
```

In the above expression, variables `a`, `b`, `c` and `z` are all integer types, hence they will take up 4 bytes each, so total memory requirement will be `(4(4) + 4) = 20 bytes`, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Let's have another example, this time a bit complex one,

```
// n is the length of array a[]
int sum(int a[], int n)
{
        int x = 0;              // 4 bytes for x
        for(int i = 0; i < n; i++)      // 4 bytes for i
        {
            x  = x + a[i];
        }
        return(x);
}
```

- In the above code, `4*n` bytes of space is required for the array `a[]` elements.

- 4 bytes each for `x`, `n`, `i` and the return value.

Hence the total memory requirement will be `(4n + 12)`, which is increasing linearly with the increase in the input value `n`, hence it is called as **Linear Space Complexity.**

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.

But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

Pseudocode

Pseudocode is a simple way of writing programming code in English. Pseudocode is not actual programming language. It uses short phrases to write code for programs before you actually create it in a specific language. Once you know what the program is about and how it will function, then you can use pseudocode to create statements to achieve the required results for your program.

Understanding Pseudocode

Pseudocode makes creating programs easier. Programs can be complex and long; preparation is the key. For years, flowcharts were used to map out programs before writing one line of code in a language. However, they were difficult to modify and with the advancement of programming languages, it was difficult to display all parts of a program with a flowchart. It is challenging to find a mistake without understanding the complete flow of a program. That is where pseudocode becomes more appealing.

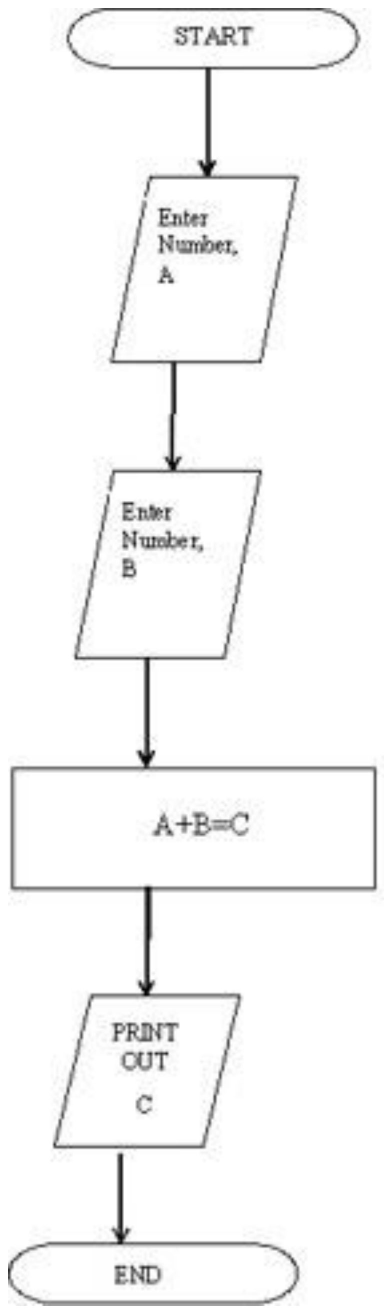
To use pseudocode, all you do is write what you want your program to say in English. Pseudocode allows you to translate your statements into any language because there are no special commands and it is not standardized. Writing out programs before you code can enable you to better organize and see where you may have left out needed parts in your programs. All you have to do is write it out in your own words in short statements. Let's look at some examples.

Examples of Pseudocode

Let's review an example of pseudocode to **create a program to add 2 numbers together and then display the result**.

```
Start Program
Enter two numbers, A, B
Add the numbers together
Print Sum
End Program
```

Compare that pseudocode to an example of a flowchart to add two numbers



Now, let's look at a few more simple examples of pseudocode. Here is a pseudocode to **compute the area of a rectangle**:

```
Get the length, l, and width, w
Compute the area = l*w
Display the area
```

Now, let's look at an example of pseudocode to **compute the perimeter of a rectangle**:

```
Enter length, l
Enter width, w
Compute Perimeter = 2*l + 2*w
Display Perimeter of a rectangle
```

Remember, writing basic pseudocode is not like writing an actual coding language. It cannot be compiled or run like a regular program. Pseudocode can be written how you want. But some companies use specific pseudocode syntax to keep everyone in the company on the same page. **Syntax** is a set of rules on how to use and organize statements in a programming language. By adhering to specific syntax, everyone in the company can read and understand the flow of a program. This becomes cost effective and there is less time spent finding and correcting errors.

Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.
The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides

the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, StackADT, Queue ADT.
An abstract data type (**ADT**) is basically a logical description or a specification of components of the data and the operations that are allowed, that is independent of the implementation.

ADTs are a theoretical concept in computer science, used in the design and analysis of algorithms, data structures, and software systems, and do not correspond to specific features of computer languages.

There may be thousands of ways in which a given ADT can be implemented, even when the coding language remains constant. Any such implementation must comply with the content-wise and behavioral description of the ADT.

Examples

For example, integers are an ADT, defined as the values 0, 1, −1, 2, 2, ..., and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc. which are independent of how the integers are represented by the computer. Typically integers are represented in as binary numbers, most often as two's complement, but might be binary-coded decimal or in ones' complement, but the user is abstracted from the concrete choice of representation, and can simply use the data as integers.

Another example can be a list, with components being the number of elements and the type of the elements. The operations allowed are inserting an element, deleting an element, checking if an element is present, printing the list etc. Internally, we may implement list using an array or a linked list, and hence the user is abstracted from the concrete implementation.
Check this out for the above two implementations of the list ADT: CS13002 Programming and Data Structures

Why ADT?

To manage the complexity of problems and the problem-solving process, abstractions are used to allow the user to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, the user can efficiently focus on the problem-solving process.

Advantages

- Encapsulation: The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.
- Localization of change: Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed, since any changes to the implementation must still comply with the properties and abilities specified in the ADT definition.
- Flexibility: Different implementations of an ADT may be more efficient in different situations and it is possible to use each in the situation where they are preferable. Hence this flexibility increases the overall efficiency.
- Easy to understand and reusable code.

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list.

get() – Return an element from the list at any given position.

insert() – Insert an element at any position of the list.

remove() – Remove the first occurrence of any element from a non-empty list.

removeAt() – Remove the element at a specified location from a non-empty list.

replace() – Replace an element at any position by another element.

size() – Return the number of elements in the list.

isEmpty() – Return true if the list is empty, otherwise return false.

isFull() – Return true if the list is full, otherwise return false.

Stack ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:
push() – Insert an element at one end of the stack called top.

pop() – Remove and return the element at the top of the stack, if it is not empty.

peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

size() – Return the number of elements in the stack.

isEmpty() – Return true if the stack is empty, otherwise return false.

isFull() – Return true if the stack is full, otherwise return false.

Queue ADT

A Queue contains elements of same type arranged in sequential order. Operations takes place at both ends, insertion is done at end and deletion is done at front. Following operations can be performed:

enqueue() – Insert an element at the end of the queue.

dequeue() – Remove and return the first element of queue, if the queue is not empty.

peek() – Return the element of the queue without removing it, if the queue is not empty.

size() – Return the number of elements in the queue.

isEmpty() – Return true if the queue is empty, otherwise return false.

isFull() – Return true if the queue is full, otherwise return false.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.