UNIT - I:

Introduction to Data Structures: Introduction to the Theory of Data Structures, Data Representation, Abstract Data Types, Data Types, Primitive Data Types, Data Structure and Structured Type, Atomic Type, Difference between Abstract Data Types, Data Types, and Data Structures, Refinement Stages.

Principles of Programming and Analysis of Algorithms: Software Engineering, Program Design, Algorithms, Different Approaches to Designing an Algorithm, Complexity, Big 'O' Notation, Algorithm Analysis, Structured Approach to Programming, Recursion, Tips and Techniques for Writing Programs in 'C'.

1.1 Introduction:

Data structure is the branch of computer science that unleashes the knowledge of how the data should be organized, how the flow of data should be controlled and how a data structure should be designed and implemented to reduce the complexity and increase the efficiency of the algorithm.

The theory of structures not only introduces you to the data structures, but also helps you to understand and use the concept of abstraction, analyze problems step by step and develop algorithms to solve real world problems. It enables you to design and implement various data structures, for example, the stacks, queues, linked lists, trees and graphs. Effective use of principles of data structures increases efficiency of algorithms to solve problems like searching, sorting, populating and handling voluminous data.

Need of a Data Structure

1. Data structure helps you to understand the relationship of one data element withy the other and organize it within memory
2. With increasing complexities in computer algorithms, the amount of data usage is increasing, this can affect the performance of the application and can create some areas of concern:

   Processing speed: To handle very large data, high-speed processing is required, but with growing data processor may fail to achieve required processing speed.

   Data Search: Getting a particular record from database should be quick and with optimum use of resources.

   Multiple requests: To handle simultaneous requests from multiple users

Data Structure Advantages

Efficient Memory use: With efficient use of data structure memory usage can be optimized, for e.g we can use linked list vs arrays when we are not sure about the size of data. When there is no more use of memory, it can be released.

Reusability: Data structures can be reused, 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 serves as the basis of abstract data types, the data structure defines the physical form of ADT(Abstract Data Type). ADT is theoretical and Data structure gives physical form to them.

1.2 Data Representation

Various methods are used to represent data in computers. Hierarchical layers of data structure are used to make the use of data structure easy and efficient. The basic unit of data representation is a bit. The value of a bit asserts on of two values-0 or 1.

Eight bits together form one byte which represents a character and one or more than one characters are used to form a string.

1.3 Abstract Data Types

• ADT stands for **Abstract Data Type.**

- It is an abstraction of a data structure.
- Abstract data type is a mathematical model of a data structure.
- It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship.
- It is a logical description of how we view the data and the operations allowed without regard to how they will be implemented.
- ADT concerns only with what the data is representing and not with how it will eventually be constructed.
- It is a set of objects and operations. For example, List, Insert, Delete, Search, Sort.

It consists of following parts:

1. Data
2. Operation

1. Data describes the structure of the data used in the ADT.
2. Operation describes valid operations for the ADT. It describes its interface.

1.4 Data Types

A data type, in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. A string, for example, is a data type that is used to classify text and an integer is a data type used to classify whole numbers.

Why do we need a data type?

We can think of a universal data type that may hold any value like character, integer, float or any complex number, use of such data type has two disadvantages:

- Large volume of memory will be occupied by even a small size of data.
- Different types of data require different interpretation of bit strings while reading or writing.

Thus, we see that the data types facilitate the optimum use of memory as well as a defined way to interpret the bit strings for different types of data.

1.5 Primitive Data Types

The following primitive data types in c are available:

Integer Data Type, int

Integer data type is used to declare a variable that can store numbers without a decimal. The keyword used to declare a variable of integer type is “int”. Thus, to declare integer data type following syntax should be followed:

int variable_name;

Float data Type, float

Float data type declares a variable that can store numbers containing a decimal number.

Syntax

float variable_name;

Double Data Type, double

Double data type also declares variable that can store floating point numbers but gives precision double than that provided by float data type. Thus, double data type are also referred to as double precision data type.

Syntax

double variable_name;

Character Data Type, char

Character data type declares a variable that can store a character constant. Thus, the variables declared as char data type can only store one single character.

Syntax

char variable_name;

Void Data Type, void

Unlike other primitive data types in c, void data type does not create any variable but returns an empty set of values. Thus, we can say that it stores null.

Syntax

void variable_name;

Data Type Qualifiers

Apart from the primitive data types mentioned above, there are certain data type qualifiers that can be applied to them in order to alter their range and storage space and thus, fit in various situations as per the requirement.

The data type qualifiers available in c are:

- short
- long
- signed
- unsigned

It should be noted that the above qualifiers cannot be applied to float and can only be applied to integer and character data types.

The entire list of data types in c available for use is given below:

C Data Types		**Size(in bytes)**	**Range**
Integer Data Types	int	2	-32,768 to 32,767
	signed int	2	-32,768 to 32,767
	unsigned int	2	0 to 65535
	short int	1	-128 to 127
	signed short int	1	-128 to 127
	unsigned short int	1	0 to 255
	long int	4	-2,147,483,648 to 2,147,483,647
	signed long int	4	Same as Above
	unsigned long int	4	0 to 4,294,967,295
Floating Point Data Types	float	4	3.4E-38 to 3.4E+38
	double	8	1.7E-308 to 1.7E+308
	long double	10	3.4E-4932 to 1.1E+4932
Character Data Types	char	1	-128 to 127
	signed char	1	-128 to 127
	unsigned char	1	0 to 255

1.6 Data Structure and Structured Type

The term **data structure** refers to a set of computers variables that are connected in some logical or mathematical manner. In another way, a **data structure** can be defined as the structural relationship present within the data set and thus should be viewed as 2 tuple,(N,R) where 'N' is the finite set of nodes representing data structure and 'R' is the set of relationship among those nodes.

A **structure type** refers to a data structure which is made up of one or more elements known as components. These elements are simpler data structures that exist in the language. The components of structured data type are grouped together according to a set of rules, for example, the representation of polynomials requires at least two components:

- Coefficient
- Exponent.

The two components together from a composite type structure to represent a polynomial.

Atomic Type

An atomic type data is a data structure that contains only the data items and not pointers. Thus, for a list of data items, several atomic type nodes may exist each with a single data item corresponding to one of the legal data types. The list is maintained using a list of node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a test node is inserted in the list, its address is stored in the next free element of the list of pointers.

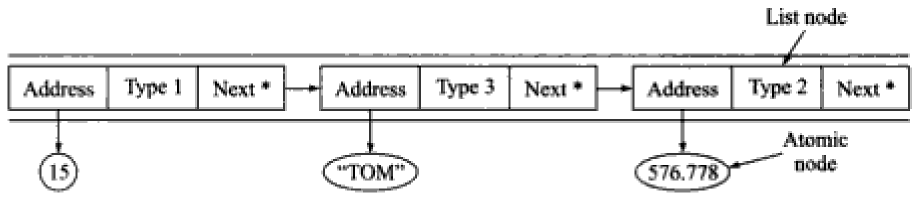


Fig above shows a list of atomic nodes maintained using list of nodes. In each node, type represents the type of data stored in the atomic node to which the list node points. 1 stands for integers type, 2 for real numbers and 3 for character type or any different assumption can be made at implementation level to indicate different data types.

1.7 Difference between Abstract Data Types, Data Types and data structures

An **abstract data type** is the specification of the data type which specifies the logical and mathematical model of the data type.

A **data type** is implementation of an abstract data type.

Data structure refers to the collection of computer variables that are connected in some specific manner.

1.8 Refinement Stages

The application or the nature of problem determines the number of refinement stages required in the specification process. Different problems have different number of refinement stages. But in general, there are four levels of refinement processes:

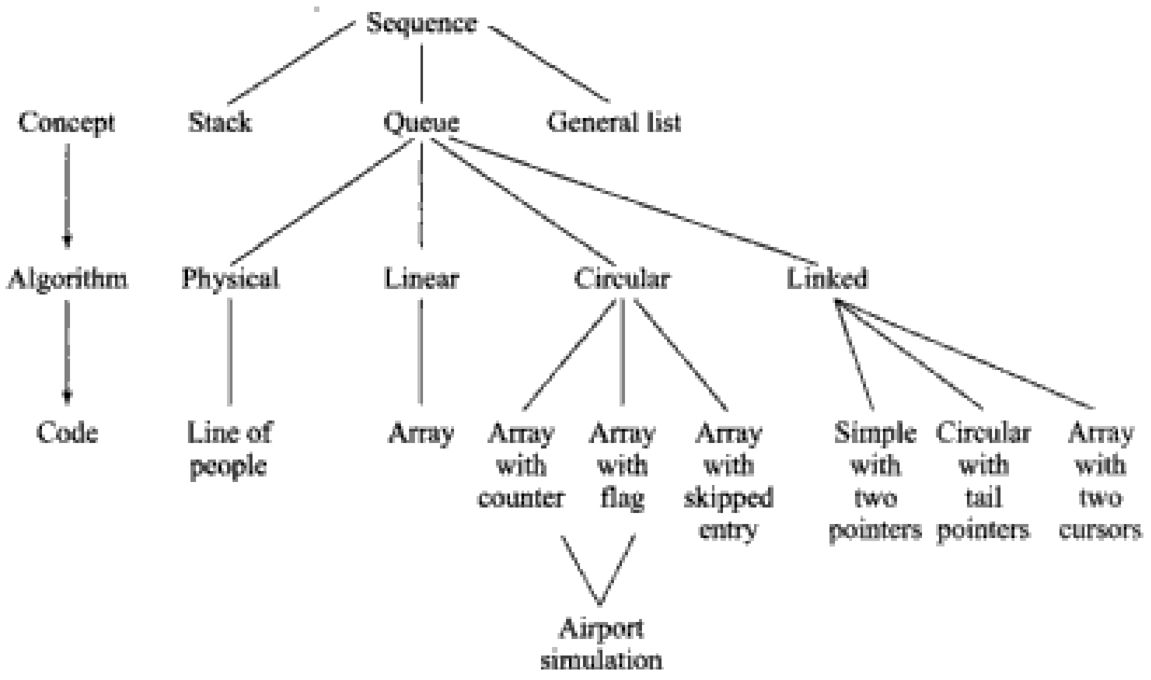
- Conceptual or abstract level
- Algorithmic or data structures
- Programming or implementation
- Applications.

Conceptual or abstract level
At this level we decide how the data is related to each other, and what operations are needed. Details about how to store data and how various operations are performed on that data are not decided at this level.

Algorithmic or implementation level
At data structure level we decide about the operations on the data as needed by our problem. For example, we decide what kind of data structure will be required to solve the problem.-contiguous list will be preferred for finding the length of a list, or for retrieving any element. Whereas for the evaluation of any expression prefix or postfix, stacks will be used.

Programming or implementation level
At implementation level, we decide the details of how the data structures will be represented in the computer memory.

Application level
This level settles all details required for particular application such as names for variables or special requirements for the operations imposed by applications.



Principles of Programming and Analysis of Algorithms: Software Engineering, Program Design, Algorithms, Different Approaches to Designing an Algorithm, Complexity, Big 'O' Notation, Algorithm Analysis, Structured Approach to Programming, Recursion, Tips and Techniques for Writing Programs in 'C'.

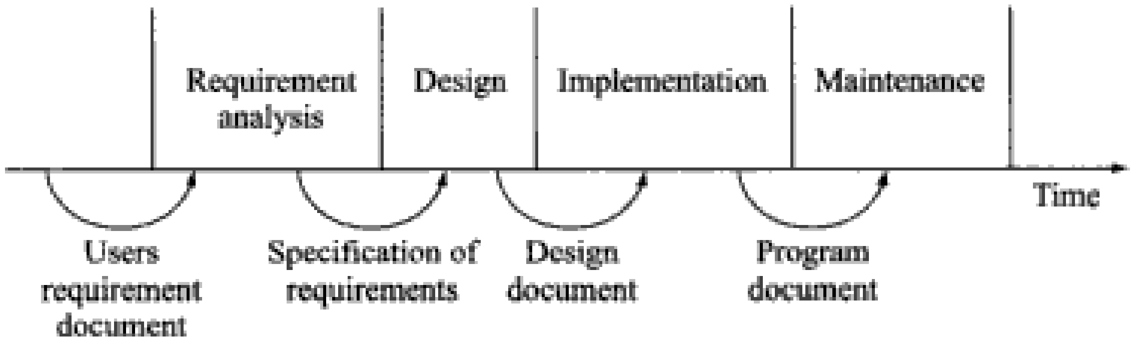
1.9 Software Engineering

Software Engineering is the theory and practice of methods helpful for the construction and maintenance of large software systems.

Development of a good software is a tedious process which continues for long time before the software or program takes the final shape and is put into use. There are many stages in the software development cycle. This process is often referred to as software development life cycle. In SDLC, the output from one stage becomes the input to the next atage.

The different steps in software development life cycle are as follows:

1. **Analyze** the problem precisely and completely.
2. **Build** a prototype and **experiment** with it until all specifications are finalized.
3. **Design** the algorithm using the tools of data structures.
4. **Verify** the algorithm, such that its correctness is self-evident.
5. **Analyze** the algorithm to determine its requirements.
6. **Code** the algorithm into an appropriate programming language.
7. **Test** and **evaluate** the program with carefully chosen data.
8. **Refine** and **repeat** the foregoing steps until the software is complete.
9. **Optimize** the code to improve performance.
10. **Maintain** the program so that it meets the changing needs of its users.



1.10 Program Design

Program design can be considered as an important phase of the software development life cycle. It is in this phase that the algorithm and data structures to solve a problem are proposed. Some of the various points that can help us evaluate the proposed program designs are as follows:

- As the design stage involves taking the specification and designing solutions to the problems, the designer needs to adopt a design strategy. The strategy adopted while designing should be according to the given specifications.

- Another important point while developing a solution strategy is that it should work correctly in all conditions.
- Generally, the people who use the system are not aware of the program design you have adopted. Thus, there is a system manual which is a detailed guide to how the design was achieved. In addition a user manual serves as a reference for the users who are not familiar with system or machines.
- A large program should be divided into small modules and sub modules by following one of the two decomposition approaches -top down approach or bottom up approach.
- Other important criteria by which a program can be judged are execution time and storage requirements..

1.11 Algorithms:

The term 'algorithm' refers to the sequence of instructions that must be followed to solve a problem. An algorithm has certain characteristics. These are as follows:

- Each instruction should be unique and concise.
- Each instruction should be relative in nature and should not be repeated infinitely.
- Repetition of same task(s) should e avoided.
- The result should be available to the user after the algorithm terminates.

Thus, an algorithm is a well defined computational procedure, along with a specified set of allowable inputs, that produce some value or set of values as output.
After an algorithm has been designed, its efficiency must be analyzed in terms of memory required by the algorithm and computational time.
The importance of an algorithm is in the correctness and program complexity.

1.12 Different Approaches to Designing an Algorithm

A complex system may be divided into smaller units called modules. Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documentation, and maintenance of the product.

A system consists of components, which have components of their own. The highest level component corresponds to the total system. To design such a hierarchy there are two possible approaches.

Top-down approach
Bottom-up approach

Top-down approach
A top down design approach starts by identifying the major components of the system or program. Decomposing them in to their lower level components and ierating until the desired level of module complexity is achieved.
Bottom-up approach
A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components.

Top-Down versus Bottom-Up Approach

Top-down Approach

- In this approach, the problem is broken down into smaller parts.

- It is generally used by structured programming languages such as C, COBOL, FORTRAN.
- It may have redundancy since every part of the code is developed separately.
- There is less interaction and communication between the modules.
- Decomposition approach is used here.
- It is generally difficult to identify.
- The implementation depends on the programming language and platform.
- It is generally used with documentation of module and debugging code.

Bottom-up Approach

- In this approach, the smaller problems are solved.
- These solved problems are integrated to find the solution to a bigger problem.
- It is generally used with object oriented programming paradigm such as C++, Java, Python.
- Data encapsulation and data hiding is implemented in this approach.
- It is generally used in testing modules.
- Composition approach is used here.

1.13 Complexity

Complexity in context of computers is a characterization of the time or space requirements for solving a problem by a particular algorithm. These requirements are expressed in terms of a single parameter that represents the size of the problem.

The complexity of a program can be considered in to types. They are

- Time complexity
- Space complexity.

Time complexity is the amount of computer time needed to run to completion.

Space complexity is the amount of memory needed to run to completion.

The space needed by the program is the sum of the following components:

Fixed space requirements: this includes the instruction space, for simple variables, fixed size structured variables, and constants.

Variable space requirements: this consists of space needed by structured variables whose size depends on particular instance of variables. It also includes the additional space required when the function uses recursion.

1.14 Big 'O' Notation

The Big O notation is used to express the upper bound of the runtime of an algorithm and thus measure the worst-case time complexity of an algorithm. It analyses and calculates the time and amount of memory required for the execution of an algorithm for an input value.

Mathematically,

For a function, **f(n)** and another function **g(n),** where both functions are defined on some unbounded set of real (positive) numbers.

Where g(n) is strictly positive for all large values of n. It can be written as:

$f(n) = O(g(n))$ where n tends to infinity **$(n \rightarrow \infty)$**

But it is seen that the assumption of n to infinity is left unstated, and so we can simply write the above expression as:

f(n) = O(g(n))

Here, f and g are the necessary functions from positive integer to non-negative real numbers.

Thus, the Big O asymptotic refers to large n values.

Properties of Big O Notation

Certain essential properties of Big O Notation are discussed below:

- **Constant Multiplication:**
  If f(n) = c.g(n), then O(f(n)) = O(g(n)) where c is a nonzero constant.
- **Summation Function:**
  If $f(n) = f_1(n) + f_2(n) + \text{--} + f_m(n)$ and $f_i(n) \leq f_i+1(n)$ ∀ i=1, 2,--, m,
  then O(f(n)) = O(max(f1(n), f2(n), --, fm(n))).
- **Polynomial Function:**
  If f(n) = a0 + a1.n + a2.n2 + -- + am.nm,
  then O(f(n)) = O(nm).
- **Logarithmic Function:**
  If f(n) = logan and g(n)=logbn,
  then O(f(n))=O(g(n))

Here, in terms of Big O, every log functions increase in the same manner.

How does Big O Notation make runtime analysis of an algorithm

For analyzing an algorithm's performance, we used to calculate and compare the worst-case running time complexities of the algorithm. The order of O(1), which is known as the **Constant Running Time**, is considered to be the fastest running time for an algorithm where the time taken by the algorithm is the same for different input sizes. However, the constant running time is the ideal runtime for an algorithm, but it is achieved very rarely. It is because the runtime of an algorithm depends on the input size of n.

For example:

As we know that the runtime performance of an algorithm depends on the input size of n. Let's see some mathematical examples for making the runtime analysis of an algorithm for different size of n:

- n = 20
  log (20) = 2.996;
  20 = 20;
  20 log (20) = 59.9;
  20^2 = 400;
  2^{20} = 1084576;
  20! = 2.432902 + 18^{18};

- n = 10
  log (10) = 1;
  10 = 10;
  10 log (10) = 10;
  10^2 = 100;
  2^{10} = 1024;
  10! = 3628800;
  Thus, similarly, we calculate the runtime performance of an algorithm. Let's see some algorithmic examples and see the runtime analysis of those algorithms:
  - For Linear Search, the runtime complexity is O(n).
  - For binary search, the runtime complexity is O(log n).
  - For Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort, the runtime complexity is O(n^c).
  - For Exponential algorithms such as Tower of Hanoi, the runtime complexity is O(c^n).
  - For Heap Sort, Merge Sort, the runtime complexity is O(n log n).

How does Big O notation analyze the Space complexity

It is essential to determine both runtime and space complexity for an algorithm. It's because on analyzing the runtime performance of the algorithm, we get to know the execution time the algorithm is taking, and on analyzing the space complexity of the algorithm, we get to know the memory space the algorithm is occupying. Thus, for measuring the space complexity of an algorithm, it is required to compare the worst-case space complexities performance of the algorithm.

In order to determine the space complexity of an algorithm, the following two tasks are necessary to be done:

Task 1: Implementation of the program for a particular algorithm is required.
Task 2: The size of the input n is required to know the memory each item will hold.

Both these are two important tasks to be accomplished first then only we can calculate the space complexity for an algorithm.

Examples of Algorithms

Below we have mentioned some algorithmic examples with their space complexities:

- For Linear Search, Bubble sort, selection sort, Heap sort, Insertion sort, and Binary Search, the space complexity is **O(1)**.
- For radix sort, the space complexity is **O(n+k)**.
- For quick SortSort, the space complexity is **O(n)**.
- For merge sort, the space complexity is **O(log n)**.
  Example of Big O Notation in C
  Below we have implemented the selection sort algorithm in C and calculated the worst-case complexity (Big O notation) of the algorithm:

```
for(int i=0; i<n; i++)
```

```
{
  int min = i;
  for(int j=i; j<n; j++)
  {
   if(array[j]<array[min])
     min=j;
  }
  int temp = array[i];
  array[i] = array[min];
  array[min] = temp;
}
```

In order to analyze the algorithm:

- We can see that the range of the for outer loop is **i < n**, which means the order of the loop is O(n).
- Next, for the inner for loop, it is also O(n) as j < n.
- The average efficiency is found n/2 for a constant c, but we ignore the constant. So, the order is O(n).
- On multiplying the order of the inner and outer loop, we get the runtime complexity as O(n^2).

1.15 Algorithm Analysis

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ.

However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- Worst-case − The maximum number of steps taken on any instance of size a.
- Best-case − The minimum number of steps taken on any instance of size a.
- Average case − An average number of steps taken on any instance of size a.

1.16 Structured Approach to Programming

The structured program consists of well structured and separated modules. But the entry and exit in a Structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

Advantages of Structured Programming Approach:

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

Disadvantages of Structured Programming Approach:

1. Since it is Machine-Independent, So it takes time to convert into machine code.
2. The converted machine code is not the same as for assembly language.
3. The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
4. Usually the development in this approach takes longer time as it is language-dependent. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.

1.17 Recursion

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called recursive function.

Example − a function calling itself.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

Example − a function that calls another function which in turn calls it again.

```
int function1(int value1) {
   if(value1 < 1)
      return;
   function2(value1 - 1);
   printf("%d ",value1);
}
int function2(int value2) {
   function1(value2);
}
```

Properties

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Tips and Techniques for Writing Programs in 'C'.

- Comments can be given anywhere in the program.
- The preprocessor directives are executed before C program code passes through the compiler.
  Ex: #define TRUE 1

```
#define FALSE 0
```

- Declaration of global variables can be accessed outside of the function main(),
- To make the program more efficient constants are used.
- **Typedef** can be used o define new data types.

  Syntax:

  ```
  Typedef type and dataname
  ```

  Here, type is the data type and dataname is the user defined name.