Chapter 1

Fundamental Notations

Learning Outcome

After reading this chapter, students will be able to

- explain the concept of problem solving
- explain the need for problem solving
- explain various approaches to problem solving
- explain the concept of structured programming
- explain the essential characteristics of an algorithm
- explain algorithm complexity
- explain the Big Oh notation
- basic terms related to data organization
- differentiate between data and information
- explain the meaning of data type and various forms of data types
- define data structure and its scope
- describe the criteria for the selection of appropriate data structure for modeling the data of a problem at hand
- differentiate between data type and data structure
- describe various data structures
- describe various operations that can be performed on data structures

1.1 INTRODUCTION TO PROBLEM SOLVING

The ability to solve problems is a most basic life skill and is essential to our day-to-day lives, at home, at school, and at the workplace.

We solve problems every day without really thinking about how we solve them.

For example, it is raining and you need to go to the market.

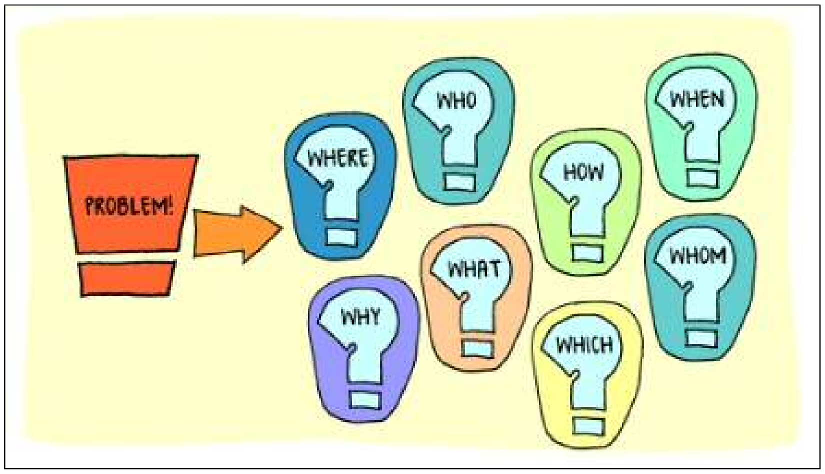
What do you do?

There are a variety of possible solutions:

- You can take your umbrella and walk down to the market.
- If you don't want to get wet, you can drive, or take the bus.
- You might decide to call a friend for a ride, or you might decide to go to the market another day.

The important point to note down here is that there is no right way to solve this problem and different people may solve it differently.

Problem solving is the process of identifying a problem, developing possible solution alternatives, and taking the appropriate course of action.



Why is problem solving important?

Good problem solving skills empower you not only in your personal life, but are very critical in your professional life.

In the currently fast-changing global economy, employers often identify everyday problem solving as crucial to the success of their organizations.

For employees, problem solving can be used to develop practical and creative solutions and to show independence and initiative to employers.

1.2 APPROACHES TO PROBLEM SOLVING

There are two approaches to problem solving:

- Top-down approach
- Bottom-up approach

1.2.1 Top-Down Approach

The basic idea of the top-down approach is to divide a complex problem into smaller sub-problems, this process is also called *decomposition*. The sub-problems are further divided into sub-problems and this process is continued until each sub-problem is atomic (can't be divided further) and can be solved independently of other sub-problems.

The top-down way of solving a program is the step-by-step process of breaking down the problem into chunks for organizing and solving the sole problem.

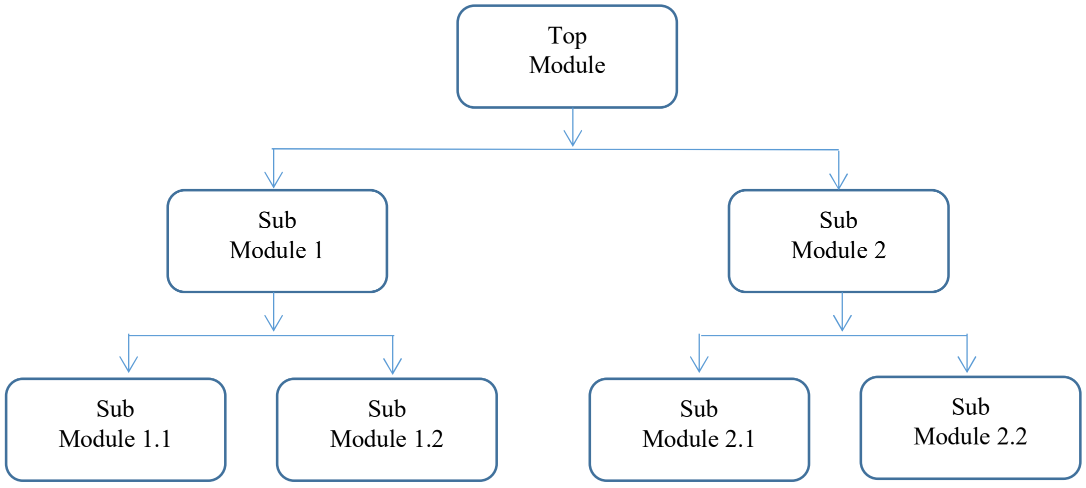


Figure 1.1: Top-down process

The structured programming languages, like the C programming language, uses the top-down approach to solving a problem in which the flow of control is in the downward direction.

1.2.2 Bottom-Up Approach

As the name suggests, this method of solving a problem works exactly opposite to the top-down approach.

In this approach, we start working from the most basic level of problem solving and moving up in conjugation of several parts of the solution to achieve the required results. The most fundamental units, modules, and sub-modules are designed and solved individually, and these units are then integrated together to get a more concrete base for problem-solving.

This bottom-up approach works in different phases or layers. Each module designed is tested at a fundamental level which means unit testing is done before the integration of the individual modules to get the solution.

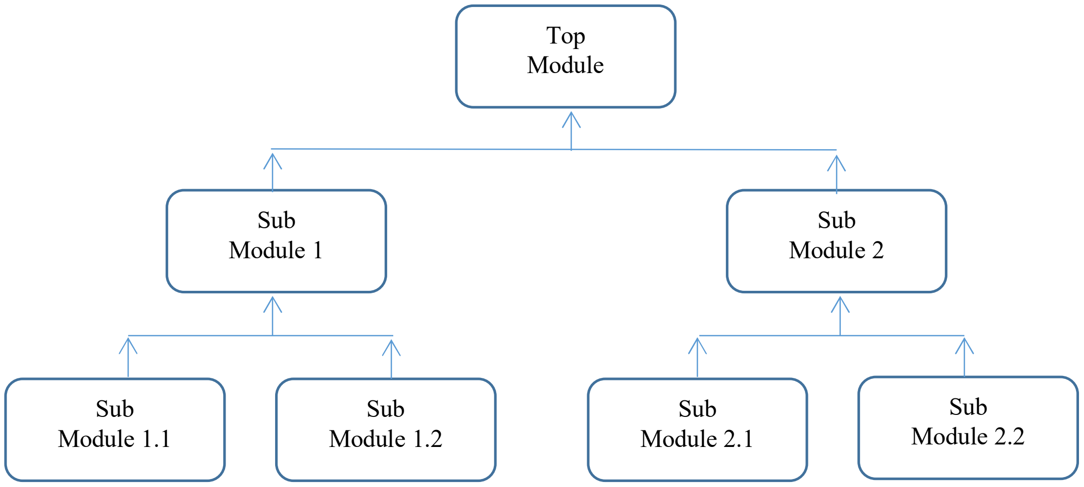


Figure 1.2: Bottom-up process

The object oriented programming languages, like the C++ or Java programming language, uses the bottom-up approach to solving a problem in which the flow of control is in the upward direction.

1.2.3 Key Differences between Top-down and Bottom-up Approach

Table 1.1 summarizes the key differences between top-down approach and bottom-up approach.

Table 1.1: Top-down V/S Bottom-up Approach

Top-down Approach	**Bottom-up Approach**
Divides a problem into smaller units and then solves it.	Starts by solving small modules and adding them up together.
This approach may contain redundant information.	Redundancy can easily be eliminated.
A well-established communication is not required.	Communication among steps is mandatory.
The individual modules are thoroughly analyzed.	Works on the concept of data-hiding and encapsulation.
Structured programming languages such as C uses a top-down approach.	OOP languages like C++ and Java etc. uses a bottom-up mechanism.
Relation among modules is not always required.	The modules must be related for better communication and workflow.
Primarily used in code implementation, test case generation, debugging, and module documentation.	Finds use primarily in testing.

The top-down approach is the conventional approach in which the decomposition of the higher-level system into a lower-level system takes place respectively while the bottom-up approach starts by designing lower abstraction modules and then integrating them into a higher-level system.

1.3 INTRODUCTION TO STRUCTURED PROGRAMMING

Structured programming is a technique devised to improve the reliability and clarity of programs.

In structured programming, control of program flow is restricted to the following three structures:

- sequence
- selection
- iteration

or to a structure derivable from a combination of these basic three structures.

Each of these structures is described below:

1.3.1 Sequence Structure

In *sequence structure*, instructions are followed or executed one after another in sequence in which they appear. The flow of logic is from top to bottom.

```
     :
     :
instruction-1

instruction-2

instruction-3
     :
     :
```

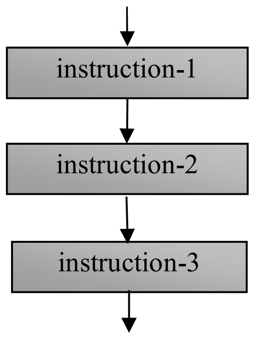


Figure 1.3: Pseudocode and flowchart for sequence structure

1.3.2 Selection Structure

Selection structure is used for making a decision. It is used for selecting a proper path out of the alternative paths in the program logic.

Selection structure may take the form as either *If . . . Endif* or *If . . . Else . . . Endif* or *If . . . Else If . . . Else . . . Endif* structure.

The *If . . . Endif* structure says that if the expression is true, then execute statement else (if the expression is false) skip over the statement.

```
    :
    :
If ( expression ) then
    statement
Endif
    :
    :
```

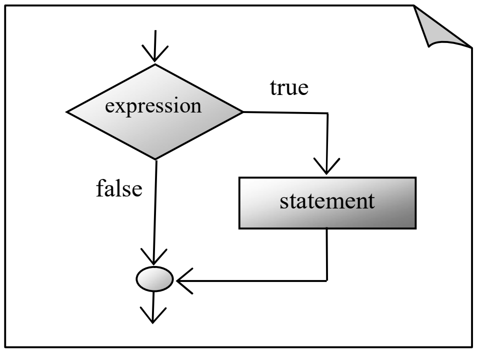


Figure 1.4: Pseudocode and flowchart for *If . . . Endif* selection structure

The *If . . . Else. . . Endif* structure says that if the expression is true then execute statement-1, else (if the expression is false) execute statement-2.

Depending on the outcome of the expression being tested, if there are multiple alternatives (execution paths), then *If . . . Else If . . . Else . . . Endif* structure is a very handy structure.

```
    :
    :
If ( expression ) then
   statement-1
Else
   statement-2
Endif
    :
    :
```

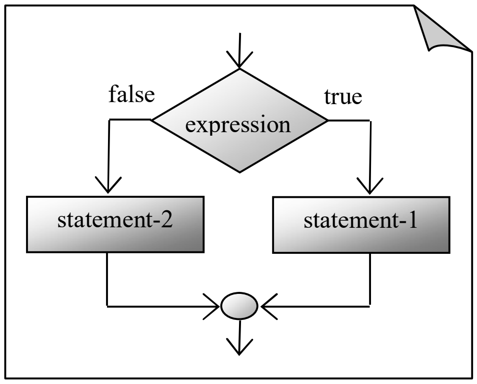


Figure 1.5: Pseudocode and flowchart for *If . . . Else . . . Endif* selection structure

```
If ( expression-1 ) then
   statement-1
Else If ( expression-2 ) then
   statement-2
Else If ( expression-3 ) then
   statement-3
      ⋮
Else If ( expression-n ) then
   statement-n
Else
   statement-s
Endif
```

Figure 1.6: Syntax of *If . . . Else If . . . Else . . . Endif* structure

The expressions are evaluated in order, and if any expression is true then the statement block associated with it is executed, and this terminates the whole chain.

The last *else* part handles *none of the above* where none of the specified expressions are satisfied.

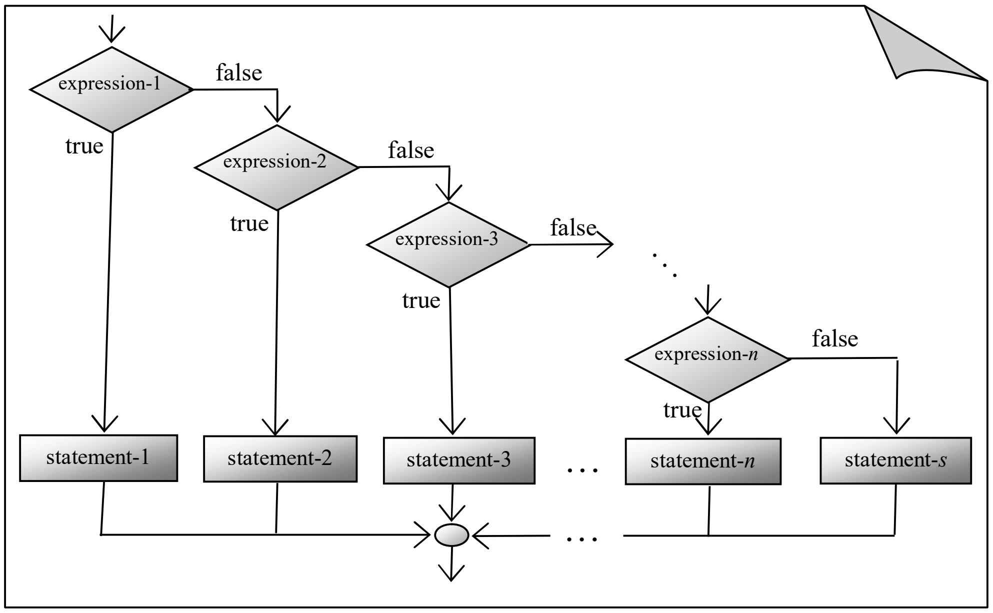


Figure 1.7: Logic flow of *If . . . Else If . . . Else . . . Endif* structure

1.3.3 Iterative Structure

The *iterative structure* is used to produce loops when one or more instructions are to be executed either a given number of times or till a certain condition is met.

The following two iterative structures are used:

- *While . . . Endwhile*
- *For . . . Endfor*

```
    :
    :
While ( expression )
   statement
Endwhile
    :
    :
```

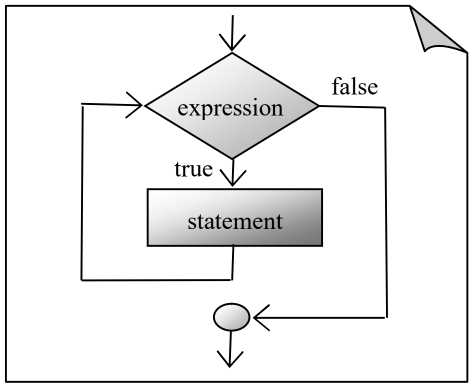


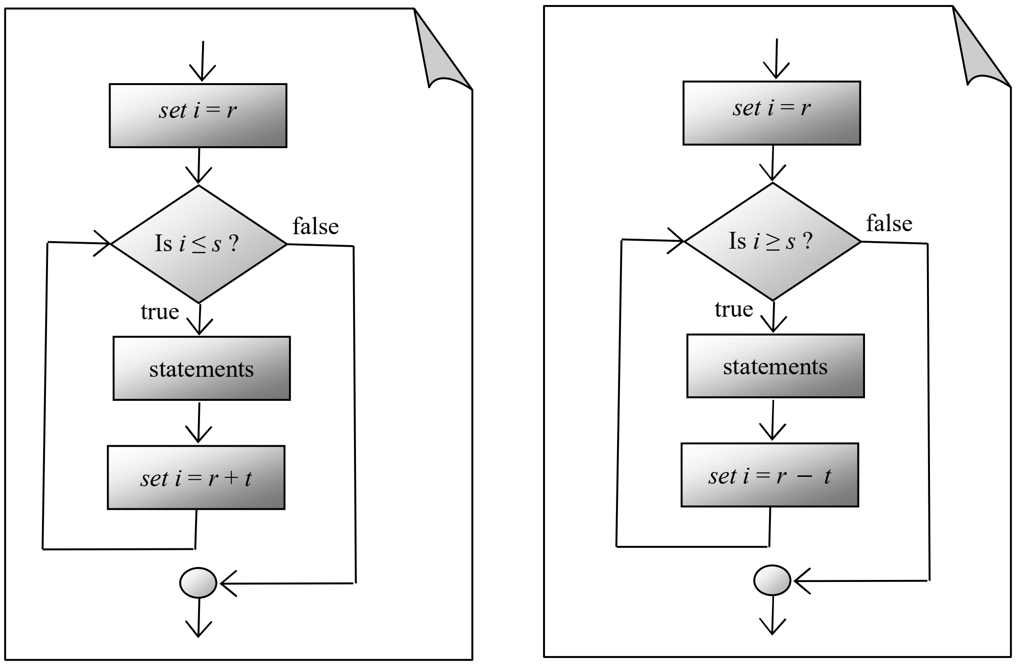
Figure 1.8: Pseudocode and flowchart for *While . . . Endwhile* iterative structure

The *While . . . Endwhile* iterative structure will continue executing until the expression is true. However, if statement or a certain group of statements are to be executed for a known number of times, the *For . . . Endfor* iterative structure is a better choice.

```
      :
      :
For i = r to s in steps of t
   statement
Endfor
      :
      :
```

Figure 1.9: Syntax for *For . . . Endfor* iterative structure

It uses an index variable *i* to control the loop. Here *r* is called the initial value, *s* is called the final value, and *t* is called the step size, which may be positive (increment) or negative (decrement).



(*a*) When step size *t* is positive (*b*) When step size *t* is negative

Figure 1.10: Working of *for* statement for positive and negative step size

The C language, which was developed around 1972, was the first language that meets all the requirements of structured programming. That is why the C language is known as a structured language.

Even after around 50 years, the C language is able to survive that shows how rich this language is. Even C language is always the first language taught to most the students including engineering students.

1.4 INTRODUCTION TO ALGORITHMS

An *algorithm* is a finite sequence of steps defining the solution of a particular problem.

Characteristics of a good algorithm:

There are five important characteristics of an algorithm that should be considered while designing an algorithm for a problem.

- ***Input*:** An algorithm must have zero or more but a finite number of inputs, which are externally supplied. An example of zero input algorithms can be to find the sum of the first 100 natural numbers. Here, the user doesn't need to supply any external input since it is already specified to find the sum of the first 100 natural numbers. However, if the above problem is re-stated as finding the sum of first *n* natural numbers, the user is required to provide single input denoting the value for *n*.
- ***Output*:** An algorithm must have at least one desirable outcome, *i.e.*, output.
- ***Definiteness (No ambiguity)*:** Each step must be clear and unambiguous, *i.e.*, having one and only one meaning.
- ***Finiteness*:** If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- ***Effectiveness*:** Each step must be sufficiently basic that it can in principle be carried out by a person using only paper and pencil. In addition, not only should each step be definite, but it must also be feasible.

An algorithm can be represented using a flowchart or pseudocode.

Since you are already familiar with flowcharts and pseudocode, we are not going to discuss these.

All the algorithms developed in the text are represented using pseudocode.

1.4.1 Algorithm Complexity

There are basically two aspects of computer programming:

- One is the data organization, *i.e.*, the data structures to represent the data of the problem in hand, and is the subject of the present text.
- The other one involves choosing the appropriate algorithm to solve the problem at hand. Data structures and algorithms are inseparably linked. Once one has developed a firm base in programming techniques required to represent information, it is logical to proceed to more theoretical study of ways to manipulate it.

As an algorithm is a sequence of steps to solve a problem, there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on the following considerations:

- Performance requirements, *i.e.*, *time complexity*
- Memory requirements, *i.e.*, *space complexity*

Performance requirements are usually more critical than memory requirements; hence, in general, it is not necessary to worry about memory unless they grow faster than performance requirements. Therefore, in general, the algorithms are analyzed only on the basis of performance requirements, *i.e.*, running-time efficiency.

1.4.1.1 Space Complexity

The space complexity of an algorithm, hence the program, is the amount of memory it needs to run to completion. Some of the reasons for studying space complexity are:

- If the program is to run on a multi-user system, it may be required to specify the amount of memory to be allocated to the program.
- We may be interested to know in advance whether sufficient memory is available to run the program.
- There may be several possible solutions with different space requirements.
- Can be used to estimate the size of the largest problem that a program can solve.

The measure the space complexity, we compute the amount of memory needed for instructions, constants, simple variables, and fixed size-structured variables.

1.4.1.2 Time Complexity

The time complexity of an algorithm is the amount of time it needs to run to completion.

Some of the reasons for studying time complexity are:

- We may be interested to know in advance whether the program will provide a satisfactory real-time response. For example, an interactive program, such as an editor, must provide such a response. If it takes even a few seconds to move the cursor one page up or down, it will not be acceptable to the user.
- There may be several possible solutions with different time requirements.

To measure the time complexity accurately, we can count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed on a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from system to system.

Our intention is to estimate the execution time of an algorithm irrespective of the computer on which it will be used. Hence, the more reasonable approach is to identify the key operation and count such operations performed until the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. The time complexity can now be expressed as a function of a number of key operations performed.

1.4.1.3 Time-Space Trade-off

The best algorithm, hence the best program, to solve a given problem is one that requires less space in memory and takes less time to complete its execution. However, in practice, it is not always possible to achieve both of these objectives. As said earlier, there may be more than one approach to solve the same problem. One such approach may require more space but takes less time to complete its execution while the other approach requires less space but takes more time to complete its execution. Thus, we may have to sacrifice one at the cost of the other. That is why we can say that there exists a *time-space* trade among algorithms.

Therefore, if space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint such as in real-time systems, we have to choose a program that takes less time to complete its execution at the cost of more space.

In the analysis of algorithms, we are interested in the *average case*, the amount of time a program might be expected to take on typical input data, and in the *worst case*, the amount of time a program would take on the worst possible input configuration.

1.4.1.4 Expressing Complexity

Space and/or time complexity is usually expressed in the form of a function *f*(*n*), where *n* is the input size for a given instance of the problem being solved.

Expressing space and/or time complexity as a function *f*(*n*) is important because of the following reasons:

- We may be interested to predict the rate of growth of complexity as the size of the problem increases.
- To compare the complexities of two or more algorithms solving the same problem in order to find which is more efficient.

The most important notation used to express this function *f*(*n*) is *Big Oh* notation, which provides the upper bound for the complexity.

Since in modern computers, memory is not a severe constraint, therefore, our analysis of algorithms will be on the basis of time complexity.

1.4.2 Big Oh Notation

Big Oh is a characterization scheme that allows measuring the properties of algorithms such as performance and/or memory requirements in a general fashion.

The algorithm complexity can be determined ignoring the implementation-dependent factors. Eliminating constant factors in the analysis of the algorithms does this. Basically, these are the constant factors that differ from computer to computer. Clearly, the complexity function *f(n)* of an algorithm increases as *n* increases. Therefore, it is the rate of increase of *f(n)* that we want to examine.

Since the purpose of *Big Oh* notation is to compare the algorithms in a general fashion, the anomalies that appear for small input sizes are ignored.

Table 1.2: Rate of growth of some standard functions

n \ *f(n)*	$\log_2 n$	n	n^2	n^3	2^n	$n\log_2 n$
5	3	5	25	125	32	15
10	4	10	100	10^3	10^3	40
100	7	100	10^4	10^6	10^{30}	700
1000	10	10^3	10^6	10^9	10^{300}	10^4

Observe that the logarithmic function $\log_2 n$ grows most slowly, whereas the exponential function 2^n grows most rapidly, and the polynomial function n^k grows according to the exponent k.

1.4.3 Categories of Algorithms

Based on *Big Oh* notation, the algorithms can be categorized as follows:

- Constant time (O(1)) algorithms
- Logarithmic time (O(log$n$)) algorithms
- Linear time (O(n)) algorithms
- Polynomial time (O(n^k), for $k > 1$) algorithms
- Exponential time (O(k^n), for $k > 1$) algorithms

Many algorithms are O($n$log$n$).

1.5 INTRODUCTION TO DATA STRUCTURES

We know that the main memory is *volatile*, *i.e.*, it loses its contents when the system is turned off, whereas the secondary memory is *non-volatile*, *i.e.*, retains its contents unless overwritten with new ones. We also know that the processor can only process the data that is available in the main memory. Therefore, if the data to be processed is not available in the main memory, then it has to be transferred from secondary memory to the main memory. Similarly, if the new data entered in the main memory under program control or intermediate and/or final results produced by the program are to be preserved for future use, they are transferred from the main memory to the secondary memory.

In order to represent and store data in main memory and/or secondary memory, we need an appropriate model. The different models used to organize data in the main memory are collectively referred to as *data structures*, whereas the different models used to organize data in the secondary memory are collectively referred to as *file structures*.

This section provides an overview of data structures and other related issues.

1.5.1 Basic Terminology of Data Organization

This section introduces the basic terminology related to data organization. Every student must understand these terms.

Data – The term *data* simply refers to a value or set of values. These values may represent some observations from an experiment, some figures collected during some survey (such as *census*, *exit polls*, etc.), marks obtained by a student in an examination, etc.

Data item – A *data item* refers to a single unit of values. For example, *roll number*, *name*, *date of birth*, *age*, *address,* and *marks in each subject* are data items. Data items that can be divided into sub-items are called *group items* whereas those that can not be divided into sub-items are called *elementary items*. For example, an address is a group item as it is usually divided into sub-items such as *house number*, *street number*, *locality*, *city*, *pin code*, etc. Likewise, a *date* can be divided into the *day*, *month* and *year*; a *name* can be divided into *first name* and *surname*. On the other hand, *roll number*, *marks*, *city*, *pin code*, etc. are normally treated as elementary items.

Entity – An *entity* is something that has certain *attributes* or *properties* which may be assigned values. The values assigned may be either numeric or non-numeric. For example, a student is an entity. The possible attributes for a student can be *roll number*, *name*, *date of birth*, *sex,* and *class*. The possible values for these attributes can be *1234*, *Surbhi*, *12/03/1993*, *F*, *9*.

Each attribute of an entity has a defined set of values, called *domain*. For example, the domain for *sex* attribute consists of only *M* (for males) and *F* (for females) values.

Entity set – An *entity set* is a collection of similar *entities.* For example, students of a class, employees of an organization, products manufactured by a manufacturing unit, etc., forms an entity set.

Record – A *record* is a collection of related data items. For example, *roll number*, *name*, *date of birth*, *sex*, and a *class* of a particular student such as *1234*, *Surbhi*, *12/03/1993*, *F*, *9*. In fact, a record represents an entity.

File – A *file* is a collection of related records. For example, a file containing records of all students in class, a file containing records of all employees of an organization. In fact, a file represents an entity set.

Key – A *key* is a data item in a record that takes unique values and can be used to distinguish a record from other records. It may happen that more than one data item has unique values. In that case, there exist multiple keys. But at a time, we may be using only one data item as a key, called the *primary key*, that too depending on the problem at hand. The other key(s) are then known as *alternate key(s)*.

In some cases, there is no field that has unique values. Then a combination of some fields can be used to form a key. Such a key is known as a *composite key*.

In the worst case, if there is no possibility of forming a key from within the record, then an extra data item can be added to the record that can be used as a key.

Information – The terms *data* and *information* have been used to mean the same thing. But actually, information is more than just data. In simple terms, information is processed data. Data is just a collection of values (raw data), from which no conclusions can be drawn. Thus data as such is not useful for decision making. When the data is processed, by applying certain rules, newly generated data becomes information. This newly generated data (information) conveys some meaning and hence can be used for decision-making. For more clarity, consider the next example.

Example 1.1: An agricultural scientist wants to study the effect of a particular pesticide on a new variety of wheat. He uses a different amount of pesticide for different fields of wheat and notes down the yield of wheat in each field. Here, the amount of pesticide used and the yield of wheat in each field represent data.

As such these values do not convey anything regarding the effect of pesticides on the yield. Next, he applies the technique of correlation analysis to determine the correlation coefficient whose value lies in the range –1 to +1. This coefficient, which is also a value, conveys valuable information about the effect of pesticides on the yield of wheat. If the correlation coefficient is greater than 0, it means an increased amount of pesticide has a positive effect on the yield. If the correlation coefficient is less than 0, it means an increased amount of pesticide has a negative effect on the yield. If the correlation coefficient is equal to 0, it means an increased amount of pesticide has no effect on the yield.

1.5.2 Concept of a Data Type

A *data type* is a collection of values and a set of operations that act on those values. Whether our program is dealing with pre-defined data types or user-defined types, the following two aspects must be considered:

- Set of values
- Set of operations

For example, consider an integer data type that consists of values {MININT, . . ., –3, –2, –1, 0, 1, 2, 3, . . ., MAXINT}, where MININT and MAXINT are the smallest and largest integers that can be represented by an integer type on a particular computer.

The operations on integers include the arithmetic operations of addition (+), subtraction (–), multiplication (*), and division (/). There are also operations that test for equality/inequality and the operation that assign an integer value to a variable.

1.5.2.1 Primitive Data Type

A *primitive data type* is a data type that is *predefined*. The predefined data type is also known as *built-in data type*. These primitive data types may be different for different programming languages. For example, C & C++ programming languages provide *built-in* support for integers (*int, long*), reals (*float, double*), and characters (*char*).

1.5.2.2 User-Defined Data Type

A *user-defined data type* is one that a user defines as per his/her own requirement. The programming languages provide support for creating user-defined data types. For various C and C++ languages support user-defined data types by means of structures (*struct*), unions (*union*), and enumerations (*enum*).

1.5.2.3 Abstract Data Type

An *abstract data type* is a data type that is organized in such a way that the specification of the values and the specification of the operations on those values are separated from the representation of the values and the implementation of the operations.

For example, consider *list* abstract data type. A *list* is basically a collection of elements that can be ordered or unordered. The primitive operations on a list may include adding new elements, deleting elements, determining the number of elements in the list, and applying a particular process to each element in the list. Here, we are not concerned with how a list is represented and how the above-mentioned operations are implemented. We only need to know that it is a list whose elements are of a given type, and what can we do with the list.

1.5.3 Concept of Variables and Constants

A *constant* is a quantity whose value remains fixed, *i.e.*, whose value does not change during the execution of the program. On the other hand, a variable is a quantity whose value may change during the execution of the program.

1.5.3.1 Constants

A *constant* that is directly encoded in the instruction, is usually, referred to as *literal*. Depending on the programming language, there can be other features to handle constants.

For example, in C language,

- a constant can be handled by assigning an identifier (name) with the literal, as shown below:

  ```
  #define PI 3.143
  ```

  The *#define* preprocessor directives associates identifier PI with the literal 3.143.

- a constant can be handled by using the *const* keyword, as shown below:

  ```
  const float PI = 3.143;
  ```

  The declaration statement declares PI as a variable of type float initializes it with value 3.143, and marks it as constant. Later on, the value of the variable PI cannot be modified.

1.5.3.2 Variables

There is three types of variables in the context of C/C++ languages:

- **Value Variables** – also called ordinary or simple variables that are used to store a value of a particular type.
- **Pointer Variables** – quite frequently referred to as *pointers*, are variables that are used to store an address of some memory location. Pointer variables play a significant role in the handling of complex data structures.
- **Reference Variables** – are aliases that are associated with some variable.

- C language supports value and pointer variables only.
- C++ support all of the above.
- References have the power of pointers and simplicity of value variables.

1.5.4 Data Structure Defined

A *data structure* is a logical model of a particular organization of data. The choice of a particular data structure depends on the following consideration:

- It must be able to represent the inherent relationship of the data in the real world.
- It must be simple enough so that it can process efficiently as and when necessary.

The study of data structures includes:

- Logical description of the data structure.
- Implementation of the data structure.
- Quantitative analysis of the data structure. This analysis includes determining the amount of memory needed to store and the time required to process it.

1.5.5 Description of Various Data Structures

The various data structures are divided into following categories:

- ***Linear data structures*** – a data structure whose elements form a sequence, and every element in the structure has a unique predecessor and unique successor. Examples of linear data structures are *arrays*, *linked lists*, *stacks,* and *queues*.
- ***Non-linear data structures*** – a data structure whose elements do not form a sequence, there is no unique predecessor or unique successor.. Examples of non-linear data structures are *trees* and *graphs*.

In the subsequent sections, we will have a look at these data structures.

1.5.5.1 Arrays

An *array* is a list of a finite number of elements of same data type, *i.e.*, integers, reals or strings etc. The individual elements of an array are accessed using an index or indices to the array. Depending on the number of indices required to access an individual element of an array, arrays can be classified as:

- *One-dimensional array* or *linear array* that requires only one index to access an individual element of the array.
- *Two-dimensional arrays* that require two indices to access an individual element of the array. In Mathematics, equivalent to a two-dimensional array, we have a *matrix* and in business terminology, we have a *table*.
- The arrays for which we need two or more indices are generally known as *multi-dimensional arrays*.

1.5.5.1.1 Linear Array

A *linear array* is a list of a finite number, say *n*, of homogeneous data elements such that

- The elements of the array are referenced by an index set consisting of *n* consecutive integer numbers.
- The elements of the array are stored in consecutive memory locations.

The number *n* of the elements is called the *size* of the linear array. In general, if *lb* is the smallest index, called the *lower bound*, and *ub* is the largest index, called *upper bound*, then size of the linear array is given by

```
size = ub – lb + 1
```

Note that the size of the array = *ub* if *lb* = 1. If not explicitly stated, generally we will consider the index set consists of 1, 2, 3, . . ., *n* or *ub*.

Suppose *a* is the name of the linear array with *n* elements, then its elements can be denoted as

$a_1, a_2, a_3, \ldots, a_n$	in mathematical notation
$a(1), a(2), a(3), \ldots, a(n)$	in BASIC and FORTRAN languages
$a[1], a[2], a[3], \ldots, a[n]$	in PASCAL language
$a[0], a[1], a[2], \ldots, a[n-1]$	in C/C++ and Java languages

The number *k* in *a*(*k*) or *a*[*k*] is called a *subscript* and *a*(*k*) or *a*[*k*] itself is called a *subscripted variable*.

To be consistent with the C language notation, if an array has *n* elements for a given dimension, its index set will be considered as 0, 1, 2, . . ., *n*–1.

Example 1.2: Consider a linear array *rn* consisting of roll numbers of 5 students of a particular class. This array can be pictured as shown in Figure 1.1.

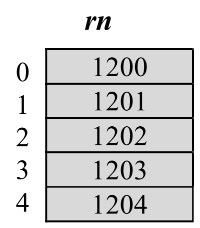


Figure 1.11: One-dimensional array *rn* storing roll numbers

Here *rn*[0] denotes 1200, *rn*[1] denotes 1201 and so on.

1.5.5.1.2 Two-dimensional Array

A two-dimensional array is a list of a finite number, say *m***n*, of homogeneous data elements such that

- The elements of the array are referenced by two index sets consisting of *m* and *n* consecutive integer numbers.

- The elements of the array are stored in consecutive memory locations.

The *size* of the two-dimensional array is denoted by $m \times n$ and pronounced as *m* by *n*.

Suppose *b* is the name of the two-dimensional array, then its element in *ith* row and *jth* column can be denoted as

b_{ij}	in mathematical notation
$b(i, j)$	in BASIC and FORTRAN languages
$b[i, j]$	in PASCAL language
$b[i][j]$	in C/C++ and Java languages

Example 1.3: Suppose a manufacturing company has a chain of five stores and five departments in each store. The weekly sales for the company can be stored using two-dimensional array named *sale*. This array can be pictured as shown in the Figure 1.2.

sale

	0	1	2	3	4
0	2000	1200	1500	1000	1005
1	1500	1450	2010	1550	1250
2	1000	1250	1400	2000	3000
3	1250	1275	1575	3500	1750
4	5200	4000	1000	1200	1575

Figure 1.12: Two-dimensional array *sale* to storing sales

Here *sale*[0][0] denotes 2000, *sale*[0][1] denotes 1200, . . ., *sale*[4][4] denotes 1575.

In business terminology, a two-dimensional array is better known as a table, whereas in mathematical terminology, it is known as a matrix.

1.5.5.2 Linked Lists

A *linked list* is a linear collection of data elements, called *nodes*. The linear order is maintained by pointers. Each node is divided into two or more parts. A linked list can be a linear linked list (one-way list) or a doubly linked list (two-way list).

1.5.5.2.1 Linear Linked List

In a linear linked list, each node is divided into two parts:

- first part contains the information about the element.
- second part, called the *link field* or *nextpointer field*, contains the address of the next node in the list.

In addition, another pointer variable, say *head*, is used that contains the address of the first element of the list. The last element of the linear linked list has *NULL* value, pictured as *X*, in the *nextpointer* field to mark the end of the list. A Linear linked list can be traversed only in one direction.

Example 1.4: The following figure shows a linear linked list with 4 nodes.

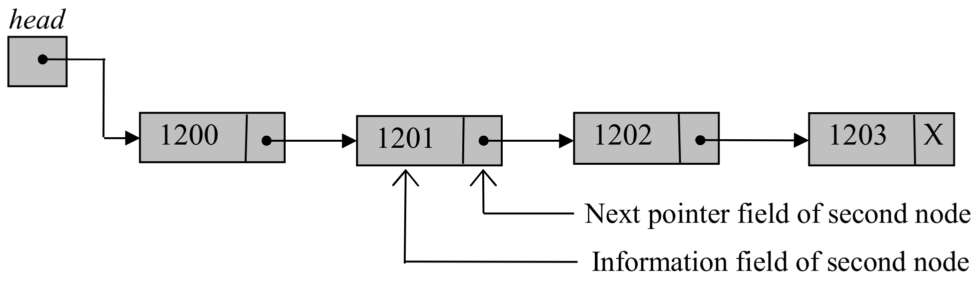

head
1200
1201
1202
1203
X
Next pointer field of second node
Information field of second node

Figure 1.13: Linear linked list with 4 nodes

1.5.5.2.2 Doubly Linked List

In the doubly linked list, each node is divided into three parts:

- o first part contains the information of the element,
- o second part, called *previouspointer field*, contains the address of the preceding element in the list, and
- o third part, called *nextpointer field*, contains the address of the succeeding element in the in the list.

Example 1.5: The following figure shows a doubly linked list with 3 nodes.

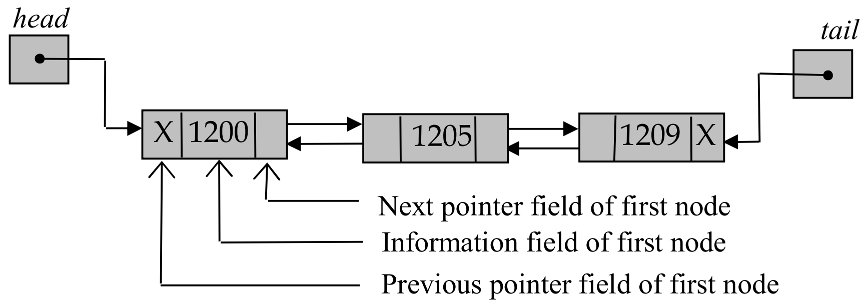

head
tail
X
1200
1205
1209
X
Next pointer field of first node
Information field of first node
Previous pointer field of first node

Figure 1.14: Doubly linked list with 3 nodes

In addition, two pointer variables, say *head* and *tail*, are used that contains the address of the first element and the last element of the doubly linked list, respectively. The first element contains *NULL* value in the *previouspointer* to indicate there is no element preceding it, and the last element of the linked list have *NULL* value in the *nextpointer* field to indicate that there is no element succeeding it. Doubly linked lists can be traversed in both directions.

1.5.5.2.3 Stack

A *stack*, also called a Last-In-First-Out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the *top*.

Example 1.6: The following figure shows a stack with 6 elements.

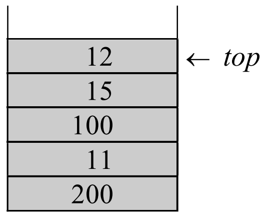


Figure 1.15: Stack with 5 elements

This structure operates in much the same way as stack of trays. If we want to place another tray, it can be placed only at the top. Likewise, if we want to remove a tray from stack of trays, it can only be removed from the top.

1.5.5.2.4 Queue

A *queue*, also called a First-In-First-Out (FIFO) or First-Come-First-Serve (FCFS) system, is a linear list in which insertions can take place at one end of the list, called the *rear* of the list, and deletions can take place only at other end, called the *front* of the list. This structure operates in much the same way a line of people waiting at a bus stop or at a cinema hall for their turn.

Example 1.7: The following figure shows a queue containing 5 elements.

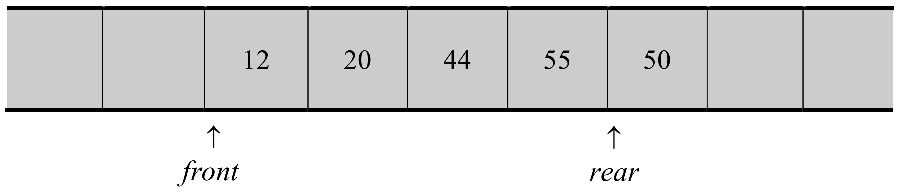


Figure 1.16: Queue with 5 elements

1.5.5.2.5 Tree

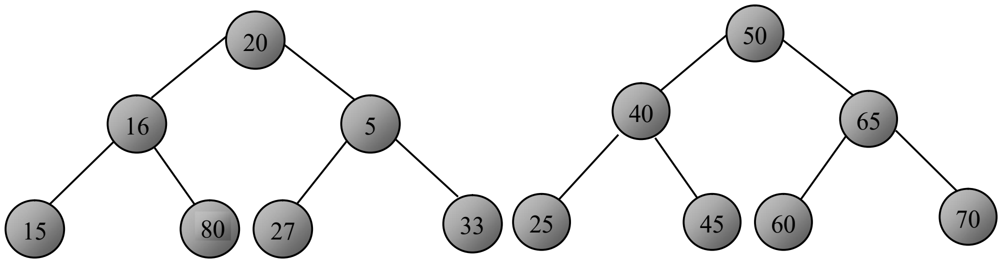
A *tree* is a data structure that represents a hierarchical relationship between various elements. A *binary tree* is a tree that can have utmost two children. Formerly, a binary tree *T* is defined as finite set of elements, called *nodes*, such that:

- Either *T* is empty, called the *null tree* or *empty tree*, or
- *T* contains a distinguished node *R*, called *root* of *T*, and remaining nodes form an ordered pair of binary trees T_1 and T_2.

Here the binary trees T_1 and T_2 are known as *left* and *right* subtrees, respectively.

A *binary search tree* is a binary tree with additional property — *the key value of any node N is larger than its left child and smaller than its right child*.

Example 1.8: The following figure shows a binary tree and a binary search tree.



(*a*) Binary tree b) Binary search tree

Figure 1.17: Binary and binary search trees

1.5.5.2.6 Heap

A *heap* is a binary tree that satisfies the following properties:

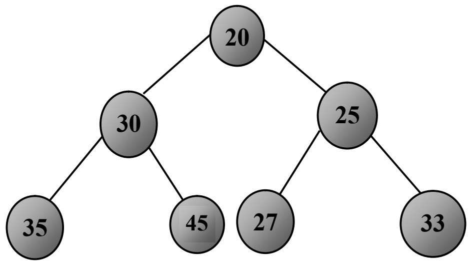
- Shape property
- Order property

The shape property states that a heap is a *complete* or *nearly complete* binary tree; and the order property states that

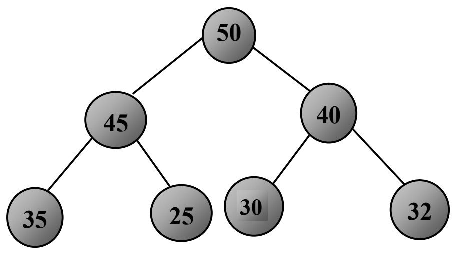
- Either the element at any node is smallest of all of its children, called *min heap*, or
- Element at any node is largest of all of its children, called *max heap*.

A heap is represented in memory by sequential representation, *i.e.*, using linear arrays. The important applications of a heap structure are to implement priority queues in addition to being used for sorting an array, a technique popularly known as *heapsort*.

Example 1.9: The following figure shows a *min heap* and a *max heap*.



(*a*) Min heap



(*b*) Max heap

Figure 1.18: Min and Max heaps

1.5.5.2.7 Graph

A *graph G* is a ordered set (*V*, *E*), where *V* represent the set of elements, called *nodes* or *vertices* in graph terminology, and *E* represents the edges between these elements. This data structure is used to represent relationship between pairs of elements, which are not necessarily hierarchical in nature. For example, graphs may be used to show the air map, where the airline flies only between the cities connected by lines as shown in Figure 1.19.

Example 1.10: The following figure shows the cities for which there is a direct flight.

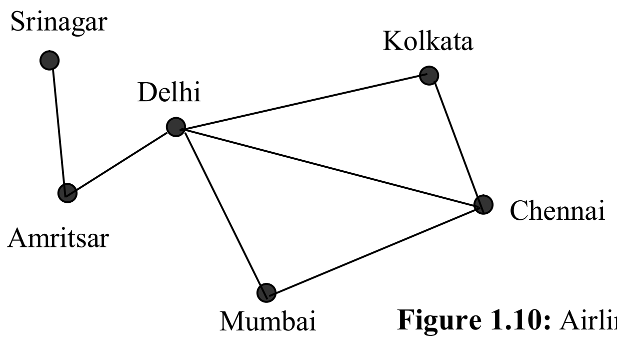


Figure 1.10: Airline Flights

Figure 1.19: Airline Flights

In the above graph

V = { Srinagar, Amritsar, Delhi, Mumbai, Chennai, Kolkata }

E = { (Srinagar, Amritsar), (Amritsar, Delhi), (Delhi, Mumbai),
(Delhi, Chennai),(Delhi, Kolkata),(Chennai, Kolkata),(Mumbai, Chennai) }

1.5.6 Common Operations on Data Structures

Various operations that can be performed on different data structures are enumerated below:

- ***Traversal*** – Accessing each element exactly once in order to process it. This operation is called *visiting* the element.
- ***Searching*** – Finding the location of a given element.
- ***Insertion*** – Adding a new element to the structure.
- ***Deletion*** – Removing a existing element from the structure.
- ***Sorting*** – Arranging the elements in some logical order. This logical order may be ascending or descending in case of numeric key. In case of alphanumeric key, it can be dictionary order.
- ***Merging*** – Combining the elements of two similar sorted structures into a single structure.

SUMMARY

In this chapter, we have learnt about the

- concept of problem solving
- need of problem solving
- basic approaches to problem solving – *Top-down* and *Bottom-up*
- concept of structured programming
- various control structures adhering to the requirement of structured programming
- notion of an algorithm
- desirable characteristics of a good algorithms
- algorithm complexity and various components – space complexity and time complexity
- big Oh notation
- various categories of algorithms
- concept of constants and variables
- type of variables
- basic terms related to data organization.
- concept of data type.
- primitive, user-defined, and abstract data types.
- concept of data structure.
- factors that influence the choice of a particular data structure for a given problem.
- brief description of various data structures with suitable examples.
- type of operations that can be performed on these data structures.

REVIEW EXERCISE

1. What is problem solving? What is the need of problems solving?
2. Name the approaches used for problem solving.
3. Describe the top-down approach of problem solving.
4. Describe the bottom-down approach of problem solving.
5. Differentiate between top-down approach and bottom-up approach to problem solving.
6. What do you mean by structured programming?
7. Describe the various control structures that meet the requirement of structured programming.
8. What is an algorithm? Describe the essential characteristics of an algorithm.
9. What is algorithm complexity?
10. How is algorithm complexity expressed?
11. What is Big Oh notation?
12. What do you understand about the time-space tradeoff?
13. Explain the terms: *data*, *elementary item*, *entity*, *primary key*, *domain*, *attribute* and *information*. Also give example in support of your answer.
14. What is the difference between *data* and *information*?
15. What is a *data type*? Differentiate between *primitive data type* and *abstract data type*.
16. What is a *data structure*? What are the factors that influence the choice of a particular data structure?
17. What are the areas of study under the domain of data structures?
18. How non-linear structures are different from linear data structures?
19. Describe, in brief, the various data structures.
20. Describe the various operations that, in general, can be performed on different data structures.

❑ ❑ ❑