Unit 1: Introduction to Data Structure and Algorithm

Notes

Structure

1.0 Introduction

A data structure is a specialized format for organizing and storing data. Some general data structure types include the array, file, record, table, tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In this unit, you will learn about data types, algorithms and complexities associated with them.

1.1 Learning Objectives

After going through this unit, you will be able to:

- discuss Primitive Data Types,
- understand Composite Data Types,
- explain Linear and Non-linear Data Structure,
- analyse Abstract Data Types,
- understand Algorithms and their Design Techniques,
- understand Time and Space Complexity of Algorithms.

1.2 Primitive and Composite Data Types

Each programming language provides various data types and each data type is represented differently within the computer's memory. The memory requirement of a data type determines the permissible range of values for that data type. The data types can be classified into several categories, including primitive data types and composite data types.

The data types provided by a programming language are known as **primitive data types** or **in-built data types**. Different programming languages provide different set of primitive data types. For example, the primitive data types in the C language are int (for integers), char (for characters), and float (for floating point numbers). The data types that are derived from primitive data types of the programming language are known as **composite data types or derived data types**. Various data types in the C language include arrays, functions, and pointers.

In addition to primitive and composite data types, programming languages allow the user to define new data types (or user-defined data types) as per his requirements. For example, the various user-defined data types as provided by C are structures, unions, and enumerations.

1.3 Abstract Data Types

Generally, handling small problems is much easier than handling comparatively larger problems. The same rule is applicable to programming also. Therefore, a large programme is decomposed into small logical units or modules, each of which does a well-defined subtask of the whole programme. The size of each module is kept as small as possible and if required, other modules are invoked from it. This modular design provides several advantages. First, several people can be employed to work on a single programme, which will increase the speed of completing the given task. Second, a well-designed modular programme has modules independent of each others, implementation, which will make the programme easily modifiable.

An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT is defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented. The data type integer is an example of the abstract data type. We frequently perform operations on integers that are associated with them like addition, subtraction, division, multiplication, modulus, etc. However, we do not know how these operations are actually performed on integers. We only know the syntax of how to perform these operations in some programming language. For example, C language defines +, –, /, ×, % to perform some basic arithmetic operations on integers.

The basic idea of ADT is that the implementation of the set of operations are written once in the programme and the part of programme which needs to perform an operation on ADT accomplishes this by invoking the required operation. If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programmes using it. The data structures, namely, linked lists, stacks, and queues are some examples of ADTs.

1.4 Data Structures

The logical or mathematical model used to organize the data in main memory is called a **data structure**. Various data structures are available, each having certain special features. These features should be kept in mind while choosing a data structure for a particular situation. Generally, the choice of a data structure depends on its simplicity and effectiveness in processing of data. In addition, we also consider how well it represents the actual relationship of the data in the real world. Data structures are divided into two categories, namely, linear data structure and non-linear data structure.

Linear Data Structures

A linear data structure is one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor. An array is the simplest linear data structure. Various other linear data structures are linked into lists, stacks, and queues.

Arrays

A finite collection of homogeneous elements is termed as an **array**. Here, the word 'homogeneous' indicates that the data type of all the elements in the collection should be same, that is, int or char or float or any other built-in or user-defined data type. However, an array cannot have elements of two or more data types together.

The elements of an array are always stored in a contiguous memory locations irrespective of the array size. The elements of an array can be referred to by using one or more indices or subscripts. An index or a subscript is a positive integer value which indicates the position of a particular element in the array. If the number of subscripts required to access any particular element of an array is one, then it is called a, **single-dimensional array**. Otherwise, it is a multi-dimensional array.

A multi-dimensional array can be a two-dimensional array or even more. Consider a single-dimensional array Arr with size *n*, where *n* is the maximum number of elements that Arr can store. Mathematically, the elements of Arr are denoted as Arr_1, Arr_2, Arr_3,…, Arr_n. In different programming languages, array elements are denoted by different notations such as by parenthesis notation or by bracket notation. **Table 1.1** shows the notation of elements of a single-dimensional array Arr with size *n* in different programming languages as follows:

Table 1.1 Different Notations of a Single-dimensional Array

S. No.	Notation	Programming Language(s)
1.	Arr(1), Arr(2), Arr(3), …, Arr(*n*)	BASIC and FORTRAN
2.	Arr[1], Arr[2], Arr[3], …, Arr[*n*]	PASCAL
3.	Arr[0], Arr[1], Arr[2], …,	Arr[*n*–1] C, C++ and Java

Note that in the languages, BASIC, PASCAL, and FORTRAN, the smallest subscript value is 1 and the largest subscript value is *n*. On the other hand, in languages like C, C++ and Java, the smallest subscript value is 0 and the largest subscript value is *n*–1. In general, the smallest subscript value that is used to access an array element is the lower bound (L_b) and the largest subscript value that is used is upper bound (U_b).

Notes

In two-dimensional arrays, the elements can be viewed as arranged in the form of rows and columns (matrix form). To access an element of a two-dimensional array, two subscripts are used—first one represents the row number and second one represents the column number. For example, consider a two-dimensional array Arr with size $m \times n$, where m and n represent the number of rows and columns, respectively. Mathematically, the array Arr is denoted as Arr_{ij}, where i and j indicate the row numbers and the column number with $i \leq m$ and $j \leq n$. **Table 1.2** shows the notation of elements of a two-dimensional array Arr in different programming languages as follows:

Table 1.2 Different Notations of Two-dimensional Array

S. Number	Notation	Programming Language(s)
1.	`Arr(i, j) with 0 ≤ i ≤ m and 0 ≤j ≤ n`	`BASIC and FORTRAN`
2.	`Arr[i, j] with 0 ≤ i ≤ m and 0 ≤ j ≤ n`	`PASCAL`
3.	`Arr[i][j] with 0 ≤ i ≤ m and 0 ≤ j ≤ n`	`C, C++ and Java`

Linked Lists

Another commonly used linear data structure is a linked list. A linked list is a linear collection of similar data elements, called **nodes**, with each node containing some data and pointer(s) pointing to other node(s) in the list. Nodes of a linked list are not constrained to be at contiguous memory locations; instead they can be stored anywhere in the memory. The linear order of the list is maintained by the pointer field(s) in each node.

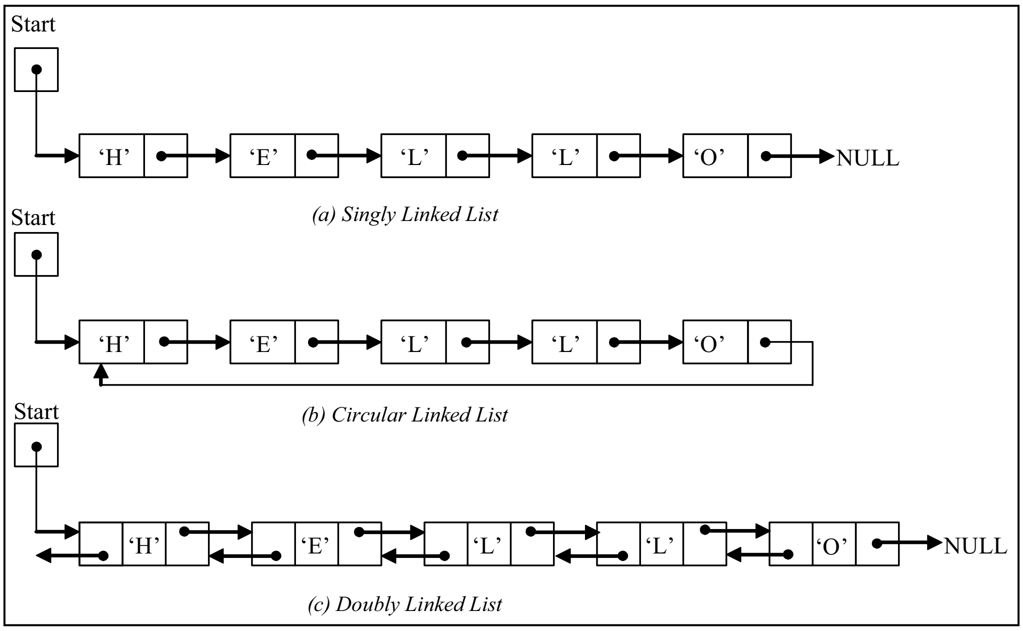

Fig. 1.1 Various Types of Linked Lists

Depending on the pointer field(s) in each node, linked lists can be of different types. If each node of a linked list contains only one pointer and it points to the next node, then it is called a **linear linked list or singly linked list**. In such type of lists, the pointer field in the last node contains NULL. However, if the pointer in the last node is modified to point to the first node of the list, then it is called a **circular linked list**.

In addition to the pointer to the next node, each node of a linked list can also contain a pointer to its previous node. This type of a linked list is called **doubly linked list. Figure 1.1** shows a singly, circular, and doubly linked list with five nodes each as follows:

Note: One major reason behind the popularity of linked lists is that it can expand or shrink during its life.

As mentioned earlier, linked lists allow the elements of the list to be stored non-contiguously. Thus, in order to access an element in the list, one has to start with the first node and follow the pointers in the nodes until the required element is found or till the end of list. In other words, linked lists allow only sequential access to their elements, meaning, in order to access the n^{th} node of the list, the n–1 preceding nodes need to be traversed.

Stacks and Queues

A stack is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end. This end is called, **'Top' of the stack**. The operation of adding a new element in the stack and deleting an element from the stack is called **push** and **pop** respectively. Since the addition and deletion of elements always occurs at one end of the stack, the last element that is pushed onto the stack is the first one to come out. Therefore, a stack is also known as a **Last-In-First-Out** (LIFO) **list**. **Figure 1.2** shows a stack with five elements and the position of top as follows:

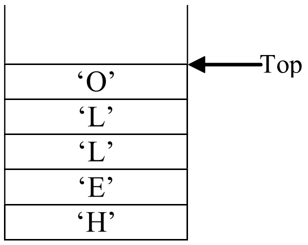

Fig. 1.2 Stack with Five Elements

A queue is a linear data structure in which the addition or insertion of a new element occurs at one end, called **Rear**, and deletion of an element occurs at other end, called **Front**. Since the insertion and deletion occur at opposite ends of the queue, the first element that is inserted in the queue is the first one to come out. Therefore, a queue is also called a **First-In-First-Out (FIFO)** list. **Figure 1.3** shows a queue with five elements and the position of Front and Rear as follows:

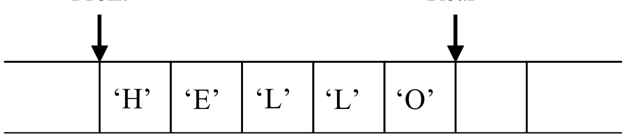

Fig. 1.3 Queue with Five Elements

Non-Linear Data Structures

A non-linear data structure is one in which its elements do not form a sequence. It means, unlike a linear data structure, each element is not constrained to have a unique the predecessor and a unique successor. Trees and graphs are the two data structures which come under this category. These data structures have been discussed in the subsequent paragraphs.

Trees

Many-a-times, we observe a hierarchical relationship between various data elements. This hierarchical relationship between data elements can be easily represented using a non-linear data structure called **trees**. A tree consists of multiple nodes, with each node containing zero, one or more pointers to other nodes called **child nodes**. Each node of a tree has exactly one parent except a special node at the top of the tree called a **root node**. An example tree with *A* as the root of the tree is shown in **Figure 1.4** as follows:

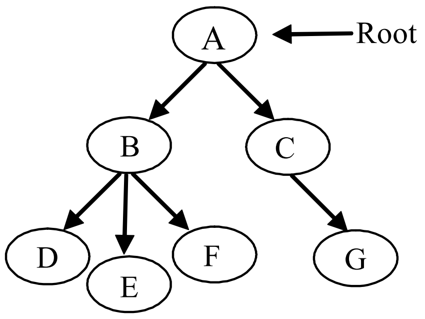

Fig. 1.4 Example of a Tree

In this tree, the root node has two child nodes *B* and *C*. In turn, the node *B* has three child nodes *D*, *E* and *F*, and the node *C* has one child *G*. The nodes *D*, *E*, *F*, and *G* have no child. The nodes without any child node are called **external nodes** or **leaf nodes**, whereas, the nodes having one or more child nodes are called **internal nodes**.

Graphs

Formally, a graph G(*V*, *E*) consists of a pair of two non-empty sets *V* and *E*, where *V* is a set of vertices or nodes and *E* is a set of edges. The graph is used to represent the non-hierarchical relationship among pairs of data elements. The data elements become the vertices of the graph and the relationship is shown by edges between the two vertices. For example, assume four places *W*, *X*, *Y*, and *Z* such that

- There exists some path from *X* to *Y*, *X* to *W*, *Y* to *W*, *Y* to *Z*, and *Z* to *W*.
- There is no direct path from *X* to *Z*.

We can simply represent this situation using a graph where the places *W*, *X*, *Y*, and *Z* are represented as the nodes of the graph and a path from one place to another place is represented by an edge between them (see **Figure 1.5**).

It is clear from Figure 1.5, that each node can have links with multiple other nodes. This analogy suggests that it is similar to a tree, however, unlike trees, there is no root node in a graph. Further, graphs show relationships which may be non-hierarchical in nature. It means there is no parent and child relationship. But, a tree can be considered as a variant or a special type of graph.

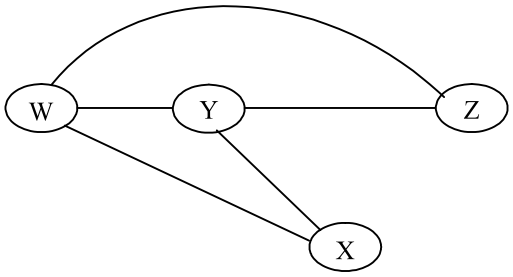

Fig. 1.5 Example of a Graph

1.5 Operations on Data Structures

As discussed earlier, the data needs to be processed by applying certain operations on it. Different data structures allow us to perform different types of operations on the data stored in them. The operations that need to be performed frequently on the data play an important role in the choice of a data structure for a particular situation. In this section, we will discuss various operations that can be performed on the data structures. They are listed as follows:

- **Traversing:** It means accessing all the data elements one by one to process all or some of them. For example, if we need to count or display the elements in a data structure, then traversing needs to be done.
- **Searching:** It is the process of finding the location of a given data element in the data structure. This operation involves one or more condition(s), and locates all the elements that satisfy the condition(s). If no element in the structure satisfies the condition(s), then the appropriate message is displayed.
- **Insertion:** It means adding a new data element in the data structure. A new element can be inserted anywhere in the structure, such as in the beginning, in the end, or in the middle. Insertion in the beginning or in the end is usually simple or straightforward. However, inserting an element somewhere in the middle requires specifying the location or the data element after (or before) which the insertion is to be done.
- **Deletion:** It means removing any existing data element from the data structure. Deletion can be performed anywhere in the structure, in the beginning, in the end, or in the middle. Deleting an element may involve first locating it by applying the search operation and then deleting it.
- **Sorting:** It is the process of arranging all the elements of a data structure in a logical order such as ascending or descending order. There are various methods that can be applied for sorting the elements.
- **Merging:** It is the process of combining the elements of two sorted data structures into a single sorted data structure. Note that both the structures to be merged should be similar.

1.6 Algorithms

In general, a problem may be defined as a state of mind of living beings to which they are not satisfied. Out of these problems, some of them can be solved with the use of computers. A solution to any solvable problem may be defined as a sequence of steps which when followed with the available (or allowed) resources lead to the satisfactory situation. A description of such a sequence of steps in some specific notation is called an **algorithm**.

Formally, an algorithm refers to a finite set of steps, when followed, solves a particular problem. Here, the word 'finite' means that the algorithm must terminate after performing a finite number of steps. An algorithm that goes on performing a set of steps infinitely is not of any use. Other than finiteness, an algorithm must have the following characteristics:

- It must take some input values supplied externally.
- It must produce some result or output.
- It must be definite, which means that all the steps in the algorithm must be clear and unambiguous.
- It must be effective, which means each step in the algorithm must be simple and basic so that any person can carryout these steps easily and effectively by using a pen and paper.

1.7 Algorithm Design Techniques

There are various techniques which can be used for designing algorithms. Some commonly used algorithm design techniques have been discussed in this section.

Brute Force Algorithm

Brute force is a general technique which is used for finding solutions to various problems. In this technique, all possible candidates for the solution are listed and then examined to check whether each candidate satisfies the problem. For example, to find the factors of a natural number *n*, it will determine the number of integers from 1 to *n* and then check all possible combinations of integers from 1 to *n*, that will form a product (equivalent to number *n*) when multiplied together.

There are various algorithms where brute force technique can be applied; some of which are selection sort, pattern matching in strings, knapsack problem, and travelling salesman problem. Let us discuss how this technique is used in pattern matching. Pattern matching is the process of determining whether a given pattern of string (say, *P*) occurs in another string (say, *S*) or not, provided the length of string *P* is not greater than that of *S*. In other words, pattern matching determines whether or not *P* is a sub-string of *S*. Using this algorithm, the string *S* is scanned character by character. Starting from the first character, each character of *S* is compared with the first character of *P*. When the match for the first character is found, the next character from the pattern string *P* is compared with the character adjacent to the searched character in string *S*. This process continues till the complete pattern string is found. If the next character does not match, string *S* is searched again for other occurrences of the first character and also the subsequent characters of the pattern string *P* in the similar manner. This process continues until a match is found or the end of pattern string *P* is reached. If a match is found, this algorithm returns the position in *S* where the pattern string *P* occurs. For example, consider a pattern string P = "ways" that is to be searched in string *S* = "hard work always pays", then this algorithm will return 13 as result.

The main advantage of brute force is that it is simple to implement. The algorithm will definitely find a solution if it exists, since it examines all possible solutions to a problem. However, the execution time of this algorithm is directly proportional to the number of solutions, that is, it increases rapidly with an increase in the size of the problem. Therefore, it is used in situations where the size of the problem is small or when some problem-specific heuristics are available that can be used to limit the number of possible solutions to a controllable size. It is also used as a baseline (an imaginary standard by which things are measured or compared) method to develop heuristics for other search algorithms.

Divide and Conquer

The divide and conquer technique is one of the widely used technique is to develop algorithms will for problems which can be divided into sub-problems (smaller in size but similar to the actual problem) so that they can be solved efficiently. The technique follows a top-down approach. To solve the problem, it recursively divides the problem into a number of sub-problems, to the extent where they cannot be sub-divided any further into more sub-problems. It then solves the sub-problems to find solutions that are then combined together to form a solution to the actual problem.

Some of the algorithms based on this technique are sorting, multiplying large numbers, syntactic analysis, etc. For example, consider the merge sort algorithm that uses the divide and conquer technique. The algorithm is composed of steps, which are as follows:

Step 1: Divide the n-element list, into two sub-lists of $n/2$ elements each, such that both the sub-lists hold half of the element in the list.

Step 2: Recursively sort the sub-lists using merge sort.

Step 3: Merge the sorted sub-lists to generate the sorted list.

Note that the merging of sub-lists starts, only when the length of sorted sub-lists (through recursive application) reach to 1. At this point, two sub-lists each of length 1 are merged (combined) by placing all the elements of the list in a sorted order.

Dynamic Programming

Dynamic programming is a technique that is generally used for solving optimisation problems where the best (optimal) solution out of the available possible solutions is to be found. One example of such a problem is the shortest path problem where, if a person in city *X* has to reach city *Z* there would be many possible routes to reach the city *Z*. The aim is to select the shortest route from all the available routes so as to reach the destination in minimum possible time. Note that in the given problem, all possible routes represent different solutions to the problem.

Using dynamic programming, when the problem is solved, there is a possibility that sub-problems of the same type may arise. The basic idea behind the technique is that it stores the solutions to such sub-problems. This helps in repeated calculation and hence, improves the efficiency of the algorithm. The algorithms that are designed using this technique consist of three steps, which are as follows:

- **Dividing the problem into simpler sub-problems:** The problem is divided into sub-problems, such that each sub-problem has a similar structure to the original problem.
- **Finding optimal solutions to sub-problems:** The solution to an original problem is computed by combining the solutions of the sub-problems. Therefore, for finding optimal solution to the original problem, the solutions to sub-problems should also be optimal.
- **Storing solutions to overlapping sub-problems:** The identified sub-problems consist of either unrelated sub-problems (each having an independent solution) or common sub-problems (having similar optimal solution). The solutions to these sub-problems are stored in a table. So, while finding optimal solutions, if any overlapping (recurring) sub-problems are found, the solutions stored in the table can be used. This increases the efficiency of the dynamic programming algorithm.

Note that dynamic programming applies a bottom-up approach to solve the problem. That is, it first finds a solution to the simplest sub-instances of the problem and then solves the more complex instances, using the results of earlier computed (sub) instances. Some of the well-known optimisation problems where the dynamic programming technique is used are knapsack problem, problem of making change, shortest path problem, and chained matrix multiplication problem.

Greedy Algorithm

We have discussed the use of dynamic programming to solve optimisation problems. However, there are many optimisation problems such as the famous minimal spanning tree problem by Kruskal, minimum number of notes problem, and activity-selection problem that can be solved more efficiently using a greedy algorithm. As we know that the algorithm for optimisation problems consist of stages, having a set of choices at each stage. The basic idea of the greedy technique is to make locally optimal choices (*i.e.*, the best choice at a particular stage) assuming that these choices will result in a globally optimal solution.

The technique follows a top-down approach. To find solution to a problem, sequence of choices is made recursively (based on minimum or maximum value criterion) from the set of given choices at each stage. After a choice is made, the problem reduces to a sub-problem (similar to the actual problem). The solution to the actual problem is found by combining the sequence of choices that are made.

The greedy technique does not always lead to an optimal solution. However, the problems that are solvable using this technique are said to possess the greedy choice property.

1.8 Time and Space Complexity of Algorithms

It is quite possible that there are one or more algorithms for solving a particular problem. If there are, they must be carefully analysed before choosing any one algorithm to be followed. The analysis of an algorithm gives us a general idea that how long it will take to solve a particular problem and what resources are required for it. Although, our aim is to choose the best algorithm, it is not always possible because of limited resources. The two main resources we consider for an algorithm are memory space and the processor time it requires.

Space complexity and time complexity are the two main measures for the efficiency of an algorithm that is considered during analysis. The space complexity of an algorithm is the maximum amount of memory space required by it at the time of execution. Frequently, the memory space required by an algorithm is the multiple the size of input. On the other hand, the time complexity of an algorithm is the amount of time required to execute it. Here, by time we are not referring to the number of seconds or minutes required, instead it is a representation of the number of operations to be performed while executing the algorithm. This is because, if time for an algorithm is measured in seconds or in such time units on one computer, then upon moving to a faster or slower computer, this analysis will become invalid.

While measuring the time for an algorithm, only significant operations are taken into consideration. Significant operations are those operations which take much time in execution. Moreover, the number of times that they are to be executed gets affected significantly with change in the size of input. For example, at the time of searching algorithms, the significant operation is comparison, which is performed to check whether the value is the one we are searching for. The number of comparisons increase with the size of the list in which the search is to be done. Other operations, like prompting whether the search is successful or not, returning the position of searched value, etc., are performed fixed number of times, irrespective of the size of the list.

Note that in a search algorithm, the number of times that the comparison is to be done is also affected by the location of value to be searched in the list. If the value is found at the first location then only one comparison needs to be done. This is the best case for an algorithm, and in this case, it has to do the least amount of work. On the other hand, if the value is found at the last location or is not present in the list at all, N comparisons will be required, where N is the size of the list. This is the worst case for the algorithm, and in this case, it has to do maximum amount of work. Another case that is also considered is an average case in which an algorithm gives average, performance. For example, in the simplest search algorithm, on an average, half the elements need to be compared.

Time-Space Tradeoff

The best or efficient algorithm is the one which utilizes minimum processor time and requires minimum memory space during its execution. However, unfortunately, it is not always possible to develop an algorithm which is efficient in terms of both space and time. Therefore, while designing an algorithm, we may have to compromise on one at the cost of the other.

Suppose, for a given problem, an algorithm is developed which takes S amount of memory and utilizes T amount of processor time during its execution. Note that S and T are not always as least as possible. However, the algorithm may be modified to minimize T at the cost of some extra memory space. Similarly, the algorithm may also be modified to minimize S at the cost of extra processor time. Therefore, if memory is the major constraint, then an algorithm which takes minimum memory space can be chosen. On the other hand, if time is the major constraint, then an algorithm which will utilize the minimum processor time can be chosen. Since with modern computers, memory is not a constraint, we mainly focus on the algorithms that utilize the minimum processor time, unless or otherwise stated.

1.9 Big O Notation

In the analysis of algorithms, the growth factor plays an important role. This is because it is quite possible that an algorithm will perform lesser number of operations than the other when the size of input is small, however, many more when the size of input gets larger. Therefore, during the analysis of algorithms, we would not be interested in determining the number of operations to be performed for some specific input, say *N*. Instead, we are interested in determining the equation that relates the number of operations to be performed to the size of the input. Once we are ready with the equations for two algorithms, we can determine the rate at which the equations will grow with growth in the size of input. **Table 1.3** shows the rate of growth for common standard functions with the increase in the input size.

Table 1.3 Rate of Growth for Common Standard Functions

Size of Input (n)	**$f(n)$**	**$f(n^2)$**	**$f(n^3)$**	**$f(2^n)$**	**$f(\log_2 n)$**	**$f(n \log_2 n)$**
1	1	1	1	2	0	0
5	5	25	125	32	2.3	11.6
10	10	100	1000	1024	3.3	33.2
20	20	400	8000	1048576	4.3	86.4
50	50	2500	125000	1125899906842624	5.6	282.2

From **Table 1.3**, it is clear that the function $\log_2^n$ is the slowest growing function and the function 2^n is the fastest growing function. One can observe that the function 2^n does not have much difference in the values with other functions when the size of the input is small. However, as the input size grows, there becomes a huge difference. To verify whether a function will grow faster or slower than the other function. We have some asymptotic or mathematical notations, which are as follows:

- **Big Omega ω(*f*):** A function $f(n)$ is $\omega(g(n))$, if there exists positive values k and c such that $f(n) \geq c*g(n)$, for all $n \geq k$. This notation defines a lower bound for a function $f(n)$.
- **Big Theta θ(*f*):** A function $f(n)$ is $\theta(g(n))$, if there exists positive values k, c_1, and c_2 such that $c_1*g(n) \geq f(n) \leq = c_2*g(n)$, for all $n \geq k$. This notation defines both a lower bound as well as an upper bound for a function $f(n)$.
- **Big Oh O(*f*):** A function $f(n)$ is $O(g(n))$, if there exists positive values k and c such that $f(n) \leq c*g(n)$, for all $n \geq k$. This notation defines an upper bound for a function $f(n)$.
- **Little oh o(*f*):** A function $f(n)$ is $o(g(n)$, if $f(n)$ is $O(g(n))$ and $f(n)$ is not $\omega(g(n))$ (that means, there exists no positive values k and c such that $f(n) \geq c*g(n)$, for all $n \geq k$.)

While comparing any two algorithms, the algorithm whose equation (that relates to the number of operations to the size of the input) grows slowly than the other, is considered better. It means that we are interested in finding that algorithm (out of the two) whose equation is in the Big Oh of another one. Such algorithm will definitely perform better than the other when the input size is large. To understand this, suppose the equation for the first and second algorithms are $f(n) = 14n^2 + 8n$ and $g(n) = 5n^3 + 3$ respectively. In order to find which algorithm works better, the values of $f(n)$ and $g(n)$ are computed for some sample values of n, which is shown in **Table 1.4** as follows:

Table 1.4 Values of *f*(*n*) and *g*(*n*) for Sample Values of n

Value of n	$f(n)$	$g(n)$
1	22	8
2	72	43
3	150	138
4	256	323
5	390	628
6	552	1083

From Table 1.4, it is clear that for all positive values of $n \geq 4$, the value of $g(n)$ is larger than $f(n)$. It implies that $f(n)$ is in $O(g(n))$ for all $n \geq 4$. Since computer programmes generally deal with large number of values, the first algorithm with complexity $f(n) = 14n^2 + 8n$ is chosen, since it works better for all $n \geq 4$.

These asymptotic notations also facilitate the recognition of essential characters of a complexity function through some simpler functions. For example, consider the function $f(n) = 2n^3 + 3n^2 + 1$. We know,

$$f(n) = 2n^3 + 3n^2 + 1$$
$$< = 2n^3 + 3n^3 + 1n^3 = 6n^3 \text{ for all } n \geq 1$$

It is found that $f(n) = O(n^3)$ with $c = 6$ and $k = 1$. It means that the function $f(n)$ has essentially the same behaviour as that of n^3, when the size of n grows and becomes larger and larger. However, computing the value of n^3 is much easier than computing the value of function $f(n)$.

1.10 Recurrences

Recurrence can be described as is an equation or inequality that defines a function in terms of its own values. It is used to express the complexity of algorithms. For example, the recurrence for the merge-sort algorithm can be expressed as follows:

$$f(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2f(n/2) + \Theta(n) & \text{if } n = 1 \end{cases}$$

Solving recurrences makes it easy to compare the complexity of any two algorithms. The recurrence can be solved by using any of the following methods:

- **Substitution method:** In this method, a reasonable guess for the solution is made and it is proved through mathematical induction.

- **Recursion tree:** In this method, recurrences are represented as a tree whose nodes indicate the cost that is incurred at the various levels of recursion.
- **Master method:** This method is used to determine the asymptotic solutions to recurrences of the specific form.

Let us see how recurrence can be solved through the master method. The master method uses the master theorem, which is as follows:

Let $a \geq 1$ and $b \geq 1$, $g(n)$ is asymptotically positive and let $f(n)$ be defined by

$$f(n) = af(n/b) + g(n)$$

Then $f(n)$ can be bounded asymptotically as follows:

1. If $g(n) = O(n^{\log_b a - \in})$ for some $\in > 0$, then $f(n) = \Theta(n^{\log_b a})$.
2. If $g(n) = \Theta(n^{\log_b a})$, then $f(n) = \Theta(\lg n)$.
3. If $g(n) = \Omega(n^{\log_b a + \in})$ for some $\in > 0$, and if $ag(n/b)cg(n)$ for some constant $c < 1$ and for all sufficiently large n, then $f(n) = \Theta(g(n))$.

In each of the above three cases, the function $f(n)$ is compared with the function $g(n)$. While the above comparison, the following three cases may arise:

1. If the function $n^{\log_b a}$ is greater than $g(n)$, then the solution is $f(n) = \Theta(n^{\log_b a})$.
2. If function is equal to $g(n)$, then the solution is $f(n) = \Theta(\lg n) = \Theta(g(n) \lg n)$.
3. If function $g(n)$ is greater than $n^{\log_b a}$, then the solution is $f(n) = \Theta(g(n))$.

For example, consider the following recurrences:

(i) $f(n) = 4f(n/2) + n$

Here, $a = 4$, $b = 2$, and $g(n) = n$

Therefore,

$n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$. Since, $g(n) = O(n^{\log_4 2 - \in})$ where $\in = 1$, apply the first case of the master theorem to conclude the solution. The solution is $f(n) = \Theta(n^2)$.

(ii) $f(n) = f(3n/4) + 1$

Here, $a = 1$, $b = 4/3$, $g(n) = 1$

log 1

Therefore, $n^{\log_b a} = n^{\log_{4/3} 1} = n^0 = 1$.

Since, $g(n) = \Theta(n^{\log_b a}) = \Theta(1)$, apply the second case of the master theorem to conclude the solution. The solution is $f(n) = \Theta(n \lg n)$.

(iii) $f(n) = 4f(n/5) + n \lg n$

Here, $a = 4$, $b = 5$, $g(n) = n \lg n$

Therefore,

$n^{\log_b a} = n^{\log_5 4}$. Since $g(n) = \Omega(n^{\log_5 4 + \in})$, where $\in \approx 0.2$, apply the third case of the master theorem to conclude the solution. The solution is $f(n) = \Theta(n \lg n)$.

1.11 Summary

- Each programming language provides various data types and each data type is represented differently within the computer's memory.

- The memory requirement of a data type determines the permissible range of values for that data type.
- The data types can be classified into several categories, including primitive data types and composite data types.
- The data types provided by a programming language are known as primitive data types or in-built data types.
- In addition to primitive and composite data types, programming languages allow the user to define new data types (or user-defined data types) as per his requirements.
- Generally, handling small problems is much easier than handling comparatively larger problems.
- The size of each module is kept as small as possible and if required, other modules are invoked from it.
- Second, a well-designed modular programme has modules independent of each other's, implementation, which will make the programme easily modifiable.
- An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT are defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented.
- The basic idea of ADT is that the implementation of the set of operations are written once in the programme and the part of programme which needs to perform an operation on ADT accomplishes this by invoking the required operation.
- If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programmes using it.
- The logical or mathematical model used to organize the data in main memory is called a **data structure**.
- These features should be kept in mind while choosing a data structure for a particular situation.
- The choice of a data structure depends on its simplicity and effectiveness in processing of data.
- Data structures are divided into two categories, namely, **linear data structure** and **non-linear data structure**.
- A linear data structure is one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor.
- A finite collection of homogeneous elements is termed as an **array**.
- The elements of an array are always stored in a contiguous memory locations irrespective of the array size.
- A stack is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end.
- A queue is a linear data structure in which the addition or insertion of a new element occurs at one end, called **Rear**, and deletion of an element occurs at other end, called **Front**.
- A tree consists of multiple nodes, with each node containing zero, one or more pointers to other nodes called **child nodes**.

Notes

1.12 Review Questions

1. Write a short note on abstract data types.
2. Write some points of differences between linear and non-linear data structures.
3. What are the different operations on data structures?
4. "The data types provided by a programming language are known as primitive data types or in-built data types. Different programming languages provide different set of primitive data types." Discuss in detail.
5. "If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programmes using it." Explain.
6. Write a detailed note on Algorithm Design Techniques.
7. What do you mean by time and space complexity of algorithms?

1.13 Further Readings

1. Patterns in C++. London: John Wiley and Sons.
2. Pandey, Hari Mohan. Data Structures and Algorithms. New Delhi: Laxmi Publications; 2009.
3. Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. Data Structures and Algorithms in Java. London: John Wiley and Sons; 2014.
4. McMillan, Michael. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press; 2007.