CHAPTER 1

INTRODUCTION TO DATA STRUCTURES

1.1 Introduction

A data structure is an efficient way of storing and organizing data elements in the computer memory. Data means a value or a collection of values. Structure refers to a method of organizing the data. The mathematical or logical representation of data in the memory is referred as a data structure. The objective of a data structure is to store, retrieve, and update the data efficiently. A data structure can be referred to as elements grouped under one name. The data elements are called members, and they can be of different types. Data structures are used in almost every program and software system. There are various kinds of data structures that are suited for different types of applications. Data structures are the building blocks of a program. For a program to run efficiently, a programmer must choose appropriate data structures. A data structure is a crucial part of data management. As the name suggests, data management is a task which includes different activities like the collection of data, the organization of data into structures, and much more. Some examples where data structures are used include stacks, queues, arrays, binary trees, linked lists, hash tables, and so forth.

A data structure helps us to understand the relationship of one element to another element and organize it within the memory. It is a mathematical or logical representation or organization of data in memory. Data structures are extensively applied in the following areas:

- Compiler Design
- Database Management Systems (DBMS)
- Artificial Intelligence
- Network and Numerical Analysis
- Statistical Analysis Packages
- Graphics
- Operating Systems (OS)
- Simulations

As we see in the previous list, there are many applications in which different data structures are used for their operations. Some data structures sacrifice speed for efficient utilization of memory, while others sacrifice memory utilization and result in faster speed. In today's world programmers aim not just to build a program but instead to build an effective program. As previously discussed, for a program to be efficient, a programmer must choose appropriate data structures. Hence, data structures are classified into various types. Now, let us discuss and learn about different types of data structures.

Frequently Asked Question

1. Define the term data structure.

Ans: *A data structure is an organization of data in a computer's memory or disk storage. In other words, a logical or mathematical model of a particular organization of data is called a data structure. A data structure in computer science is also a way of storing data in a computer so that it can be used efficiently. An appropriate data structure allows a variety of important operations to be performed using both resources, that is, memory space and execution time, efficiently.*

1.2 Types of Data Structures

Data structures are classified into various types.

1.2.1 Linear and Non-linear Data Structures

A linear data structure is one in which the data elements are stored in a linear, or sequential, order; that is, data is stored in consecutive memory locations. A linear data structure can be represented in two ways; either it is represented by a linear relationship between various elements utilizing consecutive memory locations as in the case of arrays, or it may be represented by a linear relationship between the elements utilizing links from one element to another as in the case of linked lists. Examples of linear data structures include arrays, linked lists, stacks, queues, and so on.

A non-linear data structure is one in which the data is not stored in any sequential order or consecutive memory locations. The data elements in this structure are represented by a hierarchical order. Examples of non-linear data structures include graphs, trees, and so forth.

1.2.2 Static and Dynamic Data Structures

A static data structure is a collection of datain memory which is fixed in size and cannot be changed during runtime. The memory size must be known in advance, as the memory cannot be reallocated later in a program. One example is an *array*.

A dynamic data structure is a collection of data in which memory can be reallocated during the execution of a program. The programmer can add or remove elements according to his or her need. Examples include linked lists, graphs, trees, and so on.

1.2.3 Homogeneous and Non-homogeneous Data Structures

A homogeneous data structure is one that contains data elements of the same type, for example, arrays.

A non-homogeneous data structure contains data elements of different types, for example, structures.

1.2.4 Primitive and Non-primitive Data Structures

Primitive data structures are fundamental data structures or predefined data structures which are supported by a programming language. Examples of primitive data structure types are integer, float, char, and so forth.

Non-primitive data structures are comparatively more complicated data structures that are created using primitive data structures. Examples of non-primitive data structures are arrays, files, linked lists, stacks, queues, and so on.

The classification of different data structures is shown in Figure 1.1.

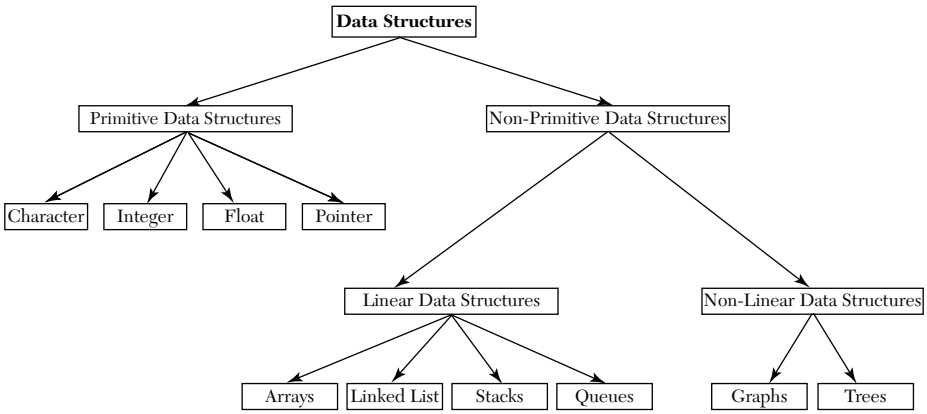

Figure 1.1. Classification of different data structures.

We know that Java supports various data structures. So, we will now introduce all these data structures, and they will be discussed in detail in the upcoming chapters.

Frequently Asked Questions

2. Write the difference between primitive and non-primitive data structures.

Ans: ***Primitive data structures:*** *The data structures that are typically directly operated upon by machine-level instructions, that is, the fundamental data types such as int, float, char, and so on, are known as primitive data structures.*

Non-primitive data structures: *The data structures which are not fundamental are called non-primitive data structures.*

3. Explain the difference between linear and non-linear data structures.

Ans: *The main difference between linear and non-linear data structures lies in the way in which data elements are organized. In a linear data structure, elements are organized sequentially, and therefore they are easy to implement in a computer's memory. In non-linear data structures, a data element can be attached to several other data elements to represent specific relationships existing among them.*

1.2.5 Arrays

An array is a collection of homogeneous (similar) types of data elements in contiguous memory. An array is a linear data structure, because all elements of an array are stored in linear order. The various elements of the array are referenced by their index value, also known as the subscript. In Java, an array is declared using the following syntax:

Syntax – `<Data type> array name [size];`

The elements are stored in the array as shown in Figure 1.2.

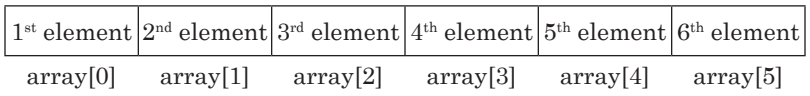

Figure 1.2. Memory representation of an array.

Arrays are used for storing a large amount of data of similar type. They have various advantages and limitations.

Advantages of using arrays

1. Elements are stored in adjacent memory locations; hence, searching is very fast, as any element can be easily accessed.
2. Arrays do not support dynamic memory allocation, so all the memory management is done by the compiler.

Limitations of using arrays

1. Insertion and deletion of elements in arrays is complicated and very time-consuming, as it requires the shifting of elements.
2. Arrays are static; hence, the size must be known in advance.
3. Elements in the array are stored in consecutive memory locations which may or may not be available.

1.2.6 Queues

A queue is a linear collection of data elements in which the element inserted first will be the element that is taken out first; that is, a queue is a FIFO (First In First Out) data structure. A queue is a popular linear data structure in which the first element is inserted from one end called the REAR end (also called the tail end), and the deletion of the element takes place from the other end called the FRONT end (also called the head).

> **Practical Application**
>
> For a simple illustration of a queue, there is a line of people standing at the bus stop and waiting for the bus. Therefore, the first person standing in the line will get into the bus first.

In a computer's memory queues can be implemented using arrays or linked lists. Figure 1.3 shows the array implementation of a queue. Every queue has FRONT and REAR variables which point to the positions where deletion and insertion are done respectively.

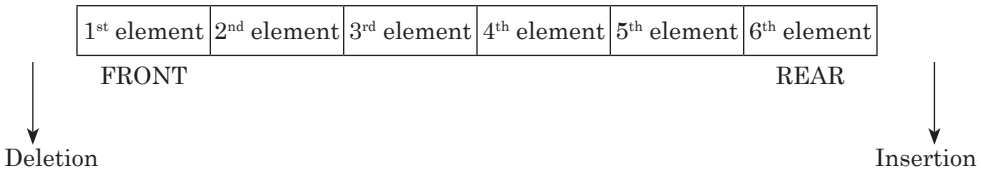

Figure 1.3. Memory representation of a queue.

1.2.7 Stacks

A stack is a linear collection of data elements in which insertion and deletion take place only at the top of the stack. A stack is a Last In First Out (LIFO) data structure, because the last element pushed onto the stack will be the first element to be deleted from the stack. The three operations that can be performed on the stack include the PUSH, POP, and PEEP operations. The PUSH operation inputs an element into the top of the stack, while the POP operation removes an element from the stack. The PEEP operation returns the value of the topmost element in the stack without deleting it from the stack. Every stack has a variable TOP which is associated with it. The TOP node stores the address of the topmost element in the stack. The TOP is the position where insertion and deletion take place.

> **Practical Application**
>
> A real-life example of a stack is if there is a pile of plates arranged on a table. A person will pick up the first plate from the top of the stack.

In a computer's memory stacks can be implemented using arrays or linked lists. Figure 1.4 shows the array implementation of a stack.

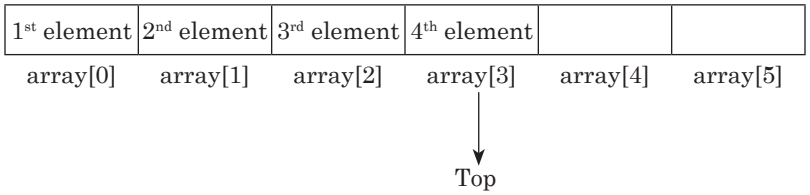

Figure 1.4. Memory representation of a stack.

1.2.8 Linked Lists

The major drawback of the array is that the size or the number of elements must be known in advance. Thus, this drawback gave rise to the new concept of a linked list. *A linked list is a linear collection of data elements.* These data elements are called nodes, and each node stores the address of the next node. *A linked list is a sequence of nodes in which each node contains one or more than one data field and an address field that stores the address of the next node.* Also, linked lists are dynamic; that is, memory is allocated as and when required.

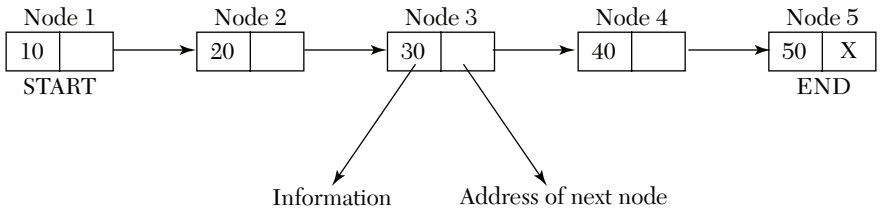

Figure 1.5. Memory representation of a linked list.

In the previous figure, we have made a linked list in which each node is divided into two slots:

1. The first slot contains the information/data.
2. The second slot contains the address of the next node.

Practical Application

A simple real-life example is a train; here each coach is connected to its previous and next coach (except the first and last coach).

The address part of the last node stores a special value called NULL, which denotes the end of the linked list. The advantage of a linked list over arrays is that now it is easier to insert and delete data elements, as we don't have to do shifting each time. Yet searching for an element has become difficult. Also, more time is required to search for an element, and it also requires high memory space. Hence, linked lists are used where a collection of data elements is required but the number of data elements in the collection is not known to us in advance.

Frequently Asked Question

4. Define the term linked list.

Ans: *A linked list or one-way list is a linear collection of data elements called nodes, which give a linear order. It is a popular dynamic data structure. The nodes in the linked list are not stored in consecutive memory locations. For every data item in a node of the linked list, there is an associated address field that gives the address location of the next node in the linked list.*

1.2.9 Trees

A tree is a popular non-linear data structure in which the data elements or the nodes are represented in a hierarchical order. Here, one of the nodes is shown as the root node of the tree, and the remaining nodes are partitioned into two disjointed sets such that each set is a part of a subtree. A tree makes the searching process very easy, and its recursive programming makes a program optimized and easy to understand.

A binary tree is the simplest form of a tree. *A binary tree consists of a root node and two subtrees known as the left subtree and the right subtree, where both subtrees are also binary trees.* Each node in a tree consists of three parts, that is, the extreme left part stores the address of the left subtree, the middle part consists of the data element, and the extreme right part stores the address of the right subtree. The root is the topmost element of the tree. When there are no nodes in a tree, that is, when ROOT = NULL, then it is called an empty tree.

For example, consider a binary tree where R is the root node of the tree. LEFT and RIGHT are the left and right subtrees of R respectively. Node A is designated as the root node of the tree. Nodes B and C are the left and right child of A respectively. Nodes B, D, E, and G constitute the left subtree of the root. Similarly, nodes C, F, H, and I constitute the right subtree of the root.

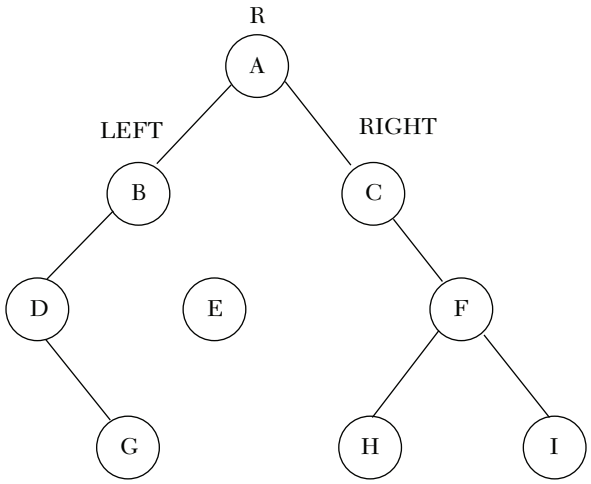

Figure 1.6. A binary tree.

Advantages of a tree

1. The searching process is very fast in trees.
2. Insertion and deletion of the elements have become easier as compared to other data structures.

Frequently Asked Question

5. Define the term binary tree.

Ans: *A binary tree is a hierarchal data structure in which each node has at most two children, that is, a left and right child. In a binary tree, the degree of each node can be at most two. Binary trees are used to implement binary search trees, which are used for efficient searching and sorting. A variation of BST is an AVL tree where the height of the left and right subtree differs by one. A binary tree is a popular subtype of a k-ary tree, where k is 2.*

1.2.10 Graphs

A graph is a general tree with no parent-child relationship. It is a non-linear data structure which consists of vertices, also called nodes, and edges which connect those vertices to one another. In a graph, any complex relationship can exist. A graph G may be defined as a finite set of V vertices and E edges. Therefore, G = (V, E) where V is the set of vertices and E is the set of edges. Graphs are used in various applications of mathematics and computer science. Unlike a root node in trees, graphs don't have root nodes; rather, the nodes can be connected to any node in the graph. Two nodes are termed as neighbours when they are connected via an edge.

Practical Application

A real-life example of a graph can be seen in workstations where several computers are joined to one another via network connections.

For example, consider a graph G with six vertices and eight edges. Here, Q and Z are neighbours of P. Similarly, R and T are neighbours of S.

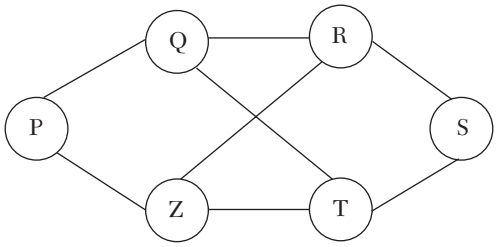

Figure 1.7. A graph.

1.3 Operations on Data Structures

Here we will discuss various operations which are performed on data structures.

- **Creation:** It is the process of creating a data structure. Declaration and initialization of the data structures are done here. It is the first operation.
- **Insertion:** It is the process of adding new data elements in the data structure, for example, to add the details of an employee who has recently joined an organization.
- **Deletion:** It is the process of removing a particular data element from the given collection of data elements, for example, to remove the name of an employee who has left the company.
- **Updating:** It is the process of modifying the data elements of a data structure. For example, if the address of a student is changed, then it should be updated.
- **Searching:** It is used to find the location of a particular data element or all the data elements with the help of a given key, for example, to find the names of people who live in New York.
- **Sorting:** It is the process of arranging the data elements in some order, that is, either an ascending or descending order. An example is arranging the names of students of a class in alphabetical order.
- **Merging:** It is the process of combining the data elements of two different lists to form a single list of data elements.
- **Traversal:** It is the process of accessing each data element exactly once so that it can be processed. An example is to print the names of all the students of a class.
- **Destruction:** It is the process of deleting the entire data structure. It is the last operation in the data structure.

1.4 Algorithms

An algorithm is a systematic set of instructions combined to solve a complex problem. It is a step-by-finite-step sequence of instructions, each of which has a clear meaning and can be executed with a minimum amount of effort in finite time. In general, an algorithm is a blueprint for writing a program to solve the problem. Once we have a blueprint of the solution, we can easily implement it in any high-level language like C, C++, Java, and so forth. It solves the problem into the finite number of steps. An algorithm written in a programming language is known as a program. A computer is a machine with no brain or intelligence. Therefore, the computer must be instructed to perform a given task in unambiguous steps. Hence, a

programmer must define his problem in the form of an algorithm written in English. Thus, such an algorithm should have following features:

1. An algorithm should be simple and concise.
2. It should be efficient and effective.
3. It should be free of ambiguity; that is, the logic must be clear.

Similarly, an algorithm must have following characteristics:

- **Input:** It reads the data of the given problem.
- **Output:** The desired result must be produced.
- **Process/Definiteness:** Each step or instruction must be unambiguous.
- **Effectiveness:** Each step should be accurate and concise. The desired result should be produced within a finite time.
- **Finiteness:** The number of steps should be finite.

1.4.1 Developing an Algorithm

To develop an algorithm, some steps are suggested:

1. Defining or understanding the problem.
2. Identifying the result or output of the problem.
3. Identifying the inputs required by the problem and choosing the best input.
4. Designing the logic from the given inputs to get the desired output.
5. Testing the algorithm for different inputs.
6. Repeating the previous steps until it produces the desired result for all the inputs.

1.5 Approaches for Designing an Algorithm

A complicated algorithm is divided into smaller units which are called modules. Then these modules are further divided into sub-modules. Thus, in this way, a complex algorithm can easily be solved. The process of dividing an algorithm into modules is called modularization. There are two popular approaches for designing an algorithm:

- Top-Down Approach
- Bottom-Up Approach

Now let us understand both approaches.

1. **Top-Down Approach:** *A top-down approach states that the complex/complicated problem/algorithm should be divided into smaller modules.* These smaller modules are further divided into sub-modules. This process of decomposition is repeated until we achieve the desired output of module complexity. A top-down approach starts from the topmost module, and the modules are incremented accordingly until level is reached where we don't require any more sub-modules, that is, the desired level of complexity is achieved.

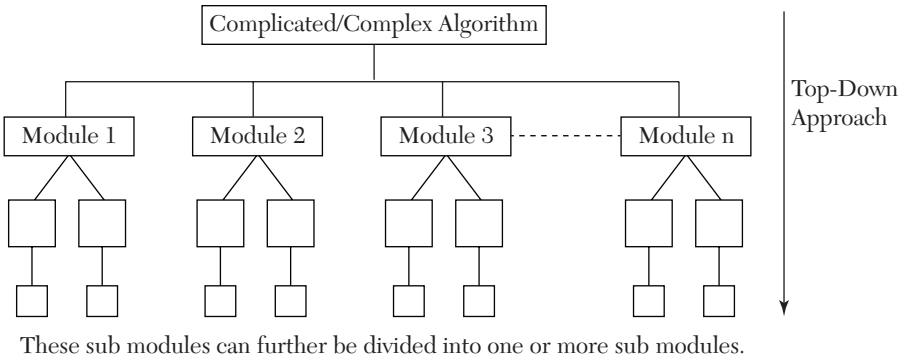

Figure 1.8. Top-down approach.

2. **Bottom-Up Approach:** A bottom-up algorithm design approach is the opposite of a top-down approach. *In this kind of approach, we first start with designing the basic modules and proceed further toward designing the high-level modules.* The sub-modules are grouped together to form a module of a higher level. Similarly, all high-level modules are grouped to form more high-level modules. Thus, this process of combining the sub-modulesis repeated until we obtain the desired output of the algorithm.

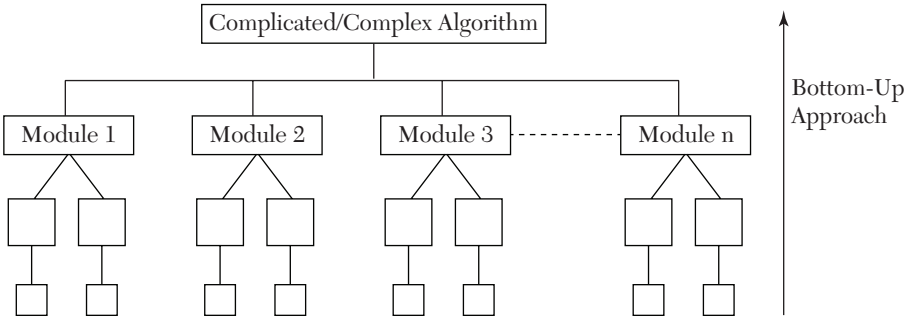

Figure 1.9. Bottom-up approach.

1.6 Analyzing an Algorithm

An algorithm can be analyzed by two factors, that is, space and time. We aim to develop an algorithm that makes the best use of both these resources. Analyzing an algorithm measures the efficiency of the algorithm. The efficiency of the algorithm is measured in terms of speed and time complexity. The complexity of an algorithm is a function that measures the space and time used by an algorithm in terms of input size.

Time Complexity: *The time complexity of an algorithm is the amount of time taken by an algorithm to run the program completely. It is the running time of the program*. The time complexity of an algorithm depends upon the input size. The time complexity is commonly represented by using big O notation. For example, the time complexity of a linear search is $O(n)$.

Space Complexity: *The space complexity of an algorithm is the amount of memory space required to run the program completely*. The space complexity of an algorithm depends upon the input size.

Time complexity is categorized into three types:

1. **Best-Case Running Time:** The performance of the algorithm will be best under optimal conditions. For example, the best case for a binary search occurs when the desired element is the middle element of the list. Another example can be of sorting; that is, if the elements are already sorted in a list, then the algorithm will execute in best time.
2. **Average-Case Running Time:** It denotes the behavior of an algorithm, when the input is randomly drawn from a given collection or distribution. It is an estimate of the running time for "average" input. It is usually assumed that all inputs of a given size are likely to occur with equal probability.
3. **Worst-Case Running Time:** The behavior of the algorithm in this case concerns the worst possible case of input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. For example, the worst case for a linear search occurs when the desired element is the last element in the list or when the element does not exist in the list.

Frequently Asked Question

6. Define time complexity

Ans: *Time complexity is a measure which evaluates the count of the operations performed by a given algorithm as a function of the size of the input. It is the approximation of the number of steps necessary to execute an algorithm. It is commonly represented with asymptotic notation, that is, O(g) notation, also known as big O notation, where g is the function of the size of the input data.*

1.6.1 Time-Space Trade-Off

In computer science, time-space trade-off is a way of solving a particular problem either in less time and more memory space or in more time and less memory space. But if we talk in practical terms, designing such an algorithm in which we can save both space and time is a challenging task. So, we can use more than one algorithm to solve a problem. One may require less time, and the other may require less memory space to execute. Therefore, we sacrifice one thing for the other. Hence, there exists a time-space or time-memory trade-off between algorithms. Thus, this time-space trade-off gives the programmer a rational choice from an informed point of view. So, if time is a big concern for a programmer, then he or she might choose a program which takes less or the minimum time to execute. On the other hand, if space is a prime concern for a programmer, then, in that case, he or she might choose a program that takes less memory space to execute at the cost of more time.

1.7 Abstract Data Types

An abstract data type (ADT) is a popular mathematical model of the data objects which define a data type alongwith various functions that operate on these objects. To understand the meaning of an abstract data type, we will simply break the term into two parts, that is, "data type" and "abstract." The data type of a variable is a collection of values which a variable can take. There are various data types in Java that include integer, float, character, long, double, and so on. When we talk about the term "abstract" in the context of data structures, it means apart from detailed specification. It can be considered as a description of the data in a structure

with a list of operations to be executed on the data within the structure. Thus, an abstract data type is the specification of a data type that specifies the mathematical and logical model of the data type. For example, when we use stacks and queues, then at that point of time our prime concern is only with the data type and the operations to be performed on those structures. We are not worried about how the data will be stored in the memory. Also, we don't bother about how push () and pop () operations work. We just know that we have two functions available to us, so we have to use them for insertion and deletion operations.

1.8 Big O Notation

The performance of an algorithm, that is, time and space requirements, can be easily compared with other competitive algorithms using asymptotic notations such as big O notation, Omega notation, and Theta notation. The algorithmic complexity can be easily approximated using asymptotic notations by simply ignoring the implementation-dependent factors. For instance, we can compare various available sorting algorithms using big O notation or any other asymptotic notation.

Big O notation is one of the most popular analysis characterization schemes, since it provides an upper bound on the complexity of an algorithm. In big O, *O(g)* is representative of the class of all functions that grow no faster than *g*. Therefore, if *f(n) = O(g(n))* then $f(n) <= c(g(n))$ for all $n > n_0$, where n_0 represents a threshold and *c* represents a constant.

An algorithm with O(1) complexity is referredto as a constant computing time algorithm. Similarly, an algorithm with $O(n)$ complexity is referredto as a linear algorithm, $O(n^2)$ for quadratic algorithms, $O(2^n)$ for exponential time algorithms, $O(n^k)$ for polynomial time algorithms, and O (log n) for logarithmic time algorithms.

An algorithm with complexity of the order of $O(\log_2 n)$ is considered as one of the best algorithms, while an algorithm with complexity of the order of $O(2^n)$ is considered as the worst algorithm. The complexity of computations or the number of iterations required in various types of functions may be compared as follows:

$$O(\log_2 n) < O(n) < O\ (n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

1.9 Summary

- A data structure determines a way of storing and organizing the data elements in the computer memory. Data means a value or a collection of values. Structure refers to a way of organizing the data. The mathematical or logical representation of data in the memory is referred as a data structure.
- Data structures are classified into various types which include linear and non-linear data structures, primitive and non-primitive data structures, static and dynamic data structures, and homogeneous and non-homogeneous data structures.
- A linear data structure is one in which the data elements are stored in a linear or sequential order; that is, data is stored in consecutive memory locations. A non-linear data structure is one in which the data is not stored in any sequential order or consecutive memory locations.

- A static data structure is a collection of data in memory which is fixed in size and cannot be changed during runtime. A dynamic data structure is a collection of data in which memory can be reallocated during execution of a program.
- Primitive data structures are fundamental data structures or predefined data structures which are supported by a programming language. Non-primitive data structures are comparatively more complicated data structures that are created using primitive data structures.
- A homogeneous data structure is one that contains all data elements of the same type. A non-homogeneous data structure contains data elements of different types.
- An array is a collection of *homogeneous* (similar) types of data elements in *contiguous* memory.
- A queue is a linear collection of data elements in which the element inserted first will be the element taken out first, that is, a FIFO data structure. A queue is a linear data structure in which the first element is inserted from one end called the REAR end and the deletion of the element takes place from the other end called the FRONT end.
- A linked list is a sequence of nodes in which each node contains one or more than one data field and an address field that stores the address of the next node.
- A stack is a linear collection of data elements in which insertion and deletion take place only at one end called the TOP of the stack. A stack is a Last In First Out (LIFO) data structure, because the last element added to the top of the stack will be the first element to be deleted from the top of the stack.
- A tree is a non-linear data structure in which the data elements or the nodes are represented in a hierarchical order. Here, an initial node is designated as the root node of the tree, and the remaining nodes are partitioned into two disjointed sets such that each set is a part of a subtree.
- A binary tree is the simplest form of a tree. A binary tree consists of a root node and two subtrees known as the left subtree and right subtree, where both the subtrees are also binary trees.
- A graph is a general tree with no parent-child relationship. It is a non-linear data structure which consists of vertices or nodes and the edges which connect those vertices with one another.
- An algorithm is a systematic set of instructions combined to solve a complex problem. It is a step-by-finite-step sequence of instructions, each of which has a clear meaning and can be executed in a minimum amount of effort in finite time.
- The process of dividing an algorithm into modules is called modularization.
- The time complexity of an algorithm is described as the amount of time taken by an algorithm to run the program completely. It is the running time of the program.
- The space complexity of an algorithm is the amount of memory space required to run the program completely.
- An ADT (Abstract Data Type) is a mathematical model of the data objects which define a data type as well as the functions to operate on these objects.
- Big O notation is one of the most popular analysis characterization schemes, since it provides an upper bound on the complexity of an algorithm.

1.10 Exercises

1.10.1 Theory Questions

1. How do you define a good program?
1. Explain the classification of data structures.
1. What is an algorithm? Discuss the characteristics of an algorithm.
1. What are the various operations that can be performed on data structures? Explain each of them with an example.
1. Differentiate an array with a linked list.
1. Explain the terms time complexity and space complexity.
1. Write a short note on graphs.
1. What is the process of modularization?
1. Differentiate between stacks and queues with examples.
1. What is meant by abstract data types (ADT)? Explain in detail.
1. How do you define the complexity of an algorithm? Discuss the worst-case, best-case, and average-case time complexity of an algorithm.
1. Write a brief note on trees.
1. Explain how you can develop an algorithm to solve a complex problem.
1. Explain time-memory trade-off in detail.

1.10.2 Multiple Choice Questions

1. Which of the following data structures is a FIFO data structure?
   (*a*) Array (*b*) Stacks
   (*c*) Queues (*d*) Linked List
2. How many maximum children can a binary tree have?
   (*a*) 0 (*b*) 2
   (*c*) 1 (*d*) 3
3. Which of the following data structures uses dynamic memory allocation?
   (*a*) Graphs (*b*) Linked Lists
   (*c*) Trees (*d*) All of these
4. In a queue, deletion is always done from the _____.
   (*a*) Front end (*b*) Rear end
   (*c*) Middle (*d*) None of these
5. Which data structure is used to represent complex relationships between the nodes?
   (*a*) Linked Lists (*b*) Trees
   (*c*) Stacks (*d*) Graphs

6. Which of the following is an example of a heterogeneous data structure?
   (*a*) Array (*b*) Structure
   (*c*) Linked list (*d*) None of these
7. In a stack, insertion and deletion takes place from the _____.
   (*a*) Bottom (*b*) Middle
   (*c*) Top (*d*) All of these
8. Which of the following is not part of the Abstract Data Type (ADT) description?
   (*a*) Operations (*b*) Data
   (*c*) Both (a) and (b) (*d*) None of the above
9. Which of the following data structures allows deletion at one end only?
   (*a*) Stack (*b*) Queue
   (*c*) Both (a) and (b) (*d*) None of the above
10. Which of the following data structures is a linear type?
    (*a*) Trees (*b*) Graphs
    (*c*) Queues (*d*) None of the above
11. Which one of the following is beneficial when the data is stored and has to be retrieved in reverse order?
    (*a*) Stack (*b*) Linked List
    (*c*) Queue (*d*) All of the above
12. A binary search tree whose left and right subtree differ in height by 1 at most is a _____.
    (*a*) Red Black Tree (*b*) M way search tree
    (*c*) AVL Tree (*d*) None of the above
13. The operation of processing each element in the list is called _____.
    (*a*) Traversal (*b*) Merging
    (*c*) Inserting (*d*) Sorting
14. Which of the following are the two primary measures of the efficiency of an algorithm?
    (*a*) Data & Time (*b*) Data & Space
    (*c*) Time & Space (*d*) Time & Complexity
15. Which one of the following cases does not exist/occur in complexity theory?
    (*a*) Average Case (*b*) Worst Case
    (*c*) Best Case (*d*) Minimal Case