Comprehensive Data Structures and Algorithms in C++

Learn fundamentals with 500+ code samples and problems

Suresh Kumar Srivastava

Deepali Srivastava

www.bpbonline.com

First Edition 2025

ISBN: 978-93-65898-576

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Dedicated to

Our teachers, family and friends

About the Authors

- **Suresh Kumar Srivastava** has 20+ years of experience in software industry - Alcatel, BNY Mellon, Unisys and has worked on architecture and design of multiple products. He is the author of popular books, *C in Depth* and *Data Structures Through C in Depth*, which have helped over 250,000 students. He has completed his 'B' level at NIELIT. He has worked on the development of Compilers, Linkers, Debuggers, IDEs, System Utilities, System Management and Telecom/Mobile/ Systems tools. He runs an online learning site called CourseGalaxy, and loves doing software architecture, design, coding, and product engineering.

- **Deepali Srivastava** has a Master's degree in Mathematics and is an author and educator in computer science and programming. Her books *Ultimate Python Programming*, *C in Depth* and *Data Structures Through C in Depth* are widely used as reference materials by students, programmers and professionals looking to enhance their understanding of programming languages and data structures. In addition to her writing, Deepali Srivastava has been involved in creating online video courses on Data structures and Algorithms, Linux, and Python programming. Her books and courses have helped over 350,000 students learn computer science concepts.

Acknowledgements

We are grateful to our teachers for making an excellent foundation for us to do better in life. We also thank our family and friends for encouraging us to do better. Our special thanks to BPB Publications for considering our work and making it available to students worldwide. We thank software organizations and academic institutions for providing excellent exposure to software development, industry standards and valuable learning on theoretical concepts, technologies, and processes. We also thank students for learning and appreciating our previous works. This encourages us to come up with more and share knowledge wherever possible.

Preface

Data Structures and Algorithms is an essential subject in any university curriculum for the computer science stream. It provides an excellent tool for software engineers and plays a significant role in software design and development. It is also becoming a must-have skill for many competitions and job interviews in the software industry. Selection of appropriate data structures and algorithms makes the software better. Software developers are always advised to use them appropriately to provide better solutions. Understanding data structures and algorithms makes them better software developers and designers. The book will benefit students in their university curriculum and open opportunities to enter and excel in the software industry.

This book provides an extensive study of data structures and algorithms. The book has various topics - algorithms analysis, arrays, linked lists, stacks and queues, recursion, trees, graphs, sorting, searching, hashing, and storage management, to learn in-depth data structures and algorithms. The book provides a good understanding of concepts with implementation. Each concept is explained with well-defined steps, figures to understand it better, and immediate code samples to understand implementation for concepts. Complete programs are provided for better understanding of concepts and implementation. The book provides 500+ illustrations, examples, code samples, and problems to learn fundamentals and understand concepts and implementation well. There are exercise problems that strengthen the learning of concepts and implementation. The problems force the students to have a better thought process and solve the problems using concepts, and develop multiple solutions. It also helps them have better problem-solving skills and learn how to implement them. There is a comprehensive chapter that covers recursion in detail. This allows students to develop a recursive approach to problem-solving and enhances their overall thought process for solving problems.

The book is written in a straightforward language, carefully explaining concepts in a way that is easy to understand for both students and experienced engineers. Anyone with a basic understanding of computer science will be able to understand the Data Structures and Algorithms concepts. The implementation requires a basic knowledge of object-oriented programming in C++. The book provides good learning for students as well as experienced engineers. It is recommended that students begin with the first chapter, as some concepts will be used in subsequent chapters. The practical learning process involves first understanding the concept, then grasping its implementation, and finally applying each concept in a program. At the end of each chapter, explore the exercise problems to strengthen your understanding of concepts and implementation. The programs follow coding conventions and include comments to better understand the code and logic. All the programs of the chapter and the exercise solutions are provided. It is always suggested that the reader first try implementing the concepts and solving the exercises independently, and only then refer to the provided programs and solutions.

Here is the brief information of all the chapters of the book:

Chapter 1: Introduction – This chapter introduces data structures, algorithms, and methods to analyze the efficiency of algorithms

Chapter 2: Arrays – This chapter covers the array, its operations, and matrices.

Chapter 3: Linked Lists - This chapter discusses various linked lists and their operations in detail.

Chapter 4: Stacks and Queues – This chapter explores stacks and queues and their applications.

Chapter 5: Recursion - This chapter explains recursion in detail with many problems.

Chapter 6: Trees - This chapter covers a variety of trees and their operations.

Chapter 7: Graphs - This chapter explores graphs and various graph algorithms.

Chapter 8: Sorting - This chapter covers different sorting algorithms in detail with their efficiency.

Chapter 9: Searching and Hashing - This chapter explains searching and hashing techniques.

Chapter 10: Storage Management - This chapter introduces storage management and its different methods.

We hope the book will provide students with good learning opportunities and help them in their college curriculum and software development.

Suresh Kumar Srivastava

Deepali Srivastava

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/c381f9

The code bundle for the book is also hosted on GitHub at
https://github.com/bpbpublications/Comprehensive-Data-Structures-and-Algorithms-in-C-Plus-Plus.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1 Introduction

Generally, any problem that has to be solved by the computer involves the use of data. If data is arranged in some systematic way, then it gets a structure and becomes meaningful. This meaningful or processed data is called information. It is essential to manage data in such a way that it can produce information. There can be many ways in which data may be organized or structured. To provide an appropriate structure to your data, you need to know about data structures. Data structures can be viewed as a systematic way to organize data, so that it can be used efficiently. The choice of proper data structures can greatly affect the efficiency of our program.

The process of program development generally requires us to represent the data efficiently and develop a step-by-step procedure that performs a given task and can be used to implement a program.

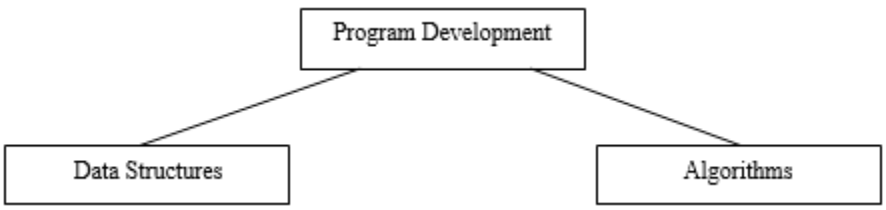

Figure 1.1 Program development

Representing the data efficiently needs the study of data structures, and the development of a step-by-step procedure requires learning about algorithms.

1.1 Data Type

A data type defines a domain of allowed values and the operations that can be performed on those values. For example, in C++, the `int` data type can take values in a range and operations that can be performed are addition, subtraction, multiplication, division, bitwise operations etc. Similarly, the data type `float` can take values in a particular range, and the operations allowed are addition, subtraction, multiplication, division, and so on (% operation and bitwise operations are not allowed). These are built-in or primitive data types and the values and operations for them are defined in the language.

If an application needs to use a data type other than the primitive data types of the language, i.e., a data type for which values and operations are not defined in the language itself, then it is the programmer's responsibility to specify the values and operations for that data type and implement it. For example, there is no built-in type for dates in C++, and if we need to process dates, we have to define and implement a data type for date.

1.2 Abstract Data Types

Abstract Data Type (ADT) is a mathematical model or concept that defines a data type logically. It specifies a set of data and a collection of operations that can be performed on that data. The definition of ADT only mentions what operations are to be performed but not how these operations will be

implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of a data type need not know how the data type is implemented; for example, we have been using `int`, `float`, `char` data types only with the knowledge of values that they can take and operations that can be performed on them without any idea of how these types are implemented. So, a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box that hides the inner structure and design of the data type. Now, we will define three ADTs, namely List ADT, Stack ADT, and Queue ADT.

List ADT

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

- `initialize( )`: Initialize the list to be empty.
- `get( )`: Return an element from the list at any given position.
- `insert( )`: Insert a new element at any list position.
- `remove( )`: Remove the first occurrence of any element from a non-empty list.
- `removeAt( )`: Remove the element at a specified location from a non-empty list.
- `replace( )`: Replace an element at any position with another element.
- `size( )`: Return the number of elements in the list.
- `isEmpty( )`: Return true if the list is empty; otherwise, return false.
- `isFull( )`: Return true if the list is full otherwise return false.

Stack ADT

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

- `initialize( )`: Initialize the stack to be empty.
- `push( )`: Insert an element at one end of the stack called top.
- `pop( )`: Remove and return the element at the top of the stack if it is not empty.
- `peek( )`: Return the element at the top of the stack without removing it if the stack is not empty.
- `size( )`: Return the number of elements in the stack.
- `isEmpty( )`: Return true if the stack is empty; otherwise, return false.
- `isFull( )`: Return true if no more elements can be pushed; otherwise, return false.

Queue ADT

A queue contains elements of the same type arranged in sequential order, and the following operations can be performed on it:

- `initialize( )`: Initialize the queue to be empty.
- `enqueue( )`: Insert an element at the end of the queue.
- `dequeue( )`: Remove and return the first element of the queue if the queue is not empty.
- `peek( )`: Return the first element of the queue without removing it if the queue is not empty.
- `size( )`: Return the number of elements in the queue.
- `isEmpty( )`: Return true if the queue is empty; otherwise, return false.
- `isFull( )`: Return true if no more elements can be inserted; otherwise, return false.

From these definitions, we can clearly see that they do not specify how these ADTs will be represented or how the operations will be carried out. There can be different ways to implement an ADT. For example, the list ADT can be implemented using arrays, single linked lists, or double linked lists. Similarly, stack

ADT and queue ADT can also be implemented using arrays or linked lists. The representation and implementation details of these ADTs are in the subsequent chapters.

The different implementations of ADT are compared for time and space efficiency, and the implementation best suited for the user's requirements is used. For example, if someone wants to use a list in a program that involves lots of insertions and deletions from the list, then it is better to use the linked list implementation of the list.

1.3 Data Structures

Data structures are data representation methods that are used to organize data. It provides a way of organizing data, so the program can use it effectively. Data structure is a programming construct used to implement an ADT. It is the physical implementation of ADT. All operations specified in ADT are implemented through functions in the physical implementation. A data structure implementing an ADT consists of a collection of variables for storing the data specified in the ADT and the algorithms for implementing the operations specified in ADT.

ADT is the logical view of data and the operations to manipulate it, while a data structure is the actual representation of data and the algorithms to manipulate it.

ADT is a specification language for data structures. ADT is a logical description, while a Data Structure is concrete. In other words, an ADT tells us **what** to do, while data structures tell us **how** to do it. The actual storage or representation of data and implementation of algorithms is done in data structures.

A program that uses a data structure is generally called a client, and the program that implements it is known as the implementation. The specification of ADT is called interface, and it is the only thing visible to the client programs that use data structure. The client programs view data structure as ADT; i.e., they have access only to the interface; the way data is represented, and operations are implemented is not visible to the client. For example, if someone wants to use a stack in the program, he can simply use the push and pop operations without any knowledge of how they are implemented. Some examples of clients of stack ADT are programs of balanced parentheses, infix to postfix, postfix evaluation.

We may change the representation or algorithm, but the client code will not be affected if the ADT interface remains the same. For example, if the implementation of the stack is changed from array to linked list, the client program should work in the same way. This helps the user of a data structure focus on his program rather than going into the details of the data structure.

Data structures can be nested; i.e., a data structure may be made up of other data structures, which may be of primitive types or user-defined types. Some of the advantages of data structures are:

- Efficiency: Proper choice of data structures makes our program efficient. For example, suppose we have some data, and we need to organize it properly and perform a search operation. If we organize our data in an array, we will have to search element by element sequentially. If the item to be searched is present at last, then we will have to visit all the elements before reaching that element. So, the use of an array is not very efficient here. There are better data structures that can make the search process efficient, like ordered arrays, binary search trees, or hash tables. Different data structures give different efficiency.
- Reusability: Data structures are reusable, i.e., once we have implemented a particular data structure, we can use it in any other place or requirement. Data structures can be formed as libraries that can be used by different clients.
- Abstraction: We have seen that a data structure is specified by an ADT, which provides a level of abstraction. The client program uses the data structure through the interface only without getting into the implementation details.

Some common operations that are performed on Data structures are-

- Insertion
- Deletion
- Traversal
- Search

There can be other operations also and the details of these operations depend on data structures. Learning about different data structures is very important because the choice of proper data structure can significantly affect the efficiency of our program.

1.3.1 Linear and Non-Linear Data Structures

A data structure is linear if all the elements are arranged in a linear order. In a linear data structure, each element has only one successor and only one predecessor. The only exceptions are the first and last elements; the first element does not have a predecessor, and the last element does not have a successor. Examples of linear data structures are arrays, strings, linked lists, stacks, and queues.

A nonlinear data structure has no linear order in the arrangement of the elements. Examples of nonlinear data structures are trees and graphs.

1.3.2 Static and Dynamic Data Structures

In a static data structure, the memory is allocated at compilation time only. Therefore, the maximum size is fixed and it cannot be changed at run time. Static data structures allow fast access to elements, but insertion and deletion are expensive. Array is an example of static data structure.

In a dynamic data structure, the memory is allocated at run time. Therefore, these data structures have flexible sizes. Dynamic data structures allow fast insertion and deletion of elements but access to elements is slow. Linked list is an example of dynamic data structure.

1.4 Algorithms

An algorithm is a procedure that contains well-defined steps for solving a particular problem. Algorithms and data structures are closely related. When we implement an algorithm, we have to choose a particular representation of data; i.e., we have to decide which data structure to use, and for using data structures, we have to develop efficient algorithms for various operations on these data structures, like searching, sorting, traversal, etc.

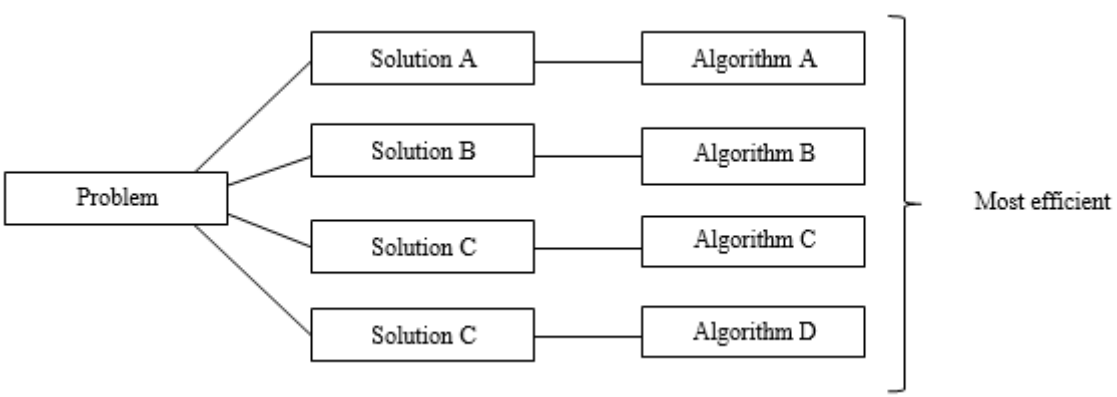

Figure 1.2 Different algorithms to solve a problem

Given a problem, there can be many ways to solve it, and these solutions are in the form of different algorithms. Hence, there can be many algorithms to solve a particular problem.

For example, there are many ways in which data can be sorted, and so we have various sorting algorithms. When we have more than one algorithm to choose from, we would like to use the most efficient algorithm or the best suited for our requirements. To decide which algorithm is best for us, we should be able to compare the algorithms and identify the most efficient one.

The efficiency of an algorithm mainly depends on these two factors:

- Running time of the algorithm
- Memory occupied by it

An efficient algorithm is one that takes less running time and occupies less memory. There can be other measures of efficiency also, like network usage or disk usage, but running time and space are the two factors which are most important. We are generally more concerned about the running time of the algorithm.

Some of the common approaches to algorithm design are explained next.

1.4.1 Greedy Algorithm

A greedy algorithm works by taking a decision that appears best at the moment, without thinking about the future. The decision, once taken, is never reconsidered. This means that a local optimum is chosen at every step in hope of getting a global optimum at the end. It is not necessary that a greedy algorithm always produce an optimal solution. Some examples where the greedy approach produces optimal solutions are Dijkstra's algorithm for single source shortest paths, Prim's and Kruskal's algorithm for minimum spanning tree, and Huffman algorithm.

1.4.2 Divide and Conquer Algorithm

A divide-and-conquer algorithm solves a problem by dividing it into smaller and similar subproblems. The solutions of these smaller problems are then combined to get the solution of the given problem. Examples of divide and conquer algorithms are merge sort, quick sort, and binary search.

1.4.3 Backtracking

In some problems we have several options, and any one of them might lead to the solution. We will take an option and try, and if we do not reach the solution, we will undo our action and select another one. The steps are retraced if we do not reach the solution; it is a trial-and-error process. An example of backtracking is the Eight Queens problem.

1.4.4 Randomized Algorithms

In a randomized algorithm, random numbers are used to make some decisions. The running time of such algorithms depends on the input as well as the random numbers that are generated. The running time may vary for the same input data. An example of randomized algorithm is a version of quick sort where a random number is chosen as the pivot.

1.5 Analysis of Algorithms

There may be many algorithms for solving any problem, and obviously, we would like to use the most efficient one. Analysis of algorithms is required to compare these algorithms and recognize the best one. As discussed previously, algorithms are generally analyzed on their time and space requirements.