Data Structures and Algorithms

Mohammed Sarhat (MangoCodes)

Contents

Chapter 1

Data Structure Introduction

"Bad programmers worry about
the code. Good programmers
worry about data structures
and their relationships."

Linus Torvalds

This notebook is a deep dive into my data structure and algorithms journey. The coverage can be found in the *Content* section. If you are a curious onlooker, it is important to note that you will absolutely need a prerequisite knowledge in basic programming and a cursory understanding of background mathematical knowledge. Without a solid foundational understanding of mathematics, it will be difficult to progress as a problem solver.

For this, we'll utilize a variety of programming languages. For now, I think the use of Python, JavaScript, Go, and C++ will do as they all have efficient libraries that can be used to solve problems.

You will also quickly realize that there are two working copies of this PDF file. Consider the raw PDF like a worksheet. The filled in version is what will be used to teach content on YouTube. At the end of each chapter, I'll include as many references as I possibly can. I highly encourage you check out each one, and skim through their irrespective material.

In all honesty, the lack of LaTex graphs, and drawings comes down to the time it will take to come up with beautifully crafted graphs of these data structures in *Tikz* (LaTex package). Coding this out is arduous and painstakingly redundant. For that reason, I've hand-drawn everything; however, as evident, mathematical notation will be done through LaTex. IF you'd like to add these changes, feel free to Fork the GitHub repository, and PR a change, of which I'll take a look.

1.1 Important Data Structure Definitions

For this section, I really love Clifford A. Shaffer's, a professor at Virginia Tech, unit definitions for data structures as they illustrate the beauty of what constitutes a data structure. If anyone is interested, his book is called *Data Structures and Algorithm Analysis*; it's very well written and packed with information.

A type is a group of values[1]. A data type is that very same type, as defined, with a collection of operations to manipulate that type. For instance, in JavaScript, one primitive data type that is commonly used is the infamous *Number* type. The *Number* data type is constructed with a 64-bit IEEE 754 double precision encoding. The *Number* data type is in fact a data type due to the various operations that it can undergo.

For example, the following are instances of some operations a *Number* data type can endure in JavaScript:

```
// The following are some basic variable declarations housing the Number data type.
let numOne = 1;
let numTwo = 2;
console.log(numOne % numTwo === 1); // true
console.log(numOne + numTwo); // 3
console.log((numOne + 2) * numTwo); // 6
```

A data item can be characterized as a piece of information that is gathered from a type[1]. This piece of information can either be stored in a variable, or be represented as a constant[2]. Simply put, it is a point of data. It could be an entry in a database (built on SQL), a string representation, among many other examples. It doesn't have to be useful, or powerful or even meaningful on its own. For instance, it's convenient to visualize a variable as a box which requires a data type to be stored within it. Suppose we're utilizing a dynamically typed language such as JavaScript, our variable (box) *can* be assigned one or more values during its runtime (JavaScript is an interpreted language). Albeit, this isn't always true with JavaScript, as we can declare some variables as constant, allowing them to remain the exact same as the specified initial value. Since variables house data types, operations can be made onto that variable, however, the scope of what can be done is very surface-level. On the other hand, an array can be characterized as a collection of data items, not always of the same type. Arrays, albeit ambiguous as covered in *Arrays and Linked Lists*, are very powerful. Alone, these data items wouldn't mean much of anything, and as previously mentioned, the scope of their operations wouldn't really get us anywhere. However, in unison with other data items, we can see how powerful data items can be.

Abstraction is a term that's highly used within Computer Science, yet its definitions are often ambiguous. According to Robert Martin, author of *"Agile Software Development, Principles, Patterns, and Practices"*, abstraction is the *"elimination of the irrelevant and the amplification of the essential"*[3]. Aside from the elegant simplistic nature of this quote, its meaning is very impactful in all aspects of computing. For example, a common point of determining what makes one programming language different from another is its level of abstraction. Typically, high level programs are highly abstracted from hardware, and already dynamically handle memory management. Java, Python, JavaScript are all highly abstracted. On the other hand, programming in assembly requires deep knowledge in hardware, and memory management showcasing its absolute lack of abstraction from these components, leading to its designation as a low level programming language.

An abstract data type (ADT) can be characterized as a logical description of data properties, and operations. Each of these operations are defined by its inputs and outputs. The inherent level of abstraction derives off of *how* the data type implementation is not specified[1]. Similar to the distinction between high-level and low-level languages, which differ in how they abstract hardware details, ADT abstraction is achieved through encapsulation. With encapsulation, the internal implementations can be hidden from the outside world, only allowing users to interact with the public interface without needing to understand the underlying code. This manifests itself into libraries with already highly-optimized implementations, as found in Python, C++, JavaScript, among many more.

A data structure is simply a physical implementation that organizes and accesses abstract data types, and refers to an organization for data in memory. Simply put, is is the actual implementation of an ADT in memory. It defined how the data is stored, and *how* operations are performed.

Data items have a logical and physical form[1]. Within an ADT, they typically hold a logical form. For example, a simple list ADT logically represents a sequence of elements, but it does not specify whether it's stored in an array or a linked list. Whereas, with the data structure, they typically hold a physical form. For example, an array stores elements in contiguous memory locations, whereas a linked list stores elements in non-contiguous nodes with pointers.

1.2 Some Basic Logic

1.3 References

[1] C. A. Shaffer, Data Structures and Algorithm Analysis in Java, Third Edition. Courier Corporation, 2012.

[2] https://www.ibm.com/docs/en/epfz/5.3?topic=elements-data-items

[3] Martin, R.C. (2003). Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Upper Saddle River, USA. 978-0-13-597444-5.

Chapter 2

Complexity Analysis

2.1 Measuring Time Complexity

2.1.1 Time Complexity and Space Complexity

2.1.2 Theoretical TC Analysis vs Experimental Complexity Analysis

2.2 Algorithmic Complexity

2.2.1 Time Complexity of An Algorithm

2.2.2 Algorithmic Correctness

2.2.3 Order of Growth

2.3 Asymptotic Notations

2.3.1 Θ-Notation

2.3.2 O-Notation

2.3.3 Λ-Notation

2.4 Complexity of Recursive Algorithms

2.5 P and NP

2.5.1 NP-Completeness

Chapter 3

Arrays, Linked Lists, and Hashing

Chapter 4

Stacks and Queues

Chapter 5

Recursion

Chapter 6

Sorting Algorithms

Chapter 7

Searching Algorithms

Chapter 8

Trees

Chapter 9

Priority Queues and Heaps

Chapter 10

Hashing

Chapter 11

Graphs

Chapter 12

Divide and Conquer

Chapter 13

Greedy Algorithms

Chapter 14

Dynamic Programming

Chapter 15

Branch and Bound