Introduction to data structures

CSC 240 - Data structures with C++ - Lyon College, FA24

Marcus Birkenkrahe

August 3, 2024

Contents

1 Overview

This is an introductory "theoretical" chapter with a little code but from a bunch of different languages because data structures are important everywhere.

Objectives:

- Understand the importance of data structures
- See some examples of data structures
- Understand the relationship between algorithms and data structures
- Discuss an extended sample use case ("Pet Emergency Room")

2 Data structures are everywhere

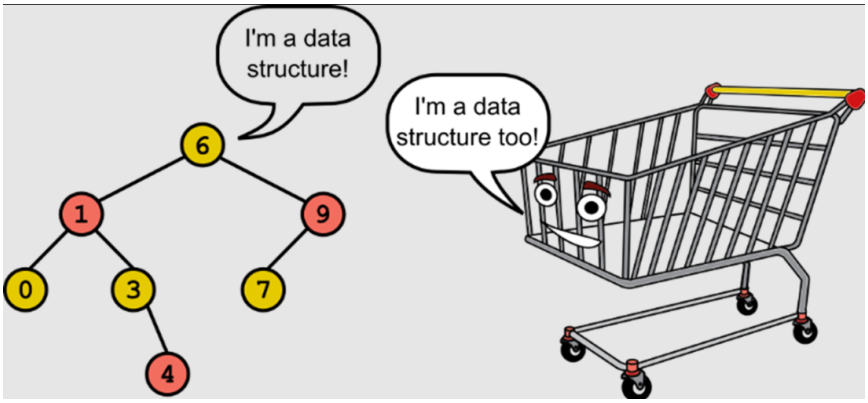

Figure 1: Source: La Rocca, Grokking Data Structures (Manning, 2024)

Why?

- Data are everywhere.
- Data need to be stored (permanently).

- Data need to be processed (moved around, added and deleted)
- Data need to be found (fast).

3 Which data structures do you already know from introductory programming?

That is, if you think about data (numbers, names, characters, etc.) how would you store them?

Examples:

1. Arrays like `a[] = {1,2,3,4}` in C/C++
2. Lists like `foo = [1,2,3,4]` in Python
3. Vectors like `vec <- c(1,2,3,4}` in R
4. Matrices like `M <- diag(4)` in R
5. Dictionaries like `d = {"key":[1,2,3,4]}` in Python
6. Structures like `struct Person { std::string name; int age;}` in C++
7. Enumeration like `enum Color { RED, GREEN, BLUE }` in C++
8. Data frames like `df = pd.DataFrame([1,2],["a","b"])` in Python
9. Class like `public class Person` in Java
10. Table like `CREATE TABLE t (int ID, name TEXT);` in SQL

What about media like videos, images, graphs etc.? Are these data, and how are they stored?

> Also as arrays, lists or dictionaries (key-value pairs) in code and for storage:
>
> - Images: as files in different formats (PNG, JPG, GIF, etc.), or as pixel matrices for programming and analytics purposes.
> - Videos: as sequences of images (frames) with audio data, in different file formats (MP4, MOV, AVI, etc.)
> - Graphs: collections of nodes and edges.

4 Data structures are necessary

Data structures are necessary because of the large of information to be organized and searched through in the digital world.

1. When you open a file, you use a **file system data structure**. Files are stored in structures containing millions of blocks of memory.
2. When you look up a contact on your phone, your phone uses partial information to narrow down the **search** (and it's not powerful) using a data structure (like a dictionary).
3. When you log into a social network, the network servers **check** your login information among possibly billions of users.
4. When you do a web **search**, the search engine uses data structures to find the web pages containing your search terms: among (as of 2024) an estimated 50 billion index by Google alone.
5. When you dial 911, the emergency services network looks up your phone number in a data structure that maps phone numbers to addresses so that someone can go to you right away.

5 Which data structures are used when searching for an address on Google Maps?

Every one of these examples employs not one but half a dozen different data structures. E.g. An address search on Google Maps uses the following data structures:

1. **Geospatial Indexing**
    - **Quadtree**: A tree data structure in which each internal node has exactly four children. Quadtrees are used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. This is useful for indexing geographical locations efficiently.
    - **R-tree**: A tree data structure used for spatial access methods, i.e., for indexing multi-dimensional information. It is commonly used for indexing spatial information such as geographical coordinates, which is crucial for efficiently querying map data.

2. **Hash Maps and Tries**

   - **Hash Map**: This data structure is used for quick lookups of specific addresses or place names. It allows for constant time complexity for search operations, which is important for performance in real-time applications.
   - **Trie (Prefix Tree)**: A specialized tree used to store associative data structures. A trie is often used for autocomplete features in search queries, allowing for quick lookup of address prefixes and suggestions.

3. **Graph Structures**

   - **Graph**: Google Maps uses graph data structures to represent the network of roads, intersections, and various routes. Each node in the graph represents a geographic location (like an intersection), and each edge represents a road segment connecting two nodes. This is essential for route planning and navigation.

4. **KD-Trees**

   - **KD-Tree (k-dimensional tree)**: A space-partitioning data structure for organizing points in a k-dimensional space. KD-trees are useful for nearest neighbor searches, which are common in mapping applications when finding the closest location or point of interest.

5. **Spatial Databases**

   - **Spatial Database**: Specialized databases designed to store and query data related to objects in space, including points, lines, and polygons. Google Maps relies on spatial databases to manage large-scale geographic data efficiently.

6 Algorithms vs. Data Structures

- An **algorithm** is a set of well-defined instructions, a step-by-step procedure to solve a problem (e.g. compute 1 + 1) or perform a task (e.g. say 'hello world').
- Algorithms can be described using pseudocode (syntax-free code)[1]. Here are three examples for a "fizzbuzz" function:

[1] Though there is no standard, efficient pseudocode is not completely arbitrary, cp.

Pascal style:

```
procedure fizzbuzz;
  for i := 1 to 100 do
    print_number := true;
    if i is divisible by 3 then
begin
      print "Fizz";
      print_number := false;
    end;
    if i is divisible by 5 then
begin
      print "Buzz";
      print_number := false;
    end;
    if print_number, print i;
    print a newline;
  end
```

C style:

```
fizzbuzz() {
  for (i = 1; i <= 100; i++)
{
    print_number = true;
    if (i is divisible by 3)
{
      print "Fizz";
      print_number = false;
    }
    if (i is divisible by 5)
{
      print "Buzz";
      print_number = false;
    }
    if (print_number) print
i;
    print a newline;
  }
}
```

Python style:

```
def fizzbuzz():
  for i in range(1,101):
    print_number = true
    if i is divisible by 3:
      print "Fizz"
      print_number = false
    if i is divisible by 5:
      print "Buzz"
      print_number = false
    if print_number: print
i
    print a newline
```

Figure 2: Source: Wikipedia

- A **data structure** is a way of organizing and storing data in a computer via a programming language. It defines the relationship between the elements of the structure, the operations that can be performed on the data, and the rules or constraints for accessing and modifying the data.
- Short formula: **Algorithms transform data like verbs acting on nouns.**

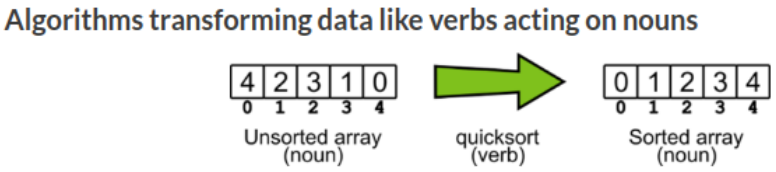

Figure 3: Source: La Rocca, Grokking Data Structures (Manning, 2024)

- Some data structures are designed to allow efficient execution of certain algorithms, e.g. **hash tables** for key-based search.
- You automatically learn about algorithms when studying data structures and vice versa (CSC 240 and CSC 265).

Helfrich (Appendix A), "Pseudocode Standard" (J Dalbey), and this pseudocode literature review (2021).

7 Data structures matter

- Different data structures have wildly differing properties and powers - compare for example **arrays** and **vectors** in C++.
- **C++ array vs. vector:**
  1. **Arrays** are collection of elements of the same type, stored in contiguous memory locations. They are used to store multiple values in one variable. Values can be accessed using an index. The size of the array is determined at compile-time.
  2. **Vectors** are dynamic arrays provided by the C++ Standard Template Library (STL). They can grow and shrink in size dynamically as elements are added or removed, that is at run-time. They offer useful member functions.
- **Array** code example: declare and initialize an array using a `for` loop.

```
// declare array of 5 integers and initialize it to 0
int array[5]{};

// assign values to the array using a traditional for loop
for (int index=0; index < 5; index++) {
  array[index] = index + 100;
 }

// print initialized array using a range-based for loop
for (int value : array) cout << value << " ";


100 101 102 103 104
```

- **Vector** code example:

```
#include <vector>

int main() {
  // create integer vector
  vector<int> myVector;

  // Adding elements to the vector
```

```
  for (int i = 0; i < 5; ++i) {
    myVector.push_back(i + 100);
  }

  // Accessing and printing elements
  for (int i = 0; i < myVector.size(); ++i) {
    cout << myVector[i] << " ";
  }

  return 0;
}
```

```
100 101 102 103 104
```

- If C++ is not (yet) your mojo, think of the difference between **lists**, **tuples**, and **dictionaries** in Python:
  1. **list** elements can be of any data type, but **tuple** elements must have the same data type.
  2. **list** elements are ordered (indexed), but **dictionaries** are unordered.
  3. **lists** are mutable (can be modified), but **tuples** are immutable.
  4. **lists** are passed by reference, but **tuples** are passed by value.
- Code examples:

```
# list elements can have any data type
foo = [1, "Hello", 'A', True]
# lists are ordered
print(foo[0], foo[1], foo[2], foo[3])
# lists are mutable
foo[2] = 'B'
print(foo)
# lists are passed by reference (not by value)
bar = foo; bar[1]=2; print(foo) # change value in bar - copied to foo
```

- R is another interesting example: the central data structure is the vector; defining and manipulating vectors is made very simple, and as an added plus, R indices start from 1 instead of 0:

```
utput :exports both
    foo <- c(1,3,-100,42)  # defining a vector
    foo  # printing vector
    bar <- c(foo, foo)  # extending the vector
    foo / 2 # vectorization: element-wise operations and implicit conversion
    matrix(1:16,4) # a 4 x 4 matrix. Full command: matrix(data=1:16,nrow=4)
```

- Unlike C/C++ and Python, a lot of visualization options are already built into R.
- Graphics depend on data structures, too! Compare e.g. the SVG (Scalable Vector Graphics) image format whose images are represented not as a grid of pixels but using geometric shapes (points, lines, curves, polygons). Advantage: images are scalable to any size without loss of quality.

8 Why you should learn data structures

- Learn an essential tool you cannot really do without (like editing, managing files or plotting data).
- Progress in Machine learning depends on new data structures like graph neural nets.
- The database landscape is evolving as data volumes grow (e.g. flexible indexing)
- Avoid Maslow's hammer ("If your toolbelt only has a hammer, you will be tempted to treat everything as a nail.") - What if you must tighten a screw? The answer is abstraction (but not too much).
- Abstraction is great (C), too much abstraction makes things harder than they need to be at the start (C++).

9 Example use cases

1. Searching through a large collection of baseball cards (binary search on sorted arrays)
2. Keeping track of logged-in users and their IP addresses (at scale, and securely - hash tables).

3. Modeling relationships between social media network users (persistent and scalable - graphs).

4. In game development, AI calculations are optimized, such as determining the best move (storing previous calculations - memoization).

5. Text processing - auto-completion and spell checking (store dynamic sets of strings - trie memoization).

6. Task scheduling with priority queues (sorting, traversing - heaps).

10 How do choose the right data structure

By building up algorithmic muscle: solve **many problems** in **different ways** and with **different data structures**:

Example: "Hello" program that greets the user by name

1. Input from the keyboard or from a file

2. Output to the screen or to a file

3. Output with `cout` (`iostream`) `or =printf` (`stdio.h`)

4. User name stored as `string` or as `char` array

5. User name retrieved with `cin` or with `cin.get` or `scanf`

6. Concatenate greeting message with `+` or stream with `<<`

7. With a user-defined function or as standalone command

8. With a user-defined `class` or `struct`

9. Print from `main` program or from a subroutine

10. Fix length of string or use variable-length arrays.

These solutions differ by performance, flexibility, readability, nstandardization, style, and portability.

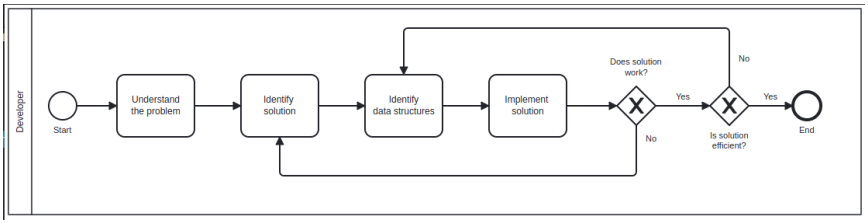

Figure 4: BPMN model for using data structures

11 A mental model for applying data structures

- The implementation comes after everything else.
- The days of finding solutions through "tinkering" are over.
- Computer programs can be formally proven to be correct[2].

12 Extended use case: "Pet emergency room"

- **Problem:** how to run an emergency waiting room for pets, in particular the registration and admission of patients.
- **Constraints:**
  - multiple species of animals
  - infinite capacity of the waiting room
  - no other constraints
- **Input:** time-ordered group of animals.
- **Output:** ordered waiting list.
- **Process:** register animals, then admit them in order.

There are (at least) four different solutions to choose from:

[2]Step-by-step proofs include: (1) error-free compilation, logical correctness, edge case handling; (2) code analysis with includes, namespace, main function, variable declarations, input and output handling, (3) Code testing with various inputs.

Bag

- Data structure: **Bag**. All patient forms are put in the container in random order.
- **Check:** Does solution work, and does it work well?
  1. A bag is easy to implement.
  2. You cannot keep control over the order.

Stack

- Data structure: **Stack**. All patient forms are stored in order, in a pile with the oldest at the bottom and the newest at the top. Forms are taken from the top.
- Does solution work, and does it work well?
  1. A stack is good to process the most recent entries first.
  2. It's bad to handle a waiting line (prioritizes recent entries).

Queue

- Data structure: : **Queue.** All patient forms are stored in order, in a pile with the oldest at the bottom and the newest at the top. Forms are taken from the bottom.
- Does solution work, and does it work well?
  1. The queue implements a first come, first serve policy.
  2. Problem: urgent entries are forced to wait.

Priority queue

- Data structure: **Priority queue**. A queue that t akes more than arrival time into account. It also contains information on **priority**. Patient forms are ordered first by priority, and then by arrival.
- Does solution work, and does it work well?
  1. Urgent entries are now processed first.
  2. This container is slower and more complex to implement.

Model each process in BPMN

- BPMN (Business Process Model and Notation) is a diagrammatic language

to create process models that represent the flow of an action from start to end for every participant of a process.

- In this case, the participant is the Pet ER Receptionist who interacts with the patients by getting forms from the patient, sorts the forms, and pulls them from the stack before calling upon the next patient.
- "Bag" solution

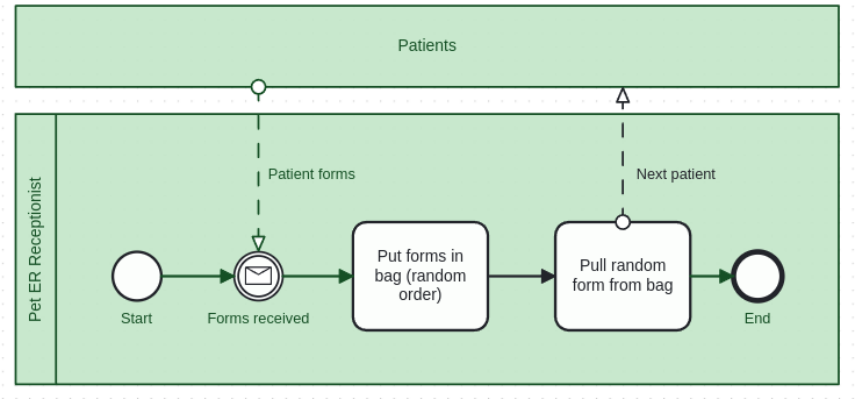

Figure 5: BPMN models for "Pet Emergency Room" use case: Bag

- "Stack" solution
- "Queue" solution
- "Priority queue" solution
- You can see that the procedural shape of these processes is much the same.

13 Abstract Data Types

- Mastering the chosen data structure involves mastering a set of operations, actions to perform with them.

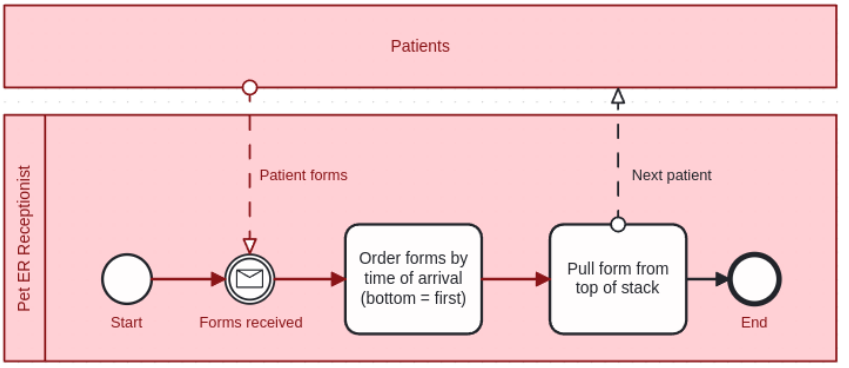

Figure 6: BPMN models for "Pet Emergency Room" use case: Stack

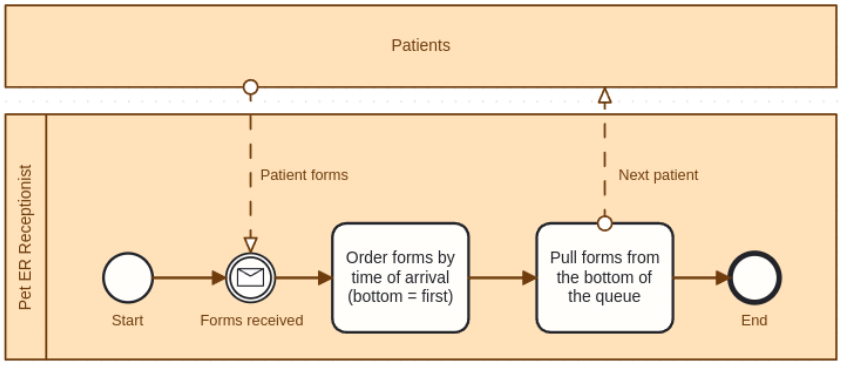

Figure 7: BPMN models for "Pet Emergency Room" use case: Queue

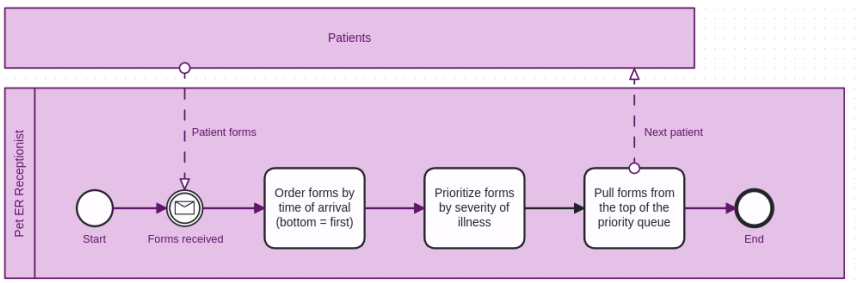

Figure 8: BPMN models for "Pet Emergency Room" use case: Priority Queue

- A data structure together with basic operations like *insert*, *remove*, *traverse*, *count* etc. is called an Abstract Data Type (ADT).
- The set of operations of an ADT constitute its *public interface*. It defines what we're allowed to do with it.
- Example: inserting an element into a linked list requires creating and deleting pointers between link elements.

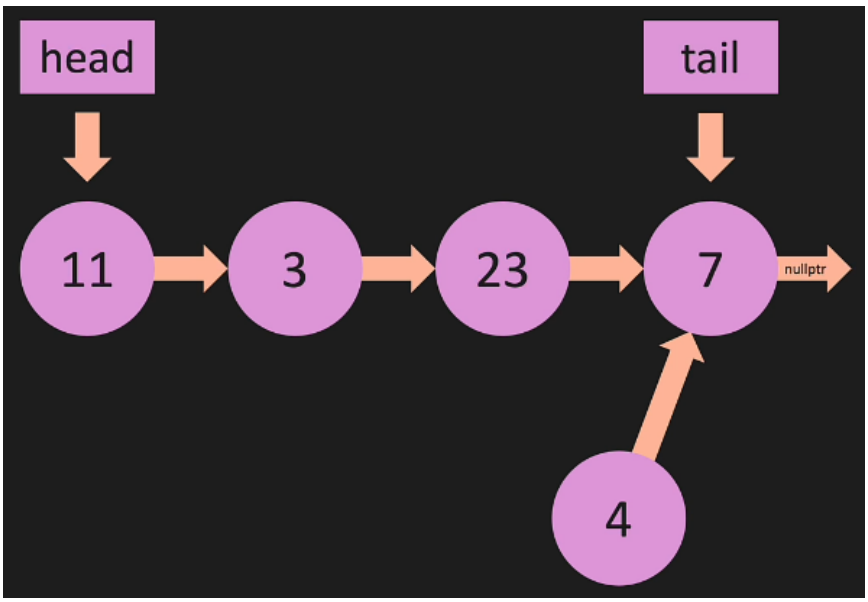

Figure 9: Image source: Scott Barrett @Udemy

14 The need for efficiency (Morin)

- **Example:** If an application looks up a dataset with 1 mio (10^{6}) items, and if each item needs to be looked up at least once, we have 10^{12} = 10^6 x 10^6 searches or number of operations.
- If a processor can perform 1 bio (10^{9}) operations per second, it will take it 10^{12}/10^{9} = 1000 seconds or 16 min 40 sec to search through the dataset once.
- Google indexes over 50 bio (50 x 10^9) web pages, which means that a query over this data would take no less than 50 seconds. Google

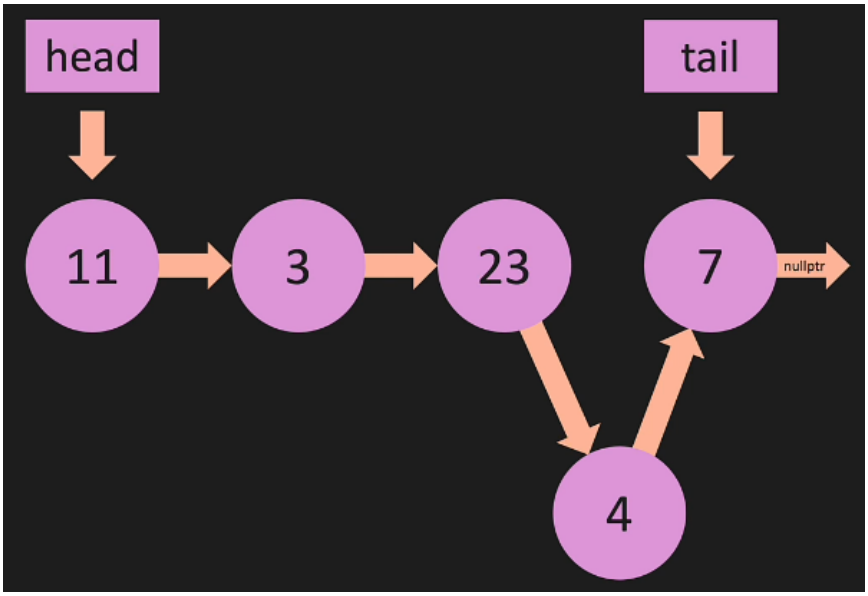

Figure 10: Image source: Scott Barrett @Udemy

received approximately 40,000 = 4 x 10^{5} queries per second. Google operates no less than 40 data centers with no less than 50,000 servers per data center (40 x 5 x 10^5 = 2 x 10^{7}) - to match this demand.

- The examples show that obvious implementations of data structures do not scale well if the number of items n in the data structure (e.g. a table) and the number of operations m performed on it (e.g. a search) are large because the time (measured in machine instructions) is roughly n x m.
- For example there are data structures where a search only needs to look at two items on average independent on the number of items stored - which means that a 1 bio instructions per second computer can complete a search of 50 bio pages in 2 x 10-9 seconds, or 2 nano seconds.
- Then there are data structures where the number of items inspected during an operation grows very slowly as a function of the number of items.

15 Summary

- A data structure is a way of organizing and storing data in a computer or a programming language, defining the relationship between data, operations that can be performed on the data, and rules or constraints for accessing and modifying the data.
- Data structures are fundamental to organizing and storing data efficiently.
- An algorithm is a set of well-defined instructions, a step-by-step procedure designed to solve a specific problem or perform a particular task.
- Algorithms and data structures complement each other like nouns and verbs complement each other in a sentence.
- Choosing the wrong data structure can have dire consequences, such as crashing your website or causing security hazards.
- There is a step-by-step process that can help you decide which data structures to use in a project.
- The process is iterative and requires you to check the quality of your solution until you meet all of your requirements.
- Abstract Data Types are Data Structures with their operations, e.g. a list with insertion, removing, counting, traversing.

16 References

- Pseudocode Literature Review - IEEE
- Geospatial Indexing - Wikipedia
- How Google Maps Works - TechRepublic
- Introduction to Spatial Databases - Esri
- Trie Data Structure - GeeksforGeeks
- Morin, Open Data Structures in C++ (2024)
- Rocca, Grokking Data Structures (2024)

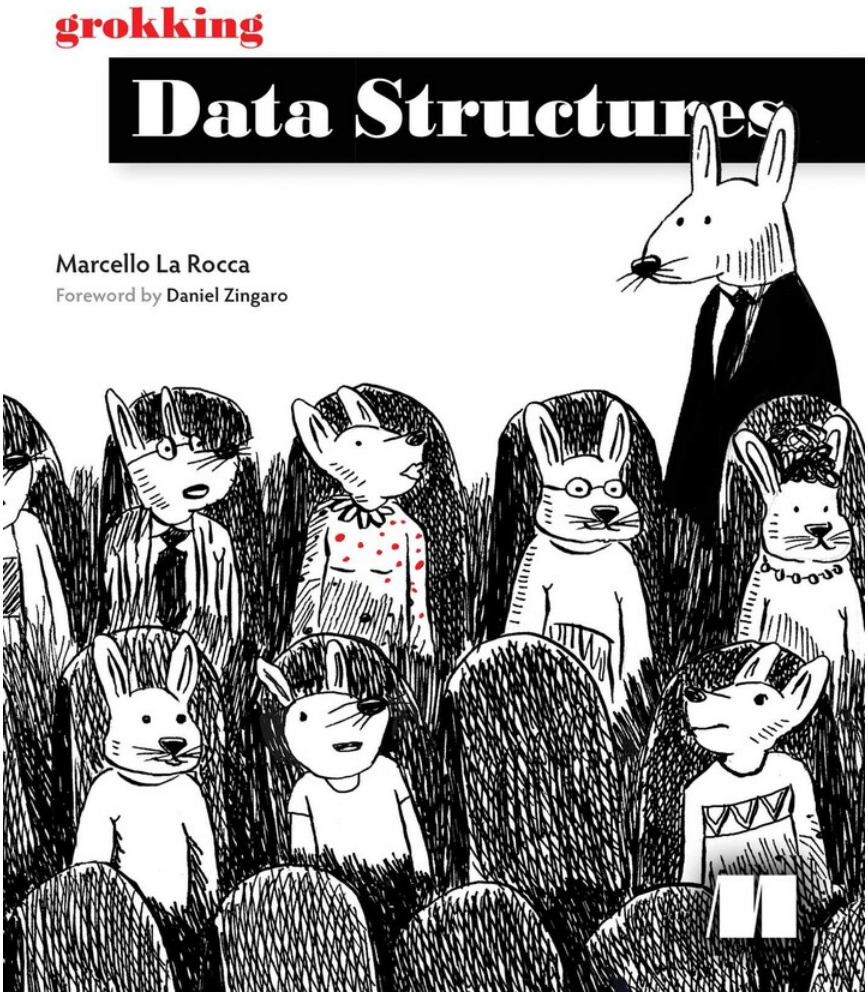

Figure 11: Cover De Rocca, Grokking Data Structures (Manning, 2024)