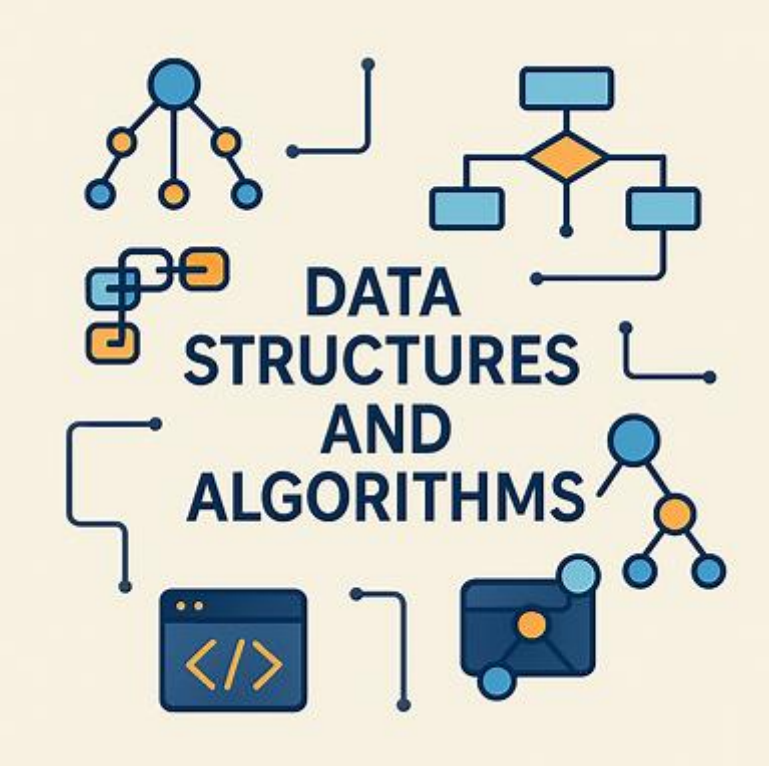
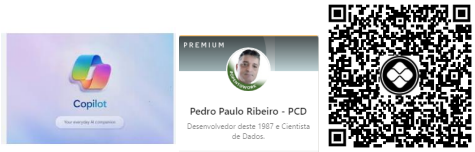
Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms





Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Quantum Computing and Data Structure and Algorithms

Quantum Computing + Data Structures and Algorithms is a powerful and extremely current combination. I'm going to give you a complete, clear, and in-depth **mini--chapter** showing how these two worlds connect, where they clash, and where they complement each other.

If you'd like, I can turn this into an official chapter of your book (Chapter 18), expanding on it with mathematical examples, quantum circuits, pseudocode, or even exercises.

📓 Quantum Computing and Data Structures and Algorithms

🧠 1. What changes when we enter the quantum world?

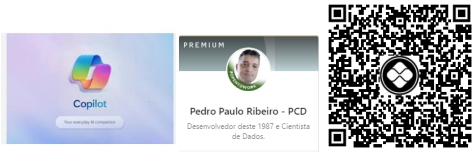
Classical computing works with **bits** (0 or 1). Quantum computing works with **qubits** , which can be:

- 0
- 1
- or **superposition** (0 and 1 at the same time)

Furthermore, qubits can be **entangled** , creating correlations that are impossible in the classical world.

This completely changes the way we think about:

- Data structures
- Algorithms
- Complexity
- Storage
- Search
- Cryptography



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Data Structures in Quantum Computing

Here's the surprising part: **there are no traditional quantum data structures** like lists, trees, graphs, stacks, queues, etc.

Why?

Because **it's not possible to read the complete state of a qubit without destroying it** (wavefunction collapse).

So, instead of data structures, we use **quantum states** and **unitary operations** .

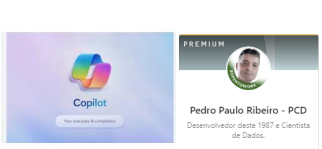
Examples of quantum “structures”:

- State vectors
- Unitary matrices
- Pauli Operators
- Quantum circuits
- Quantum recorders

These are the "structures" that quantum algorithms manipulate.

3. Famous Quantum Algorithms

Quantum computing doesn't speed up *all* algorithms — but it speeds some up dramatically.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Here are the main ones:

🔍 3.1 Grover's Algorithm — Search in O(√n)

Search in an unordered list:

- Classic: O(n)
- Quantum: **O(√n)**

This is revolutionary for:

- Search in databases
- Symmetric cryptography
- Optimization problems

🔓 3.2 Shor's Algorithm — Polynomial-Time Factorization

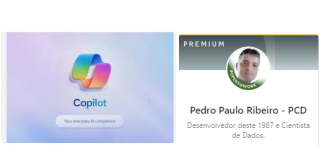
Factoring large numbers:

- Classic: sub- -exponential
- Quantum: **polynomial**

Direct impact:

- RSA Break
- Diffie -Hellman Breaks
- ECC Breakdown

This is the algorithm that has prompted governments and companies to rush to create post- **quantum cryptography -**.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

🔁 3.3 QFT — Quantum Fourier Transform

Basis for several algorithms:

- Short
- Phase estimate
- Quantum simulation

It is exponentially faster than the classic FFT.

🧠 3.4 Variational Algorithms (VQE, QAOA)

Used in:

- Quantum chemistry
- Optimization
- Machine Learning

They are hybrids: part classical + part quantum.

🟦 4. Quantum Complexity

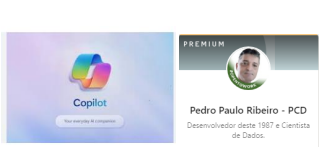
Quantum computing introduces new classes:

- **BQP** — problems solvable by quantum computers in polynomial time
- **QMA** — quantum version of NP
- **QCMA** — intermediate between NP and QMA

Comparison:

Model	Class
Classic	P, NP, NP -Complete
Quantum	BQP, QMA

Important: **BQP is not the same as NP, nor does it replace NP.** They are different models.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

5. Classical vs. Quantum Data Structures

Classical Structure	Does it exist in the quantum model?	Observation
Array	❌	Direct indexing is not possible.
Linked list	❌	Reading destroys the state.
Tree	❌	There is no such thing as deterministic navigation.
Hash table	❌	Grover replaces hashing
Graph	✔ (implied)	Represented as a quantum state
Vector	✔	Amplitude vector

6. How do classical and quantum algorithms complement each other?

Quantum computing **does not replace** classical computing. It **speeds up specific parts of it** .

Example:

- Pre- -processing → classic
- Search → Grover
- Optimization → QAOA
- Post- -processing → classic

The future is **hybrid** .

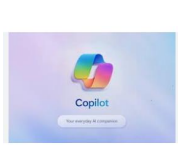
7. Examples of real-world applications

Chemistry and materials

Molecular simulation → exponentially faster.

Finances

Portfolio optimization → QAOA.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Security

Post-quantum cryptography -.

Biology

Protein simulation.

Logistics

NP problems -⟶ quantum heuristics.

Classical Computing and Data Structure and Algorithms

Data Structures and Algorithms — Fundamentals, Practice, and Mastery

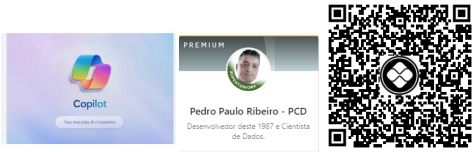
✦ Preface

This book was written for students, professionals, and enthusiasts who want to deeply understand how data structures and algorithms work, why they are essential, and how to apply -them to solve real-world problems. It combines solid theory, practical examples, complexity analyses, and exercises.

CHAPTER 1 — Introduction to Data Structures and Algorithms

What are Data Structures?

- Organized ways of storing information.
- They allow for efficient access, modification, and processing.
- Examples: arrays, lists, trees, graphs, hash tables.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

What are Algorithms?

- A sequence of steps to solve a problem.
- A good algorithm is:
  - Correct
  - Efficient
  - Simple
  - Scalable

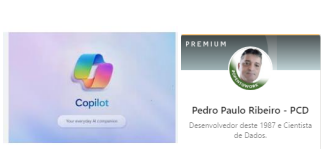
Why does this matter?

- All software relies on data structures.
- Efficient algorithms save time, money, and energy.
- Understanding these concepts is essential for technical interviews.

🔑 CHAPTER 2 — Algorithm Analysis and Big -O Notation

Why analyze algorithms?

- To predict performance.
- To compare solutions.
- To avoid bottlenecks.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

O notation-

This represents the worst-case scenario for execution time.

Complexity	Name	Example
O(1)	Constant	Array access
O(log n)	Logarithmic	Binary search
O(n)	Linear	Browse list
O(n log n)	Almost linear	Merge sort
$O(n^2)$	Quadratic	Bubble sort
$O(2^n)$	Exponential	Subsets
O(n!)	Factorial	Permutations

🔑 CHAPTER 3 — Basic Data Structures

Arrays

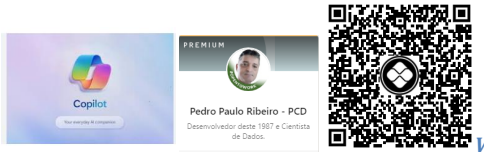
- Fixed indices
- Quick access
- Slow insertion

Strings

- Strings
- Immutable in many languages
- Classic algorithms: KMP, Rabin -Karp

Records and Objects

- They group heterogeneous data.
- Object-oriented programming foundation



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

🔑 CHAPTER 4 — Lists, Stacks, and Queues

Linked Lists

- We are connected by pointers.
- Quick insertion
- Slow access

Stacks

- LIFO
- Uses: recursion, parsing, backtracking

Queues

- FIFO
- Uses: waiting systems, BFS

🔑 CHAPTER 5 — Trees

Binary Trees

- Each node has up to 2 offspring.
- Routes:
    - In -order
    - Pre --order
    - Post -order

General Trees

- Any number of children
- Used in file systems


Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 6 — Balanced Trees and Heaps

AVL

- Strict balancing
- Height O(log n)

Red -Black Trees

- More flexible balancing
- Used in standard libraries

Heaps

- Used in priority queues
- Heap Sort Base

CHAPTER 7 — Hash Tables

Hashing

- Hash function → index
- Collisions: chaining, open addressing
- Average time: O(1)

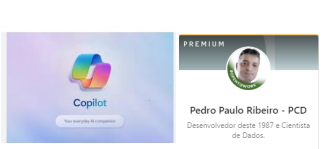
CHAPTER 8 — Graphs

Representations

- Adjacency list
- Adjacency matrix

Types

- Targeted
- Not targeted
- Considered



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

🔑 CHAPTER 9 — Sorting Algorithms

Comparisons

- Merge Sort
- Quick Sort
- Heap Sort

Not comparative

- Counting Sort
- Radix Sort

🔑 CHAPTER 10 — Search Algorithms

- Linear Search
- Binary Search
- Tree Search
- Graph Search (BFS/DFS)

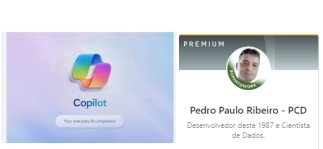
🔑 CHAPTER 11 — Dynamic Programming

Central idea

- Solve problems by breaking them down into sub-problems.
- Save intermediate results

Classic examples

- Fibonacci
- Knapsack
- Longest Common Subsequence



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 12 — Greedy Algorithms

Features

- Choose the optimal location.
- It doesn't always work.
- When it works, it's powerful.

Examples

- Dijkstra
- Huffman
- Interval Scheduling

CHAPTER 13 — Graph Algorithms

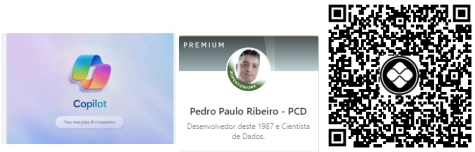
- BFS
- DFS
- Dijkstra
- Bellman -Ford
- Floyd -Warshall
- Kruskal
- First

CHAPTER 14 — Advanced Structures

- Segment Trees
- Fenwick Trees
- Tries
- Suffix Arrays
- Bloom Filters

CHAPTER 15 — Algorithm Design

- Divide and Conquer



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

- Backtracking
- Branch and Bound
- Randomization
- Heuristics

🔑 CHAPTER 16 — Classic Problems

- Towers of Hanoi
- Minimum path
- Complete NP problems-
- Scheduling
- Matching

🔑 CHAPTER 17 — Conclusion

You have now mastered the essential fundamentals of data structures and algorithms. The next step is to practice intensively, solve problems, and build projects.

✦ Want to expand the book?

Examples in Python, Java, C++, or another language.

Real-world code examples in various languages. To keep everything organized, I've prepared equivalent examples in **Python** , **Java** , and **C++** for the main structures and algorithms. If you want, I can add other languages like JavaScript, Go, Rust, or C#.

Below are clear, didactic, and ready-to-use examples.

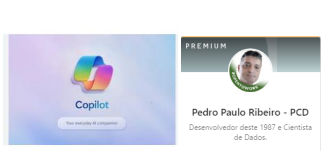
📦 1. Basic Data Structures

📌 Linked List

Python

python

```
class Node:
def __init__(self, value):
```



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

```
self.value = value
self.proximo = None

class LinkedList:
def __init__(self):
self.head = None

def insert(self, value):
new = Node(value)
new.next = self.head
self.head = new

def show(self):
current = self.head
while current:
print(atual.valor, end=" -> ")
current = current.next
print("None")

list = LinkedList()
list.insert(10)
list.insert(20)
list.insert(30)
list.show()
```

Java

Java

```
class Node {
int value;
Next node;

Node(int value) {
this.value = value;
this.next = null;
}
}

class LinkedList {
Node head;

void insert(int value) {
New Node = new Node(value);
new.next = head;
head = new;
}

void show() {
Current node = head;
while (current != null) {
```

Classical And Quantum Computing

Data Structure e Algorithms

```
System.out.print(atual.valor + " -> ");
current = current.next;
}
        System.out.println("None");
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList lista = new LinkedList();
        lista.inserir(10);
        lista.inserir(20);
        lista.inserir(30);
        lista.mostrar();
    }
}
```

C++

cpp

```
#include <iostream>
using namespace std;

struct Node {
    int valor;
    Node* proximo;
    Node(int v) : valor(v), proximo(nullptr) {}
};

class LinkedList {
public:
Node* head = nullptr;

void insert(int value) {
Node* novo = new Node(valor);
new->next = head;
head = new;
}

void show() {
Current node* = head;
while (current) {
cout << current->value << " -> ";
current = current->next;
}
cout << "None\n";
}
};

int main() {
```

Classical And Quantum Computing

Data Structure e Algorithms

```
LinkedList list;
list.insert(10);
list.insert(20);
list.insert(30);
list.show();
}
```

2. Binary Search

Python

python

```
def binary_search(arr, target):
start, end = 0, len(arr) - 1
while start <= end:
middle = (start + end) // 2
if arr[meio] == alvo:
return middle
elif arr[meio] < alvo:
start = middle + 1
else:
end = middle - 1
return -1

print(binary_search([1,2,3,4,5,6], 4))
```

Java

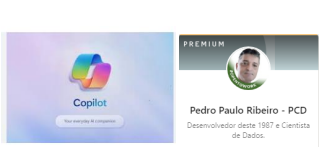
Java

```
public class Main {
public static int BinarySearch(int[] arr, int target) {
int start = 0, end = arr.length - 1;

while (start <= end) {
int middle = (start + end) / 2;

if (arr[meio] == alvo) return meio;
else if (arr[meio] < alvo) inicio = meio + 1;
else end = middle - 1;
}
return -1;
}

public static void main(String[] args) {
int[] arr = {1,2,3,4,5,6};
System.out.println(searchBinary(arr, 4));
}
}
```




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

C++

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int binarysearch(vector<int>& arr, int target) {
int start = 0, end = arr.size() - 1;

while (start <= end) {
int middle = (start + end) / 2;

if (arr[meio] == alvo) return meio;
else if (arr[meio] < alvo) inicio = meio + 1;
else end = middle - 1;
}
return -1;
}

int main() {
vector<int> arr = {1,2,3,4,5,6};
cout << binarySearch(arr, 4);
}
```

🔄 3. Algoritmos de Ordenação

Bubble Sort

Python

python

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

print(bubble_sort([5,3,8,4,2]))
```

Java

java

```
public class Main {
    public static void bubbleSort(int[] arr) {
```

Classical And Quantum Computing

Data Structure e Algorithms

```
        int n = arr.length;

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j+1]) {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}
```

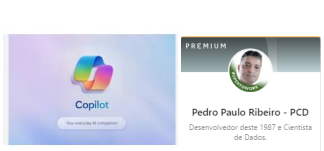
C++

cpp

```
#include <iostream>
#include <vector>
using namespace std;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
}

int main() {
    vector<int> arr = {5,3,8,4,2};
    bubbleSort(arr);
    for (int x : arr) cout << x << " ";
}
```



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

🧠 4. Programação Dinâmica — Fibonacci

Python

python

```
def fib(n):
    dp = [0, 1]
    for i in range(2, n+1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]

print(fib(10))
```

Java

java

```
public class Main {
    public static int fib(int n) {
        int[] dp = new int[n+1];
        dp[0] = 0;
        dp[1] = 1;

        for (int i = 2; i <= n; i++)
            dp[i] = dp[i-1] + dp[i-2];

        return dp[n];
    }
}
```

C++

cpp

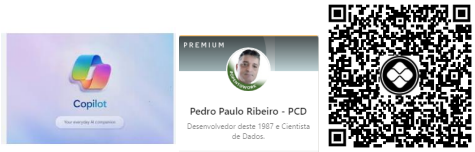
```
#include <iostream>
#include <vector>
using namespace std;

int fib(int n) {
    vector<int> dp(n+1);
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];

return dp[n];
}

int main() {
```



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

```
cout << fib(10);
```

CHAPTER 1 — Introduction to Data Structures and Algorithms

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding what data structures and algorithms are.
- Understanding why they are fundamental in computing.
- Recognizing how they influence software performance, scalability, and quality.
- Identify real problems that depend directly on these concepts.

1.1 What are Data Structures?

Data structures are **organized ways of storing and manipulating information** . They define:

- How the data is stored.
- How can they be accessed?
- How can they be modified?
- How efficient is each operation?

Everyday examples

Situation	Implicit data structure
Contact list on mobile phone	Table + index
Browsing history	Linked list
GPS calculating routes	Graphs
File system	Trees
Database	B-trees, Hash tables

Why does this matter?

Imagine a messaging app with millions of users. If every search for a conversation were done carelessly, the app would become slow, crash, and consume a lot of battery.

Efficient data structures make systems:

- Faster

- More economical
- More scalable
- More reliable

1.2 What are Algorithms?

An algorithm is a **set of well-defined steps** for solving a problem.

Simple examples:

- Cake recipe → culinary algorithm
- Assembly instructions for a piece of furniture → mechanical algorithm
- Step-by-step guide to finding a name in the phone book → search algorithm

In computing, algorithms are implemented in code, but the concept is independent of the language.

Characteristics of a good algorithm

- **Correct** — it solves the problem.
- **Efficient** — uses few resources.
- **Readable** — easy to understand and maintain.
- **Scalable** — works well even with large volumes of data.
- **Deterministic** — always produces the same result for the same input.

1.3 The Relationship between Data Structures and Algorithms

They are inseparable.

- An algorithm depends on a data structure to function.
- A data structure is only useful if there are algorithms to manipulate -it.

Practical example

To find an element:

- In an **unsorted array** , the search is **O(n)** .
- In a **sorted array** , we can use **binary search O(log n)** .
- In a **hash table** , the average search is **O(1)** .

Classical And Quantum Computing

Data Structure e Algorithms

The problem is the same. Efficiency changes drastically depending on the structure chosen.

1.4 Why study Data Structures and Algorithms?

1.4.1 Performance

Modern applications handle gigantic volumes of data. A bad algorithm can render a system unusable.

Real-life example:

- Sorting 1 million numbers with Bubble Sort ($O(n^2)$) → hours
- Sort using Merge Sort ($O(n \log n)$) → seconds

1.4.2 Scalability

Systems grow. Users increase. Data explodes.

Efficient structures and algorithms ensure that the system continues to function well.

1.4.3 Technical Interviews

Companies like Google, Amazon, Microsoft, Meta, and others strongly value:

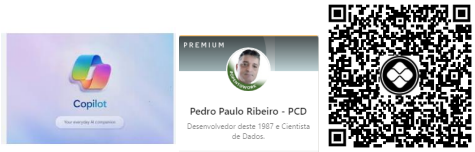
- Trees
- Graphs
- Hash tables
- Search and sorting algorithms
- Dynamic programming

Knowing these topics is practically mandatory.

1.4.4 Computational Thinking

Studying algorithms improves your ability to:

- Solve problems
- Thinking logically



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

- Creating elegant solutions
- Optimize processes

1.5 How Data Structures and Algorithms are Classified

1.5.1 Data Structures

Simple

- Variables
- Arrays
- Strings

Linear

- Linked lists
- Batteries
- Queues

Hierarchical

- Trees
- Heaps

Associations

- Hash tables
- Maps

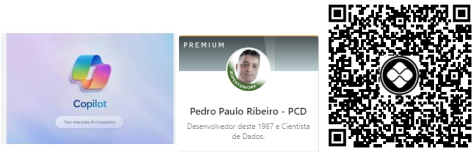
Complex

- Graphs
- Tries
- Segment Trees

1.5.2 Algorithms

For the purpose

- Ordering



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

- Search
- Compression
- Cryptography
- Minimum path

By technique

- Divide and conquer
- Dynamic programming
- Greedy algorithms
- Backtracking
- Randomization

1.6 How to measure the efficiency of an algorithm

Efficiency is measured in:

- **Time** (how many operations it performs)
- **Memory** (how much space does it use)

The standard tool is **Big -O Notation** , which we will see in Chapter 2.

1.7 Simple Examples of Algorithms

1.7.1 Finding the largest number in a list

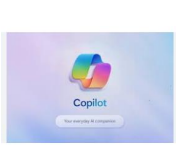
Pseudocode

Code

```
largest = list[0]
for each element in the list:
if element > larger:
largest = element
return higher
```

Complexity

- Time: O(n)
- Space: O(1)




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

1.7.2 Checking if a number is in a list

Pseudocode

Code

```
for each element in the list:
if element == target:
return true
return false
```

Complexity

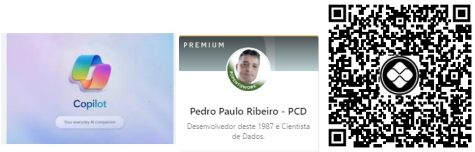
- Time: O(n)
- Space: O(1)

1.8 Chapter Conclusion

In this chapter, you learned:

- What are data structures?
- What are algorithms?
- Why are they essential?
- How do they relate to each other?
- How are they classified?
- How to measure efficiency
- Simple examples of algorithms

You now have the foundation to move on to **Chapter 2: Big -O Notation and Algorithm Analysis** , where we will delve deeper into performance analysis.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

📝 Exercises from Chapter 1

1. Conceptual

1. Explain in your own words what a data structure is.
2. Give three examples of algorithms used in everyday life.
3. Why is efficiency important in large systems?
4. What is the relationship between data structures and algorithms?

2. Practical

5. Write an algorithm (in pseudocode) that counts how many even numbers there are in a list.
6. Write an algorithm that finds the smallest number in a list.
7. Classify the following problems as search, sorting, or optimization:
    - Find the shortest route between cities.
    - Organize names in alphabetical order.
    - Check if a number is present in a list.

3. Challenge

8. Imagine you need to create a system to store millions of users.
    - Which data structure would you choose for fast searches?
    - Why?

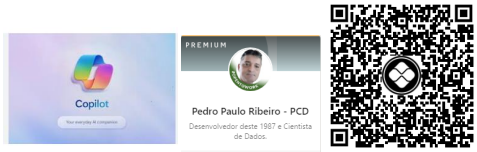
If you want, I can:

- Create **answers to the exercises.**
- Add **examples in Python, Java, or C++.**
- Create **visual diagrams**
- Expand the chapter even further.
- Go to **Chapter 2**

📓 CHAPTER 2 — Algorithm Analysis and Big -O Notation

🎯 Chapter Objectives

By the end of this chapter, you will be able to:



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

- Understanding how to measure algorithm performance.
- Understanding Big O notation -and its variations (Ω and Θ).
- Compare algorithms based on time and space.
- Identify classes of complexity.
- Analyze simple, real-world algorithms.

2.1 Why analyze algorithms?

When two algorithms solve the same problem, which one should you choose?

- The fastest?
- Which uses less memory?
- The simplest?

Algorithm analysis allows us to predict:

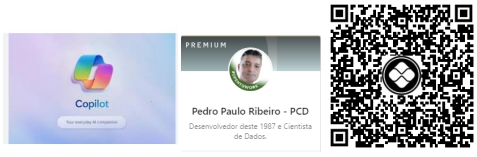
- **Execution time**
- **Memory usage**
- **Scalability**
- **Worst-case scenario behavior**

Without analysis, we would only be "guessing".

2.2 What is Complexity?

Complexity is a way of measuring **how much an algorithm consumes** .

- **Time** (how many operations it performs)
- **Space** (how much memory does it use)



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

We don't measure in seconds because:

- Computers vary in speed.
- Languages vary in performance.
- Implementations vary

Therefore, we use an **abstract measure** , based on the number of operations.

2.3 Big -O Notation

Big -O describes the **worst-case scenario** for an algorithm's execution time.

She replies:

"How does the algorithm grow when the input increases?"

Table of the most common complexities

Complexity	Name	Example
O(1)	Constant	Accessing an array index
O(log n)	Logarithmic	Binary search
O(n)	Linear	Browse list
O(n log n)	Almost linear	Merge Sort
O(n^2)	Quadratic	Bubble Sort
O(2^n)	Exponential	Subsets
O(n!)	Factorial	Permutations

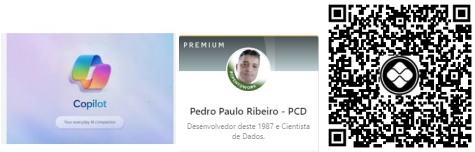
2.4 Intuition of Complexities

O(1) — Constant

Time doesn't change with the size of the entrance.

Example:

- Access `arr[0]`



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

- Insert into a hash table (average)

O(log n) — Logarithmic

Each step reduces the problem by half.

Example:

- Binary search
- Height of a balanced tree

If n = 1,000,000 $\log_2(n) \approx 20$, then only 20 steps!

O(n) — Linear

Time increases proportionally to the size of the input.

Example:

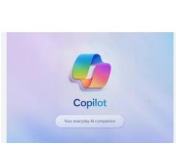
- Scroll through a list
- Find the largest number

O(n log n) — Almost linear

Common in efficient sorting algorithms.

Example:

- Merge Sort
- Quick Sort (average)
- Heap Sort




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

$O(n^2)$ — Quadratic

Two nested loops.

Example:

- Bubble Sort
- Selection Sort
- Check for duplicates using two loops.

$O(2^n)$ — Exponential

It grows absurdly fast.

Example:

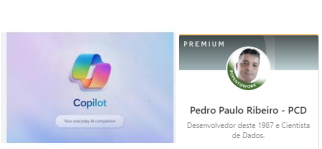
- Solving the knapsack problem by brute force
- Subsets of a set

O(n!) — Factorial

The worst of all worlds.

Example:

- Permutations
- Problems of extreme brute force



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2.5 Big -O, Big Ω and Big Θ Notations

Big -O (O)

- Upper limit
- Worst case

Big -Ω (Ω)

- Lower limit
- Best case

Big -Θ (Θ)

- Exact limit
- Average case (when known)

Example: Binary Search

- Best case: Ω(1) → found in the middle
- Worst case: O(log n)
- Average case: Θ(log n)

2.6 Rules for simplifying Big -O

1. Ignore constants

$O(2n) \rightarrow O(n)$ $O(5n + 10) \rightarrow O(n)$

2. Keep only the dominant term.

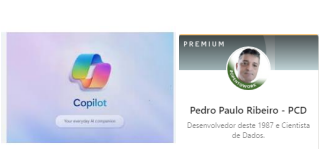
$O(n^2 + n) \rightarrow O(n^2)$

3. Nested loops multiply

Code

```
for i in n:
for j in n:
```

$\rightarrow O(n^2)$




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

4. Sequential loops add up.

Code

```
for i in n:
for j in n:
```

→ O(n + n) → O(n)

5. Divide in half → log n

Code

```
I = n
while I > 1:
I = I / 2
```

→ O(log n)

2.7 Examples of Analysis

Example 1 — Finding the largest number

Code

```
largest = arr[0]
for each element:
if element > larger:
largest = element
```

- Single loop → O(n)
- Space → O(1)

Example 2 — Checking for duplicates (naive)

Code

```
for i in n:
for j in n:
```

```
if arr[i] == arr[j]:
duplicate found
```

- Two loops → $O(n^2)$
- Space → O(1)

Example 3 — Binary Search

Code

```
start = 0
end = n-1
while start <= end:
middle = (start + end) / 2
if arr[meio] == alvo:
return halfway
if [means] < target:
start = middle + 1
if not:
end = middle - 1
```

- Divide by half → O(log n)
- Space → O(1)

2.8 Space Complexity

In addition to time, we analyze memory.

Examples

Algorithm	Space
Linear search	O(1)
Merge Sort	O(n)
Quick Sort	O(log n)
recursive Fibonacci	O(n)

2.9 Conceptual Graphs

Functional growth

Code

```
n! |[redacted] [redacted]
2^n |[redacted]
n² |[redacted]
```

Classical And Quantum Computing

Data Structure e Algorithms

```
n log n |██████
n |████
log n |██
1 |█
```

The higher the value, the greater the difference between efficient and inefficient algorithms.

2.10 Chapter Conclusion

You learned:

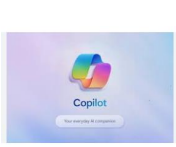
- How to measure performance
- O notation work?-
- Difference between O, Ω and Θ
- How to simplify expressions
- How to analyze real algorithms

You are now ready to move on to **Chapter 3: Basic Data Structures** .

Exercises from Chapter 2

1. Conceptual

1. Explain the difference between O(n) and O(log n).
2. Why do we ignore constants in Big -O notation?
3. What does it mean to say that an algorithm is $O(n^2)$?




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Practical

4. Analyze the complexity of the algorithm:

Code

```
for i in range(n):
for j in range(10):
print(i, j)
```

5. How complex is it to:

Code

```
i = 1
while i < n:
i *= 3
```

6. Determine the complexity:

Code

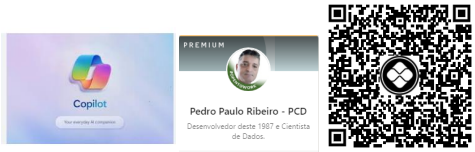
```
for i in range(n):
for j in range(i):
print(i, j)
```

3. Challenge

7. Compare two algorithms:

- Algorithm A: O(n log n)
- Algorithm B: O(n^2)

For n = 1,000,000, which tends to be faster? Explain intuitively.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 3 — Basic Data Structures

Chapter Objectives

By the end of this chapter, you will be able to:

- To have a deep understanding of arrays, lists, strings, and records.
- Knowing when to use each structure.
- Understanding the advantages, limitations, and costs of operations.
- To visualize how these structures are implemented internally.
- Prepare for more complex structures in the coming chapters.

3.1 What are Basic Data Structures?

These are the fundamental structures upon which all others are built. They are simple, straightforward, and extremely efficient.

The main ones are:

- **Arrays (Vectors)**
- **Strings**
- **Records / Structures / Objects**
- **Sequential Lists**

These structures appear in virtually all programs, from operating systems to games and mobile applications.

3.2 Arrays (Vectors)

An **array** is a collection of elements stored **in contiguous memory locations** .

Main features

- Fixed size
- Quick access by index (O(1))

Classical And Quantum Computing

Data Structure e Algorithms

- Costly insertions and removals (O(n))
- Excellent for sequential reading.

Visual representation

Code

```
Index: 0 1 2 3 4
Value: 10 20 30 40 50
```

Operations and costs

Operation	Complexity	Explanation
Access by index	O(1)	Address calculated directly
Insert at the end	O(1)*	If there is space
Insert at the beginning	O(n)	Move all elements
Remove at the end	O(1)	It simply reduces the size.
Remove at startup	O(n)	Move all elements
Search for element	O(n)	Linear search

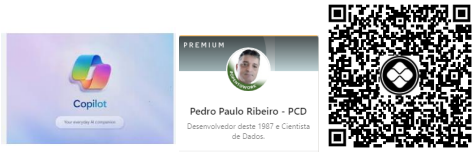
* In dynamic arrays (like Python lists), the amortized cost is O(1).

3.3 Dynamic Arrays

Modern programming languages use arrays that **grow automatically** .

Examples:

- Python → `list`
- Java → `ArrayList`
- C++ → `vector`



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

How do they work?

When the array fills up:

1. A new, larger array is created (usually 2x larger).
2. All elements are copied.
3. The old is discarded.

This makes insertions at the end **very fast on average** .

3.4 Strings

Strings are **sequences of characters** .

Features

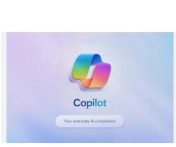
- Many programming languages treat strings as **immutable** (Python, Java).
- Operations like concatenating can be costly.
- Comparisons are O(n).
- Substrings can be O(n) or O(1), depending on the language.

Classic problems with strings.

- Pattern Search (KMP, Rabin -Karp)
- Palindromes
- Compression
- String hashing

3.5 Records, Structs, and Objects

These are structures that group **heterogeneous data** .




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Conceptual example

Code

```
Person:
name: string
age: whole
height: float
```

Why is this useful?

- It represents real-world entities.
- It makes organization easier.
- It allows object-oriented modeling.

Examples in languages

- C → `struct`
- Python → classes
- Java → classes
- C++ → classes and structs

3.6 Sequential Lists

A sequential list is an abstraction built upon arrays.

Examples:

- Python → `list`
- Java → `ArrayList`
- C++ → `vector`

Advantages

- Quick access
- Compact structure
- Good cache location

Classical And Quantum Computing

Data Structure e Algorithms

Disadvantages

- Growth can be costly.
- Insertions in the middle are slow.
- Size may need to be resized.

3.7 Comparison between Arrays, Strings, and Records

Structure	Data type	Advantages	Disadvantages
Array	Homogeneous	Quick access	Fixed size
String	Characters	Text manipulation	It can be immutable.
Record/Object	Heterogeneous	Rich modeling	More memory

3.8 Practical Examples

3.8.1 Finding the largest value in an array

Pseudocode:

Code

```
largest = arr[0]
for each element in arr:
if element > larger:
largest = element
return higher
```

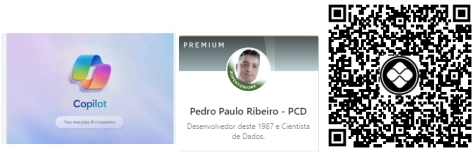
Complexity: O(n)

3.8.2 Counting characters in a string

Code

```
counter = 0
for each character in a string:
counter++
return counter
```

Complexity: O(n)



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

3.8.3 Create a student record

Code

```
Student:
name
age
notice
```

3.9 When to use each structure?

Use arrays when:

- Need quick access by index?
- The size is known.
- The structure is simple.

Use strings when:

- Manipulate text
- You need to compare, search, or transform characters.

Use records/objects when:

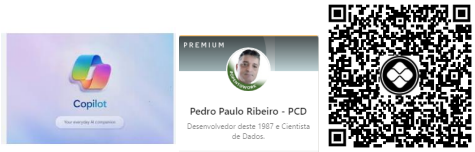
- Models complex entities.
- You need to group different data.

3.10 Chapter Conclusion

In this chapter, you learned:

- What are arrays, strings, and records?
- How they work internally
- Operating costs
- When to use each structure
- Practical examples and analyses

These structures form the basis for the next chapter: **Lists, Stacks, and Queues** , where we will look at more dynamic and flexible structures.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

📝 Exercises from Chapter 3

1. Conceptual

1. Explain the difference between static and dynamic arrays.
2. Why are strings immutable in many languages?
3. What differentiates a record from an array?

2. Practical

4. Write an algorithm that counts how many negative numbers there are in an array.
5. Write an algorithm that concatenates two strings without using pre-built functions.
6. Create a record to represent a store product.

3. Challenge

7. Suppose you need to store 10 million numbers.
    - Would you use a linked list or a dynamic array?
    - Justify your answer based on performance and memory.

📓 CHAPTER 4 — Lists, Stacks, and Queues

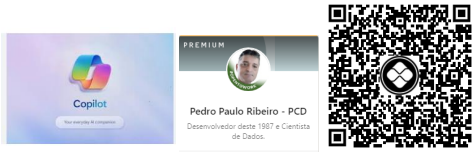
🎯 Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding how linked lists, stacks, and queues work.
- Compare these structures to arrays.
- Knowing when to use each one.
- Understanding operating costs.
- View internal implementations.

4.1 Introduction to Dynamic Linear Structures

Unlike arrays, which have a fixed size and occupy contiguous memory, the structures in this chapter are **dynamic** , growing and shrinking as needed.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

They are essential for:

- Operating systems
- Compilers
- Browsers
- Games
- Graph algorithms
- Real-time data processing

4.2 Linked Lists

A **linked list** is formed by **nodes** , where each node contains:

- A value
- A pointer to the next node

Visual representation

Code

```
[10 | *] → [20 | *] → [30 | *] → None
```

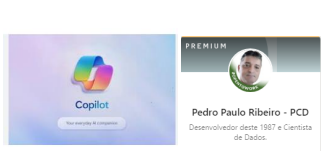
Features

- Dynamic growth
- Quick insertions and removals at the beginning.
- Slow access by index
- It does not occupy contiguous memory.

4.2.1 Types of Linked Lists

1. Singly linked list

Each node points to the next.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Doubly linked list

Each node points to the next and the previous one.

Code

```
None ← [10] ↔ [20] ↔ [30] → None
```

3. Circular list

The last node points to the first.

4.2.2 Operations and Complexity

Operation	Complexity	Explanation
Insert at the beginning	O(1)	Just change the hands.
Insert at the end	O(n)	You need to go through the list.
Remove at startup	O(1)	Update the header
Remove at the end	O(n)	Go all the way to the second-to-last one.
Search for element	O(n)	It runs sequentially.
Access by index	O(n)	There is no direct access.

4.2.3 Advantages and Disadvantages

✔ Advantages

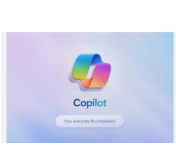
- Dynamic growth
- Quick insertions at the beginning
- Great for structures like stacks and rows.

✘ Disadvantages

- Slow access
- Increased memory usage (pointers)
- It doesn't make good use of the CPU cache.

4.3 Stacks

A **battery** follows the principle:




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

LIFO — Last In, First Out. The last one in is the first one out.

Real-world examples

- Stack of plates
- Browsing history
- Function calls (call stack)

Visual representation

Code

```
Top → [30]
[20]
[10]
```

4.3.1 Main Operations

Operation	Description	Complexity
`push(x)`	Insert at the top	O(1)
`pop()`	Remove from top	O(1)
`peek()`	See the top	O(1)
`isEmpty()`	Check if it's empty.	O(1)

4.3.2 Applications of Batteries

- Evaluation of mathematical expressions
- Balanced parenthesis check
- Backtracking algorithms
- DFS (depth-fed search)
- Undo/redo actions

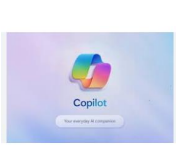
4.4 Queues

A **queue** follows the principle:

FIFO — First In, First Out. The first one in is the first one out.

Real-world examples

- Bank queue
- Document printing
- Task processing




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Visual representation

Code

```
Front → [10] → [20] → [30] → Rear
```

4.4.1 Main Operations

Operation	Description	Complexity
`enqueue(x)`	Insert at the end	O(1)
`dequeue()`	Remove from the beginning	O(1)
`peek()`	Consult the first one	O(1)
`isEmpty()`	Check if it's empty.	O(1)

4.4.2 Circular Queues

A circular queue avoids wasting space in arrays.

Code

```
[30][ ][ ][10][20]
Up Up
rear front
```

4.4.3 Queueing Applications

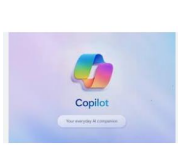
- Customer service systems
- Simulations
- Breadth-first search (BFS) algorithms
- Streaming buffers
- Asynchronous processing

4.5 Deques (Double-Ended Queues)

A **deck** allows for insertion and removal at both the beginning and the end.

Operations

- `push_front`
- `push_back`
- `pop_front`
- `pop_back`




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Applications

- Sliding window algorithm
- Implementation of optimized BFS
- Hybrid structures

4.6 General Comparison

Structure	Insertion	Removal	Access	Typical use
Linked list	Fast start	Fast start	Slow	Dynamic structures
Battery	Quick at the top	Quick at the top	Top only	Recursion, history
Queue	Quick at the ends	Quick at the ends	Just the ends	Sequential processing
Deck	Quick at the ends	Quick at the ends	Just the ends	Advanced algorithms

4.7 Practical Examples

4.7.1 Check balanced parentheses (stack)

Code

```
for each character:
if it is '(':
push
if it is ')':
pop
if empty stack → valid
```

4.7.2 Simulate a service queue

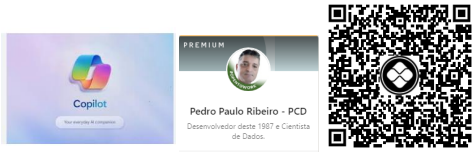
Code

```
enqueue(client)
dequeue() → next client
```

4.7.3 Insert at the beginning of a linked list

Code

```
new.no = value
new.next = head
head = new
```



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

4.8 Chapter Conclusion

In this chapter, you learned:

- How linked lists, stacks, and queues work.
- Its advantages and limitations
- When to use each structure
- Real-world applications
- Operations and complexities

These structures are essential for the next chapter: **Trees** , where we will see powerful hierarchical structures.

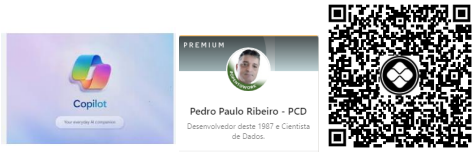
📝 Exercises from Chapter 4

1. Conceptual

1. Explain the difference between LIFO and FIFO.
2. Why do linked lists have slow index access?
3. Name three applications of batteries.

2. Practical

4. Write the pseudocode for a `push` and `pop function` on a stack.
5. Draw a model of a supermarket checkout line.
6. Write an algorithm that counts how many elements are in a linked list.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 5 — Trees

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding the concept of trees and their importance.
- Understanding binary trees, paths, and properties.
- Differentiate between general, binary, complete, full, and perfect trees.
- Apply DFS and BFS paths.
- Preparing for Balanced Trees (Chapter 6).

5.1 What is a tree?

A **tree** is a hierarchical data structure composed of **nodes** connected by **edges** .

Each node contains:

- A value
- Zero or more children

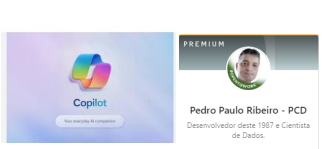
Fundamental characteristics

- There is a special node called **the root** .
- Each node (except the root) has exactly **one parent** .
- There are no cycles.

Visual representation

Code

```
A (root)
/ \
BC
/ \ \
DEF
```



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

5.2 Essential Terminology

Term	Meaning
Source	Initial node of the tree
Father	A knot above another knot.
Son	One knot below another knot.
Sheet	A knot without children
Height	Greatest distance from the root to a leaf
Depth	Distance from a node to the root.
Subtree	A tree within a tree.

5.3 General Trees vs. Binary Trees

General Trees

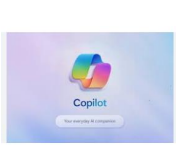
- Each node can have **any number of children** .
- Used in file systems, XML/HTML, games, AI.

Binary Trees

- Each node has **a maximum of 2 children** :
  - Left
  - Right

They are the most studied because:

- They are simple.
- They enable efficient algorithms.
- They are the basis for advanced structures.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

5.4 Types of Binary Trees

1. Complete Binary Tree

All levels are complete, except possibly the last one.

Code

```
THE
/ \
BC
/ \ /
DEF
```

2. Full Binary Tree

Each node has 0 or 2 children.

Code

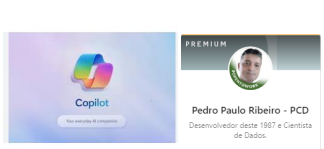
```
THE
/ \
BC
/ \ / \
DEFG
```

3. Perfect Binary Tree

All levels are complete.

Code

```
THE
/ \
BC
/ \ / \
DEFG
```




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

4. Degenerate Binary Tree

Each node has only one child (it becomes a list).

Code

```
THE
\
B
\
W
```

5.5 Representation of Binary Trees

1. Using nodes and pointers

Code

```
class Node:
value
left
right
```

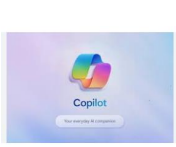
2. Using arrays (for complete trees)

Code

```
Index: 1 2 3 4 5 6 7
Value: ABCDEFG
```

Rules:

- Father(i) = i // 2
- LeftChild(i) = 2i
- FilhoDir(i) = 2i + 1




Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

5.6 Tree Traversal

There are two categories:

- **DFS (Depth and First Survey)**
- **BFS (Breadth-First Search)**

5.7 DFS — Depth-First Search

Explore as much as possible before returning.

There are 3 classic routes:

1. Pre-order

Visit: **root → left → right**

Code

```
ABDECF
```

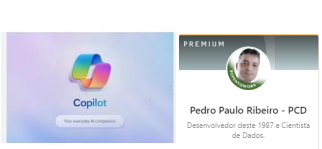
2. In-order

Visit: **left → root → right**

Code

```
DBEAFC
```

Used in **binary search trees** to obtain sorted values.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

3. Post-order

Visit: **left → right → root**

Code

```
DEBFCA
```

5.8 BFS — Breadth-First Search (Level by Level)

Visit the tree level by level.

Code

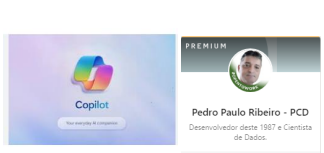
```
ABCDEF
```

Implemented with **a queue (queue)** .

5.9 Height and Depth

Height of a knot

Greater distance to a leaf.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Node depth

Distance to the root.

Example

Code

```
THE
/ \
BC
/
D
```

- Depth (D) = 2
- Height(B) = 1
- Height (A) = 2

5.10 Binary Search Trees (BST)

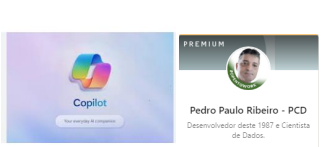
A **BST** follows the rule:

- Smaller values → left
- Larger values → right

Example:

Code

```
8
/ \
3 10
/ \ \
1 6 14
```



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Operations

Operation	Medium complexity	Worst case
Insert	O(log n)	O(n)
Search	O(log n)	O(n)
Remove	O(log n)	O(n)

The worst-case scenario occurs when the tree becomes **unbalanced** .

In the next chapter we will see how to avoid this with:

- AVL
- Red-Black Trees
- Heaps

5.11 Applications of Trees

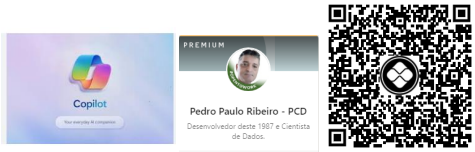
- File systems
- Compilers (syntax trees)
- Databases (B-trees and B+ trees)
- Games (decision trees)
- Compression (Huffman tree)
- Networks (routing trees)
- Artificial intelligence

5.12 Chapter Conclusion

In this chapter, you learned:

- What are trees and why are they important?
- Types of binary trees
- DFS and BFS routes
- Height, depth and properties
- Binary search trees

Now you are ready for **Chapter 6: Balanced Trees and Heaps** , where we will see how to maintain efficient trees even in extreme cases.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Exercises from Chapter 5

1. Conceptual

1. Explain the difference between a general tree and a binary tree.
2. What is a leaf?
3. What is the difference between pre-order, in-order, and post-order?

2. Practical

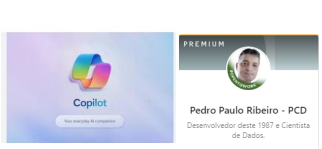
4. Draw a binary tree with 7 nodes and write its DFS (Direct-Flow-Forward) paths.
5. Given the tree below, calculate the height and depth of each node:

Code

```
THE
/ \
BC
/ \
OF
```

3. Challenge

6. Construct a binary search tree by inserting the values: `8, 3, 10, 1, 6, 14, 4, 7`
    - Draw the tree.
    - Take the DFS routes
    - Calculate the height of the tree.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 6 — Balanced Trees and Heaps

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding why trees need to be balanced.
- Understanding AVL and Red -Black Trees.
- Understanding how automatic balancing works.
- Understanding heaps and priority queues.
- Applying heaps in algorithms like Heap Sort and Dijkstra's Sort.

6.1 Why Do Trees Need to Be Balanced?

A **binary search tree (BST)** works well when it is balanced:

- Search: O(log n)
- Insertion: O(log n)
- Removal: O(log n)

But in the worst case, it could turn into a **linked list** :

Code

```
1
\
2
\
3
\
4
```

And then:

- Search: O(n)
- Insertion: O(n)
- Removal: O(n)

To avoid this, **balanced trees are developed** , which guarantee a height close to log n.

6.2 AVL Trees

Created by Adelson -Velsky and Landis, these are the first balanced trees in history.

Fundamental property

For each node:

Code

```
|height(left) - height(right)| ≤ 1
```

If this condition is violated, the tree **automatically rebalances itself** .

6.2.1 Balancing Factor (BF)

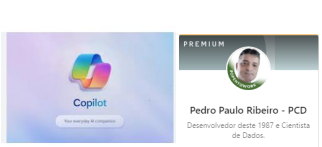
Code

```
FB = height(left) - height(right)
```

Possible values:

- -1
- 0
- +1

If FB moves out of this range, rotation occurs.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

6.2.2 Rotations in AVL

There are 4 types:

1. Simple right rotation (RR)

Corrects the case of **left- handedness -**.

2. Simple left rotation (LL)

Corrects the case **of right -right** .

3. Double left- right rotation -(LR)

left- -right case .

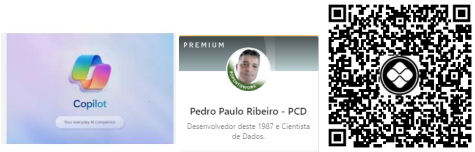
4. Double rotation right -left (RL)

Corrects the case of **right --left** .

6.2.3 Complexity

Operation	Complexity
Insert	O(log n)
Search	O(log n)
Remove	O(log n)

AVL is extremely efficient for **searches** .



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

6.3 Red -Black Trees

More flexible than AVL, used in:

- `TreeMap` and `TreeSet` (Java)
- `std::map` and `std::set` (C++)
- Database implementations

Properties

1. Each knot is either **red** or **black** .
2. The root is always **black** .
3. We reds cannot have red children.
4. Every path from the root to a null leaf has the same number of black nodes.

These rules ensure that the tree never becomes too unbalanced.

6.3.1 Why -are Red Black Trees popular?

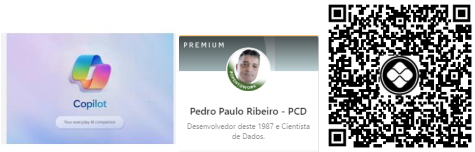
- Insertions and removals are simpler than AVL.
- They maintain a height close to log n.
- They are great for **mixed operations** (lots of insertions and searches).

6.3.2 Complexity

Operation	Complexity
Insert	O(log n)
Search	O(log n)
Remove	O(log n)

6.4 Comparison: AVL vs Red -Black

Criterion	AVL	Red -Black
Balancing	More rigid	More flexible
Search speed	Better	Good
Insertion/removal	Slower	Faster
Typical use	Search engines	General purpose structures



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

6.5 Heaps

A **heap** is a complete binary tree used to implement **priority queues** .

There are two types:

- **Max -heap** : parent ≥ children
- **Minimum -heap** : parent ≤ children

Visual representation (min -heap)

Code

```
1
/ \
3 5
/ \ /
4 8 10
```

6.5.1 Heap Properties

1. **Complete tree**
    - All levels completed, except the last one.
2. **Heap property**
    - Minimum -heap: parent ≤ children
    - Max -heap: parent ≥ children

6.5.2 Array Implementation

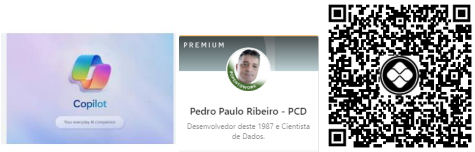
Heaps are almost always implemented using arrays:

Code

```
Index: 1 2 3 4 5 6
Value: 1 3 5 4 8 10
```

Rules:

- Father(i) = i // 2
- LeftChild(i) = 2i
- FilhoDir(i) = 2i + 1



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

6.5.3 Operations

Insert (push)

1. Put it at the end.
2. Heapify -up

Remove the top (pop)

1. Top swap with last
2. Remove last one
3. Go down (heapify -down)

Complexity

Operation	Complexity
Insert	O(log n)
Remove	O(log n)
Access top	O(1)

6.6 Heap Sort

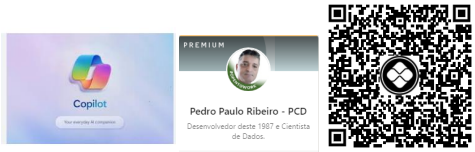
One of the most efficient sorting algorithms.

Steps

1. Build a max -heap
2. Repeatedly remove the largest element
3. Place at the end of the array.

Complexity

- Time: O(n log n)
- Space: O(1)



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

6.7 Heap Applications

- Priority queues
- Dijkstra's algorithm
- Task scheduling
- Huffman compression
- Simulations
- Operating systems

6.8 Chapter Conclusion

In this chapter, you learned:

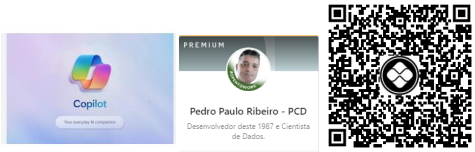
- Why trees need to be balanced
- How do AVL and Red -Black Trees work?
- How heaps implement priority queues
- Heap Sort and practical applications

Now you are ready for **Chapter 7: Hash Tables** , one of the most important structures in modern computing.

📝 Exercises from Chapter 6

1. Conceptual

1. Why is an unbalanced BST bad?
2. Explain the balancing factor of an AVL tree.
3. What is the difference between min -heap and max heap?



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Practical

4. Enter the values `10, 20, 5, 6, 1` into a min -heap and draw the result.
5. Enter the values `30, 20, 40, 10` into an AVL table and display the required rotations.
6. Explain how the heap is represented in an array.

3. Challenge

7. Create a max -heap with the values: `15, 3, 17, 10, 84, 19, 6, 22, 9`
    - Show each step of heapify.
    - Show the final heap.

📓 CHAPTER 7 — Hash Tables

🎯 Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding what hashing is and why it's so powerful.
- Understanding how hash tables work.
- Knowing how to handle collisions.
- Understanding good and bad hash functions.
- Compare hash tables with other structures.
- Applying hash tables to real-world problems.

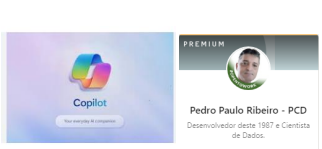
7.1 What is Hashing?

Hashing is a technique that transforms an input (key) into an **index** within a table.

The transformation is done using a **hash function** :

Code

```
index = hash(key)
```



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Simple example

Code

```
hash("Pedro") → 42
hash("Maria") → 17
hash("João") → 89
```

The hash table uses this index to store and retrieve values **in O(1) time** on average.

7.2 What is a Hash Table?

A hash table is a structure that stores pairs of:

Code

```
(key → value)
```

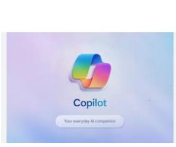
It allows:

- Quick insertion
- Quick search
- Quick removal

Visual representation

Code

```
Index | Value
----------------------
0 | -
1 | -
2 | ("John", 30)
3 | -
4 | ("Peter", 25)
5 | -
```




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

7.3 Hash Functions

The hash function is the heart of the table.

A good hash function should:

- Be quick
- Distribute values evenly
- Avoid collisions

Examples of hash functions

- String hash (sum of characters, polynomial, DJB2)
- Hash of numbers (modular)
- Cryptographic hash (SHA -256, MD5) — **not used in common hash tables.**

7.4 Collisions

A collision occurs when two different keys generate the same index:

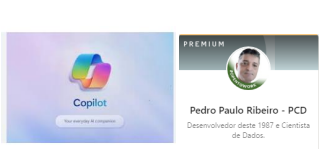
Code

```
hash("Pedro") → 4
hash("Ana") → 4
```

Collisions are inevitable. The important thing is **how to handle -them** .

7.5 Collision Handling Techniques

There are two main categories:




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

1. Chaining

Each position in the table contains a **linked list** .

Code

```
Index 4 → ("Pedro", 25) → ("Ana", 19)
```

Advantages

- Simple
- Easy to implement
- It grows dynamically.

Disadvantages

- It can degrade to O(n) if many collisions occur.

2. Open Addressing

If the position is occupied, he looks for another one.

Methods:

a) Linear Probing

Code

```
me, me+1, me+2, ...
```

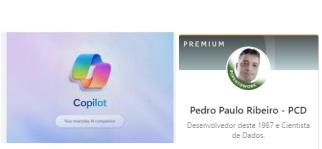
b) Quadratic Probing

Code

```
i, i+1², i+2², i+3², ...
```

c) Double Hashing

It uses a second hash function to skip positions.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

7.6 Load Factor

The load factor is:

Code

```
α = number_of_elements / table_size
```

When α gets high (e.g., > 0.75), the table is **resized** .

Resizing

1. Create a new, larger table.
2. Recalculate the hash of all elements.
3. Insert again

7.7 Complexity

Operation	Average	Worst case
Insert	O(1)	O(n)
Search	O(1)	O(n)
Remove	O(1)	O(n)

The worst-case scenario occurs when:

- The hash function is bad.
- The table is very full.
- Many collisions occur

7.8 Hash Tables in Real-World Languages

Language	Structure
Python	`dict`
Java	`HashMap`
C++	`unordered_map`
JavaScript	Objects and `Maps`
Go	`map`
W	Manual implementation

They all use hashing internally.

7.9 Applications of Hash Tables

- Dictionaries
- Database indexing
- Caches (LRU, LFU)
- Frequency count
- Duplicate detection
- Compilers (symbol tables)
- File systems
- Cryptography (secure hashing)

7.10 Practical Examples

7.10.1 Counting word frequency

Code

```
for each word:
table[word]++
```

7.10.2 Check for duplicates

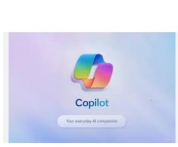
Code

```
for each element:
If an element is in a table:
duplicate found
if not:
table[element] = True
```

7.10.3 Implement simple caching

Code

```
cache[key] = value
```




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

7.11 Comparison with Other Structures

Structure	Search	Insertion	Removal	Ordering
Array	O(n)	O(n)	O(n)	Yes
Linked list	O(n)	O(1)	O(1)	No
BST	O(log n)	O(log n)	O(log n)	Yes
Hash Table	O(1)	O(1)	O(1)	No

7.12 Chapter Conclusion

In this chapter, you learned:

- What is hashing?
- How hash tables work
- How to deal with collisions
- What is load factor?
- Real-world applications
- Comparisons with other structures

Now you are ready for **Chapter 8: Graphs** , one of the most powerful structures in computing.

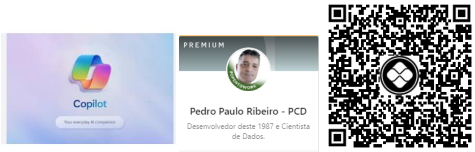
📝 Exercises from Chapter 7

1. Conceptual

1. What is a hash function?
2. Why are collisions inevitable?
3. Explain the difference between chaining and open addressing.

2. Practical

4. Given the hash `h(x) = x mod 10` , enter the values: `12, 22, 32, 42` using **linear probing** .
5. Repeat the exercise using **chaining** .
6. Calculate the load factor for a table with 50 positions and 35 elements.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

3. Challenge

7. Design a simple hash function for strings and explain why it is efficient.

CHAPTER 8 — Graphs

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding what graphs are and why they are so powerful.
- Understanding vertices, edges, weights, and directionality.
- Differentiate between simple, complete, weighted, and directed graphs.
- Representing graphs in memory (matrix and adjacency list).
- Apply BFS and DFS paths.
- Preparing for advanced algorithms (Chapter 13).

8.1 What is a Graph?

A **graph** is a structure composed of:

- **Vertices (nodes)**
- **Edges (connections between nodes)**

Visual representation

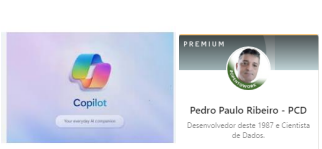
Code

```
A ── B ── C
\ \
And
\
D
```

Graphs model **relationships** between entities.

8.2 Essential Terminology

Term	Meaning
Vertex	A point (node)




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Edge	Connection between two vertices
Directed graph	Edges have direction.
Weighted graph	Edges have weight.
Path	Sequence of connected vertices
Cycle	A path that returns to the beginning.
Degree	Number of edges connected to a vertex
Component	Connected subgraph

8.3 Types of Graphs

1. Undirected Graph

Edges have no direction.

Code

```
A — B
```

2. Directed Graph (Digraph)

Edges have direction.

Code

```
A → B
```

3. Weighted Graph

Edges have weights.

Code

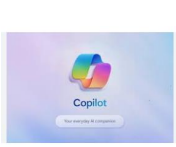
```
A —5— B
```

4. Complete Graph

All vertices are connected.

5. Bipartite Graph

Vertices divided into two groups with no internal connections.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

6. Cyclic and Acyclic Graphs

- Cyclic → has cycles
- Acyclic → without cycles (e.g., DAG — Directed Acyclic Graph)

8.4 Graph Representations

There are two main forms:

1. Adjacency Matrix

An NxN matrix where:

- 1 indicates connection
- 0 indicates absence

Example

Code

```
ABC
A [ 0 1 1 ]
B [ 1 0 0 ]
C [ 1 0 0 ]
```

Advantages

- Quick access O(1)
- Simple

Disadvantages

- It takes up a lot of space $O(n^2)$
- Inefficient for sparse graphs

2. Adjacency List

Each vertex stores a list of neighbors.

Code

```
A → B, C
```

Classical And Quantum Computing

Data Structure e Algorithms

```
B → A
C → A
```

Advantages

- It takes up little space O(V + E)
- Ideal for large graphs.

Disadvantages

- Checking if an edge exists is O(degree)

8.5 Paths in Graphs

Two fundamental paths:

- **DFS (Depth-First Search)**
- **BFS (Breadth-First Search)**

They are the basis for algorithms such as:

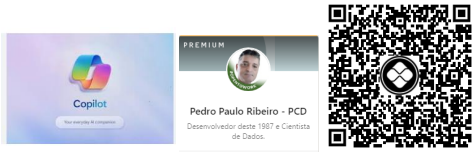
- Dijkstra
- Kruskal
- First
- Search for cycles
- Connected components
- Topological ordering

8.6 DFS — Depth-First Search

Explore the graph as deeply as possible before returning.

Strategy

- Uses **a stack** (or recursion)
- Ideal for detecting cycles.
- Useful for connectivity issues.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Typical order

Code

```
A → B → D → E → C
```

Complexity

O(V + E)

8.7 BFS — Breadth-First Search

layered graph .

Strategy

- Use **the queue**
- Find the shortest path in unweighted graphs.
- Excellent for minimum distance problems.

Typical order

Code

```
A → B → C → D → E
```

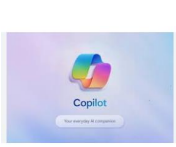
``

📘 CHAPTER 9 — Sorting Algorithms

🎯 Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding the main sorting algorithms.
- Compare the performance, complexity, and ideal use of each.
- Understanding comparative and non-comparative orderings.
- Visualize how each algorithm works step by step.
- Choosing the ideal algorithm for each situation.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

9.1 What is Ordering?

Ordering means **rearranging elements** in a specific sequence, usually ascending or descending.

Everyday examples:

- Sort contacts by name
- Sort products by price
- Sort files by date
- Sort search results

Sorting is so important that many algorithms depend on it to function.

9.2 Classification of Sorting Algorithms

There are two main categories:

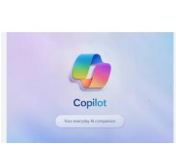
1. Comparative Orderings

They compare elements with each other.

Examples:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort

Theoretical limit: **O(n log n)** in the best case.




Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Non-Comparative Orderings

They don't compare elements directly.

Examples:

- Counting Sort
- Radix Sort
- Bucket Sort

They can be **O(n)** in specific cases.

9.3 Simple Sorting Algorithms (Didactic)

These algorithms are easy to understand, but not very efficient.

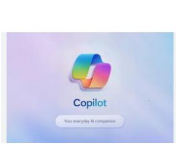
9.3.1 Bubble Sort

Compare adjacent pairs and swap them if they are out of order.

Visual example

Code

```
[5, 3, 8, 4]
→ compare 5 and 3 → swap
[3, 5, 8, 4]
→ compare 8 and 4 → swap
[3, 5, 4, 8]
```




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Complexity

Case	Complexity
Better	O(n)
Average	$O(n^2)$
Worse	$O(n^2)$

Use

- For educational purposes only.

9.3.2 Selection Sort

Select the smallest element and place it in the correct position.

Visual example

Code

```
[5, 3, 8, 4]
smallest = 3 → swap with 5
[3, 5, 8, 4]
smaller = 4 → swap with 5
[3, 4, 8, 5]
```

Complexity

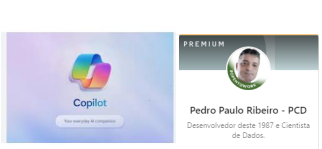
Always $O(n^2)$, regardless of the input.

Use

- Simple, but inefficient.

9.3.3 Insertion Sort

Inserts each element into the correct position within the already sorted part.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Visual example

Code

```
[5, 3, 8, 4]
[3, 5] 8 4
[3, 5, 8] 4
[3, 4, 5, 8]
```

Complexity

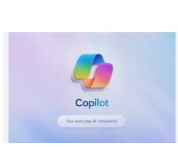
Case	Complexity
Better	O(n)
Average	$O(n^2)$
Worse	$O(n^2)$

Use

- Great for short lists.
- Great for almost ordered lists.

9.4 Efficient Sorting Algorithms

Now we move on to the algorithms used in practice.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

9.4.1 Merge Sort

Based on **divide and conquer** .

Steps

1. Divide the array in half.
2. Order each half
3. Merge the two ordered halves.

Visual example

Code

```
[5, 3, 8, 4]
→ [5, 3] and [8, 4]
→ [3, 5] and [4, 8]
→ [3, 4, 5, 8]
```

Complexity

Case	Complexity
Better	O(n log n)
Average	O(n log n)
Worse	O(n log n)

Use

- Very stable
- Excellent for handling large volumes of data.

- Used in databases.

9.4.2 Quick Sort

It also uses divide and conquer, but chooses a **pivot** .

Steps

1. Choose a pivot.
2. Place smaller items on the left.
3. Place larger ones on the right.
4. Recursion

Visual example

Code

```
Pivot = 5
[3, 4] 5 [8, 9]
```

Complexity

Case	Complexity
Better	O(n log n)
Average	O(n log n)
Worse	$O(n^2)$

The worst case occurs when the pivot is always the smallest or largest element.

Use

- Very fast in practice.
- Used in standard libraries

Classical And Quantum Computing

Data Structure e Algorithms

9.4.3 Heap Sort

Use a **heap** for sorting.

Steps

1. Builds a max -heap
2. Remove the largest element.
3. Place it at the end of the array.

Complexity

Always O(n log n)

Use

- It does not use extra memory.
- Very efficient

9.5 Non-Comparative Algorithms

9.5.1 Counting Sort

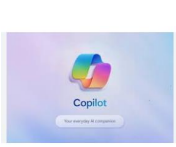
It only works for small numbers.

Steps

1. Count how many times each number appears.
2. Rebuilds the sorted array.

Complexity

O(n + k), where k is the largest value.




Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

9.5.2 Radix Sort

Sort digit by digit.

Example

Code

```
170, 45, 75, 90, 802, 24, 2, 66
```

Sort by units → tens → hundreds.

Complexity

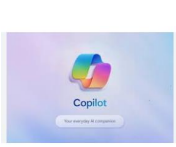
O(d × (n + k))

9.5.3 Bucket Sort

Divide the elements into "buckets" and sort each bucket.

Use

- Uniformly distributed data
- Very fast in specific cases.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

9.6 General Comparison of Algorithms

Algorithm	Better	Average	Worse	Stable	Use
Bubble	O(n)	$O(n^2)$	$O(n^2)$	Yes	Educational
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Educational
Insertion	O(n)	$O(n^2)$	$O(n^2)$	Yes	Short lists
Merge	O(n log n)	O(n log n)	O(n log n)	Yes	Big lists
Quick	O(n log n)	O(n log n)	$O(n^2)$	No	General
Heap	O(n log n)	O(n log n)	O(n log n)	No	Systems
Counting	O(n+k)	O(n+k)	O(n+k)	Yes	Small numbers
Radix	O(nk)	O(nk)	O(nk)	Yes	Structured data

9.7 Chapter Conclusion

In this chapter, you learned:

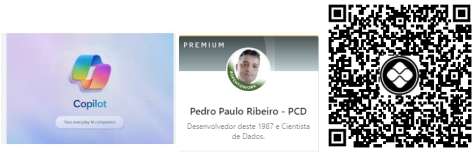
- The main sorting algorithms
- Differences between comparative and non-comparative orderings
- When to use each algorithm
- Complexities and characteristics
- Practical examples

Now you are ready for **Chapter 10: Search Algorithms** , where we will see how to find elements efficiently.

📝 Exercises from Chapter 9

1. Conceptual

1. Explain the difference between Quick Sort and Merge Sort.
2. Why might Counting Sort be faster than Quick Sort?
3. What does it mean for an algorithm to be stable?



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Practical

4. Manually perform an Insertion Sort on the array: `[7, 3, 5, 2]`
5. Perform Quick Sort on the array: `[10, 7, 8, 9, 1, 5]`
6. Create a max -heap and apply Heap Sort to the array: `[4, 10, 3, 5, 1]`

3. Challenge

7. Compare Quick Sort and Heap Sort in terms of:
    - Practical speed
    - Memory usage
    - Stability
    - Ideal cases

CHAPTER 10 — Search Algorithms

Chapter Objectives

By the end of this chapter, you will be able to:

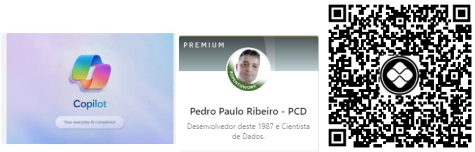
- Understanding the main search algorithms.
- Differentiate between linear search and binary search.
- Understanding tree and graph searches.
- Knowing when to use each type of search.
- Analyze complexity and performance.

10.1 What is Search?

Searching means **finding an element** within a collection of data.

Everyday examples:

- Search for a contact on mobile phone
- Searching for a file on the computer
- Searching for a product on a website
- Searching for a word in a text



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

The efficiency of the search depends on the **data structure** and the **algorithm** used.

10.2 Linear Search

It's the simplest method: it goes through all the elements until it finds the target.

Pseudocode

Code

```
for i from 0 to n-1:
if arr[i] == target:
return i
return -1
```

Example

Code

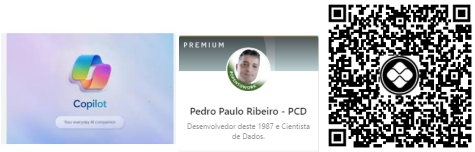
```
List: [4, 7, 1, 9]
Search: 1
→ travels 4, 7, finds 1
```

Complexity

Case	Complexity
Better	O(1)
Average	O(n)
Worse	O(n)

When to use it?

- Short list
- Unordered list
- Structure that does not allow quick access.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

10.3 Binary Search

It only works on **sorted lists** .

Strategy

1. Find the middle
2. Compare to the target.
3. Decide whether to search left or right.
4. Repeat until you find it.

Visual example

Search 23 in:

Code

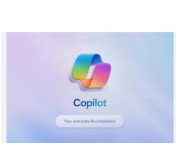
```
[10, 15, 20, 23, 30, 40]
```

1. Mean = 20 → target > 20
2. Search to the right
3. Mean = 30 → target < 30
4. Search to the left
5. Find 23

Pseudocode

Code

```
start = 0
end = n-1
while start <= end:
middle = (start + end) // 2
if arr[meio] == alvo:
return halfway
if [means] < target:
start = middle + 1
if not:
end = middle - 1
return -1
```




Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Complexity

Case	Complexity
Better	O(1)
Average	O(log n)
Worse	O(log n)

When to use it?

- Sorted list
- Structure with fast access (array)
- Frequent search

10.4 Tree Search

Trees enable efficient searches, especially **binary search trees (BSTs)** .

10.4.1 Search in BST

Rule:

- If target < node → go to the left
- If target > node → go to the right

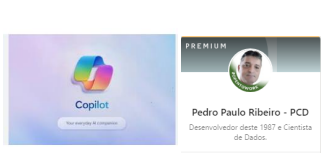
Example

Code

```
8
/ \
3 10
/ \ \
1 6 14
```

Search 6:

- 6 < 8 → left
- 6 > 3 → right
- It found



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Complexity

Case	Complexity
Better	O(1)
Average	O(log n)
Worse	O(n) (degenerate tree)

10.5 Graph Search

Graphs require specific algorithms:

- **DFS (Depth-First Search)**
- **BFS (Breadth-First Search)**

10.5.1 DFS — Depth-First Search

Explore the graph as deeply as possible before returning.

Uses

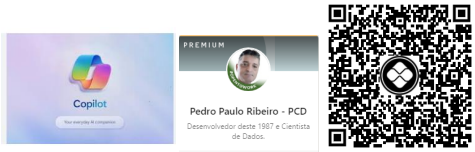
- Detect cycles
- Find connected components
- Solving mazes

Complexity

O(V + E)

10.5.2 BFS — Breadth-First Search

Explore the graph layer by layer.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Uses

- Finding the shortest path in unweighted graphs
- Recommendation systems
- Signal propagation

Complexity

O(V + E)

10.6 Searching Hash Tables

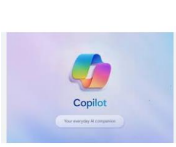
Hash tables allow **O(1) search** on the mean.

Strategy

1. Calculates the key hash.
2. Direct access to the index.
3. Resolves collisions if necessary.

Complexity

Case	Complexity
Average	O(1)
Worse	O(n)




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

10.7 General Comparison of Search Algorithms

Structure	Algorithm	Complexity	Observations
Array	Linear	O(n)	Simple
sorted array	Binary	O(log n)	Very fast
BST	BST Search	O(log n)	It can degrade.
AVL / Red -Black	Search	O(log n)	Always efficient
Hash Table	Hashing	O(1)	Best average
Graph	BFS/DFS	O(V+E)	It depends on the structure.

10.8 Practical Examples

10.8.1 Check if a number exists in a list

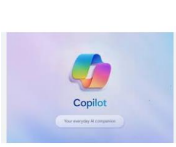
- Small list → linear
- Large sorted list → binary

10.8.2 Finding the shortest path on a map

- Uses BFS (unweighted)
- Usa Dijkstra (weighted)

10.8.3 Search for user on social network

- Hash table for ID
- Graph for connections




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

10.9 Chapter Conclusion

In this chapter, you learned:

- Linear and binary search
- Tree and graph searches
- Search in hash tables
- Comparison between algorithms
- Real-world applications

Now you are ready for **Chapter 11: Dynamic Programming** , one of the most important and challenging topics in the book.

Exercises from Chapter 10

1. Conceptual

1. Explain the difference between linear search and binary search.
2. Why does binary search require an ordered list?
3. What is the difference between BFS and DFS?

2. Practical

4. Manually perform a binary search on the array: `[2, 5, 8, 12, 16, 23, 38]` Search for the value `16` .
5. Given the BST below, find the path to retrieve the value 7:

Code

```
10
/ \
5 15
/ \ / \
2 7 12 20
```

6. In a simple graph, list the visit order using BFS and DFS starting from vertex A:

Code

```
A — B — C
| \
OF
```

3. Challenge

7. Compare binary search and hashing in terms of:
   - Speed
   - Memory
   - Ideal cases
   - Limitations

CHAPTER 11 — Dynamic Programming

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding what Dynamic Programming (DP) is.
- Identify problems that can be solved with DP.
- Understanding memorization and tabulation.
- Solving classic problems such as Fibonacci, Knapsack, and LCS.
- Transforming slow, recursive solutions into efficient solutions.

11.1 What is Dynamic Programming?

Dynamic Programming is a technique for solving complex problems **by breaking them down into smaller subproblems** , solving each subproblem **only once** , and **reusing** the results.

It is used when:

- The problem can be divided into smaller subproblems.
- The sub-problems are repeated.
- The solution to the larger problem depends on the solutions to the smaller problems.

Central idea

Save intermediate results to avoid repeating the work.

11.2 When to use Dynamic Programming?

DP is ideal when the problem has:

1. Overlapping subproblems

The same subproblem appears multiple times.

Example: Fibonacci

Code

```
fib(5) calls fib(4) and fib(3)
fib(4) calls fib(3) and fib(2)
fib(3) calls fib(2) and fib(1)
```

2. Optimal structure

The optimal solution can be constructed from smaller optimal solutions.

Example: Shortest path in an acyclic graph.

11.3 Dynamic Programming Approaches

There are two main forms:


Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

1. Memorization (Top -Down)

- It starts with the biggest problem.
- Solve recursively
- Stores results in cache.
- Avoids recomputations

Analogy

It's like asking someone for information and keeping the answer to yourself so you don't have to ask again.

2. Tabulation (Bottom -Up)

- It starts with the smallest sub-problems.
- Fill in a table.
- Build the final solution.

Analogy

It's like putting together a -jigsaw puzzle starting with the small pieces.

11.4 Classic Example: Fibonacci

11.4.1 Naive Recursive Solution

Code

```
fib(n):
if n <= 1: return n
return fib(n-1) + fib(n-2)
```

Complexity

$O(2^n)$ — extremely slow.

Classical And Quantum Computing

Data Structure e Algorithms

11.4.2 Fibonacci with Memorization

Code

```
fib(n):
if n is in cache: return cache[n]
if n <= 1: return n
cache[n] = fib(n-1) + fib(n-2)
return cache[n]
```

Complexity

O(n)

11.4.3 Fibonacci with Tabulation

Code

```
dp[0] = 0
dp[1] = 1
for i from 2 to n:
dp[i] = dp[i-1] + dp[i-2]
return dp[n]
```

Complexity

O(n)

11.5 Knapsack Problem

One of the most famous problems in computing.

Problem

Given:

- A backpack with **W capacity**
- Items with **weight** and **value**

Choose items to **maximize value** without exceeding the weight limit.

11.5.1 Solution with DP

We created a table:

Code

```
dp[i][w] = best value using the first i items with capacity w
```

Recurrence

Code

```
if weight[i] > w:
dp[i][w] = dp[i-1][w]
if not:
dp[i][w] = max(
dp[i-1][w], // does not pick up the item
value[i] + dp[i-1][w - weight[i]] // gets the item
)
```

Complexity

$O(n \times W)$

11.6 Longest Common Subsequence (LCS)

Given two strings, find the longest common subsequence.

Example:

Code

```
A = "ABCBDAB"
B = "BDCABA"

LCS = "BCBA" (size 4)
```

Classical And Quantum Computing

Data Structure e Algorithms

11.6.1 Recurrence

Code

```
if A[i] == B[j]:
dp[i][j] = 1 + dp[i-1][j-1]
if not:
dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

Complexity

$O(n \times m)$

11.7 Other Classic Problems of DP

- **Coin Change**
- **Edit Distance (Levenshtein)**
- **Subset Sum**
- **Matrix Chain Multiplication**
- **Rod Cutting**
- **Longest Increasing Subsequence (LIS)**
- **Shortest path in matrix**

DP appears in:

- AI
- Bioinformatics
- Compilers
- Optimization
- Natural language processing

11.8 How to Identify a PD Problem

Ask:

1. The problem can be divided into smaller subproblems.
2. The sub-problems are repeated.
3. The optimal solution depends on smaller solutions.
4. The recursive solution is slow.

5. There is an overlap of sub-problems.

If the answer is **yes** , DP is a good option.

11.9 Comparison: Memorization vs. Tabulation

Criterion	Memorization	Tab
Approach	Top -down	Bottom -up
Structure	Recursion + caching	Iterative table
Execution	Solve only what's necessary.	Solves everything.
Ease	More intuitive	Faster
Memory usage	It could be bigger.	Controlled

11.10 Chapter Conclusion

In this chapter, you learned:

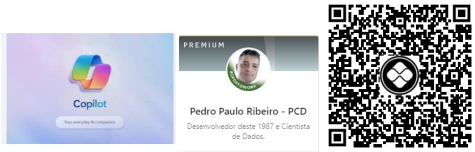
- What is Dynamic Programming?
- Memorization and tabulation
- Classic problems: Fibonacci, Knapsack, LCS
- How to identify PD problems
- How to build efficient solutions

Now you are ready for **Chapter 12: Greedy Algorithms** , which offers a different approach to optimization problems.

Exercises from Chapter 11

1. Conceptual

1. Explain the difference between memorization and tabulation.
2. What are overlapping subproblems?
3. Why is the recursive Fibonacci solution slow?



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Practical

4. Solve Fibonacci(10) using tabulation.
5. Construct the Knapsack table for:
    - Weights: [2, 3, 4]
    - Values: [4, 5, 6]
    - Capacity: 5
6. Calculate the UCL of:
    - A = "ABCBDAB"
    - B = "BDCABA"

3. Challenge

7. Explain how to transform a recursive solution into a DP solution.
8. Identify whether the problems below can be solved using DP and justify your answer:
    - Shortest path in matrix
    - List sorting
    - Subset Sum
    - Binary search

CHAPTER 12 — Greedy Algorithms

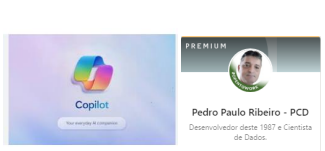
Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding what a greedy algorithm is.
- Identify when the greedy approach works.
- Distinguish between greedy and dynamic programming.
- Solving classic problems such as Dijkstra's, Huffman's, and Interval Scheduling.
- Evaluate the correctness and limitations of this technique.

12.1 What is a Greedy Algorithm?

A greedy algorithm **always makes the best local decision** , hoping that this will lead to the **best global solution** .



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Central idea

Choose the best course of action now, without looking to the future.

It's like someone who tries to save money by always choosing the lowest immediate price — sometimes it works, sometimes it doesn't.

12.2 When to use Greedy Algorithms?

The greedy approach works when the problem has:

1. Property of greedy choice

The best overall solution can be built by always choosing the best local option.

2. Optimal Substructure

The optimal solution contains optimal solutions for subproblems.

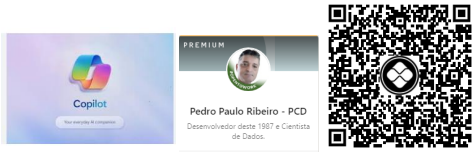
If these two properties do not exist, the greedy algorithm may fail — and then we use **dynamic programming** .

12.3 Comparison: Greedy vs. Dynamic Programming

Criterion	Greedy	Dynamic Programming
Strategy	Choose a location	Explore all the options
Complexity	Generally O(n log n)	Generally $O(n^2)$
Excellent guarantee	Not always	Always (when applicable)
Memory	Low	High
Typical use	Rapid optimization	Complex problems

12.4 Classic Problems Solved with Greedy

Now let's explore the most important greedy algorithms in computing.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

12.4.1 Interval Scheduling

Problem

Given a set of intervals (start, end), choose the **maximum number of non-overlapping intervals** .

Greedy strategy

Always choose the break that **ends earliest** .

Example

Intervals:

Code

```
(1, 4), (3, 5), (0, 6), (5, 7), (3, 9), (5, 9), (6, 10)
```

Sorting by end date:

Code

```
(1, 4), (5, 7), (6, 10)
```

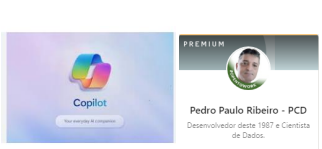
Complexity

O(n log n)

12.4.2 Fractional Knapsack

Problem

Items have weight and value. You can take **fractions** of the items.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Greedy strategy

Choose items with **the highest value per weight** .

Example

Items:

Weight	Value	Value/Weight
10	60	6
20	100	5
30	120	4

Choice:

1. Take item 1
2. Take item 2
3. Take part of item 3

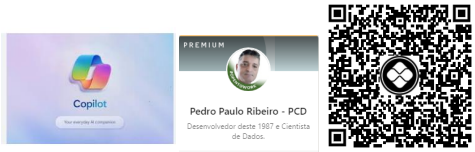
Complexity

O(n log n)

12.4.3 Huffman Coding (Data Compression)

Used in:

- ZIP
- JPEG
- MP3
- PNG



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Idea

Create a binary tree where:

- Most frequent characters → shorter codes
- Rare characters → longer codes

Greedy strategy

Always combine the **two smallest weights** .

Complexity

O(n log n)

12.4.4 Dijkstra (Shortest Path in Weighted Graphs)

Problem

Find the shortest path from one vertex to all other vertices in a graph with positive weights.

Greedy strategy

Always expand the **vertex with the smallest current distance** .

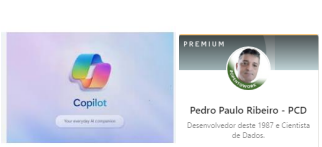
Complexity

O((V + E) log V)

12.4.5 Prim (Minimum Spanning Tree)

Problem

Find the minimum spanning tree (MST).



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Greedy strategy

Always choose the **cheapest edge** that connects to a new vertex.

Complexity

O((V + E) log V)

12.4.6 Kruskal (Minimum Spanning Tree)

Greedy strategy

Sort edges by weight and always add the smallest one that doesn't form a cycle.

Complexity

O(E log E)

12.5 When Does Greedy Fail?

Not every problem can be solved with greedy algorithms.

Example: Knapsack 0/1

Items:

Weight	Value
10	60
20	100
30	120

Capacity = 50

Classical And Quantum Computing

Data Structure e Algorithms

Greedy would choose:

- Item 1 (60)
- Item 2 (100) Total = 160

But the optimal solution is:

- Item 2 (100)
- Item 3 (120) Total = 220

Greedy fails because it cannot pick up fractions.

12.6 Proof of Correcting Greedy Algorithms

To prove that a greedy algorithm works, it is necessary to:

1. To show that the greedy choice is safe.

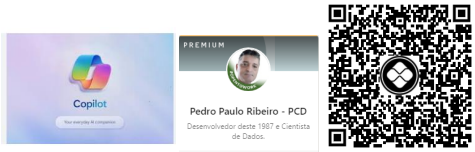
The best global solution includes local choice.

2. Show optimal substructure

The remaining problem is also great.

3. Show that the algorithm terminates.

And it produces a valid solution.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

12.7 Chapter Conclusion

In this chapter, you learned:

- What are greedy algorithms?
- When they work
- Differences between greedy and DP
- Classic problems: Dijkstra, Huffman, Prim, Kruskal
- Limitations of the greedy approach
- How to prove correctness

Now you are ready for **Chapter 13: Graph Algorithms** , where we will see advanced applications of BFS, DFS, Dijkstra, Floyd -Warshall, BellmanFord, and much more.

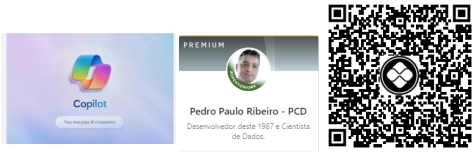
📝 Exercises from Chapter 12

1. Conceptual

1. What is the property of greedy choice?
2. Explain the difference between greedy programming and dynamic programming.
3. Name three problems that can be solved with greedy algorithms.

2. Practical

4. Solve the Interval Scheduling problem for the intervals: `(0, 3), (2, 5), (4, 7), (1, 8), (5, 9)`
5. Solve the Knapsack fractional problem using:
    - Weights: [10, 40, 20]
    - Values: [60, 40, 100]
    - Capacity: 50
6. Construct the Huffman tree for the frequencies: `{A:5, B:9, C:12, D:13, E:16, F:45}`



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

3. Challenge

7. Explain why the greedy algorithm fails on Knapsack 0/1, but works on fractional Knapsack.

CHAPTER 13 — Graph Algorithms

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding the main algorithms applied to graphs.
- Differentiate between search algorithms, shortest paths, and minimum spanning trees.
- Apply BFS, DFS, Dijkstra, Bellman -Ford, FloydWarshall, Prim and Kruskal.
- Knowing when to use each algorithm.
- Analyze complexity and limitations.

13.1 Quick Review: What is a Graph?

A graph is composed of:

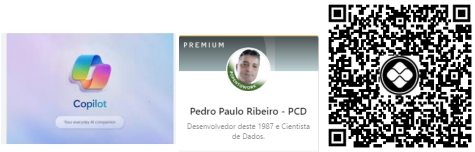
- **Vertices (nodes)**
- **Edges (connections)**
- Optionally **weights** and **steering**

Representations:

- **Adjacency list** → efficient
- **Adjacency matrix** → simple, but heavyweight

13.2 Graph Search Algorithms

These algorithms explore the graph, visiting vertices and edges.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

13.2.1 BFS — Breadth -First Search

Explore the graph layer **by layer** .

Uses

- Shortest path in **unweighted graphs**
- Recommendation systems
- Signal propagation
- Check connectivity

Strategy

- Use **queue**
- Visit your closest neighbors first.

Complexity

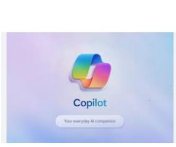
O(V + E)

13.2.2 DFS — Depth -First Search

Explore the graph **as deeply as possible** .

Uses

- Detect cycles
- Find connected components
- Topological ordering
- Solving mazes



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Strategy

- Uses **a stack** or **recursion.**

Complexity

O(V + E)

13.3 Shortest Path Algorithms

Now we move on to algorithms that find **the shortest paths** between vertices.

13.3.1 Dijkstra — Shortest Path with Positive Weights

Requirements

- **Non-negative** weights

Idea

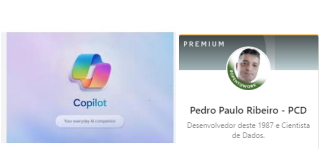
1. It starts at the initial vertex.
2. It maintains a table of distances.
3. It always expands the vertex with **the shortest current distance.**
4. Update your neighbors

Structure used

- **Minimum -heap / Priority Queue**

Complexity

O((V + E) log V)



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Uses

- GPS
- Computer networks
- Games (pathfinding)

13.3.2 Bellman -Ford — Shortest Path with Negative Weights

Requirements

- Allows negative weights
- Detects negative cycles

Idea

- Relax all edges **V − 1 times**
- If relaxation is still possible → negative cycle

Complexity

O(V × E)

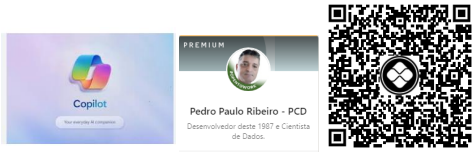
Uses

- Economics (arbitrage)
- Networks with negative costs
- Inconsistency detection

13.3.3 Floyd -Warshall — Shortest Path Between All Pairs

Requirements

- It works with negative weights (without negative cycles).



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Idea

- Three-dimensional DP
- Consider each vertex as an intermediate point.

Complexity

$O(V^3)$

Uses

- Analysis of small networks
- Recommendation systems
- Route pre -processing-

13.4 Minimum Generating Trees (MST)

A **minimum spanning tree** connects all vertices with the **lowest total cost** .

Applications:

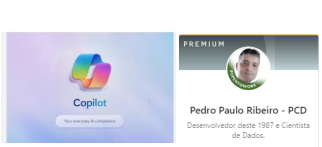
- Electrical networks
- Computer networks
- Infrastructure
- Clustering

There are two main algorithms:

13.4.1 Prim

Idea

- It starts at a vertex.
- It always adds the **cheapest edge** that expands the tree.



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Structure used

- Priority Queue

Complexity

O((V + E) log V)

13.4.2 Kruskal

Idea

1. Sort all edges by weight
2. Add the smallest number that **does not form a cycle.**
3. Uses **Union -Find** to detect cycles

Complexity

O(E log E)

13.5 Topological Sorting

Applicable only to **DAGs (directed acyclic graphs)** .

Idea

- It orders the vertices so that all edges go from "before" to "after".

Uses

- Compilers (dependency order)
- Build systems
- Task planning


Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Algorithms

- DFS
- Kahn (BFS + degree count)

13.6 Cycle Detection

In undirected graphs

- DFS with parent verification

In directed graphs

- DFS with state tagging
- If you find a vertex "in processing" → loop

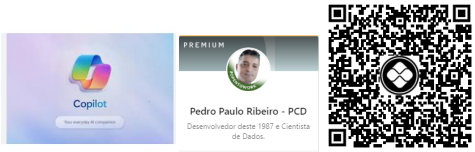
13.7 Connected Components

In undirected graphs

- BFS or DFS

In directed graphs

- Tightly connected components (SCC)
- Algorithms:
    - Kosaraju
    - Tarjan



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

13.8 General Comparison of Algorithms

Problem	Algorithm	Complexity	Observations
Shortest path (positive weights)	Dijkstra	O((V+E) log V)	Very fast
Shortest path (negative weights)	Bellman -Ford	O(VE)	Detects negative cycles
All pairs	Floyd -Warshall	$O(V^3)$	Simple and powerful
MST	First	O((V+E) log V)	Use heap
MST	Kruskal	O(E log E)	USA Union -Find
Search	BFS	O(V+E)	Shortest unweighted path
Search	DFS	O(V+E)	Detects cycles

13.9 Chapter Conclusion

In this chapter, you learned:

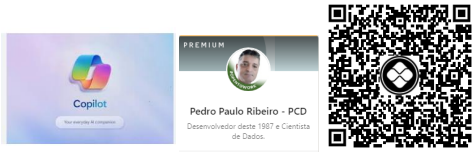
- Deep BFS and DFS
- Dijkstra, Bellman -Ford and FloydWarshall
- Prim and Kruskal for MST
- Topological ordering
- Cycle detection
- Connected components

Now you are ready for **Chapter 14: Advanced Structures** , where we will look at Tries, Segment Trees, Fenwick Trees, Suffix Arrays, and much more.

📝 Exercises from Chapter 13

1. Conceptual

1. Explain the difference between Dijkstra and Bellman -Ford.
2. Why -is Floyd Warshall $O(V^3)$?
3. What is the difference between Prim and Kruskal?



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

2. Practical

4. Execute Dijkstra on the graph:

Code

```
A —1— B —2— C
|  \
4   3
|    \
D —1— E —1— F
```

5. Execute Bellman -Ford on the chart:

Code

```
A → B (2)
A → C (4)
B → C (-1)
C → D (2)
```

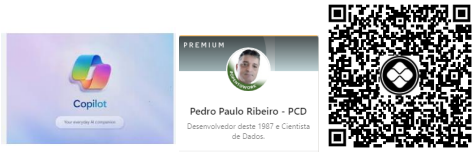
6. Build MST using Kruskal:

Code

```
A—1—B
A—3—C
B—2—C
B—4—D
C—5—D
```

3. Challenge

7. Explain why Dijkstra fails with negative weights, but Bellman -Ford does not.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 14 — Advanced Structures

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding advanced structures such as Tries, Segment Trees, Fenwick Trees, Suffix Arrays, and Bloom Filters.
- Knowing when to use each structure.
- Understanding advantages, limitations, and complexities.
- Applying these frameworks to real-world, high-performance problems.

14.1 Introduction to Advanced Structures

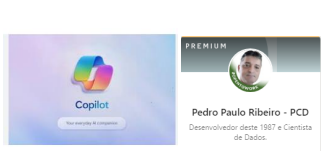
The structures seen so far (lists, trees, graphs, hash tables) solve most problems. But some applications require **extreme speed** , **efficient compression** , **fast text searches** , or **range operations** .

For this, we use advanced structures such as:

- **Tries**
- **Segment Trees**
- **Fenwick Trees (Binary Indexed Trees)**
- **Suffix Trees and Suffix Arrays**
- **Bloom Filters**

14.2 Tries (Prefix Trees)

A **Trie** is a tree specialized for efficiently storing strings.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Visual example

Storing: "car", "house", "corner"

Code

```
w
├── a
│   ├── r
│   ├── s ─ a
│   └── n ─ t ─ o
```

Uses

- Autocomplete
- Spell checkers
- Search by prefix
- Dictionaries
- Compression

Complexity

Operation Complexity

Insert O(m)

Search O(m)

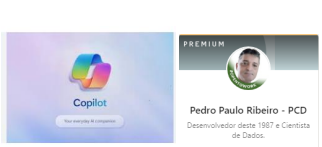
Remove O(m)

m = string length

14.3 Segment Trees

A Segment Tree allows you to:

- Quick consultations at intervals
- Quick updates




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Example of a problem

- Sum of an interval
- Minimum/maximum within a range
- Update an element

Structure

It is a binary tree where each node represents a range.

Example for array `[1, 3, 5, 7]` :

Code

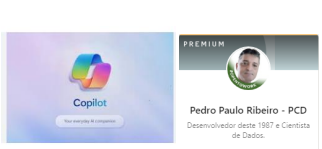
```
[1..4]
/ \
[1..2] [3..4]
/ \ / \
[1] [3] [5] [7]
```

Complexity

Operation	Complexity
Consultation	O(log n)
Update	O(log n)
Construction	O(n)

Uses

- Games
- Financial systems
- Data analysis
- Competitive algorithms



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

14.4 Fenwick Tree (Binary Indexed Tree)

A Fenwick Tree is a simpler alternative to a Segment Tree.

Uses

- Prefixed sum
- Quick updates
- Real-time statistics

Advantages

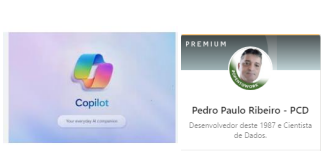
- Much simpler than Segment Tree
- Uses less memory.

Complexity

Operation	Complexity
Prefixed sum	O(log n)
Update	O(log n)

14.5 Suffix Trees and Suffix Arrays

These structures are used for **quick text searches** .



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

14.5.1 Suffix Tree

Stores all the suffixes of a string.

Example for "banana":

Code

```
banana
pineapple
nana
dwarf
in
the
```

Uses

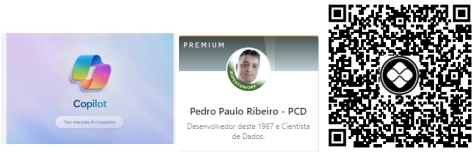
- Pattern search in O(m)
- Compression
- Bioinformatics (DNA)
- Search engines

Disadvantages

- Very complex
- It uses a lot of memory.

14.5.2 Suffix Array

A lighter alternative.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Example for "banana"

Ordered suffixes:

Code

```
0: banana
1: pineapple
2: nana
3: Ana
4: in
5: a
```

Stores only the indexes:

Code

```
[5, 3, 1, 0, 4, 2]
```

Uses

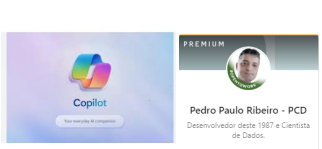
- Binary text search
- Compression (BWT)
- Bioinformatics algorithms

Complexity

- Construction: O(n log n)
- Search: O(m log n)

14.6 Bloom Filters

A probabilistic structure used to check if an element **can be** in a set.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Features

- Extremely fast
- It uses very little memory.
- It can give **false positives.**
- It never gives **false negatives.**

Operation

1. Uses multiple hashes
2. It marks positions in a bit array.
3. To check, verify if all bits are selected.

Uses

- Databases (Redis, Cassandra)
- Browsers (detection of malicious URLs)
- Distributed systems
- Caches

14.7 General Comparison of Advanced Structures

Structure	Best use	Complexity
Trie	Search by prefix	O(m)
Segment Tree	Dynamic ranges	O(log n)
Fenwick Tree	Preset sums	O(log n)
Suffix Array	Text search	O(m log n)
Bloom Filter	Belonging test	OK)

14.8 Practical Examples

14.8.1 Autocomplete with Trie

- Insert words
- Search for prefixes
- List suggestions

Classical And Quantum Computing

Data Structure e Algorithms

14.8.2 Interval Summation with Segment Tree

- Sum of `[l, r]`
- Update `arr[i]`

14.8.3 Check if an item is cached using Bloom Filter

- If "it's not," then it's absolutely certain.
- If it says "yes," it could be a false positive.

14.9 Chapter Conclusion

In this chapter, you learned:

- Tries for text searches
- Segment Trees and Fenwick Trees for intervals
- Suffix Arrays for fast string searches
- Bloom Filters for probabilistic tests
- Comparisons and real-world applications

Now you are ready for **Chapter 15: Algorithm Design** , where we will see how to create algorithms from scratch using professional techniques.

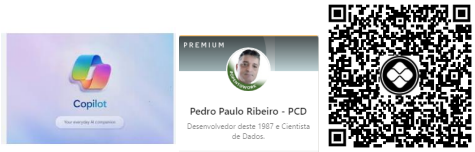
📝 Exercises from Chapter 14

1. Conceptual

1. What is a Trie and what is it used for?
2. What is the difference between a Segment Tree and a Fenwick Tree?
3. Why can Bloom Filters give false positives?

2. Practical

4. Create a trie using the words: `house, car, corner, bed.`
5. Construct a Segment Tree for the array: `[2, 4, 5, 7, 8]`
6. Generate the Suffix Array of the string `"banana"`



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

3. Challenge

7. Explain why Suffix Trees are fast but rarely used in practice.

If you want, I can:

- Create **examples in Python, Java, or C++.**
- Add **visual diagrams**
- Create the **answers to the exercises.**
- Proceed to **Chapter 15: Algorithm Design**

CHAPTER 15 — Algorithm Design

Chapter Objectives

By the end of this chapter, you will be able to:

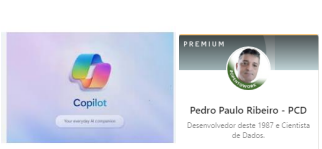
- Understanding the algorithm creation process.
- Apply classic design techniques: divide and conquer, dynamic programming, greedy design, backtracking, branch and bound, and heuristics.
- Transforming real-world problems into efficient algorithms.
- Evaluate the trade-offs between time, memory, and simplicity.
- Create elegant and scalable solutions.

15.1 What is Algorithm Design?

Algorithm design is the art of:

1. **To understand a problem deeply.**
2. **Model it mathematically.**
3. **Choosing the right technique**
4. **Create an efficient and correct solution.**

It's like designing a bridge: it's not enough for it to function — it needs to be safe, efficient, and elegant.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

15.2 Algorithm Design Steps

The process can be divided into 6 steps:

1. Understanding the Problem

Essential questions:

- What is the entrance?
- What is the solution?
- What are the restrictions?
- Are there extreme cases?
- Is the problem deterministic?

2. Modeling

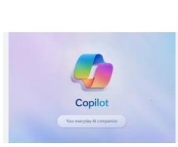
Transforming the problem into a formal structure:

- Graph?
- Tree?
- List?
- Intermissions?
- Mathematical function?

3. Choosing the Technique

The main techniques are:

- Divide and conquer
- Dynamic programming
- Greedy algorithms
- Backtracking
- Branch & Bound
- Heuristics
- Randomization




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

4. Solution Construction

Create the algorithm step by step.

5. Complexity Analysis

To assess:

- Time
- Space
- Best case
- Worst case
- Average case

6. Optimization and Refinement

To improve:

- Data structures
- Reduce recursive calls
- Avoid recomputations
- Using hybrid techniques

15.3 Algorithm Design Techniques

Now let's explore each technique in depth.


Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

15.3.1 Divide and Conquer

Divide the problem into smaller subproblems, solve each one, and combine them.

Classic examples

- Merge Sort
- Quick Sort
- Binary search
- FFT (Fast Fourier Transform)

Structure

Code

```
to divide
solve subproblems
combine results
```

When to use it?

- Recursive problems
- Hierarchical structures
- Natural divisions

15.3.2 Dynamic Programming

Used when:

- Subproblems are repeated.
- There is an optimal substructure.

Examples

- Fibonacci
- Knapsack
- LCS
- Shortest path in matrix

Classical And Quantum Computing

Data Structure e Algorithms

Strategies

- Memorization (top-down)
- Tabulation (bottom-up)

15.3.3 Greedy Algorithms

They always choose the best local decision.

Examples

- Dijkstra
- Huffman
- Interval Scheduling
- Prim and Kruskal

When to use it?

- When local choice leads to global solution.
- When there is proof of correctness

15.3.4 Backtracking

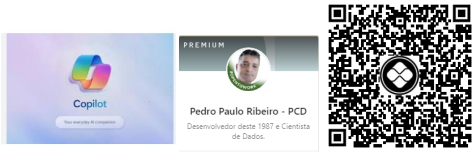
Explore all possibilities, backtracking when necessary.

Examples

- Sudoku
- N-Queens
- Permutations
- Mazes

When to use it?

- Exhaustive search problems
- Complex constraints



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

15.3.5 Branch & Bound

Optimized version of backtracking.

Idea

- Explore the search space
- Cut off branches that cannot lead to the optimal solution.

Examples

- Knapsack 0/1
- Traveling Salesman Problem (TSP)

15.3.6 Heuristics

Approximate solutions to difficult problems.

Examples

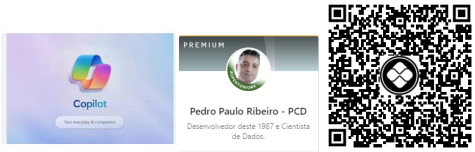
- Genetic algorithms
- Simulated annealing
- Hill climbing

Uses

- AI
- Industrial optimization
- Robotics

15.3.7 Randomization

Algorithms that use random choices.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Examples

- Quick Random Sort
- Monte Carlo Algorithm
- Las Vegas Algorithm

Advantages

- Simple
- Fast
- They avoid worst-case deterministic scenarios.

15.4 Algorithmic Thinking Strategies

To create better algorithms, develop:

1. Recursive thinking

View problems as smaller versions of yourself.

2. Graph Thinking

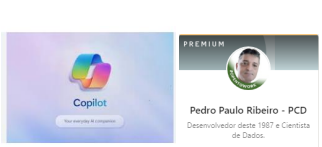
Many problems are graphs in disguise.

3. Intermittent thinking

Problems with time, schedules, windows, ranges.

4. Thought in states

The basis of dynamic programming.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

5. Thinking in terms of constraints

Backtracking basis.

15.5 Problem Patterns

There are recurring patterns:

- Two hands
- Sliding window
- Binary search in response
- Prefix sums
- Hashing
- Greedy by ordering
- DP by state

15.6 Practical Examples of Algorithm Design

Let's solve three real-world problems.

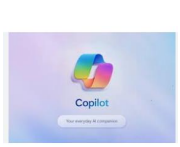
15.6.1 Problem 1 — Find the first repeated element

Modeling

- Use hash set

Solution

O(n)




Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

15.6.2 Problem 2 — Shortest path on a map

Modeling

- Weighted graph

Solution

- Dijkstra

15.6.3 Problem 3 — Best time for meetings

Modeling

- Intervals

Solution

- Greedy for early termination

15.7 Trade-offs in Algorithm Design

Speed vs. Memory

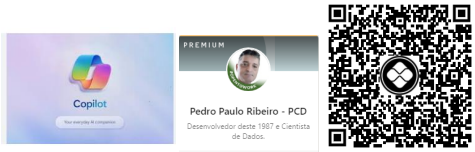
- DP uses more memory.
- Greedy uses less

Simplicity vs. Optimization

- Simple solutions are easier to maintain.
- Optimized solutions are faster.

Precision vs. Time

- Heuristics are fast, but approximate.
- DP is accurate, but slow.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

15.8 Chapter Conclusion

In this chapter, you learned:

- The complete algorithm design process
- Classic and modern techniques
- How to choose the right approach
- How to think like an algorithm designer.
- Real-world examples and trade-offs

Now you are ready for **Chapter 16: Advanced Algorithm Analysis** , where we will explore more in-depth mathematical techniques.

📝 Exercises from Chapter 15

1. Conceptual

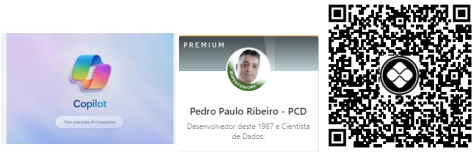
1. Explain the difference between divide and conquer and dynamic programming.
2. What characterizes a greedy algorithm?
3. When should you use backtracking?

2. Practical

4. Model the Sudoku problem as a backtracking problem.
5. Create a greedy algorithm to select non-overlapping activities.
6. Transform the recursive Fibonacci solution into a memo-based solution.

3. Challenge

7. Given a complex problem, explain how to decide between DP, greedy, or backtracking.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

CHAPTER 16 — Advanced Algorithm Analysis

Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding advanced algorithm analysis techniques.
- Solving recurrences using formal methods.
- Compare algorithms using amortized analysis.
- Understanding probability analysis.
- Apply lower bounds and complexity arguments.
- Evaluate algorithms in real-world and theoretical scenarios.

16.1 Why Advanced Analysis?

Big O notation -is excellent for understanding the growth of an algorithm, but:

- It's not always enough.
- Some algorithms exhibit variable behavior.
- Dynamic structures require more in-depth analysis.
- Complex recurrences appear in recursive algorithms.
- Probabilistic algorithms require additional mathematics.

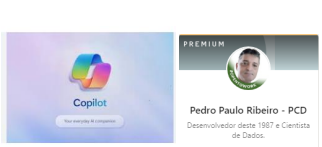
Therefore, we need more powerful tools.

16.2 Resolution of Recurrences

Recurrences appear in recursive algorithms such as:

- Merge Sort
- Quick Sort
- Divide and conquer
- Dynamic programming

Let's look at the most important methods.



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

16.2.1 Substitution Method

It consists of:

1. **Guess** the solution
2. **Prove by induction**

Example

Code

```
T(n) = T(n/2) + 1
```

We guessed it:

Code

```
T(n) = O(log n)
```

We proved it by induction.

16.2.2 Recursion Tree Method

You draw the call tree and sum the cost of each level.

Example: Merge Sort

Code

```
T(n) = 2T(n/2) + n
```

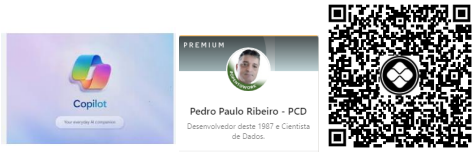
Tree:

- Level 0: n
- Level 1: n
- Level 2: n
- Height: log n

Total:

Code

```
n log n
```



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

16.2.3 Master Theorem

For recurrences of the form:

Code

```
T(n) = aT(n/b) + f(n)
```

There are three cases:

Case 1 — f(n) less than n^(log_b a)

Code

```
T(n) = Θ(n^(log_b a))
```

Case 2 — f(n) equals an^(log_b a)

Code

```
T(n) = Θ(n^(log_b a) log n)
```

Case 3 — f(n) greater than n^(log_b a)

Code

```
T(n) = Θ(f(n))
```

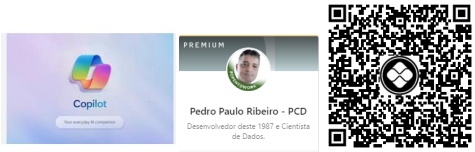
16.3 Amortized Analysis

Used when:

- An operation is occasionally expensive.
- But cheap most of the time.

Classic examples:

- **Dynamic array** (occasional reallocation)
- **Row with two stacks**
- **Union-Find with path compression**
- **Hash table with resizing**



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

16.3.1 Amortized Analysis Methods

1. Summation method

Add up the total cost and divide by the number of transactions.

2. Accounting Method

It assigns credits to cheap operations to pay for the expensive ones.

3. Potential Method

Define a potential function Φ that measures "stored energy".

16.4 Probabilistic Analysis

Probabilistic algorithms use randomness.

Examples:

- Quick Random Sort
- Monte Carlo Algorithms
- Las Vegas Algorithms
- Universal Hashing

16.4.1 Expected Value

We use:

- Probability
- Mathematical expectation
- Variance

Example: Random Quick Sort

Expected time:

Code

```
O(n log n)
```

Even if the worst case is $O(n^2)$.

16.5 Lower Limits

They show that **no algorithm** can be faster than a certain limit.

Examples

- Comparison sorting → $\Omega(n \log n)$
- Search in unsorted array → $\Omega(n)$
- Binary tree search → $\Omega(\log n)$

These limits are fundamental to understanding what is possible.

16.6 Complexity Classes

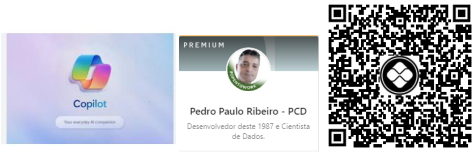
An overview of the theory of computation.

16.6.1 Class P

Problems solvable in polynomial time.

16.6.2 NP Class

Problems whose solution can be verified quickly.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/

WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

16.6.3 NP -Complete

The most difficult problems in NP.

Examples:

- Traveling salesman (TSP)
- Boolean Satisfaction (SAT)
- Backpack 0/1
- Graph coloring

16.6.4 NP -Hard

As difficult as NP -Complete, but not always verifiable.

16.7 Reductions

A technique for proving that one problem is just as difficult as another.

Code

```
A → B
```

If A is difficult, then B is also difficult.

16.8 Algorithm Analysis in Real-World Scenarios

Beyond theory, algorithms need to work well in practice.

Real factors:

- CPU Cache
- Place of memory
- Parallelism
- Processor architecture
- Actual data size
- Data distribution

Therefore, sometimes:

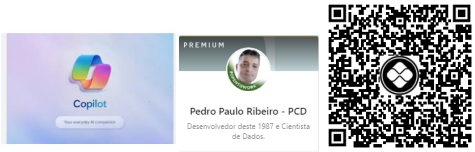
- Quick Sort is faster than Merge Sort.
- Hash tables outperform balanced trees.
- Approximate algorithms are better than exact ones.

16.9 Chapter Conclusion

In this chapter, you learned:

- Formal methods for resolving recurrences
- Amortized analysis
- Probabilistic analysis
- Lower limits
- Complexity classes
- Reductions
- Practical performance factors

Now you are ready for **Chapter 17: Structures and Algorithms in Real Systems** , where we will see how all of this applies to databases, operating systems, networks, and modern applications.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

📝 Exercises from Chapter 16

1. Conceptual

1. Explain the Master Theorem in your own words.
2. What is amortized analysis?
3. What is the difference between P, NP, and -Complete NP?

2. Practical

4. Solve the recurrence: `T(n) = 2T(n/2) + n`
5. Solve the recurrence: `T(n) = T(n/2) + 1`
6. Show that the amortized insertion cost in a dynamic array is O(1).

3. Challenge

7. Explain why no comparison sorting algorithm can be faster than O(n log n).

📓 CHAPTER 17 — Structures and Algorithms in Real Systems

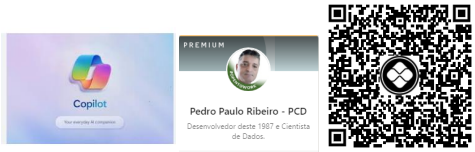
🎯 Chapter Objectives

By the end of this chapter, you will be able to:

- Understanding how data structures are used in real-world systems.
- Relating theoretical concepts to practical implementations.
- Understanding how databases, operating systems, networks, and browsers use algorithms.
- Recognizing design patterns and optimizations used in the industry.
- To see how algorithms influence performance, scalability, and reliability.

17.1 Why study real systems?

Up to this point, you've learned about structures and algorithms in theory. Now it's time to see **where all of this appears in practice** .



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Real-world systems require:

- Be quick
- Being scalable
- Being trustworthy
- Dealing with millions of users
- Processing large volumes of data

And for that, they make intensive use of **data structures and algorithms** .

17.2 Databases

Databases are one of the biggest consumers of advanced algorithms.

17.2.1 B and B+ Trees (Indices)

When you do:

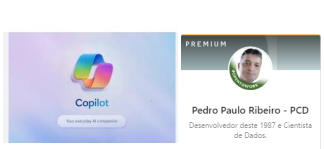
Code

```
SELECT * FROM users WHERE id = 123;
```

The database does not traverse all records. It uses **B+ trees** , an optimized version of balanced trees.

Why B+ Trees?

- Very low height
- Efficient disk access
- We big (disc pages)
- Search in O(log n) even with billions of records.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

Structures used

Function	Structure
Indices	B+ Trees
Hash indices	Hash tables
Ordering	External Merge Sort
Junctions	Hash Join, Sort-Merge Join

17.2.2 Hashing in databases

Used in:

- Hash Join
- Hash indices
- Table partitioning
- Sharding in distributed databases

17.3 Operating Systems

Operating systems use algorithms all the time.

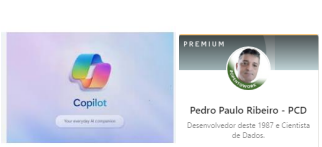
17.3.1 Process scheduling

Classic algorithms:

- Round Robin
- Shortest Job First
- Priority Scheduling
- Multilevel Queue

Structures used

- Queues
- Heaps (for priorities)
- Red Black Trees -(for managing processes)



Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

17.3.2 Memory Management

Pagination

- Replacement algorithms:
  - LRU (Least Recently Used) → uses lists and hash
  - FIFO → uses queues
  - Clock → uses circular hands

Memory allocators

- Buddy System → uses binary trees
- Slab Allocator → uses linked lists

17.3.3 File systems

Structures used:

- Trees (directories)
- Hash tables (directory cache)
- B+ Trees (modern systems like NTFS, APFS, ext4)
- Bitmaps (free blocks)

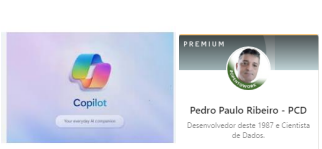
17.4 Networks and the Internet

The internet works thanks to algorithms.

17.4.1 Routing

Algorithms used:

- Dijkstra (OSPF)
- Bellman -Ford (RIP)
- Generating trees (MST)



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

17.4.2 DNS

DNS uses:

- Suffix trees (implicit)
- Caches with hash tables
- Load balancing

17.4.3 TCP/IP

- Congestion control → adaptive algorithms
- Retransmission → timers and exponential backoff
- Sliding windows → queues and buffers

17.5 Browsers and Search Engines

17.5.1 Browsers

Parsing HTML and CSS

- Syntactic trees (DOM)
- Style trees (CSSOM)

Rendering

- Layout trees
- Flow and painting algorithms

Cache

- Hash tables
- LRU


Voluntary Contribution
https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

17.5.2 Search engines

Indexing

- Tries
- Suffix Arrays
- Hash tables
- Graphs (PageRank)

PageRank

- Graph-based algorithm
- It uses linear algebra and successive iterations.

17.6 Games and Graphics Engines

Games are highly optimized systems.

17.6.1 Physics

- Quadtrees
- Octrees
- BSP Trees
- Spatial Hashing

17.6.2 Gaming AI

- Graphs (pathfinding)
- A* (Dijkstra variation)
- State machines
- Behavior trees

Classical And Quantum Computing

Data Structure e Algorithms

17.6.3 Rendering

- Rasterization algorithms
- BVH (Bounding Volume Hierarchy) Trees
- KD -Trees

17.7 Distributed Systems

17.7.1 Consistency and replication

Algorithms:

- Paxos
- Raft
- Gossip Protocols

Structures:

- Logs
- Trees
- Consistent Hashing

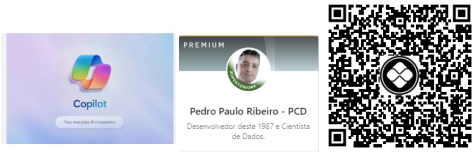
17.7.2 MapReduce

Use:

- Hash tables
- Aggregation trees
- Distributed sorting

17.8 Machine Learning and AI

Even in AI, classic algorithms still appear.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

17.8.1 Decision trees

- Binary trees
- Greedy heuristics

17.8.2 Neural networks

- Directed acyclic graphs (DAGs)
- Optimization algorithms (gradient)

17.8.3 K -Means

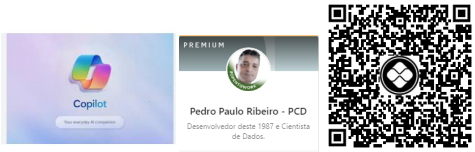
- Distances → linear algebra
- Iterations → heuristics

17.9 Chapter Conclusion

In this chapter, you learned:

- How algorithms and structures appear in real-world systems.
- How databases use B+ trees and hashing
- How operating systems use queues, heaps, and trees.
- How networks use Dijkstra and Bellman -Ford
- How browsers use syntax trees
- How games use spatial structures
- How distributed systems use consensus algorithms

Now you are ready for **Chapter 18: Final Project** , where we will integrate everything you have learned into a large, hands-on project.



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

📝 Exercises from Chapter 17

1. Conceptual

1. Why do databases use B+ Trees instead of regular BSTs?
2. Explain how operating systems use queues and heaps.
3. What is the relationship between Dijkstra and Internet routing?

2. Practical

4. Model a computer's file system using trees.
5. Explain how a browser transforms HTML into a rendered page.
6. Describe how a game uses quadtrees to detect collisions.

3. Challenge

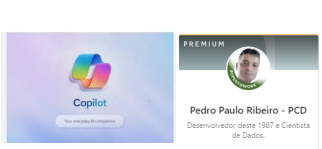
7. Choose a real-world system (database, browser, game, network, etc.) and describe **all** the data structures and algorithms it uses.

📓 CHAPTER 18 — Final Project: Building a Complete System

🎯 Chapter Objectives

By the end of this chapter, you will be able to:

- Integrating data structures and algorithms into a real-world system.
- Designing architecture, modules, and data flows.
- Choosing the right structures for each part of the system.
- Apply complexity analysis to design decisions.
- Create a scalable, efficient, and well-structured project.




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

18.1 The Final Project

Let's build a complete system called:

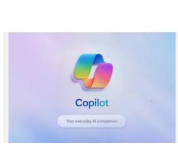
AtlasSearch — A Simplified Search Engine

He will have:

- Document indexing
- Search by keywords
- Autocomplete
- Basic ranking
- Cache
- Statistics
- Simple (conceptual) interface

This project brings together:

- Trees
- Graphs
- Hash tables
- Tries
- Search algorithms
- Sorting algorithms
- Dynamic programming (in optimizations)
- Greedy algorithms
- Advanced structures (Optional Array suffix)




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

18.2 General System Architecture

Main modules:

1. **Collector (Crawler)**
2. **Word Processor**
3. **Inverted Index**
4. **Autocomplete Trie**
5. **Ranking**
6. **Cache**
7. **Query Interface**

General flow:

Code

```
Documents → Processing → Indexing → Query → Ranking → Results
```

18.3 Module 1 — Collector (Crawler)

Responsible for:

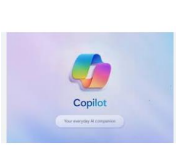
- Read documents
- Extract text
- Normalize content

Structures used:

- **Queues** → for URLs to visit
- **Hash Set** → to avoid visiting the same page
- **Graphs** → for mapping links between pages

Algorithms:

- BFS → to explore pages
- Hashing → to detect duplicates




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

18.4 Module 2 — Text Processing

Responsible for:

- Tokenization
- Stopword removal
- Stemming (word reduction)

Structures used:

- Hash tables → word count
- Lists → tokens
- Trees → stopword dictionaries

18.5 Module 3 — Inverted Index

The heart of the search engine.

What is it?

A map:

Code

```
word → list of documents where it appears
```

Structures used:

- **Hash table** → key = word
- **Ordered lists** → documents
- **Heaps** → quick ranking

Classical And Quantum Computing

Data Structure e Algorithms

Algorithms used:

- Binary search → locate documents
- Merge Sort → sort lists
- Hashing → mapping words

18.6 Module 4 — Autocomplete with Trie

For instant suggestions.

Structure:

A **Trie** where each node represents a character.

Operations:

- Insert word → O(m)
- Search for prefix → O(m)
- List suggestions → DFS or BFS

18.7 Module 5 — Ranking

Simple ranking based on:

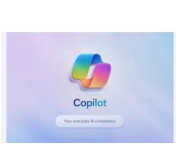
- Word frequency
- Document size
- Popularity (link graph)

Structures used:

- Heaps → top results
- Graphs → Simplified PageRank
- Hash tables → frequencies

Algorithms:

- PageRank (iterative)
- Heap Sort
- Quick Select




Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

18.8 Module 6 — Cache

To speed up repeated searches.

Structures used:

- Hash table
- Doubly linked list

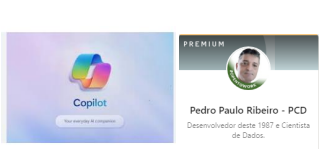
Implementation:

- **LRU Cache** (Least Recently Used)
- Complexity: O(1) for insert, remove and access

18.9 Module 7 — Query Interface

Flow:

1. User types term
2. Trie suggests words
3. Reversed index returns documents
4. Ranking orders
5. Cache stores results.
6. System displays response



Voluntary Contribution

https://www.linkedin.com/in/pedro-paulo-ribeiro-pcd-1451498/
WhatsApp 55 21 999618643

Classical And Quantum Computing

Data Structure e Algorithms

18.10 System Complexity Analysis

Module	Structure	Complexity
Crawler	BFS + Hash	O(V + E)
Processing	Hash	O(n)
inverted index	Hash + lists	O(n log n)
Autocomplete	Trie	O(m)
Ranking	Heap	O(k log n)
Cache	LRU	O(1)

18.11 Advanced Extensions

If you want to evolve the project:

- **Suffix Array for searching substrings**
- **Bloom Filter to speed up queries**
- **Machine Learning for ranking**
- **Graphs for recommendation**
- **Quantum Computing (Grover) to accelerate search**

I can incorporate any of these into your book.

18.12 Chapter Conclusion

In this chapter, you learned:

- How to integrate structures and algorithms into a real-world system.
- How to design efficient modules
- How to choose the right structure for each problem.
- How to analyze the performance of a complete system
- How to build a simplified search engine.