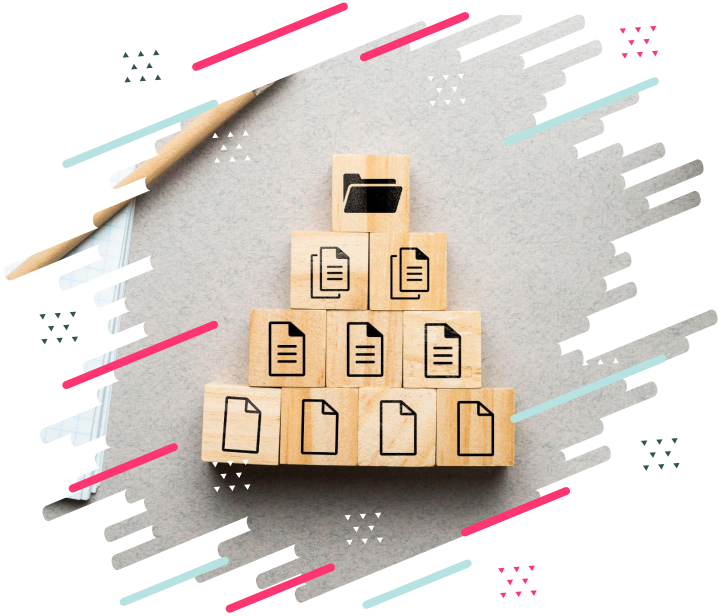
2025

ADVANCED DATA STRUCTURES CHEAT SHEET



Ben Khalfallah Héla

ADVANCED DATA STRUCTURES

- Modern file systems require a robust design to **efficiently** manage local and cloud-based files. The system must:
  - Support **hierarchical** organization for directories and files.
  - Deliver **optimal performance** for operations like search, insert, delete, and update.
  - **Scale** seamlessly to accommodate millions of files while maintaining efficiency.
- The **limitations** of basic tools like arrays and hashmaps **necessitate the use of advanced data structures**.

BINARY SEARCH TREE (BST)

The Binary Search Tree introduces a **hierarchical structure** to organize files, ensuring efficient retrieval and updates.

ALGORITHMIC DESIGN

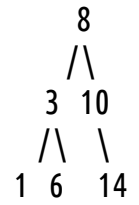
- Directories and files are stored in **nodes**.
- **Smaller** nodes (e.g., files with alphabetically earlier names) are placed in the **left subtree**, and **larger** nodes in the **right subtree**.

COMPLEXITIES

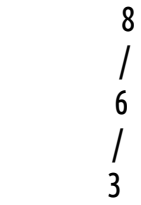
- Search, Insert, Delete (Average): **O(log n)**
- Worst Case (Unbalanced): **O(n)**
- Space Complexity: **O(n)**

USE CASE EXAMPLE

- **Scenario**: A **lightweight** desktop application managing a **small hierarchy** of files locally.
- **Limit**: Inefficient when handling large or sequentially inserted files, as it becomes **unbalanced**.



Balanced



Unbalanced

ADELSON- VELSKY AND LANDIS TREE (AVL)

The AVL Tree **improves** on the BST by **ensuring balance**, thereby **maintaining O(log n)** complexity for all operations regardless of input order.

ALGORITHMIC DESIGN

- Tracks the **balance factor** (difference in heights of left and right subtrees).
- Performs **rotations** (single or double) to maintain balance after insertions or deletions.

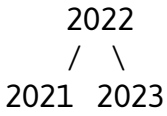
COMPLEXITIES

- Search, Insert, Delete: **O(log n)**.
- Space Complexity: **O(n)**.

USE CASE EXAMPLE

- **Scenario**: Ideal for metadata-heavy file systems **where read operations dominate (frequent searches or reads)**.
- **Limit:** Rotations **add processing overhead** during frequent updates or deletions.
- **Real-world application:** ZFS uses AVL trees (and B-trees in newer versions) for efficient metadata management.

Before Balancing



After Balancing

RED-BLACK TREE

The Red-Black Tree **balances efficiency with dynamic updates**, ensuring **fewer rotations** compared to AVL Trees.

ALGORITHMIC DESIGN

- Every node is either **red or black**.
- The **root is black**.
- If a node is **red**, then both its **children are black**.
- For each node, **all path** from the node to descendant leaves contain the **same number of black nodes**.
- **Black Nodes:** Maintain overall balance.
- **Red Nodes:** Spread out to avoid imbalance.

COMPLEXITIES

- Search, Insert, Delete: **O(log n)**
- Space Complexity: **O(n)**

USE CASE EXAMPLE

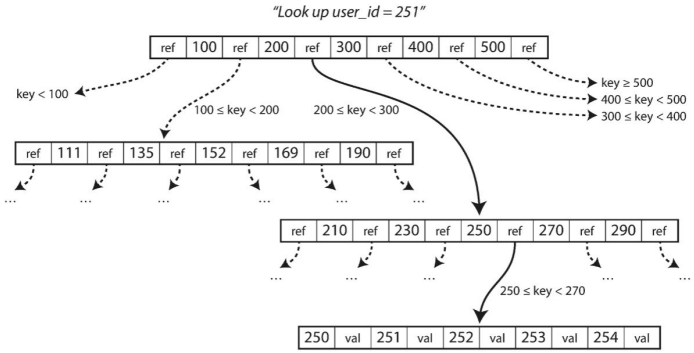
- **Scenario**: Efficient for dynamic file systems with **frequent insertions and deletions**.
- **Limit:** Slightly **less balanced** than AVL, making it **less efficient for pure read** operations.
- **Real-world application:** Completely Fair Scheduler and epoll system call of the Linux kernel use red–black trees.

BALANCED TREE (B-TREE)

The **B-Tree** is a **self-balancing** tree designed for systems managing **large datasets stored on disk**. It **minimizes disk I/O** by allowing **multiple keys per node** and maintaining a **shallow tree height**.

ALGORITHMIC DESIGN

- **Node Structure**: A B-Tree node contains:
  - A list of **keys**, sorted in ascending order.
  - A list of **pointers** (or references) to child nodes.
  - A rule that each node (except the root) must have at least **$\lceil m/2 \rceil - 1$ keys and at most $m - 1$ keys**, **where m is the order of the B-Tree**.
- **Splitting Nodes**: When a node exceeds its key capacity:
  - The **middle key** is promoted to the parent node.
  - The **node splits into two child nodes**, each containing half the keys.



Designing Data-Intensive Applications

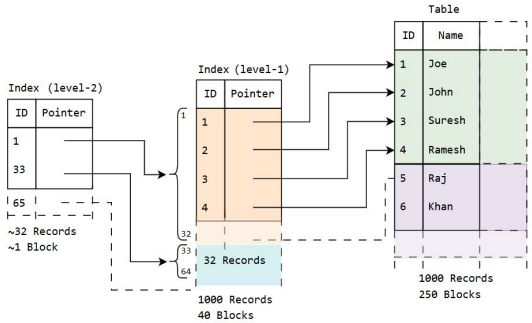
BALANCED TREE (B-TREE)

COMPLEXITIES

- Search, Insert, Delete: **O(log n)** (due to the balanced nature of the tree).
- Space Complexity: **O(n)** (all keys and pointers must be stored).

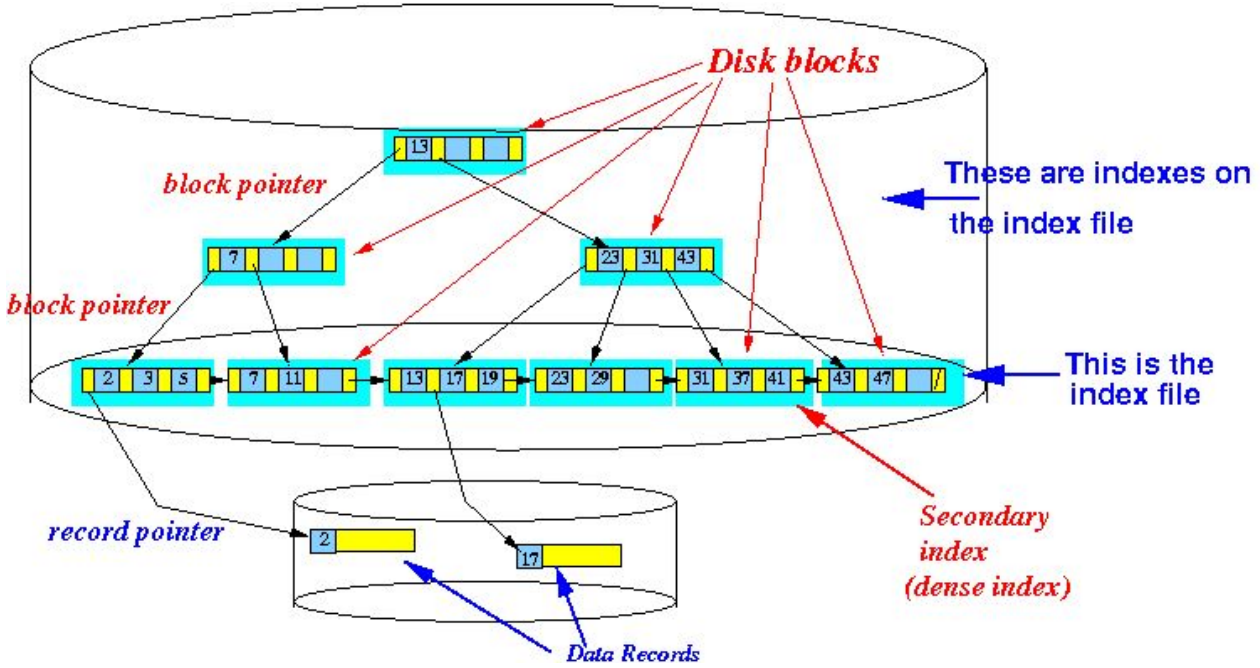
USE CASE EXAMPLE

- **Scenario**: A large-scale file system indexes millions of files stored on disk. The B-Tree minimizes the number of disk reads required to locate a specific file.
- **Limit:** In-memory operations are slower compared to AVL or Red-Black Trees.
- **Real-world application:**
  - In NTFS, B-tree structures optimize folder record management, enabling efficient indexing and search operations in large folders.
  - In APFS, B-Trees are integral for managing filesystem metadata efficiently.



https://www.pyblog.xyz/b-tree

BALANCED TREE (B-TREE)



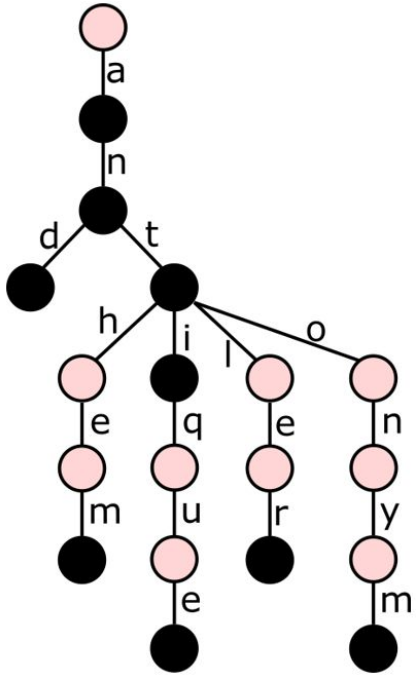
https://www.cs.emory.edu/~cheung/Courses/554/Syllabus/3-index/B-tree=delete1.html

PREFIX TREE (TRIE)

The **Trie** (pronounced "try") is a tree-like data structure that efficiently handles **prefix-based searches**, making it ideal for file systems requiring autocomplete, search-as-you-type, or lookup features.

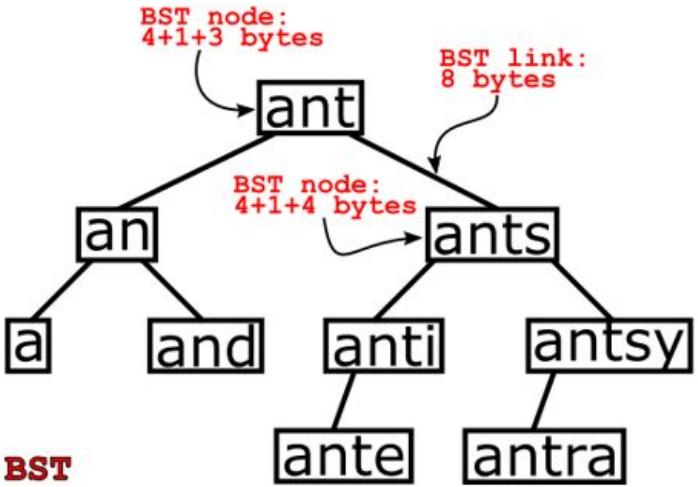
ALGORITHMIC DESIGN

- **Node Structure**: Each node **represents a character, and paths form strings** such as file or directory names.
- A node has:
  - A **character** (part of a string).
  - A **flag** indicating whether it marks the end of a valid string (e.g., a complete file or folder name).
  - References to its **child nodes**.



https://www.manning.com/books/advanced-algorithms-and-data-structures

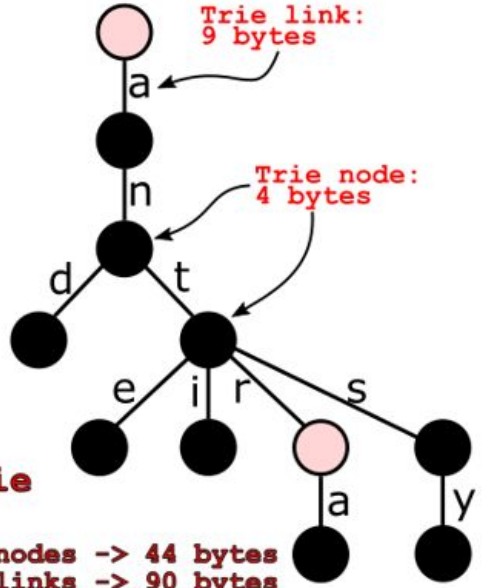
TRIE VS. BST



BST

9 nodes -> 45 bytes
31 characters -> 31 bytes
18 links -> 144 bytes

Total: 220 bytes



Trie

11 nodes -> 44 bytes
10 links -> 90 bytes

Total: 134 bytes

https://www.manning.com/books/advanced-algorithms-and-data-structures

PREFIX TREE (TRIE)

SEARCH

To search for a string:

- Traverse the nodes matching the characters of the string.
- If all characters match and the last node is marked as the end of a string, the search is successful.

PREFIX MATCHING

To find all strings starting with a prefix:

- Traverse nodes matching the prefix.
- Collect all strings reachable from the last prefix node.

COMPLEXITIES

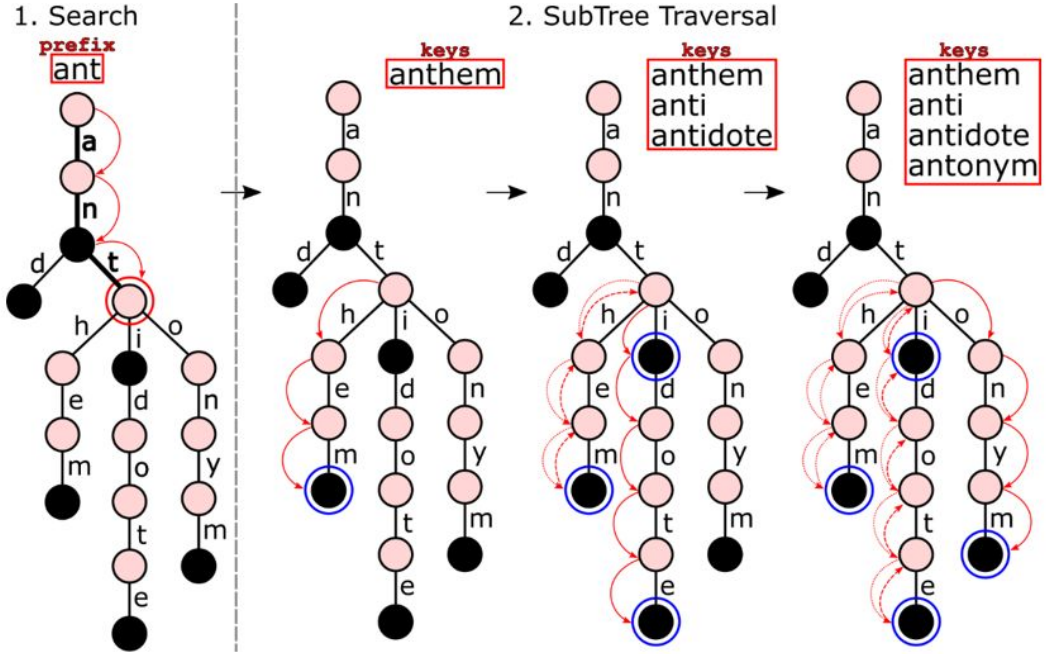
Time Complexity:

- Insert: **O(m)**
- Search: **O(m)**
- Prefix Search: **O(m + k)** (where m is the length of the prefix, and k is the number of strings with the prefix).

Space Complexity: **O(ALPHABET_SIZE × m × n)**

- Depends on the alphabet size, the length of strings, and the number of strings stored.

PREFIX TREE (SEARCH)



https://www.manning.com/books/advanced-algorithms-and-data-structures

PREFIX TREE (TRIE)

ADVANTAGES

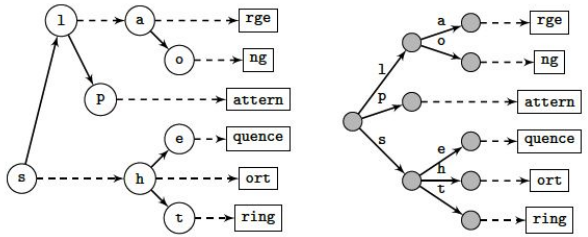
- **Fast Search**: Direct character-by-character traversal ensures O(m) time complexity.
- **Prefix Efficiency**: Ideal for implementing search suggestions or file autocompletion.
- **Scalability**: Handles large datasets where prefix-based operations are common.

LIMITATIONS

- **High Memory Usage:** Storing each character as a separate node can result in significant overhead for large alphabets or datasets.
- **Implementation Overhead:** Requires careful handling of node pointers and memory allocation.

OPTIMIZATIONS

- **Compressed Trie (Radix Tree):**
  - Combine nodes that have a single child into one node.
  - Example: Instead of storing `File` as four nodes (`F`, `i`, `l`, `e`), store it as one node with `File`.
- **Ternary Search Trie:**
  - Use three-way branching (less than, equal to, greater than) for improved space efficiency.



https://125-problems.univ-mlv.fr/problem49.php

TREES (KEY TAKEAWAYS)

Tree Type	Strengths	Ideal Use Cases
Binary Search Tree	Simple, efficient for basic search/insertion	**Small datasets** where perfect balance isn't critical. Suitable for in-memory data storage.
AVL Tree	Self-balancing, guarantees logarithmic performance	**Frequent updates** and **searches** requiring predictable O(log n) time, e.g., databases, memory caches.
Red-Black Tree	Self-balancing, efficient insertions/deletions	**Dynamic systems** with **frequent insertions and deletions**, e.g., operating system schedulers, networking applications.
B-Tree	Handles massive datasets, optimized for disk access	**Databases, file systems**, and applications handling **large-scale data** with disk I/O.
Trie (Prefix Tree)	Efficient for prefix-related operations	**Autocompletion, spell checking, IP routing**, and **dictionary lookups** where prefixes are critical.

BINARY HEAP

The Binary Heap is a **complete** binary tree that **efficiently manages priority-based operations**. It's a foundational data structure in scenarios where elements need to be **dynamically prioritized**, such as task scheduling or file synchronization.

ALGORITHMIC DESIGN

- **Max-Heap**: The value of the **root node is greater than or equal to its children** (used when the highest priority element is needed first).
- **Min-Heap**: The value of the **root node is smaller than or equal to its children** (used when the lowest priority element is needed first).

COMPLEXITIES

- **Time Complexity**:
  - Insert: **O(log n)**
  - Delete (Extract Max/Min): **O(log n)**
  - Access Max/Min: **O(1)**
- **Space Complexity**: **O(n)**

USE CASE EXAMPLE

- **Scenario**: Task Scheduling and Dynamic Priority Management.
- **Limit:** The heap only provides access to the highest or lowest priority element, which may not be suitable for tasks requiring random access or intermediate priorities.

HEAP (VARIANTS)

Operation	Binary Heap	Binomial Heap	Fibonacci Heap	d-ary Heap	Leftist Heap	Pairing Heap
Insertion	O(log n)	O(1)	O(1)	O(log d n)	O(log n)	O(1)
Deletion	O(log n)	O(log n)	O(log n)	O(d log d n)	O(log n)	O(log n)
Find Min/Max	O(1)	O(log n)	O(1)	O(1)	O(1)	O(1)
Merge	O(n)	O(log n)	O(1)	O(n)	O(log n)	O(1)
Decrease Key	O(log n)	O(log n)	O(1)	O(log d n)	O(log n)	O(1)

LEAST RECENTLY USED (LRU)

The Least Recently Used (LRU) **cache eviction policy** ensures that the **most recently accessed** files remain in memory, while the **least recently used ones are evicted** when the cache reaches its limit.

ALGORITHMIC DESIGN

- **Data Structure**: Combines a **doubly linked list** (to maintain the order of access) with a **hashmap** (to provide O(1) lookups and updates).
- **Operations**: **Access (Read/Write)**
  - If the file exists in the cache, move it to the front of the list (most recently used position).
  - If not, add it to the cache and evict the least recently used file if the cache is full.

COMPLEXITIES

- **Time Complexity**:
  - Access, Insert: **O(1)**
  - Eviction: **O(1)**
- **Space Complexity**: **O(n)** (for n cached files).

USE CASE EXAMPLE

A desktop application **frequently accessed files** like Documents, Downloads, and Pictures. The LRU cache ensures quick access by evicting the least recently used files as needed.

LIMITATIONS

Frequently accessed files may still be evicted if not recently used.

LEAST FREQUENTLY USED (LFU)

The Least Frequently Used (LFU) **cache eviction policy** prioritizes files that are **accessed frequently**, evicting files with the lowest access counts when the cache is full.

ALGORITHMIC DESIGN

- **Data Structure**: Combines a **frequency counter** with a **priority queue or hashmap** for efficient tracking and updates.
- **Operations**: **Access (Read/Write)**
  - Increment the access frequency for the file.
  - If the cache is full, evict the file with the lowest frequency.

COMPLEXITIES

- **Time Complexity**:
  - Access, Insert: **O(log n)** (using a priority queue) or **O(1)** (with advanced implementations).
  - Eviction: **O(log n)**
- **Space Complexity**: **O(n)** (for n cached files).

USE CASE EXAMPLE

Ideal for storing reference files that are accessed repeatedly, such as project documentation.

LIMITATIONS

Maintaining frequency counts adds computational complexity compared to LRU.

LRU AND LFU (KEY TAKEAWAYS)

Feature	LRU	LFU
Eviction Basis	Recency of access	Frequency of access
Best Use Case	Short-term usage patterns	Long-term usage patterns
Access Complexity	**O(1)**	**O(log n)** or **O(1)**
Adaptation Speed	Instantaneous	Gradual

A SYMPHONY OF DATA STRUCTURES

Each data structure brings its unique instrument to the orchestra:

- **Start Simple**: Use BSTs for lightweight, small-scale systems.
- **Balance Dynamically**: Move to AVL or Red-Black Trees for dynamic updates.
- **Scale Up**: Use B-Trees for large-scale datasets and cloud integrations.
- **Enhance Search**: Leverage Tries for predictive search.
- **Adapt with Priorities**: Use Treaps for dynamic ranking.
- **Prioritize Urgency**: Use Heaps for scheduling tasks.
- **Boost Performance**: Use LRU/LFU caches to reduce latency.

Together, they craft a masterpiece of efficiency and scalability.

GENERAL DESCRIPTION OF THE UNIT

TAILORED EFFICIENCY

No single data structure solves all problems. Each excels in specific scenarios—use Trees for hierarchy, Heaps for priorities, and Caches for speed to craft a balanced system.

SCALABILITY VS. SIMPLICITY

Lightweight structures like BSTs work for smaller systems, but as complexity grows, advanced structures like B-Trees and LFU Caches are essential for scaling efficiently.

TRADE-OFFS DRIVE DESIGN

Every data structure introduces trade-offs in space, time, and complexity. Combining them strategically ensures optimal performance for diverse file system demands.

THANKS !

Connect with Me

- Blog: https://helabenkhalfallah.com/blog/
- Medium Articles: https://helabenkhalfallah.medium.com/

Explore Some Of My Projects

- DSA Toolbox Repository: https://github.com/helabenkhalfallah/dsa-toolbox
- DSA Toolbox Documentation: https://helabenkhalfallah.github.io/dsa-toolbox/
- Code Health Meter: https://github.com/helabenkhalfallah/code-health-meter

Social Media

- Twitter (X): https://x.com/b_k_hela
- LinkedIn: https://www.linkedin.com/in/h%C3%A9la-ben-khalfallah-4a104014/