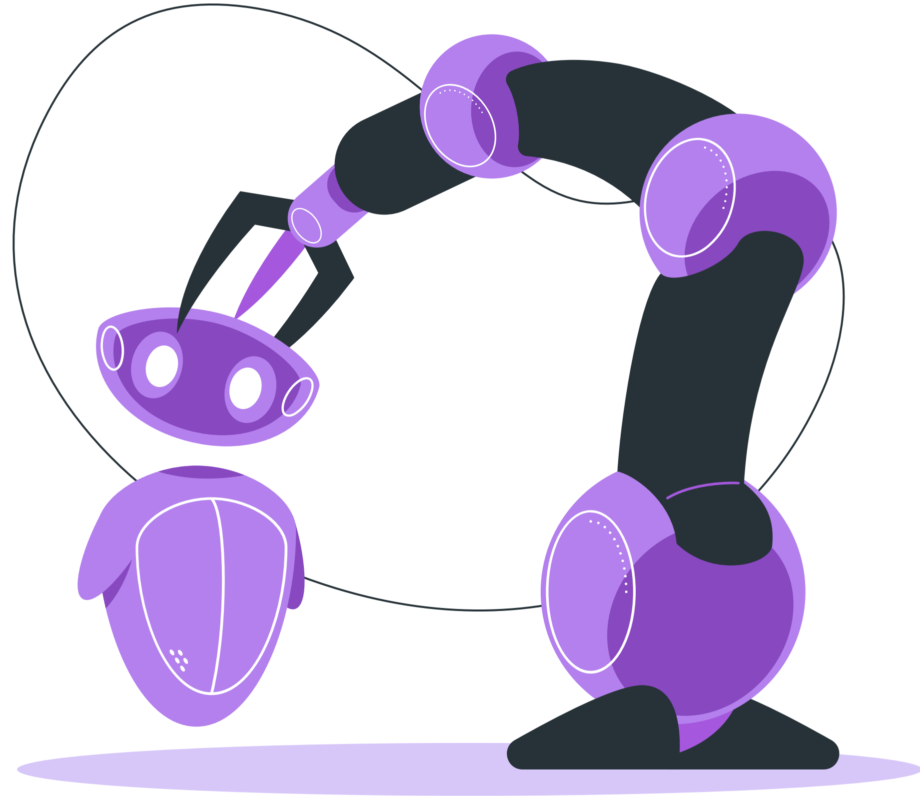
Advanced Data Structures

The world beyond Array and Hashmap



Ben Khalfallah Héla - 2024

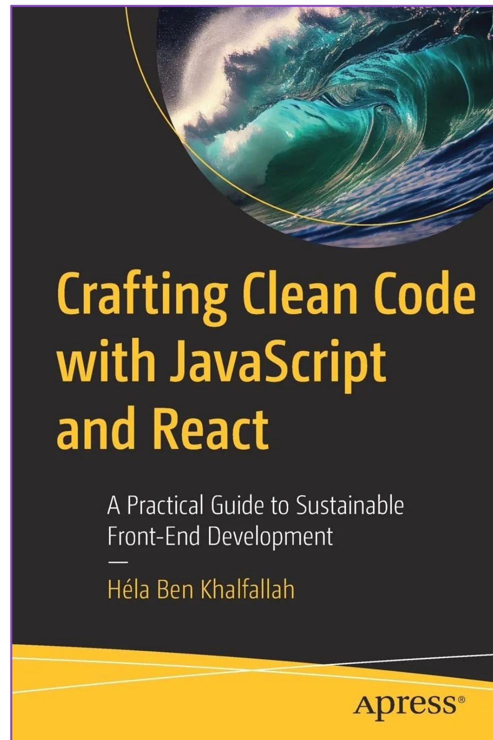
Whoami

Hello! I'm Héla Ben Khalfallah

I'm a Frontend Senior Expert, Software Architect, and Senior Web Developer with a passion for creating scalable, efficient, and user-friendly applications.

With experience in FrontendOps and a strong focus on technologies like React.js and Node.js, I enjoy building practical, high-performing solutions that are both maintainable and impactful.

In my book, **Crafting Clean Code with JavaScript and React**, I share practical insights and proven techniques to help developers write cleaner, more effective code and design applications that are built to last.



https://amzn.eu/d/5PCKUFm

Table of Contents

1 **Trees**
O(log n) as a breeze

2 **Heaps**
The challenge of priority

3 **Treaps**
Yet Another Way to Balance BST

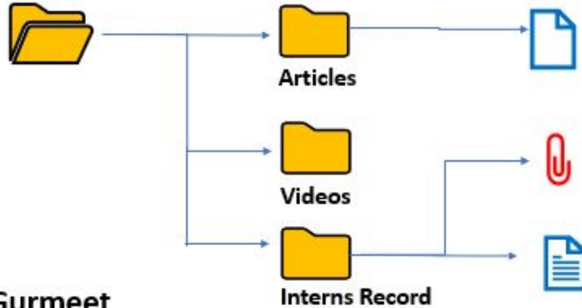
4 **LFU & LRU**
The magic of Caching

5 **Key takeaways**
When to use What?

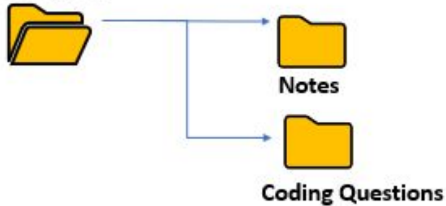
6 **Conclusion**
Next adventures

Folders hierarchy System in Computers

takeuforward



Gurmeet



https://takeuforward.org/binary-tree/application-of-tree-data-structure/

The use case

A desktop application to manage both **local and cloud file systems**.

https://itnext.io/data-structure-for-file-management-application-1cf88c3ca239

Desktop application (Internal)

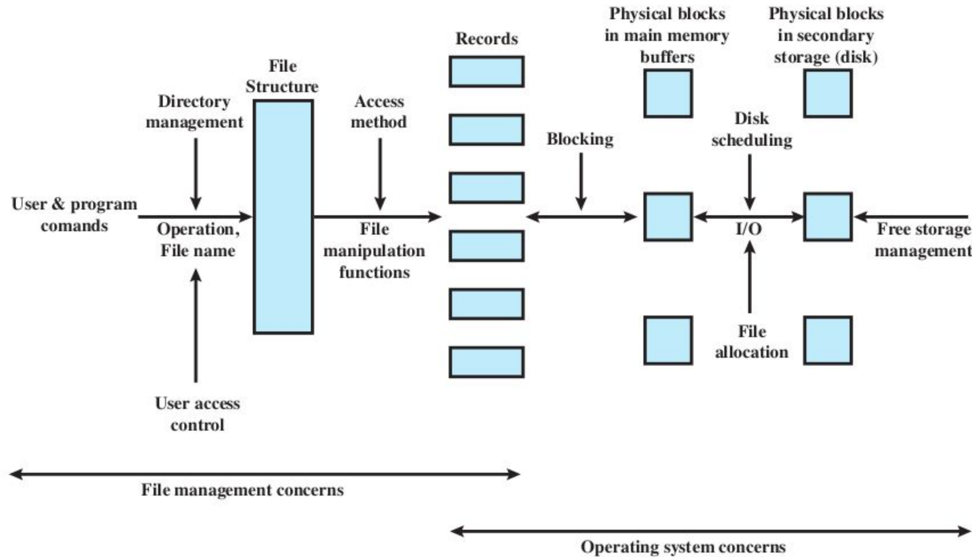
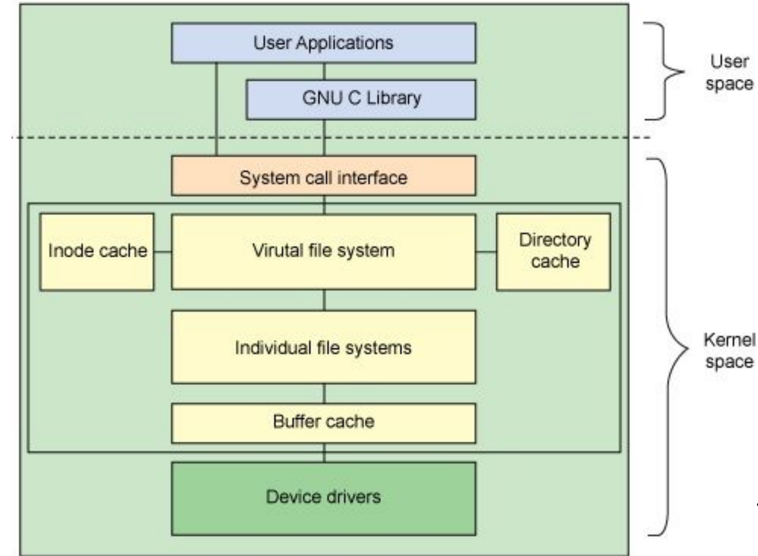


Figure 12.2 Elements of File Management

https://www.usna.edu/Users/cs/crabbe/SI411/current/io/fs.html



https://embedkari.com/linux-filesystem/

Requirements

- Desktop applications managing file systems must handle **complex**, **hierarchical** structures with **numerous** files and folders.
- Users expect **fast operations** like updating, deleting, renaming, and finding files.
- Handling **access priorities**.
- **Recently** used and **Most Frequently** used.

Performance estimation

O(n): Where n Represents the Input Size

If an algorithm has O(n) time complexity, it means the execution time grows linearly with the input size. For example, if n doubles, the time roughly doubles.

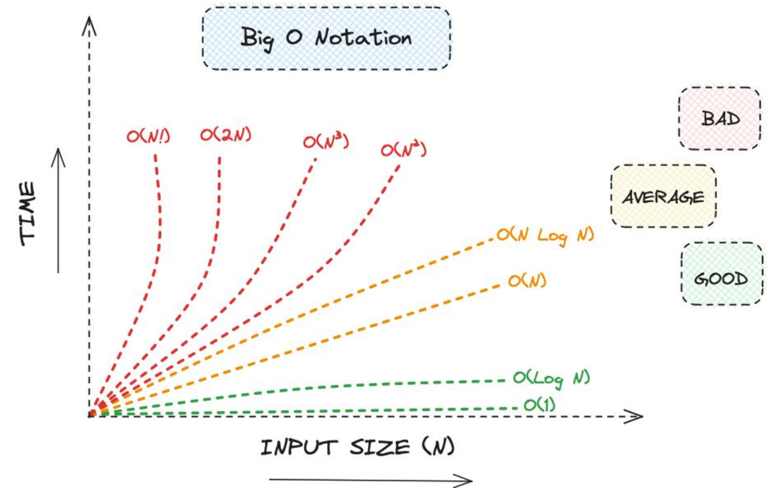
CPU Time = IC * CPI / Clock Rate

- IC (Instruction Count): Total instructions executed.
- CPI (Cycles per Instruction): Average cycles per instruction.
- Clock Rate: Processor speed (GHz).

More steps (IC) → Higher CPU Time → Worse Performance.

Optimize:

- Reduce IC (better algorithms).
- Lower CPI (efficient instructions).
- Faster Clock Rate helps, but code efficiency matters more.



https://medium.com/@agustin.ignacio.rossi/coding-interview-preparation-big-o-notation-8066c3b0ce60

Limitations

- Arrays are **inefficient for insertions and deletions in the middle of a large dataset**.
- HashTables are optimized for fast access but **lack support for hierarchical relationships**, which are essential for a file system.

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

https://www.bigocheatsheet.com/

We need something more efficient!

https://en.wikipedia.org/wiki/List_of_file_systems

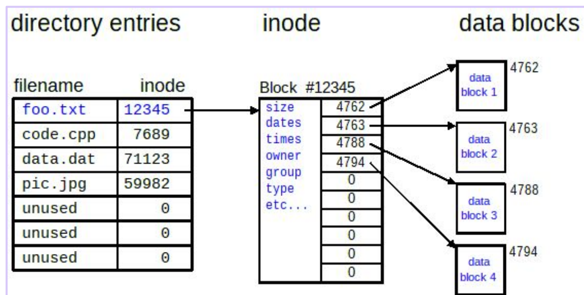
Inspirations (File systems)

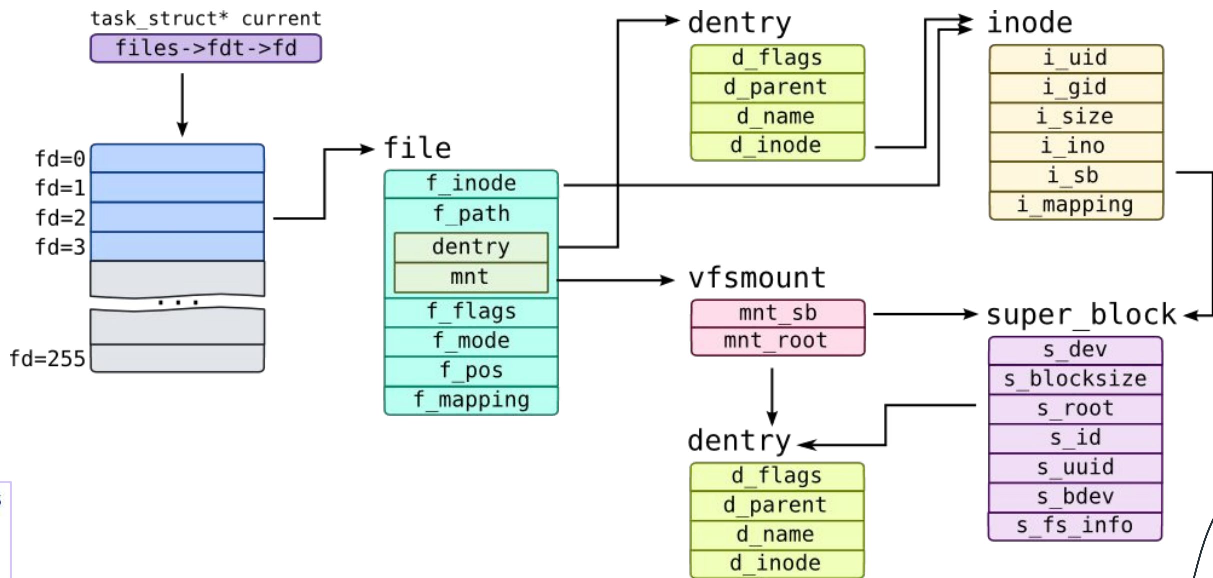
1. NTFS (New Technology File System)
   a. In NTFS, **B-tree** structures optimize folder record management, **enabling efficient indexing and search operations in large folders**.
   b. Unlike FAT, which scans all file names sequentially, NTFS leverages B-trees to group similar file names, **minimizing disk accesses** and outperforming FAT in handling large directories.
2. ZFS (Zettabyte File System)
   a. ZFS uses **AVL trees** (and **B-trees in newer versions**) for efficient **metadata management**.
   b. **Merkle trees** for robust data integrity, ensuring fast access, checksums, and redundancy in a high-performance filesystem.
3. APFS (Apple File System)
   a. In APFS, **B-Trees** are integral for **managing filesystem metadata** efficiently.
   b. B-Trees enable APFS's **scalability**, ensuring efficient operations in tasks like block allocation, metadata updates, and directory traversal.
4. JFS (IBM Journaled File System)
   a. Tree Type: **B+ trees** for directory indexing and extent management.
   b. Purpose: Uses B+ trees to **optimize** directory lookups and extent indexing, providing **efficient access** to metadata and file allocation.
5. XFS (High-Performance File System)
   a. XFS leverages **B+ trees** to manage directories and file allocations efficiently.
   b. File data is stored in contiguous blocks called extents, which are **indexed** via B+ trees **for faster lookups and access.**

File systems (metadata)

Metadata refers to **information about files and directories**, not the actual file contents. Examples include:

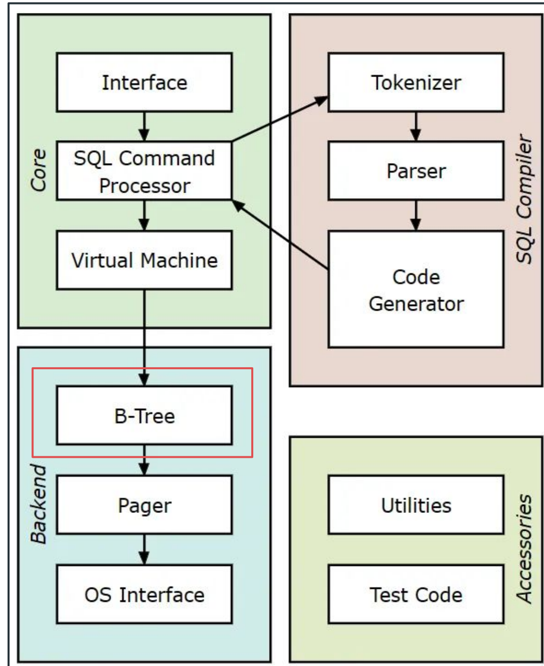
- File name
- File size
- Timestamps (created, modified, accessed)
- File location on the disk (extent information)
- Directory structure and hierarchy
- Permissions and ownership



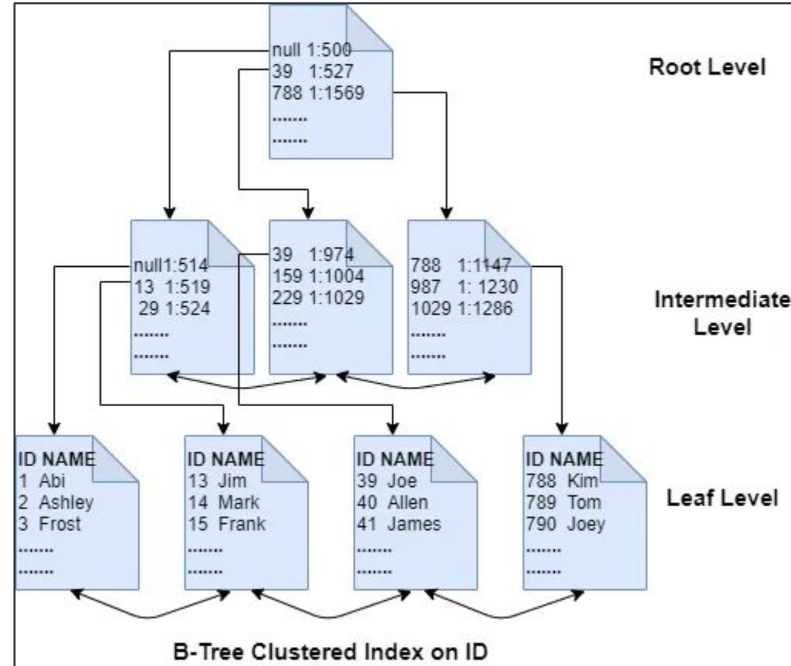


https://myaut.github.io/dtrace-stap-book/kernel/fs.html

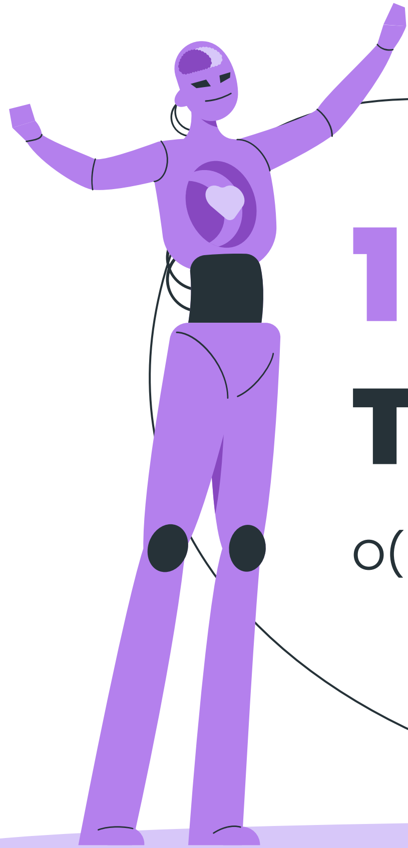
Inspirations (Databases)



Architecture of SQLite



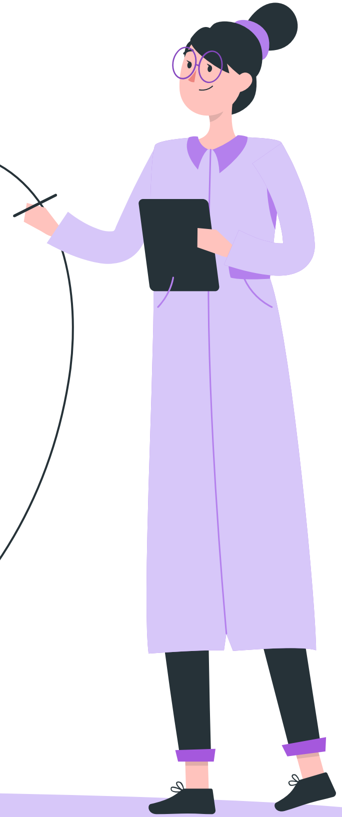
Understanding CRUD Operations on Tables with B-tree Indexes, Page-splits, and Fragmentation — SQLServerCentral



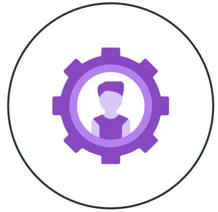
1

Trees

O(log n) as a breeze



Trees

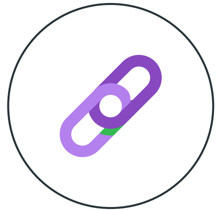


What is a Tree?

Tree structure and components

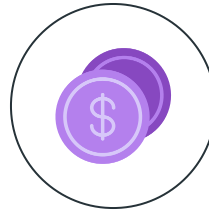
How can trees help?

Why Trees are Useful?



Types of Trees

BST, AVL, Red-Black, B-Tree and Trie

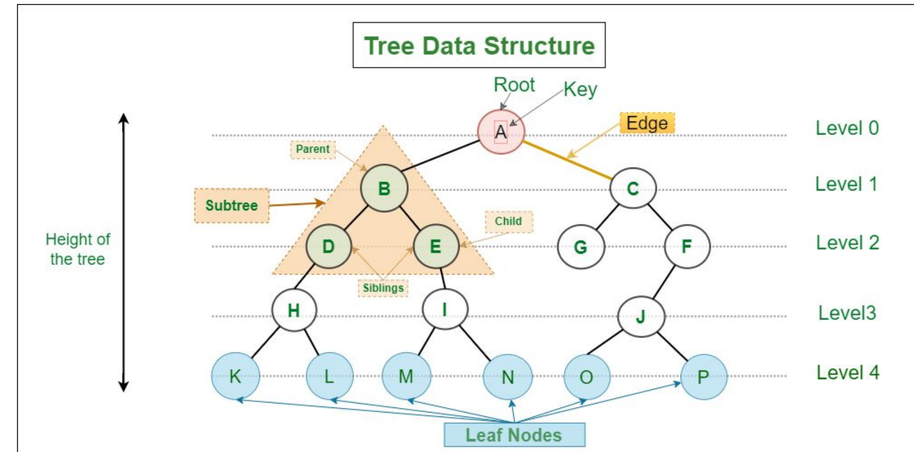


Key takeaways

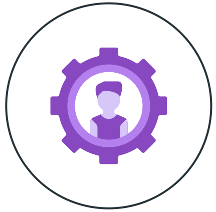
When to use What?

What is a Tree?

- **Leaves:** Represent files, which are end points with no further contents.
- **Root:** The top-level storage location, like the main hard drive or top directory, from which the file system starts.
- **Nodes:** Each folder is a node, acting as a container for files and other folders.
- **Branches:** Connections between folders, showing containment relationships and creating the hierarchy.
- **Hierarchical Structure:** The tree structure mirrors the file system's organization, enabling efficient organization, search, and manipulation.



Why Trees are Useful?

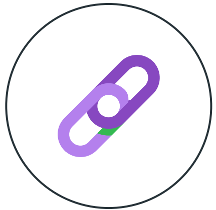


Masters of Organization

Trees bring order, arranging files and folders in a clear hierarchy, like organizing books in a library.

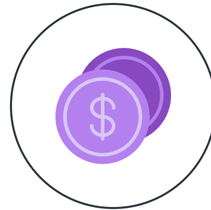
Efficient Searching

Trees allow quick navigation through the hierarchy, making it easy to locate files without scanning the entire system.



Adaptable to Changes

Trees handle dynamic changes well, easily accommodating new, deleted, or renamed files and folders.



Optimized for Operations

Trees streamline common file actions—adding folders, deleting files—making these operations efficient and intuitive.

Different Kinds of Trees

- **BST (Binary Search Tree)** : Simple, efficient for basic search and insertion.
- **AVL Tree** : Self-balancing, guarantees logarithmic performance.
- **Red-Black Tree** : Self-balancing, efficient for frequent insertions and deletions.
- **B-Tree** : Handles massive datasets, optimized for disk access.
- **Trie** : Efficient for prefix-related operations.

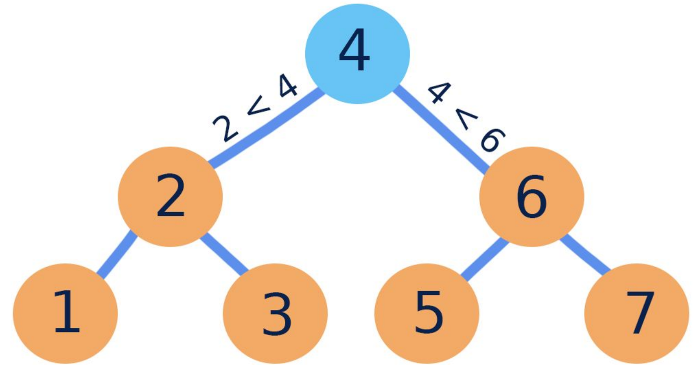
BST

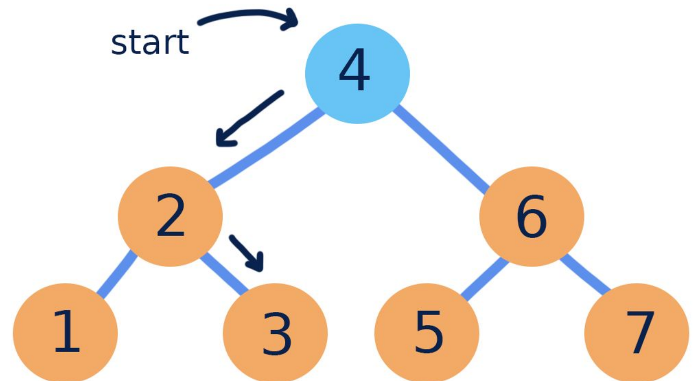
A BST is a special kind of binary tree with a unique property:

- **Left is Less:** The left child of a node always has a value less than its parent node.
- **Right is Greater:** The right child of a node always has a value greater than its parent node.

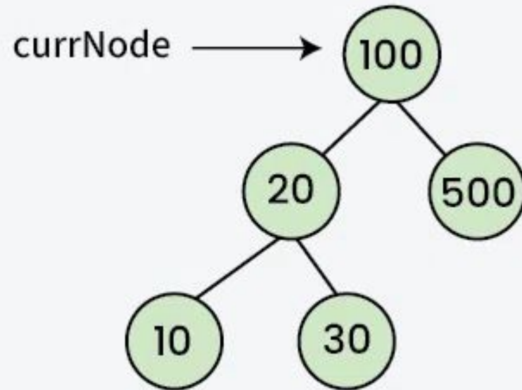
This simple rule, **applied throughout the tree**, creates a sorted structure that allows for incredibly efficient searching, insertion, and deletion of data.

https://courses.grainger.illinois.edu/cs225/sp2019/notes/bst/





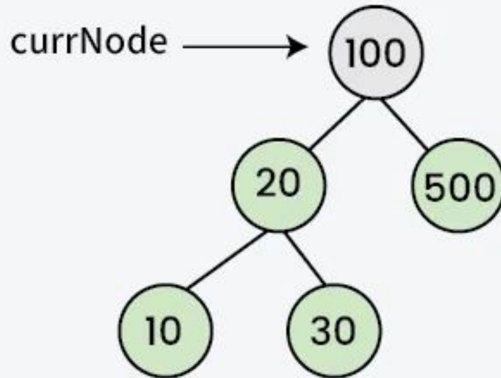
BST (Insertion)



https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

BST (Insertion)

02 Step | Comparing key with root node

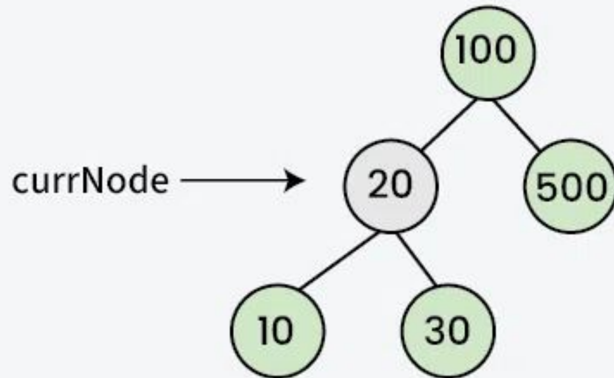


Since 100 is greater than key(40), move the pointer to the left child (20).

Insertion in BST

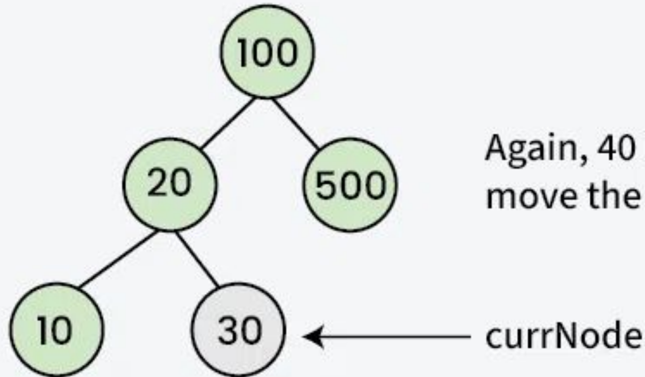
https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

BST (Insertion)



https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

BST (Insertion)

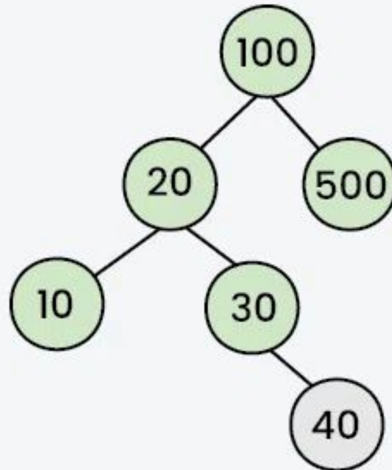


https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

BST (Insertion)

05 Step

Insert item to the right of 30



Now, pointer refers to null. Hence, Insert key(40) at this position

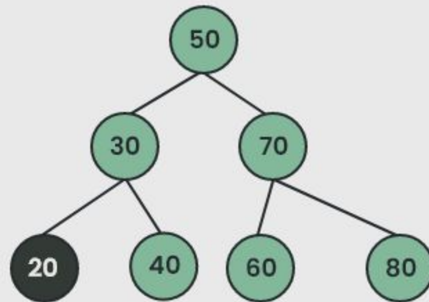
Inserted Node

Insertion in BST

https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

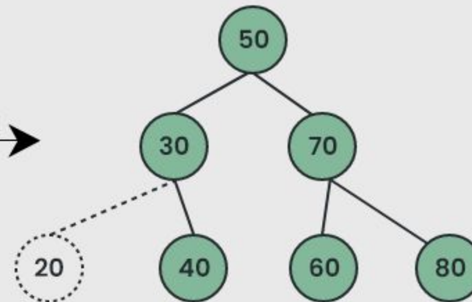
BST (Deletion)

Case 1 : Delete A Leaf Node In BST



Delete Node 20

Assign Node To Null



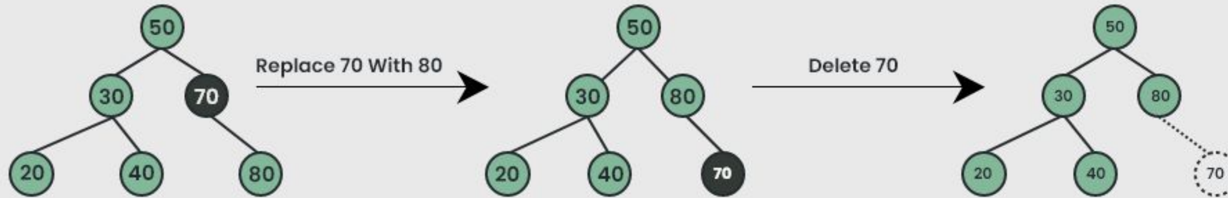
Deleted Node 20

Deletion In BST

https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

BST (Deletion)

Case 2: Delete A Node With Single Child In BST



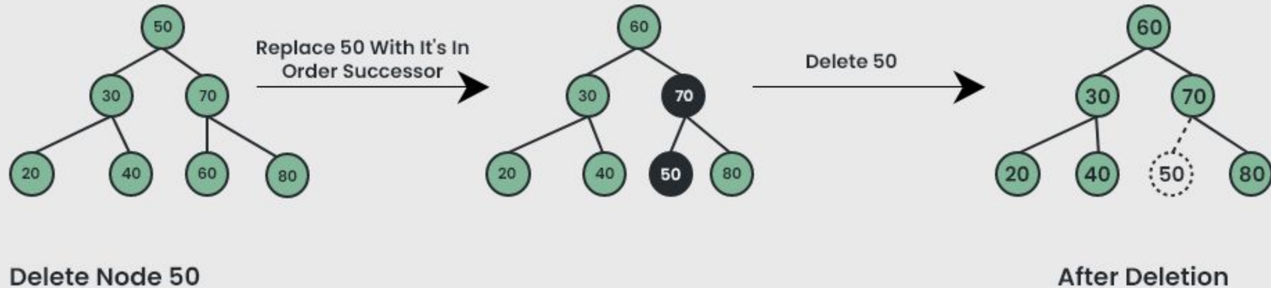
Delete Node 70

After Deletion

Deletion In BST

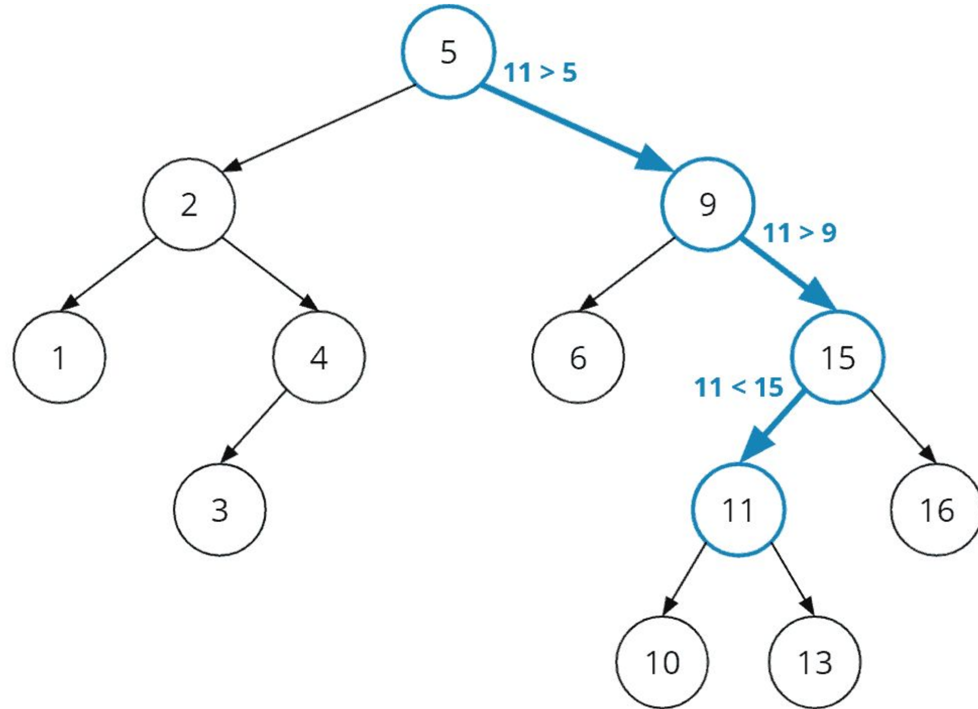
https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

BST (Deletion)



https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

BST (Search)



https://www.happycoders.eu/algorithms/binary-search-tree-java/

BST (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/trees/bst/BinarySearchTree.ts

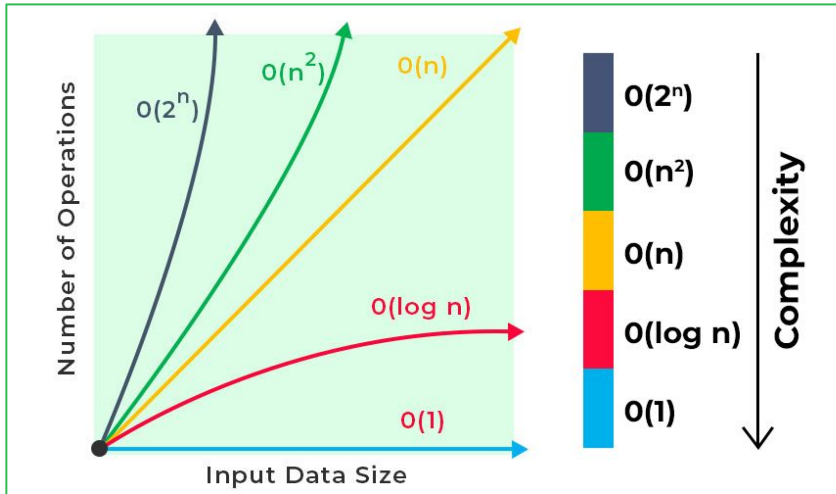
BST (In Action)

https://www.cs.usfca.edu/~galles/visualization/BST.html

BST (Time and Space Complexity)

Operation	BST (Average)	BST (Worst)	Array (Average)	Array (Worst)
Add	O(log n)	O(n)	O(1)	O(n)
Update	O(log n)	O(n)	O(1)	O(n)
Delete	O(log n)	O(n)	O(n)	O(n)
Search	O(log n)	O(n)	O(n)	O(n)

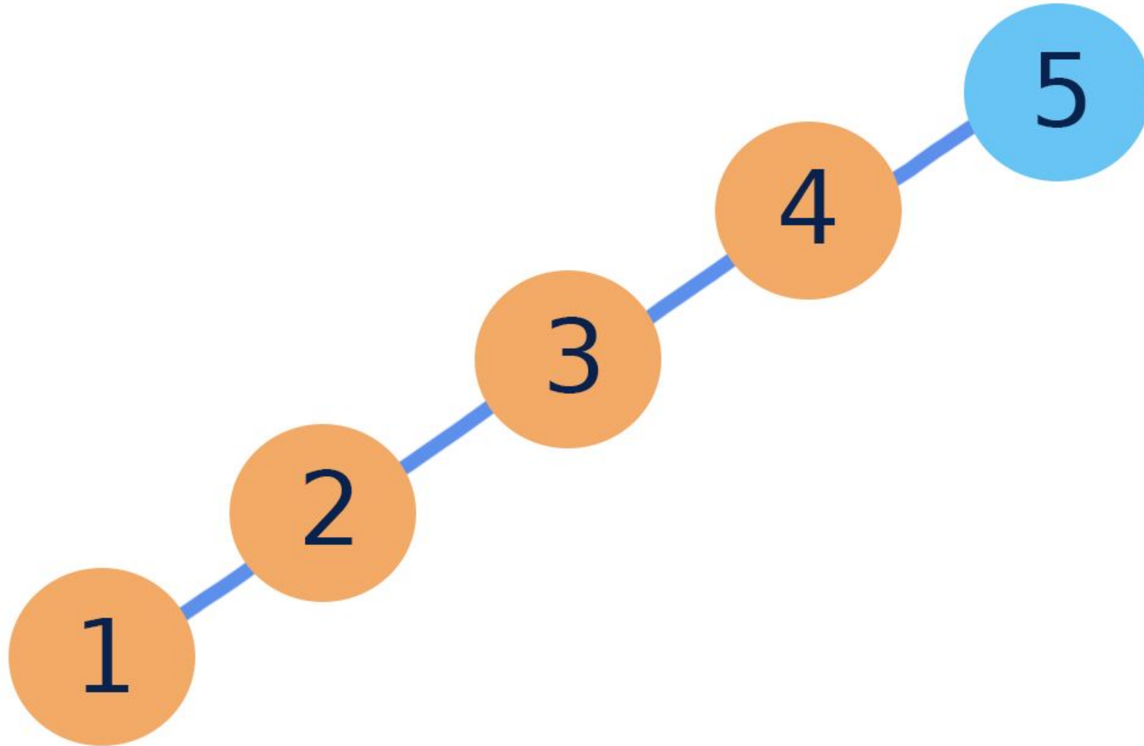
O(log n) vs O(n)



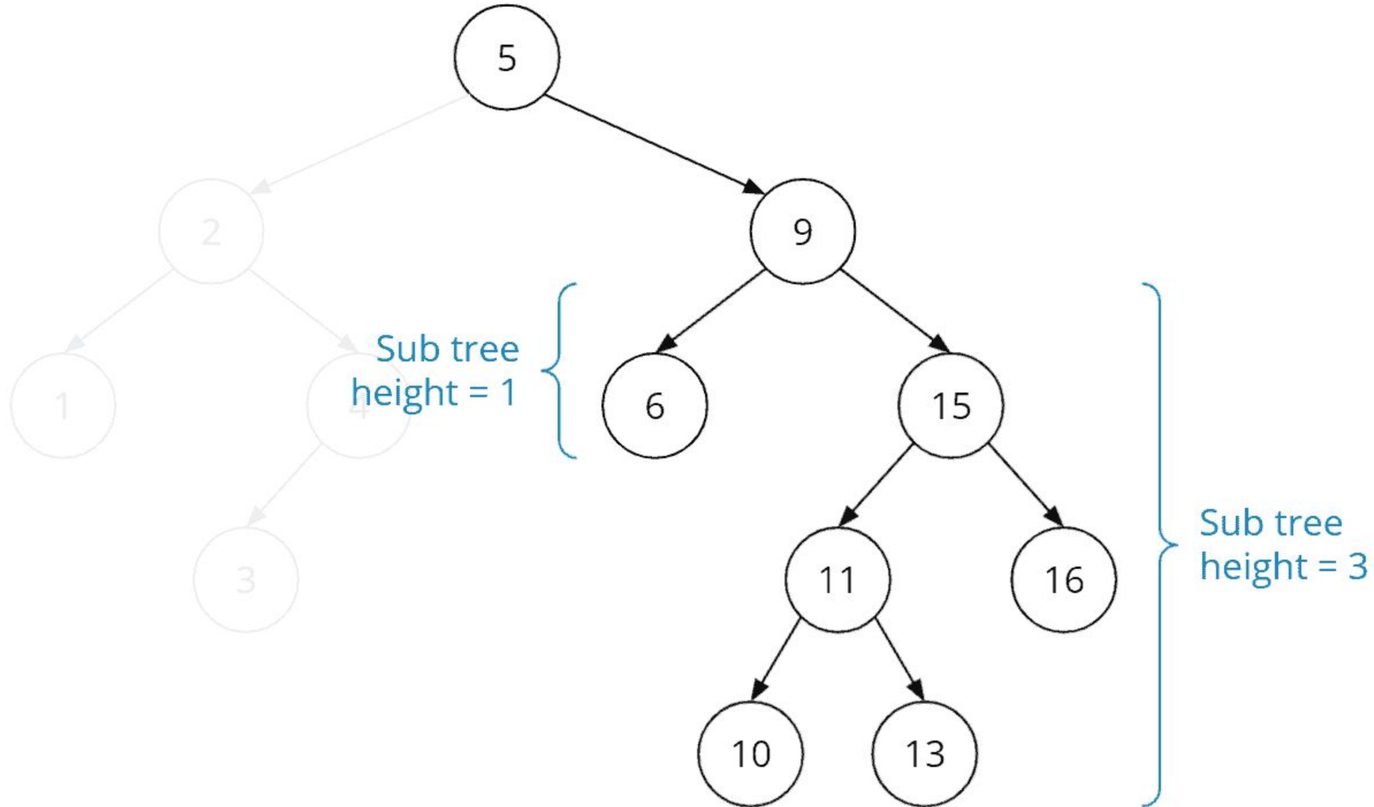
https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

Input Size (n)	O(log n)	O(n)
10	3.321928094887362	10
1000	9.965784284662087	1000
10000	13.287712379549449	10000

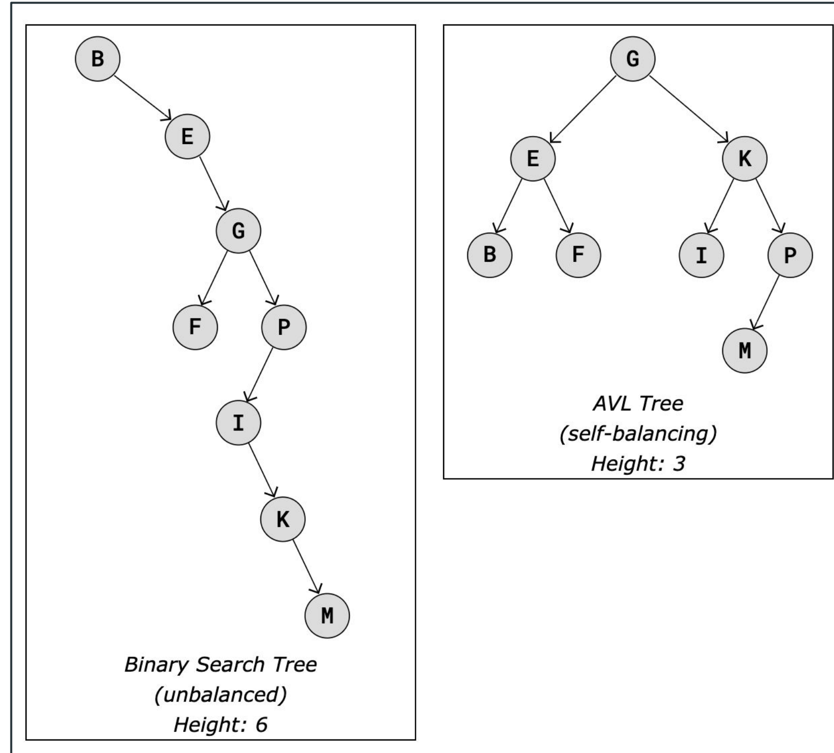
BST (Unbalanced)



BST (Unbalanced)



AVL Tree

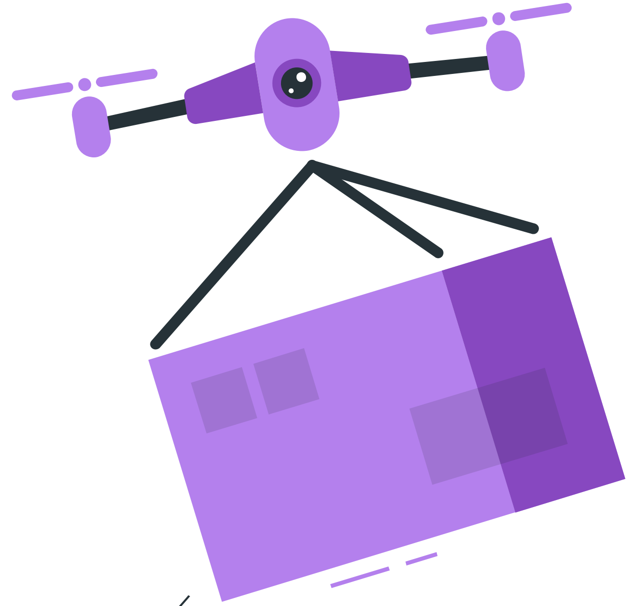


https://www.w3schools.com/dsa/dsa_data_avltrees.php

Rule 1

Every AVL Tree is a BST but not every BST is an AVL tree.

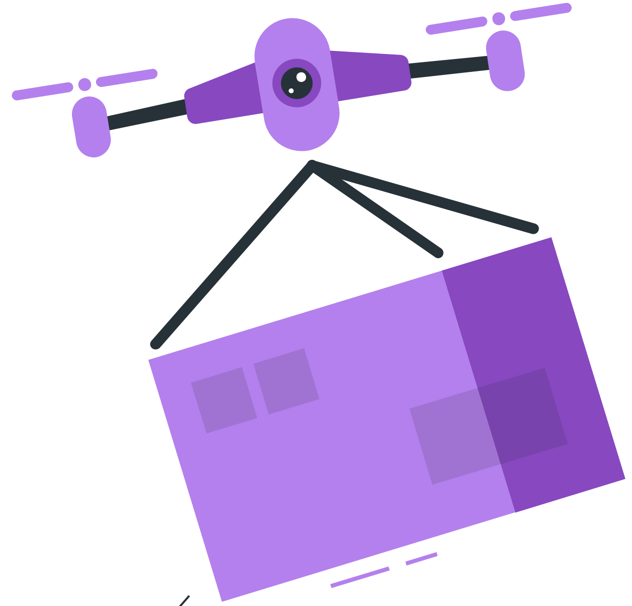
AVL Tree = BST + Balance Condition



Rule 2

In an AVL tree, **every node** maintains an extra information known as **balance factor**.

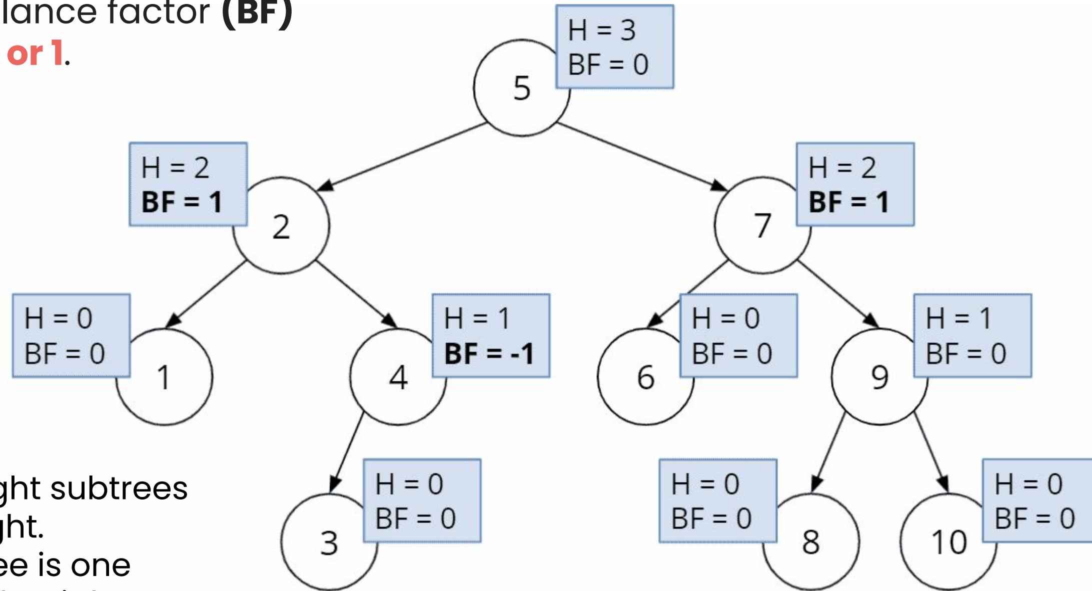
The balance factor of every node is either **-1, 0 or +1**.



Balance Factor= **height** of left subtree - **height** of right subtree

AVL Tree (Balanced)

In an AVL tree, the balance factor **(BF)** at each node **is -1, 0, or 1**.

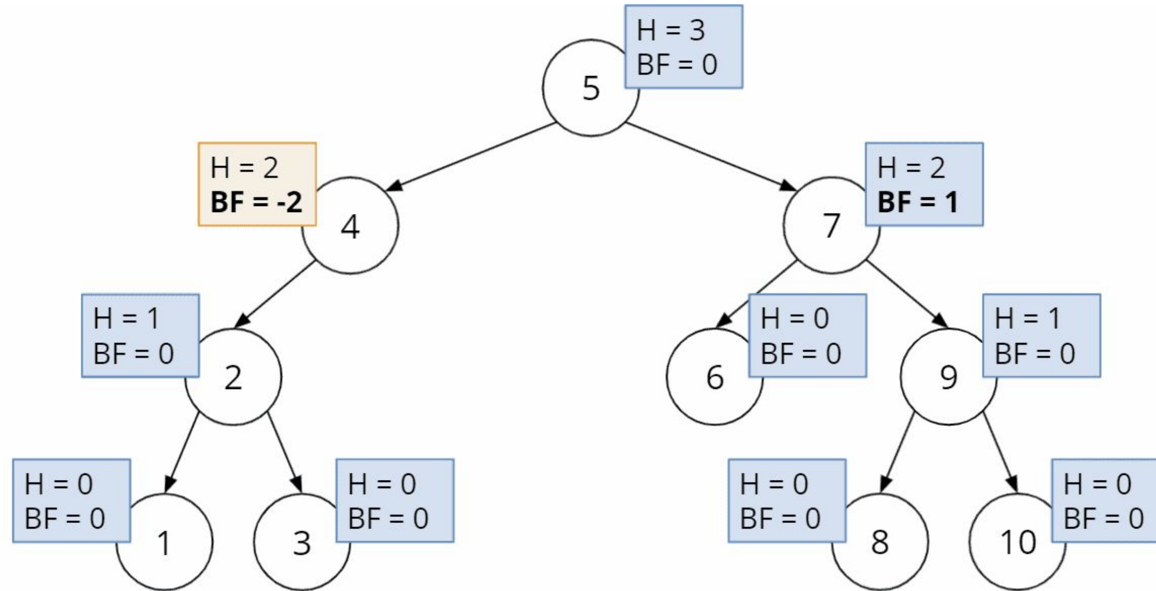


https://www.happycoders.eu/algorithms/avl-tree-java/

- **0:** The left and right subtrees are of equal height.
- **+1:** The left subtree is one level taller than the right.
- **-1:** The right subtree is one level taller than the left.

Balance Factor= **height** of left subtree - **height** of right subtree

AVL Tree (Unbalanced)



https://www.happycoders.eu/algorithms/avl-tree-java/

Balance Factor= **height** of left subtree - **height** of right subtree

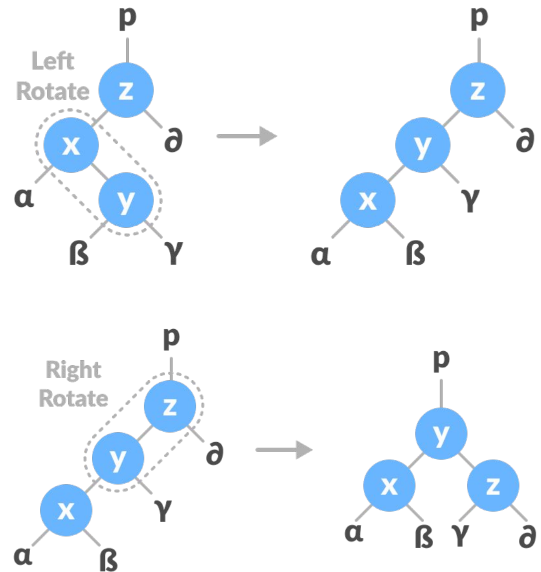
AVL Tree (Balancing)

There are three cases:

- If the **balance factor is < 0**, the node is said to be right-heavy.
- If the **balance factor is > 0**, the node is said to be left-heavy.
- A **balance factor of 0** represents a balanced node.

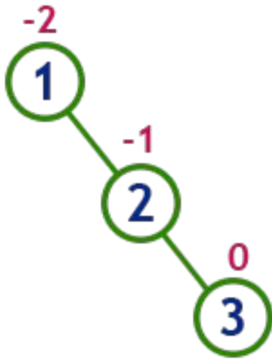
Rotations:

- **Left-Heavy (BF > 1):** Use a Right Rotation on the unbalanced node.
- **Right-Heavy (BF < -1):** Use a Left Rotation on the unbalanced node.
- **Double Rotation (cascade)** for mixed imbalances

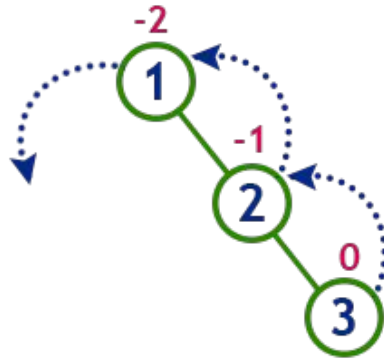


AVL Tree (Single Left Rotation)

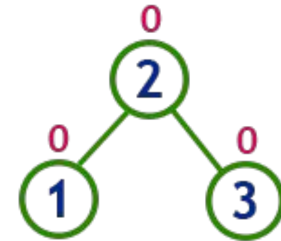
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left

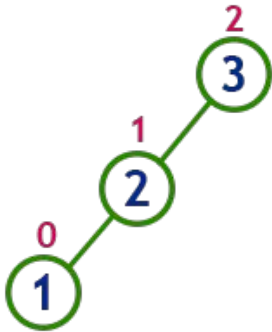


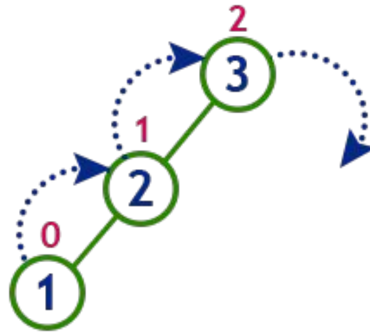
After LL Rotation Tree is Balanced

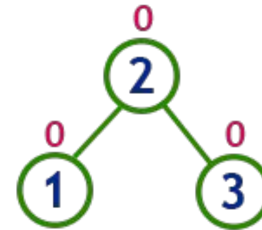
http://www.btechsmartclass.com/data_structures/avl-trees.html

AVL Tree (Single Right Rotation)

insert 3, 2 and 1







Tree is imbalanced

because node 3 has balance factor 2

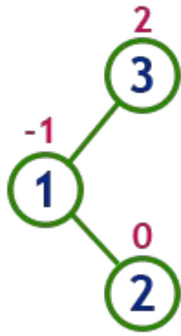
To make balanced we use RR Rotation which moves nodes one position to right

After RR Rotation Tree is Balanced

http://www.btechsmartclass.com/data_structures/avl-trees.html

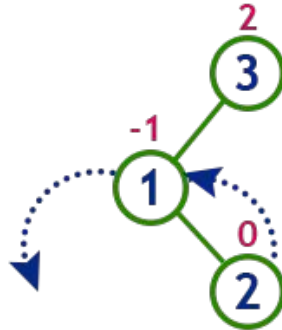
AVL Tree (Left-Right Rotation)

insert 3, 1 and 2

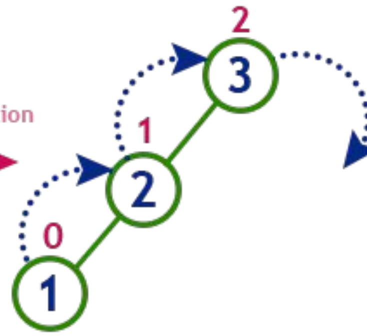


Tree is imbalanced

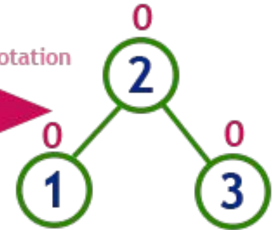
because node 3 has balance factor 2



LL Rotation



RR Rotation

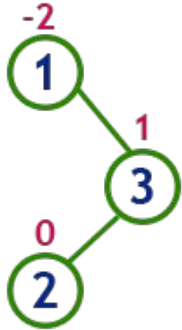


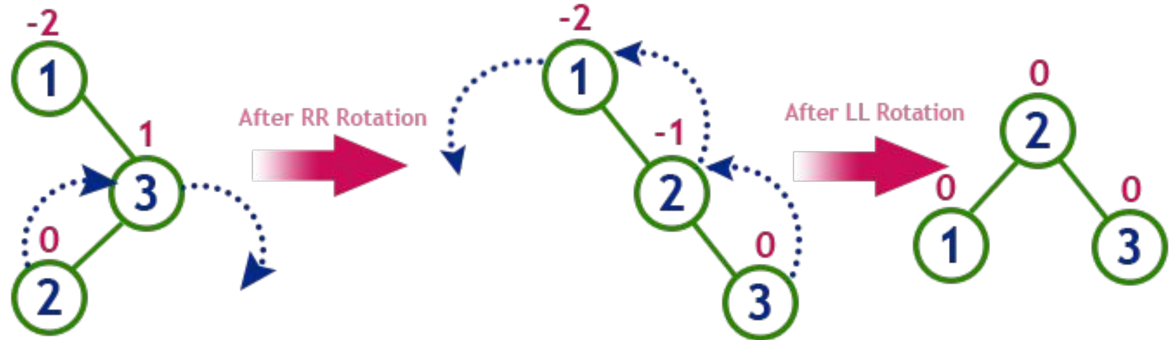
After LR Rotation
Tree is Balanced

http://www.btechsmartclass.com/data_structures/avl-trees.html

AVL Tree (Right-Left Rotation)

insert 1, 3 and 2





Tree is imbalanced

because node 1 has balance factor -2

RR Rotation

LL Rotation

After RL Rotation
Tree is Balanced

http://www.btechsmartclass.com/data_structures/avl-trees.html

AVL (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/trees/avl/AVLTree.ts

AVL (In Action)

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL Trees in File Systems (Strengths and Challenges)

Strengths:

- **Strict Balancing**: Ensures O(log n) for search, insert, and delete operations.
- **Optimal for Search**: Better lookup performance due to minimal tree height.
- **Ideal for Read-Heavy Workloads**: Perfect when file lookups dominate over updates.
- **Consistent Performance**: Predictable efficiency for managing metadata like directories and file indices.

Challenges:

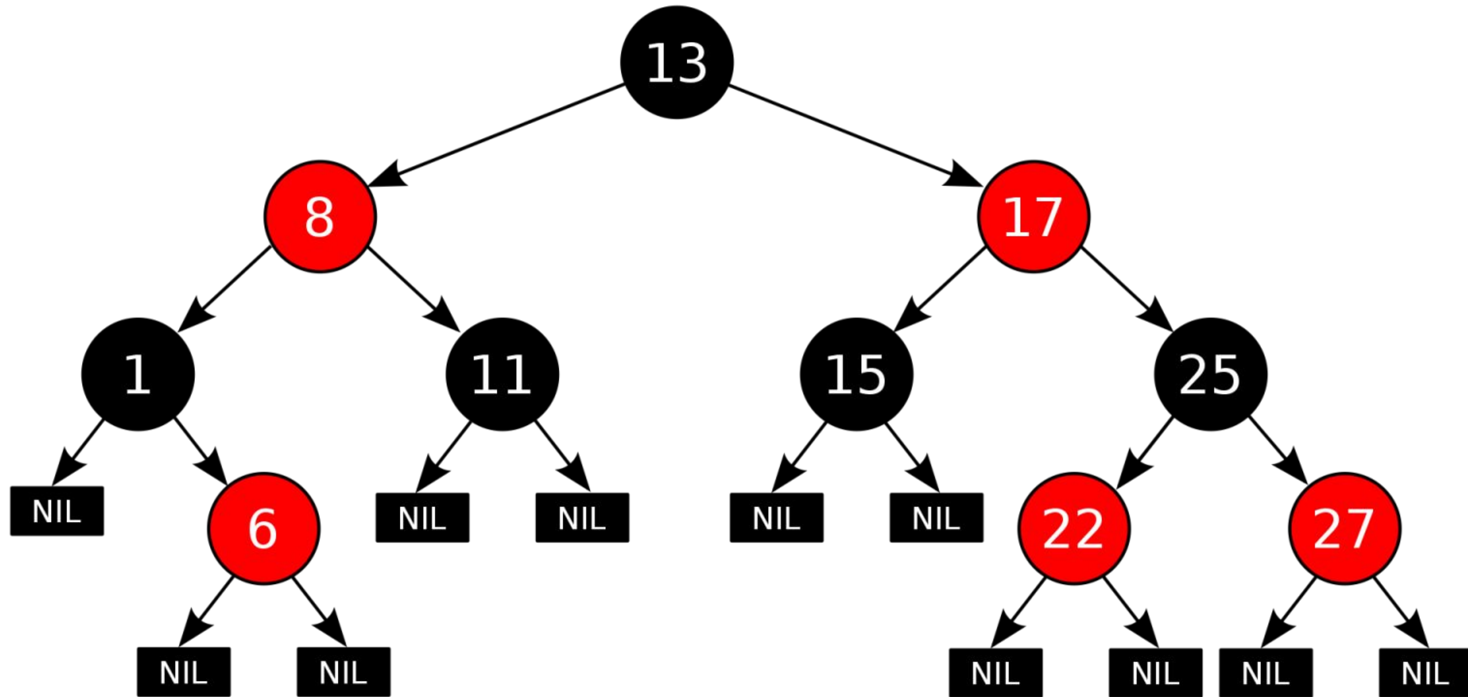
- **Insertion/Deletion Overhead**: Requires more **rotations** than Red-Black Trees, increasing update costs.
- **Complexity**: Maintaining strict balance adds implementation complexity.
- **Cache Locality**: **Nodes are scattered in memory**, leading to suboptimal cache performance.
- **Scalability**: Not ideal for disk-based systems; B-Trees perform better for large-scale storage with fewer I/O operations.

https://github.com/openzfs/zfs/pull/9181

AVL (Time and Space Complexity)

Operation	AVL Tree	BST (Average)	BST (Worst)
Add	O(log n)	O(log n)	O(n)
Update	O(log n)	O(log n)	O(n)
Delete	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Red-Black Tree

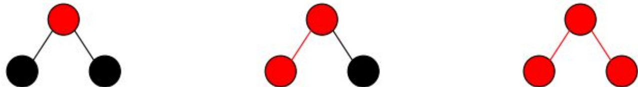


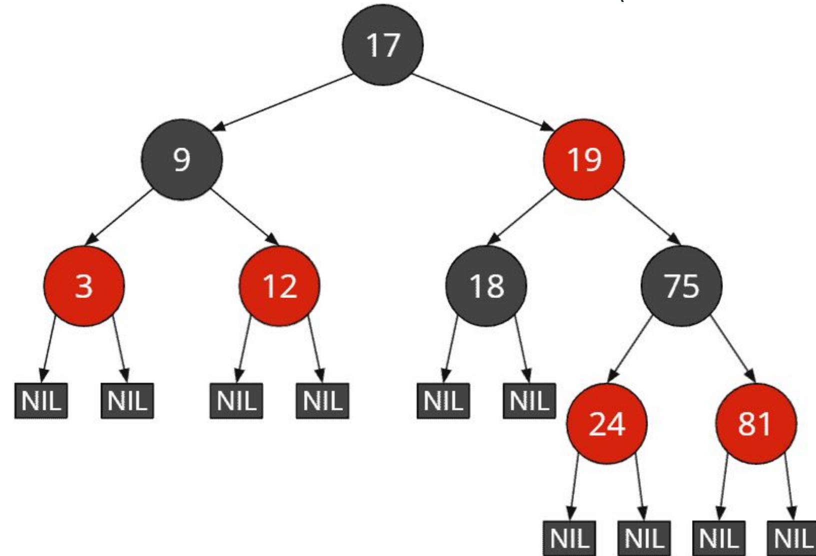
https://iq.opengenus.org/aa-trees/

Red-Black (Rules)

The following rules **enforce** the red-black tree balance:

1. Every node **is either red or black**
2. The **root is black**
3. If a node is red, then both its children are black
4. For each node, all path from the node to descendant leaves contain the **same number of black nodes**





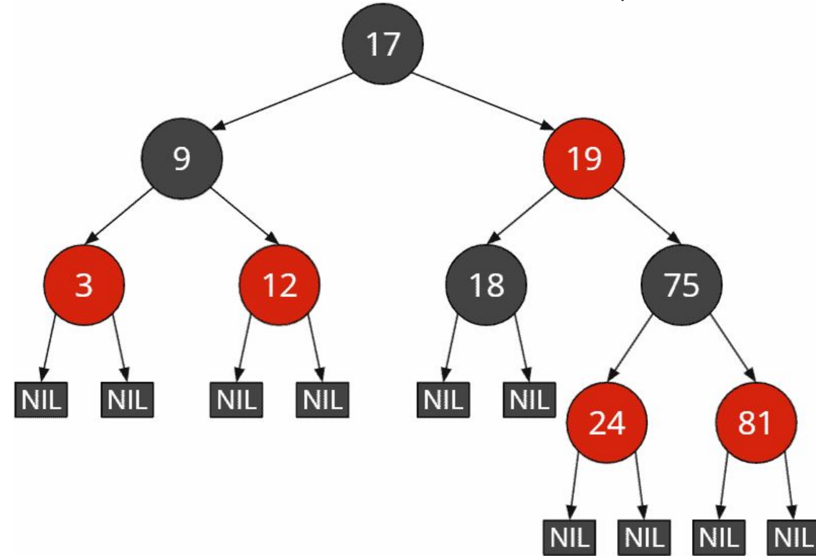
https://www.happycoders.eu/algorithms/red-black-tree-java/

Red-Black (Rules)

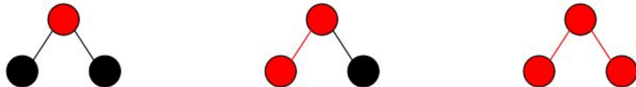
Why?

1. **Black Nodes:** Maintain overall balance.
2. **Red Nodes:** Spread out to avoid imbalance.

The structure stays balanced as we insert or delete nodes by using rotations.



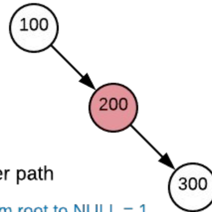
https://www.happycoders.eu/algorithms/red-black-tree-java/



Red-Black (Rules)

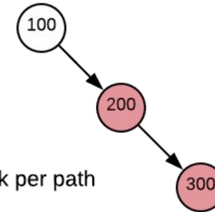
Unbalanced Trees = Invalid Red-Black

- ✓ 1 - Red or black
- ✓ 2 - Root is black
- ✓ 3 - No consecutive red
- ✗ 4 - Same number of black per path



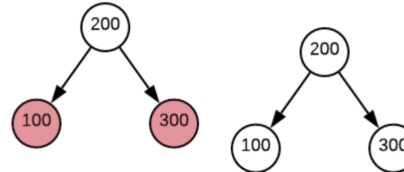
Balck node following left pointer from root to NULL = 1
Black nodes following right pointer from root to NULL = 2

- ✓ 1 - Red or black
- ✓ 2 - Root is black
- ✗ 3 - No consecutive red
- ✓ 4 - Same number of black per path



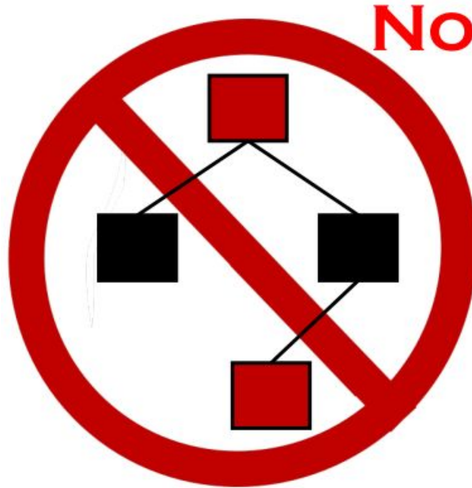
Balanced Trees = Valid Red-Black

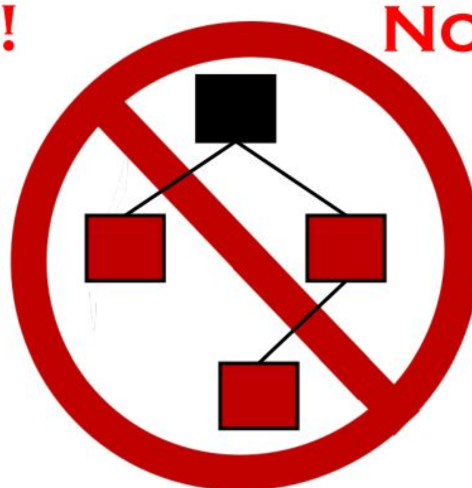
- ✓ 1 - Red or black
- ✓ 2 - Root is black
- ✓ 3 - No consecutive red
- ✓ 4 - Same number of black per path

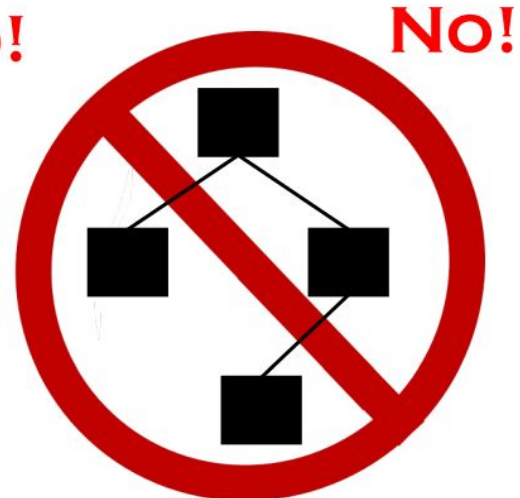


https://hideoushumpbackfreak.com/algorithms/list-data-struct-red-black-tree.html

Red-Black (Rules)







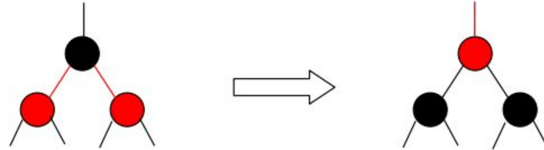
https://web.stanford.edu/class/archive/cs/cs161/cs161.1176/Slides/Lecture7-lowquality.pdf

Insertion and Deletion will violate the property of red-black tree. How to maintain the property?

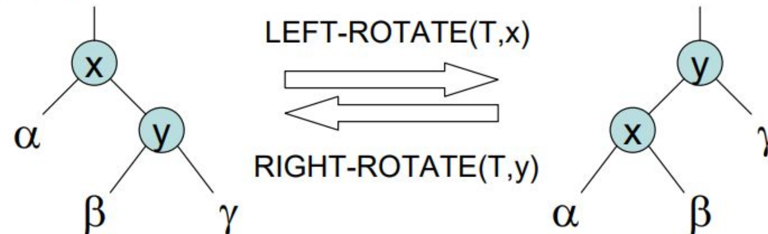
By Changing **Color** or **Rotation**

Red-Black (Maintain Property)

- Changing color

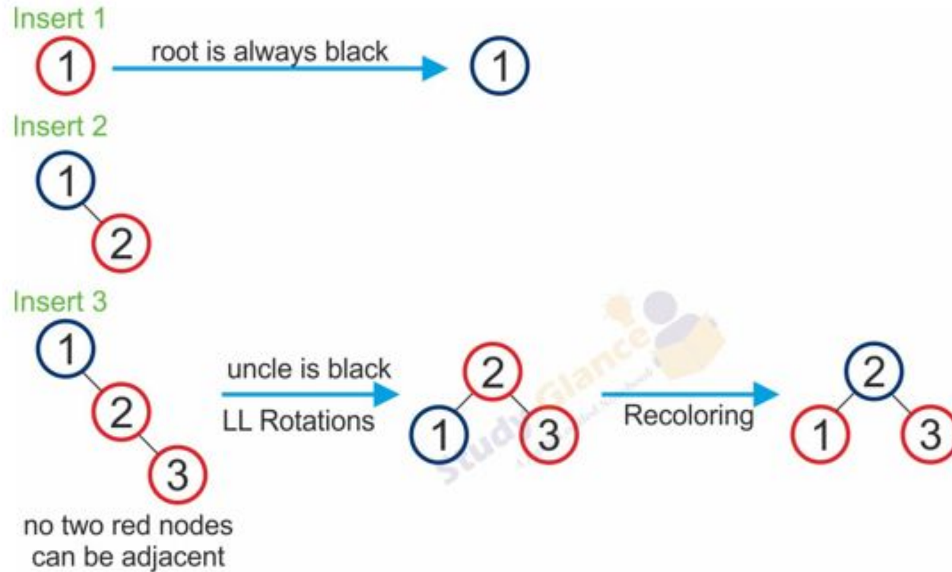


- Rotation



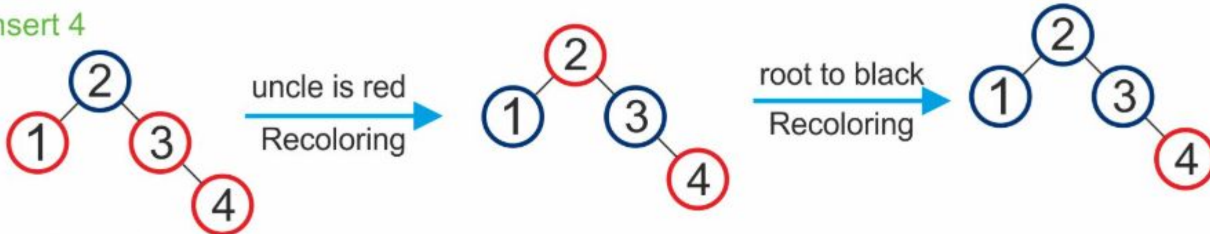
https://www.cs.nthu.edu.tw/~wkhon/algo08-tutorials/tutorial-redblack.pdf

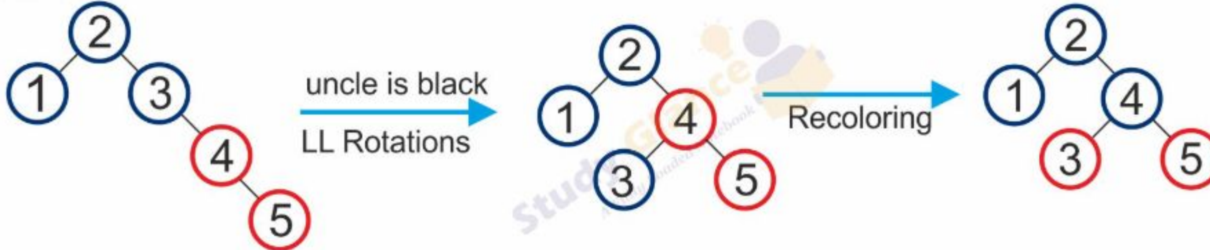
Red-Black (Insertion)



https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

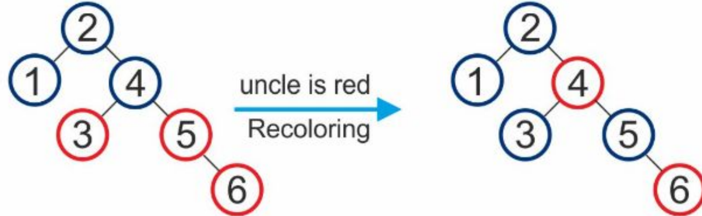
Red-Black (Insertion)

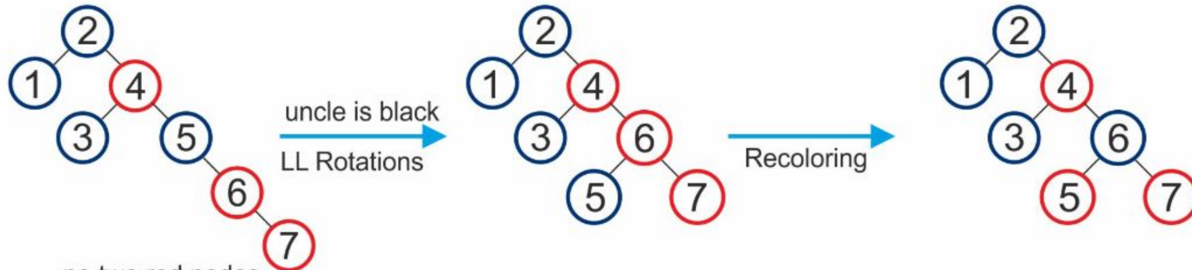




https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

Red-Black (Insertion)

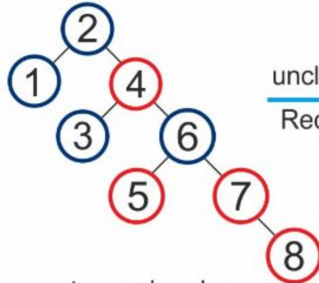




https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

Red-Black (Insertion)

Insert 8

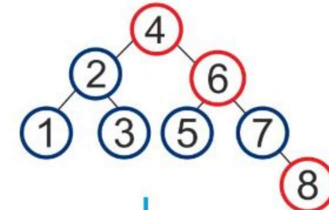


no two red nodes
can be adjacent

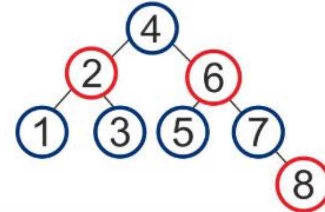
uncle is red
Recoloring

no two red nodes
can be adjacent

uncle is black
LL Rotations



Recoloring



https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

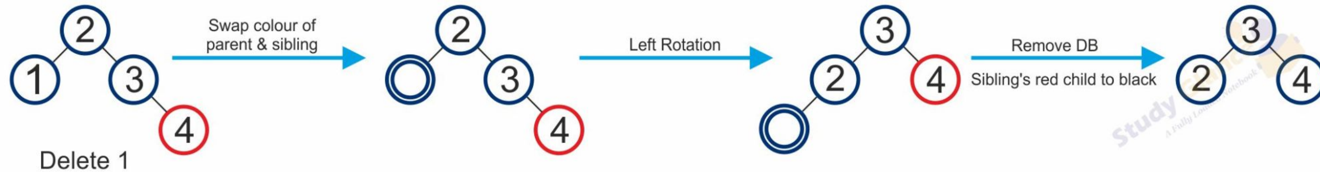
Red-Black (Deletion)

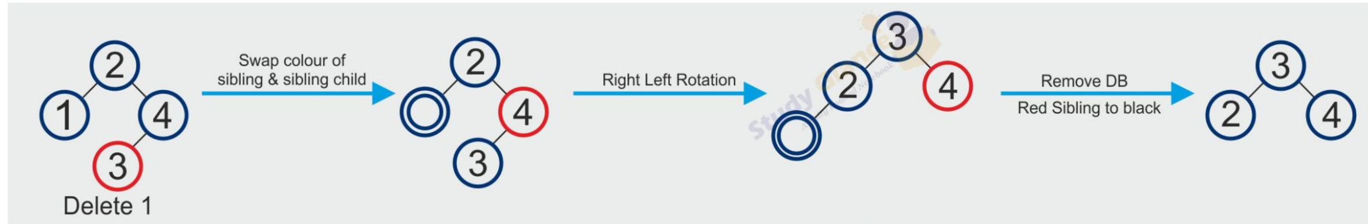
https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

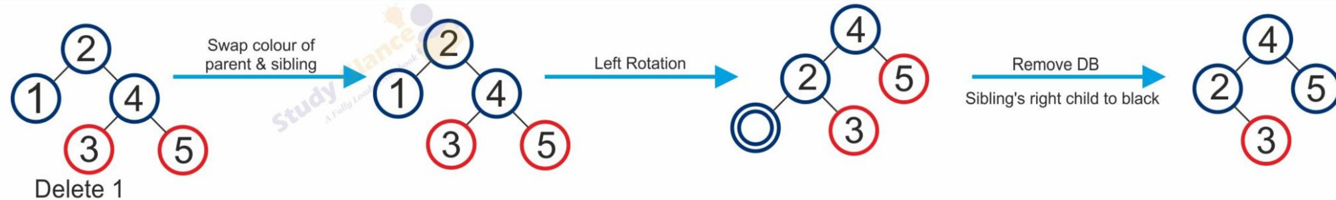
Red-Black (Deletion)

https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

Red-Black (Deletion)







https://studyglance.in/ds/display.php?tno=27&topic=Red-Black-Tree

Red-Black (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/trees/red-black/RedBlackTree.ts

Red-Black (In Action)

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Red-Black (Parallel Algorithms)

- Tufts COMP 150-08 Final Project (xuezhaokun.github.io)
- public:myassignment1.pdf (yorku.ca)
- A concurrent red-black tree — ScienceDirect

Red-Black Trees in File Systems (Strengths and Challenges)

Strengths:

- **Efficient Metadata Management**: Ideal for organizing file names, directories, and extents.
- **Dynamic Updates**: O(log n) performance for insertions, deletions, and lookups.
- **Broad Use**: Proven in Linux Kernel (e.g., scheduling, memory management).
- **Ordered Data**: Ensures sorted directories and quick range queries.

Challenges:

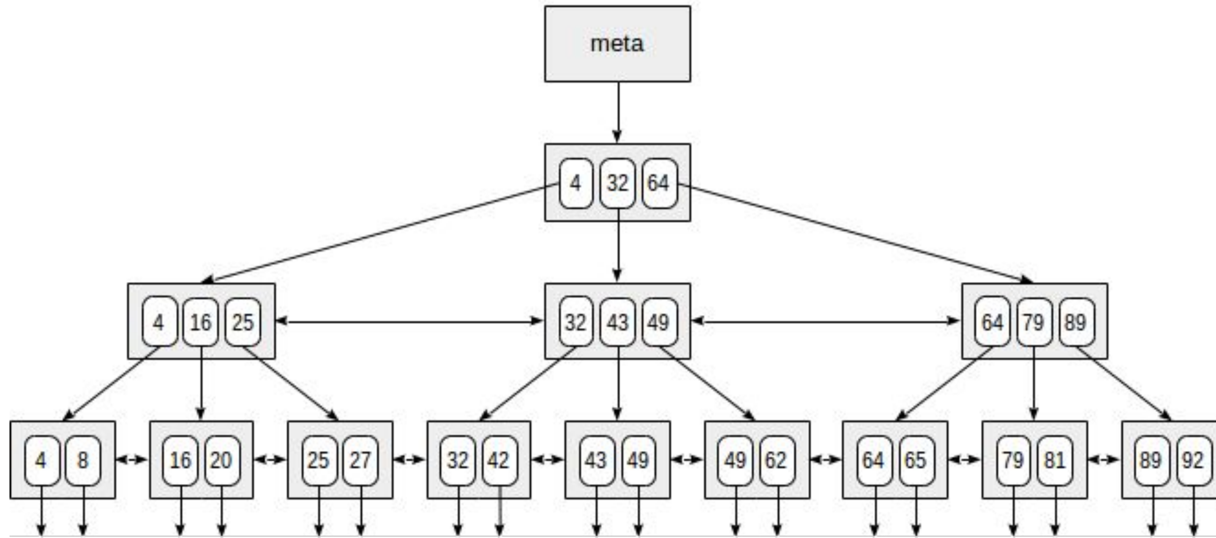
- **Cache Locality**: Scattered nodes lead to poor cache performance for large datasets.
- **Overhead for Insert/Delete**: Rotations and pointer updates can be costly.
- **Scaling Limitations**: Less efficient than B-Trees for disk-based storage due to higher I/O.
- **Complexity**: Implementing and maintaining Red-Black Trees can be challenging.

https://grantjenks.com/docs/sortedcontainers/performance.html

Red-Black (Time and Space Complexity)

Operation	Red-Black Tree	AVL Tree	BST (Average)	BST (Worst)
Insertion	O(log n)	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(log n)	O(n)

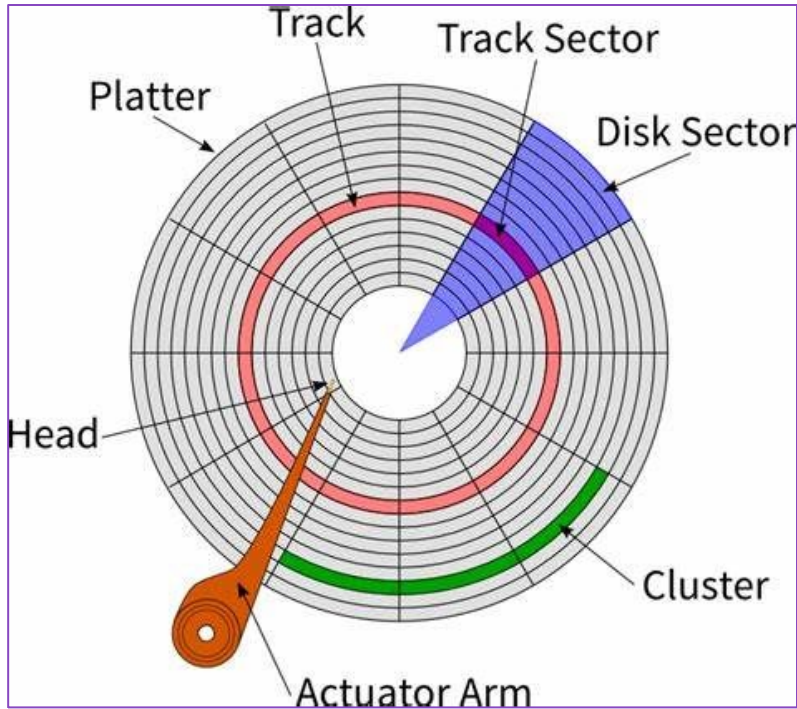
B-Tree



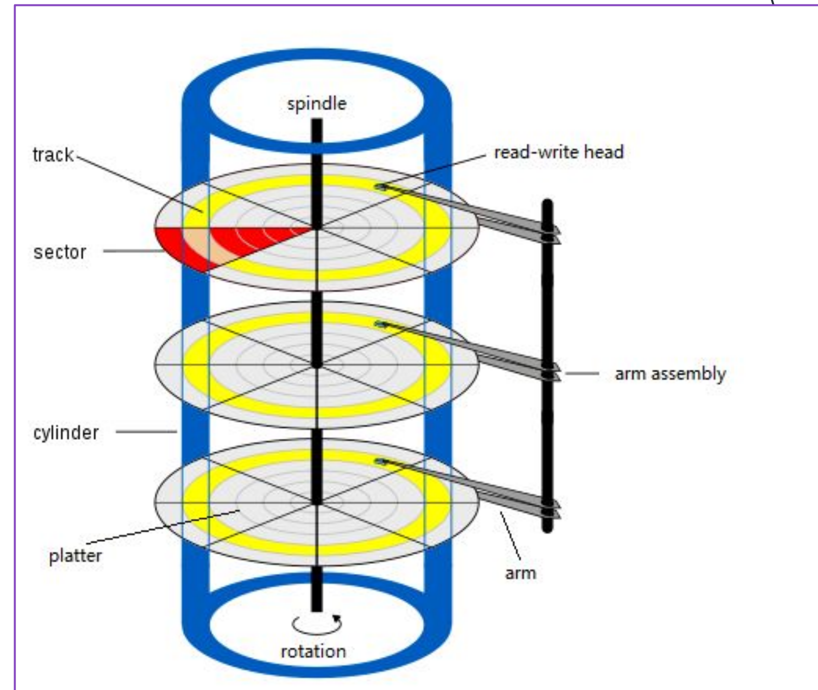
https://habr.com/ru/companies/postgrespro/articles/443284/

B-trees aren’t limited to two children per node.

B-Tree (Disk-Friendly)

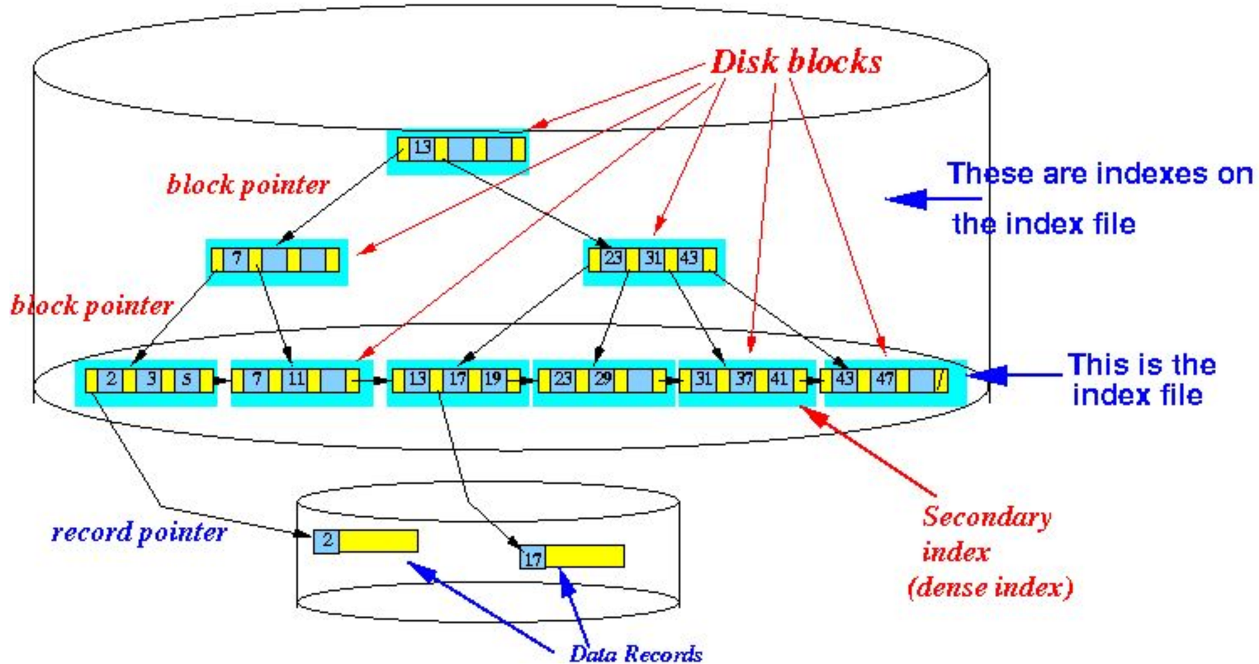


https://blog.cyber5w.com/hard-disk-investigation



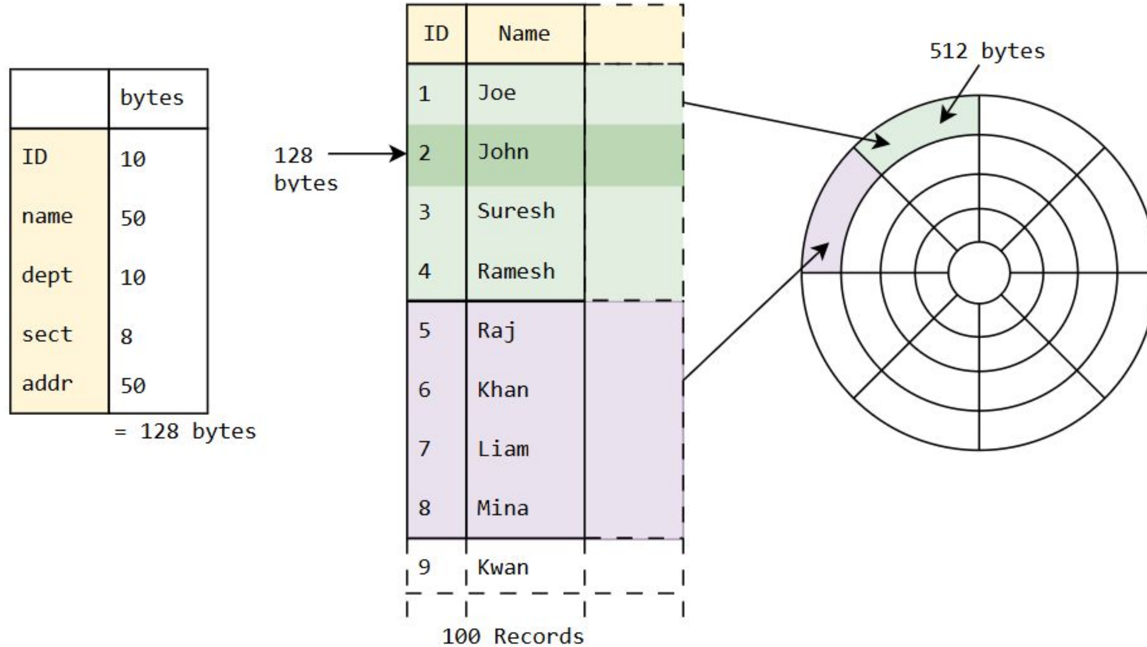
https://www.partitionwizard.com/help/what-is-chs.html

B-Tree (Disk-Friendly)



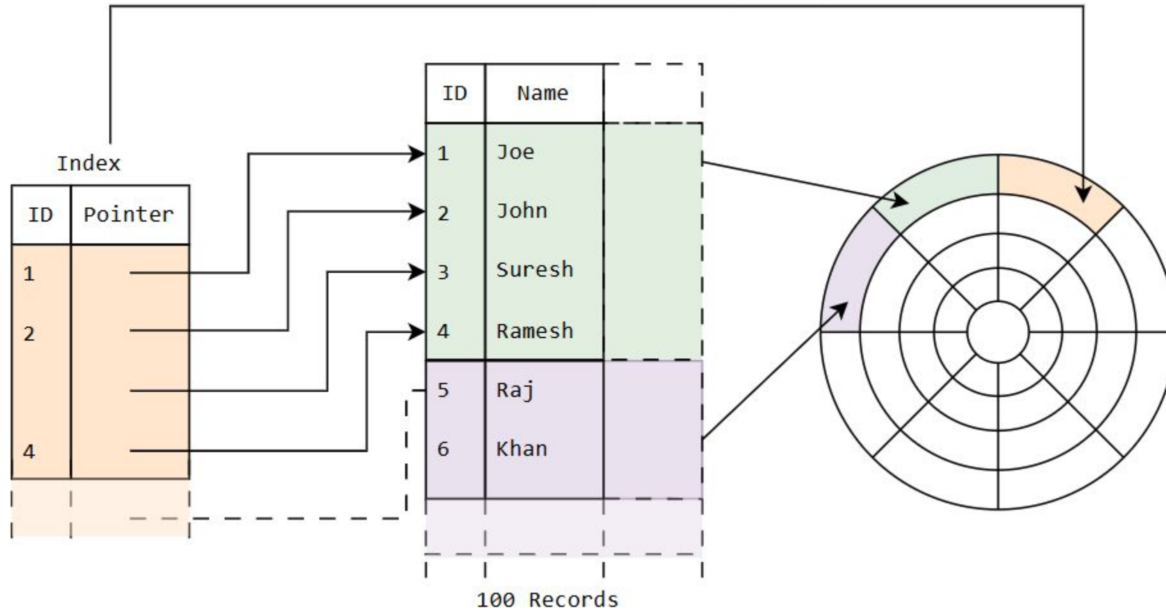
https://www.cs.emory.edu/~cheung/Courses/554/Syllabus/3-index/B-tree=delete1.html

B-Tree (Disk-Friendly)



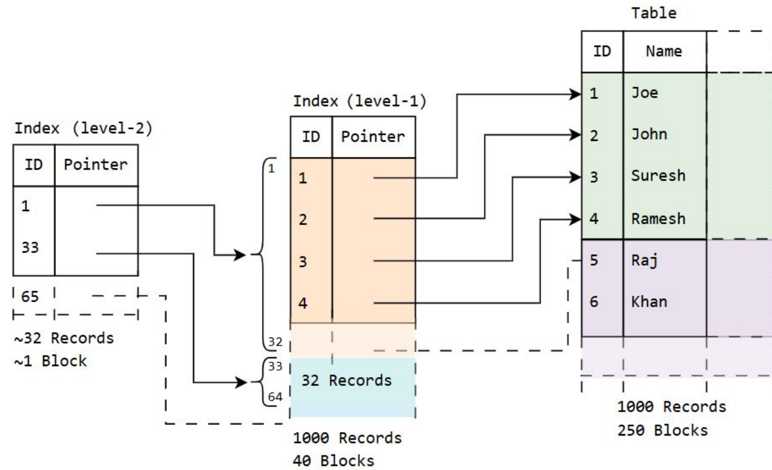
https://www.pyblog.xyz/b-tree

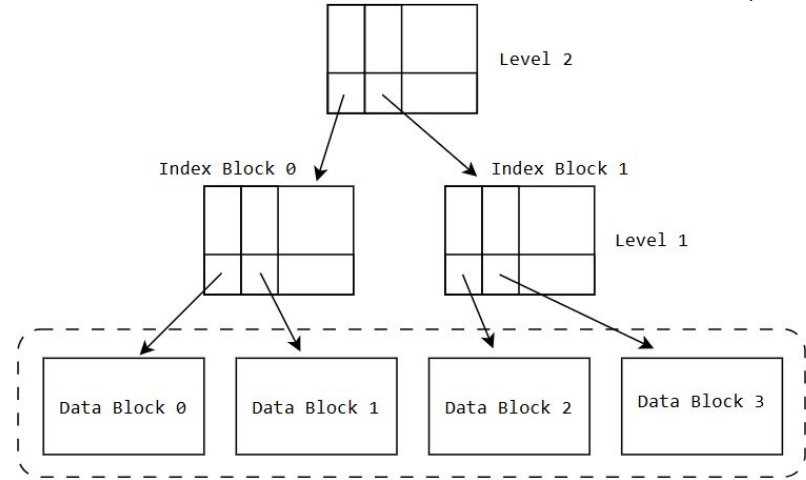
B-Tree (Indexing)



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

B-Tree (Multi-Level Indexing)



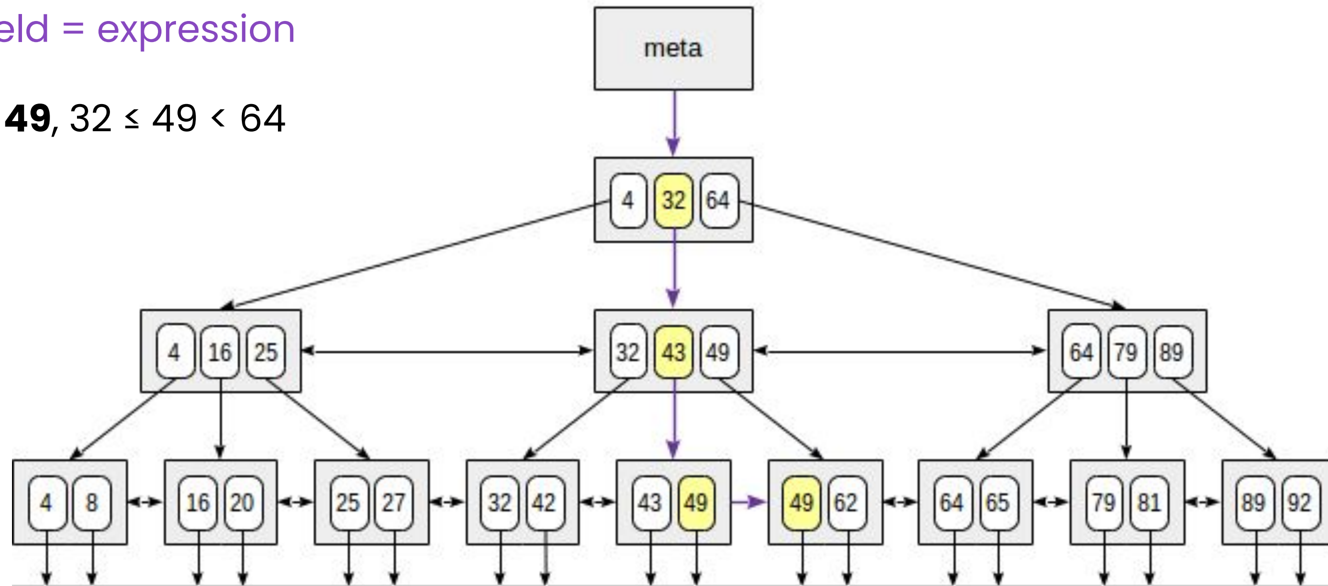


https://www.pyblog.xyz/b-tree

B-Tree (Search by equality)

indexed-field = expression

Search for **49**, 32 ≤ 49 < 64

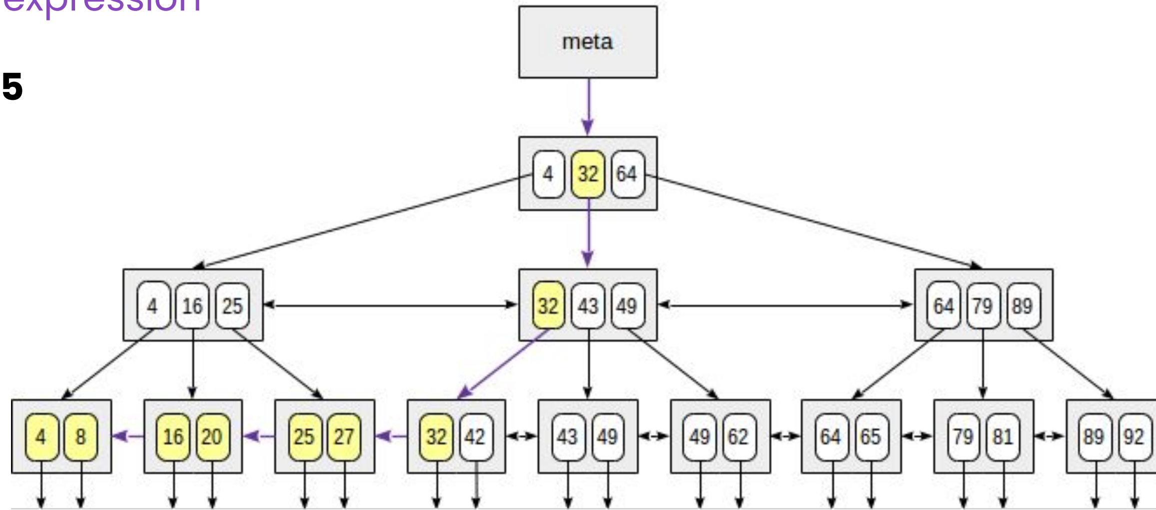


https://habr.com/ru/companies/postgrespro/articles/443284/

B-Tree (Search by inequality)

indexed-field ≤ expression or
indexed-field ≥ expression

Search for **n ≤ 35**

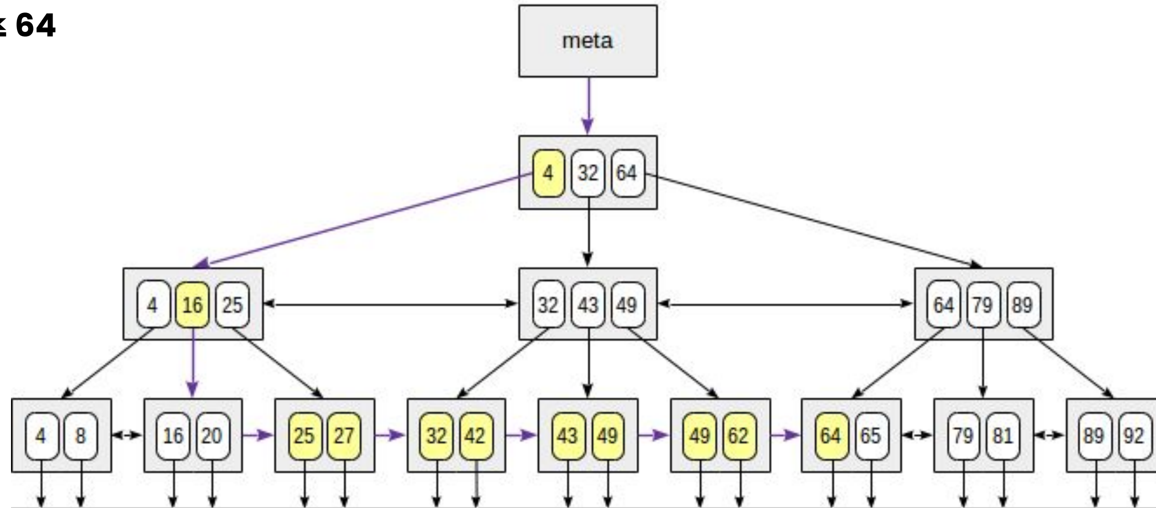


https://habr.com/ru/companies/postgrespro/articles/443284/

B-Tree (Search by range)

expression1 ≤ indexed-field ≤ expression2

Search for **23 ≤ n ≤ 64**



https://habr.com/ru/companies/postgrespro/articles/443284/

B-Tree (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/trees/b-tree/BTree.ts

B-Tree (In Action)

https://www.cs.usfca.edu/~galles/visualization/BTree.html

B- Tree (Variants)

- **Copy-on-write B-Trees** are structured like B-Trees, but their nodes are immutable and are not updated in place. Instead, pages are copied, updated, and written to new locations.
- **Lazy B-Trees** reduce the number of I/O requests from subsequent same-node writes by buffering updates to nodes.
- **Bw-Trees** separate B-Tree nodes into several smaller parts that are written in an append-only manner. This reduces costs of small writes by batching updates to the different nodes together.
- **Cache-oblivious B-Trees** allow treating on-disk data structures in a way that is very similar to how we build in-memory ones.

B-Tree (Real World Implementations)

- google/btree: BTree provides a simple, ordered, in-memory data structure for Go programs. (github.com)
- btree package — github.com/google/btree — Go Packages
- postgres/btree_bit.c at master · postgres/postgres · GitHub
- sqlite/btree.h at b609a79f4a9a0666d51db6f0f7c05e90308ed8c2 · sqlite/sqlite (github.com)
- sqlite/test_btree.c at 37d4ec86bfa78c31732132b7729b8ce0e47da891 · sqlite/sqlite (github.com)
- ntfstool/btree.cpp at master · thewhiteninja/ntfstool (github.com)
- BTreeMap in std::collections — Rust (rust-lang.org)

B-Tree in File Systems (Strengths and Challenges)

Strengths:

- **Optimized for Disk Access**: Minimizes I/O operations with wide nodes, ideal for large-scale storage.
- **Scalability**: Efficient for managing large directories, file indices, and extents.
- **Balanced Structure**: Maintains O(log n) performance for search, insert, and delete.
- **Cache-Friendly**: Nodes are stored contiguously, improving cache locality and disk performance.
- **Widely Used**: Standard in modern file systems like NTFS, XFS, HFS+, and APFS for metadata organization.

Challenges:

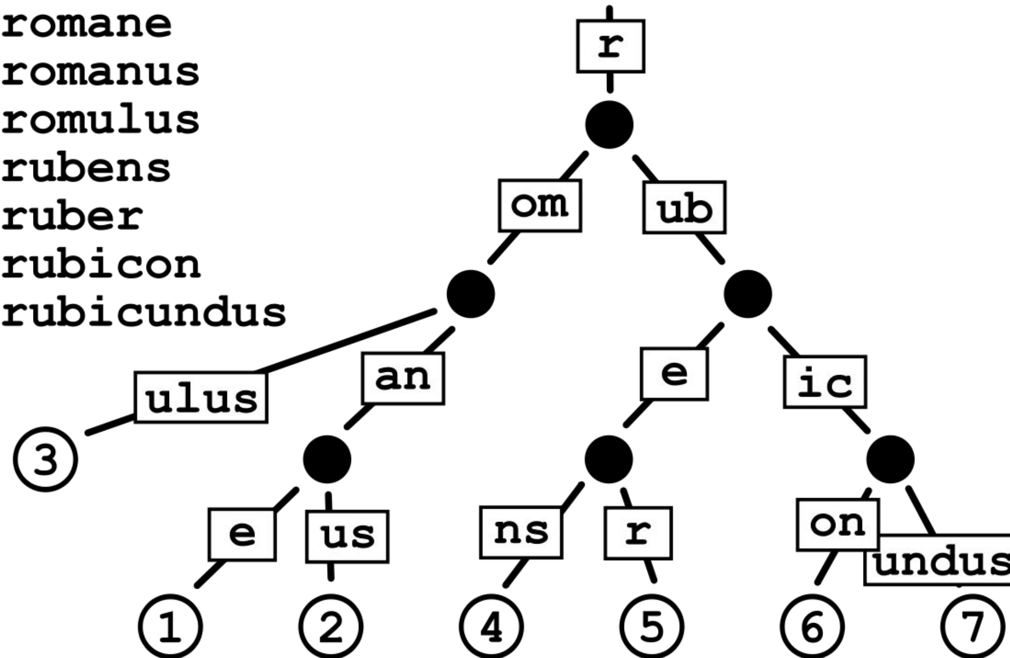
- **Complex Implementation**: Requires careful management of node splits and merges.
- **Memory Overhead**: Larger node sizes can lead to higher in-memory storage needs.
- **Performance on Small Datasets**: Overkill for simple or small file systems where binary trees suffice.
- **Write Overhead**: Insertions and deletions can involve multiple node updates, especially in disk-based systems.

https://en.algorithmica.org/hpc/data-structures/b-tree/

B-Tree (Time and Space Complexity)

Operation	B-Tree	AVL Tree	Red-Black Tree	BST (Average)	BST (Worst)
Insertion	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

Trie



https://github.com/Devinterview-io/trie-data-structure-interview-questions/blob/main/README.md

Trie (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/trees/trie/Trie.ts

Trie (File Search Bar)

https://helabenkhalfallah.com/2024/10/11/trees-in-data-structures-more-than-just-wood/

Trie (In Action)

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Trie (Time and Space Complexity)

Operation	Trie
Insertion	O(m)
Deletion	O(m)
Search	O(m)

m: Represents the length of the word or prefix being inserted, deleted, or searched.

Trees (Key takeaways)

Tree Type	Insertion	Deletion	Search	Space Complexity
Binary Search Tree (BST)	O(log n) (average), O(n) (worst)	O(log n) (average), O(n) (worst)	O(log n) (average), O(n) (worst)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	O(n)
Red-Black Tree	O(log n)	O(log n)	O(log n)	O(n)
B-Tree	O(log n)	O(log n)	O(log n)	O(n)
Trie (Prefix Tree)	O(m)	O(m)	O(m)	(See note)

Note: The space complexity of a Trie can vary depending on the number of words and the degree of prefix sharing. In the worst case, it can be O(N * M), where N is the number of words and M is the average length of a word.

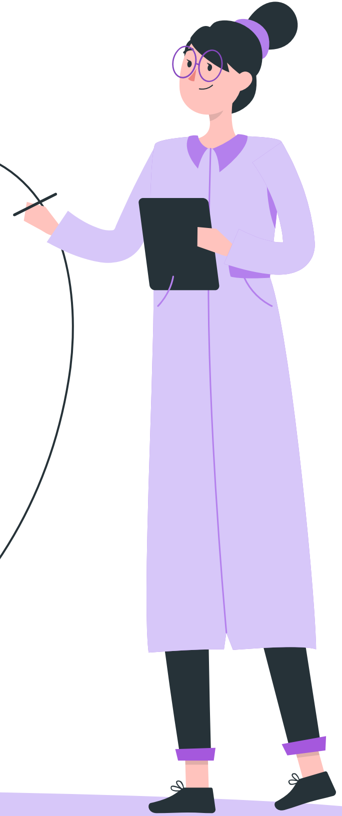
Trees (Key takeaways)

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.

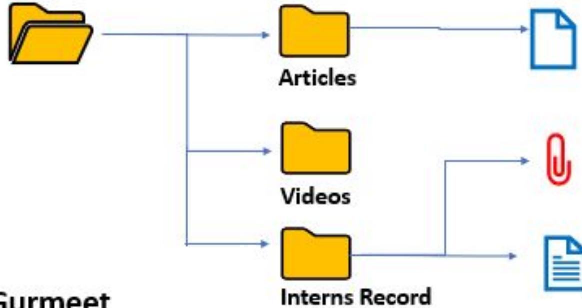
2

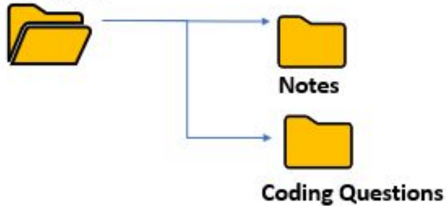
Heaps

The challenge of priority



Folders hierarchy System in Computers





https://takeuforward.org/binary-tree/application-of-tree-data-structure/

The use case

Efficiently identify and manage **the most critical snapshot or backup** (e.g., oldest or largest) **without** sorting all elements.

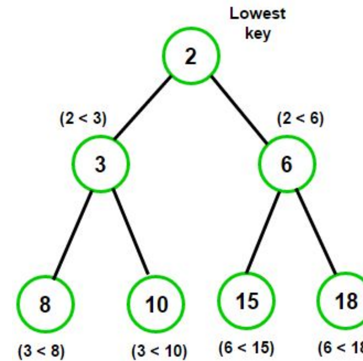
https://helabenkhalfallah.com/2024/10/28/yet-another-way-to-balance-bsts-the-treaps-approach/

Array (Limitations)

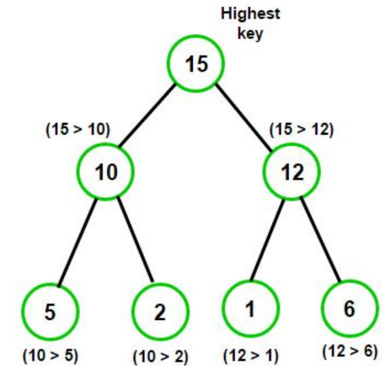
Limitations
No inherent priority management, requiring sorting.
Searching for the highest or lowest priority takes O(n).
Maintaining a sorted array incurs O(n) insertion/deletion costs.
Fixed size or resizing overhead makes it less dynamic.
Reorganization is required after every update.
Sorting requires additional memory for large datasets.
Poor adaptability to dynamic changes in priorities.

Heaps (Rules)

- **Complete Binary Tree**: A heap is always a complete binary tree, meaning all levels are fully filled except possibly the last, which is filled from left to right.
- **Heap Property**:
  - **Max-Heap**: Parent node is always greater than or equal to its children.
  - **Min-Heap**: Parent node is always smaller than or equal to its children.
- **Efficient Access**: The root node always contains the highest (Max-Heap) or lowest (Min-Heap) priority element.
- **Insertion**: Add a new element at the next available position (maintaining completeness), then "bubble up" to restore the heap property.
- **Deletion**: Remove the root node (highest/lowest priority), replace it with the last node, and "bubble down" to restore the heap property.
- **Dynamic Updates**: Supports efficient insertions and deletions while maintaining the heap structure in **O(log n)** time.



Min Heap
(Parent key is less than or equal to (≤) the child key)



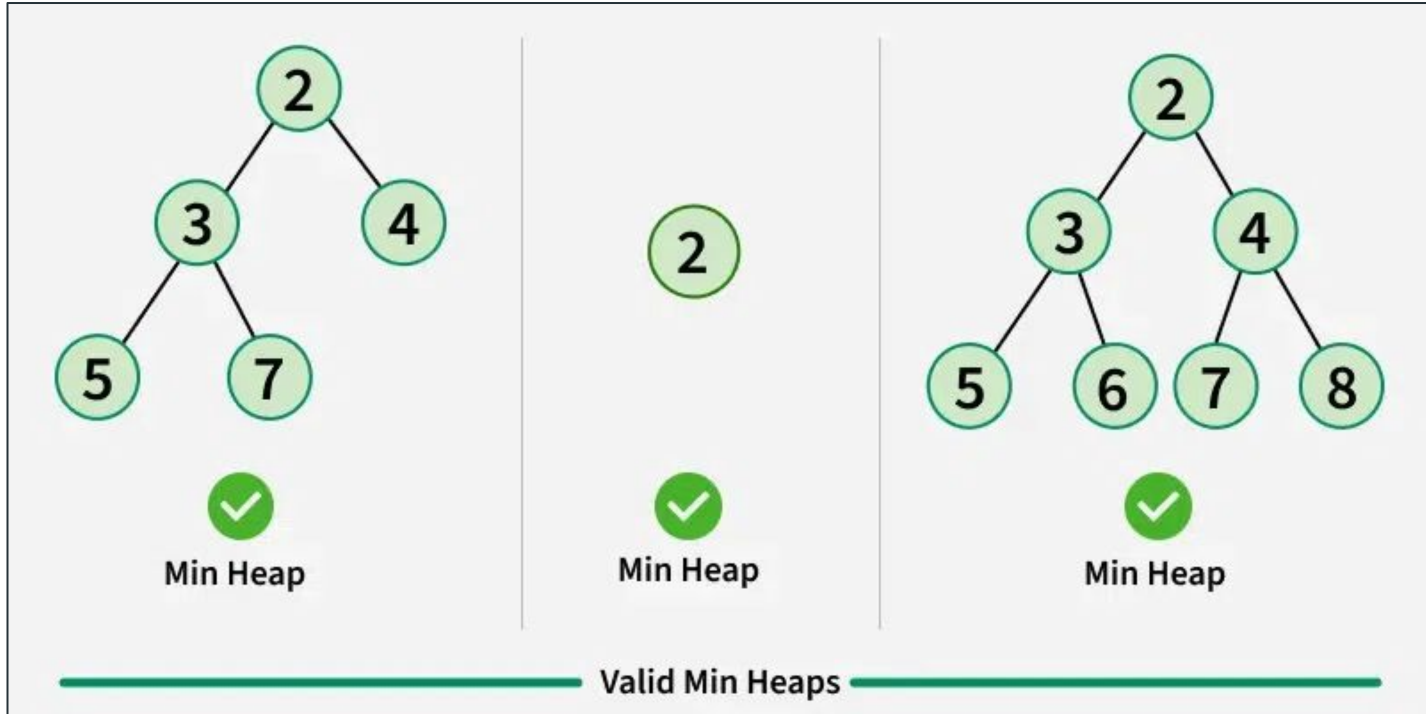
Max Heap
(Parent key is greater than or equal to (≥) the child key)

https://javascript.plainenglish.io/real-world-uses-cases-for-heaps-e57edbeb7da3

Heaps (Efficiency)

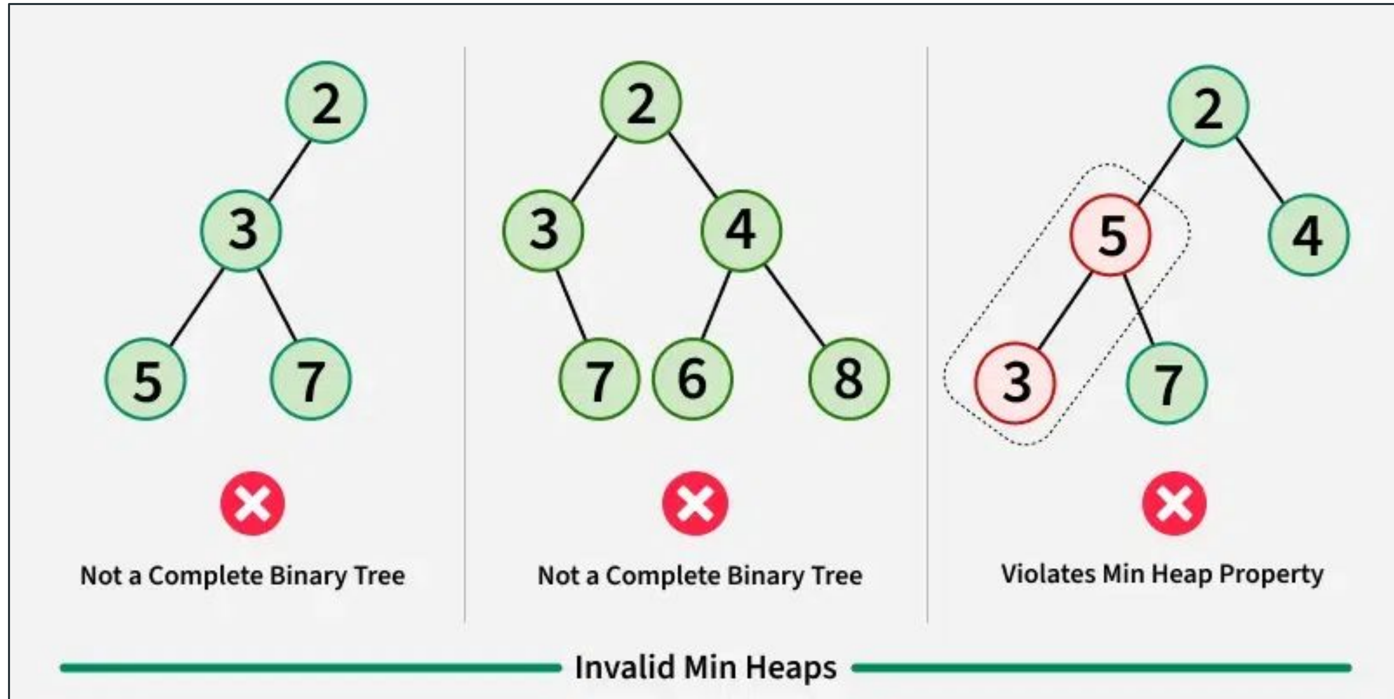
Efficiency Advantages
Partial sorting focuses only on the top priorities.
Quick access to the highest/lowest priority in O(1).
Insertions and deletions efficiently handled in O(log n).
Automatically maintains order without full reorganization.
Adapts dynamically to priority changes with minimal overhead.
Scales efficiently with larger datasets.

Min-Heap (Valid)



https://www.geeksforgeeks.org/heap-data-structure/

Min-Heap (Invalid)

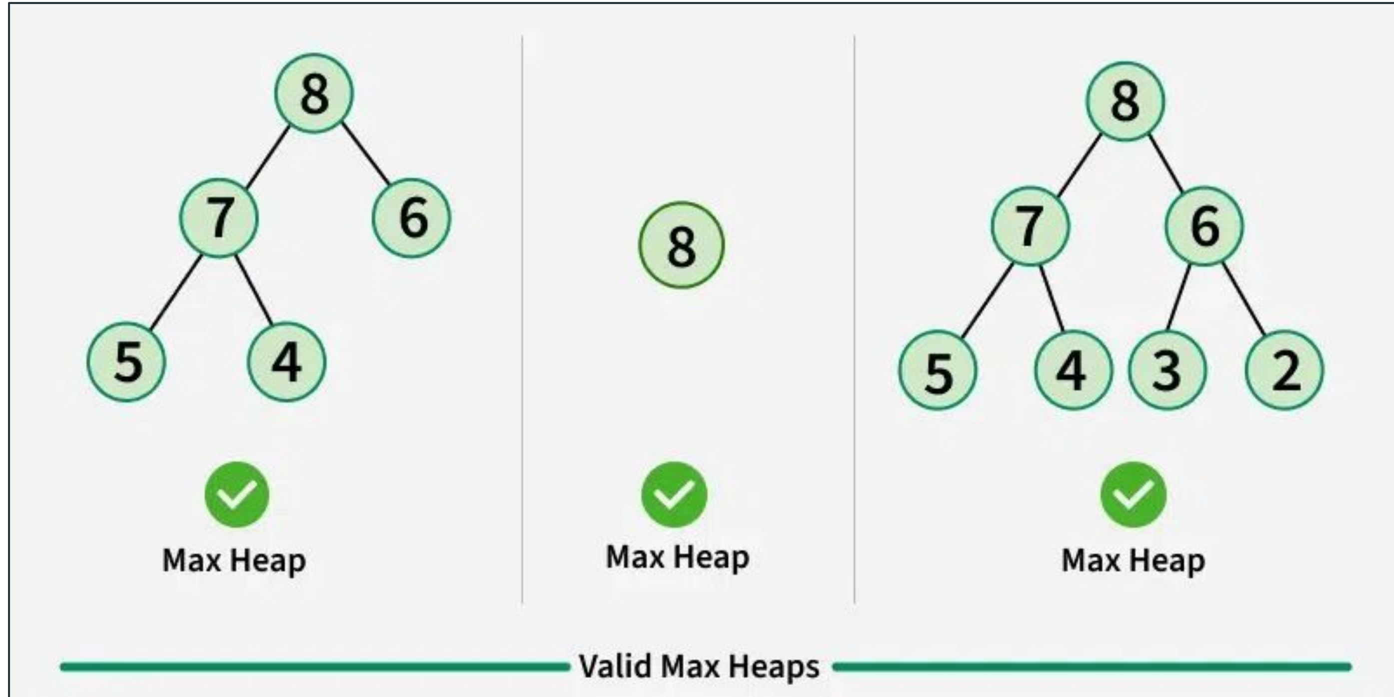


https://www.geeksforgeeks.org/heap-data-structure/

Min-Heap (JavaScript)

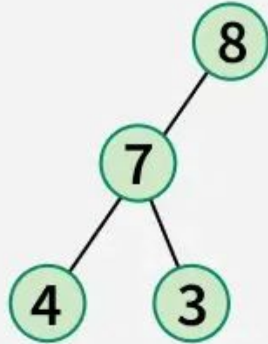
https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/heaps/MinHeap.ts

Max-Heap (Valid)

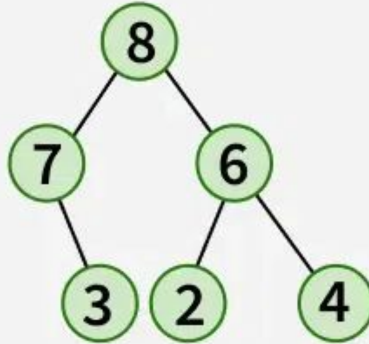


https://www.geeksforgeeks.org/heap-data-structure/

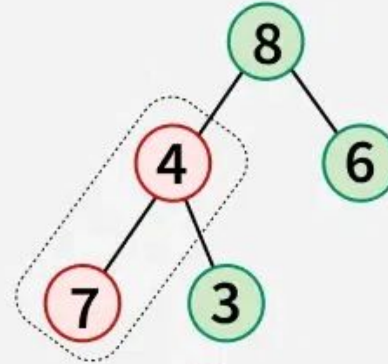
Max-Heap (Invalid)



Not a Complete Binary Tree



Not a Complete Binary Tree



Violates Max Heap Property

Invalid Max Heaps

https://www.geeksforgeeks.org/heap-data-structure/

Max-Heap (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/heaps/MaxHeap.ts

Heaps (In Action)

https://www.cs.usfca.edu/~galles/visualization/Heap.html

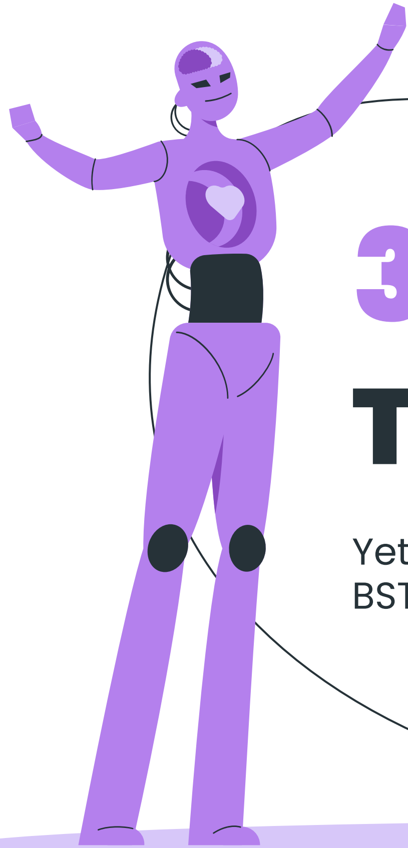
Heaps (Variants)

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)

Real-life Practical Applications

- **Load Balancing:**
  - Priority Queue of Servers.
  - Efficient Task Assignment.
  - Dynamic Updates.
- **Finding Similar Documents:**
  - Calculate Similarity.
  - Store in a Heap.
  - Efficient Retrieval: top 10 most similar documents.

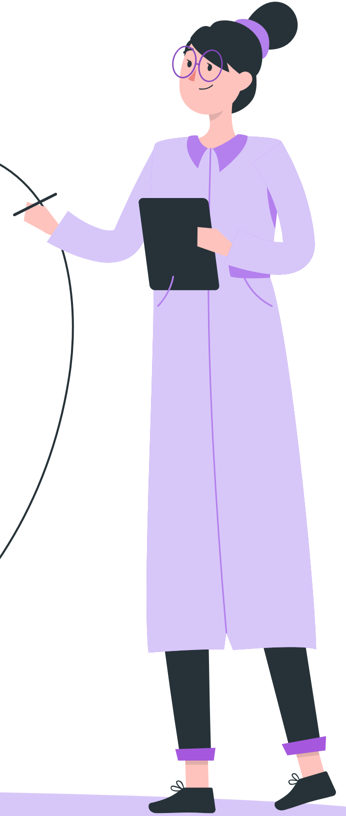
https://helabenkhalfallah.com/2024/10/28/yet-another-way-to-balance-bsts-the-treaps-approach/



3

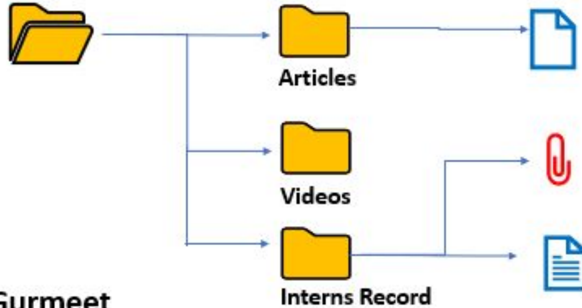
Treaps

Yet Another Way to Balance BST

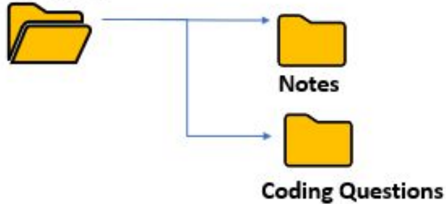


Folders hierarchy System in Computers

takeuforward



Gurmeet



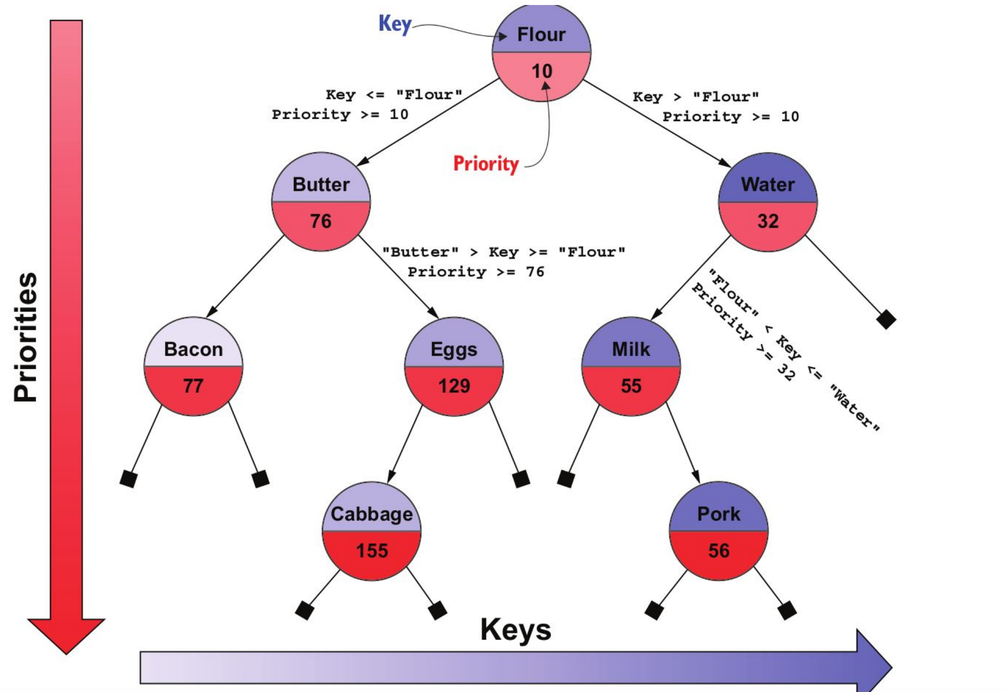
https://takeuforward.org/binary-tree/application-of-tree-data-structure/

The use case

Dynamically organize snapshots or backups **to retrieve and manage the Top-K most critical items while maintaining order and balance**.

https://helabenkhalfallah.com/2024/10/28/yet-another-way-to-balance-bsts-the-treaps-approach/

Treap (Properties)



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

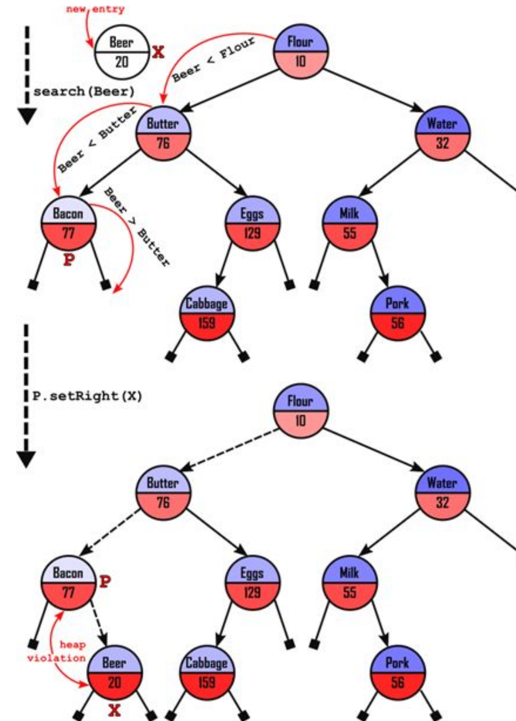
A Treap is a **hybrid data structure** that combines a Binary Search Tree (BST) and a Heap.

It maintains **two key properties**:

- Binary Search Tree (BST) Property
- Heap Property (Min-Heap or Max-Heap)

Treaps (Operations)

- **Insertion**:
  - Insert the node as in a BST (based on the key).
  - Assign a **random priority** to the node.
  - Perform **rotations** (similar to AVL trees) to **restore the heap property**.
- **Deletion**:
  - Find the node as in a BST.
  - Perform **rotations** to push the node to a leaf position while maintaining the heap property.
  - Remove the node.
- **Search**:
  - Search works exactly as in a BST (O(log n) expected time).



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

Treaps (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/treaps/Treap.ts

Treaps (In Action)

https://tjkendev.github.io/bst-visualization/treap/index.html

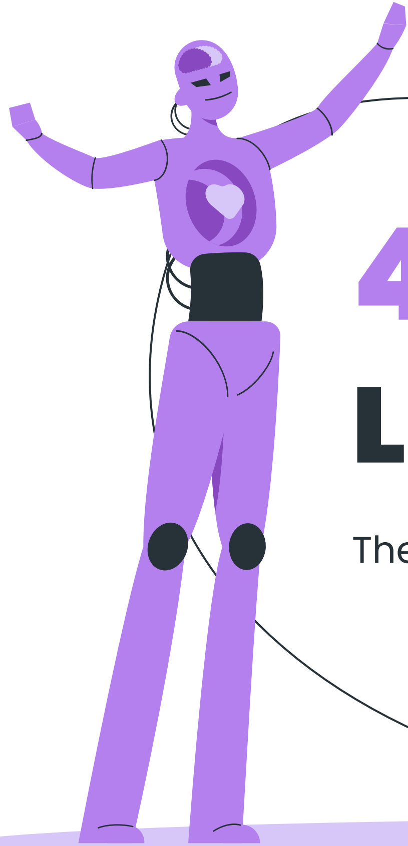
Treaps (Time and Space Complexity)

Operation	Treap (Average)	BST (Average)	BST (Worst)	AVL Tree	Red-Black Tree	Heap
Insert	O(log n)	O(log n)	O(n)	O(log n)	O(log n)	O(log n)
Top/Peek	O(1)	O(log n)	O(n)	O(log n)	O(log n)	O(1)
Remove	O(log n)	O(log n)	O(n)	O(log n)	O(log n)	O(log n)
Contains	O(log n)	O(log n)	O(n)	O(log n)	O(log n)	–
Update Priority	O(log n)	–	–	–	–	–
Min/Max	O(log n)	O(log n)	O(n)	O(log n)	O(log n)	O(1)
Space	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)

Real-life Practical Applications

- Ranked Retrieval: Top-K Elements
- In-memory Storage: Dynamic Dictionaries
- Text Editors: Efficient String Manipulation

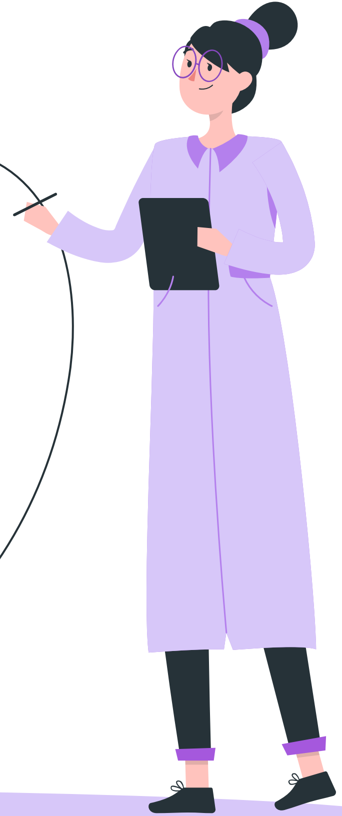
https://helabenkhalfallah.com/2024/10/28/yet-another-way-to-balance-bsts-the-treaps-approach/



4

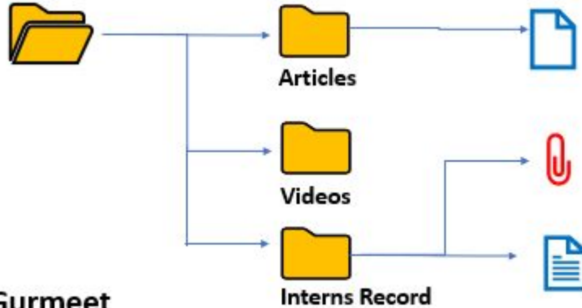
LRU & LFU

The magic of Caching

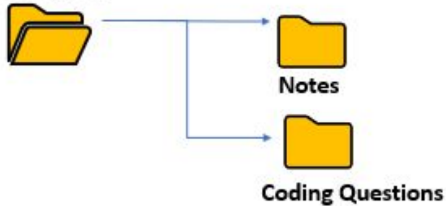


Folders hierarchy System in Computers

takeuforward



Gurmeet



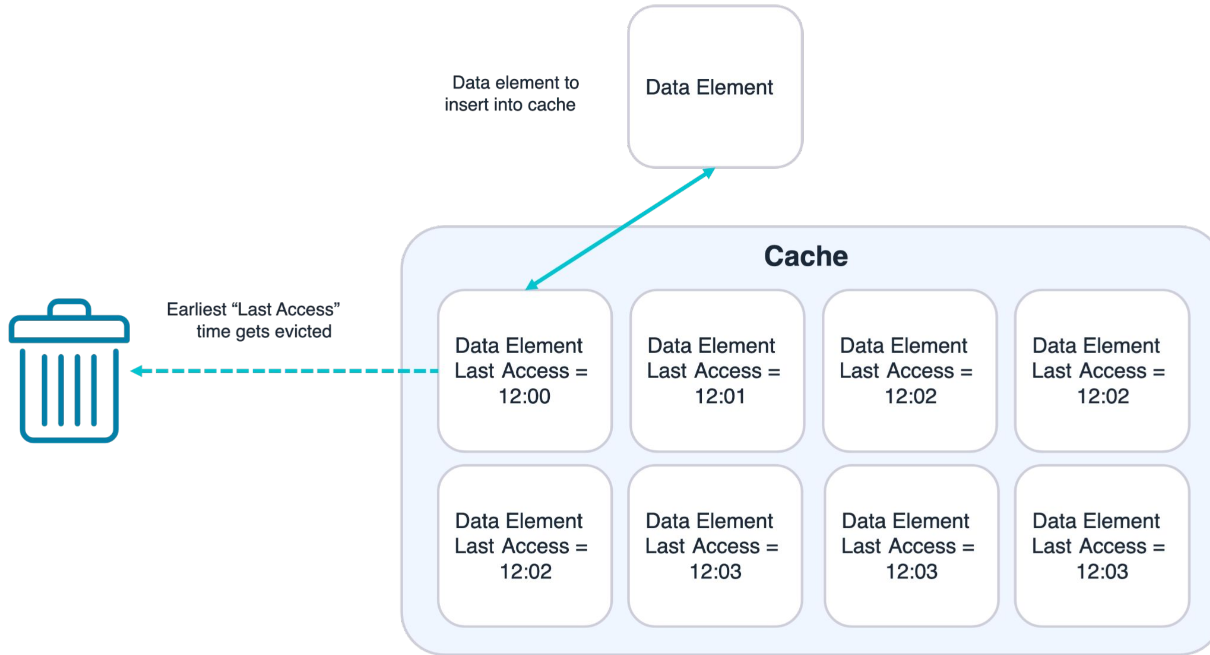
https://takeuforward.org/binary-tree/application-of-tree-data-structure/

The use case

Easy access to **recently** or **frequently** used files.

https://helabenkhalfallah.com/2024/10/28/yet-another-way-to-balance-bsts-the-treaps-approach/

Least Recently Used (LRU)

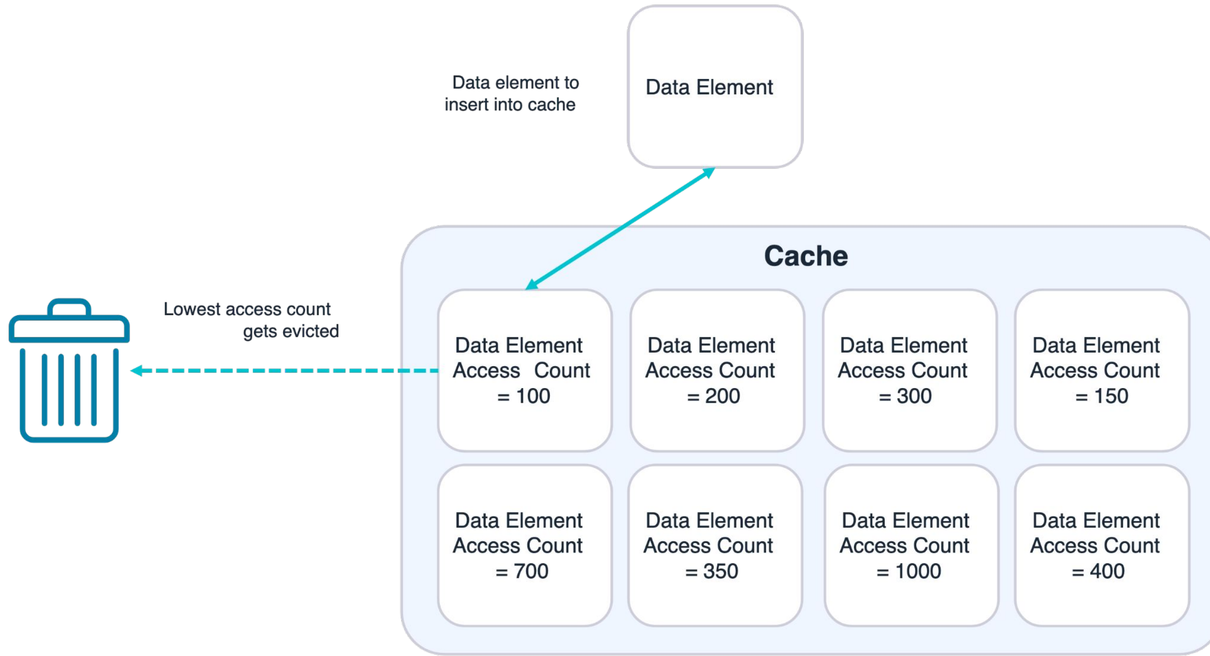


https://hazelcast.com/glossary/caching-strategies/

LRU (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/cache/LRU.ts

Least Frequently Used (LFU)



https://hazelcast.com/glossary/caching-strategies/

LFU (JavaScript)

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/cache/LFU.ts

LFU & LRU (In Action)

https://ericgopak.github.io/operating-system-concepts/#

LRU Real-Life Applications

- **Apache Solr**: Solr uses LRU caching to store frequently queried search results, improving response times by reducing redundant computations.
- **Chromium Disk Cache**: Chromium's network stack employs LRU to manage cached web resources, keeping frequently accessed data accessible to speed up page load times.
- **Mozilla SCCache**: SCCache uses LRU caching for storing compilation results, which helps minimize repeated builds and enhances build speed efficiency.
- **RocksDB Block Cache**: RocksDB uses an LRU-based block cache to store frequently accessed database blocks, reducing disk I/O for faster read operations in high-performance applications.
- **HBase Block Cache**: HBase uses an LRU block cache for in-memory storage of frequently accessed data blocks, optimizing read performance and reducing reliance on disk access.
- **MySQL InnoDB Buffer Pool**: uses LRU caching to optimize memory usage for frequently accessed database pages, minimizing disk I/O by retaining recently used data in memory.
- **Redis Eviction Policies**: Redis offers multiple LRU-based eviction policies to manage memory effectively, helping users retain frequently accessed keys while freeing up space for new data.
- **Linux Kernel LRU Cache**: The Linux kernel uses LRU caching in memory management to keep frequently accessed pages available in memory, reducing page faults and improving overall system performance.
- **Memcached LRU**: Memcached employs an LRU cache to retain "hot" data in memory, removing older data to ensure fast access to frequently requested items.

https://helabenkhalfallah.com/2024/11/01/the-secret-life-of-caches-a-deep-dive-into-lru-and-lfu/#h-lru-in-action-real-life-applications

LFU Real-Life Applications

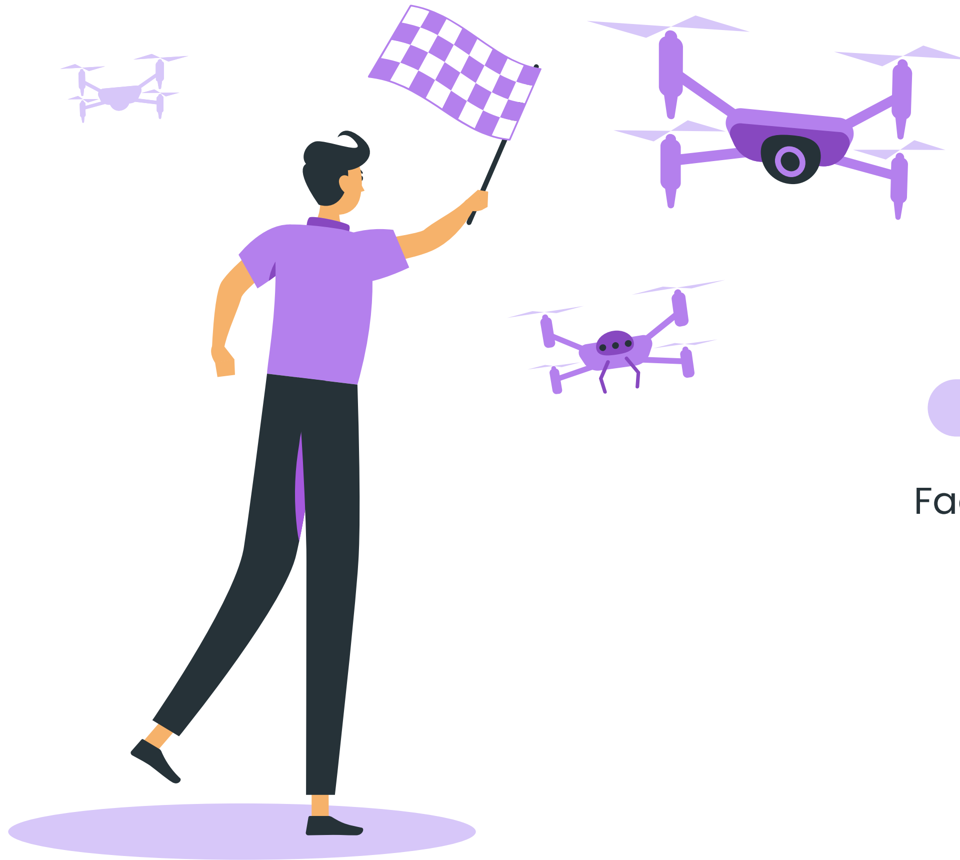
- **Varnish Massive Storage Engine (MSE)**: MSE is optimized for large-scale caching needs, such as video streaming and CDNs, managing up to 100+ TB of data per node. It minimizes fragmentation and employs an LFU strategy to prioritize frequently accessed content, maintaining high cache hit rates.
- **Redis**: Redis supports multiple eviction policies, including LFU, which keeps frequently accessed items while evicting less-used data. This policy is ideal for memory-limited scenarios with predictable access patterns.
- **Apache Pekko HTTP**: This caching feature uses a frequency-biased LFU cache for high-traffic APIs, evicting less-accessed items when at capacity. It also supports TTL and time-to-idle settings for time-based expiration, effectively reducing server load by caching popular requests.
- **Caffeine**: Caffeine, a Java caching library, includes LFU among its policies. Designed for high-performance applications, it provides flexible configurations to optimize LFU-based eviction, ensuring frequently accessed data remains readily available.

https://helabenkhalfallah.com/2024/11/01/the-secret-life-of-caches-a-deep-dive-into-lru-and-lfu/#h-lfu-in-action-real-life-applications

The Future of Caching

- **Adaptive and Hybrid Algorithms:**
  - **ARC (Adaptive Replacement Cache)**: ARC **combines LRU and LFU** by adjusting the size of each cache partition according to observed access patterns, making it ideal for unpredictable workloads.
  - **SLRU (Segmented LRU)**: By **segmenting the cache into multiple parts with distinct eviction policies**, SLRU offers more precise control over caching, prioritizing recent or frequently accessed data.
  - **Machine Learning-Based Caching**: Leveraging machine learning, these caches predict future data needs and proactively store items before they're requested, paving the way for more intelligent, predictive caching.
- **Caching in Distributed Systems:**
  - **Edge Caching**: By placing data close to users, **edge caching minimizes the time it takes to retrieve data in geographically distributed systems**, crucial for latency-sensitive applications like streaming and IoT.
  - **Two-level (Hot/Cold) Caching**: Separating frequently accessed (**hot**) from less-used (**cold**) data, this technique streamlines data retrieval for critical information, making it particularly useful in cloud storage and CDN architectures.

https://helabenkhalfallah.com/2024/11/01/the-secret-life-of-caches-a-deep-dive-into-lru-and-lfu/#h-the-future-of-caching-beyond-lru-and-lfu



Final Words

Factors to Consider & Key Takeaways

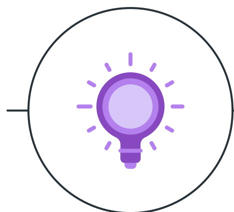
Factors to Consider

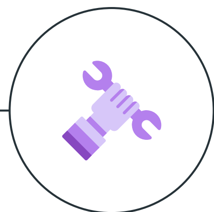
Data size

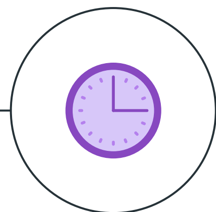
How much data will we be storing?

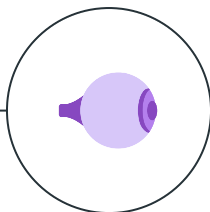
Search requirements

How important is fast search performance?









Frequency of updates

How often will we be adding, deleting, or updating data?

Storage medium

Will the data be stored in memory or on disk?

Key Takeaways

1

No one-size-fits-all

The best tree depends on the specific needs of our application.

2

Balance is key

Self-balancing trees like AVL and Red-Black trees offer consistent performance and prevent worst-case scenarios.

3

Specialized trees for specialized tasks

Tries excel at prefix-related operations, while B-trees are optimized for disk-based storage.

Books

Advanced Algorithms and Data Structures

Marcello La Rocca

Foreword by Luis Serrano

MANNING

O'REILLY

Database Internals

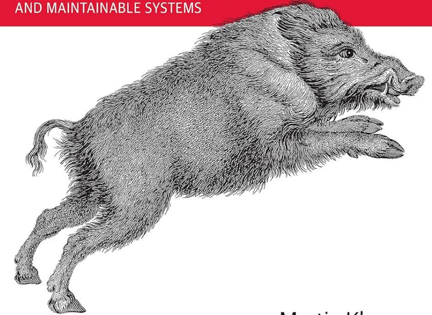
A Deep-Dive into How Distributed Data Systems Work

Alex Petrov

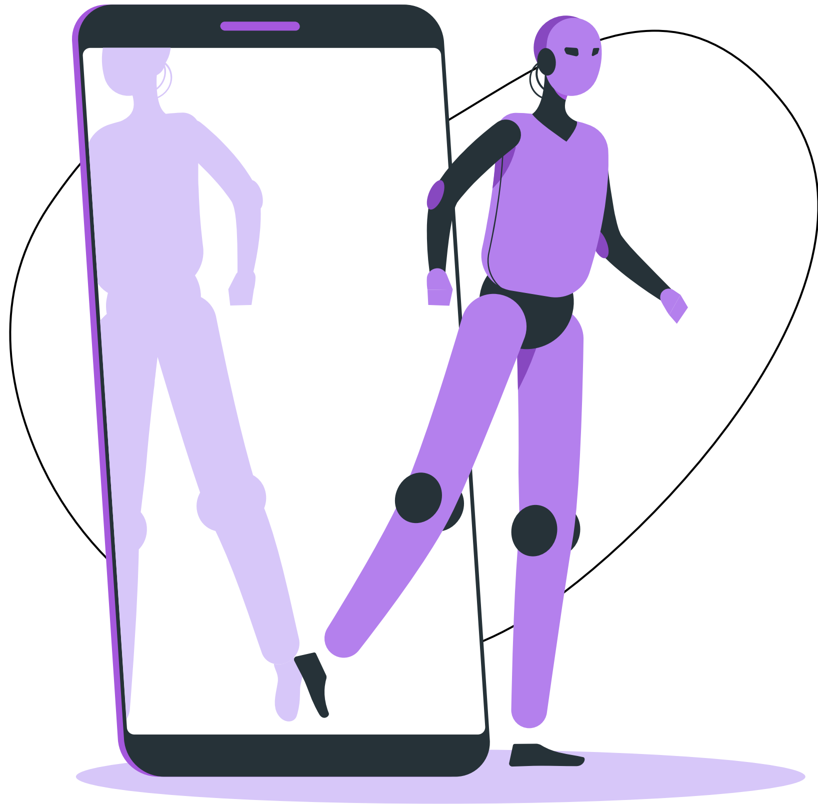
O'REILLY

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS



Martin Kleppmann



Thanks!

Do you have any questions?

- ★ **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/