# B-TREE. DATABASE INDEX

<https://www.baeldung.com/cs/b-trees-vs-btrees>

<https://www.geeksforgeeks.org/introduction-of-b-tree/>

<https://www.geeksforgeeks.org/difference-between-b-tree-and-b-tree/>

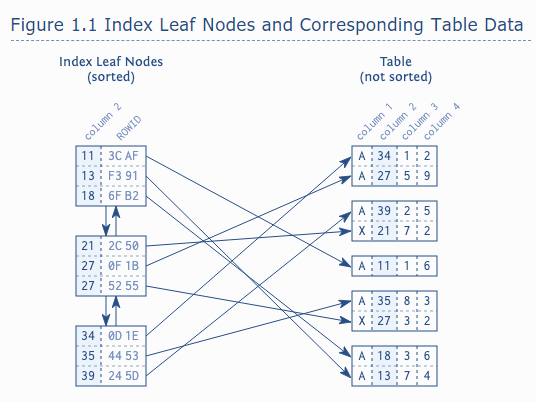
# INDEX STRUCTURE

* Block can be found by formula **block=(segment, track)**
* Block typical size is 512 byte
* **When we read or write data from/into disk we always speak in terms of block**
* Block has 512 bytes from 0 to 511
* Offset is an address of 1 byte of block
* Data can not be processed on a disk. It has to by brought to a main memory RAM
* In memory data stores in Data Structure and on a disk in DBMS
* **B-tree** – is a self-balancing m-way search tree where data stores in a sorted order (ascending) and operation insertion and deletion based on a special rules
* **B+ tree index has two differences compared to B-tree**

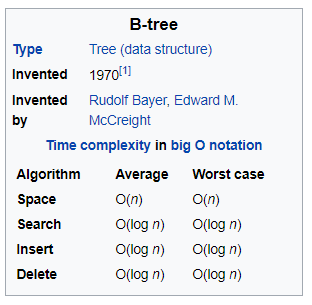
1)It stores data only in leaf nodes

2)Leaf nodes are present like linkedList (*doubly linked list*) and have connection between each other. The leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

* **A B+ tree index is considered a multi-level index(**I need to check, but it looks like DBMS uses B+-tree and NoSQL(MongoDB, for example) B-tree**)**
* It means, data in db index are in a sorted order, but table is not sorted (it only happens in cluster index)
* Deletion is a complicated operation for B-tree and has three cases
* Delete a node when there is no child
* Delete a node when there is one child
* Delete a node when there are few child

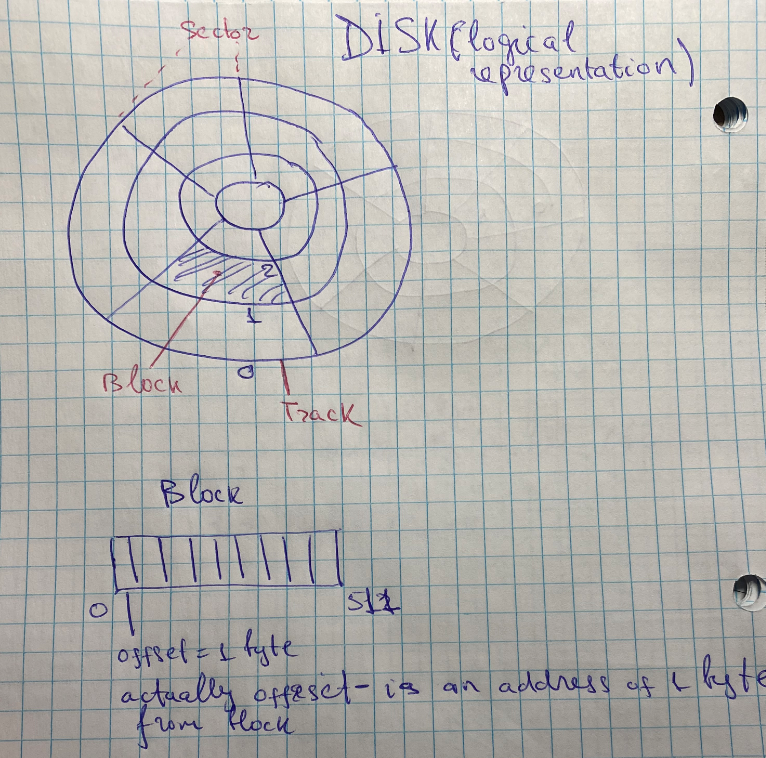






It is how a disk’s structure looks like. It is a logical division (not )

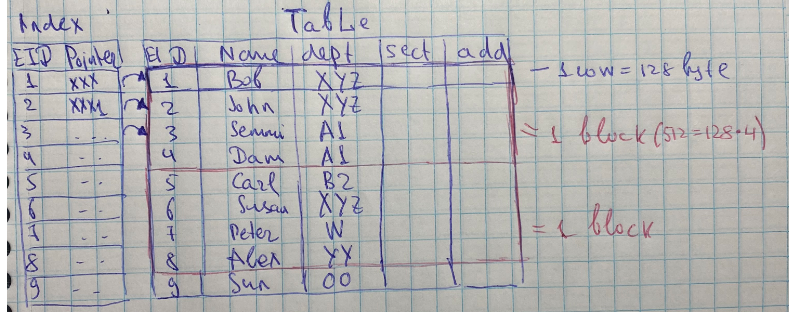
**Disk and block structure**



**Representation of table in term of disk**

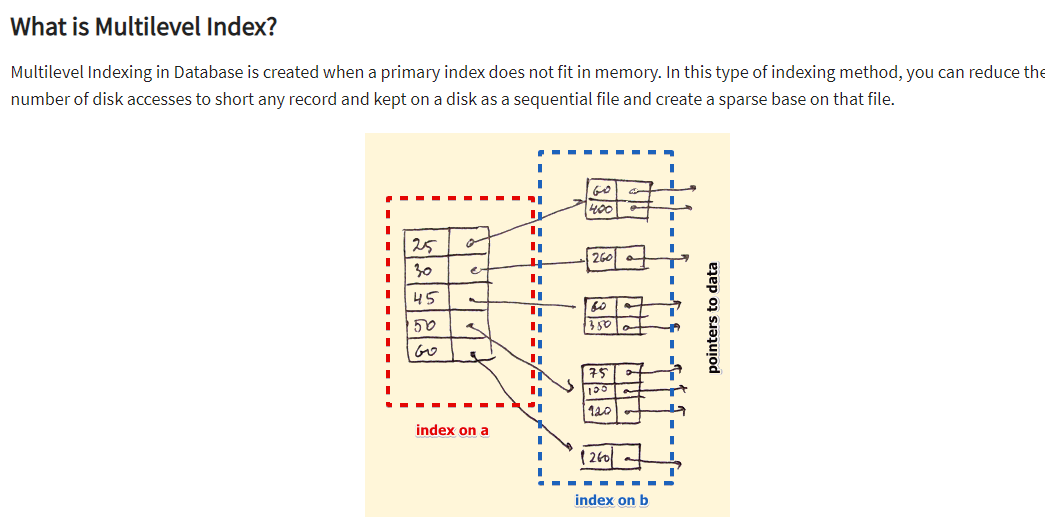


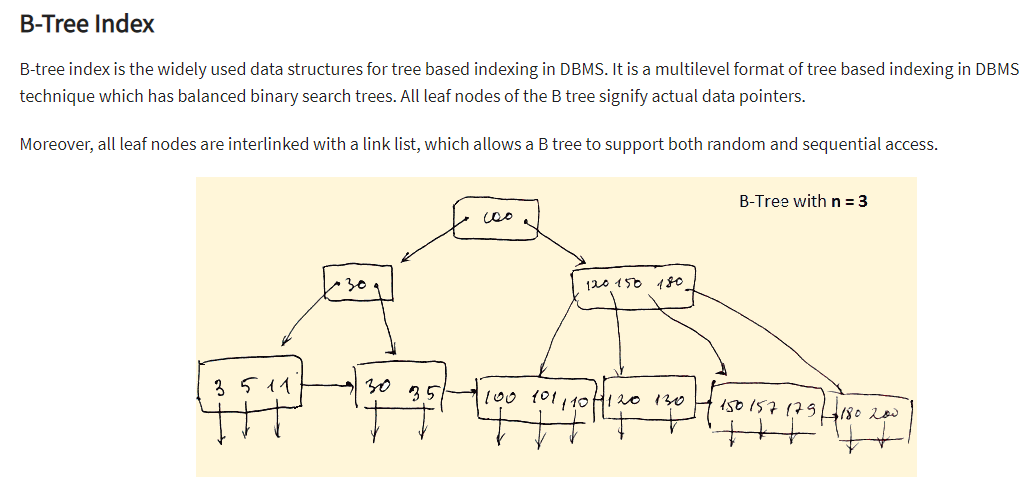
* How many block are in this table for 100 rows? = 25 blocks
* If need to find a particular row it needs to scan 25 blocks
* How many block are in this index for 100 rows? = 4 blocks



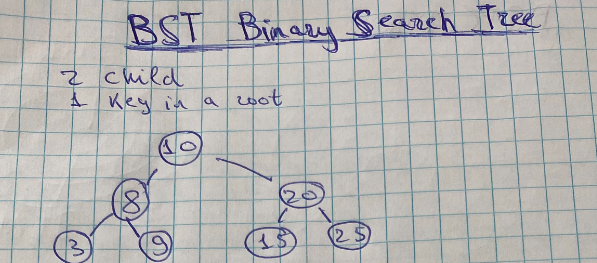
**Multilevel index**

Multi-level index is created when index is not fit into memory



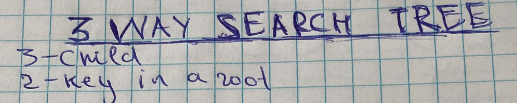
 **BST. Binary Search Tree**

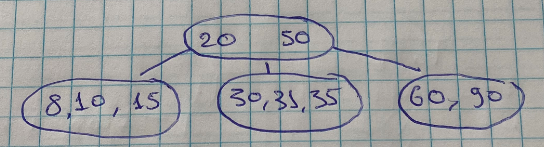
* 2 – child
* 2-1=1 – key in a root

****

**3-way Search Tree**

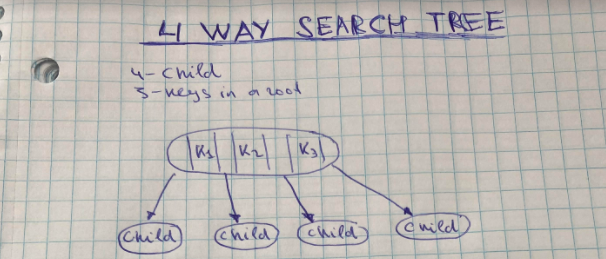
* 3 – child
* 3-1=2 – key in a root

****

****

**m-way Search Tree**

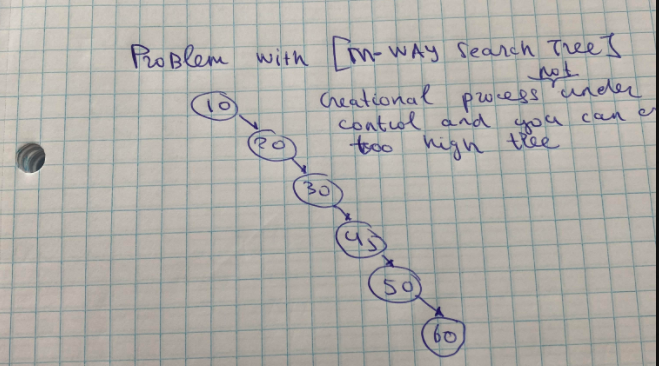
* 3 – child
* 3-1=2 – key in a root

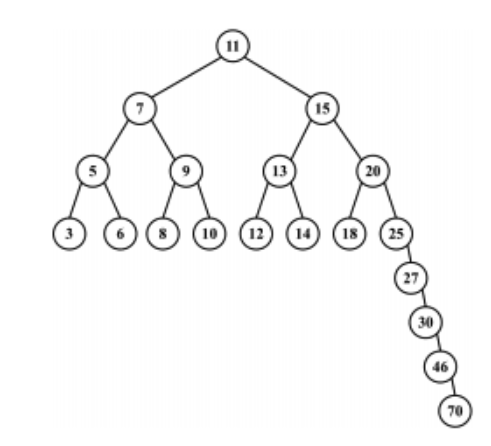
****

**Problem with [m-way search tree] for database index**

There is no a mechanism of controlling creation of nodes in a case of increasing/descending values. You may create too high tree and searching will be difficult. So, this happen for BST **when data is skewed**.

If the height of the tree is small, operations [insert], [delete] run fast whereas they are slow if the height of the tree is large. The problem with the ordinary binary search tree is that the height of the tree can, sometimes, be linear (n). Figure 1 shows the examples of binary search tree that are ‘unbalanced’ meaning that the height is large.





Note: the worst case here will be O(n), while for a self-balancing BST it will be O(logn)

**B-tree index**

**B-tree** – is a **self-balancing m-way search tree** where data stores in a sorted order (ascending) and operation insertion and deletion based on a special rules

**Self-balancing tree** (or height-balanced)– is a tree where height of tree is always minimum after insertion/deletion operation

Rules for b-tree:

1. All leaf nodes (nodes without child) are on the same level
2. [m] is a degree of tree
3. Is a balanced m-way searched tree
4. Creation is on bottom-up way
5. Node can have [m] child
6. [m-1] – the number of keys in one node

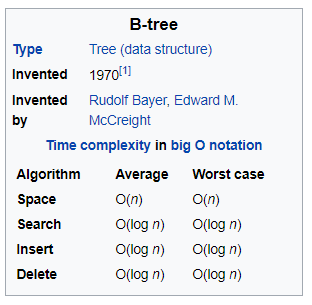
**Advantages**:

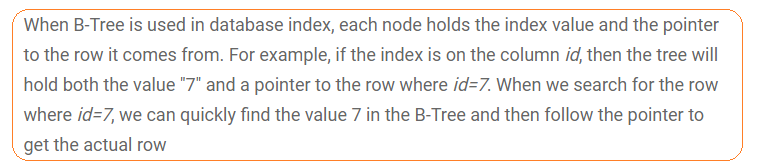
* Reduced number of I/O. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc ) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree

**Disadvantages**:

* In Btree, searching is not very efficient because the records are either stored in leaf or internal nodes.
* Deletion of internal nodes is very slow and a time-consuming process as we need to consider the child of the deleted key als

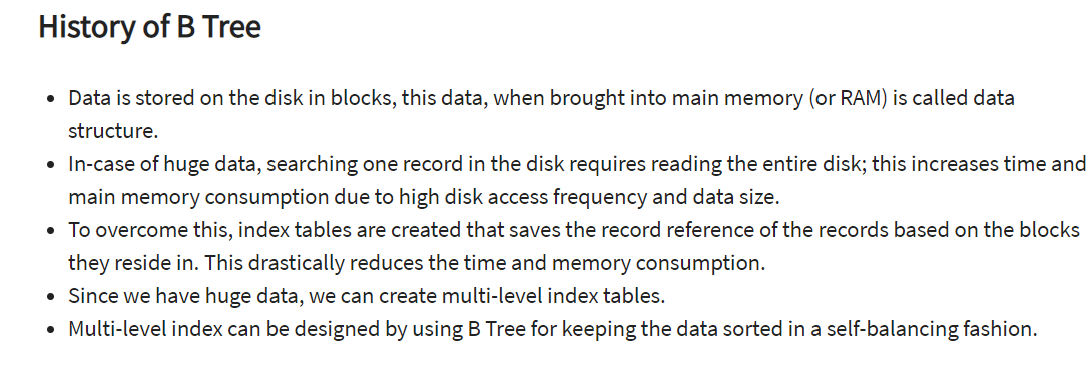




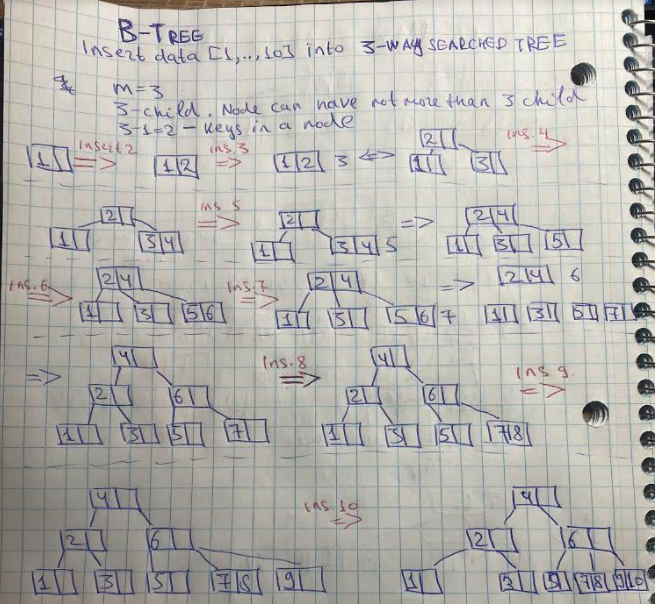


Deletion is a complicated operation for B-tree and has three cases

* Delete a node when there is no child
* Delete a node when there is one child
* Delete a node when there are few child



**Insertion into B-tree**



**B+-tree index**

B+ tree index is a similar structure that B-tree with two differences

1. It stores data only in leaf nodes
2. Leaf nodes are present like linkedList and have connection between each other. The leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. It helps do [full scan index] operation in one **pass**. B-tree index can not do this.

Note: B+ tree index is considered a multi-level index

Advantage of B+Tree:

* Does a [full scan index] operation in one pass
* Delete operation is faster
* Can contain duplicates compared to B-Tree

