B and B+ trees

Clustered and non-clustered indexes are concepts that describe how data is stored and accessed in a database, but B-trees (and B+ trees) are the data structures that actually implement these indexes.

Knowing B-trees gives you insight into how these indexes work under the hood. Understanding B-trees helps you understand why certain queries perform well or poorly based on the structure of the index. For example, how a B-tree's balanced nature affects search times or why range queries are efficient with B-tree indexes.

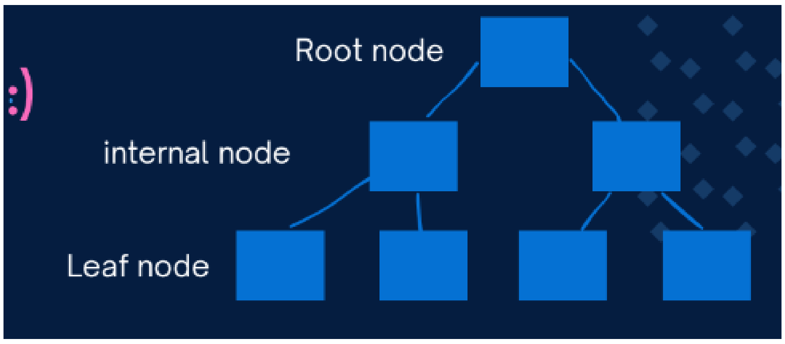
B-trees provide the balanced, efficient structure that makes these types of indexes performant, ensuring that operations like search, insert, delete, and update are done in logarithmic time (O(log n)). This makes B-tree indexing crucial for optimizing database queries, whether in clustered or non-clustered index scenarios.

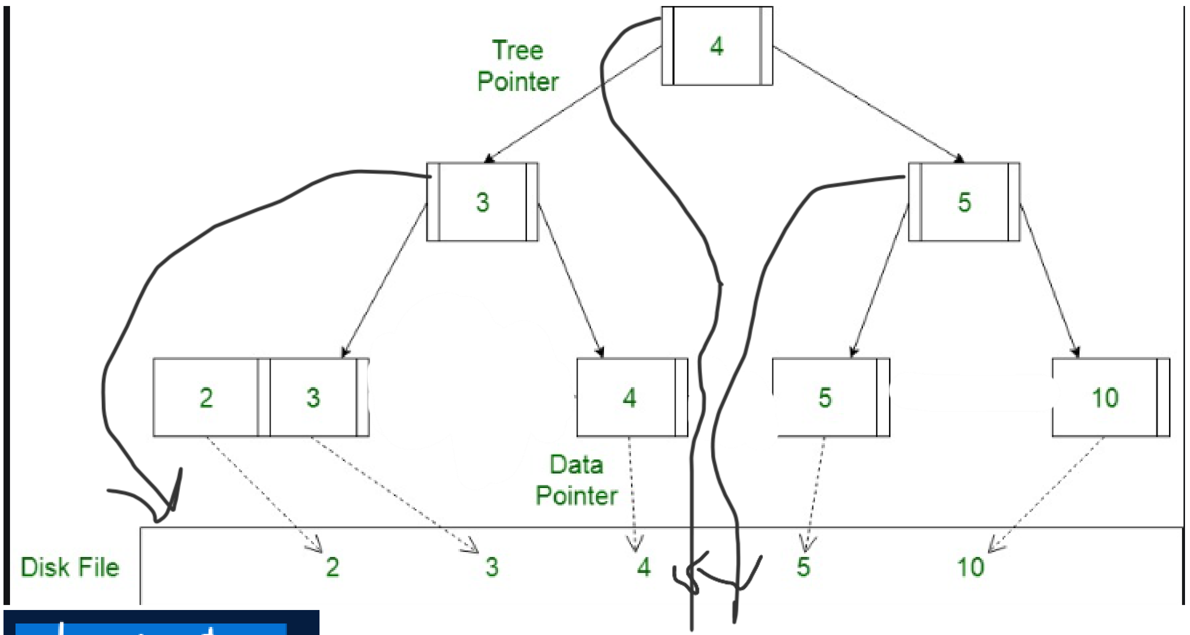
B trees

- B-trees are self-balancing tree data structures that maintain sorted data and allow searches, sequential access, insertions, and deletions in logarithmic time. All leaf nodes are at the same level.
- In B trees you can have minimum 2 children and max M children where M is the order of B tree.
- B-tree is a generalisation of Binary Search trees. In BST every node can have at most 2 children and only one key(value), but in B trees one node have M children at most and more than 1 key.
- Key are not data in B trees, they are just used for indexing. Data associated with each key is stored in DB and its reference is present in node corresponding to that key.
- The keys within a node are sorted in ascending order.

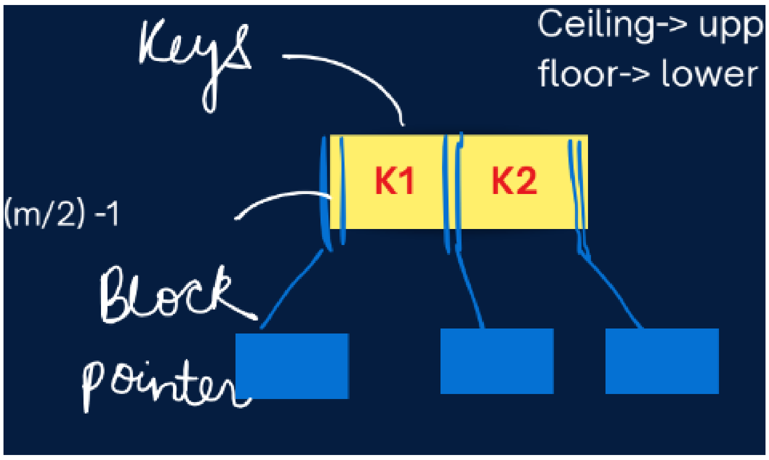
- The elements/keys in left of node key would be less than node key and the element in right would be greater than node key.

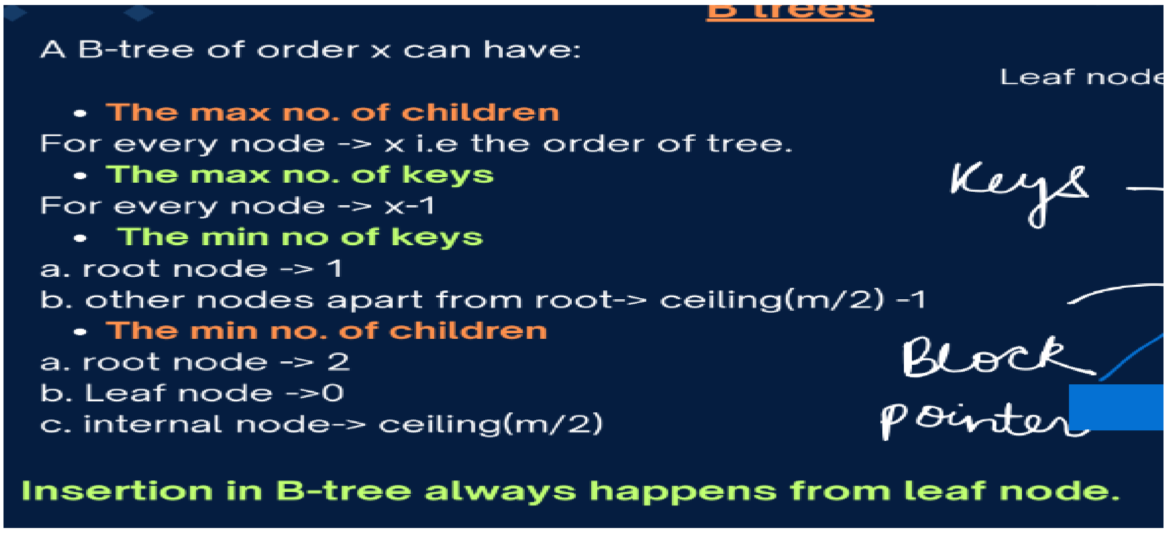
Structure of B trees:

- A B-tree is composed of nodes, each containing keys (pointers in DB to keys) and pointers (references) to child nodes. The keys within a node are sorted in ascending order.
- 

- Dp-> record/data pointer(where the record is present in secondary memory(disk)
Tp-> block/tree pointer(links to the children nodes)
- 

-

- 

- The root node is the topmost node in the tree.
- Internal nodes contain keys and child pointers.
- Leaf nodes contain keys and possibly pointers to records or other data or NULL.
- In indexing, each key in a B-tree node typically represents a value or range of values, and the associated pointer directs to a data block where records corresponding to the key(s) can be found. For example, in a database, the key might be a value in a column, and the pointer might direct to the location of a row or a set of rows in a table.
- Each key in a node represents the dividing point between two subtrees.
- No. of keys of a Node = No. of its children – 1 (always)
- Each key has an associated data pointer.
- 

B trees

A B-tree of order x can have:

- The max no. of children

For every node -> x i.e the order of tree.

- The max no. of keys

For every node -> x-1

- The min no of keys

a. root node -> 1
b. other nodes apart from root-> ceiling(m/2) -1

- The min no. of children

a. root node -> 2
b. Leaf node ->0
c. internal node-> ceiling(m/2)

Insertion in B-tree always happens from leaf node.

Here x=m=order of tree

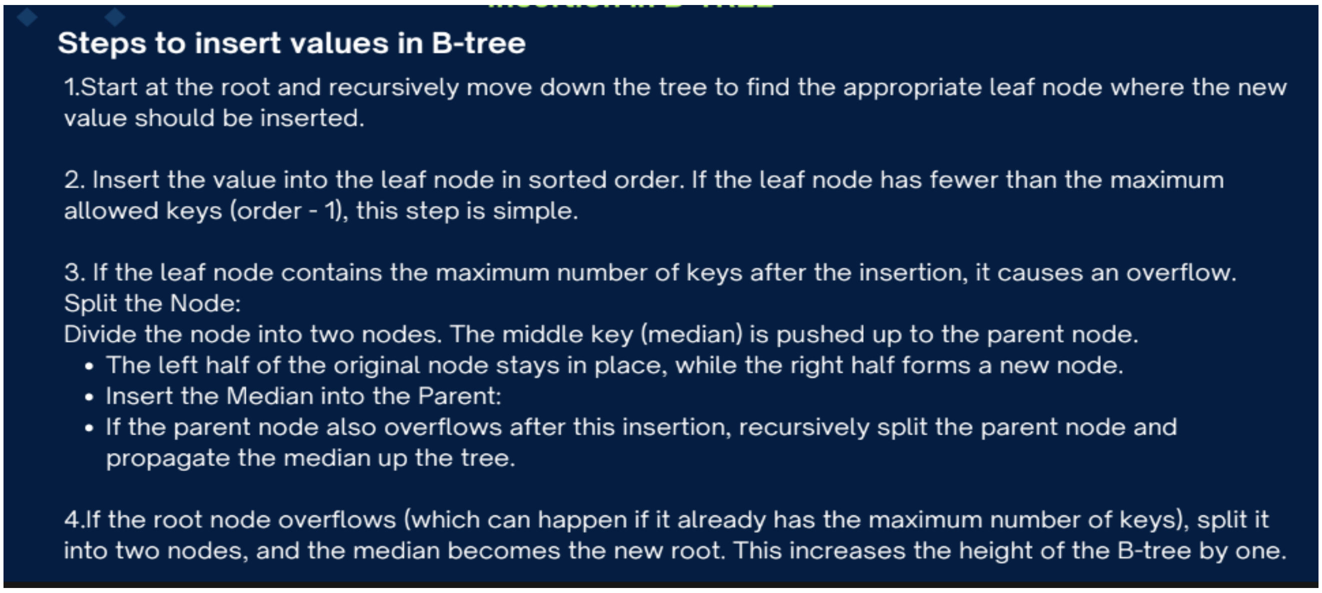
- To determine the order of a B-tree when the block size, block pointer size, and data pointer size are given:

  $$m \times Pb + (m-1) \times (K + Pd) \leq B$$

B- Block size	m- Order of tree
Pb- Block pointer size	K- Key size
Pd- Data pointer size	

<u>Insertion in B trees:</u>

- Insertion in B-tree happens from <u>leaf node</u> and values are also <u>inserted in sorted order</u>.
- The element in left of root would be less than root and the element in right would be greater than root. (therefore, median is taken)



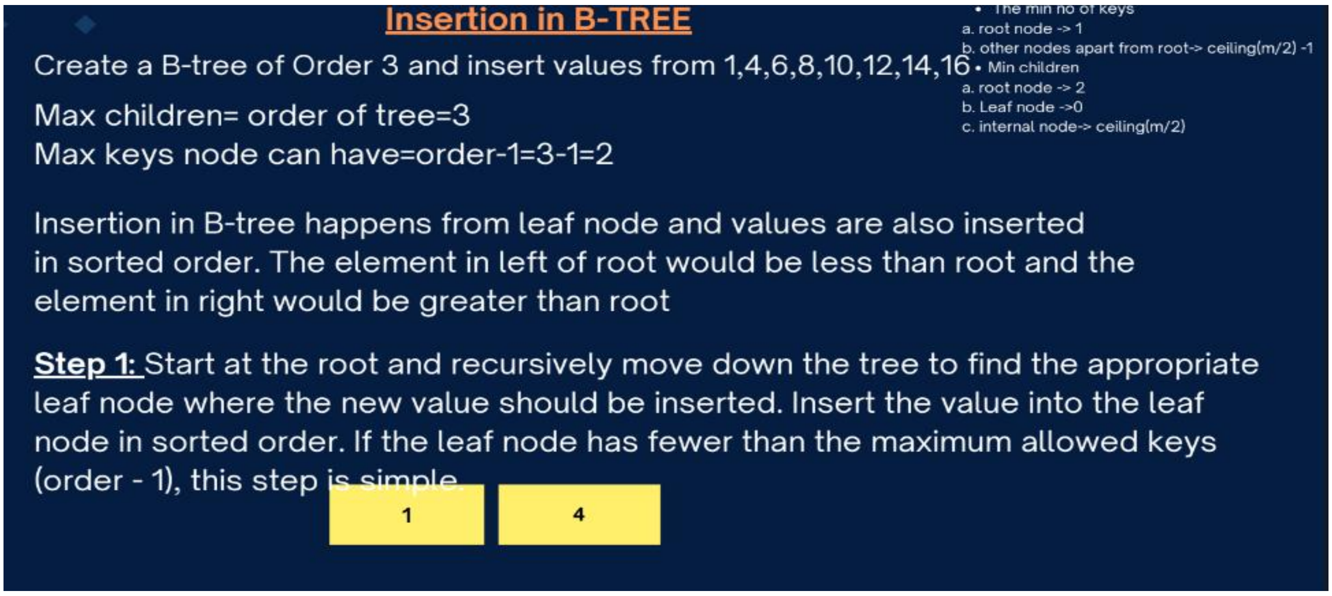
Steps to insert values in B-tree

1.Start at the root and recursively move down the tree to find the appropriate leaf node where the new value should be inserted.

2. Insert the value into the leaf node in sorted order. If the leaf node has fewer than the maximum allowed keys (order - 1), this step is simple.

3. If the leaf node contains the maximum number of keys after the insertion, it causes an overflow.
Split the Node:
Divide the node into two nodes. The middle key (median) is pushed up to the parent node.
   - The left half of the original node stays in place, while the right half forms a new node.
   - Insert the Median into the Parent:
   - If the parent node also overflows after this insertion, recursively split the parent node and propagate the median up the tree.

4.If the root node overflows (which can happen if it already has the maximum number of keys), split it into two nodes, and the median becomes the new root. This increases the height of the B-tree by one.



Insertion in B-TREE

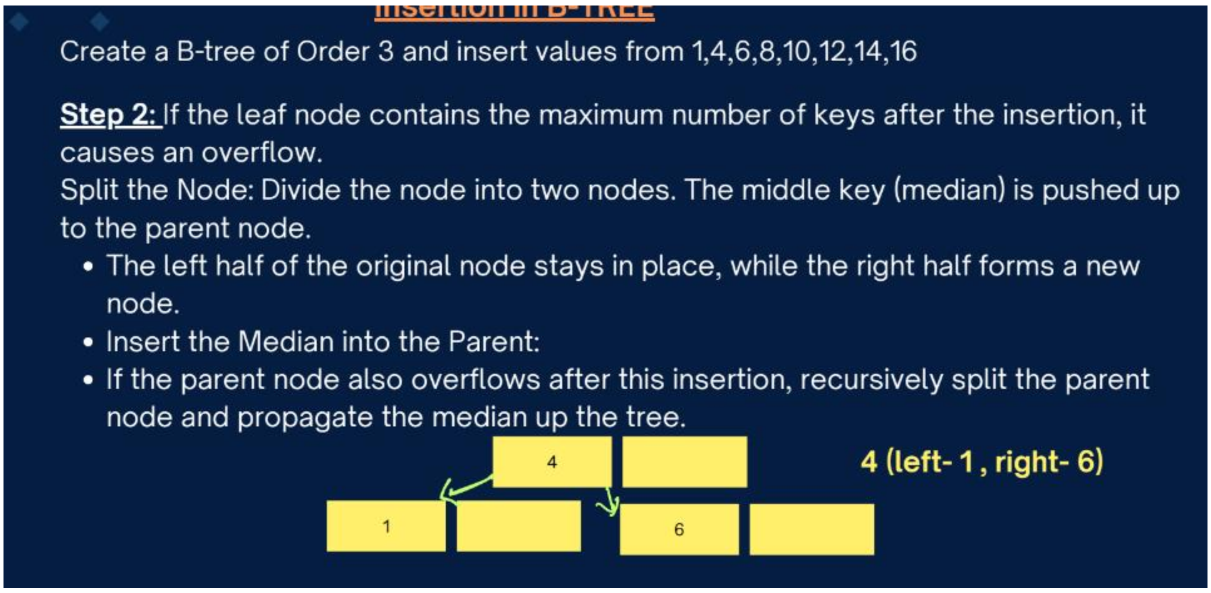
Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

Max children= order of tree=3
Max keys node can have=order-1=3-1=2

- The min no of keys
  a. root node -> 1
  b. other nodes apart from root-> ceiling(m/2) -1
- Min children
  a. root node -> 2
  b. Leaf node ->0
  c. internal node-> ceiling(m/2)

Insertion in B-tree happens from leaf node and values are also inserted in sorted order. The element in left of root would be less than root and the element in right would be greater than root

Step 1: Start at the root and recursively move down the tree to find the appropriate leaf node where the new value should be inserted. Insert the value into the leaf node in sorted order. If the leaf node has fewer than the maximum allowed keys (order - 1), this step is simple.



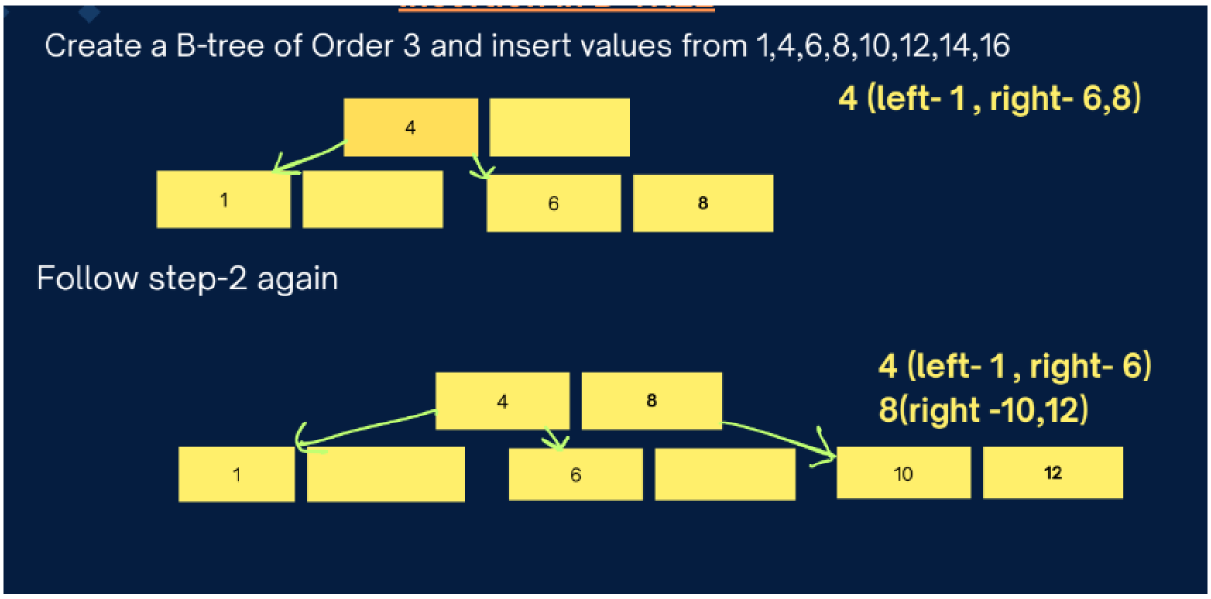
Insertion in B-TREE

Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

Step 2: If the leaf node contains the maximum number of keys after the insertion, it causes an overflow.
Split the Node: Divide the node into two nodes. The middle key (median) is pushed up to the parent node.

- The left half of the original node stays in place, while the right half forms a new node.
- Insert the Median into the Parent:
- If the parent node also overflows after this insertion, recursively split the parent node and propagate the median up the tree.

4 (left- 1 , right- 6)

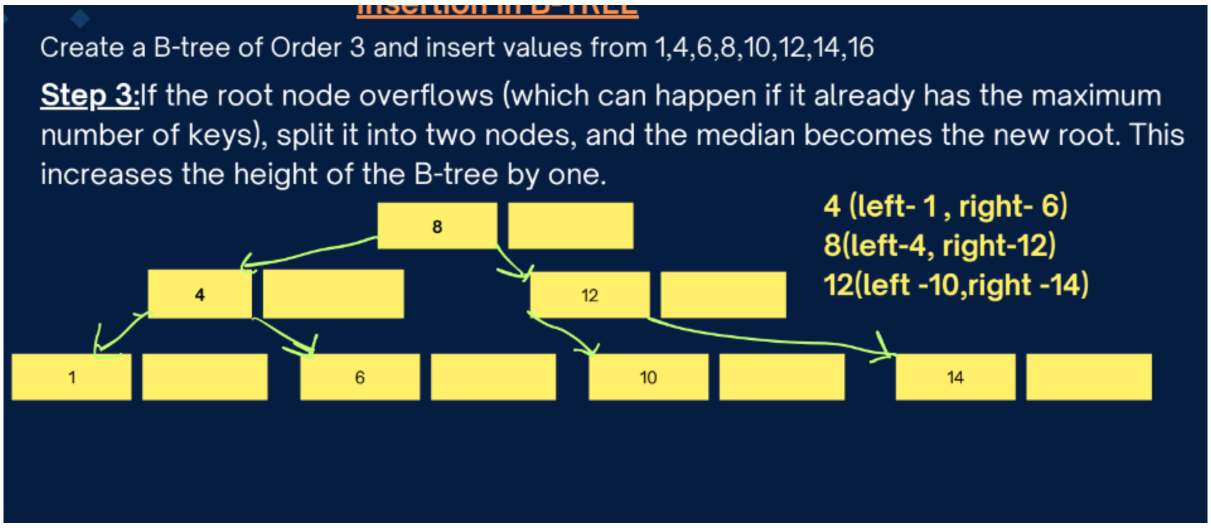


Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

4 (left- 1 , right- 6,8)

Follow step-2 again

4 (left- 1 , right- 6)
8(right -10,12)



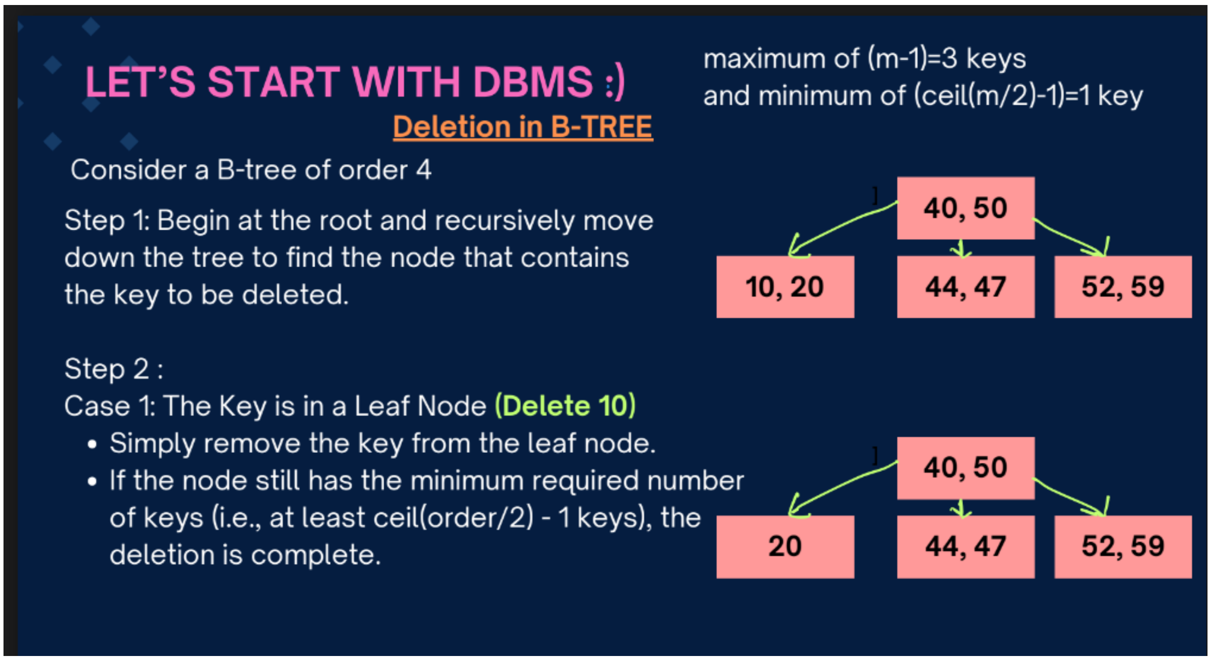
Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

Step 3: If the root node overflows (which can happen if it already has the maximum number of keys), split it into two nodes, and the median becomes the new root. This increases the height of the B-tree by one.

4 (left- 1 , right- 6)
8(left-4, right-12)
12(left -10,right -14)

Deletion in B trees:

- Take care of minimum number of keys a node can hold.



LET'S START WITH DBMS :)

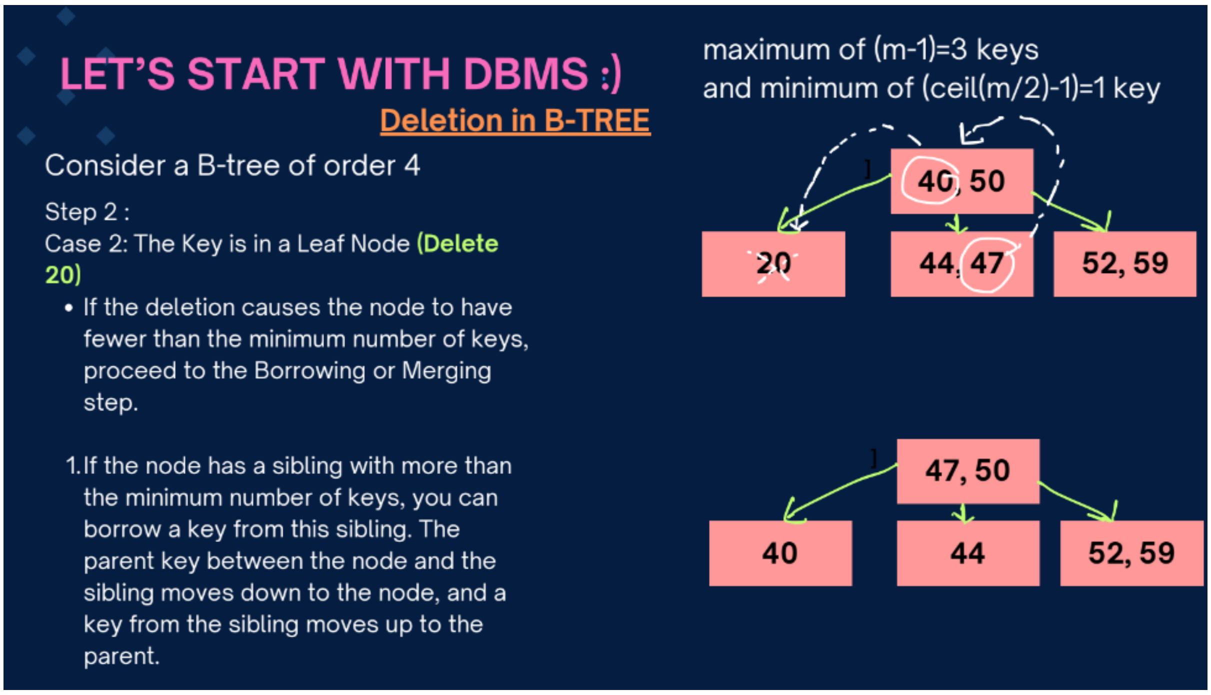
Deletion in B-TREE

Consider a B-tree of order 4

maximum of (m-1)=3 keys
and minimum of (ceil(m/2)-1)=1 key

Step 1: Begin at the root and recursively move down the tree to find the node that contains the key to be deleted.

Step 2 :
Case 1: The Key is in a Leaf Node **(Delete 10)**
- Simply remove the key from the leaf node.
- If the node still has the minimum required number of keys (i.e., at least ceil(order/2) - 1 keys), the deletion is complete.



LET'S START WITH DBMS :)

Deletion in B-TREE

Consider a B-tree of order 4

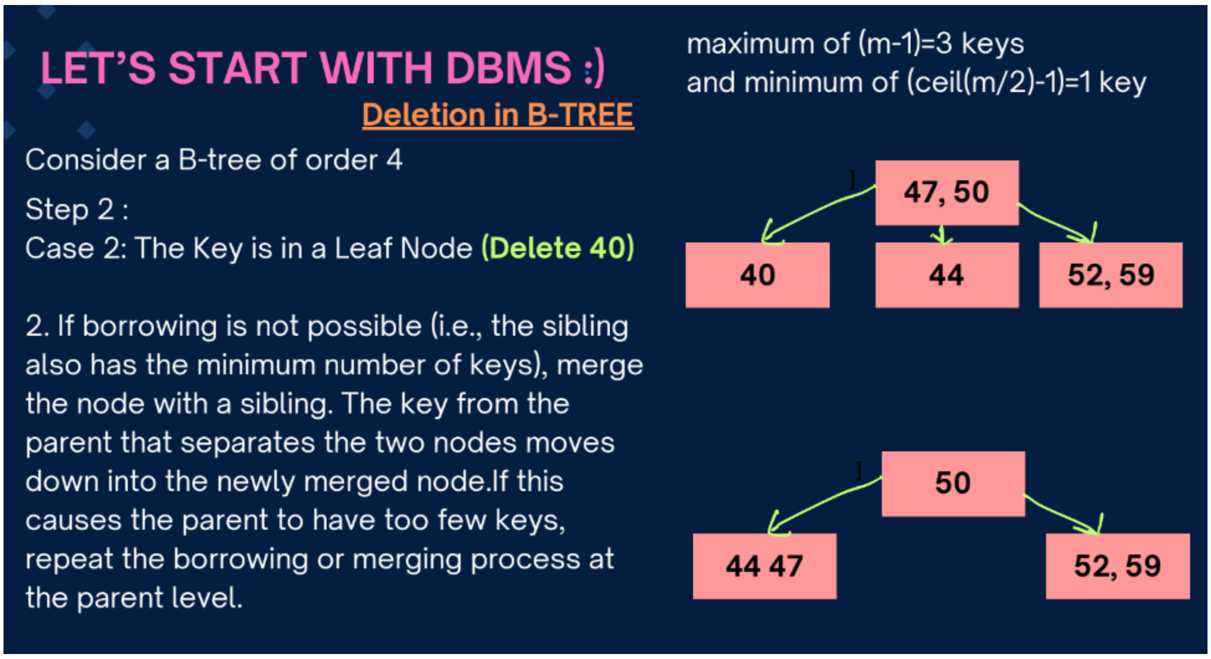
Step 2 :
Case 2: The Key is in a Leaf Node **(Delete 20)**

- If the deletion causes the node to have fewer than the minimum number of keys, proceed to the Borrowing or Merging step.

1. If the node has a sibling with more than the minimum number of keys, you can borrow a key from this sibling. The parent key between the node and the sibling moves down to the node, and a key from the sibling moves up to the parent.

maximum of (m-1)=3 keys
and minimum of (ceil(m/2)-1)=1 key

Here, 47 is moved to parent as it was right of 40 and right keys is always larger than the parent key.



LET'S START WITH DBMS :)

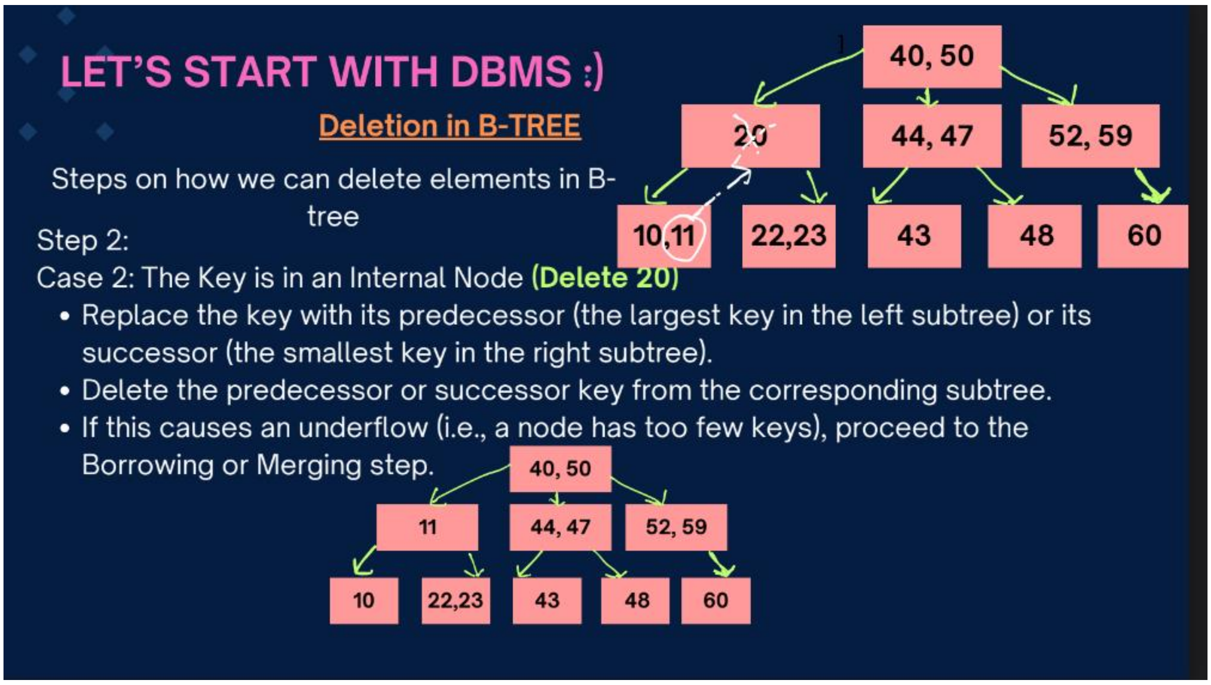
Deletion in B-TREE

Consider a B-tree of order 4

Step 2 :
Case 2: The Key is in a Leaf Node **(Delete 40)**

2. If borrowing is not possible (i.e., the sibling also has the minimum number of keys), merge the node with a sibling. The key from the parent that separates the two nodes moves down into the newly merged node.If this causes the parent to have too few keys, repeat the borrowing or merging process at the parent level.

maximum of (m-1)=3 keys
and minimum of (ceil(m/2)-1)=1 key



LET'S START WITH DBMS :)

Deletion in B-TREE

Steps on how we can delete elements in B-tree

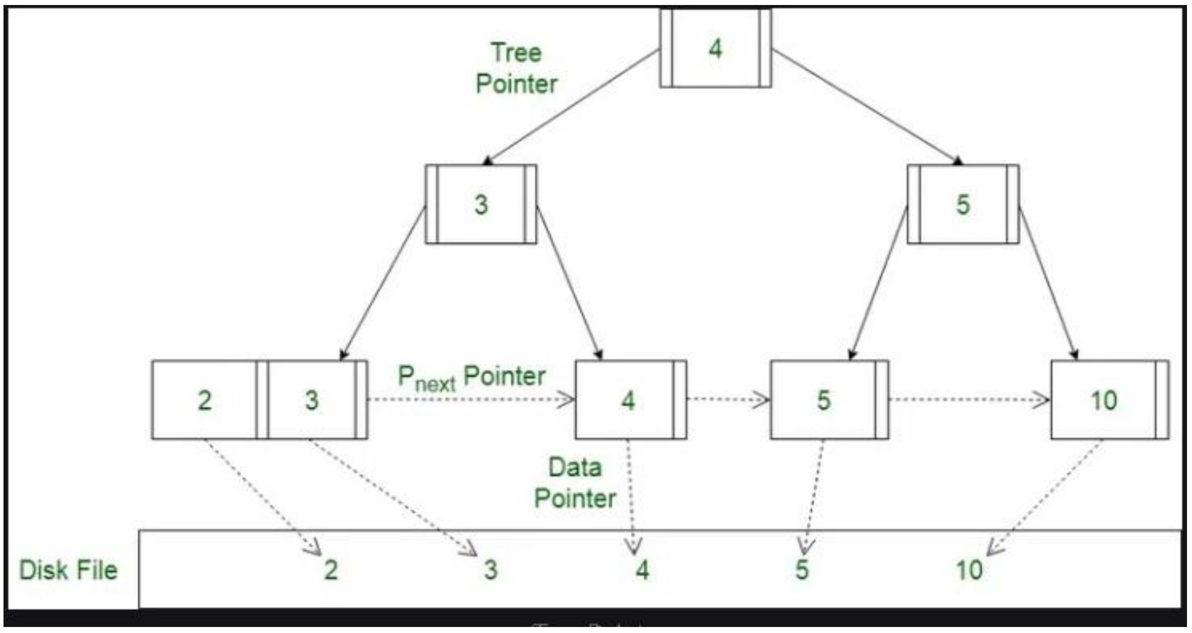
Step 2:

Case 2: The Key is in an Internal Node (Delete 20)

- Replace the key with its predecessor (the largest key in the left subtree) or its successor (the smallest key in the right subtree).
- Delete the predecessor or successor key from the corresponding subtree.
- If this causes an underflow (i.e., a node has too few keys), proceed to the Borrowing or Merging step.

B+ trees

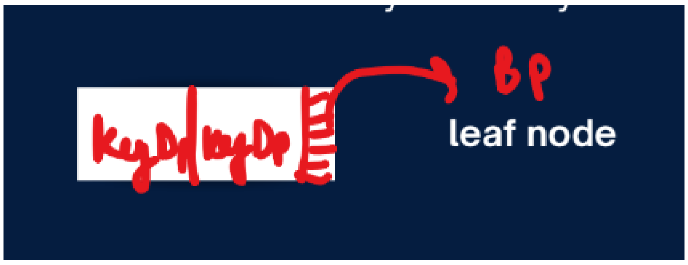
- A B+ tree is an extension of the B-tree and is commonly used in databases and file systems to maintain sorted data and allow for efficient insertion, deletion, and search operations.
- B+ tree is a balanced tree, meaning all leaf nodes are at the same level



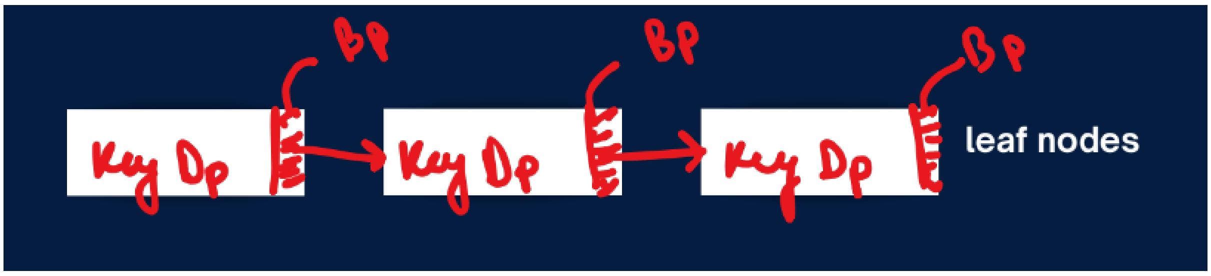
B tree Vs B+ tree

The key difference between a B+ tree and a B-tree lies in how they store data and how leaf nodes are structured.

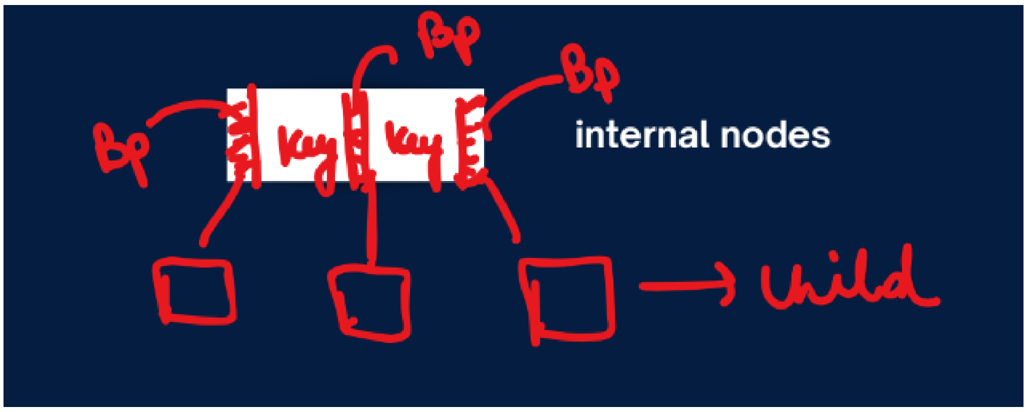
1. In a B+ tree, all actual data (or references to data Dp) are stored in the leaf nodes. Internal nodes only store keys and Bp to childern. No Dp to internal node keys.



2. In a B+ tree, Leaf nodes are linked together in a linked list fashion, allowing for efficient sequential access, one leaf node will have only 1 Bp (pointing to the next leaf node at the same level).



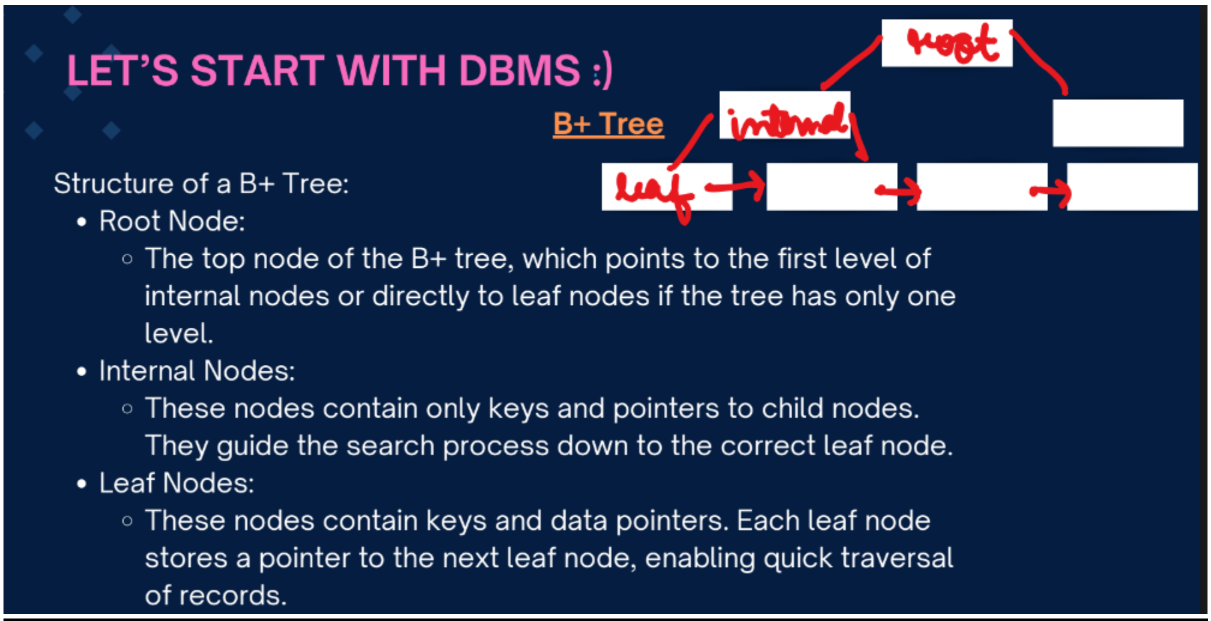
3. In a B+ tree, Leaf nodes are linked together in a linked list fashion, allowing for efficient sequential access, one leaf node will have only 1 Bp.



Insertion in B+ tree

- During inserting value in B+ tree, a copy of the key is always stored in the leaf node.
- When searching for a key in a B+ tree, the tree is traversed from the root down to the leaf nodes. The internal nodes only contain keys used for navigation, but the final matching key must be stored in the leaf node to confirm the result of the search.

Structure of B+ tree



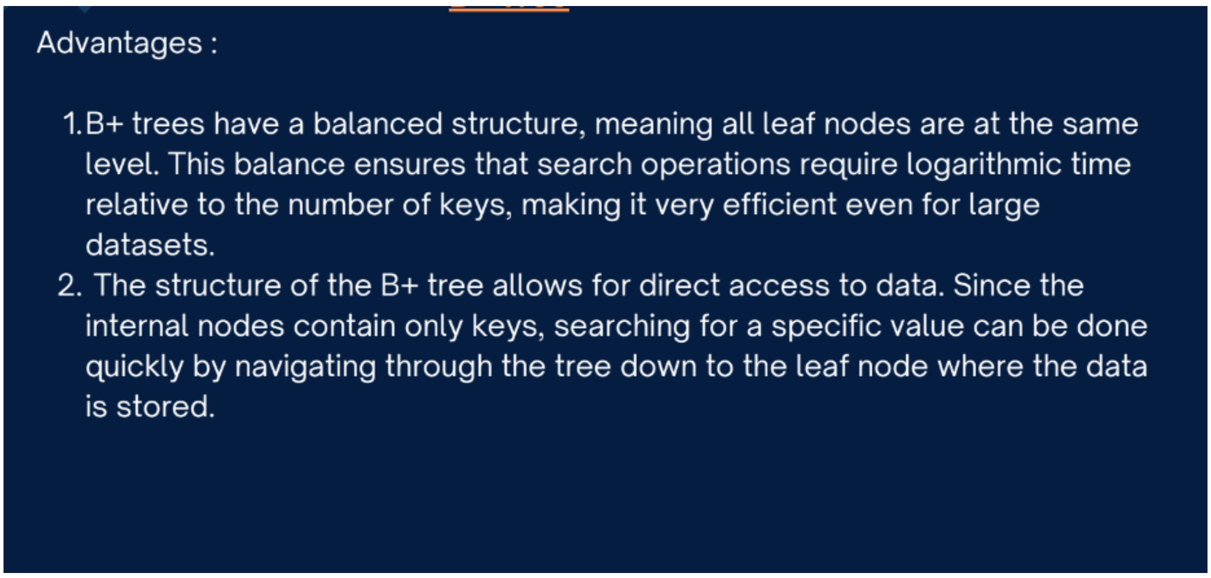
LET'S START WITH DBMS :)

B+ Tree

Structure of a B+ Tree:

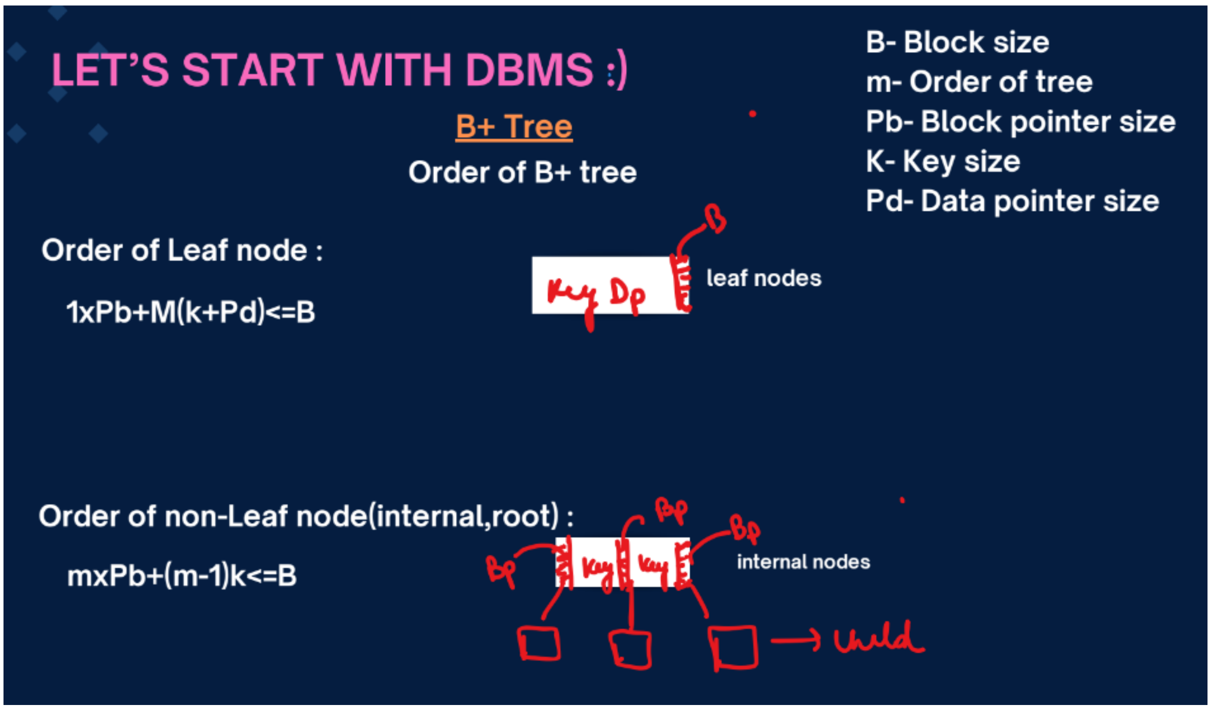
- Root Node:
  - The top node of the B+ tree, which points to the first level of internal nodes or directly to leaf nodes if the tree has only one level.
- Internal Nodes:
  - These nodes contain only keys and pointers to child nodes. They guide the search process down to the correct leaf node.
- Leaf Nodes:
  - These nodes contain keys and data pointers. Each leaf node stores a pointer to the next leaf node, enabling quick traversal of records.

Advantage of B+ tree



Advantages :

1. B+ trees have a balanced structure, meaning all leaf nodes are at the same level. This balance ensures that search operations require logarithmic time relative to the number of keys, making it very efficient even for large datasets.
2. The structure of the B+ tree allows for direct access to data. Since the internal nodes contain only keys, searching for a specific value can be done quickly by navigating through the tree down to the leaf node where the data is stored.



LET’S START WITH DBMS :)

B+ Tree

Order of B+ tree

B- Block size
m- Order of tree
Pb- Block pointer size
K- Key size
Pd- Data pointer size

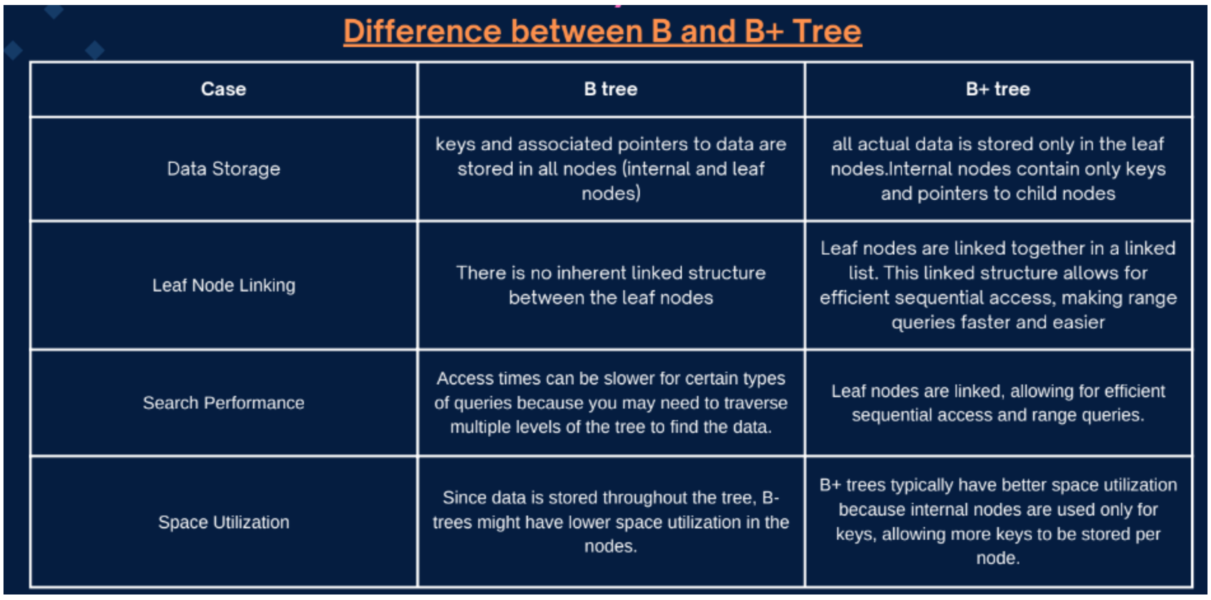
Order of Leaf node :

1xPb+M(k+Pd)<=B

Order of non-Leaf node(internal,root) :

mxPb+(m-1)k<=B

Key Dp
B
leaf nodes
Bp
Bp
Bp
key
key
internal nodes
child



Difference between B and B+ Tree

Case	B tree	B+ tree
Data Storage	keys and associated pointers to data are stored in all nodes (internal and leaf nodes)	all actual data is stored only in the leaf nodes.Internal nodes contain only keys and pointers to child nodes
Leaf Node Linking	There is no inherent linked structure between the leaf nodes	Leaf nodes are linked together in a linked list. This linked structure allows for efficient sequential access, making range queries faster and easier
Search Performance	Access times can be slower for certain types of queries because you may need to traverse multiple levels of the tree to find the data.	Leaf nodes are linked, allowing for efficient sequential access and range queries.
Space Utilization	Since data is stored throughout the tree, B-trees might have lower space utilization in the nodes.	B+ trees typically have better space utilization because internal nodes are used only for keys, allowing more keys to be stored per node.