Assignment

Roll No.: CS23B1047
Name: Dhage pratik Bhishmacharya
course: Database systems.

Q.1. compare and contrast B-Trees and B+ trees. Explain their structure and use cases.

Solution:

B Tree structure – A B-Tree is self-balancing, multi-level index designed for efficient disk-based storage. It generalizes binary search trees to allow multiple keys per node and multiple children, minimizing I/O operations.

order m: Nodes have up to m-1 keys & m children.

keys: sorted, up to m-1, minimum is $\lceil \frac{m}{2} \rceil - 1$ (except root node has minimum 1.)

pointers: k+1 children for k keys

B+ Tree structure – A B+ Tree is a variant of the B tree optimized for range queries and sequential access, commonly used in database indexing.

order m: Internal nodes have up to m-1 keys, m children; leaf nodes store up to m keys.

Internal nodes: keys only, pointers to children, No data.

Leaf Nodes: All keys and data, sorted, linked sequentially (linked-list)

Balanced, with minimum $\lfloor \frac{m}{2} \rfloor$ keys in leaves, $\lceil \frac{m}{2} \rceil$ children in internal nodes. (except root.)

comparison:

feature	B-Tree	B+ Tree
Data storage	Internal and leaf nodes	Leaf nodes only
leaf Linking	None	Doubly-linked list
search path	May end at internal nodes.	Always end at leaf node.
Range queries	Less efficient, requires traversal.	Efficient, sequential leaf access.
fanout	Lower (data reduces key capacity)	Higher (keys only in internal nodes)
space efficiency	Less efficient	More efficient.

Use cases:

1) B-Tree -
   (a) File systems: Used in file systems like NTFS and HFS+ for hierarchical directory structures.
   (b) Database with frequent updates.
   (c) Embedded systems - preferred in memory-constrai-environments.

2) B+ Tree -
   (a) Database Indexing: used in Relational DBMS for indexing tables.
   (b) Data Warehouses: Ideal for analytical queries requiring scans over large datasets.
   (c) search engines: supports efficient keyword-based range searches.

Conclusion: B+ Trees are generally preferred in database systems due to their optimization for range queries and sequential access.

Q.2. Explain the insertion operation in a B-tree with an example. How does it maintain balance?

Solution: Insertion in a B-Tree maintains balance by ensuring nodes adhere to key limits and the tree remains height balanced. Steps:

(1) Locate leaf: Traverse from root to the appropriate leaf node using key comparisons.

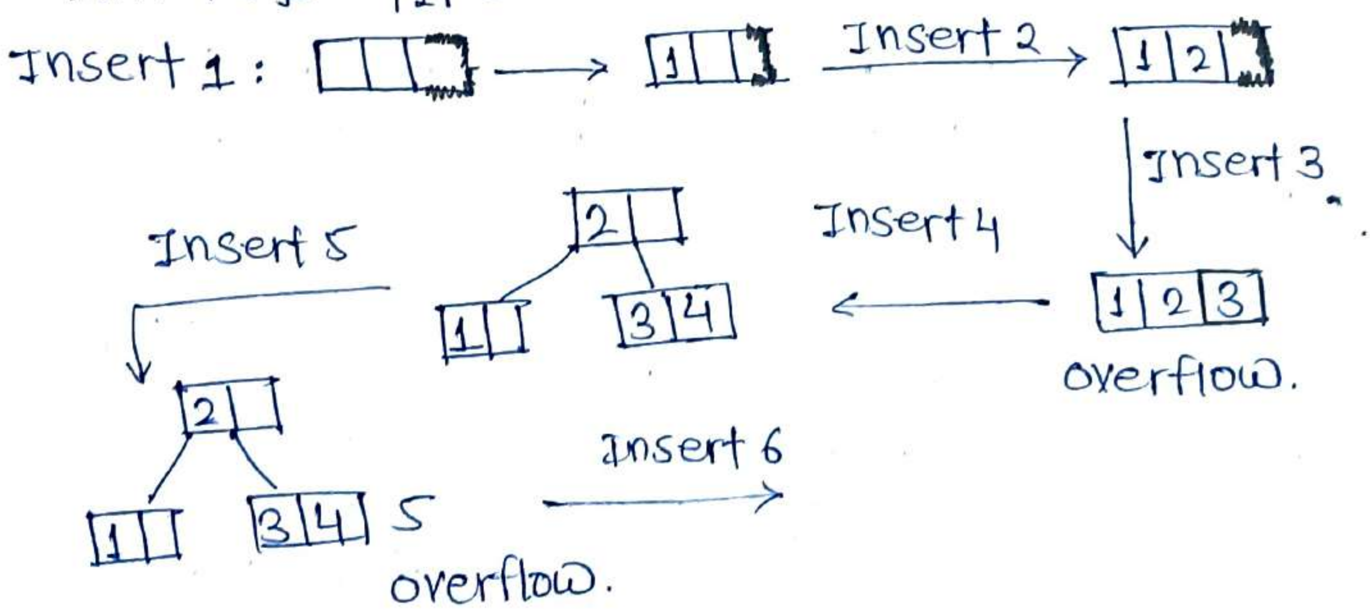
(2) Insert key:
If node has $< m-1$ keys; insert in sorted order
If overflow, $m-1$ keys full, split:
i. Divide into two nodes with $\lfloor \frac{m}{2} \rfloor$ and $\lceil \frac{m}{2} \rceil - 1$ keys.
ii. promote median key to parent
iii. Adjust child pointers

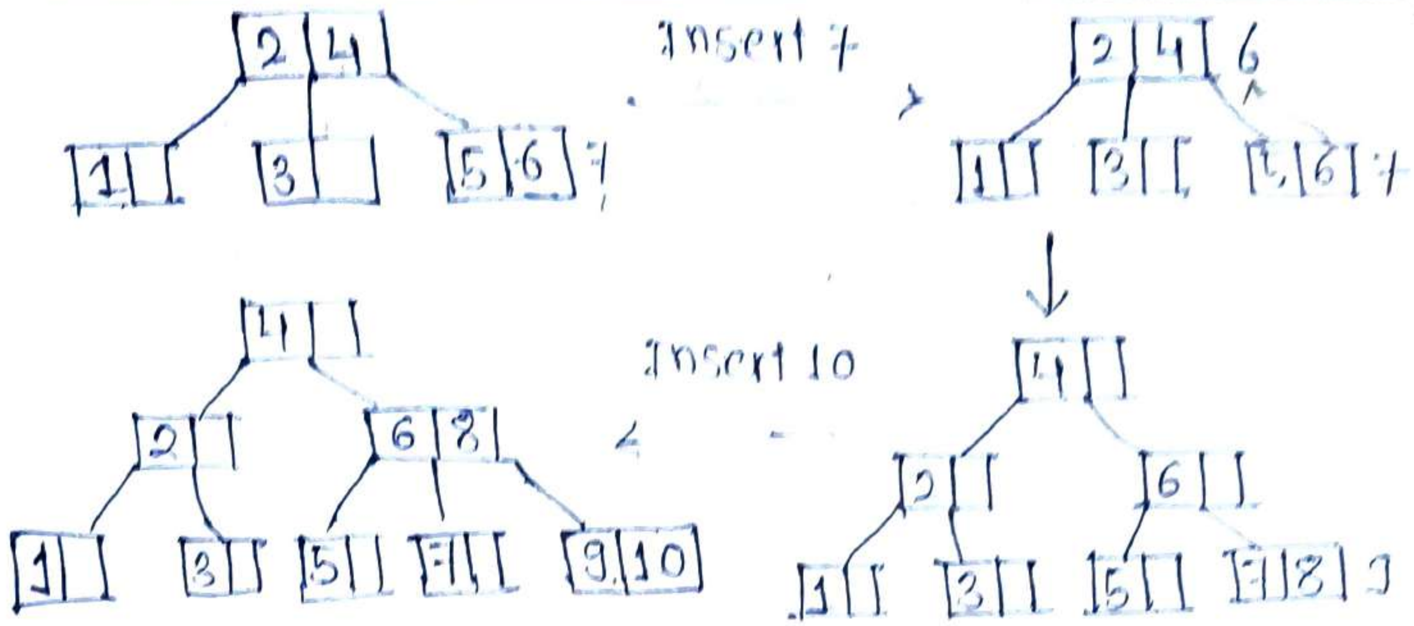
(3) propogate splits: If parent is full, split recursively up to root. If root splits, create new root, increasing height.

(4) Balance: Nodes (except root) maintain $\lceil \frac{m}{2} \rceil - 1$ to $m-1$ keys.

Example: create a B-Tree of order 3 by inserting values from 1 to 10.

max keys $= m-1 = 3-1 = 2$
min keys $= \lceil \frac{m}{2} \rceil - 1 = 2-1 = 1$





Balance Maintainance:

Node splitting: When a node becomes full, it is split into two nodes, distributing keys evenly & promoting the median key.

Minimum key constraint: After splitting, each node (except root) has at least $\lceil \frac{m}{2} \rceil - 1$ keys.

Height adjustment: splitting may increase the tree height only when root splits, ensuring logarithmic height.

Recursive propogation: splits propogate upwards, adjusting the tree structure to maintain balance.

Q.3. Describe the deletion process in a B+ Tree. What challenges are faced and how are they resolved?

solution:

Deletion process:

(1) Locate key: Traverse to leaf node ~~and ensures~~ containing the key.

(2) Delete key: (a) Remove key from leaf
(b) If leaf has $\geq \lceil \frac{m}{2} \rceil$ keys, done.
(c) If underflow ($\leq \lceil \frac{m}{2} \rceil$ keys), balance.

(3) Handle underflow:

(a) Borrow: If sibling has $> \lfloor \frac{m}{2} \rfloor$ keys, borrow a key via parent, update parent's separator.

(b) Merge: If sibling has $\leq \lfloor \frac{m}{2} \rfloor$ keys, merge with sibling and parent's separator key, remove separator from parent.

propogate underflow to parent if needed.

(4) Update internal nodes: If key was a separator, replace with predecessor / successor from leaf.

(5) Root adjustment: If root has one child post-deletion, make child the new root, reducing height.

Challenges and Resolutions:

(1) underflow in leaves-

challenge: Deletion leaves node with $< \lfloor \frac{m}{2} \rfloor$ keys.

Resolution: Borrow from sibling or merge, preferring borrowing to maintain height.

(2) parent underflow-

challenge: Merging leaves causes parent to underflow.

Resolution: Borrow or merge at parent level, propogating recursively

(3) separator key updates-

challenge: Deleted key in internal nodes must be replaced.

Resolution: use predecessor/ successor from leaf to maintain search property.

(4) Height Reduction-

challenge: Root may have one child after merges

Resolution: set child as new root, reducing height.

Q.4. Discuss the advantages and disadvantages of using B+ Trees over B-trees in database indexing.

Advantages of B+ Trees:

(1) Efficient Range Queries - linked leaf nodes allow sequential access, making range queries faster than B-Trees.

(2) Higher fanout - Internal nodes store only keys, not data, allowing more keys per node. this reduces tree height, improving search.

(3) simplified search path - All data is at leaf nodes, ensuring consistent search paths to leaves.

(4) Better space optimization - storing data only in leaves allows internal nodes to hold more keys, reducing storage overhead compared to B-Trees,

(5) support for sequential access - critical for database operations like table scans or joins.

Disadvantages of B+ Trees:

(1) complex maintenance - maintaining the linked list of leaf nodes adds complexity during insertions & deletions.

(2) slower point queries - point queries may be slightly slower than in B-trees since data is only in leaves.

(3) Higher overhead for updates - Insertions and deletions require updating the linked list & potentially adjusting separator keys in internal nodes, increasing overhead.

(4) Memory usage for linking - the doubly-linked list in leaf nodes requires additional pointers, increasing memory usage, which may be a concern in memory-constrained systems.

Q5. construct a B+ Tree of order 4 by inserting the following sequence of keys: 10, 20, 5, 6, 12, 30, 7, 17. Show each step clearly.

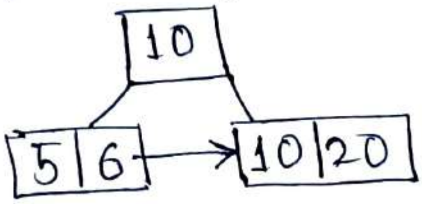
solution: order (m) = 4
min. no. of keys = 1 (root), 2 (leaf & internal)
max no. of keys = 4-1 = 3 (internal), ~~2~~ 4 (leaf)

create and insert 10. | 10 |

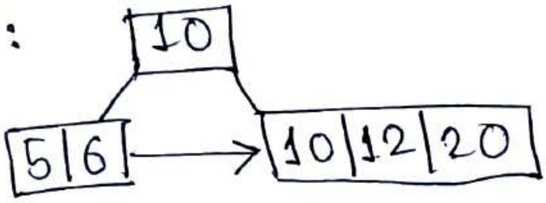
insert 20 and insert 5. | 5 | 10 | 20 |

insert 6. | 5 | 6 | 10 | 20 | .... overflow (full).

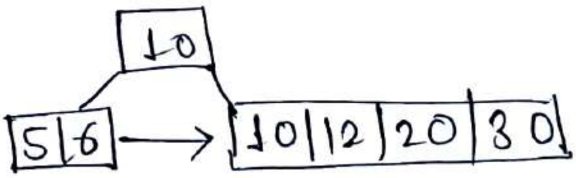
∴ split and promote 10 to parent



insert 12 :

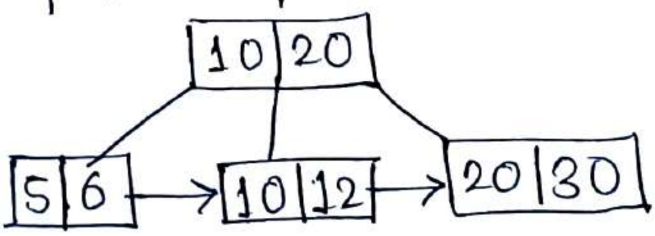


insert 30 :



.... overflow

∴ split and promote 20 to parent.



insert 7 and insert 17