

# CS525: Advanced Database Organization

## Notes 5: Indexing and Hashing Part II: Tree Indexes - B<sup>+</sup>-Tree

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

[yelmehdwi@iit.edu](mailto:yelmehdwi@iit.edu)

February 14<sup>th</sup> 2024

Slides: adapted from courses taught by Hector Garcia-Molina, Stanford, Andy Pavlo, Carnegie Mellon University Elke A. Rundensteiner, Worcester Polytechnic Institute, Shun Yan Cheung, Emory University, Marc H. Scholl, University of Konstanz, Principles of Database Management , Ellen Munthe-Kaas, Universitetet i Oslo, & Joy Arulraj, Georgia Teck

# Today's Agenda

- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level . . .
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- **B<sup>+</sup>-Trees** (Today)
  - B<sup>+</sup>-Tree Overview
  - Design Decisions
  - Optimizations
- Hash Tables (Next)

# Database Indexes

- Within a database systems, there exists a variety of data structures serving different functions, including managing internal metadata, storing essential data, handling temporary data, and optimizing data retrieval through table indexes.
- An index is a specific data structure designed to accelerate data retrieval operations on a table, albeit at the expense of increased write operations and storage space utilization.
- These indexes serve as efficient shortcuts to swiftly locate data, eliminating the need to scan every individual row within a table each time the table is accessed.

# Data Structures

- Two types of data structures:

- Order Preserving Indexes

- A tree-like structure that maintains keys in some sorted order.
    - The average number of iterations needed to reach any node depends on the height of the tree.
    - In a balanced binary search tree, the height is  $O(\log n)$ , where  $n$  is the number of nodes in the tree.
    - Therefore, on average, it would take  $O(\log n)$  iterations to reach any node.
    - Example: AGE > 20 AND AGE < 30

- Hashing Indexes

- An associative array that maps a hash of the key to a particular record.
    - The average number of iterations to reach any node is typically constant ( $O(1)$ ) for equality predicates.
      - *Only supports equality predicates with  $O(1)$  searches.*
    - This is because hashing allows for direct access to the desired record based on the hash value.
    - Example: AGE = 20

# Multilevel Indexes Revisited

- Creating index-to-an-index results in multilevel indexes
- Multilevel indexes useful for speeding up data access if lowest level index becomes too large
- Index can be considered as a sequential file and building an index-to-the-index improves access
- Higher-level index is, again, a sequential file to which index can be built and so on
- Lowest level index entries may contain pointers to disk blocks or records
- Higher-level index contains as many entries as there are blocks in the immediately lower level index
- Index entry consists of search key value and reference to corresponding block in lower level index
- Index levels can be added until highest-level index fits within single disk block
- First-level index, second-level index, third-level index etc.

# Multilevel Indexes Revisited

- Multilevel index can be considered as search tree, with each index level representing level in tree, each index block representing a node and each access to the index resulting in navigation towards a subtree in the tree
- Multilevel indexes may speed up data retrieval, but large multilevel indexes require a lot of maintenance in case of updates

# B-tree Family

- Almost every modern DBMS that supports order-preserving indexes uses a  $B^+$ -Tree.
- There is a specific data structure called a B-Tree but the term is also being used to generally refer to a class/family of data structures.
- The B-Tree family of data structures is a group of related tree-based data structures used for efficient storage and retrieval of data in databases and file systems.
  - B-Tree (1971)
  - $B^+$ -Tree (1973)
  - $B^*$ -Tree (1977)
  - $B^{\text{link}}$ -Tree (1981)<sup>1</sup>
- The primary difference between the original B-Tree and the  $B^+$ -Tree is that B-Trees store keys and values in all nodes, while  $B^+$ -Trees store values only in leaf nodes
- Modern  $B^+$ -Tree implementations combine features from other B-Tree variants, such as the sibling pointers used in the  $B^{\text{link}}$ -Tree.

---

<sup>1</sup> Efficient locking for concurrent operations on B-trees

# B<sup>+</sup>-Tree Structure

- B<sup>+</sup>-Tree<sup>1</sup>: a dynamic multi-level index
- B<sup>+</sup>-Tree is a self-balancing tree data structure<sup>2</sup> that keeps data sorted and allows searches, sequential access, insertion, and deletions in  $O(\log n)$ .
  - Generalization of a binary search tree in that a node can have more than two children.
  - Optimized for systems that read and write large blocks of data.
- It is perfectly balanced (i.e., every leaf node is at the same depth from the root node in tree).

---

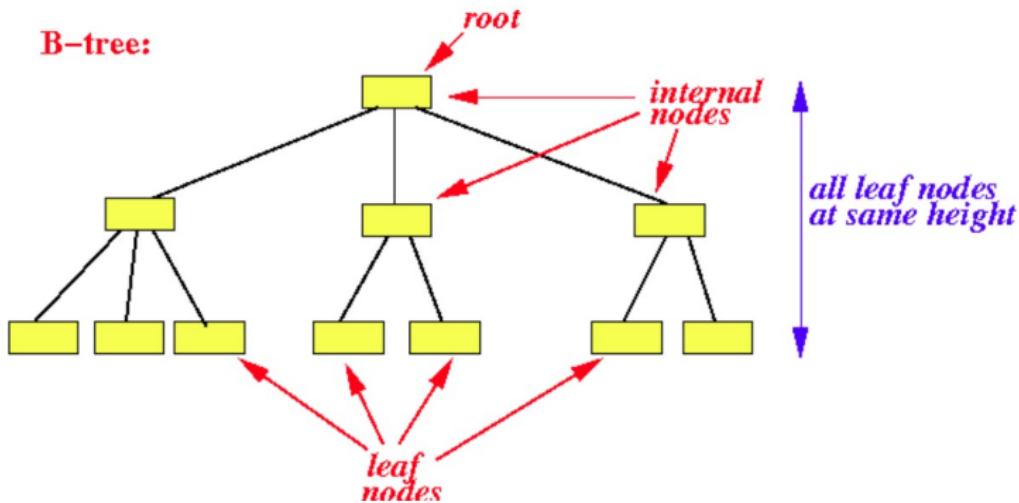
<sup>1</sup> Ubiquitous B-Tree by Douglas Comer.

<sup>2</sup> Self-Balancing Binary Search trees defined as trees that automatically keeps height (maximal number of levels below the root) as small as possible when insertion and deletion operations are performed on tree

## B<sup>+</sup>tree Nodes

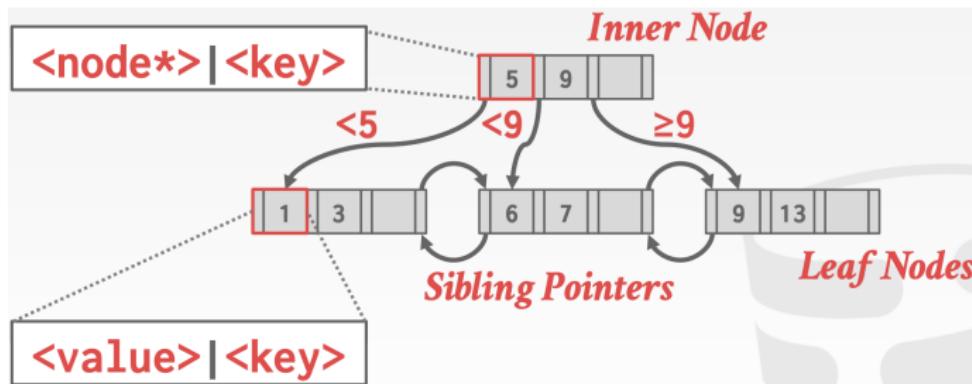
- Every node in a B<sup>+</sup>-Tree is comprised of an array of *key/value* pairs:
  - The arrays are (usually) kept in sorted key order.
  - The *keys* are derived from the attributes(s) that the index is based on.
  - The *values* will differ based on whether the node is classified as *inner nodes* or *leaf nodes*.
  - The value array for *inner nodes* will contain pointers to other nodes.

## B<sup>+</sup>tree Nodes



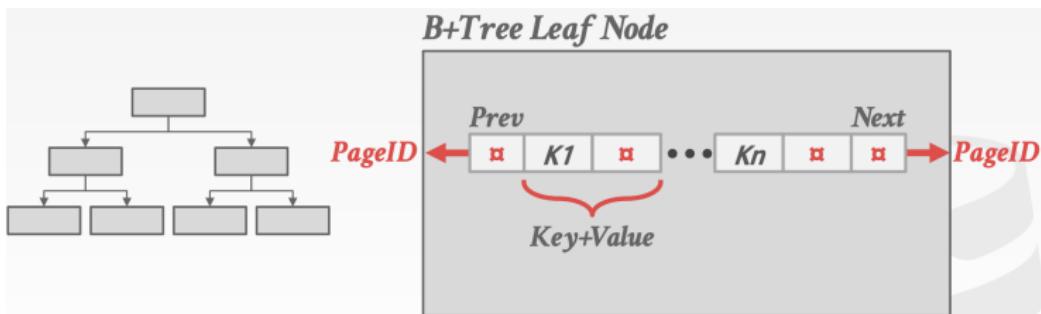
- Each node is stored in one block on disk
- **Root node**: the node at the “top” of the tree
- **Inner/Internal/intermediate node**: a node that has one or more child node(s)
- **Leaf node**: a node that does not have any children nodes

## B<sup>+</sup>-Tree Diagram: Example: $n = 3$

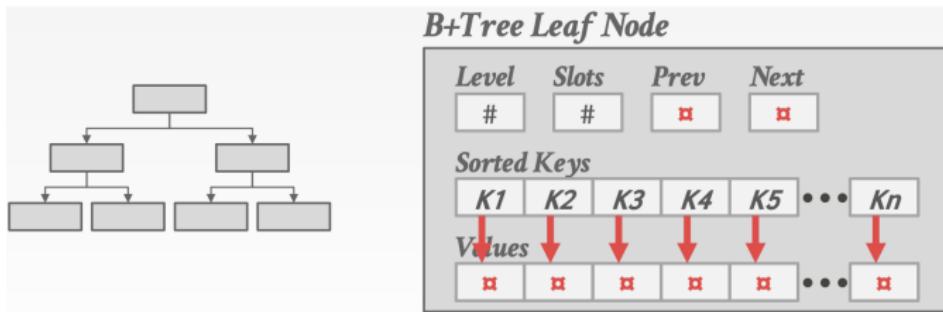


- The value array for **inner nodes** will contain pointers to other nodes.

# $B^+$ -Tree Leaf Nodes



# B<sup>+</sup>-Tree Leaf Nodes



# Leaf Node Values

- The *values* will differ based on whether the node is classified as **inner nodes** or **leaf nodes**.
- Two approaches for **leaf node** values
  - **Approach 1: Record IDs**
    - A pointer to the location of the tuple that the index entry corresponds to.
    - Used in PostgreSQL, MS SQL Server, IBM DB2, Oracle
  - **Approach 2: Tuple Data**
    - The actual contents of the tuple is stored in the **leaf node**
    - Only works for primary key indexes.
    - For secondary indexes, it have to store the **record id** (a pointer to the primary key of the corresponding data record) as their values.  
*because secondary indexes are typically based on non-unique attributes, and the primary key is needed to uniquely identify the associated data record.*
    - Used in MS SQL Server, MySQL, Oracle

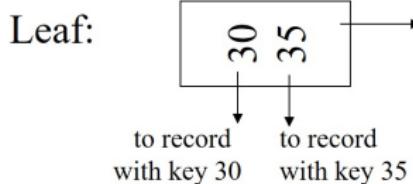
# Main Challenge in B<sup>+</sup>Trees

- To ensure good search cost, the tree should be kept of **minimum and uniform height**. Keep the tree:
  - full (packed), and
  - balanced (almost uniform height).
- This can be difficult – in face of insertions and deletions.
- **Solution:** Keep the tree “semi-full” (i.e., each tree node half-full)
  - Makes it easy to keep the tree balanced and semi-optimal.

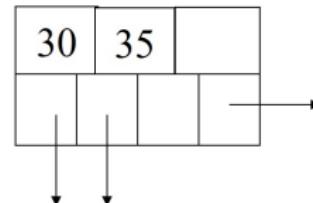
# Example B<sup>+</sup>-Tree nodes with $n = 3$

- Each internal node is stored in one block on disk
- and contains at most  $n$  keys and  $(n+1)$  pointers
- *The reason for using  $(n + 1)$  pointers in the representation of internal nodes rather than  $(n + 2)$ , is primarily a matter of convention and efficiency in tree traversal.*

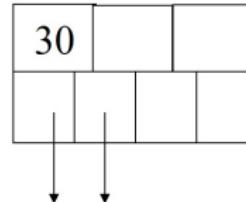
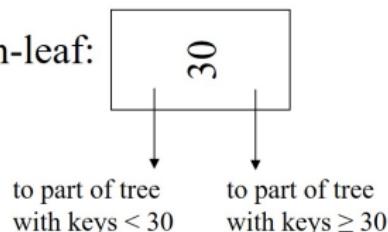
more concise notation



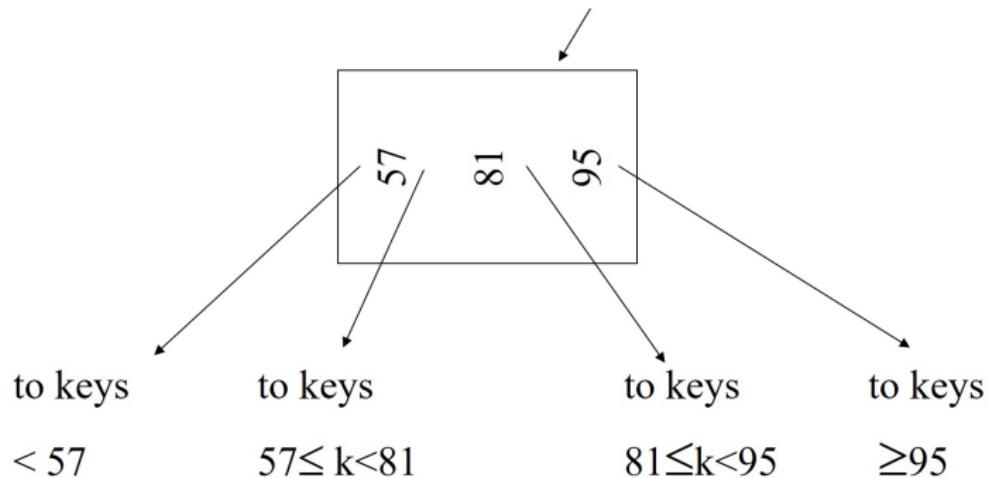
textbook notation



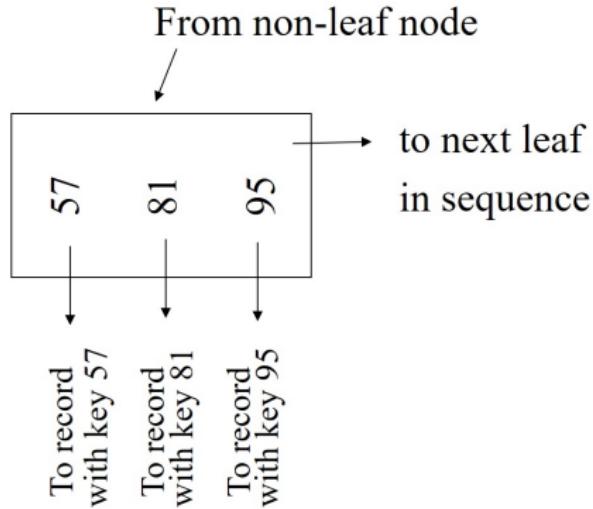
Non-leaf:



## Sample non-leaf

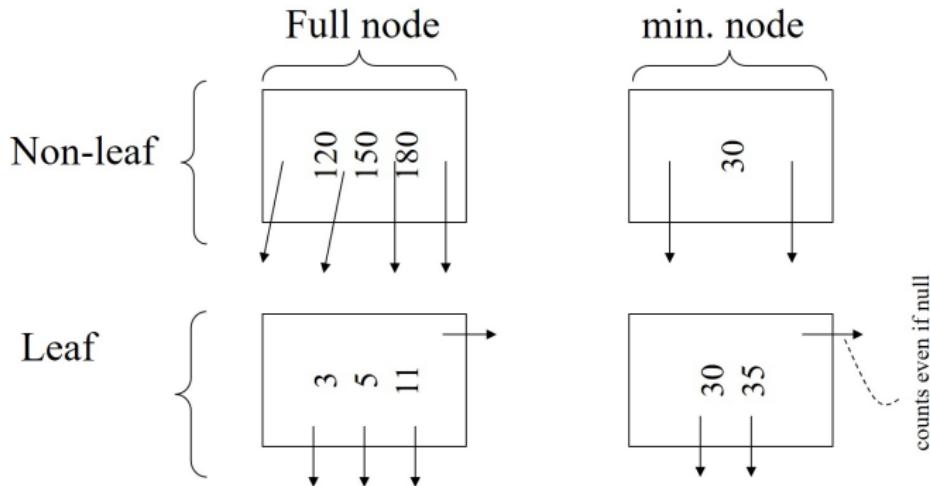


## Sample leaf node



The last pointer, *sibling pointer*, points to the next leaf node (a disk block) in the  $B^+$ -Tree

$$n = 3$$



- Each internal node is stored in one block on disk and contains at most  $n$  *keys* and  $n+1$  *pointers* (non-null children)

$$\text{Size of nodes : } \begin{cases} n + 1 \text{ pointers} \\ n \text{ keys} \end{cases} \quad (\text{fixed})$$

## B<sup>+</sup>tree Nodes

- Each node in the tree is a **block**, which contains **search keys** and **pointers**
- Parameter  **$n$** , which is largest value so that  **$n+1$**  pointers and  **$n$**  keys fit in one **block**

### Example (1)

If block size is 4096 bytes, keys be integers (4 bytes each), and pointers be 8 bytes each, then  $n = 340$ .

## Don't want nodes to be too empty

- Use a “Fill Factor” to control the growth and the shrinkage. A 50% fill factor would be the minimum for B<sup>+</sup>-Tree
- Use at least (to ensure a balanced tree)
  - Non-leaf:  $\lceil \frac{n+1}{2} \rceil$  pointers (to nodes)
    - Except **root**: required at least 2 be used
  - Leaf:  $\lfloor \frac{n+1}{2} \rfloor$  pointers (to data)

# Number of pointers/keys for B<sup>+</sup>-Tree

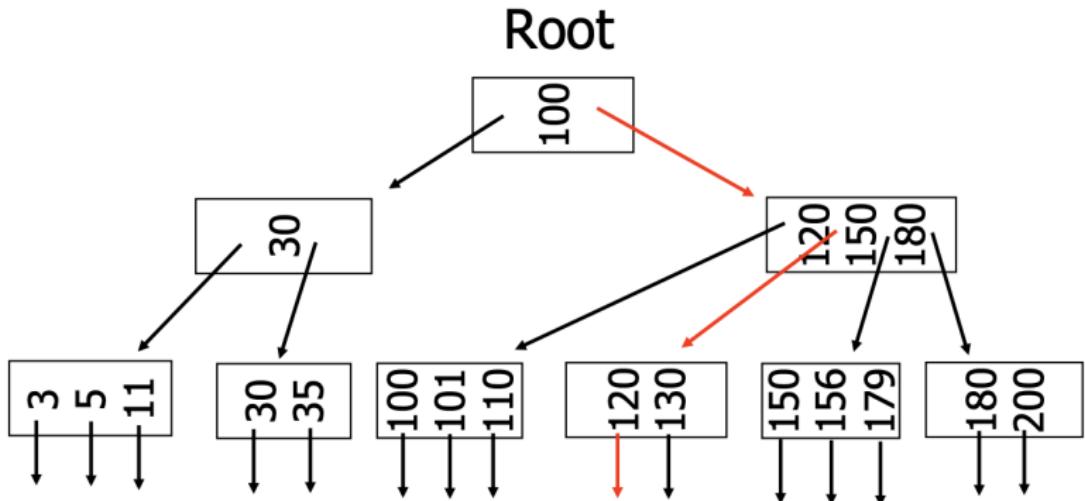
	Max ptrs	Max keys	Min ptrs	Min keys
Non-leaf (non-root)	$n + 1$	n	$\lceil \frac{n+1}{2} \rceil$	$\lceil \frac{n+1}{2} \rceil - 1$
Leaf (non-root)	$n + 1$	n	$\lfloor \frac{n+1}{2} \rfloor + 1$	$\lfloor \frac{n+1}{2} \rfloor$
Root	$n + 1$	n	$2^*$	1

\*When there is only one record in the B<sup>+</sup>-Tree, min pointers in the root is 1 (the other pointers are null)

# Search Algorithm

- Search for key  $k$
- Start from root until leaf is reached
- For current node find  $i$  so that
  - $\text{Key}[i] < k \leq \text{Key}[i + 1]$
  - Follow  $(i+1)^{\text{th}}$  pointer
- If current node is leaf, return pointer to record or fail (no such record in tree)

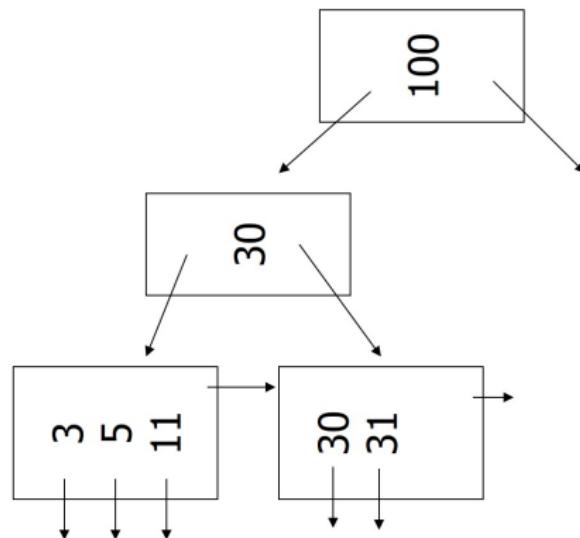
Search Example: k=120, n=3



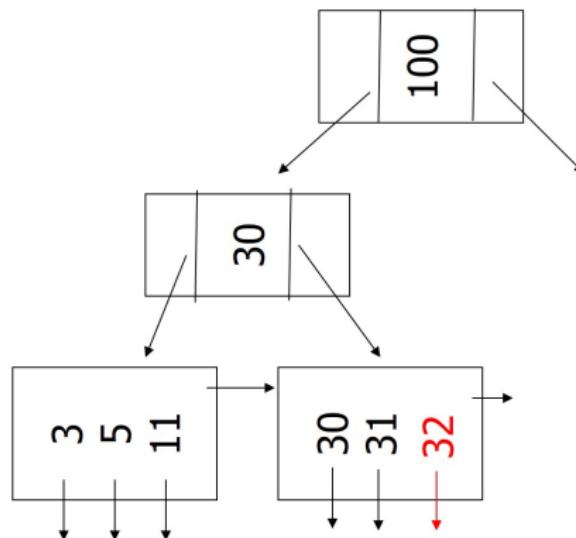
# Insert into B<sup>+</sup>-Tree

- a) simple case
  - space available in **leaf**
- b) **leaf** overflow
- c) **non-leaf** overflow
- d) new **root**

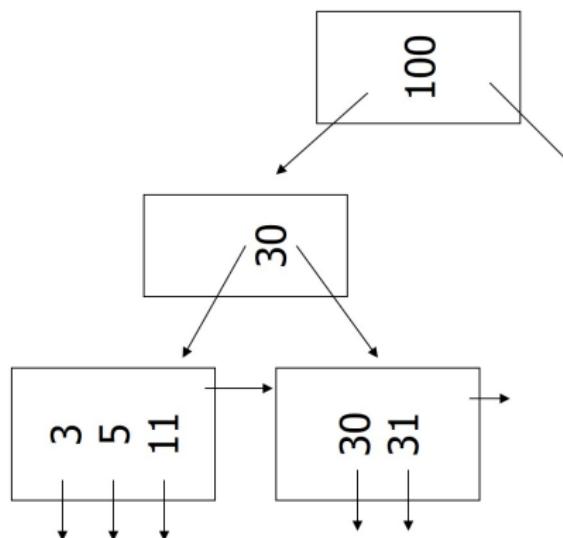
a) Insert key = 32, n = 3



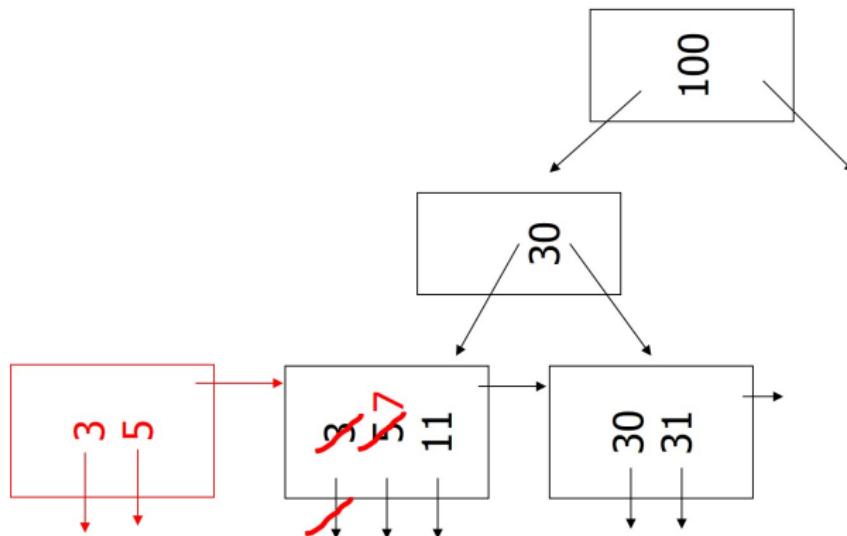
a) Insert key = 32, n = 3



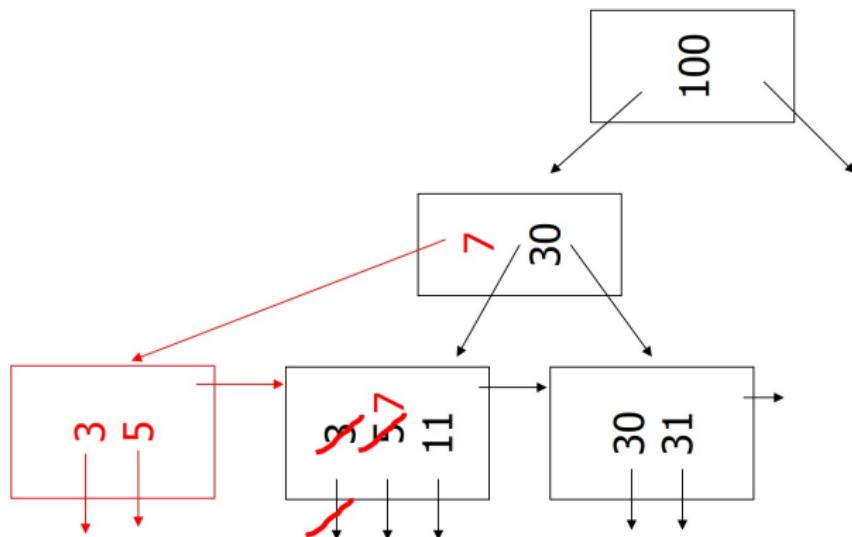
b) Insert key = 7, n = 3



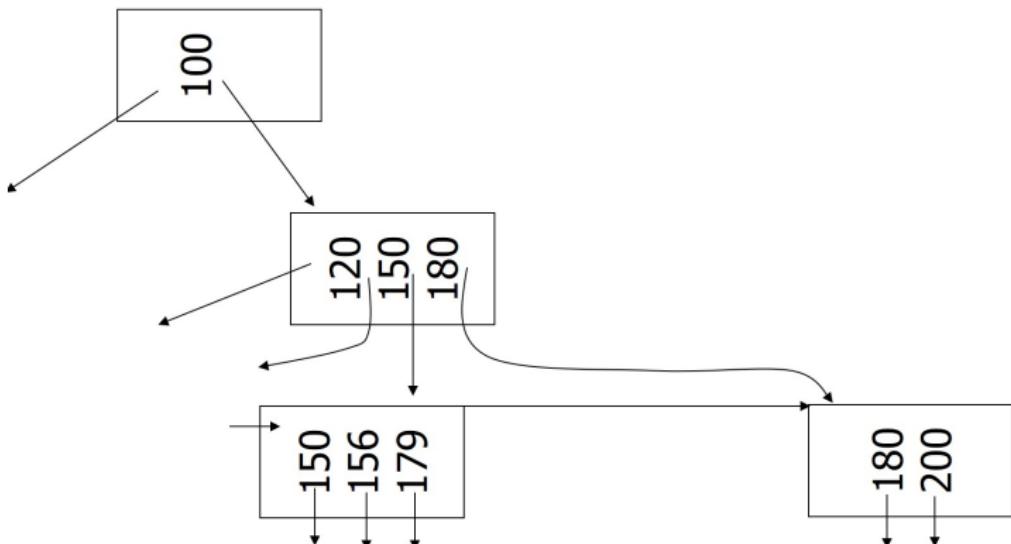
b) Insert key = 7, n = 3



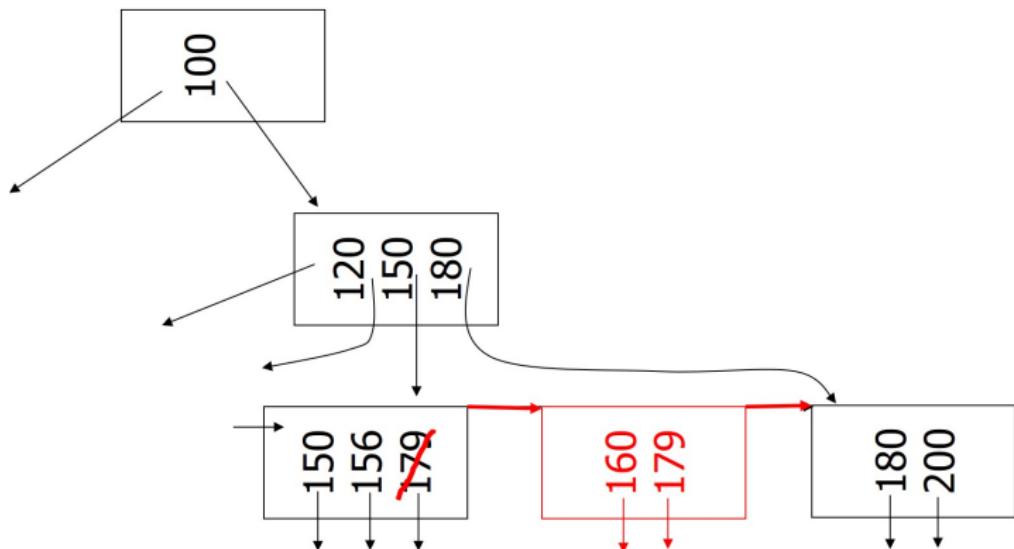
b) Insert key = 7, n = 3



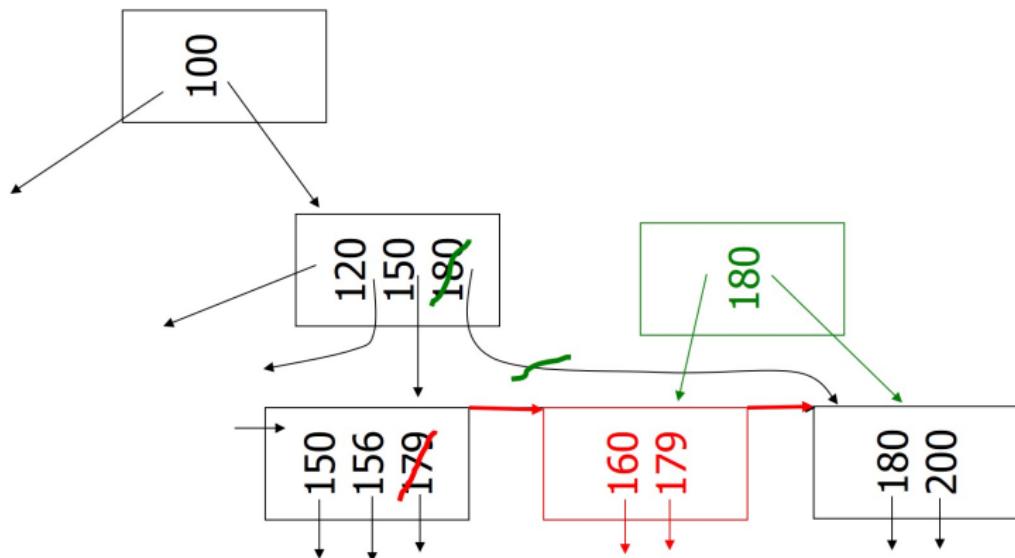
c) Insert key = 160, n = 3



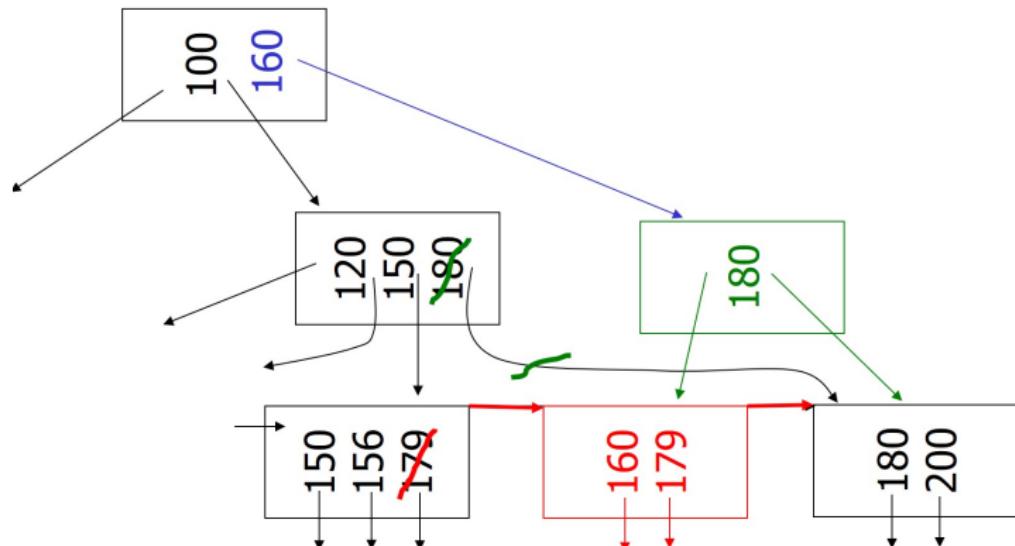
c) Insert key = 160, n = 3



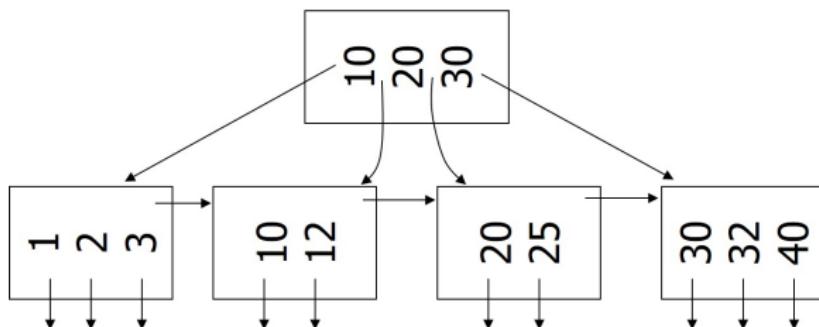
c) Insert key = 160, n = 3



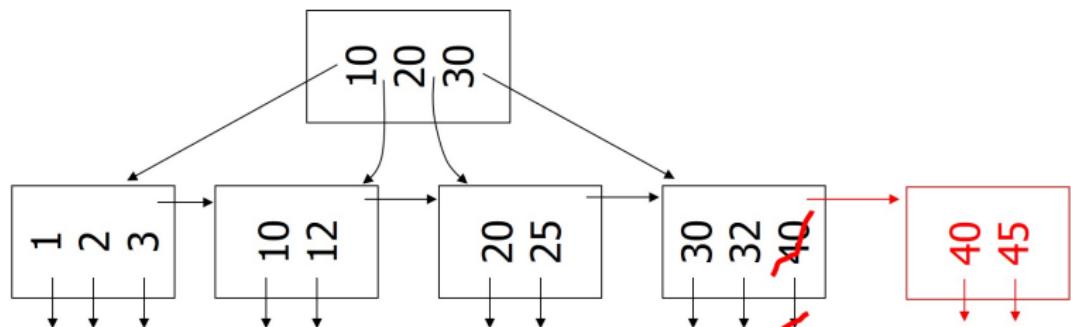
c) Insert key = 160, n = 3



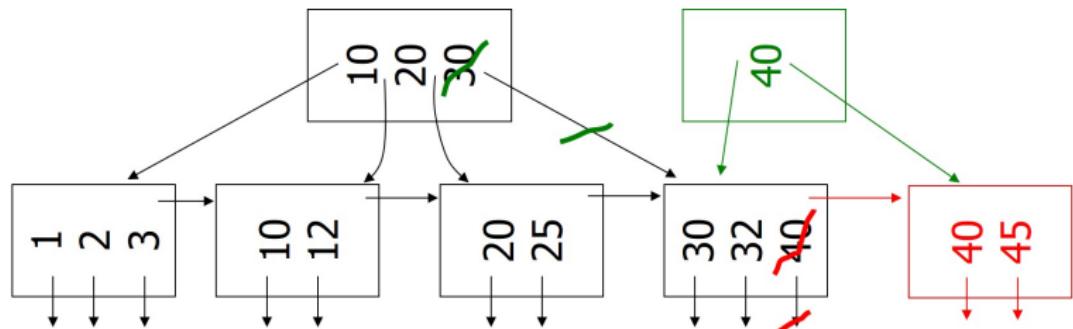
d) New root, insert 45,  $n = 3$



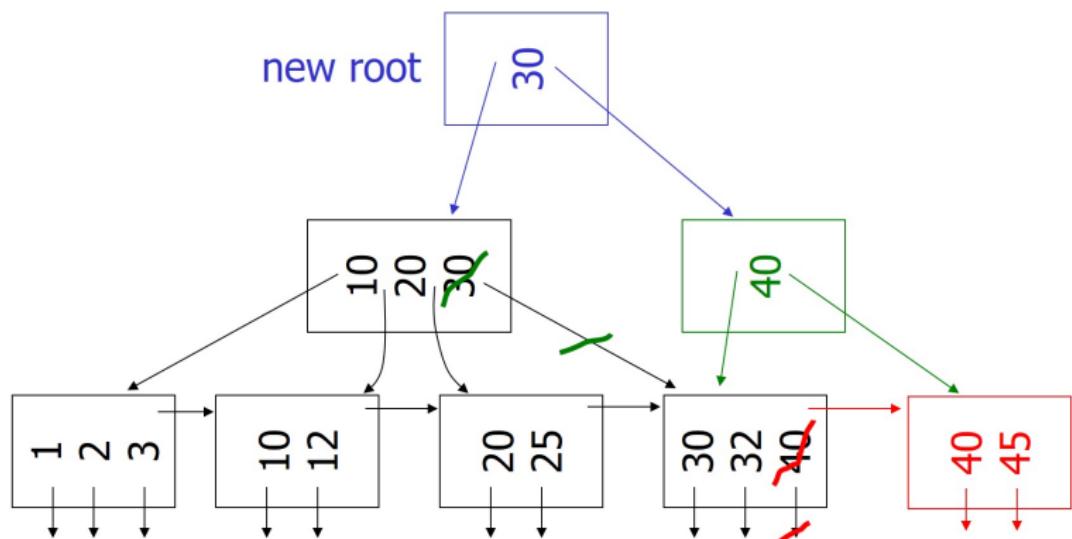
d) New root, insert 45,  $n = 3$



d) New root, insert 45,  $n = 3$



d) New root, insert 45,  $n = 3$



# Insertion Algorithm

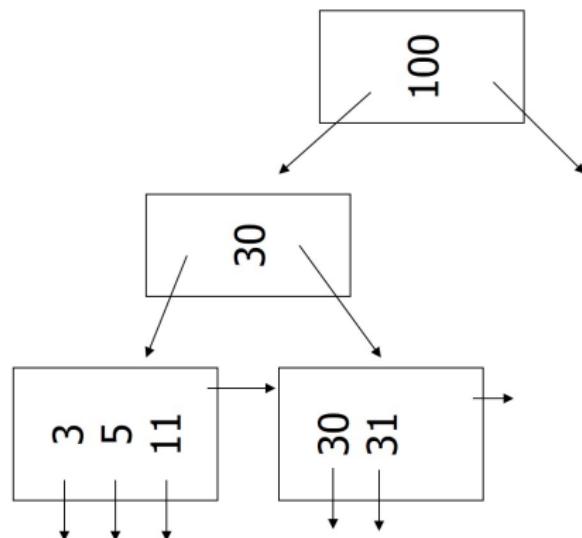
To Insert Record with key k

- ① Find correct leaf node L for k
- ② Add new entry into L in sorted order:
  - If L has enough space, the operation done.
  - Otherwise
    - Split L into two nodes L and L<sub>1</sub>.
    - Redistribute entries evenly and **copy up** middle key (new leaf's smallest key)
    - Insert index entry pointing to L<sub>1</sub> into parent of L
- ③ **inner node:** If, during the redistribution and copying of the middle key, the parent node of L becomes full, a similar split operation occurs for the parent node.
  - Entries are redistributed evenly between the parent node and a new node created as a result of the split.
  - The middle key from this split is **pushed up** further to the parent's parent node.
  - This process continues recursively up the tree until it reaches the root node.

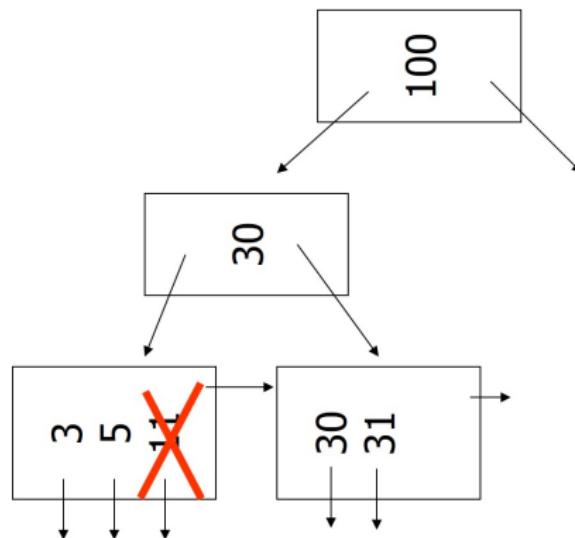
# Deletion from B<sup>+</sup>-Tree

- a) Simple case
- b) Coalesce with neighbor (sibling)
- c) Re-distribute keys
- d) Cases b) or c) at **non-leaf**

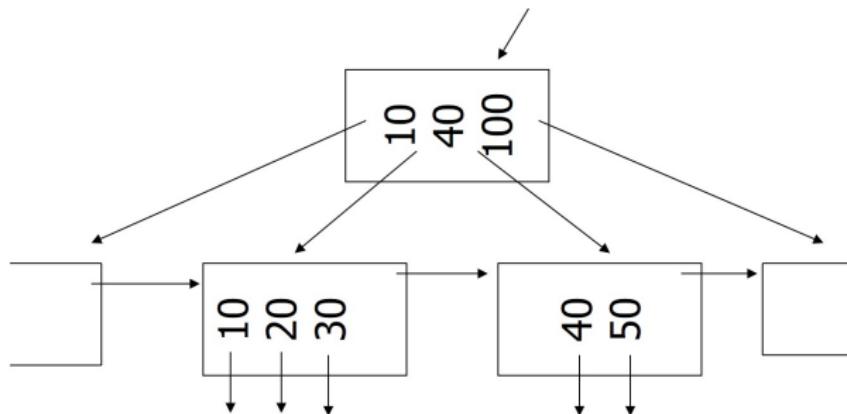
a) Delete key = 11, n = 3



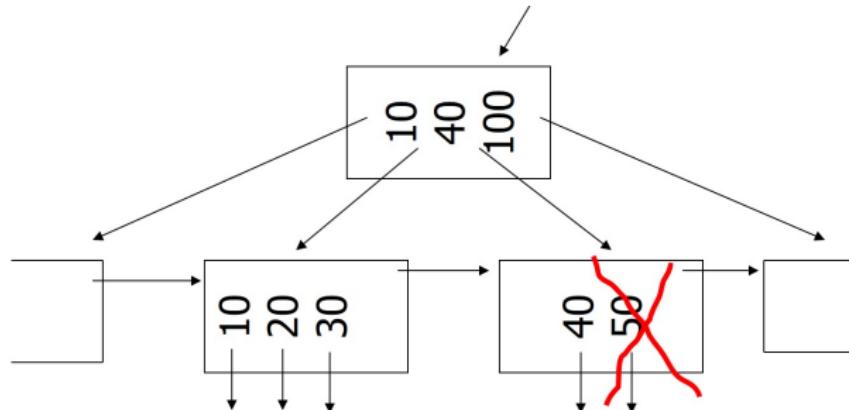
a) Delete key = 11, n = 3



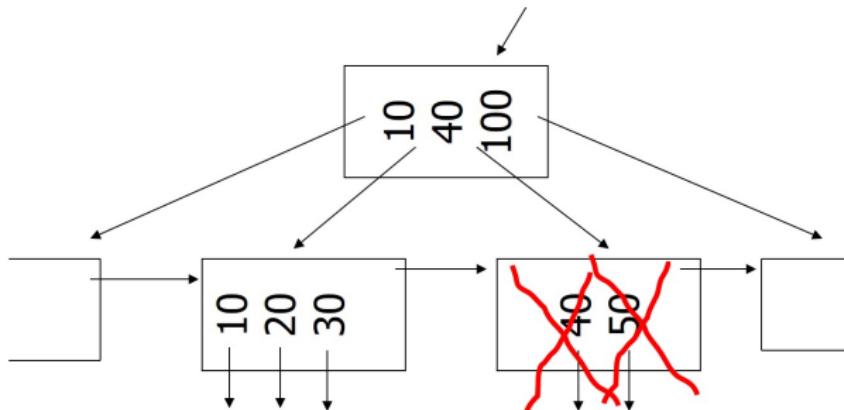
b) Coalesce with sibling: Delete 50,  $n = 4$



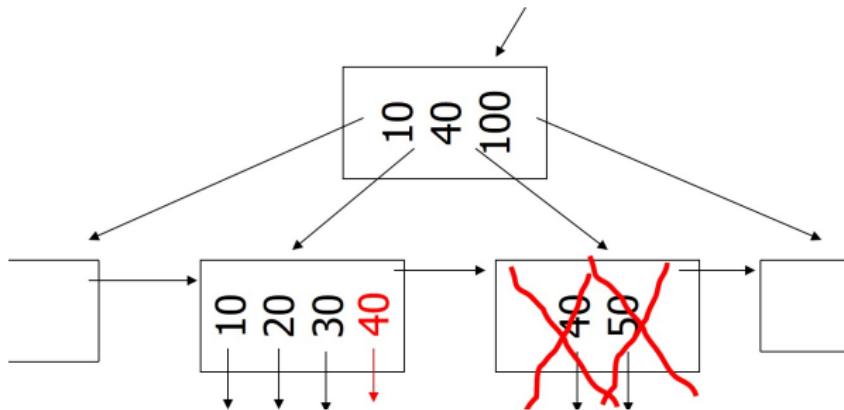
b) Coalesce with sibling: Delete 50,  $n = 4$



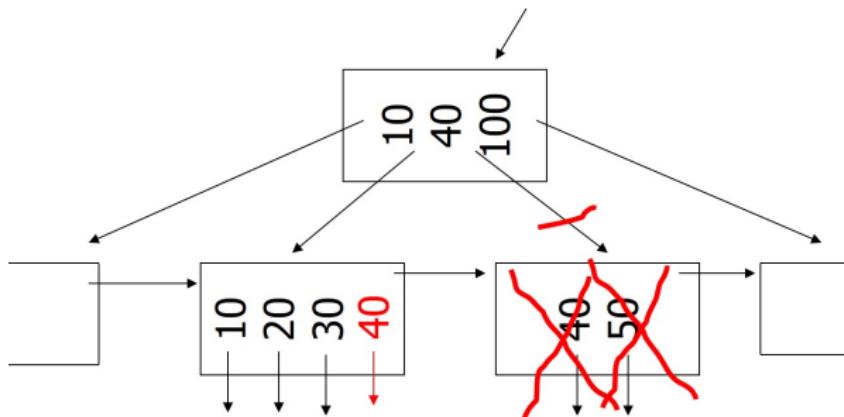
b) Coalesce with sibling: Delete 50,  $n = 4$



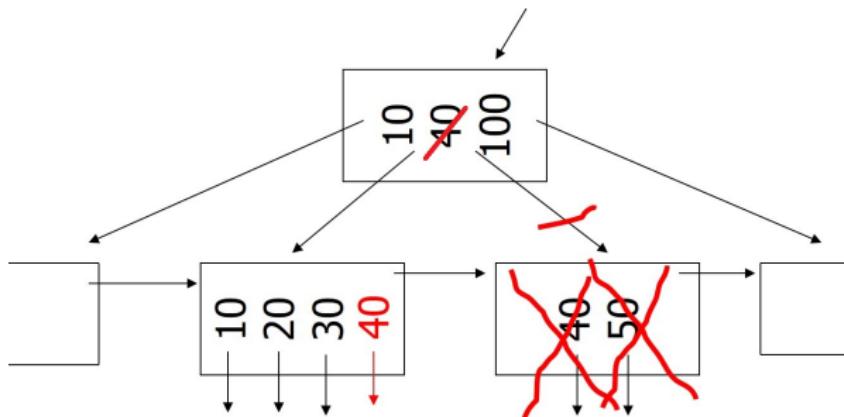
b) Coalesce with sibling: Delete 50,  $n = 4$



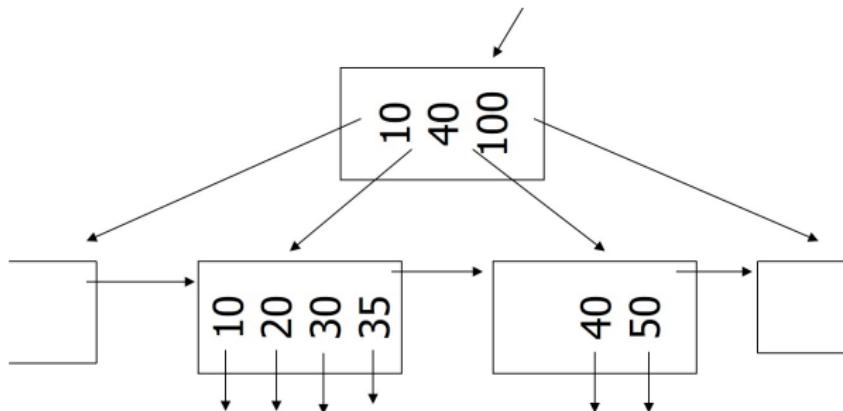
b) Coalesce with sibling: Delete 50,  $n = 4$



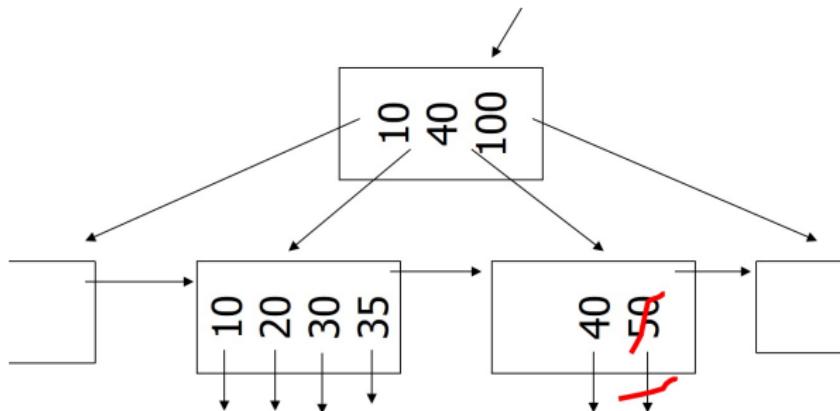
b) Coalesce with sibling: Delete 50,  $n = 4$



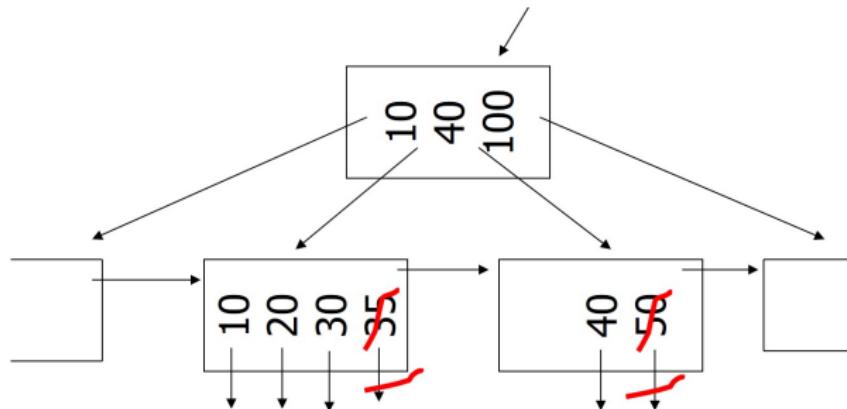
c) Redistribute keys: Delete 50,  $n = 4$



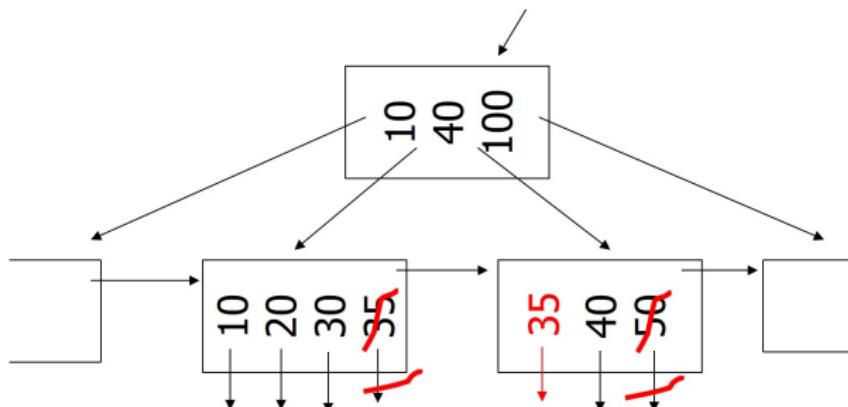
c) Redistribute keys: Delete 50,  $n = 4$



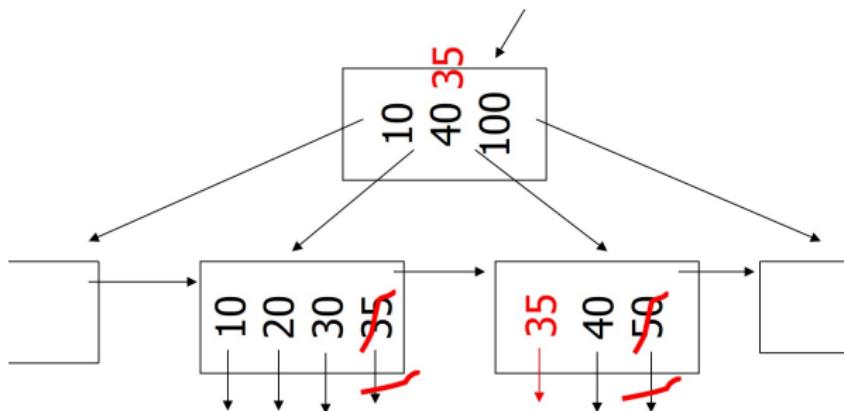
c) Redistribute keys: Delete 50,  $n = 4$



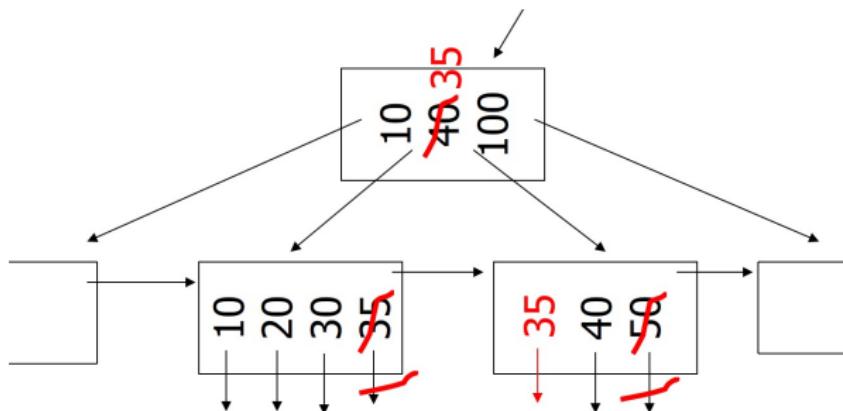
c) Redistribute keys: Delete 50,  $n = 4$



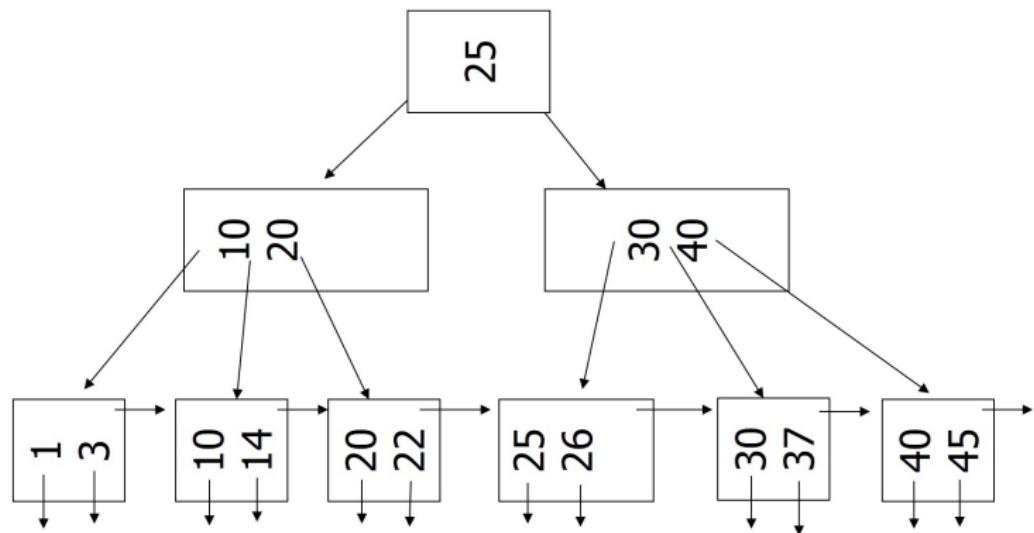
c) Redistribute keys: Delete 50,  $n = 4$



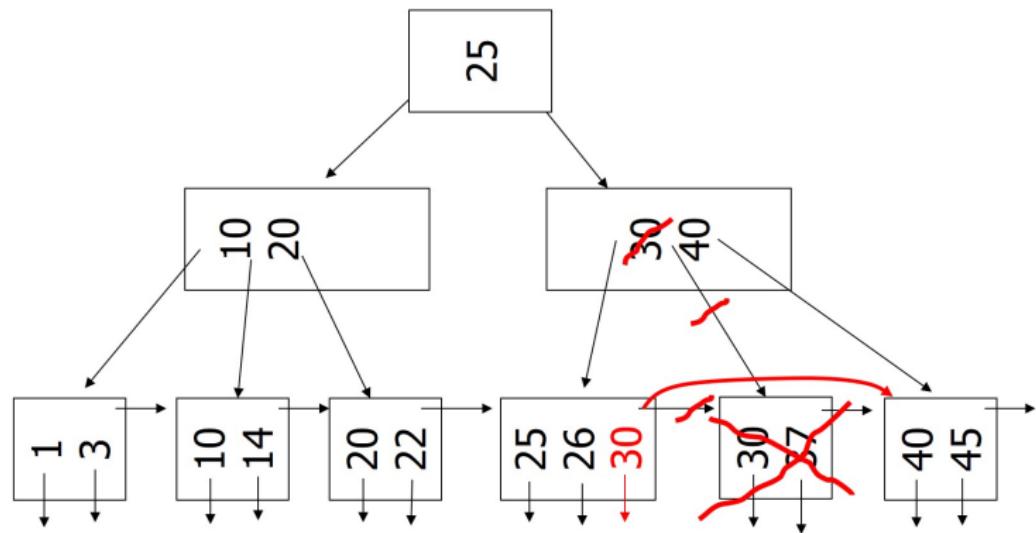
c) Redistribute keys: Delete 50,  $n = 4$



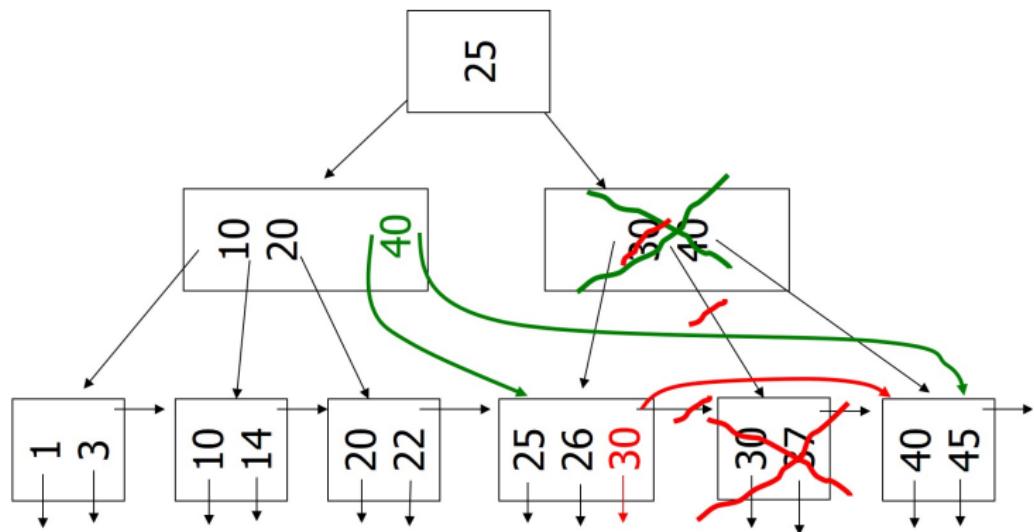
d) Non-leaf Coalesce: Delete 37,  $n = 4$



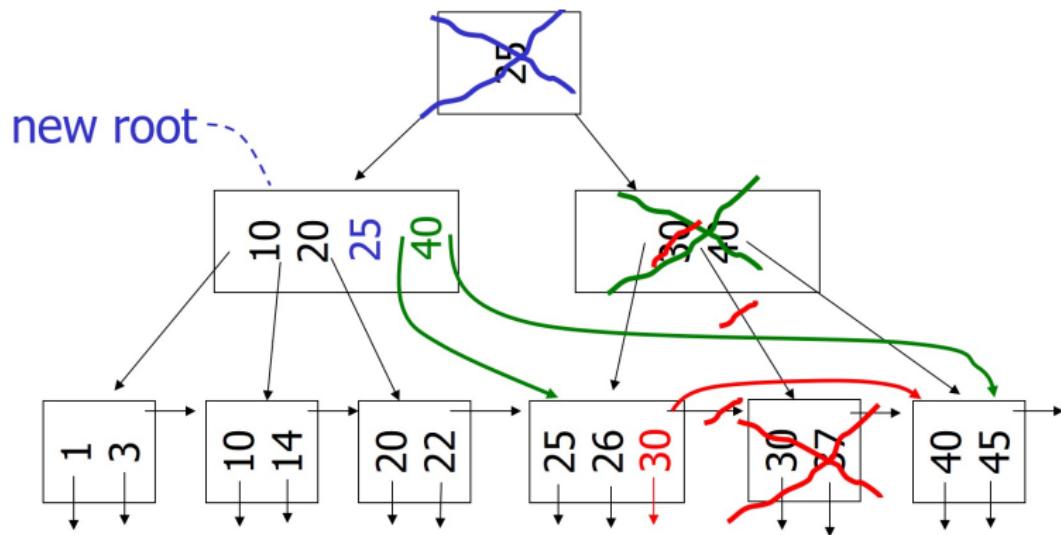
d) Non-leaf Coalesce: Delete 37,  $n = 4$



d) Non-leaf Coalesce: Delete 37,  $n = 4$



d) Non-leaf Coalesce: Delete 37,  $n = 4$



# Deletion Algorithm

To Delete record with key  $k$

- ① Find correct leaf  $L$ .
- ② Remove the entry:
  - If  $L$  is at least half full, the operation is **done**.
  - Otherwise
    - Attempt to **redistribute** entries from  $L$  by borrowing from its sibling node.
    - *If redistribution succeeds, and  $L$  is now at least half full, the operation is done.*
    - If redistribution fails, **merge**  $L$  with its sibling.
    - After the merge, the middle key (which separated  $L$  and its sibling) is removed from the parent node.
    - If the parent node becomes less than half full after removing the middle key, you may need to perform a similar redistribution or merge operation for the parent node.
- Note
  - *$B^+$ -Tree removes a key from a leaf node but not necessary from the non-leaf/inner nodes.*

# B<sup>+</sup>-Tree deletions in practice

- Often, coalescing is not implemented
  - Too hard and not worth it!
  - Assumption: nodes will fill up in time again

# Splitting or Merging nodes

- When splitting or merging nodes follow these conventions:
  - **Leaf Split:** In case a leaf node needs to be split during insertion and  $n$  is even, the left node should get the extra key. E.g, if  $n = 2$  and we insert a key 4 into a node [1,5], then the resulting nodes should be [1,4] and [5]. For odd values of  $n$  we can always evenly split the keys between the two nodes. In both cases the value inserted into the parent is the smallest value of the right node.
  - **Non-Leaf Split:** In case a non-leaf node needs to be split and  $n$  is odd, we cannot split the node evenly (one of the new nodes will have one more key). In this case the “middle” value inserted into the parent should be taken from the right node. E.g., if  $n = 3$  and we have to split a non-leaf node [1,3,4,5], the resulting nodes would be [1,3] and [5]. The value inserted into the parent would be 4
  - **Node Underflow:** In case of a node underflow you should first try to redistribute values from a sibling and only if this fails merge the node with one of its siblings. Both approaches should prefer the left sibling. E.g., if we can borrow values from both the left and right sibling, you should borrow from the left one.

## Comparison: B<sup>+</sup>-Tree vs. indexed sequential file

- Ref #1: Held & Stonebraker, “B-Trees Re-examined”, CACM, Feb. 1978
- Ref #2: M. Stonebraker, “Retrospection on a database system”, TODS, June 1980

# Comparison: B<sup>+</sup>-Tree vs. indexed sequential file

## B<sup>+</sup>tree

- Consumes more space, so lookup slower
- Each insert/delete potentially restructures
- Build-in restructuring
- Predictable performance

## indexed seq. file

- Less space, so lookup faster
- Inserts managed by overflow area
- Requires temporary restructuring
- Unpredictable performance

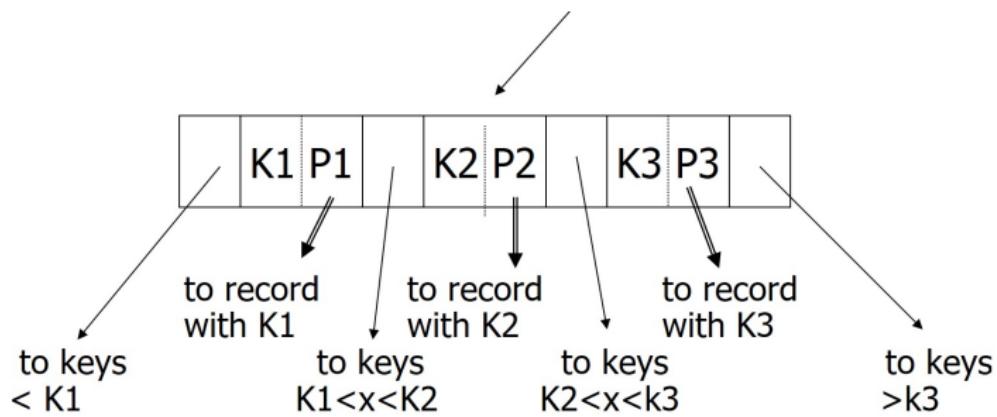
# Variation on B<sup>+</sup>-Tree: B-tree (no +)

- Avoid duplicate keys
  - Unlike B<sup>+</sup>-Trees, B-Trees do not allow duplicate keys. Each key appears only once in the tree.
  - More space-efficient, since each key only appears once in the tree
- Space Efficiency
  - B-Trees can be more space-efficient compared to B<sup>+</sup>-Trees because they don't need to store data pointers only in the leaf nodes.
  - in B<sup>+</sup>-Trees, data pointers can appear in both non-leaf (internal) and leaf nodes.
- Record Pointers in Non-Leaf Nodes
  - Have record pointers in non-leaf nodes which can reduce the need to traverse all the way to the leaf nodes to find data associated with a key.
  - This can potentially speed up lookups.
- In a B-tree, all search keys and corresponding data pointers are represented somewhere in the tree, but in a B<sup>+</sup>-tree, all data pointers appear in the leaves (sorted from left to right)
  - In B<sup>+</sup>-tree, inner nodes only guide the search process.

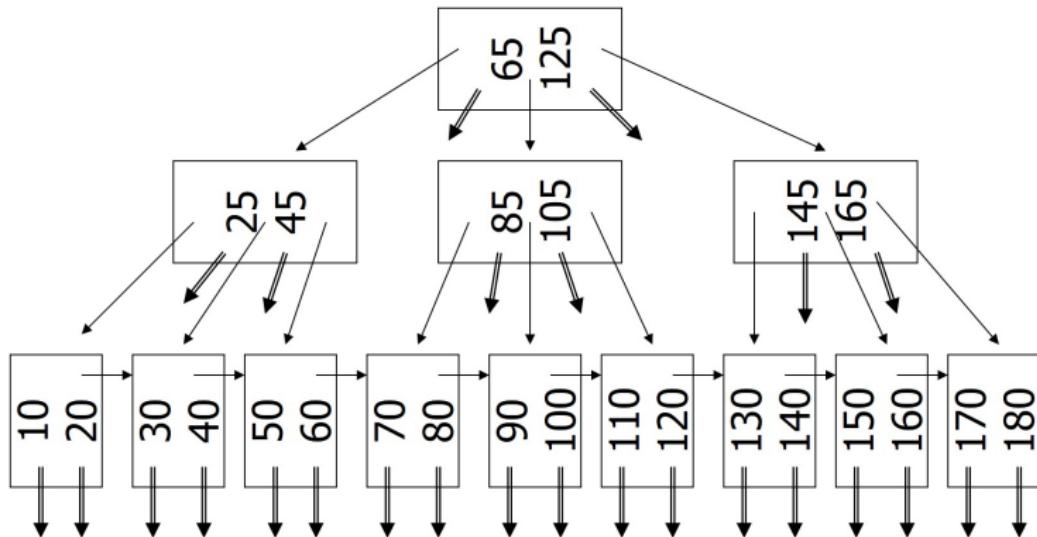
## Variation on B<sup>+</sup>-Tree: B-tree (no +)

- **B-trees** can be more expensive to update in a multi-threaded environment compared to B<sup>+</sup>-Trees.
- This is because updates might involve modifying non-leaf nodes that can be shared among multiple threads, potentially leading to more complex and contention-prone operations.

## Variation on B<sup>+</sup>-Tree: B-tree (no +)



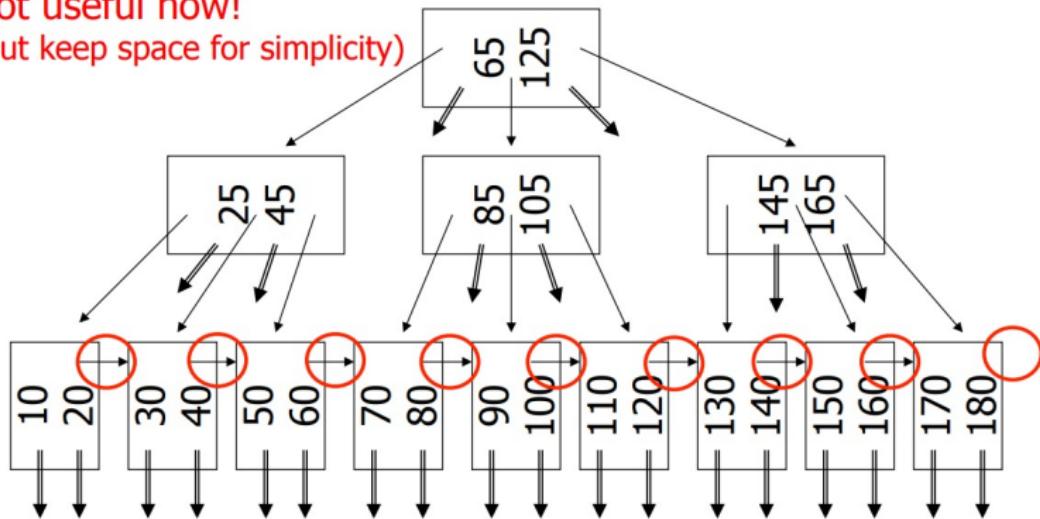
## B-tree example: $n = 2$



- The leaf nodes in a B-tree is not linked
- i.e., the last record pointer in a leaf node is not used

## B-tree example: $n = 2$

- sequence pointers  
not useful now!  
(but keep space for simplicity)



## B-tree

- B-trees have faster lookup than B<sup>+</sup>-trees
- in B-tree, non-leaf & leaf different sizes
- in B-tree, deletion more complicated
  - ⇒ B<sup>+</sup>trees preferred

But, note:

- If **blocks** are fixed size (due to disk and buffering restrictions)
- Then lookup for B<sup>+</sup>-Tree is **actually better**

# Example

- Consider a DBMS that has the following characteristics:
  - Pointers 4 bytes
  - Keys 4 bytes
  - Blocks 100 bytes
- Compute the maximum number of records we can index with:
  - a) 2-level B-tree
  - b) 2-level B<sup>+</sup>-Tree

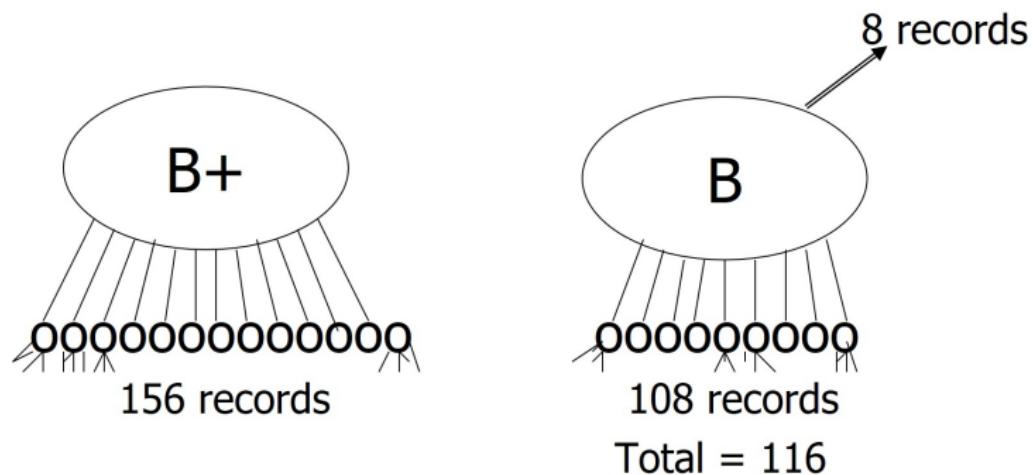
## B-tree

- Find largest integer value of  $n$  such that  $4n + 4(2n + 1) \leq 100$   
 $n = 8$
- Root has 8 keys + 8 record pointers + 9 children pointers  
 $= 8 \times 4 + 8 \times 4 + 9 \times 4 = 100\text{bytes}$
- Each of 9 children: 12 records pointers (+12 keys)  
 $= 12 \times (4 + 4) + 4 = 100 \text{ bytes}$
- 2-level B-tree, maximum # of records  $= 12 \times 9 + 8 = 116$

## B<sup>+</sup>-Tree

- Find largest integer value of  $n$  such that  $4n + 4(n + 1) < 100$   
 $n = 12$
- Root has 12 keys + 13 children pointers  
 $= 12 \times 4 + 13 \times 4 = 100\text{bytes}$
- Each of 13 children: 12 records pointers (+12 keys)  
 $= 12 \times (4 + 4) + 4 = 100\text{ bytes}$
- 2-level B<sup>+</sup>-Tree, maximum # of records  $= 13 \times 12 = 156$

So,

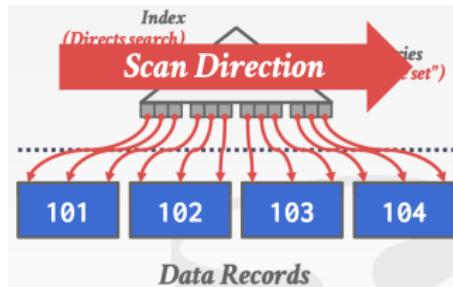


# Data Organization

- A table can be stored in two ways:
  - **Heap-organized storage:** Organizing rows in no particular order.
  - **Index-organized storage:** Organizing rows in primary key order.
- Types of indexes:
  - **Clustered index:** Organizing rows in a primary key order.
  - **Unclustered index:** Organizing rows in a secondary key order.
- The table is stored in the sort order specified by the primary key.
  - Can be either heap- or index-organized storage.
- Some DBMSs always use a clustered index.
  - If a table doesn't contain/explicitly defined a primary key, the DBMS will automatically make a hidden rowID primary key.
  - This hidden primary key is used internally by the DBMS to uniquely identify rows when there is no explicit primary key.
  - It ensures that each row can be uniquely identified even if the table doesn't have a natural primary key.
  - e.g., Oracle, Microsoft SQL Server
- Other DBMSs cannot use them at all (e.g., PostgreSQL, SQLite).

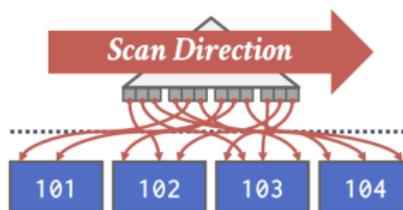
# Clustered Index

- Tuples are kept sorted on disk using the order specified by primary key.
- If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.
- Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.



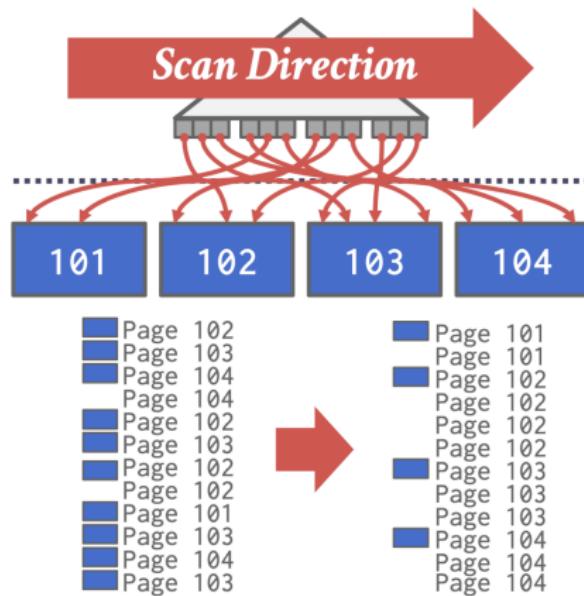
# Index Scan Page ID Sorting

- Retrieving tuples in the order that appear in an unclustered index can be inefficient because the index itself doesn't store the actual data, only pointers to the data location.
- Accessing individual data pages based on an index order requires separate I/O operations for each page, leading to increased disk reads.



# Index Scan Page ID Sorting

- The DBMS can use the page IDs from the index scan to efficiently retrieve the required tuples.
- By first identifying and sorting the page IDs based on the desired order, the DBMS optimizes disk access.
- This allows it to fetch multiple data pages in a single I/O operation, improving performance.



# Filtering Tuples/Selection Conditions

- The DBMS can use a  $B^+$ -Tree index if the query/filter provides any of the attributes of the search key.
- Example: Index on  $\langle a, b, c \rangle$ 
  - Supported:  $(a=5 \text{ AND } b=3)$
  - Supported:  $(b=3)$ .
- Not all DBMSs support this.
  - The ability to utilize a  $B^+$ -Tree index for partial or compound selection conditions may vary depending on the specific DBMS and its query optimization capabilities.
- For hash index, we must have **all attributes** in search key.
- *$B^+$ -Tree indexes can leverage prefix filtering for leading search key components. This means they can efficiently find records where the initial part of the search key matches the query condition.*

# B<sup>+</sup>-Tree Design Choices/Decisions<sup>1</sup>

- Several design choices and decisions need to be made when implementing B<sup>+</sup>-Trees.
- Some key design considerations:
  - Node Size
  - Merge Threshold
  - Variable Length keys
  - Non-Unique Indexes
  - Intra-Node Search

---

<sup>1</sup>Modern B-Tree Techniques By Goetz Graefe

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.7269&rep=rep1&type=pdf>

## B<sup>+</sup>-Tree: Node Size

- The choice of node size should be tailored to the specific characteristics of the storage medium and the workload of the database system.
- The optimal node size for a B<sup>+</sup>-Tree depends on the speed of the disk.
- Larger node sizes amortize the cost of reading from disk over more key/value pairs, reducing the number of disk I/O operations.
- The slower the disk, then the larger the optimal node size for a B<sup>+</sup>-Tree<sup>1</sup>.
  - HDD: ~ 1 MB
  - SSD: ~ 10KB
  - In-Memory: ~ 512B
- Optimal sizes can vary depending on the workload
  - Some workloads may be more scan-heavy versus having more single key look-ups.  
→ Leaf Node Scans (OLAP) vs. Root-to-Leaf Traversals (OLTP)
    - *as point queries would prefer as small a page as possible to reduce the amount of unnecessary extra info loaded, while a large sequential scan might prefer large pages to reduce the number of fetches it needs to do.*

---

<sup>1</sup> For example, nodes stored on hard drives are usually on the order of megabytes in size to reduce the number of seeks needed to find data and amortize the expensive disk read over a large chunk of data, while in-memory databases may use page sizes as small as 512 bytes in order to fit the entire page into the CPU cache as well as to decrease data fragmentation.

## B<sup>+</sup>-Tree: Node Size

- *Larger node sizes in B<sup>+</sup>-Trees allow DBMS to read more data from disk in a single operation, and this amortizes the relatively high cost of disk I/O over multiple key-value pairs, making the overall data retrieval process more efficient. It's an optimization strategy to minimize the impact of slower disk access times in scenarios where data is stored on disk, such as in large databases.*

## B<sup>+</sup>-Tree: Merge Threshold

- The merge threshold in a B<sup>+</sup>-Tree is a crucial parameter that determines when underflowed nodes (nodes with too few key-value pairs) should be merged during delete operations.
- While the standard rule is to merge when a node becomes half full or less (underflow node), there are scenarios where temporarily violating this rule can be beneficial.
- Some DBMS do not always merge when it's half full.
- Delaying the merge operation can reduce the amount of immediate reorganization needed during delete operations.
- Merging nodes requires redistributing keys and values, which can be an expensive operation in terms of time and resources.
- Instead of immediately merging underflowed nodes, some DBMS may opt to allow underflows to accumulate and **periodically rebuild** the entire B<sup>+</sup>-Tree structure to rebalance it.

# B<sup>+</sup>-Tree: Variable Length keys

- **Approach 1: Pointers**

- Store keys as pointers to the tuple's attribute (very rarely used).

- **Approach 2: Variable-length Nodes**

- We could also still store the keys like normal and allow for variable length nodes.
  - The size of each node in the B<sup>+</sup>-Tree can vary, but requires careful memory management. This approach is also rare.

- **Approach 3: Padding**

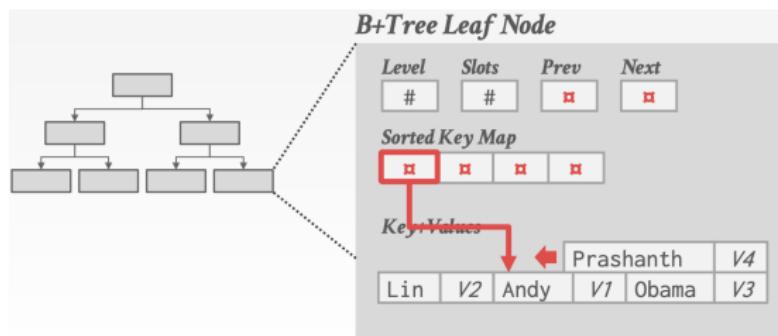
- Instead of varying the key size, we could set each key's size to the size of the maximum key and pad out all the shorter keys.
  - In most cases this is a massive waste of memory, so you don't see this used by anyone either.

- **Approach 4: Key Map/Indirection**

- An array of pointers is embedded within the node, which maps to the **variable-length key+value** pairs within the node.
    - Each pointer in the array represents an offset to a key inside the node itself
  - This is similar to slotted pages discussed before. This is the most common approach.

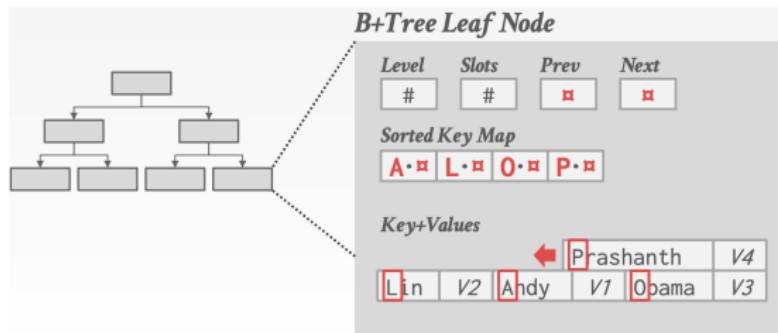
# Key Map/Indirection

- Replace the keys with an index to the key-value pair in a separate dictionary.
  - Each pointer will be an offset to a key inside the node itself
  - The sorted key map is sorted based on the value of the keys



# Key Map/Indirection

- Due to the small size of the dictionary index value, there is enough space to place a prefix of each key alongside the index,
- Potentially allowing some index searching and leaf scanning to not even have to chase the pointer
  - (if the prefix is at all different from the search key)



# B<sup>+</sup>-Tree: Non-Unique Indexes

B<sup>+</sup>-Trees can deal with non-unique indexes by duplicating keys or storing value lists:

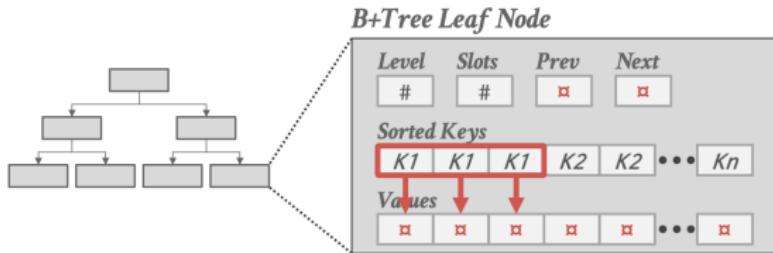
- **Approach 1: Duplicate Keys**

- Use the same leaf node layout but store duplicate keys multiple times.  
(Most common)
- Each occurrence of a non-unique key is treated as a separate entry in the leaf node.
- When you insert a new entry with a non-unique key, it's simply added as another entry with the same key value in the appropriate leaf node.
- Searching for a non-unique key returns all matching entries, which may have the same key but different associated values.

- **Approach 2: Value Lists**

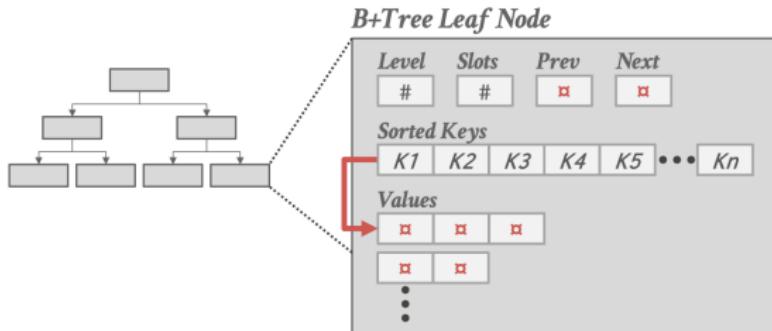
- Store each key only once and maintain a linked list of unique values.
- When you insert a new entry with a non-unique key, the system appends the new value to the linked list associated with that key.
- Searching for a non-unique key involves traversing the linked list to retrieve all matching values.

# Non-Unique Indexes: Duplicate Keys



- In practice, duplicate keys are more commonly used for non-unique indexes due to their simplicity and efficiency in most scenarios. However, it can lead to larger storage requirements because the keys are duplicated.

# Non-Unique Indexes: Value Lists



- *Reduce storage redundancy by storing keys only once and maintaining linked lists of associated values. However, they may involve extra traversal to access all values associated with a key, potentially impacting performance.*

# Duplicate Keys

- When dealing with non-unique keys in a B<sup>+</sup>-Tree, DBMSs often need a mechanism to distinguish between multiple entries with the same key.
- There are two approaches to duplicate keys in a B<sup>+</sup>-Tree:
  - **Approach 1: Append Record IDs**
    - Include a unique identifier, such as a record ID, along with each key (as a part of the key) in the leaf nodes.
    - The record ID ensures that each entry with the same key is distinguishable because even if the keys are the same, the associated record IDs are unique.
  - **Approach 2: Overflow Nodes**
    - Instead of duplicating the keys or appending record IDs, DBMS allow the leaf node to *spill* into overflow nodes.
    - Although no redundant information is stored, this approach is more complex to maintain and modify.

# B<sup>+</sup>-Tree: Intra-Node Search

- Method used to search for a specific key within a node can significantly impact the search efficiency:
- **Approach 1: Linear**
  - Scan the key/value entries in the node from beginning to end. Stop when you find the key that you are looking for.
  - Modern processors can use SIMD<sup>1</sup> instructions to vectorize comparisons, which can improve search performance.
  - This does not require the key/value entries to be pre-sorted.
- **Approach 2: Binary**
  - Jump to the middle key, and then pivot left/right depending on whether that middle key is less than or greater than the search key.
  - This requires the key/value entries to be pre-sorted.
- **Approach 3: Interpolation**
  - Approximate the starting location of the search key based on the known low/high key values in the node. Then perform linear scan from that location.
  - This requires the key/value entries to be pre-sorted.

---

<sup>1</sup> Single instruction, multiple data (SIMD): computational technique for processing a number of data values (generally a power of two) using a single instruction, with the data for the operands packed into special wide registers. One instruction can therefore do the work of many separate instructions

# B<sup>+</sup>-Tree Optimizations

- Prefix Compression
- Suffix Truncation
- Bulk Insert
- Pointer Swizzling

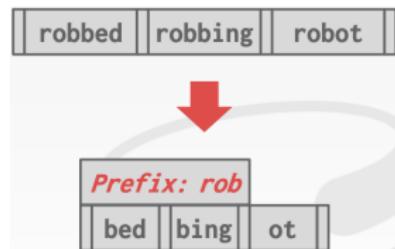
# Prefix Compression

- Sorted keys in the same leaf node are likely to have the same prefix.
- Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
  - Many variations.



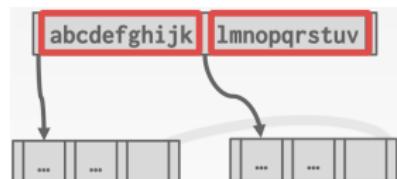
# Prefix Compression

- Sorted keys in the same leaf node are likely to have the same prefix.
- Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
  - Many variations.
    - *Variations may include how the common prefix is stored and how it is used in search operations.*



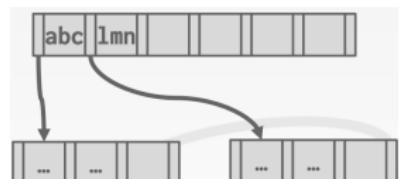
# Suffix Truncation

- The keys in the inner nodes are only used to “direct traffic”.
  - We don’t actually need the entire key.
- Store a minimum prefix that is needed to correctly route probes into the index.



# Suffix Truncation

- The keys in the inner nodes are only used to “direct traffic”.
  - We don’t actually need the entire key.
- Store a minimum prefix that is needed to correctly route probes into the index.



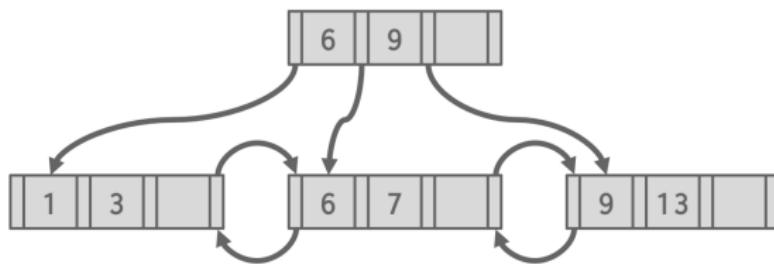
# Bulk Inserts

- The fastest/best way to build a  $B^+$ -Tree is to first sort the keys and then build the index from the bottom up.
- This will be faster than inserting one-by-one since there are no splits or merges.

# Bulk Inserts

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13



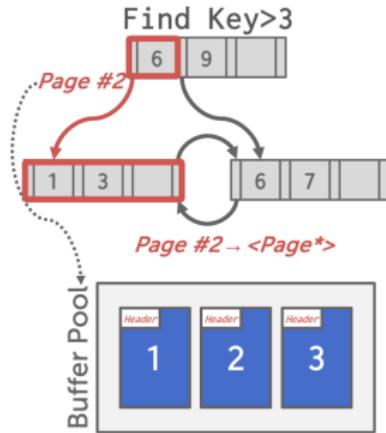
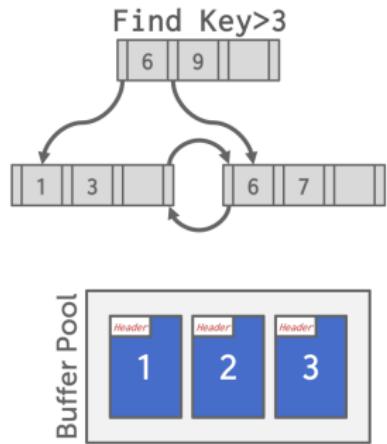
# Pointer Swizzling<sup>1</sup>

- Nodes use **page ids** to reference other nodes in the index.
  - The DBMS must get the memory location from the page table during traversal.
  - If a page is pinned in the buffer pool, then we can store **raw pointers** instead of **page ids**.
  - This avoids address lookups from the page table.
- 
- *Pointer swizzling involves temporarily replacing or swizzling disk block pointers with in-memory pointers during a  $B^+$ -Tree node's access.*

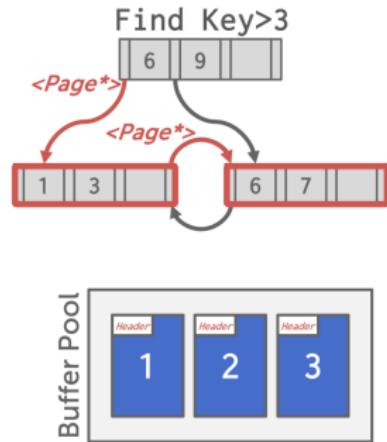
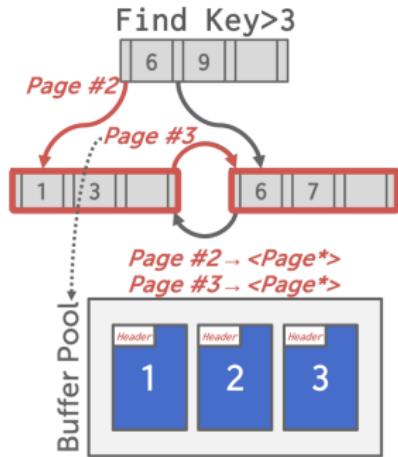
---

<sup>1</sup> The term “swizzling” generally implies a dynamic rearrangement or transformation of data to optimize access, processing, or storage. Its application depends on the specific context and the objectives of the optimization.

# Pointer Swizzling



# Pointer Swizzling



# Conclusion

- The venerable **B<sup>+</sup>-Tree** is always a good choice for your DBMS.

# Reading

- Chapter 4: Index Structures (uploaded on course Blackboard)
- Database System Concepts: Chapter 11 (6th Edition)/ Chapter 14 (7th Edition)
- Modern B-Tree Techniques, By Goetz Graefe

Next

- Hash Tables