DTS207TC Database Development and Design

Lecture 10

Chap 14. Indexing

Di Zhang, Autumn 2025

*Titles with * will not be assessed*

Outline

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- Review of Algorithms
- Ordered Indices
- B^+-Tree Index

The Core Problem: Search

- Problem: Find a specific key in a large collection of data.
- Naive Solution: Linear Search
  - Traverse the collection one element at a time.
  - Time Complexity: $O(n)$
  - Inefficient for large datasets.
- Our goal is to do better than $O(n)$.

The Array & Binary Search

- Prerequisite: Sorted Array
  - Elements are stored in contiguous memory, sorted by key.
- Algorithm: Binary Search
  - Repeatedly divides the search interval in half.
  - Time Complexity: O(log n)
  - Massive improvement over O(n).

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

The Limitation of Arrays: Insertion/Deletion

- Problem: Maintaining sorted order after changes is costly.
- Insertion:
  - Find the correct position (O(log n) with binary search).
  - Shift all subsequent elements to the right to make space (O(n)).
- Deletion:
  - Find the element (O(log n)).
  - Shift all subsequent elements to the left to fill the gap (O(n)).
- Conclusion: Sorted arrays are excellent for static data but expensive for dynamic data with frequent updates.

The Linked List

- Structure: A sequence of nodes, where each node contains:
  - data
  - A pointer (or reference) to the next node.
- Advantage: Efficient Insertion/Deletion
  - Once the position is found, insertion/deletion is O(1). Only pointers need updating. No shifting is required.

The Limitation of Linked Lists: Search

- Problem: Sequential Access
  - To find an element, you must start at the head and traverse the list.
  - Time Complexity: O(n)
- We lose the benefit of binary search.
  - Idea: Can we combine the fast search of a sorted array with the easy updates of a linked list?

Classic Interview Question*

- Solution (Tortoise and Hare):
  - We set two pointers, `slow` (the slow pointer) moves one step at a time, and `fast` (the fast pointer) moves two steps at a time.
  - If there is no cycle in the linked list, the `fast` pointer will reach the end (`null`) first.
  - If there is a cycle in the linked list, the `fast` pointer will enter the cycle first and keep moving within the cycle; the `slow` pointer will enter the cycle later. Since `fast` is faster than `slow`, they will eventually meet at some node inside the cycle.
  - If `fast` and `slow` meet, it indicates that the linked list has a cycle.

The Binary Search Tree (BST)

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- Structure:
  - Each node has at most two children: left and right.
  - The BST Property:
    - For any node, all keys in its left subtree are less than the node's key.
    - All keys in its right subtree are greater than the node's key.
- Search, Insert, Delete: O(h)
  - h is the height of the tree.
  - In a balanced tree, h = O(log n).

The Problem with BSTs: Degeneration

- What if data is inserted in sorted order?
  - The tree becomes a linear linked list.
  - Height h becomes n.
  - Search time degrades to O(n).

- We need a tree that stays balanced.

Beyond Binary: M-way Search Trees

- Core Idea: Increase the branching factor.
- A node can have up to M children.
- A node contains between ⌈ M/2⌉ -1 and M-1 keys (for balance).
- The keys in a node define ranges for its subtrees.
- Advantage: Very "bushy" and short trees.
  - Height is O(log(floor(M)) n).
  - This minimizes the number of nodes we need to visit during a search.

Key Takeaways & The Path Forward

- Sorted Array: Fast search (O(log n)), slow updates (O(n)).
- Linked List: Fast updates (O(1) after seek), slow search (O(n)).
- Binary Search Tree: Good balanced performance (O(log n)), but can degenerate to O(n).
- M-way Search Tree: A design for short, bushy trees that minimize search steps.

The ultimate solution for dynamic data combines these ideas:

A self-balancing M-way tree with linked lists at the bottom.

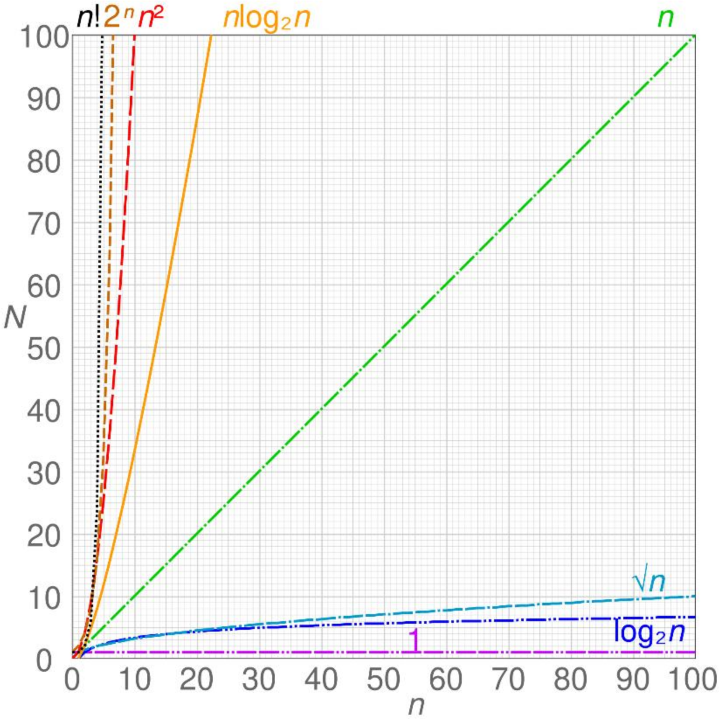
Next: We will see how B+ Trees masterfully integrate these concepts.

Algorithm Analysis: Understanding Time Complexity

- Big Idea: Time Complexity is a way to describe how the runtime of an algorithm grows as the input size increases.
  - It's not about measuring exact seconds, but about the growth rate.
- Why is it Important?
  - Helps us compare algorithms without running them on specific hardware.
  - Predicts how an algorithm will perform with large, real-world data sets.
  - Allows us to identify potential performance bottlenecks.
- Key Characteristic: It focuses on the worst-case scenario (Big O Notation).
  - This gives us a guarantee that the runtime won't be any worse.

The Language of Growth: Big O Notation

- Big O Notation (O): Describes the upper bound of growth. We care about the dominant term as n (input size) gets very large.
  - Example: $O(n^2 + n + 1)$ simplifies to $O(n^2)$.
- Common Complexity Classes (From Best to Worst):
  - O(1) - Constant Time: Runtime is independent of input size.
    - Example: Accessing an array element by index.
  - O(log n) - Logarithmic Time: Runtime grows very slowly as n increases.
    - Example: Binary search in a sorted list.
  - O(n) - Linear Time: Runtime grows proportionally with n.
    - Example: Finding the maximum value in an unsorted list.
  - O(n log n) - Linearithmic Time: Common in efficient sorting algorithms.
    - Example: Merge Sort, Quick Sort (average case).
  - $O(n^2)$ - Quadratic Time: Runtime grows with the square of n.
    - Example: Checking all pairs in a list (nested loops).
  - O(2^n) - Exponential Time: Runtime doubles with each additional element. Becomes unusable for large n.
    - Example: Naive recursive solution for the Fibonacci sequence.



From Code to Complexity & Practical Impact

- How to Calculate Time Complexity?
  - Identify the input size (n).
  - Count the basic operations in terms of n.
  - Find the fastest-growing term and drop constants.
    - Example: A single loop

```python
for i in range(n):  # This loop runs 'n' times
    print(i)        # This is a constant-time operation O(1)
```

    - Example: Nested loops

```python
for i in range(n):        # Runs 'n' times
    for j in range(n):    # For each i, runs 'n' times
        print(i, j)       # O(1)
```

- Why This Matters in the Real World:
  - O(n) vs O(n²): For n = 1,000,000, an O(n) algorithm might take ~1 second, while an O(n²) algorithm could take over 11 days!
  - Choosing the right algorithm is crucial for building scalable and responsive applications, especially in fields like data science, web development, and systems programming.

Operational Costs: Data Structure Efficiency

Data Structure	Access (By Key/Index)	Search (By Value)	Insertion	Deletion
Array / List	O(1)	O(n)	O(n) [at end: O(1)*]	O(n)
Stack / Queue	O(1) [Top/Front]	O(n)	O(1) [Push/Enqueue]	O(1) [Pop/Dequeue]
Linked List	O(n)	O(n)	O(1) [at head]	O(1) [at head]
Hash Table	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)
Balanced BST (e.g., AVL, Red-Black)	O(log n)	O(log n)	O(log n)	O(log n)

- Key Takeaways & Why This Matters:
  - Need fast access by index? Use an Array.
  - Need fast lookups/insertions/deletions by key? Use a Hash Table.
  - Need your data sorted and still have decent speed? Use a Balanced BST.
  - Trade-offs are everywhere: Arrays have fast access but slow insertions. Linked Lists have slow access but fast insertions at known positions.

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function". (won't be taught in this lecture)

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

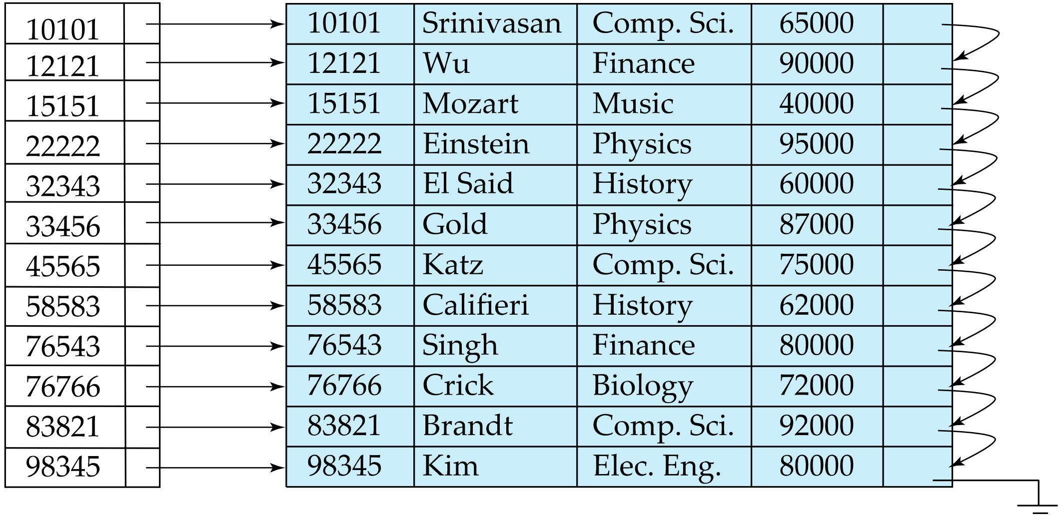
Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index.**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.

Dense Index Files

Xi'an Jiaotong-Liverpool University
西交利物浦大学

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Sparse Index Files

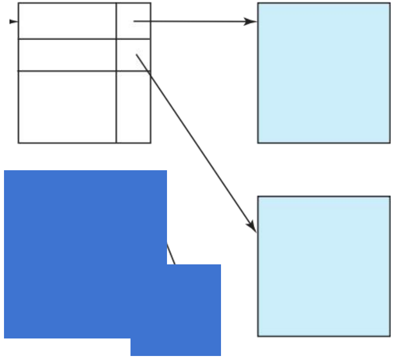
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

10101
32343
76766

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Sparse Index Files (Cont.)

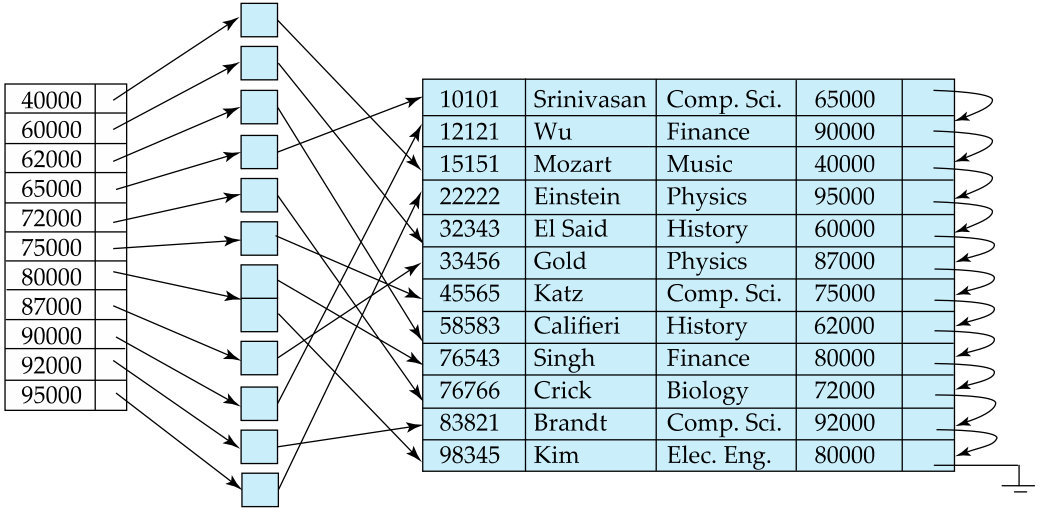
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**:
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



  - For unclustered index: sparse index on top of dense index (multilevel index)

Secondary Indices Example

- Secondary index on salary field of instructor



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

- Secondary indices have to be dense

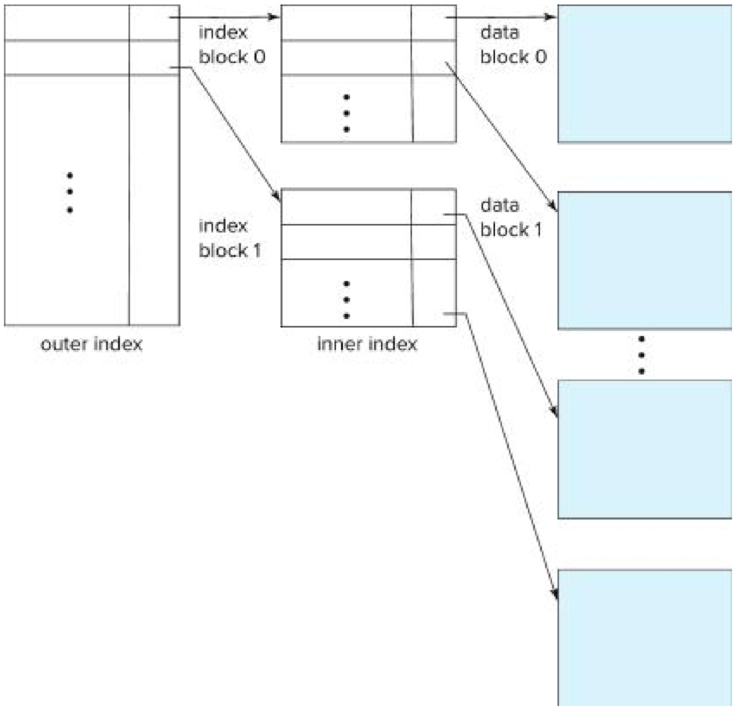
Clustering vs Nonclustering Indices

- Indices offer substantial benefits when searching for records.
- BUT: indices imposes overhead on database modification
  - when a record is inserted or deleted, every index on the relation must be updated
  - When a record is updated, any index on an updated attribute must be updated
- Sequential scan using clustering index is efficient, but a sequential scan using a secondary (nonclustering) index is expensive on magnetic disk
  - Each record access may fetch a new block from disk
  - Each block fetch on magnetic disk requires about 5 to 10 milliseconds

Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it
    - Indices are maintained as sequential files
    - Need to create space for new entry, overflow blocks may be required
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

The Starting Point: Binary Search Tree (BST)

- Core Concept:
  - Each node has at most two children: left and right.
  - For any node:
    - All keys in its left subtree are less than its key.
    - All keys in its right subtree are greater than its key.
- Advantages:
  - Simple structure.
  - Fast for in-memory operations: O(h) time for search, insert, delete, where h is the tree's height.
- The Critical Problem:
  - It can become unbalanced (e.g., if you insert sorted data).
  - In the worst case, it degrades into a linked list, and operations become O(n).
- Demo
  - https://www.cs.usfca.edu/~galles/visualization/BST.html

The In-Memory Solution: Balanced BST (e.g., AVL)

- Core Concept:
  - Self-balancing BSTs automatically maintain a height of O(log n) after every insertion and deletion.
  - They use rotation algorithms to fix imbalances.
- Advantages:
  - Guarantees efficient O(log n) time complexity for core operations.
  - The ideal choice for in-memory data storage and lookups (e.g., in Java TreeMap, C++ std::map).
- The New Bottleneck: Disk I/O
  - When data is too large for memory, it must live on disk.
  - Disk access is slow; the cost is dominated by the number of disk reads/writes.
  - Even with O(log n) comparisons, if each node is a separate disk block, you need O(log n) disk I/Os, which can be too high for large n.

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

The Shift: Thinking in Terms of Disk Blocks

- Key Insight:
  - Data is read from disk in chunks called pages or blocks (e.g., 4KB).
  - The goal is to minimize the number of disk I/Os, even if it means doing more computations in memory.
- The New Design Goal:
  - Instead of making the tree shorter in terms of nodes, make it shorter in terms of disk accesses.
  - How? Pack more keys into a single node that fits within one disk block.
  - This leads us from binary to multi-way trees.

The Bridge: B-Tree

- Core Concept:
  - A B-Tree of order m is a multi-way search tree with these **properties**:
    - Each node has at most m children.
    - Each internal node (except root) has at least ceil(m/2) children.
    - A node with k keys has exactly k+1 children.
    - All leaves are at the same depth.
- Why is it "The Bridge"?
  - It combines the principles of a balanced tree with a disk-oriented structure.
  - It's a self-balancing multi-way tree.
- Advantages:
  - Wide and Short: A single node holds many keys, drastically reducing the tree's height.
  - Efficient I/O: Loading one node (one disk block) provides a lot of information for the next step.

B-Tree Example & Mechanics

- Example: Inserting into a B-Tree of Order 5
  - Each node can have up to 4 keys and 5 children.
  - Insertion: Find the leaf. If the leaf is full, split it and promote the middle key to the parent.
  - Splitting propagates upwards, which is how the tree grows in height and stays balanced.
- Key Point:
  - In a B-Tree, every node contains both keys and the associated data (or pointers to data).

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

Identifying B-Tree's Limitations

- While B-Trees are excellent, we can optimize further.
- Limitation 1: Inefficient Range Queries
  - To perform a range query (e.g., "find all keys between 10 and 100"), you must perform a new tree traversal for each potential key or do an in-order traversal that jumps between different levels of the tree. This is inefficient.
- Limitation 2: Lower Fan-Out
  - Since internal nodes store both keys and data pointers, they can hold fewer keys per node.
- This can make the tree slightly taller than a theoretically optimal structure.

The Pinnacle: B+Tree

- Core Concept:
  - A B+Tree is a refinement of the B-Tree with one critical distinction:
    - 1. Data is only stored in the leaf nodes.
    - 2. Internal nodes act solely as a navigation index, storing only keys.
    - 3. Leaf nodes are linked together in a sorted, singly-linked list.

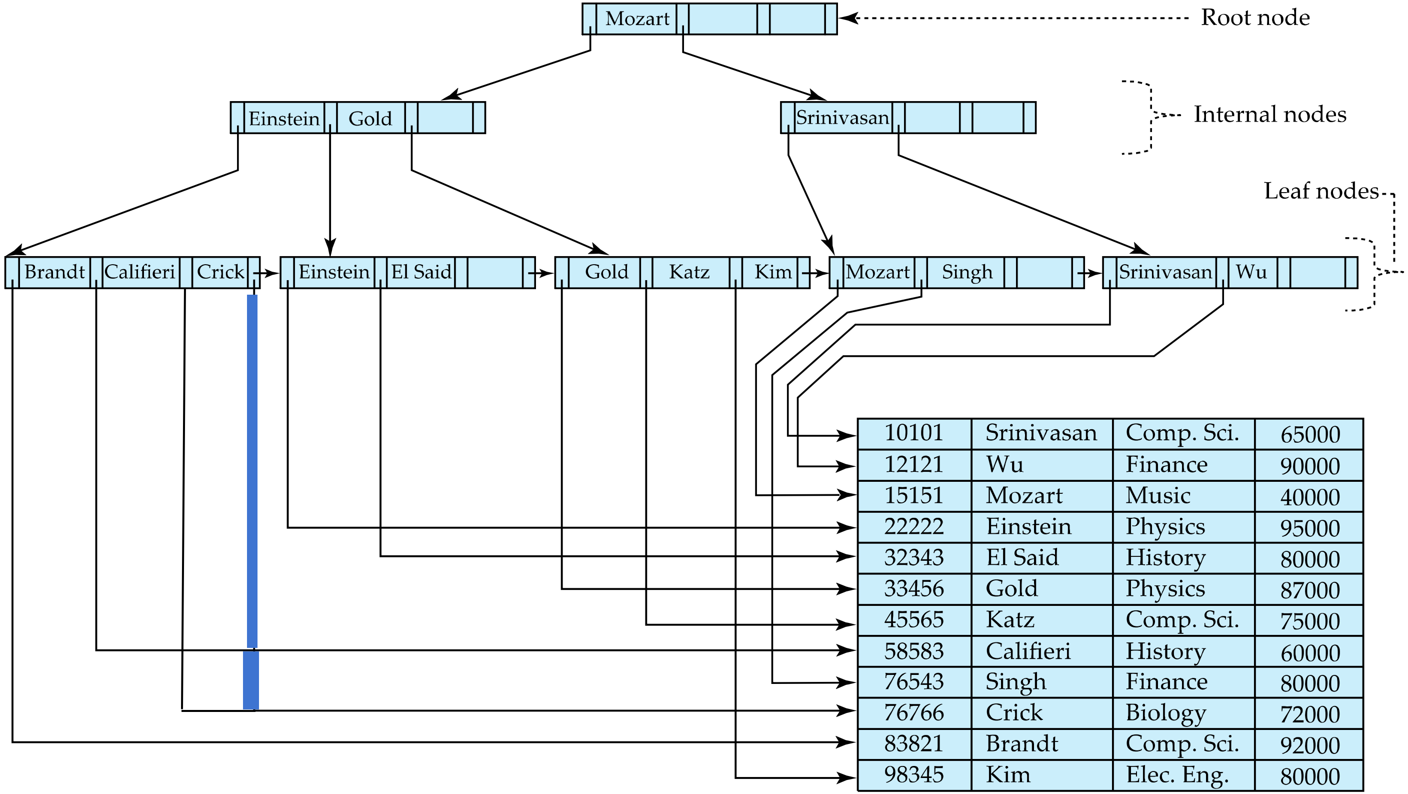
Why B+Tree is the Winner for Databases

- 1. Higher Fan-Out
  - Since internal nodes only store keys, more keys fit in a single disk block.
  - Result: An even shorter, fatter tree than a B-Tree, leading to fewer I/Os.
- 2. Blazing-Fast Range & Full Scan Queries
  - To do a range query, find the starting key in a leaf, then simply follow the leaf node pointers. No need to traverse back up the tree.
  - A full table scan is incredibly efficient—just traverse the linear list of leaves.
- 3. Stable Performance
  - Every search always goes from the root to a leaf. The I/O cost is predictable and stable.

B^+-Tree Index Files

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - Extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B+-Tree



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B^+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and n–1 values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.

B^+-Tree Node Structure

- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

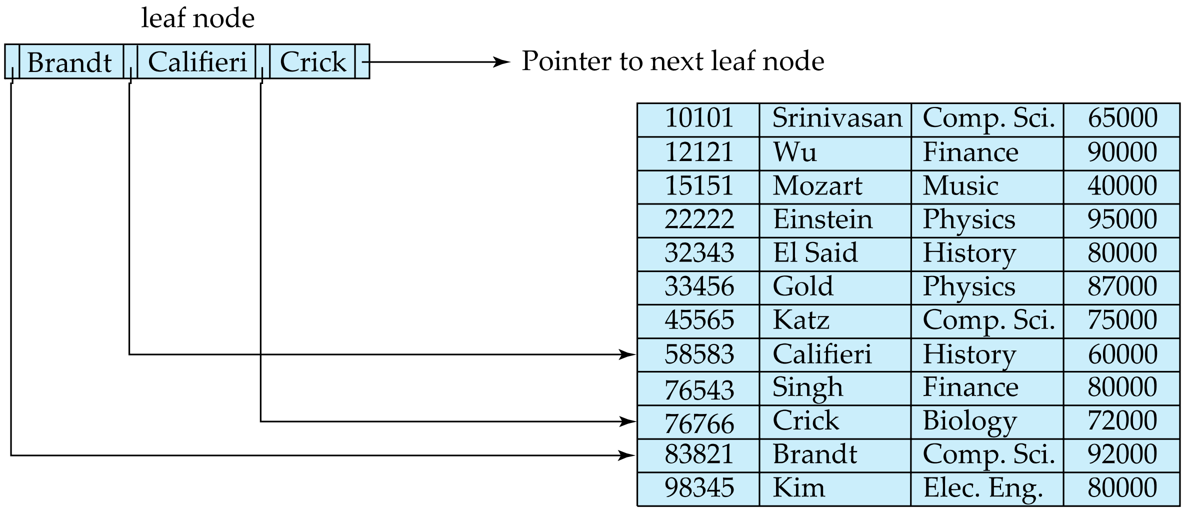
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B^+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

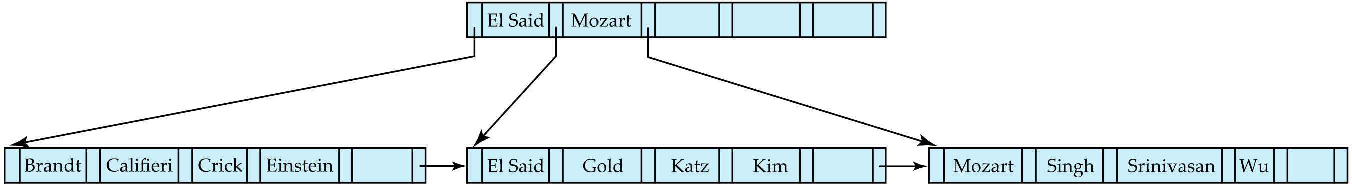
Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
  - General structure

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Example of B⁺-tree

- B⁺-tree for *instructor* file (n = 6)



- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).

- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2 \rceil$ and n with $n = 6$).

- Root must have at least 2 children.

Observations about B^+-trees

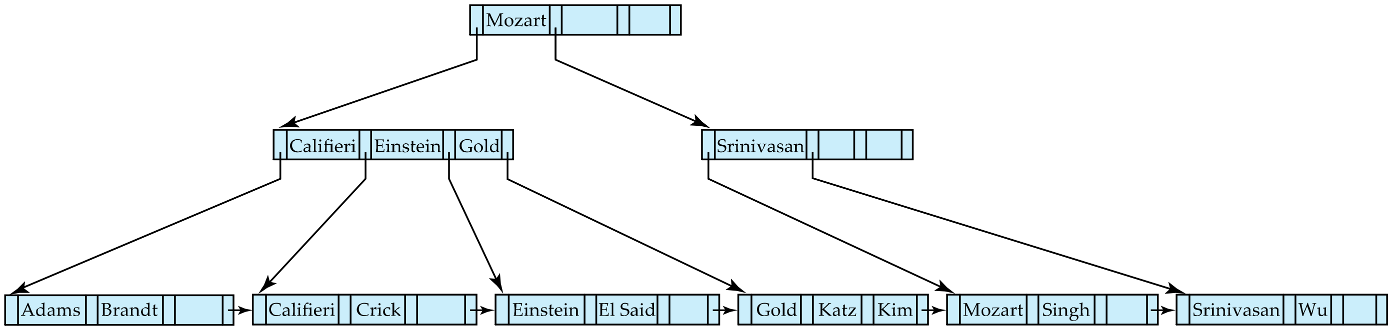
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2*\lceil n/2 \rceil$ values
  - Next level has at least $2*\lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
- If there are *K* search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
- thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Queries on B+-Trees

Xi'an Jiaotong-Liverpool University 西交利物浦大学

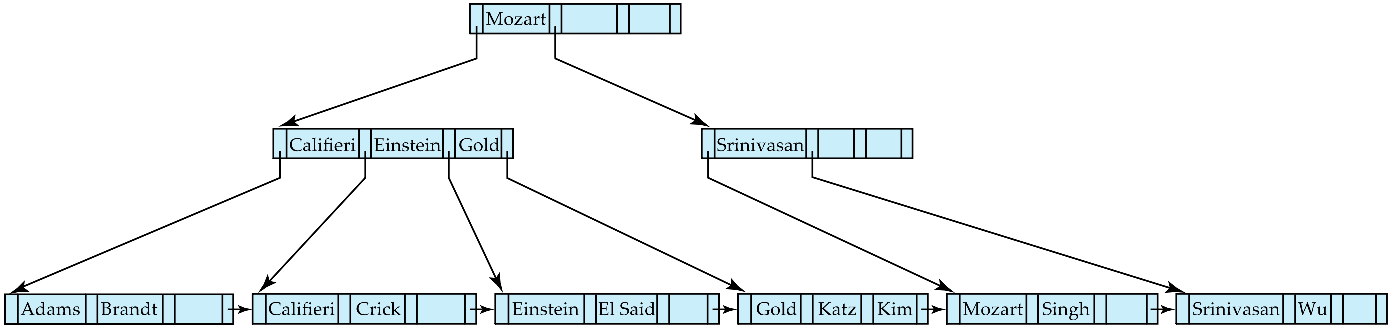
function *find*(*v*)

1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \leq K_i$.
   2. **if** there is no such number *i then*
   3. *Set C = last non-null pointer in C*
   4. **else if** (v = C.K_i) Set C = P_{i+1}
   5. **else set** *C* = C.P_i
3. **if** for some *i*, $K_i = V$ **then** return C.P_i
4. **else** return null /* no record with search-key value *v* exists. */



Queries on B+-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange*(*lb, ub*) which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function



Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Non-Unique Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i , A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation

Updates on B^+-Trees: Insertion

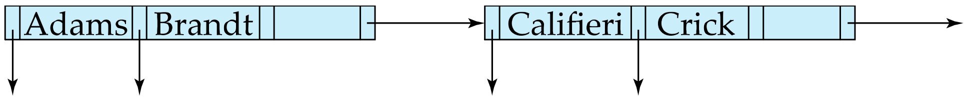
Assume record already added to the file. Let

- *pr* be pointer to the record, and let
- v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
   1. If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
   2. Otherwise, split the node (along with the new (*v, pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.

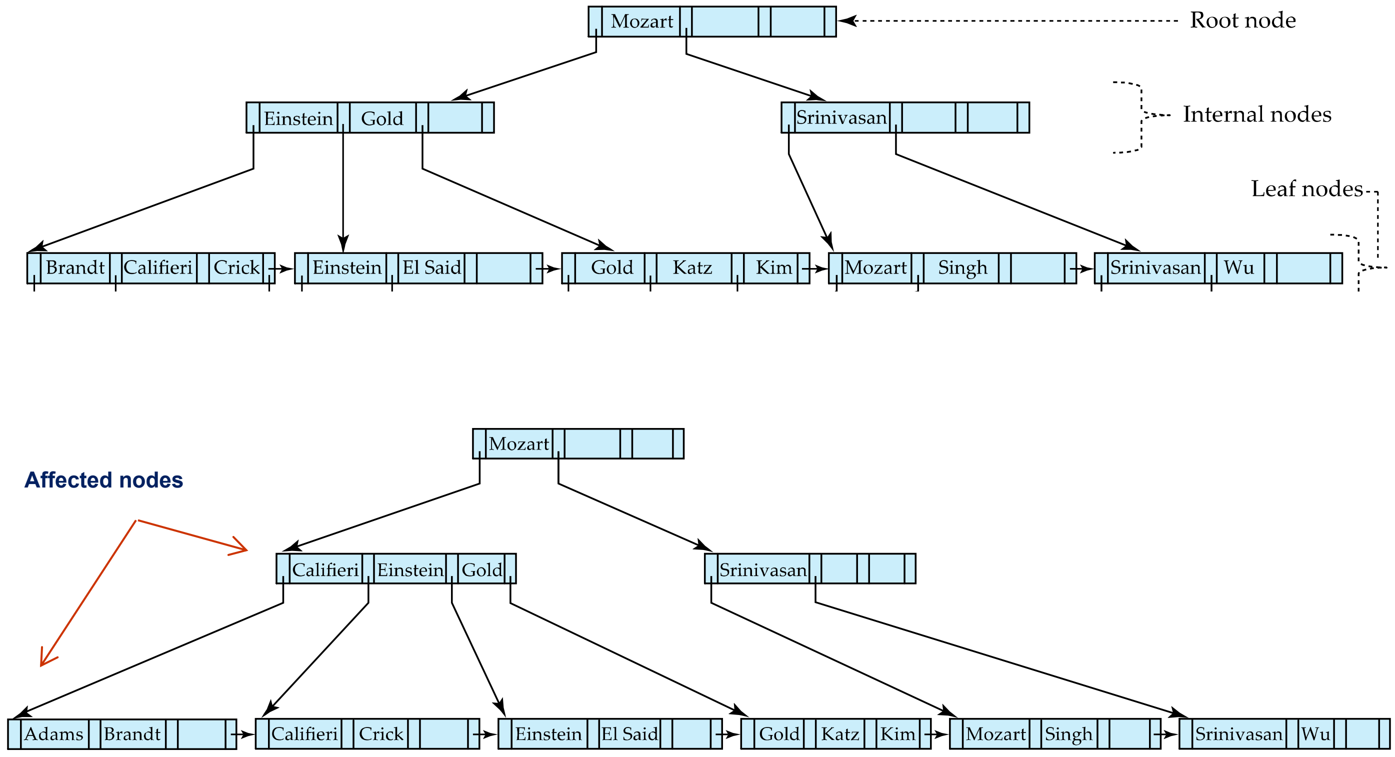
Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.



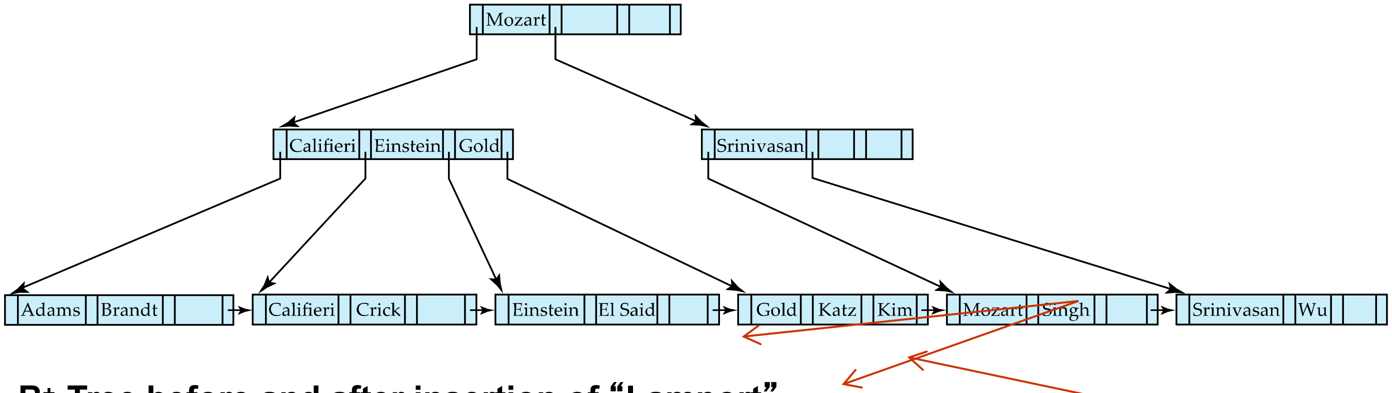
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

B+-Tree Insertion



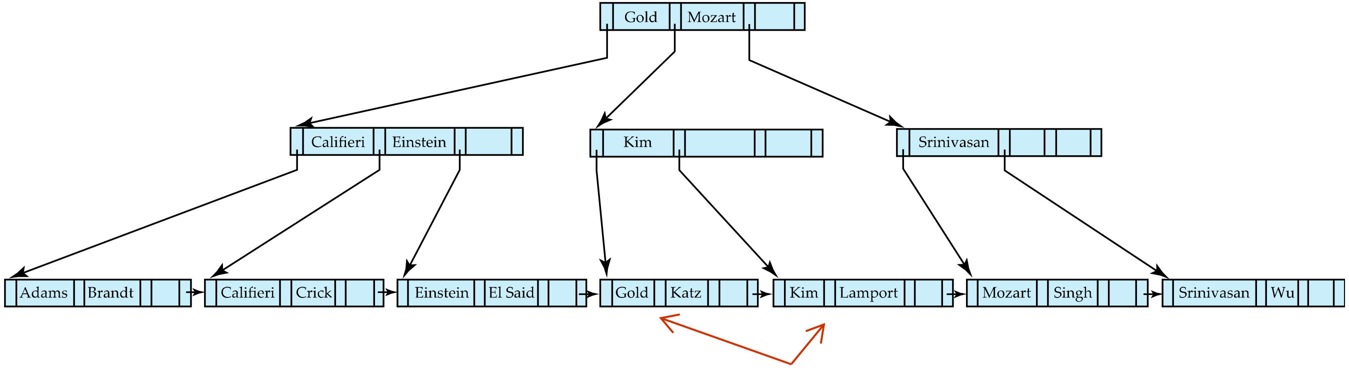
B+-Tree before and after insertion of "Adams"

B+-Tree Insertion



B+-Tree before and after insertion of "Lamport"

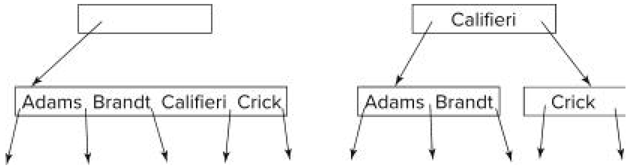
Affected nodes



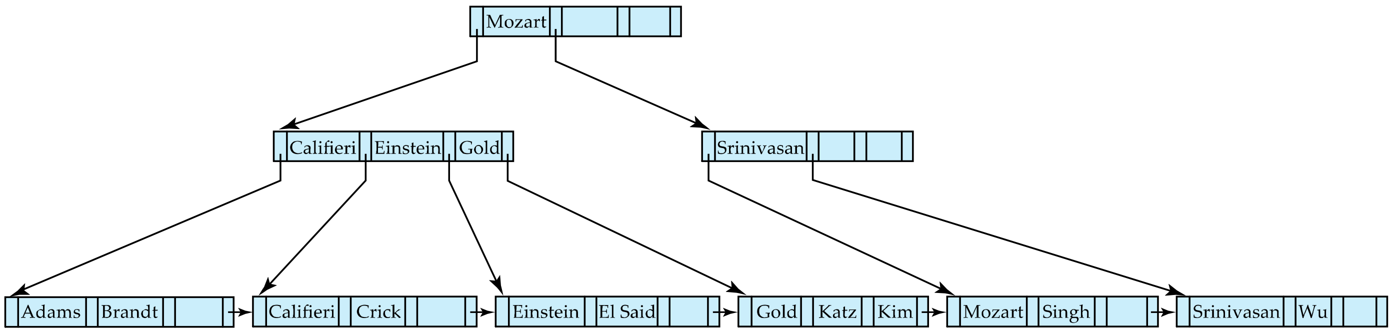
Affected nodes

Insertion in B+-Trees (Cont.)

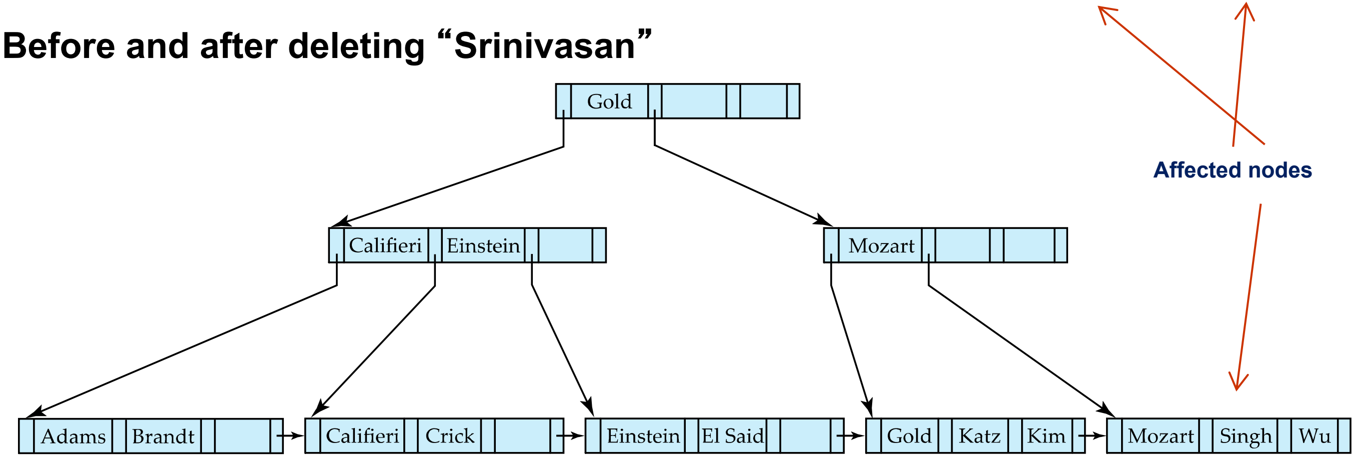
- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert ($K_{\lceil n/2 \rceil}$, N') into parent N
- Example



Examples of B+-Tree Deletion

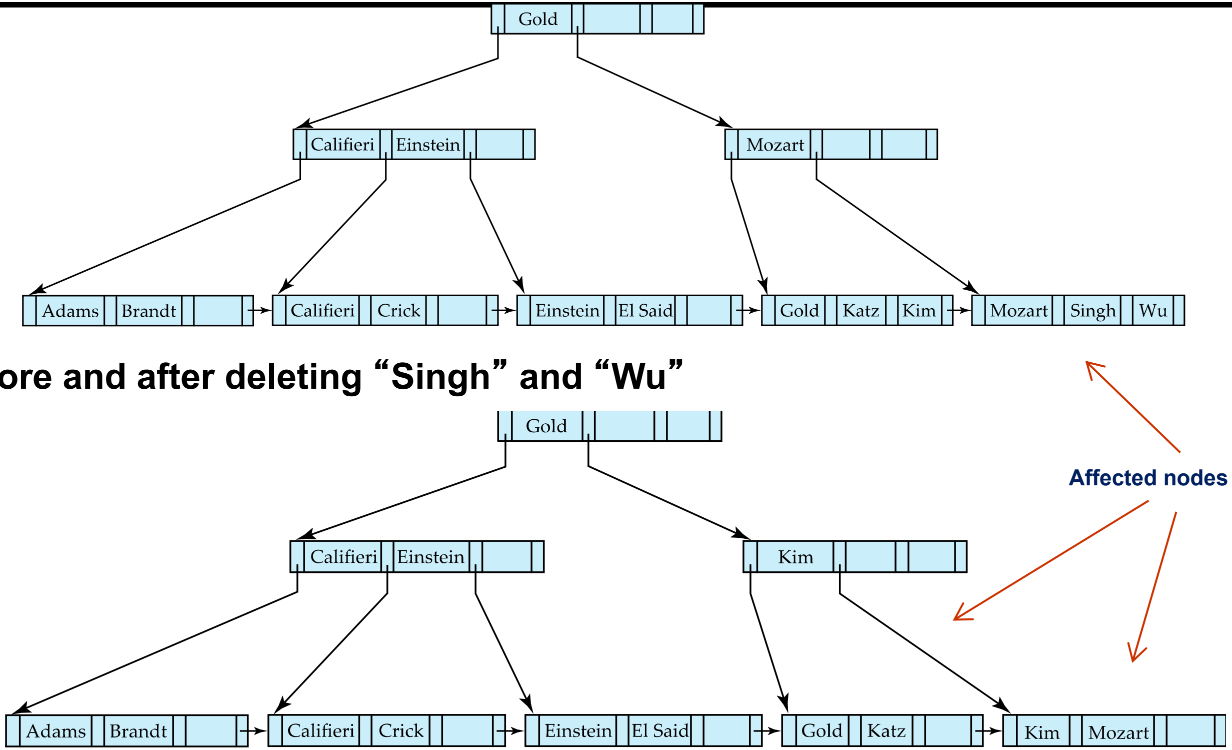


Before and after deleting "Srinivasan"



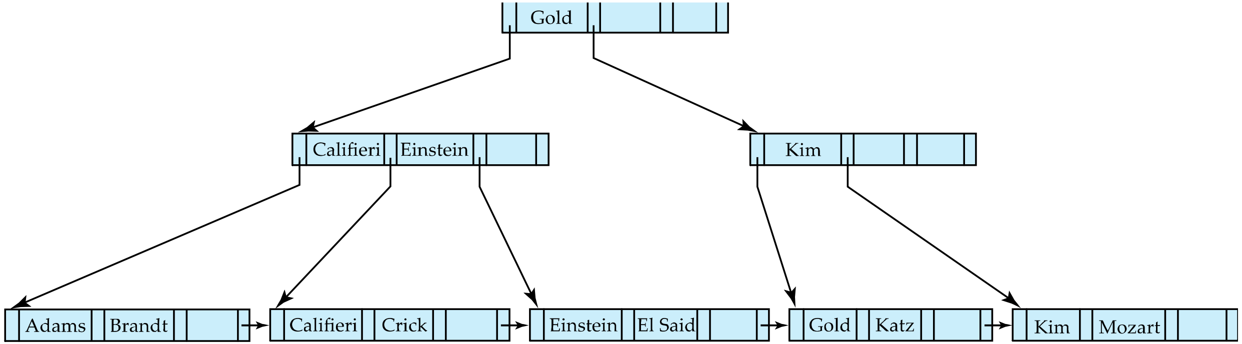
- Deleting "Srinivasan" causes **merging** of under-full leaves

Examples of B+-Tree Deletion (Cont.)

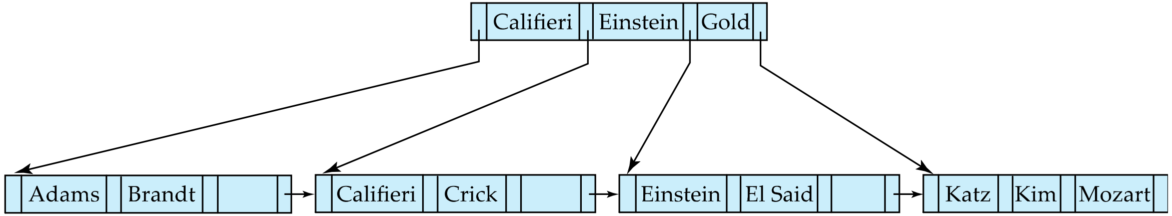


- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

Example of B+-tree Deletion (Cont.)



Before and after deletion of "Gold"



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

Updates on B+-Trees: Deletion

Assume record already deleted from file. Let *V* be the search key value of the record, and *Pr* be the pointer to the record.

- Remove (*Pr, V*) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

demo

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