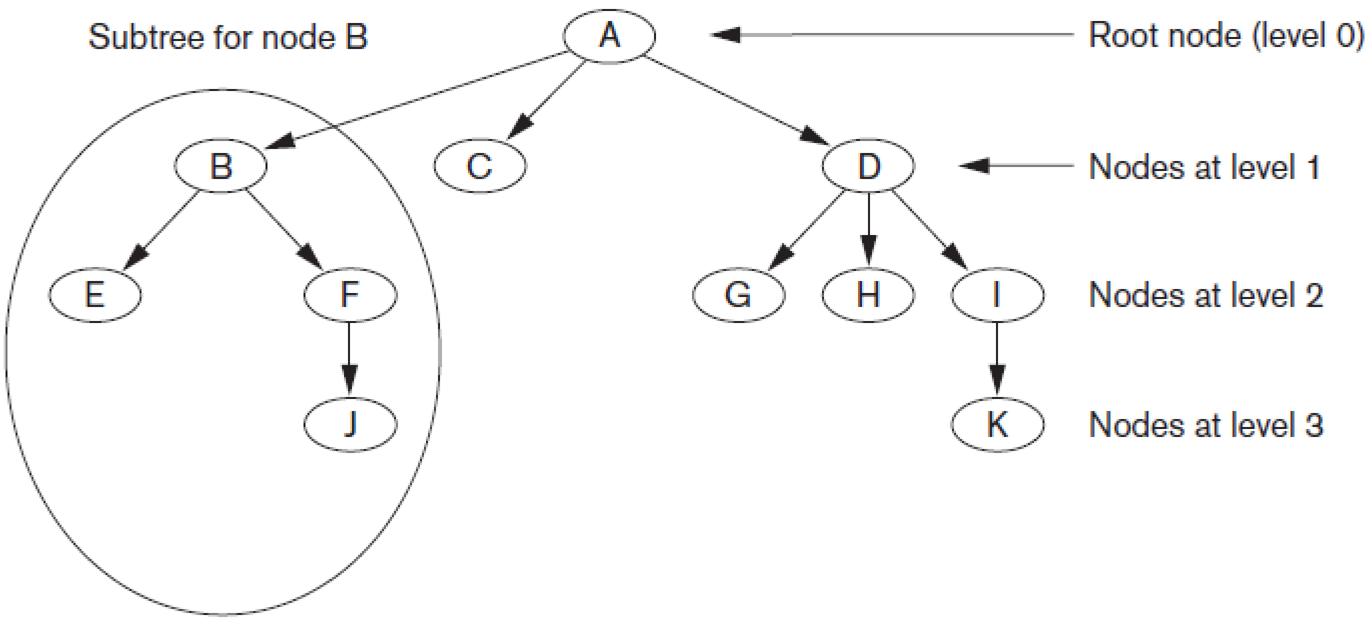
Chapter 12: B-Trees

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

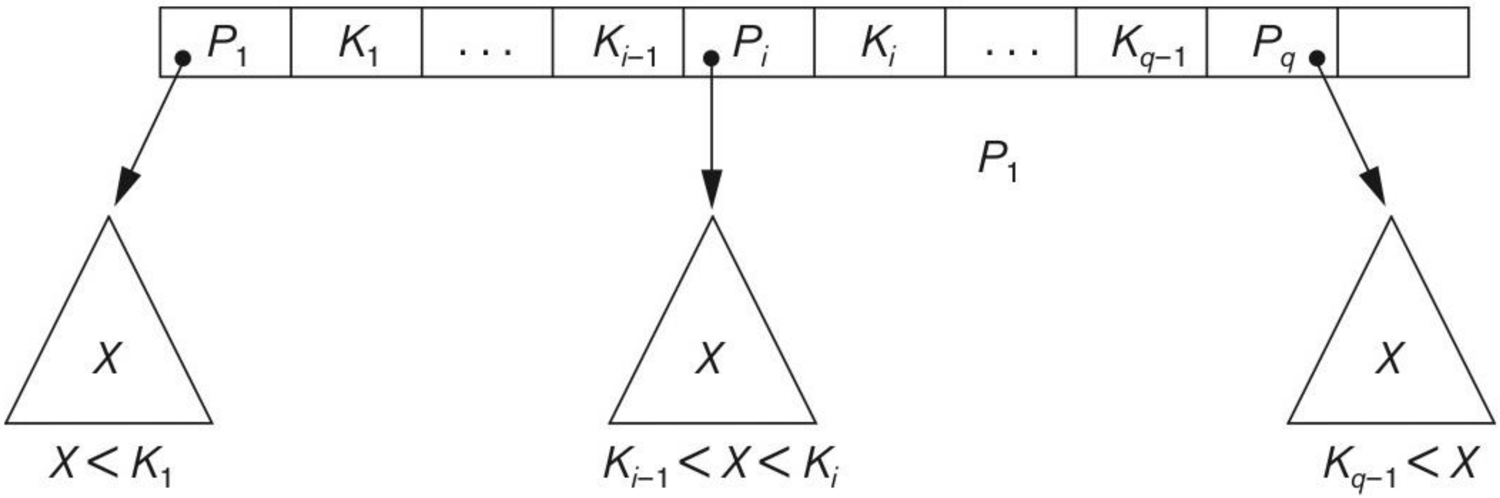
Tree Data Structure



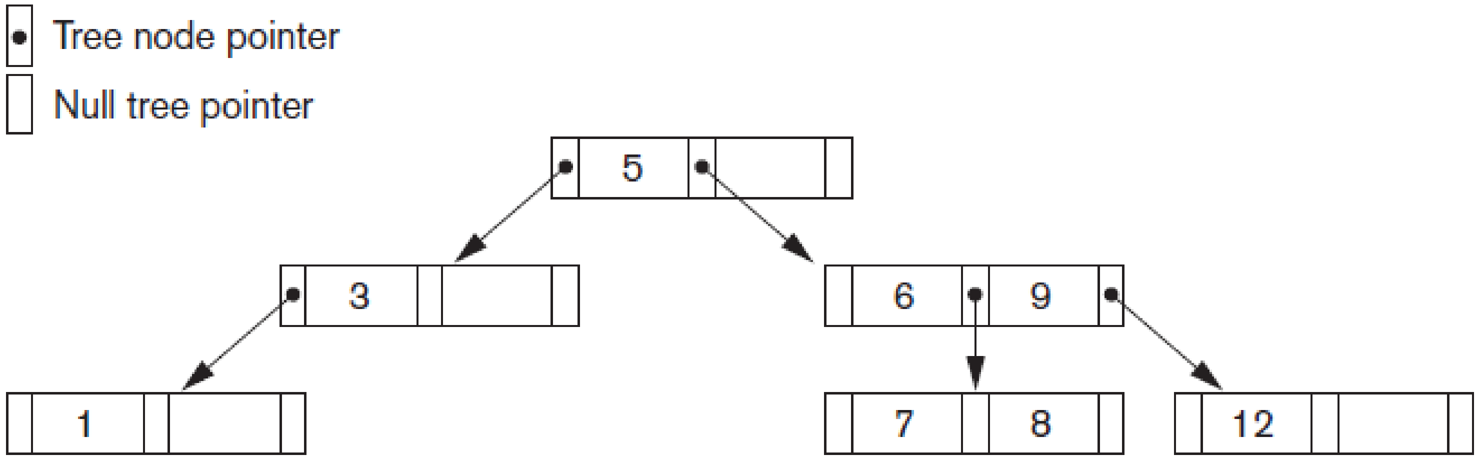
(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

Search Trees

- Search tree used to guide search for a record



Search Trees



B-Trees

- Provide multi-level access structure
- Tree is always balanced
- Space wasted by deletion never becomes excessive
  - **Each node is at least half-full**
- A B-tree can store values in nonleaf and leaf nodes
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children; i.e., n signifies the maximum number of children, and n is called the **order** of the tree

B^+-Tree Index Files

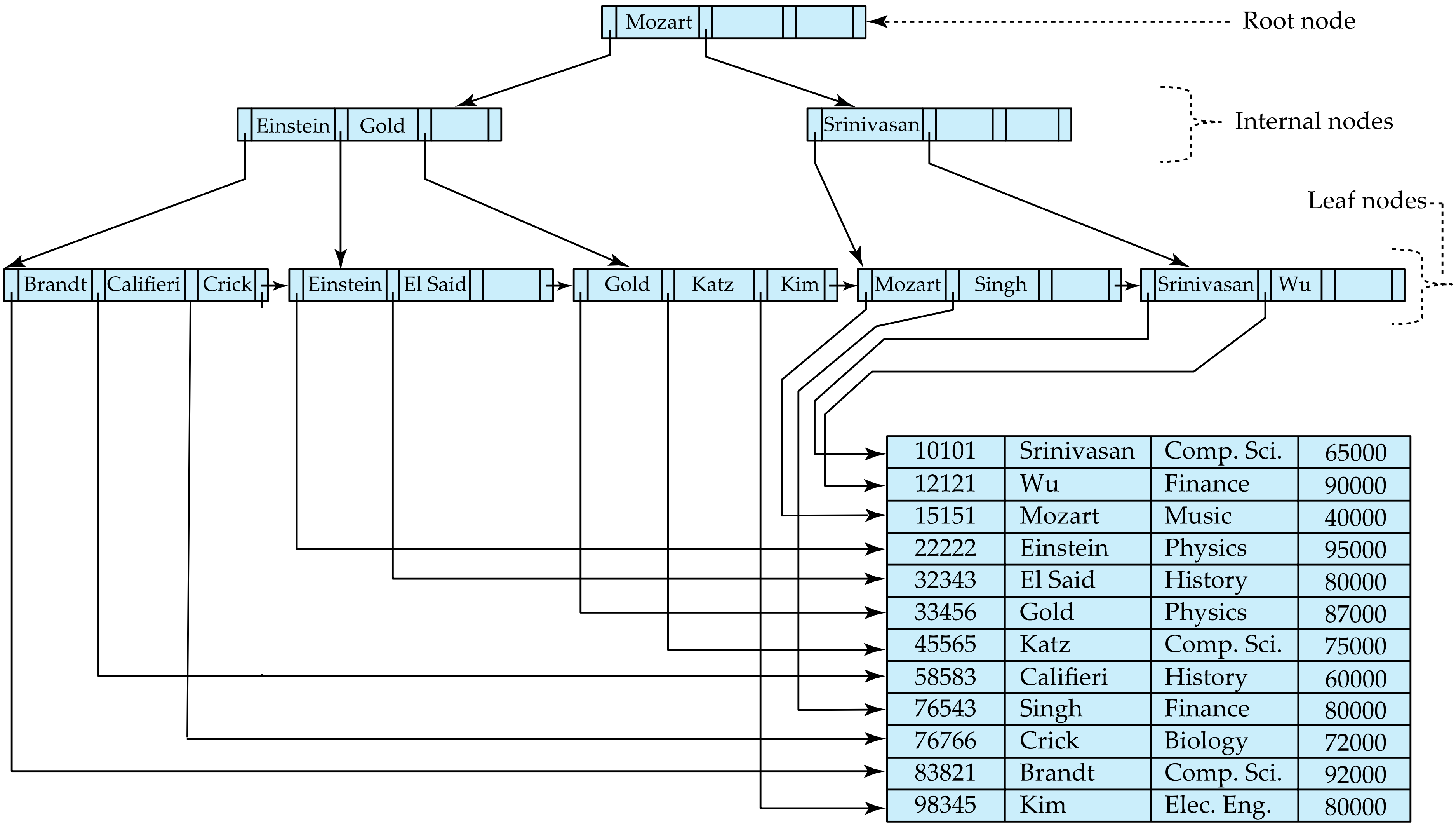
A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Non-leaf nodes are called **internal nodes**
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- Note that the number of values is one less than the number of children; i.e., between $\lceil n/2 \rceil -1$ and $n -1$ values, or equivalently, between $\lfloor (n -1)/2 \rfloor$ and $n -1$ values
- The above equivalence can be seen by noting that $\boldsymbol{\lceil n/2 \rceil -1 = \lfloor (n -1)/2 \rfloor}$; this is so because
  - for n even, $\lceil n/2 \rceil -1 = (n/2) -1$, and $\lfloor (n -1)/2 \rfloor = \lfloor (n/2) -1/2 \rfloor = (n/2) -1$; thus $\lceil n/2 \rceil -1 = \lfloor (n -1)/2 \rfloor = (n/2) -1$
  - for n odd, $\lceil n/2 \rceil -1 = (n + 1)/2 - 1 = (n -1)/2$, and $\lfloor (n -1)/2 \rfloor = (n -1)/2$; thus $\lceil n/2 \rceil -1 = \lfloor (n -1)/2 \rfloor = (n -1)/2$

B^+-Tree Index Files

- In terms of the number of values or slots, a node that is not a root or a leaf can be slightly less than half-full for n even, but would be at least half-full for n odd:
  - <u>For *n* even</u>, the maximum number of values is $(n - 1)$, and the minimum number of values is $\lceil n/2 \rceil - 1 = n/2 - 1$, which gives a minimum fullness of $(n/2 - 1)/(n - 1) = ½\,[(n - 2)/(n - 1)] < ½$, which will $\rightarrow ½$ as $n \rightarrow \infty$.
  - <u>For *n* odd</u>, the maximum number of values is $(n - 1)$, and the minimum number of values is $\lceil n/2 \rceil - 1 = (n - 1)/2$ as shown earlier, which gives a minimum fullness of $[(n - 1)/2]/(n - 1) = ½$
- To ensure that the number of slots of a leaf node is at least half-full, instead of using the lower limit as $\lceil n/2 \rceil - 1$, which for even n can be less than half-full, we modify it to $\lceil n/2 \rceil$
- If the root is not a leaf, it has at least 2 children

Example of B^+-Tree ($n = 4$)



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

©Silberschatz, Korth and Sudarshan

B^+-Tree Node Structure

- Typical internal node, similar to search trees

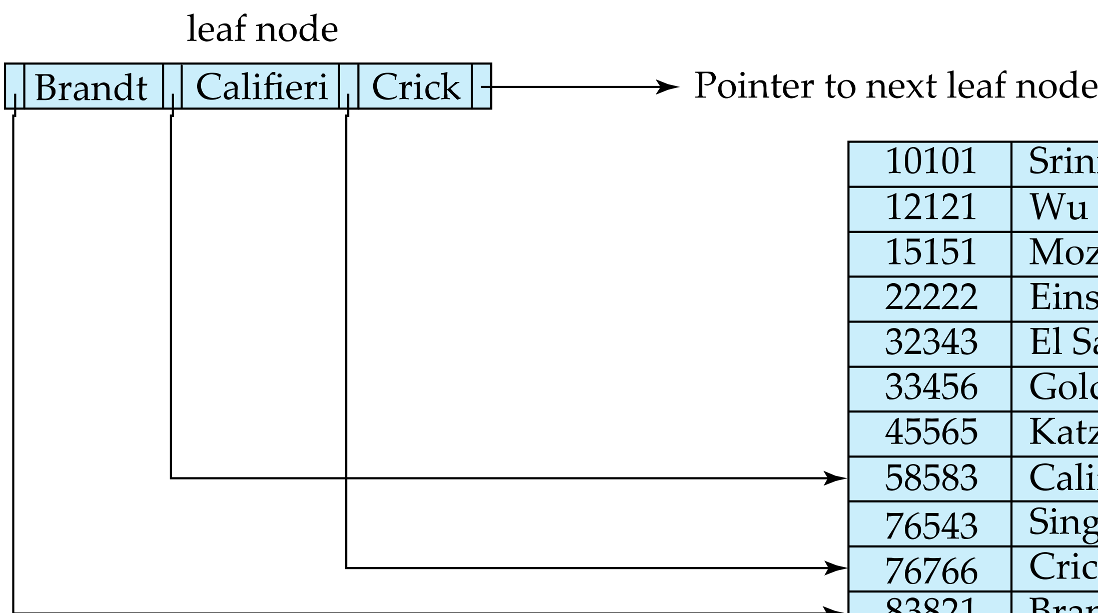
P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - 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

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

Leaf Nodes in B⁺-Trees (n = 4)

- For i = 1, 2, . . ., 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 (facilitates sequential processing)

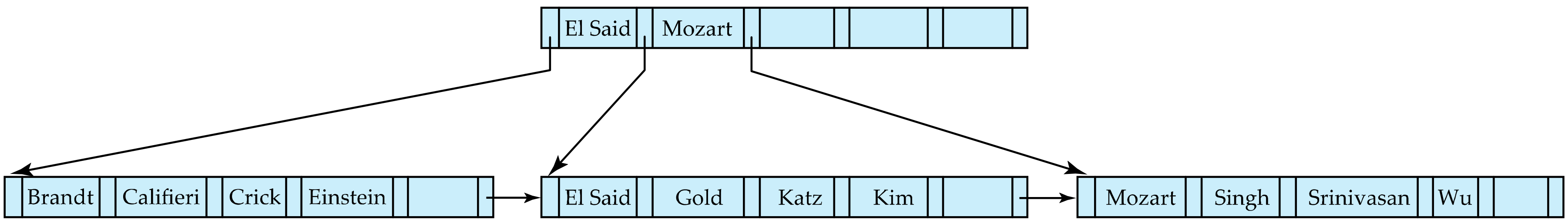


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

 ©Silberschatz, Korth and Sudarshan

Example of B$^+$-tree (n = 6)

- B$^+$-tree for *instructor* file (n = 6), i.e., 6 pointers and 5 slots



- Leaf nodes must have between 3 and 5 values ($\lceil n/2 \rceil$ 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 children) ⇒ between 2 and 5 values
  - In terms of the values, this is less than half-full (i.e., 2/5 = 40% full)
- If, however, n = 7, then the non-leaf nodes will have between 4 and 7 children ($\lceil (n/2 \rceil$ and n children) ⇒ between 3 and 6 values which is at least half-full
- Root must have at least 2 children

©Silberschatz, Korth and Sudarshan

Performance of B$^+$-trees

- For a B$^+$-tree, where each node contains between $m = \lceil n/2 \rceil$ and n children (assuming the root behaves like any other node):
  - Number of nodes at Level 1 = m^0 (root)
  - Minimum number of nodes at Level 2 = m^1
  - Minimum number of nodes at Level 3 = m^2
  - Minimum number of nodes at Level 4 = m^3
  - Minimum number of nodes at Level h = m^{h-1}
- Each leaf node will hold roughly at least m values, so that assuming the height of the tree is h, the minimum number of values held by the leaf nodes is approximately $m \times m^{h-1} = m^h$
- If there are K search-key values in the file, we have

$$m^h = K$$

giving $h = \lceil \log_{\lceil n/2 \rceil}(K) \rceil$

- By assuming minimum storage utilization of each node, we have the maximum number of nodes for the tree and hence maximum height for the tree
- Thus, if there are K search-key values in the file, the tree height should be no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ (i.e. we are assuming all nodes are minimally full; if they are not minimally full, then the height should be less)

©Silberschatz, Korth and Sudarshan

Performance of B$^+$-Trees

- 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
  - Assuming 40 bytes per index entry, n is typically around 100
- With 1 million search key values and n = 100
  - at most $log_{50}(1{,}000{,}000) = 3.53 \approx 4$ 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 ($log_2(1{,}000{,}000) = 19.93 \approx 20$) are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Average Storage Utilization of B-Tree

- Let K be the total number of items in the tree,

  n be the maximum capacity of a node (i.e. a node can hold between $n/2$ and n items)

  $\boldsymbol{N}$ be the <u>random</u> number of nodes in the tree

  $\boldsymbol{\rho}$ be the <u>random</u> storage utilization of the tree

  f be the minimum fullness factor

  - $f = ½$ for the standard B-Tree,
  - $f = ⅔$ for the B*-Tree
- The total storage capacity of the (random) tree is $\boldsymbol{N}n$
- The storage utilization is the total number of items divided by the total storage capacity of all the nodes, i.e.

$$\boldsymbol{\rho} = K/(\boldsymbol{N}n)$$

- Now a minimum number of nodes would result if all nodes are full, which is K/n
- Likewise, a maximum number of nodes would result if all nodes are half-full, which is $K/(½n) = 2K/n$

©Silberschatz, Korth and Sudarshan

Average Storage Utilization of B-Tree

- For random insertion in which every configuration in the above node range is equally likely, the distribution of ***N*** may be approximated by the continuous uniform distribution over the interval [*K/n, 2K/n*] of length *K/n,* with height *n/K.*
- That is we have

$$\boldsymbol{N} \sim \mathcal{U}(K/n,\ 2K/n)$$

where $\mathcal{U}$ signifies the uniform distribution

- Thus, we have, approximately,

$$E(\boldsymbol{\rho}) = E(K/[\boldsymbol{N}n]) = (K/n)\ E(1/\boldsymbol{N})$$

$$= \left(\frac{K}{n}\right) \times \left(\frac{n}{K}\right) \int_{\frac{K}{n}}^{\frac{2K}{n}} \left(\frac{1}{t}\right) dt = \ln\left(\frac{2K}{n}\right) - \ln\left(\frac{K}{n}\right) = \ln 2 = 69.3\%$$

Average Storage Utilization for the General Case

- For the general case with arbitrary minimum fullness f, the distribution of $\boldsymbol{N}$ may be approximated by the uniform distribution over the interval $[K/n, K/(nf)]$ of length $Kf'/(nf)$, with height $nf/(Kf')$, where $f' = 1 - f$.
- That is we have

$$\boldsymbol{N} \sim \mathcal{U}(K/n,\ K/nf)$$

- Thus, we get, approximately,

$$E(\boldsymbol{\rho}) = E(K/[\boldsymbol{N}n]) = (K/n)\ E(1/\boldsymbol{N})$$

$$= \left(\frac{K}{n}\right) \times \left(\frac{nf}{Kf'}\right) \int_{\frac{K}{n}}^{\frac{K}{nf}} \left(\frac{1}{t}\right) dt = \frac{f}{f'} \ln \frac{1}{f}$$

- For the B*-Tree, $f = 2/3$, and substituting this value into the above, we get

$$E(\boldsymbol{\rho}) = 2 \times \ln(3/2) \approx 81\%$$

Random Storage Utilization for the General Case

The average is often limited. The cumulative distribution function of **ρ** gives complete information of the random situation and can be shown to be:

$$G(x) = \begin{cases} 1 & x > 1 \\ \dfrac{1}{f'}\left(1 - \dfrac{f}{x}\right) & f \le x \le 1 \\ 0 & x < f \end{cases}$$

i.e. Prob [**ρ** ≤ *x*] = *G*(*x*)*.* The corresponding probability density function is

$$g(x) = G'(x) = \begin{cases} \dfrac{f}{f' x^2} & f \le x \le 1 \;, \\ 0 & \text{elsewhere.} \end{cases}$$

 ©Silberschatz, Korth and Sudarshan

Variance of the Storage Utilization

From the above, the variance can be readily calculated, which can be shown to be:

$$\sigma_f^2 = f - \left(\frac{f}{f'}\right)^2 \left[\ln\left(\frac{1}{f}\right)\right]^2$$

The standard deviation of storage utilization of the B-Tree is 0.14, and that of the B*-Tree is 0.094.

Updates on B$^+$-Trees: Insertion

Let

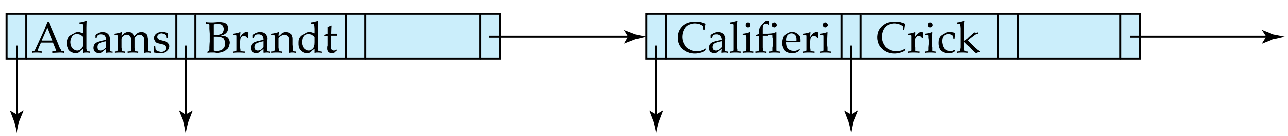
1. *Pr* be pointer to the record, and let
2. *V* be the search key value of the record

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)

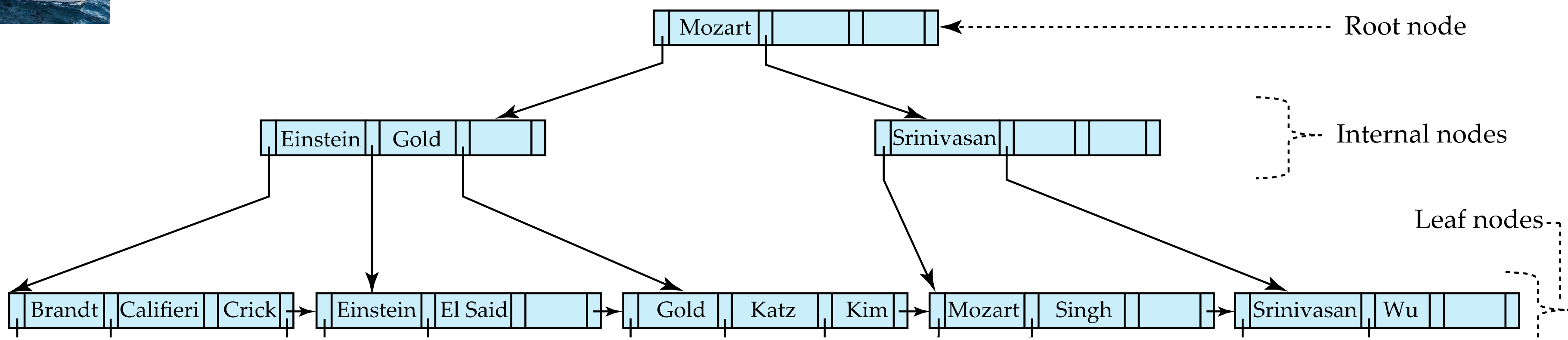
Updates on B+-Trees: Insertion

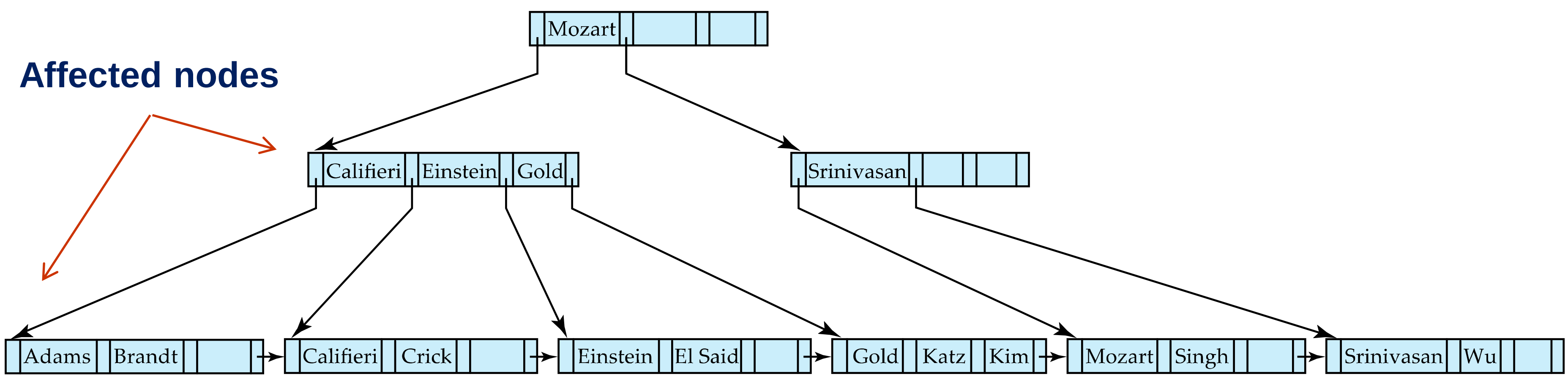
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in 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

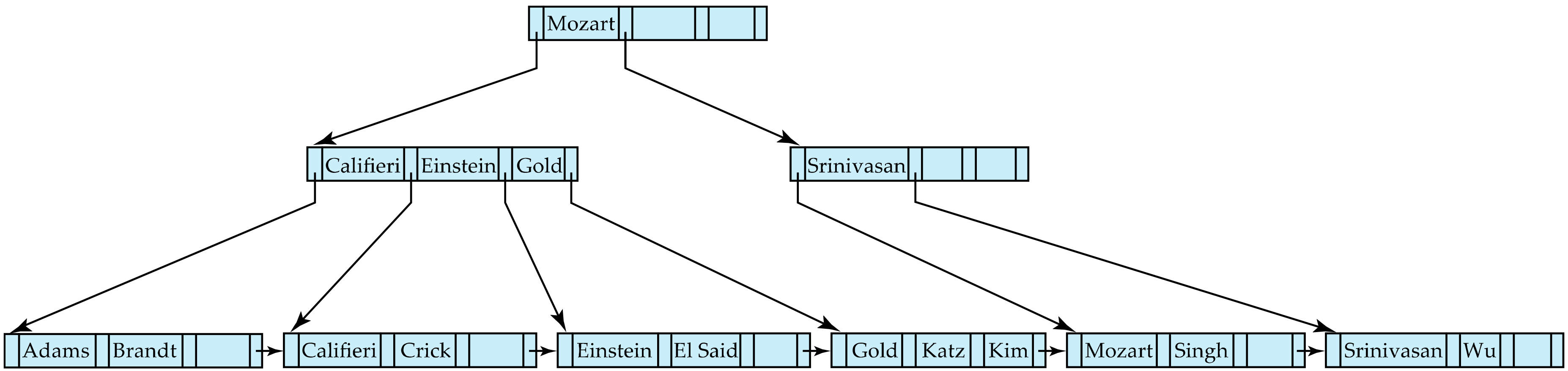
B+-Tree Insertion (*n* = 4)





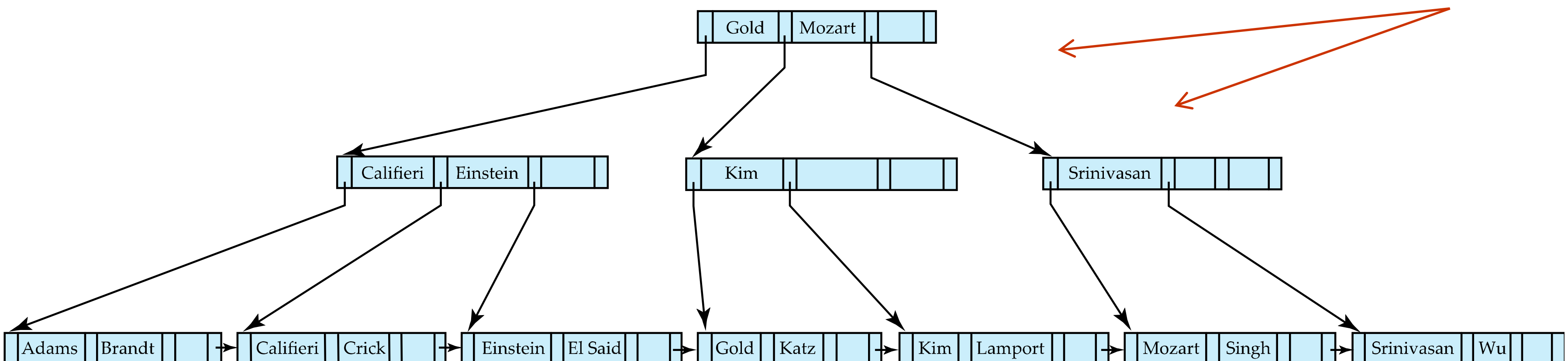
B+-Tree before and after insertion of "Adams"

B+-Tree Insertion (n = 4)



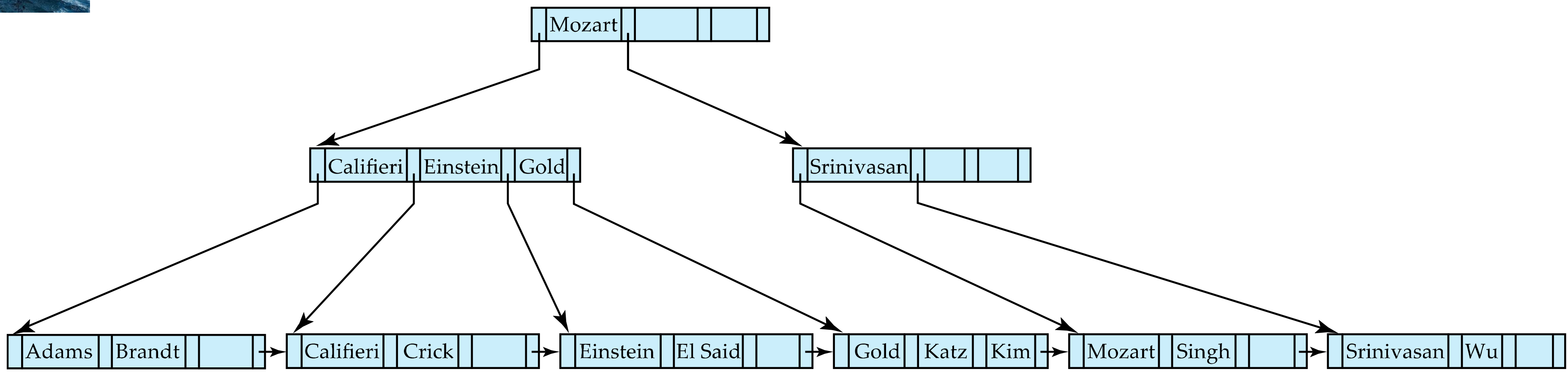
Affected nodes

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

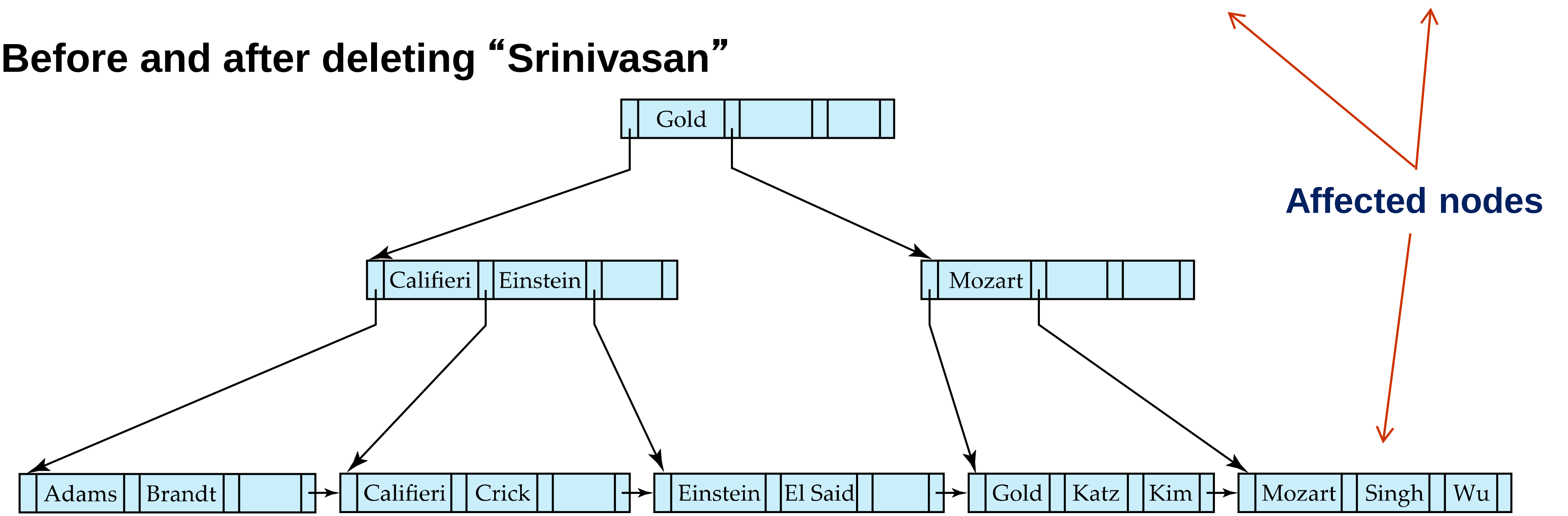


Affected nodes

Examples of B$^+$-Tree Deletion (n = 4)

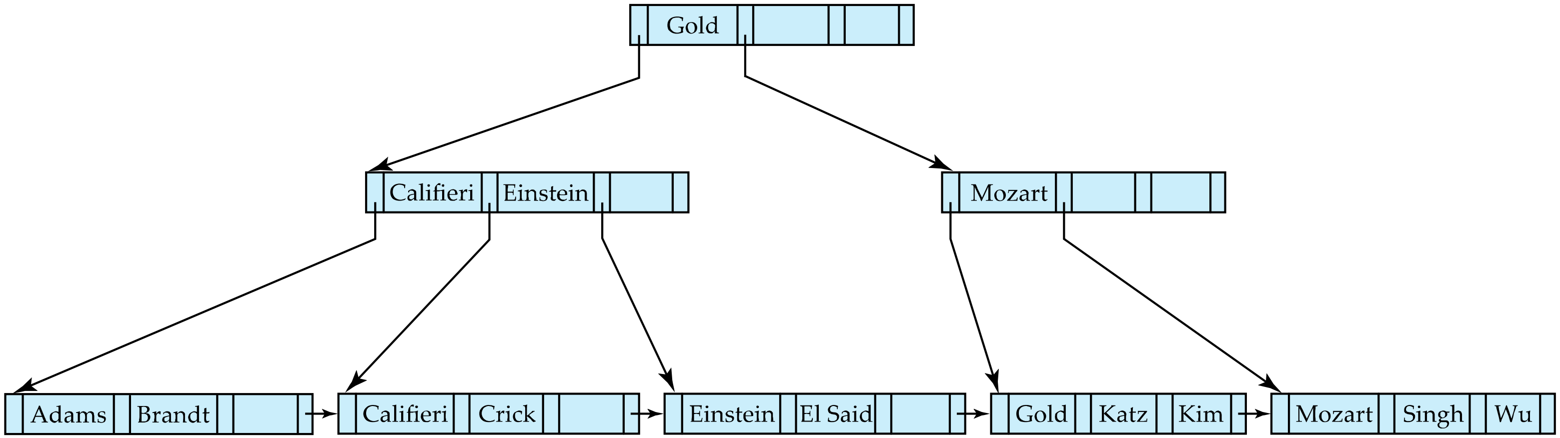


Before and after deleting "Srinivasan"

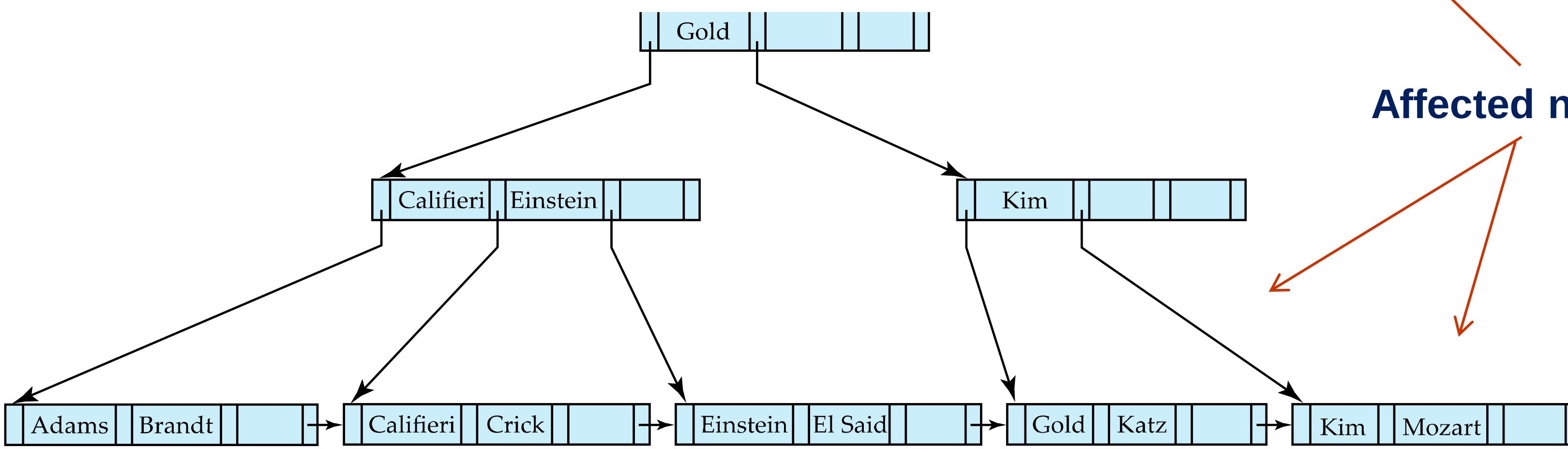


- Deleting "Srinivasan" causes **merging** of under-full leaves

Examples of B⁺-Tree Deletion ($n = 4$)



Before and after deleting "Singh" and "Wu"



Affected nodes

- 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

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* (i.e. maximum *n* children) , worst case complexity of insert/delete of an entry is, if all levels are affected, $O(\log_{\lceil n/2 \rceil}(K))$
- 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

Variations

- B^+-Tree File Organization
  - Stores actual records in leaf node, not just pointers
- B-Tree Index Files
  - Points to actual records in non-leaf nodes
- B^*-Tree
  - Can vary minimum fullness factor *f* to be different from $f = ½$,
  - for $f = ⅔$, the corresponding tree is called a B^*-Tree