Databases and Information Systems CS303

Indexing and Hashing
12-10-2023

B^+ Trees

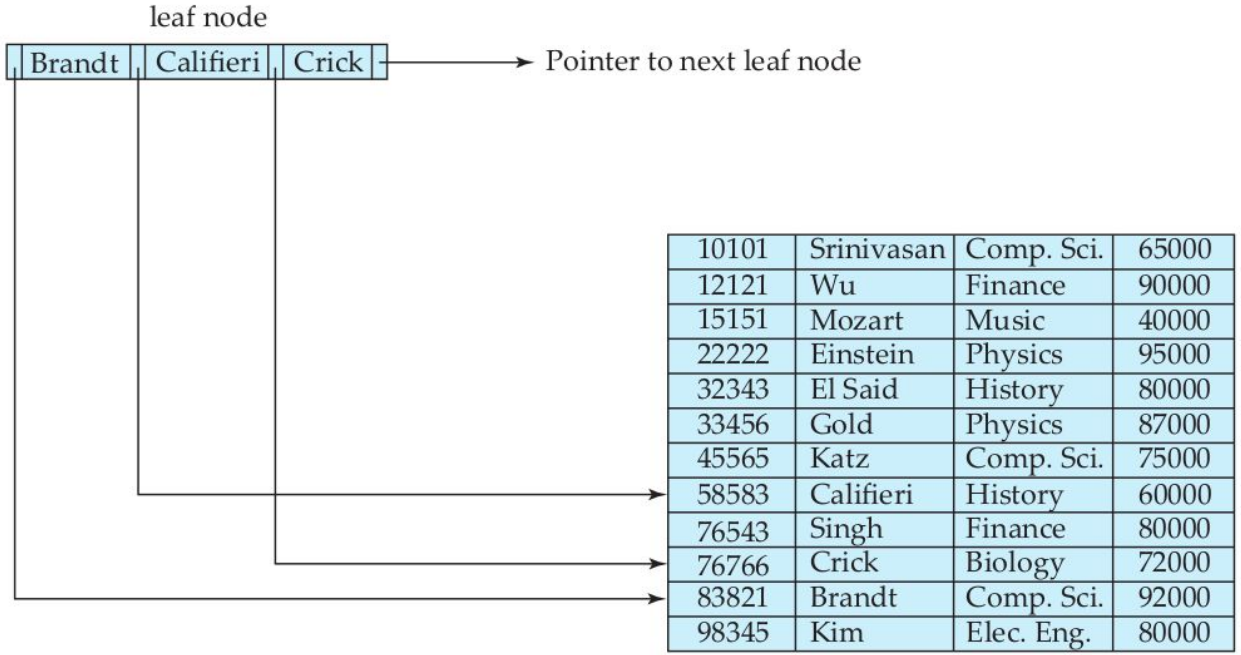
- Disadvantage of index sets : Performance degrades as file size grows
- B^+ - tree index structure is the most widely used index structures to maintain efficiency despite insertion and deletion of data.
- B^+ - tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.
    - Each non-leaf node in the tree has between n/2 and n children, where n is fixed for a particular tree.

B^+ Trees : Structure

- Let n be the maximum number of children a node can have
- Each node contains upto n-1 search-keys (K_1 ... K_n) and n Pointers (P_1 ... P_n)
- Search keys in a node are stored in an order
  - If $i < j$ then $K_i < K_j$

B^+ Trees : Structure

- Leaf Node :
  - For i = 1 to n-1 Each pointer P_i in leaf node points to a record in the database
  - Pointer P_n points to the next leaf node



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^+ Trees : Structure

- Leaf Node :
  - For $i = 1$ to $n-1$ Each pointer P_i in leaf node points to a record in the database
  - Pointer P_n points to the next leaf node

  - Leaf node can have at most n search-keys and at least $\lceil (n-1)/2 \rceil$ search-keys
    - With k search-keys, the node has k+1 pointers

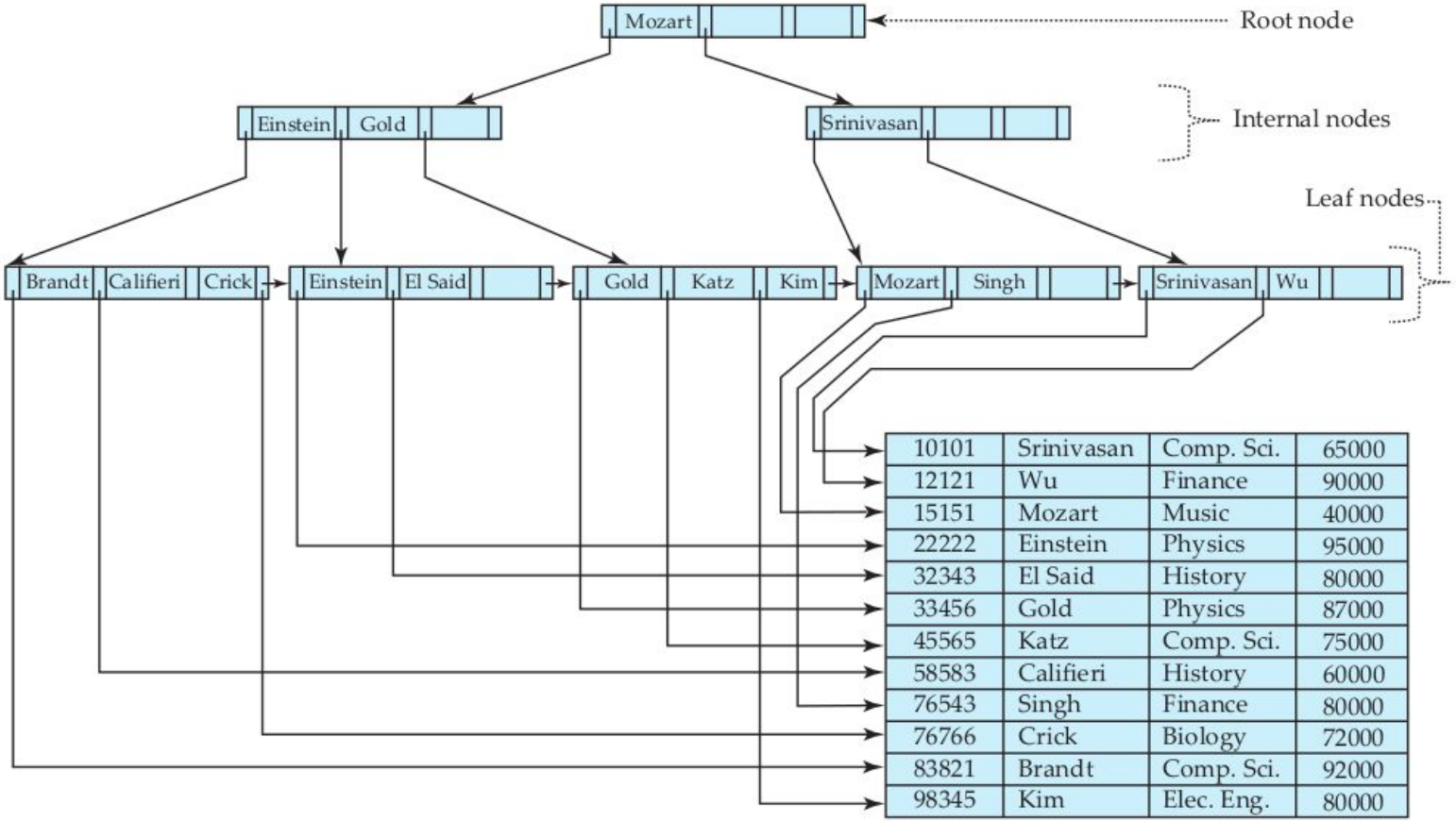
  - Values in leaf nodes do not overlap
    - Except if there are duplicates search-key values

  - B^+ - tree is a dense index (every search-key appears in some leaf node)

B^+ Trees : Structure

- Non-Leaf Node :
  - Forms multi-level index on the leaf nodes (sparse)
  - Can have at most n search-keys and at least $\lceil n/2 \rceil$ search-keys
    - With k search-keys, the node has k+1 pointers
  - Root node contains at most $\lceil n/2 \rceil$search-keys and at least 2 search-keys
  - Every pointer points to a node in the tree

B^+ Trees : Structure



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

Queries on B^+ Trees

```
function find(value V)
/* Returns leaf node C and index i such that C.P_i points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that V ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (V = C.K_i)
            then Set C = C.P_{i+1}
        else C = C.P_i /* V < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i
        then return (C, i)
        else return null ; /* No record with key value V exists*/
```

```
procedure printAll(value V)
/* prints all records with search key value V */
    Set done = false;
    Set (L, i) = find(V);
    if ((L, i) is null) return
    repeat
        repeat
            Print record pointed to by L.P_i
            Set i = i + 1
        until (i > number of keys in L or L.K_i > V)
        if (i > number of keys in L)
            then L = L.P_n
            else Set done = true;
    until (done or L is null)
```

Queries on B^+ Trees

- Can also be used for range queries
- Typically a node has same size of a block (usually 4KB)
- If search key is 12Bytes and Node pointers are 8 Bytes then n ~ 200
- Even with n = 100 and 1 million search keys, we need to access about 4 blocks
- Root node is always accessed (so stays in buffer)
  - So only 3 or fewer blocks are read from the disk

Difference between B^+ trees and Binary Trees

- Binary trees are stored in memory
- Node size differs by orders of magnitude
- B^+ trees have large branching with short paths
  - Binary trees have small branching with long paths
- Search in Binary tree takes $O(\log_2 n)$ where n is the number of nodes
- Search in B^+ trees take $O(\log_m n)$
  - where m is the max number of children and n is the number of nodes

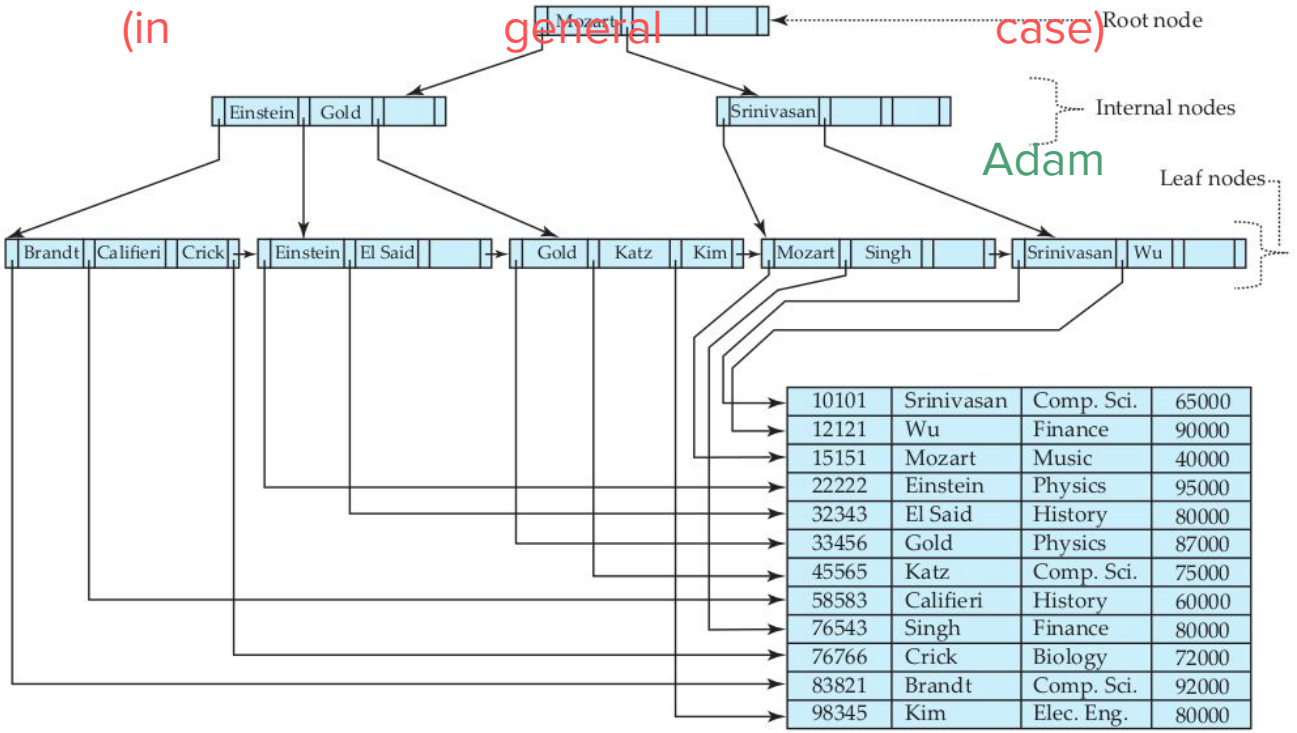
Updating B^+ trees (with n children)

- Insertion
  - First find the appropriate place in the leaf where the key should be inserted (using find procedure)
  - Suppose the leaf node has fewer than n-1 search keys, then insert the new key in the appropriate position (May have to shift few search-keys and pointers)

- Deletion
  - First find the appropriate place in the leaf where the key should be inserted (using find procedure)
  - If there are multiple search-keys with same value, go to appropriate record that should be deleted
  - Suppose the leaf has more than n/2 search keys, then delete the key and the associated record (may have to shift few search-keys and pointers)

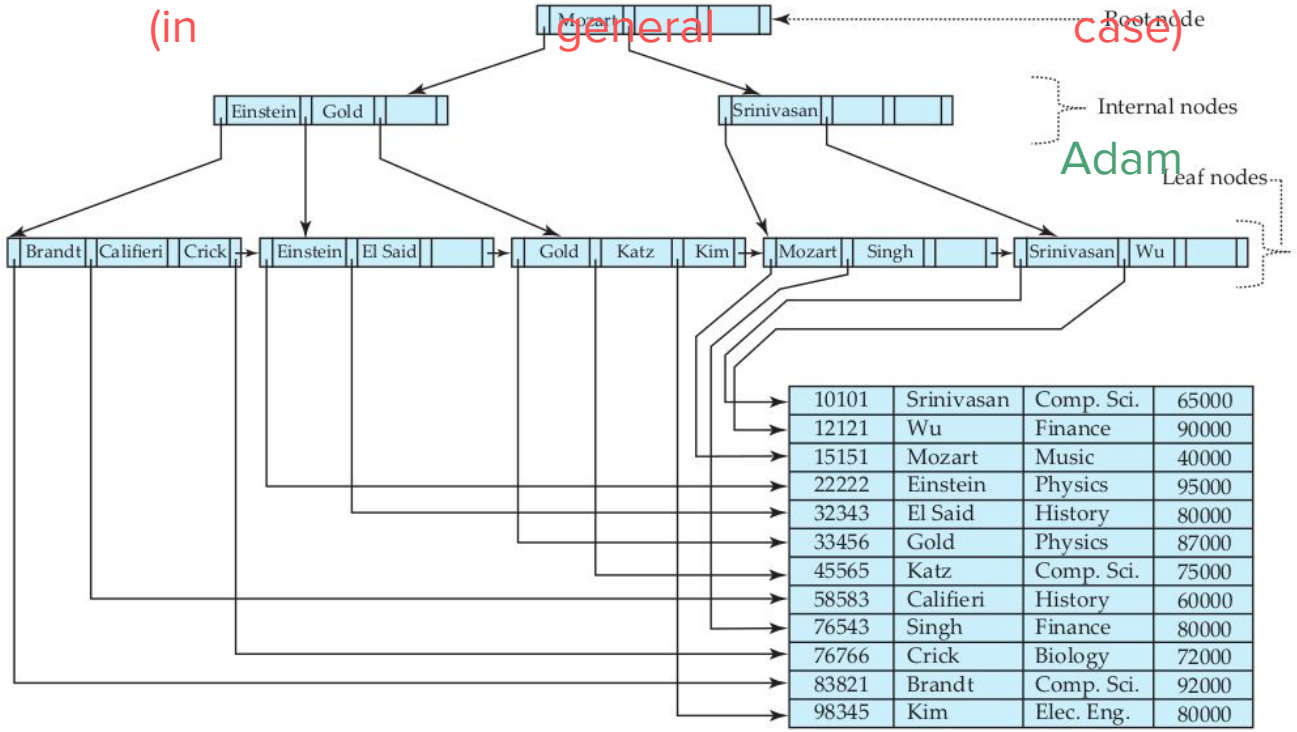
Updating B+ trees (with n children)

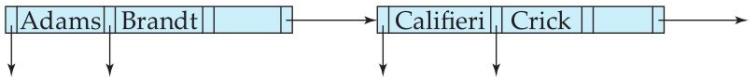
- Insertion (in general case)
- Insert Adam
- Split the node



Updating B+ trees (with n children)

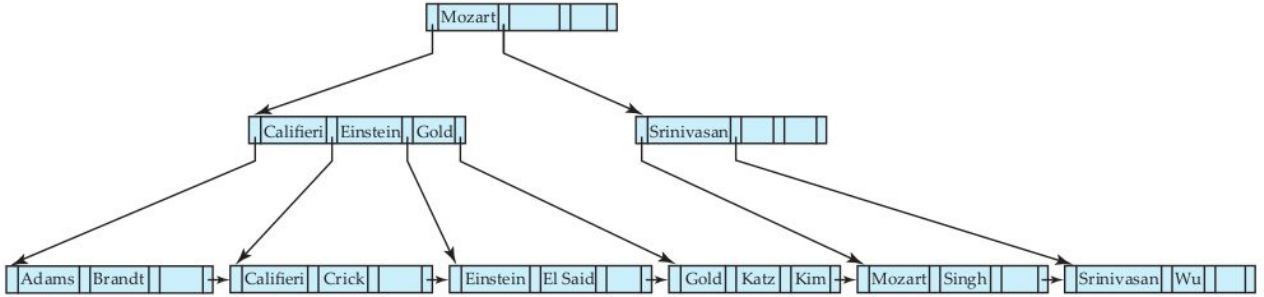
- Insertion (in general case)
- Insert Adam
- Split the node

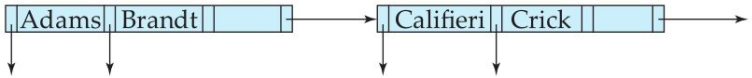




Updating B^+ trees (with n children)

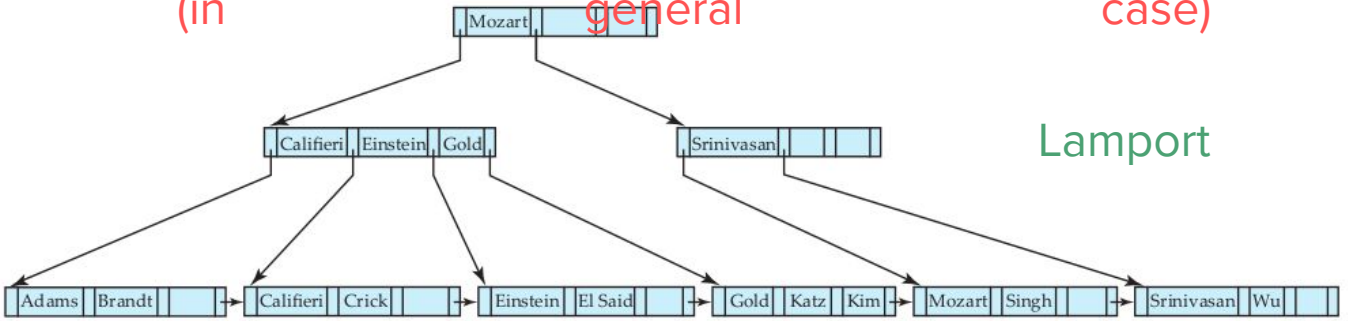
- Insertion (in general case)
- Insert
- Split the node

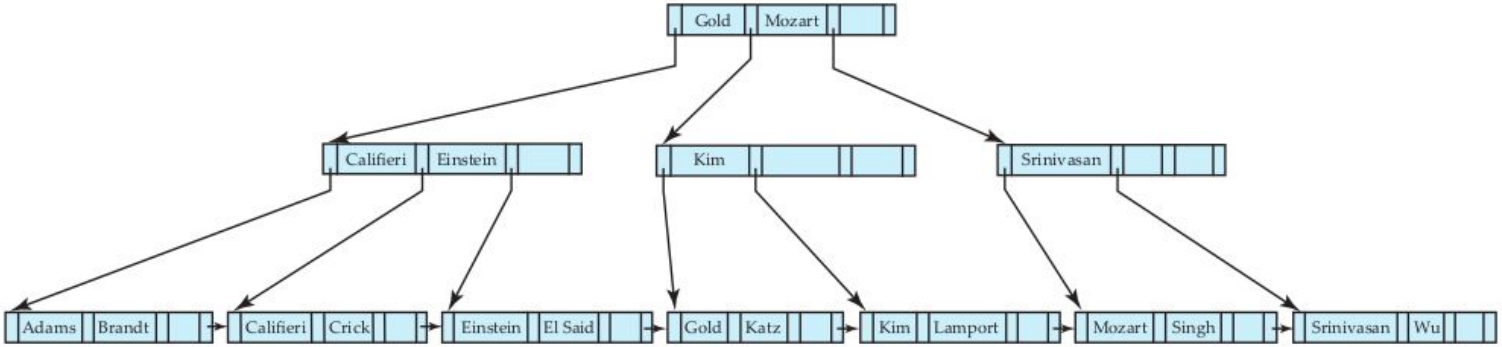




Updating B+ trees (with n children)

- Insertion (in general case)
- Insert Lamport





Insertion in B^+ trees (with n children)

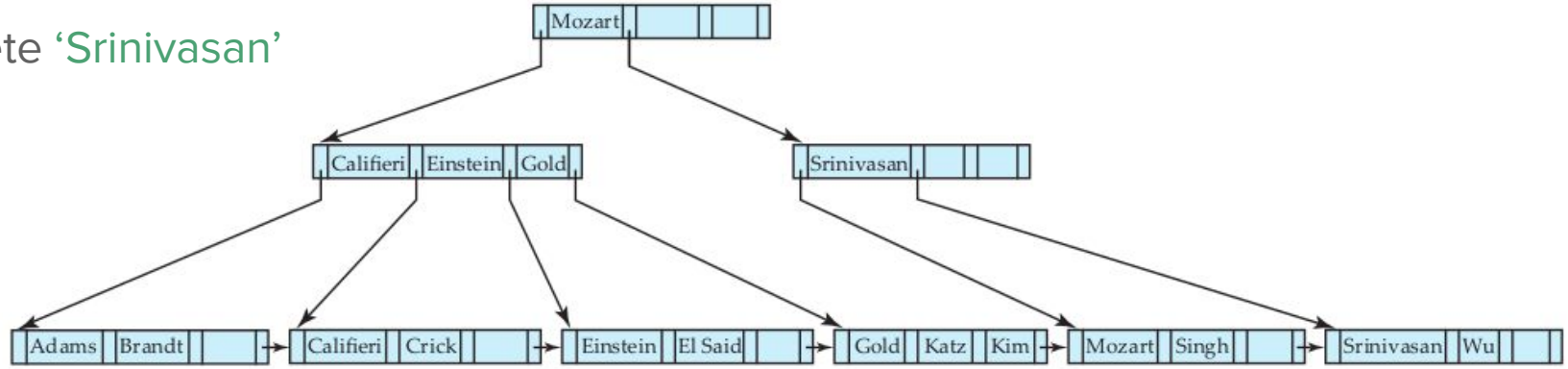
```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L'
            Copy L.P1 ... L.Kn−1 to a block of memory T that can
                 hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L'.Pn = L.Pn; Set L.Pn = L'
            Erase L.P1 through L.Kn−1 from L
            Copy T.P1 through T.K⌈n/2⌉ from T into L starting at L.P1
            Copy T.P⌈n/2⌉+1 through T.Kn from T into L' starting at L'.P1
            Let K' be the smallest key-value in L'
            insert_in_parent(L, K', L')
        end
```

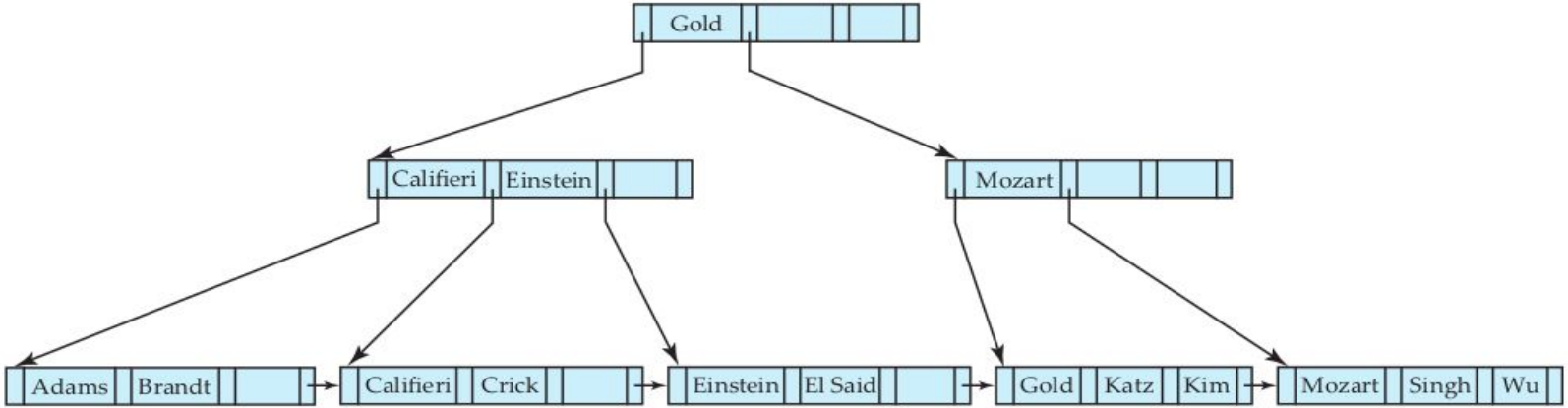
```
procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K1)
        then insert P, K into L just before L.P1
        else begin
            Let Ki be the highest value in L that is less than K
            Insert P, K into L just after T.Ki
        end

procedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'    /* N and N' are pointers
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P1 ... T.P⌈n/2⌉ into P
            Let K'' = T.K⌈n/2⌉
            Copy T.P⌈n/2⌉+1 ... T.Pn+1 into P'
            insert_in_parent(P, K'', P')
        end
```

Deletion in B^+ trees (with n children)

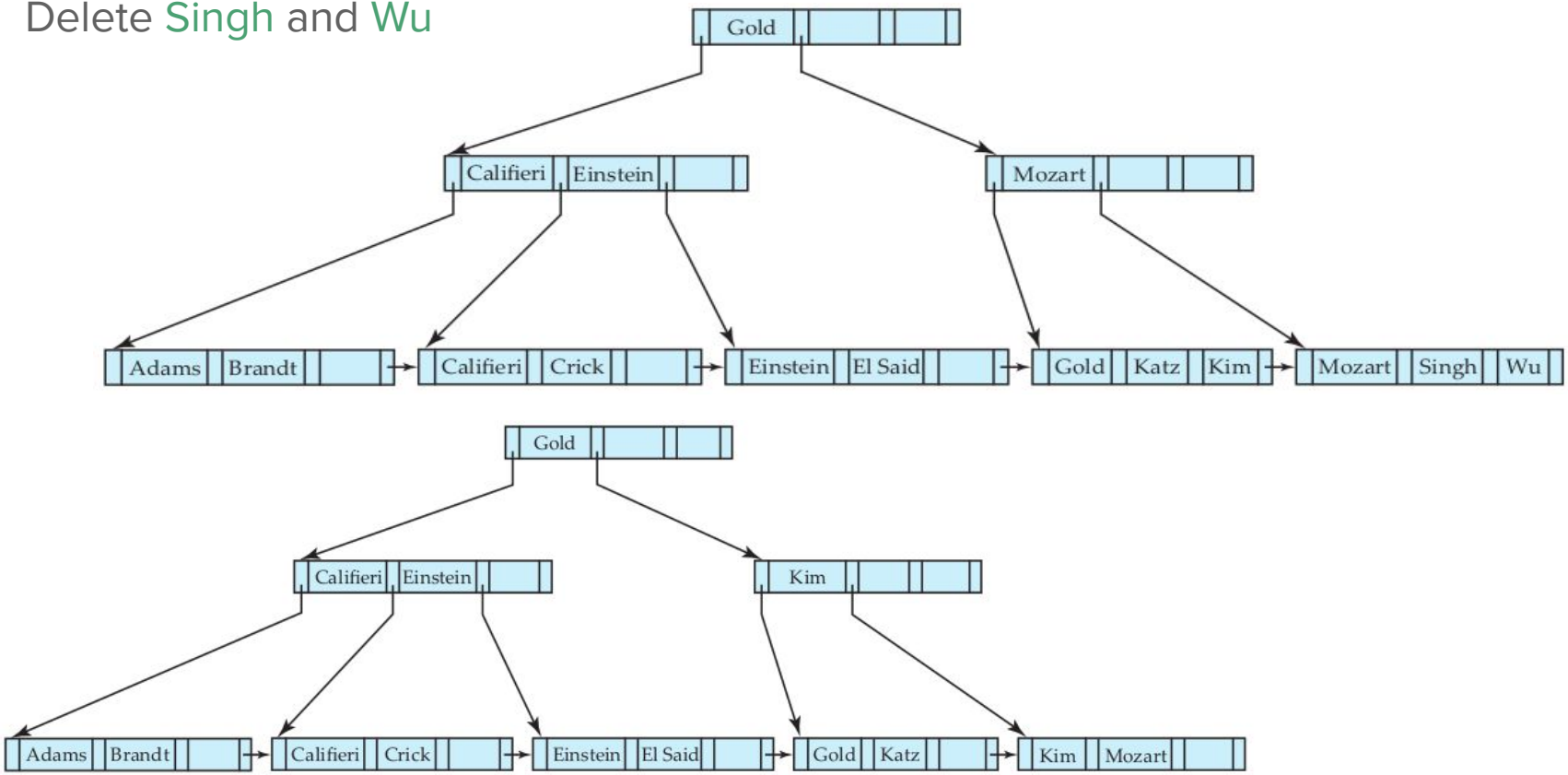
- Delete ‘Srinivasan’





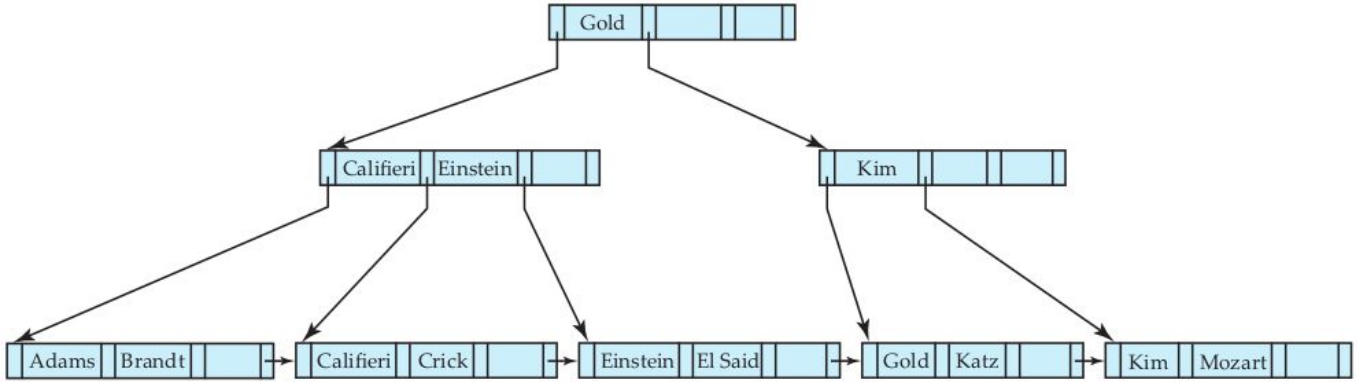
Deletion in B^+ trees (with n children)

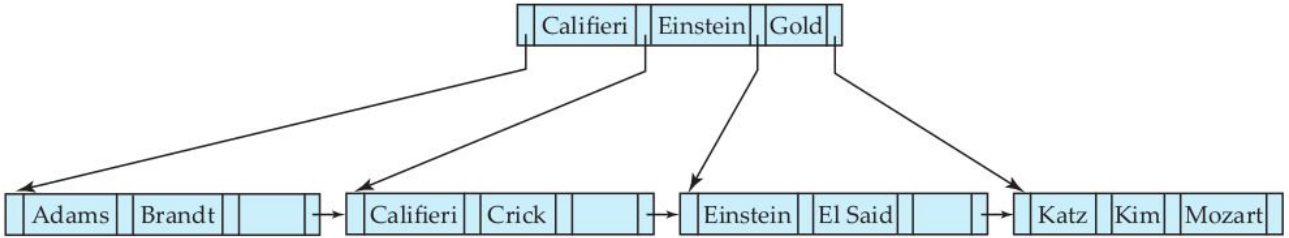
- Delete Singh and Wu



Deletion in B^+ trees (with n children)

- Delete Gold





Deletion in B^+ trees (with n children)

```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)
```

```
procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
                        pair in N'
                    remove (N'.P_m, N'.K_m) from N'
                    insert (N'.P_m, N'.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_m
                end
            end
            else ... symmetric to the then case ...
        end
    end
```

Non unique search-keys

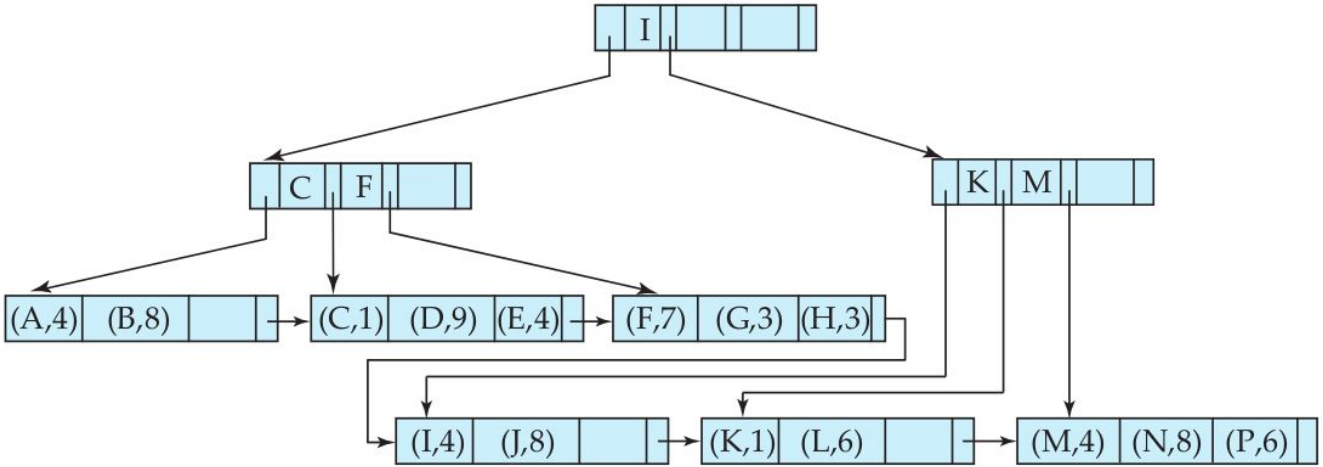
- Keys that are there in more than one tuple of a relation
- If one of them has to be deleted, finding the right node takes time
  - Solution : Create composite key with another attribute (whose value is unique among all tuples with repeating search-keys)
    - Search query can ignore the new additional attribute
- Alternatively, have unique keys in B^+ tree and leaf points have a bucket
  - Efficient space for the B^+ tree
  - But should deal with variable bucket size
  - More I/O if buckets are stored across blocks
  - Inefficient deletions

Complexity of B^+ trees Updates

- Insertion and Deletion : $\log_{\lceil n/2 \rceil}(N)$ where n is the maximum number of children
- In practise, performs fewer I/O than worst case
- For updates, probability of node spilt is very small (assuming equal search-key distribution)

B^+ tree Extensions

- B^+ tree File Organization (with respect to clustering search-key):
  - Leaf nodes of the tree stores records instead of pointers
  - Can be used for records with large data types (clob, blob)



B^+ tree Extensions

- Secondary Indices and File Locations:
  - Updates on B^+ tree file organization may change record location even if it is not modified
  - All secondary indices to the record should be updated

- Solution:
  - Instead of pointers to records, store primary search-key attribute.
  - Example: instructor table with ID as clustering search-key index, for department_name secondary search-key index, for every department_name store the ID of instructions instead of pointers to records
  - No cost for updates, but increases cost of accessing data using secondary index

B^+ tree Extensions

- Indexing strings: If we want to create a B^+ tree index on strings:
  - Strings can be of variable length
  - Long string index can lead to small branching (hence long paths in the tree)
- Branching (fan out) can be increased using prefix compression:
  - Do not store entire search-keyat non-leaf nodes
  - Store only the prefix of the string that is sufficient to distinguish between the search-key strings

B^+ tree Extensions

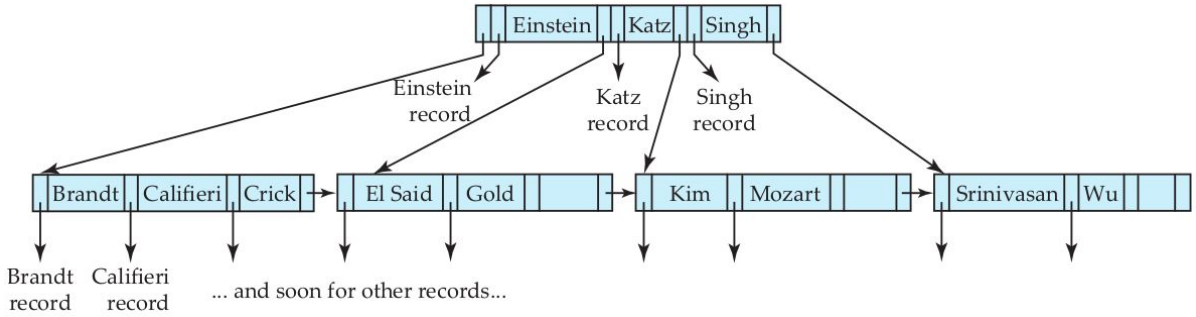
- Bulk loading of B^+ trees
  - Creating a secondary index from scratch takes time, since each insertion in the order of clustering index might access different blocks in the created B^+ tree

- Efficient way to do it (Bulk loading)
  - Create a temporary relation containing the index entries
  - Sort the temporary file using the index entries
  - Then all entries that goes into one leaf will appear consecutively
  - So all blocks (nodes) need to be brought to main memory once

  - With sorted entries, the tree can be built bottom-up.
    - Break the entries into leaf blocks
    - The minimum entry in each block along with the pointer will be the entry for the next level

B^+ tree Extensions

- B tree index Files
  - Similar to B^+ trees
  - Eliminates redundancy
  - Deletion is more complicated (since deletion can happen in non-leaf nodes too)
  - Most implementations use B^+ trees (but might be called B trees)



Multiple-Key Access

- Find ID of all instructors in the Finance department whose salary is 80000
  - SELECT ID FROM instructor WHERE dept_name = 'Finance' AND salary= 80000;
- Three strategies to use index:
  - Use the index on dept_name to find all records pertaining to the Finance department. Examine each such record to see whether salary= 80000.
  - Use the index on salary to find all records pertaining to instructors with salary of 80000. Examine each such record to see whether the department name is "Finance".
  - Use the index on dept name to find pointers to all records pertaining to the Finance department. Also, use the index on salary to find pointers to all records pertaining to instructors with a salary of 80,000. Take the intersection of these two sets of pointers.
- Third option seems to be a good option
  - But need not be if the following happens:
    - There are many instructors in Finance department
    - There are many instructors who get 80000 salary
    - There are very few instructors in Finance department who get 80000 salary
- Instead create (dept_name, salary) together as the search-key index

Multiple-Key Access

- Suppose we have (dept_name, salary) as search-key index
- Searching only for ‘Finance’ department is same has having range query on (‘Finance’, - ∞) to (‘Finance’, + ∞)

- There are some drawbacks
  - SELECT ID FROM instructor WHERE dept_name < ‘Finance’ AND salary < 80000;
  - Each record is likely to be in a different block because of the ordering of the records

- R-tree is used to handle multiple-key index
  - Generalization of B^+ trees to handle indexing in multiple dimensions

Multiple-Key Access

- Covering indices
  - Stores values of some attributes along with the pointers to the record
    - Example: Storing salary value for every ID as search-key
      - Same effect as having (ID, salary) as search-key but can have larger fanout (hence short trees)