Chapter 14:Indexing

Dr. Abu Raihan Mostofa Kamal

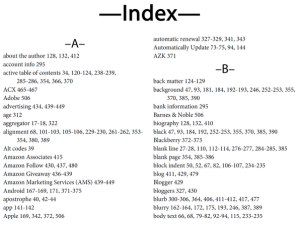
November 20, 2023

Table of Contents

Indexing: Basic Concepts

Indexing

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking.

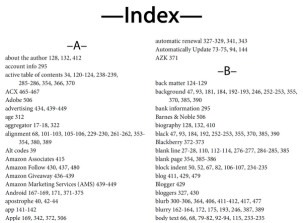


—Index—

–A–

–B–

Indexing: Basic Purpose



—Index—

–A–

–B–

- **Fast and efficient** processing of queries in databases.
- These are sophisticated techniques that **reduce the overhead of reading the entire contents** of the database. (Typically records of database are enormous in number)

Indexing: Implementation Complexity

An Example

Suppose we need to search a students information based on its ID. Now one simple solution is to sort them based on the search key (i.e. ID).

But it may not work in reality, because:

1. The index would itself be **very large** (hard to fit in main momory for millions of records)
2. Even though keeping the index sorted reduces the search time, finding a student can still be rather **time-consuming**.
3. Updating a sorted list whenever students are **added or removed** from the database can be **very expensive**.

Hence, more **sophisticated indexing techniques** are used in database systems.

Indexing: Broad Classification

There are two basic kinds of indices:

1. **Ordered indices**. Based on a sorted ordering of the values.
2. **Hash indices**. Based on a uniform distribution of values across a range of **buckets**. The bucket to which a value is assigned is determined by a function, called a **hash function**.

Note: Concentration will be on ordered indexing.

Indexing Technique: Which one to use?

No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

1. **Access types:** The types of access that are supported efficiently. Access types can include finding records with a **specified attribute value** and finding records whose attribute values fall in **a specified range**.
2. **Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.
3. **Insertion and Deletion time:** The time it takes to insert a new data item or to remove a data. This value includes the time it takes to update the index structure.
4. **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

Search Key

Search Criteria

An attribute or set of attributes used to look up records in a file is called a search key. Note that this definition of key **differs** from that used in primary key, candidate key, and superkey. Multiple search keys are common in use.

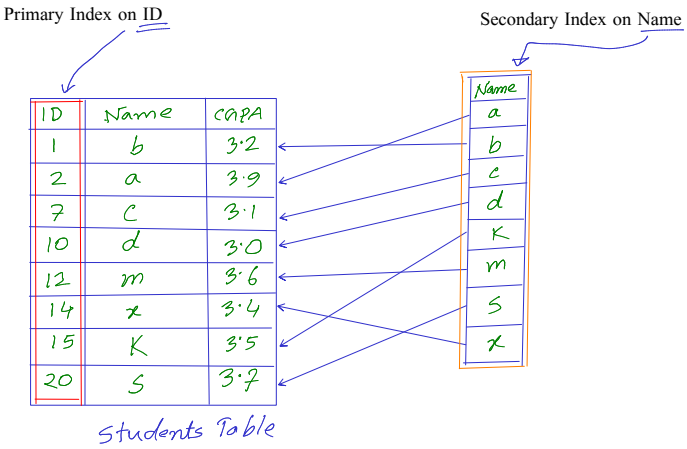
Ordered Index: Types

Ordered index can be of 2 types:

1. **Clustering Index:** Cluster index is a type of index which sorts the data rows in the table on their key values. In the Database, there is only one clustered index per table. A clustered index defines the order in which data is stored in the table which can be sorted in only one way. So, there can be an only a single clustered index for every table. In an RDBMS, usually, the primary key allows you to create a clustered index based on that specific column.This also called **Primary Index**.
2. **Nonclustering Indices:** A Non-clustered index stores the data at one location and indices at another location. The index contains pointers to the location of that data. A single table can have many non-clustered indexes as an index in the non-clustered index is stored in different places. Called **Secondary Index**.

(Definitions have been adopted from www.guru99.com)

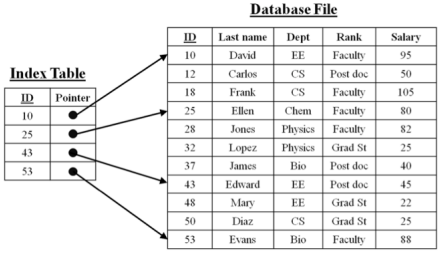
Primary and Secondary Index: Concepts



Dense and Sparse Index

Index Entry

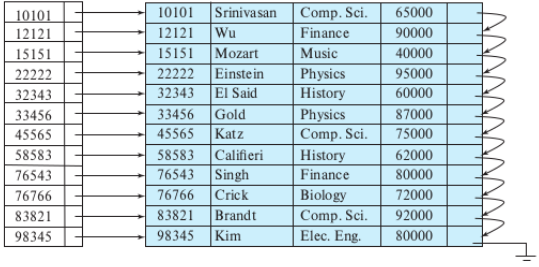
An index entry, or index record, consists of a **search-key value** and **pointers** to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.



Dense Index

Dense Index: Definition

In a dense index, **an index entry** appears **for every search-key value** in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the **first data record** with that search-key value.

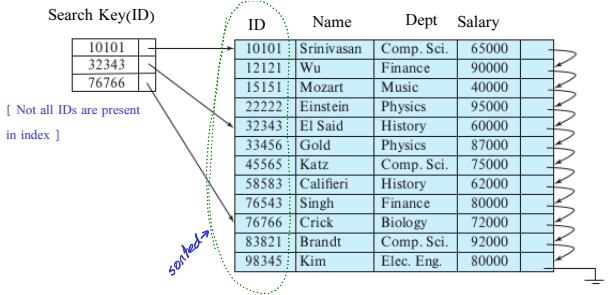


10101	10101	Srinivasan	Comp. Sci.	65000
12121	12121	Wu	Finance	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	Comp. Sci.	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Finance	80000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

Sparse Index

Sparse Index: Definition

In a sparse index, an index entry appears for **only some of the search- key values**. Sparse indices can be used only if the relation is stored in **sorted order** of the search key; that is, if the index is a clustering index.



Dense and Sparse Index: Pros and Cons

Dense Index:

- It is generally **faster** to locate a record if we have a dense index rather than a sparse index

Sparse Index:

- Sparse indices have advantages over dense indices in that they **require less space** and they impose **less maintenance overhead** for insertions and deletions.

Access Time and Space Overhead: A trade-off

- There is a trade-off that the system designer must make between access time and space overhead.
- Trade-off depends on the specific application
- The **dominant** cost in processing a database request is the time that it takes to **bring a block from disk into main memory**.
- Once we have brought in the block, the time to scan the entire block is negligible.

Multilevel Indices

Why?

Large indices are stored as sequential files on disk because of its size (can not be directly loaded into Main Memory).

For instance, Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4-kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space.(Main Memory has other functions as well)

Multilevel Indices (Cont.)

- If an index is **small** enough to be **kept entirely in main memory**, the search time to find an entry is low.
- However, for larger size index, the search for an entry in the index then requires **several disk-block reads**.
- Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy **b** blocks, binary search requires as many as $\lceil log_2 b \rceil$ blocks to be read. Even this is quite high for a very large database.
- To deal with this problem, we treat the index just as we would treat any other sequential file, and we construct a **sparse outer index** on the original index, which we now call the inner index.

Multilevel Indices (Cont.): A simple Analogy

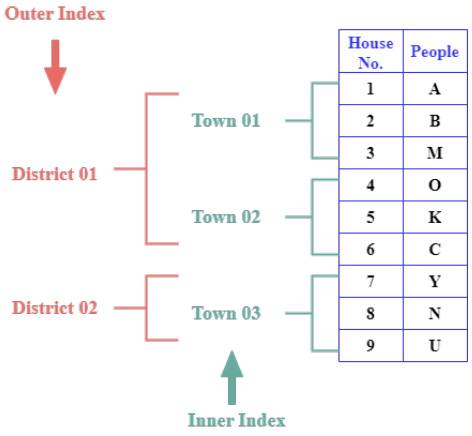


Figure: Multi-level index: Analogy

Multilevel Indices (Cont.):

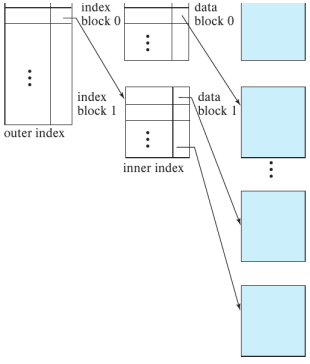


Figure: Multi-level index: Basic Idea

B+ Tree

Why B+ tree?

The primary drawback of index-sequential file organization is declining performance with file growth, both for index lookups and for sequential scans through the data, necessitating frequent file reorganization.

In contrast, the **B+-tree** index structure, among various options, remains efficient even with frequent data insertions and deletions.

B+ Tree (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.

Example of B+ Tree

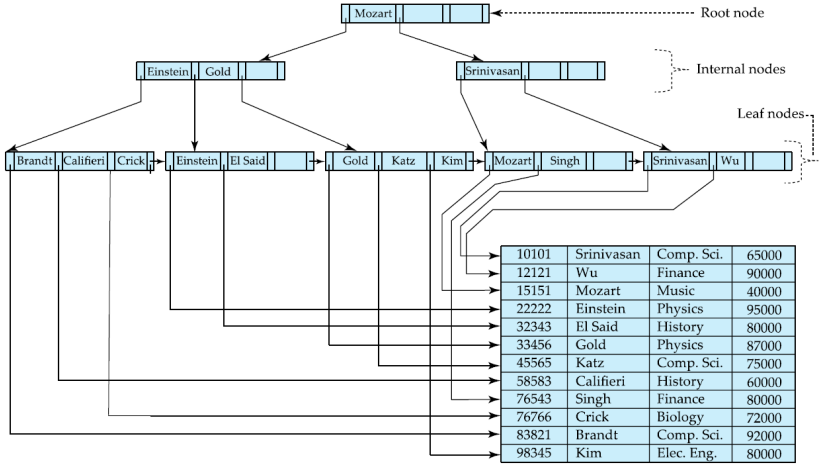
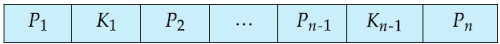


Figure: B+ Tree

B+ Tree Node Structure

- Typical node



  - 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

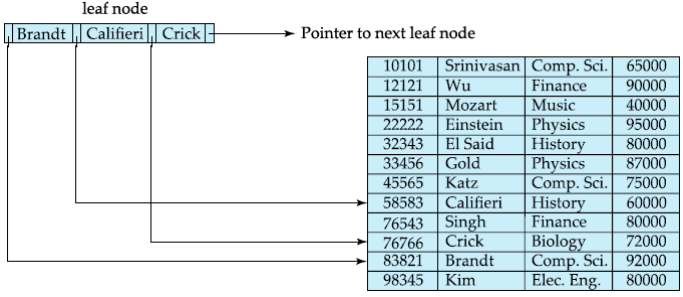
$$K_1 < K_2 < K_3 < ... < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+ Trees

Properties of a leaf node:

- For $i = 1, 2, ..., n˘1$, 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 Ki
- All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

Example of B+ tree with n=6

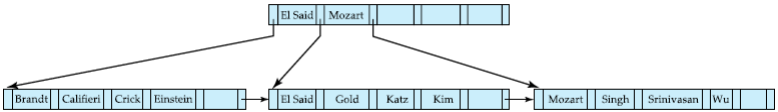


Figure: B+-tree for instructor file (n= 6)

- 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 ($((n/2\rceil$ and n with $n=6$).
- Root must have at least 2 children.

Observations about B+ trees

- The non-leaf levels of the B+ tree form a hierarchy of sparse indices.
- The B+ tree contains a relatively small number of levels. As we can see,
  - 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 $\left\lceil \log_{\lceil n/2 \rceil}(K) \right\rceil$, 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.

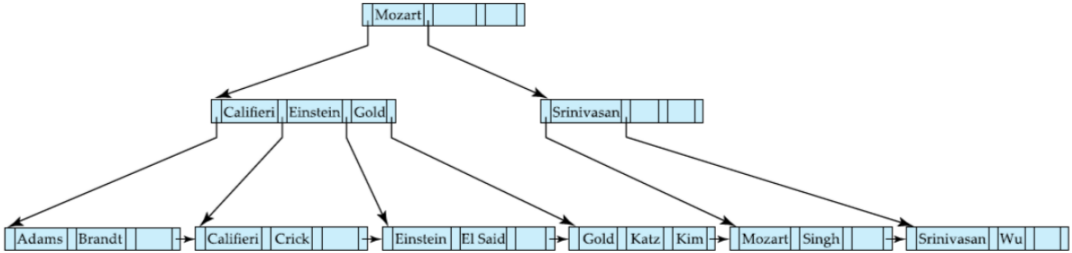
Query on B+ Tree

```
function find(v)
/* Assumes no duplicate keys, and returns pointer to the record with
 * search key value v if such a record exists, and null otherwise */
    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 Set C = C.P_i /* v < C.K_i */
    end
    /* C is a leaf node */
    if for some i, K_i = v
        then return P_i
        else return null ; /* No record with key value v exists*/
```

Query on B+ Tree Example

Initially,

v=Katz
C=Root



Query on B+ Tree Example (Cont.)

Initially, **1st iter:**

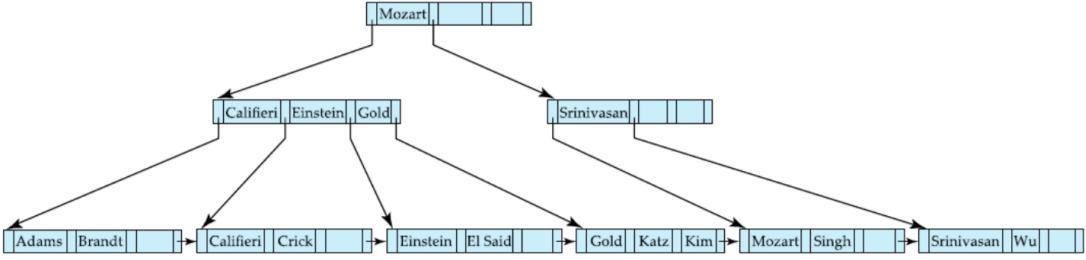
else Set $C = C.P_i$ /* $v < C.K_i$ */

v=Katz C=Root; is not leaf

C=Root Kazt <Mozart (as only one key)

Pm = Pointer prior to Mozart

C=Node pointed by Pm



Query on B+ Tree Example (Cont.)

Initially, **1st iter:** **2nd iter:**

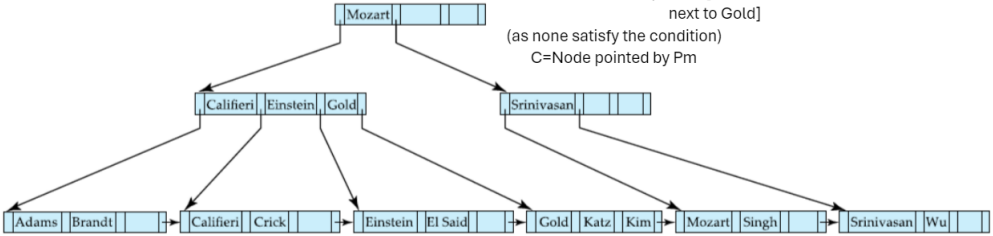
else Set $C = C.P_i$ /* $v < C.K_i$ */

if there is no such number i then begin
Let P_m = last non-null pointer in the node
Set $C = C.P_m$

v=Katz
C=Root

C=Root; is not leaf
Kazt <Mozart (as only one key)
Pm = Pointer prior to Mozart
C=Node pointed by Pm

C=Node pointed by Pm; is not leaf
Kazt >Califieri
Kazt >Einstein
Kazt >Gold
Pm=last not null pointer [Pointer next to Gold]
(as none satisfy the condition)
C=Node pointed by Pm



Query on B+ Tree Example (Cont.)

Initially,

v=Katz
C=Root

1st iter:

else Set $C = C.P_i$ /* $v < C.K_i$ */

C=Root; is not leaf
Kazt <Mozart (as only one key)
Pm = Pointer prior to Mozart
C=Node pointed by Pm

2nd iter:

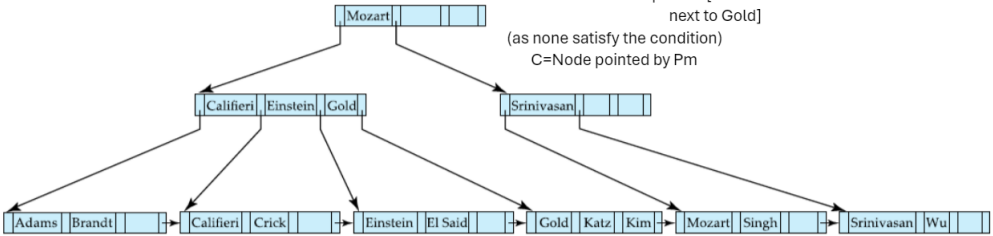
if there is no such number i then begin
Let P_m = last non-null pointer in the node
Set $C = C.P_m$

C=Node pointed by Pm; is not leaf
Kazt >Califieri
Kazt >Einstein
Kazt >Gold
Pm=last not null pointer [Pointer next to Gold]
(as none satisfy the condition)
C=Node pointed by Pm

3rd iter:

if for some i, $K_i = v$
then return P_i

C=Node pointed by Pm; is leaf
Kazt!=Gold
Kazt=Kazt
P=Pointer prior to Kazt
return P



Query on B+ Trees Another Example (Cont.)

Initially,

v=Mozart
C=Root

1st iter:

else if ($v = C.K_i$) then Set $C = C.P_{i+1}$

C=Root; is not leaf
Mozart =Mozart (as only one key)
Pm = Pointer next to Mozart
C=Node pointed by Pm

2nd iter:

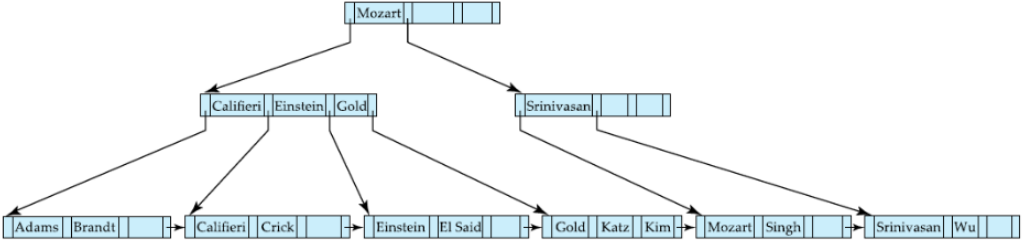
else Set $C = C.P_i$ /* $v < C.K_i$ */

C=Node pointed by Pm; is not leaf
Mozart <Srinivasan
Pm=Pointer prior to Srinivasan
C=Node pointed by Pm

3rd iter:

if for some i, $K_i = v$
then return P_i

C=Node pointed by Pm; is leaf
Mozart = Mozart
P=Pointer prior to Kazt
return P



Balanced binary tree Vs B+ Tree

Balanced binary tree

A balanced binary tree with 1 million search key values,

- at most $\log_2(1,000,000) = 20$ nodes are accessed in a lookup traversal from root to leaf.

B+ tree

In contrast, a B+ tree 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.

The above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Non-Unique Search 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

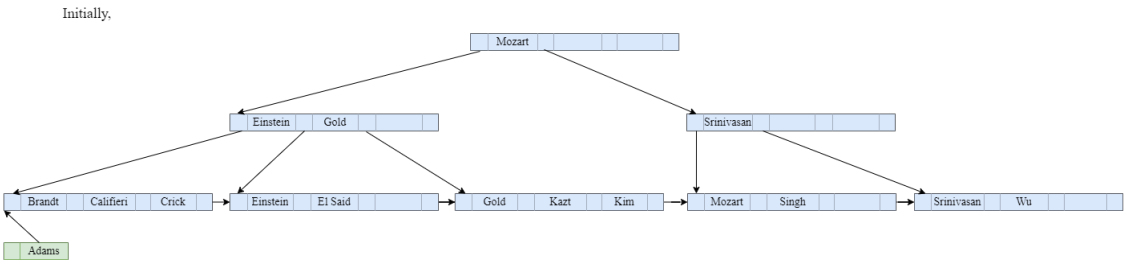
B+ Trees: Insertion (Simplified steps)

Assume the record has already been added to the file. Let P_r be the pointer to the record, and v be the search key value of the record.

- Find the leaf node in which the search-key value would appear.
  1. If that is a non-full leaf node (number of search keys in that node is less than $(n-1)$), insert (v, P_r) pair in the leaf node
  2. Else, split the node (along with the new (v, P_r) entry) and propagate updates to parent nodes till a non-full node is found. That is,
     1. 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.
     2. let k be the least key value and P_s be the pointer of the new node. Insert (k, P_s) in the parent of the node being split.
     3. If the parent is full, split it and propagate the split further up.

B+ Trees Insertion Example

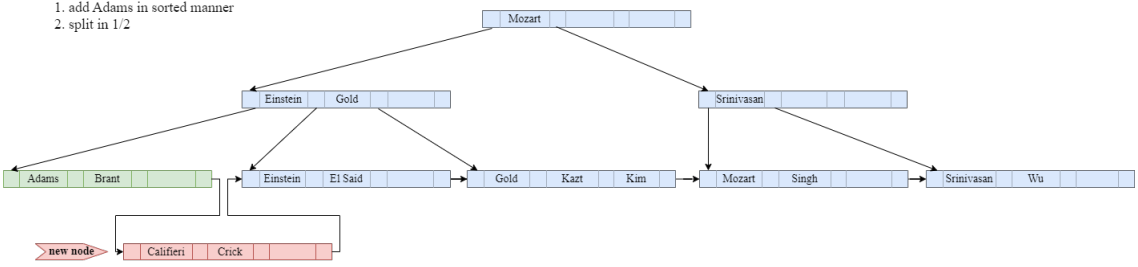
Let's say we want to add Adams.



B+ Trees Insertion Example (Cont.)

Step01:
Not a non-full node where new key will be added,

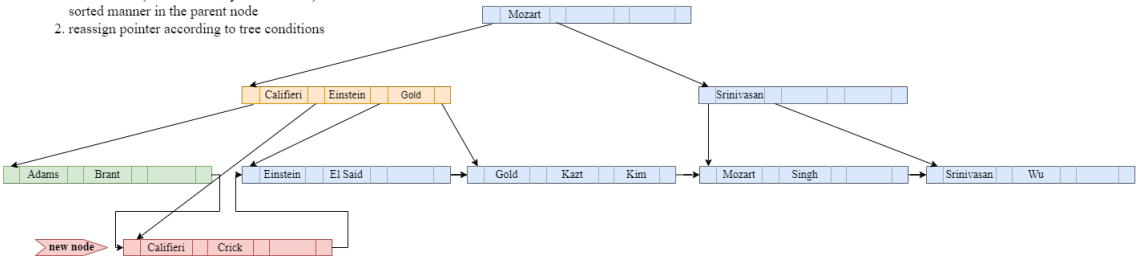
1. add Adams in sorted manner
2. split in 1/2



B+ Trees Insertion Example (Cont.)

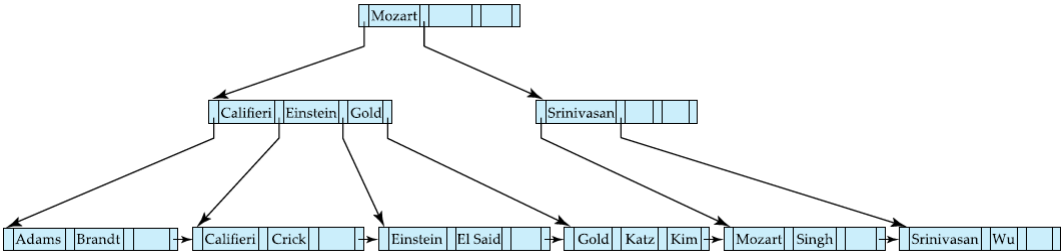
Step02:
Parent is a non-full node,

1. add Califieri(least search key of new node) in sorted manner in the parent node
2. reassign pointer according to tree conditions



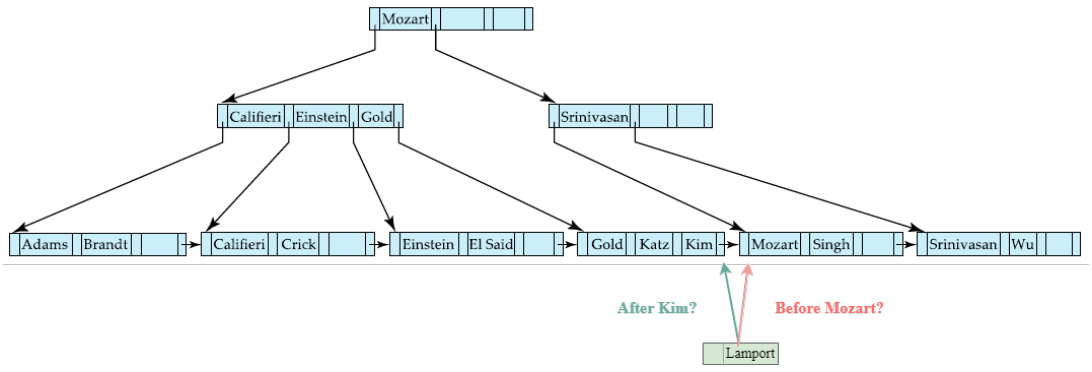
B+ Trees Insertion Example (Cont.)

So, the final state is,



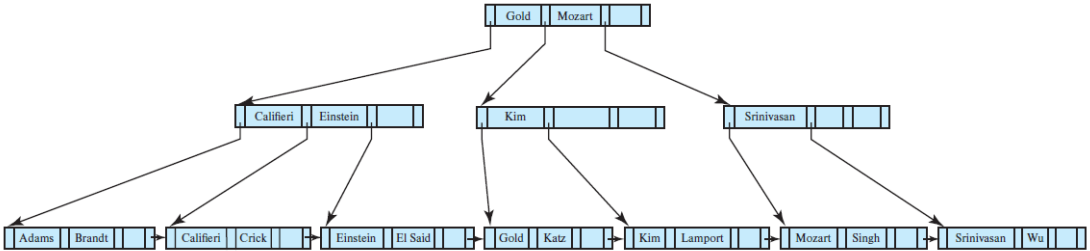
B+ Trees Insertion Another Example

Let's say, now we want to add Lamport.



B+ Trees Insertion Another Example (Cont.)

Final State,



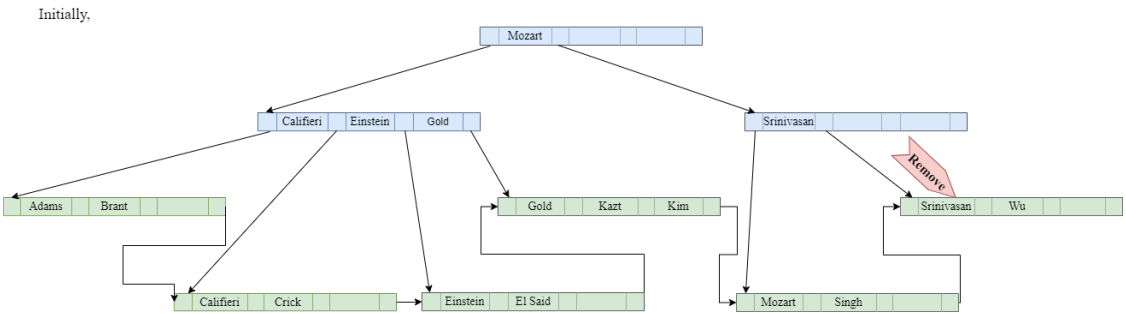
B+ Trees: Deletion (Simplified steps)

Assume the record has already been deleted from the file. Let P_r be the pointer to the record, and v be the search key value of the record.

- Remove (Pr, V) from the leaf node
  1. If the node has too few (children less than $\lceil n/2 \rceil$) search key due to the removal, and fit into a single node, then merge siblings:
     1. Insert all the search-key values of two nodes into a single node (the one on the left) and delete the other node
     2. Delete the pair (K_{i-1}, P_i) from its parent, where P_i is the points to the deleted node, recursively using the above procedure as necessary.
  2. Else if the node has too few search key due to the removal, but can't fit into a single node, then redistribute pointers:
     1. Redistribute the pointers between the node and closest sibling such that both have more than the minimum number of entries.
     2. Update the corresponding search-key value in the parent of the node.

B+ Trees Deletion Example

Let's say we want to delete Srinivasan.

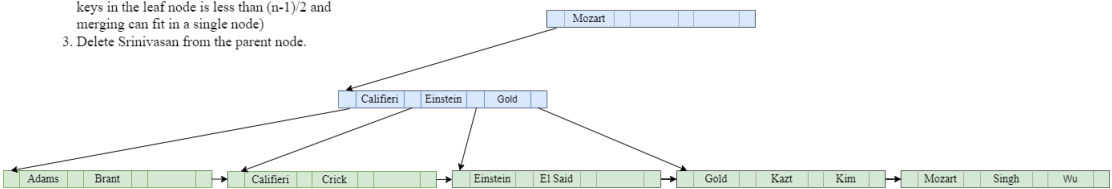


Note:green color for leaf nodes.

B+ Trees Deletion Example (Cont.)

Step01,

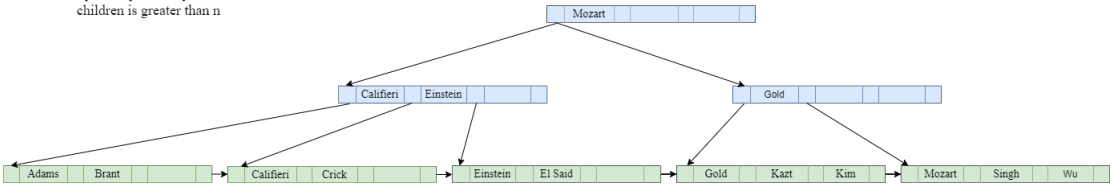
1. Remove Srivivasan.
2. Merge Wu with Sibling node (number of search keys in the leaf node is less than (n-1)/2 and merging can fit in a single node)
3. Delete Srinivasan from the parent node.



B+ Trees Deletion Example (Cont.)

Step02,

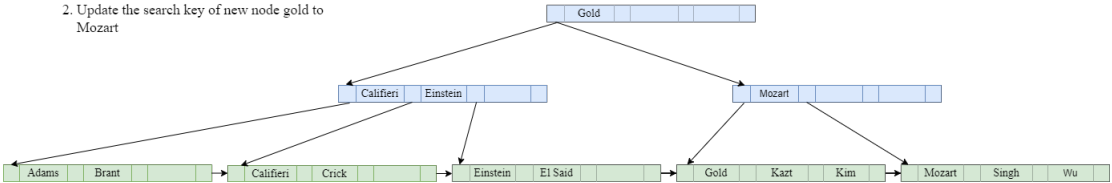
1. Split the parent key in 1/2 as the number of children is greater than n



B+ Trees Deletion Example (Cont.)

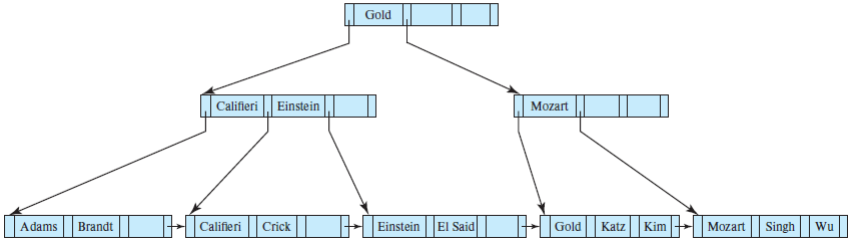
Step03:

1. Update the search key of root from Mozart to Gold.
2. Update the search key of new node gold to Mozart



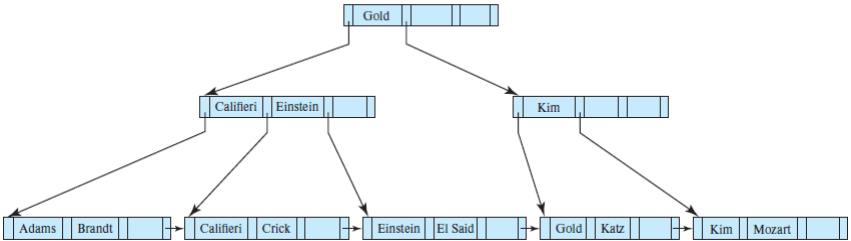
B+ Trees Deletion Example (Cont.)

So, the final state is,



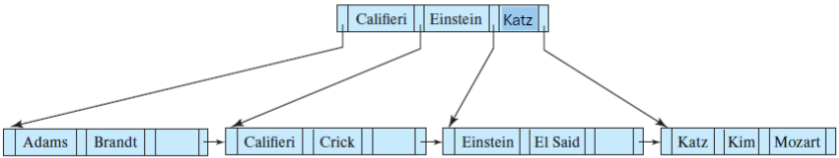
B+ Trees Deletion Another Example

After deleting Singh and Wu



B+ Trees Deletion Another Example

After deleting Gold



B+ Trees Update

How to Update B+ tree?

Updates to a record can be modeled as deletion of the old record followed by the insertion of the updated record

Complexity of Insert and Delete from a B+ Tree

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to the height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O\left(\log_{\lceil n/2 \rceil}(K)\right)$
- 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
  - $2/3$ rds with random, $1/2$ with insertion in sorted order