

Chapter 11: Indexing and Hashing

Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

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



- 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”.

Index Evaluation Metrics

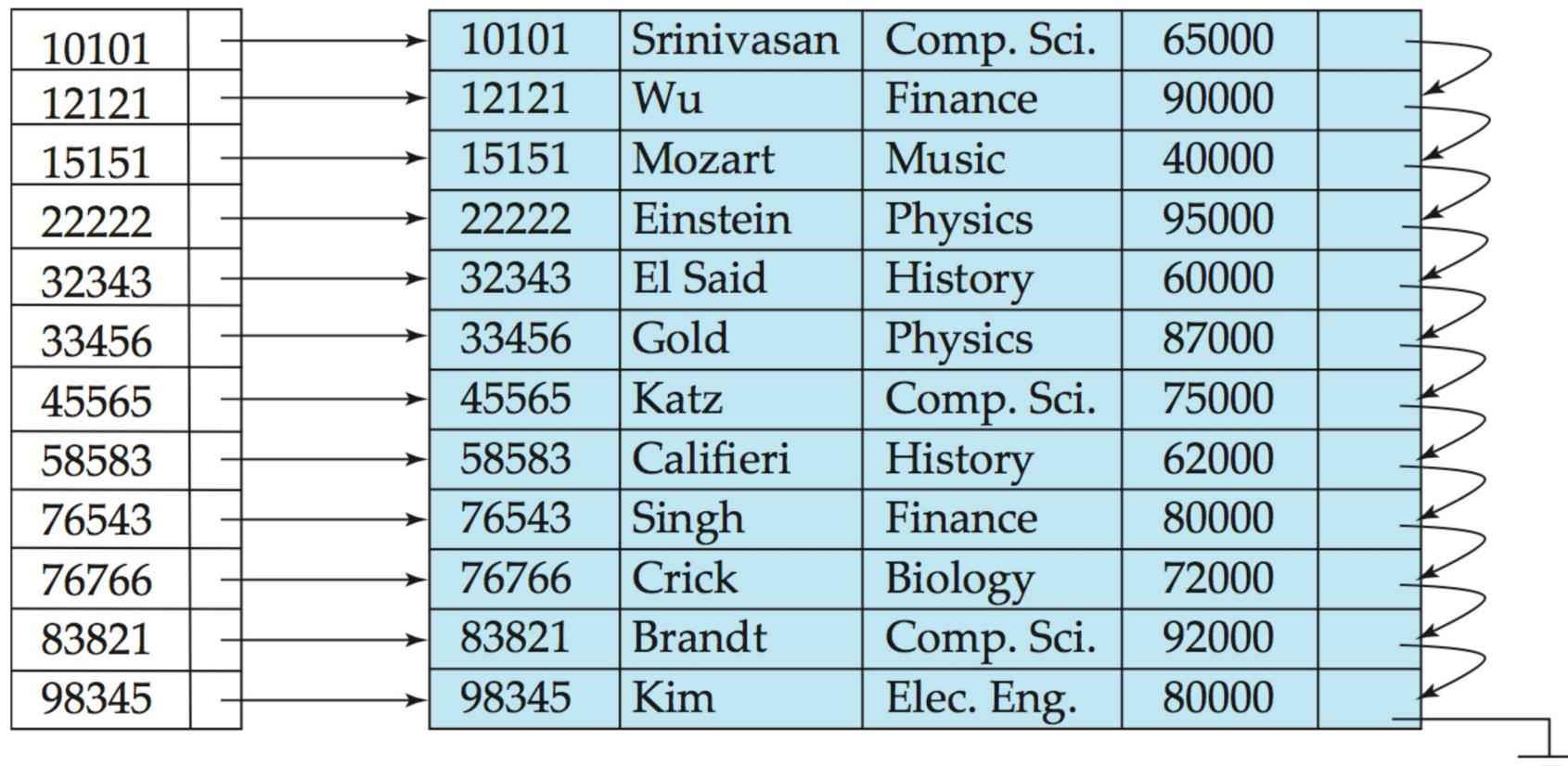
- Access types supported efficiently. E.g.,
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index**, index entries are stored on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering 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 **non-clustering index**.
- **Index-sequential file:** ordered sequential file with a primary index.

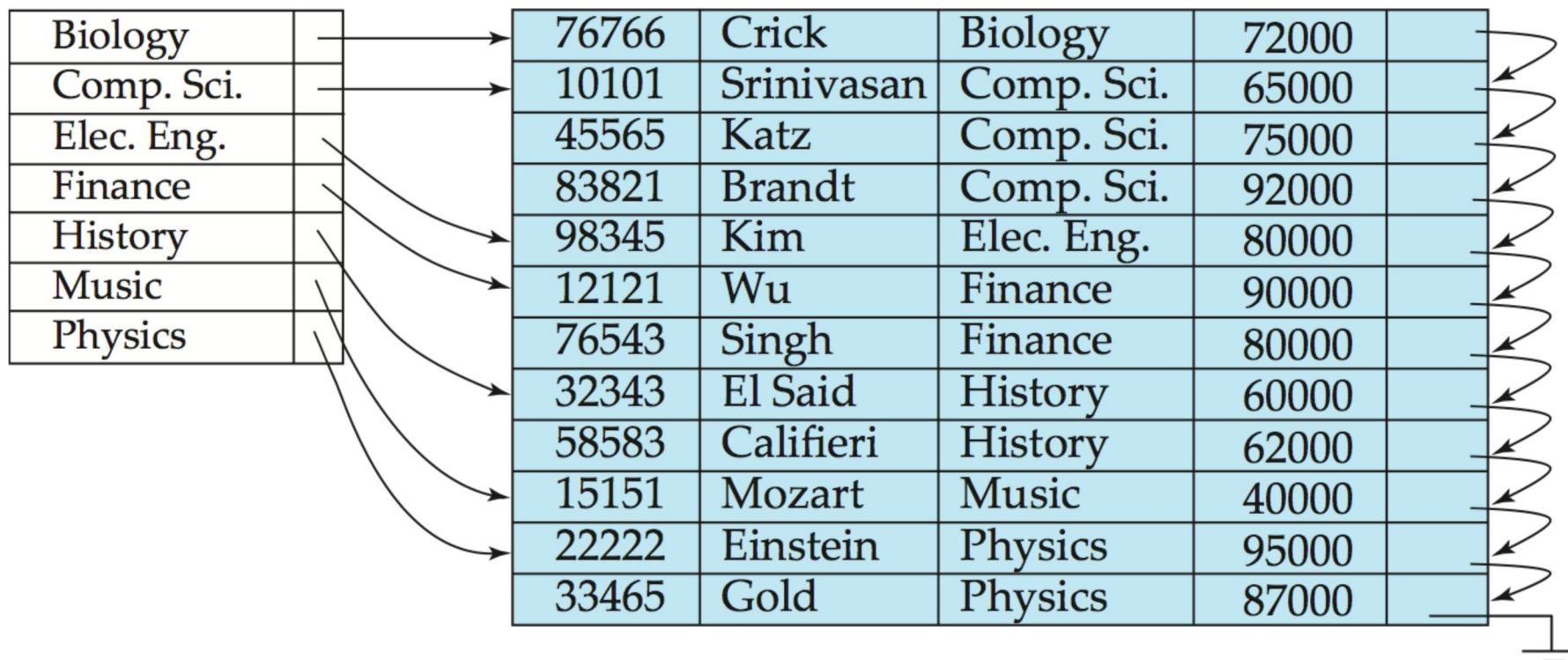
Dense Index Files

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



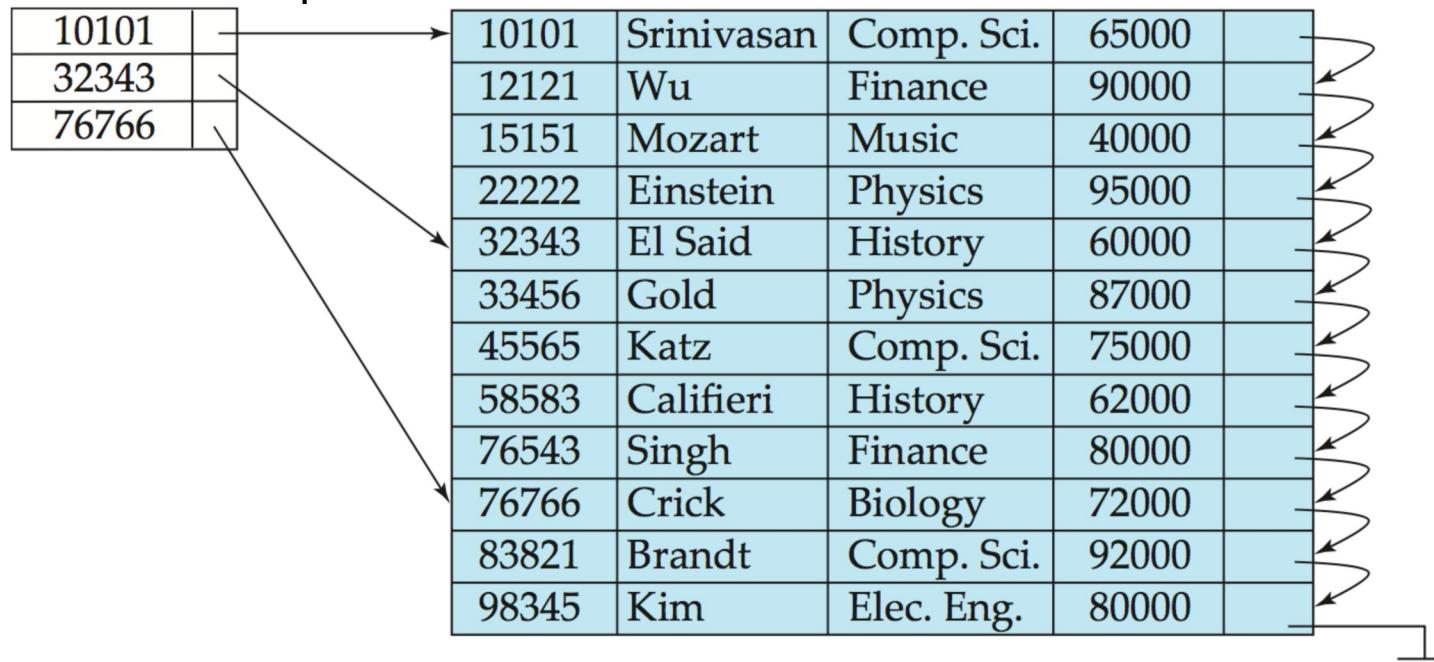
Dense Index Files (Cont.)

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



Sparse Index Files

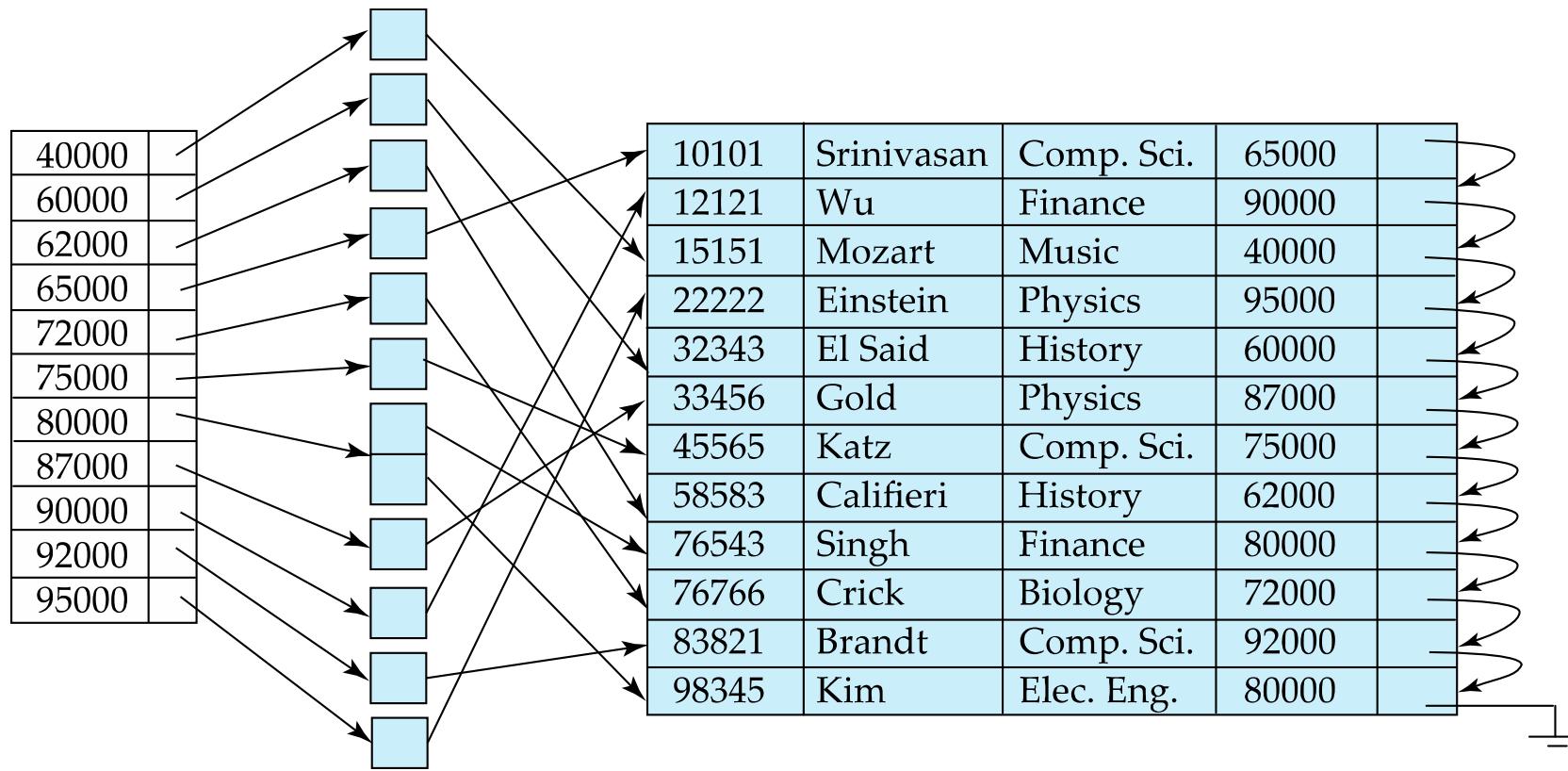
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to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



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:** sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

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

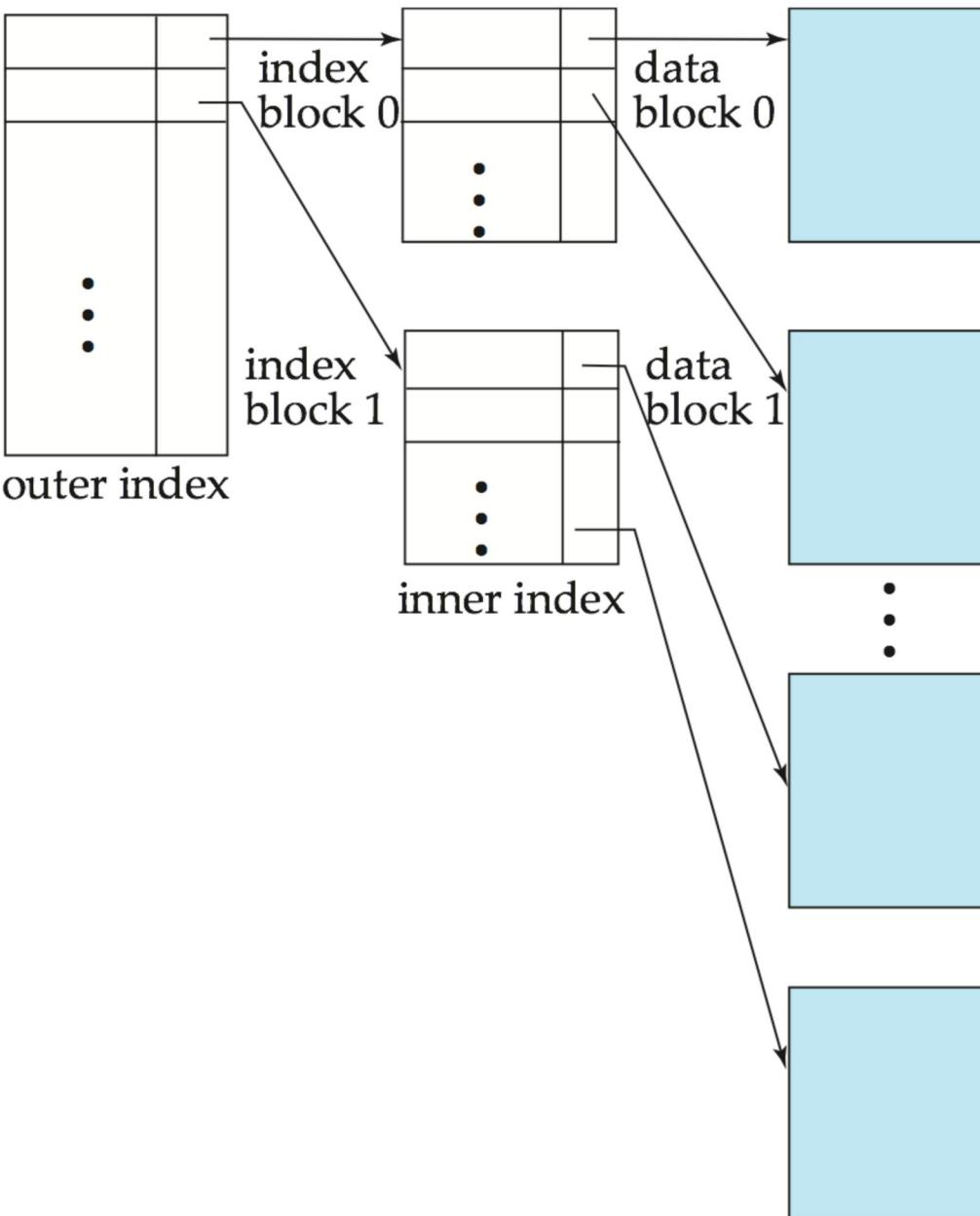
Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
 - Each record access may fetch a new block from disk
 - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

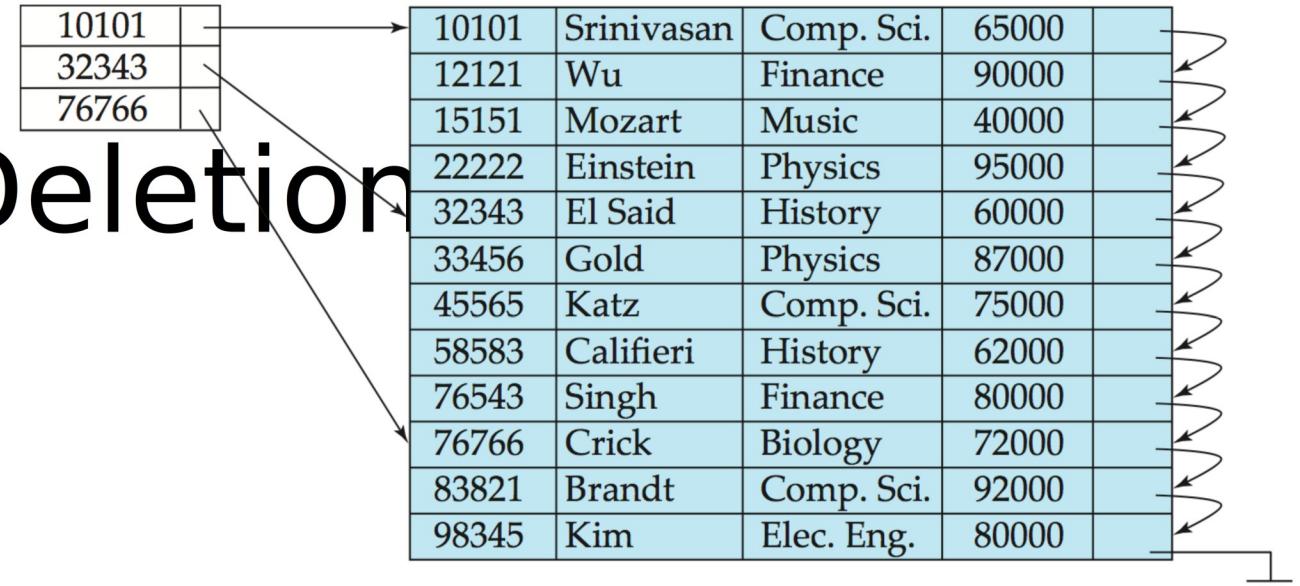
Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of primary index
 - inner index – the primary 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



- 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 appearing in the record to be inserted.
 - **Dense indices** - if the search-key value does not appear in the index, insert it.
 - **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

Secondary Indices

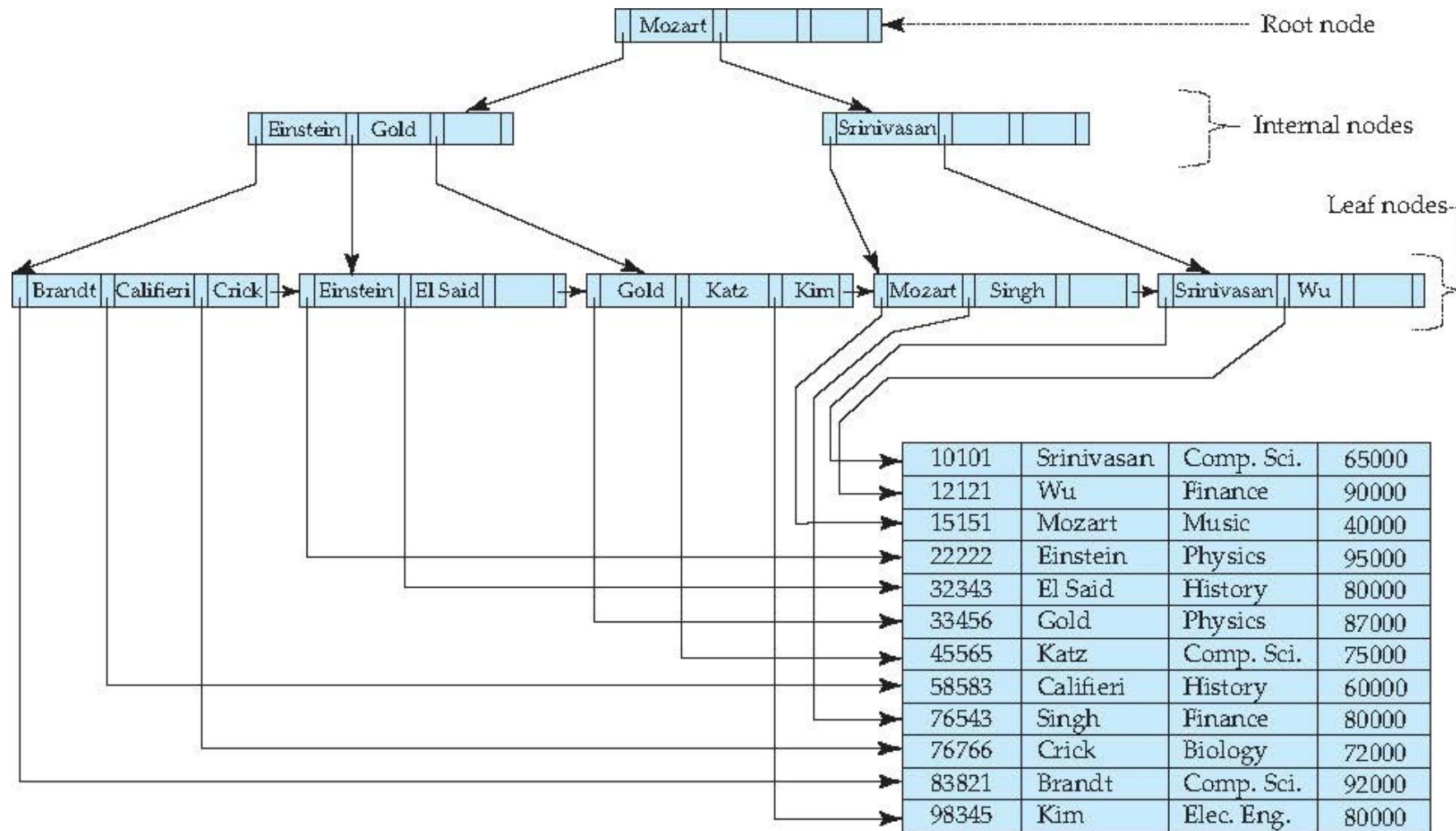
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
 - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
 - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

B⁺-Tree Index Files

B⁺-tree indices are an alternative to indexed-sequential 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



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	\dots	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

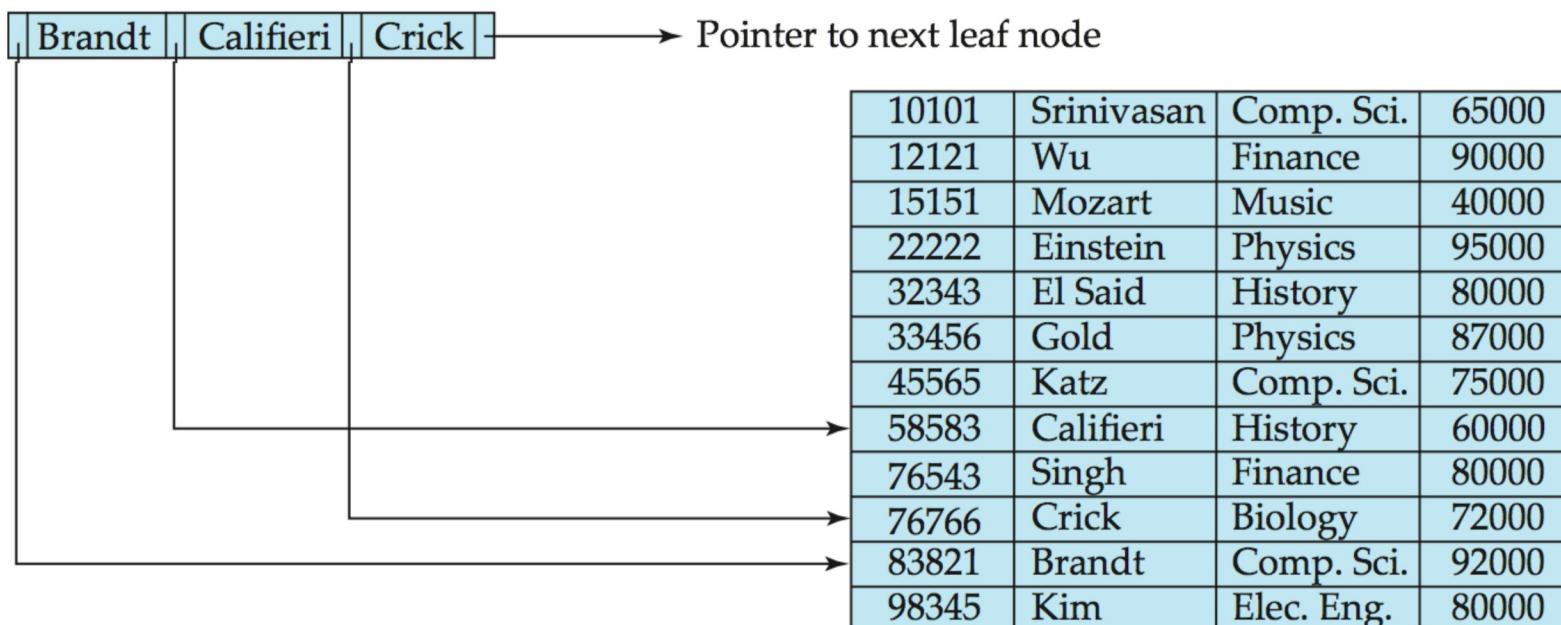
$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B⁺-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, 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
leaf node

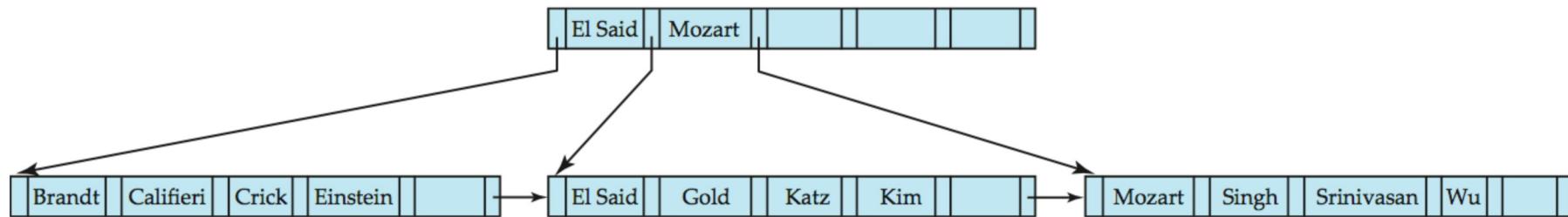


Non-Leaf Nodes in B⁺-Trees

- 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}

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

Example of B⁺-tree



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

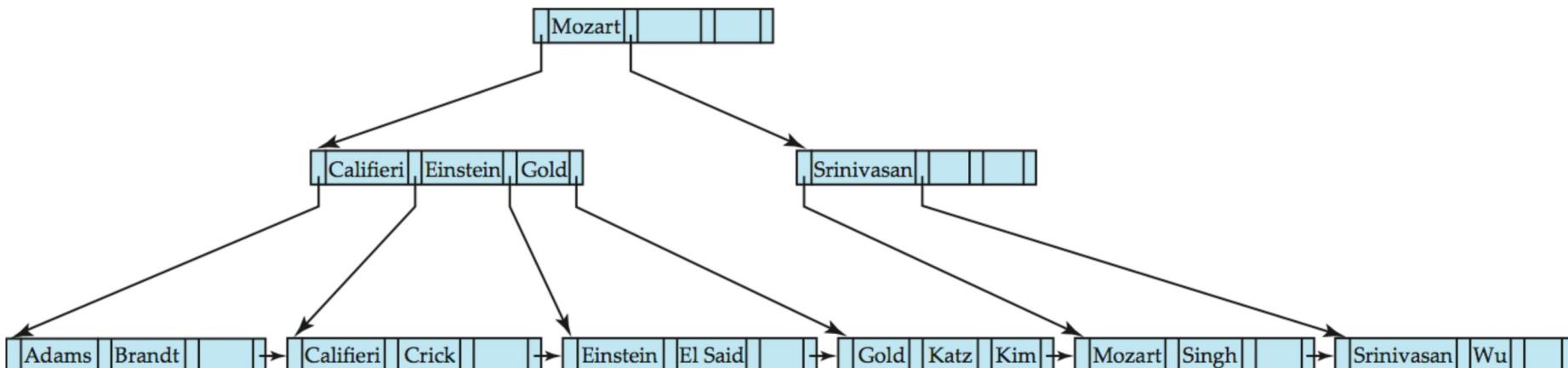
- Leaf nodes must have between 3 and 5 values ($\lceil(n-1)/2\rceil$ and $n - 1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil(n/2)\rceil$ and n with $n = 6$).
- Root must have at least 2 children.

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

- Find record with search-key value V .
 1. $C = \text{root}$
 2. While C is not a leaf node {
 1. Let i be least value s.t. $V \leq K_i$.
 2. If no such exists, set $C = \text{last non-null pointer in } C$
 3. Else { if ($V = K_i$) Set $C = P_{i+1}$ else set $C = P_i$ }
 3. Let i be least value s.t. $K_i = V$
 4. If there is such a value i , follow pointer P_i to the desired record.
 5. Else no record with search-key value k exists.



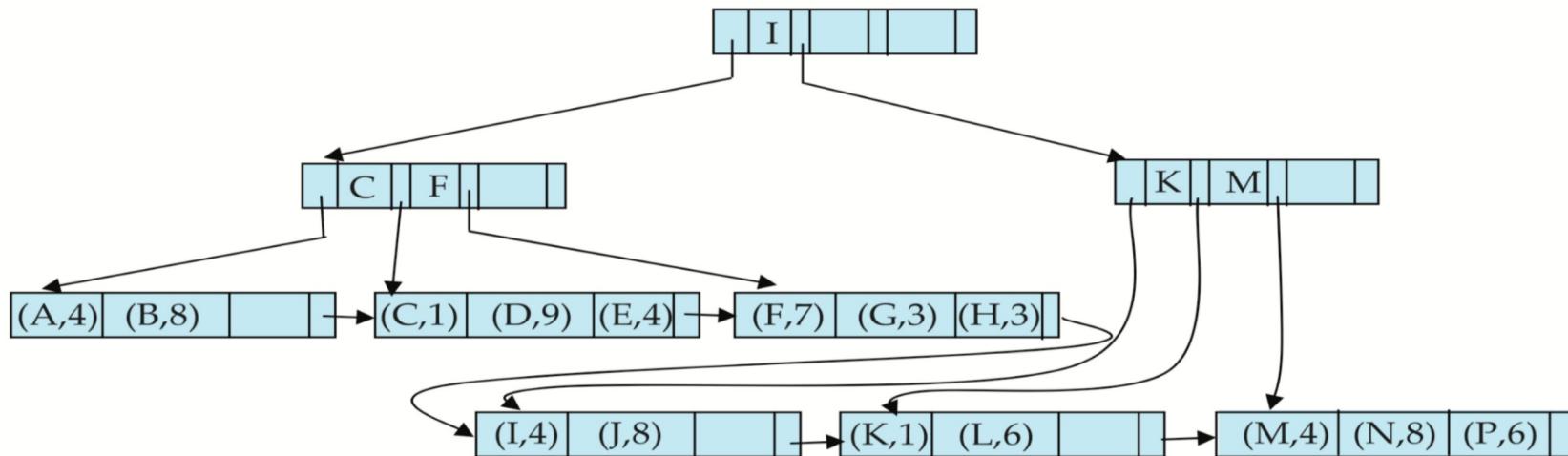
Queries on B+-Trees (Cont.)

- 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.
- 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

B⁺-Tree File Organization

- Index file degradation problem is solved by using B⁺-Tree indices.
- Data file degradation problem is solved by using B⁺-Tree File Organization.
- The leaf nodes in a B⁺-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
 - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index.

B⁺-Tree File Organization (Cont.)



Example of B⁺-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
 - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lceil \frac{2n}{3} \rceil$ entries

Other Issues in Indexing

- **Record relocation and secondary indices**
 - If a record moves, all secondary indices that store record pointers have to be updated
 - Node splits in B⁺-tree file organizations become very expensive
 - *Solution:* use primary-index search key instead of record pointer in secondary index
 - Extra traversal of primary index to locate record
 - Higher cost for queries, but node splits are cheap
 - Add record-id if primary-index search key is non-unique

Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B+-tree requires ≥ 1 IO per entry
 - assuming leaf level does not fit in memory
 - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
 - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
 - insert in sorted order
 - insertion will go to existing page (or cause a split)
 - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B+-tree construction**
 - As before sort entries
 - And then create tree layer-by-layer, starting with leaf level
 - details as an exercise
 - Implemented as part of bulk-load utility by most database systems