B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

<table>
<thead>
<tr>
<th>p₁</th>
<th>k₁</th>
<th>p₂</th>
<th>...</th>
<th>pₑ₋₁</th>
<th>kₑ₋₁</th>
<th>pₑ</th>
</tr>
</thead>
</table>

(a)

| p₁ | b₁ | k₁ | p₂ | b₂ | k₂ | ... | pₑ₋₁ | bₑ₋₁ | kₑ₋₁ | pₑ |

(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.
### B-Tree Index File Example

B-tree (above) and B+-tree (below) on same data

#### B-Tree Index Files (Cont.)

- **Advantages of B-Tree indices:**
  - May use fewer tree nodes than a corresponding B+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.

- **Disadvantages of B-Tree indices:**
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B+-Tree
  - Insertion and deletion more complicated than in B+-Trees
  - Implementation is harder than B+-Trees.

- Typically, advantages of B-Trees do not out weigh disadvantages.
Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a $B^+$-tree requires $\geq 1$ 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
  - Implemented as part of bulk-load utility by most database systems

Indexing on Flash

- Random I/O cost much lower on flash
  - 20 to 100 microseconds for read/write
- Writes are not in-place, and (eventually) require a more expensive erase
- Optimum page size therefore much smaller
- Bulk-loading still useful since it minimizes page erases
- Write-optimized tree structures (discussed later) have been adapted to minimize page writes for flash-optimized search trees
Indexing in Main Memory

- Random access in memory
  - Much cheaper than on disk/flash
  - But still expensive compared to cache read
  - Data structures that make best use of cache preferable
  - Binary search for a key value within a large B+-tree node results in many cache misses
- B+-trees with small nodes that fit in cache line are preferable to reduce cache misses
- Key idea: use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array.

Hash-based Indexes
Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function \( h \) is a function from the set of all search-key values \( K \) to the set of all bucket addresses \( B \).
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.
Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing** (also called **closed hashing** or **open hashing** depending on the book you use)
  - An alternative, called **open addressing** (also called **open hashing** or **closed hashing** depending on the book you use) which does not use over-flow buckets, is not suitable for database applications.

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.
Deficiencies of Static Hashing

- In static hashing, function $h$ maps search-key values to a fixed set of $B$ of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Periodic rehashing
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets
Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B+-trees

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:
  ```sql
  select ID
  from instructor
  where dept_name = "Finance" and salary = 80000
  ```
- Possible strategies for processing query using indices on single attributes:
  1. Use index on `dept_name` to find instructors with department name Finance; test `salary = 80000`
  2. Use index on `salary` to find instructors with a salary of $80000; test `dept_name = "Finance"`.
  3. Use `dept_name` index to find pointers to all records pertaining to the “Finance” department. Similarly use index on `salary`. Take intersection of both sets of pointers obtained.