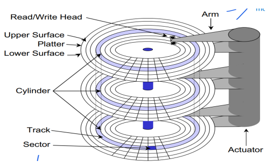
CS3223
AY22/23 Sem 2
github.com/SeekSaveServe

L1 - Data Storage

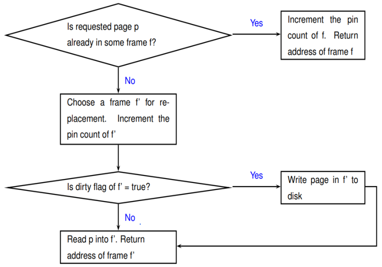
Magnetic Disks



- **Disk Access Time** Seek time + Rotational Latency + Transfer time
- **Response time** Queueing delay + Disk access time
- **Rotational Delay** $\frac{1}{2}\frac{60s}{RPM}$
- **Transfer Time** sectors on the same track * $\frac{TimePerRevolution}{SectorsPerTrack}$

Buffer Manager

- **Buffer pool** Main memory allocated for DBMS
- **pin count** is incremented upon pinning
- **dirty bit** is updated when the page is unpinned (if modified)
- Replacement is only possbile if pin count == 0



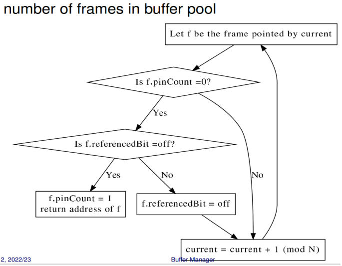
Replacement Policies

LRU Policy

- Maintains a queue of pointers to frames with pin count = 0

Clock Replacement Policy

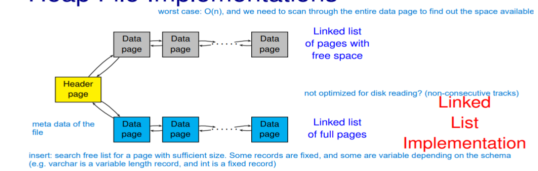
N = number of frames in buffer pool

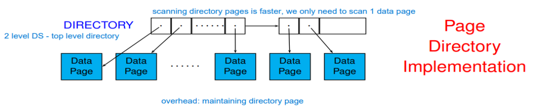


- Simplifies LRU with a second chance round robin system
- Each frame has a reference bit that is turned on when pin count reaches 0
- Repalces a page when referenced bit if off and pin count is 0

File Organisation

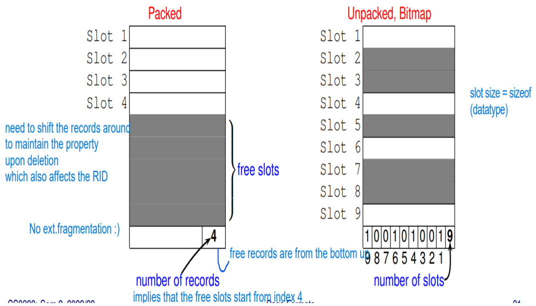
Heap File Implementations





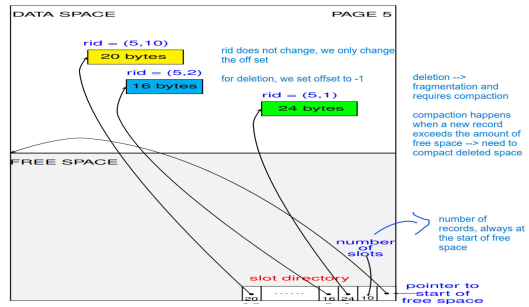
Page Formats: Fixed Length Records

- **Packed Organisation** Store records in contiguous slots
- **Unpacked Organisation** Uses a bit array to maintain free slots



Page Formats: Slotted Page (variable length record)

- Store records in slots of *(record offset, record length)*
- Record Offset: Offset of the record from the start of the page



Record Formats

- Fixed-Length Records
  - Fields are stored consecutively

F1	F2	F3	F4

- Variable-Length Records
  - Delimit fields with special symbols

F1	$	F2	$	F3	$	F4

  - Use an array of field offsets

o_1	o_2	o_3	o_4	F1	F2	F3	F4

- Each o_i is an offset to beginning of field Fi

L2 - Indexing

- A search key is a sequence of k attributes. If k ¿ 1, composite key
- A search key is an unique index if it is a candidate key
- An index is stored as a file

Format of data entries

- Format 1: k* is an actual data record with search value k
- Format 2: k* is the form (k, rid)
- Format 3: k* is the form (k, rid-list*)
- Note: Different formats affects the number of data entries stored in a page

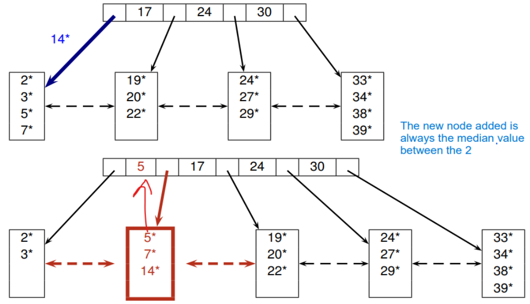
Clustered Vs Unclustered

- Clustered: Order of data entries is the same as the oreder of data records. Can only be built on ordered field (e.g. primary key)
- Unclustered: Order of data entries does not correspond to the order of data records
- The implication is that we can read an entire clustered page with 1 I/O
- B+ Tree: Format 1 is clustered, Format 2 and 3 can be clustered if data records are sorted on the search key
- Hash: Only format 1 is clustered since hashing do not store data entries in search key order

Tree Based Index - B+ Tree

- Leaf nodes are doubly linked and store Data Entries
- Internal nodes sotre index entries (p0, k, p1 ... pk, k, pk+1)
- Internal nodes contains m entries, m $\in$ [d, 2d] $\rightarrow$ space utilisation $\geq$ 50%
- Root contains m entries, m $\in$ [1, 2d]

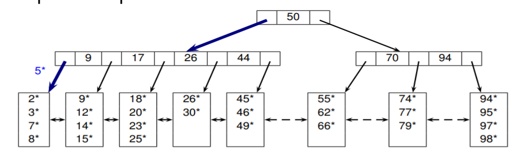
B+ Tree - Split Overflow Nodes



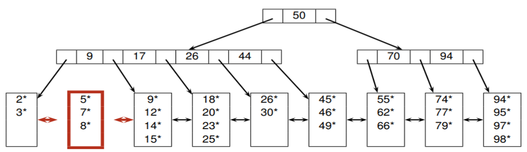
- Distribute d+1 entries to the new leaf node
- Create new entry index using smallest key in the new node (middle key)
- Insert new entry into parent node of overflowed node

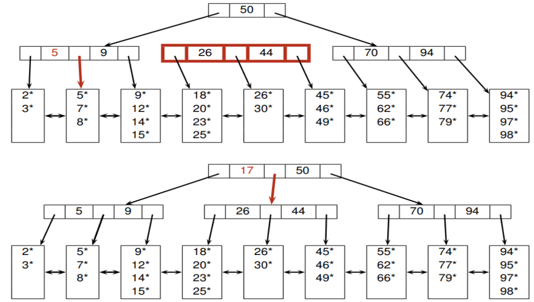
B+ Tree - Overflow Propagation

5 is pushed up



17 is pushed up

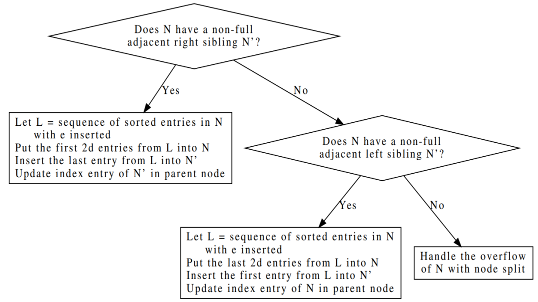




- Excess middle node is pushed updated to parent node

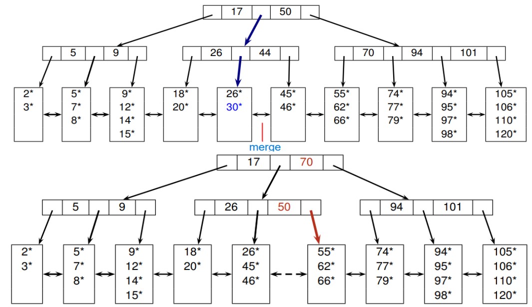
B+ Tree - Redistribution of data entries

- Two nodes are siblings if they have the same parent node



B+ Tree - Underflow

- Underflow occurs when a node has less than d entries
- Underflow is resolved by redistributing entries between siblings
- An underflow node is merged if each of its adjacent siblings have exactly d entries



B+ Tree - Bulk Loading

- Initiazing a B+ tree by insertion is expensive (need to traverse tree n times)
- 1. Sort all data entries by search key
- 2. Initialise B+ tree with an empty root page
- 3. Load data entries into leaf pages
- 4. In asc order, insert the index entry of each leaf page into the rightmost parent node

Hash Based Index

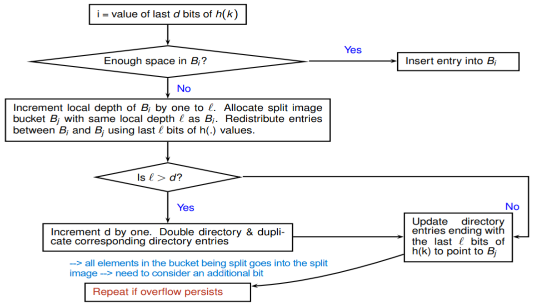
- Does not support range search, only equality queries

Static Hashing

- N buckets, each bucket has 1 primary page and $\geq$ 0 overflow pages
- To maintain performance, we need to routinely construct bigger hash tables and redistribute data entries

Dynamic Hashing - Extendible Hashing

- No overflow pages! A bucket can be thought of as a page
- At most 2 Disk I/Os for equality search (at most 1 if directory and bucket fits in memory)
- Instead of maintaining data entries, we maintain pointers to data entries in buckets
- Instead of maintaining buckets, maintain a directory of pointers to buckets
- The directory has 2^d buckets, where d is the global depth –¿ large overhead if hashing is uniform
- Each director entry diffets by a unique d-bit adddress
- Two directories are corresponding iff their addresses differ only in the dth bit
- All entries with the same local depth (l) have the same last l bits in h(k)



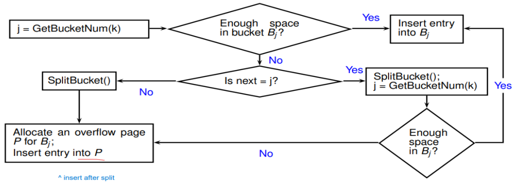
Extendible Hashing - Split, Double

- Split and doubling is checked every time a bucket is full
- Doubling only happens if local depth = global depth
- The split image has the same depth as the split bucket
- Other than the split image of the split bucket, split image of other buckets points to the same corresponding bucket
- Each bucket is pointed by $2^(d-l)$ directories

Extendible Hashing - Deletion

- B_i is deallocated
- l decrement by 1
- Directory Entries that point to B_i points to its corresponding bucket

Dynamic Hashing - Linear Hashing



^ insert after split

- **GetBucketNum(k)** returns bucket # where entry with search key k is located

$$GetBucketNum(k) = \begin{cases} h_{level}(k) & \text{if } h_{level}(k) \geq next, \\ h_{level+1}(k) & \text{otherwise.} \end{cases}$$

search key in unsplit

search key in split

- **SplitBucket()** splits bucket B_{next}
  1. Redistribute the entries in B_{next} into $B_{next+N_{level}}$ using $h_{level+1}()$
  2. next = next + 1
  3. if ($next = N_{level}$) then { level = level + 1; next = 0 }

- One I/O for equality search (more per number of overflow pages in bucket)
- Performs worse than extendible hashing if distribution is skewed
- Does not require a directory
- Higher average space utilisation, but longer overflow chains
- Has a family of hash functions, with each having a range twice of its predecessor
- N_0: initial number of buckets
- $N_i = 2^i N_0$: number of buckets at start of round i
- $next$: the next bucket to be split, this is incremented every time split happnes
- $h_i = h(k) mod N_i$: hash function for round i, if the bucket $\leq$ next (already split)
- $h_{i+1} = h(k) mod N_{i+1}$: hash function for round i+1, if the bucket $>$ next
- Split Citeria: By default, split when a bucket overflows

Linear Hashing - Deletion

- Essentially the inverse of insertion
- If the last bucket is empty –¿ delete it and decrement $next$ by 1
- If $next$ is 0, set it to $M/2 - 1$, and we can decrement level by 1 (half of buckets have been deleted if $next$ is 0)
- Merging with corresponding bucket is optional