UNIT-V

Storage and Indexing

Overview of Storage & Indexing: Data on External Storage, file organization and Indexing, Index Data structures, comparison of file organizations. Trees - structured Indexing: Intuition for tree Indexes, Indexed sequential Access method (ISAM), B+ Trees: A Dynamic Index structure, Search, Insert, Delete.

Data on External storage:

→ Disks: can retrieve random page at fixed cost

- But reading several consecutive pages is much cheaper than reading them in random order.

→ Tapes: can only read pages in sequence.

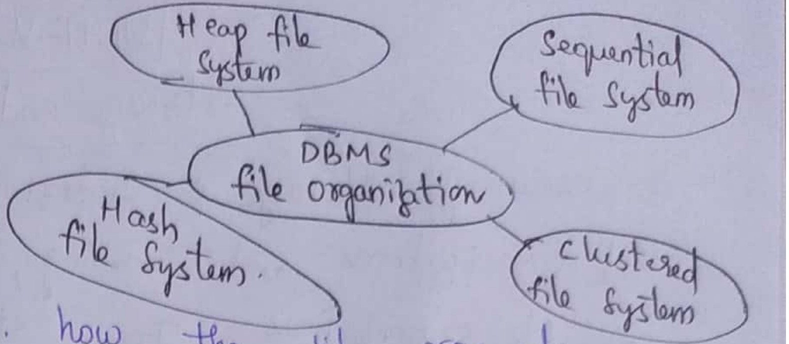
- cheaper than disks; used for archival storage.

→ file organization: Method of arranging a file of records on external storage.

- Record id (rid) is sufficient to physically locate record.
- Indexes are data structures that allow us to find the record ids of records with given values in index search key fields.

→ Architecture: Buffer manager stages pages from external storage to main memory buffer pool. file and index layers make calls to the buffer manager.

file Organization



The method of mapping file records to disk blocks defines file organization; i.e. how the file records are organized.

Heap file organization: when a file is created using Heap file organization mechanism, the OS allocates memory area to that file without any futher accounting details. file records can be placed anywhere in that memory area.

Sequential file system: Every file record contains a data field to uniquely identify that record. In sequential file organization mechanism, records are placed in the file in the Some sequential order based on the unique key field or Search key.

Hash file organization: This mechanism uses a Hash function computation on some field of the records. file is a Collection of records, which has to be mapped on some block of the disk space allocated to it.

clustered file organization: Is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, i.e, the ordering of records is not based on primary key or search key.

file structure types:

- Heap (random order) files — Suitable when typical access is a file scan retrieving all records.
- Sorted files — Best if records must be retrieved in some order, or only a 'range' of records is needed.
- Indexes = data structures to organize records i.e trees or hashing — like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields. updates are mush faster than in sorted files.

Index data structures

→ An index on a file, speeds up selections on the search key fields for the index.

→ An index contains a collection of data entries, and supports efficient retrieval of all data entries k* with a given key value k.

Data Entry k* (Index)

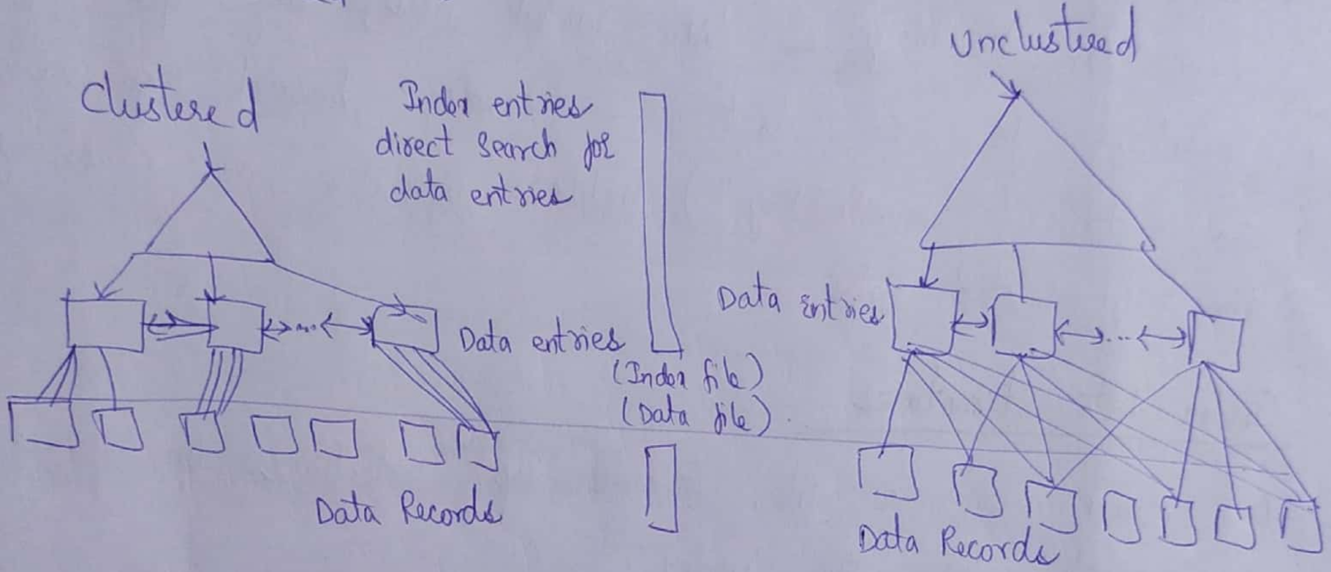
- Three options depending on what level

1) Data record with key value k (actual tuple in the table)
2) <k, rid of a data record with search key value k>
3) <k, list of rids of data records with a search key k>

i.e rid means record id.

Index classification

- primary vs Secondary: If search key contains primary key, then index called primary index.
  - unique index: Search key contains a candidate key.
- clustered vs Unclustered: If order of data records is the same as, or 'close to', order of data entries, then called clustered index.



Comparison of file organizations

Operations to compare

→ Scan : fetch all records from disk

→ Equality search.

→ Range selection

→ Insert a record

→ Delete a record.

- Heap files (random order, insert at end of file)
- Sorted files, Sorted on <age, sal>
- clustered B+ tree file, Search key <age, sal>
- Heap file with unclustered B+ tree index on search key <age, sal>
- Heap file with unclustered hash index on search key <age, sal>

Cost of operations

	Scan	Equality	Range	Insert	Delete
1. Heap	BD	0.5 BD	BD	2D	Search + D
2. Sorted	BD	$D \log_2 B$	$D \log_2 B$ + # matches	Search + BD	Search + BD
3. Clustered	1.5 BD	$D \log_f 1.5B$	$D \log_f 1.5B$ + # matches	Search + D	Search + D
4. Unclustered Tree index	BD(R+0.15)	$D(1 + \log_f 0.15B)$	$D \log_f 0.15B$ + # matches	$D(3 + \log_f 0.15B)$	Search + 2D
5. Unclustered ~~Hash~~ Hash Index	BD(R+0.125)	2D	BD	4D	Search + 2D.

Tree-structured Indexing

→ As for any index, 3 alternatives for data entries k* :

1) Index refers to actual data record with key value k.
2) Index refers to list of <k, rid> pairs.
3) Index refers to list of <k, [rid list]>

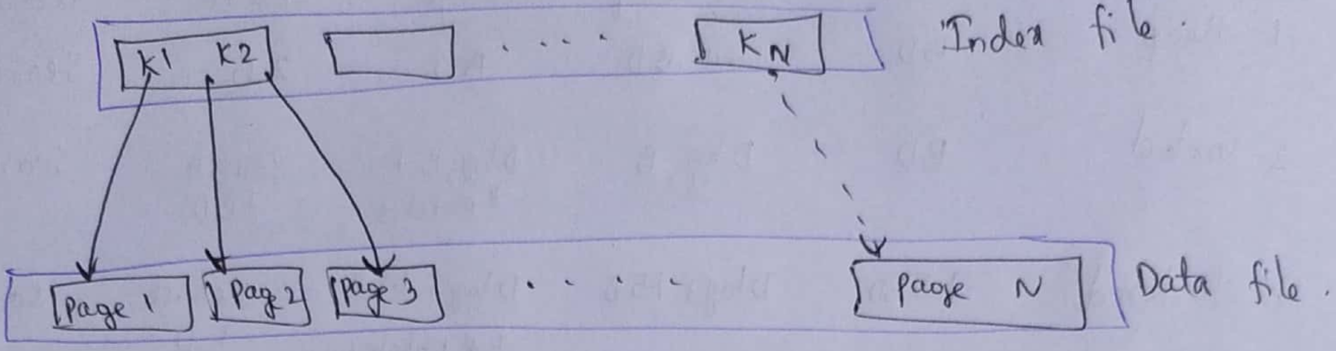
→ Tree-structured Indexing techniques support both range searches and equality searches.

→ ISAM: static structure; B+ tree : dynamic structure.

for eg: find all students with gpa > 3.0

1) If data is in sorted file, do binary search to find first such student, then scan to find others.
2) Cost of binary search can be quite high.

→ Create an 'index' file.



ISAM (Indexed Sequential Access Method):

* The index is static:

- once the separator levels have been constructed, they never change.
- Number and position of leaf pages in file stays fixed.

* Good for equality and range searches.

- leaf pages stored sequentially in file when storage structure is created to support range searches.

* supports multiple attribute search keys and partial key searches

- Contents of leaf pages can change.
- Row deletion yields empty spot in leaf page.
- Row insertion can result in overflow leaf page and ultimately overflow chain.
- chain might be long, unsorted, scattered on disk.
- ISAM can be inefficient if table is dynamic.

* Generally an integrated storage structure – clustered, index entries contain rows.

Contd:

* Separator entry = (k_i, P_i) where k_i is a Search key value, P_i is a pointer to a lower level page.

* K_i separates set of search key values in the two Sub-trees pointed by P_{i-1} and P_i.

Index file creation

1) primary pages are allocated sequentialy

- If alternative ①: all the data reside in the leaf pages, sorted by the search key value.
- If alternative ②, ③: the data records are stored in a separate file; sorted before allocating the leaf pages.

2) Index entry pages are then allocated.

3) Then space for overflow pages

- overflow pages are need if more entries inserted into a leaf cannot fit into a single page.

4) Equality search - start at root; use key comparisons to go to leaf.

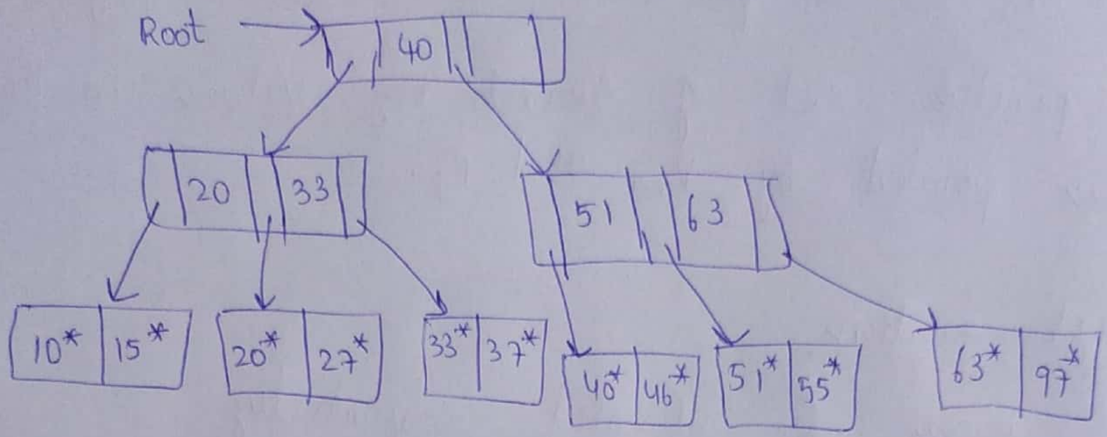
5) Range search - Determine the starting point in the leaf level by equality search.
   - Retrieve primary pages sequentially and overflow pages as needed by pointers from primary pages.

6) Insert - find the leaf page the entry belong to, and put it there; add overflow page if needed.

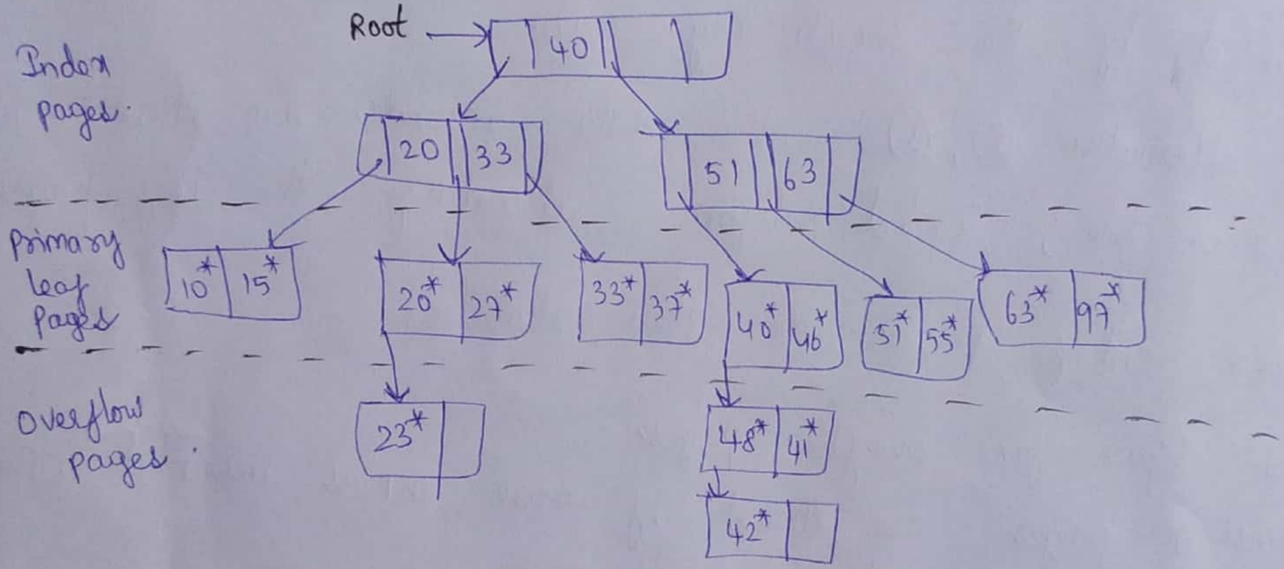
7) Delete - find & remove the entry from leaf page; if empty overflow page, de-allocate.

Example ISAM tree

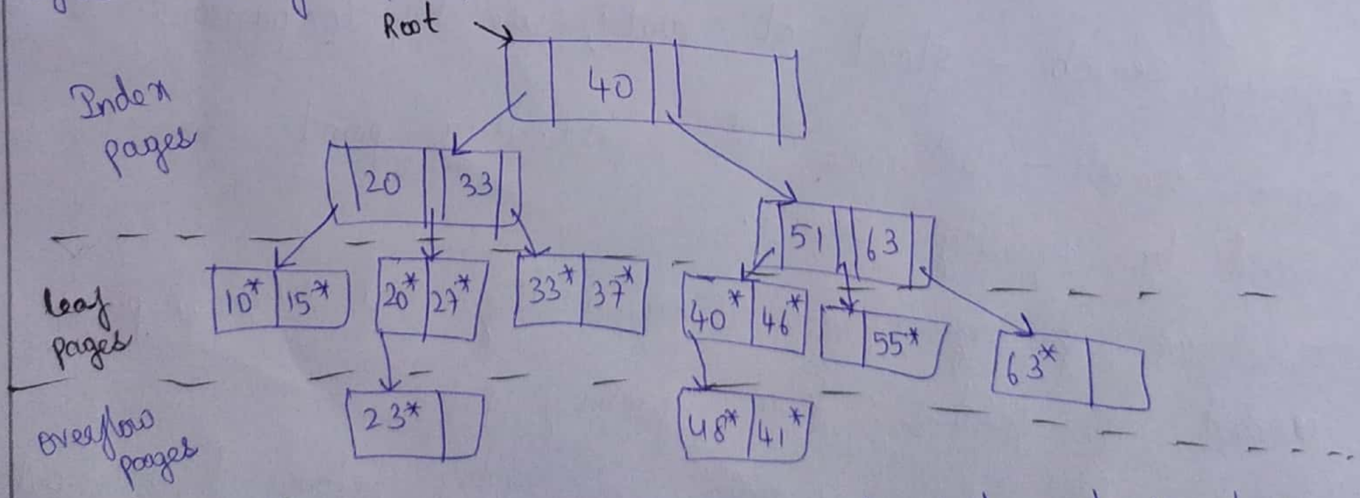
- Each node can hold 2 entries



- After Inserting 23*, 48*, 41*, 42* ...



- After Deleting 42*, 51*, 97*



i.e. 51* appears in index level, but not in leaf.

Contd ISAM

Advantage: Not locking index entry pages in case of concurrent transactions.

Disadvantage: possibly long overflow chains, which are usually not sorted.

Note: ISAM might be preferable to B+ tree if overflow chains are rare.

B+ Trees: A Dynamic Index structure.

→ B+ Tree is multi-level Index format, which is balanced binary search trees. As mentioned earlier single level index records becomes large as the db size grows, which also degrades performance. All leaf nodes of B+ tree denote actual data pointers.

→ B+ tree ensures that all leaf nodes remain at the same height, thus balanced.

→ Additionally, all leaf nodes are linked using link list, which makes B+ tree to support random access as well as sequential access.

→ Every leaf node is at equal distance from the root node. A B+ tree is of order n where n is fixed for every B+ tree.

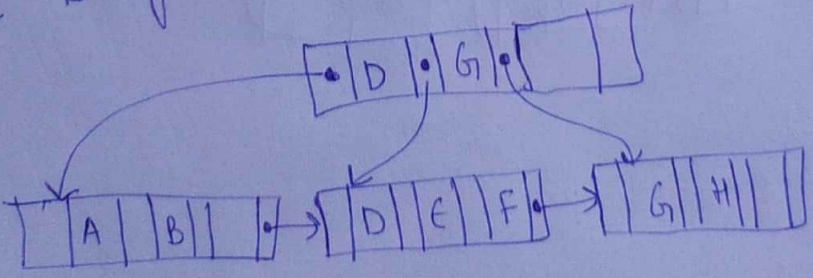


fig. structure of B+ tree

Internal nodes:

→ Internal (non-leaf) nodes contains atleast $\lceil n/2 \rceil$ pointers, except the root node.

→ At most, internal nodes contain n pointers.

leaf nodes:

→ leaf nodes contain atleast $\lceil n/2 \rceil$ record pointers and $\lceil n/2 \rceil$ key values.

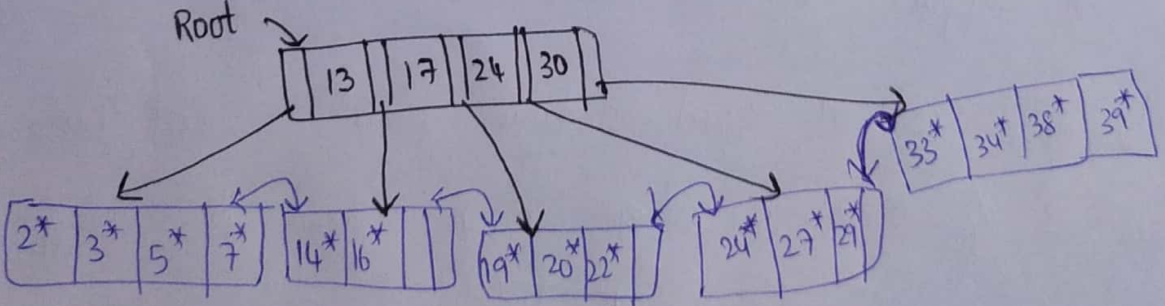
→ At most, leaf nodes contain n record pointers & n key values.

→ Every leaf node contains one block pointer p ~~leaf~~ to point to next leaf node and forms a linked list.

Example: B+ tree, order d = 2.

(1) Search:

- Begins at root, and key comparisons direct it to a leaf
- search for 5*, 15*, all data entries ≥ 24* ...



Contd: **B^+ Trees Insertion**

→ B^+ tree are filled from bottom, and each node is inserted at leaf node.

→ If leaf node overflows:

① split node into 2 parts.

② partition at $i = [(m+1)/2]$

③ ↳ first i entries are stored in one node.

④ ↳ Rest of the entries (i+1 onwards) are moved to a new node
ith key is duplicated in the parent of the leaf.

→ If non-leaf node overflows:

① split node into 2 parts.

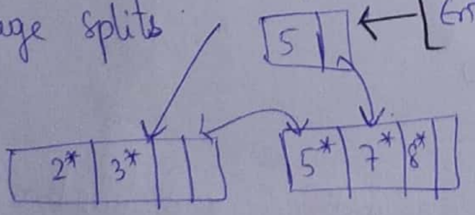
② partition ~~at~~ the node at $i = [(m+1)/2]$

③ Entries upto i are kept in one node.

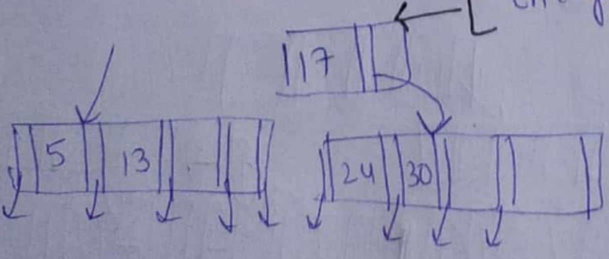
④ Rest of the entries are moved to a new node.

for example: Insert 8*.

minimum occupancy is guaranteed in both leaf and index page splits.

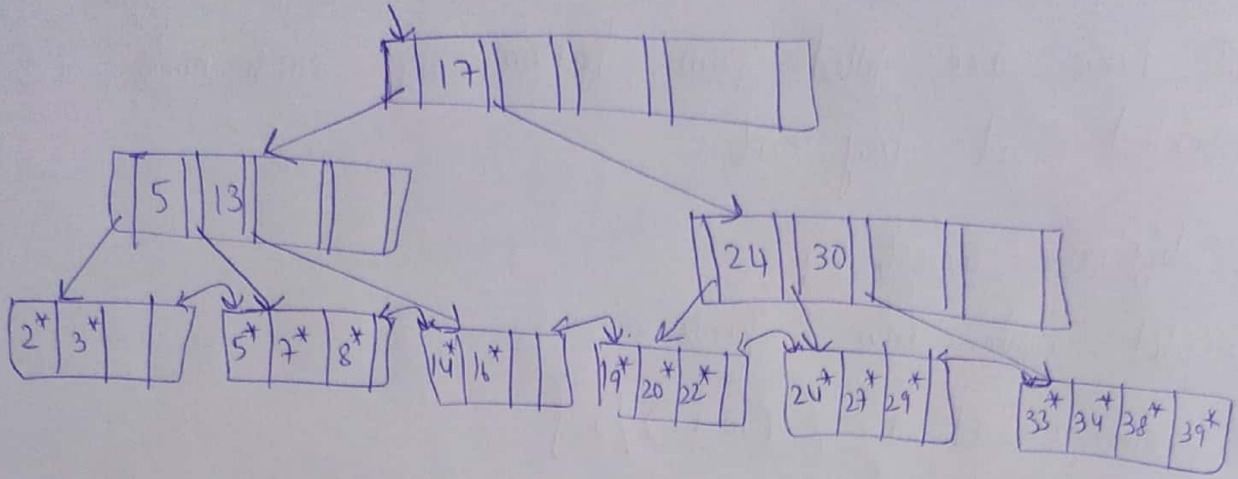


[Entry to be inserted in parent node]



[Entry to be inserted in parent node So 17 is pushed up and appears once in the index).

After Inserting 8*, the B+ Tree is:



B+ Tree Deletion

→ B+ tree entries are deleted leaf nodes.

→ The target entry is searched & deleted.

- If it is in internal node, delete and replace with the entry from the left position.

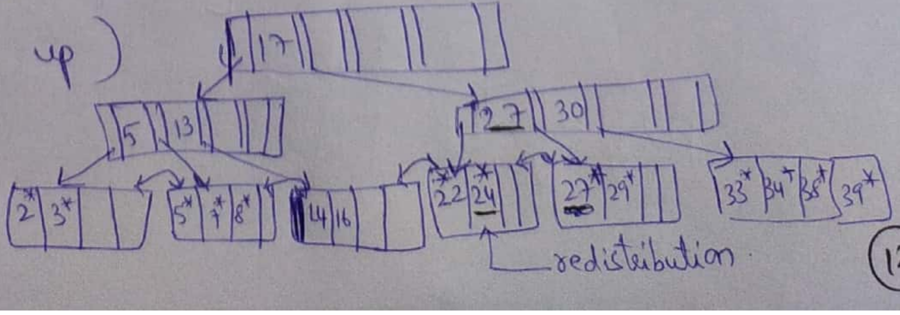
→ After deletion underflow is tested.

- If underflow occurs – Distribute entries from nodes left to it.
- If distribution from left is not possible – Distribute from nodes right to it.
- If distribution from left and right is not possible – Merge the node with left and right to it.

for Example: Delete 19* and 20*.

→ Deleting 19* is easy.

→ But deleting 20* is done with redistribution (middle key 27* is copied up)



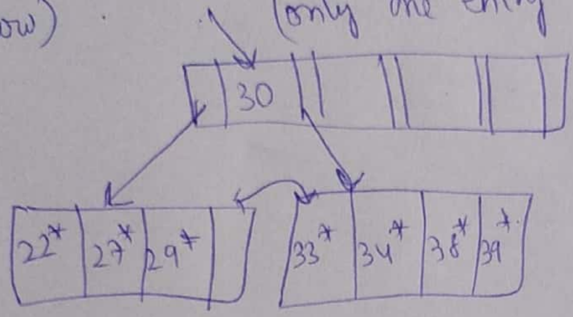
for example; If delete 24^* from B^+ Tree then.

To delete 24^*.

→ must merge.

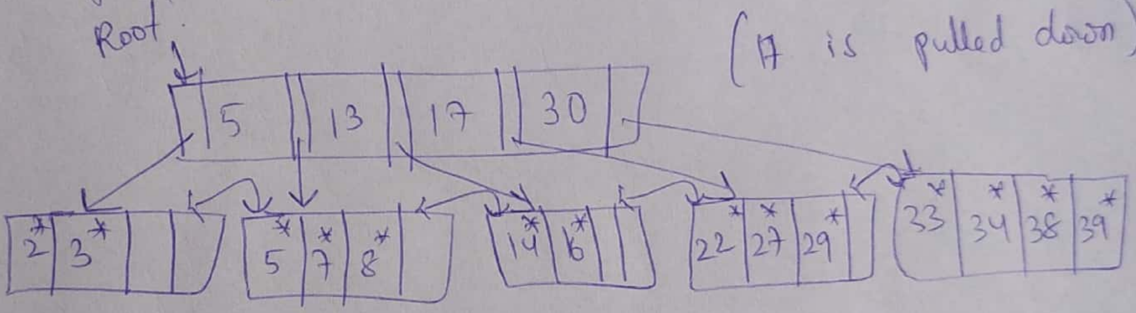
→ "toss" of index entry (right), and "pull down" of index entry (below).

(only one entry left).



So, at final B^+ Tree look a like as.

(17 is pulled down)



→ The End ←