FILE ORGANIZATION AND INDEXING DATA STRUCTURES

Contents at a glance:

1. file organizations , comparison of file organizations, RAID
2. indexing, index data structures,
3. Tree structured indexing -intuition for tree indexes,
4. indexed sequential access method (ISAM),
5. B+ Trees -a dynamic tree structure.

1. File Organizations:

Storing the files in certain order is called file organization. The main objective of file organization is

- Optimal selection of records i.e.; records should be accessed as fast as possible.
- Any insert, update or delete transaction on records should be easy, quick and should not harm other records.
- No duplicate records should be induced as a result of insert, update or delete
- Records should be stored efficiently so that cost of storage is minimal.

Some of the file organizations are

1. Sequential File Organization
2. Heap File Organization
3. Hash/Direct File Organization
4. Indexed Sequential Access Method
5. B+ Tree File Organization
6. Cluster File Organization

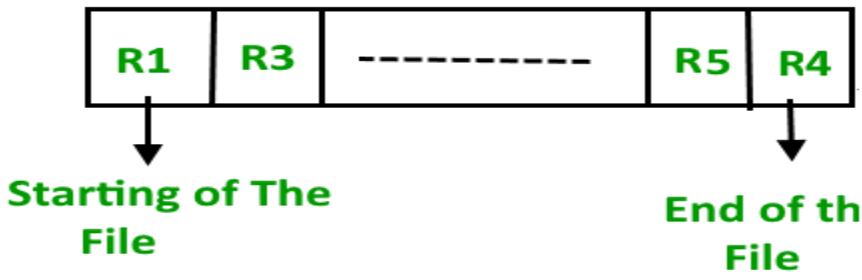
1. Sequential File Organization:

In sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key.

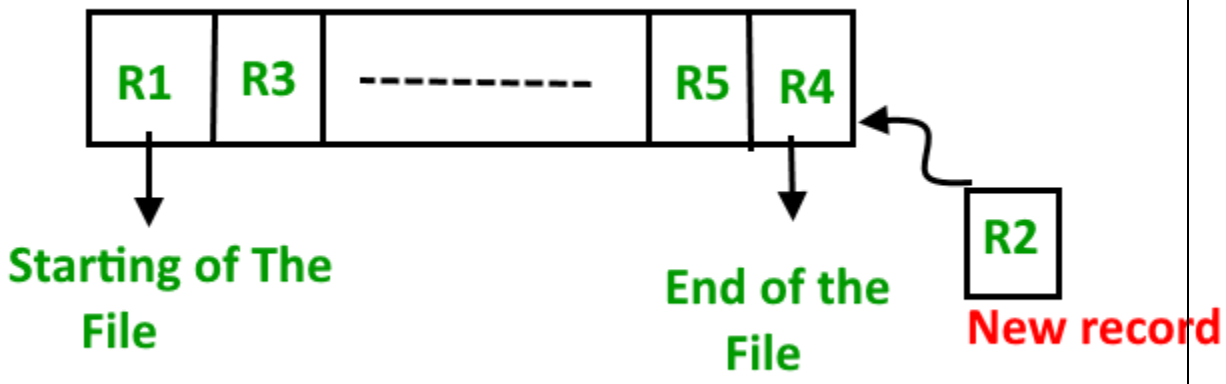
The easiest method for file Organization is Sequential method. In this method the the file are stored one after another in a sequential manner. There are two ways to implement this method:

1. Pile FIle Method
2. Sorted File

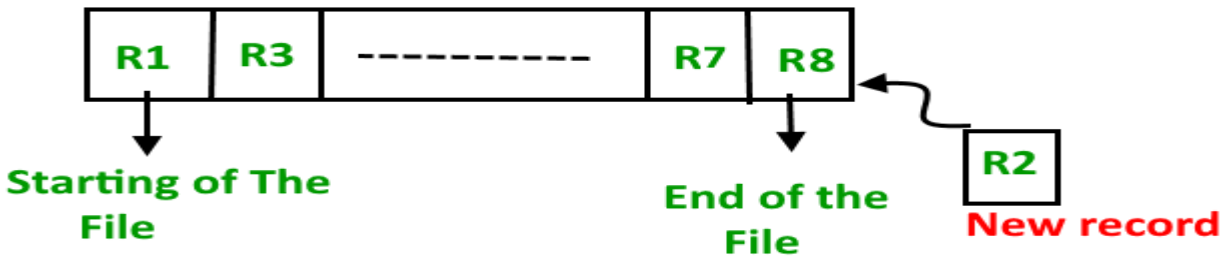
1. **Pile File Method –** This method is quite simple, in which we store the records in a sequence i.e one after other in the order in which they are inserted into the tables.



Insertion of new record –
Let the R1, R3 and so on upto R5 and R4 be four records in the sequence. Here, records are nothing but a row in any table. Suppose a new record R2 has to be inserted in the sequence, then it is simply placed at the end of the file.

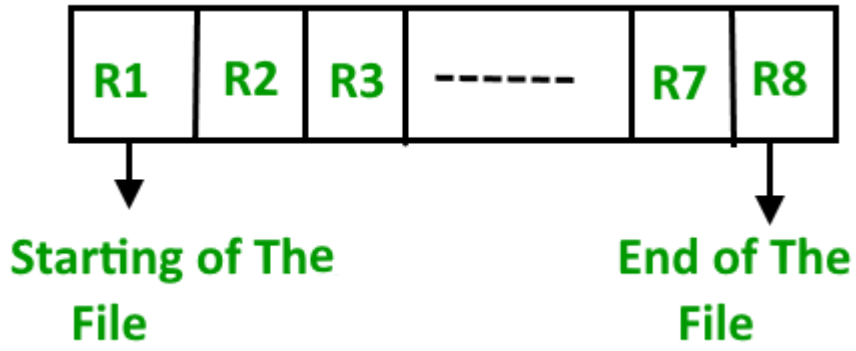


2. **Sorted File Method –**In this method, As the name itself suggest whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. Sorting of records may be based on any primary key or any other key.



Insertion of new record –
Let us assume that there is a preexisting sorted sequence of four records R1, R3, and so

on upto R7 and R8. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence .



Pros and Cons of Sequential File Organization –
Pros –

- Fast and efficient method for huge amount of data.
- Simple design.
- Files can be easily stored in magnetic tapes i.e cheaper storage mechanism.

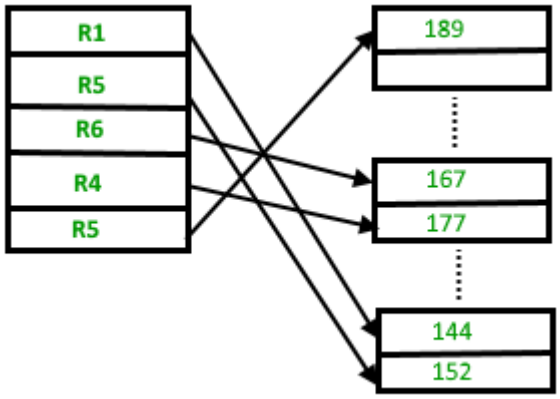
Cons –

- Time wastage as we cannot jump on a particular record that is required, but we have to move in a sequential manner which takes our time.
- Sorted file method is inefficient as it takes time and space for sorting records.

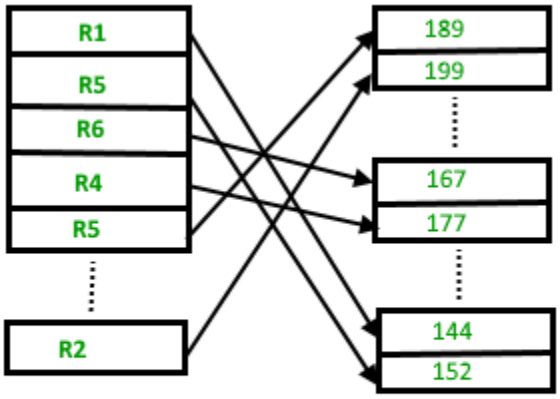
2. ***Heap File Organization:***

When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area.

Heap File Organization works with data blocks. In this method records are inserted at the end of the file, into the data blocks. No Sorting or Ordering is required in this method. If a data block is full, the new record is stored in some other block, Here the other data block need not be the very next data block, but it can be any block in the memory. It is the responsibility of DBMS to store and manage the new records.


R1
R5
R6
R4
R5
189
167
177
144
152

Insertion of new record –
Suppose we have four records in the heap R1, R5, R6, R4 and R3 and suppose a new record R2 has to be inserted in the heap then, since the last data block i.e data block 3 is full it will be inserted in any of the database selected by the DBMS, lets say data block 1.



If we want to search, delete or update data in heap file Organization the we will traverse the data from the beginning of the file till we get the requested record. Thus if the database is very huge, searching, deleting or updating the record will take a lot of time.

Pros and Cons of Heap File Organization –
Pros –

- Fetching and retrieving records is faster than sequential record but only in case of small databases.
- When there is a huge number of data needs to be loaded into the database at a time, then this method of file Organization is best suited.

Cons –

- Problem of unused memory blocks.
- Inefficient for larger databases.

3. Hash File Organization:

Hash File Organization uses Hash function computation on some fields of the records. The output of the hash function determines the location of disk block where the records are to be placed.

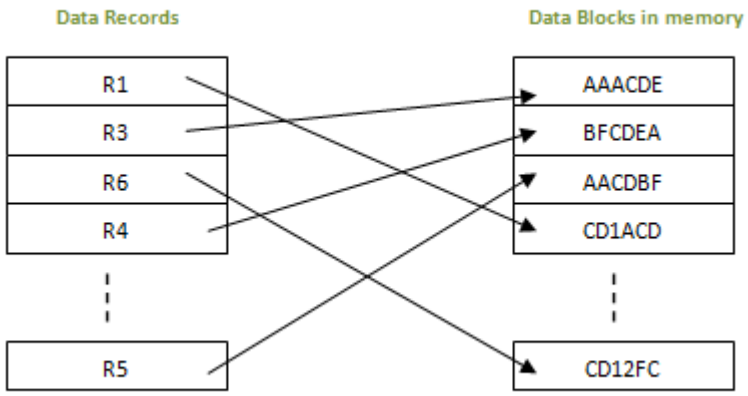
In this method of file organization, hash function is used to calculate the address of the block to store the records.

The hash function can be any simple or complex mathematical function.

The hash function is applied on some columns/attributes – either key or non-key columns to get the block address.

Hence each record is stored randomly irrespective of the order they come. Hence this method is also known as Direct or Random file organization.

If the hash function is generated on key column, then that column is called hash key, and if hash function is generated on non-key column, then the column is hash column.



When a record has to be retrieved, based on the hash key column, the address is generated and directly from that address whole record is retrieved. Here no effort to traverse through whole file. Similarly when a new record has to be inserted, the address is generated by hash key and record is directly inserted. Same is the case with update and delete.

Advantages of Hash File Organization

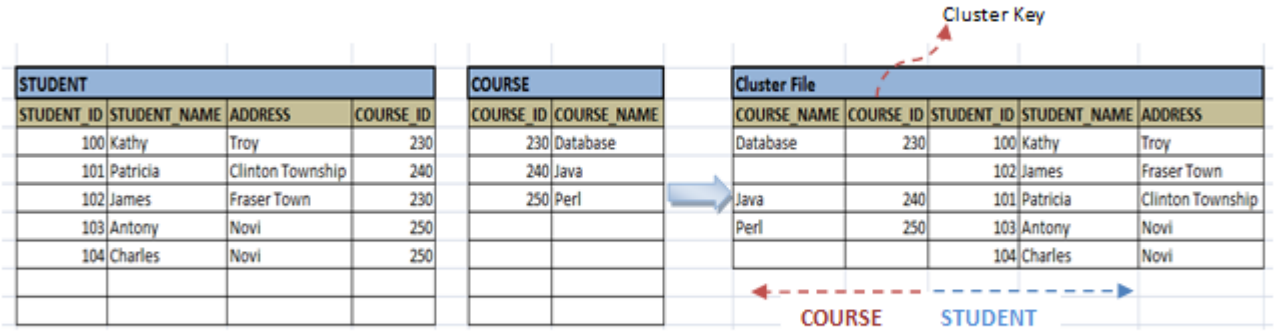
- Records need not be sorted after any of the transaction. Hence the effort of sorting is reduced in this method.
- Since block address is known by hash function, accessing any record is very faster. Similarly updating or deleting a record is also very quick.

- This method can handle multiple transactions as each record is independent of other. i.e.; since there is no dependency on storage location for each record, multiple records can be accessed at the same time.
- It is suitable for online transaction systems like online banking, ticket booking system etc.

clustered file organization:

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

In this method two or more table which are frequently used to join and get the results are stored in the same file called clusters. These files will have two or more tables in the same data block and the key columns which map these tables are stored only once. This method hence reduces the cost of searching for various records in different files. All the records are found at one place and hence making search efficient.



	Sequential	Heap/Direct	Hash	ISAM	B+ tree	Cluster
Method of storing	Stored as they come or sorted as they come	Stored at the end of the file. But the address in the memory is random.	Stored at the hash address generated	Address index is appended to the record	Stored in a tree like structure	Frequently joined tables are clubbed into one file based on cluster key
Types	Pile file and sorted file Method		Static and dynamic hashing	Dense, Sparse, multilevel indexing		Indexed and Hash
Design	Simple Design	Simplest	Medium	Complex	Complex	Simple
Storage Cost	Cheap (magnetic tapes)	Cheap	Medium	Costlier	Costlier	Medium
Advantage	Fast and efficient when there is large volumes of data, Report generation, statistical calculations etc	Best suited for bulk insertion, and small files/tables	Faster Access No Need to Sort Handles multiple transactions Suitable for Online transactions	Searching records is faster. Suitable for large database. Any of the columns can be used as key column. Searching range of data & partial data are efficient.	Searching range of data & partial data are efficient. No performance degrades when there is insert / delete / update. Grows and shrinks with data. Works well in secondary storage devices and hence reducing disk I/O. Since all datas are at the leaf node, searching is easy. All data at leaf node are sorted sequential linked list.	Best suited for frequently joined tables. Suitable for 1:M mappings
Disadvantage	Sorting of data each time for insert/delete/ update takes time and makes system slow.	Records are scattered in the memory and they are inefficiently used. Hence increases the memory size. Proper memory management is needed. Not suitable for large tables.	Accidental Deletion or updation of Data Use of Memory is inefficient Searching range of data, partial data, non-hash key column, searching single hash column when multiple hash keys present or frequently updated column as hash key are inefficient.	Extra cost to maintain index. File reconstruction is needed as insert/update/delete. Does not grow with data.	Not suitable for static tables	Not suitable for large database. Suitable only for the joins on which clustering is done. Less frequently used joins and 1: 1 Mapping are inefficient.

Comparison of file organizations:

The operations to be considered for comparisons of file organizations are below:

- **Scan:** Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page (in the pool).
- **Search with equality selection:** Fetch all records that satisfy an equality selection, for example, "Find the Students record for the student with *sid* 23." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.
- **Search with range selection:** Fetch all records that satisfy a range selection, for example, "Find all Students records with *name* alphabetically after 'Smith.' "
- **Insert:** Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.
- **Delete:** Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap	BD	$0.5BD$	BD	$2D$	$Search + D$
Sorted	BD	$Dlog_2B$	$Dlog_2B$+# *matches*	$Search + BD$	$Search + BD$
Hashed	$1.25BD$	D	$1.25BD$	$2D$	$Search + D$

Figure 8.1 A Comparison of I/O Costs

B - number of data pages

R records per page

D- average time to read or write a disk page

C- average time to process a record

REDUNDANT ARRAY OF INDEPENDENT DISKS(RAID)

RAID or **R**edundant **A**rray of **I**ndependent **D**isks, is a technology to connect multiple secondary storage devices and use them as a single storage media.

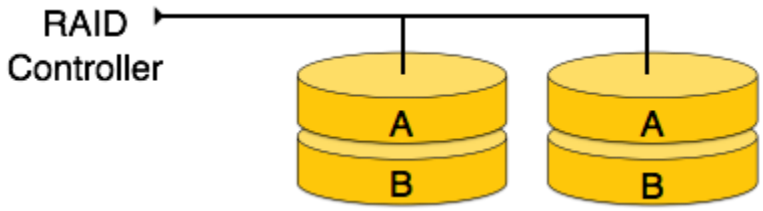
RAID consists of an array of disks in which multiple disks are connected together to achieve different goals. RAID levels define the use of disk arrays.

RAID 0

In this level, a striped array of disks is implemented. The data is broken down into blocks and the blocks are distributed among disks. Each disk receives a block of data to write/read in parallel. It enhances the speed and performance of the storage device. There is no parity and backup in Level 0.

RAID 1

RAID 1 uses mirroring techniques. When data is sent to a RAID controller, it sends a copy of data to all the disks in the array. RAID level 1 is also called **mirroring** and provides 100% redundancy in case of a failure.

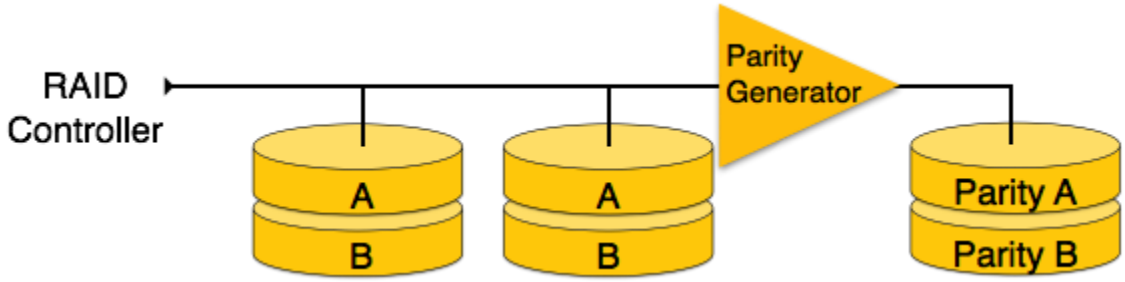


RAID 2

RAID 2 records Error Correction Code using Hamming distance for its data, striped on different disks. Like level 0, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on a different set disks. Due to its complex structure and high cost, RAID 2 is not commercially available.

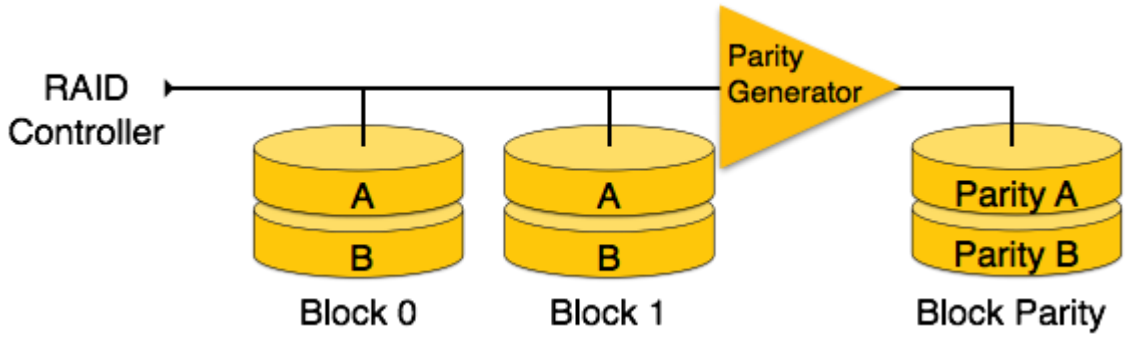
RAID 3

RAID 3 stripes the data onto multiple disks. The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failures.



RAID 4

In this level, an entire block of data is written onto data disks and then the parity is generated and stored on a different disk. Note that level 3 uses byte-level striping, whereas level 4 uses block-level striping. Both level 3 and level 4 require at least three disks to implement RAID.

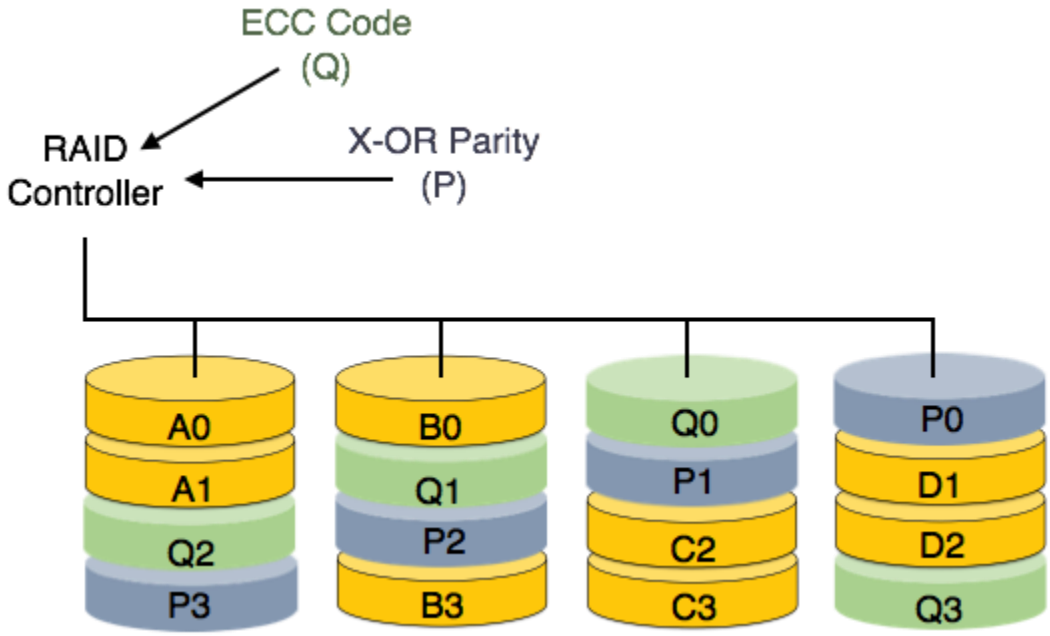


RAID 5

RAID 5 writes whole data blocks onto different disks, but the parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.

RAID 6

RAID 6 is an extension of level 5. In this level, two independent parities are generated and stored in distributed fashion among multiple disks. Two parities provide additional fault tolerance. This level requires at least four disk drives to implement RAID.



2. Indexing:

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

Indexing is defined based on its indexing attributes. Indexing can be of the following types −

- **Primary Index** − Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.
- **Secondary Index** − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- **Clustering Index** − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types −

- Dense Index
- Sparse Index

Dense Index

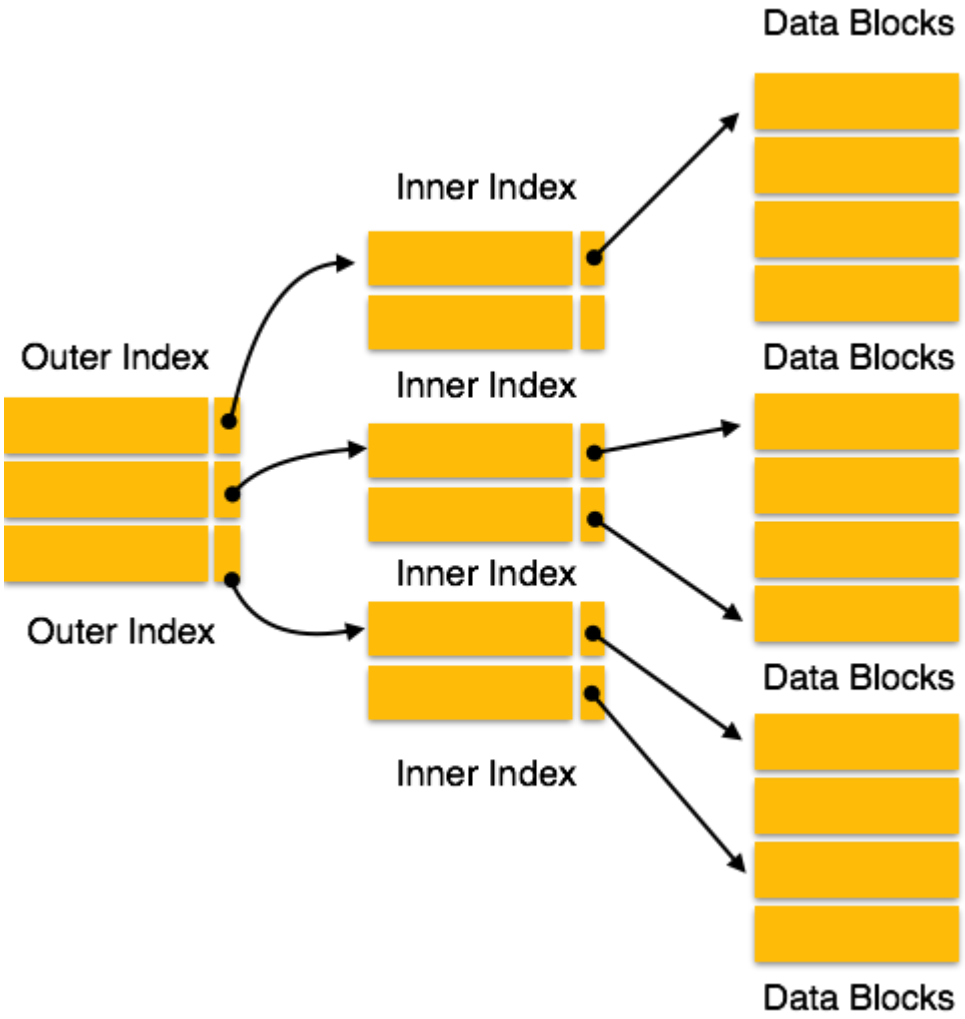
In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.

Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.

Multilevel Index

Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

3. indexing data structures

ISAM:

ISAM (Indexed Sequential Access Method) is a file management system developed at IBM that allows records to be accessed either sequentially (in the order they were entered) or randomly (with an index). Each index defines a different ordering of the records.

ISAM is a static index structure –effective when the file is not frequently updated. Not suitable for files that grow and shrink. When an ISAM file is created, index nodes are fixed, and their pointers do not change during inserts and deletes that occur later.

In ISAM, leaf pages contains data entries and non leaf pages contains index entries of the form (search key value, page id)

- **ISAM is Tree structured index**
- **ISAM Support queries**
  - **Point queries**
  - **Range queries**
- **Problems with ISAM**
  - **Static: inefficient for insertions and deletions**

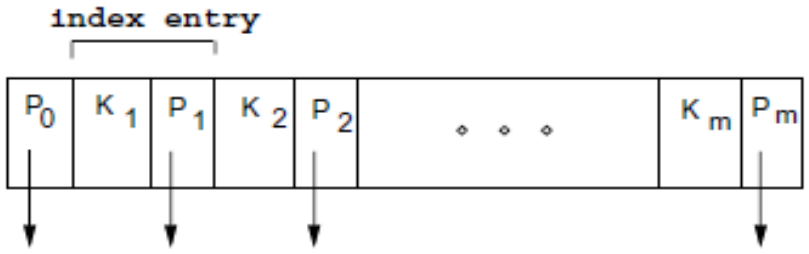


Figure: Format of index page

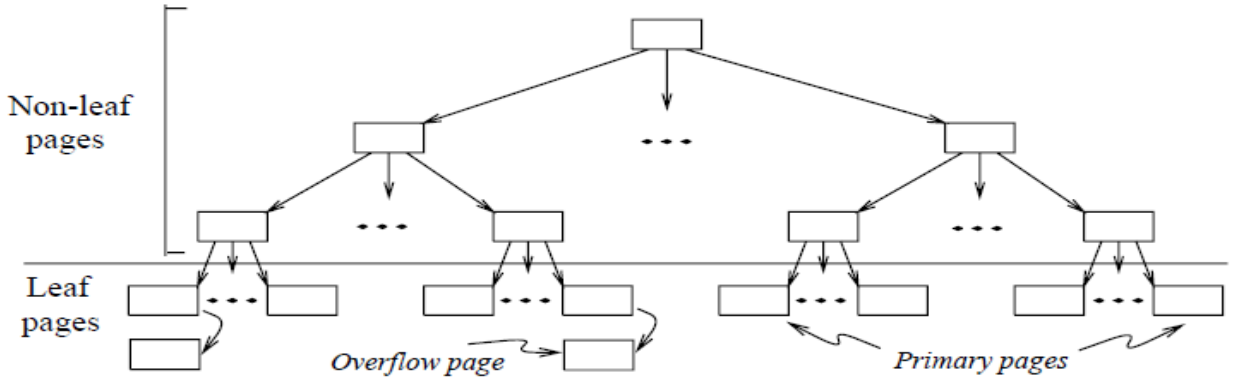
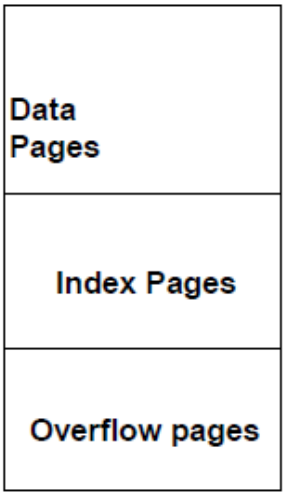


Figure: ISAM index structure

Page allocation in ISAM:



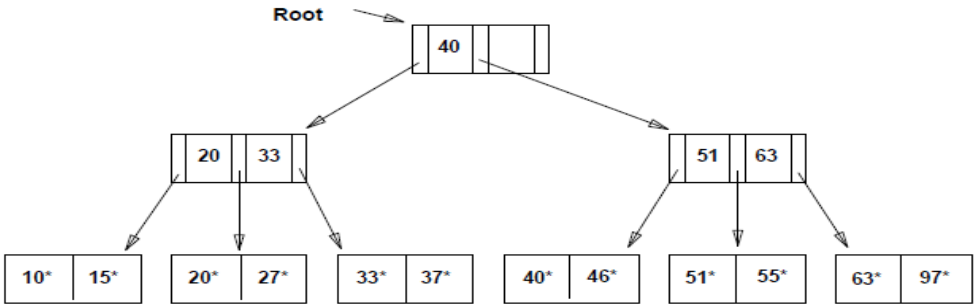
- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\log_F N$
  F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, put it there if space is available, else allocate an overflow page, put it there, and link it in.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

☛ **Static tree structure:** *inserts/deletes affect only leaf pages.*

Example for ISAM :

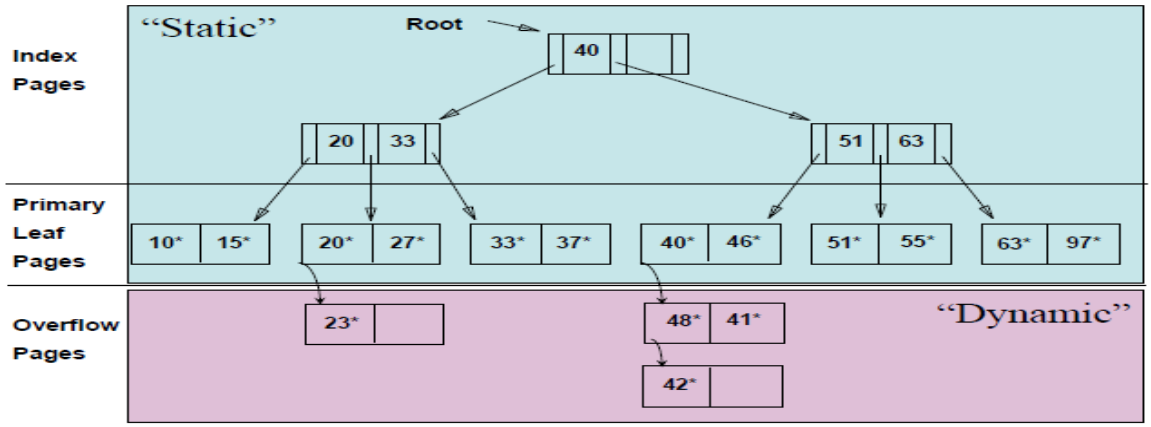
Example ISAM Tree

- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)



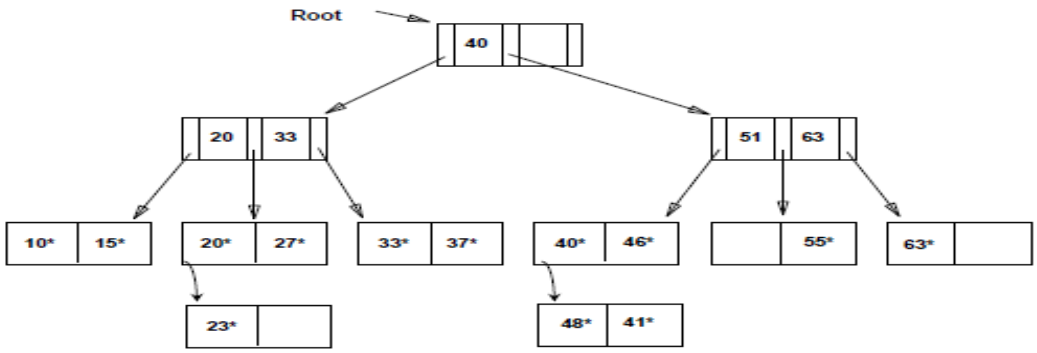
insertion:

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



Deletion:

... Then Deleting 42*, 51*, 97*



☛ *Note that 51* appears in index, but not in leaf!*

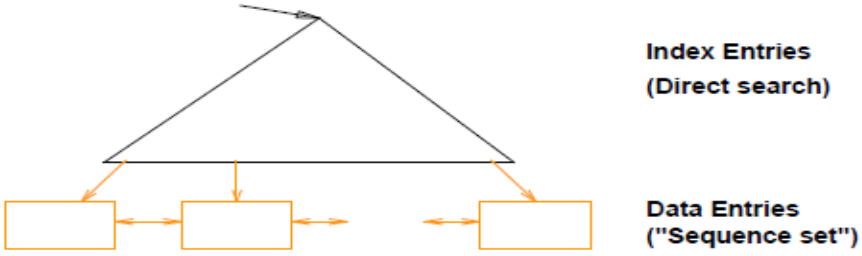
B+ Trees:

A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between [M/2] and [M] children, where n is fixed for a particular tree.

B+ tree is a variation of B trees in which

- internal nodes contain only search keys (no data)
- Leaf nodes contain pointers to data records
- Data records are in sorted order by the search key
- All leaves are at the same depth

- Insert/delete at $\log_F N$ cost; keep tree *balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy. Each internal non-root node contains **d** <= m <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.

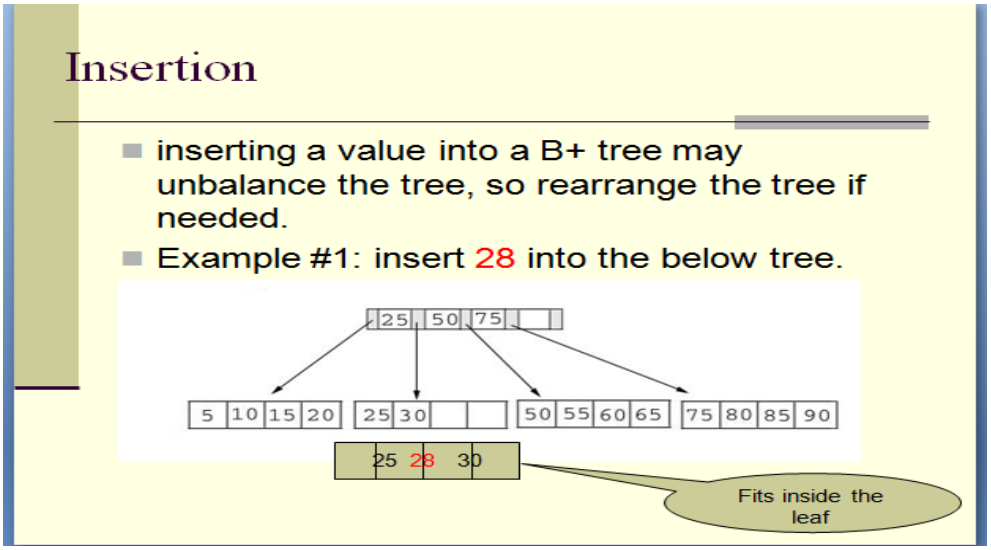


Insertion operation in B+ trees:

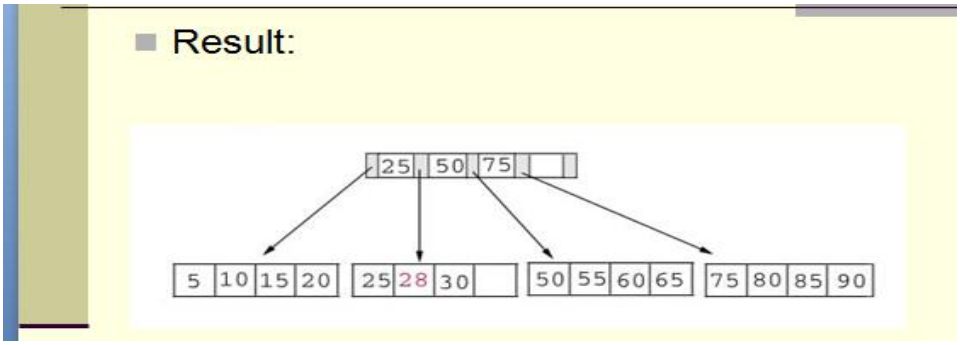
Inserting into a B+ Tree

- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Allocate new node
    - Redistribute entries evenly
    - **Copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Insertion example:

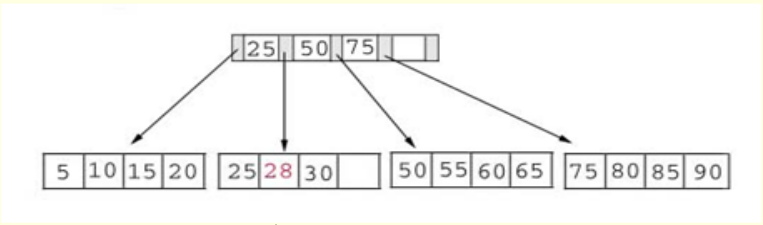


Result of above insertion:



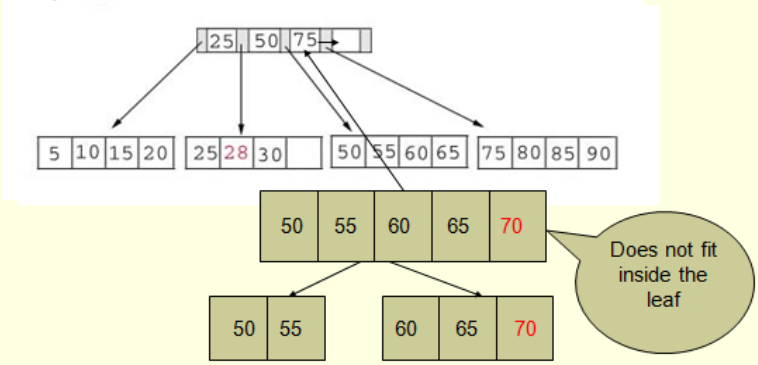
Insertion

- Example #2: insert 70 into below tree

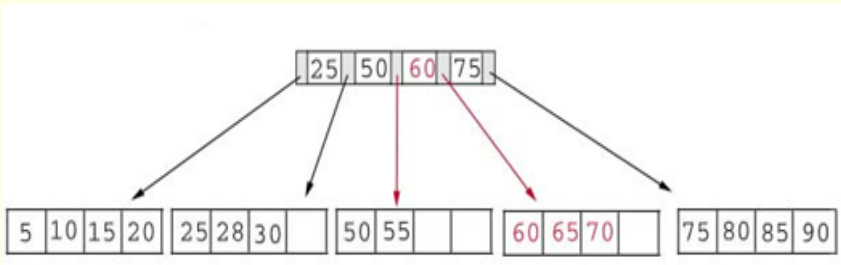


Insertion

- Process: split the leaf and propagate middle key up the tree



- Result: chose the middle key 60, and place it in the index page between 50 and 75.

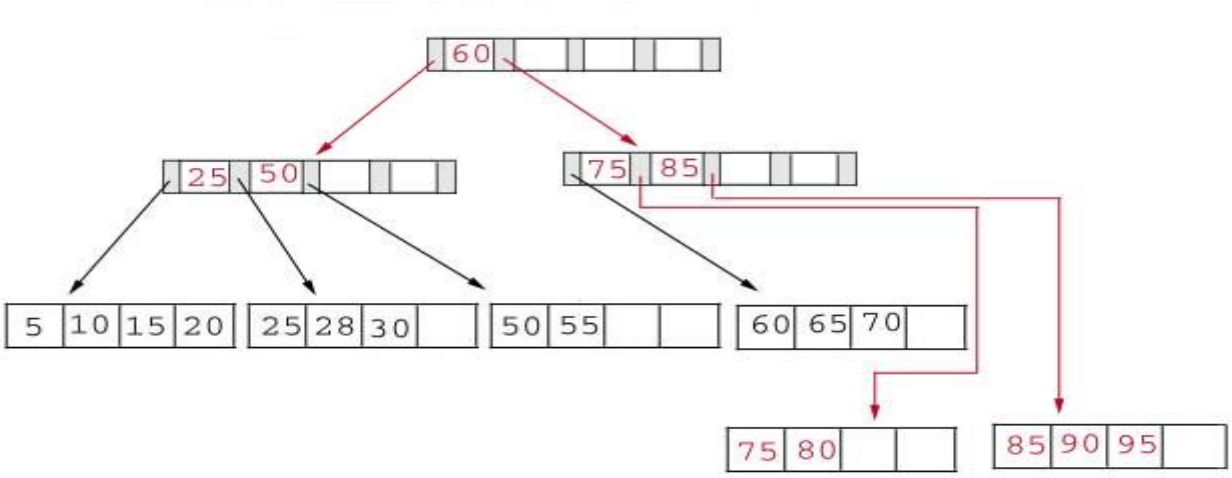


Deletion in B+ tree:

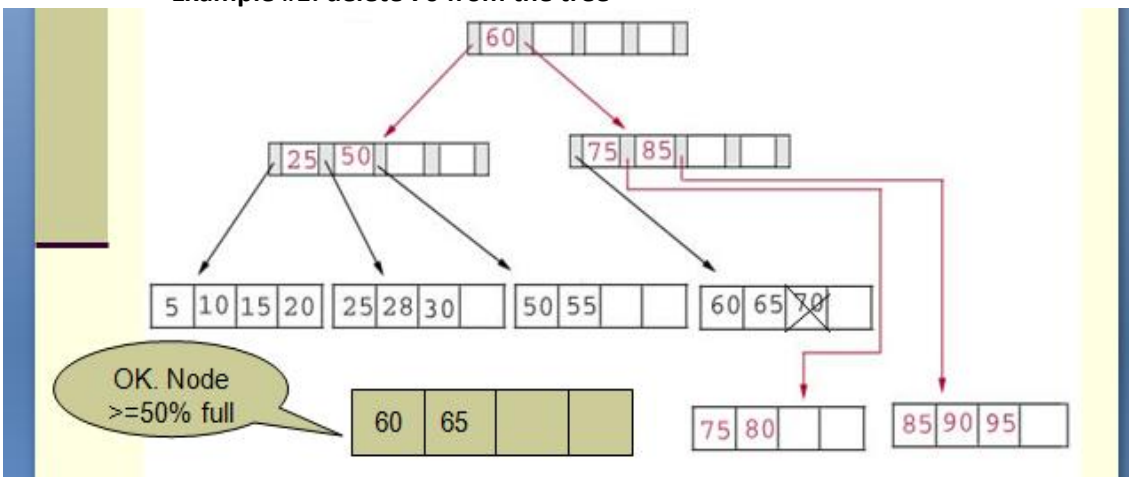
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf L with entry, if it exists.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing keys from *sibling* (*adjacent node with same parent as L*).
    - If redistribution fails, *merge* L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

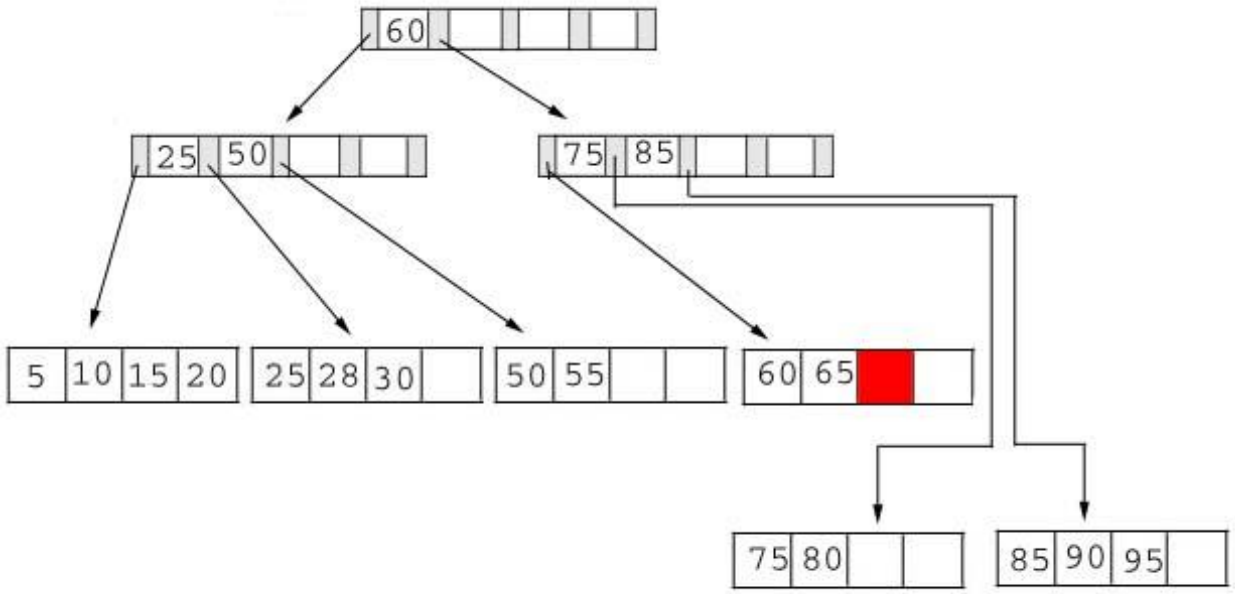
Example for B+ tree deletion:
Consider the below tree:

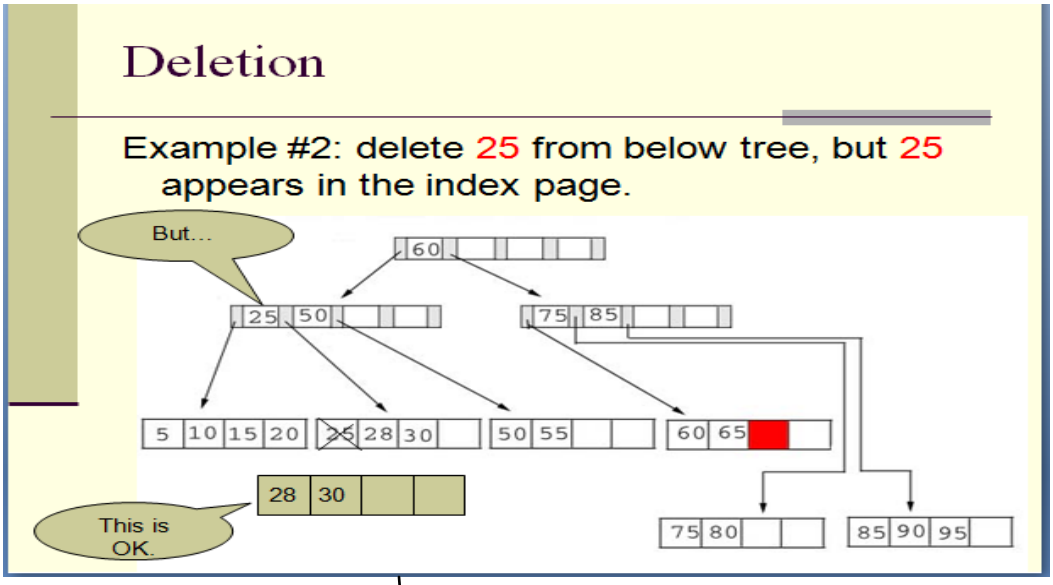


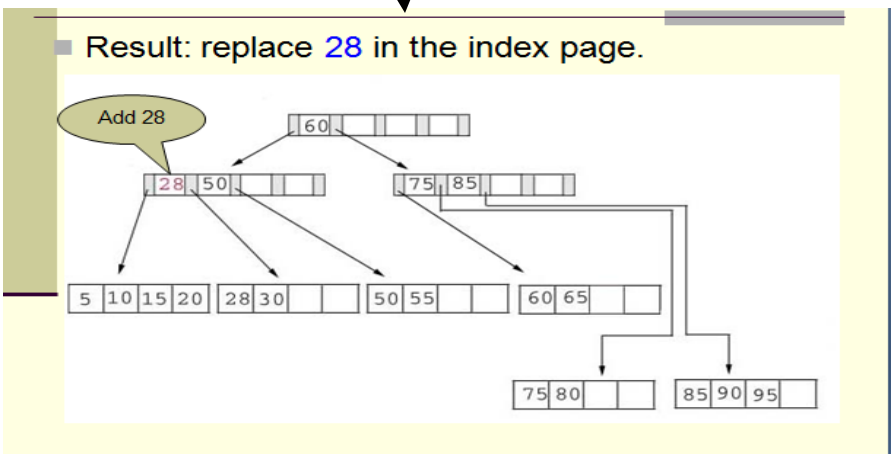
- **Same as insertion, the tree has to be rebuild if the deletion result violate the rule of B+ tree.**
- **Example #1: delete 70 from the tree**

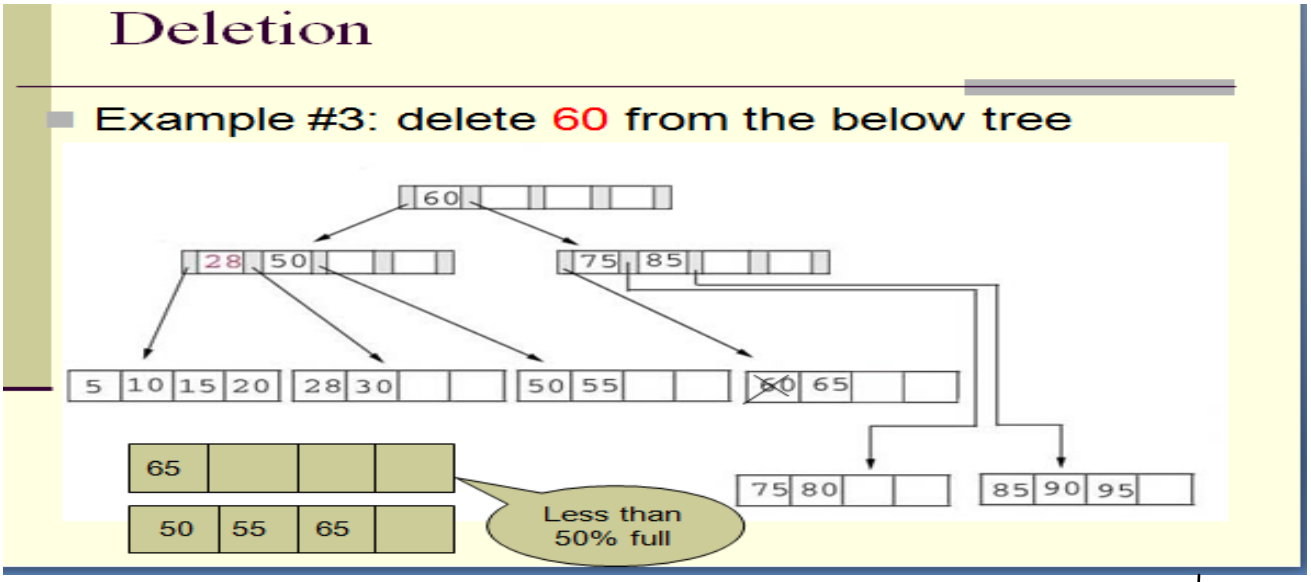


Result after deleting 70:





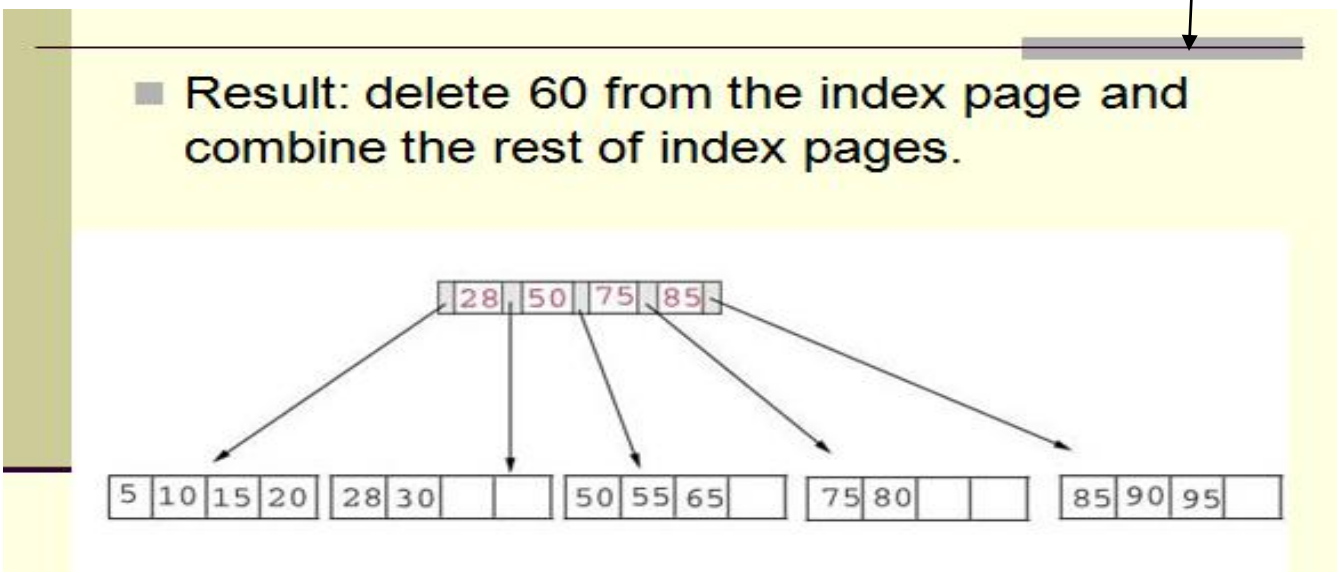




In above example:

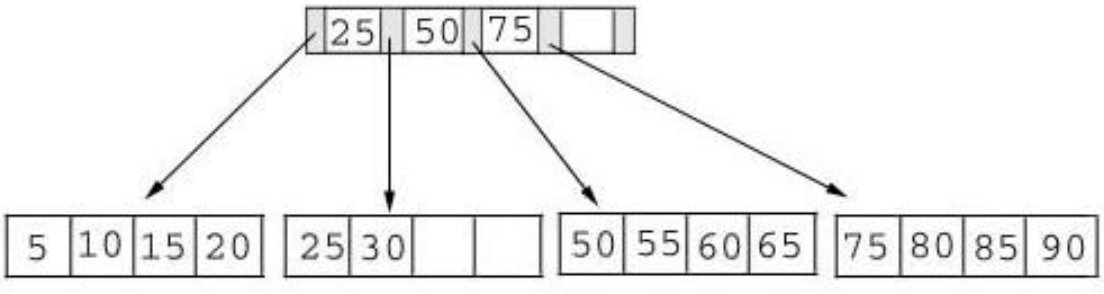
Locate 60 in leaf node Delete it. Leaf now < 50% full.

Neighbor at most 50% full, so merge with neighbor to the right. Guaranteed to fit.



Search Operation in B+ trees:

- **Just compare the key value with the data in the tree, then return the result.**
  For example: find the value 45, and 15 in below tree.



- **Result:**
  1. For the value of 45, not found.
  2. For the value of 15, return the position where the pointer located

Advantages of B+ tree usage for databases:

1. keeps keys in sorted order for sequential traversing
2. uses a hierarchical index to minimize the number of disk reads
3. uses partially full blocks to speed insertions and deletions
4. keeps the index balanced with a recursive algorithm
5. In addition, a B+ tree minimizes waste by making sure the interior nodes are at least half full. A B+ tree can handle an arbitrary number of insertions and deletions.

Differences between ISAM and B+ trees:

ISAM	B+ TREES
1. ISAM is static	1. B+ tree is dynamic
2. Only leaf pages are modified	2. Leaf and non-leaf nodes may be modified
3. Overflow pages required	3. No need of overflow pages
4. Size can increase to great extent if more insert operations are done	4. Rarely depth of 3 or 4 is reached maximum
5. Overflow chains degrade performance	5. No overflow chains . It is better than maintaining sorted file
6. Not balanced	6. Height of tree remains balanced
7. Search will be slower if more overflow pages are there	7. Faster search queries possible than ISAM as data is stored on leaf nodes
8. Not suitable for files that grow and shrink	8. Suitable for files that grow and shrink
9. Takes more time to retrieve record due to overflow chains	9. Takes less time when compared to ISAM
10. Index level pages locking can be omitted while accessing data	10. We cannot omit index level pages locking while accessing data
11. after deleting a element if leaf node (primary page)becomes empty ,then it is not removed and preserved for future insertions.	11. After deleting a element if leaf node is not 50 percent occupied, then we merge or redistribute nodes
12. At the time of file creation , all the leaf nodes are allocated sequentially	12. As file grows and shrinks dynamically it is not feasible to allocate leaf nodes sequentially