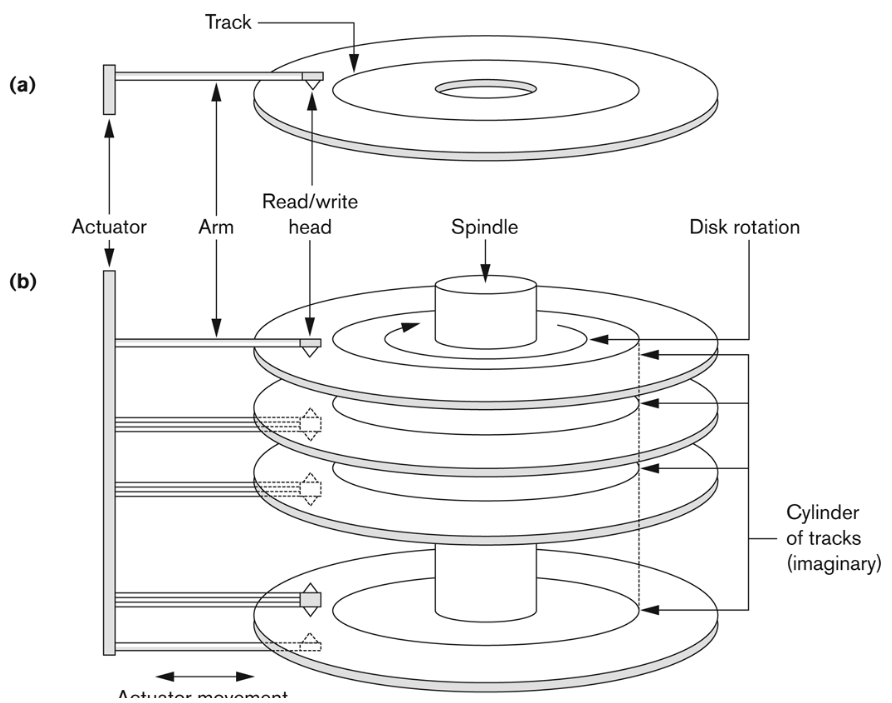
UNIT V (Part 1)
Database File Organization

Disk Storage Devices:

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A **disk pack** contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular **tracks** on each disk **surface**.
    - Track capacities vary typically from 4 to 50 Kbytes or more
- A track is divided into smaller **blocks** or **sectors**
    - because it usually contains a large amount of information
- The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.
    - One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector.
- A track is divided into **blocks**.
    - The block size B is fixed for each system.
        - Typical block sizes range from B=512 bytes to B=4096 bytes.
    - Whole blocks are transferred between disk and main memory for processing.
- A **read-write head** moves to the track that contains the block to be transferred.
    - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
    - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
    - the track number or surface number (within the cylinder)
    - and block number (within track).
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) **rd**.
- Double buffering can be used to speed up the transfer of contiguous disk blocks.

A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.



Records:

- Fixed and variable length records
- Records contain fields which have values of a particular type
    - E.g., amount, date, time, age
- Fields themselves may be fixed length or variable length
- Variable length fields can be mixed into one record:
    - Separator characters or length fields are needed so that the record can be "parsed."

Blocking:

- Refers to storing a number of records in one block on the disk.
- There may be empty space in a block if an integral number of records do not fit in one block.
- The **blocking factor bfr** for a file is the (average) number of file records stored in a disk block.
- A file can have **fixed-length** records or **variable-length** records.

Files of Records

- File records can be **unspanned** or **spanned**
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.
- Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.
  - Usually spanned blocking is used with such files.

Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.

Ordered Files

- Also called a sequential file.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A binary search can be used to search for a record on its *ordering field* value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries <field value, pointer to record>, which is ordered by field value.
- The index is called an access path on the field.
- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
- A **dense index** has an index entry for every search key value (and hence every record) in the data file.
- A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values.
- Example:
  Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
  Suppose that:
  record size R=150 bytes block size B=512 bytes r=30000 records
- Then, we get:
  blocking factor Bfr= B div R= 512 div 150= 3 records/block
  number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a key field
  - Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*

- A similar scheme can use the *last record* in a block.
- A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

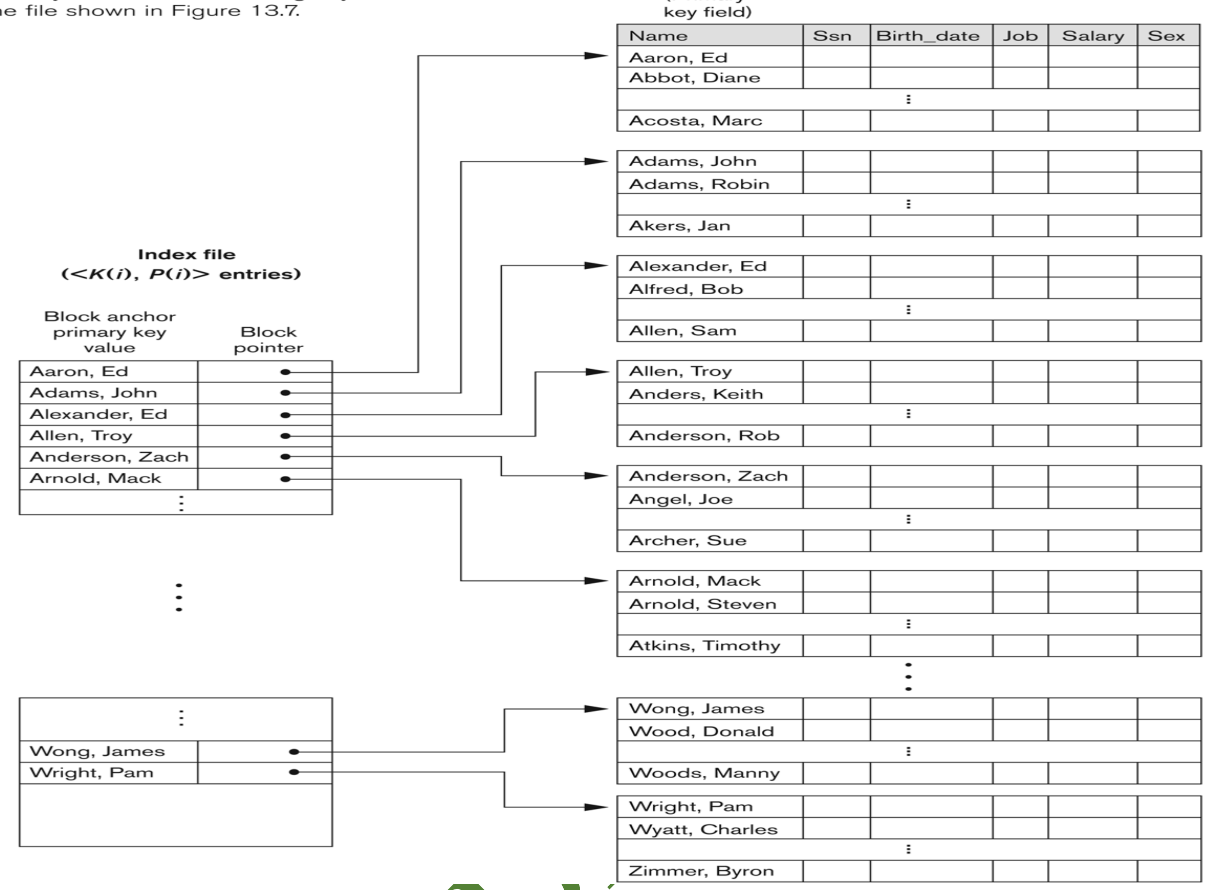
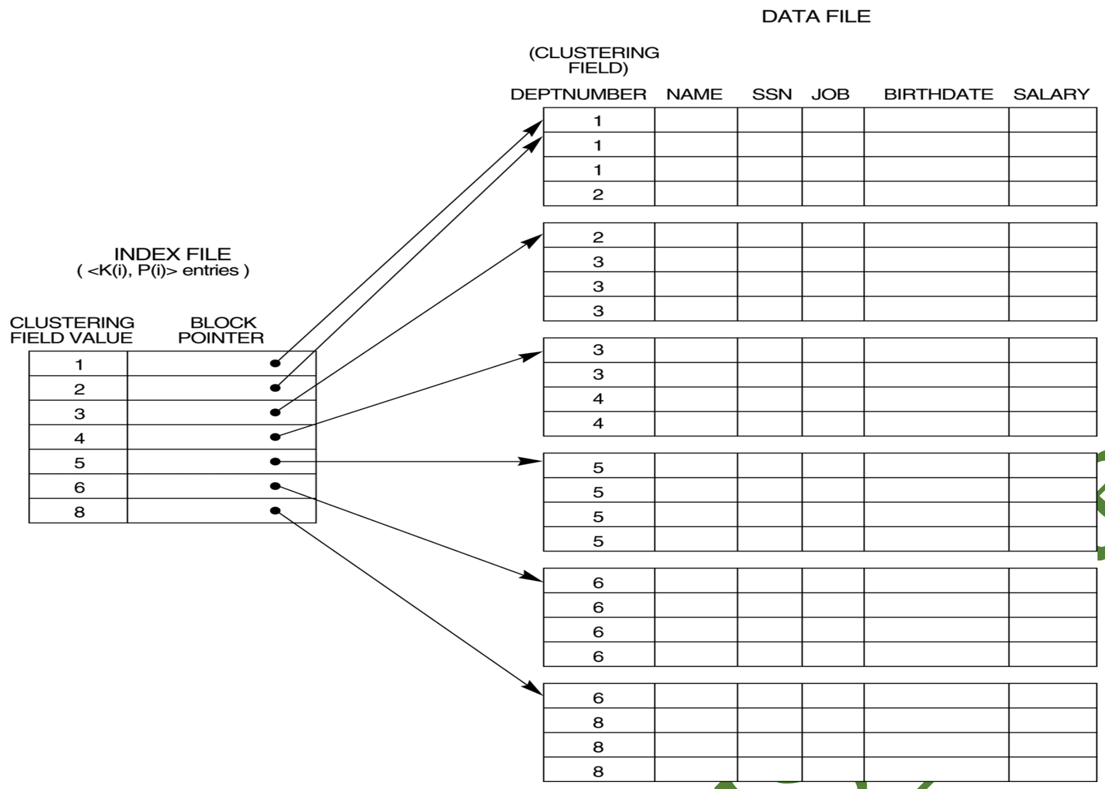


Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.

Clustering Index

- Defined on an ordered data file
- The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
- Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
- It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.



Secondary Index

- A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- The index is an ordered file with two fields.
- The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
- The second field is either a **block** pointer or a record pointer.
- There can be *many* secondary indexes (and hence, indexing fields) for the same file.
- Includes one entry *for each record* in the data file; hence, it is a *dense index*.

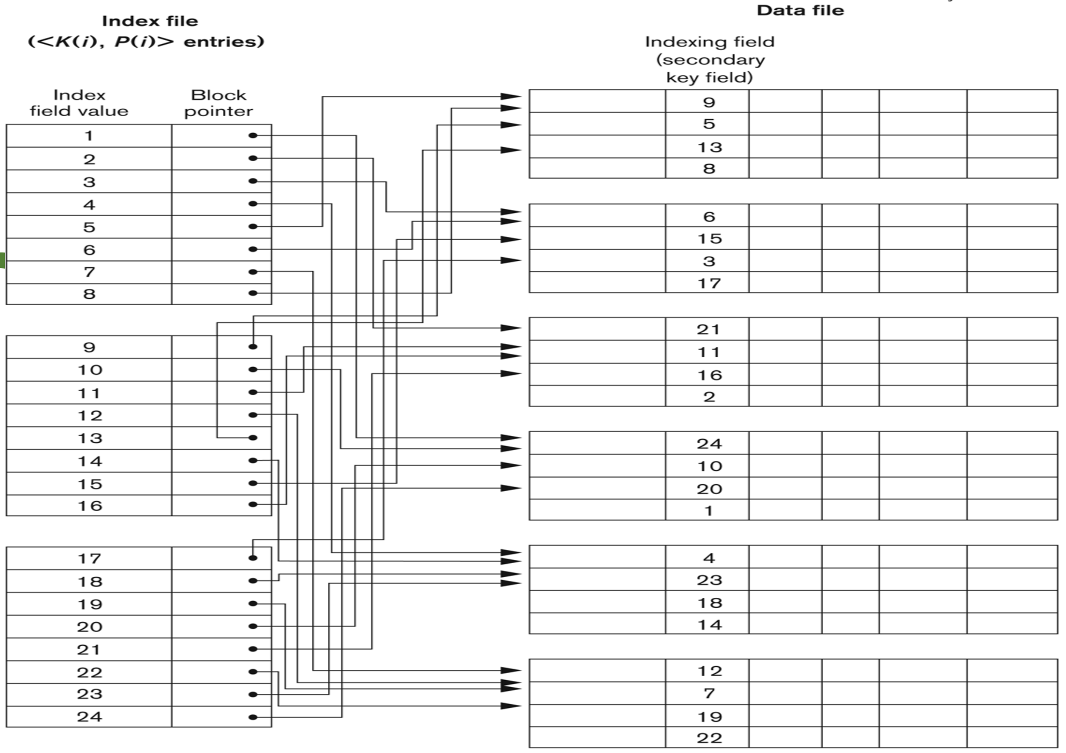


Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.

Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or Number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.

[b]For option 1.

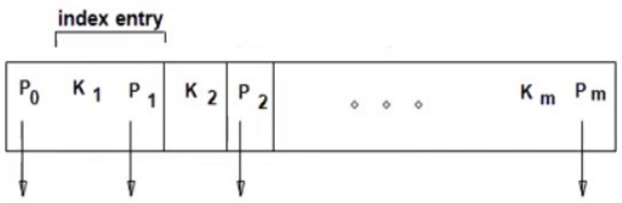
[c]For options 2 and 3.

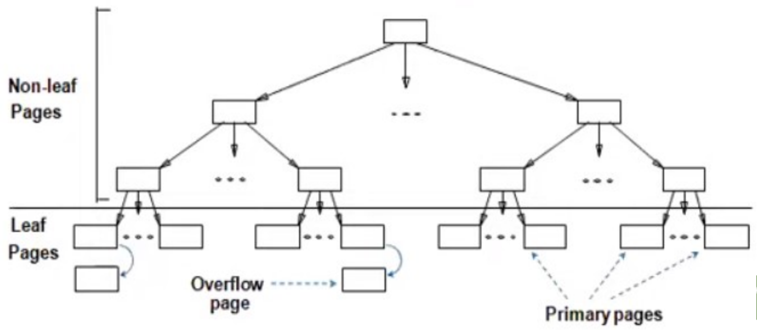
Multi-Level Indexes

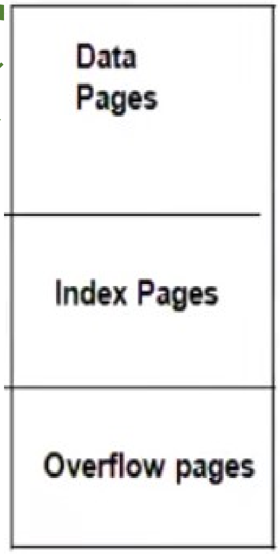
- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block
- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

Indexed sequential access method (ISAM)

- INDEXING: It is a data structure technique which is used to quickly locate and access the data in DB.
- Introduction:
  - ISAM is a static index structure.
  - ISAM is a method for creating, maintaining and manipulating files of data.
  - Records can be retrieved sequentially or randomly by one or more keys.
  - ISAM method is an advanced sequential file organisation.
- Records are stored in the file using Primary key.
- An index value is generated for each primary key and mapped with the record.
- This index contains the address of the record in the file.
- If any record must be retrieved based on its
- index value , then the address of the data block is fetched, and the record is retrieved from the memory.

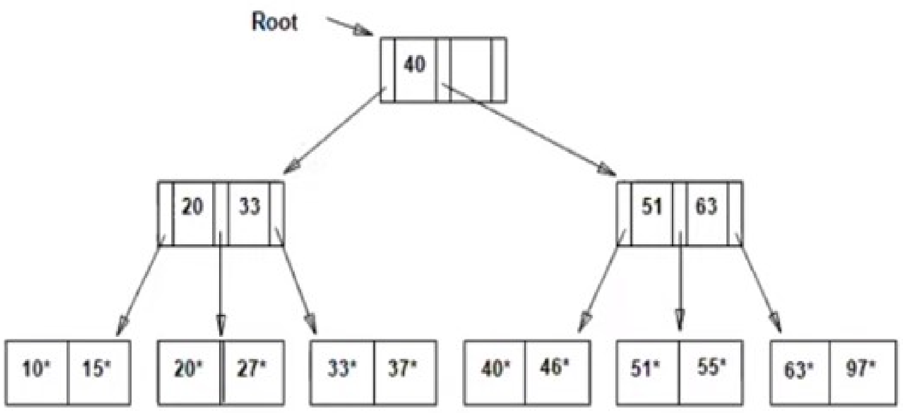




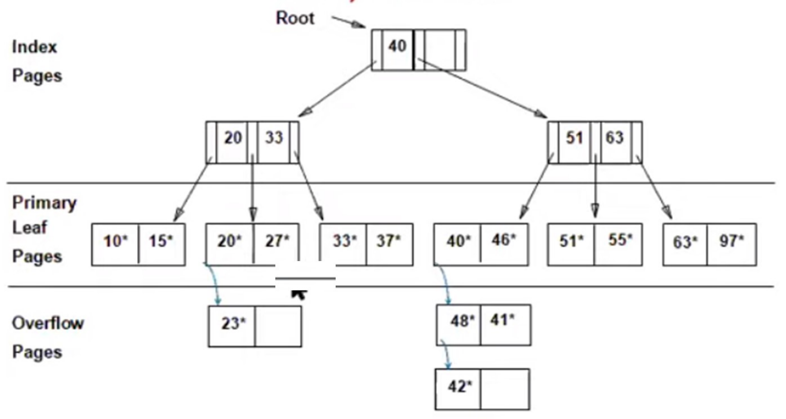


File creation: Leaf(data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
Index entries: <search key value, page id>; they 'direct' search for data entries, which are in leaf pages.
Search: Start at root; use key comparisons to go to leaf.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

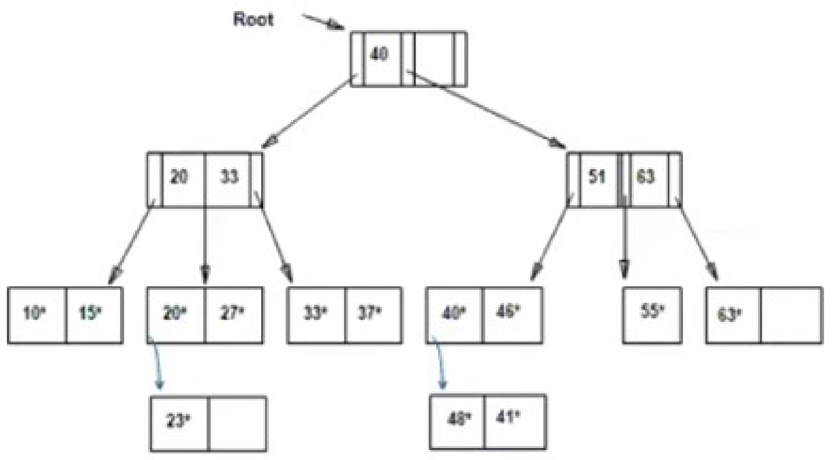
Example of ISAM Tree



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



Then Deleting 42*, 51*, 97*



Pros of ISAM:

- Each record has the address of its data block, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values , we can retrieve the data for the given range of values. In the same way, partial value also can be searched.,
- ex: ‘cvr’ from name cvrcollege.

Cons of ISAM:

- This method requires extra space in the disk to store the index value.
- When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- When the record is deleted, the space used by it needs to be released. Otherwise, the performance of the DB will shutdown.

B-trees and B+ trees

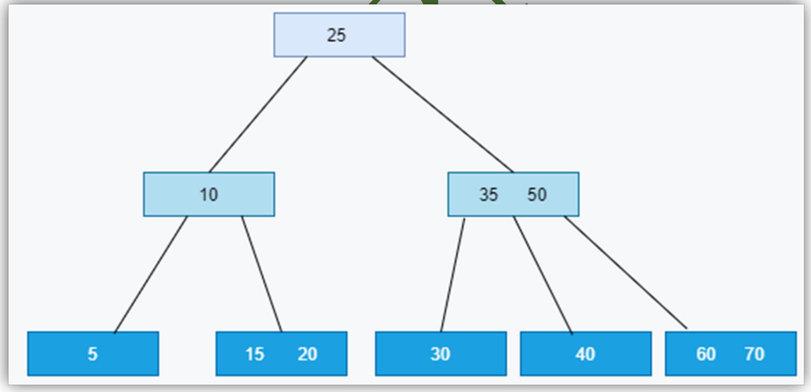
Introduction to B Trees:

- ☐ B Tree(Bayer) is a self-balancing tree data structure. It stores and maintains data in a sorted form where the left children of the root are smaller than the root and the right children are larger than the root in value.
- ☐ It makes searching efficient and allows all operations in logarithmic time. It allows nodes with more than two children.
- ☐ B-tree is used for implementing multilevel indexing. Every node of the B-tree stores the key-value along with the data pointer pointing to the block in the disk file containing that key.

Properties of B-Tree:

- Every node has at most m children where m is the order of the B-Tree.
- A node with K children contains K-1 keys.
- Every non-leaf node except the root node must have at least ⌈m/2⌉ child nodes.
- The root must have at least 2 children if it is not the leaf node too.
- All leaves of a B-Tree stay at the same level.
- Unlike other trees, its height increases upwards towards the root, and insertion happens at the leaf node.
- The time complexity of all the operations of a B-Tree is O (log n), here 'n' is the number of elements in the B-Tree.

Example of B Tree order of 4



B+ Trees:

- ■ A balanced binary search tree is the B+ Tree. It uses a multilevel indexing system.
- ■ Leaf nodes in the B plus tree represent actual data references. The B plus tree keeps all of the leaf nodes at the same height.
- ■ A link list is used to connect the leaf nodes in the B+ Tree. As a result, a B+ Tree can allow both random and sequential access.

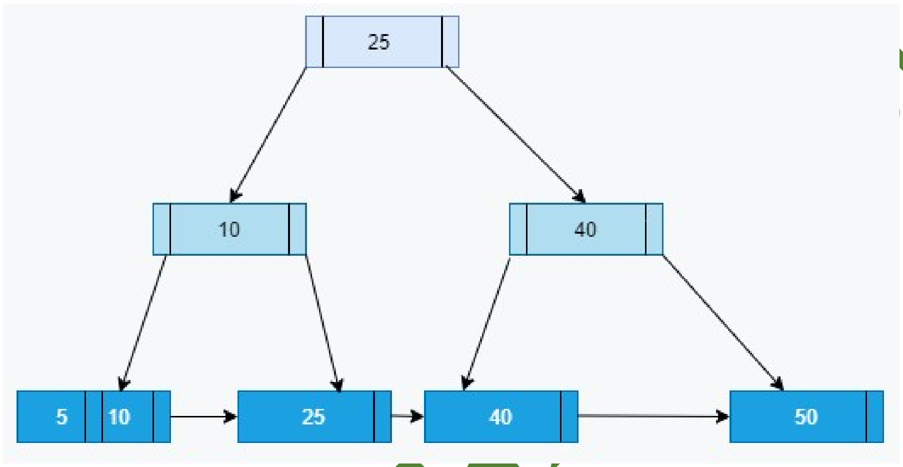
Properties of B+ Trees:

- ■ All data is stored in the leaf nodes, while the internal nodes store just the indices.
- ■ Each leaf is at the same height.
- ■ All leaf nodes have links to the other leaf nodes.
- ■ The root node has a minimum of two children.
- ■ Each node except root can have a maximum of m children and a minimum of m/2 children.
- ■ Each node can contain a maximum of m-1 keys and a minimum of ⌈m/2⌉ - 1 keys.

Advantages:

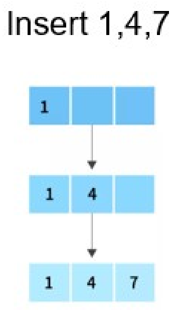
- Both keys and records can be placed in the internal and leaf nodes of a B Tree.
- In a B+ Tree, records or data can only be kept on the leaf nodes, whereas key values can only be placed on the internal nodes.
- To make search queries more efficient, the leaf nodes of the B+ tree in the data structure are connected in the form of singly linked lists.
- B+ Trees are used to store vast amounts of data that are too large to fit in the main memory. The internal nodes of the B+ Tree (the keys to access records) are securely stored memory, whereas leaf nodes are placed in the secondary memory due to the restricted amount of main memory.
- B plus tree internal nodes are often referred to as index nodes. The following diagram depicts a B+ Tree of order 3.

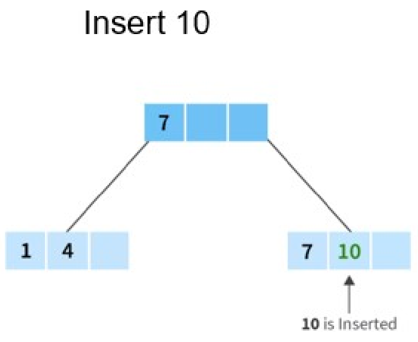
Example:

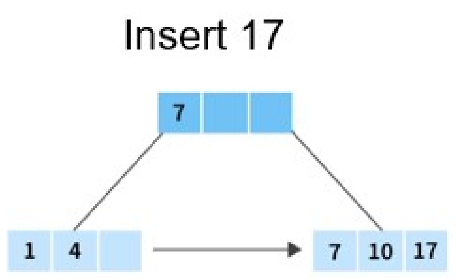


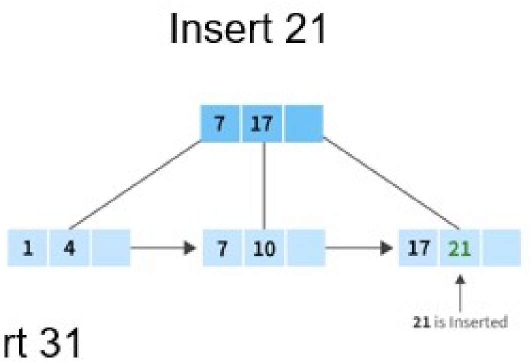
- We need to use the following data to create the B+ Tree:
  1, 4, 7, 10, 17, 21, 31
- We suppose the order(m) of the tree to be 4. The following facts can be deduced from this:
  - Max Children = 4
  - Min Children: m/2 = 2
  - Max Keys: m - 1 = 3
  - Min Keys: [m/2] - 1 = 1

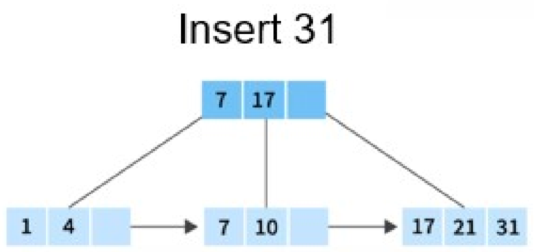
Insertion









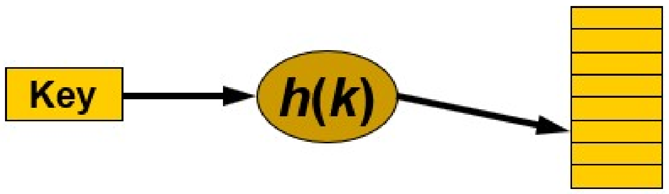


Hashing

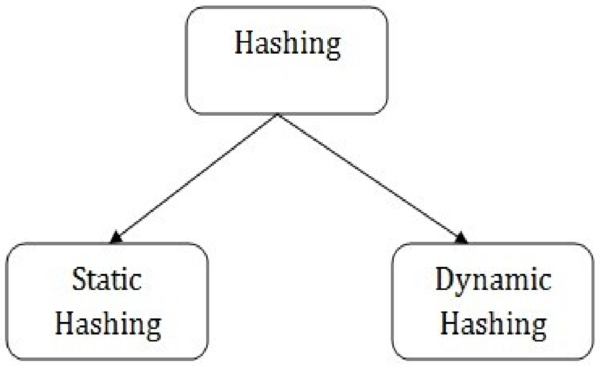
- Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.
- Hashing uses the hash function with search key as parameters to generate the address of a data record.

Fundamentals

- Define *m* target addresses (the "buckets")
- Create a ***hash function*** $h(k)$ that is defined for all possible values of the key k and returns an integer value h such that $0 \leq h \leq m - 1$



Types Hashing



Bucket –

A bucket is a type of storage container. Data is stored in bucket format in a hash file.
A bucket stores one entire disc block, which can then store one or more records.

Bucket sizes

- Each bucket consists of one or more blocks
  - Need some way to convert the hash value into a logical block address
- Selecting large buckets means we will have to search the contents of the target bucket to find the desired record.
  - If search time is critical ***and*** the database infrequently updated, we should consider sorting the records inside each bucket.

Static Hashing

- Whenever a search-key value is given in static hashing, the hash algorithm always returns the same address.
- Hence in this static hashing, the number of data buckets in memory remains constant throughout.

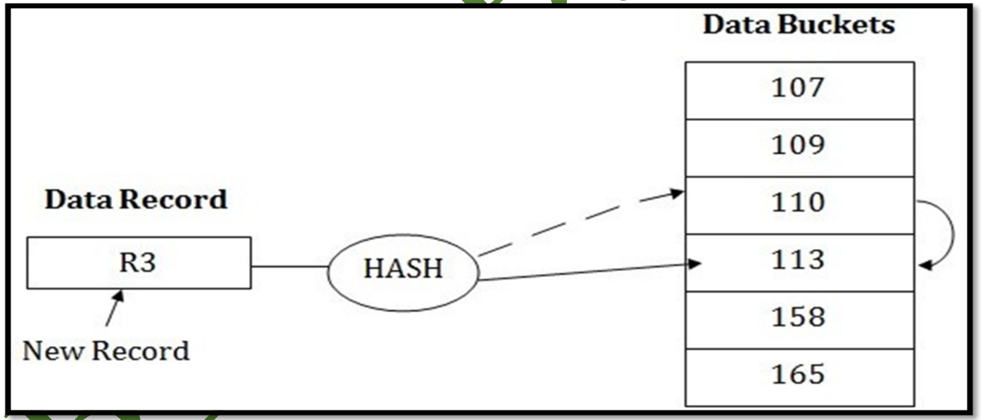
Operations of Static Hashing

- **Searching a record**
  - When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.
- **Insert a Record**
  - When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.
- **Delete a Record**
  - To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.
- **Update a Record**
  - To update a record, we will first search it using a hash function, and then the data record is updated.
  - If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address.
  - This situation in static hashing is known as **bucket overflow.** This is a critical situation in this method.

  To overcome this situation, there are various methods. Some commonly used methods are as follows:

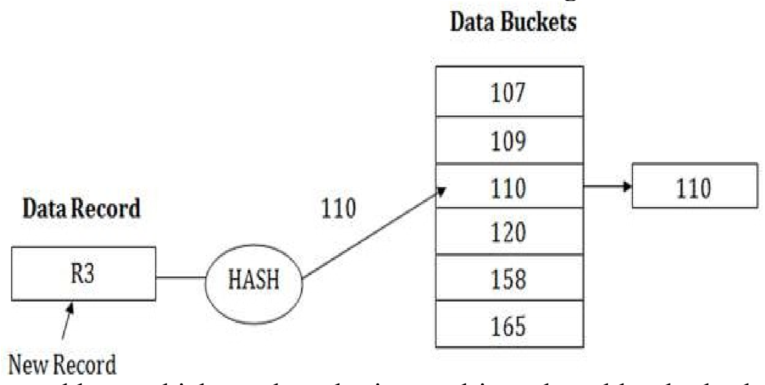
Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as ***Linear Probing.***



Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.



Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it.

But this bucket is full to store the new data.
In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.

Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases.
- This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address.
- This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

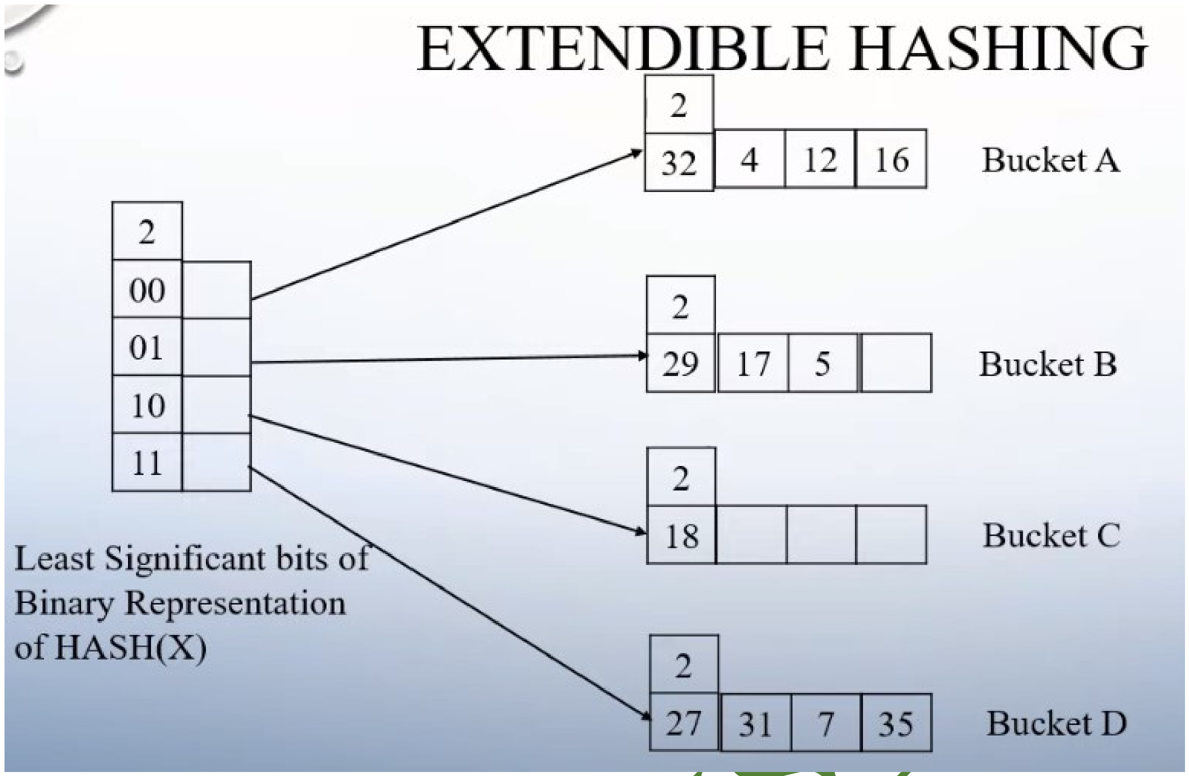
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

For example:

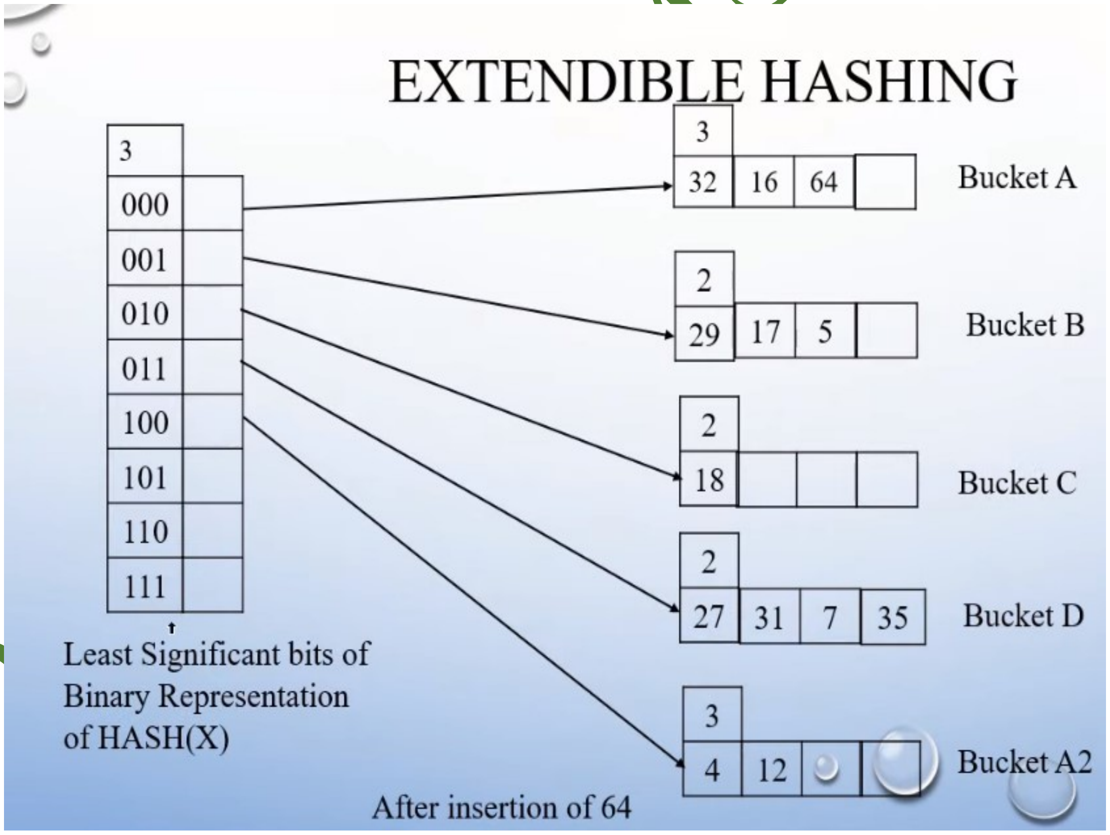
- Consider the following grouping of keys into buckets, depending on the prefix of their hash address:
- The last two bits of 2 and 4 are 00. So it will go into bucket B0.
- The last two bits of 5 and 6 are 01, so it will go into bucket B1.
- The last two bits of 1 and 3 are 10, so it will go into bucket B2.
- The last two bits of 7 are 11, so it will go into B3.

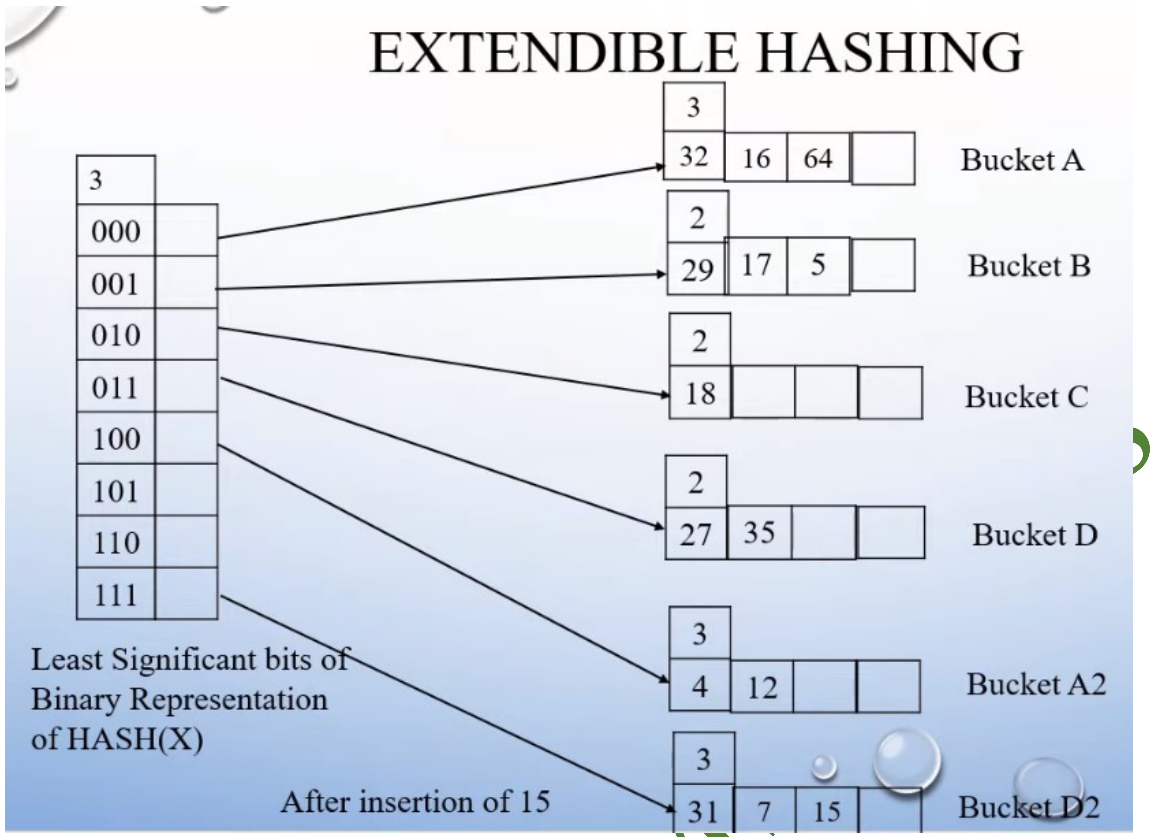
Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

EXTENDIBLE HASHING



EXTENDIBLE HASHING





Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system.
- It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data.
- There will not be any unused memory lying.
- This method is good for dynamic databases where data grows and shrinks frequently.