CHAPTER - 8

FILE ORGANIZATION AND INDEXING

8.1 Disks and Storage

Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed by the cost per unit of data to buy the medium and by the medium's reliability. Among the media typically available are these:

i. Cache
ii. Main Memory
iii. Flash Memory
iv. Magnetic Disk Storage
v. Optical Storage
vi. Tape Storage

i. Cache

The cache is the fastest and most costly form of storage. Cache memory is small, its use is managed by the computer system hardware.

ii. Main Memory

The storage medium used for data that are available to be operated on its main memory. The general-purpose machine instructions operate on main memory. Although main memory may contain many megabytes of data or even gigabytes of data in large server systems, it is generally small (or too expensive) for storing the entire database. The contents of main memory are usually lost if power failure or system crash occurs; main memory is therefore said to be volatile

iii. Flash Memory

In flash memory data survives the power failure-that is ,it is non-volatile. Flash memory has a lower cost per byte than main memory, but a higher cost per byte than magnetic disks. Flash memory is widely used for data storage in devices such as cameras and cell phones. Flash memory is also used for storing data in "USB flash drives," also known as "pen

drives," which can be plugged into the Universal Serial Bus (USB) slots of computing devices.

iv. Magnetic Disk

The primary medium for the long-term online storage of data is the magnetic disk drive, which is also referred to as the hard disk drive (HDD). Magnetic disk, like flash memory, is non-volatile: that is, magnetic disk storage survives power failures and system crashes. Disks may sometimes fail and destroy data, but such failures are quite rare compared to system crashes or power failures.

v. Optical Storage

The digital video disk (DVD) is an optical storage medium, with data written and read back using a laser light source. The Blu-ray DVD format has a capacity of 27 gigabytes to 128 gigabytes, depending on the number of layers supported. Although the original (and still main) use of DVDs was to store video data, they are capable of storing any type of digital data, including backups of database contents. DVDs are not suitable for storing active database data since the time required to access a given piece of data can be quite long compared to the time taken by a magnetic disk.

vi. Tape Storage

Tape storage is used primarily for backup and archival data. Archival data refers to data that must be stored safely for a long period of time, often for legal reasons. Magnetic tape is cheaper than disks and can safely store data for many years. However, access to data is much slower because the tape must be accessed sequentially from the beginning of the tape; tapes can be very long, requiring tens to hundreds of seconds to access data. For this reason, tape storage is referred to as sequential-access storage. In contrast, magnetic disk and SSD storage are referred to as direct-access storage because it is possible to read data from any location on disk.

8.2 Organization of Records into Blocks

A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks. A file is organized logically as a sequence of records. These records are mapped onto disk blocks. Each file is also logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.

Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created. A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used.

There are two types of Records:

i. Fixed length Records.
ii. Variable length Records.

8.2.1 Fixed Length Records

As an example, let us consider a file of *instructor* records for our university database. Each record of this file is defined (in pseudocode) as:

type *instructor* = **record**
ID **varchar** (5);
name **varchar**(20);
dept name **varchar** (20);
salary **numeric** (8,2);
end

Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Then, the *instructor* record is 53 bytes long.

A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on. However, there are two problems with this simple approach:

i. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is,

part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

ii. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	EI said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure: File containing instructor records

To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused. When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead.

Such approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record. It is undesirable to move

records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	EI said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

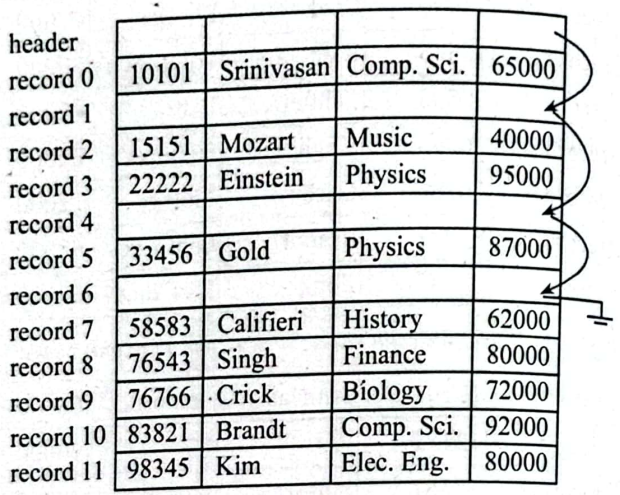
Figure: Instructor file, with record 3 deleted and all records moved

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	EI said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Figure: Instructor file with record 3 deleted and final record moved

At the beginning of the file, we allocate a certain number of bytes as a *file header*. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted.

We use this first record to store the address of the second available record, and so on. We can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**. The figure below the file of instructor, with the free list, after records 1, 4, and 6 have been deleted.



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure: *File of Instructor, with free list after deletion of records 1, 4, and 6*

On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file. Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record.

8.2.2 Variable Length Records

Variable-length records arise in database systems in several ways:

i. Storage of multiple record types in a file.

ii. Record types that allow variable lengths for one or more fields.

iii. Record types that allow repeating fields, such as arrays or multisets.

Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:

i. How to represent a single record in such a way that individual attributes can be extracted easily.

ii. How to store variable-length records within a block, such that records in a block can be extracted easily.

The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes. Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value.

Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (offset, length), where offset denotes where the data for that attribute begins within the record, and length is the length in bytes of the variable-sized attribute. The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length. An example of such a record representation is shown in figure:

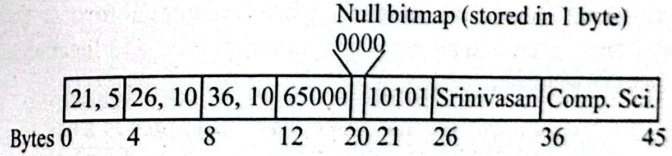


Figure: *Representation of variable-length record*

The figure shows an instructor record, whose first three attributes ID, name, and dept name are variable-length strings, and whose fourth attribute salary is a fixed-sized number. We

assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The salary attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

The figure also illustrates the use of a null bitmap, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored. The slotted-page structure is commonly used for organizing records within a block, and is shown in Figure. There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header.
2. The end of free space in the block.
3. An array whose entries contain the location and size of each record.

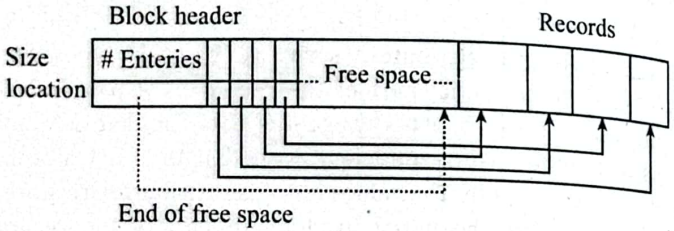


Figure: Slotted page structure

The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed, and its entry is set to deleted (its size is set to −1, for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the

header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well.

Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: typical values are around 4 to 8 kilobytes.

The slotted-page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.

Byte String Representation

In byte string representation we attach a special end of record(⊥) symbol to the end of record. Each record is stored as a string of successive bytes.

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Figure: Byte string representation

Byte string notation has several disadvantages:

i. It is not easy to reuse the space left by a deleted record

ii. In general there is no space for records to grow longer. If the records become longer it must be moved

Fixed Length Representation with Reserved Space

Can use fixed length records of a known maximum length, unused space in shorter records filled with a null or end of record symbol.

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Figure: Fixed length representation with reserved space

Fixed Length Representation with Pointer

A variable length record is represented by a list of fixed length records that are chained together via pointers. Can be used even if the maximum record length is not known.

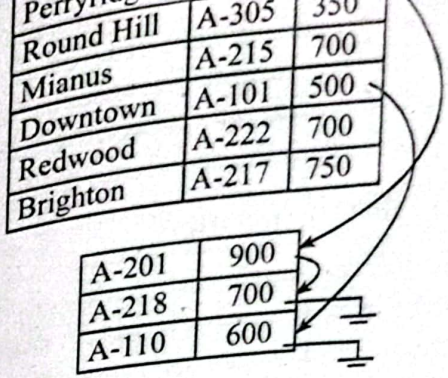
0	Perryridge	A-102	400
1	Round Hill	A-305	350
2	Mianus	A-215	700
3	Downtown	A-101	500
4	Redwood	A-222	700
5		A-201	900
6	Brighton	A-217	750
7		A-110	600
8		A-218	700

Figure: Fixed length representation with pointer

The main disadvantage of the pointer structure is that space is wasted in all records except the file in chain. Alternative solution (for fixed-length representation with pointer) which uses two different kinds of block in a file.

Anchor block: contains the first record of each chains.

Overflow block: contains records other than those that are first records of chains.



Round Hill	A-305
Mianus	A-215
Downtown	A-101
Redwood	A-222
Brighton	A-217

A-201	900
A-218	700
A-110	600

Figure: Fixed length representation with pointer using anchor block and overflow block

8.3 File Organizations

The way data is organized and stored within a file or group of files in a computer system is referred to as file organization. It governs how information is stored in the file(s), accessed, and retrieved. The type of data, access patterns, performance requirements, and simplicity of maintenance all play a role in the choice of file organization.There are different ways to logically organize records in a File. They are:

1. **Heap file organization**
   A record can be placed anywhere in the file where there is space; there is no ordering in the file.
2. **Sequential file organization**
   Store records in sequential order based on the value of the search key of each record.
3. **Hashing file organization**
   A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file, the record is placed
4. **Multitable Clustering file organization**
   Records of several different relations can be stored in the same file. Motivation of clustering file organization is to store related records on the same block to minimize I/O.

5. **B+-tree file organization.**

The B+-tree file organization is related to the B+-tree index structure and can provide efficient ordered access to records even if there are a large number of insert, delete, or update operations. Further, it supports very efficient access to specific records, based on the search key.

8.3.1 Sequential File Organization

A sequential file organization is designed for efficient processing of records in sorted order based on some key. A search key is any attribute or set of attributes; it need not be the primary key, or even super key. Sequential file organization is suitable for applications that require sequential processing of the entire file.

To permit fast retrieval of records in search key order, we chain together records by pointers. The pointer in each record points to the next record in search key order. To minimize the number of block accesses in sequential file processing, we store records physically in search key order or as close to the search key order as possible

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure: Sequential file for instructor record

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
32222	Verdi	Music	48000

Figure: Sequential file after an insertion

The sequential file organization allows records to be read in sorted order; that can be useful for display propose as well as for certain query processing algorithms. It is difficult however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion.

We manage deletion by using pointer chains as we saw previously. For insertion we apply the following rules:

1. Locate the record in the file that comes before the record to be inserted in search key order
2. If there is a free record (that is space left after a deletion) with in the same block as this record, insert new record there. Otherwise insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search key order

8.3.2 Indexed Sequential File Organization

Indexed Sequential File Organization is a method of organizing data in a computer system where the data is stored in a sequential order, but an index is used to access the data quickly.

In Indexed Sequential File Organization, the data is kept in fixed length blocks or pages, and each block is given a specific index that points to the block's first record. The index is usually maintained in a separate file or data structure, and it is sorted based on the primary key of the data.

To access a specific record in an indexed sequential file, the system first searches the index for the appropriate block or page. Once the block is located, the system can then access the record sequentially within the block. This allows for fast random access to specific records while also providing efficient sequential processing of data.

The indexed sequential file organization is commonly used in large databases where data needs to be accessed quickly and efficiently. It is also useful for applications that require a combination of sequential and random access to data, such as financial systems or inventory management systems.

Example:

Let's look at an illustration of an indexed sequential file arrangement in the context of a student database.

Suppose we have a table that contains the following fields:

Student ID (primary key)

Name

Age

Gender

Address

The data would be kept in fixed-length blocks or pages with a specific number of records on each block in an indexed sequential file organization. Consider the case when there are 50 records in each block.

Each index entry in the index file points to the starting block of a series of data blocks containing records with key values within a certain range. The index entry would typically contain the highest key value in that range and the block number where those records begin. A sample of the index file might be as follows:

Student ID	Block
1050	1
1100	2
...	...
1250	5
1300	6
...	...
1500	10

Suppose we want to access the record for student ID 1275. The system would first search the index file to find the appropriate block number (in this case, block number 6). Once it has located the correct block, the system can then access the records sequentially within the block until it finds the record for student ID 1275.This allows for fast random access to specific records while also providing efficient sequential processing of data.

Indexing is used to optimize the performance of a database by minimizing the number of disk access required when a query is processed. The index is a type of data structure. It is used to locate and access the data in database quickly

Index structure

Index can be created by using some database columns

Search Key	Data Reference

The final column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily. The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Index Evaluation Metrices

Index evaluation metrices are the metrices which are used to determine the efficiency of index. Different factors determine how efficient an index is but followings are the metrices which play key role in determining the efficiency of an index:

i. **Access type**

Access type is finding records with specified attribute value. Finding records whose attribute value falls in a specified range.

ii. **Access time**

Access time is time to find particular value.

iii. **Insertion time**

Insertion time is time to insert new value as well as to update the index structure.

iv. **Deletion time**

Deletion time is time to delete data item as well as to update the index structure.

v. **Space overhead**

Space overhead is additional space occupied by an index structure.

There are two basic kinds of indices:

1. Ordered indices
2. Hash indices

- **Ordered Indices**

In an ordered index, the indices are usually sorted to make searching faster. The indices which are sorted and associates with each search key the records that contain it are known as ordered indices.

Example:

Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDS start with 1,2,3,... and so on and we have to search student with ID-543. In case of database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543*10=5430 bytes. In case of an index,

we will search using indexes and the DBMS will read the record after reading 542*2=1084 bytes which are very less compared to previous case.

a. **Dense Index**

The dense index contains an index record for every search key value in the data file. It makes searching faster. The number of records in the index table is the same as the number of records in the main table. It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

Pakistan	→	Pakistan	Lahore	1604300
USA	→	USA	Chicago	2789378
Nepal	→	Nepal	Kathmandu	1456634
UK	→	UK	Cambridge	1360364

Figure: Dense index

b. **Sparse Index**

In sparse index, instead of pointing to each record in the main table the index points to the records in the main table in gap, each item pointing to a block.

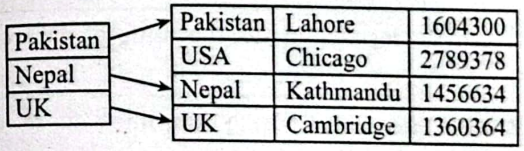


Figure: Sparse index

In this method of indexing, range of index columns store the same data block address. And when data is to be retrieved, the block address will be fetched linearly till we get the requested data. The main goal of this method should be more efficient search with less memory space

Student-ID	Student-name	Age	Course	Index address
100	Joseph	20	200	
103	Patty	22	205	
106	James	19	200	

	100	Joseph
	101	Alien
	102	Chris
	103	Patty
	104	Jack
	106	James
	107	Antomy
	108	Jacob

***Figure:** Sparse index*

Following are the differences between dense index and sparse index:

Dense Index	Sparse Index
Index size is larger	Index size is smaller
Records in the data file does not need clustering	Records in the data file needs clustering
Data can be located in less time	Locating of data takes more time
Consumes less computing time in RAM	Consumes more computing time in RAM
The overhead for the insertion and deletion is more	The overhead for the insertion and deletion is less
Pointers point to each record in the data file	Pointers point to fewer records in data file

c. Clustering index

A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record. In this case, to

The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.

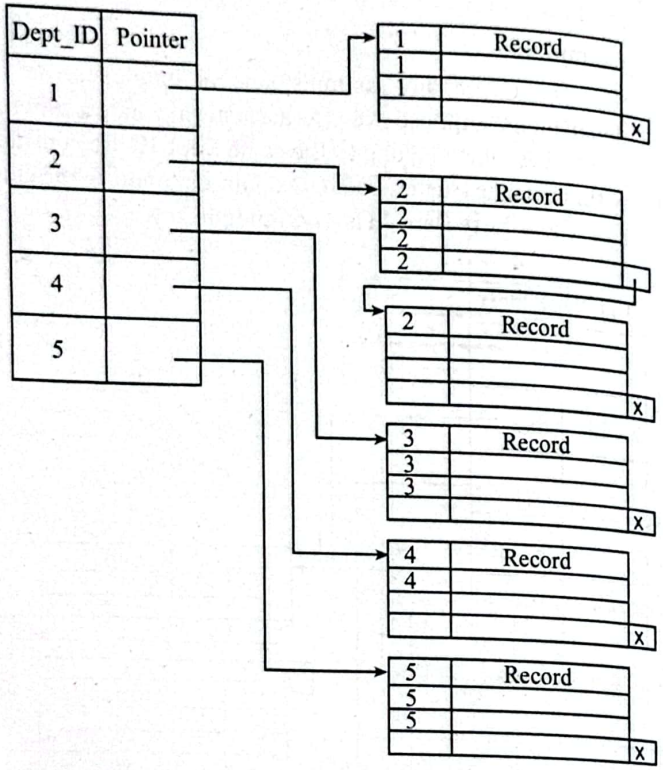


Figure: *Clustering index using separate disk block for separate clusters*

d. Primary Index

If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each and contain 1:1 relation between the records. As primary key are stored in sorted order, the performance of the searching operation is quite efficient.

e. Secondary index

Indices whose search key specifies an order different from the sequential order of the file are called non clustering indices, or secondary indices.

8.4 B+ Tree Index

B+ tree is a balanced binary search tree that uses a tree like structure to store records in File. B+ tree has three levels: the root, internal level and the leaves and also ensures that all leaf nodes remain at the same height, so it is balanced. The leaf node are linked using linked list, so B+ tree supports both random and sequential access. Order (n) of B+ tree represents the maximum possible no of children a node can have. Degree of B+ tree represents minimum possible no of children a node can have. In B + tree, Degree <= Order.

Structure of B+ tree

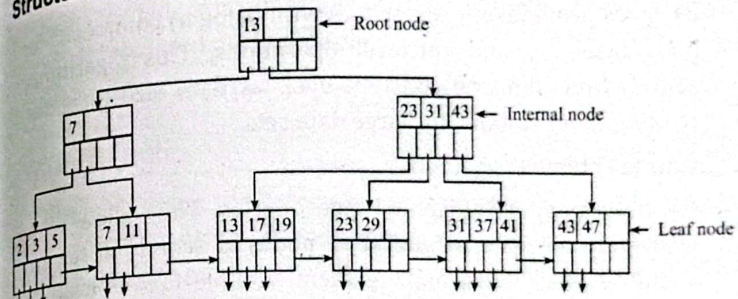


Figure: *Structure of B+ tree*

Root Node

Every leaf node is at equal distance from the root node and degree is 2.

Internal Node

Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node. At most, an internal node can contain n pointers where 'n' represents order of B-tree.

Leaf Node

Leaf nodes contain at least ⌈(n-1)/2⌉ record pointers and ⌈(n-1)/2⌉ key values. At most, a leaf node can contain n-1 record pointers and n-1 key values. Every leaf node contains one block pointer P to point to next leaf node and forms a linked list.

B+ tree Node Structure

Unlike binary tree, Node of B+ tree can have more than 1 key. Typical Node has following structure.

P_1	K_1	P_2	.....	P_{n-1}	K_{n-1}	P_n

K_i are the search key values and P_i are the pointers to children or pointers to records (in case of leaf node). The search keys in the node are ordered as:

$K_1 < k_2 < k_3 < .... < k_{n-1}$

Advantages of B+ Tree

a. Effective Search

B+ trees provide efficient search with O(log n) complexity for both insertion and retrieval operations. This logarithmic search time remains constant even as data size increases, making them suitable for large data sets.

b. Range Queries

B+ trees support efficient range queries. They allow us to efficiently navigate through tree nodes to search all records within a given area, making them suitable for applications where path-based searches are common.

c. Balanced Structure

B+ trees are balanced data structures. A balanced tree structure is automatically maintained for insertions and deletions, ensuring that the tree remains relatively shallow and the search time remains the same.

d. Sorted Order

B+ trees maintain data in a structured order, which can be useful for applications that require structured data retrieval or produce structured results without requiring additional processing.

e. Large Datasets Supported

B+ trees can handle large amounts of data more efficiently because they keep the tree height short, reducing the number of disk I/O operations required during recovery.

f. Effective Range Scan

B+ trees work particularly well for range scans or scans in consecutive parts of a data set. Their leaf nodes are often connected in overlapping lines, allowing for better sequential access.

g. Insertion and Deletion Support

B+ trees can handle insertions and deletions more efficiently without significant loss of performance. They maintain balance and avoid data fragmentation.

h. Disk I/O Reduction

B+ trees reduce disk I/O by using large fanout factors and manage data appropriately to reduce the number of disk blocks accessed during queries.

Disadvantage of B+ Tree is that it is Inefficient for static tables.

8.5 Hash Index

In a hash index, the search key is mapped to a location in the index structure using a hash function. The index structure normally consists of an array of pointers, or buckets, where each bucket holds a list of pointers to data entries with the same hash value.

Formally, let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A hash function h is a function from K to B. Let h denote a hash function. With in-memory hash indices, the set of buckets is simply an array of pointers, with the i^{th} bucket at offset i. Each pointer stores the head of a linked list containing the entries in that bucket.

To insert a record with search key Ki, we compute h(Ki), which gives the address of the bucket for that record. We add the index entry for the record to the list at offset i. Hash indices handle the case of multiple records in a bucket using overflow chaining. If the bucket does not have enough space, a bucket overflow is said to occur. We handle bucket overflow by using overflow buckets. If a record must be inserted into a bucket b, and b is already full, the system provides an overflow bucket for b and inserts the record into the overflow bucket. If the overflow bucket

is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list, as in Figure below. With overflow chaining, given search key k, the lookup algorithm must then search not only bucket h(k), but also the overflow buckets linked from bucket h(k).

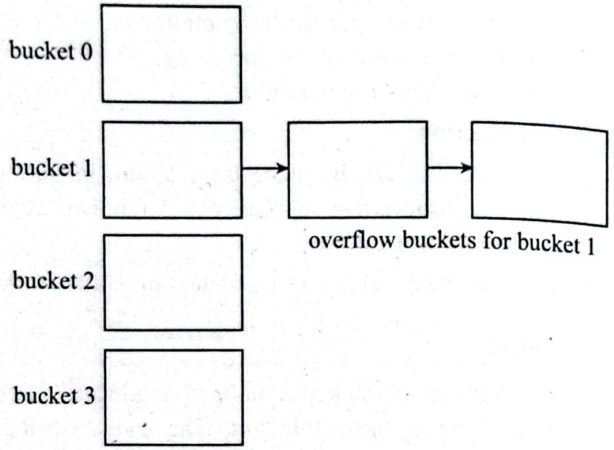


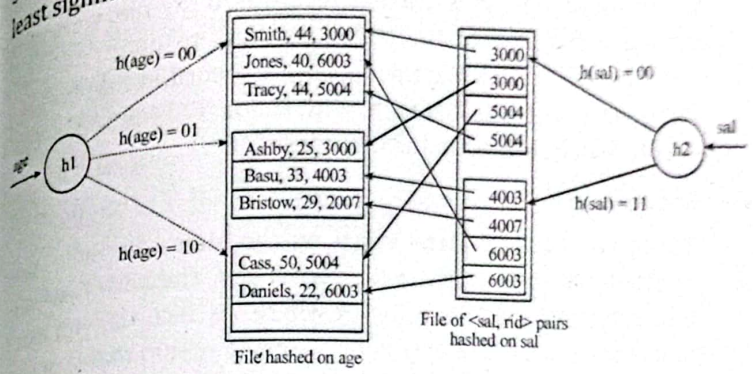
Figure: Overflow chaining

Hash indexing using overflow chaining is also called closed addressing (or, less commonly, closed hashing). An alternative hashing scheme called open addressing is used in some applications, but is not suitable for most database indexing applications since open addressing does not support deletes efficiently

To search for a record with a given search key value, we apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket. If we do not have the search key value for the record, for example, the index is based on sal and we want records with a given age value, we have to scan all pages in the file.

Hash indexing is illustrated in Figure, where the data is stored in a file that is hashed on age; the data entries in this first index file are the actual data records. Applying the hash function

to the age field identifies the page that the record belongs to. The hash function h for this example is quite simple; it converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier.



Example of File organization using hash indices

Suppose we have a database of students, and the table "Students" has the following attributes:

StudentID (primary key)

Name

Age

Major

We want to organize the file using the Major attribute as the search key.

- **Hash Function**

  Let's define a simple hash function that takes the Major attribute as input and output

  i.e., h(k)=k

  Hence, for CSE,ENG and BIO the hash values are CSE,ENG and BIO respectively

- **Buckets or Blocks**

  We divide the file into fixed-sized buckets or blocks, where each bucket corresponds to a unique major. For simplicity, let's assume we have three majors: "CSE" (Computer Science

and Engineering), "ENG" (English), and "BIO" (Biology). We create three buckets: one for each major.

- **Insertion**

  When inserting a new student record into the file, we apply the hash function to the Major attribute to determine the appropriate bucket. We then place the record in that bucket. For example, if a new student with Major "CSE" is added, the record will be placed in the CSE bucket.

- **Searching**

  To search for a student based on the Major attribute, we apply the hash function to the search key. The hash function will provide us with the bucket where the records with that major are stored. We only need to search within that specific bucket, making the search process more efficient.

- **Collision Handling**

  In case of collisions, where multiple records hash to the same bucket due to the limitations of the hash function or similar major names, collision handling techniques can be employed. For instance, separate chaining can be used, where each bucket maintains a linked list of records with the same hash value.

Example Data:

StudentID	Name	Age	Major
1	Alice	20	CSE
2	Bob	21	ENG
3	Charlie	19	CSE
4	David	20	BIO
5	Emma	22	CSE

Suppose we apply the hash function and divide the records into buckets based on the major. The resulting organization might look like this:

CSE Bucket:

StudentID: 1, Name: Alice, Age: 20, Major: CSE

StudentID: 3, Name: Charlie, Age: 19, Major: CSE

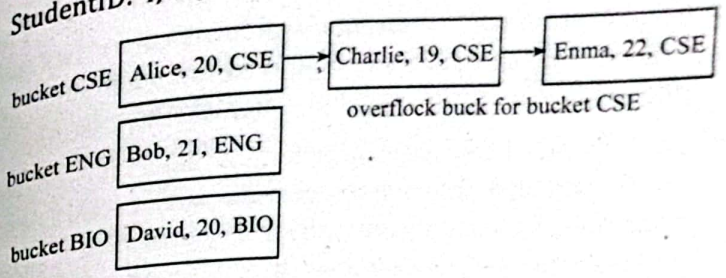
StudentID: 5, Name: Emma, Age: 22, Major: CSE

ENG Bucket:

StudentID: 2, Name: Bob, Age: 21, Major: ENG

BIO Bucket:

StudentID: 4, Name: David, Age: 20, Major: BIO



When searching for a student with a specific major, such as "CSE," we can directly access the CSE bucket and retrieve the relevant records without having to search through the entire dataset. This provides efficient and quick access to the desired data based on the search key.

SOLUTION TO EXAMS' AND OTHER IMPORTANT QUESTIONS

1. How do you evaluate the performance of magnetic disk? What are the optimization techniques to reduce the disk block access? [2021 Fall]

ANS: The performance of a magnetic disk can be evaluated using several metrics, including:

i. **Transfer Rate:** It measures how quickly information may be read from or written to a disk. It usually refers to the speed at which data may be moved to and from the disk and is represented in megabytes per second (MB/s).

ii. **Seek Time:** It calculates how long it takes the read/write head of the disk to move between various tracks. The seek time contributes to the overall latency of accessing data on the disk and is often expressed in milliseconds (ms).

iii. **Latency:** It shows the interval of time between a data request and the beginning of a data transfer. It comprises the seek time as well as the rotational delay (the amount of time it takes for the read/write head to revolve around the required data sector).

iv. **Input/Output Operations Per Second (IOPS):** It counts how many read/write operations the disk can handle in a single second. In general, better performance is indicated by more IOPS.

To optimize disk block access and reduce the number of disk block accesses, the following techniques can be employed:

i. **Caching:** By keeping frequently used data in faster storage (such RAM or solid-state drives) closer to the processor, a cache can greatly boost performance. Data served straight from the cache through caching decreases the need for disk block access.

ii. **Disk Partitioning:** Performance can be enhanced by partitioning a big disk into several smaller ones. We can cut down on the amount of time needed to seek out specific data blocks by grouping data according to usage patterns or access frequency.

iii. **Defragmentation:** Defragmenting the disk on a regular basis keeps files and data blocks organized and sequential, cutting down on search times and enhancing overall disk performance.

iv. **File System Optimization:** Performance on disks can be improved by selecting an effective file system. Clustering, journaling, and complex data structures are some of the methods used by file systems like NTFS (New Technology File System) and ext4 (Fourth Extended File System) to optimize disk access.

v. **RAID (Redundant Array of Independent Disks):** RAID implementation can increase performance and offer fault tolerance. For instance, disk striping is used in RAID 0 to distribute data over many disks, enabling simultaneous disk block access and improved performance.

vi. **Intelligent Disk Scheduling**

vii. **Data Compression**

viii. **Prefetching and Read-Ahead**

2. In terms of file organization, define indexing, Elevator Algorithm, Log disk. How does mechanical hard disk work? [2020 Fall]

ANS: **Indexing**

Indexing is a file organization technique that improves data retrieval efficiency by creating additional data structures called indexes. An index contains key-value pairs. The key is usually a field or combination of fields in the data file and the value is a pointer to the corresponding record. Indexes let us search and retrieve specific records faster based on indexed key values instead of scanning the entire file.

Elevator Algorithm

The elevator algorithm is a disk scheduling algorithm used to optimize the order of disk access requests. This algorithm works by maintaining a queue of pending disk requests and processing them in a particular direction. First, the head starts at one end of the disk and moves in one direction, servicing all requests in that direction until it reaches the end. Then change direction and process the remaining requests in the opposite direction.

Log Disk

Log disk is a technique used by some file systems to improve the efficiency and reliability of disk operations. It keeps log files in a separate area of the disk that record changes to the file system before they are applied to the main data structures. A log file serves as a continuous record of all disk operations, such as file modifications, deletions, and metadata updates. A log disk enables fast and recoverable file system operations because the log can be used to restore the file system to a consistent state in the event of a system failure or crash.

Working of Mechanical Hard Disk

A mechanical hard drive or hard disk drive (HDD) is a non-volatile storage device that uses spinning disks or platters to store and retrieve data. Here's an overview of how mechanical hard drives work:

Platters

A hard drive consists of several circular platters coated with a magnetic material. These platters are stacked one on top of the other and spin at high speeds (typically thousands of revolutions per minute) on spindles.

Read/Write Heads

Each platter has its own read/write head floating just above its surface. The read/write head is attached to an actuator arm assembly so it can move across the surface of the platter.

Data Access

To read or write data, the read/write head moves to the desired location on the platter. When reading data, the head detects the magnetic field on the disk surface and converts it into an electrical signal. When writing data, the head magnetically encodes the data onto the platters.

Sectors and Tracks

The disk surface is divided into concentric tracks and sectors. Tracks are concentric circles on the platter and sectors are small pie-shaped areas within the track. Each sector can store a fixed amount of data.

Spindle Motor

The platter is rotated by a spindle motor that rotates the platter at a constant speed. Rotation speed is measured in revolutions per minute (RPM).

Seek Time

The time it takes for the read/write head to reach the desired track is called the seek time. Faster search times mean faster data access.

Latency

The time it takes for the read/write head to be positioned and for the desired sector to spin underneath it is called latency. Affected by disk rotation speed.

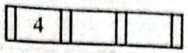
---

3. Create a B+ tree of order 4 with following data: (4,9,16,25,1,20,13,15,10,11,12) of order 4. Assume that, tree is initially empty and values are added in ascending order. Show the formation of tree after the deletion of 16. **[2019 Spring]**

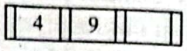
ANS: To create a B+ tree of order 4 with the given data (4,9,16,25,1,20,13,15,10,11,12), we will follow these steps:

Start with an empty B+ tree.

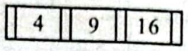
Insert 4:



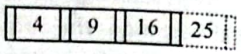
Insert 9:



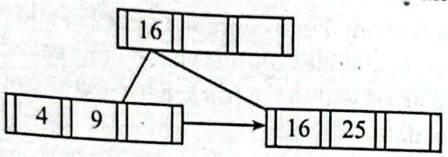
Insert 16:



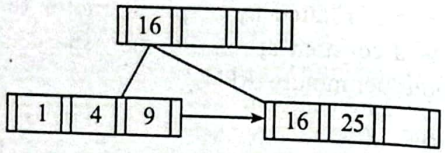
Insert 25:



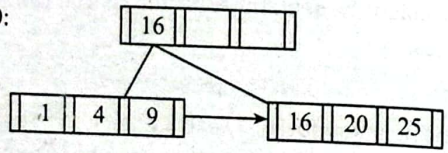
Split from 9 or 16. Here we choose 16 and move 16 up.



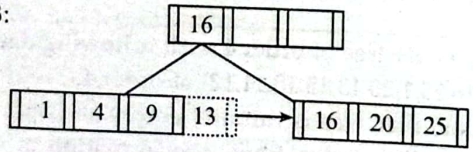
Insert 1:



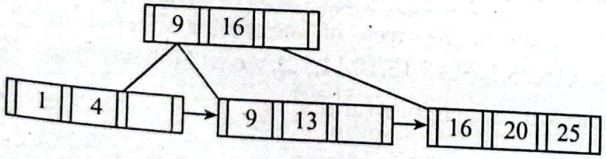
Insert 20:



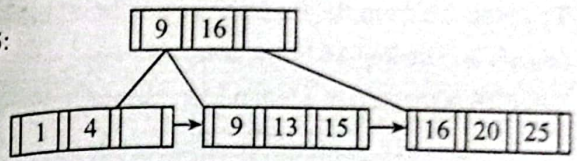
Insert 13:



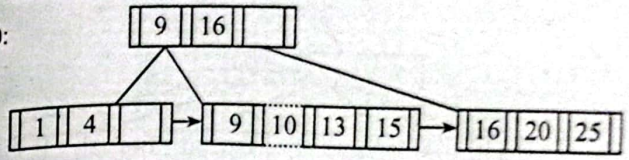
Split from 9 and move 9 up.



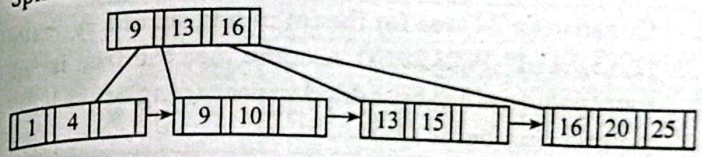
Insert 15:



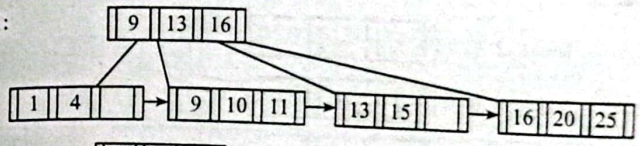
Insert 10:



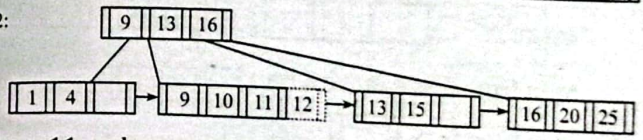
Split from 13 and move 13 up



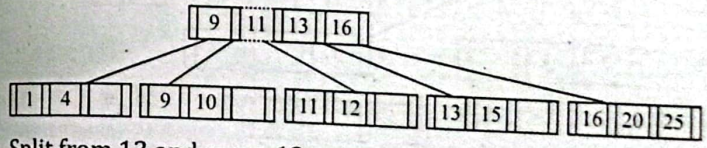
Insert 11:



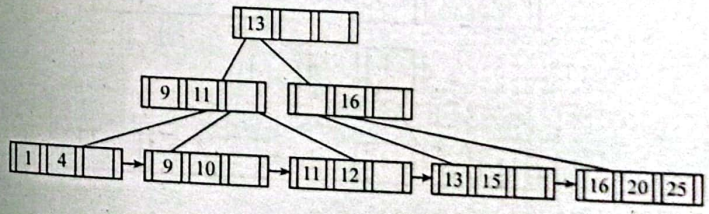
Insert 12:



Split from 11 and move 11 up



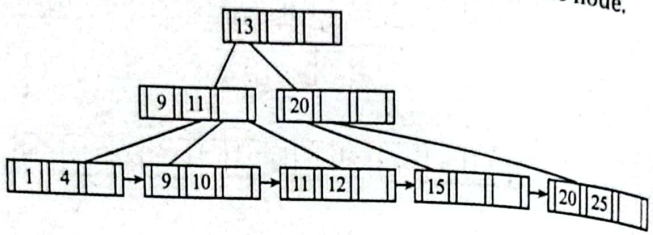
Split from 13 and move 13 up



To delete 16 from the B+ tree, we need to follow these steps:

Search for the key 16 in the tree.

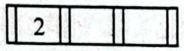
Since 16 is located in a leaf node, remove it from the node.



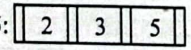
4. Construct a B+ tree for the following set of key values: (2,3,5,7,11,17,19,23,29,31) Assume that the tree is initially empty and values are added in ascending order where the pointer number is four. [2019 Fall]

ANS:

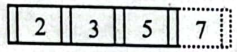
Insert 2:



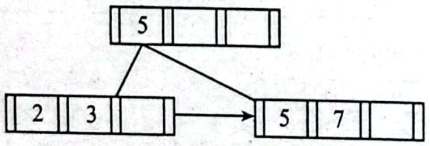
Insert 3 & 5:



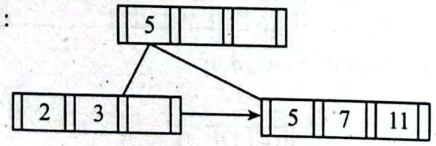
Insert 7:



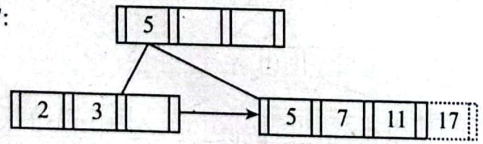
Split from 5 and move 5 up.



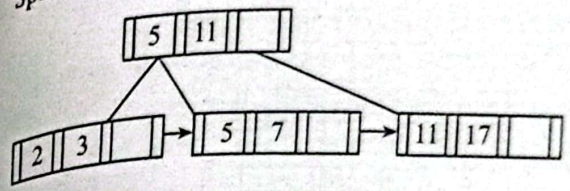
Insert 11:



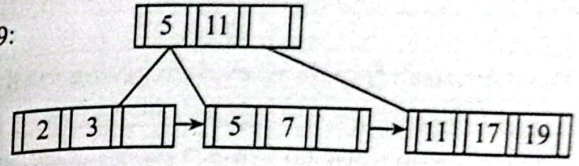
Insert 17:



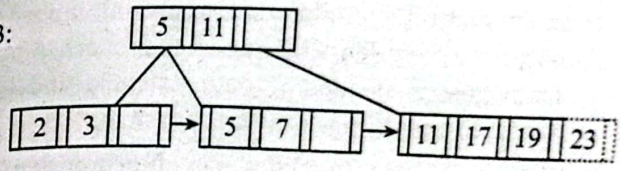
Split from 11 and move 11 up.



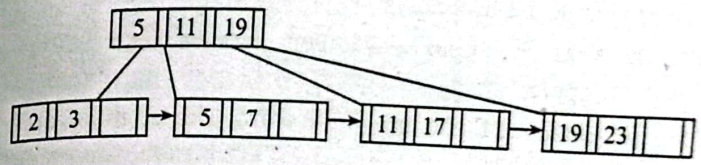
Insert 19:



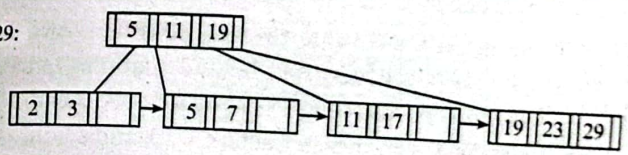
Insert 23:



Split from 19 and move 19 up.

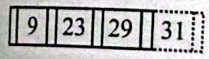


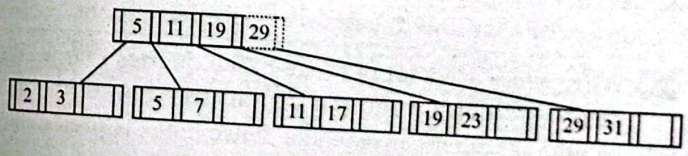
Insert 29:



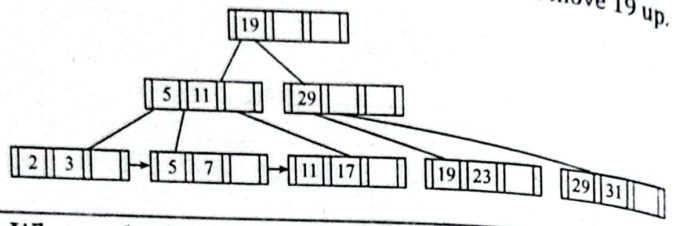
Insert 31:

31 will go to the leaf containing 19, 23 & 29 but its full. So split from 29 & move 29 up.





Also root 15, 10, 19 & 29 is full so split from 19 & move 19 up.



5. What are hash functions, explain giving example. [2016 Fall]

ANS: A hash function, also known as a hash value or hash code, is a mathematical operation that accepts an input (or "key") and outputs a fixed-length string of characters. A hash function's main goal is to identify data uniquely and make efficient retrieval and storage activities possible.

Let's consider a simplified hash function that calculates the hash value based on the length of a word. The steps for this example are as follows:

i. Take the input word: For instance, let's use the word "apple."

ii. Calculate the length of the word: The length of "apple" is 5 characters.

iii. Assign a fixed value to each possible word length: Let's assume we have a predefined mapping where the hash values for word lengths are as follows: 3 characters = 10, 4 characters = 20, 5 characters = 30, and so on.

iv. Generate the hash value: Based on the length of "apple" (5 characters), the hash value would be 30 according to the predefined mapping.

6. When is it preferred to use dense index rather than sparse index? Explain with suitable example. [2014 Fall]

ANS: When there are a lot of range-based queries that need to be completed quickly and the data distribution in the indexed column is rather consistent, a dense index is preferable over a sparse index.

Example:

Imagine a database table that stores information about books in a library. One of the indexed columns is "Publication Year," which indicates the year each book was published.

If the distribution of publication years in the database is relatively uniform, a dense index would be a suitable choice.

With a dense index on the "Publication Year" column, an index entry would be created for each book record in the table. This means that every book, regardless of its publication year, would have an entry in the index.

Now, let's suppose we need to execute a query to find all books published between the years 2000 and 2010. In this case, a dense index is advantageous because it covers the entire range of publication years. The index entries allow for efficient retrieval of the books falling within the specified range, as each entry in the index points directly to the corresponding book record.

On the other hand, if the distribution of publication years is uneven, with some years having a large number of books and others having very few, a sparse index may be more appropriate.

For instance, consider a scenario where the majority of books in the database were published in the last five years, but there are only a few books from earlier years. In this case, a sparse index on the "Publication Year" column would be useful. The sparse index would have entries only for the years with a significant number of books, skipping the entries for years with fewer books.

When executing a query to retrieve books from a specific year, the sparse index allows for efficient access to the relevant records by using the index entry for that particular year. This helps reduce the size of the index and improve query performance, as the index focuses on the years that have a higher density of data.

◆ ◆ ◆