

Storage and File Structure

We will discuss two topics:

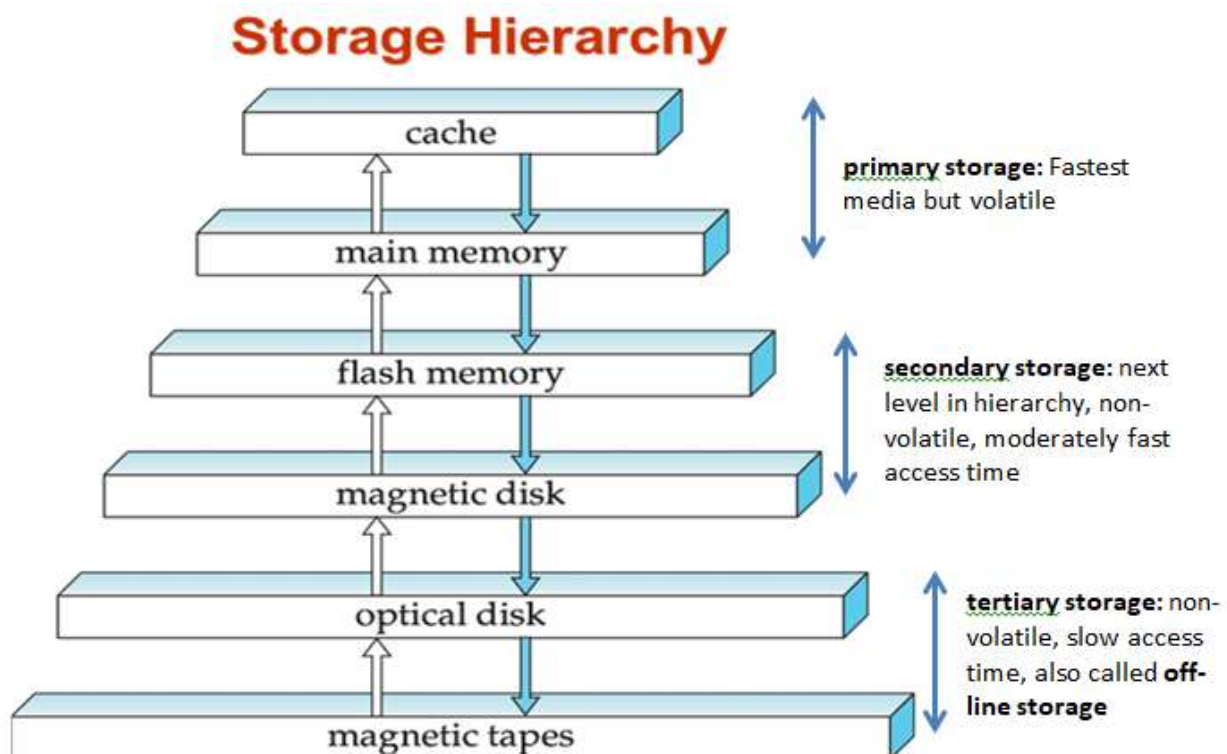
- Organization of databases in storage and
- The techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called **indexes**.

Storage of Databases

- Databases typically store large amounts of data in **hard disk** which is the primary choice for large databases. However, in future it may reside at different levels of the memory.
- Hence, it is important to study and understand the properties and characteristics of magnetic disks and the **data files** and **records** organization on disk for effective physical design of DB with acceptable performance.
- Usually, the DBMS has **several options available for organizing the data**. The process of **physical database design** involves choosing the most appropriate one that best suit the given application requirements from those options
- There are several primary file organizations like **heap file** (or unordered file), **sorted file** (or sequential file), **hashed file**, **B-trees** etc. We will discuss some of the file organizations and access technique for each organization.

First we will discuss briefly various storage devices available in computer and their hierarchy:

Storage Hierarchy



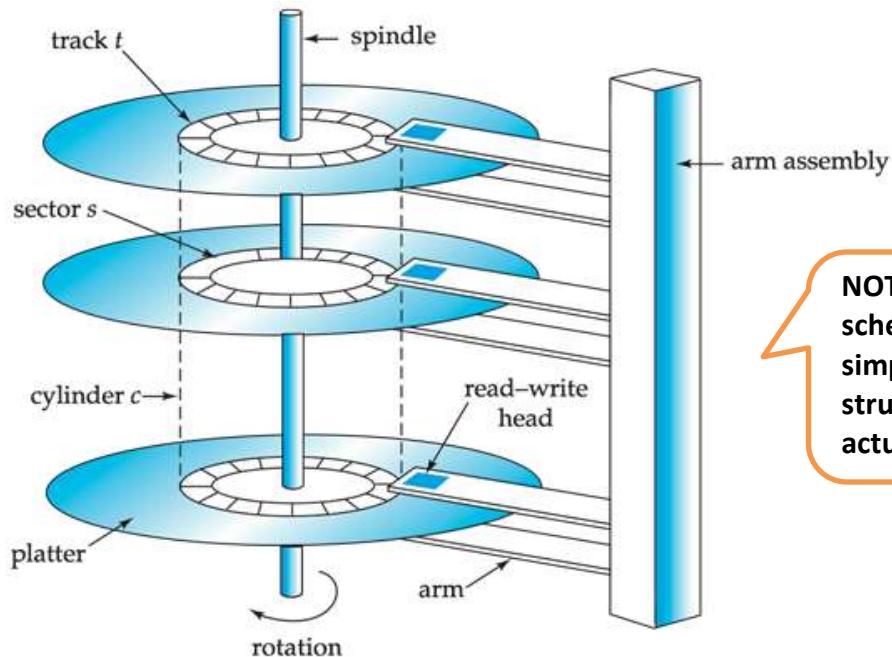
- **Cache** – fastest and most costly form of storage; volatile; managed by the system hardware.

- **Main memory** - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds) & volatile. Generally too small (or too expensive) to store the entire database.
- **Flash memory** -
 - Data survives power failure
 - Data can be written at a location only once, but location can be erased and written to again
 - ▶ Can support only a limited number (10K – 1M) of write/erase cycles.
 - ▶ Erasing of memory has to be done to an entire bank of memory
 - Reads are roughly as fast as main memory
 - But writes are slow (few microseconds), erase is slower
 - Widely used in embedded devices such as digital cameras, phones, and USB keys
- **Magnetic-disk** - Data is stored on spinning disk, and read/written magnetically and survives power failures and system crashes. It is the primary medium for the long-term storage of data; typically stores entire database. Data must be moved from disk to main memory for access, and written back for storage.
- **Optical storage**
 1. **non-volatile**, data is read optically from a spinning disk using a **laser**
 2. CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
 3. Blu-ray disks: 27 GB to 54 GB
 4. Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
 5. Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
 6. Reads and writes are slower than with magnetic disk
 7. **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Magnetic Hard Disk Mechanism

Different constituent parts of HD and their function discussed briefly.

- **Read-write head** : Positioned very close to the platter surface (almost touching it) and reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**. Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors** (size typically 512 bytes). A sector is **the smallest unit of data** that can be **read** or **written**.
- Typical **sectors per track**: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
 - 1) disk arm swings to position head on right track
 - 2) platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies : **multiple disk platters** on a single **spindle** (1 to 5 usually) and one head per platter, mounted on a common arm.
- **Cylinder** i consists of i^{th} track of all the platters



- **Disk controller** – interfaces between the computer system and the disk drive hardware.
 - 1) accepts high-level commands to read or write a sector
 - 2) initiates actions such as moving the disk arm to the right track and actually reading or writing the data
 - 3) Performs remapping of bad sectors

Optimization of Disk-Block Access

Disk access is optimized using following techniques:

- **Block** –A contiguous sequence of sectors from a single track . Typical block sizes today range from 4 to 16 kilobytes. Data is transferred between disk and main memory in blocks
- **Disk controller** uses a Disk-arm-scheduling algorithms to order pending accesses to tracks so that disk arm movement is minimized. One such algorithm known as **elevator algorithm** is used to determine the motion of the disk's arm and head in servicing read and write requests. This algorithm is named after the behavior of a building elevator, where the elevator continues to travel in its current direction (up or down) until empty, stopping only to let individuals off or to pick up new individuals heading in the same direction.

From an implementation perspective, the drive maintains a buffer of pending read/write requests, along with the associated cylinder number of the request. (Lower cylinder numbers generally indicate that the cylinder is closer to the spindle, and higher numbers indicate the cylinder is farther away.)

When a new request arrives while the drive is idle, the initial arm/head movement will be in the direction of the cylinder where the data is stored, either *in* or *out*. As additional requests arrive, requests are serviced only in **the current direction of arm movement until the arm reaches the**

edge of the disk. When this happens, the direction of the arm reverses, and the requests that were remaining in the opposite direction are serviced, and so on.

- **File organization** – To optimize block access we store related information on the same or nearby cylinders.
- **Non-volatile RAM:** battery backed up RAM or flash memory is used to speed up disk writes
 - First write blocks to a non-volatile RAM buffer immediately. Even if power fails, the data is safe and will be written to disk when power returns
 - Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
 - Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
 - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
 - Used exactly like nonvolatile RAM
 - Write to log disk is very fast since no seeks are required
 - No need for special hardware (NV-RAM)

File Organization

File organization determines how records are represented in a file structure. Various file organizations are used to store a relation in disk.

- A file consists of records. These records are mapped onto disk blocks. Block size is fixed but record size can vary.
 - Each file has a file header to contain a variety of information about the file.
 - In a relational database, tuples of distinct relations are generally of different sizes. Following are approaches for mapping the database to files :
 - fixed record length file to store one relation in one file.
 - variable record length file to store one relation in one file.
1. In RDBMS a particular relation (same record type) may need variable length records for several reasons :
 - One or more of the fields are of varying size (variable-length fields). For example, the NAME field of EMPLOYEE can be a varchar(50) field.
 - One or more of the fields may have multiple values for individual record
 - One or more of the fields are optional
 - The file contains records of different record types and hence of varying size (mixed file). This would occur if related records of different types were clustered (placed together) on disk blocks; for example, the GRADCREPORT records of a particular student may be placed following that STUDENT'S record.
 - Files of fixed length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case. Thus, we begin by

considering a file of fixed-length records.

Fixed-Length Records

<div>header</div>				As an example, let us consider a file of account records for bank database. Assume record length = 40 bytes.
record 0	A-102	Perryridge	400	<p>A simple approach to add a record one by one with fixed record length 40 bytes as shown in figure. However, there are two problems with this simple approach:</p> <ol style="list-style-type: none">1. It is difficult to delete a record. The space of the deleted record must be used while new record inserted, or we must have a way of marking deleted records so that they can be ignored.2. Unless the block size happens to be a multiple of 40 (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.
record 1	A-305	Round Hill	350	
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4	A-222	Redwood	700	
record 5	A-201	Perryridge	900	
record 6	A-217	Brighton	750	
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	
<div>Following approaches can be used to overcome the deletion problem.</div>				
record 0	A-102	Perryridge	400	<p>After deleting record 2 move all the records after that one by one to occupy the deleted space as shown in figure.</p> <p>Such an approach requires moving a large number of records.</p>
record 1	A-305	Round Hill	350	
record 3	A-101	Downtown	500	
record 4	A-222	Redwood	700	
record 5	A-201	Perryridge	900	
record 6	A-217	Brighton	750	
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	
record 0	A-102	Perryridge	400	
record 1	A-305	Round Hill	350	
record 8	A-218	Perryridge	700	
record 3	A-101	Downtown	500	
record 4	A-222	Redwood	700	
record 5	A-201	Perryridge	900	
record 6	A-217	Brighton	750	
record 7	A-110	Downtown	600	
In the above approach a simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. So we need to use file header . Here we need to store the address of the first record whose contents are deleted . The first deleted record will store the address of the second available record, and so on. The deleted records thus form a linked list , which is often referred to as a free list . The figure shows the free list, after records 1, 4, and 6 have been deleted.				

header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

- On insertion of a new record, we use the **record pointed 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**.
- For fixed-length records the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.
- So we need some alternative.

Variable-Length Records

For purposes of illustration we consider a different representation of the account information stored in the file, in which we use one variable-length record for each **branch name** and for **all the account information for that branch**.

Various ways to implement variable-length records are discussed below :

1. **Byte-String Representation** : Store each record as a string of consecutive 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	⊥				

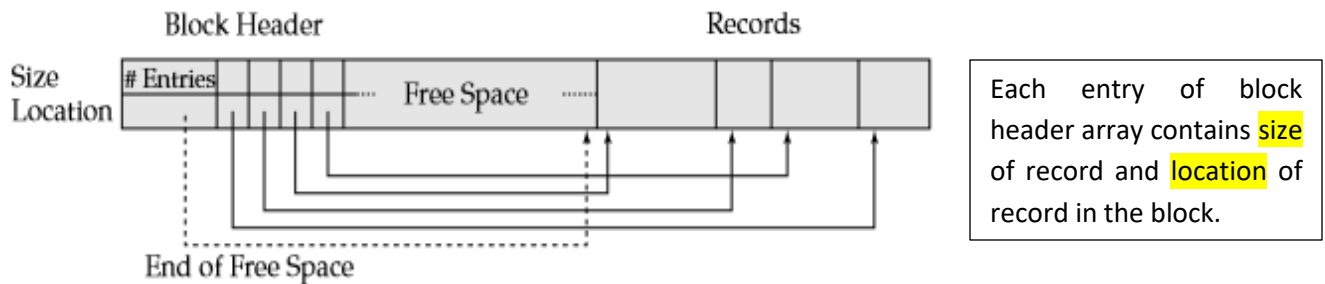
A simple method for implementing variable-length records is

- Either attach a special **end of-record (⊥)** symbol to the end of each record
- Or stores the **record length at the beginning of each record**.

Some disadvantages:

- It is not easy to reuse space occupied formerly by a deleted record. Although techniques exist to manage insertion and deletion, they lead to a large number of small fragments of disk storage that are wasted.
- There is no space, in general, for **records to grow longer**. If a variable-length record becomes longer, it must be moved—movement is costly if pointers to the record are stored elsewhere in the database (e.g., in indices, or in other records), since the pointers must be located and updated.

Thus, the basic byte-string representation is not usually used for implementing variable-length records. However, a modified form of the byte-string representation, called the **slotted-page structure** (as shown in figure below), is commonly used for organizing **records within a single block**.



- There is a **header** at the beginning of **each block**, containing the following information:
 - The number of record entries in the header
 - The end of free space in the block
 - An **array** whose entries contain the **location** and **size** of each record.

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**.

Record insert : space is allocated for it at the end of free space, and an entry containing its **size** and **location** is added to the header.

Record Delete : 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**: A typical value is **4 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.

2. **Fixed-Length Representation** for variable-length records

Another way to implement variable-length records efficiently in a file system is to use one or more fixed-length records to represent one variable-length record. There are two ways of doing this:

Reserved space	List representation
If there is a maximum record length that is never exceeded, we can use fixed-length records of that length. Unused space (for records shorter than the maximum space) is filled with a special null, or end-of-record, symbol .	We can represent variable-length records by lists of fixed length records, chained together by pointers.


Following figure assumes a maximum of three accounts per branch.

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	⊥	⊥	⊥	⊥

Those branches with fewer than three accounts (for example, Round Hill) have records with null fields.

The reserved-space method is useful when most records have a length close to the maximum. Otherwise, a significant amount of space may be wasted.

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	

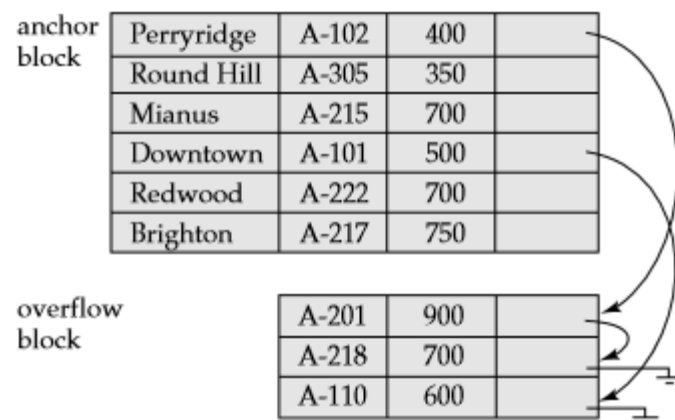


As shown in above figure pointers are used to chain together all records pertaining to the same branch. Whereas for fixed-length record we use pointers to chain together only deleted records.

A disadvantage to the above structure is that there are space in all records except the first in a chain. The first record needs to have the branch-name value, but subsequent records do not. This wasted space is significant, since we expect, in practice, that each branch has a large number of accounts. To deal with this problem, we allow two kinds of blocks in our file:

- **Anchor block**, which contains the first record of a chain
- **Overflow block**, which contains records other than those that are the first record of a chain

Thus, all records within a block have the same length, even though not all records in the file have the same length. Following figure shows this file structure.



Anchor-block and overflow-block structures.

Organization of Records in Files

An instance of a relation is a set of records. Given a set of records, the next question is how to organize them in a file. Several of the possible ways of organizing records in files are:

2. **Heap file organization.** Any record can be placed anywhere in the file where there is space for the record. There is **no ordering** of records. Typically, **there is a single file for each relation.**
3. **Sequential file organization.** Records are stored in **sequential order**, according to the value of a "search key" of each record.
4. **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 should be placed. This is **closely related to the indexing structures** described later.

Here we are discussing only the **sequential** file organization and **clustering** file organization. Others will be discussed in index structure.

Generally, a separate file is used to store the records of each relation. However, in a **clustering** file organization, records of **several different relations** are stored in the same file; further, related records of the different relations are stored on the same block, so that one I/O operation fetches related records from all the relations. For example, records of the two relations can be considered to be related if they would match in a join of the two relations.

Sequential File Organization

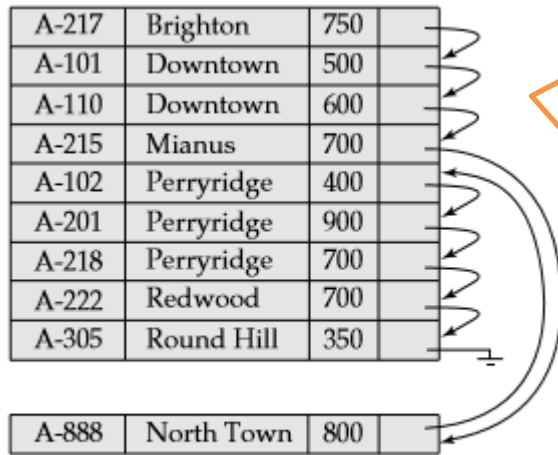
A sequential file is designed for efficient processing of records in **sorted order** based on some search-key. A search key is any attribute or set of attributes; it need not be the primary key, or even a superkey. To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the **next record in search-key order**. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

It shows a sequential file of account records. Here branch name as the search key. The sequential file organization allows records to be read in sorted order. It is useful for display purposes, 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 operation.

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

- Locate the record in the file that comes before the record to be inserted in search-key order.
- If there is a free record (that is, space left after a deletion) within the same block as this record, insert the 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.



This Figure shows the above file after the insertion of the record (North Town, A-888, 800). This structure allows fast insertion of new records, but it will not **match the physical order** of the records. If relatively few records need to be stored in overflow blocks, this approach works well.

Sequential file after an insertion.

Eventually, after insertion of large records, **the correspondence between search-key order and physical order may be totally lost**, in which case sequential processing will become much **less efficient**.

At this point, the file should be reorganized so that it is **once again physically in sequential order**. Such reorganizations are **costly**, and must be done during times when the system load is low. The frequency with which reorganizations are needed depends on the frequency of insertion of new records.

Clustering File Organization

Many relational-database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides.

A clustering file organization is a file organization that stores **related records of two or more relations in each block**. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.

Many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, **one large operating-system file is allocated to the database system**. The **database system stores all relations in this one file**, and manages the file itself.

To see the advantage of storing many relations in one file, consider the following SQL query for the bank database:

```
select account-number, customer-name, customer-street, customer-city
from depositor, customer
where depositor.customer-name= customer.customer-name
```

This query computes a join of the depositor and customer relations. Thus, for each tuple of depositor, the system must locate the **customer** tuples with the same value for customer-name. Regardless of how these records are located, however, they **need to be transferred from disk into main memory**. In the **worst case**, each record will reside on a different block, forcing us to do one block read for each record required by the query.

customer-name	account-number
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

depositor relation

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Clustering File Structure

This file organization allows us to read records that would satisfy the join condition by using one block read. So query processing is more efficiently.

customer-name	customer-street	customer-city
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

customer relation

In this clustering, the **depositor tuples** are stored near the **customer tuple** for the corresponding customer name. When a tuple of the customer relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding depositor tuples are stored on the disk near the customer tuple, the block containing the customer tuple contains tuples of the depositor relation needed to process the query. If a customer has so many accounts that the depositor records do not fit in one block, the remaining records appear on nearby blocks.

Clustering can enhanced **processing of a particular join** (depositor customer), but it **results in slowing processing of other types of query**. For example, **select * from customer** requires more block accesses than it did in the scheme under which we stored each relation in a separate file. Instead of several customer records appearing in one block, each record is located in a distinct block.

Indeed, simply finding all the customer records is not possible without some additional structure. To locate all tuples of the customer relation in the structure of above figure we need to chain together all the records of that relation using pointers, as shown below.

Hayes	Main	Brooklyn	
Hayes	A-102		
Hayes	A-220		
Hayes	A-503		
Turner	Putnam	Stamford	
Turner	A-305		

Careful use of clustering can produce significant performance gains in query processing.

Clustering file structure with pointer chains.

Index

Basic Concepts

An index for a file in a DBMS works in much the same way as the index of a textbook.

- Index at the back of book contains topic (specified by a word or a phrase) in the textbook and the page nos. where it appears.
- We can search for the topic in the index page, find the pages where it occurs, and then read the pages to find the information we are looking for.
- The words in the index are in sorted order, making it easy to find the word we are looking for. Moreover, the index is much smaller than the book, further reducing the effort.

Database system indices play the same role as book indices. For example, to retrieve an *account* record given the account number, the database system would look up an index to find on which **disk block** the corresponding record resides, and then fetch the disk block, to get the *account* record.

We will discuss several indexing techniques and their advantages and disadvantages.

Indexing in Databases

- An index or database index is a **data structure** which is used to quickly locate and access the data in a database table.
- It helps to optimize performance of a database by minimizing the number of disk accesses required when a query is processed.

Indexes are created using some database columns and it has two pieces of information :

<table><tr><td>Search Key</td><td>Data Reference</td></tr></table> <p>Structure of an index</p> <p>An attribute or set of attributes used to look up records in a file is called a search key.</p>	Search Key	Data Reference	<ul style="list-style-type: none">• The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).• The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.
Search Key	Data Reference		

There are two basic kinds of indices:

1. **Ordered** indices : Based on a **sorted ordering of the values**.
2. **Hash** indices : Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a **hash function**.
 - The target address space is made of **buckets**, each of which holds **multiple records**. A bucket is either **one disk block or a cluster of contiguous blocks**.

- The hashing function maps a key into a **relative bucket number**, rather than assign an absolute block address to the bucket. A table maintained in the **file header** converts the bucket number into the corresponding disk block address.

There is no comparison between both the techniques, it depends on the database application on which it is being applied. Each technique must be evaluated on the basis of following factors:

- **Access Types:** e.g. value based search, range access, etc.
- **Access Time:** Time to find particular data element or set of elements.
- **Insertion Time:** Time taken to find the appropriate space and insert a new data + update the index structure
- **Deletion Time:** Time taken to find an item and delete it as well as update the index structure + update the index structure
- **Space Overhead:** Additional space required by the index.

Advantage & Disadvantage of Index

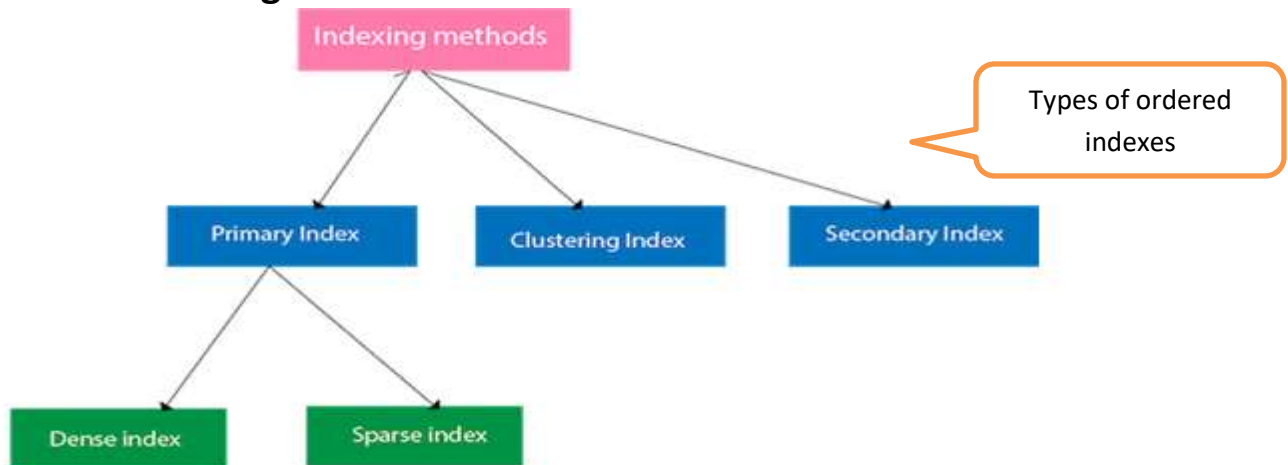
- Index makes search operation perform very fast.

Suppose a table has a 100 rows of data, each row size = 20 bytes and there is no index file. If you read the record number 100, DBMS read each and every row and after reading $99 \times 20 = 1980$ bytes it will find record number 100.

If we have an index, the search for record number 100 starts by reading the index file, not from the table. The index, containing only two columns, may be just 4 bytes wide in each of its rows. After reading only $99 \times 4 = 396$ bytes of data from the index the management system finds an entry for record number 100, reads the address of the disk block where record number 100 is stored and directly points at the record in the physical storage device. The result is a much quicker access to the record (a speed advantage of 1980:396).

- The only minor disadvantage of using index is that it takes up a **little more space** than the main table. Additionally, index needs to be **updated periodically for insertion or deletion** of records in the main table.

Ordered Indexing Methods



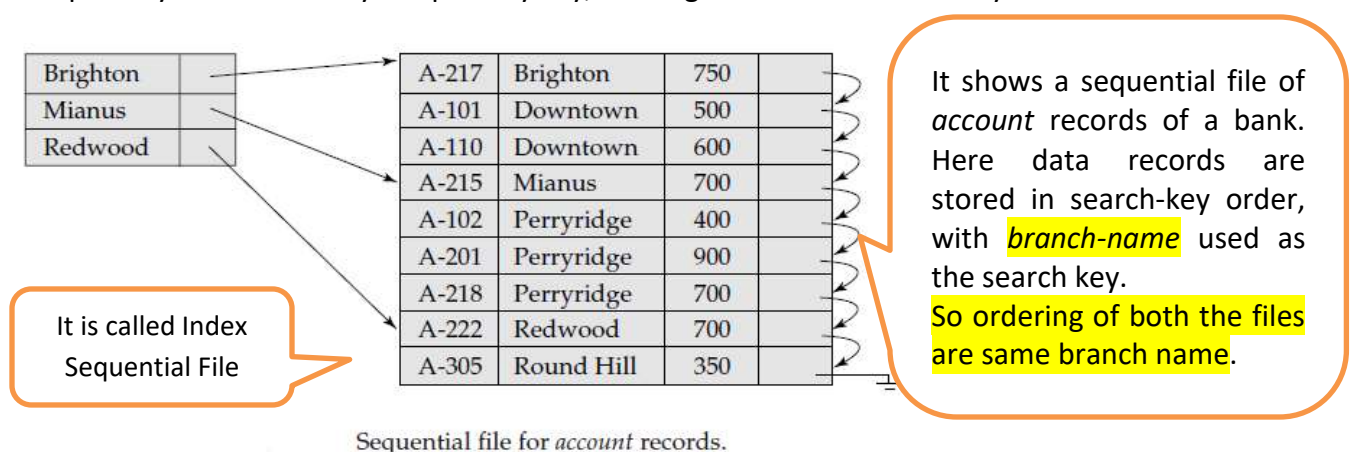
- The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value.
- The values in the index are ordered so that we can do a binary search on the index.
- The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient.

There are several types of ordered indexes.

Primary Index

- Primary Index requires the rows in data blocks to be ordered on the index key or search key.
- So apart from the index entries being themselves sorted in the index block, primary index also enforces an ordering of rows in the data blocks.

(The term *primary index* is sometimes used to mean an index on a primary key. However, such usage is nonstandard and should be avoided.) Primary indices are also called **clustering indices**. The search key of a primary index is usually the primary key, although that is not necessarily so.

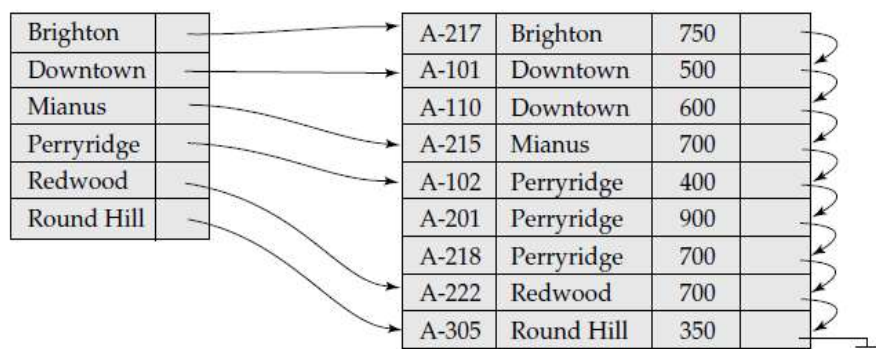


If the **ordering field** in data file is **not a key field**, i.e. more than one record in the file can have the same value for the ordering field, we can use another type of index, called a **clustering** index. Notice that a file can have **at most one physical ordering field**, so it can have **at most one primary index** or **one clustering index, but not both**.

Two possible index structures of primary index are as follows.

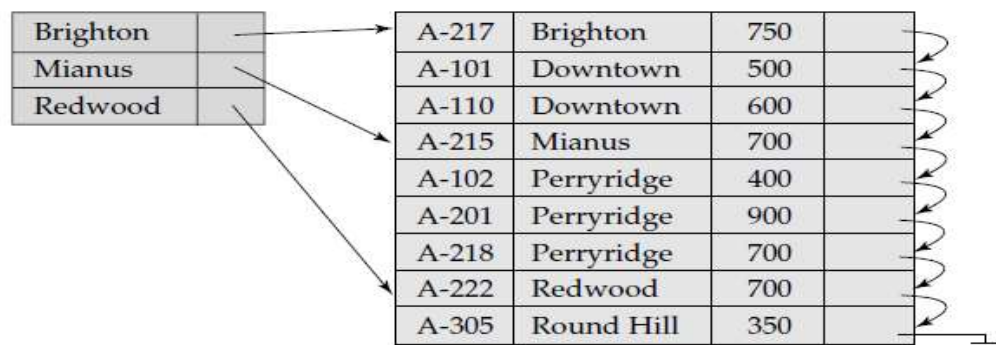
- **Dense index:** An index record **appears for every search-key value** in the file. In a dense primary index, the index record contains the search-key value and a pointer to the **first data record with that search-key value**. The rest of the records with the same search key-value would be stored sequentially after the first record, since, because the index is a primary one, records are sorted on the same search key.

Dense index implementations may store a list of pointers to all records with the same search-key value; **doing so is not essential for primary indices**.



Dense index.

- **Sparse index:** An index record appears for only **some of the search-key values**. Search key pointers to the first data record with that search-key value. **To locate a record with a search key, we find the index entry which is less than or equal to that search-key**. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.



Sparse index.

Suppose that we are looking up records for the **Downtown** branch. There is no index record for that search key. Since the last entry (in alphabetic order) before "Downtown" is "Brighton," we follow that pointer. We process this record, and follow the pointer in that record to locate the next record in

search-key (*branch-name*) order. We continue processing records until we encounter a record for a branch other than Downtown in data file.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require **less space and they impose less maintenance overhead for insertions and deletions.**

There is a trade-off that the system designer must make between access time and space overhead. A good compromise is to have **a sparse index with one index entry per block.** Note that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. **Once we have brought in the block, the time to scan the entire block is negligible.** Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an **overflow** block, we minimize block accesses while keeping the size of the index (and thus, our space overhead) as small as possible.

EXAMPLE: Comparing access time in a data file with primary index and without primary index

Suppose we have an **ordered data file** with **$r = 30,000$** records stored on a disk with block size **$B = 1024$** . File records are of **fixed size and are unspawnd**, with **record length $R = 100$ bytes.**

Without primary Index	With Primary index
<p>The blocking factor for the file $bfr = (B/R) \sim 10$ records per block.</p> <p>The number of blocks needed for the file is $(b) = (r/bfr) = (30,000/10) = 3000$ blocks.</p> <p>As the file is ordered to access the record we can use a binary search on the data file which would need $(\log_2 3000) = 11.55075 \sim$ 12 block accesses.</p>	<p>Now suppose that the ordering key field size $(V) = 9$ bytes and a block pointer takes $P = 6$ bytes, and we have constructed a primary index for the file.</p> <p>The size of each index entry $= (9 + 6) = 15$ bytes, Blocking factor for the index $= 1024/15 = 68$ entries per block.</p> <p>The total number of index entries = total number of blocks in the data file, which is 3000.</p> <p>The no. of index blocks $= 3000/68 = 45$ blocks. To perform a binary search on the index file would need $(\log_2 45) = 5.4 \sim 6$ block accesses.</p> <p>To search for a record using the index, we need one additional block access to the data file for a total of $6 + 1 = 7$ block accesses. So an improvement over binary search on the data file, which required 12 block accesses.</p>

A major problem with a primary index

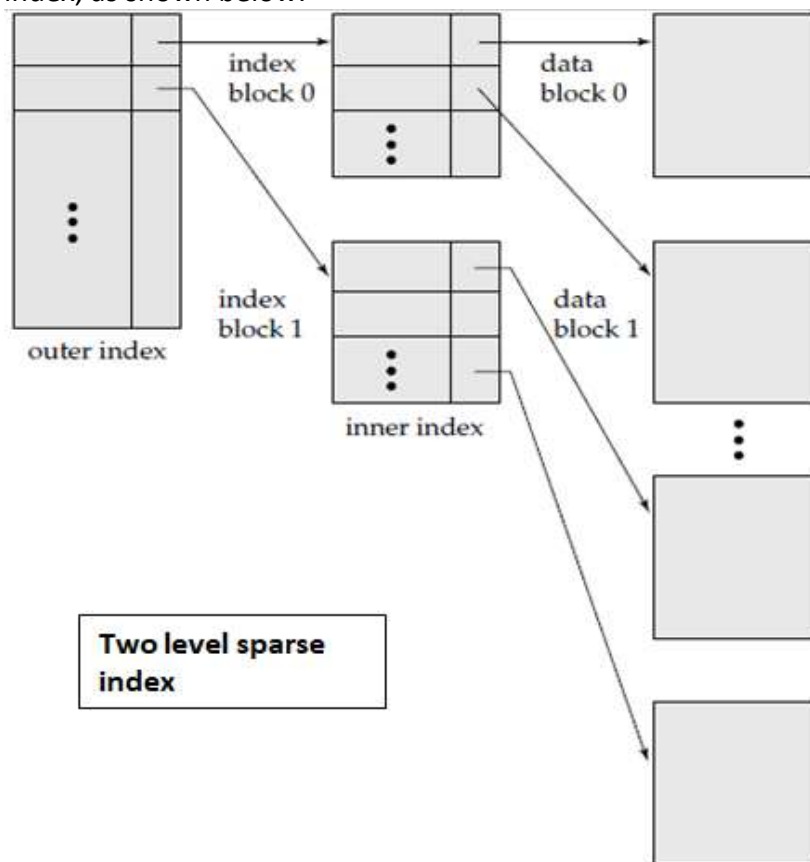
Insertion and deletion of records in an order list has some problems. With a primary index, the problem is compounded because, if we attempt to insert a record in its correct position in the data file,

we have to not only move records to make space for the new record but also change some index entries, **since moving records will change the anchor records of some blocks.**

Using **an unordered overflow file** can reduce this problem. Another possibility is to use a **linked list of overflow records** for each block in the data file. This is similar to the method of dealing with overflow records described with hashing. (we are not discussing here)

- **Multilevel Indices**

Even if we use a sparse index, sometimes index file size becomes very large for big data file. So access time may be long although we are using binary search. Note that, if overflow blocks have been used, binary search will not be possible. In that case, a sequential search is typically used that will take even longer. **Thus, the process of searching a large index may be costly.** To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse index on the primary index, as shown below.



To locate a record, we first use **binary search on the outer index** to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the **inner index**. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire.

The pointer in this record **points to the block of the file** that contains the record for which we are looking.

Using the two levels of indexing, we have read only one index block, rather than the seven we read with binary search, if **we assume that the outer index is already in main memory.**

If our file is extremely large, even the outer index may **grow too large to fit in main memory.** In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary

search. Each level of index could correspond to a unit of physical storage. Thus, we may have indices at the track, cylinder, and disk levels. Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing.

- **Secondary Indices**

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.

Secondary indexes are sorted w.r.t. the search key but the DataFile IS NOT sorted w.r.t. the Secondary Index Search Key!

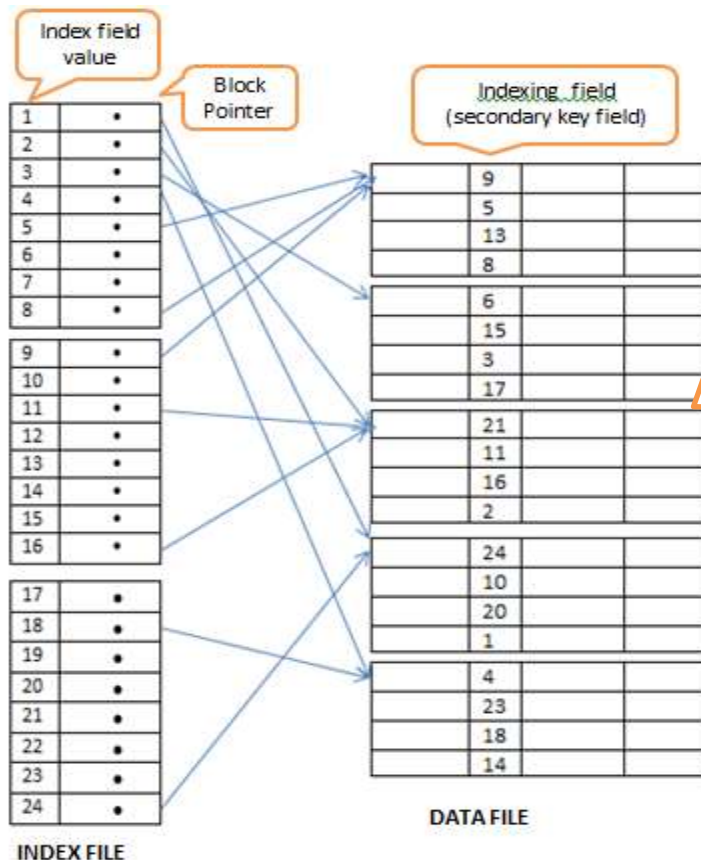
Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file. Secondary Indexes are always Dense: Sparse secondary indexes make no sense!

A secondary index on a candidate key looks just like a dense primary index, except that the records pointed to by successive values in the index are not stored sequentially.

The index is an ordered file with two fields. The first field is an indexing field (some non- ordering field of the data file). 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.

A primary index is an index on a file sorted w.r.t. the search key. Then a primary index “controls” the storage of records in the data file. Since a file can have at most one physical order then it can have at most one primary index. But Secondary Index does not have any impact on how the rows are actually organized in data blocks. They can be in any order. The only ordering is w.r.t the index key in index blocks.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries.



The diagram shows how the index entries in the index blocks (left side) contains pointers (these are row locators in database terminology) to corresponding rows in data blocks (right side). The data blocks do not have rows **sorted** on the index key.

Each secondary index contains the pointer to the block where the record exists.

Once the appropriate block is transferred to main memory, a **search for the desired record within the block can be carried out.**

B-tree

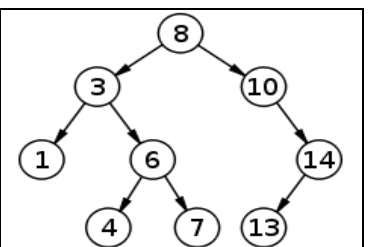
In a multi-level index insertion and deletion of new index entries is a severe problem because every level of the index is an **ordered file**. **It can be very expensive to keep the index in sorted order.** So we need a way to make insertions and deletions to indexes that will not require **massive reorganization**. B-tree is a suitable alternative.

- B-Tree was invented by two researchers at Boeing, R. Bayer and E. McCreight in 1972
- By 1979 B-trees were the **"de facto, the standard"** organization for indexes in a database system"

Using B-trees as dynamic multi-level indexes

A **search tree** is a tree data structure used for locating specific values from within a set. In order for a tree to function as a search tree, the **key for each node must be greater than any keys in subtrees on the left and less than any keys in subtrees on the right.**

In search trees like **binary search tree**, AVL Tree, Red-Black tree, etc., **every node** contains only **one value (key)** and a maximum of two children.



B-Tree can be defined as follows...

- A **B-tree** is a **self-balancing tree** data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in **logarithmic time**.
- **Self-Balancing Binary Search Trees** are **height-balanced** binary search trees that automatically keeps **height as small as possible** when insertion and deletion operations are performed on tree. The height is typically maintained in **order of $\log n$** so that all operations take $O(\log n)$ time on average.
- Here, the **number of keys in a node** and **number of children for a node** depends on the **order of B-Tree**. Every B-Tree has an order.
- The height of the tree is automatically adjusted on each update in order to keep it **evenly balanced**. B-tree also keeps the data sorted by storing it in a specific order, the **lowest** value being on the **left** and the **highest** value on the **right**.
- The B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in **databases** and **file systems**.
- Each internal node of a B-tree contains a number of **keys**. The keys act as **separation values** which divide its **subtrees**. For example, if an internal node has **3 child nodes** (or subtrees) then it must have **2 keys: a_1 and a_2** .
- All values in the **leftmost subtree** will be less than a_1 , all values in the **middle subtree** will be between a_1 and a_2 , and all values in the **rightmost subtree** will be **greater than a_2** .

Time complexity in big O notation

Algorithm Average Worst case

Space $O(n)$ $O(n)$

Search $O(\log n)$ $O(\log n)$

Insert $O(\log n)$ $O(\log n)$

Delete $O(\log n)$ $O(\log n)$

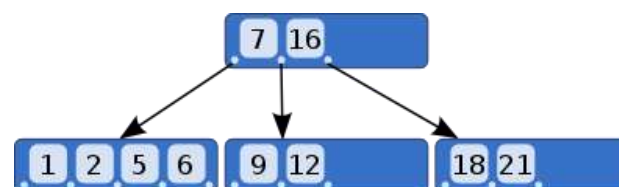
In following visual representation left node contains all keys less than 7, middle one for keys between 7 to 16, right node contains keys > 16

- A B Tree can have nodes with different number of children under each node.
- **Degree** represents the **minimum number of children** possible (**lower bound**) under a node (except for the root).
- **Order** represents the **upper bound** on the number of children. i.e. the maximum number possible.
- The maximum number of access operations required to reach the desired record is called the **depth**.
- When **order** = **depth** then it is called a **balanced** tree.
- When they are not same it is called **unbalanced/ asymmetrical** structure.

B-tree of **order m** satisfies following properties :

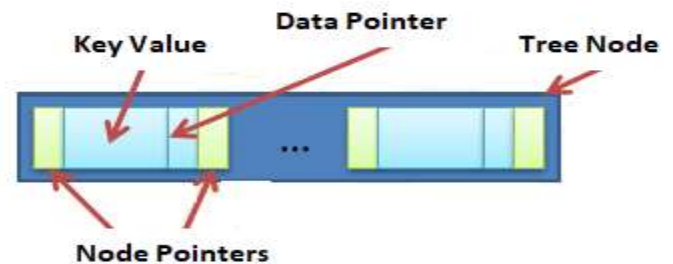
1. Each node has **at most m children**
2. Each internal node has at least $m/2$ children
3. Root has at least 2 children if it is not leaf
4. A non-leaf node with k children has (k-1) keys
5. All leaves appear in same level

The visual representation



- In B-trees, internal (non-leaf) nodes can have a **variable number of child nodes** within some pre-defined range.
- When data is inserted or removed from a node, its **number of child nodes changes**. In order to maintain the pre-defined range, **internal nodes may be joined or split**.
- Because a range of child nodes is permitted, B-trees do not need **re-balancing as frequently as other self-balancing search trees**, but may **waste some space**, since nodes are not entirely full.

Basic structure of a B-tree node – Nodes contain **key** values and respective **data (block) pointers** to the actual data records – Additionally, there are node pointers for the **left, resp. right interval** around a key value

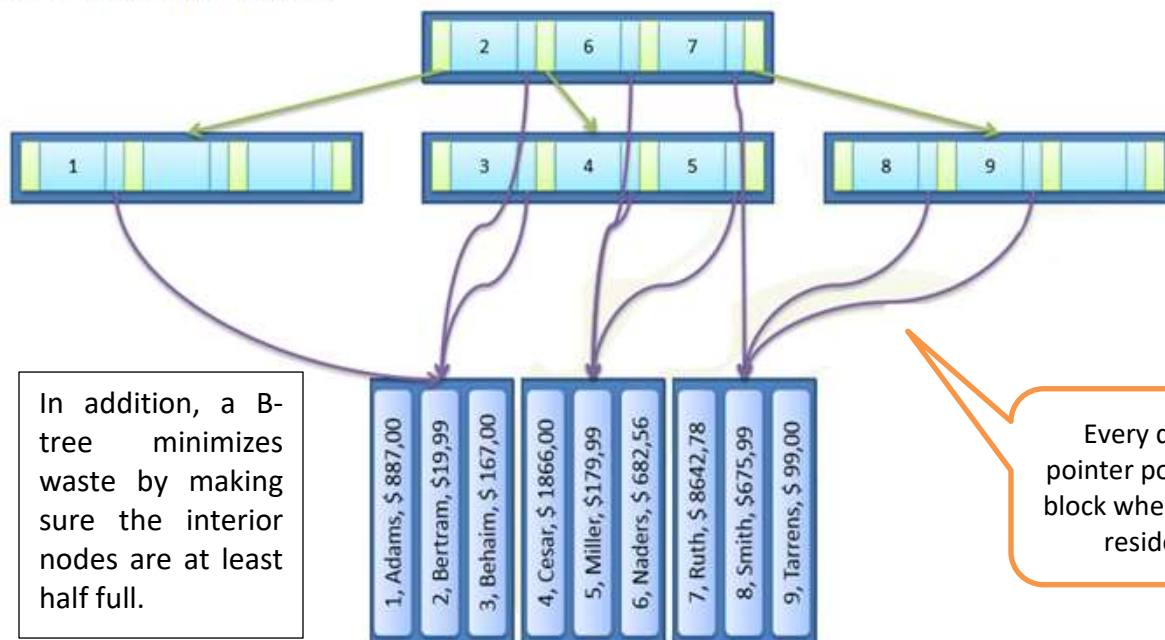


Advantages of B-tree usage for databases

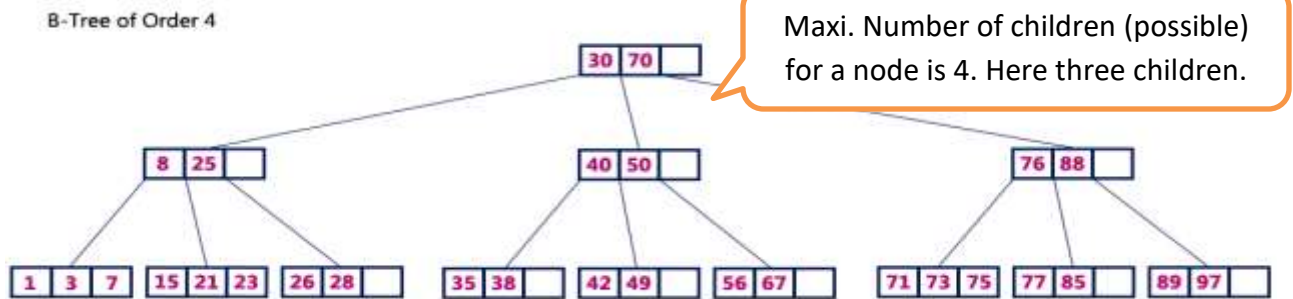
The B-tree uses all of the ideas described above. In particular, a B-tree:

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks
- speed insertions and deletions
- keeps the index balanced with a recursive algorithm

B-Tree as Primary Index



Example



Operations on a B-Tree

The operations are performed on a B-Tree are **Search, Insertion, Deletion**. We will only consider search and insert operation here.

Search

The search operation in B-Tree is similar to the search operation in Binary Search Tree. The search operation is performed as follows.

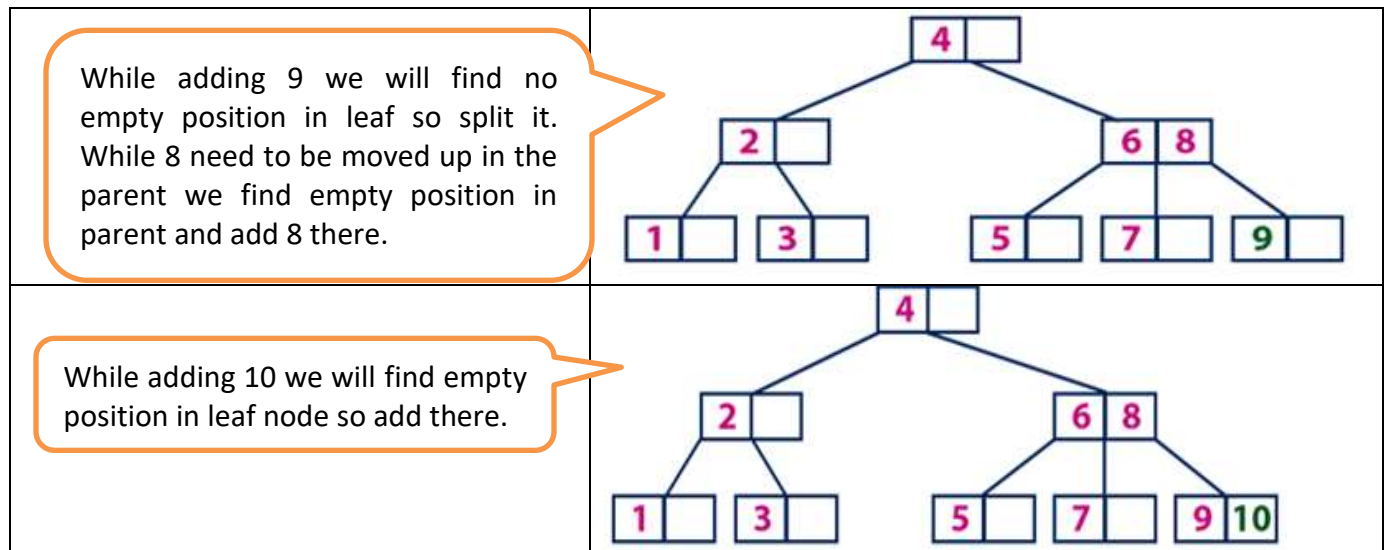
- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with **first key value of root node** in the tree.
- **Step 3** - If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4** - If both are not matched, then check whether search element is **smaller or larger** than that key value.
- **Step 5** - If search element is smaller, then continue the search process in **left subtree**.
- **Step 6** - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7** - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion

In a B-Tree, a new element must be **added only at the leaf node**. That means, the new keyValue is always attached to the leaf node only. Let us understand the insertion operation by constructing a **B-Tree of Order 3** by inserting numbers from **1 to 10**.

Step 1 - Check whether tree is Empty.	
Step 2 - If tree is Empty , then create a new node with new key value and insert it into the tree as a root node.	<div><div>1</div><div></div></div> <p>As 1 is the first key insert it into new node.</p>

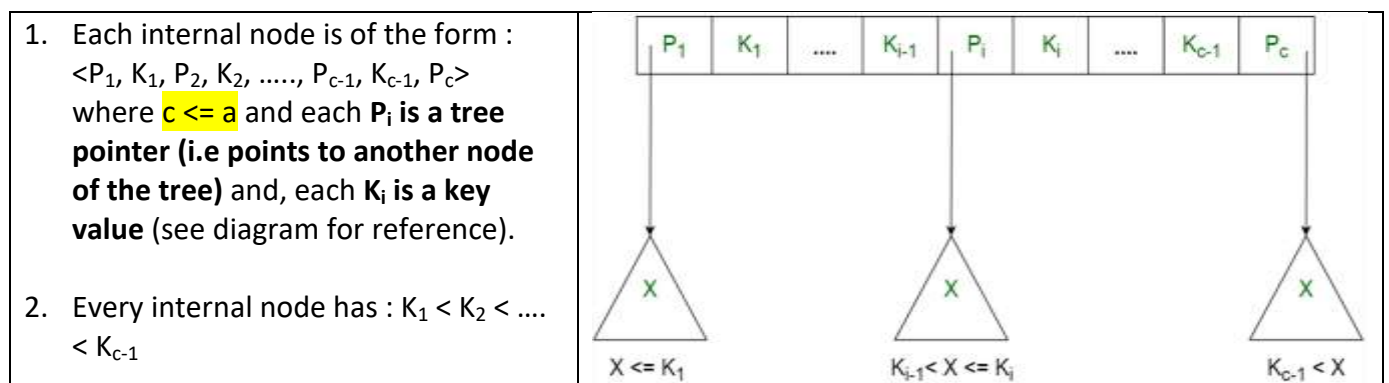
<p>Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.</p>	<p>While adding 2 we will get the above leaf node which is also the root not.</p>
<p>Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.</p>	<div data-bbox="776 352 906 409" data-label="Diagram"> </div> <p>Element 2 is added in the existing leaf node as it has empty position.</p>
<p>Step 5 - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.</p>	<div data-bbox="776 508 1396 646" data-label="Diagram"> </div> <p>While adding 3 we will find the leaf node has no vacant position so we split and add new node as shown above.</p>
<p>Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.</p>	<div data-bbox="776 777 1039 924" data-label="Diagram"> </div> <div data-bbox="1071 777 1526 913" data-label="Text"> <p>While adding 4 we will find empty position in the leaf node. So add it to that node.</p> </div>
<p>While adding 5 we will find no empty position in leaf node so again split it and add.</p>	<div data-bbox="776 966 1510 1134" data-label="Diagram"> </div>
<p>While adding 6 we will find empty position in leaf node so add there.</p>	<div data-bbox="776 1155 1136 1302" data-label="Diagram"> </div>
<p>While adding 7 we will find no empty position in leaf so split it. While 6 need to be moved up in the parent we find no empty position in parent, so we need to split the parent. In this step height will increase by one.</p>	<div data-bbox="776 1316 1510 1554" data-label="Diagram"> </div>
<p>While adding 8 we will find empty position in leaf node so add there.</p>	<div data-bbox="776 1568 1291 1795" data-label="Diagram"> </div>



B+Tree

- In order, to implement **dynamic multilevel indexing**, B-tree and B+ tree are generally employed.
- The **drawback of B-tree** : it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. So no of entries that can be packed into a node of a B-tree is reduced, resulting increase in the number of levels in the B-tree, hence **increasing the search time of a record**.
- B+ tree eliminates the above drawback by storing **data pointers** only at the leaf nodes of the tree. Thus, the structure of **leaf nodes of a B+ tree** is quite different from the structure of internal nodes of the B+ tree.
- 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**

The structure of the internal nodes of a B+ tree of order 'a' is as follows:



- For each search field values 'X' in the sub-tree pointed at by P_i , the following condition holds :
 $K_{i-1} < X \leq K_i$, for $1 < i < c$ and, $K_{i-1} < X$, for $i = c$
- Each internal nodes has at most 'a' tree pointers (as a is the order of tree)
- The root node has, at least two tree pointers, while the other internal nodes have at least $\lceil a/2 \rceil$ tree pointers each.
- If any internal node has 'c' pointers, $c \leq a$, then it has 'c - 1' key values.

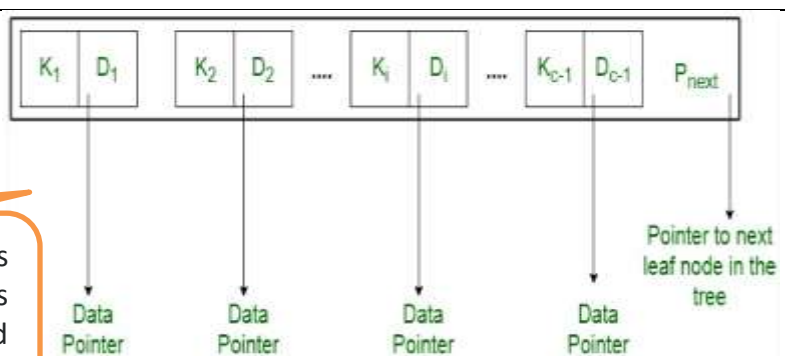
The structure of the leaf nodes of a B+ tree of order 'b' is as follows:

- Each leaf node is of the form :
 $\langle \langle K_1, D_1 \rangle, \langle K_2, D_2 \rangle, \dots, \langle K_{c-1}, D_{c-1} \rangle, P_{next} \rangle$

where $c \leq b$ and each D_i is a data pointer (i.e points to actual record in the disk whose key value is K_i or to a disk file block containing that record) and, each K_i is a key value and, P_{next} points to next leaf node in the B+ tree (see diagram below).

- Every leaf node has : $K_1 < K_2 < \dots < K_{c-1}$, $c \leq b$
- Each leaf node has at least $\lceil b/2 \rceil$ values.
- All leaf nodes are at same level.

The last pointer present in a leaf node is not considered a children since it links one leaf to another (is not a record pointer).

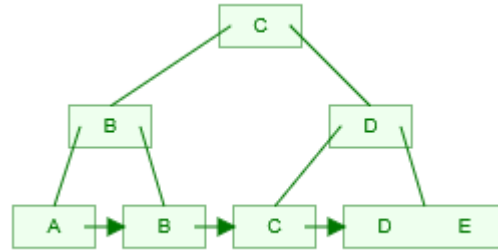
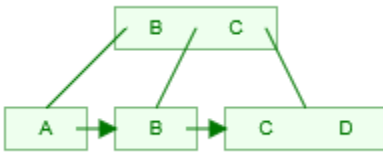


From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

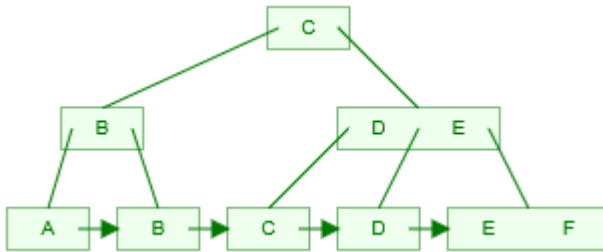
Let's understand the creation of B+ tree by inserting keys one by one

Say we want to add keys A to M in a B+ tree of order 3

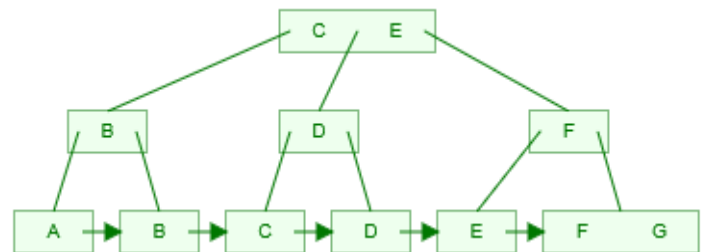
<p>Add A : a node will be allocated and A is added there.</p> <p>Order 3</p>	<p>Add B : there is empty place in that node so B will be added there.</p>	<p>Add C : there is no empty place in that node so new node added and B is promoted as parent node and C is added in leaf node after B.</p>
<p>Add D : No empty place in leaf node, C is promoted in parent node and D is added in leaf.</p>	<p>Add E : No empty place in parent node, D is promoted, but no empty place in parent node. Split the node and promote C to next higher parent.</p>	



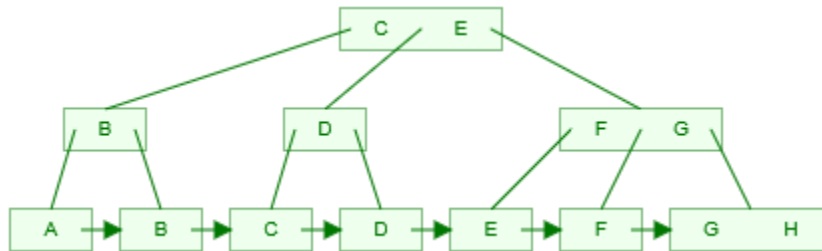
Add F : No place in leaf split node and promote E to parent.



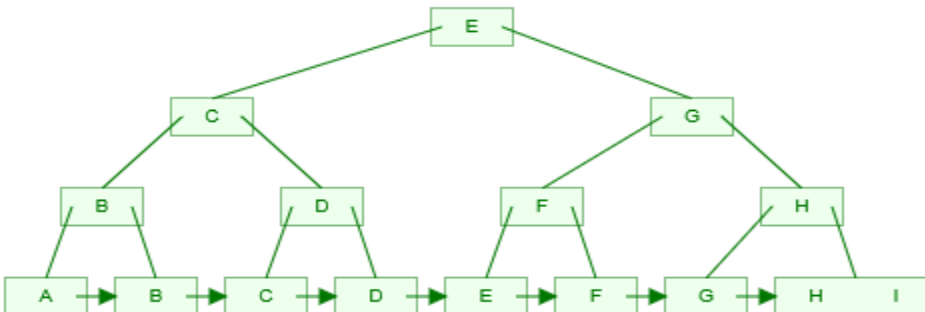
Add G : No empty place in leaf split and promote F (as it is middle value). But higher level parent has also no empty place, split it and promote E to next higher level.



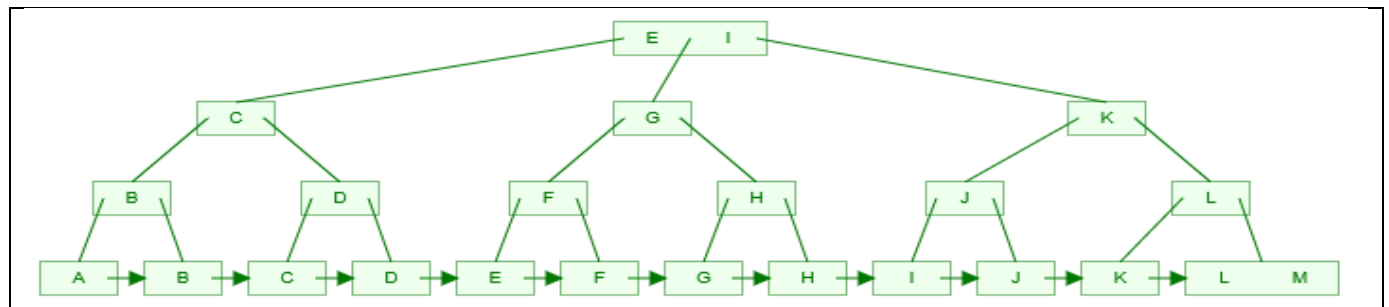
Add H : No empty place in leaf split and promote G (as it is middle value) in higher level parent.



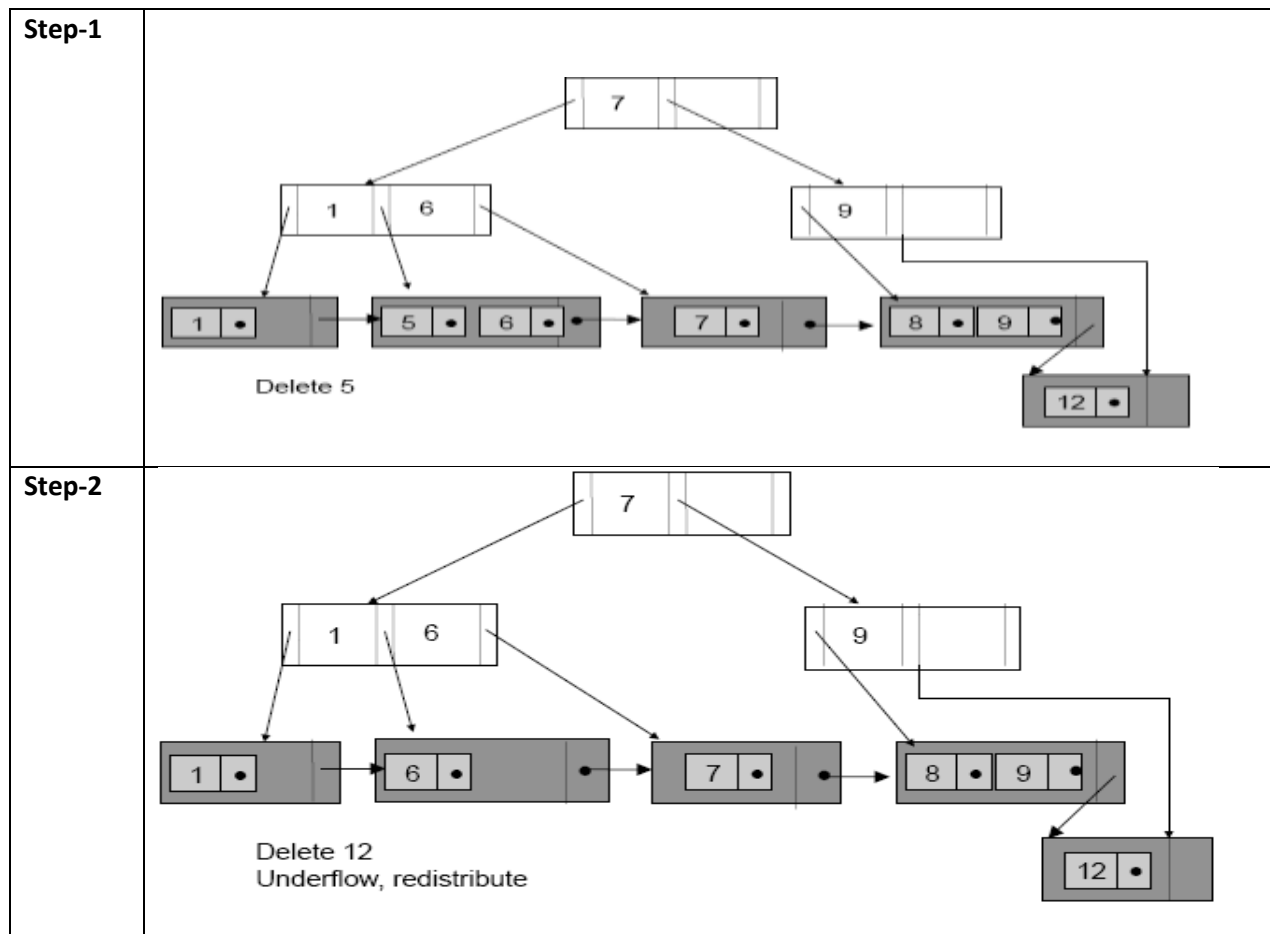
Add I : No empty place in leaf split and promote H (as it is middle value) in higher level parent. But higher level parent has also no empty place, split it and promote G to next higher level. Higher level parent has also no empty place (C, E, G), split it and promote E to next higher level.

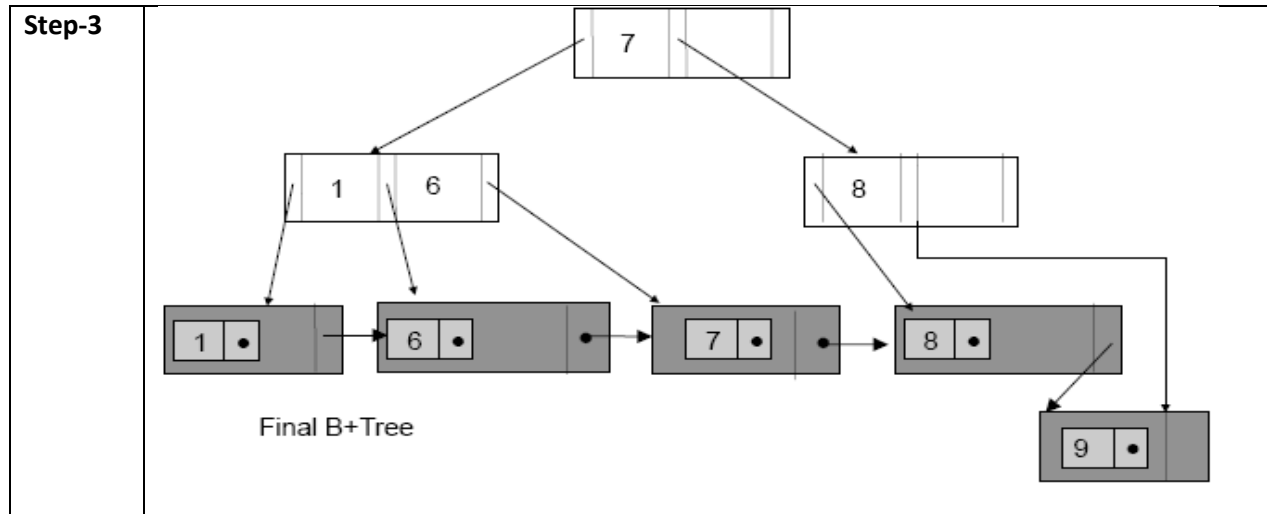


Similar after adding J,K,L and M the final tree will be as shown below :



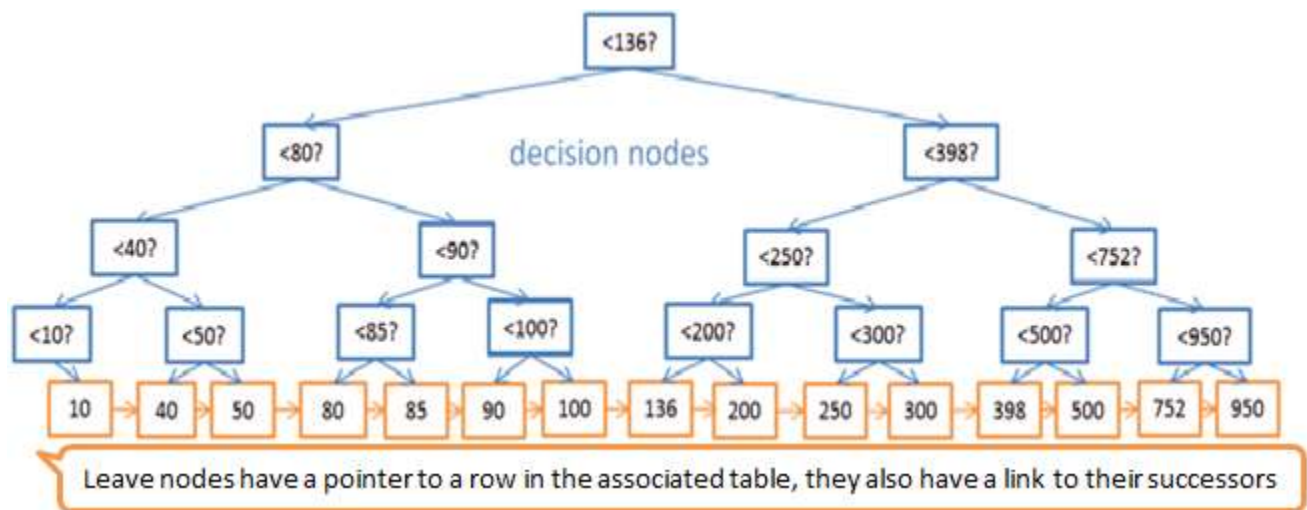
Deletion from a B+Tree





B+Tree Index : Modern databases use mainly B+Tree index which perform **range query** efficiently.

- only the leaf nodes **store information** (the location of the rows in the associated table)
- other nodes are just here **to route** to the right node **during the search**.



As you can see, there are **more nodes (twice more)**. Indeed, you have additional nodes, the “**decision nodes**” that will help you to find the right node (that stores the location of the rows in the associated table). But the search complexity is still in $O(\log(N))$ (there is just one more level). The big difference is that **the lowest nodes are linked to their successors**.

With this B+Tree, if you’re looking for values between 40 and 100:

- You just have to look for 40 (or the closest value after 40 if 40 doesn’t exist) like you did with the previous tree.
- Then gather the **successors of 40 using the direct links to the successors until you reach 100**.

Let's say you found M successors and the tree has N nodes. The search for a specific node costs $\log(N)$ like the previous tree. But, once you have this node, you get the M successors in M operations with the links to their successors. **This search only costs $M + \log(N)$ operations** vs N operations with the previous tree (B tree). Moreover, you don't need to read the full tree (just $M + \log(N)$ nodes), which means less disk usage. If M is low (like 200 rows) and N large (1 000 000 rows) it makes a BIG difference.

But there are new problems (again!). If you add or remove a row in a database (and therefore in the associated B+Tree index):

- you have to keep the order between nodes inside the B+Tree otherwise you won't be able to find nodes inside the mess.
- you have to keep the lowest possible number of levels in the B+Tree otherwise the time complexity in $O(\log(N))$ will become $O(N)$.

In other words, the B+Tree needs to be self-ordered and self-balanced.

Creating Indexes using SQL

Example: Create an index file for Lname attribute of the EMPLOYEE Table of our Database.

```
CREATE INDEX myLnameIndex ON EMPLOYEE( Lname);  
CREATE INDEX myNamesIndex ON EMPLOYEE( Lname, Fname);  
DROP INDEX myNamesIndex;
```

Choosing An Index

By default, the Oracle creates a B+Tree index.

```
SQL> create index sales_keys on sales (book_key, store_key, order_number);
```

- A multicolumn index can be used by the database but **only from the first or lead column.**

Our *sales_keys* index can be used in the following query.

```
Select order_number, quantity from sales where book_key = 'B103';
```

Note that the lead column of the index is the **book_key**, so the **database can use the index in the query above.** We can also use the *sales_keys* index in the queries below.

```
Select order_number, quantity from sales  
where book_key = 'B103' and store_key = 'S105' and order_number = 'O168';
```

However, the database cannot use that index in the query if **WHERE clause does not contain the index lead column.**

Also, note that in the query below, the database can answer the query from the index and so will not access the table at all.

Select order_number from sales where store_key = 'S105' and book_key = 'B108';

As you can see, b-tree indexes are very powerful. You must remember that a multicolumn index cannot skip over columns, so the **lead index column must be in the WHERE clause filters.**

However, the Oracle database provides specialized indexes that can **provide additional capabilities**; the **bit-mapped index** and the **function-based index**.

Function-Based Index?

- The default type of index is a b-tree index, which creates an index on one or more columns using a tree-like structure.
- bitmap index is one that creates a two-dimensional map of rows and values in a specific column.
- A function-based index, on the other hand, is an index that is **created on the results of a function or expression.**
- In Oracle, when you create an index on a column (such as a b-tree index), you need to mention the value exactly (without modification) for the index to be used. For example, if you index a column called **sale_amount** on the sales table, and query the table using:
WHERE **sale_amount = 104.95** The index should be considered.
- However, if you have the same index and run a query to say:
WHERE **ROUND(sale_amount) = 105** Then the index won't be used.
This is because the sale_amount is not the same as ROUND(sale_amount).

So, how can we use an index in this situation?

We create something called a function-based index. This means **we create indexes on columns where we are performing a function on that column or running an expression.**

If you have a query, or many queries, that use a function or expression in them and you'd like to improve the performance, a function-based index might be important to you.

Syntax of a Function-Based Index

How do we create a function-based index? We use the CREATE INDEX command.

```
CREATE INDEX index_name ON table_name(function(column_name));
```

The parameters of this command are:

- **index_name**: This is the name you can give to your new index.
- **table_name**: The name of the table to create a new index on
- **function**: this is the function being used on the column
- **column_name**: This is the column in the table that you're performing the function on for the index.

The function-based index can handle many kinds of expressions. It works in the same way as a b-tree index. Some examples are:

- ROUND(sale_amount)
- sale_amount * 0.2
- SUBSTR(first_name, 1, 10)
- first_name || ' ' || last_name

Bitmap Indexing

Bitmap Indexing is a special type of database indexing that uses **bitmaps**. This technique is used for **huge databases**, when column is of **low cardinality** (High **cardinality** means that the column contains a large percentage of totally **unique** values. Low **cardinality** means that the column contains a lot of “repeats” in its data range.) and these **columns are most frequently** used in the query.

Need of Bitmap Indexing –

The need of Bitmap Indexing will be clear through the below given example :

For example, Let us say that a company holds an employee table with entries like EmpNo, EmpName, Job, **New_Emp** and salary. Let us assume that the employees are **hired once in the year**, therefore the table will be updated very less and will remain static most of the time. But the columns will be frequently used in queries to retrieve data like : No. of female employees in the company etc.

In this case we need a **file organization method which should be fast enough to give quick results**. But any of the **traditional file organization** method is not that fast, therefore we switch to a better method of storing and retrieving data known as Bitmap Indexing.

How Bitmap Indexing is done –

In the above example of table **employee**, we can see that the column **New_Emp** has only two values **Yes** and **No** based upon the fact that the employee is new to the company or not (so it is a low cardinality column). Similarly let us assume that the Job of the Employees is divided into **4 categories only** i.e **Manager, Analyst, Clerk and Salesman**. Such columns are called columns with **low cardinality**. Even though these columns have less unique values, they can be queried very often.

Bit: Bit is a basic unit of information used in computing that can have only one of two values either 0 or 1 (logical values true/false or yes/no.)

In Bitmap Indexing these bits are used to represent the unique values in those low cardinality columns. This technique of storing the low cardinality rows in form of bits are called **bitmap indices**. Continuing the Employee example, Given below is the Employee table :

EmpNo	EmpName	Job	New_Emp	Salary
1	Alice	Analyst	Yes	15000
2	Joe	Salesperson	No	10000
3	Katy	Clerk	No	12000
4	Annie	Manager	Yes	25000

New_Emp Values	Bitmap Indices
Yes	1001
No	0110

If New_Emp is the data to be indexed, the content of the bitmap index is shown as four(As we have four rows in the above table) columns under the heading Bitmap Indices. Here Bitmap Index “Yes” has value 1001 because row 1 and row four has value “Yes” in column New_Emp.

In this case there are two such bitmaps, one for “New_Emp” Yes and one for “New_Emp” NO. It is easy to see that each bit in bitmap indices shows that whether a particular row refer to a person who is New to the company or not.

Job Values	Bitmap Indices
Analyst	1000
Salesperson	0100
Clerk	0010
Manager	0001

The above scenario is the simplest form of Bitmap Indexing. Most columns will have more distinct values. For example the column Job here will have only 4 unique values (As mentioned earlier). Variations on the bitmap index can effectively index this data as well. For Job column the bitmap Indexing is shown in figure.

Now Suppose, If we want to find out the details for the Employee who is not new in the company and is a sales person then we will run the query:

SELECT * FROM EMPLOYEE WHERE New_Emp = "No" and Job = "Salesperson";

For this query the DBMS will search the bitmap index of both the columns and perform logical AND operation on those bits and find out the actual result:

Here the result 0100 represents that the second row has to be retrieved as a result.

Bitmap Index for "NO"	→	0 1 1 0
		AND
Bitmap Index for Salesperson	→	0 1 0 0
Result	→	0 1 0 0

Bitmap Indexing in SQL – The syntax for creating bitmap index in sql is given below:

CREATE BITMAP INDEX Index_Name ON Table_Name (Column_Name);

For the above example of employee table, the bitmap index on column New_Emp will be created as follows: **CREATE BITMAP INDEX index_New_Emp ON Employee (New_Emp);**

Advantages –

- Efficiency in terms of insertion deletion and updating
- Faster retrieval of records

Disadvantages –

- Only suitable for large tables

- Bitmap Indexing is time consuming

Note on Cluster Index

In **Clustered** Index:

1. Data is stored in the order of the clustered index.
2. Only one clustered index per table.
3. When a primary key is created a cluster index is automatically created as well.

These facts are true for **SQL Server** but **not for Oracle**.

There is no such thing as create **clustered** index in Oracle. **Tables (at least ordinary ones) in Oracle do not have a clustered index**. There is a special kind of tables, called **Index Organized Tables (IOT)** that are of similar concept.

To create an index organized table, you use the create table statement with the organization index option. In Oracle you usually use IOTs for very narrow tables. Very often for tables that only consist of the primary key columns (e.g. m:n mapping tables), e.g.

```
create table assignment
(
    person_id integer not null,
    job_id    integer not null,
    primary key (person_id, job_id)
)
organization index;
```