Database indexing plays a crucial role in improving the performance and efficiency of database systems. By utilizing indexing techniques, we can speed up data retrieval operations and enhance overall system responsiveness. This article will delve into various database indexing techniques, including B-tree, Hash Indexing, and Bitmap Indexing. We will explore their unique characteristics, advantages, disadvantages, and scenarios where each technique is most suitable.

What is B-tree Indexing?

B-tree is one of the popular techniques in relational database management systems (RDBMS) is b-tree indexing. It arranges information in a balanced tree structure to facilitate effective insertion, deletion, and searching. Large dataset databases that need frequent range queries and dynamic updates are best suited for B-trees.

Advantages of B-tree Indexing

- Facilitates effective range queries and sorting procedures: B-tree indexing enables fast access to data within a specified range because of its tree structure.
- Ideal for big datasets with real-time updates: B-trees easily manage frequent inserts, updates, and deletes without compromising speed.
- delivers strong performance for a variety of queries: Faster search times are achieved because of the balanced tree structure, which guarantees a relatively low height.

Disadvantages of B-tree Indexing

- More storage space is needed: In comparison to other indexing techniques, the tree structure and internal nodes need more storage space.
- Slower insertion and deletion operations: Rebalancing the tree is a possible step in these operations, which may take some time.
- Performance during intensive writing operations may be impacted by tree rebalancing: Performance degradation may occur when tree balance is required in settings with frequent updates.

What is Hash Indexing?

Hash functions are used in hash indexing to map keys to certain places in a hash table. When handling precise match queries—such as getting a record based on a unique identifier like an ID—this approach is very helpful. For range queries or situations where data is changed often, it is not the best option.

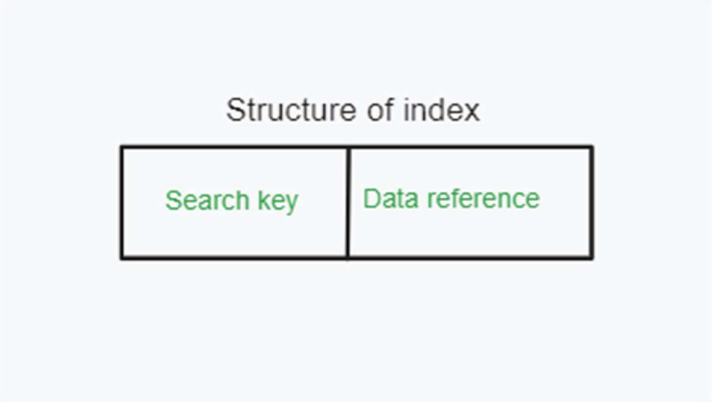
Advantages of Hash Indexing

- Constant-time lookup for precise match queries: When a query calls for exact matches, hash indexing offers quick access to information.
- Perfect for situations with low-range queries and strong data uniformity: It works best in scenarios when the likelihood of querying every entry is equal.
- less storage space is needed than with B-tree indexing: Because the hash table structure is usually more compact, less storage space is needed.

Disadvantages of Hash Indexing

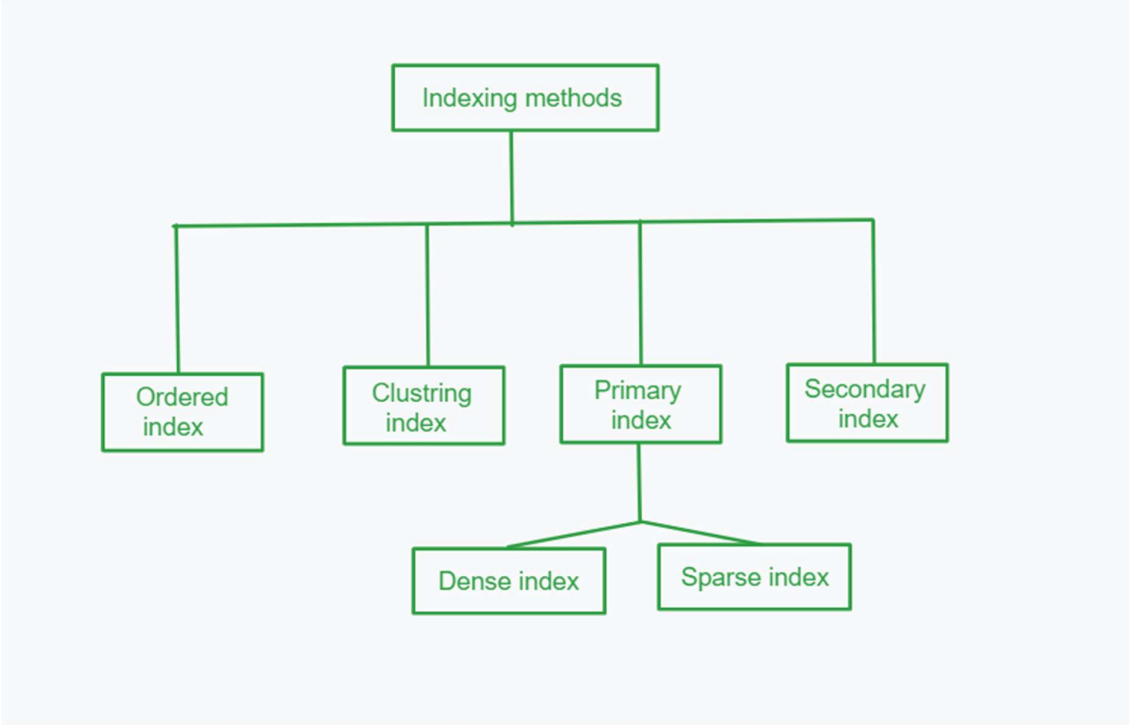
- Not suitable for range queries or partial matches: Hash indexing only works effectively for exact match queries.
- Collisions can occur, affecting performance: Multiple keys may hash to the same location, necessitating collision resolution techniques.
- Hash function selection is crucial for optimal performance: A poorly chosen hash function can lead to an uneven distribution of keys and degrade performance.

Indexing is a technique in **DBMS** that is used to optimize the performance of a database by reducing the number of disk access required. An index is a type of data structure. With the help of an index, we can locate and access data in database tables faster. The dense index and Sparse index are two different approaches to organizing and accessing data in the data structure. These are commonly used in databases and information retrieval systems.



Different Types of Indexing Methods

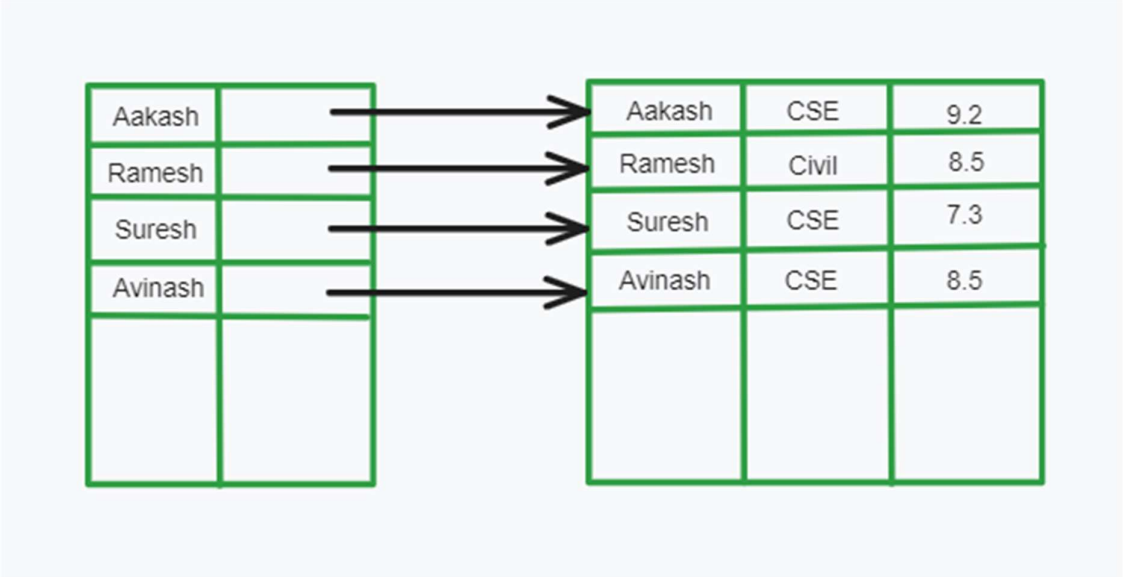
Indexing methods in a database management system (DBMS) can be classified as dense or sparse indexing methods, depending on the number of index entries in the database. Let's take a look at the differences between the two types of indexing methods:



Types of index

Dense Index

It contains an index record for every search key value in the file. This will result in making searching faster. The total number of records in the index table and main table are the same. It will result in the requirement for more space to store the index of records itself.



Dense indexing

Advantages

- Gives quick access to records, particularly for Small datasets.
- Effective for range searches since each key value has an entry.

Disadvantages

- Can be memory-intensive and may require a significant amount of storage space.
- Insertions and deletions result in a higher maintenance overhead because the index must be updated more frequently.

Sparse Index

Sparse index contains an index entry only for some records. In the place of pointing to all the records in the main table index points records in a specific gap. This indexing helps you to overcome the issues of dense indexing in DBMS.

Advantages

- Uses less storage space than thick indexes, particularly for large datasets.
- Lessens the effect that insertions and deletions have from index maintenance operations.

Disadvantages

- Since there may not be an index entry for every key value, access may involve additional steps.
- might not be as effective as dense indexes for range queries.
-

Difference Between Dense Index and Sparse Index

Dense index	Sparse index
The index size is larger in dense index.	In sparse index, the index size is smaller.
Time to locate data in index table is less.	Time to locate data in index table is more.
There is more overhead for insertions and deletions in dense index.	Sparse indexing have less overhead for insertions and deletions.
Records in dense index need not to be clustered.	In case of sparse index, records need to be clustered.
Computing time in RAM (Random access memory) is less with dense index.	In sparse index, computing time in RAM is more.
Data pointers in dense index point to each record in the data file.	In sparse index, data pointers point to fewer records in data file.
Search performance is generally faster in dense index.	In sparse index, search performance may require additional steps, which will result in slowing down the process.

Indexing and hashing are two crucial techniques used in databases to improve the efficiency of data retrieval and query performance. You can search and retrieve entries from databases rapidly thanks to a data structure that indexing makes feasible. However because hashing uses a mathematical hash function to transfer data to its storage location directly on disk, it does not need index structures. Understanding the differences between these two ways may help in choosing the optimal option based on the kind of query, database size, and performance requirements.

What is Indexing?

Indexing, as the name suggests, is a technique or mechanism generally used to speed up access of data. The index is a type of data structure that is used to locate and access data in a database table quickly. Indexes can easily be developed or created using one or more columns of a database table.

Advantages of Indexing

- **Faster Data Retrieval:** Indexing improves query speed by drastically lowering the amount of disk accesses needed to obtain data.
- **Efficient Sorting and Searching**: This makes it easier to quickly retrieve sorted data, which is useful for activities like organizing, searching, and grouping.
- Reduces table space by storing just pointers to data rather than the actual data, hence minimizing storage capacity.
- **Supports Random Lookups**: By providing efficient access to ordered data, this feature helps to speed up random lookups.

Disadvantages of Indexing

- **Increased Maintenance Overhead**: When indexes are updated often, this may lead to an increase in maintenance overhead and the need for extra storage.
- **Not Suitable for Big Databases**: Performance may suffer if the database is too large or has an excessive number of indexes, which may slow down writes and updates.
- **Performance Impact on Insertions and Updates**: The need to update the indexes after each data insertion, deletion, or update may lead to slower data modification operations.

What is Hashing?

Hashing, as name suggests, is a technique or mechanism that uses hash functions with search keys as parameters to generate address of data record. It calculates direct location of data record on disk without using index structure. A good hash functions only uses one-way hashing algorithm and hash cannot be converted back into original key. In simple words, it is a process of converting given key into another value known as hash value or simply hash.

Advantages of Hashing

- This approach, which determines the precise storage location using a hash function, enables quick and easy access to data.
- efficient with large databases Large databases may benefit from its ability to handle large volumes of data without negatively affecting search performance.
- **Increased Flexibility and Reliability**: By organizing data into easily searchable "buckets," this method offers a trustworthy means of retrieving information.
- **Fast Search Results**: it allows for efficient comparison of large datasets and is faster than more traditional data structures like lists and arrays.

Disadvantages of Hashing

- **Fixed Hash Values**: When two keys map to the same location, the hash function produces a fixed-length hash result that might cause collisions.
- **Unsuitable for Range Inquiries**: Hashing is ineffective when doing ordered retrievals or range searches.
- **Data Integrity Problems**: Data integrity problems may arise from improper handling of hash collisions.
- **Complex Execution**: To avoid collisions and provide a consistent data distribution, hash functions need to be carefully selected.

Difference Between Indexing and Hashing in DBMS

Indexing	Hashing
It is a technique that allows to quickly retrieve records from database file.	It is a technique that allows to search location of desired data on disk without using index structure.

Indexing	Hashing
It is generally used to optimize or increase performance of database simply by minimizing number of disk accesses that are required when a query is processed.	It is generally used to index and retrieve items in database as it is faster to search that specific item using shorter hashed key rather than using its original value.
It offers faster search and retrieval of data to users, helps to reduce table space, makes it possible to quickly retrieve or fetch data, can be used for sorting, etc.	It is faster than searching arrays and lists, provides more flexible and reliable method of data retrieval rather than any other data structure, can be used for comparing two files for quality, etc.
Its main purpose is to provide basis for both rapid random lookups and efficient access of ordered records.	Its main purpose is to use math problem to organize data into easily searchable buckets.
It is not considered best for large databases and its good for small databases.	It is considered best for large databases.
Types of indexing includes ordered indexing, primary indexing, secondary indexing, clustered indexing.	Types of hashing includes static hashing and dynamic hashing.
It uses data reference to hold address of disk block.	It uses mathematical functions known as hash function to calculate direct location of records on disk.
It is important because it protects file and documents of large size business organizations, and optimize performance of database.	It is important because it ensures data integrity of files and messages, takes variable length string or messages and compresses and converts it into fixed length value.

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

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

Ordered Indexing is of two types –

- Dense Index
- Sparse Index

Dense Index

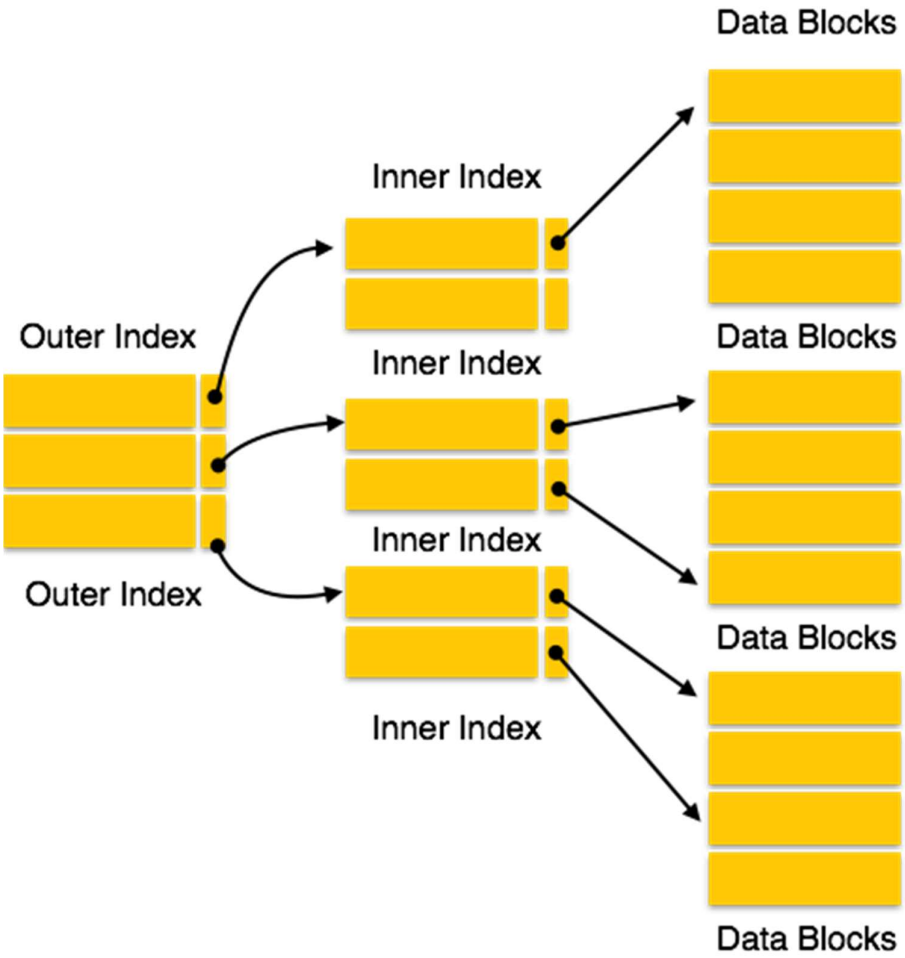
In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.

Sparse Index

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

Multilevel Index

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



Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved

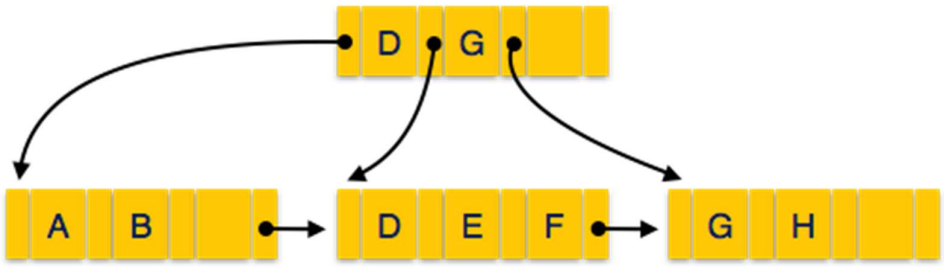
in a single disk block, which can easily be accommodated anywhere in the main memory.

B+ Tree

A B+ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B+ tree denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, the leaf nodes are linked using a link list; therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

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



Internal nodes −

- Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
- At most, an internal node can contain **n** pointers.

Leaf nodes −

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

B+ Tree Insertion

- B+ trees are filled from bottom and each entry is done at the leaf node.
- If a leaf node overflows −
  - Split node into two parts.
  - Partition at **i** = ⌊**(m+1)**/2⌋.
  - First **i** entries are stored in one node.

    - Rest of the entries (i+1 onwards) are moved to a new node.
    - i^{th} key is duplicated at the parent of the leaf.
- If a non-leaf node overflows −
    - Split node into two parts.
    - Partition the node at $i = \lceil (m+1)/2 \rceil$.
    - Entries up to i are kept in one node.
    - Rest of the entries are moved to a new node.

B^+ Tree Deletion

- B^+ tree entries are deleted at the leaf nodes.
- The target entry is searched and deleted.
    - If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
    - If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
    - Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then
    - Merge the node with left and right to it.

Primary Indexing in Databases

Indexing is a technique used to reduce access cost or I/O cost, now the question arrives what is access cost? Access cost is defined as the number of secondary memory blocks which is transferred from secondary memory to main memory in order to access required data. In this article, we are going to discuss every point about primary indexing.

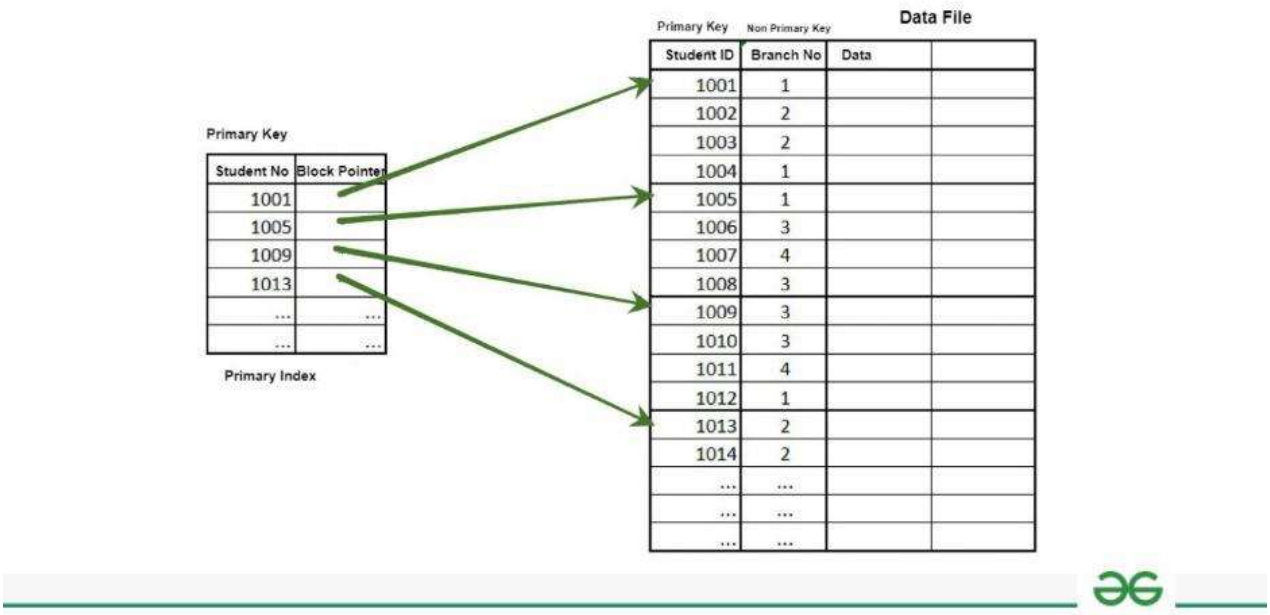
What is Primary Indexing?

A Primary Index is an ordered file whose records are of fixed length with two fields. The first field of the index is the primary key of the data file in an ordered manner, and the second field of the ordered file contains a block pointer that points to the data block where a record containing the key is available.

Primary Indexing

Working of Primary Indexing

- In primary indexing, the data file is sorted or clustered based on the primary key as shown in the below figure.
- An **index file** (also known as the **index table**) is created alongside the data file.
- The index file contains pairs of primary key values and pointers to the corresponding data records.
- Each entry in the index file corresponds to a block or page in the data file.



Primary Indexing

Types of Primary Indexing

- **Dense Indexing**: In Dense Index has an index entry for every search key value in the data file. This approach ensures efficient data retrieval but requires more storage space.
  No of Index Entry = No of DB Record

- **Sparse Indexing**: Sparse indexing involves having fewer index entries than records. The index entries point to blocks of records rather than individual records. While it reduces storage overhead, it may require additional disk accesses during retrieval.
  No of Index Entry = No of Block

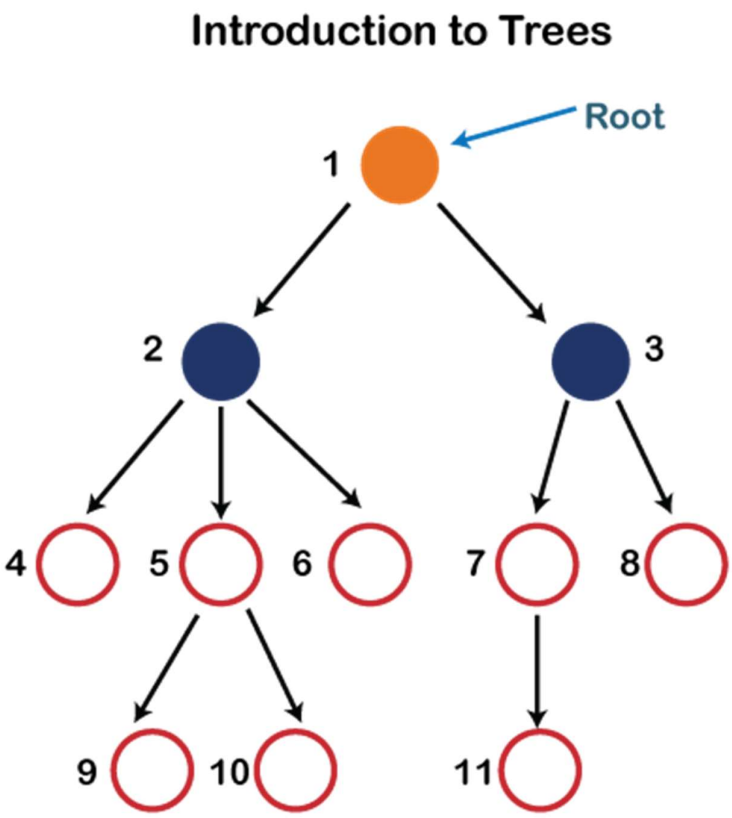
Advantages of Primary Indexing

- Primary indexing allowed fast retrieval of records based on their primary key values.
- Primary indexing reduces the need for full-table scans or sequential lookups, due to which it reduce disk I/O operations.
- The data file is organized based on the primary key, ensuring that records are stored in a logical order. This organization simplifies range queries and other operations.
- Each block in the data file corresponds to an entry in the primary index. Therefore, the average number of blocks accessed using the primary index can be estimated as approximately $\mathbf{\log_2(B + 1)}$, where **B** represents the number of index blocks.

Disadvantages of Primary Indexing

- Primary indexing requires additional space to store the index field alongside the actual data records. This extra storage can impact overall system costs, especially for large datasets.
- When records are added or deleted, the data file needs reorganization to maintain the order based on the primary key.
- After record deletion, the space used by the deleted record must be released for reuse by other records. Managing this memory cleanup can be complex.

Let's consider the tree structure, which is shown below:



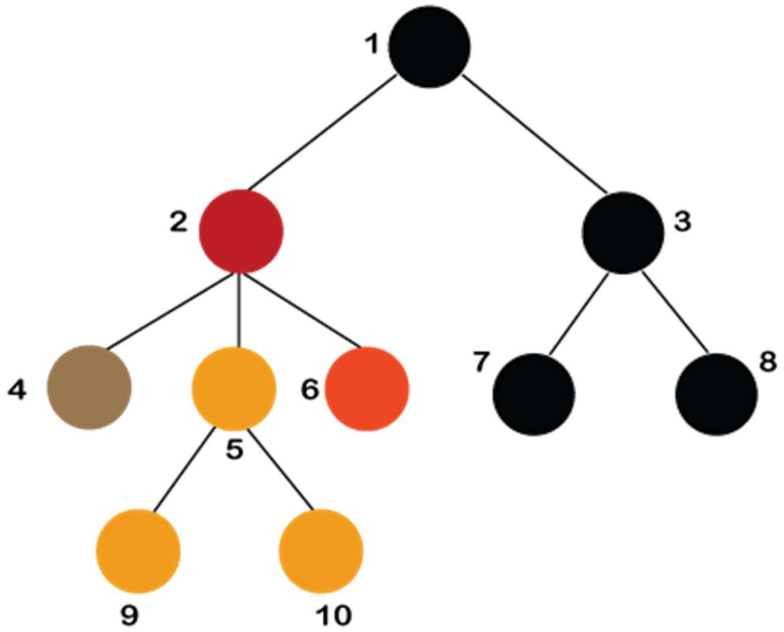
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.

- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.

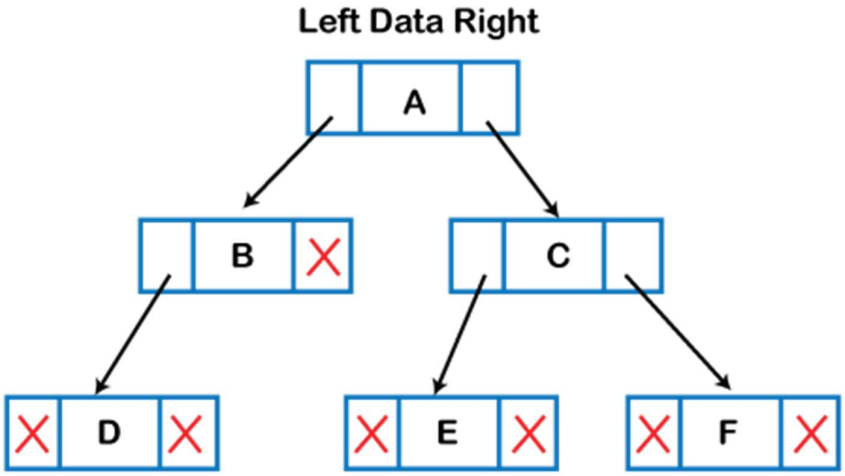


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file

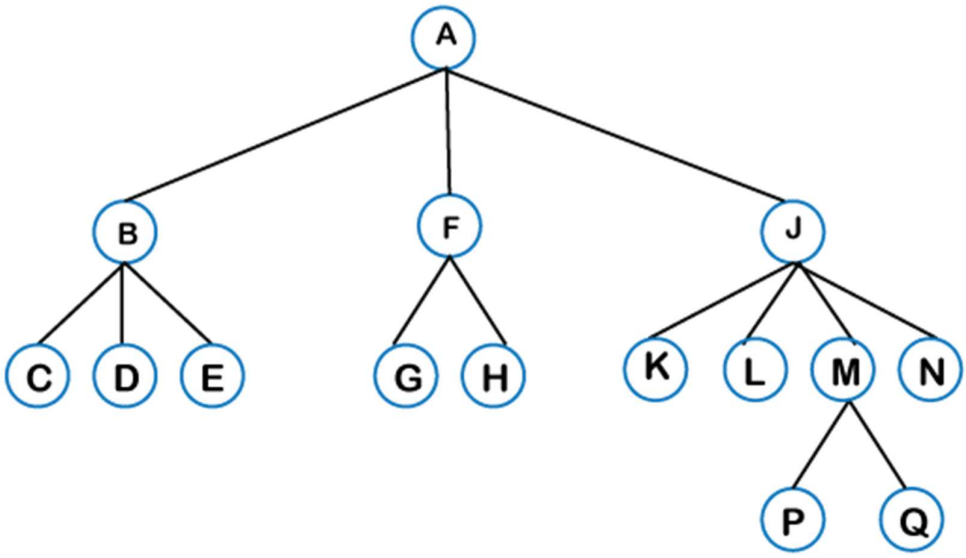
system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.

- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

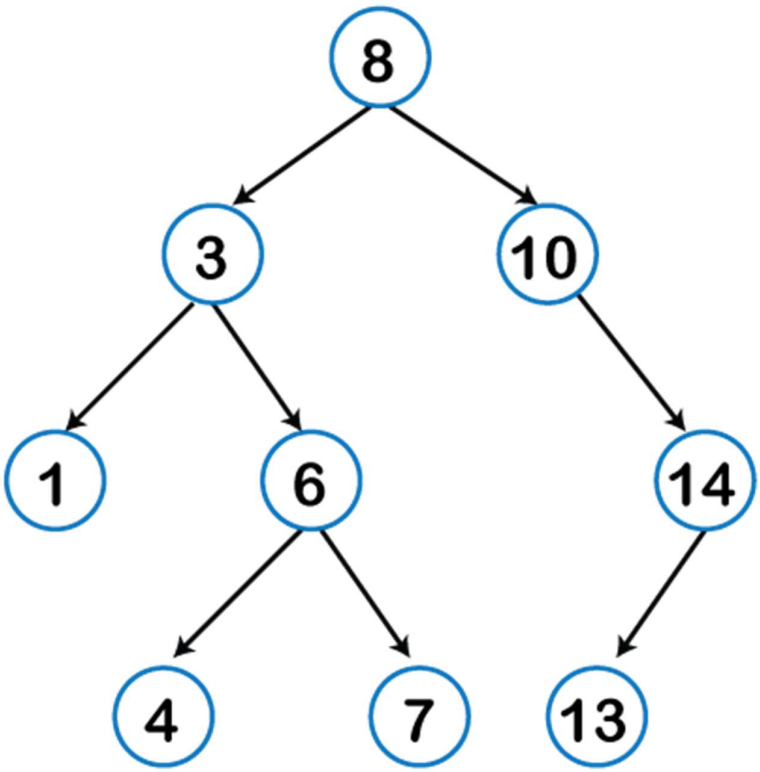
The following are the types of a tree data structure:

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered. Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

- Binary tree**:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



To know more about the binary tree, click on the link given below:

- Binary Search tree**:** Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer). Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

- AVL tree

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

B.Tree

B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

If order is m then node has the following properties:

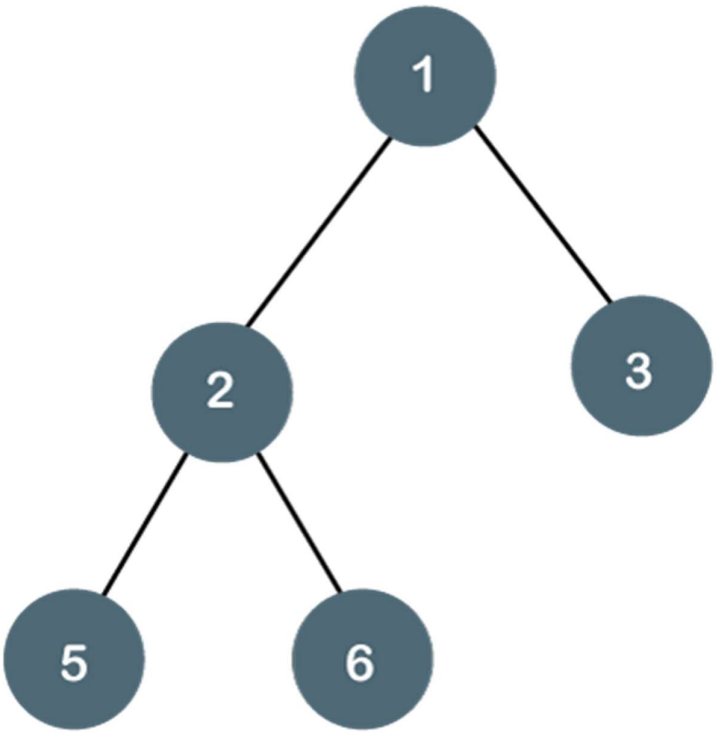
- Each node in a b-tree can have maximum **m** children
- For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
- Each node has maximum (m-1) keys.

The root node must contain minimum 1 key and all other nodes must contain atleast **ceiling of m/2 minus 1** keys.

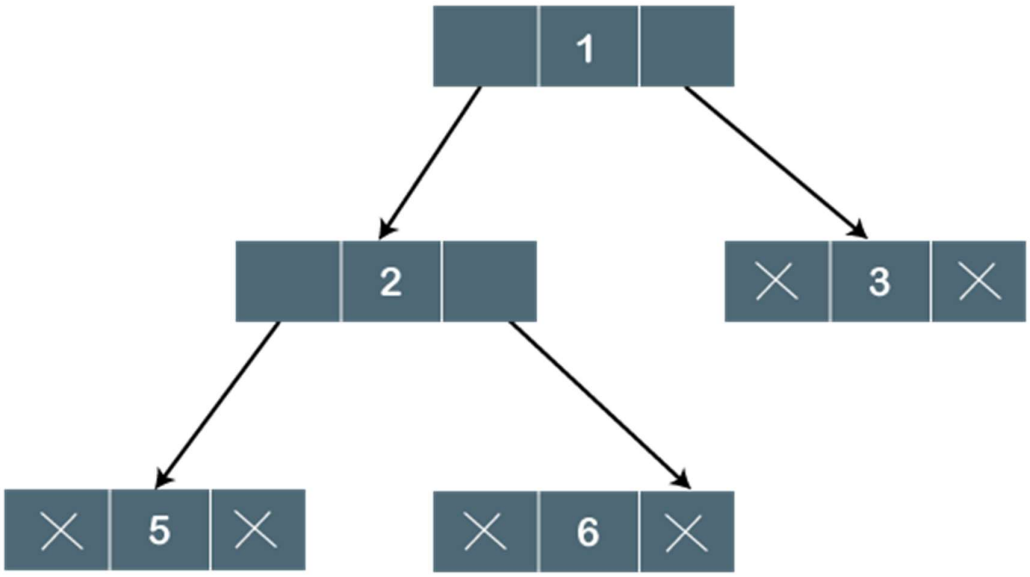
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.

- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + ...2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.