DBMS

(DATABASE MANAGEMENT SYSTEM)

LESSON 8

File Organization in DBMS:

A database consist of a huge amount of data. The data is grouped within a table in RDBMS, and each table have related records. A user can see that the data is stored in form of tables, but in actual this huge amount of data is stored in physical memory in form of files.

File – A file is named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tables and optical disks.

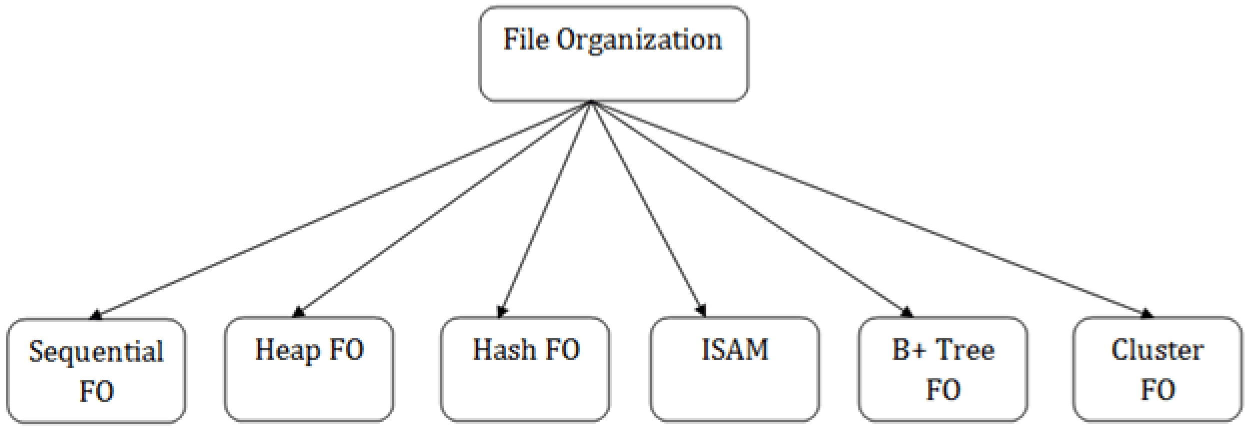
What is File Organization?

File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, Storing the files in certain order is called file Organization. **File Structure** refers to the format of the label and data blocks and of any logical control record.

Types of file organization:

File organization contains various methods. These particular methods have pros and cons on the basis of access or selection. In the file organization, the programmer decides the best-suited file organization method according to his requirement.

Types of file organization are as follows:



- Sequential File Organization
- Heap File Organization
- Hash File Organization
- B+ Tree File Organization
- Clustered File Organization

Indexing in Databases :

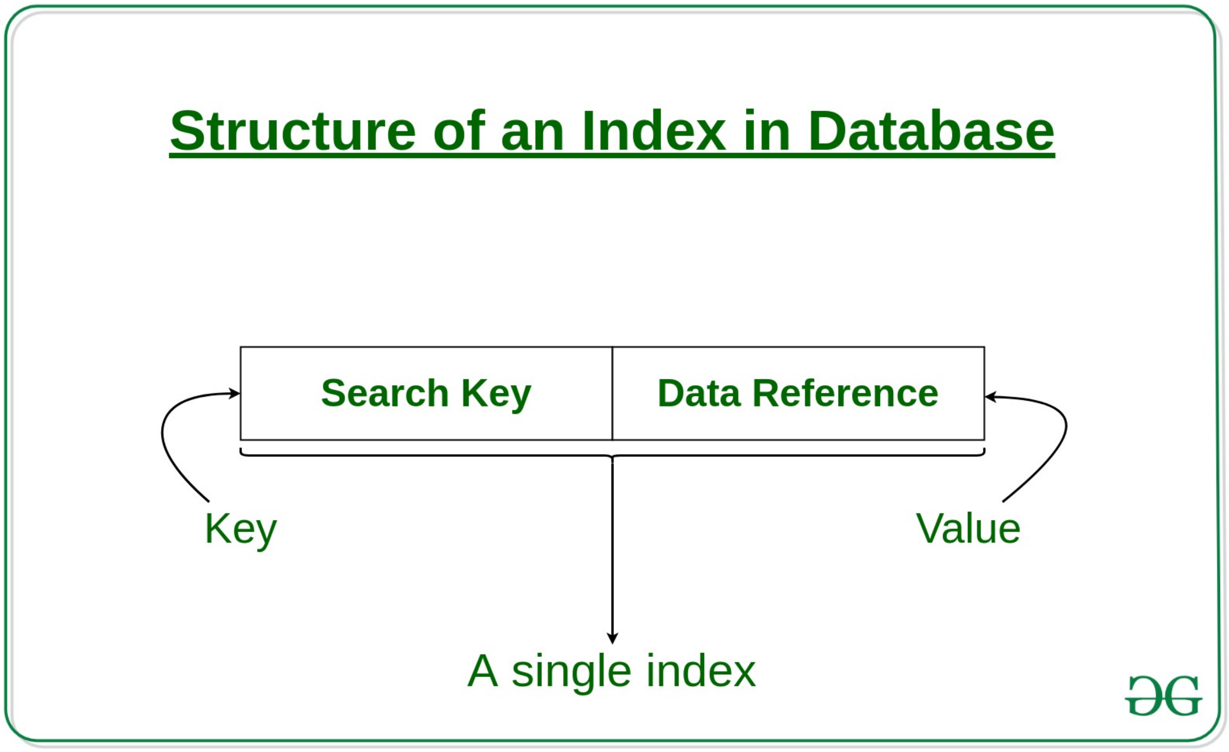
Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

- 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: The data may or may not be stored in sorted order.

- The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.



The indexing has various attributes:

- ***Access Types***: This refers to the type of access such as value based search, range access, etc.

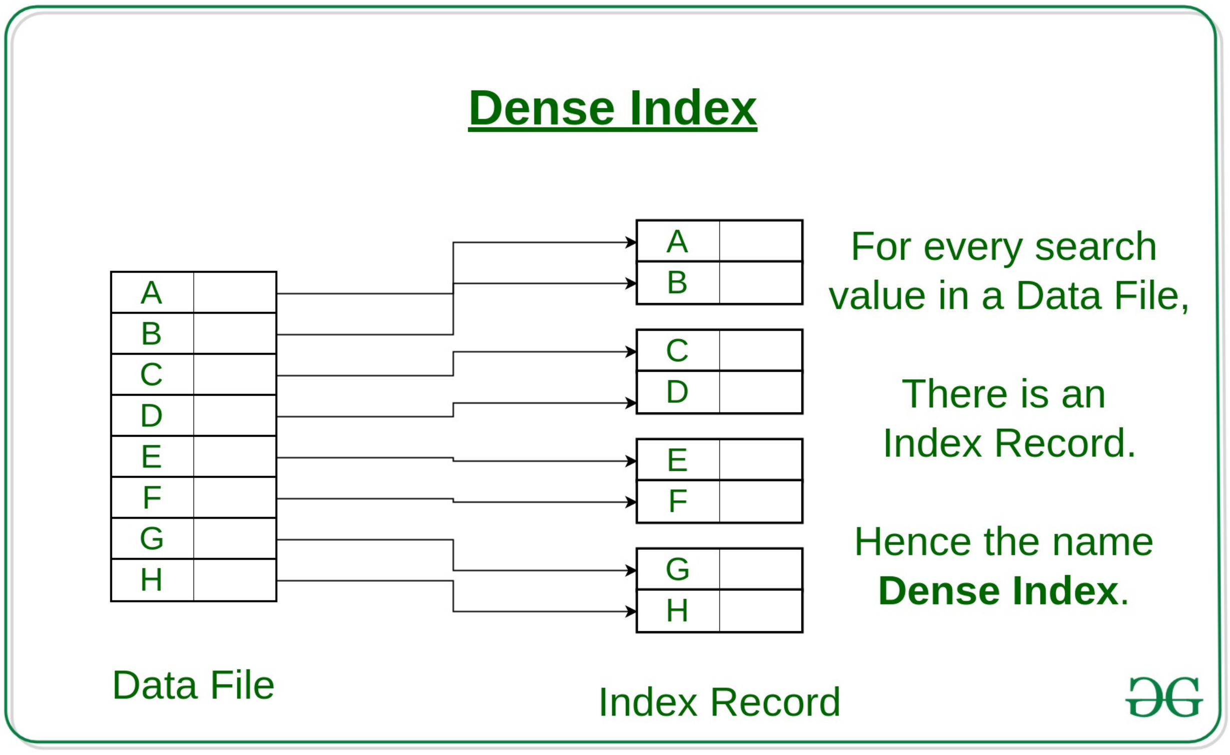
- **Access Time**: It refers to the time needed to find particular data element or set of elements.
- **Insertion Time**: It refers to the time taken to find the appropriate space and insert a new data.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: It refers to the additional space required by the index.

In general, there are two types of file organization mechanism which are followed by the indexing methods to store the data:

1. Sequential File Organization or Ordered Index File: In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organization might store the data in a dense or sparse format:

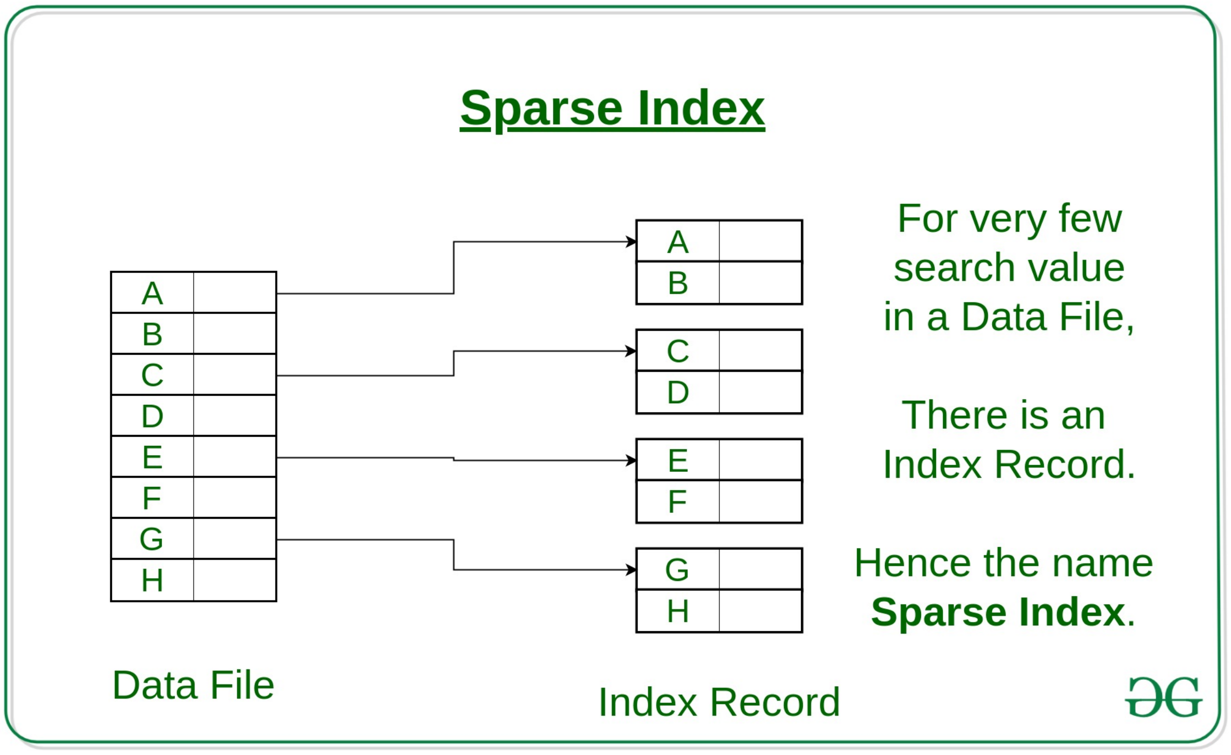
(i) Dense Index:

- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value.


Dense Index
A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H
For every search value in a Data File,
There is an Index Record.
Hence the name Dense Index.
Data File
Index Record

(ii) Sparse Index:

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.


Sparse Index
A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H
For very few search value in a Data File,
There is an Index Record.
Hence the name Sparse Index.
Data File
Index Record

2. Hash File organization: Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function called a hash function.

There are primarily three methods of indexing:

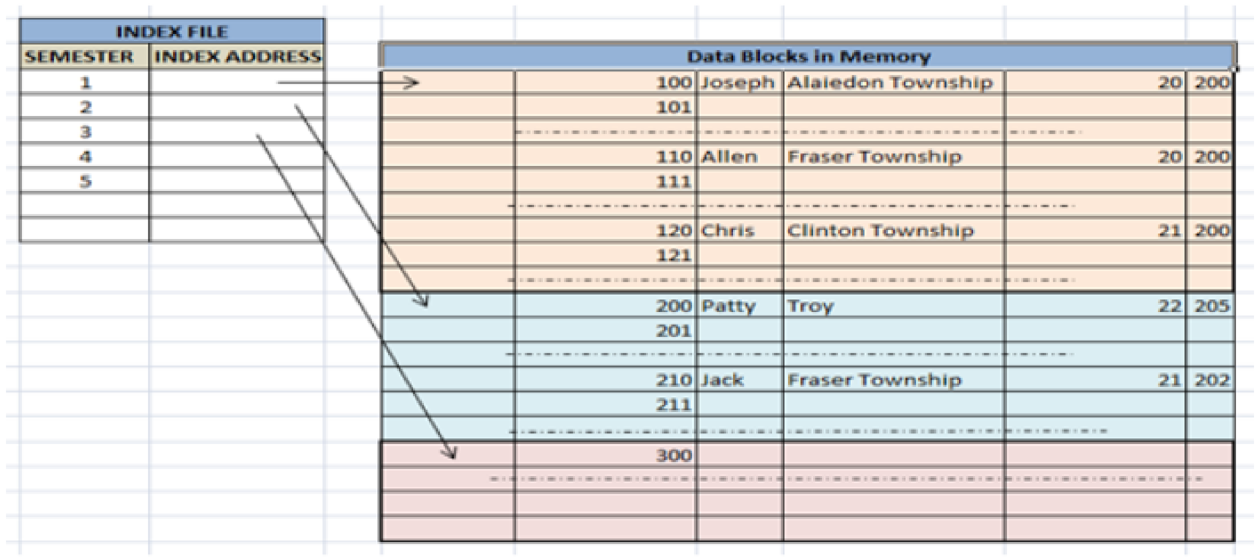
- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

1. Clustered Indexing

When more than two records are stored in the same file these types of storing known as cluster indexing. By using the cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored at one place and it also gives the frequent joining of more than two tables (records).

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as the clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc. are grouped.



INDEX FILE

SEMESTER	INDEX ADDRESS
1	
2	
3	
4	
5	

Data Blocks in Memory

	100	Joseph	Alaiedon Township	20	200
	101				
	110	Allen	Fraser Township	20	200
	111				
	120	Chris	Clinton Township	21	200
	121				
	200	Patty	Troy	22	205
	201				
	210	Jack	Fraser Township	21	202
	211				
	300				

Clustered index sorted according to first name (Search key)

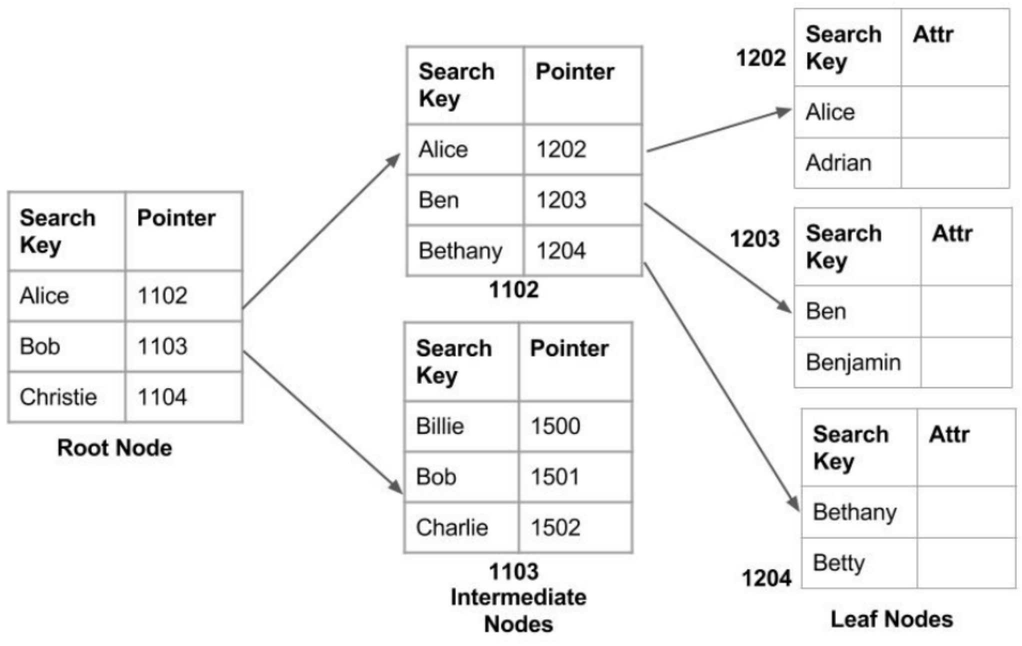
Primary Indexing:

This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.

2. Non-clustered or Secondary Indexing

A non clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.

It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.



Non clustered index

3. Multilevel Indexing

With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.

Introduction of B-Tree:

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

"n" is the total number of elements in the B-tree.

Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3. Every node except root must contain at least t-1 keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most 2*t – 1 keys.

5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).
9. Insertion of a Node in B-Tree happens only at Leaf Node.

Interesting Facts:

1. The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is: $h_{min} = \lceil \log_m(n+1) \rceil - 1$
2. The maximum height of the B-Tree that can exist with n number of nodes and d is the minimum number of children that a non-root node can have is: $t = \lceil \frac{m}{2} \rceil$ and $h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$

Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

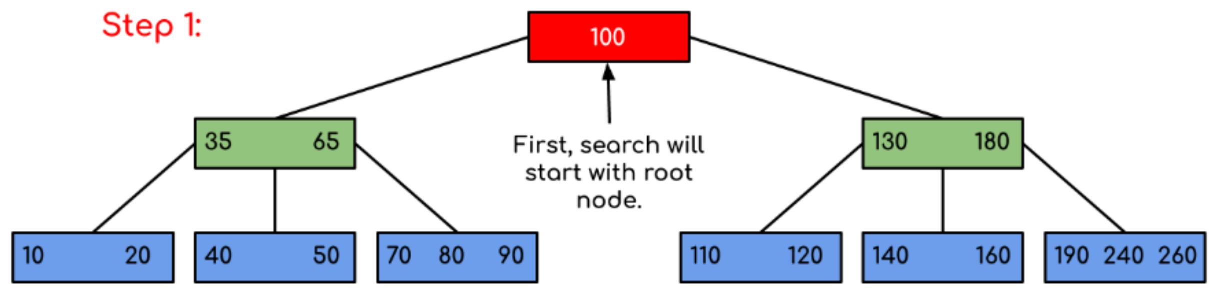
Search Operation in B-Tree:

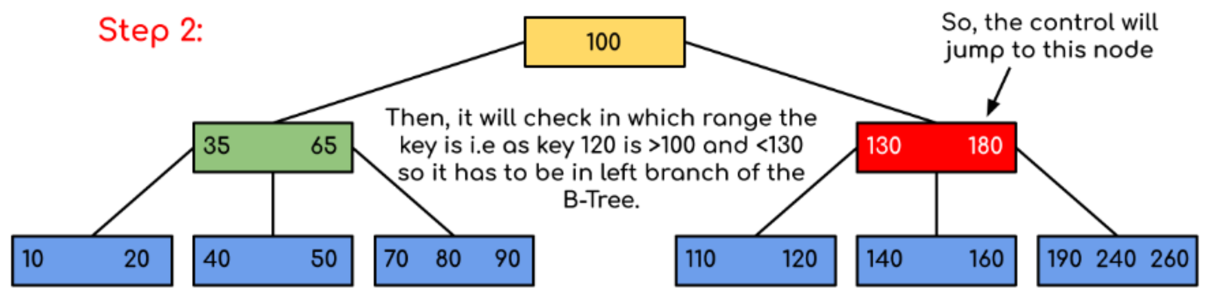
Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

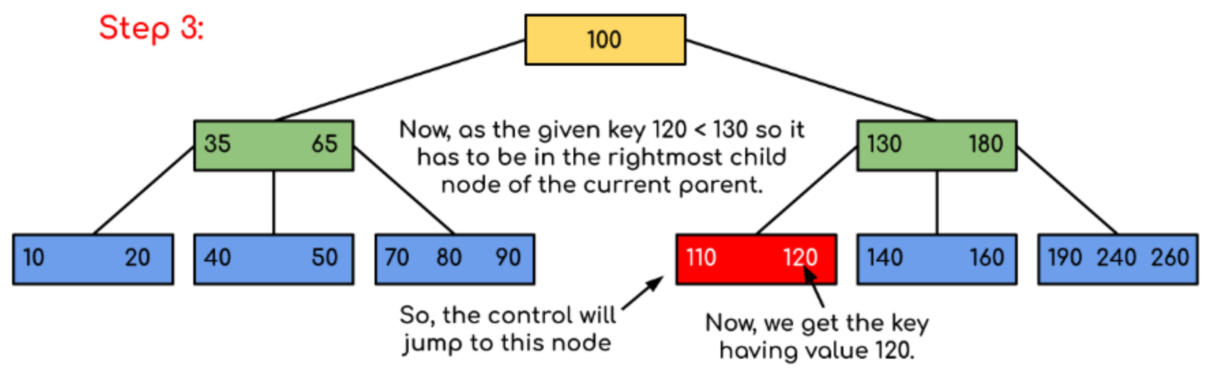
Logic:

Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of parent then the key is present in another branch. As these values limit the search they are also known as limiting value or separation value. If we reach a leaf node and don't find the desired key then it will display NULL.

Example: Searching 120 in the given B-Tree.

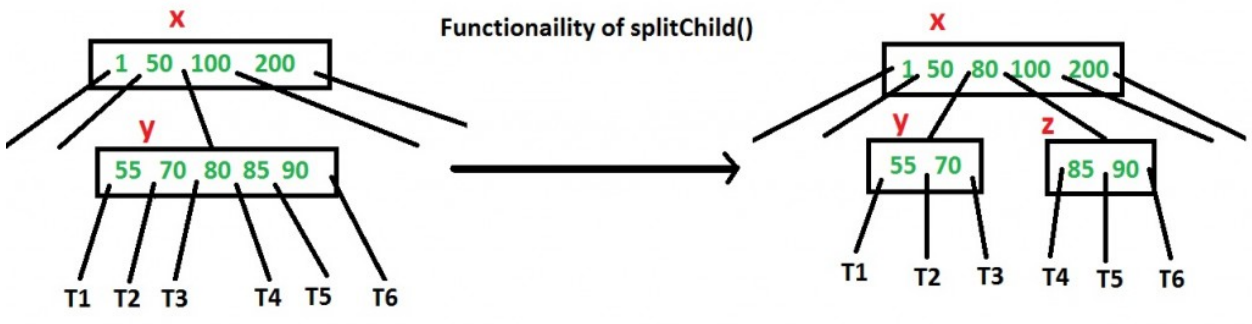



Step 2:
100
So, the control will jump to this node
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260


Step 3:
100
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

Insert Operation in B-Tree:

A new key is always inserted at the leaf node. Let the key to be inserted be k. Like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.



Insertion :

1) Initialize x as root.

2) While x is not leaf, do following

..**a)** Find the child of x that is going to be traversed next. Let the child be y.

..**b)** If y is not full, change x to point to y.

..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Delete Operation in B-Tree:

We sketch how deletion works with various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

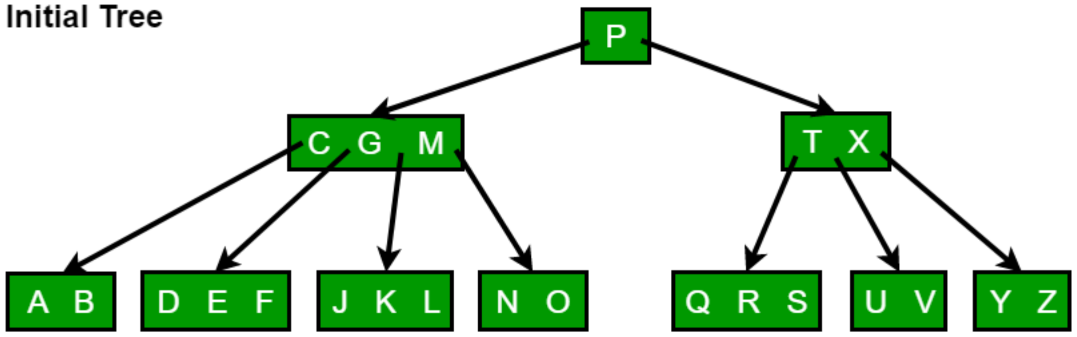
a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

b) If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

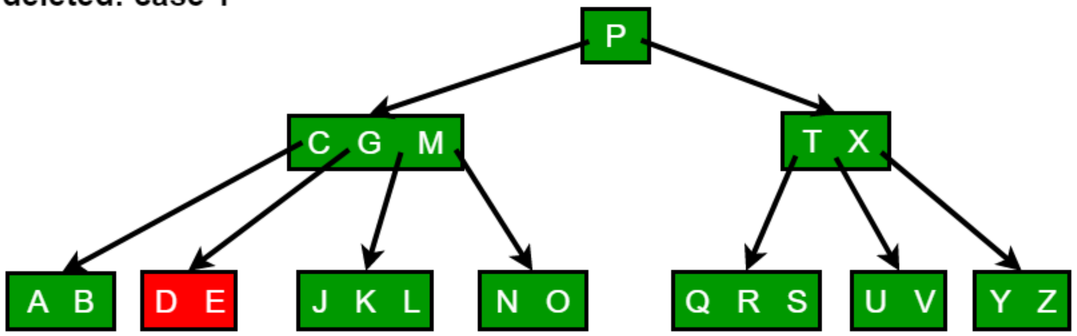
Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

The following figures explain the deletion process.

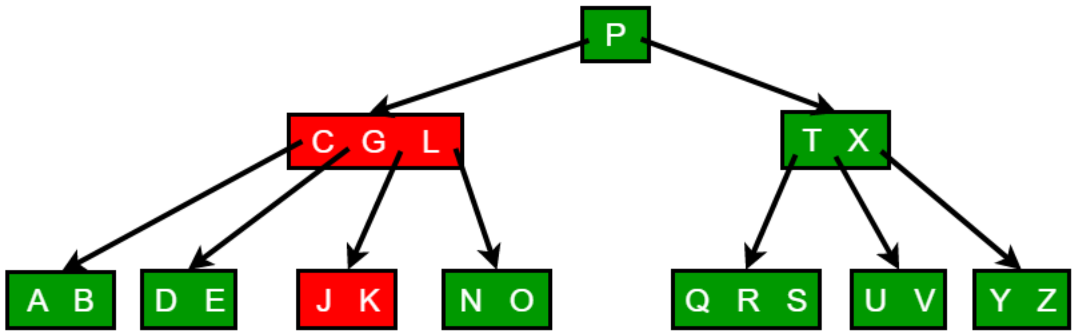
(a) Initial Tree



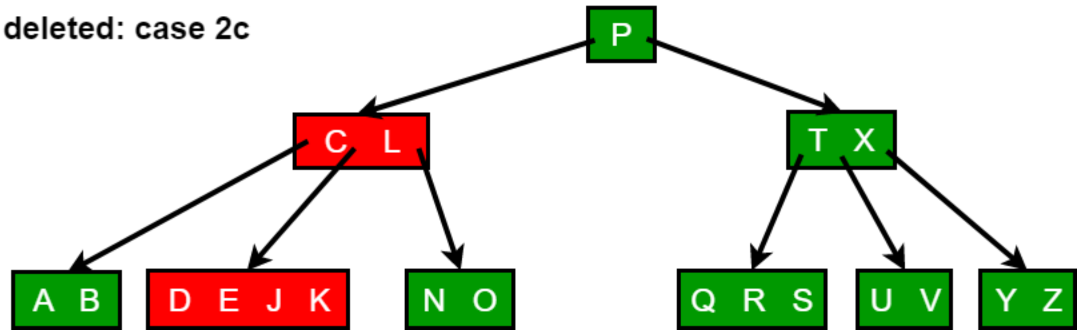
(b) F deleted: case 1

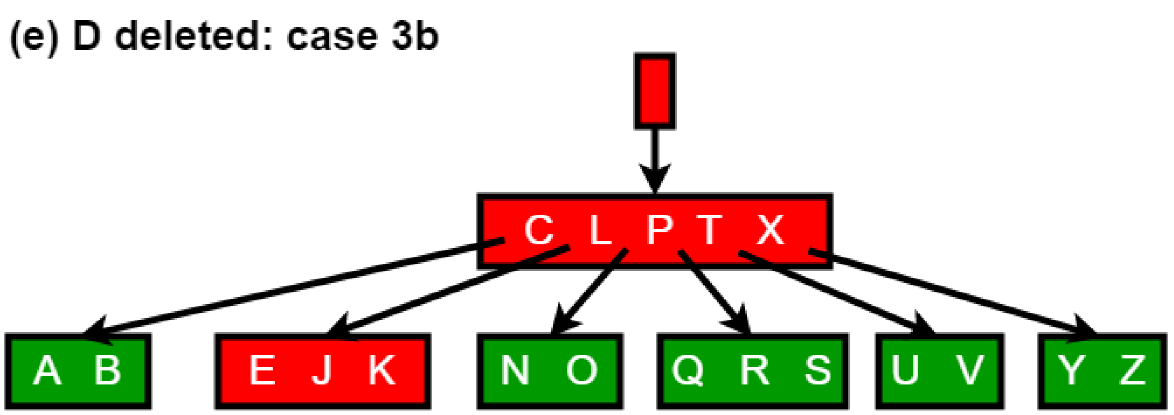


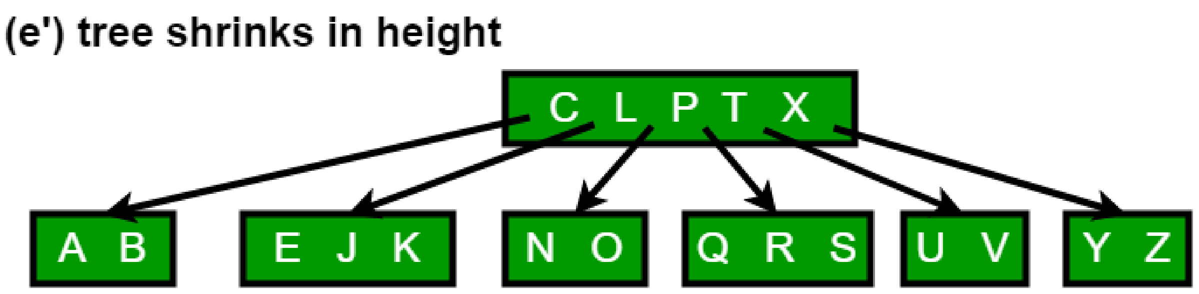
(c) M deleted: case 2a

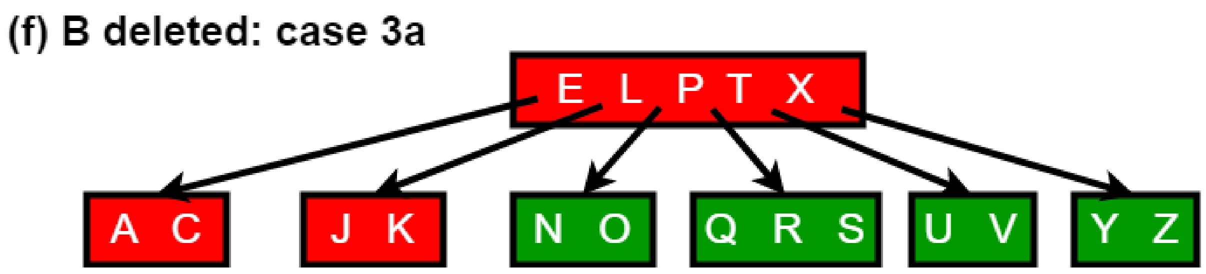


(d) G deleted: case 2c









Introduction of B+ Tree:

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of B-tree used for indexing, however is that 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. This technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the 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. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form the first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

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

1. Each internal node is of the form :
   $<P_1, K_1, P_2, K_2, \ldots, P_{c-1}, K_{c-1}, P_c>$
   where c <= a and each **P_i is a tree pointer (i.e points to another node of the tree)** and, each **K_i is a key value** (see diagram-I for reference).

2. Every internal node has : $K_1 < K_2 < .... < K_{c-1}$
3. For each search field values 'X' in the sub-tree pointed at by P_i, the following condition holds :

   $K_{i-1} < X <= K_i$, for $1 < i < c$ and,

   $K_{i-1} < X$, for $i = c$

   (See diagram I for reference)
4. Each internal nodes has at most 'a' tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
6. If any internal node has 'c' pointers, c <= a, then it has 'c – 1' key values.

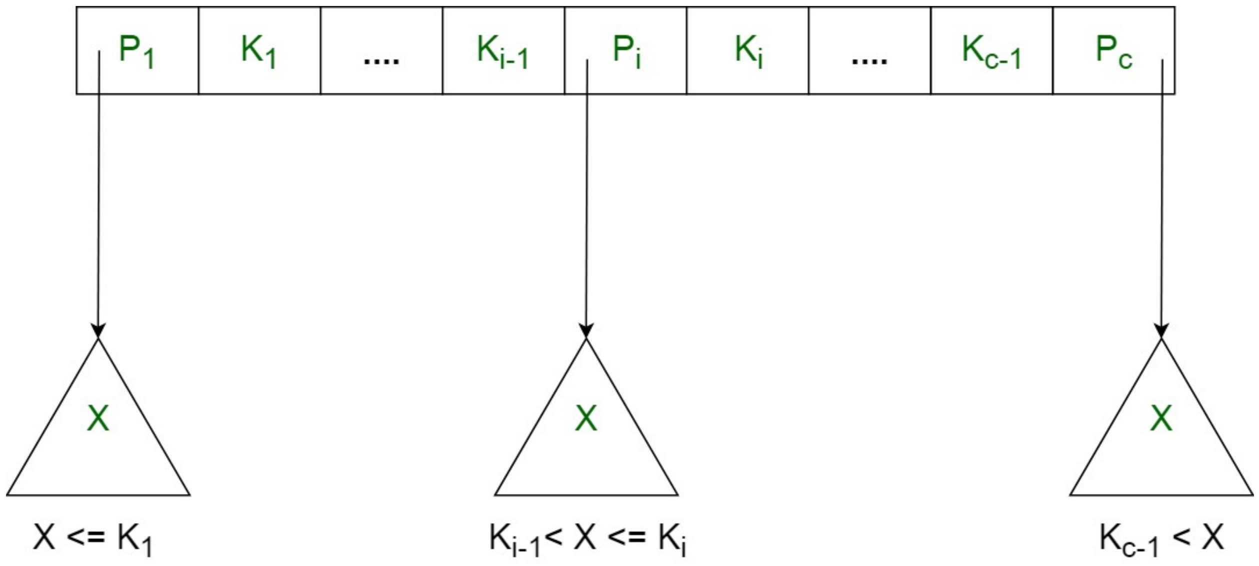


Diagram-I

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

1. Each leaf node is of the form :

   $<<K_1, D_1>, <K_2, D_2>, ..., <K_{c-1}, D_{c-1}>, P_{next}>$

   where c <= 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 II for reference).
2. Every leaf node has : $K_1 < K_2 < .... < K_{c-1}$, c <= b
3. Each leaf node has at least \ceil(b/2) values.
4. All leaf nodes are at same level.

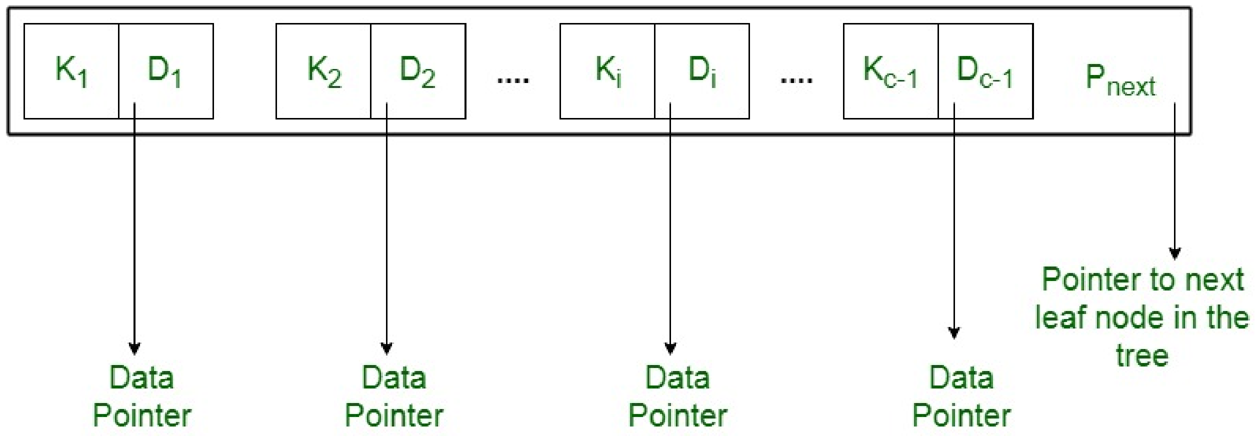
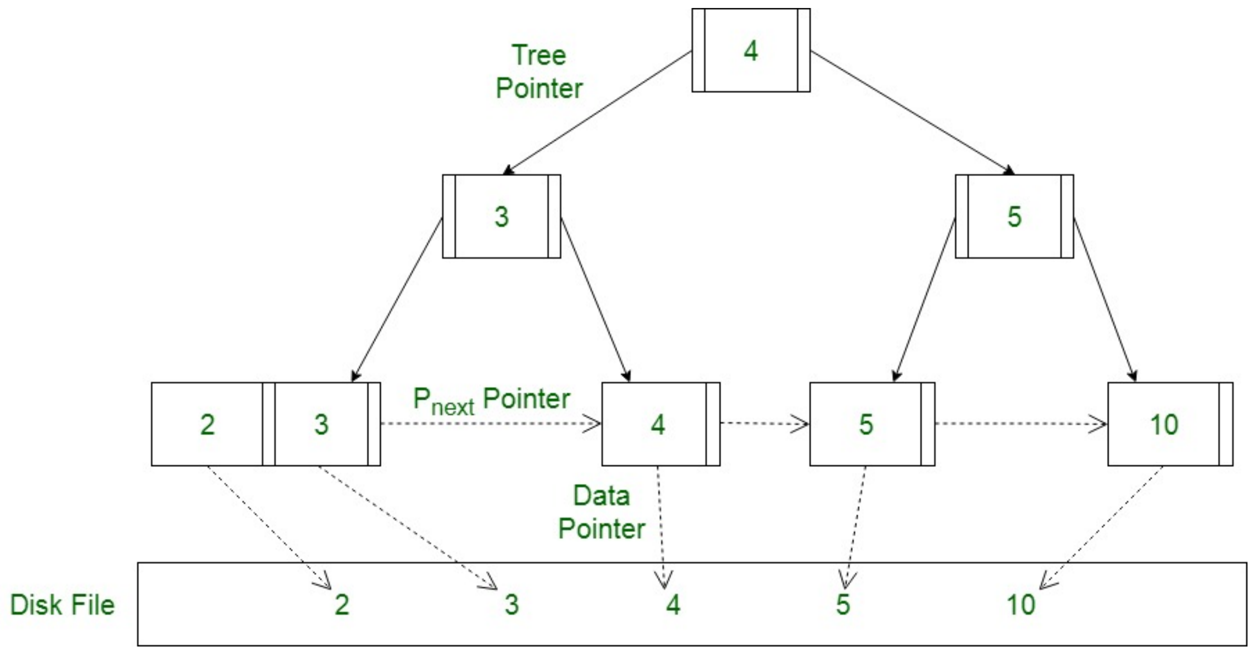


Diagram-II

Using the P_{next} pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.

A Diagram of B+ Tree –



Advantage –

A B+ tree with 'l' levels can store more entries in its internal nodes compared to a B-tree having the same 'l' levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and presence of P_{next} pointers imply that B+ tree are very quick and efficient in accessing records from disks.