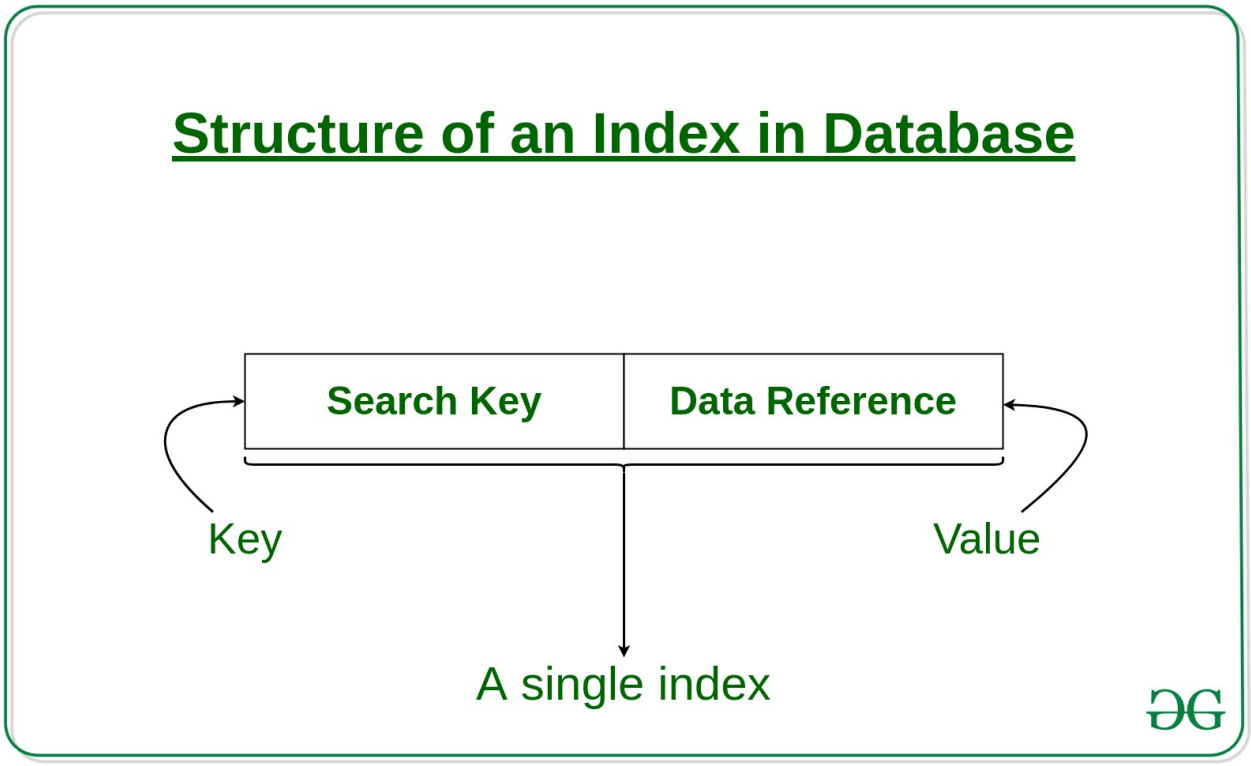
Indexing in Databases

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.

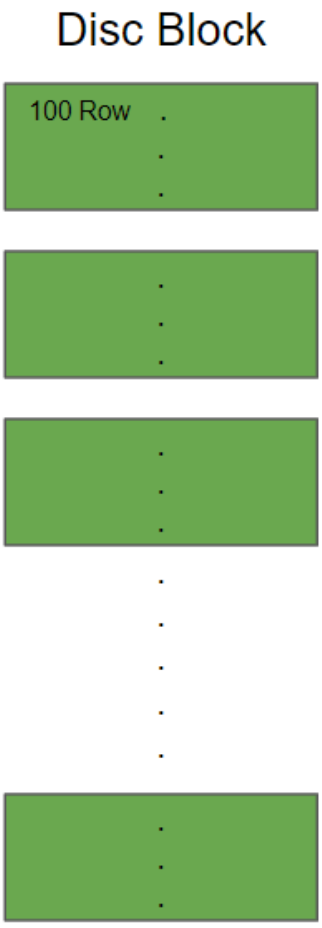
- The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.
  Note: The data may or may not be stored in sorted order.
- The second column is the **Data 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.

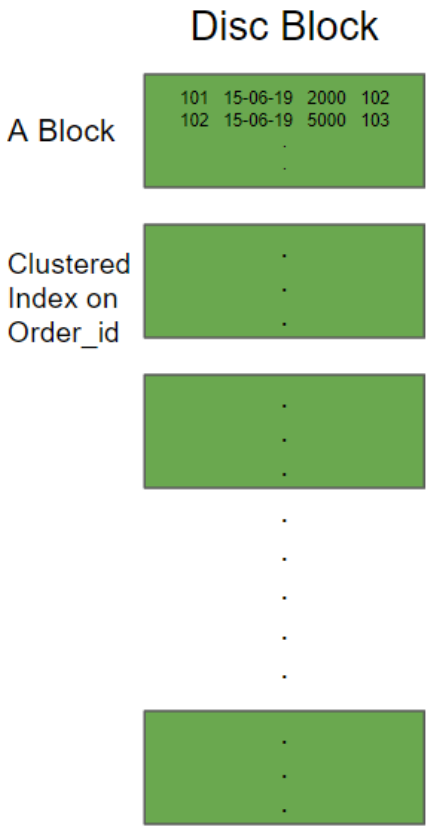
Let's look at an example:



Order

Order_id	Order_date	Cost	Customer_id
101	15-06-19	2000	102

The DBMS uses hard disk to store all the records in the above databse. As we know that the access time of the hard disk is very slow, searching for anything in such huge databases could cost performance issue. Moreover, searchin for a repeated item in the database could lead to a greater consumption of time as this will require searching for all the items in every block.
Suppose there are 100 rows in each block, so when a customer id is searched for in the database, will take too much of time. The hard disk does not store the data in a particular order.
One solution to this problem is to arrange the indexes in a database in sorted order so that any looked up item can be found easily using Binary Search. This creation of orders to store the indexes is called clustered indexing.



Order

Order_id	Order_date	Cost	Customer_id
101	15-06-19	2000	102
102	15-06-19	5000	103
105	15-06-19	6000	101
106	15-06-19	8000	102

This sort of indexing is called as the clustered indexing and this can be applied to any attribute. It is not necessary that the focus of order should be any key. When the primary key is used to order the data in a heap, it is called as Primary Indexing.

Although we have solved one problem of time cost, another problem arises when there are multiple values that are needed to be fetched using an id. The clustered indexing allows the physical arrangement of the data only on one key, therefore multiple fetching will raise a problem. To solve this we have another method of indexing called the non-clustered indexing or secondary indexing.

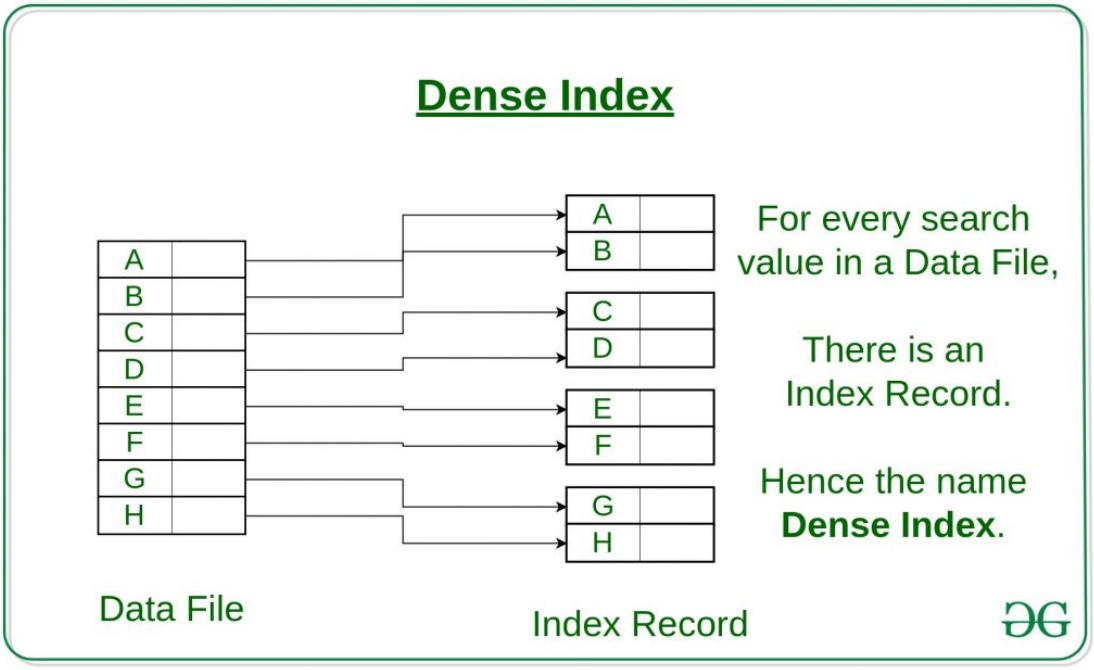
To make the understanding clearer, let's take an example of a book, where the index or the content page in which all the chapter numbers are mentioned are organised in a clustered index manner. The dictionary sort of thing mentioned at the end of the book, that provides a reference to the words in various chapters along with the page numbers are arranged in a non-clustered manner.

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:

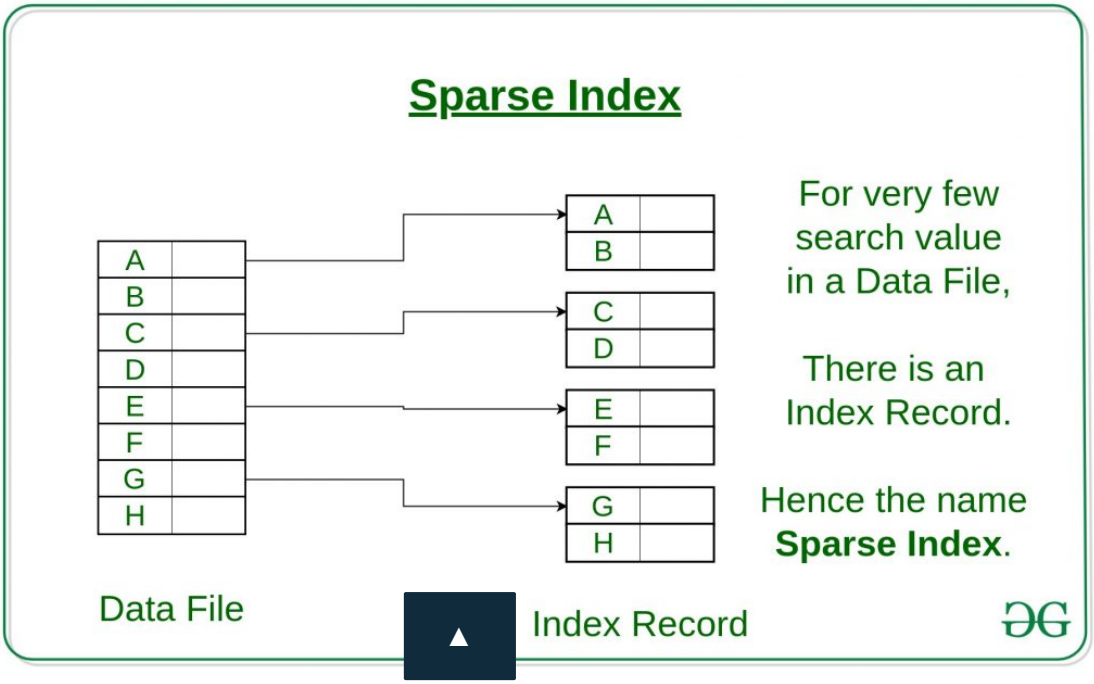
Dense Index:

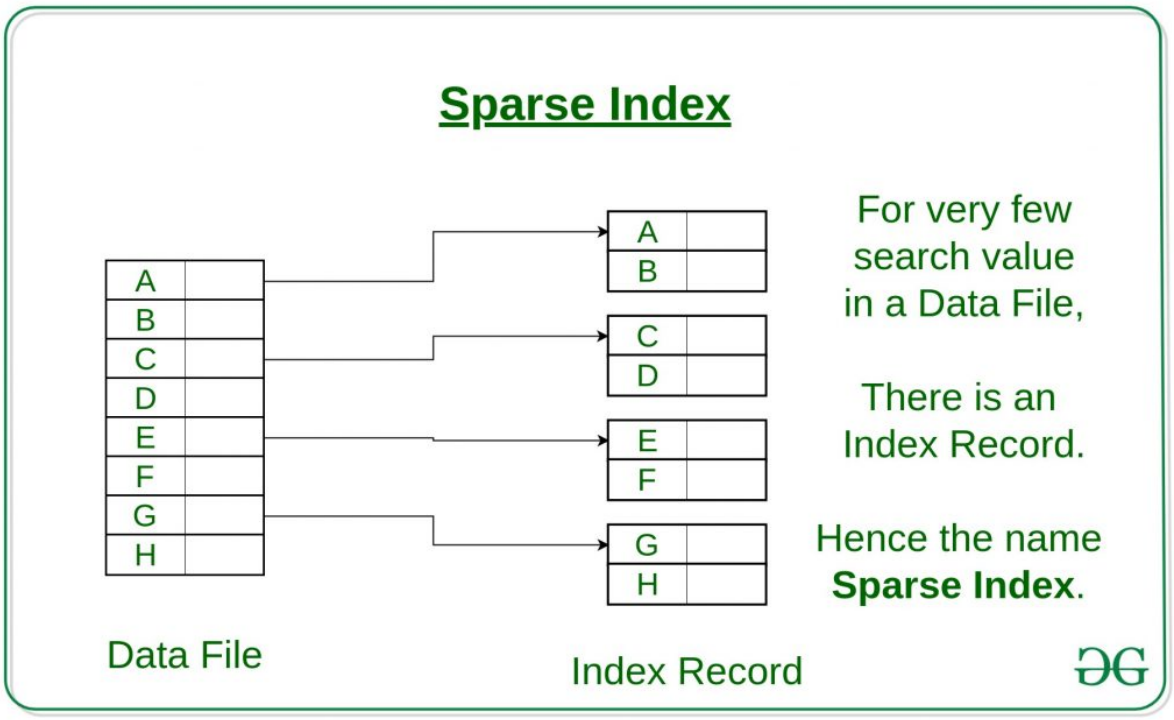
- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value.



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.





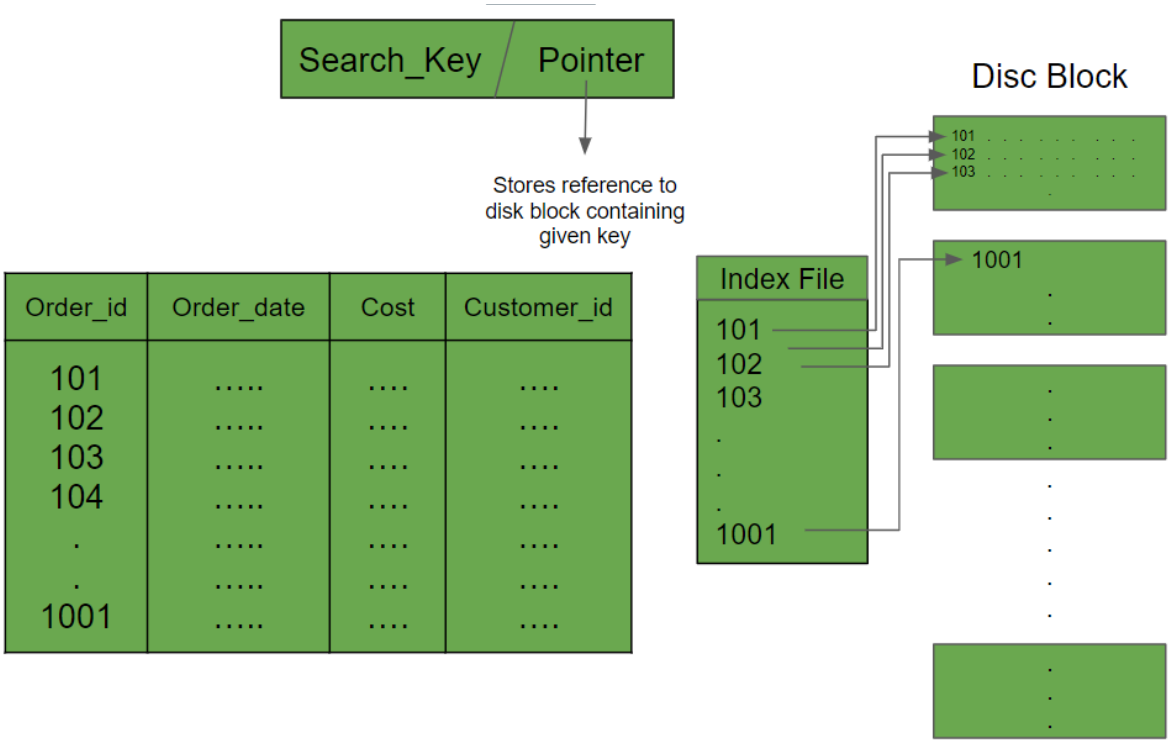
2. **Hash File organization:** Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function called a hash function.

There are primarily two methods of indexing:

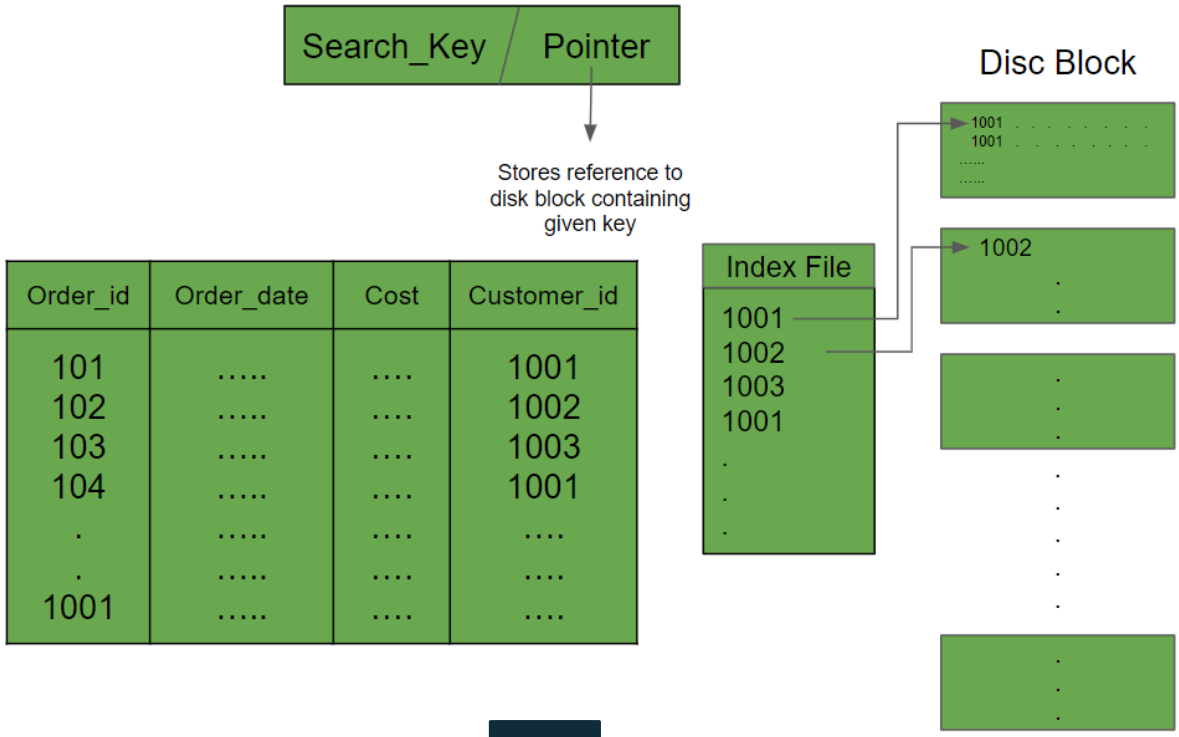
- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

1. **Clustered Indexing** 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.

   Let's look at this online retailing database, where each Order_id is stored along with Order_date, Cost and Customer_ID.

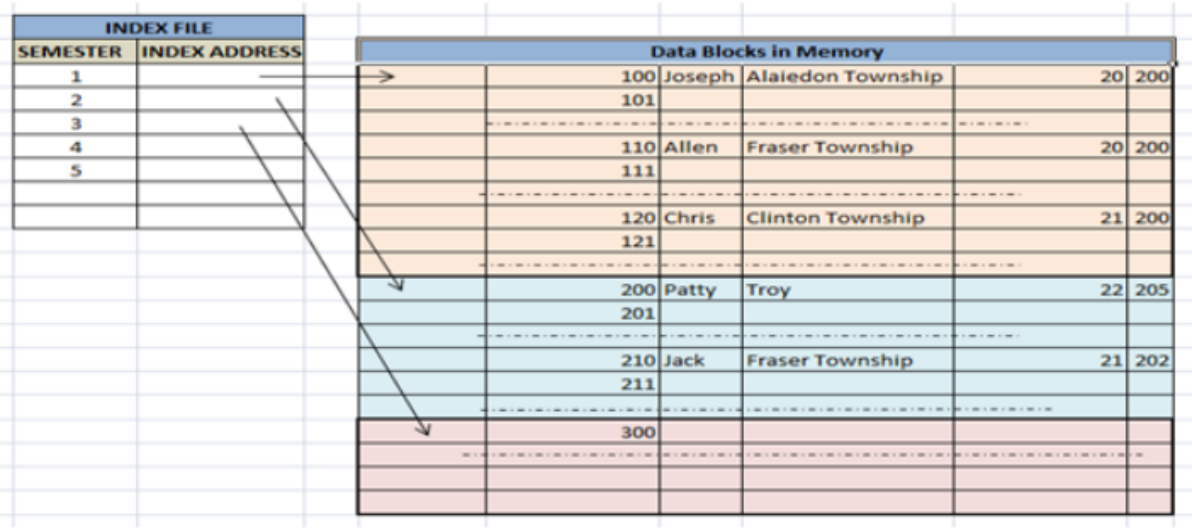


Working is easy to understand. The DBMS maintains an extra index file in clustered indexing. It has entries in the form of Search_key and Pointer, which stores the reference to disk block containing the given key. Here we have assumed that the Order_id is our Primary key on which indexing is done. So the Seach_key consists of the Order_id. The index file can store the data in either the Order Index File or Hashing Index File as shown earlier. The order indexing can be done in either Dense indexing format or sparse indexing format as we have learnt earlier. Dense Indexing saves the time of search and Sparse Index saves index file size. Let's look at another type where Clustered indexing is done on Customer_ID instead of Order_id.



Although there can be multiple kleys in CUstomer_ID and Disk Blocks, but there is only one entry in the Index File.

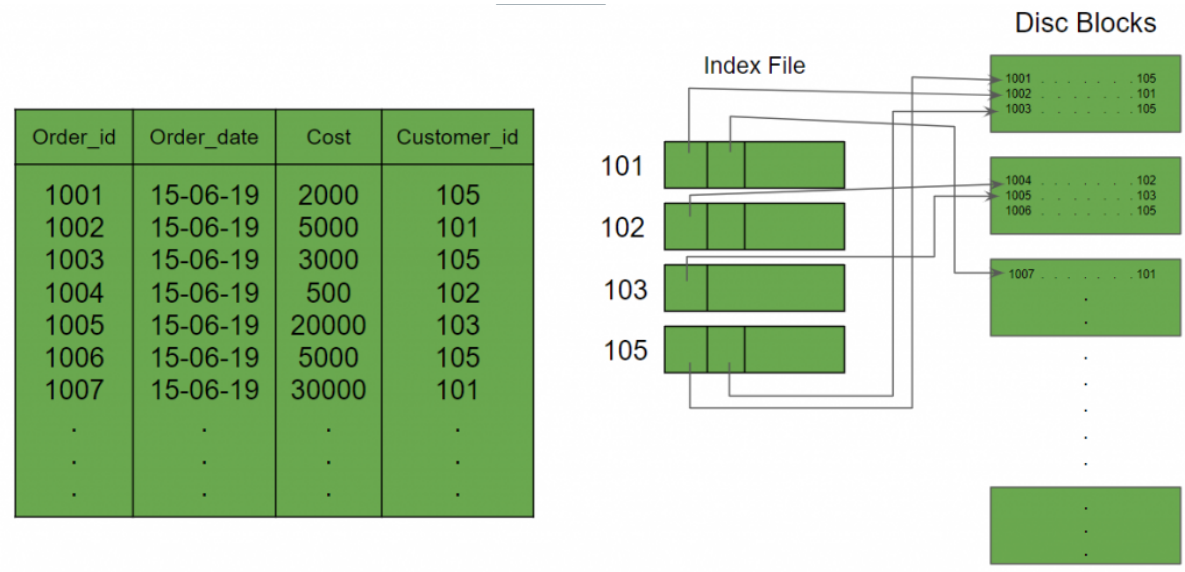
For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.



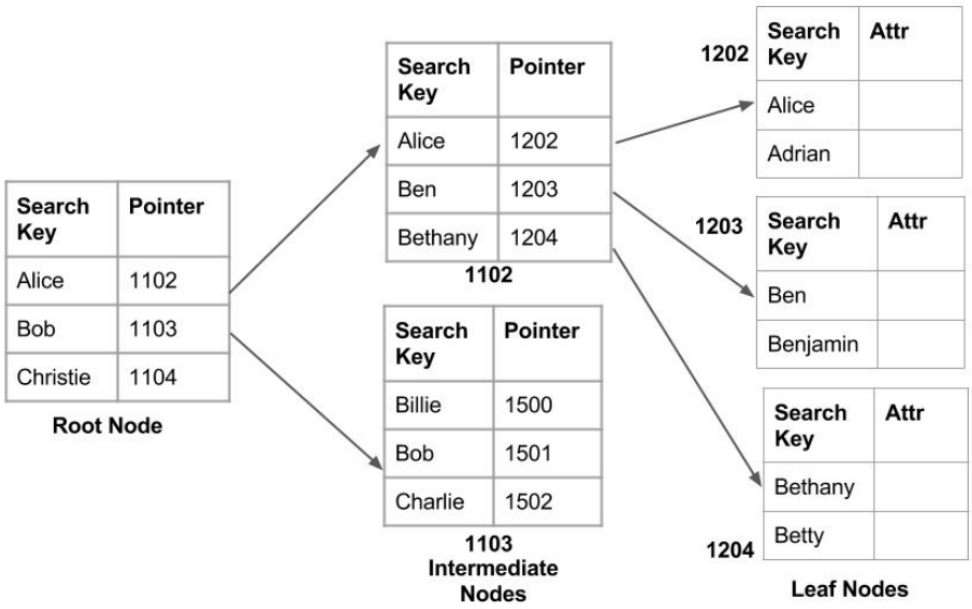
Clustered index sorted according to first name (Search key)

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.
   Let's look at the same order table and see how the data is arranged in a non-clustered way in an index file.

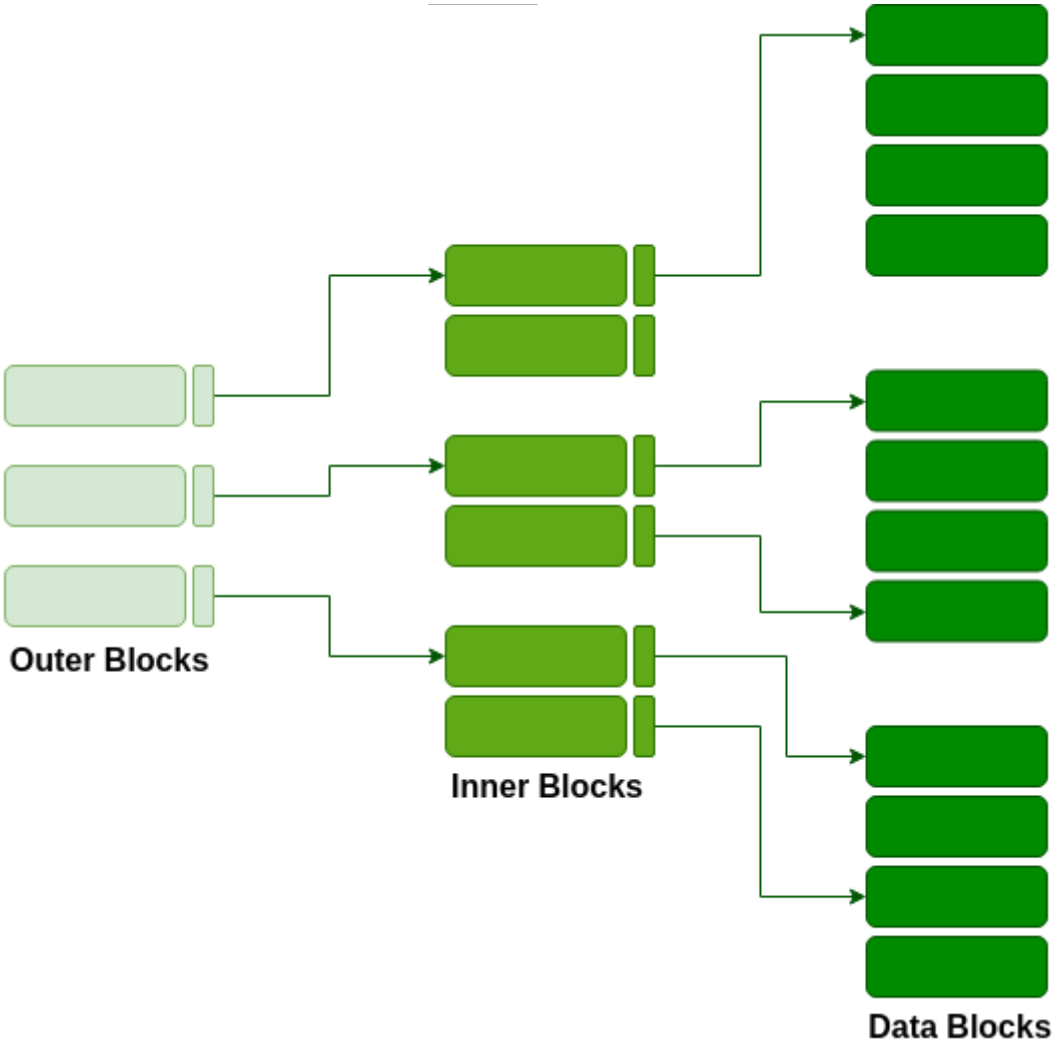


Here we did the indexing according to the Customer_ID. In this, the index file has multiple blocks to point to the multiple Customer_ID. The non-clustered index can only be arranged in a dense manner and not in a sparse manner. We have used an ordered form of indexing to understand this example. A secondary index is created only on the most frequently searched attribute.



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

Introduction:

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:

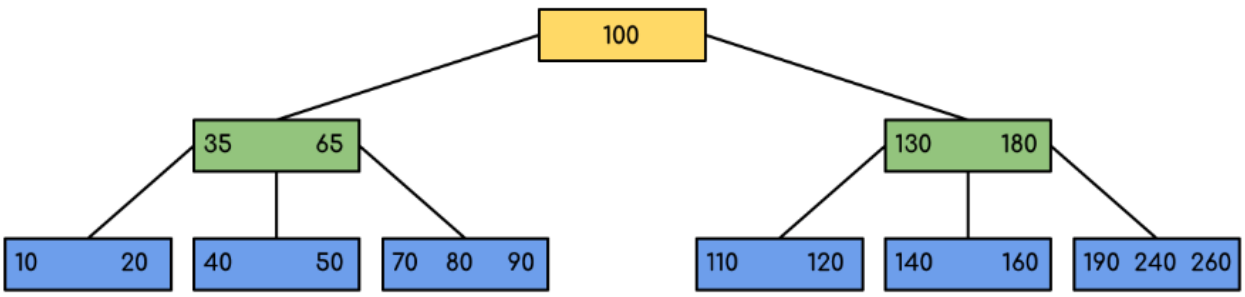
Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

"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 (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most 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).

Following is an example of B-Tree of minimum order 5. Note that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.

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:
   $h_{\min} = l\lceil \log \rceil_m (n+1) r\lceil - \rceil 1$

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:

$h_{\max} = \left\lfloor \log_t \frac{n+1}{2} \right\rfloor$ and $t = l\left\lceil \frac{m}{2} \right\rceil r\lceil \rceil$

Traversel 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.

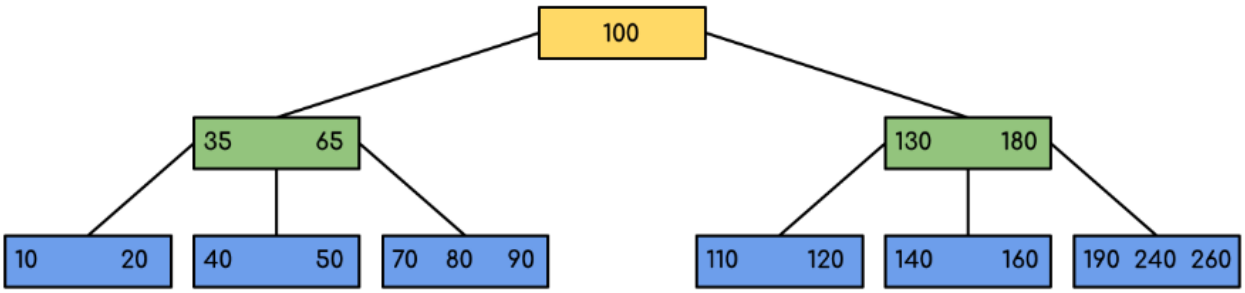
Search Operation in B-Tree:

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.

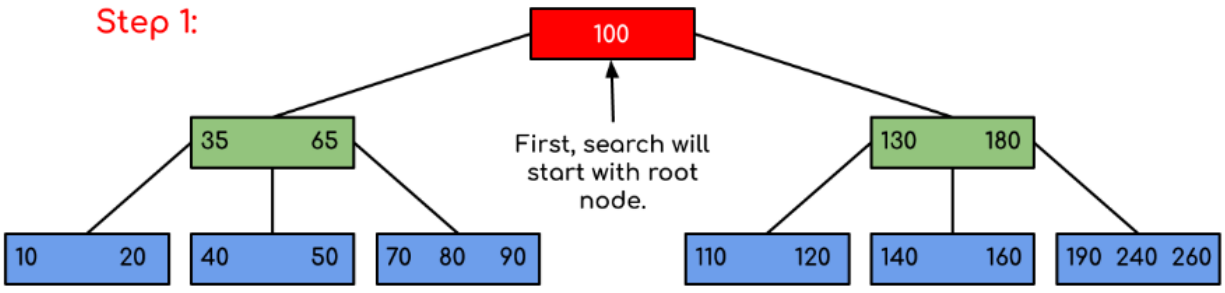
Logic:

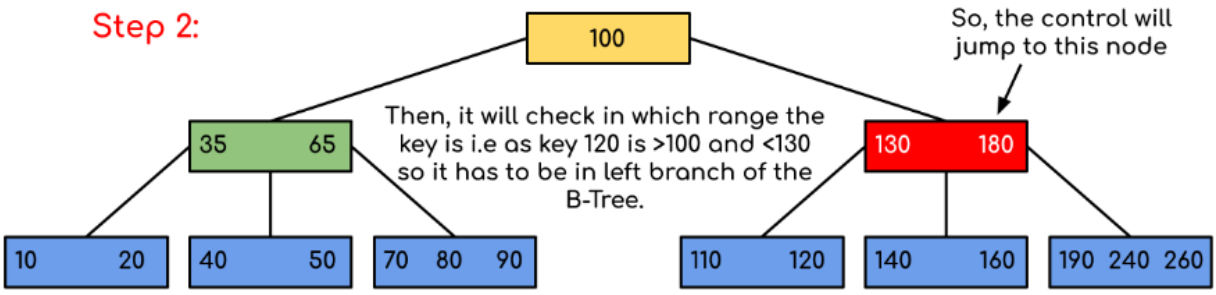
Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimised as if the key value is not present in the range of parent then the key is present in another branch. As these values limit the search they are also known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

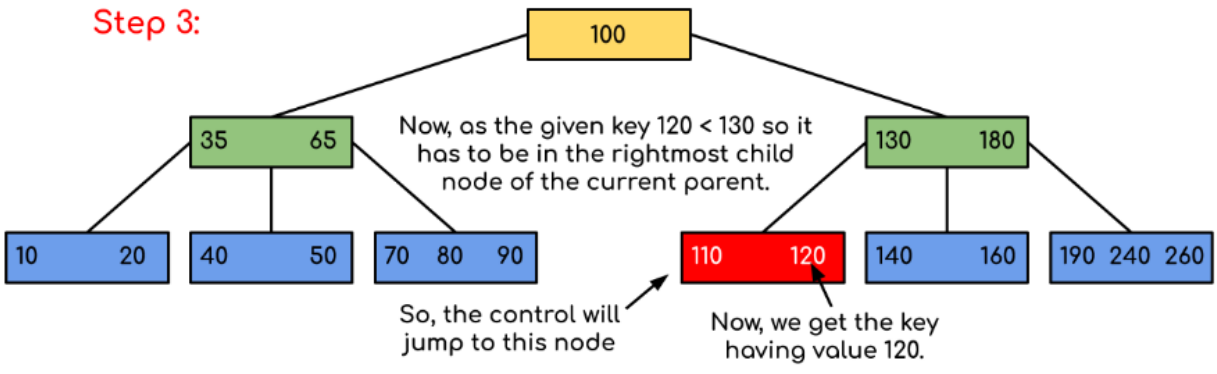
Example: Searching 120 in the given B-Tree.



Solution:







In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically and therefore the control flow will go similarly as shown within the above example.

C++

```
1
2
3  // C++ implemntation of search() and traverse() methods
4  #include<iostream>
5  using namespace std;
6
7  // A BTree node
8  class BTreeNode
9  {
10     int *keys;  // An array of keys
11     int t;      // Minimum degree (defines the range fo
12     BTreeNode **C; // An array of child pointers
13     int n;     // Current number of keys
14     bool leaf; // Is true when node is leaf. Otherwise
15 public:
16     BTreeNode(int _t, bool _leaf);   // Constructor
17
18     // A function to traverse all nodes in a subtree ro
19     void traverse();
20
21     // A function to search a key in the subtree rooted
22     BTreeNode *search(int k);   // returns NULL if k is
23
24 // Make the BTree friend of this so that we can access
25 // class in BTree functions
26 friend class BTree;
27 };
28
29 // A BTree
30 class BTree
```

Run

Java

C#

Complete Code:

```
// C++ implemntation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys;  // An array of keys
    int t;      // Minimum degree (defines the range for number of ke
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf);   // Constructor

    // A function to traverse all nodes in a subtree rooted with this
    void traverse();

    // A function to search a key in the subtree rooted with this nod
    BTreeNode *search(int k);   // returns NULL if k is not present.

// Make the BTree friend of this so that we can access private member
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t;  // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {  root = NULL;  t = _t; }

    // function to traverse the tree
    void traverse()
    {  if (root != NULL) root->traverse(); }
```

```
    // function to search a key in this tree
    BTreeNode* search(int k)
    {  return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;
```

```
    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If the key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}
```

```
// Java program to illustrate the sum of two numbers

// A BTree
class Btree {
    public BTreeNode root; // Pointer to root node
    public int t; // Minimum degree

    // Constructor (Initializes tree as empty)
    Btree(int t) {
        this.root = null;
        this.t = t;
    }

    // function to traverse the tree
    public void traverse() {
        if (this.root != null)
            this.root.traverse();
        System.out.println();
    }

    // function to search a key in this tree
    public BTreeNode search(int k) {
        if (this.root == null)
            return null;
        else
            return this.root.search(k);
    }
}

// A BTree node
class BTreeNode {
    int[] keys; // An array of keys
    int t; // Minimum degree (d[illegible] the range for number of keys)
    BTreeNode[] C; // An array [illegible]ld pointers
```

```
    int n; // Current number of keys
    boolean leaf; // Is true when node is leaf. Otherwise false

    // Constructor
    BTreeNode(int t, boolean leaf) {
        this.t = t;
        this.leaf = leaf;
        this.keys = new int[2 * t - 1];
        this.C = new BTreeNode[2 * t];
        this.n = 0;
    }

    // A function to traverse all nodes in a subtree rooted with this
    public void traverse() {

        // There are n keys and n+1 children, travers through n keys
        // and first n children
        int i = 0;
        for (i = 0; i < this.n; i++) {

            // If this is not leaf, then before printing key[i],
            // traverse the subtree rooted with child C[i].
            if (this.leaf == false) {
                C[i].traverse();
            }
            System.out.print(keys[i] + " ");
        }

        // Print the subtree rooted with last child
        if (leaf == false)
            C[i].traverse();
    }

    // A function to search a key in the subtree rooted with this nod
    BTreeNode search(int k) { // returns NULL if k is not present.

        // Find the first key greater than or equal to k
        int i = 0;
        while (i < n && k > keys[i])
            i++;

        // If the found key is equal to k, return this node
        if (keys[i] == k)
            return this;

        // If the key is not found here and this is a leaf node
        if (leaf == true)
            return null;
```

```
        // Go to the appropriate child
        return C[i].search(k);

    }
}
```

```
// C# program to illustrate the sum of two numbers
using System;

// A BTree
class Btree
{
  public BTreeNode root; // Pointer to root node
  public int t; // Minimum degree

  // Constructor (Initializes tree as empty)
  Btree(int t)
  {
    this.root = null;
    this.t = t;
  }

  // function to traverse the tree
  public void traverse()
  {
    if (this.root != null)
      this.root.traverse();
    Console.WriteLine();
  }

  // function to search a key in this tree
  public BTreeNode search(int k)
  {
    if (this.root == null)
      return null;
    else
      return this.root.search(k);
  }
}

// A BTree node
class BTreeNode
{
  int[] keys; // An array of keys
  int t; // Minimum degree (def[illegible]he range for number of keys)
  BTreeNode[] C; // An array of[illegible] pointers
```

```
int n; // Current number of keys
bool leaf; // Is true when node is leaf. Otherwise false

// Constructor
BTreeNode(int t, bool leaf) {
  this.t = t;
  this.leaf = leaf;
  this.keys = new int[2 * t - 1];
  this.C = new BTreeNode[2 * t];
  this.n = 0;
}

// A function to traverse all nodes in a subtree rooted with this n
public void traverse() {

  // There are n keys and n+1 children, travers through n keys
  // and first n children
  int i = 0;
  for (i = 0; i < this.n; i++) {

    // If this is not leaf, then before printing key[i],
    // traverse the subtree rooted with child C[i].
    if (this.leaf == false) {
      C[i].traverse();
    }
    Console.Write(keys[i] + " ");
  }

  // Print the subtree rooted with last child
  if (leaf == false)
    C[i].traverse();
}

// A function to search a key in the subtree rooted with this node.
public BTreeNode search(int k) { // returns NULL if k is not presen

  // Find the first key greater than or equal to k
  int i = 0;
  while (i < n && k > keys[i])
    i++;

  // If the found key is equal to k, return this node
  if (keys[i] == k)
    return this;

  // If the key is not found here and this is a leaf node
  if (leaf == true)
    return null;
```

```
    // Go to the appropriate child
    return C[i].search(k);

  }
}

// This code is contributed by Rajput-Ji
```

The above code doesn't contain the driver program. We will be covering the complete program in our next post on B-Tree Insertion.
There are two conventions to define a B-Tree, one is to define by minimum degree (followed in Cormen book), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion

B-Tree Insertion
B-Tree Deletion

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

- 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 le[illegible]s must necessarily store all the key values along with their corresponding da[illegible]ters 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.

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.

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

1. Each internal node is of the form :
   $<P_1, K_1, P_2, K_2, ..., P_{c-1}, K_{c-1}, P_c>$
   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

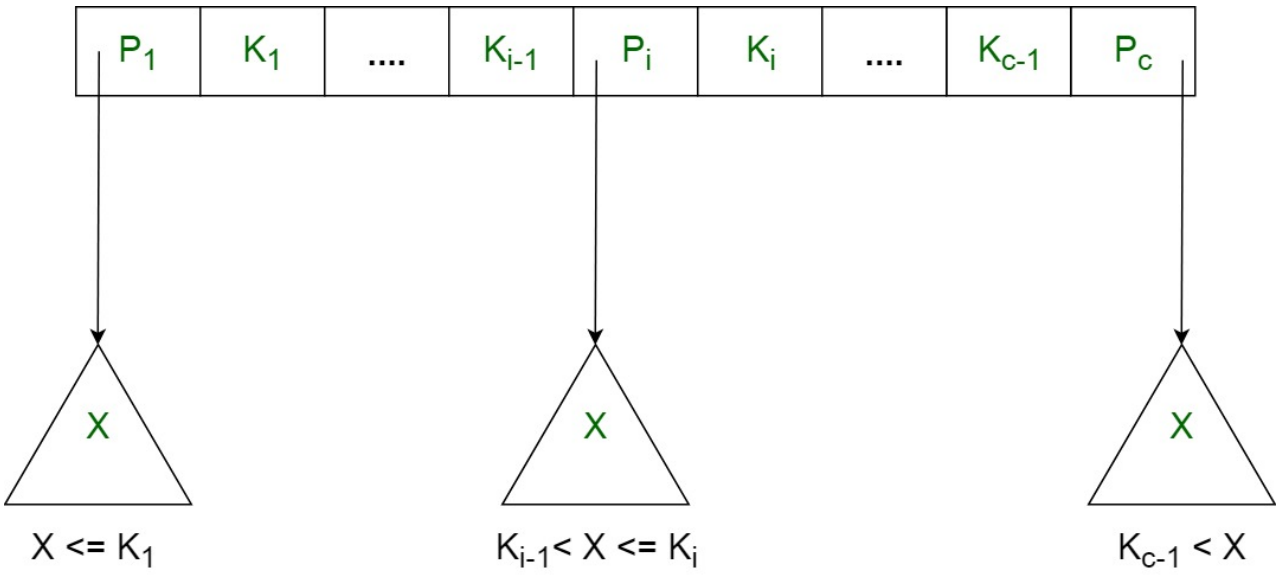


Diagram-I

The structure of the leaf nodes of a B+ t rder '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.

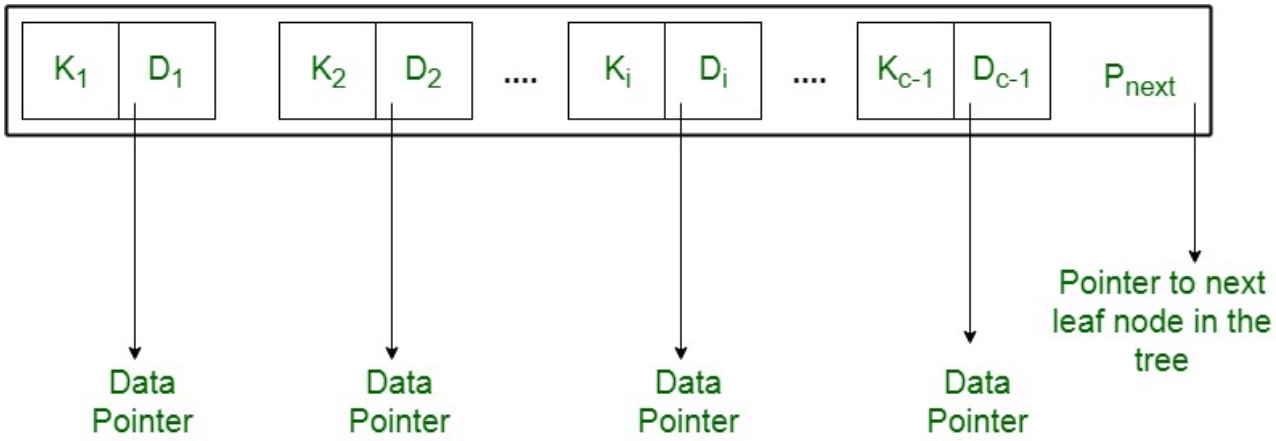
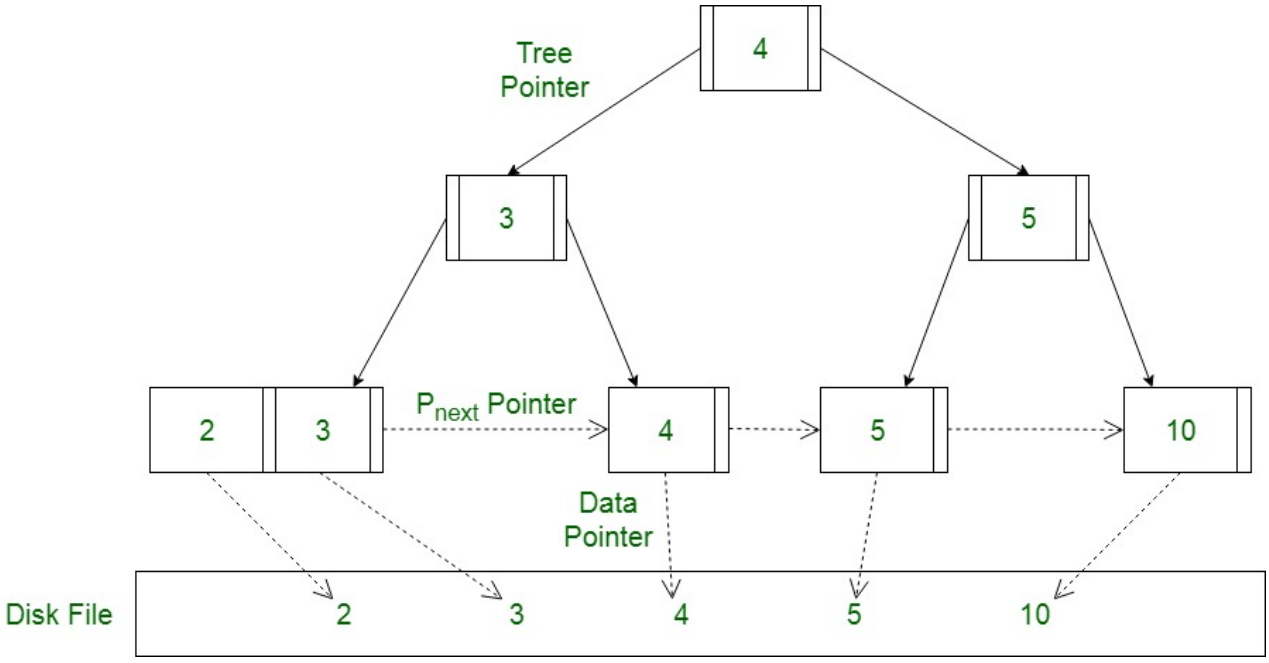


Diagram-II

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 ▲ 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.