INDEXING

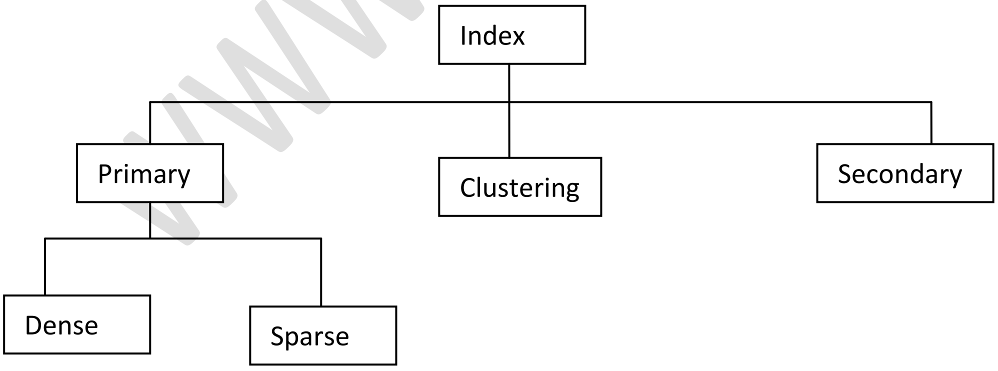
What is an Index?

An index is a small table having only two columns. The first column contains a copy of the primary or candidate key of a table and the second column contains a set of pointers holding the address of the disk block where that particular key value can be found.

The advantage of using index lies in the fact is that index makes search operation perform very fast. Suppose a table has a several rows of data, each row is 20 bytes wide. If you want to search for the record number 100, the management system must thoroughly read each and every row and after reading 99x20 = 1980 bytes it will find record number 100. If we have an index, the management system starts to search for record number 100 not from the table, but from the index. The index, containing only two columns, may be just 4 bytes wide in each of its rows. After reading only 99x4 = 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. However, the advantages are so huge that these disadvantages can be considered negligible.

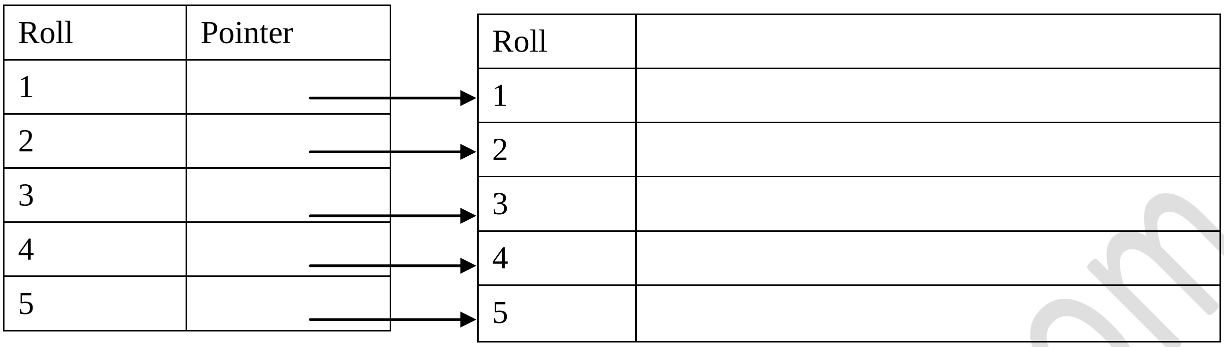
Types of Index



Primary Index

In primary index, there is a one-to-one relationship between the entries in the index table and the records in the main table. Primary index can be of two types

Dense Primary Index: the number of entries in the index table is the same as the number of entries in the main table. In other words, each and every record in the main table has an entry in the index.

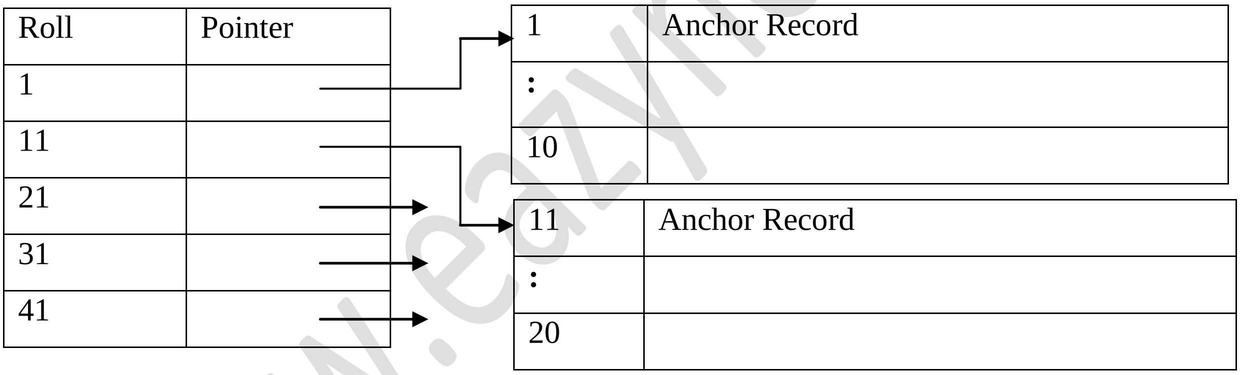


Roll	Pointer
1	
2	
3	
4	
5	

Roll	
1	
2	
3	
4	
5	

Sparse or Non-Dense Primary Index:

For large tables the Dense Primary Index itself begins to grow in size. To keep the size of the index smaller, instead of pointing to each and every record in the main table, the index points to the records in the main table in a gap. See the following example.



Roll	Pointer
1	
11	
21	
31	
41	

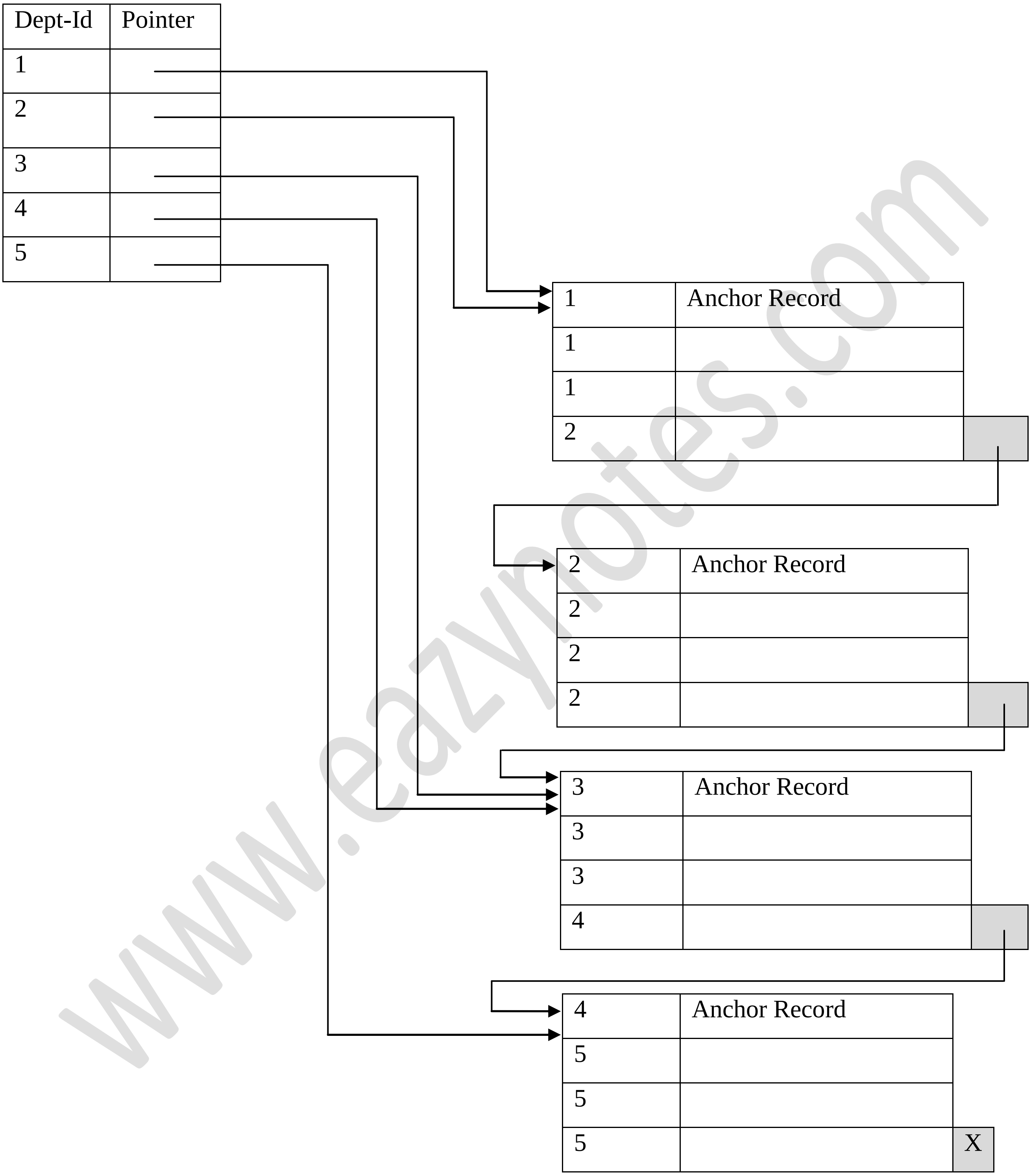
1	Anchor Record
:	
10	

11	Anchor Record
:	
20	

As you can see, the data blocks have been divided in to several blocks, each containing a fixed number of records (in our case 10). The pointer in the index table points to the first record of each data block, which is known as the Anchor Record for its important function. If you are searching for roll 14, the index is first searched to find out the highest entry which is smaller than or equal to 14. We have 11. The pointer leads us to roll 11 where a short sequential search is made to find out roll 14.

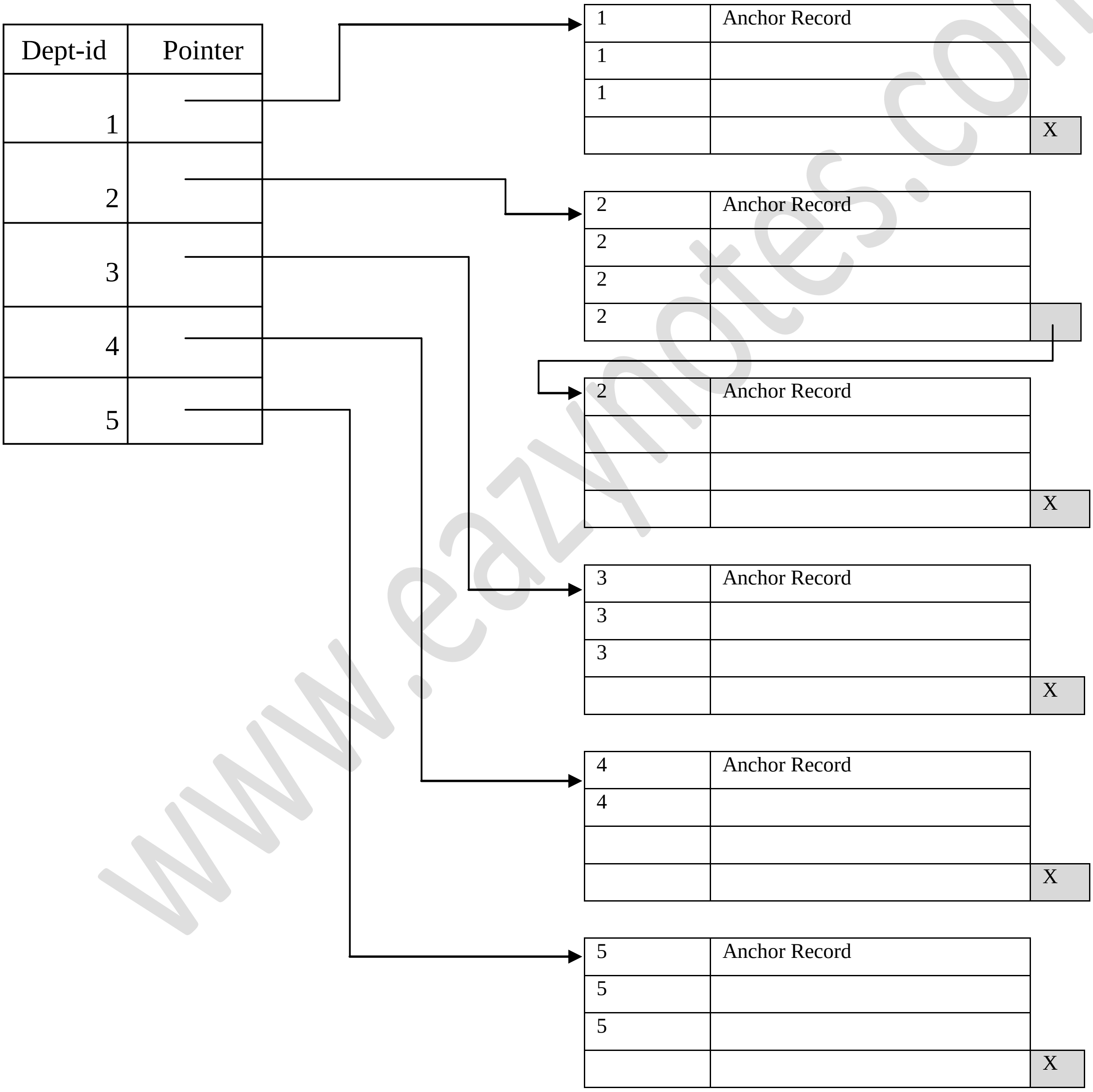
Clustering Index

It may happen sometimes that we are asked to create an index on a non-unique key, such as Dept-id. There could be several employees in each department. Here we use a clustering index, where all employees belonging to the same Dept-id are considered to be within a single cluster, and the index pointers point to the cluster as a whole.



Let us explain this diagram. The disk blocks contain a fixed number of records (in this case 4 each). The index contains entries for 5 separate departments. The pointers of these entries point to the anchor record of the block where the first of the Dept-id in the cluster can be found. The blocks themselves may point to the anchor record of the next block in case a cluster overflows a block size. This can be done using a special pointer at the end of each block (comparable to the next pointer of the linked list organization).

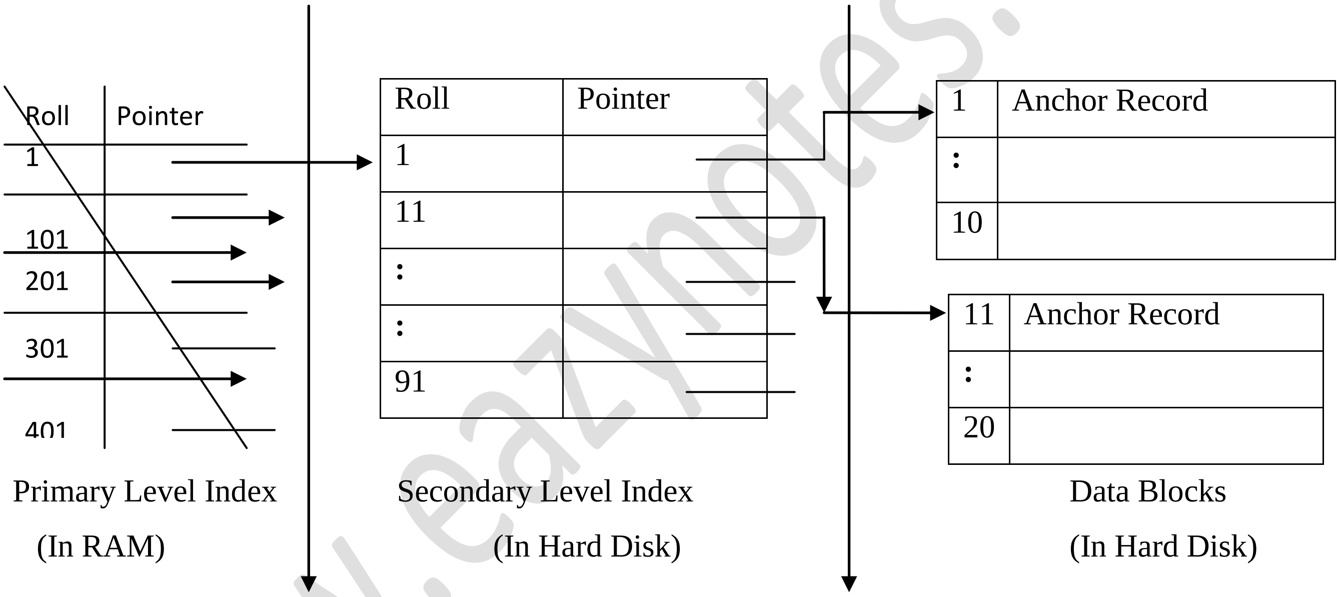
The previous scheme might become a little confusing because one disk block might be shared by records belonging to different cluster. A better scheme could be to use separate disk blocks for separate clusters. This has been explained in the next page.



In this scheme, as you can see, we have used separate disk block for the clusters. The pointers, like before, have pointed to the anchor record of the block where the first of the cluster entries would be found. The block pointers only come into action when a cluster overflows the block size, as for Dept-id 2. This scheme takes more space in the memory and the disk, but the organization in much better and cleaner looking.

Secondary Index

While creating the index, generally the index table is kept in the primary memory (RAM) and the main table, because of its size is kept in the secondary memory (Hard Disk). Theoretically, a table may contain millions of records (like the telephone directory of a large city), for which even a sparse index becomes so large in size that we cannot keep it in the primary memory. And if we cannot keep the index in the primary memory, then we lose the advantage of the speed of access. For very large table, it is better to organize the index in multiple levels. See the following example.



Primary Level Index
(In RAM)

Secondary Level Index
(In Hard Disk)

Data Blocks
(In Hard Disk)

In this scheme, the primary level index, (created with a gap of 100 records, and thereby smaller in size), is kept in the RAM for quick reference. If you need to find out the record of roll 14 now, the index is first searched to find out the highest entry which is smaller than or equal to 14. We have 1. The adjoining pointer leads us to the anchor record of the corresponding secondary level index, where another similar search is conducted. This finally leads us to the actual data block whose anchor record is roll 11. We now come to roll 11 where a short sequential search is made to find out roll 14.

Multilevel Index

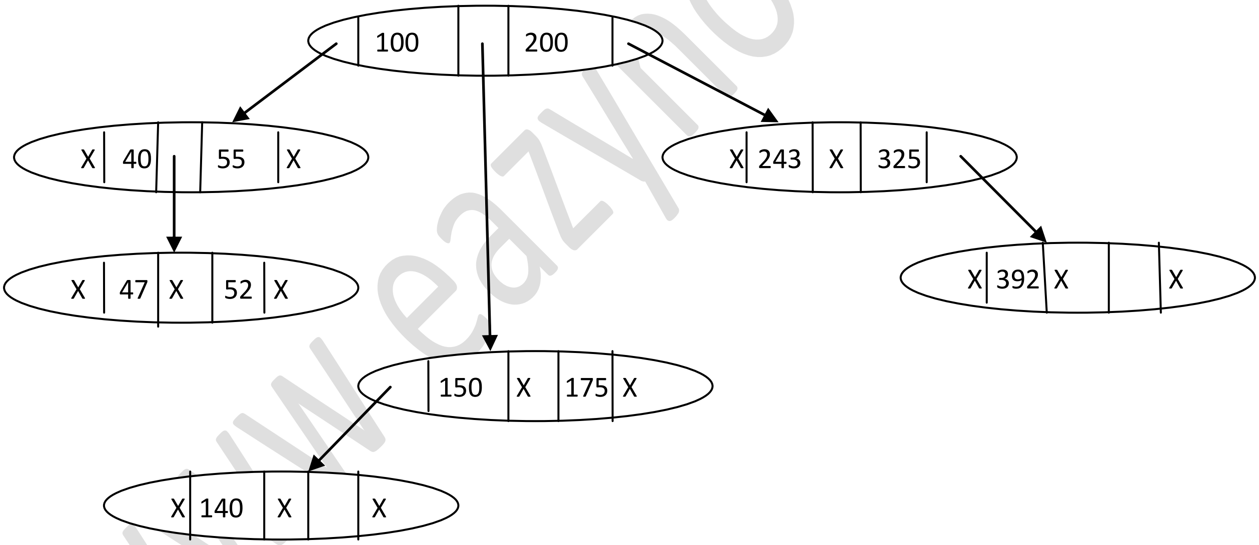
The Multilevel Index is a modification of the secondary level index system. In this system we may use even more number of levels in case the table is even larger.

Index in a Tree like Structure

We can use tree-like structures as index as well. For example, a binary search tree can also be used as an index. If we want to find out a particular record from a binary search tree, we have the added advantage of binary search procedure, that makes searching be performed even faster. A binary tree can be considered as a **2-way Search Tree**, because it has two pointers in each of its nodes, thereby it can guide you to two distinct ways. Remember that for every node storing 2 pointers, the number of value to be stored in each node is one less than the number of pointers, i.e. each node would contain 1 value each.

M-Way Search Tree

The abovementioned concept can be further expanded with the notion of the m-Way Search Tree, where m represents the number of pointers in a particular node. If m = 3, then each node of the search tree contains 3 pointers, and each node would then contain 2 values.
A sample m-Way Search Tree with m = 3 is given in the following.



The above notes are submitted by:

Sabyasachi De

MCA

sabyasachide@yahoo.com