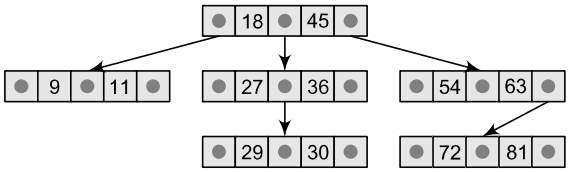


Figure 11.3 M-way search tree of order 3

In an M-way search tree, it is not compulsory that every node has exactly `M-1` values and `M` sub-trees. Rather, the node can have anywhere from 1 to `M-1` values, and the number of sub-trees can vary from 0 (for a leaf node) to `i + 1`, where `i` is the number of key values in the node. `M` is thus a fixed upper limit that defines how many key values can be stored in the node.

Consider the M-way search tree shown in Fig. 11.3. Here `M = 3`. So a node can store a maximum of two key values and can contain pointers to three sub-trees.

In our example, we have taken a very small value of `M` so that the concept becomes easier for the reader, but in practice, `M` is usually very large. Using a 3-way search tree, let us lay down some of the basic properties of an M-way search tree.

- Note that the key values in the sub-tree pointed by P_0 are less than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_1 are less than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are less than K_i, where $0 \leq i \leq n-1$.
- Note that the key values in the sub-tree pointed by P_1 are greater than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_2 are greater than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are greater than K_{i-1}, where $0 \leq i \leq n-1$.

In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

11.2 B TREES

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order `m` can have a maximum of `m-1` keys and `m` pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. A B tree of order `m` (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

1. Every node in the B tree has at most (maximum) `m` children.
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) `m/2` children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.
3. The root node has at least two children if it is not a terminal (leaf) node.
4. All leaf nodes are at the same level.

An internal node in the B tree can have `n` number of children, where $0 \leq n \leq m$. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least `m/2` children. As B tree of order 4 is given in Fig. 11.4.

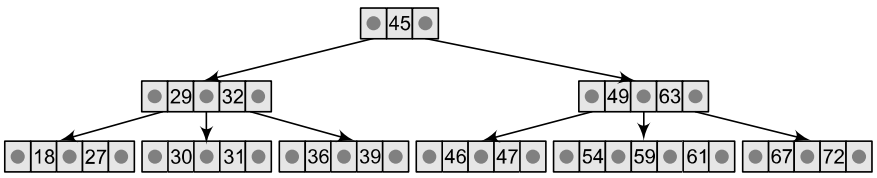


Figure 11.4 B tree of order 4

While performing insertion and deletion operations in a B tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split. We will discuss search, insertion, and deletion operations in this section.

11.2.1 Searching for an Element in a B Tree

Searching for an element in a B tree is similar to that in binary search trees. Consider the B tree given in Fig. 11.4. To search for 59, we begin at the root node. The root node has a value 45 which is less than 59. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since $49 \leq 59 \leq 63$, we traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful.

Take another example. If you want to search for 9, then we traverse the left sub-tree of the root node. The left sub-tree has two key values, 29 and 32. Again, we traverse the left sub-tree of 29. We find that it has two key values, 18 and 27. There is no left sub-tree of 18, hence the value 9 is not stored in the tree.

Since the running time of the search operation depends upon the height of the tree, the algorithm to search for an element in a B tree takes $O(\log_t n)$ time to execute.

11.2.2 Inserting a New Element in a B Tree

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

1. Search the B tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is, it contains less than `m-1` key values, then insert the new element in the node keeping the node's elements ordered.
3. If the leaf node is full, that is, the leaf node already contains `m-1` key values, then
   (a) insert the new value in order into the existing set of keys,
   (b) split the node at its median into two nodes (note that the split nodes are half full), and
   (c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Example 11.1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.

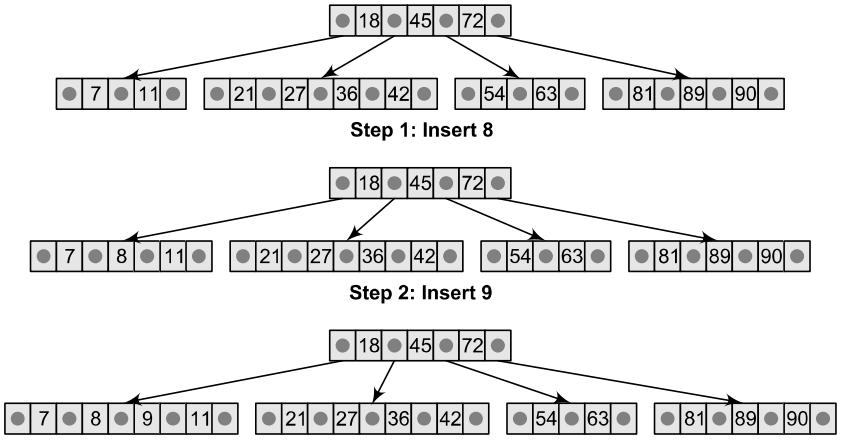


Figure 11.5(a)

Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

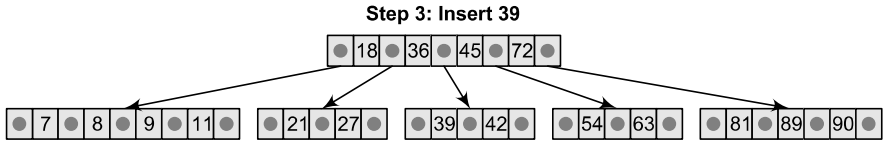


Figure 11.5(b)

Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.

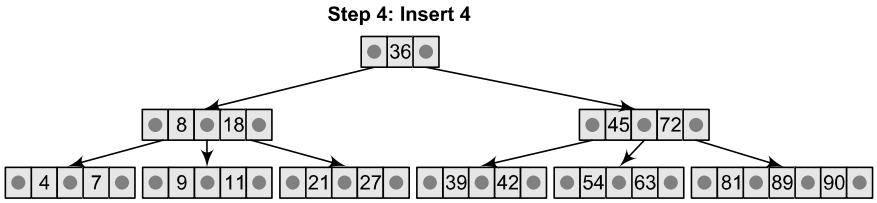


Figure 11.5(c) B tree

11.2.3 Deleting an Element from a B Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.
2. If the leaf node contains more than the minimum number of key values (more than `m/2` elements), then delete the value.
3. Else if the leaf node does not contain `m/2` elements, then fill the node by taking an element either from the left or from the right sibling.
   (a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
   (b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, `m`). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

Example 11.2 Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it (Fig. 11.6(a)).

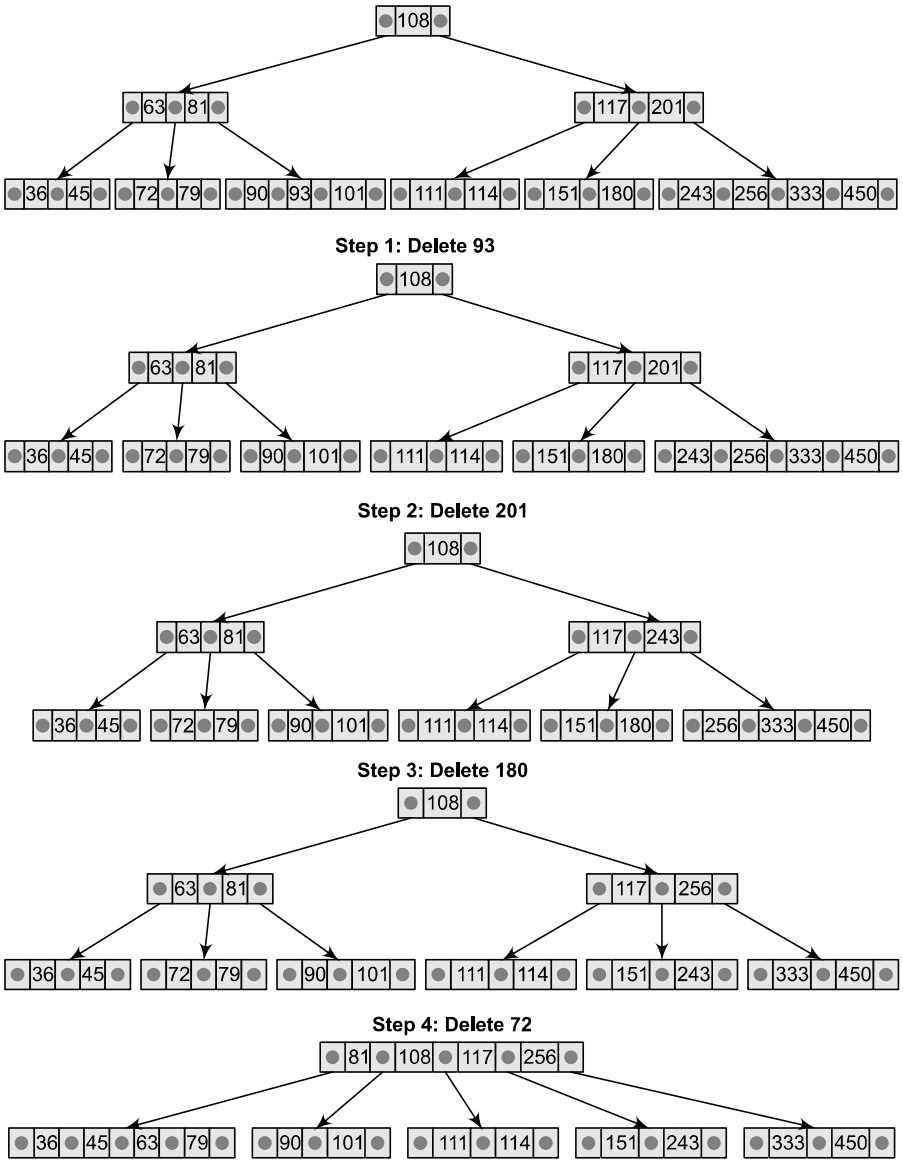


Figure 11.6 B tree

Example 11.3 Consider the B tree of order 3 given below and perform the following operations: (a) insert 121, 87 and then (b) delete 36, 109.

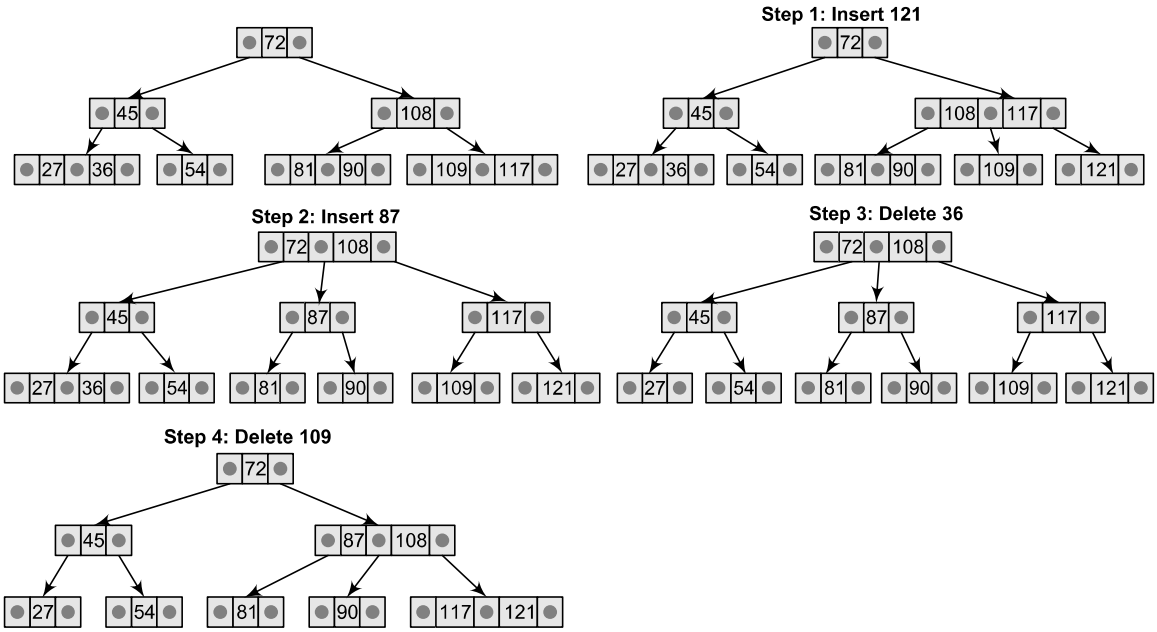
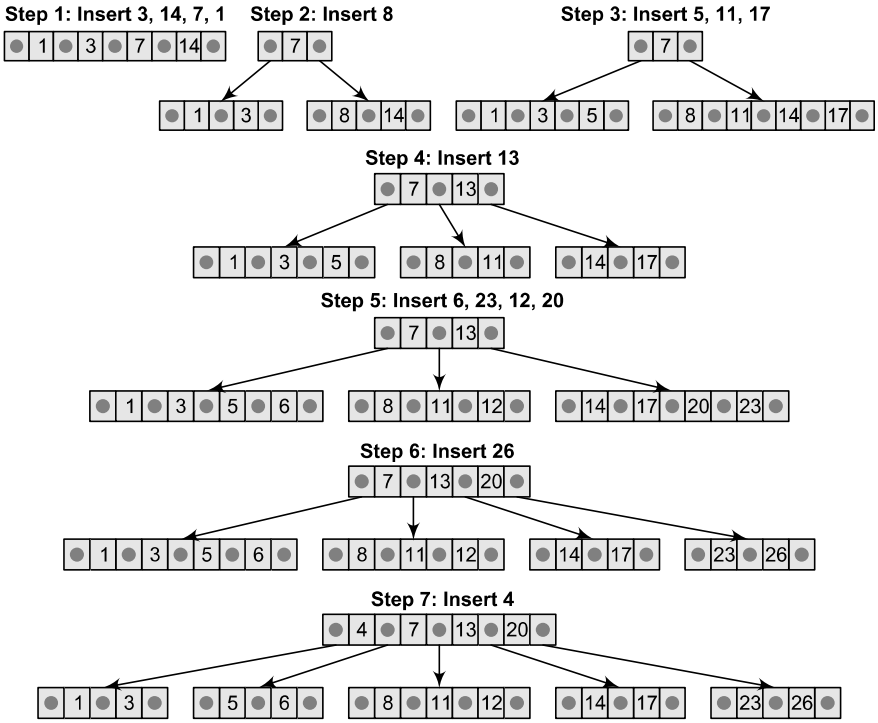


Figure 11.7 B tree

Example 11.4 Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.



(Contd)

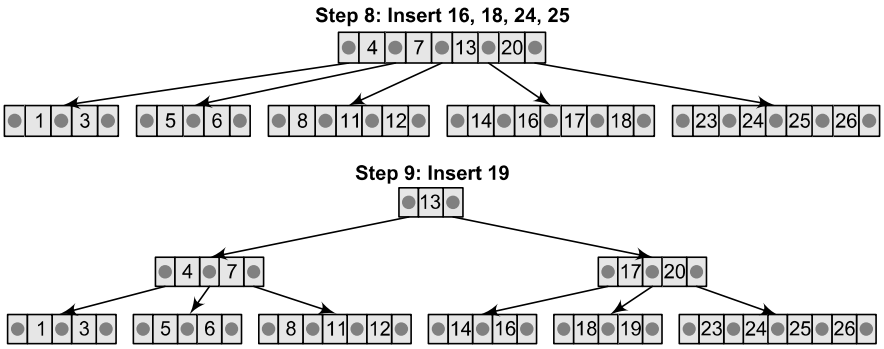


Figure 11.8 B tree

11.2.4 Applications of B Trees

A database is a collection of related data. The prime reason for using a database is that it stores organized data to facilitate its users to update, retrieve, and manage the data. The data stored in the database may include names, addresses, pictures, and numbers. For example, a teacher may wish to maintain a database of all the students that includes the names, roll numbers, date of birth, and marks obtained by every student.

Nowadays, databases are used in every industry to store hundreds of millions of records. In the real world, it is not uncommon for a database to store gigabytes and terabytes of data. For example, a telecommunication company maintains a customer billing database with more than 50 billion rows that contains terabytes of data.

We know that primary memory is very expensive and is capable of storing very little data as compared to secondary memory devices like magnetic disks. Also, RAM is volatile in nature and we cannot store all the data in primary memory. We have no other option but to store data on secondary storage devices. But accessing data from magnetic disks is 10,000 to 1,000,000 times slower than accessing it from the main memory. So, B trees are often used to index the data and provide fast access.

Consider a situation in which we have to search an un-indexed and unsorted database that contains `n` key values. The worst case running time to perform this operation would be `O(n)`. In contrast, if the data in the database is indexed with a B tree, the same search operation will run in `O(log n)`. For example, searching for a single key on a set of one million keys will at most require 1,000,000 comparisons. But if the same data is indexed with a B tree of order 10, then only 114 comparisons will be required in the worst case.

Hence, we see that indexing large amounts of data can provide significant boost to the performance of search operations.

When we use B trees or generalized M-way search trees, the value of `m` or the order of B trees is often very large. Typically, it varies from 128–512. This means that a single node in the tree can contain 127–511 keys and 128–512 pointers to child nodes.

We take a large value of `m` mainly because of three reasons:

1. Disk access is very slow. We should be able to fetch a large amount of data in one disk access.
2. Disk is a block-oriented device. That is, data is organized and retrieved in terms of blocks. So while using a B tree (generalized M-way search tree), a large value of `m` is used so that one single node of the tree can occupy the entire block. In other words, `m` represents the maximum number of data items that can be stored in a single block. `m` is maximized to speed up processing. More the data stored in a block, lesser the time needed to move it into the main memory.
3. A large value minimizes the height of the tree. So, search operation becomes really fast.