BLM267

Chapter 11: Multi-way Search Trees

Data Structures Using C, Second Edition

- Introduction
- B Trees
- B+ Trees
- 2-3 Trees

Introduction

- We have discussed that every node in a binary search tree contains one value and two pointers, left and right, which point to the node's left and right sub-trees, respectively.
- The structure of a binary search tree node is shown in Fig. 11.1.
- The same concept is used in an M-way search tree which has M – 1 values per node and M subtrees.
- In such a tree, M is called the degree of the tree. Note that in a binary search tree M = 2, so it has one value and two sub-trees.
- In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains M – 1 keys, where M > 2.
- The structure of an M-way search tree node is shown in Fig. 11.2.

Introduction

Pointer to left sub-tree	Value or Key of the node	Pointer to right sub-tree

Figure 11.1 Structure of a binary search tree node

P_0	K_0	P_1	K_1	P_2	K_2	......	P_{n-1}	K_{n-1}	P_n

Figure 11.2 Structure of an M-way search tree node

- **In the structure shown, P_0, P_1, P_2, ..., Pn are pointers to the node's sub-trees and K_0, K_1, K_2, ..., Kn–1 are the key values of the node.**
- **All the key values are stored in ascending order. That is, Ki < Ki+1 for $0 \leqslant i \leqslant n-2$.**

Introduction

- In an M-way search tree, it is not compulsory that every node has exactly M–1 values and M subtrees.
- 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.

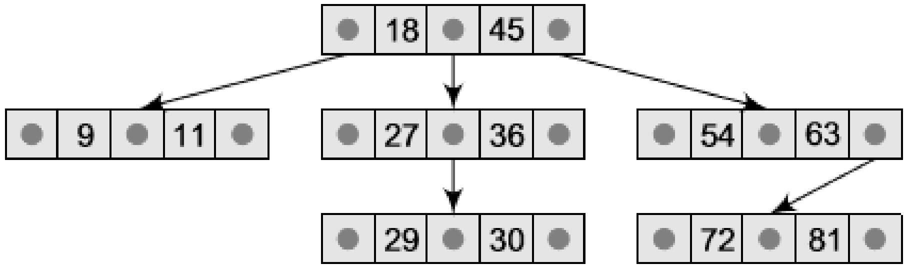


Figure 11.3 M-way search tree of order 3

Introduction

- 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 Pi are less than Ki, where $0 \leqslant i \leqslant n-1$.
  - 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 Pi are greater than Ki–1, where $0 \leqslant i \leqslant n-1$. In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

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.

B Trees

- An internal node in the B tree can have n number of children, where $0 \leqslant n \leqslant m$.
- 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.
- 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.

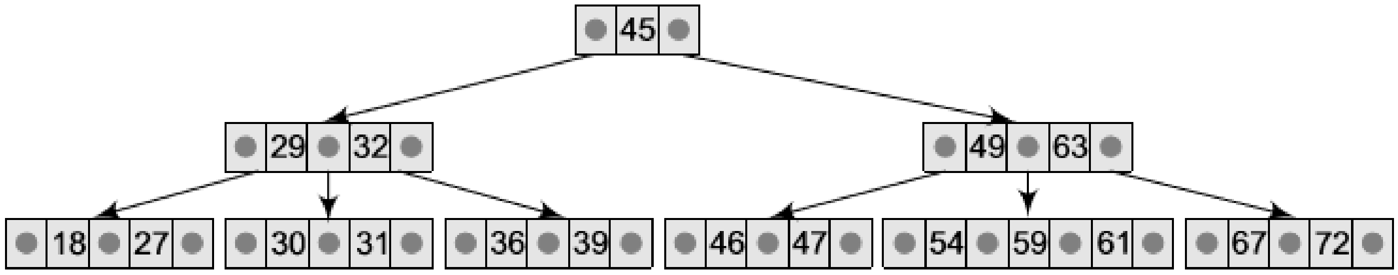


Figure 11.4 B tree of order 4

B Trees

- 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 ≤ 59 ≤ 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(logt n) time to execute.

B Trees

- 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.

B Trees

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

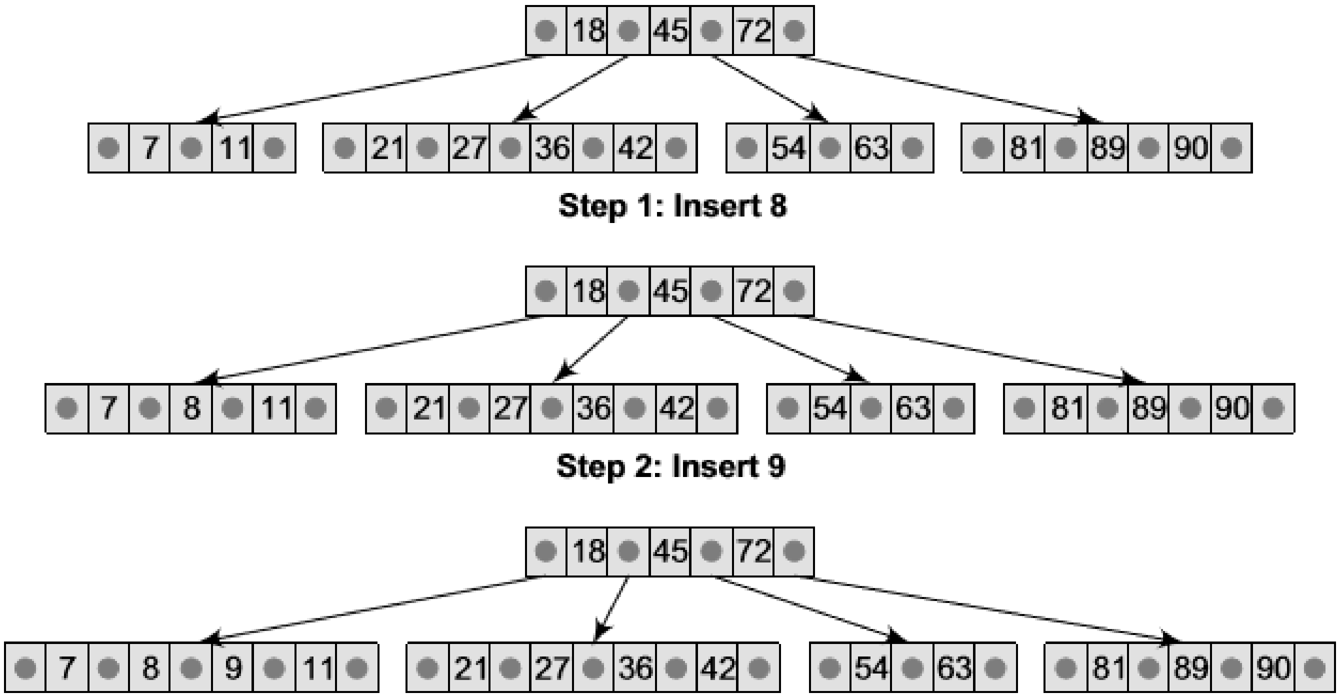


Figure 11.5(a)

B Trees

- 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.

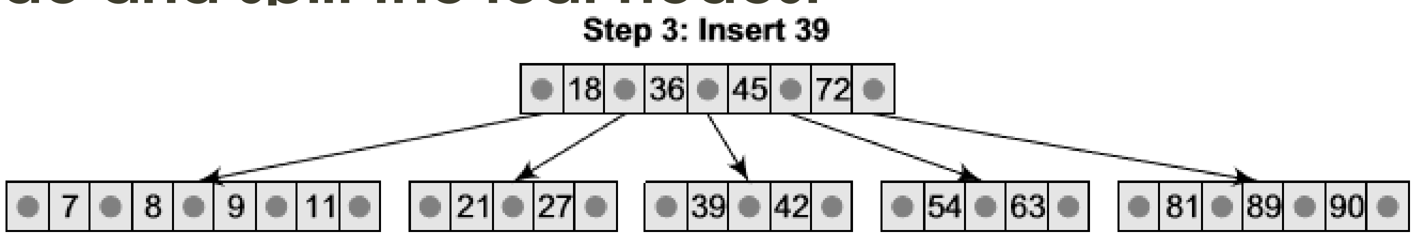


Figure 11.5(b)

B Trees

- 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.

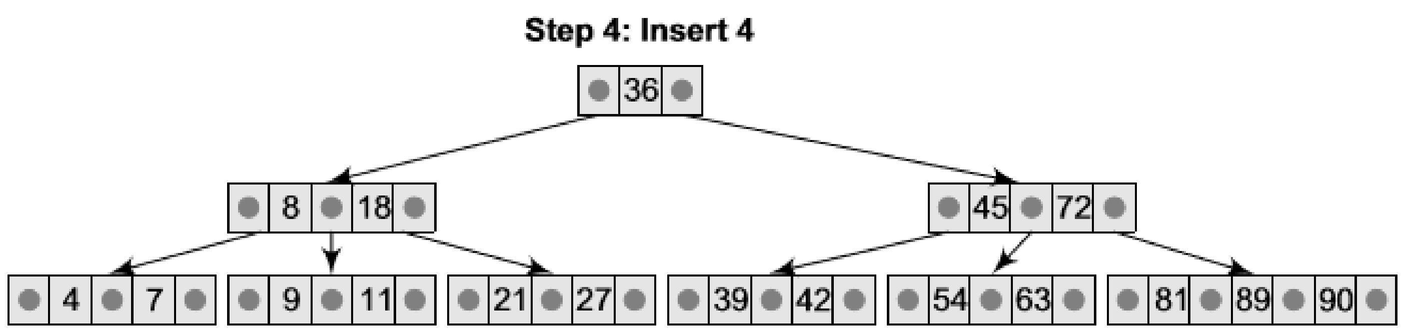


Figure 11.5(c) B tree

B Trees

- 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.

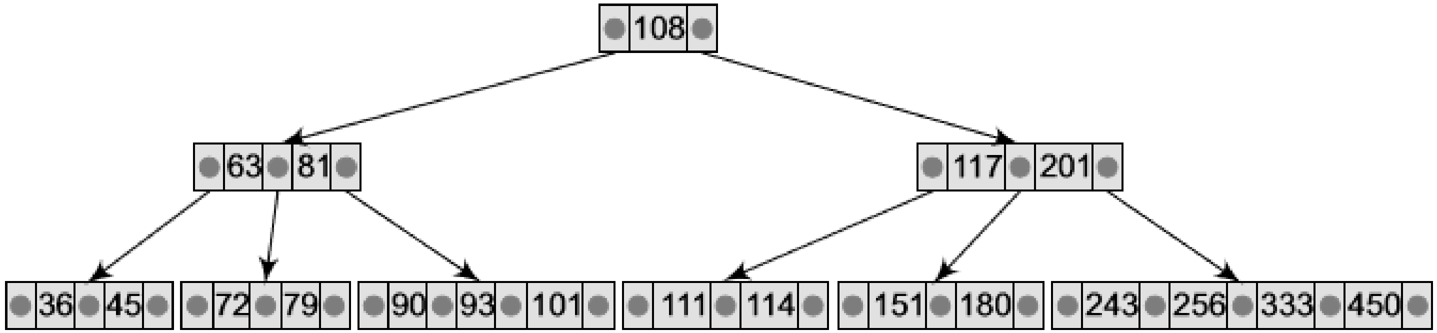
B Trees

- 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.

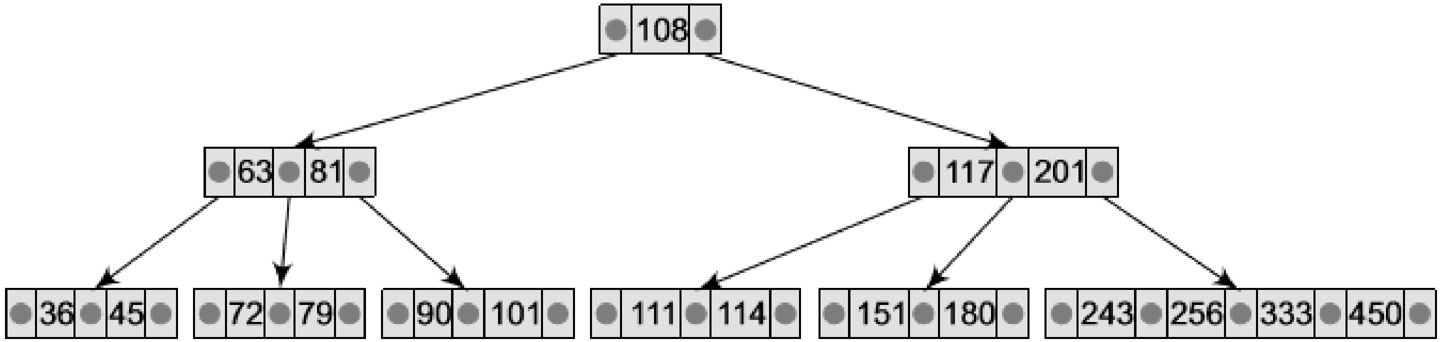
- To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key.
- This predecessor or successor will always be in the leaf node.
- So the processing will be done as if a value from the leaf node has been deleted.

B Trees

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)).



Step 1: Delete 93



Step 2: Delete 201

B Trees

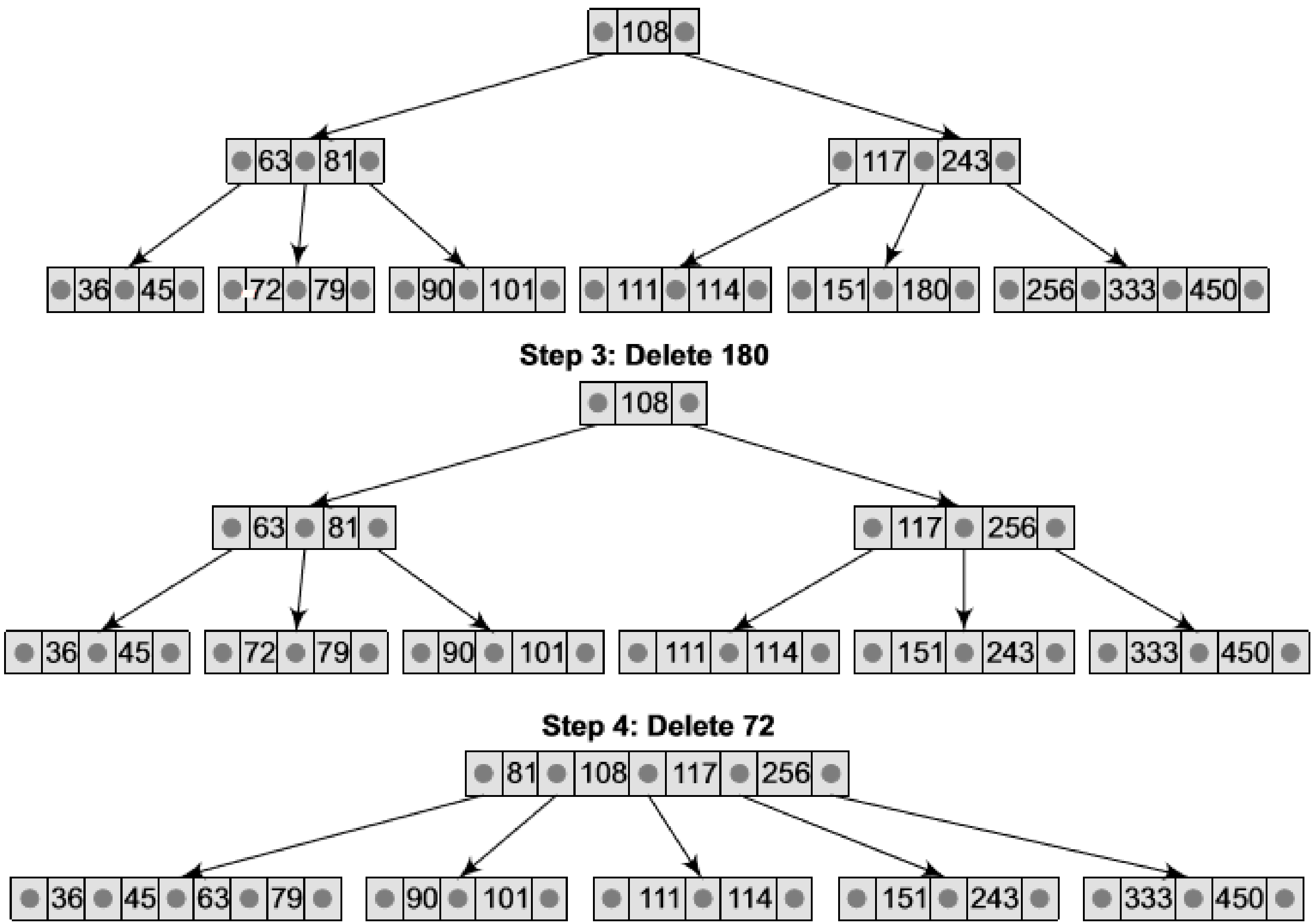


Figure 11.6 B tree

B Trees

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.

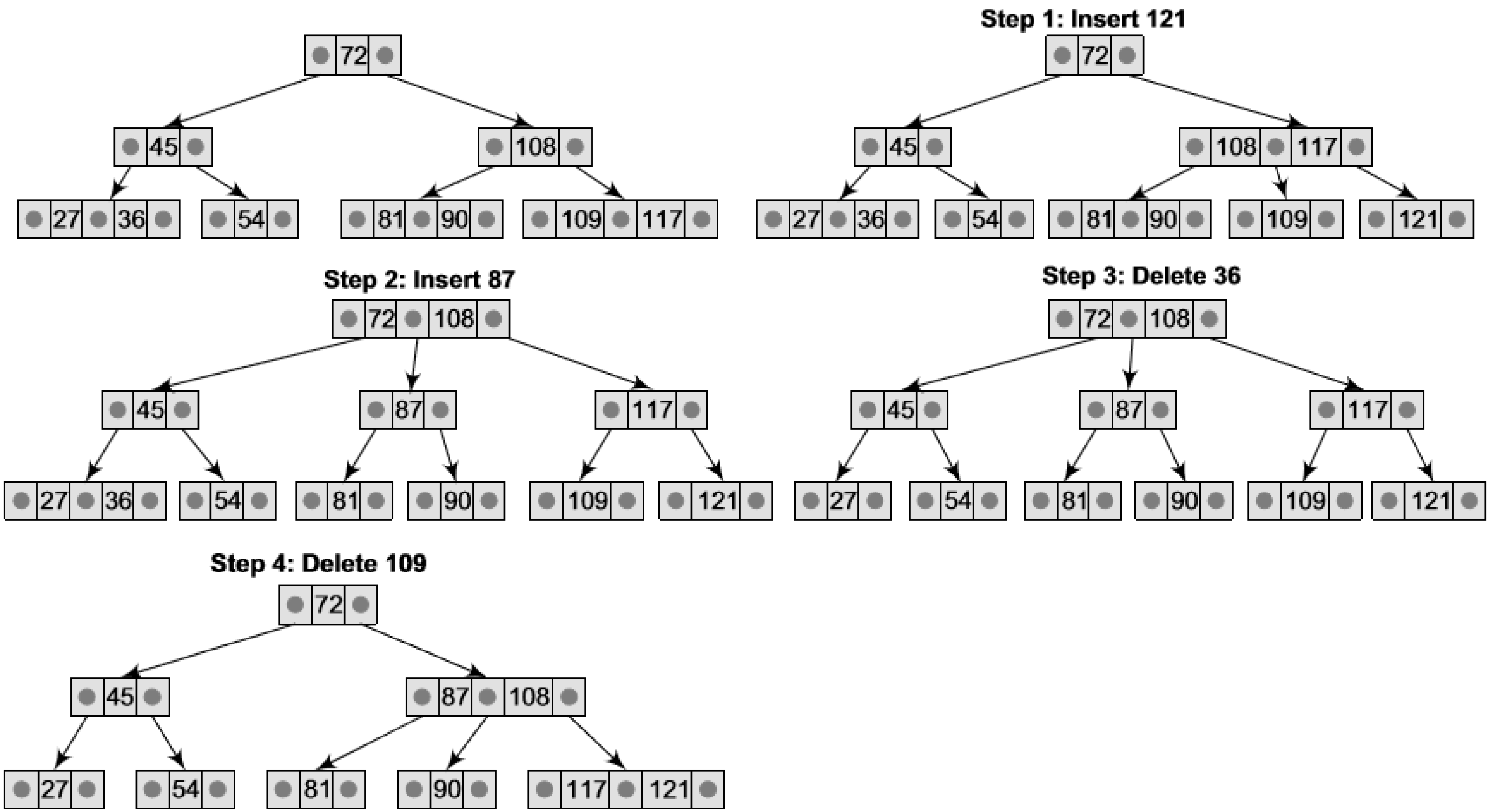
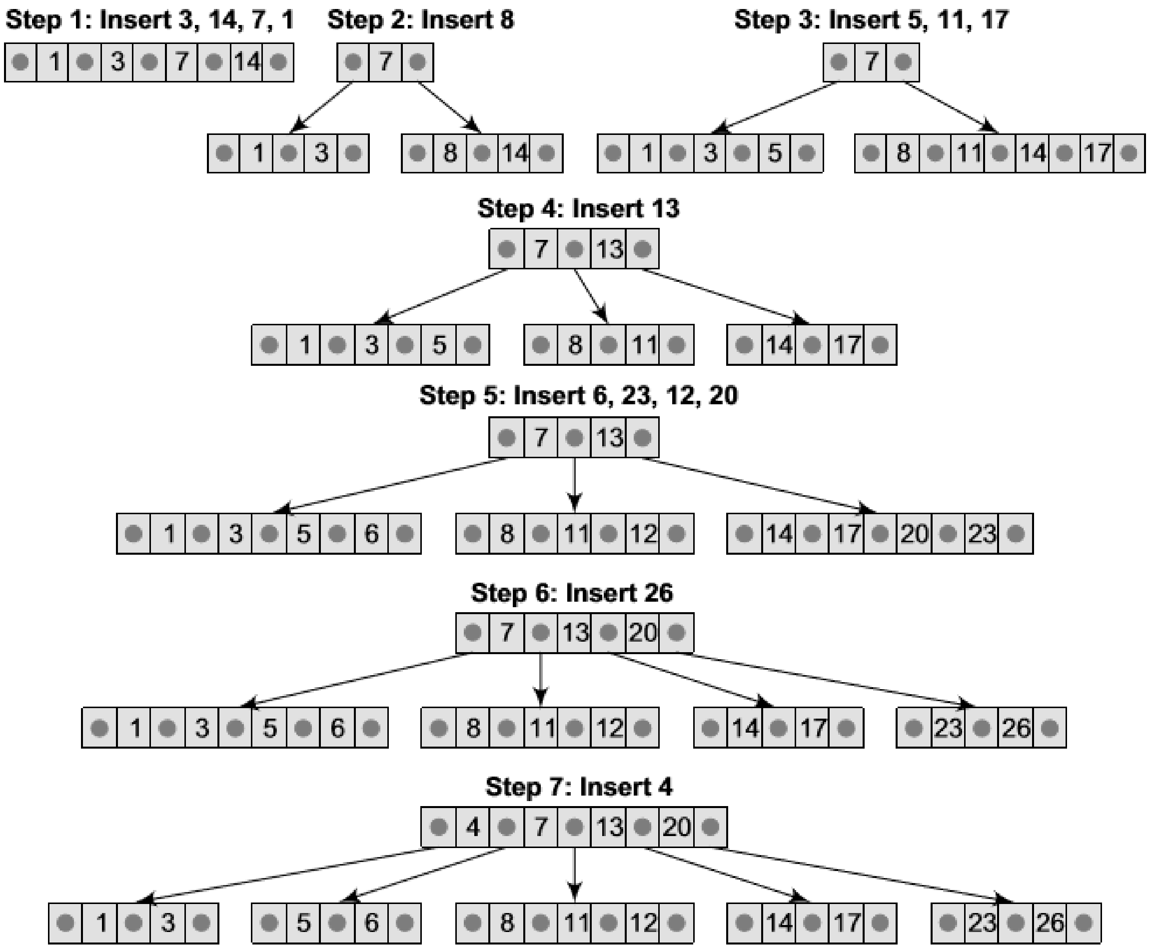


Figure 11.7 B tree

B Trees

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)

B Trees

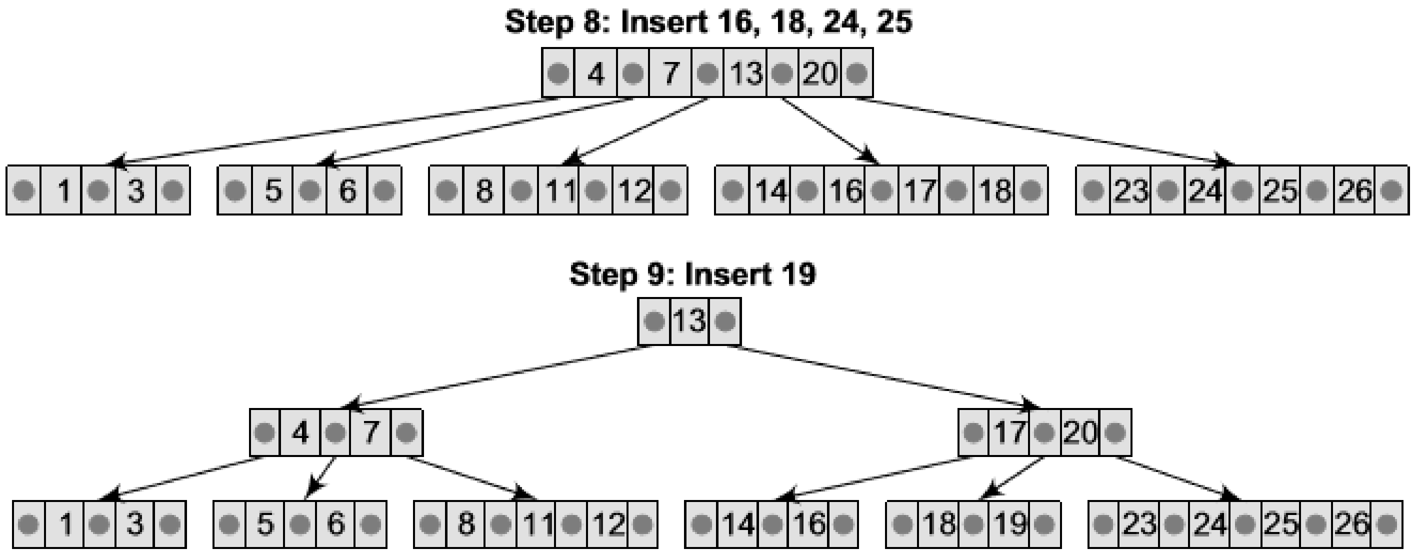


Figure 11.8 B tree

B+ Trees

- A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key.
- While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes.
- The leaf nodes of a B+ tree are often linked to one another in a linked list.
- This has an added advantage of making the queries simpler and more efficient.
- Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory.
- With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.
- B+ trees store data only in the leaf nodes.
- All other nodes (internal nodes) are called index nodes or i-nodes and store index values.
- This allows us to traverse the tree from the root down to the leaf node that stores the desired data item.

B+ Trees

- Figure 11.9 shows a B+ tree of order 3.
- Many database systems are implemented using B+ tree structure because of its simplicity.
- Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.
- A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index.
- The advantages of B+ trees can be given as follows:
  - 1. Records can be fetched in equal number of disk accesses
  - 2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
  - 3. Height of the tree is less and balanced
  - 4. Supports both random and sequential access to records
  - 5. Keys are used for indexing

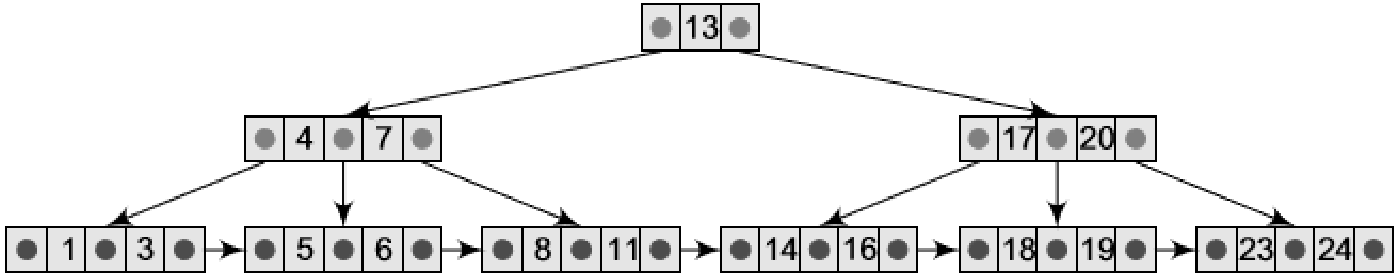


Figure 11.9 B+ tree of order 3

B+ Trees

Comparison Between B Trees and B+ Trees

Table 11.1 shows the comparison between B trees and B+ trees.

Table 11.1 Comparison between B trees and to B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

B+ Trees

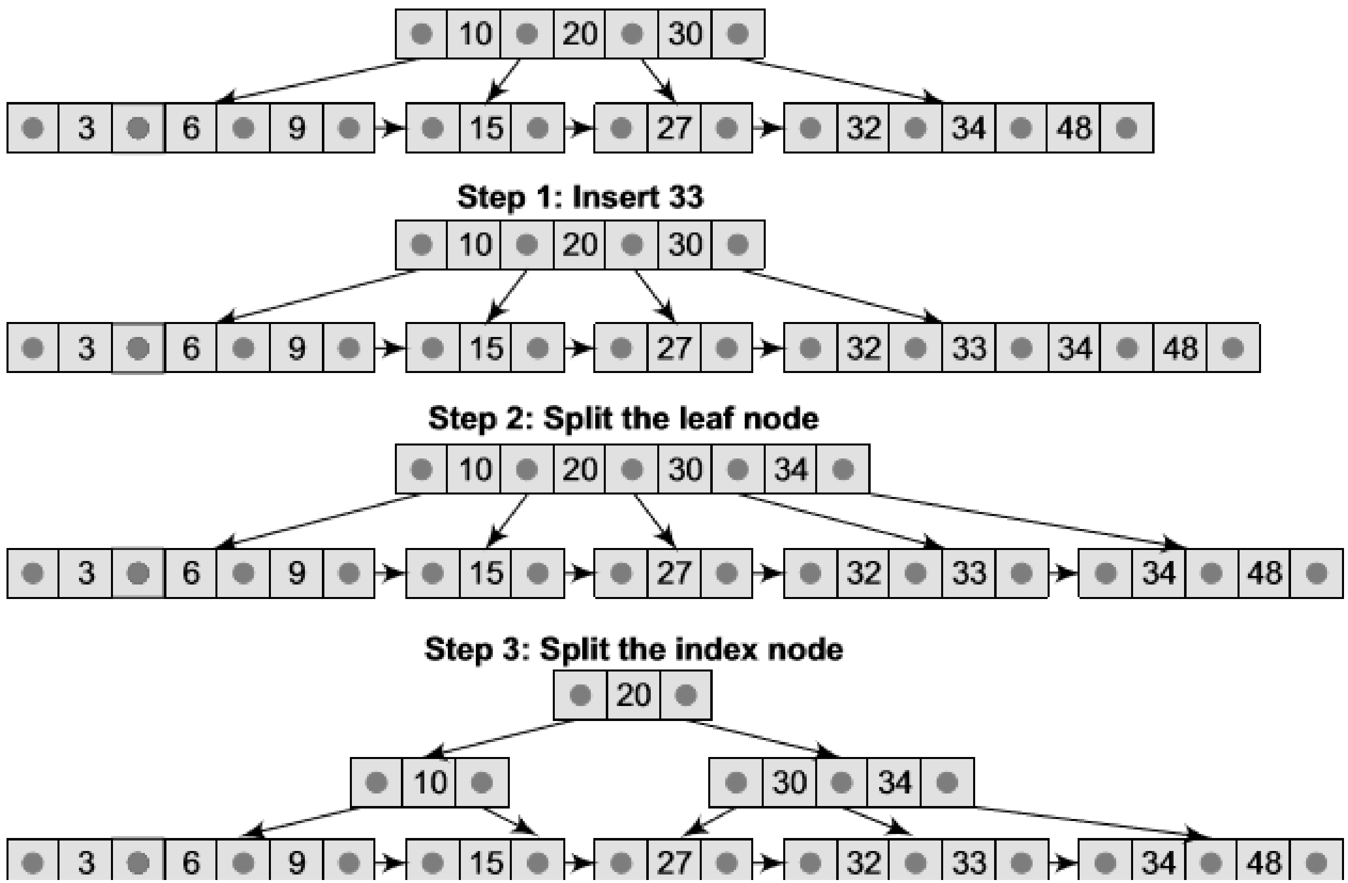
- Inserting a New element in a B+ Tree
- A new element is simply added in the leaf node if there is space for it.
- But if the data node in the tree where insertion has to be done is full, then that node is split into two nodes.
- This calls for adding a new index value in the parent index node so that future queries can arbitrate between the two new nodes.
- However, adding the new index value in the parent node may cause it, in turn, to split.
- In fact, all the nodes on the path from a leaf to the root may split when a new value is added to a leaf node.
- If the root node splits, a new leaf node is created and the tree grows by one level.
- The steps to insert a new node in a B+ Tree are summarized in Fig. 11.10.

B+ Trees

Step 3: If the index node overflows, split that node and move the middle element to next index page.

Figure 11.10 Algorithm for inserting a new node in a B+ tree

Example 11.5 Consider the B+ tree of order 4 given and insert 33 in it.



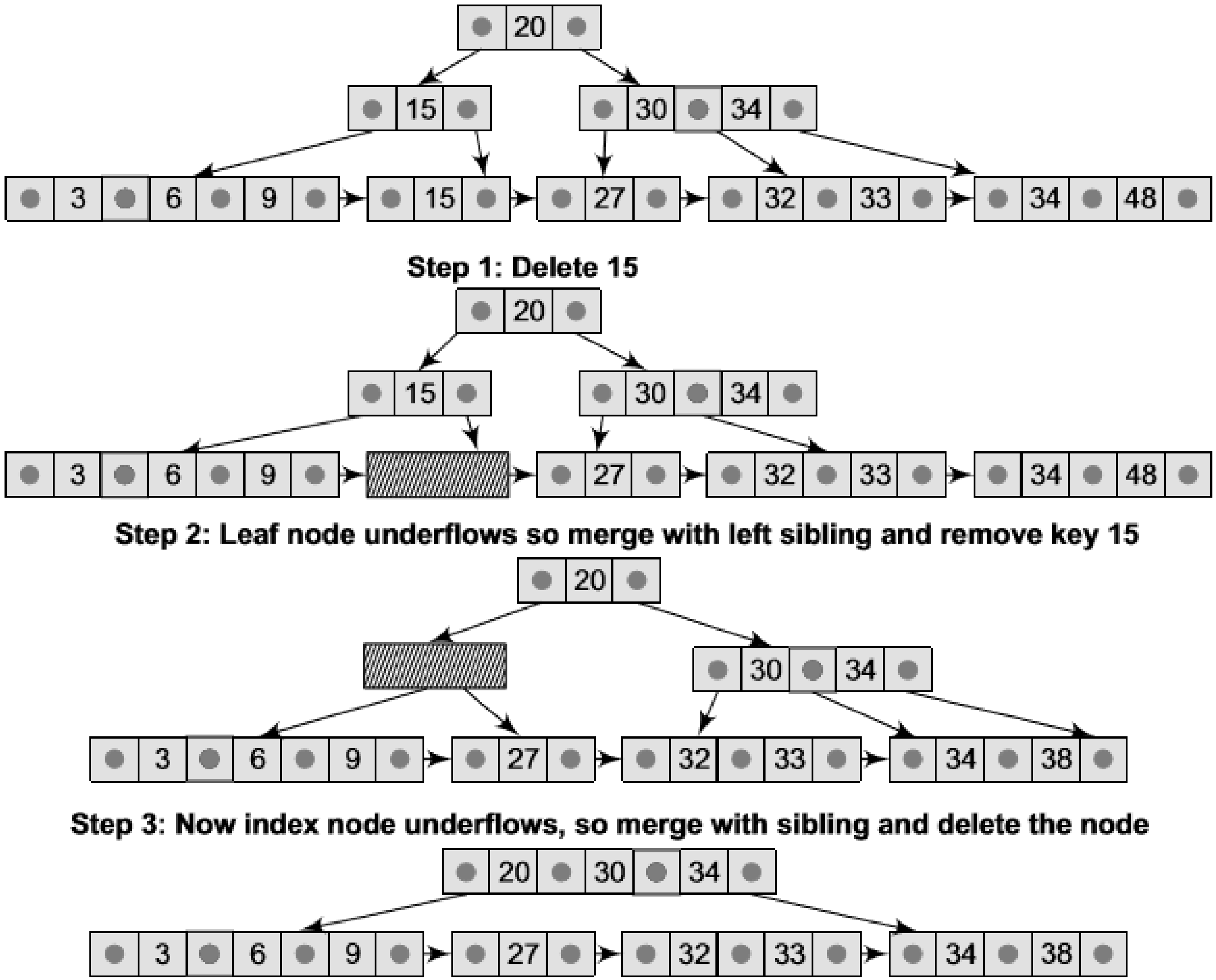
B+ Trees

- Deleting an element from a B+ Tree
- As in B trees, deletion is always done from a leaf node.
- If deleting a data element leaves that node empty, then the neighboring nodes are examined and merged with the underfull node.
- This process calls for the deletion of an index value from the parent index node which, in turn, may cause it to become empty.
- Similar to the insertion process, deletion may cause a merge-delete wave to run from a leaf node all the way up to the root.
- This leads to shrinking of the tree by one level. The steps to delete a node from a B+ tree are summarized in Fig. 11.12.

Step 1:	Delete the key and data from the leaves.
Step 2:	If the leaf node underflows, merge that node with the sibling and delete the key in between them.
Step 3:	If the index node underflows, merge that node with the sibling and move down the key in between them.

Figure 11.12 Algorithm for deleting a node from a B+ Tree

B+ Trees


20
15
30 34
3 6 9
15
27
32 33
34 48
Step 1: Delete 15
20
15
30 34
3 6 9
27
32 33
34 48
Step 2: Leaf node underflows so merge with left sibling and remove key 15
20
30 34
3 6 9
27
32 33
34 38
Step 3: Now index node underflows, so merge with sibling and delete the node
20 30 34
3 6 9
27
32 33
34 38

2-3 Trees

- In the last chapter, we have seen that for binary search trees the average-case time for operations like search/insert/delete is O(log N) and the worst-case time is O(N) where N is the number of nodes in the tree.
- However, a balanced tree that has height O(log N) always guarantees O(log N) time for all three methods.
- Typical examples of height balanced trees include AVL trees, red-black trees, B trees, and 2-3 trees.
- We have already discussed these data structures in the earlier chapter and section; now we will discuss 2-3 trees. In a 2-3 tree, each interior node has either two or three children.
  - Nodes with two children are called 2-nodes. The 2-nodes have one data value and two children
  - Nodes with three children are called 3-nodes. The 3-nodes have two data values and three children (left child, middle child, and a right child) This means that a 2-3 tree is not a binary tree. In this tree, all the leaf nodes are

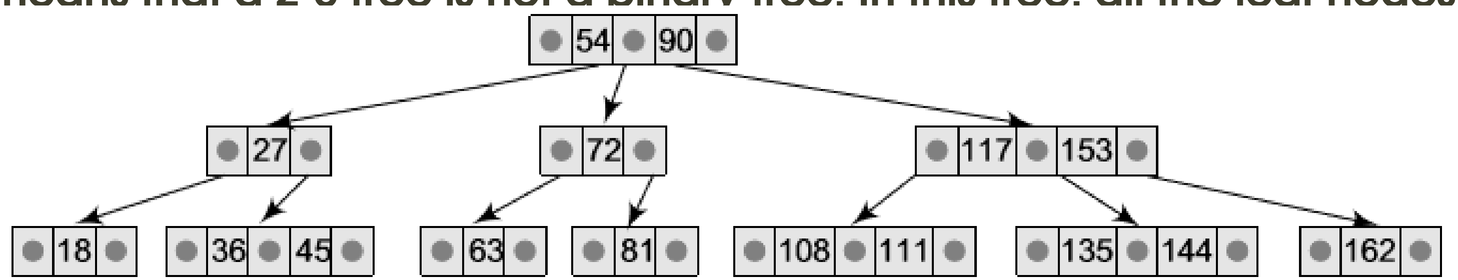


Figure 11.14 2-3 Tree

2-3 Trees

- Searching for an element in a 2-3 Tree
- The search operation is used to determine whether a data value x is present in a 2-3 tree T.
- The process of searching a value in a 2-3 tree is very similar to searching a value in a binary search tree.
- The search for a data value x starts at the root. If k_1 and k_2 are the two values stored in the root node, then
  - if $x < k1$, move to the left child.
  - if $x \geqslant k1$ and the node has only two children, move to the right child.
  - if $x \geqslant k1$ and the node has three children, then move to the middle child if $x < k_2$ else to the right child if $x \geqslant k_2$. At the end of the process, the node with data value x is reached if and only if x is at this leaf.

2-3 Trees

Example 11.7 Consider the 2-3 tree in Fig. 11.14 and search 63 in the tree.

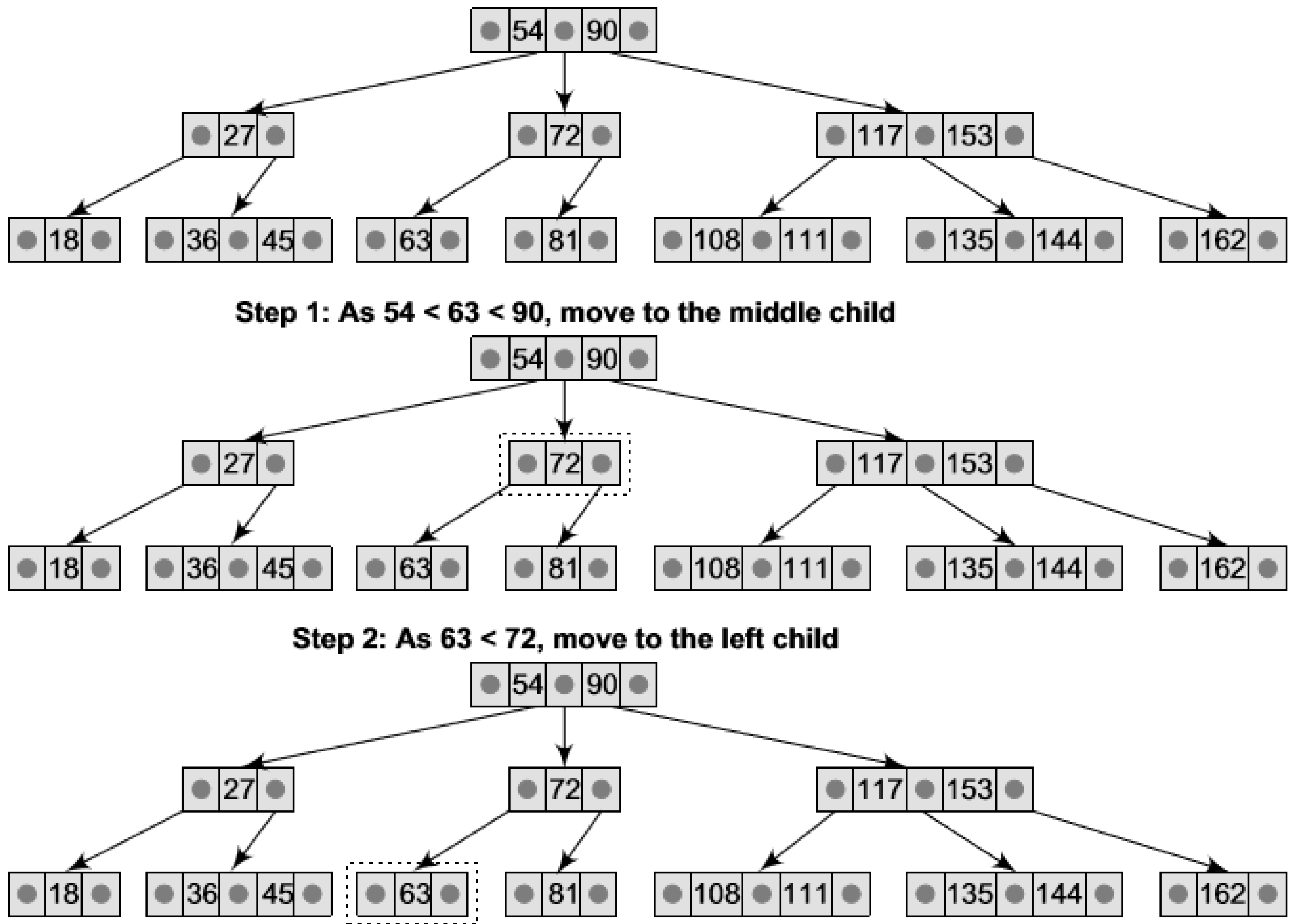


Figure 11.15 Searching for element 63 in the 2-3 tree of Fig. 11.14

2-3 Trees

- Inserting a New element in a 2-3 Tree
- To insert a new value in the 2-3 tree, an appropriate position of the value is located in one of the leaf nodes.
- If after insertion of the new value, the properties of the 2-3 tree do not get violated then insertion is over.
- Otherwise, if any property is violated then the violating node must be split (Fig. 11.16).
- Splitting a node A node is split when it has three data values and four children. Here, P is the parent and L, M, R denote the left, middle, and right children.

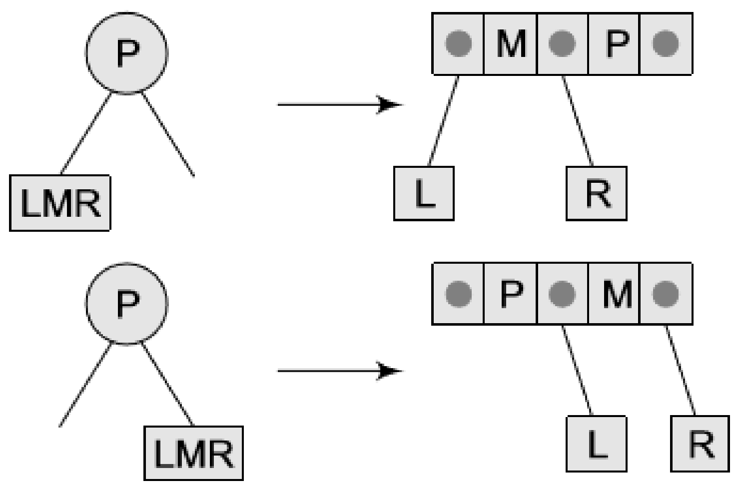


Figure 11.16(a)

2-3 Trees

Example 11.8 Consider the 2-3 tree given below and insert the following data values into it: 39, 37, 42, 47.

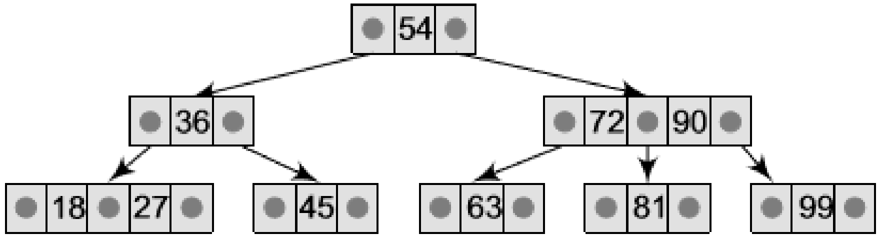


Figure 11.16(b)

Step 1: Insert 39 in the leaf node The tree after insertion can be given as

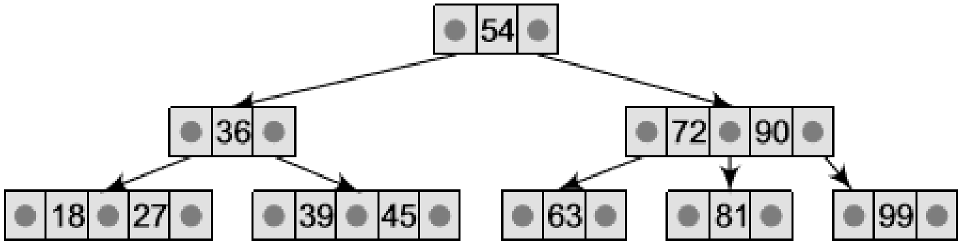


Figure 11.16(c)

2-3 Trees

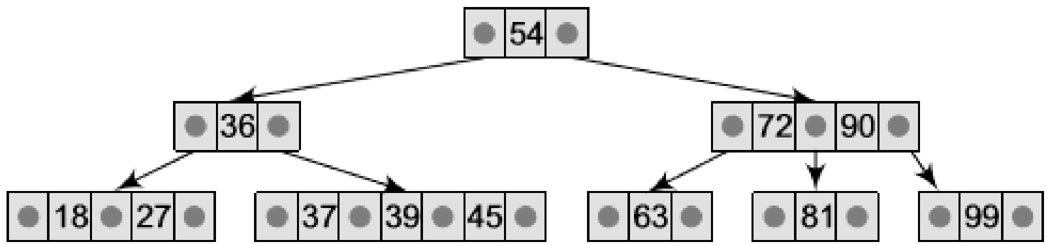


Figure 11.16(d)

After splitting the leaf node, the tree can be given as below.

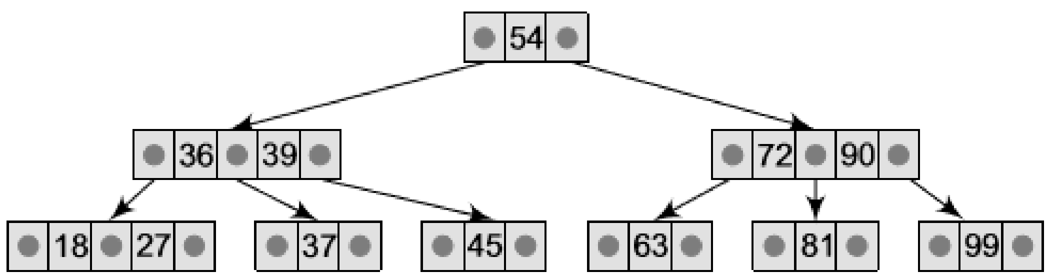
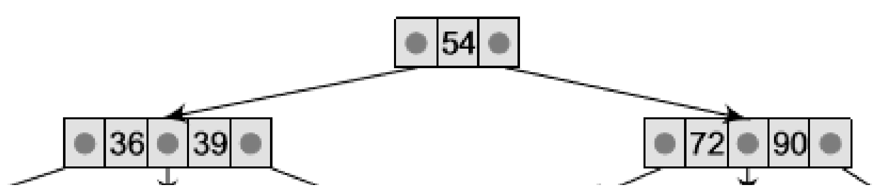


Figure 11.16(e)

Step 3: Insert 42 in the leaf node The tree after insertion can be given as follows.



2-3 Trees

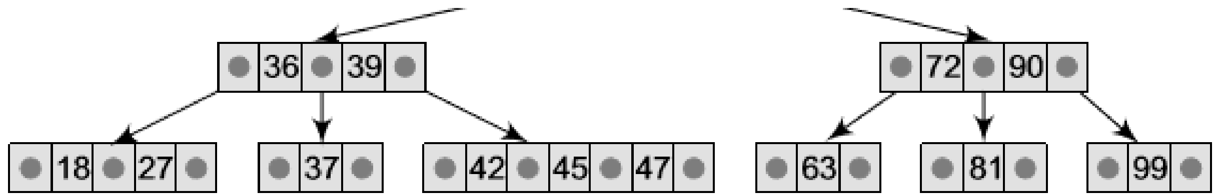


Figure 11.16(g)

The leaf node has three data values. Therefore, the node is violating the properties of the tree and must be split.

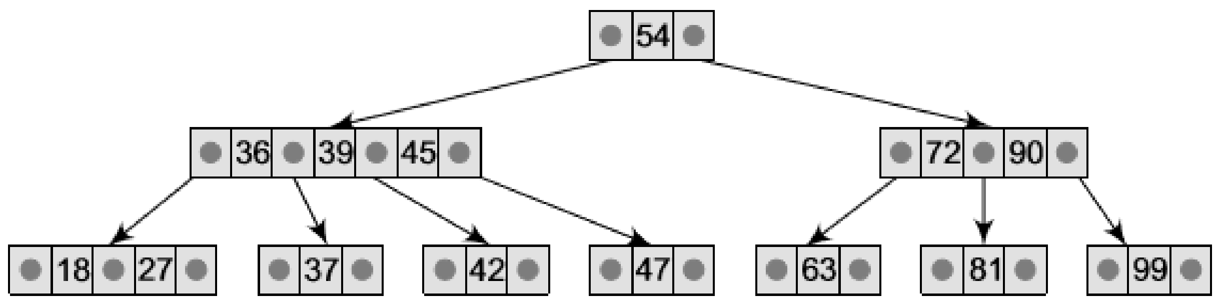
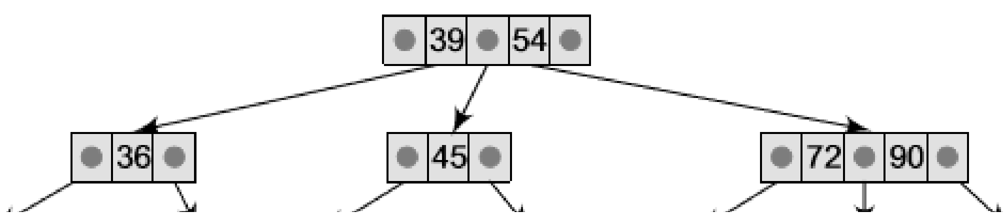


Figure 11.16(h)

The parent node has three data values. Therefore, the node is violating the properties of the tree and must be split.



2-3 Trees

- Deleting an element from a 2-3 Tree
- In the deletion process, a specified data value is deleted from the 2-3 tree.
- If deleting a value from a node violates the property of a tree, that is, if a node is left with less than one data value then two nodes must be merged together to preserve the general properties of a 2-3 tree.
- In insertion, the new value had to be added in any of the leaf nodes but in deletion it is not necessary that the value has to be deleted from a leaf node.
- The value can be deleted from any of the nodes.
- To delete a value x, it is replaced by its in-order successor and then removed. If a node becomes empty after deleting a value, it is then merged with another node to restore the property of the tree.

Example 11.9 Consider the 2-3 tree given below and delete the following values from it: 69, 72, 99, 81.

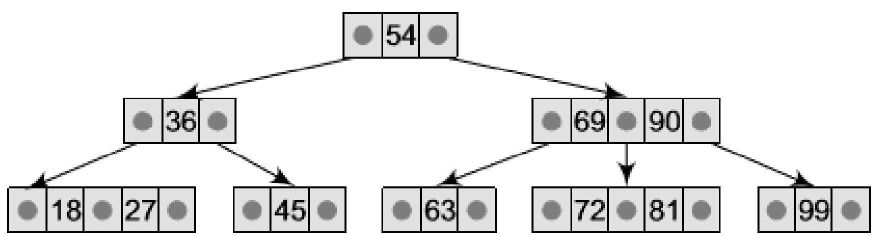


Figure 11.17(a)

To delete 69, swap it with its in-order successor, that is, 72. 69 now comes in the leaf node. Remove the value 69 from the leaf node.

2-3 Trees

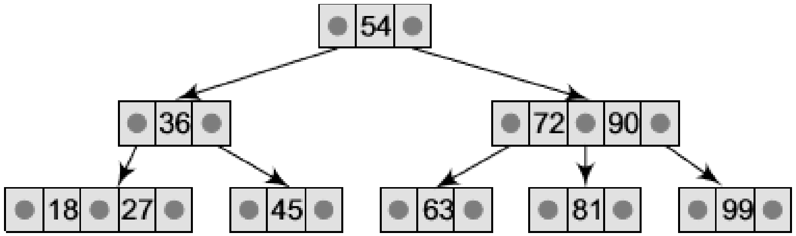


Figure 11.17(b)

72 is an internal node. To delete this value swap 72 with its in-order successor 81 so that 72 now becomes a leaf node. Remove the value 72 from the leaf node.

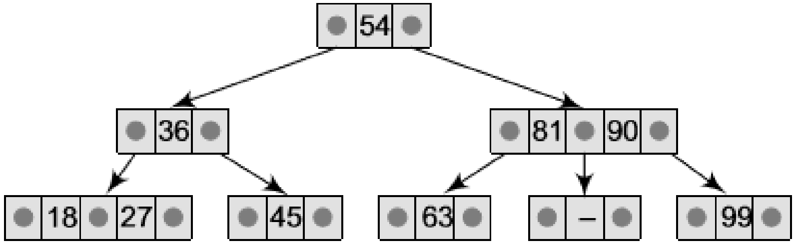


Figure 11.17(c)

Now there is a leaf node that has less than 1 data value thereby violating the property of a 2-3 tree. So the node must be merged. To merge the node, pull down the lowest data value in the parent's node and merge it with its left sibling.

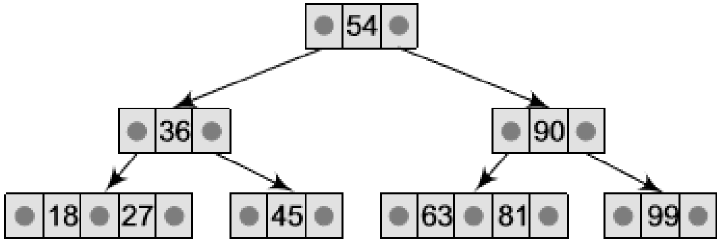


Figure 11.17(d)

99 is present in a leaf node, so the data value can be easily removed.

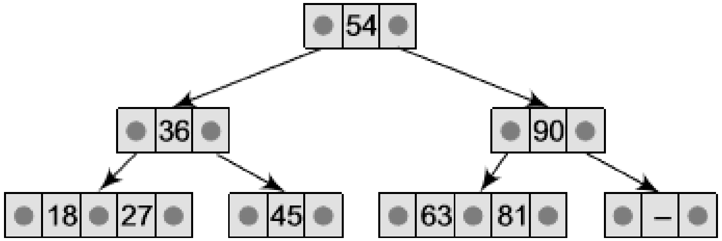


Figure 11.17(e)

Now there is a leaf node that has less than 1 data value, thereby violating the property of a 2-3 tree. So the node must be merged. To merge the node, pull down the lowest data value in the parent's node and merge it with its left sibling.

2-3 Trees

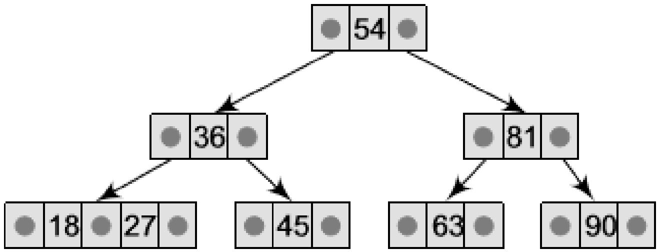


Figure 11.17(f)

81 is an internal node. To delete this value swap 81 with its in-order successor 90 so that 81 now becomes a leaf node. Remove the value 81 from the leaf node.

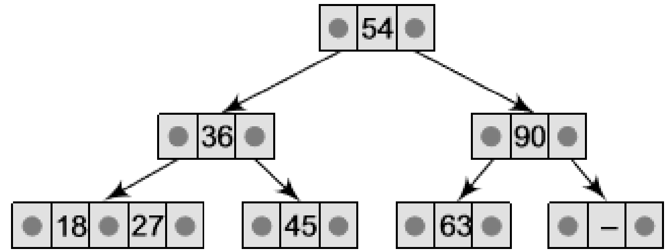


Figure 11.17(g)

Now there is a leaf node that has less than 1 data value, thereby violating the property of a 2-3 tree. So the node must be merged. To merge the node, pull down the lowest data value in the parent's node and merge it with its left sibling.

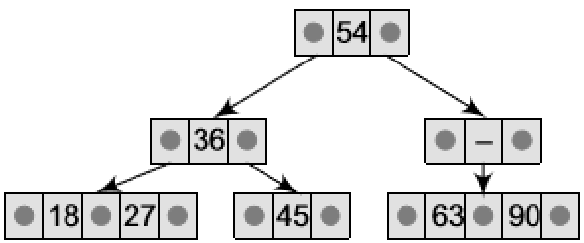


Figure 11.17(h)

An internal node cannot be empty, so now pull down the lowest data value from the parent's node and merge the empty node with its left sibling.

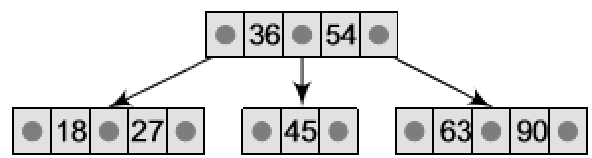


Figure 11.17(i) Deleting values from the given 2-3 tree