Trees

October 12, 2023

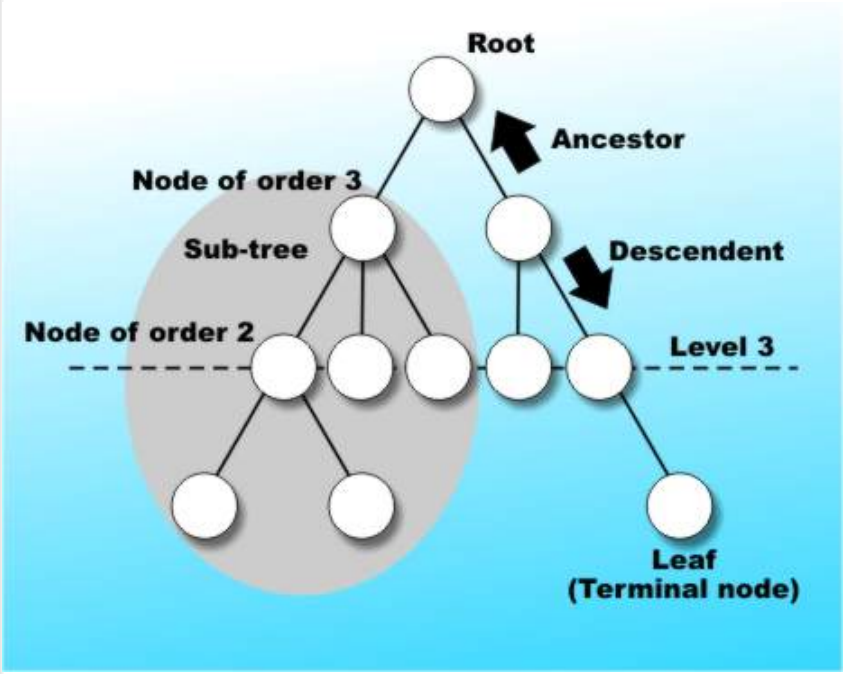
Tree

A tree is a collection of nodes such that:

- The collection can be empty, otherwise there is a specially designated node called the root.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, ..., T_n$, where each of these sets is also a tree.
- We call $T_1, ..., T_n$ the subtrees of the root.
- Each subtree is connected by a directed edge from root.

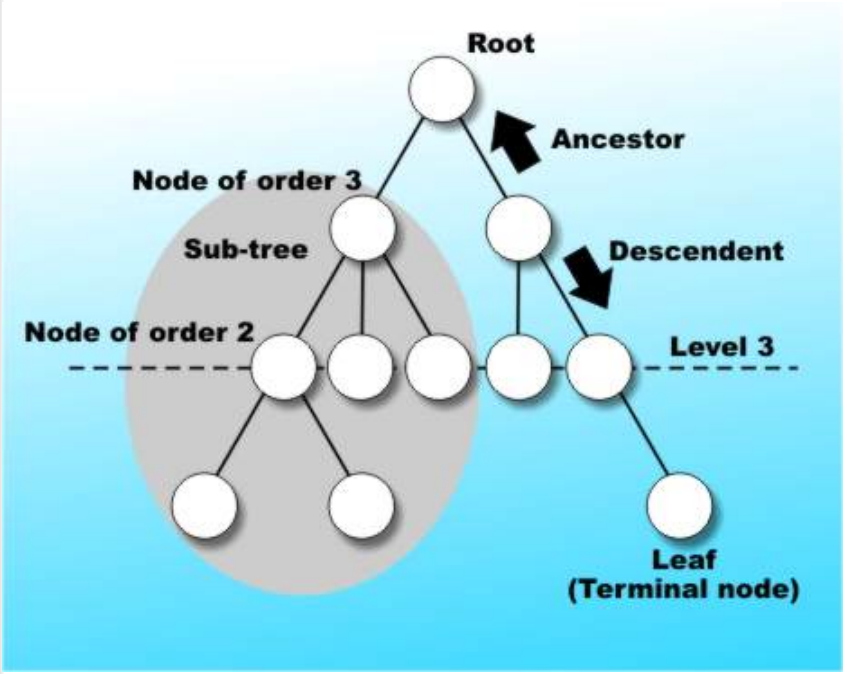
Terminology

- The degree of a node is the number of subtrees of the node
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the parent of the roots of the subtrees.
- The roots of these subtrees are the children of the node.
- Children of the same parent are siblings.
- The ancestors of a node are all the nodes along the path from the root to the node.

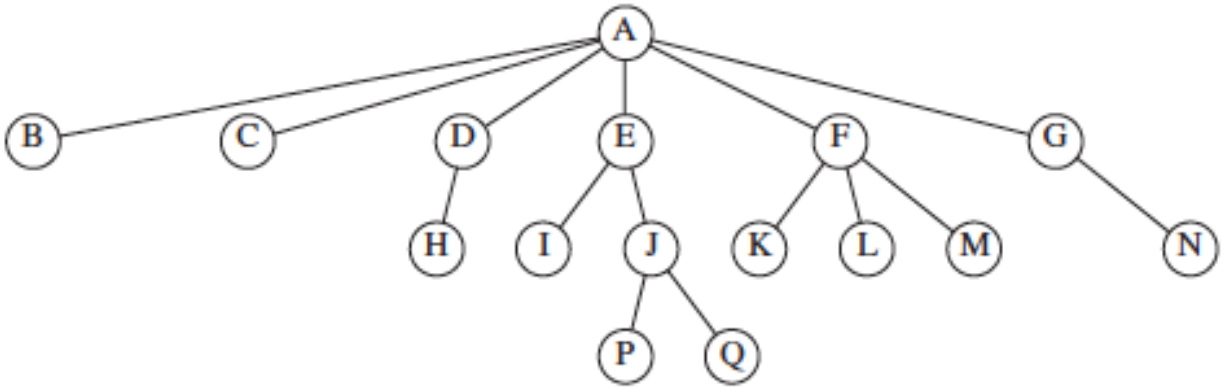


Terminology

- A path from $node_1$ to $node_k$ is defined as a sequence of nodes $node_1$, $node_2$ ... $node_k$ such that $node_i$ is the parent of $node_{i+1}$ for $1 \leq i < k$
- For any node $node_i$ the depth of $node_i$ is the length of the unique path from the root to $node_i$
- The height of $node_i$ is the length of the longest path from $node_i$ to a leaf.
- The height of a tree is the height of root.



A Tree

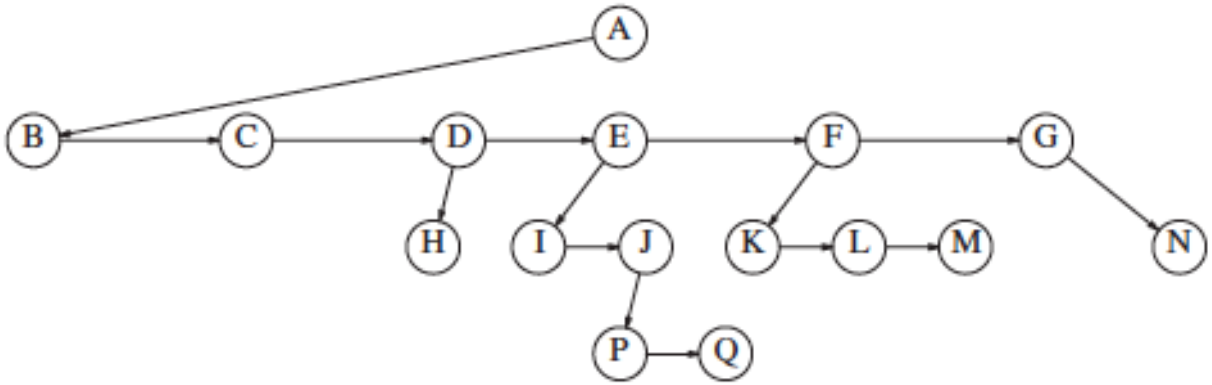


An Implementation of Tree Node

```
public class TreeNode {
    Object element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

- Each node is part of a Linked List of siblings
- Additionally, each node stores a reference to its children

An Implementation of Tree



- First child/next sibling representation of the the same tree
- Arrows that point downward are **firstChild** links.
- Horizontal arrows are **nextSibling** links

Tree Traversals

- Suppose we want to print all the nodes in a tree
- What order should we visit the nodes?
  - **Preorder** - read the parent before its children
  - **Postorder** - read the parent after its children

Preorder/Postorder

Algorithm Preorder(tree):

- Visit the root.
- Traverse the left subtree, i.e., call Preorder(left->subtree)
- Traverse the right subtree, i.e., call Preorder(right->subtree)

Algorithm Postorder(tree)

- Traverse the left subtree, i.e., call Postorder(left->subtree)
- Traverse the right subtree, i.e., call Postorder(right->subtree)
- Visit the root

Preorder/Postorder

InOrder(root) visits nodes in the following order:

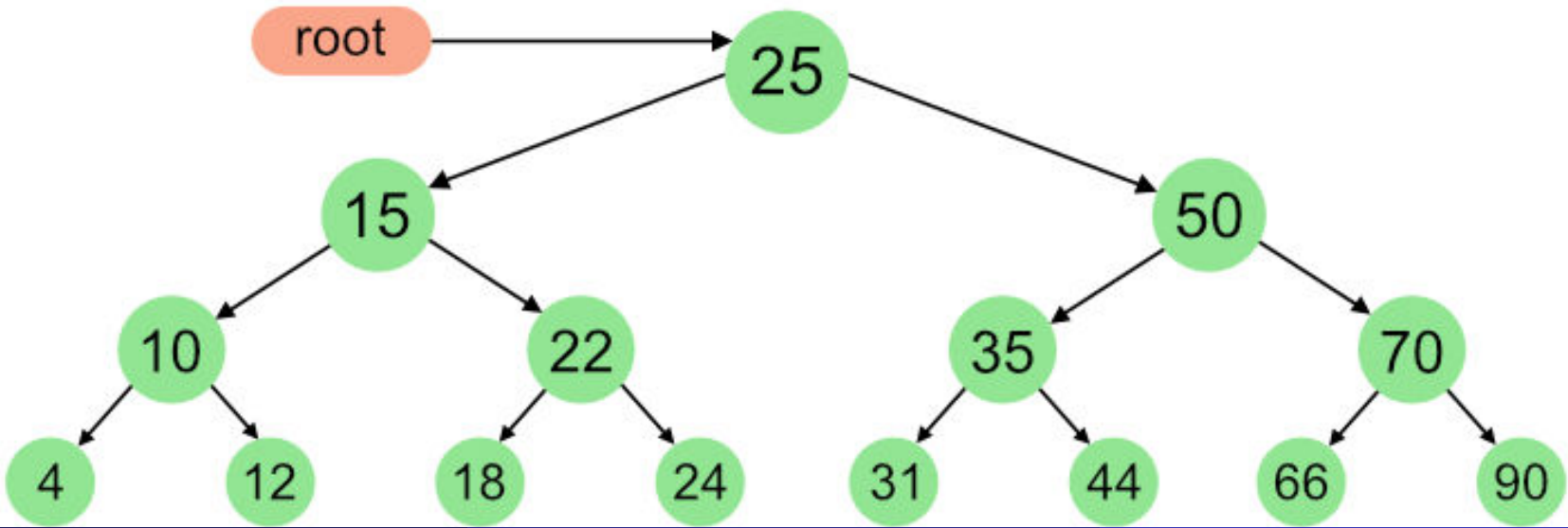
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

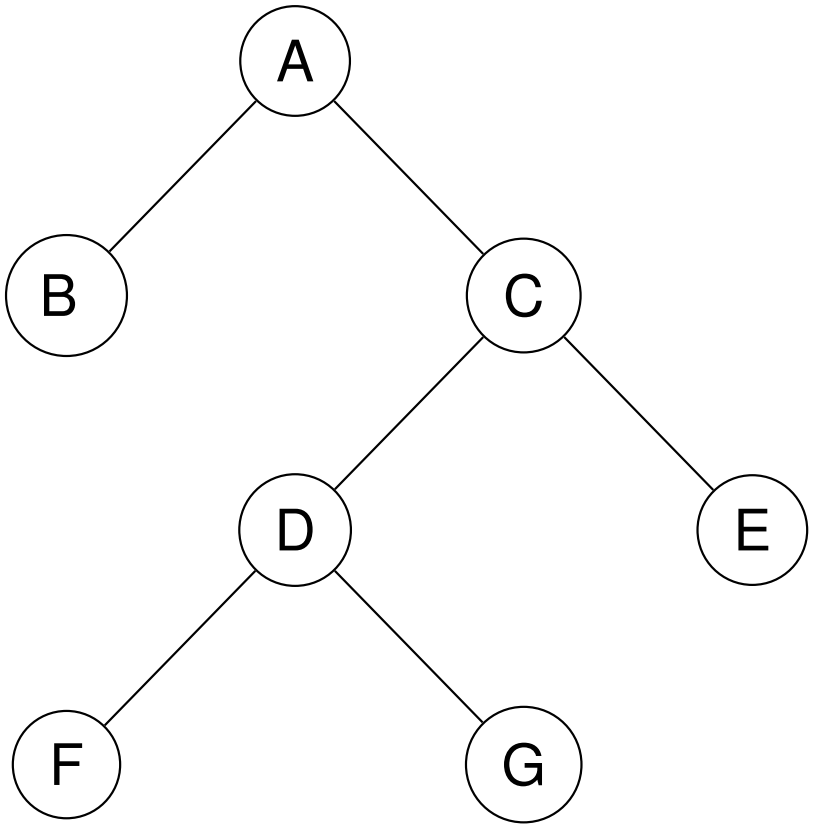


Binary Tree

- A binary tree is a tree in which each node can have at most two children.
- The two children of a node are called the left child (T_L) and the right child (T_R), if they exist.
- Simplifies implementation and logic
- Provides new **inorder** traversal:
  - Read left child, then parent, then right child
  - Essentially scans whole tree from left to right

```
public class BinaryNode {
    Object element;
    BinaryNode left;
    BinaryNode right;
}
```

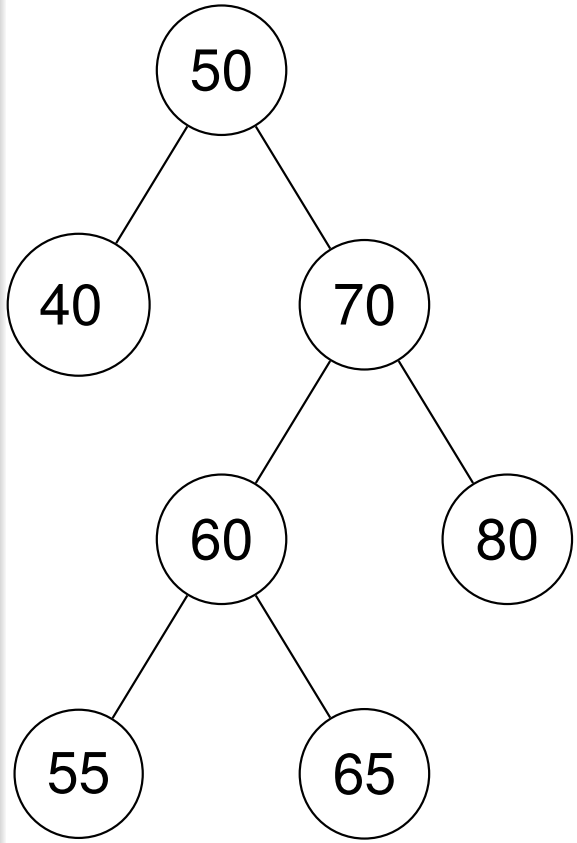
Binary Tree Example



- A binary tree is **full** if each node has 2 or 0 children
- A binary tree is **perfect** if it is full and each leaf is at the same depth
- That depth is O(log N)
- What about the tree on the left?

The Search Tree ADT—Binary Search Trees

- An important application of binary trees is their use in searching.
- ADT that allows insertion, removal, and searching by key (a key is a value that can be compared)
- A special kind of binary tree in which:
  - Each node contains a distinct data value,
  - The key values in the tree can be compared using "greater than" and "less than" operators
  - **The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree**
  - BST property holds for all subtrees of a BST



Examples

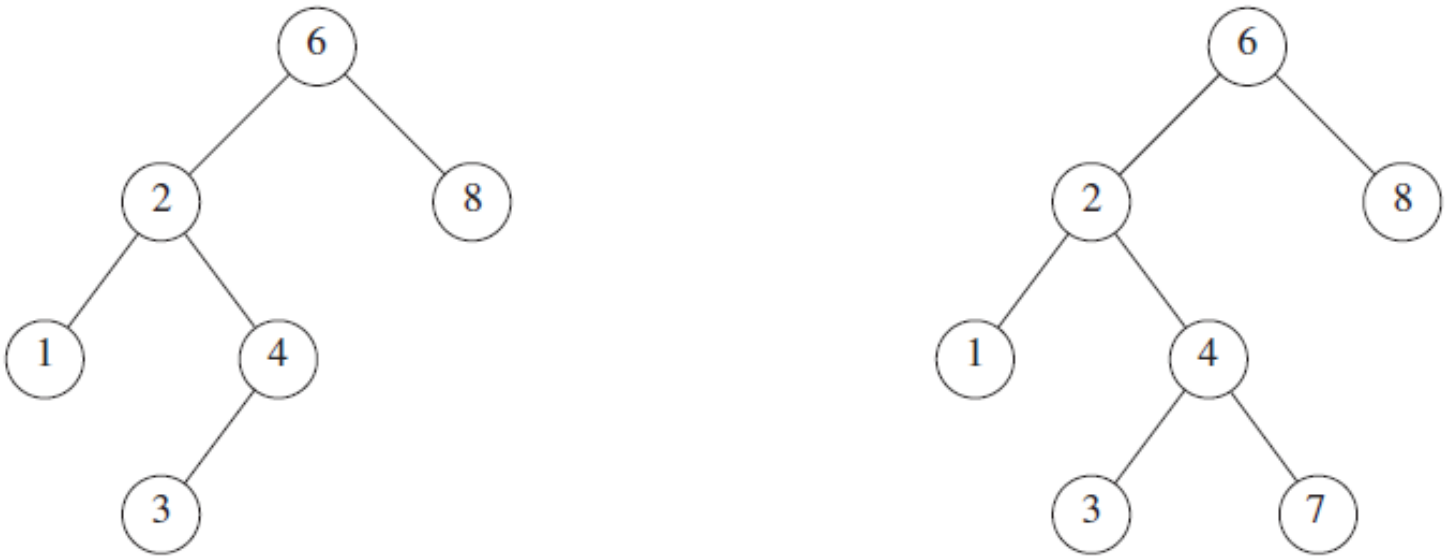


Figure 4.15 Two binary trees (only the left tree is a search tree)

Searching a BST

```
findMin(t):
    if (t.left == null) return t.key
    else return findMin(t.left)


contains(x,t):
    if (t == null) return false
    if (x == t.key) return true
    if (x > t.key), then return contains(x, t.right)
    if (x < t.key), then return contains(x, t.left)
```

Insertion into a Binary Search Tree

Pseudocode

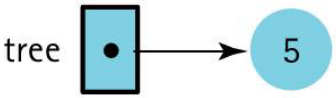
```
insert(x, Node t):
    if (t == null) return new Node(x)
    if (x > t.key), then t.right = insert(x, t.right)
    if (x < t.key), then t.left = insert(x, t.left)
    return t
```

Why? : insert is written as a method that returns a reference to the root of the new tree.

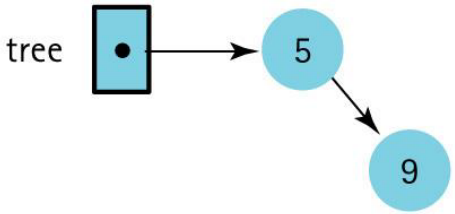
Binary Search Tree Insertion Example

(a) tree

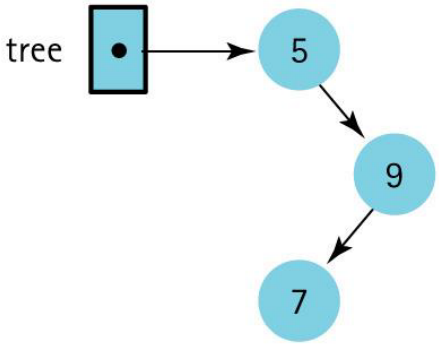
(b) Insert 5



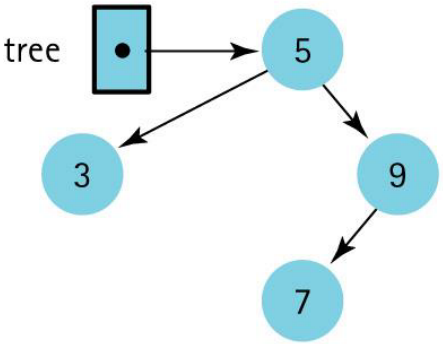
(c) Insert 9



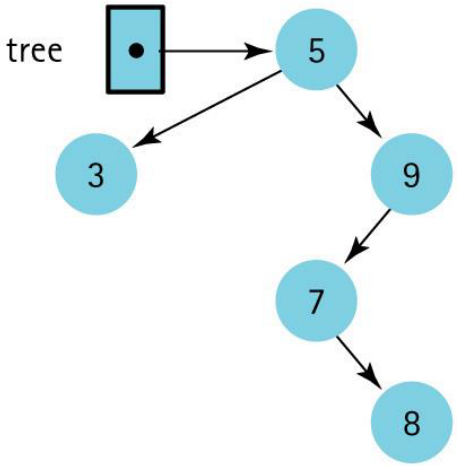
(d) Insert 7



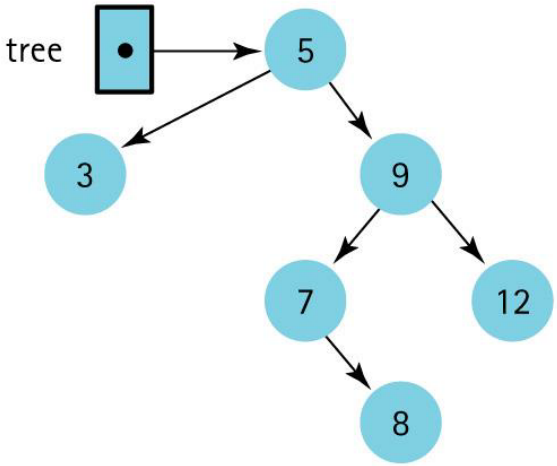
(e) Insert 3



(f) Insert 8



(g) Insert 12



Remove Operation

- Removing a leaf is easy, delete immediately
- removing a node with one child is also easy (after its parent adjusts a link to bypass the node)
- the complicated case deals with a node with two children.
  - First, find node to be removed, t
  - Replace with the smallest node from the right subtree (relabel this node with the key of its immediate successor in sorted order)
    - a = findMin(t.right);
    - t.key = a.key;
  - Then delete original smallest node in right subtree
    - remove(a.key, t.right)

Removing a Node in BST

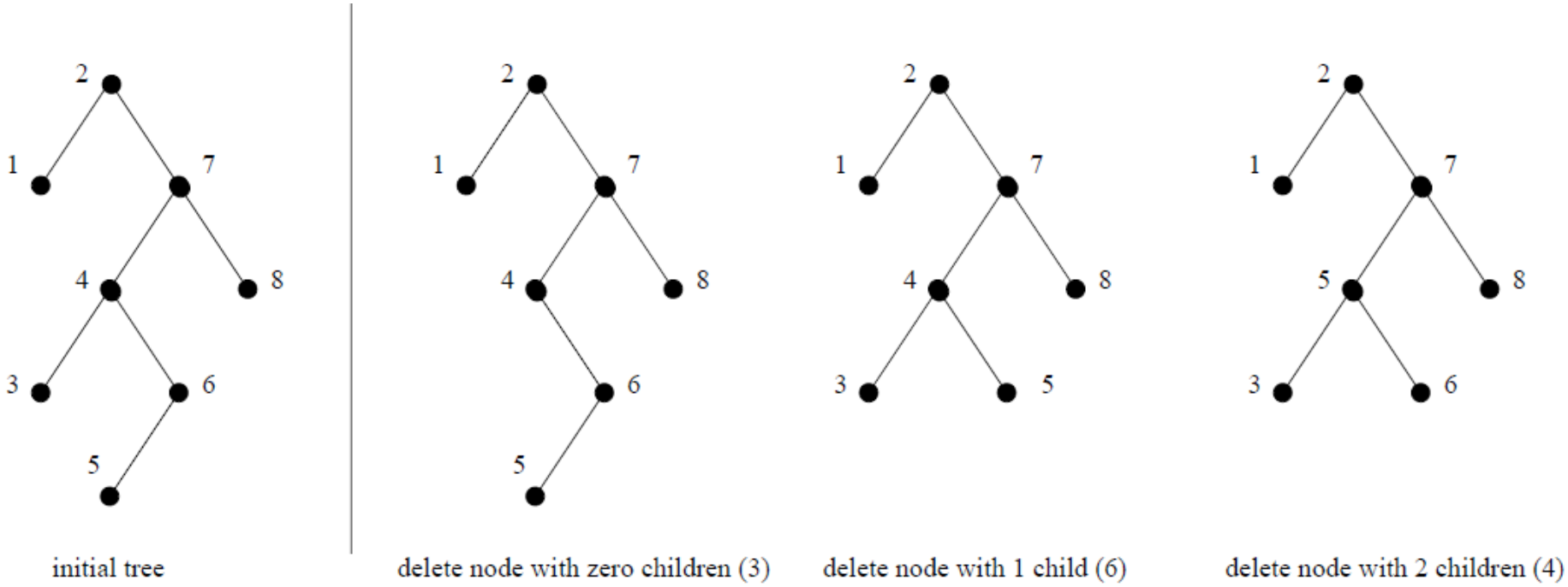
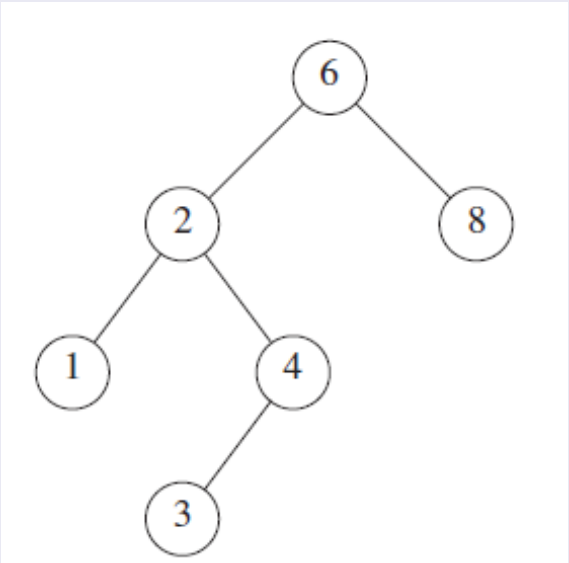


Figure 3.4: Deleting tree nodes with 0, 1, and 2 children

Image from: Algorithm Design Manual

Inorder Tree Traversal (Walk)

```
inorder-tree-walk(t):
    if (t != null)
        inorder-tree-walk(t.left)
        print t.key
        inorder-tree-walk(t.right)
```



1, 2, 3, 4, 6, 8

Average case analysis

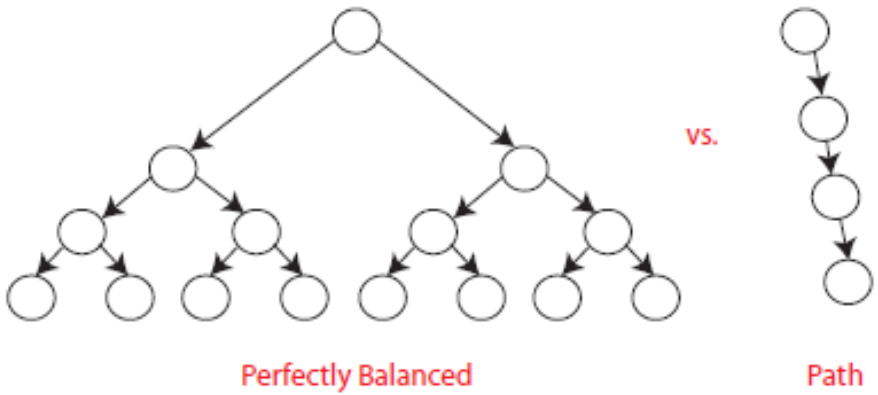
- All operations run in O(d) time, but what is d?
  - Worst case d = N
  - Best case d = log(N+1)-1
  - Average case?
- Consider the internal path length: the sum of the depths of all nodes in a tree
- Let D(N) be the internal path length for some tree **T** with **N** nodes
  - Suppose **i** nodes are in the left subtree of T.
  - Then D(N) = $D(i) + D(N - i - 1) + N - 1$

Average case analysis (cont'd)

- Assume all insertion sequences are equally likely
- Subtree sizes only depend on the 1st key inserted
- all subtree sizes equally likely
  - Average of D(i) and D(N-i-1) is $\frac{1}{N}\sum_{j=0}^{N-1} D(j)$
- Average case D(N) then becomes: D(N) = $\frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$
- This is a recurrence, which can be solved to show that D(N) = O(N logN) page 272-273 in Weiss
- Then the average depth over all N nodes is O(logN)

Question-Importance of Being Balanced

- BSTs support insert, delete, min, max, next-larger, next-smaller, etc. in O(h) time, where h = height of tree and is between *logn* and *n*:
- How do we implement Search Trees that explicitly avoid worst case O(N) operations?
- What is the cost of avoiding worst case?
- Motivation: want height of tree to be close to log N
- balanced BST maintains h = O(log n) all operations run in O(log n) time.



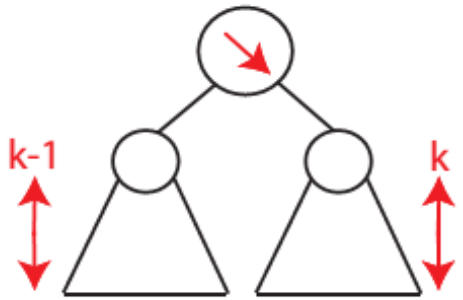
Examples of Balanced Trees

- AVL Trees Adel'son-Velskii and Landis 1962
- B-Trees/2-3-4Trees Bayer and McCreight 1972
- Red-black Trees
- Splay-Trees
- ...

Take-Home Lesson: Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.

AVL Trees

- Adelson-Velskii and Landis tree is a binary search tree with a balance condition
- **AVL Tree Property**
  - For each node, all keys in its left subtree are less than the node's and all keys in its right subtree are greater.
  - the height of the left and right subtrees differ by at most 1
- An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (Recall: height is max number of edges from root to a leaf)
- The height of an empty tree is defined to be -1 ($|h_l\text{-}h_r| \leq 1$)



Examples

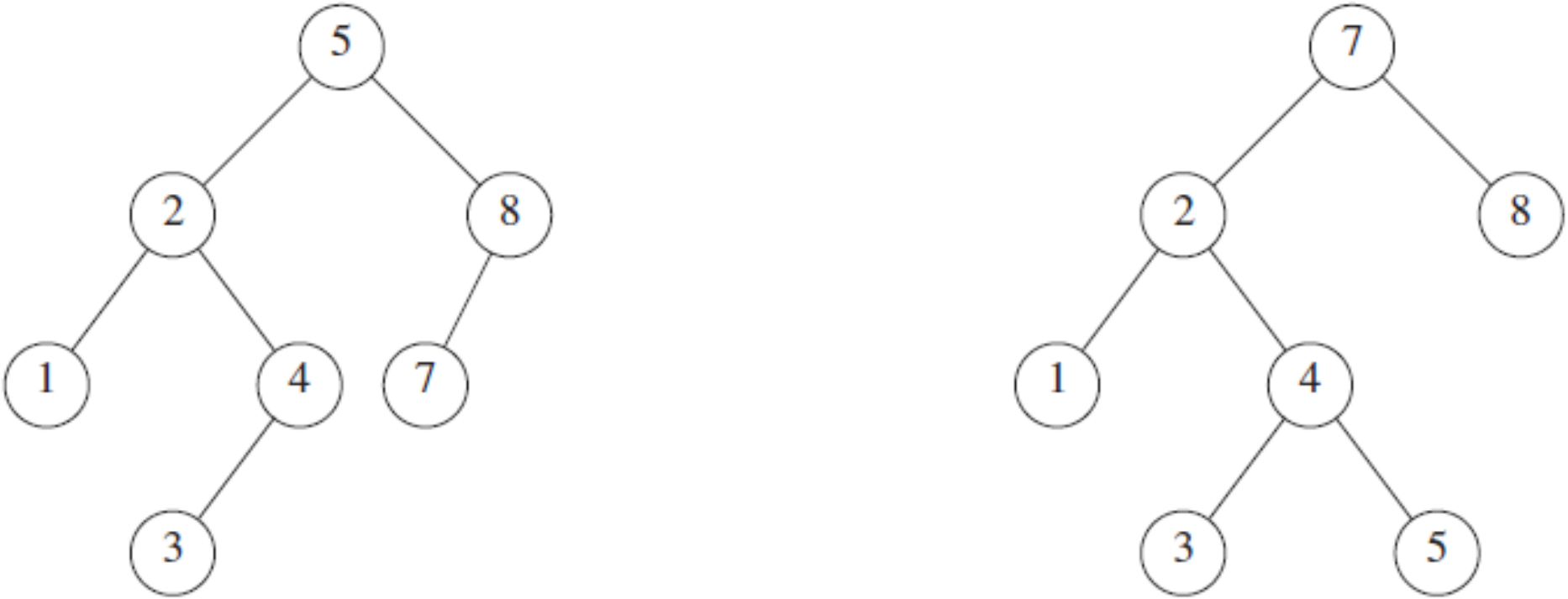


Figure 4.29 Two binary search trees. Only the left tree is AVL

Rotations-How to keep a BST balanced

- Thus, all the tree operations can be performed in O(logN) time, except possibly insertion.
- Inserting a node could violate the AVL tree property.
  1. insert as in simple BST
  2. work your way up tree, restoring AVL property (and updating heights as you go).
- To balance the tree after an insertion violates the AVL property
  - rearrange the tree; make a new node the root.
  - This rearrangement is called a rotation.
  - There are 2 types of rotations: single and double

Cases of Rebalancing

Let α be the node where an imbalance occurs. Four cases to consider:

1. An insertion into the left subtree of the left child of α (left-left or left child's left subtree)
2. An insertion into the right subtree of the left child of α (left-right or left child's right subtree)
3. An insertion into the left subtree of the right child of α (right-left or right child's left subtree)
4. An insertion into the right subtree of the right child of α (right-right or right child's right subtree)

Cases 1 and 4 can be solved by single rotation whereas Cases 2 and 3 require double rotation

Single (Right) Rotation for Case 1: left-left case

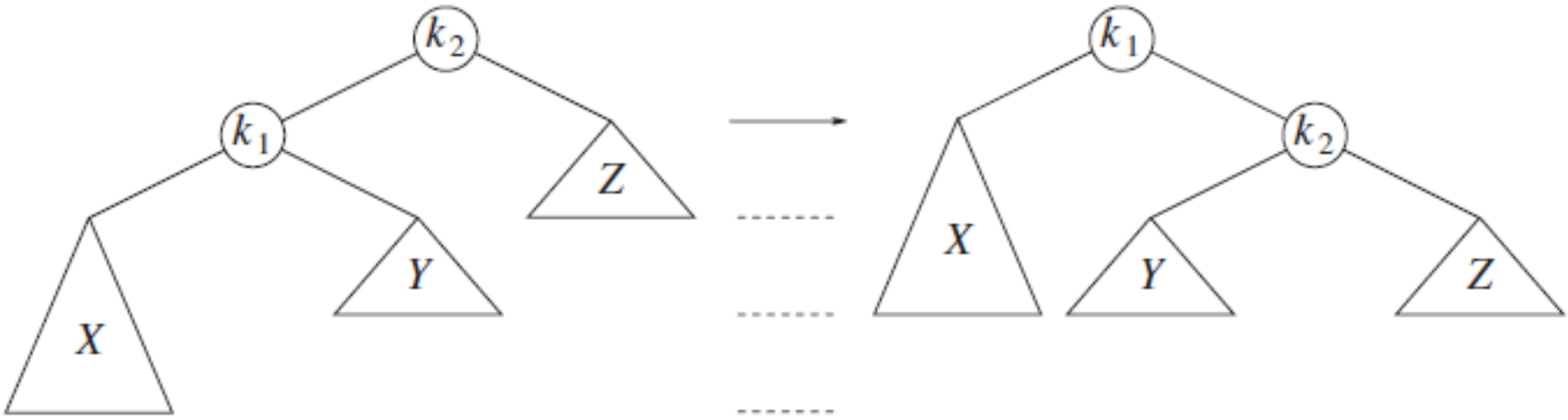


Figure 4.31 Single rotation to fix case 1

- k_1 will be the new root. It should be $k_2 > k_1$, so k_2 becomes the right child of k_1
- X and Z keep their parents.
- Subtree Y that holds items between k_1 and k_2 can be placed as k_2's left child.

Single (Left) Rotation for Case 4: right-right case

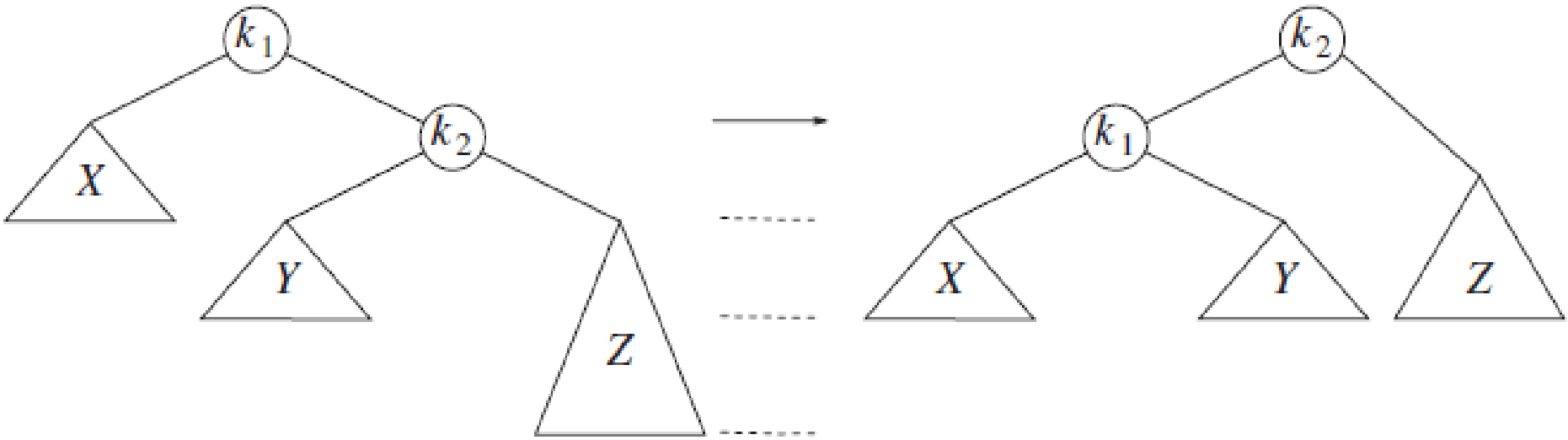
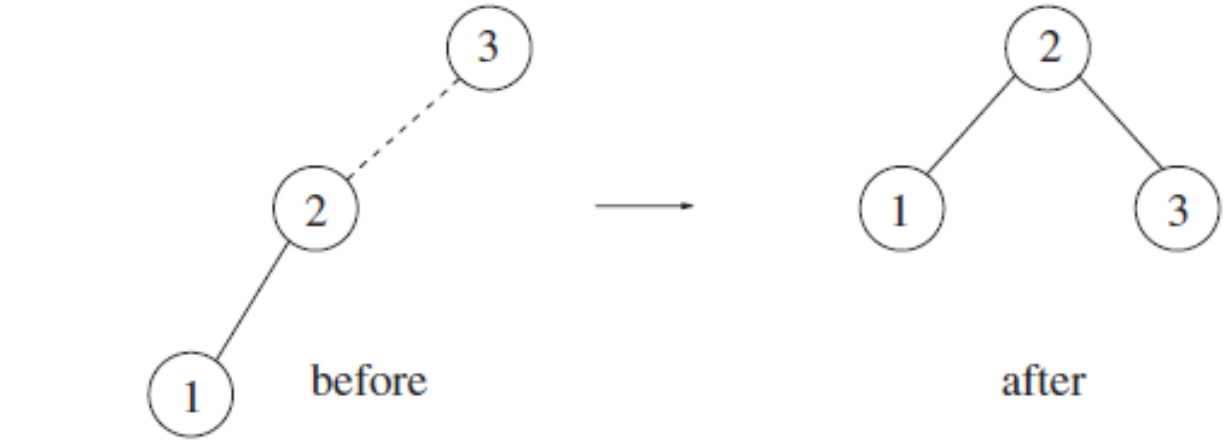


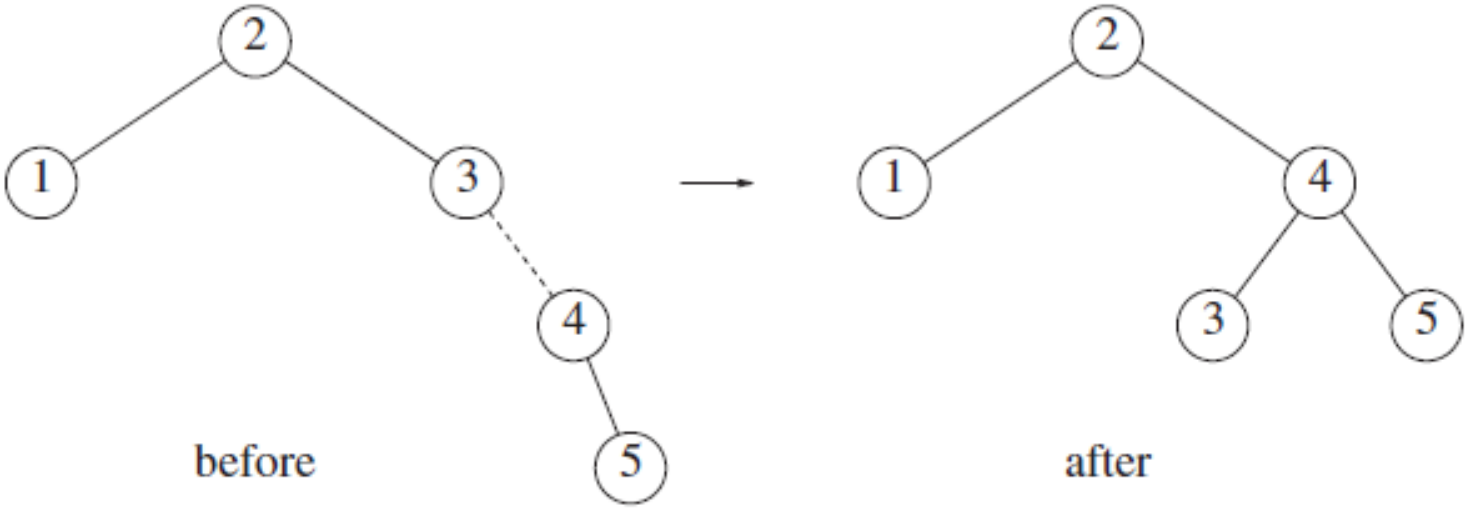
Figure 4.33 Single rotation fixes case 4

- k_2 will be the new root. It should be $k_1 < k_2$, so k_1 becomes the left child of k_2
- X and Z keep their parents.
- Subtree Y that holds items between k_1 and k_2 can be placed as k_1's right child.

Example

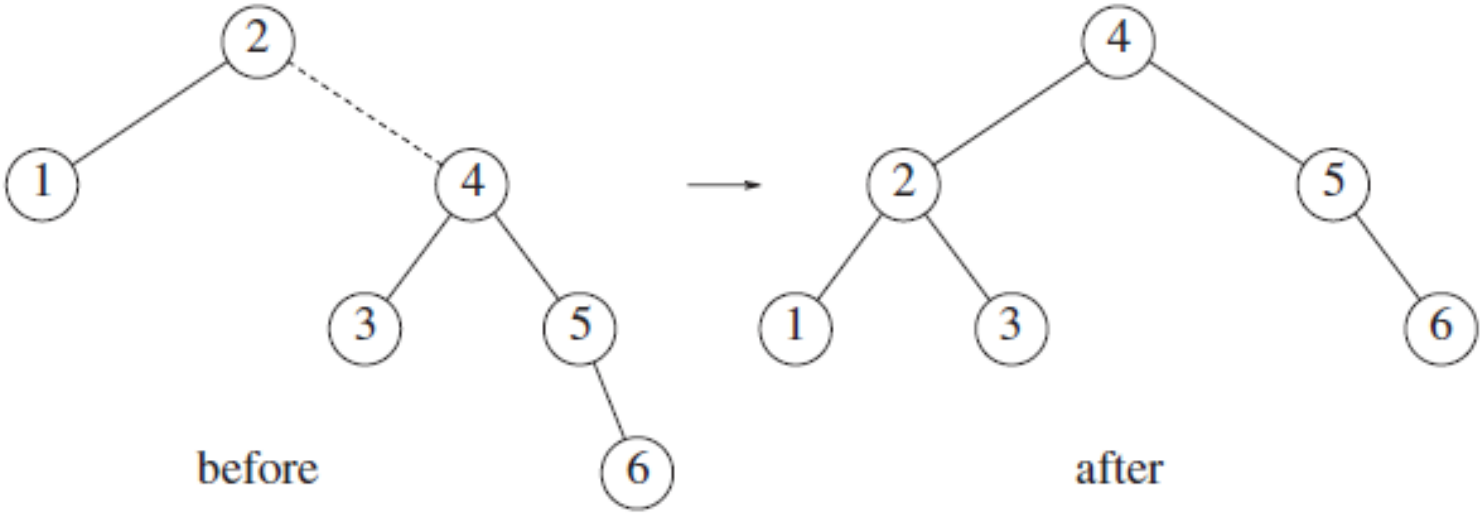
Suppose we insert 3,2,1 and 4 to 7 to an empty AVL tree.

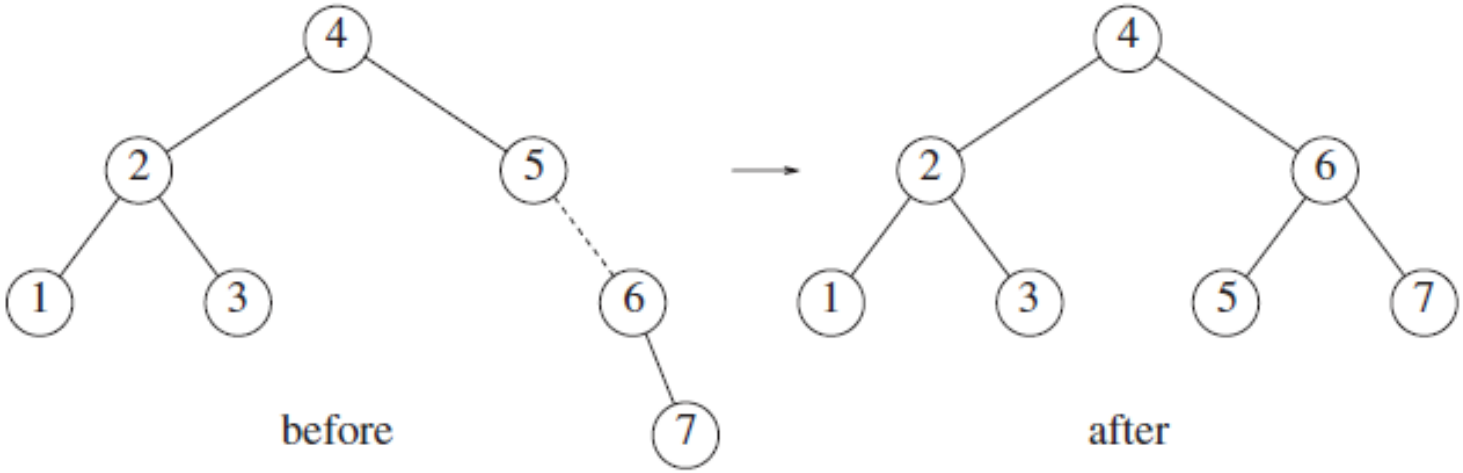




Example

Continue: insert 6 and 7 to the AVL tree.





Why does not Single Rotation work for Case 2

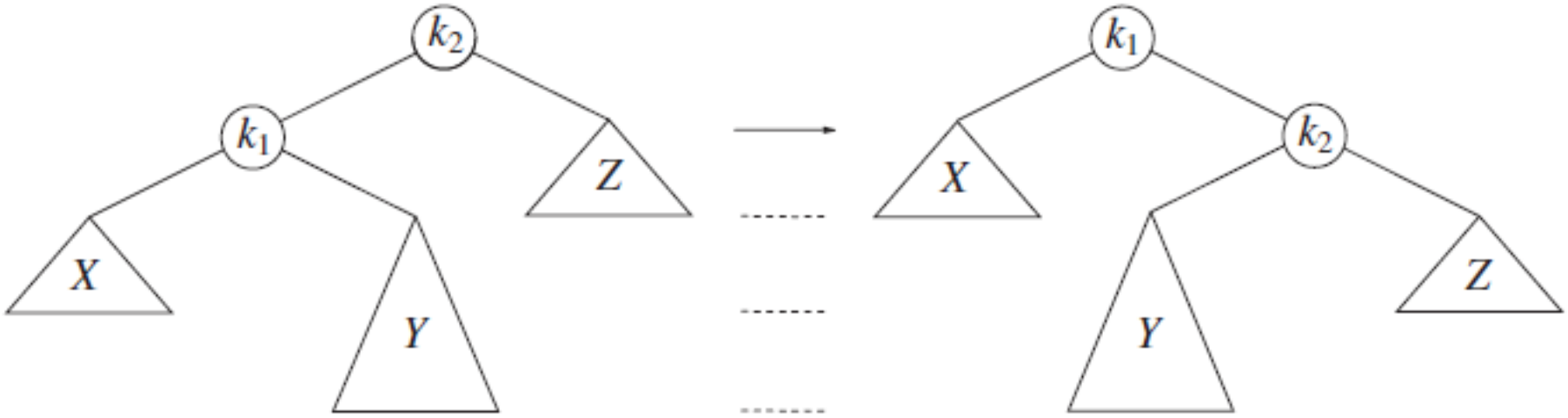


Figure 4.34 Single rotation fails to fix case 2

- subtree Y is too deep and single rotation does not make it any less deep

Double (Left,Right) Rotation for Case 2: left-right case

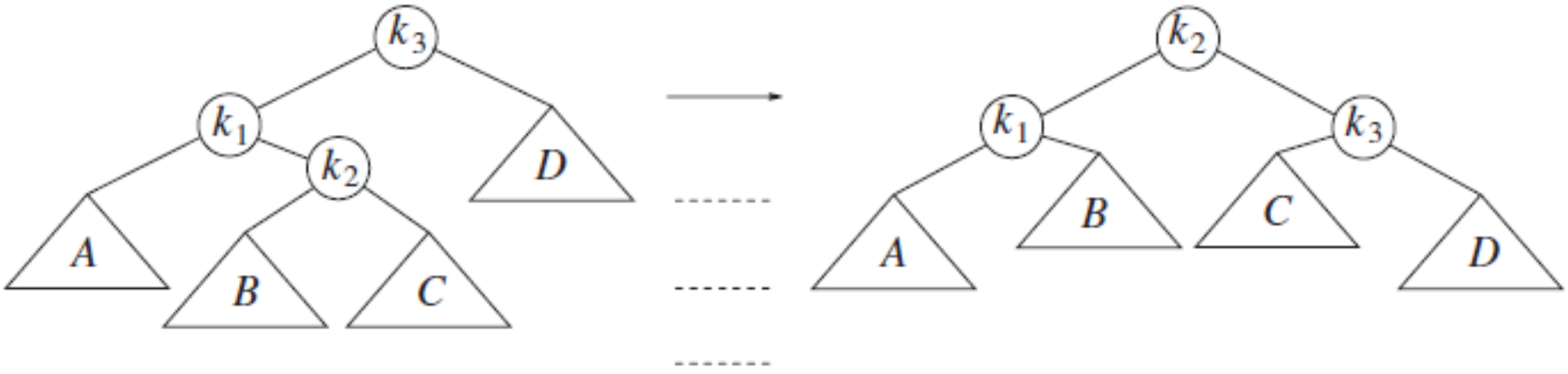


Figure 4.35 Left–right double rotation to fix case 2

- Left Rotation between k_2 and k_1
- Right rotation between k_2 and k_3

Double (Right,Left) Rotation for Case 3: right-left case

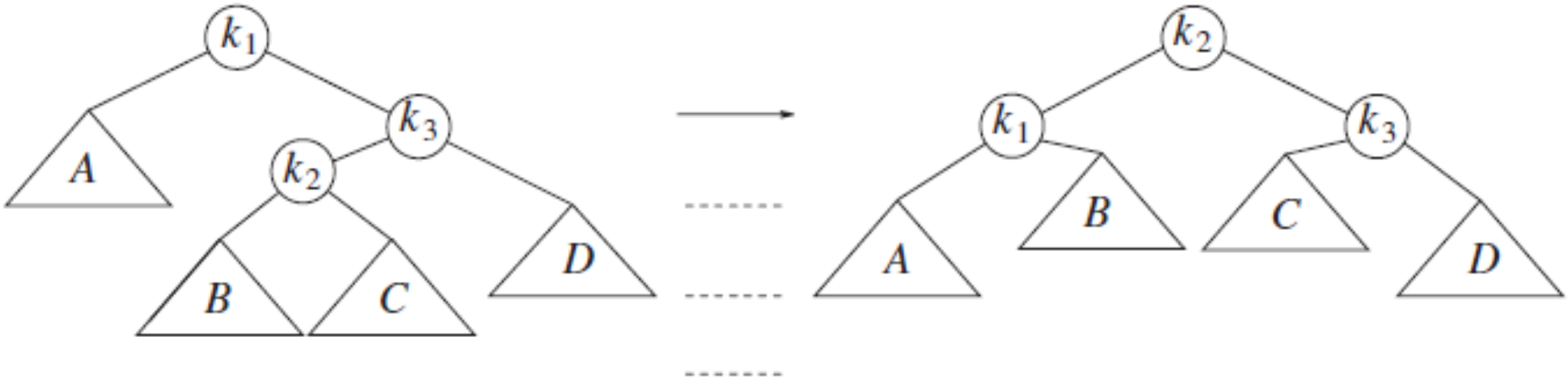
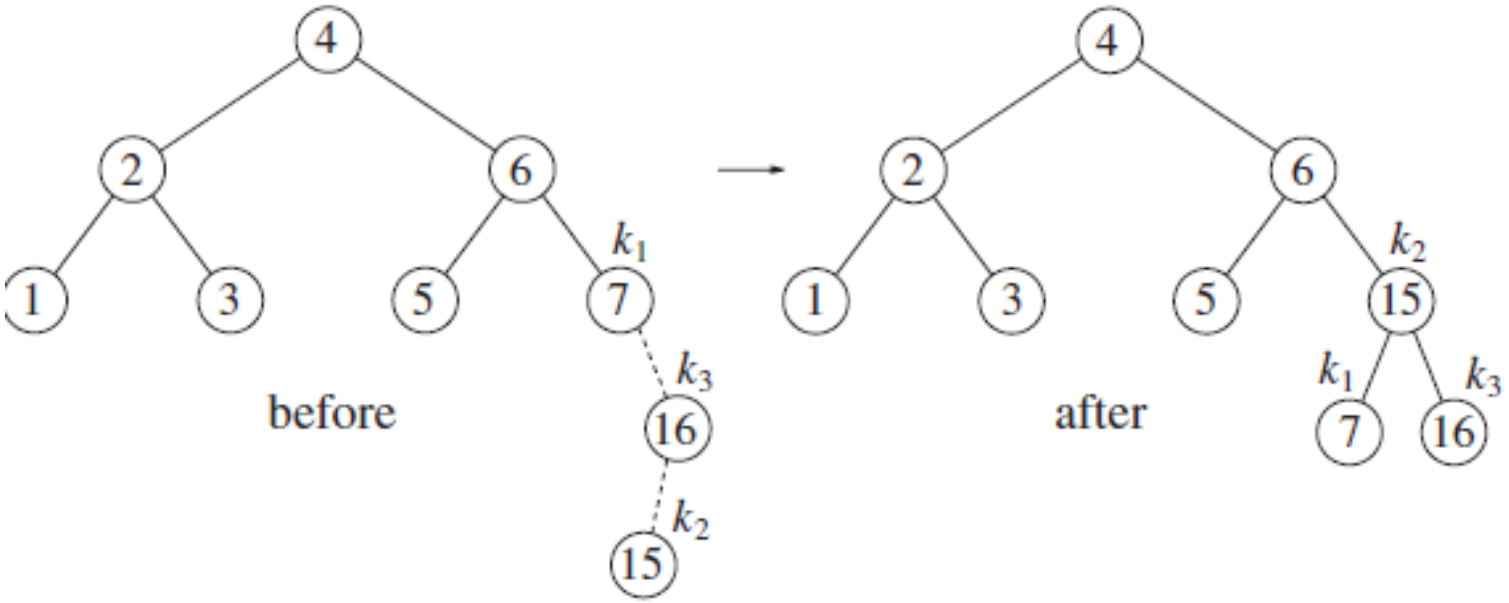


Figure 4.36 Right–left double rotation to fix case 3

- Right Rotation between k_2 and k_3
- Left rotation between k_2 and k_1

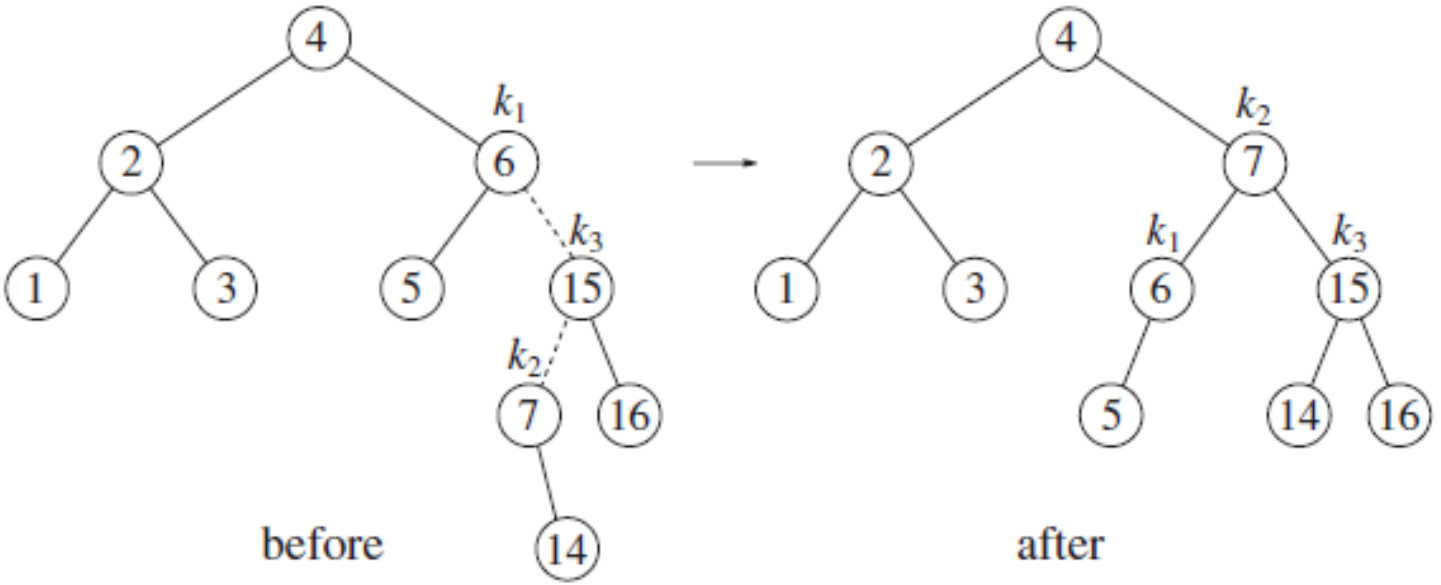
Example continued

continue our previous example by inserting 14 through 16 in reverse order. Inserting 16 is easy, since it does not destroy the balance property, but inserting 15 causes a height imbalance at node 7. This is case 3, which is solved by a right–left double rotation.



Example continued

we insert 14, which also requires a double rotation. Here the double rotation that will restore the tree is again a right–left double rotation that will involve 6, 15, and 7.



Rotations Running Time

- Constant number of link rearrangements
- Double rotation needs twice as many, but still constant
  - So AVL rotations do not change O(d) running time of the BST operations

Summary-Why AVL Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
- The cost of these operations may become O(n) for a skewed Binary tree.
- If we make sure that the height of the tree remains O(log(n)) after every insertion and deletion, then we can guarantee an upper bound of O(log(n)) for all these operations.
- The height of an AVL tree is always O(log(n)) where n is the number of nodes in the tree.

B-trees

- Thus far, we have assumed that we can store an entire data structure in the main memory of a computer. Suppose, however, that we have more data than can fit in main memory, meaning that we must have the data structure reside on disk.
- A B-tree is a tree data structure suitable for disk drives (databases, file systems)
  - It may take up to 11 ms to access data on disk.
  - Today's modern CPUs can execute billions of instructions per second.
  - Therefore, it's worth spending a few CPU cycles to reduce the number of disk accesses (Memory access is much faster than disk access)

B-trees

A B-tree is an m-ary tree.

- We can use B-Trees to reduce the number of disk accesses. Basic idea:
- Read an entire B-Tree node (containing M items) into memory in single disk access. Find the next reference using binary search.

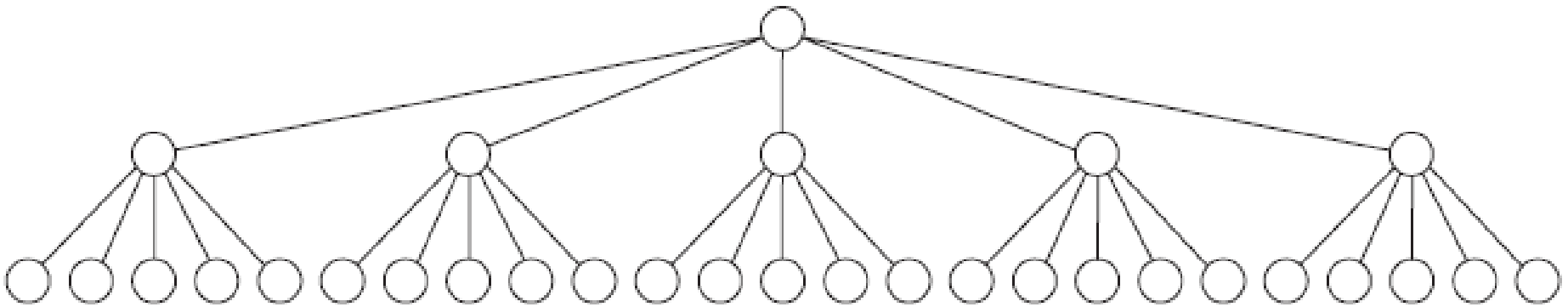


Figure 4.59 5-ary tree of 31 nodes has only three levels

Properties of B-trees

A B-tree of order M is an M-ary tree with the following properties:

- The data items are stored at leaves.
- The nonleaf nodes store up to M - 1 keys to guide the searching; key i represents the smallest key in subtree i + 1.
- The root is either a leaf or has between two and M children
- All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
- All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data items, for some L

Example B-tree

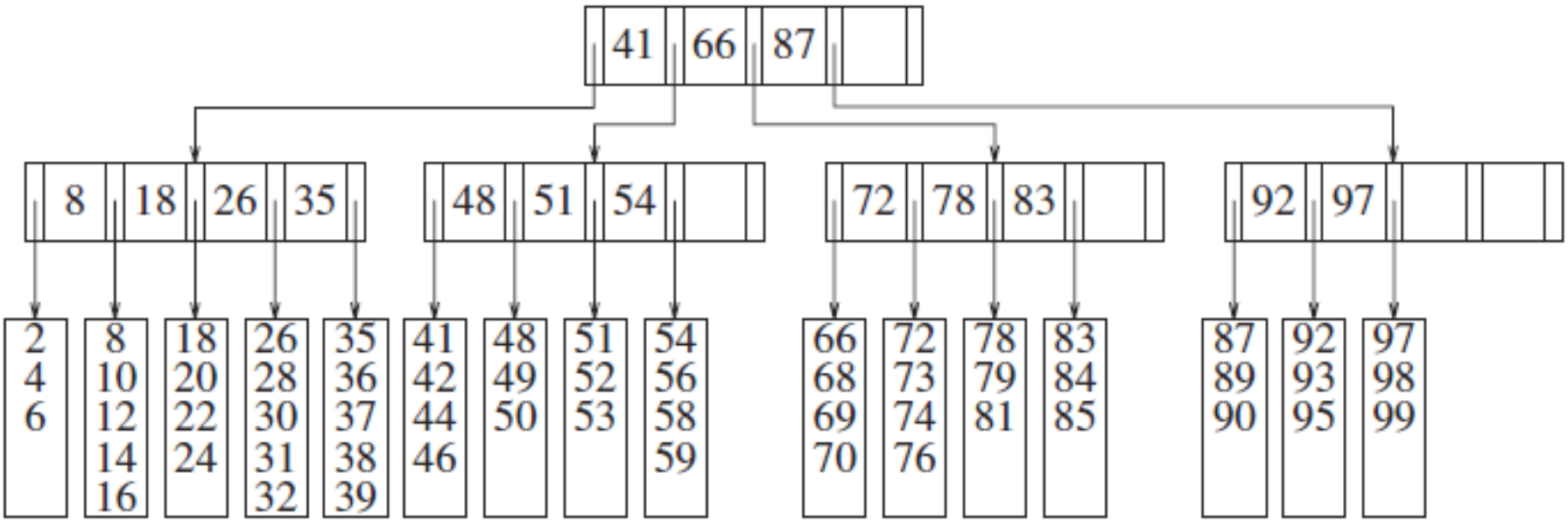


Figure 4.60 B-tree of order 5

- Only leafs store full (key, value) pairs.
- Internal nodes only contain keys to help find the right leaf.
- Insert/removal only at leafs

B-tree insertion

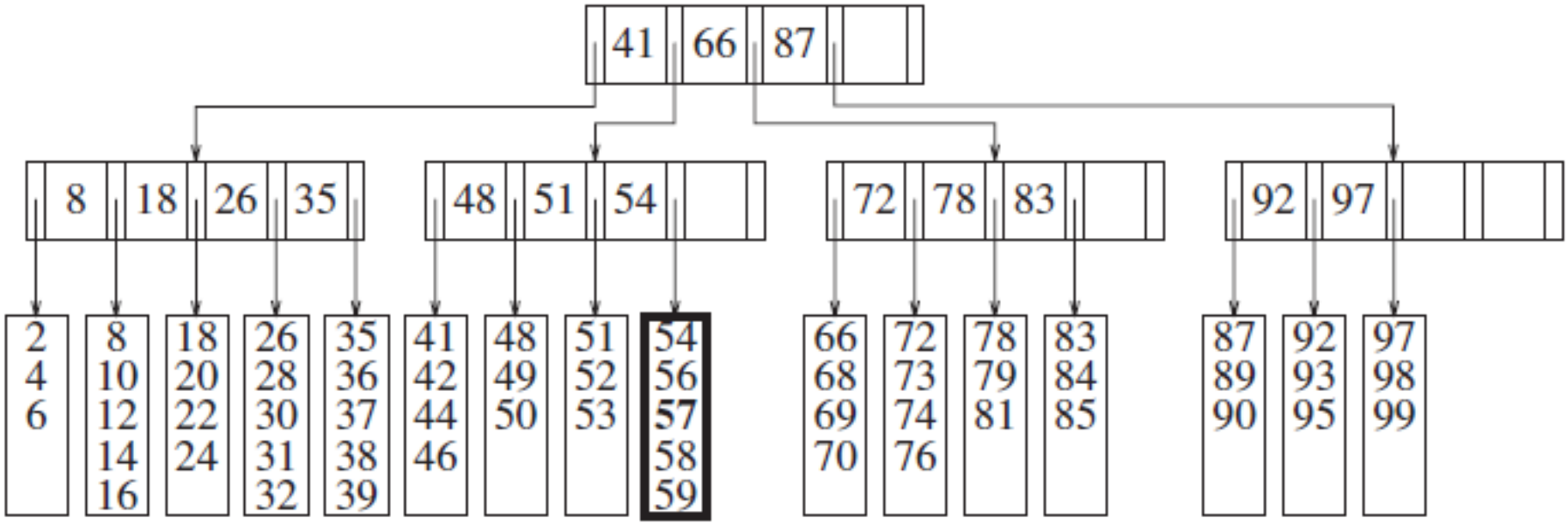


Figure 4.61 B-tree after insertion of 57 into the tree in Figure 4.60

B-tree insertion 2

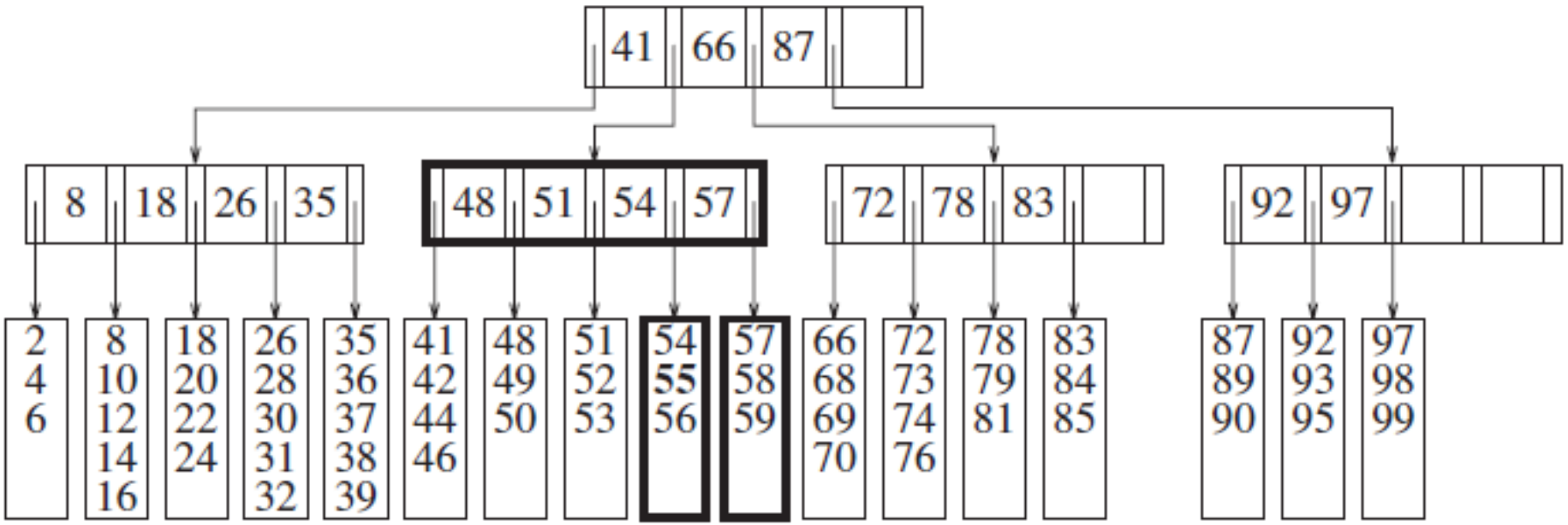


Figure 4.62 Insertion of 55 into the B-tree in Figure 4.61 causes a split into two leaves

The leaf where 55 wants to go is already full. The solution is simple: Since we now have L+1 items, we split them into two leaves, both guaranteed to have the minimum number of data records needed. We form two leaves with three items each. Although splitting nodes is time-consuming because it requires at least two additional disk writes, it is a relatively rare occurrence

B-tree insertion 3

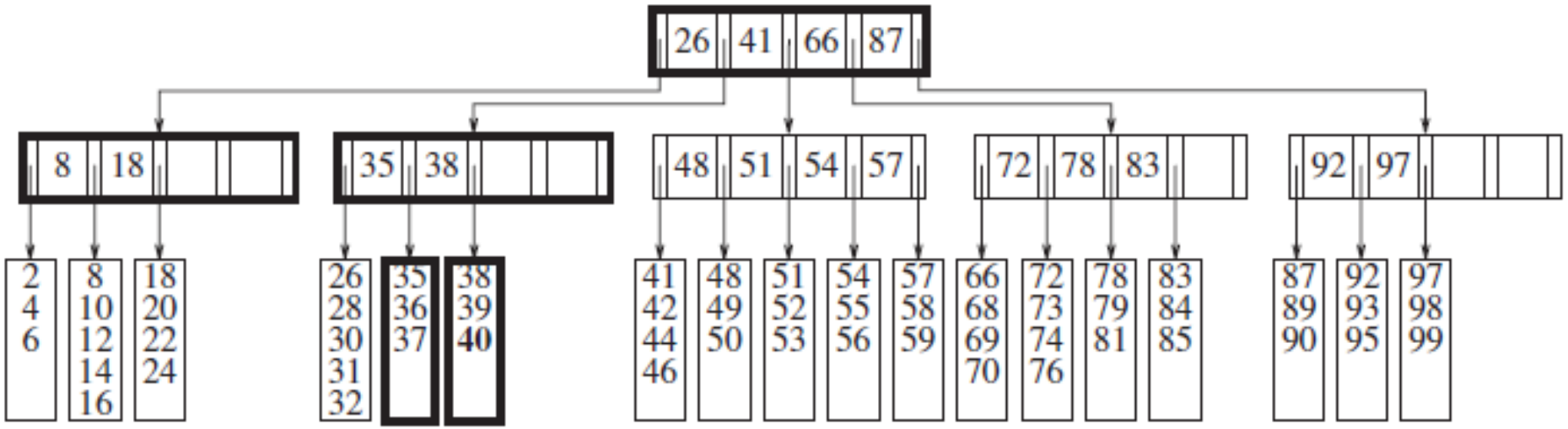


Figure 4.63 Insertion of 40 into the B-tree in Figure 4.62 causes a split into two leaves and then a split of the parent node

When the parent is split, we must update the values of the keys and also the parent's parent, thus incurring an additional two disk writes (so this insertion costs five disk writes).

B-tree deletion

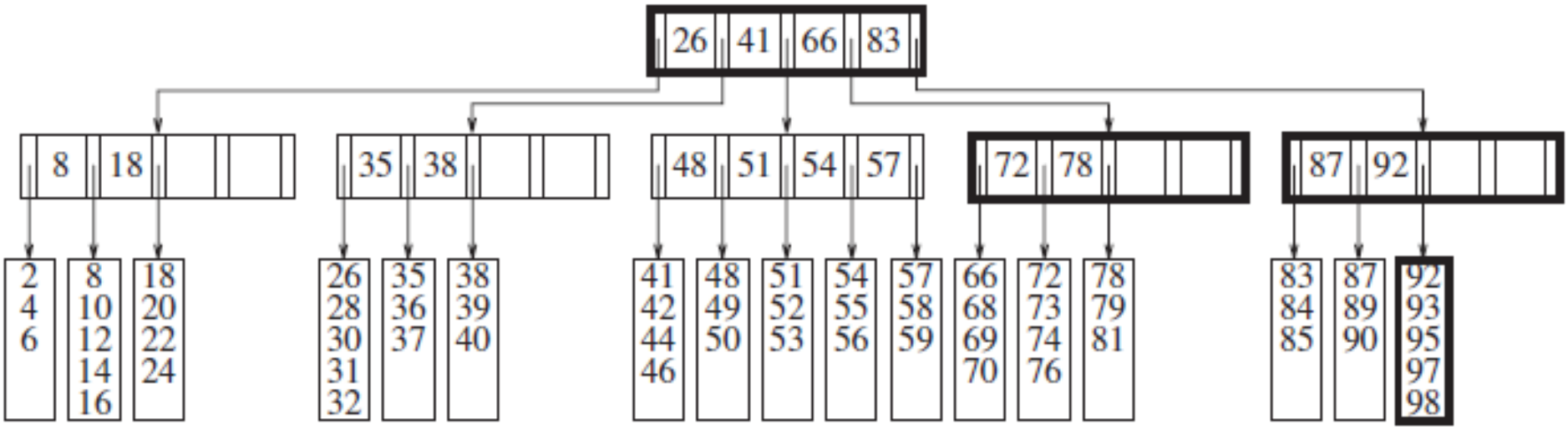


Figure 4.64 B-tree after the deletion of 99 from the B-tree in Figure 4.63

Since the leaf has only two items, and its neighbor is already at its minimum of three, we combine the items into a new leaf of five items. As result, the parent has only two children. However, it can adopt from a neighbor because the neighbor has four children. As a result, both have three children