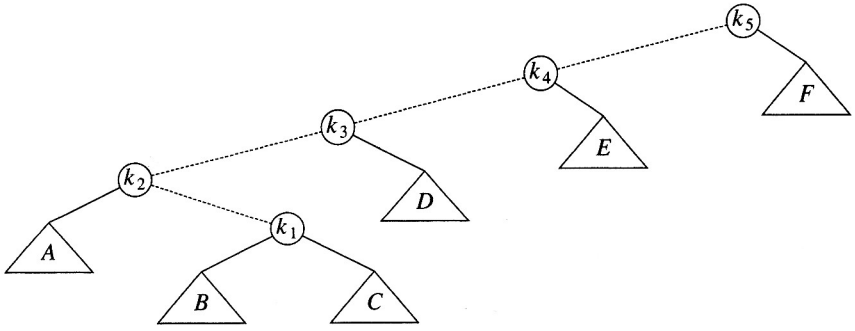
Splay Trees

1 Splay Trees

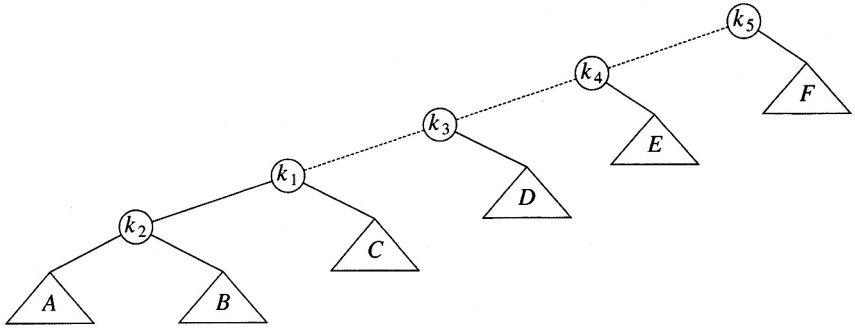
- Splay Trees are a variant of Binary Search Trees that guarantee $O(M \, log \, N)$ performance for any M consecutive tree operations (inserts, finds, deletions).
- Some tree accesses may be $O(N)$, but over time, the accesses are efficient.
- Main idea: $O(N)$ accesses for a Binary Search Tree is not so bad if it happens infrequently. Splay Trees guarantee that *repeated* bad accesses don't occur.
- How it works: once we find a bad access, we move the bad node up the tree so it won't happen again.
- What we do: take a bad node, and througha series of AVL like rotations, we move the offending node to the root of the tree. Then, future access to that node are efficient. The tree also gets better balanced as a side effect.
- In real world, practical situations, a node is often accessed more than once, making this strategy payoff.
- Splay Trees do not require maintaining height info for rebalancing as AVL trees do.

2 Splaying

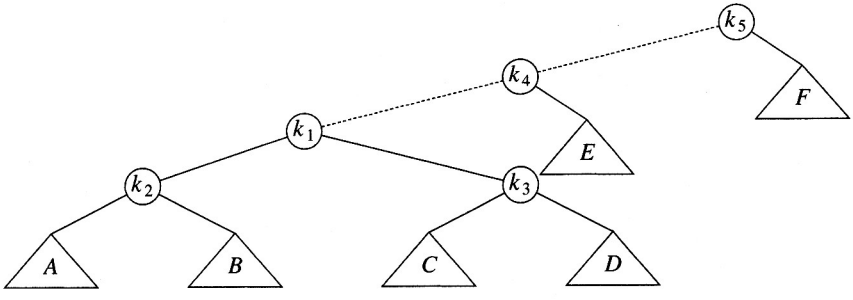
- A simple strategy is to take the node that was accessed and do single AVL-like rotations with its parent all the way up the tree until it is the root.
- The example in Weiss section 4.5.1 shows that this is not a good strategy. While the bad node is brought to the root, making its accesses efficient, it also drives other nodes deeper in the tree, making their access poor. This is shown in the next 2 pages, where a node k_1 is pushed to the top (root) but at the same time it drives another node k_3 down deeper into the tree and not really improving the height of the tree for future accesses.
- A pathological case can be built in which the cost of M accesses in $O(M \times N)$ as oppposed to the required $O(M \, Log \, N)$
- Splay Strategy: Let X be the node to be accessed and rotated
  - If X's parent is the root, just do a simple single rotation.
  - Otherwise, X has a parent P and a grandparent G along the access path.
  - Case I: Zig-Zag (fig. 4.45). X is a right child and P is a left child (or vice versa). Perform a double AVL rotation on the three nodes. (see Figure 4.44)
  - Case II: Zig-Zig (fig. 4.46). X and P are both left children (or right children in the symmetric case). (see Figure 4.45)
- Splaying not only moves the node up, but it also tend sto balance out the tree, roughly halving the depth of nodes along the access path.
- The example starting with fig. 4.47 shows how the splay tree works.



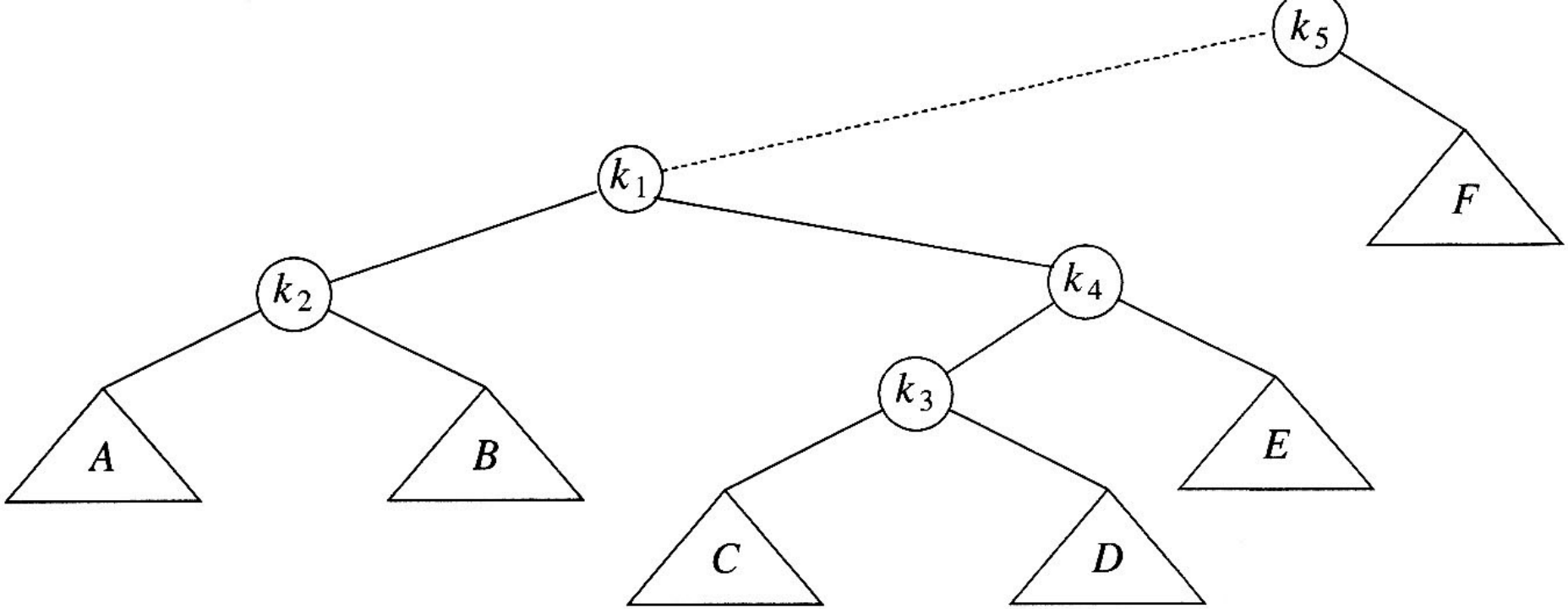
The access path is dashed. First, we would perform a single rotation between k_1 and parent, obtaining the following tree.

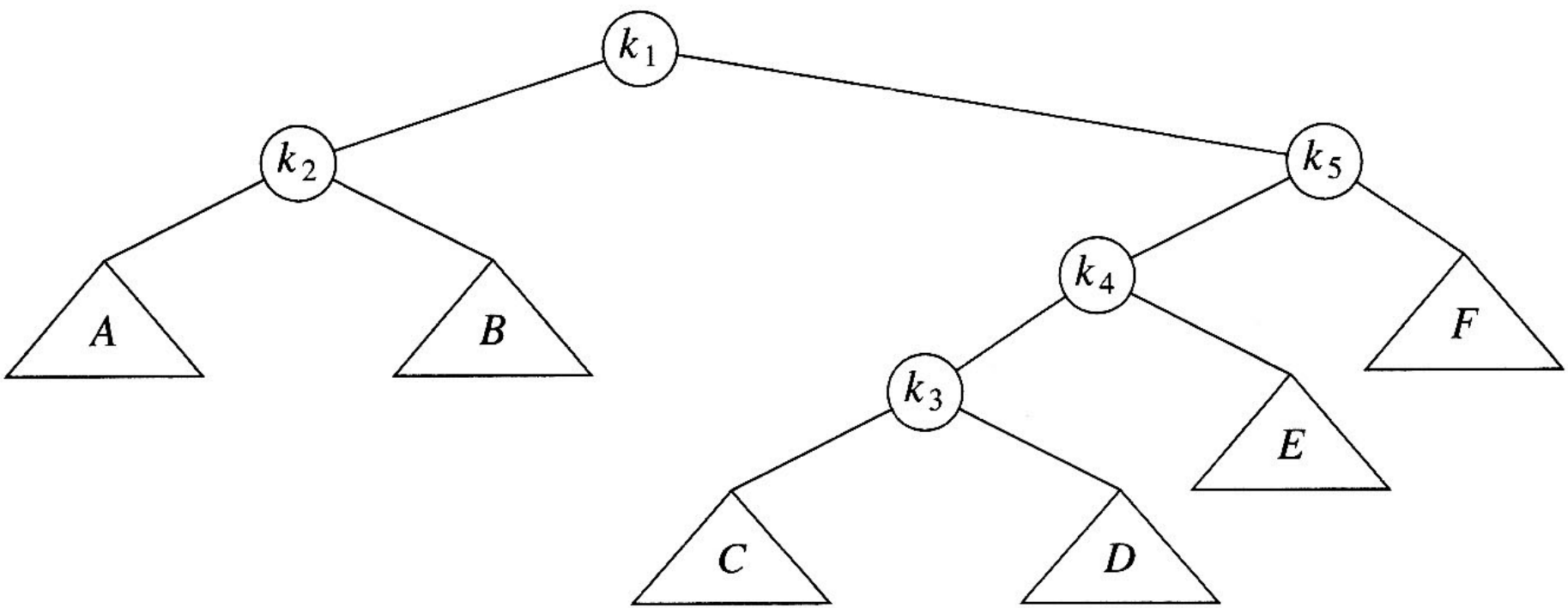


Then, we rotate between k_1 and k_3, obtaining the next tree.



Then two more rotations are performed until we reach the root.


k_5
k_1
F
k_2
k_4
A
B
k_3
E
C
D


k_1
k_2
k_5
A
B
k_4
F
k_3
E
C
D

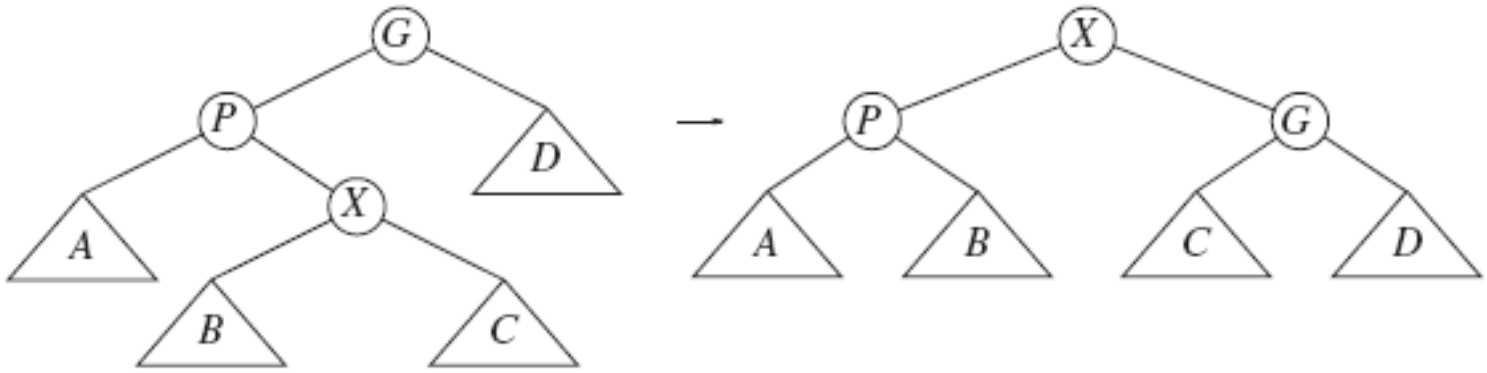


Figure 4.45 Zig-zag

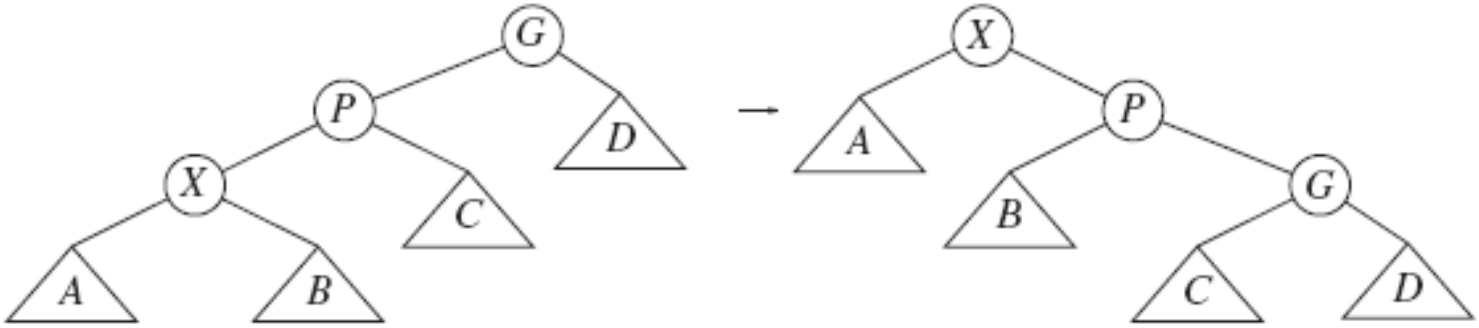


Figure 4.46 Zig-zig

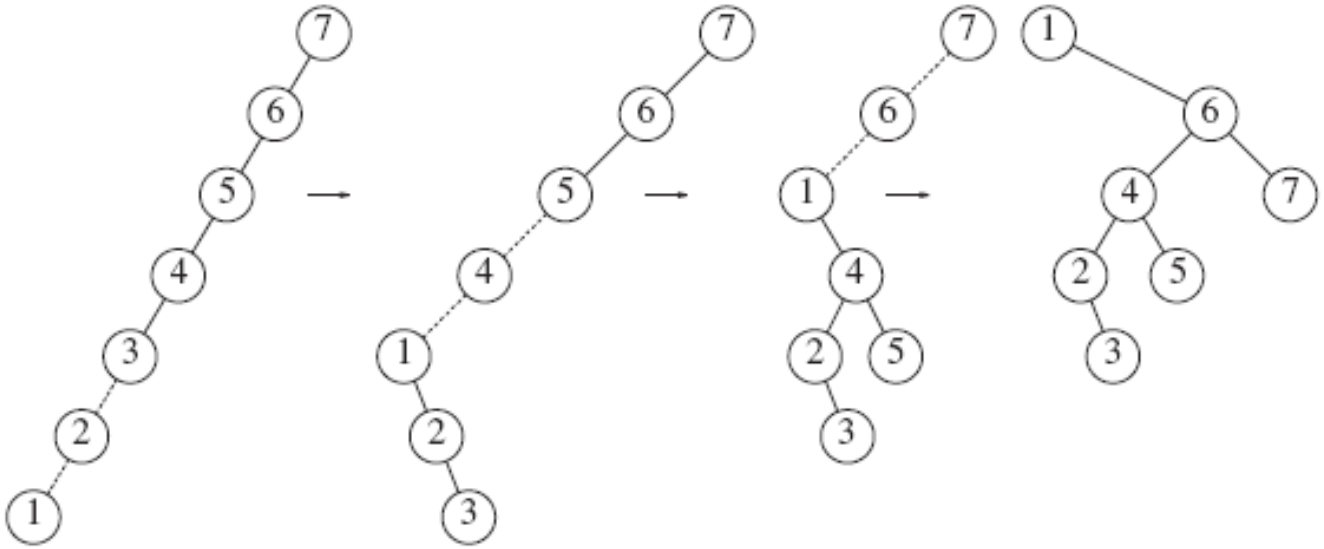


Figure 4.47 Result of splaying at node 1

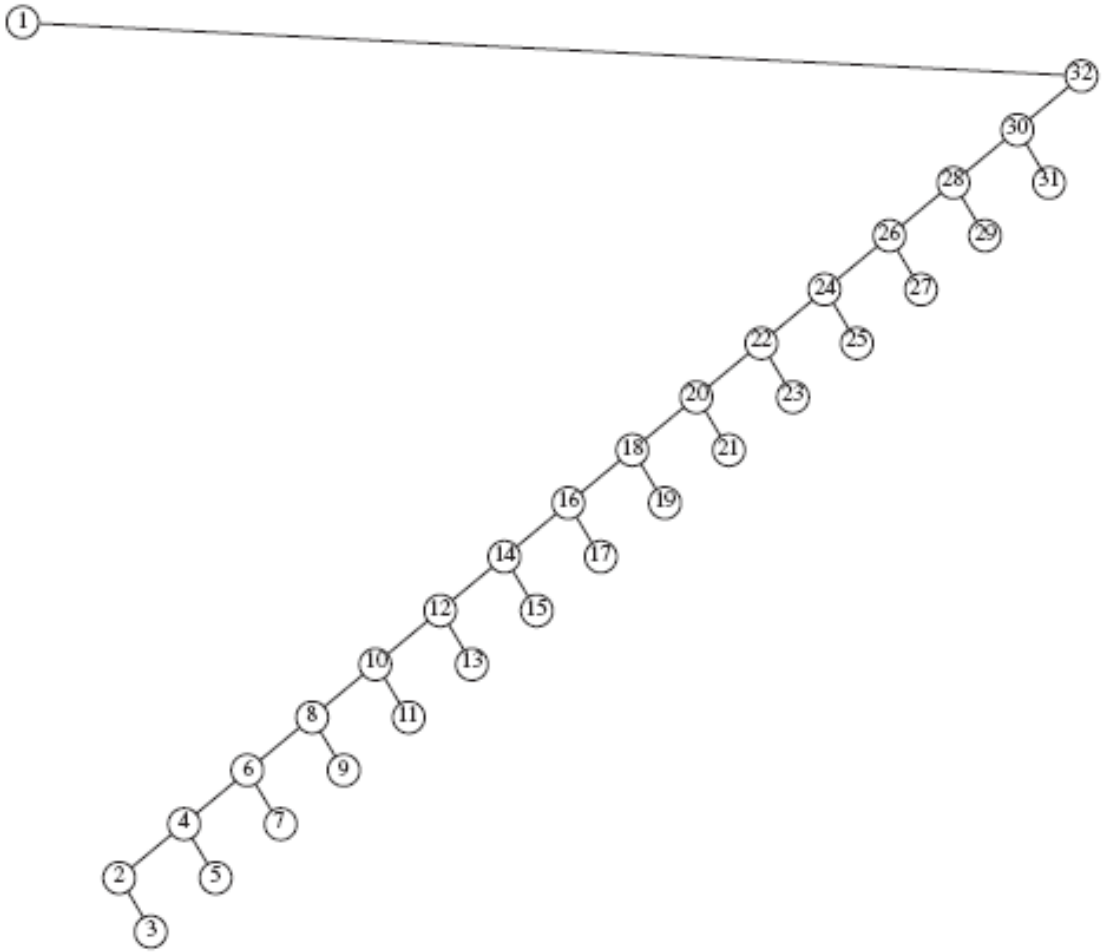


Figure 4.48 Result of splaying at node 1 a tree of all left children

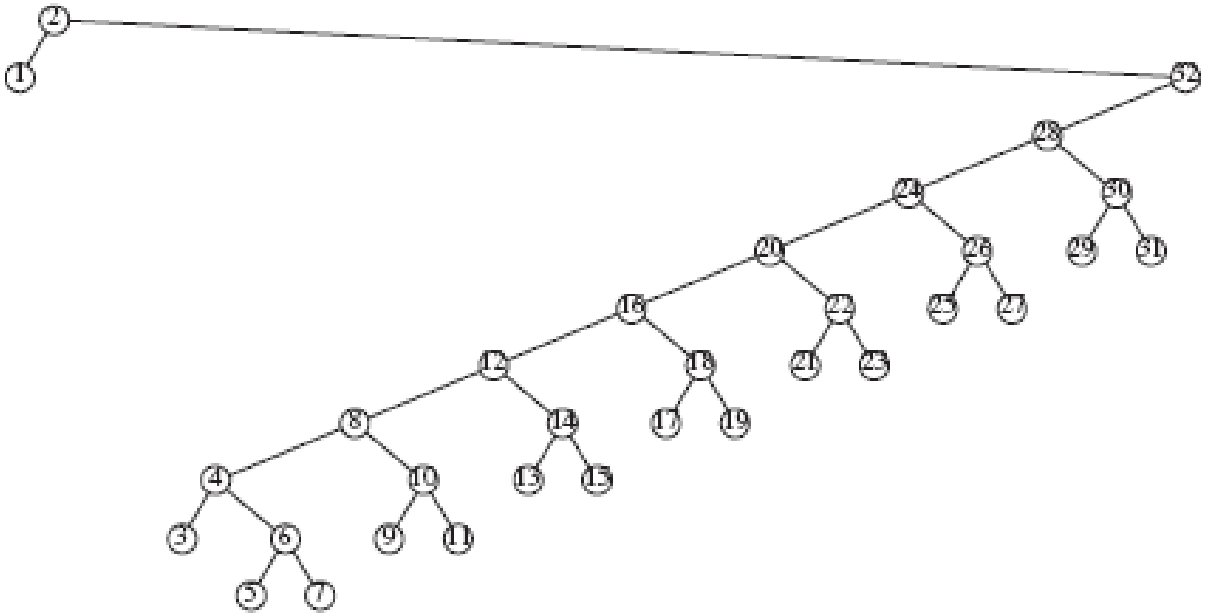


Figure 4.49 Result of splaying the previous tree at node 2

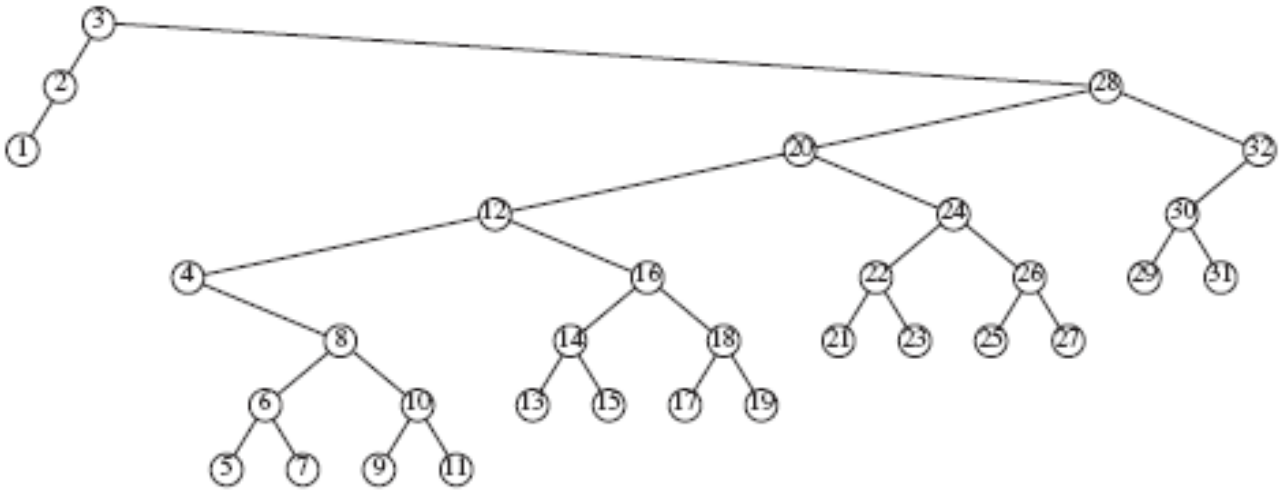


Figure 4.50 Result of splaying the previous tree at node 3

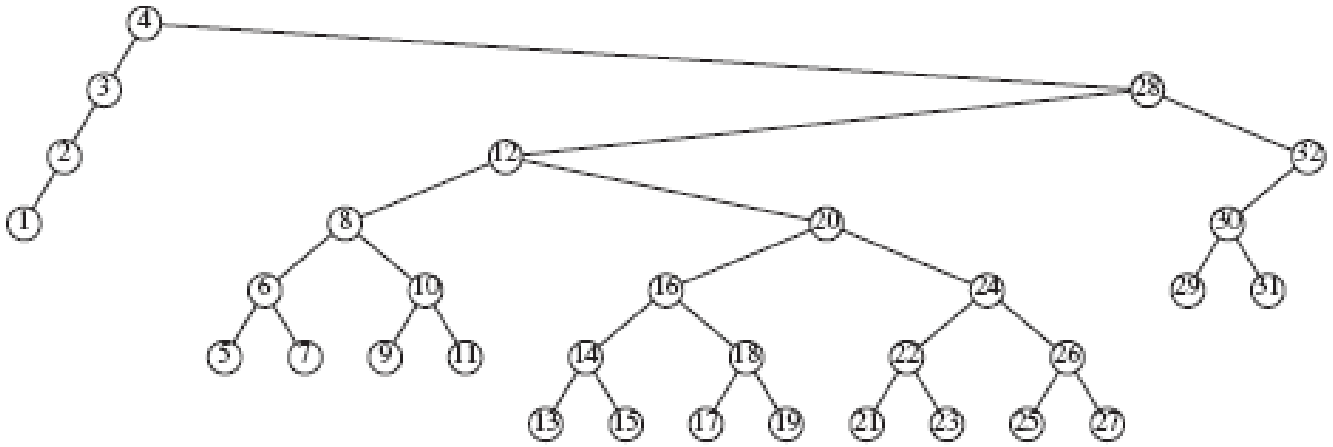


Figure 4.51 Result of splaying the previous tree at node 4

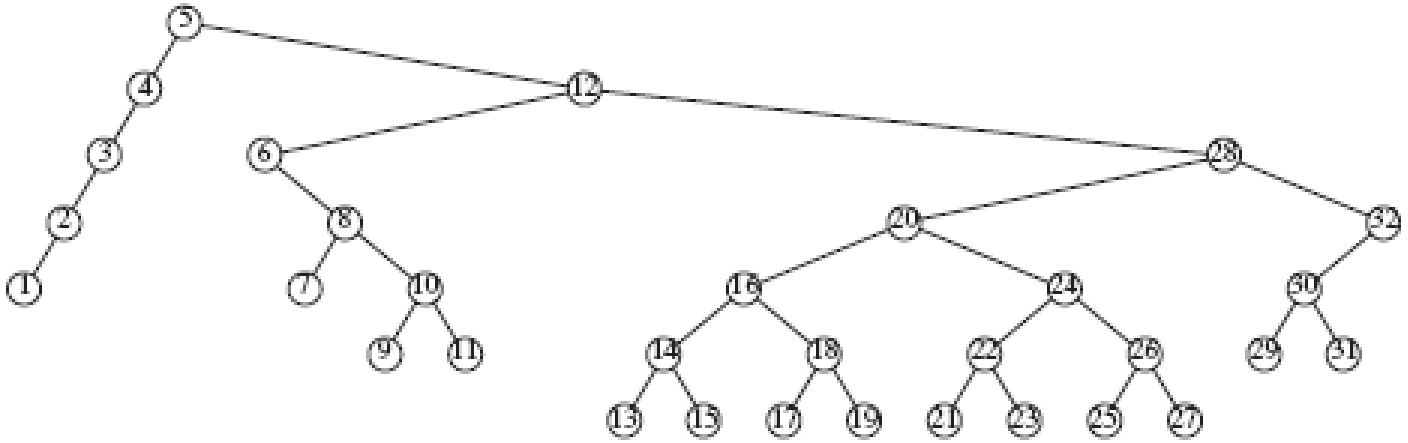


Figure 4.52 Result of splaying the previous tree at node 5

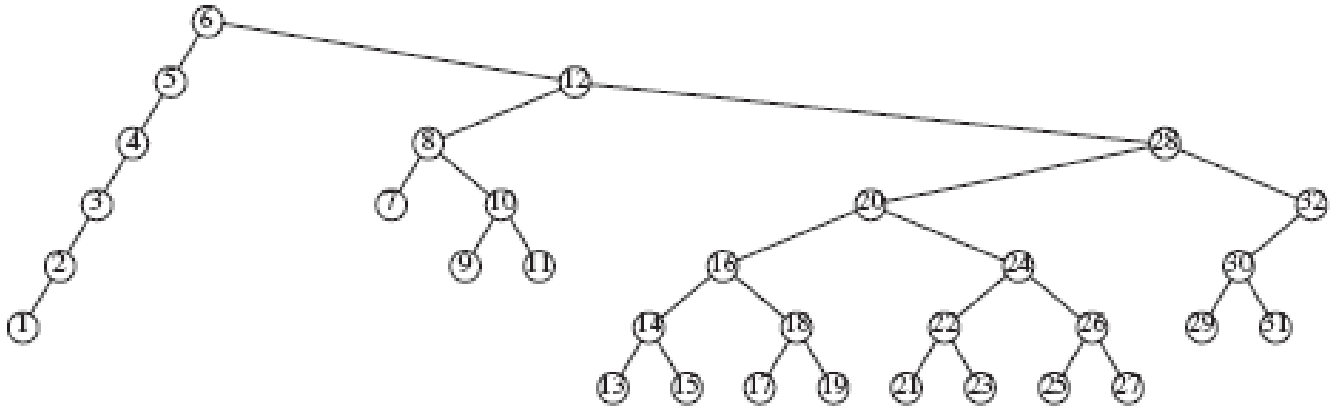


Figure 4.53 Result of splaying the previous tree at node 6

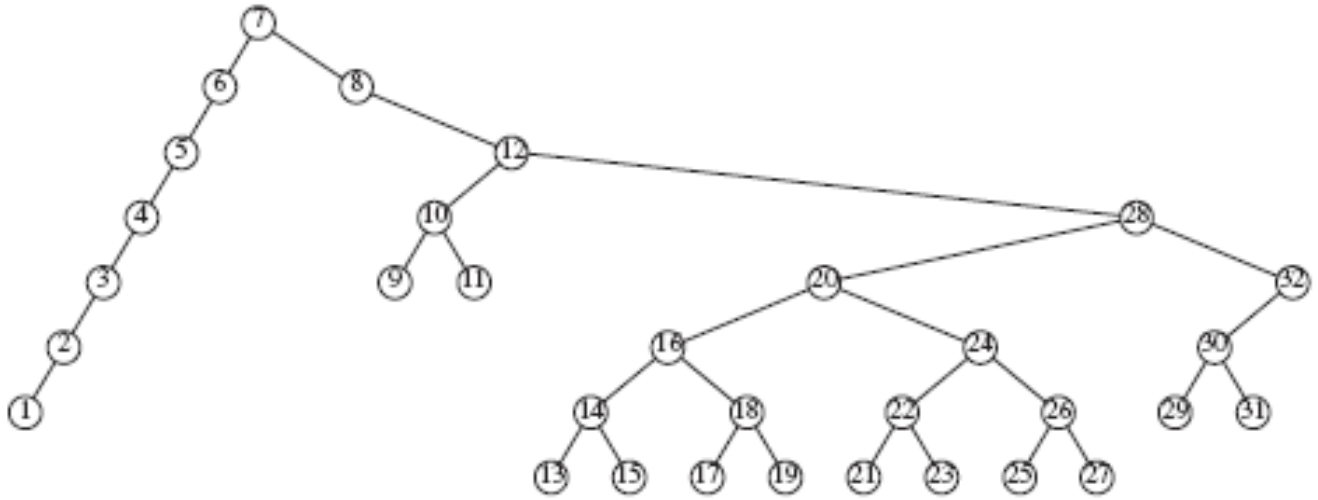


Figure 4.54 Result of splaying the previous tree at node 7

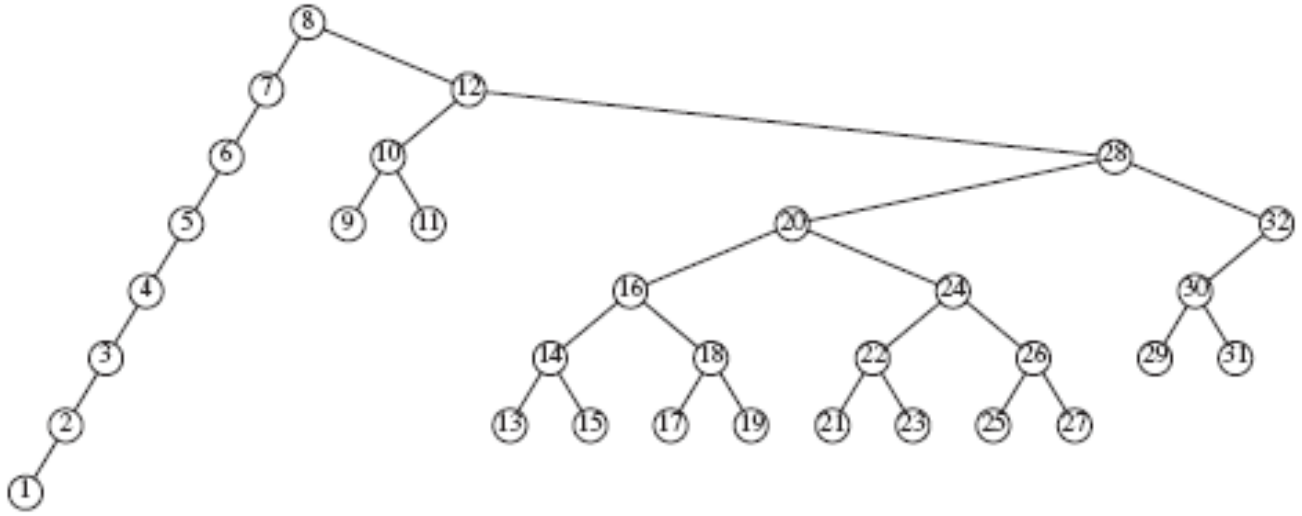


Figure 4.55 Result of splaying the previous tree at node 8

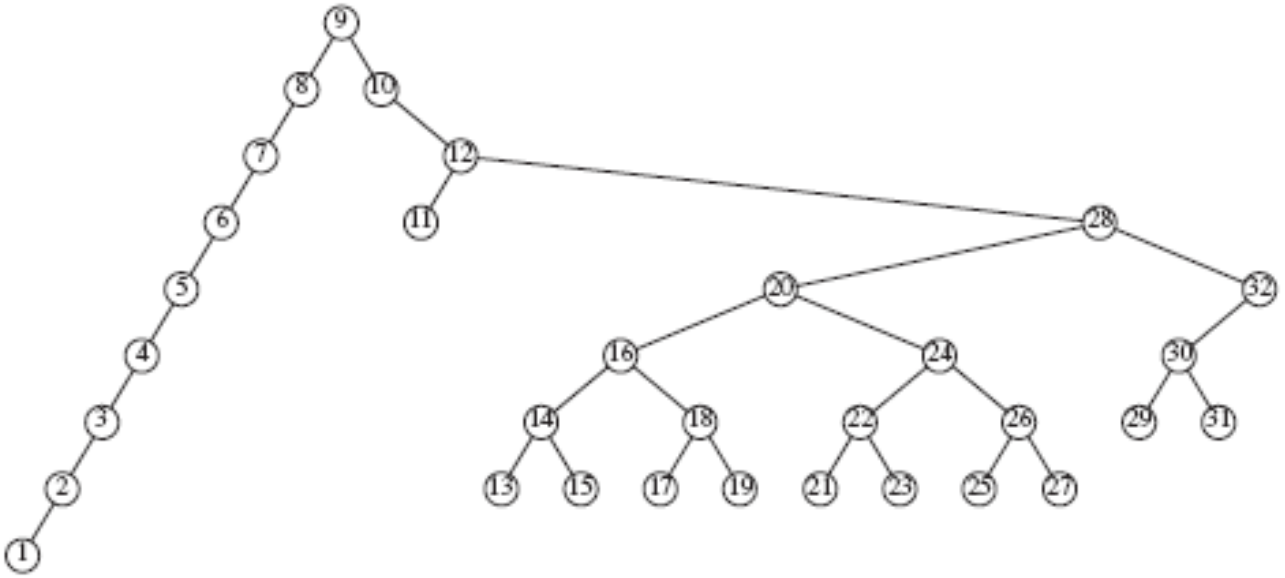


Figure 4.56 Result of splaying the previous tree at node 9