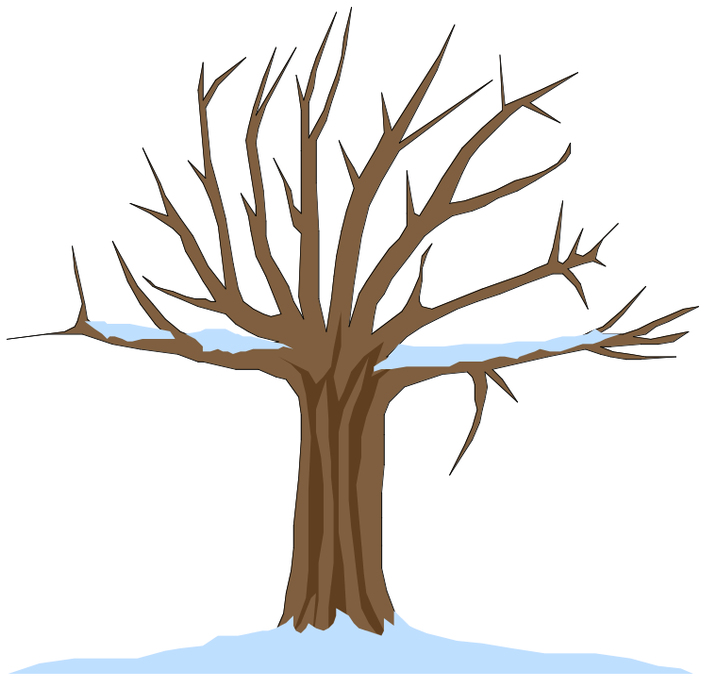
CSCI2100B Data Structures
Trees

Irwin King

king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Introduction



- General Tree
  - Definition
- Binary Search Tree
- File systems in operating systems.
- Used to evaluate arithmetic expressions.
- Show how to use trees to support searching and other operations in O(log n) average time.

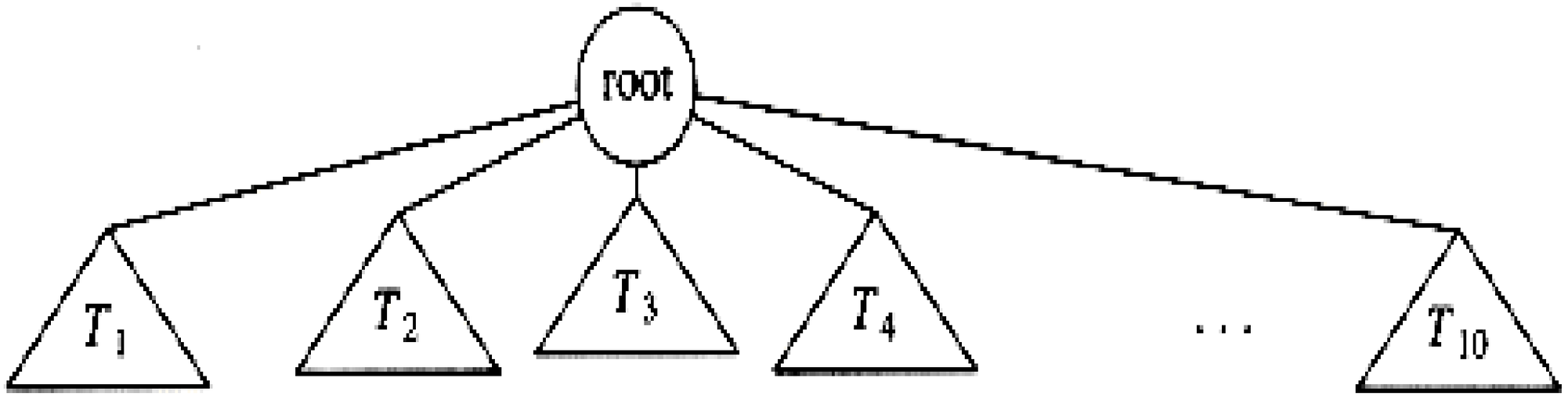
Introduction

- AVL Trees
  - It is a tree that has a balance condition so that the search of the tree is bounded.
- B-Trees
  - This is a general m-ary tree for searching.

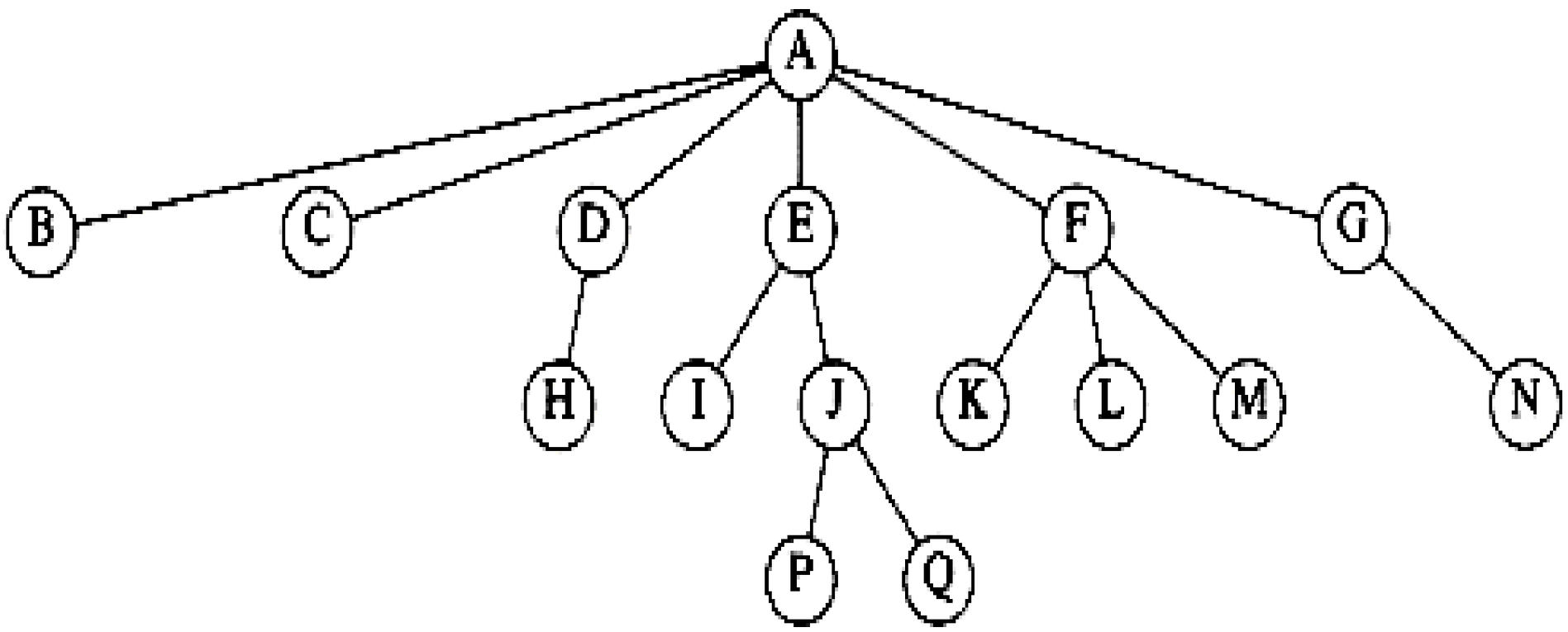
Preliminaries

- A tree is a collection of nodes.
- The collection can be empty, which is sometimes denoted as A.
- Otherwise, a tree consists of a distinguished node r, called the root, and zero or more (sub)trees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by a directed edge to r.
- The root of each subtree is said to be a child of r, and r is the parent of each subtree root.

Example



Example



Definition

- The root is A. Node F has A as a parent and K, L, and M as children.
- Each node may have an arbitrary number of children, possibly zero.
- Nodes with no children are known as leaves; the leaves in the tree above are B, C, H, I, P, Q, K, L, M, and N.
- Nodes with the same parent are siblings; thus K, L, and M are all siblings.
- Grandparent and grandchild relations can be defined in a similar manner.

Path Definition

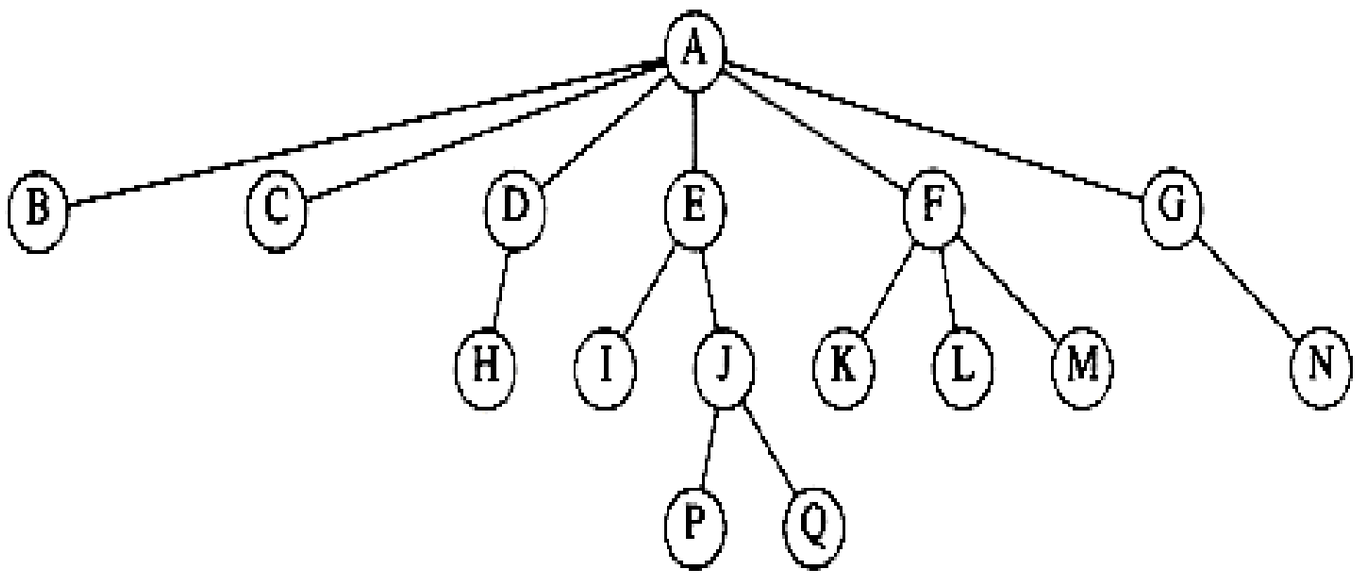
- A path from node n1 to nk is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$.
- The length of this path is the number of edges on the path, namely k -1.
- There is a path of length zero from every node to itself.
- Notice that in a tree there is exactly one path from the root to each node.

Depth Definition

- For any node n_i, the depth of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0.
- The height of n_i is the longest path from n_i to a leaf. Thus all leaves are at height 0.
- The height of a tree is equal to the height of the root.
- The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

Example

- E is at depth 1 and height 2.



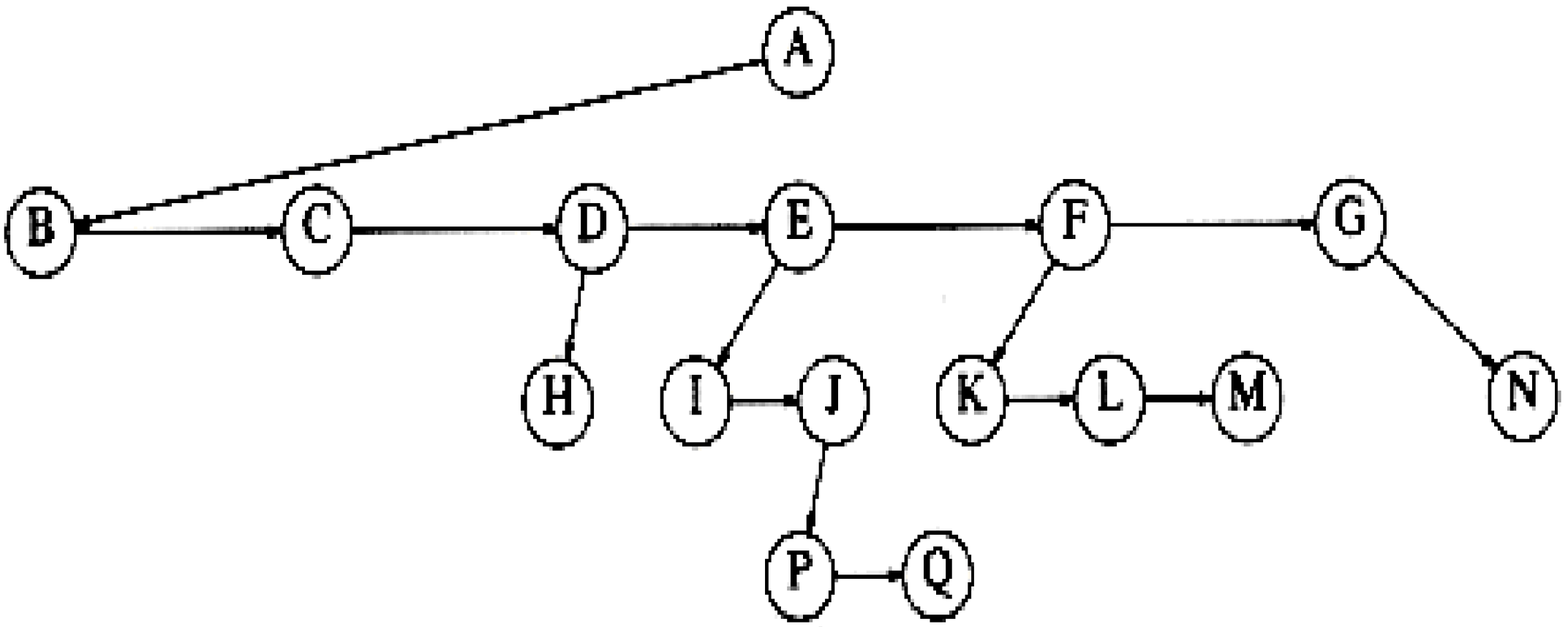
Notes

- If there is a path from n_1 to n_2,
  - then n_1 is an ancestor of n_2
  - and n_2 is a descendant of n_1.
- If $n_1 \neq n_2$,
  - then n_1 is a proper ancestor of n_2
  - and n_2 is a proper descendant of n_1.

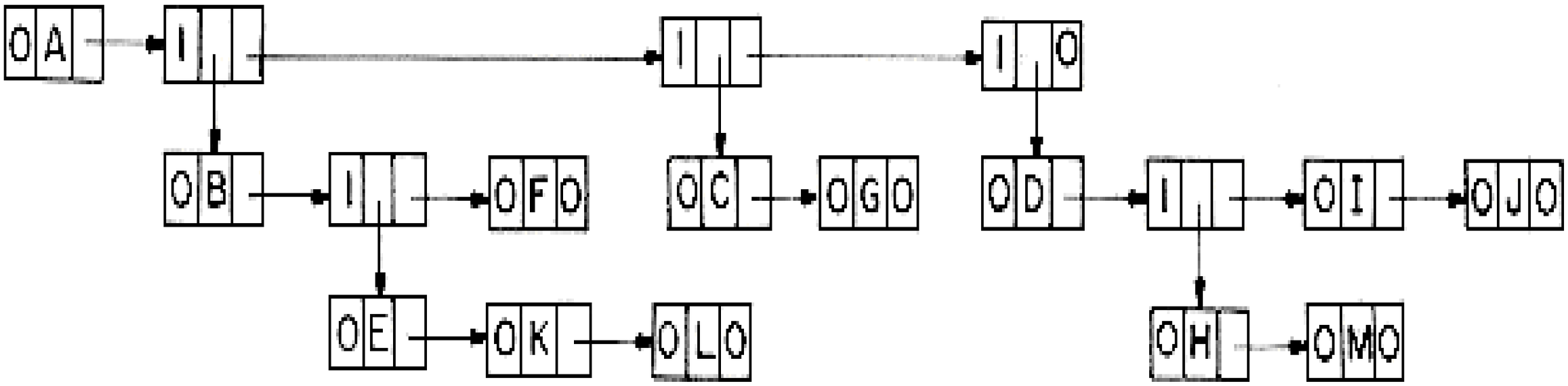
Implementation of Trees

- One way to implement a tree would be to have in each node, besides its data, a pointer to each child of the node.
- What if the node have many children?
- How do you declare it in advance?
- The solution is simple: Keep the children of each node in a linked list of tree nodes.

Example



Another Example

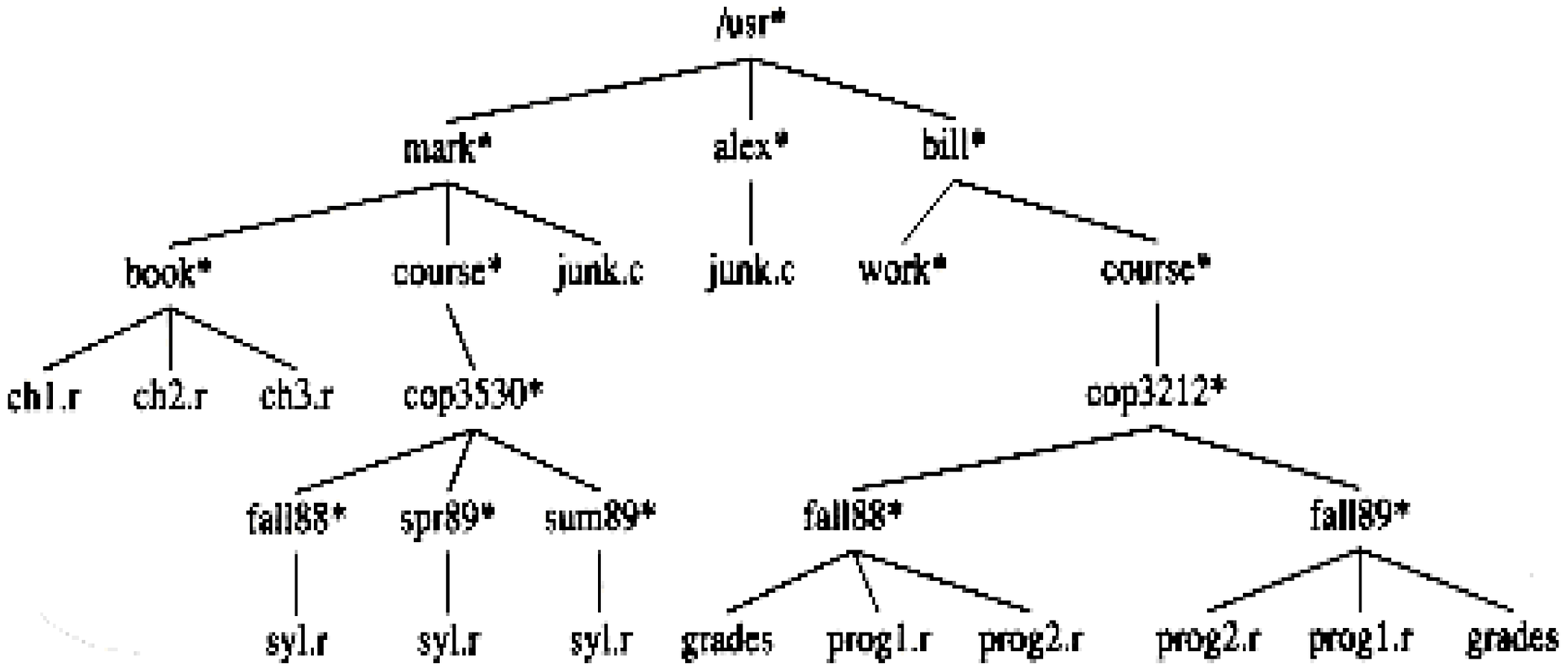


Notes

- Arrows that point downward are first_child pointers.
- Arrows that go left to right are next_sibling pointers.
- Null pointers are not drawn, because there are too many.
- Node E has both a pointer to a sibling (F) and a pointer to a child (I), while some nodes have neither.

Tree Traversal

- File directory in OSs



Tree Traversal

- Suppose we would like to list the names of all of the files in the directory.

- Our output format will be that files that are depth d will have their names indented by d tabs.

- How to perform such a task?

Intended Output

- /usr
  - mark
    - book
      - chr1.c
      - chr2.c
      - chr3.c
- course
  - cop3530
    - fall88
      - syl.r

Traversal Definition

- General Definition: to traverse a data structure is to process, however you like, every node in the data structure exactly once .

- Note: You may ``pass through'' a node as many times as you like but you must only process the node once.

Traversal Strategy

- This traversal strategy is known as a preorder traversal.
- In a preorder traversal, work at a node is performed before (pre) its children are processed.

- What is the time complexity?
  - The total amount of work is constant per node.
  - If there are n file names to be output, then the running time is O(n).

Standard Traversal Orders

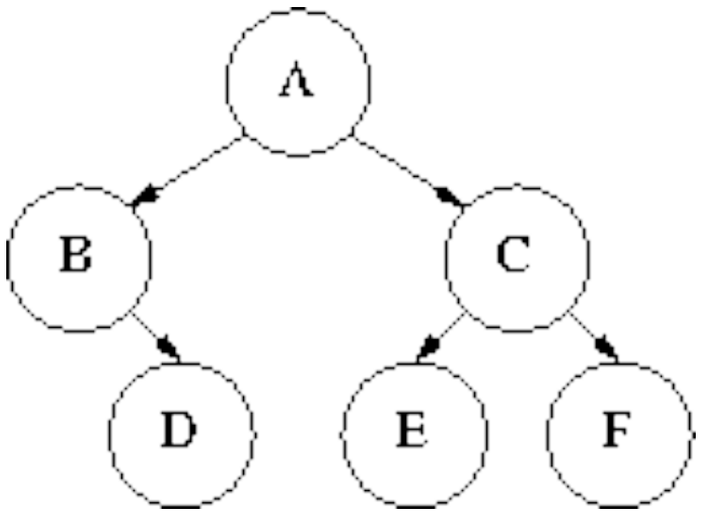
- Assuming that the left subtree is L, the right subtree is R, and the vertex is V, there are 6 ways (permutations) to organize the order to visit these three nodes in a binary tree (a tree with at most two children).
- They are:
  - VLR, LVR, LRV, VRL, RVL, RLV
  - However, not all combinations listed are useful.

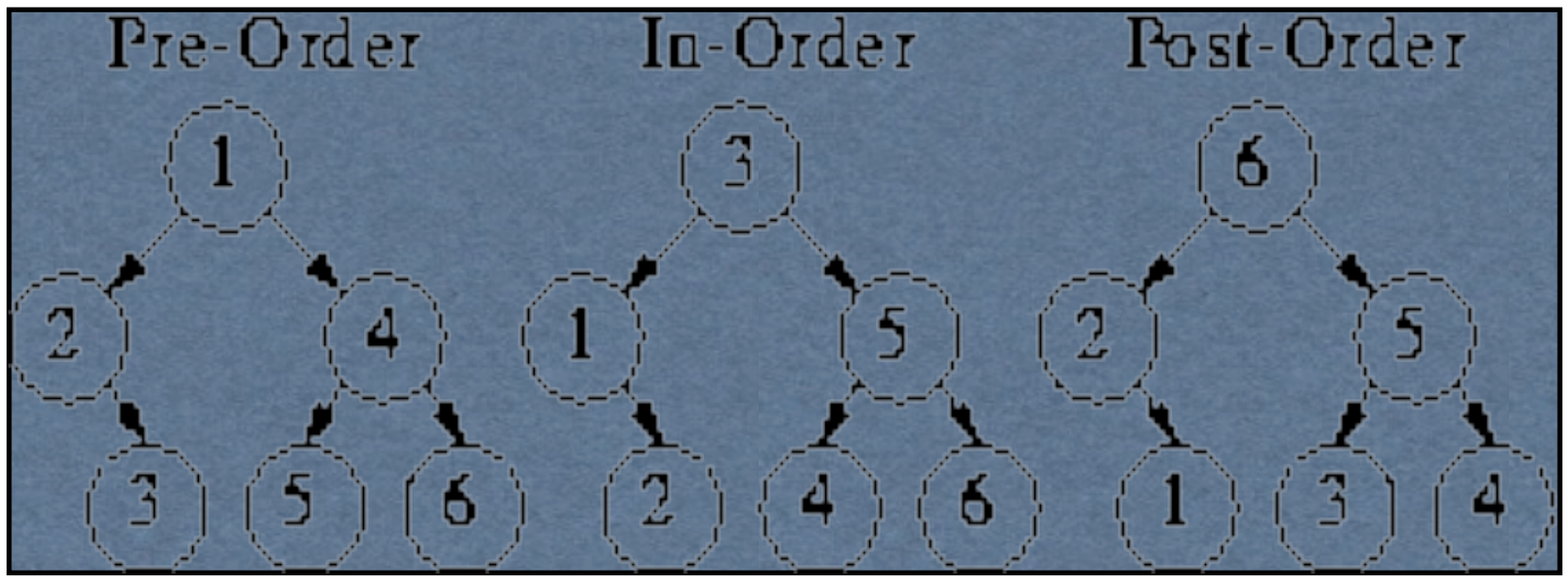
Traversal Orders

- The 3 main traversal orders for a binary tree are:
- VLR, LVR, LRV

- VLR is called the preorder.
- LVR is called the inorder.
- LRV is called the postorder.

Traversal Examples

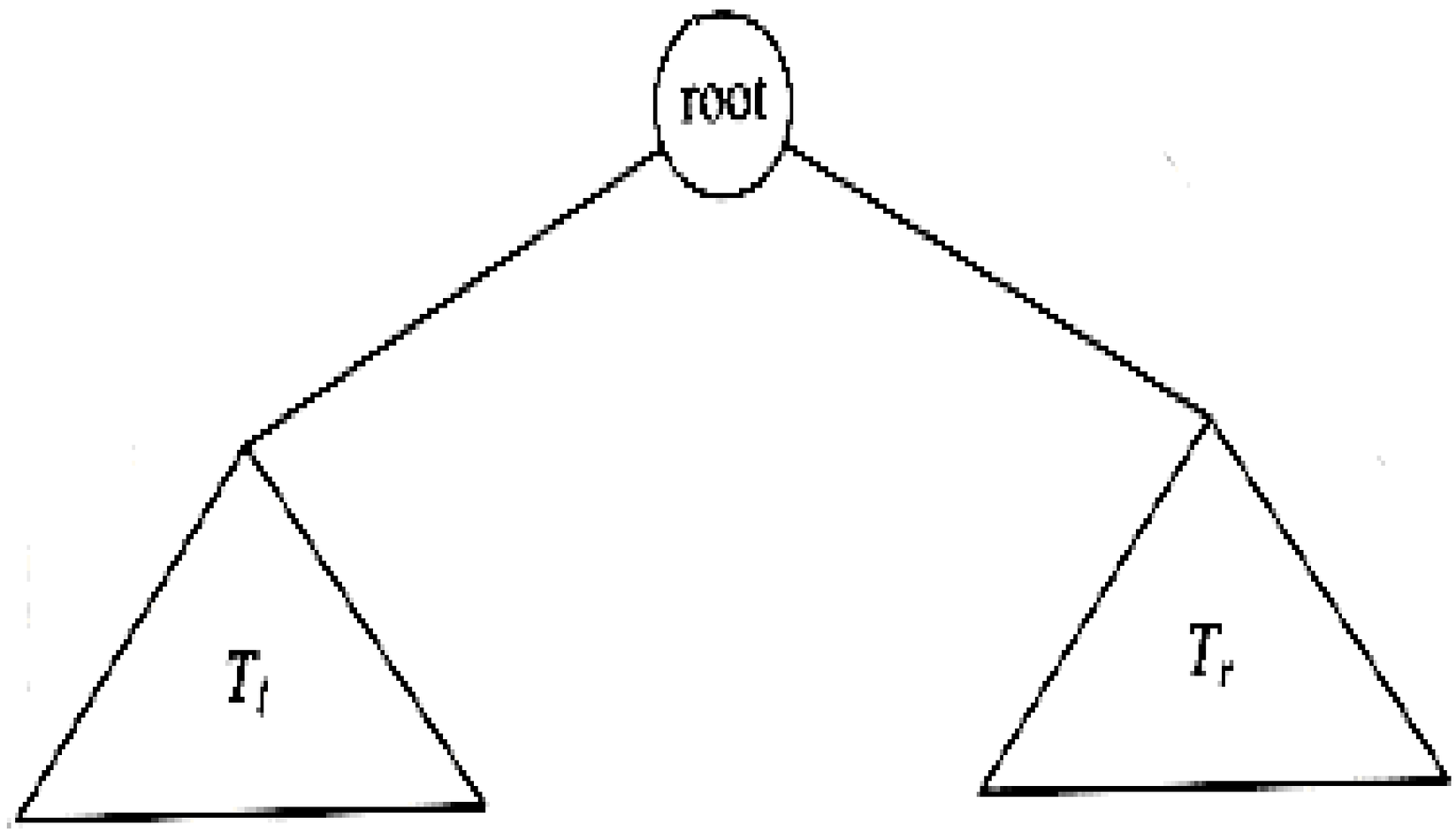




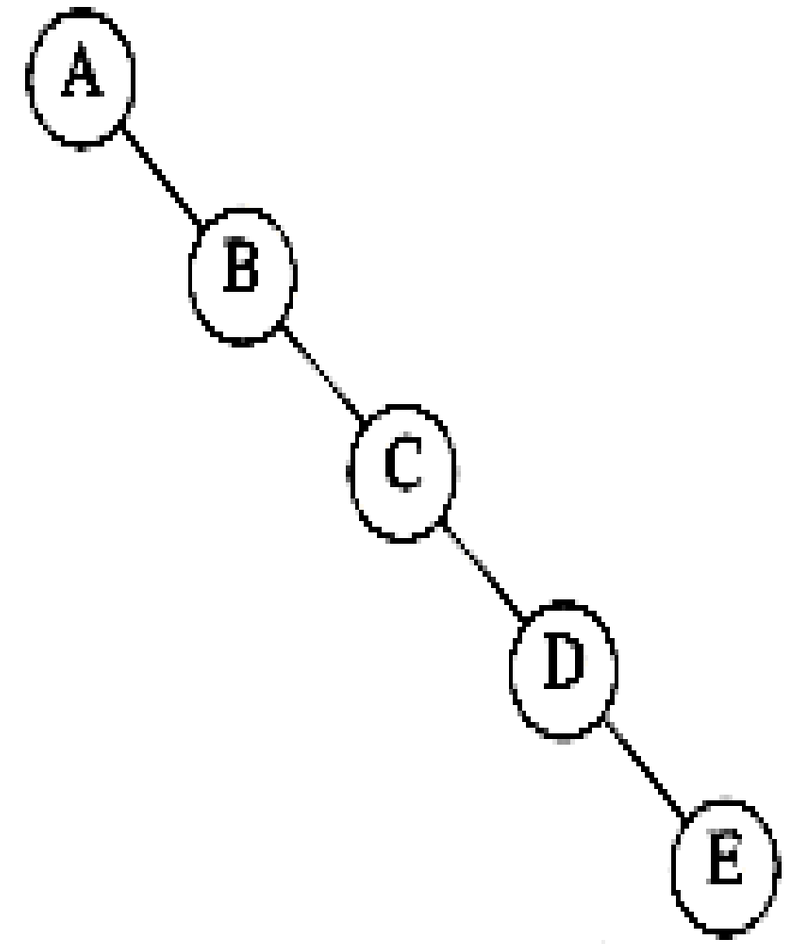
Binary Tree

- A binary tree is a tree in which no node can have more than two children.
- Property--The depth of an average binary tree is considerably smaller than n.
- The average depth is O(h/2), and that for a special type of binary tree, namely the binary search tree, the average value of the depth is O(log n).

Example-Generic Binary Tree



Degenerated Binary Tree



- The depth can be as large as n-1.

Implementation

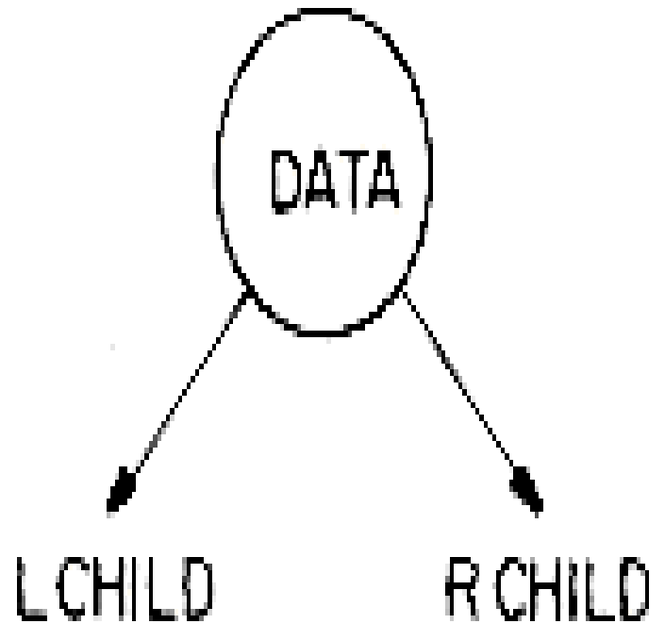
- Because a binary tree has at most two children, we can keep direct pointers to them.
- The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the key information plus two pointers (left and right) to other nodes.

Structure of a Tree Node

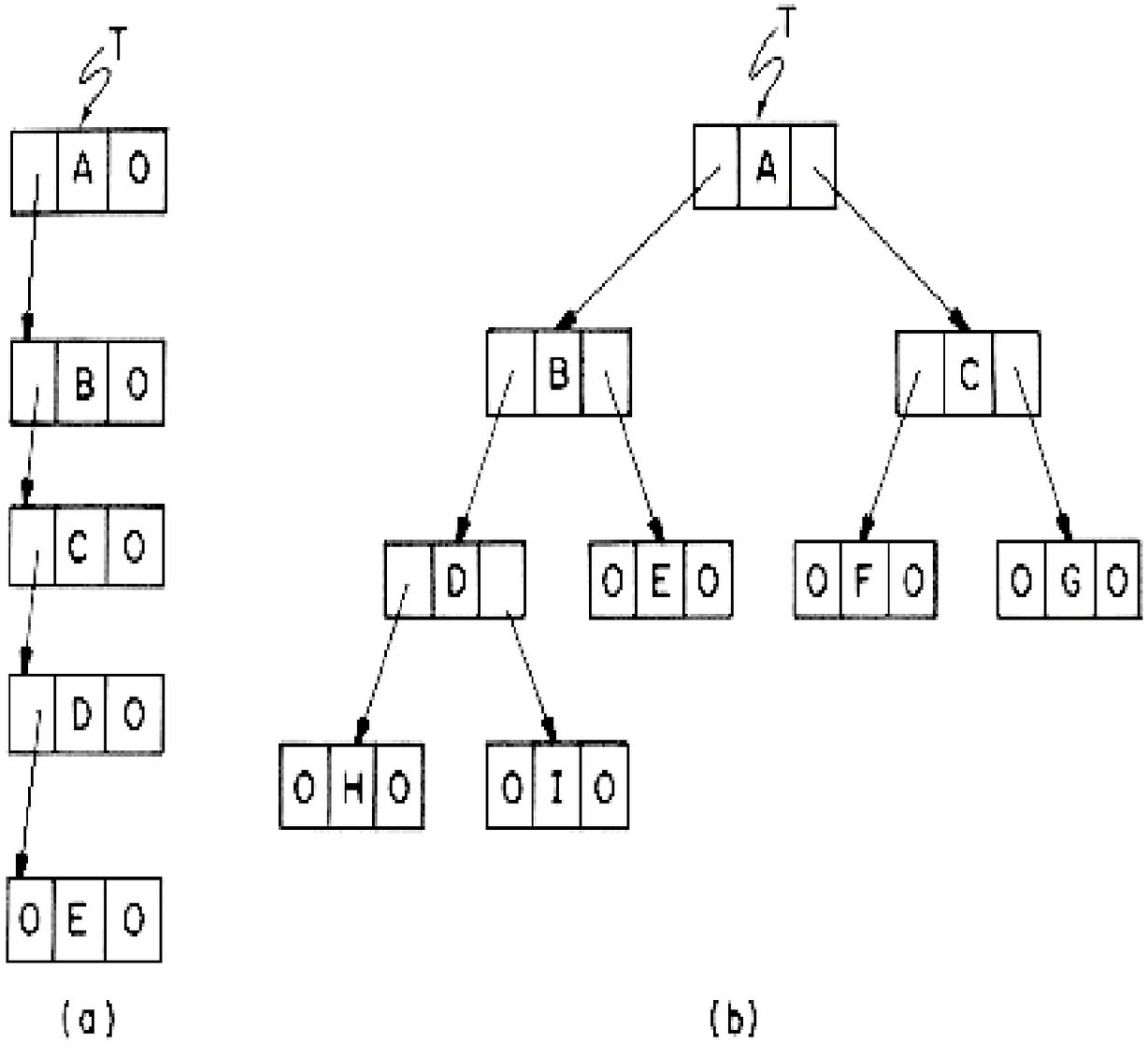
```
typedef struct tree_node *tree_ptr;

struct tree_node
{
        element_type element;
        tree_ptr left;
        tree_ptr right;
};
```


Tree Node Implementation

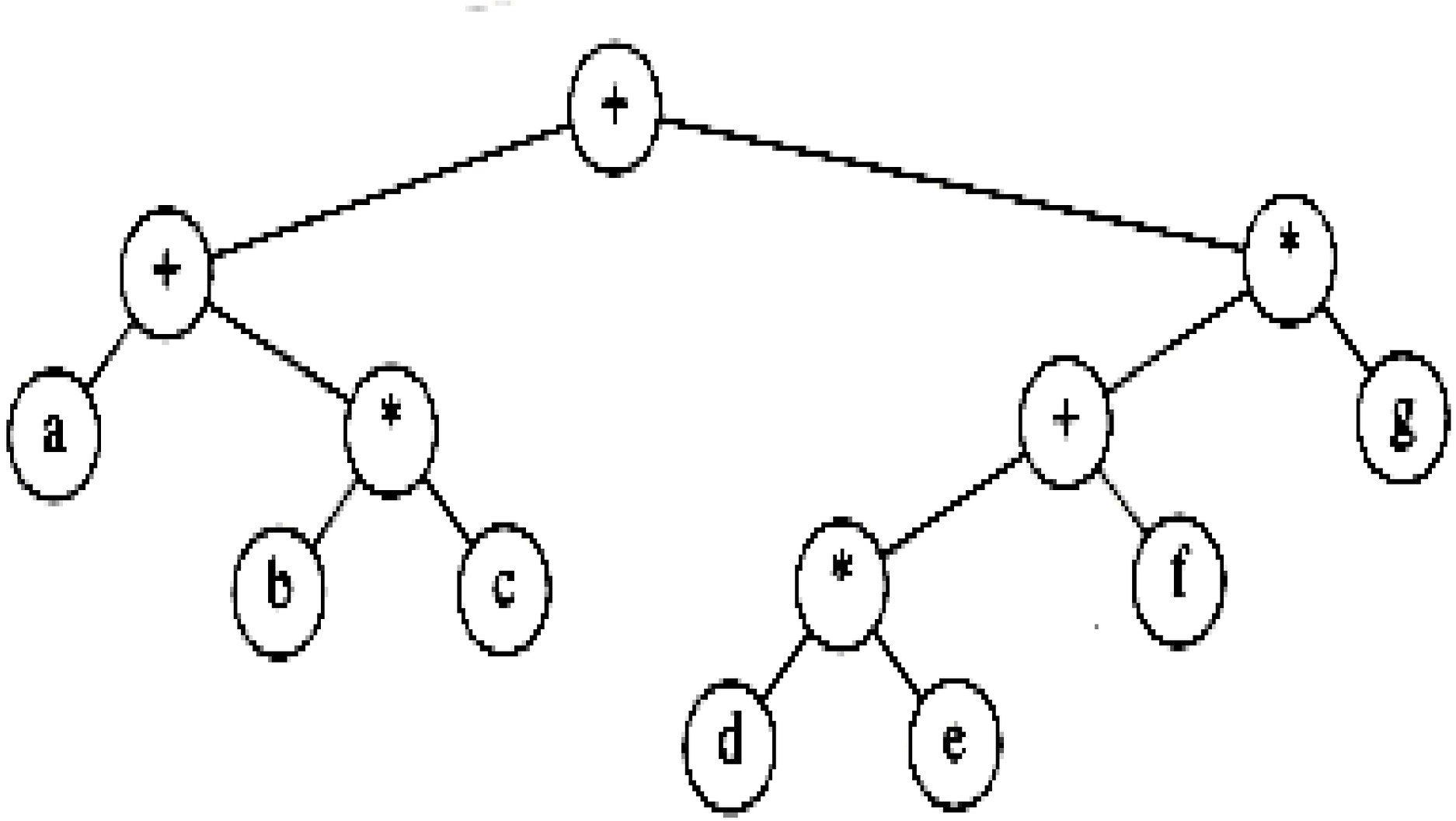


Example



Expression Trees

- Expression tree for (a + b * c) + ((d * e + f) * g)



Expression Tree

- The leaves of an expression tree are operands, such as constants or variable names, and the other nodes contain operators.
- This particular tree happens to be binary, because all of the operations are binary.
- It is possible for nodes to have more than two children.

Notes

- It is possible for a node to have only one child, e.g., unary minus operator.
- We can evaluate an expression tree, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.
- The left subtree evaluates to a + (b * c) and the right subtree evaluates to ((d *e) + f)*g.

Postorder Traversal

- An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator.
- If we apply this strategy to our tree above, the output is a b c * + d e * f + g * +.
- This is the postfix representation of the expression.

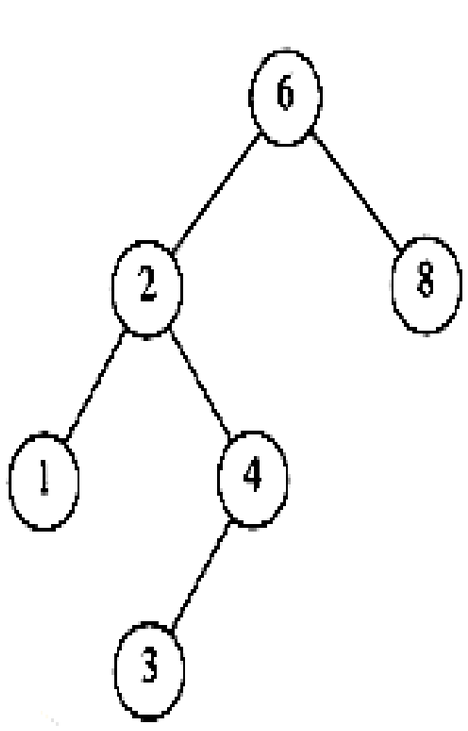
Preorder Traversal

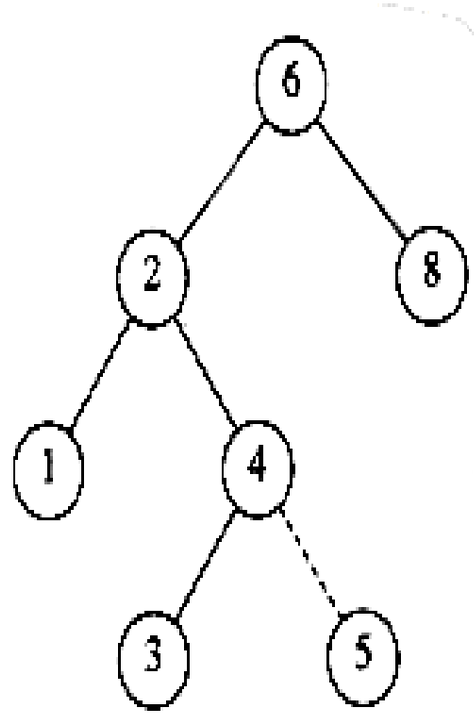
- A third traversal strategy is to print out the operator first and then recursively print out the left and right subtrees.
- The resulting expression is + + a * b c * + * d e f g.
- This is the less useful prefix notation and the traversal strategy is a preorder traversal.

Binary Search Trees

- An important application of binary trees is their use in searching.
- Let us assume that each node in the tree is assigned a unique integer key value.
- The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X,
- and the values of all the keys in the right subtree are larger than the key value in X.

Example





- Note that this implies that all the elements in the tree can be ordered in some consistent manner.

The tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with key 7 in the left subtree of a node with key 6 (which happens to be the root).

Typical Operations

- Make_null
- Find
- Find_min and find_max
- Insert
- Delete

Find

- This operation generally requires returning a pointer to the node in tree T that has key x, or NULL if there is no such node.
- The structure of the tree makes this simple.
- If T is empty, then we can just return NULL.
- Otherwise, if the key stored at T is x, we can return T.
- Otherwise, we make a recursive call on a subtree of T, either left or right, depending on the relationship of x to the key stored in T.

Find_min and Find_max

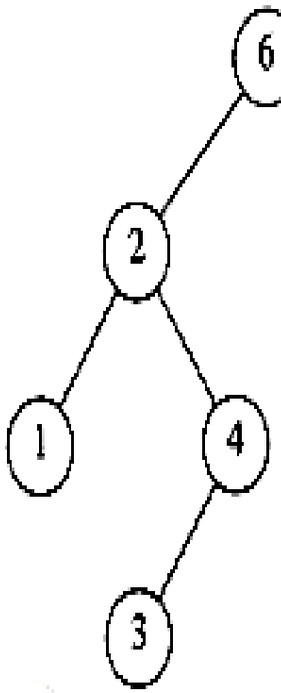
- These routines return the position of the smallest and largest elements in the tree, respectively.
- Although returning the exact values of these elements might seem more reasonable, this would be inconsistent with the find operation.

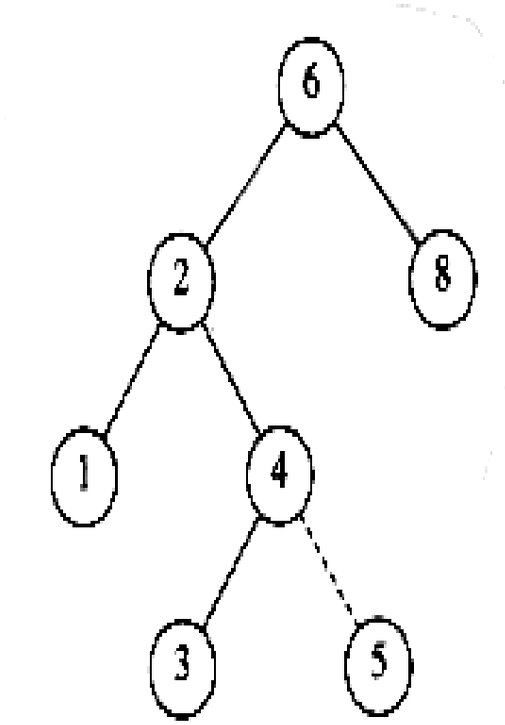
- To perform a find_min, start at the root and go left as long as there is a left child.
- The stopping point is the smallest element.
- The find_max routine is the same, except that branching is to the right child.

Insert

- To insert x into tree T, proceed down the tree as you would with a find.
- If x is found, do nothing (or "update" something).
- Otherwise, insert x at the last spot on the path traversed.
- Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence.

Insertion Example



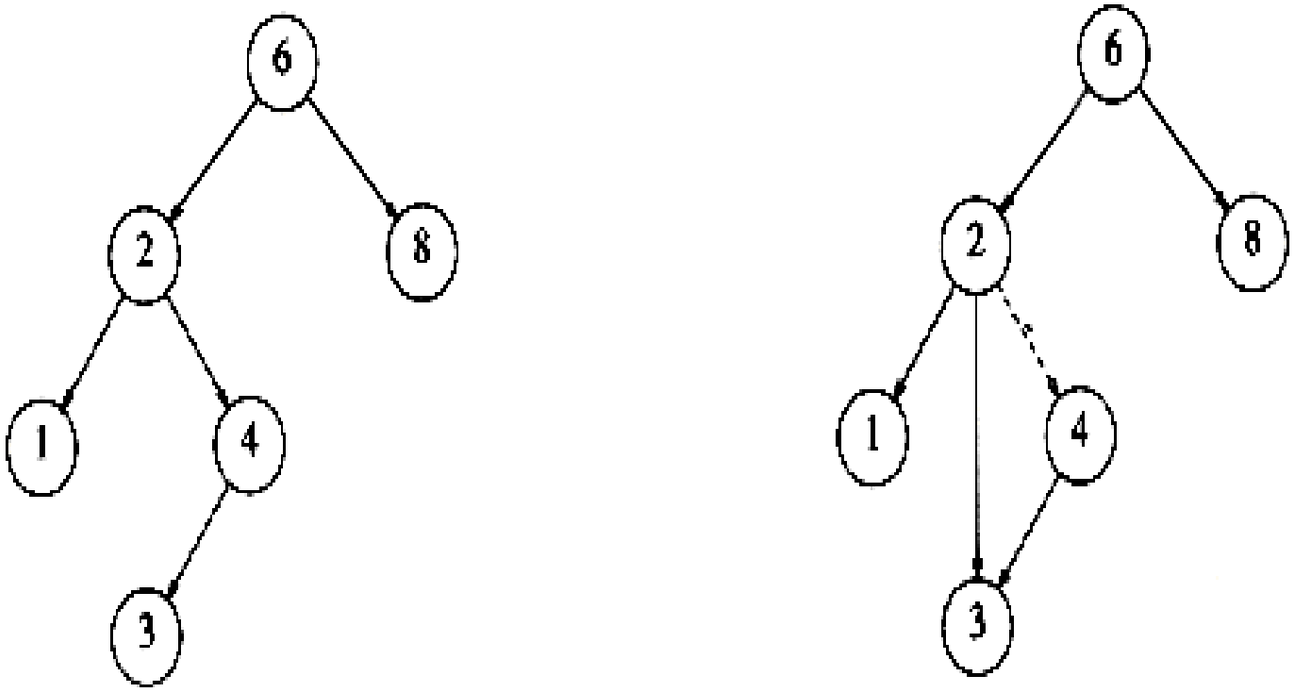


- To insert 5, we traverse the tree as though a find were occurring.
- At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Deletion

- Deletion is a bit tricky since we need to consider several possibilities.
- If the node is a leaf, it can be deleted immediately.
- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity).

Deletions Example

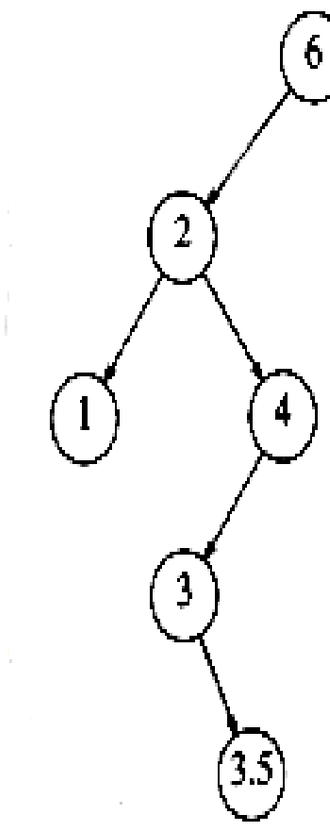


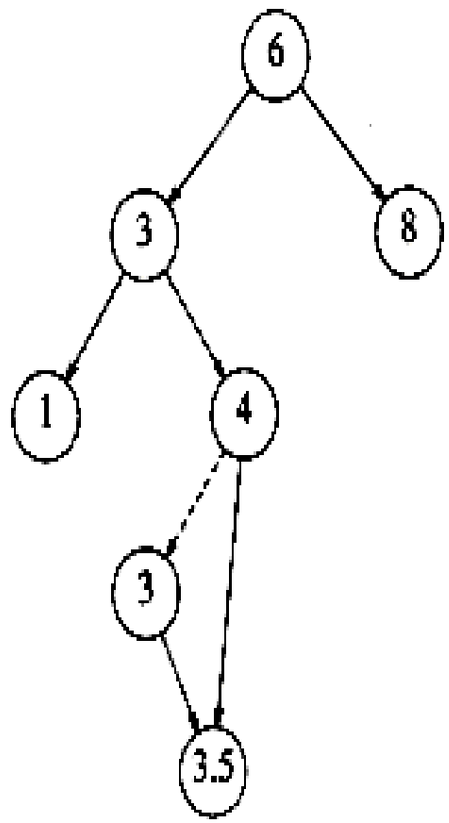
- We are trying to delete node 4.
- Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved.

Deletion

- The complicated case deals with a node with two children.
- The general strategy is to replace the key of this node with the smallest key of the right subtree (which is easily found) and recursively delete that node (which is now empty).
- Because the smallest node in the right subtree cannot have a left child, the second delete is an easy one.

Example





- This shows an initial tree and the result of a deletion of node 2.
- The node to be deleted is the left child of the root; the key value is 2.
- It is replaced with the smallest key in its right subtree (3), and then that node is deleted as before.

Average-Case Analysis

- Intuitively, all operations, except make_null, should take O(log n) time.
- Indeed, the running time of all the operations, except make_null, is O(d), where d is the depth of the node containing the accessed key.
- We prove in this section that the average depth over all nodes in a tree is O(log n) on the assumption that all trees are equally likely.

Analysis

- Let D(n) be the internal path length for some tree T of n nodes.
- Internal path length is the sum of the depths of all nodes in a tree and D(1) = 0.
- An n-node tree consists of an i-node left subtree and an (n - i - 1)-node right subtree, plus a root at depth zero for $0 \leq i < n$.
- D(i) is the internal path length of the left subtree with respect to its root.
- We obtain D(N) = D(i) + D(N-i-1) + N - 1

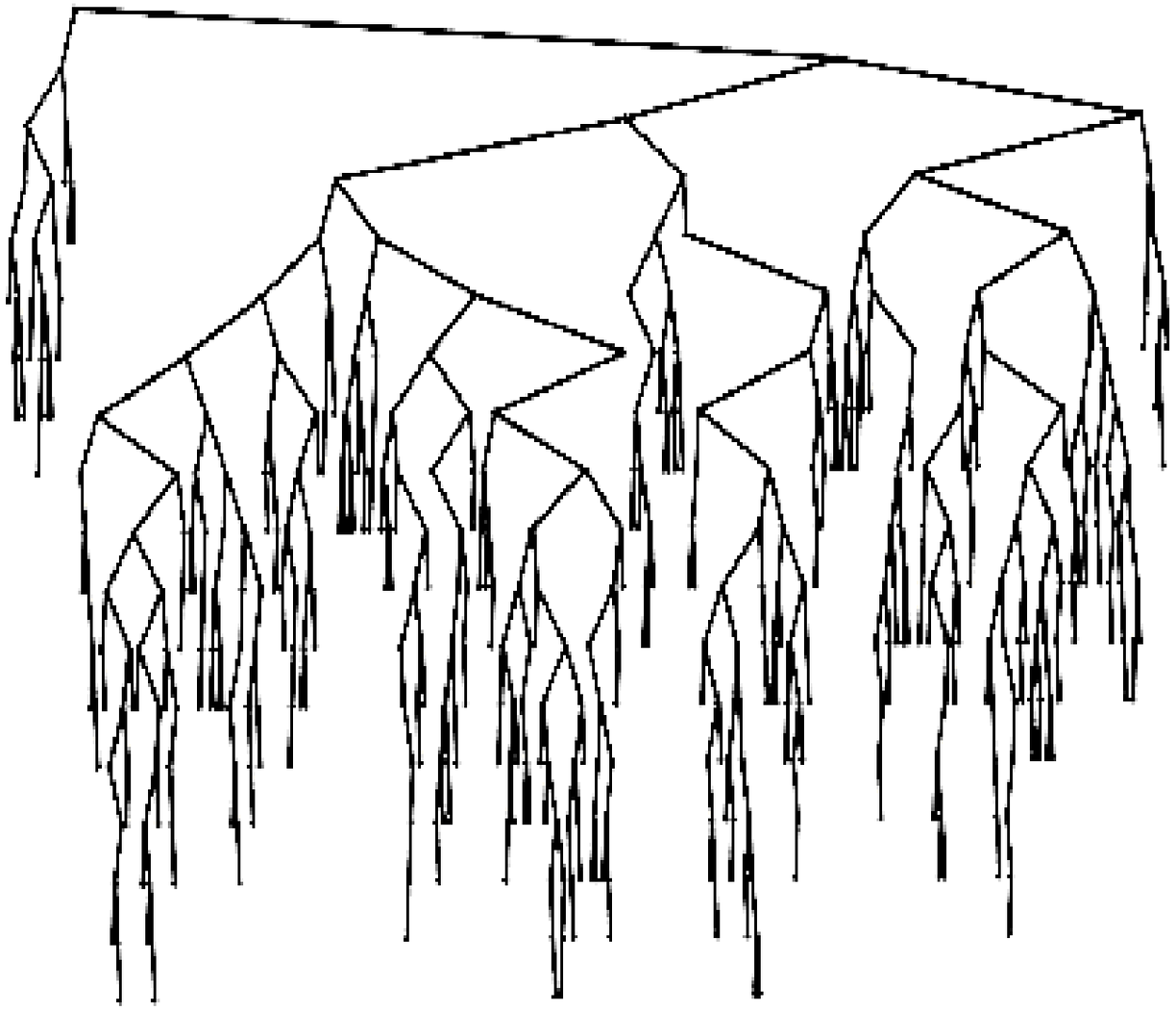
N-1 nodes that are one level deeper from the root.

Analysis

- If all subtree sizes are equally likely, which is true for binary search trees (since the subtree size depends only on the relative rank of the first element inserted into the tree), but not binary trees, then the average value of both D(i) and D(n - i -1) is
- $$\frac{1}{n}\sum_{j=0}^{n-1} D(j)$$
- This yields

$$D(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} D(j)\right] + n - 1$$

Example

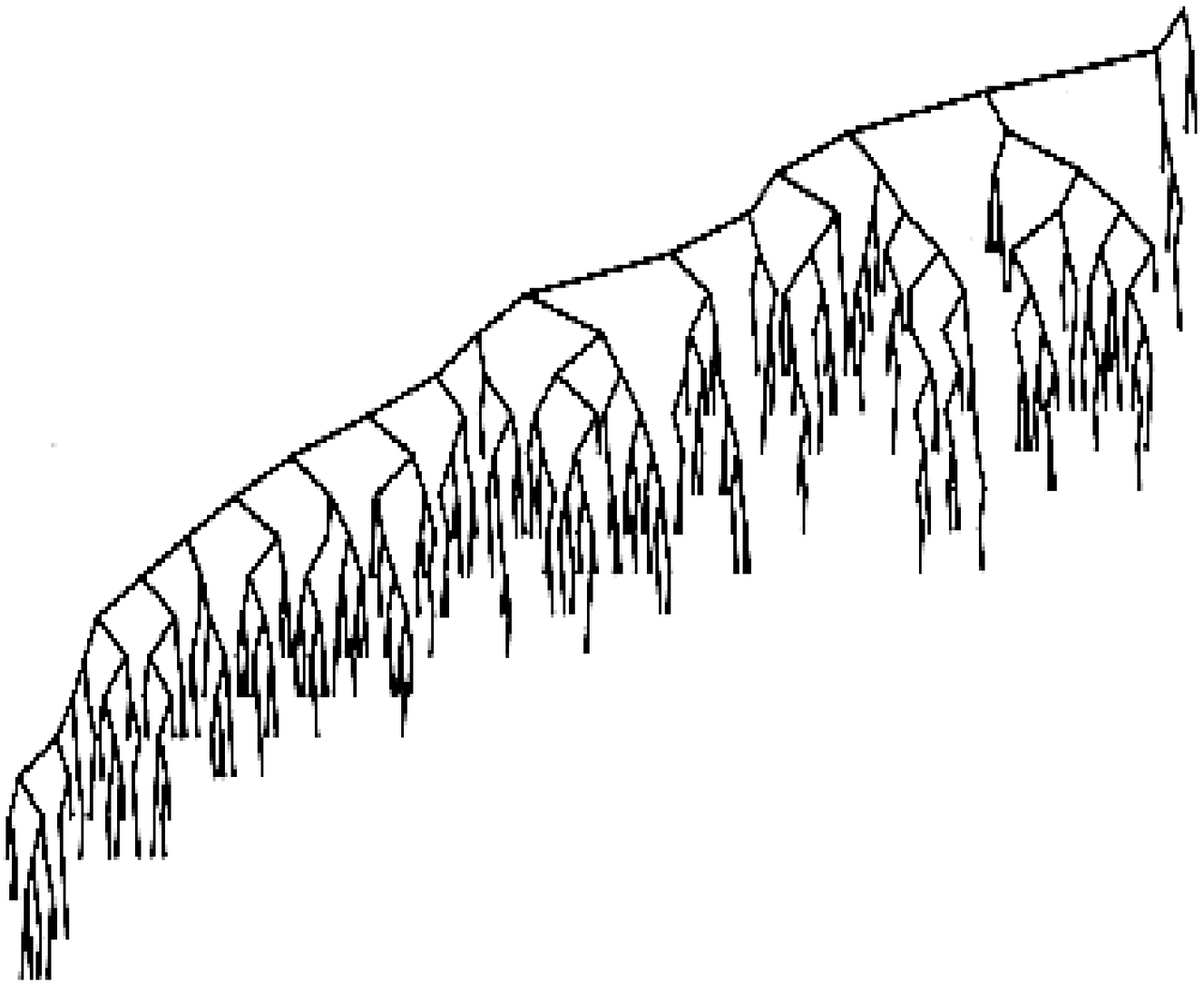


This recurrence gets an average value of D(n) = O(n log n).

Thus, the expected depth of any node is O(log n).

As an example, the randomly generated 500-node tree shown has nodes at expected depth 9.98.

Unbalanced Binary Tree



- Our deletion routine favors the left subtree so after many insertion and deletion operations we may end up with an unbalanced binary tree as shown here.

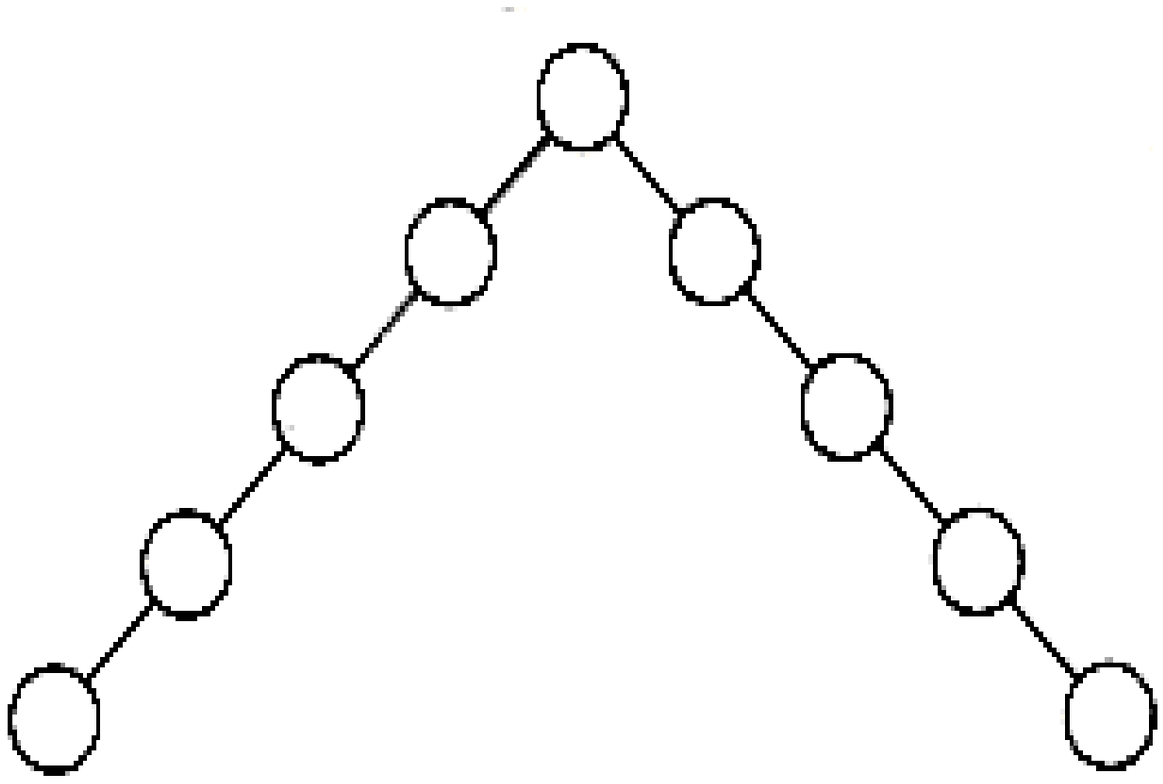
Observation

- Balancing of a binary search tree will be important to ensure that the tree does not degenerate into an unbalanced tree.
- This is a more difficult problem
  - since balancing a tree often requires the structure of the tree be changed.

AVL Trees

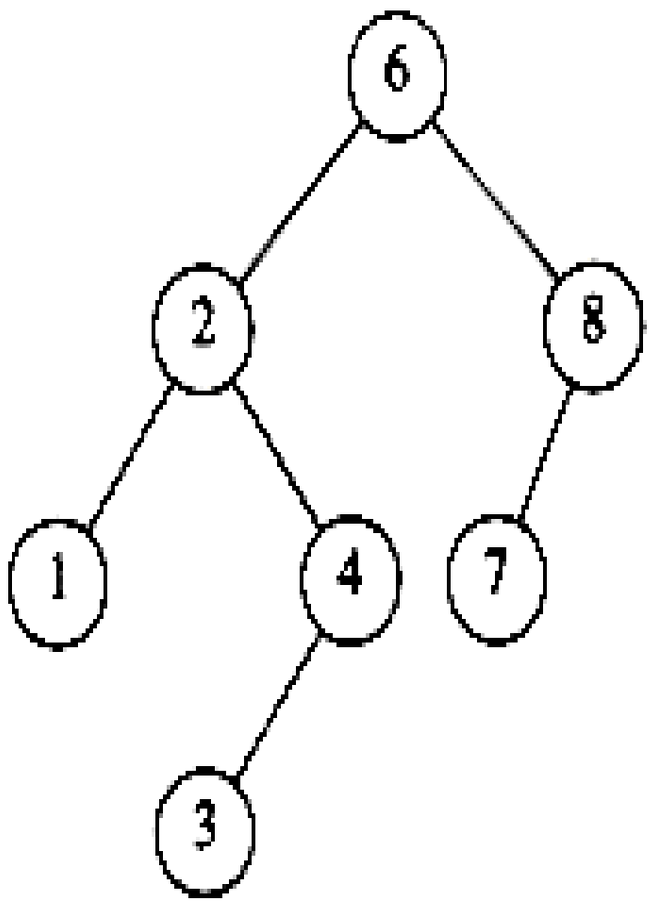
- An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition.
- An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1.
- With an AVL tree, all the tree operations can be performed in O(log n) time, except insertion.

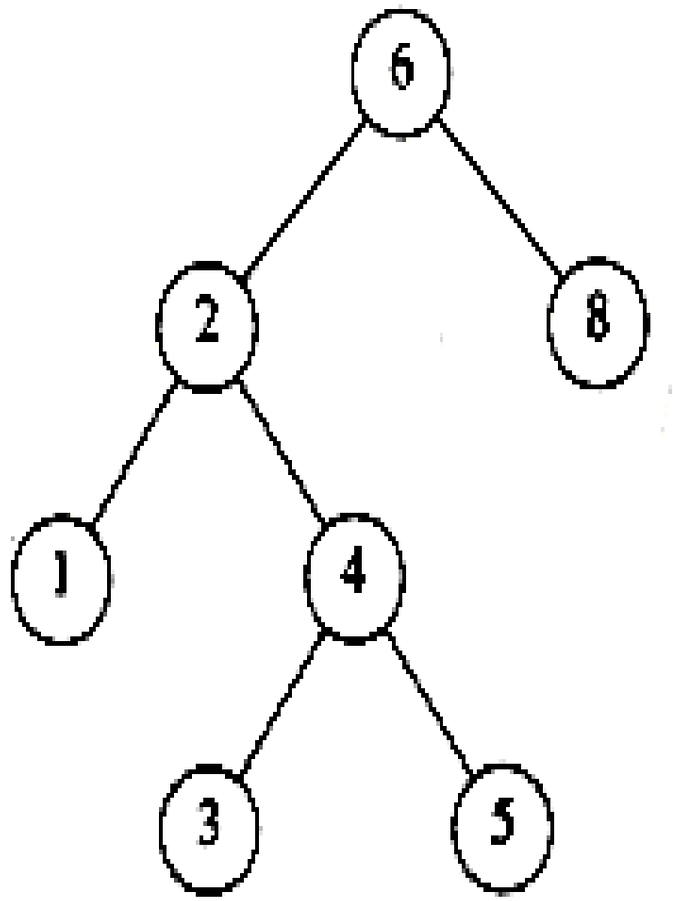
Example



is not enough.

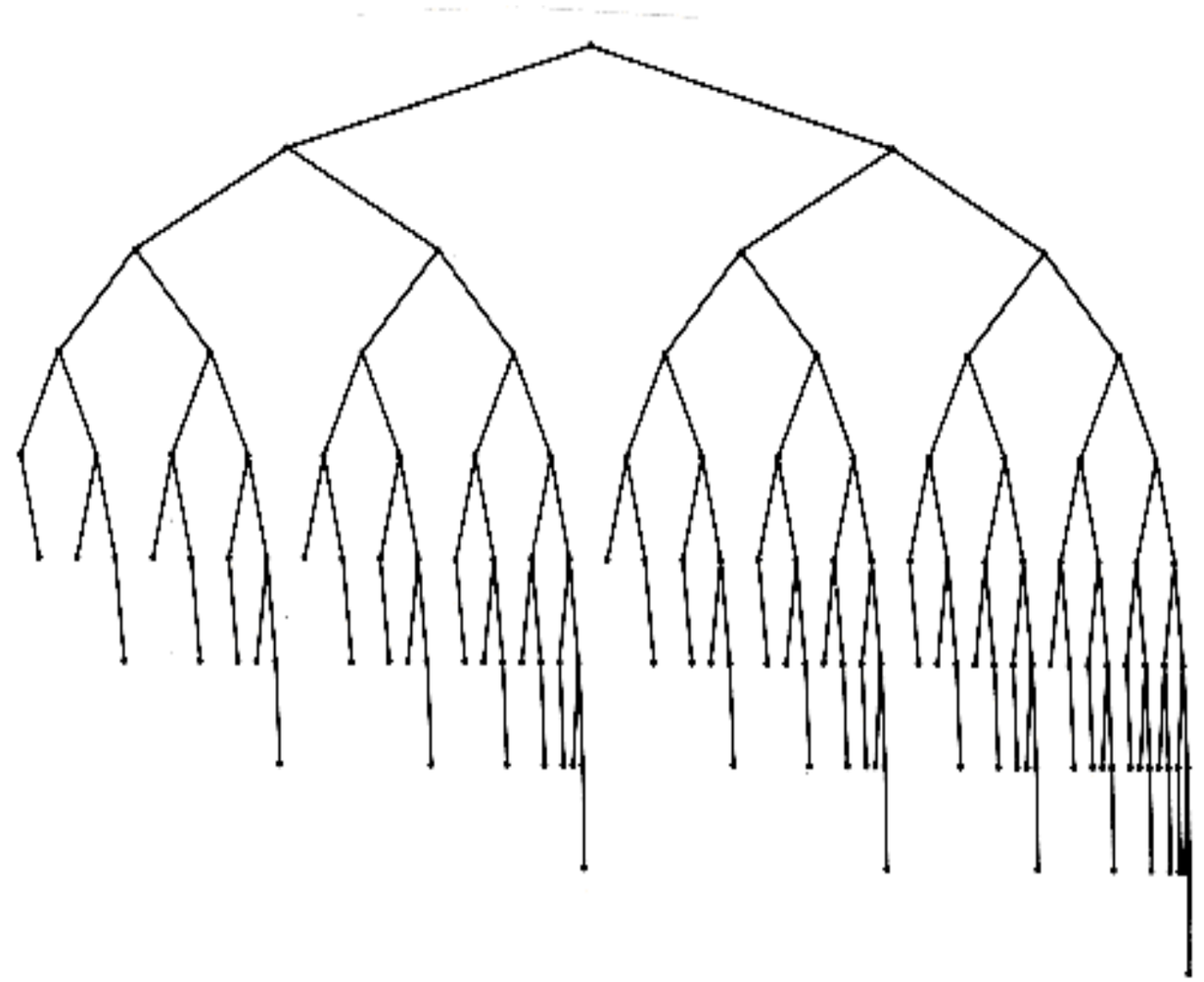
Example





Example

- Smallest AVL tree of height 9.
- Notice that the construction of the smallest AVL tree of height n is to use two smallest AVL sub-trees that are of n-1 and n-2 height.



Height of AVL Tree

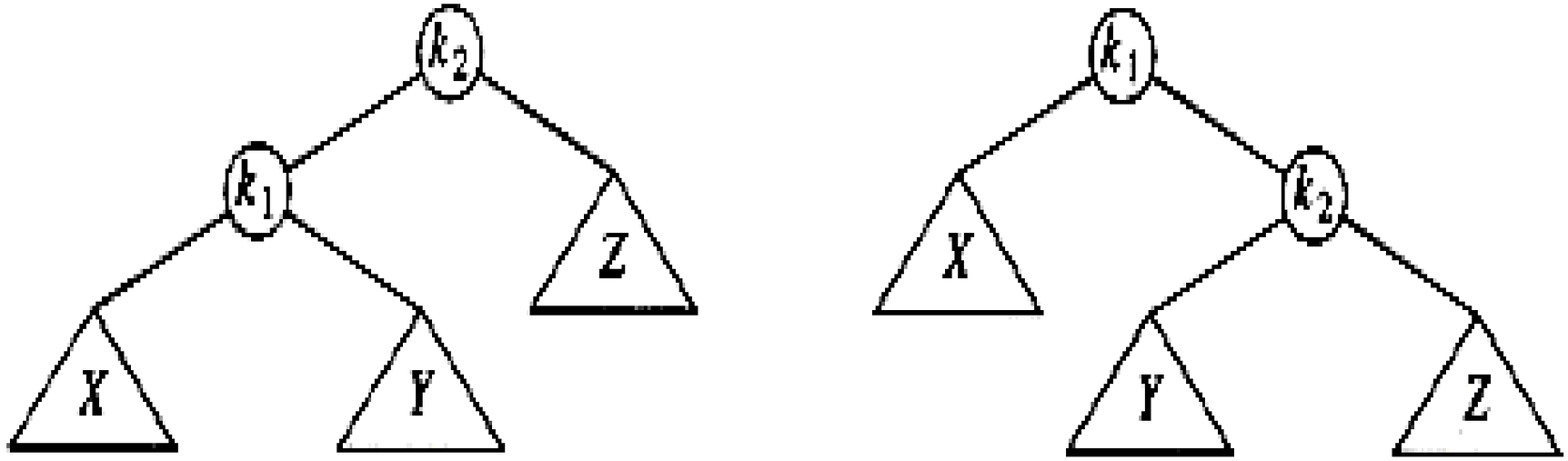
- The height of an empty tree is defined to be -1
- Height information is kept for each node (in the node structure).
- The height of an AVL tree is at most roughly 1.44 log(n + 2) - .328, but in practice it is about log(n + 1) + 0.25 (although this has not been proven).

Observations

- All the tree operations can be performed in O(log n) time, except possibly insertion.
- Insertion and deletion operations need to update the balancing information.
- It is sometimes difficult since that inserting a node could violate the AVL tree property.
- If this is the case, then the property has to be restored before the insertion step is considered over.
- The property can be achieved through a rotation.

Single Rotation

- A rotation involves only a few pointer changes, and changes the structure of the tree while preserving the search tree property.

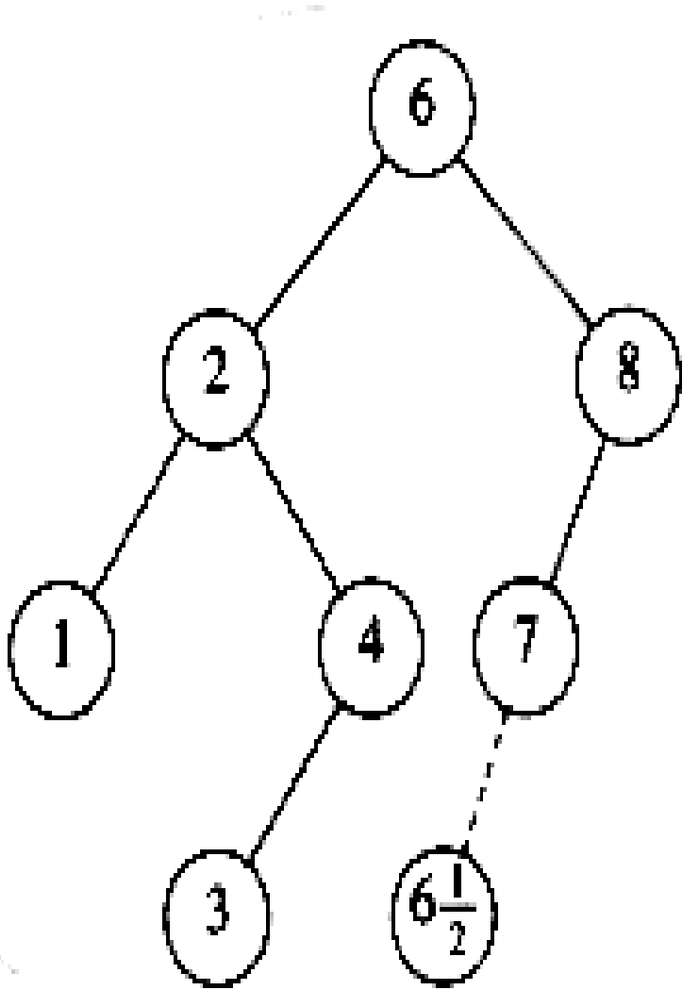


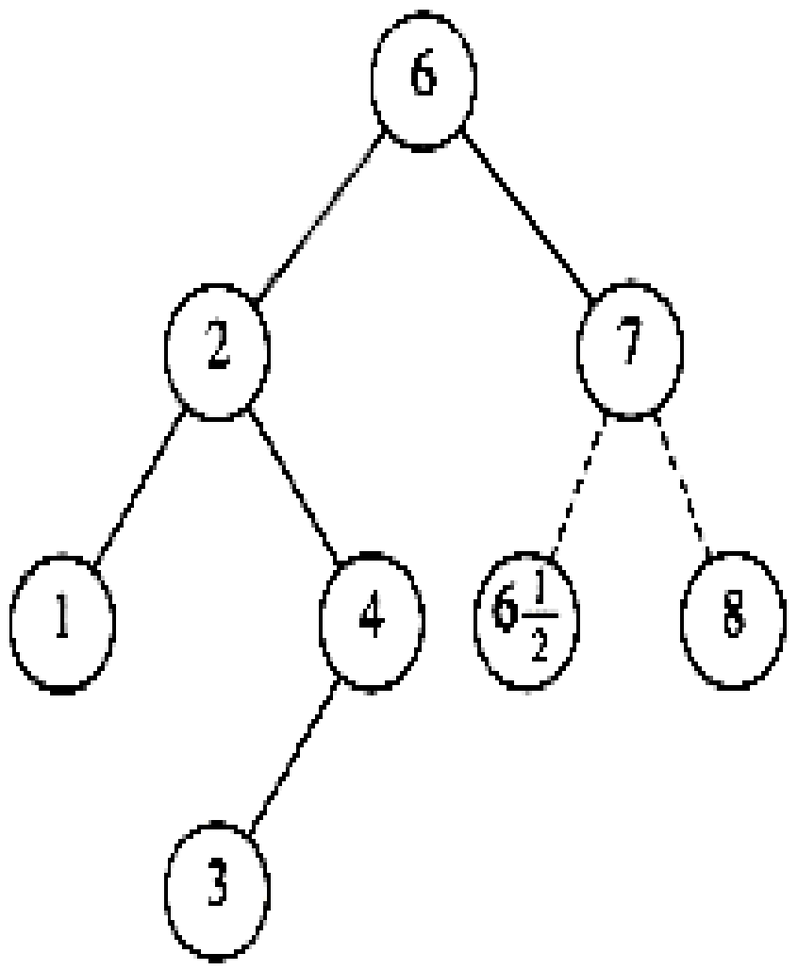
Notes

- The rotation does not have to be done at the root of a tree; it can be done at any node in the tree.
- This works from the bottom and upwards.

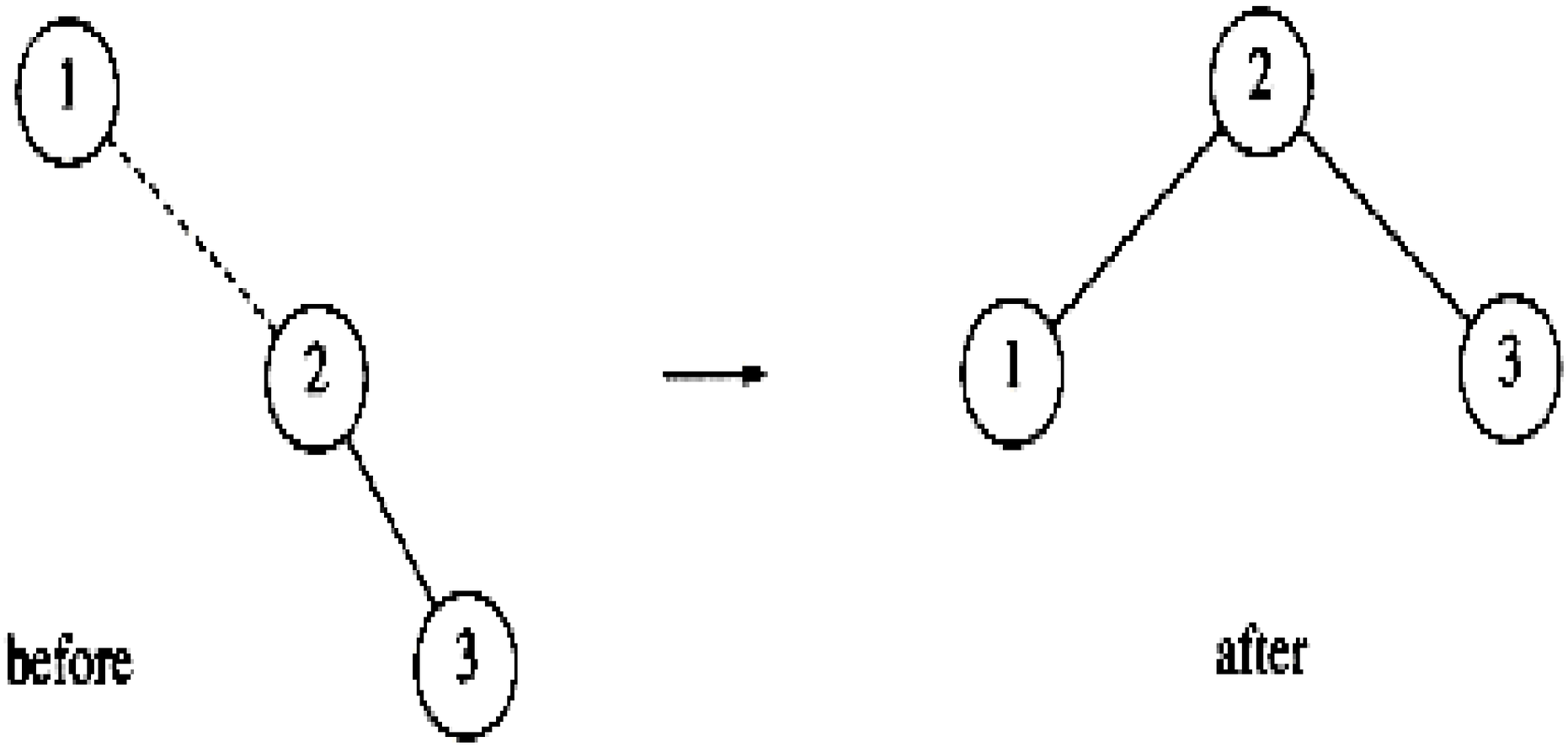
Example

- AVL property destroyed by insertion of 6 1/2, then fixed by a rotation

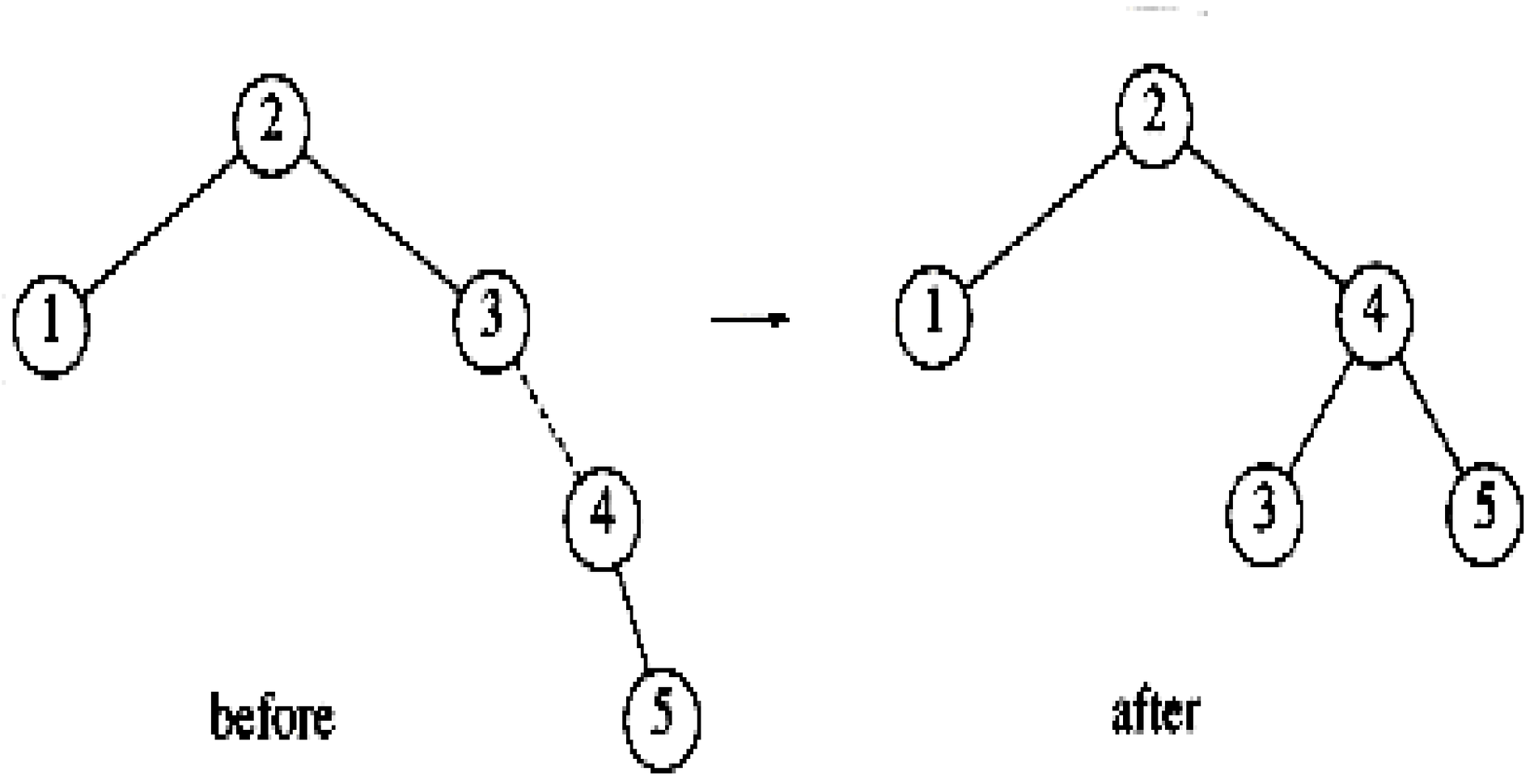




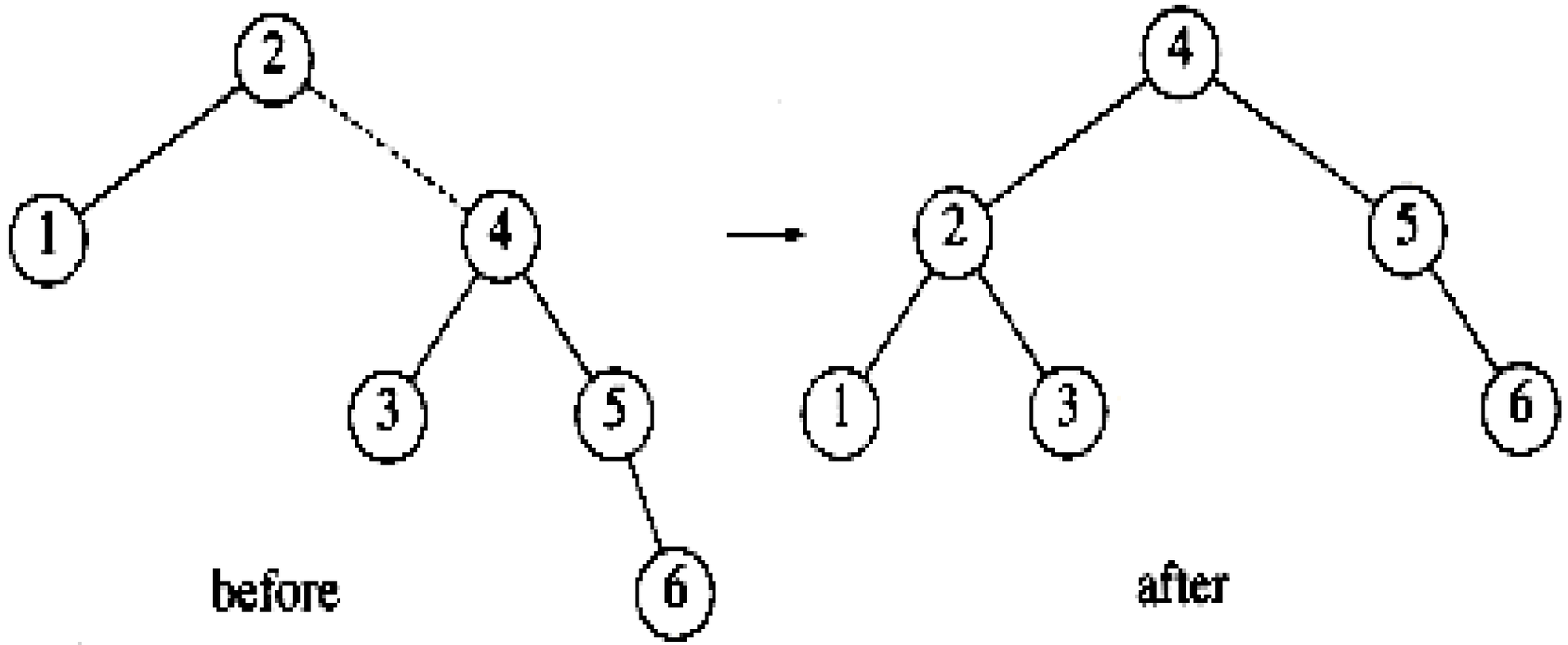
Sequential Insertion Example



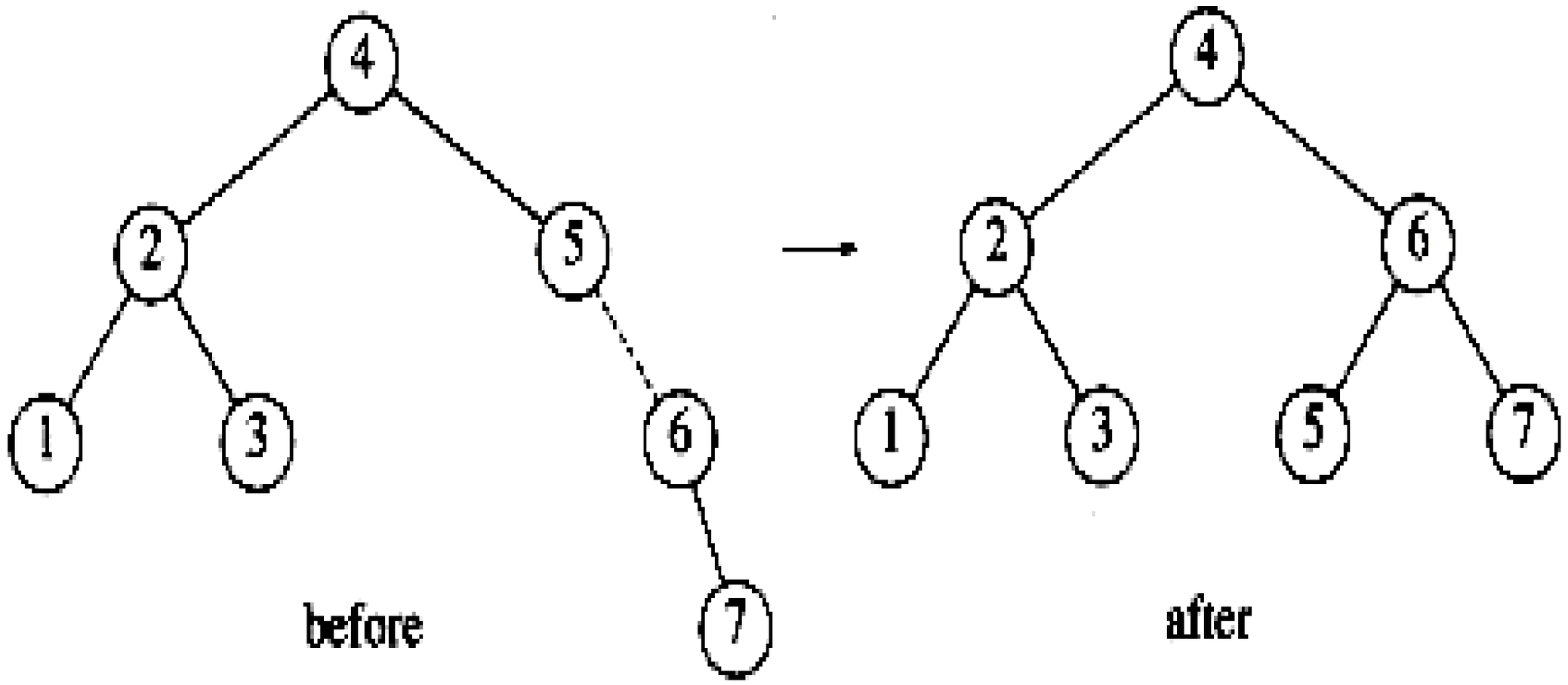
Example



Example



Example

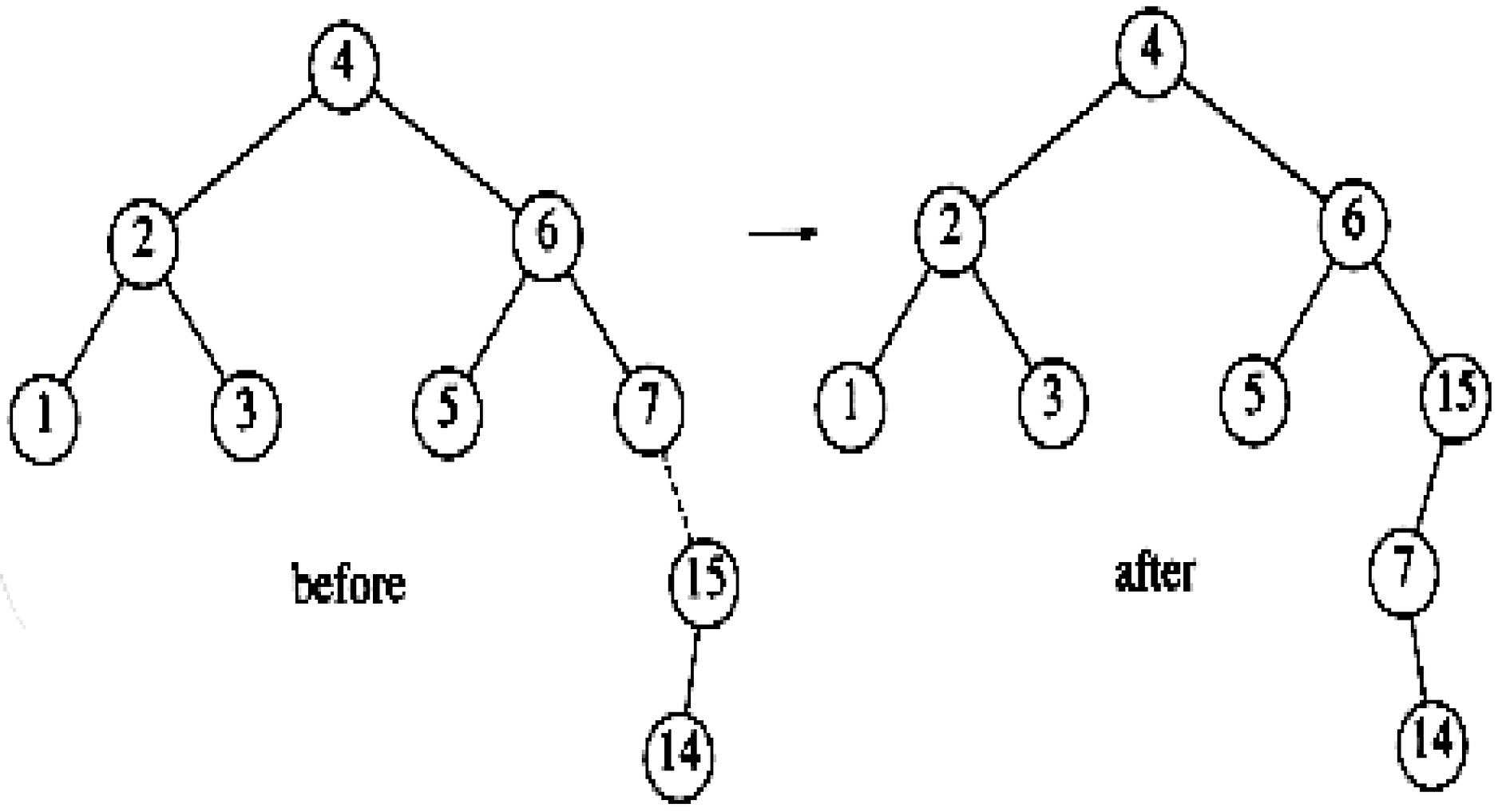


Double Rotation

- There is a case where the rotation does not fix the tree.

- Suppose we insert keys 8 through 15 in reverse order.
- Inserting 15 is easy, since it does not destroy the balance property, but inserting 14 causes a height imbalance at node 7.

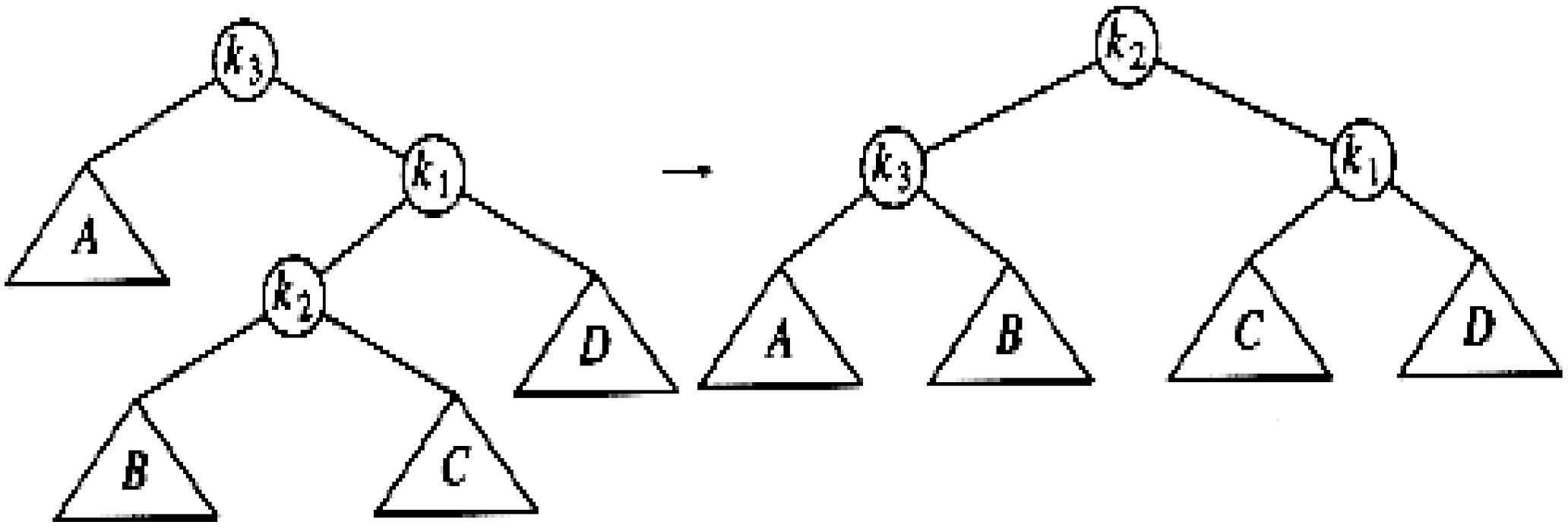
Example



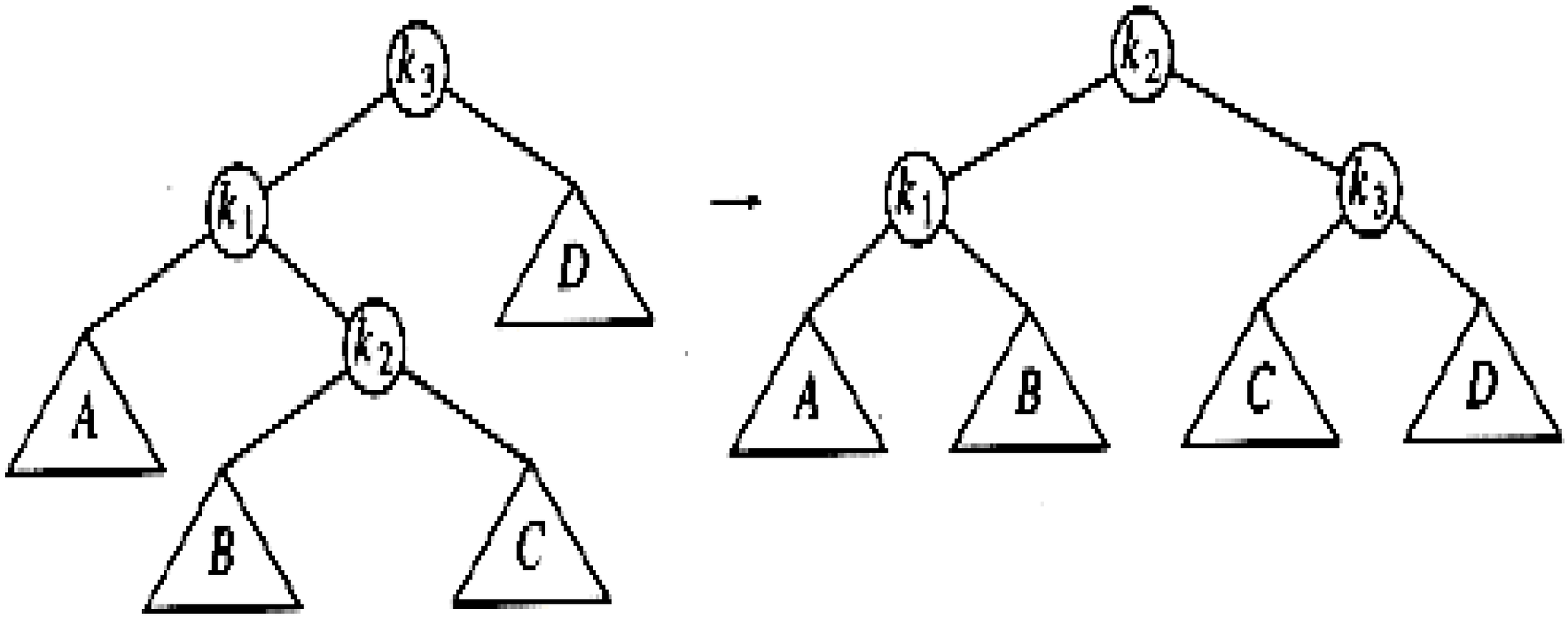
Double Rotation

- The problem is that the height imbalance was caused by a node inserted into the tree containing the middle elements at the same time as the other trees had identical heights.
- The case is easy to check for, and the solution is called a double rotation, which is similar to a single rotation but involves four subtrees instead of three.

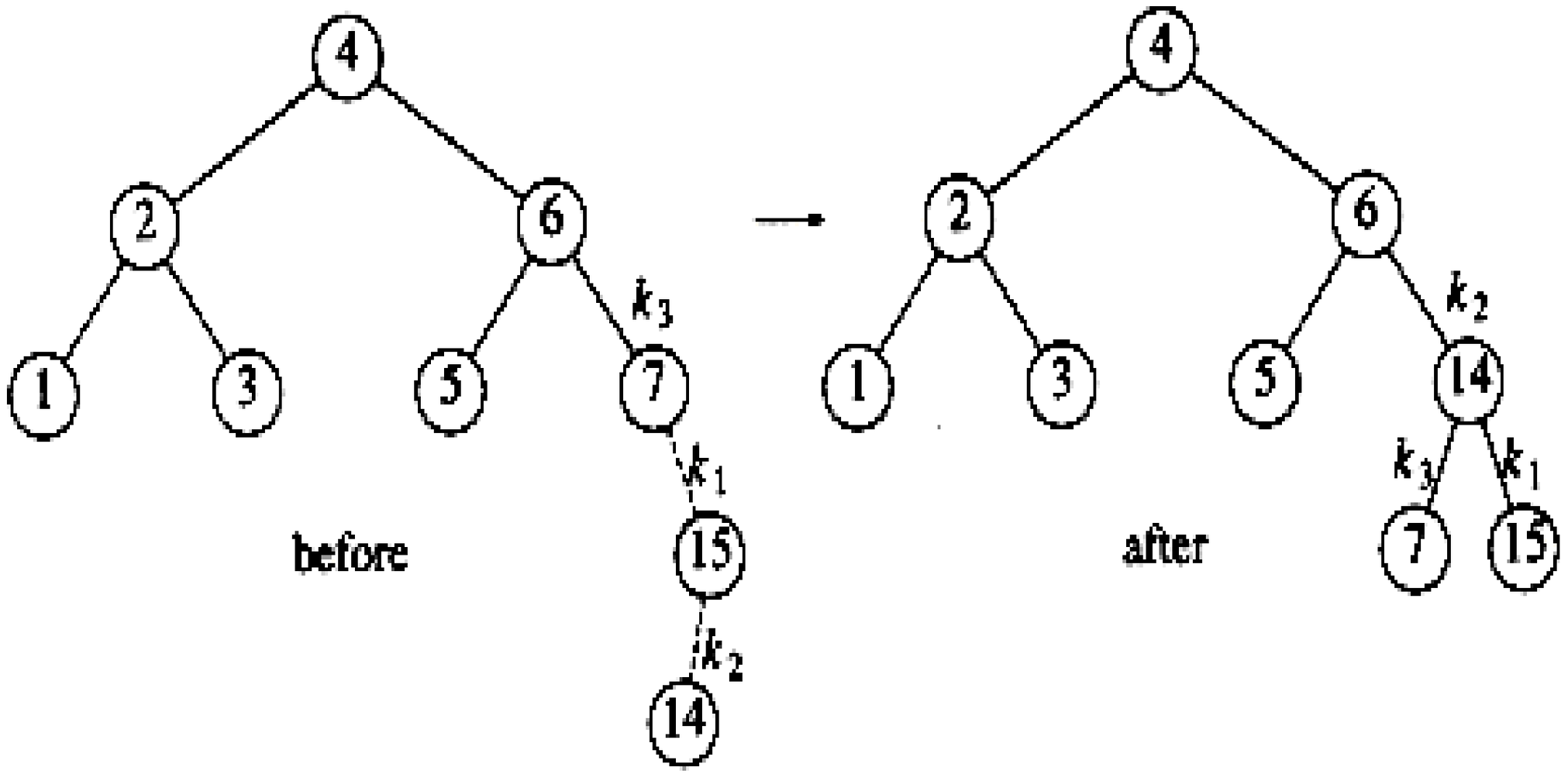
Example- (Right-left) double rotation



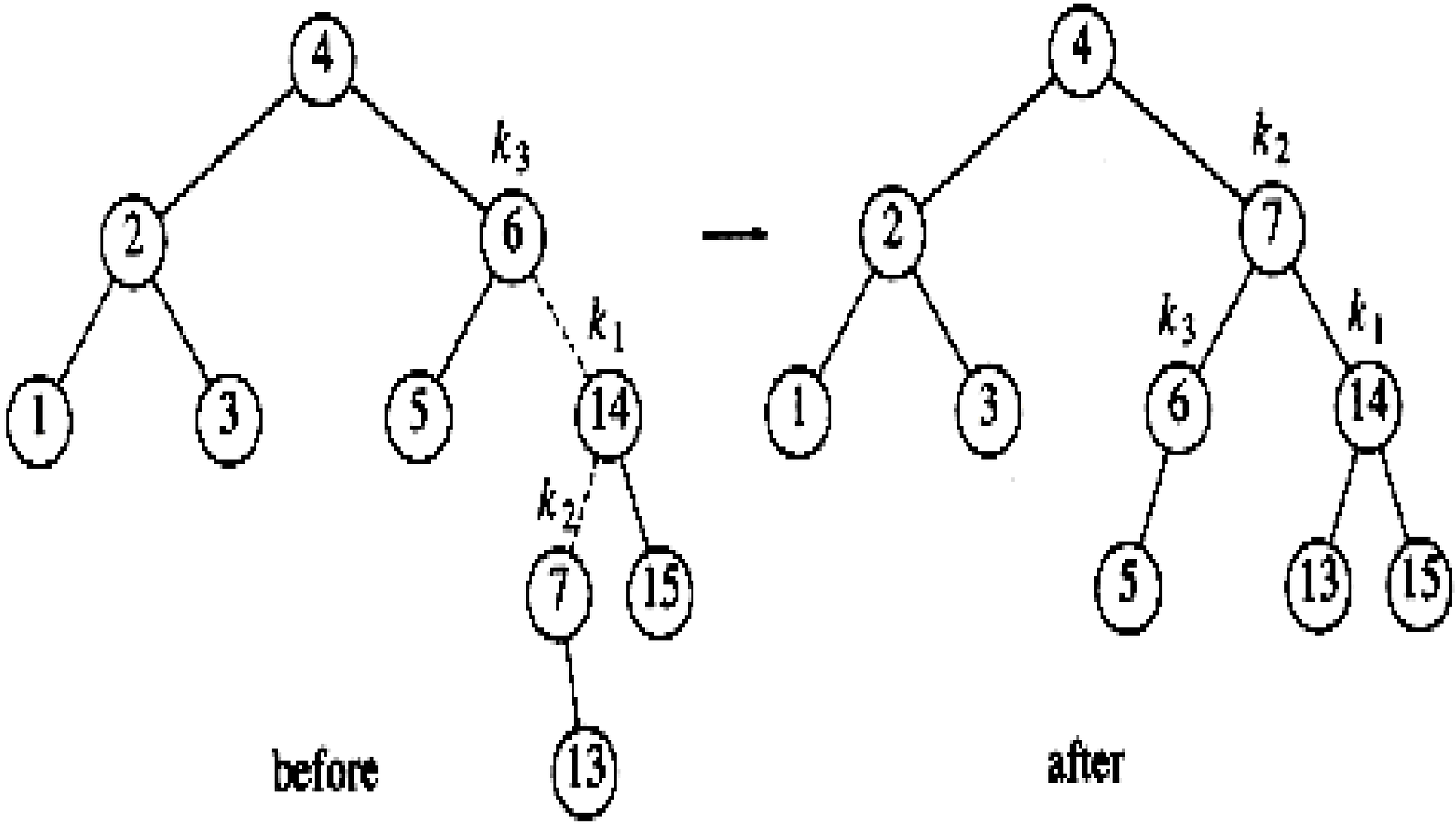
Example-(Left-right) double rotation



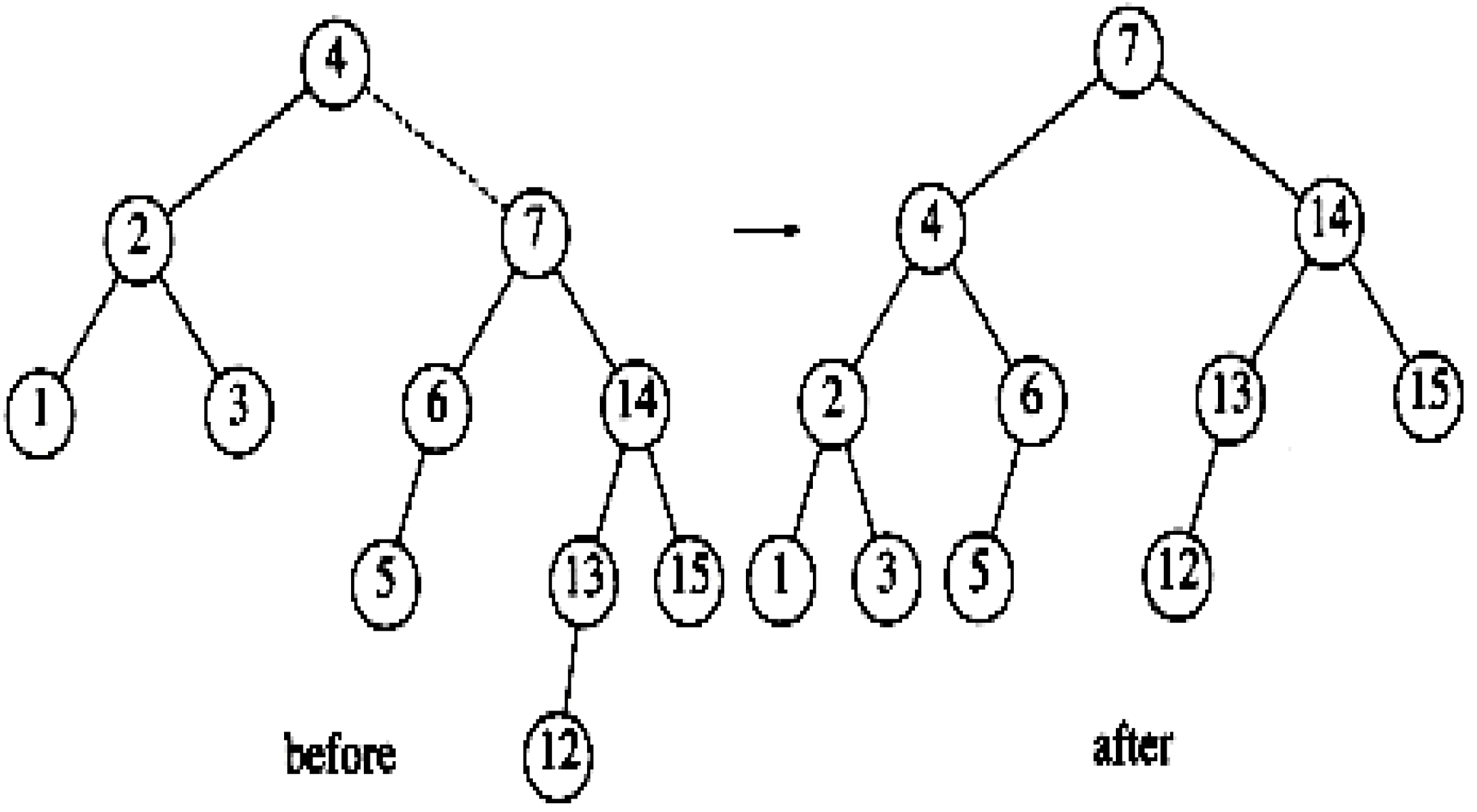
Example



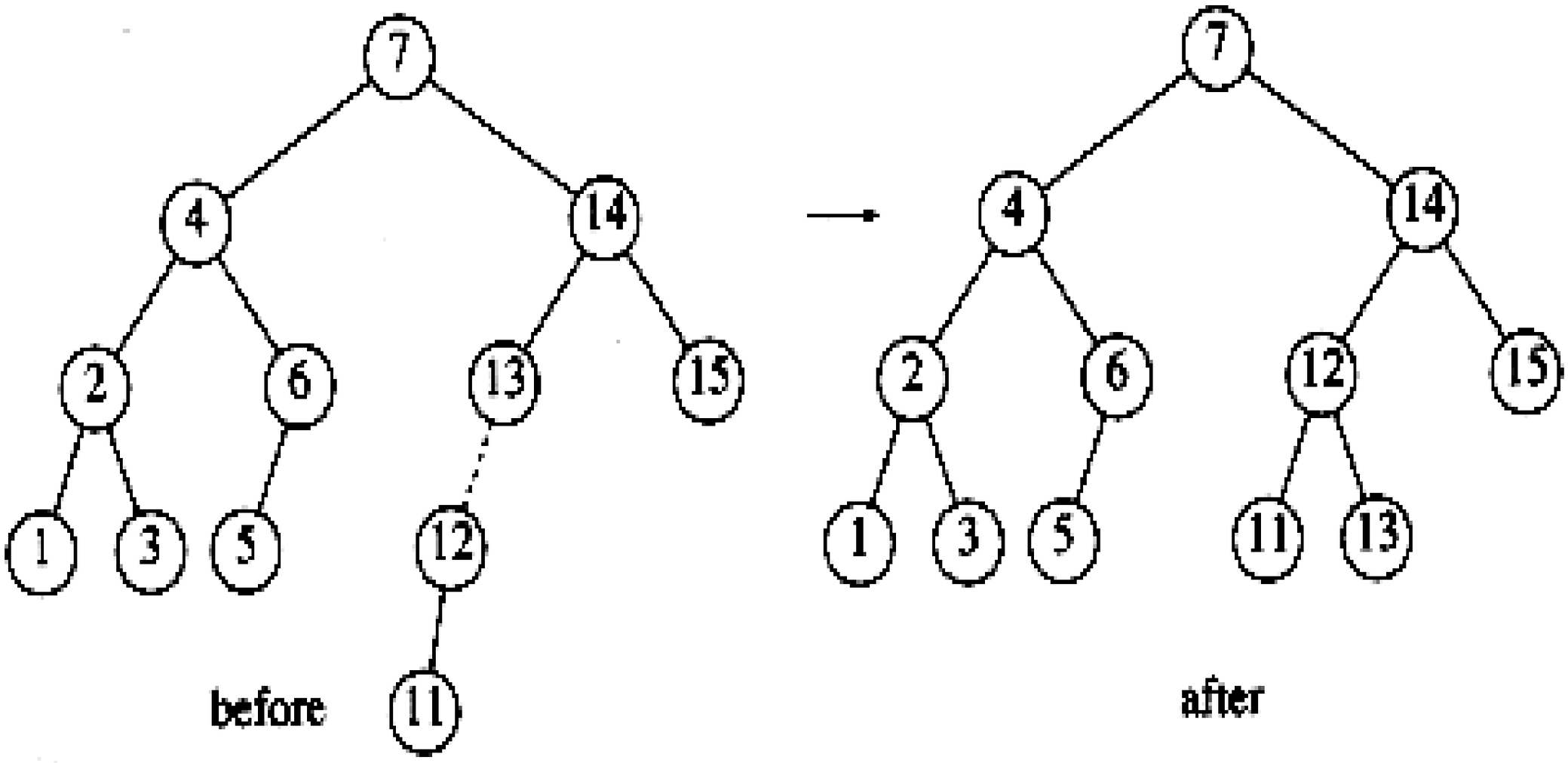
Example-Insert 13, Double Rotation



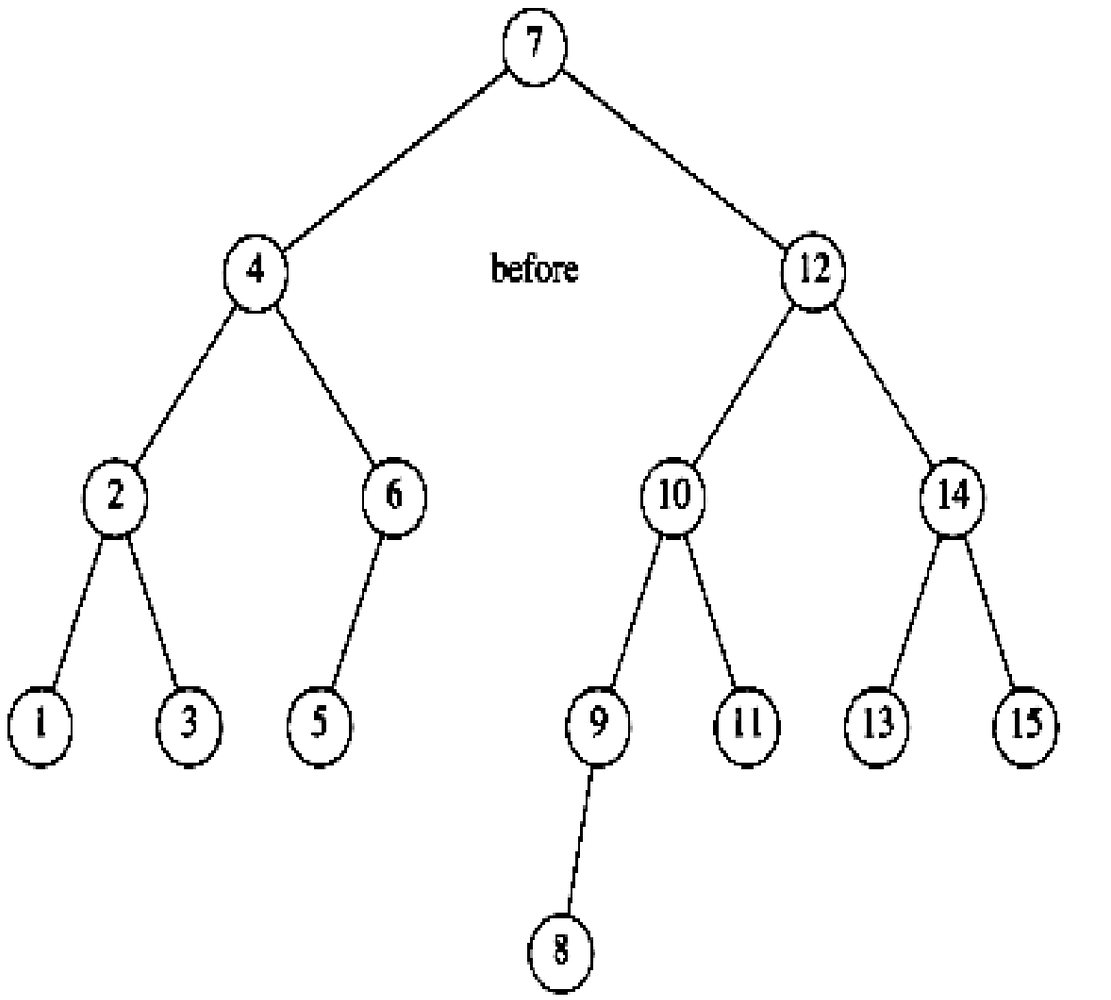
Example-Insert 12, Single Rotation



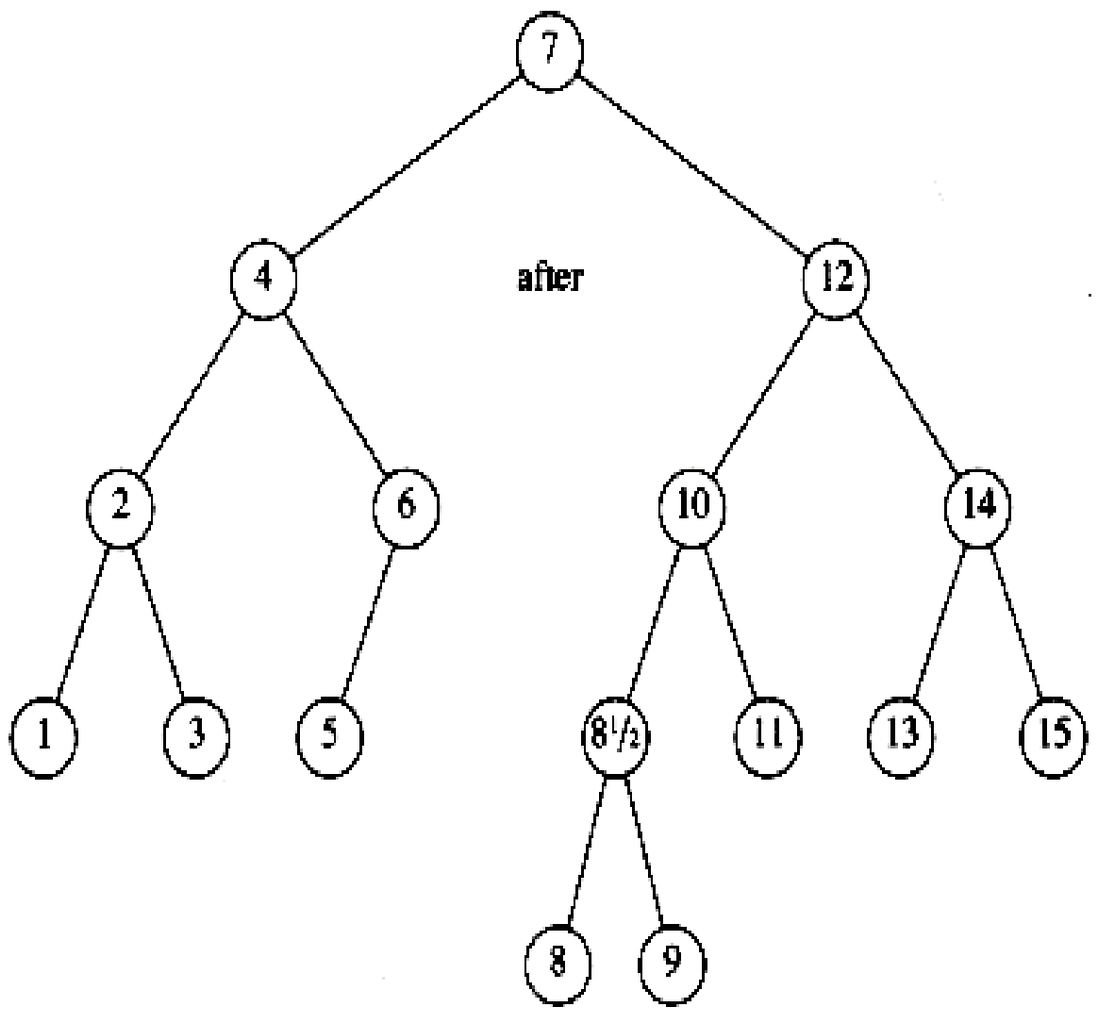
Example-Insert 11, Single Rotation



Example-Insert 81/2, Double Rotation



before



after

Notes

- To insert a new node with key x into an AVL tree T, we recursively insert x into the appropriate subtree of T (let us call this T_{lr}).
- If the height of T_{lr} does not change, then we are done.
- Otherwise, if a height imbalance appears in T, we do the appropriate single or double rotation depending on x and the keys in T and T_{lr}.
- Update the heights, and you are done.

Question

- What if the tree is not a binary tree?
- Can we still use the non-binary tree to perform search?

B-Trees

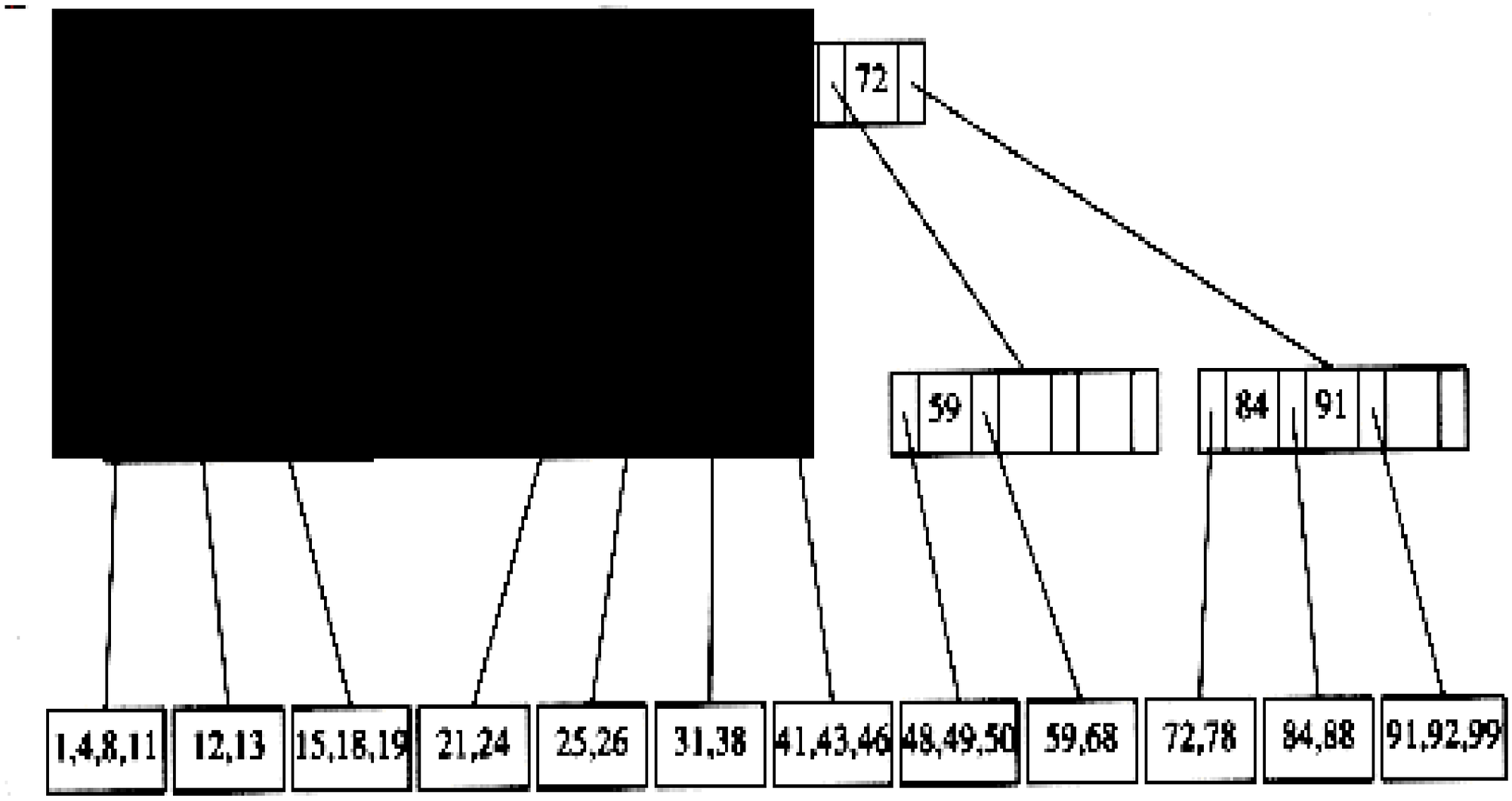
- A B-tree of order m is a tree with the following structural properties:
  - The root is either a leaf or has between 2 and m children.
  - The nonleaf nodes store up to m-1 keys.
  - All nonleaf nodes (except the root) have between $\lceil m/2 \rceil$ and m children.
  - All leaves are at the same depth and have $\lceil l/2 \rceil$ and l elements.
  - All data is stored at the leaves. Contained in each interior node are pointers $P_1, P_2, \ldots, P_m$ to the children, and values $K_1, K_2, \ldots, K_{m-1}$, representing the smallest key found in the subtrees $P_2, P_3, \ldots, P_m$ respectively.

B-Trees

- For every node, all the keys in subtree P_1 are smaller than the keys in subtree P_2, and so on.
- The leaves contain all the actual data
  - keys themselves
  - pointers to records containing the keys

Example

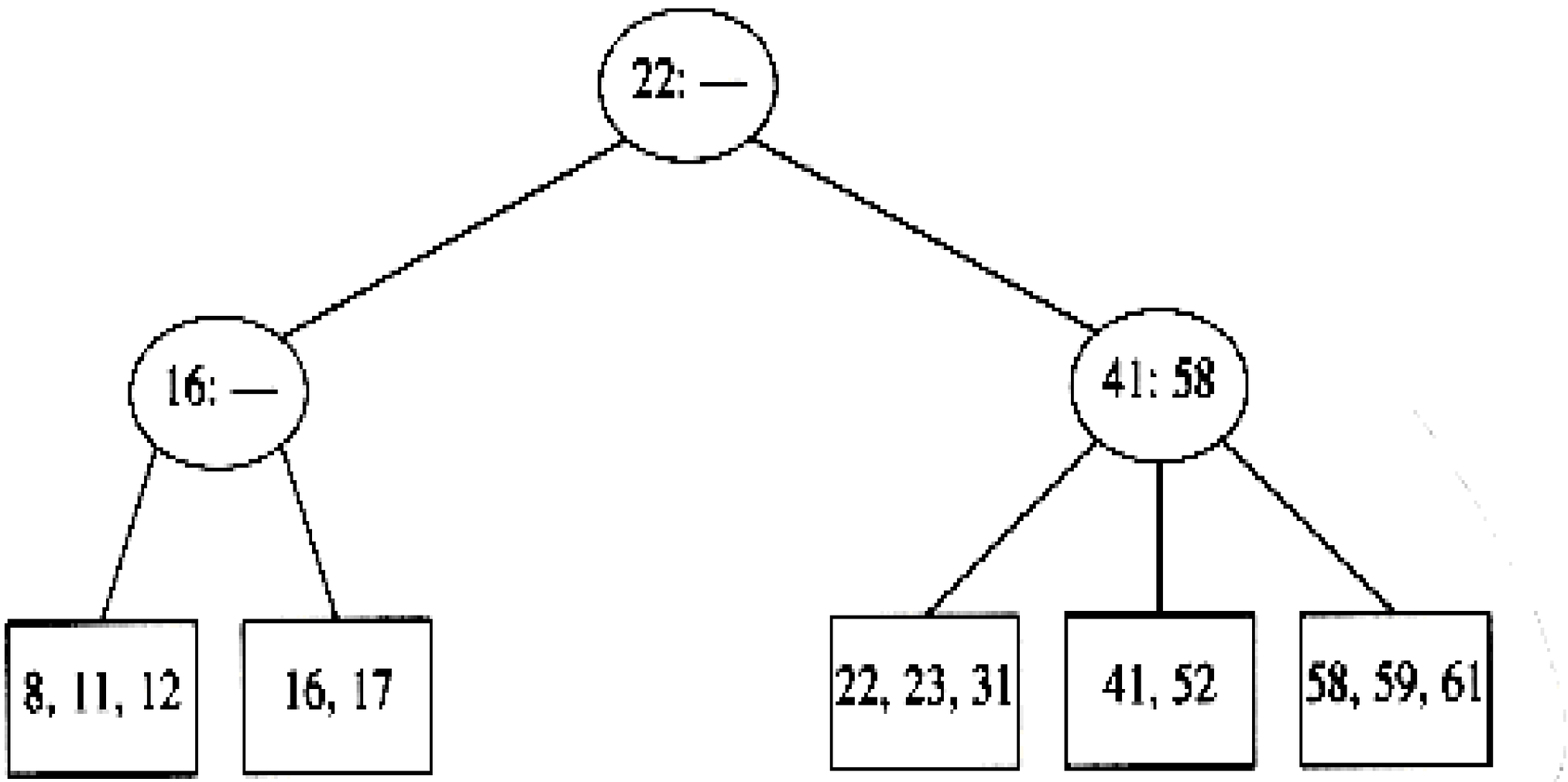
Example (B-tree of order 4)



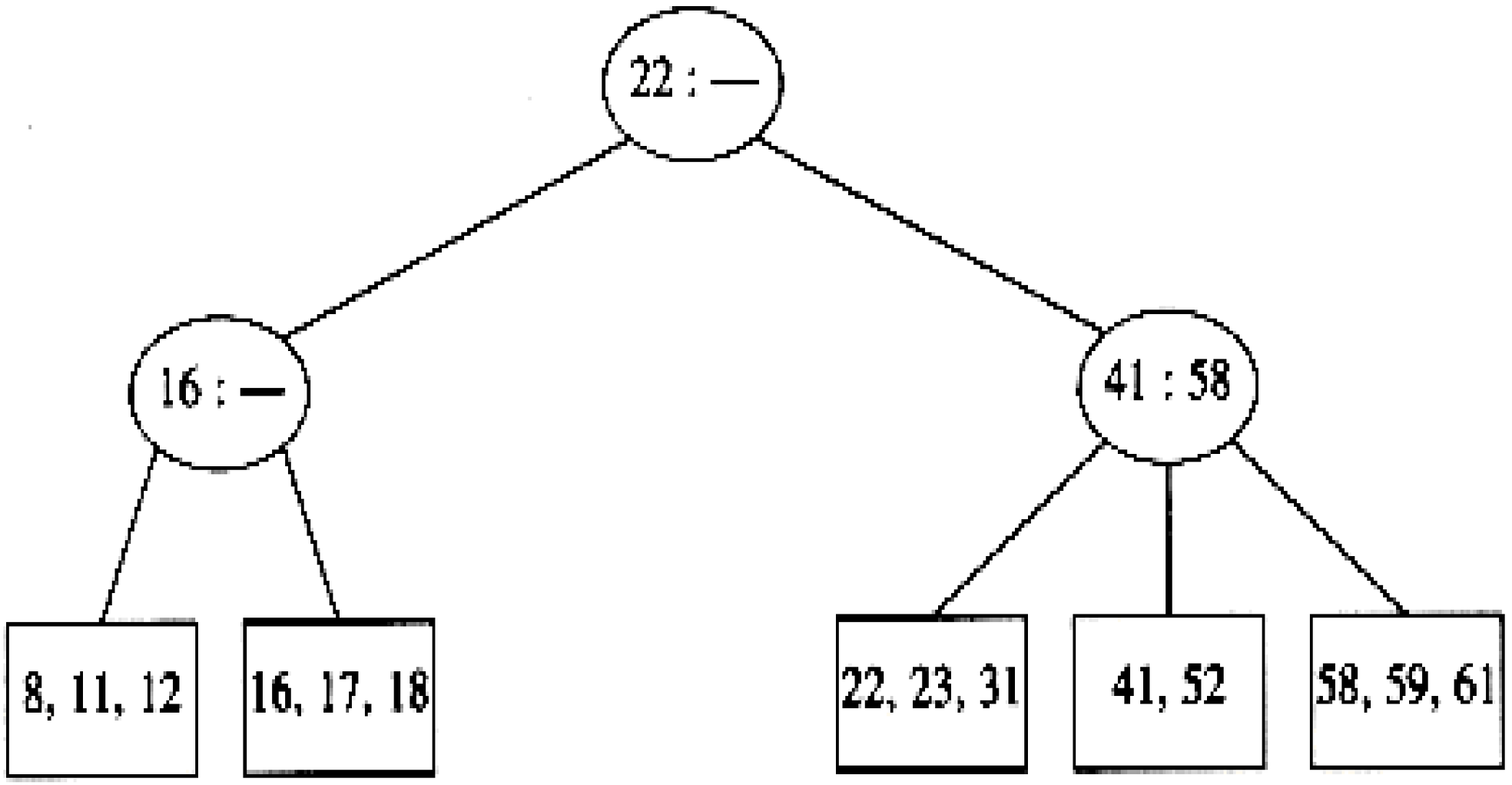
Notes

- A B-tree of order 4 is also known as a 2-3-4 tree.
- A B-tree of order 3 is also known as a 2-3 tree.

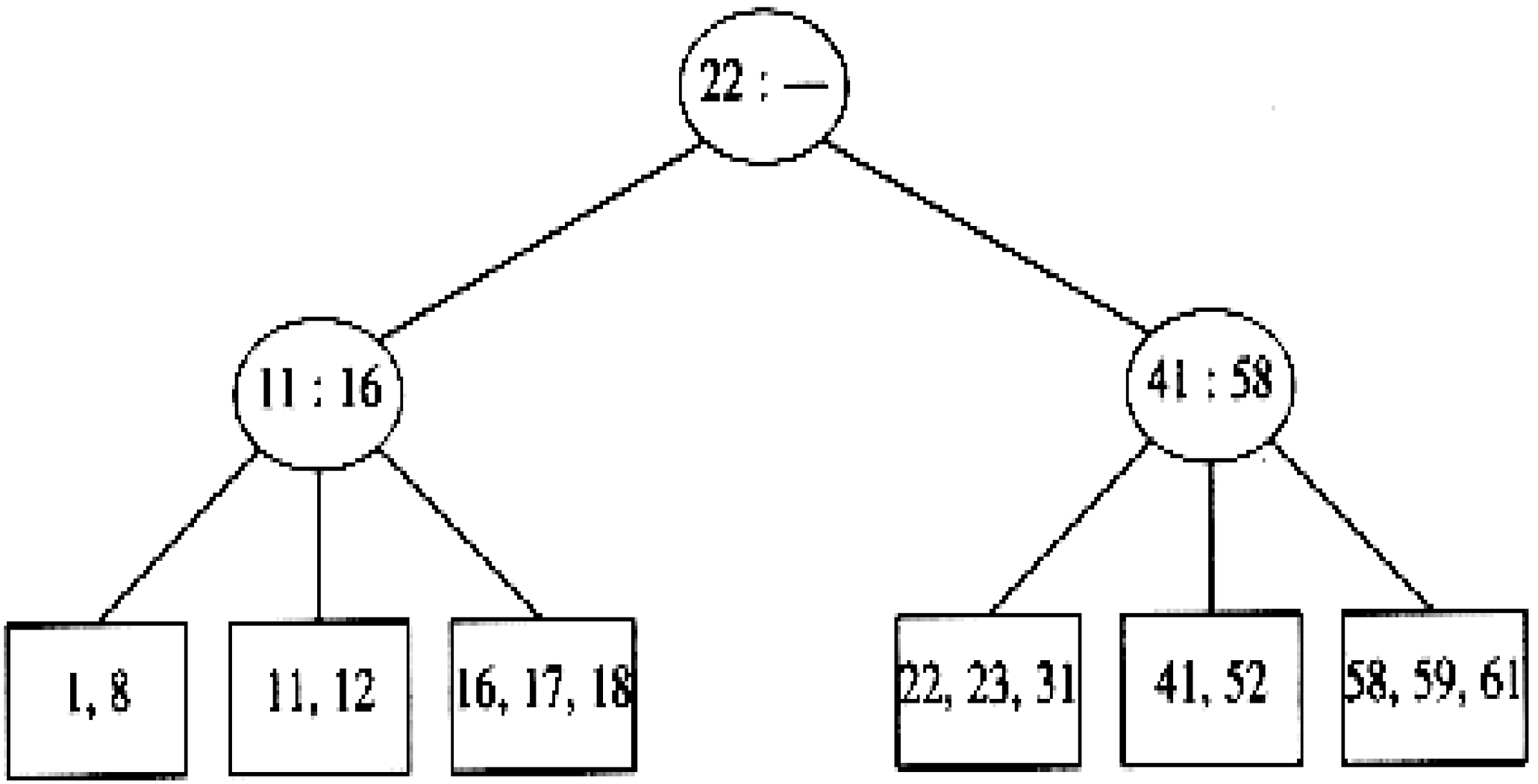
Example of a 2-3 Tree



Example - Insert 18



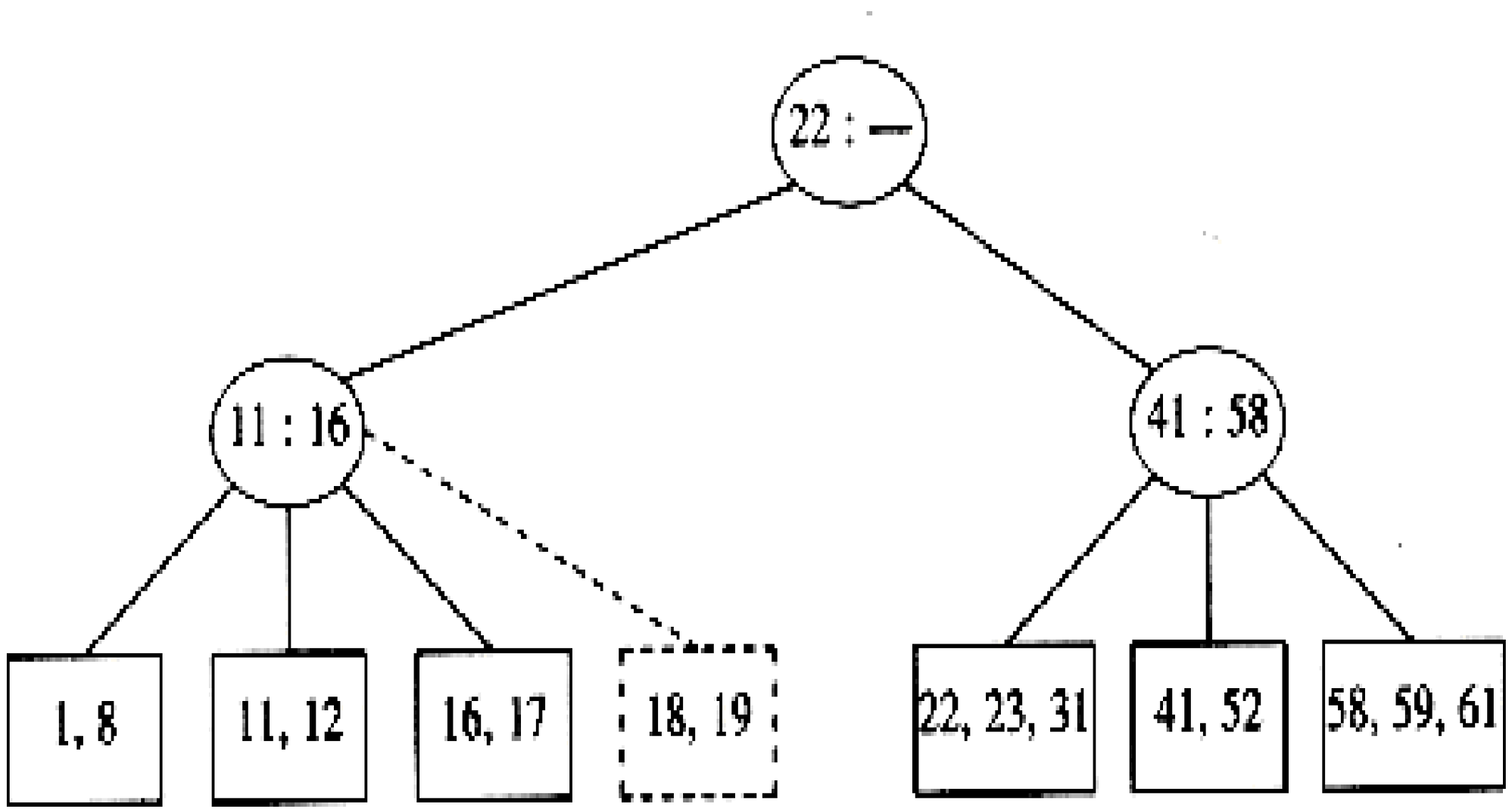
Example - Insert 1



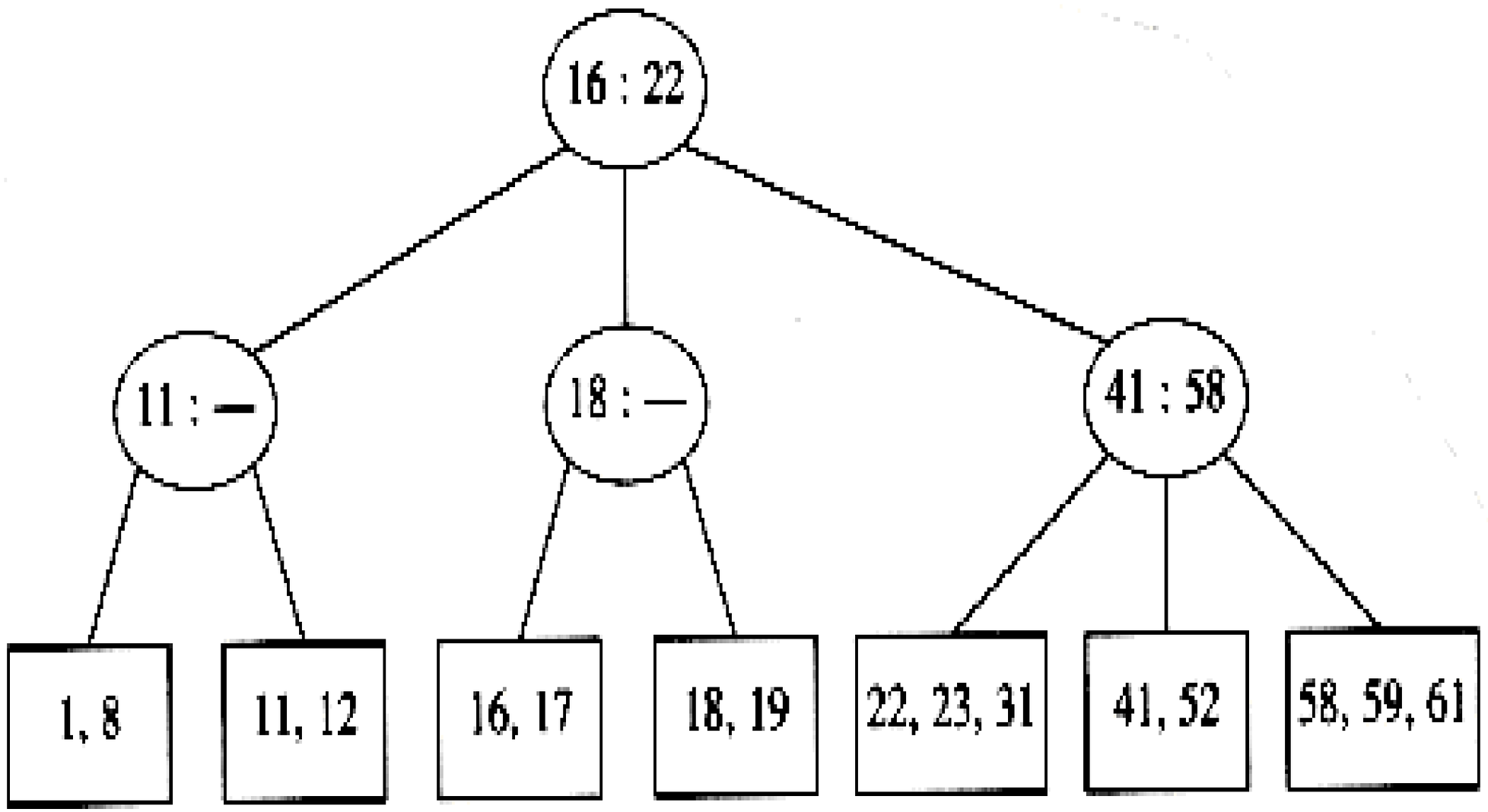
Notes

- Placing 1 into the node will give it a fourth element, which is not allowed.
- It can be solved by making two nodes of two keys each and adjusting the information in the parent.
- However, this does not work all the time.
- For example, let's insert 19.

Example



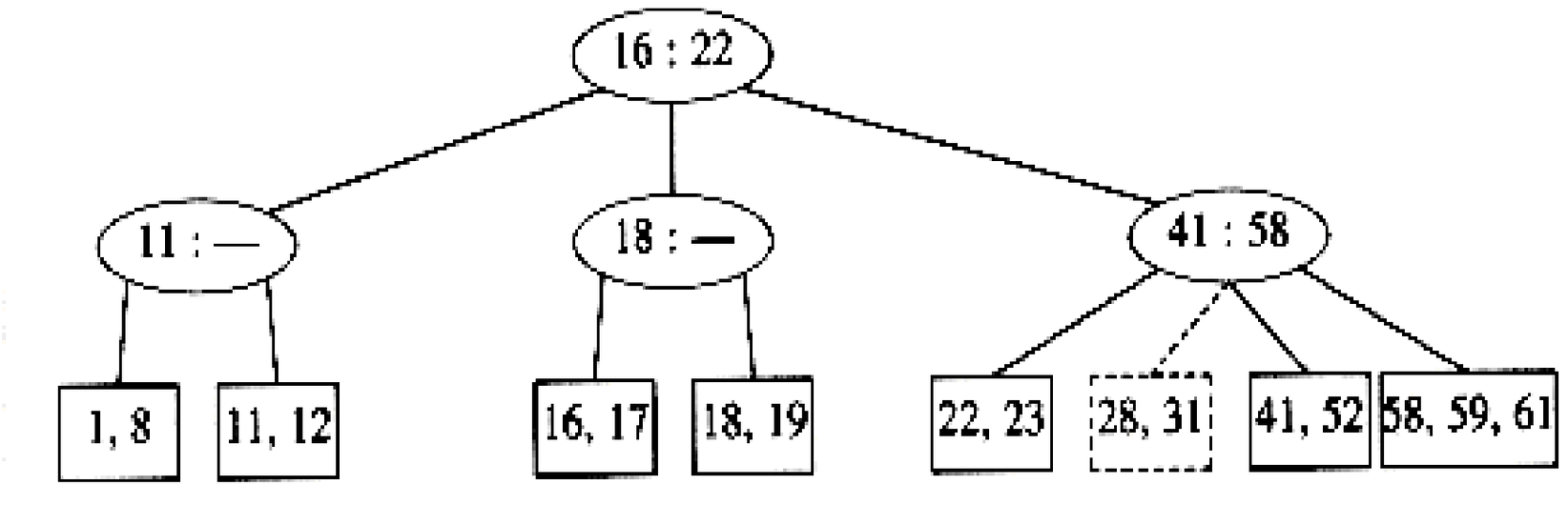
Example

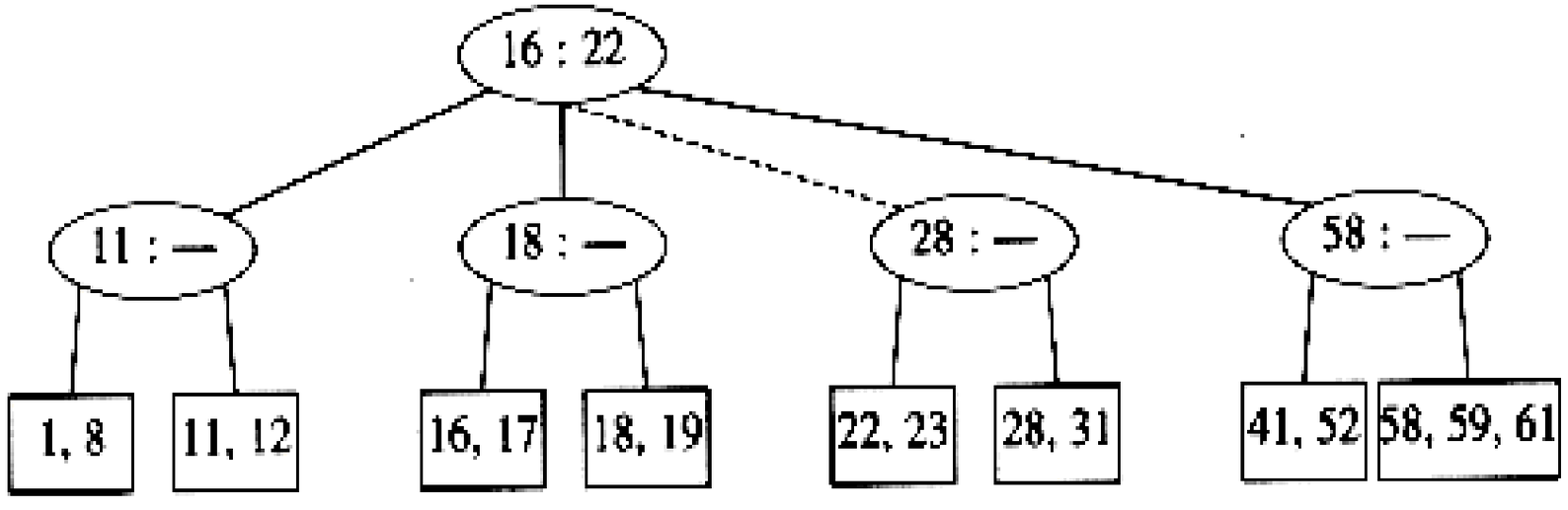


Notes

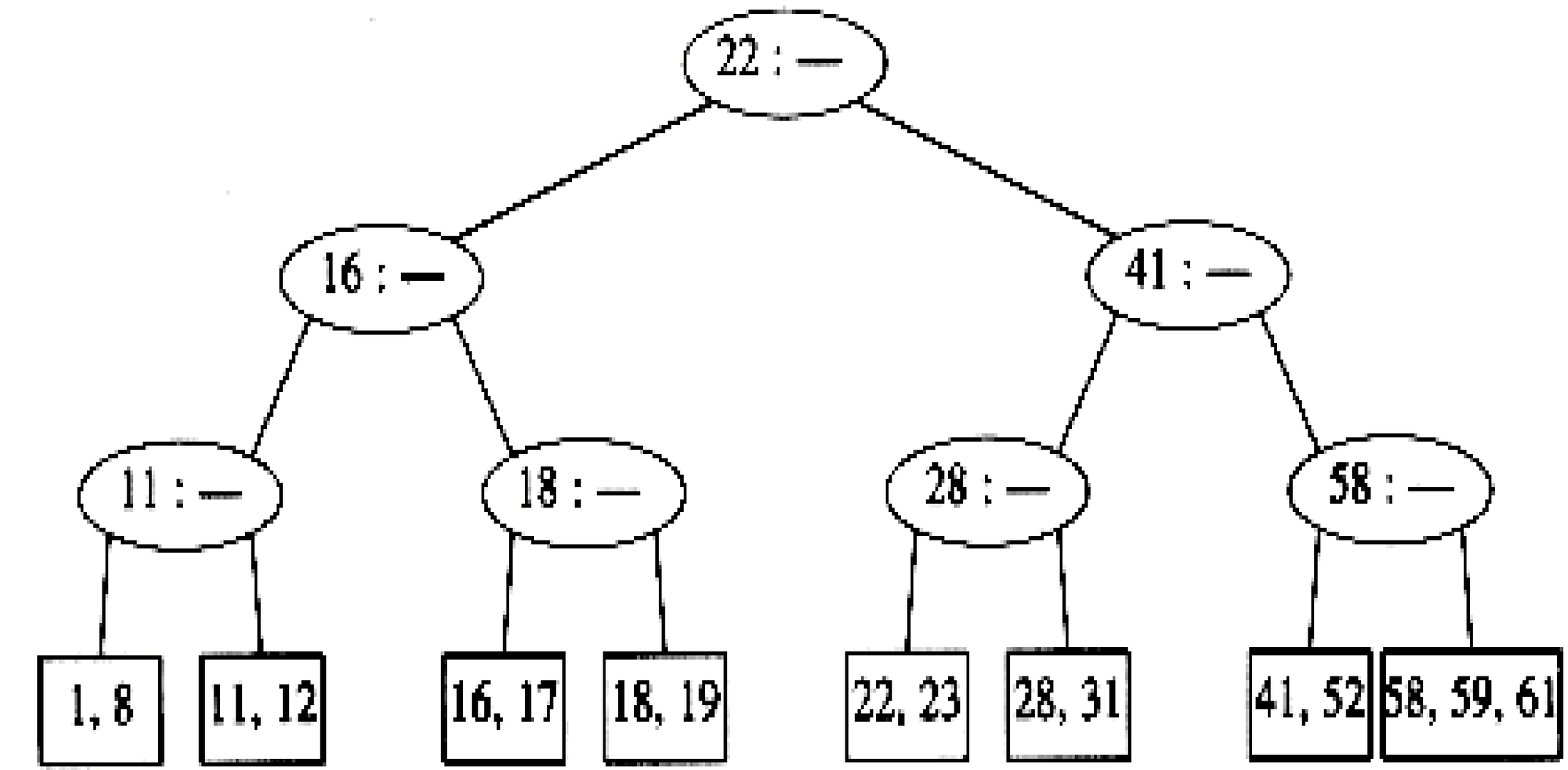
- The solution is to split the full node into two nodes with two children.
- We should continue to split the node upwards to the root until
  - Either get to the root node
  - Or find a node with only two children.

Example - Insert 28





Example - Final Configuration



Notes

- The depth of a B-tree is at most $\lceil \log_{\lceil m/2 \rceil} n \rceil$.
- At each node on the path, we perform O(log m) work to determine which branch to take.
- An Insert or Remove could require O(m) work to fix up all the information at the node.
- The worst-case for Insert and Remove is O(m logm n) = O((m/ logm) log n).
- What is the best m? 3 or 4
- B-trees are used extensively in database systems.

Summary

- Uses of trees in operating systems, compiler design, and searching.
- Expression trees are a small example of a more general structure known as a parse tree, which is a central data structure in compiler design.
- Search trees are of great importance in algorithm design. They support almost all the useful operations, and the logarithmic average cost is very small.

Summary

- AVL trees work by insisting that all nodes' left and right subtrees differ in heights by at most one.
- This ensures that the tree cannot get too deep.
- The operations that do not change the tree, as insertion does, can all use the standard binary search tree code.
- Operations that change the tree must restore the tree.
- We showed how to restore the tree after insertions in O(log n) time.

Summary

- B-trees are balanced m-way trees that are well suited for disks.
- In practice, all the balanced tree schemes is worst than the simple binary search tree, but this is acceptable.